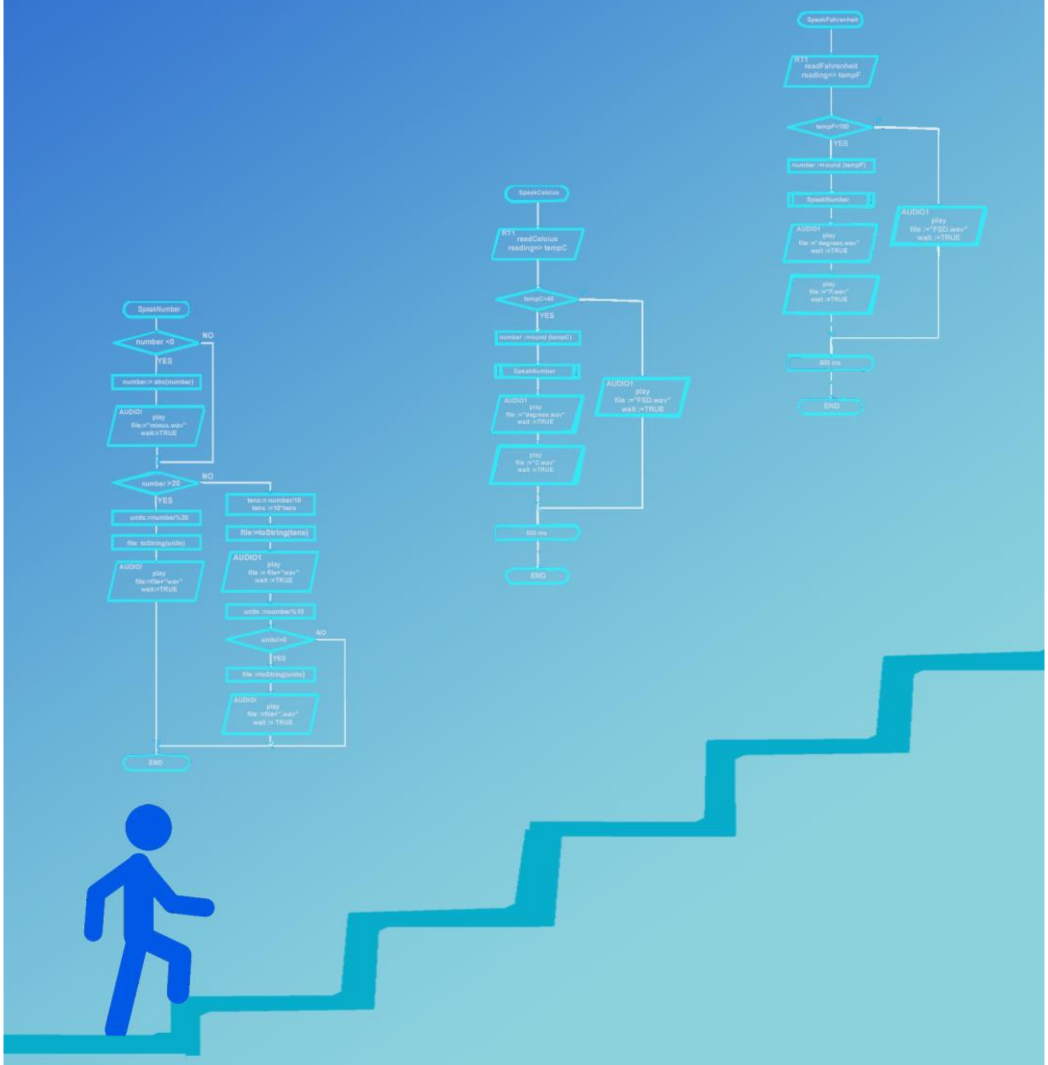
LEARN PROGRAMMING AND ELECTRONICS WITH PROTEUS VISUAL DESIGNER



PROTEUS

labcenter Electronics www.labcenter.com

Revision 1.3

Contents

Preface – “Making Life Easier” 8

1. Introduction 9

1.1. The Technology of Today 9

1.2. Terms and Technologies 10

1.2.1. Flowchart 10

1.2.2. “C” Code 11

1.2.3. Microcontroller 11

1.2.4. Arduino 12

1.2.5. Schematic 13

1.2.6. Proteus 13

1.3. What to Expect 13

2. Creating Your First Project 14

2.1. Pre-Requisites 14

2.2. Creating a New Project 14

2.3. Familiarisation with the Proteus Environment 18

2.3.1. Schematic Capture Tab 18

2.3.2. Visual Designer Tab 19

2.3.3. Flowchart Sheets 19

2.3.4. Peripherals 19

2.3.5. Navigating around a Design 20

2.4. Essentials of Flowchart Programming 21

2.4.1. Types of Flowchart Blocks 21

2.4.2. Flowlines 23

2.4.3. Setup and Loop and Routines 23

2.5. Designing a Flowchart to Blink an LED 23

2.5.1. Drawing the Flowchart 24

2.6. Running the Simulation 30

2.7. Further Information 32

2.7.1. What LED Are We Blinking? 32

2.7.2. “Active Popups” 34

2.7.3. Flowchart Editing Skills 34

2.7.4. Uploading the Program to Physical Hardware (Optional) 35

2.7.5. Additional Resources 37

2.8. Challenges 37
3. Reading Inputs and Making Decisions 38
3.1. Introduction to Decision Blocks 38
3.2. Project Summary 38
3.3. Starting the Project 39
3.4. Adding Peripherals to the Project 39
3.5. Drawing the Flowchart 41
3.6. Running the Simulation 43
3.7. Inverting the Logic 44
3.8. Further Information 45
3.8.1. The “Grove” System 45
3.8.2. Changing the Grove Connector Number in Proteus 46
4. Variables 48
4.1. Displaying Text on an LCD 48
4.2. Adding a Variable to the Project 53
4.3. Types of Variables 56
4.3.1. Possible Operations for Different Variable Types 57
4.3.2. Variable Storage within the Computer 57
4.4. Using Variables in a Project 57
4.4.1. Mathematical Operations using a Variable 58
4.4.2. Displaying the Value of a Variable on an LCD 59
4.4.3. Completing the Flowchart which Displays an Incrementing Variable on the LCD 59
4.5. Using Text in Arguments 60
4.6. Mathematical Operations Using Variables 60
4.7. Using Variables in Conditions 60
4.8. Relational Operators 63
4.9. Logical Operators 63
4.10. Final Project Step 63
4.11. The Reset Button 65
4.12. Sleep on it 65
5. Basic Electric Circuit Theory and Simulation 66
5.1. The Schematic Capture Environment 66
5.2. Simple Electrical Circuit 68
5.3. Ohms Law (Voltage, Current and Resistance) 69

5.4. Electrical Measurements 72
5.5. Further Circuits 72
6. Drawing and Simulating Circuits 74
6.1. For the Curious, an Overview of the Electronic Design Process 74
6.2. Schematic Capture Environment Overview 74
6.3. Picking, Placing and Wiring Up Components 76
6.3.1. Picking Components 76
6.3.2. Placing Components 77
6.3.3. Component “Reference”, “Value” and “Body” 78
6.3.4. Moving, Rotating and Mirroring Components 78
6.3.5. Editing Component References and Values 79
6.3.6. Making Electrical Connections between Components (“Wiring Up”) 81
6.4. Drawing and Simulating Your Own Schematic 84
7. Drawing Custom Peripherals 86
7.1. Project Summary: Sequencing LEDs 87
7.2. Drawing the Circuitry 87
7.2.1. Copy and Paste Facilities 88
7.3. LED Circuits 89
7.4. Analogue and Digital Electronics 89
7.4.1. Analogue-to-Digital Converters 90
7.4.2. Using the Potentiometer to Generate an Analogue Input Voltage 90
7.5. Inputs and Outputs 92
7.6. The Flowchart 93
7.7. Active Popups 96
7.8. Testing the Project 97
7.9. Chapter Summary 97
8. Sub-Routines and Conditional Loops 98
8.1. Setting up the Starter Project 99
8.2. Constructing Colours 99
8.3. Setting a Single Pixel 100
8.4. Setting 3 Pixels 102
8.5. Setting 40 Pixels 103
8.6. “For” Conditional Loops 104
8.7. Types of Conditional Loops 106

8.7.1. “While” Loops 106
8.7.2. “Repeat” Loops 106
8.7.3. “For” Loops 107
8.7.4. “Count” Loops 107
8.8. Adding More Colours 107
8.9. Sub-Routines 108
8.10. Getting Fancy 111
9. Introduction to C 114
9.1. Comparing C and Flowcharts 114
9.2. C Language Syntax Introduction 118
9.2.1. Comments 119
9.2.2. Pre-Processor Directives 120
9.2.3. Indentation 120
9.2.4. Semicolons 120
9.2.5. Variable Declarations 120
9.2.6. Sub-Routines 121
9.2.7. Statements 122
9.3. C Statements 123
9.3.1. “If” Statement 123
9.3.2. “While” Statement 124
9.3.3. “Do” Statement 124
9.3.4. “For” Statement 124
9.3.5. “Switch” Statement 125
9.4. Next Steps for Further Learning 126
9.5. C Operator Reference 128
9.6. C Variable Type Reference (Arduino) 128
9.7. C Pointers Reference 128
Pointer Declaration 128
“Value At” a Pointed Address 128
“Address Of” a Variable 128
9.8. Converting a Flowchart to C/C++ Code 129
9.8.1. PreProcessor Include Directives 131
9.8.2. Class Declarations 131
10. Summary 132

11. Bonus Chapter: Robotics 133
11.1. Line-Following Theory 136
11.2. Constructing the Program: The Plan 138
11.2.1. "drive" Method 138
11.2.2. "readLinePos" Method 139
11.2.3. The Plan 139
11.3. Constructing the Program: Putting "The Plan" Into Action 140
11.3.1. Constructing the Flowchart 140
11.3.2. Specifying the Map 141
11.4. The Moment of Truth 144
11.5. Summary 145
12. Author's Notes 146
12.1. Topics Not Yet Covered 146
12.2. Implementation of Technology 146
Appendix A : Challenges 147
General Procedure for Challenges 147
Chapter 2 (Creating Your First Project) 147
Heartbeat on LED 147
Chapter 3 (Reading Inputs and Making Decisions) 148
Blink on Demand 148
Blink Frequency Change 148
Chapter 4 (Variables) 148
Nightlight 148
Countdown to Blastoff 149
Chapter 6 (Drawing and Simulating Circuits) 150
Ohm's Law 150
Chapter 7 (Drawing Custom Peripherals) 150
Two LEDs, One Pin 150
Chapter 8 (Sub-Routines and Conditional Loops) 151
TwoTone 151
Checkerboard 151
Pulsing Glow 152
Appendix B : Solutions to Challenges 154
Chapter 2 (Creating Your First Project) 154

Heartbeat on LED 154
Chapter 3 (Reading Inputs and Making Decisions) 154
Blink on Demand 154
Blink Frequency Change 155
Chapter 4 (Variables) 155
Nightlight 155
Countdown to Blastoff 156
Chapter 6 (Drawing and Simulating Circuits) 156
Ohm’s Law 156
Chapter 7 (Drawing Custom Peripherals) 157
Two LEDs, One Pin 157
Chapter 8 (Sub-Routines and Conditional Loops) 157
TwoTone 157
Checkerboard 158
Pulsing Glow 158

Preface – “Making Life Easier”

Seemingly since the dawn of time, man has been looking for ways to make his life easier. From a humble spoon to the modern Smartphone, the invention of tools and mechanisms has allowed him to speed-up, ease and automate everyday tasks – reducing labour and improving standard of living. How many times a day do you switch on a light, or use transport (bicycle/car/train), or flush a toilet? The modern age of technology arguably started with The Industrial Revolution of the 1700’s, which brought (amongst other things) mechanised textile (cloth) production: Because people needed to spend less time producing cloth (and obviously therefore clothing), they could spend more time working on other things (including inventing other time-saving machines), and the trend continued. After all the mechanical cog-and-gear innovation of the Industrial Revolution in the 1700's – water wheels and steam engines – these days there is a trend towards machines becoming “smart”. Smartphones, Smart watches, Smart homes, Smart cities. Self-driving cars, machine-learning and artificial intelligence (AI) are no longer science-fiction, but a reality. The aim of this course is to show you how easy it can be to start building your own smart machines! From the starter project of controlling a simple light up to projects using machines-within-machines (“sub-routines”), step-by-step instructions are provided to teach you how to start building your own smart machines – for making your and other people’s lives easier.

1. Introduction

1.1. The Technology of Today

In the world of today, rapidly developing technology is all around us and shapes the way in which we live and work each day. Powering this modern technology is electronics and software. Electronic devices called microprocessors execute the list of commands contained in the software instructions, and in this way the software tells modern electronic machines what to do – together serving as the “brains” of the machines. The writing of software code, or “programming” as it is known, is thus an increasingly relevant and useful skill in the modern age. To look at ways in which software shapes the world around us, just think of common devices used on a daily basis: smartphones, cars (almost all modern cars feature an ECU or “Engine Control Unit”), computers, microwaves, gaming consoles, washing machines, and so on.

A famous technological entrepreneur and businessman once said “Learning to write programs stretches your mind, and helps you to think better; creates a way of thinking about things that I think is helpful in all domains”[1] - meaning that learning how to program teaches us not only how to write code for computers, but perhaps also helps us to develop a logical way of thinking which can help us to find innovative solutions to other day-to-day problems.

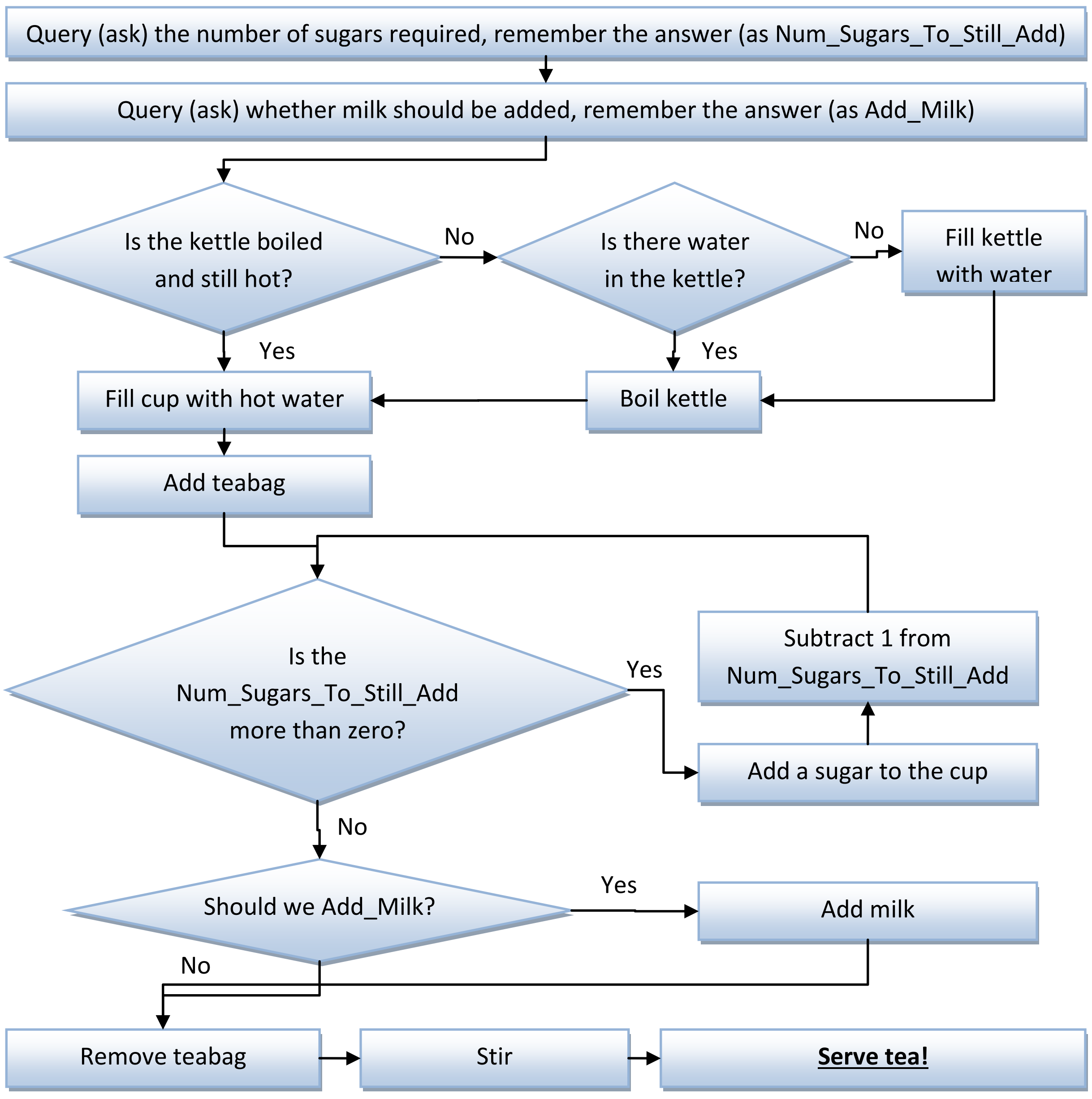
The purpose of this book is to introduce the reader to programming and electronics from the ground up, in an easy step-by-step way. No prior experience with programming or electronics is required. At the same time, the book is also designed in such a way that readers with some prior experience should be able to easily identify and skim through topics which they may already be familiar with. To begin, let's first cover some terms and technologies used throughout the book:

[1]Quote by Bill Gates, founder of Microsoft. https://code.org/quotes.

1.2. Terms and Technologies

1.2.1. Flowchart

A flowchart is a visual, easy-to-understand diagram which illustrates a sequence of decisions and actions required to perform a task. For example, let's look at a program which describes how to make a cup of tea:


Query (ask) the number of sugars required, remember the answer (as Num_Sugars_To_Still_Add)
Query (ask) whether milk should be added, remember the answer (as Add_Milk)
Is the kettle boiled and still hot?
No
Is there water in the kettle?
No
Fill kettle with water
Yes
Yes
Fill cup with hot water
Boil kettle
Add teabag
Is the Num_Sugars_To_Still_Add more than zero?
Yes
Subtract 1 from Num_Sugars_To_Still_Add
Add a sugar to the cup
No
Should we Add_Milk?
Yes
Add milk
No
Remove teabag
Stir
Serve tea!

Notice the procedural ("step-by-step") structure of the flowchart, as well as the binary ("yes or no") decision making process – this is the way in which a computer "thinks".

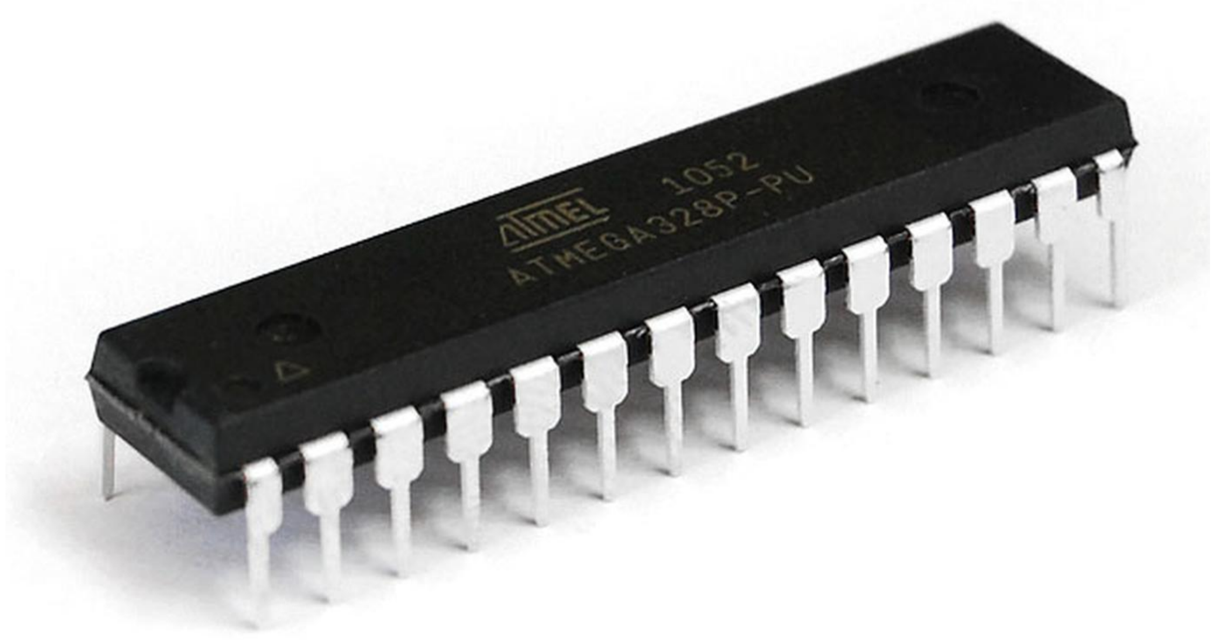
1.2.2. “C” Code

“C” is an industry-standard “high-level” programming language, used for writing software programs. “Industry standard” means that the technology is very popular and common in the software industry. “High-level” means that the code is more like English and less like raw computer-code. A tool called a “compiler” converts the high-level language into low-level computer-code. For beginners, C can seem confusing and difficult at first. This book focuses first on flowchart programming, which is easier to understand and less prone to errors such as “typing mistakes”, and then introduces C towards the end of the book. Note that even professional C programmers will still use flowcharts from time to time, even if just on paper; to plan out a program, or to provide a summary of how a program works for a report.

An even higher-level version of C, called “C++”, adds some more complex functionality to the C “language”.

Note that all of the flowcharts in this book can be converted into C/C++ code with a few clicks (the exact procedure for doing this will be explained towards the end of the book, in section 9.8).

1.2.3. Microcontroller



A microcontroller is an electronic chip which can process instructions (run software), and also has input/output connections (“pins” or “legs”) which enable it to interact with the “outside world”. Microcontrollers also generally include some “extra” features (called “peripherals”), such as timers, which can help them to accomplish more than they could by just processing code only. You may have heard of the term “CPU” or “microprocessor”, which is the “brain” inside a computer that processes instructions; a micro<u>controller</u> is a micro<u>processor</u>, but with extra features (peripherals) built-in.

1.2.4. Arduino

Arduino is an extremely popular and open-source series of low-cost electronic circuit boards ("open-source" means that the design "source" files are freely available to the public). The boards are designed to make it quick and easy to build electronic machines – by providing a ready-made platform on which to build – and feature a microcontroller as well as pins for inputs and outputs.

The term "Arduino" technically also refers to the Arduino "IDE[2]" and compiler; these are computer programs in which software code for the Arduino boards can be written (in C/C++). Since we will be programming Arduino boards using flowcharts first, we will not worry about the Arduino IDE / compiler or C/C++ code for now (these topics are introduced towards the end of the book, in chapter 9).

[2] IDE stands for "Integrated Development Environment", and is largely a document editor with some special features which assist in writing computer code – such as highlighting elements of the code in different colours.

1.2.5. Schematic
A schematic is an electronic wiring diagram – it is meant to show what is electrically connected (or "wired up") to what, in the easiest and clearest way possible. Whilst the actual physical "wires" themselves may follow complex routes in the "real world", the schematic simply shows what is connected to what (without worrying about the complexities of exactly how this is implemented in a "real-world" product).

1.2.6. Proteus
Proteus is a popular electronics design and simulation software. With Proteus it is possible to draw electronic circuits and simulate them on the computer, including microcontrollers running software code! Being able to simulate electronics on the computer has a number of advantages:

- It is much quicker and easier to draw circuits on the computer than it is to build them in real-life. There is also much less chance of making a mistake, such as having a loose connection (which can take hours to find in the real-world).
- Circuits can be built using components which you don't necessarily already have physically available. This saves both time and money, since you don't have to buy the components first or wait for them to arrive.
- There is no chance of breaking anything or blowing anything up. There is a joke which says that all electronics works on smoke inside; the reasoning for this is that if the smoke escapes, then the electronics don't work anymore. With simulation, the smoke can never escape.
- Instruments which are expensive to buy in the real-world, such as oscilloscopes, are freely available inside the simulation environment.
- It is possible to pause the simulation and take measurements (a term known as "debugging"), which is not always possible in real-life.

Proteus also features a module called "Visual Designer for Arduino", which is a flowchart designer for Arduino, and which we will be using in this book.

Note that this entire book can be completed using only Proteus, without the need for any physical hardware boards – however if hardware boards are available, then they can be easily programmed from within Proteus (this is described in section 2.7.4).

1.3. What to Expect
This book starts with the very basics, with step-by-step instructions; beginning with simple projects and then moving onto progressively more advanced topics. The bulk of this book focuses on flowchart programming, with an introduction to C at the end. As previously mentioned, all of the flowcharts can be converted to C/C++ code with a few clicks (the exact method for doing so is explained towards the end of this book, in section 9.8). All of the projects in this book can be completed and simulated entirely within Proteus without the need to purchase any physical hardware, however they can also be downloaded to physical hardware (if available) very easily if desired (section 2.7.4).

Without further ado, let's get started!

