GEBZE TECHNICAL UNIVERSITY

ELECTRONICS ENGINEERING

ELEC 334

Microprocessors Lab

PROJECT #2

Operational Scientific Calculator

Prepared by
1) 1801022037 Ömer Emre POLAT

1. Introduction

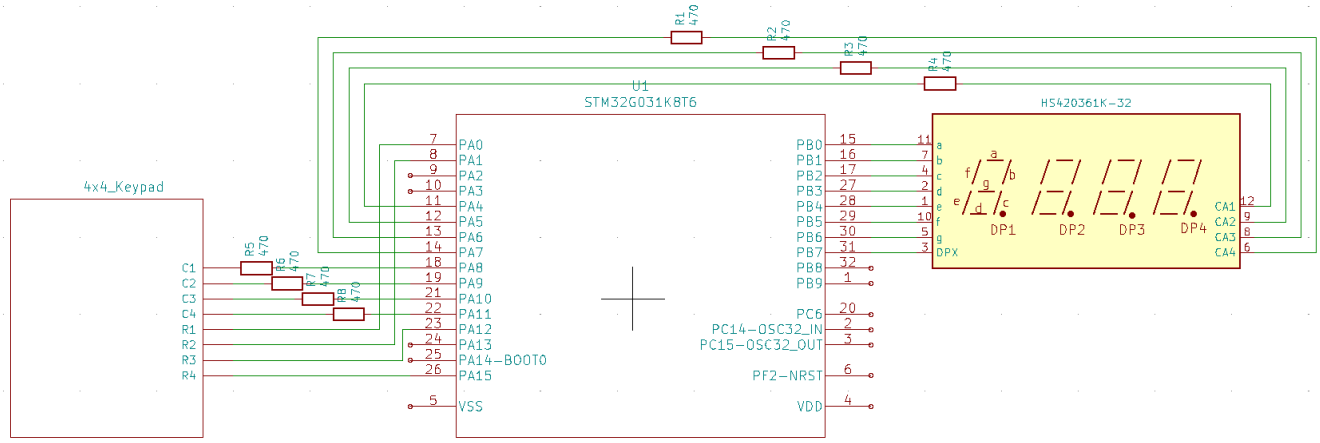
In this Project we will make a fully operational scientific calculator in programming language C. This calculator will have a keypad for button inputs and will use a 4-digit seven segment display to display the calculations.

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	

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

Now that the absolute maximum DC forward current values are adjusted we can layout the pins.

The pin connections are as follows:

PB0 -> A (SSD)	PA0 -> R1 (KP)
PB1 -> B (SSD)	PA1 -> R2 (KP)
....	PA8 -> C1 (KP)
PB6 -> G (SSD)	PA9 -> C2 (KP)
PB7 -> DPX (SSD)	PA10 -> C3 (KP)
PA4 -> D1 (SSD)	PA11 -> C4 (KP)

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

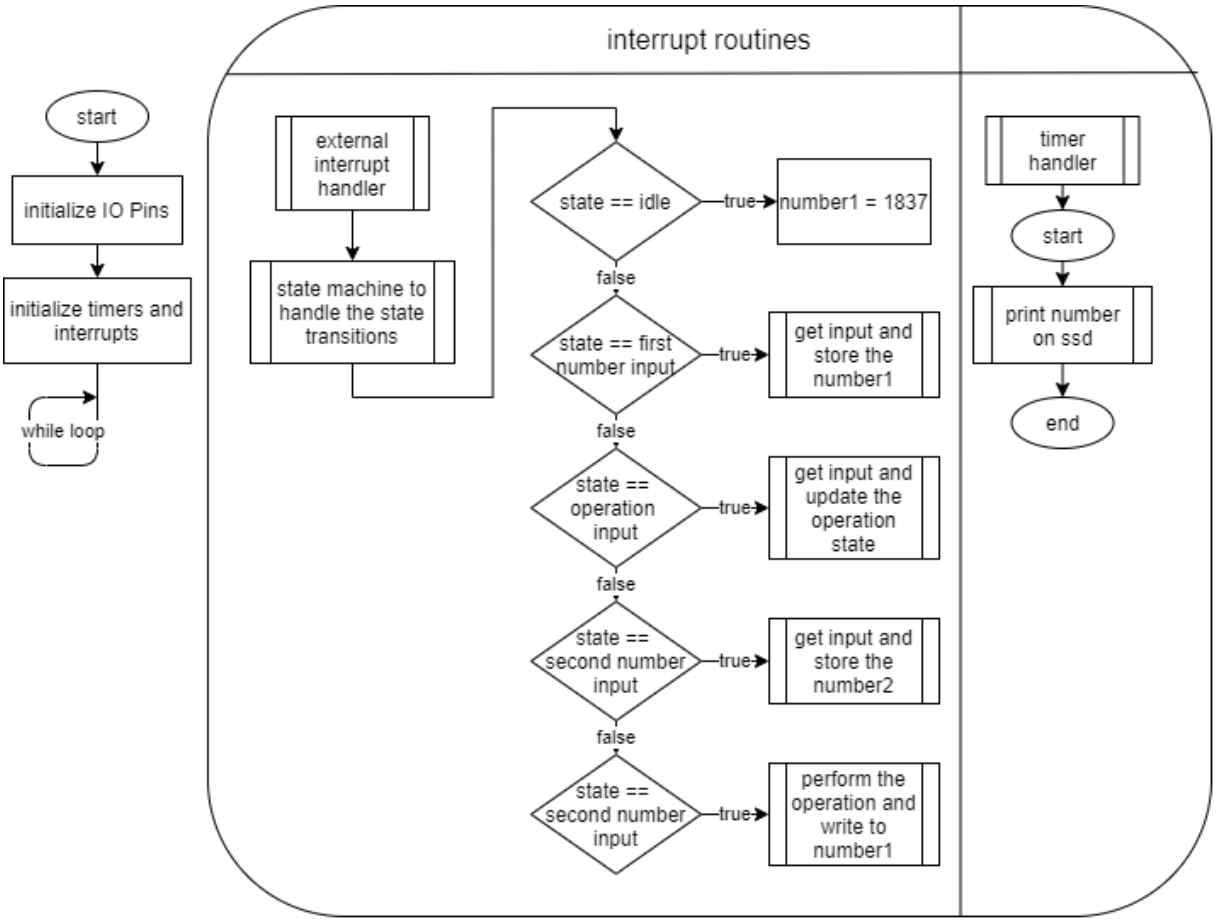
PA6 -> D3 (SSD) PA15 -> R4 (KP)

PA7 -> D4 (SSD)

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

Code will first set the peripherals and pin input output modes. The runtime code will start with the idle state where the 1837 number is shown. After an input is registered the state will be changed to the first number input. After the state is changed the number input will be taken and will be stored in a global number1 variable.

After the number1 is taken an operation input will again change the state to operation input and take the operation input. When a number input is registered in operation input the state will be changed to the second number input and the input number will be kept in number2 global variable. Lastly if a result button is pressed the result will be shown or if an operation input is given it will do the operation, store the result in number1 and go back to operation input. This will allow operation chaining.



The state machine will be added later when the code is typed since it has too much statements to check and the flowchart would become too big. This state machine will be explained in detail later when the code is written.

2.2 Writing the Timer Handler Function

The timer handler function will be implemented as just a caller that calls the print function in given intervals.

```
void TIM1_BRK_UP_TRG_COM_IRQHandler(void)
{
	TIM1->SR &= ~(1U << 0); /* reset status register */
	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 a single digit more brighter than the others.

```
/* timer and interrupt setup */

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

	TIM1->CR1 = 0; /* resetting control register */
	TIM1->CR1 |= (1U << 7); /* ARPE buffering */
	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 setup is shown above (partially combined timer and interrupt setup) with commented explanations of each line.

