Digital Instrumentation

Journal for Exercise 2

Participants: **Tibor Illés**
Benedikt Klingebiel
Christian Jehle

Sep 18, 2022

1. Writing data to the LCD

Function description:

main

Syntax `int main(void);`

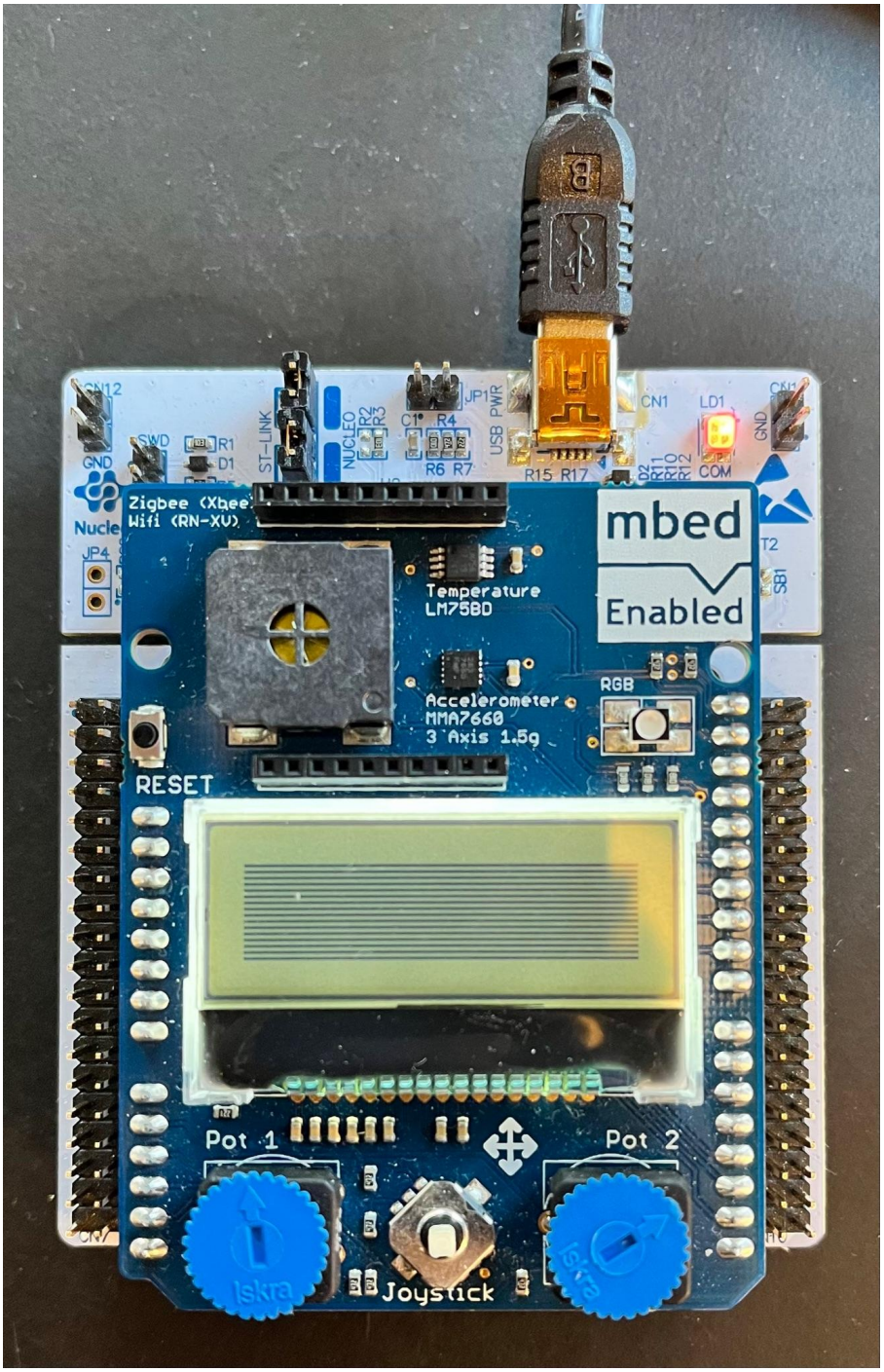
Parameters -

The function is the main function. It initializes the UART port and the SPI LCD using the lcd.h library. After that the local buffer for the lcd is created and is filled with the hex value 0xAA. This local buffer is then transmitted to the lcd using lcd_push_buffer() function.

Source Code:

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"               // Input/output library for this course

#include "lcd.h"
#include <string.h>
#include "flash.h"


int main(void)
{
        uart_init(9600); //Initialize USB serial at 9600 baud
        init_spi_lcd();       //Initialize SPI LCD

        printf("LCD");

        uint8_t fbuffer[512]; //Create local frame buffer
        memset(fbuffer, 0xAA, 512);

        lcd_push_buffer(fbuffer); //push local frame buffer to lcd

        while(1){
        }
}
```

Results:


mbed
Enabled
Zigbee (Xbee
Wifi (RN-XV)
Temperature
LM75BD
Accelerometer
MMA7660
3 Axis 1.5g
RGB
RESET
Pot 1
Pot 2
Joystick
Iskra
Iskra

2. Writing strings on the LCD

Function description:

main

Syntax `int main(void);`

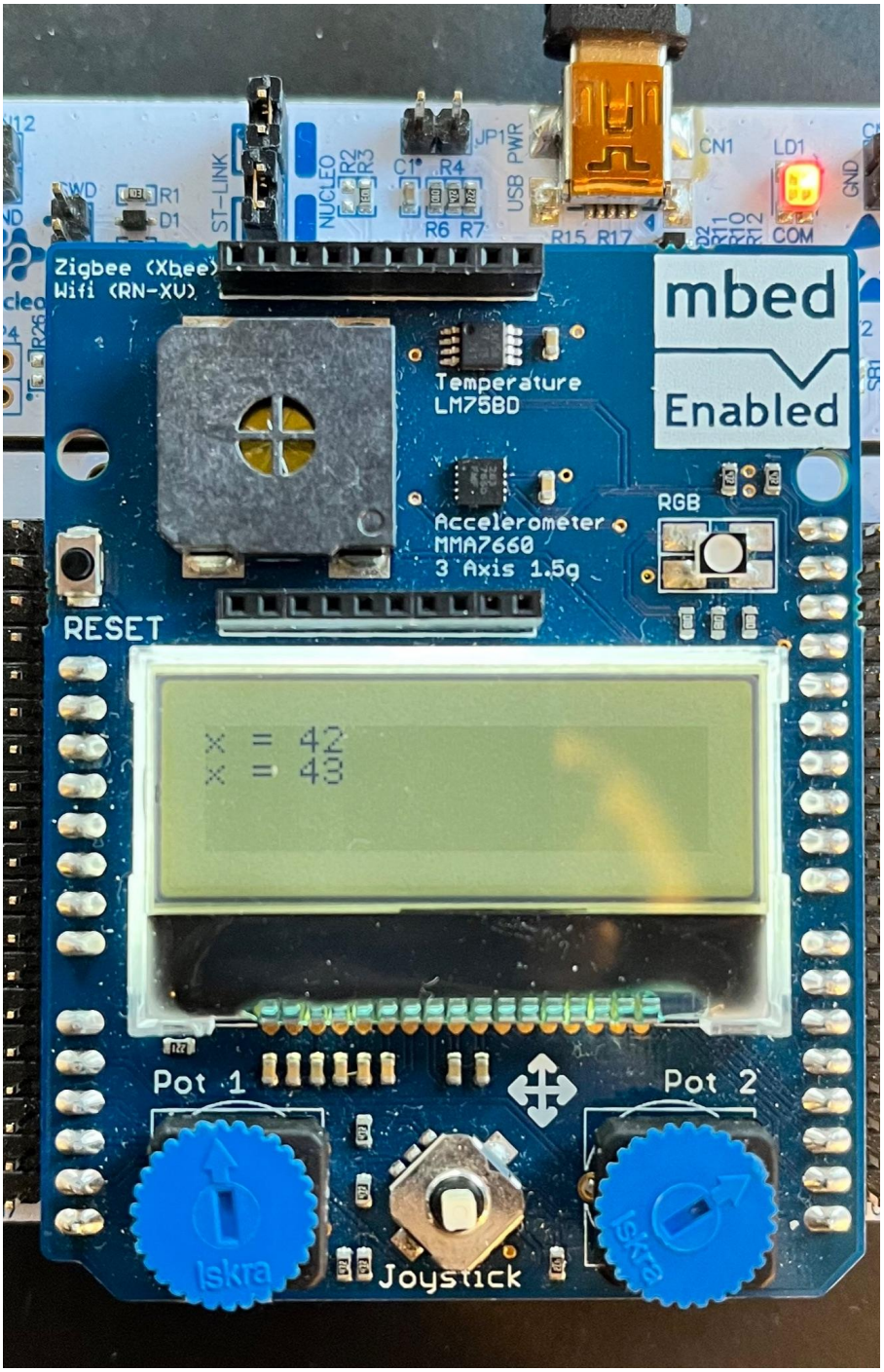
Parameters -

The function is the main function. It initializes the UART port and the SPI LCD using the lcd.h library. After that the local buffer for the lcd is created. This local buffer is then transmitted to the lcd using lcd_push_buffer() function. In the following, two character arrays are created with the sprint function, each representing an LCD row. For demonstration purposes, both line arrays are filled with dummy formulas that contain a single integer number. Afterwards both lines are loaded into the frame buffer with the lcd_write_string function. Finally the frame buffer is pushed to the lcd.

Source Code:

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"               // Input/output library for this course

#include "lcd.h"
#include <string.h>
#include "flash.h"


int main(void)
{
        uart_init(9600); // Initialize USB serial at 9600 baud
        init_spi_lcd(); //Initialize SPI LCD

        printf("LCD");

        uint8_t fbuffer[512]; //Create local frame buffer
        memset(fbuffer, 0x00, 512);

        char line[128]; //Create character array for line 1
        sprintf(line, "x = %d", 42);

        char line1[128]; //Create character array for line 2
        sprintf(line1, "x = %d", 43);

        lcd_write_string((uint8_t*)line, fbuffer, 0, 0); //write line to frame buffer
        lcd_write_string((uint8_t*)line1, fbuffer, 0, 1); //write line to frame buffer

        lcd_push_buffer(fbuffer); //push frame buffer to lcd

        while(1){
        }
}
```

