GEBZE TECHNICAL UNIVERSITY

ELECTRONIC ENGINEERING

ELEC334 – MICROPROCESSORS

Project 2

HAZIRLAYANLAR
1801022035 – Ruveyda Dilara Günal
1801022071 – Alperen Arslan
1901022255 – Emirhan Köse

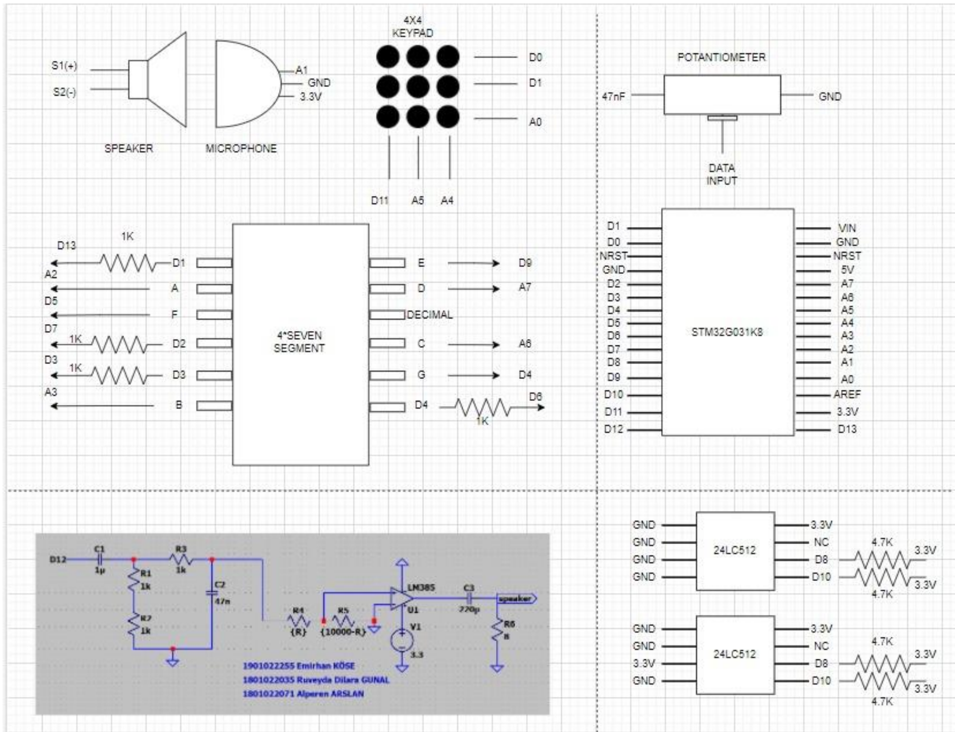

Figure 1: Block Diagram

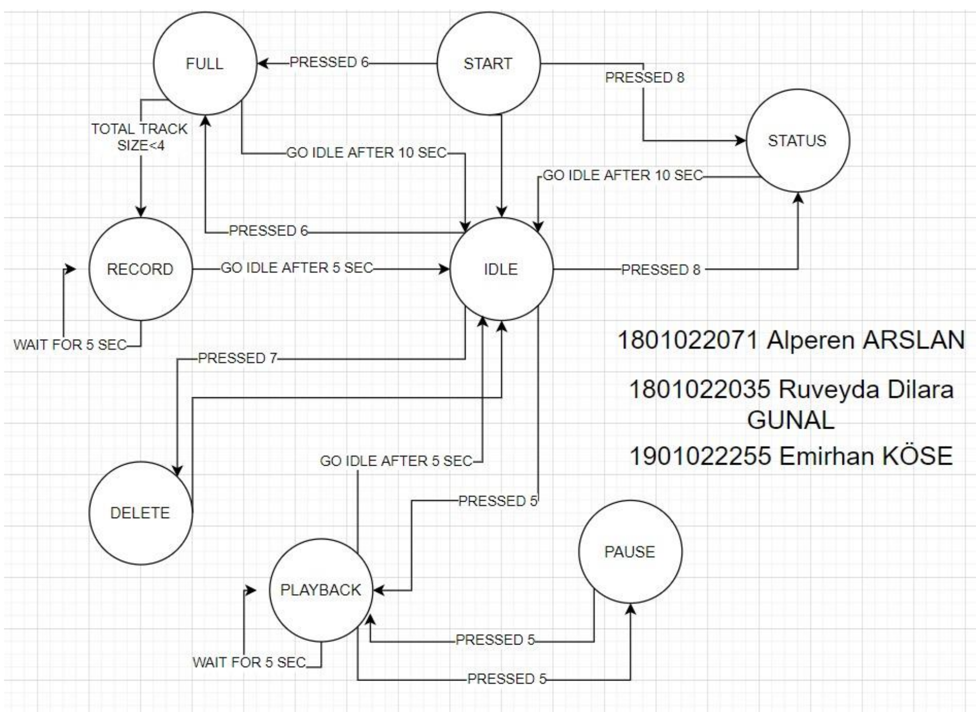

Figure 2: State Machine Diagram

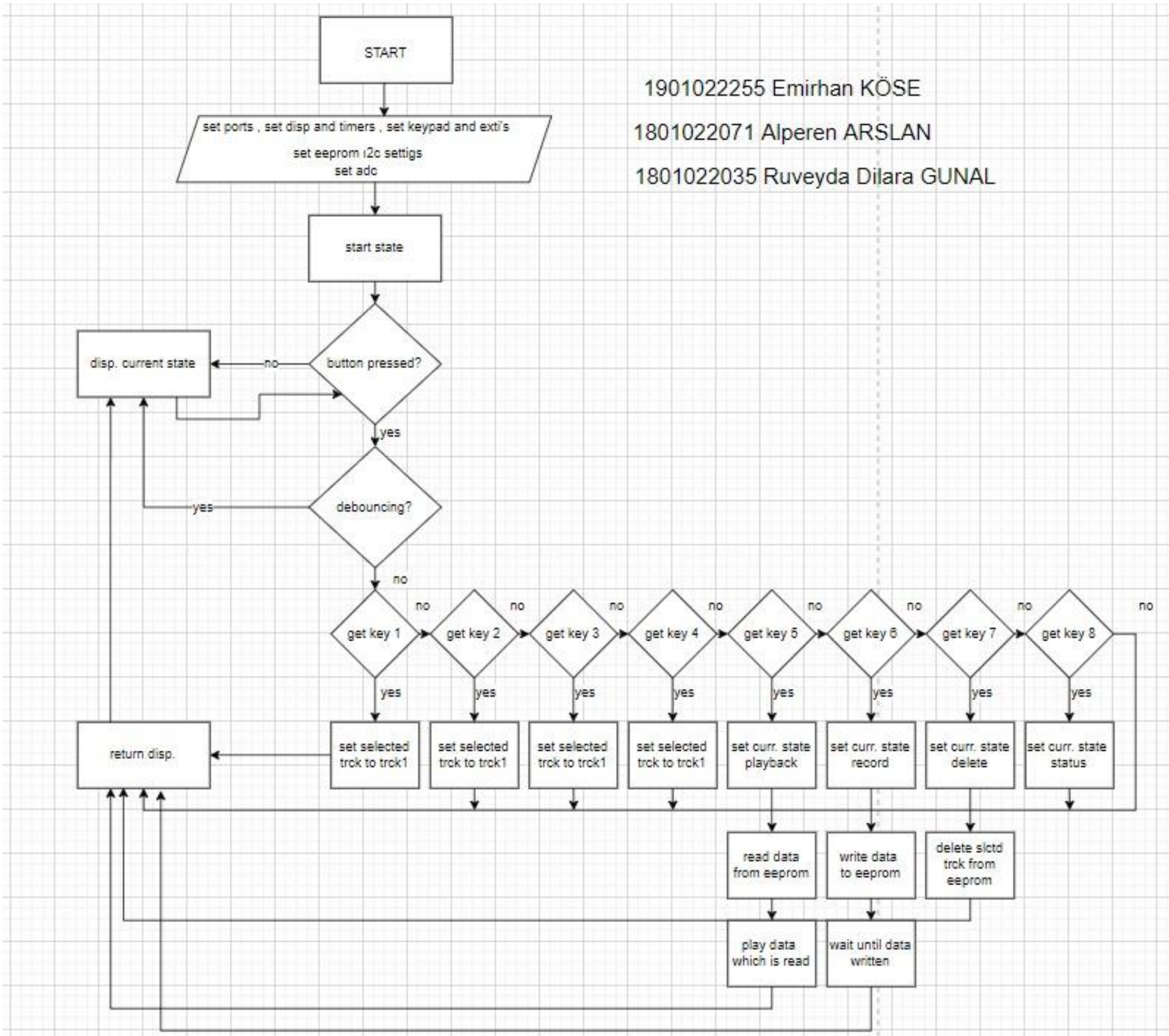

Figure 3: Flowchart

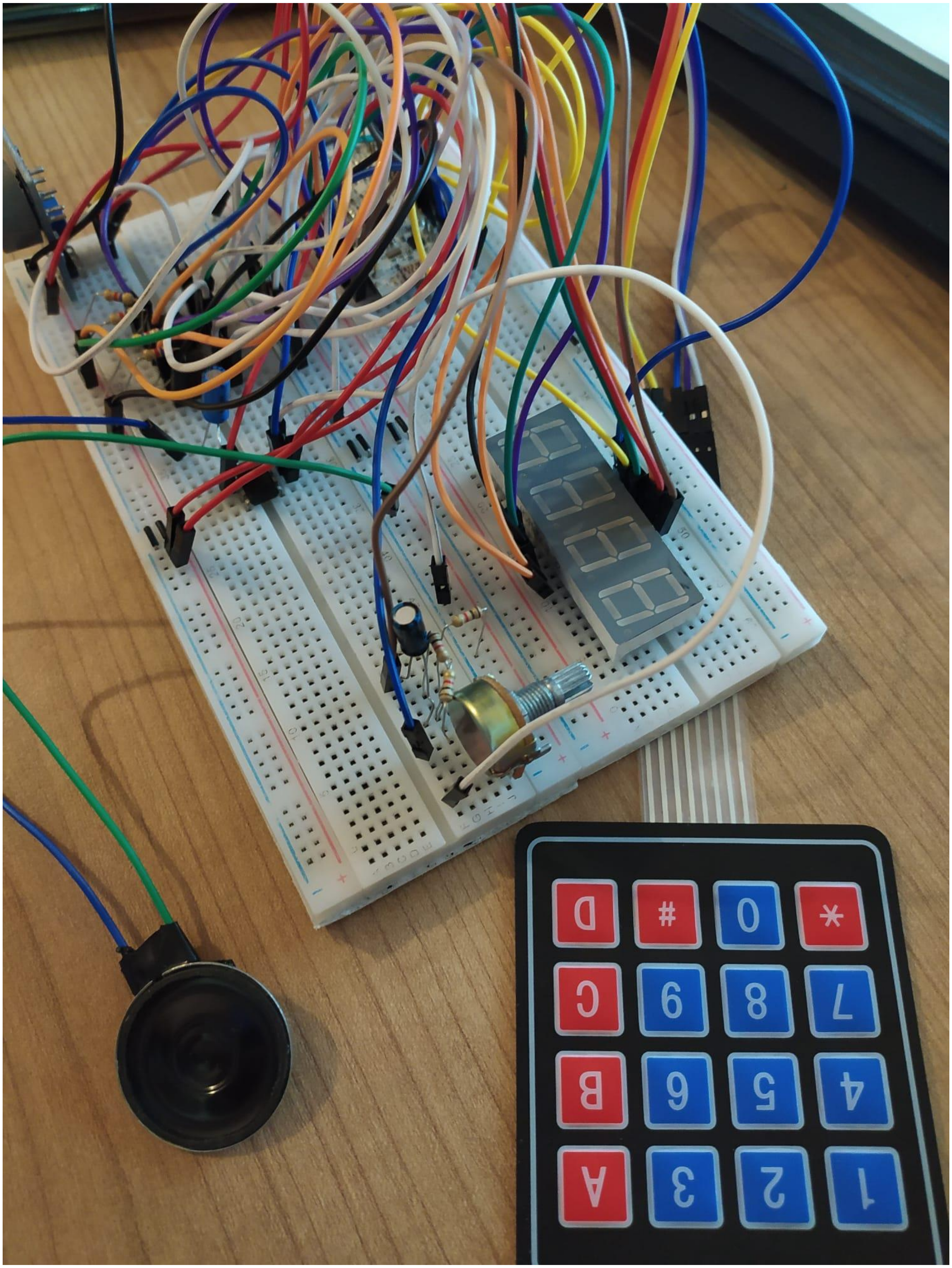

Figure 4: Circuit

Main

```
/*
* project2.c
*/
#include "lc512driver.h"
#include "ssdconfig.h"
#include "stm32g0xx.h"
#include <stdlib.h>
#include <stdbool.h>
//FIRST EEPROM COMMUNICATION ADDRESS
#define EEPROM_ADR1 0x50 //1010(Control Byte)000(A2 A1 A0)
//SECOND EEPROM COMMUNICATION ADDRESS
#define EEPROM_ADR2 0x54 //1010(Control Byte)100(A2 A1 A0)
//Max one track size that will write to EEPROM
#define MAX_TRACK_BYTE_SIZE 32000
//Write buffer
uint8_t* buffer_write;
//Read buffer
uint8_t* buffer_read;
//Write index
uint16_t bw_index = 0;
//Read index
uint16_t read_index = 0;
//Current recorded size
uint16_t record_size = 0;
//Current write memory address
uint16_t curr_memaddr = 0;
//Current read device address
uint8_t curr_devaddr = EEPROM_ADR1;
//Able to read operation
bool can_read = true;
//Able to button press
bool canButtonPress = true;
//Debouncing counter
uint8_t buttonPressCounter = 0;
//Returning idle counter
uint16_t idle_counter = 0;
//Storing state chars to display on SSD
char disp_letters[4];
//Recorded Track size
uint8_t track_size = 0;
//Recording time duration
uint8_t record_time = 5;
//Refers to initialized address of reading memory
uint16_t read_init_memaddr = 0;
```

```
//Current reading device address
uint8_t curr_readdevaddr = EEPROM_ADR1;
//Current reading memory address
uint16_t curr_readmemaddr = 0;
//Boolean array that refers to
//whether track is recorded or not
bool playable_track[4];
//STATES
typedef enum STATE
{
START,
IDLE,
RECORD,
PLAY,
PAUSE,
DELETE,
STATUS,
FULL,
INVALID
}STATE;
//TRACKS
typedef enum TRACK
{
TRACK1 = 1,
TRACK2,
TRACK3,
TRACK4
}TRACK;
//Selected Track
TRACK selected_track;
//Selected Current State
STATE curr_state;
//Basic common delay function
void delay(volatile unsigned int s)
{
for (; s > 0; s--);
}
//CLEAR KEYPAD ROWS
void clearRow(void) {
GPIOB->BRR = (1U << 6);
GPIOB->BRR = (1U << 7);
GPIOA->BRR = (1U << 0);
}
//SET KEYPAD ROWS
void setRow(void) {
GPIOB->ODR |= (1U << 6);
```

```
GPIOB->ODR |= (1U << 7);
GPIOA->ODR |= (1U << 0);
}
//Assigning proper chars to display array
//to show current state properly
void SetStateProperty(STATE state)
{
switch(state)
{
case START:
disp_letters[0] = '1';
disp_letters[1] = '7';
disp_letters[2] = '3';
disp_letters[3] = '4';
break;
case IDLE:
disp_letters[0] = 'i';
disp_letters[1] = 'd';
disp_letters[2] = 'l';
disp_letters[3] = 'e';
break;
case FULL:
disp_letters[0] = 'f';
disp_letters[1] = 'u';
disp_letters[2] = 'l';
disp_letters[3] = 'l';
break;
case RECORD:
disp_letters[0] = 'r';
disp_letters[1] = 'c';
disp_letters[2] = 'd';
disp_letters[3] = IntToChar(record_time);
break;
case PLAY:
disp_letters[0] = 'p';
disp_letters[1] = 'l';
disp_letters[2] = 'b';
disp_letters[3] = IntToChar(selected_track);
break;
case STATUS:
disp_letters[0] = 'a';
disp_letters[1] = 'v';
disp_letters[2] = 'a';
disp_letters[3] = IntToChar(track_size);
break;
case DELETE:
```

```
break;
case PAUSE:
disp_letters[0] = 'p';
disp_letters[1] = 'a';
disp_letters[2] = 'u';
disp_letters[3] = IntToChar(selected_track);
break;
case INVALID:
disp_letters[0] = 'i';
disp_letters[1] = 'n';
disp_letters[2] = 'v';
disp_letters[3] = 'd';
break;
default:
break;
}
}
//TIMER3 USED BY PWM
void INIT_PWM()
{
//Used TIM3 at D12 pin
RCC->APBENR1 |= (1U << 1);
//Set PB4 as alternate function
GPIOB->MODER &= ~(3U << 2 * 4);
GPIOB->MODER |= (2U << 2 * 4);
//Configure PB4 pins AF0
GPIOB->AFR[0] |= 1U << 4 * 4;
//Sets the duty cycle
TIM3->CCR1 = 500;
//PWM configuration begins
TIM3->CCMR1 |= (TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1);
//Enabling preload register
TIM3->CCMR1 |= (TIM_CCMR1_OC1PE);
//Enabling auto reload
TIM3->CR1 |= TIM_CR1_ARPE;
//Output pin active high
TIM3->CCER &= ~(TIM_CCER_CC1P);
//Enabling output pin
TIM3->CCER |= TIM_CCER_CC1E;
//Timer Prescaler Value
TIM3->PSC = 2;
//Auto reload value.
TIM3->ARR = 255;
//Start Timer
TIM3->DIER = (1 << 0);
TIM3->CR1 = (1 << 0);
```

```
}
//TRACK PLAYER & RECORDER
void INIT_TIMER2()
{
RCC->APBENR1 |= (1U);
TIM2->CR1 = 0;
TIM2->CR1 |= (1 << 7);
TIM2->CNT = 0;
// 6300Hz(Sampling Freq) = 1600000 / (ARR + 1) * (PSC + 1)
TIM2->PSC = 1;
TIM2->ARR = 1268;
TIM2->DIER = (1 << 0);
TIM2->CR1 = (1 << 0);
NVIC_SetPriority(TIM2_IRQn, 0);
NVIC_EnableIRQ(TIM2_IRQn);
}
//DEBOUNCING & DISPLAYER TIMER
void INIT_TIMER14()
{
RCC->APBENR2 |= (1U << 15);
TIM14->CR1 = 0;
TIM14->CR1 |= (1 << 7);
TIM14->CNT = 0;
TIM14->PSC = 1;
TIM14->ARR = 16000;//1MS
TIM14->DIER = (1 << 0);
TIM14->CR1 = (1 << 0);
NVIC_SetPriority(TIM14_IRQn, 0);
NVIC_EnableIRQ(TIM14_IRQn);
}
//Play sound
void PlaySound(uint8_t sound)
{
TIM3->CCR1 = (uint32_t)sound;
}
//Track and recorder interrupt
void TIM2_IRQHandler()
{
//RECORD STATE
if(curr_state == RECORD)
{
//ADC START CONVERSION
ADC1->CR |= (1U << 2);
//Wait until end of conversion flag is true
while (!(ADC1->ISR & (1U << 2))) {}
//Read data register and write it to buffer
```

```
if (bw_index < 128)
{
buffer_write[bw_index] = ADC1->DR;
++bw_index;
}
//Go inside after writing 128 byte to buffer
if (bw_index >= 127)
{
//Write 128 byte data buffer to EEPROM
//Page writing method used
WriteMultipleByte(curr_devaddr, curr_memaddr, buffer_write, 128);
//Increasing 128 byte recorded size of current EEPROM
record_size = (uint16_t)(record_size + 128);
//Forward 128 byte from current memory address of current EEPROM
curr_memaddr= (uint16_t)(curr_memaddr + 128);
//Decrease recording time one when we write multiples of 6400
bytes(32000/5=6400)
