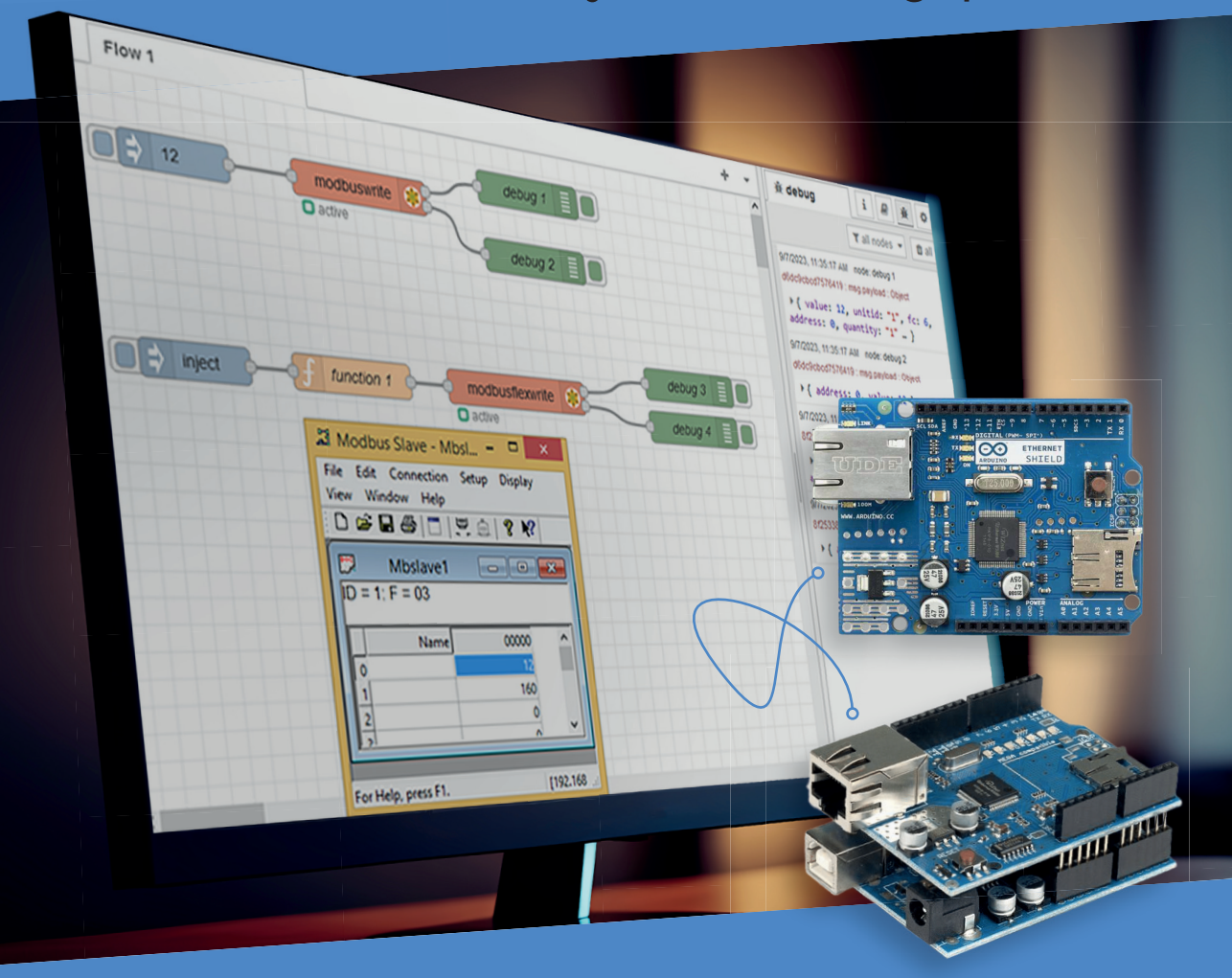
elektor books

Coding Modbus TCP/IP for Arduino

Example projects with Node-RED, MQTT, WinCC SCADA, Blynk, and ThingSpeak



Dr. Majid Pakdel

elektor
design > share > earn

Coding Modbus TCP/IP for Arduino

Example projects with Node-RED, MQTT, WinCC SCADA, Blynk, and ThingSpeak

Dr. Majid Pakdel

elektor
design › share › earn

This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
PO Box 11, NL-6114-ZG Susteren, The Netherlands
Phone: +31 46 4389444

Declaration

The authors and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause.

ISBN 978-3-89576-614-5 Print
ISBN 978-3-89576-615-2 eBook

www.elektor.com
Prepress Production: D-Vision, Julian van den Berg
Printers: Ipskamp, Enschede, The Netherlands

Contents

Preface

This book provides a comprehensive and detailed guide on implementing the ModbusTCP/IP protocol with Arduino development boards. Some of the information presented was derived from my personal notes taken during Emile Ackbarali's "How to Program an Arduino as a Modbus TCP/IP Client & Server" training course on Udemy. Additionally, I have successfully established TCP/IP communication between two Arduino Uno boards.

The book also delves into the topic of Modbus TCP/IP communication in the Node-RED environment and explains the process of interfacing Modbus TCP/IP with WinCCSCADA. Also, it demonstrates how to control sensors using Ethernet and the Blynk app, and explores the utilization of the MQTT protocol with Ethernet/ESP8266. Projects are showcased that combine Arduino, Node-RED, and Blynk IoT cloud, providing guidance on using the MQTTprotocol in the Node-RED environment. Finally, the book elucidates how to connect to the ThingSpeak IoTcloud using Ethernet/ESP8266.

I dedicate this book to the loving memory of my mother, Rahimeh Ghorbani, a great and kind soul who dedicated her entire blessed life to education and love for her children. Regrettably, I did not fully appreciate her while she was alive, and only after her passing did I realize what a precious gem she was. Unfortunately, I lost her, and nothing in the world can replace her. I hope she will forgive me and I pray that her soul rests in eternal peace.

Chapter 1 • Introduction

Modbus TCP/IP (also Modbus-TCP) is simply the Modbus RTU protocol with a TCP interface that runs on Ethernet. The Modbus messaging structure is the application protocol that defines the rules for organizing and interpreting the data, independent of the data transmission medium.

TCP/IP refers to the Transmission Control Protocol and Internet Protocol, which provides the transmission medium for Modbus TCP/IP messaging. Simply stated, TCP/IP allows blocks of binary data to be exchanged between computers. It is also a world-wide standard that serves as the foundation for the World Wide Web. The primary function of TCP is to ensure that all packets of data are received correctly, while IP makes sure that messages are correctly addressed and routed. Note that the TCP/IP combination is merely a transport protocol, and does not define what the data means or how the data is to be interpreted (this is the job of the application protocol, Modbus in this case).

So, in summary, Modbus TCP/IP uses TCP/IP and Ethernet to carry the data of the Modbus message structure between compatible devices. That is, Modbus TCP/IP combines a physical network (Ethernet) with a networking standard (TCP/IP) and a standard method of representing data (Modbus as the application protocol). Essentially, the Modbus TCP/IP message is simply a Modbus communication encapsulated in an Ethernet TCP/IP wrapper.

The automation industry wants you to be able to program Modbus which means you should be able, for example, to do the following tasks:

- Program an Arduino to be a Modbus TCP/IP Client/Server.
- Write an Android App to read Modbus data.
- Write a Windows Modbus Master application using Microsoft.Net.

All these tasks require more advanced knowledge of Modbus than just using it to interconnect devices that are already Modbus-compliant. In this book, we will focus on the first one. We are going to learn how to program an Arduino to be a Modbus TCP/IP Client/Server. At the end of this book, you will be able to create your own custom Modbus TCP/IP Client/Server hardware device using the Arduino Prototyping System.

You are going to build, configure and program an Arduino microcontroller board to be as follows:

- A Modbus TCP Client device capable of reading/displaying Modbus data from a TCP server device.
- A Modbus TCP Client device capable of writing Modbus data to a TCP server device.
- A Modbus TCP Server device capable of being read by any Modbus TCP Client application.

You will be using the Arduino IDE software to write the code. We are going to develop code gradually so you will learn everything in this book.

Chapter 2 • Hardware

In this section, we will provide a list of the hardware components required for the examples in this book. We will examine each component in detail, explaining their purpose. Firstly, we have the Arduino Uno, which serves as the system's central component and development board. Alongside the Arduino Uno, we have other variations such as Mega and Nano, all falling under the category of different types of Arduino Uno's.

On top of that, there is the Ethernet Shield for Arduino Uno, which acts as an interface converter. We also have other shields like the Modbus RS485 shield for Arduino, CAN bus Shield, and Profibus Shield, each serving different purposes. Since the Arduino's communication voltage level is TTL (ranging from zero to five volts), we need to convert it to Ethernet for use on our local network, particularly for Modbus TCP.

The third component is the Ethernet Network Hub, typically functioning as a multi-port hub for Ethernet connections from various devices. It can operate as a standard Ethernet hub, without the need for a switch.

Additionally, the hardware setup phase will include interconnecting cables. To summarize, the three primary components required for our purposes are as follows:

- The Arduino Uno.
- The Ethernet Shield for Arduino Uno.
- The Ethernet Network Hub.

Chapter 3 • The Arduino Uno

In this book, our focus will be on programming and configuring the Arduino as a TCP/IP Client and Server. While the Arduino has numerous features, we will only discuss the ones necessary for our tasks, therefore excluding the majority of its features. Figure 3.1 illustrates the top view of the Arduino Uno.

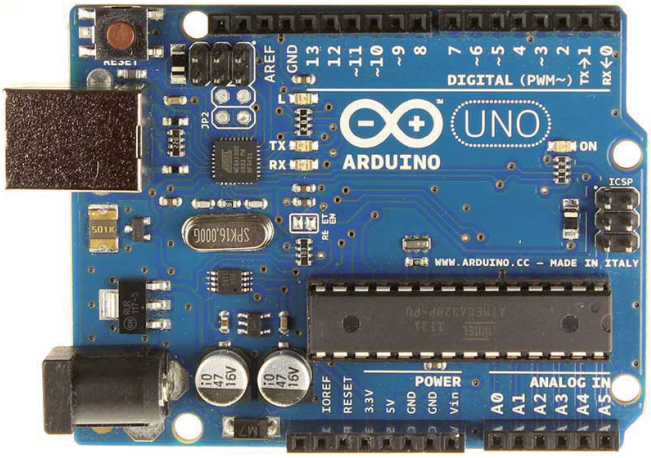


Figure 3.1: The top view of the Arduino Uno.

The figure labeled as Figure 3.2 displays the various components of the Arduino Uno. The first element is the USB interface, which serves two purposes: programming the Arduino development board and providing power to the Arduino. Essentially, the Arduino draws power from the USB interface. Additionally, the Arduino Uno can be powered by an AC/DC converter through an External Power Supply.

Furthermore, the digital pins section includes TX and RX female pins, which function as serial communication pins at the TTL level (ranging from zero to five volts). These pins facilitate serial communication with other devices and with the computer via the USB interface for programming. Digital devices can be connected through the digital pins, while analog devices (within the voltage range of zero to five volts) can be connected to the analog pins.

The power pins play a role in supplying power to both the digital and analog pins. In our case, we will utilize the digital pins to connect to an LCD display. This display will be used to present the data stored in the memory locations of the Arduino Uno.

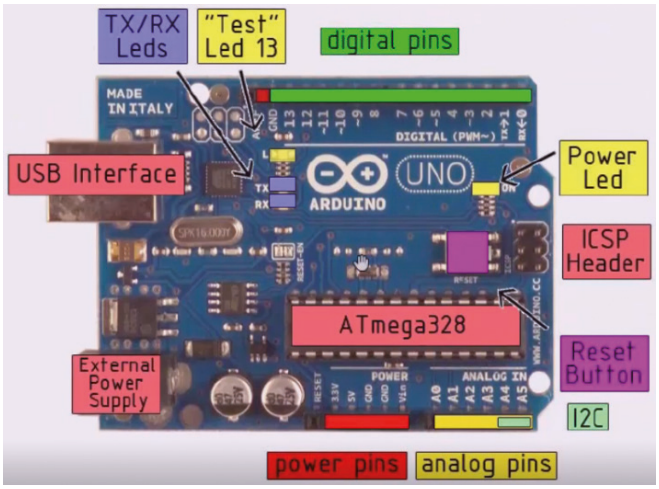


Figure 3.2: The diagram of the Arduino Uno.

Figure 3.3: The 3D view of the Arduino Uno development board.

Figure 3.3 presents the visual depiction of the Arduino Uno development board in a 3D format. This provides a comprehensive understanding of the physical features and dimensions of the Arduino Uno hardware, which is known for its simplicity and compact size.

Chapter 4 • Ethernet Shield for Arduino Uno

In this section, our focus will be on the Arduino Ethernet shield. Figure 4.1 illustrates how the shield appears. It is designed to perfectly fit on top of the Arduino Uno.

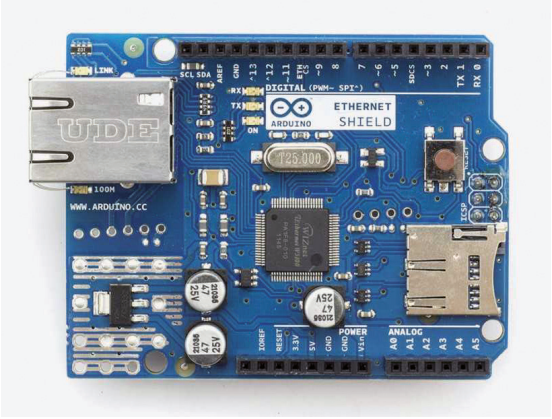


Figure 4.1: The top view of the Arduino Ethernet Shield.

On the shield, you can see identical sockets to those found on the Arduino Uno. These sockets directly connect to the pins on the Arduino Uno, making it stackable and allowing for communication through these pins. The Ethernet shield is necessary for our project, as it requires a TCP/IP network connection, specifically an Ethernet-based one. Although there are various types of TCP/IP networks, such as coaxial cable or fiber optic cable, we have chosen to use Ethernet on CAT5 cable due to its affordability, ease of use, and widespread availability. Additionally, the Ethernet Shield board features indicator lights that provide important information, so understanding their meaning is crucial.

The shield contains a number of informational LEDs:

PWR: indicates that the board and shield are powered.
LINK: indicates the presence of a network link and flashes when the shield transmits or receives data.
FULLD: indicates that the network connection is full duplex.
100M: indicates the presence of a 100 Mb/s network connection (as opposed to 10 Mb/s).
RX: flashes when the shield receives data.
TX: flashes when the shield sends data.
COLL: flashes when network collisions are detected.

So, if we are doing proper communication, we should see the TX and RX lights flashing a lot, and that is essentially the Ethernet Shield. Most of what you are supposed to understand are these lights. These indicator lights will give us feedback as to how the network is performing and how everything is working.

Chapter 5 • Network Connection Overview

In the previous section, we provided a list of the hardware components used in the setup, particularly focusing on the Arduino Uno and the Ethernet shield for the Arduino Uno. In this section, we aim to present a visual representation or block diagram of how these components are interconnected to achieve the goals outlined in this book. Additionally, we will showcase the physical setup of this configuration on our workbench in the following section. By the end of this current section, you will have a clear understanding of the interconnections through the block diagram and a visual representation of the physical setup.

To begin, the main development board used is the Arduino Uno. This is then connected to the Ethernet shield for the Arduino Uno. Additionally, a network hub is incorporated into the setup. In our illustration, a five-port network hub is displayed, but in experimental scenarios, a 16-port network hub can be used. You have the flexibility to use a four-port or any other suitable option for your needs.

For the setup, only two ports are necessary. One port will be used to connect the laptop to the Arduino Ethernet shield, while the other port will be utilized for connecting the laptop to the network hub. Essentially, we are creating a small and simplified local area network for our Modbus TCP communication. Refer to Figure 5.1 for a visual representation of the required setup.

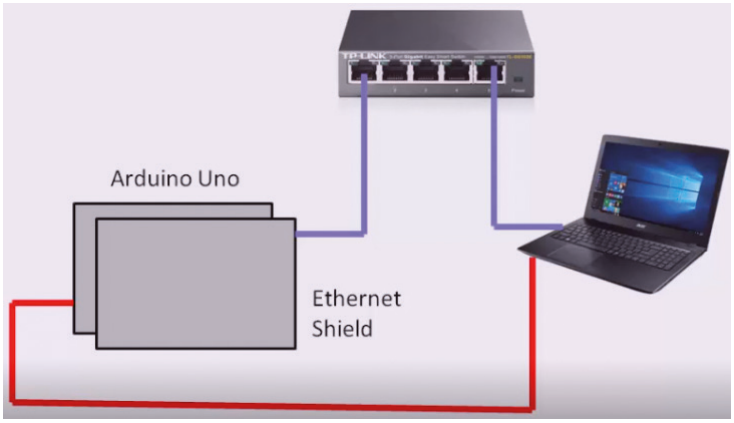


Figure 5.1: Network connection scheme.

To set up the hardware, you will need to purchase some standard network cables (CAT5) called straight-through network cables and crossover cables, which are inexpensive and available at computer stores. Additionally, you will need a programming cable for the Arduino Uno, indicated by a red line, which is essential for any tasks involving the Arduino. The programming cable serves two purposes: programming the Arduino and receiving text messages from the running application through a special serial monitor window. This feature is particularly useful since the Arduino Uno lacks a visual interface. When working with Modbus and dealing with values in the Arduino and Modbus registers, it is crucial to be

able to visualize these values. Therefore, the application will be designed to send the values to the laptop via this cable, allowing for monitoring changes in value, reading and writing registers, and more. This summarizes the hardware setup, and in the following section, we will demonstrate how it appears on our desktop.

Chapter 6 • Setting up the Hardware

We are providing a visual representation of the physical hardware setup on the desktop, as in the previous section we only showed a diagram. Now, we will look at the actual setup. In Figure 6.1, you can see an Arduino with an Ethernet shield on top. The Arduino is located at the bottom, and the Ethernet shield is connected to it. The pins on the shield physically connect it to the Arduino. There are also indicator lights on the Ethernet shield that display different statuses. The Ethernet port is located on the shield, and a blue cable connects it to the Ethernet hub. The Ethernet hub has ports where the blue cable is inserted, specifically the port connected to the Ethernet shield on the Arduino. Additionally, there is a gray network cable that extends to the laptop, connecting to the Ethernet port.

This provides network connectivity to both the laptop and the Arduino via the Ethernet shield. The USB cable serves as the programming cable for the Arduino and allows for receiving feedback in the form of text messages. It is connected to the Arduino's port in order to program it. This configuration aligns with what was described in the previous section and is depicted in Figure 6.1. Now that you have a clear understanding of the setup, we can proceed with exploring various programming examples and how they appear on the desktop workbench.

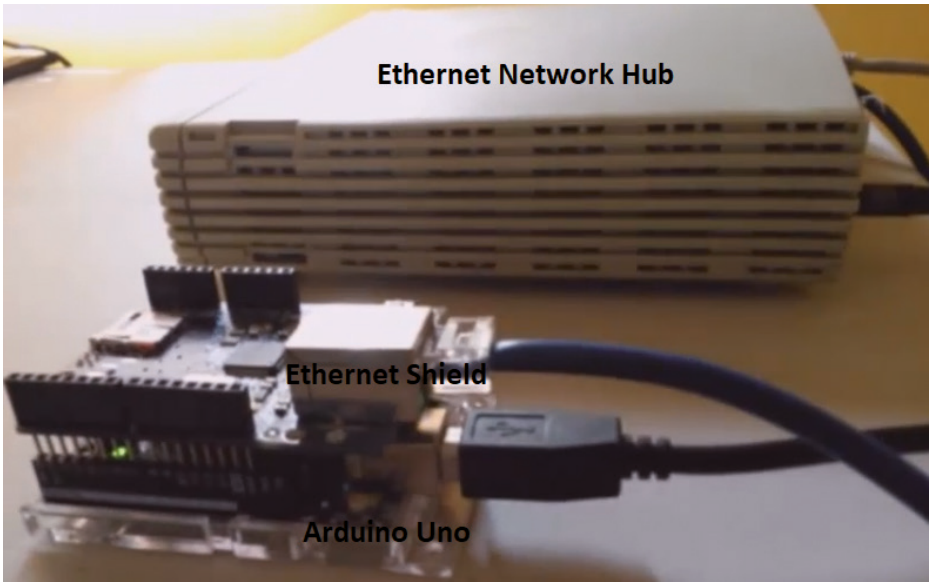


Figure 6.1: Ethernet network physical hardware setup.

Chapter 7 • Arduino Programming Software

In this segment, we will be discussing where to obtain the Arduino programming software, as well as the process of downloading and installing it. To get the software, we recommend visiting the official Arduino website at Arduino.CC. This website not only provides the software download, but also offers a plethora of resources for learning about Arduino basics, including information about both hardware and software. It is highly recommended that you explore this website to gain a better understanding of Arduino, especially if you are unfamiliar with it. However, in this particular section, our focus will be solely on the software.

To access the software, navigate to the "Software" section on the website. Although there is an online integrated development environment (IDE), we do not need that. Instead, we want to download the Arduino IDE. The IDE is an open-source software that enables easy coding and uploading to the Arduino board. By clicking on the Windows installer, you can save the current Arduino version to your device. If you wish, you can choose to contribute to support the open-source nature of Arduino. Otherwise, simply click on "Download" and save the file as an executable. After saving, double-click the executable to run a regular Windows installation process. We assume that you already have the knowledge of how to install Windows software. With these steps, you can obtain the Arduino software, which is completely free. We will demonstrate how to use the software in upcoming sections. For now, our focus is solely on downloading the necessary components.

Chapter 8 • Modbus Libraries

In this section, we will demonstrate the libraries we plan to use in order to implement our Modbus server and client on the Arduino. We will provide information on where to download them, and once we begin coding, we will also guide you on how to utilize them. If you are unfamiliar with Arduino libraries, they are pre-existing pieces of code that eliminate the need for starting from scratch. Various individuals online have developed libraries for a wide range of purposes, including the Modbus TCP server and client. If you have experience implementing Modbus RS485 on the Arduino, you may have used libraries for that as well. The objective of this book is to download these libraries, incorporate them into our project, and utilize function or command calls within them. We will explain the process in detail, but it is important to understand that these libraries are code modules designed to simplify the process of writing programs for Modbus TCP server and client. In this book, we will download separate libraries for the Modbus TCP server and the Modbus TCP client. Attached to this book are two resources. The first is a link to the Modbus TCP server library. The weblink is as follows:

https://github.com/andresarmento/modbus-arduino

Once you click on it, you will be directed to a web page. On this webpage, you will find information about a Modbus library developed by Andre Sarmento specifically for Modbus TCP server implementation. This is the designated page for the library. Our request is for you to bookmark this page as you will need to refer to it for the provided documentation. Andre Sarmento has put together comprehensive documentation on using the library, making it a valuable resource. Upon reaching this page, your next step is to click on "Clone or download" and select "Download ZIP" as demonstrated in Figure 8.1.

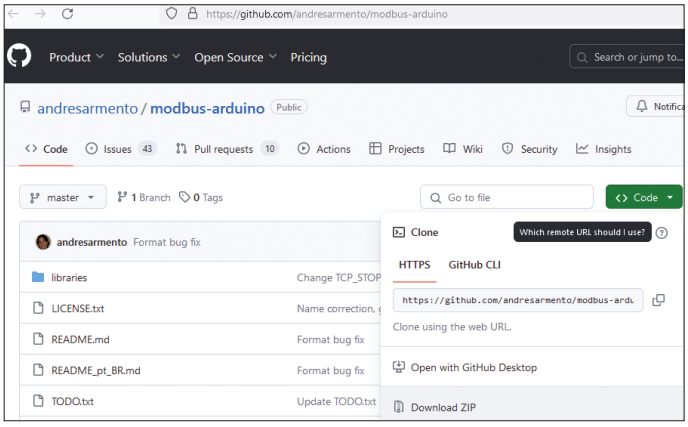


Figure 8.1: Clicking on "Clone or download" and "Download ZIP".

After clicking on the "Download Zip" button, a prompt will appear asking you to save the file. You should choose a folder to save it in. It is recommended to create a separate folder for your libraries, such as Arduino apps and Arduino libs (Arduino libraries). The Arduino app folder should be used to store all the applications you write, while the Arduino libs

folder is where the libraries will be stored. In this case, we downloaded a file called Modbus TCP server, which is mistakenly named Modbus Arduino master. To avoid confusion, we will rename the zip file to modbus_server_tcp.zip. Once you have the zip file, you should extract it by selecting "Extract All", which will create a folder with several subfolders containing libraries. You will see these folders as shown in Figure 8.2, but you do not need to use all of them, just some. Once you have downloaded and extracted the file, you are ready to learn how to use it later.

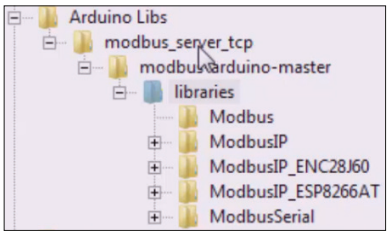


Figure 8.2: The modbus_server_tcp.zip extracted folders.

The subsequent resource in the attached section consists of downloadable files, specifically a file called MgsModbus.zip. It is not a hyperlink, rather a file that can be downloaded. The purpose of this file is to provide our library for implementing a Modbus TCP client. We chose not to direct you to a webpage because the file is directly accessible from myarduinoprojects.com. The corresponding weblink is as follows:

https://myarduinoprojects.com/modbus.html

The library featured on this page is not functioning properly due to the presence of several bugs. To address this issue, we have made necessary corrections to the library. The corrected version, which is guaranteed to work, can be downloaded from the appendix section of this book. Upon downloading, you will receive a file named MgsModbus.zip. Extracting this file will yield a folder containing two files, namely MgsModbus.cpp and MgsModbus.h, as shown in Figure 8.3.

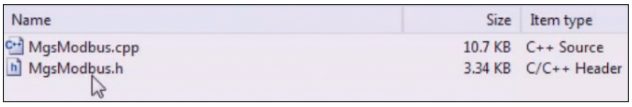


Figure 8.3: The files inside the MgsModbus.zip.

These are the library files needed for the Modbus TCP client implementation. After downloading and extracting these two files, store them in a location such as the Arduino Libs folder. It is important to keep them in one folder for easy accessibility. This completes the process of obtaining the Modbus libraries. In the coding phase, we will demonstrate how to properly use these libraries, as it involves steps like copying the files and including them.

Chapter 9 • Modscan32 and Modsim32

The modscan32 and modsim32 download links are in the following weblink:

https://www.win-tech.com/html/demos.htm

To complete the process, you will need to click on each of them individually. This will lead you to screens that will prompt you to download two files. The first file is modsim32.zip, which you will need to click on to start the download. After downloading it, proceed to download the second file, modscan32.zip. Once both files are downloaded, you can continue with installation. There is no need for installation, as the program runs without it.

Now, we have both modsim32.zip and modscan32.zip files. After extracting them, we obtained two directories: modsim32 and modscan32. To access modscan32, simply double-click on it, and you will see the file named MODSCAN3.exe, as shown in Figure 9.1.

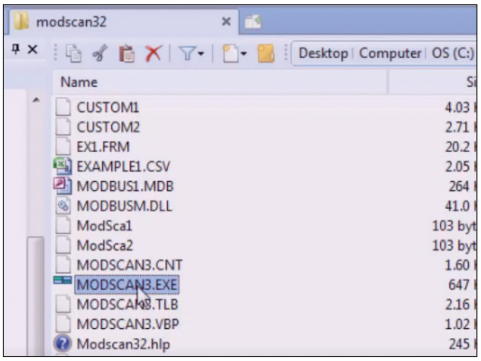


Figure 9.1: MODSCAN3.exe in the modscan32 folder.

This is the executable that you want to run, and you should create a shortcut to your desktop, which is a shortcut to this executable file. So, when we double-click on that, it just runs as depicted in Figure 9.2. We do not have to do any installation process and we are going to use modscan32 as a Modbus TCP client when we configure the Arduino as a Modbus TCP server. So modscan32 is going to be used to communicate with it and will be reading data from the Modbus TCP server (Arduino).

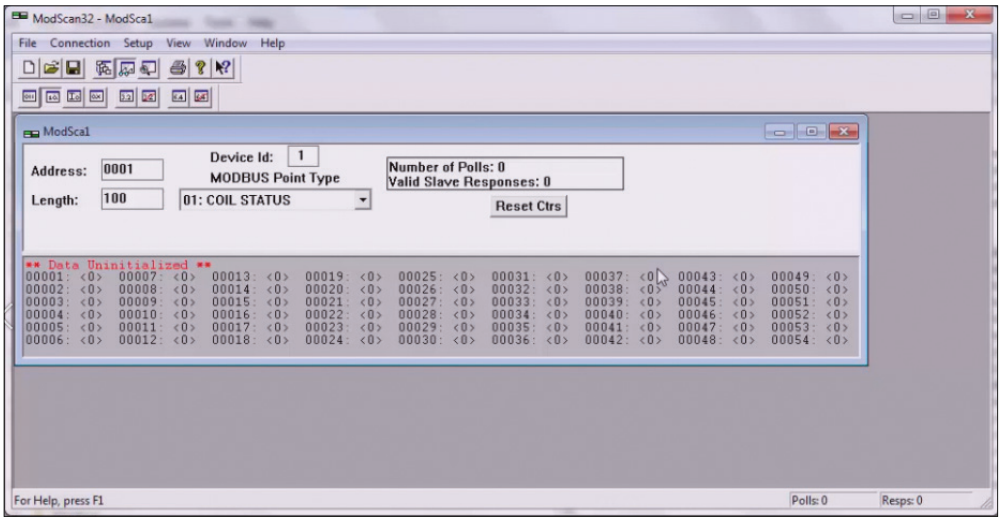


Figure 9.2: Running the MODSCAN3.exe file.

In the modsim32 folder, there is an executable called modsim32.exe. You can create a shortcut for it. When you double-click on the shortcut, a modsim32 pop-up window, as shown in Figure 9.3, will appear.

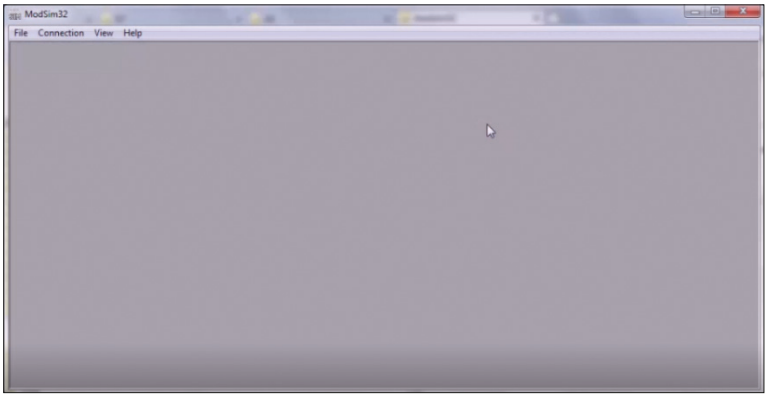


Figure 9.3: The modsim32 pop up window.

We will use modsim32 as a Modbus TCP server when configuring the Arduino as a Modbus TCP client. This means that the Modbus TCP client will read data from modsim32. Throughout the rest of this book, we will be using two Windows-based Modbus applications called modscan32 and modsim32. As we progress through the sections and develop various applications, you will learn how to use these applications.

Chapter 10 • The Programming Software Arduino IDE

Now that you understand what the hardware looks like, and you may have already downloaded some software, it is important to note that you may not have the hardware in front of you at the moment. However, once you see how amazing and affordable the hardware is, we are confident that you will acquire it and revisit the book to try it out yourself. In this section, our aim is to introduce you to the Arduino programming software, which is known as the Arduino IDE, or Integrated Development Environment. But before we proceed with that, it is worth mentioning that when the Arduino software was installed, it would have also installed a device driver. This driver, when connecting the Arduino to your laptop or PC via a USB cable, creates a virtual COM port. We encourage you to look at this virtual COM port now by accessing the device manager and navigating to the ports section. As shown in Figure 10.1, you will find it there.

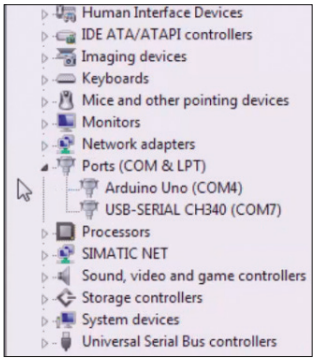


Figure 10.1: Ports in the "Device manager".

Before you begin programming the Arduino, make sure to check your port settings. The port should be set to Arduino Uno COM4, although it may be different for your specific device. Additionally, there is another virtual COM port called COM7, which is our USB to RS485 converter.

Moving on, the Arduino icon can be seen in Figure 10.2, and this is what you should store once you have installed and started it. In Figure 10.3, you can see the Arduino development environment. This is where you will write your code, compile it, and download it into the Arduino to make it work. While there are many features available, we will only focus on the ones that are most important and commonly used in the book's projects. The code area, as depicted in Figure 10.3, is where you will write your code.

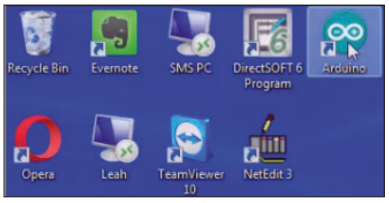


Figure 10.2: The Arduino icon.

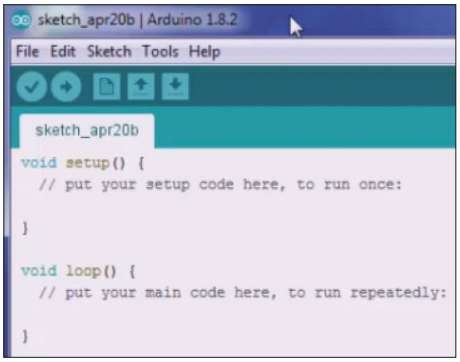


Figure 10.3: The Arduino IDE code area.

We will briefly discuss that, however, the primary tools you use in this setting are the buttons located at the top. We will go through each one individually. The button labeled "Verify" is the same as the compile function shown in Figure 10.4.

Figure 10.4: The "Verify" button.

After writing your code, when you are prepared to compile it and check if it is functioning properly, you will click on the upload button as shown in Figure 10.5. However, despite being labeled as "Upload," this button transfers your code and program to the Arduino board. This difference in terminology is similar to a US-Europe distinction. In the US, you would typically say "Download" to the Arduino, while in Europe, the term used is "Upload." Nonetheless, the objective remains the same: to transfer the program successfully into the Arduino board.

Figure 10.5: The "Upload" button.

By using the button illustrated in Figure 10.6, you can generate a fresh window. When you select the "New" option in the Arduino interface, a completely new window opens instead of opening an additional tab.

Figure 10.6: The "New" button.

If you wish to open an existing project, you can use the Open button depicted in Figure 10.7. Furthermore, to save your project or code, use the button shown in Figure 10.8.

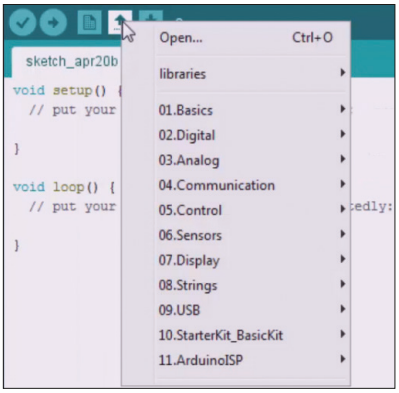


Figure 10.7: The "Open" button.

Figure 10.8: The "Save" button.

These buttons are the most often used by us. We are now asking that you perform an action: navigate to file preferences and select the option to display line numbers. We believe having line numbers is beneficial, and for that reason, we have demonstrated this in Figure 10.9 and Figure 10.10.

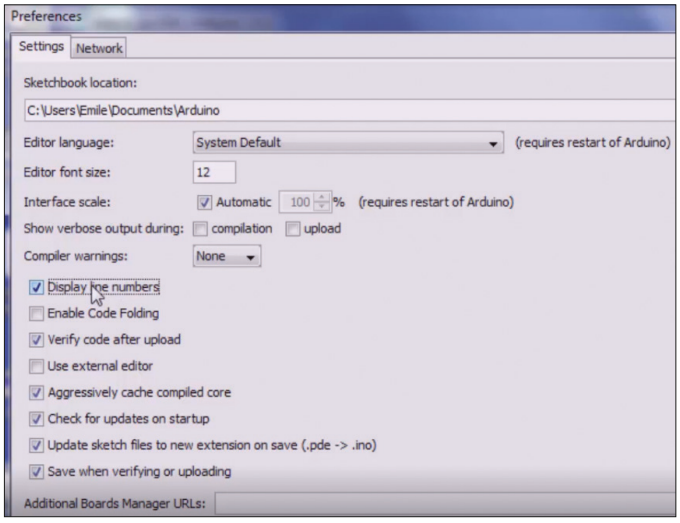


Figure 10.9: Clicking on "Display line numbers".

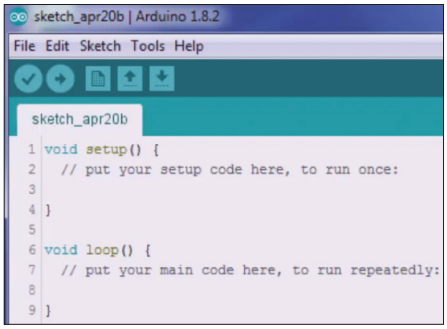


Figure 10.10: Displayed line numbers.

As shown in Figure 10.10, the top menus on this interface, such as "Verify," "Compile," and "Upload," perform similar functions. Many of these menu items have not been used yet. Now let us look at the user window, which is the code area. When starting a new project, some code is automatically provided. This includes two functions that are present in every Arduino program: setup and loop. The comment in the code area states that the setup function is for initialization, and the loop function is for the main code that will be repeatedly executed, similar to how a PLC operates. Embedded systems typically work by continuously repeating the code placed in the loop function. On the other hand, Windows operates on an event-based system, where applications run differently.

When you are ready, you can click on the "Verify" button, even if there is no code present. This will initiate the sketch compilation process, and it will show a "Done compiling" message. Currently, we are connected to our Arduino device. To confirm this, go to the "Tools" menu, click on "Port," and select the appropriate communication port. This port selection will remain active once you have made your choice, as shown in Figure 10.11. When initially opening the Arduino development environment, you will need to go through this process of selecting the port.

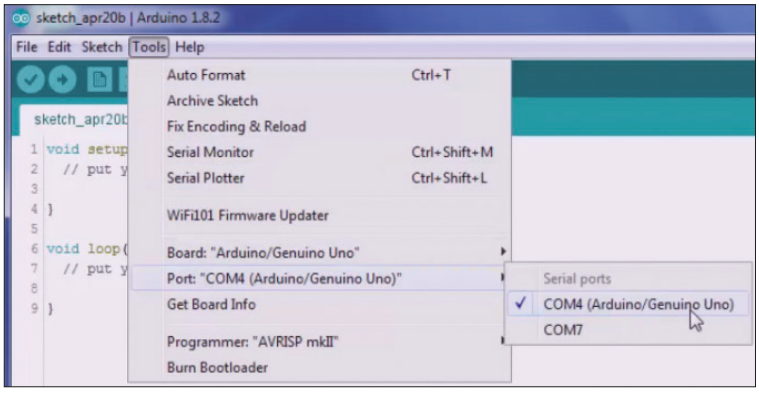


Figure 10.11: Choosing your communication port.

We have gathered our program here and will now try to initiate the download process. After recompiling, the program states "Uploading" and then "Done uploading." This indicates the completion of the upload and signals that the Arduino will begin executing the new program. The Arduino IDE is relatively straightforward compared to software like Microsoft Visual Studio, which has an abundance of buttons and menu options. This concludes a brief introduction to the Arduino IDE. In the later section, we will proceed with our first application, downloading it and observing its functionality.

Chapter 11 • Testing Serial Communication

We will start by writing code for this task. The reason for doing so is because, as mentioned earlier, we will use serial communications on the Arduino as our means of interfacing with it. This will allow us to gain insight into the Arduino's operations. Furthermore, we will configure it to provide continuous feedback through Modbus communication.

To begin, we need to save this file by selecting "Save As". We will be storing everything in the Arduino apps folder. It is recommended that you choose a folder of your preference to store all the applications, and for this specific task, we will name the file "Serial01" as depicted in Figure 11.1.

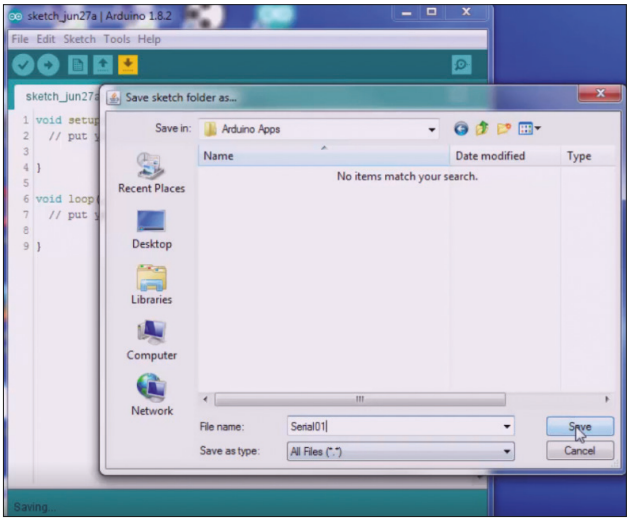


Figure 11.1: Saving the file as "Serial01".

Therefore, we will remove the comments as we no longer need them. What the code will do is send the message "Hello World!" to the serial monitor within the Arduino software. Let us show how this works. Firstly, we need to include the SPI.h library, which is a pre-existing library that manages serial communication. Then, we will place all the code within the setup function so that when the Arduino boots up, it will automatically send the "Hello World!" message to the serial monitor. The **serial.begin(9600)** command sets the baud rate to 9600 bits per second, as it is necessary for serial communication. Finally, the **serial.println** command handles printing the message as a line. This is our first program and is depicted as a simple program in Figure 11.2.

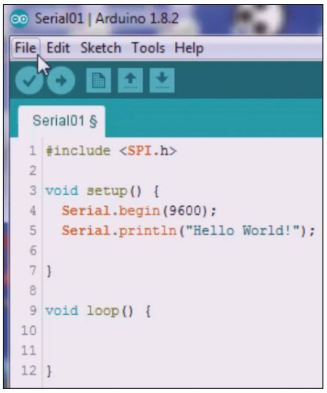


Figure 11.2: Our first program.

Firstly, the code initializes the serial port and then prints "Hello World!" but it does not display it. Now, let us demonstrate this process. We are going to save the code, verify it, and compile the sketch. The initial run may take some time. Fortunately, there are no errors. Now, we will navigate to the tools menu and open the serial monitor, which is shown in Figure 11.3. This serial monitor window within the Arduino application displays anything that is sent using certain commands.

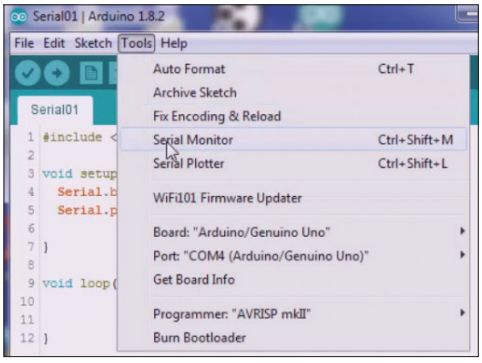


Figure 11.3: Choosing "Serial Monitor".

Therefore, the information in this text is transmitted through serial communications and displayed in this window using the programming cable. It is important to note that this is unrelated to Modbus RS485 or TCP. We are solely utilizing Arduino serial communications to receive feedback. To send our application to the Arduino, we will click the upload button. As you can see, the application has started and printed "Hello World!", indicating that it is functioning correctly. However, it does not perform any additional actions. This is the desired outcome we would like you to observe. In Figure 11.4, you can see an example of how we can print and receive feedback from the Arduino while it is executing its tasks.

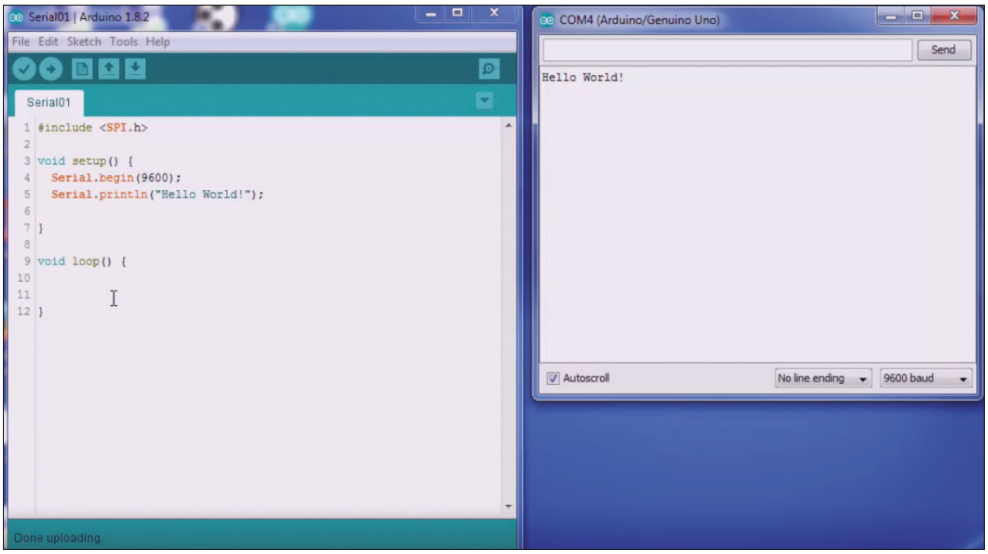


Figure 11.4: Printing and getting feedback from the Arduino.

Chapter 12 • Displaying the Value of a Variable using Serial Communication

In this section, we will further enhance our application. Our goal is to improve the previous program by implementing a Modbus TCP server on Arduino. This server will have memory locations such as holding registers, input registers, coils, and input statuses, just like any other Modbus device. To achieve this, we need to create and display values in these memory locations using variables. In this application, we will demonstrate how to print the value of a variable in the serial monitor window. To begin, we will modify the application and save it as "Serial02", as illustrated in Figure 12.1.

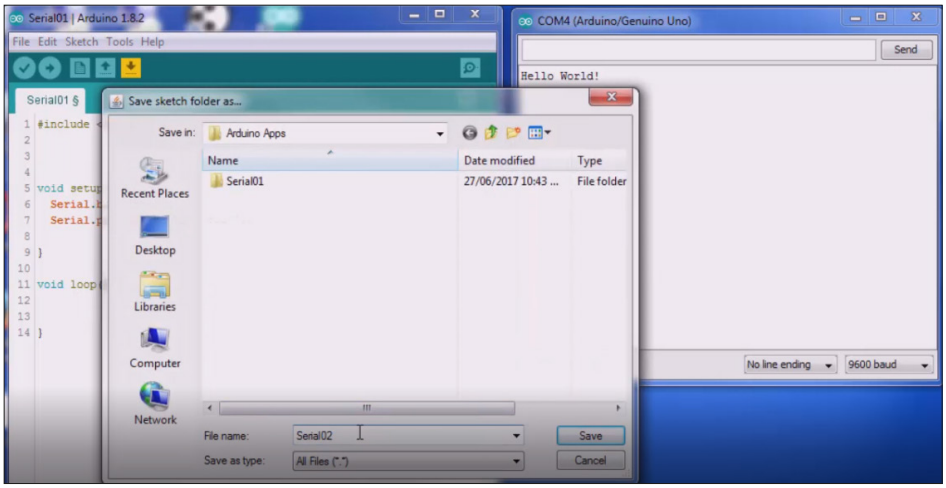


Figure 12.1: Saving the file as "Serial02".

We will refer to the tank level as an integer value. Thus, in this hypothetical scenario, we have connected an Arduino to a tank level sensor that measures the water level in a water tank. We have declared a variable named "tempstr" as a string data type to represent this tank level. Essentially, we are defining a string variable called "tempstr" which will be utilized for a specific purpose. Moving ahead, we may see the command **serial.begin(9600)** and for now, we will assign a fixed value of 12 to the tank level variable. Typically, if an Arduino were connected to a sensor, it would receive readings and assign them to the tank level variable. However, since we do not have a sensor, we are hardcoding the value. As for the "tempstr" variable, we aim to make it more easily understandable to humans by combining the tank level with the word "feet" using the statement "**tank level plus string(tanklevel) plus ft**." Finally, we will replace the phrase "hello world" with the variable "tempstr" as shown in Figure 12.2.

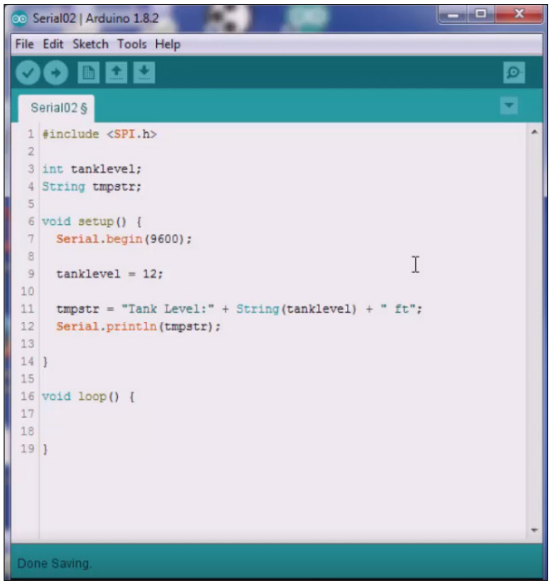


Figure 12.2: The "Serial02" code.

Let us confirm that first. As a result, it was successfully compiled. Now, we will proceed to send it to the Arduino and run it by clicking on the upload button. Then, we can see the output in the serial window. The expected output should be "Tank Level: 12 ft" as the value of the variable is being printed. Let us proceed to send it to the Arduino. Additionally, it is important to mention that the window clears every time you perform an upload, and there you have it — "Tank Level: 12 ft" as shown in Figure 12.3.

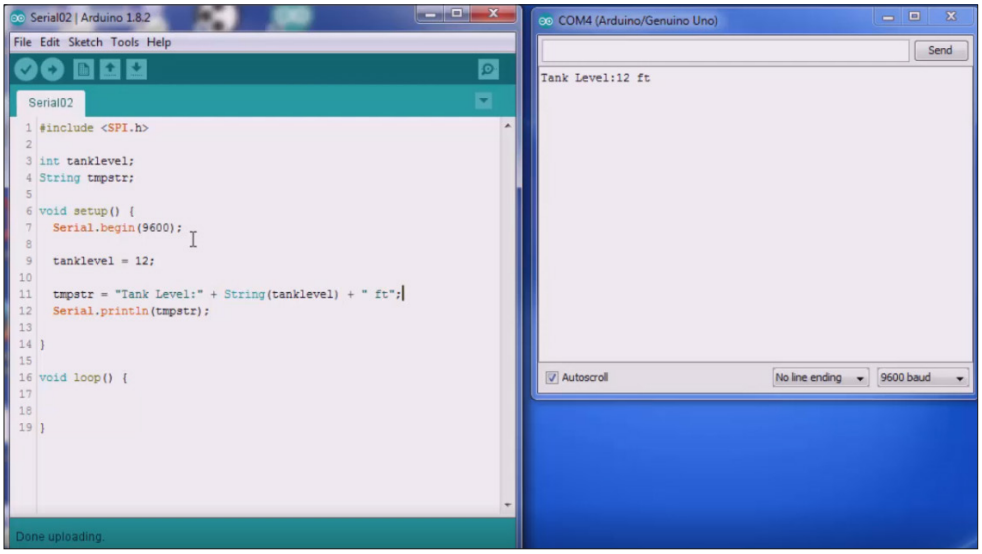


Figure 12.3: Running the "Serial02" file.

Instead of using hard coded text, the variable value was printed. The printed output includes both the hard coded text "Tank Level:" and "ft", as well as the value stored in the variable "tanklevel". This allows us to check if values have been successfully changed by using modscan32 on a holding register. We can verify any changes by viewing the serial window output through the **serial.println** function. This represents the second level of our application.

Chapter 13 • Additional Code to Support the TCP Server Operation

All right, so we will be making changes to this application once again. We will click on the "File" option, then select "Save as" and name it "Serial03" as shown in Figure 13.1. Now, let us explain the modifications we would like to make to this code. Our objective is to insert code within this loop that can identify any alterations in the variables we will be utilizing. Once it detects these changes, it will display the updated values of those variables on the serial monitor.

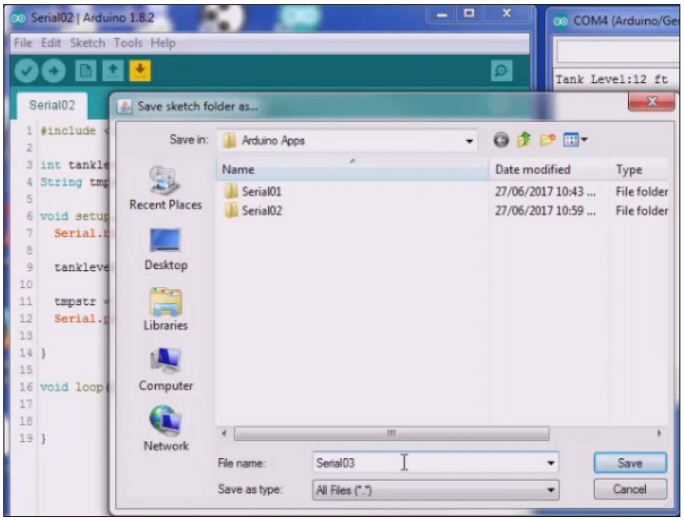


Figure 13.1: Saving the file as "Serial03".

Our goal is to bring the loop into the equation. The reason behind this decision is that our Modbus TCP server operates in real-world scenarios where values frequently change. It is crucial for us to be informed when these values do change. Furthermore, we plan to utilize modscan32 to alter values and require confirmation that these changes have occurred successfully. To achieve this, we will begin by removing certain code from within the setup and migrating it into a separate function that can be called from the loop. This function, named "**void DisplayCurrentValues()**", will house the relevant code from the setup section. Subsequently, we will reintegrate this function back into the setup, calling it "**DisplayCurrentValues**" as shown in Figure 13.2. Essentially, we have created a new function to extract the code from the setup, allowing us to easily print all the values through the serial monitor by calling the "**DisplayCurrentValues()**" function.

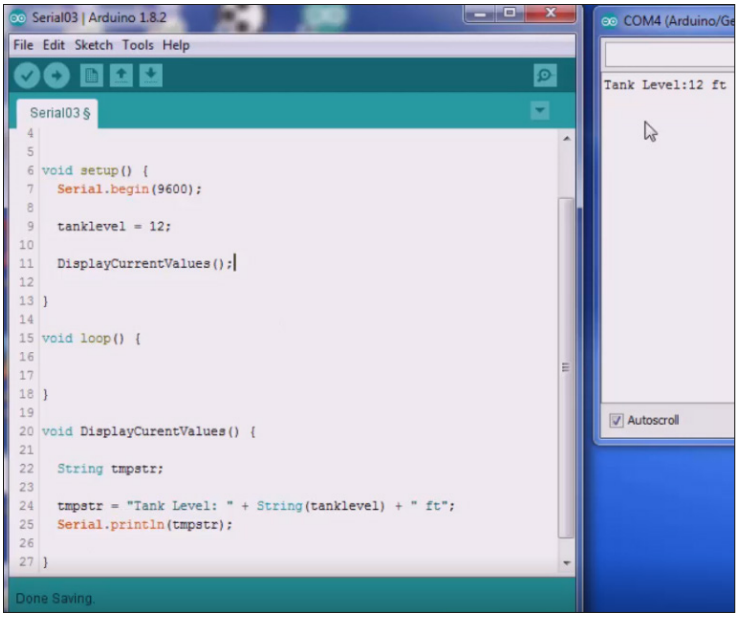


Figure 13.2: Defining the function **DisplayCurrentValues()**.

So, how can we determine if the time level has changed? Well, we need to introduce a new variable. We will refer to it as "tanklevel_old," which will hold the previous value of the tank level. Whenever the tank level value is updated, we will compare it with the previous value. If they are different, we will proceed with the update. Thus, we have the "tanklevel_old" variable here. To start off, we will initialize both variables and set the value of "tanklevel_old" to 12, as shown in Figure 13.3.

Moving forward, let's create a function that will be invoked within the loop. We will name it "Send updated values" as it will send the newly updated values to the monitor.

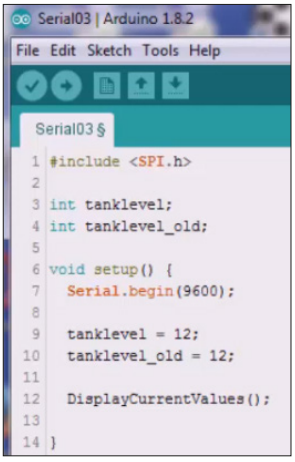


Figure 13.3: Defining the tanklevel_old variable.

We are going to call this **CheckForDatachange()** and define that function. Then, we are going to declare a Boolean variable data_has_changed. Initialize it to false. Remember the purpose of this? The purpose of this function is to check to see if any of the data has changed. So, we are going to see if the tanklevel_old variable is not equal to the tank_level variable. That means a change has taken place. We want to set the Boolean data_has_changed to be equal to true and update the tanklevel_old variable with the same value that is currently in the tank_level variable. Then we are going to write if Boolean data_has_changed equal to true, then call the function **DisplayCurrentValues()**. So, this loop is constantly going to call function **CheckDataForChange()**. This function what compares the tanklevel variable with the tanklevel_old variable and if tanklevel is updated, let us say for some reason by outside sensor in a real world or for some other method, it will compare it with tanklevel_old. So, both of them start off at 12. Suppose the tanklevel goes to 15. Is 12 equal to 15? No, then the data_has_changed is true. Then take 15 and put it in the old data (tanklevel_old), and call the function **DisplayCurrentValues()**. So, it will only execute the function **DisplayCurrentValues()** when there is a change as shown in Figure 13.4.

```
16 void loop() {
17
18   // send updated values
19   CheckForDataChange();
20
21 }
22
23 void CheckForDataChange() {
24
25   boolean data_has_changed = false;
26
27   if (tanklevel_old != tanklevel) {
28     data_has_changed = true;
29     tanklevel_old = tanklevel;
30   }
31
32   if (data_has_changed == true) {
33     DisplayCurrentValues();
34   }
35
36 }
```

Figure 13.4: Defining the function **CheckDataForChange()**.

This is our updated application, which we believe will greatly assist us. The inclusion of support code will facilitate receiving feedback from the Arduino, as we can observe through the serial monitor. Once we validate the code, we will be ready to proceed. Now, we will proceed to send this application to the Arduino, where we should obtain the same result, indicated by a tank level of 12 ft. If there are any changes in the tank level, it will display additional lines with the updated data, as shown in Figure 13.5.

Now, we have completed the development of our support code that will facilitate Modbus functionality. In the following section, we will integrate this code into our application and create a new application to implement the Modbus TCP server.

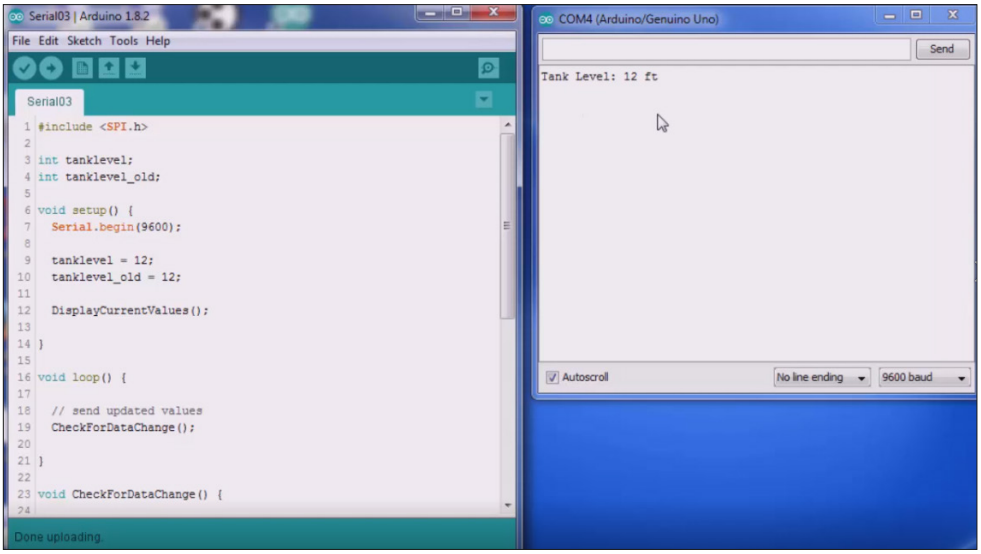


Figure 13.5: Running the "Serial03" file.