Results:


Zigbee (Xbee)
Wifi (RN-XV)
mbed
Enabled
Temperature
LM75BD
Accelerometer
MMA7660
3 Axis 1.5g
RGB
RESET
x = 42
x = 43
Pot 1
Pot 2
Joystick
Iskra
Iskra
ST-LINK
NUCLEO
USB PWR
JP1
CN1
LD1
COM
GND

3. Writing to FLASH

Function description:

main

Syntax `int main(void);`

Parameters -

The Main function first initializes the UART port. Then, according to the task description, four variables are declared that are to be stored in the flash (an 8-bit integer, a 16-bit integer, a 32-bit integer, and a float variable). Then the process starts to store the variables into the flash:

1. *First the Flash is unlocked with the Flash_Unlock function to be able to use the Write functions.*
2. *Then the base address page of the flash memory is erased to be able to write to it. For this the command sequence from the instruction is used.*
3. *Next, the 3 integer variables are written to the Flash. For this the function write_word_flash is used, which writes a 32-bit word to a given address with given offset. To be able to convert the 8-bit and 16-bit integers into a 32-bit integer variable, the uint_t_32 conversion function is used. Also, to prevent the 3 variables from overwriting each other we use an incrementing address offset.*
4. *Finally we write the float variable into the flash memory. The write_word_flash function is used for this.*
5. *Finally, the flash is locked again to prevent further manipulation.*

After writing these four different variable types to the flash, the program starts to fill the flash with an incremental pattern according to the task description. This is done in the following way:

1. *First the Flash is unlocked again to allow writing.*
2. *After that, we erase the flash as it's not possible two write to flash twice in a row.*
3. *Then we start to fill the flash. We do that using a for loop, that increments the address offset and the integer that we want to write to the memory each iteration by one. As a result, the flash is filled with a 512 word long sequence of numbers. Corresponding result screenshots can be seen on the next page.*

Source Code:

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"           // Input/output library for this course

#include "flash.h"

int main(void)
{
        //Create 4 Variables that are supposed to be written to the flash
        uint8_t uint8_flash = 0xAD;      //8-bit Integer
        uint16_t uint16_flash = 0xADAD; //16-bit Integer
        uint32_t uint32_flash = 0xADADADAD; //32-bit Integer
        float float_flash = 1.5; //float
        uint32_t uint32_flash2 = 0x00000000; //32-bit Integer

        FLASH_Unlock(); //unlock flash
        //ERASE FLASH
        printf("Write Float to Flash\n");
        FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_PGERR | FLASH_FLAG_WRPERR);

        //WRITE UINT8 TO FLASH
        FLASH_ErasePage(PG31_BASE); //erase flash
        write_word_flash(PG31_BASE, 0, (uint32_t)uint8_flash);
        printf("Write done.\n");

        //WRITE UINT16 TO FLASH
        printf("Write Float to Flash\n");
        write_word_flash(PG31_BASE, 1, (uint32_t)uint16_flash);
        printf("Write done.\n");

        //WRITE UINT32 TO FLASH
        printf("Write Float to Flash\n");
        write_word_flash(PG31_BASE, 2, uint32_flash);
        printf("Write done.\n");

        //WRITE FLOAT TO FLASH
        printf("Write Float to Flash\n");
        write_float_flash(PG31_BASE, 3, float_flash);
        printf("Write done.\n");

        FLASH_Lock();  //lock flash
        FLASH_Unlock(); //unlock flash

        //WRITE INCREMENTAL PATTERNS
        FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_PGERR | FLASH_FLAG_WRPERR);
        FLASH_ErasePage(PG31_BASE); //erase flash

        for(uint32_t i=0;i<512;i++){ //Fill Flash Page with incremental patterns
                write_word_flash(PG31_BASE, i, (uint32_t)(uint32_flash2+i));
        }
        printf("Write done.\n");
        FLASH_Lock();  //lock flash