if(record_size == 6400 || record_size == 12800 || record_size ==
19200 ||
record_size == 25600 || record_size == 32000)
{
//Decrease recording time
record_time = (uint8_t)(record_time - 1);
//Assign current time to state char array
disp_letters[3] = IntToChar(record_time);
}
//Go inside when we write one track to EEPROM
if (record_size >= MAX_TRACK_BYTE_SIZE)
{
//ONE TRACK RECORDED
//Recorded track index
uint8_t track_recorded = 0;
//If recorded track size is one go inside
if(curr_memaddr == 32000)
{
//If we have used first EEPROM then it represents first track else
third track
track_recorded = curr_devaddr == EEPROM_ADR1 ? 0 : 2;
//Set recorded track to true
playable_track[track_recorded] = true;
}
//If recorded track size is two go inside
else if(curr_memaddr == 64000)
{
//If we have used first EEPROM then it represents second track
else fourth track
```

```
track_recorded = curr_devaddr == EEPROM_ADR1 ? 1 : 3;
//Set recorded track to true
playable_track[track_recorded] = true;
}
//Finish writing state, return idle state
curr_state = IDLE;
SetStateProperty(curr_state);
//Reset writing index
bw_index = 0;
//Reset recorded size
record_size = 0;
//Reset record time
record_time = 5;
//Update track size by increasing one
track_size = (uint8_t)(track_size + 1);
}
//Keep writing
else
{
//Reset writing index
bw_index = 0;
}
//If EEPROM is full then change EEPROM
if(curr_memaddr >= MAX_TRACK_BYTE_SIZE * 2)
{
//Reset current memory address
curr_memaddr = 0;
//Change current device address
curr_devaddr = curr_devaddr == EEPROM_ADR2 ? EEPROM_ADR1 :
EEPROM_ADR2;
}
}
}
//PLAYBACK STATE
else if(curr_state == PLAY)
{
//If read buffer is empty go inside
if(can_read)
{
//Read 128 byte data to buffer
ReadMultipleByte(curr_readdevaddr, curr_readmemaddr, buffer_read,
128);
//Ignore reading until whole buffer has been read
can_read = false;
}
//If read buffer has sounds then play it
```

```
if(read_index < 128 && !can_read)
{
PlaySound(buffer_read[read_index]);
read_index++;
}
//If reading of current buffer finish, then go inside
else if(read_index >= 128 && !can_read)
{
//If reading of last 128 byte has been occured then go inside
if(curr_readmemaddr >= ((read_init_memaddr + MAX_TRACK_BYTE_SIZE) -
128))
{
//Reset reading memory address
curr_readmemaddr = 0;
//Reset reading device address
curr_readdevaddr = EEPROM_ADR1;
//Reset reading index
read_index = 0;
//Finish reading and return idle state
curr_state = IDLE;
SetStateProperty(curr_state);
//Reset can_read bool to continue reading
can_read = true;
}
//If there are values that must read then go inside
else
{
//Forward 128 byte from current reading memory address
curr_readmemaddr = (uint16_t)(curr_readmemaddr + 128);
//Reset reading index
read_index = 0;
//Reset can_read bool to continue reading
can_read = true;
}
}
}
//DELETE STATE
else if(curr_state == DELETE && track_size > 0 &&
playable_track[selected_track - 1])
{
//Return idle state after deleting