2. Creating Your First Project

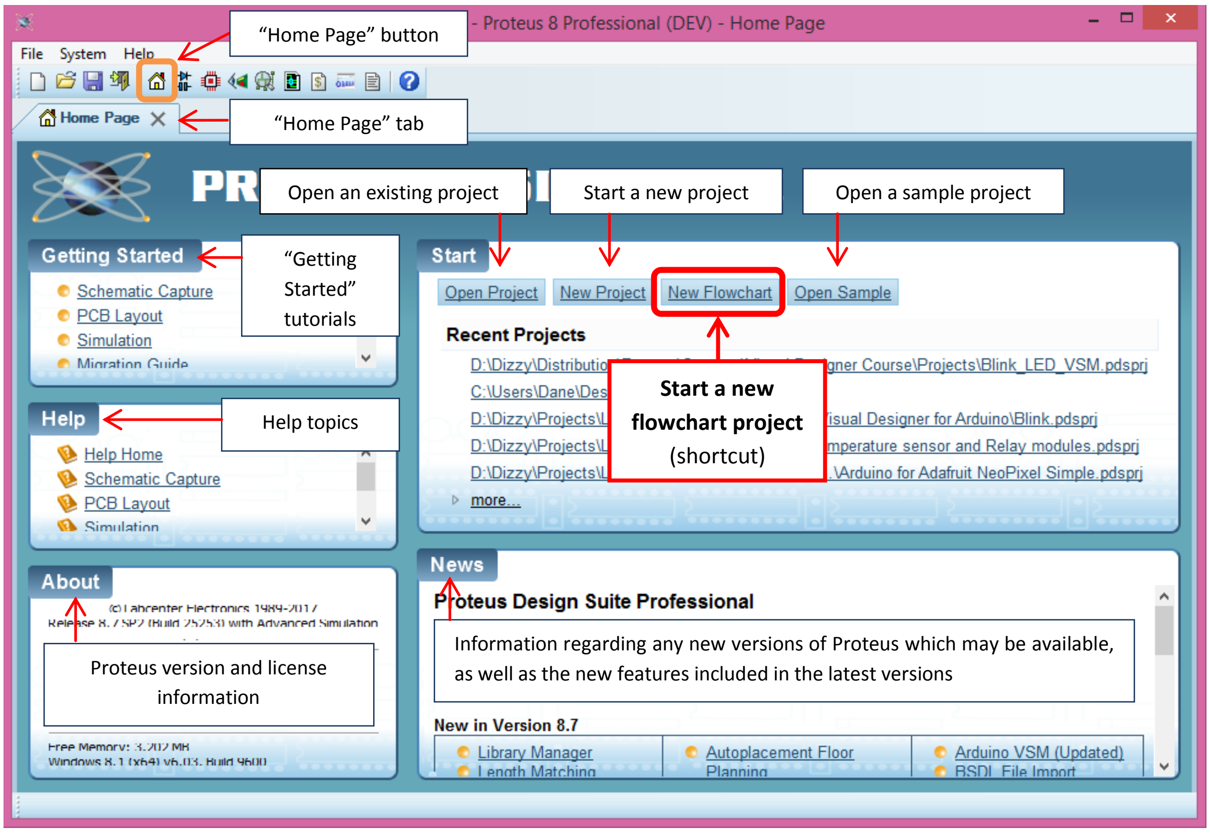
To get started, we are going to jump right in and see how easy it is to create our first working project!

2.1. Pre-Requisites

- This chapter assumes that you already have some basic computer knowledge and experience (that you know how to use a computer).
- Please ensure that Proteus is installed on your computer before proceeding.

2.2. Creating a New Project

Step 1: To start with, please ensure that Proteus is running (this can normally be done by double-clicking on the Proteus icon on your Windows desktop). When Proteus starts, the Proteus Home Page is shown, as illustrated in the following figure.



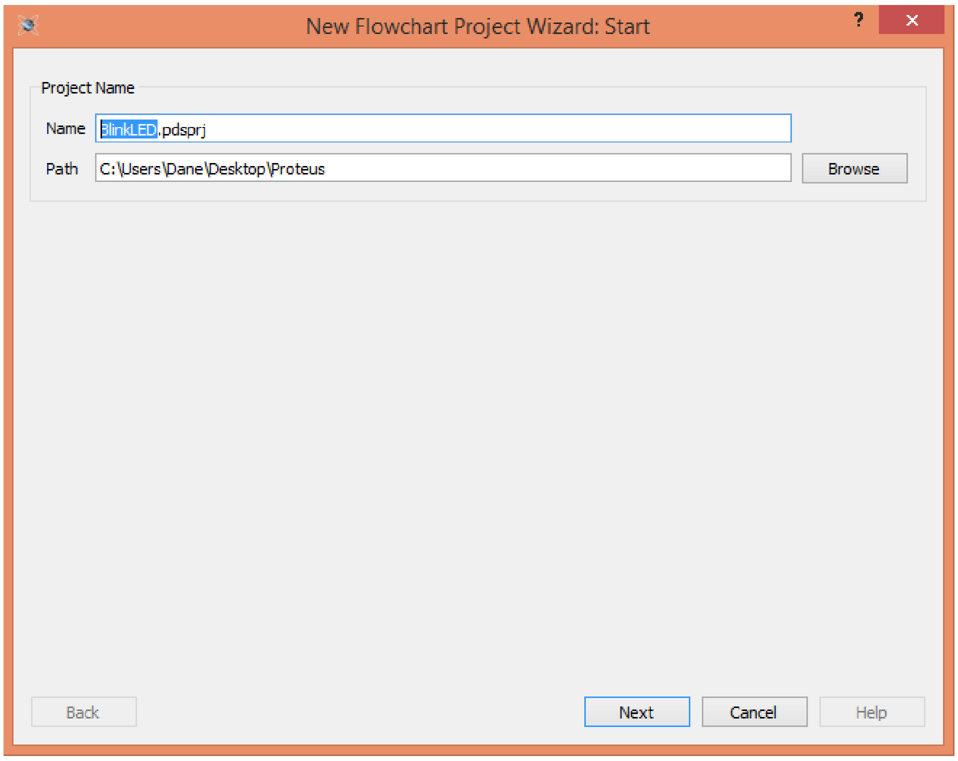
Proteus 8 Professional (DEV) - Home Page
"Home Page" button
"Home Page" tab
Open an existing project
Start a new project
Open a sample project
"Getting Started" tutorials
Help topics
Start a new flowchart project (shortcut)
Proteus version and license information
Information regarding any new versions of Proteus which may be available, as well as the new features included in the latest versions
Getting Started
Schematic Capture
PCB Layout
Simulation
Migration Guide
Help
Help Home
Start
Open Project
New Project
New Flowchart
Open Sample
Recent Projects
more...
About
News
Proteus Design Suite Professional
New in Version 8.7
Library Manager
Autoplacement Floor Planning
Arduino VSM (Updated)

Figure 2-1: Proteus "Home Page"

Step 2: In order to start a new flowchart project, the **"New Flowchart"** option should be selected in the **Start section** of the Proteus desktop screen (see Figure 2-1). This will then launch the New Project Wizard for starting a new flowchart project (Figure 2-2).

(Tip: The "New Flowchart" option is similar to the more advanced "New Project" option, but streamlines the process by leaving out (or using default values) for steps which are not usually used in a flowchart project. Proteus has other capabilities besides for flowchart design, but we will not be looking at these just yet.)

In the New Project Wizard, the first step is to give the project a name, and specify where (in which folder) it should be created.



Name: BlinkLED.pdsprj
Path: C:\Users\Dane\Desktop\Proteus

Figure 2-2: New Project Wizard for starting a new flowchart project

By default the project is named "New Project" (note that the file extension (.pdsprj) should not be modified); however for our new project please use the name "**BlinkLED**" ("BlinkLED.pdsprj").

The path location can be chosen by using the "Browse" button to navigate to the folder in which you would like to save the project files. Please choose any folder which makes sense to you, such as a new folder called "Proteus" on your Desktop or in your Documents folder.

Once you have set the project name and path, please click the **"Next"** button at the bottom of the New Flowchart Project Wizard in order to proceed further through the New Flowchart Project Wizard process.

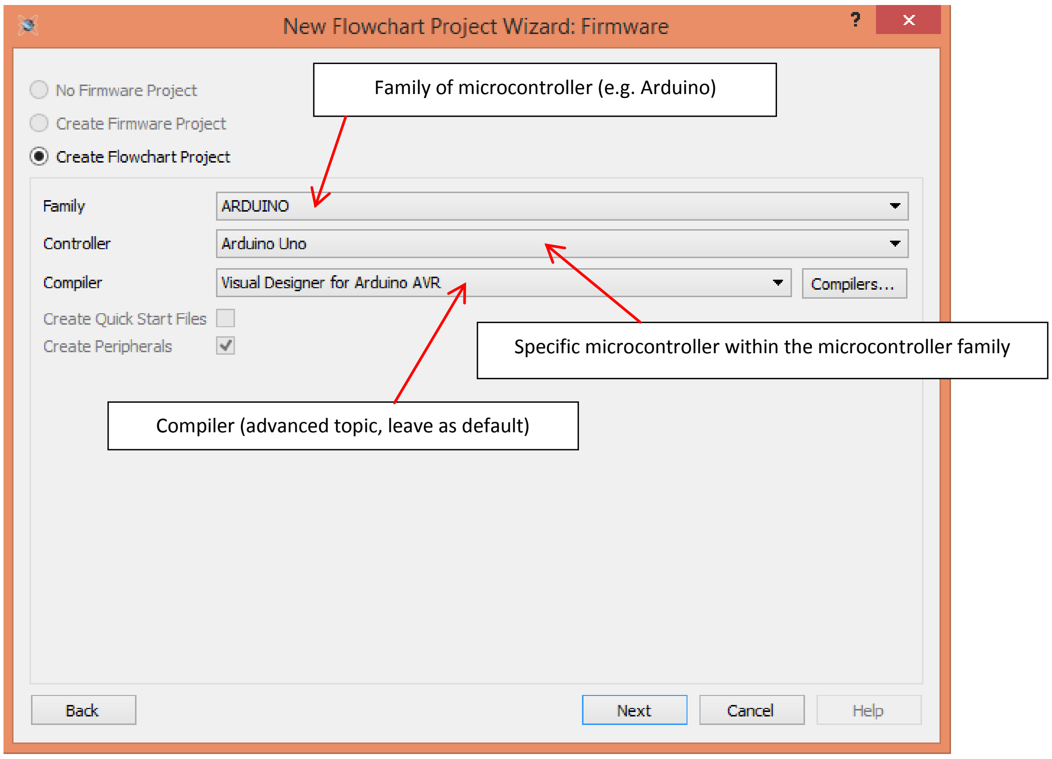
(Tip: The "**Back"** button obviously allows you to go back to the previous step in the New Project Wizard process, and the **"Cancel"** button obviously allows you to exit the New Project Wizard.)

Step 3: The next step of the New Flowchart Project Wizard process is the Firmware setup step (see Figure 2-3). "Firmware" is the name for a microcontroller[3] software program – it is the instructions (which we will be developing) which tell the microcontroller what to do. This step in the wizard is used to specify which microcontroller we would like to develop firmware for.

(Tip: The term "Software" is typically used for computer code, whilst the term "Firmware" is typically used for microcontroller code. The short explanation of the reason for this is as follows: It is typically more difficult to change or replace the code in a microcontrollers' memory, than it is to change the

[3] If you need a reminder on what a microcontroller is, please see section 1.2.3.

code in a computers memory. Thus, the microcontroller code could be considered to be more "firm" (less changeable). For our purposes however, we can consider the terms Firmware and Software to mean about the same thing.)


New Flowchart Project Wizard: Firmware

- No Firmware Project
- Create Firmware Project
- (●) Create Flowchart Project

Family: ARDUINO
Controller: Arduino Uno
Compiler: Visual Designer for Arduino AVR
Create Quick Start Files: ☐
Create Peripherals: ✓

Family of microcontroller (e.g. Arduino)
Specific microcontroller within the microcontroller family
Compiler (advanced topic, leave as default)

Figure 2-3: Firmware selection options for New Project

We can safely leave all of the values as per the defaults.

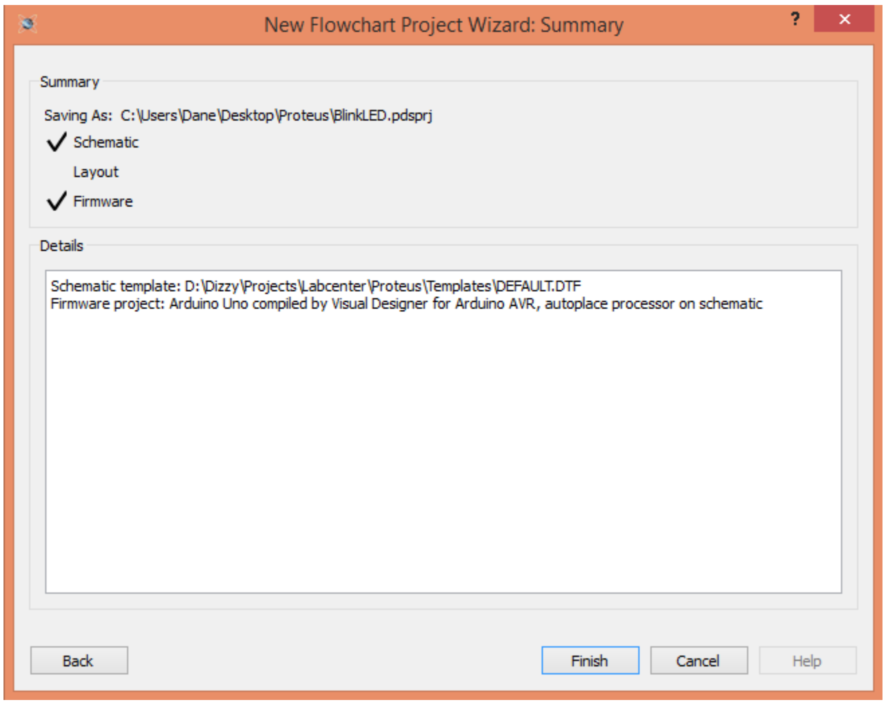
"Family" refers to a brand of microcontroller, similar to how there are different brands or manufacturers of cars (e.g. BMW / Audi / Mercedes Benz / Toyota / etc). Arduino is technically not a microcontroller brand but rather a selection of AVR brand microcontrollers; the name Arduino however serves as a useful group identifier nonetheless.

"Controller" refers to the actual microcontroller product (using the analogy of cars again, this would be something like BMW 328i or Audi A4 3.0 TDI). Once again Arduino Uno is not technically a microcontroller, but rather a product which uses an ATmega328P microcontroller; but it once again serves as a useful and easy identifier nonetheless.

"Compiler" refers to a tool that takes program instructions which a human can understand (diagrams or words, such as a flowchart or C code), and converts them into something which a microprocessor can understand (instructions which are represented purely by numbers). A compiler could thus also be thought of as a "translator", translating human-readable instructions into microprocessor-readable instructions.

When ready, click the "Next" button.

Step 4: The last step of the New Project Wizard displays a summary of the details for the new project to be created (Figure 2-4).

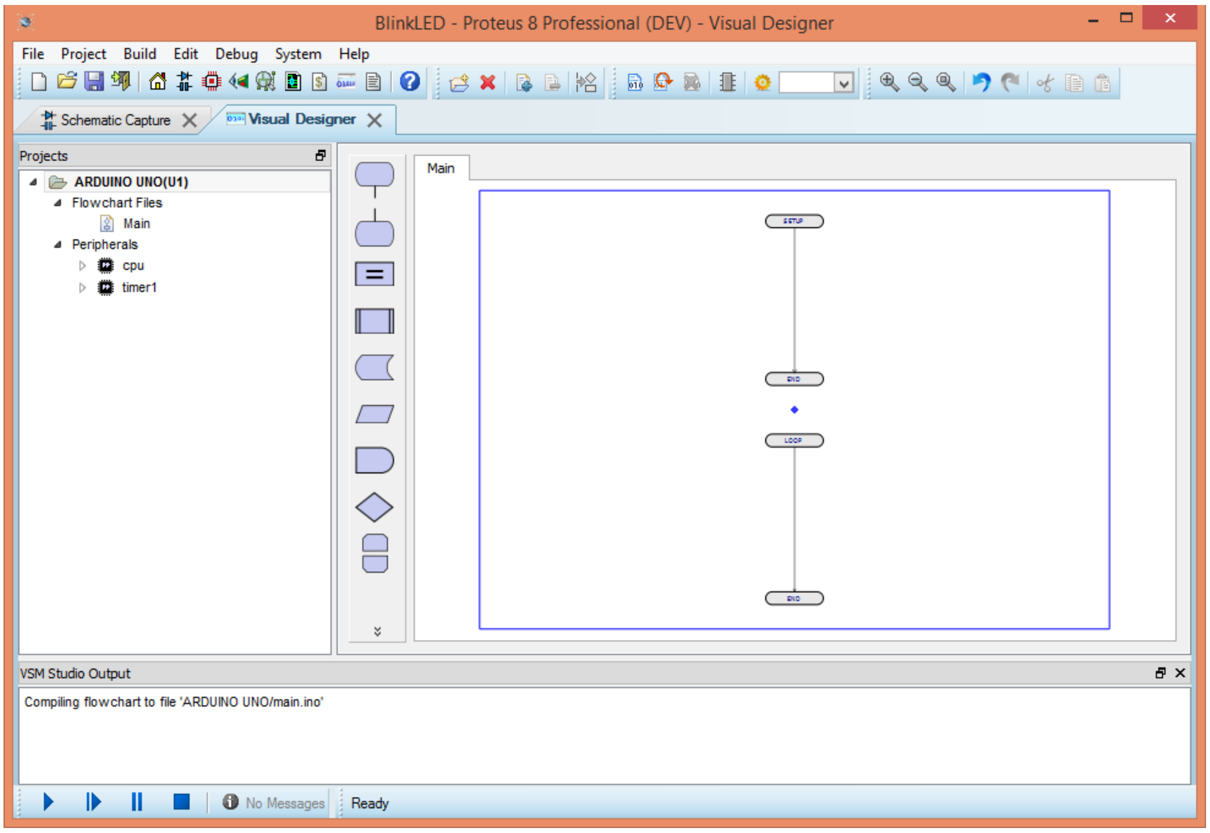


Saving As: C:\Users\Dane\Desktop\Proteus\BlinkLED.pdsprj
✓ Schematic
Layout
✓ Firmware

Schematic template: D:\Dizzy\Projects\Labcenter\Proteus\Templates\DEFAULT.DTF
Firmware project: Arduino Uno compiled by Visual Designer for Arduino AVR, autoplace processor on schematic

Figure 2-4: Summary of details for new project

Click the **"Finish"** button at the bottom of the screen to finalise the wizard process. After clicking the Finish button, Proteus will automatically open the newly created project (Figure 2-5). Once you have reached this stage, you have successfully created a Proteus project and can commence flowchart design.



```
Compiling flowchart to file 'ARDUINO UNO/main.ino'
```

Figure 2-5: New flowchart project

2.3. Familiarisation with the Proteus Environment

After creating your new project using the Wizard process, Proteus will have two tabs open – the Visual Designer tab and the Schematic Capture tab – with the Visual Designer tab selected by default.

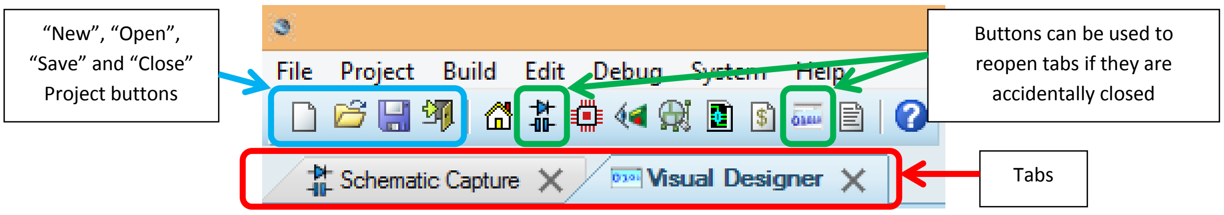

"New", "Open", "Save" and "Close" Project buttons
File
Project
Build
Edit
Debug
System
Help
Buttons can be used to reopen tabs if they are accidentally closed
Schematic Capture
Visual Designer
Tabs

Figure 2-6: Schematic Capture and Visual Designer tabs

2.3.1. Schematic Capture Tab

The Schematic Capture tab contains the electronic circuitry for the project. **We can safely ignore this tab for now**, as the Wizard process automatically places the necessary circuitry for our project onto the schematic. If you're curious, then you can take a quick peek to see what it looks like.

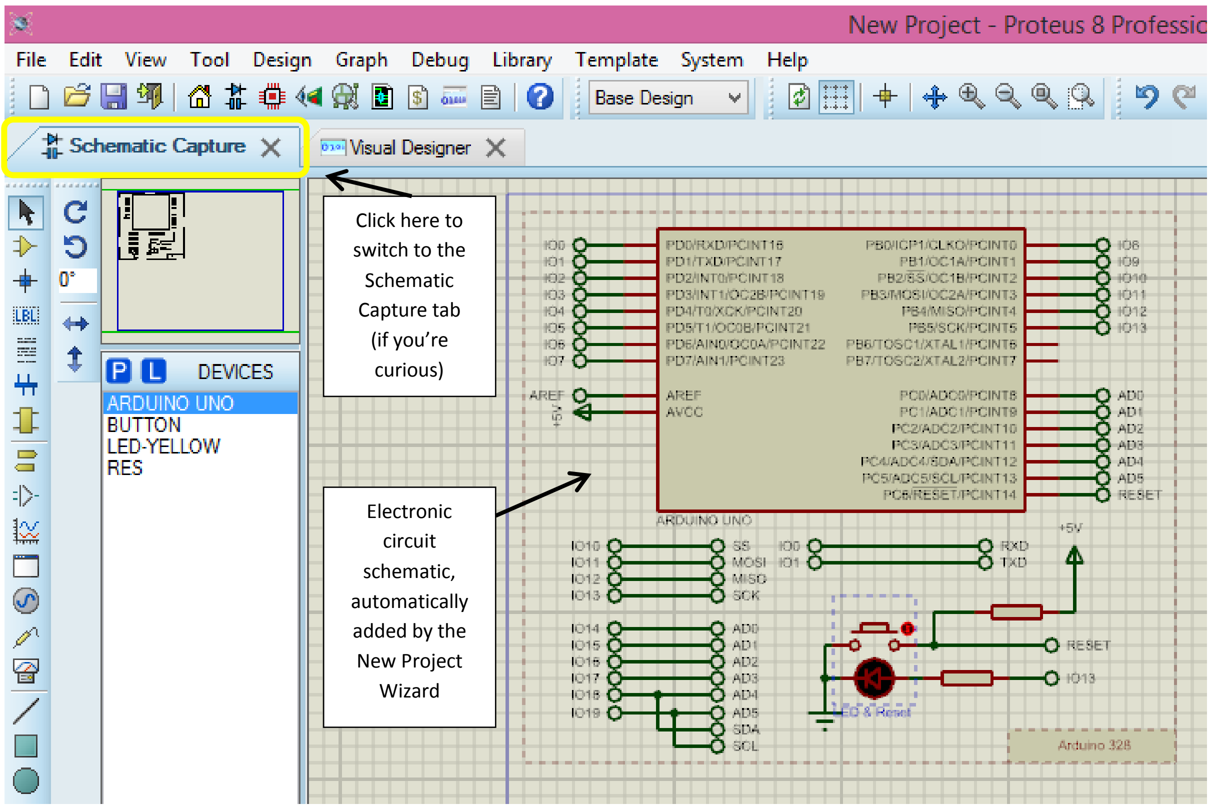
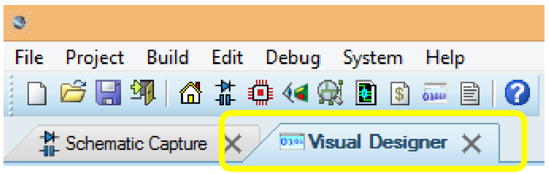

New Project - Proteus 8 Professio
File
Edit
View
Tool
Design
Graph
Debug
Library
Template
System
Help
Base Design
Schematic Capture
Visual Designer
Click here to switch to the Schematic Capture tab (if you're curious)
Electronic circuit schematic, automatically added by the New Project Wizard
DEVICES
ARDUINO UNO
BUTTON
LED-YELLOW
RES
ARDUINO UNO
Arduino 328

Figure 2-7: Schematic Capture tab

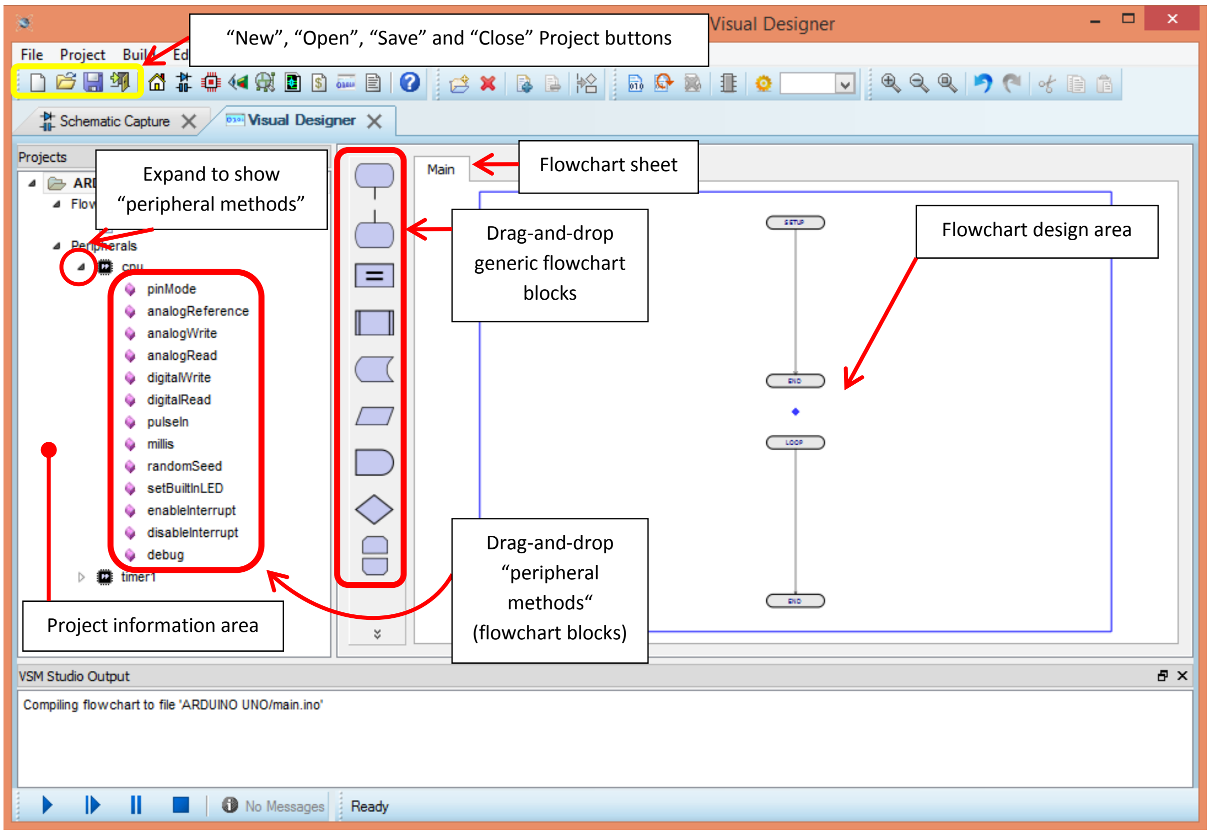
(Tip: For a reminder of what a schematic is, please see section 1.2.5 in the Introduction.)

