GEBZE TECHNICAL UNIVERSITY

ELECTRONICS ENGINEERING

ELEC 334

Microprocessors Lab

PROJECT #3

Digital Voice Recorder

Prepared by
1) 1801022037 Ömer Emre POLAT

1. Introduction

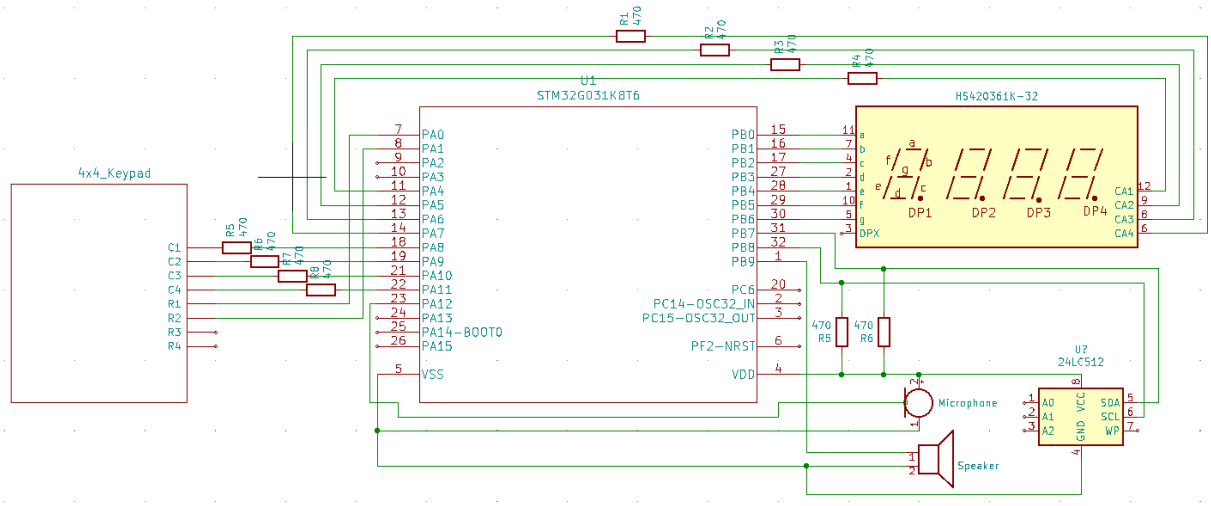
In this Project we will make a fully operational digital voice recorder. This voice recorder will be at least able to store one 5 second track in its EEPROM units. There will be a UI with the seven-segment display and keypad for user interaction.

2. Project Design

2.1. Flowchart and Block Diagram

First, we will design the board connections and the flowchart for our code for ease of design.

The board connections will be drawn on KiCAD using the model libraries of the components.



All the resistors seen in the figure are 470 ohm resistors. There are single resistor sets going to the both keypad and the seven segment display because the lines that do not have a resistor will be connected in series to ones that has a resistor so there is no need to use two sets of resistors.

The resistance values are calculated using the datasheet of the STM32 board and the HS420361K-32.

Absolute Maximum Rating (Ta = 25℃)

PARAMETER	RED
DC Forward Current Per Segment	30
Peak Current Per Segment (1)	70

5.2 Absolute maximum ratings

Stresses above the absolute maximum ratings listed in *Table 18*, *Table 19* and *Table 20* may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these conditions is not implied. Exposure to maximum rating conditions for extended periods may affect device reliability.

All voltages are defined with respect to V_{SS}.

Table 18. Voltage characteristics

Symbol	Ratings	Min	Max	Unit
V_{DD}	External supply voltage	- 0.3	4.0	V
V_{BAT}	External supply voltage on VBAT pin	- 0.3	4.0	
V_{REF+}	External voltage on VREF+ pin	- 0.3	Min(V_{DD} + 0.4, 4.0)	
V_{IN}(1)	Input voltage on FT_xx	- 0.3	V_{DD} + 4.0(2)	
	Input voltage on any other pin	- 0.3	4.0	

Table 19. Current characteristics

Symbol	Ratings	Max	Unit
$I_{VDD/VDDA}$	Current into VDD/VDDA power pin (source)(1)	100	mA
$I_{VSS/VSSA}$	Current out of VSS/VSSA ground pin (sink)(1)	100	
$I_{IO(PIN)}$	Output current sunk by any I/O and control pin except FT_f	15	
	Output current sunk by any FT_f pin	20	
	Output current sourced by any I/O and control pin	15	
$\Sigma I_{IO(PIN)}$	Total output current sunk by sum of all I/Os and control pins	80	
	Total output current sourced by sum of all I/Os and control pins	80	
$I_{INJ(PIN)}$(2)	Injected current on a FT_xx pin	-5 / NA(3)	
$\Sigma\lvert I_{INJ(PIN)}\rvert$	Total injected current (sum of all I/Os and control pins)(4)	25	

$$I_{keypad,ssd} = \frac{V}{R_{total}} = \frac{3.33V}{470} = 7.08mA < 15mA \ \& \ 30mA$$

$$R_{i2c,min} = \frac{(V_{bus} - V_{OL})}{I_{OL}} = \frac{3.33 - 0.99}{13.5mA} = 171\ ohms < 470\ ohms$$

Now that the absolute maximum DC forward current values and I2C pull-up resistor values are set, we can layout the pins.

The pin connections are as follows:

PB0 -> A (SSD)

PB1 -> B (SSD)

....

PB6 -> G (SSD)

PA0 -> R1 (KP)

PA1 -> R2 (KP)

PA8 -> C1 (KP)

PA9 -> C2 (KP)

PA10 -> C3 (KP)

PA4 -> D1 (SSD) PA11 -> C4 (KP)

PA5 -> D2 (SSD) PA12 -> R3 (KP)

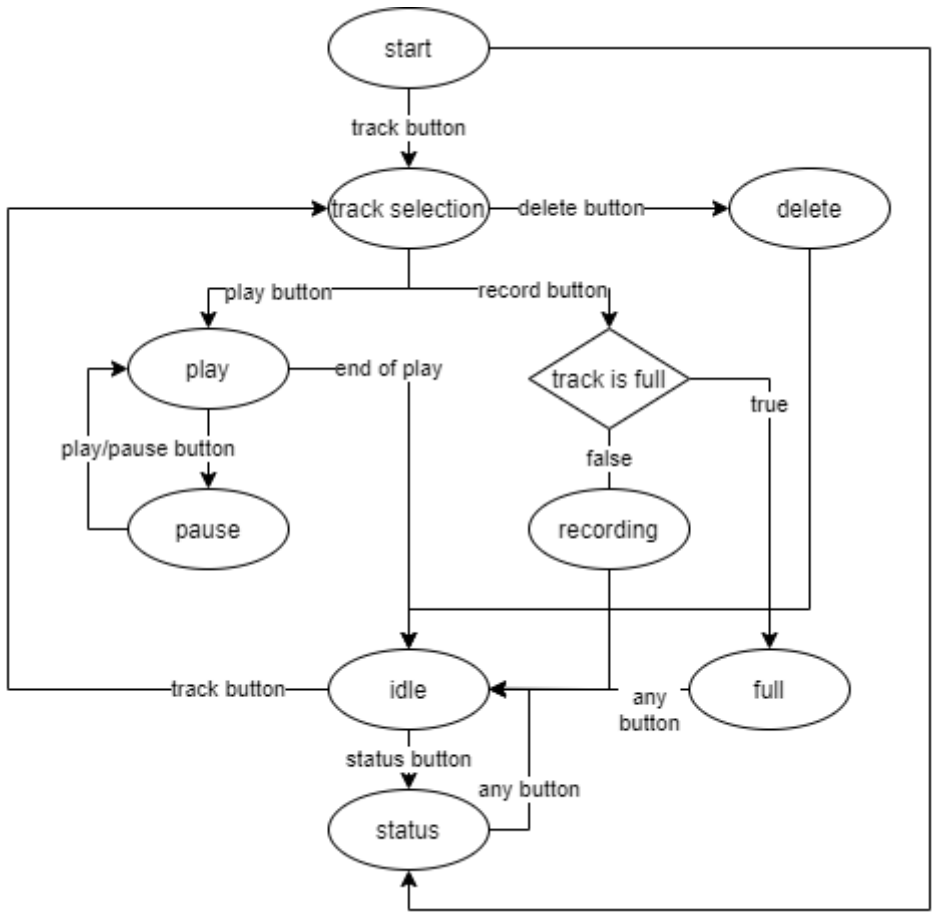
PA6 -> D3 (SSD) PA15 -> R4 (KP)

PA7 -> D4 (SSD)

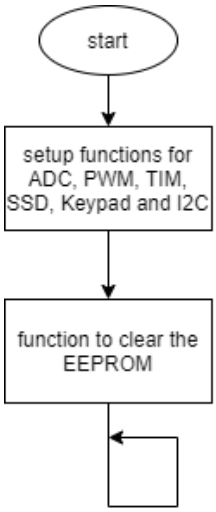
Now that the pins are chosen we can start designing the state diagram and the flowchart for the code.

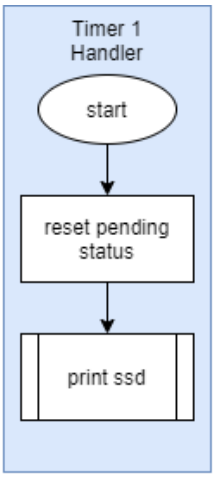
Code will first set the peripherals and pin input output modes. The runtime code will start with start state and when a button is pressed or 10 seconds have passed it will switch to the idle state where the 1837 number is shown. After an input is registered in idle state the state will be changed to the according state. After the state is changed the state operation will be performed whether if it is a record or playback or status etc.