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.

```
/* PA8-PA11 */
void EXTI4_15_IRQHandler(void)
{
	static int op_input_count = 0; /* counter for valid inputs  */
	static int op_input_complete = 0; /* operation input status */

	kp_get_input(); /* get the input and write it to global input */

	switch(state) /* state machine */
	{
		case idle: /* reset numbers or switch to according state */
			number_1 = 0;
			number_2 = 0;
			if(kp_input_is_number())
			{
				state = first_number_input;
			}
			else
			{
				state = operation_input;
			}
			break;
		case first_number_input:
			if(kp_input_is_number())
			{
				//stay in first
			}
			else
			{
				if(kp_input == 'F') /* if input is result */
				{
					state = result;
				}
				else
				{
					state = operation_input;
```

```
                        }
                }
                break;
        case operation_input:
                if(kp_input_is_number())
                {
                        state = second_number_input;
                        op_input_complete = 0;
                }
                else
                {
                        if(kp_input == 'F') /* if input is result */
                        {
                                state = result;
                                op_input_complete = 0;
                        }
                        else
                        {
                                if(op_input_complete)
                                {
                                        state = second_number_input;
                                        op_input_complete = 0;
                                }
                                else
                                {
                                        // stay in op
                                }
                        }
                }
                break;
        case second_number_input:
                if(kp_input_is_number())
                {
                        //stay in second
                }
                else
                {
                        if(kp_input == 'F') /* if input is result */
                        {
                                state = result;
                        }
                        else
                        {
                                perform_op(); /* this part allows for chaining
ops */
                                number_2 = 0;
                                op = 0;
                                op_input_count = 0;
                                op_input_complete = 0;
                                state = operation_input;
                        }
                }
                break;
        case result:
                if(kp_input_is_number())
                {
                        state = first_number_input;
                        number_1 = 0;
                        number_2 = 0;
```

```
                }
                else
                {
                        if(kp_input == 'F') /* if input is result */
                        {
                                //stay in result
                        }
                        else
                        {
                                state = operation_input;
                                number_2 = 0;
                                op = 0;
                                op_input_count = 0;
                                op_input_complete = 0;
                        }
                }
                break;
        }

        switch(state) /* performs operations depending on the switched state */
        {
                case idle:
                        // do nothing
                        break;
                case first_number_input:
                        /* this part is where the char input is turned into int value
*/
                        /* reasoning for the operation is: ((char) 1 ) - 48 == (int) 1
*/
                        number_1 = (10 * number_1) + (float) (((uint32_t) kp_input) -
48);
                        break;
                case operation_input:
                        if(kp_input == 'F') /* if input is result */
                        {
                                // do nothing
                        }
                        else if(kp_input == 'E') /* if input is alternate mode */
                        {
                                if((op_input_count == 0) || (op_input_count == 1))
                                {
                                        op_input_count++;
                                        op += 4;
                                }
                                else
                                {
                                        // do nothing
                                }
                        }
                        else /* if input is not result or alternate mode */
                        {
                                if((kp_input == 'A') || (kp_input == 'B') || (kp_input
== 'C') || (kp_input == 'D'))
                                {
                                        if(op_input_count == 2)
                                        {
                                                // do nothing
                                        }
                                        else
```

```
                                {
                                        op += (((uint32_t) kp_input) - 65);
                                        op_input_complete = 1;
                                }
                        }
                        else
                        {
                                // do nothing
                        }
                }
                break;
        case second_number_input:
                /* this part is where the char input is turned into int value
*/
                /* reasoning for the operation is: ((char) 1 ) - 48 == (int) 1
*/
                number_2 = (10 * number_2) + (float) (((uint32_t) kp_input) -
48);
                break;
        case result:
                if(kp_input_is_number())
                {
                        number_1 = 0;
                        state = first_number_input;
                }
                else if(kp_input == 'F') /* if input is result */
                {
                        //do nothing
                }
                else
                {
                        state = operation_input;
                }
                perform_op();
                break;
        }

        for(uint32_t i=0; i<400000; 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 idle state and resets the number values. If the keypad input is number, then it transitions to first number input state.

In first number state, if the input is a number then in stays in the state and the main function body that constructs the new number ex.(1st input 5, 2nd input 7, number construction 57). If the input is not a number, then the first number input is already given or is zero. Then the state transitions to operation input state.

In operation input state, if the input is number then the operation input is complete and state is set to second number input state. If the input is not a number and is result, then the

state is changed to result. Lastly if the input is alternate function or other operations the op state is changed accordingly.

In second number input state, if the input is a number then the state is kept at second number input state. The number input will be used to construct a new number accordingly. If the input is not a number and is result, then the state is set to result. If the input is an operation different than result then we have to chain operations ex.(“1 + 3 – 7” second operation as indicated in red). This chaining happens by performing the operation on the state machine itself and jumping back to the operation input state so that the input will be counted as another operation.

Lastly in result state, if the input is result then the state doesn’t change. If the input is a number then the number1 and number2 are reset and state is set back to first number input (new calculation starts). If the input is operation, state is set back to the operation state just like how we chained operations in second input state.

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 with functions like perform_op(). This function will be explained below.

2.4 Perform Operation and Get Input Functions

Perform operation functions works with the current operation state enumerator and the number1, number 2 global variables. It just performs the chosen operation in call.

```
/* performs operation depending on the global enumerator, */
/* op and global variables number1 and number 2           */
void perform_op(void)
{
	switch(op)
	{
		case addition:
			number_1 += number_2;
			break;
		case subtraction:
			number_1 -= number_2;
			break;
		case multiplication:
			number_1 *= number_2;
			break;
		case division:
			number_1 /= number_2;
			break;
		case logarithm:
			number_1 = (float) log10((double) number_2);
			break;
		case ln:
			number_1 = (float) log(number_2);
			break;
		case squareroot:
			number_1 = (float) sqrt(number_2);
			break;
```

```
            case square:
                  number_1 = (float) pow(number_2, 2);
                  break;
            case sine:
                  number_1 = (float) sin(number_2);
                  break;
            case cosine:
                  number_1 = (float) cos(number_2);
                  break;
            case tangent:
                  number_1 = (float) tan(number_2);
                  break;
            case cotangent:
                  number_1 = (float) (1.0 / tan(number_2));
                  break;
            case pi:
                  number_2 = (float) M_PI;
                  break;
      }
}
```

Perform operation function performs operations and writes the operation result on number_1 depending on the current op enumerators state. Its pretty simple so there wont be much detailed explanation on this.

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 */

	GPIOA->ODR &= ~(1U << 12); /* shut off A12 */
	//for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 2;
	}
	GPIOA->ODR |= (1U << 12); /* turn on A12 */

	GPIOA->ODR &= ~(1U << 15); /* shut off A15 */
	//for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 3;
	}
	GPIOA->ODR |= (1U << 15); /* turn on A15 */

	kp_input = kp_inputs[(c + (4 * r))];
}
```

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 &= (0xC0U);
			break;
		case 1:
			GPIOB->ODR &= (0xF9U);
			break;
		case 2:
			GPIOB->ODR &= (0xA4U);
			break;
```

```
            case 3:
                  GPIOB->ODR &= (0xB0U);
                  break;
            case 4:
                  GPIOB->ODR &= (0x99U);
                  break;
            case 5:
                  GPIOB->ODR &= (0x92U);
                  break;
            case 6:
                  GPIOB->ODR &= (0x82U);
                  break;
            case 7:
                  GPIOB->ODR &= (0xF8U);
                  break;
            case 8:
                  GPIOB->ODR &= (0x80U);
                  break;
            case 9:
                  GPIOB->ODR &= (0x90U);
                  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();

      if((uint32_t) dp_pos == (digit + 1))
      {
            GPIOB->ODR &= ~(0x80U);
      }
      else
      {
            GPIOB->ODR |= (0x80U);
      }

