ELEC 458 – EMBEDDED SYSTEMS

PROJECT 3

AZİZ CAN AKKAYA 171024005
BERK SARI 141024068

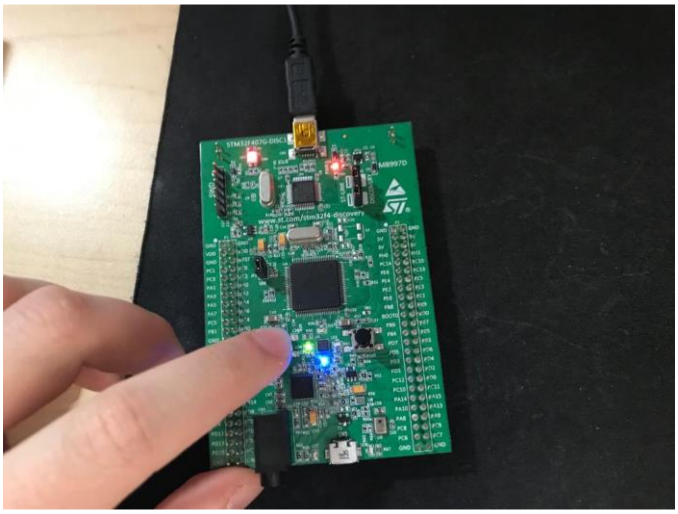
Contents

1. Introduction........2
    1.1 Briefing........2
    1.2 Objective........2
        1.2.1 Project Tasks........3
2. Technical Aspects........3
    2.1 Materials........3
    2.2 Method........3
    2.3 Diagrams........4
        2.3.1 Hardware Block Diagram........4
        2.3.2 Software Flowchart........5
3. Conclusion........8
    3.1 Design Overview........9
    3.2 Discussion........9
4. Appendix........10
5. Sources........15

1 Intorduction

1.1 Briefing

Document Publish Date : 29/06/2020



1.2 Objective

This project is developed on a STM32F407VG Discovery Kit. The ARM Cortex-M4 architecture gained functionality with “main.c” file developed with Embedded C language. The aim of the Project is to understand and gain a brief knowledge about ARM Cortex-M4 and it’s functionality along side with system’s supported features.

1.2.1 Project Tasks

Berk SARI	- SPI Initialization & Usage - I2S Initialization & Usage - I2C Initialization & Usage - TIMER Initialization & usage - Interrupt Initializations & Usages - LIS302DL Accelerometer Initialization - CMSIS PDM to PCM library Initialization & Usage - CS43L22 DAC Initialization & Usage - MEMS MP45DT02 Microphone Usage - Overall Algorithms
Aziz Can AKKAYA	- CMSIS PDM to PCM library Initialization & Usage - SPI Initialization & Usage - Accelerometer Functionality

2 Technical Aspects

2.1 Materials

We decided to use STM32F407VG ARM Cortex-M4 microcontroller. It has fast execution, power efficient and respond with a low possibility of data hazard errors. As for the software we decide to use STM32Cude IDE 1.3.0. The reason behind is rising popularity of the IDE and it has same components as Keil (the most popular IDE currently) but with a different and easier approach to UI.

2.2 Specifications

The system supports 2 different modes; Mode 1 microphone and Mode 2 metronome.

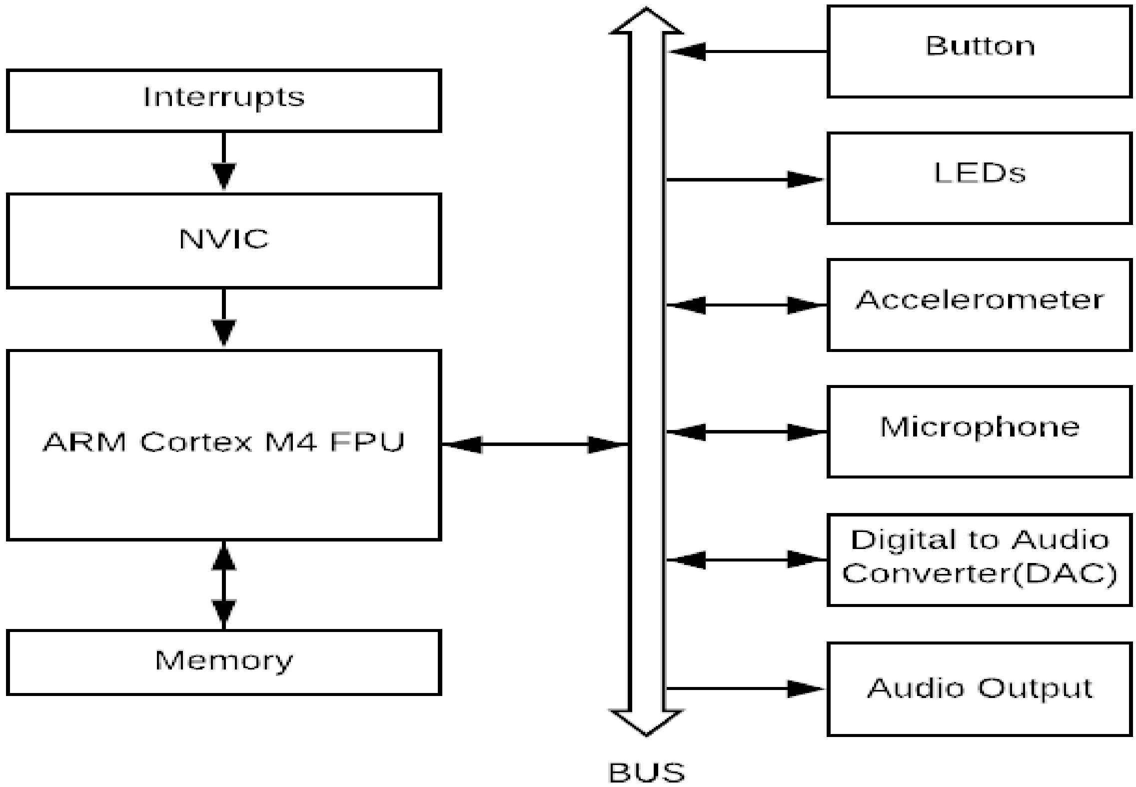
In Mode 1 microphone, system listens a instrument that is played. Collects sound data that produced by instrument. Filters the sound and lights a LED according to wich octave the sound belongs.

In Mode 2 metronome, system generates beep sound starting from 60 Bpm. When board is tilted to left, green LED lit up and beeping bpm decreases When board is tilted to right, red LED lit up and beeping bpm increases. The range of bpm is 30 to 600(these numbers are calculaed).

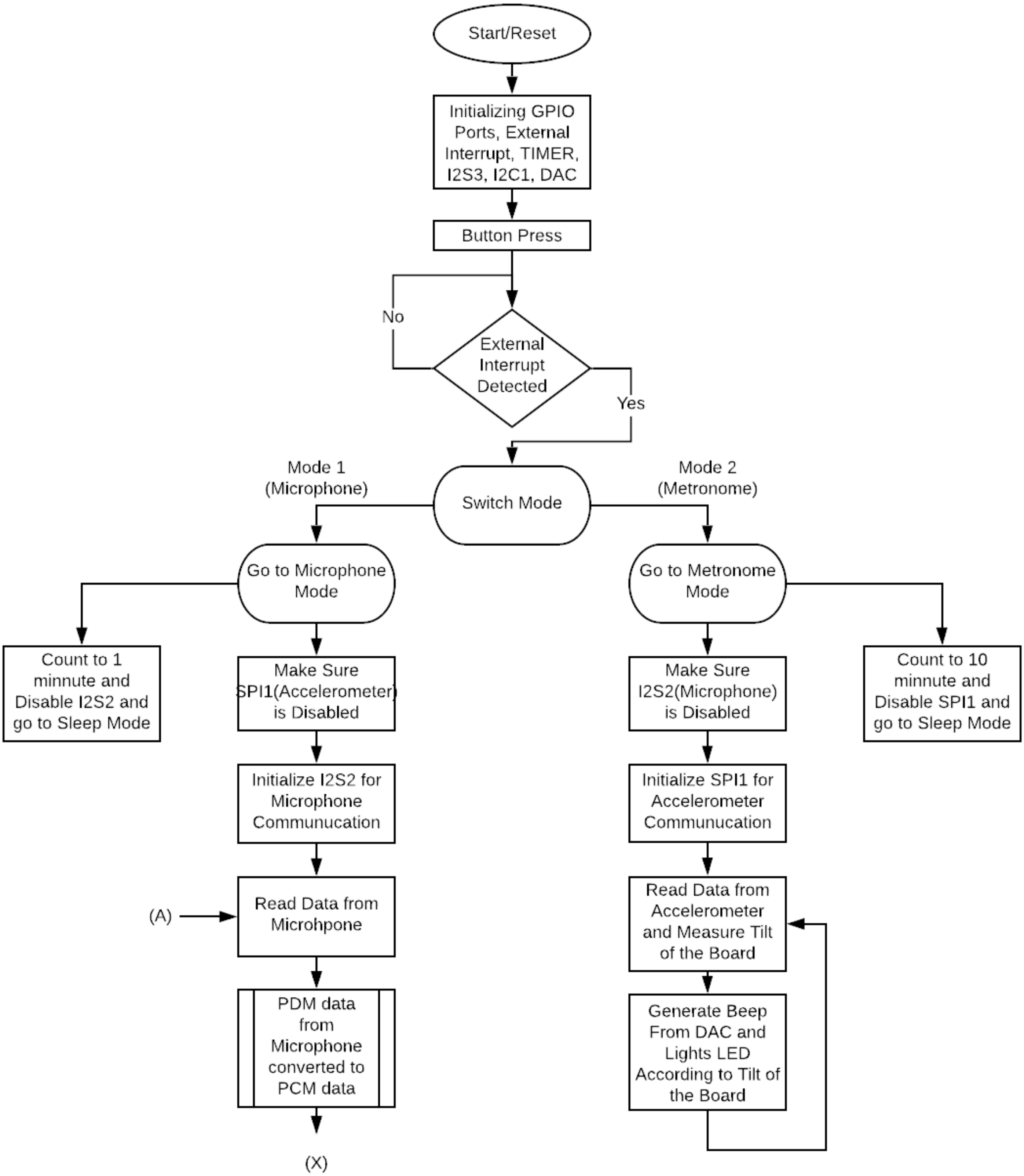
If the system is not used for some time, it closes communucation interfaces and goes to sleep. For mode 1, time before closing the modüle is 1 minute and for mode 2, it’s 10 minnute.

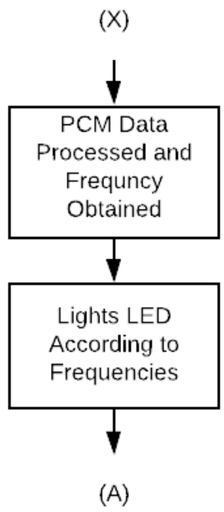
2.3 Diagrams

2.3.1 Hardware Block Diagram



2.3.2 Software Block Diagram





3 Conclusion

3.1 Design Overview

This Project, divided in to 2 pieces; Mode 1 Microphone and Mode 2 Metronome. We searched modules that we are gonna use in this system and collected data. Then communucation protocols decided for each module. We collected data for them, how they work and what are their differences.

The first thing we did was, to cycle modes an external button interrupt created. Then 2 modes created. When system is powered, it waits without doing anything until user gives first input by pressing button, then system directly goes to Mode 1 Microphone. When user gives another button press, system goes to Mode 2 Metronome. Like this, button can cycle between 2 modes.