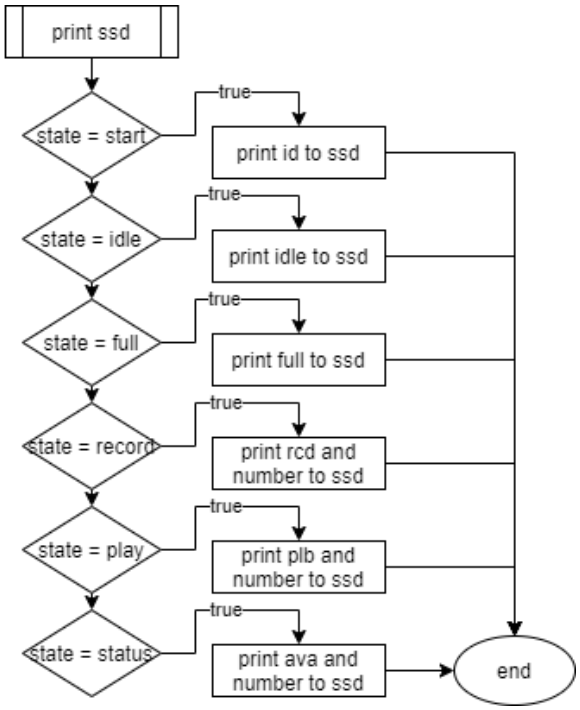
After the track 1 is recorded a playback input will again change the state to playback and start reading the EEPROM for the analog values. These values will be used the generate the output PWM signal for the speaker. This PWM will be calculated by scaling the 8 bit resolution value (0-255) to the duty cycle percentage of the PWM.

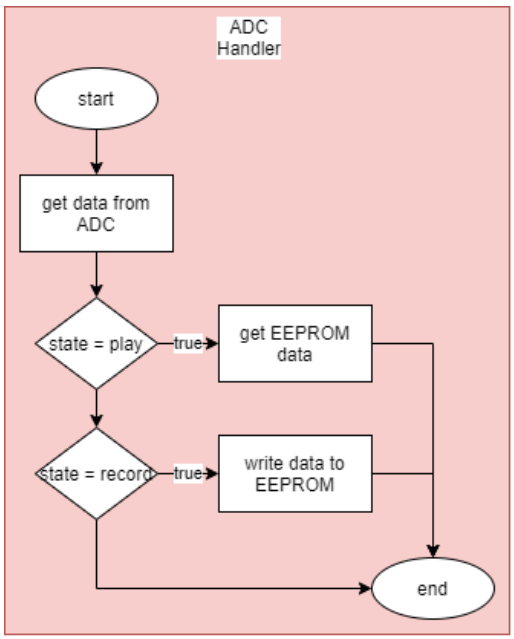


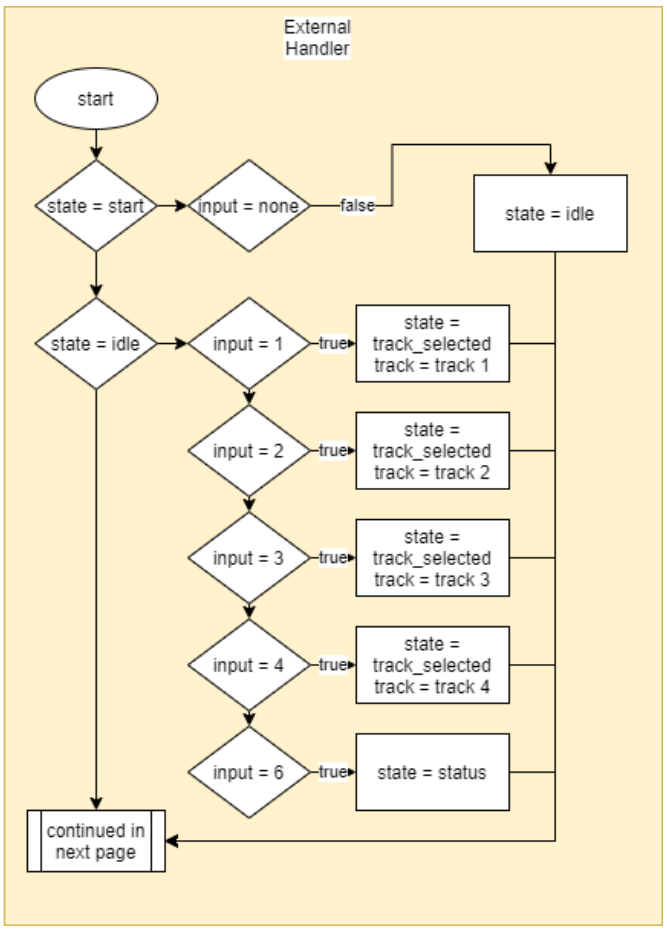
The state machine can be seen above. Now that the state diagram is created, we can design the flowchart of the main code. The flowchart will include all the interrupt routines.

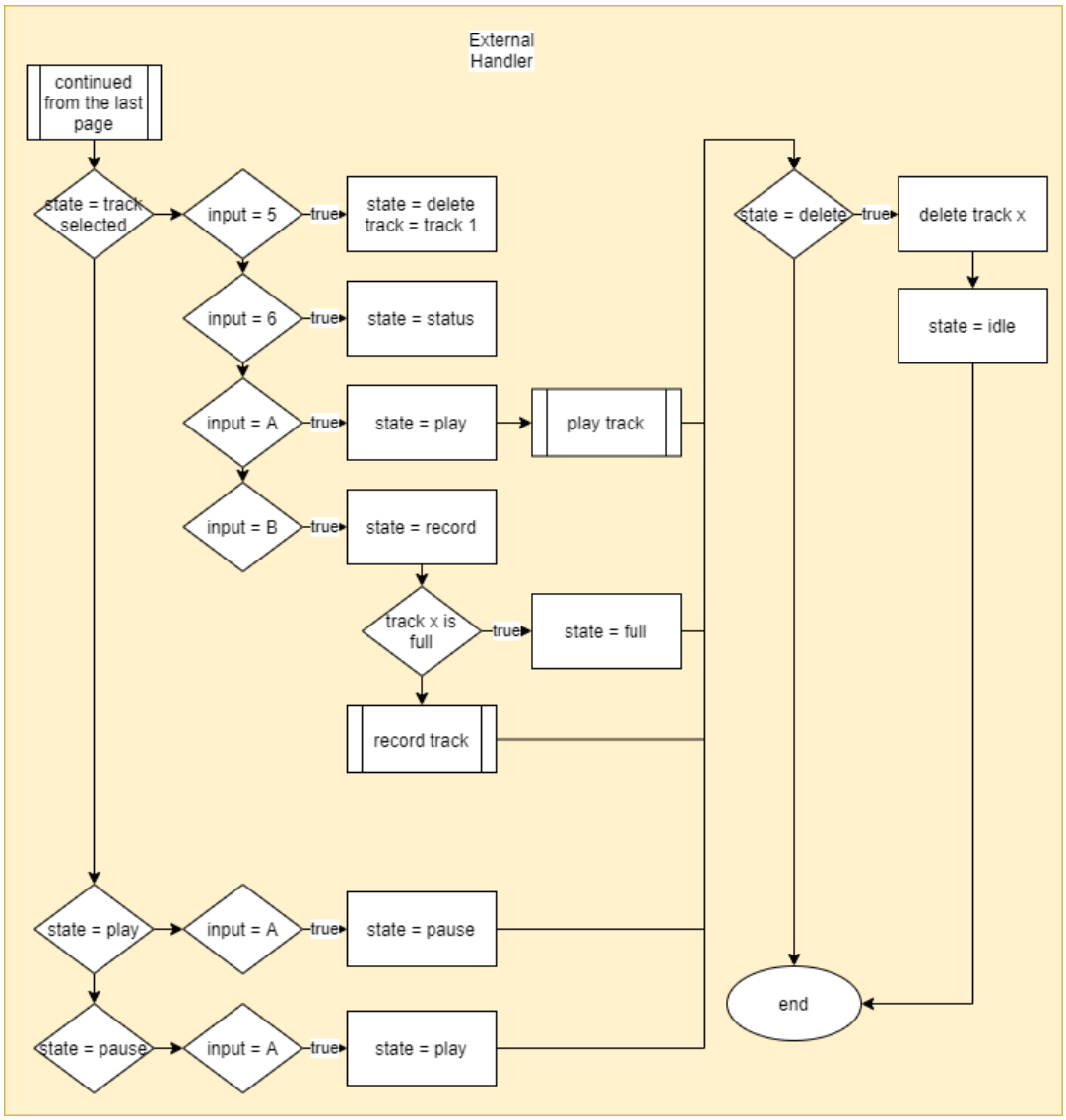

start
setup functions for ADC, PWM, TIM, SSD, Keypad and I2C
function to clear the EEPROM


Timer 1 Handler
start
reset pending status
print ssd


print ssd
state = start
true
print id to ssd
state = idle
true
print idle to ssd
state = full
true
print full to ssd
state = record
true
print rcd and number to ssd
state = play
true
print plb and number to ssd
state = status
true
print ava and number to ssd
end


ADC Handler
start
get data from ADC
state = play
true
get EEPROM data
state = record
true
write data to EEPROM
end


External Handler
start
state = start
input = none
false
state = idle
state = idle
input = 1
true
state = track_selected track = track 1
input = 2
true
state = track_selected track = track 2
input = 3
true
state = track_selected track = track 3
input = 4
true
state = track_selected track = track 4
input = 6
true
state = status
continued in next page



Now that the flowchart is created, we can start to write the code partially. Each part will be explained in detail on the code.