Chapter 14 • Adding Code to Implement the Modbus TCP Server

In this section, we will be adding the Modbus TCP server code to our application. To begin, we need to save the current file as a new one called "Serial04", as depicted in Figure 14.1.

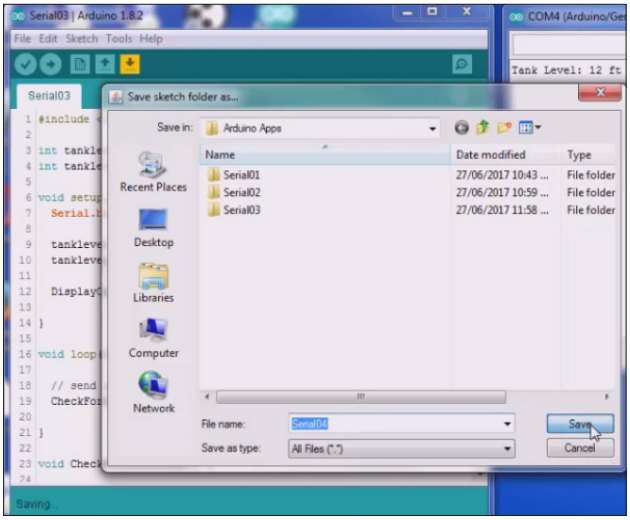


Figure 14.1: Saving the file as "Serial04".

In the preceding sections of this book, we instructed you to download certain libraries. Now, we will utilize one of those libraries. Our attention will be directed towards the modbus_server_tcp folder. To access it, click on this folder. Upon doing so, you will observe that there are Modbus and ModbusIP subfolders within the libraries section. Although there are additional folders, our focus should solely be on the Modbus and ModbusIP subfolders, which are indicated in Figure 14.2.

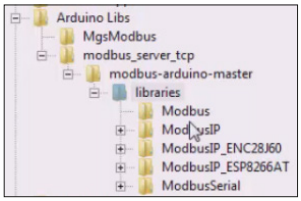


Figure 14.2: Modbus and ModbusIP folders.

So, within Modbus, there are Modbus.h and Modbus.cpp files and within ModbusIP, there are ModbusIP.h and ModbusIP.cpp files as illustrated in Figure 14.3 and Figure 14.4 respectively.

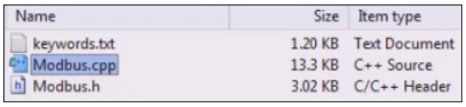


Figure 14.3: Contents of the Modbus folder.

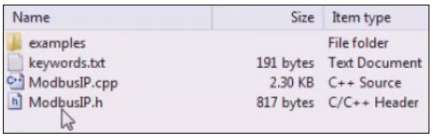


Figure 14.4: Contents of the ModbusIP folder.

We have an interest in those specific files, and here are the instructions on what you need to do with them. This particular step can be a bit challenging. Locate the "Libraries" folder in Explorer, which can be found under the desktop. Within the "Libraries" folder, you will see the "Documents" folder. Within the "Documents" folder, there are both the libraries folder and another folder. These are the locations where you need to copy the files. We have already completed this task, so now it is your turn to do it. However, we would like to highlight that you need to create a new directory or folder under the libraries folder. This directory should be named "Modbus," and there should also be another one named "ModbusIP," as shown in Figure 14.5.

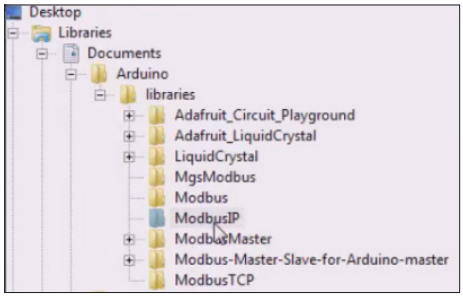


Figure 14.5: Creating Modbus and ModbusIP folders.

We will return to the libraries folder before continuing. For the Modbus library, please copy the Modbus.h and Modbus.cpp files and paste them into this location. Similarly, for the ModbusIP folder, copy the ModbusIP.h and ModbusIP.cpp files and paste them into the ModbusIP directory. Therefore, both folders, along with their respective files, need to be placed under the libraries folder. Once that is done, go ahead to the next step. Go to "Sketch", select "Include Library", and scroll down to find the Modbus folder. Click on it to include the Modbus library. Repeat this process by going to "Sketch", "Include Library" and including the ModbusIP library as shown in Figure 14.6. Now, we have successfully included both libraries that were part of the download package. Additionally, we need to include a built-in library called Ethernet.h. Therefore, we have included the ModbusIP.h, Modbus.h, Ethernet.h, and SPI.h libraries in our project, as depicted in Figure 14.7. Moving on to step two, for this example, our goal is to create a Modbus TCP server with only one register, serving as an input register. We will name this register tanklevel_ir.

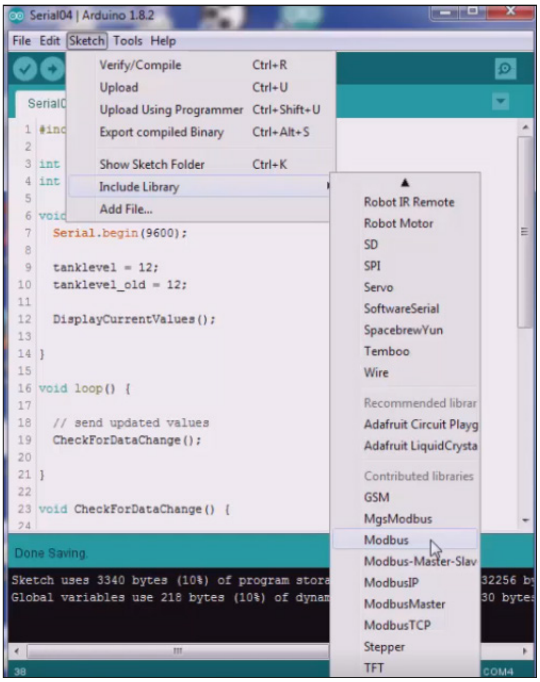


Figure 14.6: Including the Modbus and ModbusIP libraries.

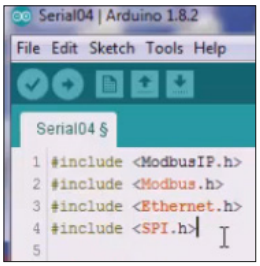


Figure 14.7: All the libraries included in our project.

So, our task is to create a designated area in the memory and name it as a register. To do this, we define a constant integer called tanklevel_ir and set it equal to 100. Although tanklevel_ir is a variable, it represents the address of the register. It should be noted that the value 100 is not arbitrary, but rather the starting point within the input register range of 30,001 to 40,000. Therefore, the actual register address is 30,100. This location within the Arduino is now our register address. In addition, we declare a Modbus IP object, for which we will refer to it as ModbusIP mb, as depicted in Figure 14.8.

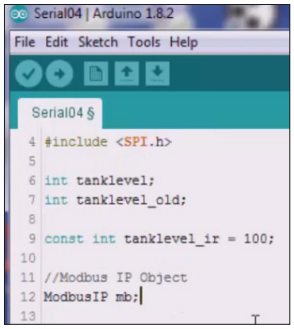


Figure 14.8: Defining tank level input register and Modbus IP object.

Next, our next step is to insert certain codes obtained from the provided examples within this library. Let us guide you to the location of these examples. They can be found in the ModbusIP folder, specifically in the "examples" folder. We obtained them from the "Lamp" folder, as shown in Figure 14.9. Additionally, we gathered a significant amount of information from the library's webpage. The documentation available on this webpage thoroughly guides you through the entire process. The weblink is as follows:

https://github.com/andresarmento/modbus-arduino

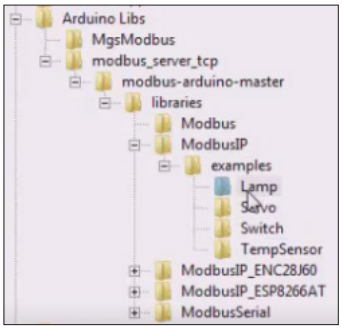


Figure 14.9: Directory to the Lamp folder.

Now let's return to our code and demonstrate its functionality. Remember, we are setting up the Arduino as a node on a network. In this context, the "mac" refers to the Ethernet address or MAC address. We have arbitrarily chosen 192.168.1.120 as our IP address. It is essential to assign an IP address to the Arduino because, on a Modbus TCP network, the IP address serves as the Modbus ID and must be unique. We use the command **mb.config(Mac, IP)** to execute this configuration. Moving forward, we need to inform the Arduino that "tanklevel_ir" should be recognized as an input register. To achieve this, we use the "mb" Modbus object and employ the "**add**" command to add and read the register. We assign it the address of tanklevel_ir, which in this case is 100. Once the register is set up, we must assign it a value corresponding to the tanklevel. To do this, we use the **mb.Ireg(tanklevel_ir, tanklevel)** command. This action takes the tanklevel value and stores it in the specified register location, tanklevel_ir, as demonstrated in Figure 14.10.

How did we become aware of all these functions? Simply go through the documentation, and you will discover all of them listed. However, for now, keep in mind that our objective is to get you started as fast as possible. Therefore, we will not go over each function in detail because it could be too tedious. Our focus remains on facilitating your quick start.

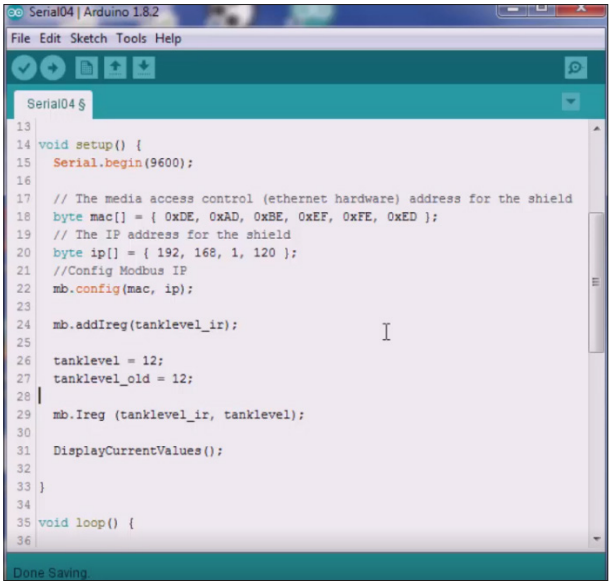


Figure 14.10: Completed setup code section.

In summary, we have completed the necessary steps such as adding our libraries, defining an input register, assigning a value to it, and handling our network requirements. The final step is to execute the **mb.task()** command depicted in Figure 14.11.

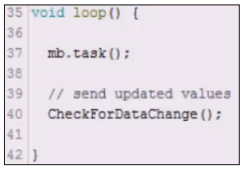


Figure 14.11: Adding the ***mb.task()*** *command.*

That's it. We have successfully created a Modbus TCP server. Now we can proceed to check for any errors by clicking on the verify button, and once it is verified, we can send the application to our Arduino by clicking on the upload button. After that, we should be able to open modscan32.exe, switch it to Modbus TCP client mode, and retrieve the value of the tanklevel input register, which is currently 12. Since our Modbus TCP server is capable of being read by a Modbus TCP client, we will be able to obtain a value. This is the next task we will be undertaking in the following section.

Chapter 15 • Last Change Before Running the Program

We are now prepared to test our latest application, Serial04. Before we proceed, let's review the quick start example by referring back to the block diagram shown in Figure 5.1 from a previous section. This diagram illustrates our setup, where the Arduino and its shield are configured as a Modbus TCP server with a single input register representing the tank level. This setup is connected to the hub.

Next, we will use modscan32.exe to configure it as a Modbus TCP client. With this configuration, we will read the single register (addressed as 100) that represents the tank level. However, in the Modbus TCP protocol, the address will be 30,100. The process involves the modscan32.exe sending Modbus queries as the TCP client and the Arduino responding with Modbus responses. This exchange will continue as modscan32.exe continuously requests information from the Arduino. This dynamic of a client pulling information from a server is typical in the world of Modbus TCP.

Currently, we have assigned the IP address 192.168.1.120 to the Arduino. For our laptop, which will be connected to the Ethernet interface, we need to assign a different IP address. In this case, we will configure it as 192.168.1.121, as shown in Figure 15.1.

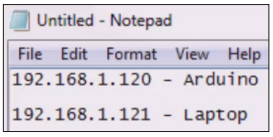


Figure 15.1: IP addresses for the Arduino and Laptop.

All right, and modscan32.exe is going to use that IP address when it connects. Thus, let us show you how to do that. We will search for a network and click on the Network and Sharing Center, as shown in Figure 15.2.

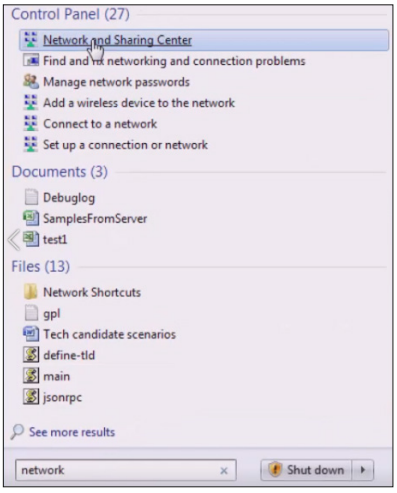


Figure 15.2: Clicking on the Network and Sharing Center.

We will click on "Change adapter settings", as depicted in Figure 15.3. Then go to "Local Area Connection" because this is our physical Ethernet connection on the laptop. Double click it as illustrated in Figure 15.4. We are going to click on "Properties", as shown in Figure 15.5, scroll down there to Internet Protocol Version 4 (TCP/IPv4), as in Figure 15.6.

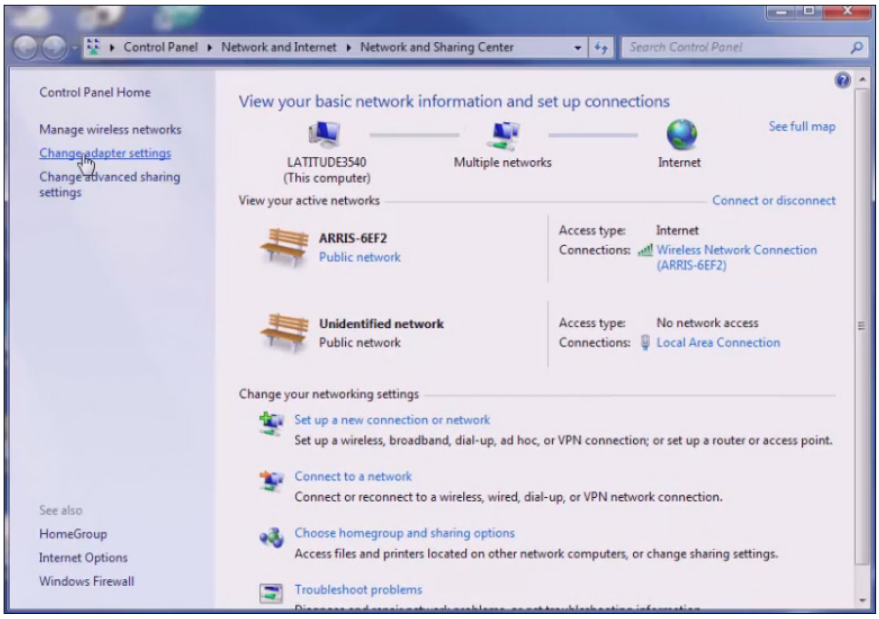


Figure 15.3: Clicking on "Change adapter settings".

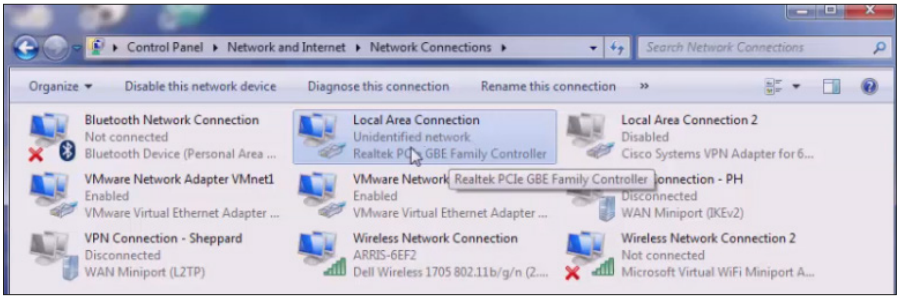


Figure 15.4: Clicking on "Local Area Connection".

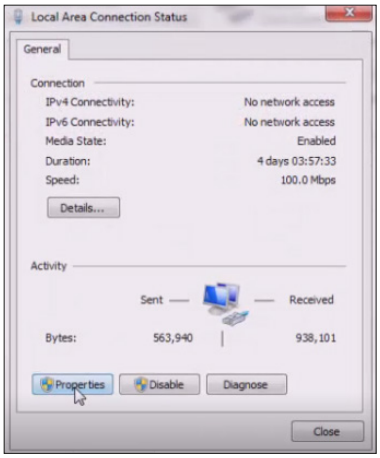


Figure 15.5: Clicking on the "Properties".

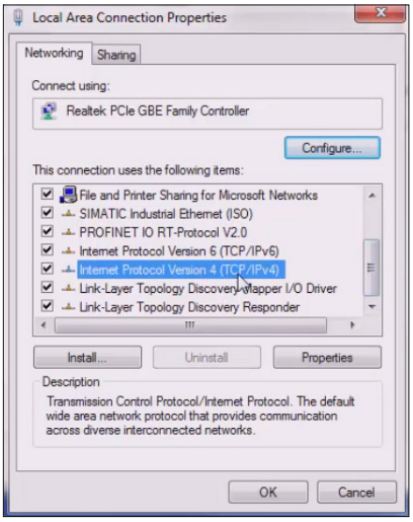


Figure 15.6: Choosing the Internet Protocol Version 4 (TCP/IPv4).

This is where we have already set it, but we just wanted to show you, we clicked on "Use the following IP address", and I hard coded my IP address, 192.168.1.121, subnet mask by default, and its default gateway is 192.168.1.1, as illustrated in Figure 15.7.

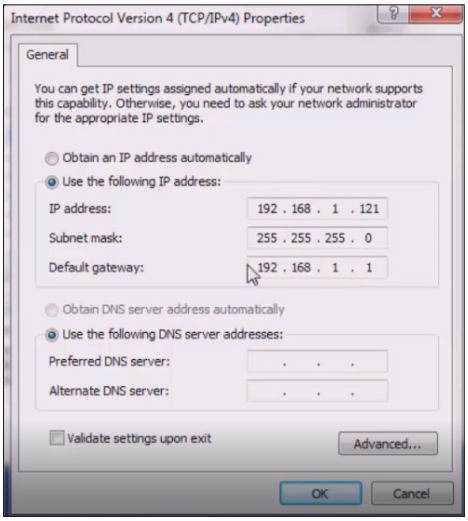


Figure 15.7: Setting the Internet Protocol Version 4 (TCP/IPv4) Properties.

All right, so the gateway really does not matter much in this instance, but the IP address is very important. So, we are giving the Ethernet interface as where our cable is plugged into our laptop which is 192.168.1.121. Remember, every element on a Modbus TCP network is uniquely identified by its IP address, so the IP addresses must be unique. So, my laptop is configured network-wise, and now we are going to send our application into our Arduino by using the verify and upload buttons. All right, so it is compiling and uploading, and done. Therefore, we get our "Tank Level: 12 ft" message as shown in Figure 15.8.

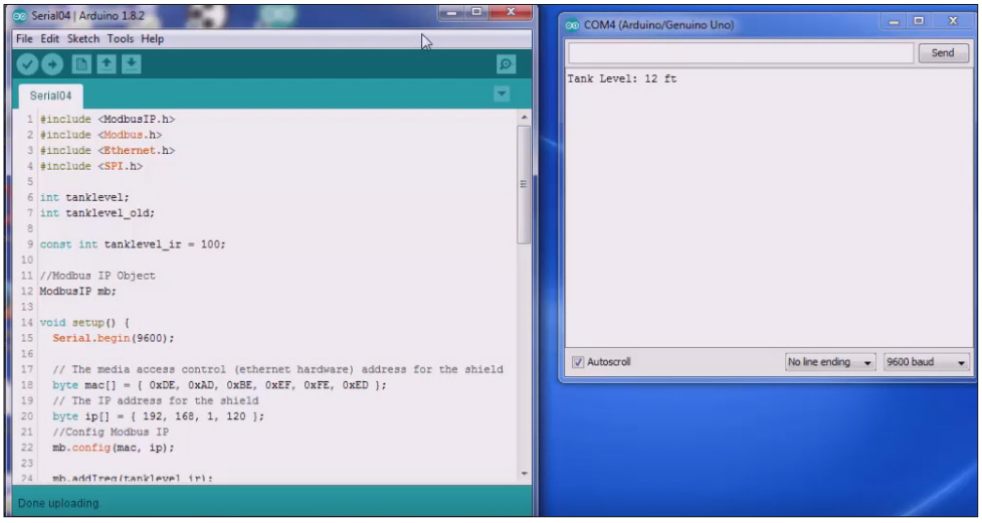


Figure 15.8: Getting the "Tank Level: 12 ft" message.

We are going to minimize the program, then we are going to start modscan32.exe. The device ID does not matter for the IP address. We are going to click input register, leaving

this as address 101. Now, this modscan.exe uses this offset. If you have enough Modbus knowledge, you know that the offset used is one. When you want to read 100, you put 101. So, 30,101 is actually 30,100 in the Arduino. It is just a Modbus offset thing, which is quite irritating when dealing with Modbus in the field. Next go to the "Connection2 tab and select the "Connect" icon as depicted in Figure 15.9.

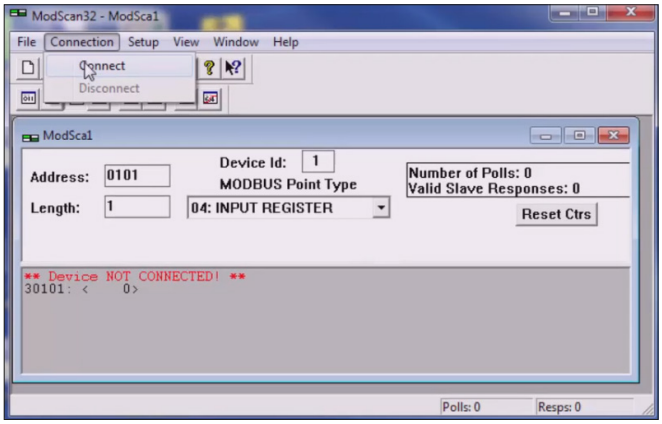


Figure 15.9: Selecting the "Connect" icon.

Usually, you pick something like a COM port here, but you are going to choose Remote TCP/IP Server and type in the IP Address of the Arduino as illustrated in Figure 15.10.

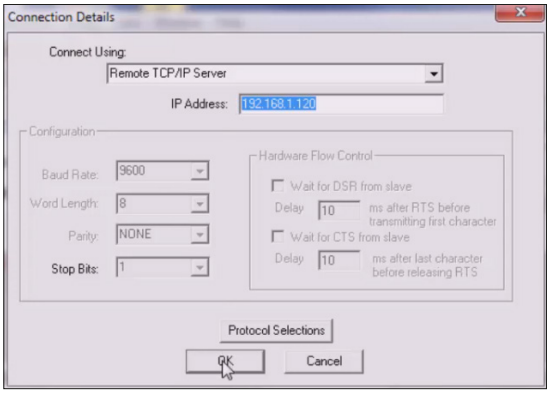


Figure 15.10: Setting the "Connection Details".

So, you're telling modscan32.exe that this is the IP address of the Arduino, and this is where you will read from. Click on the OK button. There we go, perfection 12, which is tank height. So right now, we are reading the input register that we created within the Arduino and we are doing so via Modbus TCP as shown in Figure 15.11.

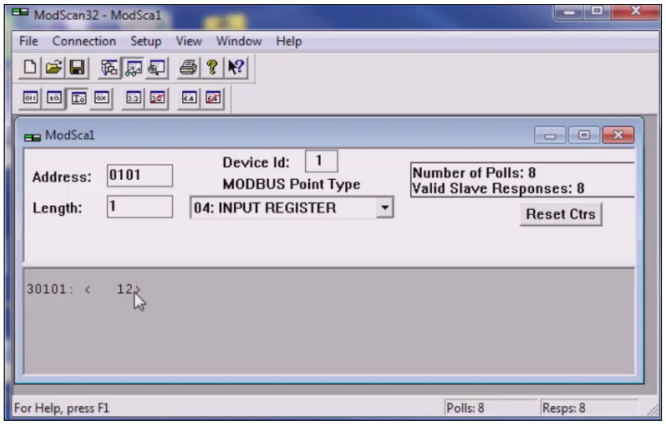


Figure 15.11: Reading the input register with address 100.

So, we have created a brand-new Modbus TCP server. All right, it has one register, but it is working. Now have a look at the blinking lights that you are seeing here on the Arduino in the Figure 15.12.

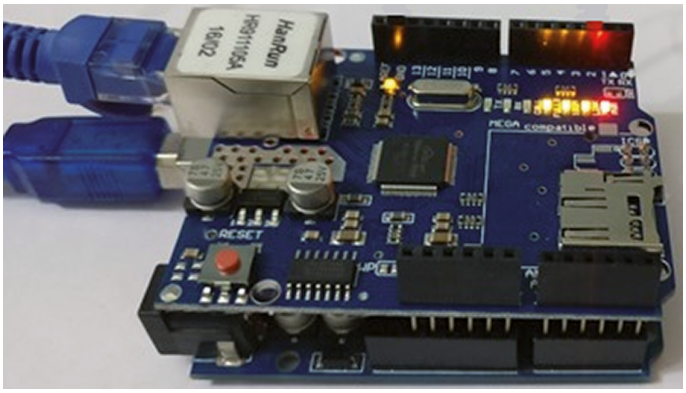


Figure 15.12: Blinking lights on the Ethernet Shield.

So, you are seeing those lights going like crazy there. That is the data transmission messages that are going back and forth there on the Arduino while the modscan32.exe pulls. Thus, we have done it. Now in the next section, we are going to expand our application to holding registers, input status registers, and coils.

Chapter 16 • Running the Arduino Modbus TCP Server Program

In the last section, there is one thing that we forgot to mention: make an edit to one of the libraries and this must be done before you run the application in the next section. So let us look at it here. Go to "Desktop", "Libraries", "Documents", "Arduino", and "libraries" and have a look at ModbusIP, and the actual files in it. We are going to concentrate on the ModbusIP.h. Open it. You can open it in any text editor that you have. It is an old Windows application, for old Windows operating systems. We are going to use one called Notepad++, our text editor of preference, as shown in Figure 16.1.

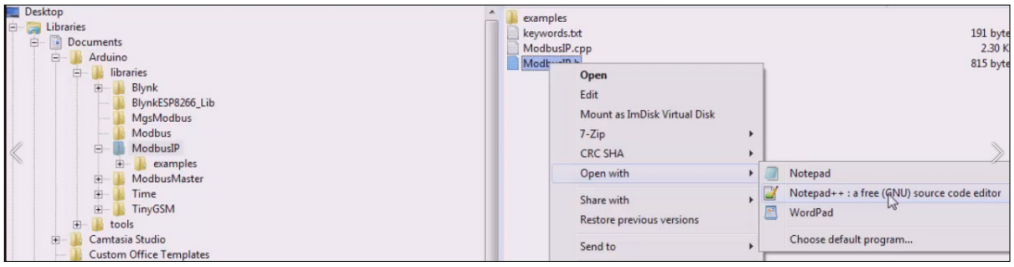


Figure 16.1: Editing ModbusIP.h file with Notepad++.

Now in your file, we want you to remove those comments and then save that file. So, when you edit the file, it should look like Figure 16.2. The reason that we have removed the comments here and essentially activated the line, TCP_KEEP_ALIVE, is for a particular reason that has to with calling a TCP connection.

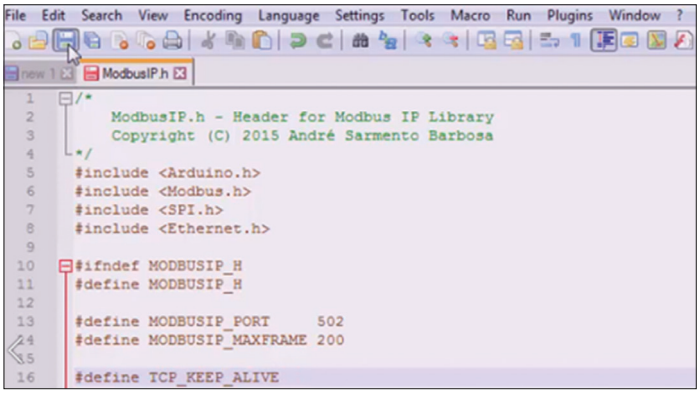


Figure 16.2: Activating the line, TCP_KEEP_ALIVE.

In Modbus communications, the underlying network connection is what you call a TCP connection, and if this line remains commented out, what will happen when we actually run the program, and we use modscan32.exe to read data from the Arduino as a Modbus TCP server? It will be one. It will do one read, and then the connection will be terminated. This TCP_KEEP_ALIVE directive instructs the code in the Arduino to keep the Modbus TCP connection alive so that Modbus requests and responses over Modbus TCP can keep going

until you end the conversation. We shut off the Arduino, that sort of thing. Do this edit, remove those comments so that you activate this line. Do this before you verify and upload, and do a final upload of the actual code into the Arduino. Now it should be fine with the test for the other sections.

Chapter 17 • Adding Code to Read a Holding Register

All right, so in the last section, we created a Modbus TCP server with a single register, which is an input register, and we read it successfully using the modscan32.exe as a Modbus TCP client. Now what we are going to do in this section is include a holding register in this application. So as before, we are going to save it as a new application with "File", then select "Save as" , and we name it "Serial05" as shown in Figure 17.1.

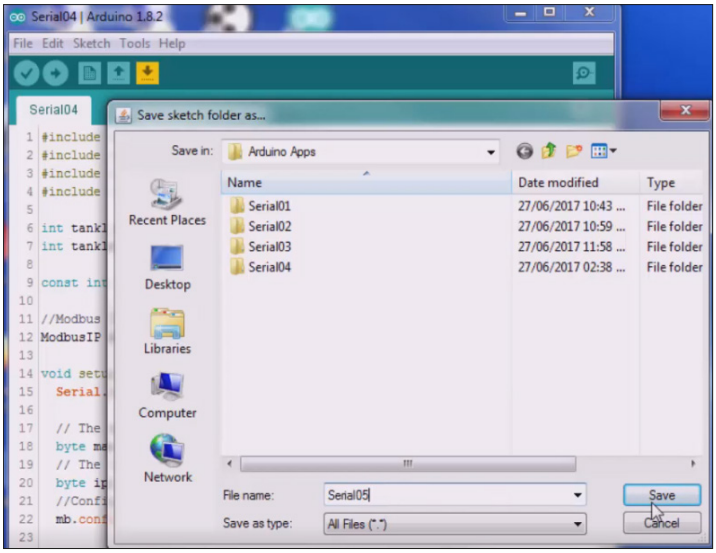


Figure 17.1: Saving the file as "Serial05".

All right let us get started with making modifications. In our new hypothetical scenario, let us imagine that we have a Modbus TCP server. This server has a holding register, which functions as an analog output connected to a variable frequency drive (VFD) on a motor. To represent the VFD speed, we will declare another integer variable. Additionally, we need to include a variable called vfdspeed_old to detect changes in the VFD speed. This variable will be used by the application to send a serial message indicating a change in value. Now, let's assign an address for the holding register, which we will call vfdspeed_hr. We will assign it the value 105, but you can choose any value you prefer. You can refer to Figure 17.2 for further clarification. Moving on, we need to add **mb.addHreg(vfdspeed_hr)** to our program. This command is for adding the holding register, and you can find all of these commands in the libraries provided. We have shared a weblink with you to access these libraries, which we recommend marking as favorite. Feel free to explore and read more about these commands in detail.

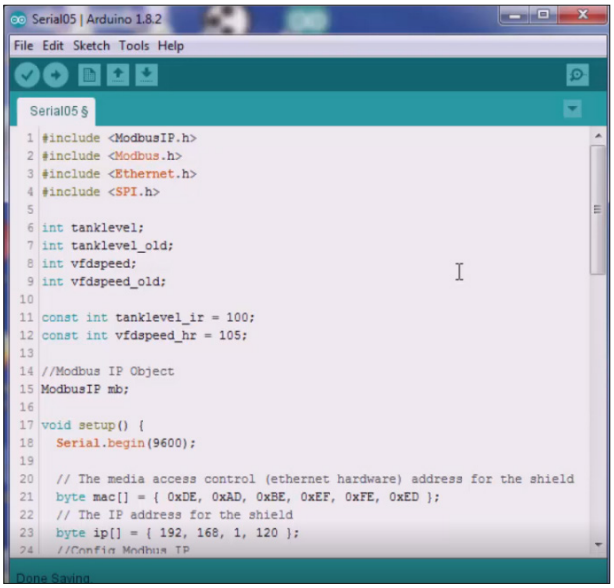


Figure 17.2: Defining vfdspeed, vfdspeed_old and vfdspeed_hr.

So, we show you how to use it practically, and you can always go back and refer to it. Now, we have added a holding register. Let us put an initial value for vfdspeed of 150 RPM. The vfdspeed_old value is also 150 RPM. So, we have set the vfdspeed variable to 150, and then we take the same value from vfdspeed and put it into our holding register location, which is vfdspeed_hr. Just like that, we just added another register, this time a holding register, as shown in Figure 17.3. Now let us take care of the visualization. Display current values using the function **DisplayCurrentValues(**). We not only want to display the tanklevel, now we want to display the vfdspeed. So, what we are going to do is to add a line temp string called "tempstr".

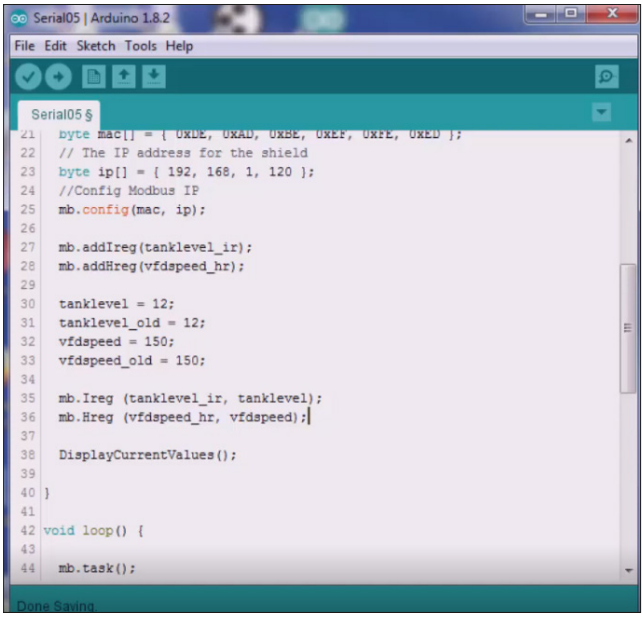


Figure 17.3: Adding code to the setup section.

We want to add onto tempstr, including the introduction of a separator, "VFD speed:", followed by vfdspeed and "rpm." Consequently, when displayed as illustrated in Figure 17.4, it should read tank level, distance in feet, separator, VFD speed, and RPM.

Furthermore, we need to modify the function **CheckForDataChange()**. Our objective is not only to monitor changes in tank level but also in vfdspeed. To streamline this process, copy and paste. Thank heavens for copy paste. One of Microsoft's greatest inventions. Thus, if vfdspeed is not equal to vfdspeed_old, the data has indeed changed. Subsequently, we update one to match the other, as showed in Figure 17.5.

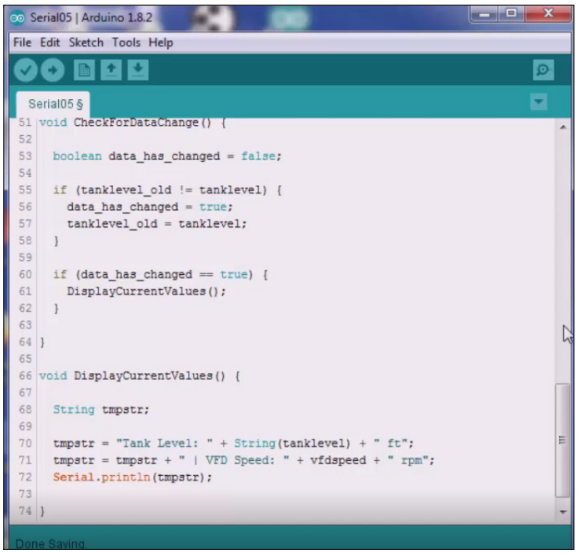


Figure 17.4: Adding code to the function **DisplayCurrentValues()**.

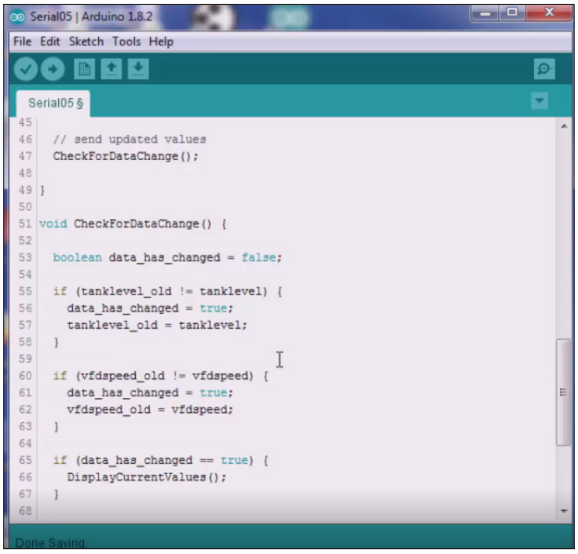


Figure 17.5: Adding code to the function **CheckForDataChange()**.

All right, so now there is one last thing that we need to do in this loop here. We have the mb.task() responsible for running in Modbus, and following that, we check for data changes using the function **CheckForDataChange()**. What we're going to do is integrate the following: vfdspeed is equal to mb.Hreg(vfdspeed_hr), as depicted in Figure 17.6.

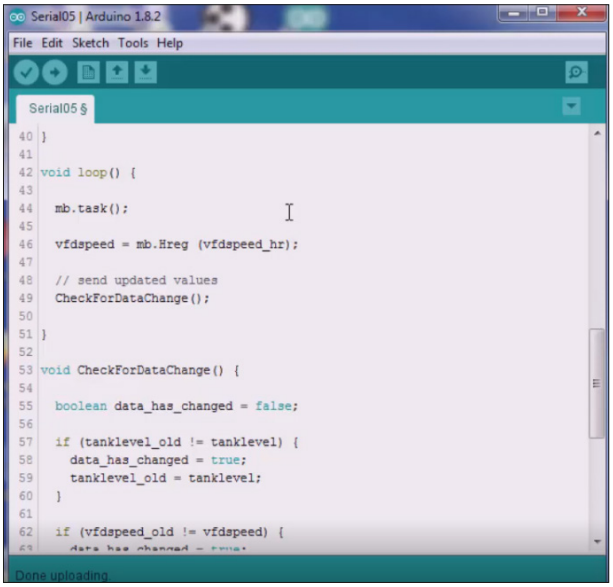


Figure 17.6: Adding code to the loop section.

Let us understand the purpose of this operation. In addition to the conventional display and read process of the polling register using modscan32.exe, we are making modscan32.exe to write a new value to this register. When the write occurs, it directly updates the value in the memory location, vfdspeed_hr. However, it is important to note that this value doesn't automatically transfer to the vfdspeed variable; you need to implement code to manage this transition. That is precisely what this segment of code accomplishes.

In essence, during each loop iteration, it retrieves the content from the holding register memory location and transfers it to the vfdspeed variable. It is worth mentioning that this process isn't necessary for the input register, as writing to an input register is not permissible. That sums up the functionality in this section.

Now we will bring up our serial monitor as shown in Figure 17.7, and we click there, upload the program, and transfer this application into the Arduino. There we go, "Tank Level: 12 ft | VFD speed 150 rpm" as depicted in Figure 17.8.

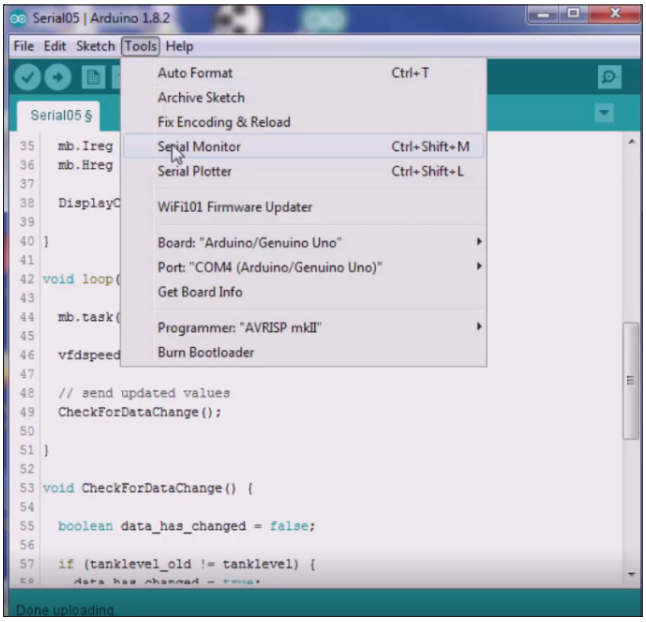


Figure 17.7: Selecting the "Serial Monitor".

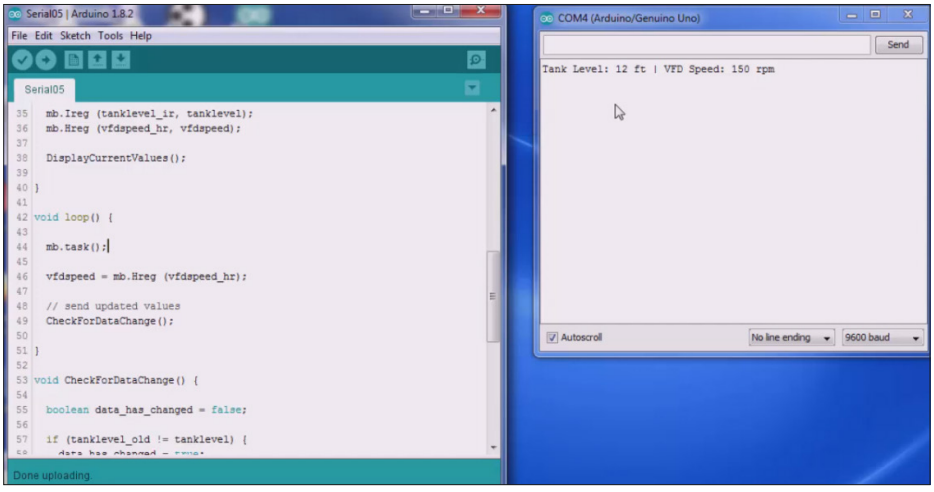


Figure 17.8: Uploading the program.

We have successfully configured our serial display. Now, let us move on to modscan32.exe. We previously used modscan32.exe to monitor the input register. Now, we'll create a new window for the holding register, specifically one register at offset 106, corresponding to our vfdspeed_hr address, which we set as 105. After setting up the connection, we click "Connect" (Figure 17.9).

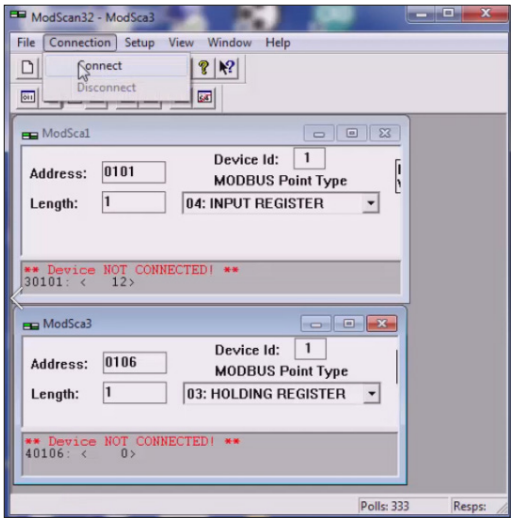


Figure 17.9: Choosing "Connection" and "Connect" in modscan32.exe.

In the "Connection Details" window, we click on the OK button as depicted in Figure 17.10, hoping to see our 150 RPM (Figure 17.11). Now, let us proceed to change this to 200 RPM by clicking on the displayed number, initially set at 150. Observe what happens in both the serial monitor window and here. Adjust the value to 200 and click on the update button, as illustrated in Figure 17.12.

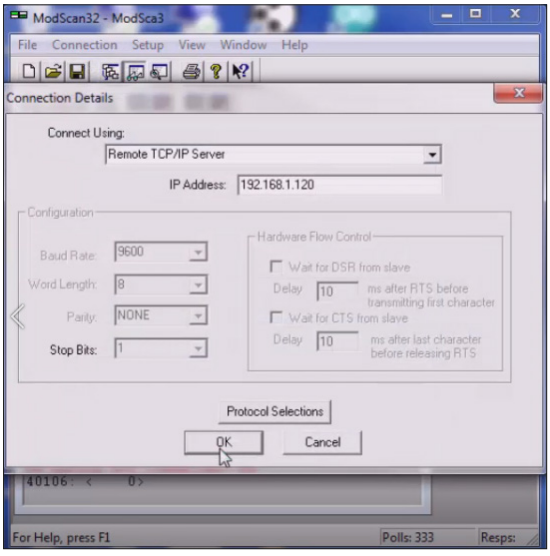


Figure 17.10: Clicking on the "OK" button.

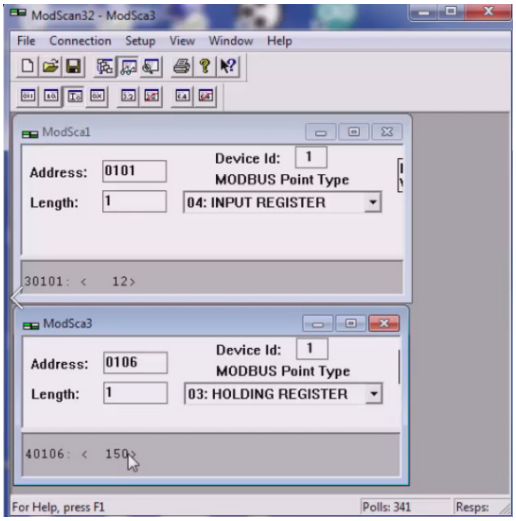


Figure 17.11: Getting our 150 RPM.

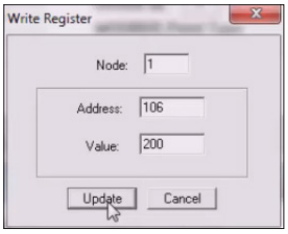


Figure 17.12: Clicking on the "Update" button.

As the values are updated, take note of the new line that appears. The application detects the change in vfdspeed and executes the **DisplayCurrentValues()** function again, presenting us with 200 RPM, as shown in Figure 17.13.

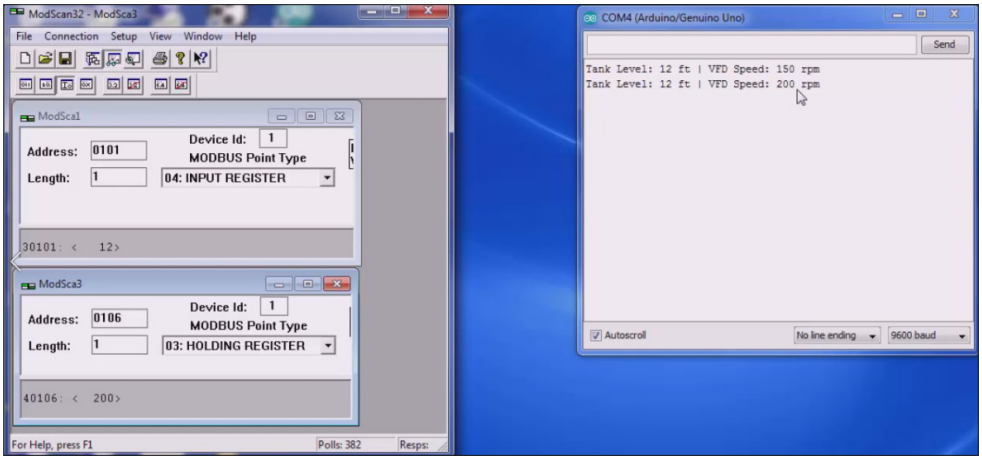


Figure 17.13: Updating and displaying a new line.

If you change this again, perhaps to 160 RPM, you will receive another update, as depicted in Figure 17.14. The serial monitor window gives really great feedback on changes in register values, allowing you to get a feel of the ongoing processes. In addition to that, this information is visible in modscan32.exe, where the changes are reflected. So good! We have expanded our application a little further. In the next two sections, we will go into input statuses and coils.

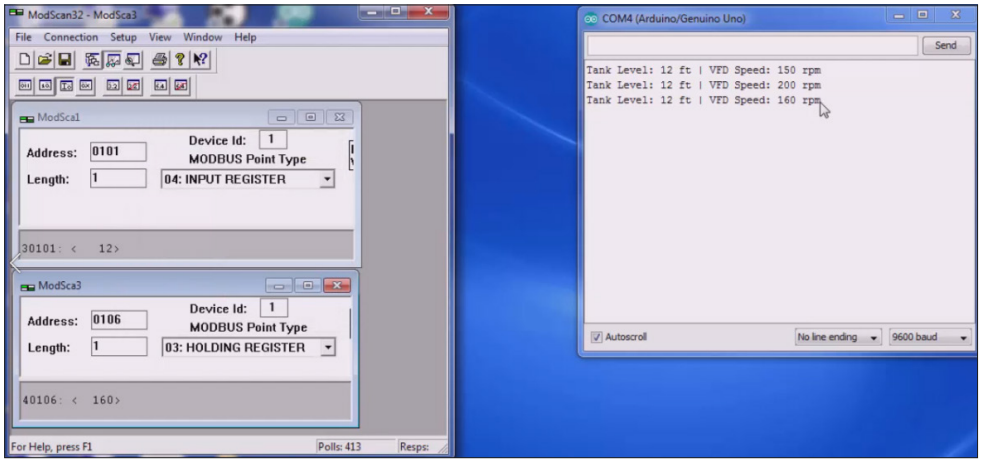


Figure 17.14: Updating and displaying a new line.

Chapter 18 • Adding Code to Read an Input Status

All right, so we are moving one step further in this section. We will add an input status, single input status to the application. So, go to "File", then select "Save as", and we name it "Serial06", as shown in Figure 18.1.

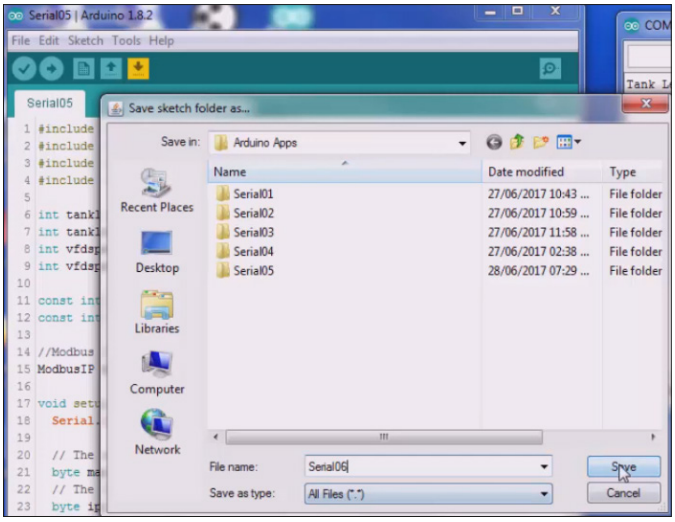


Figure 18.1: Saving the file as "Serial06".

We essentially do something similar to what we did in the last section. We are adding an input status, imagining a hypothetical situation where a valve is connected to the Arduino. The status of the valve, whether open or closed, is connected via a discrete input or digital input to the Arduino. Let us allocate a memory location, named the valvepos_is with value 110. The valvepos and valvepos_old variables are actually supposed to be of type Boolean, as the valve can only have one of two states: on or off, as depicted in Figure 18.2.

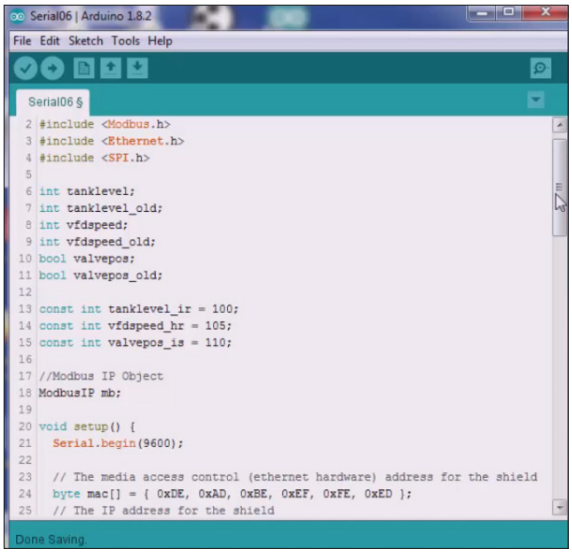


Figure 18.2: Defining valvepos, valvepos_old and valvepos_is.

All right, let us do the work in setup. So, we will use **mb.addIsts(valvepos_is)**. This is a command again from the documentation. We will add an input status along with its initial value. To initiate this, we will set the valve position to true. Thus, both the valvepos and valvepos_old variables will be assigned the value true. Simultaneously, we need to place the actual value of the valve position into our memory location, the input status memory location, using **mb.Ists(valvepos_is, valvepos)**, as illustrated in Figure 18.3.

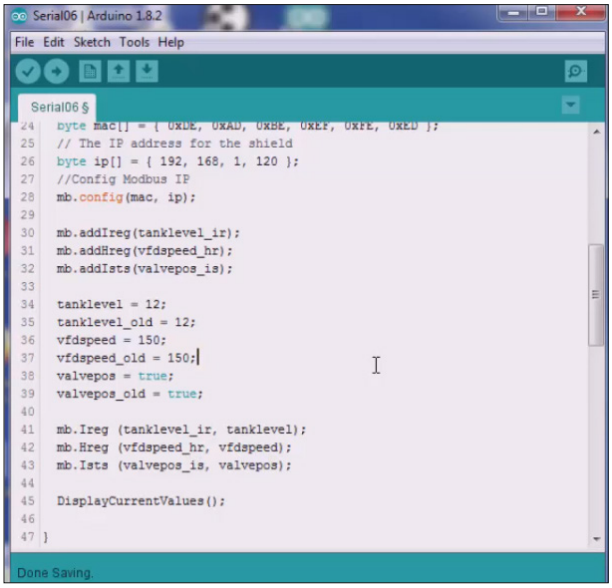


Figure 18.3: Adding code to the setup section.

All right let's address the operations within **CheckForDataChange** and **DisplayCurrentValues**. Now, for the valve position, we want to see something more descriptive than a binary one or zero. So, we are going to use a string position, named "pos". If valve position is true, then pos is equal to "closed" (indicating that the valve is closed). If it is not true, then the valve position is "open", as shown in Figure 18.4. considering we are initializing it to true, we should observe it displaying as "closed".

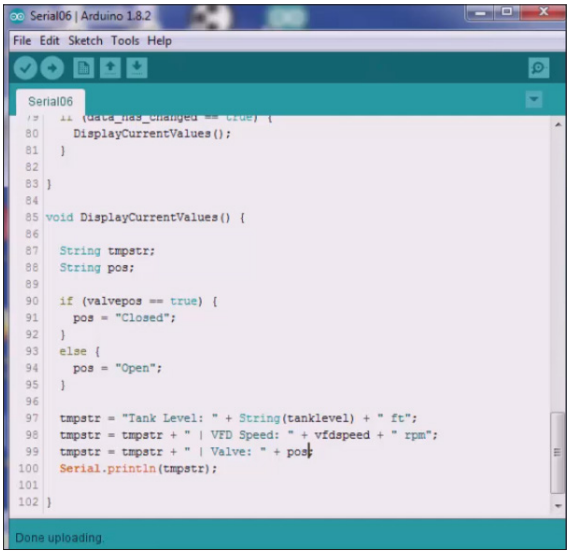


Figure 18.4: Adding code to the function **DisplayCurrentValues()**.

Then we will add some code to the **CheckForDataChange()** function to account for any changes, as depicted in Figure 18.5. What you can do also is physically link a discrete input on the Arduino to the valve position, activating it with wires. However, for simulation purposes, we will not need to include anything in the loop for transferring any value, as we cannot write to an input status; it changes from the actual input side. With these modifications, our application has been adjusted accordingly.

Now, we proceed to upload the program to Arduino. Upon completion, we should see something on the serial monitor. After uploading, there it is—Tank Level, Speed, and "Valve closed" because we initialized the variable to true, as illustrated in Figure 18.6.

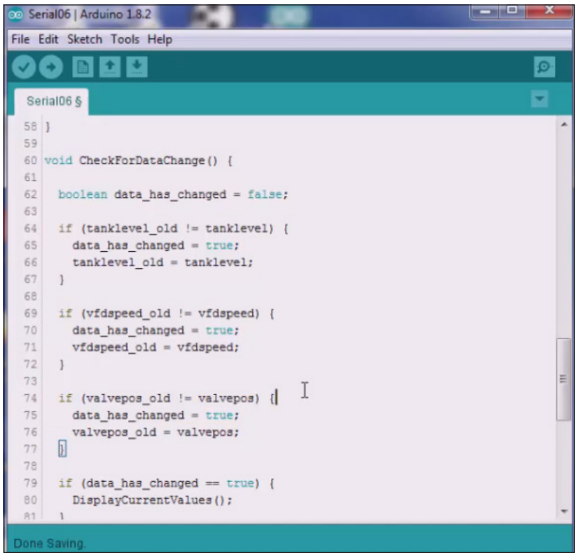


Figure 18.5: Adding code to the function **CheckForDataChange()**.

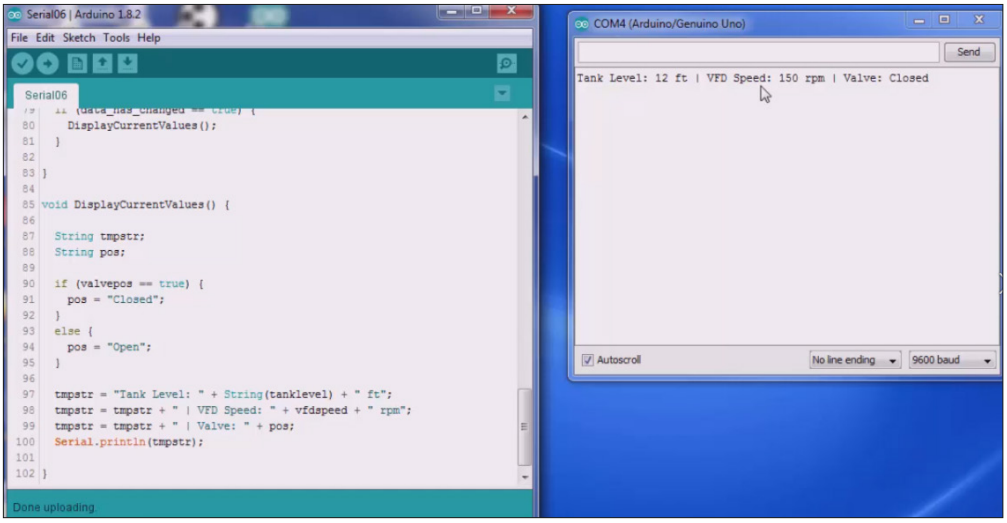


Figure 18.6: Uploading the program.

Now, let us minimize this program and bring back modescan32.exe, and go through the other same process. Go to "File", select "New", then proceed to "Input Status". Set the address to 111, aligning with our choice of 110 in the program. Place it there, move to the "Connection" tab, and finally, click "Connect", as shown in Figure 18.7.

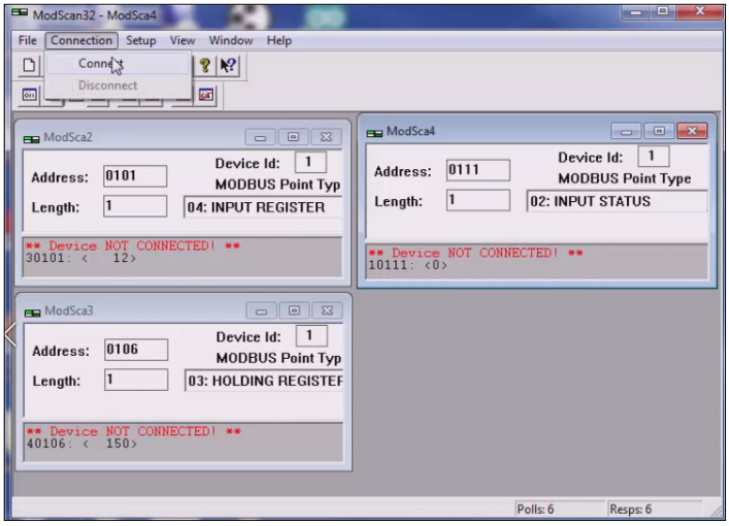


Figure 18.7: Choosing "Connection" and "Connect" in modscan32.exe.