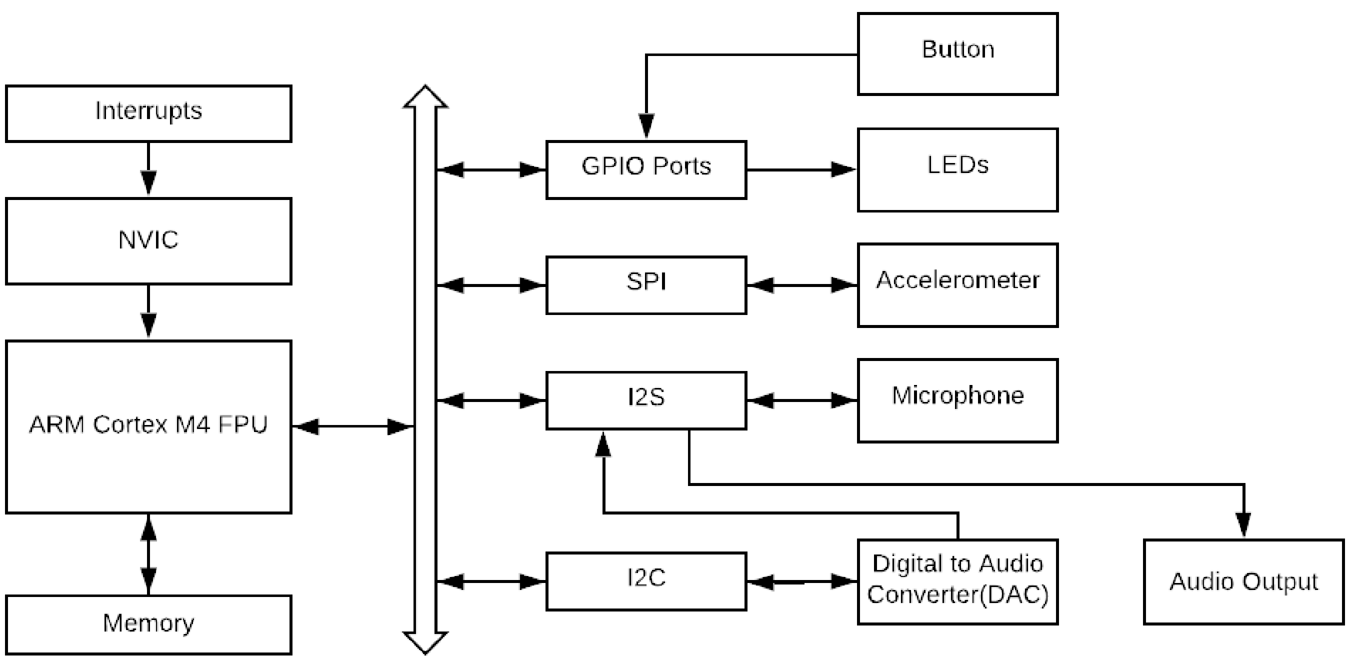
On Mode 1 Microphone, MEMS MP45DT02 microphone used which is located right down side of the STM47F407G Discovery board. To communucate with microphone, we decided to use I2S2 interface in half-dublex mode(to read). I2S2 is initialized as a master, with this way we supplied MEMS MP45DT02 microphone module with clock. I2S2 initialized after entering Mode 1. We will handle the reason in Mode 2. When MEMS MP45DT02 is supplied with clock, it activites itself. Microphone module gives PDM output data according to supplied clock. Tha data produced by microphone module collected via I2S2 interrupt handler which is receiver mode. This data is a PDM and to process, it has to be converted to PCM. For this job, we used CMSIS PDM to PCM converter library. After converting data type, frequency of microphone data is decided. According to the frequency of data, a LED is lit.

On Mode 2 Metronome, there are two different motion sensors used in STM47F407G Discovery boards. The sensor depends on the board's iteration. For our case, it is MEMS LIS302DL accelerometer sensor. After finding which motion sensor we have, we decided to use SPI1 communucation in full-dublex mode(to write and read registers) for it. SPI1 initialized as a master to rule the sensor. For CS43L22 DAC, I2C1 and to receive audio output from audio line SPI3 is used. Before the system enters the Mode 2 Metronome and after the system starts, it initializes I2C1 and SPI3 in main for DAC and its audio output but SPI1 for accelerometer can’t be initialized in there. Because I2S2 which is used in Mode 1, uses same alternate functions for pins as SPI1 and they can’t be used together at the same time. So, SPI1 is initialized after entering Mode 2. Then a flag is set, so gyroscope function which will analyze accelerometers data and light up LEDs according to tilt of the board can work now. Whenever the system goes to Mode 1, the flag is lifted an this gyroscope function no longer works.

As for the bpms, we created two different formulas for each tilt side. We used counters to determine bpm. A sound signal is sent to DAC, counter stars to start counting and when it finishes to counting, a turn off the sound signal is sent to DAC then counter stars again from its reference value. With this way, we can rule bpm with counter. With tilt data(how much board is tilted) we alter the counter number according to our needs. When the board is slightly tilted to right side, minimum tilt value ocurs and bpm number becomes slightly over the 60. But when the board tilted 90 degrees to right side, it maximizes the tilt value and bpm reaches 600. Left tilt uses same logic and it alters bpm between 60 to 30.

As for sleep mode, we used TIMER to count time. TIMER stars to count right after system starts. We use a variable inside a TIMER that each time interrupt hits, that variable increases by 1. As we now the how frequent interrupt hits we can calculate time by looking what is the value of that counter variable. In Mode 1, timer resets it’s counter value upon entry and when 1 minute passes, I2S2 interface is closed and the system sleeps. We discussed Mode 1’s sleep mode in discussion part more detailed. In Mode 2, timer reset it’s counter value again upon entry. While using this mode, when the system is used/tilted it constantly reset TIMER’s counter value and raises a flag that prevents conditions to enter sleep mode. If the system is not used for 10 minutes in this mode, it closes SPI1 interface and the system sleeps.

All the communucation interfaces and their connections can be seen in the Picture below.



3.2 Discussion

While working on this Project, we learned and improved on;

- How to use interrupts and it’s handlers.
- Reading datasheet/manuel and doing this effectively.
- Adding and using predefined function library.
- Crucial usage of breakpoints and expressions.
- Improved our knowledge on C and Embedded C programming.
- What are I2C, I2S, SPI means how to use them and their differences.
- Looking from board schematics.
- Better understanding of PDM and PCM.

While working on this project, because of our inexperience and lack of information on communucation protocols, we had hard times while using them.

In mode 1, the part that after collecting data from microphone and converting it to PCM which arm cortex can process, we should process and filter it to look it’s frequency. I couldn’t be able to obtain frequency data because we weren’t successfull at understanding , how CMSIS’s PCM to PDM converter function works. Whenever we call this function, the system goes to infinite loop and we can’t figure out what causes this. Eventough we looked the code step by step, everything works fine including this function initializations until last line. Because of this obsticle, we were stuck at this step in this mode and couldn’t be able to process microphone data.