2.2 Setup of Timer1 and Handler Function

The timer handler function will be implemented as a refresh rate and reset time timer that will count to 10 and set the state to idle if no buttons are pressed.

```
void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
	TIM1->SR &= ~(1U << 0); /* reset status register */

	if(ms_ticks >= 10000) /* 10 seconds */
	{
		state = idle; /* go back to idle */
		clock_ticks = 0; /* reset clock ticks */
		ms_ticks = 0; /* reset millisecond ticks */
	}
```

```
	else
	{
		clock_ticks += 3200;
		ms_ticks = clock_ticks / (16000);
	}
	ssd_print();
}
```

The timer function is set to refresh around 10000 because the other functionalities are short and when the print function doesn't run, the seven segment display because of its functionality that only allows it to show a single number, will show the last digit brighter than the others.

```
/* timer1 and interrupt setup */

	RCC->APBENR2 |= (1U << 11); /* Enable TIM1 clock */

	TIM1->CR1 = 0; /* resetting control register */
	TIM1->CNT = 0; /* reset the timer counter */

	TIM1->PSC = ((SystemCoreClock/RefreshRate) - 1); /* prescaler set to 1600-1
*/
	TIM1->ARR = 1; /* set the autoreload register for 1 milliseconds */

	TIM1->DIER |= (1U << 0); /* update interrupt enable */
	TIM1->CR1 |= (1U << 0); /* Enable TIM1 */

	NVIC_SetPriority(TIM1_BRK_UP_TRG_COM_IRQn, 0U); /* Setting priority for
TIM1 */
	NVIC_EnableIRQ(TIM1_BRK_UP_TRG_COM_IRQn); /* Enabling TIM1 */
```

The timer 1 setup is shown above. The timers prescaler and auto reload register is chosen so that the refresh rate is 10000.

$$refresh\ rate = \frac{clock\ speed}{prescaler * (autoreload\ register + 1)}$$

$$autoreload\ = 1\ then\ prescaler = 1600 - 1 = 1599$$

2.3 Writing the External Interrupt Handler Function and State Machine

First we setup the interrupt and define the external interrupt handler function.

```
/* interrupt setup */

    RCC->IOPENR |= (3U << 0);

    EXTI->RTSR1 |= (15U << 8); /* Rising edge selection */
    EXTI->EXTICR[2] &= ~(1U << 8*0); /* 1U to select A8 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*1); /* 1U to select A9 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*2); /* 1U to select A10 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*3); /* 1U to select A11 from mux */
    EXTI->IMR1 |= (15U << 8); /* interrupt mask register */

    NVIC_SetPriority(EXTI4_15_IRQn, 1U); /* Setting priority for EXTI0_1 */
    NVIC_EnableIRQ(EXTI4_15_IRQn); /* Enabling EXTI0_1 */
```

After the setup is done we can define and write the external handler function code including the state machine.

```
void EXTI4_15_IRQHandler(void)
{
      kp_get_input(); /* get the input and write it to global input */

      switch(state) /* state transitions */
      {
            case start: /* start state */
                  switch(kp_input)
                  {
                        case 'x':
                              /* do nothing */
                              break;
                        case '6':
                              state = show_status;
                              break;
                        default:
                              state = idle;
                              break;
                  }
            case idle: /* idle state */
                  switch(kp_input)
                  {
                        case '1':
                              state = track_selected;
                              track_selection = track_one;
                              break;
                        case '2':
                              state = track_selected;
                              track_selection = track_two;
                              break;
                        case '3':
                              state = track_selected;
                              track_selection = track_three;
                              break;
                        case '4':
                              state = track_selected;
```

```
                                track_selection = track_four;
                                break;
                        case '6':
                                state = show_status;
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case track_selected:
                switch(kp_input)
                {
                        case '5': /* delete */
                                state = delete;
                                track_selection = track_one;
                                break;
                        case '6': /* show status */
                                state = show_status;
                                break;
                        case 'A': /* play/pause */
                                if(track_is_full[track_selected])
                                {
                                        TIM17->BDTR |= (1U << 15); /* main output
enable for pwm */
                                        state = play;
                                }
                                else
                                {
                                        state = idle;
                                }
                                break;
                        case 'B': /* record */
                                if(track_is_full[track_selected])
                                {
                                        state = full;
                                }
                                else
                                {
                                        state = record;
                                        recording_state = recording;
                                }
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case play:
                switch(kp_input)
                {
                        case 'A': /* play/pause */
                                state = pause;
                                TIM17->BDTR &= ~(1U << 15); /* main output
disable for pwm */
                                break;
                        default:
                                /* do nothing */
                                break;
```

```
                }
                if(track_index >= MaxTrackIndex)
                {
                        track_index = 0;
                        state = idle;
                        TIM17->BDTR &= ~(1U << 15); /* main output disable for
pwm */
                        playing_state = playing_done;
                }
                break;
        case pause:
                switch(kp_input)
                {
                        case 'A': /* play/pause */
                                state = play;
                                TIM17->BDTR |= (1U << 15); /* main output enable
for pwm */
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case record:
                if(track_index >= MaxTrackIndex)
                {
                        track_index = 0;
                        state = idle;
                        recording_state = recording_done;
                }
                break;
        case delete:

                break;
        case full:

                break;
        case show_status:

                break;
    }

    switch(state) /* performs operations depending on the switched state */
    {
        case start:
                /* do nothing */
                break;
        case idle:
                /* do nothing */
                break;
        case track_selected:

                break;
        case play:

                break;
        case pause:

                break;
```

```
            case record:

                  break;
            case delete:

                  break;
            case full:
                  /* do nothing */
                  break;
            case show_status:

                  break;
      }

      for(volatile uint32_t i=0; i<333333; i++);

      EXTI->RPR1 |= (15U << 8); /* reset interrupt pending */
}
```

The External Interrupt Handler includes the state machine that transitions the machine depending on the current state and the input of the system.

State machine starts with start state and this state shows the first two and last two numbers of the student id which is 1837. After an input is entered or after 10 seconds the state will change to idle.

In idle state the seven-segment display will show idle text and will expect an input. Depending on the input the state will change to a track selection.

After a track is selected an enumerator will set the chosen track and state will switch to track selection. In this state no text will be written to the seven-segment display because no design rules were given on it. While in this state certain inputs describe in the state machine will switch the state to play, record, delete and status states.

In order for the board to switch to play state the selected track must be full with a recording otherwise the state will just switch back to idle. For the board to switch to record state the selected track must be empty otherwise the state will switch to full state and show a full text on seven segment display. The delete state does not have a requirement except the delete input which is the 5 button and the delete state will delete the previous selected track.

Lastly in show status state number of available tracks will be shown. The term available is confusing because the availability can mean available to record or available to play. I've understood it as available to play and the AvA and the number of available tracks are shown when this state is switched.

Extra details are given in the comments on the code above. Not that the state machine part is explained the main body of the handler performs operations according to the current state after the state machine. This part isn't complex and the main body performs simple tasks like track deletion which runs when the state is switched to delete.

2.4 Get Keypad Input Functions

Get input function works by having an external interrupt with input column pins and row output pins. Row pins start as high so that the column pins can trigger an interrupt. When column pins go into the handler, we can check the rows by turning on and off the row outputs and checking the column inputs.

```
void kp_get_input(void)
{
	uint8_t c = 0;
	uint8_t r = 0;

	static const char kp_inputs[] =
	{'1', '2', '3', 'A',
	 '4', '5', '6', 'B',
	 '7', '8', '9', 'C',
	 'E', '0', 'F', 'D'};

	if((EXTI->RPR1 >> 8) & (1U))
	{
		c = 0;
	}
	else if ((EXTI->RPR1 >> 9) & (1U))
	{
		c = 1;
	}
	else if ((EXTI->RPR1 >> 10) & (1U))
	{
		c = 2;
	}
	else if ((EXTI->RPR1 >> 11) & (1U))
	{
		c = 3;
	}

	GPIOA->ODR &= ~(1U << 0); /* shut off A0 */
	for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 0;
	}
	GPIOA->ODR |= (1U << 0); /* turn on A0 */

	GPIOA->ODR &= ~(1U << 1); /* shut off A1 */
	for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 1;
	}
	GPIOA->ODR |= (1U << 1); /* turn on A1 */

	kp_input = kp_inputs[(c + (4 * r))]; /* constructing the input with the row
and column values */
}
```

Keypad get input function is explained in detail on code comments.

2.5 Seven Segment Display Print and Set Functions

Seven segment display set function works by getting an input as a digit and it prints that digit by editing the GPIOB ODR register.

```
void ssd_set(volatile uint32_t digit)
{
	switch(digit)
	{
		case 0:
			GPIOB->ODR &= ~(0x3FU);
			break;
		case 1:
			GPIOB->ODR &= ~(0x6U);
			break;
		case 2:
			GPIOB->ODR &= ~(0x5BU);
			break;
		case 3:
			GPIOB->ODR &= ~(0x4FU);
			break;
		case 4:
			GPIOB->ODR &= ~(0x66U);
			break;
		case 5:
			GPIOB->ODR &= ~(0x6DU);
			break;
		case 6:
			GPIOB->ODR &= ~(0x7DU);
			break;
		case 7:
			GPIOB->ODR &= ~(0x7U);
			break;
		case 8:
			GPIOB->ODR &= ~(0x7FU);
			break;
		case 9:
			GPIOB->ODR &= ~(0x6FU);
			break;
	}
}
```

Set seven segment display is a simple function that sets the GPIOB ODR according to the SSD truth tables.

Same as above the seven segment display set digit function works by setting the GPIOB ODR according to the correct digit selection on seven segment display.

```
void ssd_digit(volatile uint32_t digit) /* 0 = Digit1 */
{
	ssd_digit_clear();
	ssd_set_clear();

	GPIOA->ODR |= (1U << (4 + digit));
}
```

Because the digits start at 0 like arrays we can use that information and the fact that the pins are in series (A4, A5, A6, A7).

