



Bao Vu – e1800948

Huong Dang – e1800943

Phuc Le – e1800951

Temperature & humidity sensor system

Final project of Embedded system design course 2021

Information Technology IT2018-3C-Group1-Vaasa University of
Applied Sciences

ABSTRACT

Author	Huong Dang, Bao Vu and Phuc Le
Title	Temperature and humidity sensor system design
Year	2021
Language	English
Name of Supervisor	Jani Ahvonen

The project is made for studying and researching of Embedded system design purpose. It is also a reference of how to use DHT22 – temperature and humidity sensor in an embedded system, then send the data to the master device. This project will focus on the DHT22 sensor and how to receive and send the signal to the master device. When the master sends out the request signal, we will respond by sending the value of humidity or temperature corresponding.

In this project, we design the system by testing it on the breadboard, using STM32L152RE to process the temperature and humidity data, then send the result to the master device. In this course, we used a python program that queries the data from different slave devices (one is our) and saves the data in the Wapice IoT ticket, and we have successfully sent our results to the IoT ticket.

The project is now completed. The hardware, communication is working on the breadboard and are now in the testing phase of the PCB, the results sending to the IoT ticket are true compare to other sensors in the environment.

Keywords	STM32L152RE, DHT22 temperature and humidity sensor, IoT Ticket, embedded system design
----------	--

CONTENTS

ABSTRACT

1	INTRODUCTION – BRIEFLY ABOUT PROJECT PROCESSING	4
2	SENSOR INFORMATION – DHT22.....	5
2.1	Theory briefly information about DHT22	5
2.2	Development of sensor code.....	8
3	MODBUS RTU FRAME.....	11
3.1	Protocol Description.....	11
3.1.1	RTU Mode.....	11
3.1.2	Address Field	11
3.1.3	Function Field.....	12
3.1.4	Error Checking Field.....	12
3.2	Slave frame implementation	13
3.2.1	Request frame	13
3.2.2	Response frame.....	14
3.3	Frame program development with C language	15
3.3.1	Sensor code (as shown in part 2 of this report).	16
3.3.2	USART RX Interrupt	16
3.3.3	Read the bytes from USART and respond to master	17
4	DESIGN AND BUILD THE CIRCUIT FOR DHT22 SENSOR USING MODBUS RS 485.....	23
4.1	Circuit component explanation	23
4.2	Design the circuit on PADs.	26
4.3	The final result	28
5	WAPICE IOT TICKET	30
5.1	What is IoT-Ticket	30
5.2	Upload final project on Wapice IoT-Ticket.....	30
5.2.1	Instruction for IoT-Ticket Master/Gateway	30
5.2.2	Create Dashboard and make real-time graph of DHT22 sensor on my.iot-ticket	34
6	CONCLUSION AND FINAL RESULT	37

1 INTRODUCTION – BRIEFLY ABOUT PROJECT PROCESSING

The purpose of this project is to learn how to design a sensor embedded system using STM32L152RE and process the received data. In this project, we choose the DHT22 sensor which is a temperature and humidity sensor. Then, we develop the code in C using the Modbus RTU network to contact the master device. After testing the design on a breadboard, we design the PCB using PADS logic and PADS layout, then in the processing of making the PCB. After testing the results received and sent to the master device using RealTerm, we used the python program which is mostly received from the teacher to finish sending our result to the Wapice IoT ticket.

Then we check the final result with our PCB, the conclusion of this project will be found in the last chapter of this report.

We appreciate the support from our teacher, Jani Ahvonen for leading our group to finish this project as well as VAMK for all the supplying components.

2 SENSOR INFORMATION – DHT22

2.1 Theory briefly information about DHT22

The most important component in this project is the DHT22 sensor. The DHT22 is a basic, low-cost digital temperature and humidity sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air and spits out a digital signal on the data pin (no analogue input pins needed). It is simple to use but requires careful timing to grab data. The only real downside of this sensor is you can only get new data from it once every 2 seconds. In Table 1, we have the technical specification information of the DHT22.

Table 1. Technical information

Model	AM2302	
Power supply	3.3-5.5V DC	
Output signal	digital signal via 1-wire bus	
Sensing element	Polymer humidity capacitor	
Operating range	humidity 0-100%RH;	temperature -40~80Celsius
Accuracy	humidity +2%RH (Max +-5%RH);	temperature +-0.5Celsius
Resolution or sensitivity	humidity 0.1%RH;	temperature 0.1Celsius
Repeatability	humidity +-1%RH;	temperature +-0.2Celsius
Humidity hysteresis	+-0.3%RH	
Long-term Stability	+-0.5%RH/year	
Interchangeability	fully interchangeable	

From the datasheet of DHT22, we study the electrical connection diagram to test our sensor on a breadboard with just a really simple design:

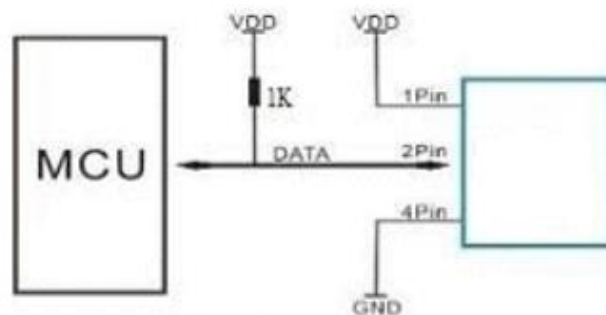


Figure 1. Electrical connection diagram

The power supply required for DHT22 is from 3.3V-5.5V DC, this power supply will be taken from STM32L152RE (this microcontroller can convert +12V to 3.3V because our power source for the device is +12V DC, so the whole design should be fitted to Nucleo152RE development board inside Arduino connectors).

Operating specification of DHT22:

The output data of DHT22 is 40 bits which will contain:

DATA = 16 bits Humidity data + 16 bits Temperature data + 8 bits checksum

The Nucleo-board will receive 40 bits of data from DHT22 as the first 16 bits will be humidity data, the next 16 bits will be temperature data and the last 8 bits will be the checksum for check the data. The checksum of this component is calculated as the sum of the first 8 bits of humidity plus the first 8 bits of the temperature signal.

Check-Sum (8 bits) = 8 first bits Humidity + 8 last bits Humidity + 8 first bits Temperature + 8 last bits Temperature

When MCU send the start signal, DHT22 change from standby status to running status. When MCU finishes sending the start signal, DHT22 will send a response signal of 40-bit data that reflect the relative humidity and temperature to MCU. Without a start signal from MCU, DHT22 will not give a response signal to MCU. One start signal for one response data from AM2302 that reflect the relative humidity and temperature. DHT22 will change to standby status when data collecting finished if it does not receive a start signal from MCU again.

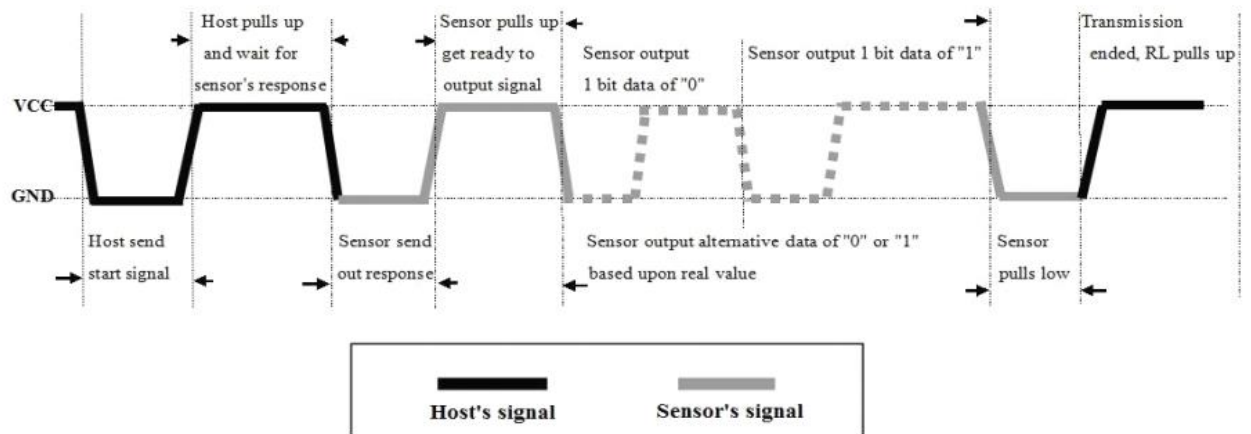


Figure 2. Overall communication process

As figure 2 shows, the interval of the whole process must be beyond 2 seconds.

To demonstrate the process of overall communication based on time processing, there are the following steps:

1. Step 1: STM32L152RE sends out start signal to DHT22 and DHT22 send a response signal to STM32

Data-bus's free status is high voltage level. When communication between STM32 and DHT22 begins, MCU will pull low data-bus and this process must be beyond at least 1~10ms to ensure DHT22 could detect STM32's signal, then STM32 will pull up and wait 20-40us for DHT22's response. (see Figure 3)

When DHT22 detect the start signal, DHT22 will pull low the bus 80us as a response signal, then DHT22 pulls up 80us for preparation to send data. (see Figure 3)

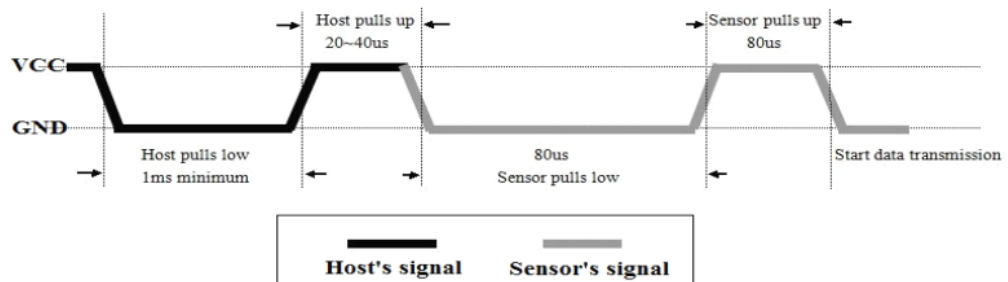


Figure 3. MCU send a start signal to DHT and DHT send a response signal to MCU

2. Step 2: DHT22 send data (40 bits) to MCU

When DHT22 is sending data to MCU, every bit's transmissions begin with low-voltage-level that last 50us, the following high-voltage-level signal's length decide the bit is "1" or "0". (see figure 4)

As can be seen from Figure 4, we can say that if the signal is pulled up from 26~28us, the bit data will be 0. In contrast, if the signal is pulled up for about ~70us, then the bit data will be 1. Between sending each bit of data, the DHT22 will pull down the signal for an unknown amount of time.

