A picture containing icon

Description automatically generated

|  |
| --- |
| Nguyen Hoang - e1900313  Hoang Hieu - e1900314  Hong Huynh - e1900318 |
| LIGHT INTENSITY SENSOR SYSTEM |
| Report for Embedded System Design Course |
|  |

|  |
| --- |
| Information Technology IT2019-3C |
|  |

VAASAN AMMATTIKORKEAKOULU

UNIVERSITY OF APPLIED SCIENCES

Information Technology

**ABSTRACT**

Author Hoang Nguyen, Hieu Hoang and Hong Huynh

Title Light intensity sensor system

Year 2022

Language English

Name of Supervisor Jani Ahvonen

The purpose of this project is for studying and understanding of embedded system design. It is also a reference for how to make use of Light intensity sensor, and then transfer the data to the master device. The project focusing on the LDR sensor and how to send / receive data result to the master device.

Firstly, we have tested the system on bread board using STM32L152RE-Nucleo, using Lux Meter to compare with the LDR sensor and develop the equation in Excel, after that we develop sensor code to receive the LDR sensor data, then send the data to the master device. In this course, we also draw the circuit diagram with LTspice and PAD logics.

Keywords STM32L152RE, Light intensity sensor, LDR, Modbus RTU NSL 19M51.

**CONTENTS**

ABSTRACT

[1 Introduction 4](#_Toc100658802)

[2 light intensity Sensor (NSL 19M51) 5](#_Toc100658803)

[2.1 Description about NSL 19M51 5](#_Toc100658804)

[2.2 Development of sensor code 6](#_Toc100658805)

[3 Design the circuit on PADs logic and Ltspice 12](#_Toc100658806)

[3.1 PADS Logic 12](#_Toc100658807)

[3.2 Ltspice 13](#_Toc100658808)

[4 MODBus rtu 14](#_Toc100658809)

[4.1 Protocol Description 14](#_Toc100658810)

[4.1.1 RTU Mode 15](#_Toc100658811)

[4.1.2 Address Field 15](#_Toc100658812)

[4.1.3 Function Field 15](#_Toc100658813)

[4.1.4 Contents of the Error Checking Field 16](#_Toc100658814)

[4.2 Slave frame 16](#_Toc100658815)

[4.2.1 Request frame 16](#_Toc100658816)

[4.3 Frame program development 19](#_Toc100658817)

[4.3.1 USARTx Interrupt 20](#_Toc100658818)

[4.3.2 Read the bytes from USART 21](#_Toc100658819)

[5 Conclusion 24](#_Toc100658820)

[References 25](#_Toc100658821)

# Introduction

In this project, we will study and learn about embedded system design. We will also use the light intensity sensor to operate the system, and then transfer the data to a master device, with this project focusing on the LDR sensor.

Then, we develop Modbus RTU frame using C programming language to connect the master device. After testing design the circuit on a breadboard and developing sensor code, we made the design of circuit using PADs logic and LTspice. Finally, testing the data sent/received to the master device using Real Term.

Along with our members, we would like to thank Jani Ahvonen, our supervisor, for intructions and provide all the components for our group to finish this project.

# light intensity Sensor (NSL 19M51)

## Description about NSL 19M51

The NSL-19M51 is a photoconductive cell on a TO-18 ceramic plastic encapsulated for moisture resistance. Photoresistors are Semiconductor devices that use light energy to control the flow of electrons, and hence the current flowing through them. The commonly used Photoconductive Cell is called the Light Dependent Resistor or LDR.

([Photocell, Ceramic Substrate (farnell.com)](https://www.farnell.com/datasheets/77395.pdf))

Table

Description automatically generated

Figure 1. Technical Information

We have the electrical connection diagram to test our sensor on a breadboard:

Diagram, schematic

Description automatically generated

Figure 2. Circuit Diagram

**Signal conditioning:**

The NSL-19M51 is an analog sensor. To perform from A-to-D conversion on these signals, the transducer produce output as voltage.