Seven segment display print function works by checking the state of the board and printing the according to the given design rules. Set id function prints the id, set idle function prints idle, set full function prints full, set record function prints rcd and the recording time, set play function prints the plb and the number of the track being played and the set status function prints AvA and the amount of available tracks.

```
void ssd_set_id(void)
{
	ssd_digit(0);
	ssd_set(1); /* 1 */
	ssd_digit(1);
	ssd_set(8); /* 8 */
	ssd_digit(2);
	ssd_set(3); /* 3 */
	ssd_digit(3);
	ssd_set(7); /* 7 */
	ssd_digit_clear();
}

void ssd_set_idle(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x30U); /* I */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x5EU); /* d */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit(3);
	GPIOB->ODR &= ~(0x79U); /* E */
	ssd_digit_clear();
}

void ssd_set_full(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x71U); /* F */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x1CU); /* u */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit(3);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit_clear();
}

void ssd_set_record(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x50U); /* r */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x39U); /* c */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x5EU); /* d */
	ssd_digit(3);
```

```
	ssd_set(((5000 - (ms_ticks + 1)) / 1000) + 1); /* timer calculation */
	ssd_digit_clear();
}

void ssd_set_play(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x73U); /* P */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x7CU); /* b */
	ssd_digit(3);
	ssd_set(1U + track_selection); /* track enum */
	ssd_digit_clear();
}

void ssd_set_status(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x77U); /* A */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x1CU); /* v */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x77U); /* A */
	ssd_digit(3);
	ssd_set(4U - empty_tracks); /* show empty tracks */
	ssd_digit_clear();
}

/* print and refresh function */

void ssd_print(void)
{
	switch(state)
	{
		case start: /* print id */
			ssd_set_id();
			break;
		case idle:
			ssd_set_idle();
			break;
		case full:
			ssd_set_full();
			break;
		case record:
			ssd_set_record();
			break;
		case play:
			ssd_set_play();
			break;
		case show_status:
			ssd_set_status();
			break;
		default:
			ssd_digit_clear();
			ssd_set_clear();
	}
}
```

The main print function is simple and it only checks the current state and prints a format accordingly. The formatted print functions are explained above this part.

2.6 Timer3 Trigger Setup and ADC Handler

The Timer3 is used to trigger the ADC which in turn goes into the ADC handler. with that in mind we can set up the timer 3 for TRG3 and set its frequency to 10kHz for the sample rate of ADC.

```
/* timer3 setup */

	RCC->APBENR1 |= (1U << 1); /* Enable TIM3 clock */

	TIM3->CR1 = 0; /* resetting control register */
	TIM3->CNT = 0; /* reset the timer counter */

	TIM3->PSC = ((SystemCoreClock/RefreshRate) - 1); /* prescaler set to 1600-1
*/
	TIM3->ARR = 1; /* set the autoreload register for 1 milliseconds */

	TIM3->CR2 |= (2U << 4); /* master mode to update for trigger */

	TIM3->CR1 |= (1U << 0); /* Enable TIM3 */
```

As seen above the Timer 3 is only used to trigger the ADC Handler and no Timer interrupt is enabled. Auto reload and prescaler values are chosen so that the frequency of the interrupts will be 10kHz. The MMS (master mode selection) register is changed to update mode for trigger functionality.

After the trigger is set up we can setup the ADC and the ADC Handler to get the values.

```
/* ADC setup */

	RCC->APBENR2 |= (1U << 20); /* ADC clock enable */

	RCC->APBRSTR2 |= (1U << 20); /* reset adc */
	RCC->APBRSTR2 &= ~(1U << 20);

	ADC1->CR |= (1U << 28); /* open voltage reg */
	for(volatile int i = 0; i<16000; i++);

	ADC1->CR |= (1U << 31); /* ADC calibration */
	while((ADC1->CR >> 31) & (0x1U)); /* wait for end of calibration */

	ADC1->CFGR1 |= (2U << 3); /* 8 bit resolution */

	ADC1->CFGR1 &= ~(1U << 13); /* single conversion mode */

	ADC1->SMPR &= ~(1U << 0);

	ADC1->CFGR1 &= ~(7U << 6); /* trigger selection trg3 */
	ADC1->CFGR1 |= (3U << 6);
	ADC1->CFGR1 |= (1U << 10); /* rising edge external trigger */
```

```
	ADC1->CHSELR |= (1U << 16); /* PA12 = ADC1/16 channel selection */

	GPIOA->MODER |= (3U << 2*12);

	ADC1->IER |= (1U << 3); /* end of conversion interrupt enable */

	ADC1->CR |= (1U << 0); /* adc enable */
	while(!((ADC1->ISR) & (0x1U)));

	NVIC_SetPriority(ADC1_IRQn, 1U); /* Setting priority for ADC handler */
	NVIC_EnableIRQ(ADC1_IRQn); /* Enabling ADC handler */

	ADC1->CR |= (1U << 2); /* start conversion */
```

In the ADC setup we chose TRG3 from configuration register and the channel from channel selection register as 16 which is PA12. Lastly we can setup the ADC handler which will run once every time trigger is created by the timer 3.

```
void ADC_COMP_IRQHandler(void)
{
	uint8_t analog_data = (uint8_t) (ADC1->DR);

	switch(state)
	{
		case play: /* play data using the ADC data */
			if((track_index + 1) < MaxTrackIndex)
			{
				TIM17->CCR1 = track_1[track_index]; /* set the new duty
cycle for PWM */
				track_index++;
			}
			else
			{
				/* do nothing */
			}
			break;
		case record: /* record data using the ADC data */
			if((track_index + 1) < MaxTrackIndex)
			{
				track_1[track_index] = analog_data;
				track_index++;
			}
			else
			{
				/* do nothing */
			}
			break;
		default:
			/* do nothing */
			break;
	}

	if(((state == record)) && ((((ms_ticks + 1) / 1000) + 1) > 5))
	{
		state = idle; /* switch back to idle is more than 5 seconds passed
*/
```

```
			track_index = 0;
			recording_state = recording_done;
			track_is_full[track_selection] = 1; /* mark the track as full */
		}

		if(((state == play)) && ((((ms_ticks + 1) / 1000) + 1) > 5))
		{
			state = idle; /* switch back to idle is more than 5 seconds passed
*/
			track_index = 0;
			playing_state = playing_done;
			TIM17->BDTR &= ~(1U << 15); /* main output disable for pwm */
		}

		ADC1->ISR |= (1U << 3); /* reset pending */
}
```

ADC handler is used in a multipurpose way to record and play the tracks. This is done inside the ADC handler so that the operation is in sync with the ADC conversion and won't be offset by the time that the conversion took place and the data was used.

