Meteorological station

Documentation & report

Date: 15.04.2019

Information Technology

Monday 10:15, group 9

IFE, 4th semester

Team & task assignment

Piotr Kaźmierski (leader)	• Bluetooth + UART • temperature & humidity sensor
Piotr Kocik	• button + interrupts
Michał Kuśmidrowicz	• LCD screen + I^2C
Kacper Kubicki	• pressure sensor + SPI • LED + PWM

Devices

- Arduino Uno Rev3
- DHT11 temperature & humidity sensor
- BMP280 temperature, pressure & altitude sensor
- Green LCD 2x16 with LED backlight through a I^2C converter
- HC-05 Bluetooth module
- RGB LED with common anode
- Tact Switch 6x6mm / 4,3mm THT

1. Project description

1.1. General description

The aim of the project was to create a meteorological station which could display the information relating to the weather conditions using an LCD screen with the ability to change currently shown information via a button or remotely, via Bluetooth. Additionally a persistent indicator of a humidity in the form of an RGB LED diode was added.

1.2. Information to be displayed

Each reading/information is displayed separately, centred on the LCD screen with a heading that informs which mode is currently used, e.g.:

TEMPERATURE
28°C

Possible information to be displayed are:

- Temperature
- Heat index (perceived temperature)
- Atmospheric pressure

The information about the reading in the serial communication is displayed the same as on the screen, but the heading and the value along with the unit are all on the same line.

The Bluetooth app can display all the reading simultaneously, stacked vertically.

1.3. How to use it?

The currently displayed information can be changed via interrupt driven button (tactile switch) or remotely via Bluetooth using for example a smartphone with an appropriate app installed. The button allows to loop through all the screens one by one in a forward manner. Bluetooth allows to go back and forth between the screens as well as read the data remotely repeatedly or on demand.

1.3.1. Control using Bluetooth

To use the meteorological station via Bluetooth one must have a device with an app that can perform serial communication via Bluetooth. In our case we're using "Bluetooth Electronics" by keuwlsoft which is available for Android devices via Google Play. It allows users to easily read all the data simultaneously.

The button in the bottom right corner turns on or off continuous sending of the data from the meteorological station.

1.3.2. Control via the serial port

The device can also be controlled via the use of serial communication, either by the built-in USB connection or Bluetooth adapter. In case one wants to use the USB connection the Bluetooth module must be disconnected from the TX/RX pins since they are shared with the USB connection.

