Contents

List of Figures

List of Tables

List of Arduino Code

List of Scilab 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 `http://os-hardware.in/arduino/scilab-arduino-files.zip`. 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 microcontoller 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,

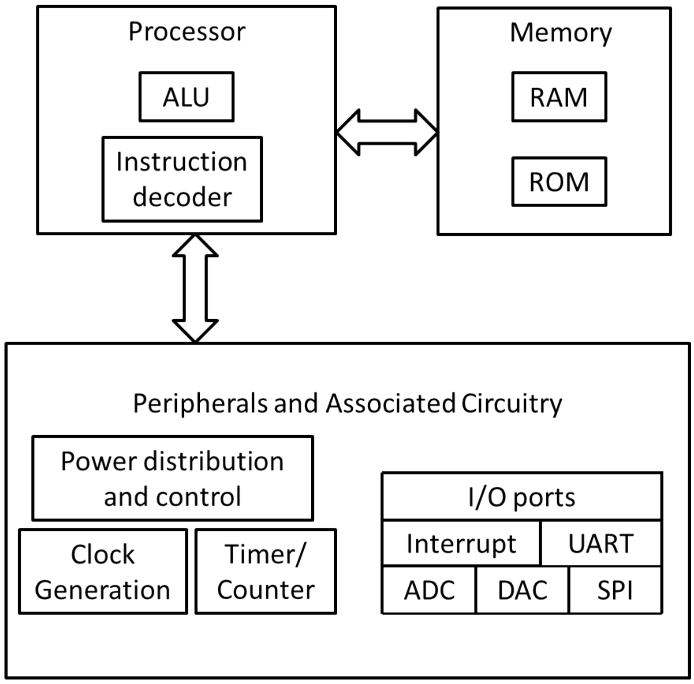


Figure 2.1: Functional block diagram of a microcontroller

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 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 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 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 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.

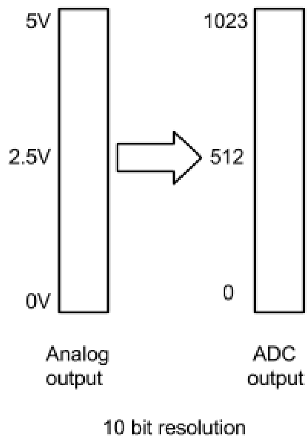


Figure 2.2: ADC resolution

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 CPU is converted to analog signals using 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 a hardware can be called as 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 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 *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 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 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 Arduino boards to interface with Android mobile platform.

Today Arduino is 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.

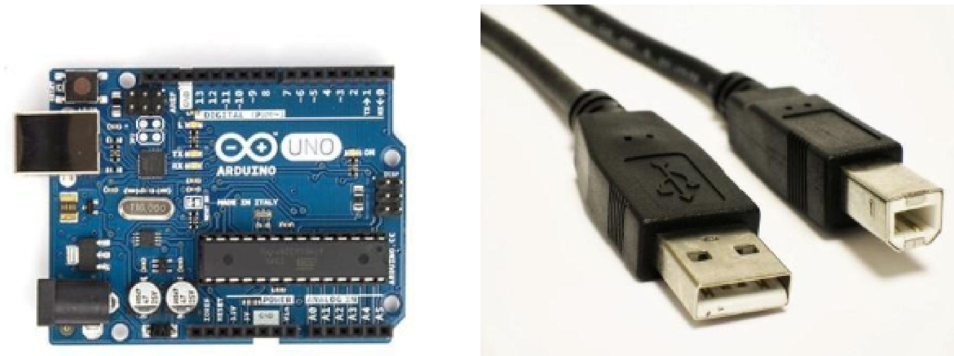


Figure 2.4: Arduino Uno Board

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 on board USB to serial converter, and can be connected to PC using 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

People around the globe are using Arduino in innovative ways. Its ease to setup and use, intuitive, simple software and low cost. We list a few of these projects to give a flavour of some of these interesting applications.

Parameter	Value
Microcontroller	2
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	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328), 0.5 KB used by bootloader
SRAM	2KB (ATmega328)
EEPROM	1KB (ATmega328)
Clock Speed	16 MHz
Length	68.6 mm
Width	53.4 mm
Weight	25g

Table 2.1: Arduino Uno hardware specifications

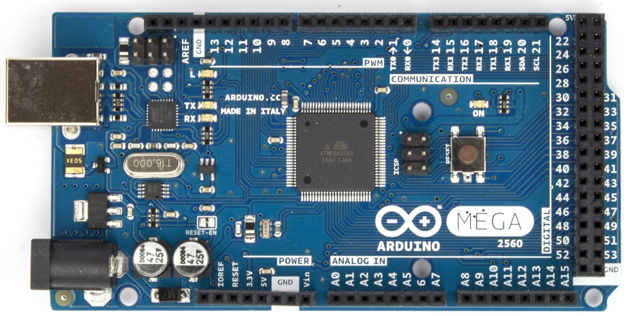


Figure 2.5: Arduino Mega Board

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

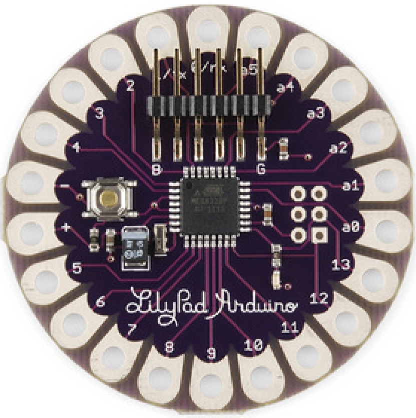


Figure 2.6: LilyPad Arduino Board

Figure 2.7: Arduino Phone

Candy sorting machine: As the name suggest 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 are relatively slow and lack precision, but 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 Diode Codeshield board [11], which makes it easy to perform experiments on the Arduino Uno board.

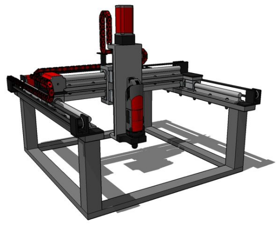


Figure 2.8: 3D printer

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 Diode CodeShield, as mentioned above. We make available all the required information to make a shield, thus making this a 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, which also helps locate the PCB file. The pictorial representation of the schematic for the shield is given in Fig. 2.10, which also helps locate the schematic of the shield. 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 number on Arduino Board [11]. A picture of the completed shield is in Fig. 2.12.

2.5 Experimental Test Bed

We experimented 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

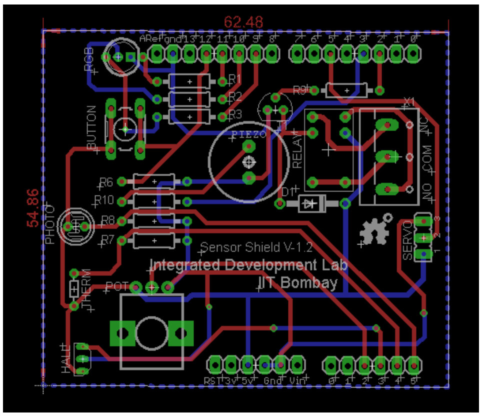


Figure 2.9: PCB image of the shield. The PCB file can be found at `Origin/tools/shield/shield-V1p2.brd`, see Footnote 2 on page 2.

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

3. DC motor and its controller board
4. Servo motor
5. Energy meter with modbus interface

The /arduino/ 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 at the file, mentioned in Footnote 2.

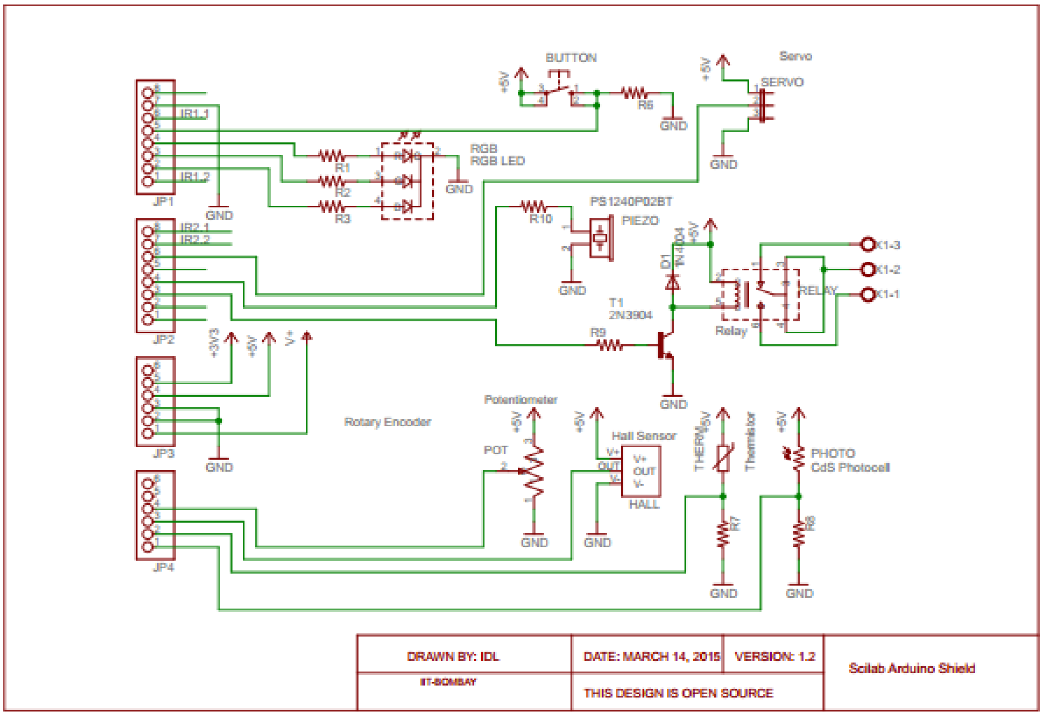


Figure 2.10: Pictorial representation of the schematic of the shield. The actual schematic can be found at `Origin/tools/shield/shield-V1p2.sch.`, see Footnote 2 on page 2

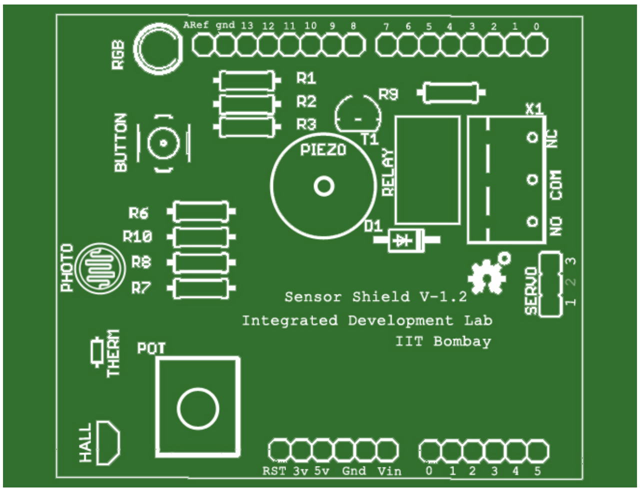


Figure 2.11: PCB of the shield. The actual image can be found at `Origin/tools/shield/shield-V1p2.sch.`, see Footnote 2 on page 2

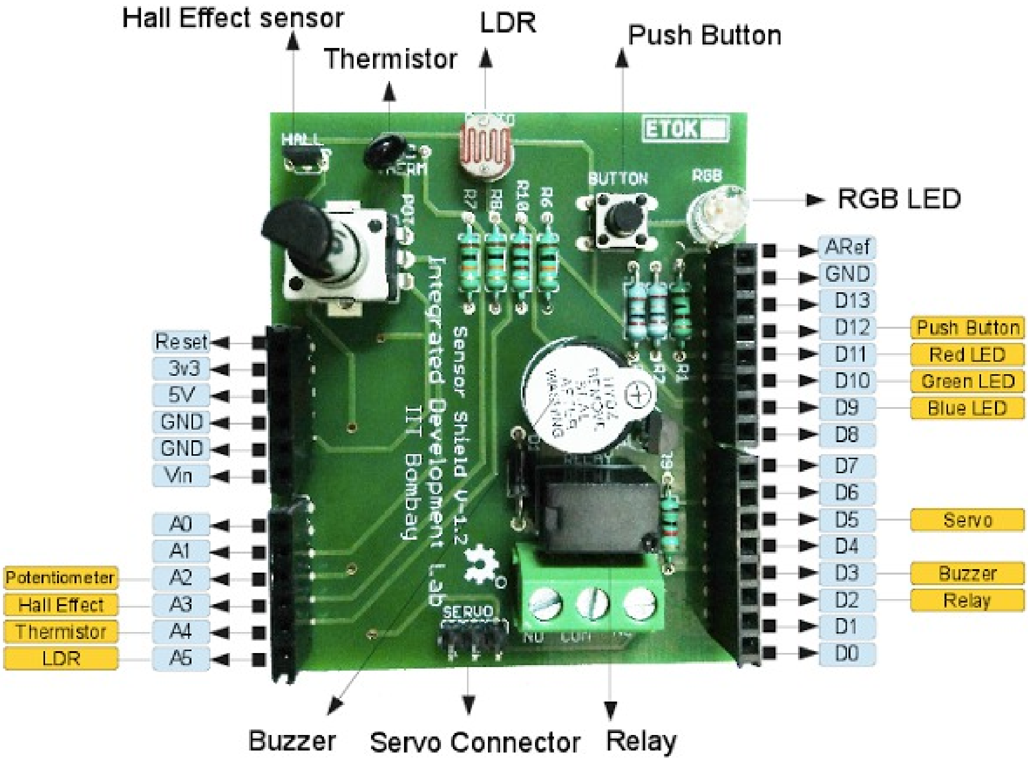


Figure 2.12: Picture of the shield with all components

Table 2.2: Values of components used in the shield

Name	Description	Quantity
R1	100Ω Resistor (Br-Bl-Br)	1
R2, R3, R4	91Ω Resistor (Wt-Br-Bl)	3
R5, R6, R7, R8	10$K\Omega$ Resistor (Br-Bl-Or)	4
R9	1$K\Omega$ Resistor (Br-Bl-Rd)	1
D1	Diode	1
Relay	Relay	1
X1	Terminal block	1
Piezo	Piezo	1
LED 1	LED - 2 lead	1
LED 2	RGB LED	1
T1	Transistor	1
SWITCH	Switch	1
BUTTON, RESET	Push button	2
PHOTO	Photo resistor	1
HALL	Hall effect sensor	1
POT	Potentiometer	1
ENC	Rotary encoder	1
THERM	Thermistor	1
SERVO	Servo	1
SERVO-PARTS	Servo parts	1
NUT, BOLT	Nut, bolt	2
HEADER	6x pin header	2
HEADER	8x pin header	2

Table 2.3: Information on sensors and pin numbers

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

Chapter 3

Software Environment

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 firmware on to the microcontroller. The 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, through 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.

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. 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 the Fig. 2.4.
2. Connect it to a computer and power it up.

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,
   `http://www.arduino.cc/en/Main/Software`
   locate on the right side of the page the link *Windows ZIP file for non admin install* and click it. This may redirect you to 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.

8. Right click on the "Unknown device" and select the "Update Driver Software" option as shown in Fig. 3.2.

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

10. Navigate to 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.

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 14.04 LTS (Trusty Tahr) 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. Find out your operating system support for 64 bit instructions. Open the terminal emulator and type, `uname -m`

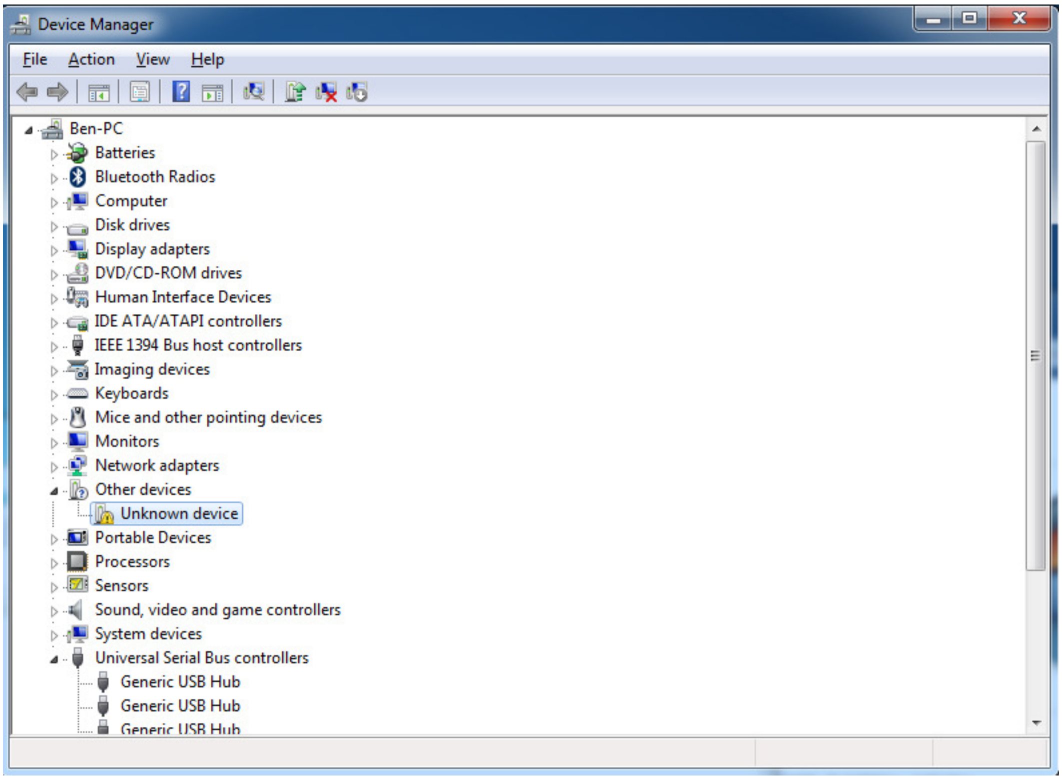


Figure 3.1: Windows device manager

4. 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.

5. Download the suitable Arduino Software version (32 or 64 bits) from `http://arduino.cc/en/Main/Software`. As mentioned earlier, we will perform experiments with a 64 bit installation.

6. At the time of writing this book, we worked with version 1.05. Assuming that you have downloaded tar file in ~/Downloads directory, perform the following steps on the terminal:

```
cd ~/Downloads
tar -zxvf arduino-1.0.5-linux64.tgz
sudo mv arduino-1.0.5 /opt
```

7. In the same terminal session, install the required Java Runtime Environment with a command like, `sudo apt-get -y install openjdk-7-jre`

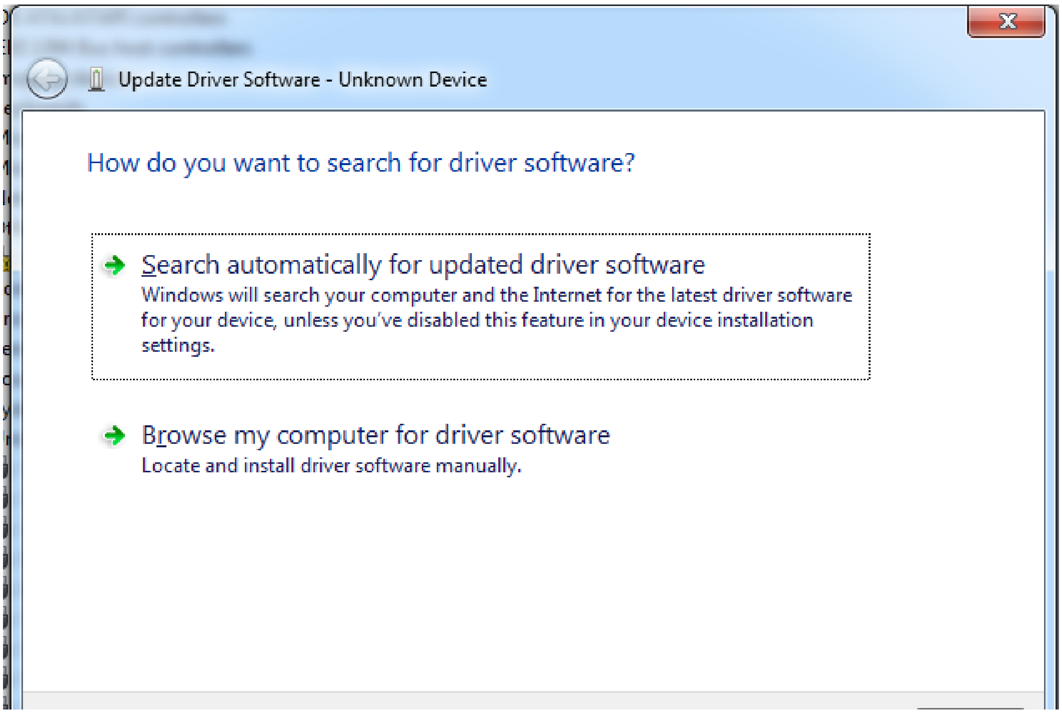


Figure 3.2: Windows update driver option

8. Execute the following command on the terminal to list the serial port number.
   `ls -l /dev/ttyACM*`
   Note down the serial device filename. Suppose that it is `ttyACM0`.