2.7 Combined Code

After all the functions are written we can combine all the code and debug it using the STM32 debugger.

```
/* PROJECT3.C               */
/* Author: Ömer Emre Polat */
/* Student No: 1801022037  */

#include "bsp.h"
#include "stdlib.h"

int main(void) {

	track_index = 0; /* read write index for array */
	kp_input = 'x'; /* no input */
	state = start; /* start state */
	track_is_full[0] = 0;
	track_is_full[1] = 0;
	track_is_full[2] = 0;
	track_is_full[3] = 0;
	empty_tracks = 4;
	track_selection = track_one;
	BSP_system_init();

	while(1)
    {

    }

    return 0;
}
```

```c
/* BSP.H                    */
/* Author: Ömer Emre Polat */
/* Student No: 1801022037  */

#ifndef BSP_H_
#define BSP_H_

#include "stm32g031xx.h"

#define RefreshRate 10000
#define ButtonBounceTime 400
#define MaxTrackIndex 5

uint8_t track_is_full[4];
uint8_t empty_tracks;

uint32_t track_index;
uint8_t track_1[5];

uint32_t clock_ticks;
uint32_t ms_ticks;
char kp_input;

typedef enum
{
	start = 0, idle = 1, track_selected = 2, play = 3, pause = 4, record = 5,
delete = 6, full = 7, show_status = 8
}status;

typedef enum
{
	track_one = 0, track_two = 1, track_three = 2, track_four = 3
}track_selector;

typedef enum
{
	recording = 0, recording_done = 1
}recording_states;

typedef enum
{
	playing = 0, playing_done = 1
}play_states;

status state;

track_selector track_selection;

recording_states recording_state;

play_states playing_state;

void BSP_system_init(void);

void BSP_IWDG_init(void);
void BSP_IWDG_refresh(void);
```

```
void ssd_print(void);

void ssd_set(volatile uint32_t);
void ssd_set_operation(volatile char);
void ssd_set_id(void);
void ssd_set_idle(void);
void ssd_set_full(void);
void ssd_set_record(void);
void ssd_set_play(void);
void ssd_set_status(void);
void ssd_set_clear(void);

void ssd_digit(volatile uint32_t);
void ssd_digit_clear(void);

void kp_get_input(void);

#endif
```

```
/* BSP.C                    */
/* Author: Ömer Emre Polat */
/* Student No: 1801022037  */

#include "bsp.h"

void ADC_COMP_IRQHandler(void)
{
	uint8_t analog_data = (uint8_t) (ADC1->DR);

	switch(state)
	{
		case play: /* play data using the ADC data */
			if((track_index + 1) < MaxTrackIndex)
			{
				TIM17->CCR1 = track_1[track_index]; /* set the new duty
cycle for PWM */
				track_index++;
			}
			else
			{
				/* do nothing */
			}
			break;
		case record: /* record data using the ADC data */
			if((track_index + 1) < MaxTrackIndex)
			{
				track_1[track_index] = analog_data;
				track_index++;
			}
			else
			{
				/* do nothing */
			}
			break;
		default:
			/* do nothing */
```

```
                        break;
        }

        if(((state == record)) && ((((ms_ticks + 1) / 1000) + 1) > 5))
        {
                state = idle; /* switch back to idle is more than 5 seconds passed
*/
                track_index = 0;
                recording_state = recording_done;
                track_is_full[track_selection] = 1; /* mark the track as full */
        }

        if(((state == play)) && ((((ms_ticks + 1) / 1000) + 1) > 5))
        {
                state = idle; /* switch back to idle is more than 5 seconds passed
*/
                track_index = 0;
                recording_state = recording_done;
        }

        ADC1->ISR |= (1U << 3); /* reset pending */
}

void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
        TIM1->SR &= ~(1U << 0); /* reset status register */

        if(ms_ticks >= 10000) /* 10 seconds */
        {
                state = idle; /* go back to idle */
                clock_ticks = 0; /* reset clock ticks */
                ms_ticks = 0; /* reset millisecond ticks */
        }
        else
        {
                clock_ticks += 3200;
                ms_ticks = (clock_ticks + 1) / (16000);
        }

        ssd_print();
}

/* PA8-PA11 */
void EXTI4_15_IRQHandler(void)
{
        kp_get_input(); /* get the input and write it to global input */

        clock_ticks = 0; /* reset clock ticks */
        ms_ticks = 0; /* reset millisecond ticks */

        switch(state) /* state transitions */
        {
                case start: /* start state */
                        switch(kp_input)
                        {
                                case 'x':
                                        /* do nothing */
                                        break;
                                case '6':
```

```
                                state = show_status;
                                break;
                        default:
                                state = idle;
                                break;
                }
                break;
        case idle: /* idle state */
                switch(kp_input)
                {
                        case '1':
                                state = track_selected;
                                track_selection = track_one;
                                break;
                        case '2':
                                state = track_selected;
                                track_selection = track_two;
                                break;
                        case '3':
                                state = track_selected;
                                track_selection = track_three;
                                break;
                        case '4':
                                state = track_selected;
                                track_selection = track_four;
                                break;
                        case '6':
                                state = show_status;
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case track_selected:
                switch(kp_input)
                {
                        case '5': /* delete */
                                state = delete;
                                track_selection = track_one;
                                break;
                        case '6': /* show status */
                                state = show_status;
                                break;
                        case 'A': /* play/pause */
                                if(track_is_full[track_selection])
                                {
                                        TIM17->BDTR |= (1U << 15); /* main output
enable for pwm */
                                        state = play;
                                        playing_state = playing;
                                        track_index = 0;
                                }
                                else
                                {
                                        state = idle;
                                }
                                break;
                        case 'B': /* record */
```

```
                                if(track_is_full[track_selection])
                                {
                                        state = full;
                                }
                                else
                                {
                                        state = record;
                                        recording_state = recording;
                                        track_is_full[track_selection] = 1;
                                        empty_tracks--;
                                        track_index = 0;
                                }
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case play:
                switch(kp_input)
                {
                        case 'A': /* play/pause */
                                state = pause;
                                TIM17->BDTR &= ~(1U << 15); /* main output
disable for pwm */
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                if((track_index + 1) >= MaxTrackIndex)
                {
                        track_index = 0;
                        state = idle;
                        TIM17->BDTR &= ~(1U << 15); /* main output disable for
pwm */
                        playing_state = playing_done;
                }
                break;
        case pause:
                switch(kp_input)
                {
                        case 'A': /* play/pause */
                                state = play;
                                TIM17->BDTR |= (1U << 15); /* main output enable
for pwm */
                                break;
                        default:
                                /* do nothing */
                                break;
                }
                break;
        case record:
                if((track_index + 1) >= MaxTrackIndex)
                {
                        track_index = 0;
                        state = idle;
                        recording_state = recording_done;
                }
```

```
                        break;
                case delete:

                        break;
                case full:

                        break;
                case show_status:

                        break;
        }

        switch(state) /* performs operations depending on the switched state */
        {
                case start:
                        /* do nothing */
                        break;
                case idle:
                        /* do nothing */
                        break;
                case track_selected:

                        break;
                case play:

                        break;
                case pause:

                        break;
                case record:

                        break;
                case delete:
                        track_is_full[track_selection] = 0;
                        empty_tracks++;
                        break;
                case full:
                        /* do nothing */
                        break;
                case show_status:

                        break;
        }

        for(volatile uint32_t i=0; i<333333; i++);

        EXTI->RPR1 |= (15U << 8); /* reset interrupt pending */
}

/*//////////////KEYPAD FUNCTIONS//////////////*/

void kp_get_input(void)
{
        uint8_t c = 0;
        uint8_t r = 0;

        static const char kp_inputs[] =
        {'1', '2', '3', 'A',
         '4', '5', '6', 'B',
```

```
        '7', '8', '9', 'C',
        'E', '0', 'F', 'D'};

      if((EXTI->RPR1 >> 8) & (1U))
      {
            c = 0;
      }
      else if ((EXTI->RPR1 >> 9) & (1U))
      {
            c = 1;
      }
      else if ((EXTI->RPR1 >> 10) & (1U))
      {
            c = 2;
      }
      else if ((EXTI->RPR1 >> 11) & (1U))
      {
            c = 3;
      }

      GPIOA->ODR &= ~(1U << 0); /* shut off A0 */
      for(uint32_t i=0; i<ButtonBounceTime; i++);
      if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
      {
            r = 0;
      }
      GPIOA->ODR |= (1U << 0); /* turn on A0 */

      GPIOA->ODR &= ~(1U << 1); /* shut off A1 */
      for(uint32_t i=0; i<ButtonBounceTime; i++);
      if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
      {
            r = 1;
      }
      GPIOA->ODR |= (1U << 1); /* turn on A1 */

      kp_input = kp_inputs[(c + (4 * r))]; /* constructing the input with the row
and column values */
}

/*///////////////BSP FUNCTIONS///////////////*/

void BSP_IWDG_init(void)
{
    IWDG->KR = 0x5555;
    IWDG->PR = 1; // prescaler
    while(IWDG->SR & 0x1); // wait while status update
    IWDG->KR = 0xCCCC;
}

void BSP_IWDG_refresh(void)
{
    IWDG->KR = 0xAAAA;
}

void BSP_system_init()
{
    __disable_irq();
```

```
    /* input and output pins setup */

    GPIOB->MODER |= (0x1555U); /* Setup (B0, B6) as output */
    GPIOB->MODER &= ~(0x2AAAU); /* Outputs for seven segments */

    GPIOA->MODER |= (0x55U << 2*4); /* Setup (A4, A7) as output */
    GPIOA->MODER &= ~(0xAAU << 2*4); /* Outputs for digit selections */

    GPIOA->MODER &= ~(255U << 2*8); /* Setup (A8, A11) as input */

    GPIOA->PUPDR &= ~(3U << 2*8); /* A8 Pull down */
    GPIOA->PUPDR |= (2U << 2*8);

    GPIOA->PUPDR &= ~(3U << 2*9); /* A9 Pull down */
    GPIOA->PUPDR |= (2U << 2*9);

    GPIOA->PUPDR &= ~(3U << 2*10); /* A10 Pull down */
    GPIOA->PUPDR |= (2U << 2*10);

    GPIOA->PUPDR &= ~(3U << 2*11); /* A11 Pull down */
    GPIOA->PUPDR |= (2U << 2*11);

    GPIOA->MODER &= ~(3U << 2*0); /* A0 output */
    GPIOA->MODER |= (1U << 2*0);
    GPIOA->ODR |= (1U << 0); /* set A0 to high */

    GPIOA->MODER &= ~(3U << 2*1); /* A1 output */
    GPIOA->MODER |= (1U << 2*1);
    GPIOA->ODR |= (1U << 1); /* set A1 to high */

//    GPIOA->MODER &= ~(3U << 2*12); /* A12 output */
//    GPIOA->MODER |= (1U << 2*12);
//    GPIOA->ODR |= (1U << 12); /* set A12 to high */
//
//    GPIOA->MODER &= ~(3U << 2*15); /* A15 output */
//    GPIOA->MODER |= (1U << 2*15);
//    GPIOA->ODR |= (1U << 15); /* set A15 to high */

    /* timer1 and interrupt setup */

      RCC->APBENR2 |= (1U << 11); /* Enable TIM1 clock */

      TIM1->CR1 = 0; /* resetting control register */
      TIM1->CNT = 0; /* reset the timer counter */

      TIM1->PSC = ((SystemCoreClock/RefreshRate) - 1); /* prescaler set to 1600-1
*/
      TIM1->ARR = 1; /* set the autoreload register for 1 milliseconds */

      TIM1->DIER |= (1U << 0); /* update interrupt enable */
      TIM1->CR1 |= (1U << 0); /* Enable TIM1 */

      NVIC_SetPriority(TIM1_BRK_UP_TRG_COM_IRQn, 0U); /* Setting priority for
TIM1 */
      NVIC_EnableIRQ(TIM1_BRK_UP_TRG_COM_IRQn); /* Enabling TIM1 */

      /* timer3 setup */

      RCC->APBENR1 |= (1U << 1); /* Enable TIM3 clock */
```

```
      TIM3->CR1 = 0; /* resetting control register */
      TIM3->CNT = 0; /* reset the timer counter */

      TIM3->PSC = ((SystemCoreClock/RefreshRate) - 1); /* prescaler set to 1600-1
*/
      TIM3->ARR = 1; /* set the autoreload register for 1 milliseconds */

      TIM3->CR2 |= (2U << 4); /* master mode to update for trigger */

      TIM3->CR1 |= (1U << 0); /* Enable TIM3 */

      /* ADC setup */

      RCC->APBENR2 |= (1U << 20); /* ADC clock enable */

      RCC->APBRSTR2 |= (1U << 20); /* reset adc */
      RCC->APBRSTR2 &= ~(1U << 20);

      ADC1->CR |= (1U << 28); /* open voltage reg */
      for(volatile int i = 0; i<16000; i++);

      ADC1->CR |= (1U << 31); /* ADC calibration */
      while((ADC1->CR >> 31) & (0x1U)); /* wait for end of calibration */

      ADC1->CFGR1 |= (2U << 3); /* 8 bit resolution */

      ADC1->CFGR1 &= ~(1U << 13); /* single conversion mode */

      ADC1->SMPR &= ~(1U << 0);

      ADC1->CFGR1 &= ~(7U << 6); /* trigger selection trg3 */
      ADC1->CFGR1 |= (3U << 6);
      ADC1->CFGR1 |= (1U << 10); /* rising edge external trigger */

      ADC1->CHSELR |= (1U << 16); /* PA12 = ADC1/16 channel selection */

      GPIOA->MODER |= (3U << 2*12);

      ADC1->IER |= (1U << 3); /* end of conversion interrupt enable */

      ADC1->CR |= (1U << 0); /* adc enable */
      while(!((ADC1->ISR) & (0x1U)));

      NVIC_SetPriority(ADC1_IRQn, 1U); /* Setting priority for ADC handler */
      NVIC_EnableIRQ(ADC1_IRQn); /* Enabling ADC handler */

      ADC1->CR |= (1U << 2); /* start conversion */

    /* pwm timer 17 AF2 setup */

      GPIOB->MODER &= ~(3U << 2*9); /* PB9 as alternate function */
      GPIOB->MODER |= (2U << 2*9);

      GPIOB->AFR[1] &= ~(0xFU << 4*1); /* choose AF2 from mux */
      GPIOB->AFR[1] |= (0x2U << 4*1);

      RCC->APBENR2 |= (1U << 18); /* Enable TIM17 clock */
```

```
	TIM17->CR1 = 0U; /* resetting control register */
	TIM17->CR1 |= (1U << 7); /* ARPE buffering */
	TIM17->CNT = 0U; /* reset the timer counter */

	TIM17->PSC = (5U); /* prescaler set to 5 */
	TIM17->ARR = 255U; /* set the autoreload register for 8 bit res */

	TIM17->CR1 |= (1U << 0); /* Enable TIM17 */

	TIM17->CCMR1 |= (6U << 4);

	TIM17->CCMR1 |= (1U << 3);

	TIM17->CCER |= (1U << 0);

	TIM17->CCR1 |= (1U << 0);

    /* interrupt setup */

    RCC->IOPENR |= (3U << 0);

    EXTI->RTSR1 |= (15U << 8); /* Rising edge selection */
    EXTI->EXTICR[2] &= ~(1U << 8*0); /* 1U to select A8 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*1); /* 1U to select A9 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*2); /* 1U to select A10 from mux */
    EXTI->EXTICR[2] &= ~(1U << 8*3); /* 1U to select A11 from mux */
    EXTI->IMR1 |= (15U << 8); /* interrupt mask register */

    NVIC_SetPriority(EXTI4_15_IRQn, 1U); /* Setting priority for EXTI0_1 */
    NVIC_EnableIRQ(EXTI4_15_IRQn); /* Enabling EXTI0_1 */

//    BSP_IWDG_init(); /* Watchdog init */

    TIM1->SR  =  0x00000000U; /* reset status register */

    EXTI->RPR1 = 0x00000000U; /* reset interrupt pending */

    __enable_irq();
}

/*///////////////SSD FUNCTIONS///////////////*/

void ssd_set_clear(void)
{
	GPIOB->ODR |= (0xFFU);
}

void ssd_set(volatile uint32_t digit)
{
	switch(digit)
	{
		case 0:
			GPIOB->ODR &= ~(0x3FU);
			break;
		case 1:
			GPIOB->ODR &= ~(0x6U);
			break;
		case 2:
			GPIOB->ODR &= ~(0x5BU);
```

```
                break;
        case 3:
                GPIOB->ODR &= ~(0x4FU);
                break;
        case 4:
                GPIOB->ODR &= ~(0x66U);
                break;
        case 5:
                GPIOB->ODR &= ~(0x6DU);
                break;
        case 6:
                GPIOB->ODR &= ~(0x7DU);
                break;
        case 7:
                GPIOB->ODR &= ~(0x7U);
                break;
        case 8:
                GPIOB->ODR &= ~(0x7FU);
                break;
        case 9:
                GPIOB->ODR &= ~(0x6FU);
                break;
    }
}

void ssd_set_id(void)
{
    ssd_digit(0);
    ssd_set(1); /* 1 */
    ssd_digit(1);
    ssd_set(8); /* 8 */
    ssd_digit(2);
    ssd_set(3); /* 3 */
    ssd_digit(3);
    ssd_set(7); /* 7 */
    ssd_digit_clear();
}

void ssd_set_idle(void)
{
    ssd_digit(0);
    GPIOB->ODR &= ~(0x30U); /* I */
    ssd_digit(1);
    GPIOB->ODR &= ~(0x5EU); /* d */
    ssd_digit(2);
    GPIOB->ODR &= ~(0x38U); /* L */
    ssd_digit(3);
    GPIOB->ODR &= ~(0x79U); /* E */
    ssd_digit_clear();
}

void ssd_set_full(void)
{
    ssd_digit(0);
    GPIOB->ODR &= ~(0x71U); /* F */
    ssd_digit(1);
    GPIOB->ODR &= ~(0x1CU); /* u */
    ssd_digit(2);
    GPIOB->ODR &= ~(0x38U); /* L */
```

```
	ssd_digit(3);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit_clear();
}

void ssd_set_record(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x50U); /* r */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x39U); /* c */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x5EU); /* d */
	ssd_digit(3);
	ssd_set(((5000 - (ms_ticks + 1)) / 1000) + 1); /* timer calculation */
	ssd_digit_clear();
}

void ssd_set_play(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x73U); /* P */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x38U); /* L */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x7CU); /* b */
	ssd_digit(3);
	ssd_set(1U + track_selection); /* track enum */
	ssd_digit_clear();
}

void ssd_set_status(void)
{
	ssd_digit(0);
	GPIOB->ODR &= ~(0x77U); /* A */
	ssd_digit(1);
	GPIOB->ODR &= ~(0x1CU); /* v */
	ssd_digit(2);
	GPIOB->ODR &= ~(0x77U); /* A */
	ssd_digit(3);
	ssd_set(4U - empty_tracks); /* show empty tracks */
	ssd_digit_clear();
}

void ssd_digit_clear(void)
{
	GPIOA->ODR &= ~(15U << 4); /* digit clear */
}

void ssd_digit(volatile uint32_t digit) /* 0 = Digit1 */
{
	ssd_digit_clear();
	ssd_set_clear();

	GPIOA->ODR |= (1U << (4 + digit));
}

/* print and refresh functions */
```