We had to put static timer for sleep activation for mode 1 because of we don’t have any processable data for this mode. We know that if the line is actively used, the data we receive will be different then when it’s not used. So, we won’t be able to understand the line is actively used or not. The timer starts right after it enters this mode and closes communucation interface after 1 minute.

4 Appendix

```c
/*
 * ELEC 458
 * PROJECT 3
 *
 * Berk Sarı             141024068
 * Aziz can AKKAYA 171024005
 */

#include "stm32f4xx.h"
#include "system_stm32f4xx.h"
#include "cs43l22.h"
#include "lis302dl.h"
#include "pdm2pcm_glo.h"

// magic volume function from st lib
#define VOLUME_CONVERT(Volume) (((Volume) > 100)? 255:((uint8_t)(((Volume) * 255)
/ 100)))

/*************************************************
* variables
*************************************************/
volatile uint8_t DeviceAddr = CS43L22_ADDRESS;
volatile uint8_t Button_pos = 1;
volatile uint16_t spi_rxdata;
volatile uint16_t spi_data;
volatile uint16_t spi_regaddr;

int16_t buffer1_i2s2[48];
int16_t buffer2_i2s2[48];
int16_t pcm_buffer;
volatile uint32_t mic_pcm_data;
volatile uint8_t q = 0;
volatile int i = 0;
int flag_pdm = 0;
int flag_mode2 = 0;

volatile int16_t rbuf;
int16_t s = 0;
int flag_tilt = 0;


/*************************************************
* function declarations
*************************************************/
int main(void);
void spi_write(uint8_t reg, uint8_t data);
uint8_t spi_read(uint8_t reg);
void microphone();
void metronome();
void init_TIM();
void init_spi1();
void init_i2s2();
void init_LIS3DSH();
//void pdm_pcm(uint16_t *pdm_buffer);
```

```
////////////////////////////////////////////////////////////////////////////////////

static inline void __i2c_start() {
    I2C1->CR1 |= I2C_CR1_START;
    while(!(I2C1->SR1 & I2C_SR1_SB));
}

////////////////////////////////////////////////////////////////////////////////////

static inline void __i2c_stop() {
    I2C1->CR1 |= I2C_CR1_STOP;
    while(!(I2C1->SR2 & I2C_SR2_BUSY));
}

////////////////////////////////////////////////////////////////////////////////////

void spi_write(uint8_t reg, uint8_t data)
{
    GPIOE->ODR &= ~(1U << 3); // enable
    // bit 15 is 0 for write for lis302dl
    uint32_t frame = 0;
    frame = data;
    frame |= (uint16_t)(reg << 8);
    // Send data
    SPI1->DR = frame;
    // wait until transmit is done (TXE flag)
    while (!(SPI1->SR & (1 << 1)));
    // wait until rx buf is not empty (RXNE flag)
    while (!(SPI1->SR & (1 << 0)));

    GPIOE->ODR |= (1 << 3); // disable
    (void)SPI1->DR;              // dummy read
}

////////////////////////////////////////////////////////////////////////////////////

uint8_t spi_read(uint8_t reg)
{
    GPIOE->ODR &= ~(1U << 3); // enable
    // bit 15 is 1 for read for lis302dl
    uint16_t frame = 0;
    frame |= (uint16_t)(reg << 8);
    frame |= (1 << 15);   // read bit
    // Send data
    SPI1->DR = frame;
    // wait until tx buf is empty (TXE flag)
    while (!(SPI1->SR & (1 << 1)));
    // wait until rx buf is not empty (RXNE flag)
    while (!(SPI1->SR & (1 << 0)));

    uint8_t b = (uint8_t)SPI1->DR;
    GPIOE->ODR |= (1 << 3); // disable
    return b;
}

////////////////////////////////////////////////////////////////////////////////////
```

```c
void i2c_write(uint8_t regaddr, uint8_t data) {
    // send start condition
    __i2c_start();

    // send chipaddr in write mode
    // wait until address is sent
    I2C1->DR = DeviceAddr;
    while (!(I2C1->SR1 & I2C_SR1_ADDR));
    // dummy read to clear flags
    (void)I2C1->SR2; // clear addr condition

    // send MAP byte with auto increment off
    // wait until byte transfer complete (BTF)
    I2C1->DR = regaddr;
    while (!(I2C1->SR1 & I2C_SR1_BTF));

    // send data
    // wait until byte transfer complete
    I2C1->DR = data;
    while (!(I2C1->SR1 & I2C_SR1_BTF));

    // send stop condition
    __i2c_stop();
}

/////////////////////////////////////////////////////////////////////////////////

uint8_t i2c_read(uint8_t regaddr) {
    uint8_t reg;

    // send start condition
    __i2c_start();

    // send chipaddr in write mode
    // wait until address is sent
    I2C1->DR = DeviceAddr;
    while (!(I2C1->SR1 & I2C_SR1_ADDR));
    // dummy read to clear flags
    (void)I2C1->SR2; // clear addr condition

    // send MAP byte with auto increment off
    // wait until byte transfer complete (BTF)
    I2C1->DR = regaddr;
    while (!(I2C1->SR1 & I2C_SR1_BTF));

    // restart transmission by sending stop & start
    __i2c_stop();
    __i2c_start();

    // send chipaddr in read mode. LSB is 1
    // wait until address is sent
    I2C1->DR = DeviceAddr | 0x01; // read
    while (!(I2C1->SR1 & I2C_SR1_ADDR));
    // dummy read to clear flags
    (void)I2C1->SR2; // clear addr condition

    // wait until receive buffer is not empty
    while (!(I2C1->SR1 & I2C_SR1_RXNE));
    // read content
```

```c
    reg = (uint8_t)I2C1->DR;

    // send stop condition
    __i2c_stop();

    return reg;
}

/////////////////////////////////////////////////////////////////////////////////

void init_EXT_int()       // Externel interrupt initialization for button
{
      RCC->APB2ENR |= (1 << 14);   //for ext interrupt
      SYSCFG->EXTICR[0] |= 0x00000000;
      // Choose either rising edge trigger (RTSR) or falling edge trigger (FTSR)