      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 with multiple checks before printing with cases being printing number1 global variable, printing number2 global variable, printing floating point and printing negative numbers.

```
/* print and refresh functions */

void ssd_print(void)
{
	uint32_t digits[4] = {0};
	int number; /* copy of the number */

	if((number_1 < -999) || (number_2 < -999))
	{
		dp_pos = none;
		ssd_set_overflow();
		return;
	}
	if((number_1 > 9999) || (number_2 > 9999))
	{
		dp_pos = none;
		ssd_set_overflow();
		return;
	}

	switch(state) /* state preperation */
	{
		case idle:
			number = 1837;
			dp_pos = none; /* float printing */
			break;
		case first_number_input:
			if(number_1 < 0) /* negative case */
			{
				if((-1 * number_1) < 10) /* float printing */
				{
					dp_pos = second;
					number = (int) (100 * number_1);
				}
				else if((-1 * number_1) < 100)
				{
					dp_pos = third;
					number = (int) (10 * number_1);
				}
				else if((-1 * number_1) < 1000)
				{
					dp_pos = fourth;
					number = (int) number_1;
				}
				else
				{
					dp_pos = none;
				}
			}
			else /* positive case */
			{
				if(number_1 < 10) /* float printing */
				{
					dp_pos = first;
					number = (int) (1000 * number_1);
				}
				else if(number_1 < 100)
				{
```

```
                        dp_pos = second;
                        number = (int) (100 * number_1);
                }
                else if(number_1 < 1000)
                {
                        dp_pos = third;
                        number = (int) (10 * number_1);
                }
                else
                {
                        dp_pos = none;
                        number = (int) number_1;
                }
        }
        break;
case operation_input:
        number = 0;
        break;
case second_number_input:
        if(number_2 < 0) /* negative case */
        {
                if((-1 * number_2) < 10) /* float printing */
                {
                        dp_pos = second;
                        number = (int) (100 * number_2);
                }
                else if((-1 * number_2) < 100)
                {
                        dp_pos = third;
                        number = (int) (10 * number_2);
                }
                else if((-1 * number_2) < 1000)
                {
                        dp_pos = fourth;
                        number = (int) number_2;
                }
                else
                {
                        dp_pos = none;
                }
        }
        else /* positive case */
        {
                if(number_2 < 10) /* float printing */
                {
                        dp_pos = first;
                        number = (int) (1000 * number_2);
                }
                else if(number_2 < 100)
                {
                        dp_pos = second;
                        number = (int) (100 * number_2);
                }
                else if(number_2 < 1000)
                {
                        dp_pos = third;
                        number = (int) (10 * number_2);
                }
                else
```

```
                        {
                                dp_pos = none;
                                number = (int) number_2;
                        }
                }
                break;
        case result:
                if(number_1 < 0) /* negative case */
                {
                        if((-1 * number_1) < 10) /* float printing */
                        {
                                dp_pos = second;
                                number = (int) (100 * number_1);
                        }
                        else if((-1 * number_1) < 100)
                        {
                                dp_pos = third;
                                number = (int) (10 * number_1);
                        }
                        else if((-1 * number_1) < 1000)
                        {
                                dp_pos = fourth;
                                number = (int) number_1;
                        }
                        else
                        {
                                dp_pos = none;
                        }
                }
                else /* positive case */
                {
                        if(number_1 < 10) /* float printing */
                        {
                                dp_pos = first;
                                number = (int) (1000 * number_1);
                        }
                        else if(number_1 < 100)
                        {
                                dp_pos = second;
                                number = (int) (100 * number_1);
                        }
                        else if(number_1 < 1000)
                        {
                                dp_pos = third;
                                number = (int) (10 * number_1);
                        }
                        else
                        {
                                dp_pos = none;
                                number = (int) number_1;
                        }
                }
                break;
}

if(number < 0) /* negative printing */
{
        number = -1 * number;
```

```
		if(state != operation_input)
		{
			*(digits+1) = (uint32_t) (number/100); /* hundreds digit
extraction */
			number -= (int) (*(digits+1) * 100);
			*(digits+2) = (uint32_t) (number/10); /* tens digit extraction
*/
			number -= (int) (*(digits+2) * 10);
			*(digits+3) = (uint32_t) number; /* ones digit extraction */

		for(uint32_t i=0; i<4; i++)
		{
			ssd_digit(i);
			if(i == 0) /* print - sign */
			{
				GPIOB->ODR &= (0xBFU);
			}
			else
			{
				ssd_set(digits[i]);
			}
		}
		ssd_digit_clear();
		}
		else
		{
			// print operation depending on the operation enum
			ssd_digit_clear();
			ssd_set_clear();
		}
	}
	else /* positive printing */
	{

		if(state != operation_input)
		{
			*(digits+0) = (uint32_t) (number/1000); /* thousands digit
extraction */
			number -= (int) (*(digits+0) * 1000);
			*(digits+1) = (uint32_t) (number/100); /* hundreds digit
extraction */
			number -= (int) (*(digits+1) * 100);
			*(digits+2) = (uint32_t) (number/10); /* tens digit extraction
*/
			number -= (int) (*(digits+2) * 10);
			*(digits+3) = (uint32_t) number; /* ones digit extraction */

		for(uint32_t i=0; i<4; i++)
		{
			ssd_digit(i);
			ssd_set(digits[i]);
		}
		ssd_digit_clear();
		}
		else
		{
			// print operation depending on the operation enum
			ssd_digit_clear();
			ssd_set_clear();
```

```
            }
        }
}
```

The float printing works by first checking how many digits there are after the decimal point and setting the decimal point accordingly. (10.32 has 2 digits after dp). After that we check and scale the number so that it would fit our seven segment display. (10.32 is scaled into 1032 dp_pos = 2). Scaling is different on negative numbers since we can only print only 3 number digits ex.(-10.32 scaled to -103.2). After all the processing is done the printing can happen. Printing works by having a for loop with 4 loops printing each digit using int division. We typecast the float value to int so that we can get a single digit then this digit is subtracted from the main number ex.(103.2 / 1000 = 0, 103.2 – 0*1000, 103.2 / 100 = 1, 103.2 – 1*100…). Then the extracted digits are printed to ssd using the ssd_set and ssd_digit functions.

2.6 Combined Code

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

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

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

int main(void) {

      dp_pos = none;
      op = addition;
      state = idle;

      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
```

```h
#define ButtonBounceTime 400

float number_1;
float number_2;
char kp_input;

typedef enum
{
	idle = 0, first_number_input = 1, operation_input = 2, second_number_input
= 3, result = 4
}status;

typedef enum
{
	none = 0, first = 1, second = 2, third = 3, fourth = 4
}decimal_point_pos;

typedef enum
{
	addition = 0, subtraction = 1, multiplication = 2, division = 3, logarithm
= 4, ln = 5, squareroot = 6, square = 7, sine = 8, cosine = 9, tangent = 10,
cotangent = 11, pi = 12
}operation;

status state;

decimal_point_pos dp_pos;

operation op;

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_overflow(void);
void ssd_set_clear(void);

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

void kp_get_input(void);
uint8_t kp_input_is_number(void);

void perform_op(void);

#endif
```

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

```c
#include "bsp.h"
#include <math.h>

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

/* PA8-PA11 */
void EXTI4_15_IRQHandler(void)
{
	static int op_input_count = 0; /* counter for valid inputs  */
	static int op_input_complete = 0; /* operation input status */

	kp_get_input(); /* get the input and write it to global input */

	switch(state) /* state machine */
	{
		case idle: /* reset numbers or switch to according state */
			number_1 = 0;
			number_2 = 0;
			if(kp_input_is_number())
			{
				state = first_number_input;
			}
			else
			{
				state = operation_input;
			}
			break;
		case first_number_input:
			if(kp_input_is_number())
			{
				//stay in first
			}
			else
			{
				if(kp_input == 'F') /* if input is result */
				{
					state = result;
				}
				else
				{
					state = operation_input;
				}
			}
			break;
		case operation_input:
			if(kp_input_is_number())
			{
				state = second_number_input;
				op_input_complete = 0;
			}
			else
			{
				if(kp_input == 'F') /* if input is result */
				{
```

```
                                state = result;
                                op_input_complete = 0;
                        }
                        else
                        {
                                if(op_input_complete)
                                {
                                        state = second_number_input;
                                        op_input_complete = 0;
                                }
                                else
                                {
                                        // stay in op
                                }
                        }
                }
                break;
        case second_number_input:
                if(kp_input_is_number())
                {
                        //stay in second
                }
                else
                {
                        if(kp_input == 'F') /* if input is result */
                        {
                                state = result;
                        }
                        else
                        {
                                perform_op(); /* this part allows for chaining
ops */
                                number_2 = 0;
                                op = 0;
                                op_input_count = 0;
                                op_input_complete = 0;
                                state = operation_input;
                        }
                }
                break;
        case result:
                if(kp_input_is_number())
                {
                        state = first_number_input;
                        number_1 = 0;
                        number_2 = 0;
                }
                else
                {
                        if(kp_input == 'F') /* if input is result */
                        {
                                //stay in result
                        }
                        else
                        {
                                state = operation_input;
                                number_2 = 0;
                                op = 0;
                                op_input_count = 0;
```

```
                                op_input_complete = 0;
                        }
                }
                break;
        }

        switch(state) /* performs operations depending on the switched state */
        {
                case idle:
                        // do nothing
                        break;
                case first_number_input:
                        /* this part is where the char input is turned into int value
*/
                        /* reasoning for the operation is: ((char) 1 ) - 48 == (int) 1
*/
                        number_1 = (10 * number_1) + (float) (((uint32_t) kp_input) -
48);
                        break;
                case operation_input:
                        if(kp_input == 'F') /* if input is result */
                        {
                                // do nothing
                        }
                        else if(kp_input == 'E') /* if input is alternate mode */
                        {
                                if((op_input_count == 0) || (op_input_count == 1))
                                {
                                        op_input_count++;
                                        op += 4;
                                }
                                else
                                {
                                        // do nothing
                                }
                        }
                        else /* if input is not result or alternate mode */
                        {
                                if((kp_input == 'A') || (kp_input == 'B') || (kp_input
== 'C') || (kp_input == 'D'))
                                {
                                        if(op_input_count == 2)
                                        {
                                                // do nothing
                                        }
                                        else
                                        {
                                                op += (((uint32_t) kp_input) - 65);
                                                op_input_complete = 1;
                                        }
                                }
                                else
                                {
                                        // do nothing
                                }
                        }
                        break;
                case second_number_input:
                        /* this part is where the char input is turned into int value
```