Once done, switch back to the Visual Designer tab.



2.3.2. Visual Designer Tab

The Visual Designer tab is where flowchart programs are designed.



Compiling flowchart to file 'ARDUINO UNO/main.ino'

Figure 2-8: Visual Designer tab

When a new flowchart project is created then the flowchart area contains by default a "Setup" and a "Loop" routine (these will be explained in greater detail shortly). The small "Projects" area on the left lists all of the flowchart sheets in the project, as well as the "peripherals" used in the project. More information on these concepts is presented next.

2.3.3. Flowchart Sheets

A flowchart sheet is a "page" on which you can design a flowchart. For many projects only one flowchart sheet is needed, but for larger projects more flowchart sheets can be added in order to provide more space for designing, or to better organise the project.

2.3.4. Peripherals

In short, in Visual Designer, peripherals are largely objects which we can interact with – for example a button or a display screen. Peripherals have "methods" (instructions), such as "clear the screen" or "display this text on the screen". The longer, and more technically correct, explanation is as follows:

As mentioned in the introduction (section 1.2.3), a micro<u>controller</u> is a micro<u>processor</u> with some extra features ("peripherals") built-in. Whilst a micro<u>processor</u> can only process instructions, *peripherals* provide additional functionality – such as a timer peripheral for (obviously) timing things. The peripherals work independently from the microprocessor, and as such don't slow it down from processing instructions. This is like having a GPS in your car – you can focus on driving, whilst the

GPS "peripheral" handles the navigation. Besides for the peripherals which are built into the microcontroller, additional *external* peripherals can also be added.

Proteus Visual Designer includes a **peripheral gallery** which contains dozens of devices, such as temperature sensors and displays. Peripherals contain **methods**, which are flowchart blocks applicable to that specific peripheral – such as "on" and "off" commands for a light.

The default peripherals included in a new Arduino project are the "cpu" and "timer1" peripherals:

- The cpu "peripheral" is not a peripheral in the strict sense, because it relates mostly to the microprocessor (CPU) itself. It is however listed as a peripheral, as a convenient way to access various methods applicable to the CPU.
- The timer peripheral[4] can be used, as its name suggests, to time things.

For our first project we won't be using any special additional peripherals (these will be covered in later sections).

2.3.5. Navigating around a Design

Navigating a design refers to zooming in/out of it, as well as panning around it (moving the view left/right/up/down). The same tools are used to navigate both flowchart designs as well as schematic designs.

2.3.5.1. Zooming

The most common method used for zooming is:

- Point the mouse where you want to zoom in / out of, and roll the middle mouse button (roll forwards to zoom in and backwards to zoom out).

Depending on your preference however, additional options include:

- Point the mouse where you want to zoom in or out of and press the F6 or F7 keys respectively.
- Hold the SHIFT key down and drag out a box with the left mouse button around the area you want to zoom in to.
- Use the Zoom In, Zoom Out, Zoom All or Zoom Area icons on the toolbar.

Note that the F8 key can be used as a shortcut at any time to display the whole design ("zoom extents").

2.3.5.2. Panning

Panning refers to navigating left/right/up/down. Usually, if using the mouse-wheel for zooming, then this can be used for panning as well (by zooming out and then zooming in again to the specific area which you would like to view). Panning can also be performed explicitly however. The most common method used for panning is:

- Click on the middle mouse button to "pick up" the design. Move the mouse to move the design, and then click the middle mouse button again to "put down" the design.

[4] Usage of the timer peripheral is an advanced topic and is currently beyond the scope of this book. Suggestions for further learning on this subject are made in section 12.1.

Other options include the below, although these generally only work within the schematic capture tab:

- To simply “pan” the Editing Window up, down, left or right; position the mouse pointer over the desired part of the design and press the F5 key.
- Hold the SHIFT key down and bump the mouse against the edges of the design area to pan up, down, left or right.
- Use the Pan Icon on the toolbar.

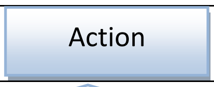
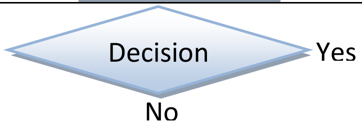
Take some time now to practice zooming and panning around the flowchart sheet (and possibly the schematic sheet as well, if you feel so inclined).

2.4. Essentials of Flowchart Programming

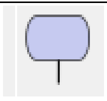
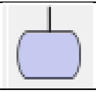
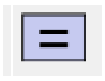
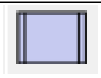
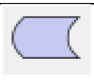
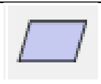
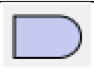
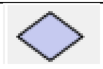
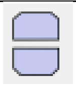
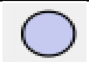
A flowchart consists of a sequence of actions and decisions required to accomplish a task, drawn in a graphical way. For an example, see the flowchart from section 1.2.1 in the Introduction again. A flowchart is generally easier to understand than written (text) code (e.g. C code). By using flowcharts for programming we can focus on getting results, without first having to tackle the complexity of learning for example C or C++ code.

2.4.1. Types of Flowchart Blocks

Flowcharts are constructed from “blocks”. Traditionally, flowcharts are built out of only action and decision blocks:

Block	Name	Description
	Action Block	Represents an action to be taken, for example switch on a light.
	Decision Block	Represents a decision which should be taken, with different possible outcomes.

Proteus however has a richer set of flowchart blocks to choose from, which offer some additional features; as listed in the following table. **Don’t worry about understanding or remembering all of the blocks right now** – they will be covered in greater detail in the following sections, and are listed here only for the sake of completeness.

Functional block	Name	Description
	Event Block	The event block, together with the end block, is used to create a sub-routine (covered in Chapter 8). The event block can also be used to handle events such as interrupts, however this is currently beyond the scope of this book[5].
	End Block	The end block is used to signify the end of a sub-routine or event.
	Assignment Block	The assignment block is used to assign a value to a variable in the program. Variables are used whenever something needs to be remembered, and will be described in greater detail later. In the “Make Tea” example from the introduction (section 1.2.1), “Num_Sugars_To_Still_Add” and “Add_Milk” are variables.
	Sub-routine Call Block	A “Sub-Routine” in Visual Designer is a group of flowchart blocks which are placed separately to the main flowchart. Grouping certain tasks into sub-routines can help to make the flowchart more organised and easier to understand, and can also prevent duplication of “code”. Sub-routines are covered in Chapter 8. A sub-routine is executed (or “run”) by using a Sub-routine Call Block.
	Stored data block	The stored data block is used to perform actions (i.e. reading, writing) on storage objects (e.g. SD cards).
	I/O[6] (Peripheral) Operation Block	This block is used to represent a peripheral method (see section 2.3.4), such as switching a light on or off or writing some text to a display.
	Time Delay Block	This block creates a time delay in the program. For example, “Wait 5 seconds before moving onto the next task”.
	Decision Block	Represents a decision which should be taken, with different possible outcomes, such as “Is the button pressed?” (yes or no).
	Loop Construct Blocks	Loops can also be created using decision blocks, but the Loop Construct Blocks make it easier to, for example, repeat a task a certain number of times.
	Interconnector	If a flowchart starts to become too large, then interconnectors can be used to break the flowchart up into separate sheets or sections (joined by the interconnectors).
ABC	Comment Text	This “block” can be used to add some descriptive text or notes to your flowchart, but does not affect the operation of the program in any way.

[5] Suggestions for further learning on this subject are made in section 12.1.
[6] “I/O” stands for “Input / Output”. Reading the temperature from a thermometer would be an example of an input, and playing a song via an audio speaker would be an example of an output.

2.4.2. Flowlines

Flowchart blocks are linked, or joined, by lines called “Flowlines” (see Figure 2-9).

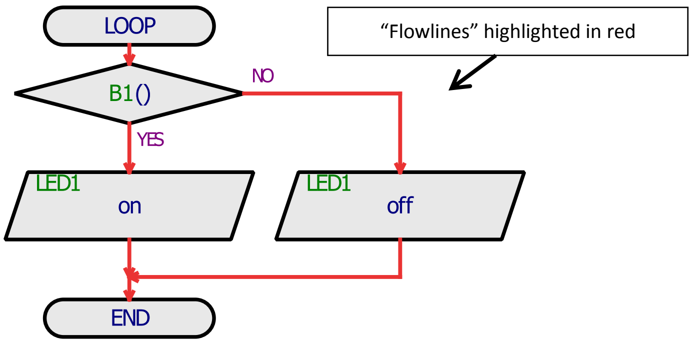


Figure 2-9: Flowlines join flowchart blocks

2.4.3. Setup and Loop and Routines

When a new Proteus Visual Designer project is created, there are by default two routines included on the flowchart: “SETUP” and “LOOP”.

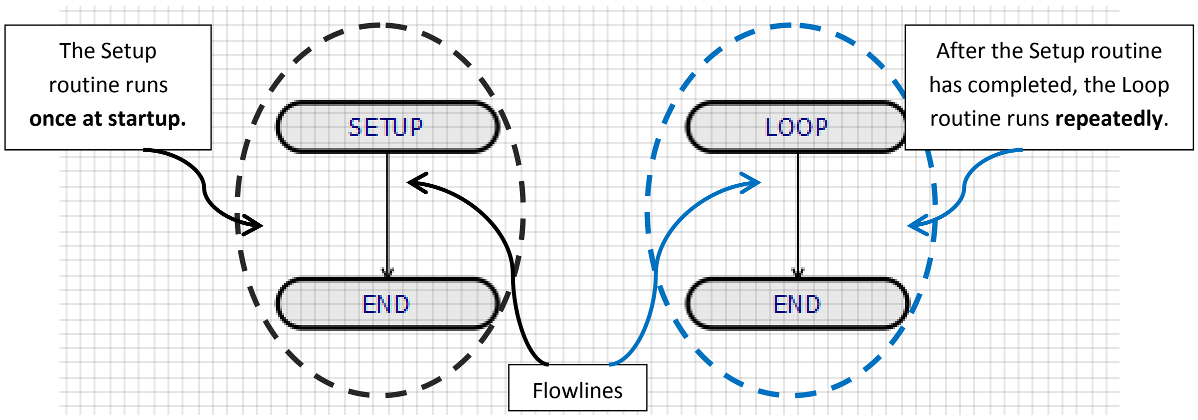

The Setup routine runs once at startup.
SETUP
END
Flowlines
LOOP
END
After the Setup routine has completed, the Loop routine runs repeatedly.

Figure 2-10: Default flowchart program with setup and loop routines

The **Setup** routine is executed (or “run”) **once** at initial start-up.

After the Setup routine, The main **Loop** routine is executed (or “run”) **repeatedly**. Most of the program is normally constructed within the Loop routine.

2.5. Designing a Flowchart to Blink an LED

We will now proceed to design a flowchart which turns an “LED” on and off. LED stands for “Light Emitting Diode”, and is essentially a small light. LEDs are extremely common, and you will have encountered them many times in everyday life (for example, many electronics device have a small LED light which indicates whether they are powered on or not).

For this project we will toggle (switch) the LED on and off, with a 1 second delay in-between. This is a popular task when starting any new project; it is a program which can be relatively quickly developed, and serves to prove that the system (microcontroller and electronics) is “working”, which is normally done as a test before moving onto developing more complex programs.

Details regarding exactly what LED we will be blinking are presented in section 2.7.1, but for now we will jump straight into designing the flowchart program (please feel free to read section 2.7.1 before continuing, if you are curious).

2.5.1. Drawing the Flowchart

The flowchart which we will be building looks like the following figure:

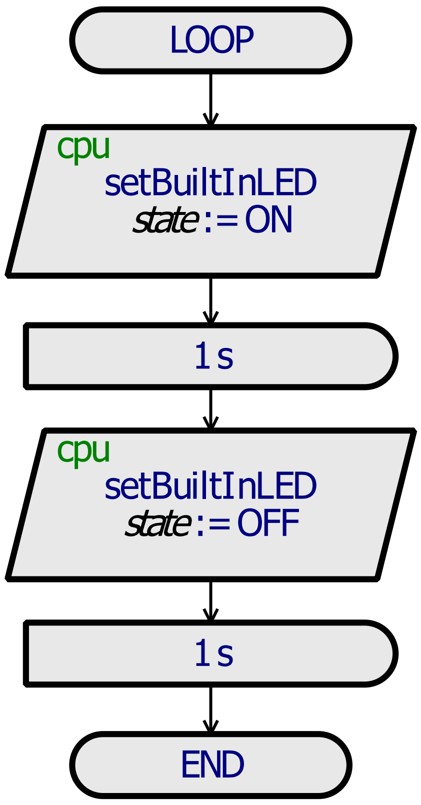
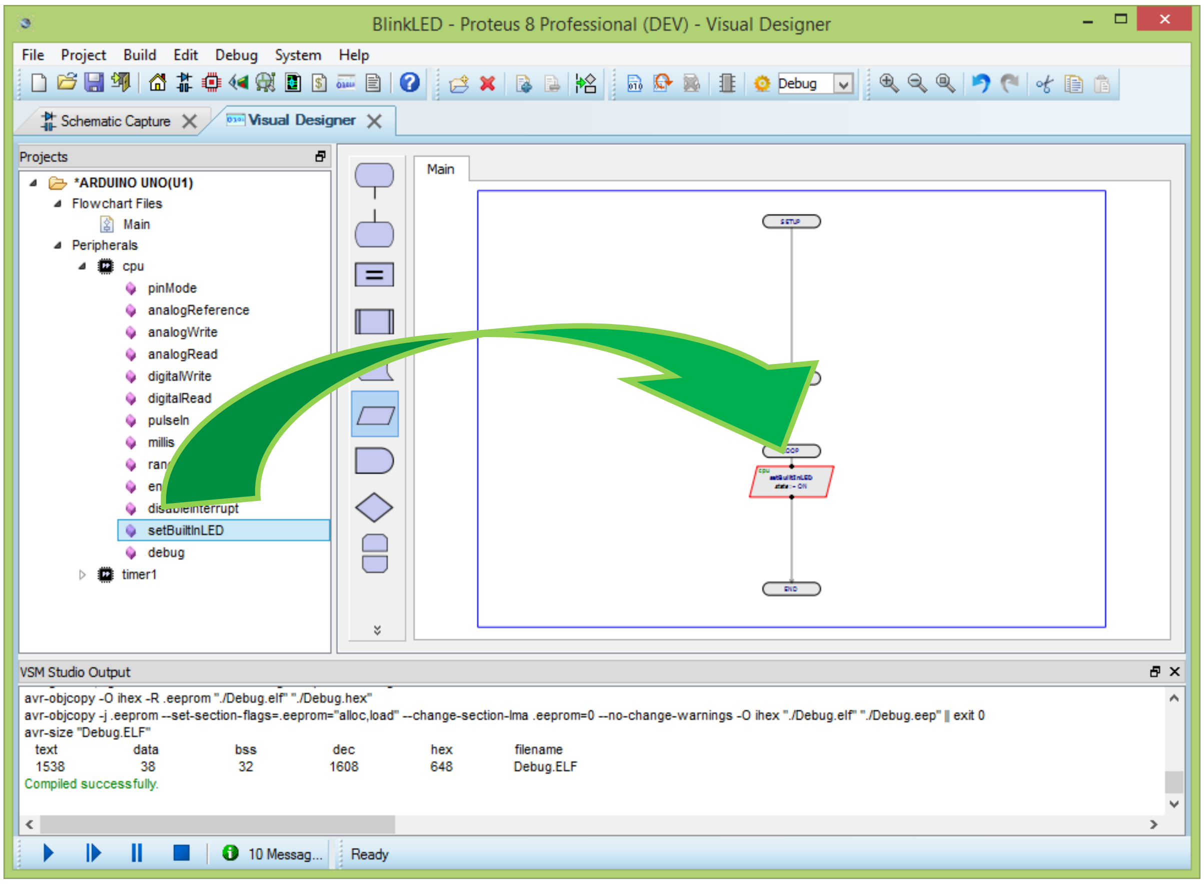


Figure 2-11: Blink LED flowchart

To construct the flowchart, please complete the following instructions:

Step 1: Drag a "setBuiltInLED" method from the CPU peripheral into the flowchart **Loop** routine (Figure 2-12). The method is named "setBuiltInLED" because it can be used to turn a built-in LED on the Arduino Uno board on or off (this is discussed in section 2.7.1 – feel free to read that section and then come back if you are curious).



```
avr-objcopy -O ihex -R .eeprom "./Debug.elf" "./Debug.hex"
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 --no-change-warnings -O ihex "./Debug.elf" "./Debug.eep" || exit 0
avr-size "Debug.ELF"
  text        data        bss        dec        hex        filename
  1538          38         32       1608        648        Debug.ELF
Compiled successfully.
```

Figure 2-12: Adding the "setBuiltInLED" method to the flowchart

When correctly aligned, black dots will appear where the block intersects the flowline (Figure 2-13).

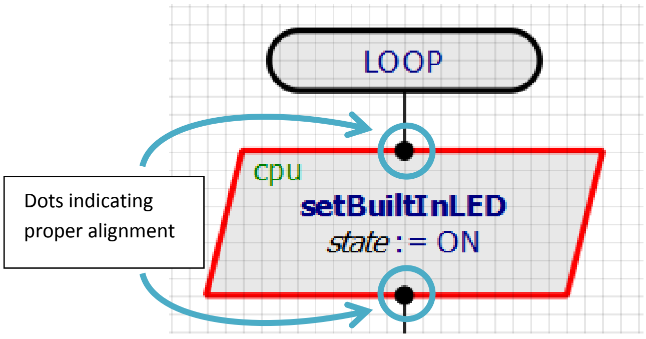


Figure 2-13: Inserting a flowchart block

Once properly inserted, there should be a flowline coming in and a flowline going out of the block, with no dots displayed anymore (Figure 2-14), and arrows at the end of the flowlines.

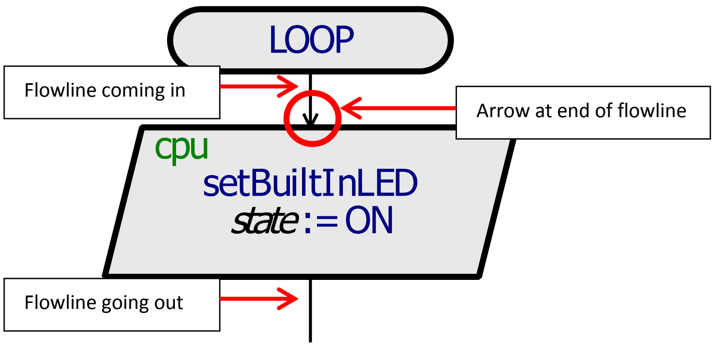


Figure 2-14: Inserted flowchart block

By default, the “setBuiltInLED” method will switch the LED “on” (“state := ON”). This is the correct operation for now, so we do not need to change it. How to change a setBuildInLED block so that it switches the LED off again will be covered in steps 4 and 5.

Step 2: The next step is to add a delay, before we switch the LED off again. This is done using a delay block; so drag a delay block into the flowchart, inserting it after the setBuiltInLED block (Figure 2-15).

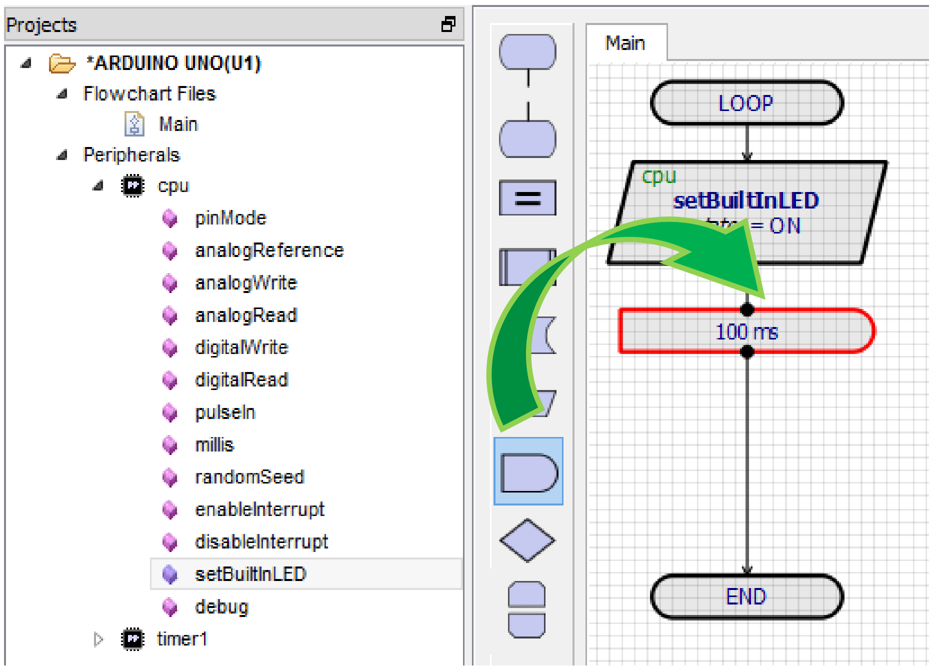


Figure 2-15: Inserting a delay block

Step 3: Next, we want to set the time of the delay to be 1 second. Right-click on the new delay block which was just added to the flowchart, and select "Edit" from the popup menu (Figure 2-16). An alternative method is to left-click on the delay block twice.

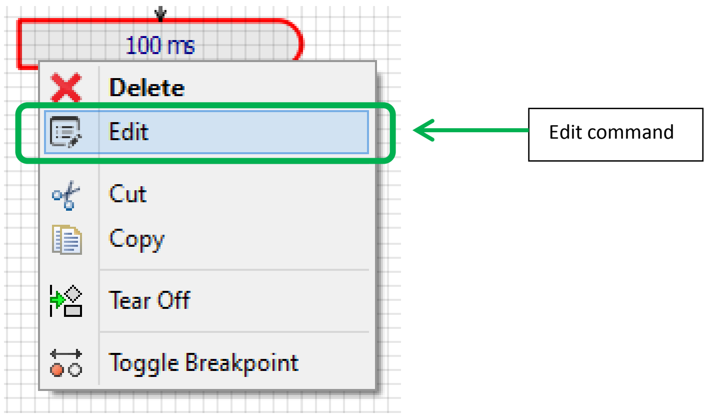


Figure 2-16: Delay block popup menu

Specify "1 second" for the delay (Figure 2-17), and click "OK" when done.

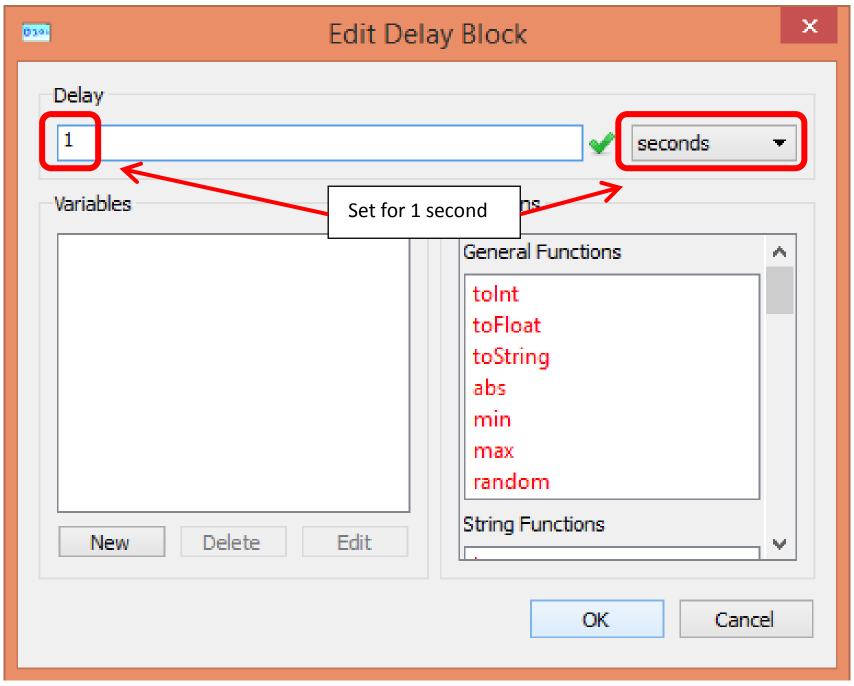


Figure 2-17: Specifying the delay time

The delay block should now look like Figure 2-18.

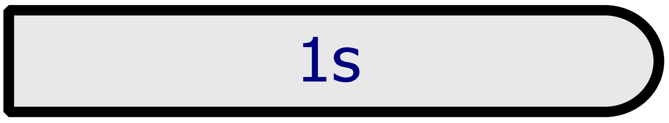


Figure 2-18: Delay block set to 1 second

Step 4: After switching the LED on for 1 second, we now need to switch it off again for another 1 second.
In order to switch the LED off again we need another setBuiltInLED block – so drag another setBuiltInLED method from the cpu peripheral into the flowchart (similar to Step 1), inserting it after the 1s delay.

By default, the setBuiltInLED method switches the LED “on”; we need to change our newly added block so that it instead switches the LED “off”. In order to do so, right click on the block and select “Edit” from the popup menu.

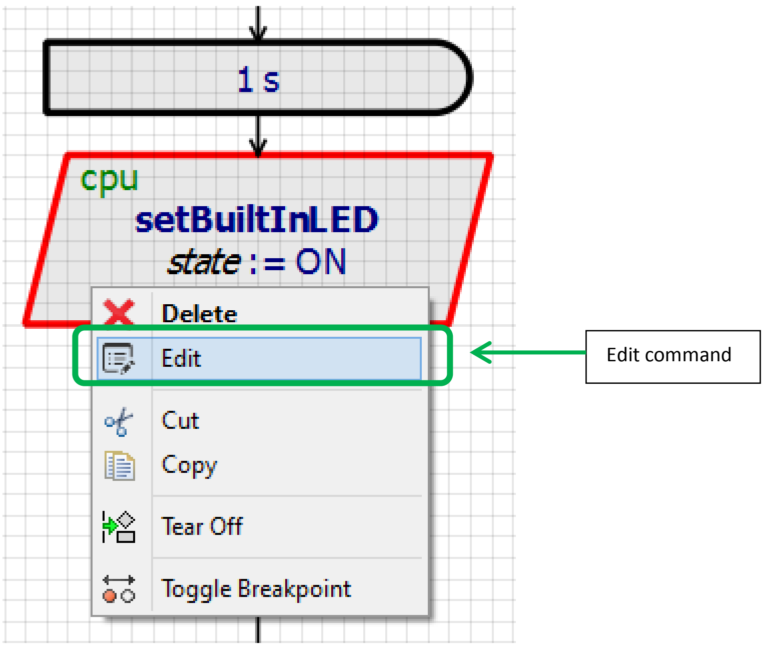


Figure 2-19: setBuiltInLED block popup menu

The “Edit I/O Block” Dialog is now displayed (Figure 2-20).

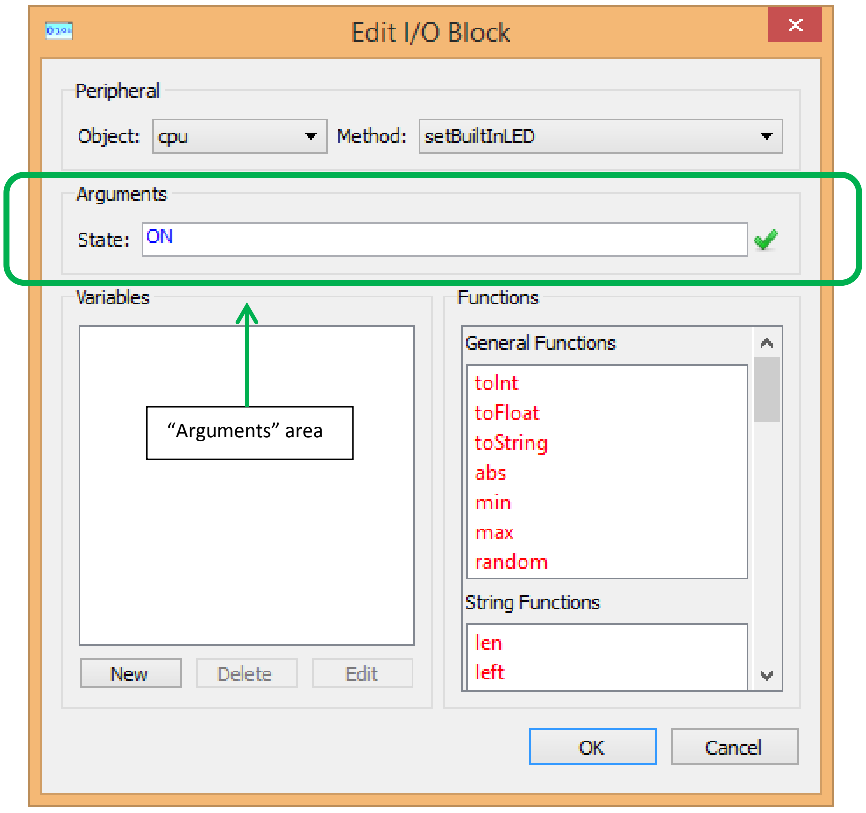


Figure 2-20: Edit I/O Block dialog

Step 5: In the Edit I/O Block dialog, enter "OFF" for the "State:" argument.

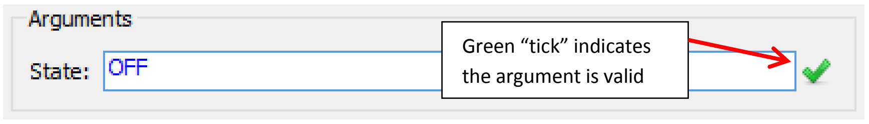


Figure 2-21: Specifying the "State" argument

In programming, an **argument** means a piece of information passed to a method (also sometimes called a "parameter"), to give it more information regarding what it should do. In this case we have the setBuiltInLED method which can turn the LED on or off, but we have to specify which of the two options we want. Note that the term "argument" in programming has **nothing to do with the normal sense of the word** ("diasagreement")!

Once ready click "OK". The setBuiltInLED flowchart block should now look like the following figure:

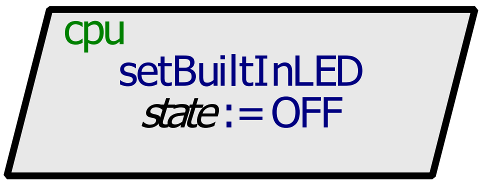


Figure 2-22: setBuiltInLED routine with "state" set to "OFF"

(Tip: For readers who have some prior programming experience, you may be interested to know that "ON" in Proteus Visual Designer is a defined alias for "TRUE".)

Step 6: Now that the LED has been switched off again, we need to keep it off for 1 second before the loop repeats and switches the LED back on again. In order to do so, add a 1s delay after the second setBuiltInLED routine (similar to steps 2 and 3).

Once done, the Loop routine of the flowchart should look like the following figure.

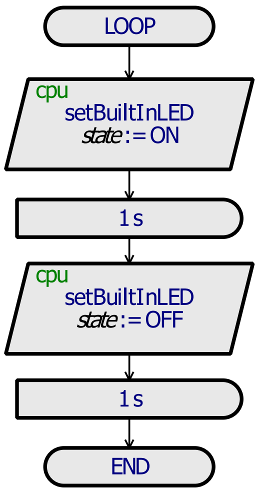


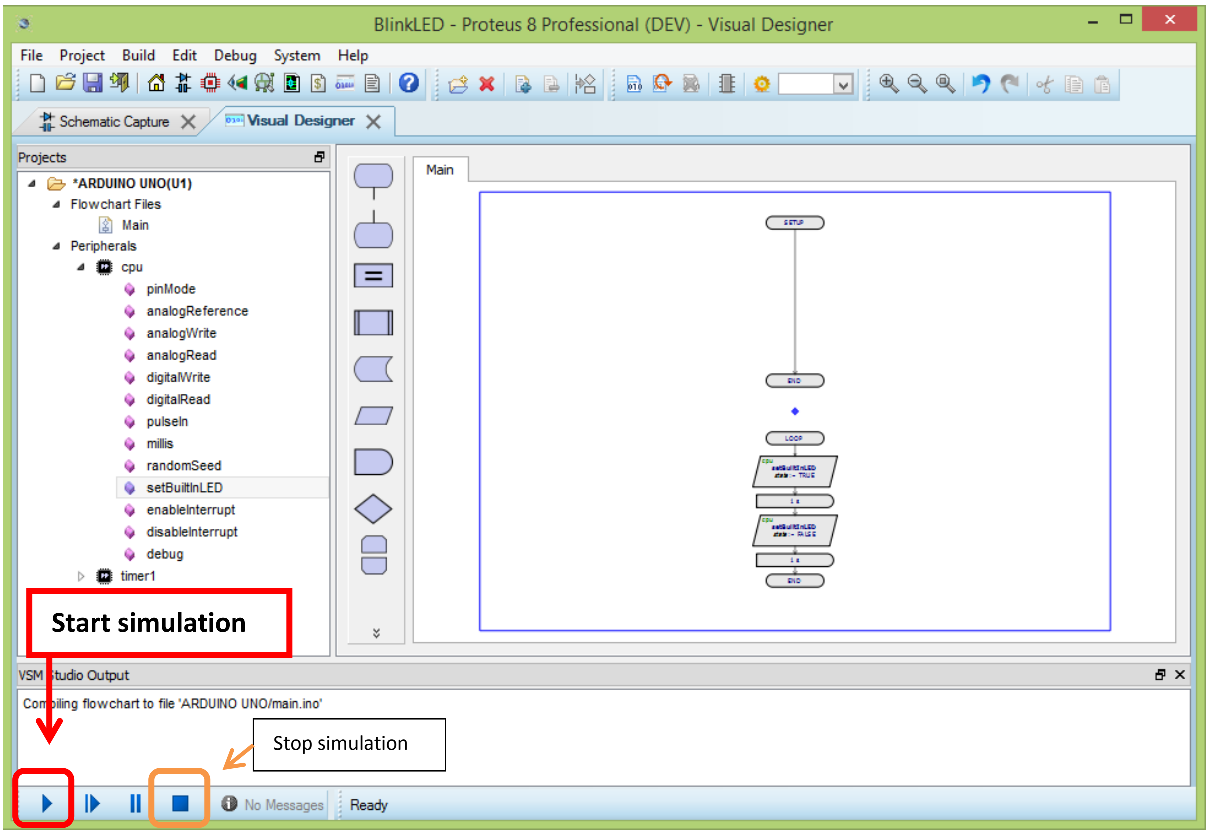
Figure 2-23: Blink LED flowchart

Check that the sequence and details of the blocks in your flowchart exactly match those of Figure 2-23 (if not, then please go through the steps of this tutorial again to check that nothing has been missed). The flowchart Setup routine should still be empty.

Congratulations! The “Blink LED” program flowchart is now complete.

2.6. Running the Simulation

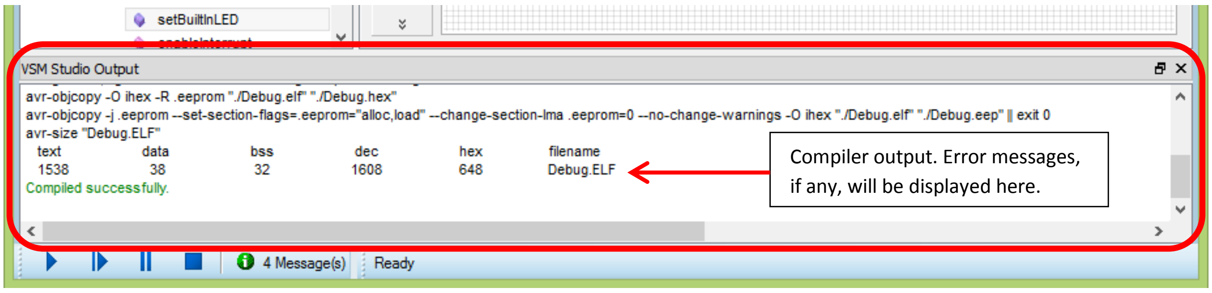
To see our flowchart program in action, we can proceed to running the simulation. To do so, click the blue “Play” button at the bottom-left of the Proteus window (Figure 2-24).



```
Compiling flowchart to file 'ARDUINO UNO/main.ino'
```

Figure 2-24: Starting the simulation

Upon clicking the blue triangular play button, Proteus first compiles the flowchart so that it can be simulated (please see Step 3 of section 2.2 if you need a reminder of what a compiler does).


VSM Studio Output
Compiler output. Error messages, if any, will be displayed here.
4 Message(s)
Ready

```
avr-objcopy -O ihex -R .eeprom "./Debug.elf" "./Debug.hex"
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 --no-change-warnings -O ihex "./Debug.elf" "./Debug.eep" || exit 0
avr-size "Debug.ELF"
   text       data        bss        dec        hex    filename
   1538         38         32       1608        648    Debug.ELF
Compiled successfully.
```

Figure 2-25: Compiler output

If there are any problems with the flowchart, error messages will be displayed in the compiler output. If not, then a green “Completed successfully” message will be displayed.

(Tip: Compilation can also be triggered manually by selecting “Build Project” from the “Build” menu.)

Once the project has compiled successfully, the simulation will start (Figure 2-26).

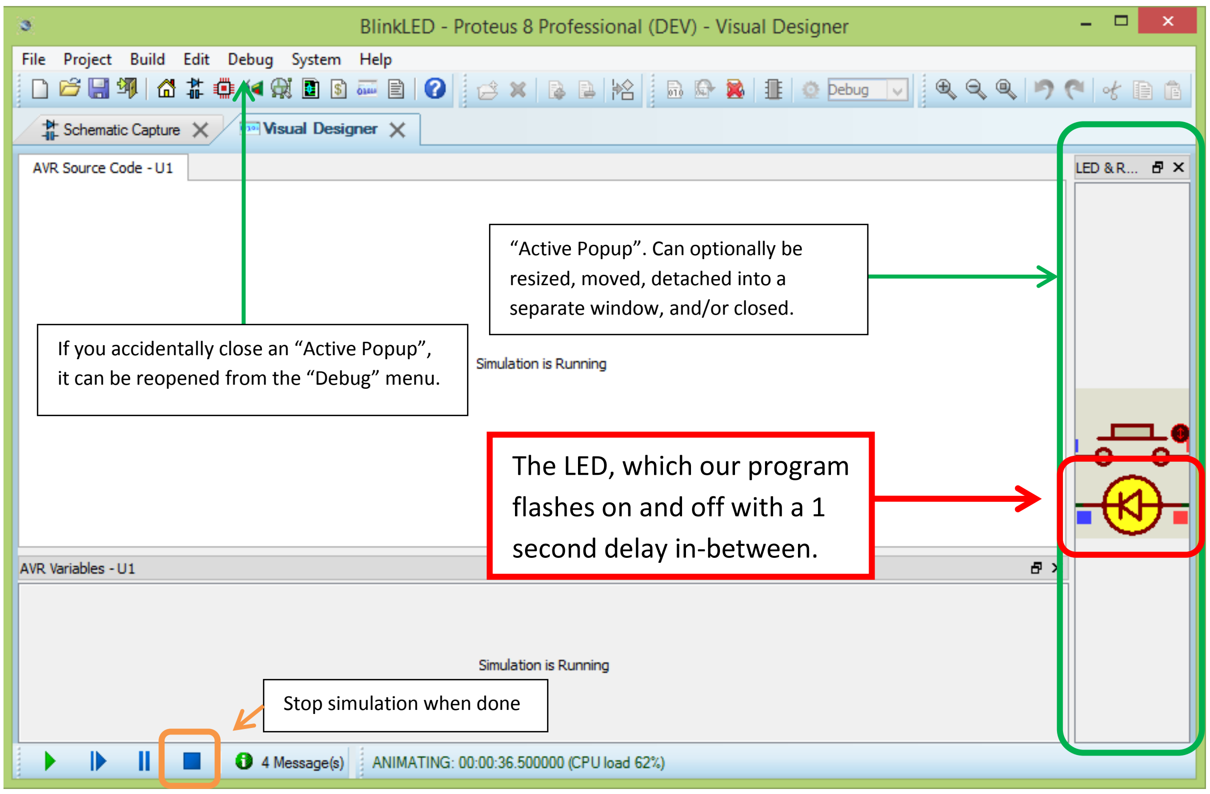
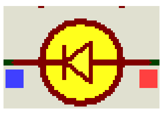
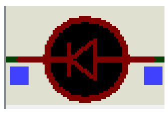

BlinkLED - Proteus 8 Professional (DEV) - Visual Designer
File Project Build Edit Debug System Help
Debug
Schematic Capture
Visual Designer
AVR Source Code - U1
LED & R...
"Active Popup". Can optionally be resized, moved, detached into a separate window, and/or closed.
If you accidentally close an "Active Popup", it can be reopened from the "Debug" menu.
Simulation is Running
The LED, which our program flashes on and off with a 1 second delay in-between.
AVR Variables - U1
Simulation is Running
Stop simulation when done
4 Message(s)
ANIMATING: 00:00:36.500000 (CPU load 62%)

Figure 2-26: Running simulation

You should see a yellow LED flashing on and off with a 1 second delay in-between, as specified by our flowchart program.

LED On:  LED Off: 

Once done, click the blue rectangular "Stop" button (see Figure 2-26) to end the simulation. Congratulations! You have now built and simulated a working program.

(Tip: You may wish to save your work. This can be done using the "Save Project" command available from the File menu, by clicking on the Save button ( ) on the toolbar, or by pressing Ctrl + S on your keyboard.)

If you would like to experiment a little more, then try changing the time of the delays and see how this changes the timing of the LED blinking.

If you are wondering how your program can be transferred to "real world" hardware, then please note that this is discussed in section 2.7.4.

2.7. Further Information

2.7.1. What LED Are We Blinking?

The Arduino Uno which we selected for use in our project during the "New Project Wizard" (see Figure 2-3) is a low-cost "development board"; it essentially features a microcontroller, some supporting electronic circuitry (such as circuitry for handling the power supply to the board), and a built-in on-board LED. Connectors ("headers" or "sockets") are also featured around the edges of the board, which can be used to more easily connect external peripheral circuits to the microcontroller (see Figure 2-27).

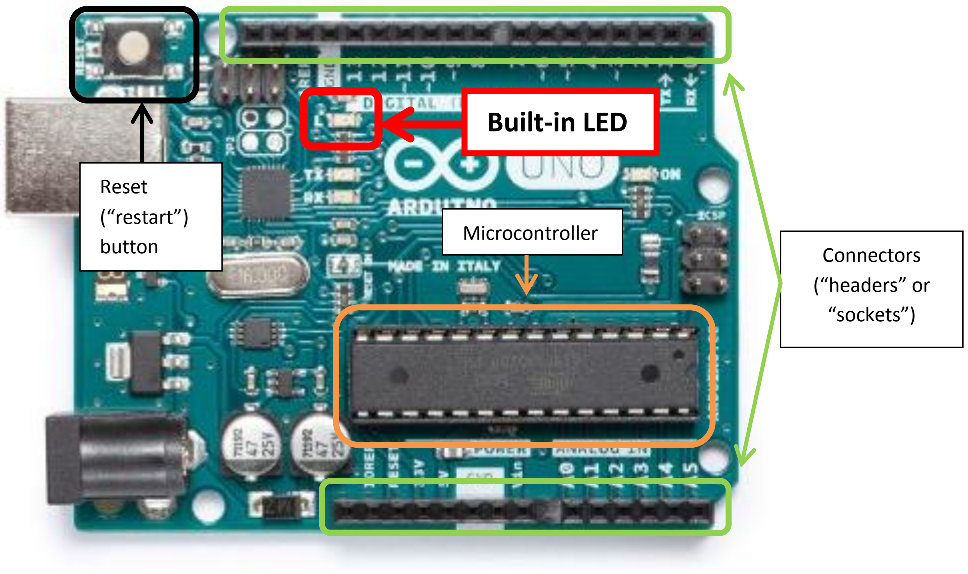


Figure 2-27: Arduino Uno board[7]

In Proteus, the Arduino Uno circuitry is contained in the Schematic Capture tab. As mentioned in section 2.3.1, **it is not necessary to understand the schematic capture tab or circuitry for now**, however for the curious the schematic circuit looks as follows:

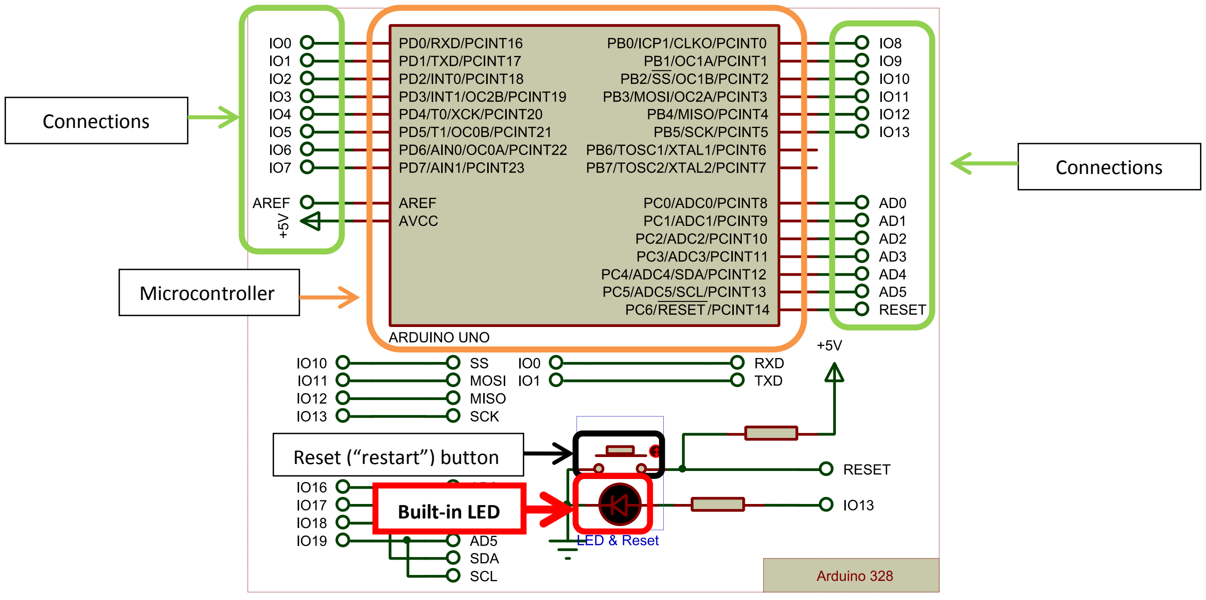


Figure 2-28: Arduino Uno schematic

[7] Image from https://store.arduino.cc/usa/arduino-uno-rev3

2.7.2. "Active Popups"

When we click the "Play" button, Proteus actually simulates the circuit which exists on the Schematic Capture tab. "Windows" onto certain areas of the schematic capture tab however – called "Active Popups" – can be used to display those specific areas of the schematic within the Visual Designer tab during simulation. Thus, it is possible to view the simulation on either the Schematic Capture or Visual Designer tabs. "Active Popups" are covered in detail in section 7.7. This information is once again provided for the curious, and **it is once again not necessary to understand the schematic capture tab or circuitry for now**.

Note that Active Popups (see Figure 2-26) can be resized, moved, detached into a separate window, and/or closed. If you should accidentally close an Active Popup, it can be reopened again from the Debug menu.

2.7.3. Flowchart Editing Skills

Flowchart editing in Visual Designer is designed to be intuitive. Some details are however listed here, to make sure that nothing is missed.

- Blocks already on the sheet can be moved by dragging and dropping them.
- Blocks can be deleted by selecting them and then pressing the "Delete" button on the keyboard, or by right-clicking on them and then selecting "Delete" from the popup menu.
- Multiple blocks can be selected by dragging a box around them with the mouse.
- Blocks can be cut, copied and pasted using the relevant commands available from the Edit menu, and/or using the keyboard shortcuts:
    - Cut: Ctrl + X
    - Copy: Ctrl + C
    - Paste: Ctrl + V
- Flowlines can also be adjusted and deleted, using similar methods to those just listed.

One operation which may not be intuitive however, is detaching blocks so that they can be repositioned within the schematic. To detach a block, right-click on it and then select "Tear Off" from the popup menu (as illustrated in the following figure):

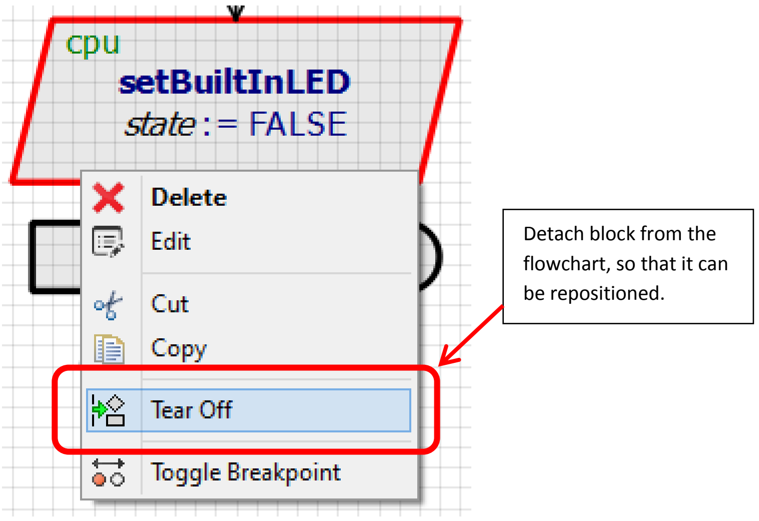


Figure 2-29: Detaching a block from the flowchart

The “Tear Off” command is also available from the main Edit menu. Blocks can also be detached by holding down the Ctrl key on the keyboard, then clicking on them and dragging them out of the flowchart.

You may wish to try out some of the above techniques now, to make sure that you are familiar with them.

2.7.4. Uploading the Program to Physical Hardware (Optional)

If you have the physical hardware for a project available (in this case, just an Arduino Uno board), then your program can be uploaded to that hardware with ease straight from within Proteus. The commands needed to accomplish this are available from the Build menu (when the Visual Designer tab is open).