9. To make USB port available to all users, set the read write permission to the listed port: `sudo chmod a+rw /dev/ttyACM0`

10. Create a shortcut on the desktop:
    `cd ~/Desktop`
    `ln -s /opt/arduino-1.0.5/arduino`

11. Give executable permission to this file through the following command on the terminal: `chmod +x arduino`

Then double click the Arduino shortcut on the Desktop and, click "Run" in the dialog window to start the Arduino IDE. The dialog box is shown in Fig. 3.3 for reference. The Arduino IDE is now ready for use.

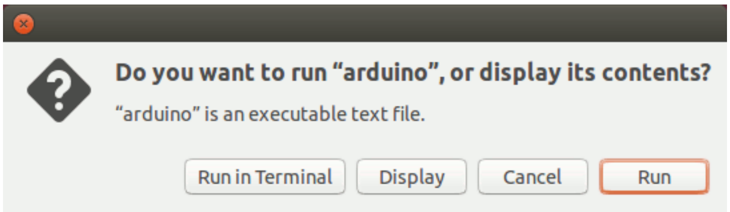


Figure 3.3: Confirmation for executing Arduino script

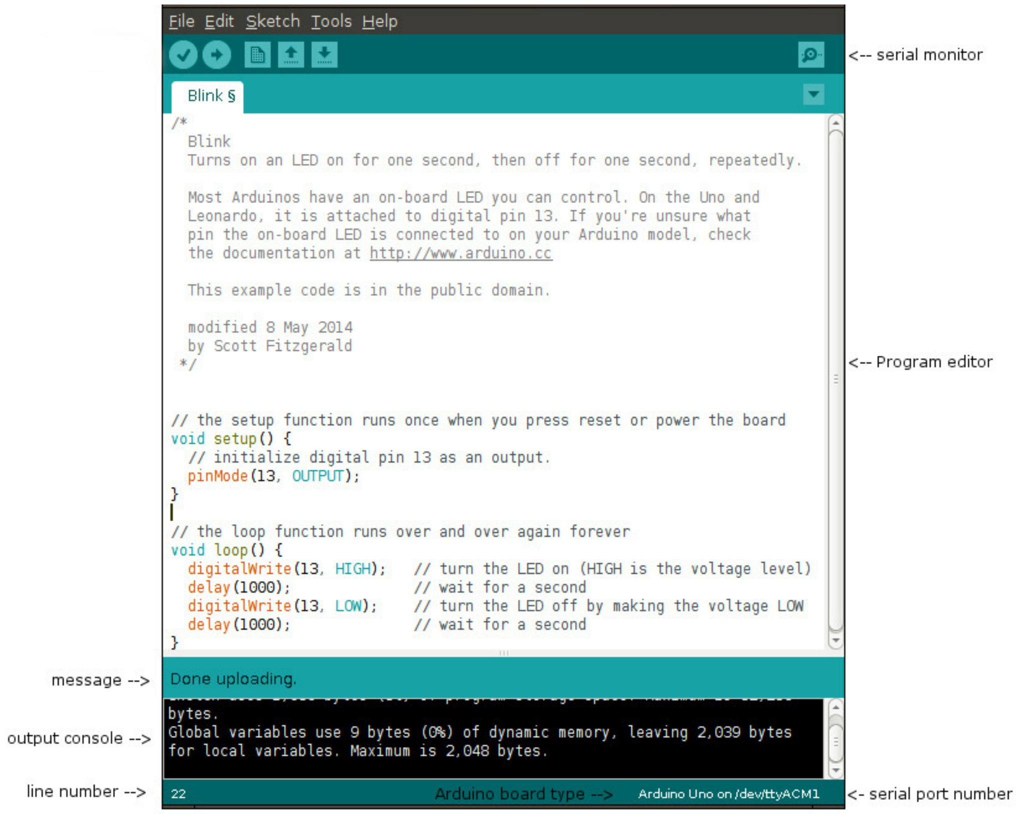


Figure 3.4: Arduino IDE

3.1.3 Arduino Development Environment

The Arduino development environment, as shown in Fig. 3.4, consists of a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions, and a series of menus. It connects to the Arduino hardware to upload programs and communicate with them.

Software written using Arduino are 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. 3.4

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

1. File
   (a) Examples: Examples that come at the time of installation
   (b) Page Setup: Configures the page parameters for printer
   (c) Preferences: Customizes font, language and other parameters for the IDE
2. Sketch
   (a) Import 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) Serial 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 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 port option is greyed out (or disabled) then reinsert the USB cable to PC.

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

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

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

Arduino programming syntax is different from other languages. In an embedded set up, a program is expected to run forever. To facilitate this, 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 power up 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 explanation on IDE by navigating to "Help" and then Environment. Access Arduino language reference at "Help" and then Reference. Access FAQs and troubleshooting tricks at "Help" and then Troubleshooting.

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 licence. 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.

3.2.1 Downloading and installing Scilab

Scilab can be downloaded free of cost from `www.scilab.org`. It is available for all popular operating systems, including Windows, Linux and Mac OS X systems. This book uses Scilab-5.5.2 for demonstrating the experiments, both on Windows and Linux.

For Windows system, the downloaded file is an executable (.exe) file. It has to be double clicked to start the installation. All the default parameters of installation are acceptable. It has to be noted that Scilab requires internet connectivity during installation on Windows. There is an option in the beginning of the installation to

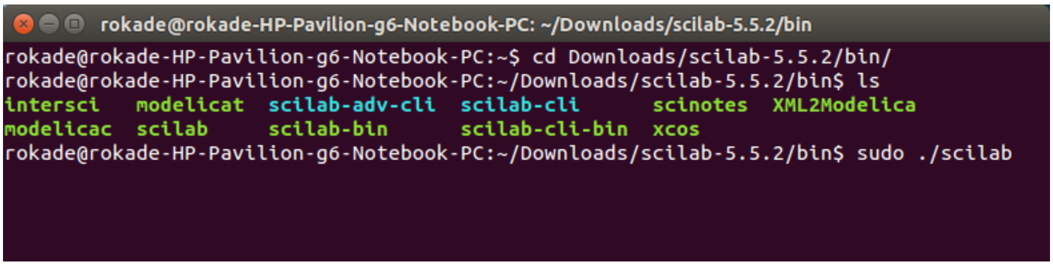


Figure 3.5: Linux terminal to launch Scilab

continue offline but it is not recommended. Scilab can be launched either from the Start menu or by double clicking on Scilab icon created on the Desktop (if any).

Package managers of Linux do not have the latest versions of Scilab. As a result, downloading directly from the Scilab website, `http://scilab.org`, is recommended, as there are differences in Scilab from version to version. We expect such problems to be overcome in the future. The file downloaded will be in a tar.gz format. It has to be extracted before use. 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. 3.5.

3.2.2 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/windows` or `Origin/tools/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 shorcut icon created on the Desktop. On a Linux system, one has to start Scilab through a terminal

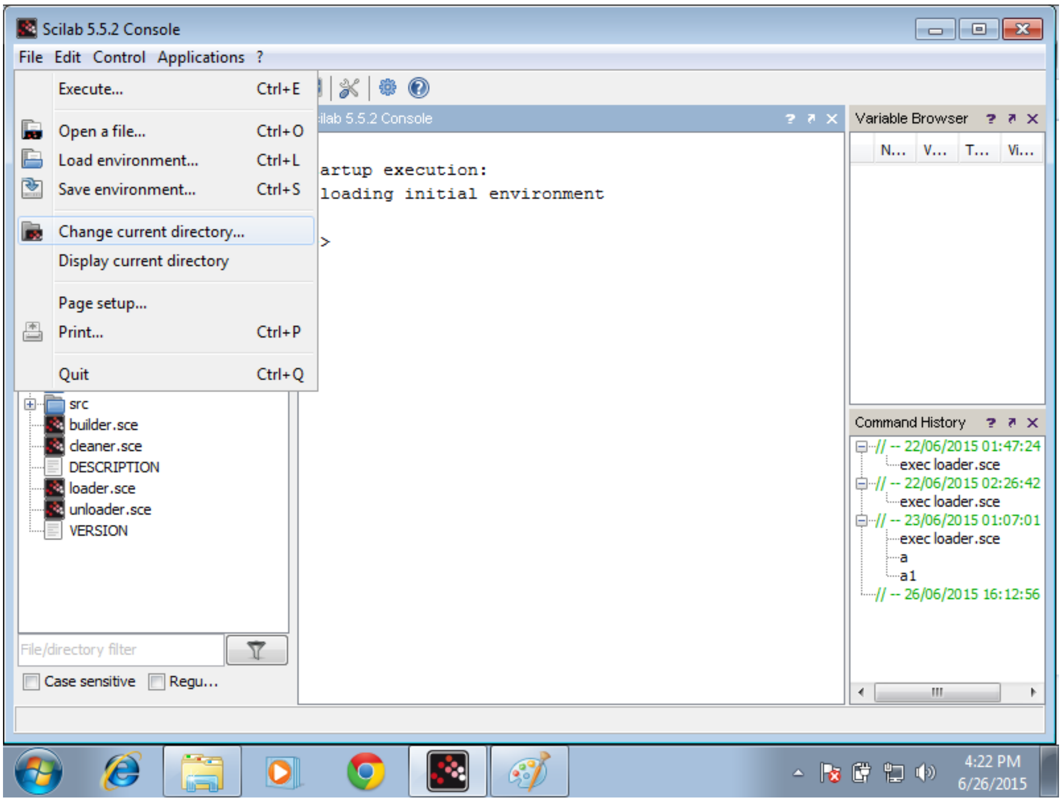


Figure 3.6: Changing scilab directory

with root permissions, as explained in section 3.2.1.

2. After launching Scilab, first we have to change the working directory. To do so, click on the `File` menu and then click on the `Change current directory` option as shown in Fig. 3.6.

3. Then, one has to browse to the toolbox folder `Origin/tools/windows` or `Origin/tools/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.

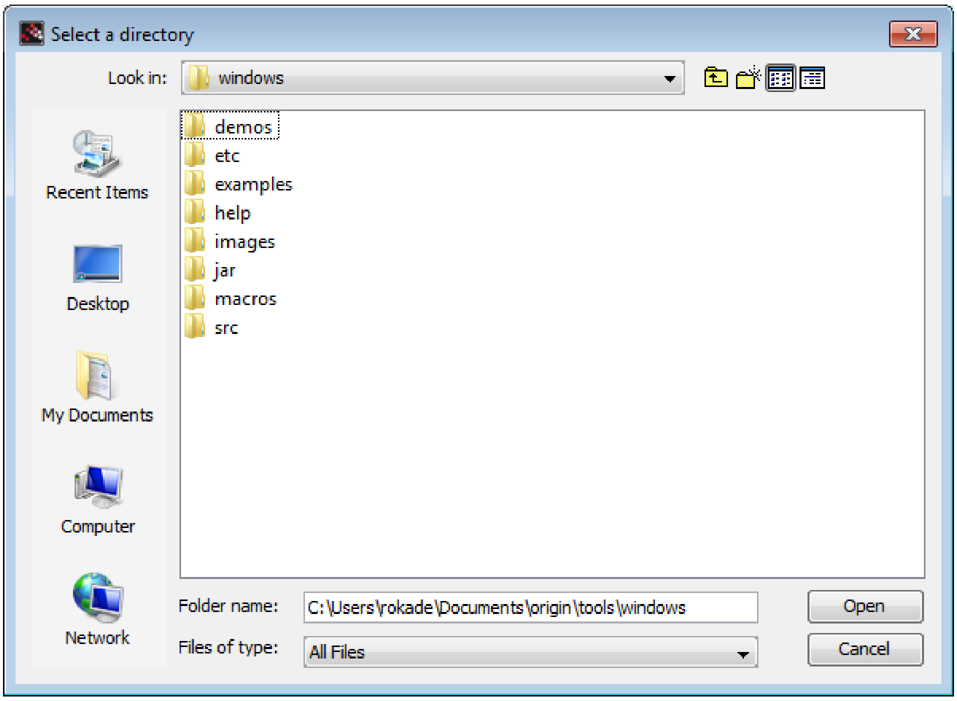


Figure 3.7: Browsing toolbox directory

5. Next, type the following command on the Scilab console: `exec builder.sce` - this will build the toolbox and create a file `loader.sce`. This step has to be executed only the first time. 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 all 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 work space for any reason, you will have to execute this command once again[3]

The toolbox is now loaded and available for use.

3.2.3 Identifying Arduino communication port number

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

[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.

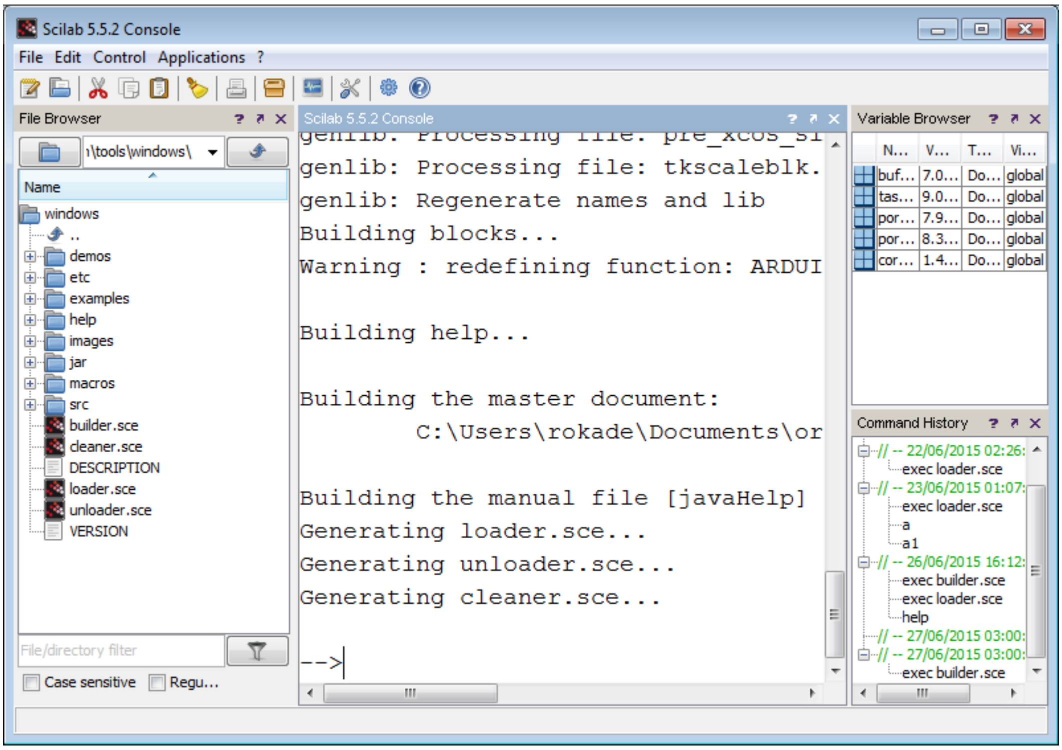


Figure 3.8: Output of builder.sce

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

[4]It is possible to leave it at whatever port number one gets. It is also possible to choose any

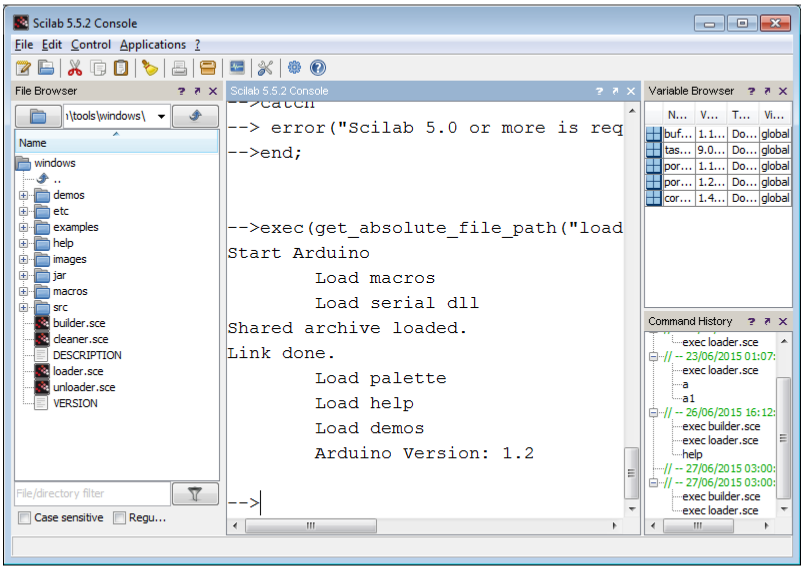


Figure 3.9: Output of loader.sce

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 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 -l /dev/ttyACM*` - the output of this command is shown in Fig. 3.12. It has detected the Arduino with 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.

3.2.4 Testing Scilab-Arduino toolbox

Now let us test the functioning of the toolbox.

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.

