Laboratory manual

Embedded Systems Laboratory

UbiLAB

Универзитет "Св. Кирил и Методиј" во Скопје
ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ

Hochschule Anhalt
Anhalt University of Applied Sciences

University of Maribor

Erasmus+

NATIONAL AGENCY
for European Educational
Programmes and Mobility

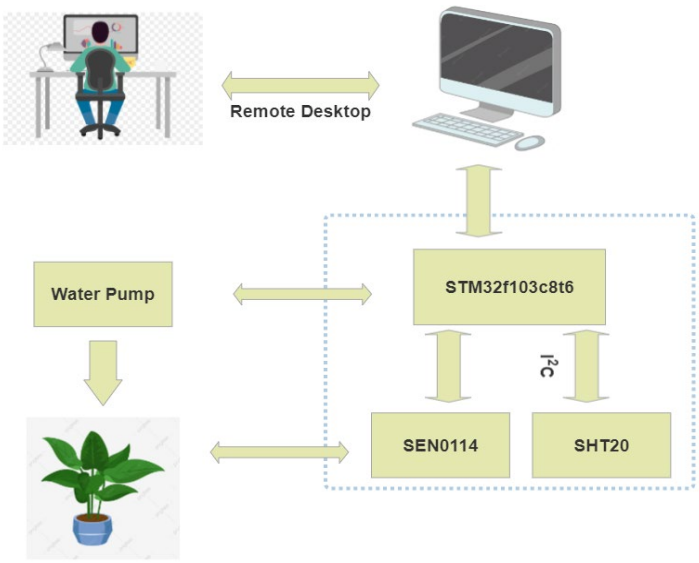
5 Embedded Systems Laboratory

5.1 Introduction

Remote Embedded laboratory is a remote laboratory that is part of the UbiLAB framework. The ultimate purpose of this remote laboratory is to provide a unique learning experience for students and enthusiasts interested in the field of embedded systems and microcontroller programming. The course consists of five laboratory exercises each designed to be done in a single session. The topics that are covered in this course are: General purpose I/O ports, Timers for general purpose, analog/digital convertor (ADC), universal asynchronous/synchronous receiver/transmitter (USART) and inter-integrated circuits (I2C). The combination of all five exercises leads to an autonomous irrigation system project. Students can access the connected system through the Apache Guacamole framework embedded into the LMS course from anywhere with internet connection. When the connection is established, they can control the hardware setup. The user has the freedom to choose between Arduino IDE and CooCox for the software development environment. Additionally, Hercules SETUP software is used to interface with the Serial monitor. After uploading and running the program, results can be observed through the Serial monitor and the live video stream.

The hardware solution consists of a personal computer, STM32f103c8t6 microcontroller, LED diodes that are connected on three different digital pins of the microcontroller, SEN0114 soil moisture sensor that is connected to an analog pin, SHT20 - temperature and humidity sensor connected to I2C compatible pins. Additionally, USB to TTL Adapter is connected between the microcontroller and personal computer for USART communication for serial print of the outputs and a video camera for observation of all visible outputs. This remote lab setup allows real-time interaction with the hardware solution. Students can send commands, read sensor data and observe the responses in real-time through a user-friendly interface. In this way students can understand the concepts better and verify the expected behavior of embedded systems.

The laboratory setup and the remote access is presented in Figure 5-1.



5.2 Laboratory experiment 1

The first laboratory experiment consists of a simple code that configures a selected I/O port which turns a LED diode connected through a 220 ohms resistor on or off depending on the set logic level. Additionally, two more diodes can be added in order to simulate the operation of a traffic light by using a for loop delay.

GPIO stands for General Purpose Input/Output. It is a type of interface that allows an electronic device, such as a microcontroller or a processor, to communicate with external devices by providing digital input/output signals. GPIO pins can be configured as either input or output pins, depending on the system requirement. When configured as an input, the GPIO pin can read the state of external devices, such as sensors or switches, by detecting changes in the voltage level on the pin. On the other hand, when configured as an output pin, GPIO can control the state of external devices, turning LEDs on/off or driving a motor, by outputting a voltage level on the pin.