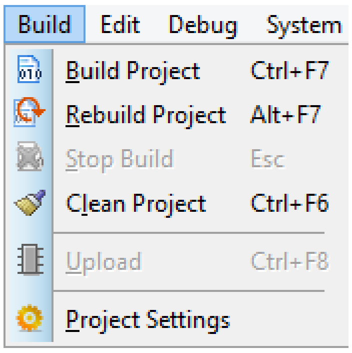
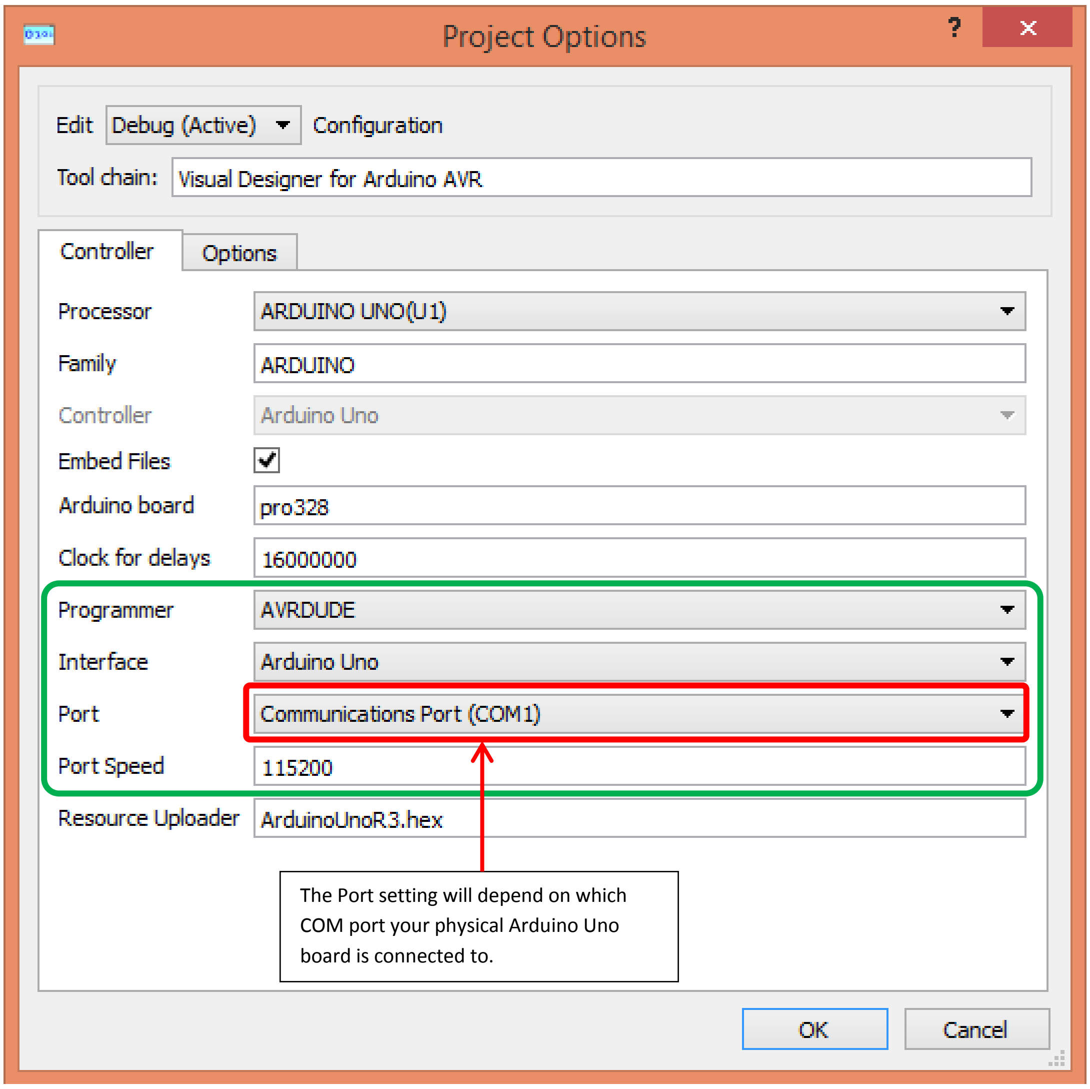


Figure 2-30: Build menu

First, the correct settings should be set for uploading. This is done in the “Project Options” dialog, which can be opened by selecting the “Project Settings” option from the Build menu (when the Visual Designer tab is open). The Project Options dialog is displayed in the following figure (Figure 2-31).



Debug (Active)

Tool chain: Visual Designer for Arduino AVR

Processor: ARDUINO UNO(U1)
Family: ARDUINO
Controller: Arduino Uno
Embed Files: ✓
Arduino board: pro328
Clock for delays: 16000000
Programmer: AVRDUDE
Interface: Arduino Uno
Port: Communications Port (COM1)
Port Speed: 115200
Resource Uploader: ArduinoUnoR3.hex

The Port setting will depend on which COM port your physical Arduino Uno board is connected to.

Figure 2-31: Project Options dialog

1. Select "AVRDUDE" as the Programmer.
2. Select "Arduino Uno" as the Interface.
3. Select the COM port to which your physical Arduino Uno board is connected to. Usually this will be labelled as "USB Serial Device", and will be COM3 or higher. One technique which can be used to determine the COM port number is as follows:
    a. Make sure that the Project Options dialog is closed.
    b. Unplug the Arduino Uno from the computer, open the Project Options dialog, and note the COM ports which are listed.
    c. Plug the Arduino Uno board into the computer.
    d. Close and reopen the Project Options dialog. A new COM port should be listed, and this will be the one to which the Arduino Uno board is connected. If no new COM port appears, then it is likely that the USB drivers are not installed, or otherwise that there is a hardware problem (faulty board / faulty cable / faulty computer). The "Programming the Physical Hardware" chapter in the Visual Designer help file (available from the Help menu when the Visual Designer tab is selected), may be of assistance.
4. Port Speed should be 115200.

Once the correct settings have been set, it must be ensured that the project has been compiled ("Built"). Select "Build Project" from the Build menu in Visual Designer (when the Visual Designer tab is open) in order to compile the project.

If the project settings are correct and the project has been built (compiled), then uploading can be performed by selecting "Upload" from the Build menu, or using the dedicated button on the toolbar (see Figure 2-32).

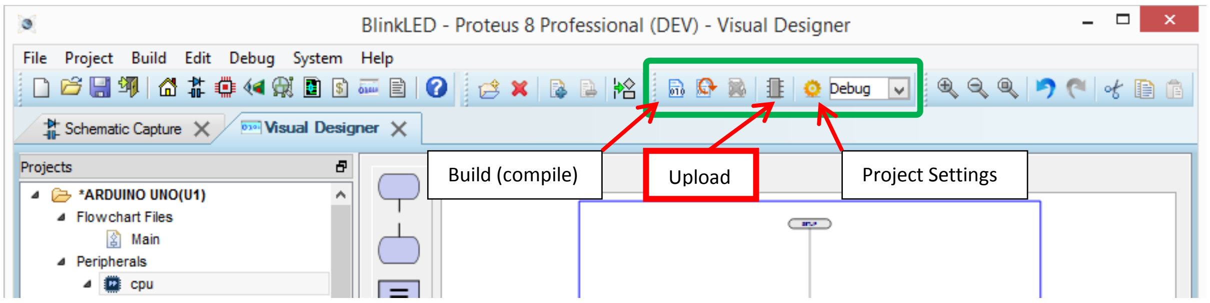


Figure 2-32: Build, Upload and Project Settings buttons

2.7.5. Additional Resources

A number of step-by-step tutorials and explanations are also included in the Proteus Visual Designer Help File, which is available from the Help menu when the Visual Designer tab is open (and selected).

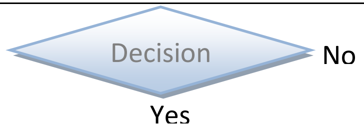
2.8. Challenges

Optional challenges (grouped by chapter) are listed in Appendix A. Challenges can be used to practice skills learned in a chapter (which can help to make sure that the chapter was understood, and can also assist with learning), but it is not required to complete them in order to progress through the book. Possible solutions to the challenges are listed in Appendix B. The challenge for this chapter is to blink an LED in a way that visually mimics the sound of a human heartbeat – if you're interested and up for the challenge than take a look at Appendix A, otherwise just carry straight on to the next chapter (you can always get back to the challenges later).

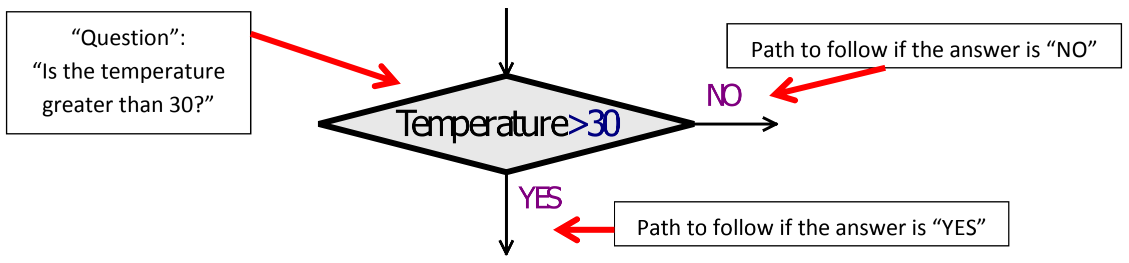
3. Reading Inputs and Making Decisions

3.1. Introduction to Decision Blocks

In the world of programming there are two fundamental types of operations:

	
Actions ("Telling")	Decisions ("Asking")

Whilst our first project introduced actions (by "telling" a light to turn on and off), this chapter introduces decisions ("asking"). Whilst actions only have one flowline coming in and one flowline going out, decisions have one flowline coming in and two flowlines going out. Depending on the answer to the question which the decision block asks, the program will choose to follow one of the two outgoing lines.


"Question":
"Is the temperature greater than 30?"
Temperature>30
Path to follow if the answer is "NO"
NO
YES
Path to follow if the answer is "YES"

3.2. Project Summary

For our next project we will be asking a button whether it is pressed or not, and then telling a light to turn on or off depending on the answer:

a) If the button is pressed then the light should turn on.
b) If the button is not pressed then the light should turn off.

The flowchart for this project is detailed in the following figure.

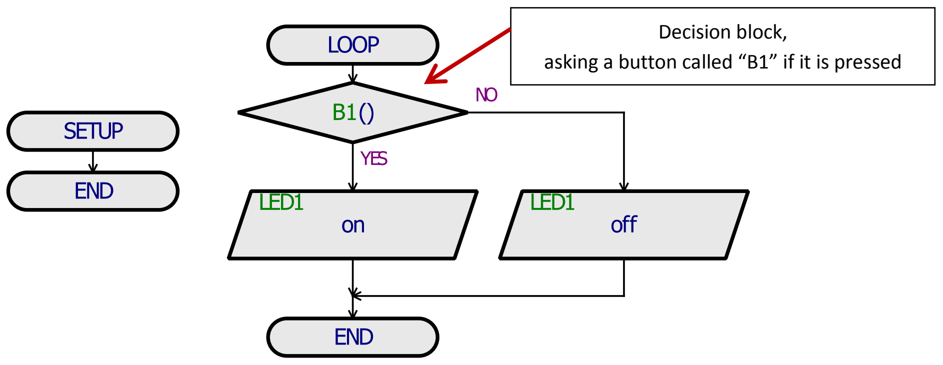


Figure 3-1: Button-LED flowchart

When the loop starts, the program asks the button (B1) whether it is pressed or not (decision block). If the answer is "Yes" then the program goes the route of switching the LED on, and if the answer is

"No" then the program goes the route of switching the LED off. Since these flowchart blocks are placed within the "Loop" routine (see section 2.4.3), the process will obviously repeat continually.

3.3. Starting the Project

To get started building the flowchart, please complete the following steps:

1. Save ( ) any existing project which you may have open (if you do not want to lose it). This can be done using the "Save Project" command available from the File menu.
2. Close ( ) any existing project which you may have open. This can be done using the "Close Project" command available from the File menu. You should now see the Proteus Home Page ( ) tab, as illustrated in Figure 2-1.
3. Start a new flowchart project, using "Button-LED" as the project name (if you need a reminder on the steps, then please see section 2.2).

3.4. Adding Peripherals to the Project

As discussed in section 2.3.4, peripherals are largely objects which can be added to the project, such as buttons / display screens / temperature sensors / etc. For this project, we are going to add a button and an external LED (different to the built-in LED which we used in the previous chapter) to the project.

Peripherals are added to the project using the "Peripheral Gallery", which can be accessed by selecting "Add Peripheral" from the Project menu. The Peripheral Gallery can also be accessed by right-clicking on the Peripherals section of the project summary, and then selecting "Add Peripheral" from the popup menu (see Figure 3-2).

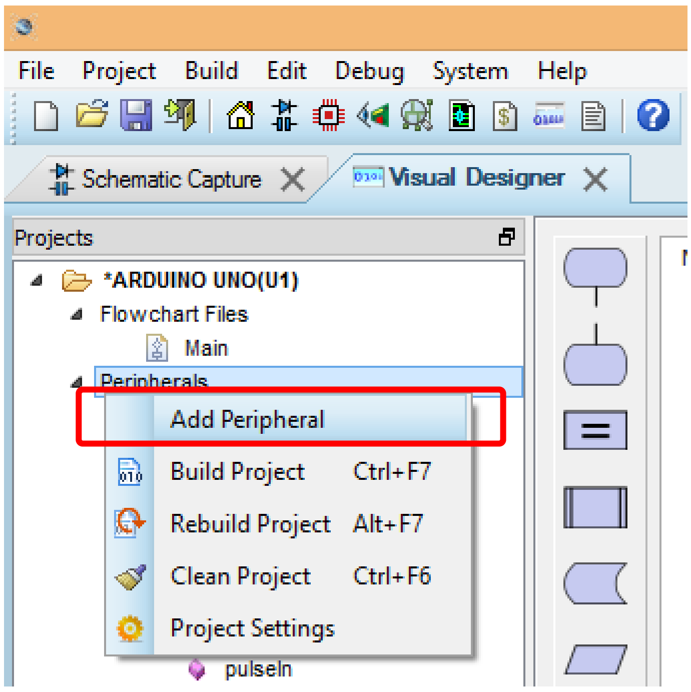


Figure 3-2: Add Peripheral command

Once the Peripheral Gallery opens, select "Grove" as the category and then add a "**Momentary Action Push Button**" as well as a "**Grove LED (Green)**" to the project. Peripherals can be added to the project from the peripheral gallery by double-clicking on them, or by selecting them and then clicking the “Add” button.

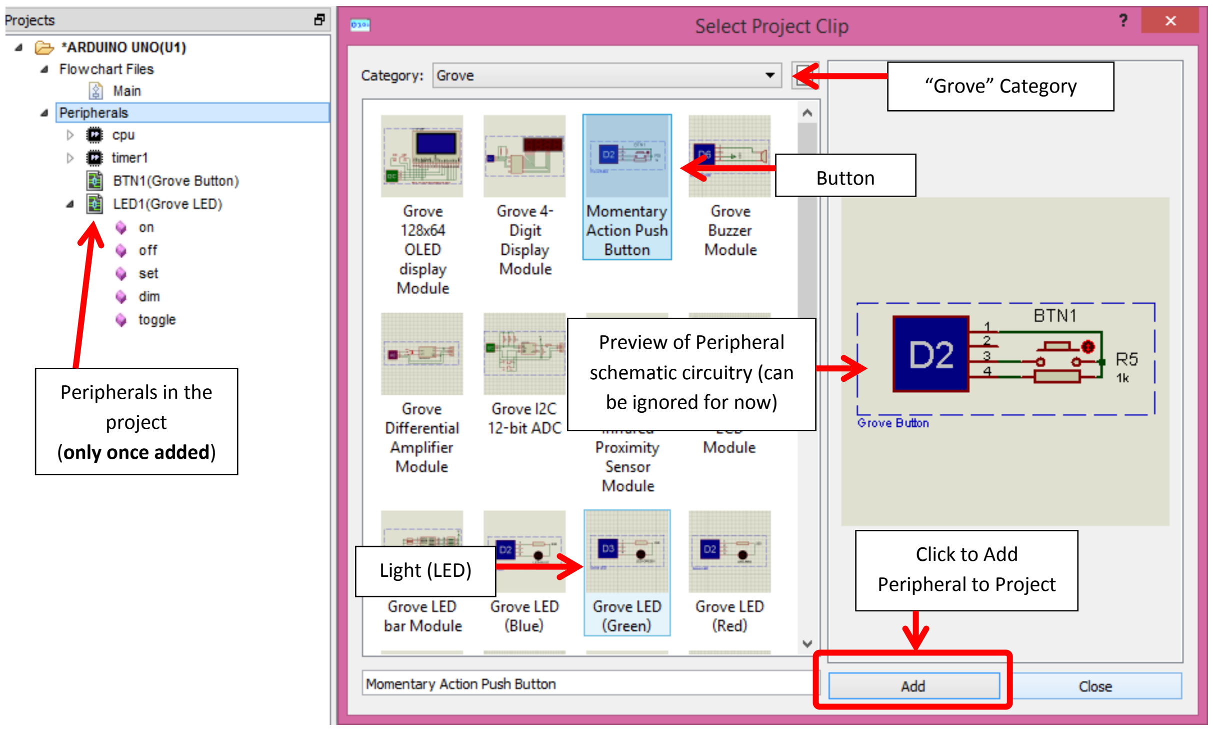


Figure 3-3: Button-LED Project Peripherals

The “Peripherals” section of the Project Summary should now contain “BTN1 (Grove Button)” and “LED1 (Grove LED)” peripherals (as illustrated in the following figure).

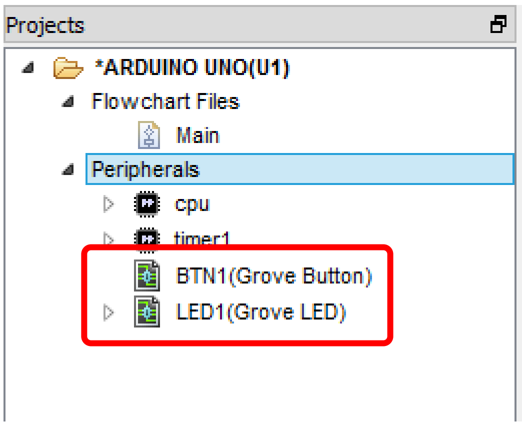


Figure 3-4: Peripherals added to Project

If you have accidentally added more peripherals than were required, then these can be removed from the project again by right-clicking on them and then selecting “Remove Peripheral” from the popup menu.

More information regarding what exactly the “Grove” peripherals are is included at the end of this chapter (sections 3.8.1 and 3.8.2).

3.5. Drawing the Flowchart

Once the peripherals have been added to our project, then the next step is to add the decision block, which asks the button whether it is pressed or not, to our flowchart. This can be done by clicking on the button peripheral and then dragging it into the Loop routine (as illustrated by the following figure).

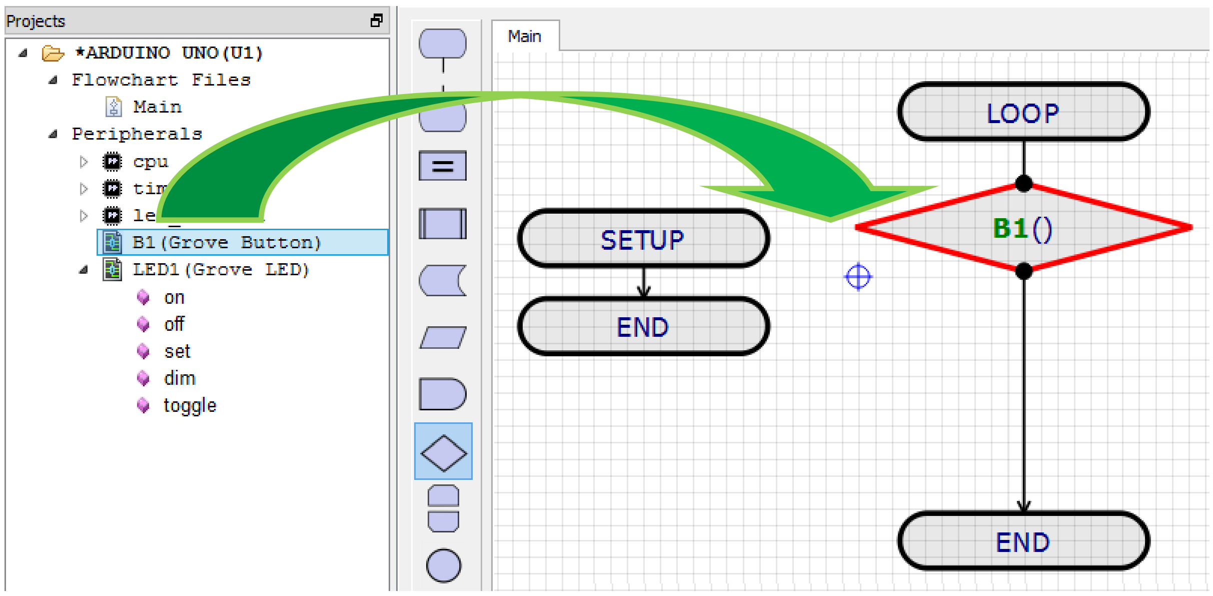


Figure 3-5: Inserting the button decision block

Now that the decision block has been added, the next steps are to add the actions to be taken based on the possible outcomes of the decision. To proceed, drag an "on" method from LED1 into the flowchart, inserting it after the button decision block (as illustrated in the following figure).

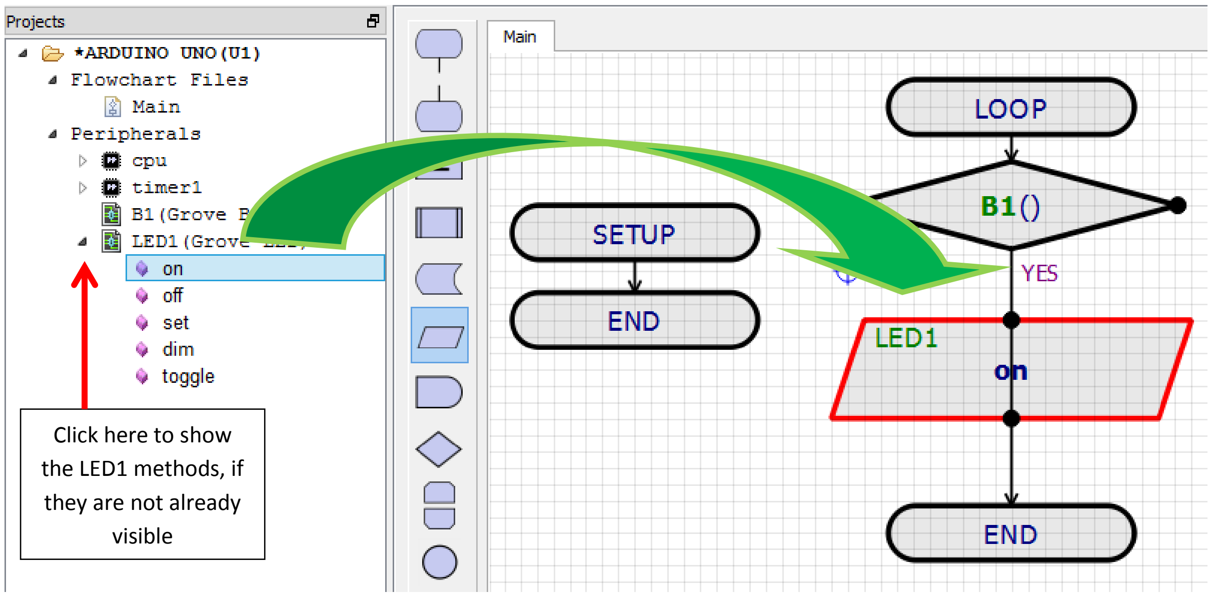


Figure 3-6: Inserting LED "on" method

The last block needed to complete the flowchart is an "off" method from LED1. Add it as depicted in the following figure.

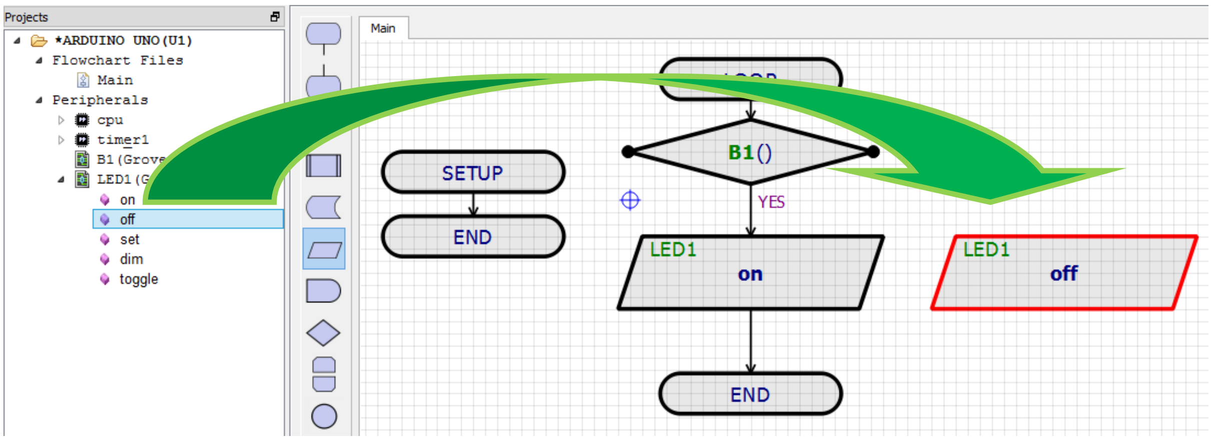


Figure 3-7: Inserting LED "off" method

Whilst the "off" block has now been added to the flowchart, it is still not "connected" to it yet. To connect it to the button decision block, please complete the following sequence:

Click once on the black dot on the right of the button decision block, and then click again on the black dot on the top of the "off" action block. This procedure is illustrated in the following figure.