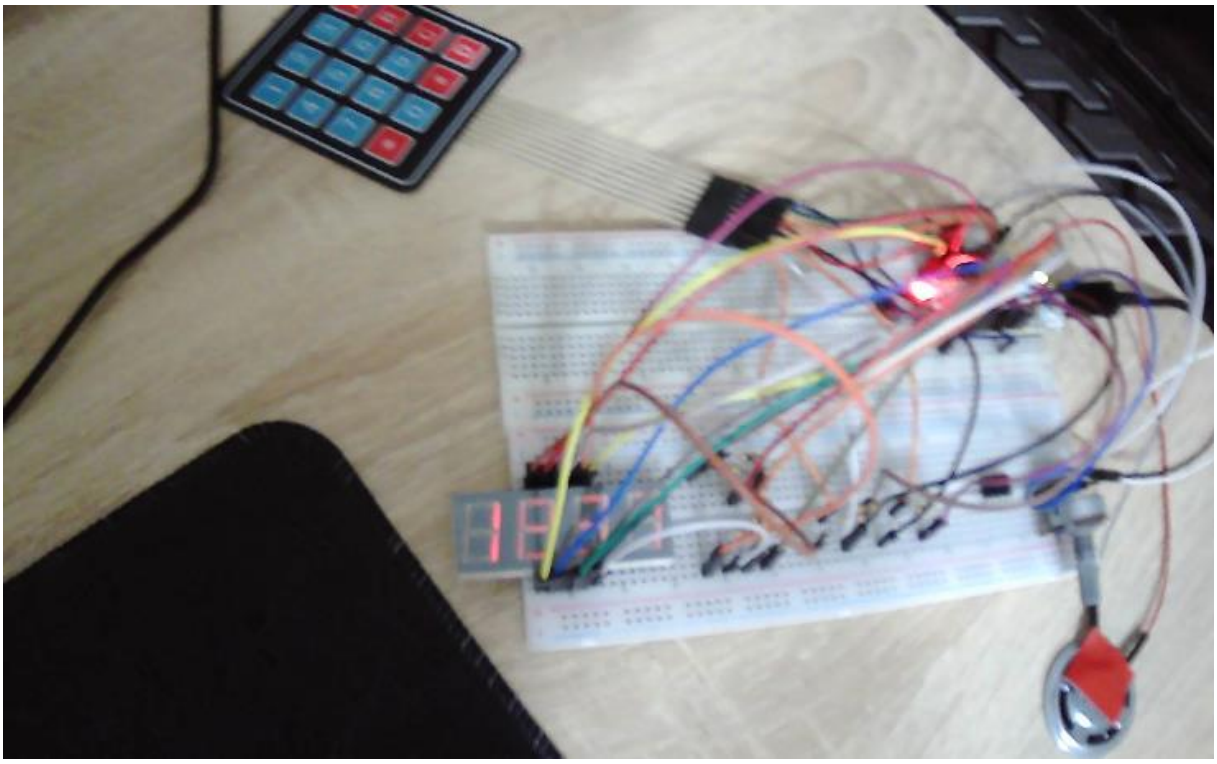
```
void ssd_print(void)
{
	switch(state)
	{
		case start: /* print id */
			ssd_set_id();
			break;
		case idle:
			ssd_set_idle();
			break;
		case full:
			ssd_set_full();
			break;
		case record:
			ssd_set_record();
			break;
		case play:
			ssd_set_play();
			break;
		case show_status:
			ssd_set_status();
			break;
		default:
			ssd_digit_clear();
			ssd_set_clear();
	}
}
```

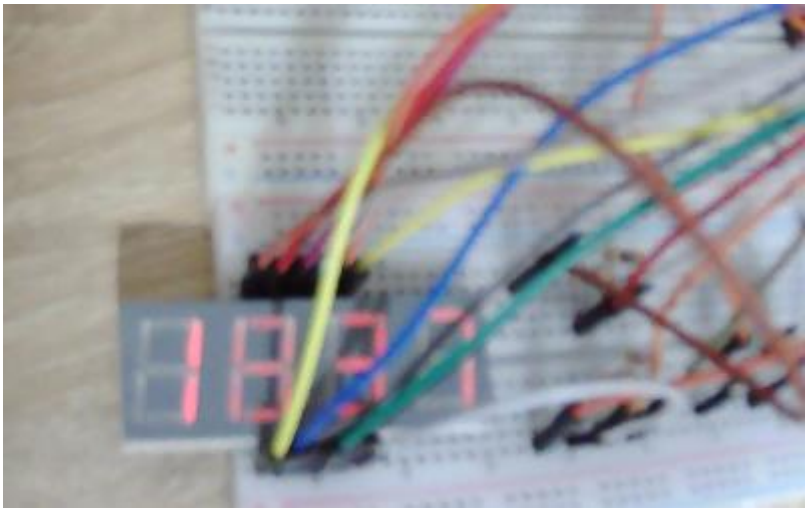
The code may be long and hard to read but analyzing the functions individually like it was done in the parts before the combined code is a lot easier.