In order to use the STM GPIO pins, first we have to configure them. The GPIO configuration consists of the following steps:

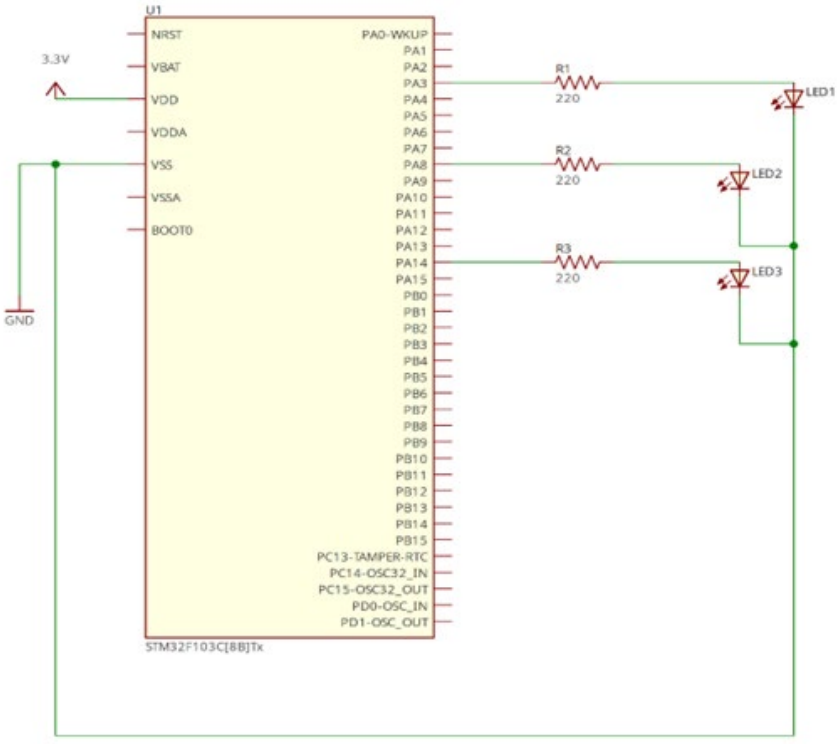
- Enable the peripheral clock
- Specify the GPIO pins to be configured
- Specify the speed for the configured pins
- Specify the operating mode for the configured pins
- Remap the pin if an alternate function is used

5.2.1 Experimental setup

Equipment:

- Embedded system development board (STM32f103c8t6)
- Three LEDs (red, green and yellow color)
- Three 220 ohm resistors
- Breadboard and jumper wires

The anode (positive lead) of the red LED is connected to pin PA3 of the development board and the cathode (negative lead) is connected to one end of the 220 ohm resistor. The other end of the resistor is connected to the GND pin of the STM. These connections are repeated for the yellow and green diode by connecting them to pins PA8 and PA14 respectively, as shown in Figure 5-2.



5.2.2 Goals of the experiment

The goal of the first exercise is to get started and become familiar with the tools and environment, as well as understanding the general purpose I/O ports.

5.2.3 Experimental results

Write a program in C language to configure and control the GPIO pins. Your program should do the following:

- Initialize pins PA3, PA8 and PA14 as output pins. In order to do that the peripheral clock must be enabled, select the mode of the pin as output and select the GPIO speed.
- Turn on the LED diodes on and off by using the GPIO_WriteBit() function.
- Insert a delay by using a for loop for longer on and off time.

The code is given below. For additional exercise, students can write a program to simulate the operation of traffic light by implementing multiple for loop delays.

```
#include<stm32f10x_gpio.h>
#include<stm32f10x_rcc.h>

GPIO_InitTypeDef GPIO_InitSTruct;
int led=0;

int main(void){

	RCC_APB2PeriphClock (RCC_APB2Periph_GPIOA, ENABLE);
	GPIO_InitSTruct.GPIO_Pin=GPIO_Pin_1;
	GPIO_InitSTruct.GPIO_Mode=GPIO_Mode_Out_PP;
	GPIO_InitSTruct.GPIO_Speed=GPIO_Speed_50MHz;
	GPIO_Init (GPIOA, &GPIO_InitSTruct);

}
while(1){
	GPIO_WriteBit (GPIOA, GPIO_Pin_1, led);
	for(int i=0;i<900000;i++){
		led=~led;
	}}
```

5.3 Laboratory experiment 2

This experiment begins with a brief overview of the operation of integrated timers and prescaler. In its most basic form timer modules represent a digital logic circuit that counts up every clock cycle. More functionalities are implemented in hardware to support the timer module so it can count up or down. The timer module can have a prescaler to divide the input clock frequency by a selectable value.