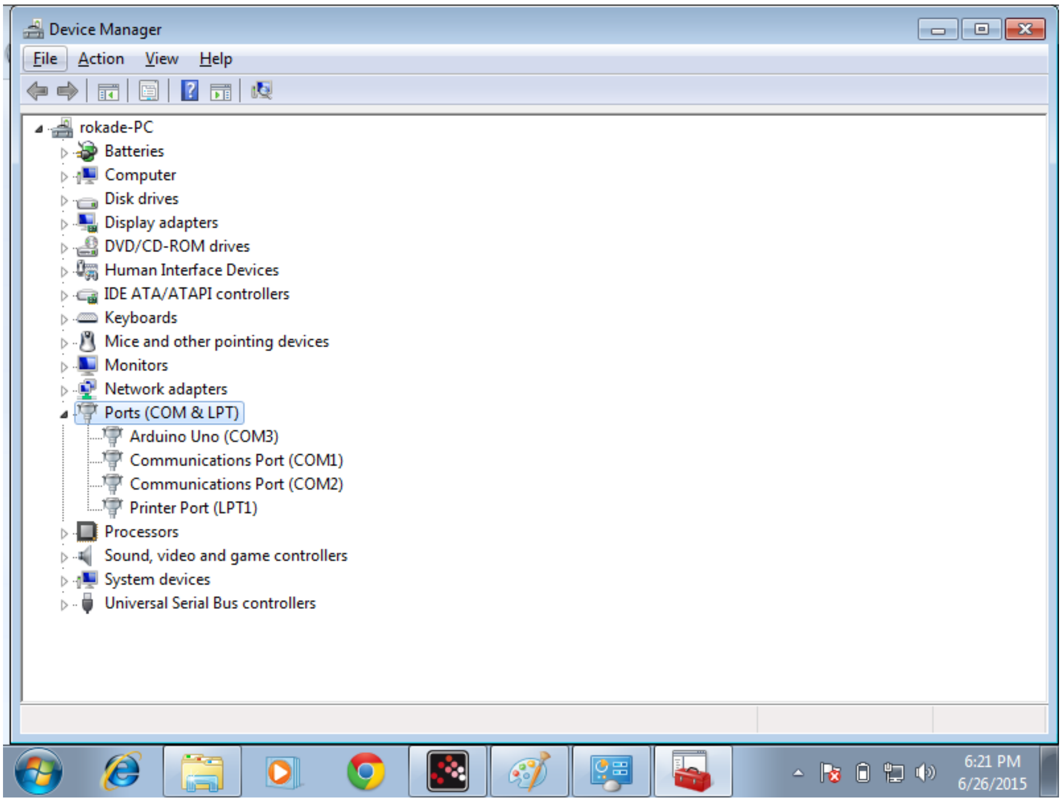


Figure 3.10: Device Manager in windows

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` 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

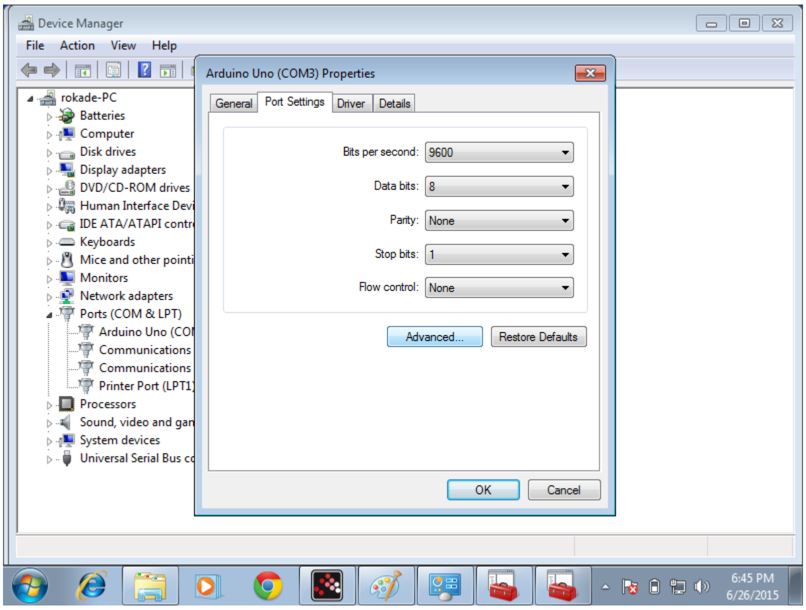


Figure 3.11: COM port properties window

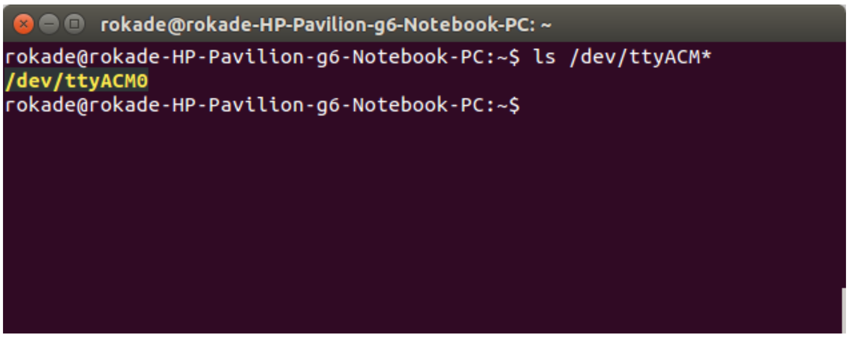


Figure 3.12: Port number on Linux terminal

the editor. Click on “File" menu and choose “Open". Browse to the directory `Origin/tools` 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

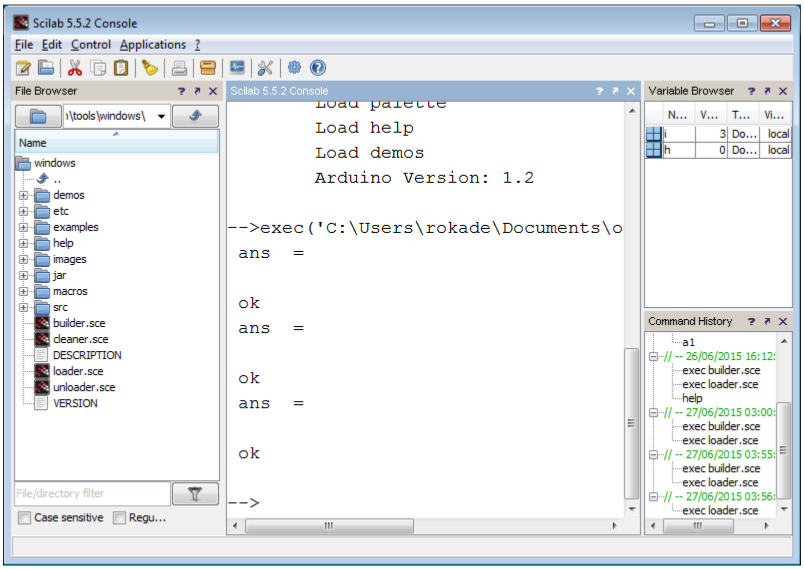


Figure 3.13: Scilab test code output

```
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.

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

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

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 form 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 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 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 which generate

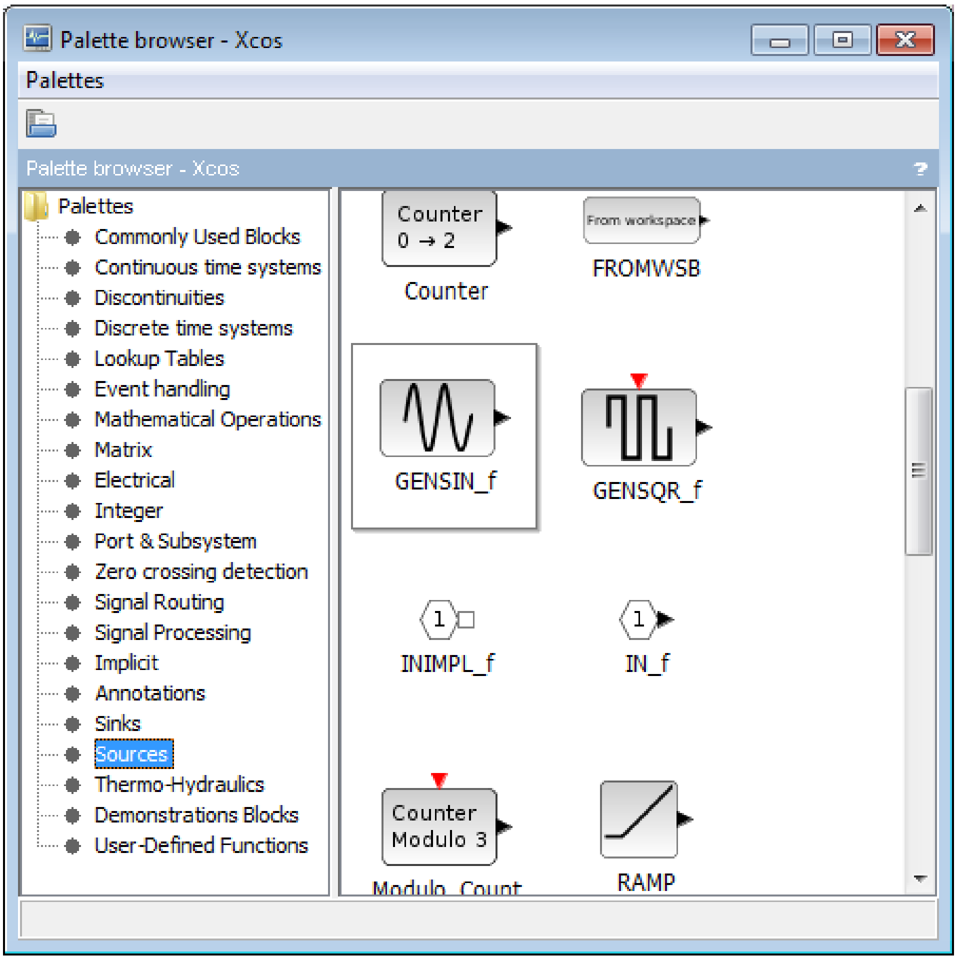


Figure 3.15: Sine generator in palette browser

signals/values without any input, fall under the category `Sources`. Similarly, blocks which 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.

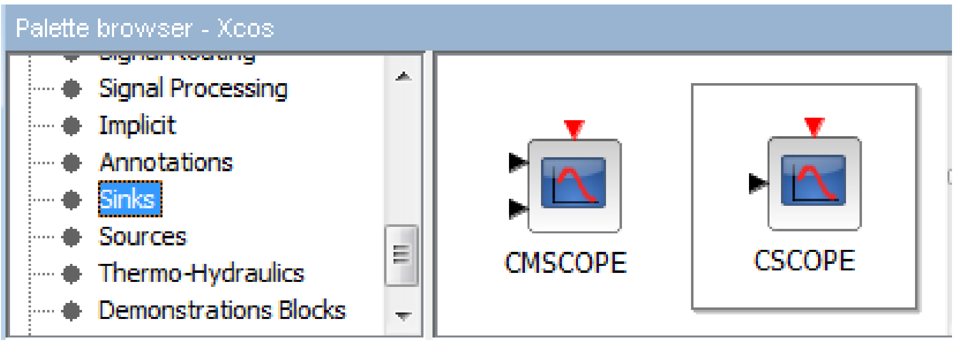


Figure 3.16: CSCOPE block in xcos

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 print out. 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 event port. This is red in colour. This port is used to synchronise it with event generating devices.

As the CSCOPE block has an input event port, we need a source which 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 other 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 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 in the Fig. 3.18. Save this file with name `sine-generator`.

Let us simulate this Xcos code. On the menu bar, click on the `Simulation`

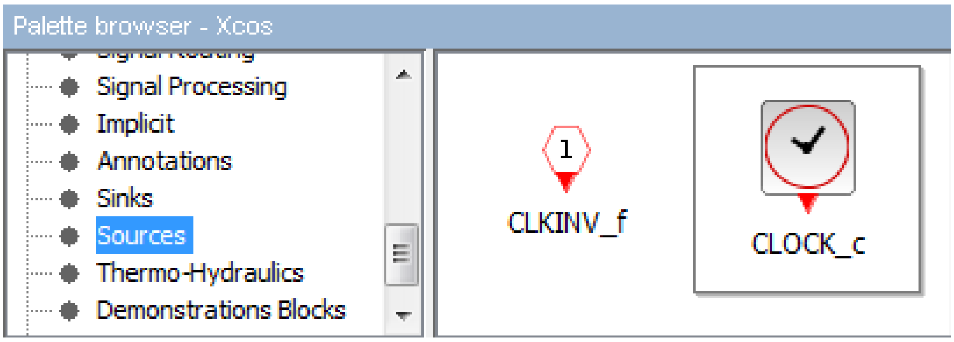


Figure 3.17: CLOCK_c block in xcos

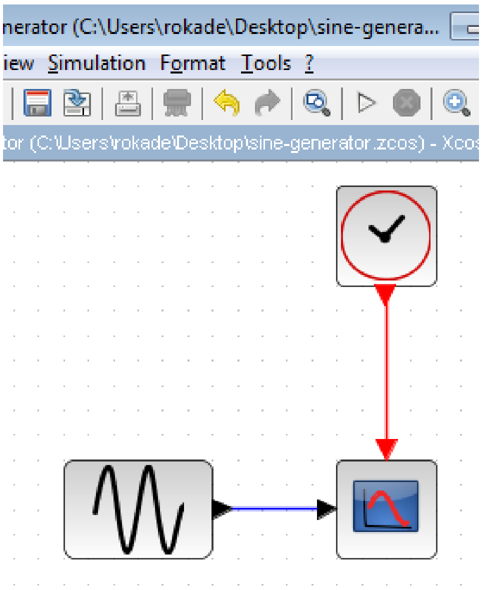


Figure 3.18: Sine generator in Xcos

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.

Next, on the menu bar, click on the `simulation` menu and choose `Setup`. The

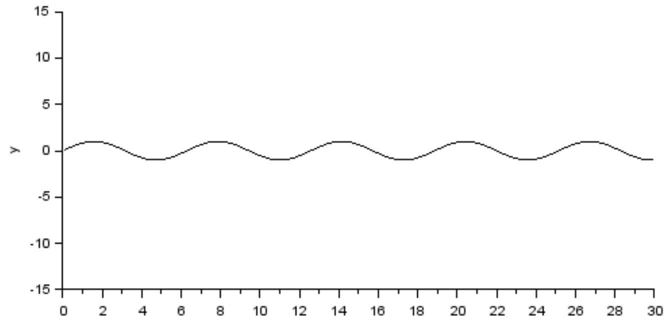


Figure 3.19: Sine generator Xcos output

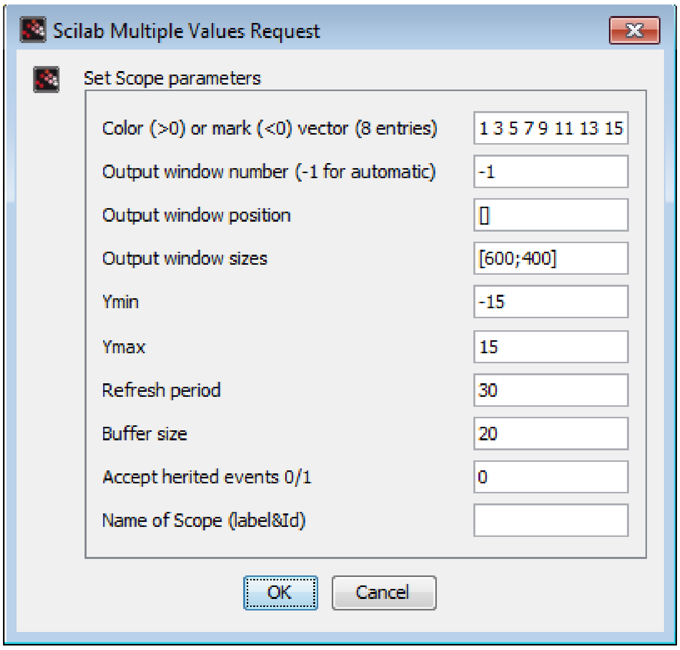


Figure 3.20: CSCOPE configuration window

`Set parameter` 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 paramenters 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.

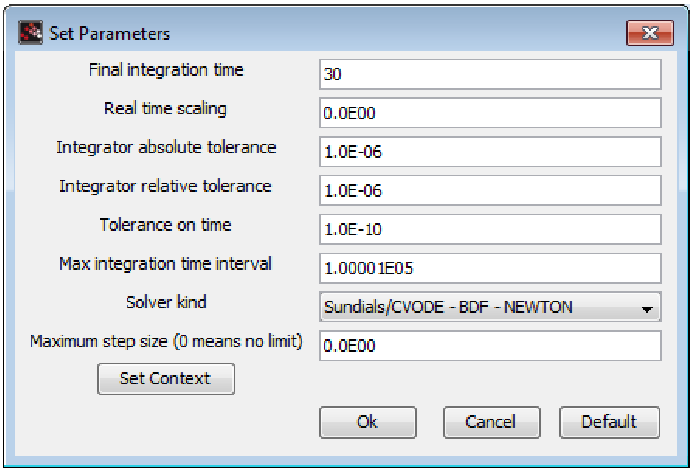


Figure 3.21: Simulation setup window

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 is beyond the scope for this book.

3.3.3 Xcos-Arduino

The Scilab Arduino toolbox not only provides functions to be used in Scilab scripts but also provides with 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 througout 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.

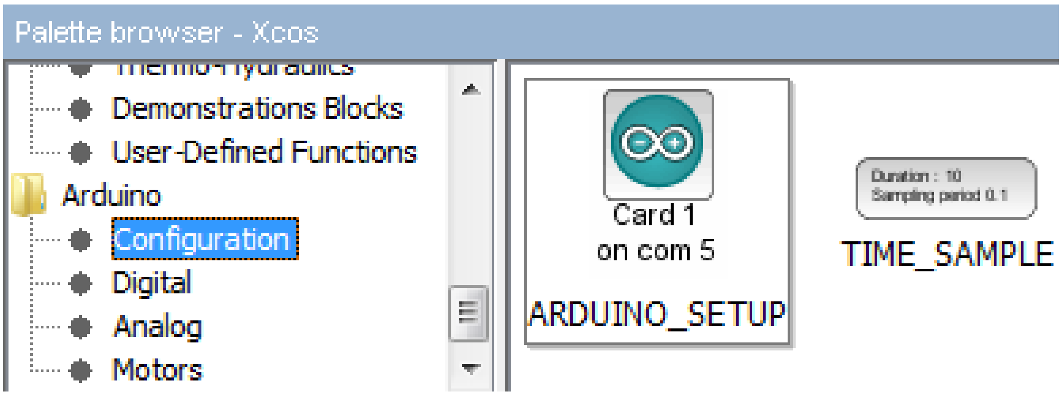


Figure 3.22: Palette browser showing Arduino blocks

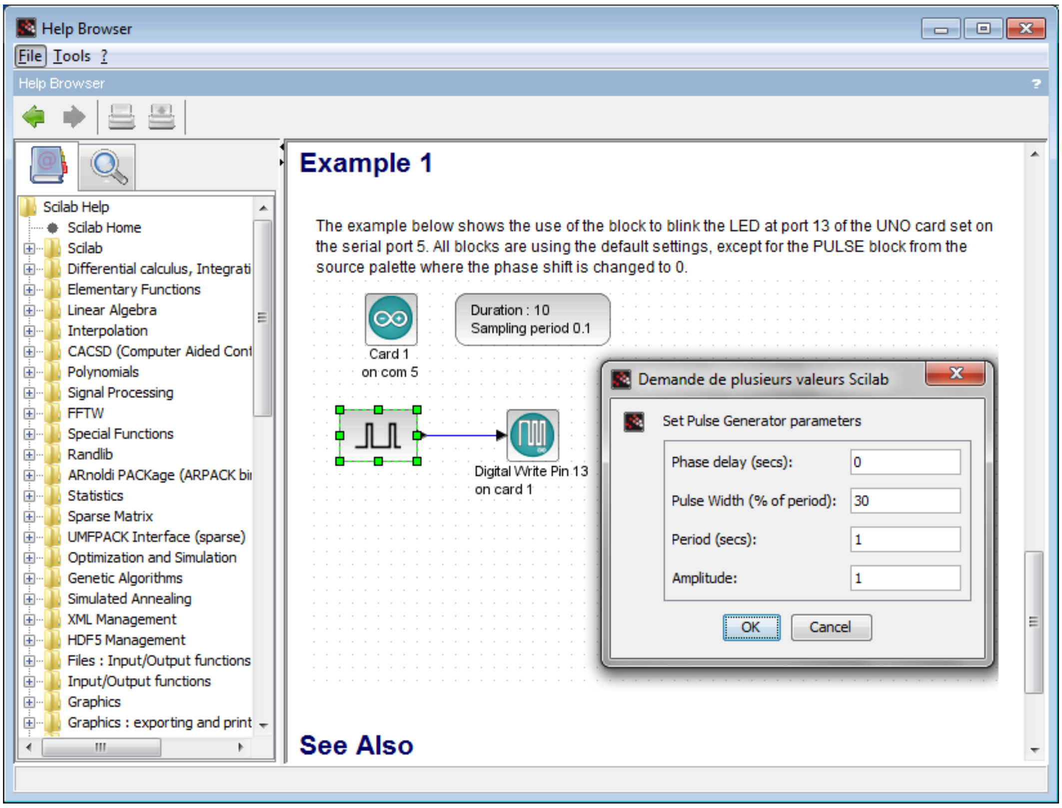


Figure 3.23: Xcos block help

3.4 Arduino Code

Arduino Code 3.1 First 10 lines of the Arduino firmware. Available at `Origin/tools/arduino-firmware/arduino-firmware.ino`, see Footnote 2 on page 2.