The I2C setup was not done because the I2C setup or functions could not work and I couldn't store the values I got from ADC because of the I2C read and write fucntions.

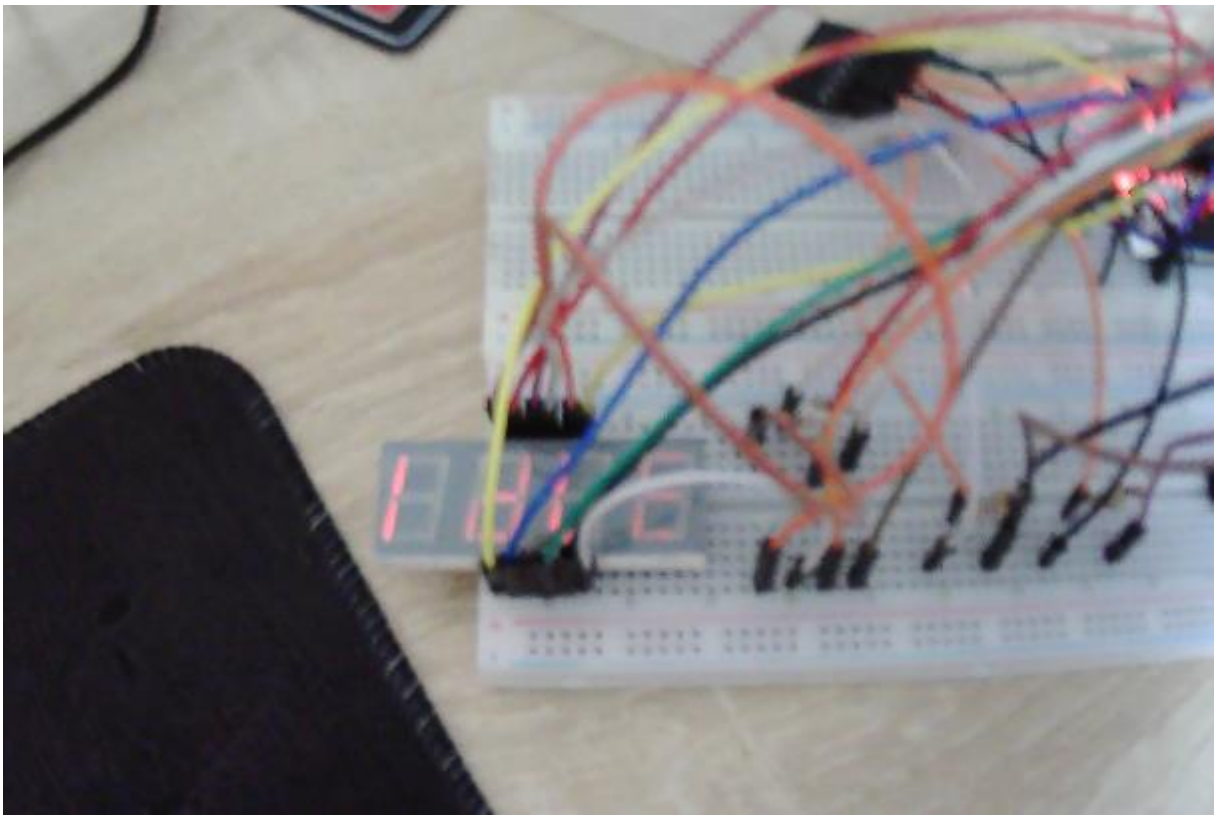
2.8 Testing of the Board with Peripherals

After the peripherals are connected we can test it on the peripherals. Code will be first run on debug mode and after deciding that it works correctly, we can test run it at full board speed.

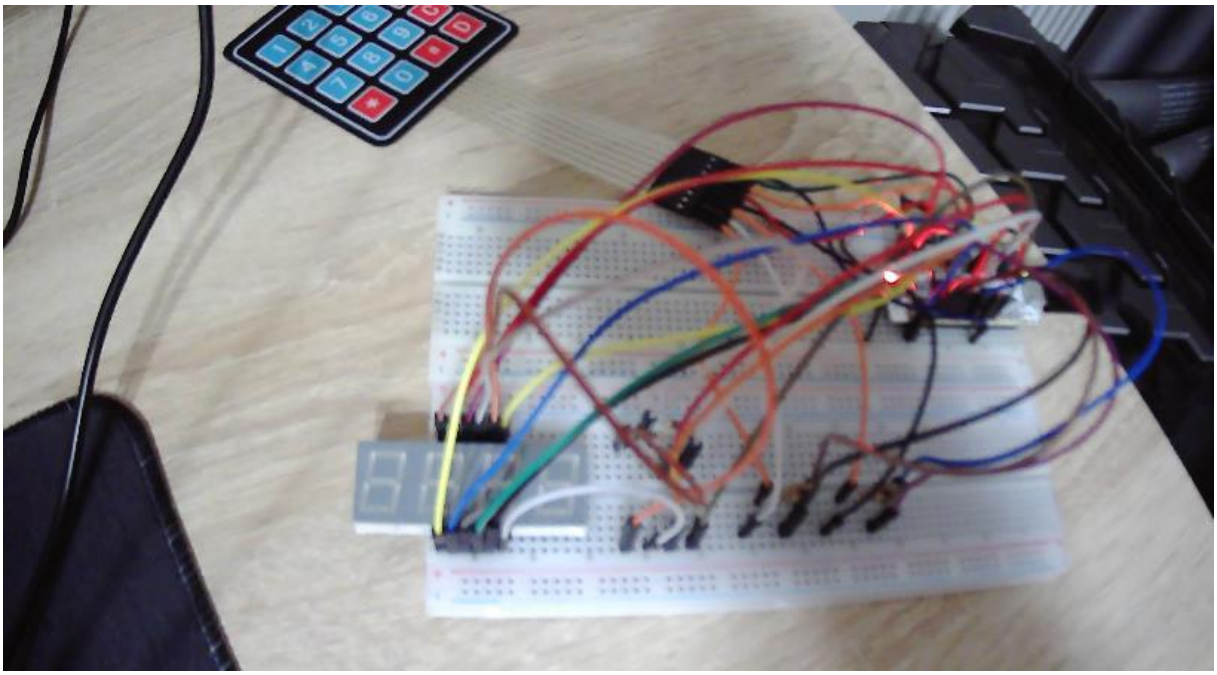




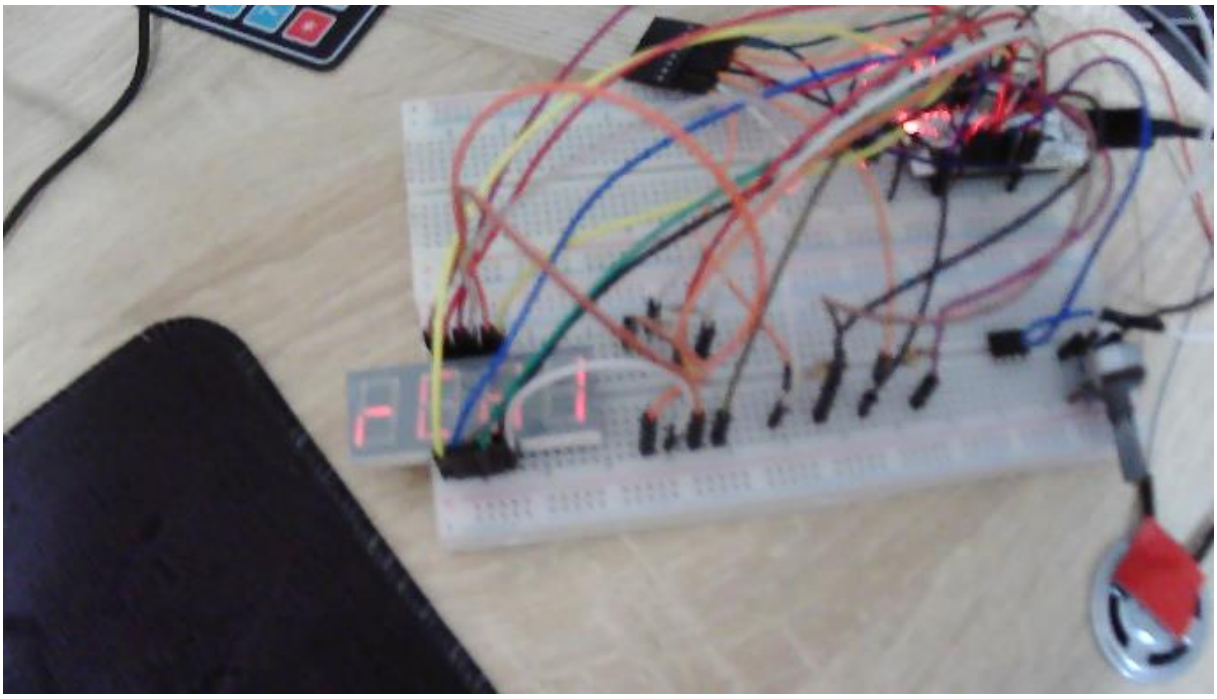
As expected the code writes the 1837 number (first two and last two digits of my school number) in idle state.