STM32 microcontrollers have a variety of timer modules that can be used to generate interrupts, measure time intervals and perform other time-based operations. Here are some of the most common timer modules of the STM32 microcontroller family:

- Basic Timers (TIM6 and TIM7): simple timers that can generate interrupts with a frequency up to 84 MHz They can be used for timekeeping, periodic events and software timing loops.
- General Purpose Timers (TIM2, TIM3, TIM4, TIM5, TIM9, TIM10, TIM11): versatile timers that can be configured for a wide range of timing and control applications.

They support input capture, PWM (Pulse Width Modulation) modes, output compare and other modes of operation.

- Advanced Control Timers (TIM1, TIM8): high-end timers that support advanced features such as motor control and high-resolution PWM. These timers operate at frequencies up to 168 MHz and offer multiple channels for advanced control applications.
- Low-Power Timer (LPTIM1, LPTIM2): timers that operate in ultra-low-power modes for extended battery use.

These different hardware timers in the STM32 microcontroller can operate in multiple modes. An STM32 timer module can operate in any of the following modes: timer mode, counter mode, PWM mode, advanced PWM mode, output compare mode, input compare mode etc.

5.3.1 Experimental setup

The setup for this experiment is the same as in experiment 1.

Equipment:

- Embedded system development board (STM32f103c8t6)
- Three LEDs (red, green and yellow color)
- Three 220 Ω resistors
- Breadboard and jumper wires

The anode (positive lead) of the red LED is connected to pin PA3 of the development board and the cathode (negative lead) is connected to one end of the 220 Ω resistor. The other end of the resistor is connected to the GND pin of the STM board. These connections are repeated for the yellow and green diodes by connecting them to pins PA8 and PA14 respectively, as shown in Figure 5-3.

5.3.2 Goals of the experiment

In this experiment, students will learn how to program the timers and PWM of an embedded system to control the brightness of an LED.

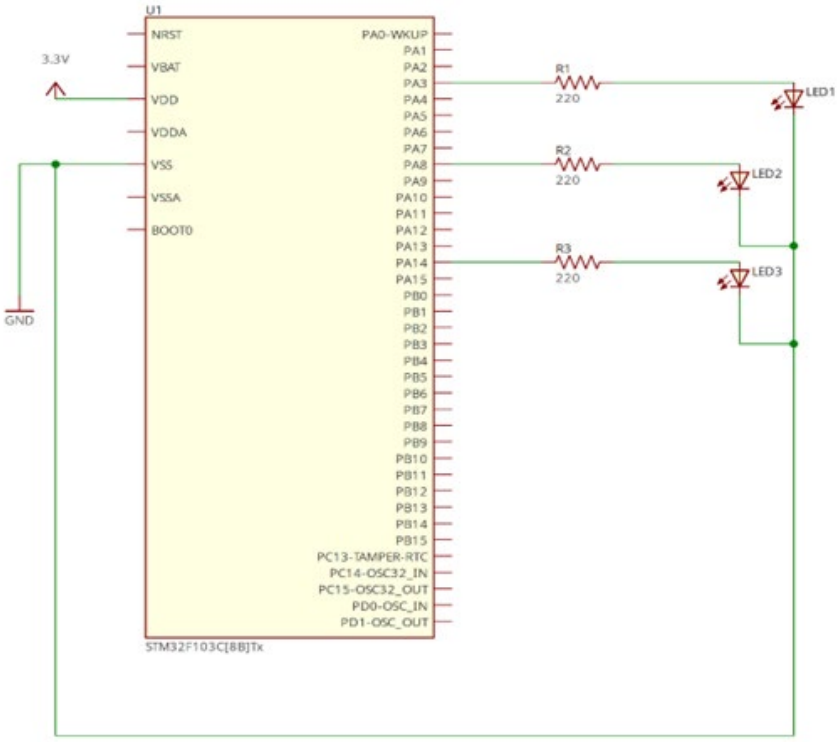


Figure 5-3. Experiment 2 connection schematic.

5.3.3 Experimental results

Write a program in C language to control and configure the timers and PWM:

- Turn on the clock for timer 2 (TIM2).
- Set the prescaler value to 71 and set the auto-reload register of TIM2 to 999. With this the timer will count till it reaches 999.
- Generate an update event to update the prescaler buffer.
- Reset the interrupt flag and turn on the timer.
- Write a loop to count the given time. Lastly, turn-off the timer.