Diagram

Description automatically generated

Figure 3. Getting analog data to the CPU

## Development of sensor code

**Equation development:**

After finished the circuit connection (Figure 2), to make the **Equation** for the sensor, we will record the voltage of the LDR (using a Multimeter) along with the Lux values collect from the Lux Meter. Connect circuit to the Multimeter and put the Lux meter beside the LDR (Figure 4).

Diagram

Description automatically generated

Figure 4. Connect to Multimeter

Diagram

Description automatically generated

Figure 5. Lux meter next to LDR

The results after recording in Excel:

Table

Description automatically generated with low confidence

Figure 6. Excel records

We have made another column for ADC to compare with formular (third column):

Because the ADC of STM32L152 ARM Microcontroller has 12-bit resolution with a maxor the accuracy of the result, we only use the values above the red line with ADC under **4091**.

After that we will create an Excel chart from the values of column **LUX** and **ADC,** find the nearest **Trendline,** and dislay **Equation:**

Chart

Description automatically generated

Figure 7. Equation for sensor

**Equation:**

**Sensor code:**

This section is the code of developed sensor with comments. After reading, the result of sensor will be printed to Realterm in Lux.

|  |
| --- |
| **int read\_sensor(int input\_register){**  **int** result=0;  **double** lux=0;  **char** buf[] = "";  **int** lux\_degree = 0;  **int** lux\_decimals = 0;  /\* Configure the system clock to 32 MHz and update SystemCoreClock \*/  SetSysClock();  SystemCoreClockUpdate();  RCC->AHBENR |= 1; //enable GPIOA clock  RCC->AHBENR |= 4; //enable GPIOA clock  GPIOA->MODER |= 0x3; //PA0 analog (A0)  //setup ADC1. p272  RCC->APB2ENR |= 0x00000200; //enable ADC1 clock  ADC1->CR2 = 0; //bit 1=0: Single conversion mode  ADC1->SQR1 = 0;// conversion sequence length 1  ADC1->CR2 |= 1;// enable ADC1  // initialize USART2 to transmit at 9600 Baud  USART2\_Init();  /\* Infinite loop \*/  **while** (1) {  ADC1->CR2 |= 0x40000000;// start a conversion  **while** (!(ADC1->SR & 2)) {} //wait for conversion complete  result = ADC1->DR;// read conversion result  //EQUATION  lux=-0.5825\*(**double**)result+2537.7;  // Print Decimal Value  lux = lux \* 100; //remove decimals (34.54 = 3454)  lux\_degree = (**int**) lux / 100;  lux\_decimals = **abs**((**int**) lux % 100);  sprintf(buf, "%d.%d lux ", lux\_degree, lux\_decimals);  //Print results  **int** len = 0;  **while** (buf[len] != '\0'){  len++;  }  **for** (**int** i = 0; i < len; i++) {  USART2\_write(buf[i]);  }  USART2\_write('\n');  USART2\_write('\r');  **for** (**int** i = 0; i < len; i++) {  buf[i]=0;  }  //if (input\_register == 0x01){  return lux;  }    delay\_Ms(500);  }  } |

After building code and circuit in a breadboard for testing, we have the following result in Realterm when voltage at **2.358 V**:

A picture containing graphical user interface

Description automatically generated

Figure 8. Capture 1

A picture containing text, indoor

Description automatically generated

Figure 9. Covering sensor with 1 hand

# Design the circuit on PADs logic and Ltspice

## PADS Logic

A screenshot of a computer

Description automatically generated with low confidence

Figure 10. PADS Logic

**Connections made in circuit (Figure 9):**

* From sensor connect to pin A0 from Nucleo Board to receive data from the sensor.
* A 1.5K resistor between LDR to VCC.
* Two capacitors C1 100nF connect between VCC and GND, C2 100nF between LDR and pin A0 to GND to help stablize the supply and reduce the noise.