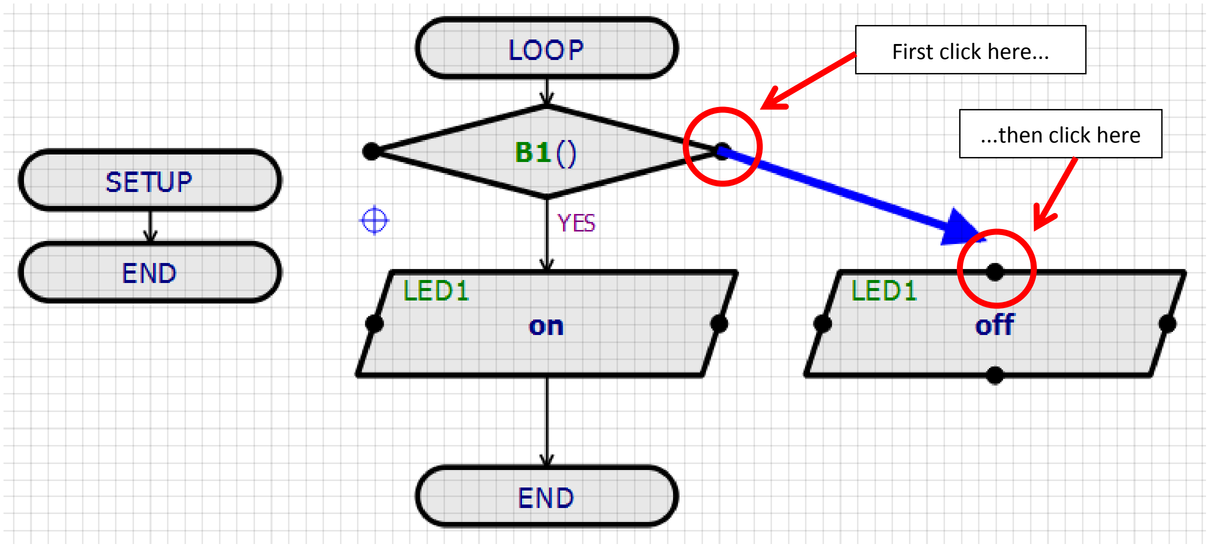


Figure 3-8: Connecting flowchart blocks

The "off" block can now be reached from the button decision block, however the flowchart doesn't have anywhere to go from there (once it has reached the "off" block, it is "stuck"). In order to complete the connection process, add a flowline from the "off" block back to the main loop routine, as illustrated in the following figure:

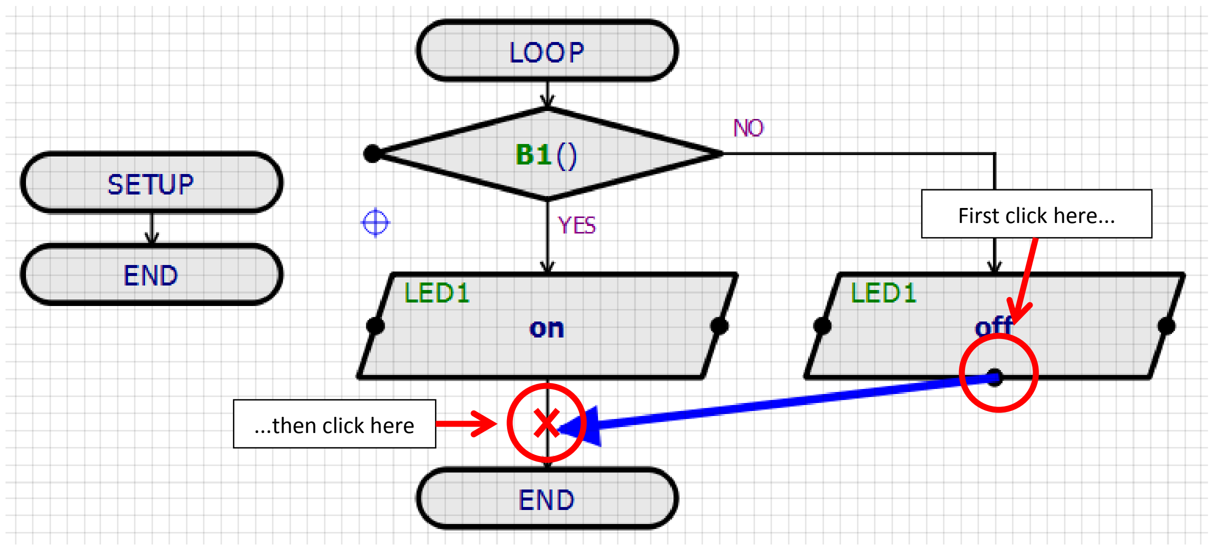


Figure 3-9: Connecting flowchart blocks 2

Now, if the program flow reached the "off" block, it can carry on back to the loop routine, and continue program execution. Your flowchart should now look similar to the following figure.

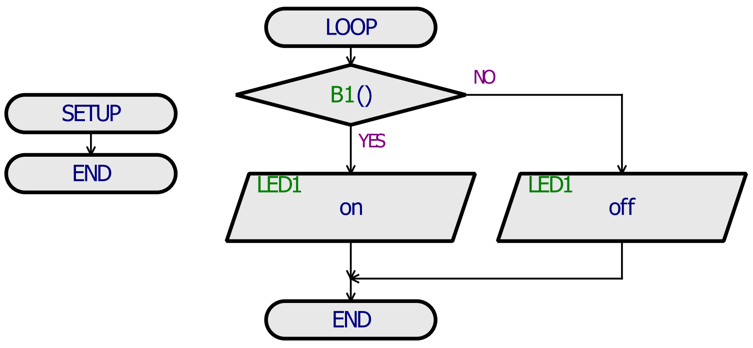


Figure 3-10: Completed Button-LED flowchart

Don't worry if the positions of the blocks don't exactly match those of Figure 3-10. So long as the ordering ("sequence") of the blocks is the same, the program will work.

3.6. Running the Simulation

Once ready, click on the blue triangular "Play" button (at the bottom-left of the window) to run the simulation. If you receive any error messages, then double-check that the flowchart matches Figure 3-10, and that it was a "Grove LED (**Green**)" peripheral which was added to the project (the reason for this is discussed at the end of this chapter, in section 3.8.2). Once the simulation starts, you can click on the Grove Button to turn the Grove LED on (see Figure 3-11; the Grove Button and LED would normally appear at the bottom-right of the screen).

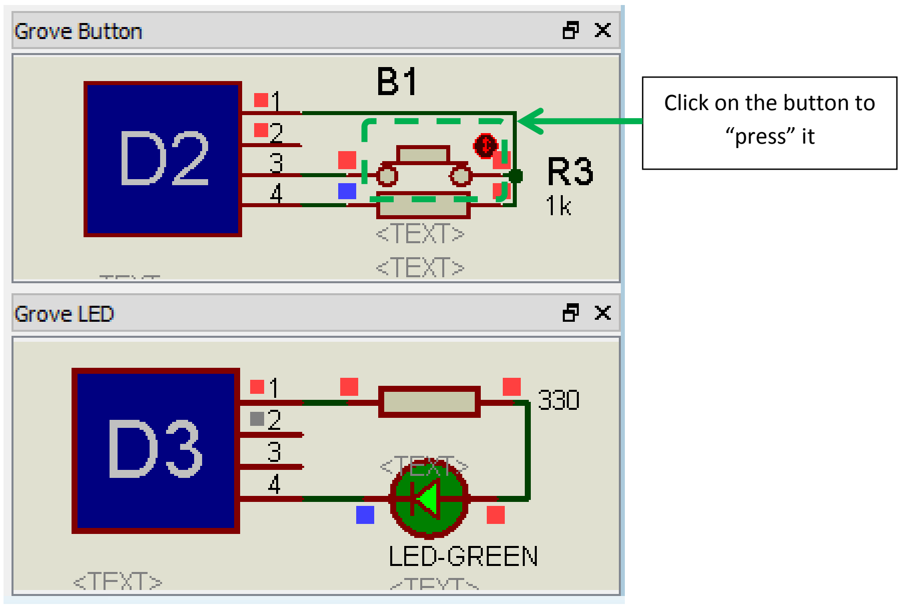


Figure 3-11: Grove Button and Grove LED Active Popups

Remember to stop the simulation when done (blue square "Stop" button at the bottom-left of the window).

3.7. Inverting the Logic

Next, let's swop things around so that the LED is off when the button is pressed, and on when it is not pressed. To do so, first make sure that the simulation has been stopped (blue rectangular stop button at the bottom-left of the window). Next right-click on the decision block in the project and select "Swap Yes/No" (Figure 3-12).

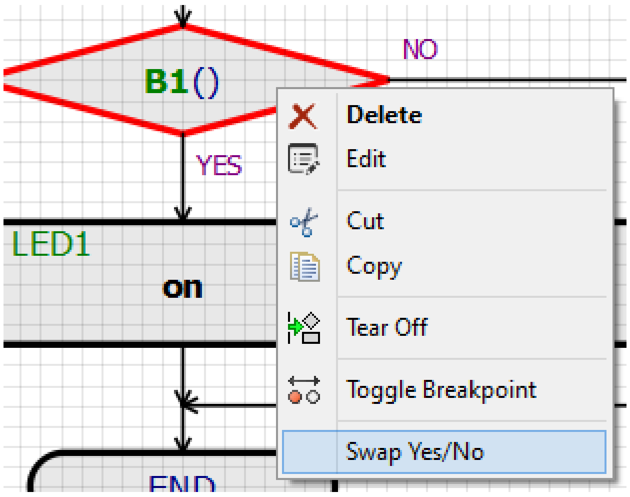


Figure 3-12: Swapping Yes and No on a decision block

The flowchart should now look like the following figure:

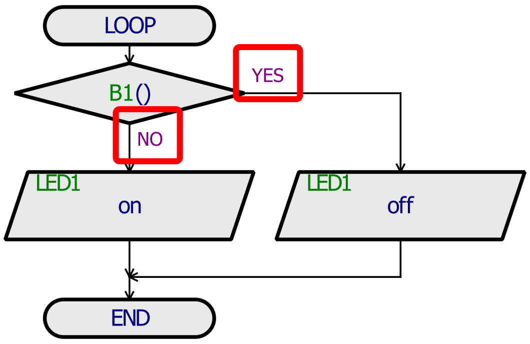


Figure 3-13: Flowchart with logic inverted

Run the simulation again, and you should now find that the button does the opposite of what it did before – the LED should now turn on when the button is <u>not</u> pressed, and turn off when it is pressed (Figure 3-14 and Figure 3-15).

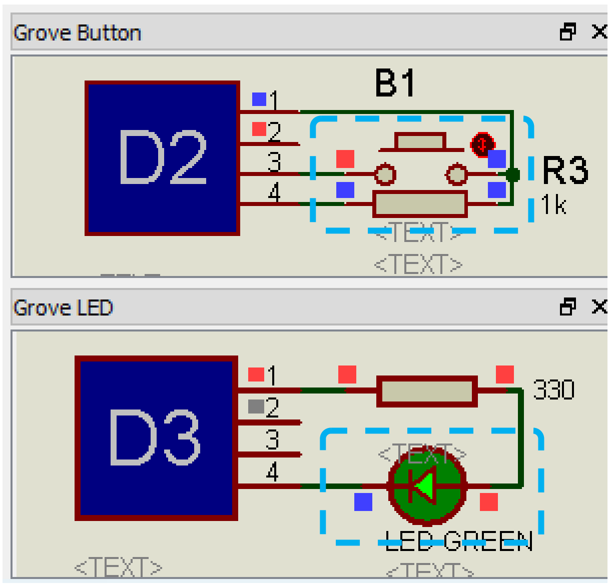


Figure 3-14: Button NOT pressed

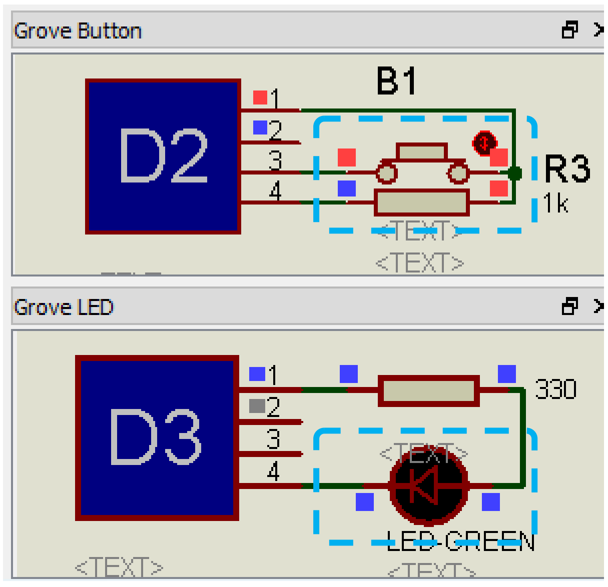


Figure 3-15: Button pressed

That concludes the project for this chapter! If you have completed and understood the chapter, then you now know how to use decision blocks in a project!

3.8. Further Information

3.8.1. The "Grove" System

"Grove" is a system of add-on peripheral boards produced by Seeed Studio (<u>www.seeedstudio.com</u>). The system includes many interface and sensor boards, such as the button and LED boards which we have just used in this project. The peripheral boards are connected to a "Base Board" via cables. The "Base Board" is plugged in on top of the Arduino Uno.

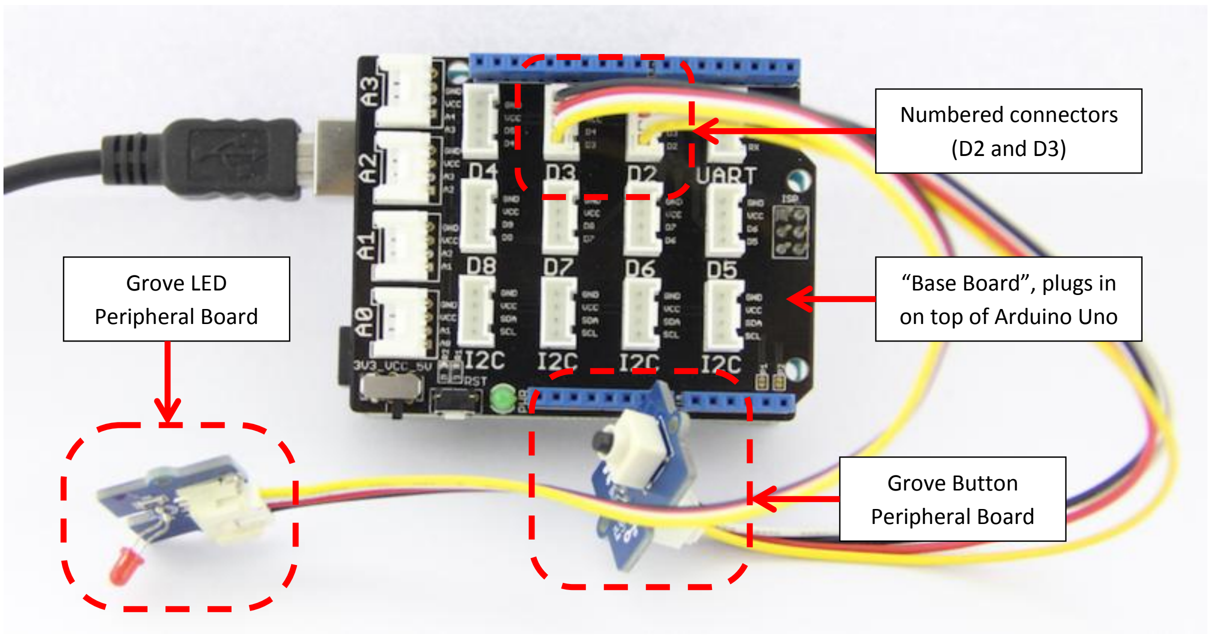

Numbered connectors (D2 and D3)
Grove LED Peripheral Board
"Base Board", plugs in on top of Arduino Uno
Grove Button Peripheral Board

Figure 3-16: Grove System[8]

3.8.2. Changing the Grove Connector Number in Proteus

As is visible in Figure 3-16, the connectors on the Grove Base Board are numbered. When we added the Grove Button and Grove LED peripherals to our project from the peripheral gallery (section 3.4), then Proteus automatically added the relevant electronic circuitry for these peripherals to the schematic.

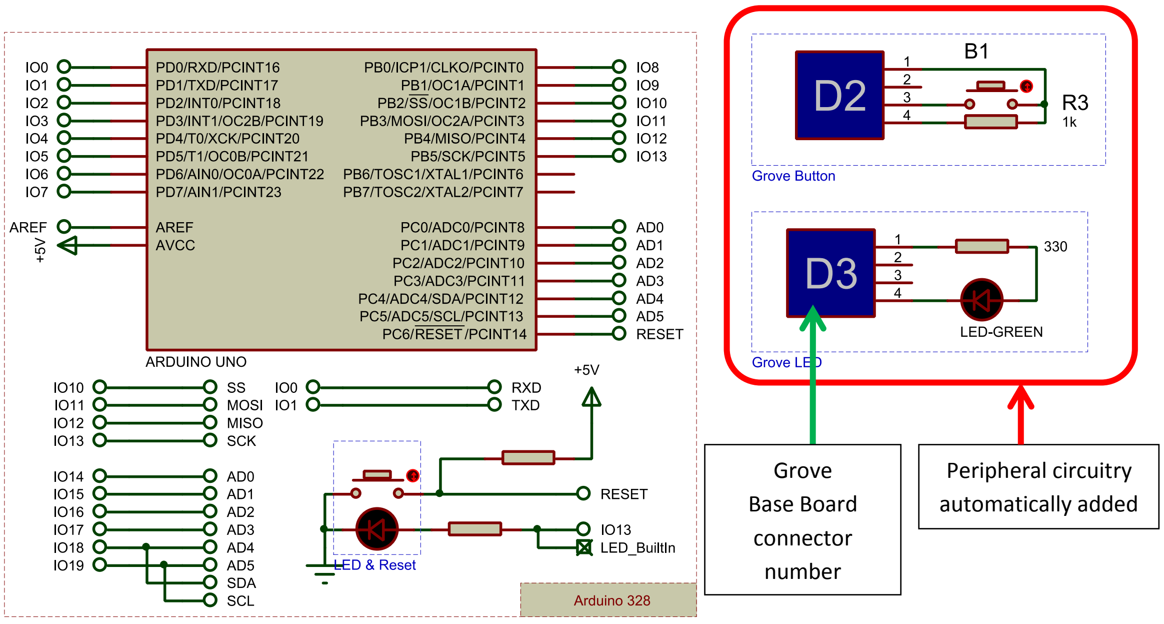
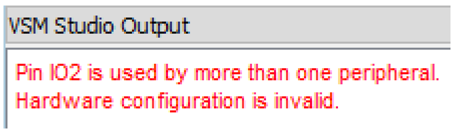


Figure 3-17: Schematic with peripheral circuits

The peripherals for this project were carefully chosen, as the Grove Button is connected to "D2" by default, and the Grove LED (Green) is connected to "D3" by default. The Blue, Red and Yellow Grove LEDs are however connected to "D2" by default – obviously it is not possible to plug two boards into

[8] Image from https://pmtechnologie.shost.ca/spip.php?article21, Creative Commons Attribution 2.5 License, unmodified except for labels added.

the same connector, and so if we had tried to use one of these LEDs then Proteus would have reported:


VSM Studio Output

```
Pin IO2 is used by more than one peripheral.
Hardware configuration is invalid.
```

Figure 3-18: Hardware conflict error

It is however quite simple to “plug” Grove peripherals into a different connector number in Proteus! This can be done on the Schematic Capture tab by right-clicking on the connector number, and then selecting “Edit Properties” from the popup menu.

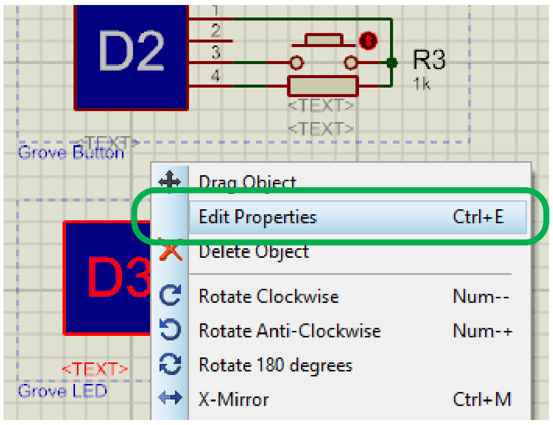
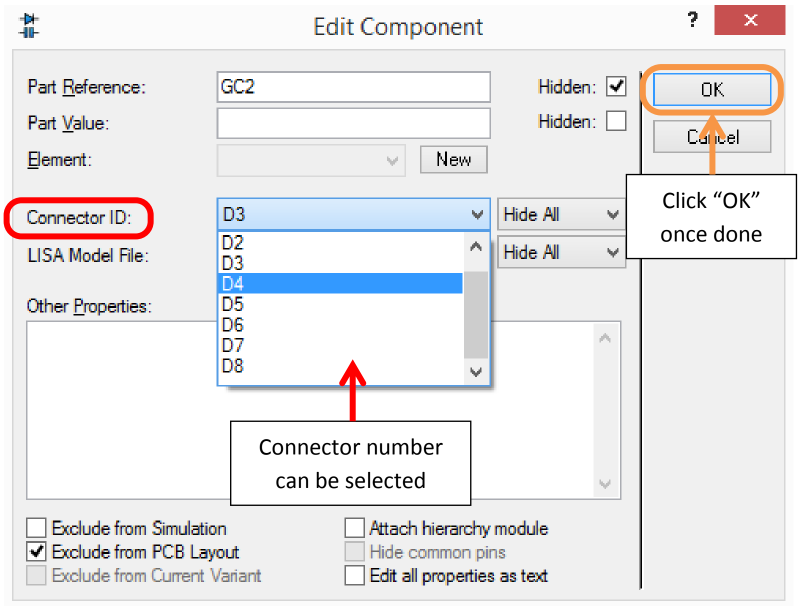


Figure 3-19: Grove Connector popup menu with "Edit Properties"

In the “Edit Component” dialog which appears, simply select a different connector number under “Connector ID”, and click “OK”.


Edit Component

Part Reference: GC2 Hidden: ✓
Part Value: Hidden: ☐
Element: New
Connector ID: D3 Hide All
LISA Model File: Hide All
Other Properties:

D2
D3
D4
D5
D6
D7
D8

☐ Exclude from Simulation
✓ Exclude from PCB Layout
☐ Exclude from Current Variant
☐ Attach hierarchy module
☐ Hide common pins
☐ Edit all properties as text

Figure 3-20: Selecting a different Grove connector number

The Grove peripheral board is now “plugged in” to a different connector, and there is no longer any “hardware conflict”!

4. Variables

Variables are a way for computers to remember things. To consider an example, let’s imagine somebody who is keeping track of stock in a shop. They have a notepad with the names of some products on it, and a place next to each product name where they can write the quantity of that product which is in stock.

Watermelons	21
Bags of Potatoes	35
Lettuces	39

To start off with, let’s imagine that they initially counted that there were 21 watermelons in stock, and wrote this down on their sheet.

Next, a customer comes in and purchases 3 watermelons. The number of watermelons in stock is now 3 less. To put this another way, the new watermelon count is equal to the old watermelon count minus 3:

```
Watermelons = Watermelons - 3
```

The new watermelon stock count is thus now 21 – 3 = 17. Imagine that the person keeping stock is writing the stock counts in pencil, so they can erase them and write new stock counts as they change. They erase the old count of 21, and write the new count of 17 instead.

Next, the store receives a delivery from the supplier of another 9 watermelons. The number of watermelons in stock is now plus 9, or put another way the new watermelon count is equal to the old watermelon count plus 9:

```
Watermelons = Watermelons + 9
```

The new watermelon stock count is thus now 17 + 9 = 26. The person keeping stock erases the old count of 17, and writes the new count of 26.

The watermelon count in the previous example is a “variable”. The name of the variable is “Watermelons” and the value of the variable is the number of watermelons in stock (starting at 21, then becoming 17, and then 26).

4.1. Displaying Text on an LCD

To illustrate the use of variables in a flowchart program, let’s start with an example:

1. Save ( ) any existing project which you may have open (if you do not want to lose it).
2. Close ( ) any existing project which you may have open. You should now see the Proteus Home Page ( ) tab, as illustrated in Figure 2-1.
3. Start a new flowchart project, using “Variables” as the project name (if you need a reminder on the steps then please see section 2.2).
4. Add a “Grove RGB LCD Module” peripheral to the project from the project gallery (“Grove” Peripheral Category). If you need a reminder on how to add a peripheral to the project, then please see section 3.4. Once added, the peripheral should be listed as “LCD1 (Grove RGB LCD)”.

"LCD" stands for "Liquid Crystal Display[9]", and is a type of electronic display which you will surely have encountered many times in everyday life.