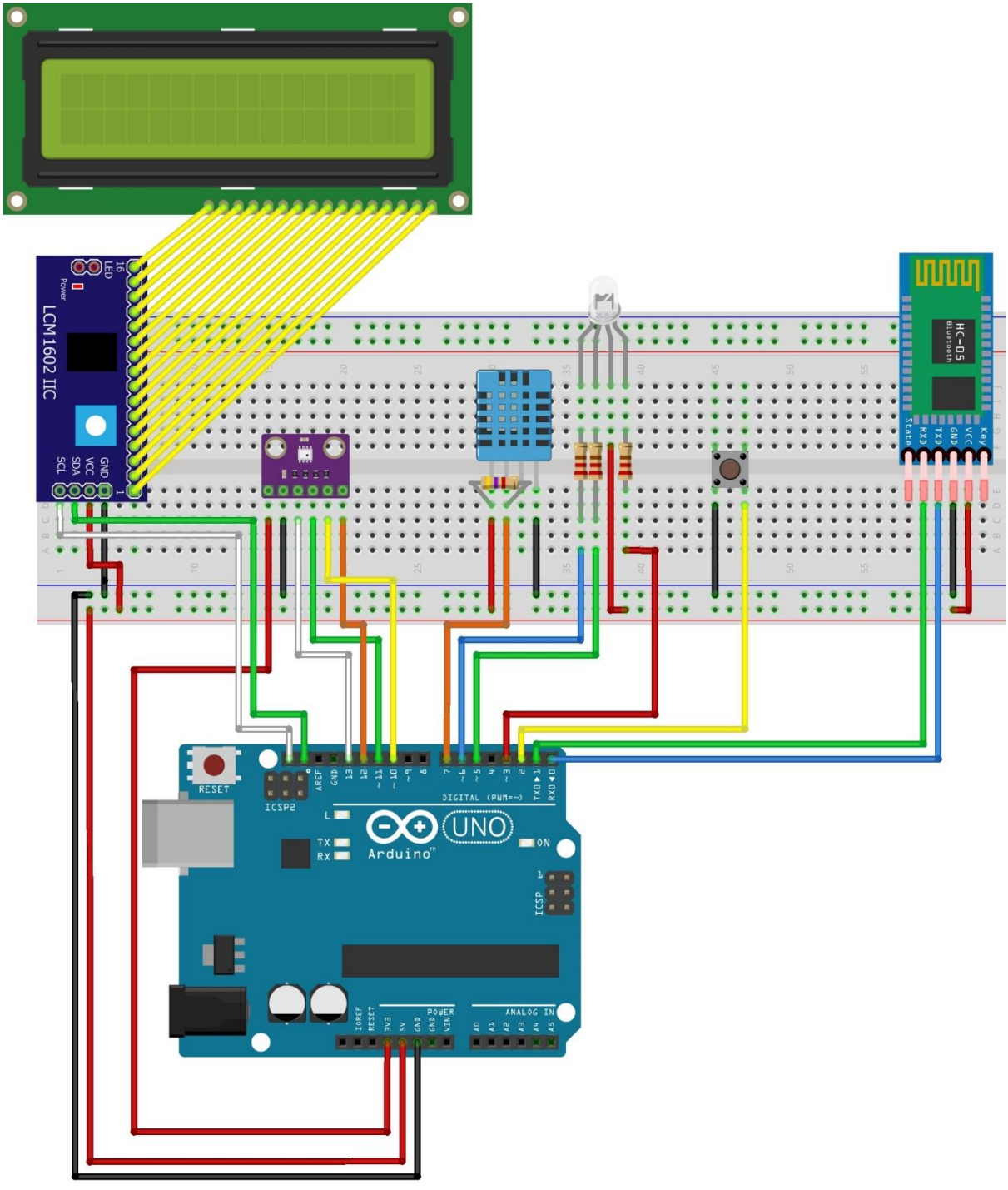
The list below presents the available commands:

Command	Description
n	Switches to the next mode (loops back to the first one at the last mode).
p	Switches to the previous mode (loops back to the last one at the first mode).
e	Enable the continuous sending of the current measurement via the serial port (in human readable form).
d	Disable the continuous sending of the current measurement via the serial port (in human readable form).
o	Pulls the current measurement, regardless of the continuous data sending setting (in human readable form).
b	Enable the continuous sending of all measurements via the serial port (in app readable form).
z	Enable the continuous sending of all measurements via the serial port (in app readable form).

2. Peripherals and interface configuration

2.1. GPIO

Below the GPIO connection diagram is presented:



The table below presents the pinout:

Pin	Alias	PWM	Assigned to
0	RX	No	Bluetooth module TX
1	TX	No	Bluetooth module RX
2	IRQ0	No	Tactile switch (interrupt enabled)
3	IRQ1	Yes	RBG LED red component
4	-	No	-
5	-	Yes	RGB LED green component
6	-	Yes	RGB LED blue component
7	-	No	DHT11 data pin
8	-	No	-
9	-	Yes	-
10	SS	Yes	BMP280 CSB pin
11	MOSI	Yes	BMP280 SDA pin
12	MISO	No	BMP280 SDO pin
13	SCK	No	BMP280 SCL pin
14	A0	N/A	-
15	A1	N/A	-
16	A2	N/A	-
17	A3	N/A	-
18	A4, SDA	N/A	I^2C LCD converter SDA pin
19	A5, SCL	N/A	I^2C LCD converter SCL pin