```
/* This file is meant to be used with the SCILAB arduino
   toolbox, however, it can be used from the IDE environment
   (or any other serial terminal) by typing commands like:

   Conversion ascii -> number
   48->'0' ... 57->'9' 58->':' 59->';' 60->'<' 61->'=' 62->'>' 63->'?'
    64->'@'
   65->'A' ... 90->'Z' 91->'[' 92->'\' 93->']' 94->'^' 95->'_' 96->'`'
   97->'a' ... 122->'z'

   Dan0 or Dan1 : attach digital pin n (ascii from 2 to b) to input (0)
     or output (1)
```

3.5 Scilab Code

Scilab Code 3.1 A code to check whether the firmware is properly installed or not. Available at `Origin/tools/test_firmware.sce`, see Footnote 2 on page 2.

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

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 and Scilab Xcos. 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

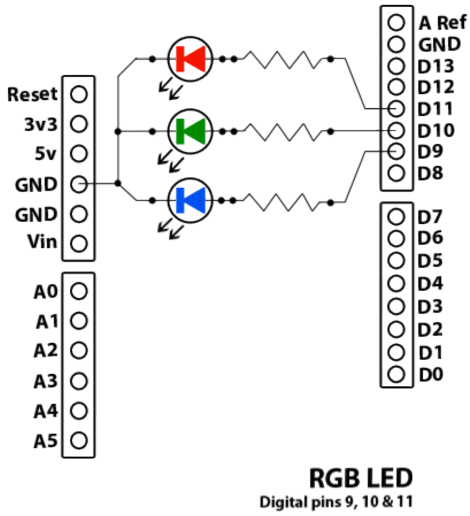


Figure 4.2: Internal connection diagram for LED on the shield

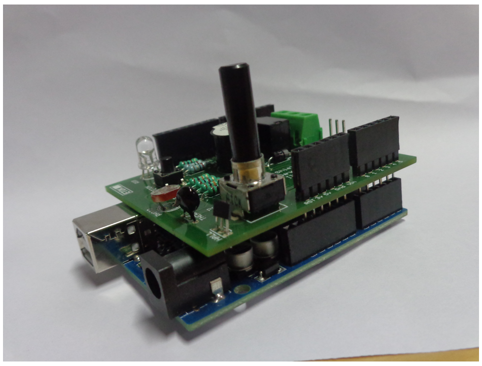


Figure 4.3: Connecting Arduino and Shield

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, respectively, connected to pins 11, 10 and 9. 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 internally made available. 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 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 Lighting the LED from the Arduino IDE

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. 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:

   ```
   Serial.begin(115200);
   ```

   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:

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

   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,

   ```
   delay(2000);
   ```

   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. 4.4.

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.

∎

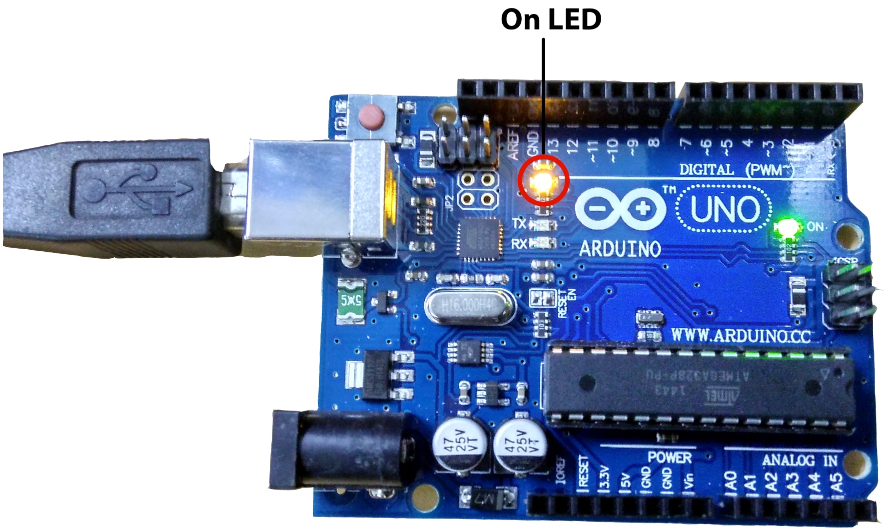


Figure 4.4: LED experiments directly on Arduino Uno board, without the shield

4.3 Lighting the LED from Scilab Scripts

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 before carrying out these experiments, as in Sec. 4.2. 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

   ```
   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.

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

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. 4.2, 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.

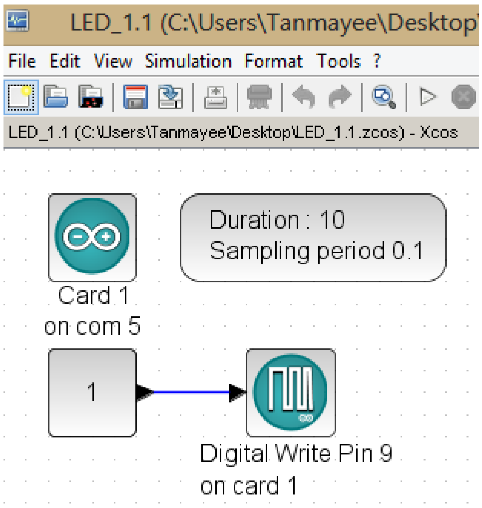


Figure 4.5: Turning the blue LED on through Xcos. This is what one sees when `Origin/user-code/led/scilab/led-blue.zcos`, see Footnote 2 on page 2 is invoked.

Exercise 4.2 Repeat the exercise of the previous section.

4.4 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.

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. 4.5. 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 4.1. All other parameters are to be left unchanged.

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 28
TIME_SAMPLE	Duration of acquisition(s)	10
	Sampling period(s)	0.1
DIGITAL_WRITE_SB	Digital pin	9
	Arduino card number	1

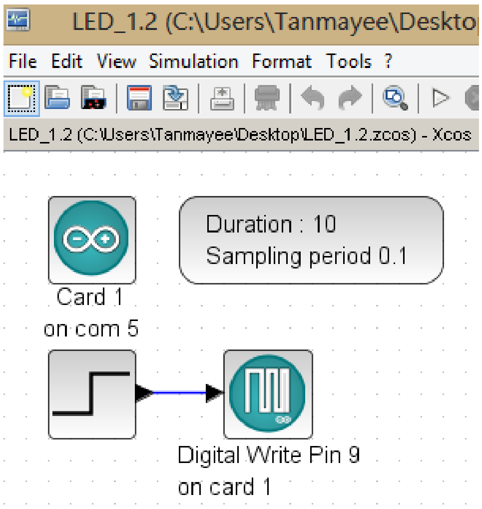


Figure 4.6: Turning the blue LED on through Xcos for two seconds. This is what one sees when `Origin/user-code/led/scilab/led-blue-delay.zcos`, see Footnote 2 on page 2 is invoked.

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. 4.6. 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. 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. 4.7. In the caption of this figure, one can see where

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 28
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

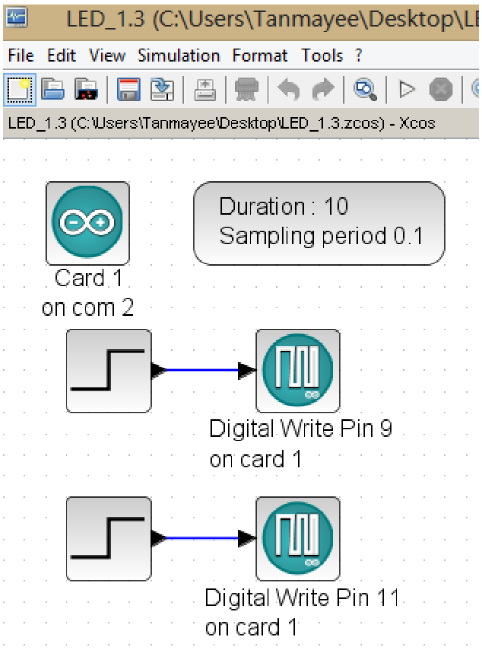


Figure 4.7: Turning the blue and red LEDs on through Xcos and turning them off one by one. This is what one sees when `Origin/user-code/led/scilab/led-blue-red.zcos`, see Footnote 2 on page 2 is invoked.

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.

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 28
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

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 28
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

4. We will conclude this section with an experiment to blink the green LED on and off. 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 4.4. All other parameters are to be left unchanged.

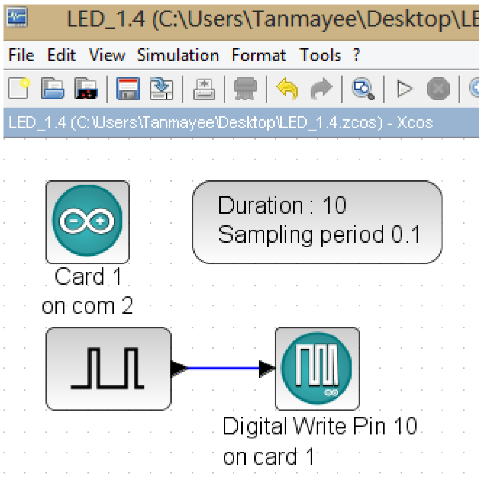


Figure 4.8: Blinking the green LED every second through Xcos. This is what one sees when `Origin/user-code/led/scilab/led-green-blink.zcos`, see Footnote 2 on page 2 is invoked.

Exercise 4.3 Carry out the following exercise:

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

∎

4.5 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.

```
void setup() {
pinMode(9,OUTPUT);
Serial.begin(115200);
digitalWrite(9,HIGH);
}
void loop() {
}
```

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.

```
void setup() {
pinMode(9,OUTPUT);
Serial.begin(115200);
digitalWrite(9,HIGH);
delay(2000);
digitalWrite(9,LOW);
}
void loop() {
}
```

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.

```
void setup() {
pinMode(9,OUTPUT);
pinMode(11,OUTPUT);
Serial.begin(115200);
digitalWrite(9,HIGH);
digitalWrite(11,HIGH);
delay(5000);
digitalWrite(9,LOW);
delay(3000);
digitalWrite(11,LOW);
}
void loop() {
}
```

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

```
int i=0;
void setup() {
  pinMode(10, OUTPUT);
  Serial.begin(115200);
  for(i=0;i<5;i++)
  {
  digitalWrite(10, HIGH);    // turn the LED on (HIGH is the voltage
    level)
  delay(1000);               // wait for a second
  digitalWrite(10, LOW);     // turn the LED off by making the voltage
    LOW
  delay(1000);               // wait for a second
  }
}
void loop() {
}
```

4.6 Scilab Code

Scilab Code 4.1 Turning on the 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');
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
```

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 a few experiments to read the status of the pushbutton mounted on the shield of the Arduino Uno board. Advancing further, we shall perform a few tasks 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 provided by the manufacturer in the datasheet.

5.2 Reading the Pushbutton status from the Arduino IDE

In this section, we shall learn commands to read the status of a pushbutton through Arduino IDE. Later, we shall change the state of the LED depending on the status

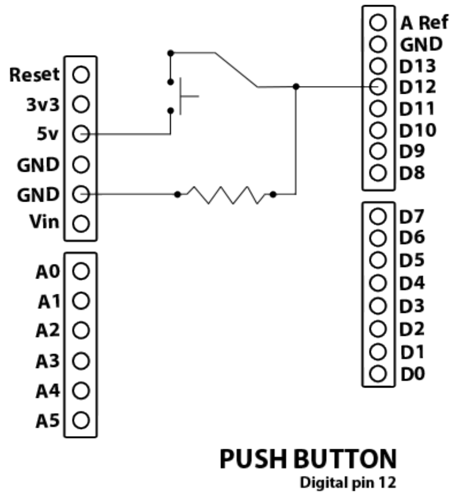


Figure 5.1: Connection Diagram

of the pushbutton.

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 9600 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:

```
pinMode(sensorPin, INPUT);    // declare the sensorPin as an
  INPUT
```

   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,

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

   We repeat this read and print process 1000 times by putting the commands in a `for` loop. At the same time, the user must press and release the pushbutton and observe the values printed on the serial monitor.

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,

```
  Serial.begin(115200);
```

   Next, we read the potentiometer 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 1000 iterations.

5.3 Reading the Pushbutton status from Scilab Scripts

In this section, we shall perform the pushbutton operation using Scilab-Arduino toolbox commands.

1. In the first experiment, we will read the pushbutton status in Scilab Console. The code for this experiment is given in Scilab Code 5.1. As explained earlier, we begin with serial port initialization. Then, using the command,

```
    val = cmd_digital_in(1,12);   // Read the status of pin 12
```

   we read the input of digital pin 12. Note that the middle terminal of the potentiometer is connected to this pin. The read value is displayed as a GUI using the command,

```
    cmd_arduino_meter(val);
```

   where `val` contains the potentiometer value acquired by the previous command. To encourage the user to have a good hands-on, we run these commands in a `for` loop for 1000 iterations.

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 lines,

```
    if p==0
        cmd_digital_out(1,9,0)
    else
        cmd_digital_out(1,9,1)
    end
```

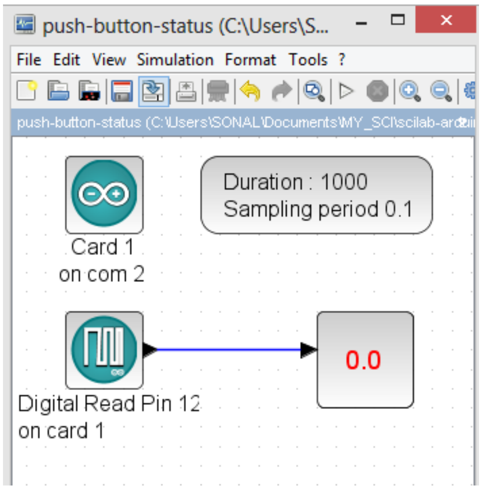


Figure 5.2: Printing the push button status on the display block. This is what one sees when `Origin/user-code/push/scilab/push-button-status.zcos`, see Footnote 2 on page 2, is invoked.

perform the condition check and corresponding LED state control operation.

5.4 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. 5.2. 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 potentiometer 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.

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 28
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

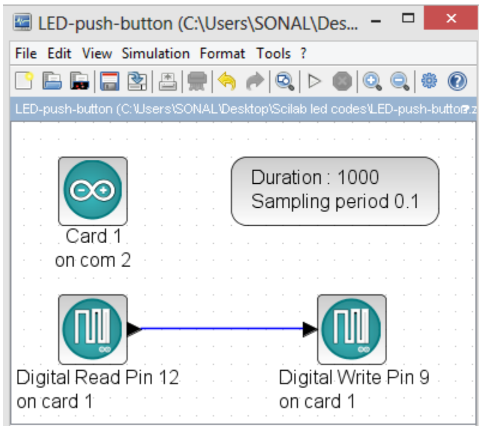


Figure 5.3: Turning the LED on or off, depending on the pushbutton. This is what one sees when `Origin/user-code/push/scilab/led-push-button.zcos`, see Footnote 2 on page 2, is invoked.

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

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 28
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

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.

∎

5.5 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.

```
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 < 1000;i++){
     sensorValue = digitalRead(sensorPin); // read push-button value
     Serial.println(sensorValue); // print it at the Serial Monitor
  }
}
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;
int i;
void setup() {
  Serial.begin(115200);
  pinMode(9, OUTPUT);
  pinMode(12, INPUT);
  for (i = 0; i < 1000; i++) {
    sensorValue = digitalRead(12);
    if (sensorValue==0) {
      digitalWrite(9, LOW);
      delay(5);
    }
    else {
     digitalWrite(9, HIGH);
     delay(5);
    }
  }
}
void loop() {
}
```

5.6 Scilab Code

Scilab Code 5.1 Read the status of the pushbutton and displaying on the serial monitor. Available at `Origin/user-code/push/scilab/push-button-status.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 i=1:1000                      // Run for 1000 iterations
    val = cmd_digital_in(1,12);   // Read the status of pin 12
    cmd_arduino_meter(val);
end
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.

```
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=1:1000                         //Run for 1000 iterations
    p=cmd_digital_in(1,12)
    if p==0
        cmd_digital_out(1,9,0)
    else
        cmd_digital_out(1,9,1)
    end
end
close_serial(1)
```

Chapter 6

Interfacing a Light Dependent Resistor

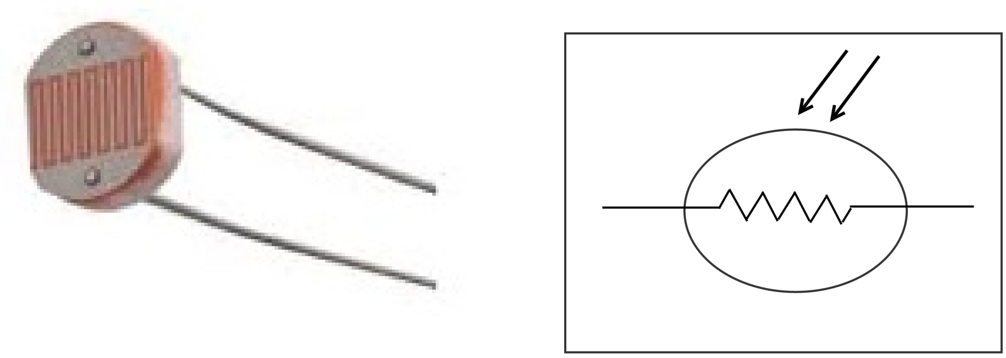
A Light Dependent Resistor (LDR) or Photoresistor is a light sensitive semiconductor device whose resistance varies with the variation in the intensity of light falling on it. As the intensity of the incident light increases, resistance offered by the LDR decreases. Typically, in dark, the resistance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

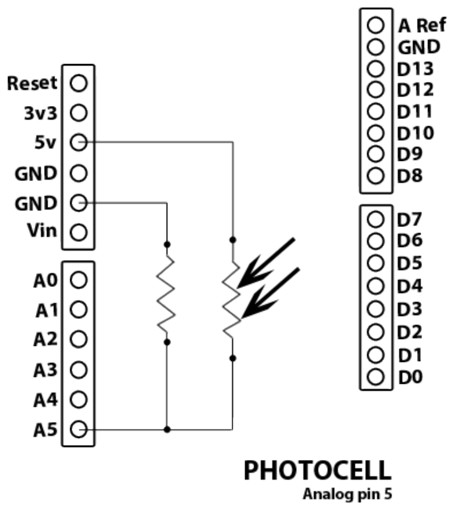


Figure 6.2: Internal connection diagram for the LDR on the shield

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.

6.2 Arduino Experiments

In this section, we shall learn 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.

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,

   ```
   6    val1=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 `val1`. The read values are then displayed using,

   ```
   7    Serial.println(val1);   // for display
   ```

   The command, on line 8,

   ```
   8    delay(500);
   ```

   is given so that the readings do not scroll away very fast. The entire reading and display operation is carried out 20 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.

2. In this experiment, 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 `val1`, which is written on to the Serial Monitor. In case it is above some threshold (it is 300 in the code), it puts a high in pin number 11. From Sec. 4.1, one can see that this pin is for the red LED. If the LDR value is about 300, the red LED will be on, else, it will be turned off. This loop is repeated 2,000 times.

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 Scilab Experiments

In this section, we will explain a few Scilab experiments to read the LDR values corresponding to the incident light. The LDR values can be read using the following function of Scilab Arduino toolbox:

```
cmd_analog_in(1,port number on Arduino Uno)
```

where the input argument 1 is fixed for this kit, and the port number corresponds to the analog pin of Arduino Unothat needs to be read. We will carry out two experiments using Scilab.

1. We use Scilab Code 6.1 to read the LDR values. We find the port number from the computer settings and give it as input to the `open_serial` command to start serial port communication. In our case, the port number is 2. Next, we shall fetch LDR values using the command, `cmd_analog_in`, as explained above. This is indicated on line 4 of the code. We run this command in a `for` loop 20 times. In each iteration of the `for` loop, we acquire LDR data fed to analog pin 5, display it in the Scilab command window and suspend Scilab operation for 500 milliseconds. The output of this experiment is displayed on the Scilab command window. After reading the values, we close the serial port using the command, `close_serial`, of Scilab-Arduino toolbox.

2. In this experiment, we will observe the saturation point of LDR, see Scilab Code 6.2. We know that as incident light intensity increases, voltage at analog input of the Arduino Uno board increases. Thus the ADC values being read by the Arduino Uno board also increase. But after certain high intensity, ADC values reach its maximum. For 10 bit ADC in Arduino, this high intensity corresponds to 1023. Beyond this value, the LDR is incapable of sensing the change in light intensity and is considered to be saturated. To observe this saturation point, we can do a simple task of exposing LDR to high intensity. We can put a torch/light source sensor to close proximity of LDR.

Exercise 6.2 Carry out the exercise below:

1. Carry out the exercise in the previous section

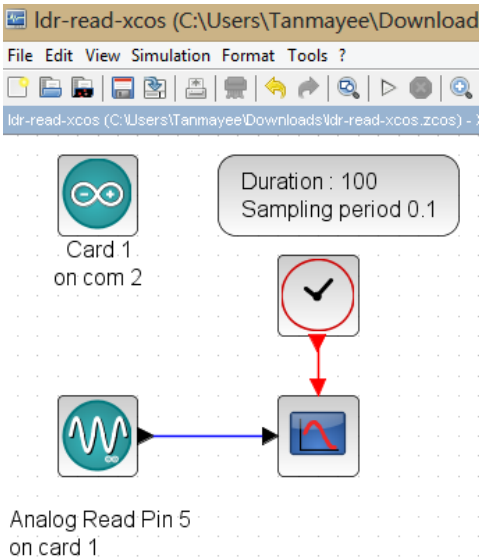


Figure 6.3: Xcos diagram to read LDR values. This is what one sees when `Origin/user-code/ldr/scilab/ldr-read.zcos`, see Footnote 2 on page 2, is invoked.