```
*/
                    /* reasoning for the operation is: ((char) 1 ) - 48 == (int) 1
*/
                    number_2 = (10 * number_2) + (float) (((uint32_t) kp_input) -
48);
                    break;
             case result:
                    if(kp_input_is_number())
                    {
                           number_1 = 0;
                           state = first_number_input;
                    }
                    else if(kp_input == 'F') /* if input is result */
                    {
                           //do nothing
                    }
                    else
                    {
                           state = operation_input;
                    }
                    perform_op();
                    break;
      }

      for(uint32_t i=0; i<400000; i++);

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

/* performs operation depending on the global enumerator, */
/* op and global variables number1 and number 2           */
void perform_op(void)
{
      switch(op)
      {
             case addition:
                    number_1 += number_2;
                    break;
             case subtraction:
                    number_1 -= number_2;
                    break;
             case multiplication:
                    number_1 *= number_2;
                    break;
             case division:
                    number_1 /= number_2;
                    break;
             case logarithm:
                    number_1 = (float) log10((double) number_2);
                    break;
             case ln:
                    number_1 = (float) log(number_2);
                    break;
             case squareroot:
                    number_1 = (float) sqrt(number_2);
                    break;
             case square:
                    number_1 = (float) pow(number_2, 2);
                    break;
```

```
		case sine:
			number_1 = (float) sin(number_2);
			break;
		case cosine:
			number_1 = (float) cos(number_2);
			break;
		case tangent:
			number_1 = (float) tan(number_2);
			break;
		case cotangent:
			number_1 = (float) (1.0 / tan(number_2));
			break;
		case pi:
			number_2 = (float) M_PI;
			break;
	}
}

/*//////////////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 */

	GPIOA->ODR &= ~(1U << 12); /* shut off A12 */
	//for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 2;
	}
	GPIOA->ODR |= (1U << 12); /* turn on A12 */

	GPIOA->ODR &= ~(1U << 15); /* shut off A15 */
	//for(uint32_t i=0; i<ButtonBounceTime; i++);
	if(((GPIOA->IDR >> (8+c)) & (1U)) ^ 1U) /* check if input is not there */
	{
		r = 3;
	}
	GPIOA->ODR |= (1U << 15); /* turn on A15 */

	kp_input = kp_inputs[(c + (4 * r))];
}

uint8_t kp_input_is_number(void)
{
	if(((kp_input == 'A')||(kp_input == 'B')||(kp_input == 'C')||(kp_input ==
'D')||(kp_input == 'E')||(kp_input == 'F')))
	{
		return 0;
	}
	else
	{
		return 1;
	}
}

/*///////////////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 |= (0x5555U); /* Setup (B0, B7) as output */
```

```
    GPIOB->MODER &= ~(0xAAAAU); /* 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 */

    /* timer and interrupt setup */

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

      TIM1->CR1 = 0; /* resetting control register */
      TIM1->CR1 |= (1U << 7); /* ARPE buffering */
      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 */

    /* 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 |= (0x7FU);
}

void ssd_set(volatile uint32_t digit)
{
      switch(digit)
      {
            case 0:
                  GPIOB->ODR &= (0xC0U);
                  break;
            case 1:
                  GPIOB->ODR &= (0xF9U);
                  break;
            case 2:
                  GPIOB->ODR &= (0xA4U);
                  break;
            case 3:
                  GPIOB->ODR &= (0xB0U);
                  break;
            case 4:
                  GPIOB->ODR &= (0x99U);
                  break;
            case 5:
                  GPIOB->ODR &= (0x92U);
                  break;
            case 6:
                  GPIOB->ODR &= (0x82U);
                  break;
            case 7:
                  GPIOB->ODR &= (0xF8U);
                  break;
            case 8:
                  GPIOB->ODR &= (0x80U);
                  break;
            case 9:
                  GPIOB->ODR &= (0x90U);
```

```
                    break;
        }
}

void ssd_set_operation(volatile char oper)
{
        switch(oper)
        {
                case 'A':

                    break;
        }
}

void ssd_set_overflow(void)
{
        ssd_digit(0);
        GPIOB->ODR &= (0xC0U); /* O */
        ssd_digit(1);
        GPIOB->ODR &= (0xE3U); /* v */
        ssd_digit(2);
        GPIOB->ODR &= (0x8EU); /* F */
        ssd_digit(3);
        GPIOB->ODR &= (0xC7U); /* L */
        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();

        if((uint32_t) dp_pos == (digit + 1))
        {
                GPIOB->ODR &= ~(0x80U);
        }
        else
        {
                GPIOB->ODR |= (0x80U);
        }

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

/* print and refresh functions */

void ssd_print(void)
{
        uint32_t digits[4] = {0};
        int number; /* copy of the number */

        if((number_1 < -999) || (number_2 < -999))
        {
                dp_pos = none;
```

```
		ssd_set_overflow();
		return;
	}
	if((number_1 > 9999) || (number_2 > 9999))
	{
		dp_pos = none;
		ssd_set_overflow();
		return;
	}

	switch(state) /* state preperation */
	{
		case idle:
			number = 1837;
			dp_pos = none; /* float printing */
			break;
		case first_number_input:
			if(number_1 < 0) /* negative case */
			{
				if((-1 * number_1) < 10) /* float printing */
				{
					dp_pos = second;
					number = (int) (100 * number_1);
				}
				else if((-1 * number_1) < 100)
				{
					dp_pos = third;
					number = (int) (10 * number_1);
				}
				else if((-1 * number_1) < 1000)
				{
					dp_pos = fourth;
					number = (int) number_1;
				}
				else
				{
					dp_pos = none;
				}
			}
			else /* positive case */
			{
				if(number_1 < 10) /* float printing */
				{
					dp_pos = first;
					number = (int) (1000 * number_1);
				}
				else if(number_1 < 100)
				{
					dp_pos = second;
					number = (int) (100 * number_1);
				}
				else if(number_1 < 1000)
				{
					dp_pos = third;
					number = (int) (10 * number_1);
				}
				else
				{
					dp_pos = none;
```