Pins 0 to 13 are digital pins. Some of them are PWM.
Pins A0 to A5 are analogue pins.
Pins 0 and 1 are used for UART communication via the Bluetooth module.
Pin 2, which is assigned to interrupt 0 (IRQ0) is used for the tactile switch.
Pins 3, 5, and 6 are used for red, green and blue components of the RGB diode respectively.
Pin 7 is used for the DHT11 sensor data send & receive.
Pins 10 through 13 are used for SPI communication with the BMP280 sensor.
Pins A4 and A5 are used as SDA and SCL in I^2C protocol to communicate with the I^2C LCD converter

Some of the pins are configured by the library functions, such as SPI pins, I^2C pins and DHT11 pin.

2.1.1 DHT11

DHT11 is configured using the library functions:

```
//meteo.ino - our code
#define DHTPIN 7
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE); //DHT type object allows us to interface with the
sensor via it’s class methods

void setup() {
      //...
      dht.begin();
      //...
}
```

```
//DHT.cpp - DHT11 library
void DHT::begin(uint8_t usec) { //the declaration has a default value for
usec set
      //...
      pinMode(_pin, INPUT_PULLUP); //_pin is a variable set by the DHT
constructor that corresponds to the first constructor arugment (in our case
DHTPIN that equals 7)
      //...
}
```

An equivalent direct way of doing it is:

```
pinMode(DHTPIN, INPUT_PULLUP);
```

Doing it via directly setting registers:

```
DDRD |= B10000000; //sets pin 7 to INPUT
PORTD |= B10000000;  //sets pin 7 to HIGH, enabling pullup resistors
```

`INPUT_PULLUP` sets a pin into an input mode with a pullup resistor attached. This reverses the way input should be read, since now high voltage = 0, and low voltage = 1.

2.2. I²C

I²C uses two wires for communication- **SDA** (Serial Data) and **SCL** (Serial Clock). On Atmel ATmega 328P I2C is also called **TWI** (two wire interface). Under the hood, the I2C connection is established in the following way – firstly, the register bit **TWINT** (TWI Interrupt Flag) is reset by setting its value to 1. Then, the interface TWI is unlocked by setting a value of bit called **TWEN** (TWI Enable bit). After that, a bit **TWSTA** (TWI Start Condition bit) is set to 1, which means that the transmission will be performed in 'Master' mode. Then, a while loop is waiting for re-setting flag TWINT, which will confirm that the hardware chip correctly performed the start sequence of the transmission.

```
void I2C_start() {
      // Assigning a value to control register - sending START condition
      TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTA);

      // Wait for TWINT flag set. This indicates that the START
      // condition has been transmitted
      while (!(TWCR & (1 << TWINT)));
}
```

2.3. SPI

The SPI – serial peripheral interface – is used for the communication with a BMP280 temperature, pressure & altitude sensor. The SPI communication is handled by a library function.

```
//meteo.ino - our code
#define BMP_CS 10
Adafruit_BMP280 bme(BMP_CS); //creating a bme object in order to use
Adafruit_BMP280 class functions
```

In the code above, by giving numbers referring only to CS pin one calls a Adafruit_BMP280 constructor using hardware SPI, which is handled by "SPI.cpp" library functions.

SPI communication is realized using 4 pins – SCK, MISO, MOSI and CS. The communication is initialized by setting a CS pin to 0 and finishes when CS pin is set to 1. Communication from a master device to a

slave device occurs through a MOSI (Master Output Slave Input) pin. Communication from a slave device to a master device is realized using a MISO (Master Input Slave Output) pin. SCK pin is used to generate a serial clock signal. The frequency of that signal, and therefore a speed of the transmission is not specified and depends on the capabilities of the slave device. During a one clock cycle, a full-duplex data transmission occurs.