        while(1){}
}
```

Results:

Integer Values:

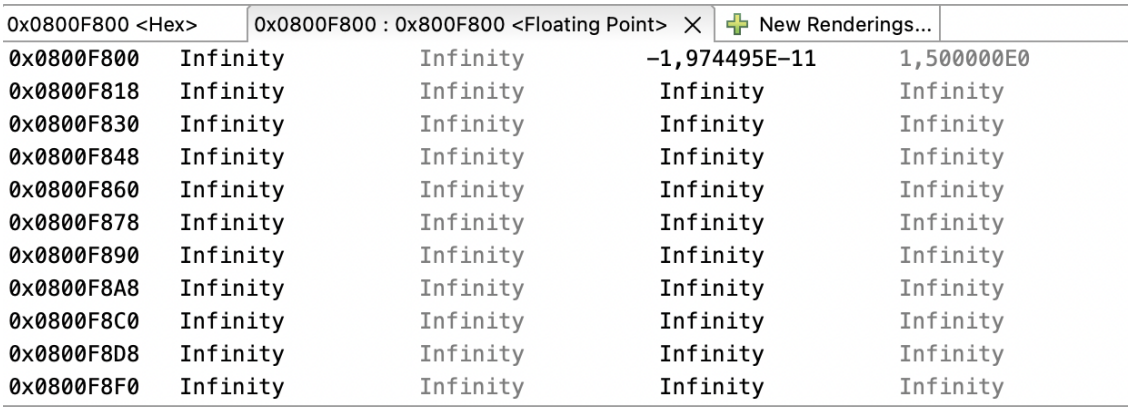
First column shows 8-bit Integer, second column the 16-bit Integer and the third one the 32-bit Integer variable:

0x0800F800 : 0x800F800 <Hex> | 0x0800F800 <Floating Point>

Address	0 - 3	4 - 7	8 - B	C - F
0800F800	AD000000	ADAD0000	ADADADAD	0000C03F
0800F810	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F820	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFF 0x800F
0800F830	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F840	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F850	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F860	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F870	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F880	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0800F890	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF

Float Value:

Fourth column shows the float value (1.5)



0x0800F800 <Hex> | 0x0800F800 : 0x800F800 <Floating Point> | New Renderings...

0x0800F800	Infinity	Infinity	-1,974495E-11	1,500000E0
0x0800F818	Infinity	Infinity	Infinity	Infinity
0x0800F830	Infinity	Infinity	Infinity	Infinity
0x0800F848	Infinity	Infinity	Infinity	Infinity
0x0800F860	Infinity	Infinity	Infinity	Infinity
0x0800F878	Infinity	Infinity	Infinity	Infinity
0x0800F890	Infinity	Infinity	Infinity	Infinity
0x0800F8A8	Infinity	Infinity	Infinity	Infinity
0x0800F8C0	Infinity	Infinity	Infinity	Infinity
0x0800F8D8	Infinity	Infinity	Infinity	Infinity
0x0800F8F0	Infinity	Infinity	Infinity	Infinity

Incremental Patterns:

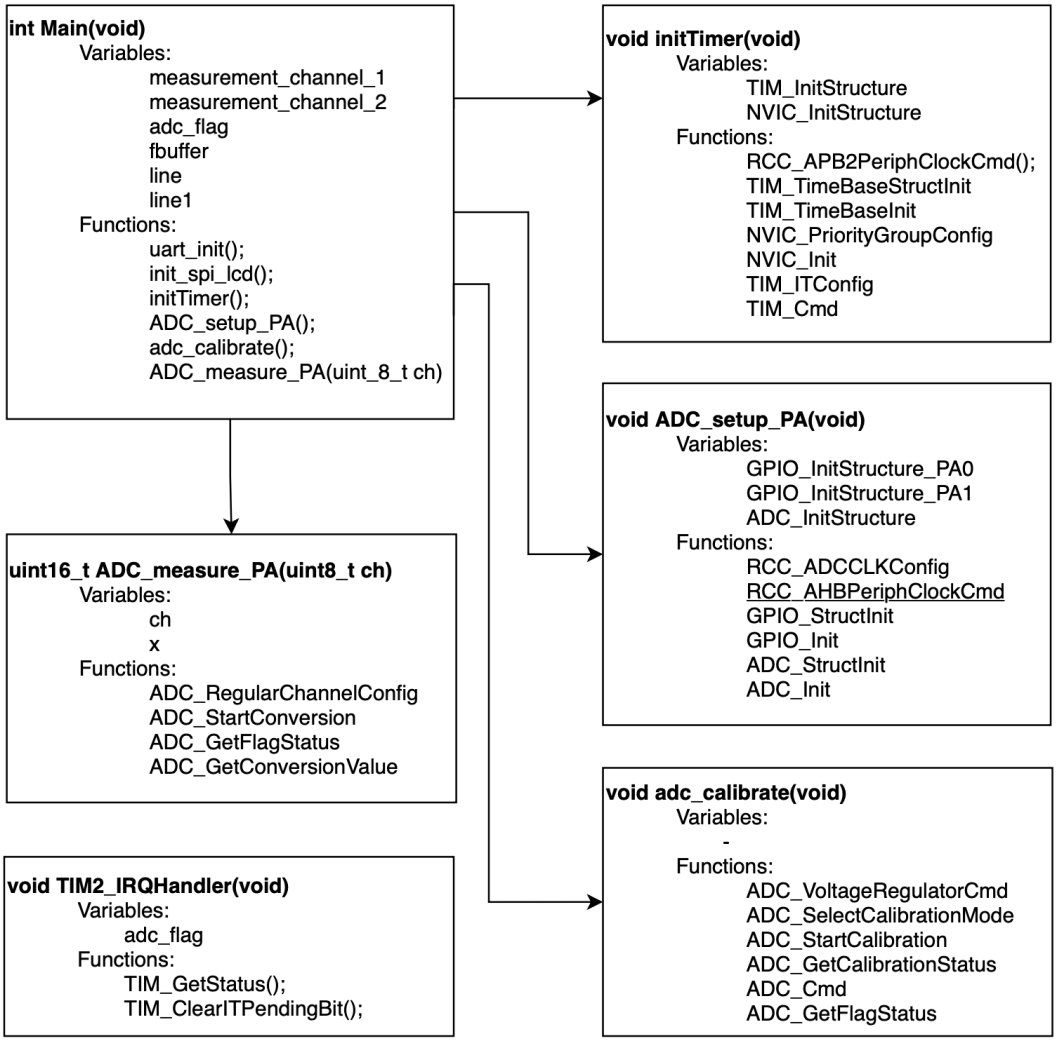
Memory page is completely filled with incrementing integer values.

0800F800	0	1	2	3
0800F810	4	5	6	7
0800F820	8	9	10	11
0800F830	12	13	14	15
0800F840	16	17	18	19
0800F850	20	21	22	23
0800F860	24	25	26	27
0800F870	28	29	30	31
0800F880	32	33	34	35
0800F890	36	37	38	39

4. Analog-to-Digital Conversion

Function description:

Block Diagram



main

Syntax `int main(void);`

Parameters -

The Main function first initializes the required peripherals: UART, SPI LCD, the timer for the ADC update frequency, and the ADC. Then the ADC is calibrated and activated. In the following while loop, the adc_flag is used to constantly check whether a timer interrupt has occurred. If this is the case, both potentiometers are measured with the ADC and their measured values are stored in the variables measurement_channel_1 and measurement_channel_2. Of course it might seem to be more convenient to do the measurement directly in the timer interrupt routine. However, this was avoided in our solution to keep the computing time within the ISR as low as possible. After the two

ADCs the measured values are displayed on the LCD. More detailed information about the program flow can be found in the flow chart diagram.

initTimer

Syntax `void initTimer(void);`

Parameters -

The timer2 of the controller is configured to generate a timer interrupt 10 times per second (10Hz). For this purpose a prescaler of 64,000 and a period of 100 was chosen.

ADC_setup_PA

Syntax `void ADC_setup_PA(void);`

Parameters -

This function configures the ADC. The parameters have been set according to the assignment description.

adc_calibrate

Syntax `void adc_calibrate(void);`

Parameters -

This function calibrates the ADC according to the assignment description.

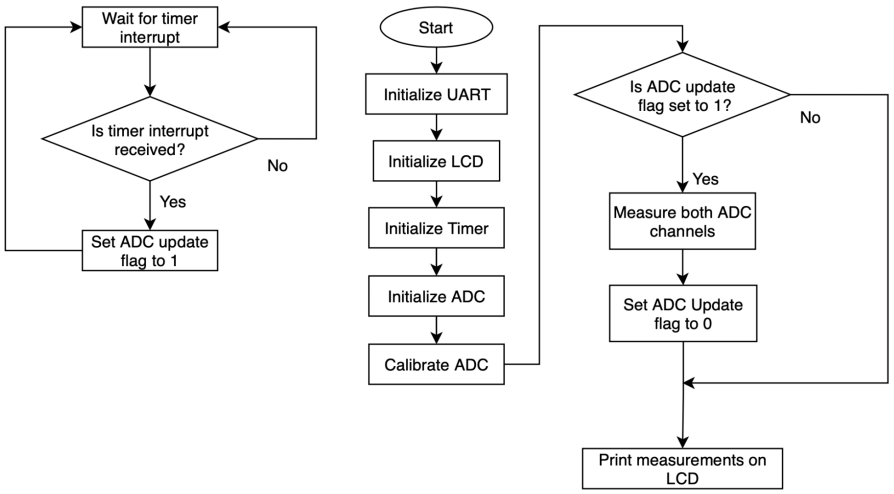
ADC_measure_PA

Syntax `uint16_t ADC_measure_PA(uint8_t ch);`

Parameters ch: ADC channel to be measured

This function reads either ADC channel 1 or 2 depending on the input. For this purpose the channel is collected for 1.5 clock cycles and then the measurement is started. As soon as this is completed, the measured value is output in the form of a 16-bit integer.

Flow Chart



Source Code:

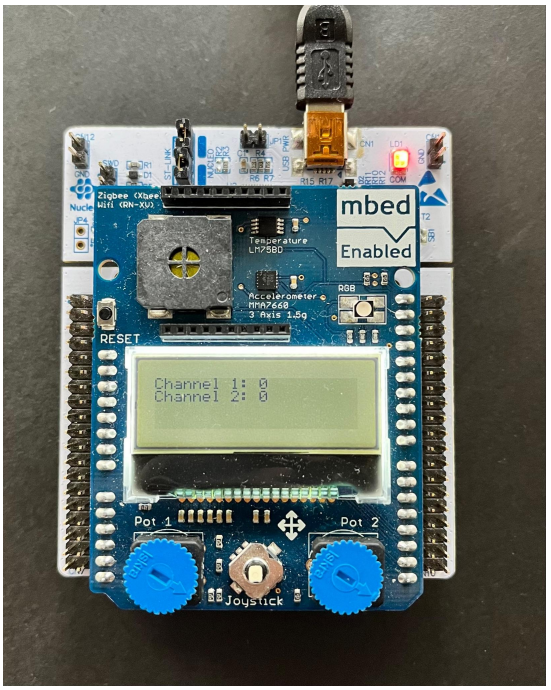
```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"               // Input/output library for this course