## Ltspice

Diagram

Description automatically generated with medium confidence

Figure 11. LTspice circuit

**Simulation:**

Chart, line chart

Description automatically generated

Figure 12. Vout

Chart, line chart

Description automatically generated

Figure 13. Vout + Vin

# MODBus rtu

## Protocol Description

* **Modbus RTU** is an open serial protocol derived from the Master/Slave architecture originally developed by Modicon (now Schneider Electric). It is a widely accepted serial level protocol due to its ease of use and reliability. Modbus RTU is widely used within Building Management Systems (BMS) and Industrial Automation Systems (IAS). Modbus RTU messages are a simple 16-bit structure with a CRC (Cyclic-Redundant Checksum). The simplicity of these messages is to ensure reliability. Due to this simplicity, the basic 16-bit Modbus RTU register structure can be used to pack in floating point, tables, ASCII text, queues, and other unrelated data. Since Modbus protocol is just a messaging structure, it is traditionally implemented using RS232, RS422, or RS485.
* **The Request**  
  The function code in the request tells the addressed slave device what kind of action to perform. The data bytes contain any additional information that the slave will need to perform the function. For example, function code 03 will request the slave to read holding registers and respond with their contents. The data field must contain the information telling the slave which register to start at and how many registers to read. The error check field provides a method for the slave to validate the integrity of the message contents.
* **The Response**  
  If the slave makes a normal response, the function code in the response is an echo of the function code in the request. The data bytes contain the data collected by the slave, such as register values or status. If an error occurs, the function code is modified to indicate that the response is an error response, and the data bytes contain a code that describes the error. The error check field allows the master to confirm that the message contents are valid.
* **ASCII Mode**  
  When controllers are setup to communicate on a Modbus network using ASCII (American Standard Code for Information Interchange) mode, each eight-bit byte in a message is sent as two ASCII characters. The main advantage of this mode is that it allows time intervals of up to one second to occur between characters without causing an error.

### RTU Mode

Modbus RTU mode is the most common implementation, using binary coding and CRC error-checking. ASCII mode uses ASCII characters to begin and end messages whereas RTU uses time gaps (3.5-character times) of silence for framing.

* **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)

### 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.

### Function Field

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

01 (0x01) Read Coils Status

02 (0x02) Read Discrete Inputs Status

03 (0x03) Read Holding Registers

04 (0x04) Read Input Registers (**Used in this project**)

05 (0x05) Write Single Coil

06 (0x06) Write Single Register

15 (0x0F) Write Multiple Coils

16 (0x10) Write Multiple Registers

### Contents of the Error Checking Field

Two kinds of error-checking methods are used for standard Modbus networks. The error checking field contents depend upon the method that is being used.

**ASCII**  
When ASCII mode is used for character framing, the error-checking field contains two ASCII characters. The error check characters are the result of a Longitudinal Redundancy Check (LRC) calculation that is performed on the message contents, exclusive of the beginning colon and terminating CRLF characters.  
The LRC characters are appended to the message as the last field preceding the CRLF characters.

**RTU**

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.

## Slave frame

### Request frame

For LDR sensor, we used Starting Address Lo to read value by using 0x01 and Starting Address Hi 0x00 for light intensity request frame.

Checksum CRC calculated by CRC calculator:

<https://www.lammertbies.nl/comm/info/crc-calculation>

Graphical user interface, table

Description automatically generated

Figure 14. CRC-16 (Modbus)

We order **605F** in the frame.

Master sends format for light intensity:

Format sent from master

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Device Address | Function Code | Starting Address Hi | Starting Address Lo | Quantity Hi | Quantity Lo | Error Check Hi | Error Check Lo |
| 04 | 04 | 00 | 01 | 00 | 01 | 60 | 5F |

Table

Description automatically generated

Figure 15. Slave addresses

We use the light intensity sensor with 0x04 for the slave address.

Text, table

Description automatically generated with medium confidence

Figure 16. Modbus Parser

## Frame program development

**The frame developed as flow chart below:** Diagram

Description automatically generatedFigure 17. Modbus RTU frame flow chart

### USARTx Interrupt

First, the slave takes a request from the master and send back the response, in order to do that we need to enable USART RX Interrupt. RS-485 is used so we are going to use the USART2 and enable the RX interrupt.

|  |
| --- |
| int main(void)  {  \_\_disable\_irq(); //global disable IRQs, M3\_Generic\_User\_Guide p135.  USART2\_Init();  /\* Configure the system clock to 32 MHz and update SystemCoreClock \*/  SetSysClock();  SystemCoreClockUpdate();  /\* TODO - Add your application code here \*/  USART2->CR1 |= 0x0020; //enable RX interrupt  NVIC\_EnableIRQ(USART2\_IRQn); //enable interrupt in NVIC  \_\_enable\_irq(); //global enable IRQs, M3\_Generic\_User\_Guide p135 |