curr_state = IDLE;
SetStateProperty(curr_state);
//Delete selected track
playable_track[selected_track - 1] = false;
//Decrease current track size one
```

```
track_size = (uint8_t)(track_size - 1);
}
//INVALID OPERATION STATE
else if(curr_state == DELETE && (track_size <= 0 ||
!playable_track[selected_track - 1]))
{
//Set current state to invalid state
curr_state = INVALID;
SetStateProperty(curr_state);
}
//Resetting pending register to continue
TIM2->SR &= ~(1U << 0);
}
void TIM14_IRQHandler(void)
{
//If debouncing exist, go inside
if(!canButtonPress)
{
idle_counter = 0;
//Decreasing button counter time
buttonPressCounter++;
//Checking whether buttonPressCounter has reached zero or lower than
zero
if (buttonPressCounter >= 100)
{
//Resetting debouncing preventer elements so that any button press
can be read
buttonPressCounter = 0;
canButtonPress = true;
}
}
//If there are no button press go inside
else if(curr_state != RECORD && curr_state != PLAY && curr_state !=
PAUSE && curr_state !=
IDLE)
{
//Increase returning idle counter
idle_counter++;
//If returning idle counter reachs 10000
//then return idle state(1MS TIMER CLOCK * 10000 = 10 SEC)
if(idle_counter >= 10000)
{
//Reset returning idle counter
idle_counter = 0;
//Set current state to idle state
curr_state = IDLE;
```

```
SetStateProperty(curr_state);
}
}
//Displaying on SSD
uint8_t offset = 0;
int iterator;
for(iterator = 3; iterator >= 0; --iterator)
{
//Display one of current state letters
DisplayChar(disp_letters[iterator]);
//For displaying smoothness
delay(40);
//Turn off all ssd leds
ResetDisplay();
//Shift digit section
ShiftDigit((unsigned int)offset);
//Increase offset from rightmost digit on SSD
offset = (uint8_t)(offset + 1);
}
//Resetting pending register to continue
TIM14->SR &= ~(1U << 0);
}
void EXTI4_15_IRQHandler(void)
{
 //COLUMN 1
 if (((GPIOB->IDR >> 5) & 1))
 {
//If debouncing does not exist go inside
if(canButtonPress)
{
clearRow();
//SELECT TRACK 1
GPIOB->ODR ^= (1U << 7);
if(((GPIOB->IDR >> 5) & 1))
{
//BUTTON(1)
canButtonPress = false;
selected_track = TRACK1;
}
GPIOB->ODR ^= (1U << 7);
//SELECT TRACK 4
GPIOB->ODR ^= (1U << 6);
if((((GPIOB->IDR >> 5) & 1)))
{
//BUTTON(4)
canButtonPress = false;
```

```
selected_track = TRACK4;
}
GPIOB->ODR ^= (1U << 6);
//SELECT DELETE STATE
GPIOA->ODR ^= (1U << 0);
if((((GPIOB->IDR >> 5) & 1)))
{
//BUTTON(7)
canButtonPress = false;
curr_state = DELETE;
SetStateProperty(curr_state);
}
GPIOA->ODR ^= (1U << 0);
setRow();
}
EXTI->RPR1 |= (1U << 5);
 }
 //COLUMN 2
 if (((GPIOA->IDR >> 11) & 1))
 {
 //If debouncing does not exist go inside
if(canButtonPress)
{
clearRow();
//SELECT TRACK 2
GPIOB->ODR ^= (1U << 7);
if (((GPIOA->IDR >> 11) & 1) && curr_state != PLAY)
{
//BUTTON(2)
canButtonPress = false;
selected_track = TRACK2;
}
GPIOB->ODR ^= (1U << 7);
//SELECT PLAYBACK STATE
GPIOB->ODR ^= (1U << 6);
if((((GPIOA->IDR >> 11) & 1)) && selected_track > 0)
{
//BUTTON(5) PLAY
