Contents

List of Figures

List of Tables

List of Arduino Code

List of OpenModelica Code

List of Acronyms

ACM	Abstract Control Model
ADC	Analog to Digital Converter
ADK	Accessory Development Kit
ALU	Arithmetic and Logic Unit
ARM	Advanced RISC Machines
BIOS	Basic Input/ Output System
CD	Compact Disc
CNES	National Centre for Space Studies
COM Port	Communication Port
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DC	Direct Current
DIY	Do It Yourself
DVD	Digital Versatile Disc
EEPROM	Electronically Erasable Programmable Read-Only Memory
FPGA	Field-programmable Gate Array
GNU	GNU's Not Unix
GPS	Global Positioning System
GPL	General Public License
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
ICSP	In-Circuit Serial Programming
IDE	Integrated Development Environment
LAPACK	Linear Algebra Package
LCD	Liquid Crystal Display
LDR	Light Dependent Resistor
LED	Light Emitting Diode

MRI	Magnetic Resonance Imaging
MISO	Master Input, Slave output
MOSI	Master out, Slave input
NTC	Negative Temperature Coefficient
OGP	Open Graphics Project
OS	Operating System
OSHW	Open Source Hardware
PCB	Printed Circuit Board
PTC	Positive Temperature Coefficient
PWM	Pulse width modulation
RAM	Random-access Memory
ROM	Read Only Memory
RS	Recommended Standard
RTC	Real Time Clock
Rx	Receiver
SD Card	Secure Digital Card
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TCL	Tool Command Language
Tx	Transmitter
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

Microcontrollers are the foundation for a modern, manufacturing-based economy. One cannot fulfill the dreams of one's citizens without a thriving manufacturing sector. As it is open-source, Arduino Uno is of particular interest to hobbyists, students, small and medium scale manufacturers, and people from developing countries, in particular.

Scilab is a state-of-the-art computing software. It is also open-source. As a result, this is also extremely useful to the groups mentioned above. If the French National Space Agency CNES can extensively use Scilab [1], why can't others rely on it? If many of India's satellites can be placed in their precise orbits by the Ariane rockets launched by CNES through Scilab calculations, why can't others use Scilab?

The above argument can be extended to other open-source software systems Python, Julia, and OpenModelica. Python is a versatile programming language with a high degree of expressiveness, which allows code written in it to be small. Python is also the preferred language in emerging areas, such as machine learning. Julia is also highly expressive, just as Scilab and Python. The only difference is that Julia is generally faster to execute, compared to Scilab and Python.

OpenModelica implements the Modelica language. It is especially created for modeling. It is an object-oriented programming language, especially when it comes to modeling. It has a GUI that lends itself to connecting different building blocks to create models. Models in OpenModelica are solved by collecting equations from different building blocks and solving them simultaneously.

Xcos is a GUI based system building tool for Scilab, somewhat similar to Simulink®[1]. Through Xcos, it is possible to build interconnected systems graphically. Xcos also is an open-source software tool.

Although Scilab, Xcos, Python, Julia, and OpenModelica are powerful and free,

[1] Simulink® is a registered trademark of Mathworks, Inc.

there has not been much literature that teaches how to use them to program the versatile Arduino Uno. To address this gap, we have written this series of books. We have provided code written in all of these open-source software. The reader is recommended to go through the book that covers a particular software.

The only way we can become versatile in hardware is through hands-on training. To this end, we make use of the easily available low-cost Arduino Uno board to introduce the reader to computer interfacing. We also make available the details of a shield that makes the Arduino Uno use extremely easy and intuitive. We tell the user how to install the firmware to make the Arduino Uno board communicate with the computer. We explain how to control the peripherals on the Arduino Uno board with user-developed software.

The Scilab-Arduino toolbox is already available for Windows [2]. We have suitably modified it so that it works on Linux also. We give the required programs to experiment with the sensors and actuators that come with the shield, a DC motor, and a servomotor. These programs are available for all of the following environments: Arduino IDE, Scilab, Xcos, Python, Julia, and OpenModelica. In addition to these toolboxes, we provide the firmware for each software and a program to check its working.

This book teaches how to access the following sensors and actuators: LED, push-button, LDR, Potentiometer, Thermistor, Servo motor, and DC motor. A set of two to five programs are given for each. These are given for all the software mentioned above. The reader has to see the book devoted to the appropriate software. We also explain where to find these programs and how to execute them for each experiment.

This book is written for self-learners and hobbyists. It has been field-tested by 250 people who attended a hands-on workshop conducted at IIT Bombay in July 2015. It has also been field-tested by 25 people who participated in a TEQIP course held in Amravati in November 2015.

All the code described in this book is available at `https://floss-arduino.fossee.in/`. On downloading and unzipping it, it will open a folder `Origin` in the current directory. All the files mentioned in this book are with reference to this folder[2].

[2]This naming convention will be used throughout this book. Users are expected to download this file and use it while reading this book.

Chapter 2

Hardware Environment

In this book, we shall use an Arduino Uno board and associated circuitry to perform several experiments on data acquisition and control. This chapter will briefly take you through the hardware environment needed to perform these experiments. We will start with the introduction to a microcontroller followed by a brief on Open Source Hardware. Then, we shall go through the history and hardware specifications of the Arduino Uno board and the schema and uses of the shield provided in the kit.

2.1 Microcontroller

A microcontroller is a "smart" and complex programmable digital circuit that contains a processor, memory and input/output peripherals on a single integrated circuit. Effectively, it can function as a small computer that can perform a variety of applications. A few of these day-to-day applications include:

- Automotive: Braking, driver assist, fault diagnosis, power steering
- Household appliances: CD/DVD players, washing machines, microwave ovens, energy meters
- Telecommunication: Mobile phones, switches, routers, ethernet controllers
- Medical: Implantable devices, MRI, ultrasound, dental imaging
- General: Automation, safety systems, electronic measurement instruments

2.1.1 Organization of a Microcontroller

In this section, we will give a brief overview of the organization of a typical microcontroller. A microcontroller consists of three major components, namely, Processor,

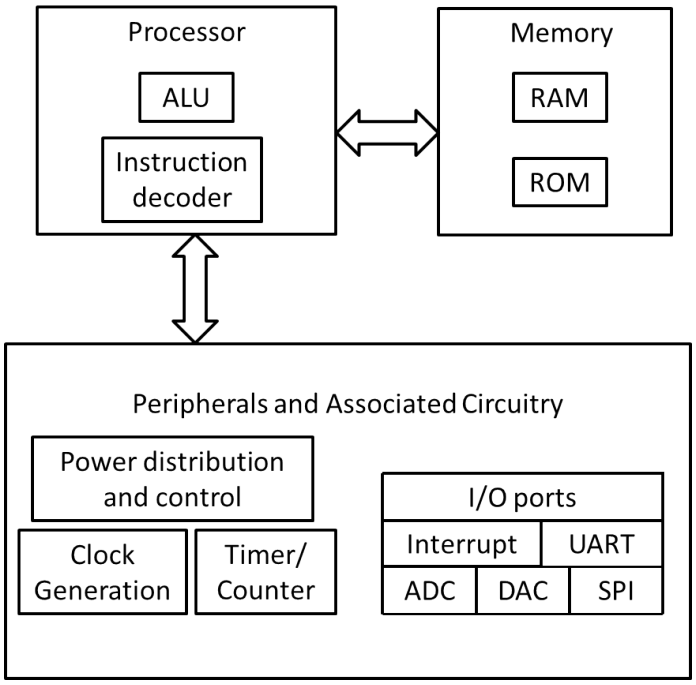


Figure 2.1: Functional block diagram of a microcontroller

Memory and Peripherals. The basic block diagram of a microcontroller is shown in Fig. 2.1. We shall briefly review the functionality of each block.

Processor: It is also known as a Central Processing Unit (CPU). A processor is the heart of any computer/embedded system. The applications running on these systems involve arithmetic and logic operations. These operations are further simplified into instructions and fed to the processor. The Instruction decoder decodes these instructions while arithmetic and logic operations are taken care of by an Arithmetic and Logic Unit (ALU). A modern day CPU can execute millions of instructions per second (MIPS).

Memory: A computer memory, usually a semiconductor device, is used to hold data and instructions. Depending on the make, it could be volatile or non-volatile in nature. There are different types of memory:

1. Read Only Memory (ROM): It is a non-volatile storage entity. It is used in computers, phones, modems, watches and other electronic devices. A program is typically uploaded (flashed) to ROM through PC. Its content

cannot be modified; it can only be erased and flashed using compatible tools.

2. Random-access Memory: RAM is a volatile storage entity. It is used by CPU to store intermediate data during the execution of a program. RAM is usually faster than ROM.
3. Electronically Erasable Programmable Read-Only Memory: EEPROM is an optional non-volatile storage entity. It can be erased and written by the running program. For example, it can be used to store the values of a temperature sensor connected to the microcontroller.

2.1.2 Microcontroller Peripherals

Microcontrollers have a few built-in peripherals. In this section, we will review them briefly.

Clock: A complex digital circuit, such as the one that is present in a microcontroller, requires a clock pulse to synchronize different parts of it. The clock is generated through internal or external crystal oscillator. A typical microcontroller can execute one instruction per clock cycle (time between two consecutive clock pulses).

Timer/Counter: A timer is a pulse counter. A timer circuit is controlled by registers. An 8-bit timer can count from 0 to 255. A timer is primarily used to generate delay, and could be configured to count events.

Input/Output Ports: I/O ports correspond to physical pins on the microcontroller. They are used to interface external peripherals. A port can be configured as input or output by setting bits in I/O registers. Each pin can be individually addressed too.

Interrupts: An interrupt to the CPU suspends the running program and executes a code block corresponding to it. After serving/attending interrupts, the CPU resumes the previous program and continues. An interrupt could be originated by the software or the hardware. A hardware interrupt normally has a higher priority.

Universal Asynchronous Receiver/Transmitter (UART): UART is a standard microcontroller peripheral to communicate with external serial enabled devices. It has two dedicated pins to be used as Rx (Receiver) and Tx (Transmitter). The baud rate defines the speed of the UART and can be configured using registers.

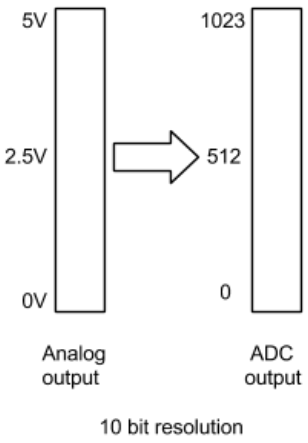


Figure 2.2: ADC resolution

Analog to Digital Converter (ADC): Most of the signals around us are continuous. Digital circuits cannot process them. An ADC converts them into digital signals. The resolution of the ADC determines the efficiency of conversion. For example, a 10-bit resolution of the ADC relates to 1024 values per sample. This is shown pictorially in Fig. 2.2. Higher resolution relates to better translation of an analog signal.

Digital to Analog Converter (DAC): Digital output of the CPU is converted to analog signals using the pulse width modulation (PWM) technique. The output of a DAC is used to drive analog devices and actuators.

Serial Peripheral Interface (SPI): SPI is a synchronous 4 wire serial communication device. It requires a master and slave configuration. The SPI peripheral has dedicated pins and marked as:

1. SCLK (from Master)
2. MOSI (Master out, Slave input)
3. MISO (Master Input, Slave output)
4. Slave select (Active when 0V, originates from Master)

Firmware: Firmware is an application that configures the hardware. It is programmed to a non-volatile memory such as ROM, EPROM (Erasable Programmable ROM). This concept is used in computer BIOS and embedded

Figure 2.3: The logo of Open Source Hardware

devices. In a microcontroller setup, a firmware file contains addresses and hexadecimal values.

Interfacing: Some of the popular connections with microcontrollers include,

1. Digital input devices: Switch, keypad, encoder, multiplexer, touchscreen
2. Digital output devices: LED, LCD, relay, buzzer
3. Digital input and output devices: RTC (Real Time Clock), SD Card, external ROM
4. Analog input devices: Audio, sensor, potentiometer
5. Analog output devices: Brightness control, speaker
6. Serial communication (UART): GSM, GPS, Zigbee, Bluetooth

2.2 Open Source Hardware (OSHW)

In this section, we will introduce the reader to Open Source Hardware (OSHW), which is *defined* as follows [3]:

> Open source hardware is a hardware whose design is made publicly available so that anyone can study, modify, distribute, make, and sell the design or hardware based on that design...

The OSHW website [3] gives additional conditions to be fulfilled before the hardware can be called OSHW. It also argues why we should promote and contribute to OSHW. The logo of OSHW is given in Fig. 2.3 [4]. The open-source hardware initiative is popular in the electronic, computing hardware and automation industry. Here are some examples of open-source hardware projects:

1. The "open compute project" at Facebook shares the design of data center products.
2. Beagle board, Panda board, OLinuXino are ARM based development boards.

3. "Open Graphics Project (OGP)" releases the designs of graphics card.
4. "ArduCopter" is a UAV (unmanned aerial vehicle) created by the *DIY Drones* community.
5. "NetFPGA" is a prototyping of computer network devices.
6. "OpenROV" project (Open Source Remotely Operated Vehicle) aims at affordable underwater exploration.
7. "OpenMoko" project set the foundation for open-source mobile phones. "Neo 1973" was the first smartphone released in 2007 with Linux based operating system, it had 128MB RAM and 64MB ROM.

Companies like Adafruit Industries, Texas Instruments, Solarbotics, Sparkfun electronics, MakerBot industries and DIY Drones have proven the power of OSHW with their revenues. Nevertheless, collaborative innovation using OSHW is yet to establish itself in the mainstream. But the trend has certainly started and is going strong. There are now many robotics startups taking full use of OSHW.

2.3 Arduino

Arduino is an open-source microcontroller board and a software development environment. Arduino language is a *C* like programming language which is easy to learn and understand. Arduino has two components, open source hardware and open source software. We will cover the basics of the Arduino hardware in this section.

2.3.1 Brief History

Arduino project was started at the *Interaction Design Institute Ivrea* in Ivrea, Italy. The aim was to create a low-cost microcontroller board that anyone with little or no background domain knowledge can design and develop. Arduino uses expansion circuit boards known as *shields*. Shields can provide GPS, GSM, Bluetooth, Zigbee, motor and other functionality.

Within the first two years of its inception, the Arduino Team sold more than 50,000 boards. In 2011, Google announced *The Android Open Accessory Development Kit (ADK)*, which enables the Arduino boards to interface with Android mobile platform.

Today Arduino is the first choice for electronic designers and hobbyists. There are more than 13 official variants of Arduino and many more third-party Arduino software compatible boards.

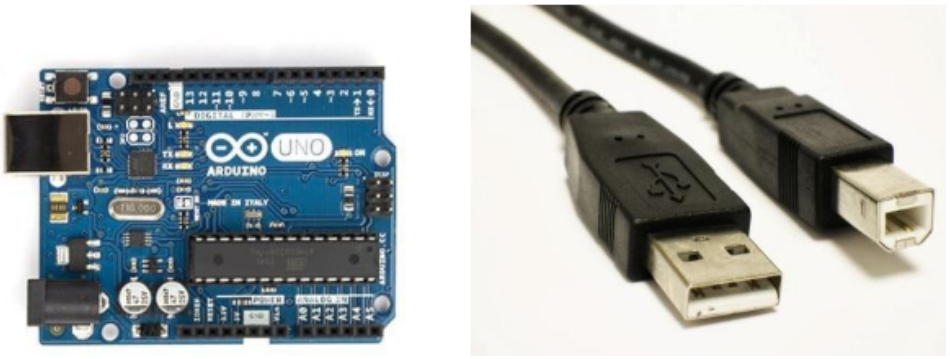


Figure 2.4: Arduino Uno Board

2.3.2 Arduino Uno Board

There are different Arduino boards for different requirements. All original Arduino boards are based on ATMEL microcontrollers. In this section, we will briefly discuss the Arduino Uno board, the most popular Arduino board. We will illustrate all applications using the Arduino Uno board in this book.

Based on ATmega328, the Arduino Uno board has 14 digital input/output pins, 6 analog inputs, 6 PWM pins, a 16 MHz ceramic resonator, a power jack, an ICSP (In-Circuit Serial Programming) header, and a reset button. It has an on-board USB to serial converter and can be connected to a PC using a USB cable. Fig. 2.4 has a picture of this board [5]. Table 2.1 has the specifications of the Arduino Uno board.

Another popular board is Arduino Mega board. Based on ATmega2560, this board has almost double the size of program memory (ROM) compared to Arduino Uno. It also has extra serial ports, digital and PWM pins. Fig. 2.5 has a picture of this board [6].

Yet another popular board is LilyPad Arduino, a small circular board for fabric designers. It can be stitched with conductive thread, and it supports sensors and actuators. Fig. 2.6 has a picture of this board [7].

There are other similar configuration boards with different form factors, such as Arduino Fio, Arduino Mini, Arduino Nano, Arduino Duemilanove, Arduino serial and so on.

2.3.3 Popular Arduino Projects

Arduino is intuitive and it's easy to setup and use. That's why people around the globe are using Arduino in innovative ways. We list a few of these projects to give

Parameter	Value
Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328), 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Table 2.1: Arduino Uno hardware specifications

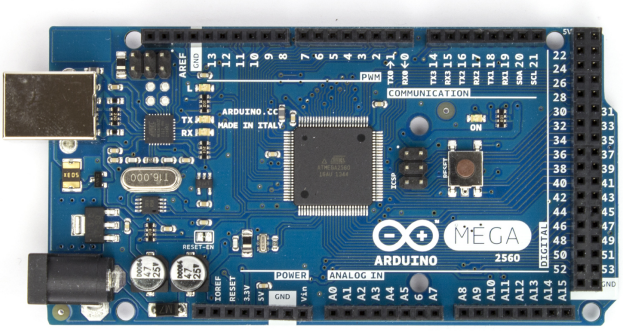


Figure 2.5: Arduino Mega Board

a flavor of some of these interesting applications.

Arduino phone: An Arduino connected with a graphic LCD and a GSM shield. This low-tech phone, shown in Fig. 2.7 can be built in a few hours [8].

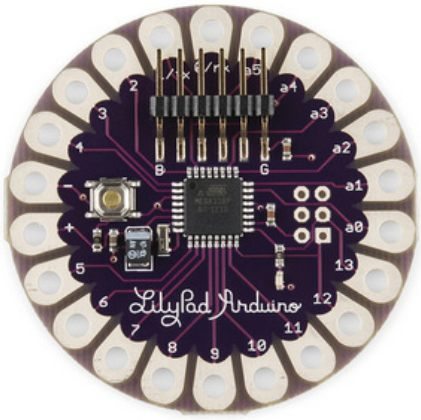


Figure 2.6: LilyPad Arduino Board

Figure 2.7: Arduino Phone

Candy sorting machine: As the name suggests, this machine can sort candy based on its color to separate jars [9].

3D printers: There are open-source 3D printers based on Arduino and Raspberry Pi. Although 3D printers, shown in Fig. 2.8, are relatively slow and lack precision, they can be ideal for building prototypes by hobbyists [10].

2.4 Shield

The shield that we use in this book is a modified version of the Diyode Codeshield board [11], which makes it easy to perform experiments on the Arduino Uno board.

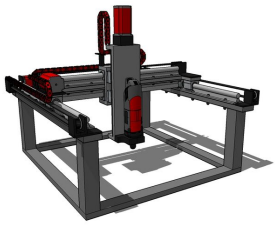


Figure 2.8: 3D printer

The shield is a printed circuit board (PCB) with a large number of sensors, already wired and hence, ready to use. It obviates the need for a breadboard as an intermediate tool for electronics circuit prototyping, which is quite cumbersome for beginners. The shield provides the user a faster way of circuit prototyping without worrying much about troubleshooting.

The numbering on the shield is identical to that on the Arduino Uno board. The shield fits snugly on to the Arduino Uno board, obviating the need to do the wiring in many experiments. One can even say that shields have made the hardware experiments involving Arduino boards as easy as writing software.

All the experiments in this book have been verified with the use of a modified version of Diyode Codeshield, as mentioned above. We make available all the required information to make a shield, thus making this an OSHW, see Sec. 2.2.

We now explain where the required files to make our shield are given. The gerber file to make the shield is given in `Origin/tools/shield/gerber-V1.2`, see Footnote 2 on page 2. The image of the PCB file is given in Fig. 2.9. The PCB project files are available in a folder at `Origin/tools/shield/kicad-import`, see Footnote 2 on page 2. The pictorial representation of the schematic for the shield is given in Fig. 2.10. A photograph of the PCB after fabrication is given in Fig. 2.11.

The values of the various components used in the shield are given in Table 2.2. Table 2.3 provides information about various sensors, components on shield and its corresponding pin on Arduino Uno board [11]. A picture of the completed shield is in Fig. 2.12.

2.5 Experimental Test Bed

We experimented with the contents of this book with the following list. We will refer to this as a *kit* in the rest of this book.

1. Arduino Uno board
2. Shield containing

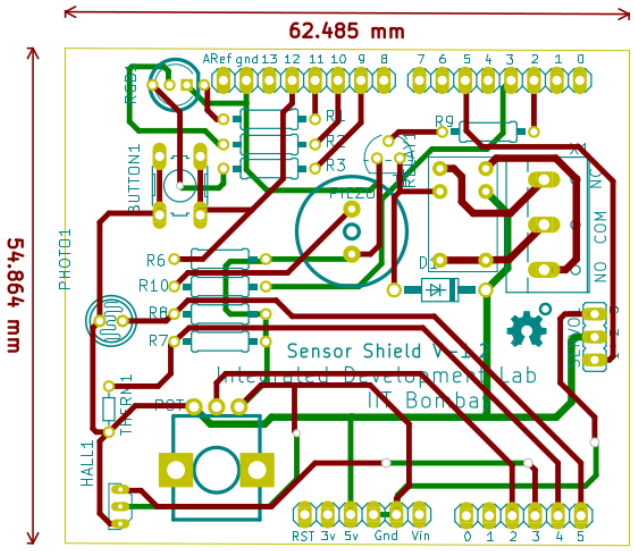


Figure 2.9: PCB image of the shield

   (a) LED
   (b) LDR
   (c) Push Button
   (d) Thermistor

3. DC motor and its controller board

4. Servomotor

5. Energy meter with Modbus interface

The Arduino Uno board is easily available in the market. The shield is designed by us. Details of most of these units are provided in the previous sections. Information on all of these is available in the file, mentioned in Footnote 2.

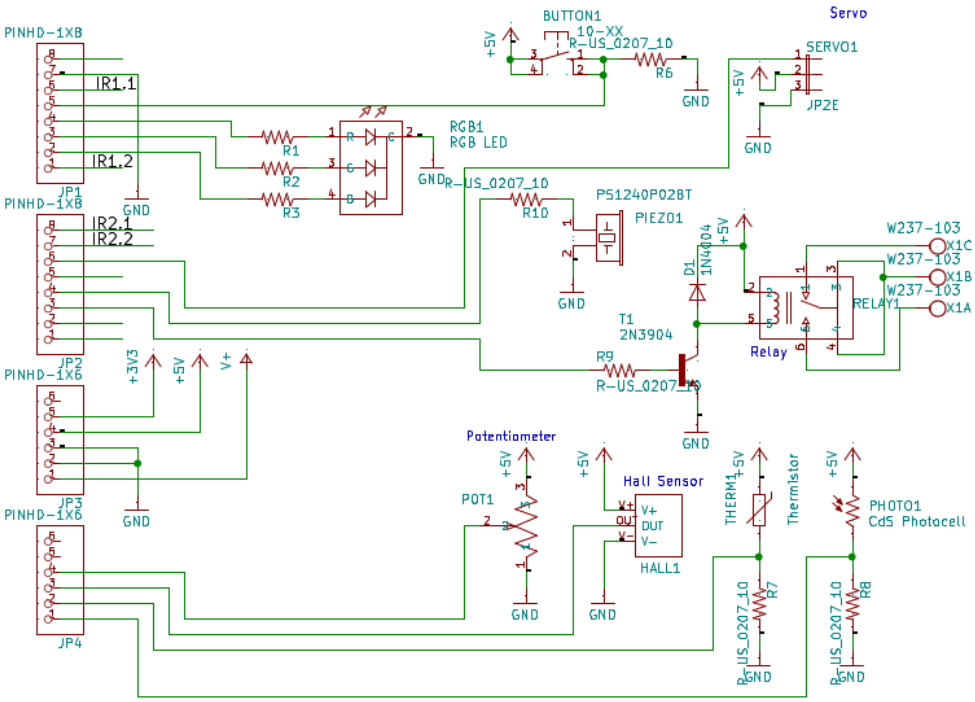


Figure 2.10: Pictorial representation of the schematic of the shield

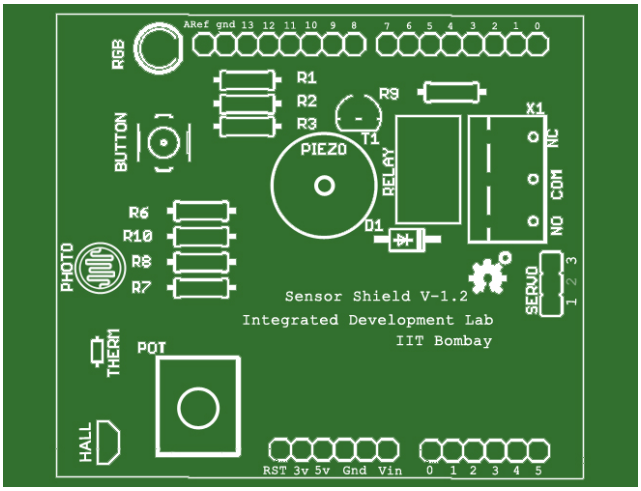


Figure 2.11: PCB of the shield

Table 2.2: Values of components used in the shield

Name	Description	Quantity
R1, R10	100Ω Resistor (Br-Bl-Br)	2
R2, R3	91Ω Resistor (Wt-Br-Bl)	2
R6, R7, R8	$10K\Omega$ Resistor (Br-Bl-Or)	3
R9	$1K\Omega$ Resistor (Br-Bl-Rd)	1
D1	Diode	1
Relay	Relay	1
X1	Terminal block	1
Piezo	Buzzer	1
RGB	RGB LED	1
T1	Transistor	1
BUTTON	Pushbutton	1
PHOTO	Light dependent resistor	1
HALL	Hall effect sensor	1
POT	Potentiometer	1
THERM	Thermistor	1
SERVO	Servomotor	1
HEADER	6x pin header	2
HEADER	8x pin header	2

Table 2.3: Information on sensors and pin numbers

Shield components	Arduino pin
RELAY	Digital pin 2
BUZZER	Digital pin 3
SERVO	Digital pin 5
RGB LED BLUE	Digital pin 9
RGB LED GREEN	Digital pin 10
RGB LED RED	Digital pin 11
PUSHBUTTON	Digital pin 12
POTENTIOMETER	Analog pin 2
HALL EFFECT SENSOR	Analog pin 3
THERMISTOR	Analog pin 4
PHOTORESISTOR (LDR)	Analog pin 5

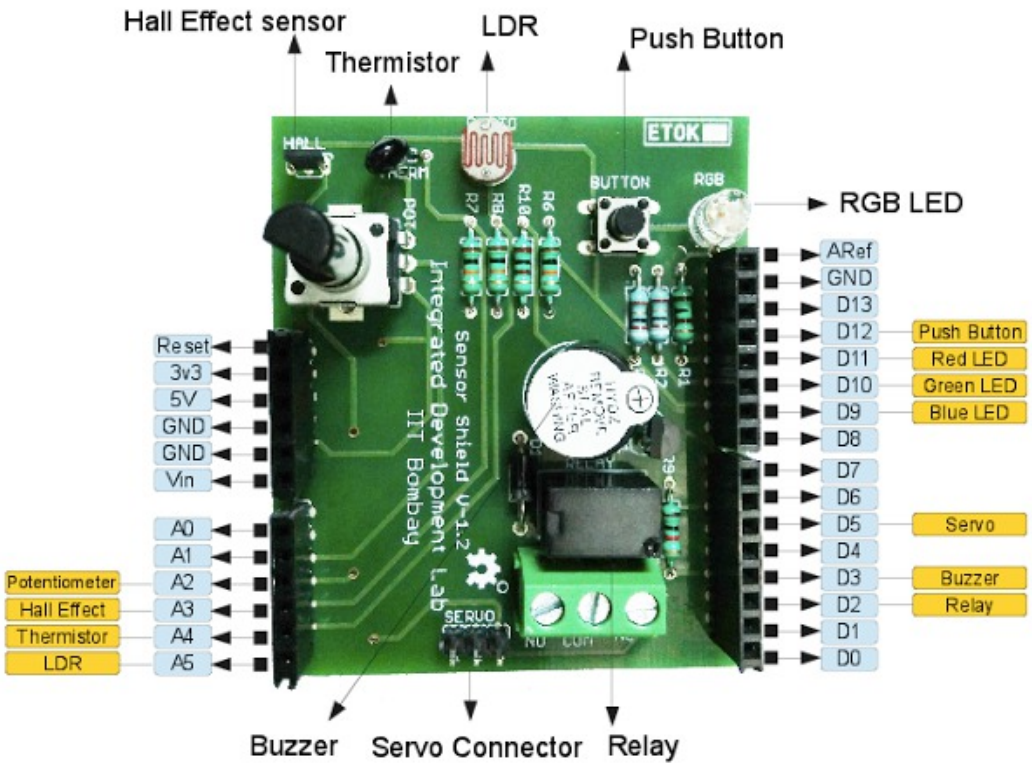


Figure 2.12: Picture of the shield with all components

Chapter 3

Communication between Software and Arduino

In this chapter, we shall briefly walk through the software environment that needs to be set up before we could start with the Arduino Uno board-based experiments. We shall start with the Arduino Uno compatible Integrated Development Environment (IDE), termed as Arduino IDE, that would be used to load the FLOSS firmware on to the microcontroller. The FLOSS firmware to be loaded could be developed to serve different purposes as per the requirement. For example,

- To run Arduino Uno stand-alone, without waiting for any commands from other software or hardware, for the specified time or until power off
- To decode the commands sent by other software, such as Scilab, Python, Julia, OpenModelica, etc., through a serial port, and execute the given instructions

Next, we shall discuss other open-source software tools and a related toolbox that can communicate with Arduino Uno over a serial port using RS232 protocol.

3.1 Arduino IDE

Arduino development environment is compatible with popular desktop operating systems. In this section, we will learn to set up this tool for the computers running Microsoft Windows or Linux. Later, we shall explore the important menu options in the Arduino IDE and run a sample program. The following two steps have to be followed whatever operating system is used:

1. To begin, we need an Arduino Uno board with a USB cable (A plug to B plug) as shown in Fig. 2.4.

2. Connect it to a computer and power it up. The moment you connect Arduino Uno to the computer, an on-board power LED will turn ON.

3.1.1 Downloading and installing on Windows

First, carry out the steps numbered 1 and 2 given above. Starting from download, we shall go through the steps to set up Arduino IDE on Windows OS:

3. Visit the URL, `https://www.arduino.cc/en/software`. On the right right side of the page, locate the link *Windows ZIP file* and click on it. This may redirect you to the download/donate page. Read the instructions and proceed with the download.

4. Extract the downloaded ZIP file to Desktop. Do not alter any file or directory structure.

5. Click on the Windows Start Menu, and open up the "Control Panel".

6. While in the Control Panel, navigate to "System and Security", click on "System" and then choose the "Device Manager".

7. Look for "Other devices" in the "Device Manager" list, expand and locate "Unknown device". This may be similar to what is shown in Fig. 3.1. In case, you don't see "Unknown device," look for "Ports (COM & LPT)" and expand it to locate "USB Serial Device (COM2)". This may be similar to what is shown in Fig. 3.2.

8. Right-click on the "Unknown device" (or "USB Serial Device (COM2)" as shown in the previous step) and select the "Update Driver Software" (or "Update driver") option as shown in Fig. 3.3.

9. Next, choose the "Browse my computer for Driver software" option.

10. Navigate to the newly extracted Arduino folder on the Desktop and select "drivers" folder.

11. Windows will now finish the driver installation. The Arduino IDE is ready for use.

To launch Arduino IDE, browse to extracted Arduino folder on the Desktop and double click on "arduino.exe".

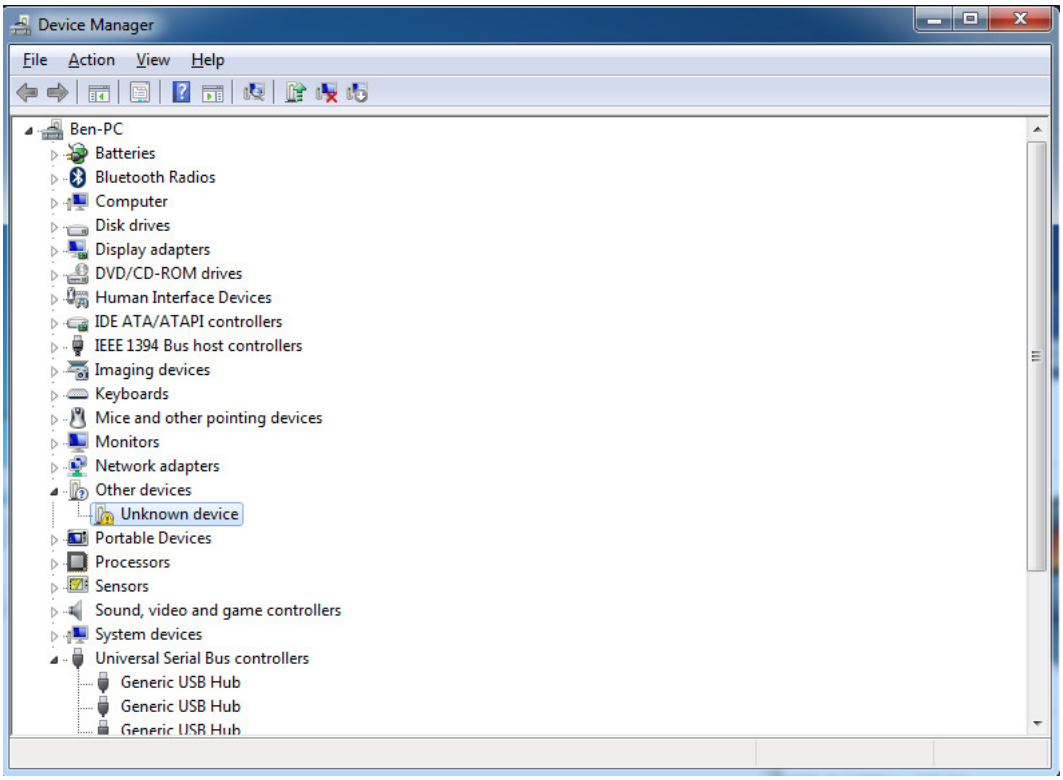


Figure 3.1: Windows device manager

3.1.2 Downloading and installing on GNU/Linux Ubuntu

We will now explain the installation of Arduino software on the GNU/Linux operating system. We shall perform the installation on the 64-bit Ubuntu 18.04 LTS operating system. These instructions will work for other GNU distributions too, with little or no modification. First, carry out the steps numbered 1 and 2 given above. Then carry out the following:

3. First, update your system. Open the terminal emulator, type, `sudo apt-get update` and press Enter.

4. Find out your operating system support for 64-bit instructions. Open the terminal emulator and type, `uname -m`

5. If it returns "x86_64", then your computer has 64-bit operating system. There is no visible performance difference in 32 and 64-bit Arduino versions.

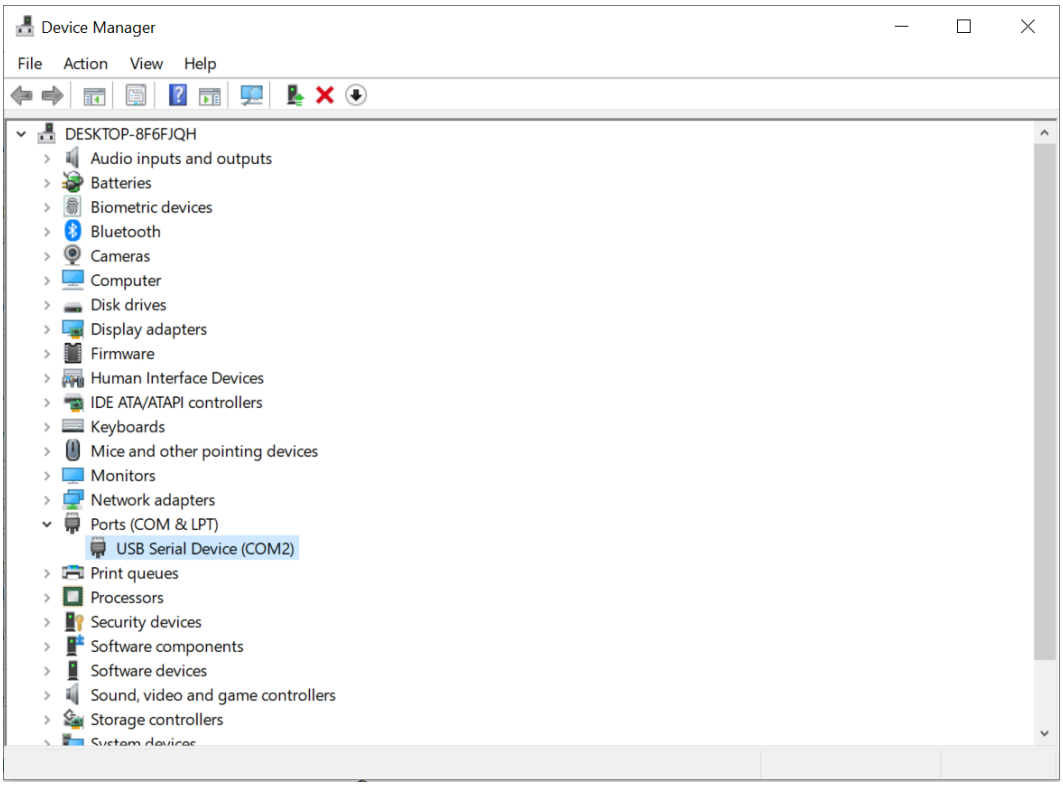


Figure 3.2: Windows device manager

6. Download the suitable Arduino Software version (32 or 64-bit) from `https://www.arduino.cc/en/software`. As mentioned earlier, we will perform experiments with a 64-bit installation.

7. At the time of writing this book, we worked with version 1.8.13. Assuming that you have downloaded the tar file in the Downloads directory, execute the following commands on the terminal:

```
cd ~/Downloads
tar -xvf arduino-1.8.13-linux64.tar.xz
sudo mv arduino-1.8.13 /opt
```

8. In the same terminal session, install the required Java Runtime Environment with a command like, `sudo apt-get -y install openjdk-8-jre`

9. Execute the following command on the terminal to list the serial port number. `ls /dev/ttyACM*`

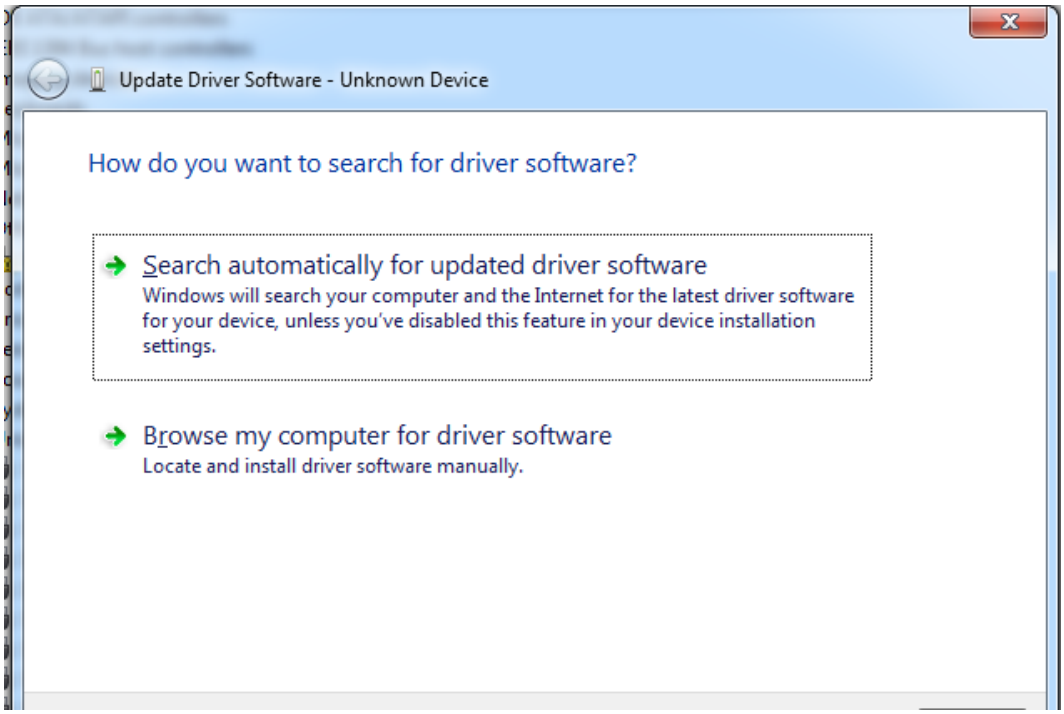


Figure 3.3: Windows update driver option

Note down the serial device filename. Suppose that it is `ttyACM0`.

10. To make the USB port available to all users, set the read-write permission to the listed port: `sudo chmod a+rw /dev/ttyACM0`. Each time you plug the Arduino Uno into the computer, you need to execute the commands given in the steps numbered 9 and 10.

The Arduino IDE is now ready for use. To launch it, carry out the steps given below:

1. Open a terminal by pressing the Alt+Ctrl+T keys together.
2. Navigate into the `opt` directory, as shown in Fig. 3.4.
   `cd /opt/arduino-1.8.13/`
3. Start the Arduino IDE by executing the command `./arduino`

3.1.3 Arduino Development Environment

The Arduino development environment, as shown in Fig. 3.5, consists of a text editor for writing code, a message area, a text console, a toolbar with buttons for common

Figure 3.4: Linux terminal to launch Arduino IDE

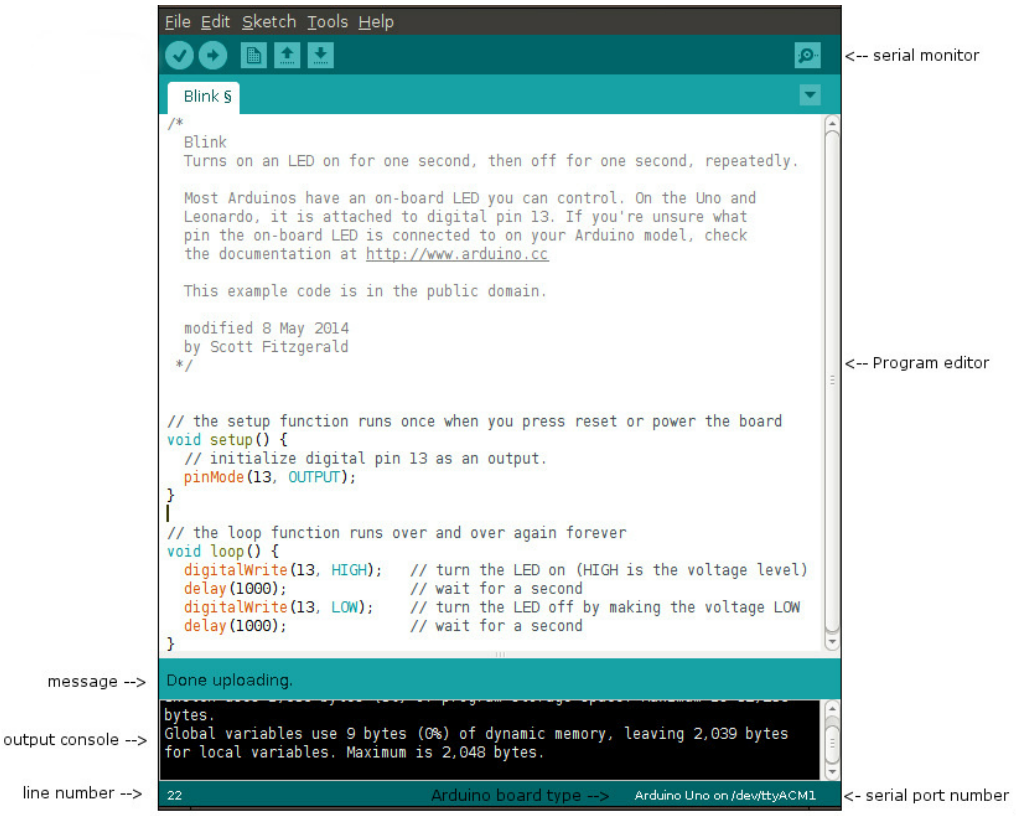


Figure 3.5: Arduino IDE

functions, and a series of menus. It connects to the Arduino hardware to upload programs and communicate with them.

Software written using Arduino is called sketches. These sketches are written in

the text editor. Sketches are saved with the file extension ".ino". The frequently used icons shown in the toolbar, below the menu bar, are explained next. The names of these icons can be viewed by hovering the mouse pointer over each of them.

1. Verify: Checks your code for errors
2. Upload: Compiles your code and uploads it to the Arduino I/O board
3. New: Creates a new sketch
4. Open: Presents a menu of all the sketches in your sketchbook - clicking one will open it within the current window
5. Save: Saves your sketch
6. Serial Monitor: Opens the serial port window - the location of this is shown in the top right-hand corner of Fig. 3.5

Note that these appear from left to right in the editor window. Next, we shall go through the additional useful options under the menu.

1. File
   (a) Examples: Examples that come at the time of installation
   (b) Page Setup: Configures the page parameters for the printer
   (c) Preferences: Customizes font, language, and other parameters for the IDE
2. Sketch
   (a) Include Library: Adds a library to your sketch by inserting `#include` statements at the start of your code
3. Tools
   (a) Auto Format: Indents code so that opening and closing curly braces line up
   (b) Archive Sketch: Archives a copy of the current sketch in .zip format. The archive is placed in the same directory as the sketch.
   (c) Board: Selects the board that you're using
   (d) Port: This menu contains all the serial devices (real or virtual) on your machine. It should automatically refresh every time you open the top-level tools menu.

(e) Programmer: This can be used to select a hardware programmer when programming a board or chip and not using the onboard USB-serial connection. Normally you won't need this, but if you're burning a bootloader to a new microcontroller, you will use this.

(f) Burn Bootloader: The items in this menu allow you to burn a bootloader onto the microcontroller on an Arduino board. This is not required for normal use of an Arduino board but is useful if you purchase a new ATmega microcontroller (which normally comes without a bootloader). Ensure that you've selected the correct board from the Boards menu before burning the bootloader.

3.1.4 Testing Arduino with a sample program

Now, as we have a basic understanding of Arduino IDE, let us try an example program.

1. Open the Arduino IDE by clicking the shortcut "arduino" from Desktop in Ubuntu. In MS Windows browse to extracted Arduino folder on Desktop and double click on "arduino.exe".

2. In the Arduino IDE, to know the path of your sketch files, navigate to File, then Preferences and then locate the "Sketchbook location" text box at the top. You may change the path of your storage location. In this book, we will keep it unchanged. The path will be different for Windows and Ubuntu.

3. To load a sample program, navigate and click on sketch "File", then Examples, then 01.Basics, and then Blink.

4. A new IDE instance will open with Blink LED code. You may close the previous IDE window now.

5. Click "verify" to compile. The "status bar" below the text editor shall show "Done compiling" on success.

6. Connect Arduino UNO board to PC. You may connect the board before writing the sketch too.

7. Now, navigate to "Tools", then Port, and select the available port. If the port option is greyed out (or disabled) then reinsert the USB cable to the PC.

8. Now select the upload button to compile and send the firmware to the Arduino Uno board.

9. If the upload is successful, you will notice the onboard orange LED next to the Arduino logo will start blinking.

10. It is safe to detach the USB cable at any moment.

Arduino programming syntax is different from other languages. In an embedded setup, a program is expected to run forever. To facilitate this, the Arduino programming structure has two main functions: `setup()`: Used to initialize variables, pin modes, libraries, etc. The setup function will run only once after each powerup or board reset. `loop()`: Code inside this function runs forever. An Arduino program must have `setup()` and `loop()` functions. We will give several examples in this book to explain this usage.

An inbuilt offline help is available within the IDE. You may access the explanation on IDE by navigating to "Help" and then Environment.

3.1.5 FLOSS Firmware

We have provided a code to check whether the FLOSS firmware has been properly installed. The first few lines of this code follow.

Arduino Code 3.1 First 10 lines of the FLOSS firmware. Available at `Origin /tools/floss-firmware/floss-firmware.ino`, see Footnote 2 on page 2. Following the steps given in sections 3.1.3 and 3.1.4, open this code in Arduino IDE and upload it to Arduino Uno. Once the upload is successful, you should expect a success message at the bottom of Arduino IDE, as shown in Fig. 3.5.

```
1  /* This file is meant to be used with the SCILAB arduino
2     toolbox, however, it can be used from the IDE environment
3     (or any other serial terminal) by typing commands like:
4
5     Conversion ascii -> number
6     48->'0' ... 57->'9' 58->':' 59->';' 60->'<' 61->'=' 62->'>' 63->'?'
       64->'@'
7     65->'A' ... 90->'Z' 91->'[' 92->'\' 93->']' 94->'^' 95->'_' 96->'`'
8     97->'a' ... 122->'z'
9
10    Dan0 or Dan1 : attach digital pin n (ascii from 2 to b) to input (0)
        or output (1)
```

3.2 OpenModelica

OpenModelica is a free and open-source environment based on the Modelica modeling language for simulating, optimizing, and analyzing complex dynamic systems [12]. It is a powerful tool that can be used to design and simulate complete control systems.

In the upcoming sections, we have provided the steps to install OpenModelica on Windows and Linux. After installing OpenModelica, the readers should watch the tutorials on OpenModelica provided on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the second and third tutorials, i.e., `Introduction to OMEdit` and `Examples through OMEdit`.

3.2.1 Downloading and installing on Windows

This book uses Stable Development of OpenModelica 1.17.0 for demonstrating the experiments, both on Windows and Linux. It may be noted that OpenModelica requires approximately 8 GB of space for its installation. Starting from download, we shall go through the steps to set up OpenModelica 1.17.0 on Windows OS:

1. Visit the URL `https://openmodelica.org/`. At the top of the page, locate the Download tab. On hovering the cursor on this tab, a drop-down menu appears. In that menu, click on Windows.

2. From the section Download Windows, click on the binaries 1.17.0 (32bit/64bit) next to the Stable Development of OpenModelica.

3. A webpage named Index of /omc/builds/windows/releases/1.17/0 appears. Now, click on 32-bit or 64-bit depending on your operating system. We will continue with a 64-bit installation.

4. Once you select 64-bit, a webpage named Index of /omc/builds/windows/releases/1.17/0/64bit appears. You should get a list of files here. Click on the executable (.exe) file to download the binaries for OpenModelica.

5. Locate the executable file and double-click on it to begin the installation. After double-clicking on the executable file, you might get an alert on your screen (something like Windows protected your PC). If this happens, locate More info in this alert window and click on Run Anyway, as shown in Fig. 3.6 to continue with the installation. All the default parameters of the installation are acceptable.

Once OpenModelica has been installed, OpenModelica Connection Editor (OMEdit) can be launched from the Start menu. When you launch OMEdit for the first time, you might get a notification for setting up Modelica Standard Library version, as shown in Fig. 3.7. Here, you should choose the option "Load MSL v3.2.3" and click OK. To know how to execute models in OMEdit, the readers are advised to refer to Sec. 3.2.3.

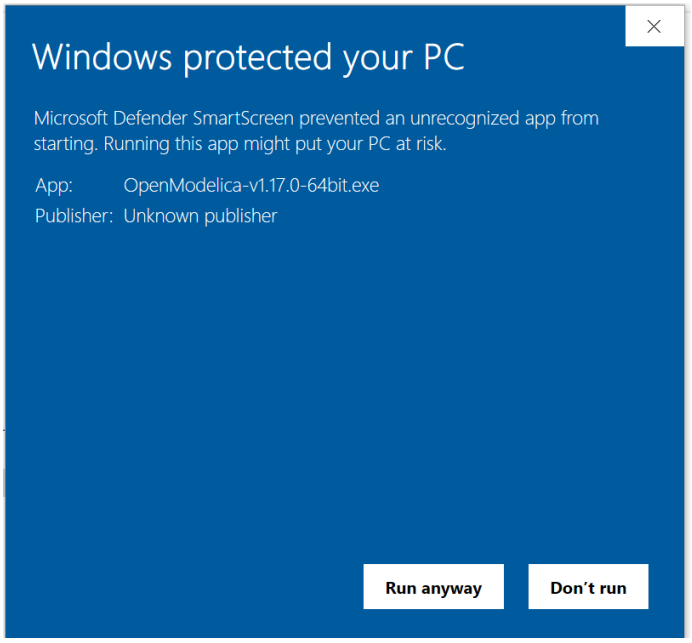


Figure 3.6: Allowing Microsoft Defender to run the executable file

3.2.2 Downloading and installing on GNU/Linux Ubuntu

On Linux, we can install OpenModelica from the terminal. The readers are advised to visit `https://openmodelica.org/download/download-linux` and follow the instructions for installing OpenModelica. We recommend the readers should install the latest stable version of OpenModelica. Once OpenModelica has been installed successfully, OpenModelica Connection Editor (OMEdit) can be launched from the terminal. Open a terminal by pressing Alt+Ctrl+T and type OMEdit. When you launch OMEdit for the first time, you might get a notification for setting up Modelica Standard Library version, as shown in Fig. 3.7. Here, you should choose the option "Load MSL v3.2.3" and click OK. To know how to execute models in OMEdit, the readers are advised to refer to Sec. 3.2.3.

3.2.3 Simulating models in OpenModelica

Once you launch OMEdit (either on Windows or on Linux Ubuntu), you should expect a user interface, as shown in Fig. 3.8. In the bottom right of Fig. 3.8, we can see that there are four different tabs - Welcome, Modeling, Plotting, and Debugging. In the language of OpenModelica, we refer to these tabs as Perspectives. By default, OMEdit gets launched in the Welcome Perspective. We now briefly describe each of

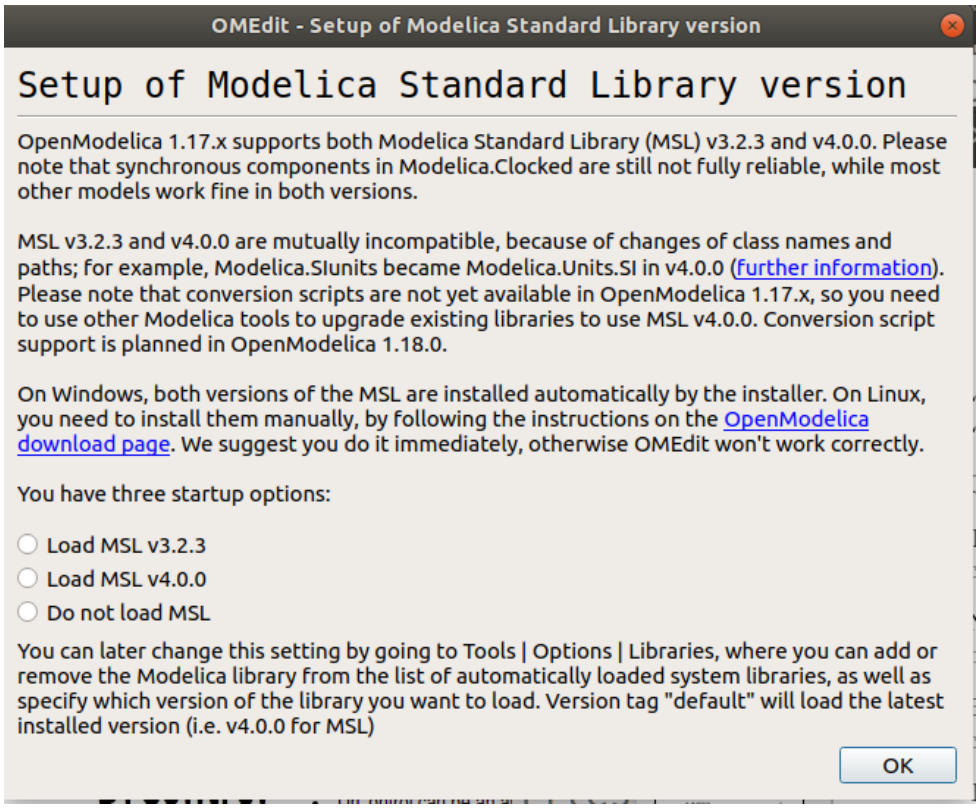


Figure 3.7: Setup of Modelica Standard Library version

these Perspectives, as given below:

1. Welcome Perspective: It shows the list of recent files and the list of the latest news from `https://www.openmodelica.org`.

2. Modeling Perpective: It provides the interface where users can create and design their models.

3. Plotting Perspective: It shows the simulation results of the models. Plotting Perspective will automatically become active when the simulation of the model is finished successfully.

4. Debugging Perspective: The application automatically switches to Debugging Perspective when the user simulates the class with an algorithmic debugger [12].

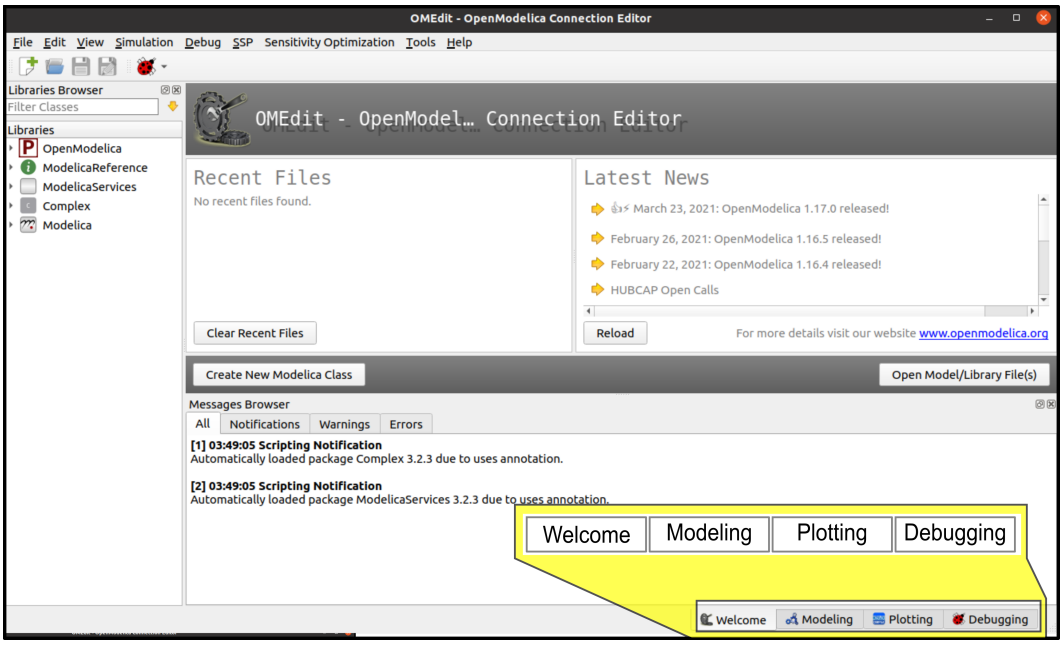


Figure 3.8: User Interface of OMEdit

In the left of Fig. 3.8, there is Libraries Browser below which you can view the list of libraries loaded in your current session of OMEdit. By default, OMEdit comes with a few default libraries, like Modelica, ModelicaReference, etc., as shown in Fig. 3.8. These default libraries might not be visible if you have not set up the Modelica Standard Library version, as given in Fig. 3.7.

The files or models in OpenModelica have '.mo' extensions. Though there are several ways to simulate or run an OpenModelica model, we will execute the models by utilizing the user interface of OMEdit. To open a model in OMEdit, go to the menu bar of OMEdit and click on File -> Open Model/Library File(s), as shown in Fig. 3.9. Then, select the desired model (with '.mo' extension) and click Open. The names of tabs in this book have been mentioned according to OpenModelica 1.17.0. You might observe a bit of difference in these names while working with other versions of OpenModelica.

Once you have opened the model in OMEdit, that model should appear under the Libraries browser, as shown in Fig. 3.8. To view or simulate that model, you need to double-click on the model. It will open the model in Modeling perspective with a Diagram View, as shown in the Fig. 3.10. In this perspective, there are four different views of a model: Icon View, Diagram View, Text View, and Documentation View. All these views have been highlighted in Fig. 3.11. By default, OMEdit opens any

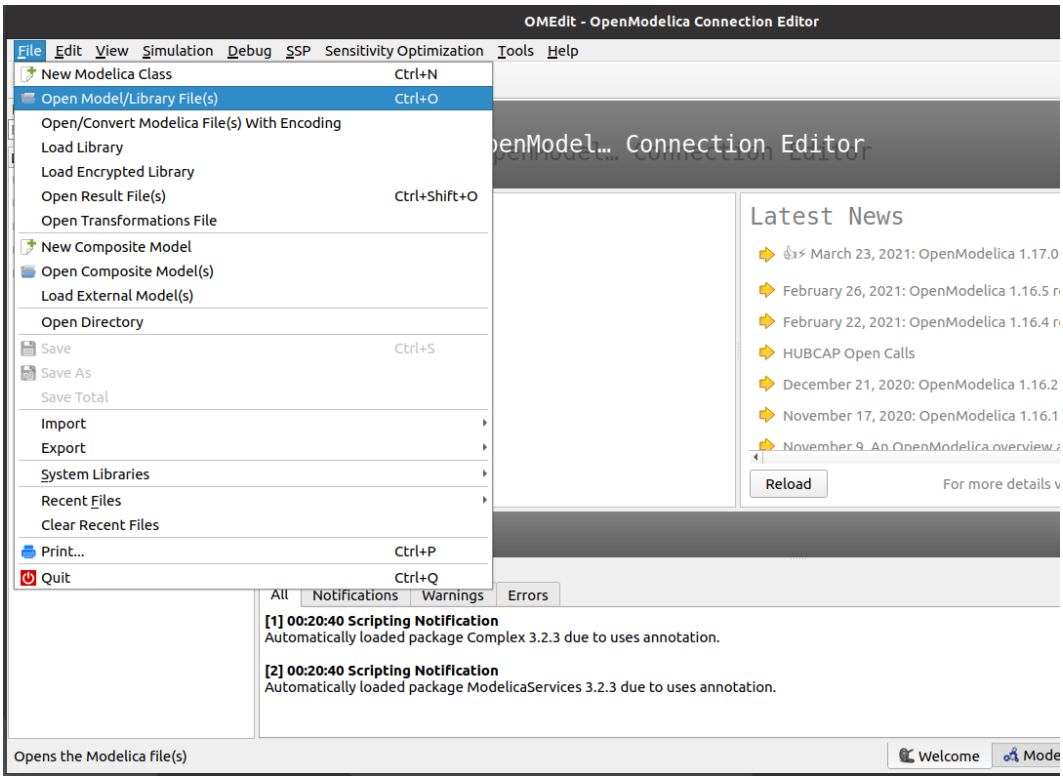


Figure 3.9: Opening a model in OMEdit

model in Diagram View. Hence, the models having code in text format won't be visible by default in Modeling Perspective. To see the code in text format, we need to open the model in Text View. For our experiments, we will use Text view mainly. To view the code written for this model, we need to click on Text View, as shown in Fig. 3.11. In Text view, the code is now visible, as shown in Fig. 3.12. Now, one can modify the model as per the requirements.

Now, we will see how to simulate this model. For this, we need to ensure that OMEdit is in Modeling Perspective. Next, we will click on the green right-sided arrow, named as Simulate, as shown in Fig. 3.13. When we click on Simulate, OMEdit will first compile the model and then, it will simulate the model for the time specified in the model itself. As OMEdit provides an elegant user interface for simulating the models, it will open an output window the moment you click on Simulate. Fig. 3.14 shows the output window after the simulation of our model is finished. Also, we can observe that the OMEdit is now in Plotting Perspective.

As shown in Fig. 3.14, OMEdit displays the message that "The Simulation fin-

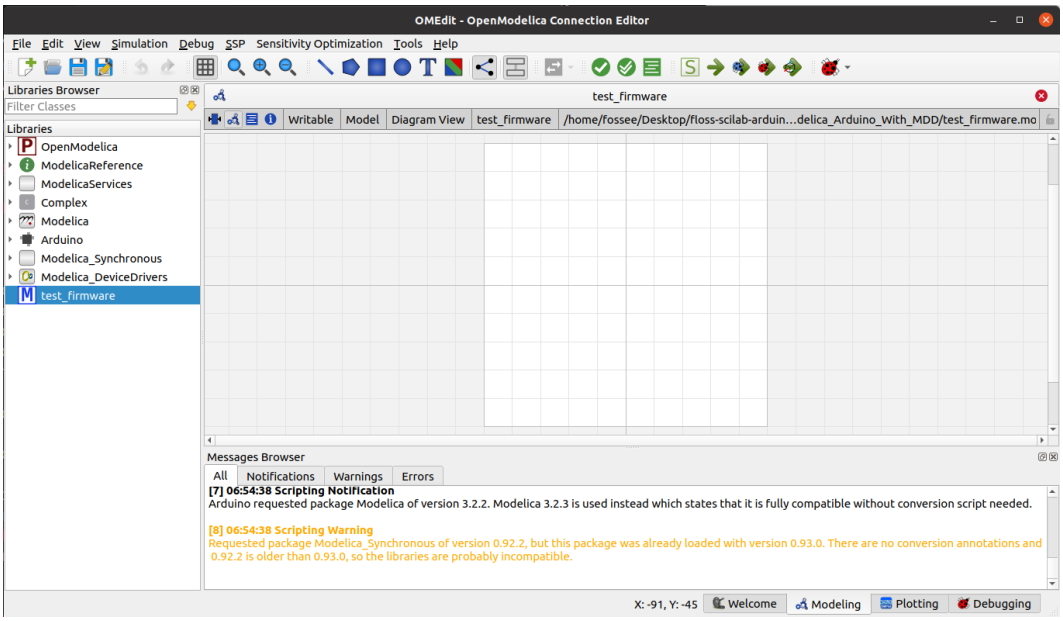


Figure 3.10: Opening a model in diagram view in OMEdit

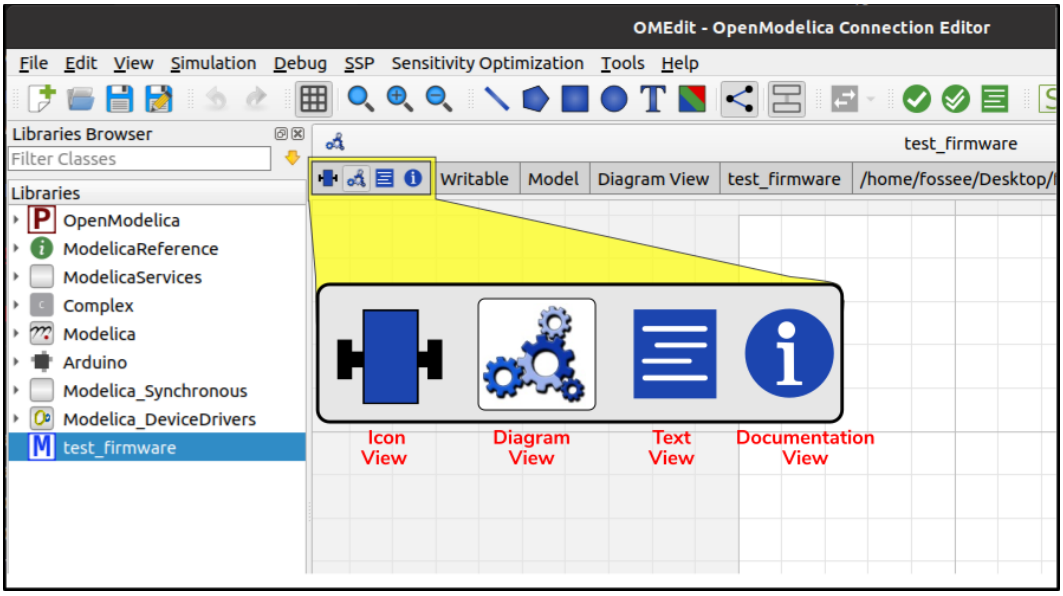


Figure 3.11: Different views of a model in OMEdit

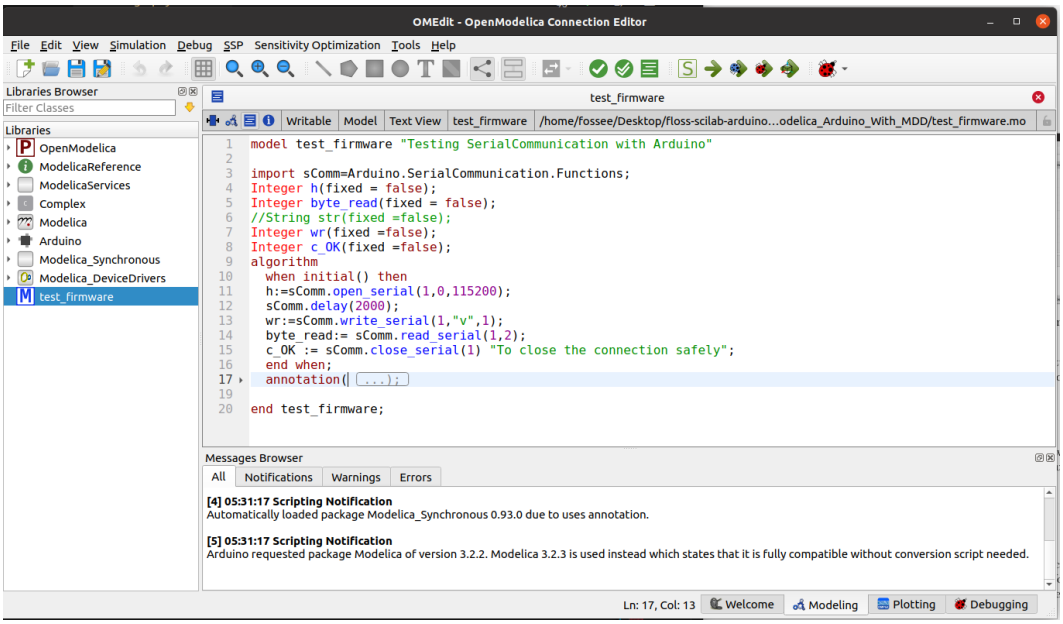


Figure 3.12: Opening a model in text view in OMEdit

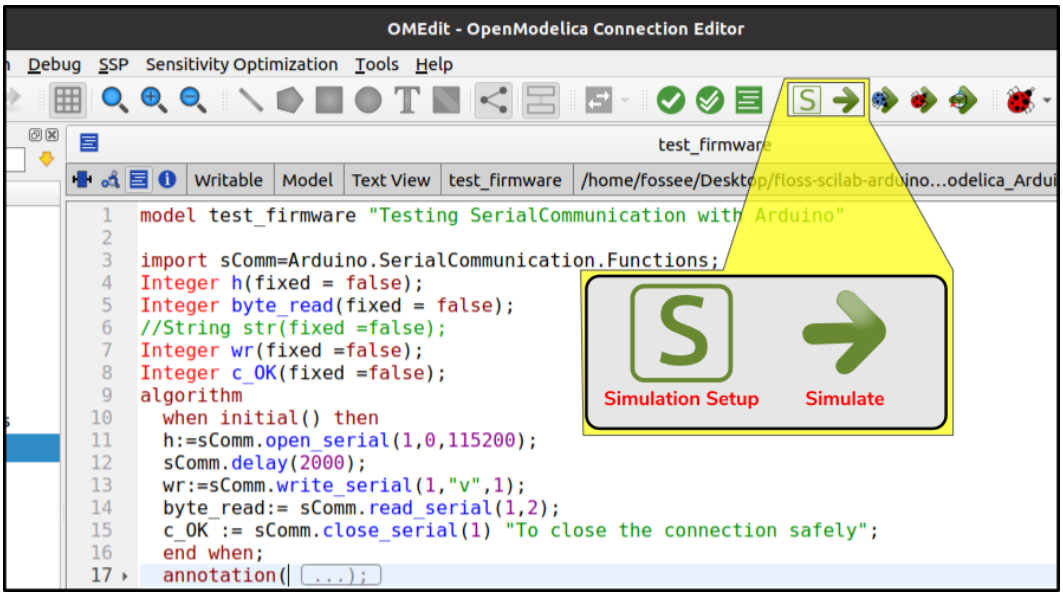


Figure 3.13: Simulating a model in OMEdit

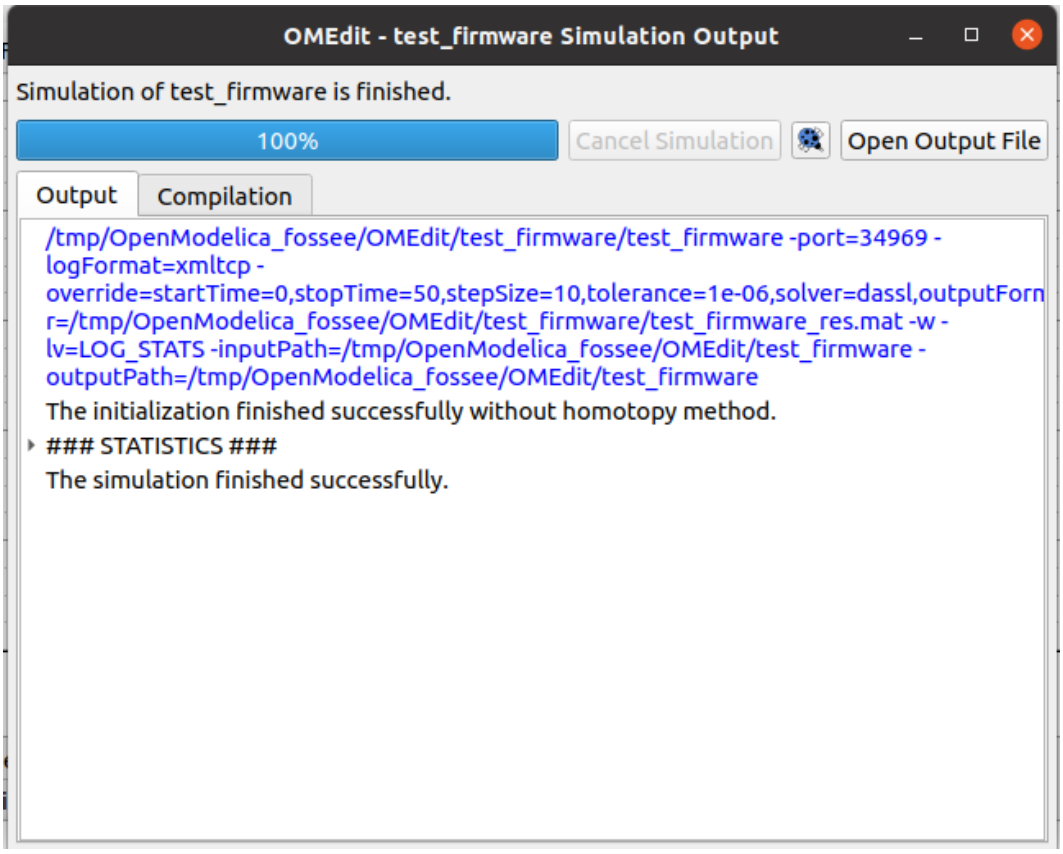


Figure 3.14: Output window of OMEdit

ished successfully". Had there been any error in simulating the model, we would not have received this message.

3.2.4 OpenModelica-Arduino toolbox

OpenModelica, by default, does not have the capability to connect to Arduino. All such add-on functionalities are added to OpenModelica using toolboxes. The OpenModelica Arduino toolbox can be found inside `Origin/tools/openmodelica/windows` or `Origin/tools/openmodelica/linux` directory, see Footnote 2 on page 2. Use the one depending upon which operating system you are using. The OpenModelica codes for various experiments mentioned throughout this book can be found in `Origin/user-code` directory. The `user-code` directory will have many sub-directories as per the experiments.

Let us now see how to load the OpenModelica Arduino toolbox.

1. First launch OMEdit. On a Windows system, one may start/launch OMEdit from the Start menu. On a Linux system, one has to start OMEdit through a terminal, as explained in section 3.2.2.

2. After launching, we have to load OpenModelica-Arduino toolbox. To do so, go to the menu bar of OMEdit. Click on `File` and then click on the `Open Model/Library File(s)` option as shown in Fig. 3.9.

3. Navigate to `Origin/tools/openmodelica/windows` or `Origin/tools/openmodelica/linux`, as the case maybe. Select `Arduino.mo` and `test_firmware.mo`, and click Open. The toolbox should now be loaded and available for use.

4. We will check whether the toolbox has been loaded successfully or not. In OMEdit, under Libraries Browser, look for three new libraries: Arduino, Modelica_Synchronous, Modelica_DeviceDrivers, and one model test_firmware.mo. If you are able to view these files, it means that OpenModelica-Arduino toolbox has been loaded successfully. Please note that each time you launch OMEdit, you need to load this toolbox for performing the experiments.

5. Now, we will locate the models for running the experiments in the upcoming chapters. Under Libraries Browser, click on the arrow before Arduino to see the contents inside this package. Next go to SerialCommunication -> Examples. Under Examples, you will find the list of experiments like led, push, etc., as shown in Fig. 3.15.

6. For running the experiments of a particular chapter, click on the corresponding experiment's name. Subsequently, you will find a list of experiments which you can simulate one by one, as explained in Sec. 3.2.3.

7. For each of the experiments using OpenModelica given in the upcoming chapters, the readers are advised to locate the corresponding model by following the steps numbered 5 and 6 and simulate it.

3.2.5 Firmware

We have provided an OpenModelica code/model to check whether the firmware has been properly installed. That code/model is listed below. Please ensure that you have uploaded the FLOSS firmware given in Arduino Code 3.1 on the Arduino Uno board.

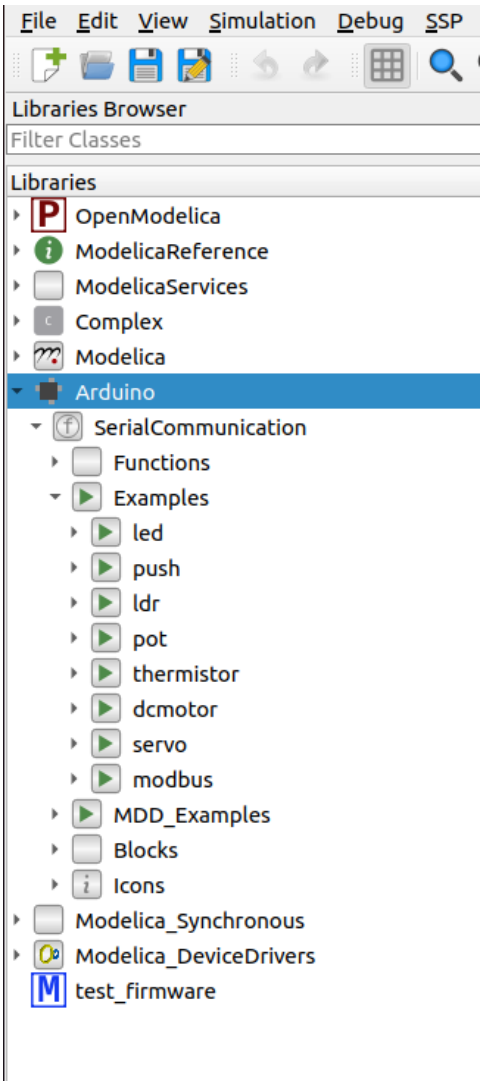


Figure 3.15: Examples provided in the OpenModelica-Arduino toolbox

OpenModelica Code 3.1 An OpenModelica code/model to check whether the firmware is properly installed or not. Available at `Origin/tools/openmodelica/windows/test_firmware.mo`, see Footnote 2 on page 2. Locate this file inside OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Simulate this code/model by following the steps given in Sec. 3.2.3. If the simulation is successful, you should expect an output window in OMEdit as shown in Fig. 3.14.

```
1 model test_firmware "Testing SerialCommunication with Arduino"
2
3 import sComm=Arduino.SerialCommunication.Functions;
4 Integer h(fixed = false);
5 Integer byte_read(fixed = false);
6 //String str(fixed =false);
7 Integer wr(fixed =false);
8 Integer c_OK(fixed =false);
9 algorithm
10   when initial() then
11   h:=sComm.open_serial(1,2,115200);
12   sComm.delay(2000);
13   wr:=sComm.write_serial(1,"v",1);
14   byte_read:= sComm.read_serial(1,2);
15   c_OK := sComm.close_serial(1) "To close the connection safely";
16   end when;
17   annotation(
18     experiment(StartTime = 0, StopTime = 50, Tolerance = 1e-6, Interval
       = 10));
19
20 end test_firmware;
```

Chapter 4

Interfacing a Light Emitting Diode

In this chapter, we will learn how to control the LEDs on the shield and on the Arduino Uno board. We will do this through the Arduino IDE and other open-source software tools. These are beginner level experiments, and often referred to as the *hello world* task of Arduino. Although simple, controlling LED is a very important task in all kinds of electronic boards.

4.1 Preliminaries

A light emitting diode (LED) is a special type of semiconductor diode, which emits light when voltage is applied across its terminals. A typical LED has 2 leads: Anode, the positive terminal and Cathode, the negative terminal. When sufficient voltage is applied, electrons combine with the holes, thereby releasing energy in the form of photons. These photons emit light and this phenomenon is known as electroluminescence. The symbolic representation of an LED is shown in Fig. 4.1. Generally, LEDs are capable of emitting different colours. Changing the composition of alloys that are present in LED helps produce different colours. A popular LED is an RGB LED that actually has three LEDs: red, green and blue.

An RGB LED is present on the shield provided in the kit. In this section, we will see how to light each of the LEDs present in the RGB LED. As a matter of fact,

Figure 4.1: Light Emitting Diode

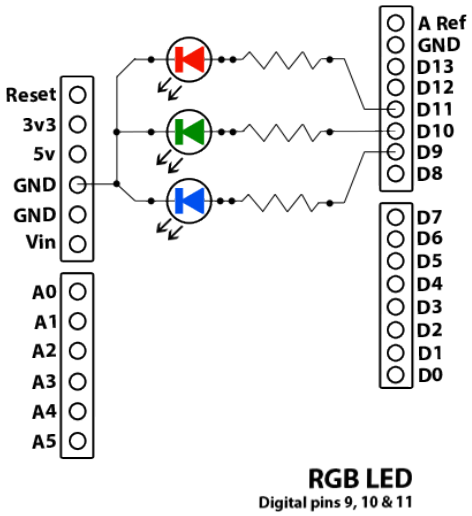


Figure 4.2: Internal connection diagram for the RGB LED on the shield

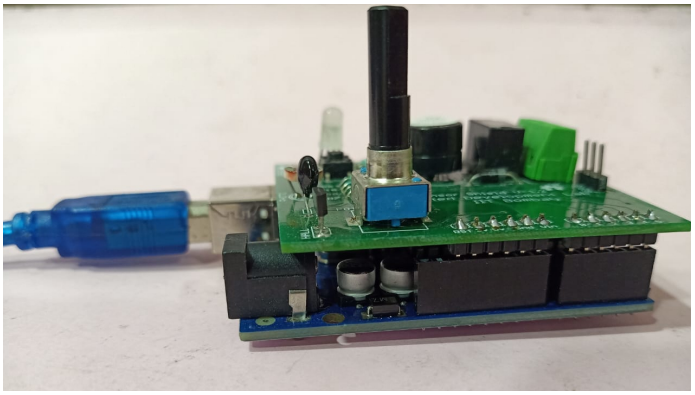


Figure 4.3: Connecting Arduino Uno and shield

it is possible to create many colours by combining these three. A schematic of the RGB LED in the shield is given in Fig. 4.2. The anode pins of red, green, and blue are connected to pins 11, 10, and 9, respectively. Common Cathode is connected to the ground.

It should be pointed out, however, that no wire connections are to be made by the learner: all the required connections are already made internally and it is ready to use. The LED of any colour can be turned on by putting a high voltage on the corresponding anode pin.

One should remember to connect the shield on to the Arduino Uno board, as

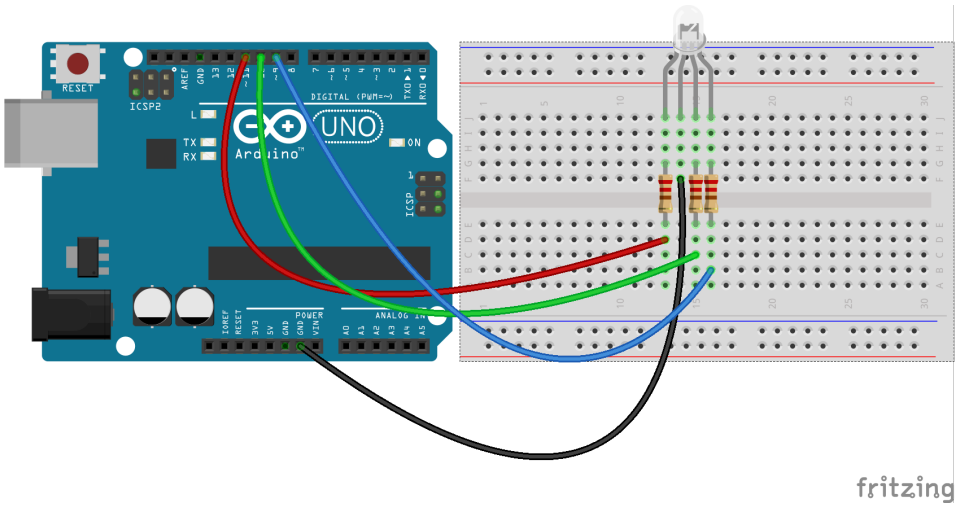


Figure 4.4: An RGB LED with Arduino Uno using a breadboard

shown in Fig. 4.3. All the experiments in this chapter assume that the shield is connected to the Arduino Uno board. It is also possible to do some of the experiments without the shield, which is pointed out in the next section.

4.2 Connecting an RGB LED with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with Tricolor LED and Push button`.

In case you have an RGB LED and want to connect it with Arduino Uno on a breadboard, please refer to Fig. 4.4. As shown in Fig. 4.4, there is an RGB LED with four legs. From the left, the first leg represents the anode (+) pin for the red LED. The second leg represents the common cathode for every color. The third and fourth legs represent the anode (+) pins for the green LED and blue LED respectively. The

anode pins of red, green, and blue are connected to digital pins 11, 10, and 9 of Arduino Uno, respectively. Common cathode is connected to the ground (GND) terminal of Arduino Uno.

4.3 Lighting the LED from the Arduino IDE

4.3.1 Lighting the LED

In this section, we will describe some experiments that will help the LED light up based on the command given from the Arduino IDE. We will also give the necessary code. We will present four experiments in this section. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

1. First, we will see how to light up the LED in different colours. An extremely simple code is given in Arduino Code 4.1. On uploading this code, you can see that the LED on the shield turns blue. It is extremely easy to explain this code. Recall from the above discussion that we have to put a high voltage (5V) on pin 9 to turn the blue light on. This is achieved by the following command:

   ```
   digitalWrite(9, HIGH);
   ```

   Before that, we need to define pin 9 as the output pin. This is achieved by the command,

   ```
   pinMode(9, OUTPUT);
   ```

   One can see that the blue light will be on continuously.

2. Next, we will modify the code slightly so that the blue light remains on for two seconds and then turns off. Arduino Code 4.2 helps achieve this. In this, we introduce a new command `delay` as below:

   ```
   delay(2000);
   ```

   This delay command halts the code for the time passed as in input argument. In our case, it is 2,000 milliseconds, or 2 seconds. The next command,

   ```
   digitalWrite(9, LOW);
   ```

   puts a low voltage on pin 9 to turn it off.

   What is the role of the `delay` command? To find this, comment the delay command. That is, replace the above delay command with the following and upload the code.

```
// delay(2000);
```

If you observe carefully, you will see that the LED turns blue momentarily and then turns off.

3. We mentioned earlier that it was possible to light more than one LED simultaneously. We will now describe this with another experiment. In this, we will turn on both blue and red LEDs. We will keep both of them on for 5 seconds and then turn blue off, leaving only red on. After 3 seconds, we will turn red also off. This code is given in Arduino Code 4.3. Remember that before writing either `HIGH` or `LOW` on to any pin, its mode has to be declared as `OUTPUT`, as given in the code. All the commands in this code are self explanatory.

4. Finally, we will give a hint of how to use the programming capabilities of the Arduino IDE. For this, we will use Arduino Code 4.4. It makes the LED blink 5 times. Recall from the previous section that a `HIGH` on pin 10 turns on the green LED. This cycle is executed for a total of five times. In each iteration, it will turn the green LED on for a second by giving the `HIGH` signal and then turn it off for a second by giving the `LOW` signal. This cycle is carried out for a total of 5 times, because of the `for` loop.

Note: All the above four experiments have been done with the shield affixed to the Arduino Uno board. One may run these experiments without the shield as well. But in this case, pin number 13 has to be used in all experiments, as pin 13 lights up the LED that is on the Arduino Uno board. For example, in Arduino Code 4.1, one has to replace both occurrences of number 9 with 13. In this case, one will get the LED of Arduino Uno board light up, as shown in Fig. 4.5.

Note: It should also be pointed out that only one colour is available in Arduino Uno board. As a result, it is not possible to conduct the experiments that produce different colours if the shield is not used.

Exercise 4.1 Carry out the following exercise:

1. In Arduino Code 4.2, remove the delay, as discussed above, and check what happens.
2. Light up all three colours simultaneously, by modifying Arduino Code 4.3. Change the combination of colours to get different colours.
3. Incorporate some of the features of earlier experiments into Arduino Code 4.4 and come up with different ways of blinking with different colour combinations.

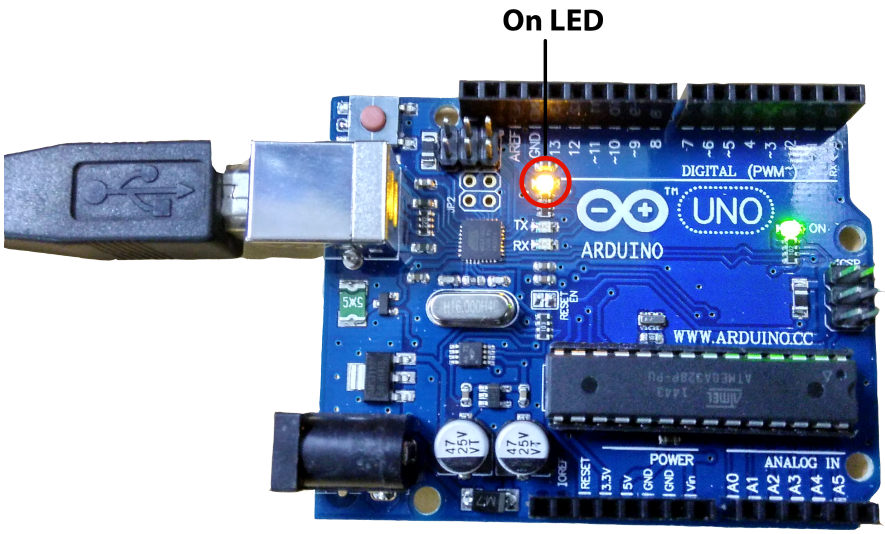


Figure 4.5: LED experiments directly on Arduino Uno board, without the shield

4.3.2 Arduino Code

Arduino Code 4.1 Turning on the blue LED. Available at `Origin/user-code/led/arduino/led-blue/led-blue.ino`, see Footnote 2 on page 2.

```
1 void setup() {
2 pinMode(9, OUTPUT);
3 Serial.begin(115200);
4 digitalWrite(9, HIGH);
5 }
6 void loop() {
7 }
```

Arduino Code 4.2 Turning on the blue LED and turning it off after two seconds. Available at `Origin/user-code/led/arduino/led-blue-delay/led-blue-delay.ino`, see Footnote 2 on page 2.

```
1 void setup() {
2 pinMode(9, OUTPUT);
```

```
3 Serial.begin(115200);
4 digitalWrite(9, HIGH);
5 delay(2000);
6 digitalWrite(9, LOW);
7 }
8 void loop() {
9 }
```

Arduino Code 4.3 Turning on blue and red LEDs for 5 seconds and then turning them off one by one. Available at `Origin/user-code/led/arduino/led-blue-red/led-blue-red.ino`, see Footnote 2 on page 2.

```
1 void setup() {
2 pinMode(9, OUTPUT);
3 pinMode(11, OUTPUT);
4 Serial.begin(115200);
5 digitalWrite(9, HIGH);
6 digitalWrite(11, HIGH);
7 delay(5000);
8 digitalWrite(9, LOW);
9 delay(3000);
10 digitalWrite(11, LOW);
11 }
12 void loop() {
13 }
```

Arduino Code 4.4 Blinking the green LED. Available at `Origin/user-code/led/arduino/led-green-blink/led-green-blink.ino`, see Footnote 2 on page 2.

```
1 int i = 0;
2 void setup() {
3   pinMode(10, OUTPUT);
4   Serial.begin(115200);
5   for(i = 0; i < 5; i++)
6   {
7   digitalWrite(10, HIGH);    // turn the LED on (HIGH is the voltage
      level)
8   delay(1000);               // wait for a second
9   digitalWrite(10, LOW);     // turn the LED off by making the voltage
      LOW
10  delay(1000);               // wait for a second
11  }
12 }
13 void loop() {
14 }
```

4.4 Lighting the LED from OpenModelica

4.4.1 Lighting the LED

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same four experiments, in the same order. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

1. In the first experiment, we will light up the blue LED on the shield. The code for this is given in OpenModelica Code 4.1. It begins with importing the two packages: Streams and SerialCommunication from the toolbox, as given in Sec. 3.2.4. Following line imports this package:

```
1  import sComm = Arduino.SerialCommunication.Functions;
2  import strm = Modelica.Utilities.Streams;
```

   We define some variables to collect the results coming from different functions. Following lines are used for these:

```
1  Integer ok(fixed = false);
2  Integer digital_out(fixed = false);
3  Integer c_ok(fixed = false);
```

   Now, we have a command of the form

```
ok := sComm.open_serial(1, PORT NUMBER, BAUD RATE)
```

   We have used 2 for `PORT NUMBER` and 115200 for `BAUD RATE`. As a result, this command becomes

```
1    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate
      of 115200";
```

   This command is used to open the serial port. When the port is opened successfully, it returns a value of 0, which gets stored in the variable `ok`.

   Sometimes, the serial port does not open, as mentioned in the above command. This is typically due to not closing the serial port properly in a previous experiment. If this condition is not trapped, the program will wait forever, without any information about this difficulty. One way to address this difficulty is to terminate the program if the serial port does not open. This is achieved using the error message of the following form:

```
if ok <> 0 then strm.print(Error Message in Quotes);
```

We turn the LED on in the upcoming lines. This is achieved using a command of the form

```
digital_out := sComm.cmd_digital_out(1, PIN NUMBER,
VALUE)
```

As we want to turn on the blue light in the shield, as discussed in Sec. 4.3.1, we choose `PIN NUMBER` as 9. We can put any positive integer in the place of `VALUE`. We arrive at the following command:

```
1    digital_out := sComm.cmd_digital_out(1, 9, 1) "This will
     turn ON the blue LED";
```

Subsequently, we close the serial port and then define the simulation parameters.

2. OpenModelica Code 4.2 does the same thing as what Arduino Code 4.2 does. It does two more things than what OpenModelica Code 4.1 does: It makes the blue LED light up for two seconds. This is achieved by the command

```
1    sComm.delay(2000) "let the blue LED be on for two seconds";
```

The second thing this code does is to turn the blue LED off. This is achieved by the command

```
1    digital_out := sComm.cmd_digital_out(1, 9, 0) "turn off blue
     LED";
```

It is easy to see that this code puts a 0 on pin 9.

3. OpenModelica Code 4.3 does the same thing as what Arduino Code 4.3 does. It turns blue and red LEDs on for five seconds. After that, it turns off blue first. After 3 seconds, it turns off red also. So, when the program ends, no LED is lit up.

4. OpenModelica Code 4.4 does exactly what its counterpart in the Arduino IDE does. It makes the green LED blink five times.

4.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 4.1 Turning on the blue LED. Available at Arduino -> SerialCommunication -> Examples -> led -> led_blue.

```
model led_blue "Turn on Blue LED"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer digital_out(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
    115200";
    if ok <> 0 then
      strm.print("Check the serial port and try again");
    else
      sComm.delay(1000);
      digital_out := sComm.cmd_digital_out(1, 9, 1) "This will turn ON
    the blue LED";
    end if;
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
//strm.print(String(time));
  annotation(
    experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
    = 10));
end led_blue;
```

OpenModelica Code 4.2 Turning on the blue LED and turning it off after two seconds. Available at Arduino -> SerialCommunication -> Examples -> led -> led_blue_delay.

```
model led_blue_delay "Turn on Blue LED for a period of 2 seconds"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer digital_out(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
    115200";
    sComm.delay(2000);
    if ok <> 0 then
      strm.print("Check the serial port and try again");
    else
```

```
15        digital_out := sComm.cmd_digital_out(1, 9, 1) "This will turn the
       blue LED";
16        sComm.delay(2000) "let the blue LED be on for two seconds";
17        digital_out := sComm.cmd_digital_out(1, 9, 0) "turn off blue LED"
      ;
18      end if;
19      c_ok := sComm.close_serial(1) "To close the connection safely";
20    end when;
21  //strm.print(String(time));
22    annotation(
23      experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 10));
24  end led_blue_delay;
```

OpenModelica Code 4.3 Turning on blue and red LEDs for 5 seconds and then turning them off one by one. Available at Arduino -> SerialCommunication -> Examples -> led -> led_blue_red.

```
1  model led_blue_red "Turn on Red & Blue LED"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer digital_out(fixed = false);
7    Integer c_ok(fixed = false);
8  algorithm
9    when initial() then
10     ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
       115200";
11     sComm.delay(2000);
12     if ok <> 0 then
13       strm.print("Check the serial port and try again");
14     else
15       digital_out := sComm.cmd_digital_out(1, 9, 1) "This will turn the
        blue LED";
16       digital_out := sComm.cmd_digital_out(1, 11, 1) "This will turn
       the red LED";
17       sComm.delay(5000) "Delay for 5 seconds";
18       digital_out := sComm.cmd_digital_out(1, 9, 0) "This turns off the
        blue Led";
19       sComm.delay(3000) "Delay for 3 seconds";
20       digital_out := sComm.cmd_digital_out(1, 11, 0) "This turns off
       the red Led";
21     end if;
22     c_ok := sComm.close_serial(1) "To close the connection safely";
23   end when;
24 //strm.print(String(time));
25   annotation(
```

```
26     experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 10));
27 end led_blue_red;
```

OpenModelica Code 4.4 Blinking the green LED. Available at Arduino -> SerialCommunication -> Examples -> led -> led_green_blink.

```
1  model led_green_blink "This will turn on and turn off the green LED for
       every second for 5 times"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer digital_out(fixed = false);
7    Integer c_ok(fixed = false);
8  algorithm
9    when initial() then
10     ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
       115200";
11     sComm.delay(2000);
12     if ok <> 0 then
13       strm.print("Check the serial port and try again");
14     else
15       for i in 1:5 loop
16         digital_out := sComm.cmd_digital_out(1, 10, 1) "This will turn
       off the green LED";
17         sComm.delay(1000) "Delay for 1 second";
18         digital_out := sComm.cmd_digital_out(1, 10, 0) "This turns the
       green Led";
19         sComm.delay(1000) "Delay for 1 second";
20       end for;
21     end if;
22     c_ok := sComm.close_serial(1) "To close the connection safely";
23   end when;
24 //     strm.print(String(time));
25   annotation(
26     experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 10));
27 end led_green_blink;
```

Chapter 5

Interfacing a Pushbutton

A pushbutton is a simple switch which is used to connect or disconnect a circuit. It is commonly available as a *normally open* or *push to make* switch which implies that the contact is made upon the push or depression of the switch. These switches are widely used in calculators, computer keyboards, home appliances, push-button telephones and basic mobile phones, etc. In this chapter, we shall perform an experiment to read the status of the pushbutton mounted on the shield of the Arduino Uno board. Advancing further, we shall perform a task depending on the status of the pushbutton. Digital logic based status monitoring is a very basic and important task in many industrial applications. This chapter will enable us to have a smooth hands-on for such functionalities.

5.1 Preliminaries

A pushbutton mounted on the shield is connected to the digital pin 12 of the Arduino Uno board. The connection diagram for the pushbutton is shown in Fig. 5.1. It has 2 pairs of terminals. Each pair is electrically connected. When the pushbutton is pressed all the terminals short to complete the circuit, thereby allowing the flow of current through the switch. As you might expect, there is a limit to the maximum current that could flow through a pushbutton. This maximum current is also called the rated current and is usually provided by the manufacturer in the datasheet.

5.2 Connecting a pushbutton with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

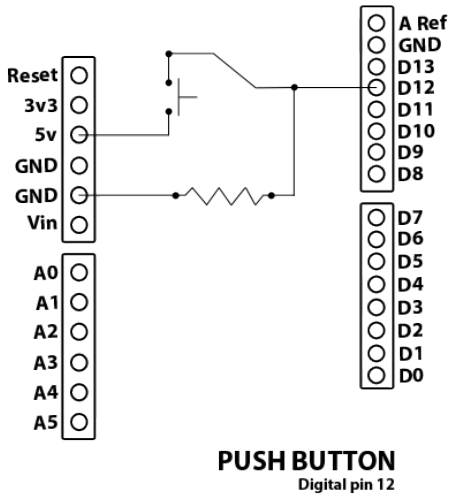


Figure 5.1: Internal connection diagram for the pushbutton on the shield

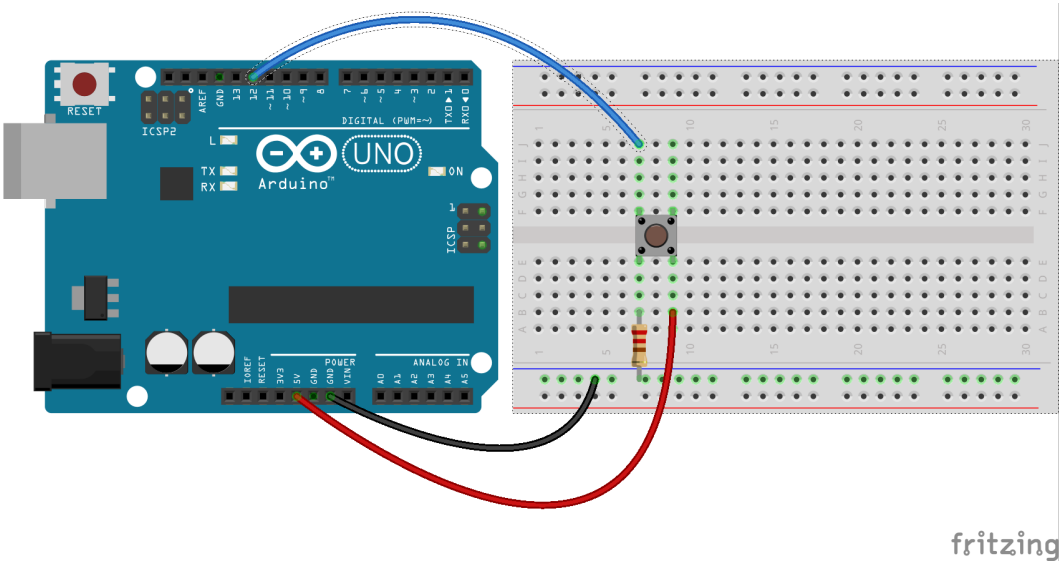


Figure 5.2: A pushbutton to read its status with Arduino Uno using a breadboard

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on

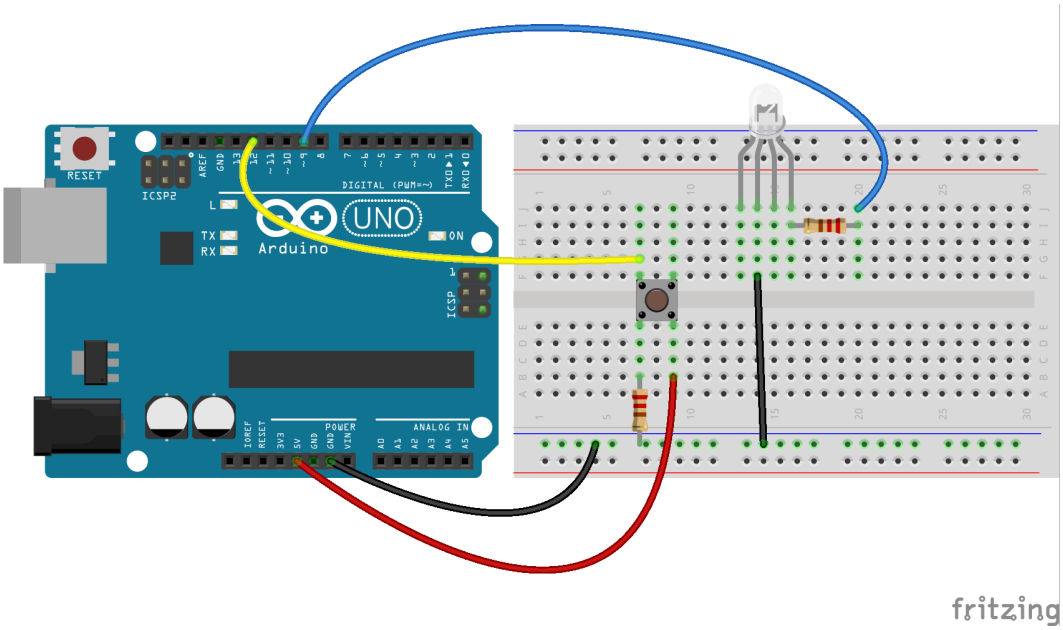


Figure 5.3: A pushbutton to control an LED with Arduino Uno using a breadboard

`https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with Tricolor LED and Push button`.

In case you have a pushbutton, and you want to connect it with Arduino Uno on a breadboard, please refer to Fig. 5.2. The connections given in this figure can be used to read the status of a pushbutton. As shown in Fig. 5.2, there are three different wires - red, black, and blue. The red wire is used to connect 5V on Arduino Uno and one leg of the pushbutton. The black wire connects to one long vertical row on the side of the breadboard to provide access to the ground (GND) on Arduino Uno. The blue wire goes from digital pin 12 to one leg of the pushbutton on another side. That same leg of the pushbutton connects through a pull-down resistor to GND on Arduino Uno. When the pushbutton is open (unpressed), there is no connection between the two legs of the pushbutton, so the pin is connected to the ground (through the pull-down resistor), and we read a LOW on digital pin 12. When the pushbutton is closed (pressed), it makes a connection between its two legs, connecting the pin to 5V so that we read a HIGH on digital pin 12.

The connections shown in Fig. 5.3 can be used to control an RGB LED, depending on the status of the pushbutton. As shown in Fig. 5.3, digital pin 9 on Arduino Uno

is connected to the rightmost leg of the RGB LED. Rest of the connections are same as that in Fig. 5.2.

5.3 Reading the pushbutton status from the Arduino IDE

5.3.1 Reading the pushbutton status

In this section, we shall describe an experiment that will help to read the status of a pushbutton through Arduino IDE. Later, we shall change the state of an LED depending on the status of the pushbutton. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

1. In the first experiment, we shall simply read the status of the pushbutton. Recall that it is a normally open type of switch. So, in an unpressed state, the logic read will be "0", corresponding to 0V. And, when the user presses the pushbutton, the reading would be "1", corresponding to 5V. The code for this experiment is given in Arduino Code 5.1. In the initialization part of the code, we assign the sensor pin to be read, 12 in this case, to a variable for ease. Next, we initialize the port for serial port communication at data rate of 115200 bits per second and declare the digital pin 12 as an input pin using the command `pinMode`. After initialization, we start reading the status of the pushbutton using the following command:

```
1     sensorValue = digitalRead(sensorPin); // read push-button
      value
```

   Note that the input argument to this command is the digital pin 12 corresponding to the pin to which the pushbutton is connected. After acquiring the values, we print them using,

```
1     Serial.println(sensorValue); // print it at the Serial
      Monitor
```

   We repeat this read and print process 50 times by putting the commands in a `for` loop. While running this experiment, the readers must press and release the pushbutton and observe the values being printed on the `Serial Monitor` of Arduino IDE.

2. In the second experiment, we shall control the power given to an LED as per the status of the pushbutton. The code for this experiment is given in

Arduino Code 5.2. This experiment can be taken as a step further to the previous one. We declare the LED pin to be controlled as an output pin by,

```
pinMode(sensorPin, INPUT);
```

Next, we read the pusbhutton value from digital pin 12. If the value is "1", we turn on the LED at pin 9 else we turn it off. The condition check is performed using `if else` statements. We run these commands for 50 iterations. While running this experiment, the readers must press and release the pushbutton. Accordingly, they can observe whether the LED glows when the pushbutton is pressed.

5.3.2 Arduino Code

Arduino Code 5.1 Read the status of the pushbutton and display it on the Serial Monitor. Available at `Origin/user-code/push/arduino/push-button-status/push-button-status.ino`, see Footnote 2 on page 2.

```
const int sensorPin = 12; // Declare the push-button
int sensorValue = 0;
void setup() {
  Serial.begin(115200);
  pinMode(sensorPin, INPUT);    // declare the sensorPin as an INPUT
  for (int i = 0; i < 50; i++){
     sensorValue = digitalRead(sensorPin); // read push-button value
     Serial.println(sensorValue); // print it at the Serial Monitor
     delay(200);
  }
}
void loop() {
}
```

Arduino Code 5.2 Turning the LED on or off depending on the pushbutton. Available at `Origin/user-code/push/arduino/led-push-button/led-push-button.ino`, see Footnote 2 on page 2.

```
const int sensorPin = 12;
const int ledPin = 9;
int sensorValue = 0;
int i;
void setup() {
  Serial.begin(115200);
  pinMode(sensorPin, INPUT);
  pinMode(ledPin, OUTPUT);
  for (i = 0; i < 50; i++) {
    sensorValue = digitalRead(sensorPin);
    Serial.println(sensorValue); // print it at the Serial Monitor
```

```
12      if (sensorValue == 0) {
13        digitalWrite(ledPin, LOW);
14        delay(200);
15      }
16      else {
17       digitalWrite(ledPin, HIGH);
18       delay(200);
19      }
20    }
21 }
22 void loop() {
23 }
```

5.4 Reading the pushbutton status from OpenModelica

5.4.1 Reading the pushbutton status

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same two experiments, in the same order. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

1. In the first experiment, we will read the pushbutton status. The code for this experiment is given in OpenModelica Code 5.1. As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Then, we read the input coming from digital pin 12 using the following command:

   ```
   1      val := sComm.cmd_digital_in(1, 12);
   ```

   Note that the one leg of the pushbutton on the shield is connected to digital pin 12 of Arduino Uno as given in Fig. 5.1. The read value is displayed (or printed) by the following lines:

   ```
   1      if val == 0 then
   2        strm.print("0");
   3        sComm.delay(200);
   4      else
   5        strm.print("1");
   6        sComm.delay(200);
   7      end if;
   ```

   where `val` contains the pushbutton value acquired by the previous command. When the pushbutton is not pressed, `val` will be "0". On the other hand,

when the pushbutton is pressed, `val` will be "1". While executing this model in OpenModelica, the readers must press and release the pushbutton and observe the values being printed on the output window of OMEdit, as shown in Fig. 3.14.

2. This experiment is an extension of the previous experiment. Here, we control the state of an LED as per the status of the pushbutton. In other words, digital output to an LED is decided by the digital input received from the pushbutton. The code for this experiment is given in OpenModelica Code 5.2. After reading the pushbutton status, we turn the LED on if the pushbutton is pressed, otherwise we turn it off. The following lines,

```
1     if val == 0 then
2       strm.print("0");
3       digital_out := sComm.cmd_digital_out(1, 9, 0) "This will
      turn OFF the blue LED";
4       sComm.delay(200);
5     else
6       strm.print("1");
7       digital_out := sComm.cmd_digital_out(1, 9, 1) "This will
      turn ON the blue LED";
8       sComm.delay(200);
9     end if;
```

perform the condition check and corresponding LED state control operation. While running this experiment, the readers must press and release the pushbutton. Accordingly, they can observe whether the LED glows when the pushbutton is pressed.

5.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 5.1 Read the status of the pushbutton and display it on the output window. Available at Arduino -> SerialCommunication -> Examples -> push -> push_button_status.

```
1 model push_button_status "Checking Status of PushButton"
2   extends Modelica.Icons.Example;
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
5   Integer ok(fixed = false);
6   Integer val(fixed = false);
7   Integer c_ok(fixed = false);
```

```
8  algorithm
9    when initial() then
10     ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
       115200";
11   end when;
12   if ok <> 0 then
13     strm.print("Unable to open serial port, please check");
14   else
15     val := sComm.cmd_digital_in(1, 12);
16     if val == 0 then
17       strm.print("0");
18       sComm.delay(200);
19     else
20       strm.print("1");
21       sComm.delay(200);
22     end if;
23   end if;
24 //for i in 1:1000 loop
25 //end for;
26   when terminal() then
27     c_ok := sComm.close_serial(1) "To close the connection safely";
28   end when;
29 //sComm.cmd_arduino_meter(digital_in);
30   annotation(
31     experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 0.1));
32 end push_button_status;
```

OpenModelica Code 5.2 Turning the LED on or off depending on the pushbutton. Available at Arduino -> SerialCommunication -> Examples -> push -> led_push_button.

```
1  model led_push_button "Conrolling LED with PushButton"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer val(fixed = false);
7    Integer digital_out(fixed = false);
8    Integer c_ok(fixed = false);
9  algorithm
10   when initial() then
11     ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
       115200";
12     sComm.delay(2000);
13   end when;
14   if ok <> 0 then
15     strm.print("Unable to open serial port, please check");
16   else
```

```
17      val := sComm.cmd_digital_in(1, 12);
18      if val == 0 then
19        strm.print("0");
20        digital_out := sComm.cmd_digital_out(1, 9, 0) "This will turn OFF
        the blue LED";
21        sComm.delay(200);
22      else
23        strm.print("1");
24        digital_out := sComm.cmd_digital_out(1, 9, 1) "This will turn ON
       the blue LED";
25        sComm.delay(200);
26      end if;
27    end if;
28  //for i in 1:1000 loop
29  //end for;
30  //  strm.print(String(time));
31    when terminal() then
32      c_ok := sComm.close_serial(1) "To close the connection safely";
33    end when;
34    annotation(
35      experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
        = 0.1));
36  end led_push_button;
```

Chapter 6

Interfacing a Light Dependent Resistor

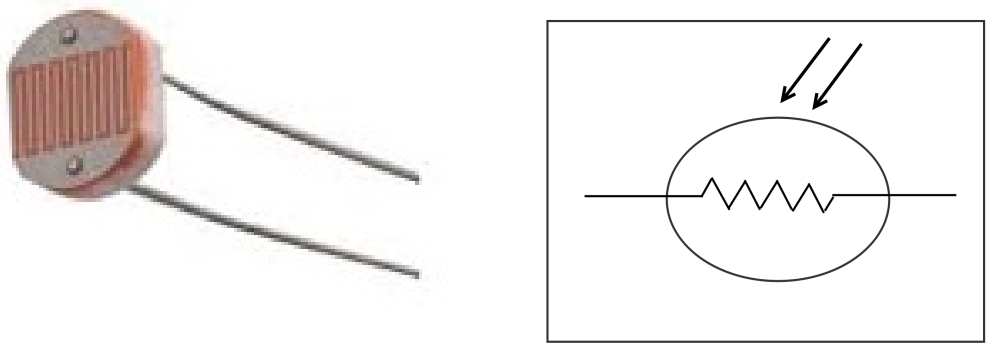
A Light Dependent Resistor (LDR) or Photoresistor is a light sensitive semiconductor device whose resistance varies with the variation in the intensity of light falling on it. As the intensity of the incident light increases, resistance offered by the LDR decreases. Typically, in dark, the resistance offered by an LDR is in the range of a few mega ohms. With the increase in light intensity, the resistance reduces to as low as a few ohms.

An LDR is widely used in camera shutter control, light intensity meters, burglar alarms, street lighting control, automatic emergency lights, etc. In this chapter we shall interface an LDR with the Arduino Uno board.

6.1 Preliminaries

A typical LDR and its symbolic representation are shown in Fig. 6.1a and Fig. 6.1b respectively. The shield provided with the kit has an LDR mounted on it. The LDR mounted on the shield looks exactly like the picture in Fig. 6.1a, although, the picture looks a lot larger. This LDR is connected to the analog pin 5 of the Arduino Uno board. The connections for this experiment are shown in Fig. 6.2. However, the user doesn't need to connect any wire or component explicitly.

The LDR mounted on the shield is an analog sensor. Hence, the analog voltage, corresponding to the changing resistance, across its terminals needs to be digitized before being sent to the computer. This is taken care of by an onboard Analog to Digital Converter (ADC) of ATmega328 microcontroller on the Arduino Uno board. ATmega328 has a 6-channel, 0 through 5, 10 bit ADC. Analog pin 5 of the Arduino Uno board, to which the LDR is connected, corresponds to channel 5 of the ADC.



(a) Pictorial representation of an LDR (b) Symbolic representation of an LDR

Figure 6.1: Light Dependent Resistor

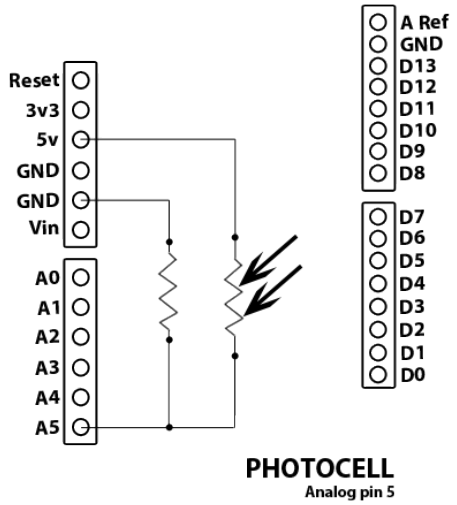


Figure 6.2: Internal connection diagram for the LDR on the shield

As there are 10 bits, 0-5V readings from LDR are mapped to the ADC values from 0 to 1023.

LDR is a commonly available sensor in the market. It costs about Rs. 100. There are multiple manufacturers which provide commercial LDRs. Some examples are VT90N1 and VT935G from EXCELITAS TECH, and N5AC501A085 and NSL19M51 from ADVANCED PHOTONIX.

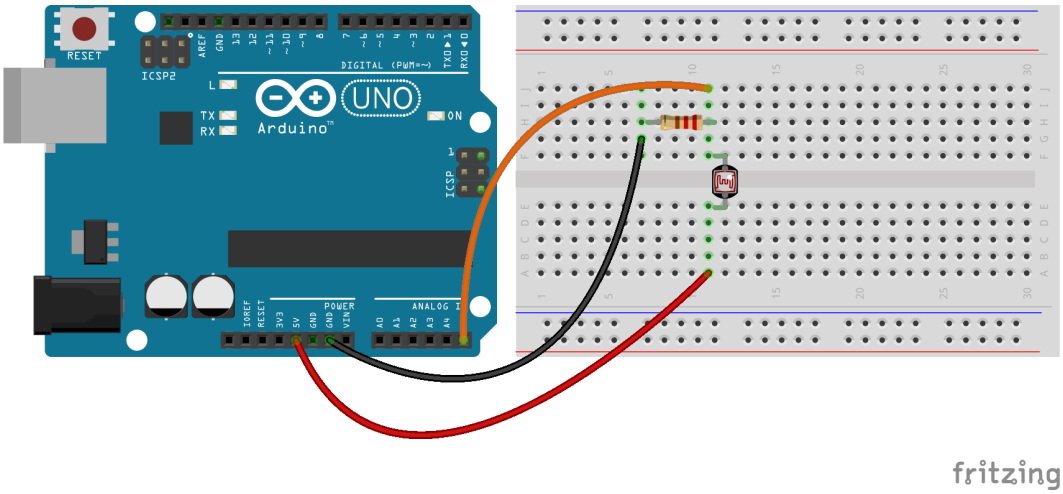


Figure 6.3: An LDR to read its values with Arduino Uno using a breadboard

6.2 Connecting an LDR with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with Tricolor LED and Push button`.

In case you have an LDR, and you want to connect it with Arduino Uno on a breadboard, please refer to Fig. 6.3. The connections given in this figure can be used to read the voltage values from an LDR connected to the analog pin 5 on Arduino Uno board. As shown in Fig. 6.3, one leg of the LDR is connected to 5V on Arduino Uno and the other leg to the analog pin 5 on Arduino Uno. A resistor is also connected to the same leg and grounded. From Fig. 6.2 and Fig. 6.3, one can infer that a resistor along with the LDR is used to create a voltage divider circuit. The varying resistance of the LDR is converted to a varying voltage. Finally, this voltage is used by the analog pin 5 of Arduino Uno in its logic.

The connections shown in Fig. 6.4 can be used to control an RGB LED, depending

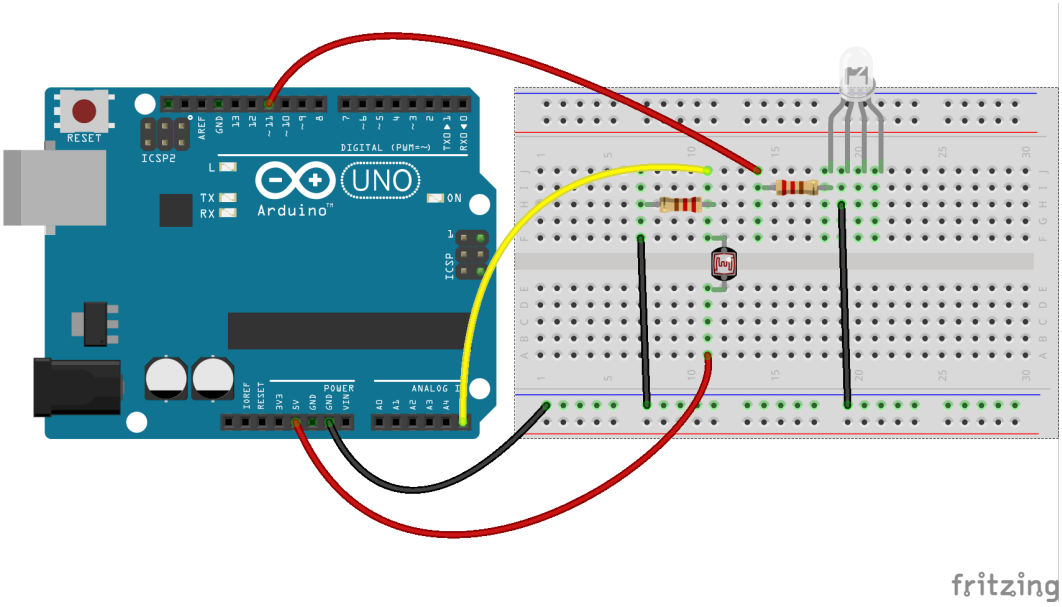


Figure 6.4: An LDR to control an LED with Arduino Uno using a breadboard

on the voltage values from the LDR. As shown in Fig. 6.4, digital pin 11 on Arduino Uno is connected to the leftmost leg of the RGB LED. Rest of the connections are same as that in Fig. 6.3.

6.3 Interfacing the LDR through the Arduino IDE

6.3.1 Interfacing the LDR

In this section, we shall describe an experiment that will help to read the voltage values from an LDR connected to the analog pin 5 of the Arduino Uno board. Later, the read values will be used to change the state of an LED. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

1. A simple code to read the LDR values is given in Arduino Code 6.1. As discussed earlier, the 0-5V LDR readings are mapped to 0-1023 through an ADC. The Arduino IDE based command for the analog read functionality is given by,

```
1    val = analogRead(A5);    // value of LDR
```

where `A5` represents the analog pin 5 to be read and the read LDR values are stored in the variable `val`. The read values are then displayed using,

```
1    Serial.println(val);   // for display
```

The delay in the code

```
1    delay(500);
```

is added so that the readings do not scroll away very fast. The entire reading and display operation is carried out 50 times.

To observe the values, one has to open the `Serial Monitor` of the Arduino IDE. The numbers displayed are in the range 0 to 1023 and depend on the light falling on the LDR. If one does the experiment in a completely dark room, the reading will be 0. If on the other hand, a bright light, say for instance the torch light from mobile, is shined, the value displayed is close to 1023. One will get intermediate values by keeping one's finger on the LDR. While running this experiment, the readers must keep their fingertips on the LDR and observe the change in values being printed on the `Serial Monitor` of Arduino IDE.

2. This experiment is an extension of the previous experiment. Here, depending the resistance of the LDR, we will turn the red LED on. The program for this is available at Arduino Code 6.2. The value of LDR is read and stored in `val`. In case it is below some threshold (like 300 in Arduino Code 6.2), it puts a high in pin number 11. From Sec. 4.1, one can note that this pin is for the red LED. If the LDR value is below 300, the red LED will be on, else, it will be turned off. While running this experiment, the readers must keep their fingertips on the LDR so that the threshold is achieved. Accordingly, they can observe whether the red LED is turned on.

Exercise 6.1 Carry out the following exercise:

1. Carry out the experiment in a dark room and check what values get displayed on the `Serial Monitor`.
2. Carry out the experiment with the torch light from the mobile phone shining on the LDR.

▮

6.3.2 Arduino Code

Arduino Code 6.1 Read and display the LDR values. Available at `Origin/user-code/ldr/arduino/ldr-read/ldr-read.ino`, see Footnote 2 on page 2.

```
int val;                    // for LDR
int i = 1;
void setup() {
Serial.begin(115200);
for(i = 1; i <= 50; i++){
    val = analogRead(A5);    // value of LDR
    Serial.println(val);  // for display
    delay(500);
  }
}
void loop() {
}
```

Arduino Code 6.2 Turning the red LED on and off. Available at `Origin/user-code/ldr/arduino/ldr-led/ldr-led.ino`, see Footnote 2 on page 2.

```
int val;
int i = 1;
void setup() {
pinMode(11, OUTPUT);            // LED Pin
Serial.begin(115200);
for(i = 1; i <= 50; i++){
    val = analogRead(A5);       // Value of LDR
    Serial.println(val);
    if(val < 300){              // Threshold
       digitalWrite(11, HIGH);
    }
    else
    {
       digitalWrite(11, LOW);
    }
    delay(500);
}
}
void loop() {
}
```

6.4 Interfacing the LDR through OpenModelica

6.4.1 Interfacing the LDR

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same two experiments, in the same order. The

shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

1. In the first experiment, we will read the LDR values. The code for this experiment is given in OpenModelica Code 6.1 . As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Then, we read the input coming from analog pin 5 using the following command:

```
val := sComm.cmd_analog_in(1, 5) "read analog pin 5 (ldr)";
```

   Note that the one leg of the LDR on the shield is connected to analog pin 5 of Arduino Uno as given in Fig. 6.2. The read value is displayed by the following command:

```
strm.print("LDR Readings: " + String(val));
```

   where `val` contains the LDR values ranging from 0 to 1023. If one does the experiment in a completely dark room, the reading will be 0. If on the other hand, a bright light, say for instance the torch light from mobile, is shined, the value displayed is close to 1023. One will get intermediate values by keeping one's finger on the LDR. While simulating this experiment, the readers must keep their fingertips on the LDR and observe the change in values being printed on on the output window of OMEdit, as shown in Fig. 3.14.

2. This experiment is an extension of the previous experiment. Here, depending the resistance of the LDR, we will turn the red LED on. The program for this is available at OpenModelica Code 6.2. The value of LDR is read and stored in `val`. In case it is below some threshold (like 300 in OpenModelica Code 6.2), it puts a high in pin number 11. From Sec. 4.1, one can note that this pin is for the red LED. If the LDR value is below 300, the red LED will be on, else, it will be turned off. While running this experiment, the readers must keep their fingertips on the LDR so that the threshold is achieved. Accordingly, they can observe whether the red LED is turned on.

6.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 6.1 Read and display the LDR values. Available at Arduino -> SerialCommunication -> Examples -> ldr -> ldr_read.

```
model ldr_read "Reading light intensity using ldr"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer val(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
    115200";
    sComm.delay(2000);
  end when;
  if ok <> 0 then
    strm.print("Unable to open serial port, please check");
  else
    val := sComm.cmd_analog_in(1, 5) "read analog pin 5 (ldr)";
    strm.print("LDR Readings: " + String(val));
    sComm.delay(500);
  end if;
  when time >= 10 then
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  annotation(
    experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
    = 1));
end ldr_read;
```

OpenModelica Code 6.2 Turning the red LED on and off. Available at Arduino -> SerialCommunication -> Examples -> ldr -> ldr_led.

```
model ldr_led "LED indicating light sensor readings"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer val(fixed = false);
  Integer digital_out(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
    115200";
    sComm.delay(2000);
  end when;
  if ok <> 0 then
```

```
15      strm.print("Unable to open serial port, please check");
16    else
17      val := sComm.cmd_analog_in(1, 5) "read analog pin 5 (ldr)";
18      strm.print("LDR Readings: " + String(val));
19      if val < 300 then
20        digital_out := sComm.cmd_digital_out(1, 11, 1) "Turn ON LED";
21      else
22        digital_out := sComm.cmd_digital_out(1, 11, 0) "Turn OFF LED";
23      end if;
24      sComm.delay(500);
25    end if;
26  //strm.print(String(time));
27    when time >= 10 then
28      c_ok := sComm.close_serial(1) "To close the connection safely";
29    end when;
30  //Setting Threshold value of 300
31    annotation(
32      experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
        = 0.2));
33  end ldr_led;
```

Chapter 7

Interfacing a Potentiometer

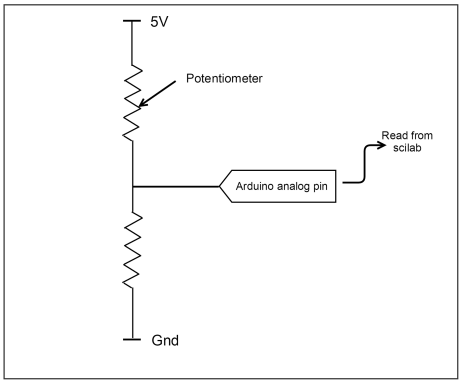
A potentiometer is a three-terminal variable resistor with two terminals connected to the two ends of a resistor and one connected to a sliding or rotating contact, termed as a wiper. The wiper can be moved to vary the resistance, and hence the potential, between the wiper and each terminal of the resistor. Thus, a potentiometer functions as a variable potential divider. It finds wide application in volume control, calibration and tuning circuits, motion control, joysticks, etc.

In this chapter, we will perform an experiment to read the analog values from a potentiometer mounted on the shield of Arduino Uno board. The analog values read from the potentiometer will then be used to control the actuation of other components.

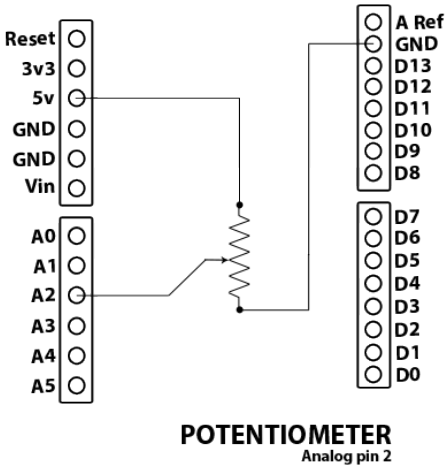
7.1 Preliminaries

The shield provided with the kit has a 1K potentiometer mounted on it. The mechanical contact at the middle terminal is rotated to vary the resistance across the middle terminal and the two ends of the potentiometer. With the fixed voltage across the two terminals of the potentiometer, the position of the wiper determines the potential across the middle terminal and either of the two end terminals. Nowadays, digital potentiometer integrated circuits, which vary resistance across two pins on the basis of the set value, are also available.

The potentiometer used in the kit can be seen on the shield in Fig. 4.3 on page 38. It is mounted on the shield. The two end terminals of the potentiometer are connected to 5V supply and ground. The middle terminal is connected to analog pin 2 of the Arduino Uno board. The resistance between the middle terminal and either of the two ends can be varied by rotating the middle terminal by hand. The connection diagram for the potentiometer is shown in Fig. 7.1.



(a) Pictorial representation of a potentiometer



(b) Internal connection diagram for the potentiometer on the shield

Figure 7.1: Potentiometer's schematic on the shield

The reading of a potentiometer is an analog voltage varying from 0 to 5V. Like LDR, we use the ADC functionality of the Arduino Uno board. Thus, we obtain digital values between 0 and 1023. In the experiment explained in this chapter, we shall also use an RGB LED mounted on the shield. An RGB LED is a tri-color LED which can illuminate in Red, Green, and Blue colors. It has 4 leads of which one lead is connected to ground and other three leads are connected to digital I/O pins 9, 10, and 11 of Arduino. In order to switch on a particular LED, we need to provide HIGH (5V) voltage to the corresponding pin of the Arduino Uno board.

7.2 Connecting a potentiometer with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch

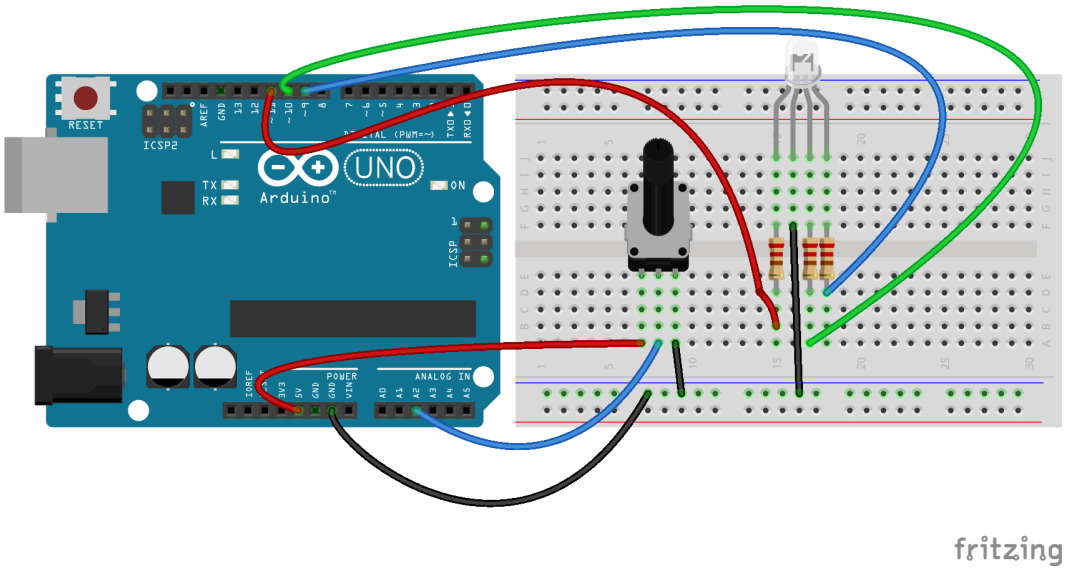


Figure 7.2: A potentiometer to control an LED with Arduino Uno using a breadboard

the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with Tricolor LED and Push button`.

In case you have a potentiometer, and you want to connect it with Arduino Uno on a breadboard, please refer to Fig. 7.2. The connections given in this figure can be used to control an RGB LED depending upon the values from the potentiometer. As shown in Fig. 7.2, the three legs of the potentiometer are connected to 5V, analog pin 2, and GND on Arduino Uno. Depending upon how much the potentiometer's shaft is rotated, one can get a value on analog pin 2. On the other hand, there is an RGB LED, and its four legs are connected to three different digital pins and GND on Arduino Uno as discussed in Chapter 4.

7.3 Reading the potentiometer from the Arduino IDE

7.3.1 Reading the potentiometer

In this section, we shall learn how to read the potentiometer input through Arduino IDE. Depending on the acquired potentiometer values, we will change the state of the RGB LED. The shield has to be attached to the Arduino Uno board before doing this experiment and the Arduino Uno needs to be connected to the computer with

a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

The Arduino code for this experiment is given in Arduino Code 7.1. In this code, lines 1 through 4 are used to assign relevant PINs to the potentiometer and RGB LED. The purpose of these lines is to avoid confusion, with the PINs, for beginners. Next, we start serial port communication, as on line 9, with the baud rate of 115200. To take the potentiometer input, we need to initialize the pins by giving the following commands:

```
1  pinMode(POT, INPUT);
2  pinMode(RGB_RED, OUTPUT);
3  pinMode(RGB_GREEN, OUTPUT);
4  pinMode(RGB_BLUE, OUTPUT);
```

where `pinMode` command is used to configure the specified pin as an input or an output pin. The first argument for the above command corresponds to the pin number and second argument corresponds to the mode of operation. In this experiment, we configure digital pin 2 as an input pin while digital pins 9, 10, and 11 as output pins. Next, we check the value of potentiometer using `analogRead` command for 10 iterations. These values range from 0 to 1023. Depending on the read value, we turn on and turn off the Red, Green or Blue LED. For example, when the position of the potentiometer corresponds to the values between 0 and 319, inclusive, we turn on the Red LED, keep it on for 1000 ms and then turn it off. This functionality is carried out by,

```
1  if(val >= 0 & val < 320) {                    //threshold 1
2    digitalWrite(RGB_RED, HIGH);
3    delay(1000);
4    digitalWrite(RGB_RED, LOW);
```

In a similar manner, we check the potentiometer values and correspondingly turn on and off the Green and Blue LEDs. Note that, we have used `if and else if` statements to check the conditions. While running this experiment, the readers must rotate the knob of the potentiometer and observe the change in the color of the RGB LED.

7.3.2 Arduino Code

Arduino Code 7.1 Turning on LEDs depending on the potentiometer threshold. Available at `Origin/user-code/pot/arduino/pot-threshold/pot-threshold.ino`, see Footnote 2 on page 2.

```
1 const int POT = 2;
2 const int RGB_RED = 11;
3 const int RGB_GREEN = 10;
```

```
4  const int RGB_BLUE = 9;
5  int val = 0;
6  int i = 0;
7  void setup() {
8    Serial.begin(115200);
9    pinMode(POT, INPUT);
10   pinMode(RGB_RED, OUTPUT);
11   pinMode(RGB_GREEN, OUTPUT);
12   pinMode(RGB_BLUE, OUTPUT);
13   for(i = 0; i < 20; i++){
14       val = analogRead(POT);
15       Serial.println(val);
16   if(val >= 0 & val < 320) {                  //threshold 1
17     digitalWrite(RGB_RED, HIGH);
18     delay(1000);
19     digitalWrite(RGB_RED, LOW);
20   }else if(val >= 320 & val < 900) {          //threshold 2
21     digitalWrite(RGB_GREEN, HIGH);
22     delay(1000);
23     digitalWrite(RGB_GREEN, LOW);
24   }else if(val >= 900 & val <= 1023) {         //threshold 3
25     digitalWrite(RGB_BLUE, HIGH);
26     delay(1000);
27     digitalWrite(RGB_BLUE, LOW);
28   }
29   }
30 }
31 void loop() {
32 }
```

7.4 Reading the potentiometer from OpenModelica

7.4.1 Reading the potentiometer

In this section, we will use a OpenModelica model to read the potentiometer values. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

Based on the acquired potentiometer values, we will change the state of the RGB LED as explained earlier. The code for this experiment is given in OpenModelica Code 7.1. As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Then, we read the analog input at pin 2 using,

```
1     val := sComm.cmd_analog_in(1, 2) "read analog pin 2";
```

where the first argument is for the serial port and the second argument corresponds to the analog pin to be read. Next, we compare the read values with the set range, and then turn on and off the corresponding LED. For example,

```
1     if val >= 0 and val < 320 then
2       digital_out := sComm.cmd_digital_out(1, 11, 1) "Turn ON LED";
3       sComm.delay(1000);
4       digital_out := sComm.cmd_digital_out(1, 11, 0) "Turn OFF LED";
```

where `cmd_digital_out` is used to set the pin 11 high (1) or low (0). We used `delay(1000)` to retain the LED in the on state for 1000 milliseconds. A similar check is done the other two bands. While running this experiment, the readers must rotate the knob of the potentiometer and observe the change in the color of the RGB LED.

7.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 7.1 Turning on LEDs depending on the potentiometer threshold. Available at Arduino -> SerialCommunication -> Examples -> pot -> pot_threshold.

```
1 model pot_threshold
2   extends Modelica.Icons.Example;
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
5   Integer ok(fixed = false);
6   Integer val(fixed = false);
7   Integer digital_out(fixed = false);
8   Integer c_ok(fixed = false);
9 algorithm
10  when initial() then
11    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
      115200";
12  end when;
13  if ok <> 0 then
14    strm.print("Unable to open serial port, please check");
15  else
16    val := sComm.cmd_analog_in(1, 2) "read analog pin 2";
17    strm.print("Potentiometer Readings: " + String(val));
18    if val >= 0 and val < 320 then
19      digital_out := sComm.cmd_digital_out(1, 11, 1) "Turn ON LED";
20      sComm.delay(1000);
21      digital_out := sComm.cmd_digital_out(1, 11, 0) "Turn OFF LED";
```

```
22      elseif val >= 320 and val < 900 then
23        digital_out := sComm.cmd_digital_out(1, 10, 1) "Turn ON LED";
24        sComm.delay(1000);
25        digital_out := sComm.cmd_digital_out(1, 10, 0) "Turn OFF LED";
26      elseif val > 900 and val <= 1023 then
27        digital_out := sComm.cmd_digital_out(1, 9, 1) "Turn ON LED";
28        sComm.delay(1000);
29        digital_out := sComm.cmd_digital_out(1, 9, 0) "Turn OFF LED";
30      end if;
31    end if;
32  //Threshold 1
33  //Threshold 2
34    when time >= 10 then
35      c_ok := sComm.close_serial(1) "To close the connection safely";
36    end when;
37    annotation(
38      experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
        = 1));
39  end pot_threshold;
```

Chapter 8

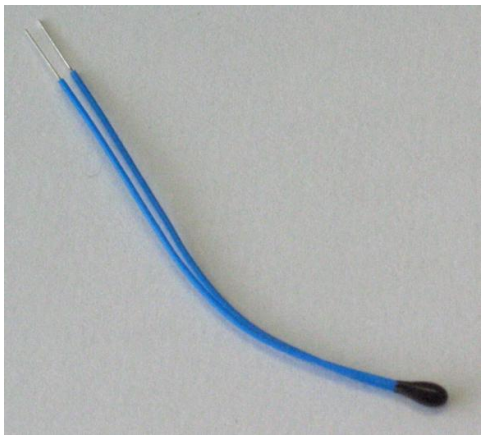
Interfacing a Thermistor

A thermistor, usually made of semiconductors or metallic oxides, is a temperature dependent/sensitive resistor. Depending on the temperature in the vicinity of the thermistor, its resistance changes. Thermistors are available in two types, NTC and PTC. NTC stands for Negative Temperature Coefficient and PTC for Positive Temperature Coefficient. In NTC thermistors, the resistance decreases with the increase in temperature and vice versa. Whereas, for PTC types, the resistance increases with an increase in temperature and vice versa. The temperature ranges, typically, from $-55°$ Celsius to $+125°$ Celsius.

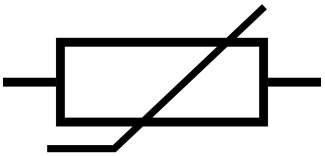
Thermistors are available in a variety of shapes such as beads, rods, flakes, and discs. Due to their compact size and low cost, they are widely used in the applications where even imprecise temperature sensing is sufficient. They, however, suffer from noise and hence need noise compensation. In this chapter we shall interface a thermistor with the Arduino Uno board.

8.1 Preliminaries

A typical thermistor and its symbolic representation are shown in 8.1a and 8.1b respectively. The thermistor is available on the shield provided with the kit. It is a bead type thermistor having a resistance of 10k at room temperature. A voltage divider network is formed using thermistor and another fixed 10k resistor. A voltage of 5 volts is applied across the series combination of the thermistor and the fixed 10k resistor. Voltage across the fixed resistor is sensed and is given to the ADC via pin 4. Hence at room temperature, both the resistors offer 10k resistance resulting in dividing the 5V equally. A buzzer is also connected on pin 3 which is a digital output pin. Connections for this experiment are shown in 8.2a and 8.2b. Nevertheless, the user doesn't need to connect any wire or component explicitly.

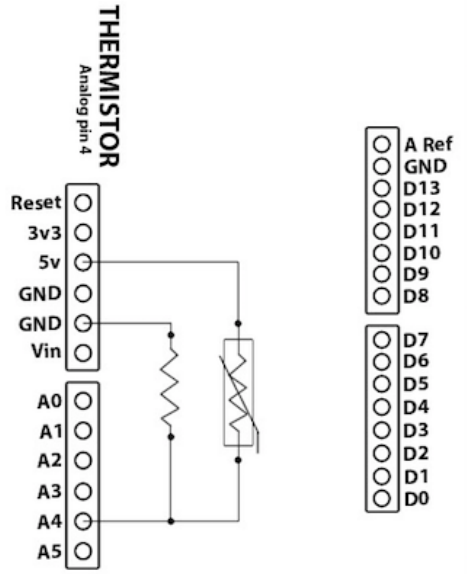


(a) Pictorial representation of a thermistor [13]

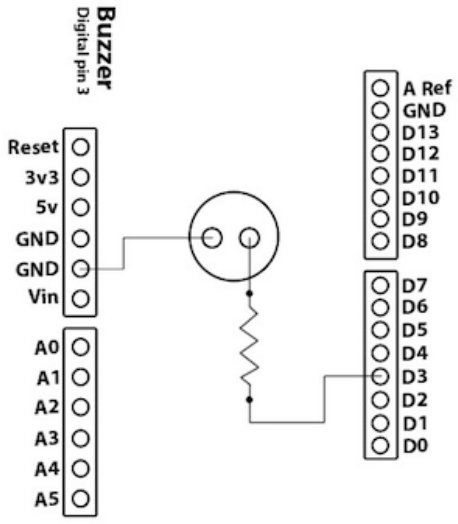


(b) Symbolic representation of a thermistor

Figure 8.1: Pictorial and symbolic representation of a thermistor



(a) Thermistor connection diagram



(b) Buzzer connection diagram

Figure 8.2: Internal connection diagrams for thermistor and buzzer on the shield

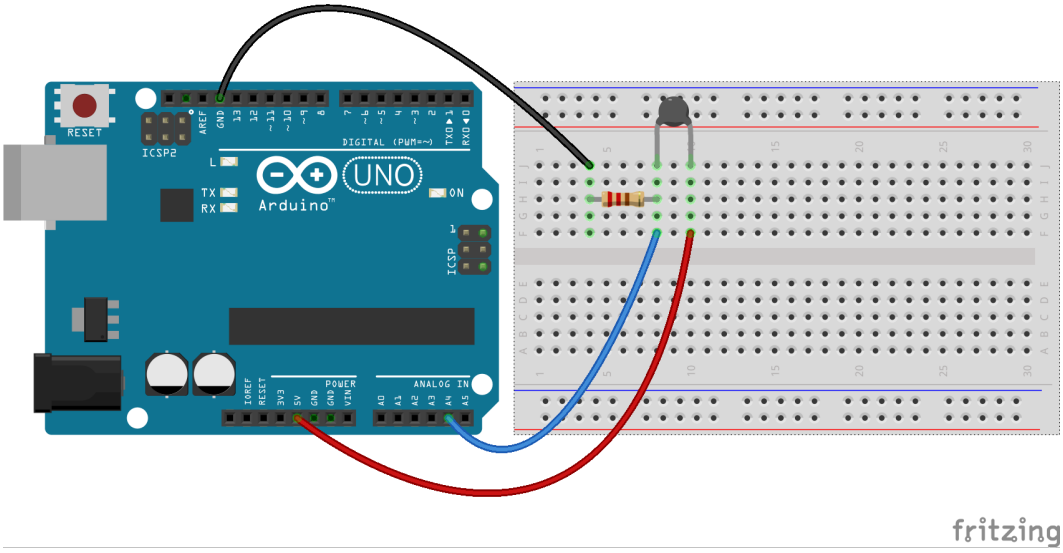


Figure 8.3: A thermistor to read its values with Arduino Uno using a breadboard

8.2 Connecting a thermistor with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with Tricolor LED and Push button`.

In case you have a thermistor, and you want to connect it with Arduino Uno on a breadboard, please refer to Fig. 8.3. The connections given in this figure can be used to read values from the thermistor connected to analog pin 4 on Arduino Uno board. As shown in Fig. 8.3, one leg of the thermistor is connected to 5V on Arduino Uno and the other leg to the analog pin 4 on Arduino Uno. A resistor is also connected to the same leg and grounded. From Fig. 8.2a and Fig. 8.3, one can infer that a resistor along with the thermistor is used to create a voltage divider circuit. The varying resistance of the thermistor is converted to a varying voltage.

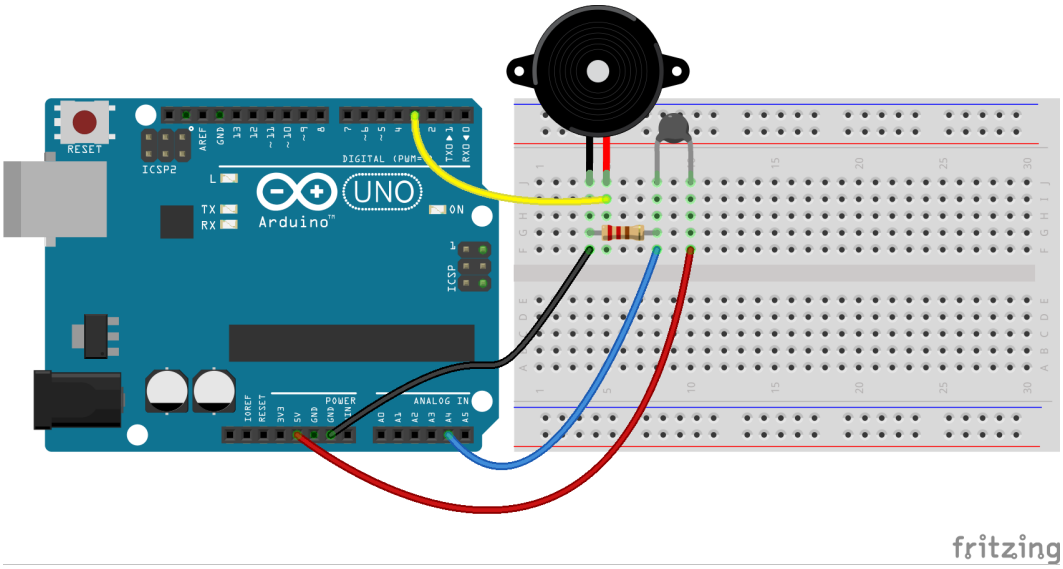


Figure 8.4: A thermistor to control a buzzer with Arduino Uno using a breadboard

Finally, this voltage is used by the analog pin 4 of Arduino Uno in its logic.

The connections shown in Fig. 8.4 can be used to control a buzzer, depending on the values from the thermistor. As shown in Fig. 8.4, digital pin 3 on Arduino Uno is connected to the one of the legs of the buzzer. Another leg of the buzzer is connected to GND of Arduino Uno.

8.3 Interfacing the thermistor from the Arduino IDE

8.3.1 Interfacing the thermistor

In this section we will learn how to read values from the thermistor connected at pin 4 of the Arduino Uno board. We shall also see how to drive a buzzer depending upon the thermistor values. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

1. A simple code to read the values from thermistor is given in Arduino Code 8.1. The Arduino IDE based command for the analog read functionality is given by.

```
1       val = analogRead(A4); //read value from thermistor
```

where `A4` represents the analog pin 4 to be read. The read value is stored in variable `val` and is displayed using

```
Serial.println(val); //display
```

The command on next line

```
delay(500);
```

is used to put a delay of 500 milliseconds. This is to avoid very fast display of the read values. The entire reading and display operation is carried out 40 times.

The values can be observed over the `Serial Monitor` of Arduino IDE. The numbers displayed range from 0 to 1023. At room temperature you may get the output of ADC around 500. If a heating or cooling source is available, one can observe the increase or decrease in the ADC output. Although the thermistor is of NTC type, the ADC output increases with increase in temperature. This is because the voltage across the fixed resistor is sensed.

While running this experiment, the readers should try holding (or rubbing) the thermistor with their fingertips. Doing so will transfer heat from the person holding the thermistor, thereby raising the temperature of the thermistor. Accordingly, they should observe the change in the thermistor values on the `Serial Monitor`.

2. In this experiment, we will turn the buzzer on depending on the temperature sensed by the thermistor. This experiment can be considered as a simple fire alarm circuit that detects fires based on a sudden change in temperature and activates the buzzer.

   The program for this is available at Arduino Code 8.2. We shall use the ADC output to carry this out. The buzzer is connected to pin 3, which is a digital output pin. The ADC output value is displayed on the serial monitor. At the same time, it is compared with a user-defined threshold, which has been set as 550 in this experiment. One may note that this threshold would vary according to the location and time of performing this experiment. Accordingly, the readers are advised to change this threshold in Arduino Code 8.2. For testing purposes, one may note the normal thermistor readings generated from the execution of Arduino Code 8.1 and set a threshold that is approximately 10 more than these readings.

   In this experiment, as soon as the ADC output exceeds 550, the buzzer is given a digital high signal, turning it on. The following lines of code perform this comparison and sending a HIGH signal to digital pin 3 on Arduino Uno:

```
1     if(val > 550)
2     {
3       digitalWrite(3, HIGH); // Turn ON buzzer
4     }
5     else
6     {
7       digitalWrite(3, LOW); // Turn OFF buzzer
8     }
```

A delay of half a second is introduced before the next value is read. While running this experiment, the readers should try holding (or rubbing) the thermistor with their fingertips. Doing so will transfer heat from the person holding the thermistor, thereby raising the temperature of the thermistor. Accordingly, they should observe whether the threshold of 550 is achieved and the buzzer is enabled.

Note: Once the thermistor value reaches 550 (the threshold), the value will remain the same (unless it is cooled). Therefore, the buzzer will continuously produce the sound, which might be a bit annoying. To get rid of this, the readers are advised to execute some other code on Arduino Uno like Arduino Code 8.1.

Exercise 8.1 Carry out the following exercise:

1. Put the thermistor in the vicinity of an Ice bowl. Take care not to wet the shield while doing so. Note down the ADC output value for 0°Celsius.

8.3.2 Arduino Code

Arduino Code 8.1 Read and display the thermistor values. Available at `Origin/user-code/thermistor/arduino/therm-read/therm-read.ino`, see Footnote 2 on page 2.

```
1 int val;
2 int i;
3
4 void setup()
5 {
6   Serial.begin(115200);
7   for(i = 1; i <= 20; i++)
8   {
```

```
9       val = analogRead(A4); //read value from thermistor
10      Serial.println(val); //display
11      delay(500);
12    }
13
14 }
15
16 void loop()
17 {
18 }
```

Arduino Code 8.2 Turning the buzzer on using the thermistor values read by ADC. Available at `Origin/user-code/thermistor/arduino/therm-buzzer/therm-buzzer.ino`, see Footnote 2 on page 2.

```
1  int val;
2  int i;
3
4  void setup()
5  {
6    pinMode(3, OUTPUT);
7    Serial.begin(115200);
8
9    for(i = 1; i <= 20; i++)
10   {
11     val = analogRead(A4); //read value from thermistor
12     Serial.println(val); //display
13
14     if(val > 550)
15     {
16       digitalWrite(3, HIGH); // Turn ON buzzer
17     }
18     else
19     {
20       digitalWrite(3, LOW); // Turn OFF buzzer
21     }
22     delay(500);
23   }
24   digitalWrite(3, LOW); // Turn OFF buzzer
25 }
26
27 void loop()
28 {
29 }
```

8.4 Interfacing the thermistor from OpenModelica

8.4.1 Interfacing the thermistor

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same two experiments, in the same order. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

1. In the first experiment, we will read the thermistor values. The code for this experiment is given in OpenModelica Code 8.1. As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Then, we read the input coming from analog pin 4 using the following command:

```
val := sComm.cmd_analog_in(1, 4) "read analog pin 5 (ldr)";
```

   Note that the one leg of the thermistor on the shield is connected to analog pin 4 of Arduino Uno as given in Fig. 8.2a. The read value is displayed by the following command:

```
strm.print("Thermistor Readings: " + String(val));
```

   where `val` contains the thermistor values ranging from 0 to 1023.

   While simulating this model, the readers should try holding (or rubbing) the thermistor with their fingertips. Doing so will transfer heat from the person holding the thermistor, thereby raising the temperature of the thermistor. Accordingly, they should observe the change in values being printed on on the output window of OMEdit, as shown in Fig. 3.14.

2. This experiment is an extension of the previous experiment. Here, we will turn a buzzer on using the thermistor values. This experiment can be considered as a simple fire alarm circuit that detects fires based on a sudden change in temperature and activates the buzzer.

   The program for this is available at OpenModelica Code 8.2. As explained earlier, the ADC maps the thermistor voltage readings in to values ranging from 0 to 1023. This means 0 for 0 volts and 1023 for 5 volts. In this experiment we compare the ADC output value with a user-defined threshold, which has been set as 550 in this experiment. One may note that this threshold would vary according to the location and time of performing this experiment. Accordingly,

the readers are advised to change this threshold in OpenModelica Code 8.2. For testing purposes, one may note the normal thermistor readings generated from the execution of OpenModelica Code 8.1 and set a threshold that is approximately 10 more than these readings.

In this experiment we compare the ADC output value with 550 and as soon as the value exceeds 550 the buzzer is turned on. The following lines of code perform this comparison and sending a HIGH signal to digital pin 3 on Arduino Uno:

```
1     if val > 550 then
2       digital_out := sComm.cmd_digital_out(1, 3, 1) "Turn ON
      Buzzer";
3     else
4       digital_out := sComm.cmd_digital_out(1, 3, 0) "Turn OFF
      Buzzer";
5     end if;
```

A delay of 500 milliseconds is introduced before the next value is read. While simulating this model, the readers should try holding (or rubbing) the thermistor with their fingertips. Doing so will transfer heat from the person holding the thermistor, thereby raising the temperature of the thermistor. Accordingly, they should observe whether the threshold of 550 is achieved and the buzzer is enabled.

Note: Once the thermistor value reaches 550 (the threshold), the value will remain the same (unless it is cooled). Therefore, the buzzer will continuously produce the sound, which might be a bit annoying. To get rid of this, the readers are advised to execute some other code on Arduino Uno like OpenModelica Code 8.1.

8.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 8.1 Read and display the thermistor values. Available at Arduino -> SerialCommunication -> Examples -> push -> therm_read.

```
1 model therm_read "Thermistor Readings"
2   extends Modelica.Icons.Example;
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
```

```
5   Integer ok(fixed = false);
6   Integer val(fixed = false);
7   Integer c_ok(fixed = false);
8 algorithm
9   when initial() then
10    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
      115200";
11    sComm.delay(2000);
12  end when;
13  if ok <> 0 then
14    strm.print("Unable to open serial port, please check");
15  else
16    val := sComm.cmd_analog_in(1, 4) "read analog pin 5 (ldr)";
17    strm.print("Thermistor Readings: " + String(val));
18    sComm.delay(500);
19  end if;
20  when terminal() then
21    c_ok := sComm.close_serial(1) "To close the connection safely";
22  end when;
23  annotation(
24    experiment(StartTime = 0, StopTime = 20, Tolerance = 1e-6, Interval
      = 1));
25 end therm_read;
```

OpenModelica Code 8.2 Turning the buzzer on using the thermistor values read by ADC. Available at Arduino -> SerialCommunication -> Examples -> push -> therm_buzzer.

```
1 model therm_buzzer "Sound buzzer depending on thermistor readings"
2   extends Modelica.Icons.Example;
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
5   Integer ok(fixed = false);
6   Integer val(fixed = false);
7   Integer digital_out(fixed = false);
8   Integer c_ok(fixed = false);
9 algorithm
10  when initial() then
11    ok := sComm.open_serial(1, 2, 115200) "At port 2 with baudrate of
      115200";
12    sComm.delay(2000);
13  end when;
14  if ok <> 0 then
15    strm.print("Unable to open serial port, please check");
16  else
17    val := sComm.cmd_analog_in(1, 4) "read analog pin 4";
18    strm.print("Thermistor Readings: " + String(val));
19    if val > 550 then
20      digital_out := sComm.cmd_digital_out(1, 3, 1) "Turn ON Buzzer";
```

```
21      else
22        digital_out := sComm.cmd_digital_out(1, 3, 0) "Turn OFF Buzzer";
23      end if;
24      sComm.delay(500);
25    end if;
26    digital_out := sComm.cmd_digital_out(1, 3, 0) "Turn OFF Buzzer";
27  //for i in 1:500 loop
28  //end for;
29  //Run for 500 iterations
30  //Setting Threshold value of 500
31    when terminal() then
32      c_ok := sComm.close_serial(1) "To close the connection safely";
33    end when;
34    annotation(
35      experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
        = 0.1));
36  end therm_buzzer;
```

Chapter 9

Interfacing a Servomotor

A servomotor is a very useful industrial control mechanism. Learning to control it will be extremely useful for practitioners. In this chapter, we will explain how to control a servomotor using the Arduino Uno board. We will begin with preliminaries of servomotors and explain how to connect a typical servomotor to the Arduino Uno board and shield. We will then explain how to control it through the Arduino IDE and other open-source software tools. We will provide code for all the experiments.

9.1 Preliminaries

A servomotor is a rotary control mechanism. It can be commanded to rotate to a specified angle. It can rotate in positive or negative direction. Using servomotors, one can control angular position, velocity and acceleration. Servomotors are useful in many applications. Some examples are robotics, industrial motors and printers.

Typical servomotors have a maximum range of 180°, although some have different ranges[3]. Servomotors typically have a position sensor, using which, rotate to the commanded angle. The minimum angle to which a servomotor can be rotated is its least count, which varies from one model to another. Low cost servomotors have a large least count, say, of the order of 10°.

A servomotor typically comes with three terminals for the following three signals: position signal, Vcc and ground. Position signal means that this terminal should be connected to one of the PWM (Pulse Width Modulation) pins [14] on Arduino Uno. This book uses PWM pin 5 for this purpose. Rest two terminals (Vcc and ground) need to be connected to 5V and GND on Arduino Uno. Table 9.1 summarizes these connections.

[3] All the angles in a servomotor are absolute angles, with respect to a fixed reference point, which can be taken as 0°.

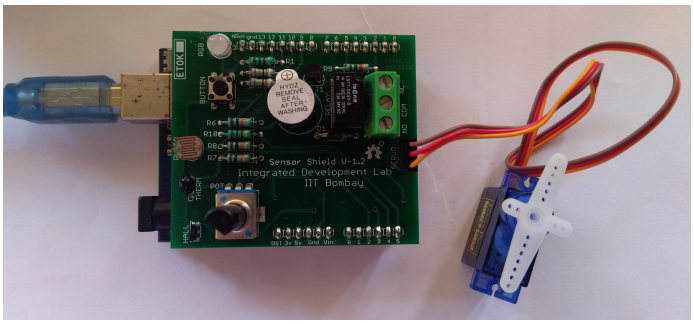


Figure 9.1: Connecting servomotor to the shield attached on Arduino Uno

Table 9.1: Connecting a typical servomotor to Arduino Uno board

Servomotor terminal	Arduino board
Position signal (orange or yellow)	5
Vcc (red or orange)	5V
Ground (black or brown)	GND

We now explain how to connect a typical servomotor to the shield attached on the Arduino Uno board. On the shield, there is a three-pin header at one of the ends. The pins of this header have been marked as 1, 2, and 3. These pins: 1, 2, and 3 are internally connected to 5V, PWM pin 5, and GND on Arduino Uno respectively. As discussed before, a typical servomotor has three terminals. Thus, the readers need to connect these three terminals with the three-pin header, as shown in Fig. 9.1 before running the experiments given in this chapter.

9.2 Connecting a servomotor with Arduino Uno using a breadboard

This section is useful for those who either don't have a shield or don't want to use the shield for performing the experiments given in this chapter.

A breadboard is a device for holding the components of a circuit and connecting them together. We can build an electronic circuit on a breadboard without doing any soldering. To know more about the breadboard and other electronic components, one should watch the Spoken Tutorials on Arduino as published on `https://spoken-tutorial.org/`. Ideally, one should go through all the tutorials labeled as Basic. However, we strongly recommend the readers should watch the fifth and sixth tutorials, i.e., `First Arduino Program` and `Arduino with`

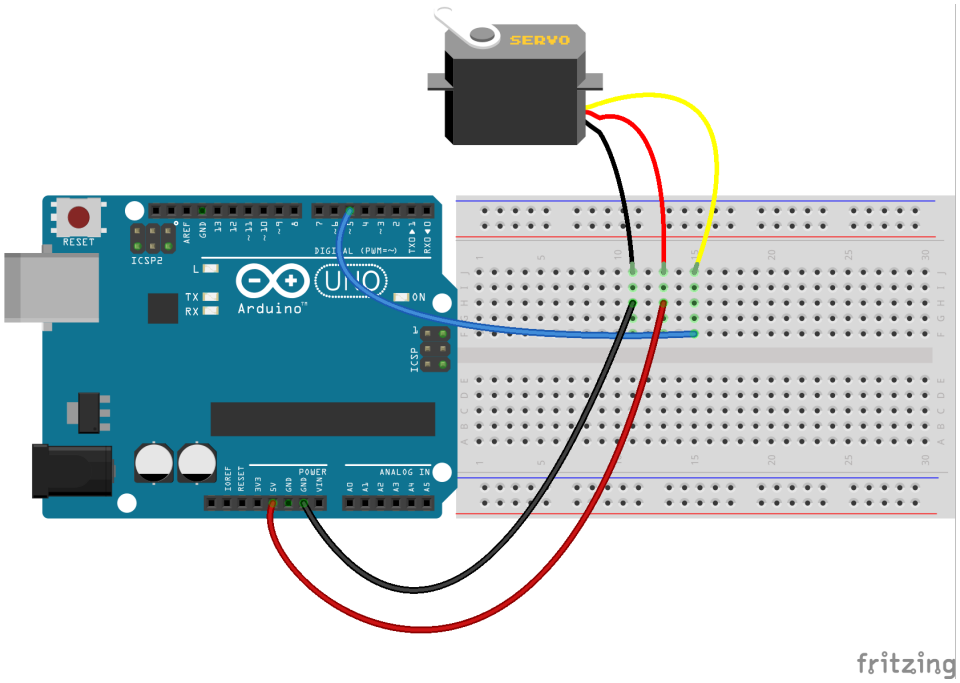


Figure 9.2: A servomotor with Arduino Uno using a breadboard

`Tricolor LED and Push button`.

In case you have a servomotor and want to connect it with Arduino Uno on a breadboard, please refer to Fig. 9.2. As shown in Fig. 9.2, there is a servomotor with three terminals. These terminals are used for the same three signals, as that explained in Sec. 9.1. The connections shown in Fig. 9.3 can be used to control the position of the servomotor, depending on the values coming from a potentiometer. As shown in Fig. 9.3, analog pin 2 on Arduino Uno is connected to the middle leg of the potentiometer. Rest of the connections are same as that in Fig. 9.2.

9.3 Controlling the servomotor through the Arduino IDE

9.3.1 Controlling the servomotor

In this section, we will describe some experiments that will help rotate the servomotor based on the command given from Arduino IDE. We will also give the necessary code. We will present four experiments in this section. The shield has to be attached to the Arduino Uno board before doing these experiments and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader

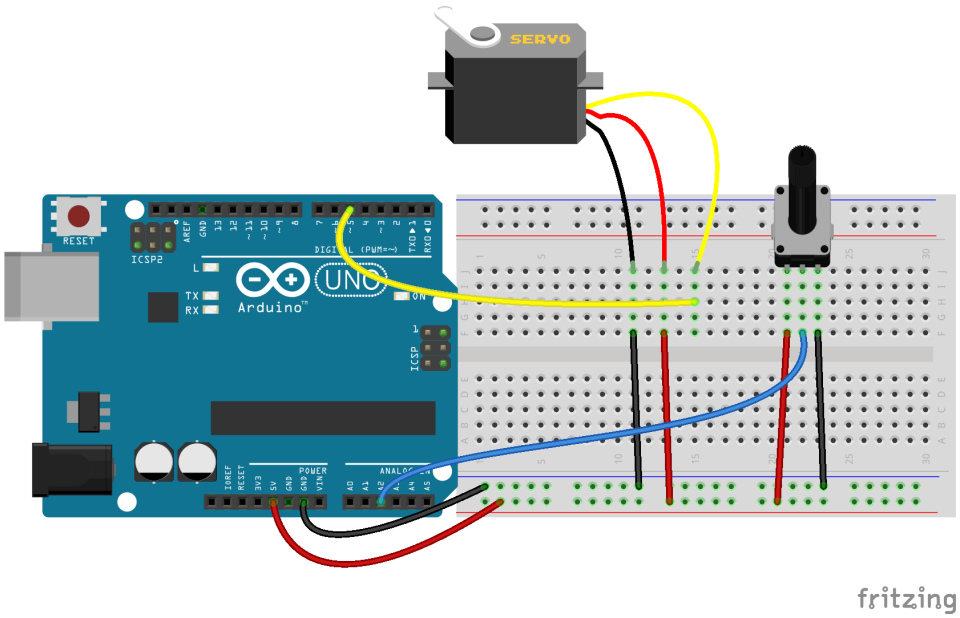


Figure 9.3: A servomotor and a potentiometer with Arduino Uno using a breadboard

should go through the instructions given in Sec. 3.1 before getting started.

1. In the first experiment, we will move the servomotor by 30°. Arduino Code 9.1 has the required code for this. This code makes use of a library named `Servo` [15]. Thus, we include its header file at the top of Arduino Code 9.1:

```
1 #include <Servo.h>
```

   Next, we create a `Servo` object and call it `myservo`, as shown below:

```
1 Servo myservo;  // create servo object to control a servo
```

   Most Arduino boards allow the creation of 12 servo objects. Next, we initialize the port for serial communication at data rate of 115200 bits per second. Following to this, we mention the pin to which the servo is attached, as shown below:

```
1   myservo.attach(5); // attach the servo object on to pin 5
```

   With this, we issue the command to rotate the servomotor by 30° followed by a delay of 1000 milliseconds:

```
1   myservo.write(30); // tell servo to rotate by 30 degrees
2   delay(1000);
```

At last, we detach the servomotor.

Once this code is executed, the servomotor would move by 30°, as commanded. What happens if this code is executed once again? The motor will not move at all. What is the reason? Recall that what we assign to the motor are absolute positions, with respect to a fixed origin. As a result, there will be no change at all.

2. In the second experiment, we move the motor by 90° in the forward direction and 45° in the reverse direction. This code is given in Arduino Code 9.2. In this code, we have added a delay of 1000 milliseconds between the two instances of rotating the servomotor:

```
1  myservo.write(90); // tell servo to rotate by 90 degrees
2  delay(1000);
3  myservo.write(45);
```

What is the reason behind this delay? If the delay were not there, the motor will move only by the net angle of $90 - 45 = 45$ degrees. The reader should verify this by commenting on the delay command.

3. In the third experiment, we move the motor in increments of 20°. This is achieved by the `for` loop, as in Arduino Code 9.3. Both `i`, the loop variable and `angle`, the variable to store angle, are declared as `int` in this code. The code helps the motor move in steps of 20° all the way to 180°.

4. Finally, in the last experiment, we read the potentiometer value from the shield and use it to drive the servomotor, see Arduino Code 9.4. The resistance of the potentiometer is represented in 10 bits. As a result, the resistance value could be any one of 1024 values, from 0 to 1023. This entire range is mapped to 180°, as shown below:

```
1  val = analogRead(potpin); // reads a value in (0,1023) through
     pot
2  val = map(val, 0, 1023, 0, 180); // maps it in the range (0,180)
      degrees
3  myservo.write(val);          // moves the motor to the mapped
     degree
```

By rotating the potentiometer, one can make the motor move by different amounts.

As mentioned in Chapter 7, the potentiometer on the shield is connected to analog pin 2 on Arduino Uno. Through this pin, the resistance of the potentiometer, in the range of 0 to 1023, depending on its position, is read. Thus, by rotating the potentiometer, we make different values appear on pin 2. This

value is used to move the servo. For example, if the resistance is half of the total, the servomotor will go to 90° and so on. The servomotor stops for 500 milliseconds after every move. The loop is executed for 50 iterations. During this period, the servomotor keeps moving as dictated by the resistance of the potentiometer. While running this experiment, the readers must rotate the knob of the potentiometer and observe the change in the position (or angle) of the servomotor.

Exercise 9.1 Let us carry out this exercise:

1. In Arduino Code 9.3, the loop parameter `i` starts from 1. From what angle will the motor start? If one wants the motor to start from 0°, what should one do?
2. How does one find the least count of the servomotor? If the variable `angle` is chosen to be less than this least count in Arduino Code 9.3, what happens?
3. What happens if 180 in Line 10 of Arduino Code 9.4 is changed to 90? What does the change 180 to 90 mean?

▮

9.3.2 Arduino Code

Arduino Code 9.1 Rotating the servomotor to a specified degree. Available at `Origin/user-code/servo/arduino/servo-init/servo-init.ino`, see Footnote 2 on page 2.

```
1 #include <Servo.h>
2 Servo myservo;  // create servo object to control a servo
3 void setup() {
4   Serial.begin(115200);
5   myservo.attach(5); // attach the servo object on to pin 5
6   myservo.write(30); // tell servo to rotate by 30 degrees
7   delay(1000);
8   myservo.detach();
9 }
10 void loop() {
11 }
```

Arduino Code 9.2 Rotating the servomotor to a specified degree and reversing. Available at `Origin/user-code/servo/arduino/servo-reverse/servo-reverse.ino`, see Footnote 2 on page 2.

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
void setup() {
  Serial.begin(115200);
  myservo.attach(5); // attach the servo object on to pin 5
  myservo.write(90); // tell servo to rotate by 90 degrees
  delay(1000);
  myservo.write(45);
  delay(1000);
  myservo.detach();
}
void loop() {
}
```

Arduino Code 9.3 Rotating the servomotor in increments. Available at `Origin/user-code/servo/arduino/servo-loop/servo-loop.ino`, see Footnote 2 on page 2.

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
int angle = 20;
int i = 0;
void setup() {
  for(i = 1; i < 10; i++) {
    Serial.begin(115200);
    myservo.attach(5);       // attach the servo object on to pin 5
    myservo.write(angle*i); // tell servo to rotate by 20 degrees
    delay(1000);             // waits for a sec
  }
  myservo.detach();
}
void loop() {
}
```

Arduino Code 9.4 Rotating the servomotor through the potentiometer. Available at `Origin/user-code/servo/arduino/servo-pot/servo-pot.ino`, see Footnote 2 on page 2.

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
int potpin = 2; // analog pin used to connect the potentiometer
int val;        // variable to read the value from the analog pin
int i;
void setup(){
  Serial.begin(115200);
  myservo.attach(5);        // attach the servo object on to pin 5
  for(i = 0; i < 50; ++i){
  val = analogRead(potpin); // reads a value in (0,1023) through pot
```

```
11   val = map(val, 0, 1023, 0, 180); // maps it in the range (0,180)
       degrees
12   myservo.write(val);          // moves the motor to the mapped degree
13   delay(500);                  // waits for a second for servo to reach
14   }
15   myservo.detach();
16 }
17 void loop(){
18 }
```

9.4 Controlling the servomotor through OpenModelica

9.4.1 Controlling the servomotor

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same four experiments, in the same order. As mentioned earlier, the shield must be removed from the Arduino Uno and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

1. The first experiment makes the servomotor move by 30°. The code for this experiment is given in OpenModelica Code 9.1. As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Next, we attach the servomotor by issuing the command given below:

   ```
   1       sComm.cmd_servo_attach(1, 1) "To attach the motor to pin 5
         of servo1";
   ```

   As shown above, the servomotor is attached on board 1 (the first argument) to pin 1 (the second argument). In the OpenModelica-Arduino toolbox discussed in Sec. 3.2.4, pin 1 and pin 5 are connected. As a result, we connect the wire physically to pin 5, which is achieved by the shield as discussed in Sec. 9.1.

   With this, we issue the command to move the servomotor by 30° followed by a delay of 1 second:

   ```
   1       sComm.cmd_servo_move(1, 1, 30) "tell servo to rotate by 30
         degrees";
   ```

   At last, we detach the servomotor followed by closing the serial port.

   Once this code is executed, the servomotor would move by 30°, as commanded. What happens if this code is executed once again? The motor will not move at

all. What is the reason? Recall that what we assign to the motor are absolute positions, with respect to a fixed origin. As a result, there will be no change at all.

2. In the second experiment, we move the servomotor by 90° in the forward direction and 45° in the reverse direction. This code is given in OpenModelica Code 9.2. As mentioned earlier, the angles are absolute with respect to a fixed reference point and not relative. In this code, we have added a delay of 1000 milliseconds between the two instances of moving the servomotor:

```
1       sComm.cmd_servo_move(1, 1, 90) "Move the servo to 90 degree"
    ;
2       sComm.delay(1000) "be there for one second";
3       sComm.cmd_servo_move(1, 1, 45) "Move the servo to 45 degree"
    ;
```

   What is the reason behind this delay? If the delay were not there, the motor will move only by the net angle of $90 - 45 = 45$ degrees. The reader should verify this by commenting on the delay command.

3. In the third experiment, we move the motor in increments of 20°. This is achieved by the `for` loop, as in OpenModelica Code 9.3. The code helps the motor move in steps of 20° all the way to 180°.

4. Finally, in the last experiment, we read the potentiometer value from the shield and use it to drive the servomotor, see OpenModelica Code 9.4. The resistance of the potentiometer is represented in 10 bits. As a result, the resistance value could be any one of 1024 values, from 0 to 1023. This entire range is mapped to 180°, as shown below:

```
1         val := sComm.cmd_analog_in(1, 2) "Read potentiometer value
    ";
2         val := integer(val * 180 / 1023);
3        sComm.cmd_servo_move(1, 1, val) "Command the servo motor";
```

   By rotating the potentiometer, one can make the motor move by different amounts.

   As mentioned in Chapter 7, the potentiometer on the shield is connected to analog pin 2 on Arduino Uno. Through this pin, the resistance of the potentiometer, in the range of 0 to 1023, depending on its position, is read. Thus, by rotating the potentiometer, we make different values appear on pin 2. This value is used to move the servo. For example, if the resistance is half of the total, the servomotor will go to 90° and so on. The servomotor stops for 500 milliseconds after every move. The loop is executed for 50 iterations. During

this period, the servomotor keeps moving as dictated by the resistance of the potentiometer. While running this experiment, the readers must rotate the knob of the potentiometer by a fixed amount and observe the change in the position (or angle) of the servomotor.

9.4.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 9.1 Rotating the servomotor to a specified degree. Available at Arduino -> SerialCommunication -> Examples -> servo -> servo_init.

```
model servo_init "Rotate Servo Motor "
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
    is 115200";
    if ok <> 0 then
      strm.print("Check the serial port and try again");
    else
      sComm.cmd_servo_attach(1, 1) "To attach the motor to pin 5 of
    servo1";
      sComm.cmd_servo_move(1, 1, 30) "tell servo to rotate by 30
    degrees";
      sComm.delay(1000);
    end if;
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  sComm.cmd_servo_detach(1, 1);
  annotation(
    experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
    = 5));
end servo_init;
```

OpenModelica Code 9.2 Rotating the servomotor to a specified degree and reversing. Available at Arduino -> SerialCommunication -> Examples -> servo -> servo_reverse.

```
model servo_reverse
  extends Modelica.Icons.Example;
```

```
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
5   Integer ok(fixed = false);
6   Integer c_ok(fixed = false);
7 algorithm
8   when initial() then
9     ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
      is 115200";
10    sComm.delay(2000);
11    if ok <> 0 then
12      strm.print("Check the serial port and try again");
13    else
14      sComm.cmd_servo_attach(1, 1) "Attach the motor to pin 5. 1 means
      5";
15      sComm.cmd_servo_move(1, 1, 90) "Move the servo to 90 degree";
16      sComm.delay(1000) "be there for one second";
17      sComm.cmd_servo_move(1, 1, 45) "Move the servo to 45 degree";
18      sComm.delay(1000) "be there for one second";
19      sComm.cmd_servo_detach(1, 1) "Detach the motor";
20    end if;
21    c_ok := sComm.close_serial(1) "To close the connection safely";
22  end when;
23  annotation(
24    experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
      = 5));
25 end servo_reverse;
```

OpenModelica Code 9.3 Rotating the servomotor in steps of 20°. Available at Arduino -> SerialCommunication -> Examples -> servo -> servo_loop.

```
1 model servo_loop "Rotate servo motor by 20 degrees 10 times"
2   extends Modelica.Icons.Example;
3   import sComm = Arduino.SerialCommunication.Functions;
4   import strm = Modelica.Utilities.Streams;
5   Integer ok(fixed = false);
6   Integer c_ok(fixed = false);
7   Integer angle(fixed = true);
8 algorithm
9   when initial() then
10    ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
      is 115200";
11    if ok <> 0 then
12      strm.print("Check the serial port and try again");
13    else
14      sComm.cmd_servo_attach(1, 1) "Attach motor to pin 5. 1 means pin
      5.";
15      sComm.delay(2000);
16      angle := 20 "Angle by which it has to move";
17      for i in 1:10 loop
```

```
18          sComm.cmd_servo_move(1, 1, angle * i) "tell servo to rotate by
      20 degrees";
19          sComm.delay(1000) "waits for a sec";
20        end for;
21        sComm.cmd_servo_detach(1, 1) "Detach the motor";
22      end if;
23      c_ok := sComm.close_serial(1) "To close the connection safely";
24    end when;
25    annotation(
26      experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
      = 5));
27 end servo_loop;
```

OpenModelica Code 9.4 Rotating the servomotor to a degree specified by the potentiometer. Available at Arduino -> SerialCommunication -> Examples -> servo -> servo_pot.

```
1  model servo_pot "Control Servo Motor using Potentiometer"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer c_ok(fixed = false);
7    Integer val(fixed = false);
8  algorithm
9    when initial() then
10     ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
       is 115200";
11     if ok <> 0 then
12       strm.print("Check the serial port and try again");
13     else
14       sComm.cmd_servo_attach(1, 1) "Attach the motor to pin 5";
15       sComm.delay(2000);
16       for i in 1:50 loop
17         val := sComm.cmd_analog_in(1, 2) "Read potentiometer value";
18         val := integer(val * 180 / 1023);
19         sComm.cmd_servo_move(1, 1, val) "Command the servo motor";
20         sComm.delay(500) "sleep for 1000 milliseconds";
21       end for;
22       sComm.cmd_servo_detach(1, 1) "Detach the motor";
23     end if;
24     c_ok := sComm.close_serial(1) "To close the connection safely";
25   end when;
26 //         strm.print(String(integer(analog_in * 180 / 1023)));
27 //     analog_in := sComm.math_floor(analog_in * (180 / 1023)) "Scale
       Potentiometer value to 0-180";
28 //strm.print(String(analog_in));
29   annotation(
```

```
30      experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
        = 5));
31 end servo_pot;
```

Chapter 10

Interfacing a DC Motor

Motors are widely used in commercial applications. DC motor converts electric power obtained from direct current to mechanical motion. This chapter describes the experiments to control DC motor with Arduino Uno board. We will observe the direction of motion of the DC motor being changed using the microcontroller on Arduino Uno board. Control instruction will be sent to Arduino Uno using the Arduino IDE and other open-source software tools. The experiments provided in this chapter don't require the shield. Therefore, the readers must remove the shield from the Arduino Uno before moving further in this chapter. Before removing the shield, the readers are advised to detach Arduino Uno from the computer.

10.1 Preliminaries

In order to change its direction, the sign of the voltage applied to the DC motor is changed. For that, one needs to use external hardware called H-Bridge circuit DC motor with Arduino Uno. H-Bridge allows direction of the current passing through the DC motor to be changed. It avoids the sudden short that may happen while changing the direction of current passing through the motor. It is one of the essential circuits for the smooth operation of a DC motor. There are many manufacturers of H-bridge circuit viz. L293D, L298, etc. Often they provide small PCB breakout boards. These modules also provide an extra supply that is needed to drive the DC motor. Fig. 10.1 shows the diagram of a typical breakout board containing IC L293D, which will be used in this book. One may note that the toolboxes presented in this book supports three types of H-Bridge circuit. Table 10.1 provides the values to be passed for different H-Bridge circuits.

Input from Arduino Uno to H-bridge IC is in pulse width modulation (PWM) form. PWM is a technique to generate analog voltages using digital pins. We know

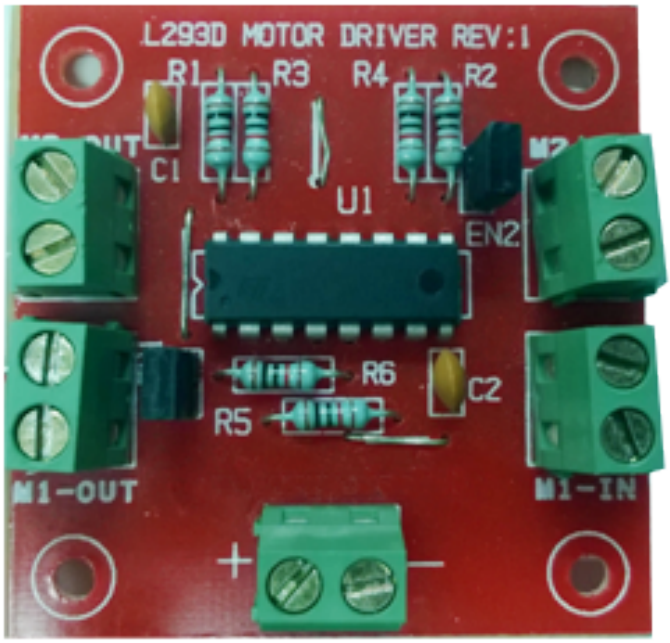


Figure 10.1: L293D motor driver board

Table 10.1: Values to be passed for different H-Bridge circuits

Type of H-Bridge circuit	Value
MotorShield Rev3	1
PMODHB5/L298	2
L293D	3

that Arduino Uno has digital input-output pins. When these pins are configured as an output, they provide High (5V) or Low (0V) voltage. With PWM technique, these pins are switched on and off iteratively and fast enough so that the voltage is averaged out to some analog value in between 0-5V. This analog value depends on "switch-on" time and "switch-off" time. For example, if both "switch-on" time and "switch-off" time are equal, average voltage on PWM pin will be 2.5V. To enable fast switching of digital pin, a special hardware is provided in microcontrollers. PWM is considered as an important resource of the microcontroller system. Arduino Uno board has 6 PWM pins (3, 5, 6, 9, 10, 11) [14]. On an original Arduino Uno board, these pins are marked with a tilde sign next to the pin number, as shown in Fig. 10.2. For each of these pins, the input can come from 8 bits. Thus we can generate $2^8 = 256$ different analog values (from 0 to 255) in between 0-5V with these pins.

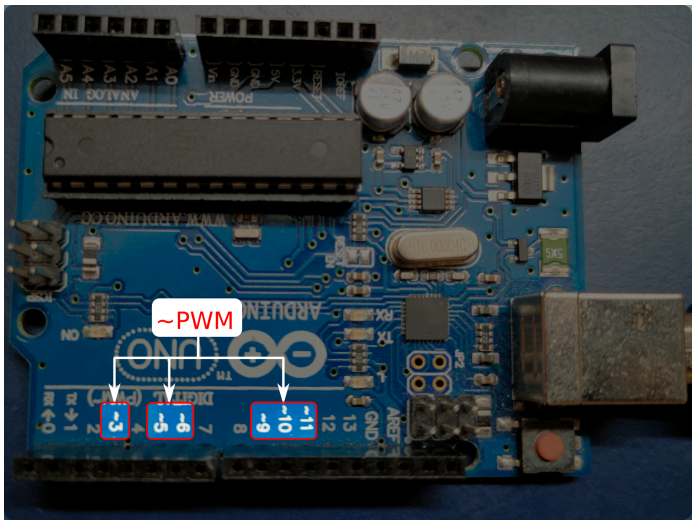


Figure 10.2: PWM pins on an Arduino Uno board

We now carry out the following connections:

1. Connect input of L293D (M1_IN) pins to two of the PWM pins available on Arduino Uno. We have used pins 9 and 10 of the Arduino Uno board.
2. Connect the output of the L293D (M1_OUT) pins directly to the 2 wires of the DC motor. As the direction is changed during the operation, the polarity of the connection does not matter.
3. Connect supply (Vcc) and ground (Gnd) pins of L293D to 5V and Gnd pins of the Arduino Uno board, respectively.

A schematic of these connections is given in Fig. 10.3. The actual connections can be seen in Fig. 10.4.

10.2 Controlling the DC motor from Arduino

10.2.1 Controlling the DC motor

In this section, we will describe some experiments that will help drive the DC motor from the Arduino IDE. We will also give the necessary code. We will present three experiments in this section. As mentioned earlier, the shield must be removed from the Arduino Uno and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.1 before getting started.

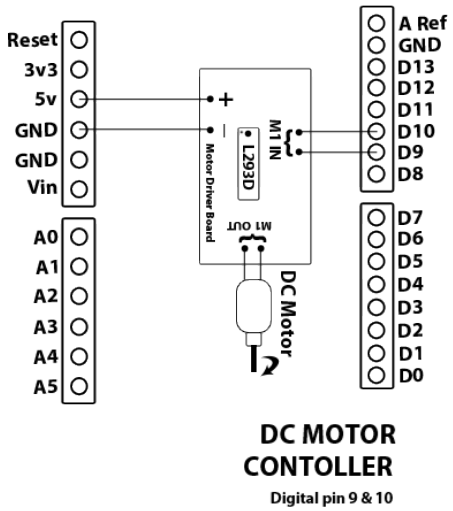


Figure 10.3: A schematic of DC motor connections

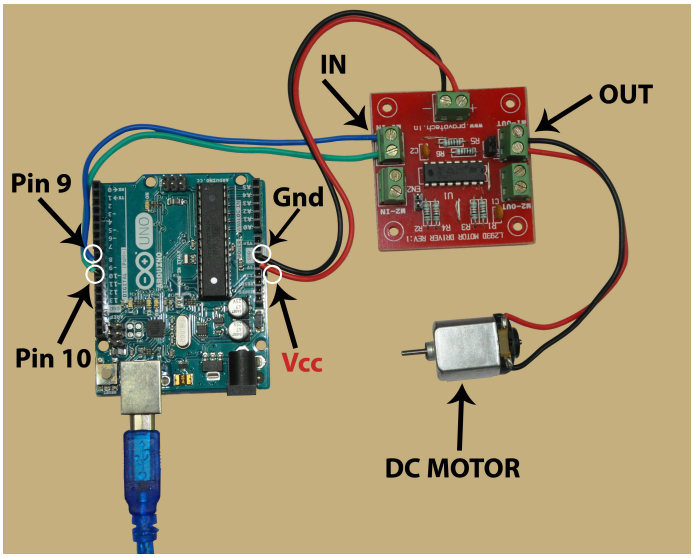


Figure 10.4: How to connect the DC motor to the Arduino Uno board

Note: The readers are advised to affix a small (very lightweight) piece of paper at the tip of the shaft of the DC motor. That will help them observe the direction of rotation of the DC motor while running the experiments.

1. We now demonstrate how to drive the DC motor from the Arduino IDE. Arduino Code 10.1 has the required code for this. It starts the serial port at a baud rate of 115200. Pins 9 and 10 are declared as output pins and hence values can be written on to them. The following lines are used to declare these two pins as output pins:

```
pinMode(9, OUTPUT);   // use pins 9 and 10 for motor output
pinMode(10, OUTPUT);
```

   Next, we write PWM 100 on pin 9 and PWM 0 on pin 10, as shown below:

```
analogWrite(9, 100); // PWM 100 on pin 9 makes the motor rotate
analogWrite(10, 0);
```

   Recall from Fig. 10.4 that pins 9 and 10 are connected to the input of the breakout board, which in turn makes the DC motor run at an intermediate speed. Some of the breakout boards may not have enough current driving capability and hence tend to heat up. To avoid these difficulties, the DC motor is run at an intermediate value of PWM 100. Remember, we can increase this value upto 255.

   The line containing `delay` makes the previous command execute for 3 seconds. As a result, the DC motor continues to rotate for 3 seconds. After this, we put a 0 in both pins 9 and 10, as shown below:

```
analogWrite(9, 0);    // 0 on pin 9 stops the motor
analogWrite(10, 0);
```

   With this, the motor comes to a halt.

2. It is easy to make the DC motor run in the reverse direction by interchanging the values put on pins 9 and 10. This is done in Arduino Code 10.2. In this code, we make the DC motor run in one direction for 3 seconds and then make it rotate in the reverse direction for 2 seconds. The rotation in reverse direction is achieved by putting 100 in pin 10, as shown below:

```
analogWrite(9, 0);      //
analogWrite(10, 100);   // Motor runs in the reverse direction for
```

   Next, we release the motor by writing 0 in both pins 9 and 10, as shown below:

```
analogWrite(9, 0);      // Motor is stopped
analogWrite(10, 0);     //
```

   With this, the motor comes to a halt.

3. Next, we make the DC motor run in forward and reverse directions, in a loop. This is done through Arduino Code 10.3. We first write PWM 100 in pin 9 for 3 seconds. After that, make the motor stop for 2 seconds. Finally, make the motor rotate in the reverse direction by writing PWM 100 in pin 10 for two seconds. Finally, we make the motor stop for one second. The entire thing is put in a `for` loop which runs for 5 iterations.

Exercise 10.1 Carry out the following exercise:

1. Try out some of the suggestions given above, *i.e.*, removing certain numbers from the code
2. See if the DC motor runs if you put 1 instead of 100 as the PWM value. Explain why it does not run. Find out the smallest value at which it will start running.

▮

10.2.2 Arduino Code

Arduino Code 10.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/arduino/dcmotor-clock/dcmotor-clock.ino`, see Footnote 2 on page 2.

```
1 void setup() {
2 Serial.begin(115200); // set the baudrate
3 pinMode(9, OUTPUT);  // use pins 9 and 10 for motor output
4 pinMode(10, OUTPUT);
5 analogWrite(9, 100); // PWM 100 on pin 9 makes the motor rotate
6 analogWrite(10, 0);
7 delay(3000);         // This is allowed to continue for 3 seconds
8 analogWrite(9, 0);   // 0 on pin 9 stops the motor
9 analogWrite(10, 0);
10 }
11 void loop() {
12 // what is put here will run in an infinite loop
13 }
```

Arduino Code 10.2 Rotating the DC motor in both directions. Available at `Origin/user-code/dcmotor/arduino/dcmotor-both/dcmotor-both.ino`, see Footnote 2 on page 2.

```
1 void setup() {
2 Serial.begin(115200); // set the baudrate
```

```
3 pinMode(9, OUTPUT);     // use pins 9 and 10 for motor output
4 pinMode(10, OUTPUT);
5 analogWrite(9, 100);    // Motor runs at a low speed
6 analogWrite(10, 0);
7 delay(3000);           // 3 second delay
8 analogWrite(9, 0);      //
9 analogWrite(10, 100);   // Motor runs in the reverse direction for
10 delay(2000);           // 2 seconds
11 analogWrite(9, 0);      // Motor is stopped
12 analogWrite(10, 0);     //
13 }
14 void loop(){
15   // Code here runs in an infinite loop
16 }
```

Arduino Code 10.3 Rotating the DC motor in both directions in a loop. Available at `Origin/user-code/dcmotor/arduino/dcmotor-loop/dcmotor-loop.ino`, see Footnote 2 on page 2.

```
1 int i;
2 void setup() {
3 Serial.begin(115200);    // set the baudrate
4 pinMode(9,OUTPUT);       // use pins 9 and 10 for motor output
5 pinMode(10,OUTPUT);
6 for(i = 0; i < 4; i++){
7   analogWrite(9, 100);    // Motor runs at a low speed
8   analogWrite(10, 0);
9   delay(3000);           // 3 second delay
10   analogWrite(9, 0);
11   analogWrite(10, 0);     // Motor stops for
12   delay(2000);           // 1 seconds
13   analogWrite(9, 0);      //
14   analogWrite(10, 100);   // Motor runs in the reverse direction for
15   delay(2000);           // 2 seconds
16   analogWrite(9, 0);      // Stop the
17   analogWrite(10, 0);     // motor rotating
18   delay(1000);           // for 1 second
19 }
20 }
21 void loop(){
22 }
```

10.3 Controlling the DC motor from OpenModelica

10.3.1 Controlling the DC motor

In this section, we discuss how to carry out the experiments of the previous section from OpenModelica. We will list the same three experiments, in the same order.

As mentioned earlier, the shield must be removed from the Arduino Uno and the Arduino Uno needs to be connected to the computer with a USB cable, as shown in Fig. 2.4. The reader should go through the instructions given in Sec. 3.2 before getting started.

Note: The readers are advised to affix a small (very lightweight) piece of paper at the tip of the shaft of the DC motor. That will help them observe the direction of rotation of the DC motor while running the experiments.

1. In the first experiment, we will learn how to drive the DC motor from OpenModelica. The code for this experiment is given in OpenModelica Code 10.1. As explained earlier in Sec. 4.4.1, we begin with importing the two packages: Streams and SerialCommunication followed by setting up the serial port. Next, the code has a command of the following form:

```
cmd_dcmotor_setup(1, H-Bridge type, Motor number, PWM pin 1, PWM pin 2)
```

   As mentioned earlier, this chapter makes use of an H-Bridge circuit which allows direction of the current passing through the DC motor to be changed. We are using L293D as an H-Bridge circuit in this book. Thus, we will pass the value 3 for H-Bridge type. The OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4, supports three types of H-Bridge circuit. Table 10.1 provides the values to be passed for different H-Bridge circuits. Next argument in the command given above is Motor number. Here, we pass the value 1. Finally, we provide the PWM pins to which the DC motor is connected. As shown in Fig. 10.4, pins 9 and 10 are connected to the input of the breakout board. As a result, the command `cmd_dcmotor_setup` becomes

```
1 sComm.cmd_dcmotor_setup(1, 3, 1, 9, 10) "Setup DC motor of type 3 (L293D), motor 1, pin 9 and 10";
```

   The next line of OpenModelica Code 10.1 is of the following form:

```
cmd_dcmotor_run(1, Motor number, [sign] PWM value)
```

   Here, we will pass the value 1 in Motor number. As mentioned earlier, for each of the PWM pins on Arduino Uno board, the input can come from 8 bits. Thus, these pins can supply values between -255 and $+255$. Positive values correspond to clockwise rotation while negative values correspond to anti-clockwise rotation. Based on the PWM value and polarity, corresponding analog voltage is generated. We put a PWM value of 100 to make the DC motor

run at an intermediate speed. As a result, the command `cmd_dcmotor_run` becomes

```
1       sComm.cmd_dcmotor_run(1, 1, 100) "Motor 1 runs at PWM 100";
```

The above-mentioned command does not say for how long the motor should run. This is taken care of by the `sleep` command, as given below:

```
1       sComm.delay(3000) "This is allowed to continue for 3 seconds
    ";
```

With this, the DC motor will run for 3000 milliseconds or 3 seconds. At last, we release the DC motor, as shown below:

```
1       sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
```

With the execution of this command, the PWM functionality on the Arduino Uno pins is ceased. This has the motor number as an input parameter. At last, we close the serial port.

Note: If the sleep command (at line 17 of OpenModelica Code 10.1) were not present, the DC motor will not even run: soon after putting the value 100, the DC motor would be released, leaving no time in between. On the other hand, if the DC motor is not released (*i.e.*, line number 18 of OpenModelica Code 10.1 being commented), the DC motor will go on rotating. That's why, it may be inferred that line number 18 of OpenModelica Code 10.1 is mandatory for every program. We encourage the readers to run OpenModelica Code 10.1 by commenting any one or two of the lines numbered 17 and 18. Go ahead and do it - you will not break anything. At the most, you may have to unplug the USB cable connected to Arduino Uno and restart the whole thing from the beginning.

2. It is easy to make the DC motor run in the reverse direction by changing the sign of PWM value being written. This is done in OpenModelica Code 10.2. In this code, we make the DC motor run in one direction for 3 seconds and then make it rotate in the reverse direction for 2 seconds. The rotation in reverse direction is achieved by putting -100 in the command `cmd_dcmotor_run`, as shown below:

```
1       sComm.cmd_dcmotor_run(1, 1, -100) "Motor 1 runs at PWM -100
    in reverse direction";
```

After adding a `sleep` of 2 seconds, we release the motor by issuing the command `cmd_dcmotor_release`, followed by closing the serial port:

```
sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
```

With this, the motor comes to a halt.

3. Next, we make the DC motor run in forward and reverse directions, in a loop. This is done through OpenModelica Code 10.3. We first write PWM +100 for 3 seconds. After that, halt the motor for 2 seconds by writing zero PWM value. Next, make the motor rotate in the reverse direction by writing PWM −100 for two seconds. Next, we make the motor stop for one second. This procedure is put in a `for` loop which runs for 4 iterations. At last, we release the motor by issuing the command `cmd_dcmotor_release`, followed by closing the serial port.

10.3.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 10.1 Rotating the DC motor. Available at Arduino -> SerialCommunication -> Examples -> dcmotor -> dcmotor_clock.

```
model dcmotor_clock "Rotate DC Motor clockwise"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer c_ok(fixed = false);
algorithm
  when initial() then
    ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
    is 115200";
    sComm.delay(2000);
    if ok <> 0 then
      strm.print("Unable to open serial port, please check");
    else
      sComm.delay(1000);
      sComm.cmd_dcmotor_setup(1, 3, 1, 9, 10) "Setup DC motor of type 3
     (L293D), motor 1, pin 9 and 10";
      sComm.cmd_dcmotor_run(1, 1, 100) "Motor 1 runs at PWM 100";
      sComm.delay(3000) "This is allowed to continue for 3 seconds";
      sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
    end if;
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  annotation(
```

```
23     experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 10));
24 end dcmotor_clock;
```

OpenModelica Code 10.2 Rotating DC motor in both directions. Available at Arduino -> SerialCommunication -> Examples -> dcmotor -> dcmotor_both.

```
1  model dcmotor_both "Rotate DC Motor in both directions"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer c_ok(fixed = false);
7  algorithm
8    when initial() then
9      ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
       is 115200";
10     sComm.delay(2000);
11     if ok <> 0 then
12       strm.print("Unable to open serial port, please check");
13     else
14       sComm.cmd_dcmotor_setup(1, 3, 1, 9, 10) "Setup DC motor of type 3
        (L293D), motor 1, pin 9 and 10";
15       sComm.cmd_dcmotor_run(1, 1, 100) "Motor 1 runs at PWM 100";
16       sComm.delay(3000) "for 3 seconds";
17       sComm.cmd_dcmotor_run(1, 1, -100) "Motor 1 runs at PWM -100 in
       reverse direction";
18       sComm.delay(2000) "for 2 seconds";
19       sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
20     end if;
21     c_ok := sComm.close_serial(1) "To close the connection safely";
22   end when;
23   annotation(
24     experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
       = 10));
25 end dcmotor_both;
```

OpenModelica Code 10.3 Rotating the DC motor in both directions in a loop. Available at Arduino -> SerialCommunication -> Examples -> dcmotor -> dcmotor_loop.

```
1  model dcmotor_loop "Rotate DC Motor in both directions in a loop"
2    extends Modelica.Icons.Example;
3    import sComm = Arduino.SerialCommunication.Functions;
4    import strm = Modelica.Utilities.Streams;
5    Integer ok(fixed = false);
6    Integer c_ok(fixed = false);
7  algorithm
```

```
8   when initial() then
9     ok := sComm.open_serial(1, 2, 115200) "COM port is 2 and baud rate
      is 115200";
10    sComm.delay(2000);
11    if ok <> 0 then
12      strm.print("Unable to open serial port, please check");
13    else
14      sComm.cmd_dcmotor_setup(1, 3, 1, 9, 10) "Setup DC motor of type 3
      (L293D), motor 1, pins 9 and 10";
15      for i in 1:4 loop
16        sComm.cmd_dcmotor_run(1, 1, 100) "Motor 1 runs at PWM 100";
17        sComm.delay(3000) "for 3 seconds";
18        sComm.cmd_dcmotor_run(1, 1, 0) "Halt the motor";
19        sComm.delay(2000) "for 2 seconds";
20        sComm.cmd_dcmotor_run(1, 1, -100) "Run it at PWM 100 in reverse
      direction";
21        sComm.delay(2000) "for 2 seconds";
22      end for;
23      sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
24    end if;
25    c_ok := sComm.close_serial(1) "To close the connection safely";
26  end when;
27  annotation(
28    experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
      = 10));
29 end dcmotor_loop;
```

Chapter 11

Implementation of Modbus Protocol

In the previous chapters, we have discussed the programs to experiment with the sensors and actuators that come with the shield, a DC motor, and a servomotor. One may categorize these programs as either basic or intermediate. In this chapter, we will learn one of the advanced applications that can be built using the toolbox. Recall the FLOSS discussed in the book, by default, does not have the capability to connect to Arduino. All such add-on functionalities are added to the FLOSS using toolboxes. Beginners might want to skip this chapter in the first reading. This experiment enables interfacing Modbus-based devices with the FLOSS-Arduino toolbox. This functionality has a wide number of applications in the industrial sector.

11.1 Preliminaries

Modbus is an open serial communication protocol developed and published by Modicon in 1979 [16] [17]. Because of ease of deployment and maintenance, it finds wide applications in industries. The Modbus protocol provides a means to transmit information over serial lines between several electronic devices to control and monitor them. The controlling device requests for reading or writing information and is known as the Modbus master/client. On the other hand, the device supplying the information is called Modbus slave/server. All the slaves/servers have a unique id and address. Typically, there is one master and a maximum of 247 slaves [18]. Fig. 11.1 shows a representation of Modbus protocol.

During the communication on a Modbus network, the protocol determines how the controller gets to know its device address, recognizes the the message provided and decides the action to be taken, and accordingly extracts data and information

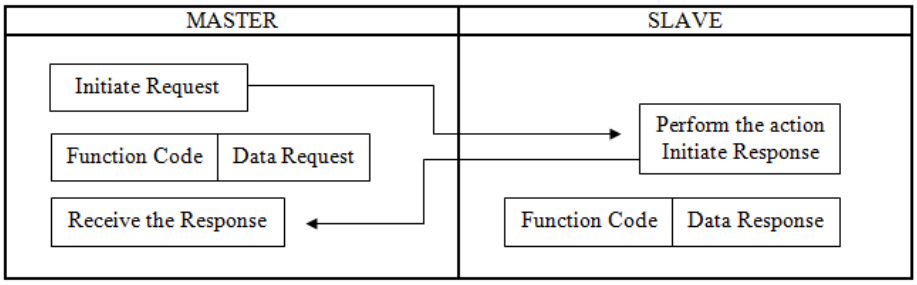


Figure 11.1: Block diagram representation of the Protocol

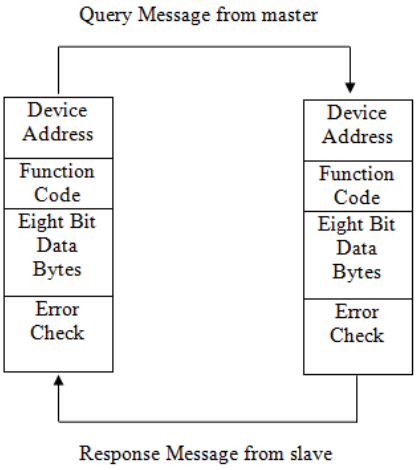


Figure 11.2: Master-Slave Query-Response Cycle

contained in the message. A typical structure of the communication protocol is shown in Fig. 11.2. The data is sent as a series of zeros and ones, *i.e.*, bits wherein zeros are sent as positive voltages and ones as negative.

Different versions of Modbus protocol exist on serial lines, namely Modbus RTU, ASCII, and TCP [18]. The energy meter used in this chapter supports the Modbus RTU protocol. In Modbus RTU, the data is coded in binary and requires only one communication byte. This is ideal for use over RS232 or RS485 networks at baud rates between 1200 and 115K.

RS485 is one of the most widely used bus standards for industrial applications. It uses differential communication lines to communicate over long distances and requires a dedicated pair of signal lines, say A and B, to exchange information.

Table 11.1: Pins available on RS485 and their usage

Pin name	Usage
Vcc	5V
B	Inverting receiver input
A	Non-inverting receiver input
GND	Ground (0V)
RO	Receiver output
RE	Receiver enable
DE	Data enable
DI	Data input

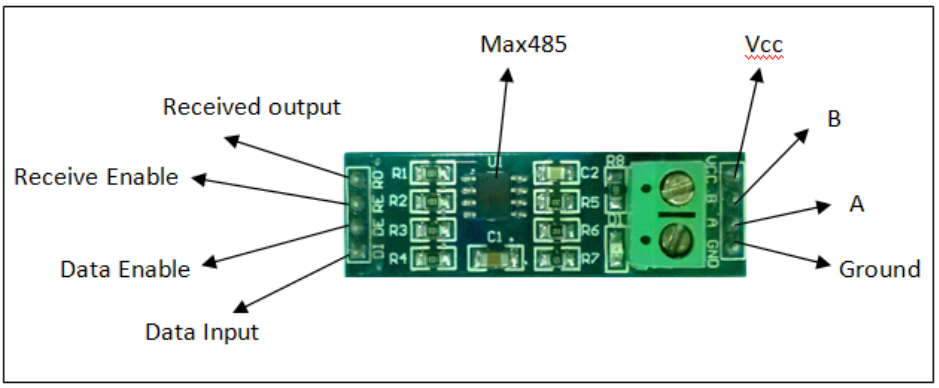


Figure 11.3: Pins in RS485 module

Here, the voltage on one line equals to the inverse of the voltage on the other line. In other words, the output is 1, if A - B $>$ 200mV, and 0, if B - A $>$ 200mV. Fig. 11.3 shows the pins available on a typical RS485 module. As shown in Fig. 11.3, there are four pins on each side of the module. Table 11.1 summarizes the usage of these pins.

11.1.1 Energy meter

An energy meter is a device that measures the amount of electricity consumed by the load. This book makes use of the EM6400 series energy meter. It is a multifunction digital power meter by Schneider Electric. It reads various parameters such as phase voltage, current, active power, reactive power, power factor, etc. Before using the meter, one has to program system configuration, PT, CT ratios, communication

Table 11.2: Operations supported by Modbus RTU

Function Code	Action	Table Name
01 (01 hex)	Read	Discrete Output Coils
05 (05 hex)	Write single	Discrete Output Coil
15 (0F hex)	Write multiple	Discrete Output Coils
02 (02 hex)	Read	Discrete Input Contacts
04 (04 hex)	Read	Analog Input Registers
03 (03 hex)	Read	Analog Output Holding Registers
06 (06 hex)	Write single	Analog Output Holding Register
16 (10 hex)	Write multiple	Analog Output Holding Registers

Table 11.3: Individual parameter address in EM6400

Parameter	Description	Address
V1	Voltage phase 1 to neutral	3927
A1	Current, phase 1	3929
W1	Active power, phase 1	3919

parameters through front panel keys. The reason behind using this energy meter is the fact that it supports the Modbus RTU protocol for communication.

Multiple operations can be performed with devices supporting Modbus. Every operation has its own fixed function code (coil status - 01, input status - 02, holding registers - 03, input registers - 04, etc.), which is independent of devices. The function code tells the slave which table to access and whether to read from or write to the table. All the parameter values are stored in the output holding registers. Different holding registers hold the values of different parameters. Table 11.2 summarizes the various operations which Modbus RTU supports. One can locate the addresses of individual parameters in the user manual for EM6400. Table 11.3 provides the addresses for three individual parameters, which will be accessed in this chapter.

In Modbus protocol, the master needs to send a request packet (referred as RQ hereafter) to the slave to read any of the slave's parameters. When the slave receives an RQ, it needs to come up with a response packet (referred as RP hereafter), which contains the value requested by the master. In other words, an RQ is a message from the master to a slave and an RP is a message from the slave back to the master. We will first explain the structure of an RQ, followed by an example. An RQ consists of the following fields:

1. Slave id: The first byte of every Modbus message is a slave id. The master specifies the id of the slave to which the request message is addressed. Slaves

Table 11.4: A request packet to access V1 in EM6400

Field of the RQ	Value for reading V1
Slave id	01
Function code	03
Address of the register	3926 (hex value = 0F56)
Number of registers	02
CRC bytes	270F

must specify their own id in every response message (RP).

2. Function code: The second byte of every Modbus message is a function code. This code determines the type of operation to be performed by the slave. Table 11.2 enlists the various function codes.

3. Address of the register: After the above two bytes, RQ specifies the data address of the first register requested.

4. Number of registers: This field denotes the total number of registers requested.

5. CRC bytes: The last two bytes of every Modbus message are CRC bytes. CRC stands for Cyclic Redundancy check. It is added to the end of every Modbus message for error detection. Every byte in the message is used to calculate the CRC. The receiving device also calculates the CRC and compares it to the CRC from the sending device. If even one bit in the message is received incorrectly, the CRCs will be different, resulting in an error.

   Note: There are some online tools [19] by which one can calculate the CRC bytes. However, one should note that the calculated CRC bytes should be mentioned in little-endian format, which means that the first register contains the least significant bit (LSB) and the next register contains the most significant bit (MSB).

Let us say, we want to access V1 (Voltage phase 1 to neutral) in the energy meter. From Table 11.3, it may be noted that the address of V1 is 3927. The size of each Modbus register is 16 bits and all EM6400 readings are of 32 bits. So, each reading occupies two consecutive Modbus registers. Thus, we need to access two consective holding registers (starting from 3926) to get V1. Table 11.4 summarizes the values for the various fields in the RQ required to read/access V1. Now, we explain the structure of an RP, followed by an example. An RP consists of following fields:

1. Slave id: In an RP, the slaves must specify their own id.

Table 11.5: A response packet to access V1 in EM6400

Field of the RP	Value for reading V1
Slave id	01
Function code	03
Number of data bytes to follow	04
Data in the first requested register	2921
Data in the second requested register	4373
CRC bytes	D2B0

2. Function code: Like the RQ, the second byte of RP is the function code. This code determines the type of operation to be performed by the slave. Table 11.2 enlists the various function codes.

3. Number of data bytes to follow: It refers to the total number of bytes read. As our RQ has 2 registers each of two bytes, we expect a total of 4 bytes.

4. Data in the first requested register: It refers to the data stored in the first register.

5. Data in the second requested register: It refers to the data stored in the second register.

6. CRC bytes: As stated earlier, the last two bytes of every Modbus message are CRC bytes. Like RQ, the receiving device also calculates the CRC and compares it to the CRC from the sending device.

Let us consider the RP, which we have received as a response to the RQ mentioned in Table 11.4. Table 11.5 summarizes the values for the various fields in this RP. In this RP, we consider the data in the two requested registers to be 43732921 in hexadecimal. The reason behind keeping the data in the second requested register as the MSB is that the obtained values are being read in little-endian format. After converting this value to floating point using the IEEE Standard for Floating-Point Arithmetic (IEEE 754), we obtain the value as 243.16. Thus, the value of V1 (Voltage phase 1 to neutral) in the energy meter is found to be 243.16 Volts.

11.1.2 Endianness

Most of the numeric values to be stored in the computer are more than one byte long. Thus, there arises a question of how to store the multibyte values on the computer machines where each byte has its own address *i.e.*, which byte gets stored at the "first" (lower) memory location and which bytes follow in higher memory locations.

Table 11.6: Memory storage of a four-byte integer in little-endian and big-endian

Memory Address	Byte	Little-endian	Big-endian
3900	8A43	MSB	LSB
3901	436B	LSB	MSB

For example, let us picture this. A two-byte integer 0x5E5F is stored on the disk by one machine, with the 0x5E (MSB) stored at the lower memory address and the 0x5F (LSB) stored at a higher memory address. But there is a different machine that reads this integer by picking 0x5F for the MSB and 0x5E for the LSB, giving 0x5F5E. Hence, it results in a disagreement on the value of the integer between the two machines. However, there is no so-called "right" ordering to store the bytes in the case of multibyte quantities. Hardware is built to store the bytes in a particular fashion, and as long as compatible hardware reads the bytes in the same fashion, things are fine. Following are the two major types of storing the bytes:

1. Little-endian: If the hardware is designed so that the LSB of a multibyte integer is stored "first"at the lowest memory address, then the hardware is said to be little-endian. In this format, the "little" end of the integer gets stored first and the next bytes are stored in higher (increasing) memory locations.

2. Big-endian: Here, the hardware is designed so that the MSB of a multibyte integer is stored "first"at the lowest memory address. Thus, the "big" end of the integer gets stored first and accordingly the next bytes get stored in higher (increasing) memory locations.

For example, let us take a four-byte integer 0x436B84A3. Considering that the read holding registers in Modbus protocol are 16-bits each, the LSB (or the little end) of this integer is 0x84A3, and the MSB (or the big end) of this integer is 0x436B. Then, the memory storage patterns for the integer would be like that shown in Table 11.6.

To represent the hexadecimal values of the read holding registers into user friendly decimal (floating point) values, we follow IEEE 754 standard. Most common standards for representing floating point numbers are:

1. Single precision: In this standard, 32 bits are used to represent a floating-point number. Out of these 32 bits, one bit is for the sign bit, 8 bits for the exponent, and the remaining 23 bits for mantissa.

2. Double precision: Here, 64 bits are used to represent a floating-point number. Out of these 64 bits, one bit represents the sign bit, 11 bits for the exponent,

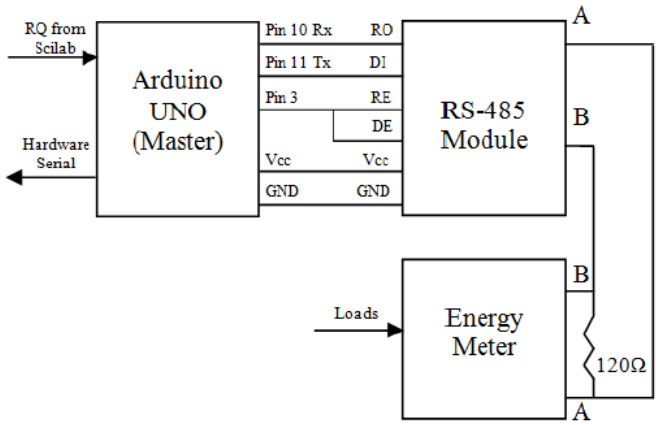


Figure 11.4: Block diagram for reading the parameters in energy meter

and the remaining 52 bits for mantissa. As the name indicates, this standard is used where precision matters more.

There are several online converters [20] which peform the IEEE 754 floating point conversion. In this chapter, a function has been formulated for this conversion, wherever needed.

11.2 Setup for the experiment

This section discusses the setup for configuring Arduino Uno as Modbus master and energy meter as the slave. The block diagram is shown in Fig. 11.4 , whereas Fig. 11.5 presents the actual setup. The following steps discuss the various connections of this setup:

1. Arduino Uno has only one serial port. It communicates on the digital pins 0 and 1 as well as on the computer via USB. Since we want serial communication which shouldn't be disturbed by the USB port and the Serial Monitor, we use the Software Serial library. Using this library, we can assign any digital pins as RX and TX and use it for serial communication. In this experiment, pin 10 (used as RX) and pin 11 (used as TX) are connected to RO (Receive Out) and DI (Data In) pins of the RS485 module respectively.

2. DE (Data Enable) and RE (Receive Enable) pins of RS 485 are shorted and connected to digital pin 3 of the Arduino Uno board. This serves as a control pin that will control when to receive and transmit serially.

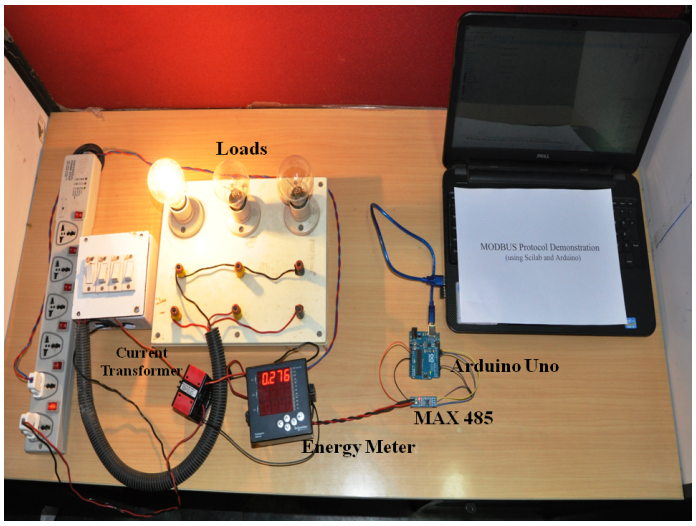


Figure 11.5: Experimental set up for reading energy meter

3. Vcc and GND of the RS485 module are connected to Vcc and GND of the Arduino Uno board.

4. A and B pins of RS485 module are connected to A (Pin 7) and B (Pin 14) pins of the energy meter. These two pins of the energy meter are meant for RS485 communication.

5. A $120k\Omega$ termination resistor is connected between pins A and B to avoid reflection losses in the transmission line.

11.3 Software required for this experiment

Apart from the FLOSS-Arduino toolbox, the software for this experiment comprises two parts:

1. Firmware for Arduino Uno: This firmware is needed to communicate with the FLOSS (using serial interface), and with RS485 module (using Software Serial interface). Control logic to enable receive and transmit modes of MAX485 chip is also present in this firmware. Fig. 11.6 demonstrates the overall implementation of this firmware. The firmware is provided in Sec. 11.3.1.

2. FLOSS code: This code requests the parameters in the energy meter by sending an RQ to Arduino Uno from the FLOSS. Then it waits till an RP is available

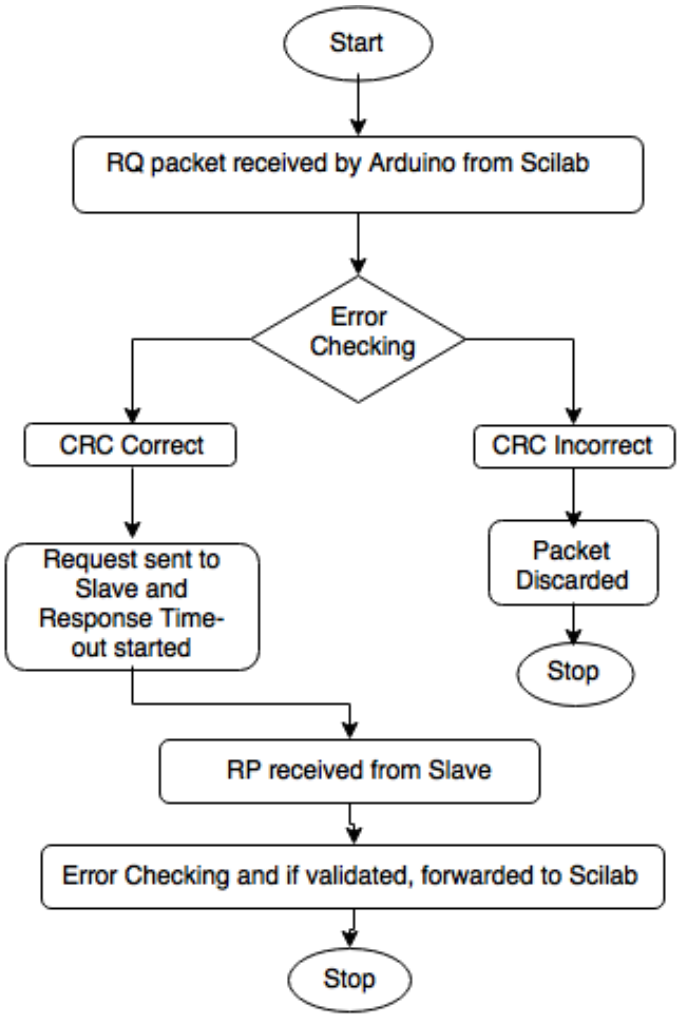


Figure 11.6: Flowchart of Arduino firmware

from the Arduino Uno. After receiving the RP, it extracts the data from this packet and converts it into IEEE 754 floating-point format. The overall implementation is being described below:

(a) Frame an RQ to be sent to the energy meter (slave) in ASCII coded decimal format.

(b) Send the RQ serially to Arduino Uno.

(c) Let Arduino Uno send the RQ to the energy meter via RS485 module.

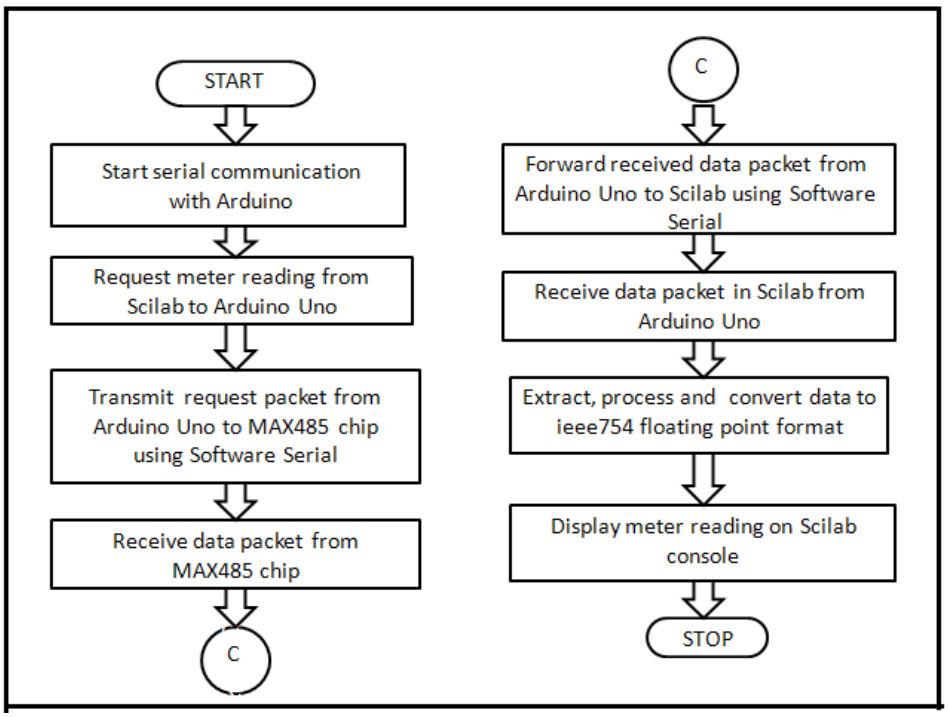


Figure 11.7: Flowchart of the steps happening in the FLOSS code

(d) Let the energy meter send the RP to Arduino Uno via RS485 module.

(e) Read the RP available on Arduino Uno.

(f) Extract the data stored in holding registers from the RP.

(g) Assuming this data to be stored in little-endian format, convert this data in floating-point values using IEEE 754 standard.

(h) Display the value in the Console, output window, Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

Fig. 11.7 presents the sequence in which the steps mentioned above are executed.

11.3.1 Arduino Firmware

Arduino Code 11.1 First 10 lines of the firmware for Modbus. Available at `Origin/user-code/modbus/arduino`, see Footnote 2 on page 2.

```
1 /* ..... crc function ..... */
```

```
2
3 static unsigned char auchCRCHi[] = {0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
      0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
4 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
      0x40, 0x01, 0xC0,
5 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1,
      0x81, 0x40, 0x01,
6 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
      0xC0, 0x80, 0x41,
7 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
      0x00, 0xC1, 0x81,
8 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80,
      0x41, 0x01, 0xC0,
9 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
      0x80, 0x41, 0x01,
10 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00,
      0xC1, 0x81, 0x40,
```

11.4 Manifestation of Modbus protocol through OpenModelica

The objective of this experiment is to make the user acquainted with the demonstration of Modbus protocol through the OpenModelica-Arduino toolbox. It gives an insight into how to acquire readings from the energy meter and interpret them accordingly. As explained in Sec. 11.1.1, an energy meter is a device that gives us different electrical parameters, including voltage, current, and power, consumed by a device. Here, we aim to obtain these values using the Python-Arduino toolbox. For data transmission, we have used an RS485 module.

OpenModelica is used for giving the required parameters to Arduino Uno. For example, the user will tell the required slave address to be accessed and the number of registers to be read from or written to. Here, Arduino Uno acts as a master and energy meter as a slave. Therefore, referring to a particular slave address will refer to the registers that hold the desired electrical parameters (current, voltage, power, etc.), which we want to read from the energy meter.

In this experiment, Arduino Uno is connected to the energy meter via an RS485 module which facilitates long-distance communication. OpenModelica sends the RQ to the Arduino Uno which in turn sends it to the energy meter. The energy meter then accesses the values in the required addresses in its memory and transfers them back. This again is in the form of another packet called RP. In this packet, the data is stored in a little-endian hexadecimal format. Thus, we make use of IEEE 754 to obtain the decimal value from this data.

Note: The OpenModelica models presented in this section were tested on the older versions. Now, these codes may require minor changes in the newer versions. We invite the experts to contribute the revised version of the code.

11.5 Reading the electrical parameters from OpenModelica

11.5.1 Reading the electrical parameters

In this section, we will show how to access the three parameters (voltage, current, and active power) in the energy meter. As discussed above, we will send an RQ from OpenModelica to Arduino Uno. Subsequently, Arduino Uno will provide us with an RP, which can be decoded to extract the desired parameter. The reader should go through the instructions given in Sec. 3.2 before getting started.

11.5.2 OpenModelica Code

Unlike other code files, the code/ model for running experiments using OpenModelica are available inside the OpenModelica-Arduino toolbox, as explained in Sec. 3.2.4. Please refer to Fig. 3.15 to know how to locate the experiments.

OpenModelica Code 11.1 Code for Single Phase Current Output. Available at `Origin/user-code/modbus/OpenModelica`, see Footnote 2 on page 2.

```
1  function read_current
2    extends Modelica.Icons.Function;
3
4    external read_voltage() annotation(Library = "Modbus");
5    annotation(Documentation(info = "<html>
6          <h4>Syntax</h4>
7          <blockquote><pre>
8          Arduino.SerialCommunication.Examples.modbus.<b>read_current</b
       >();
9          </pre></blockquote>
10         <h4>Description</h4>
```

OpenModelica Code 11.2 Code for Single Phase Voltage Output. Available at `Origin/user-code/modbus/OpenModelica`, see Footnote 2 on page 2.

```
1  function read_voltage
2    extends Modelica.Icons.Function;
3
4    external read_voltage() annotation(Library = "Modbus");
5    annotation(Documentation(info = "<html>
6          <h4>Syntax</h4>
```

```
7         <blockquote><pre>
8         Arduino.SerialCommunication.Examples.modbus.<b>read_voltage</b>();
9         </pre></blockquote>
10        <h4>Description</h4>
```

OpenModelica Code 11.3 Code for Single Phase Active Power Output. Available at `Origin/user-code/modbus/OpenModelica`, see Footnote 2 on page 2.

```
1 function read_active_power
2   extends Modelica.Icons.Function;
3
4   external read_active_power() annotation(Library = "Modbus");
5   annotation(Documentation(info = "<html>
6         <h4>Syntax</h4>
7         <blockquote><pre>
8         Arduino.SerialCommunication.Examples.modbus.<b>
      read_active_power</b>();
9         </pre></blockquote>
10        <h4>Description</h4>
```

References

[1] T. Martin. Use of scilab for space mission analysis. https://www.scilab.org/community/scilabtec/2009/Use-of-Scilab-for-space-mission-analysis. Seen on 28 June 2015.

[2] B. Jofret. Scilab arduino toolbox. http://atoms.scilab.org/. Seen on 28 June 2015.

[3] oshwa.org. http://www.oshwa.org/definition. Seen on 28 June 2015.

[4] Mateo Zlatar. Open source hardware logo. http://www.oshwa.org/open-source-hardware-logo. Seen on 28 June 2015.

[5] Arduino uno. https://www.arduino.cc/en/uploads/Main/ArduinoUnoFront240.jpg. Seen on 28 June 2015.

[6] Arduino mega. https://www.arduino.cc/en/uploads/Main/ArduinoMega2560_R3_Fronte.jpg. Seen on 28 June 2015.

[7] Lilypod arduino. https://www.arduino.cc/en/uploads/Main/LilyPad_5.jpg. Seen on 28 June 2015.

[8] Arduino phone. http://www.instructables.com/id/ArduinoPhone/. Seen on 28 June 2015.

[9] Candy sorting machine. http://beta.ivc.no/wiki/index.php/Skittles_M%26M%27s_Sorting_Machine. Seen on 28 June 2015.

[10] 3d printer. http://www.instructables.com/id/Arduino-Controlled-CNC-3D-Printer/. Seen on 28 June 2015.

[11] Shield. http://codeshield.diyode.com/about/schematics/. Seen on 28 June 2015.

[12] Openmodelica. https://www.openmodelica.org/. Seen on 2 April 2021.

[13] Thermistor - wikipedia. https://en.wikipedia.org/wiki/Thermistor. Seen on 2 May 2021.

[14] Secrets of arduino pwm. https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM. Seen on 5 May 2021.

[15] Servo. https://www.arduino.cc/reference/en/libraries/servo/. Seen on 6 May 2021.

[16] Modbus. https://modbus.org/. Seen on 6 May 2021.

[17] Paavni Shukla, Sonal Singh, Tanmayee Joshi, Sudhakar Kumar, Samrudha Kelkar, Manas R. Das, and Kannan M. Moudgalya. Design and development of a modbus automation system for industrial applications. In *2017 6th International Conference on Computer Applications In Electrical Engineering-Recent Advances (CERA)*, pages 515–520, 2017.

[18] Simply modbus. https://simplymodbus.ca/. Seen on 6 May 2021.

[19] Online crc. https://www.lammertbies.nl/comm/info/crc-calculation. Seen on 2 May 2021.

[20] Floating point converter. https://www.h-schmidt.net/FloatConverter/IEEE754.html. Seen on 6 May 2021.