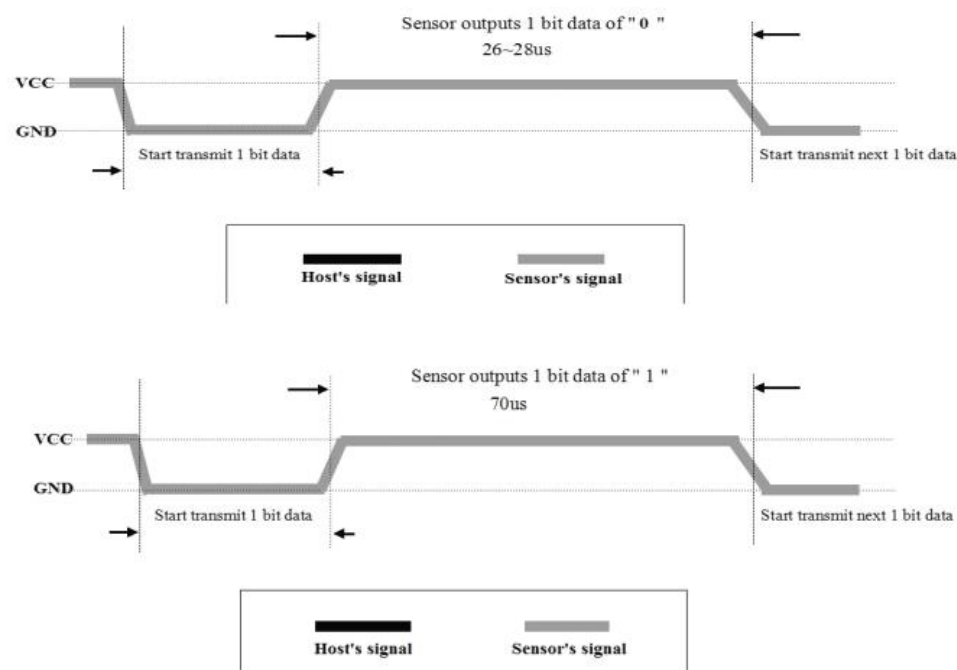


Figure 4. DHT22 send a data signal to MCU

2.2 Development of sensor code

We have the function `int read_sensor(int input_reg)`, which will send the signal to the DHT22, and receive the data from it. Below is our function for processing with DHT22, this function is used to take values of humidity and temperature recording to the `input_reg`.

```
/*Take the values from the sensor*/  
int read_sensor(int input_reg){
```



```

// DHT22 connected to PA6 (D12)
// Do not need to take the median value, due to a filter in dht22 built to get and
send back the good quality of measurements.
    unsigned int humidity=0, i=0, temperature=0;
    unsigned long long mask=0x80000000;           // mask will be shifted from left
to right

//step1:STM sends start signal, then DHT send response signal through PA6
GPIOA->MODER|=0x1000;           //set GPIOA pin 6 to output (p184) to send signal from
STM32 to DHT22
GPIOA->ODR|=0x40;               //0100 0000 set bit 6. p186
delay_Ms(10);                  //first, the data-bus's free status is high voltage
level
//Now, the communication between STM and DHT begins
GPIOA->ODR&=~0x40;              //STM32 pulls low for 1ms to ensure DHT22 could detect
STM32 signal
delay_Ms(1);
GPIOA->ODR|=0x40;               //pin 6 high state and sensor gives this 20us-40us =
STM32 pulls up and wait for DHT22 response
GPIOA->MODER&=~0x3000;          //set GPIOA pin 6 to input to receive the data from
DHT22

//step2: DHT22 send temperature, humidity signal to STM32
//the bits sent from the left to the right that means the humidity is the first 16
bits and checksum is the last
//8 bits transfered from dht22 to output register of stm32nucleo
//And the first 8-high bits are transfered first then 8-low bits came next in the
16 bits data resolution
    while((GPIOA->IDR & 0x40)){           //sensor send out response
    while(!(GPIOA->IDR & 0x40)){           //sensor pulls up get ready to output signal
    while((GPIOA->IDR & 0x40)){           //after this while() sensor pulls down = state
for transmission output
    while(i<32)
    {
        while(!(GPIOA->IDR & 0x40)){           // wait for the signal becomes high-voltage-
level
        delay_Us(35); // the signal goes high for 35us
        //after 35us, if the signal is still high, it means the output bit = 1
        //after 35us, if the signal is no longer high = !(GPIOA->IDR & 0x40), then
the output bit is = 0

        //if the signal high, use the mask to shift and change the bit to 1 respec-
tively
        //if not, then ignore it, that bit will still be 0 as defined
        if((GPIOA->IDR & 0x40) && i<16)
        {
            humidity=humidity|(mask>>16);           // the bits of humid-
ity value is right shifted by 16 positions from the high bist then the low bits
        }
        if((GPIOA->IDR & 0x40) && i>=16)
        {
            temperature=temperature|mask;
        }
        mask=(mask>>1);                               // mask
is right shifted 1 position each loops 32 times
        i++;

        while((GPIOA->IDR & 0x40)){           // wait for the low
data-bus to start transmitting next 1 bit data (wait for the signal becomes low)
    }

    /* Return the values */
    if(input_reg == 0x01){
        return (int)temperature;

```

```

    }
    else return (int)humidity;
    return 0;
}

```

After building code and circuit in a breadboard for testing, we have the following result as capture by using an oscilloscope:

The figure following is the signal we capture of the data-bus communication between the DHT22 and the STM32

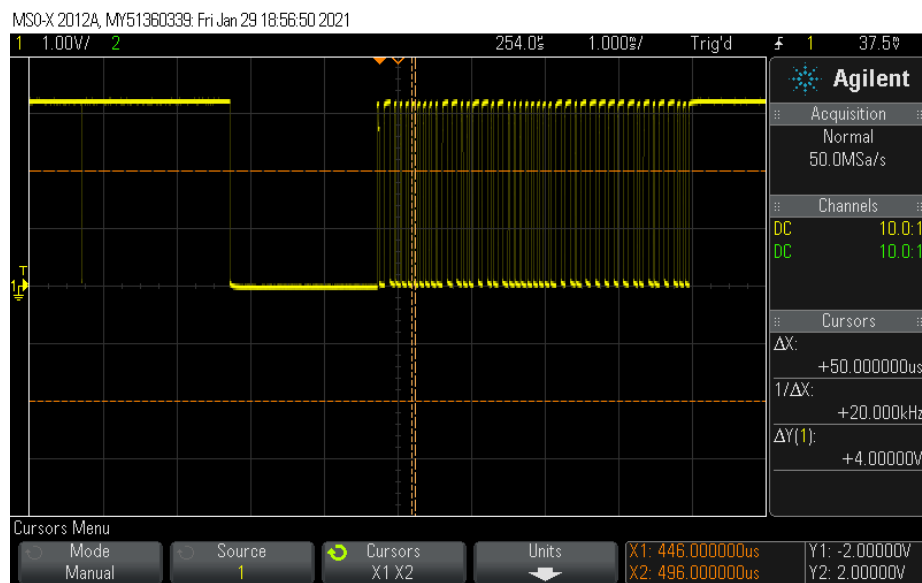


Figure 5. The signal communication between 2 devices

3 MODBUS RTU FRAME

3.1 Protocol Description

MODBUS Protocol is a messaging structure, widely used to establish master-slave communication between intelligent devices. A MODBUS message sent from a master to a slave contains the address of the slave, the 'command' (e.g. 'read register' or 'write register'), the data, and a checksum (LRC or CRC).

Since Modbus protocol is just a messaging structure, it is traditionally implemented using RS232, RS422, or RS485.

Controllers can be set up to communicate on standard Modbus networks using either of two transmission modes: ASCII or RTU.

In our project, RTU, RS485 and CRC will be implemented.

3.1.1 RTU Mode

When controllers are set up to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, each eight-bit byte in a message contains two four-bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII for the same baud rate. Each message must be transmitted in a continuous stream.

Coding System

Eight-bit binary, hexadecimal 0 ... 9, A ... F

Two hexadecimal characters contained in each eight-bit field of the message

Bits per Byte

1 start bit

8 data bits, least significant bit sent first

1 bit for even / odd parity-no bit for no parity

1 stop bit if parity is used-2 bits if no parity

Error Check Field

Cyclical Redundancy Check (CRC)

3.1.2 Address Field

The address field of a message frame contains two characters (ASCII) or eight bits (RTU).

The individual slave devices are assigned addresses in the range of 1 ... 247.

3.1.3 Function Field

The Function Code field tells the addressed slave what function to perform.