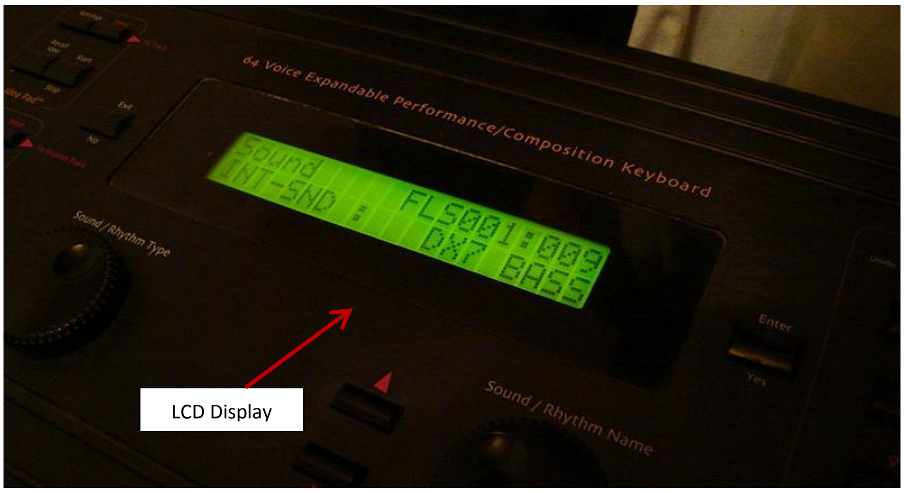

64 Voice Expandable Performance/Composition Keyboard
Sound INT-SND : FLS001:009 DX7 BASS
LCD Display
Sound / Rhythm Type
Sound / Rhythm Name
Enter
Yes

Figure 4-1: LCD display in a product[10]

The Grove RGB LCD module has two lines (rows) of text, and up to 16 characters can be displayed on each row.

To start constructing our flowchart, drag the "print" method from the Grove RGB LCD into the flowchart "**Setup**" routine.

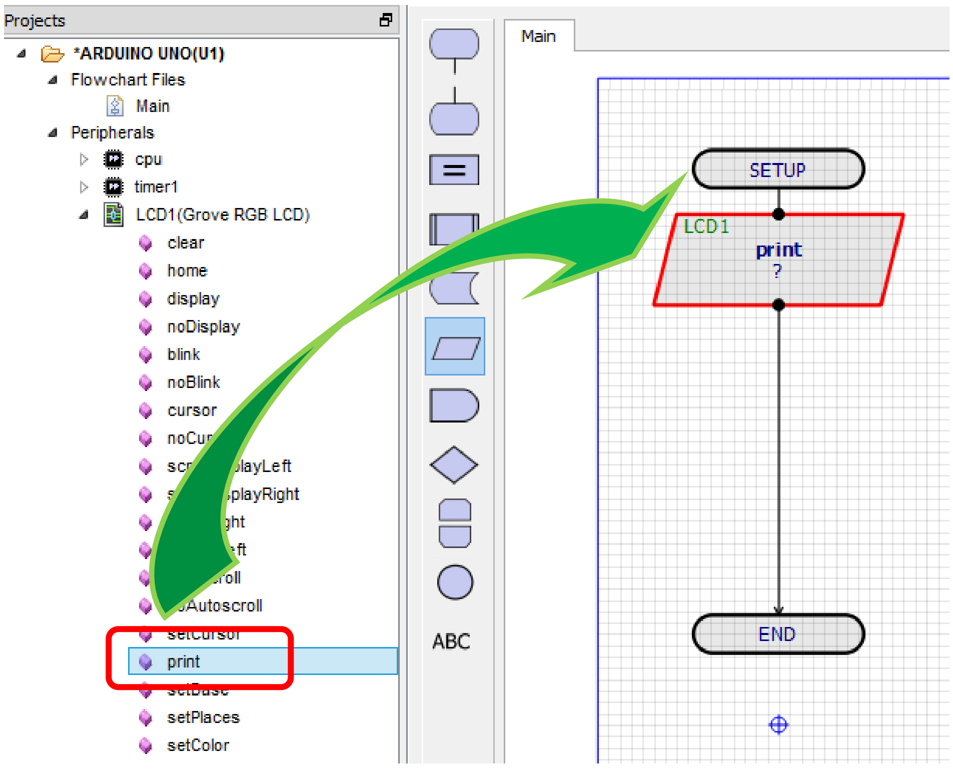


Figure 4-2: LCD "print" method

[9] If you are curious, please see https://en.wikipedia.org/wiki/Liquid-crystal_display for more information.
[10] Product with LCD image from https://commons.wikimedia.org/wiki/File:Ensoniq_MR-61_(LCD).jpg, Attribution-Share Alike 2.0 Generic license, unmodified except for labels added.

The “print” method is used to display text or numbers on the LCD.

To specify the text which we would like to display on the LCD, right click on the “print” flowchart block and then click “Edit” (or alternatively left-click twice on the block). The following dialog is displayed:

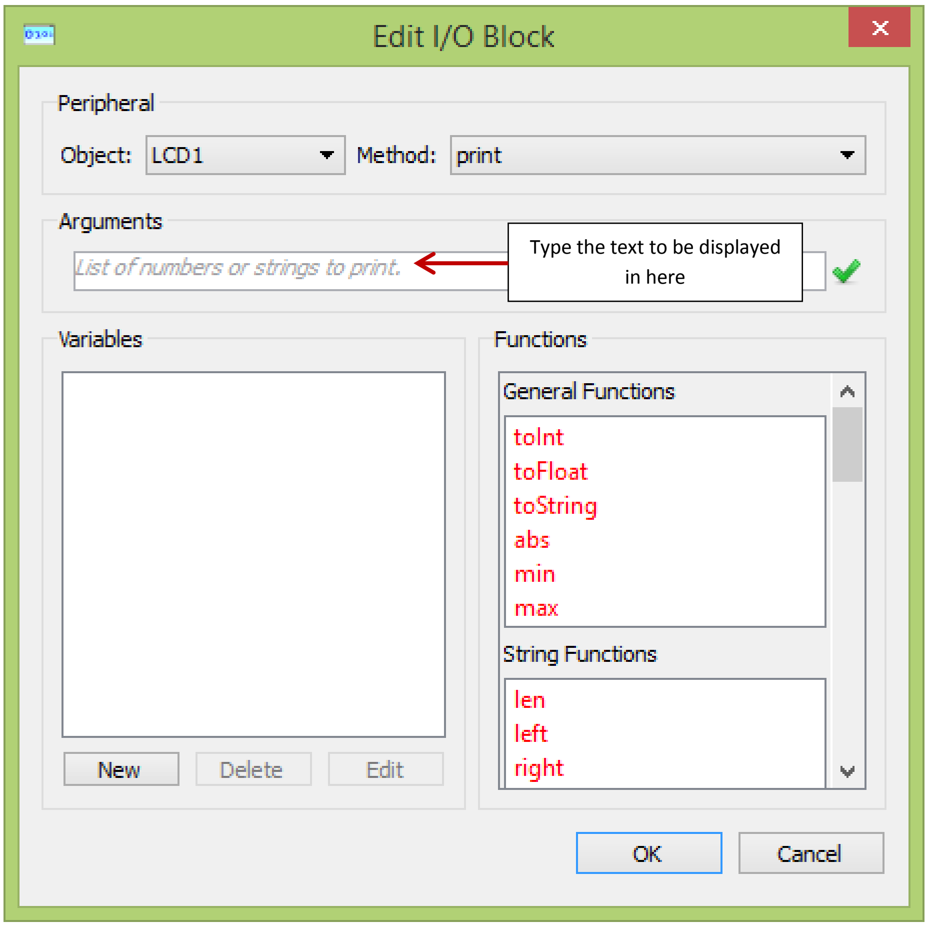


Figure 4-3: "print" method dialog box

The text to be displayed on the LCD should be entered under “Arguments”, where it says “*List of numbers or strings to print*”. Enter the text "Hello World!" (including the quotes):

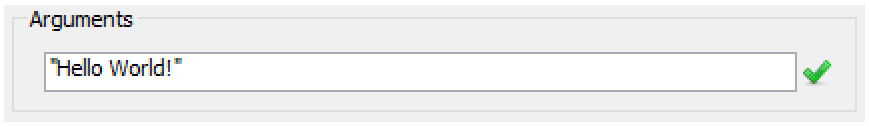


Figure 4-4: "Hello World!" text entered

In programming, text should always be enclosed in quotes (" "). The reason for this will be discussed in more detail shortly (section 4.5), but for now it is sufficient to know that this is the rule.

When done, click “OK”. Your flowchart should now look like the following figure:

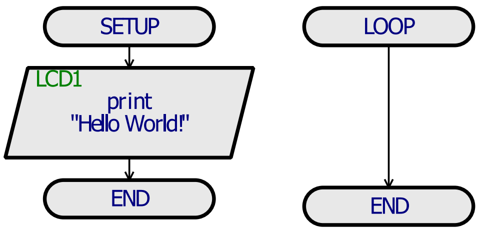


Figure 4-5: "Hello World!" flowchart

Run the simulation ("Play" button), and you should see the text "Hello World!" displayed on the LCD.

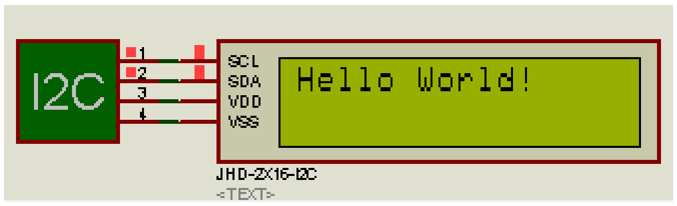
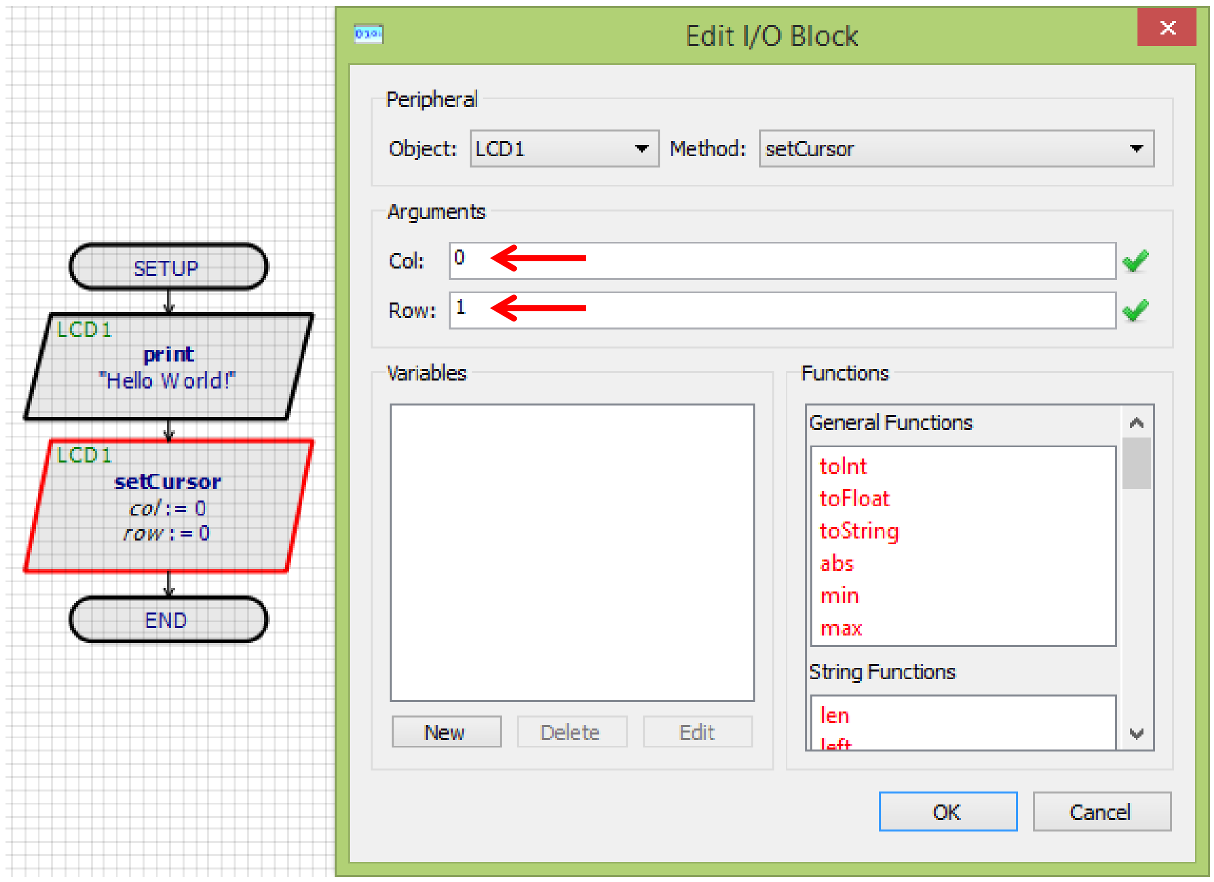


Figure 4-6: "Hello World!" displayed on LCD

Great! That was very easy to do, wasn't it?

For the next part of the project, we will display the text "Count:" on the bottom line of the LCD, and display a number next to the text. The number will increment (increase by 1, i.e. count) every second.

To display text on the bottom line of the LCD (underneath our "Hello World!" text), we need a way to tell the LCD where we want the text displayed. This is accomplished using the "setCursor" method of the Grove RGB LCD peripheral. Drag the setCursor method into the flowchart **Setup** routine, placing it after the "print" block. Next, right-click on the newly added setCursor block and select "Edit" from the popup menu (or left-click twice on the block). Change the value of the "Row" argument to 1.



Object: LCD1
Method: setCursor
Col: 0
Row: 1

Figure 4-7: setCursor method dialog box

Note that the row and column ("col") values are "zero-based". This means that the top row is "row 0" and the bottom row is "row 1"; the leftmost column is "column 0" and the rightmost column is "column 15".

Columns

Rows	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	H	e	l	l	o		W	o	r	l	d	!				
1																

In order to display text at the start of the second row on the LCD, we need to specify row 1 and column 0, as depicted in the previous figure (Figure 4-7). Once done, click "OK".

You may be wondering why we didn't need to do this when we displayed the "Hello World!" text originally; the answer is that the LCD is set to start displaying text at the top-left by default ("home" position).

Now that we've told the LCD where we want to display text next (using the setCursor method), we can proceed to telling the LCD what we'd like to print there. Drag another "print" method into the flowchart, and insert it after the setCursor block. Edit the properties for the new cursor block (please see the steps for this described earlier in this chapter if you need a reminder), and specify "Count:" (including the quotations) as the text to be displayed. Your flowchart should now look like the following figure:

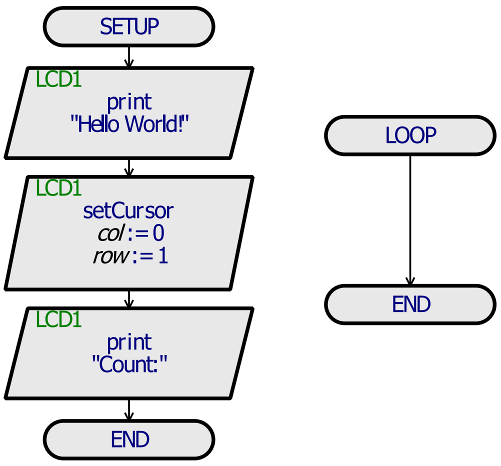


Figure 4-8: Flowchart with "Count:" added

Run the simulation, and you should see that "Count:" is now displayed on the bottom line of the LCD (in addition to the "Hello World!" text on the top line).

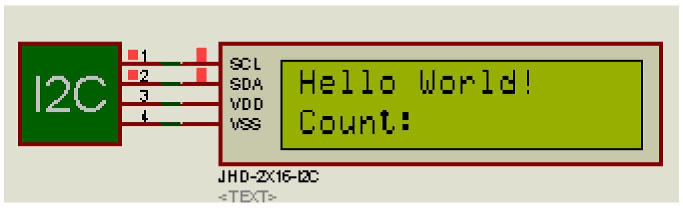


Figure 4-9: LCD with "Count:" added

4.2. Adding a Variable to the Project

The next step in our project is to add the number next to the "Count:" text, which will increment (increase by 1) every second.

Referring back to the example of someone keeping track of stock in a shop (from the start of this chapter), we now need to tell the computer to make a space on its notepad where it will remember the value of the number which we will be incrementing. We need to give that number a name, and should also set an initial (starting) value for it. This is similar to how the person keeping stock had the name of the product on the notepad (e.g. "Watermelons), and wrote down the stock count next to it (21 initially). The term for this number which can be identified by a name, is a **variable**.

To add a new variable to our project, drag an Assignment Block (=) into the flowchart, placing it after the second "print" block (Figure 4-10).

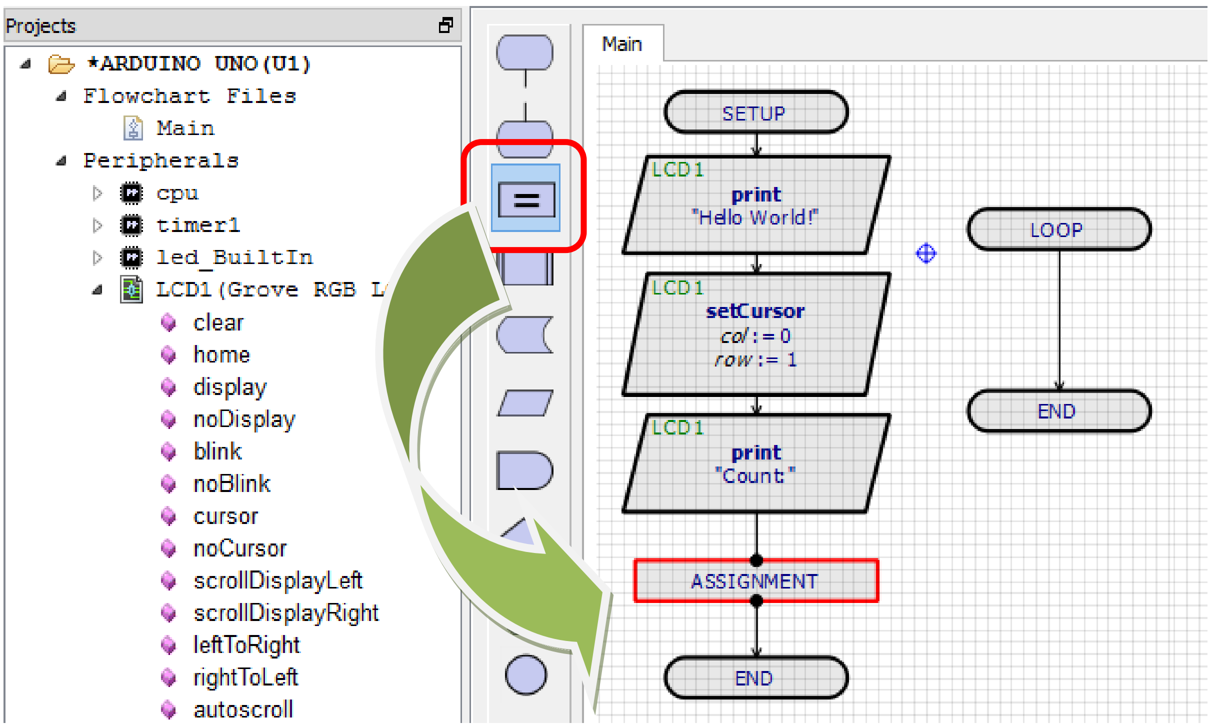


Figure 4-10: Assignment block insertion

Next, edit the properties of the newly-inserted assignment block (right-click on the assignment block and then select "Edit", or left-click twice on the assignment block).

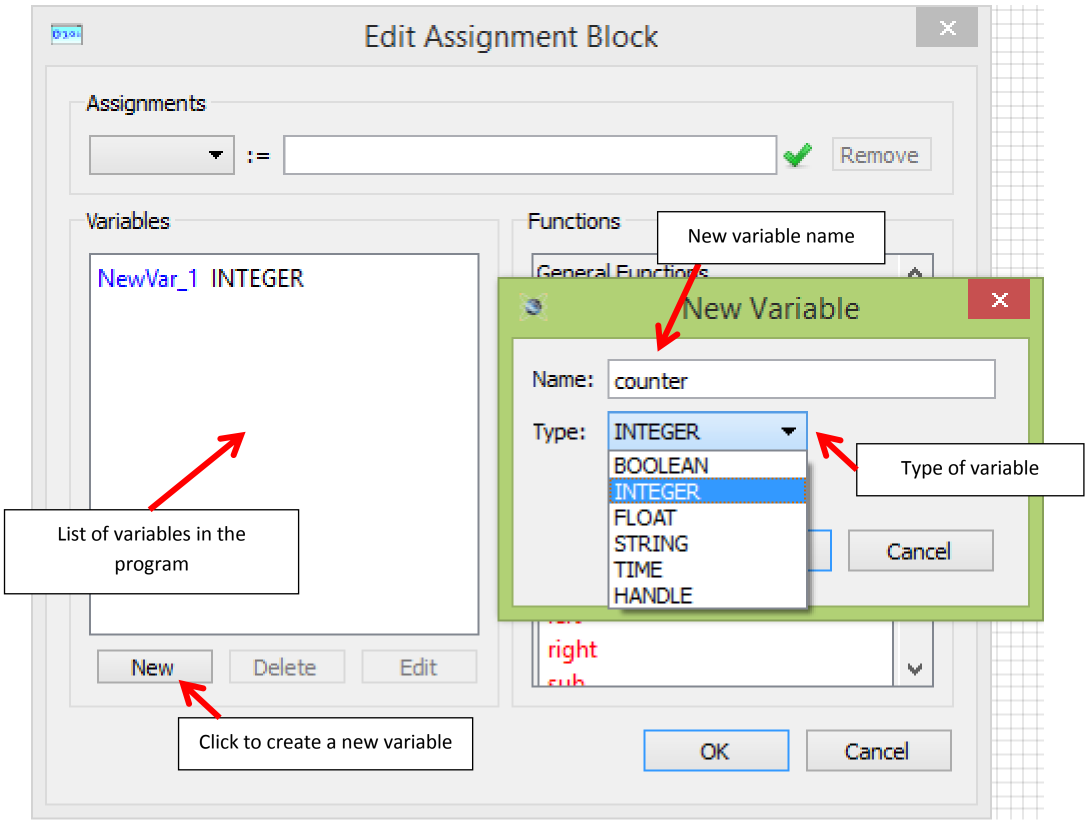


Figure 4-11: Adding a new variable

At the bottom-left of the dialog ("Variables" area), click "New" to create a new variable. Enter "counter" as the name for the variable (**excluding** the quotes[11]), as depicted in the previous figure (Figure 4-11). Leave the type as INTEGER (variable types will be discussed shortly), and click "OK".

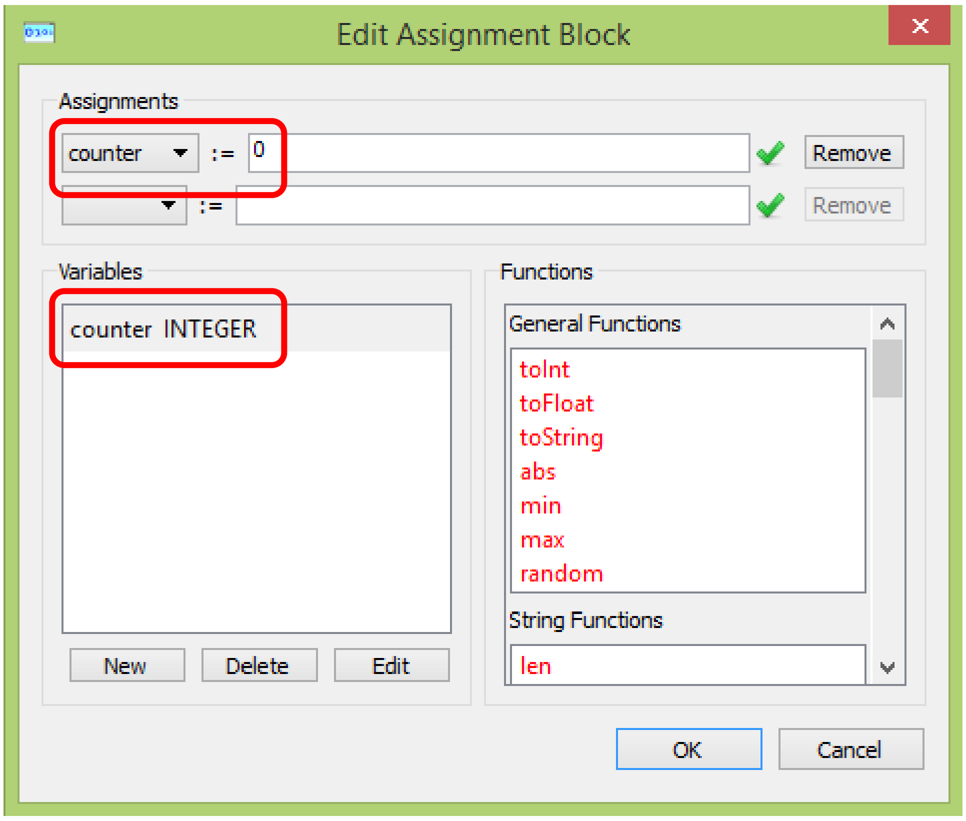


Figure 4-12: Assigning a value to the variable

Congratulations! You have now created your new variable called "counter", which should appear under the list of variables.

Under "Assignments", select the "counter" variable from the dropdown, and type 0 into the box to the right of the dropdown. This means that we are going to set "counter" to a value of 0 in this assignment block. When done, click "OK". Your flowchart should now look like the following figure:

[11] The reason for using quotes or not using quotes is discussed in section 4.5.