SPI interface settings in Atmel ATMega328P are stored in 8-bit SPI Control Register (SPCR). In order to enable any SPI operations, bit 6 of the register (SPE) must be set on 1. Bit 5 (DORD) is used to choose order of the data – when it is set to 1 least significant bit of the data is send first, when it is set to 1 most significant bit is send first. Bits 2 (CPHA) and 3 (CPOL) control whether data is shifted in and out on the rising or falling edge of the data clock signal (called the clock phase), and whether the clock is idle when high or low (called the clock polarity). The speed of a transmission can be regulated by setting bits 0 and 1 in eight different configurations depending on the oscillator clock frequency. In our case, both of them are set to 0. Function of "SPI.cpp" library requires only to provide a desired speed which will be automatically set to the closest possible speed of eight available, order of the data and configuration of CPHA and CPOL bits as one of the 4 possible modes. An example code is presented below.

```
//Adafruit_BMP280.cpp
void Adafruit_BMP280::write8(byte reg, byte value) {
	//...
		if (_sck == -1)
			_spi->beginTransaction(SPISettings(500000, MSBFIRST,
SPI_MODE0)); //in sequence: desired speed (Hz), data order (MSB first),
SPI_MODE0 – CPHA set to 0 and CPOL set to 0
	//...
}
```

2.4. USART

USART is configured via library function call:

```
const int DEFAULT_BAUD_RATE = 9600;

void setup() {
	Serial.begin(DEFAULT_BAUD_RATE);
	//...
}
```

To configure USART via register manipulation we have to enable the receiver and transmitter, as well as set the baud rate.

```
UCSR0B = (1<<RXEN0)|(1<<TXEN0); //enables the receiver and transmitter
UBRR0L = 103; //sets the baud rate to 9600
```

The rest of the settings are kept at default values and the most important of those are:

- Mode of operation: asynchronous
- Parity: disabled
- Stop bits: 1
- Character size: 8 bits

To send data via UART we have to first check if the USART Data Register Empty (UDRE0) bit is set to 0. If so, we can load the data into the USART transmit data buffer register. In case of 9-bit characters the highest bit has to be manually put into the Transmit Data Bit 8 (TXB80). Example:

```
void USART_Transmit(unsigned int data) {
	/* Wait for empty transmit buffer */
	while (!(UCSR0A & (1 << UDRE0)))
	;
	/* Copy 9th bit to TXB8 */
	UCSR0B &= ~(1 << TXB8);
	if (data & 0x0100)
		UCSR0B |= (1 << TXB80);
	/* Put data into buffer, sends the data */
	UDR0 = data;
}
```

To receive data via UART we have to first check if there have been no errors during transmission. To do that, we have to check that USART Parity Error (UPE0) and Frame Error (FE0) bits are set to 0. If so, we can check if there's data available for us by determining whether USART Receive Complete is set to 1. If that's true we can read data from USART receive data buffer register (UDR0) and optionally the Receive Data Bit 8 (RXB80), if we want to receive 9 bits of data. Example:

```
unsigned int USART_Receive(void) {
	unsigned char status, resh, resl;
	/* Wait for data to be received */
	while (!(UCSRnA & (1 << RXCn)))
		;
	/* Get status and 9th bit, then data */
	/* from buffer */
	status = UCSRnA;
	resh = UCSRnB;
	resl = UDRn;
	/* If error, return -1 */
	if (status & (1 << FEn) | (1 << DORn) | (1 << UPEn))
		return -1;
	/* Filter the 9th bit, then return */
	resh = (resh >> 1) & 0x01;
	return ((resh << 8) | resl);
}
```

2.5. Interrupts

The button (tactile switch) used in this device uses interrupts to switch displayed information on the screen. The button allows to loop through all of the information on the consecutive screens one by one. The tactile switch is configured in a following way:

```
const int BUTTON_PIN = 2;
const int BOUNCE_TIME = 200;
void setup() {
	pinMode(BUTTON_PIN, INPUT_PULLUP);
	attachInterrupt(digitalPinToInterrupt(BUTTON_PIN),	buttonHandler,
FALLING);
	//…
}
```

In this device, we make use of the INT0 interrupt, which is defined on PD2 pin as it serves as an external interrupt source.