#include "lcd.h"
#include <string.h>

uint8_t adc_flag = 0;
uint16_t measurement_channel_1 = 0;
uint16_t measurement_channel_2 = 0;

void ADC_setup_PA(){
	RCC_ADCCLKConfig(RCC_ADC12PLLCLK_Div8); //set ADC prescaler to 8 --> ADC clock
frequency 8MHz
	RCC_AHBPeriphClockCmd(ADC1, ENABLE); // enable ADC interface

	//enable clock
	RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE); //enable clock GPIOA
	RCC_AHBPeriphClockCmd(ADC1, ENABLE);                    // enable clock for
ADC1

	//configure GPIO to ADC
	//PA0
	GPIO_InitTypeDef GPIO_InitStructure_PA0;
	GPIO_StructInit(&GPIO_InitStructure_PA0);
	GPIO_InitStructure_PA0.GPIO_Pin = GPIO_Pin_0 ;
	GPIO_InitStructure_PA0.GPIO_Mode = GPIO_Mode_AN;
	GPIO_InitStructure_PA0.GPIO_PuPd = GPIO_PuPd_NOPULL ;
	GPIO_Init(GPIOA, &GPIO_InitStructure_PA0);

	// PA1
	GPIO_InitTypeDef GPIO_InitStructure_PA1;
	GPIO_StructInit(&GPIO_InitStructure_PA1);
	GPIO_InitStructure_PA1.GPIO_Pin = GPIO_Pin_1 ;
	GPIO_InitStructure_PA1.GPIO_Mode = GPIO_Mode_AN;
	GPIO_InitStructure_PA1.GPIO_PuPd = GPIO_PuPd_NOPULL ;
	GPIO_Init(GPIOA, &GPIO_InitStructure_PA1);

	//configure ADC
	ADC_InitTypeDef ADC_InitStructure;
	ADC_StructInit(&ADC_InitStructure);
	ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; //single-conversion
	ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b; //12 bit-resolution
	ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; //allign data to the right
	ADC_Init(ADC1, &ADC_InitStructure);
}

void adc_calibrate()
{
    // Calibrate ADC
	ADC_VoltageRegulatorCmd(ADC1,ENABLE);   // set internal reference voltage source
and wait
	for(uint32_t i = 0; i<10000;i++);       //Wait for at least 10uS before
continuing...
```

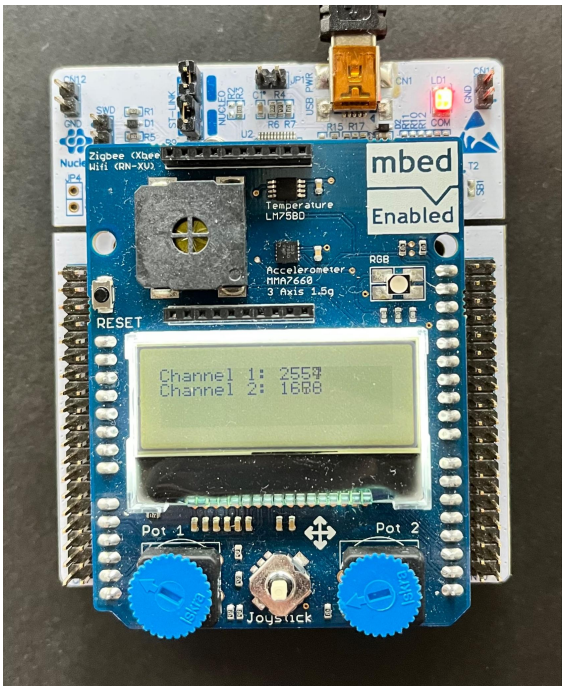
```
        ADC_SelectCalibrationMode(ADC1,ADC_CalibrationMode_Single);
        ADC_StartCalibration(ADC1);

        while(ADC_GetCalibrationStatus(ADC1)){}
        for(uint32_t i = 0; i<100;i++);

        // Enable ADC
        ADC_Cmd(ADC1,ENABLE);
        while(!ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY)){}
}

uint16_t ADC_measure_PA(uint8_t ch){
        if(ch == 1){
                ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_1Cycles5);
        }
        else{
                ADC_RegularChannelConfig(ADC1, ADC_Channel_2, 1, ADC_SampleTime_1Cycles5);
        }

        ADC_StartConversion(ADC1); // Start ADC read
        while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); // Wait for ADC read

        uint16_t x = ADC_GetConversionValue(ADC1); // Read the ADC value
        return(x);
}

void initTimer(){
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);

    // Timer Setup
    TIM_TimeBaseInitTypeDef TIM_InitStructure;
    TIM_TimeBaseStructInit(&TIM_InitStructure);
    TIM_InitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_InitStructure.TIM_Prescaler = 64000; // 64MHz / 64000 = 1kHz
    TIM_InitStructure.TIM_Period = 100; // 1kHz/100 = 10Hz
    TIM_TimeBaseInit(TIM2,&TIM_InitStructure);

    // NVIC Setup
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_Init(&NVIC_InitStructure);

    TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE);

    TIM_Cmd(TIM2,ENABLE);
}

void TIM2_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
```

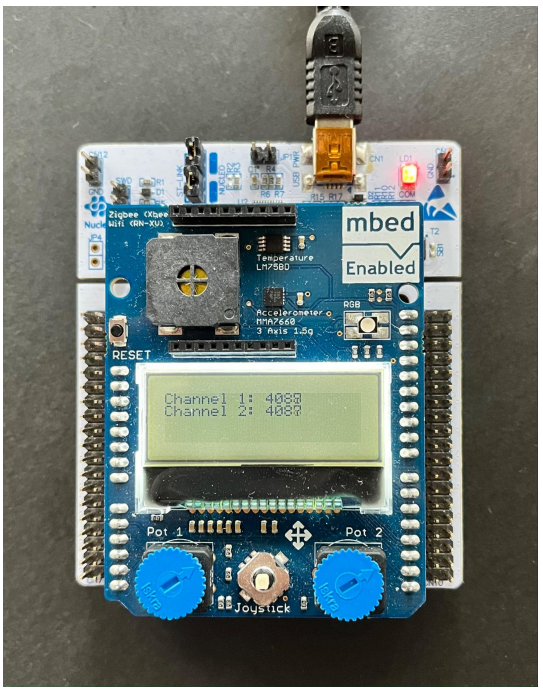
```
    {
        adc_flag = 1;
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    }
}

int main(void)
{
        uart_init(9600); // Initialize USB serial at 9600 baud
        printf("ADC Demo\n");

        //LCD
        init_spi_lcd(); //Initialize SPI LCD

        uint8_t fbuffer[512]; //Create local frame buffer
        memset(fbuffer, 0x00, 512);

    //Timer
    initTimer();
    printf("Timer Init done.\n");

    //ADC
        ADC_setup_PA(); //setup
        printf("ADC Init done.\n");

        adc_calibrate(); //calibrate
        printf("Calibration done.\n");