Function 01 (0x01) Read Coils

Function 02(0x02) Read Discrete Inputs

Function 03 (0x03) Read Holding Registers

Function 04 (0x04) Read Input Registers

Function 05 (0x05) Write Single Coil

Function 06 (0x06) Write Single Register

Function 15 (0x0F) Write Multiple Coils

Function 16 (0x10) Write Multiple Registers

In our project, we are going to use Function 4 Read Input Registers to read the binary contents of input registers in our slave.

3.1.4 Error Checking Field

When RTU mode is used for character framing, the error-checking field contains a 16-bit value implemented as two eight-bit bytes. The error check value is the result of a Cyclical Redundancy Check calculation performed on the message contents.

The CRC field is appended to the message as the last field in the message. When this is done, the low-order byte of the field is appended first, followed by the high-order byte. The CRC high-order byte is the last byte to be sent in the message.

3.2 Slave frame implementation

3.2.1 Request frame

Due to DHT22 capability, we can check the house temperature and humidity. So, we used Starting Address Lo to read those values by using 0x01 for temperature request frame and 0x02 for humidity request frame.

Therefore, we also have 2 different CRCs. Those are 0x6093 for temperature and 0x9093 for humidity.

Both values use the same method to convert the values and respond to master as temperature. We will use a temperature request frame to be an example.

Master send format for temperature:

Table 2. Format sent from master for temperature

Device Address	Function Code	Starting Address Hi	Starting Address Lo	Quantity Hi	Quantity Lo	CRC Hi	CRC Lo
08	04	00	01	00	01	60	93

Device	Slave address
Freezer temperature	01
Refrigenrator temperature	02
Dishwasher Leak detection	03
Smoke detection	04
Carbon monoxide detection	05
Magnetic switch door supervisor	06
Outdood temperature, humidity	07
House temperature, humidity	08
Motion (human) detection	09
Carbon dioxide measurement	0A
Light intensity measurement	0B

Figure 6 Slave addresses

We use the House temperature and humidity sensor with 0x08 for the slave address.

3.2.2 Response frame

Slave response format for temperature:

Table 3. Slave response format

Device Address	Function Code	Number of Bytes	Register value Hi	Register value Lo	CRC Hi	CRC Lo
08	04	02	01	07	25	63

The slave address and function code are the same values of the request frame.

The register value responded to master at the same time the slave was requested is 0x0107, converted to a decimal 263, the actual temperature is $263/10 = 26.3$ Celsius.

Table 4 Register value

Part of Data Package	Description	Value
08	Slave address	0x08 (8)
04	Function code	0x04 (4) - Read Input Registers
02	Byte count	0x02 (2)
01 07	Register value	0x0107 (263)
25 63	CRC	0x2563 (9571)

And CRC was calculated based on the first five bytes in the frame is 0x2563.

3.3 Frame program development with C language

The program structure would be like the flow chart below:

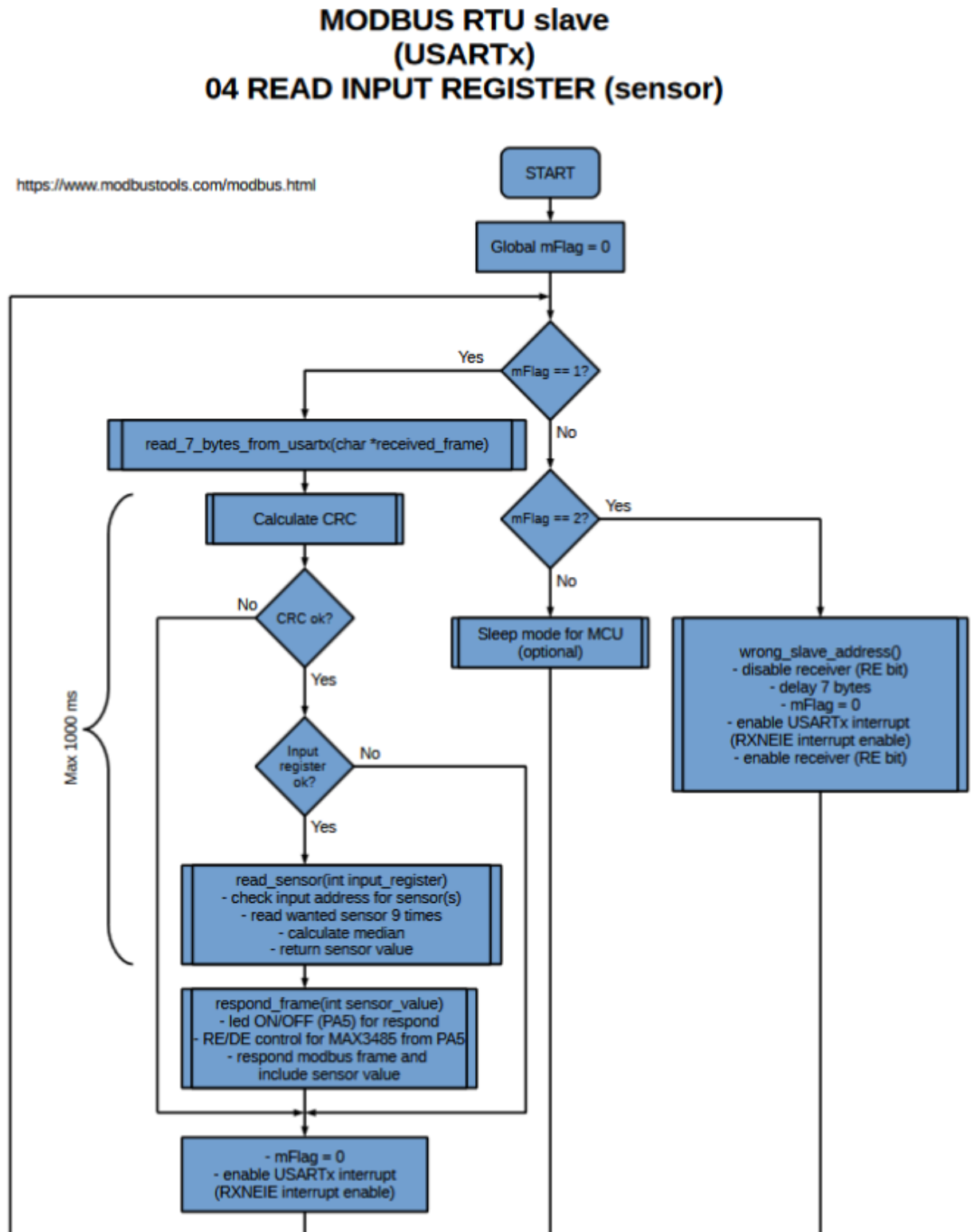


Figure 7 Modbus RTU frame programming flow chart

3.3.1 Sensor code (as shown in part 2 of this report).

Before programming the MODBUS RTU frame, we did investigate the DHT22 sensor operating specifications and wrote the program to read and show the values sent back from DHT22, which are humidity, temperature and checksum.

To implement the sensor code into the RTU frame, we customized the sensor code with some modifications.

The data is transferred and received via pin PA6 (D12) on the development board.

The returned value will be the data without the decimal, the master would convert it to the true value later.

3.3.2 USART RX Interrupt

The slave would take the input, which is a request frame from the master and sends back the response frame immediately. To do so we need to trigger the USART RX interrupt.

To communicate with the master via RS-485(MAX3485 chip), we are going to use the USART1 and trigger the RX interrupt.

```
int main(void)
{
    __disable_irq();    //global disable IRQs, M3_Generic_User_Guide p135.
    USART1_Init();

    /* Configure the system clock to 32 MHz and update SystemCoreClock */
    SetSysClock();
    SystemCoreClockUpdate();

    /* TODO - Add your application code here */

    USART1->CR1 |= 0x0020;    //enable RX interrupt
    NVIC_EnableIRQ(USART1_IRQn);    //enable interrupt in NVIC
    __enable_irq();    //global enable IRQs, M3_Generic_User_Guide p135
}
```


27.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 8 USART Control register 1

Enable the RX interrupt by putting 1 in bit 5 RXNEIE and also enable interrupt in NVIC.

```
void USART1_IRQHandler(void)
{
    int c = 0;

    //This bit is set by hardware when the content of the
    //RDR shift register has been transferred to the USART_DR register.
    if(USART1->SR & 0x0020) //if data available in DR register. p739
    {
        c = USART1->DR;
        if(c == 0x08)
        {
            mFlag = 1; // if the slave address is correct the flag is 1
            USART1->CR1 &= ~0x0020; //disable RX interrupt
        }
        else
        {
            mFlag = 2;
            USART1->CR1 &= ~0x0020; //disable RX interrupt
        }
    }
}
```

Every time master sends a request to the slave the USART1 interrupt will be triggered, then check if the device address sent in the request frame matches with our slave address then put the global variable flag to 1 and temporarily disable interrupt until this communication end.

3.3.3 Read the bytes from USART and respond to master

When the flag variable put to 1, the program would start to read the remaining bytes of the frame and store them in an array.

```
unsigned char request_frame[READ_LENGTH+1]={0x08}; // initialize the frame
with the slave address
int crc=0;
char crc_high=0;
char crc_low=0;
```

```

int sensor_value = 0;

/* Infinite loop */
while (1)
{
    if (mFlag == 1)
    {
        /* Read 7 bytes from sensor if the slave's address(0x08) is matched */
        read_7_bytes_from_usartx(request_frame);
    }
}

```

```

/* Read input from Terminal */
void read_7_bytes_from_usartx(unsigned char *received_frame)
{
    int i=0;

    //Assign the rest bytes of the request frame into the array
    for(i = 1; i <= READ_LENGTH; i++)
    {
        *(received_frame+i) = USART_read();
    }
}

```

And in the request frame, we calculated the CRC based on the first 6 bytes of the frame and compare it to the CRC sent by the master.