There we go; you can see it reads "1" because it was initialized to true, as depicted in Figure 18.8. We are now successfully reading an input status from our Modbus TCP server via our modscan32 TCP client. In the next section, we will discuss coils, marking the completion of all registers and bringing us to the end of Modbus TCP implementation on the Arduino.

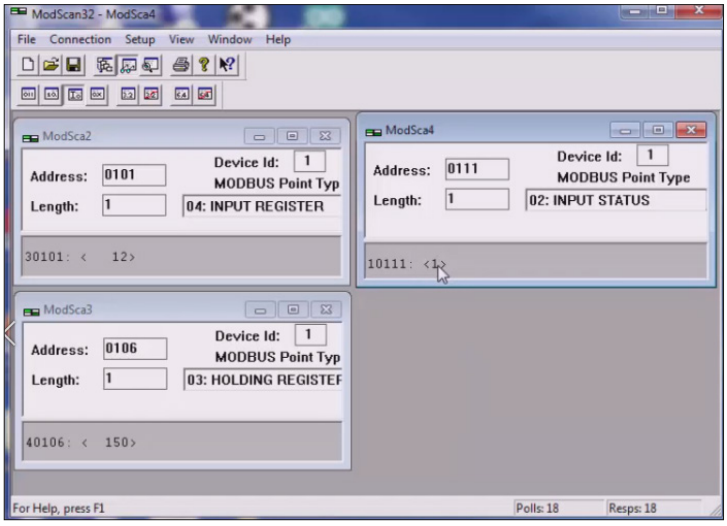


Figure 18.8: Getting the value of "1", closed.

Chapter 19 • Adding Code to Read a Coil

In this section, we are expanding this application one final time, including the coil. Remember, there are four main types of memory in any Modbus slave or server: holding registers, input registers, input statuses, and coils. Having covered three of them, we will now tackle the last one, the coil. Following the same procedures as before, let's consider a hypothetical situation where the coil is connected to a pump, controlling it's on and off state. When the coil is zero, the pump is off, and when the coil is one, the pump is on. So, let's declare a Boolean variable for the pump—pump, using Boolean because, like an input status, it can have one of two values, one or zero. Now, we need to allocate a memory location for it. Let's call this pump_cl for coil, choosing its value as 115, selected arbitrarily as shown in Figure 19.1.

To support this in the code, we add the following in the setup: **mb.addCoil(pump_cl)**. Let's give this an initial value of pump equal to false and pump_old equal to false. This implies that initially, the pump will be off, having a value of zero, and we must initialize it with this value.

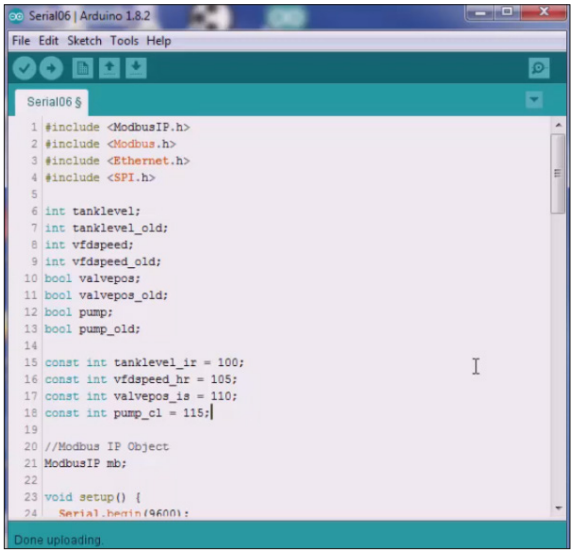


Figure 19.1: Defining pump, pump_old and pump_cl.

Again, we have **mb.Coil(pump_cl, pump)**. That command is in the documentation, so what we are doing is that we are taking the value of pump which is false, we have initialized it, and we are putting it into the memory location allocated for the coil, pump_cl, as depicted in Figure 19.2.

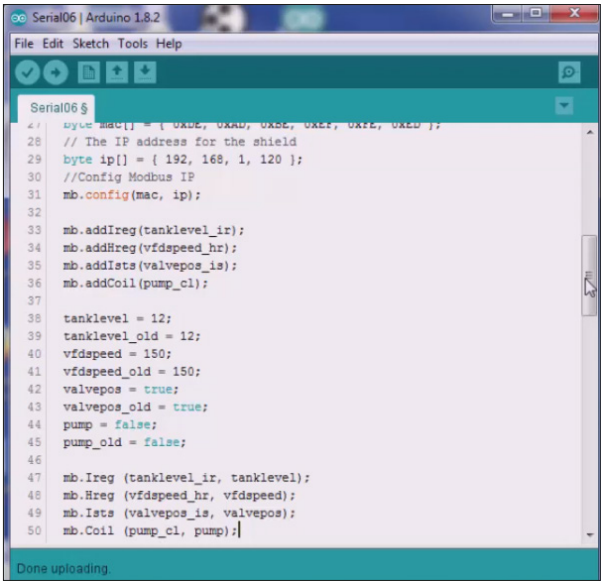


Figure 19.2: Adding code to the setup section.

Moving on to the loop section, given that the coil is writable, we need to transfer any value written to it by modscan32.exe to the variable. Therefore, we add the following line: pump is equal to mb.Coil(pump_cl), as illustrated in Figure 19.03.

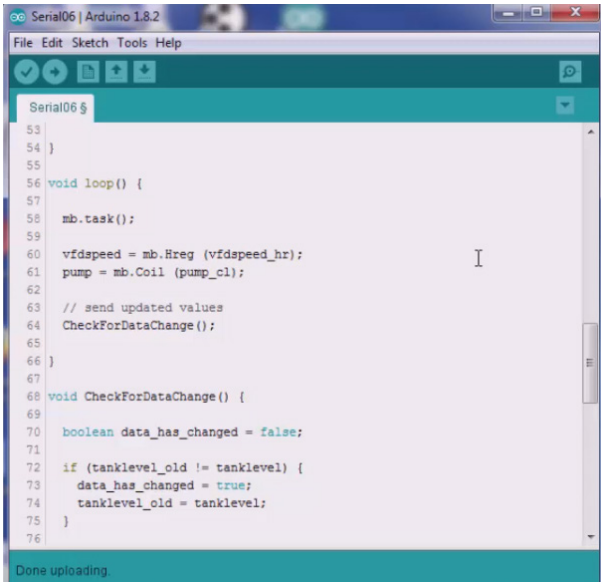


Figure 19.3: Adding code to the loop section.

Essentially, if modscan32.exe writes to the pump_cl location and changes the value from 1 to 0 or 0 to 1, it will be sent into the pump variable, allowing the **CheckForDataChange()**

function to carry out its work. Now, with that in place, let us do our magic to the display section. We define a string variable called pstat, representing pump status. If pump is equal to true, then pstat is equal to "Pump is in run mode". If it is not, then pstat is equal to "Pump is stopped", as shown in Figure 19.4. This will display "Pump run" or "Pump stop" based on the variable. Of course, we also need to check for data change in the **CheckForDataChange()** function, as depicted in Figure 19.5.

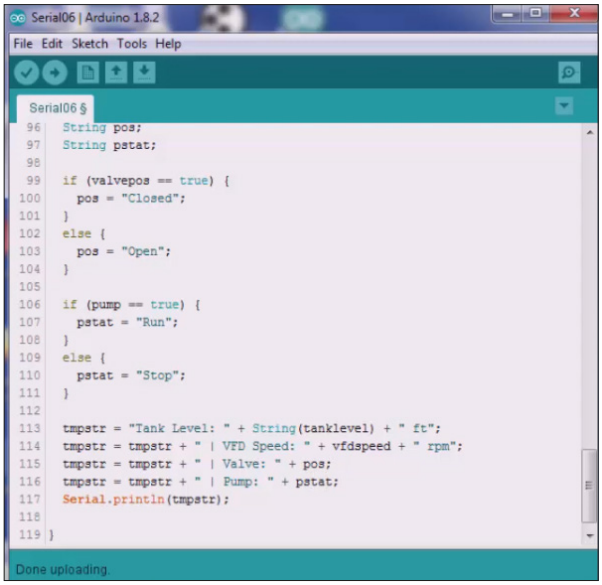


Figure 19.4: Adding code to the function **DisplayCurrentValues()**.

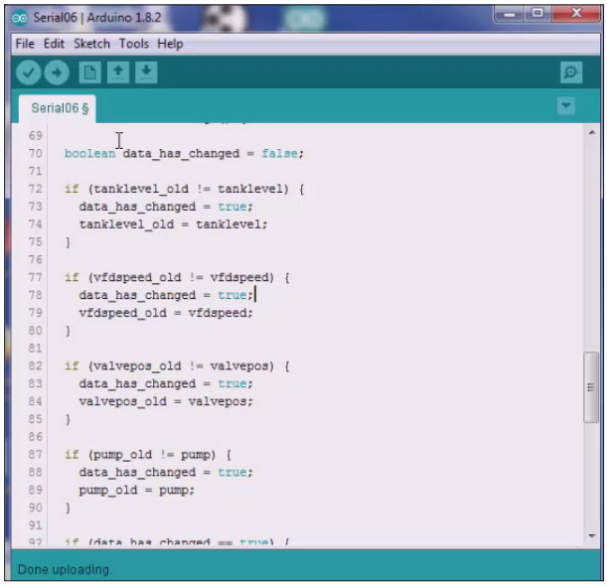


Figure 19.5: Adding code to the function **CheckForDataChange()**.

We are going to save this application as "Serial07", as illustrated in Figure 19.6. We should have done that before, but do not forget to do it. So, keep your application separate.

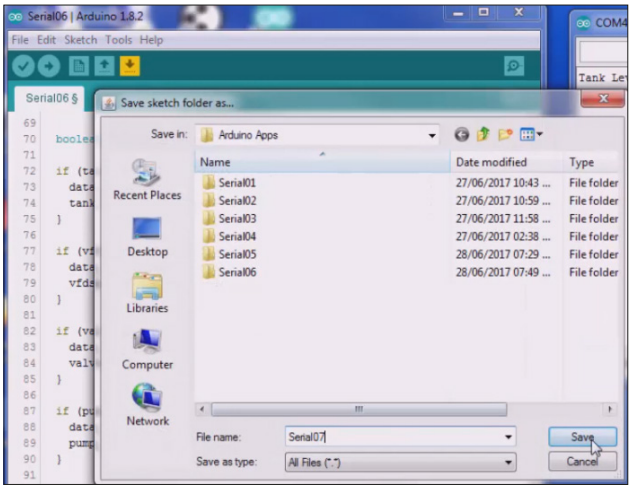


Figure 19.6: Saving the file as "Serial07".

All right, so let us click upload and the application that we have just written is uploading to the Arduino. Let us see what we see on the serial monitor. There's Tank Level, VFD speed, Valve and Pump. As expected, the pump is in stop mode because we initialized it to false, as shown in Figure 19.7.

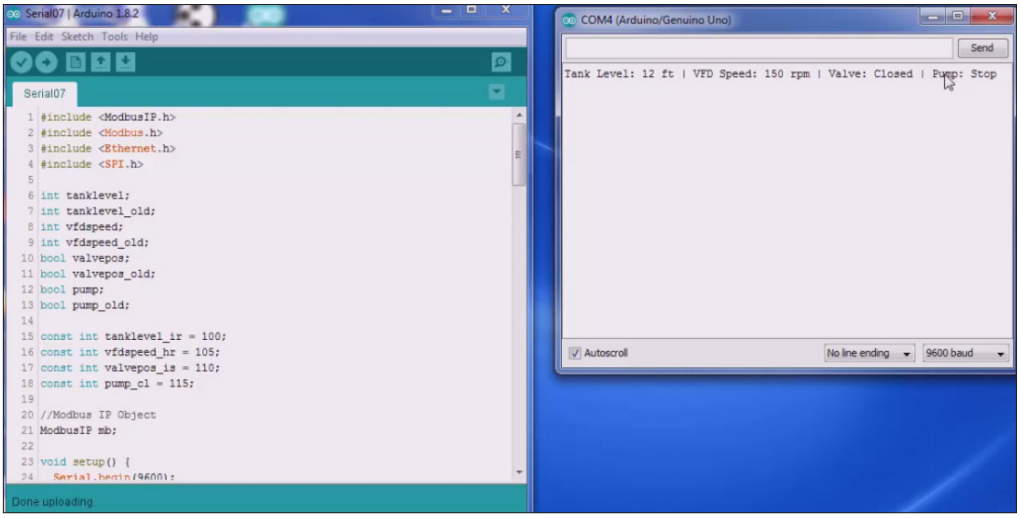


Figure 19.7: Uploading the program.

All right let us close this down for now and let us go to modscan32.exe. Going to "File" → "New", we have the coil already chosen with the address 115. However, we will select 116 due to the offset. We want to read a single coil, then proceed to the "Connection" tab and

click on "Connect", as depicted in Figure 19.8.

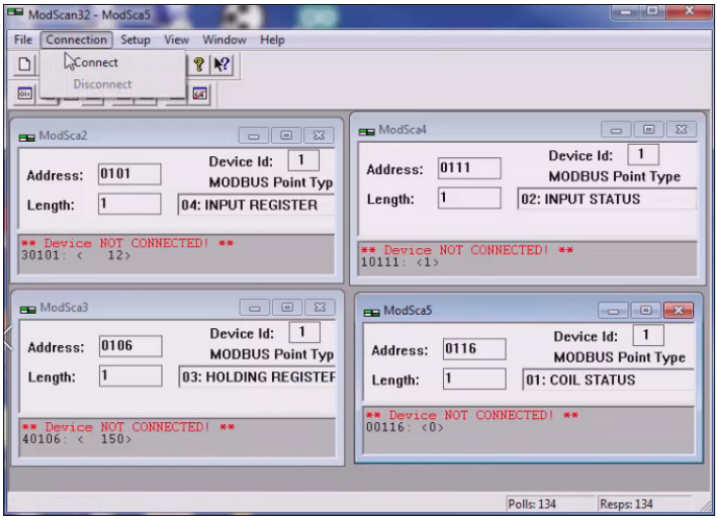


Figure 19.8: Choosing "Connection" and "Connect" in modscan32.exe.

We are reading zero there because it was initially set to false as shown in Figure 19.9.

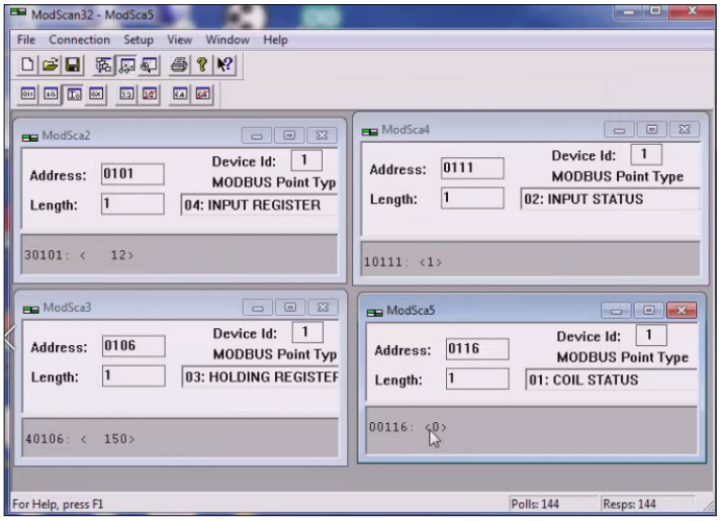


Figure 19.9: Getting the value of "0", stop.

We are going to double click and change it to one, as depicted in Figure 19.10. Watch what happens here; observe the serial monitor. As soon as it changes to one, a new line appears with the tank level, VFD speed, valve closed, and the pump now in run mode, as illustrated in Figure 19.11.

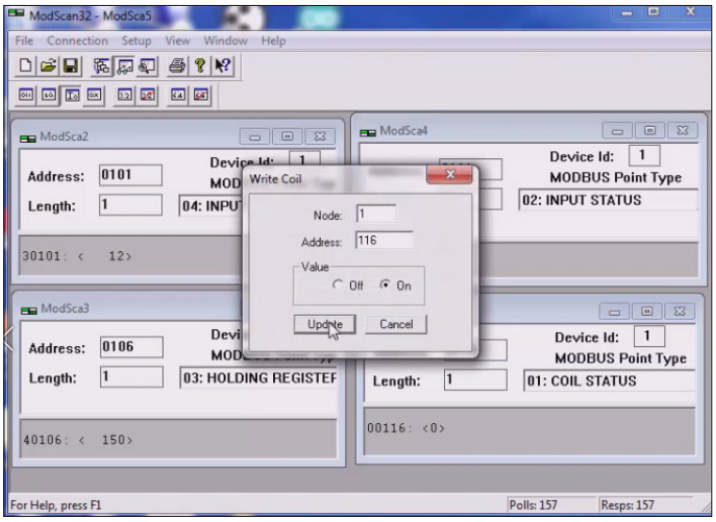


Figure 19.10: Changing the coil value.

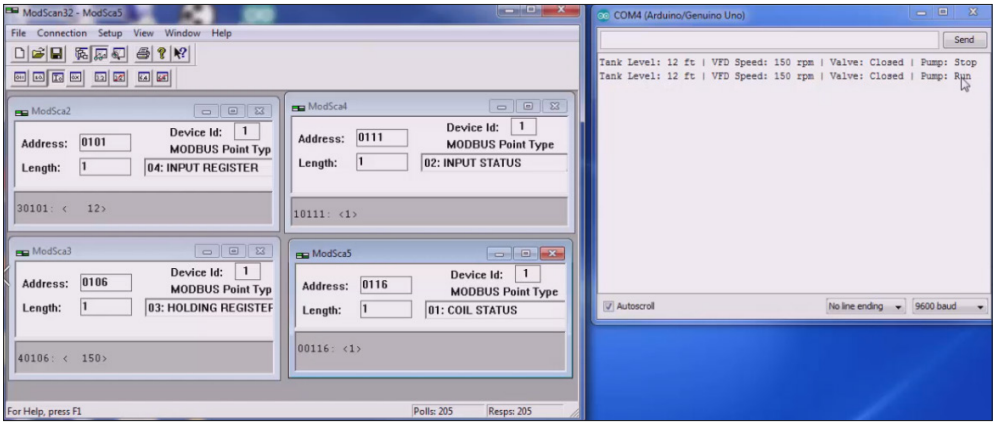


Figure 19.11: Change in modscan32 and serial communication window.

That is very cool. We can change the RPM as well and we will update it to 230 RPM. Change this back to stop mode, then click update. You will see "Stop" occurring right there, as shown in Figure 19.12, Figure 19.13, Figure 19.14, and Figure 19.15, respectively.

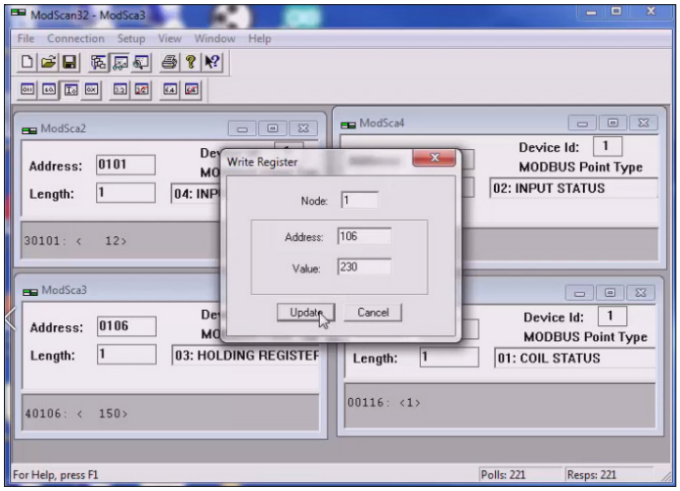


Figure 19.12: Changing the holding register RPM value.

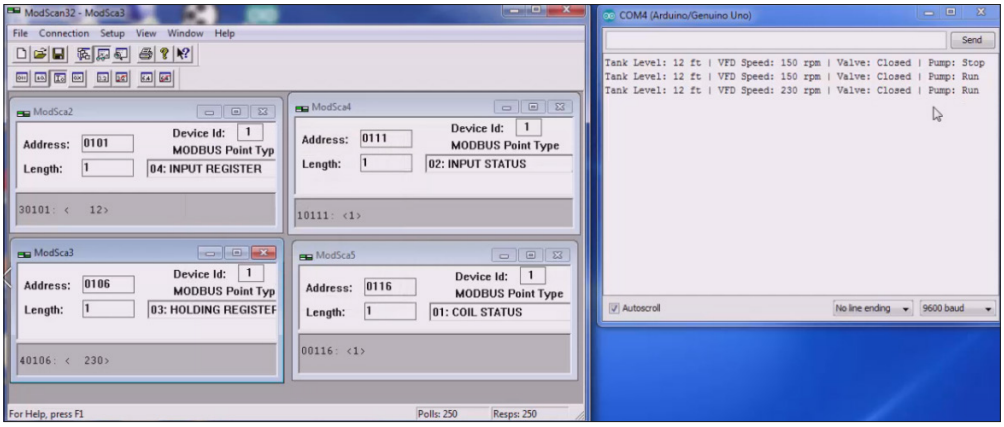


Figure 19.13: Change in modscan32 and serial communication window.

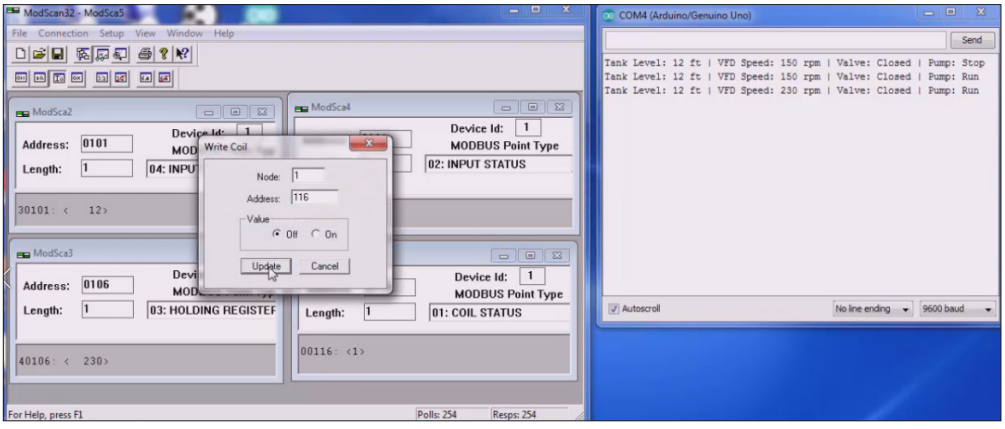


Figure 19.14: Changing again the coil value.

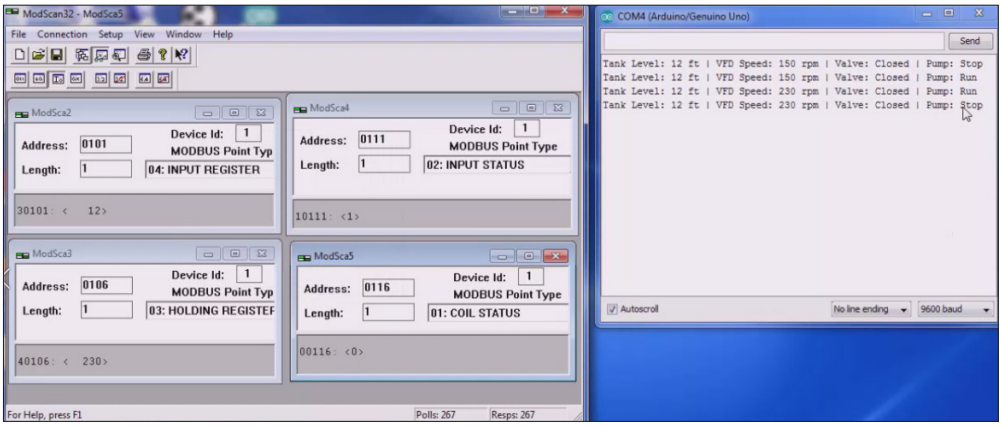


Figure 19.15: Change in modscan32 and serial communication window.

So, we have done it! Now, we have a Modbus TCP server that encompasses all four types of registers. We can read these registers and write to the holding registers and coils. That is a full Modbus TCP server, and you can take this code now, port it to your own application, and customize it. It serves as an excellent starting point for your Modbus TCP implementation.

Chapter 20 • Understanding the Modbus TCP Client Task

All right, so we are on to a brand-new section. In the previous two sections, what we did was show how to configure the Arduino as a Modbus TCP server. Looking back at the diagram in Figure 5.1, the Arduino Uno, together with the Ethernet shield, was configured as a Modbus TCP server. On the laptop, we used modscan32.exe to be a Modbus TCP client. In the world of Modbus TCP/IP, the client is like the master. So, what modscan32.exe did was, through the network, it read values from the Arduino Uno. We had set up the tank level as an input register, vfdspeed as a holding register, valvepos as an input status, and pump as a coil, as depicted in Figure 20.1.

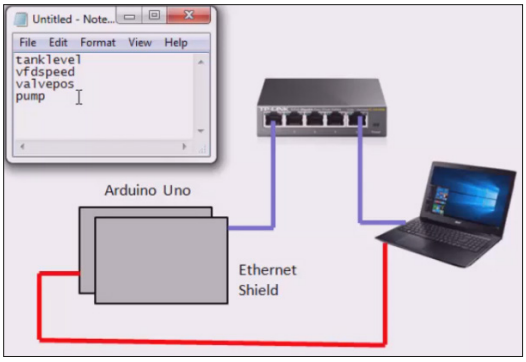


Figure 20.1: Defining input register, holding register, input status and coil.

Thus, a Modbus TCP server obviously must have registers in it to be read and written to. We use modscan32.exe to both read and write values. Every time we wrote a value and it changed, we use the serial monitor, the red line in Figure 97, to get feedback that there was a change. So, the Arduino sent a message through the programming cable, which appeared on the serial monitor, providing feedback.

But the real core of the network is essentially the Ethernet network. Now, in this section, we are going to do the opposite. The Arduino Uno will be the Modbus TCP client, and the laptop will be the Modbus TCP server. We will run modsim32.exe as the Modbus TCP server and configure registers within it. In the Arduino Uno (the TCP client), we will write code to read data from the laptop, acting as the Modbus TCP server, and record that data. Additionally, it will provide feedback to the laptop's serial monitor, confirming that the Arduino Uno is indeed reading the values from modsim32.exe.

Remember the red line shown in Figure 97. This programming cable is using the serial function just for feedback to do our testing. Now we will take our final application and modify it. So, we are going to leave the tanklevel, vfdspeed, valvepos and pump registers. However, those will not be registers in a Modbus TCP server. We are going to leave them as variables in the to store the values read from the Modbus TCP server on the laptop. Therefore, we will leave all of these intact, including the serial display, but change the purpose of these variables. They will now store the data that we will read from our Modbus TCP server. This is the goal of this section.

Chapter 21 • Configuring Modbus TCP Client Library in the Arduino IDE

In the last two sections where we configured the Arduino and the Ethernet shield together as a Modbus TCP server, we used a specific library. Before that, we had you download two libraries—one as an attachment and the other from an external link. For this section, we will be using the second library, the one you downloaded directly and was attached as a resource. Upon downloading and extracting the files, you would have obtained this folder, as we walked you through earlier, named "MgsModbus". This is the library we will use to implement the Modbus TCP client on the Arduino, as seen in our directory in Figure 21.1.

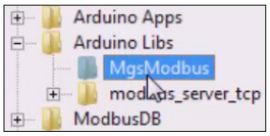


Figure 21.1: The directory of the "MgsModbus" folder.

So, you are going to copy this entire folder here, and then follow the same steps as before. Navigate to "Desktop", "Libraries", "Documents", "Arduino", "libraries", and paste it into the "libraries" folder, as depicted in Figure 21.2.

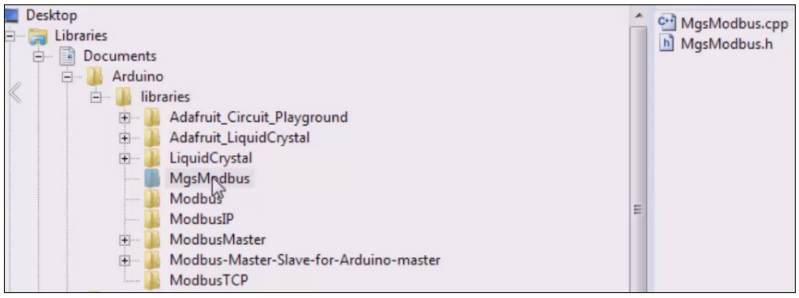


Figure 21.2: Doing copy and paste the "MgsModbus" folder.

This is what you should do. You should have a library called "MgsModbus" with these two files in it— "MgsModbus.h" and "MgsModbus.cpp". You should have those two files in your folder, ready to be used in our application. In the next section, we will show you how to include the new library into your code file and then use it from there by calling functions and using properties.

Chapter 22 • Removing the Modbus TCP Server Code from the Program

What you are seeing here is the application from the last section, named "Serial07". We are going to take this application, remove the code that implements the Modbus TCP server, and replace it with new code that implements the Modbus TCP client. However, as mentioned before, we will keep our tanklevel, vfdspeed, valvepos, and pump variables. We will do this gradually because modifying code can be tricky, and it's easy to forget or delete something important. Let's go step by step. This might take a couple of sections.

The first thing we are going to do is remove these two libraries, as shown in Figure 22.1. These libraries, combined into one, were downloaded to implement the Modbus TCP server. We no longer need them.

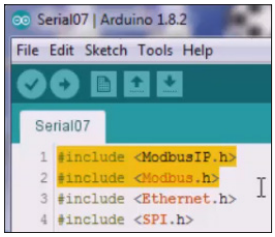


Figure 22.1: Deleting the Modbus TCP server libraries.

We are going to delete that and save the program as "Serial20" (Figure 22.2).

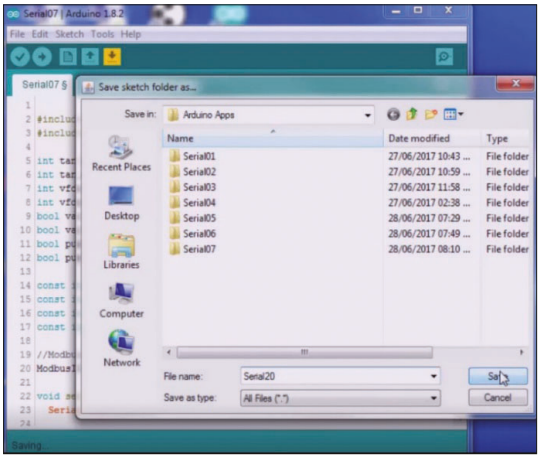


Figure 22.2: Saving the file as "Serial20".

Now we are going to include the copied library in the previous section. Go to "Sketch", then "Include Library", and select "MgsModbus", as illustrated in Figure 22.3.

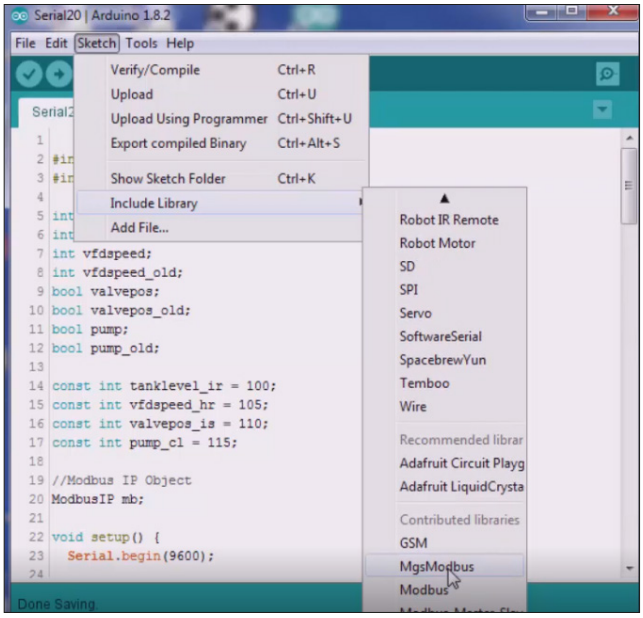


Figure 22.3: Selecting the MgsModbus library.

It is included right there. We have essentially removed the library responsible for Modbus TCP server and included the one that handles Modbus TCP client. These registers are no longer necessary because they allocated spaces for holding registers, input registers, etc., which are not needed for a Modbus TCP client, as shown in Figure 22.4. So we are going to take out the object "ModbusIP mb" because we do not need it as it was a Modbus TCP server object as depicted in Figure 22.5.

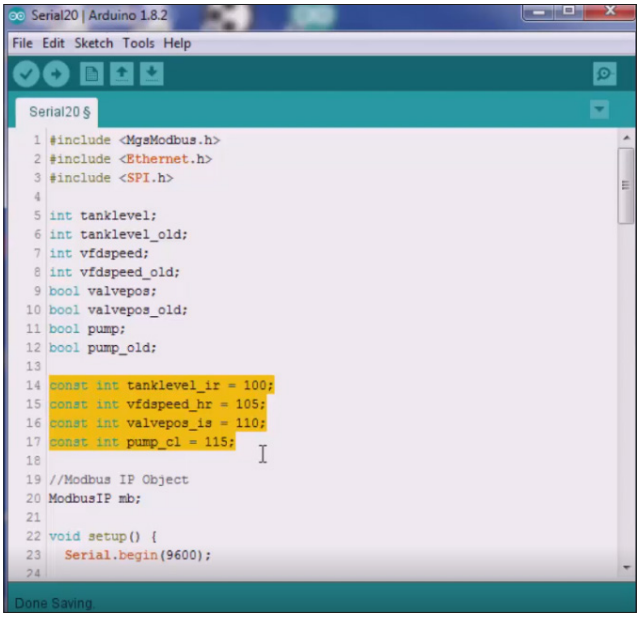


Figure 22.4: Deleting Modbus TCP server registers.

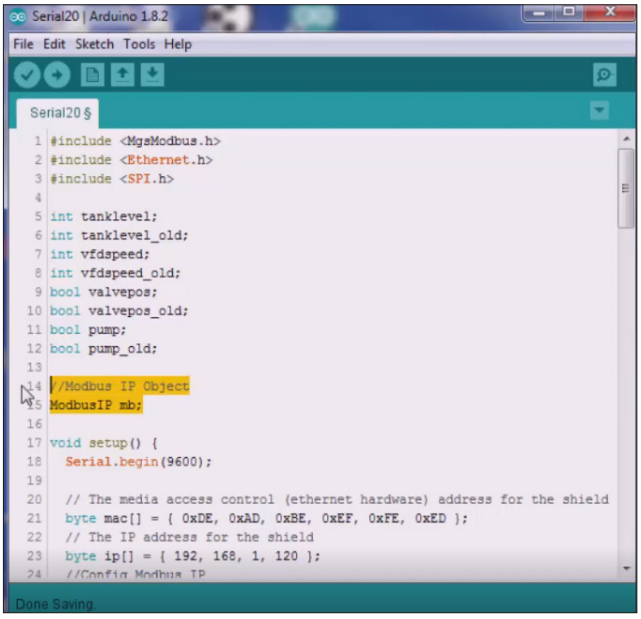


Figure 22.5: Deleting the object "ModbusIP mb".

Let us see what else we do not need. We can remove the MAC and IP configurations (Figure 22.6), as well as the registers related to creating (Figure 22.7). The initialization of variables can stay for now, so let us leave that. Remove the unnecessary codes related to registers, as shown in Figure 22.8.

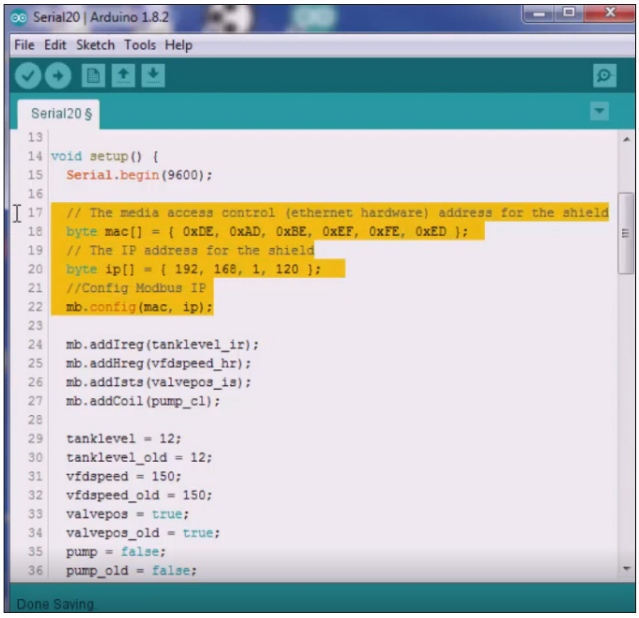


Figure 22.6: Deleting MAC and IP configurations.

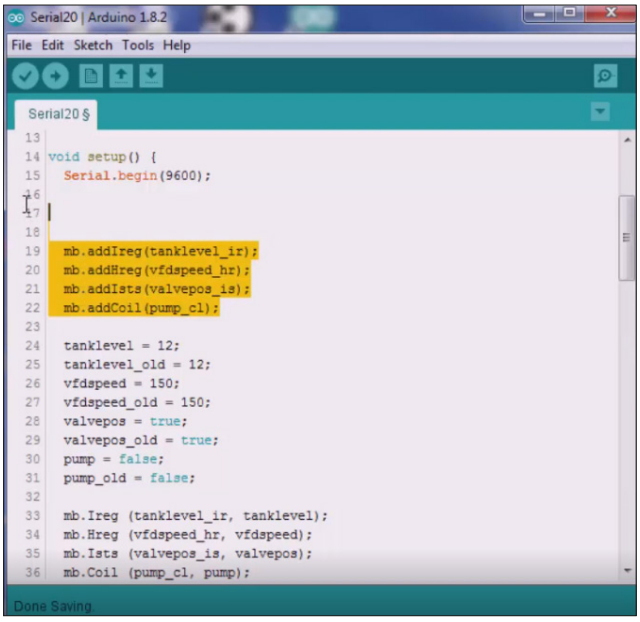


Figure 22.7: Removing registers related to creating.

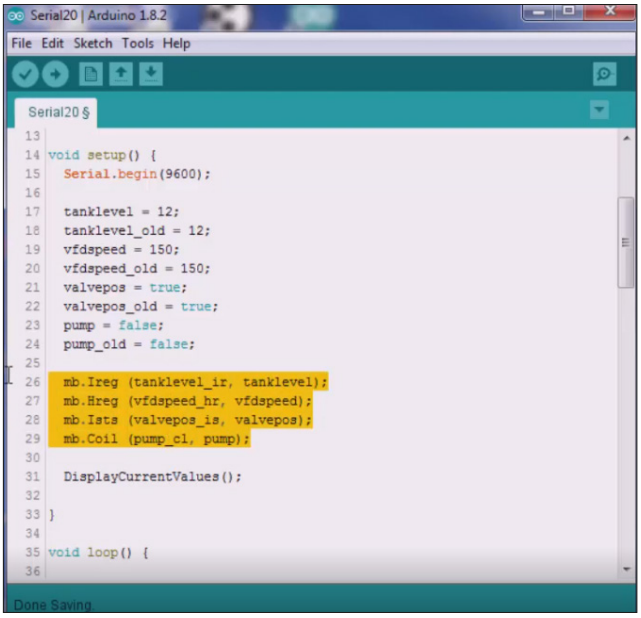


Figure 22.8: Removing codes related to registers.

We are going to leave our **DisplayCurrentValues()** function right there. Let us go further down to our loop section. We do not need the codes illustrated in Figure 22.9 and because this code implements a Modbus TCP server.

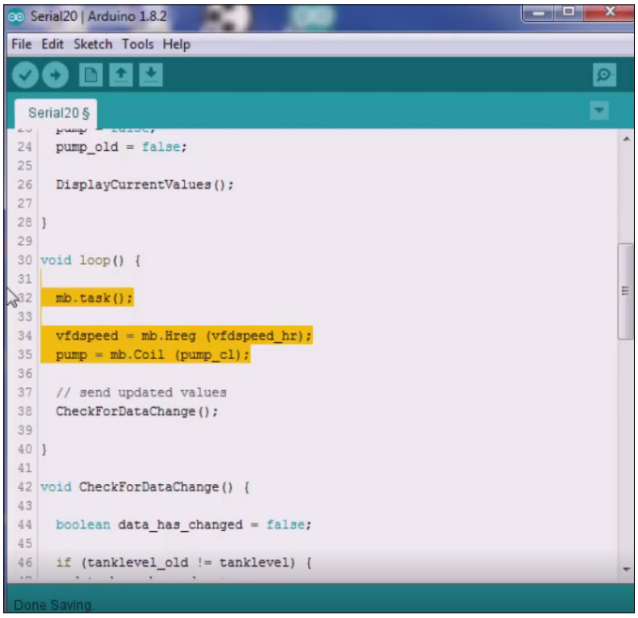


Figure 22.9: Removing codes from the loop section.

So, we leave the functions **CheckForDataChange()** and **DisplayCurrentValues()**. We have essentially removed the code that implemented the Modbus TCP server. In the next section, we are going include the code that implements the Modbus TCP client.

Chapter 23 • Setup Codes to Support the TCP Client Task

In the previous section, we included our new Modbus TCP client library and removed the code that implemented the Modbus TCP server. Now, we are ready to start including some code that supports Modbus TCP client.

Before the setup, declare an MgsModbus object called "Mb". In the previous code, we had to declare a Modbus object as well. You are declaring an object that encapsulates all the features of the library. We will declare another variable, just a normal integer and initialize it to zero. We are going to call this "accumulator, seconds counter". For now, we will leave that there and show you what this does later on when we are writing code to do the actual polling. In addition to that, we are going to paste in some code as shown in Figure 23.1, and we will explain it.

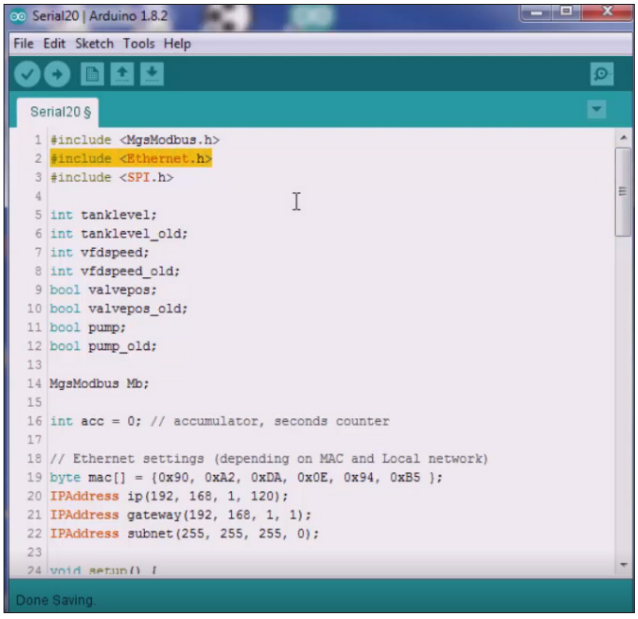


Figure 23.1: Adding codes to implement the TCP client.

So, these codes were taken from one of the examples from the downloaded library, which is **myarduinoprojects.com/modbus.html** as depicted in Figure 23.2.

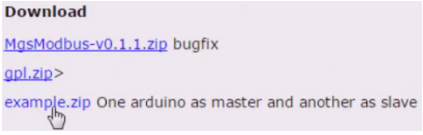


Figure 23.2: Adding IP address codes from the examples.zip.

As previously mentioned, the raw library did not function as expected and required modifications. However, helpful code examples can be found in the example.zip file provided by the library. Despite encountering challenges, useful code was obtained from this source. In the code snippet (Figure 23.3), these commands are related to declaring variables for the Ethernet library, rather than specifically for the MgsModbus library.

We have declared a MAC address for the Ethernet interface and set the IP address. We declare a variable which is the IP address. The IP address of the Arduino was 192.168.1.20 and the laptop one was 192.168.1.1.21. So, we are going to stick with that convention, and we define a gateway and subnet and so on.

Now let's move into the sort of setup area and we are going to put a statement in there called **ethernet.begin(mac, ip, gateway, subnet)**. This initializes the network connections, MAC, IP, gateway and subnet. Moreover, we put here the initialization of ethernet shield.

Now, let us observe how each library is used differently with various commands. In the next step, we will paste another statement in here, as illustrated in Figure 23.3. This statement declares the server's IP address that we will be polling. In this case, the laptop has the IP address of the server, which is 192.168.1.21. So, we define "Mb" as a Modbus object, specifying the server we will be interrogating for values through polling. The statement "Mb.remServerIP" is an array, and we assign the server's IP address to positions zero, one, two, and three, as shown in Figure 23.3.

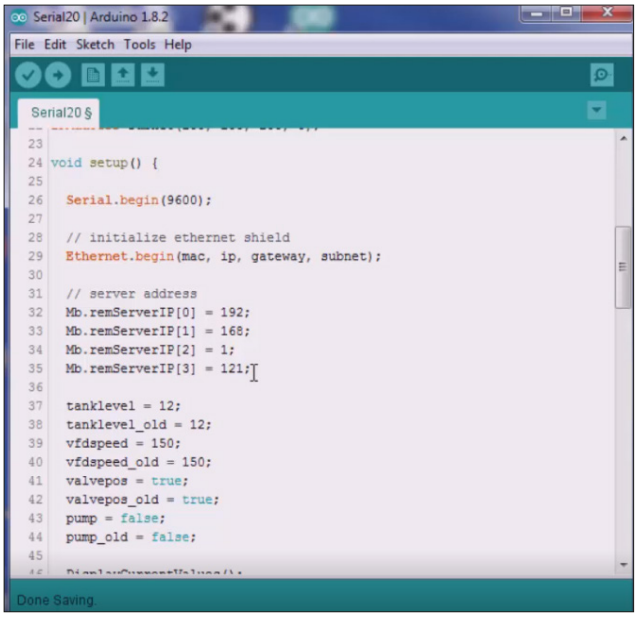


Figure 23.3: Adding codes to the setup section.

It is just how it is described; this is how it is done with this library. We have made some progress, though it's advisable to save when we have a completed, working version.

Currently, if you download this, it will not function as it lacks all the necessary code. While we've added some code related to the Modbus TCP client, particularly in the setup area and just before that, there's still no code for the actual polling of registers in the Modbus TCP server. We will get into that code in the next section.

Chapter 24 • Writing Codes to Poll a Single Register in Modbus TCP Server

In the last section, we introduced some code to support the actual polling of the Modbus TCP server by this Modbus TCP client. This included network support and the IP address of the server (modsim32.exe) we'll be polling, running on the laptop. Now in this section, we want to get into the guts of what we have to put in the loop. Specifically, we will add code to pull a single input register from modsim32.exe. We are starting small and building up. We will create a function called **Poller()** beneath the loop function. The **Poller()** function will be called every time the loop executes, and within it, we will include the function **Mb.MbmRun()** as depicted in Figure 24.1.

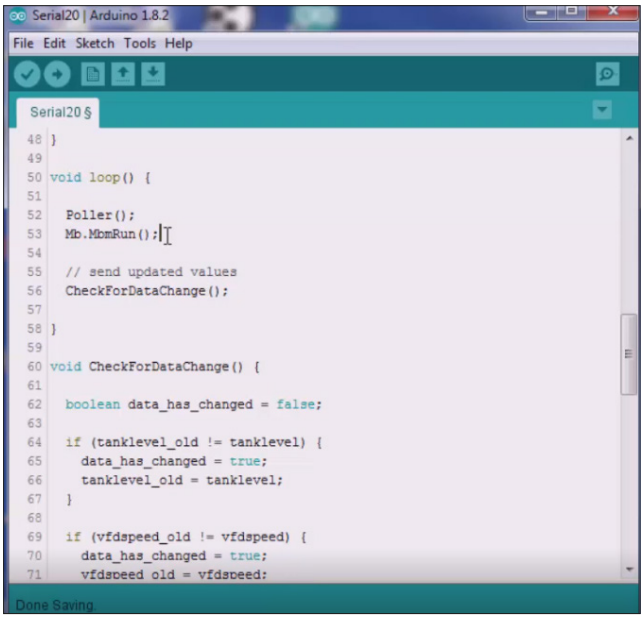


Figure 24.1: Adding **Poller()** *and* **Mb.MbmRun()** *in the loop section.*

So **MbmRun** is essentially the function that runs and implements Modbus TCP client services. Thus, this is the core of the system implemented by the library that we are using. Once this executes, Modbus TCP client will execute. What **Poller()** function will do is sort of set up what registers are to be pulled every time this loop runs. We will define this function, **Poller()**, then we are going to write some initial code and explain what it does afterwards. Similarly, the accumulator is incremented by one every time. If the accumulator equals 3000 and subsequently, if the accumulator equals 3200, a specific action happens. Following this action, the accumulator is set to zero, as illustrated in Figure 24.2.

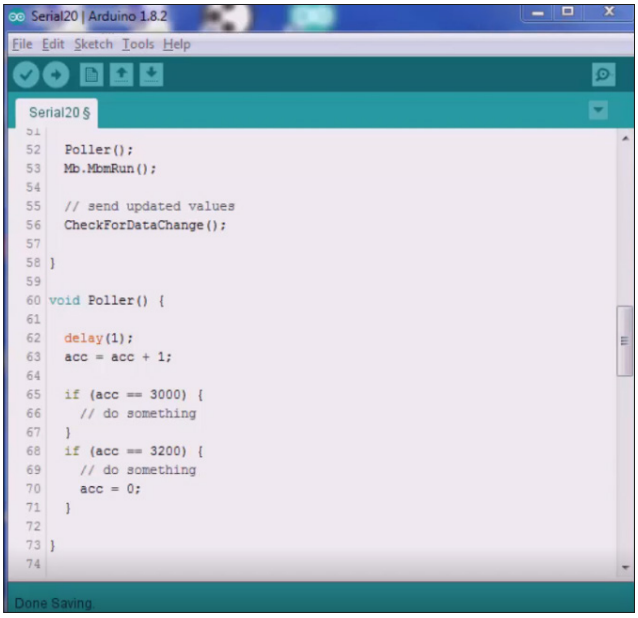


Figure 24.2: Defining the **Poller()** *function.*

Let me explain what this does. Remember, loop just executes as fast as it can. So, when it hits the **Poller()** function, it will execute this code and this delay actually stops execution for one millisecond, **delay(1)**, it stops the code and causes the processor to wait. So, every time this comes in here, there is a delay for one millisecond. We use an accumulator (acc) to record that delay. The accumulator is incremented by one each time, counting up to 3000. This piece of code checks if the accumulator equals 3000, then it does something. When it reaches 3200, it does something else and resets the accumulator, like resetting a timer. This is how we implement a three-second poll, doing something every three seconds.

What we must do this way is because the way that this library is implemented is not the most efficient. The MbmRun function needs to run as quickly as possible, and we are introducing one-millisecond delays in between. So, we have to make adjustments here and there. What does that "something" look like? Well, in this case, the "something" is represented by the statement Mb.Req(MB_FC_READ_INPUT_REGISTER, 100, 1, 0) with acc = 3000. Let us break down what this is doing. This statement is actually defined within the library, and we'll show you where it is defined. This means it is a command to read an input register. The value after, 100, represents the register to read. It will be an input register, such as 30,000, 30,001, 30,002, and so on. You do not need to put the 30,000; just register 100 is sufficient, representing 30,100. The next value, 1, indicates the number of registers to read, and the zero is a default, defining the area in which the return data is stored.

Here is how it works: when the accumulator hits 3000 after three seconds, this command is set up to request reading an input register. Then, MbmRun executes it, and it must wait for the TCP server to send a response. Between this point and there are 200 milliseconds

(0.2 seconds). After that wait, it checks to see what data is received, and the data is stored in the array MbData. The first element of this array, MbData[0], is the data received. Since we put zero there in Mb.Req, it corresponds to the first element, MbData[0]. Therefore, MbData serves as our return value, and we store it in the variable tanklevel, as shown in Figure 24.3.

Everything else happens from there. In the 'poll' function, we've created a sort of timer, and these are the commands responsible for reading input registers and retrieving the data, subsequently placing it in the designated variables. We know that it might seem a bit complex at first, but as we delve into more projects, you will become more accustomed to it. We will continue to expand this program in future sessions. For now, let us conclude here. In the next section, we will proceed to test this. This program, as it stands, is complete by itself. We will save it, verify it, compile it, upload it to the Arduino, and conduct tests.

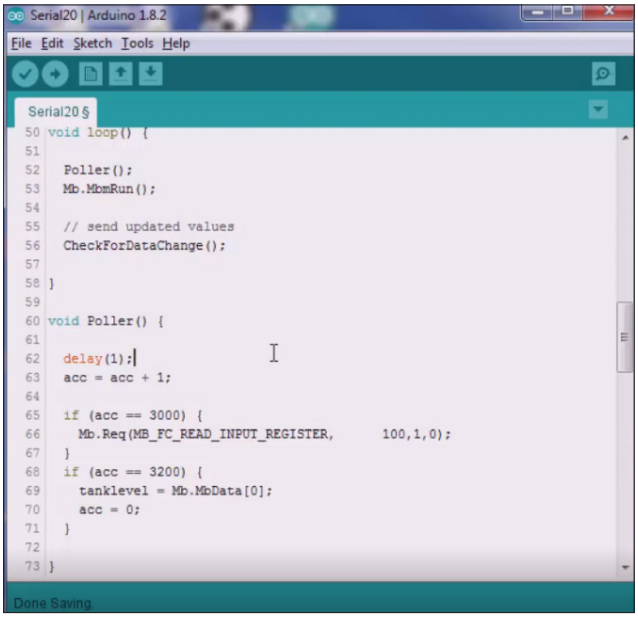


Figure 24.3: Completed **Poller()** *function.*

Chapter 25 • Testing the Modbus TCP Client Program

So, we have come to the best part, testing the code. Thus, what we are going to do is first set up modsim32.exe, which is our Modbus TCP server. We are going to create this one input register, address 101. We are using 101 instead of 100 because of the offset that modsim32.exe has. Modsim32.exe has the same sort of offset as modscan32.exe because it is made by the same manufacturers. Remember, in our code, we are reading register 100, so we are using 101 to account for the offset, as shown in Figure 25.1. Let's set the initial value of the tank level to 12, just for demonstration, as depicted in Figure 25.2. Now, with modsim32.exe set up let's start our serial monitor window and position it accordingly. We will then click "Connect" as a Modbus TCP server, as illustrated in Figure 25.3.

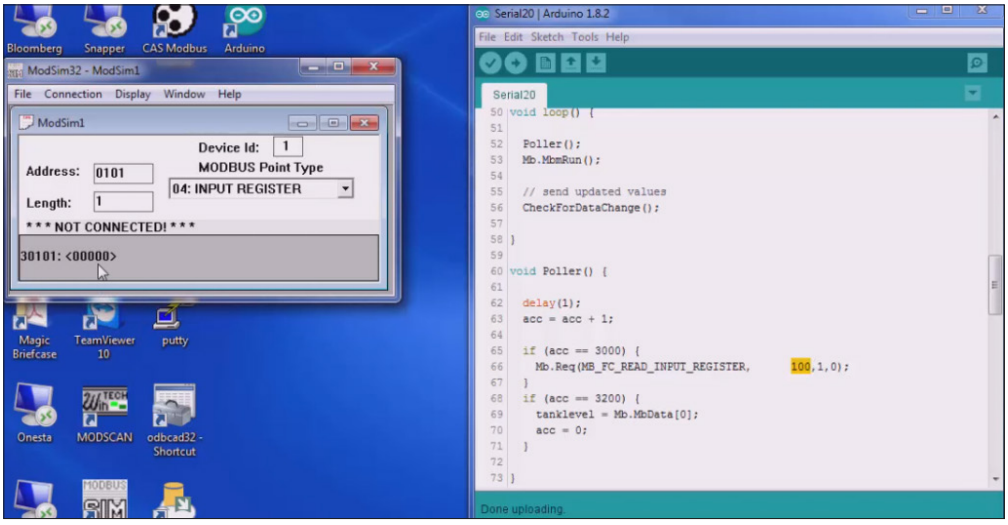


Figure 25.1: Setting input register in the modsim32.exe.

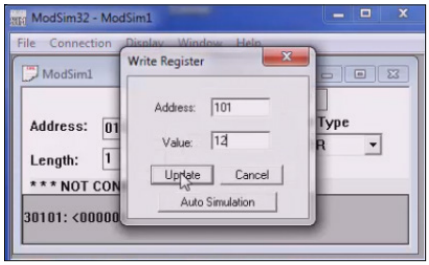


Figure 25.2: Updating the input register value.

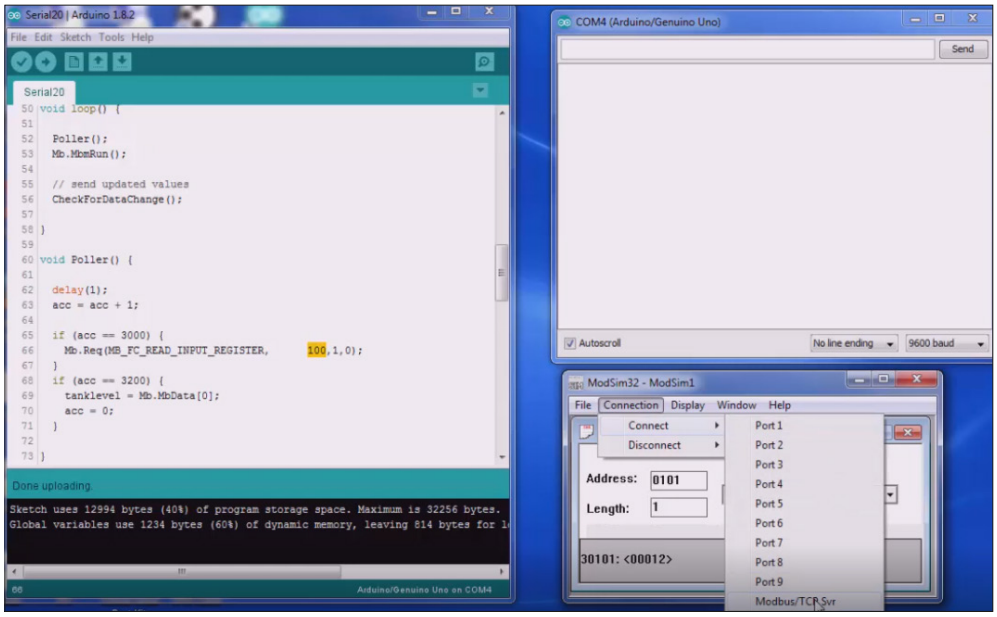


Figure 25.3: Selecting connect as Modbus TCP server.

Next, we are going to send our program to the Arduino by using the upload button. Therefore, it is compiling successfully, then it is uploading. We have the tank level, we have the VFD Speed, Valve and Pump just like last time, as depicted in Figure 25.4. So, we are going to change 12 ft to 8 ft and see what happens. There we go, it works. The tank level is now 8 ft as shown in Figure 25.5 and Figure 25.6, respectively.

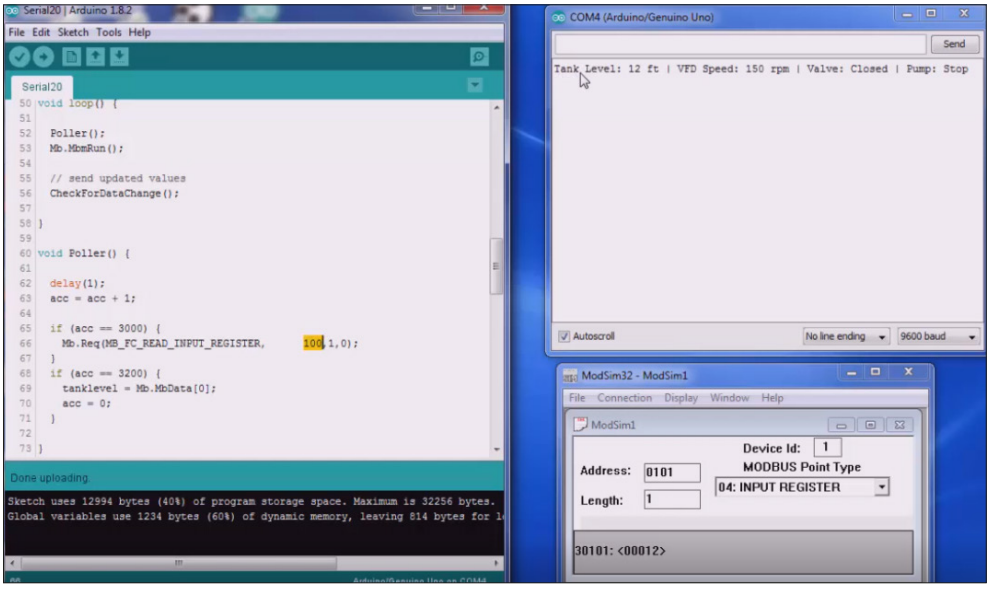


Figure 25.4: Register values just like last time.