INPUT_PULLUP sets a pin into an input mode with a pullup resistor attached between input line 2 and 5V power supply and setting the work mode of this input pin with pullup. Because the button is connected to ground, pushing it forces to set 0 on the microcontroller.

Doing it via directly setting registers:

```
DDRD |= B00000000;
PORTD |= B00000100;
```

Until the button had been pressed, the pin level was set to HIGH, so that event cause decrease in logical state, hence the FALLING keyword was used to set the falling edge as an interrupt trigger. This instruction (`attachInterrupt`) sets the Interrupt Sense Control (ISC) bit number 0 to 0 and the ISC bit number 1 to 1, in the External Interrupt Control Register A (EICRA), what corresponds to the falling edge state, which, when detected, triggers the interrupt.

Doing it directly via registers:

```
EICRA |= (1 << ISC01)|(0 << ISC00);
```

We have to enable INT0 interrupt handling in the External Interrupt Mask Register (EIMSK). attachInterrupt function does that automatically, manually it would be as follows:

```
EIMSK |= (1 << INT0);
```

The last register involved in interrupt is the External Interrupt Flag Register (EIFR). When an the processor detects a falling state), the flag in bit number 0 is set and the interrupt occurs.

After an interrupt is detected, the following Interrupt Service Routine (ISR) is invoked:

```
void buttonHandler()
{
	static unsigned long last_interrupt_time = 0;
	unsigned long interrupt_time = millis();
	//if interrupts come faster than BOUNCE_TIME, assume it's a bounce and
ignore
	if (interrupt_time - last_interrupt_time > BOUNCE_TIME) {
		if (screen == SCREEN_COUNT - 1) {
			//we are at the last screen, so we have to loop back to
the beginning
			screen = 0;
		}
		else {
			screen++;
		}
	}
	last_interrupt_time = interrupt_time;
}
```

When an interrupt is detected, the processor is directed 0x002 address, which is an Interrupt Vector address that holds the aforementioned ISR. The external interrupt flag in EIFR register is cleared automatically by the attachInterrupt function.

The button also provides debouncing feature, that prevents from excessive interrupt invocation. const int BOUNCE_TIME = 200; instruction sets the debouncing period for 200ms.

2.6. PWM

PWM or Pulse Width Modulation is a technique that bases on rapid change of the digital signal that are able to simulate voltages in between two logical states. PWM on ATMega 328P is controlled by three timers known as Timer 0, Timer 1, and Timer 2. Each of those two timers has two output compare registers. When the timer reaches the compare register value, the corresponding output is toggled. The Timer/Counter Control Registers TCCRnA and TCCRnB are used to control each timer by holding the main control bits.

In our project PWM is used in order to control the color of a RGB diode. The diode is used to signal the level of air humidity read from DHT11 sensor. The diode has been programmed to have 3 different states – blue, which corresponds to the humidity lower than 30%, green, which corresponds to the humidity between 30% and 55% and red for humidity higher than 55%. A RGB diode uses 3 of the 6 PWM outputs provided in ATMega 328P. The code used to handle the diode is presented below.

```
#define RED_RGB_PIN 3
#define GREEN_RGB_PIN 5
#define BLUE_RGB_PIN 6

void setColor(uint8_t red, uint8_t green, uint8_t blue)
{
	analogWrite(RED_RGB_PIN, 255 - red);
	analogWrite(GREEN_RGB_PIN, 255 - green);
	analogWrite(BLUE_RGB_PIN, 255 - blue);
}

void humidityHandler() {
	static HumidityMode h("HUMIDITY", "%", "%", 0, 2, 4);
	if (h.getValue() < 30.0) {
		setColor(0, 0, 255);
	}
	else if (h.getValue() > 55.0) {
		setColor(255, 0, 0);
	}
	else {
		setColor(0, 255, 0);
	}
}

void loop() {
	//...
	humidityHandler();
	//...
}
```