"080400010001" (hex)	
1 byte checksum	14
CRC-16	0x8860
CRC-16 (Modbus)	0x9360
CRC-16 (Sick)	0x45A3
CRC-CCITT (XModem)	0xA355
CRC-CCITT (0xFFFF)	0xAD45
CRC-CCITT (0x1D0F)	0x926B
CRC-CCITT (Kermit)	0x1D47
CRC-DNP	0xDDC6
CRC-32	0xDEDED4DFAF

Input type: ☐ ASCII ☒ Hex

Figure 9 CRC calculation

Our CRC calculation function will be output as the figure above but in the Modbus frame it has to be in reverse order, which means 0x6093. And the CRC calculated by the function needs to be divided into 2 bytes separately to compared with the CRC sent from the master in the request frame.

```

/* CRC calculated and divided from 2-bytes form into 1-byte form */
crc = CRC16(request_frame,6);
crc_high = (crc>>8);           //take out the 9 high bits (1 byte) of crc
crc_low = crc & 0xff;          //take out the 8 low bits of crc

```

If the CRC from the master and the calculation match each other, the program starts to read sensor values based on the Starting Address Lo in the request frame. This Starting Address is the third element in the frame.

```

/*Read the sensor values if the checksum in the frame is correct*/
if(crc_high==request_frame[7] && crc_low==request_frame[6]){

    // 0x08 0x04 0x00 0x01 0x00 0x01 0x60 0x93 (Temperature request frame)
    // 0x08 0x04 0x00 0x02 0x00 0x01 0x90 0x93 (Humidity request frame)
    sensor_value = read_sensor(request_frame[3]); //take the sensor value
    based on the starting address Lo(0x10: temp, 0x20:Hum)
    respond_frame(sensor_value); //use the returned sensor value to create
    the respond frame
    delay_Ms(20);
}

```

Finally, we constructed a response frame with the sensor values returned.

```

/*Response frame to Master*/
void respond_frame(int sensor_value){

    GPIOA->MODER|=0x400;
    GPIOA->ODR|=0x20; //0010 0000 set bit 5. p186, transmit max3485 ON <=>
    LED ON
    unsigned char respond[7]={0x08,0x04,0x02}; // the same de-
    vice address and function code as request frame. 0x02 means

        // 2 bytes of sensor value transfered to master
    int crc=0;

    // divide the sensor value into 2 number of 1-byte then put into respond
    frame
    respond[3] = (sensor_value>>8); // take the temperature
    respond[4] = sensor_value & 0xff; // take the decimal part

    // calculate the crc for respond frame, and assign it into the frame
    crc = CRC16(respond,5); // calculate the CRC for response
    frame
    respond[5] = crc & 0xff; // crc low
    respond[6] = (crc>>8); // crc high

    /* Write the data into Data register and master can take the data from
    there */
    for(int i = 0; i < READ_LENGTH; i++)
    {
        USART_write(respond[i]);
    }

    delay_Ms(100);
}

```

```

GPIOA->ODR&=~0x20;           //0000 0000 clear bit 5. p186, trans-
mit max3485 OFF <=> LED OFF
    delay_Ms(100);
}

```

In this response frame, we used the PA5 to control the RE/DE (Receiver Output Enable/Driver Output Enable) for the MAX3485 chip. Basically, the RE/DE is always in low state that means the MAX3485 is always receiving the request. When PA5 is set to 1 the RE/DE are in high state and the MAX3485 is transmitting the response to the master.

An array was created to store the frame, which is initialized with the device address, function code, and the number of sensor value bytes to send (0x02 for 2 bytes).

As we know the response frame has the same structure as the request frame but 1 byte less. So, the value read from the sensor will be divided into 2 separated bytes and put into the array.

CRC bytes are calculated and assigned to the array as well. When the array is already filled with the frame elements, the byte will be transmitted to master via USART1_DR.

```

void USART_write(char data)
{
    //wait while TX buffer is empty
    while(!(USART1->SR&0x0080)){ } //TXE: Transmit data register empty.
p736-737    USART1->DR=(data);           //p739
}

```

After all the bytes are sent to the master, the transmit of MAX3485 will be terminated, and get ready for new communication.

The figures below are examples of response frames.

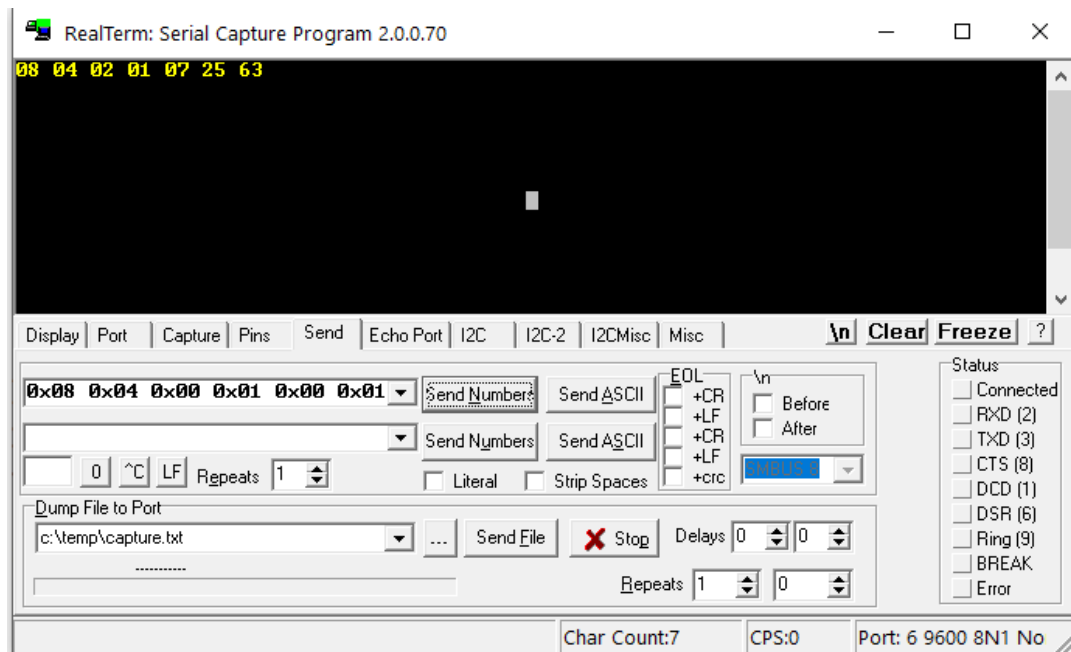


Figure 10 Temperature response frame

Temperature request frame is 0x08 0x04 0x00 **0x01** 0x00 0x01 **0x60 0x93**.

And the response frame is transmitted with 2 bytes of temperature data included that is 0x01 and 0x07.

We can convert these bytes to decimal and divide them by 10 to get the real temperature value. 0x0107 to decimal is 263 and the real value equals $263/10 = 26.3$ Celsius.

CRC is the last two bytes: 0x25 and 0x63, calculated by using the same method with the request frame.

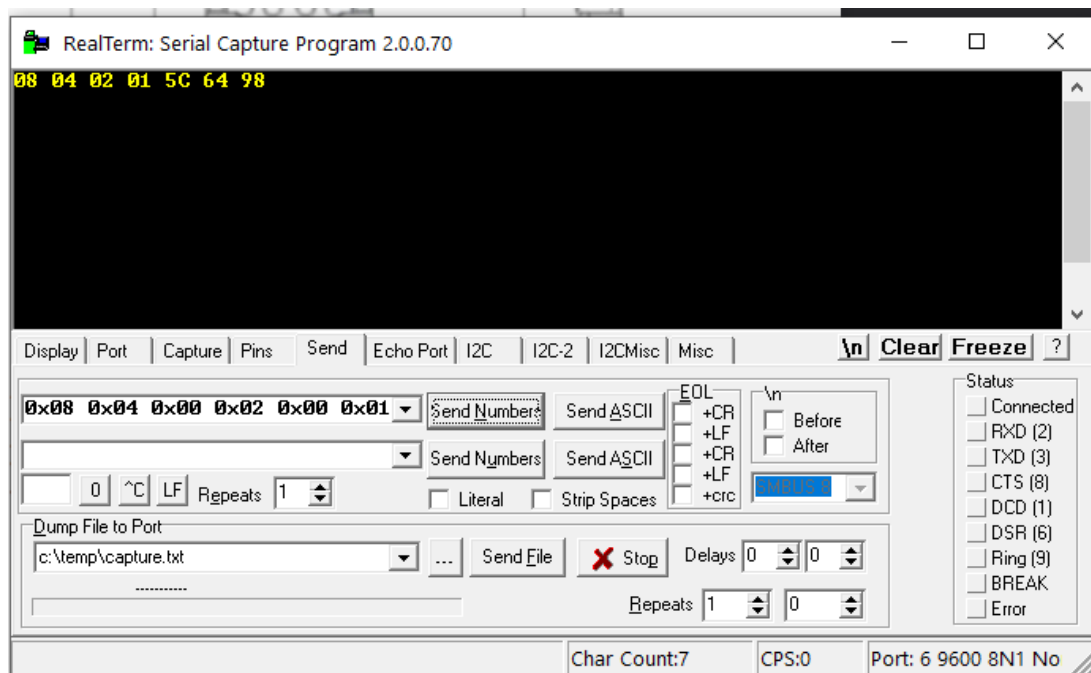


Figure 11 Humidity response frame

Humidity response frame is 0x08 0x04 0x00 **0x02** 0x00 0x01 **0x90 0x93**.