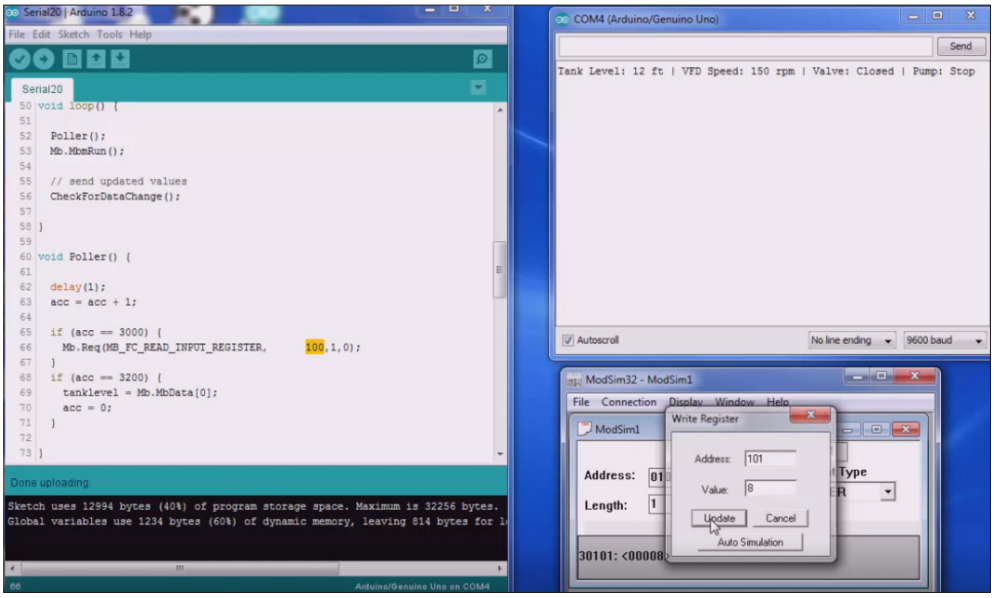


Figure 25.5: Updating value of the tank level.

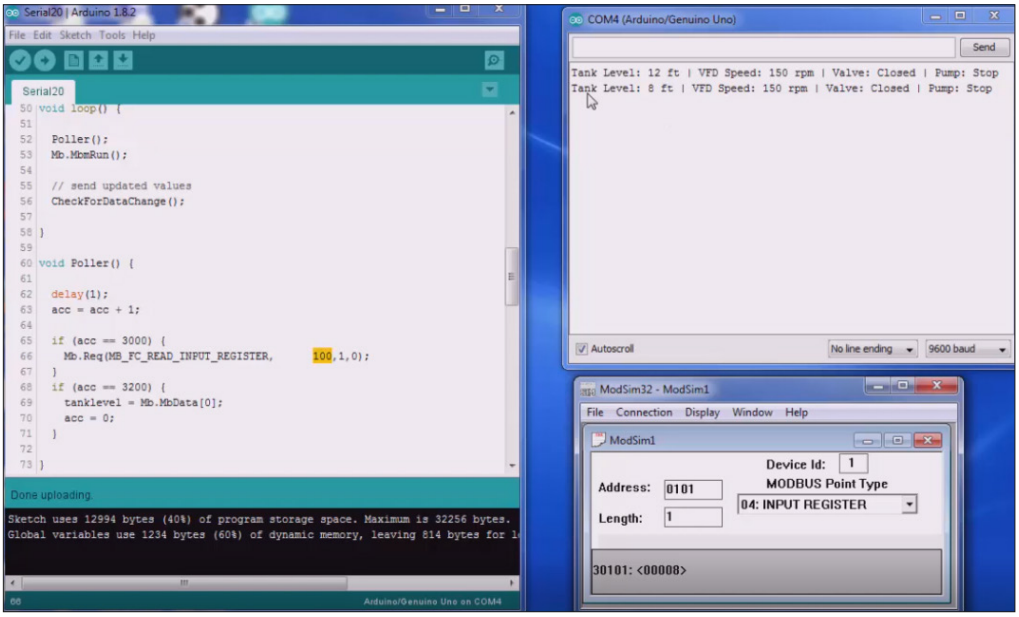


Figure 25.6: Changing value after updating the tank level value.

So, what just happened there? We changed the value to 8. Remember the three-second interval. **Poll**, the register got it into its tank level memory, recognized a change, and then displayed it on our serial monitor right there. If we change the tank level value to 10 it will change to 10, as illustrated in Figure 25.7 and Figure 25.8, respectively.

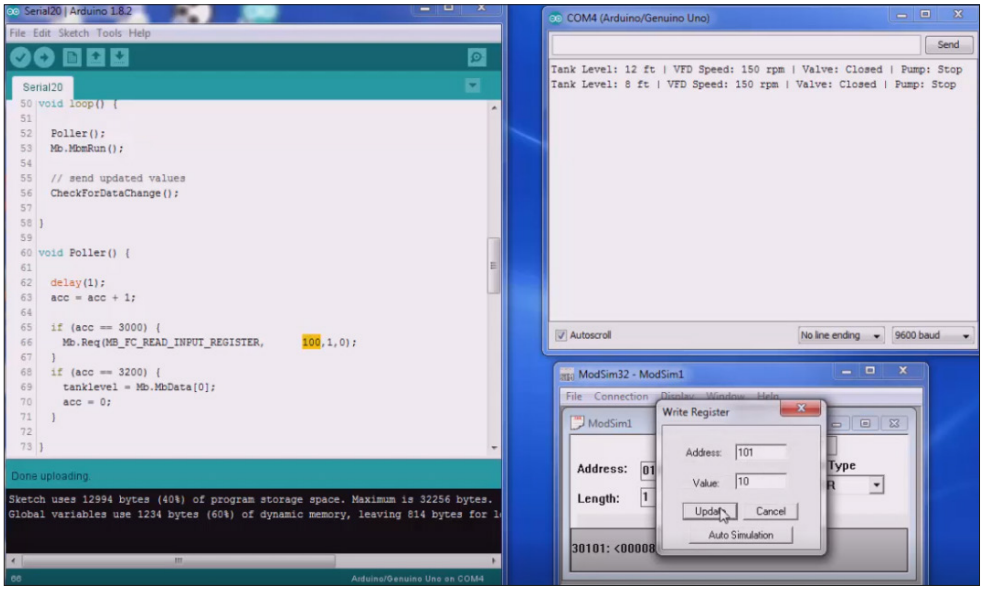


Figure 25.7: Updating value of the tank level.

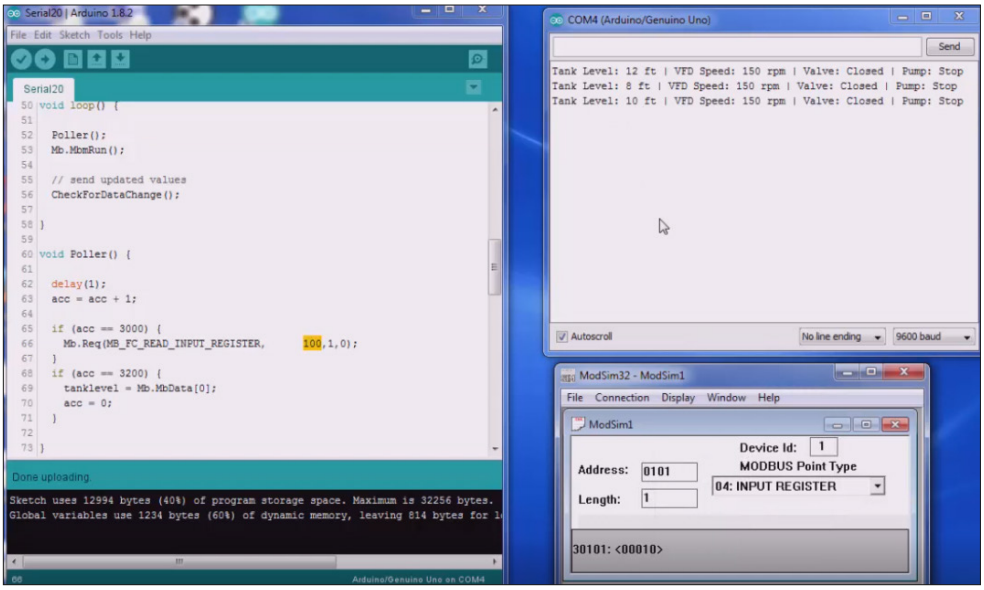


Figure 25.8: Changing value after updating the tank level value.

So, we are getting instant feedback, and we know our code is working. We can read an input register. Our Modbus TCP client is doing what it is supposed to do, and so, it works.

Chapter 26 • Writing Codes to Read the other Modbus Register Types

In the last section, we successfully tested this application that you are seeing here, " Serial20", and we read input register value from modsim32.exe. In this section, what we are going to expand the **Poller()** function to read the remaining registers: a holding register, an input register, and a coil register. This way, you would have an idea of how to handle all four types of memory locations.

We are going to add a little comment here, "read input register", and this would be for the tank level. After that, we will read the holding register, which is the vfdspeed. We are going to quickly cut and paste some code and explain what it does. It is very similar to the one before, as shown in Figure 26.1.

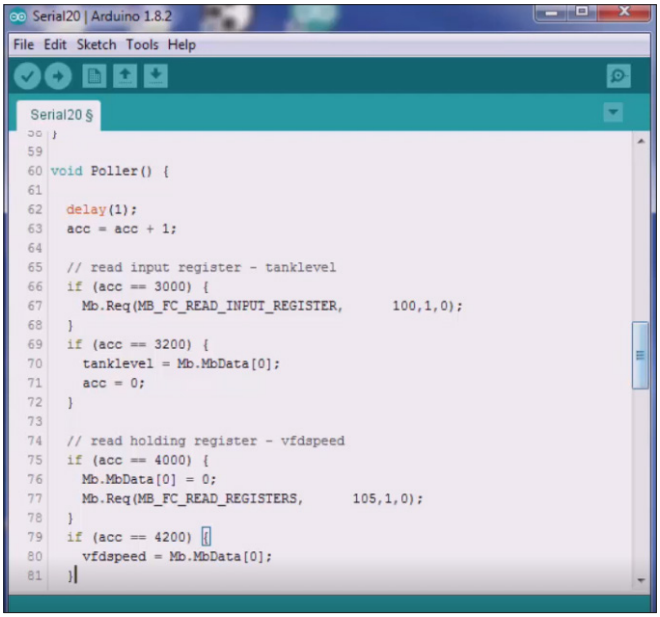


Figure 26.1: Adding codes to read holding register.

If acc is equal to 4000, at 3 seconds, it sends a message. At 3.2 seconds, it checks to see if the data is received. We should remove the **acc = 0** command in the read input register section. Then, at 4 seconds, it sends the **read holding register** command. Again, we will explain all these terms, MB_FC_READ_INPUT_REGISTER and MB_FC_READ_REGISTERS in a subsequent section. Right now, these are built-in commands in the library, and we are using address 105, keeping with the convention we were using before. So, this is really 40,105, and 0.2 seconds later, we get the vfdspeed, from MbData. Now we are going to read input status, which will be our valvepos. Since this is a status, which is true or false, we have to translate the data that we get back to true or false. Let us show you how we are going to do that.

At the 5-second mark, we set to zero our buffer that receives the data and we send a request to read the discrete input. We are going to make that as 110 and keeping the convention before. So, this will be like 10,110. 200 milliseconds later, which is 0.2 of a second later, we check for data. But in this case, if the MbData[0] is equal to 1, as in binary 1, that means it is true and valve position is true; else valve position is false. Thus, we just translate the data received, which is a binary 1 or 0, into true or false as depicted in Figure 26.2.

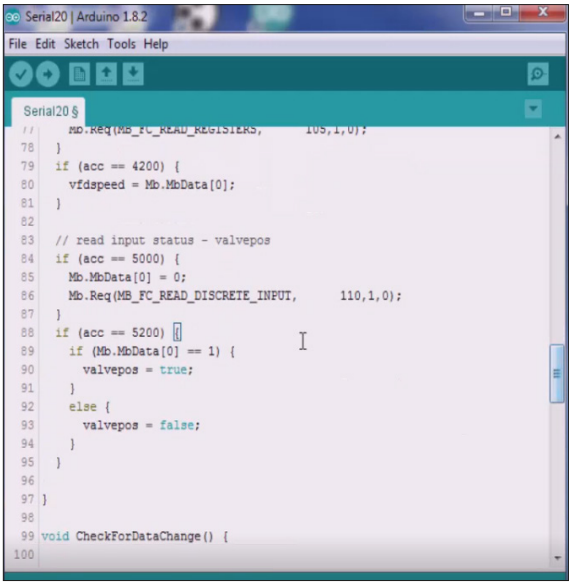


Figure 26.2: Adding codes to read input status.

Finally, we have **read coil**, which is our pump, and we are going to paste the codes in there. As you can see in Figure 26.3, it is very similar and we are using our convention. We are going up to 115 as the address of the register of the coil and we are up to the six second mark. 200 milliseconds later, we read and do the same thing; we test MbData. If it is one, pump is true, if not false. As this is the last read that we do in this cycle, we set acc back to zero. We wait three seconds again, before we restart the polling once again. That is essentially it. We have included now reads for the holding register and input status and a coil.

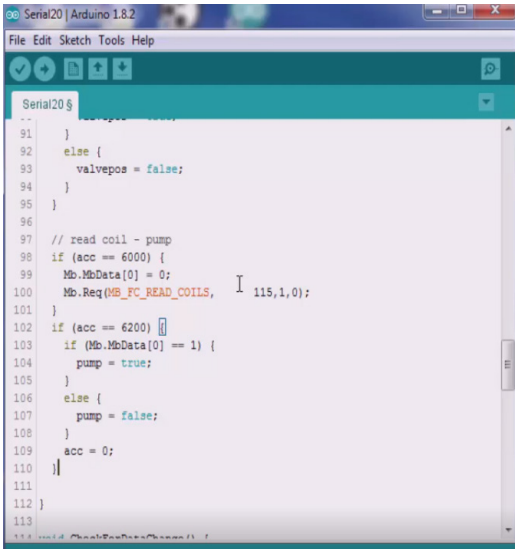


Figure 26.3: Adding codes to read coil.

In the next section, we are going to compile, upload and test them.

Chapter 27 • Testing the Modbus TCP Client Program

In this section we test the application of your program within a previous section. We have it open here and it has been saved as Serial21. What we are going to do now is execute the modsim32.exe. We set up the registers that are to be read. The first one is our input register, which is 12, the default value. We use default values from the code here and add them from "File → New Section". The next one is holding register, with address 105, so we input 106 due to the offset, setting the initial value at 150. That is our holding register. Next is our input status with its value one. We will make it 111 because of the offset and set it as true. So, it will start at one. When we are testing it, we can actually see the changes happening. Let's go finally to the coil, which was 115. It will be 116 because of the offset. Its default is false which is zero. Now we are going to "Connection → Connect Modbus TCP Server", as shown in Figure 27.1.

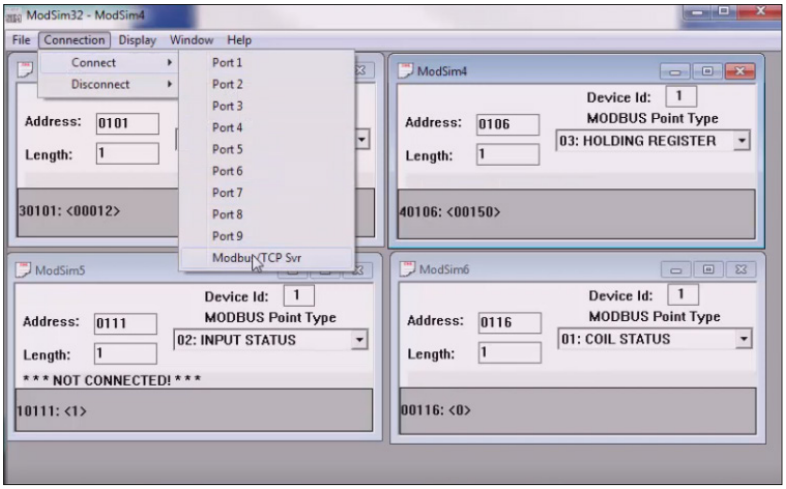


Figure 27.1: Choosing connection, connect Modbus TCP server.

Now that we have modsim32.exe set up and connected, with our registers ready for reading, the setup is complete. Next, we will upload the application to the Arduino. As shown in Figure 27.2, the download to the Arduino is successful.

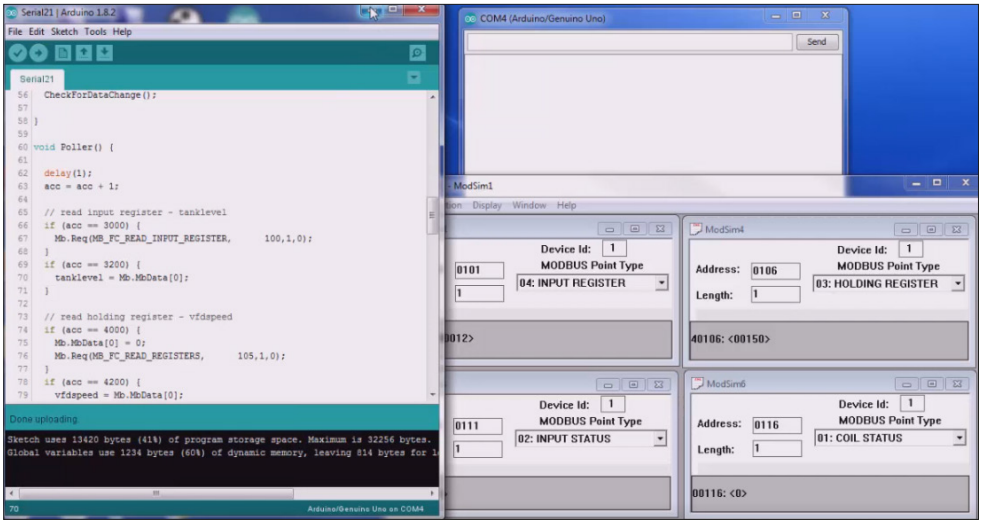


Figure 27.2: Uploading the application into the Arduino.

We have got the initial serial message right there, and we will change some values and see what happens. We change the input register value to 8 as illustrated in Figure 27.3.

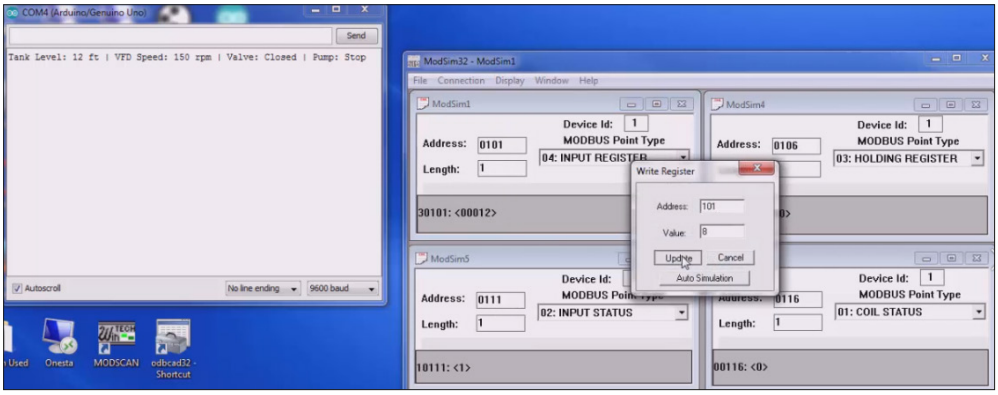


Figure 27.3: Changing the input register value to 8.

We should see a message there coming up and tank levels now is 8 feet as shown in Figure 27.4.

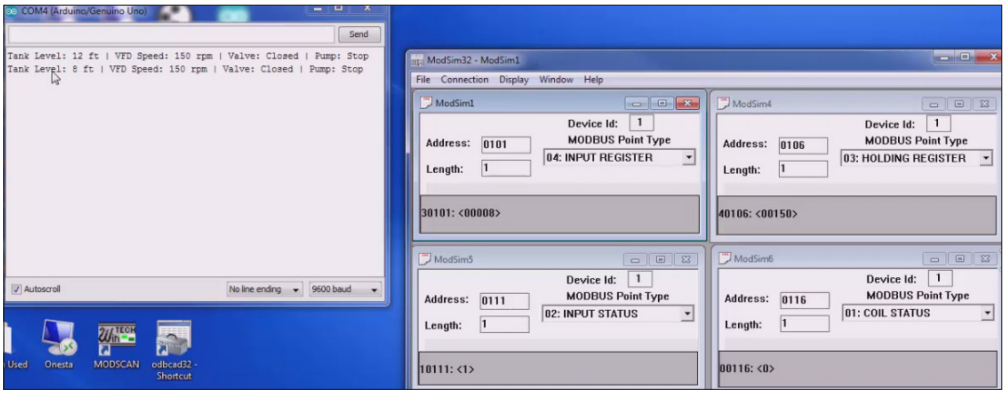


Figure 27.4: A new message in the serial window.

Let us change the rpm from 150 to 200. Go in the serial window as depicted in Figure 27.5 and Figure 27.6, respectively.

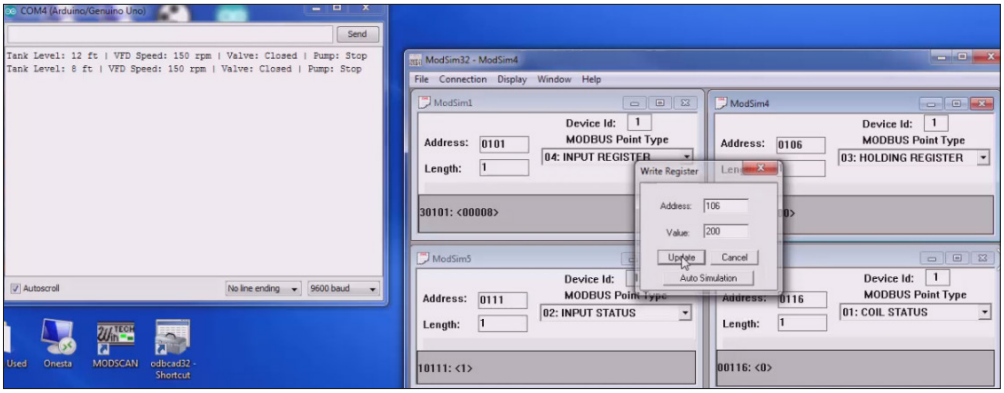


Figure 27.5: Changing the rpm from 150 to 200.

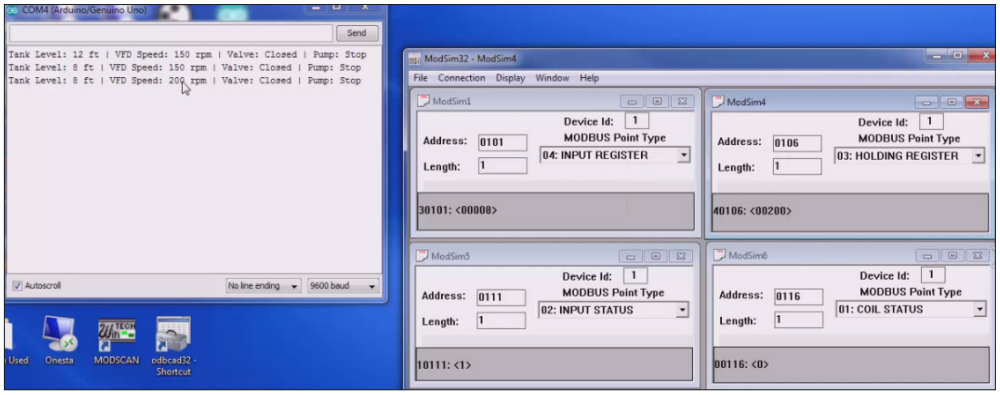


Figure 27.6: A new message in the serial window.

We will change the valve from closed to open, meaning one to zero. Valve is now open as illustrated in Figure 27.7 and Figure 27.8, respectively.

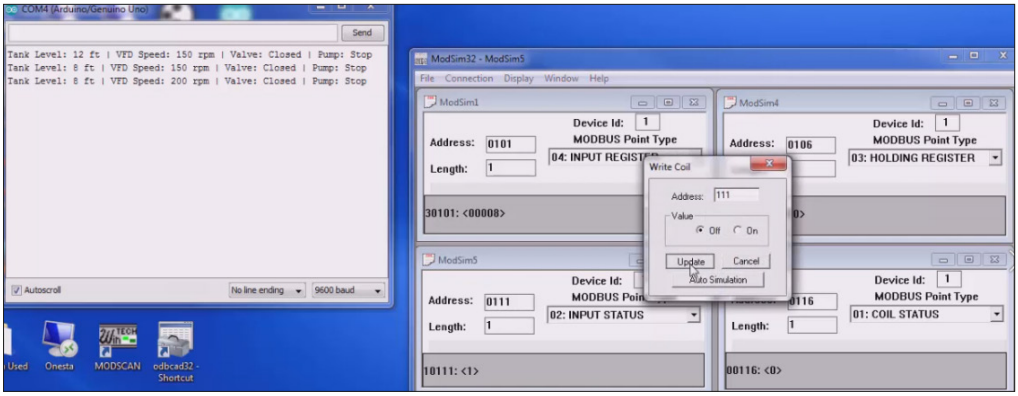


Figure 27.7: Changing the valve from closed to open.

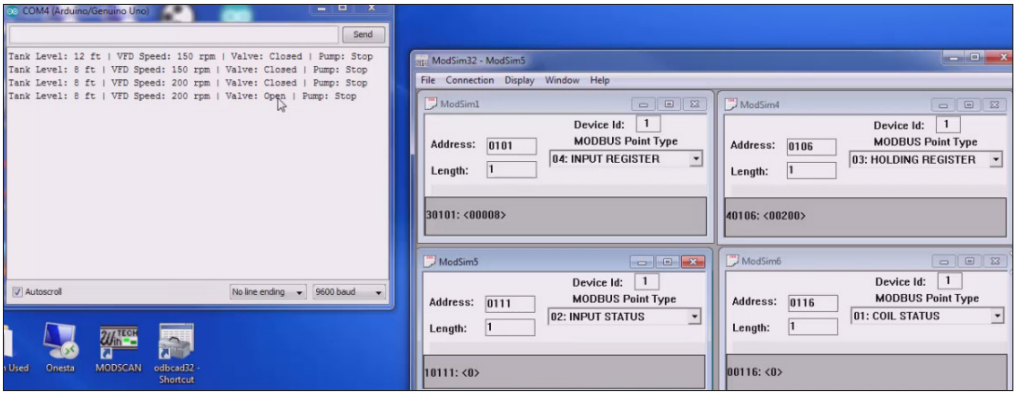


Figure 27.8: A new message in the serial window.

Let us change the coil status, transitioning the pump from stop to run, and there we go—now in the run mode as shown in Figure 27.9 and Figure 27.10 respectively. Thus, we have confirmed that our Modbus TCP server is successfully reading all four registers, including input register, holding register, input status, and coil status. With this accomplishment, our Modbus TCP client is now capable of interfacing with all four types of registers implemented in the Arduino. You can then take this code and use it in your own applications. You can read more registers, more certain types of registers. You have the core base code that you can use.

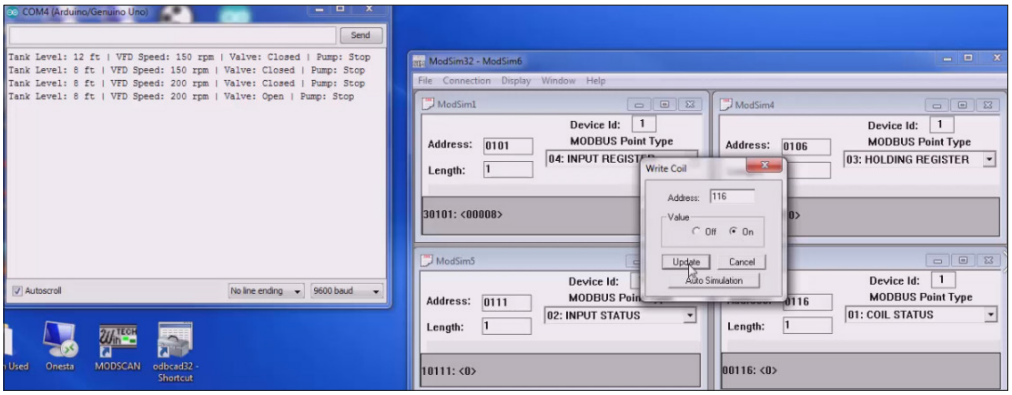


Figure 27.9: Changing the coil status, the pump from stop to run.

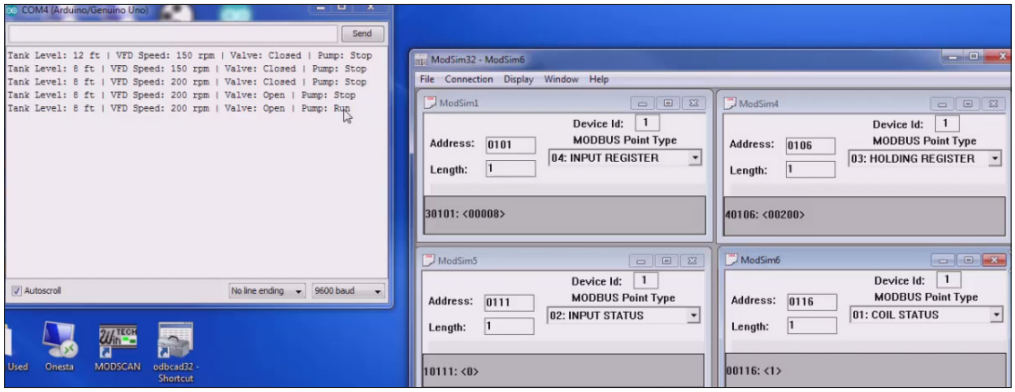


Figure 27.10: A new message in the serial window.

Chapter 28 • The TCP/IP communication between two Arduino Uno boards

In this section, we showcase a project that illustrates how Arduino UNO boards can establish a connection and communicate using the TCP/IP protocol. This project highlights the capability of Arduino boards to communicate seamlessly over an Ethernet network. One Arduino board is configured as a client, while the other operates as a server. The client board transmits commands to the server board, instructing it to toggle the LED state between on and off. The server board responds accordingly to the received commands. This project is a great opportunity to gain experience about network communication and explore the capabilities of Arduino boards. An overview and block diagram of the project are shown in the Figure 28.1.

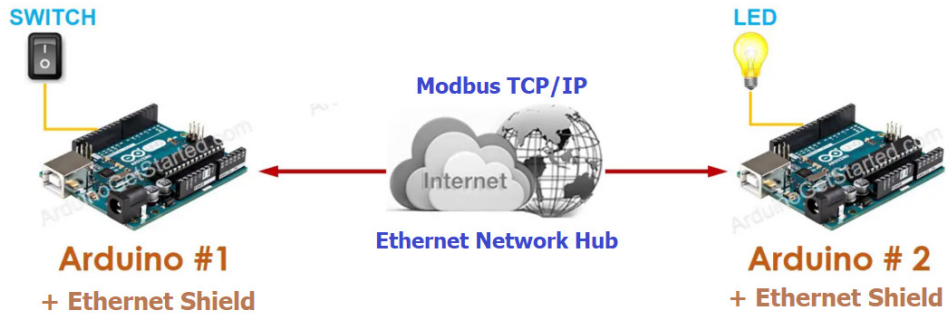


Figure 28.1: An overview and block diagram of the project.

The Arduino codes for Arduino board #1 and Arduino board #2 are as follows.

```
Arduino Code for Arduino #1:
#include <ezButton.h>
#include <SPI.h>
#include <Ethernet.h>
const int BUTTON_PIN = 7;
const int serverPort = 4080;
ezButton button(BUTTON_PIN);  // create ezButton that attach to pin 7;
byte mac[] = {0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x03};
IPAddress serverAddress(192, 168, 0, 180);
EthernetClient TCPclient;
void setup() {
  Serial.begin(9600);
  button.setDebounceTime(50); // set debounce time to 50 milliseconds
  Serial.println("ARDUINO #1: TCP CLIENT + A BUTTON/SWITCH");
  // Initialize Ethernet Shield:
  if (Ethernet.begin(mac) == 0)
    Serial.println("Failed to configure Ethernet using DHCP");
  // connect to TCP server (Arduino #2)
  if (TCPclient.connect(serverAddress, serverPort))
```

```
    Serial.println("Connected to TCP server");
  else
    Serial.println("Failed to connect to TCP server");
}
void loop() {
  button.loop(); // MUST call the loop() function first
  if (!TCPclient.connected()) {
    Serial.println("Connection is disconnected");
    TCPclient.stop();
    // reconnect to TCP server (Arduino #2)
    if (TCPclient.connect(serverAddress, serverPort))
      Serial.println("Reconnected to TCP server");
    else
      Serial.println("Failed to reconnect to TCP server");
  }
  if (button.isPressed()) {
    TCPclient.write('1');
    TCPclient.flush();
    Serial.println("- The button is pressed,  sent command: 1");
  }
  if (button.isReleased()) {
    TCPclient.write('0');
    TCPclient.flush();
    Serial.println("- The button is released, sent command: 0");
  }
}

Arduino Code for Arduino #2:
#include <SPI.h>
#include <Ethernet.h>
const int LED_PIN = 7;
const int serverPort = 4080;
byte mac[] = {0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x02};
EthernetServer TCPserver(serverPort);
void setup() {
  Serial.begin(9600);
  pinMode(LED_PIN, OUTPUT);
  Serial.println("ARDUINO #2: TCP SERVER + AN LED");
  // Initialize Ethernet Shield:
  if (Ethernet.begin(mac) == 0)
    Serial.println("Failed to configure Ethernet using DHCP");
  // Print your local IP address:
  Serial.print("TCP Server IP address: ");
  Serial.println(Ethernet.localIP());
  Serial.println("-> Please update the serverAddress in Arduino #1 code");
  // Listening for a TCP client (from Arduino #1)
  TCPserver.begin();
```

```
}
void loop() {
  // Wait for a TCP client from Arduino #1:
  EthernetClient client = TCPserver.available();
  if (client) {
    // Read the command from the TCP client:
    char command = client.read();
    Serial.print("- Received command: ");
    Serial.println(command);
    if (command == '1')
      digitalWrite(LED_PIN, HIGH); // Turn LED on
    else if (command == '0')
      digitalWrite(LED_PIN, LOW);  // Turn LED off
    Ethernet.maintain();
  }
}
```

In summary, the first Arduino code monitors the button state and sends commands to the second Arduino over Ethernet. The second Arduino code listens for incoming connections and reads the commands received, controlling the LED based on the command. The first code uses the ezButton library and **TCPclient.write()** function, while the second code uses the EthernetServer class and **EthernetClient.read()** function. The LED state is controlled using the **digitalWrite()** function.

Chapter 29 • Modbus TCP/IP Temperature Control using WinCC SCADA

Currently, the use of Arduino in supervisory control and data acquisition (SCADA) systems has become popular for small-scale control and monitoring applications in various industries. There are multiple approaches to implement Arduino SCADA, and one method involves using the Modbus TCP/IP communication protocol. In this section, we will explore a liquid temperature control system that incorporates an Arduino board connected to WinCC SCADA via Modbus TCP/IP. In the following steps, we will guide you through the implementation of this project, providing you with insights on how to use Modbus protocol to create an Arduino-based SCADA system. To better understand the system, please refer to Figure 29.1, which depicts the schematic diagram.

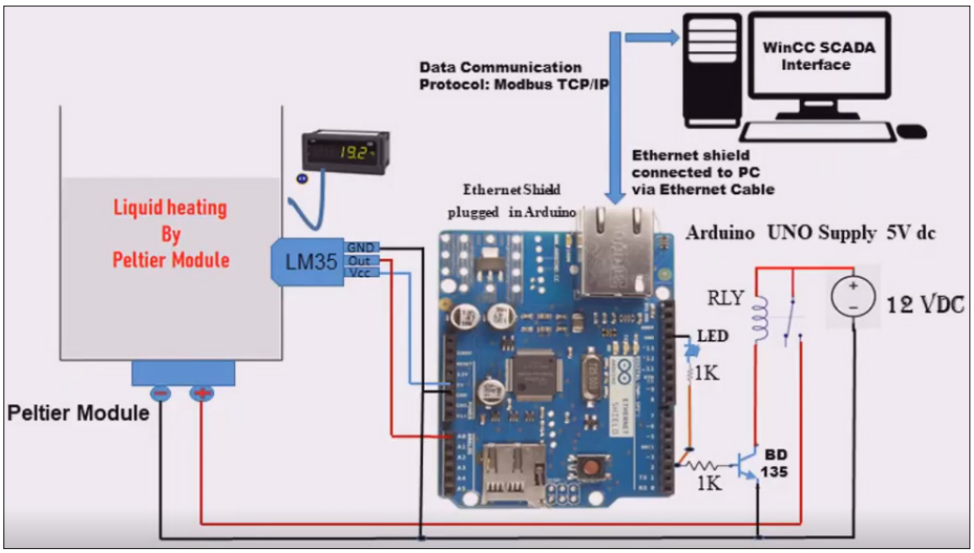


Figure 29.1: The schematic diagram for this project.

In this project, we will use a heating element, specifically a Peltier module, to maintain the temperature of a liquid. The temperature of the liquid will be kept within a range of 40 to 47 degrees Celsius. The Peltier module is attached to the bottom of a pot that contains the liquid. To achieve this, the Peltier module will be turned on by the Arduino's digital output when the liquid temperature falls below 40 degrees Celsius. Conversely, when the temperature rises above around 47 degrees, the Peltier module will be turned off. This process will continue until a stop command is received from the SCADA interface.

The LM35 temperature sensor is used to detect the temperature of the liquid, and an additional temperature indicator is used to locally monitor the temperature. To control and monitor this temperature control operation with the Arduino, a suitable Modbus Ethernet shield is plugged into the Arduino Uno microcontroller. The communication between the Arduino and the SCADA interface is established using the Modbus protocol. Two variants of Modbus protocol are utilized: Modbus RTU protocol is used for serial network communication,

and Modbus TCP/IP protocol is used for Ethernet network communication, as shown in Figure 29.2.

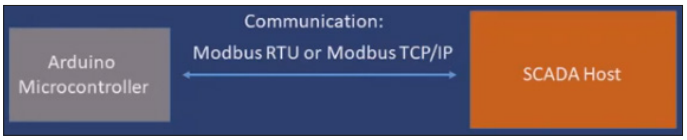


Figure 29.2: Communication for Arduino with the SCADA interface.

This project uses the Modbus TCP/IP protocol for communication between the Arduino and the SCADA interface. To establish this communication, an Ethernet shield is connected to the Arduino Uno microcontroller, and an Ethernet cable is used to connect the Arduino to the computer. The Arduino utilizes the Modbus protocol to transmit input/output data to the SCADA interface, so it is necessary to upload the Modbus library to the Arduino board to enable communication with the SCADA interface using the Modbus protocol.

Now, let's move on to the coding for the Arduino. If we examine the data access mechanism between the Arduino and the SCADA interface, we can observe that Modbus TCP/IP is used as the data communication protocol. In Modbus TCP/IP, data sharing between the Arduino and the SCADA client occurs on a client/server model. In this setup, the Arduino is configured as the server or slave, while the SCADA host acts as the client or master, as shown in Figure 29.3.

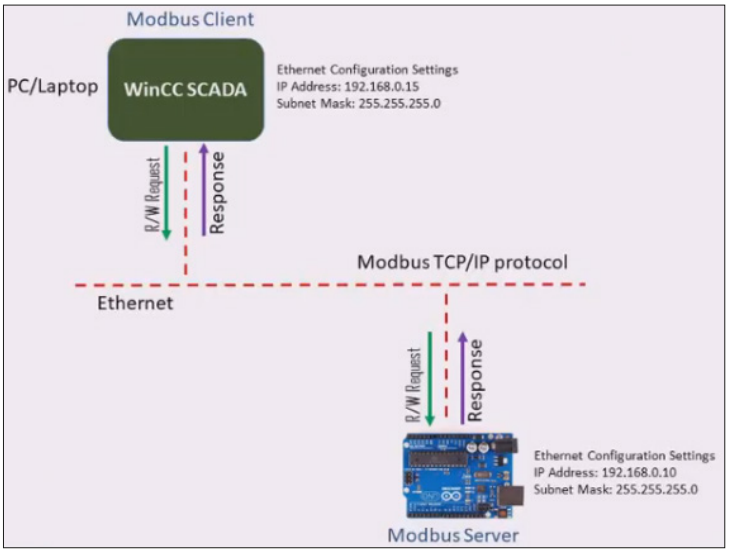


Figure 29.3: The client/server model in Modbus TCP/IP.

The Modbus TCP/IP protocol utilizes a standardized form of data transmission known as an application protocol, which operates on an Ethernet physical network. It is important to use an Ethernet crossover cable when physically connecting an Arduino to a PC. To begin, you will need to include two libraries in the Arduino IDE. To do this, navigate to the Sketch

menu, select "Include Library," and then choose "Add .zip Library" as shown in Figure 29.4. In the dialog box that appears, locate the desired library file on your desktop. There are two library files available in zip format within this folder. You can now add these two libraries to the Arduino IDE from the dialog box shown in Figure 29.5. These libraries can be downloaded from online resources.

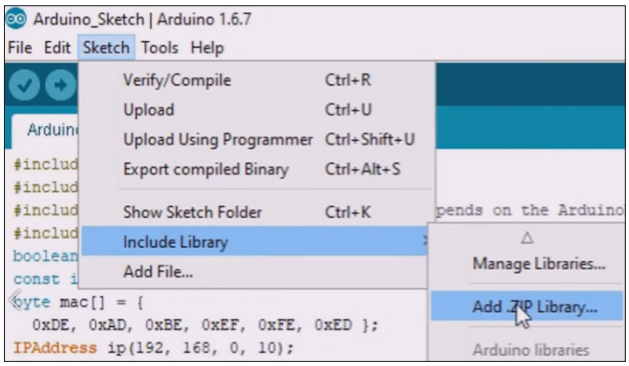


Figure 29.4: Adding zip library files.

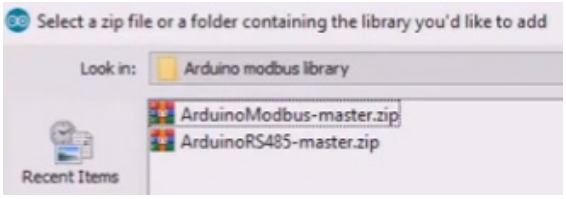


Figure 29.5: Adding two libraries into Arduino IDE.

If we proceed to look at the coding for Arduino, we will see that the coding has been started with a number of library headers as shown in Figure 29.6.

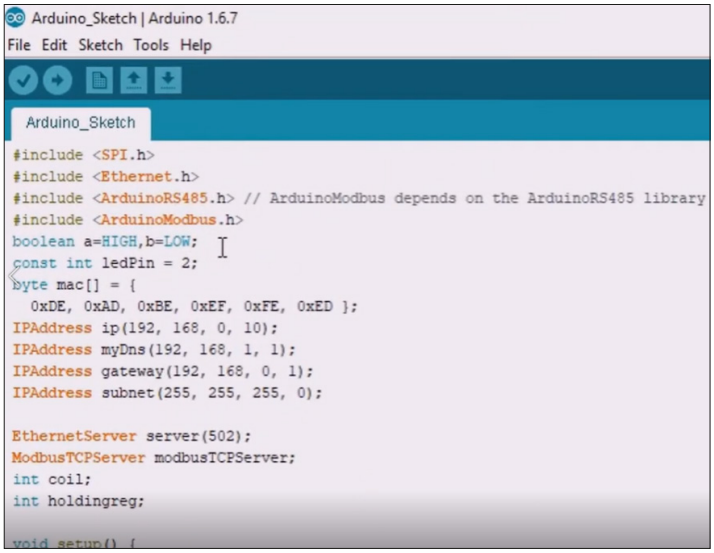


Figure 29.6: The coding for Arduino.

The first two libraries enable Arduino to work with the Ethernet shield, through which Arduino establishes a physical network with this kind of host. The other libraries implement the Modbus protocol in the Arduino input/output data transmission mechanism. In the next five lines of the coding, different network configuration settings for Ethernet network are available. In configuration settings, MAC and DNS and IP address must be assigned to the Ethernet shield. The rest are the settings you can keep optional.

The wide set of functions of this Arduino program initializes the Ethernet library with network configuration settings, and the Arduino is starting to communicate physically with the PC as depicted in Figure 29.7. As well as the physical network being established, the Modbus TCP server also initiated and becomes ready to connect with any SCADA program as a client. In the Modbus communication protocol, the data transfer functions offer some new registers to configure, monitor, and control the Arduino input/output devices from SCADA interface as illustrated in Figure 29.8. Modbus devices usually have a register map to register data. The Modbus data model has four basic data types, discrete inputs, coils, input registers and holding registers, as shown in Figure 29.9.

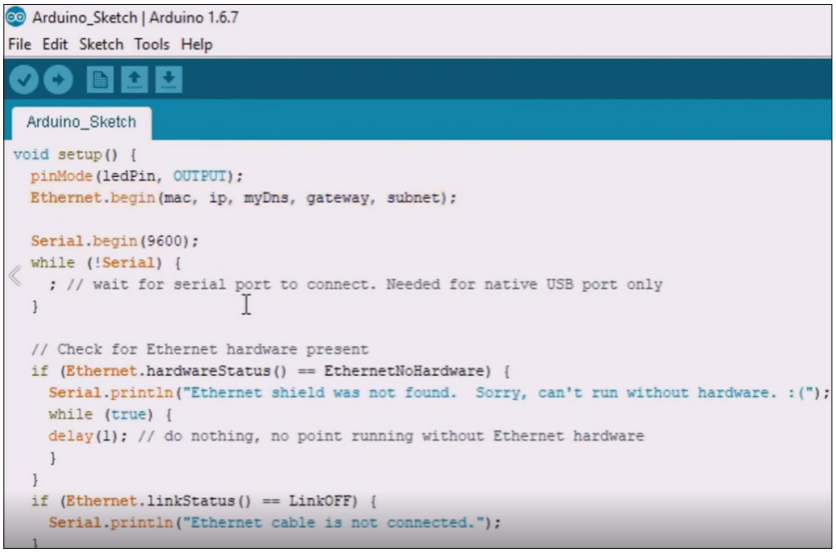


Figure 29.7: Ethernet library initialization with network configuration settings.

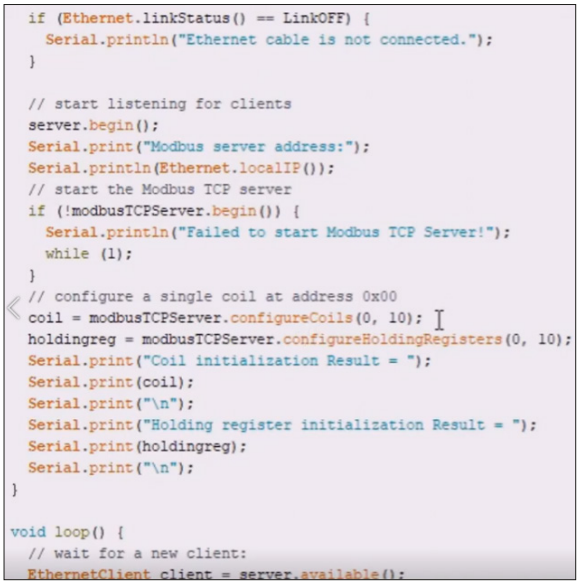


Figure 29.8: Configuration of Arduino monitoring and control.

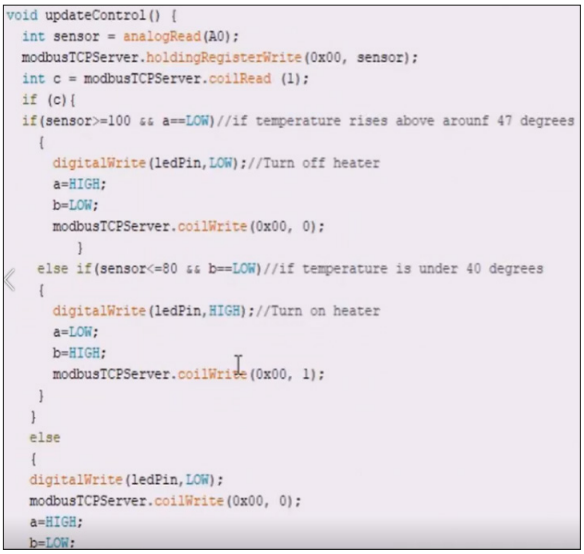
Object type	Access	Size	Address Range
Discrete Inputs	Read only provided by an I/O system	1 bit	0001 ~ 0999
Coils (Outputs)	Read-Write alterable by an application program	1 bit	10001 ~ 19999
Input Registers (Input Data)	Read only provided by an I/O system	2 byte	30001 ~ 39999
Holding Registers (Output Data)	Read-Write alterable by an application program	2 byte	40001 ~ 49999

Figure 29.9: Basic data types of the Modbus data model.

We have created holding registers in the Arduino program to store temperature sensor values and coil registers to store digital output data. In the void loop function, the actual operational functionality of the Arduino is implemented. This function sends data to a SCADA client for real-time temperature control monitoring and control. The void loop function also includes the "**updateControl**" function, which is responsible for the desired temperature control operation. This function uses one holding register address to transfer temperature sensor data and two coil register addresses to read from address 1 and write Arduino digital output to address 0, as shown in Figure 29.10 and Figure 29.11. The complete Arduino sketch code for this project is given below.

```
void loop() {
  // wait for a new client:
  EthernetClient client = server.available();

  // when the client sends the first byte, say hello:
  if (client) {
    // a new client connected
    Serial.println("new client");

    // let the Modbus TCP accept the connection
    modbusTCPServer.accept(client);

    while (client.connected()) {
      // poll for Modbus TCP requests, while client connected
      modbusTCPServer.poll();
      updateControl();
      }
    Serial.println("client disconnected");
  }

  }

void updateControl() {
  int sensor = analogRead(A0);
  modbusTCPServer.holdingRegisterWrite(0x00, sensor);
  int c = modbusTCPServer.coilRead (1);
  if (c){
```

Figure 29.10: The void loop function.



```
void updateControl() {
  int sensor = analogRead(A0);
  modbusTCPServer.holdingRegisterWrite(0x00, sensor);
  int c = modbusTCPServer.coilRead (1);
  if (c){
  if(sensor>=100 && a==LOW)//if temperature rises above arounf 47 degrees
    {
      digitalWrite(ledPin,LOW);//Turn off heater
      a=HIGH;
      b=LOW;
      modbusTCPServer.coilWrite(0x00, 0);
        }
    else if(sensor<=80 && b==LOW)//if temperature is under 40 degrees
    {
      digitalWrite(ledPin,HIGH);//Turn on heater
      a=LOW;
      b=HIGH;
      modbusTCPServer.coilWrite(0x00, 1);
    }
   }
   else
   {
   digitalWrite(ledPin,LOW);
   modbusTCPServer.coilWrite(0x00, 0);
   a=HIGH;
   b=LOW;
```

Figure 29.11: The **updateControl** *function.*

```
#include <SPI.h>
#include <Ethernet.h>
#include <ArduinoRS485.h> // ArduinoModbus depends on the ArduinoRS485 library
#include <ArduinoModbus.h>
boolean a=HIGH,b=LOW;
const int ledPin = 2;
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 10);
IPAddress myDns(192, 168, 1, 1);
IPAddress gateway(192, 168, 0, 1);
IPAddress subnet(255, 255, 255, 0);
EthernetServer server(502);
ModbusTCPServer modbusTCPServer;
int coil;
int holdingreg;
void setup() {
  pinMode(ledPin, OUTPUT);
  Ethernet.begin(mac, ip, myDns, gateway, subnet);
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  // Check for Ethernet hardware present
  if (Ethernet.hardwareStatus() == EthernetNoHardware) {
    Serial.println("Ethernet shield was not found.  Sorry, can't run without
hardware. :(");
```

```
    while (true) {
    delay(1); // do nothing, no point running without Ethernet hardware
    }
  }
  if (Ethernet.linkStatus() == LinkOFF) {
    Serial.println("Ethernet cable is not connected.");
  }
  // start listening for clients
  server.begin();
  Serial.print("Modbus server address:");
  Serial.println(Ethernet.localIP());
  // start the Modbus TCP server
  if (!modbusTCPServer.begin()) {
    Serial.println("Failed to start Modbus TCP Server!");
    while (1);
  }
  // configure a single coil at address 0x00
  coil = modbusTCPServer.configureCoils(0, 10);
  holdingreg = modbusTCPServer.configureHoldingRegisters(0, 10);
  Serial.print("Coil initialization Result = ");
  Serial.print(coil);
  Serial.print("\n");
  Serial.print("Holding register initialization Result = ");
  Serial.print(holdingreg);
  Serial.print("\n");
}
void loop() {
  // wait for a new client:
  EthernetClient client = server.available();
  // when the client sends the first byte, say hello:
  if (client) {
    // a new client connected
    Serial.println("new client");
    // let the Modbus TCP accept the connection
    modbusTCPServer.accept(client);
    while (client.connected()) {
      // poll for Modbus TCP requests, while client connected
      modbusTCPServer.poll();
      updateControl();
      }
    Serial.println("client disconnected");
  }
  }
void updateControl() {
  int sensor = analogRead(A0);
  modbusTCPServer.holdingRegisterWrite(0x00, sensor);
```

```
int c = modbusTCPServer.coilRead (1);
if (c){
if(sensor>=100 && a==LOW)//if temperature rises above around 47 degrees
  {
    digitalWrite(ledPin,LOW);//Turn off heater
    a=HIGH;
    b=LOW;
    modbusTCPServer.coilWrite(0x00, 0);
       }
  else if(sensor<=80 && b==LOW)//if temperature is under 40 degrees
  {
    digitalWrite(ledPin,HIGH);//Turn on heater
    a=LOW;
    b=HIGH;
    modbusTCPServer.coilWrite(0x00, 1);
  }
 }
 else
 {
 digitalWrite(ledPin,LOW);
 modbusTCPServer.coilWrite(0x00, 0);
 a=HIGH;
 b=LOW;
 }
}
```

To create PC-based visualization for monitoring and controlling a liquid temperature control system, a SCADA project has been created using WinCC SCADA, which is integrated in TIA Portal software. WinCC SCADA supports interfacing Modbus TCP/IP data from any Modbus device. Already, a complete SCADA project has been created using TIA Portal software, and in the next section, we will describe how to create it step by step. You can download the TIA Portal software from the Google Drive link provided on the following webpage:

https://plc247.com/download-tia-portal-v16-full-googledrive

We will now provide a brief overview of this project. By clicking on the project view, we can open the project. If we examine the data interface technique used between Arduino and WinCC SCADA, we will notice that the WinCC SCADA project uses variables to obtain input/output values from Arduino. These variables are referred to as text in the WinCC SCADA environment. To transmit data from Arduino to WinCC SCADA using this text, a connection must be established in the WinCC project. This connection requires the configuration of Modbus TCP/IP communication settings. Therefore, two steps must be taken to configure the communication settings in this SCADA project. The first step involves creating and configuring a connection, while the second step involves creating and configuring multiple tags. To access the navigation menu on the left side, double click on "Connections", as displayed in Figure 29.12.

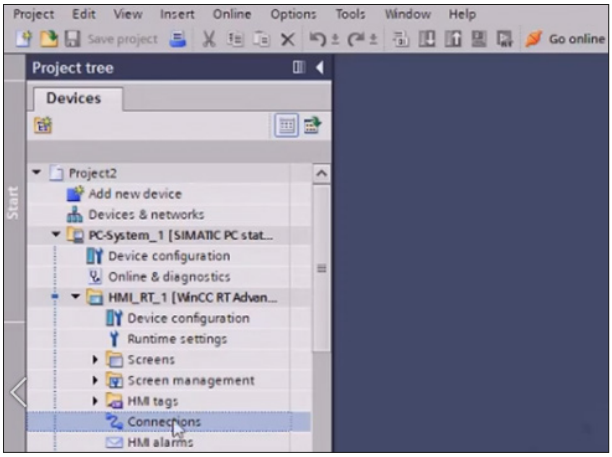


Figure 29.12: Double clicking on "Connections".

On the right side, you will see that a connection for Arduino Uno has already been created with the required configuration. Here the configuration for connection has been made on some settings, as depicted in Figure 29.13. Primarily, select the communication driver, which is Modbus TCP/IP. At the bottom, to make settings for connection parameters, you have to fill up field for CPU type, IP address for Arduino controller and the port address. The IP address and port address you have to provide there depend on the network configuration settings for Arduino, which we provided in the Arduino sketch as illustrated inside the red rectangle in Figure 29.13.

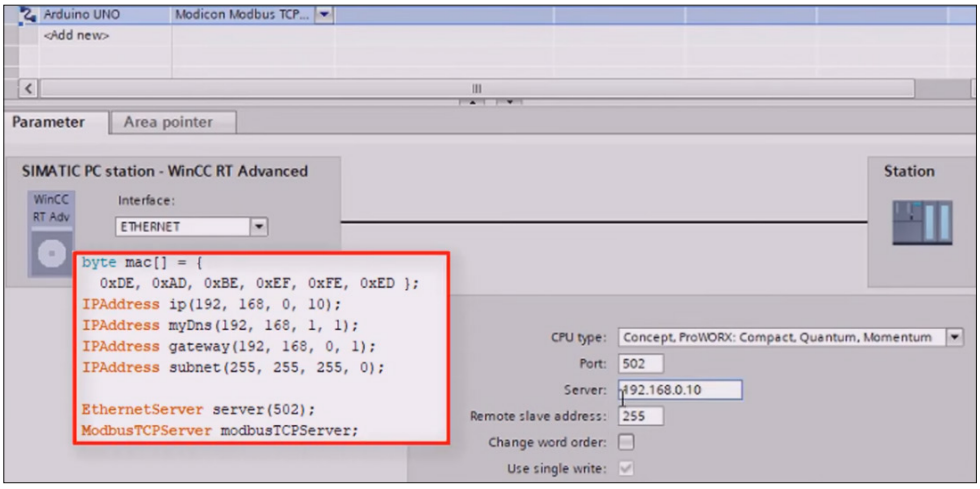


Figure 29.13: Setting the connections parameters.

Now, look for the created tags and their configuration settings. To get these created tags, double click on "Show all tags" under HMI tags as shown in Figure 29.14.

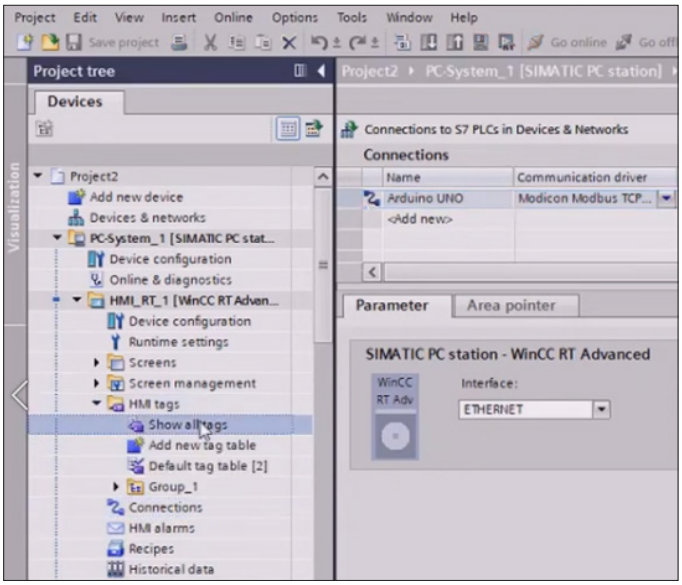


Figure 29.14: Double clicking on "Show all tags".