As it can be seen the humidityHandler() method is called in a main loop of the program. When called the method will read the humidity using DHT11 sensor and set the color of the diode accordingly to the routine mentioned before using analogWrite() function. An analogWrite() function uses a scale of 0 – 255, where 255 corresponds with a 100% duty cycle and 127 with a 50% duty cycle.

2.7. BMP280 temperature, pressure & altitude sensor

BMP280 is a piezoelectric sensor featuring high EMC robustness, high accuracy and linearity and long term stability. It can use both I^2C and SPI communication interfaces. For the purpose of this project we have decided to use SPI communication. BMP280 can only work as a slave device for both protocols.

BMP280 sensor is equipped with 8 bit registers, two of which are control registers which can be written into, as well as 6 data registers of read only type used to read measurements data. Register 0xF5 is used to set the rate, filter and interface options of the device. Register 0xF4 is used the data acquisition options of the device. The configuration of those registers is handled by Adafruit_BMP280 library as follows:

```
enum sensor_sampling {
      //..
      /** 16x over-sampling. */
      SAMPLING_X16 = 0x05 };

enum sensor_mode {
      //...
      /** Normal mode. */
      MODE_NORMAL = 0x03,
      //... };

enum sensor_filter {
      /** No filtering. */
      FILTER_OFF = 0x00,
      //... };

enum standby_duration {
      /** 1 ms standby. */
      STANDBY_MS_1 = 0x00,
      //... };

void setSampling(sensor_mode mode = MODE_NORMAL,
      sensor_sampling tempSampling = SAMPLING_X16,
      sensor_sampling pressSampling = SAMPLING_X16,
      sensor_filter filter = FILTER_OFF,
      standby_duration duration = STANDBY_MS_1);
```

The pressure measurement is read from registers 0xF7, 0xF8 and 0xF9 and is handled by Adafruit_BMP280 library. In order to obtain the proper measurement of atmospheric pressure, raw data read from the registers has to be processed by an algorithm provided by a producer in BMP280 datasheet.

SPI interface supported by BMP280 sensor allows only to use 7 bits of a register at once, as the most significant bit is replaced by a RW (read/write) bit (with value ‘0’ for write and ‘1’ for read).

BMP280 configuration is realized by library function call on an Adafruit_BMP280 object described in section 2.3.

```
void setup()
{
	//...
	if (!bme.begin()) { //check if BMP280 is connected and initialize it
		Serial.println("Could not find a valid BMP280 sensor, check wiring!");
	}
	//...
}
```

The pressure is read each time the mode of the device is changed to a “Pressure” mode as follows:

```
const int PRESSURE_CALIBRATION_DIFFERENCE = 26;

class PressureMode : public Mode {
public:
	using Mode::Mode;
	inline float getValue() override {
		return bme.readPressure() / 100 + PRESSURE_CALIBRATION_DIFFERENCE;
	}

};
```

An introduction of additional `PRESSURE_CALIBRATION_DIFFERENCE` factor of 26 hPa was necessary, since after a comparison with other barometers, the measured atmospheric pressure has been consistently and repeatedly lower by 26 hPa, which indicates at the wrong calibration of a sensor.

2.8. DHT11 temperature & humidity sensor

Single-bus data format is used for communication and synchronization between MCU and DHT11 sensor. One communication process is about 4ms. Data consists of decimal and integral parts. A complete data transmission is 40 bit, with the last 8 bits being the checksum.