      EXTI->RTSR |= 0x00001;   // Enable rising edge trigger on EXTI0
      // Mask the used external interrupt numbers.
      EXTI->IMR |= 0x00001;    // Mask EXTI0
      // Set Priority for each interrupt request
      NVIC->IP[EXTI0_IRQn] = 0x10; // Priority level 1
      // enable EXT0 IRQ from NVIC
      NVIC_EnableIRQ(EXTI0_IRQn);
}

/////////////////////////////////////////////////////////////////////////////////

void EXTI0_IRQHandler(void)            // Externel interrupt for button
{
      if (EXTI->PR & (1 << 0)){

          Button_pos ^= 0x1;          // changes button position
          q = 0;                            // resets timer
          for(uint32_t j=0; j<500000; j++);

         switch(Button_pos)
         {
               case 0:
                     q = 0;                   // Reset timer

                     microphone();    // go to mic
                     break;

               case 1:
                     q = 0;                   // Reset timer

                     metronome();             // go to acceletometer
                     break;
         }
          EXTI->PR = (1 << 0);
      }
}

/////////////////////////////////////////////////////////////////////////////////

void microphone()
{
      flag_mode2 = 0;
      SPI1->CR1 &= (0 << 6);    // SPI1 disabled(metronome data transfer stops)
      //Initialize I2S2 -- mic
```

```
	init_i2s2();

	GPIOD->ODR ^= (uint16_t)(1 << 13);	// orange

}

////////////////////////////////////////////////////////////////////////////////

void metronome()
{
	SPI2->I2SCFGR &= (0 << 10); // I2S2 disabled(mic disabled and data transfer 
stops)
	// initialize SPI1
	init_spi1();
	// initialize LIS3DSH metronome
	init_LIS3DSH();

	GPIOD->ODR ^= (uint16_t)(1 << 13);	// orange
	flag_mode2 = 1;
}

////////////////////////////////////////////////////////////////////////////////

void init_spi1()		// LIS3DSH accelerometer communucation
{
	// enable GPIOE clock, bit 4 on AHB1ENR
	RCC->AHB1ENR |= (1 << 4);
	GPIOE->MODER &= 0xFFFFFF3F;	// reset bits 6-7
	GPIOE->MODER |= 0x00000040;	// set bits 6-7 to 0b01 (output)
	GPIOE->ODR |= (1 << 3);

	// SPI1 data pins setup
	RCC->AHB1ENR |= (1 << 0);	// Enable GPIOA clock
	RCC->APB2ENR |= (1 << 12);		// Enable SPI1
	// PA7 MOSI
	GPIOA->MODER		|= (2 << 14);	// Pin altenate function mode
	GPIOA->OSPEEDR		|= (3 << 14);	// Pin very high speed mode
	GPIOA->AFR[0]		|= (5 << 28);	// Manage alternate function
	// PA6 MISO
	GPIOA->MODER |= (2 << 12);	// Pin altenate function mode
	GPIOA->OSPEEDR		|= (3 << 12);	// Pin very high speed mode
	GPIOA->AFR[0]		|= (5 << 24);	// Manage alternate function
	// PA5 SCK
	GPIOA->MODER		|= (2 << 10);	// Pin altenate function mode
	GPIOA->OSPEEDR		|= (3 << 10);	// Pin very high speed mode
	GPIOA->AFR[0]		|= (5 << 20);	// Manage alternate function

	SPI1->CR1 |= (4 << 3);	// Baud rate control
	SPI1->CR1 |= (0 << 0);	// Clock phase (CPAL)
	SPI1->CR1 |= (0 << 1);	// Clock polarity (CPOL); @idle high SCK
	SPI1->CR1 |= (1 << 11);	// Data frame length; 0-> 8bit, 1-> 16bit
//	SPI1->CR1 |= (1 << 7);	// Frame format; 0-> MSB first, 1-> LSB first
	SPI1->CR1 |= (1 << 9);	// SSM; software NSS enabled
    SPI1->CR1 |= (1 << 8);	// SSI
	SPI1->CR1 |= (1 << 2);	// Master config; 0-> slave, 1-> master

	SPI1->CR1 |= (1 << 6);	// SPI enable
}
```

```
////////////////////////////////////////////////////////////////////////////////

void init_i2c1()                // DAC communucation
{
    // enable I2C clock
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;

    // setup I2C pins
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
    GPIOB->MODER &= ~(3U << 6*2); // PB6
    GPIOB->MODER |=  (2 << 6*2); // AF
    GPIOB->OTYPER |= (1 << 6);   // open-drain
    GPIOB->MODER &= ~(3U << 9*2); // PB9
    GPIOB->MODER |=  (2 << 9*2); // AF
    GPIOB->OTYPER |= (1 << 9);   // open-drain

    // choose AF4 for I2C1 in Alternate Function registers
    GPIOB->AFR[0] |= (4 << 6*4);     // for pin 6
    GPIOB->AFR[1] |= (4 << (9-8)*4); // for pin 9

    // reset and clear reg
    I2C1->CR1 = I2C_CR1_SWRST;
    I2C1->CR1 = 0;

    I2C1->CR2 |= (I2C_CR2_ITERREN); // enable error interrupt

    // fPCLK1 must be at least 2 Mhz for SM mode
    //        must be at least 4 Mhz for FM mode
    //        must be multiple of 10Mhz to reach 400 kHz
    // DAC works at 100 khz (SM mode)
    // For SM Mode:
    //    Thigh = CCR * TPCLK1
    //    Tlow  = CCR * TPCLK1
    // So to generate 100 kHz SCL frequency
    // we need 1/100kz = 10us clock speed
    // Thigh and Tlow needs to be 5us each
    // Let's pick fPCLK1 = 10Mhz, TPCLK1 = 1/10Mhz = 100ns
    // Thigh = CCR * TPCLK1 => 5us = CCR * 100ns
    // CCR = 50
    I2C1->CR2 |= (10 << 0); // 10Mhz periph clock
    I2C1->CCR |= (50 << 0);
    // Maximum rise time.
    // Calculation is (maximum_rise_time / fPCLK1) + 1
    // In SM mode maximum allowed SCL rise time is 1000ns
    // For TPCLK1 = 100ns => (1000ns / 100ns) + 1= 10 + 1 = 11
    I2C1->TRISE |= (11 << 0); // program TRISE to 11 for 100khz
    // set own address to 00 - not really used in master mode
    I2C1->OAR1 |= (0x00 << 1);
    I2C1->OAR1 |= (1 << 14); // bit 14 should be kept at 1 according to the
datasheet

    // enable error interrupt from NVIC
    NVIC_SetPriority(I2C1_ER_IRQn, 1);
    NVIC_EnableIRQ(I2C1_ER_IRQn);