An example code is given below. In the main program write a code to simulate traffic light operation by using the delay function.

```
void delay_ms (int ms)
{
	RCC -> APB1ENR |= 0x1;
	TIM2 -> PSC = 72-1;
	TIM2 -> ARR = 1000-1;
	TIM2 -> CNT = 0;
	TIM2 -> EGR |= 0x1;
	TIM2 -> SR &= ~(0x1);
	TIM2 -> CR1 |= 0x1;
```

```
        while (ms > 0)
        {
              while ((TIM2 -> SR & 0x1) == 0);
                    TIM2 -> SR &= ~(0x1);
                    ms--;
              }
        TIM2 -> CR1 &= ~(0x1);
        RCC -> APB1ENR &= ~(0x1 );
}
```

5.4 Laboratory experiment 3

In experiment 3 students will write a code to configure and read the ADC (Analog to Digital Converter) and use a development board (STM32) to test the program. The STM32 series of microcontrollers have a built-in ADC module that can be used to convert analog signals into digital for processing in software. Some of the key features of the SMT32 ADC are:

- 12 bits resolution
- 16 multiplexed channels
- Conversion modes: continuous conversion, single conversion and scan conversion.
- DMA (Direct Memory Access) support: the ADC can be configured to use DMA to transfer data from the ADC buffer to memory without CPU intervention.
- Trigger modes: software trigger, external trigger and timer trigger.
- Sampling rates: ranging from a few samples per second to several mega-samples per second.

To use the STM32 ADC module, the ADC registers have to be configured using a programming language and then start the conversion by a software or hardware trigger. Once the conversion is complete, the converted digital value can be read from the ADC data register and processed in software.

The SEN0114 moisture sensor, shown in Figure 5-4, is a capacitive soil moisture sensor that can be used to measure the water content in soil. The two metal probes that are inserted into the soil pass current through the soil and then the sensor measures the resistance between them to get the relative moisture level. The resistance between the probes varies depending on the water content in the soil, higher water level results in lower resistance and vice versa. The sensor has three pins: VCC, GND and SIG. The meaning of the measured values is given in Table 5-1.

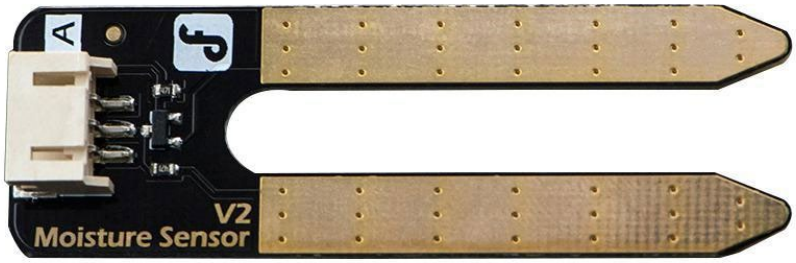


Figure 5-4. The SEN0114 moisture sensor.

Table 5-1. Meaning of measured values of SEN0114.

Measured values	Meaning
0-500	Very dry
500-1200	Good
>1200	Wet

5.4.1 Experimental setup

Equipment:

- Embedded system development board (STM32f103c8t6)
- SEN0114 moisture sensor
- Breadboard and jumper wires

The moisture sensor is connected to the development board as follows:

- Connect pin VCC to 3.3 V or 5 V
- Connect pin GND to ground
- Connect pin SIG to GPIO pin A5

5.4.2 Goals of the experiment

The goal of the experiment is to convert the measured value from analog to digital form and to store this information in the microcontroller memory.

5.4.3 Experimental results

Write a program in C language to configure and read the ADC. Your program should do the following:

- Configure GPIO pin A5 as an analog input. Enable the peripheral clock and set the GPIO speed.
- Set ADC mode as continuous and disable the external trigger.
- Read the analog voltage value from the ADC data register after each conversion. The voltage value corresponds to the resistance value of the SEN0114 moisture sensor value.
- Convert the analog voltage value to the corresponding moisture content using the calibration equation or lookup table, which can be found in the sensor datasheet.