Two bytes of humidity value sent to master are 0x01 and 0x5C. Using the same conversion method, we got the house humidity of 34.8%.

4 DESIGN AND BUILD THE CIRCUIT FOR DHT22 SENSOR USING MODBUS RS 485

In this section, we will design a circuit for our DHT22 sensor (measure the Humidity and Temperature), connect the circuit to MODBUS RTU (RS-485) and make a PCB that will be fitted to the NucleoL152RE development board inside Arduino connectors.

4.1 Circuit component explanation

In this project, here is the list of components for the circuit:

- 1 TM_BLOCK/3P
- 1 TM_BLOCK/2P
- 1 MAX3485CPA+
- 1 USB to RS485 serial converter cable
- 1 PTC resistor as a fuse MF-MSMF050-2
- 1 DHT22 sensor
- 3 100n capacitor
- 1 100u capacitor

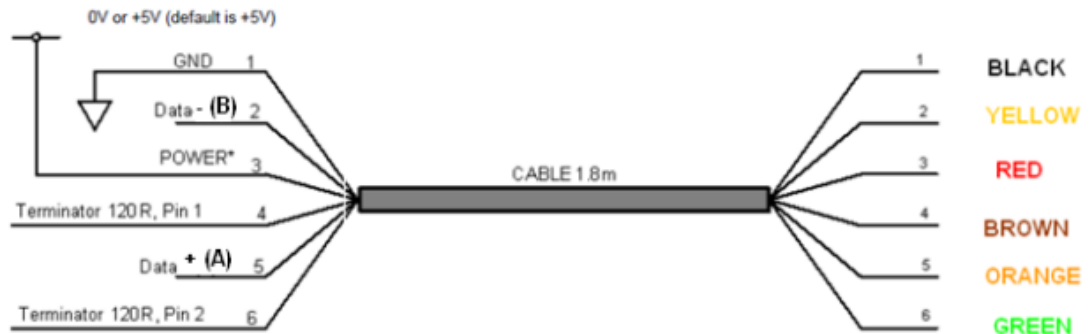


Figure 12 USB-RS485-WE connections

The USB-RS485 cable (as shown in figure 1) is a USB to RS485 levels serial UART converter cable incorporating FTDI's FT232RQ USB to serial UART interface IC device with handles all the USB signal and protocols.

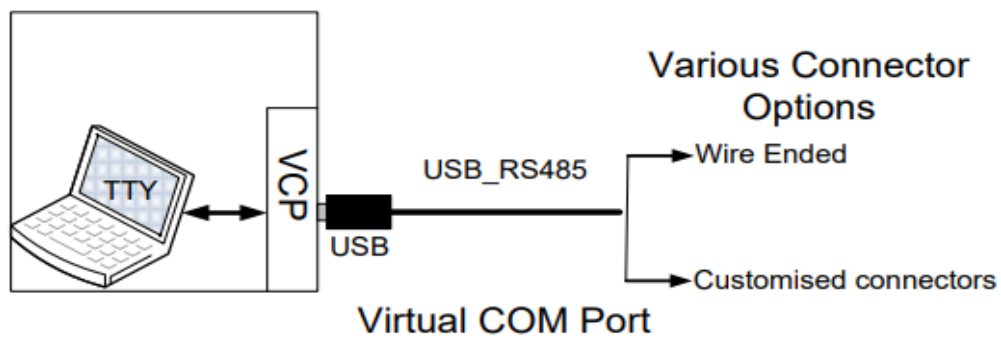


Figure 13 Using the USB-RS485 cable

The USB-RS485 cables require USB drivers, which are used to make the FT232R in the cable appear as a virtual COM port (as shown in figure 2). This allows the user to communicate with the USB interface via a standard PC serial emulation port.

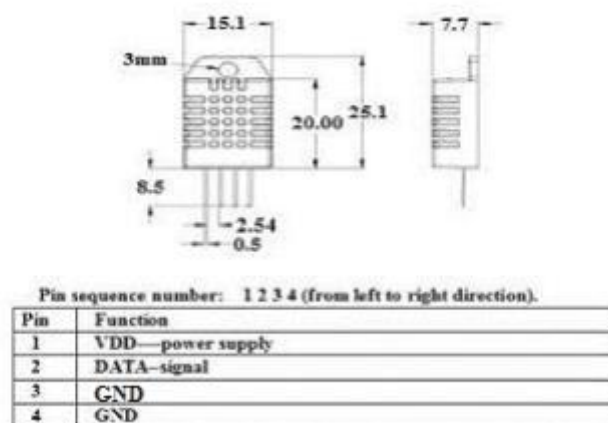


Figure 14 DHT22 Sensor Dimensions and Pin Configuration.

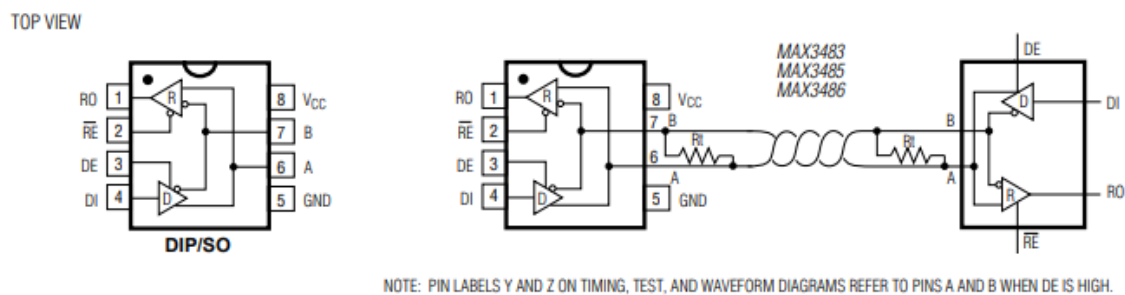


Figure 15 MAX3485 Pin Configuration and Typical Operating Circuit.

The MAX3485 (as shown in figure 9) are low-power transceivers for RS-485 communications. The MAX485 can transmit and receive at data rates up to 10Mbps. The MAX3485 are half-duplex. Driver Enable (DE) and Receiver Enable (RE) pins are included on the MAX3485. When disabled, the driver and receiver output are high impedance.

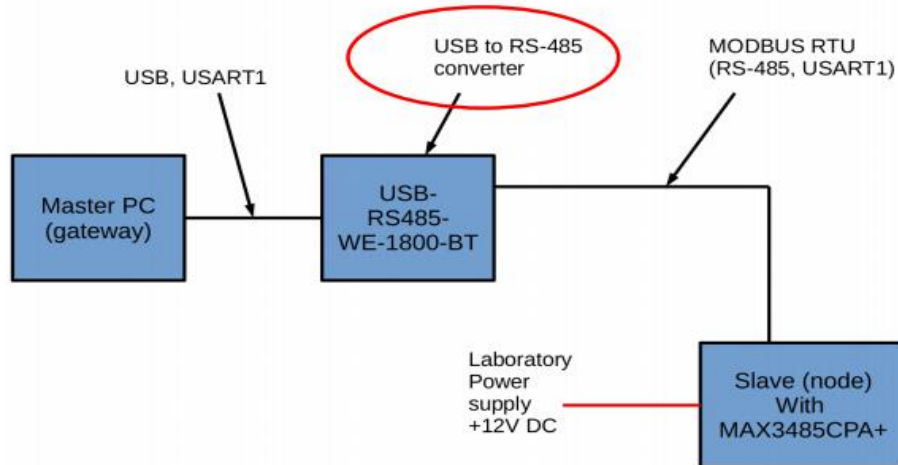


Figure 16 Block hardware.

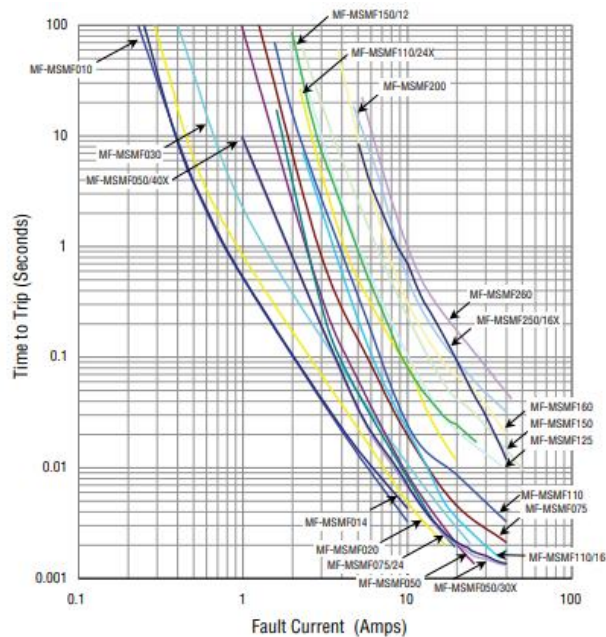


Figure 17 Typical Time to Trip at 23 °C for MF-MSMF050-2.

In this project, we will use the MF-MSMF050-2 fuse to protect automotive electronics when overcurrent and overtemperature. Moreover, we need an MF-MSMF050-2 fuse to protect the whole Modbus system, if any Slave in the system is damaged, it will not affect other Slaves.