canButtonPress = false;
//PAUSE STATE CONTROLLER
bool pass_init = true;
//If play button pressed when playing then set current state
//to pause state
if(curr_state == PLAY && playable_track[selected_track-1])
{
//Pause state exist
```

```
pass_init = false;
curr_state = PAUSE;
SetStateProperty(curr_state);
}
//If play button pressed when pausing then set current state
//to play state
else if(curr_state == PAUSE && playable_track[selected_track-1])
{
//Pause state exist
pass_init = false;
curr_state = PLAY;
SetStateProperty(curr_state);
}
//Initialize track that will be played
//If pause state exist then do not initialize
//to prevent resetting memory
if(pass_init && playable_track[selected_track-1])
{
//Selecting from first EEPROM
if((selected_track == TRACK1 && playable_track[0]) || (selected_track
==
TRACK2 && playable_track[1]))
{
//Set reading device address to first EEPROM address
curr_readdevaddr = EEPROM_ADR1;
//Set reading memory address to selected track beginning address
curr_readmemaddr = selected_track == TRACK1 ? 0 : 32000;
//Set current state to playback state
curr_state = PLAY;
SetStateProperty(curr_state);
}
//Selecting from second EEPROM
else if((selected_track == TRACK3 && playable_track[2]) ||
(selected_track == TRACK4 && playable_track[3]))
{
//Set reading device address to second EEPROM address
curr_readdevaddr = EEPROM_ADR2;
//Set reading memory address to selected track beginning address
curr_readmemaddr = selected_track == TRACK3 ? 0 : 32000;
//Set current state to playback state
curr_state = PLAY;
SetStateProperty(curr_state);
}
//Set initializing address to beginning of reading current memory
address
read_init_memaddr = curr_readmemaddr;
```

```
}
}
GPIOB->ODR ^= (1U << 6);
//Set current state to status state
GPIOA->ODR ^= (1U << 0);
if ((((GPIOA->IDR >> 11) & 1)))
{
//BUTTON(8)
canButtonPress = false;
curr_state = STATUS;
SetStateProperty(curr_state);
}
GPIOA->ODR ^= (1U << 0);
setRow();
}
EXTI->RPR1 |= (1U << 11);
 }
 //COLUMN 3
 if (((GPIOA->IDR >> 12) & 1))
 {
 //If debouncing does not exist go inside
if(canButtonPress)
{
clearRow();
//SELECT TRACK 3
GPIOB->ODR ^= (1U << 7);
if (((GPIOA->IDR >> 12) & 1))
{
//BUTTON(3)
canButtonPress = false;
selected_track = TRACK3;
}
GPIOB->ODR ^= (1U << 7);
//SELECT RECORD STATE
GPIOB->ODR ^= (1U << 6);
if ((((GPIOA->IDR >> 12) & 1)))
{
canButtonPress = false;
//BUTTON(6)
//If EEPROMs are not full
if(track_size < 4)
{
//Set current state to current state
curr_state = RECORD;
SetStateProperty(curr_state);
//Check selected track is recorded
```

```
if(!playable_track[0] || !playable_track[1])
{
//Set device address to first EEPROM address
curr_devaddr = EEPROM_ADR1;
//Set reading memory address
curr_memaddr = !playable_track[0] ? 0 : 32000;
}
//Check selected track is recorded
else if(!playable_track[2] || !playable_track[3])
{
//Set device address to first EEPROM address
curr_devaddr = EEPROM_ADR2;
//Set reading memory address