The code for initialization and reading the measured value is given bellow.

```
#include"moisture_sensor.h"

GPIO_InitTypeDef GPIO_ADC_InitStruct;
void adc_init(void){

	RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);
	GPIO_ADC_InitStruct.GPIO_Pin=GPIO_Pin_5;
	GPIO_ADC_InitStruct.GPIO_Mode=GPIO_Mode_AIN;
	GPIO_ADC_InitStruct.GPIO_Speed=GPIO_Speed_50MHz;
	GPIO_Init(GPIOA, &GPIO_ADC_InitStruct);
	GPIO_PinRemapConfig(GPIO_Remap_ADC1_ETRGREG, ENABLE);
	ADC_DeInit(ADC1);
	RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1,ENABLE);
	RCC_ADCCLKConfig(RCC_PCLK2_Div4);

	ADC_InitStruct.ADC_Mode==ADC_Mode_Independent;
	ADC_InitStruct.ADC_ContinuousConvMode=ENABLE;
	ADC_InitStruct.ADC_ScanConvMode=DISABLE;
	ADC_InitStruct.ADC_NbrOfChannel=1;
	ADC_InitStruct.ADC_DataAlign=ADC_DataAlign_Right;
	ADC_InitStruct.ADC_ExternalTrigConv=ADC_ExternalTrigConv_None;
	ADC_Init(ADC1,&ADC_InitStruct);

	ADC_RegularChannelConfig(ADC1,ADC_Channel_5,1,ADC_SampleTime_1Cycles5);
	ADC_Cmd(ADC1,ENABLE);
	ADC_ResetCalibration(ADC1);
	while(ADC_GetResetCalibrationStatus(ADC1));
	ADC_StartCalibration( ADC1);
	ADC_SoftwareStartConvCmd(ADC1, ENABLE);


}
uint16_t read_adc_value(ADC_TypeDef* ADCx){
	uint16_t value=ADC_GetConversionValue(ADC1);
	return value;
}
```

5.5 Laboratory experiment 4

In experiment 4, we will learn about Universal synchronous asynchronous receiver transmitter (USART). This is serial communication that uses two wires, one for transmitting and the other one for receiving messages. Tx (transmit) pin is used for transmission of data. This pin is connected to Rx (receive) pin. Rx pin is used for receiving messages, as shown in Figure 5-5. Data is sent and received as serial bit stream of 7 or 8 bits at programmed rate. This experiment is a continuation of the previous experiment. The measured values, from the previous experiment, stored on the microcontroller memory are sent and displayed on the serial monitor using the USART module and Hercules SETUP software.

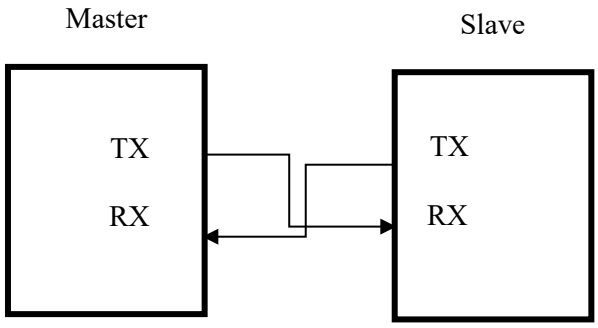


Figure 5-5. USART master-slave connection.

Frame Format

The asynchronous data format consists of a start bit, 7 or 8 bits for data, even/odd or no parity bit and one or two bits for stop condition.

USART transmission

Start bit

The communication starts with the start bit. Tx line is set to high level when it is not transmitting data. When start is requested, the Tx line is switched to low level for one clock cycle. When the transition is detected by receiver, it starts to read data frame.

Data frame

Data frame is 7 or 8 bits. Usually the least significant bit is sent first.

Parity bit

Parity bit is used to check the reliability of data. The bit can be changed during data transfers. When data is received it checks the number of bits with value of 1. If the parity bit is set to even parity the number of 1's in the data should be even or if parity bit is set to odd parity, then the number of 1's should be an odd number.

Stop bit

Stop bit is used to force the communication to end. This is realized when transiter switches Tx line to high level.

5.5.1 Experimental setup

Equipment:

- Embedded system development board (STM32f103c8t6)
- Breadboard and jumper wires
- SEN0114 moisture sensor
- Hercules SETUP utility

The moisture sensor is connected to the development board as follows:

- Connect pin VCC to 3.3 V or 5 V
- Connect pin GND to ground
- Connect pin SIG to GPIO pin A5

5.5.2 Goals of the experiment

The goal of the experiment is to send the measured values stored on the microcontroller memory to the serial monitor of the PC.