4.2 Design the circuit on PADs.

Using the PADs Logic, we have drawn our Modbus system as the figure below:

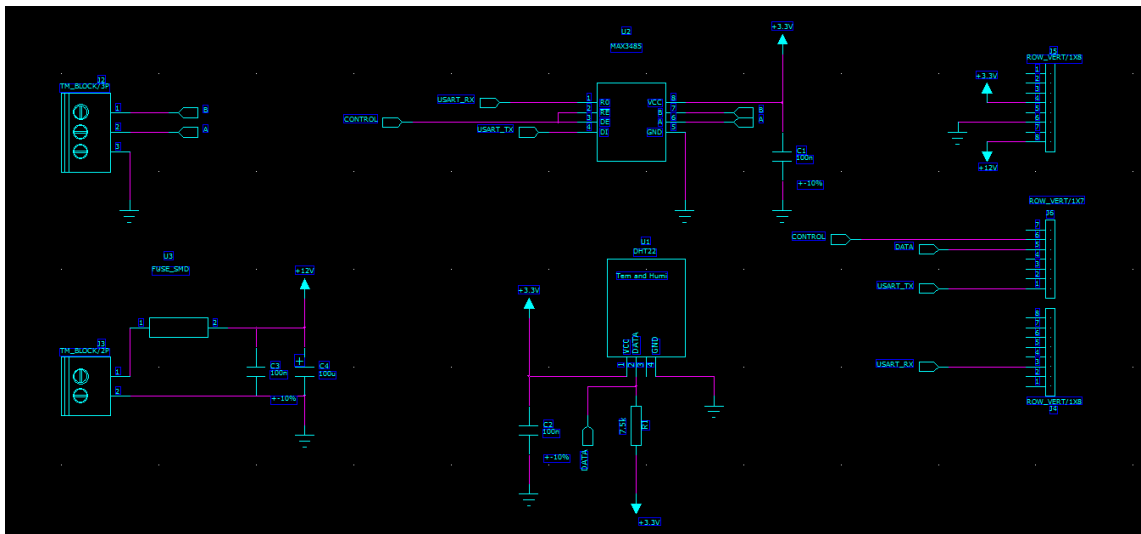


Figure 18 PADs Logic circuit of Modbus system and DHT22.

Important connections in the circuit that we made (as shown in Figure 12):

- Pin 2 from DHT22 sensor to pin 5 which is GPIO port PA6 from Nucleo Board to receive data from the sensor.
- Pin 4 from MAX3485CPA+ directly to pin 9 which is GPIO port PA10 from Nucleo Board to connect USART1 TX.
- Pin 1 from MAX3485CPA+ directly to pin 3 which is GPIO port PA10 from Nucleo Board to connect USART1 RX.
- Pin 2 and pin 3 from MAX3485CPA+ to pin 6 which is GPIO port PA5 from Nucleo Board.
- One 7.5k Ohm's resistor connected to pin 2 of the DHT22 sensor to keep the data line high and to enable the communication between the sensor and STM32.
- Four capacitors 100nF and 100uF are connected to +12V, +3.3V to help stabilize the power supply, shut off the noise.

From this circuit, we had the PADs Layout as follow:

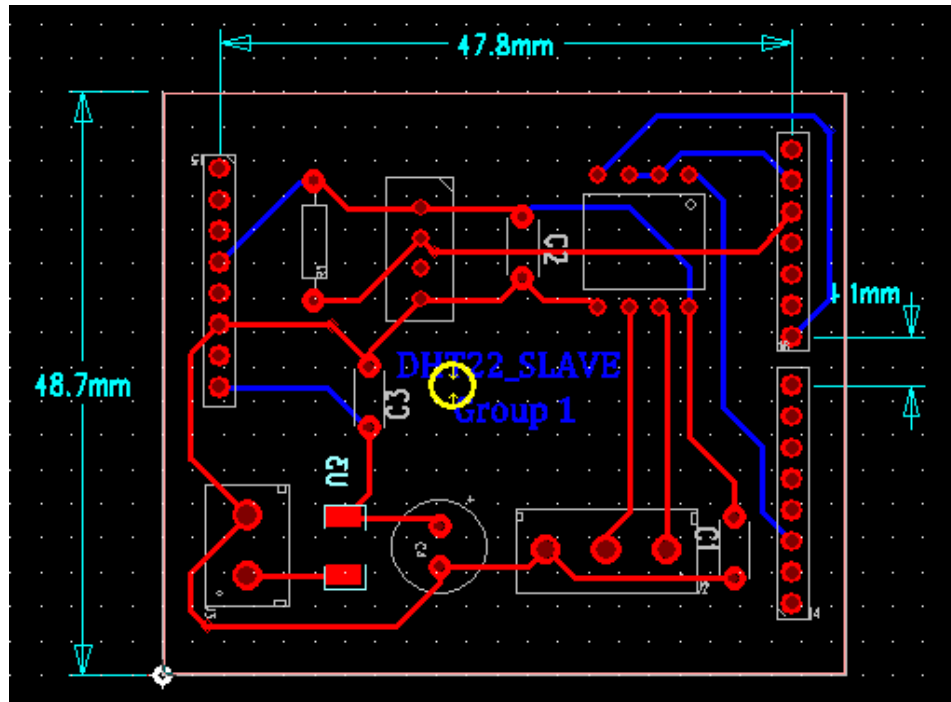


Figure 19 PADs layout.

In our case, we must use 47.8mm width instead of the standard width of 48.26mm because the different printers may print different sizes of output. So, we need to calibrate the width to fit the output to the STM32 board. Here is our CAM preview for 2 layouts: bottom and top:

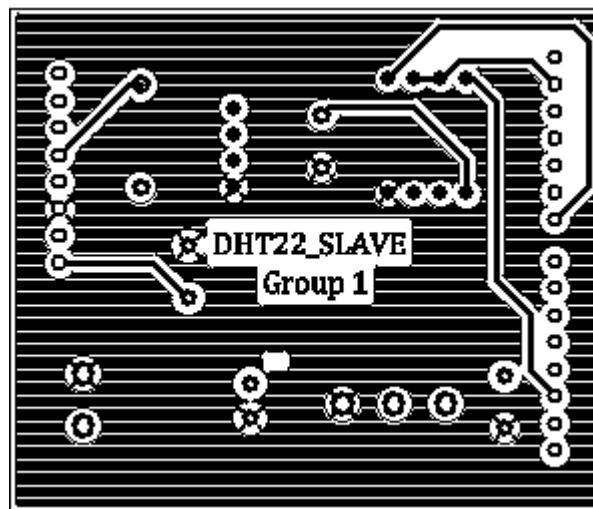


Figure 20 Top component and traces preview.

Our bottom layout only contains traces since we have put every component in the top layer.

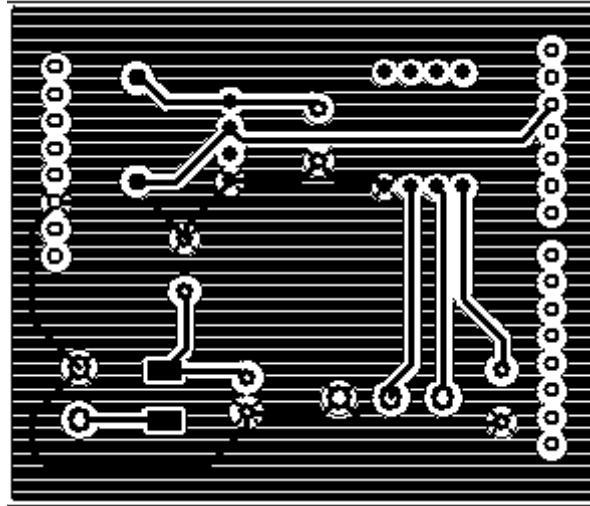


Figure 21 Bottom traces, with the via holes.

4.3 Test the system on breadboard

After we have the circuit diagram, now we can test our system on breadboard (as shown in Figure 22)

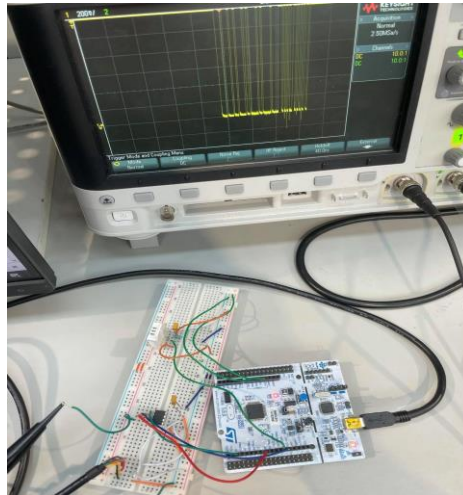


Figure 22 Test the system on breadboard

4.4 The final result

After designing the circuit on PADS and test the system, now we can build the Modbus and DHT22 sensor circuit on a PCB as follow:

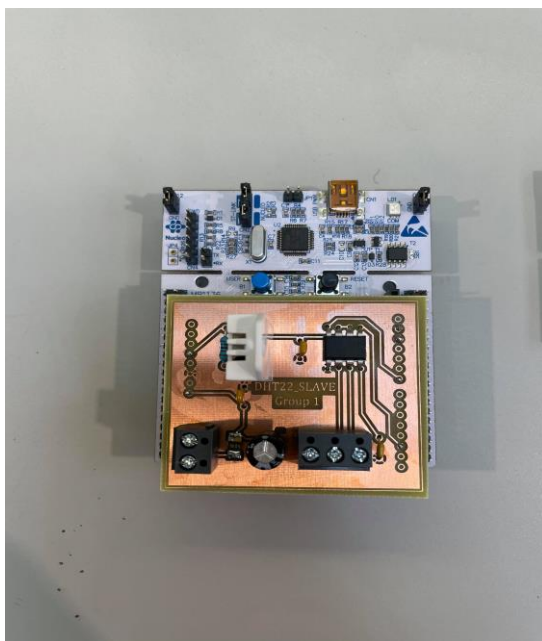


Figure 23. Final PCB

5 WAPICE IOT TICKET

5.1 What is IoT-Ticket

IoT-TICKET® is a complete Internet of Things (IoT) tool suite and platform allowing you to make web, mobile, cloud and reporting applications in minutes with big data analytics and straightforward tools. From drag & drop content building to plug & play connectivity, edge computing and scalable enterprise management.