```
                        number = (int) number_1;
                }
        }
        break;
case operation_input:
        number = 0;
        break;
case second_number_input:
        if(number_2 < 0) /* negative case */
        {
                if((-1 * number_2) < 10) /* float printing */
                {
                        dp_pos = second;
                        number = (int) (100 * number_2);
                }
                else if((-1 * number_2) < 100)
                {
                        dp_pos = third;
                        number = (int) (10 * number_2);
                }
                else if((-1 * number_2) < 1000)
                {
                        dp_pos = fourth;
                        number = (int) number_2;
                }
                else
                {
                        dp_pos = none;
                }
        }
        else /* positive case */
        {
                if(number_2 < 10) /* float printing */
                {
                        dp_pos = first;
                        number = (int) (1000 * number_2);
                }
                else if(number_2 < 100)
                {
                        dp_pos = second;
                        number = (int) (100 * number_2);
                }
                else if(number_2 < 1000)
                {
                        dp_pos = third;
                        number = (int) (10 * number_2);
                }
                else
                {
                        dp_pos = none;
                        number = (int) number_2;
                }
        }
        break;
case result:
        if(number_1 < 0) /* negative case */
        {
                if((-1 * number_1) < 10) /* float printing */
                {
```

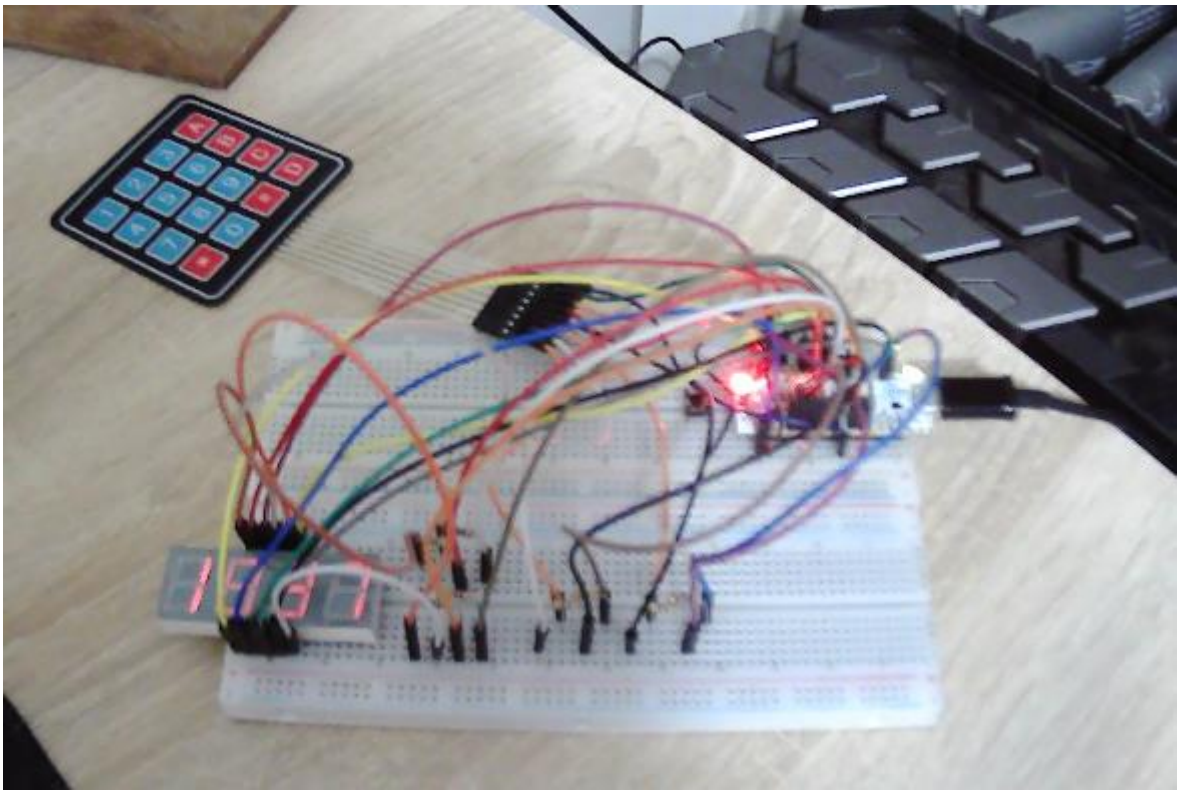
```
                                dp_pos = second;
                                number = (int) (100 * number_1);
                        }
                        else if((-1 * number_1) < 100)
                        {
                                dp_pos = third;
                                number = (int) (10 * number_1);
                        }
                        else if((-1 * number_1) < 1000)
                        {
                                dp_pos = fourth;
                                number = (int) number_1;
                        }
                        else
                        {
                                dp_pos = none;
                        }
                }
                else /* positive case */
                {
                        if(number_1 < 10) /* float printing */
                        {
                                dp_pos = first;
                                number = (int) (1000 * number_1);
                        }
                        else if(number_1 < 100)
                        {
                                dp_pos = second;
                                number = (int) (100 * number_1);
                        }
                        else if(number_1 < 1000)
                        {
                                dp_pos = third;
                                number = (int) (10 * number_1);
                        }
                        else
                        {
                                dp_pos = none;
                                number = (int) number_1;
                        }
                }
                break;
        }

        if(number < 0) /* negative printing */
        {
                number = -1 * number;

                if(state != operation_input)
                {
                        *(digits+1) = (uint32_t) (number/100); /* hundreds digit
extraction */
                        number -= (int) (*(digits+1) * 100);
                        *(digits+2) = (uint32_t) (number/10); /* tens digit extraction
*/
                        number -= (int) (*(digits+2) * 10);
                        *(digits+3) = (uint32_t) number; /* ones digit extraction */

                for(uint32_t i=0; i<4; i++)
```

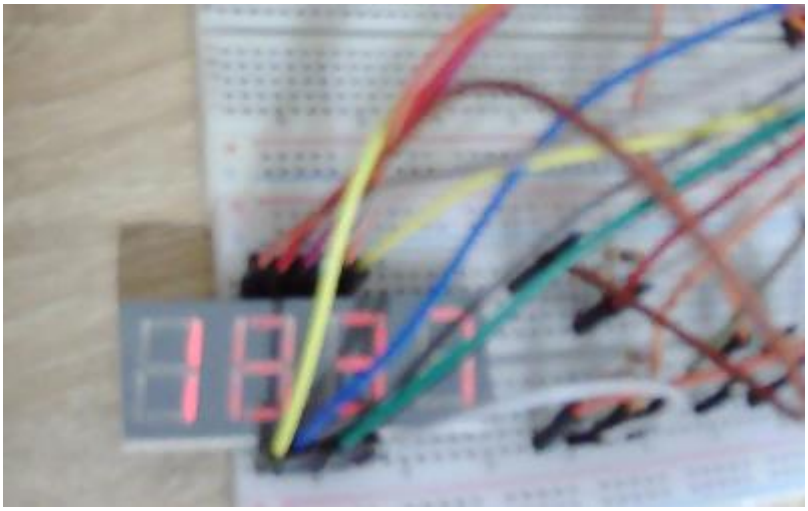
```
		{
			ssd_digit(i);
			if(i == 0) /* print - sign */
			{
				GPIOB->ODR &= (0xBFU);
			}
			else
			{
				ssd_set(digits[i]);
			}
		}
		ssd_digit_clear();
		}
		else
		{
			// print operation depending on the operation enum
			ssd_digit_clear();
			ssd_set_clear();
		}
	}
	else /* positive printing */
	{

		if(state != operation_input)
		{
			*(digits+0) = (uint32_t) (number/1000); /* thousands digit
extraction */
			number -= (int) (*(digits+0) * 1000);
			*(digits+1) = (uint32_t) (number/100); /* hundreds digit
extraction */
			number -= (int) (*(digits+1) * 100);
			*(digits+2) = (uint32_t) (number/10); /* tens digit extraction
*/
			number -= (int) (*(digits+2) * 10);
			*(digits+3) = (uint32_t) number; /* ones digit extraction */