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 LDR experiments through Xcos

Next, we shall perform the above mentioned experiment, to read LDR values, through Xcos. We will carry out the same four 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. The Xcos diagram in Fig. 6.3 performs data acquisition from analog pin 5 and displays the read values on the scope. When the file required for this experiment is invoked, one gets the GUI as in Fig. 6.3. In the caption of this figure, one can see where to locate the file.

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 28
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 several light sources and obstacles such as torch light, paper, hand, etc. and observe the LDR readings. We observe that with a constant light source, the LDR output saturates after some time.

2. In the second experiment, we read the value of the LDR and using it, turn the red LED on or off. When the file required for this experiment is invoked, one gets the GUI as in Fig. 6.4. 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. 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.

6.5 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.

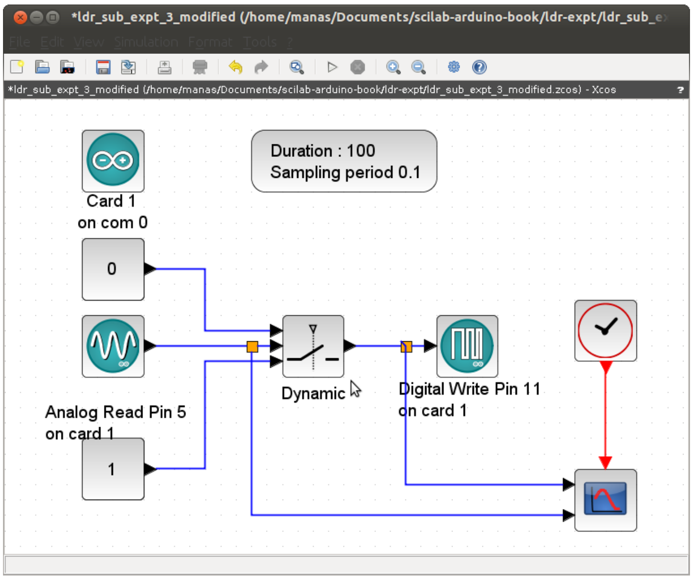


Figure 6.4: Xcos diagram to read the value of the LDR, which is used to turn the blue LED on or off. This is what one sees when `Origin/user-code/ldr/scilab/ldr-led-xcos.zcos`, see Footnote 2 on page 2, is invoked.

```
int val1;                    // for LDR
int i=1;
void setup() {
Serial.begin(115200);
for(i=1;i<=20;i++){
    val1=analogRead(A5);    // value of LDR
    Serial.println(val1);   // 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 val1;
int i=1;
void setup() {
pinMode(11,OUTPUT);           // LED Pin
Serial.begin(115200);
for(i=1;i<=2000;i++){
    val1=analogRead(A5);       // Value of LDR
```

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 28
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

```
    Serial.println(val1);
    if(val1<300){                // Threshold
        digitalWrite(11,HIGH);
    }
    else
    {
        digitalWrite(11,LOW);
    }
}
}
void loop() {
}
```

6.6 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=1:20                  // Run for 20 iterations
```

```
    p=cmd_analog_in(1,5);    // read analog pin 5 (ldr)
    disp(p);
    sleep(500)               // Delay of 500 milliseconds
end
c = close_serial(1)          // close serial connection
```

Scilab Code 6.2 Turning the blue 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=1:500 //Run for 500 iterations
    p=cmd_analog_in(1,5)        // read analog pin 5 (ldr)
    disp(p);
    if(p<300)                   // Setting Threshold value of 300
        cmd_digital_out(1,9,1)  // Turn ON LED
    else
        cmd_digital_out(1,9,0)  // Turn OFF LED
    end
end
close_serial(1)
```

Chapter 7

Interfacing a DC motor

Motors are widely used in commercial applications. DC motor converts electric power obtained from direct current to the mechanical motion. This chapter describes an experiment to control DC motor with Arduino Uno board. We will observe the direction of motion of DC motor being changed using the microcontroller on Arduino Uno board. Control instruction will be sent to Arduino Uno using Scilab scripts, Arduino IDE and Scilab Xcos.

7.1 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. 7.1 shows the diagram of a typical breakout board containing IC L293D, which will be used in this book.

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

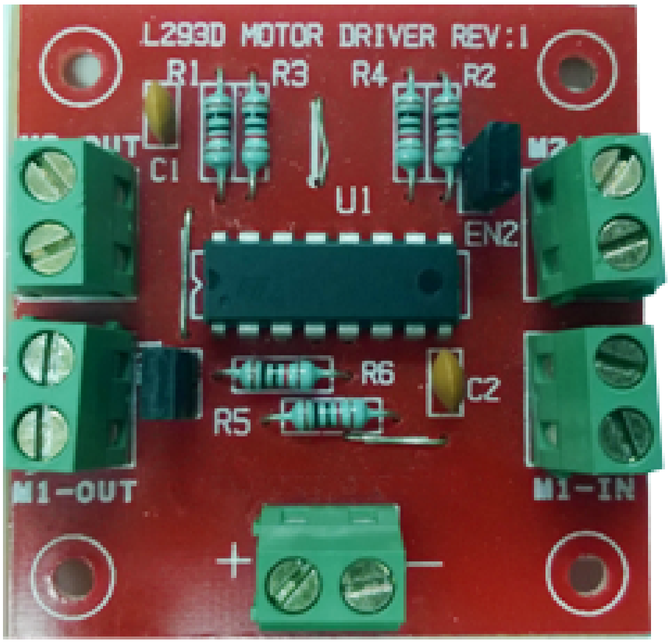


Figure 7.1: L293D motor driver board

"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 for each of which, the input can come from 8 bits. Thus we can generate 256 different analog values 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. 7.2. The actual connections can be seen in Fig. 7.3.

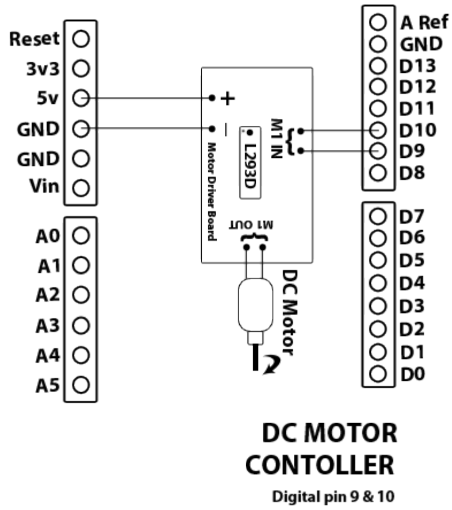


Figure 7.2: A schematic of DC motor connections

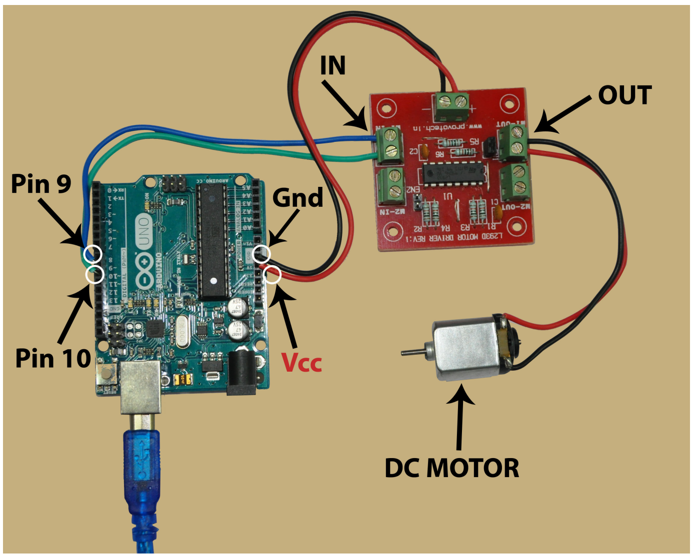


Figure 7.3: How to connect the DC motor to the Arduino Uno board

7.2 Controlling the DC motor from Arduino

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 four experiments in this section. We assume the shield to be attached to the Arduino Uno board while doing these experiments. The reader should go through the instructions given in Sec. 3.1 before getting started.

1. We now demonstrate how to drive the DC motor from the Arduino IDE. Arduino Code 7.1 has the required code for this. It starts the serial port at a baud rate of 9600. Pins 9 and 10 are declared as output pins and hence values can be written on to them. Next, we write PWM 100 on pin 9 and PWM 0 on pin 10. Recall from Fig. 7.3 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. Recall from Sec. 4.1 that a high on pin 9 also makes the blue LED come on. As a result, the blue LED also lights up.

   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.

   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, as we put a 0 in both pins 9 and 10, the motor comes to a halt. The blue LED is also turned off.

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 7.2. In this program, 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. This makes the green LED light up, recall the discussion in Sec. 4.1. After that, we release the motor by writing 0 in both pins 9 and 10. This turns the green LED off.

3. Next, we make the DC motor run in forward and reverse directions, in a loop. This is done through Arduino Code 7.3. We first put PWM 100 in the motor for 3 seconds. After that, make the motor stop for 2 seconds. Finally, make the motor rotate in the reverse direction by putting PWM -100 for two seconds. Finally, we make the motor stop for one second. The entire thing is put in a loop.

Exercise 7.1 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.

7.3 Controlling the DC motor from Scilab

In this section, we will explain a few experiments to rotate the DC motor. We will first initialize it and then rotate it clockwise and counterclockwise. We will explain some of the other required commands, such as sleep.

7.3.1 Initialization

In all the experiments in this section, we need to initialize the DC motor first, using a Scilab command of the following type:

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

As mentioned earlier, number 1 in the above list refers to the Arduino Uno board. We now discuss how to choose values for the other parameters in this command. As mentioned above, there are many H-bridge IC manufacturers. The inbuilt function `cmd_dcmotor_setup` can work with most of the widely used ICs, through a suitable input parameter. Users have to provide the type number of the breakout board they have. Popular numbering convention for different types of DC motor breakout boards is given in Table 7.1. For example, L293D is type 3. Next, we have to provide the motor number we want to control. In our case, it is number 1. Finally we want to provide PWM pin numbers on Arduino Uno. As mentioned earlier, we are using pins 10 and 11. In Table 7.2, we list the choices that we have made. Inserting these parameter values in the above shown Scilab command, we get the following command

```
2 cmd_dcmotor_setup(1,3,1,9,10)  // Setup DC motor of type 3 (L293D),
    motor 1, pin 9 and 10
```

which is line number 2 in Scilab Code 7.1. We have already seen the first two lines of this code and hence will not explain here. We will add more lines to this code as we go along.

7.3.2 Rotation for a specified time

We will now explain how to run the DC motor. We have to provide motor number and the PWM value. The Scilab command is of the form,

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

Table 7.1: A numbering convention used in the DC motor breakout board

DC Motor Type	Number
MotorShield Rev3	1
PMODHB5/L298	2
L293D	3

Table 7.2: Parameters for DC motor initialization

Parameter	Value
H-Bridge type	3
Motor number	1
PWM 1 pin	9
PWM 2 pin	10

Motor number is 1, as mentioned earlier. Considering that the input to a PWM pin comes from two 8 digital pins, we can provide 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 to run at an intermediate speed. Assigning these values, we get the following command:

```
3 cmd_dcmotor_run(1,1,100) // Motor 1 runs at PWM 100
```

This is line number 3 in Scilab Code 7.1. This command does not say for how long the motor should run. This is taken care of by the `sleep` statement. The units of sleep are milliseconds. For example, line number 4 of Scilab Code 7.1, given next, says that Scilab should go to sleep for three seconds.

```
4 sleep(3000)              // This is allowed to continue for 3 seconds
```

Line number 5 of Scilab Code 7.1, shown below, is mandatory for every program.

```
5 cmd_dcmotor_release(1,1) // Motor 1 is released
```

It releases the DC motor. The PWM functionality on the Arduino Uno pins is ceased using this command. This has the motor number as an input parameter.

If the sleep command discussed above 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. If on the other hand, the DC motor is not released (*i.e.*, line number 6 being absent), the DC motor will go on rotating. Line number 6 of Scilab Code 7.1 closes the serial port.

We encourage you to run the above code without either line numbers 4, 5 or 6 or all combinations. Go ahead and do it - you will not break anything. At the most, you may have to unplug the USB cable and restart the whole thing from the beginning.

Scilab Code 7.1 can easily be extended to make the DC motor run in both directions. The modified code is available in Scilab Code 7.2.

Exercise 7.2 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.

7.3.3 Using the capabilities of Scilab

Given that Scilab has a powerful programming syntax, a lot of different experiments can be tried out. We illustrate a few in this section. We begin with a `for loop`.

In the previous section, we presented Scilab Code 7.2, where we made the motor run in both directions, five seconds in the clockwise direction and two seconds in reverse. This code can be embedded in a loop and the motor be made to repeat a certain number of times. This idea is implemented through Scilab Code 7.3. Through the `for loop` in between line numbers 3 and 8, we make the DC motor repeat four times the cycle containing one rotation in each direction.

It is not difficult to see how some of the other features of the Scilab programming language can be used along with this DC motor. For example, it is possible to read a temperature value and based on its value, start or stop the motor. For real world applications, one has to provide extra current carrying capabilities through external hardware.

7.4 Driving the DC motor from Xcos

In this section, we will see how to drive the DC motor from Xcos. 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. If the rotation of the DC motor is blocked by any obstacle in any of the experiments given below, you may want to hold it in your hand and let it run unhindered.

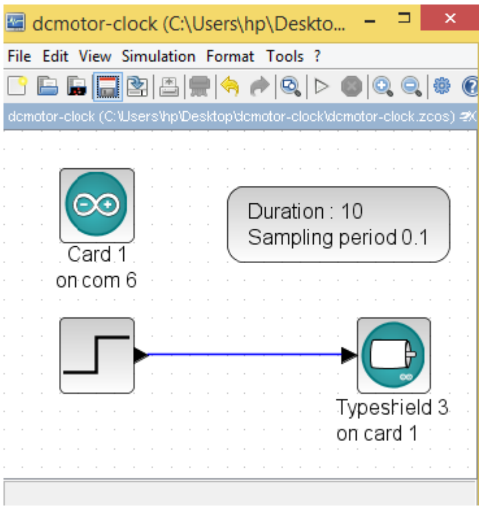


Figure 7.4: Control of DC motor for a specified time from Xcos. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-clock.zcos`, see Footnote 2 on page 2, is invoked.

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. 7.4. 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 7.3. In case of `DCMOTOR_SB`, enter 3 to indicate for L293D board. After clicking on OK, another 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. 7.5. 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 7.4. All other parameters are to be left unchanged.

3. Next, we will describe the Xcos code that drives the DC motor in a loop. When the file required for this experiment is invoked, one gets the GUI as in Fig. 7.6.

Table 7.3: 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 28
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

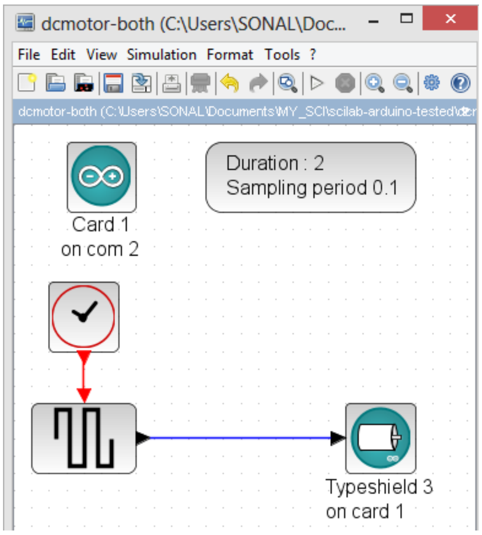


Figure 7.5: Xcos control of the DC motor in forward and reverse directions. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-both.zcos`, see Footnote 2 on page 2, is invoked.

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 7.5. All other parameters are to be left unchanged.

Table 7.4: 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 28
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

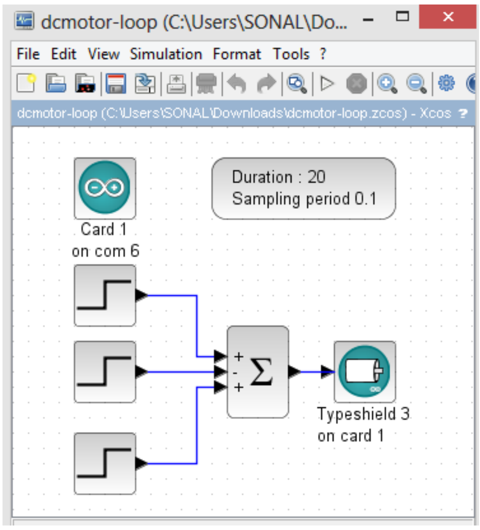


Figure 7.6: Xcos control of the DC motor in forward and reverse directions. This is what one sees when `Origin/user-code/dcmotor/scilab/dcmotor-loop.zcos`, see Footnote 2 on page 2, is invoked.

Exercise 7.3 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

Table 7.5: 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 28
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]

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.

∎

7.5 Arduino Code

Arduino Code 7.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/arduino/dcmotor-clock/dcmotor-clock.ino`, see Footnote 2 on page 2.

```
void setup() {
Serial.begin(9600); // set the baudrate=9600
pinMode(9,OUTPUT); // use pins 9 and 10 for motor output
pinMode(10,OUTPUT);
analogWrite(9,100); // PWM 100 on pin 9 makes the motor rotate
analogWrite(10,0);
delay(3000);        // This is allowed to continue for 3 seconds
analogWrite(9,0);   // 0 on pin 9 stops the motor
analogWrite(10,0);
}
void loop() {
// what is put here will run in an infinite loop
}
```

Arduino Code 7.2 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.

```
void setup() {
Serial.begin(115200); // set the baudrate=115200
pinMode(9,OUTPUT);    // use pins 10 and 11 for motor output
pinMode(10,OUTPUT);
analogWrite(9,100);   // Motor runs at a low speed
analogWrite(10,0);
delay(3000);          // 3 second delay
analogWrite(9,0);     //
analogWrite(10,100);  // Motor runs in the reverse direction for
delay(2000);          // 2 seconds
analogWrite(9,0);     // Motor is stopped
analogWrite(10,0);    //
}
void loop(){
  // Code here runs in an infinite loop
}
```

Arduino Code 7.3 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.

```
int i;
void setup() {
Serial.begin(115200);    // set the baudrate=115200Hz
pinMode(9,OUTPUT);       // use pins 9 and 10 for motor output
pinMode(10,OUTPUT);
for(i=0; i<4; i++){
  analogWrite(9,100);    // Motor runs at a low speed
  analogWrite(10,0);
  delay(3000);           // 3 second delay
```

```
  analogWrite(9,0);
  analogWrite(10,0);      // Motor stops for
  delay(2000);            // 1 seconds
  analogWrite(9,0);       //
  analogWrite(10,100);    // Motor runs in the reverse direction for
  delay(2000);            // 2 seconds
  analogWrite(9,0);       // Stop the
  analogWrite(10,0);      // motor rotating
  delay(1000);            // for 1 second
}
}
void loop(){
}
```

7.6 Scilab Code

Scilab Code 7.1 Rotating the DC motor. Available at `Origin/user-code/dcmotor/scilab/dcmotor-clock.sce`, see Footnote 2 on page 2.

```
ok = open_serial(1,4,115200)    //COM port is 4 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 7.2 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(1,4,115200)    //COM port is 4 and baud rate is 115200
cmd_dcmotor_setup(1,3,1,10,11)  // 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 7.3 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(1,4,115200)//COM port is 4 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 x=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)
```

Chapter 8

Interfacing a Potentiometer

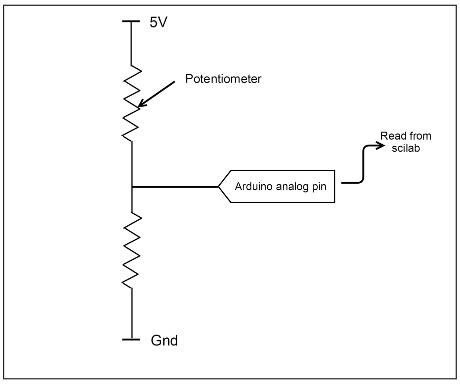
A potentiometer is a three-terminal variable resistor with two terminals connected to the two ends of a resistor and one connected to a sliding or rotating contact, termed as a wiper. The wiper can be moved to vary the resistance, and hence the potential, between the wiper and each terminal of the resistor.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

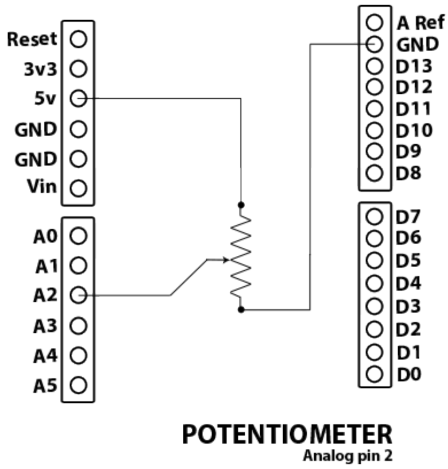
8.1 Preliminaries

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 42. 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. 8.1.