5.2 Upload final project on Wapice IoT-Ticket

In this section, we will create a student profile on the dashboard and make a real-time graph of the DHT22 humidity and temperature sensor.

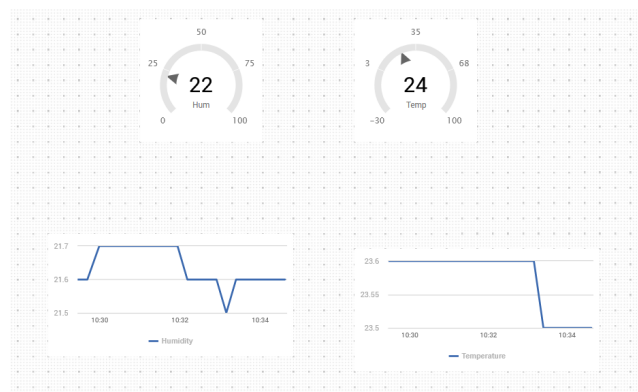


Figure 24 DHT22 real-time graph on IoT-Ticket

5.2.1 Instruction for IoT-Ticket Master/Gateway

- Install Python Idle. Python Idle can be found from this link: https://portal.vamk.fi/pluginfile.php/702386/mod_assign/intro/python%20%281%29.exe



Figure 25 Installation of Python 3.8.3.

- Download the IoT-Ticket python client and save it in your Python folder. Download file can be found from this link: https://portal.vamk.fi/pluginfile.php/702386/mod_assign/intro/iotticketmaster.zip

« Users > e1800948 > AppData > Local > Programs > Python > Python38

Figure 26 Python 3.8.3 path.

- Download the Master.py file which will send a request to your Modbus system, receive data from sensors and upload to IoT-Ticket
- Download the pyserial-3.4 serial folder to your python folder. Download file can be found from this link: https://portal.vamk.fi/pluginfile.php/702386/mod_assign/intro/pyserial.zip
- Log in to my.iot-ticket, choose “Management Mode” => “Student” => “Add data acquisition” => add your device (as shown in Kuva 4)

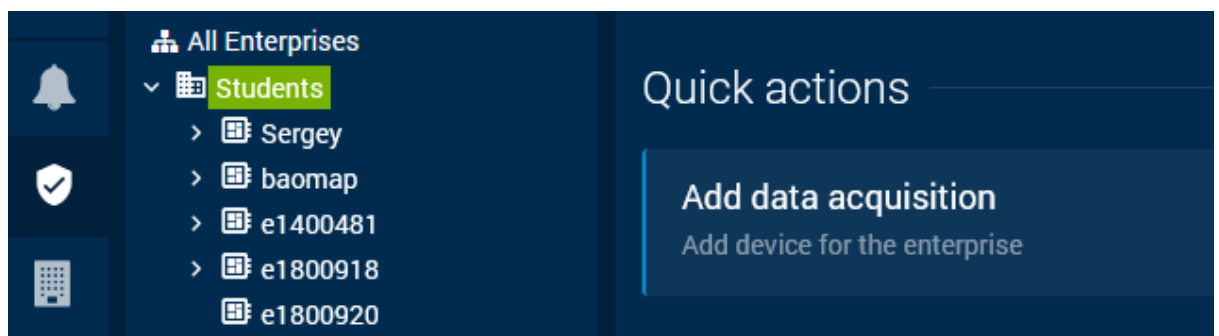


Figure 27 Our IoT-Ticket resource browser.

Students > Add a device

Communication method: MQTT, Name: DHT22, Type: Sensor

Manufacturer: None, Model: , Manufacturer serial number: None

Description: This is a humidity and temperature sensor

Metadata: No added metadata. Add key, Add value, + Add

Figure 28 IoT-Ticket Device's profile

- In your IoT-Ticket library file, find a JSON file and modify it as your IoT-Ticket username, password and your device ID which you can find from the resource browser.

```
{
  "username": "E1800951@VAMK.FI",
  "password": "*****",
  "deviceId": "rNuIAri8Hi9x1oO8jQXu59",
  "baseurl": "https://my.iot-ticket.com/api/v1/"
}
```

Figure 29 config.json file modification

- Install driver for RS485. Download file can be found from this link: <https://portal.vamk.fi/mod/resource/view.php?id=536954>

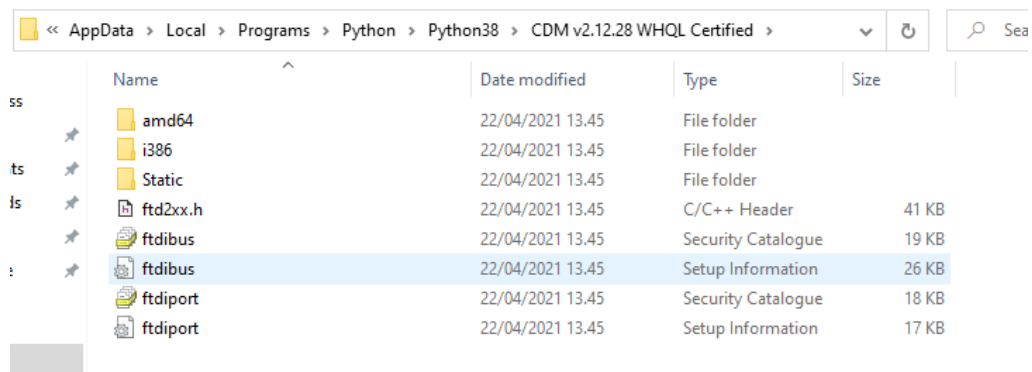


Figure 30 config.json file modification

- Open masterpython.py and modify the file.
 - Change the path of your config.json into your config.json path


```
data = json.load(open("demo\config.json"))
username = data["username"]
password = data["password"]
deviceId = data["deviceId"]
baseurl = data["baseurl"]
```

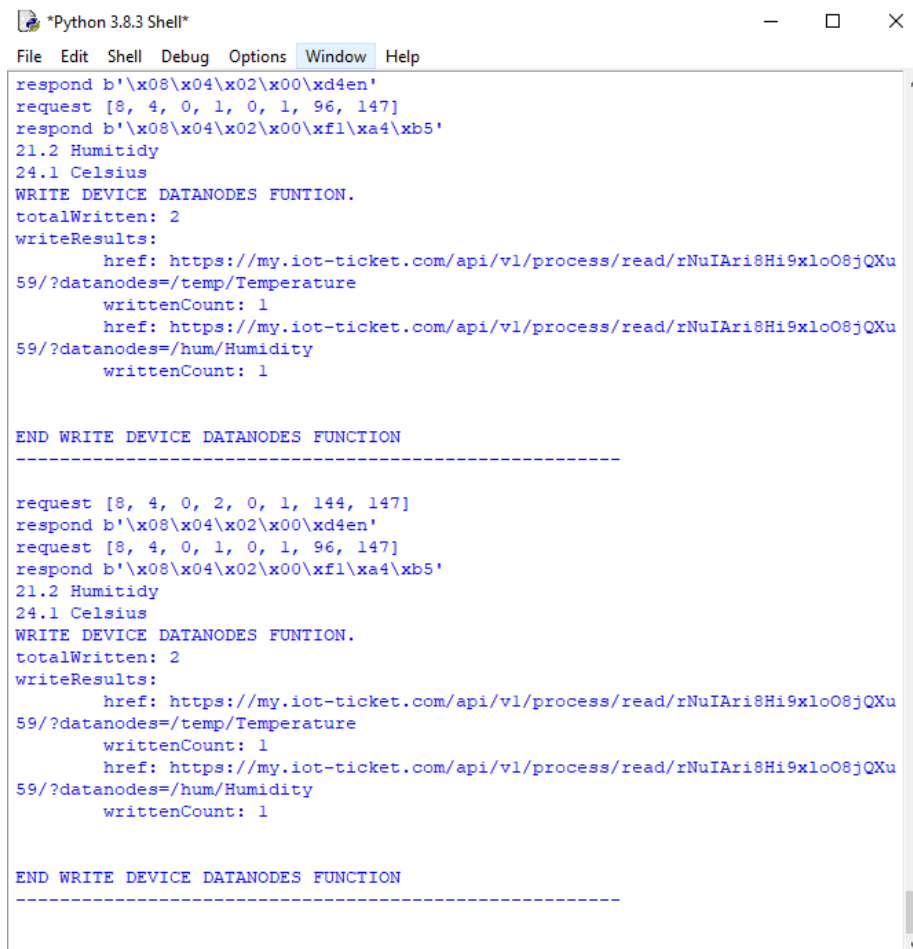
- In function modbus_request, we need to take the median (value number 3 and 4) value of 9 value that the sensor responds. But in our case, we are not necessary to do that since the DHT22 sensor has already been hard-code to return the median value, and each response is 2 seconds, 9-time respond will be too long.
- Change your received port, you can find your port on Device Manager => Ports (COM & LPT)
- In the last “while” function, hardcode one string for humidity response, one for temperature response (as shown in figure 30)

```
while True:
    hexadecimal_string1 = [0x08,0x04,0x00,0x01,0x00,0x01,0x60,0x93];
    hexadecimal_string2 = [0x08,0x04,0x00,0x02,0x00,0x01,0x90,0x93];
    #write datanode demo
    value=modbus_request(hexadecimal_string2)/10.0; #humidity value
    value2=modbus_request(hexadecimal_string1)/10.0; #read temperature and divided by 10 (*-1 for demo)
    print(value,"Humidity")
    print(value2,"Celsius")
    send_data_to_iot_ticket(value,value2);
    time.sleep(10);
```

Figure 31 "While" function for sending a request to Modbus system

Note: You must put the masterpython.py, serial folder in the same folder with IoTTicket-PythonLibrary-master folder.