		for(uint32_t i=0; i<4; i++)
		{
			ssd_digit(i);
			ssd_set(digits[i]);
		}
		ssd_digit_clear();
		}
		else
		{
			// print operation depending on the operation enum
			ssd_digit_clear();
			ssd_set_clear();
		}
	}
}
```

The code may be long and hard to read but analyzing the functions individually is a lot easier.

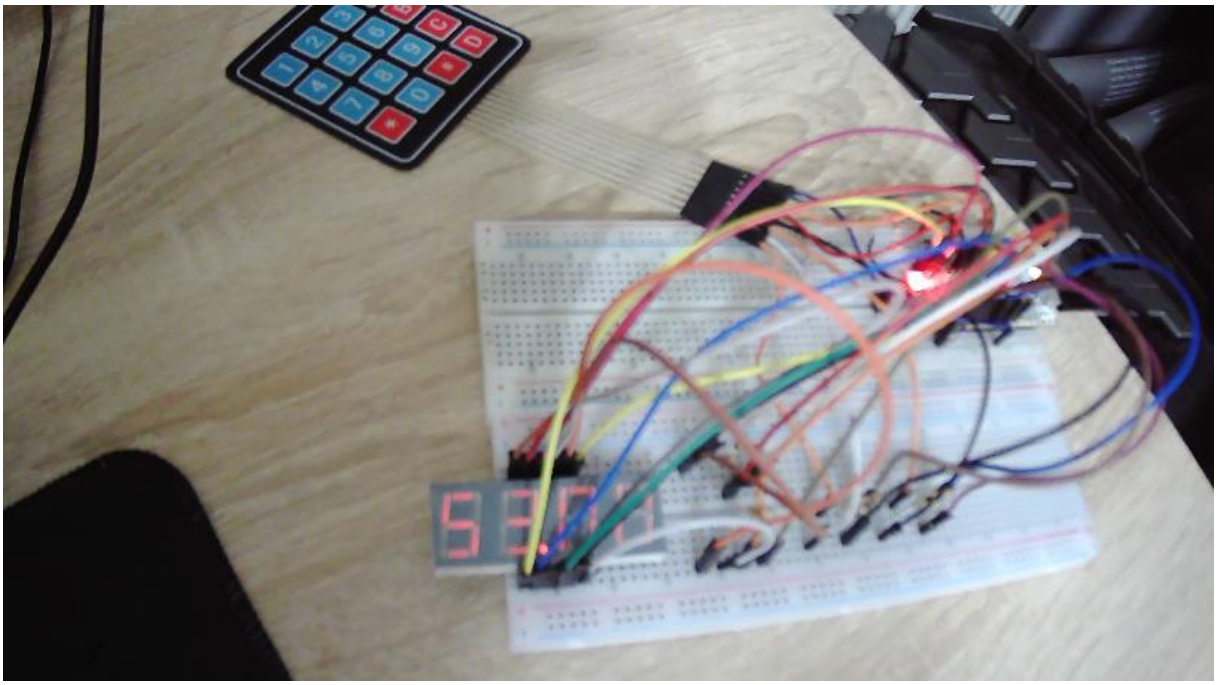
2.7 Testing of the Board with Peripherals

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.

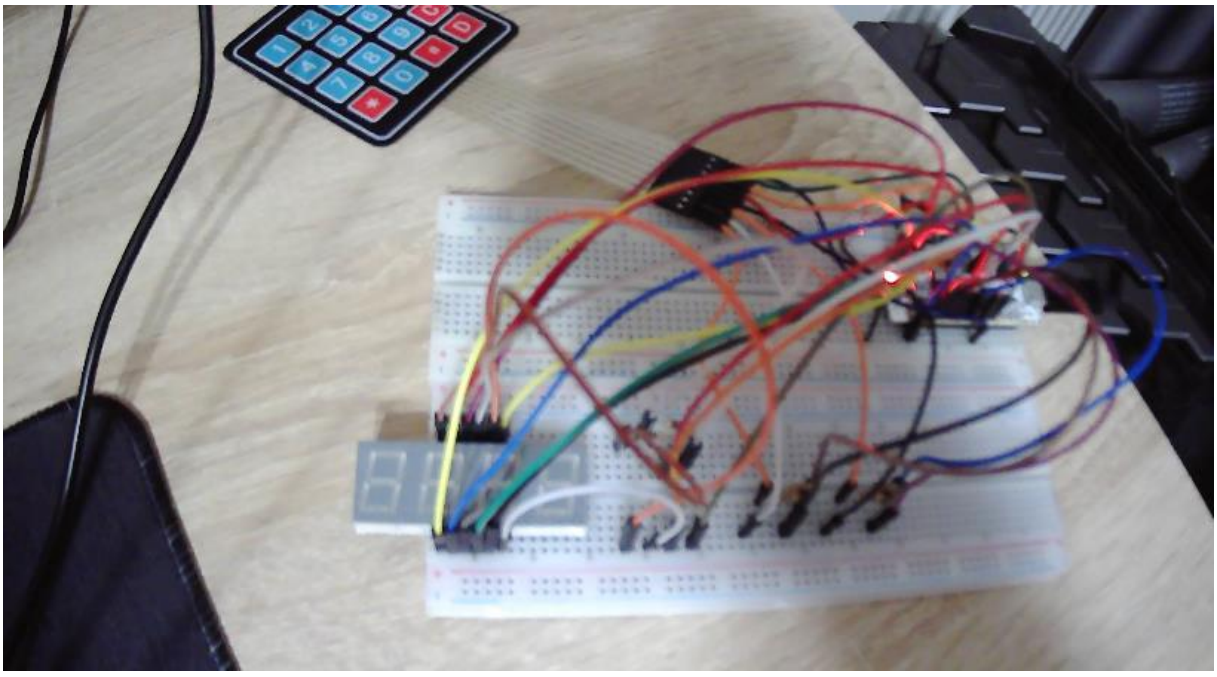




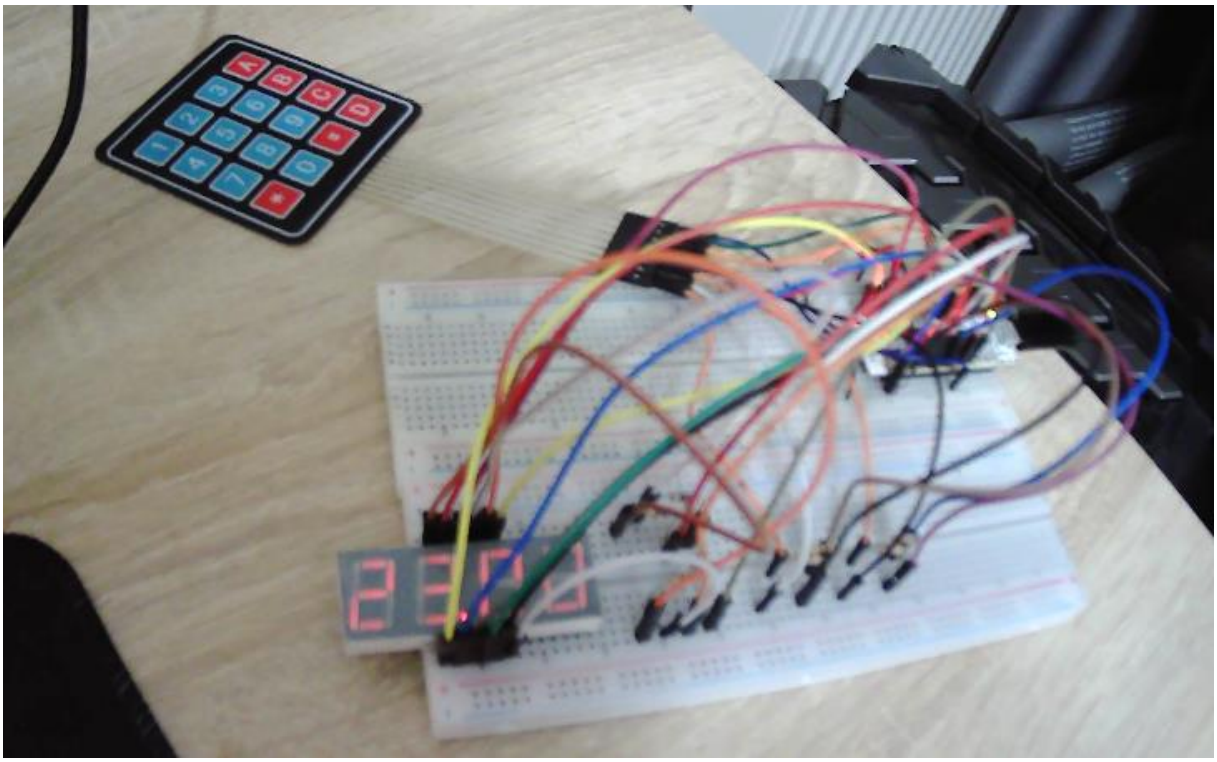
As expected the code writes the 1837 number (first two and last two digits of my school number) in idle state.



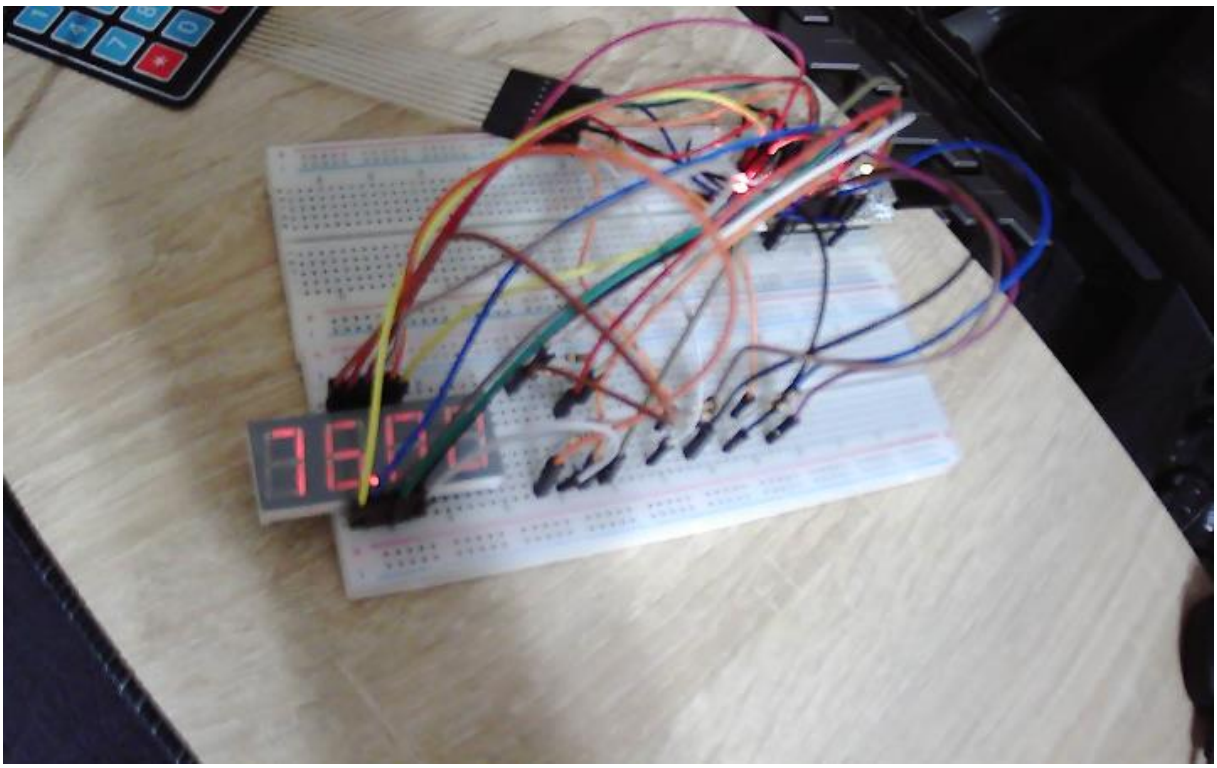
When we enter a number (in this case 5 and 3) we get the 53.00 on the seven segment display.