    I2C1->CR1 |= I2C_CR1_PE; // enable i2c
}

////////////////////////////////////////////////////////////////////////////////
```

```
void I2C1_ER_IRQHandler(){              // Interrupt for I2C-1 error
    // error handler
    GPIOD->ODR |= (1 << 15); // blue LED
}

/////////////////////////////////////////////////////////////////////////////////

void init_i2s2() {         // Microphone

    // Setup pins PC6 - MCLK, PB10 - SCK, PC3 - SD, PB12 - WS
    RCC->AHB1ENR |= ((1 << 2) | (1 << 1)); // enable GPIOC and GPIOB clocks
    RCC->APB1ENR |= (1 << 14); // enable SPI2 clock
    // PC6 alternate function mode MCLK
    GPIOC->MODER   |= (2 << 12);       // Pin altenate function mode
    GPIOC->OSPEEDR |= (3 << 12);       // Pin very high speed mode
    GPIOC->AFR[0]  |= (5 << 24);       // Manage alternate function
    // PB10 alternate function mode SCL
    GPIOB->MODER   |= (2 << 20);       // Pin altenate function mode
    GPIOB->OSPEEDR |= (3 << 20);       // Pin very high speed mode
    GPIOB->AFR[1]  |= (5 << 8);        // Manage alternate function
    // PC3 alternate function mode SD
    GPIOC->MODER   |= (2 << 6);        // Pin altenate function mode
    GPIOC->OSPEEDR |= (3 << 6);        // Pin very high speed mode
    GPIOC->AFR[0]  |= (5 << 12);       // Manage alternate function
    // PB12 alternate function mode WS
    GPIOB->MODER   |= (2 << 24);       // Pin altenate function mode
    GPIOB->OSPEEDR |= (3 << 24);       // Pin very high speed mode
    GPIOB->AFR[1]  |= (5 << 16);       // Manage alternate function

    // enable PLL I2S for 48khz Fs (768k bit rate)
    RCC->PLLI2SCFGR |= (258 << 6); // N value = 258
    RCC->PLLI2SCFGR |= (3 << 28); // R value = 3
    RCC->CR |= (1 << 26); // enable PLLI2SON
    while(!(RCC->CR & (1 << 27))); // wait until PLLI2SRDY

    // Configure I2S
    SPI2->I2SCFGR = 0; // reset registers
    SPI2->I2SPR   = 0; // reset registers

    SPI2->I2SPR |= (3 << 0); // Linear prescaler (I2SDIV)
    SPI2->I2SPR |= (1 << 8); // Odd factor for the prescaler (I2SODD)
    SPI2->I2SPR |= (1 << 9); // Master clock output enable

    SPI2->I2SCFGR |= (1 << 11); // I2S mode is selected
    SPI2->I2SCFGR |= (0 << 4);  // I2S standard select, 00 Philips standard, 11
PCM standard
    SPI2->I2SCFGR |= (0 << 1);  // I2S data length 16bit
    SPI2->I2SCFGR |= (0 << 0);  // Channel length, 0 - 16bit, 1 - 32bit
    SPI2->I2SCFGR |= (3 << 8);  // I2S config mode, 11 - Master receive
    SPI2->I2SCFGR |= (1 << 3);  // Steady state clock polarity, 0 - low, 1 - high

    //I2S interrupt enable
    SPI2->CR2 |= (1 << 6);      // Enable interrupt
//  SPI2->CR2 |= (1 << 5);      // Enable error interrupt (no error int. handler?)
    NVIC_EnableIRQ(SPI2_IRQn);

    SPI2->I2SCFGR |= (1 << 10); // I2S enabled(mic enabled and starts data
transfer)
```

```
}

//////////////////////////////////////////////////////////////////////////////

void SPI2_IRQHandler(void)          // Interrupt for collecting microphone data
{

            buffer1_i2s2[i] = SPI2->DR;           // Mic data loaded to buffer
            i++;
            if(i == 49){
                  for(int p=0; p < 49; p++){
                        buffer2_i2s2[p] = buffer1_i2s2[p];
                  }
                  i=0;
//                flag_pdm = 1;       // Activate PDM to PCM function
            }
}

//////////////////////////////////////////////////////////////////////////////

void init_i2s3() {        // DAC

    // Setup pins PC7 - MCLK, PC10 - SCK, PC12 - SD, PA4 - WS
    RCC->AHB1ENR |= (RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOCEN); // enable GPIOA
and GPIOC clocks
    RCC->APB1ENR |= RCC_APB1ENR_SPI3EN; // enable SPI3 clock
    // PC7 alternate function mode MCLK
    GPIOC->MODER   &= ~(3U << 7*2);
    GPIOC->MODER   |= (2 << 7*2);             // Pin altenate function mode
    GPIOC->OSPEEDR |= (3 << 7*2);             // Pin very high speed mode
    GPIOC->AFR[0]  |= (6 << 7*4);             // Manage alternate function
    // PC10 alternate function mode SCL
    GPIOC->MODER   &= ~(3U << 10*2);
    GPIOC->MODER   |= (2 << 10*2);            // Pin altenate function mode
    GPIOC->OSPEEDR |= (3 << 10*2);            // Pin very high speed mode
    GPIOC->AFR[1]  |= (6 << (10-8)*4);        // Manage alternate function
    // PC12 alternate function mode SD
    GPIOC->MODER   &= ~(3U << 12*2);
    GPIOC->MODER   |= (2 << 12*2);            // Pin altenate function mode
    GPIOC->OSPEEDR |= (3 << 12*2);            // Pin very high speed mode
    GPIOC->AFR[1]  |= (6 << (12-8)*4);        // Manage alternate function
    // PA4 alternate function mode WS
    GPIOA->MODER   &= ~(3U << 4*2);
    GPIOA->MODER   |= (2 << 4*2);             // Pin altenate function mode
    GPIOA->OSPEEDR |= (3 << 4*2);             // Pin very high speed mode
    GPIOA->AFR[0]  |= (6 << 4*4);             // Manage alternate function

    // enable PLL I2S for 48khz Fs
    RCC->PLLI2SCFGR |= (258 << 6); // N value = 258
    RCC->PLLI2SCFGR |= (3 << 28); // R value = 3
    RCC->CR |= (1 << 26); // enable PLLI2SON
    while(!(RCC->CR & (1 << 27))); // wait until PLLI2SRDY

    // Configure I2S
    SPI3->I2SCFGR = 0; // reset registers
    SPI3->I2SPR   = 0; // reset registers

    SPI3->I2SCFGR |= (1 << 11); // I2S mode is selected
    SPI3->I2SCFGR |= (3 << 8);  // I2S config mode, 11 - Master Transmit