    ADC_Cmd(ADC1,ENABLE); //activate
    while(!ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY)){}
    printf("ADC enabled.\n");

        while(1){
                if(adc_flag == 1){
                        measurement_channel_1 = ADC_measure_PA(1);
                        measurement_channel_2 = ADC_measure_PA(2);
                        adc_flag = 0;
                }

                char line[128]; //Create character array for line 1
                sprintf(line, "Channel 1: %d", measurement_channel_1);

                char line1[128]; //Create character array for line 2
                sprintf(line1, "Channel 2: %d", measurement_channel_2);

                lcd_write_string((uint8_t*)line, fbuffer, 0, 0); //write line to frame
buffer
                lcd_write_string((uint8_t*)line1, fbuffer, 0, 1); //write line to frame
buffer

                lcd_push_buffer(fbuffer); //push frame buffer to lcd

                printf("1: %d, 2: %d\n", measurement_channel_1, measurement_channel_2);
        }
}
```

Results:

Both ADC channels are read out with a frequency of 10Hz and shown on the LCD display.

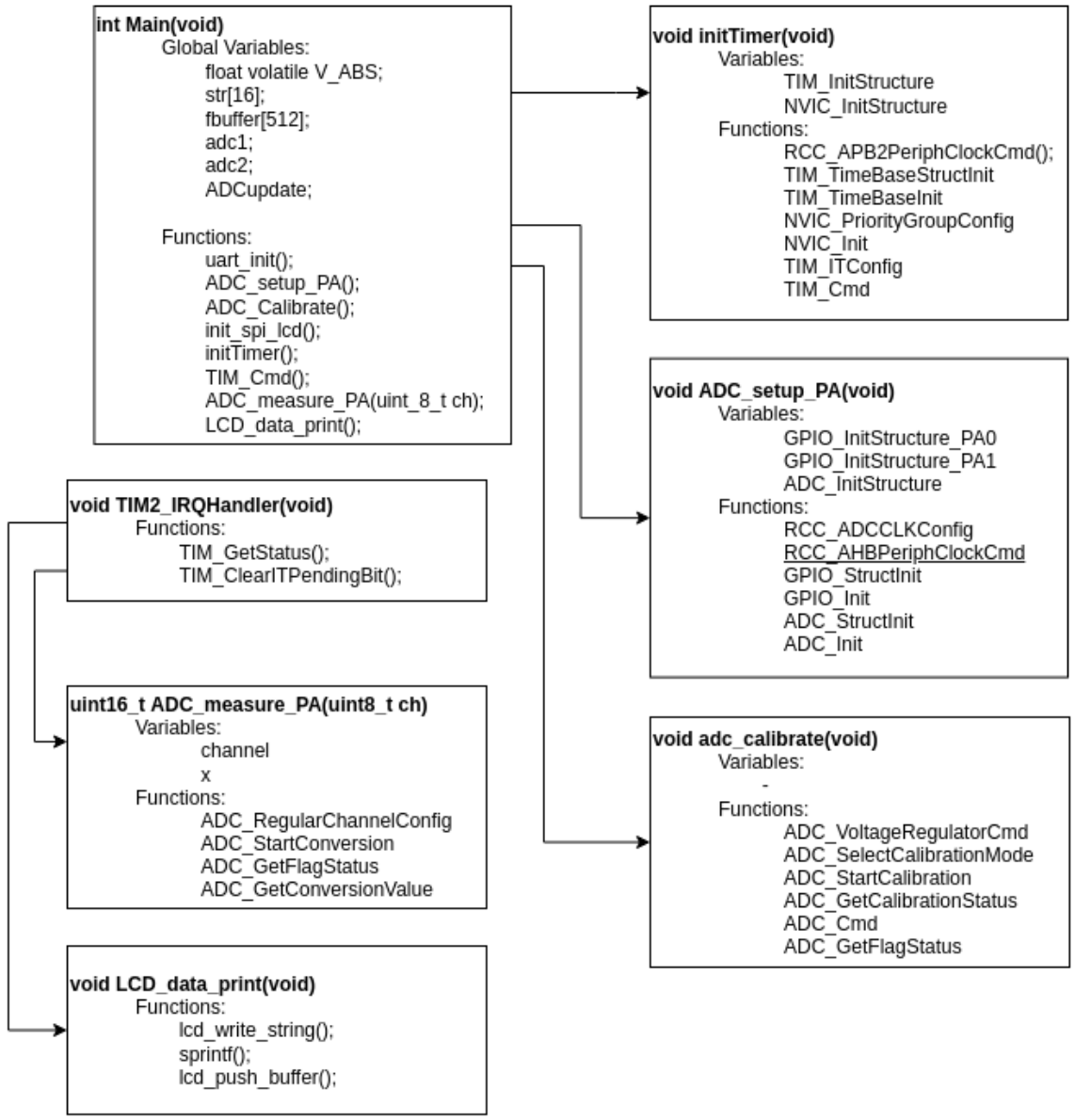




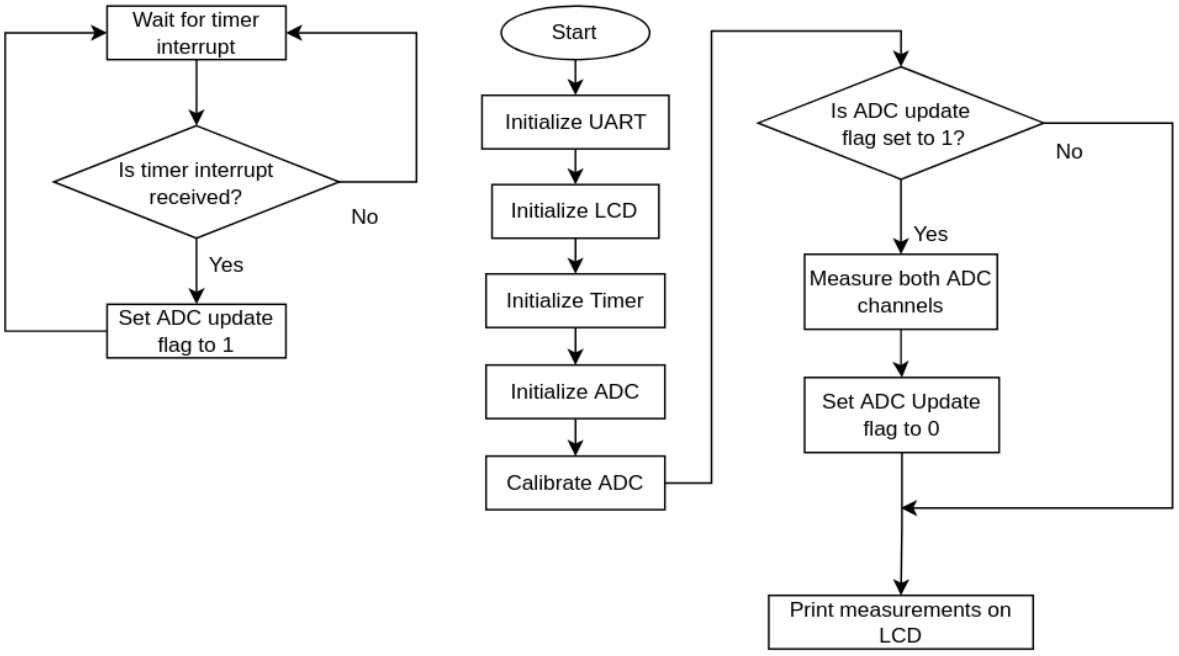


5. Absolute voltage from ADC measurement

Block diagram:



Flowchart:



Function Description:

initTimer

Syntax `void initTimer(void);`

Parameters -

The timer2 of the controller is configured to generate a timer interrupt 10 times per second (10Hz). For this purpose a prescaler of 64,000 and a period of 100 was chosen

ADC_setup_PA

Syntax `void ADC_setup_PA(void);`

Parameters -

This function configures the ADC. The parameters have been set according to the assignment description.

ADC_Calibrate

Syntax `void ADC_Calibrate(void);`

Parameters -

This function calibrates the ADC according to the assignment description.

ADC_measure_PA

Syntax `uint16_t ADC_measure_PA(uint8_t ch);`

Parameters ch: ADC channel to be measured

This function reads either ADC channel 1 or 2 depending on the input. For this purpose the channel is collected for 1.5 clock cycles and then the measurement is started. As soon as this is completed, the measured value is output in the form of a 16-bit integer.

TIM2_IRQHandler

Syntax `void TIM2_IRQHandler(void);`

Parameters -

This function Handles the timer interrupts and sets the ADC update flag to 0.

LCD_data_print

Syntax `void LCD_data_print(void);`

Parameters -

This function prints the ADC Absolute voltage to the lcd.

Source Code:

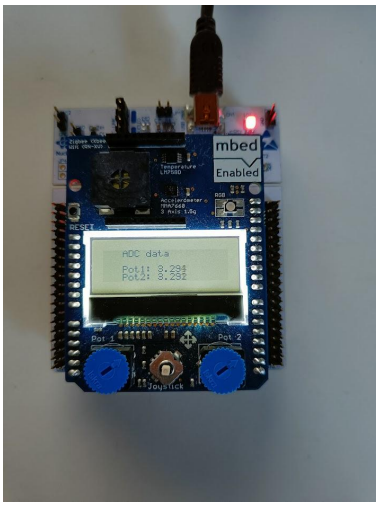
```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"         // Input/output library for this course
#include "flash.h"
#include "lcd.h"
#include <string.h>