The data is either continuously pulled from the sensor each 2 seconds or it is pulled at request. However, pulling data more often than once every 2 seconds will return the last reading.

The communication process starts with the MCU sending the start signal. Once the start signal completes the sensor sends a response of 40 bits of data. Once the transfer finishes the sensor will start waiting for the next start signal.

2.9. HC-05 Bluetooth module

HC-05 Bluetooth modules default UART settings are the same as Arduino’s default UART settings, therefore it requires no additional setup. It also doesn’t require any special actions to be interfaced with, since it works like a standard serial connection. The only required thing to use it is a serial monitor app installed on the Bluetooth client device.

2.10. LCD Screen with an I^2C converter

In our project the I2C bus is used for communication with the LCD screen. We decide to use utterly popular library called “LiquidCrystal_I2C”, so as to make this process easy to establish and control. Firstly, one has to create object representing the LCD screen and configure it via constructor and method `begin()`:

```
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
```

```
void setup {
      //...
      lcd.begin(16,2);
      lcd.backlight();
      //...
}
```

Numbers in constructor refer to:

- Hexadecimal address of a connected device
- **En** ("enabling" pin) - when this pin is set to logical low, the LCD does not care what is happening with RW, RS, and the data bus lines; when this pin is set to logical high, the - LCD is processing the incoming data
- **RW** - pin writing/reading to/from LCD
- **RS** (Register Selector) - pin selects registers between Instruction Register and Data Register
- **Data pins** (4,5,6,7) - perform read/write of data
- **The back light control pin** (3)
- **POSITIVE** is the back light polarity.

16 and 2 in `begin()` method stand for size of the screen, respectively its width and height. Then, the backlight is turned on by calling a `backlight()` method. Here is the exemplary snippet from our script with added explanation how one can control LCD screen via methods provided by "LiquidCrystal_I2C" library:

```
void displayDataOnScreen() {
      // Clearing the whole content of the screen
      lcd.clear();
      // Setting a cursor at given position (x,y) on the screen
      lcd.setCursor(headerShift,0);
      // Printing String at position of the cursor
      lcd.print(header);
      lcd.setCursor(valueShift, 1);
      lcd.print(getValue(),decimalDigits);
      lcd.print(" ");
      lcd.print(unitLCD);
}
```

The HD44780 has two 8-bit registers, an instruction register (**IR**) and a data register (**DR**):

- **IR** - stores instruction codes, such as display clear and cursor shift, and address information for display data RAM (**DDRAM**) and character generator RAM (**CGRAM**).
- **DR** - temporarily stores data to be written into DDRAM or CGRAM and temporarily stores data to be read from DDRAM or CGRAM.

The HD44780 can send data in either two 4-bit operations or one 8-bit operation. Chosen library uses 4-bit interface data, that's why only four bus lines are used (**DB4** to **DB7**), the rest is inactive. Each command has to be sent twice, first 4 bits are the same as values of DB4 to DB7 in 8bit mode and the second message consist of 4 bits which would represent DB0 to DB3 in 8bit mode.

According to library's and LCD's documentations, at first the program has to wait at least 40ms after V_{cc} (power supply voltage) rises above 2.7V. Then, both pins RS and RW are pulled to low, so as to begin commands. Next, the script makes a three-fold attempt to set the communication into four-bit mode (a default is 8bit mode). There is a delay of 4500ms between each attempt. The last step of display's configuration is setting a number of displayed lines and a character font. This parameters cannot be changed later. If everything went as expected, the display is cleared and set into an entry mode. After performing this procedure, the display should be properly initialized and ready to use.