```

```
    //SPI2->I2SCFGR |= (0x0 << 7);  // PCM frame sync, 0 - short frame
    //SPI2->I2SCFGR |= (0x0 << 4);  // I2S standard select, 00 Philips standard, 
11 PCM standard
    //SPI3->I2SCFGR |= (1 << 3);  // Steady state clock polarity, 0 - low, 1 - 
high
    //SPI2->I2SCFGR |= (0x0 << 0);  // Channel length, 0 - 16bit, 1 - 32bit

    SPI3->I2SPR |= (1 << 9); // Master clock output enable
    // 48 Khz
    SPI3->I2SPR |= (1 << 8); // Odd factor for the prescaler (I2SODD)
    SPI3->I2SPR |= (3 << 0); // Linear prescaler (I2SDIV)

    SPI3->I2SCFGR |= (1 << 10); // I2S enabled
}

/////////////////////////////////////////////////////////////////////////////////

void init_TIM()
{
      RCC->APB1ENR |= (1 << 0); // TIM2 clock enable
      // Timer clock runs at ABP1 * 2
      //   since ABP1 is set to /4 of fCLK
      //   thus 168M/4 * 2 = 84Mhz
      // set prescaler to 83999
      //   it will increment counter every prescalar cycles
      // fCK_PSC / (PSC[15:0] + 1)
      // 84 Mhz / 8399 + 1 = 10 khz timer clock speed
      TIM2->PSC = 8399;
      // Set the auto-reload value to 10000
      //   which should give 1 second timer interrupts
      TIM2->ARR = 60000; // 6s
      // Update Interrupt Enable
      TIM2->DIER |= (1 << 0);
      // enable TIM2 IRQ from NVIC
      NVIC_EnableIRQ(TIM2_IRQn);
      // Enable Timer 2 module (CEN, bit0)
      TIM2->CR1 |= (1 << 0);
}

/////////////////////////////////////////////////////////////////////////////////

void TIM2_IRQHandler(void)              //6sec between interrupts
{
    TIM2->SR = (uint16_t)(~(1 << 0));

    if((Button_pos == 0) & (q == 10)){       // 1min
      q = 0;
      flag_pdm = 0;                          // deactivate PDM to PCM filter
      SPI2->I2SCFGR &= (0 << 10);                         // I2S2 disabled(mic 
disabled and data transfer stops)
    }

    else if((Button_pos == 1) & (q == 100) & (flag_tilt == 0)){        //10min
      q = 0;
      flag_mode2 = 0;                        // deactivate gyroscope function
      i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0x00); // turn off the beep(silence)
      SPI1->CR1 &= (0 << 6);                                    // SPI1 
disabled(Accelerometer data transfer stops)
    }
```

```
    else{
      q++;
    }
}

////////////////////////////////////////////////////////////////////////////////

void init_cs43l22()              // DAC initialization -- i2c
{
        //*******************************
        // setup reset pin for CS43L22 - GPIOD 4
        //*******************************
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
        GPIOD->MODER &= ~(3U << 4*2);
        GPIOD->MODER |=  (1 << 4*2);
        // activate CS43L22
        GPIOD->ODR   |=  (1 << 4);

        // read Chip ID - first 5 bits of CHIP_ID_ADDR
        uint8_t ret = i2c_read(CS43L22_REG_ID);

        if ((ret >> 3) != CS43L22_CHIP_ID) {
            GPIOD->ODR |= (1 << 13); // orange led on error
        }

    // power off
    i2c_write(CS43L22_REG_POWER_CTL1, CS43L22_PWR_CTRL1_POWER_DOWN);
    // headphones on, speakers off
    i2c_write(CS43L22_REG_POWER_CTL2, 0xAF);
    // auto detect speed MCLK/2
    i2c_write(CS43L22_REG_CLOCKING_CTL, 0x81);
    // slave mode, I2S data format
    i2c_write(CS43L22_REG_INTERFACE_CTL1, 0x04);

    // set volume levels to 50. magic functions from st
    uint8_t convertedvol = VOLUME_CONVERT(50);
    if(convertedvol > 0xE6)
    {
        i2c_write(CS43L22_REG_MASTER_A_VOL, (uint8_t)(convertedvol - 0xE7));
        i2c_write(CS43L22_REG_MASTER_B_VOL, (uint8_t)(convertedvol - 0xE7));
    }
    else
    {
        i2c_write(CS43L22_REG_MASTER_A_VOL, (uint8_t)(convertedvol + 0x19));
        i2c_write(CS43L22_REG_MASTER_B_VOL, (uint8_t)(convertedvol + 0x19));
    }

    // disable the analog soft ramp
    i2c_write(CS43L22_REG_ANALOG_ZC_SR_SET, 0);
    // disable the digital soft ramp
    i2c_write(CS43L22_REG_MISC_CTL, 0x04);
    // disable the limiter attack level
    i2c_write(CS43L22_REG_LIMIT_CTL1, 0);
    // bass and treble levels
    i2c_write(CS43L22_REG_TONE_CTL, 0x0F);
    // pcm volume
    i2c_write(CS43L22_REG_PCMA_VOL, 0x0A);
    i2c_write(CS43L22_REG_PCMB_VOL, 0x0A);
```

```c
    // power on
    i2c_write(CS43L22_REG_POWER_CTL1, CS43L22_PWR_CTRL1_POWER_UP);
    // wait little bit
    for (volatile int i=0; i<500000; i++);
}

////////////////////////////////////////////////////////////////////////////////////

void init_LIS3DSH()       // Accelerometer
{
      // reboot memory
      spi_write(LIS302_REG_CTRL_REG2, 0x40);
      // active mode, +/-2g
      spi_write(LIS302_REG_CTRL_REG1, 0x47);
      // wait
      for(int i=0; i<10000000; i++);
      // read who am i
      rbuf = (int8_t)spi_read(LIS302_REG_WHO_AM_I);
}

////////////////////////////////////////////////////////////////////////////////////

/**************************************************
* main code starts from here
*************************************************/
int main(void)
{
    /* set system clock to 168 Mhz */
      set_sysclk_to_168();

    //*******************************
    // setup LEDs - GPIOD 12,13,14,15
    //*******************************
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
    GPIOD->MODER &= ~(0xFFU << 24);
    GPIOD->MODER |= (0x55 << 24);
    GPIOD->ODR    = 0x0000;

    //Initialize TIMER