#define VREFINT_CAL *((uint16_t*) ((uint32_t) 0x1FFFF7BA))
float volatile V_ABS;
char  str[16];
uint8_t fbuffer[512];
uint16_t volatile adc1;
uint16_t volatile adc2;
uint8_t volatile ADCupdate=0;

void initTimer(void){
      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
      NVIC_InitTypeDef NVIC_InitStructure;
      // NVIC for timer
      NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
      NVIC_Init(&NVIC_InitStructure);
      TIM_ITConfig(TIM2,TIM_IT_Update,ENABLE);
      TIM_Cmd(TIM2,ENABLE);
      //Settings timer
      RCC->APB1ENR |= RCC_APB1Periph_TIM2; // Enable clock line to timer
2
      TIM2->CR1=0xB01;
      TIM2->PSC=6399; //change pre-scaler frequency to 10kHz
      TIM2->ARR=999; //count up to 100
      TIM2->DIER |= 0x0001; // Enable timer 2 interrupts
    //NVIC settings
      NVIC_SetPriority(TIM2_IRQn, 1); // Set interrupt priority
```

```
interrupts
	NVIC_EnableIRQ(TIM2_IRQn); // Enable interrupt
	TIM_Cmd(TIM2,DISABLE);
}

void ADC_setup_pA(void){
	RCC_ADCCLKConfig(RCC_ADC12PLLCLK_Div8); //adc clk
	RCC_AHBPeriphClockCmd(RCC_AHBPeriph_ADC12, ENABLE); //enable adc
	RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE); //gpio clock

	GPIO_InitTypeDef GPIO_InitStructAll; // Define typedef struct for
setting pins

	GPIO_StructInit(&GPIO_InitStructAll); // Initialize GPIO struct
	GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_AN; // Set as input
	GPIO_InitStructAll.GPIO_PuPd = GPIO_PuPd_DOWN; // Set as pull down
	GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_0; // Set so the
configuration is on pin 4
	GPIO_Init(GPIOA, &GPIO_InitStructAll); // Setup of GPIO with the
settings chosen

	GPIO_StructInit(&GPIO_InitStructAll); // Initialize GPIO struct
	GPIO_InitStructAll.GPIO_Mode = GPIO_Mode_AN; // Set as input
	GPIO_InitStructAll.GPIO_PuPd = GPIO_PuPd_DOWN; // Set as pull down
	GPIO_InitStructAll.GPIO_Pin = GPIO_Pin_1; // Set so the
configuration is on pin 4
	GPIO_Init(GPIOA, &GPIO_InitStructAll); // Setup of GPIO with the
settings chosen

	ADC_InitTypeDef ADC_InitStructAll; //struct for adc config

	ADC_StructInit(&ADC_InitStructAll); //settings for the adc
	ADC_InitStructAll.ADC_ContinuousConvMode = DISABLE;
	ADC_InitStructAll.ADC_Resolution = ADC_Resolution_12b;
	ADC_InitStructAll.ADC_ExternalTrigEventEdge =
ADC_ExternalTrigEventEdge_None;
	ADC_InitStructAll.ADC_DataAlign = ADC_DataAlign_Right;
	ADC_InitStructAll.ADC_NbrOfRegChannel = 1;
	ADC_Init(ADC1,&ADC_InitStructAll); // init the adc settings
	ADC_Cmd(ADC1,ENABLE); //enable adc
	// set internal reference voltage source and wait
}

void ADC_Calibrate(){
	ADC_VoltageRegulatorCmd(ADC1,ENABLE);
	//Wait for at least 10uS before continuing...
```

```
	for(uint32_t i = 0; i<10000;i++);

	ADC_Cmd(ADC1,DISABLE);
	while(ADC_GetDisableCmdStatus(ADC1)){} // wait for disable of ADC

	ADC_SelectCalibrationMode(ADC1,ADC_CalibrationMode_Single);
//select calibration mode
	ADC_StartCalibration(ADC1); //calibrate adc
	while(ADC_GetCalibrationStatus(ADC1)){} //wait for calibration
	for(uint32_t i = 0; i<100;i++);//wait more

	ADC_VrefintCmd(ADC1,ENABLE); // setup ref voltage to channel 18
	for(uint32_t i = 0; i<10000;i++); // wait for some time

	ADC_Cmd(ADC1,ENABLE);// turn on ADC
	while((!ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY))){ } //wait for adc to
turn on

	ADC_RegularChannelConfig(ADC1, ADC_Channel_18, 1,
ADC_SampleTime_19Cycles5); //wait for 2.2us
	ADC_StartConversion(ADC1); // Start ADC read
	while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); // Wait for
ADC read

	uint16_t VREFINT_DATA = ADC_GetConversionValue(ADC1); // save
measured data
	V_ABS = ((3.3 * (VREFINT_CAL / VREFINT_DATA)) / 4095); //
calculate the voltage/adc step
}

uint16_t ADC_measure_PA(uint8_t channel){
	uint16_t x;
	ADC_RegularChannelConfig(ADC1, channel, 1,
ADC_SampleTime_1Cycles5);
	ADC_StartConversion(ADC1); // Start ADC read
	while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); // Wait for
ADC read
	x = ADC_GetConversionValue(ADC1) ; // savemeasured data
	return x;
}

void TIM2_IRQHandler(void) { //timer interrupt handler
	if(TIM_GetITStatus(TIM2,TIM_IT_Update) != RESET){ //if interrupt
occurs
		TIM_ClearITPendingBit(TIM2,TIM_IT_Update); // Clear
interrupt bit
```

```
            ADCupdate=1;
      }
}

void LCD_data_print(void){
      lcd_write_string((uint8_t*)"ADC data", fbuffer, 20, 0);
      sprintf(str,"Pot1: %0.3f",(double)adc1* (double)V_ABS);
      lcd_write_string(str, fbuffer, 20, 2);
      sprintf(str,"Pot2: %0.3f",(float)adc2* (float)V_ABS);
      lcd_write_string(str, fbuffer, 20, 3);
      lcd_push_buffer(fbuffer);
}