curr_memaddr = !playable_track[2] ? 0 : 32000;
}
}
//If EEPROMs are full change current state to current state
else
{
curr_state = FULL;
SetStateProperty(curr_state);
}
}
GPIOB->ODR ^= (1U << 6);
//EMPTY KEY
GPIOA->ODR ^= (1U << 0);
if ((((GPIOA->IDR >> 12) & 1)))
{
//BUTTON(9)
canButtonPress = false;
}
GPIOA->ODR ^= (1U << 0);
setRow();
}
EXTI->RPR1 |= (1U << 12);
 }
}
//ADC INITIALIZATION
void ADCInit()
{
//PA1 USED FOR ADC
//GPIOA-B PORT ENABLED
RCC->IOPENR |= (3U);
//ADC CLOCK ENABLED
RCC->APBENR2 |= (1U << 20);
//ADC VOLTAGE REGULATOR ENABLED
```

```
ADC1->CR |= (1U << 28);
//WAIT AT LEAST 20US, AS SAID IN LECTURE
delay(300000);
//8 BIT RESOLUTION SELECTED
ADC1->CFGR1 |= (1U << 4);
//ADC CALIBRATION ENABLED
ADC1->CR |= (1U << 31);
//WAIT UNTIL END OF CALIBRATION FLAG IS SET
while (!(ADC1->ISR & (1U << 11)));
//ADC SAMPLING TIME SELECTED MAX 3.5 ADC CLOCK CYCLE
ADC1->SMPR |= (1U);
//ADC CHANNEL 1 SELECTED
ADC1->CHSELR |= (1U << 1);
//ADC ENABLED
ADC1->CR |= (1U);
//WAIT UNTIL ADC GETS READY
while (!(ADC1->ISR & 1U));
}
//KEYPAD & SSD INITIALIZER
void Keypad_SSD_GPIO_Init(void)
{
//Set B6 B7 A0 as output rows
GPIOB->MODER &= ~(3U << 2 * 6);
GPIOB->MODER |= (1U << 2 * 6);
GPIOB->MODER &= ~(3U << 2 * 7);
GPIOB->MODER |= (1U << 2 * 7);
GPIOA->MODER &= ~(3U << 2 * 0);
GPIOA->MODER |= (1U << 2 * 0);
//Set B5 A11 A12 as input column
GPIOB->MODER &= ~(3U << 2 * 5);
GPIOB->PUPDR |= (2U << 2 * 5);
GPIOA->MODER &= ~(3U << 2 * 11);
GPIOA->PUPDR |= (2U << 2 * 11);
GPIOA->MODER &= ~(3U << 2 * 12);
GPIOA->PUPDR |= (2U << 2 * 12);
//SSD B0 PORT
GPIOB->MODER &= ~(3U << 2 * 0);
GPIOB->MODER |= (1U << 2 * 0);
GPIOB->ODR |= (1U << 1 * 0);
//KEYPAD +5V PORTS
GPIOB->ODR |= (1U << 6);
GPIOB->ODR |= (1U << 7);
GPIOA->ODR |= (1U << 0);
//Set external interrupt for keypad
EXTI->EXTICR[1] |= (1U << 8 * 1);//B5
EXTI->RTSR1 |= (1U << 5);
```

```
EXTI->IMR1 |= (1U << 5);
EXTI->EXTICR[2] |= (0U << 8 * 3);//A11
EXTI->RTSR1 |= (1U << 11);
EXTI->IMR1 |= (1U << 11);
EXTI->EXTICR[3] |= (0U << 8 * 0);//A12
EXTI->RTSR1 |= (1U << 12);
EXTI->IMR1 |= (1U << 12);
NVIC_SetPriority(EXTI4_15_IRQn, 0);
NVIC_EnableIRQ(EXTI4_15_IRQn);
}
int main(void)
{
//Allocating 128 byte data on heap memory for write buffer
buffer_write = (uint8_t*)malloc(sizeof(uint8_t) * 128);
//Allocating 128 byte data on heap memory for read buffer
buffer_read = (uint8_t*)malloc(sizeof(uint8_t) * 128);
//Set playable track bools to false not to play or delete
playable_track[0]=playable_track[1]=playable_track[2]=playable_track[
3]=false;
//Set current state START
curr_state = START;
//Set current state properties
SetStateProperty(curr_state);
//Initializing ADC
ADCInit();
//Initializing PWM
INIT_PWM();
//Initializing EEPROMs
INIT_EEPROM_512();
//Initializing TIMER14
INIT_TIMER14();
//Initializing KEYPAD & SSD
Keypad_SSD_GPIO_Init();
//Initializing TIMER2
INIT_TIMER2();
while (1)
{
}
//Releasing memory
free(buffer_read);
free(buffer_write);
}
```