The communication between Arduino and screen can be performed with the following set of commands (in 4bit mode):

Instruction	RS	RW	DB7	DB6	DB5	DB4
Clear display	0	0	0	0	0	0
	0	0	0	0	0	1
Cursor shift left/right	0	0	0	0	0	1
	0	0	0	R/L	-	-
Write data to CG/DDRAM	1	0	Written data			
	0	0				
Read data from CG/DDRAM	1	1	Read data			
	0	0				

- indicates no effect
R/L = 0: shift to the left
R/L = 1: shift to the right

3. Failure Mode and Effect Analysis

Some components of the device are of the highest importance – without them, the device cannot fulfil its basic function. However, failure of some other components does not render the whole device unusable. Some components' importance is context-dependent. The table below presents items or functionalities together with their importance for the project's operation.

Item/function	Importance
Microcontroller	critical
Power supply	critical
Screen & I^2C converter	low / critical
Button	low / critical
Bluetooth module	low / critical
Temperature and humidity sensor	high
Pressure sensor	high
LED	low

The microcontroller and power supply are essential to device's functioning. The microcontroller is responsible for the program logic and controlling the devices, therefore it cannot fail. Because the device does not have a battery power supply must be constantly available.

The low / critical importance stems from the fact that we can read data and change the currently displayed information in a few ways:

- Changing
  - Using a button
  - via Bluetooth using an app
- Displaying
  - On the LCD screen

- via Bluetooth using an app

Therefore, one way of changing the displayed information can fail, as well as one way of changing the displayed information. However, if both fail, the device wouldn't be able to function properly: worst case scenario is that nothing is displayed (Bluetooth & screen fail) or we're stuck on a given mode (button and Bluetooth fail).

The sensors are of high importance since they provide all the data to be displayed. Without them, the data would be either unavailable or "frozen", that is the reading would always be the same.

LED is of the lowest importance – it only serves as an indicator of one value and it does not affect other components in any way. Moreover, it can be replaced by just adding another mode that would display the reading normally indicated by the LED.

Power supply failure can be detected in a number of ways, for example the LED or screen may flicker. Power-related faults may also appear when a short is caused by making improper electrical connections, such as connecting Vin pin to the ground. This may happen either by putting cables in the incorrect places, or dropping a piece of metal on the board.

A microcontroller failure would probably be easy to detect, as most likely it would be a complete failure of the chip.

Screen's or I^2C converter's failure would be easily noticeable, since it would either have problems with backlight or displaying the content properly.

Button's failure might result in fake presses or no reaction when pressed. That would be easy to notice since the screen would change by itself (fake presses) or stay the same even if the button was pressed.

Bluetooth module's failure could have multiple symptoms, for example the device could disconnect spontaneously, the communication could stop working and so on.

Sensors could stop responding altogether or start giving wrong readings. The 2^{nd} symptom could be hard to realise if the error would be small, unless we had other, reference sensors.

LED's failure might not be so obvious, since it could also be a power supply failure. Nevertheless, if the LED started shining more dimly or stopped shining altogether it could be sign that it has been damaged.

All of the components could be easily replaced by just swapping them, since they're connected to the breadboard. The microcontroller is socketed, so it too could be easily replaced. In case of power supply failure the whole component would have to be replaced or if the problem is with the board, the quickest way would be to replace it whole.

4. References

- ATmega328P Datasheet (http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega328PB-Automotive-Data-Sheet-40001980B.pdf)
- DHT11 Datasheet (https://www.mouser.com/ds/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf)
- BMP280 Datasheet (https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMP280-DS001.pdf)

- HC-05 Datasheet (http://www.electronicaestudio.com/docs/istd016A.pdf)
- HD44780 Datasheet (https://www.sparkfun.com/datasheets/LCD/HD44780.pdf)
- Arduino reference materials:
  - Port manipulation: https://www.arduino.cc/en/Reference/PortManipulation
  - Digital pins use guide: https://www.arduino.cc/en/Tutorial/DigitalPins

5. Other information

Due to the fact that the screen and Bluetooth module use a lot of energy, the screen may sometimes flicker. Moreover, the Bluetooth app may sometimes display wrong data. It has been confirmed that is a problem with the app itself, since it does not occur with other apps or when using USB connection,