(a) Pictorial representation of a potentiometer



(b) Schematic representation of the potentiometer

Figure 8.1: Potentiometer's schematic on the shield

The reading of a potentiometer is an analog voltage varying from 0 to 5V. As for LDR, we use the ADC functionality of the Arduino Uno board. Thus, we obtain digital values between 0 and 1023 in Scilab Console or Arduino Serial Monitor. 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.

8.2 Reading the potentiometer from the Arduino IDE

In this section, we shall learn to read the potentiometer input through Arduino IDE. Depending on the acquired potentiometer values, we will change the state of RGB LED. The Arduino code for this experiment is given in Arduino Code 8.1. Lines 1 through 4 are used to assign relevant pin numbers to potentiometer and RGB LED. The purpose of these lines is to avoid confusion, with the pin numbers, for the beginners. Next, we start serial port communication, as on line 9, with the baud rate of 115,200 bps. In order to take the potentiometer input, we need to initialize the pins by giving the following commands:

```
  pinMode(RGB_RED,OUTPUT);
  pinMode(RGB_GREEN,OUTPUT);
  pinMode(RGB_BLUE,OUTPUT);
  for(i=0;i<10;i++){
```

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 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,

```
    digitalWrite(RGB_RED,LOW);
  }else if(p>=320 &p<=900) {         //threshold 2
    digitalWrite(RGB_GREEN,HIGH);
    delay(1000);
    digitalWrite(RGB_GREEN,LOW);
```

In a similar manner, we check the potentiometer values and correspondingly turn on and off the Green and Blue LEDs. Note that, we used `if and else if` statements to check the conditions and rotated the potentiometer knob to vary the resistance.

8.3 Reading the potentiometer from Scilab Script

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. 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 8.1. 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,

```
    p=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 range, and then turn on and off the corresponding LED. For example,

```
        cmd_digital_out(1,11,0)
    elseif p>=320 &p<=900          //threshold 2
```

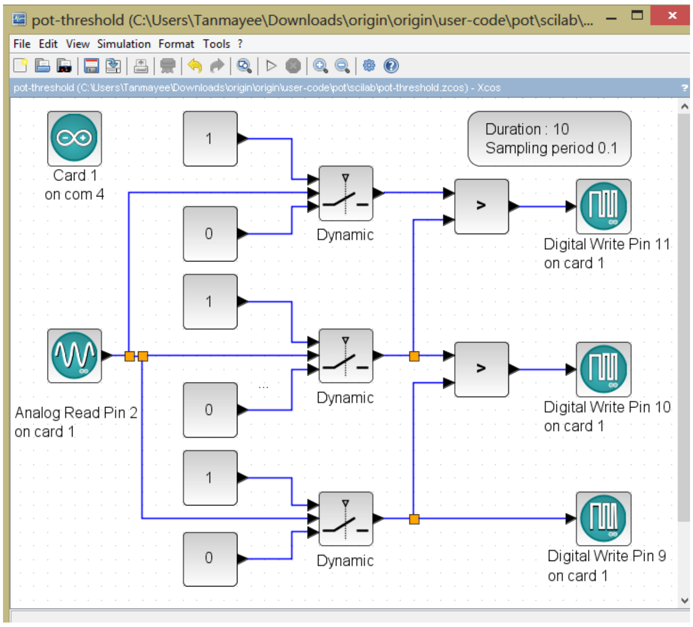


Figure 8.2: Turning LEDs on through Xcos depending on the potentiometer threshold. This is what one sees when `Origin/user-code/pot/scilab/pot-threshold.zcos`, see Footnote 2 on page 2, is invoked.

```
        cmd_digital_out(1,10,1)
        sleep(1000)
        cmd_digital_out(1,10,0)
```

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 the other two bands. Note that we need to vary the resistance by rotating the knob of the potentiometer.

8.4 Reading the potentiometer from Scilab Xcos

In this section, we discuss how to perform the experiment explained above. When the file required for this experiment is invoked, one gets the GUI as in Fig. 8.2. 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.

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.

Table 8.1: 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 28
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

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.

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 potentiometer.

8.5 Exercise

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.

8.6 Arduino Code

Arduino Code 8.1 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.

```
const int POT = 2;
const int RGB_RED = 11;
const int RGB_GREEN = 10;
const int RGB_BLUE = 9;
int p=0;
int i=0;
void setup() {
  Serial.begin(115200);
  pinMode(POT,INPUT);
  pinMode(RGB_RED,OUTPUT);
  pinMode(RGB_GREEN,OUTPUT);
  pinMode(RGB_BLUE,OUTPUT);
  for(i=0;i<10;i++){
     p = analogRead(POT);
  if(p>=0 & p<320) {                //threshold 1
    digitalWrite(RGB_RED,HIGH);
    delay(1000);
    digitalWrite(RGB_RED,LOW);
  }else if(p>=320 &p<=900) {         //threshold 2
    digitalWrite(RGB_GREEN,HIGH);
    delay(1000);
    digitalWrite(RGB_GREEN,LOW);
  }else if(p>900 & p<=1023) {        //threshold 3
    digitalWrite(RGB_BLUE,HIGH);
    delay(1000);
    digitalWrite(RGB_BLUE,LOW);
  }
  }
}
void loop() {
}
```

8.7 Scilab Code

Scilab Code 8.1 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=1:10//Run for 10 iterations
    p=cmd_analog_in(1,2)
    if(p>=0 & p<320) then        //threshold 1
        cmd_digital_out(1,11,1)
        sleep(1000)
        cmd_digital_out(1,11,0)
    elseif p>=320 &p<=900        //threshold 2
        cmd_digital_out(1,10,1)
        sleep(1000)
        cmd_digital_out(1,10,0)
    elseif p>900 & p<=1023       //threshold 3
        cmd_digital_out(1,9,1)
        sleep(1000)
        cmd_digital_out(1,9,0)
    end
end
close_serial(1)
```

Chapter 9

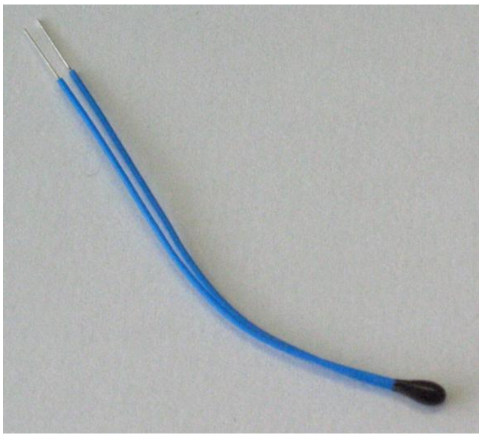
Interfacing a Thermistor

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.

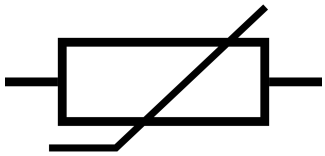
Thermistors are available in a variety of shapes such as beads, rods, flakes, and discs. Due to their compact size and low cost, they are widely used in the applications where even imprecise temperature sensing is sufficient. They, however, suffer from noise and hence need noise compensation. In this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

9.1 Preliminaries

A typical thermistor and its symbolic representation are shown in 9.1a and 9.1b 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 9.2a and 9.2b. Nevertheless, the user doesn't need to connect any wire or component explicitly.

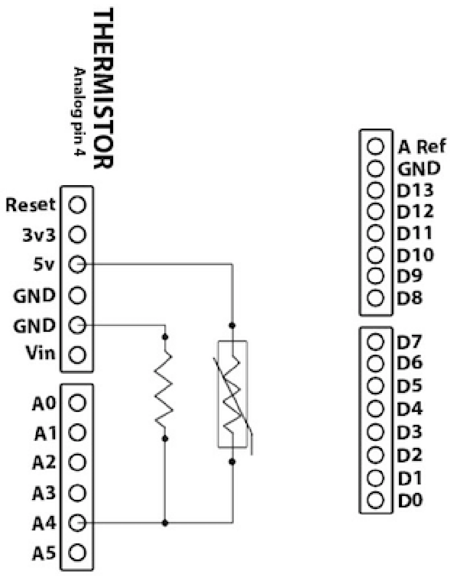


(a) Pictorial representation of a thermistor [?]

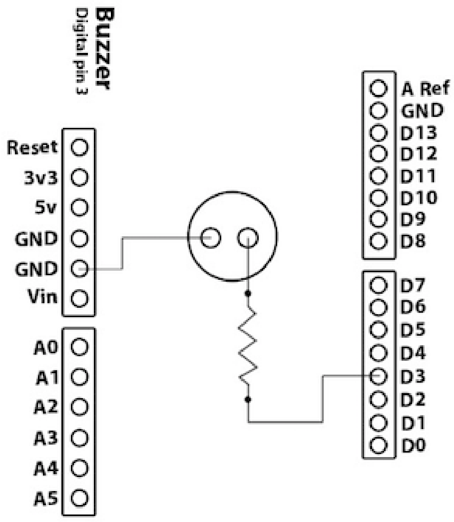


(b) Symbolic representation of a thermistor

Figure 9.1: Pictorial and symbolic representation of a thermistor



(a) Thermistor connection diagram



(b) Buzzer connection diagram

Figure 9.2: Thermistor and buzzer connection diagrams

9.2 Reading thermistor from Arduino IDE

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.

1. A simple code to read the values from thermistor is given in Arduino Code 9.1. The arduino IDE based command for the analog read functionality is given by.

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

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

   ```
   Serial.println(value); //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. 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.

2. In this experiment, we will turn the buzzer on and off depending on the temperature sensed by the thermistor. The program for this is available at Arduino Code 9.2. We shall use the ADC output to carry this out. The buzzer is connected on 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 value 550. Temperature of the thermistor can be raised by just holding it for a while. Doing so will transfer heat from the person holding the thermistor, thereby raising the temperature of the thermistor. As soon as the ADC output exceeds 550, the buzzer is given a digital high signal, turning it On. A delay of half a second is introduced before the next value is read. This loop is executed 100 times.

Exercise 9.1 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.

∎

9.3 Reading thermistor from Scilab scripts

In this section we will explain a few Scilab scripts to read values from thermistor and to use them. The `cmd_analog_in` command will be used to read from thermistor connected to an analog input pin. The experiments will be carried out using Scilab.

1. In the first experiment, Scilab Code 9.1 is used to read values from thermistor. First the serial port is opened using the command `open_serial` and passing the correct port number to it. The command `cmd_analog_in` is used to read from the analog pin. The pin number is passed to this command as an argument. The read value is stored in some variable. The value is then displayed on the scilab console. A sleep of 500 millisecond is executed using the `sleep` command and then the reading process is repeated 20 times by putting it in a `for` loop. After the loop is finished the serial port is closed using the `close_serial` command.

2. In the second experiment, we will use the Scilab script to turn a buzzer on and off using the thermistor values. 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. In this experiment we compare the ADC output value with 550 and as soon as the value exceeds 550 the buzzer is turned on. See Scilab Code 9.2 for the complete code. We use `if` loop to achieve this functionality. Command `cmd_digital_out` is used to turn the buzzer on and off. The correct port number on which the buzzer is connected is passed to this command as an argument. The entire process is repeated 500 times.

Exercise 9.2 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,

$$\frac{1}{T} = A + B * \ln(R) + C * (\ln(R))^3 \tag{9.1}$$

equation 9.1 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,

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{\beta} * \ln\left(\frac{R}{R_0}\right) \tag{9.2}$$

equation 9.2 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.

∎

9.4 Reading thermistor 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.

1. The xcos diagram for performing the simple thermistor read operation is as shown in Fig. 9.3. 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 9.1. All other parameters are to be left unchanged.

   The thermistor readings can be varied by bringing a heating or cooling source in the vicinity of it. The graph as shown in Fig. 9.4 will show the variations in the ADC output that is displayed.

2. In the second experiment, we will switch On and Off a buzzer depending on the thermistor readings (ADC output). The xcos diagram for this experiment is as shown in Fig. 9.5. The parameters of the blocks can be changed by right clicking on the block and choosing `Block Parameters`. One can also double

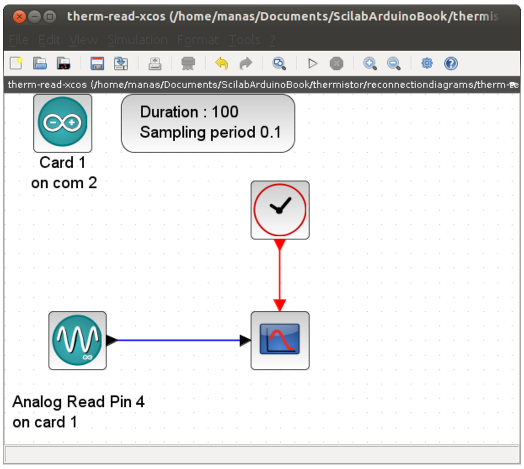


Figure 9.3: Xcos diagram to read thermistor values. This is what one sees when `Origin/user-code/thermistor/scilab/therm-read-xcos.zcos`, see Footnote 2 on page 2, is invoked.

Table 9.1: 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 28
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

click on the block. The values for each block is tabulated in Table 9.1. All other parameters are to be left unchanged.

The graph as shown in Fig. 9.6 will show the variations in the ADC output that is displayed and the corresponding switching of buzzer whenever the limits are crossed.

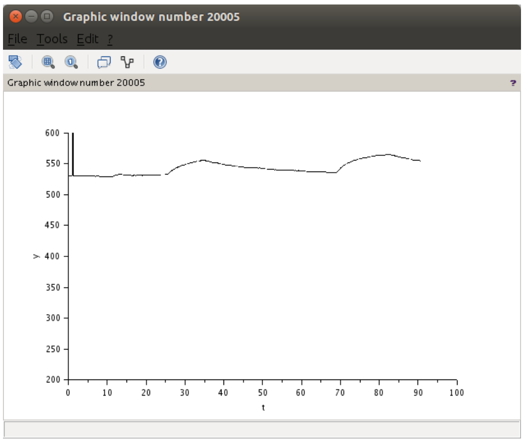


Figure 9.4: Output of Xcos diagram to read thermistor values. This is what one sees when `Origin/user-code/thermistor/scilab/therm-read-xcos.zcos`, see Footnote 2 on page 2, is executed.

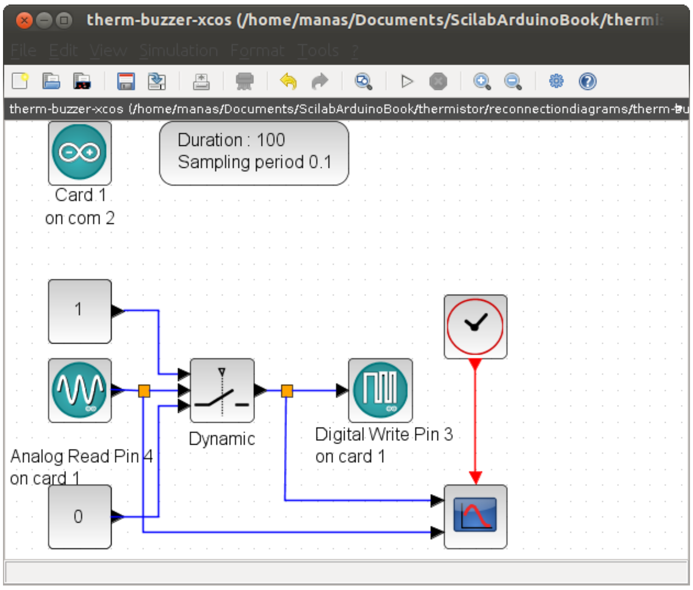


Figure 9.5: Xcos diagram to read the value of the thermistor, which is used to turn the buzzer on or off. This is what one sees when `Origin/user-code/thermistor/scilab/therm-buzzer-xcos.zcos`, see Footnote 2 on page 2, is invoked.

Table 9.2: 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 28
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

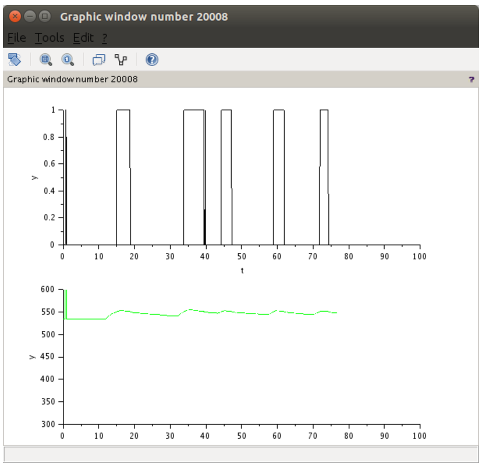


Figure 9.6: Output of Xcos diagram to switch buzzer through thermistor values. This is what one sees when `Origin/user-code/thermistor/scilab/therm-buzzer-xcos.zcos`, see Footnote 2 on page 2, is executed.

9.5 Arduino Code

Arduino Code 9.1 Read and display the thermistor values. Available at `Origin/user-code/thermistor/arduino/therm-read/therm-read.ino`, see Footnote 2 on page 2.

```
int value;
int i;

void setup()
{
  Serial.begin(115200);
  for(i=1;i<=40;i++)
  {
    value=analogRead(A4); //read value from thermistor
    Serial.println(value); //display
    delay(500);
  }

}

void loop()
{
}
```

Arduino Code 9.2 Turning the buzzer on and off 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.