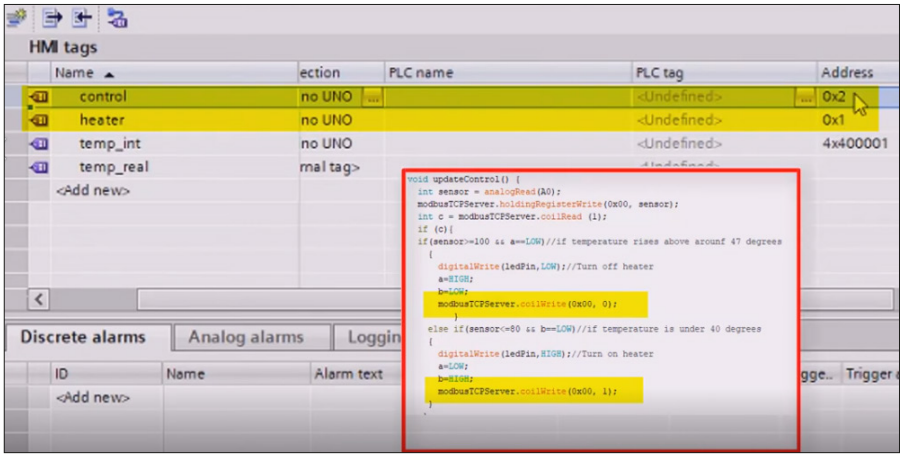


Figure 29.15: The coil register addresses for first two tags.

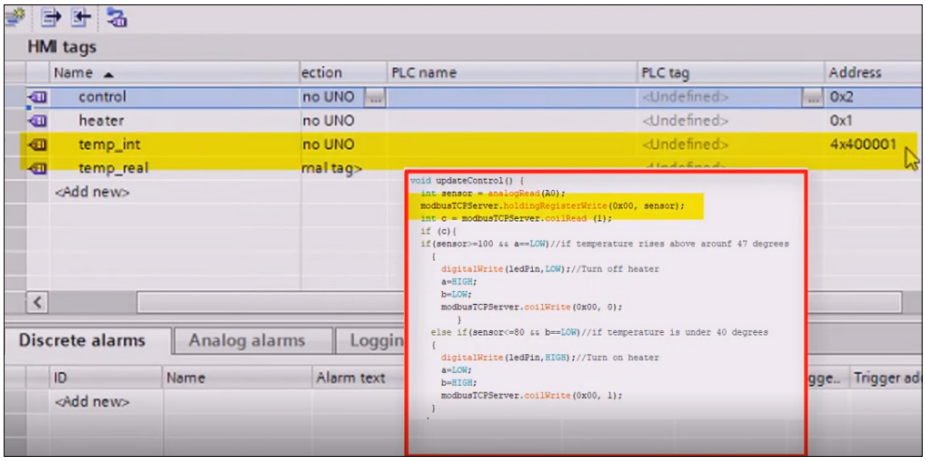


Figure 29.16: The holding register addresses for temperature data.

A few newly created tags have appeared on the right side. If we go for tags configuration, on important settings, we must set addresses for these tags on data communication between the Arduino and SCADA. Those are our Modbus addresses and are used as source for the Arduino input/output data, which have been created in Arduino sketch. The SCADA project gets the required data from these Modbus addresses using these tags. So, you must show these Modbus addresses when you create and configure tags. The addresses for the first two tags are coil registers for handling digital input/output data. The address for next tag is for the holding register, which contains the temperature data as depicted in Figure 29.15 and Figure 29.16, respectively. Now, go to the user interface that has been created to visualize the temperature control system as illustrated in Figure 29.17.

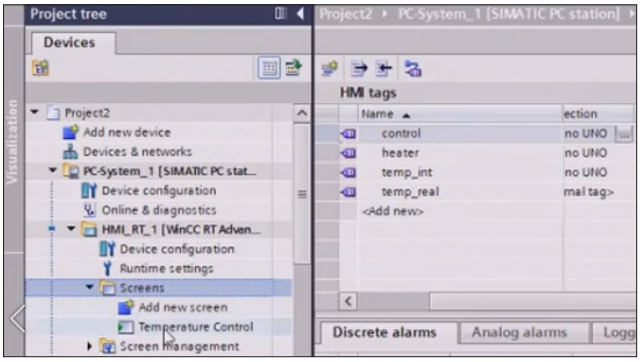


Figure 29.17: Visualizing the temperature control system.

Some functional graphical options have been created in this user interface. Through this we can start or restart the system, monitor the functional status real time, and look at the temperature value as shown in Figure 29.18.

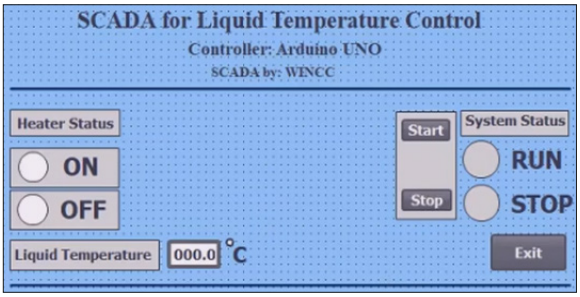


Figure 29.18: Functional graphical options in the user interface.

To make these objects functional, the created tags have been assigned to or linked to these functional objects. When you finish creating your user interface with required graphical options, and connect these objects with those tags, then it will be ready to run the user interface. In this section, we have tried to explain the configuration technique for the SCADA project. If you need this SCADA project, it will be described in detail in the next section. Now, it is the time to run it. When this SCADA project will be running on the PC, you will observe a functional process for temperature control in real time from that SCADA interface. To run this SCADA project, click on the "Run" icon as shown in Figure 29.19. By clicking on the start button the heater is on as depicted in Figure 29.20.

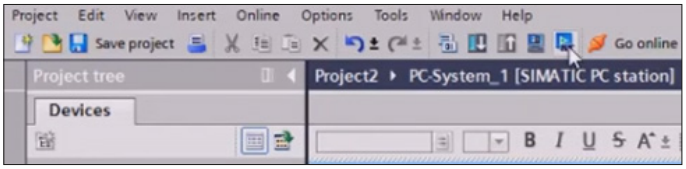


Figure 29.19: Clicking on the "Run" icon.

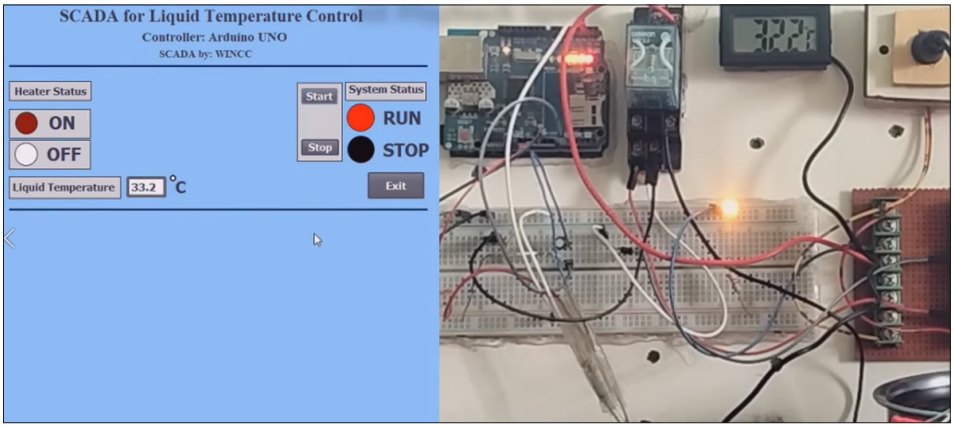


Figure 29.20: Clicking on "Start" button to make the heater on.

We hope this section will help you understand what the Modbus TCP/IP protocol is, how this communication protocol works, and how it can be applied to Arduino Uno microcontroller to build an Arduino-based SCADA system.

Chapter 30 • Creating SCADA Project with WinCC

WinCC is a supervisory control and data acquisition (SCADA) system developed by Siemens. It is used for monitoring and controlling industrial processes in various industries, such as manufacturing, energy, and pharmaceuticals. WinCC facilitates real-time data acquisition, visualization, and data analysis for plant operations. The software allows operators to monitor processes, make adjustments, and receive alarms and notifications. It also offers features like historical data logging, trend analysis, and reporting. You can download the TIA Portal software from the Google Drive link provided on the following webpage:

https://plc247.com/download-tia-portal-v16-full-googledrive

WinCC is highly customizable and scalable, enabling it to handle both small and large industrial systems. To begin creating a SCADA project with WinCC we click on the TIA Portal software and run it as administrator as shown in Figure 30.1.

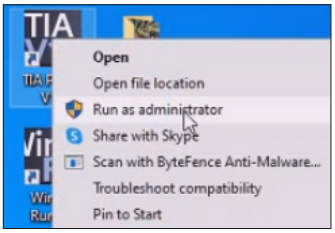


Figure 30.1: Running TIA Portal as administrator.

We create a new project and name it Temp_SCADA as shown in the Figure 30.2

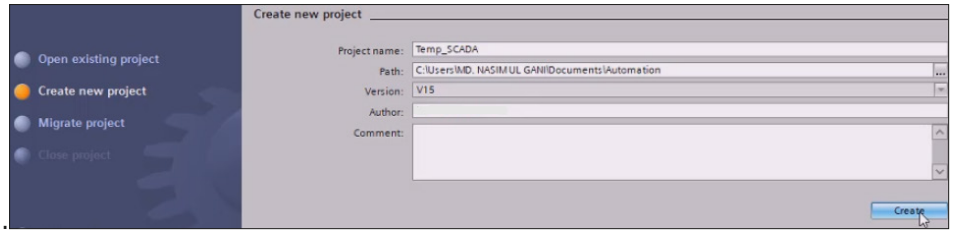


Figure 30.2: Creating a new project.

To begin, we open the window and proceed to select the project view. Within the Devices tab, we choose to add a new device. From the options available, we specifically select the PC systems icon and then the SIMATIC HMI application. From there, we choose the "WinCC RT Advanced" option and confirm our selection by clicking the "OK" button, as shown in Figure 30.3.

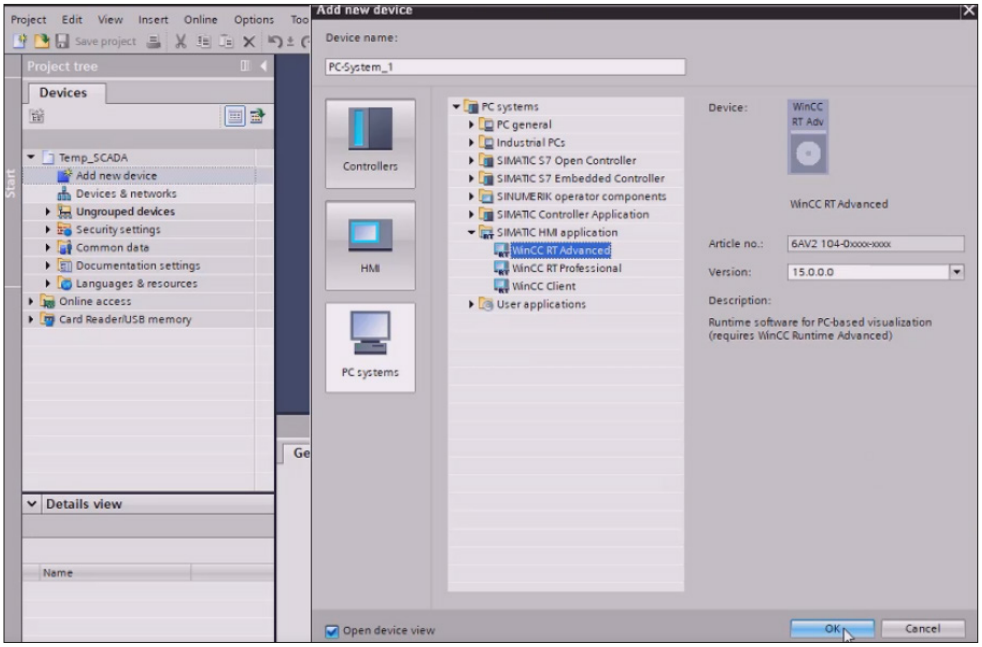


Figure 30.3: Adding "WinCC RT Advanced" icon.

Then, as you see in Figure 30.4 we add the "IE general" in the "SIMATIC PC station" slot.

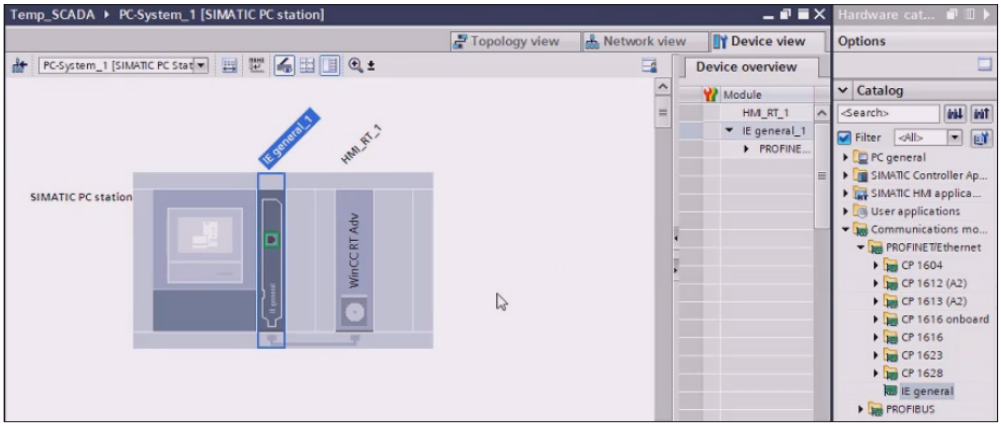


Figure 30.4: Adding the IE general in the SIMATIC PC station slot.

Within the HMI_RT_1 section located on the left side; we proceed to select the "Connections" icon. In the corresponding "Connections" window located on the right side, we proceed to choose the option under the "Name" tab and specify the "Communication driver" as Modicon Modbus TCP. For the "Parameter" tab, we specify the "CPU type" as Concept or ProWORX, the "Port" as 502, the "Server" as 192.168.0.10, and the "Remote slave address" as 1. This configuration is shown in Figure 30.5.

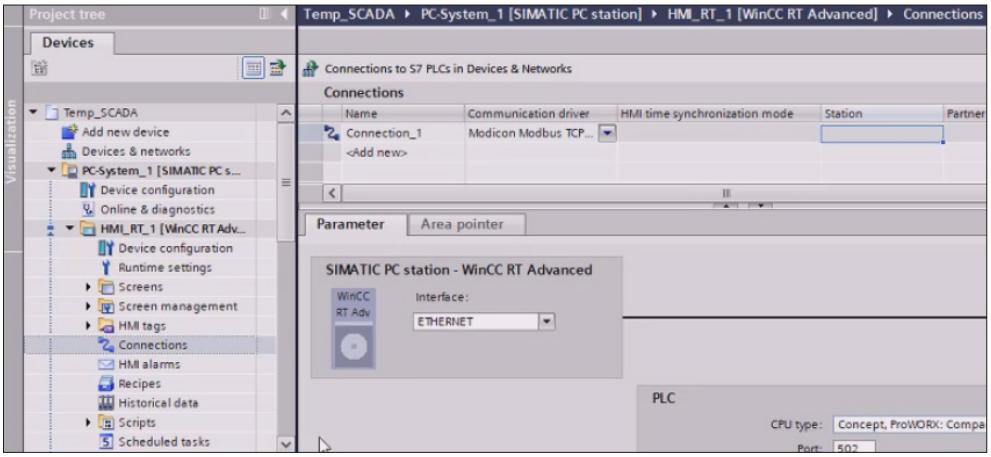


Figure 30.5: Setting the "Connections" icon.

In the project tree on the left side, we select the "Show all tags" option under HMI tags. Then, within the HMI tags window, we click on the option located under the "Name" section, as shown in Figure 30.6.

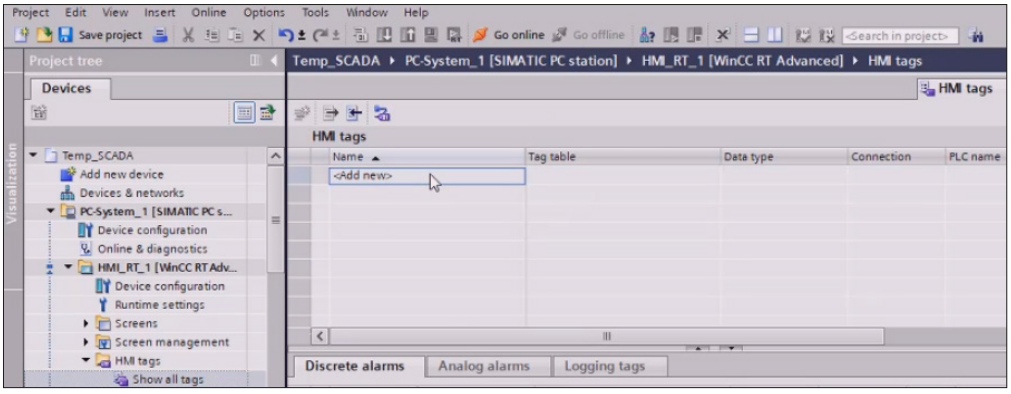


Figure 30.6: Opening the "HMI tags" window.

Next, we incorporate a fresh tag labeled "temp_int" and configure its parameters in alignment with the depiction provided in Figure 30.7.

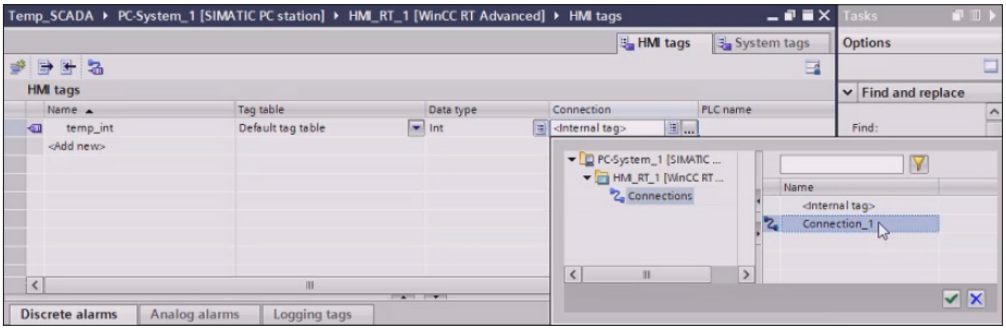


Figure 30.7: Setting the tag "temp_int" parameters.

In Figure 30.8, we assign the address 4x400001 to the tag "temp_int".

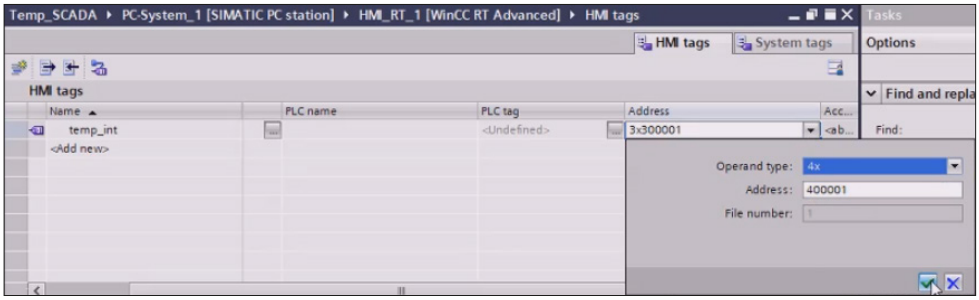


Figure 30.8: Setting the tag "temp_int" address.

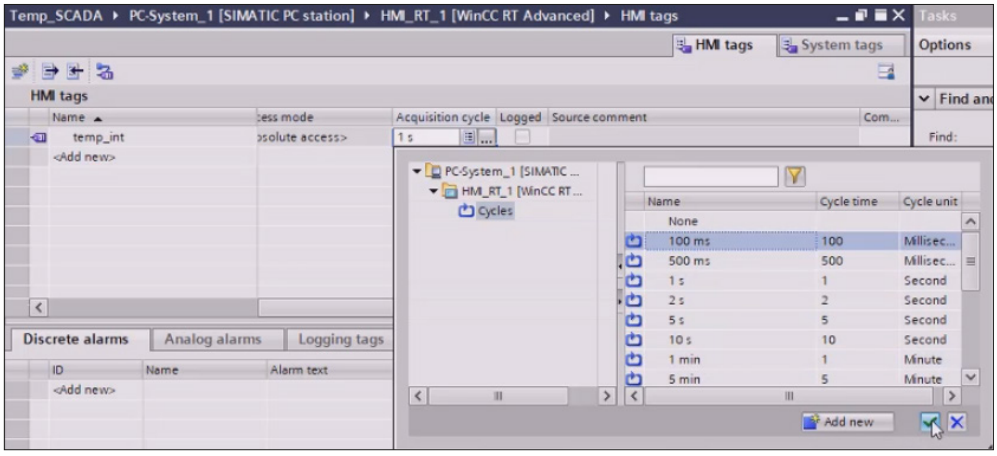


Figure 30.9: Setting the "Acquisition cycle".

In Figure 30.9, we adjust the acquisition cycle parameter to 100 ms. The acquisition cycle refers to the frequency at which data from a specific tag is obtained or refreshed by the WinCC system. It determines how often the values related to WinCC tags are retrieved from the underlying data source, such as a PLC or database. Each WinCC tag can have its own acquisition cycle configuration, enabling users to specify how frequently the values are

updated. This ensures that the data displayed and logged in WinCC is accurate and current. The acquisition cycle can be specified in milliseconds, seconds, or minutes, depending on the application requirements. As shown in Figure 30.10, we create a new tag called "heater" and specify its "Data type" as Bit.

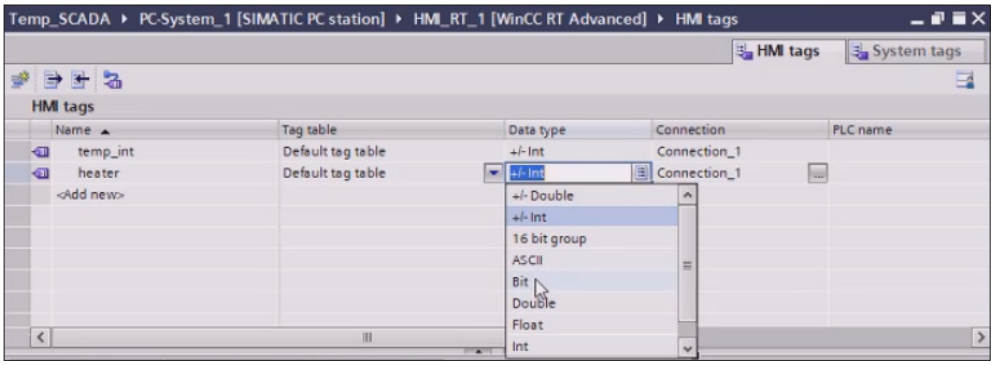


Figure 30.10: Defining a new tag called heater.

In Figure 30.11, we assign the address 0x1 to the tag "heater".

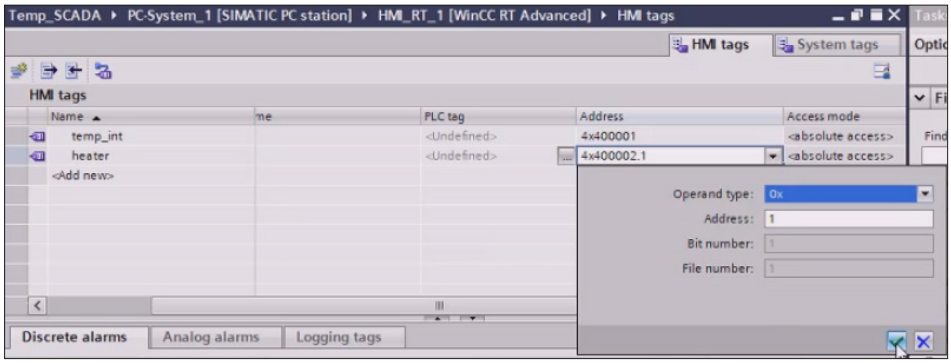


Figure 30.11: Setting the tag "heater" address.

Once again, we establish the acquisition cycle parameter as 100 ms, as illustrated in Figure 30.12.

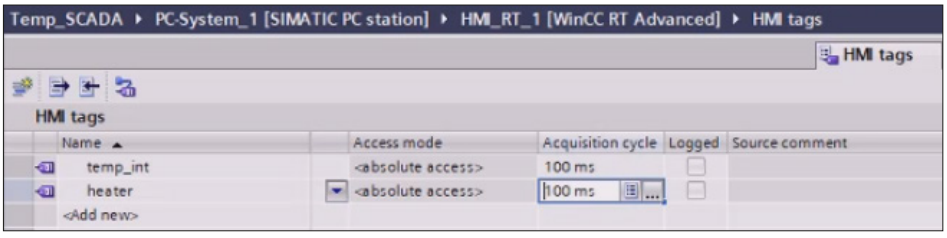


Figure 30.12: Setting the "Acquisition cycle".

Another tag called "control" is included, which has identical parameters as the "heater" tag, but with a distinct address. In this scenario, the address is designated as 0x2, as depicted in Figure 30.13 and Figure 30.14 correspondingly.

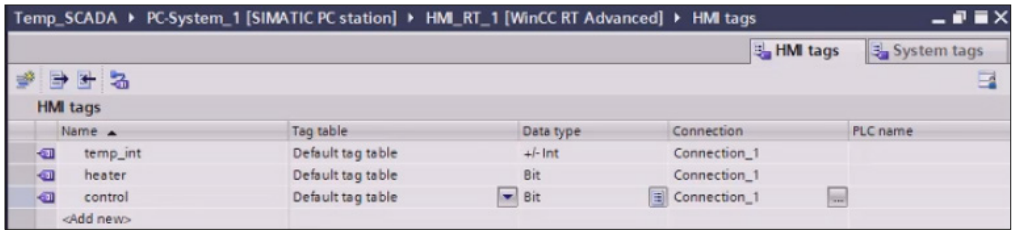


Figure 30.13: Setting the new tag "Data type".

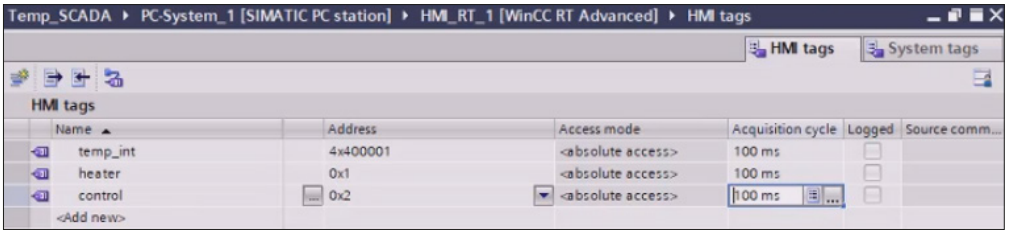


Figure 30.14: Setting the new tag "Address" and "Acquisition cycle".

In WinCC, users can use the internal tag connection feature to set up a connection between different tags within the project. By enabling communication and data exchange between these tags, the flow of information is made easier. The internal tag connection functionality in WinCC offers several applications, such as connecting process tags to display tags, linking internal memory tags to process tags, or connecting a script tag to a user interface tag. By setting up internal tag connections, users can develop a dynamic and interactive HMI system, allowing for the visualization and manipulation of real-time data in real-time operations. To illustrate, a new tag named "temp_real" with a data type of "Float" is created, and its connection is adjusted following Figure 30.15 and Figure 30.16.

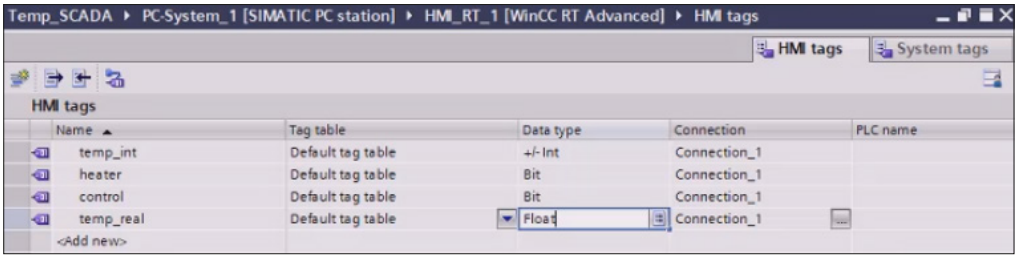


Figure 30.15: Setting the new tag "Data type" as Float.

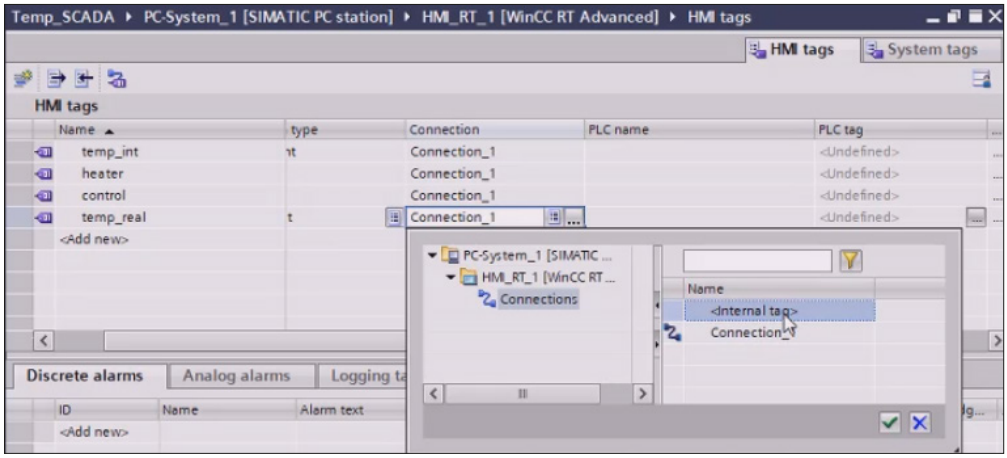


Figure 30.16: Setting the new tag "Connection" as <internal tag>.

The screens in WinCC can incorporate various elements, including buttons, text fields, data tables, trend charts, alarms, and other visualization objects. These screens can be customized and configured to display real-time data from the industrial automation system that is connected to, thereby allowing operators to effectively monitor and control processes. WinCC offers a robust screen editor that enables users to design and create screens with ease, using a drag-and-drop interface. These screens can be organized hierarchically and interconnected to establish a structured navigation system, enabling users to access different parts of the HMI system. Additionally, the screens can be designed to accommodate different languages and provide a user-friendly interface for operators to interact with the system. To create a new screen named "Temperature Control", we navigate to the Screens icon in the project tree and select the "Add new screen" option. Then, from the right-side "Elements" section, we choose a button, as shown in Figure 30.17.

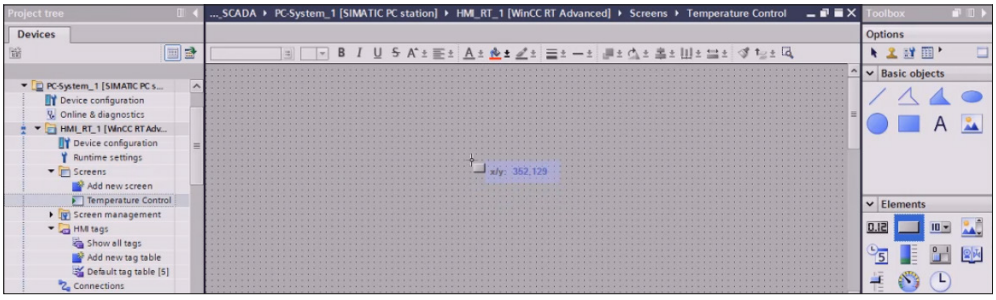


Figure 30.17: Selecting a text box from "Elements" section.

First, we perform the action of copying and pasting, and then we include a new button and change the names of these buttons to "Stat" and "Stop." Next, from the Basic objects section, we select the circle object, as showed in Figure 30.18. Afterward, we place two circle objects next to the inserted buttons and include a text object on the screen, as depicted in Figure 30.19.

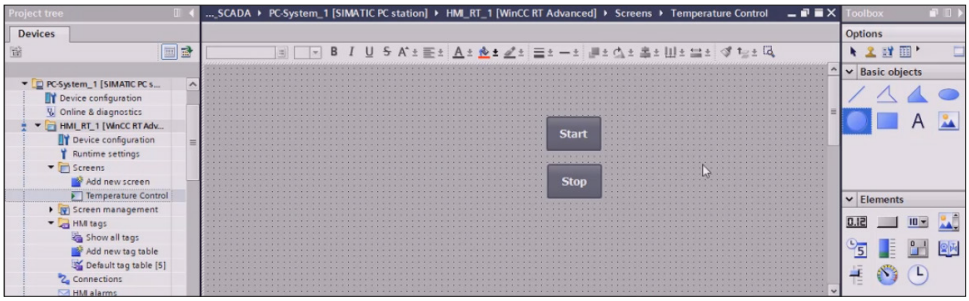


Figure 30.18: Adding elements and objects to the "Temperature Control" screen.

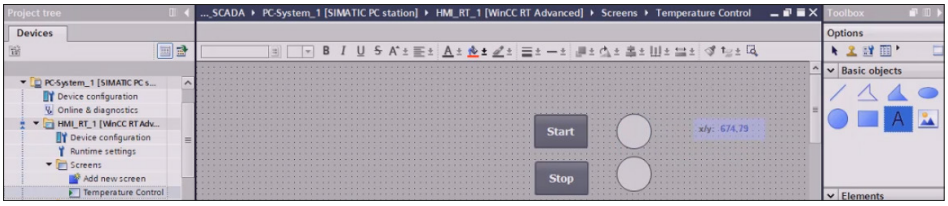


Figure 30.19: Added two circles objects.

In Figure 30.20, we depict the addition of three text objects and their renaming as "System Status," "System Running," and "System Stop."

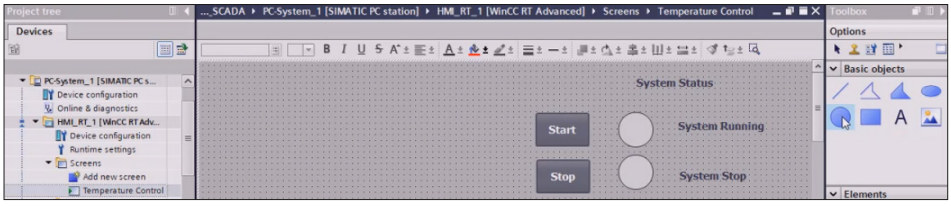


Figure 30.20: Added two text objects.

Following the same process as before, we include two circles and two text elements and give them the names "Heater ON" and "Heater OFF" as indicated in Figure 30.21. Then, as depicted in Figure 30.21, we select the display number element and place it on the screen. Additionally, we locate a text object in close proximity to the display number and label it as "Liquid Temperature" as seen in Figure 30.22.

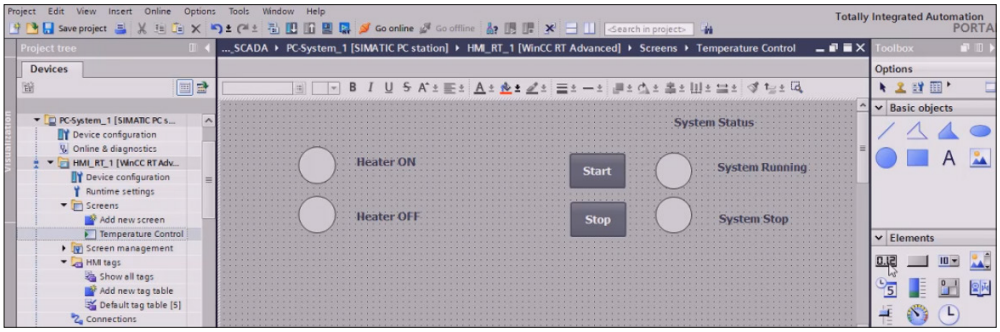


Figure 30.21: Adding new two circles and two text objects.

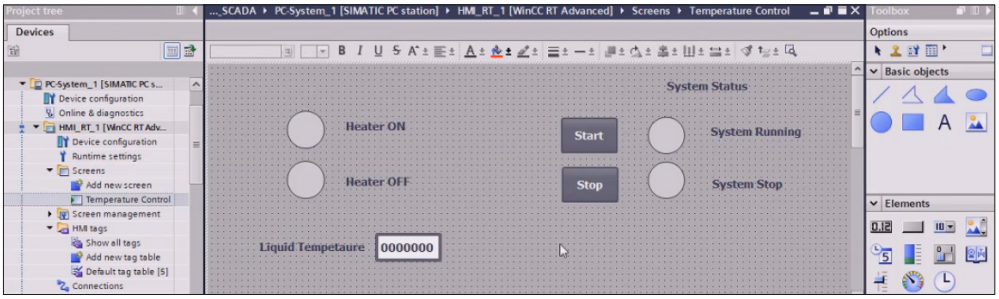


Figure 30.22: Adding display element and text object.

To perform the desired action, proceed by right-clicking on the "Start" button and choosing "Properties." Then, navigate to the "Events" tab and select the "Click" option. From there, locate the **SetBit** function in the right window and opt for the "control" tag for the Tag(input/output) option as shown in Figure 30.23.

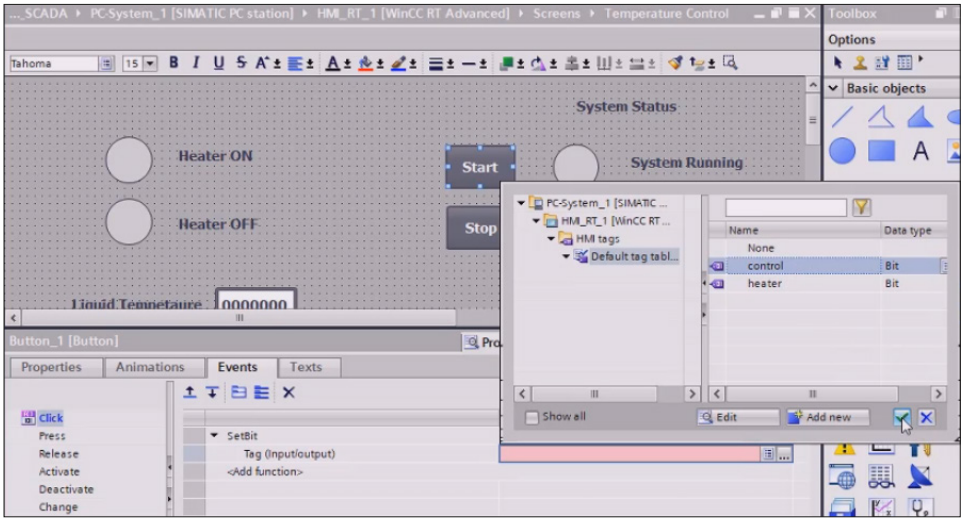


Figure 30.23: Choosing the "control" tag for the Tag(input/output) option.

In this case, we follow the same steps as we did for the "Start" button when dealing with the "Stop" button. However, we choose the "ResetBit", as shown in Figure 30.24.

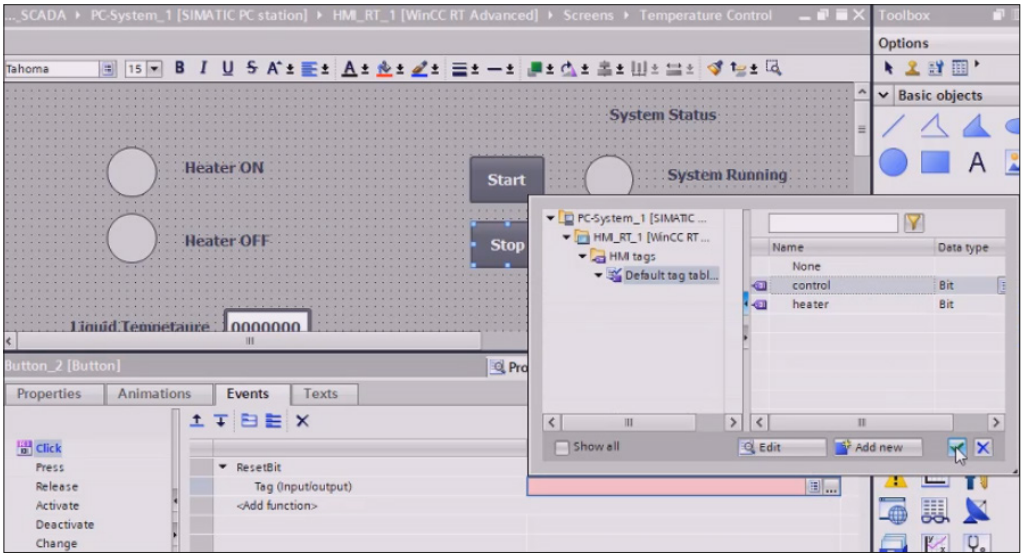


Figure 30.24: Choosing the "control" tag for the Tag(input/output) option.

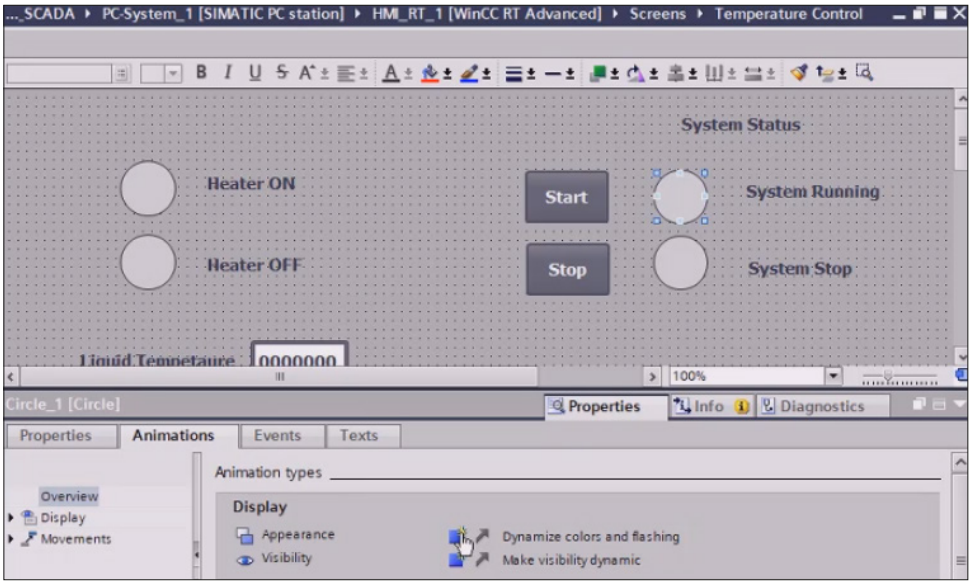


Figure 30.25: Dynamizing colors and flashing for circle object.

To activate the dynamic colors and flashing of the "System Running" circle object, we need to click on it and navigate to the "Animations" tab. In the "Animation types" section, we select the "Dynamize colors and flashing" option, as shown in Figure 30.25. In the newly opened "Appearance" window, we specify the tag name as "control" and the type as

"Range". Afterwards, we fill the Ranges 0 and 1 with the corresponding black and red colors for the "Background" colors, as illustrated in Figure 30.26.

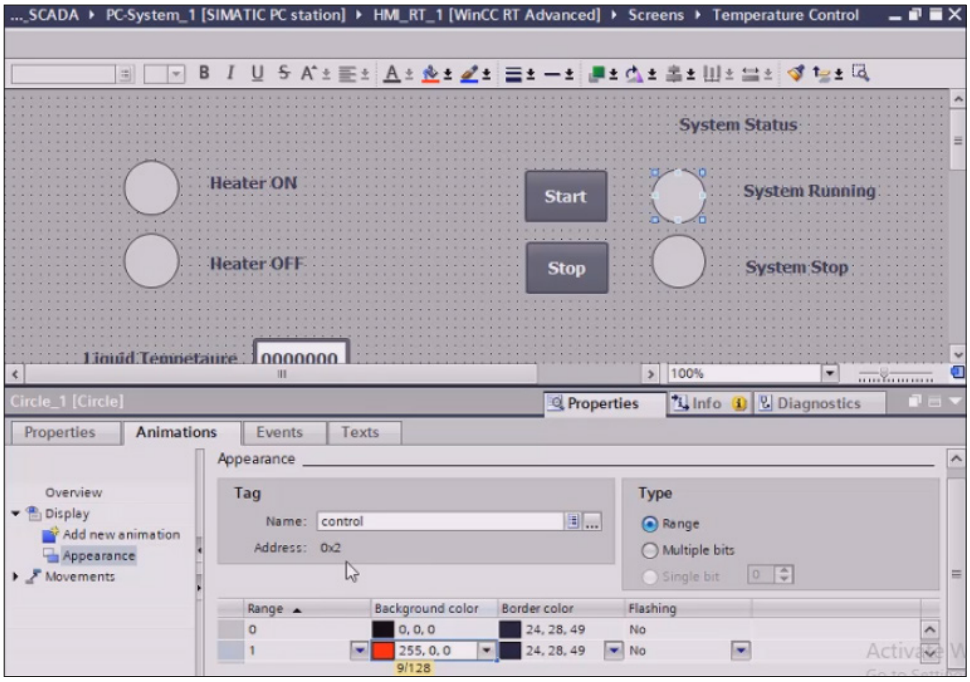


Figure 30.26: Filling "Background" colors for Ranges.

The process of dynamizing colors and flashing for the "System Stop" circle object is carried out in the same way as for other objects. However, in this case, we fill Ranges 0 and 1 with red and black colors, respectively for the corresponding "Background" colors, as shown in Figure 30.27. Similarly, for the "Heater ON" circle object, we follow the same procedures as for the "System Running" circle object, but distinguishably, we choose the tag name as "heater", as showed in Figure 30.28.

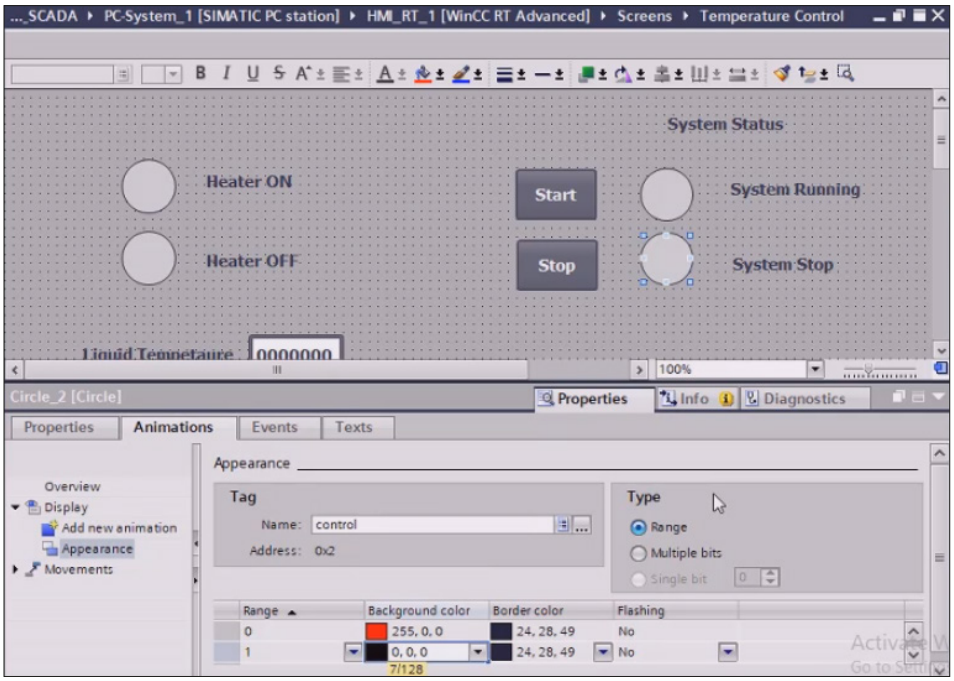


Figure 30.27: Filling "Background" colors for Ranges.

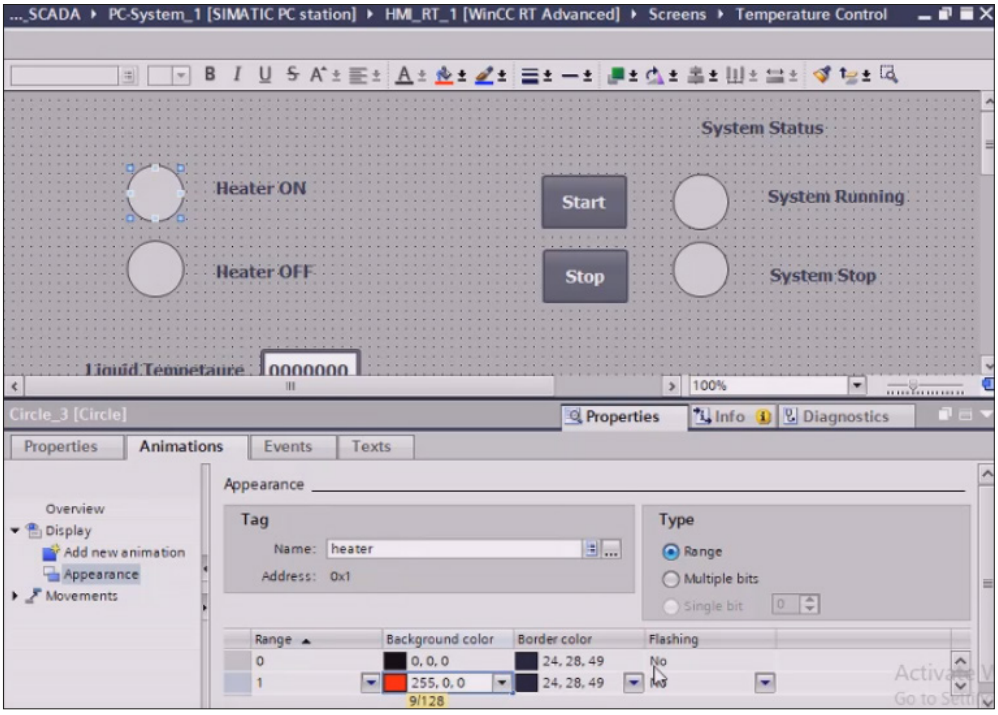


Figure 30.28: Filling "Background" colors for Ranges.

We follow the same steps for making the colors dynamic and flashing for the "Heater OFF" circle object, similar to what we did for the "System Stop" circle object. The only difference is that in this case, we choose the tag name as "heater", as shown in Figure 30.29.

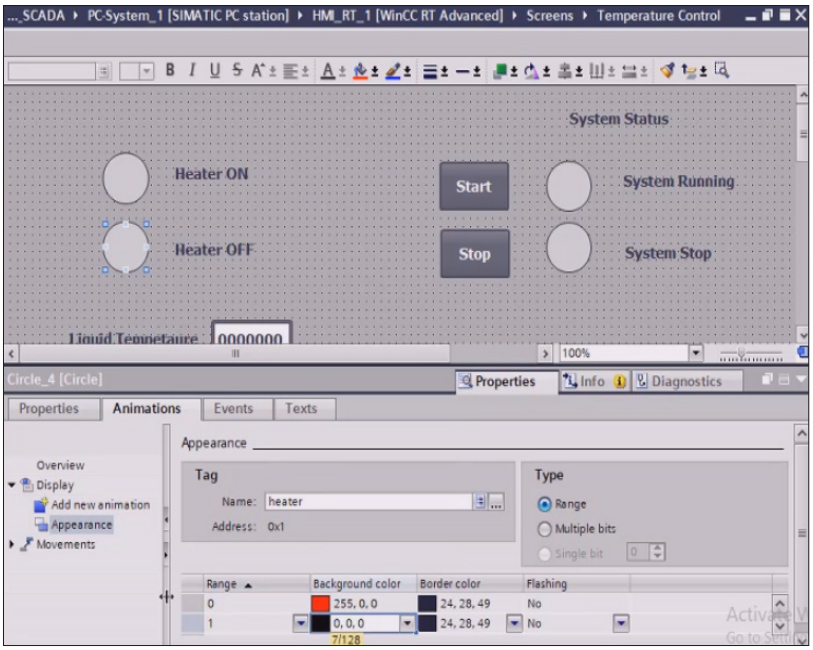


Figure 30.29: Filling "Background" colors for Ranges.

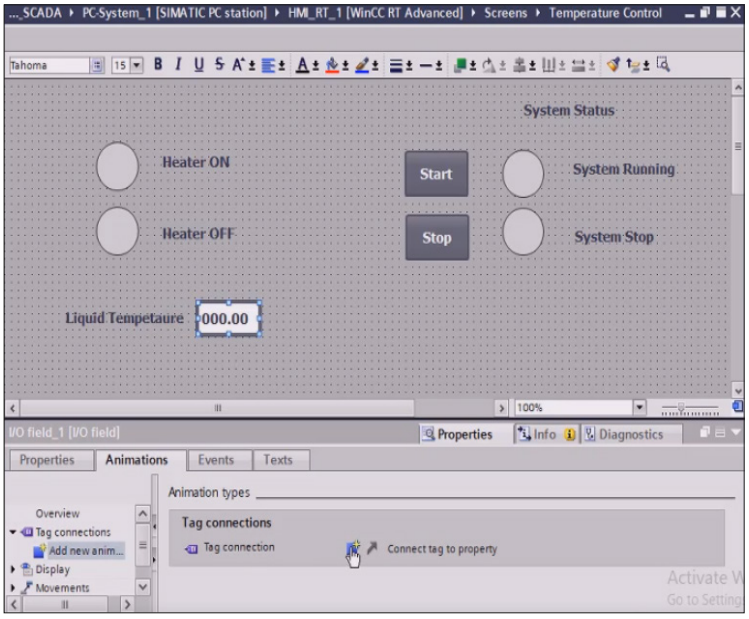


Figure 30.30: The "Liquid Temperature" tag connection.

First, we click on the "Liquid Temperature" display element. Next, we go to the "Animations" tab and choose "Add new animation" in the "Animation types" window. In the "Tag connections" section, we select "Connect tag to property" following the steps shown in Figure 30.30. Then, we open the "Add tag connection" window and choose the "Process value" option, as shown in Figure 30.31.

Figure 30.31: Selecting the "Process value".

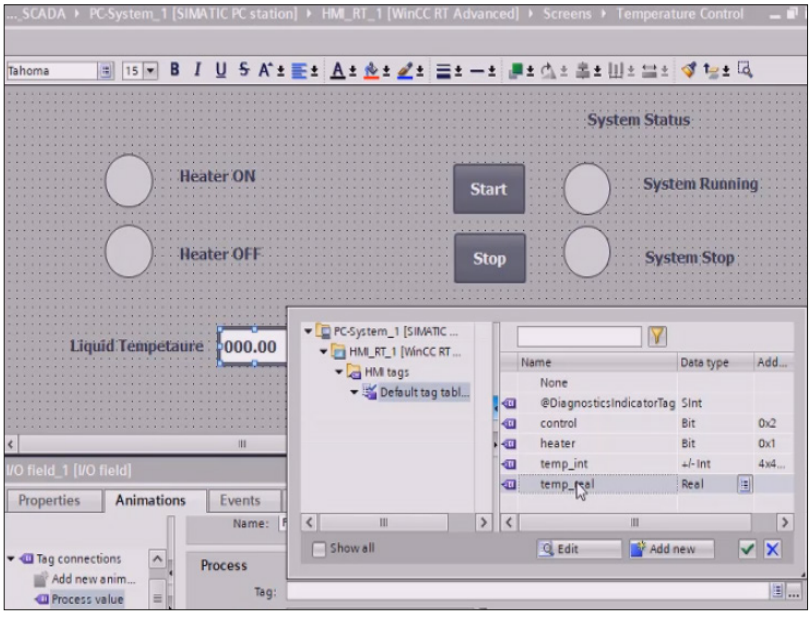


Figure 30.32: Selecting "temp_real" as Process Tag.

Next, we select "temp_real" as the Process Tag and then continue to click on the "Edit" button, as indicated in Figure 30.32, to open the "temp_real" window. Within the "Events" tab, we choose to click on the "LinearScaling" option. In the "Y(Output)" row, we designate the "temp_real" tag. For the "a" row, we input a value of 0.488. In the "X" row, we select the "temp_int" tag and assign a value of zero in the "b" row. Finally, we confirm our selections by clicking on the "OK" button, as displayed in Figure 30.33. It is important to note that the **LinearScaling** function uses the linear equation Y = (aX) + b to assign a value to the Y tag, which is determined by the value of the X tag provided.

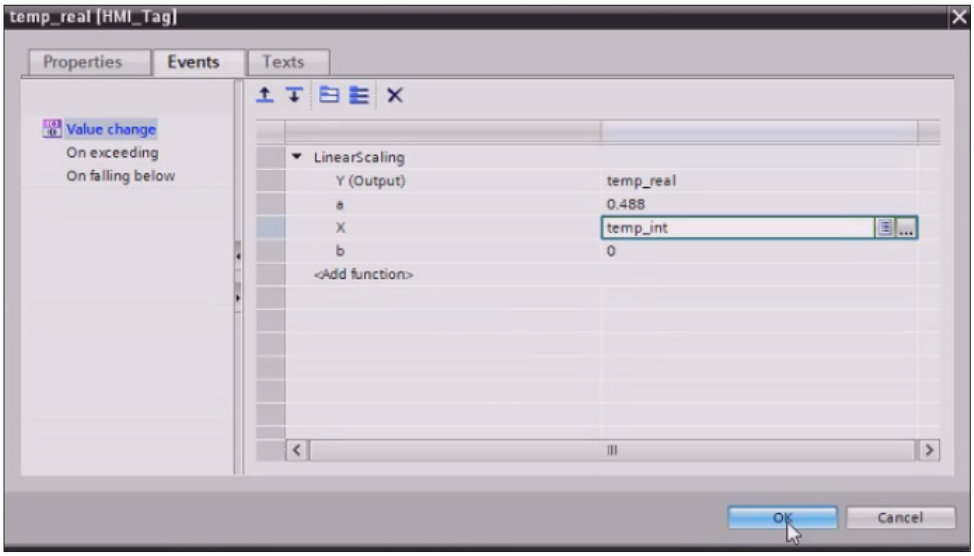


Figure 30.33: Setting the "LinearScaling" parameters.

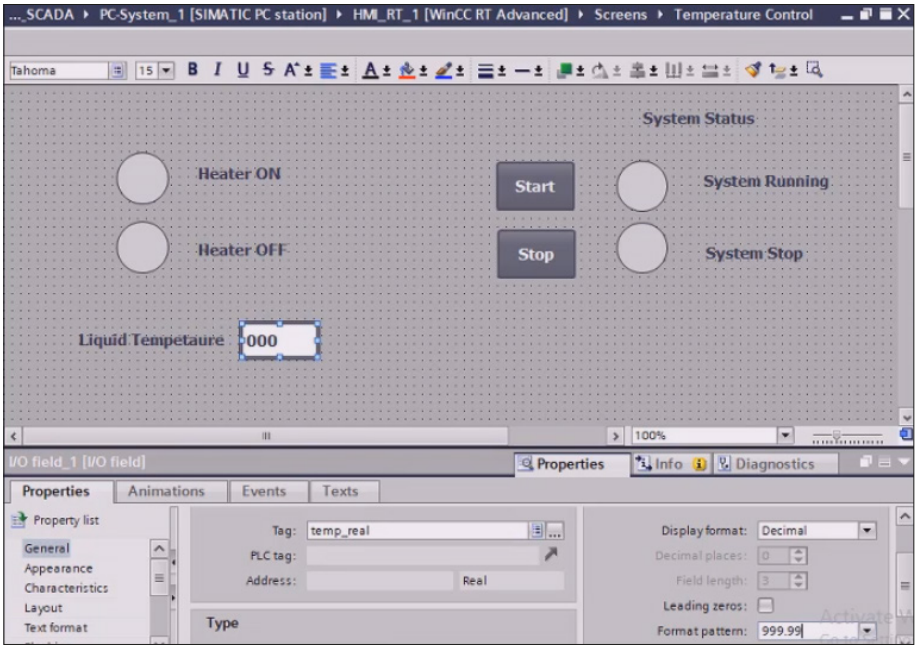


Figure 30.34: Setting the Format pattern of "Liquid Temperature".

Afterward, input the value 999.99 into the "Format pattern" field in the "Properties" tab, as shown in Figure 30.34. Finally, include a text object and change its name to "Heater Status" before saving the project, as showed in Figure 30.35.

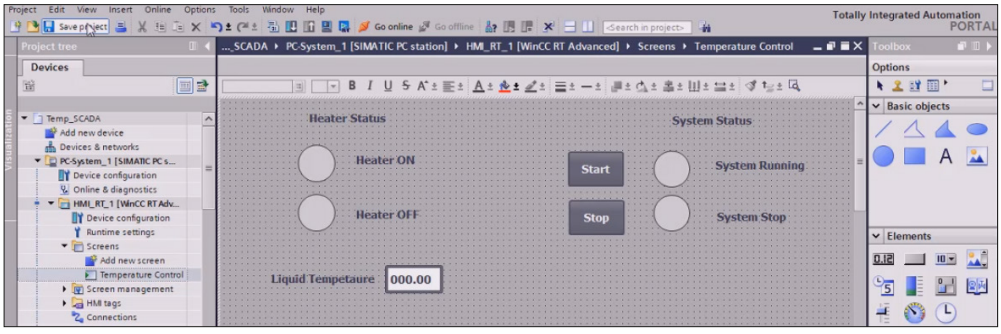


Figure 30.35: Adding a text object and saving the project.

Chapter 31 • The Ethernet and Blynk Project

The Ethernet Shield W5100 will be used in this project for controlling the display and enabling the turning of LED lights on and off from any location. Additionally, it will connect to the internet using the Blynk app, which simplifies the process of connecting our devices to the internet. Based on the Arduino UNO + Blynk project, LED lights can now be controlled remotely through the internet. To address this issue, we will incorporate an Ethernet Shield W5100. The current setup enables the Arduino to set up a connection with the outside world through the internet. Refer to Figure 31.1 for a visual representation of the connection between the Arduino, Ethernet Shield, and Internet.

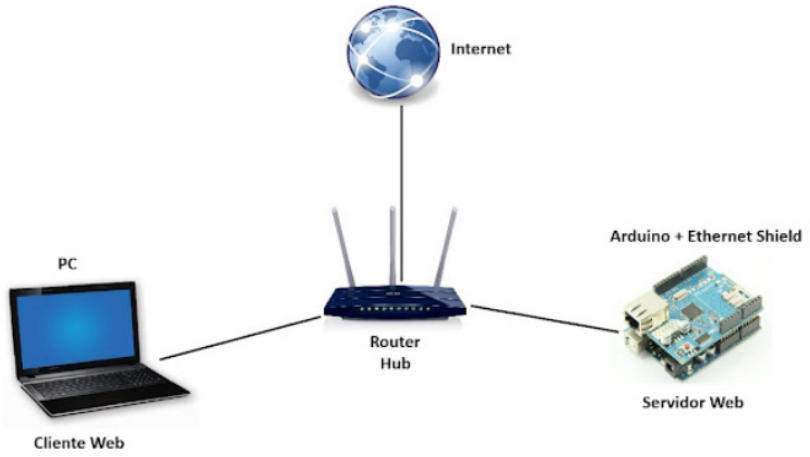


Figure 31.1: The connection between Arduino, Ethernet Shield, and Internet.

The IoT, also known as the Internet of Things, refers to a technology where devices and appliances are connected through the internet network. This concept has generated significant interest and is the subject of ongoing discussions. There is a continuous stream of new hardware advancements, standard announcements, and the introduction of new services and tools. One intriguing service in this domain is Blynk, as represented in Figure 31.2. You can download the Blynk mobile app for iOS and Android systems from the link provided on the following webpage:

https://docs.blynk.io/en/downloads/blynk-apps-for-ios-and-android

Figure 31.2: The Blynk app.

The Blynk Application is a mobile program used to control or display data connected to microcontrollers (such as Arduino, ESP8266, Raspberry Pi) and can also be controlled remotely.

It can be connected to the Internet. The hardware used in this project is as follows:

- Arduino UNO R3 - Made in Italy.
- Ethernet Shield W5100.
- 1 Channel DC 5V Relay Module.
- DC 5 V 2 A Power Supply Adapter
- LED Lamp + Power Supply

Assemble the Ethernet Shield, W5100, on the Arduino Uno and connect it to a single relay module, as shown in Figure 31.3.

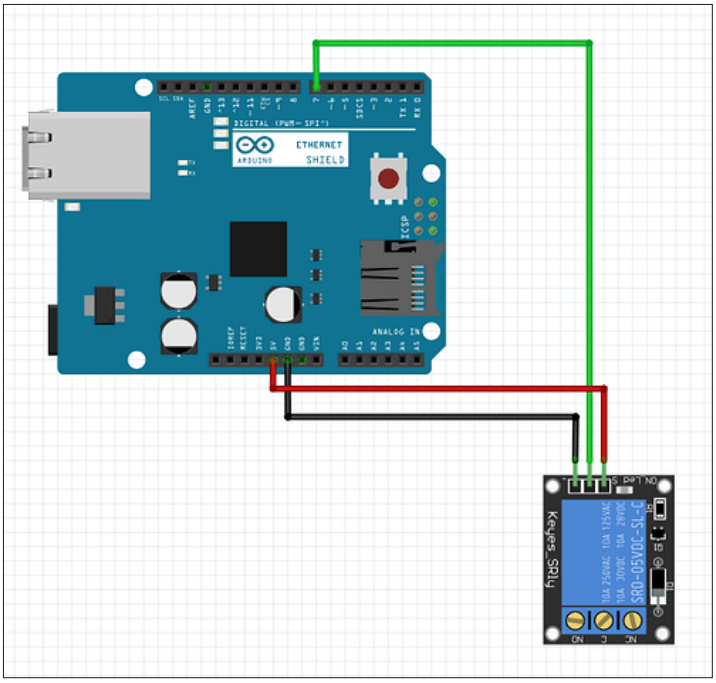


Figure 31.3: Connecting the Ethernet Shield to a single relay module.

Afterwards, launch the Blynk app and choose the option "Create New Account", as shown in Figure 31.4.

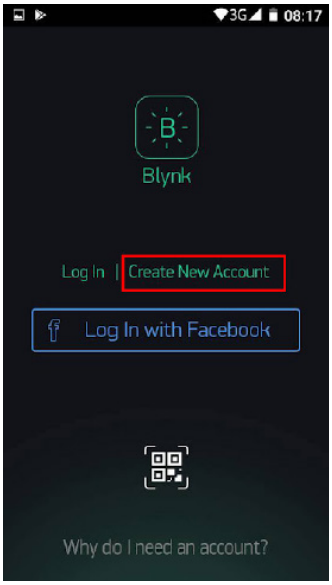


Figure 31.4: Selecting "Create New Account" in the Blynk application.

Fill in your email and password, and then click on the "Sign-Up" button, as shown in Figure 31.5.

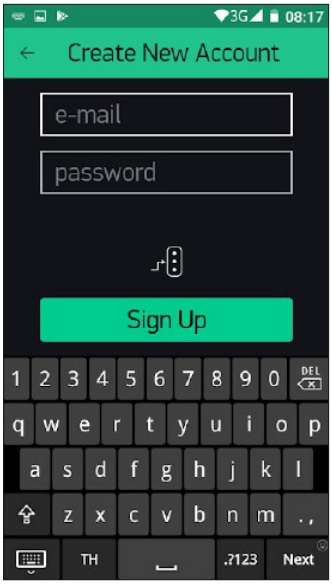


Figure 31.5: Fill in your email and password.

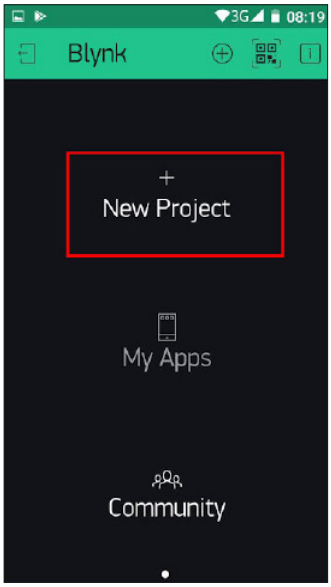


Figure 31.6: Clicking on the "New Project".

Afterwards, we proceed to select the "New Project" icon, which can be seen in Figure 31.6. Subsequently, we choose the "Arduino Uno" as our device, as illustrated in Figure 31.7.

Figure 31.7: Selecting "Arduino Uno" as our device.

The project's connection type is Ethernet, and it has been named "W5100". To email the authentication token to your email address, click on the "E-mail" button, as shown in Figure 31.8.

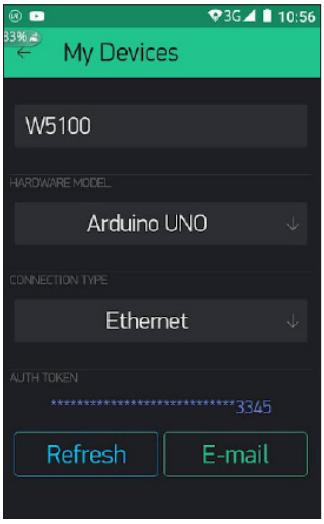


Figure 31.8: Getting the authentication token via email.

After moving to the following page, a notification will appear indicating that the program will forward an authentication token to the provided email address. Proceed by selecting the "OK" button and locate the symbol resembling a plus sign as in Figure 31.9.

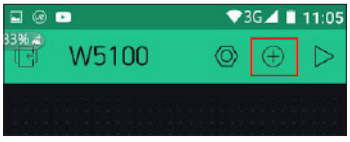


Figure 31.9: Clicking the + sign.

Click on the "Add" button and then select the "Button" option as shown in Figure 31.10.

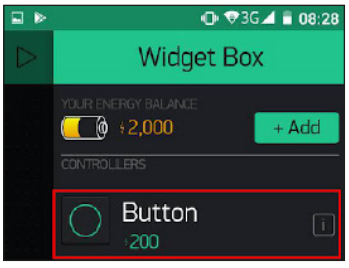


Figure 31.10: Selecting "Button" from the "Widget Box".

"BUTTON" will be included on the screen for you to click on, enabling you to set the value according to the illustration shown in Figure 31.11.

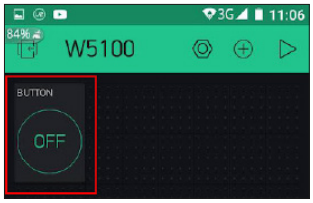


Figure 31.11: Clicking on "BUTTON" to set the value.

Afterward, choose the "SWITCH" mode and continue to click on "PIN" according to the presentation in Figure 31.12.

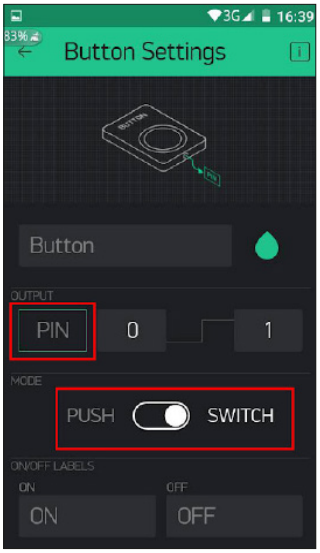


Figure 31.12: Selecting "SWITCH" type and clicking on "PIN".

Next, follow the steps shown in Figure 31.13: Choose pin → Digital → D7 → CONTINUE.

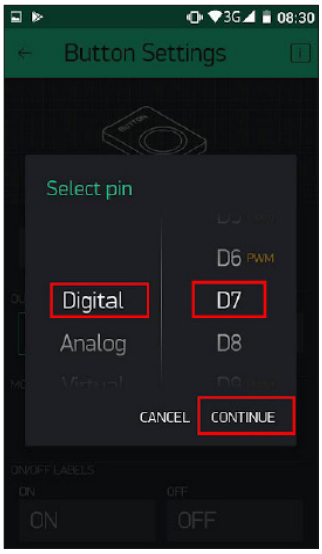


Figure 31.13: Setting the PIN.

As shown in Figure 31.14, "BUTTON" will be situated on the left-hand side.

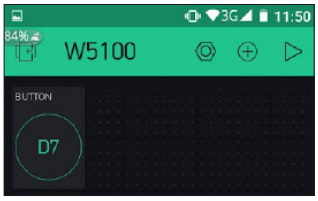


Figure 31.14: "BUTTON" on the left-hand side.

To achieve a visually pleasing result, we should move the button to the center of the screen.

The next steps are how to add a new library to your Arduino IDE using the "Library Manager" (accessible from IDE version 1.6.2 onwards). Start by opening the IDE and navigating to the "Sketch" menu. From there, select "Include Library" and then choose "Manage Libraries", as shown in Figure 31.15.

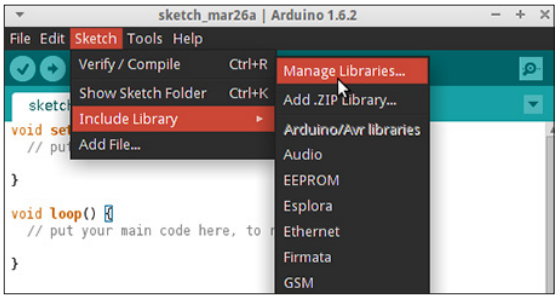


Figure 31.15: Installing a new library into your Arduino IDE.

The "Library Manager" will open, and you will find a list of libraries that are already installed or ready for installation. Search for Blynk library and in the version, selection choose the latest version to date, as shown in Figure 31.16.

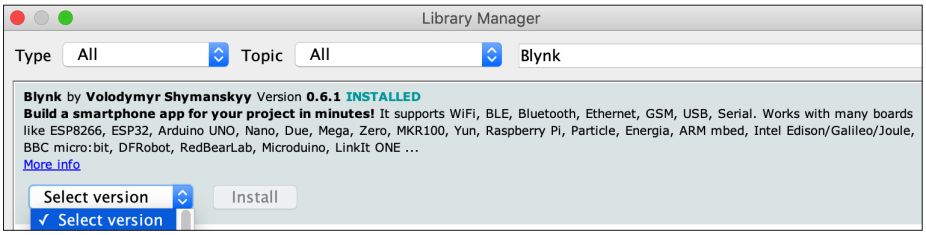


Figure 31.16: Installing the Blynk library.

Finally, click on "Install" and wait for the IDE to install the new library. Downloading may take time depending on your connection speed. Once it has finished, an "Installed" tag should appear next to the "Bridge" library. You can close the library manager. You can now find the new library available in the "Sketch" → "Include Library" menu. When finished installing the Blynk library, go to "File" → "Examples" → "Blynk" → "Boards_Ethernet" → "Arduino_Ethernet", as depicted in Figure 31.17.

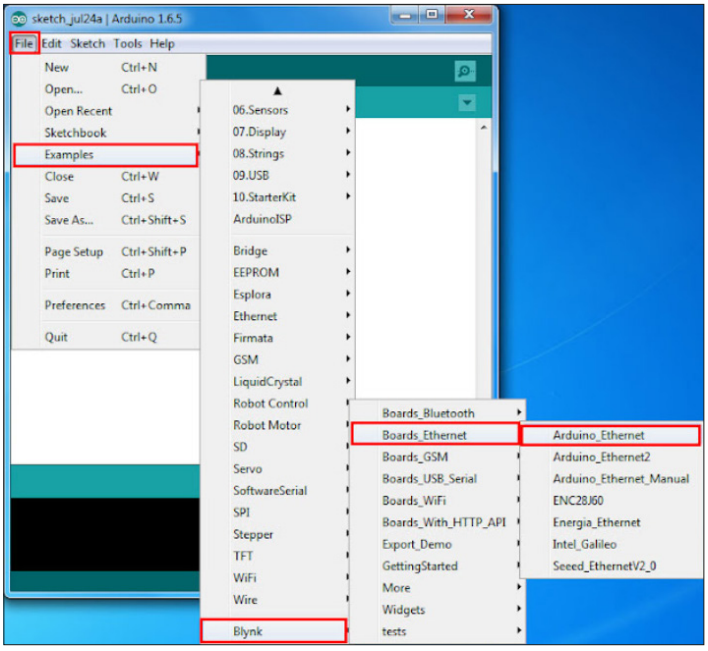


Figure 31.17: Selecting "Arduino_Ethernet" examples.

Edit the "YourAuthToken" obtained from the email and incorporate it into the Arduino sketch, as shown in Figure 31.18.

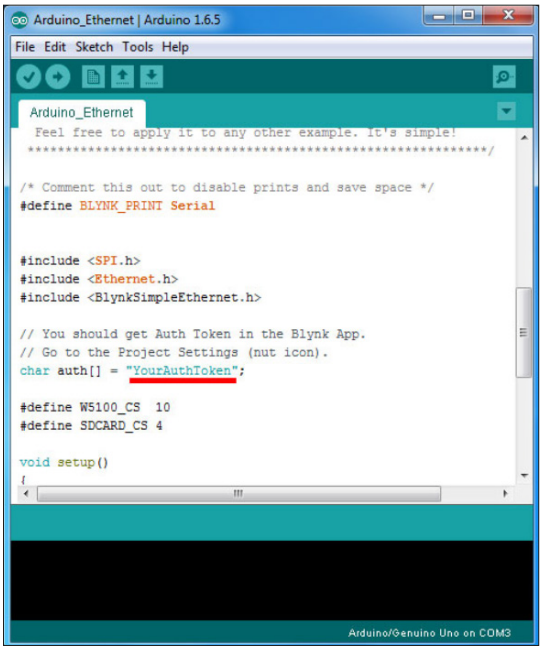


Figure 31.18: Edit the «YourAuthToken» into the Arduino sketch.

Once edited, upload the code as follows to Arduino Uno.

```
/*************************************************************
Download latest Blynk library here:
https://github.com/blynkkk/blynk-library/releases/latest
Blynk is a platform with iOS and Android apps to control
Arduino, Raspberry Pi and the likes over the Internet.
You can easily build graphic interfaces for all your
projects by simply dragging and dropping widgets.
Downloads, docs, tutorials: http://www.blynk.cc
Sketch generator: http://examples.blynk.cc
Blynk community: http://community.blynk.cc
Social networks: http://www.fb.com/blynkapp
http://twitter.com/blynk_app
Blynk library is licensed under MIT license
This example code is in public domain.
*************************************************************
This example shows how to use Arduino Ethernet shield (W5100)
to connect your project to Blynk.
NOTE: Pins 10, 11, 12 and 13 are reserved for Ethernet module.
DON'T use them in your sketch directly!
WARNING: If you have an SD card, you may need to disable it
by setting pin 4 to HIGH. Read more here:
https://www.arduino.cc/en/Main/ArduinoEthernetShield
```

```
Feel free to apply it to any other example. It's simple!
*************************************************************/
/* Comment this out to disable prints and save space */
#define BLYNK_PRINT Serial
#include <SPI.h>
#include <Ethernet.h>
#include <BlynkSimpleEthernet.h>
// You should get Auth Token in the Blynk App.
// Go to the Project Settings (nut icon).
char auth[] = "b8152a62d1c54b20862ab77ae4b23345"; // Must to be fixed
#define W5100_CS 10
#define SDCARD_CS 4
void setup()
{
// Debug console
Serial.begin(9600);
pinMode(SDCARD_CS, OUTPUT);
digitalWrite(SDCARD_CS, HIGH); // Deselect the SD card
Blynk.begin(auth);
// You can also specify server:
//Blynk.begin(auth, "blynk-cloud.com", 8442);
//Blynk.begin(auth, IPAddress(192,168,1,100), 8442);
// For more options, see Boards_Ethernet/Arduino_Ethernet_Manual example
}
void loop()
{
Blynk.run();
}
```

Disconnect the USB cable and replace it with an RJ45 LAN cable to establish the connection between the Ethernet Shield and the router. This will enable the shield to communicate with the network and receive power from a 5-V adapter connected to the Arduino UNO. Return to the Blynk Application and click on the symbol indicated in Figure 31.19 three times to evaluate the functionality of the W5100 program. Refer to Figure 31.20 for a visual representation of the hardware setup for the project.

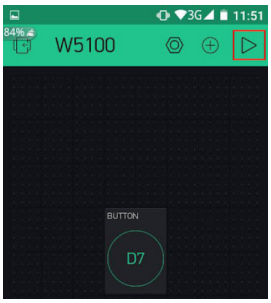


Figure 31.19: Clicking the symbol.

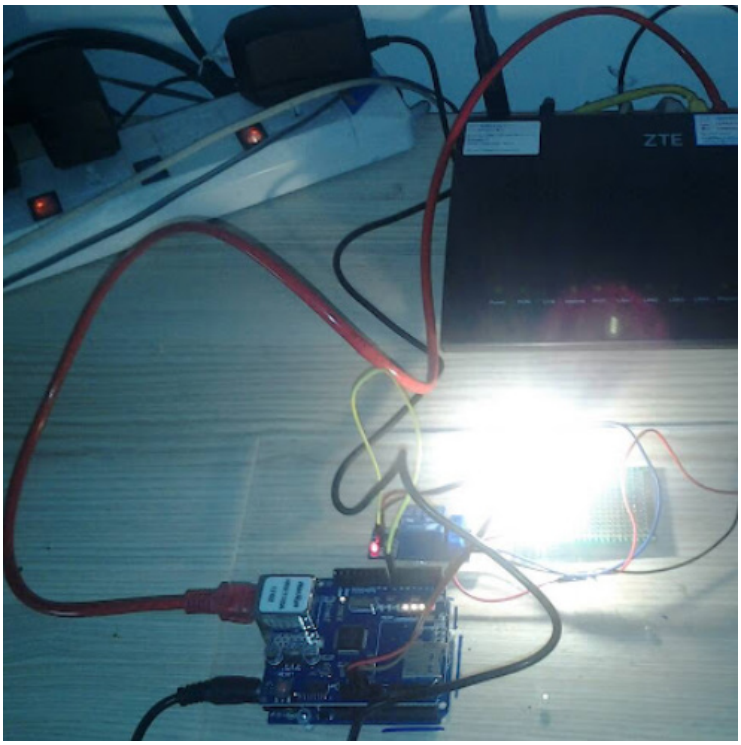


Figure 31.20: The hardware implementation of the project.

Chapter 32 • Temperature/Humidity Sensor, Ethernet and Blynk Project

This section will cover a mobile application that provides real-time information about the temperature and humidity of a room. The application allows users to connect their Android devices to a hardware setup. To detect the temperature and humidity, a DHT11 sensor is connected to an Arduino Uno. These readings are then transmitted to the Android device and presented neatly in an app. This setup is simple to build and is perfect for monitoring a remote room with an impressive display. To create this system, you will need an Arduino Uno, an Arduino Ethernet Shield, a DHT11 Sensor, three male-to-female jumper wires, the Arduino IDE, the Blynk app, and a LAN Cable. The sensor is connected to digital pin 2 on the Arduino. The app generates a virtual display that constantly updates with real-time values, as depicted in Figure 32.1.

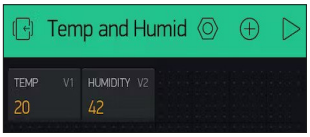


Figure 32.1: Blynk app displaying the values.

In Figure 32.2, the schematic diagram illustrates the connection between the DHT11 temperature and humidity sensor and the Ethernet shield.

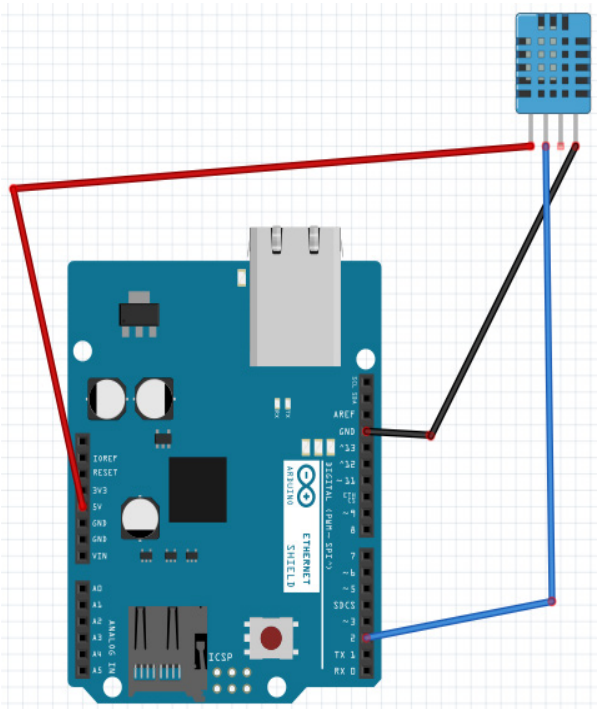


Figure 32.2: Ethernet shield connection to the DHT11 sensor.

Here is the Arduino code provided:

```
#include <SPI.h>
#define BLYNK_PRINT Serial
#include <Ethernet.h>
#include <BlynkSimpleEthernet.h>
#include <SimpleTimer.h>
#include <dht11.h>
dht11 DHT11;
SimpleTimer timer;

char auth[] = "";        // write your auth code here
void setup()
{
  DHT11.attach(2);      // connect pin 2 to the sensor
  Serial.begin(9600);
  Blynk.begin(auth);     // start blynk server and connect to the cloud
 while (Blynk.connect() == false) {
    // Wait until connected
  }

void loop()
{
  int chk = DHT11.read();
   Blynk.virtualWrite(1, DHT11.temperature);  // Write values to the app
  Blynk.virtualWrite(2, DHT11.humidity);
Serial.println(DHT11.temperature);

  Blynk.run();
  timer.run();

}
```

Chapter 33 • MQTT Protocol with Ethernet/ESP8266 Project

We will explore the usage of the MQTT protocol with Ethernet, and ultimately, we will be able to manipulate an LED connected to an Ethernet shield, as well as monitor data from a temperature sensor using the publish and subscribe functionality of the MQTT protocol. Since our Arduino or ESP8266 device lacks knowledge about the MQTT protocol, we need to install the MQTT library.

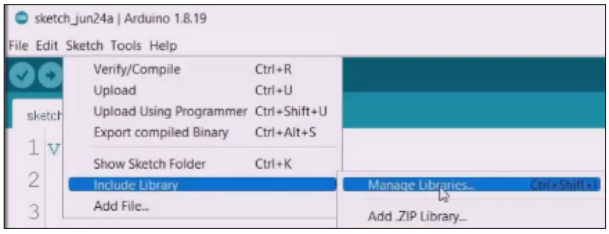


Figure 33.1: Clicking on "Manage Libraries".

First, we need to open "Sketch" and access the "Library" menu. In Figure 33.1, you can see that we need to select "Manage Libraries". It may take a few seconds or minutes for the "Library Manager" to appear. There, search for "PupSubClient" and press "Enter". Look for the version by Nick O'Leary in the search results. The default version also includes the latest MQTT library. You do not need to be concerned about the version of the MQTT library; it is automatically included with the "PupSubClient" library. Simply click on the "Install" button to install the MQTT library for Arduino or ESP8266, illustrated in Figure 33.2.

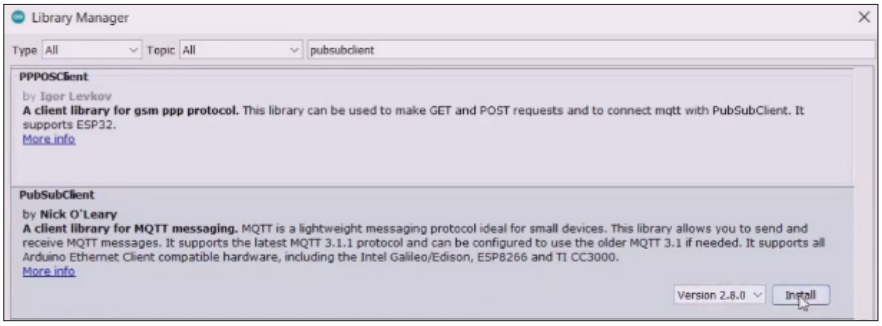


Figure 33.2: Installing the "PubSubClient" library.

After installing the library, we need to close the "Library Manager". Then, we can navigate to "File", "Examples", and scroll down to find "PupSubClient". Within it, we can find the "mqtt_esp8266" as shown in Figure 33.3. This example project will serve as a reference for us. We can remove the initial commands in the code to ensure a cleaner code for better explanation. In the code, we can see the "ESP8266WiFi.h" header file and the MQTT client library, which is located in "PupSubClient.h". As we are using the Ethernet shield, we need to input the SSID and password for the ESP8266 to connect to the internet. Thus, the code we should use is as follows:

```
/* Repeating Web client
 Circuit:
 * Ethernet shield attached to pins 10, 11, 12, 13 */
#include <SPI.h>
#include <Ethernet.h>
// assign a MAC address for the ethernet controller.
// fill in your address here:
byte mac[] = {
  0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED
};
// Set the static IP address to use if the DHCP fails to assign
IPAddress ip(192, 168, 0, 177);
IPAddress myDns(192, 168, 0, 1);
// initialize the library instance:
EthernetClient client;
char server[] = "www.arduino.cc";  // also change the Host line in httpRequest()
//IPAddress server(64,131,82,241);
unsigned long lastConnectionTime = 0;           // last time you connected to the
server, in milliseconds
const unsigned long postingInterval = 10*1000;  // delay between updates, in
milliseconds
void setup() {
  // You can use Ethernet.init(pin) to configure the CS pin
  // start serial port:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  // start the Ethernet connection:
  Serial.println("Initialize Ethernet with DHCP:");
  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    // Check for Ethernet hardware present
    if (Ethernet.hardwareStatus() == EthernetNoHardware) {
      Serial.println("Ethernet shield was not found.  Sorry, can't run without
hardware. :(");
      while (true) {
        delay(1); // do nothing, no point running without Ethernet hardware
      }
    }
    if (Ethernet.linkStatus() == LinkOFF) {
      Serial.println("Ethernet cable is not connected.");
    }
    // try to configure using IP address instead of DHCP:
    Ethernet.begin(mac, ip, myDns);
    Serial.print("My IP address: ");
```

```
    Serial.println(Ethernet.localIP());
  } else {
    Serial.print("  DHCP assigned IP ");
    Serial.println(Ethernet.localIP());
  }
  // give the Ethernet shield a second to initialize:
  delay(1000);
}
void loop() {
  // if there's incoming data from the net connection.
  // send it out the serial port.  This is for debugging
  // purposes only:
  if (client.available()) {
    char c = client.read();
    Serial.write(c);
  }
  // if ten seconds have passed since your last connection,
  // then connect again and send data:

  if (millis() - lastConnectionTime > postingInterval) {
    httpRequest();
  }
}
// this method makes a HTTP connection to the server:
void httpRequest() {
  // close any connection before send a new request.
  // This will free the socket on the WiFi shield
  client.stop();
  // if there's a successful connection:
  if (client.connect(server, 80)) {
    Serial.println("connecting...");
    // send the HTTP GET request:
    client.println("GET /latest.txt HTTP/1.1");
    client.println("Host: www.arduino.cc");
    client.println("User-Agent: arduino-ethernet");
    client.println("Connection: close");
    client.println();
    // note the time that the connection was made:
    lastConnectionTime = millis();
  } else {
    // if you couldn't make a connection:
    Serial.println("connection failed");
  }
}
```

Then, of course, we need an MQTT server for our project. However, we recommend against using the "broker.mqtt-dashboard.com" as it is a commonly used server in MQTT projects. If you upload the same code, it may cause issues. Therefore, we will search for a different MQTT server. To find our preferred MQTT server, you can open a command prompt and enter "ping test.mosquitto.org". This is a free-to-use MQTT broker or MQTT server, which can be represented by the IP address 5.196.95.208, as depicted in Figure 33.4.

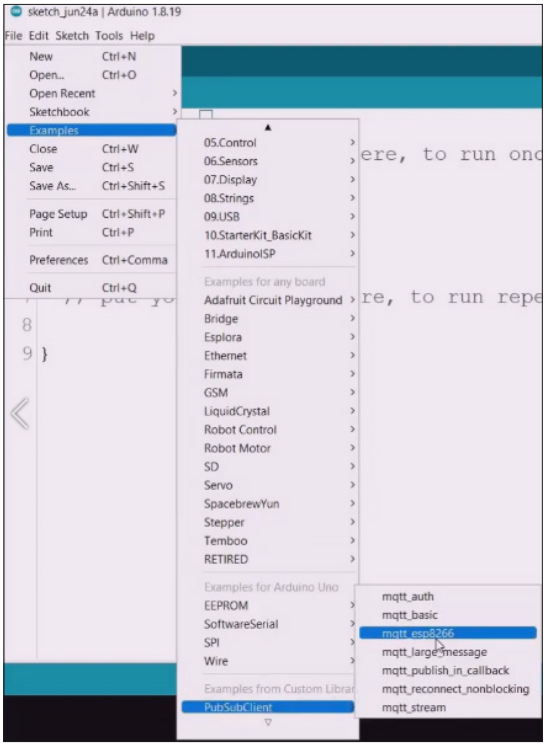


Figure 33.3: Opening the mqtt_esp8266 example.

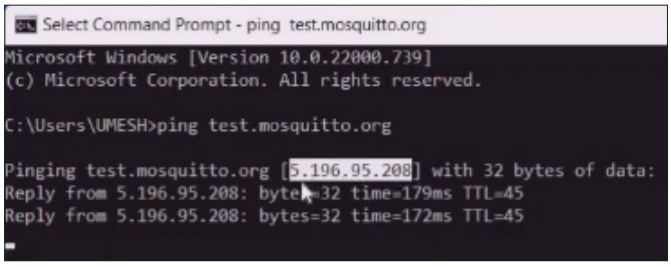


Figure 33.4: Getting the server IP address.

So, we have an IP address that we need to use. We will copy it and then no longer need it, so we can close it and open our project. We will need to insert a double code address for the server, which can be either 5.196.95.208 or "test.mosquitto.org". Both options are acceptable, so I will comment out test.mosquitto.org. This is the MQTT server we are using,

and if you want to learn more about it, you can copy the URL and visit it in a browser. The server we are using is "test.mosquitto.org", and the port for MQTT unencrypted is 1883, as seen in Figure 33.5. However, this information is no longer necessary, and all we need is the IP address of the server, which can be either the IP address or the server name, as shown in Figure 33.6.

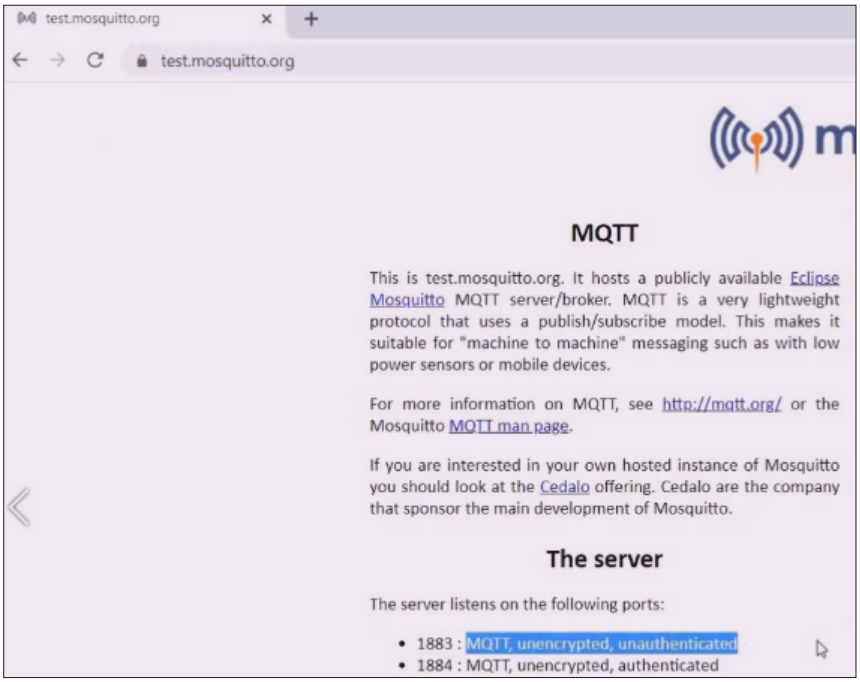


Figure 33.5: Opening the "test.mosquitto.org" with the browser.

Now if we scroll down further then we see that there will be a topic and you can see we are using a built-in LED which is basically connected to GPIO2 on ESP8266.

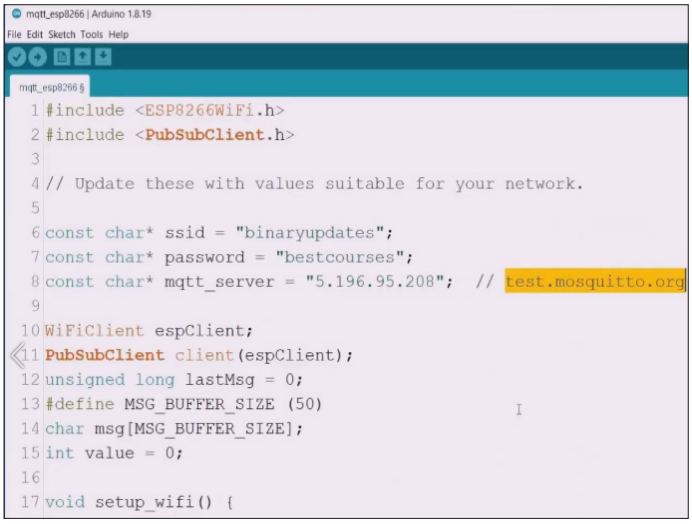


Figure 33.6: Putting the MQTT server IP address or name.

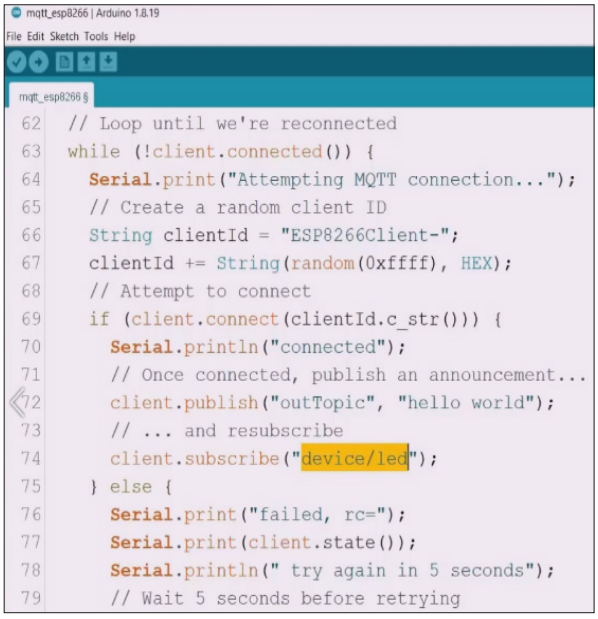


Figure 33.7: Renaming the "client.subscribe" to "device/led".

When we publish a message using the topic "device/led", you'll notice, upon scrolling down, the option to either "client" or "subscribe to" enclosed in double quotes. To enhance readability, we prefer to refer to this topic as "device/led", specifically addressing the control of the onboard LED connected to GPIO2 on the ESP8266, as depicted in Figure 33.7.

In essence, when we publish a message (message one) on the "device/led" topic, the ESP8266 subscribes to this topic, resulting in the activation of the LED. It is worth noting that

the onboard LED on the ESP8266 operates on an active-low configuration. Consequently, publishing message one generates a logic low, turning on the LED. Conversely, a logic high, generated by publishing message zero on the "device/led" topic, turns off the LED, as detailed in Figure 33.8. This simple mechanism allows for the convenient control of the LED on the ESP8266.

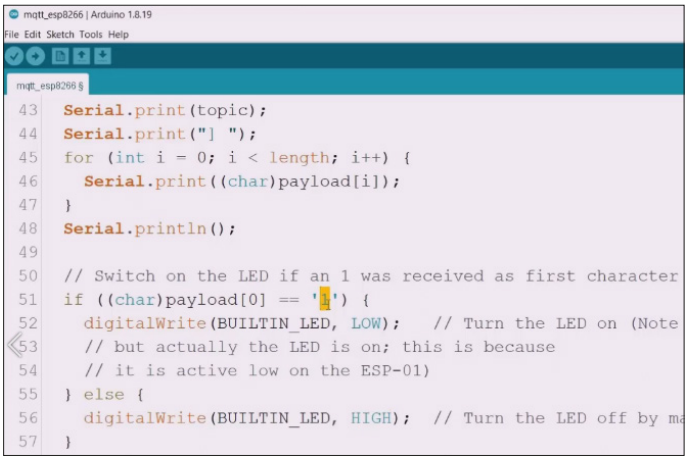


Figure 33.8: Publishing message one on "device/led" topic.

If you continue scrolling, you will come across an option called "outTopic" where you can send sensor data. To send temperature data, we need to specify the name as "device/temp". After doing so, we can confirm that the MQTT server is connected, as shown in Figure 33.9.

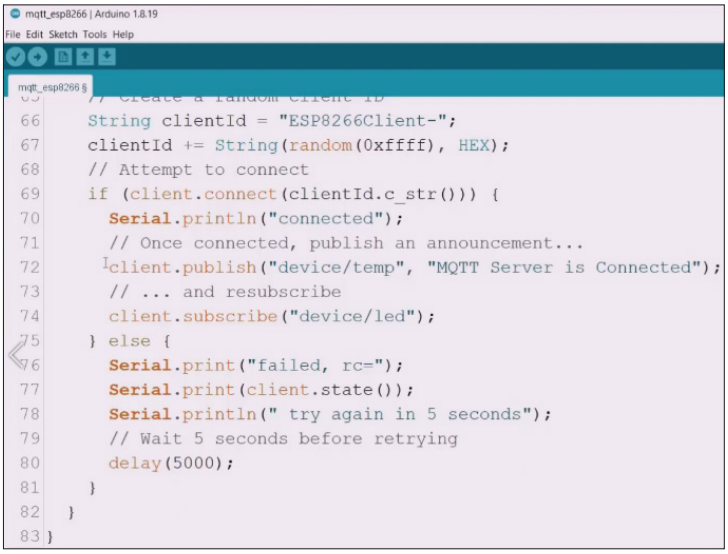


Figure 33.9: Editing "client.publish".

Therefore, that is a message that will be displayed even if you include this line of code. It may not seem very meaningful, but the true essence lies within the loop function.

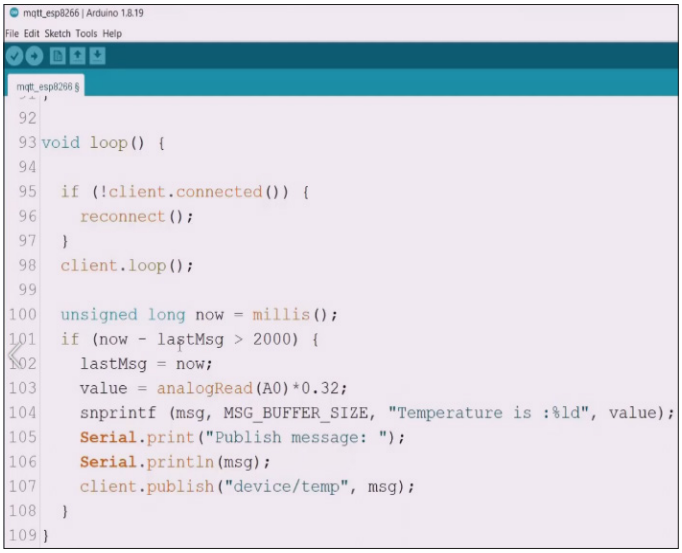


Figure 33.10: Publishing the temperature to the MQTT server.

In the loop function, the code continuously sends the message "Hello world," incrementing the value variable each time. To adapt this code for sending temperature data from an LM35 sensor connected to pin A0 on the ESP8266, we will modify the global variable "value." Instead of incrementing, we set "value" equal to analogRead(A0) multiplied by 0.32, converting the sensor reading to degrees Celsius. This temperature value is then printed on the serial monitor with the label "Temperature is."

The modified code will publish this temperature data to the MQTT server using the "device/temp" topic, as shown in Figure 33.10. The MQTT server, with the IP address 5.196.95.208, will receive and process the temperature data.

For LED control, the "device/led" topic is important. Subscribing to "device/led" allows the ESP8266 to respond to messages from the server. Sending "1" to "device/led" turns on the LED, while "0" turns it off. Thus, the ESP8266 effectively controls the LED and publishes temperature data on "device/temp".

We successfully connected the ESP8266 to the laptop, but we still need an MQTT client. We recommend installing the MQTT client called MQTTLens, which is a simple and easy-to-use client that runs on Google Chrome. To install it, search for MQTTLens on Google Chrome, and you will find it as a Google Chrome extension. Please note that MQTTLens only works on Google Chrome, so make sure you have it installed. To install MQTTLens, simply click on it and it will take you to the Google Chrome app page. Click on "Add to Chrome" as shown in Figure 33.11, and it will be quickly installed. It may ask you to log into your Google account, so just click on "Add app" and then close it. You can see the progress of the installation in

the lower-left corner of your screen. Once it is added to Chrome, you will be able to see MQTTLens in your extensions.

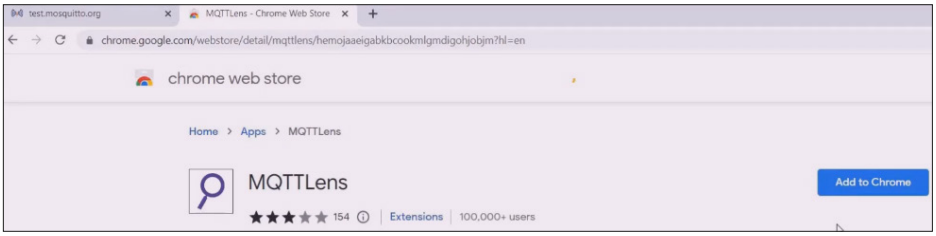


Figure 33.11: Adding MQTTLens to Chrome.

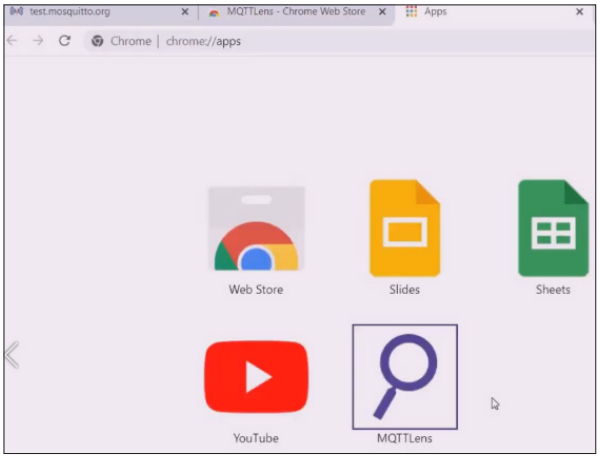


Figure 33.12: Installed MQTTLens app.

As you can see in Figure 33.12, when we click, the software opens, allowing us to see that MQTT Lens is the software or MQTT client we will utilize to control the ESP8266. To set up MQTTLens, click on the plus button illustrated in Figure 33.13.

Figure 33.13: Clicking on the plus button.

To set up a connection, we are required to provide a connection name. Let's name it BinaryUpdates. We also need the IP address of the server. We prefer inputting the server's numerical address rather than its name. Simply copy the IP address from our code and paste it under the "Hostname" section. We want to use MQTTlens as the MQTT client to connect to the MQTT server on port 1883. Once everything is in place, we can scroll down without making any changes to the advanced settings. Finally, click on "CREATE CONNECTION" as shown in Figure 33.14.

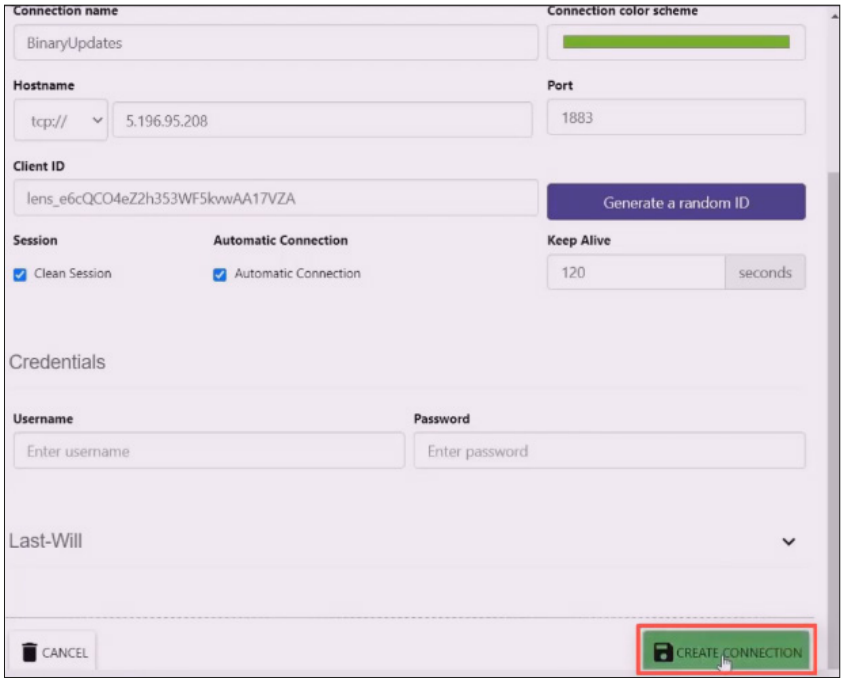


Figure 33.14: Creating the connection.

By hovering over your mouse, you can see that a connection has been established when it displays a green color, as depicted in Figure 33.15. Conversely, if it remains red or any other color, it indicates that something is amiss and is not functioning properly.

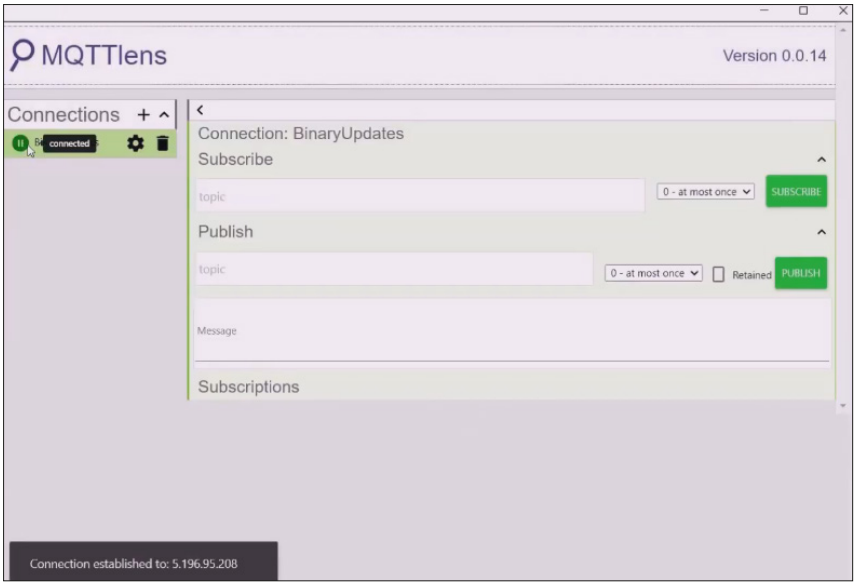


Figure 33.15: The connection created with green color.

Let us go ahead and open the Arduino IDE side by side, so you can see the process. Make sure the code is ready, go to tools, check that the port and board are properly selected, and upload the code. Now we can monitor the temperature sensor data and control the built-in LED on the ESP8266. The code is uploading and once it is done, go to tools and open the serial monitor. Set the baud rate to 115200. Now you will start seeing the data being displayed. To confirm that the ESP8266 is connected to the internet, press the reset button on the device. It will show that it is connecting to the access point and then indicate that Wi-Fi is connected, and an IP address has been obtained. It will also try to establish a connection with MQTT. Once connected, the data will start streaming as illustrated in Figure 33.16.

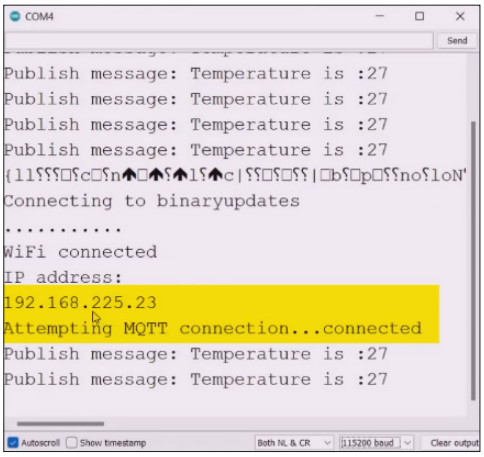


Figure 33.16: Coming up published messages.

To begin, we will minimize the Arduino IDE and keep the serial monitor open. In our Arduino code, we have the topic "device/temp" where our sensor data is transmitted. To view the temperature data, we simply navigate to the subscribe section on MQTTlens and right-click to paste "device/temp" as the topic for subscription. By subscribing, we can now see the continuous stream of temperature data, which is indicated by the increasing count and timestamp. Additionally, the temperature can be seen in the MQTT and serial monitor. This functionality is working correctly. Moving on to controlling the LED, if we take the topic for the LED as "device/led", we can publish the message "one" to turn the LED on. It is important to not press unnecessary keys and only send the value "one" on the "device/led" topic. Clicking on "Publish" will trigger the arrival of the message, resulting in the LED lighting up. This can be seen on the hardware side and in the serial monitor where "one" is received. Similarly, sending the message "zero" on the same topic will turn the LED off, and we will receive the message "zero" as depicted in Figure 33.17.

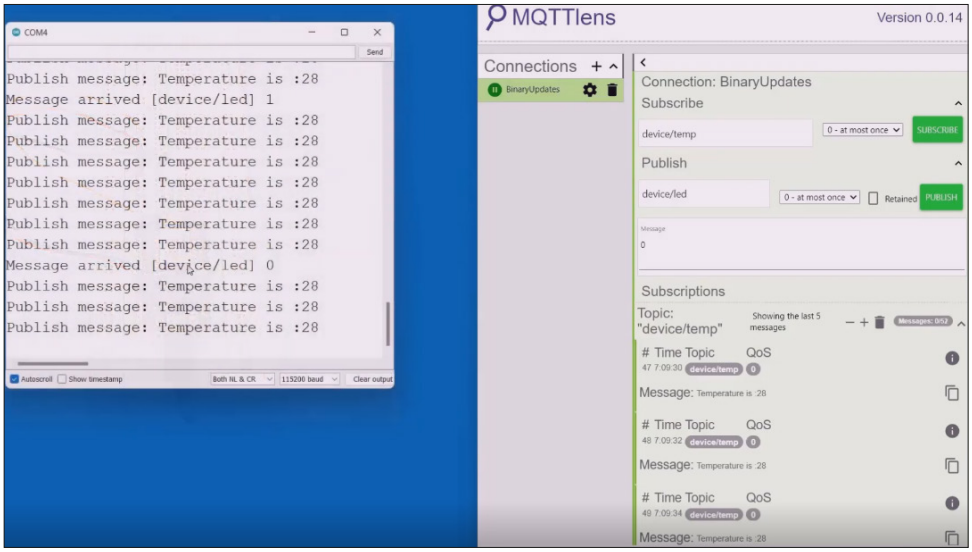


Figure 33.17: Publishing the messages in MQTTlens and serial monitor.

To operate the ESP8266 through a mobile app, it is necessary to download and install the Android app called "MY MQTT", as shown in Figure 33.18.

Figure 33.18: The MY MQTT app icon.

When you open this app, there is a view that allows you to select what MQTT can access. In the lower-right corner, there is a "Continue" button that you click to proceed. A pop-up appears and you confirm by clicking "OK". Now, in the top-left corner, there is a dashboard and a back icon. When you click on the back icon, it takes you to the settings. In the settings, you will find a setting button. Clicking on it allows you to enter various parameters. The first parameter is the Broker URL, which is the IP address of our MQTT server (0.5.196.95.208) that MQTT should connect to. Below that, there is a port number (1883) for MQTT. You do not need to fill in the username and password, just click on "Save". Once saved, it will show a message saying that the settings have been saved successfully. In the top-left corner, there is a back icon again. Clicking on it will show a green indication and the MQTT server IP address (5.196.95.208) displayed in Figure 33.19.

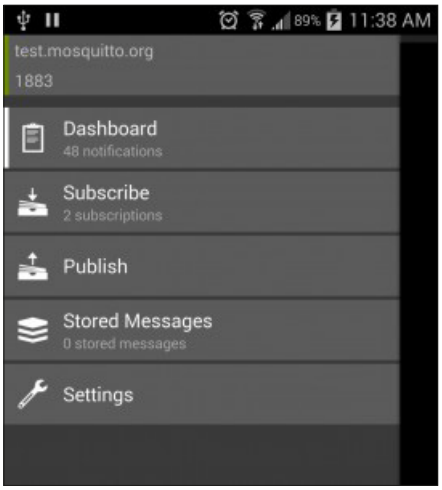


Figure 33.19: MY MQTT app connection to MQTT server.

We need to look at our hardware setup. We have an ESP8266, and there is an onboard LED. Before publishing any messages, let us click the reset button once. Now, we will navigate to the mobile app's publish section. In the topic field, we will input "device/led," and for the message, we'll enter '0' to publish the message '0' on the 'device/led' topic. Clicking "Publish", you will notice the onboard LED turns off. This demonstrates how to turn off the LED.

Next, if we publish the message "1" and click "Publish" again, you will observe the LED lighting up. Repeating the process by publishing "0" turns the LED off, and publishing "1" lights it up once more. This illustrates how to control the LED using published messages.

Now, let us explore the subscribe feature. Clicking the top-left back button, we will navigate to the subscribe section. Here, we'll type "device/temp" as the topic for receiving temperature data. Clicking "Add", you will see the topic subscribed and added. Returning to the dashboard by clicking back, you can see the streaming data on the dashboard, as depicted in Figure 33.20 and Figure 33.21, respectively.

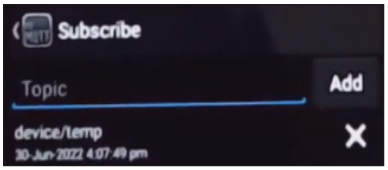


Figure 33.20: Subscribing the device/temp topic

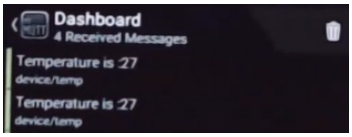


Figure 33.21: Data streaming on "Dashboard".

Chapter 34 • ThingSpeak IoT Cloud with Ethernet/ ESP8266

In this section, we will explore how to transmit sensor data from the Ethernet/ESP8266. We will connect a temperature sensor, LM35, to the Ethernet/ESP8266 and then send the data to the ThingSpeak cloud platform. ThingSpeak is a well-known IoT platform that enables data transmission to a server, where it can be displayed in attractive widgets or stored in a database. It is user-friendly, and if you already have a ThingSpeak account, you can access the dashboard, which is depicted in Figure 34.1.

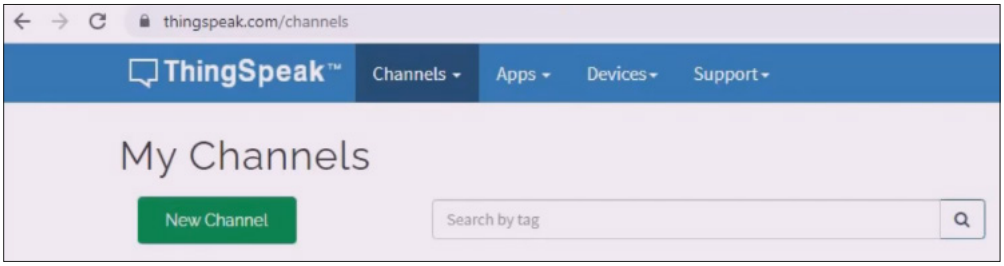


Figure 34.1: The ThingSpeak dashboard.

In the world of ThingSpeak, a channel is required to display data from the Ethernet/ ESP8266, specifically temperature data. To create a channel, we must click on the "New channel" option and fill out a form with the desired channel name. For our purposes, we will name it "Temperature Monitoring" and provide an optional description, such as "LM35 Temperature Sensor Data". Within ThingSpeak, there is a concept called "Field". Multiple fields can be enabled, depending on the number of sensor data points to be sent. In this case, we only have one temperature sensor connected to Ethernet/ESP8266, so we are only interested in displaying the temperature value. "Field 1", as shown in Figure 34.2, is automatically selected, and we can proceed to click the "Save channel" button.

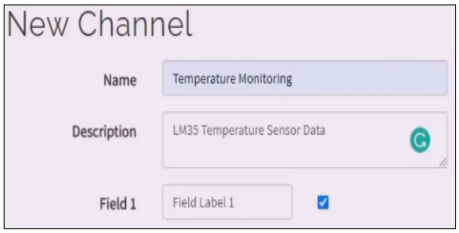


Figure 34.2: Selecting the "Field 1".

Now, upon creating this channel, you can see a "Field 1" chart. Within the chart, you will find your channel name, unique channel ID, author details, and the access is currently set to private. The specifics of this are not important now. If you look at this, it will show you a private view which includes a chart for "Field 1", as depicted in Figure 34.3. Currently, "Field 1" does not have any sensor value. However, once a real sensor value is obtained, you will be able to view temperature data in that field. Additionally, you can add more widgets and explore other features, which we will discuss at the right time. So far, everything is going

well. Now, let's move on to the programming aspect of the microcontroller. We will be using an LM35 temperature sensor and connecting its output to the A0 pin. Since the LM35 is an analog temperature sensor, one of its pins will be connected to 3.3 V to provide power.

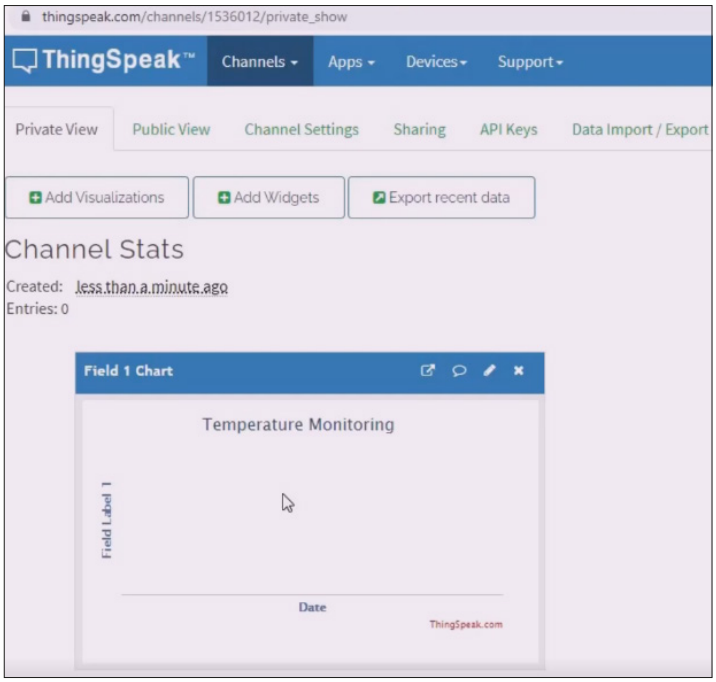


Figure 34.3: A chart of a "Field 1".

Furthermore, we need to connect another pin of the temperature sensor to the ground pin to set up the connection between LM35 and ESP8266. Next, we must launch the Arduino IDE. If you already have the Arduino IDE, we need to go to the sketch include library and select "Manage libraries" because we want ESP8266 to communicate with ThingSpeak. Therefore, we need to install the ThingSpeak libraries to avoid writing lengthy and complex bare-metal code or hard-coded HTTP requests. To keep it simple, we can search for "ThingSpeak" in the library search bar. Once the search is complete, we will find the ThingSpeak library developed by MathWorks Corporation, the same company known for the popular MATLAB software. We select the library and click on the "install" option, as depicted in Figure 34.4. After clicking "Install," we need to patiently wait for the ThingSpeak library to be installed for ESP8266. Once the installation is complete, we can see that it has been successfully installed.

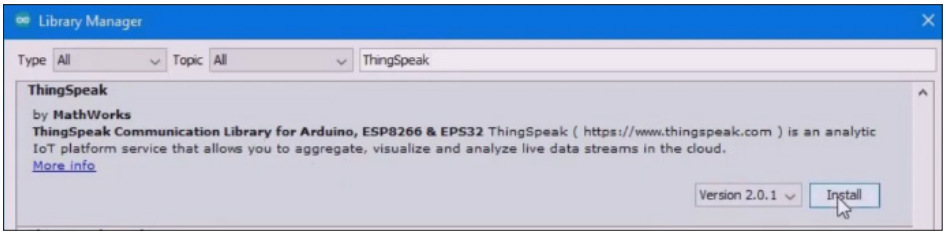


Figure 34.4: Installing the ThingSpeak library.

Therefore, we need to show some respect and close the "Library Manager". Afterwards, we can proceed to create a new project. As a result, a default project is automatically generated. We have already written some code, and now we will explain it line by line. Before doing anything else, it is important to save the file as good practice. Click on "Save" and name the file as "ThingSpeak_ESP8266". Now the program is saved. Our ESP8266 communicates with the ThingSpeak cloud platform via Wi-Fi, so we need to enter the SSID (Wi-Fi access point name) in the code. Additionally, we need to provide the password. In the code, there is an integer variable named "pin," which is set to A0. This is because we will connect the LM35 temperature sensor to the ESP8266 via pin A0. Furthermore, we create an instance of the Wi-Fi client class, as shown in Figure 34.5. This client object serves as the hardware client that communicates with the ThingSpeak server.

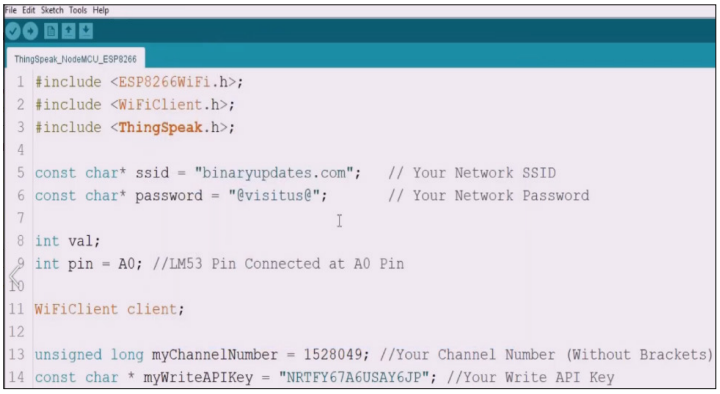


Figure 34.5: Beginning header codes for ESP8266.

Afterward, you will notice that our channel is assigned a distinct channel number. Essentially, every ThingSpeak channel requires a unique channel number, or ID. Therefore, we need to navigate to our ThingSpeak dashboard and select the API keys section. Within the API keys, you will find a write API key that needs to be copied, as indicated in Figure 34.6.

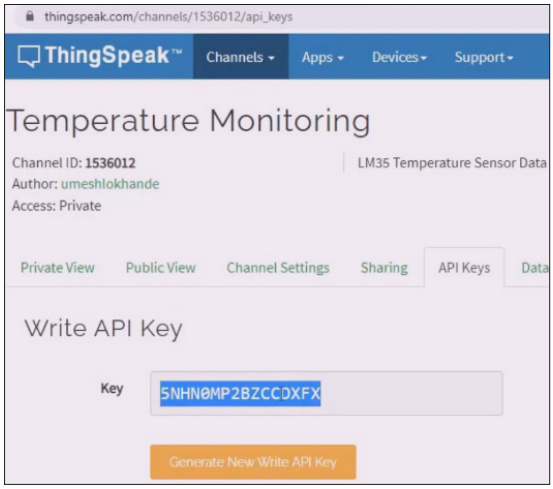


Figure 34.6: Getting the "Write API Key".

So, the next step is to return to the ThingSpeak channel and retrieve the channel ID. Once we have it, we can proceed to replace it within the code. We need to obtain "Write API Key" and insert it. Another part involves initializing the serial connection with a baud rate of 9600, followed by a delay. The code then connects to the internet via Wi-Fi using the SSID and password in Figure 34.7. If you have experience with Arduino programming, you may recognize the usage of the global variable "val". This variable is defined at the beginning and is used to read the A0 pin. To convert the data from the LM35 temperature sensor from analog to degrees Celsius, we multiply the read value by 0.322265. This conversion is necessary because the microcontroller receives ADC counts from the analog sensor and needs to convert it to degrees Celsius. The reason for multiplying by this specific number is due to the voltage ranges of the LM35 and the node MCU. The LM35 operates between four and 30 volts, while the node MCU functions with a voltage of 3.3 volts.

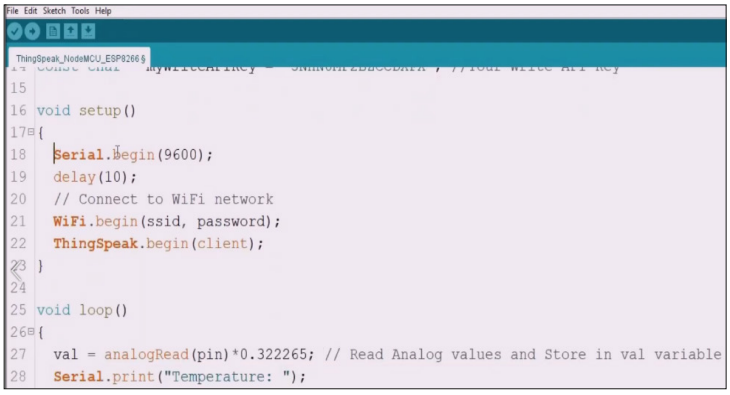


Figure 34.7: The setup section code.

```
// Connect to WiFi network
WiFi.begin(ssid, password);
ThingSpeak.begin(client);
}

void loop()
{
val = analogRead(pin)*0.322265; // Read Analog values and Store in val variable
Serial.print("Temperature: ");
Serial.print(val);                // Print on Serial Monitor
Serial.println("*C");
delay(1000);
ThingSpeak.writeField(myChannelNumber, 1,val, myWriteAPIKey); //Update in ThingSpeak
delay(100);
}
```

Figure 34.8: The loop section code.

So, we have just calculated this number and stored the temperature sensor value in the "val" variable. After printing it on the serial monitor and waiting for one second, we call the "writeField" function, a part of the ThingSpeak library we have installed. This function pushes the data into our unique ThingSpeak channel, with the channel number and write API key being specific to our temperature monitoring channel, as created earlier. The "val" variable represents the data we want to send to the server.

To execute this, we save the file, check that the correct board and COM port are selected in the "Tools" menu, and then click on the upload button in the top-left corner. Before finalizing, it is crucial to open the serial monitor to observe the temperature sensor data. Confirming its appearance on the serial monitor, we can cross-verify the data in the ThingSpeak channel.

ThingSpeak, supporting the HTTP protocol, uses the "ThingSpeak.h" header file, which implements this protocol. ESP8266 communicates with ThingSpeak by sending data in the form of HTTP requests. The ThingSpeak library streamlines this process, avoiding the need to write HTTP requests from scratch.

Heading to the tools menu and opening the serial monitor, we can witness the temperature data appearing on the ThingSpeak channel, indicating a successful communication between ESP8266 and the ThingSpeak server. Figure 34.9 shows an example where we're receiving a temperature reading of 28 degrees Celsius.

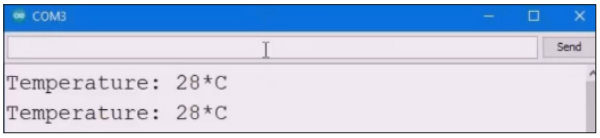


Figure 34.9: The temperature data on serial monitor.

If we navigate to our ThingSpeak dashboard, we need to go to the private view because the channel is set as private by default. This means that the data will not be shared with everyone. Once in the view, we can observe the temperature displayed. It is worth noting

that when comparing it to the window side by side, as shown in Figure 34.10, there might be a noticeable delay in the arrival of the data.

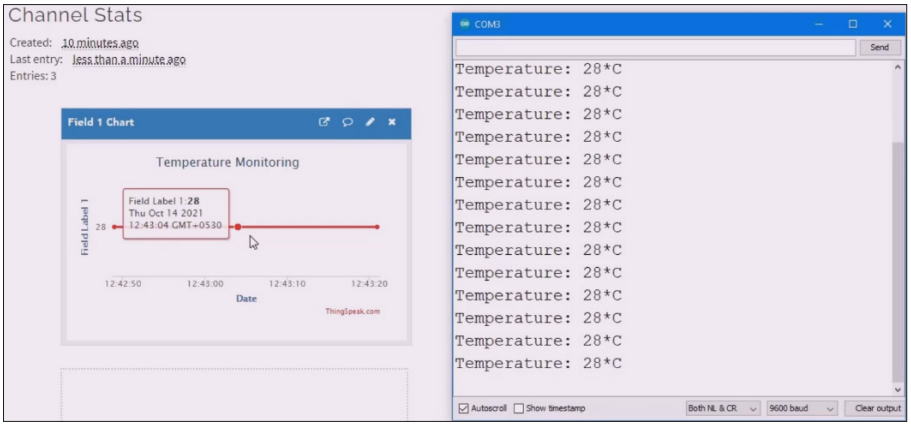


Figure 34.10: The side-by-side view of "Field 1" chart and serial monitor.

In the free version of ThingSpeak, there is a limitation that data can only be sent every 15 seconds. This means that the data does not appear every second and is bound to a 15-second interval. This is how the code operates. If you want to display the temperature data in visually appealing widgets instead of just a chart, you need to click on the "Add Widgets" option, as illustrated in Figure 34.11.

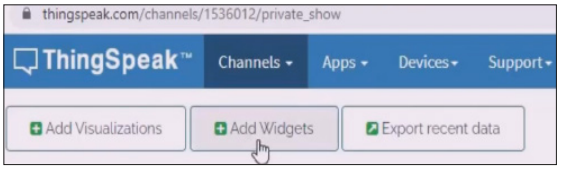


Figure 34.11: Adding widgets.

Afterwards, we need to choose the widget. Suppose for this example, we opt for a gauge. Therefore, select the gauge, continue by clicking on "Next", and proceed to assign the name to the gauge, as depicted in Figure 34.12 and Figure 34.13. We prefer to label it as "Temperature".

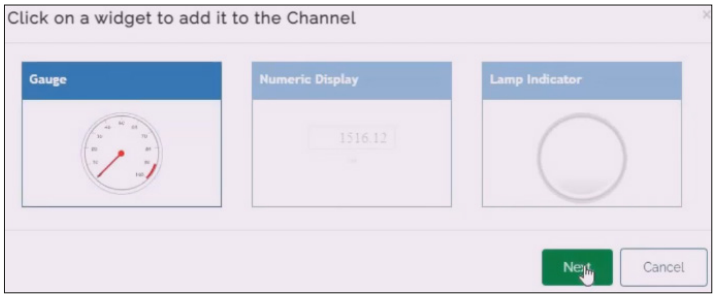


Figure 34.12: Selecting the gauge.

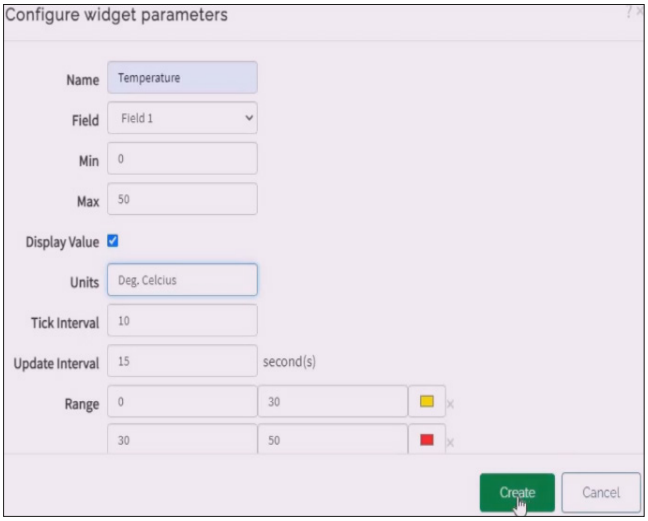


Figure 34.13: Configuring widget parameters.

After obtaining LM35 temperature data in "Field 1," we aim to display this data with a range between 0 and 50 degrees Celsius. To enhance aesthetics, we have added color coding. Within the range of 0 to 30 degrees Celsius, the color is set to yellow, and for values between 30 to 50 degrees Celsius, we have chosen a random color, let's say red. You have the flexibility to customize these colors based on your preferences.

Additionally, you can add more details like units; for example, we have set it to "degree Celsius." You have the creative freedom to design it according to your liking. Clicking "Create" (as shown in Figure 34.13), you will now have a visually appealing widget. As illustrated in Figure 34.14, the data will appear in this widget, providing a clear and aesthetically pleasing representation of the temperature readings.

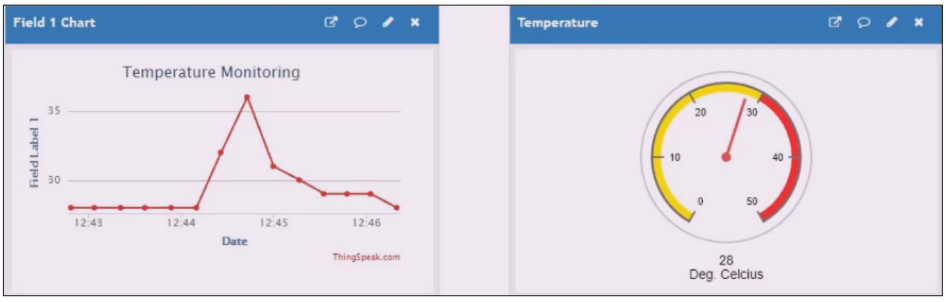


Figure 34.14: Added gauge widget.

To share this data with someone, you need to make sure that the channel is set to public. To do this, you must go to the shading option and select "Share channel view with everyone", as shown in Figure 34.15.

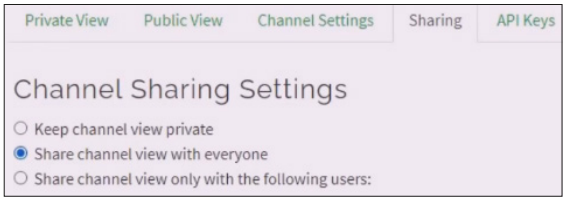


Figure 34.15: Making the channel public.

This is essentially how the channel will appear from now on. In the public view, the gauge will not be visible. However, if you want to include the gauge, we can add it again and adjust its settings, similar to what was done in Figure 34.13. Consequently, our public view will be identical to our private view. Now, the URL at the top of the browser, as shown in Figure 34.16, will be accessible to the public.

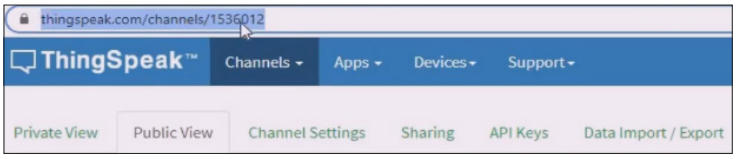


Figure 34.16: The public view URL.

If we share this URL with our friends, they will be able to see our data in real-time. Let us demonstrate by opening a separate browser, such as Firefox, where we are not signed in to our ThingSpeak account. We will simply paste the copied URL from our ThingSpeak channel. This URL is unique to our channel and is public. Once we hit enter, we can see that the channel is now public, as depicted in Figure 34.17.

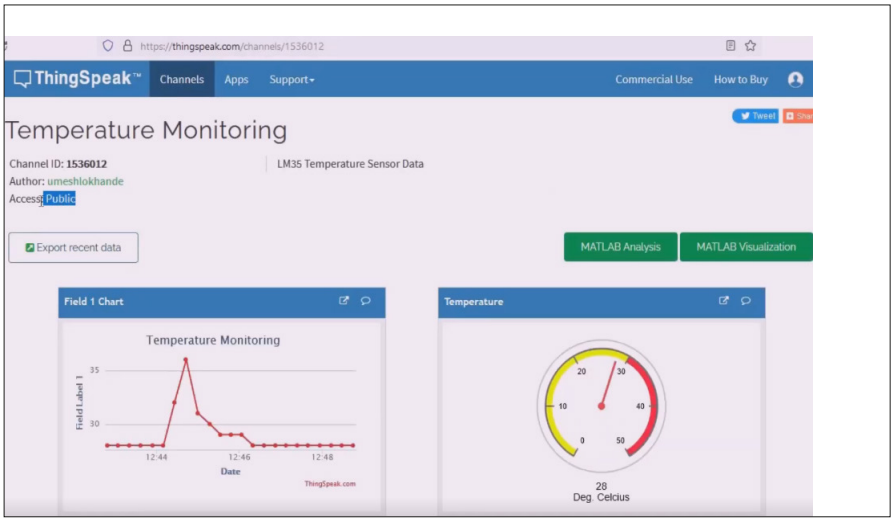


Figure 34.17: Opening the public URL in Firefox.

Regardless of location, anyone in the world can view this information on ThingSpeak without needing to log in. This means that even if we have not signed in, we can still access the data and make it visible to the world. This is how you can create an attractive dashboard with a unique URL and other features.

Chapter 35 • The Modbus TCP/IP Communication using Node-RED

Node-RED is a visual programming tool for wiring together Internet of Things (IoT) devices, APIs, and online services. It provides a web-based flow editor that allows users to create and deploy flows of logic in a browser-based interface. Node-RED is built on top of Node.js and provides a browser-based editor that makes it easy to create and deploy flows. It supports a wide range of IoT devices, APIs, and online services, making it a popular choice for IoT development and automation. Node-RED is open-source and has a large community of contributors, which makes it easy to find help and support. In this section, we are going to implement the Modbus TCP/IP communication using Node-RED.

To install Node-RED on Windows, follow these steps:

Step 1: Install Node.js
Node-RED requires Node.js to be installed on your machine. You can download the latest version of Node.js from the official website and install it on your Windows machine.

Step 2: Install Node-RED
Once Node.js is installed, you can install Node-RED using the Node Package Manager (NPM) by running the following command in the command prompt or terminal:

```
npm install -g --unsafe-perm node-red
```

This command will install Node-RED globally on your machine.

Step 3: Launch Node-RED
After the installation is complete, you can launch Node-RED by running the following command in the command prompt or terminal:

```
node-red
```

This will start Node-RED and open the Node-RED editor in your default browser.

Step 4: Accessing Node-RED
To access the Node-RED, open your web browser and navigate to "http://localhost:1880". This will display the Node-RED editor, where you can start creating your first flow.

That's it! You have successfully installed Node-RED on your Windows machine. An overview of these steps is shown in Figure 35.1.

To install Modbus TCP/IP in Node-RED, you should open the Node-RED and launch your Node-RED editor by navigating to "http://localhost:1880" in your web browser, and install the Modbus TCP/IP node. In the Node-RED editor, click on the menu icon in the top right corner and select "Manage palette", as depicted in Figure 35.2. In the "Install" tab, search for "node-red-contrib-modbus" and click on the "Install" button next to it, as illustrated in

Figure 35.3. This will install the Modbus TCP/IP node into your Node-RED environment as shown in Figure 35.4.

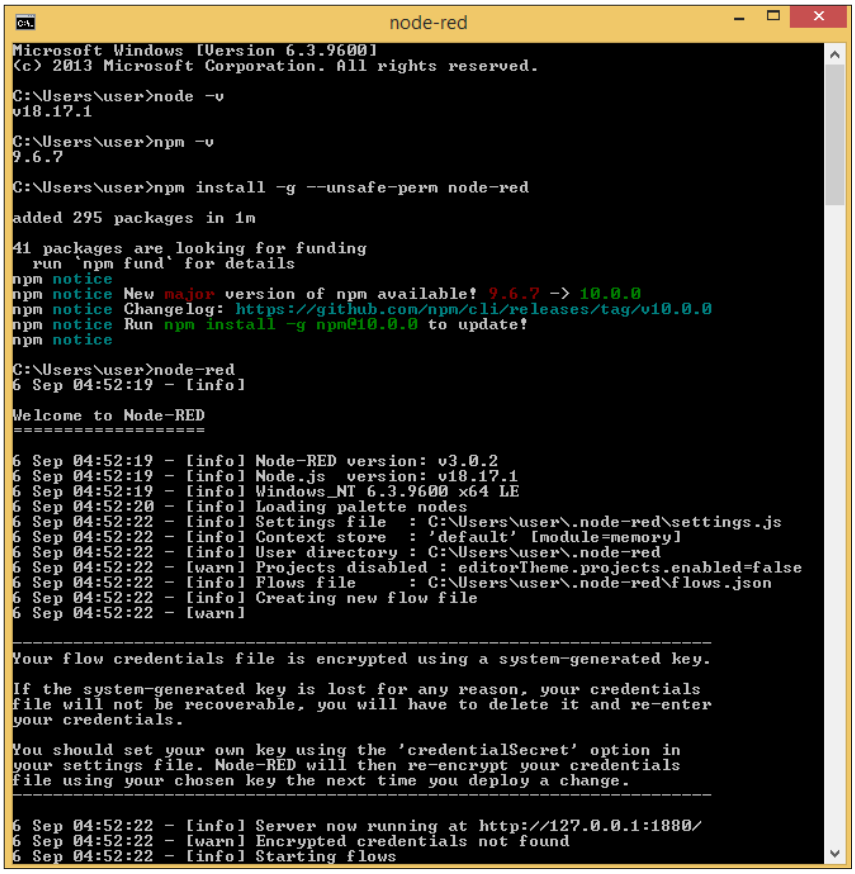


Figure 35.1: An overview of the Node-RED installation steps.

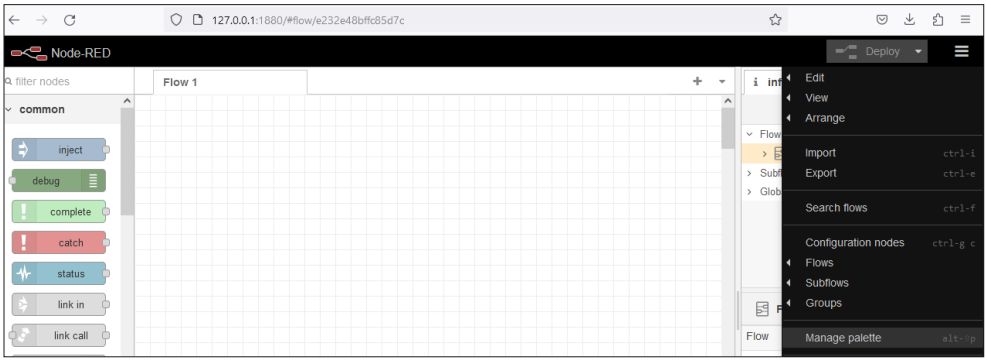


Figure 35.2: Selecting "Manage palette".

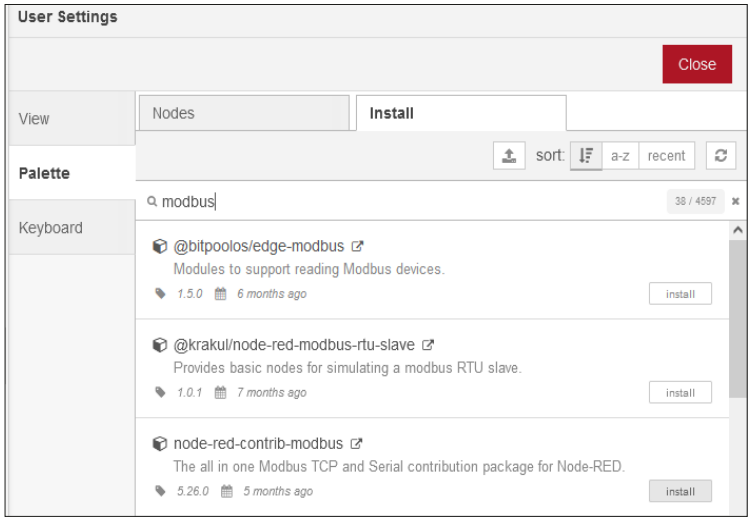


Figure 35.3: Searching for and installing the "node-red-contrib-modbus".

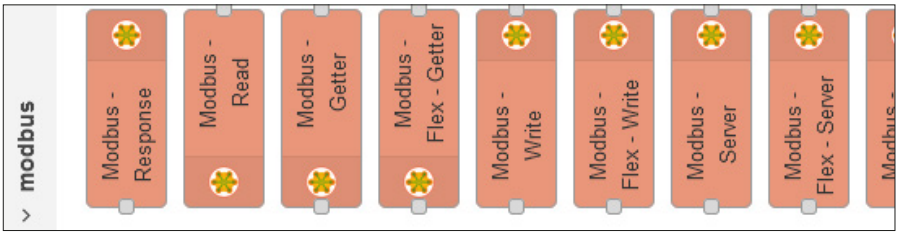


Figure 35.4: Installed Modbus TCP/IP node into the Node-RED environment.

To begin working with Modbus TCP/IP in the Node-RED environment, it is recommended to retrieve the IPv4 address of the Windows system using the "ipconfig" command in the Windows command environment (cmd), as shown in Figure 35.5. Please note that the Node-RED program must be temporarily stopped to execute the "ipconfig" command. Once the IPv4 address has been obtained, you can reactivate the Node-RED program by executing the "node-red" command, as depicted in Figure 29.4.

To start working within the Node-RED environment, you need to bring the Modbus Read node into the interface and double-click on it to access the Edit Modbus-Read node window. The settings for this window should be configured according to Figure 35.6.

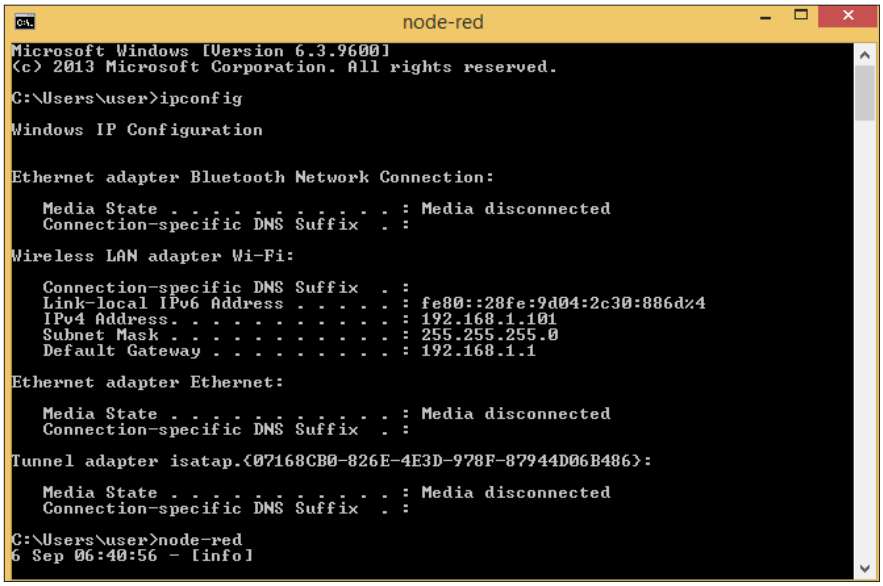


Figure 35.5: Obtaining the IPv4 address.

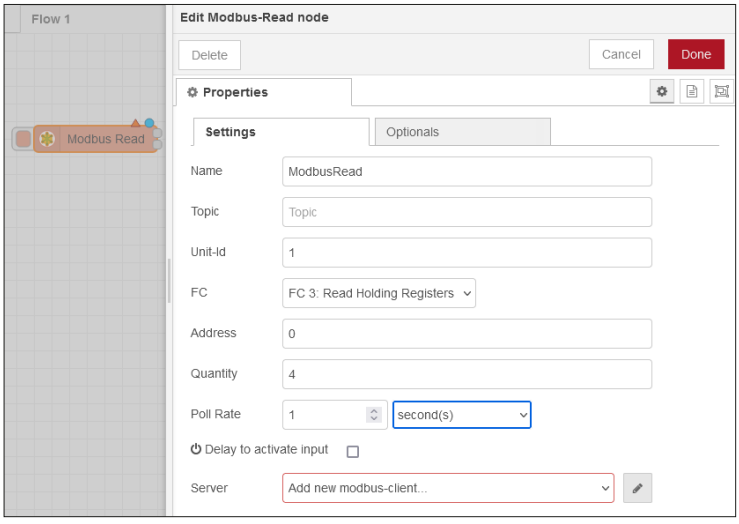


Figure 35.6: The "Edit Modbus-Read node" window settings.

To conclude, we select the pencil icon associated with "Add new modbus-client..." to access the window for configuring the new modbus-client. In this window, we configure the Properties tab as shown in Figure 35.7. Afterward, we click the "Add" button and proceed to the "Edit Modbus-Read node" window, where we click the "Done" button.

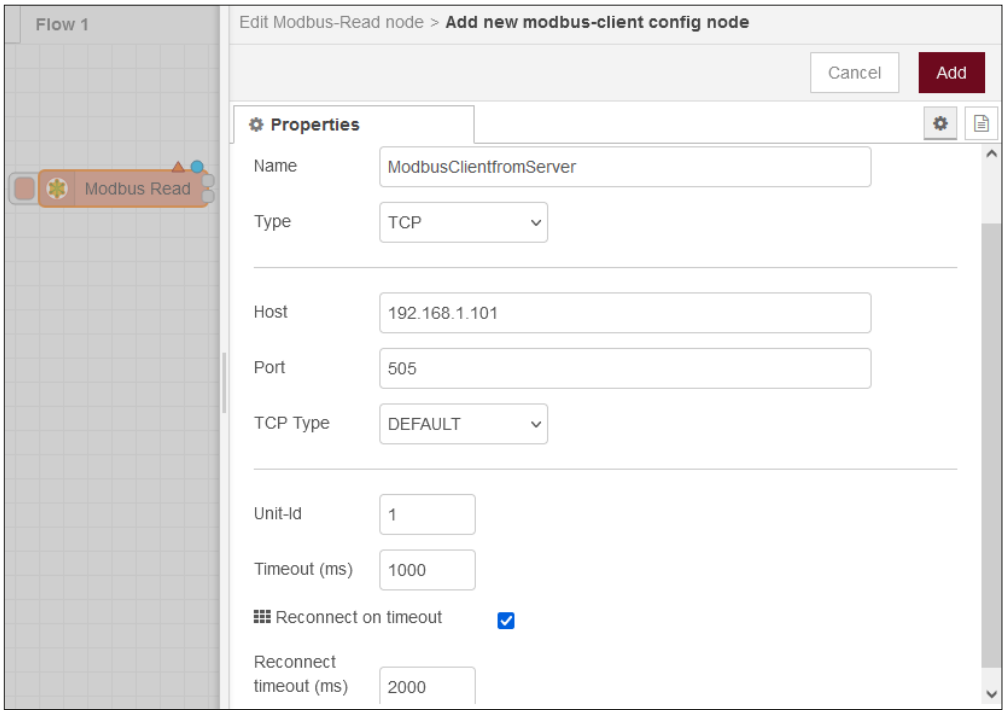


Figure 35.7: "Add new modbus-client config node" window settings.

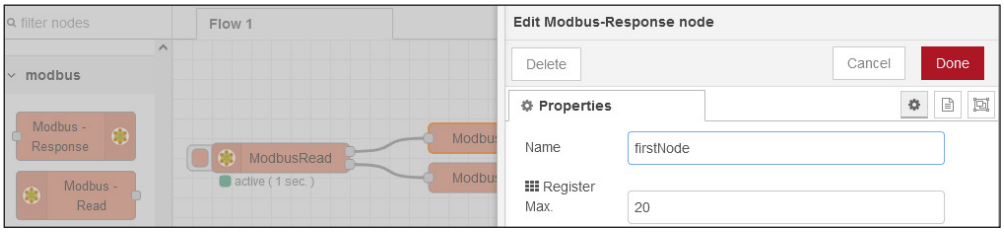


Figure 35.8: "Edit Modbus-Response node" for the first output.

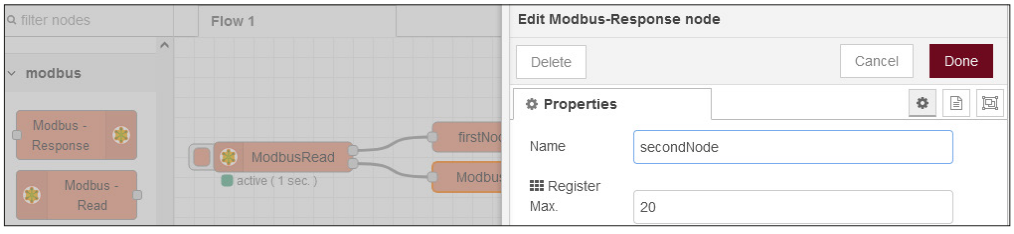


Figure 35.9: "Edit Modbus-Response node" for the second output.

To receive a Modbus response in Node-RED, you can utilize the Mod-Response node. To the two output ports in thebusRead node, we make of two Modbus- nodes referred to as "first" and "secondNode" as depicted in Figure 35.8 and Figure 35.9 respectively. Following that, the "Deploy" button is clicked. We set up communication between the Modbus Slave

software and Node-RED using the TCP/IP protocol, employing the same port, ID, and IP address, along with the holding register (F = 03) previously selected in the ModbusRead node settings. Subsequently, an integer is entered in the address 0, which was previously chosen in the ModbusRead node, as shown in Figure 35.10. In Figure 35.10, the integer is represented as an array comprising four elements since the Quantity was set to four in the ModbusRead node settings.

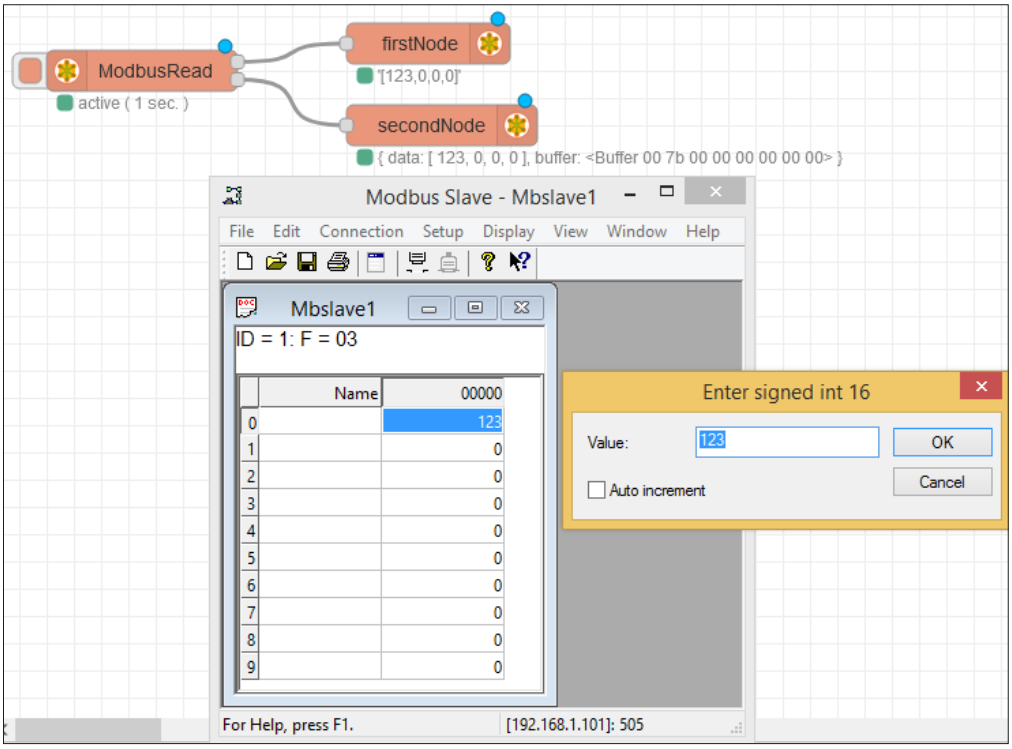


Figure 35.10: Using the Modbus Slave software to communicate with the Node-RED.

The Debug node is a pre-installed feature in Node-RED that displays messages while constructing and executing flows. It is widely used for identifying and resolving problems in Node-RED flows, as well as visualizing data movement within the flow. To see the flow of data, we employ two Debug nodes named "firstout" and "secondout" which are connected to the outputs of the ModbusRead node, as illustrated in Figure 35.11 and Figure 35.12 correspondingly.

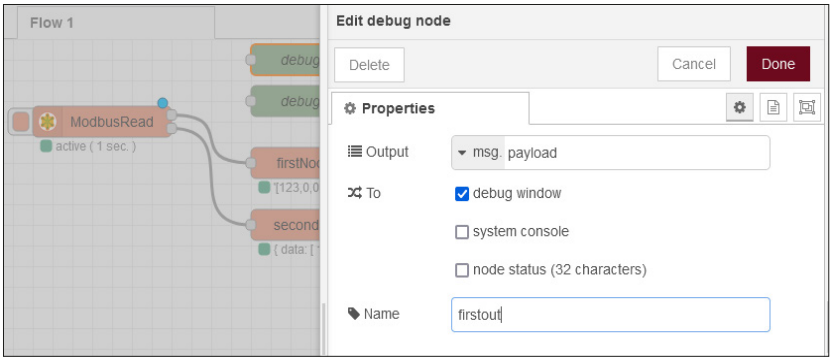


Figure 35.11: "Edit debug node" called "firstout".

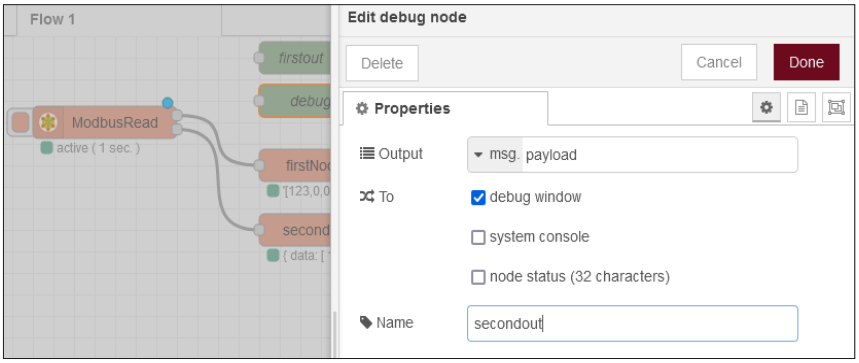


Figure 35.12: "Edit debug node" called "secondout".

Node-RED provides a special type of node called the Function node, which allows users to create customized functions to manipulate messages within their flows. These function nodes are beneficial for performing tasks such as data transformations, message filtering, and implementing complex logic. Users can use standard JavaScript functions and libraries in their function nodes to manipulate messages as required. To separate the array elements of the ModbusRead node output, we connect the ModbusRead node's first output to the function node, as shown in Figure 35.13. Additionally, the debug window displays relevant results related to the Debug nodes in Figure 35.13.

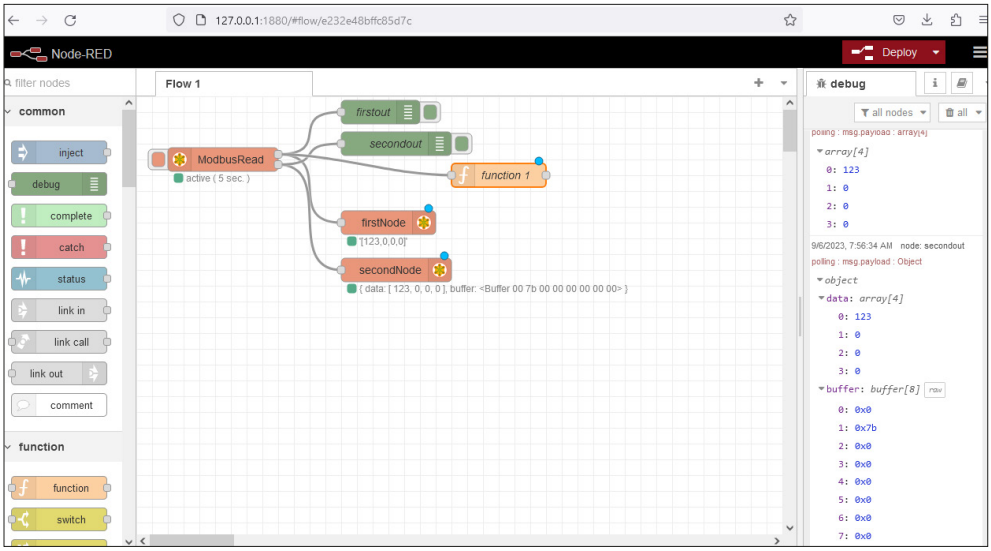


Figure 35.13: Connecting the ModbusRead node first output to the function node.

To input the JavaScript codes shown in Figure 35.14, we simply need to double-click on the function node.

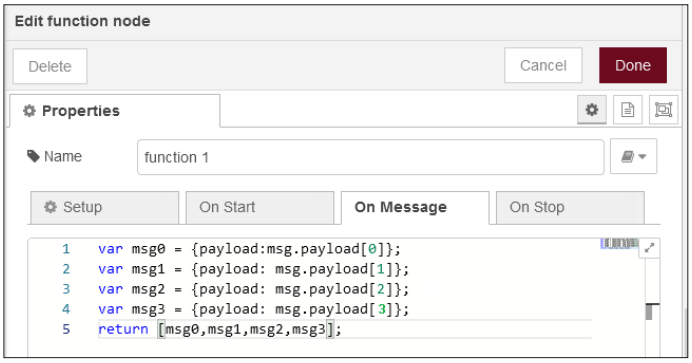


Figure 35.14: JavaScript codes in the "Edit function node".

In "Setup" tab of the Edit function node window, we set the outputs to four since our array has four elements as depicted in Figure 35.15.

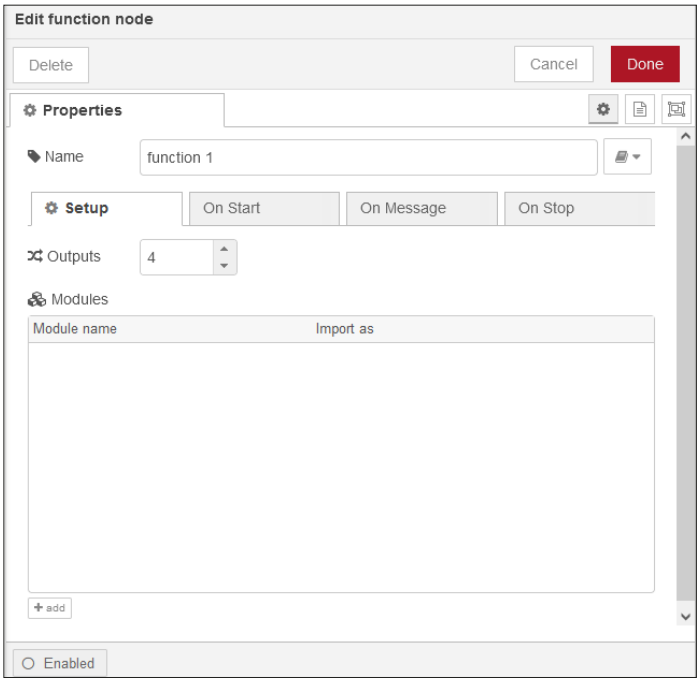


Figure 35.15: Setting the Outputs as four.

By attaching two Debug nodes to the function node's outputs and after clicking on the "Deploy" button, the results can be seen on the debug window, as illustrated in Figure 35.16.

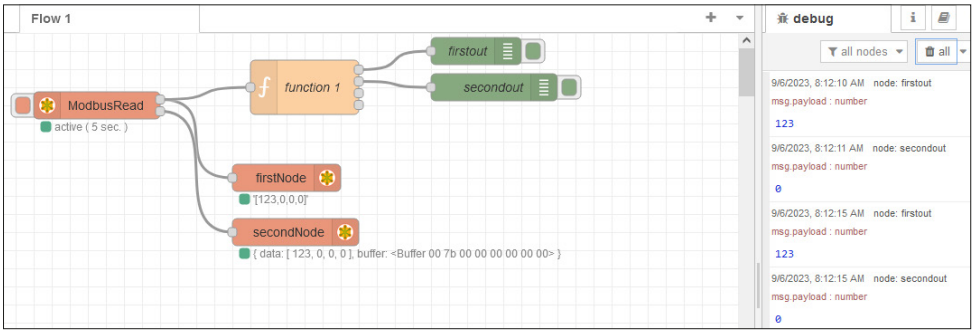


Figure 35.16: Connecting two Debug nodes to the two outputs of the function node.

By using the Modbus Slave software, we can input an integer into address 1. Subsequently, the debug window will display the entered integer, as depicted in Figure 35.17.

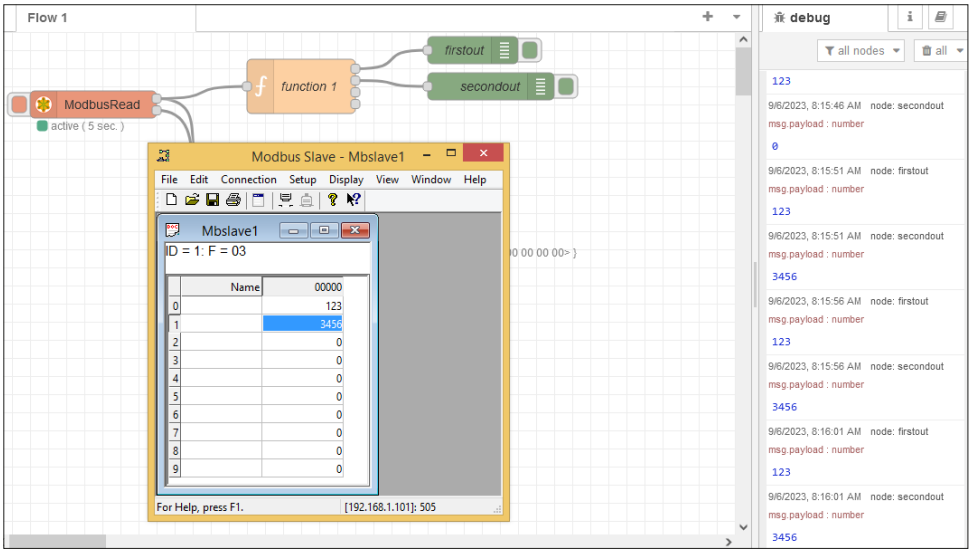


Figure 35.17: Adding an integer in the address 1.

To showcase a 32-bit integer, you need to insert the specified JavaScript code into the "Edit function node" window, as depicted in Figure 35.18.

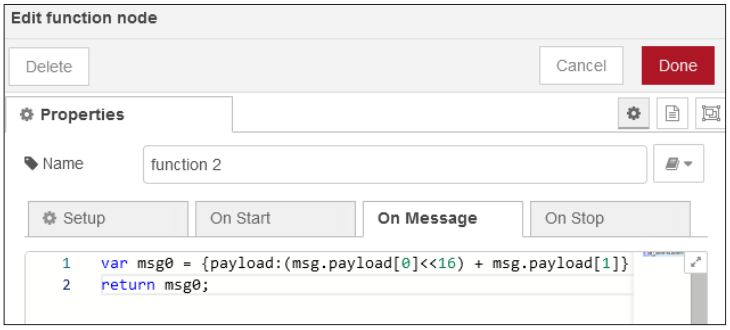


Figure 35.18: Adding JavaScript codes to display a 32-bit integer.

To present a 32-bit integer, we incorporate a Function node with JavaScript codes, as illustrated in Figure 35.18. This node is then connected to a Debug node. Afterward, we click on the Deploy button. To see the outcome on the debug window, we apply the Modbus Slave software and include a 32-bit integer at address 0, as demonstrated Figure 35.19.

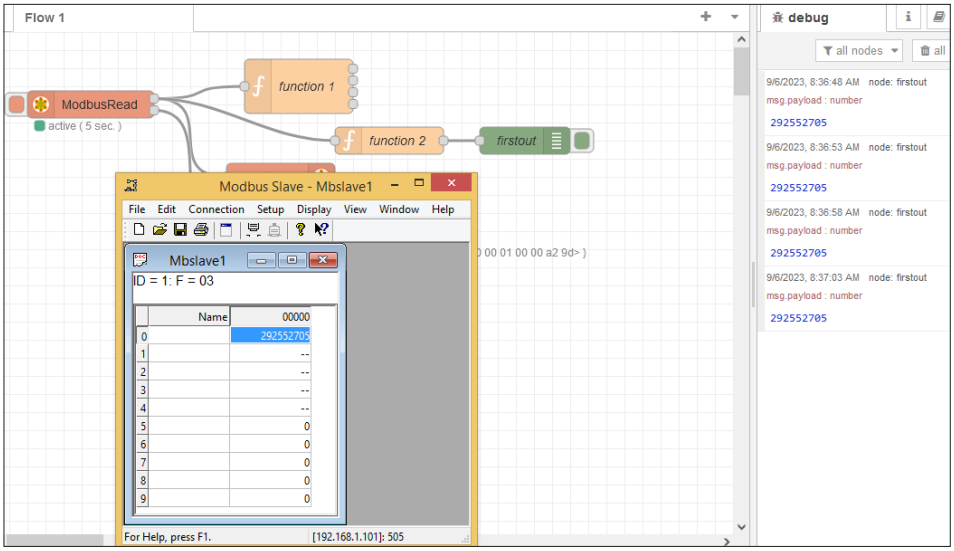


Figure 35.19: Adding a 32-bit integer in the address 0.

To display a float number in the Node-RED editor, you need to follow these steps: click on the menu icon located at the top right corner, choose "Manage palette", switch to the "Install" tab, search for "float" and locate the "node-red-contrib-float". Finally, click on the "Install" button as shown in Figure 35.20.

Figure 35.20: Clicking on the "Install" button.

Once the "node-red-contrib-float" is installed, a new node called "toFloat" will appear in the nodes section of the Node-RED editor. To show a float number, we place the "toFloat" node between the "function2" and "firstout" nodes in Figure 35.19 and input a float number at address 0 using the Modbus Slave software. The resulting output can be seen in the debug window, as illustrated in Figure 35.21.

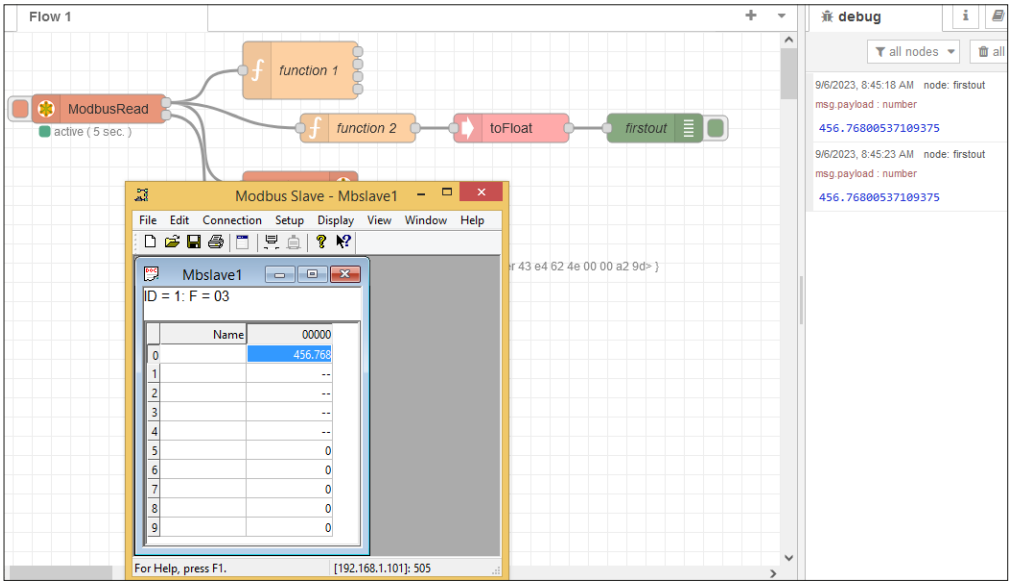


Figure 35.21: Adding a float number in the address 0.

The ModbusWrite node is a versatile tool for writing data to various types of Modbus devices, such as PLCs, RTUs, and other industrial automation devices. It can be combined with other Modbus nodes in Node-RED to create flexible and powerful Modbus-based applications. Our specific task involves writing an integer to address 0 of the holding register with a Modbus ID of 1 in the EasyModbusTCP Server Simulator, as shown in Figure 35.22. To accomplish this, we need to implement the corresponding flows in the Node-RED editor, as depicted in Figure 35.23. The first node, referred to as the inject node, serves as a useful tool for testing and debugging Node-RED flows. It allows manual injection of messages into the flow for observation of the processing. Additionally, the inject node can be used to trigger flows at specific times or intervals, or to simulate real-world events within the flows. The "Properties" window of the "Edit inject node" is displayed in Figure 35.24.

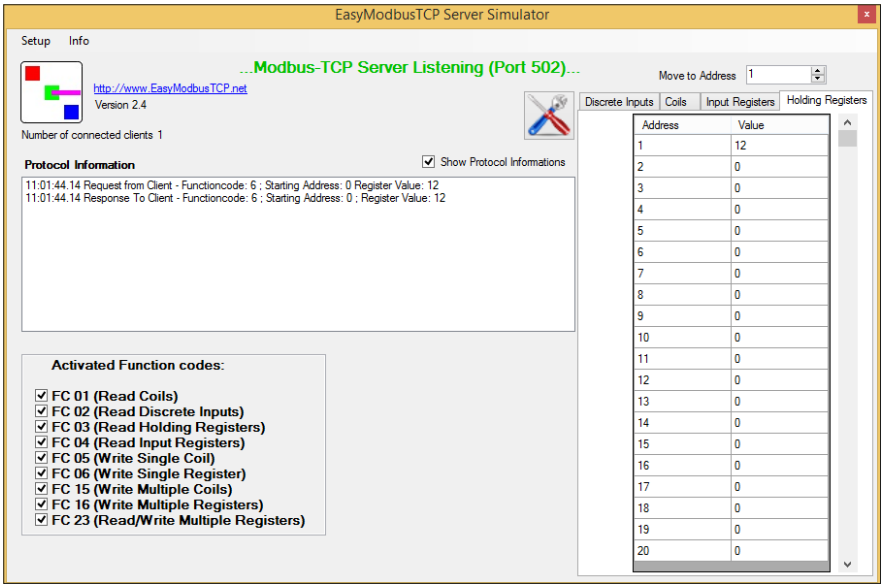


Figure 35.22: Writing an integer in the address 0 of the holding register.

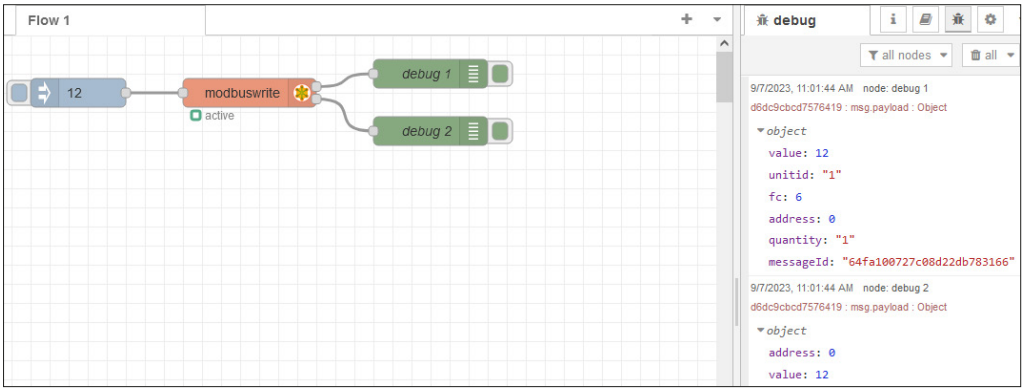


Figure 35.23: Implementing the Modbus Write flows in the Node-RED Editor.

After selecting the msg.payload that matches the value of 12 in Figure 35.23, we proceed by clicking the "Done" button.

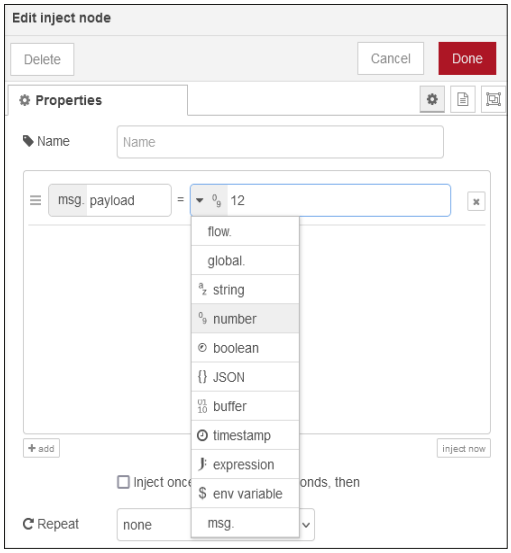


Figure 35.24: Choosing the msg.payload = 12.

To open the "Edit Modbus-Write" node and adjust the settings as shown in Figure 35.25, we simply need to double-click on the ModbusWrite node, which is the following node.

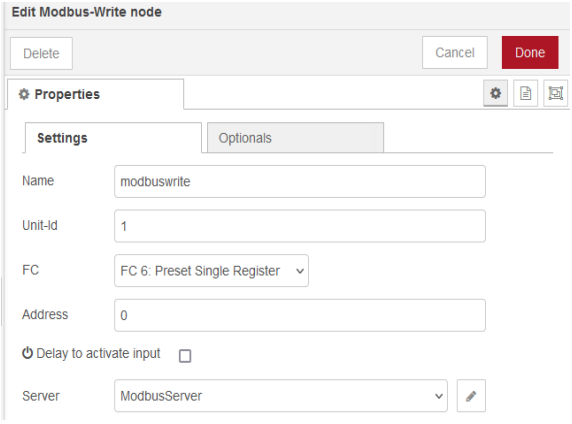


Figure 35.25: Settings of the "Edit Modbus-Write" node.

Next, we will select the pencil icon located next to the "ModbusServer" in Figure 30.4. This action will open the window for editing the modbus-client node, where we can make the necessary settings depicted in Figure 35.26.

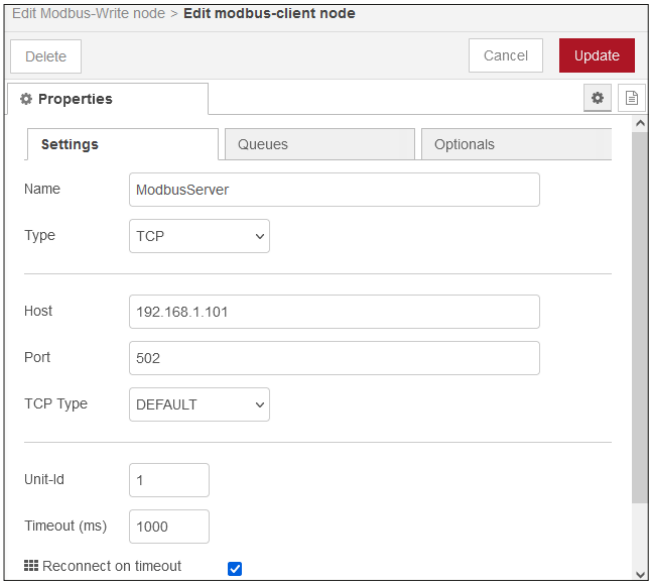


Figure 35.26: Settings of the "Edit modbus-client node" window.

We explore an alternative method of writing in Modbus using the function and Modbus-Flex-Write nodes. The Modbus Flex Write node in Node-RED is a contributed node that enables writing data to Modbus registers. It offers flexibility by allowing single-value writes to a single register as well as multiple-value writes to multiple registers. Do not forget to deploy your flow to apply any changes made. The Modbus Flex Write node will send the specified write request to the Modbus server and update the corresponding registers with the provided data. Figure 35.27 provides an overview of the flows in the Node-RED Editor. The inject node is used solely to trigger the flow, but it can also include the value to be written. You can see the "Edit inject node" window in Figure 35.28.

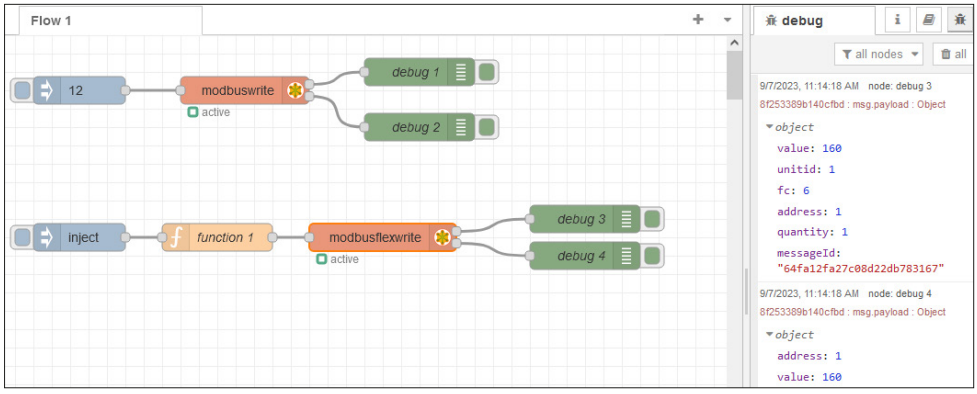


Figure 35.27: The overview of the flows in the Node-RED Editor.

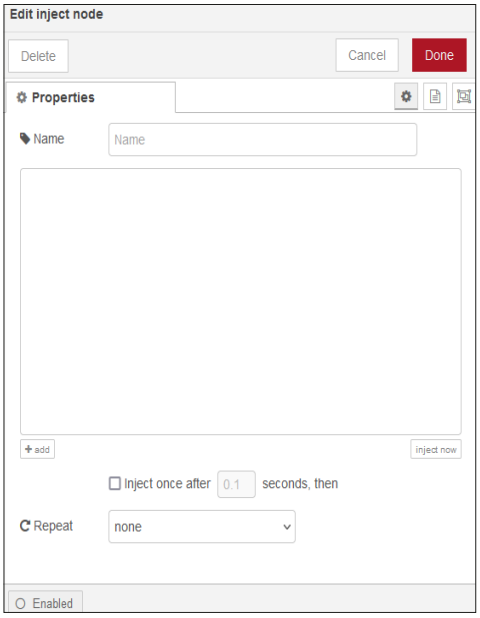


Figure 35.28: The "Edit inject node" window.

Figure 35.29 illustrates the function node and the "Edit function node" window, which contains JavaScript codes.

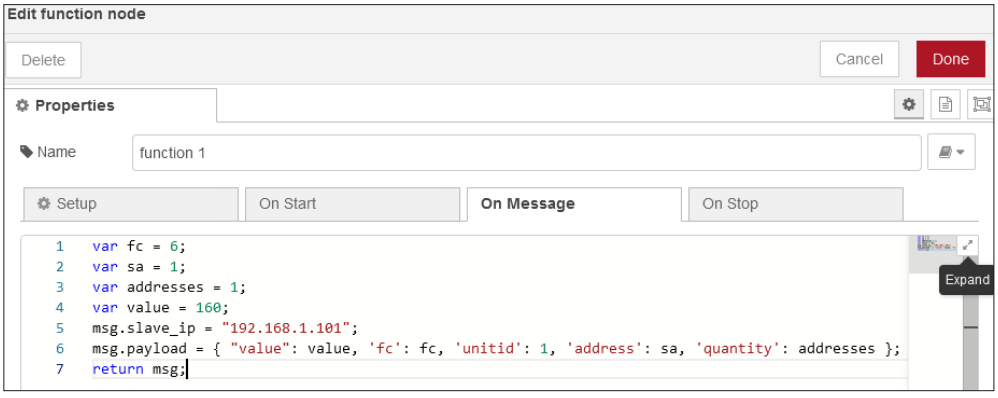


Figure 35.29: JavaScript codes for the function node.

For writing a 32-bit integer we have:

```
var fc=16;
var sa=1;
var addresses=2;
var buf=Buffer.alloc(4);//create buffer
buf.writeInt32BE(68001);
var values=[(buf[0]*256+buf[1]),(buf[2]*256)+buf[3]]
```

```
msg.slave_ip="192.168.1.101";
msg.payload={"value":values , 'fc': fc, 'unitid': 1, 'address': sa , 'quantity':
addresses };
return msg;
```

For writing a 32-bit float we have:

```
var fc=16;
var sa=1;
var addresses=2;
var value=16001.5;
buf=Buffer.alloc(4);
buf.writeFloatBE(value);
//buf.writeFloatBE(16001.5);
var values=[(buf[0]*256+buf[1]),(buf[2]*256)+buf[3]]
msg.slave_ip="192.168.1.101";
msg.payload={"value":values , 'fc': fc, 'unitid': 1, 'address': sa , 'quantity':
addresses };
return msg;
```

For writing a 64-bit float we have:

```
var fc=16;
var sa=1;
var addresses=4;
var value=16001.5;
buf=Buffer.alloc(8);
buf.writeDoubleBE(value);
//buf.writeFloatBE(16001.5);
var values=[(buf[0]*256+buf[1]),(buf[2]*256)+buf[3],(buf[4]*256+buf[5]),(buf[6]*2
56)+buf[7]];
msg.slave_ip="192.168.1.101";
msg.payload={"value":values , 'fc': fc, 'unitid': 1, 'address': sa , 'quantity':
addresses };
return msg;
```

When you double-click on the Modbus-Flex-Write node, the window for editing the node called "Edit Modbus-Flex-Write" will appear, as shown in Figure 35.30.

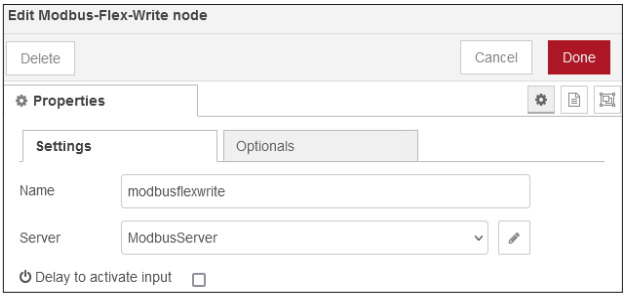


Figure 35.30: "Edit Modbus-Flex-Write" node window settings.

When we click on the pencil icon next to the "ModbusServer" in Figure 281, it opens the Edit modbus-client node window. In this window, we make the settings shown in Figure 35.26. Afterward, we click on the Deploy button to view the integers located at addresses 0 and 1 in the holding registers of the EasyModbusTCP Server Simulator. This is illustrated in Figure 35.31.

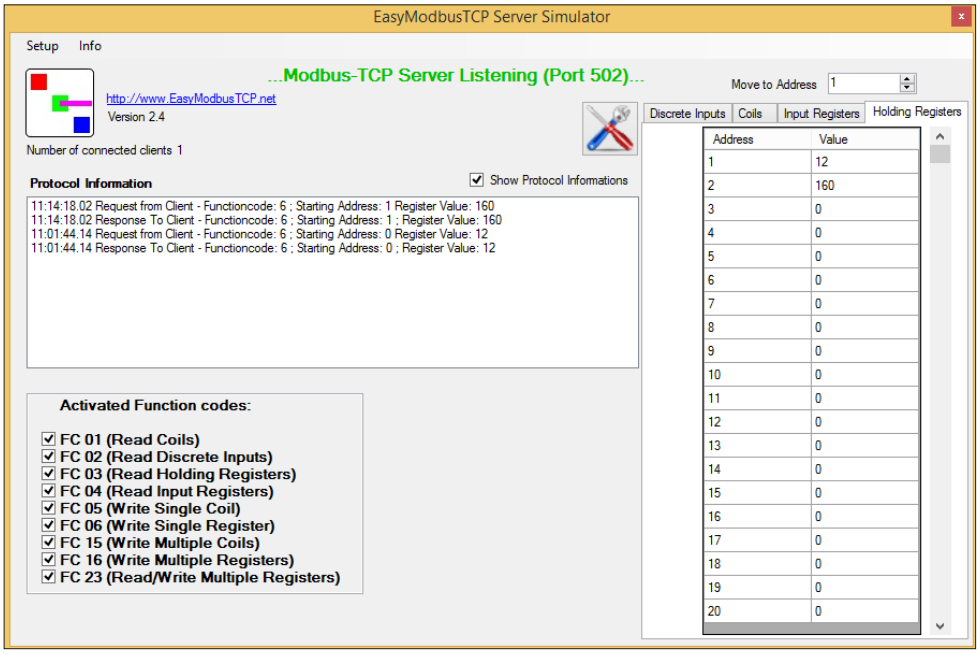


Figure 35.31: Displaying results in the addresses 0 and 1 of the holding registers.

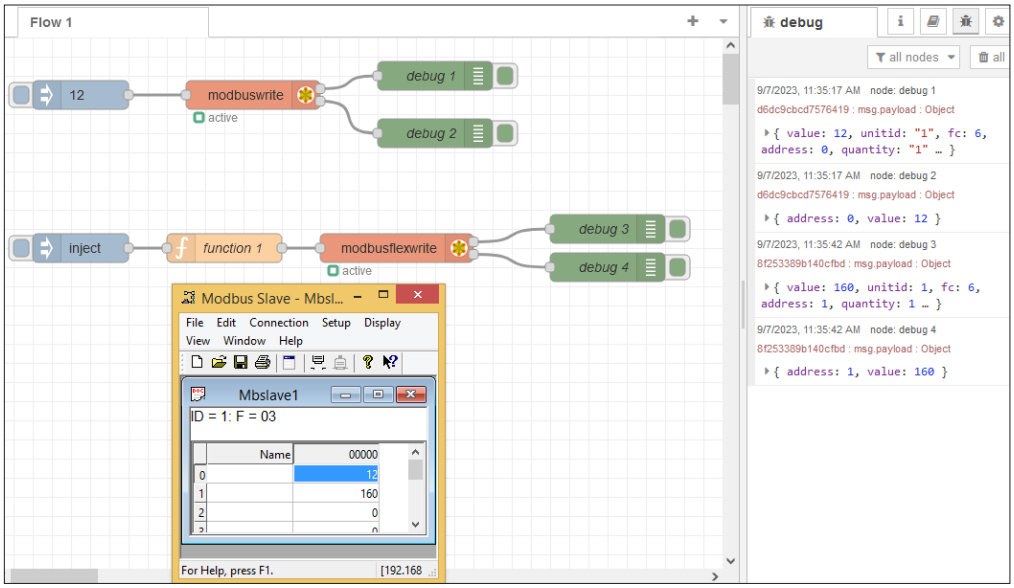


Figure 35.32: Displaying results in the holding registers of Modbus Slave software.

Additionally, it is possible to utilize the previous Modbus Slave software for showcasing the outcomes of Modbus writing in holding registers, as depicted in Figure 35.32.

Chapter 36 • Arduino, Node-RED and Blynk IoT Project

In this section, we will be discussing Blynk IoT and its many advantages. One of the main benefits is the ability to send the dashboard to your cell phone and monitor it from there. Additionally, Blynk allows you to set up alerts, such as email notifications, when a signal is out of range. These features are available in the free version. Another interesting aspect is the ability to import data from Node-RED, which we will demonstrate in this section. To get started, you simply create a Blynk account using your email and then add a device. There are several ways to add a device, including entering a code, scanning a QR code, or using a template like the Quick Start template we used in our case. As shown in Figure 36.1, Figure 36.2, and Figure 36.3 respectively, you can see that the device is already set up.

Figure 36.1: Creating a new device.

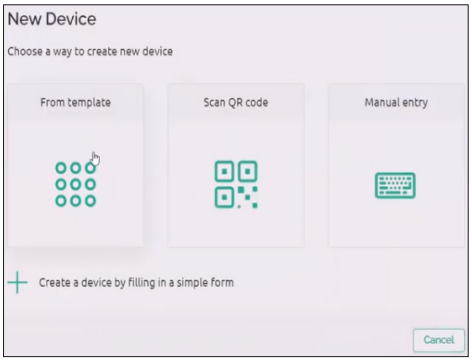


Figure 36.2: Choosing "From template".

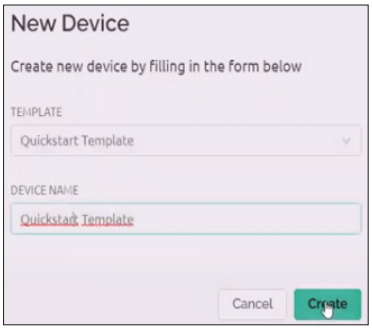


Figure 36.3: Filling in the quick start template.

Currently, there is no connectivity, making it uncertain where Node-RED will source the data from. However, by selecting the Quickstart Template, we can access the necessary information. In this particular scenario, the dashboard has already been established and features a gauge and a graph, thus demonstrating it's functioning as shown in Figure 36.4.

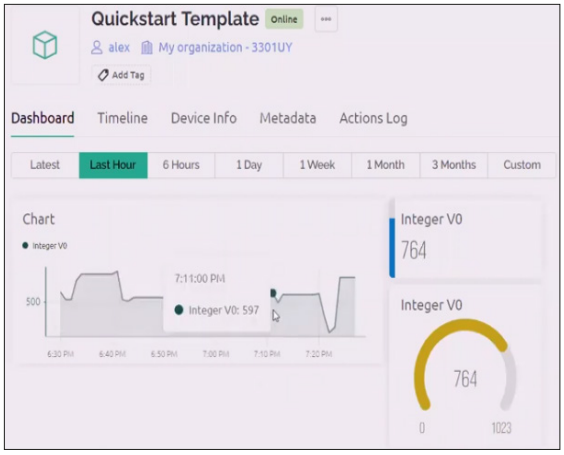


Figure 36.4: The Quickstart Template dashboard.

Using our potentiometer, we successfully adjust the signal range from 0 to 1023. This adjustment was received through email, meaning that setting it to the maximum in the email would result in its arrival here, as depicted in Figure 36.5.

Figure 36.5: Arriving email for Blynk.

We have set up the sensor to receive alerts or alarms when it goes out of range, which is quite interesting. The dashboard displays various topics and widgets that we can check on our device. Additionally, we can also monitor and control automation settings from this device. For example, we configured email notifications to be sent if the value of the virtual pin falls below 100 or exceeds 1000. We received the emails successfully, and if you want to make any changes or add new automation, you can refer to Figure 36.6 for guidance.

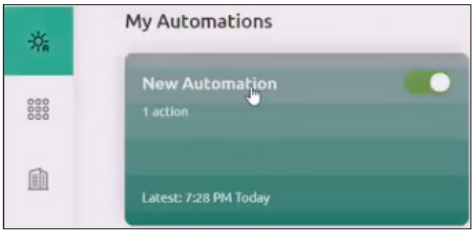


Figure 36.6: "My Automations" settings section.

That is when things become intriguing, as illustrated in Figure 36.7.

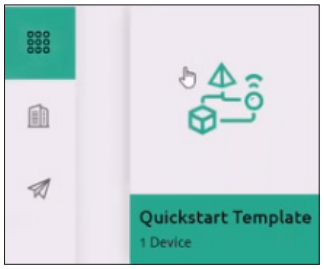


Figure 36.7: The Quickstart Template settings section.

In Figure 36.8, we can see a virtual pin labeled as "v0" that possesses the mentioned attributes as indicated in this line and can be found on the web dashboard mentioned here.

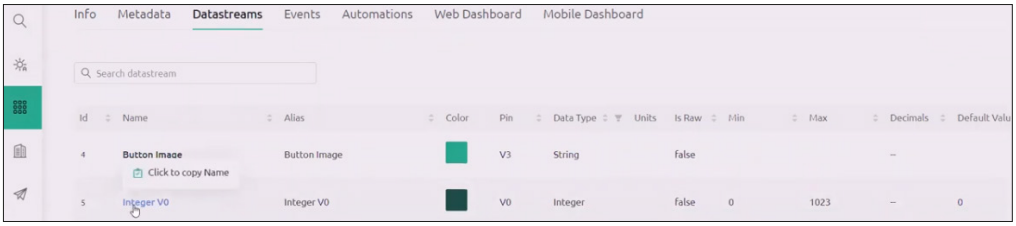


Figure 36.8: Added a virtual pin, v0.

To include additional data, you can access the available options by selecting "Edit" and then selecting "New Datastreams" as demonstrated in Figure 36.9.

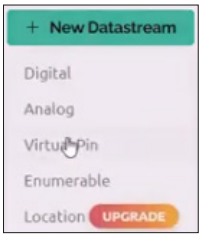


Figure 36.9: Adding a new virtual pin.

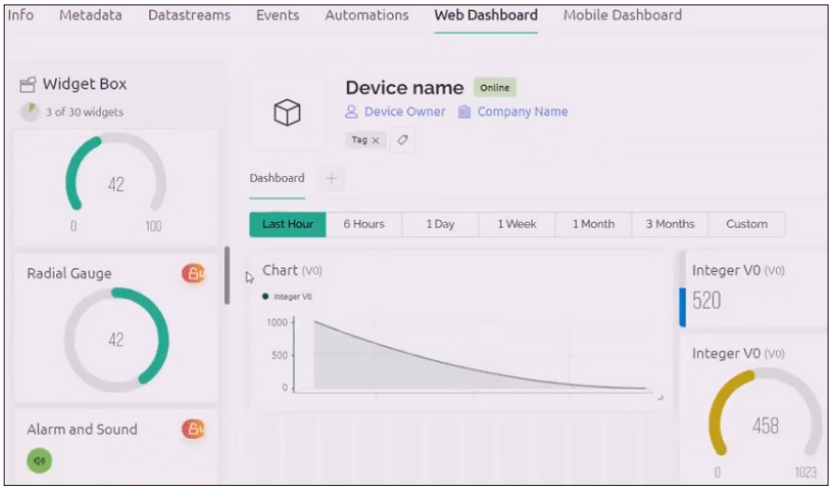


Figure 36.10: Adding widget box elements from web dashboard.

To add widget box elements, simply click on the "Web Dashboard" tab as depicted in Figure 36.10. In the free version, we included a graph, a gauge, and an integer value. It is worth noting that there are additional options available for a fee. However, for a proof of concept, we believe these options are satisfactory. If you wish to view the dashboard on your phone, instructions on how to do so and where to download it from the Android or iOS App Store will be provided. The installation process is straightforward, making it even more helpful, as showed in the mobile dashboard tab. Furthermore, in the "Device Info" tab, displayed in Figure 36.11, we can access valuable device information and create a Quickstart Template.

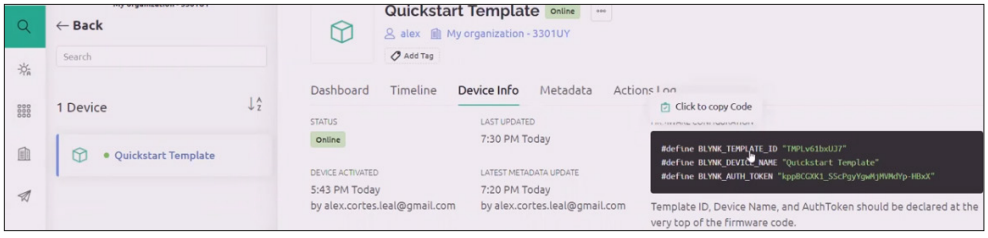


Figure 36.11: The "Device Info" tab.

Figure 36.12 illustrates the Node-RED flow featuring the Quickstart Template and the installed Blynk IoT's write node.

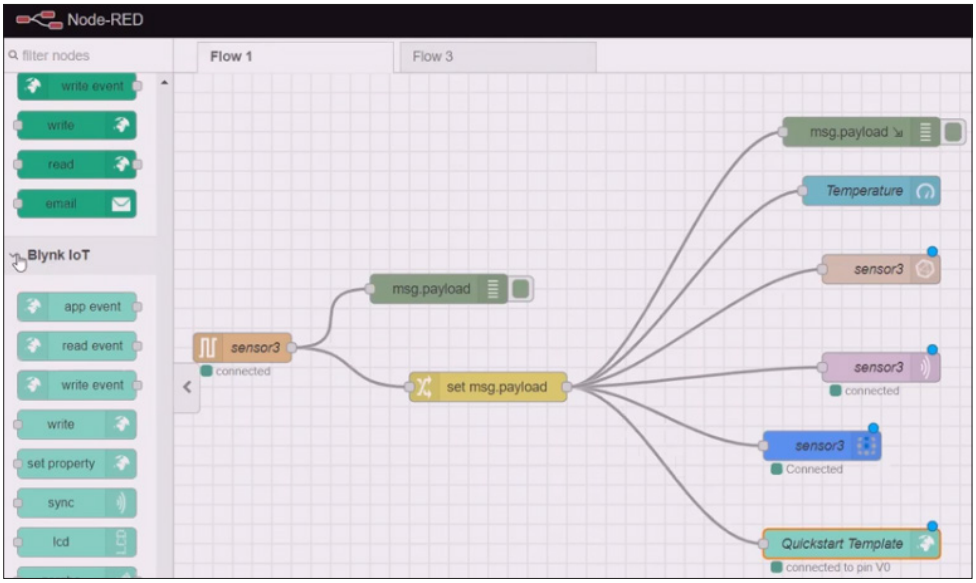


Figure 36.12: The Quickstart Template (Blynk IoT write node) in Node-RED flow.

To add an item that is not currently in the menu, you need to select the "Manage palette" option. Once selected, click "Install" and search for "Blynk IoT". Once you find it, click "Install". This is depicted in Figure 36.13.

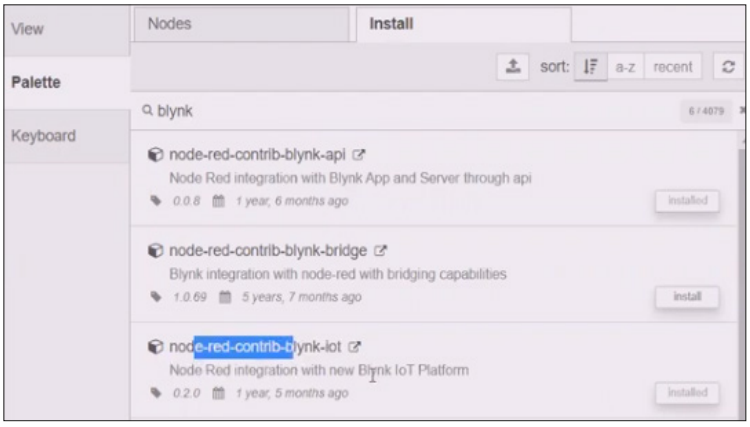


Figure 36.13: Installing the Blynk IoT.

When we open the Quickstart Template in Node-RED flow, it reveals a Blynk IoT write node. By doing so, we can access the "Edit write node" window, as depicted in Figure 36.14.

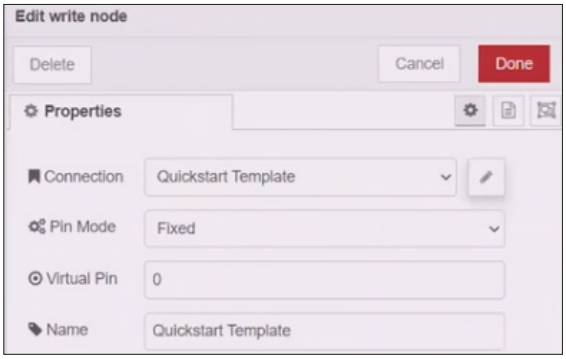


Figure 36.14: "Edit write node" window.

Next, we need to select the pencil icon found beside the "Connection" row to access the "Edit blynk-iot-client node" interface. Here, we will adjust the parameters as showed in Figure 36.15.

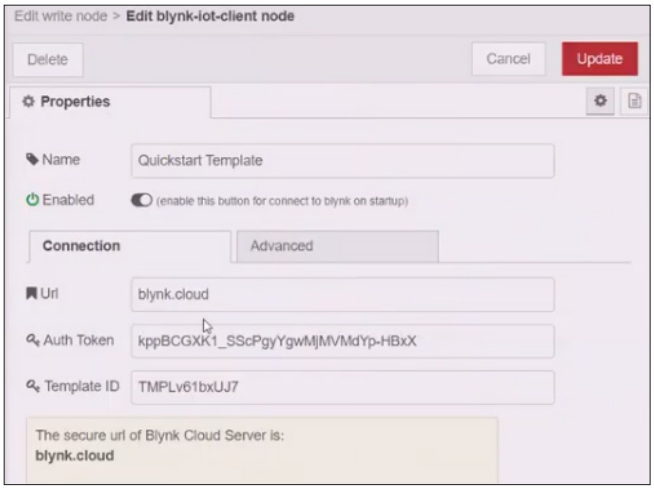


Figure 36.15: Editing the Blynk IoT client node.

Using the "Device Info" tab in Figure 36.11, you can populate the parameters. Upon clicking deploy in the Node-RED flow, we can then see our dashboard. In this particular scenario, let's say we have successfully identified a matching value of 236, as depicted in Figure 36.16.

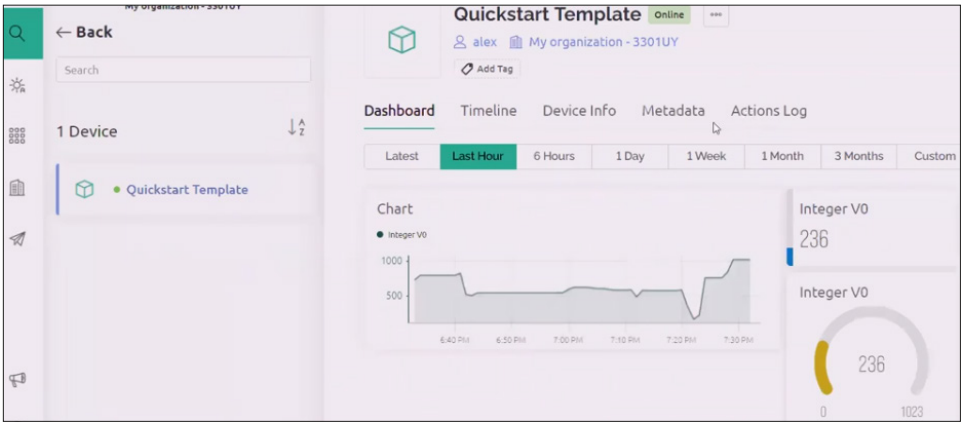


Figure 36.16: Our Blynk dashboard.

Additionally, within the automations, we will be introducing a new one where the objective is to send an email.

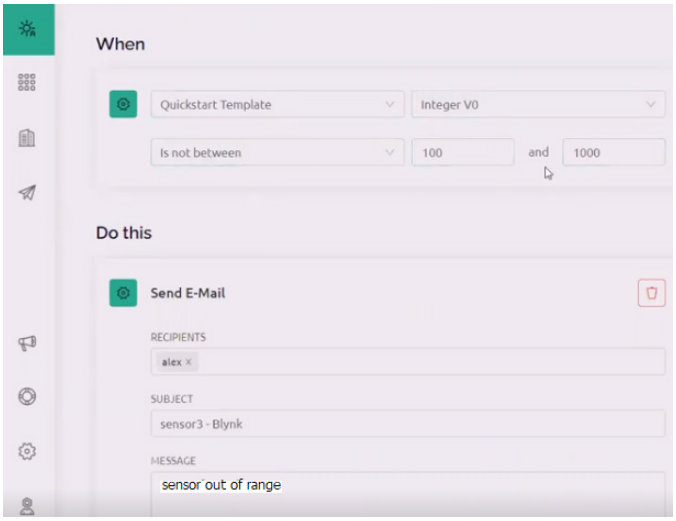


Figure 36.17: Setting the new automation.

Furthermore, we have instructed the platform to execute the action shown in Figure 36.17 when the integer V0 falls outside the range of 100 to 1000. This ensures a visually appealing and user-friendly platform experience. Additionally, users have the convenience of importing data from platforms such as Node-RED.

Chapter 37 • The MQTT DHT22 ESP32 and Node-RED Project

The Wokwi Simulator is an online platform that offers users the opportunity to simulate and experiment with their Arduino and Raspberry Pi projects. It creates a virtual hardware environment that eliminates the requirement for physical components, allowing users to write and execute code. Through the Wokwi Simulator, users can design, prototype, debug code, and simulate circuit components' interactions. It is an invaluable tool for individuals interested in learning and exploring programming and electronics. We conducted a search on the Wokwi website for the MQTT DHT22 ESP32, which yielded the following result.

https://wokwi.com/projects/374901947392501761

We go to the Node-RED dashboard and look for MQTT, and then add two MQTT input/output nodes to the flow, as depicted in Figure 37.1.

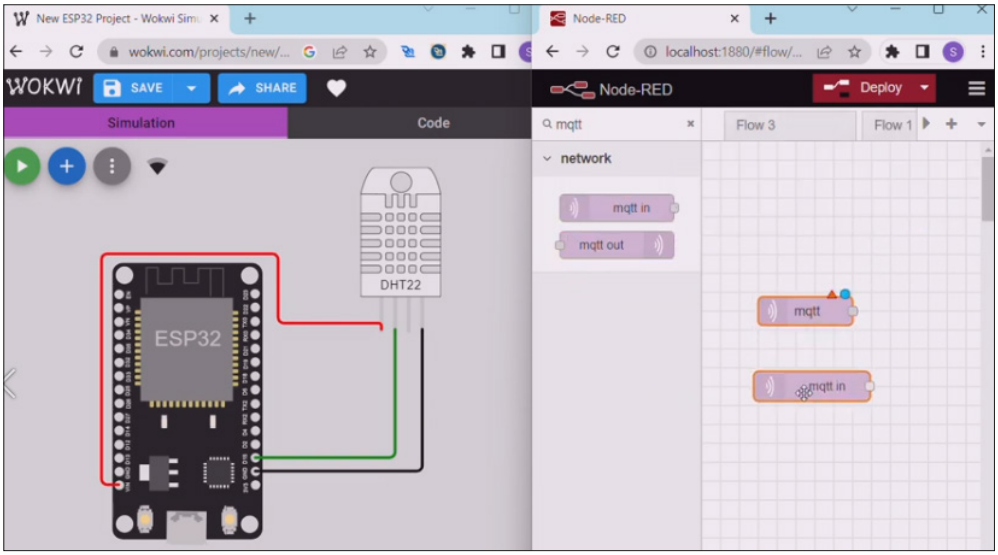


Figure 37.1: Putting two MQTT in nodes in the flow.

Next, we locate the gauge node and incorporate two-gauge nodes into the flow, as illustrated in Figure 37.2.

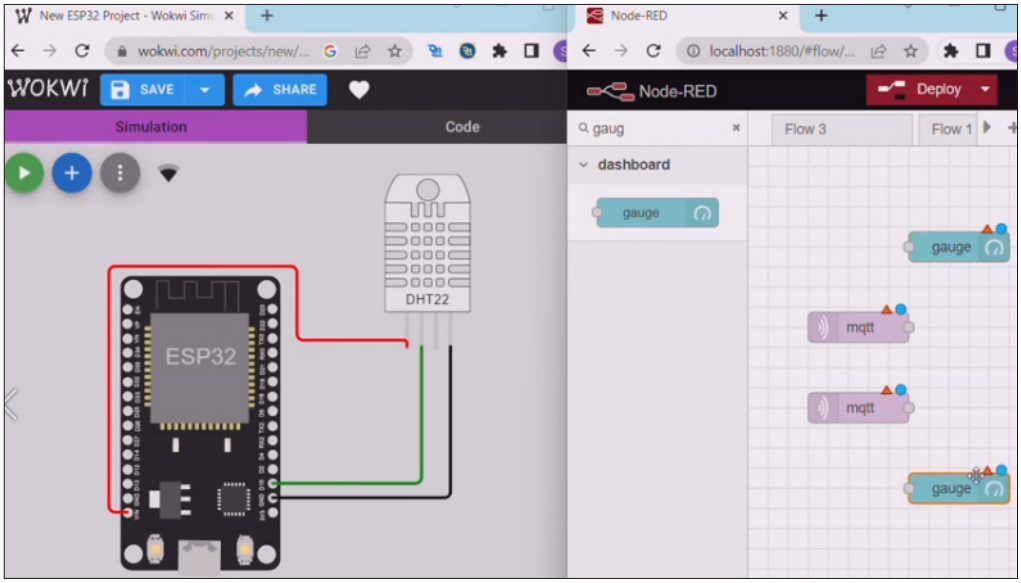


Figure 37.2: Putting two-gauge nodes in the flow.

Next, we proceed to select "mqqt in node" and adjust the parameters in the corresponding "Edit mqqt in node" window. Afterward, we complete the setup by clicking on the "Done" button, as shown in Figure 37.3. Similarly, we repeat these configurations for the other mqqt in node, with the exception that the topic will be "/ThinkIOT/hum", showed in Figure 37.4.

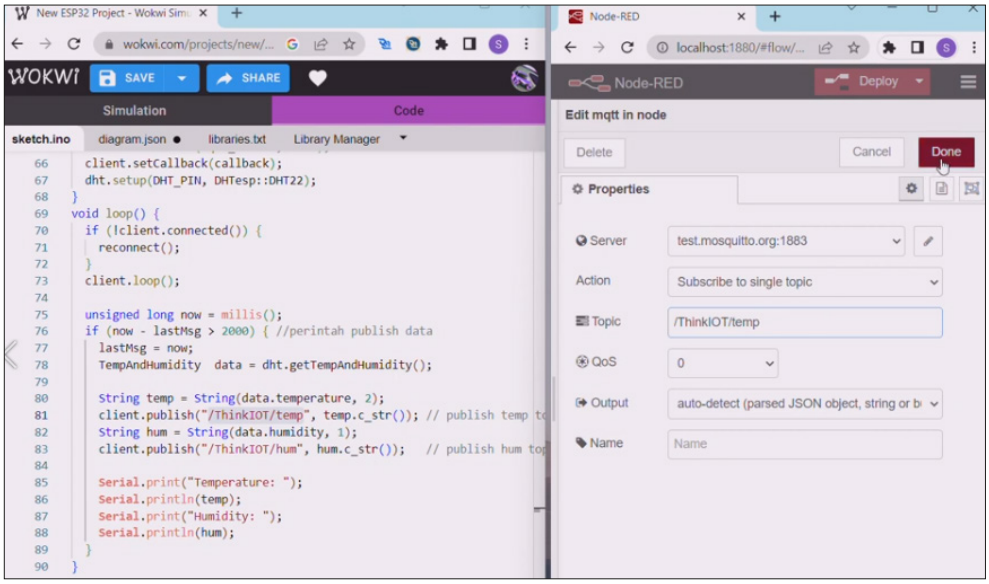


Figure 37.3: Setting the "Edit mqqt in node" parameters.

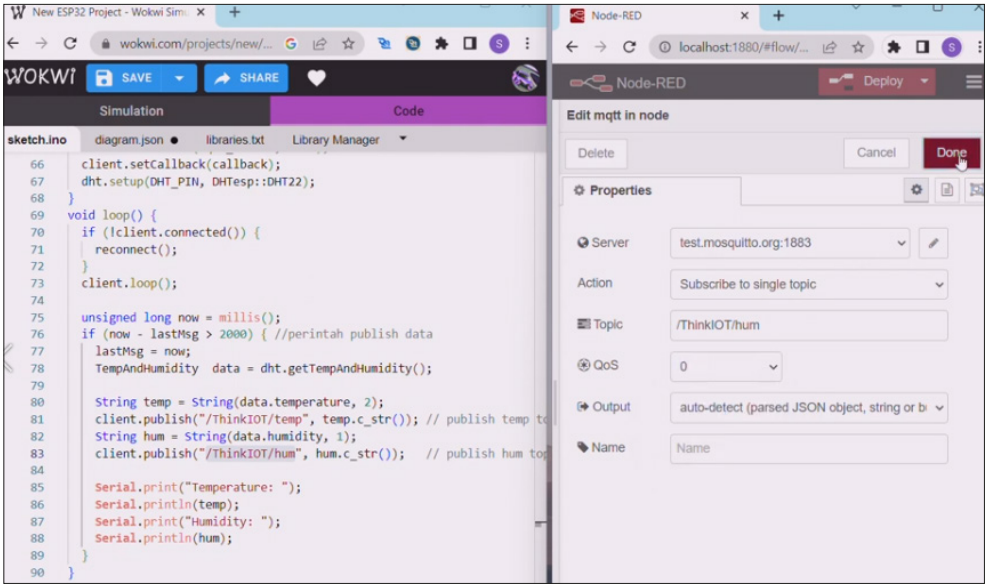


Figure 37.4: Setting the "Edit mqqt in node" parameters.

Next, we select the top gauge, which opens the "Edit gauge node" window. We proceed to fill in the parameters and then click on the "Done" button, as shown in Figure 37.5.

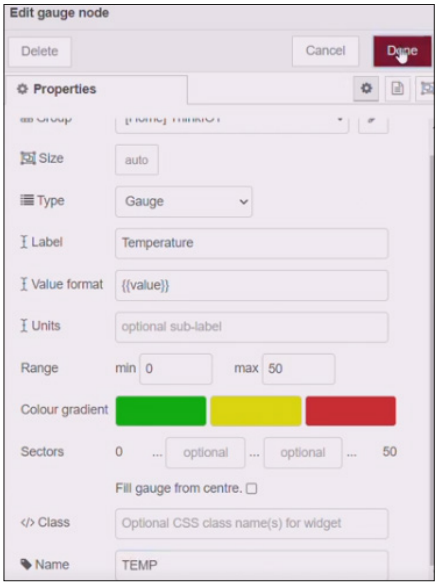


Figure 37.5: Setting the "Edit gauge node" parameters.

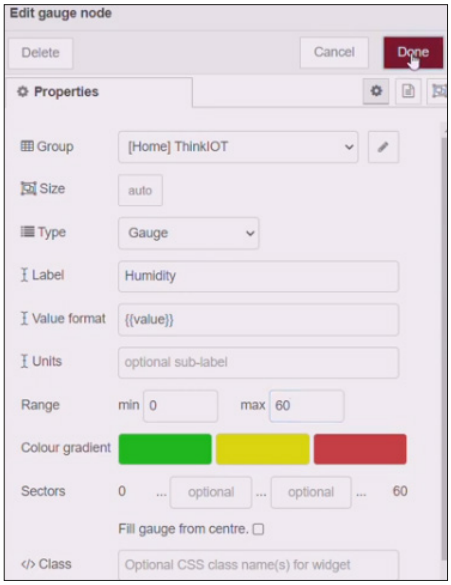


Figure 37.6: Setting the "Edit gauge node" parameters.

Next, we continue to click on the lower gauge, which will open the window for editing the gauge node. We fill in the necessary parameters and then click on the "Done" button, as illustrated in Figure 37.6. We set up connections between the mqqt in nodes and their corresponding gauges, and then click on the "Deploy" button, as displayed in Figure 37.7.

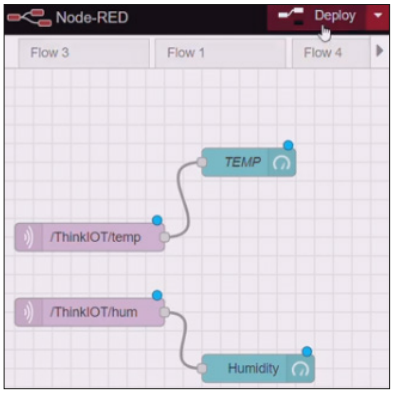


Figure 37.7: Clicking on the "Deploy" button.

Once Node-RED is deployed, we can proceed to run the Wokwi simulator. To find the dashboard, we need to click on the search icon located at the top right corner, as shown in Figure 37.8.

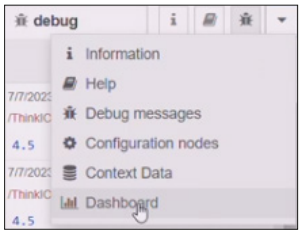


Figure 37.8: Clicking on "Dashboard" icon.

Next, we open the gauge view window by clicking on the icon shown in Figure 37.9 and Figure 37.10 respectively after selecting the "Theme" tab.

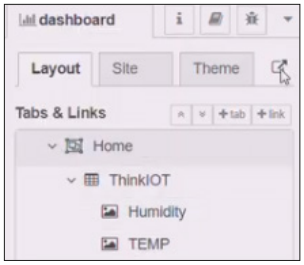


Figure 37.9: Clicking on the icon after the "Theme" tab.

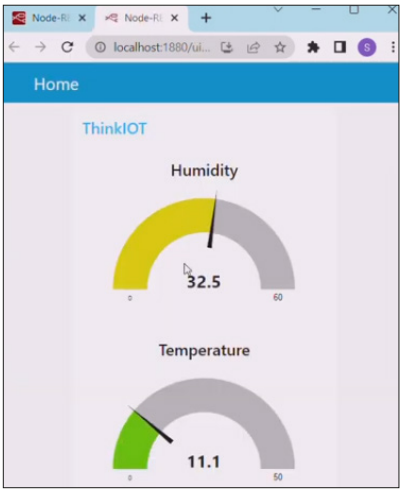


Figure 37.10: The gauges data view.

Hence, by following the approach explained in this section, we can grasp the procedure to showcase data through the MQTT protocol within the Node-RED framework.

Appendix

MgsModbus.cpp:

```
#include "MgsModbus.h"
// For Arduino 1.0
EthernetServer MbServer(MB_PORT);
EthernetClient MbmClient;
//#define DEBUG
MgsModbus::MgsModbus()
{
}
//****************** Send data for ModBusMaster ****************
void MgsModbus::Req(MB_FC FC, word Ref, word Count, word Pos)
{
  MbmFC = FC;
  byte ServerIp[] = {0,0,0,0};
  ServerIp[0] = remServerIP[0];
  ServerIp[1] = remServerIP[1];
  ServerIp[2] = remServerIP[2];
  ServerIp[3] = remServerIP[3];
  MbmByteArray[0] = 0;  // ID high byte
  MbmByteArray[1] = 1;  // ID low byte
  MbmByteArray[2] = 0;  // protocol high byte
  MbmByteArray[3] = 0;  // protocol low byte
  MbmByteArray[5] = 6;  // Lenght low byte;
  MbmByteArray[4] = 0;  // Lenght high byte
  MbmByteArray[6] = 1;  // unit ID
  MbmByteArray[7] = FC; // function code
  MbmByteArray[8] = highByte(Ref);
  MbmByteArray[9] = lowByte(Ref);
  //****************** Read Coils (1) & Read Input discretes (2)
**********************
  if(FC == MB_FC_READ_COILS || FC == MB_FC_READ_DISCRETE_INPUT) {
    if (Count < 1) {Count = 1;}
    if (Count > 125) {Count = 2000;}
    MbmByteArray[10] = highByte(Count);
    MbmByteArray[11] = lowByte(Count);
  }
  //****************** Read Registers (3) & Read Input registers (4)
******************
  if(FC == MB_FC_READ_REGISTERS || FC == MB_FC_READ_INPUT_REGISTER) {
    if (Count < 1) {Count = 1;}
    if (Count > 125) {Count = 125;}
    MbmByteArray[10] = highByte(Count);
    MbmByteArray[11] = lowByte(Count);
```

```
  }
  //****************** Write Coil (5) **********************
  if(MbmFC == MB_FC_WRITE_COIL) {
    if (GetBit(Pos)) {MbmByteArray[10] = 0xFF;} else {MbmByteArray[10] = 0;} //
0xFF coil on 0x00 coil off
    MbmByteArray[11] = 0; // always zero
  }
  //****************** Write Register (6) ******************
  if(MbmFC == MB_FC_WRITE_REGISTER) {
    MbmByteArray[10] = highByte(MbData[Pos]);
    MbmByteArray[11] = lowByte(MbData[Pos]);
  }
  //****************** Write Multiple Coils (15) **********************
  // not fuly tested
  if(MbmFC == MB_FC_WRITE_MULTIPLE_COILS) {
    if (Count < 1) {Count = 1;}
    if (Count > 800) {Count = 800;}
    MbmByteArray[10] = highByte(Count);
    MbmByteArray[11] = lowByte(Count);
    MbmByteArray[12] = (Count + 7) /8;
    MbmByteArray[4] = highByte(MbmByteArray[12] + 7); // Lenght high byte
    MbmByteArray[5] = lowByte(MbmByteArray[12] + 7); // Lenght low byte;
    for (int i=0; i<Count; i++) {
      bitWrite(MbmByteArray[13+(i/8)],i-((i/8)*8),GetBit(Pos+i));
    }
  }
  //****************** Write Multiple Registers (16) ******************
  if(MbmFC == MB_FC_WRITE_MULTIPLE_REGISTERS) {
    if (Count < 1) {Count = 1;}
    if (Count > 100) {Count = 100;}
    MbmByteArray[10] = highByte(Count);
    MbmByteArray[11] = lowByte(Count);
    MbmByteArray[12] = (Count*2);
    MbmByteArray[4] = highByte(MbmByteArray[12] + 7); // Lenght high byte
    MbmByteArray[5] = lowByte(MbmByteArray[12] + 7); // Lenght low byte;
    for (int i=0; i<Count;i++) {
      MbmByteArray[(i*2)+13] = highByte (MbData[Pos + i]);
      MbmByteArray[(i*2)+14] = lowByte (MbData[Pos + i]);
    }
  }
  //****************** ?? ******************
  if (MbmClient.connect(ServerIp,502)) {
    #ifdef DEBUG
      Serial.println("connected with modbus slave");
      Serial.print("Master request: ");
      for(int i=0;i<MbmByteArray[5]+6;i++) {
```

```
        if(MbmByteArray[i] < 16){Serial.print("0");}
        Serial.print(MbmByteArray[i],HEX);
        if (i != MbmByteArray[5]+5) {Serial.print(".");} else {Serial.println();}
      }
    #endif
    for(int i=0;i<MbmByteArray[5]+6;i++) {
      MbmClient.write(MbmByteArray[i]);
    }
    MbmCounter = 0;
    MbmByteArray[7] = 0;
    MbmPos = Pos;
    MbmBitCount = Count;
  } else {
    #ifdef DEBUG
      Serial.println("connection with modbus master failed");
    #endif
    MbmClient.stop();
  }
}
//****************** Recieve data for ModBusMaster ****************
void MgsModbus::MbmRun()
{
  //****************** Read from socket ****************
  while (MbmClient.available()) {
    MbmByteArray[MbmCounter] = MbmClient.read();
    if (MbmCounter > 4)  {
      if (MbmCounter == MbmByteArray[5] + 5) { // the full answer is recieved
        MbmClient.stop();
        MbmProcess();
        #ifdef DEBUG
          Serial.println("recieve klaar");
        #endif
      }
    }
    MbmCounter++;
  }
}
void MgsModbus::MbmProcess()
{
  MbmFC = SetFC(int (MbmByteArray[7]));
  #ifdef DEBUG
    for (int i=0;i<MbmByteArray[5]+6;i++) {
      if(MbmByteArray[i] < 16) {Serial.print("0");}
      Serial.print(MbmByteArray[i],HEX);
      if (i != MbmByteArray[5]+5) {Serial.print(".");
      } else {Serial.println();}
```

```
    }
  #endif
  //****************** Read Coils (1) & Read Input discretes (2)
**********************
  if(MbmFC == MB_FC_READ_COILS || MbmFC == MB_FC_READ_DISCRETE_INPUT) {
    word Count = MbmByteArray[8] * 8;
    if (MbmBitCount < Count) {
      Count = MbmBitCount;
    }
    for (int i=0;i<Count;i++) {
      if (i + MbmPos < MbDataLen * 16) {
        SetBit(i + MbmPos,bitRead(MbmByteArray[(i/8)+9],i-((i/8)*8)));
      }
    }
  }
  //****************** Read Registers (3) & Read Input registers (4)
******************
  if(MbmFC == MB_FC_READ_REGISTERS || MbmFC == MB_FC_READ_INPUT_REGISTER) {
    word Pos = MbmPos;
    for (int i=0;i<MbmByteArray[8];i=i+2) {
      if (Pos < MbDataLen) {
        MbData[Pos] = (MbmByteArray[i+9] * 0x100) + MbmByteArray[i+1+9];
        Pos++;
      }
    }
  }
  //****************** Write Coil (5) **********************
  if(MbmFC == MB_FC_WRITE_COIL){
  }
  //****************** Write Register (6) ******************
  if(MbmFC == MB_FC_WRITE_REGISTER){
  }
  //****************** Write Multiple Coils (15) ***********************
  if(MbmFC == MB_FC_WRITE_MULTIPLE_COILS){
  }
  //****************** Write Multiple Registers (16) ******************
  if(MbmFC == MB_FC_WRITE_MULTIPLE_REGISTERS){
  }
}
//****************** Recieve data for ModBusSlave ****************
void MgsModbus::MbsRun()
{
  //****************** Read from socket ****************
  EthernetClient client = MbServer.available();
  if(client.available())
  {
```

```
    delay(10);
    int i = 0;
    while(client.available())
    {
      MbsByteArray[i] = client.read();
      i++;
    }
    MbsFC = SetFC(MbsByteArray[7]);  //Byte 7 of request is FC
  }
  int Start, WordDataLength, ByteDataLength, CoilDataLength, MessageLength;
  //****************** Read Coils (1 & 2) *********************
  if(MbsFC == MB_FC_READ_COILS || MbsFC == MB_FC_READ_DISCRETE_INPUT) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    CoilDataLength = word(MbsByteArray[10],MbsByteArray[11]);
    ByteDataLength = CoilDataLength / 8;
    if(ByteDataLength * 8 < CoilDataLength) ByteDataLength++;
    CoilDataLength = ByteDataLength * 8;
    MbsByteArray[5] = ByteDataLength + 3; //Number of bytes after this one.
    MbsByteArray[8] = ByteDataLength;     //Number of bytes after this one (or
number of bytes of data).
    for(int i = 0; i < ByteDataLength ; i++)
    {
      MbsByteArray[9 + i] = 0; // To get all remaining not written bits zero
      for(int j = 0; j < 8; j++)
      {
        bitWrite(MbsByteArray[9 + i], j, GetBit(Start + i * 8 + j));
      }
    }
    MessageLength = ByteDataLength + 9;
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
  //****************** Read Registers (3 & 4) ******************
  if(MbsFC == MB_FC_READ_REGISTERS || MbsFC == MB_FC_READ_INPUT_REGISTER) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    WordDataLength = word(MbsByteArray[10],MbsByteArray[11]);
    ByteDataLength = WordDataLength * 2;
    MbsByteArray[5] = ByteDataLength + 3; //Number of bytes after this one.
    MbsByteArray[8] = ByteDataLength;     //Number of bytes after this one (or
number of bytes of data).
    for(int i = 0; i < WordDataLength; i++)
    {
      MbsByteArray[ 9 + i * 2] = highByte(MbData[Start + i]);
      MbsByteArray[10 + i * 2] =  lowByte(MbData[Start + i]);
    }
    MessageLength = ByteDataLength + 9;
```

```
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
  //****************** Write Coil (5) **********************
  if(MbsFC == MB_FC_WRITE_COIL) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    if (word(MbsByteArray[10],MbsByteArray[11]) == 0xFF00){SetBit(Start,true);}
    if (word(MbsByteArray[10],MbsByteArray[11]) == 0x0000){SetBit(Start,false);}
    MbsByteArray[5] = 2; //Number of bytes after this one.
    MessageLength = 8;
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
  //****************** Write Register (6) ******************
  if(MbsFC == MB_FC_WRITE_REGISTER) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    MbData[Start] = word(MbsByteArray[10],MbsByteArray[11]);
    MbsByteArray[5] = 6; //Number of bytes after this one.
    MessageLength = 12;
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
  //****************** Write Multiple Coils (15) **********************
  if(MbsFC == MB_FC_WRITE_MULTIPLE_COILS) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    CoilDataLength = word(MbsByteArray[10],MbsByteArray[11]);
    MbsByteArray[5] = 6;
    for(int i = 0; i < CoilDataLength; i++)
    {
      SetBit(Start + i,bitRead(MbsByteArray[13 + (i/8)],i-((i/8)*8)));
    }
    MessageLength = 12;
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
  //****************** Write Multiple Registers (16) ******************
  if(MbsFC == MB_FC_WRITE_MULTIPLE_REGISTERS) {
    Start = word(MbsByteArray[8],MbsByteArray[9]);
    WordDataLength = word(MbsByteArray[10],MbsByteArray[11]);
    ByteDataLength = WordDataLength * 2;
    MbsByteArray[5] = 6;
    for(int i = 0; i < WordDataLength; i++)
    {
      MbData[Start + i] =  word(MbsByteArray[ 13 + i * 2],MbsByteArray[14 + i *
2]);
    }
```

```
    MessageLength = 12;
    client.write(MbsByteArray, MessageLength);
    MbsFC = MB_FC_NONE;
  }
}
//****************** ?? ******************
MB_FC MgsModbus::SetFC(int fc)
{
  MB_FC FC;
  FC = MB_FC_NONE;
  if(fc == 1) FC = MB_FC_READ_COILS;
  if(fc == 2) FC = MB_FC_READ_DISCRETE_INPUT;
  if(fc == 3) FC = MB_FC_READ_REGISTERS;
  if(fc == 4) FC = MB_FC_READ_INPUT_REGISTER;
  if(fc == 5) FC = MB_FC_WRITE_COIL;
  if(fc == 6) FC = MB_FC_WRITE_REGISTER;
  if(fc == 15) FC = MB_FC_WRITE_MULTIPLE_COILS;
  if(fc == 16) FC = MB_FC_WRITE_MULTIPLE_REGISTERS;
  return FC;
}
word MgsModbus::GetDataLen()
{
  return MbDataLen;
}
boolean MgsModbus::GetBit(word Number)
{
  int ArrayPos = Number / 16;
  int BitPos = Number - ArrayPos * 16;
  boolean Tmp = bitRead(MbData[ArrayPos],BitPos);
  return Tmp;
}
boolean MgsModbus::SetBit(word Number,boolean Data)
{
  int ArrayPos = Number / 16;
  int BitPos = Number - ArrayPos * 16;
  boolean Overrun = ArrayPos > MbDataLen * 16; // check for data overrun
  if (!Overrun){
    bitWrite(MbData[ArrayPos],BitPos,Data);
  }
  return Overrun;
}
```

MgsModbus.h:

```
/*
  MgsModbus.h - an Arduino library for a Modbus TCP master and slave.
  V-0.1.1 Copyright (C) 2013  Marco Gerritse
  written and tested with Arduino 1.0
    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
  For this library the following library is used as start point:
    [1] Mudbus.h - an Arduino library for a Modbus TCP slave.
        Copyright (C) 2011  Dee Wykoff
    [2] Function codes 15 & 16 by Martin Pettersson
  The following references are used to write this library:
    [3] Open Modbus/Tcp specification, Release 1.0, 29 March 1999
        By Andy Swales, Schneider Electric
    [4] Modbus application protocol specification V1.1b3, 26 april 202
        From http:/www.modbus.org
  External software used for testing:
    [5] modpoll - www.modbusdriver.com/modpoll.html
    [6] ananas - www.tuomio.fi/ananas
    [7] mod_rssim - www.plcsimulator.org
    [8] modbus master - www.cableone.net/mblansett/
  This library use a single block of memory for all modbus data (mbData[] array).
The same data can be reached via several modbus functions, either via a 16 bit
access
  or via an access bit. The length of MbData must at least 1.
  For the master the following modbus functions are implemented: 1, 2, 3, 4, 5,
6, 15, 16
  For the slave the following modbus functions are implemented: 1, 2, 3, 4, 5, 6,
15, 16
  The internal and external addresses are 0 (zero) based
  V-0.1.1 2013-06-02
  bugfix
  V-0.1.0 2013-03-02
  initinal version
*/
#include "Arduino.h"
#include <SPI.h>
```

```
#include <Ethernet.h>
#ifndef MgsModbus_h
#define MgsModbus_h
#define MbDataLen 30 // length of the MdData array
#define MB_PORT 502

enum MB_FC {
  MB_FC_NONE                    = 0,
  MB_FC_READ_COILS              = 1,
  MB_FC_READ_DISCRETE_INPUT     = 2,
  MB_FC_READ_REGISTERS          = 3,
  MB_FC_READ_INPUT_REGISTER     = 4,
  MB_FC_WRITE_COIL              = 5,
  MB_FC_WRITE_REGISTER          = 6,
  MB_FC_WRITE_MULTIPLE_COILS    = 15,
  MB_FC_WRITE_MULTIPLE_REGISTERS = 16
};
class MgsModbus
{
public:
  // general
  MgsModbus();
  word MbData[MbDataLen]; // memory block that holds all the modbus user data
  boolean GetBit(word Number);
  boolean SetBit(word Number,boolean Data); // returns true when the number is in
the MbData range
  // modbus master
  void Req(MB_FC FC, word Ref, word Count, word Pos);
  void MbmRun();
  IPAddress remSlaveIP;
  byte remServerIP[4];
  // modbus slave
  void MbsRun();
  word GetDataLen();
private:
  // general
  MB_FC SetFC(int fc);
  // modbus master
  uint8_t MbmByteArray[260]; // send and recieve buffer
  MB_FC MbmFC;
  int MbmCounter;
  void MbmProcess();
  word MbmPos;
  word MbmBitCount;
  //modbus slave
  uint8_t MbsByteArray[260]; // send and recieve buffer
```

```
  MB_FC MbsFC;
};
#endif
```

References

How to Program an Arduino as a Modbus TCP/IP Client & Server from Emile Ackbarali

https://www.udemy.com/course/how-to-program-an-arduino-as-a-modbus-tcp-client-server/

The modbus-arduino library

https://github.com/andresarmento/modbus-arduino

The Modbus Library

https://myarduinoprojects.com/modbus.html

Arduino & Modbus Protocol

https://docs.arduino.cc/learn/communication/modbus

Arduino-Modbus Tutorials

https://arduinogetstarted.com/tutorials/arduino-modbus

Writing Modbus Data with Node-Red

https://stevesnoderedguide.com/modbus-writing-data

Modbus RTU TCP/IP on Node-RED (16bit, 32bit and floating data)

https://youtu.be/uillyhwBoEU?si=xVtQ8AgTA1YlxYOo

Node-RED - Modbus TCP - Comment réaliser une application Client/Serveur - Partie 1/2

https://youtu.be/WcOsJNsHbfw?si=W0_I-0ga3nl8pAZY

Node RED - Modbus TCP - Comment réaliser une application Client/Serveur - Partie 2/2 - Home I/O

https://youtu.be/Bmpm52wCoQI?si=dsQZ8LF39m4mCJMe

Communication between two Arduino's

https://www.circuits-diy.com/communication-between-two-arduino/

Index

elektor books

Coding Modbus TCP/IP for Arduino

Example projects with Node-RED, MQTT, WinCC SCADA, Blynk, and ThingSpeak

This comprehensive guide unlocks the power of Modbus TCP/IP communication with Arduino. From the basics of the Modbus protocol right up to full implementation in Arduino projects, the book walks you through the complete process with lucid explanations and practical examples.

Learn how to set up Modbus TCP/IP communication with Arduino for seamless data exchange between devices over a network. Explore different Modbus functions and master reading and writing registers to control your devices remotely. Create Modbus client and server applications to integrate into your Arduino projects, boosting their connectivity and automation level.

With detailed code snippets and illustrations, this guide is perfect for beginners and experienced Arduino enthusiasts alike. Whether you're a hobbyist looking to expand your skills or a professional seeking to implement Modbus TCP/IP communication in your projects, this book provides all the knowledge you need to harness the full potential of Modbus with Arduino.

Projects covered in the book:

- TCP/IP communication between two Arduino Uno boards
- Modbus TCP/IP communication within the Node-RED environment
- Combining Arduino, Node-RED, and Blynk IoT cloud
- Interfacing Modbus TCP/IP with WinCC SCADA to control sensors
- Using MQTT protocol with Ethernet/ESP8266
- Connecting to ThingSpeak IoT cloud using Ethernet/ESP8266

Majid Pakdel, born in 1981 in Mianeh, Iran, has a PhD in Electrical Engineering and an MS in Computer Engineering focusing on AI and Robotics. He has published 20+ papers and 4 books. He was a visiting PhD student at Aalborg University in Energy Technology from 2015-2016.

Elektor International Media
www.elektor.com

ISBN 978-3-89576-614-5
9 783895 766145

elektor
design > share > earn