5.5.3 Experimental results

Write a program in C language to configure the USART module and display the measured values on the serial monitor. Your program should do the following:

- Configure the USART peripheral: enable the USART peripheral and choose the appropriate USART instance. Next set up the baud rate, data, stop and parity bits accordingly. Lastly, configure the GPIO pins for USART communication.
- Implement a function to send data over USART. The function should take the measured values stored in the microcontroller memory and use the USART_SendData() or equivalent function to send the data.
- Display the values using the Hercules SETUP. Configure the Hercules SETUP serial terminal to match the settings used in STM32 program (data bits, parity bits, stop bits and baud rate) and start the serial communication in the Hercules SETUP terminal.

5.6 Laboratory experiment 5

In experiment 5, we will learn about Inter-Integrated-Circuit (I2C) protocol and how to use an I2C driver to interface with a digital temperature and humidity sensor. I2C communication is predefined serial communication between several devices. I2C protocol is defined between master and slaves. Master is the device that generates a clock to synchronize the communication (Figure 5-6).

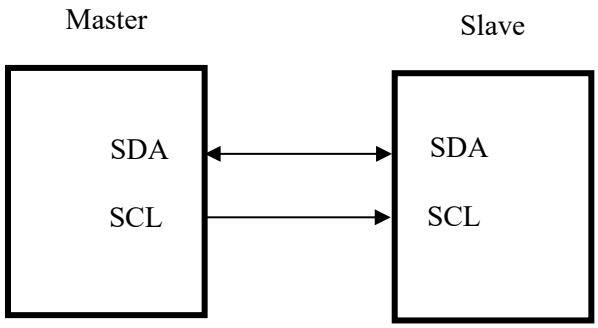


Figure 5-6. I2C master-slave connection.

This protocol requires two wires for communication, but these two wires can support up to 1008 slaves. SDA line is used for data transmission in both directions and SCL wire is used for clock and the direction is from master to the slaves.

Messages contain two frames of data: start condition, an address frame, acknowledge bit and data (Table 5-2). Data transmission starts with start condition by switching SDA line from high level to low before switching SCL line from high to low. The address frame is the starting frame of any communication sequence. It contains the address of the device (7 bits), one bit for read or write command and the 9th bit is used for acknowledge. If the 9th bit is not set up that means that receiving device does not receive frame or the message is not understandable. The direction of SDA depends of write or read bit in address frame. Stop condition initiates end of communication by switching SCL line form high to low voltage level before switching SCL line from high to low voltage level.

Table 5-2. I2C message.

Start	Address frame	Read/Write Bit	ACK/ NACK bit	Data frame	ACK/ NACK bit	Data frame	ACK/ NACK bit	Stop

5.6.1 I2C transmission

I2C transmission is realized using following steps:

1. The master sends start conditions to the connected devices by switching the SDA line from high to low level before switching SCL line from high to low level.
2. The master sends the address of the device that wants to communicate with and the address is followed by a read/write bit.
3. Slave that has recognized the address that is sent from the master, sends acknowledge bit by switching SDA line to low level for one bit.
4. The master sends or receives data
5. After successful transmission, the acknowledge bit is sent
6. Stop condition is generated by switching SCL line high before switching SDA line high.

The SHT20 is a digital temperature and humidity sensor designed for various applications that require precise temperature and humidity measurements. Some of its key features are:

1. Measurement accuracy: up to ±3% for relative humidity (RH) and ±0.3°C for temperature measurements, thus ensuring reliable and precise data.
2. Digital interface: communication using standard I2C interface.
3. Wide measurement range: it can measure humidity from 0% to 100% RH and temperature from -40°C to +125°C, allowing it to be used in a variety of environments and applications.
4. Fast response time: allowing quick and accurate measurements in dynamic conditions.
5. Calibration and digital signal processing: the sensor is factory calibrated and incorporates digital signal processing techniques that enhance the reliability and stability of the measured data.
6. Low power consumption: suitable for battery powered applications.

The SHT20 sensor is commonly used in weather stations, HVAC systems, industrial automation, medical devices and consumer electronics. Its small size, high accuracy and digital interface make it a reliable choice for humidity and temperature measurement applications. The sensor is given in Figure 5-7.

Figure 5-7. The SHT20 digital temperature and humidity sensor.

5.6.2 Experimental setup

Equipment:

- Embedded system development board (STM32f103c8t6)
- SHT20 I2C sensor for temperature and humidity
- Breadboard and jumper wires

The temperature and humidity sensor are connected to the development board as follows:

- Connect pin VCC to 3.3 V or 5 V
- Connect pin GND to ground
- Connect pin SCL to GPIO pin B6
- Connect pin SDA to GPIO pin B7

5.6.3 Goals of the experiment

The goal of this experiment is to establish I2C communication and measure the values of ambient temperature and humidity using SHT20 sensor.

Write a program in C language to configure I2C driver and read the values from SHT20 senor. Your program should do the following:

- Initialization of I2C communication:
  - Enable APB1 peripheral clock for I2C
  - Enable APB2 clock for SCL and SDA pins
  - Set the pins, mode and speed
  - Set the GPIO mode
  - Remap the configuration of pins
  - Set I2C clock speed, mode, duty cycle, enable acknowledge and address of master

5.6.4 Experimental results

```
#include <stm32f10x_gpio.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_tim.h>
#include <stm32f10x_i2c.h>


#define SHT20_I2C_ADDR                          0x80
#define SHT20_NOHOLD_TEMP_REG_ADDR              0xF3
#define SHT20_NOHOLD_HUMDTY_REG_ADDR            0xF5
#define SHT20_HOLD_TEMP_REG_ADDR              0xF3
#define SHT20_HOLD_HUMDTY_REG_ADDR            0xF5
#define ERROR_I2C_TIMEOUT                       998
#define ERROR_BAD_CRC                           999
#define SHIFTED_DIVISOR                         0x988000
I2C_InitTypeDef I2C_InitStruct;
GPIO_InitTypeDef GPIO_I2C_InitStruct;
```

```
void init_I2C1(void){

	// enable APB1 peripheral clock for I2C1
			RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
			// enable clock for SCL and SDA pins
			RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

			/* setup SCL and SDA pins
			 * You can connect I2C1 to two different
			 * pairs of pins:
			 * 1. SCL on PB6 and SDA on PB7
			 * 2. SCL on PB8 and SDA on PB9
			 */
			GPIO_I2C_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;

				// we are going to use PB6 and PB7
			GPIO_I2C_InitStruct.GPIO_Mode = GPIO_Mode_AF_OD;
	// set pins to alternate function
			GPIO_I2C_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
	// set GPIO speed
				//GPIO_InitStruct.GPIO_OType = GPIO_OType_OD;
	// set output to open drain --> the line has to be only pulled low, not
driven high
				//GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
	// enable pull up resistors
			GPIO_Init(GPIOB, &GPIO_I2C_InitStruct);
	// init GPIOB
			GPIO_PinRemapConfig(GPIO_Remap_I2C1, ENABLE);

			// Connect I2C1 pins to AF

			 I2C_InitStruct.I2C_ClockSpeed=100000;
			 I2C_InitStruct.I2C_Mode=I2C_Mode_I2C;
			 I2C_InitStruct.I2C_DutyCycle=I2C_DutyCycle_2;
			 I2C_InitStruct.I2C_Ack=I2C_Ack_Disable;

I2C_InitStruct.I2C_AcknowledgedAddress=I2C_AcknowledgedAddress_7bit;  //size
of the address
			 I2C_InitStruct.I2C_OwnAddress1=0x00;  //address of the
Microcontroller

			 I2C_Init( I2C1,&I2C_InitStruct);   //init I2C
			 I2C_Cmd(I2C1, ENABLE);             //Enables or disables
the specified I2C peripheral.

}
void I2C_start(I2C_TypeDef* I2Cx, uint8_t address, uint8_t direction){
		// Send I2C1 START condition
			I2C_GenerateSTART(I2Cx, ENABLE);

			// Send slave Address for write
			I2C_Send7bitAddress(I2Cx, address, direction);

}
void I2C_write(I2C_TypeDef* I2Cx, uint8_t data)
```

```
                {

                        I2C_SendData(I2Cx, data);
                        // wait for I2C1 EV8_2 --> byte has been transmitted
                        //while(!I2C_CheckEvent(I2Cx,
I2C_EVENT_MASTER_BYTE_TRANSMITTED));
                }

                /* This function reads one byte from the slave device
                 * and acknowledges the byte (requests another byte)
                 */
uint8_t I2C_read_ack(I2C_TypeDef* I2Cx){
                        // enable acknowledge of recieved data
                        I2C_AcknowledgeConfig(I2Cx, ENABLE);
                        // wait until one byte has been received
                        //while( !I2C_CheckEvent(I2Cx,
I2C_EVENT_MASTER_BYTE_RECEIVED) );
                        // read data from I2C data register and return data
byte
                        uint8_t data = I2C_ReceiveData(I2Cx);
                        return data;
                }

                /* This function reads one byte from the slave device
                 * and doesn't acknowledge the recieved data