```
int value;
int i;

void setup()
{
  pinMode(3,OUTPUT);
  Serial.begin(115200);

  for(i=1;i<100;i++)
  {
    value=analogRead(A4); //read value from thermistor
    Serial.println(value); //display

    if(value>550)
    {
      digitalWrite(3,HIGH); // Turn ON buzzer
    }
    else
    {
```

```
      digitalWrite(3,LOW); // Turn OFF buzzer
    }
    delay(500);
  }
}

void loop()
{
}
```

9.6 Scilab Code

Scilab Code 9.1 Read and display the thermistor values. Available at `Origin/user-code/thermistor/scilab/therm-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=1:20                  // Run for 20 iterations
    p=cmd_analog_in(1,4);   // read analog pin 4 (thermistor)
    disp(p);
    sleep(500)              // Delay of 500 milliseconds
end
c = close_serial(1)         // close serial connection
```

Scilab Code 9.2 Turning the buzzer on and off 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=1:500 //Run for 500 iterations
    p=cmd_analog_in(1,4)        // read analog pin 4 (thermistor)
    disp(p);
    if(p>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
close_serial(1)
```

Chapter 10

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 Arduino IDE, Scilab and Xcos. We will give code for all the experiments.

10.1 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 (PWM), Vcc and ground. We now explain how to connect a typical servomotor to the Arduino Uno board, through Table 10.1.

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

Table 10.1: Connecting a typical servomotor to Arduino Uno board

Servomotor terminal	Arduino board
Position signal	9
Ground (black/brown wire)	Ground
Vcc (red or orange middle-wire)	5V
Signal (orange or yellow)	Pin 9

10.2 Control through Arduino IDE

In this section, we will describe some experiments that will help rotate the servomotor based on the command given from Arduino IDE. We will also give the necessary code. We will present four experiments in this section. The shield has to be attached to the Arduino Uno board before doing these experiments. 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° using Arduino Code 10.1. Line 1 of this code includes a header file that initializes some of the parameters. Line 2 creates a `Servo` object and calls it `myservo`. Most Arduino boards allow the creation of 12 servo objects. Line 4 commands myservo to be attached to pin 9. Line 5 asks the servomotor to rotate by 90°. Other commands are as in the previous chapters.

   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 10.2. In Line 6, we provide a delay of one second. What is the reason? 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 10.3. 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°. Please see below a few exercise questions.

4. Finally, in the last experiment, we read the potentiometer value from the shield

and use it to drive the servomotor, see Arduino Code 10.4. The resistance of the potentiometer is represented in 10 bits. As a result, the resistance value could be any one of 1,024 values, from 0 to 1,023. This entire range is mapped to 180°. By rotating the potentiometer, one can make the motor move by different amounts.

The potentiometer is connected to pin number 2. Through this pin, the resistance of the potentiometer, in the range of 0 to 1,023, 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 half a second after every move. The loop is executed 5,000 times, with half a second delay for each iteration. During this period, the servomotor keeps moving as dictated by the resistance of the potentiometer.

Exercise 10.1 Let us carry out this exercise:

1. In Arduino Code 10.3, 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 10.3, what happens?
3. What happens if 180 in Line 10 of Arduino Code 10.4 is changed to 90? What does the change 180 to 90 mean?

∎

10.3 Control through Scilab Scripts

In this section, we will carry out the servomotor control experiments using Scilab. We will follow the same order as in Sec. 10.2. We assume that the shield is attached to the Arduino Uno board while doing these experiments. They will work without the shield also, but in this case, our comments on colour LEDs lighting will not be applicable. 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 which is given in Arduino Code 10.1. It first opens com port 2 in Arduino Uno card

number 1 with baud rate of 115200. If the port opening is unsuccessful `ok` will not be 0 and the program terminates, asking the user to correct the problem. Else If the port opening is successful, `ok` will be 0 and the program proceeds. In Line number 3 of the code, *i.e.*,

```
3 cmd_servo_attach(1,1)  // To attach the motor to pin 9
```

we say that the servomotor is attached on board 1 (the first entry) to pin 1 (the second entry). In the Scilab toolbox, pin 1 and pin 9 are connected and as a result, we connect the wire physically to pin 9. Similarly, pins 2 and 10 are connected through the Scilab toolbox.

2. In Scilab Code 10.2, we make the servomotor rotate to 90°, wait for a second and go to 45°. As mentioned earlier, the angles are absolute with respect to a fixed reference point and not relative.

3. In the next experiment, we rotate the servomotor in discrete steps of 20°. This is achieved by multiplying 20° by an integer `i`, which varies from 0 to 10. Once the maximum angle reaches 180°, it stops.

4. Finally, in the last experiment, we position the servomotor through the potentiometer in the code Scilab Code 10.4. As we rotate the potentiometer, the servomotor's angle also changes. The potentiometer value is read through pin 2, in line number 5, as below:

```
5     p=cmd_analog_in(1,2)  // Read potntiometer value
```

This value is mapped into a value between 0 and 180° by multiplying with 180/1023 in line 6:

```
6     p=floor(p*(180/1023)) // Scale Potentiometer value to 0-180
```

The `floor` function gets the integer part of the number by truncation. This is the angle by which the potentiometer is to be moved. Truncation is a not a crucial calculation, however. In every iteration, the servomotor's position is calculated, and placed for half a second. This loop is iterated upon 5,000 times.

10.4 Control through Xcos code

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.

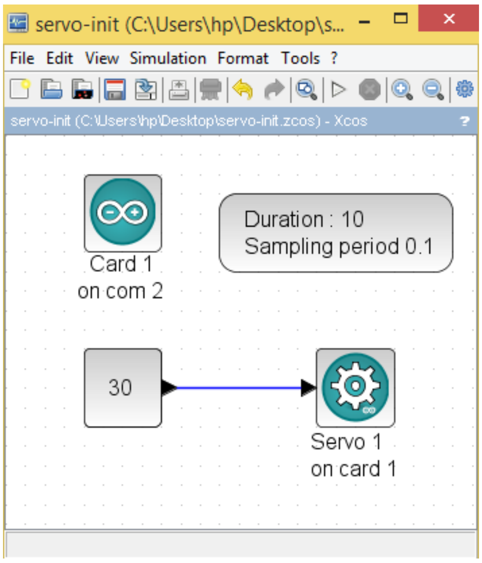


Figure 10.1: Rotating the servomotor by a fixed angle. This is what one sees when `Origin/user-code/led/scilab/servo-init.zcos`, see Footnote 2 on page 2, is invoked.

Table 10.2: 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 28
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

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. 10.1. 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. All other parameters are to be left unchanged.

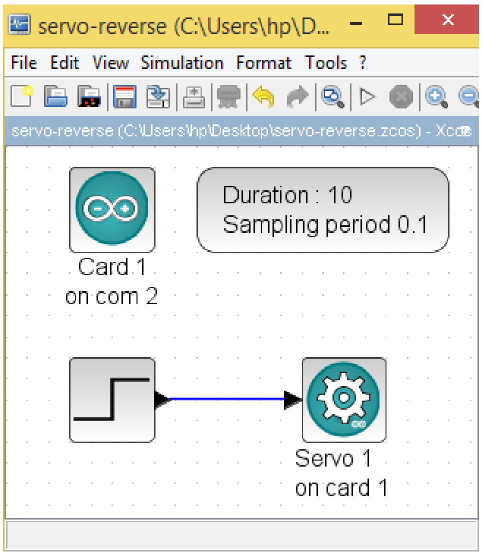


Figure 10.2: Rotating the servomotor forward and then reverse. This is what one sees when `Origin/user-code/led/scilab/servo-reverse.zcos`, see Footnote 2 on page 2, is invoked.

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. 10.2. 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 10.3. 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. 10.3. 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. `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 from the potentiometer. When the file required for this experiment is invoked,

Table 10.3: 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 28
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

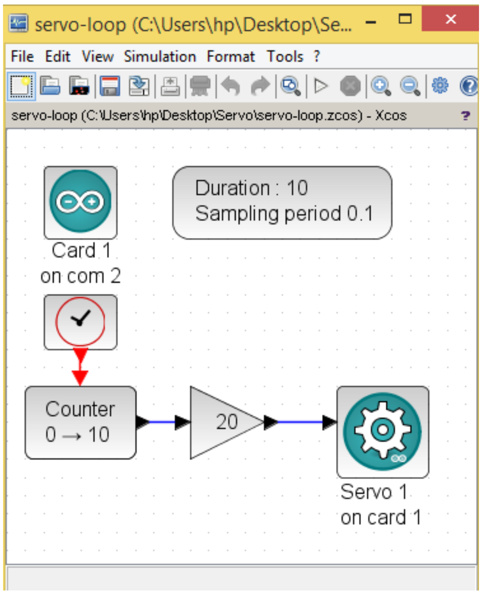


Figure 10.3: Rotating the servomotor in increments of 20°. This is what one sees when `Origin/user-code/led/scilab/servo-loop.zcos`, see Footnote 2 on page 2, is invoked.

one gets the GUI as in Fig. 10.4. 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 10.5. All other

Table 10.4: 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 28
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

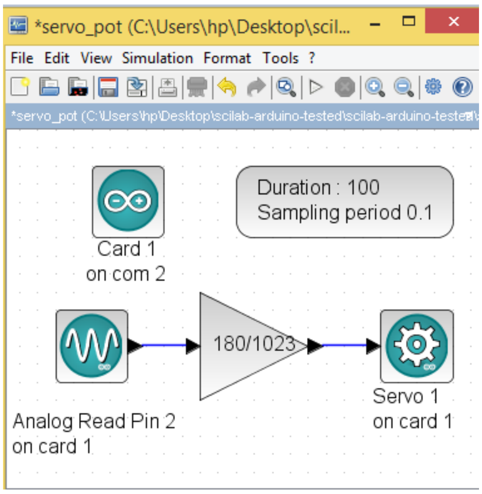


Figure 10.4: Rotating the servomotor as suggested by the potentiometer. This is what one sees when `Origin/user-code/led/scilab/servo-pot.zcos`, see Footnote 2 on page 2, is invoked.

parameters are to be left unchanged. The `ANALOG_READ_SB` block reads the value of potentiometer and is converted into rotation angle (180/1023), computed by `GAIN_f`.

Table 10.5: 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 28
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

10.5 Arduino Code

Arduino Code 10.1 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.

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

Arduino Code 10.2 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(9); // attach the servo object on to pin 9
  myservo.write(90); // tell servo to rotate by 90 degrees
  delay(1000);
  myservo.write(45);
  delay(1000);
  myservo.detach();
```

```
}
void loop() {
}
```

Arduino Code 10.3 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(9);       // attach the servo object on to pin 9
    myservo.write(angle*i); // tell servo to rotate by 20 degrees
    delay(1000);             // waits for a sec
  }
  myservo.detach();
}
void loop() {
}
```

Arduino Code 10.4 Rotating the servomotor through the potentiometer. Available at `Origin/user-code/servo/arduino/servo-pot/servo-pot.ino`, see Footnote 2 on page 2.

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
int potpin = 2; // analog pin used to connect the potentiometer
int val;        // variable to read the value from the analog pin
int i;
void setup(){
  Serial.begin(115200);
  myservo.attach(9);         // attach the servo object on to pin 9
  for(i=0;i<5000;++i){
  val = analogRead(potpin); // reads a value in (0,1023) through pot
  val = map(val, 0, 1023, 0, 180); // maps it in the range (0,180)
    degrees
  myservo.write(val);        // moves the motor to the mapped degree
  delay(500);                // waits for a second for servo to reach
  }
  myservo.detach();
}
void loop(){
}
```

10.6 Scilab Code

Scilab Code 10.1 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 9
cmd_servo_move(1,1,30) // tell servo to rotate by 30 degrees
sleep(1000)
close_serial(1)
```

Scilab Code 10.2 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 9. 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
close_serial(1)               // To close the connection safely
```

Scilab Code 10.3 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
cmd_servo_attach(1,1)       // Attach motor to pin 9. 1 means pin 9.
sleep(1000)
angle=20;                     // Angle by which it has to move
for i=0:10
   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
```

Scilab Code 10.4 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.