When an operation button is pressed (in this case A which is addition) the ssd turns off to indicate that it is in operation input state.

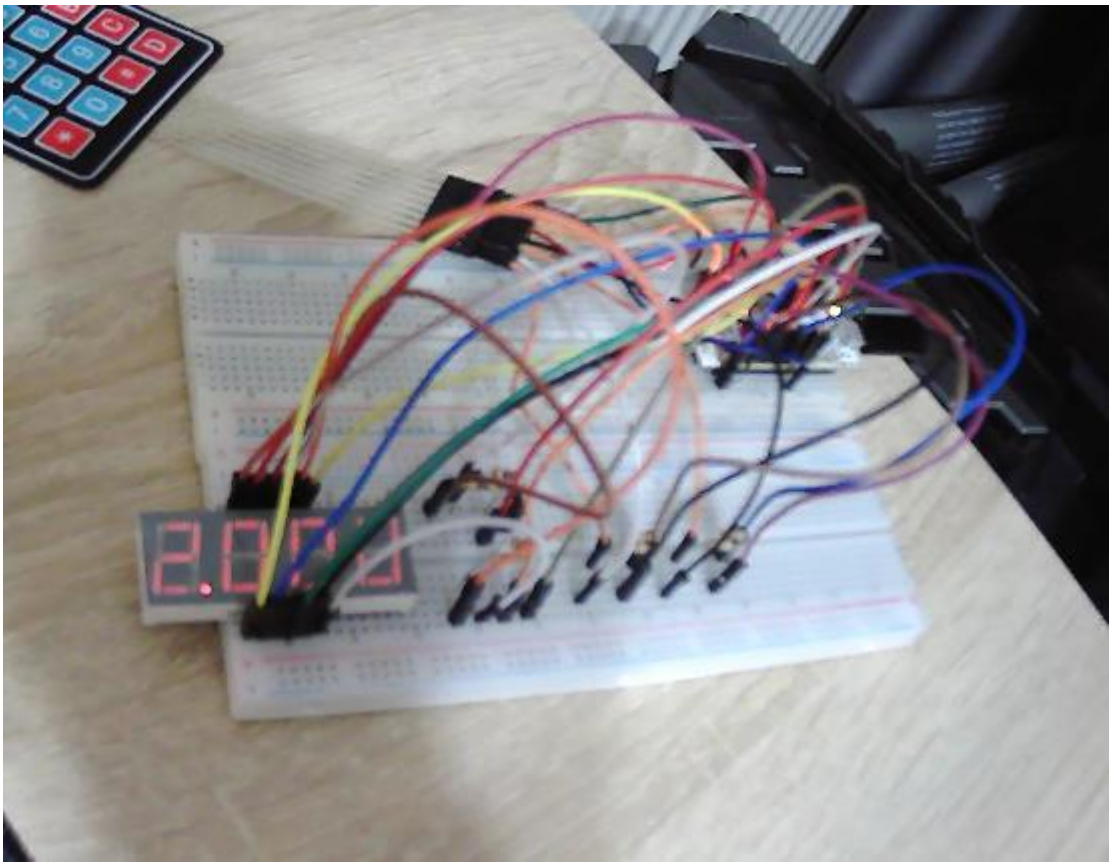


As the second input we enter 23 (as 2 and 3) and we get 23.00 on ssd.

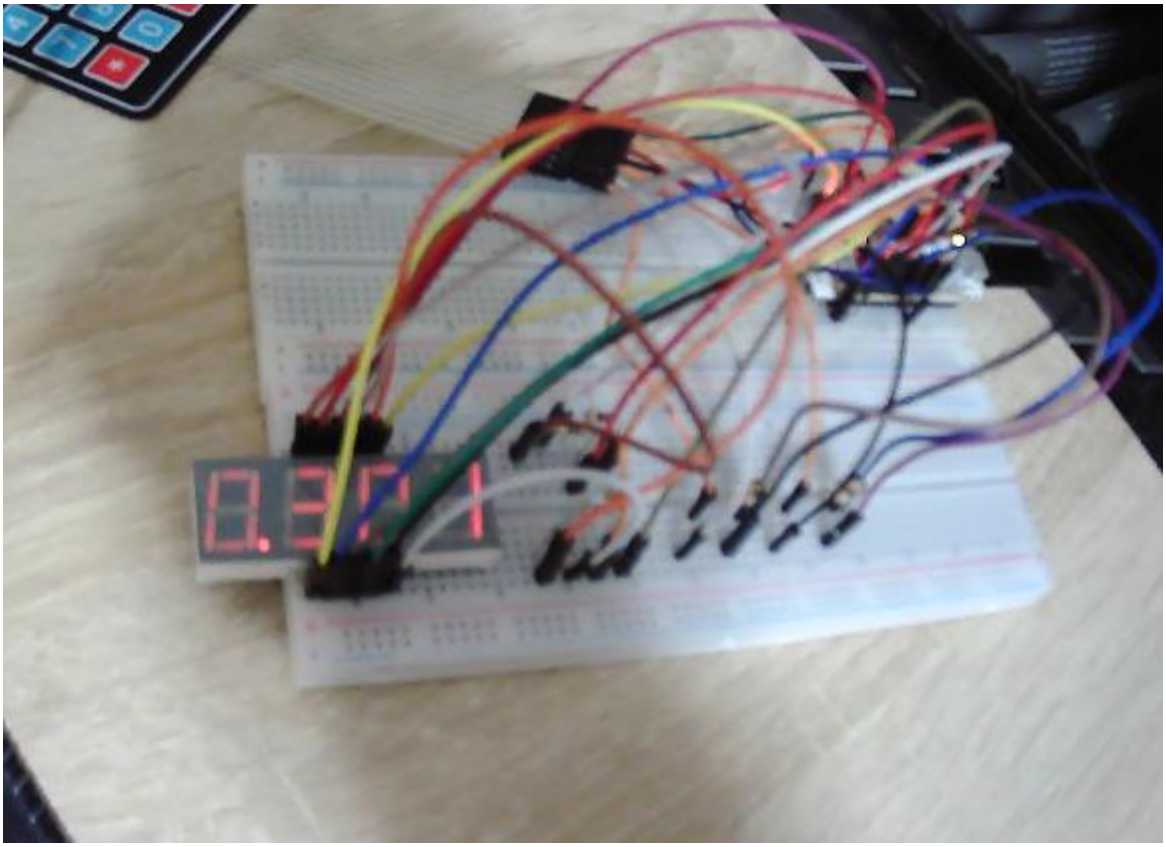


When the result operation is pressed we get the result 76.00 which is the correct value.

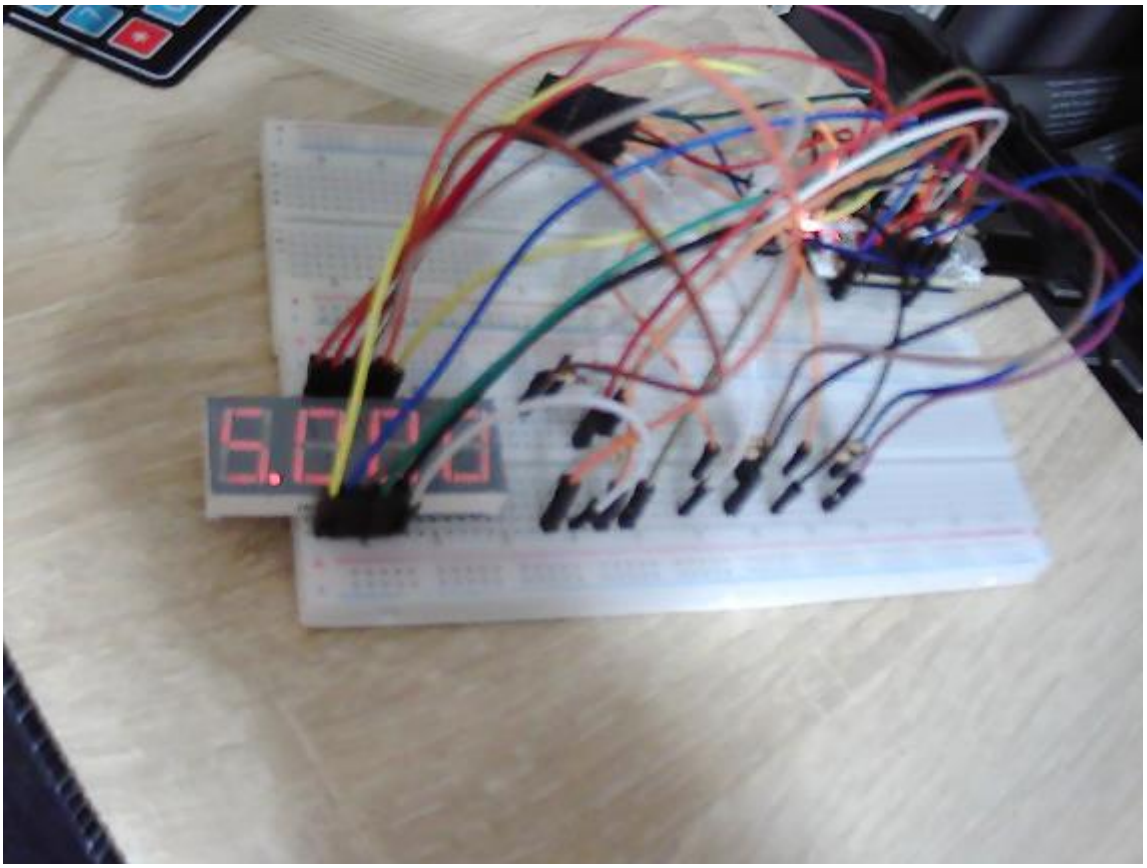
Now we can test an alternate function to check if it works too.



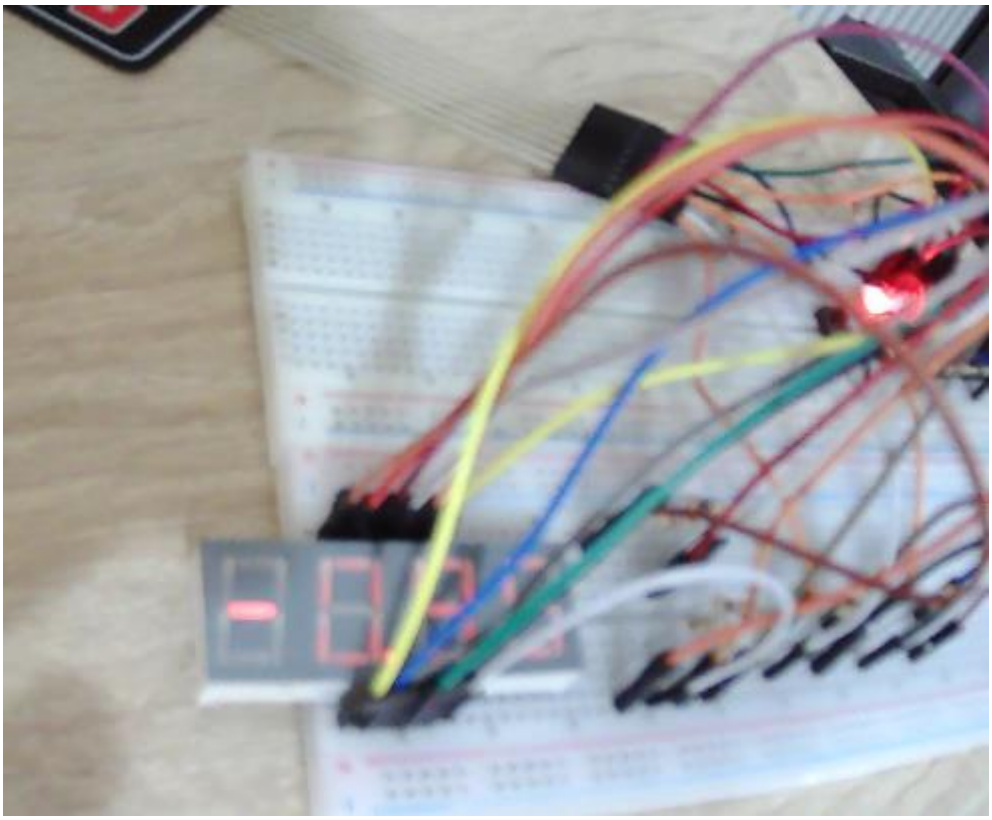
Upon entering the input E, A (logarithm base 10) and then as secondary number 2, we get the result 0.301 which is correct.



After the first alternate function test, now we can test the second alternate function sin.



After entering E, E and A then 5 we can press the result operation.