int main(void)
{
      uart_init(9600);
      ADC_setup_pA();
      ADC_Calibrate();
      initTimer();
      init_spi_lcd();
      memset(fbuffer,0x00,512); // Sets each element of the buffer to
0xAA
      lcd_push_buffer(fbuffer);
      TIM_Cmd(TIM2,ENABLE);
      while(1){
            if(ADCupdate == 1){
                  adc1 = ADC_measure_PA(1);
                  adc2 = ADC_measure_PA(2);
                  LCD_data_print();
                  ADCupdate=0;
            }
      }
}
```

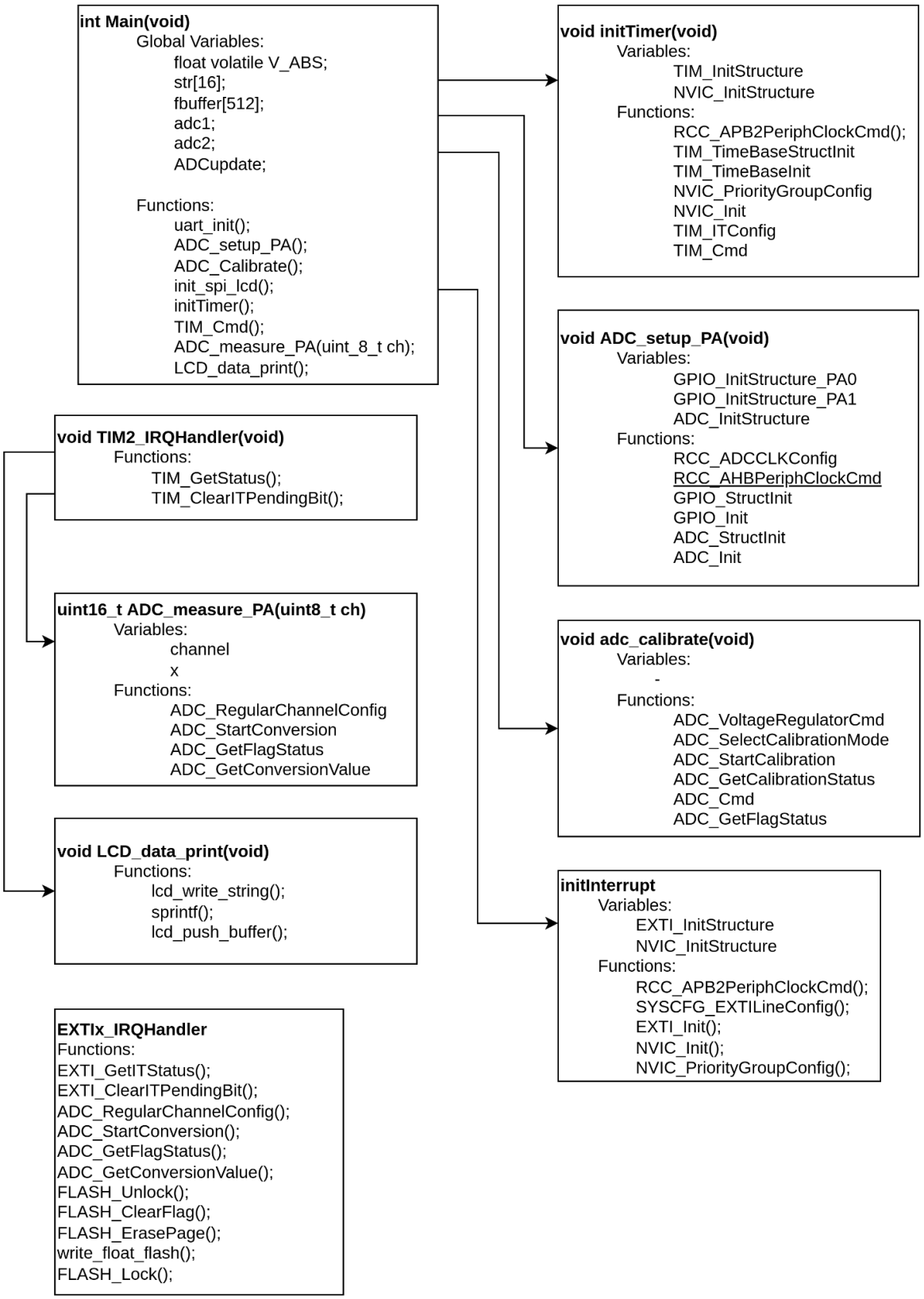
Results:



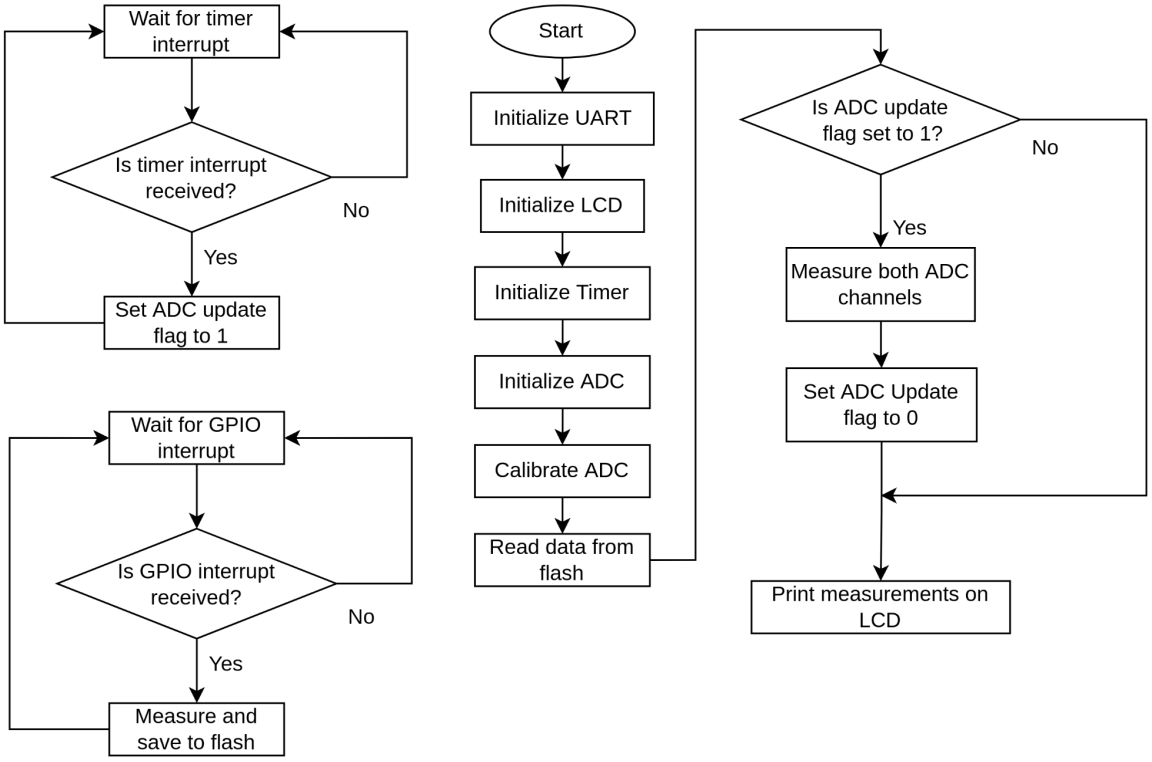
In the pictures, the potentiometer was set to 3 different positions (left, center, right). The measured voltages are printed to the LCD with a 10Hz refresh rate.

6. Calibrate the ADC

Block Diagram:



Flowchart:



Compared to the previous exercise only one function and a flash read code snippet in was added. The journal will only focus on them because the overall functionality was not modified.

Function Description:

EXTI9_5_IRQHandler

Syntax `void EXTI9_5_IRQHandler(void);`

Parameters -

This function reads both ADC channels 16 times then calculates the average for each channel. After the average was calculated the data is stored in the flash. The function is only triggered once while the channels are being measured with a bench multimeter.

Source Code:

```
void EXTI9_5_IRQHandler(void){ // for pausing timer
      if(EXTI_GetITStatus(EXTI_Line5) != RESET){
            for(uint8_t j =0;j<16;j++){

                  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1,
ADC_SampleTime_1Cycles5);
                  ADC_StartConversion(ADC1); // Start ADC read
                  while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); //
Wait for ADC read
```

```
                x_global += (float)ADC_GetConversionValue(ADC1) ; //
save measured data

                ADC_RegularChannelConfig(ADC1, ADC_Channel_2, 1,
ADC_SampleTime_1Cycles5);
                ADC_StartConversion(ADC1); // Start ADC read
                while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); //
Wait for ADC read
                y_global += (float)ADC_GetConversionValue(ADC1) ; //
save measured data
            }
            x_global = x_global/16*V_ABS;
            y_global = y_global/16*V_ABS;