    init_TIM();
    // Initialize external interrupt ( Button )
    init_EXT_int();

    // Initialize I2S3 -- audio output
    init_i2s3();
    // Initialize I2C -- dac
    init_i2c1();
    // Initialize cs43l22 -- mic
    init_cs43l22();

    // beep setup
    i2c_write(CS43L22_REG_BEEP_VOL_OFF_TIME, 0x06);       // set beep volume
    uint8_t beep = {0x01};              // beep frequency

//    // Enabling sleep mode
//    __enable_irq();
//    SCB->SCR |= (1 << 1); //Sleep on exit
//    __WFI();
```

```
    while(1)
    {

      if(flag_pdm == 1)                 // PDM to PCM converter
      {
            flag_pdm = 0;
            GPIOD->ODR |= (1 << 15);  // blue led

            //Initialize Pdm to Pcm library
                  RCC->APB1ENR |= (1 << 12);       // Enabled CRC
            CRC->CR |= (1 << 0);       // CRC reset

                  PDM_Filter_Handler_t PDM1_filter_handler;
                  PDM_Filter_Config_t PDM1_filter_config;

                  PDM1_filter_handler.bit_order = PDM_FILTER_BIT_ORDER_LSB;
                  PDM1_filter_handler.endianness = PDM_FILTER_ENDIANNESS_BE;
                  PDM1_filter_handler.high_pass_tap = 2122358088;
                  PDM1_filter_handler.out_ptr_channels = 1;
                  PDM1_filter_handler.in_ptr_channels = 1;
                  PDM_Filter_Init((PDM_Filter_Handler_t
*)(&PDM1_filter_handler));

                  PDM1_filter_config.output_samples_number = 16;
                  PDM1_filter_config.mic_gain = 0;
                  PDM1_filter_config.decimation_factor =
PDM_FILTER_DEC_FACTOR_48;
                  PDM_Filter_setConfig((PDM_Filter_Handler_t
*)&PDM1_filter_handler, &PDM1_filter_config);

                  // Convert Pdm data to Pcm data
                  PDM_Filter(&buffer2_i2s2, &pcm_buffer, &PDM1_filter_handler);
      }

      if(flag_mode2 == 1)               // Read gyroscope data & manage beeping
      {
            // Accelerometer data transferred to r buffer
            rbuf = (int8_t)spi_read(LIS302_REG_OUT_X);

            if(rbuf > 8) {                               // decrease bpm - right
tilt
              GPIOD->ODR &= (uint16_t)~0x1000;
              GPIOD->ODR ^= 0x4000;          // toggle red led

              // Transfer Accelerometer data to s from buffer
              s = rbuf;

              //decreasing waiting time/increasing bpm
              // This function calculates bpm according to tilt
              int k = ((-(s)*(96428.57142857143))+(6771428.571428572));
              // continuous beep mode
              i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0xC0);

              // generate beep
              i2c_write(CS43L22_REG_BEEP_FREQ_ON_TIME, beep);
              for (volatile int j=0; j<k; j++);     // 1sn = 2*6.000.000
```

```
            GPIOD->ODR ^= 0x4000;           // toggle green led

            // turn offthe beep(silence)
            i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0x00);
            for (volatile int j=0; j<k; j++);     // 1sn = 2*6.000.000

            flag_tilt = 1;    // Shows that system is used/tilted or not
            q = 0;            // Resets timer
          }
          else if (rbuf < -8 ) {                    //increase bpm - left tilt
            GPIOD->ODR &= (uint16_t)~0x4000;
            GPIOD->ODR ^= 0x1000;           // toggle red led

            // Transfer Accelerometer data to s from buffer
            s = rbuf;

            //decreasing waiting time/increasing bpm
            // This function calculates bpm according to tilt
            int k = ((-(s)*(107142.85714285714))+(5142857.142857143));

            // continuous beep mode
            i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0xC0);

            // generate beep
            i2c_write(CS43L22_REG_BEEP_FREQ_ON_TIME, beep);
            for (volatile int j=0; j<k; j++);     // 1sn = 2*6.000.000

            GPIOD->ODR ^= 0x1000;           // toggle red led

            // generate beep
            i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0x00);
            for (volatile int j=0; j<k; j++);     // 1sn = 2*6.000.000

            flag_tilt = 1;    // Shows that system is used/tilted or not
            q = 0;            // Resets timer
          }
          else {
            GPIOD->ODR &= (uint16_t)~0x0000;

            // continuous beep mode
            i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0xC0);

            // generate beep
            i2c_write(CS43L22_REG_BEEP_FREQ_ON_TIME, beep);
            for (volatile int j=0; j<6000000; j++);     // 1sn = 2*6.000.000

            // turn offthe beep(silence)
            i2c_write(CS43L22_REG_BEEP_TONE_CFG, 0x00);
            for (volatile int j=0; j<6000000; j++);     // 1sn = 2*6.000.000

            flag_tilt = 0;    // Shows that system is used/tilted or not
          }
      }
    }
    return 0;
```

5 Sources

- RM0090 Reference Manual
- UM1472 User Manual
- CS43L22 Low Power, Stereo with Headphone & Speaker Amps Datasheet
- LIS3DSH MEMS Digital Output Motion Sensor Datasheet
- AN3393 LIS3DSH Application Note
- MP45DT02 MEMS Audio Sensor Omnidirectional Digital Microphone Datasheet
- UM2372 PDM2PCM Software Library User Manual
- AN3998 PDM Audio Software Decoding Application Note
- AN5027 Interfacing PDM Digital Microphones Using STM32 MCUs Application Note