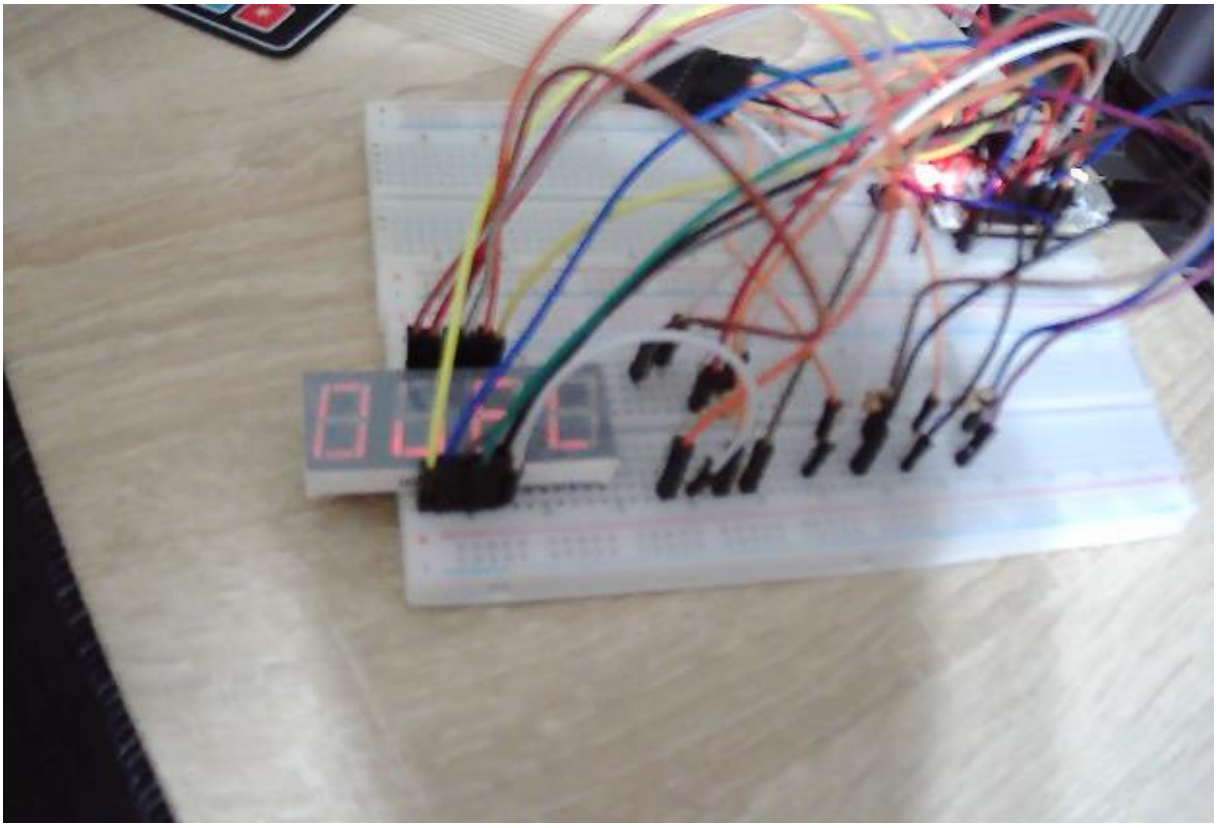


As we can see we get the result -0.95 which is correct.

sin(5 radians) =

-0.95892427466

Lastly we can check if the overflow statement works and we press 5 button 5 times and overflow appears on the ssd.



2.8 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	8	~
6	Seven Segment Display	1	7.99TL
8	STM32G031K8T6 Board	1	102.50TL
			TOTAL
			126.56TL

Total price list comes to around 126.56TL excluding the cargo costs. Build takes around 5 to 10 minutes using the block diagram given at the beginning of the report.

2.9 Video and Documenting the Project

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

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.

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