- Test the Modbus system on the breadboard



```
*Python 3.8.3 Shell*
File Edit Shell Debug Options Window Help

respond b'\x08\x04\x02\x00\xd4en'
request [8, 4, 0, 1, 0, 1, 96, 147]
respond b'\x08\x04\x02\x00\xf1\xa4\xb5'
21.2 Humidity
24.1 Celsius
WRITE DEVICE DATANODES FUNTION.
totalWritten: 2
writeResults:
  href: https://my.iot-ticket.com/api/v1/process/read/rNuIAri8Hi9xloO8jQXu
59/?datanodes=/temp/Temperature
  writtenCount: 1
  href: https://my.iot-ticket.com/api/v1/process/read/rNuIAri8Hi9xloO8jQXu
59/?datanodes=/hum/Humidity
  writtenCount: 1

END WRITE DEVICE DATANODES FUNCTION
-----

request [8, 4, 0, 2, 0, 1, 144, 147]
respond b'\x08\x04\x02\x00\xd4en'
request [8, 4, 0, 1, 0, 1, 96, 147]
respond b'\x08\x04\x02\x00\xf1\xa4\xb5'
21.2 Humidity
24.1 Celsius
WRITE DEVICE DATANODES FUNTION.
totalWritten: 2
writeResults:
  href: https://my.iot-ticket.com/api/v1/process/read/rNuIAri8Hi9xloO8jQXu
59/?datanodes=/temp/Temperature
  writtenCount: 1
  href: https://my.iot-ticket.com/api/v1/process/read/rNuIAri8Hi9xloO8jQXu
59/?datanodes=/hum/Humidity
  writtenCount: 1

END WRITE DEVICE DATANODES FUNCTION
-----
```

Figure 32 Testing the system with Masterpython.py

5.2.2 Create Dashboard and make a real-time graph of the DHT22 sensor on my.iot-ticket.

We will make a readable graph of humidity and temperature values over time which offers by IoT-Ticket. By doing so, it will be easier for us to access our data.

To create IoT-Ticket Dashboards, go to my.iot-ticket.com, click on “Dashboard Browser” => “Create Dashboard” => Enter Dashboard name (Figure 9).

Create Dashboard ×

Dashboard Name

DHT22 Sensor

Templates

☐ Empty Layout

Create

Cancel

Figure 33 Dashboard Tab

After you create Dashboard. A board will pop up where you can edit your interface. In Widgets, choose “AngularGauge” and “Chart” as figure 32.

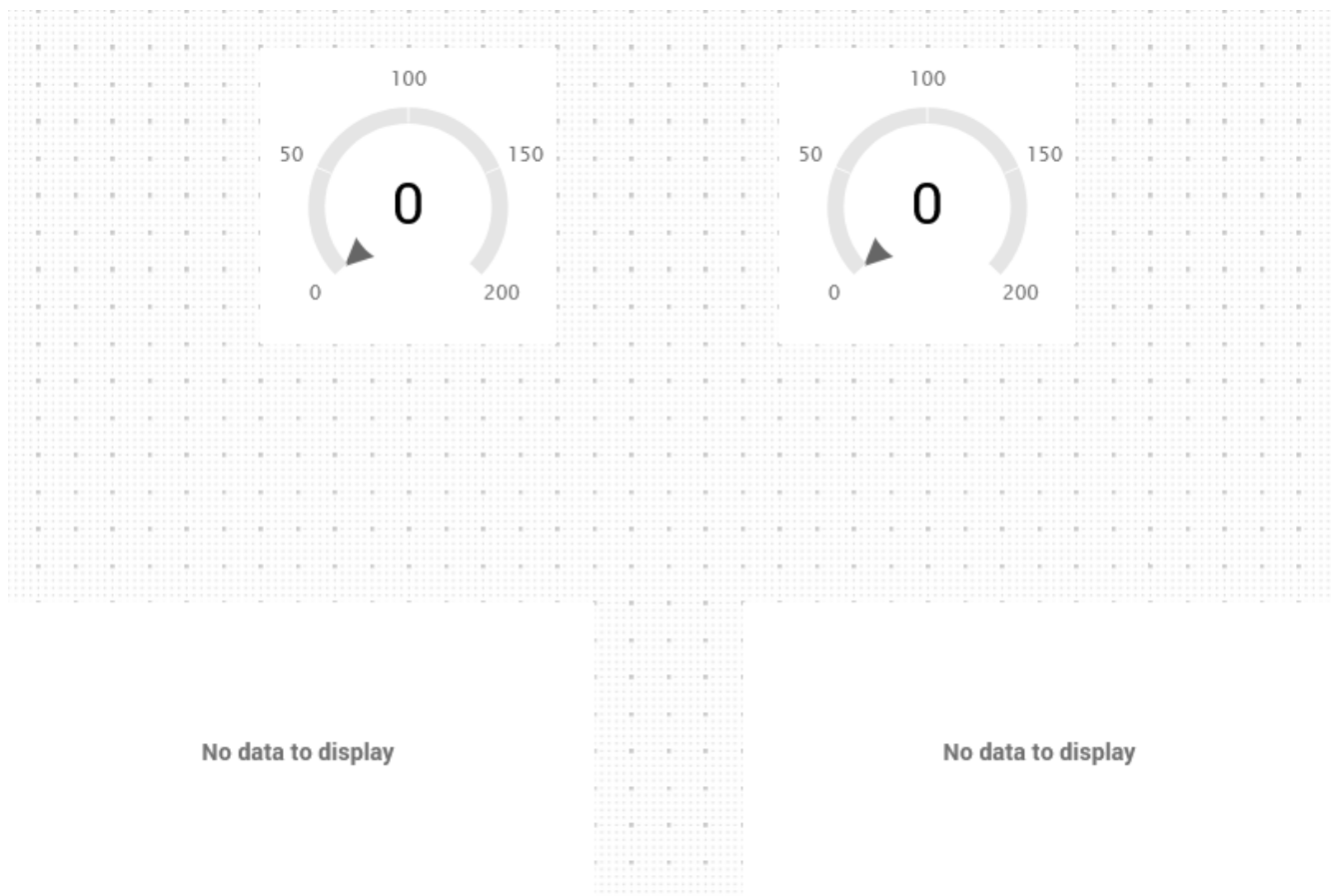


Figure 34 Data graph

After you save the file, now you can test your system on IoT-Ticket.

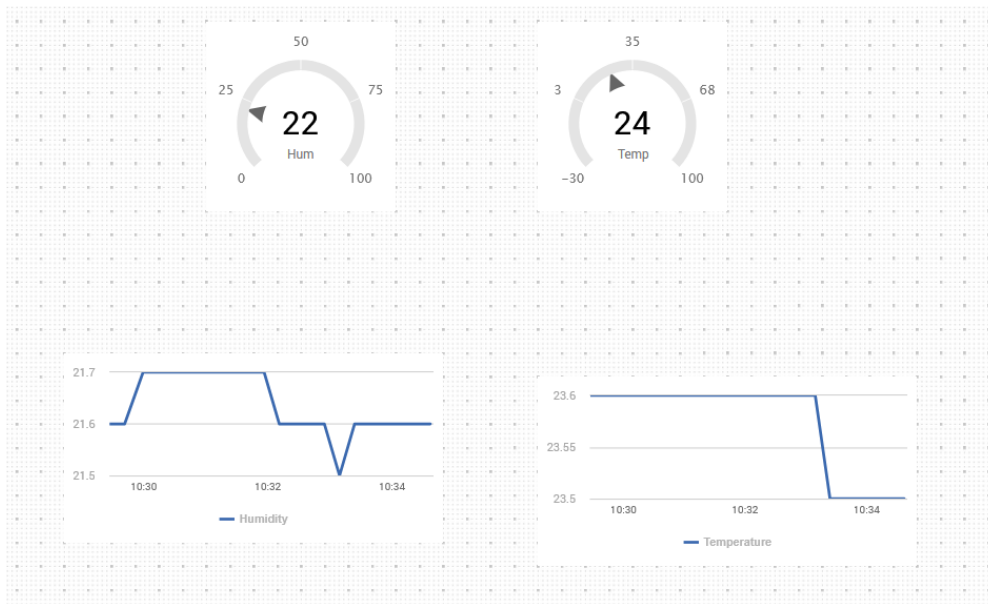


Figure 35 Testing the whole system.

6 CONCLUSION AND FINAL RESULT

The purpose of this project is to learn how to design and build an embedded system with a sensor and then send the data due to the request from the master device as a slave system. Our team has chosen DHT22 – a temperature and humidity sensor to build the system and make the PCB as well as send our data to the Wapice IoT ticket.

After uploading code into STM32L152RE, and connect cable, implement PCB and run to see the result, our project is now completed.

The breadboard testing is working great with the results shown in the IoT ticket as in part 5 of this report, but the PCB testing is now still having some problem. We are trying to fix the PCB to show the result.

The temperature and humidity data recorded by DHT22 is approximately the same as the result we have in our phone temperature and humidity sensor. It seems like there are some differences since the data we captured was in the laboratory environment.

During making this system, our team has learnt how to work with an embedded system, especially know how to work with a new method of the process of the data captured by the sensor. We have completed the project with the breadboard testing and the PCB is now in the last process, even though we face some problems with the final result of this PCB but we will try our best to make it work in the future.