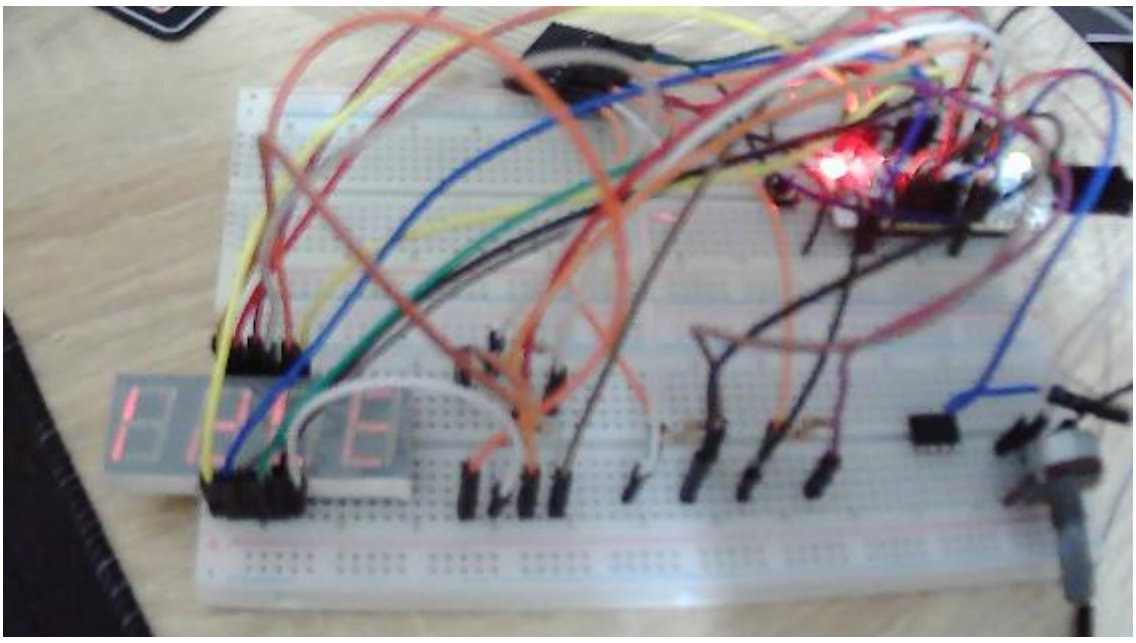
When we press a number or wait 10 second the board will automatically switch to idle state.



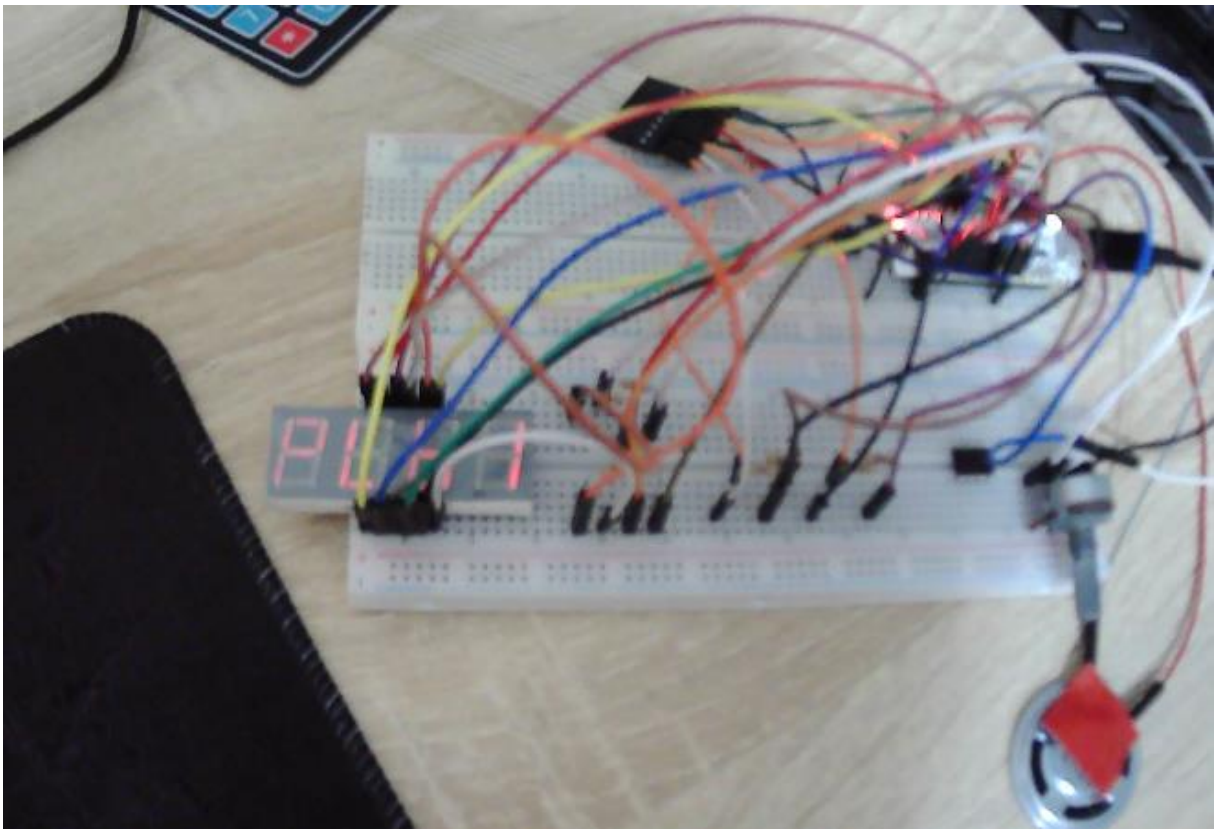
When a track selection button is pressed in this case 1, the ssd wont show anything because there were no design rules about it.



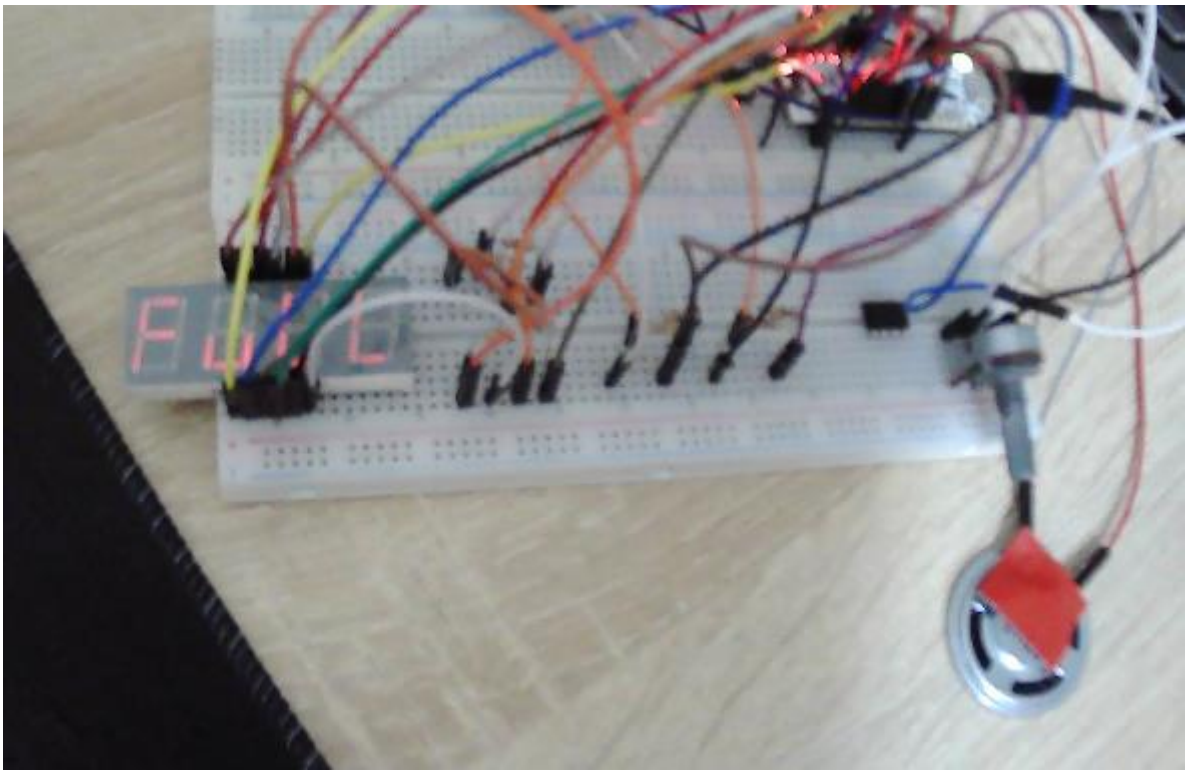
As the second input we enter record in this case B button the rcd will show up and count down from 5 to record the ADC input to storage. After the recording is done the board will return back to idle state.



Now when we press the same track selection in this case 1, and press the play button in this case A we can see the plb1 which indicates that the track 1 is being played back.



Upon entering the input B after selecting track 1 we can see the full on ssd.



2.9 Prices and Parts List

Parts that are used in the project can be written into the table down below.

	Part Name	Amount	Price
1	Breadboard	1	7.50TL
2	Jumper Cable (Male-Male)	40	40 piece is around 3.16TL
3	4x4 Keypad	1	5.41TL
4	470 ohm resistors	12	~
5	Seven Segment Display	1	7.99TL
6	MAX4466 Microphone Module	1	18.26TL
7	8R 0.5W Speaker	1	6.70TL
8	24LC512	2	9.82 x 2 TL
9	STM32G031K8T6 Board	1	102.50TL
			TOTAL
			171.16TL

Total price list comes to around 171.16TL excluding the cargo costs. Build takes around 5 to 10 minutes using the block diagram given at the beginning of the report.

2.10 Video and Documenting the Project

Video documentation of the project will be loaded into the youtube. The link is given below.

https://youtu.be/Nip0dnUNqao

3. Results and General Comments

The results were as expected. I've learned to design a code implement it to a board and build a project on top of it all. The code had a lot of problems because of the keypad get input is not working consistently.

As general comments there were a lot of debugging to see if thing were working properly or not. These debugging sessions could get confusing especially when the onboard written pin names are dramatically different than the software names of the pins. Lastly keypad get input function was not consistent so consistent but when the correct inputs were registered from the keypad the calculator worked as intended. Setup for I2C read and write was not working probably because i wasn't able to properly write the code for read and write functions for I2C.

4. References

[1]. RM0444 Reference manual
https://www.st.com/resource/en/reference_manual/dm00371828-stm32g0x1-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[2]. STM32G031K8 Datasheet

https://www.st.com/en/microcontrollers-microprocessors/stm32g031k8.html