```
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) // Attach the motor to pin 9
for i=1:5000                // 5,000 itterations
    p=cmd_analog_in(1,2)    // Read potntiometer value
    p=floor(p*(180/1023))   // Scale Potentiometer value to 0-180
    cmd_servo_move(1,1,p)   // Command the servo motor
    sleep(500)              // sleep for 500 milliseconds
end
cmd_servo_detach(1,1)// Detach the motor
close_serial(1)
```

Chapter 11

Implementation of Modbus Protocol

In this chapter we will learn one of the advanced applications that can be built using Scilab-Arduino toolbox. Beginners might want to skip this chapter in the first reading. This experiment enables interfacing Modbus based devices with Scilab-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. 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 in order 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 or devices supplying the information are called Modbus Slaves/Server. All the slaves/servers have a unique id and address. Typically, there is one Master and maximum 247 Slaves.

During the communications on a Modbus network, the protocol determines how the controller gets to know its device address, recognizes the message provided and decides the action to be taken and accordingly extracts data and information contained in the message. 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. The Energy Meter used in this experiment supports Modbus RTU protocol. In Modbus RTU, the data is coded in binary and requires only one com-

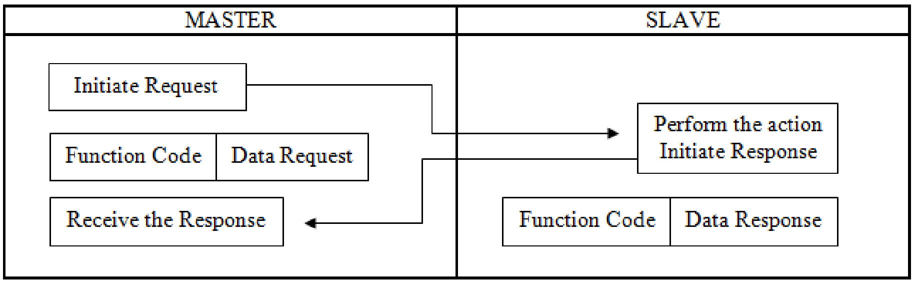


Figure 11.1: Block diagram representation of the Protocol

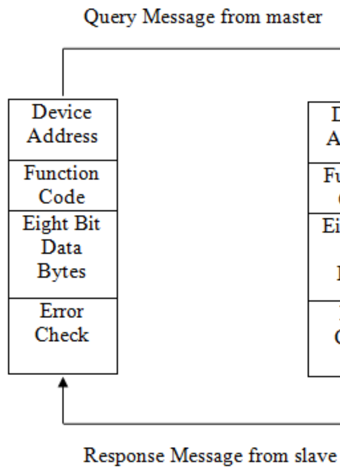


Figure 11.2: Master-Slave Query-Response Cycle

munication byte. This is ideal for use over RS232 or RS485 networks at baud rates between 1200 and 115K.

The 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. 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.

Energy Meter is a device that measures amount of electricity consumed by the load. We are using Energy Meter EM6400, which is a multifunction digital power meter by Schneider Electric India. It reads various parameters such as phase voltage, current, active power, reactive power, power factor etc. Before using the meter, one

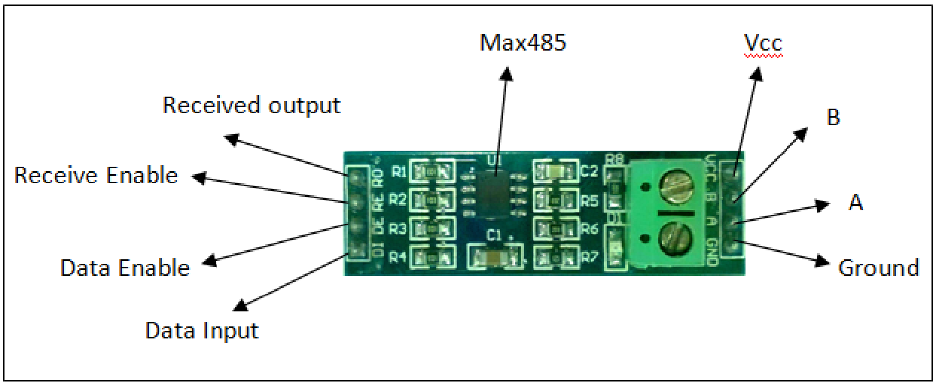


Figure 11.3: Pins in RS485 module

has to program system configuration, PT, CT ratios, communication parameters through front panel keys. EM6400 supports Modbus 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. All the parameter values are stored in the holding registers. Different holding registers hold values of different parameters. Individual parameter addresses can be found in user manual for EM6400. For example,

Current (phase 1):	3929
Voltage (phase 1):	3927
Active power (phase 1):	3919

The size of each Modbus register is 16 bits and all EM6400 readings are 32 bits. So, each reading occupies two consecutive Modbus registers. Values in every register are in little endian format (1st register contains LSB and next register contains MSB). In our case, Energy Meter is a slave and slave addresses can be set between 1 and 247.

A request to read holding registers has to be sent in a specified format. An example of a request packet is as follows. Suppose that the request is 01 03 0F56 0002 270F. Its meaning is explained in Table 11.1. The response packet corresponding the above request packet is given as 01 03 04 2921 4373 D2B0. Its meaning is explained in Table 11.2.

Values in required registers are 43732921 in hex (since obtained values are being read in little endian format) which is 243.16 when converted to floating point using

Table 11.1: Interpretation of a request packet

01	Slave address
03	Function code to read holding registers
0F56	Data Address of the first requested register (address for voltage phase1 to neutral) and (0F56 hex = 3927, +40001 offset = 43928)
0002	Total number of registers requested for read
270F	CRC (Cyclic Redundancy Check) for error checking (LSB first)

Table 11.2: Interpretation of a response packet

01	Slave address
03	Function code to read holding registers
04	Total number of bytes read
2921	Data in 1st requested register
4373	Data in 2st requested register
D2B0	CRC for error checking (LSB first)

IEEE 754 norms. Obtained value is a voltage (phase1 to neutral) which is 243.16 Volts.

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. For example, if a two byte integer 0x5E5F is stored on disk by one machine with the 0x5E (high byte or MSB) stored at the lower memory address and the 0x5F (low byte or LSB) stored at a higher memory address, but a different machine reads that integer by picking 0x5F for the high byte and the 0x5E for the low byte, giving 0x5F5E, thus resulting into an 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 byte ordering:

Little Endian: If the hardware is designed so that the lowest or the least significant byte (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 get stored in higher (increasing)

Table 11.3: Hexadecimal to Decimal

Four Bytes Integer Reading from Meter			
Memory Address	Memory Address	Little Endian	Big Endian
3900	8A43	MSB	LSB
3901	436B	LSB	MSB

Table 11.4: Single and Double Precision Representation

Single	Sign (1 bit)	Exponent (8 bit)	Mantissa (23 bit)
Double	Sign (1 bit)	Exponent (11 bit)	Mantissa (52 bit)

memory locations.

Big Endian: Here, the hardware is designed so that the highest or the most significant byte (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. Quite obvious, the "little" end byte, LSB is 0x84A3, and the "big" end byte, MSB is 0x436B; taking into consideration that the Read Holding Registers are 16 bits each. Thus the aforesaid memory storage patterns for the integer would be Table 11.3.

In order 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: Used for 32 bits. Out of those 32 bits, one bit represents the sign bit, 8 bits for exponent and the remaining 23 bits for mantissa, as depicted in Table 11.4.

2. Double Precision: Used for 64 bits. Out of those 32 bits, one bit represents the sign bit, 8 bits for exponent and the remaining 23 bits for mantissa, as depicted in Table 11.4.

Finally, the decimal value is given by, Decimal Value $= (-1) * \text{sign} * 2^{exponent} *$ Mantissa. Hence, for 32 bit values, the sign is stored in bit 32. The exponent can be calculated from bits 24-31 by subtracting 127. The mantissa is stored in bits 1-23. An invisible leading bit (i.e. it is not actually stored) with value 1.0 is placed in front, then bit 23 has a value of 1/2, bit 22 has value 1/4 etc. As a result, the mantissa has a value between 1.0 and 2. At last, the decimal value is calculated using the above mentioned equation. Though there are several online converters available as IEEE 754 Converter, a function has been formulated in Scilab for this conversion here.

11.2 Objective

The objective of this experiment is to make the user acquainted with the use of Modbus protocol through Arduino Uno. It gives an insight on how to acquire readings from the Energy Meter and interpret them accordingly. As mentioned earlier, 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 Scilab and Arduino Uno. For data transmission, we have used 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 from the Energy Meter (Slave).

This Arduino Uno is then connected to the Energy Meter via a MAX485 chip which facilitates long distance communication. The information packet is sent 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. Data which is in Little Endian hex format is obtained from this and is converted to floating point number using IEEE 754.

11.3 Energy Meter set up for Modbus protocol with Arduino Uno

1. As we know, Arduino Uno has 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 for serial communication. Pin 10 (used as RX) and Pin 11(used as TX) is connected to RO (Receive Out) and DI (Data In) pins of MAX485 module respectively.

2. DE (Data Enable) and RE (Receive Enable) pins of RS 485 are shorted and connected to digital pin 3 of Arduino Uno. This serves as Control Pin which

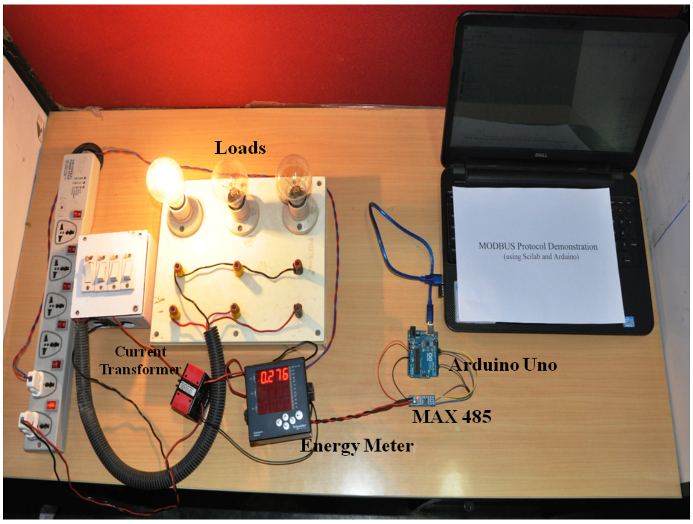


Figure 11.4: MODBUS Set Up for Energy Meter

will control when to receive and transmit serially.

3. Vcc and GND of the MAX485 module are connected to Vcc and GND of Arduino.

4. A and B pins of MAX485 are connected to A (Pin 7) and B (Pin 14) pins of the Energy Meter (meant for RS485 communication).

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

11.4 Software

Software for the demonstration comprises two parts:

1. Arduino Uno firmware code: This code is written to communicate with Scilab (using serial interface), and with MAX485 chip (using Software Serial interface). Control logic to enable receive and transmit modes of MAX485 chip is also present in Arduino Uno firmware code.The overall implementation is being described in Fig. 11.6.

2. Scilab code: This code requests Energy Meter readings by sending request packet to Arduino Uno from Scilab. Then it waits till requested packet is available from Arduino Uno. After receiving the packet, it extracts data from

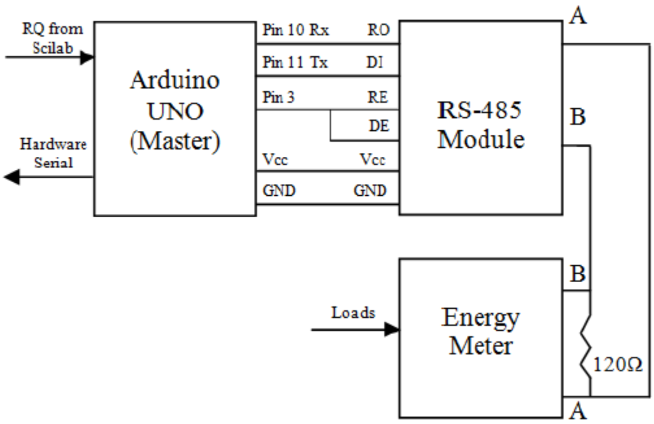


Figure 11.5: Block Diagram for Energy Meter Setup

the packet and converts it into IEEE 754 floating point format. The overall implementation is being described below:

(a) Frame request packet to be sent to slave in ASCII coded decimal format

(b) Send the packet serially to Arduino Uno board (Arduino Uno sends this packet to Energy Meter via RS 485 module)

(c) Read the response packet available on Arduino Uno board (sent by Energy meter to Arduino via RS 485)

(d) Extract holding register contents from received packet

(e) Convert 32 bit register contents which are in little endian format to floating point number using ieeesingle2num function

(f) Display the value of electrical parameter read(i.e. voltage, current or power)

11.5 Output

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

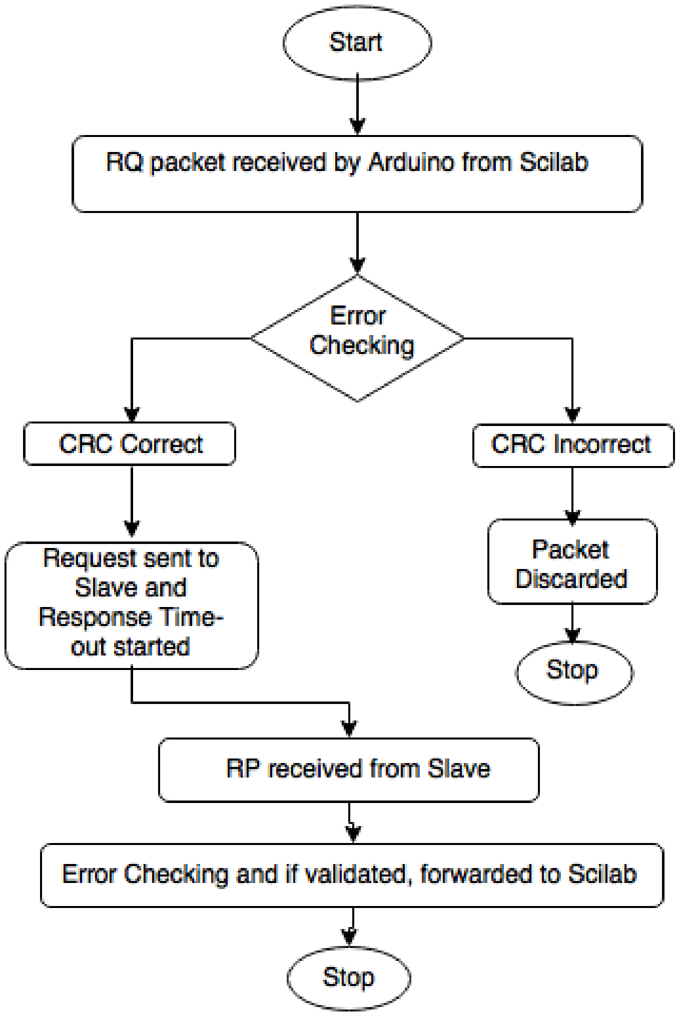


Figure 11.6: Flowchart of Arduino firmware

3. Single phase active power outputs are shown in Fig. 11.12 and Fig. 11.13.

In output, user could see the requested energy parameter on Scilab console. For demonstration we have taken single phase current, single phase voltage and single phase active power reading. We can always verify the Scilab output with the value being displayed on the Energy Meter display screen.

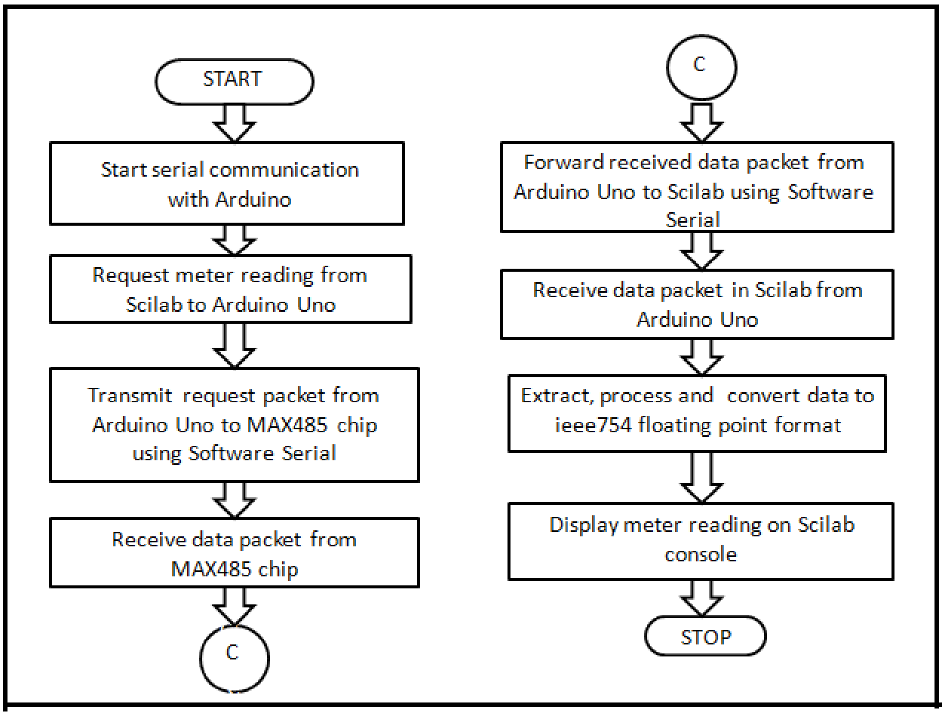


Figure 11.7: Flow Chart of the Modbus Energy Meter Implementation

11.6 Reading 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.

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 11.5. All other parameters are to be left unchanged.

11.6.1 Troubleshooting

After we send the query using Modbus protocol from Scilab (using write_serial command) to the Energy Meter, we will receive a packet from the Energy Meter

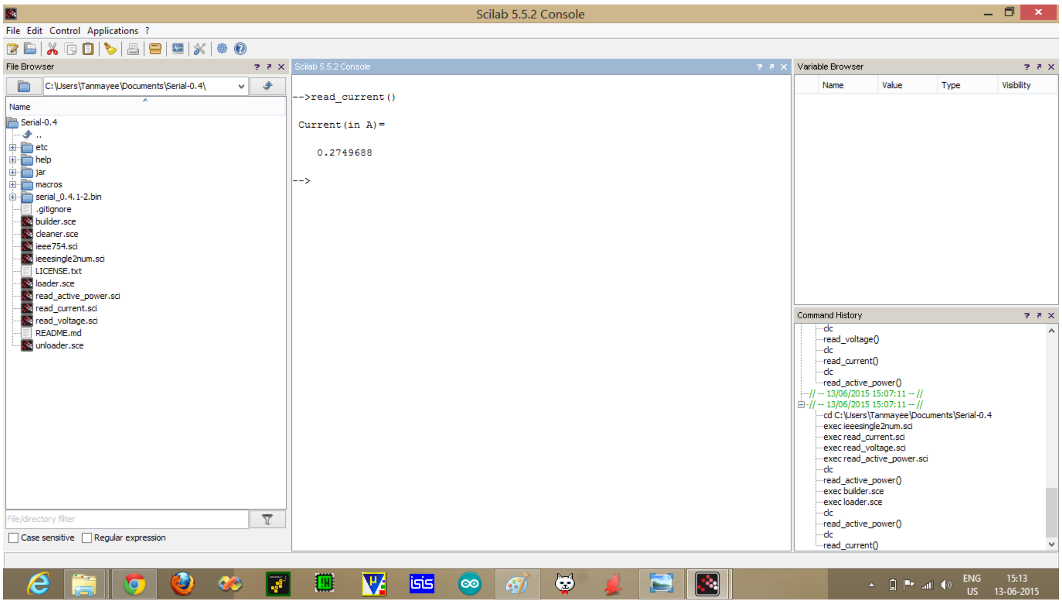


Figure 11.8: Single Phase Current Output on Scilab Console

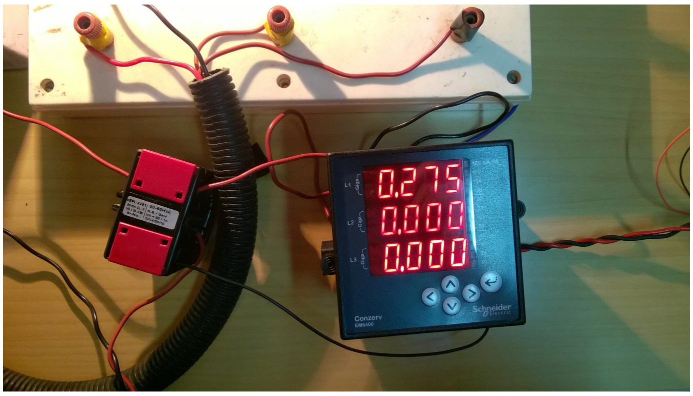


Figure 11.9: Single Phase Current Output on Energy Meter

which will contain the data requested. This data is read serially using (read_serial command) in Scilab and the bytes so received are stored in the 'buf' variable. On analyzing the bytes received (by observing the value of ASCII(buf) or myresult) we see that there might be some spaces(value 32 in myresult) received. 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 starting at n+4 position(i.e., we have to analyse the four bytes starting at (n+4)th position). Note

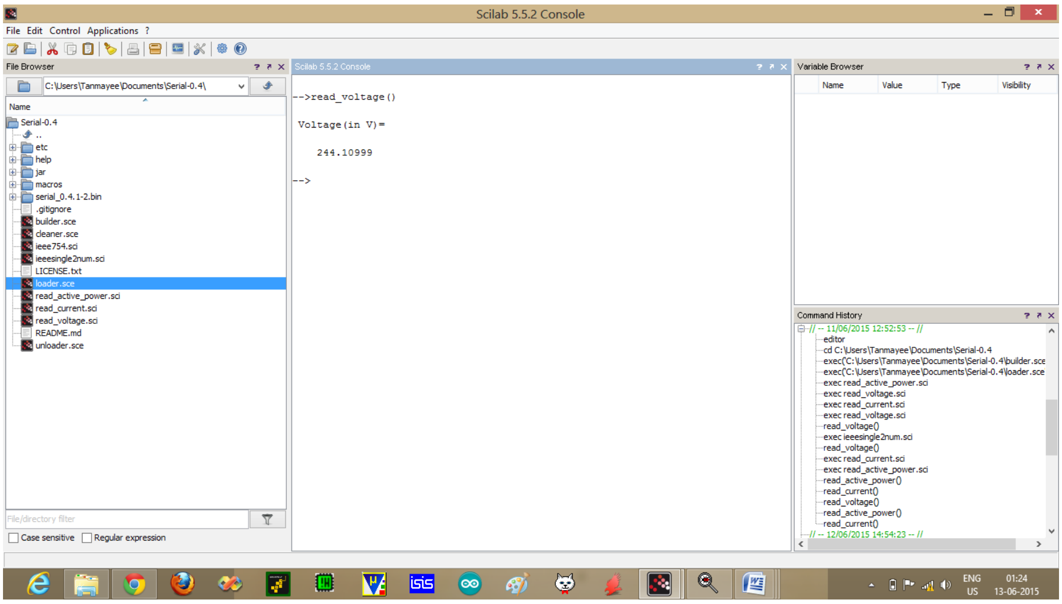


Figure 11.10: Single Phase Voltage Output on Scilab Console

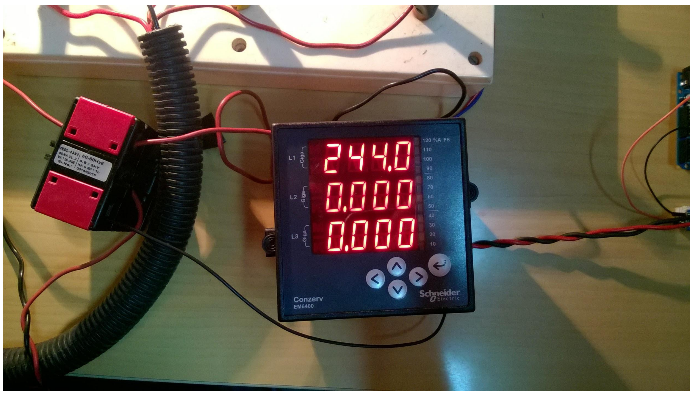


Figure 11.11: Single Phase Voltage Output on Energy Meter

that the packet received may have one or more spaces at the starting or the ending and that is the reason why we may have to shift our indexing for analyzing data.

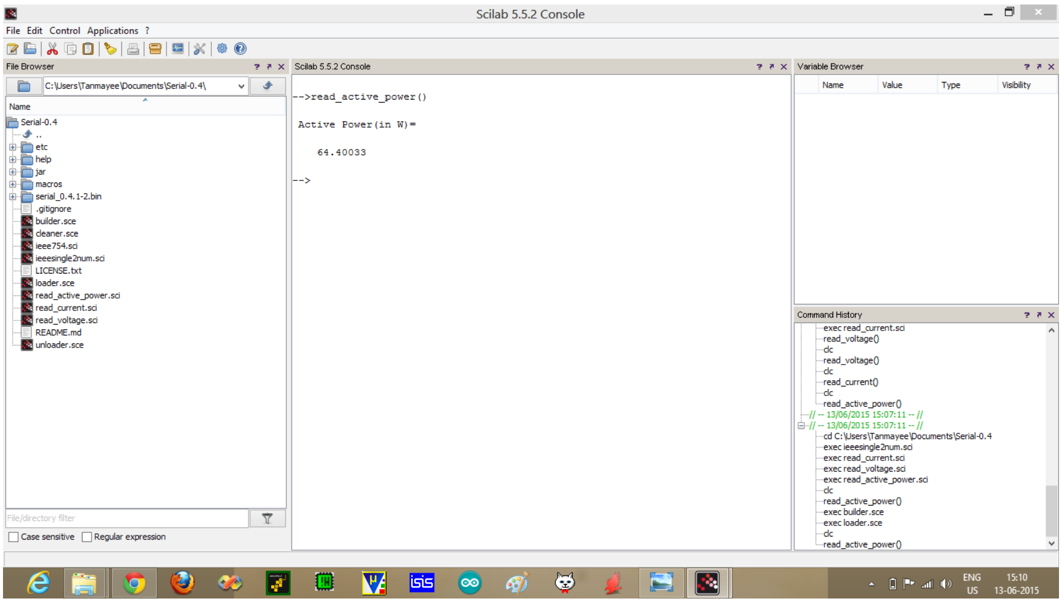


Figure 11.12: Single Phase Voltage Output on Scilab Console

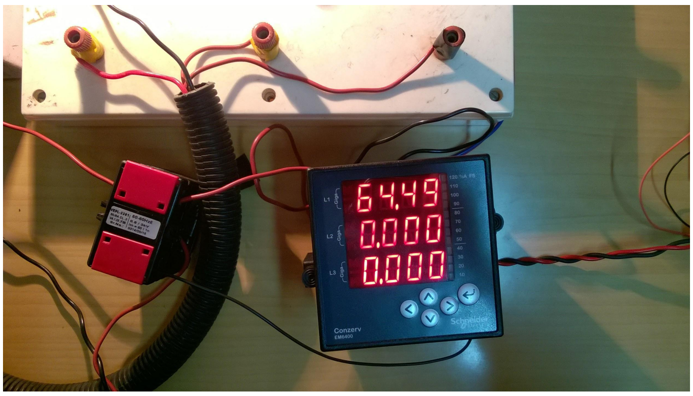


Figure 11.13: Single Phase Voltage Output on Energy Meter

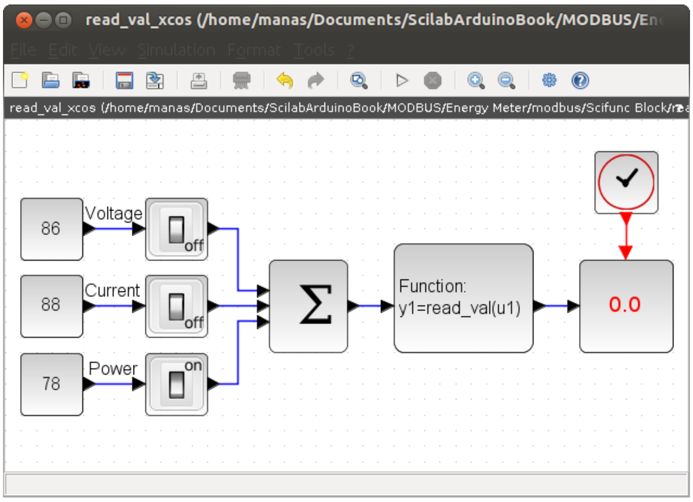


Figure 11.14: Xcos diagram to read Energy Meter values. This is what one sees when `Origin/user-code/modbus/scilab, see Footnote 2 on page 2read_value_xcos.zcos` is invoked.

Table 11.5: 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

11.7 Arduino Code

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.

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

static unsigned char auchCRCHi[] = {0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
```

11.8 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.

```

//voltage
function p=read_val(addr_byte)
h=open_serial(1,2,9600)

//for x=1:5
    //arr=[ascii(01) ascii(03) ascii(15) ascii(86) ascii(00) ascii(02) ascii(39) ascii(15)];
    //arr1=string(arr)
    if(addr_byte==86)
    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.

```
//current
```

```
//function read_current()
function read_current()

h=open_serial(1, 2, 9600)
//for x=1:5
    arr=[ascii(01) ascii(03) ascii(15) ascii(88) ascii(00) ascii(02)
    ascii(70) ascii(204)];

     arr1=ascii(01) + ascii(03) + ascii(15) + ascii(88) + ascii(00) +
    ascii(02) + ascii(70) + ascii(204);
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.

```
//voltage
function read_voltage()
//endfunction
h=open_serial(1,2,9600)

//for x=1:5
    arr=[ascii(01) ascii(03) ascii(15) ascii(86) ascii(00) ascii(02)
    ascii(39) ascii(15)];
    //arr1=string(arr)
    aac=ascii(01)+ ascii(03) + ascii(15) + ascii(86) + ascii(00) +
    ascii(02) + ascii(39) + ascii(15)
    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.

```
//energy
function read_active_power()

h=open_serial(1,2,9600)
//for x=1:5
    arr=[ascii(01) ascii(03) ascii(15) ascii(78) ascii(00) ascii(02)
    ascii(167) ascii(08)];
     asc1=ascii(01) + ascii(03) + ascii(15) + ascii(78) + ascii(00) +
    ascii(02) + ascii(167) + ascii(08);

write_serial(1,asc1,8);
```

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.