            FLASH_Unlock(); //unlock flash
            FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_PGERR |
FLASH_FLAG_WRPERR);
            FLASH_ErasePage(PG31_BASE); //erase flash
            write_float_flash(PG31_BASE, 0, x_global);
            write_float_flash(PG31_BASE, 1, y_global);
            FLASH_Lock();  //lock flash
            EXTI_ClearITPendingBit(EXTI_Line5);
      }
}
```

Reading from the flash:

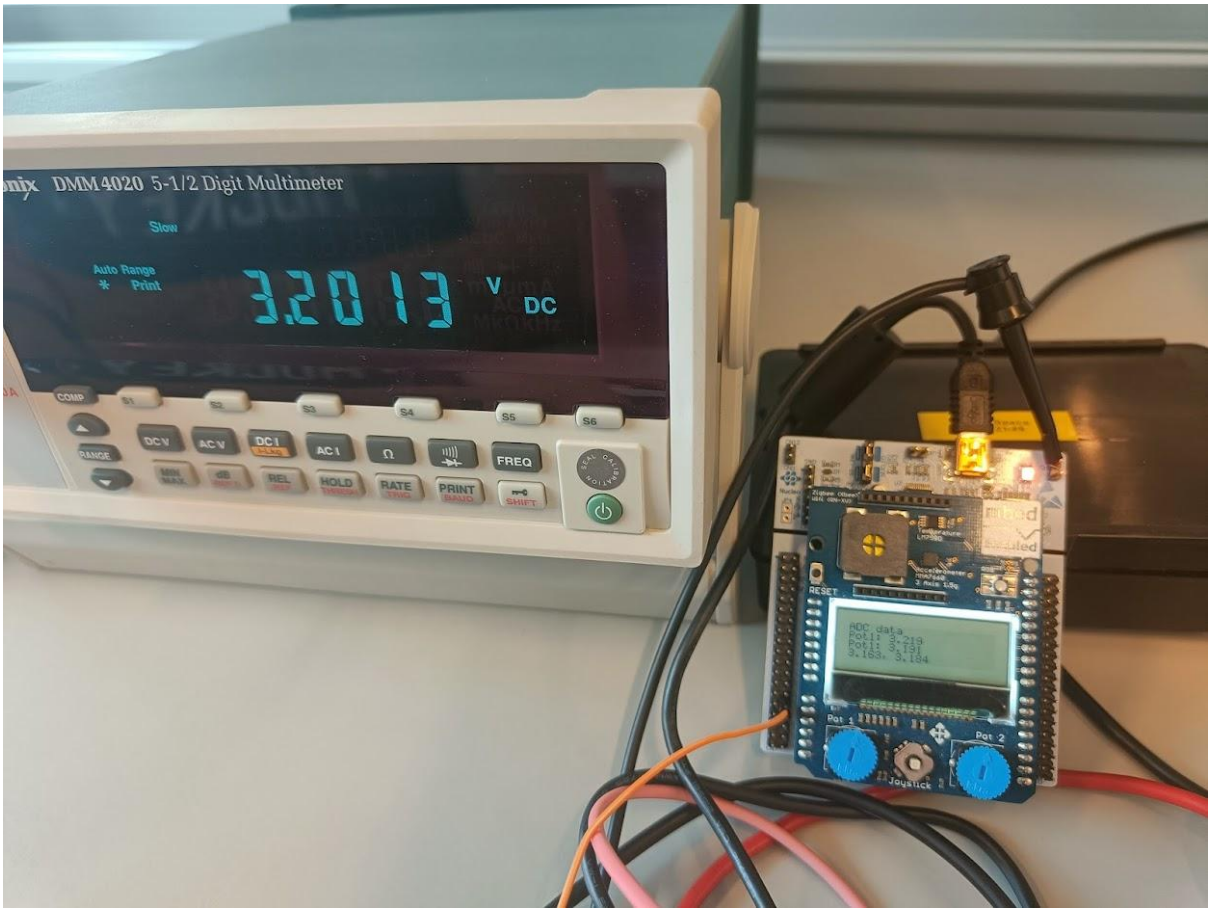
```
uint32_t address = 0x0800F800;
float tempVal;
for ( int i = 0 ; i < 2 ; i++ ){
tempVal = *(float *)(address + i * 4); // Read Command
printf("%f \n", tempVal);
}
x_global = *(float *)(address + 0 * 4);
y_global = *(float *)(address + 1 * 4);
```

The modified print function for the calibrated data:

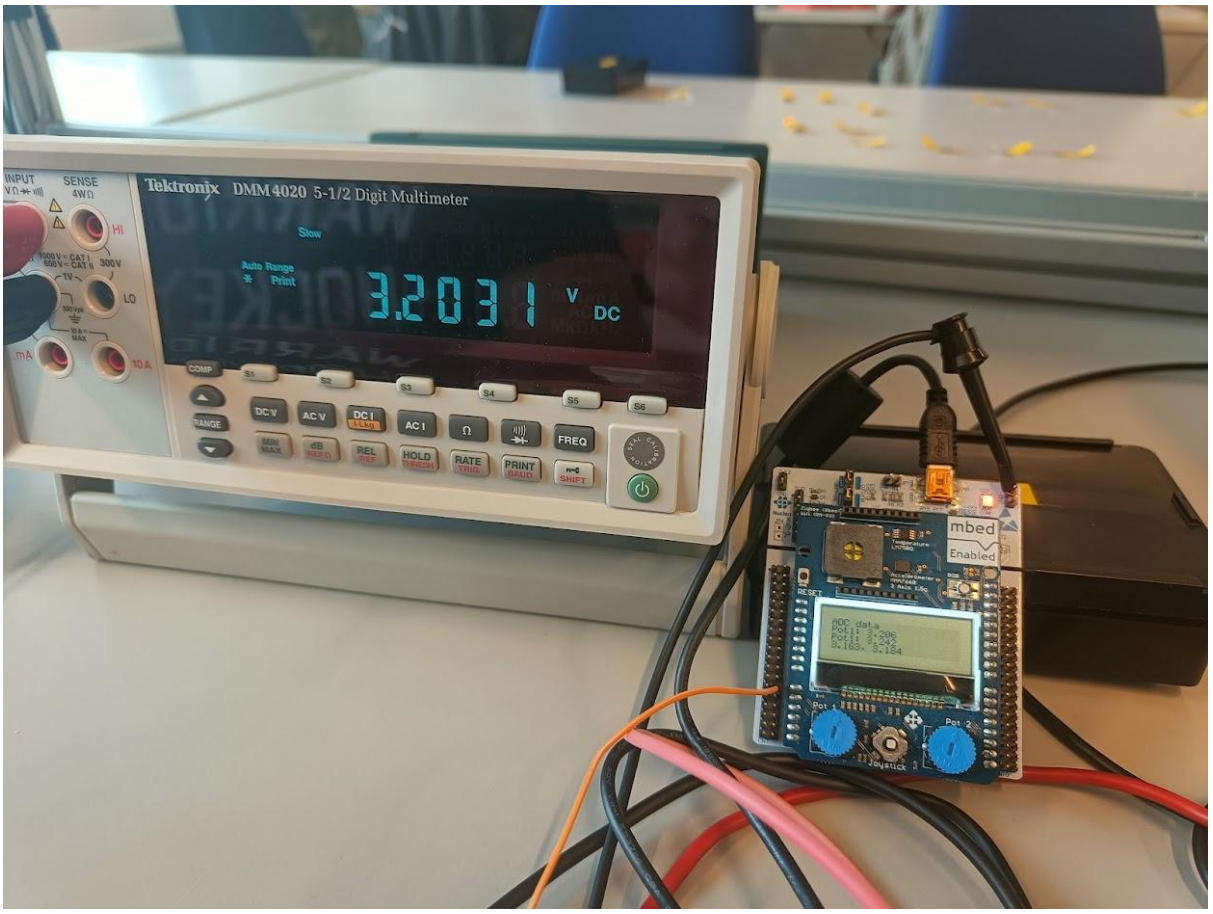
```
void LCD_data_print(void){
      lcd_write_string((uint8_t*)"ADC data", fbuffer, 20, 0);
      sprintf(str,"Pot1: %0.3f",(double)adc1*
(double)V_ABS*((double)3.2/(double)x_global));
      lcd_write_string(str, fbuffer, 20, 2);
      sprintf(str,"Pot2: %0.3f",(float)adc2*
(float)V_ABS*((double)3.2/(double)y_global));
      lcd_write_string(str, fbuffer, 20, 3);
      lcd_push_buffer(fbuffer);
}
```

Results:

Channel 1:



Channel 2:



We measured both channels after the calibration with the external multimeter and the results were great. The calibrated values were within the ±20mV range, which is below 1%.