.h

```
#ifndef SSDCONFIG_H_
#define SSDCONFIG_H_
#include "stm32g0xx.h"
#include <string.h>
//Segments BEGIN
//To light A LED
void SetSegmentA() {
GPIOA->MODER &= ~(3U << 2 * 4);
GPIOA->MODER |= (1U << 2 * 4);
}
//To light B LED
void SetSegmentB() {
GPIOA->MODER &= ~(3U << 2 * 5);
GPIOA->MODER |= (1U << 2 * 5);
}
//To light C LED
void SetSegmentC() {
GPIOA->MODER &= ~(3U << 2 * 6);
GPIOA->MODER |= (1U << 2 * 6);
}
//To light D LED
void SetSegmentD() {
GPIOA->MODER &= ~(3U << 2 * 7);
GPIOA->MODER |= (1U << 2 * 7);
}
//To light E LED
void SetSegmentE() {
GPIOA->MODER &= ~(3U << 2 * 8);
GPIOA->MODER |= (1U << 2 * 8);
}
//To light F LED
void SetSegmentF() {
GPIOA->MODER &= ~(3U << 2 * 9);
GPIOA->MODER |= (1U << 2 * 9);
}
//To light G LED
void SetSegmentG() {
GPIOA->MODER &= ~(3U << 2 * 10);
GPIOA->MODER |= (1U << 2 * 10);
}
//Segments END
//Numbers BEGIN
//To light Number 0
void SetNumberZero()
```

```
{
SetSegmentA();
SetSegmentB();
SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegmentF();
}
//To light Number 1
void SetNumberOne() {
SetSegmentB();
SetSegmentC();
}
//To light Number
2
void SetNumberTwo() {
SetSegmentA();
SetSegmentB();
SetSegmentD();
SetSegmentE();
SetSegmentG();
}
//To light Number 3
void SetNumberThree() {
SetSegmentA();
SetSegmentB();
SetSegmentC();
SetSegmentD();
SetSegmentG();
}
//To light Number 4
void SetNumberFour() {
SetSegmentB();
SetSegmentC();
SetSegmentF();
SetSegmentG();
}
//To light Number 5
void SetNumberFive() {
SetSegmentA();
SetSegmentC();
SetSegmentD();
SetSegmentF();
SetSegmentG();
}
//To light Number 6
```

```
void SetNumberSix() { SetSegmentA();
SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Number 7
void SetNumberSeven()
{
SetSegmentA();
SetSegmentB();
SetSegmentC();
}
//To light Number 8
void SetNumberEight() {
SetSegmentA();
SetSegmentB();
SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Number 9
void SetNumberNine() {
SetSegmentA();
SetSegmentB();
SetSegmentC();
SetSegmentD();
SetSegmentF();
SetSegmentG();
}
//To light Negative Sig
n
void SetNegativeSign() {
SetSegmentG();
}
//Numbers END
//Letters Begin
//To light Letter A
void SetLetterA() {
SetSegmentA();
SetSegmentB();
SetSegmentC();
SetSegmentE();
```

```
SetSegmentF();
SetSegmentG();
}
//To light Letter B
void SetLetterB() { SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Letter C
void SetLetterC() {
SetSegmentD();
SetSegmentE();
SetSegmentG();
}
//To light Letter D
void SetLetterD() {
SetSegmentB();
SetSegmentC();
SetSegmentD()
;
SetSegmentE();
SetSegmentG();
}
//To light Letter E
void SetLetterE() {
SetSegmentA();
SetSegmentD();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Letter I
void SetLetterI() {
SetSegmentE();
SetSegmentF();
}
//To light Letter N
void SetLetterN() {
SetSegmentC();
SetSegmentE();
SetSegmentG();
}
//To light Letter V
void SetLetterV() {
```

```
SetSegmentC();
SetSegmentD();
SetSegmentE();
}
//To light Letter O
void SetLetterO() {
SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegment
G();
}
//To light Letter U
void SetLetterU() {
SetSegmentB();
SetSegmentC();
SetSegmentD();
SetSegmentE();
SetSegmentF();
}
//To light Letter F
void SetLetterF()
{
SetSegmentA();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Letter L
void SetLetterL()
{
SetSegmentD();
SetSegmentE();
SetSegmentF();
}
//To light Letter P
void SetLetterP()
{
SetSegmentA();
SetSegmentB();
SetSegmentE();
SetSegmentF();
SetSegmentG();
}
//To light Letter R
void SetLetterR()
```

```
{
SetSegmentE();
SetSegmentG();
}
//Letters End
//To turn leftmost digit when shifter reachs to rightmost digit
void TurnBeginning()
{
//B0
GPIOB->MODER &= ~(3U << 2 * 0);
GPIOB->MODER |= (1U << 2 * 0);
GPIOB->BRR = (1U << 0);
GPIOB->ODR |= (1U << 0);
//B1
GPIOB->MODER &= ~(0U << 2 * 1);
GPIOB->BRR = (1U << 1);
//B2
GPIOB->MODER &= ~(0U << 2 * 2);
GPIOB->BRR = (1U << 2);
//B3
GPIOB->MODER &= ~(0U << 2 * 3);
GPIOB->BRR = (1U << 3);
}
//To shift digits that will be displayed
void ShiftDigit(unsigned int currIndex)
{
if (currIndex >= 3)
{
TurnBeginning();
return;
}
GPIOB->MODER &= ~(0U << 2 * (currIndex));
GPIOB->MODER &= ~(3U << 2 * (currIndex + 1));
GPIOB->MODER |= (1U << 2 * (currIndex + 1));
GPIOB->BRR = (1U << currIndex);
GPIOB->ODR |= (1U << (currIndex + 1));
}
//To display numbers
void DisplayChar(char ch)
{
switch (ch)
{
case '0':
SetNumberZero();
break;
case '1':
```

```
SetNumberOne();
break;
case '2':
SetNumberTwo();
break;
case '3':
SetNumberThree();
break;
case '4':
SetNumberFour();
break;
case '5':
SetNumberFive();
break;
case '6':
SetNumberSix();
break;
case '7':
SetNumberSeven();
break;
case '8':
SetNumberEight();
break;
case '9':
SetNumberNine();
break;
case 'a':
SetLetterA();
break;
case 'b':
SetLetterB();
break;
case 'c':
SetLetterC();
break;
case 'd':
SetLetterD();
break;
case 'e':
SetLetterE();
break;
case 'i':
SetLetterI();
break
;
case 'n'
```

```
:
SetLetterN();
break
;
case 'v'
:
SetLetterV();
break
;
case 'o'
:
SetLetterO();
break
;
case 'u'
:
SetLetterU();
break
;
case 'f'
:
SetLetterF();
break
;
case 'l'
:
SetLetterL();
break
;
case 'p'
:
SetLetterP();
break
;
case 'r'
:
SetLetterR();
break
;
default
:
break
;
}
}
char IntToChar
```

```
(uint8_t digi
t
)
{
switch
(digit
)
{
 case
0
:
return '0'
;
 case
1
:
return '1'
;
 case
2
:
return '2'
;
 case
3
:
return '3'
;
 case
4
:
return '4'
;
 case
5
:
return '5'
;
 case
6
:
return '6'
;
 case
7
:
```

```
return '7'
;
 case
8
:
return '8'
;
 case
9
:
return '9'
;
 default
:
break
;
}
return '0'
;
}
//Clearing displayer
void ResetDisplay()
{
uint8_t index;
for(index = 4;index < 11;index++)
{
GPIOA->MODER &= ~(3U << 2 * index);
GPIOA->MODER |= (3U << 2 * index);
GPIOA->ODR &= ~(1U << index);
}
}
#endif /* SSDCONFIG_H_ */
```