Table

Description automatically generated with medium confidence

Figure 18. USART Control register 1

Enable the RX interrupt:

|  |
| --- |
| void USART2\_IRQHandler(void)  {  int received\_slave\_address=0;  //This bit is set by hardware when the content of the  //RDR shift register has been transferred to the USART\_DR register.  if(USART2->SR & 0x0020) //if data available in DR register. p739  {  c = USART2->DR;  if(c == 0x04)  {  mFlag = 1;// if the slave address is correct the flag is 1  USART2->CR1 &= ~0x0020;//disable RX interrupt  }  else  {  mFlag = 2;  USART2->CR1 &= ~0x0020;//disable RX interrupt  }  }  } |

### Read the bytes from USART

If the flag equal 1, all the remaining bytes of the frame will be read and stored in an array.

|  |
| --- |
| void read\_7\_bytes\_from\_usartx(unsigned char \*received\_frame)  {  int i=0;  for(i = 1; i <= READ\_LENGTH; i++)  {  \*(received\_frame+i) = USART2\_read();  }  } |

CRC had been calculated and compare it to the CRC sent by the master.

Graphical user interface, table

Description automatically generated

Figure 19 CRC calculation

The CRC result above is 0x5F60 but in the Modbus frame it must be in reverse order, which means 0x605F.

The main code:

|  |
| --- |
| while (1)  {  if (mFlag == 1)  {  /\* Read 7 bytes from sensor \*/  read\_7\_bytes\_from\_usartx(request\_frame);  /\* CRC calculated \*/  crc = CRC16(request\_frame[0],6);  /\*Read the sensor values if the checksum correct\*/  if(crc == 0x605F){  sensor\_value = read\_sensor(request\_frame[3]);  //respond\_frame(sensor\_value); //not finished  delay\_Ms(20);  }  /\* Disable receiver \*/  USART2->CR1 &= ~0x04;  /\* mFlag = 0, ready for a new request \*/  mFlag = 0;  /\* enable USARTx interrupt \*/  USART2->CR1 |= 0x0020;  /\* Enable receiver \*/  USART2->CR1 |= 0x04;  }  else if (mFlag == 2)  {  /\* Disable receiver\*/  USART2->CR1 &= ~0x04;  /\* Delay for 7 bytes \*/  delay\_Ms(10);  /\* mFlag = 0 \*/  mFlag = 0;  /\* enable USARTx interrupt \*/  USART2->CR1 |= 0x0020;  /\* Enable receiver \*/  USART2->CR1 |= 0x04;  }  }  return 0; |

Trasmit through USART2\_DR:

|  |
| --- |
| void USART2\_write(char data)  {  //wait while TX buffer is empty  while(!(USART2->SR&0x0080)){} //TXE: Transmit data register empty. p736-737  USART2->DR=(data); //p739  } |

# Conclusion

Through this project, we have learnt how to develop an embedded system design using a sensor, and then sent the data to a master device. This project will focus on the LDR sensor.

The light intensity data recorded by the sensor is just slightly different from the result of the Lux meter. It might be because of the equation we made not one hundred percent correct but for that reason we have borrowed the sensor to keep digging and testing it at home for a better result.

After several of laboratories, our team has become familiar with the development process of STM-Microcontroller-based device. Furthermore, we have learnt how to develop the interface for peripheral devices using both an electronic and a software approach.

# References

**[1] NSL-19M51 sensor**

[Photocell, Ceramic Substrate (farnell.com)](https://www.farnell.com/datasheets/77395.pdf)

**[2] Mobus RTU**

<https://www.modbustools.com/modbus.html>

**[3] CRC calculation**

<https://www.lammertbies.nl/comm/info/crc-calculation>

**[4] Modbus Parser**

<https://rapidscada.net/modbus/ModbusParser.aspx>

**[5] STM32 Nucleo-64 boards**

https://www.st.com/resource/en/user\_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf