Contents

List of Figures

List of Tables

List of Arduino Code

List of Scilab Code

List of Python Code

List of Julia 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 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?

Although Arduino and Scilab are versatile, powerful and free, there has not been much literature that teaches how to integrate them. To address this gap, we have written this book. 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. In this book, we provide Xcos code to drive Arduino Uno board.

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 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. In addition to these toolboxes,

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

we provide the firmware and a program to check it. Finally, 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 three environments: Arduino IDE, Scilab scripts and Xcos.

This book teaches how to access the following sensors and actuators: LED, push-button, DC motor, Potentiometer and Servo motor. A set of two to five programs are given for each. These are given for Arduino IDE, Scilab and Xcos. We 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,

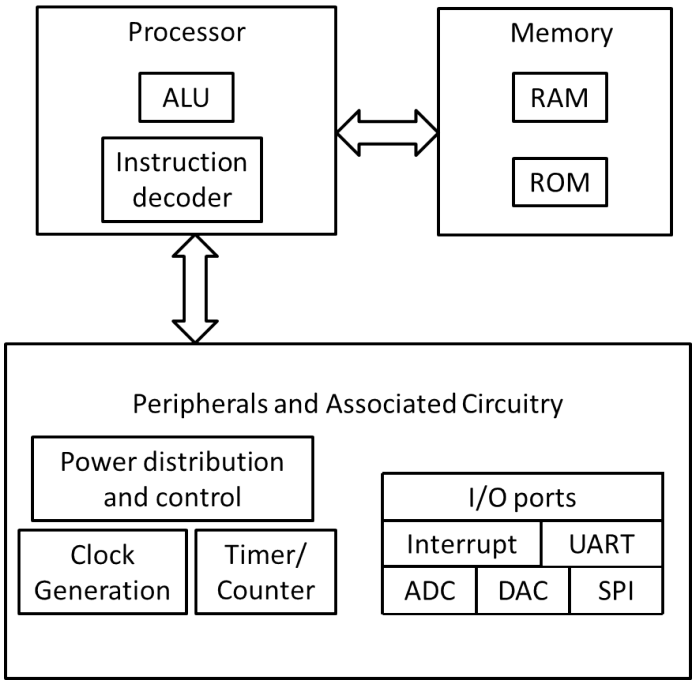


Figure 2.1: Functional block diagram of a microcontroller

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.

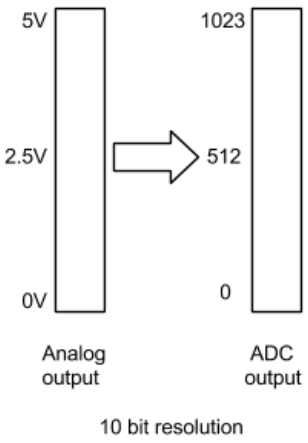


Figure 2.2: ADC resolution

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.

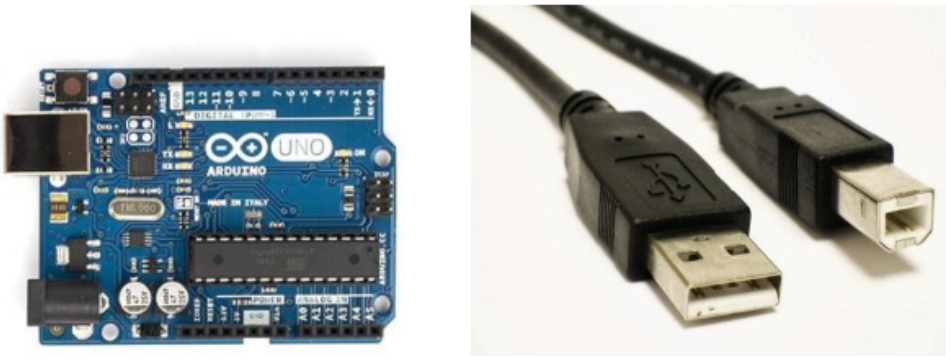


Figure 2.4: Arduino Uno Board

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

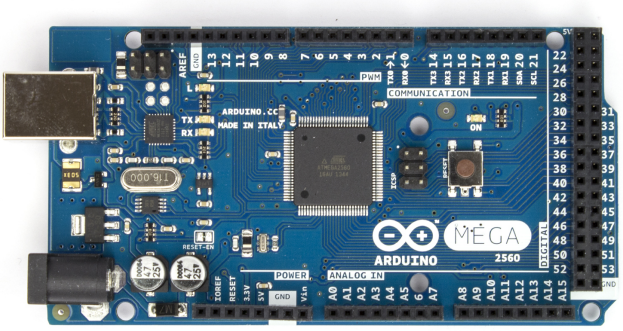


Figure 2.5: Arduino Mega Board

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].

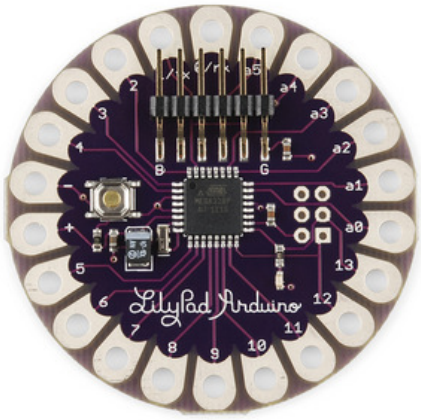


Figure 2.6: LilyPad Arduino Board

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.

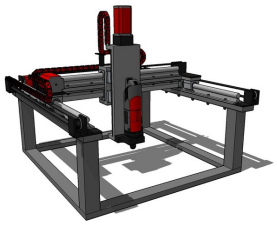


Figure 2.8: 3D printer

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

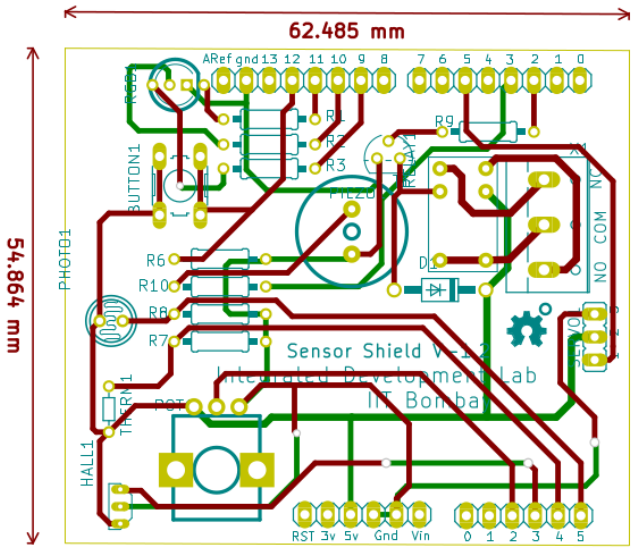


Figure 2.9: PCB image of the shield

   (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.

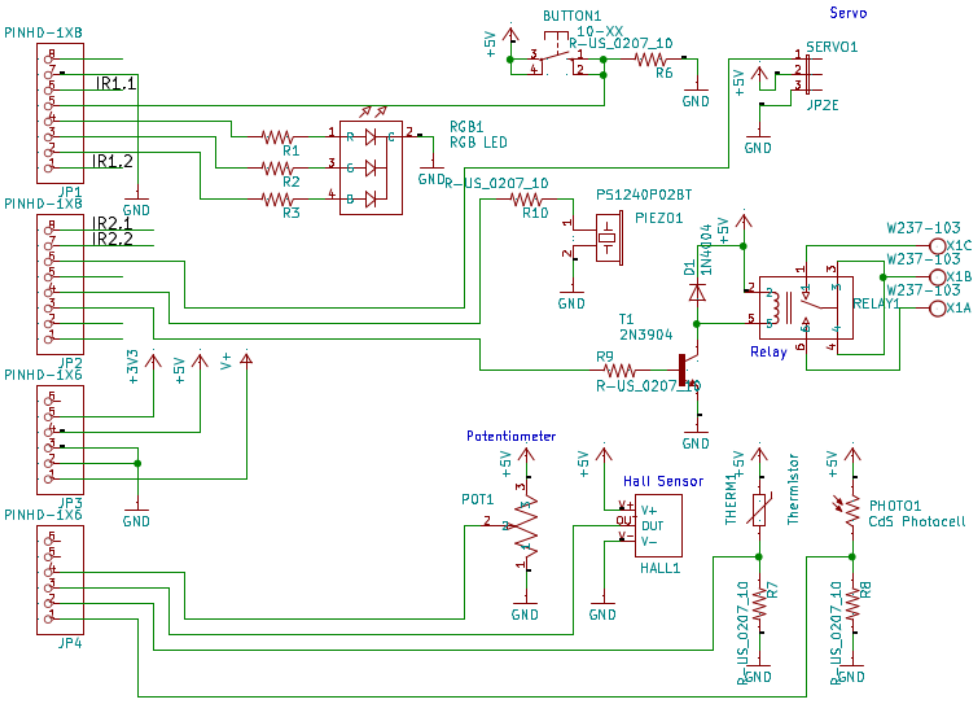


Figure 2.10: Pictorial representation of the schematic of the shield

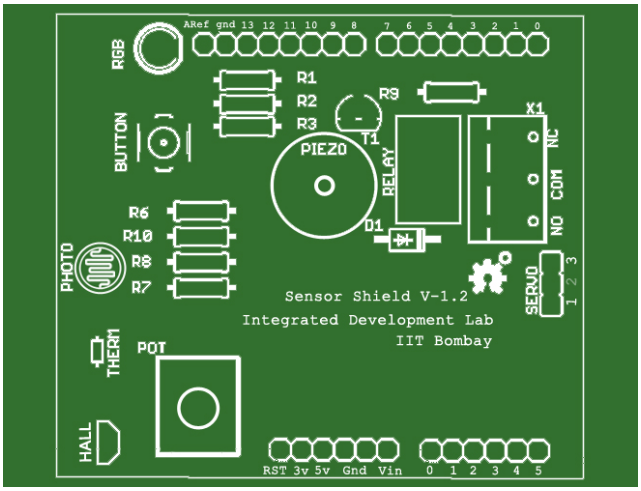


Figure 2.11: PCB of the shield

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

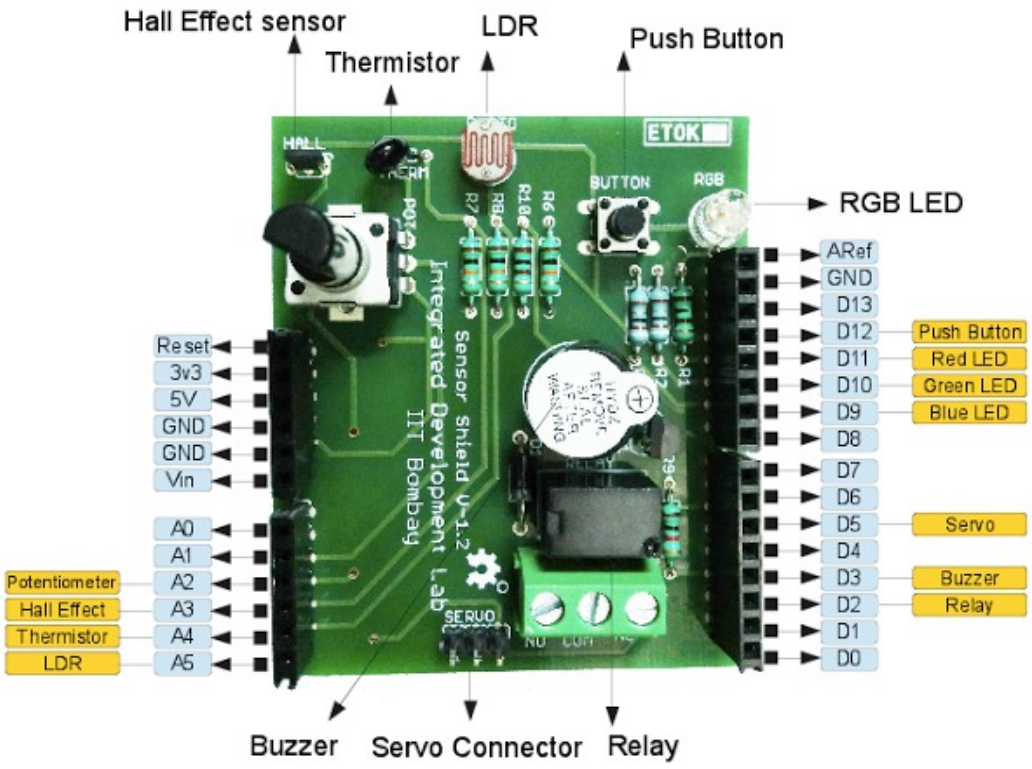


Figure 2.12: Picture of the shield with all components

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 Scilab and Xcos, which are open-source software tools, and a related toolbox that can communicate with Arduino Uno over a serial port using RS232 protocol. Subsequently, we shall discuss other open-source software tools such as Python, Julia, and OpenModelica.

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".

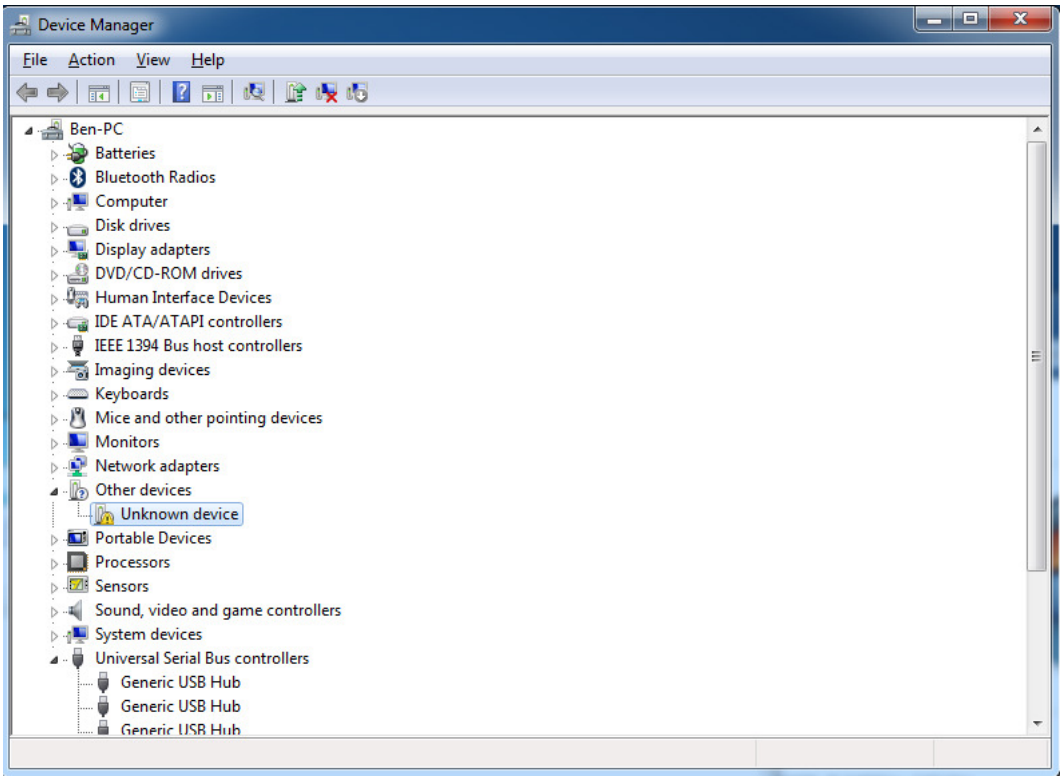


Figure 3.1: Windows device manager

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.

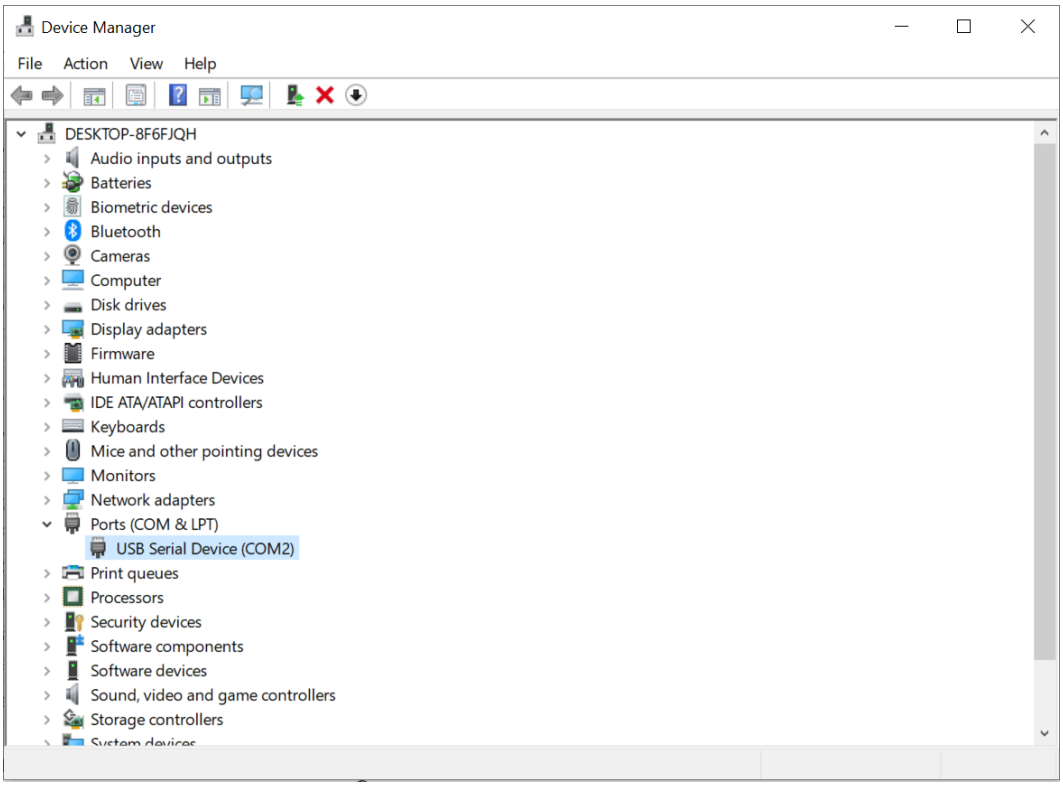


Figure 3.2: Windows device manager

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*`

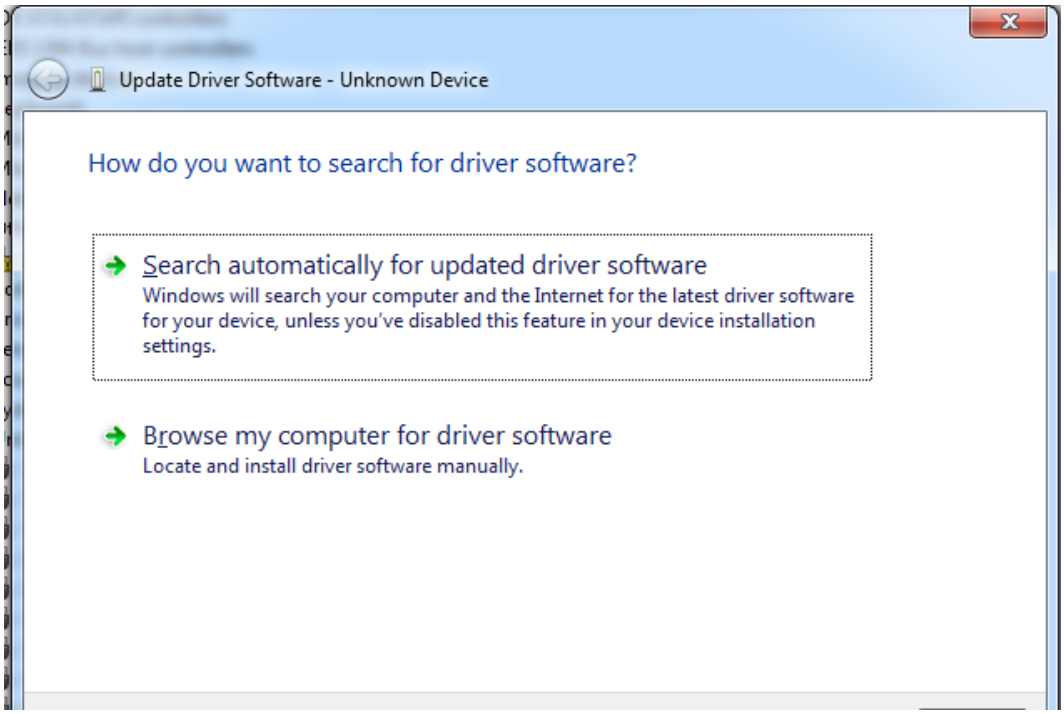


Figure 3.3: Windows update driver option

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

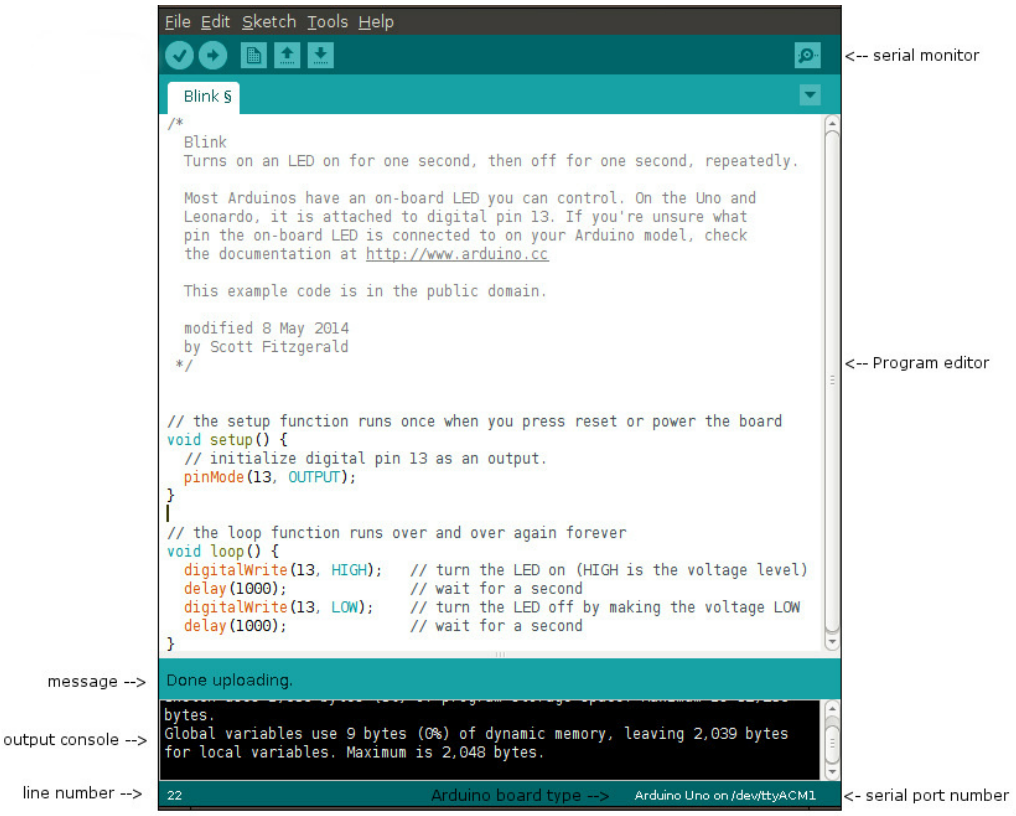


Figure 3.5: Arduino IDE

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 Scilab

Scilab is a free and open-source computing software for science and engineering applications [12]. It is released under GPL compatible CeCILL license. It uses the

state-of-the-art linear algebra package LAPACK, just as in Matlab. Scilab has hundreds of inbuilt functions which cater to a variety of areas such as signal processing, control system design, statistics, optimization, and many more. It has 2D and 3D visualisation capabilities for generating excellent plots. It provides Matlab binary files reading and writing capabilities and also a Matlab to Scilab conversion tool. Scilab can also interact with other major programming languages such as Fortran, C, C++, Python, Java, and TCL/TK [13]. It has a graphical editor called Xcos, which is similar to Simulink of Matlab.

In the upcoming sections, we have provided the steps to install Scilab on Windows and Linux. After installing Scilab, the readers are advised to watch the tutorials on Scilab 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 sixth and thirteenth tutorials, i.e., `Getting Started` and `Xcos Introduction`.

3.2.1 Downloading and installing on Windows

This book uses Scilab 5.5.2 for demonstrating the experiments, both on Windows and Linux. Starting from download, we shall go through the steps to set up Scilab 5.5.2 on Windows OS:

1. Visit the URL `https://www.scilab.org/`. At the top of the page, locate the Download tab and click on it. It will take you to various software versions available for Scilab. On the left side of this page, find Scilab 5.5.2 and click on it. Now, download the executable file for Scilab 5.5.2 compatible with your machine. We will download Scilab 5.5.2 - Windows 64 bits provided under Windows Vista, 7, 8, 10.

2. Locate the executable (.exe) file and double-click on it to begin the installation. All the default parameters of installation are acceptable. It may be noted that Scilab requires internet connectivity during installation on Windows. There is an option at the beginning of the installation to continue offline, but it is not recommended.

Once the installation is complete, Scilab can be launched either from the Start menu or by double-clicking on the Scilab icon created on the Desktop.

3.2.2 Downloading and installing on GNU/Linux Ubuntu

Package managers of Linux do not have the latest versions of Scilab. So, we will install Scilab by downloading the executable file from `https://www.scilab.org/`. Starting from download, we shall go through the steps to set up Scilab 5.5.2 on Linux:

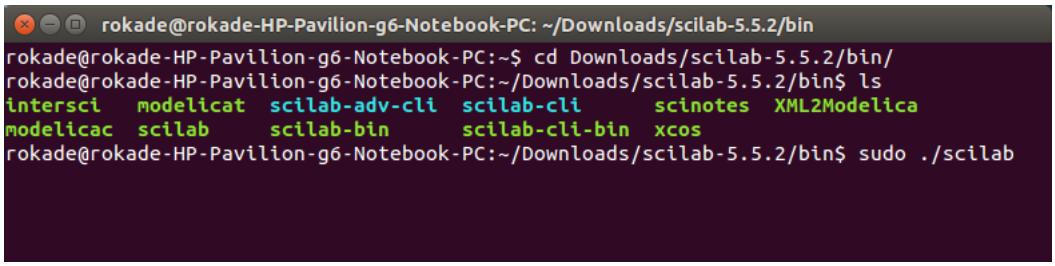


Figure 3.6: Linux terminal to launch Scilab

1. Visit the URL `https://www.scilab.org/`. At the top of the page, locate the Download tab and click on it. It will take you to various software versions available for Scilab. On the left side of this page, find Scilab 5.5.2 and click on it. Now, download the executable file for Scilab 5.5.2 compatible with your machine. We will download Scilab 5.5.2 - Linux 64 bits provided under GNU/Linux.

2. Locate the executable (tar.gz) file and extract it. It is a portable version and needs no installation. Scilab can be launched and used right away.

To launch Scilab, open a terminal by pressing the Alt+Ctrl+T keys together. Change the directory where Scilab is extracted. Browse till the `/bin` directory. Type the command `ls` to see a few Scilab files. Then execute the command `sudo ./scilab`. Note that Scilab needs to be launched with root permissions to be able to communicate with Arduino Uno. This process is illustrated in Fig. 3.6.

3.2.3 Scilab-Arduino toolbox

Scilab, by default, does not have the capability to connect to Arduino. All such add-on functionalities are added to Scilab using toolboxes. Just like we have different installation binaries of Scilab for Windows and Linux, we have different toolboxes types for Windows and Linux. The Scilab Arduino toolbox can be found inside the `Origin/tools/scilab/windows` or `Origin/tools/scilab/linux` directory, see Footnote 2 on page 2. Use the one depending upon which operating system you are using. The Scilab 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 Scilab-Arduino toolbox.

1. First launch Scilab. On a Windows system, one may start/launch Scilab either through the Start menu or by double-clicking on the shortcut icon created on

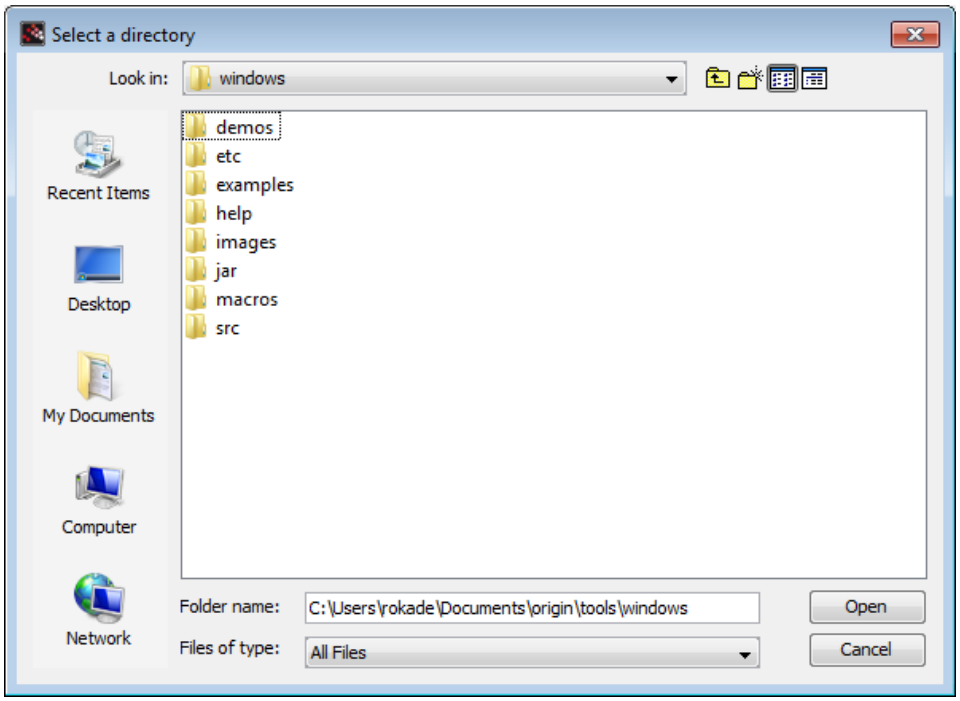


Figure 3.7: Browsing toolbox directory

the Desktop. On a Linux system, one has to start Scilab through a terminal with root permissions, as explained in section 3.2.2.

2. After launching Scilab, first we have to change the working directory. Below the menu bar, locate the tab called `File Browser`. Just below `File Browser`, locate a folder-shaped icon. This icon is used to `Select a directory`. Click on this icon.

3. Then, one has to browse to the toolbox folder `Origin/tools/scilab/windows` or `Origin/tools/scilab/linux`, as the case may be, and click on, `Open`, as shown in Fig. 3.7.

4. After the previous step, the Scilab working directory becomes the toolbox folder. See the `file browser` panel on the left-hand side of the Scilab console, see Fig. 3.8. It will list out the contents of your current working directory. For a check, look for the file `builder.sce`. If you see this file, then you are in the right directory.

5. Next, type the following command on the Scilab console: `exec builder.sce`

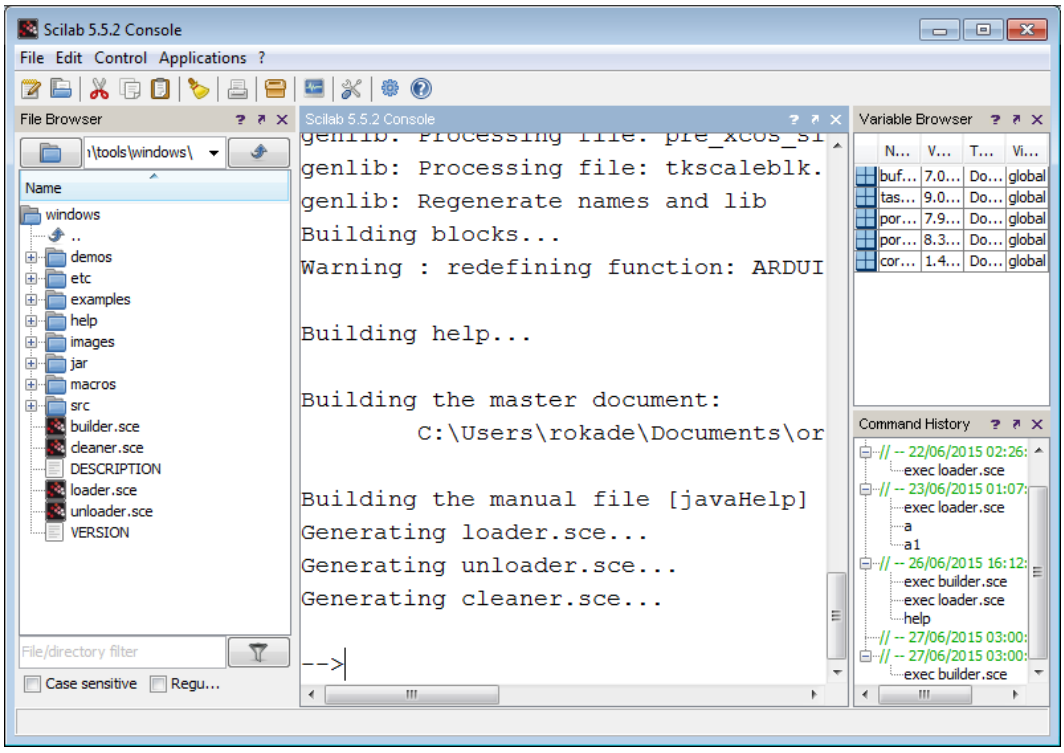


Figure 3.8: Output of builder.sce

- this will build the toolbox and create a file `loader.sce`. This step has to be executed only the first time. The output of this step is illustrated in Fig. 3.8.

6. Next, type the command, `exec loader.sce` - this will load the toolbox. This means all the new functions corresponding to the toolbox are loaded in the workspace. It will also make available new Xcos blocks, if any. The output of this command is as shown in Fig. 3.9. If you clear the workspace for any reason, you will have to execute this command once again[3].

The toolbox is now loaded and available for use.

3.2.4 Identifying Arduino communication port number

Connect Arduino Uno board to your computer. On a Windows system, doing so for the first time will initiate the Windows device identification routine. It may

[3]Be careful not to execute the `clear` command. This will clear the loaded toolbox and you will have to execute the loader.sce file again.

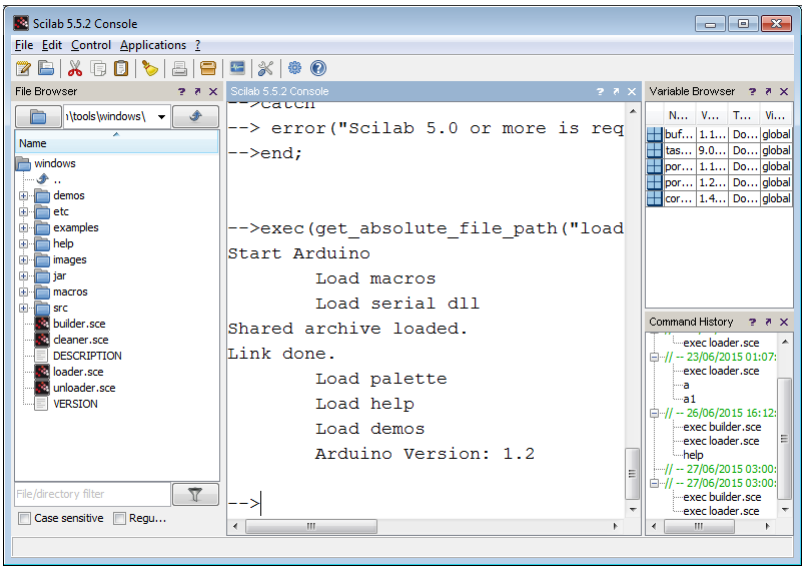


Figure 3.9: Output of loader.sce

take a while before it finishes assigning a COM port number to the Arduino Uno board. If Arduino IDE is installed using the procedure outlined in Sec. 3.1, required USB drivers for Arduino get installed automatically. Hence if you have installed the Arduino IDE, it should not ask for drivers after you connect it. As usually Linux systems come with required drivers, the device is automatically detected by the OS on connection.

Now let us see how to identify the COM port number. For a Windows system, open the Device Manager. To do so, right-click on "My Computer" and choose Properties. The Properties window that will open will have Device Manager in the list on the left-hand side. In the Device Manager window, look for "Ports (COM and LPT)". Double click on it. It will show you the COM number for Arduino Uno.

The result of the above exercise is shown in Fig. 3.10. In this case, the system has detected Arduino with port number 3, which appears as COM3. In this book, we have taken the port for communication as 2 and written code consistent with this assumption. As a result, we will now change it to COM2[4]. To change the port number, double click on the port number. Its properties window will appear. Click on the "Port Settings" tab and then click on "Advanced" button as shown in Fig. 3.11.

Click on the drop-down menu for COM port numbers. Choose the port number

[4]It is possible to leave it at whatever port number one gets. It is also possible to choose any number between 2 and 99. In this case, the port number should be changed accordingly in the code. We will point this out throughout the book.

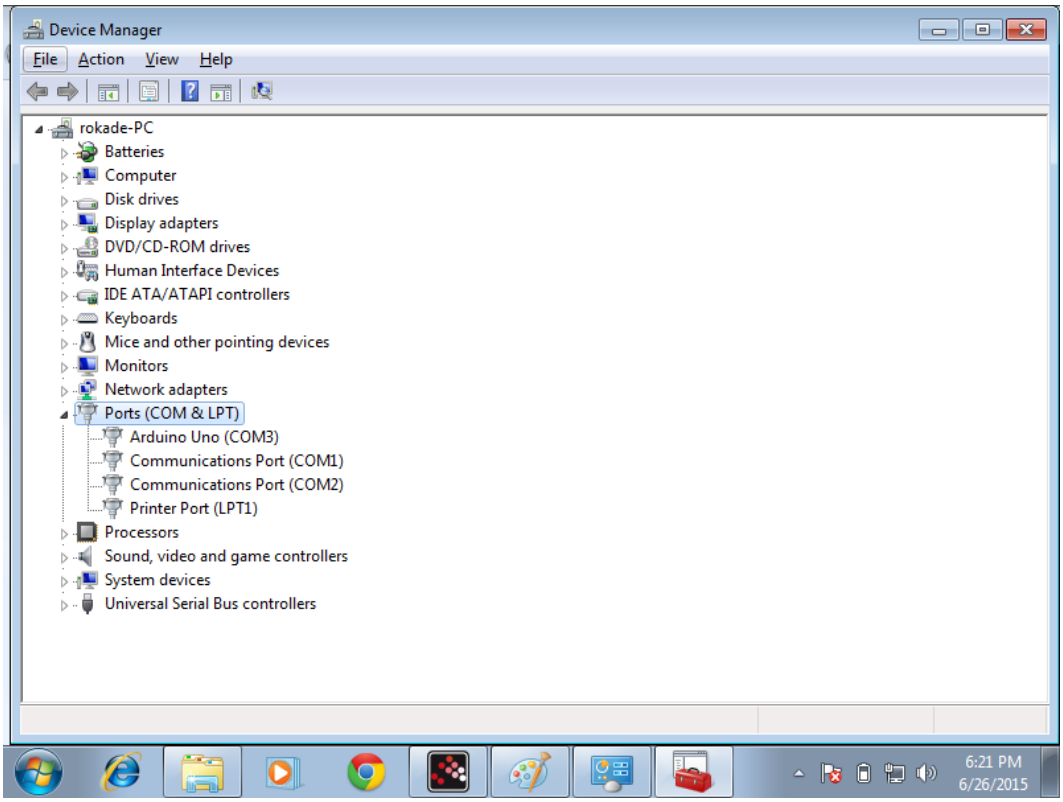


Figure 3.10: Device Manager in windows

COM2. On clicking on "OK", Windows may warn you that the port number is already in use. But given that you do not have any other USB device connected you may force change it. Click on "OK" to close all of the device manager windows. Now, we are set to go ahead with port number 2. The stress on using port number 2 is just to be consistent throughout the book. It is mainly for a beginner.

Now, let us see how to identify the port number on a Linux system. Open a terminal by pressing Alt+Ctrl+T keys together. Then type the following command and press enter, `ls /dev/ttyACM*` - the output of this command is shown in Fig. 3.12. It has detected the Arduino with the port number "ttyACM0". The last character in this string, namely 0, is the port number. You may get 0 or a number such as 1 or 2 in your case, for the port number.

3.2.5 Testing Scilab-Arduino toolbox

Now let us test the functioning of the toolbox.

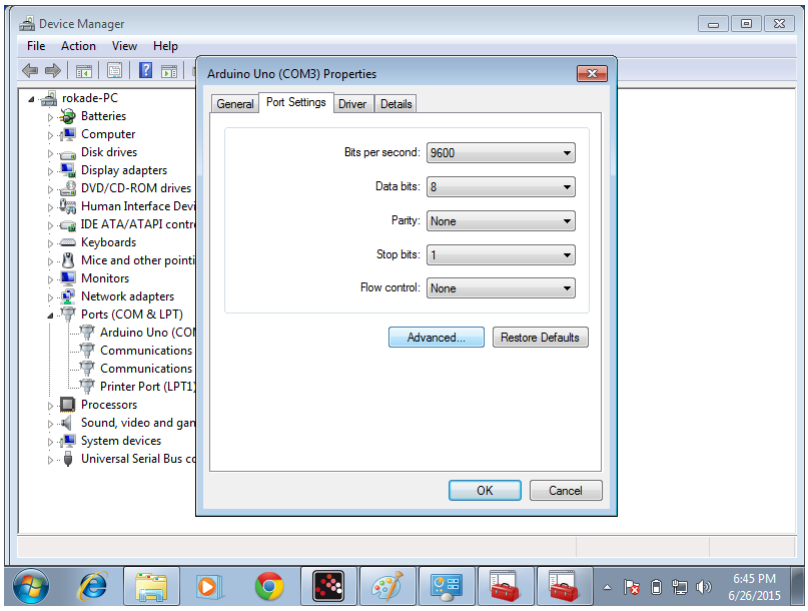


Figure 3.11: COM port properties window

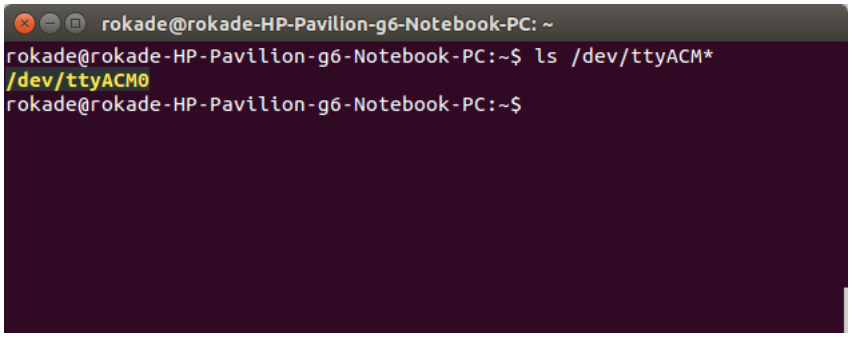


Figure 3.12: Port number on Linux terminal

1. Install Arduino IDE, as explained in Sec. 3.1 and launch it.

2. Read into the Arduino IDE, the firmware Arduino Code 3.1.

3. Using the `Upload` option of the Arduino IDE, load this firmware on to the Arduino Uno board.

4. Inside the `Origin/tools/scilab` directory, locate a file `test_firmware.sce`. This file will be used to test whether the firmware is properly installed. This

is an important step, as the connection between the computer and Arduino breaks down sometimes. The Scilab toolbox is unable to identify this difficulty - it has to be externally done. If this difficulty is not identified and rectified, one will waste a lot of time and effort trying to debug the error. This test has to be done in case of difficulties.

5. In the Scilab console, type `editor` and press the enter key. This will launch the editor. Click on "File" menu and choose "Open". Browse to the directory `Origin/tools/scilab` and choose the file `test_firmware.sce`. It will open Scilab Code 3.1.

6. If you are using a Windows system and have set your port number as COM2, you need not make any changes to the file. Linux users, however, will mostly identify the port number as "ttyACM0". Hence, they need to change the following line number

```
1 h = open_serial(1,2,115200);
```

to

```
h = open_serial(1,0,115200);
```

7. To execute this code, on the menu bar, click on the `Execute`, option. Then choose `File with no echo`. The output will appear on the Console as shown in Fig. 3.13. As shown in the figure, we see the response of this code as "ans = " and "ok" three times. The code basically gives some input to Arduino three times and the program inside it returns "ok" three times. This code thus confirms the working of the Scilab-Arduino toolbox. The code also confirms that the firmware inside the Arduino is correct. It is alright if one or two of the attempts out of three give a blank response. But all the three responses certainly should not be blank[5].

Now let us take a look at the various functions facilitated by the toolbox. The functions provided in the toolbox are as shown in Fig. 3.14. They are basically categorized into four categories: configuration, digital, analog and motors. These functions will be explained in detail in the subsequent chapters as and when they are used.

3.2.6 Firmware

We have provided a Scilab code to check whether the firmware has been properly installed. That code is listed below.

[5]If this step is unsuccessful, one should check the connections and re-install the firmware

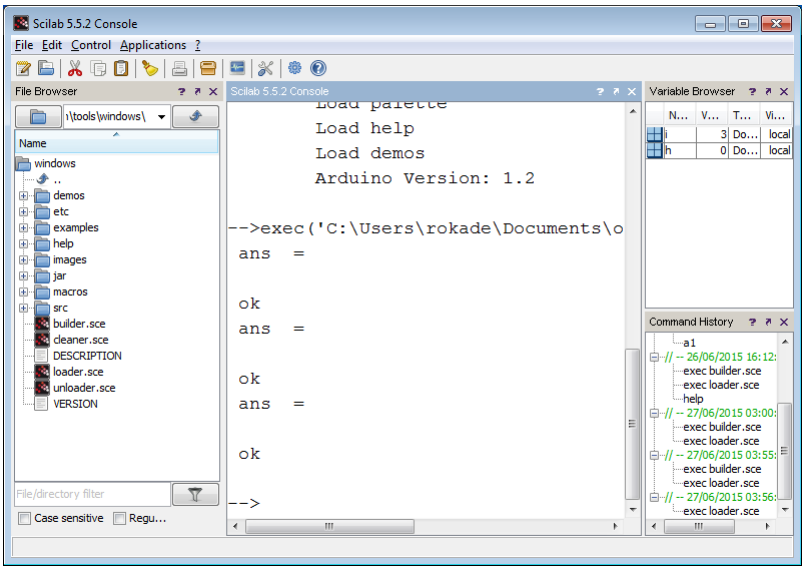


Figure 3.13: Scilab test code output

Configuration	Digital	Analog	Motors
open_serial	cmd_digital_in	cmd_analog_in	cmd_dcmotor_release
close_serial	cmd_digital_out	cmd_analog_in_volt	cmd_dcmotor_run
cmd_arduino_a_control		cmd_analog_out	cmd_dcmotor_setup
cmd_arduino_d_control		cmd_analog_out_volt	cmd_servo_attach
cmd_arduino_meter			cmd_servo_detach
			cmd_servo_move

Figure 3.14: Arduino toolbox functions used in this book

Scilab Code 3.1 A Scilab code to check whether the firmware is properly installed or not. Available at `Origin/tools/scilab/test_firmware.sce`, see Footnote 2 on page 2. Execute this code by following the steps given in section 3.2.5.

```
1 mode(0)
2 h = open_serial(1,2,115200);
3 for i = 1:3
4   write_serial(1,"v",1);
5   read_serial(1,2)
6 end
7 close_serial(1);
```

3.3 Xcos

Xcos is a graphical editor for Scilab [14]. Most of the mathematical manipulations that can be done using Scilab scripts, can be done using Xcos also. The major advantage of Xcos is the intuitive interface and easy connectivity across blocks. Xcos even supports `if else`, `for`, and `while` looping which forms an integral part of any programming language. It is possible to code the entire algorithm using Xcos blocks alone. It is also possible to read from and write to the Scilab workspace through Xcos.

3.3.1 Downloading, installing and testing

Xcos comes pre-installed with Scilab. Hence a separate installation of Xcos is not required. Let us explore the functionalities Xcos has to offer. Xcos basically provides a graphical interface to Scilab.

Xcos can be launched from Scilab by clicking on the Xcos icon available on the Scilab menu bar. It can also be launched by simply typing the command `xcos` in the Scilab console. When Xcos is launched, it will open a palette browser. We have shown this in Fig. 3.15, where we have selected a sine block. At the time of launch, Xcos will also open an empty canvas, called an untitled Xcos window.

Palette browser shows all of the available blocks that can be used. It has been nicely categorized as per the functionality. For example, blocks that generate signals/values without any input, fall under the category `Sources`. Similarly, blocks that take signals/values without giving any output are categorized as `Sinks`. This makes finding a particular block very easy, specially when one does not know the name of a block.

The untitled window is the one where one creates the Xcos code/diagram. The relevant blocks have to be dragged and dropped from the palette browser to the untitled window. The blocks are then interconnected and configured as per the simulation, which we will demonstrate next.

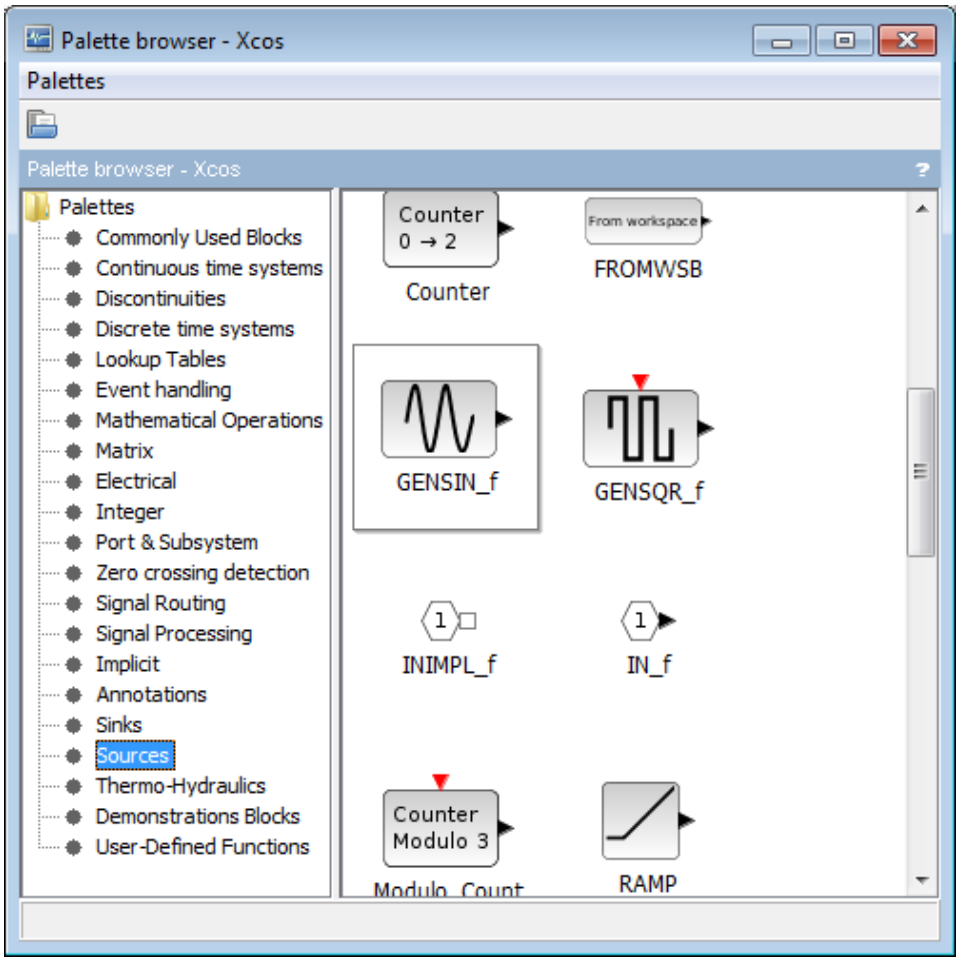


Figure 3.15: Sine generator in palette browser

3.3.2 Use case

Let us build a simple Xcos simulation to plot a sine wave. This simulation requires a sine wave source. It can be found in the `Sources` category as shown in Fig. 3.15. Drag and drop this block in the untitled Xcos window.

Next, we need a block to plot the sine wave. A plotting block can be found in the `Sinks` category as shown in Fig. 3.16. The name of this block is CSCOPE. Drag and drop this block in the untitled Xcos window. On the left-hand side, this block has an input port for data. It is black in colour, which may not be obvious in a black and white printout. The output from the sine block has to be connected to this port. At its top side, the CSCOPE block has another input port, called an

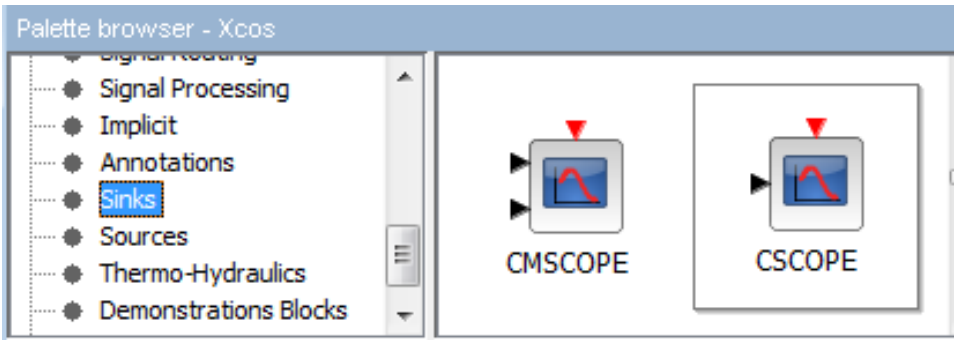


Figure 3.16: CSCOPE block in xcos

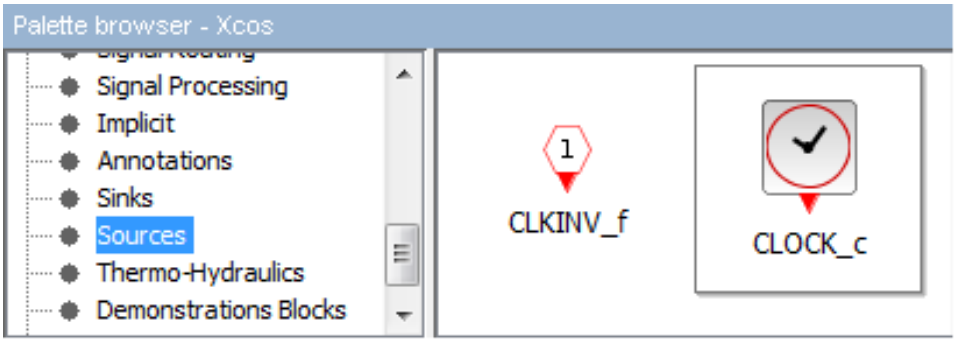


Figure 3.17: CLOCK_c block in xcos

event port. This is red in colour. This port is used to synchronize it with event generating devices.

As the CSCOPE block has an input event port, we need a source that generates events. Hence, the next block that we need is an event generator block and it can be found in the `Sources` category. This is illustrated in figure 3.17. The name of this block is CLOCK_c. Drag and drop this block in the untitled Xcos window.

The next step is to interconnect the blocks together. A black color port can only be connected to another black color port. A black color port cannot be connected to a red color port and vice versa. That is, a data port cannot be connected to an event port. Linking two blocks is a bit crucial and may need a few attempts before one gets comfortable. To link two blocks, first click and hold the left mouse button over the output port of the source block. Without releasing the mouse button, touch the mouse pointer to the input port of the sink block. If a connection is possible there, the port will turn “green”. At this point, release the mouse button and the blocks should get connected. Follow this procedure and complete the connection as shown

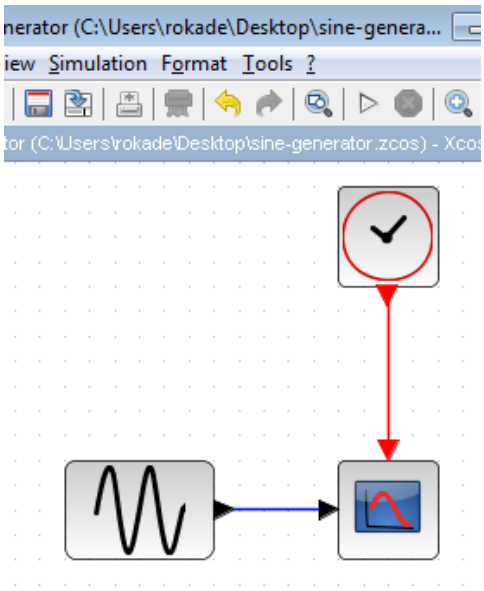


Figure 3.18: Sine generator in Xcos

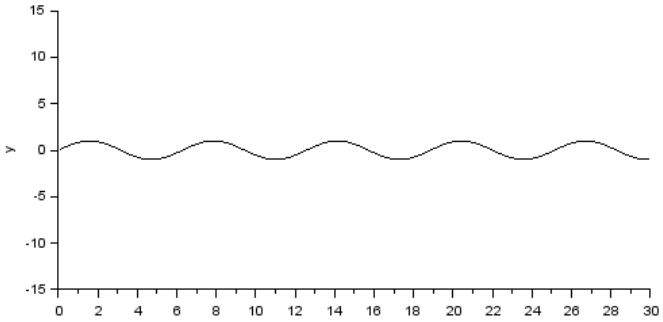


Figure 3.19: Sine generator Xcos output

in Fig. 3.18. Save this file with the name `sine-generator`.

Let us simulate this Xcos code. On the menu bar, click on the `Simulation` menu and choose `Start`. You will get a graphic window with a running sine wave as shown in Fig. 3.19.

This is because we are running the simulation using the default configuration. We would like a stationary plot. If the simulation is still running, go to the Simulation menu and choose Stop. Double click on the CSCOPE block. Its properties window will appear as shown in 3.20. Note the value of the `Refresh period`. It is by default 30. Click on Ok.

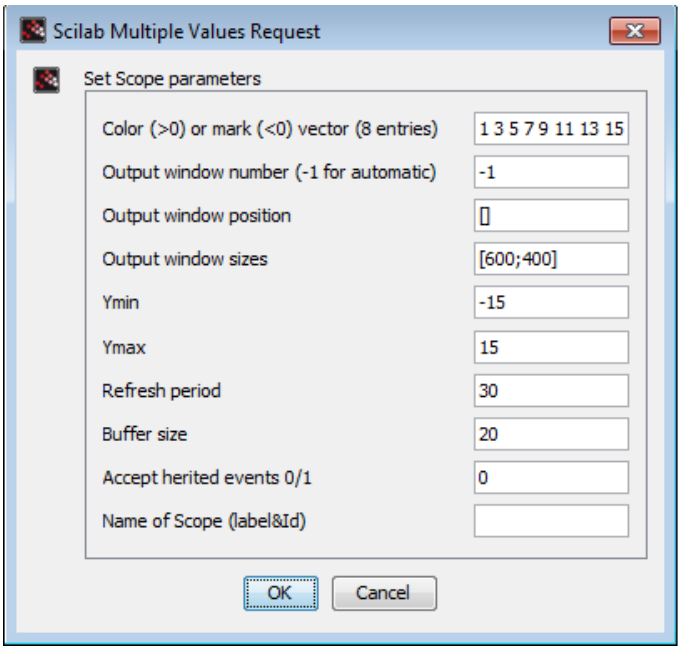


Figure 3.20: CSCOPE configuration window

Next, on the menu bar, click on the `Simulation` menu and choose `Setup`. The `Set Parameters` window will open. The first parameter is `Final integration time`. It decides for how long the simulation will run. Change it to be equal to the `Refresh period` of the CSCOPE block. That is, change it to 30 as shown in Fig. 3.21. Now start the simulation and you will get a static plot. Other parameters of blocks can also be changed. For example, one may want to change the input amplitude/frequency or change the plot scales, etc. All these are left to the reader to explore.

Although we have demonstrated a very basic level of Xcos simulation, this idea can be used for complex processes as well. Using Xcos, it is possible to have user-defined blocks. The user can code the working of the block as a function in Scilab script and then call it from Xcos. It is also possible to create subsystems. One can even read from and write to C binaries. Xcos comes with several pre-defined libraries and hence, it is possible to carry out other kinds of simulation, such as electrical circuit simulation and basic thermo-hydraulic simulation, for example. A detailed explanation and demonstration are beyond the scope of this book.

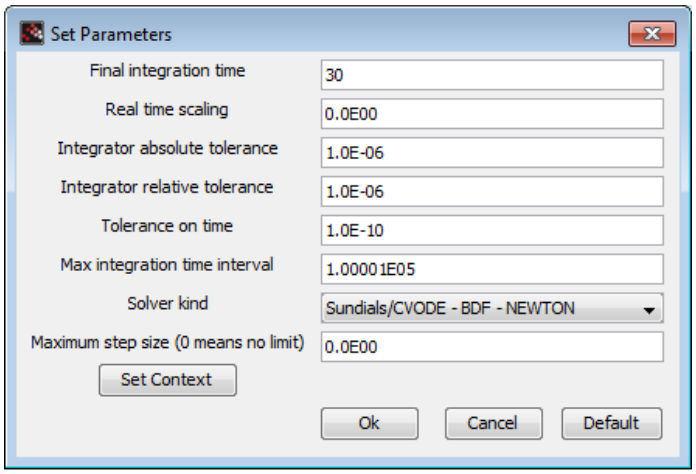


Figure 3.21: Simulation setup window

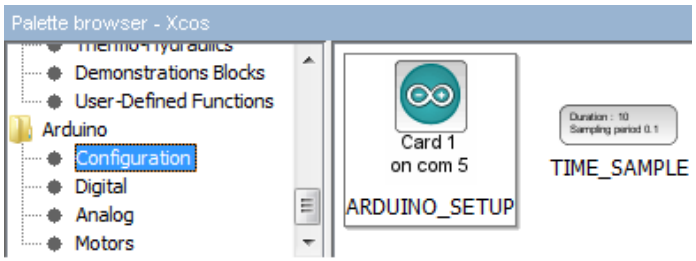


Figure 3.22: Palette browser showing Arduino blocks

3.3.3 Xcos-Arduino

The Scilab Arduino toolbox not only provides functions to be used in Scilab scripts but also provides new Arduino-specific blocks. As shown in Fig. 3.22 new Arduino blocks are now available for use. Similar to the categorization of the functions, the Xcos blocks are also categorized as configuration, digital, analog and motors. Again, it is possible to conduct the experiments only using Xcos. Xcos codes for every experiment are provided throughout the book. The Arduino blocks can be easily connected to Xcos native blocks. A detailed block help for every block can be sought by right-clicking on the block and choosing “Block help”. This is illustrated in Fig. 3.23.

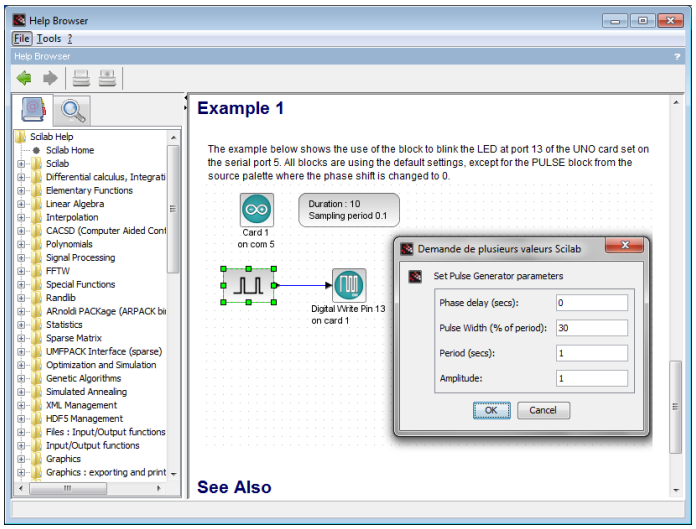


Figure 3.23: Xcos block help

3.4 Python

Python is a general-purpose, high-level, remarkably powerful dynamic programming language that is used in a wide variety of application domains. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed [15].

3.4.1 Downloading and installing on Windows

This book uses Python 3 for demonstrating the experiments, both on Windows and Linux. Since Python uses indentation to indicate a block of code, the users are advised to install a programmer text editor like Atom. This editor will allow the readers to modify the Python scripts on their machines if they want to. Starting from download, we shall go through the steps to set up Python 3 on Windows OS:

1. Visit the URL `https://www.python.org/`. At the top of the page, locate the Downloads tab and click on it. At the time of writing this book (as of

Figure 3.24: Installing Python 3 on Windows

21 April 2021), Python 3.9.4 is the latest version. Click on Download Python 3.9.4 to download the executable file. The readers may want to download the other versions of Python 3. However, we recommend the installation of Python 3.5 or above. It may be noted that some of the Python 3 versions cannot be used on Windows 7 or earlier.

2. Locate the executable file and double-click on it to begin the installation. Python 3.9.4 Setup window appears, as shown in Fig. 3.24. In this window, check the box which says, Add Python 3.9 to PATH and click on Install now.

Once the installation is finished, Python 3.9 App can be launched from the Start menu. In this book, we will use the Command Prompt to execute the Python scripts. Please note that a Python script has .py as its extension. Carry out the steps given below to execute a Python script from the Command Prompt:

1. Launch the Command Prompt. Press the Windows key+R together. A window, as shown in Fig. 3.25 appears. In the text box adjacent to `Open`, type `cmd`, and press Enter. The Command Prompt, as shown in Fig. 3.26 appears. By default, it points to the home directory.

2. Now, we will check whether Python 3.9 was installed successfully or not. In the Command Prompt, type `py --version` and press Enter. If this step displays Python 3.9.4 in the following line, the installation was successful.

3. Using the `cd` command, navigate to the directory where your Python script is located. Assuming that your Command Prompt points to the home directory,

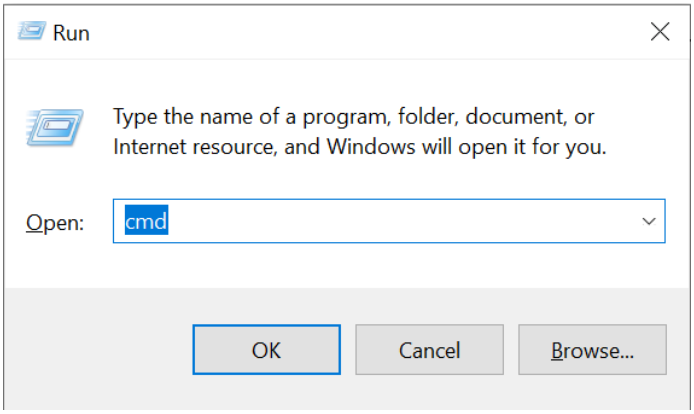


Figure 3.25: Launching the Command Prompt on Windows

Figure 3.26: Command Prompt on Windows

and you want to navigate to the folder Origin on Desktop, execute the following command: `cd Desktop\Origin`
It may be noted that a backslash (\) has been used between Desktop and Origin.

4. To view the contents of this folder, type `dir` and press Enter.

5. Suppose you have a Python script named `FILENAME.py` in this folder. To execute this script, type `python FILENAME.py` and press Enter. The required output will be displayed in the Command Prompt itself. We don't expect the readers to run the command `python FILENAME.py` at this instant. This command will be helpful while running the Python scripts in the upcoming sections and chapters.

6. To exit the Command Prompt, type `exit` and press Enter.

Apart from Python, we need to install Python Serial Port Extension, also known as pyserial [16]. To do so, carry out the steps given below:

1. Launch the Command Prompt, as shown in Fig. 3.25.

2. First, we need to make sure we have pip available. In the Command Prompt, execute the following command: `py -m pip --version`
   This step should display an output with the version of pip installed.

3. Now, install pyserial. In the Command Prompt, execute the following command: `py -m pip install pyserial`

4. We will verify whether the pyserial package was installed successfully or not. In the Command Prompt, execute the following command: `pip show pyserial`
   It should show the name, version, etc., of the package.

5. To exit the Command Prompt, type `exit` and press Enter.

3.4.2 Downloading and installing on GNU/Linux Ubuntu

On Linux, we can install Python from the terminal. Please ensure that you are connected to the Internet. To install Python 3.5, carry out the steps given below:

1. Open a terminal by pressing the Alt+Ctrl+T keys together.

2. Update the system by executing the command `sudo apt-get update`

3. Install Python3.5 by executing the command given below:
   `sudo apt-get install python3.5`

4. Now, we will check whether Python was installed successfully or not. In the terminal, type `py --version` and press Enter. If this step displays Python 3.8.4 in the following line, the installation was successful.

Once the installation is finished, Python 3 can be launched from the terminal. In this book, we will use the Linux terminal to execute the Python scripts. Please note that a Python script has .py as its extension. Carry out the steps given below to execute a Python script from the terminal:

1. Open a terminal by pressing the Alt+Ctrl+T keys together.

2. Using `cd` command, navigate to the directory where the Python script is located.

3. Suppose you have a Python script named `FILENAME.py`. To execute this script, type `python3 FILENAME.py` and press Enter. The required output will be displayed in the terminal itself. We don't expect the readers to run the command `python3 FILENAME.py` at this instant. This command will be helpful while running the Python scripts in the upcoming sections and chapters.

Apart from Python, we need to install Python Serial Port Extension, also known as pyserial [16]. To do so, carry out the steps given below:

1. Open a terminal by pressing the Alt+Ctrl+T keys together.
2. First, we need to install pip. In the terminal, execute the following command: `sudo apt-get install python3-pip`
3. Now, install pyserial. In the terminal, execute the following command: `pip3 install pyserial`
4. We will verify whether the pyserial package was installed successfully or not. In the terminal, execute the following command: `pip3 show pyserial` It should show the name, version, etc., of the package.

3.4.3 Python-Arduino toolbox

Python, by default, does not have the capability to connect to Arduino. All such add-on functionalities are added to Python using packages. The Python-Arduino toolbox can be found inside the `Origin/tools/python` directory, see Footnote 2 on page 2. This toolbox is compatible for both of the operating systems: Windows and Linux. The Python scripts (or 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.

In this book, we have created a package named "Arduino" in Python 3. This package is available at `Origin/tools/python`. This package makes use of the functions available in pyserial [16] to establish serial communication with Arduino. In this package, we have added functions required to run various experiments on Arduino Uno. Using this basic set of functions, the user can define other functions to operate upon the Arduino. Please note that the "Arduino" package and the FLOSS firmware given in Arduino Code 3.1 are required to run the experiments.

Now, we will see how to import (or load) the "Arduino" package inside a Python script to run various experiments provided in this book. In a Python script, add `from Arduino.Arduino import Arduino` at the top of the script. When we add `from Arduino.Arduino import Arduino` in a script, the function "from" searches for "Arduino" only in that directory where our script is saved. That's why all the scripts in Python must be saved in a folder that contains the "Arduino" package. In this book, "Arduino" package has been saved in the folder where the Python scripts for each chapter are available. For the sake of convenience, we have added `from Arduino.Arduino import Arduino` in all the Python scripts provided in this book. To run a particular experiment, one can follow the steps as given in Sec. 3.4.1 or Sec. 3.4.2.

3.4.4 Firmware

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

Python Code 3.1 A Python script to check whether the firmware is properly installed or not. Available at `Origin/tools/python/test_firmware.py`, see Footnote 2 on page 2. Execute this script by following the steps given in Sec. 3.4.1 or Sec. 3.4.2. If the execution is successful, you should expect three "ok" messages.

```
1  import os
2  import sys
3  cwd = os.getcwd()
4  (setpath,Examples) = os.path.split(cwd)
5  sys.path.append(setpath)
6
7  from Arduino.Arduino import Arduino
8  from time import sleep
9
10 class TEST_FIRMWARE:
11     def __init__(self, baudrate):
12         self.baudrate = baudrate
13         self.setup()
14         self.run()
15         self.exit()
16
17     def setup(self):
18         self.obj_arduino = Arduino()
19         self.port = self.obj_arduino.locateport()
20         self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22     def run(self):
23       self.obj_arduino.checkfirmware()
24
25     def exit(self):
26         self.obj_arduino.close_serial()
27
28 def main():
29     obj_led = TEST_FIRMWARE(115200)
30
31 if __name__ == '__main__':
32     main()
```

3.5 Julia

Julia is a high-level, high-performance dynamic programming language for technical computing, with a syntax familiar to users of other technical computing environments [17]. While it is a general-purpose language and can be used to write any application, many of its features are well suited for numerical analysis and computational science. Julia provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

3.5.1 Downloading and installing on Windows

This book uses Julia 1.6.0 for demonstrating the experiments, both on Windows and Linux. Julia does not use indentation to indicate a block of code, unlike Python. However, the users are advised to install a programmer text editor like Atom. This editor will allow the readers to modify the Julia source files on their machines if they want to. Alternatively, one can also use Notepad (on Windows) or gedit (on Linux Ubuntu) to edit Julia source files. Starting from download, we shall go through the steps to set up Julia 1.6.0 on Windows OS:

1. Visit the URL `https://julialang.org/`. At the top of the page, locate the Download tab and click on it. From the Current stable release, download the required Julia binaries (32 or 64-bit) for Windows, as shown in Fig. 3.27. At the time of writing this book, the Current stable release refers to Julia 1.6.0 as of March 24, 2021, as shown in Fig. 3.27. In this book, we will perform experiments with a 64-bit installation of Julia 1.6.0.

2. Locate the executable file. Right-click on it and hit Run as administrator to begin the installation. After selecting the Installation Directory, a window named Select Additional Tasks appears as shown in Fig. 3.28. In this window, check the box which says, Add Julia to PATH and click on Next to continue the installation.

Once the installation is finished, Julia 1.6.0 App can be launched either from the Start menu or from the Command Prompt. In this book, we will use the Command Prompt to execute the Julia source files. Please note that a Julia source file has .jl as its extension. Carry out the steps given below to execute a Julia source file from the Command Prompt:

1. Launch the Command Prompt. Press the Windows key+R together. A window, as shown in Fig. 3.29 appears. In the text box adjacent to Open, type `cmd`, and press Enter. The Command Prompt, as shown in Fig. 3.30 appears. By default, it points to the home directory.

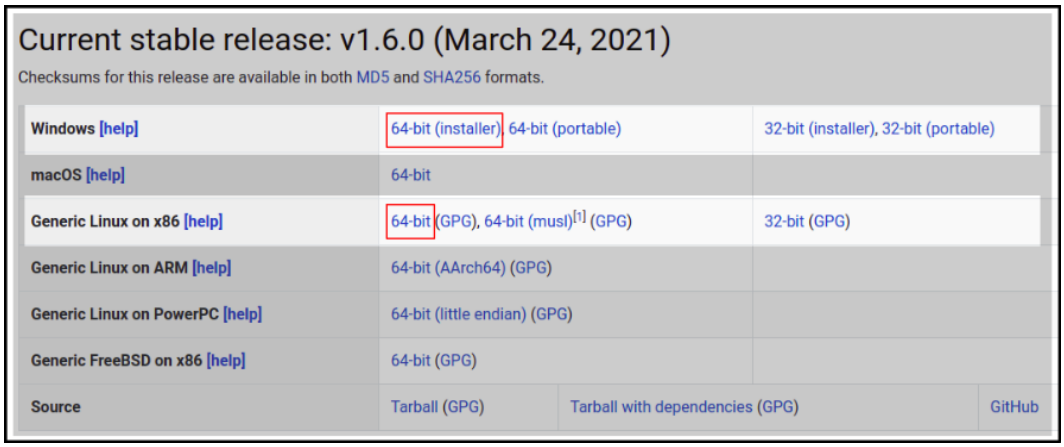


Figure 3.27: Julia's website to download 64-bit Windows/Linux binaries

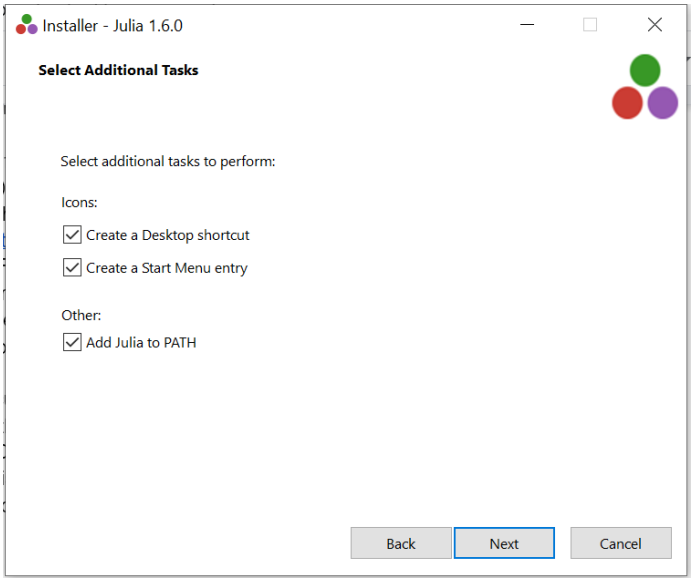


Figure 3.28: Installing Julia 1.6.0 on Windows

2. Now, we will check whether Julia 1.6.0 was installed successfully or not. In the Command Prompt, type `julia --version` and press Enter. If this step displays julia version 1.6.0 in the following line, the installation was successful.

3. Using the `cd` command, navigate to the directory where your Julia source file is located. Assuming that your Command Prompt points to the home directory,

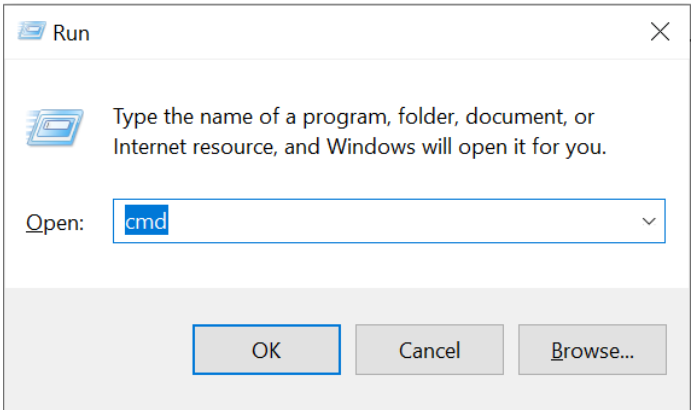


Figure 3.29: Launching the Command Prompt on Windows

Figure 3.30: Command Prompt on Windows

and you want to navigate to the folder Origin on Desktop, execute the following command: `cd Desktop\Origin`
It may be noted that a backslash (\) has been used between Desktop and Origin.

4. To view the contents of this folder, type `dir` and press Enter.

5. Suppose you have a Julia source file named `FILENAME.jl` in this folder. To execute this script, type `julia FILENAME.jl` and press Enter. The required output will be displayed in the Command Prompt itself. We don't expect the readers to run the command `julia FILENAME.jl` at this instant. This command will be helpful while running the Python scripts in the upcoming sections and chapters.

6. To exit the Command Prompt, type `exit` and press Enter.

Apart from Julia, we need to install the SerialPorts [18] package in Julia. This package will be required to establish serial communication with Arduino boards. Please make sure that you are connected to the Internet. To install the package, we will use `Pkg`. It is Julia's built-in package manager and handles operations such

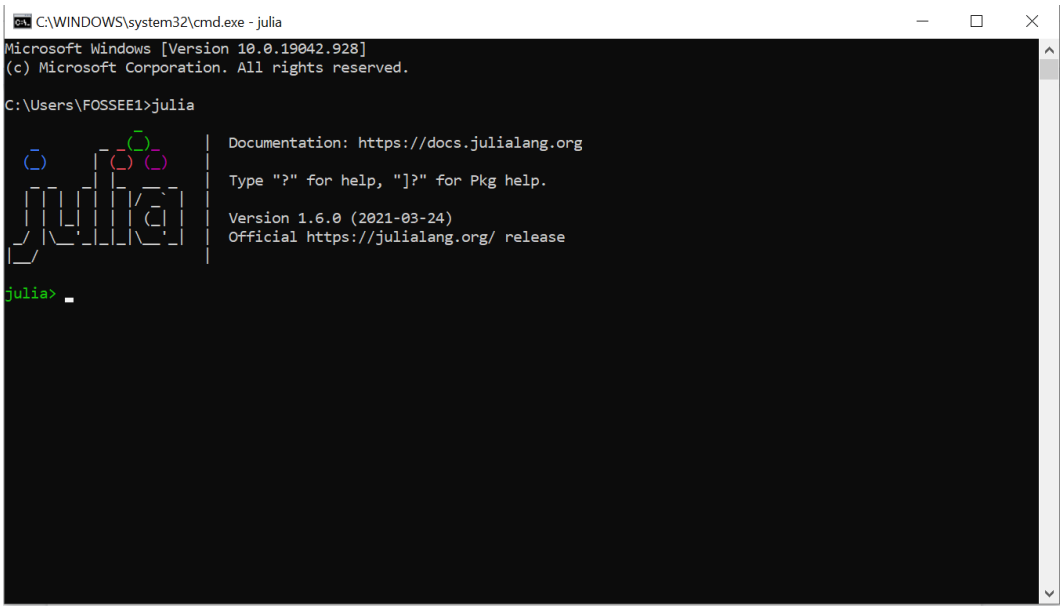


Figure 3.31: Windows command prompt to launch Julia REPL

as installing, updating and removing packages in Julia. To install the SerialPorts package, carry out the steps given below:

1. Launch the Command Prompt, as shown in Fig. 3.29.
2. In the Command Prompt, type `julia` and press Enter. It should launch Julia in an interactive session (also known as a read-eval-print loop or "REPL"), as shown in Fig. 3.31. When run in interactive mode, Julia displays a banner and prompts the user for input. By default, Julia REPL appears in Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`, as shown in Fig. 3.31. It is here that you can enter Julia expressions. Hitting return or Enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.
3. Now, we need to enter the `Pkg` REPL in Julia. From the Julia REPL, press `]` to enter the `Pkg` REPL. The moment you press `]`, you enter `Pkg` REPL, as shown in Fig. 3.32.
4. In `Pkg` REPL, type `add SerialPorts` and press Enter. It might take a few seconds/minutes to get this package installed. Once it is installed, `Pkg` REPL should show a message, like "3 dependencies successfully precompiled in 9 seconds (4 already precompiled)."

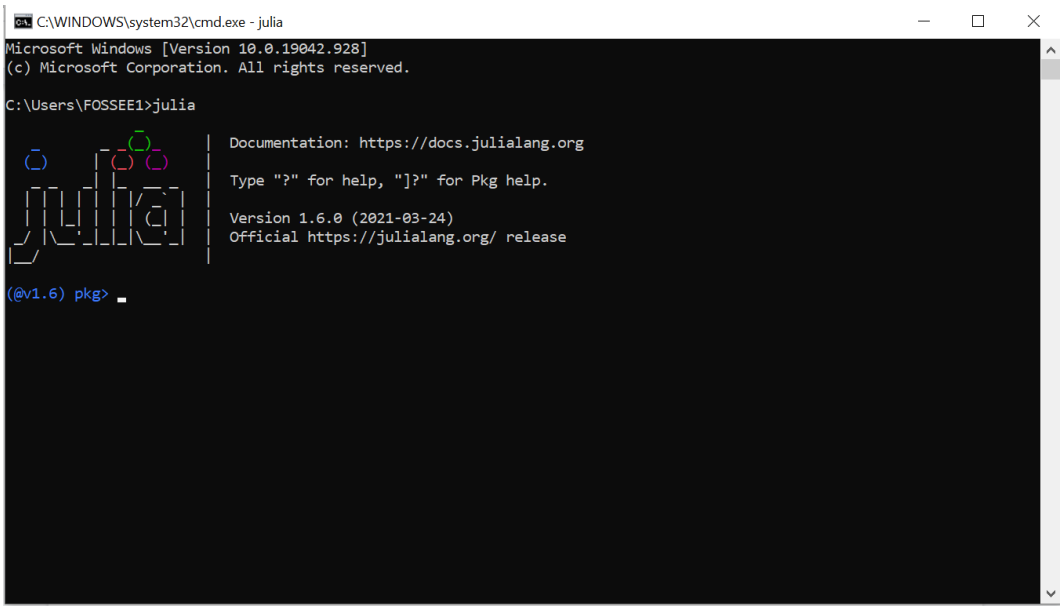


Figure 3.32: Windows command prompt to enter Pkg REPL in Julia

We can also check the status of this package to verify whether it has been installed successfully or not. For this, we need to get back to Julia REPL. Inside `Pkg` REPL, press backspace. The moment you press backspace, you get back to Julia REPL, as shown in Fig. 3.31. Now, type `using Pkg` and press Enter. This command will not generate any output. Now type `Pkg.status()` and press Enter. It should display the list of packages installed in Julia's environment. Please make sure that SerialPorts is present in the list of packages being shown. To exit the interactive session, type CTRL+D or type `exit()`.

3.5.2 Downloading and installing GNU/Linux Ubuntu

We will now explain the installation of Julia 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. This book uses Julia 1.6.0. To install it, carry out the steps given below:

1. First, update your system. Open the Terminal. Type `sudo apt-get update` and press Enter.

2. Find out your operating system support for 64-bit instructions. Open the

Terminal. Type `uname -m` and press Enter. If it returns "x86_64", then your computer has 64-bit operating system.

3. Visit the URL `https://julialang.org/`. At the top of the page, locate the Download tab and click on it. From the Current stable release, download the required Linux binaries (32 or 64-bit) for Generic Linux on x86, as shown in Fig. 3.27. At the time of writing this book, the Current stable release refers to Julia 1.6.0 as of March 24, 2021, as shown in Fig. 3.27. In this book, we will perform experiments with a 64-bit installation of Julia v1.6.0.

4. Locate the executable (tar.gz) file. Assuming that you have downloaded the tar file in `Downloads` directory, perform the following steps on the Terminal:

```
cd ~/Downloads
tar -xvzf julia-1.6.0-linux-x86_64.tar.gz
sudo cp -r julia-1.6.0 /opt/
```

5. Finally, create a symbolic link to `julia` inside the `/usr/local/bin` folder. In the same Terminal session from the previous step, issue the following command:
`sudo ln -s /opt/julia-1.6.0/bin/julia /usr/local/bin/julia`

Julia is now installed and can be invoked from the Terminal. There are two modes in which Julia source files can be executed: Interactive mode and Non-interactive mode. In this book, we will execute source files in the latter mode i.e., from the Terminal. The readers are encouraged to explore the former mode on their own. Please note that a Julia source file has .jl as its extension. Carry out the steps given below to execute a Julia source file from the Terminal:

1. Open a Terminal by pressing Ctrl+Alt+T keys together.

2. Now, we will check whether Julia 1.6.0 was installed successfully or not. In the Terminal, type `julia --version` and press Enter. If this step displays `julia version 1.6.0` in the following line, the installation was successful.

3. Using the `cd` command, navigate to the directory where your Julia source file is located. Assuming that your Terminal points to the home directory, and you want to navigate to the folder Origin on Desktop, execute the following command: `cd Desktop/Origin/`

4. Suppose you have a Julia source file named `FILENAME.jl` in this folder. To execute this script, type `julia FILENAME.jl` and press Enter. The required output will be displayed in the Terminal itself. We don't expect the readers

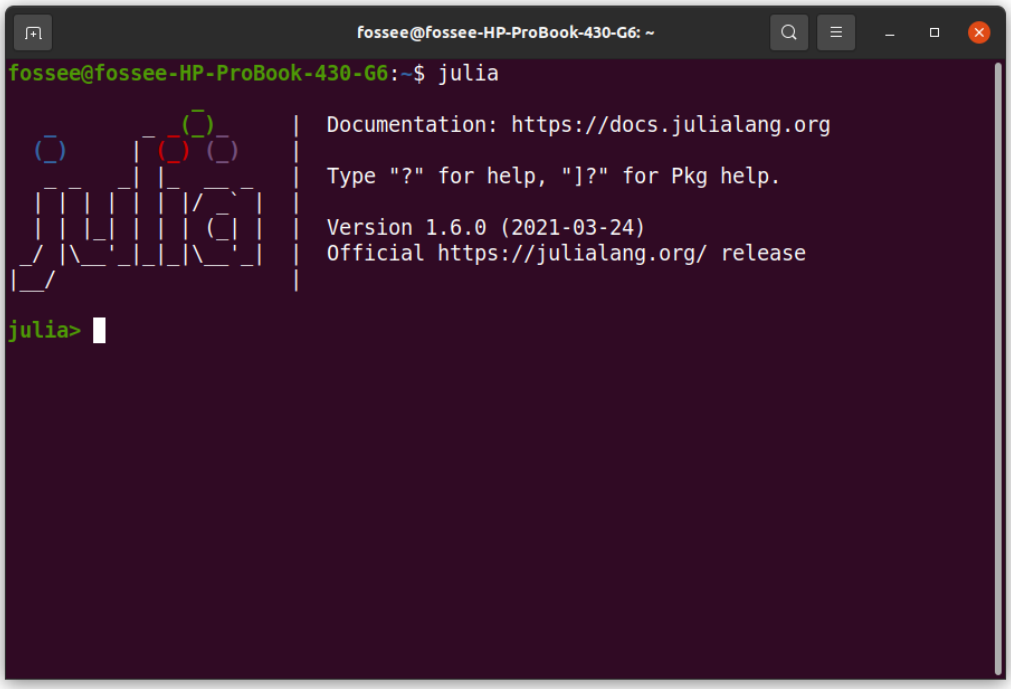


Figure 3.33: Linux terminal to launch Julia REPL

to run the command `julia FILENAME.jl` at this instant. This command will be helpful while running the Python scripts in the upcoming sections and chapters.

Now, we will install a package named SerialPorts [18]. This package will be required to establish serial communication with Arduino boards. Please ensure that you are connected to the Internet. To install the package, we will use `Pkg`. It is Julia's built-in package manager and handles operations such as installing, updating, and removing packages in Julia. For installing LibSerialPort, carry out the steps given below:

1. Open a Terminal by pressing Ctrl+Alt+T keys together. Type `julia` and press Enter. It should launch Julia in an interactive session (also known as a read-eval-print loop or "REPL"), as shown in Fig. 3.33. When run in interactive mode, Julia displays a banner and prompts the user for input. By default, Julia REPL appears in Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`, as shown in Fig. 3.33. It is here

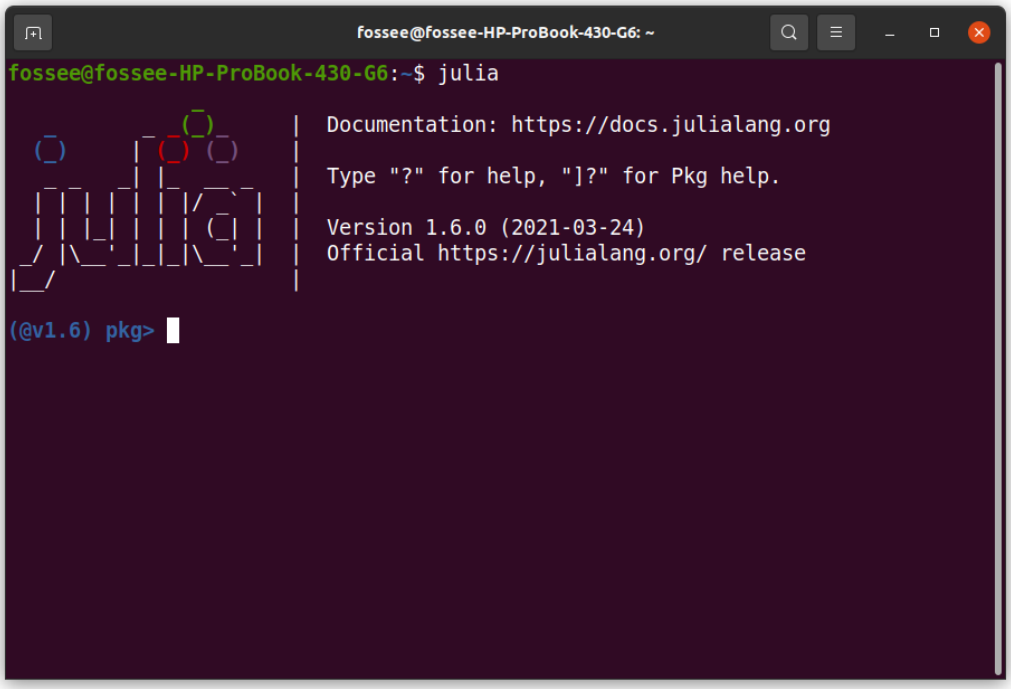


Figure 3.34: Linux terminal to enter Pkg REPL in Julia

that you can enter Julia expressions. Hitting return or Enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

2. From the Julia REPL, press `]` to enter the `Pkg` REPL. The moment you press `]`, you enter `Pkg` REPL, as shown in Fig. 3.34.

3. In `Pkg` REPL, type `add SerialPorts` and press Enter. It might take a few seconds to get this package installed. Once it is installed, `Pkg` REPL should show a message, like "5 dependencies successfully precompiled in 12 seconds."

We can also check the status of this package to verify whether it has been installed successfully or not. For this, we need to get back to Julia REPL. Inside `Pkg` REPL, press backspace. The moment you press backspace, you get back to Julia REPL, as shown in Fig. 3.33. Now, type `using Pkg` and press Enter. This command will not generate any output. Now type `Pkg.status()` and press Enter. It should display the list of packages installed in Julia's environment. Please make sure that

SerialPorts is present in the list of packages being shown. To exit the interactive session, type CTRL+D or type `exit()`.

3.5.3 Julia-Arduino toolbox

Julia, by default, does not have the capability to connect to Arduino. All such add-on functionalities are added to Julia using toolboxes. The Julia-Arduino toolbox can be found inside the `Origin/tools/julia` directory, see Footnote 2 on page 2. This toolbox is compatible for both of the operating systems: Windows and Linux. The Julia source files (or 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.

In this book, we have created a module named "ArduinoTools" in Julia. This module is available at `Origin/tools/julia`. This module makes use of the functions available in the SerialPorts package to establish serial communication with Arduino. In this module, we have added functions required to run various experiments on Arduino Uno. Using this basic set of functions, the user can define other functions to operate upon the Arduino.

Now, we will see how to import (or load) the module named "ArduinoTools.jl" inside a Julia source file to run various experiments provided in this book. In a Julia source file, add `include("ArduinoTools.jl")` at the top of the file. When we add `include("ArduinoTools.jl")` in a source file, the function "include" searches for "ArduinoTools.jl" only in that directory where our source file is saved. That's why all the source files in Julia must be saved in a folder that contains the file "ArduinoTools.jl." In this book, "ArduinoTools.jl" has been saved in the folder where the Julia source files for each chapter are available. For the sake of convenience, we have added `include("ArduinoTools.jl")` in all the Julia source files provided in this book. To run a particular experiment, one can follow the steps as given in Sec. 3.5.1 and Sec. 3.5.2.

3.5.4 Firmware

We have provided a Julia source file (code) to check whether the firmware has been properly installed. That file is listed below. Please ensure that you have uploaded the FLOSS firmware given in Arduino Code 3.1 on Arduino Uno.

Julia Code 3.1 A Julia source file (code) to check whether the firmware is properly installed or not. Available at `Origin/tools/julia/test_firmware.jl`, see Footnote 2 on page 2. Execute this source file by following the steps given in Sec. 3.5.1 and Sec. 3.5.2. If the execution is successful, you should expect three "ok" messages.

```
using SerialPorts
include("ArduinoTools.jl")

h = ArduinoTools.connectBoard(115200)

for i = 1:3
  write(h,"v")
  s = read(h, 2)
  println(s)
end

close(h)
```

3.6 OpenModelica

OpenModelica is a free and open-source environment based on the Modelica modeling language for simulating, optimizing, and analyzing complex dynamic systems [19]. 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`.

3.6.1 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.

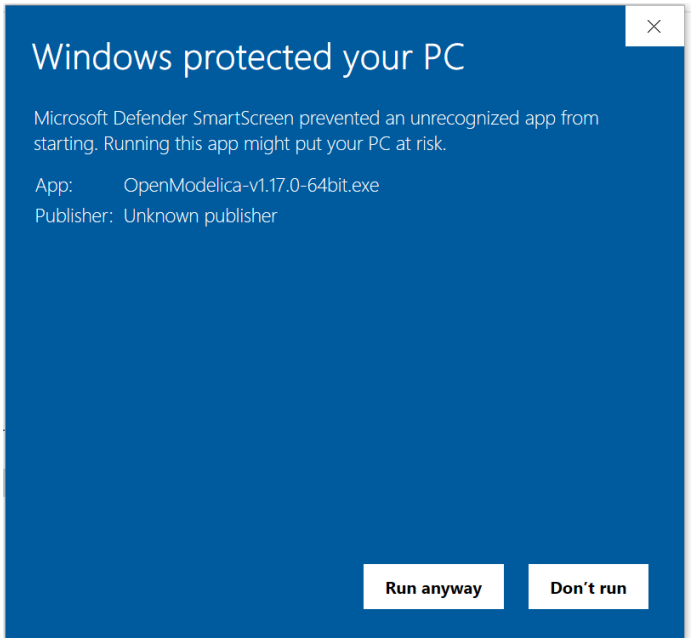


Figure 3.35: Allowing Microsoft Defender to run the executable file

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. 3.35 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. 3.36. 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. 3.6.3.

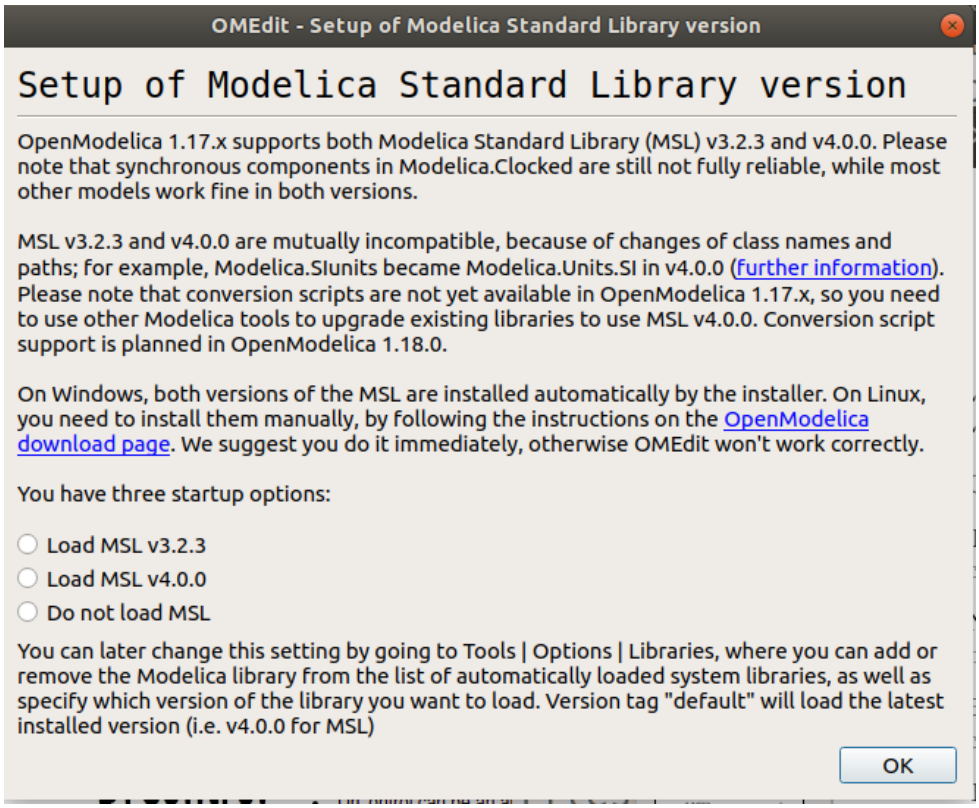


Figure 3.36: Setup of Modelica Standard Library version

3.6.2 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. 3.36. 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. 3.6.3.

3.6.3 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. 3.37. In the bottom right of Fig. 3.37, 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 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 [19].

In the left of Fig. 3.37, 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. 3.37. These default libraries might not be visible if you have not set up the Modelica Standard Library version, as given in Fig. 3.36.

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. 3.38. 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. 3.37. 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. 3.39. 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. 3.40. By default, OMEdit opens any

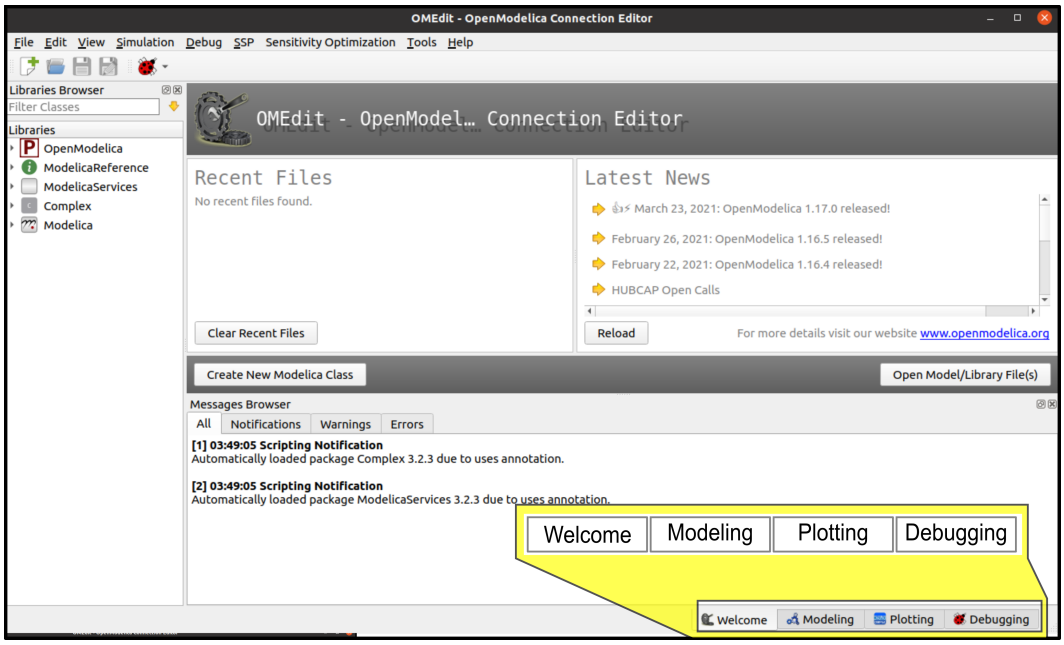


Figure 3.37: User Interface of OMEdit

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. 3.40. In Text view, the code is now visible, as shown in Fig. 3.41. 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. 3.42. 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. 3.43 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. 3.43, OMEdit displays the message that "The Simulation finished successfully". Had there been any error in simulating the model, we would not have received this message.

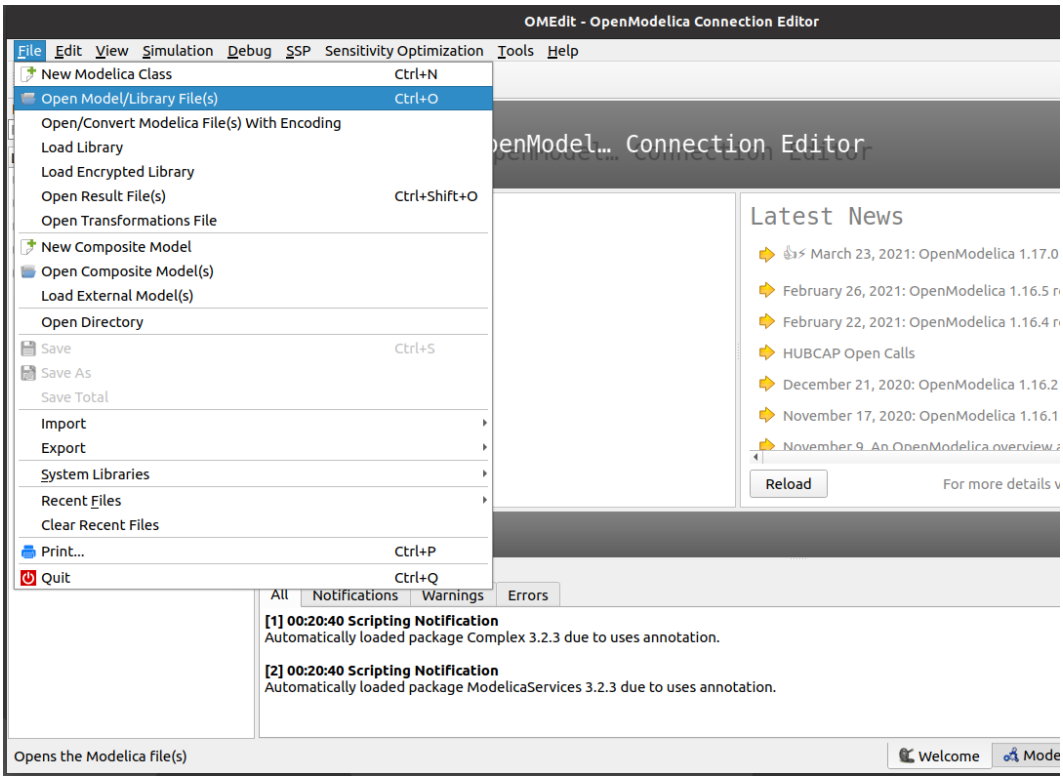


Figure 3.38: Opening a model in OMEdit

3.6.4 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 3.6.2.

2. After launching, we have to load OpenModelica-Arduino toolbox. To do so,

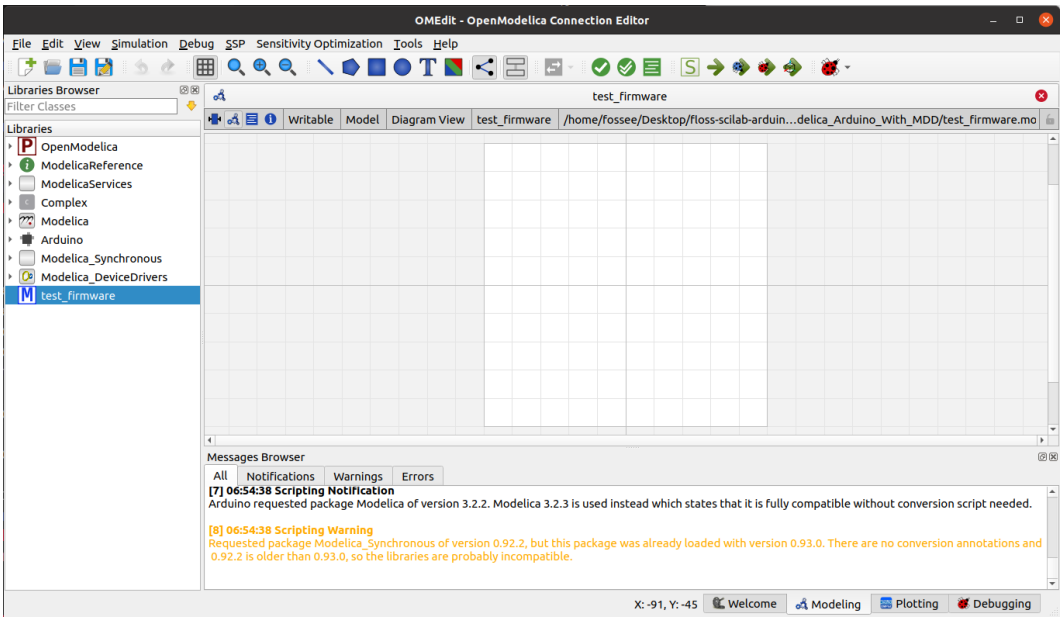


Figure 3.39: Opening a model in diagram view in OMEdit

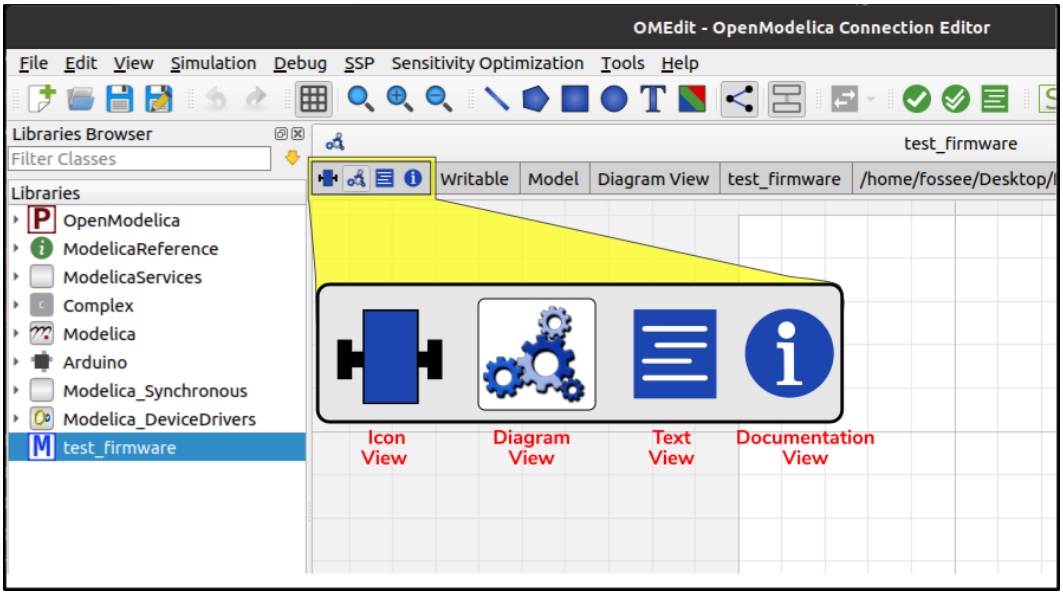


Figure 3.40: Different views of a model in OMEdit

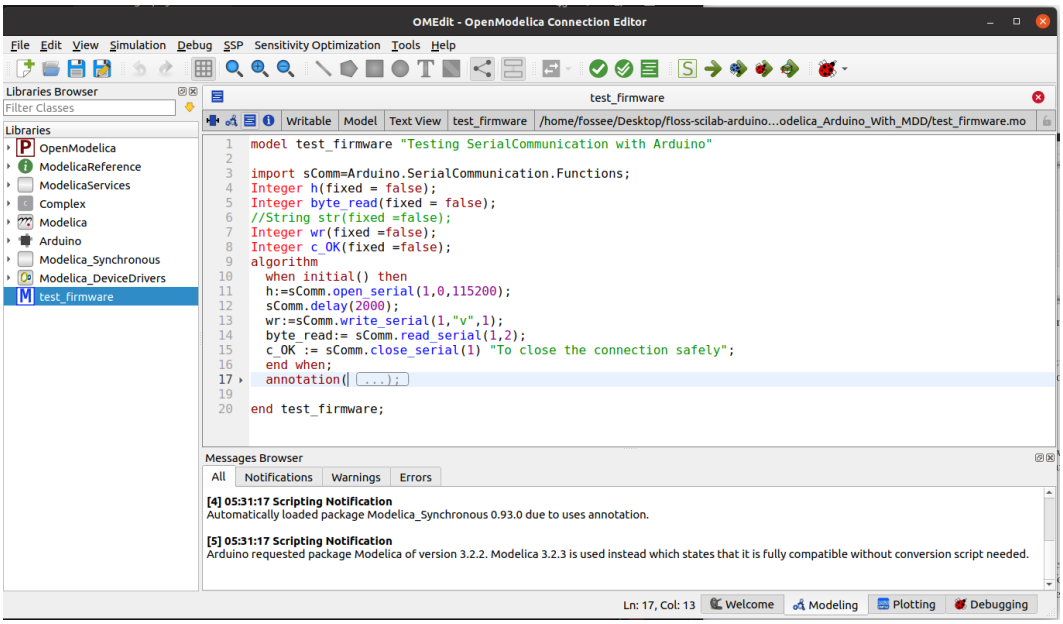


Figure 3.41: Opening a model in text view in OMEdit

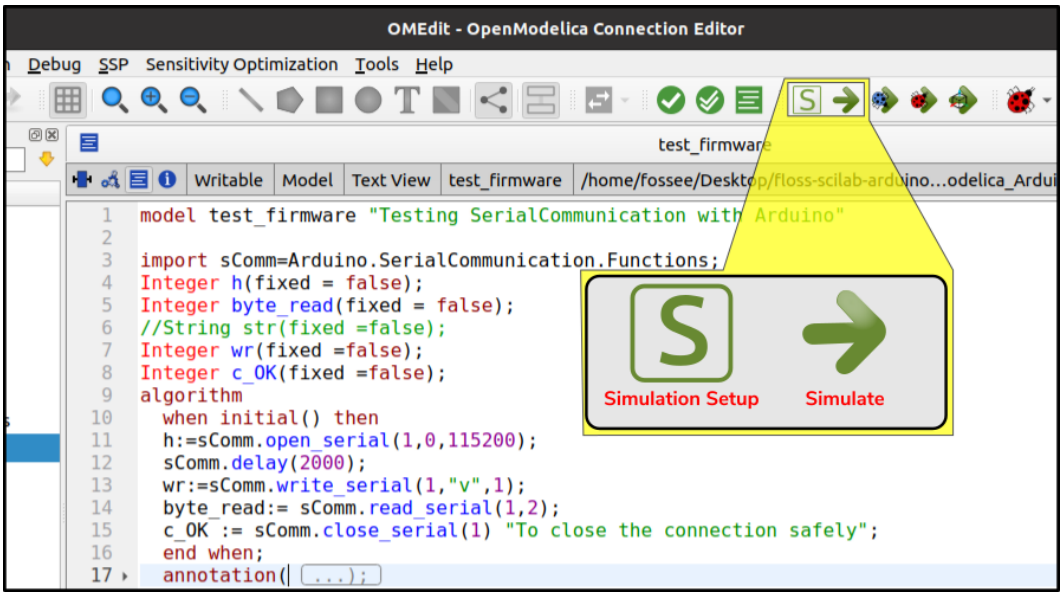


Figure 3.42: Simulating a model in OMEdit

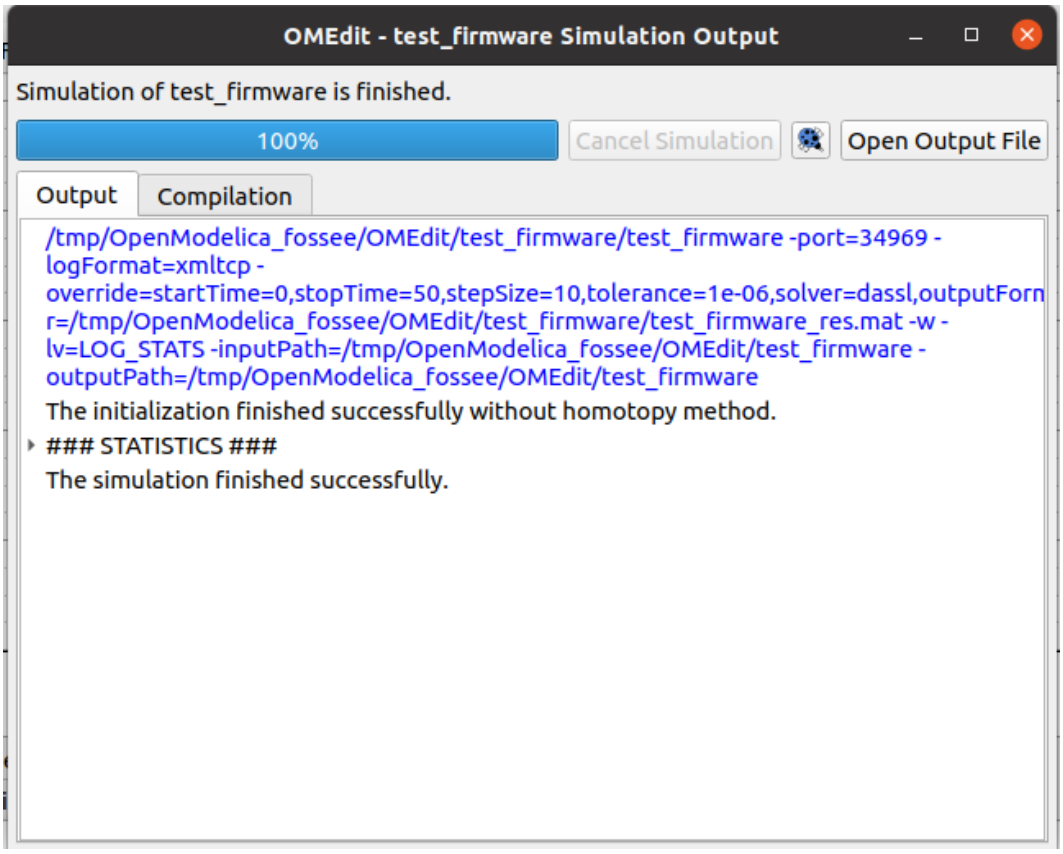


Figure 3.43: Output window of OMEdit

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. 3.38.

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. 3.44.

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. 3.6.3.

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.

3.6.5 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.

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. 3.6.4. Simulate this code/model by following the steps given in Sec. 3.6.3. If the simulation is successful, you should expect an output window in OMEdit as shown in Fig. 3.43.

```
model test_firmware "Testing SerialCommunication with Arduino"

import sComm=Arduino.SerialCommunication.Functions;
Integer h(fixed = false);
Integer byte_read(fixed = false);
//String str(fixed =false);
Integer wr(fixed =false);
Integer c_OK(fixed =false);
algorithm
  when initial() then
  h:=sComm.open_serial(1,2,115200);
  sComm.delay(2000);
  wr:=sComm.write_serial(1,"v",1);
  byte_read:= sComm.read_serial(1,2);
  c_OK := sComm.close_serial(1) "To close the connection safely";
  end when;
```

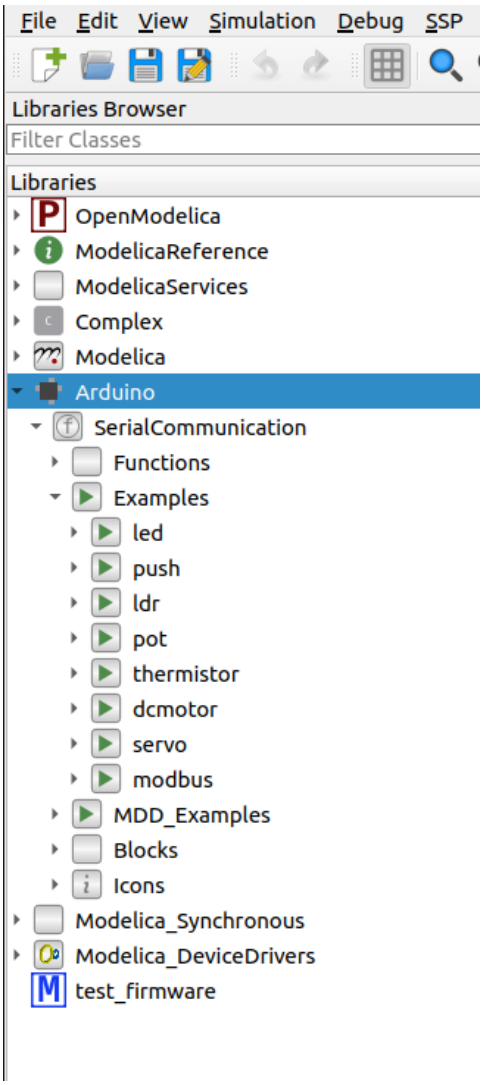


Figure 3.44: Examples provided in the OpenModelica-Arduino toolbox

```
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, Scilab scripts, Scilab Xcos, Python, Julia, and OpenModelica. 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

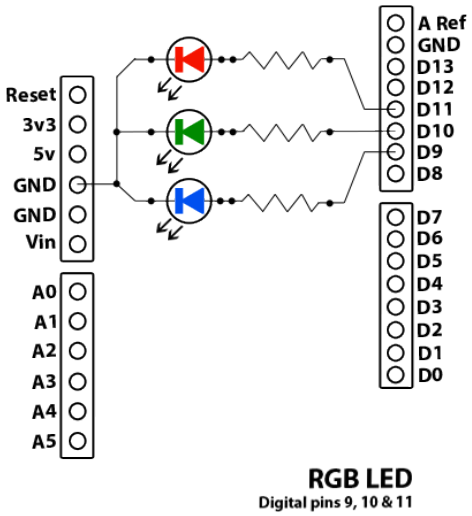


Figure 4.2: Internal connection diagram for the RGB LED on the shield

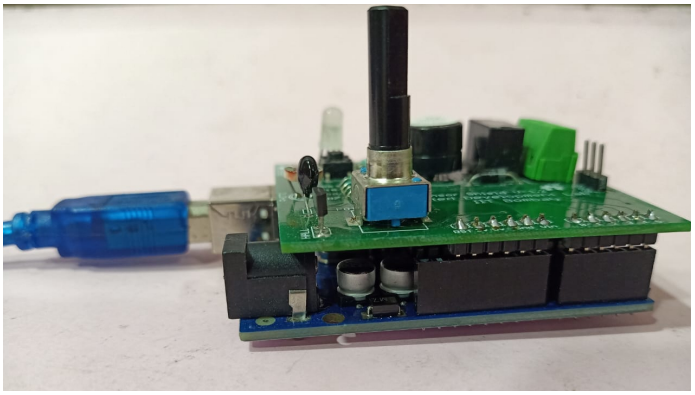


Figure 4.3: Connecting Arduino Uno and shield

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

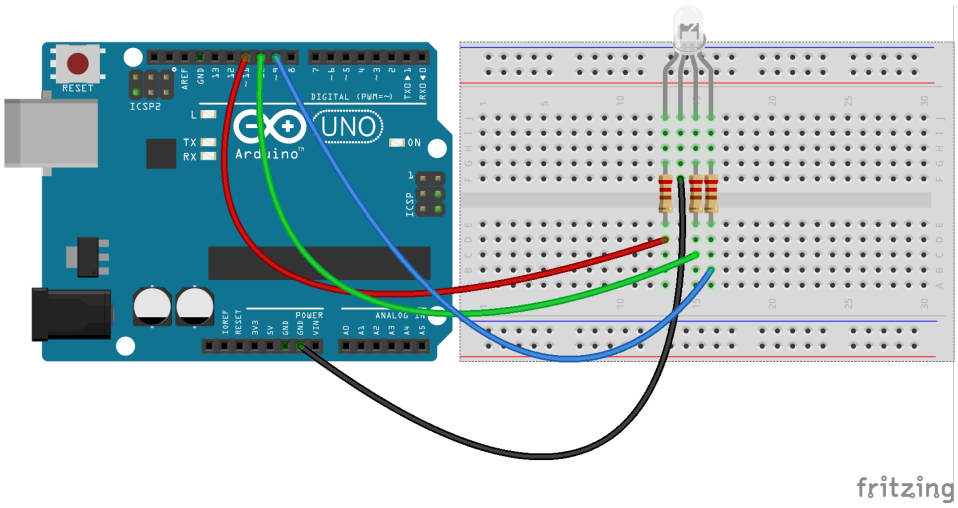


Figure 4.4: An RGB LED with Arduino Uno using a breadboard

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.

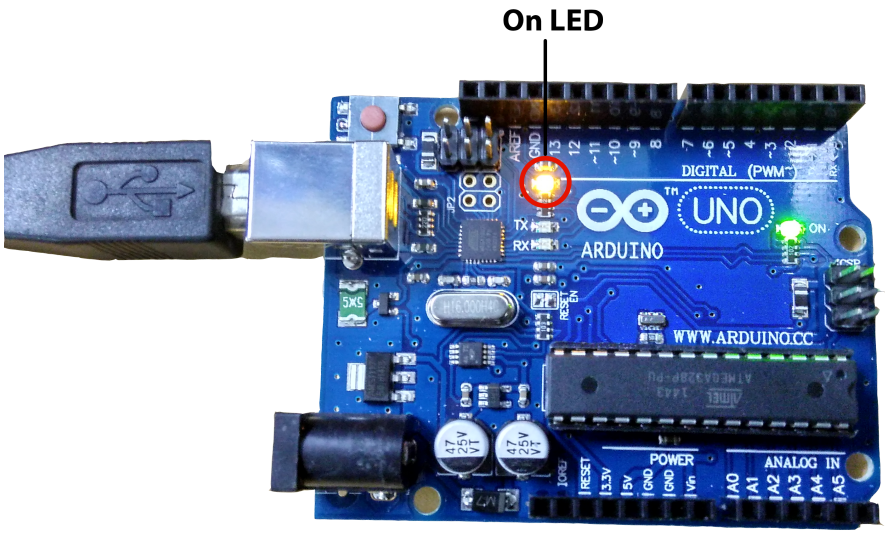


Figure 4.5: LED experiments directly on Arduino Uno board, without the shield

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 Scilab

4.4.1 Lighting the LED

In this section, we discuss how to carry out the experiments of the previous section from Scilab. 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 Scilab Code 4.1. It begins with a command of the form

   ```
   ok = 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 = 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, error(Error Message in Quotes);
   ```

   It checks if `ok = 0`. If not, it flashes an error message and terminates. This line gets implemented in the following way in Scilab Code 4.1.

   ```
   1 if ok ~= 0, error('Check the serial port and try again'); end
   ```

   We turn the LED on in the next line. This is achieved using a command of the form

   ```
   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:

```
cmd_digital_out(1, 9, 1)          // This will turn the blue LED
```

The last line in the code closes the serial port. As mentioned above, it is extremely important to close the serial port properly. If not closed properly, there could be difficulties in running subsequent programs.

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

```
sleep(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

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

Exercise 4.2 Repeat the exercise of the previous section. ∎

4.4.2 Scilab Code

Scilab Code 4.1 Turning on the blue LED. Available at `Origin/user-code/led/scilab/led-blue.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200); // At port 2 with baudrate of 115200
if ok ~= 0, error('Check the serial port and try again'); end
cmd_digital_out(1, 9, 1)          // This will turn the blue LED
close_serial(1);                 // To close the connection safely
```

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

```
ok = open_serial(1, 2, 115200);
if ok ~= 0, error('Check the serial port and try again'); end
cmd_digital_out(1, 9, 1) // turn blue LED on
sleep(2000)              // let the blue LED be on for two seconds
cmd_digital_out(1, 9, 0) // turn off blue LED
close_serial(1);         // close the connection safely
```

Scilab 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/scilab/led-blue-red.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200); // At port 2 with baudrate of 115200
if ok ~= 0 error('Check the serial port and try again'); end
cmd_digital_out(1, 9, 1);      // This turns on the blue Led
cmd_digital_out(1, 11, 1);     // This turns on the red Led
sleep(5000);                  // Delay for 5 seconds
cmd_digital_out(1, 9, 0);      // This turns off the blue Led
sleep(3000);                  // Delay for 3 seconds
cmd_digital_out(1, 11, 0);     // This turns off the red Led
close_serial(1);              // To close the connection safely
```

Scilab Code 4.4 Blinking the green LED. Available at `Origin/user-code/led/scilab/led-green-blink.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200); // At port 2 with baudrate of 115200
if ok ~= 0 error('Check the serial port and try again'); end
for i = 1:5                      // Running for loop, 5 times
   cmd_digital_out(1, 10, 1);  // This turns on the green Led
   sleep(1000);               // Delay for 1 second
   cmd_digital_out(1, 10, 0);  // This turns off the green Led
   sleep(1000);               // Delay for 1 second
end
close_serial(1);              // To close the connection safely
```

4.5 Lighting the LED from Scilab Xcos

In this section, we will see how to light the LEDs from Scilab Xcos. We will carry out the same four experiments as in the previous sections. For each, we will give the location of the zcos file and the parameters to set. The reader should go through the instructions given in Sec. 3.3 before getting started.

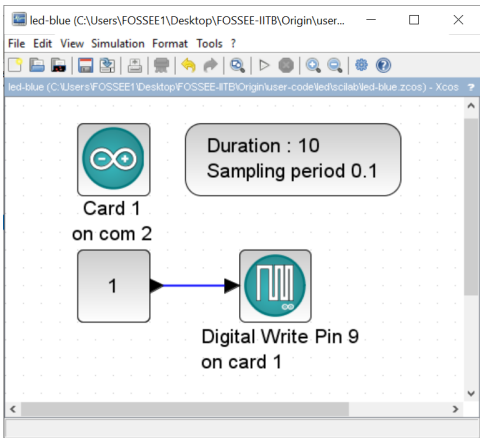


Figure 4.6: Turning the blue LED on through Xcos. This is what one sees when `Origin/user-code/led/scilab/led-blue.zcos`, see Footnote 2 on page 2 is invoked.

Table 4.1: Parameters to light the blue LED in Xcos

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_WRITE_SB	Digital pin	9
	Arduino card number	1

1. First we will see how to turn on the blue LED. When the file required for this experiment is invoked, one gets the GUI as in Fig. 4.6. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 4.1. All other parameters are to be left unchanged.

2. In the second experiment, we will show how to turn on the blue LED on for two seconds and then to turn it off. When the file required for this experiment is invoked, one gets the GUI as in Fig. 4.7. In the caption of this figure, one can see where to locate the file.

   The values for each block required in this program are tabulated in Table 4.2.

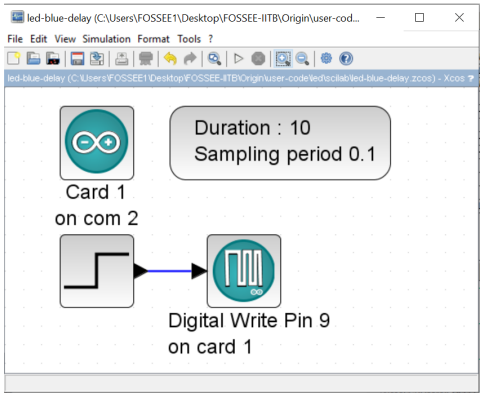


Figure 4.7: Turning the blue LED on through Xcos for two seconds. This is what one sees when `Origin/user-code/led/scilab/led-blue-delay.zcos`, see Footnote 2 on page 2 is invoked.

Table 4.2: Parameters to light the blue LED in Xcos for two seconds

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_WRITE_SB	Digital pin	9
	Arduino card number	1
STEP_FUNCTION	Step time	2
	Initial value	1
	Final value	0

All other parameters are to be left unchanged.

3. In the third experiment, we will show how to turn the blue LED and the red LED on for five seconds, turn off the blue LED and three seconds later, turn off the red LED also. When the file required for this experiment is invoked, one gets the GUI as in Fig. 4.8. In the caption of this figure, one can see where to locate the file.

   The values for each block required in this program are tabulated in Table 4.3. All other parameters are to be left unchanged.

4. We will conclude this section with an experiment to blink the green LED on

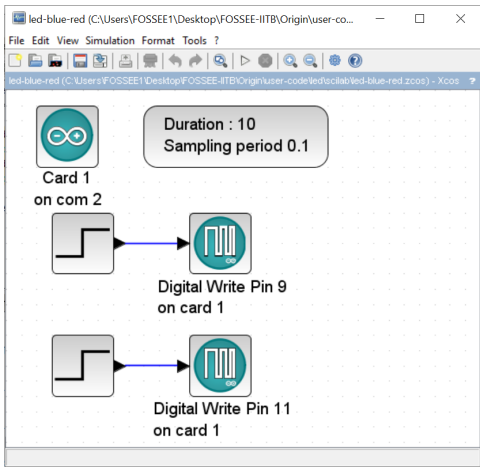


Figure 4.8: Turning the blue and red LEDs on through Xcos and turning them off one by one. This is what one sees when `Origin/user-code/led/scilab/led-blue-red.zcos`, see Footnote 2 on page 2 is invoked.

Table 4.3: Parameters to turn the blue and red LEDs on and then turn them off one by one

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_WRITE_SB 1	Digital pin	9
	Arduino card number	1
STEP_FUNCTION 1	Step time	5
	Initial value	1
	Final value	0
DIGITAL_WRITE_SB 2	Digital pin	11
	Arduino card number	1
STEP_FUNCTION 2	Step time	8
	Initial value	1
	Final value	0

and off. When the file required for this experiment is invoked, one gets the GUI as in Fig. 4.9. In the caption of this figure, one can see where to locate the file.

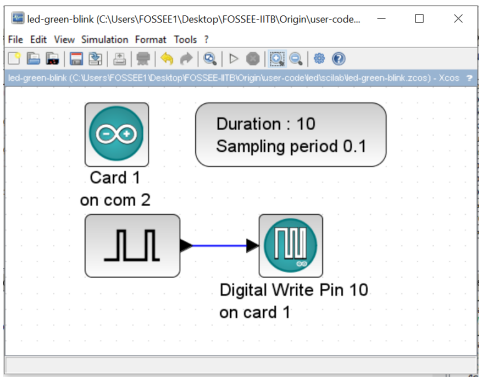


Figure 4.9: Blinking the green LED every second through Xcos. This is what one sees when `Origin/user-code/led/scilab/led-green-blink.zcos`, see Footnote 2 on page 2 is invoked.

Table 4.4: Parameters to make the green LED blink every second

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_WRITE_SB	Digital pin	10
	Arduino card number	1
PULSE_SC	Pulse width(% of period)	50
	Period(secs)	2
	Phase delay(secs)	0.1
	Amplitude	1

The values for each block required in this program are tabulated in Table 4.4. All other parameters are to be left unchanged.

Exercise 4.3 Carry out the following exercise:

1. Change the blink pattern for an array of LEDs.
2. Change the delays.

▮

4.6 Lighting the LED from Python

4.6.1 Lighting the LED

In this section, we discuss how to carry out the experiments of the previous section from Python. 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. 3.4 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 Python Code 4.1. It begins with importing necessary modules, as given below:

```
1 import os
2 import sys
```

   Here, `os` module provides functions for interacting with the operating system. On the other hand, the `sys` module provides access to some variables used or maintained by the interpreter and functions that interact strongly with the interpreter. Next, the following lines of code are used to get the current directory followed by splitting it and appending the path to `PYTHONPATH` environment:

```
1 cwd = os.getcwd()
2 (setpath, Examples) = os.path.split(cwd)
3 sys.path.append(setpath)
```

   After importing the necessary modules, following line imports the Arduino module from the Python-Arduino toolbox, as explained in Sec. 3.4.3:

```
1 from Arduino.Arduino import Arduino
```

   Next, we will import `sleep` module, which is a function available in the package pyserial [16]. For the sake of simplicity, we have configured each experiment as a class. In Python Code 4.1, we have defined the experiment as `class LED_ON`. The following lines of code initilize the parameters and functions available in this class.

```
1 class LED_ON:
2     def __init__(self, baudrate):
3         self.baudrate = baudrate
4         self.setup()
5         self.run()
6         self.exit()
```

The function `setup` creates an object of the Arduino class through which we can call all the methods available in the base class. Along with this, it locates the port to which Arduino Uno is connected and opens the port for serial communication. Thus, we require port number and BAUD RATE for opening the serial port.

The function `run` is used to define the functionality of the experiment. In this experiment, we have to switch on the blue LED. For this, we define the pin number (which is 9 in this case) and send the required signal to this pin. Following lines are used to implement this functionality:

```
    def run(self):
        self.blue = 9
        self.obj_arduino.cmd_digital_out(1, self.blue, 1)
```

At last, the function `exit` is invoked to close the serial port.

Once all the parameters and functions available in `class LED_ON` have been initialized, we create a main method and call it, as given below:

```
def main():
    obj_led = LED_ON(115200)

if __name__== '__main__':
    main()
```

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

```
        sleep(2)
```

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

```
        self.obj_arduino.cmd_digital_out(1, self.blue, 0)
```

As evident, this line of code puts a 0 on pin 9.

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

Exercise 4.4 Repeat the exercise of the previous section. ∎

4.6.2 Python Code

Python Code 4.1 Turning on the blue LED. Available at `Origin/user-code/led/python/led-blue.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class LED_ON:
11     def __init__(self, baudrate):
12         self.baudrate = baudrate
13         self.setup()
14         self.run()
15         self.exit()
16
17     def setup(self):
18         self.obj_arduino = Arduino()
19         self.port = self.obj_arduino.locateport()
20         self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22     def run(self):
23         self.blue = 9
24         self.obj_arduino.cmd_digital_out(1, self.blue, 1)
25
26     def exit(self):
27         self.obj_arduino.close_serial()
28
29 def main():
30     obj_led = LED_ON(115200)
31
32 if __name__== '__main__':
33     main()
```

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
```

```
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class LED_ON_OFF:
11     def __init__(self,baudrate):
12         self.baudrate = baudrate
13         self.setup()
14         self.run()
15         self.exit()
16
17     def setup(self):
18         self.obj_arduino = Arduino()
19         self.port = self.obj_arduino.locateport()
20         self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22     def run(self):
23         self.blue = 9
24         self.obj_arduino.cmd_digital_out(1, self.blue, 1)
25         sleep(2)
26         self.obj_arduino.cmd_digital_out(1, self.blue, 0)
27
28     def exit(self):
29         self.obj_arduino.close_serial()
30
31 def main():
32     obj_led = LED_ON_OFF(115200)
33
34 if __name__== '__main__':
35     main()
```

Python 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/python/led-blue-red.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class LED_ON_OFF_MULTICOLOR:
11
12     def __init__(self,baudrate):
13         self.baudrate = baudrate
14         self.setup()
15         self.run()
```

```
16          self.exit()
17
18      def setup(self):
19          self.obj_arduino = Arduino()
20          self.port = self.obj_arduino.locateport()
21          self.obj_arduino.open_serial(1, self.port, self.baudrate)
22
23      def run(self):
24          self.blue = 9
25          self.green = 10
26          self.red = 11
27          self.obj_arduino.cmd_digital_out(1, self.blue, self.baudrate)
28          self.obj_arduino.cmd_digital_out(1, self.red, self.baudrate)
29          sleep(5)
30          self.obj_arduino.cmd_digital_out(1, self.blue, 0)
31          sleep(3)
32          self.obj_arduino.cmd_digital_out(1, self.red, 0)
33
34      def exit(self):
35          self.obj_arduino.close_serial()
36
37  def main():
38      obj_led = LED_ON_OFF_MULTICOLOR(115200)
39
40  if __name__=='__main__':
41      main()
```

Python Code 4.4 Blinking the green LED. Available at `Origin/user-code/led/python/led-green-blink.py`, see Footnote 2 on page 2.

```
1  import os
2  import sys
3  cwd = os.getcwd()
4  (setpath, Examples) = os.path.split(cwd)
5  sys.path.append(setpath)
6
7  from Arduino.Arduino import Arduino
8  from time import sleep
9
10 class LED_ON_OFF_LOOP:
11
12   def __init__(self,baudrate):
13     self.baudrate = baudrate
14     self.setup()
15     self.run()
16     self.exit()
17
18   def setup(self):
19     self.obj_arduino = Arduino()
```

```
20      self.port = self.obj_arduino.locateport()
21      self.obj_arduino.open_serial(1, self.port, self.baudrate)
22
23    def run(self):
24      self.blue = 9
25      self.green = 10
26      self.red = 11
27      for i in range(5):
28        self.obj_arduino.cmd_digital_out(1, self.green, 1)
29        sleep(1)
30        self.obj_arduino.cmd_digital_out(1, self.green, 0)
31        sleep(1)
32
33    def exit(self):
34      self.obj_arduino.close_serial()
35
36  def main():
37    obj_led = LED_ON_OFF_LOOP(115200)
38
39  if __name__=='__main__':
40    main()
41
```

4.7 Lighting the LED from Julia

4.7.1 Lighting the LED

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 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 Julia Code 4.1. It begins with importing the SerialPorts [18] package and the module ArduinoTools, as given in Sec. 3.5.3. Following lines import SerialPorts and ArduinoTools:

   ```
   1 using SerialPorts
   2 include("ArduinoTools.jl")
   ```

   Next, the following line of code is used to detect the port on which Arduino Uno is connected.

   ```
   1 ser = ArduinoTools.connectBoard(115200)
   ```

Apart from detecting the port, it also opens a serial port with given BAUD RATE. In this experiment, we have to put a high voltage (5V) on pin 9 to turn the blue light on. This is achieved by the following command:

```
ArduinoTools.digiWrite(ser, 9, 1)
```

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

```
ArduinoTools.pinMode(ser, 9, "OUTPUT")
```

The last line in the code closes the serial port. In this Julia source file, one can observe that the blue light will be on continuously.

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

```
sleep(2)
```

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

```
ArduinoTools.digiWrite(ser, 9, 0)
```

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

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

4.7.2 Julia Code

Julia Code 4.1 Turning on the blue LED. Available at `Origin/user-code/led/julia/led-blue.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 9, "OUTPUT")
ArduinoTools.digiWrite(ser, 9, 1)
close(ser)
```

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 9, "OUTPUT")
ArduinoTools.digiWrite(ser, 9, 1)
sleep(2)
ArduinoTools.digiWrite(ser, 9, 0)
close(ser)
```

Julia 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/julia/led-blue-red.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 9, "OUTPUT")
ArduinoTools.pinMode(ser, 11, "OUTPUT")
ArduinoTools.digiWrite(ser, 9, 1)
ArduinoTools.digiWrite(ser, 11, 1)
sleep(5)
ArduinoTools.digiWrite(ser, 9, 0)
sleep(3)
ArduinoTools.digiWrite(ser, 11, 0)
close(ser)
```

Julia Code 4.4 Blinking the green LED. Available at `Origin/user-code/led/julia/led-green-blink.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 10, "OUTPUT")
for i = 1:5
  ArduinoTools.digiWrite(ser, 10, 1)
  sleep(1)
  ArduinoTools.digiWrite(ser, 10, 0)
  sleep(1)
end
close(ser)
```

4.8 Lighting the LED from OpenModelica

4.8.1 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. 3.6 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. 3.6.4. 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.

4.8.2 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. 3.6.4. Please refer to Fig. 3.44 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.

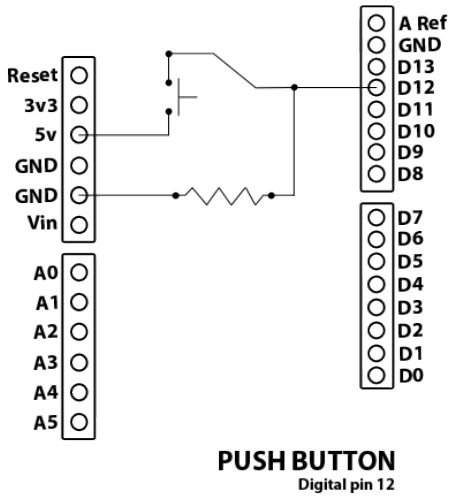


Figure 5.1: Internal connection diagram for the pushbutton on the shield

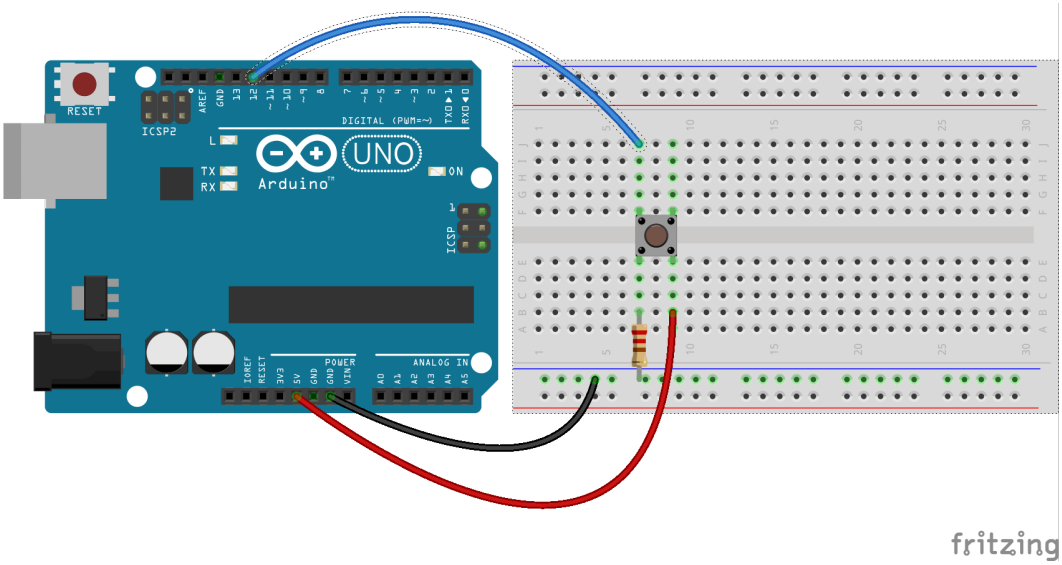


Figure 5.2: A pushbutton to read its status with Arduino Uno using a breadboard

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

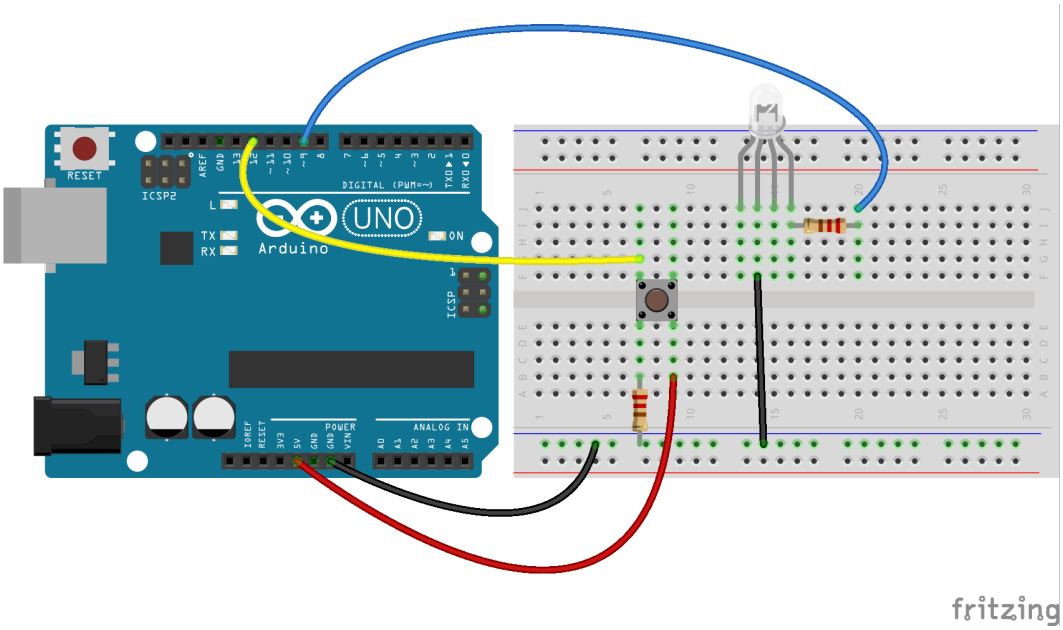


Figure 5.3: A pushbutton to control an LED with Arduino Uno using a breadboard

`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 Scilab

5.4.1 Reading the pushbutton Status

In this section, we discuss how to carry out the experiments of the previous section from Scilab. 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 using a Graphical user interface (GUI) in Scilab. The code for this experiment is given in Scilab Code 5.1. As explained earlier in Sec. 4.4.1, we begin with serial port initialization. Then, we read the input coming from digital pin 12 using the following command:

   ```
   1      val = cmd_digital_in(1, 12);  // Read the status of pin 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 as a GUI using the following command:

   ```
   1      cmd_arduino_meter(val);
   ```

   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". To encourage the user to have a good hands-on, we run these commands in a `for` loop for 1000 iterations. While running this experiment, the readers must press and release the pushbutton and observe the values being printed on the GUI, as shown in Fig. 5.4.

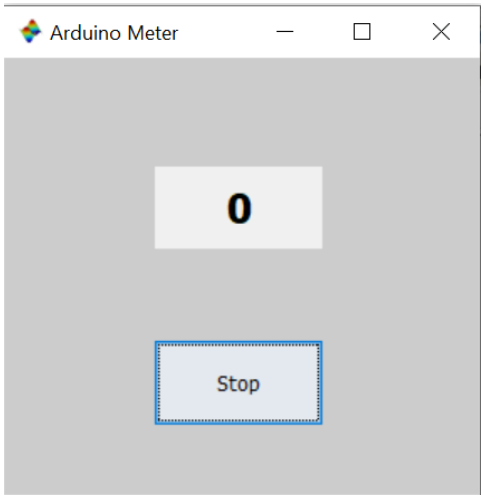


Figure 5.4: GUI in Scilab to show the status of the pushbutton

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 Scilab 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
2         cmd_digital_out(1, 9, 0)
3     else
4         cmd_digital_out(1, 9, 1)
```

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

5.4.2 Scilab Code

Scilab Code 5.1 Read the status of the pushbutton and display it on the GUI. Available at `Origin/user-code/push/scilab/push-button-status.sce`, see Footnote 2 on page 2.

```
1 ok = open_serial(1, 2, 115200);       // port 2, baud rate 115200
2 if ok ~= 0 then error('Unable to open serial port, please check'); end
3 for i = 1:1000                        // Run for 1000 iterations
```

```
4     val = cmd_digital_in(1, 12);  // Read the status of pin 12
5     cmd_arduino_meter(val);
6 end
7 close_serial(1);                  // To close the connection safely
```

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

```
1 ok = open_serial(1, 2, 115200);          // port 2, baudrate 115200
2 if ok ~= 0 then error('Unable to open serial port, please check'); end
3 for i = 1:1000                           //Run for 1000 iterations
4     val = cmd_digital_in(1, 12)
5     cmd_arduino_meter(val);
6     if val == 0
7        cmd_digital_out(1, 9, 0)
8     else
9        cmd_digital_out(1, 9, 1)
10    end
11 end
12 close_serial(1);
```

5.5 Accessing the pushbutton from Xcos

In this section, we will see how to access the pushbutton from Scilab Xcos. We will carry out the same two experiments as in the previous sections. For each, will give the location of the zcos file and the parameters to set. The reader should go through the instructions given in Sec. 3.3 before getting started.

1. First we will read the push button value and print it. When the file required for this experiment is invoked, one gets the GUI as in Fig. 5.5. In the caption of this figure, one can see where to locate the file.

   As discussed in earlier chapters, we start with the initialization of the serial port. Next, using `Digital Read` block, we read the status of pushbutton connected on digital pin 12. The read values are displayed. When a user presses the pushbutton, change in the logic value from low to high can be observed.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 5.1. All other parameters are to be left unchanged.

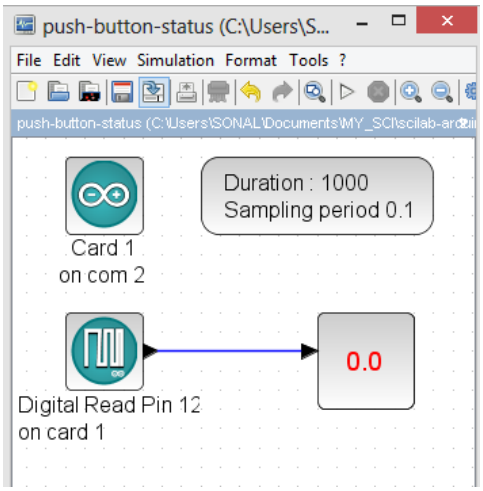


Figure 5.5: Printing the push button status on the display block. This is what one sees when `Origin/user-code/push/scilab/push-button-status.zcos`, see Footnote 2 on page 2, is invoked.

Table 5.1: Parameters to print the push button status on the display block

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_READ_SB	Digital pin	12
	Arduino card number	1
AFFICH_m	Block inherits (1) or not (0)	1

2. In the second experiment, we take a step further and control the state of an LED in accordance with the status of the pushbutton. The Xcos implementation for this experiment is shown in Fig. 5.6. Each time a user presses the pushbutton, the LED on digital pin 9 of the shield is switched on. If the shield is connected, the blue LED turns on. When the pushbutton is released, the LED is switched off. Here, we note that the digital logic level of the pin of the Arduino Uno board connected to pushbutton changes only for the time the pushbutton is being pressed.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or

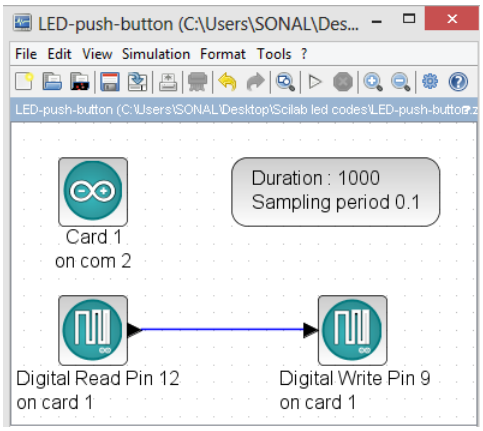


Figure 5.6: Turning the LED on or off, depending on the pushbutton. This is what one sees when `Origin/user-code/push/scilab/led-push-button.zcos`, see Footnote 2 on page 2, is invoked.

Table 5.2: Xcos parameters to turn the LED on through the pushbutton

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_READ_SB	Digital pin	12
	Arduino card number	1
DIGITAL_WRITE_SB	Digital pin	9
	Card number	1

double click. The values for each block is tabulated in Table 5.2. All other parameters are to be left unchanged.

Exercise 5.1 Let us carry out the following exercise:

1. In the above experiment, we controlled only one LED upon pushbutton press. Next, control multiple devices upon the pushbutton press. For example, upon press, turn on an LED and a motor and turn them off upon release.
2. Control several devices depending on the number of pushbutton press in a definite time span. For example, if the pushbutton is pressed once in time 't',say, turn on the LED. If it is pressed twice in time 't', turn on the motor. Here, you may want to consider the timing between two consecutive press.

5.6 Reading the pushbutton status from Python

5.6.1 Reading the pushbutton status

In this section, we discuss how to carry out the experiments of the previous section from Python. 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. 3.4 before getting started.

1. In the first experiment, we will read the pushbutton status. The code for this experiment is given in Python Code 5.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules followed by setting up the serial port. Then, we read the input coming from digital pin 12 using the following command:

```
1       val = self.obj_arduino.cmd_digital_in(1, self.pushbutton)
```

   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     for i in range(20):
2       val = self.obj_arduino.cmd_digital_in(1, self.pushbutton)
3       print(val)
4       sleep(0.5)
```

   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". To encourage the user to have a good hands-on, we run these commands in a `for` loop for 20 iterations. The readers are encouraged to change the number of iterations as per their requirements. While running this experiment, the readers must press and release the pushbutton and observe the values being printed on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

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 Python 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       for i in range(20):
2         val = self.obj_arduino.cmd_digital_in(1, self.pushbutton)
3         print(val)
4         self.obj_arduino.cmd_digital_out(1, self.blue, val)
```

control the LED state based on the status of the pushbutton. 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.

5.6.2 Python Code

Python Code 5.1 Read the status of the pushbutton and display it on Command Prompt or the Terminal. Available at `Origin/user-code/push/python/push-button-status.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath,Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class PUSHBUTTON:
11
12   def __init__(self,baudrate):
13     self.baudrate = baudrate
14     self.setup()
15     self.run()
16     self.exit()
17
18   def setup(self):
19     self.obj_arduino = Arduino()
20     self.port = self.obj_arduino.locateport()
21     self.obj_arduino.open_serial(1, self.port, self.baudrate)
22
23   def run(self):
24     self.pushbutton = 12
25     for i in range(20):
26       val = self.obj_arduino.cmd_digital_in(1, self.pushbutton)
27       print(val)
28       sleep(0.5)
```

```
29
30   def exit(self):
31     self.obj_arduino.close_serial()
32
33 def main():
34   obj_pushbutton = PUSHBUTTON(115200)
35
36 if __name__=='__main__':
37   main()
```

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath,Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class PUSHBUTTON_LED:
11
12   def __init__(self,baudrate):
13     self.baudrate = baudrate
14     self.setup()
15     self.run()
16     self.exit()
17
18   def setup(self):
19     self.obj_arduino = Arduino()
20     self.port = self.obj_arduino.locateport()
21     self.obj_arduino.open_serial(1, self.port, self.baudrate)
22
23   def run(self):
24     self.blue = 9
25     self.green = 10
26     self.red = 11
27     self.pushbutton = 12
28     for i in range(20):
29       val = self.obj_arduino.cmd_digital_in(1, self.pushbutton)
30       print(val)
31       self.obj_arduino.cmd_digital_out(1, self.blue, val)
32       sleep(0.5)
33
34   def exit(self):
35     self.obj_arduino.close_serial()
```

```
36
37 def main():
38     obj_pushbutton = PUSHBUTTON_LED(115200)
39
40 if __name__=='__main__':
41     main()
```

5.7 Reading the pushbutton status from Julia

5.7.1 Reading the pushbutton status

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 before getting started.

1. In the first experiment, we will read the pushbutton status. The code for this experiment is given in Julia Code 5.1. As explained earlier in Sec. 4.7.1, we begin with importing the SerialPorts [18] package and the module ArduinoTools followed by setting up the serial port. Then, we read the input coming from digital pin 12 using the following command:

   ```
   1    val = ArduinoTools.digiRead(ser, 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 for i = 1:20
   2     val = ArduinoTools.digiRead(ser, 12)
   3     println(val)
   ```

   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". To encourage the user to have a good hands-on, we run these commands in a `for` loop for 20 iterations. The readers are encouraged to change the number of iterations as per their requirements. While running this experiment, the readers must press and release the pushbutton and observe the values being printed on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

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 Julia 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,

```
if val == 0
  ArduinoTools.digiWrite(ser, 9, 0)
else
  ArduinoTools.digiWrite(ser, 9, 1)
```

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.

5.7.2 Julia Code

Julia Code 5.1 Read the status of the pushbutton and display it on Command Prompt or the Terminal. Available at `Origin/user-code/push/julia/push-button-status.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 12, "INPUT")
for i = 1:20
  val = ArduinoTools.digiRead(ser, 12)
  println(val)
  sleep(0.5)
end
close(ser)
```

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 9, "OUTPUT")
ArduinoTools.pinMode(ser, 12, "INPUT")
for i = 1:20
```

```
8   val = ArduinoTools.digiRead(ser, 12)
9   println(val)
10  if val == 0
11    ArduinoTools.digiWrite(ser, 9, 0)
12  else
13    ArduinoTools.digiWrite(ser, 9, 1)
14  end
15  sleep(0.5)
16 end
17 close(ser)
```

5.8 Reading the pushbutton status from OpenModelica

5.8.1 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. 3.6 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. 4.8.1, 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. 3.43.

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.

5.8.2 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. 3.6.4. Please refer to Fig. 3.44 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 push-button. 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

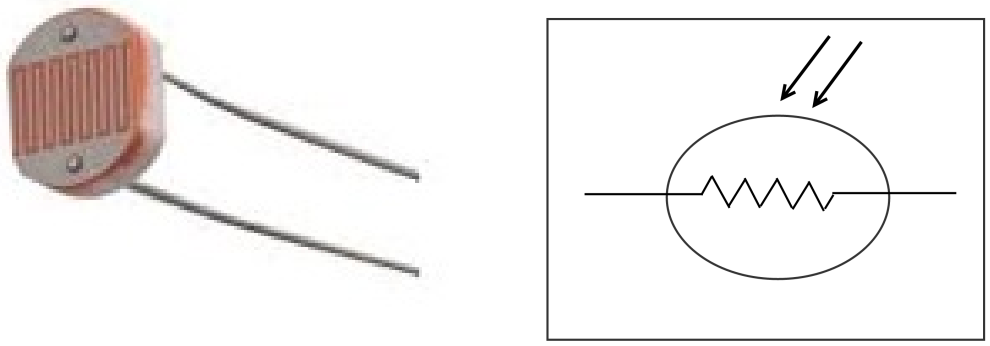
A Light Dependent Resistor (LDR) or Photoresistor is a light sensitive semiconductor device whose resistance varies with the variation in the intensity of light falling on it. As the intensity of the incident light increases, resistance offered by the LDR decreases. Typically, in dark, the resistance 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

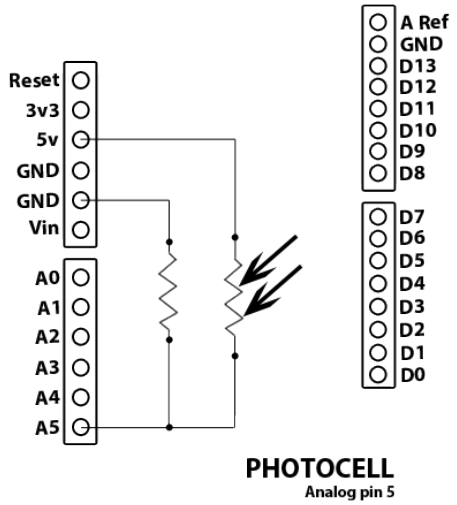


Figure 6.2: Internal connection diagram for the LDR on the shield

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.

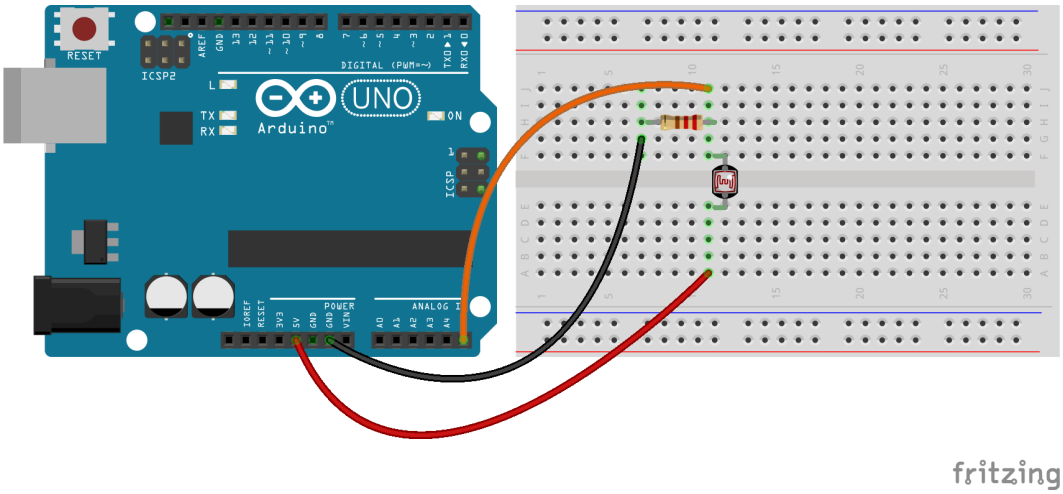


Figure 6.3: An LDR to read its values with Arduino Uno using a breadboard

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

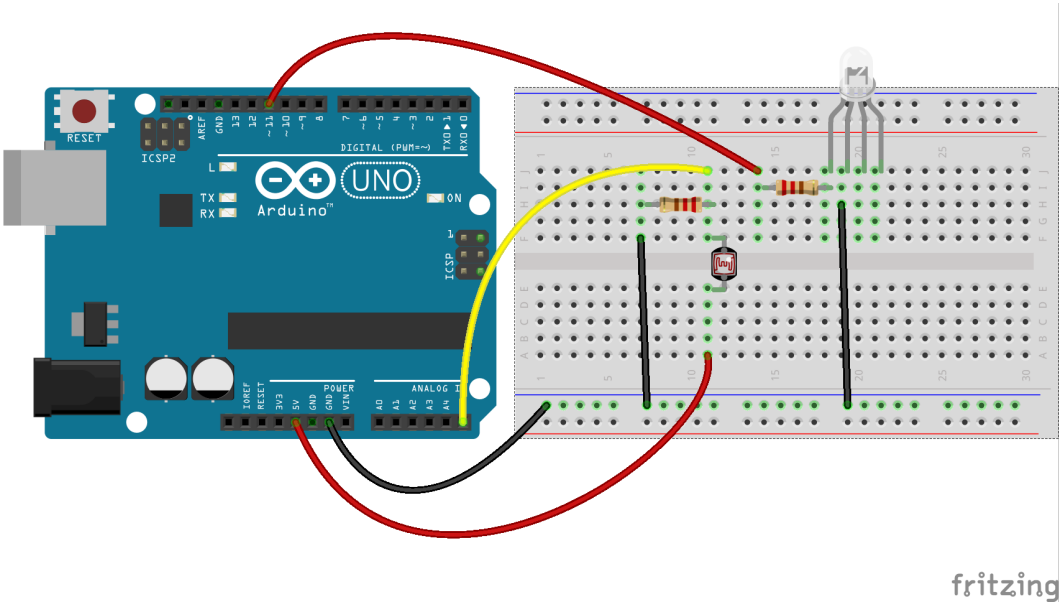


Figure 6.4: An LDR to control an LED with Arduino Uno using a breadboard

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 Scilab

6.4.1 Interfacing the LDR

In this section, we discuss how to carry out the experiments of the previous section from Scilab. 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 and display it in Scilab Console. The code for this experiment is given in Scilab Code 6.1. As explained earlier in Sec. 4.4.1, we begin with serial port initialization. Then, we read the input coming from analog pin 5 using the following command:

```
val = 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 in the Scilab Console by the following command:

```
disp(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. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 50 iterations. While running this experiment, the readers must keep their fingertips on the LDR and observe the change in values being printed on the Scilab Console.

2. This experiment is an extension of the previous experiment. Here, depending on the resistance of the LDR, we will turn the red LED on. The program for this is available at Scilab 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 6.2 Carry out the exercise below:

1. Carry out the exercise in the previous section.
2. Calculate the difference in LDR readings in indoor room before lighting the lamp and after lighting the lamp. You can also record changes in the room lighting at different times of the day.

6.4.2 Scilab Code

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

```
ok = open_serial(1, 2, 115200); // Port 2 with baudrate 115200
if ok ~= 0 then error('Unable to open serial port.  Please check'); end
for i = 1:50                    // Run for 50 iterations
    val = cmd_analog_in(1, 5);   // read analog pin 5 (ldr)
    disp(val);
    sleep(500)                 // Delay of 500 milliseconds
end
c = close_serial(1);            // close serial connection
```

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

```
ok = open_serial(1, 2, 115200);     // port 2, baudrate 115200
if ok ~= 0 then error('Unable to open serial port, please check'); end
for i = 1:50 //Run for 50 iterations
    val = cmd_analog_in(1, 5)        // read analog pin 5 (ldr)
    disp(val);
    if(val < 300)                    // Setting Threshold value of 300
        cmd_digital_out(1, 11, 1) // Turn ON LED
    else
        cmd_digital_out(1, 11, 0) // Turn OFF LED
    end
    sleep(500)
end
close_serial(1);
```

6.5 Interfacing the LDR through Xcos

Next, we shall perform the above mentioned experiments, to read LDR values, through Xcos. We will carry out the same two experiments as in the previous sections. For each, will give the location of the zcos file and the parameters to set. The reader should go through the instructions given in Sec. 3.3 before getting started.

1. First we will read the LDR values and display it. When the file required for this experiment is invoked, one gets the GUI as in Fig. 6.5. In the caption of this figure, one can see where to locate the file.

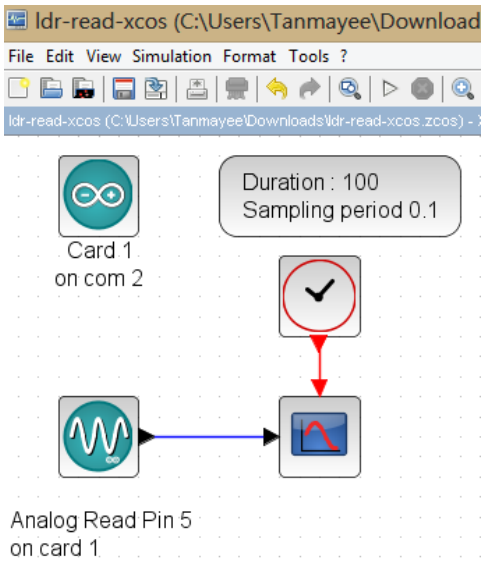


Figure 6.5: Xcos diagram to read LDR values. This is what one sees when `Origin/user-code/ldr/scilab/ldr-read.zcos`, see Footnote 2 on page 2, is invoked.

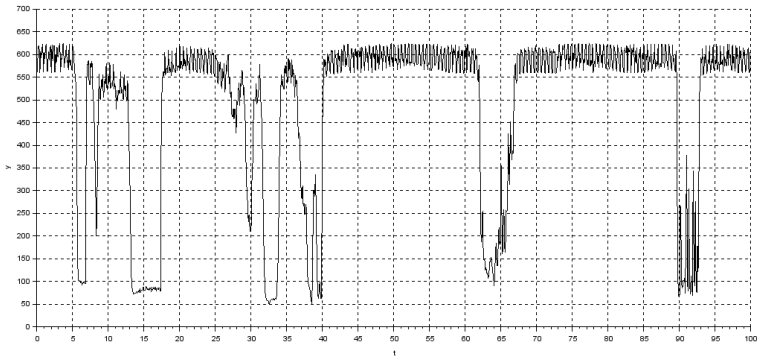


Figure 6.6: Plot window in Xcos to read LDR values

As discussed in earlier chapters, we start with the initialization of the serial port. Next, using `Analog Read` block, we read the values of LDR connected on analog pin 5. Next, we use a scope to plot the values coming from this pin. When this Xcos file is simulated, a plot is opened, as shown in Fig. 6.6.

Table 6.1: Xcos parameters to read LDR

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
ANALOG_READ_SB	Analog Pin	5
	Arduino card number	1
CSCOPE	Ymin	0
	Ymax	1023
	Refresh period	100
CLOCK_c	Period	0.1
	Initialisation Time	0

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 6.1. All other parameters are to be left unchanged.

During this experiment, we vary the light incident on LDR by using light sources and obstacles such as torch light, paper, hand (or fingertips), etc. and observe the LDR readings in the plot, as shown in Fig. 6.6. We observe that with a constant light source, the LDR output saturates after some time.

2. In the second experiment, we take a step further and control the state of red LED in accordance with the LDR values. When the file required for this experiment is invoked, one gets the GUI as in Fig. 6.7. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 6.2. In the CSCOPE_c block, the two values correspond to two graphs, one for digital write and other for analog read values. All other parameters are to be left unchanged. When this Xcos file is simulated, a plot is opened, as shown in Fig. 6.8.

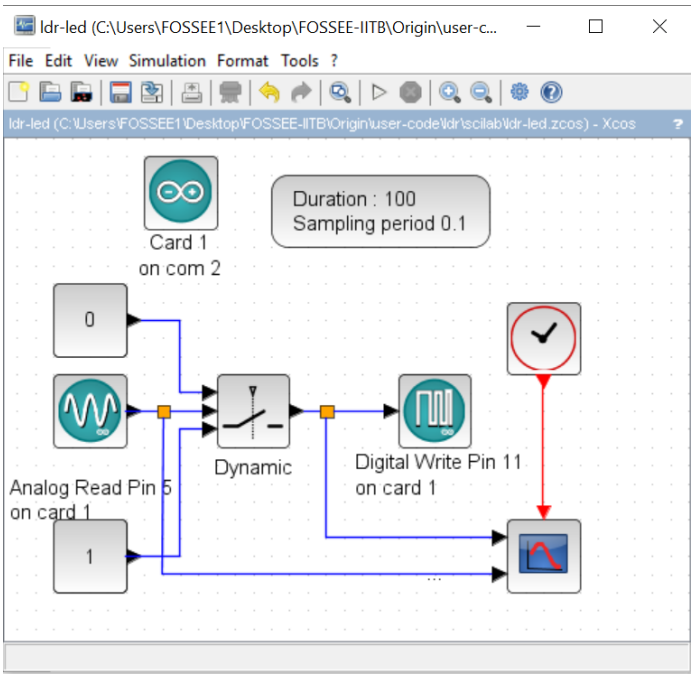


Figure 6.7: Xcos diagram to read the value of the LDR, which is used to turn the blue LED on or off. This is what one sees when `Origin/user-code/ldr/scilab/ldr-led-xcos.zcos`, see Footnote 2 on page 2, is invoked.

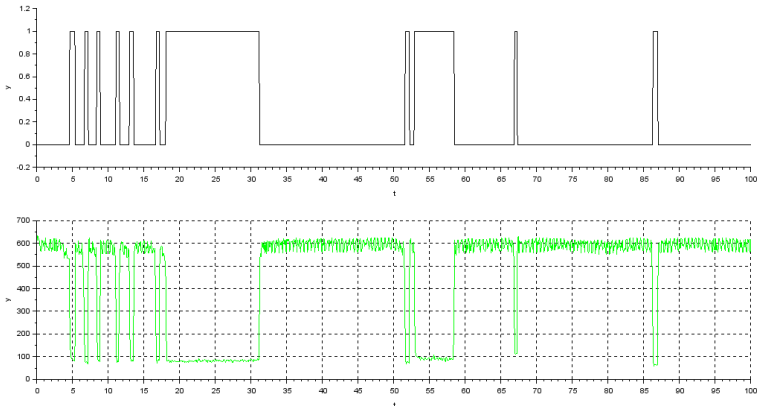


Figure 6.8: Plot window in Xcos to read LDR values and the state of LED

Table 6.2: Xcos parameters to read LDR and regulate blue LED

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
ANALOG_READ_SB	Analog pin	5
	Arduino card number	1
CMSCOPE	Ymin	0 0
	Ymax	1 1023
	Refresh period	100 100
CLOCK_c	Period	0.1
	Initialisation time	0
SWITCH2_m	Datatype	1
	threshold	300
	pass first input if field	0
	use zero crossing	1
DIGITAL_WRITE_SB	Digital pin	9
	Arduino card number	1

6.6 Interfacing the LDR through Python

6.6.1 Interfacing the LDR

In this section, we discuss how to carry out the experiments of the previous section from Python. 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. 3.4 before getting started.

1. In the first experiment, we will read the LDR values. The code for this experiment is given in Python Code 6.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules followed by setting up the serial port. Then, we read the input coming from analog pin 5 using the following command:

```
1         val = self.obj_arduino.cmd_analog_in(1, self.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:

```
1        print(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. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 50 iterations. While running this experiment, the readers must keep their fingertips on the LDR and observe the change in values being printed on on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

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 Python Code 6.2. The value of LDR is read and stored in `val`. In case it is below some threshold (like 300 in Python 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 6.3 Carry out the exercise below:

1. Carry out the exercise in the previous section.
2. Calculate the difference in LDR readings in indoor room before lighting the lamp and after lighting the lamp. You can also record changes in the room lighting at different times of the day.

6.6.2 Python Code

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath,Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
```

```
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class LDR:
11   def __init__(self, baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.ldr = 5
24     for i in range(50):
25       val = self.obj_arduino.cmd_analog_in(1, self.ldr)
26       print(val)
27       sleep(0.5)
28
29   def exit(self):
30     self.obj_arduino.close_serial()
31
32 def main():
33   obj_ldr = LDR(115200)
34
35 if __name__=='__main__':
36   main()
```

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath,Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class LDR:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
```

```
15      self.exit()
16
17    def setup(self):
18      self.obj_arduino = Arduino()
19      self.port = self.obj_arduino.locateport()
20      self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22    def run(self):
23      self.ldr = 5
24      self.blue = 9
25      self.green = 10
26      self.red = 11
27      for i in range(50):
28        val = self.obj_arduino.cmd_analog_in(1, self.ldr)
29        print (val)
30        if int(val) < 300:
31          self.obj_arduino.cmd_digital_out(1, self.red, 1)
32        else:
33          self.obj_arduino.cmd_digital_out(1, self.red, 0)
34        sleep(0.5)
35
36    def exit(self):
37      self.obj_arduino.close_serial()
38
39  def main():
40    obj_ldr = LDR(115200)
41
42  if __name__== '__main__':
43    main()
```

6.7 Interfacing the LDR through Julia

6.7.1 Interfacing the LDR

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 before getting started.

1. In the first experiment, we will read the LDR values. The code for this experiment is given in Julia Code 6.1. As explained earlier in Sec. 4.7.1, we begin with importing the SerialPorts [18] package and the module ArduinoTools followed by setting up the serial port. Then, we read the input coming from analog pin 5 using the following command:

```
val = ArduinoTools.analogRead(ser, 5)
```

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:

```
println(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. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 50 iterations. While running this experiment, the readers must keep their fingertips on the LDR and observe the change in values being printed on on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

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 Julia Code 6.2. The value of LDR is read and stored in `val`. In case it is below some threshold (like 300 in Julia 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 6.4 Carry out the exercise below:

1. Carry out the exercise in the previous section.
2. Calculate the difference in LDR readings in indoor room before lighting the lamp and after lighting the lamp. You can also record changes in the room lighting at different times of the day.

∎

6.7.2 Julia Code

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
for i = 1:50
  val = ArduinoTools.analogRead(ser, 5)
  println(val)
  sleep(0.5)
end
close(ser)
```

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 11, "OUTPUT")
for i = 1:50
  val = ArduinoTools.analogRead(ser, 5)
  println(val)
  if val > 300
    ArduinoTools.digiWrite(ser, 11, 0)
  else
    ArduinoTools.digiWrite(ser, 11, 1)
  end
  sleep(0.5)
end
close(ser)
```

6.8 Interfacing the LDR through OpenModelica

6.8.1 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. 3.6 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. 4.8.1, 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. 3.43.

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.

6.8.2 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. 3.6.4. Please refer to Fig. 3.44 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
    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));
    if val < 300 then
      digital_out := sComm.cmd_digital_out(1, 11, 1) "Turn ON LED";
    else
      digital_out := sComm.cmd_digital_out(1, 11, 0) "Turn OFF LED";
    end if;
    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

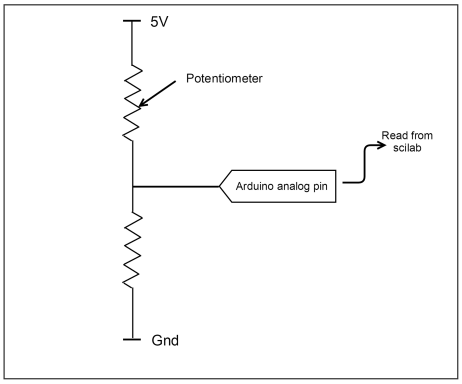
A potentiometer is a three-terminal variable resistor with two terminals connected to the two ends of a resistor and one connected to a sliding or rotating contact, termed as a wiper. The wiper can be moved to vary the resistance, and hence the potential, between the wiper and each terminal of the resistor. 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.

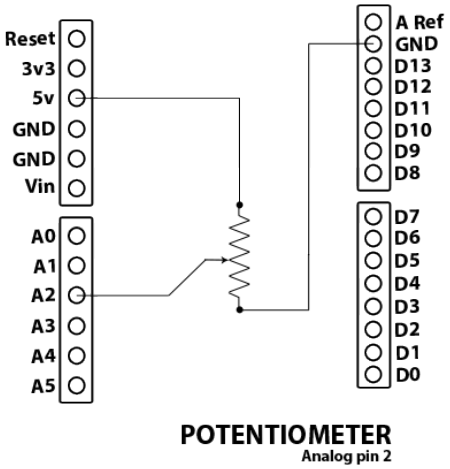
7.1 Preliminaries

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 68. 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

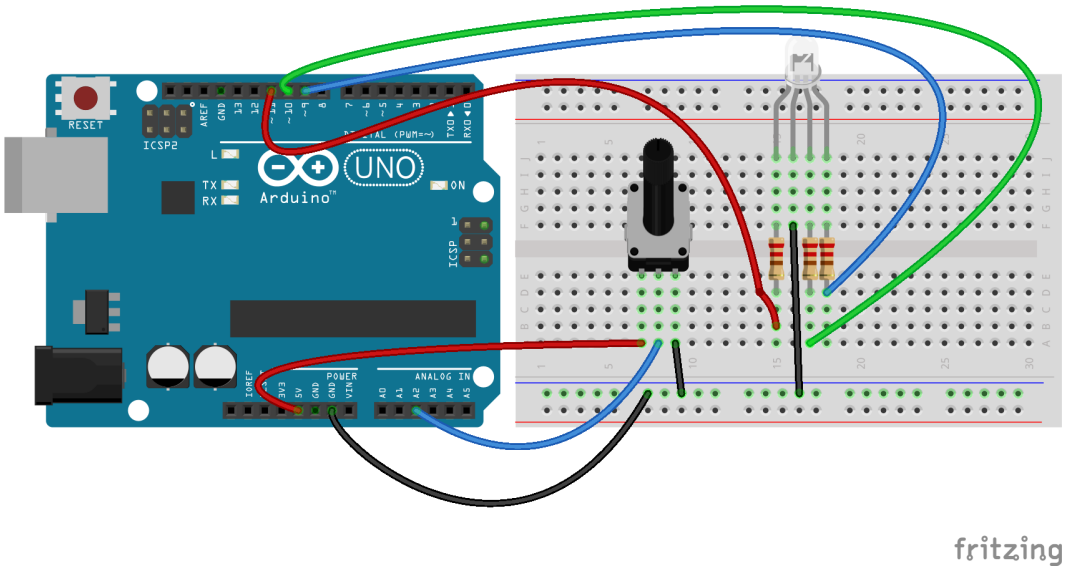


Figure 7.2: A potentiometer to control an LED with Arduino Uno using a breadboard

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 Scilab

7.4.1 Reading the potentiometer

In this section, we will use a Scilab script to read the potentiometer values. Based 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 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.

As explained earlier, the potentiometer values range from 0 to 1023. We will divide this entire range into 3 bands, 0-319, 320-900, and 901-1023. For each read value, we use an `if elseif` statement and correspondingly turn on either the Red, Green or Blue LED. The code for this experiment is given in Scilab Code 7.1. We start the experiment by opening the serial port for communication between Scilab

and the Arduino Uno board. Then, we read the analog input at pin 2 using,

```
val = cmd_analog_in(1, 2)
```

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

```
if (val >= 0 & val < 320) then          //threshold 1
    cmd_digital_out(1, 11, 1)
    sleep(1000)
    cmd_digital_out(1, 11, 0)
```

where `cmd_digital_out` is used to set the pin 11 high (1) or low (0). We used `sleep(1000)` to retain the LED in the on state for 1000 milliseconds. A similar check is done for 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 Scilab Code

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

```
ok = open_serial(1, 2, 115200); // port 2, baud rate 115200
if ok ~= 0 then error('Unable to open serial port, please check'); end
for x = 1:20 //Run for 20 iterations
    val = cmd_analog_in(1, 2)
    disp(val)
    if (val >= 0 & val < 320) then          //threshold 1
        cmd_digital_out(1, 11, 1)
        sleep(1000)
        cmd_digital_out(1, 11, 0)
    elseif (val >= 320 & val <= 900)          //threshold 2
        cmd_digital_out(1, 10, 1)
        sleep(1000)
        cmd_digital_out(1, 10, 0)
    elseif (val > 900 & val <= 1023)          //threshold 3
        cmd_digital_out(1, 9, 1)
        sleep(1000)
        cmd_digital_out(1, 9, 0)
    end
end
close_serial(1);
```

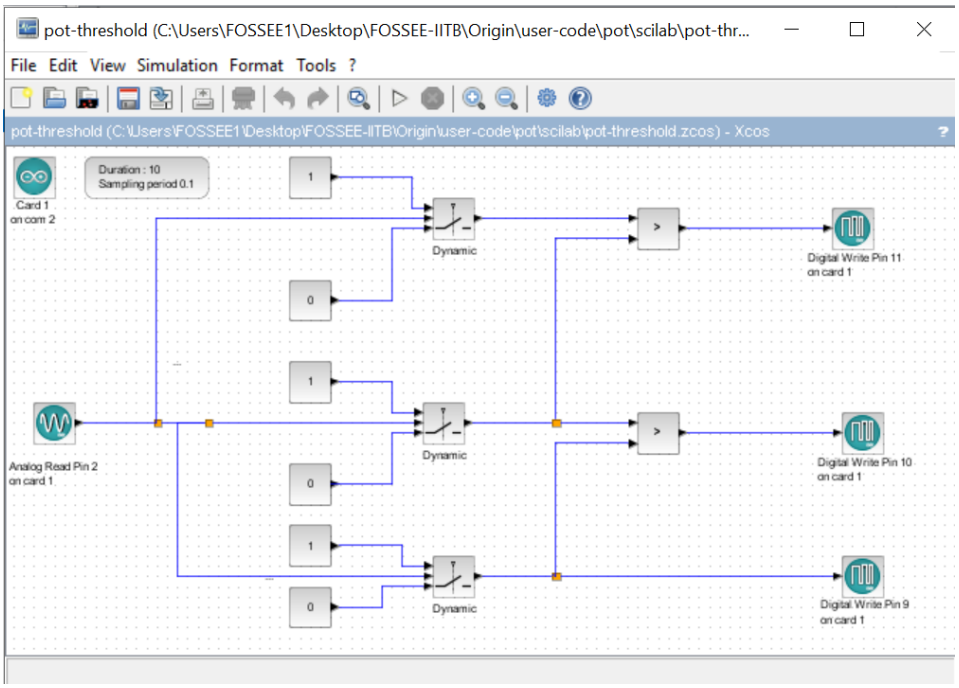


Figure 7.3: Turning LEDs on through Xcos depending on the potentiometer threshold. This is what one sees when `Origin/user-code/pot/scilab/pot-threshold.zcos`, see Footnote 2 on page 2, is invoked.

7.5 Reading the potentiometer from Xcos

In this section, we discuss how to read the potentiometer values using Xcos blocks. When the file required for this experiment is invoked, one gets the GUI as in Fig. 7.3. In the caption of this figure, one can see where to locate the file. The reader should go through the instructions given in Sec. 3.3 before getting started.

In this experiment, the block `Analog Read Pin 2` performs the read operation from pin 2. The threshold is set using the block, `Dynamic`. Depending on the condition met, a 1 or 0 is given to pin 9, 10 or 11.

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 7.1. All other parameters are to be left unchanged.

Note that, when the potentiometer value read by Scilab crosses either of the thresholds, color of the LED changes. This can be observed by rotating the knob of the potentiometer.

Table 7.1: Xcos parameters to turn on different LEDs depending on the potentiometer value

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
CONST_m	Constant Value	1, 0
DIGITAL_WRITE_SB	Digital Pin	9(blue)
	Digital Pin	10(green)
	Digital Pin	11(red)
	Arduino card number	1
ANALOG_READ_SB	analog pin	2
	Arduino card number	1
SWITCH2_m	Datatype	1
	Pass first input	1
	threshold	0
	use zero crossing	1
SWITCH2_m	Datatype	1
	Pass first input	0
	threshold	320
	use zero crossing	1
SWITCH2_m	Datatype	1
	Pass first input	0
	threshold	900
	use zero crossing	1
RELATIONALOP	Operator	4
	zero crossing	0
	Datatype	1

Exercise 7.1 List out the applications in day to day life where potentiometer is being used/can be used? For example, old fan regulators used potentiometer to change the fan speed.

▮

7.6 Reading the potentiometer from Python

7.6.1 Reading the potentiometer

In this section, we will use a Python script 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. 3.4 before getting started.

Based on the acquired potentiometer values, we will change the state of the RGB LED. As explained earlier, the potentiometer values range from 0 to 1023. We will divide this entire range into 3 bands, 0-319, 320-900, and 901-1023. For each read value, we use an `if elif` statement and correspondingly turn on either the Red, Green or Blue LED. The code for this experiment is given in Python Code 7.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules followed by setting up the serial port. Then, we read the analog input at pin 2 using,

```
1       val = self.obj_arduino.cmd_analog_in(1, self.pot)
```

where the first argument is for the kit number 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 (int(val) >= 0 and int(val) < 320):
2         self.obj_arduino.cmd_digital_out(1, self.red, 1)
3         sleep(1)
4         self.obj_arduino.cmd_digital_out(1, self.red, 0)
```

where `cmd_digital_out` is used to set the pin 11 high (1) or low (0). We used `sleep(1)` to retain the LED in the on state for one second. 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.

7.6.2 Python Code

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
```

```
8  from time import sleep
9
10 class POT:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.pot = 2
24     self.blue = 9
25     self.green = 10
26     self.red = 11
27     for i in range(20):
28       val = self.obj_arduino.cmd_analog_in(1, self.pot)
29       print(val)
30
31       if (int(val) >= 0 and int(val) < 320):
32         self.obj_arduino.cmd_digital_out(1, self.red, 1)
33         sleep(1)
34         self.obj_arduino.cmd_digital_out(1, self.red, 0)
35       elif (int(val) >= 320 and int(val) < 900):
36         self.obj_arduino.cmd_digital_out(1, self.green, 1)
37         sleep(1)
38         self.obj_arduino.cmd_digital_out(1, self.green, 0)
39       elif (int(val) >= 900 and int(val) <= 1023):
40         self.obj_arduino.cmd_digital_out(1, self.blue, 1)
41         sleep(1)
42         self.obj_arduino.cmd_digital_out(1, self.blue, 0)
43
44   def exit(self):
45     self.obj_arduino.close_serial()
46
47 def main():
48   obj_pot = POT(115200)
49
50 if __name__=='__main__':
51   main()
```

7.7 Reading the potentiometer from Julia

7.7.1 Reading the potentiometer

In this section, we will use a Julia source file 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. 3.5 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 Julia Code 7.1. As explained earlier in Sec. 4.7.1, we begin with importing the SerialPorts [18] package and the module ArduinoTools followed by setting up the serial port. Then, we read the analog input at pin 2 using,

```
val = ArduinoTools.analogRead(ser, 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,

```
if (val >= 0 && val < 320)
  ArduinoTools.digiWrite(ser, 11, 1)
  sleep(1)
  ArduinoTools.digiWrite(ser, 11, 0)
```

where `digiWrite` is used to set the pin 11 high (1) or low (0). We used `sleep(1)` to retain the LED in the on state for 1 second. 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.

7.7.2 Julia Code

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.pinMode(ser, 9, "OUTPUT")
ArduinoTools.pinMode(ser, 10, "OUTPUT")
ArduinoTools.pinMode(ser, 11, "OUTPUT")
for i = 1:20
  val = ArduinoTools.analogRead(ser, 2)
```

```
10   println(val)
11   if (val >= 0 && val < 320)
12     ArduinoTools.digiWrite(ser, 11, 1)
13     sleep(1)
14     ArduinoTools.digiWrite(ser, 11, 0)
15   elseif (val >= 320 && val < 900)
16     ArduinoTools.digiWrite(ser, 10, 1)
17     sleep(1)
18     ArduinoTools.digiWrite(ser, 10, 0)
19   elseif (val >= 900 && val <= 1023)
20     ArduinoTools.digiWrite(ser, 9, 1)
21     sleep(1)
22     ArduinoTools.digiWrite(ser, 9, 0)
23   end
24 end
25 close(ser)
```

7.8 Reading the potentiometer from OpenModelica

7.8.1 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. 3.6 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. 4.8.1, 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.

7.8.2 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. 3.6.4. Please refer to Fig. 3.44 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.

```
model pot_threshold
  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";
  end when;
  if ok <> 0 then
    strm.print("Unable to open serial port, please check");
  else
    val := sComm.cmd_analog_in(1, 2) "read analog pin 2";
    strm.print("Potentiometer Readings: " + String(val));
    if val >= 0 and val < 320 then
      digital_out := sComm.cmd_digital_out(1, 11, 1) "Turn ON LED";
      sComm.delay(1000);
      digital_out := sComm.cmd_digital_out(1, 11, 0) "Turn OFF LED";
    elseif val >= 320 and val < 900 then
      digital_out := sComm.cmd_digital_out(1, 10, 1) "Turn ON LED";
      sComm.delay(1000);
      digital_out := sComm.cmd_digital_out(1, 10, 0) "Turn OFF LED";
    elseif val > 900 and val <= 1023 then
      digital_out := sComm.cmd_digital_out(1, 9, 1) "Turn ON LED";
      sComm.delay(1000);
      digital_out := sComm.cmd_digital_out(1, 9, 0) "Turn OFF LED";
    end if;
  end if;
//Threshold 1
//Threshold 2
  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

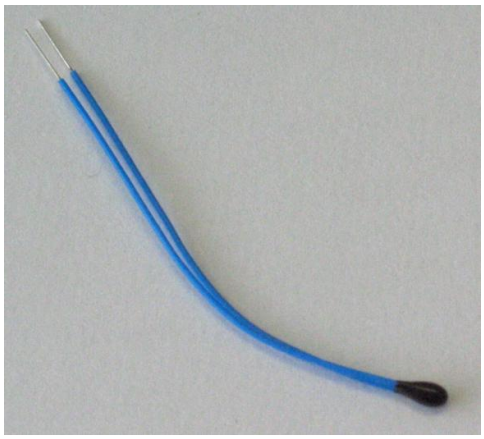
Interfacing a Thermistor

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.

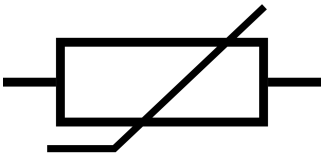
Thermistors are available in a variety of shapes such as beads, rods, flakes, and discs. Due to their compact size and low cost, they are widely used in the applications where even imprecise temperature sensing is sufficient. They, however, suffer from noise and hence need noise compensation. In this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

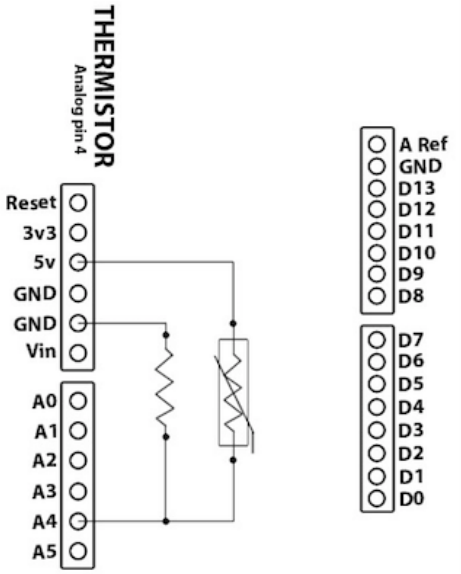


(a) Pictorial representation of a thermistor [20]

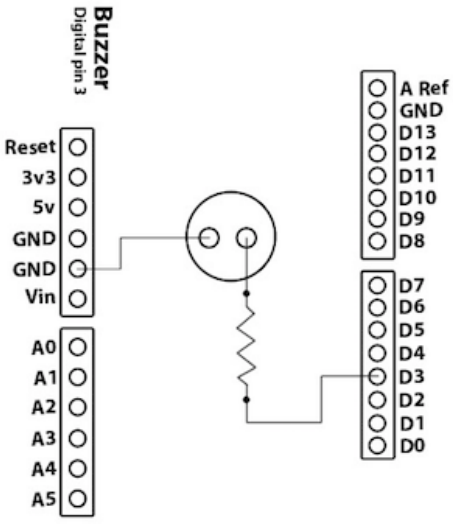


(b) Symbolic representation of a thermistor

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

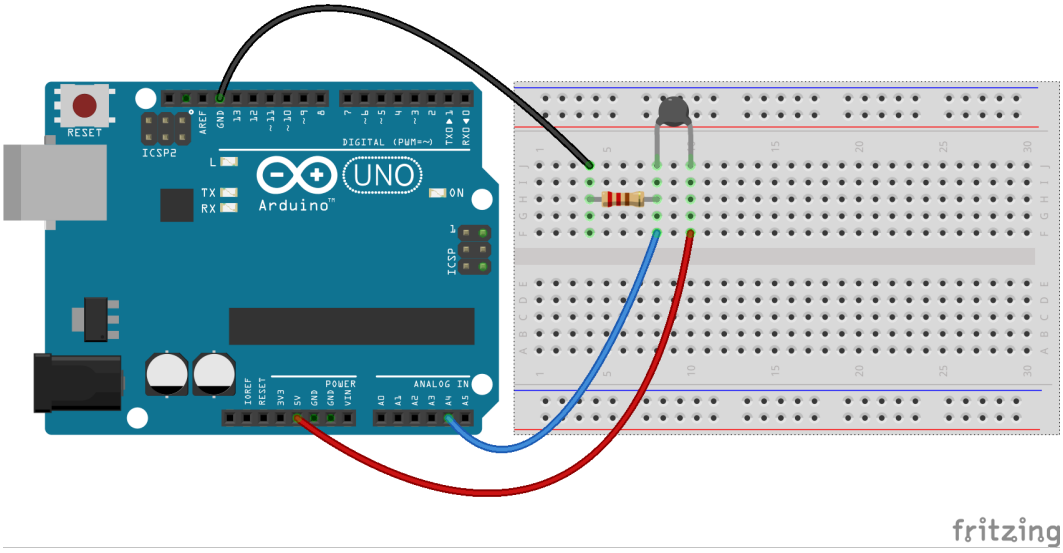


Figure 8.3: A thermistor to read its values with Arduino Uno using a breadboard

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.

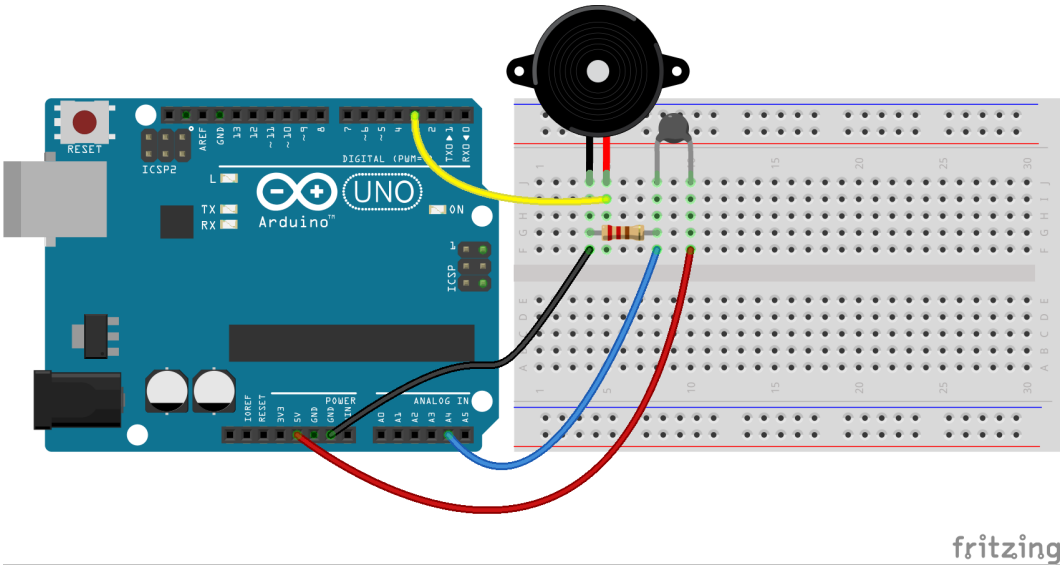


Figure 8.4: A thermistor to control a buzzer with Arduino Uno using a breadboard

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 Scilab

8.4.1 Interfacing the thermistor

In this section we will explain a Scilab script to read the thermistor values. Based on the acquired values, we will change the state of the buzzer. 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 and display it in Scilab Console. The code for this experiment is given in Scilab Code 8.1. As explained earlier in Sec. 4.4.1, we begin with serial port initialization. Then, we read the input coming from analog pin 4 using the following command:

```
val = cmd_analog_in(1, 4);    // read analog pin 4 (thermistor)
```

   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 stored in variable `val` and displayed in the Scilab Console by the following command:

```
disp(val);
```

   where `val` contains the thermistor values ranging from 0 to 1023. The changes in the thermistor resistance is sensed as a voltage change between 0 to 5V. 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. 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. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 20 iterations.

   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 Scilab Console.

2. This experiment is an extension of the previous experiment. Here, we will use a Scilab script to 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 Scilab 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 Scilab Code 8.2. For testing purposes, one may note the normal thermistor readings generated from the execution of Scilab Code 8.1 and set a threshold that is approximately 10 more than these readings.

In this experiment, 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)                    // Setting Threshold value of
     550
2        cmd_digital_out(1, 3, 1) // Turn ON BUZZER
3    else
4        cmd_digital_out(1, 3, 0) // Turn OFF BUZZER
5    end
```

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 Scilab Code 8.1.

Exercise 8.2 Carry out the exercise below: Convert the ADC output readings to degree Celsius. There are two ways to do so.

1. In the first method,

$$\frac{1}{T} = A + B * \ln(R) + C * (\ln(R))^3 \tag{8.1}$$

equation 8.1 can be used if the value of A, B, C and R are known. The temperature T is in kelvin and thermistor resistance R is in ohms. The values of A, B and C can be found out by measuring thermistor resistance against three known values of temperatures. The values of temperature

must be within the operating range and should typically include the room temperature. Once a set of three values of T and R are known it will result in three equations with three unknowns. The values of A, B, C can be found out by solving the three equations simultaneously. Once the values of A, B, C are known, the same equation can be used to directly convert resistance to kelvin. It can be then converted to Celsius. This method is preferred when the temperature coefficient of thermistor is not known or is known very approximately. This method is bit cumbersome but can give accurate temperature conversion.

2. In the second method,

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{\beta} * \ln\left(\frac{R}{R_0}\right) \tag{8.2}$$

equation 8.2 can be used if the value of β i.e. the Temperature Coefficient of Resistance of the thermistor used is known. The value of β can be found in the data sheet of the thermistor used. R is the resistance of thermistor at temperature T in kelvin. R_0 is the resistance of thermistor at room temperature T_0 in kelvin.

8.4.2 Scilab Code

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

```
1 ok = open_serial(1, 2, 115200); // Port 2 with baudrate 115200
2 if ok ~= 0 then error('Unable to open serial port.  Please check') end
3 for i = 1:20                    // Run for 20 iterations
4     val = cmd_analog_in(1, 4);    // read analog pin 4 (thermistor)
5     disp(val);
6     sleep(500);                 // Delay of 500 milliseconds
7 end
8 c = close_serial(1);            // close serial connection
```

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

```
ok = open_serial(1, 2, 115200);     // port 2, baudrate 115200
if ok ~= 0 then error('Unable to open serial port, please check'); end
for i = 1:20 //Run for 20 iterations
    val = cmd_analog_in(1, 4)        // read analog pin 4 (thermistor)
    disp(val);
    if(val > 550)                    // Setting Threshold value of 550
        cmd_digital_out(1, 3, 1) // Turn ON BUZZER
    else
        cmd_digital_out(1, 3, 0) // Turn OFF BUZZER
    end
    sleep(500);
end
cmd_digital_out(1, 3, 0) // Turn OFF BUZZER
close_serial(1);
```

8.5 Interfacing the thermistor from Xcos

In this section, we discuss how to read and use the thermistor values using Xcos blocks. The reader should go through the instructions given in Sec. 3.3 before getting started.

1. First we will read the thermistor values and display it. When the file required for this experiment is invoked, one gets the GUI as in Fig. 8.5. In the caption of this figure, one can see where to locate the file.

   As discussed in earlier chapters, we start with the initialization of the serial port. Next, using `Analog Read` block, we read the values of thermistor connected on analog pin 4. Next, we use a scope to plot the values coming from this pin. When this Xcos file is simulated, a plot is opened, as shown in Fig. 8.6.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 8.1. All other parameters are to be left unchanged.

   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 in the output plot, as shown in Fig. 8.6.

2. In the second experiment, we will switch on a buzzer depending on the thermistor readings (ADC output). When the file required for this experiment is

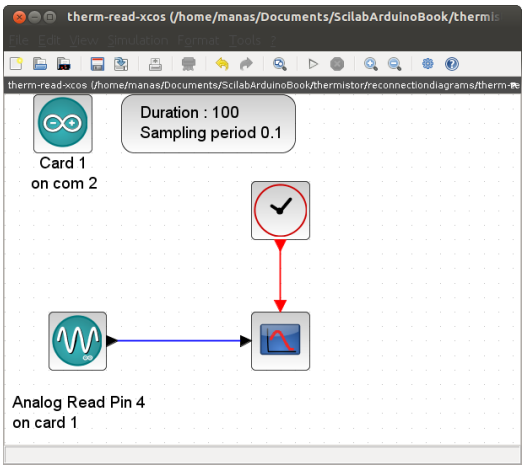


Figure 8.5: Xcos diagram to read thermistor values. This is what one sees when `0rigin/user-code/thermistor/scilab/therm-read.zcos`, see Footnote 2 on page 2, is invoked.

Table 8.1: Xcos parameters to read thermistor

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	100
	Sampling period(s)	0.1
ANALOG_READ_SB	Analog Pin	4
	Arduino card number	1
CSCOPE	Ymin	200
	Ymax	600
	Refresh period	100
CLOCK_c	Period	0.1
	Initialisation Time	0

invoked, one gets the GUI as in Fig. 8.7. In the caption of this figure, one can see where to locate the file.

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 8.2. In the CSCOPE_c block, the two values correspond to two graphs, one for digital write and other for analog read values. All other parameters are to be left

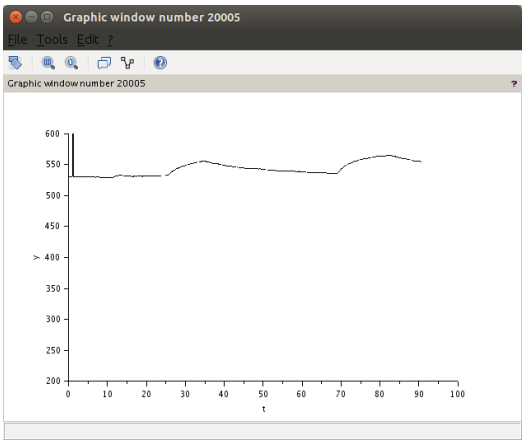


Figure 8.6: Plot window in Xcos to read thermistor values

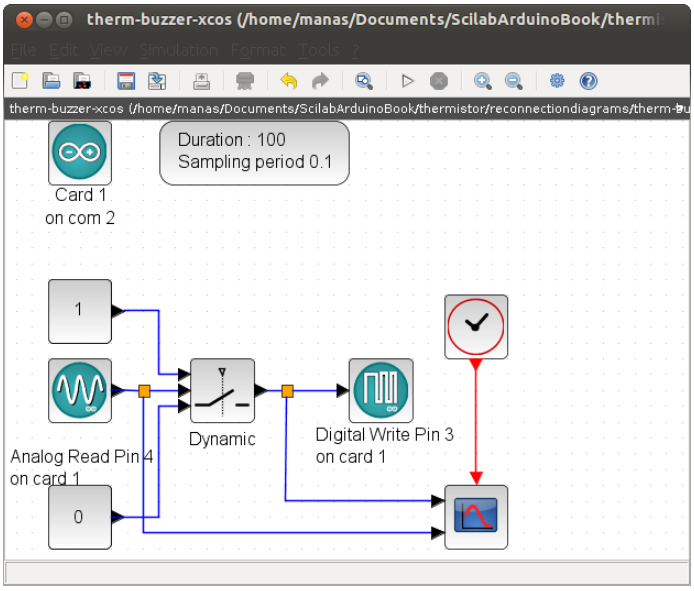


Figure 8.7: Xcos diagram to read the value of the thermistor, which is used to turn the buzzer on. This is what one sees when `Origin/user-code/thermistor/scilab/therm-buzzer.zcos`, see Footnote 2 on page 2, is invoked.

unchanged. When this Xcos file is simulated, a plot is opened, as shown in Fig. 8.8.

While running this experiment, the readers should try holding (or rubbing)

Table 8.2: Xcos parameters to read thermistor and switch the buzzer

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	100
	Sampling period(s)	0.1
ANALOG_READ_SB	Analog pin	4
	Arduino card number	1
CMSCOPE	Ymin	0 300
	Ymax	1 600
	Refresh period	100 100
CLOCK_c	Period	0.1
	Initialisation time	0
SWITCH2_m	Datatype	1
	threshold	550
	pass first input if field	0
	use zero crossing	1
DIGITAL_WRITE_SB	Digital pin	3
	Arduino card number	1

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 the Xcos file shown in Fig. 8.5.

8.6 Interfacing the thermistor from Python

8.6.1 Interfacing the thermistor

In this section, we discuss how to carry out the experiments of the previous section from Python. 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

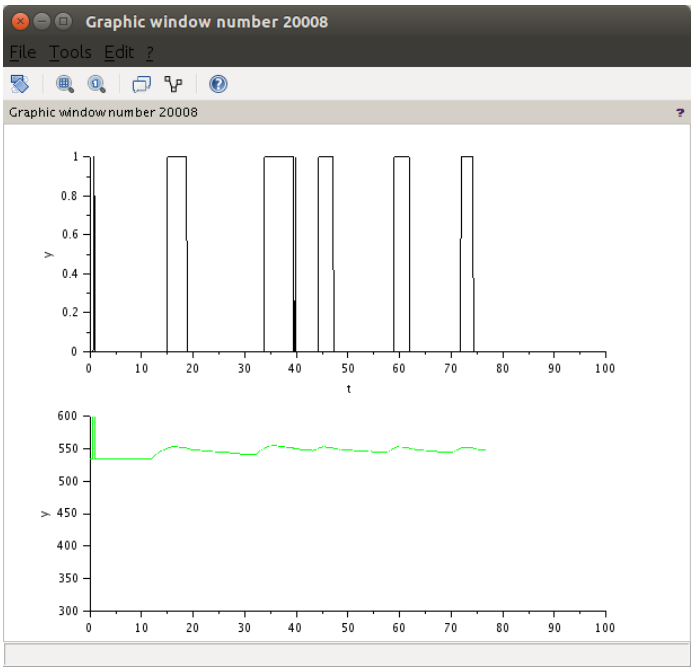


Figure 8.8: Plot window in Xcos to read thermistor values and the state of LED

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. 3.4 before getting started.

1. In the first experiment, we will read the thermistor values. The code for this experiment is given in Python Code 8.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules followed by setting up the serial port. Then, we read the input coming from analog pin 4 using the following command:

```
1        val = self.obj_arduino.cmd_analog_in(1, self.therm)
```

   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:

```
1        print(val)
```

   where `val` contains the thermistor values ranging from 0 to 1023. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 20 iterations.

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 values being printed on on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

2. This experiment is an extension of the previous experiment. Here, we will use a Python script to 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 Python 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 Python Code 8.2. For testing purposes, one may note the normal thermistor readings generated from the execution of Python 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 (int(val) > 550):
2         self.obj_arduino.cmd_digital_out(1, self.buzzer, 1)
3       else:
4         self.obj_arduino.cmd_digital_out(1, self.buzzer, 0)
5       sleep(0.5)
```

   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 Python Code 8.1.

8.6.2 Python Code

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

```
1  import os
2  import sys
3  cwd = os.getcwd()
4  (setpath,Examples) = os.path.split(cwd)
5  sys.path.append(setpath)
6
7  from Arduino.Arduino import Arduino
8  from time import sleep
9
10 class THERM_BUZZER:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.therm = 4
24
25     for i in range(20):
26       val = self.obj_arduino.cmd_analog_in(1, self.therm)
27       print(val)
28       sleep(0.5)
29
30   def exit(self):
31     self.obj_arduino.close_serial()
32
33 def main():
34   obj_pot = THERM_BUZZER(115200)
35
36 if __name__== '__main__':
37   main()
```

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class THERM_BUZZER:
11   def __init__(self, baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.therm = 4
24     self.buzzer = 3
25
26     for i in range(20):
27       val = self.obj_arduino.cmd_analog_in(1, self.therm)
28       print(val)
29
30       if (int(val) > 550):
31         self.obj_arduino.cmd_digital_out(1, self.buzzer, 1)
32       else:
33         self.obj_arduino.cmd_digital_out(1, self.buzzer, 0)
34       sleep(0.5)
35     self.obj_arduino.cmd_digital_out(1, self.buzzer, 0)
36
37
38   def exit(self):
39     self.obj_arduino.close_serial()
40
41 def main():
42   obj_pot = THERM_BUZZER(115200)
43
44 if __name__== '__main__':
45   main()
```

8.7 Interfacing the thermistor from Julia

8.7.1 Interfacing the thermistor

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 before getting started.

1. In the first experiment, we will read the thermistor values. The code for this experiment is given in Julia Code 8.1. As explained earlier in Sec. 4.7.1, we begin with importing the SerialPorts [18] package and the module ArduinoTools followed by setting up the serial port. Then, we read the input coming from analog pin 4 using the following command:

```
val = ArduinoTools.analogRead(ser, 4)
```

   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:

```
println(val)
```

   where `val` contains the thermistor values ranging from 0 to 1023. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 20 iterations.

   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 values being printed on on the Command Prompt (on Windows) or Terminal (on Linux), as the case maybe.

2. This experiment is an extension of the previous experiment. Here, we will use a Julia source file to 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 Julia 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 Julia Code 8.2. For testing purposes, one may note the normal thermistor readings generated from the execution of Julia 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)
2    ArduinoTools.digiWrite(ser, 3, 1)
3  else
4    ArduinoTools.digiWrite(ser, 3, 0)
5  end
```

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 Julia Code 8.1.

8.7.2 Julia Code

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

```
1 using SerialPorts
2 include("ArduinoTools.jl")
3
4 ser = ArduinoTools.connectBoard(115200)
5
6 for i = 1:20
7   val = ArduinoTools.analogRead(ser, 4)
```

```
8   println(val)
9   sleep(0.5)
10 end
11 close(ser)
```

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

```
1  using SerialPorts
2  include("ArduinoTools.jl")
3
4  ser = ArduinoTools.connectBoard(115200)
5  ArduinoTools.pinMode(ser, 3, "OUTPUT")
6  for i = 1:20
7    val = ArduinoTools.analogRead(ser, 4)
8    println(val)
9    if (val > 550)
10     ArduinoTools.digiWrite(ser, 3, 1)
11   else
12     ArduinoTools.digiWrite(ser, 3, 0)
13   end
14   sleep(0.5)
15 end
16 ArduinoTools.digiWrite(ser, 3, 0)
17 close(ser)
```

8.8 Interfacing the thermistor from OpenModelica

8.8.1 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. 3.6 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. 4.8.1, 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:

```
1     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. 3.43.

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:

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

8.8.2 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. 3.6.4. Please refer to Fig. 3.44 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.

```
model therm_read "Thermistor Readings"
  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, 4) "read analog pin 5 (ldr)";
    strm.print("Thermistor Readings: " + String(val));
    sComm.delay(500);
  end if;
  when terminal() then
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  annotation(
    experiment(StartTime = 0, StopTime = 20, Tolerance = 1e-6, Interval
    = 1));
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.

```
model therm_buzzer "Sound buzzer depending on thermistor 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
    strm.print("Unable to open serial port, please check");
  else
    val := sComm.cmd_analog_in(1, 4) "read analog pin 4";
    strm.print("Thermistor Readings: " + String(val));
    if val > 550 then
      digital_out := sComm.cmd_digital_out(1, 3, 1) "Turn ON Buzzer";
    else
      digital_out := sComm.cmd_digital_out(1, 3, 0) "Turn OFF Buzzer";
    end if;
    sComm.delay(500);
  end if;
  digital_out := sComm.cmd_digital_out(1, 3, 0) "Turn OFF Buzzer";
//for i in 1:500 loop
//end for;
//Run for 500 iterations
//Setting Threshold value of 500
  when terminal() then
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  annotation(
    experiment(StartTime = 0, StopTime = 10, Tolerance = 1e-6, Interval
    = 0.1));
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, Scilab scripts, Scilab Xcos, Python, Julia, and OpenModelica. 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[6]. 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 [21] on Arduino Uno. This book uses PWM pin 5 for this purpose. Rest two terminals (Vcc and ground)

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

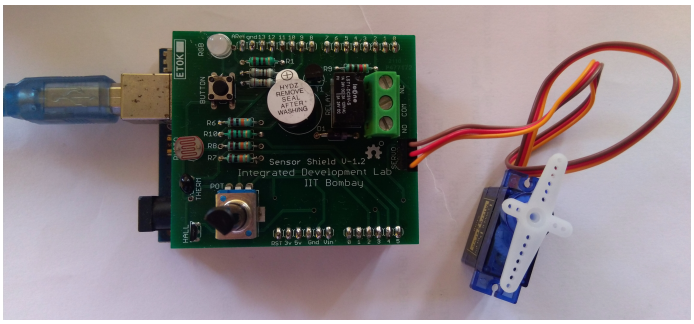


Figure 9.1: Connecting servomotor to the shield attached on Arduino Uno

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

need to be connected to 5V and GND on Arduino Uno. Table 9.1 summarizes these connections.

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 tuto-

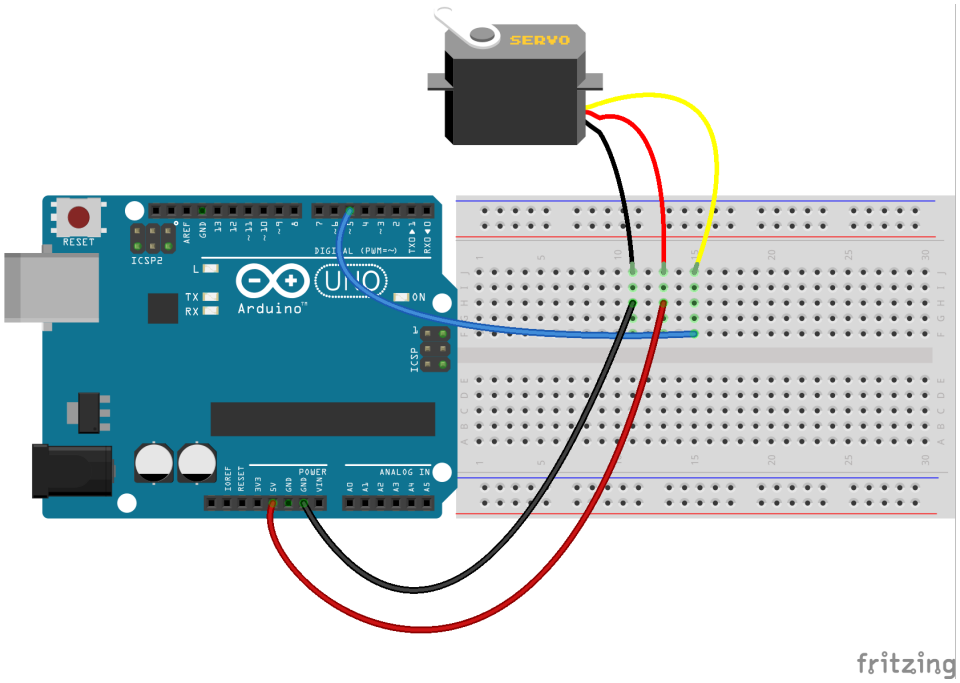


Figure 9.2: A servomotor with Arduino Uno using a breadboard

rials 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 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

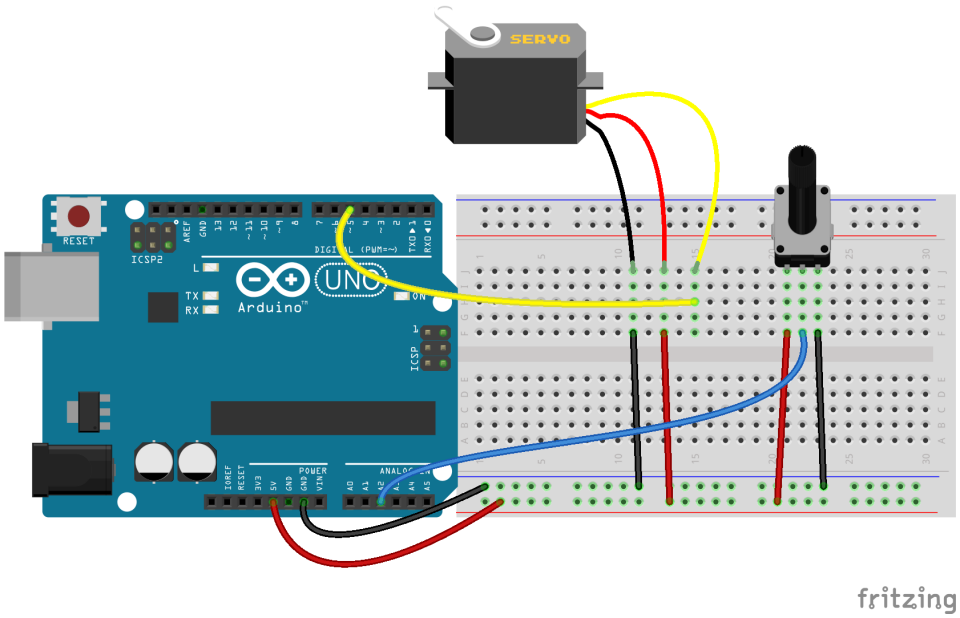


Figure 9.3: A servomotor and a potentiometer with Arduino Uno using a breadboard

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 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` [22]. 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(){
```

```
7   Serial.begin(115200);
8   myservo.attach(5);          // attach the servo object on to pin 5
9   for(i = 0; i < 50; ++i){
10  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 Scilab

9.4.1 Controlling the servomotor

In this section, we discuss how to carry out the experiments of the previous section from Scilab. 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. The first experiment makes the servomotor move by 30°. The code for this experiment is given in Scilab Code 9.1. As explained earlier in Sec. 4.4.1, we begin with serial port initialization. Next, we attach the servomotor by issuing the command given below:

   ```
   cmd_servo_attach(1, 1)  // To attach the motor to pin 5
   ```

   As shown above, the servomotor is attached on board 1 (the first argument) to pin 1 (the second argument). In the Scilab-Arduino toolbox discussed in Sec. 3.2.3, 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 1000 milliseconds:

   ```
   cmd_servo_move(1, 1, 30) // tell servo to rotate by 30 degrees
   sleep(1000)
   ```

   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 Scilab Code 9.2. In this code, we have added a delay of 1000 milliseconds between the two instances of moving the servomotor:

```
1 cmd_servo_move(1, 1, 90)          // Move the servo to 90 degree
2 sleep(1000) // be there for one second
3 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 Scilab 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 Scilab 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 = cmd_analog_in(1, 2)  // Read potntiometer value
2     val = floor(val*(180/1023)) // Scale Potentiometer value to
      0-180
3     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 Scilab Code

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

```
ok = open_serial(1, 2, 115200) // At port 2 with baud rate of 115200
if ok ~= 0 error('Check the serial port and try again'); end
cmd_servo_attach(1, 1)  // To attach the motor to pin 5
cmd_servo_move(1, 1, 30) // tell servo to rotate by 30 degrees
sleep(1000)
cmd_servo_detach(1, 1)          // Detach the motor
close_serial(1)
```

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

```
ok = open_serial(1, 2, 115200)  // Connect to Arduino at port 2
if ok ~= 0 error('Check the serial port and try again'); end
cmd_servo_attach(1, 1)         // Attach the motor to pin 5. 1 means 9
cmd_servo_move(1, 1, 90)        // Move the servo to 90 degree
sleep(1000) // be there for one second
cmd_servo_move(1, 1, 45)        // Move the servo to 45 degree
sleep(1000)      // be there for one second
cmd_servo_detach(1, 1)          // Detach the motor
close_serial(1)               // To close the connection safely
```

Scilab Code 9.3 Rotating the servomotor in steps of 20°. Available at `Origin/user-code/servo/scilab/servo-loop.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200); // At port 2 with baudrate of 115200
if ok ~= 0 error('Check the serial port and try again'); end
angle = 20;                      // Angle by which it has to move
for i = 0:10
    cmd_servo_attach(1, 1)        // Attach motor to pin 5. 1 means pin
     9.
    cmd_servo_move(1, 1, angle*i) // tell servo to rotate by 20 degrees
    sleep(1000)                  // waits for a sec
end
cmd_servo_detach(1, 1)          // Detach the motor
close_serial(1);               //To close the connection safely
```

Table 9.2: Parameters to rotate the servomotor by 30°

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
SERVO_WRITE_SB	Servo number	1
	Arduino card number	1
CONST_m	Constant value	30

Scilab Code 9.4 Rotating the servomotor to a degree specified by the potentiometer. Available at `Origin/user-code/servo/scilab/servo-pot.sce`, see Footnote 2 on page 2.

```
1 ok = open_serial(1, 2, 115200) // At port 2 with baud rate of 115200
2 if ok ~= 0 error('Check the serial port and try again'); end
3 cmd_servo_attach(1, 1) // Attach the motor to pin 5
4 for i=1:50                // 5,000 itterations
5     val = cmd_analog_in(1, 2)  // Read potntiometer value
6     val = floor(val*(180/1023)) // Scale Potentiometer value to 0-180
7     cmd_servo_move(1, 1, val) // Command the servo motor
8     sleep(500)              // sleep for 500 milliseconds
9 end
10 cmd_servo_detach(1, 1)// Detach the motor
11 close_serial(1)
```

9.5 Controling the servomotor through Xcos

In this section, we will see how to rotate the servomotor from Scilab Xcos. We will carry out experiments similar to the ones in earlier sections. For each, we will give the location of the zcos file and the parameters to set. The reader should go through the instructions given in Sec. 3.3 before getting started.

1. First we will rotate the servomotor by 30°. When the file required for this experiment is invoked, one gets the GUI as in Fig. 9.4. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 9.2. All other parameters are to be left unchanged.

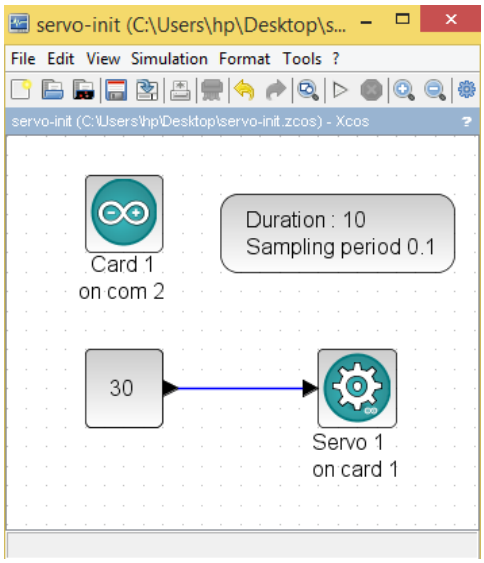


Figure 9.4: Rotating the servomotor by a fixed angle. This is what one sees when `Origin/user-code/led/scilab/servo-init.zcos`, see Footnote 2 on page 2, is invoked.

2. Next, we will rotate the servomotor by 90° and bring it to 45°, all absolute values. When the file required for this experiment is invoked, one gets the GUI as in Fig. 9.5. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 9.3. All other parameters are to be left unchanged.

3. Next, we will rotate the servomotor in increments of 20°. When the file required for this experiment is invoked, one gets the GUI as in Fig. 9.6. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 9.4. `Do on Overflow 0` means that we need to do nothing when there is an overflow. All other parameters are to be left unchanged.

4. Finally, we will use Xcos to rotate the servomotor as per the input received

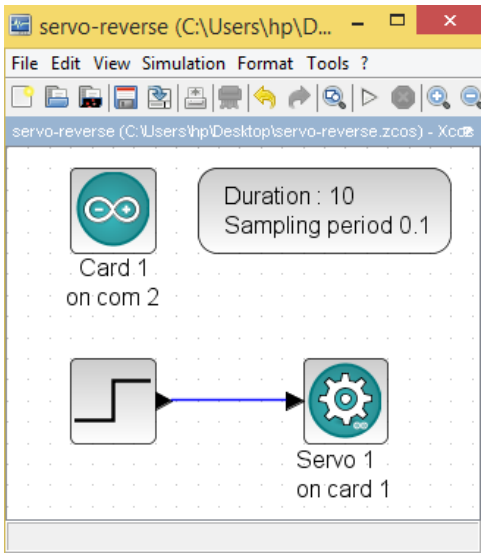


Figure 9.5: Rotating the servomotor forward and then reverse. This is what one sees when `Origin/user-code/led/scilab/servo-reverse.zcos`, see Footnote 2 on page 2, is invoked.

Table 9.3: Parameters to rotate the servomotor forward and reverse

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
SERVO_WRITE_SB	Servo number	1
	Arduino card number	1
STEP_FUNCTION	Step time	1
	Initial value	90
	Final value	45

from the potentiometer. When the file required for this experiment is invoked, one gets the GUI as in Fig. 9.7. In the caption of this figure, one can see where to locate the file.

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 9.5. All other

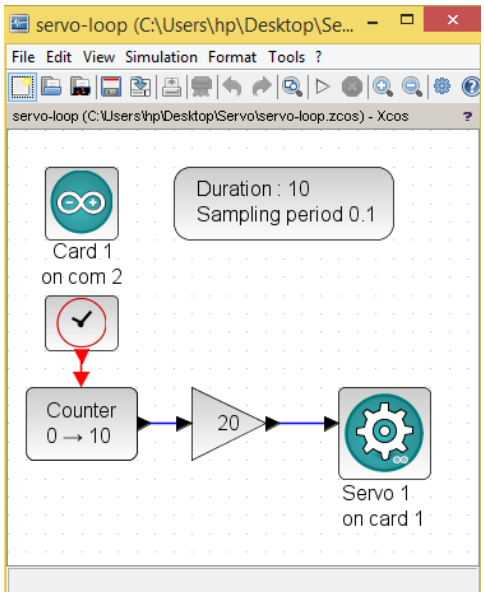


Figure 9.6: Rotating the servomotor in increments of 20°. This is what one sees when `Origin/user-code/led/scilab/servo-loop.zcos`, see Footnote 2 on page 2, is invoked.

Table 9.4: Parameters to make the servomotor to sweep the entire range in increments

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
SERVO_WRITE_SB	Servo number	1
CLOCK_c	Period	1
	Initialization time	0.1
Counter	Minimum value	0
	Maximum value	10
	Rule	1
GAINBLK	Gain	20
	Do on overflow	0

parameters are to be left unchanged. The `ANALOG_READ_SB` block reads the

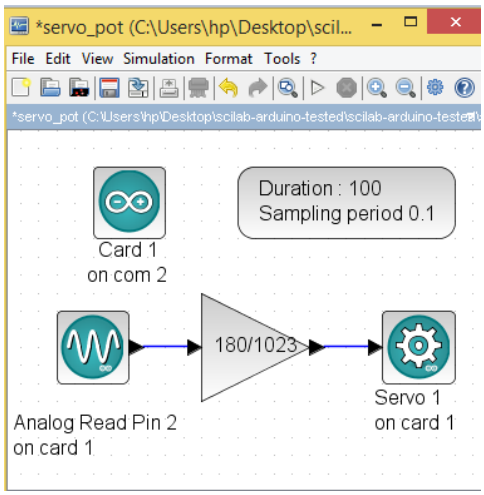


Figure 9.7: Rotating the servomotor as suggested by the potentiometer. This is what one sees when `Origin/user-code/led/scilab/servo-pot.zcos`, see Footnote 2 on page 2, is invoked.

Table 9.5: Parameters to rotate the servomotor based on the input from the potentiometer

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	The duration of acquisition(s)	100
	Sampling period(s)	0.1
SERVO_WRITE_SB	Servo number	1
ANALOG_READ_SB	Analog Pin	2
	Arduino card number	1
GAIN_f	Gain	180/1023

value of potentiometer. Next, `GAIN_f` is used to convert the potentiometer values into rotation angle by multiplying the values with 180/1023.

9.6 Controlling the servomotor through Python

9.6.1 Controlling the servomotor

In this section, we discuss how to carry out the experiments of the previous section from Python. 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. 3.4 before getting started.

1. The first experiment makes the servomotor move by 30°. The code for this experiment is given in Python Code 9.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules followed by setting up the serial port. Next, we attach the servomotor by issuing the command given below:

```
1       self.obj_arduino.cmd_servo_attach(1, 1)
```

   As shown above, the servomotor is attached on board 1 (the first argument) to pin 1 (the second argument). In the Python-Arduino toolbox discussed in Sec. 3.4.3, 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       self.obj_arduino.cmd_servo_move(1, 1, 30)
2       sleep(1)
```

   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 Python Code 9.2. In this code, we have added a delay of 1 second between the two instances of moving the servomotor:

```
1       self.obj_arduino.cmd_servo_move(1, 1, 90)
2       sleep(1)
3       self.obj_arduino.cmd_servo_move(1, 1, 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 Python 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 Python 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 = self.obj_arduino.cmd_analog_in(1, self.pot)
2         val = int(val*180/1023)
3         self.obj_arduino.cmd_servo_move(1, 1 ,val)
```

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.

9.6.2 Python Code

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
```

```
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class SERVO_INIT:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.pin1 = 5
24     self.obj_arduino.cmd_servo_attach(1, 1)
25     self.obj_arduino.cmd_servo_move(1, 1, 30)
26     sleep(1)
27     self.obj_arduino.cmd_servo_detach(1, 1)
28     sleep(1)
29
30
31   def exit(self):
32     self.obj_arduino.close_serial()
33
34 def main():
35   obj_servo = SERVO_INIT(115200)
36
37 if __name__== '__main__':
38   main()
```

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

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class SERVO_ANGULAR_ROTATION:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
```

```
13      self.setup()
14      self.run()
15      self.exit()
16
17    def setup(self):
18      self.obj_arduino = Arduino()
19      self.port = self.obj_arduino.locateport()
20      self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22    def run(self):
23      self.pin1 = 5
24      self.obj_arduino.cmd_servo_attach(1, 1)
25      self.obj_arduino.cmd_servo_move(1, 1, 90)
26      sleep(1)
27      self.obj_arduino.cmd_servo_move(1, 1, 45)
28      sleep(1)
29      self.obj_arduino.cmd_servo_detach(1, 1)
30
31
32    def exit(self):
33      self.obj_arduino.close_serial()
34
35  def main():
36    obj_servo = SERVO_ANGULAR_ROTATION(115200)
37
38  if __name__== '__main__':
39    main()
```

Python Code 9.3 Rotating the servomotor in steps of 20°. Available at `Origin/user-code/servo/python/servo-loop.py`, see Footnote 2 on page 2.

```
1  import os
2  import sys
3  cwd = os.getcwd()
4  (setpath, Examples)=os.path.split(cwd)
5  sys.path.append(setpath)
6
7  from Arduino.Arduino import Arduino
8  from time import sleep
9
10 class SERVO_INCR:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
```

```
19      self.port = self.obj_arduino.locateport()
20      self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22    def run(self):
23      self.pin1 = 5
24      self.obj_arduino.cmd_servo_attach(1, 1)
25      sleep(1)
26      self.angle = 20
27      for i in range(10):
28        self.obj_arduino.cmd_servo_move(1, 1, self.angle*i)
29        sleep(1)
30      self.obj_arduino.cmd_servo_detach(1, 1)
31
32    def exit(self):
33      self.obj_arduino.close_serial()
34
35 def main():
36    obj_servo=SERVO_INCR(115200)
37
38 if __name__=='__main__':
39    main()
```

Python Code 9.4 Rotating the servomotor to a degree specified by the potentiometer. Available at `Origin/user-code/servo/python/servo-pot.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
9
10 class SERVO_POT:
11    def __init__(self,baudrate):
12      self.baudrate = baudrate
13      self.setup()
14      self.run()
15      self.exit()
16
17    def setup(self):
18      self.obj_arduino = Arduino()
19      self.port = self.obj_arduino.locateport()
20      self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22    def run(self):
23      self.pin1 = 5
```

```
24      self.pot = 2
25      self.obj_arduino.cmd_servo_attach(1, 1)
26      for i in range(50):
27        val = self.obj_arduino.cmd_analog_in(1, self.pot)
28        val = int(val*180/1023)
29        self.obj_arduino.cmd_servo_move(1, 1 ,val)
30        sleep(0.5)
31      self.obj_arduino.cmd_servo_detach(1, 1)
32
33    def exit(self):
34      self.obj_arduino.close_serial()
35
36  def main():
37    obj_servo = SERVO_POT(115200)
38
39  if __name__=='__main__':
40    main()
```

9.7 Controlling the servomotor through Julia

9.7.1 Controlling the servomotor

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 before getting started.

1. The first experiment makes the servomotor move by 30°. The code for this experiment is given in Julia Code 9.1. As explained earlier in Sec. 4.7.1, we begin with importing necessary modules followed by setting up the serial port. Next, we attach the servomotor by issuing the command given below:

   ```
   ArduinoTools.ServoAttach(ser, 1)
   ```

   As shown above, the servomotor is attached on the serial port (the first argument) to pin 1 (the second argument). In the Julia-Arduino toolbox discussed in Sec. 3.5.3, 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:

   ```
   ArduinoTools.ServoMove(ser, 1, 30)
   ```

   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 Julia 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 1 second between the two instances of moving the servomotor:

```
ArduinoTools.ServoMove(ser, 1, 90)
sleep(1)
ArduinoTools.ServoMove(ser, 1, 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 Julia 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 Julia 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:

```
  val = ArduinoTools.analogRead(ser, 2)
  val = val*(180/1023)
  val = round(Int, floor(val))
  ArduinoTools.ServoMove(ser, 1, val)
```

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.

9.7.2 Julia Code

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.ServoAttach(ser, 1)
ArduinoTools.ServoMove(ser, 1, 30)
sleep(1)
close(ser)
```

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.ServoAttach(ser, 1)
ArduinoTools.ServoMove(ser, 1, 90)
sleep(1)
ArduinoTools.ServoMove(ser, 1, 45)
sleep(1)
close(ser)
```

Julia Code 9.3 Rotating the servomotor in steps of 20°. Available at `Origin/user-code/servo/julia/servo-loop.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.ServoAttach(ser, 1)
angle = 20
for i = 1:10
  ArduinoTools.ServoMove(ser, 1, angle*i)
  sleep(1)
end
```

```
11 ArduinoTools.ServoDetach(ser, 1)
12 close(ser)
```

Julia Code 9.4 Rotating the servomotor to a degree specified by the potentiometer. Available at `Origin/user-code/servo/julia/servo-pot.jl`, see Footnote 2 on page 2.

```
1 using SerialPorts
2 include("ArduinoTools.jl")
3
4 ser = ArduinoTools.connectBoard(115200)
5 ArduinoTools.ServoAttach(ser, 1)
6 for i = 1:50
7   val = ArduinoTools.analogRead(ser, 2)
8   val = val*(180/1023)
9   val = round(Int, floor(val))
10  ArduinoTools.ServoMove(ser, 1, val)
11  sleep(0.5)
12 end
13 ArduinoTools.ServoDetach(ser, 1)
14 close(ser)
```

9.8 Controlling the servomotor through OpenModelica

9.8.1 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. 3.6 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. 4.8.1, 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. 3.6.4, 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.

9.8.2 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. 3.6.4. Please refer to Fig. 3.44 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);
```

```
20   annotation(
21     experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
     = 5));
22 end servo_init;
```

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

```
1 model servo_reverse
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("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);
```

```
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) "Attach motor to pin 5. 1 means pin
    5.";
      sComm.delay(2000);
      angle := 20 "Angle by which it has to move";
      for i in 1:10 loop
        sComm.cmd_servo_move(1, 1, angle * i) "tell servo to rotate by
    20 degrees";
        sComm.delay(1000) "waits for a sec";
      end for;
      sComm.cmd_servo_detach(1, 1) "Detach the motor";
    end if;
    c_ok := sComm.close_serial(1) "To close the connection safely";
  end when;
  annotation(
    experiment(StartTime = 0, StopTime = 5, Tolerance = 1e-6, Interval
    = 5));
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.

```
model servo_pot "Control Servo Motor using Potentiometer"
  extends Modelica.Icons.Example;
  import sComm = Arduino.SerialCommunication.Functions;
  import strm = Modelica.Utilities.Streams;
  Integer ok(fixed = false);
  Integer c_ok(fixed = false);
  Integer val(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) "Attach the motor to pin 5";
      sComm.delay(2000);
      for i in 1:50 loop
        val := sComm.cmd_analog_in(1, 2) "Read potentiometer value";
        val := integer(val * 180 / 1023);
        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 Arduino IDE, Scilab scripts, Scilab Xcos, Python, Julia, and OpenModelica. 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)

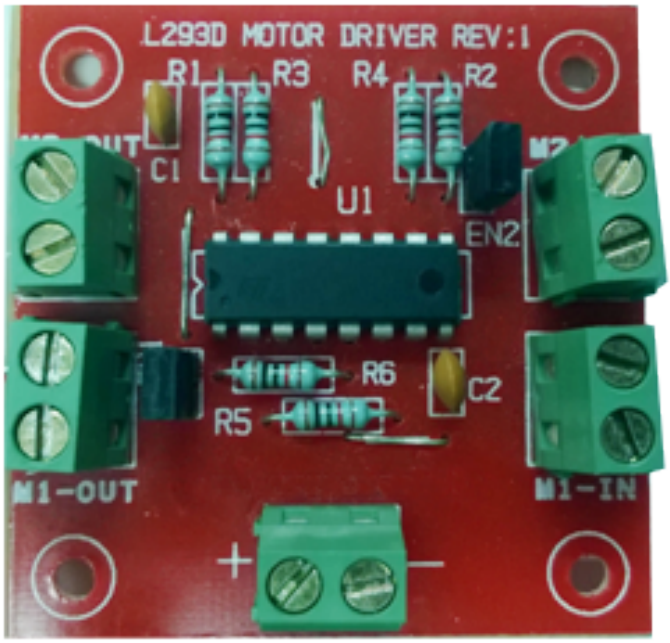


Figure 10.1: L293D motor driver board

Table 10.1: Values in the Scilab command for different H-Bridge circuits

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

form. PWM is a technique to generate analog voltages using digital pins. We know 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) [21]. 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 $2^8 = 256$

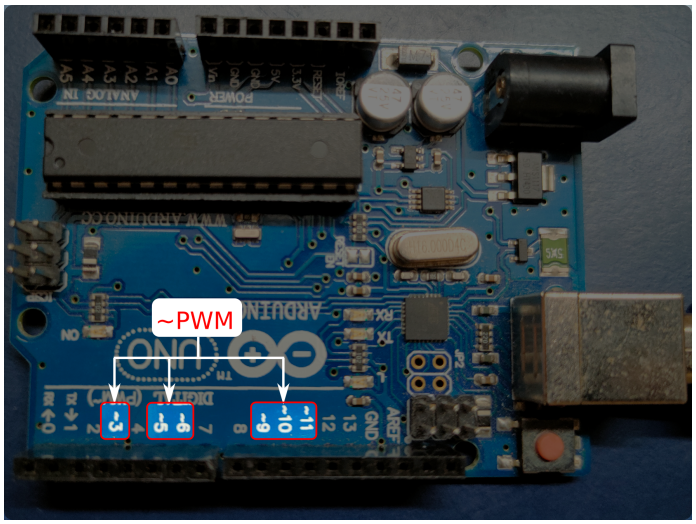


Figure 10.2: PWM pins on an Arduino Uno board

different analog values (from 0 to 255) in between 0-5V with these pins.

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

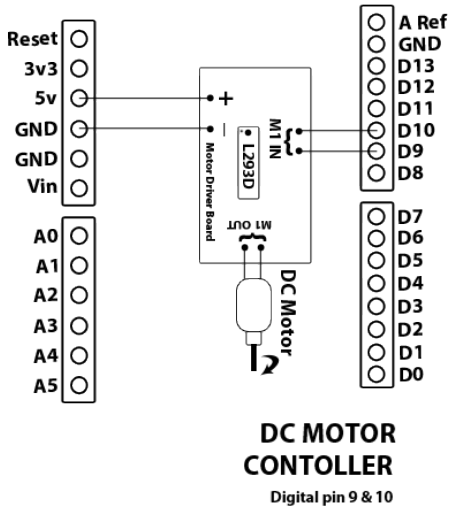


Figure 10.3: A schematic of DC motor connections

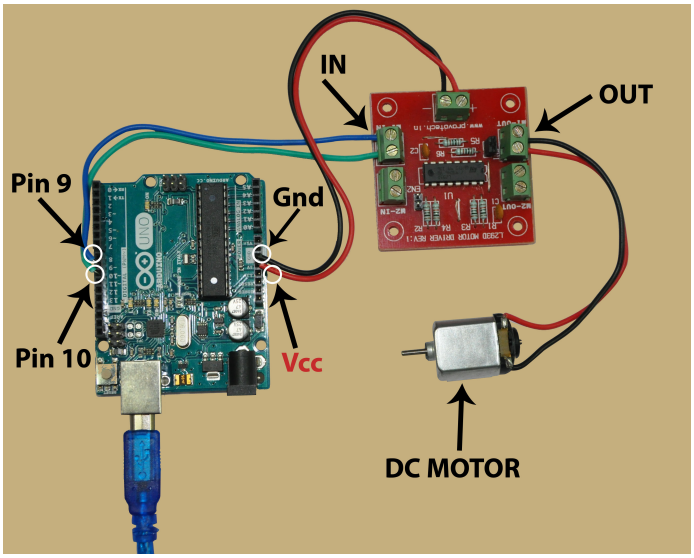


Figure 10.4: How to connect the DC motor to the Arduino Uno board

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

10.3.1 Controlling the DC motor

In this section, we discuss how to carry out the experiments of the previous section from Scilab. 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 Scilab. The code for this experiment is given in Scilab Code 10.1. As explained earlier in Sec. 4.4.1, we begin with serial port initialization. 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 Scilab-Arduino toolbox, as explained in Sec. 3.2.3, 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 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 Scilab 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 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 sleep(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 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 4 of Scilab 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 5 of Scilab Code 10.1 being commented), the DC motor will go on rotating. That's why, it may be inferred that line number 5 of Scilab Code 10.1 is mandatory for every program. We encourage the readers to run Scilab Code 10.1 by commenting any one or two of the lines numbered 4, 5 or 6. 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 Scilab 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 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:

```
cmd_dcmotor_release(1, 1)        // Motor 1 is released
close_serial(1);
```

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 Scilab 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

Exercise 10.2 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.

10.3.2 Scilab Code

Scilab Code 10.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/scilab/dcmotor-clock.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200)    //COM port is 2 and baud rate is
    115200
cmd_dcmotor_setup(1, 3, 1, 9, 10)  // Setup DC motor of type 3 (L293D),
     motor 1, pin 9 and 10
cmd_dcmotor_run(1, 1, 100) // Motor 1 runs at PWM 100
sleep(3000)                // This is allowed to continue for 3 seconds
cmd_dcmotor_release(1, 1) // Motor 1 is released
close_serial(1);
```

Scilab Code 10.2 Rotating DC motor in both directions. Available at `Origin/user-code/dcmotor/scilab/dcmotor-both.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1, 2, 115200)    //COM port is 2 and baud rate is
    115200
cmd_dcmotor_setup(1, 3, 1, 9, 10) // Setup DC motor of type 3 (L293D),
    motor 1, pin 9 and 10
cmd_dcmotor_run(1, 1, 100)        // Motor 1 runs at PWM 100
sleep(3000)                       // for 3 seconds
cmd_dcmotor_run(1, 1, -100)       // Motor 1 runs at PWM -100 in reverse
    direction
sleep(2000)                       // for 2 seconds
cmd_dcmotor_release(1, 1)         // Motor 1 is released
close_serial(1);
```

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

```
ok = open_serial(1, 2, 115200)//COM port is 2 and baud rate is 115200
if ok ~= 0, error('Serial port is not accesible'); end
cmd_dcmotor_setup(1, 3, 1, 9, 10) // Setup DC motor of type 3 (L293D),
    motor 1, pins 9 and 10
for i = 1:4
    cmd_dcmotor_run(1, 1, 100)  // Motor 1 runs at PWM 100
    sleep(3000)                 // for 3 seconds
    cmd_dcmotor_run(1, 1, 0)    // Halt the motor
    sleep(2000)                 // for 2 seconds
    cmd_dcmotor_run(1, 1, -100) // Run it at PWM 100 in reverse
    direction
    sleep(2000)                 // for 2 seconds
end
cmd_dcmotor_release(1, 1)       // Motor 1 is released
close_serial(1);
```

10.4 Controlling the DC motor from Xcos

In this section, we will see how to drive the DC motor from Scilab Xcos. We will carry out the same three experiments as in the previous sections. For each experiment, we will give the location of the zcos file and the parameters to set. The reader should go through the instructions given in Sec. 3.3 before getting started.

1. First we will see a simple code that drives the DC motor for a specified time. When the file required for this experiment is invoked, one gets the GUI as in Fig. 10.5. In the caption of this figure, one can see where to locate the file.

   We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 10.2. In case

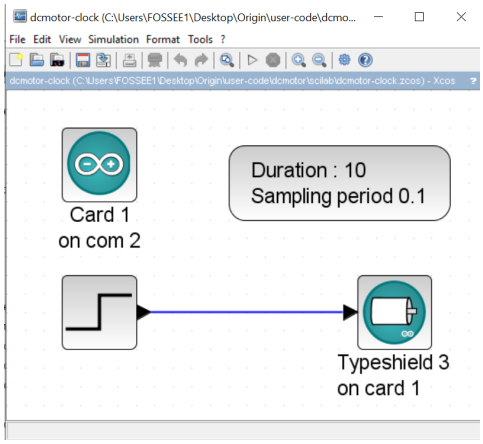


Figure 10.5: Control of DC motor for a specified time from Xcos. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-clock.zcos`, see Footnote 2 on page 2, is invoked.

Table 10.2: Xcos parameters to drive the DC motor for a specified time

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DCMOTOR_SB	Type of Shield	3
	Arduino card number	1
	PWM pin numbers	9 10
	Motor number	1
STEP_FUNCTION	Step time	5
	Initial Value	100
	Final Value	0

of `DCMOTOR_SB`, enter 3 to indicate for L293D board. After clicking on OK, another dialog box will pop up. In that, enter the PWM pin numbers as 9 and 10 and click OK. All other parameters are to be left unchanged.

2. Next, we will describe the Xcos code that drives the DC motor in both forward and reverse directions. When the file required for this experiment is invoked, one gets the GUI as in Fig. 10.6. In the caption of this figure, one can see where to locate the file.

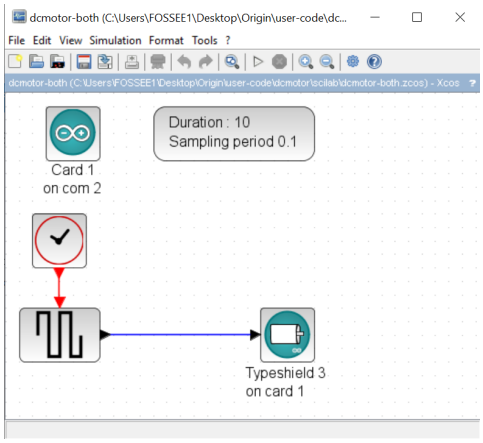


Figure 10.6: Xcos control of the DC motor in forward and reverse directions. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-both.zcos`, see Footnote 2 on page 2, is invoked.

Table 10.3: Xcos parameters to drive the DC motor in forward and reverse directions

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DCMOTOR_SB	Type of Shield	3
	Arduino card number	1
	PWM pin numbers	9 10
	Motor number	1
STEP_FUNCTION	Step time	5
	Initial Value	100
	final value	0
CLOCK_c	Period	1
	Initialisation Time	0.1

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 10.3. All other parameters are to be left unchanged.

3. Next, we will describe the Xcos code that drives the DC motor in a loop.

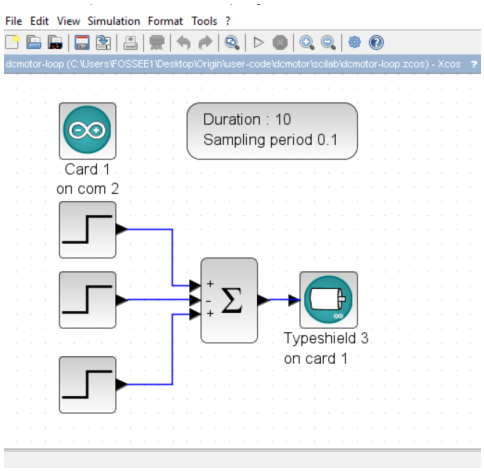


Figure 10.7: Xcos control of the DC motor in both directions in a loop. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-loop.zcos`, see Footnote 2 on page 2, is invoked.

When the file required for this experiment is invoked, one gets the GUI as in Fig. 10.7. In the caption of this figure, one can see where to locate the file.

We will next explain how to set the parameters for this simulation. To set value on any block, one needs to right click and open the `Block Parameters` or double click. The values for each block is tabulated in Table 10.4. All other parameters are to be left unchanged.

Exercise 10.3 Carry out the following exercise:

1. Keep reducing the PWM value and find out the minimum value required to run the DC motor. Is this value in agreement with what we found in the previous section?
2. Change the PWM value to -100 and check if the DC motor rotates in the opposite direction.
3. Find out the smallest PWM value required to make the motor run in the opposite direction. That is, find the least count for both directions.
4. Come up with a method to rotate the motor in two directions for different time periods.

∎

Table 10.4: Xcos parameters to drive the DC motor in a loop

Name of the block	Parameter name	Value
ARDUINO_SETUP	Identifier of Arduino Card	1
	Serial com port number	2, see Footnote 4 on page 30
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DCMOTOR_SB	Type of Shield	3
	Arduino card number	1
	PWM pin numbers	9 10
	Motor number	1
STEP_FUNCTION 1	Step time	3
	Initial Value	100
	Final Value	0
STEP_FUNCTION 2	Step time	5
	Initial Value	0
	Final Value	100
STEP_FUNCTION 3	Step time	7
	Initial Value	0
	Final Value	100
BIGSOM_f	Inputs ports signs/gain	[1;-1;1]

10.5 Controlling the DC motor from Python

10.5.1 Controlling the DC motor

In this section, we discuss how to carry out the experiments of the previous section from Python. 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. 3.4 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 Python. The code for this experiment is given in Python Code 10.1. As explained earlier in Sec. 4.6.1, we begin with importing necessary modules 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 Python-Arduino toolbox, as explained in Sec. 3.4.3, 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 self.obj_arduino.cmd_dcmotor_setup(1, 3, 1, self.pin1, self.
  pin2)
```

The next line of Python 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 self.obj_arduino.cmd_dcmotor_run(1, 1, 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 sleep(3)
```

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

```
1 self.obj_arduino.cmd_dcmotor_release(1, 1)
```

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 29 of Python 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 30 of Python Code 10.1 being commented), the DC motor will go on rotating. That's why, it may be inferred that line number 30 of Python Code 10.1 is mandatory for every program. We encourage the readers to run Python Code 10.1 by commenting any one or two of the lines numbered 29 and 30. 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 Python 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     self.obj_arduino.cmd_dcmotor_run(1, 1, -100)
```

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

```
1     self.obj_arduino.cmd_dcmotor_release(1, 1)
```

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 Python 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

10.5.2 Python Code

Python Code 10.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/python/dcmotor-clock.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 import os
```

```
4 import sys
5 cwd = os.getcwd()
6 (setpath, Examples) = os.path.split(cwd)
7 sys.path.append(setpath)
8
9 from Arduino.Arduino import Arduino
10 from time import sleep
11
12 class DCMOTOR_ROTATION:
13   def __init__(self,baudrate):
14     self.baudrate = baudrate
15     self.setup()
16     self.run()
17     self.exit()
18
19   def setup(self):
20     self.obj_arduino = Arduino()
21     self.port = self.obj_arduino.locateport()
22     self.obj_arduino.open_serial(1, self.port, self.baudrate)
23
24   def run(self):
25     self.pin1 = 9
26     self.pin2 = 10
27     self.obj_arduino.cmd_dcmotor_setup(1, 3, 1, self.pin1, self.pin2)
28     self.obj_arduino.cmd_dcmotor_run(1, 1, 100)
29     sleep(3)
30     self.obj_arduino.cmd_dcmotor_release(1, 1)
31
32
33   def exit(self):
34     self.obj_arduino.close_serial()
35
36 def main():
37   obj_dcmotor = DCMOTOR_ROTATION(115200)
38
39 if __name__=='__main__':
40   main()
```

Python Code 10.2 Rotating DC motor in both directions. Available at `Origin/user-code/dcmotor/python/dcmotor-both.py`, see Footnote 2 on page 2.

```
1 import os
2 import sys
3 cwd = os.getcwd()
4 (setpath, Examples) = os.path.split(cwd)
5 sys.path.append(setpath)
6
7 from Arduino.Arduino import Arduino
8 from time import sleep
```

```
9
10 class DCMOTOR_ROTATION:
11   def __init__(self,baudrate):
12     self.baudrate = baudrate
13     self.setup()
14     self.run()
15     self.exit()
16
17   def setup(self):
18     self.obj_arduino = Arduino()
19     self.port = self.obj_arduino.locateport()
20     self.obj_arduino.open_serial(1, self.port, self.baudrate)
21
22   def run(self):
23     self.pin1 = 9
24     self.pin2 = 10
25     self.obj_arduino.cmd_dcmotor_setup(1, 3, 1, self.pin1, self.pin2)
26     self.obj_arduino.cmd_dcmotor_run(1, 1, 100)
27     sleep(3)
28     self.obj_arduino.cmd_dcmotor_run(1, 1, -100)
29     sleep(2)
30     self.obj_arduino.cmd_dcmotor_release(1, 1)
31
32   def exit(self):
33     self.obj_arduino.close_serial()
34
35 def main():
36   obj_dcmotor = DCMOTOR_ROTATION(115200)
37
38 if __name__ == '__main__':
39   main()
```

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

```
1 import os
2 import sys
3 import os
4 import sys
5 cwd = os.getcwd()
6 (setpath, Examples) = os.path.split(cwd)
7 sys.path.append(setpath)
8
9 from Arduino.Arduino import Arduino
10 from time import sleep
11
12 class DCMOTOR_ROTATION:
13   def __init__(self,baudrate):
```

```
14      self.baudrate = baudrate
15      self.setup()
16      self.run()
17      self.exit()
18
19    def setup(self):
20      self.obj_arduino = Arduino()
21      self.port = self.obj_arduino.locateport()
22      self.obj_arduino.open_serial(1, self.port, self.baudrate)
23
24    def run(self):
25      self.pin1 = 9
26      self.pin2 = 10
27      for i in range(4):
28        self.obj_arduino.cmd_dcmotor_setup(1, 3, 1, self.pin1, self.pin2)
29        self.obj_arduino.cmd_dcmotor_run(1, 1, 100)
30        sleep(3)
31        self.obj_arduino.cmd_dcmotor_run(1, 1, 0)
32        sleep(2)
33        self.obj_arduino.cmd_dcmotor_run(1, 1, -100)
34        sleep(2)
35      self.obj_arduino.cmd_dcmotor_release(1, 1)
36
37    def exit(self):
38      self.obj_arduino.close_serial()
39
40  def main():
41    obj_dcmotor = DCMOTOR_ROTATION(115200)
42
43  if __name__ == '__main__':
44    main()
```

10.6 Controlling the DC motor from Julia

10.6.1 Controlling the DC motor

In this section, we discuss how to carry out the experiments of the previous section from Julia. 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. 3.5 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 Julia. The code for this experiment is given in Julia Code 10.1. As explained earlier in Sec. 4.7.1, we begin with importing necessary modules followed by setting up the serial port. Next, the code has a command of the following form:

```
DCMotorSetup(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 Julia-Arduino toolbox, as explained in Sec. 3.5.3, 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 `DCMotorSetup` becomes

```
ArduinoTools.DCMotorSetup(ser, 3, 1, 9, 10)
```

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

```
DCMotorRun(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 `DCMotorRun` becomes

```
ArduinoTools.DCMotorRun(ser, 1, 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:

```
sleep(3)
```

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

```
1 ArduinoTools.DCMotorRelease(ser, 1)
```

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 7 of Julia 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 8 of Julia Code 10.1 being commented), the DC motor will go on rotating. That's why, it may be inferred that line number 8 of Julia Code 10.1 is mandatory for every program. We encourage the readers to run Julia Code 10.1 by commenting any one or two of the lines numbered 7 and 8. 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 Julia 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 `DCMotorRun`, as shown below:

```
1 ArduinoTools.DCMotorRun(ser, 1, -100)
```

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

```
1 ArduinoTools.DCMotorRelease(ser, 1)
2 close(ser)
```

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 Julia 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 `DCMotorRelease`, followed by closing the serial port.

10.6.2 Julia Code

Julia Code 10.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/julia/dcmotor-clock.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.DCMotorSetup(ser, 3, 1, 9, 10)
ArduinoTools.DCMotorRun(ser, 1, 100)
sleep(3)
ArduinoTools.DCMotorRelease(ser, 1)
close(ser)
```

Julia Code 10.2 Rotating DC motor in both directions. Available at `Origin/user-code/dcmotor/julia/dcmotor-both.jl`, see Footnote 2 on page 2.

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.DCMotorSetup(ser, 3, 1, 9, 10)
ArduinoTools.DCMotorRun(ser, 1, 100)
sleep(3)
ArduinoTools.DCMotorRun(ser, 1, -100)
sleep(2)
ArduinoTools.DCMotorRelease(ser, 1)
close(ser)
```

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

```
using SerialPorts
include("ArduinoTools.jl")

ser = ArduinoTools.connectBoard(115200)
ArduinoTools.DCMotorSetup(ser, 3, 1, 9, 10)
for i = 1:4
  ArduinoTools.DCMotorRun(ser, 1, 100)
  sleep(3)
  ArduinoTools.DCMotorRun(ser, 1, 0)
  sleep(2)
  ArduinoTools.DCMotorRun(ser, 1, -100)
  sleep(2)
end
ArduinoTools.DCMotorRelease(ser, 1)
close(ser)
```

10.7 Controlling the DC motor from OpenModelica

10.7.1 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. 3.6 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. 4.8.1, 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. 3.6.4, 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:

```
1       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.

10.7.2 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. 3.6.4. Please refer to Fig. 3.44 to know how to locate the experiments.

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

```
1  model dcmotor_clock "Rotate DC Motor clockwise"
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.delay(1000);
15       sComm.cmd_dcmotor_setup(1, 3, 1, 9, 10) "Setup DC motor of type 3
       (L293D), motor 1, pin 9 and 10";
16       sComm.cmd_dcmotor_run(1, 1, 100) "Motor 1 runs at PWM 100";
```

```
17      sComm.delay(3000) "This is allowed to continue for 3 seconds";
18      sComm.cmd_dcmotor_release(1, 1) "Motor 1 is released";
19    end if;
20    c_ok := sComm.close_serial(1) "To close the connection safely";
21  end when;
22  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 [23] [24]. 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 [25]. 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

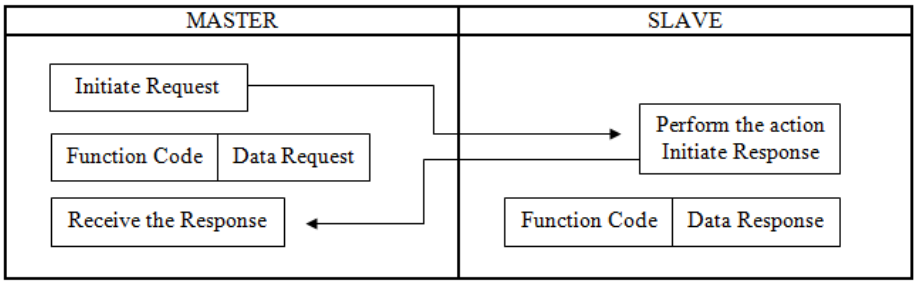


Figure 11.1: Block diagram representation of the Protocol

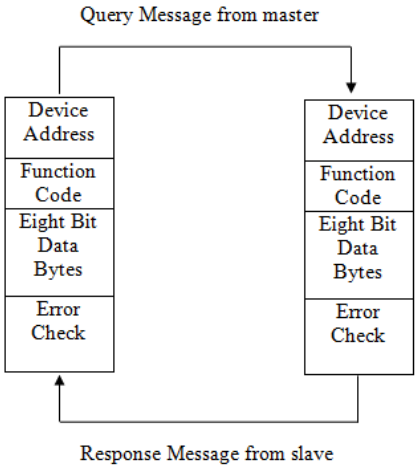


Figure 11.2: Master-Slave Query-Response Cycle

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 [25]. 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

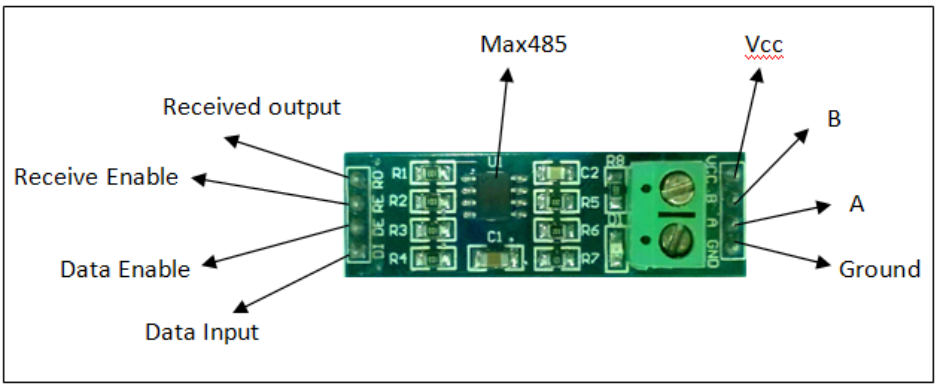


Figure 11.3: Pins in RS485 module

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 [26] 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,

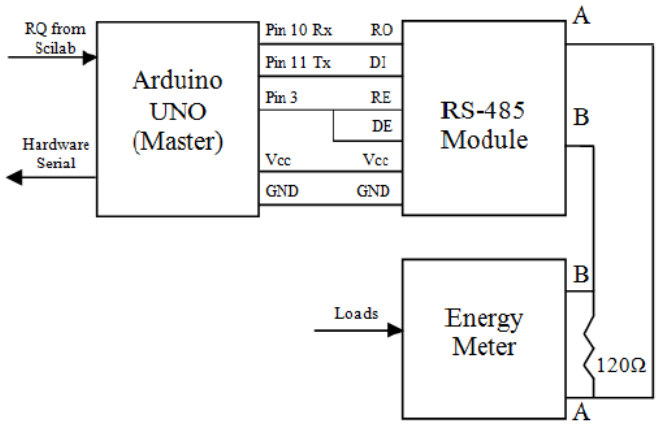


Figure 11.4: Block diagram for reading the parameters in energy meter

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

There are several online converters [27] 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.

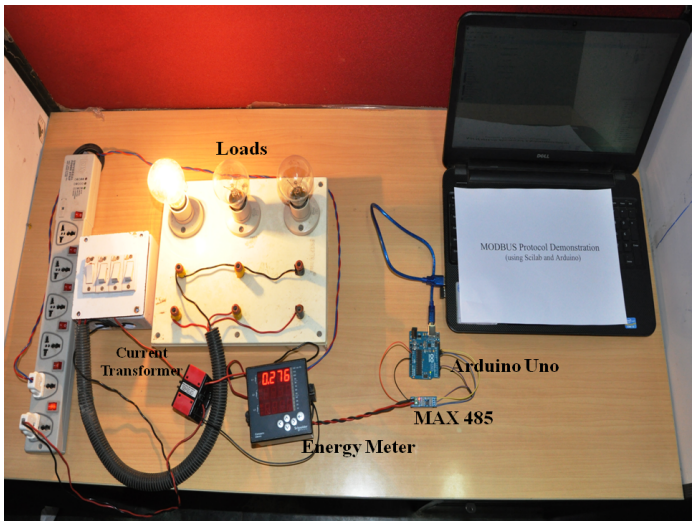


Figure 11.5: Experimental set up for reading energy meter

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

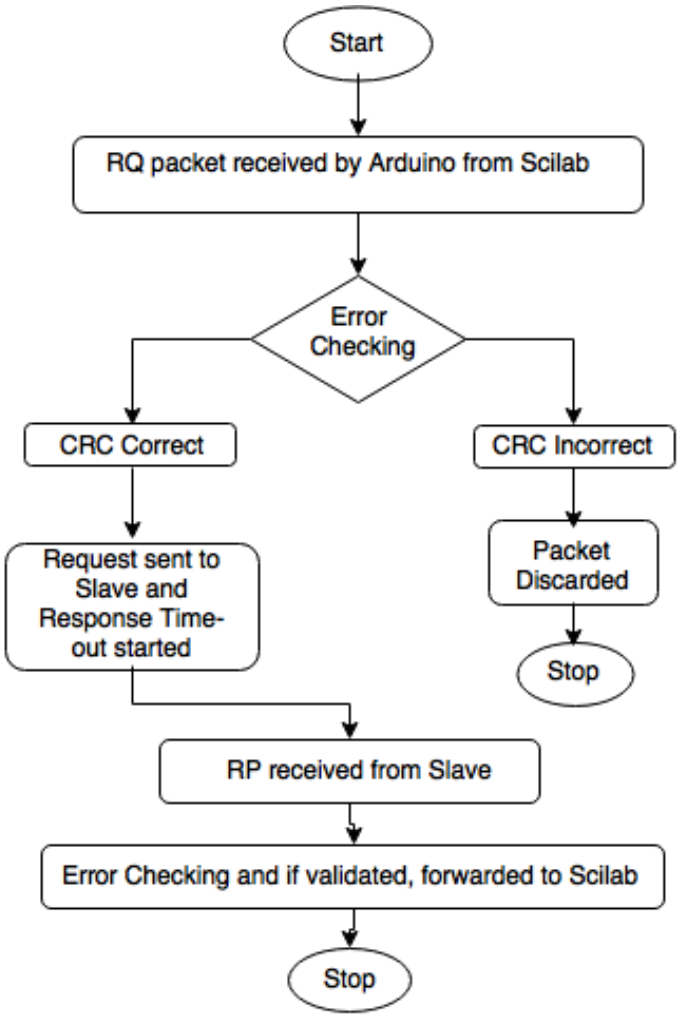


Figure 11.6: Flowchart of Arduino firmware

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.

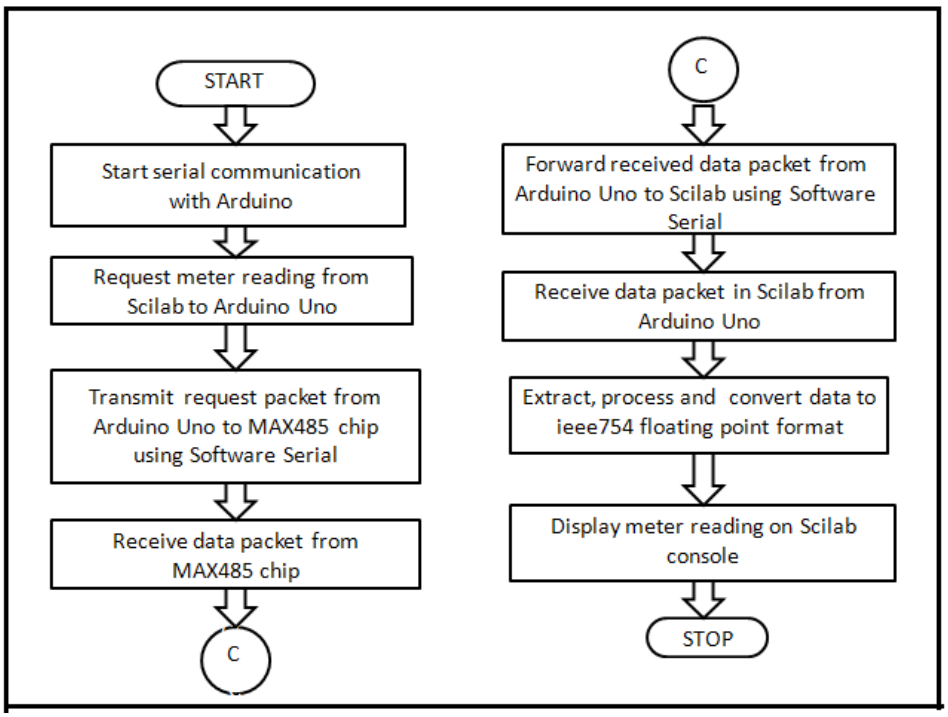


Figure 11.7: Flowchart of the steps happening in the FLOSS code

(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 Scilab

The objective of this experiment is to make the user acquainted with the demonstration of Modbus protocol through the Scilab-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 Scilab-Arduino toolbox. For data transmission, we have used an RS485 module.

Scilab 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. Scilab 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.

11.5 Reading the electrical parameters from Scilab

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 Scilab 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 Scilab Code

Scilab Code 11.1 First 10 lines of the Scifunc block function. Available at `Origin/user-code/modbus/scilab`, see Footnote 2 on page 2.

```
1
2 //voltage
3 function p=read_val(addr_byte)
4 h=open_serial(1,2,9600)
5
6 //for x=1:5
7     //arr=[ascii(01) ascii(03) ascii(15) ascii(86) ascii(00) ascii(02)
      ascii(39) ascii(15)];
8     //arr1=string(arr)
9     if(addr_byte==86)
10    array1=ascii(01)+ ascii(03) + ascii(15) + ascii(86) + ascii(00) +
      ascii(02) + ascii(39) + ascii(15)
```

Scilab Code 11.2 First 10 lines of the code for Single Phase Current Output. Available at `Origin/user-code/modbus/scilab`, see Footnote 2 on page 2.

```
1 //current
2 //function read_current()
3 function read_current()
4
5 h=open_serial(1, 2, 9600)
6 //for x=1:5
7     arr=[ascii(01) ascii(03) ascii(15) ascii(88) ascii(00) ascii(02)
      ascii(70) ascii(204)];
8
9      arr1=ascii(01) + ascii(03) + ascii(15) + ascii(88) + ascii(00) +
      ascii(02) + ascii(70) + ascii(204);
10 write_serial(1, arr1, 8);
```

Scilab Code 11.3 First 10 lines of the code for Single Phase Voltage Output. Available at `Origin/user-code/modbus/scilab`, see Footnote 2 on page 2.

```
1 //voltage
2 function read_voltage()
3 //endfunction
4 h=open_serial(1,2,9600)
5
6 //for x=1:5
7     arr=[ascii(01) ascii(03) ascii(15) ascii(86) ascii(00) ascii(02)
      ascii(39) ascii(15)];
8     //arr1=string(arr)
9     aac=ascii(01)+ ascii(03) + ascii(15) + ascii(86) + ascii(00) +
      ascii(02) + ascii(39) + ascii(15)
10    write_serial(1,aac,8);
```

Scilab Code 11.4 First 10 lines of the code for Single Phase Active Power Output. Available at `Origin/user-code/modbus/scilab`, see Footnote 2 on page 2.

```
1 //energy
2 function read_active_power()
3
4 h=open_serial(1,2,9600)
5 //for x=1:5
6     arr=[ascii(01) ascii(03) ascii(15) ascii(78) ascii(00) ascii(02)
      ascii(167) ascii(08)];
7      asc1=ascii(01) + ascii(03) + ascii(15) + ascii(78) + ascii(00) +
      ascii(02) + ascii(167) + ascii(08);
8
9 write_serial(1,asc1,8);
```

Note: After we send the RQ from Scilab to the energy meter, we will receive an RP from the energy meter. RP contains the data requested. This data is read serially in Scilab and the bytes so received are stored in a variable. On analyzing the bytes received, we observe some blank spaces received along with the data. So the required data starts from the fourth byte available, excluding spaces. For example, If there are `n` spaces received before the packet, so the required data would be available from `(n+4)`th position onwards. It means that we have to analyze the four bytes starting at `(n+4)`th position. Note that the RP may have one or more spaces at the start or the end. That's why we may have to shift our index to extract the desired data.

11.5.3 Output in the Scilab Console

In this section, we present the results. In this experiment, the three parameters: voltage, current, and active power in the energy meter have been accessed and displayed on the Scilab console. For each of these three parameters, we present two

```
Scilab 5.5.2 Console

-->read_current()

 Current(in A)=

    0.2749688

-->
```

Figure 11.8: Single phase current output on Scilab Console

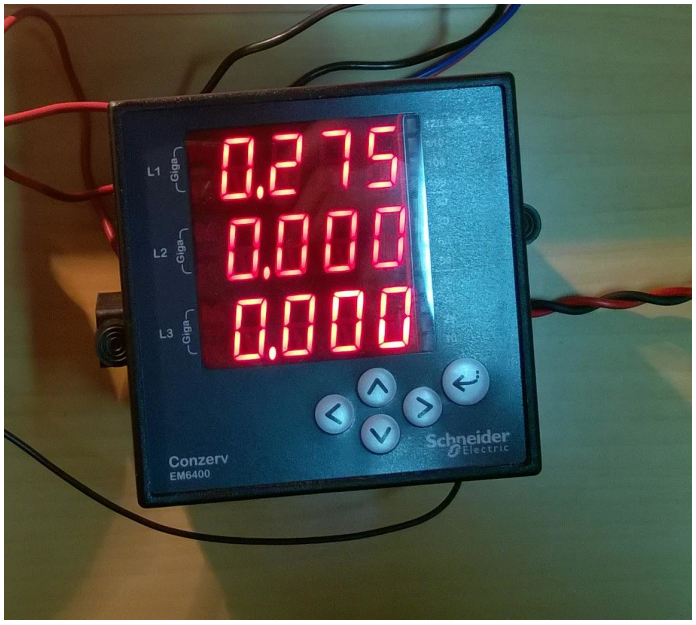


Figure 11.9: Single phase current output in energy meter

image: one showing the reading being shown in the energy meter and the another showing the value being displayed in the Scilab Console.

1. Single phase current output: Fig. 11.8 and Fig. 11.9 show Scilab code output of current in Amperes and corresponding snapshot of energy meter display with a single load rated 60W-230V.

2. Single phase voltage output: Fig. 11.10 and Fig. 11.11 show Scilab code output

```
Scilab 5.5.2 Console

-->read_voltage()

 Voltage(in V)=

    244.10999

-->
```

Figure 11.10: Single phase voltage output on Scilab Console

Figure 11.11: Single phase voltage output in energy meter

of voltage in Volts and corresponding snapshot of energy meter display with a single load rated 60W-230V.

3. Single phase active power output: Fig. 11.12 and Fig. 11.13 show Scilab code output of active power in Watts and corresponding snapshot of energy meter

```
Scilab 5.5.2 Console

-->read_active_power()

 Active Power(in W)=

    64.40033

-->
```

Figure 11.12: Single phase active power output on Scilab Console

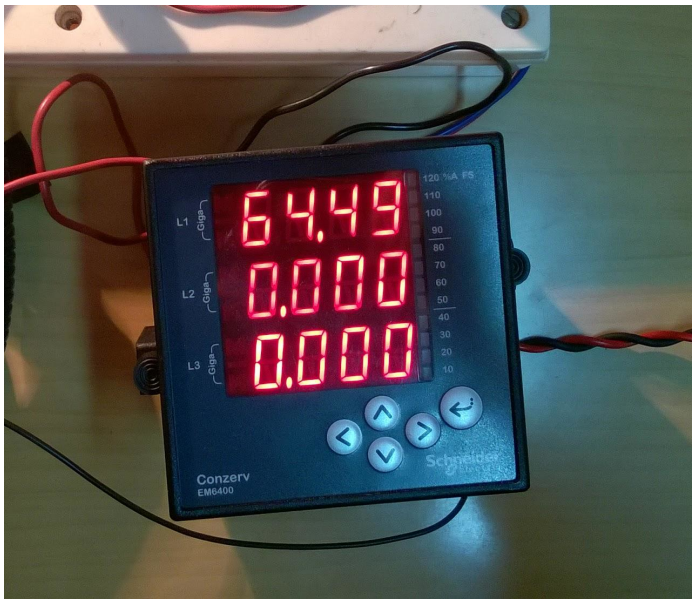


Figure 11.13: Single phase active power output in energy meter

display with a single load rated 60W-230V.

11.6 Reading the electrical parameters from Xcos

In this section we will carry out the same experiments discussed in the previous sections but through Xcos. One should go through Sec. 3.3 before continuing.

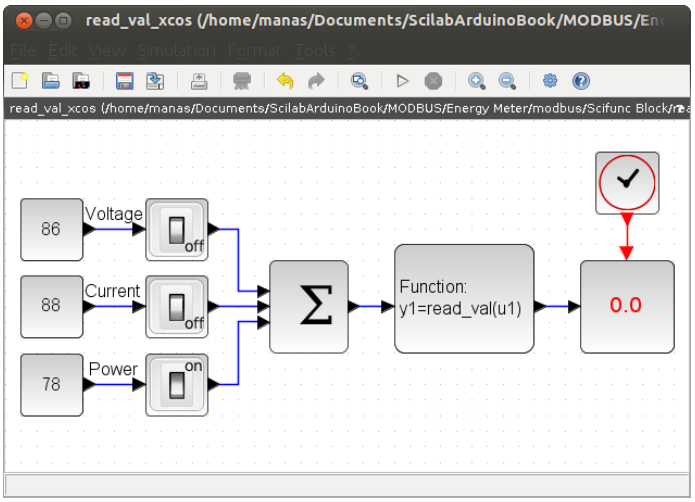


Figure 11.14: Xcos diagram to read Energy Meter values. This is what one sees when `Origin/user-code/modbus/scilab, see Footnote 2 on page 2read_value_xcos.zcos` is invoked.

Table 11.7: Xcos parameters to read Energy Meter

Name of the block	Parameter name	Value
CONST_m	Address byte for voltage	86
	Address byte for current	88
	Address byte for power	78
SELF_SWITCH	Signal Routing	on/off
BIGSOM_f	Scalar vector addition/subtraction Summation	[1;1;1]
scifunc_block_m	Block for userdefined function	read_value.sci
AFFICH_m	Block inherits(1) or not (0)	0
CLOCK_c	Period	0.1
	Initialisation Time	0

1. The Xcos diagram for performing the read values for single phase current, single phase voltage and single phase power operation is as shown in Fig. 11.14. The location of the xcos file is mentioned in the caption of the figure.

   The parameters of the blocks can be changed by right clicking on the block and choosing `Block Parameters`. One can also double click on the block. The values for each block is tabulated in Table 11.7. All other parameters are to be left unchanged.

11.7 Manifestation of Modbus protocol through Python

The objective of this experiment is to make the user acquainted with the demonstration of Modbus protocol through the Python-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.

Python 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. Scilab 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 Python code files 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.

11.8 Reading the electrical parameters from Python

11.8.1 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 Python 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. 3.4 before getting started.

11.8.2 Python Code

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

```
1 #!/usr/bin/python
2
3 import serial,time
4 import struct
5 import sys
6
7 if serial.Serial('COM2',9600).isOpen():
8   serial.Serial('COM2',9600).close()
9 s= serial.Serial('COM2',9600)
10 #s.write("A")
```

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

```
1 #!/usr/bin/python
2
3 import serial,time
4 import struct
5 import sys
6
7 if serial.Serial('COM2',9600).isOpen():
8   serial.Serial('COM2',9600).close()
9 s= serial.Serial('COM2',9600)
10 #s.write("A")
```

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

```
1 #!/usr/bin/python
2
3 import serial,time
4 import struct
5 import sys
6
7 if serial.Serial('COM2',9600).isOpen():
8   serial.Serial('COM2',9600).close()
9 s= serial.Serial('COM2',9600)
10 #s.write("A")
```

11.9 Manifestation of Modbus protocol through Julia

The objective of this experiment is to make the user acquainted with the demonstration of Modbus protocol through the Julia-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.

Julia 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. Scilab 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 Julia source files 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.

11.10 Reading the electrical parameters from Julia

11.10.1 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 Julia 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. 3.5 before getting started.

11.10.2 Julia Code

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

```
1 using SerialPorts
2 include("ArduinoTools.jl")
3
```

```
4 function readCurrent()
5   str = string(Char(1))*string(Char(3))*string(Char(15))*string(Char
      (88))*string(Char(0))*string(Char(2))*string(Char(70))*string(Char
      (204))
6   ser = connectBoard(9600)
7   write(ser,str)
8   sleep(0.1)
9   s = readavailable(ser)
10  close(ser)
```

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

```
1 using SerialPorts
2 include("ArduinoTools.jl")
3
4 function readVoltage()
5   str = string(Char(1))*string(Char(3))*string(Char(15))*string(Char
      (86))*string(Char(0))*string(Char(2))*string(Char(39))*string(Char
      (15))
6   ser = connectBoard(9600)
7   write(ser,str)
8   sleep(0.1)
9   s = readavailable(ser)
10  close(ser)
```

Julia Code 11.3 First 10 lines of the code for Single Phase Active Power Output. Available at `Origin/user-code/modbus/julia`, see Footnote 2 on page 2.

```
1 using SerialPorts
2 include("ArduinoTools.jl")
3
4 function readPower()
5   str = string(Char(1))*string(Char(3))*string(Char(15))*string(Char
      (78))*string(Char(0))*string(Char(2))*string(Char(167))*string(Char
      (8))
6   ser = connectBoard(9600)
7   write(ser,str)
8   sleep(0.1)
9   s = readavailable(ser)
10  close(ser)
```

11.11 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. Scilab 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.

11.12 Reading the electrical parameters from OpenModelica

11.12.1 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. 3.6 before getting started.

11.12.2 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. 3.6.4. Please refer to Fig. 3.44 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] scilab.org. http://www.scilab.org/scilab/about. Seen on 28 June 2015.

[13] scilab.org. http://www.scilab.org/scilab/interoperability. Seen on 28 June 2015.

[14] scilab.org. http://www.scilab.org/scilab/features/xcos. Seen on 28 June 2015.

[15] python.org. https://www.python.org/doc/essays/blurb/. Seen on 24 February 2021.

[16] pyserial - pypi. https://pypi.org/project/pyserial/. Seen on 21 April 2021.

[17] julialang.org/. https://julialang.org/. Seen on 12 April 2021.

[18] Juliaio/serialports.jl: Serialport io streams in julia backed by pyserial. https://github.com/JuliaIO/SerialPorts.jl. Seen on 15 April 2021.

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

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

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

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

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

[24] 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.

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

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

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