.h

```
#ifndef LC512DRIVER_H_
#define LC512DRIVER_H_
#include "stm32g0xx.h"
//Initializing EEPROM
void INIT_EEPROM_512();
//Writing single byte
void WriteSingleByte(uint8_t devAddr, uint16_t destAddr, uint8_t
data);
//Writing 128 bytes
```

```
void WriteMultipleByte(uint8_t devAddr, uint16_t startAddr, uint8_t* data, uint8_t size);
//Reading single byte
void ReadSingleByte(uint8_t devAddr, uint16_t resAddr, uint8_t* data);
//Reading 128 bytes
void ReadMultipleByte(uint8_t devAddr, uint16_t resAddr, uint8_t* data, uint16_t size);
#endif /* LC512DRIVER_H_ */
```

.c

```
#include "lc512driver.h"
//Initializing EEPROM
void INIT_EEPROM_512()
{
//PB8-PB9 pins will activate
//alternate function 6
//PB8 as AF6
GPIOB->MODER &= ~(3U << 2*8);
GPIOB->MODER |= (2U << 2*8);
GPIOB->OTYPER|= (1U << 8);
//choose AF from mux
GPIOB->AFR[1] &= ~(0xFU << 4*0);
GPIOB->AFR[1] |= (6 << 4*0);
//PB9 as AF6
GPIOB->MODER &= ~(3U << 2*9);
GPIOB->MODER |= (2U << 2*9);
GPIOB->OTYPER|= (1U << 9);
//choose AF from mux
GPIOB->AFR[1] &= ~(0xFU << 4*1);
GPIOB->AFR[1] |= (6 << 4*1);
//enable I2C1
RCC->APBENR1 |= (1U << 21);
I2C1->CR1 = 0;
I2C1->CR1 |= (1 << 7);//ERR1, error interrupt
//TIMING REGISTERS FOR STANDART MODE
I2C1->TIMINGR |= (3 << 28);//PRESC
I2C1->TIMINGR |= (0x13 << 0);//SCLL
I2C1->TIMINGR |= (0xF << 8);//SCLR
I2C1->TIMINGR |= (0x2 << 16);//SDADEL
I2C1->TIMINGR |= (0x4 << 20);//SCLDEL
I2C1->CR1 = (1U << 0);//PE=0
NVIC_SetPriority(I2C1_IRQn,0);
NVIC_EnableIRQ(I2C1_IRQn);
```

```
}
//Writing single byte
void WriteSingleByte(uint8_t devAddr, uint16_t destAddr, uint8_t
data)
{
//Data sheet pattern applied
//Write operation
I2C1->CR2=0;
//Slave address
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
//Number of bytes that will send
I2C1->CR2 |= (3U << 16);
I2C1->CR2 |= (1U << 25);//AUTOEND
I2C1->CR2 |= (1U << 13);//Start condition
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(destAddr >> 8);//transmit data register
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(destAddr & 0xFF);//transmit data register
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)data;//transmit data register
}
//Writing 128 bytes
void WriteMultipleByte(uint8_t devAddr, uint16_t startAddr, uint8_t*
data, uint8_t size)
{
//Avoid writing more than 128 byte and less than 0 byte
if(size > 128 && size <= 0)
{
return;
}
//Data sheet pattern applied
//Write operation
I2C1->CR2=0;
//Slave address
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
//Number of bytes that will send
I2C1->CR2 |= ((uint32_t)(2U + size) << 16);
I2C1->CR2 |= (1U << 25);//AUTOEND
I2C1->CR2 |= (1U << 13);//Start condition
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(startAddr >> 8);//transmit data register
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(startAddr & 0xFF);//transmit data register
uint8_t i;
for(i = 0; i < size; i++)
{
```

```
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)data[i];//transmit data register
}
}
//Reading single byte
void ReadSingleByte(uint8_t devAddr, uint16_t resAddr, uint8_t* data)
{
//Data sheet pattern applied
//Write operation
I2C1->CR2 = 0;
//Slave address
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
//Number of bytes that will send
I2C1->CR2 |= (2U << 16);
I2C1->CR2 |= (1U << 13);//Start condition
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(resAddr >> 8);//transmit data register
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(resAddr & 0xFF);//transmit data register
while(!(I2C1->ISR & (1 << 6)));//TC
//read operation(read data)
I2C1->CR2=0;
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
I2C1->CR2 |= (1U << 10);//READ MODE
I2C1->CR2 |= (1U << 16);//NUMBER OF BYTES
I2C1->CR2 |= (1U << 15);//NACK
I2C1->CR2 |= (1U << 25);//AUTOEND
I2C1->CR2 |= (1U << 13);//start condition
while(!(I2C1->ISR & (1 << 2)));//wait until RXNE=1
*data = (uint8_t)I2C1->RXDR;
}
//Reading 128 bytes
void ReadMultipleByte(uint8_t devAddr, uint16_t resAddr, uint8_t*
data, uint16_t size)
{
//Data sheet pattern applied
//Write operation
I2C1->CR2 = 0;
//Slave address
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
//Number of bytes that will send
I2C1->CR2 |= (2U << 16);
I2C1->CR2 |= (1U << 13);//Start condition
while(!(I2C1->ISR & (1 << 1)));//TXIS
I2C1->TXDR = (uint32_t)(resAddr >> 8);//transmit data register
while(!(I2C1->ISR & (1 << 1)));//TXIS
```

```
I2C1->TXDR = (uint32_t)(resAddr & 0xFF);//transmit data register
while(!(I2C1->ISR & (1 << 6)));//TC
//Read operation
I2C1->CR2=0;
//Slave address
I2C1->CR2 |= ((uint32_t)(devAddr << 1));
I2C1->CR2 |= (1U << 10);//READ MODE
//Number of bytes that will send
I2C1->CR2 |= ((1U * size) << 16);
I2C1->CR2 |= (1U << 25);//AUTOEND
I2C1->CR2 |= (1U << 13);//Start condition
uint16_t i;
for(i = 0; i < size; ++i)
{
while(!(I2C1->ISR & (1 << 2)));//wait until RXNE=1
data[i] = (uint8_t)I2C1->RXDR;
}
I2C1->CR2 |= (1U << 15);//NACK
}
```