                 */
uint8_t I2C_read_nack(I2C_TypeDef* I2Cx){
                        // disabe acknowledge of received data
                        // nack also generates stop condition after last byte
received
                        // see reference manual for more info
                        I2C_AcknowledgeConfig(I2Cx, DISABLE);

                        // wait until one byte has been received
                        //while( !I2C_CheckEvent(I2Cx,
I2C_EVENT_MASTER_BYTE_RECEIVED) );
                        // read data from I2C data register and return data
byte
                        uint8_t data = I2C_ReceiveData(I2Cx);
                        return data;
                }

void I2C_stop(I2C_TypeDef* I2Cx){
      // Send I2C1 STOP Condition
      I2C_GenerateSTOP(I2Cx, ENABLE);
}
uint16_t SHT20_readValue(I2C_TypeDef* I2Cx, uint8_t address,uint8_t reg){
      uint8_t lsb,msb;
      I2C_start(I2Cx,address,I2C_Direction_Transmitter);
      I2C_write(I2Cx,reg);
      I2C_start(I2Cx,address,I2C_Direction_Receiver);
      delay_ms(20);
      msb=I2C_read_ack(I2Cx);
      delay_ms(20);
      lsb=I2C_read_ack(I2Cx);
```

```
      u8 checksum=I2C_read_ack(I2Cx);
      delay_ms(20);
      I2C_stop(I2Cx);

      uint16_t raw_value=((uint16_t) msb<<8)|(uint16_t)(lsb);
//    if (SHT20_checkCRC((u16)raw_value, (u8)checksum) != 0)
//            return (ERROR_BAD_CRC);
      return (raw_value & 0xfffc);


}
float SHT20_readHumidity(I2C_TypeDef* I2Cx, uint8_t address){
      float tempRH;
      float rh;
      uint16_t
rawHumidity=SHT20_readValue(I2Cx,address,SHT20_NOHOLD_HUMDTY_REG_ADDR);

//    if (rawHumidity == ERROR_I2C_TIMEOUT || rawHumidity == ERROR_BAD_CRC)
//        return (rawHumidity);
    tempRH=rawHumidity*125.0/65535.0;
    tempRH=tempRH-6.0;
    return tempRH;

}

float SHT20_readTemp(I2C_TypeDef* I2Cx, uint8_t address){
      float temp;
      float rh;
      uint16_t
rawTemp=SHT20_readValue(I2Cx,address,SHT20_NOHOLD_TEMP_REG_ADDR);
//
//    if (rawTemp == ERROR_I2C_TIMEOUT || rawTemp == ERROR_BAD_CRC)
//        return (rawTemp);
    temp=rawTemp*(175.72/65536.0);
    temp=temp-46.85;
    return temp;
}
u8 SHT20_checkCRC(u16 message_from_sensor, u8 check_value_from_sensor)
{
      u8 i;
    u32 divsor = (u32)SHIFTED_DIVISOR;
    u32 remainder = (u32)message_from_sensor << 8;

    remainder |= (u32)check_value_from_sensor;

    for(i = 0 ; i < 16 ; i++) {
        if(remainder & (u32)1 << (23 - i))
           remainder ^= divsor;
        divsor >>= 1;
    }
  return (u8)remainder;

}
```

5.7 Conclusion

The remote Embedded laboratory offers a convenient and interactive platform for students and enthusiasts interested in embedded systems. By providing virtual access to the real hardware and comprehensive documentation, this laboratory enables its users to gain practical experience in microcontroller programming and further develop their understanding of embedded systems.

The key advantages of the remote embedded laboratory are:

- **Virtual Access**: allowing users to access the embedded system setup from anywhere as if they are physically using them in the laboratory.
- **Real-time interaction**: sending commands, reading sensor data and observing the system response in real time.
- **Experiment library**: the laboratory has a set of pre-designed experiments covering various topics that can be combined to form a practical embedded system project.
- **Comprehensive documentation**: a detailed documentation is provided for every experiment, including circuit diagrams, code examples, step-by-step instructions and explanations of the fundamental concepts.

Overall, the remote embedded laboratory offer an innovative and effective approach to remote learning and experimentations in the field of embedded systems. It provides a flexible and accessible platform for individuals to explore and expand their knowledge of embedded systems and microcontroller programming.