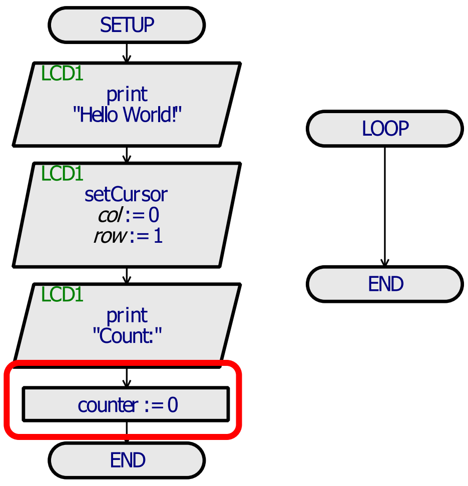
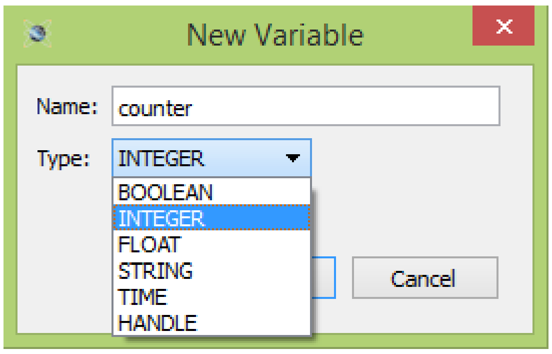


Figure 4-13: Flowchart with variable assignment

4.3. Types of Variables

In the previous section, when creating a new variable, you may have noticed that there was an option to choose the variable "type".



Name: counter

Type: INTEGER

Figure 4-14: New Variable dialog

Variable "type" refers to the "type" of information which can be stored in that variable. The possible variable types in Visual Designer are listed below:

Variable type	Information which can be stored
Boolean	"Yes" or "No" ("True" or "False").
Integer	Numbers (including negative numbers). E.g. 1, 2, 3, 10, 5000, -5, -123, etc.
Float	Numbers with decimal places (including negative numbers). E.g. 3.14, -5.6, etc.
String	Text (a "string" of characters). E.g. "Hello World!"
Time	A date and time. For example, 11:07:15 05/08/1983.
Handle	(Advanced topic. A handle to a resource, such as a file on an SD Card.)

The reason for having different variable types stems from the way in which the computer stores them internally, as well as what operations can be performed with them. These topics are discussed briefly below, however from a usability point of view you can **simply choose the type of variable which best suits the type of information which you would like to store in it**. Ask yourself what you would like to store in the variable: Is it a number? Is it text? Can it only be either "Yes" or "No"?

4.3.1. Possible Operations for Different Variable Types

You will surely agree that the mathematical operation 5 + 5 makes sense – the answer is obviously 10. What if somebody asked you to calculate 5 + "Hello" however? That just wouldn't make sense. In the same way, the type of variable determines the sort of operations which can be performed with it. Numbers can be added, subtracted, multiplied, divided, etc. Text can be joined together ("Hello " + "World!" = "Hello World!").

Note that it can also be possible to have a number within text – for example "Buy 5 apples". The computer considers this to be just text however, and treats the "5" as a sort of a letter rather than a number. So if two numbers with the value of 5 are added, then the answer is obviously 10. If two strings containing the *text* "5" are joined together however, then the answer is "55". Note that whilst the symbol "+" can be used in some programming languages to join text together, this is not a *mathematical* operation, and none of the other "mathematical" symbols (multiply, divide, etc) can be applied to text (once again, this just wouldn't make sense – what is "hello" multiplied by "cat"?).

4.3.2. Variable Storage within the Computer

Please note that this section has been inserted just for the curious, and you can safely skip to section 4.4 if you find this section confusing.

It is entirely unnecessary, for the purposes of flowchart design, to understand how the computer stores variables internally. By way of a short explanation as to another reason for having different types of variables however, consider the following:

Imagine you had to fill out one of those forms which have a different block for each letter:

Name:																				

Date of Birth:			

Note: Insert only one letter or number per block

You may be able to fill your name in within the number of blocks provided, but there is no way that you could fill your date of birth in with just 3 blocks! The form should rather have 10 blocks:

Date of Birth:			/			/				

This way you could fill in a full date, such as 05/08/1983.

In much the same way, the computer only has a certain number of blocks available depending on the variable type. A Boolean variable can only be 1 or 0 (interpreted as "Yes" or "No" / "True" or "False") – you cannot store the number 972 in it, because there just isn't enough space in it.

As mentioned however, so long as you **simply choose the right type of variable for the type of information which you would like to store**, you needn't worry about how much space the computer has available to store it for practical purposes – this section has just been inserted for the curious.

4.4. Using Variables in a Project

Now that we have added the "counter" variable to our project, we can proceed to using it in the program.

We essentially want to perform the following steps in our program:

1. Increment the value of the number stored in the counter variable (increase it by 1).
2. Tell the LCD where we would like to display something next (on which row and column, “setCursor” method.).
3. Tell the LCD to display the value of the counter variable.
4. Wait 1 second.
   (Repeat)

This is represented by the following flowchart:

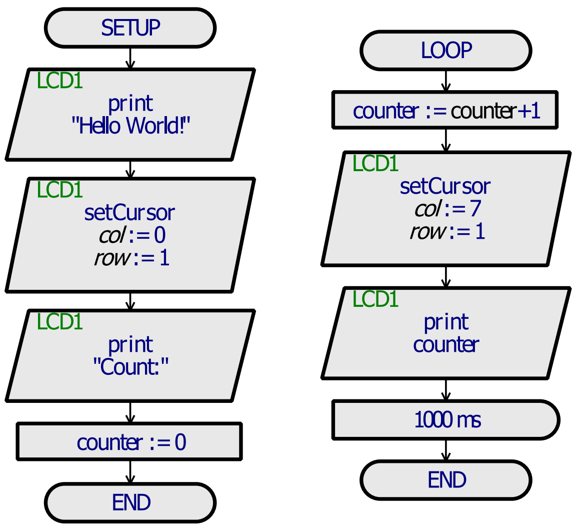


Figure 4-15: Displaying an incrementing a number on an LCD

4.4.1. Mathematical Operations using a Variable

The first step is to increment the counter (increase it by 1, or add 1 to it).

Mathematical operations on variables (such as incrementing them) are performed in assignment blocks, so drag an assignment block into the **Loop** routine (for a reminder on how to do this, please see Figure 4-10 – except note that this time we are placing the assignment block in the Loop routine rather than the Setup routine).

Next, edit the assignment block (please see section 4.2 if you need a reminder). Under “Assignments”, select the counter variable from the dropdown and type the text "counter+1" (without quotations[12]) alongside (as illustrated by the following figure).

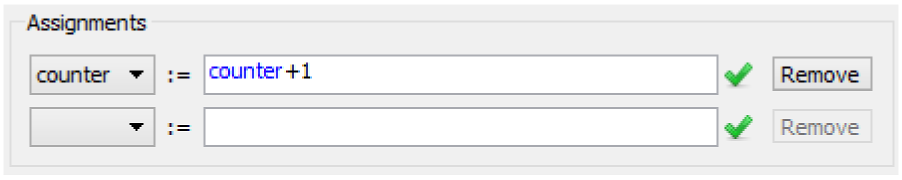
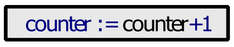


Figure 4-16: Incrementing a variable

Click “OK” once done. The full statement now reads counter := counter + 1 (). When this instruction is executed during flowchart operation, it takes the old value of the counter, adds 1 to it, and stores the new value back into the counter variable. So if counter was 0, the

[12] The reason for using quotes or not using quotes is discussed in section 4.5.

computer adds 1 to it; 0 + 1 equals 1, so the computer stores the new value of 1 back in the counter variable. The next time the statement is executed the counter is currently 1, so the calculation becomes 1 + 1 which equals 2. The next time the calculation becomes 2 + 1 which equals 3, and so forth. Many other mathematical operations besides for + 1 can be constructed – for example counter := counter - 1, counter := counter + 5 or counter := counter * 3 (multiplied by 3) – a summary of the mathematical operators which can be used is provided in section 4.6.

4.4.2. Displaying the Value of a Variable on an LCD

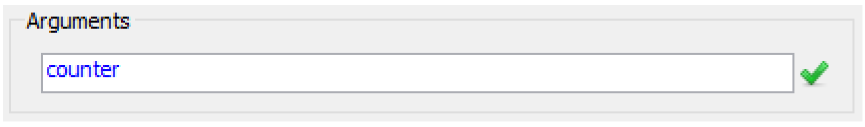
The next steps after incrementing the counter is to tell the LCD where we would like to display text next ("setCursor" method), and then display the value of the counter variable at that location. At this stage in the program, we are already displaying the following on the LCD:

Columns

Rows		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	H	e	l	l	o		W	o	r	l	d	!				
	1	C	o	u	n	t	:										

A good place to display the value of our counter variable at is therefore going to be row 1 and column 7 (highlighted in red). To tell the LCD that we would like to display text at row 1 and column 7 next, drag a setCursor method into the flowchart, inserting it after the assignment block which was added in the previous step. Then, edit the setCursor block and set the row to 1 and column to 7. If you need a reminder on how to add a setCursor method to the flowchart and edit its properties, then please see the latter part of section 4.1.

Next, drag a "print" method into the flowchart, inserting it after the setCursor method which was inserted in the previous step. Edit the print block and enter "counter" (<u>excluding</u> the quotes) under "Arguments", as illustrated in the following figure:


Arguments

counter

Figure 4-17: Entering a variable name as an argument

You may be wondering why we sometimes enter text under the print methods arguments with quotes, and sometimes without – this will be discussed in a following section (section 4.5), but the short answer for now is that variable names are entered without quotes.

4.4.3. Completing the Flowchart which Displays an Incrementing Variable on the LCD

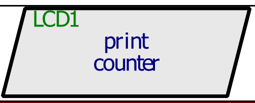
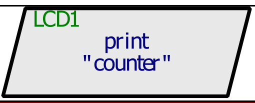
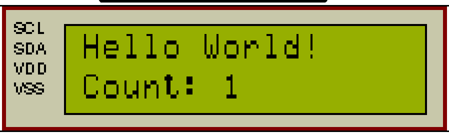
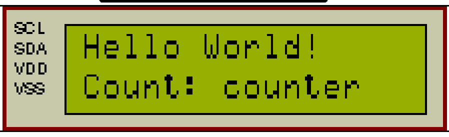
The final step is to wait 1 second before repeating the process (otherwise it would happen so fast that we wouldn't be able to see anything!). Drag a delay block () into the flowchart after the print block, and set the delay time to 1000ms (by editing the delay block).

Your flowchart should now look like the one in Figure 4-15. If it does not, then please go through go through the steps again carefully.

Once ready, run the simulation. There should now be a number displayed next to "Count:", which increments each second!

4.5. Using Text in Arguments

You may have noticed that when we wanted to display text on the LCD then we placed it in quotes (e.g. "Counter:"), but when we wanted to display the value of a variable we entered the variable name without quotes (just counter). The reason for this can be explained by the following problem: What if we wanted to display the actual *text* "counter" on the display, rather than the value of the variable called "counter"? This is a problem which the designers of computer programming languages faced, and the solution was to place *text* within quotes (" "). The quotes do not form part of the interpreted text – in this case they are not displayed on the LCD – they simply help the compiler to understand whether we mean the *variable* counter, or the *text* counter.

Variable	Text
	
	

4.6. Mathematical Operations Using Variables

As we saw when incrementing the counter variable (counter := counter + 1), it is possible to perform mathematical calculations using variables. The following mathematical operations can be performed:

Symbol (Operator)	Operation
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus[13] (remainder after division)

Note that mathematical operations can become much more complex than simply incrementing a variable, and brackets can be used within equations. For example, to calculate the area of a triangle, using variables "area", "base" and "height", the following equation could be used: area = (base * height) / 2 ("area equals base times height, divided by two").

4.7. Using Variables in Conditions

Variables can also be used when asking questions (such as in decision blocks). For the next part of our project, we will build on our flowchart so that the text "More than 10!" is displayed on the top line of the LCD when the value of the "counter" variable becomes greater than 10. The flowchart needed to accomplish this is detailed in the following figure:

[13] Note that "Modulus" (remainder after division) has nothing to do with "Percentage", although its operator in C is a percentage symbol. Some examples of Modulus are:
7 divided by 2 is 3½ (three and one half) – thus 7 modulus 2 is 1, because 2 fits fully into 7 up to 3 times, leaving 1 remaining to actually reach 7. I.e. 7 % 2 = 1.
31 divided by 4 is 7¾ (seven and three quarters) – thus 31 modulus 4 is 3, because 4 fits full into 31 up to 7 times, with 3 remaining to reach 31. I.e. 31 % 4 = 3.
Somre more examples of modulus are:
8 % 4 = 0, 11 % 3 = 2, 12 % 3 = 0, 13 % 3 = 1, 14 % 3 = 2, 8 % 7 = 1

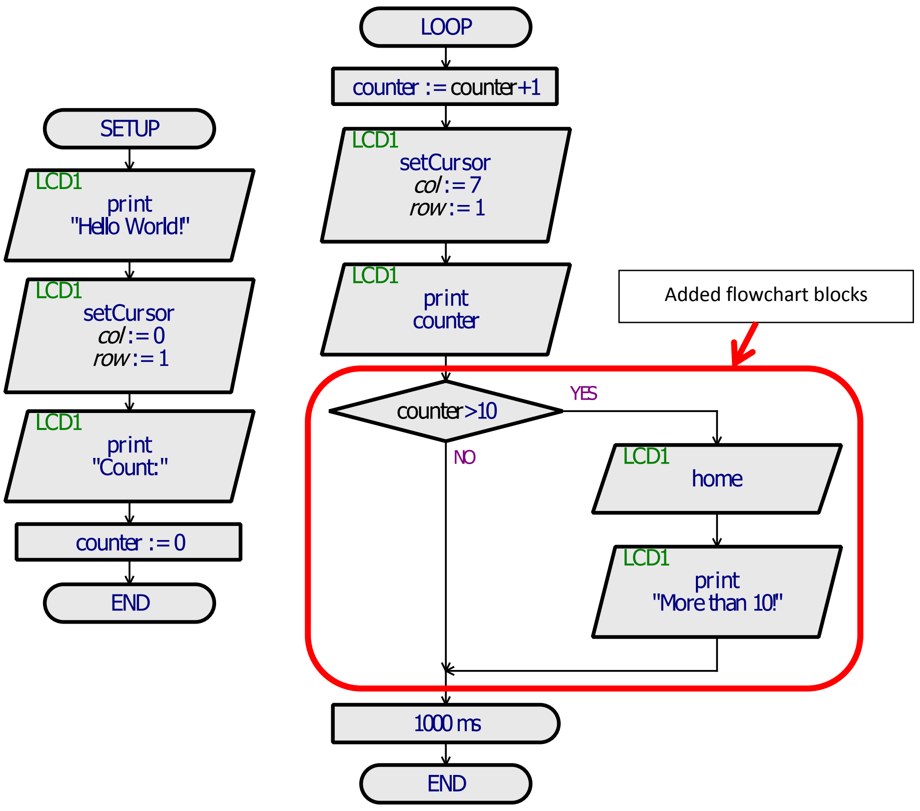


Figure 4-18: Flowchart using a variable in a decision

The following flowchart blocks have been added (highlighted in Figure 4-18):

1. A decision block which checks whether the value of the counter variable is greater than 10.
2. The LCD “home” method, which is a shortcut for (equivalent to) the setCursor method with row and column values of 0.
3. An LCD “print” method to display the text “More than 10!” on the LCD.

In order to update your flowchart so that it matches the one in Figure 4-18, start by adding a decision block into the flowchart, above the 1000ms delay block:

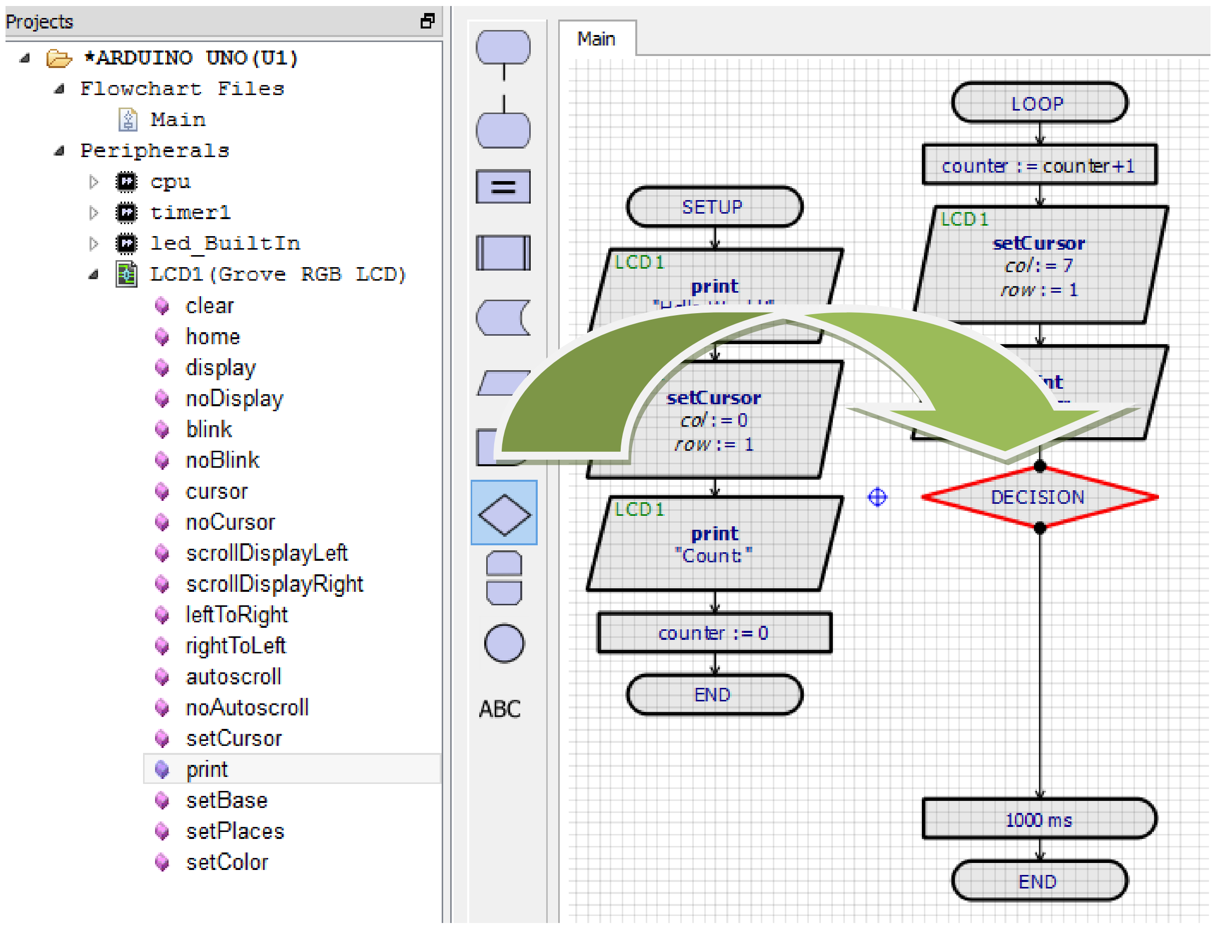
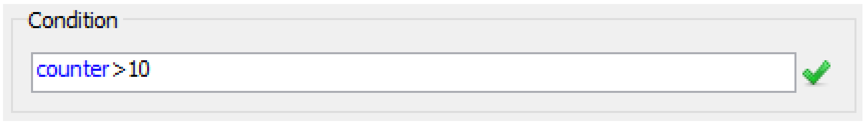


Figure 4-19: Inserting a decision block

Right-click on the decision block and select "Edit" from the popup menu (or left-click on the decision block twice), in order to bring up the "Edit Decision Block" dialog. Enter "counter>10" (without the quotations) under the "Condition" section of the dialog (as illustrated by the following figure).


Condition

counter>10

Figure 4-20: Inserting the condition in the Edit Decision Block dialog

Once completed click "OK". Next add the LCD "home" and "print" methods to the flowchart, so that it matches the one in Figure 4-18. The LCD "print" block should be edited and the text "More than 10!" inserted under "Arguments" (for a reminder on how to do this, please see section 4.1). Swap the "Yes" and "No" on the decision block so that "Yes" it matches Figure 4-18 (right click on the decision block and then select "Swap Yes/No" from the popup menu).

Once ready, run the simulation. Once the counter reaches the number 11, the text "More than 10!" is displayed on the top line of the LCD. (If the simulation does not perform as expected, then please check that your flowchart exactly matches the one from Figure 4-18.)

4.8. Relational Operators

Relational operators are used in conditions (such as in the decision block), to ask a question which will have an answer of “Yes” or “No”. In the example, we asked the question “Is the value of counter greater than 10?” (“counter > 10”). Other relational operators besides for “greater than” are also available however.

Symbol (Operator)	Operation (Question)
=	Equal to?
!=	Not equal to?
>	Greater than?
<	Smaller than?
>=	Greater than or equal to?
<=	Smaller than or equal to?

4.9. Logical Operators

Logical operators can be used in conjunction with relational operators to ask more complex questions, and include the operators AND, OR and NOT. For example, “If the kettle is boiled AND there is a mug on the counter, then pour the hot water into the mug”. Brackets can also be used when creating more complex conditions, in much the same way as they can be used with mathematical operators. The result achieved by using logical operators can also be achieved by using multiple decision blocks, so **do not worry if you do not understand them**. We will not be using them in this book, and they are mentioned only for the sake of being thorough.

Symbol (Operator)	Operation (Question)
&&	And
||	Or
!	Not

For example, (distance > 5 && distance < 10) || (weight > 15) (“distance greater than 5 and distance smaller than 10, or weight greater than 15?”).

4.10. Final Project Step

For the final project step, we will be adding a button which sets the value of the counter variable to 20 whenever it is pressed. The final flowchart, after adding this feature, is detailed in the following figure.

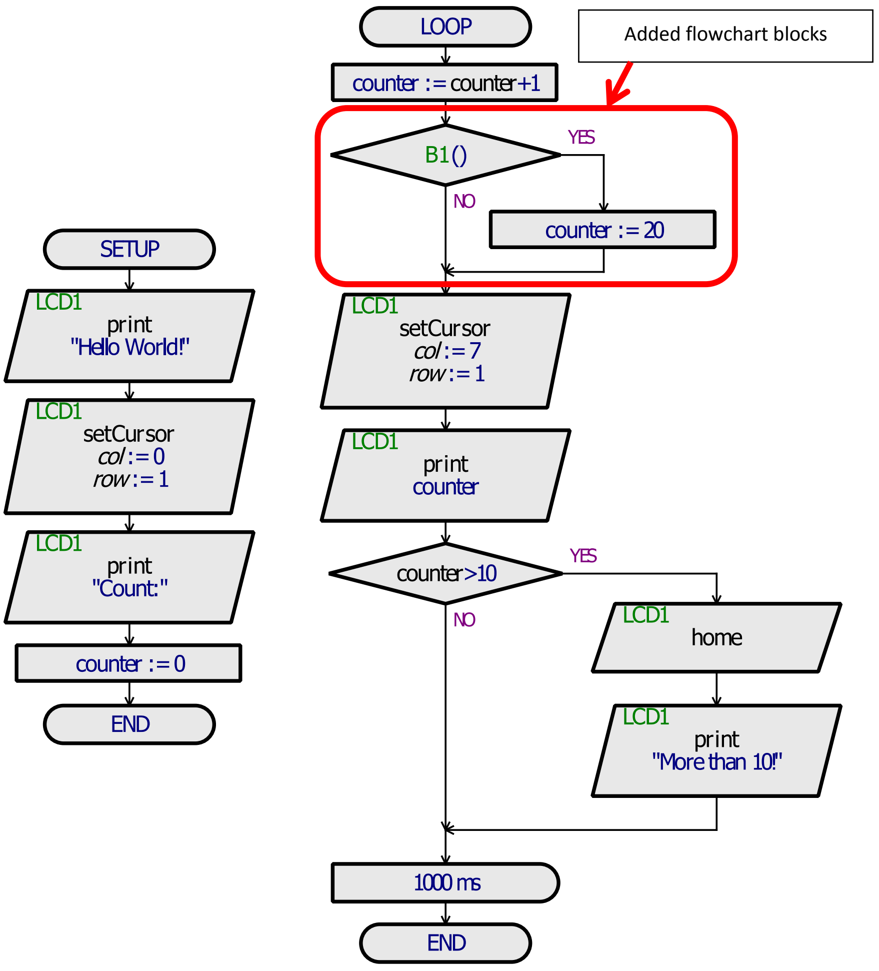


Figure 4-21: Final "variable" project flowchart

In order to update your flowchart so that it matches Figure 4-21, please complete the following steps:

1. Add a “Momentary Action Push Button” (from the “Grove” Peripheral Category) to the project (using the Peripheral Gallery). If you need a reminder on how to add a peripheral to the project, then please see section 3.4.
2. Drag the button (“B1”) onto the flowchart, so that it inserts a decision block (as depicted in Figure 4-21). Please see section 3.5 if you need a reminder on how to drag a button onto the flowchart.
3. Drag an assignment block onto the flowchart, placing and connecting it as depicted in Figure 4-21.
4. Edit the assignment block so that the assignment is “counter := 20”.
5. Swap the “Yes” and “No” on the decision block, so that it matches Figure 4-21.

Once ready, run the simulation. Press and hold the newly added Grove Button (do not confuse the Grove button with the Reset button), and you should see that the value of the counter jumps to 20. It is necessary to hold the button, as the button is only checked at the top of the loop routine, and not during the 1 second delay at the bottom. Thus, if you press and release the button during the 1 second delay, the program will not “see” that the button was pressed. There are techniques which can be used to work around this and get the button press to register regardless of how short it is, however they are beyond the scope of this chapter. If the simulation does not behave as expected, then please carefully compare your flowchart to Figure 4-21 to make sure that they are exactly the same, and ensure that you are pressing the Grove button and not the Reset button (which is discussed in the next section).

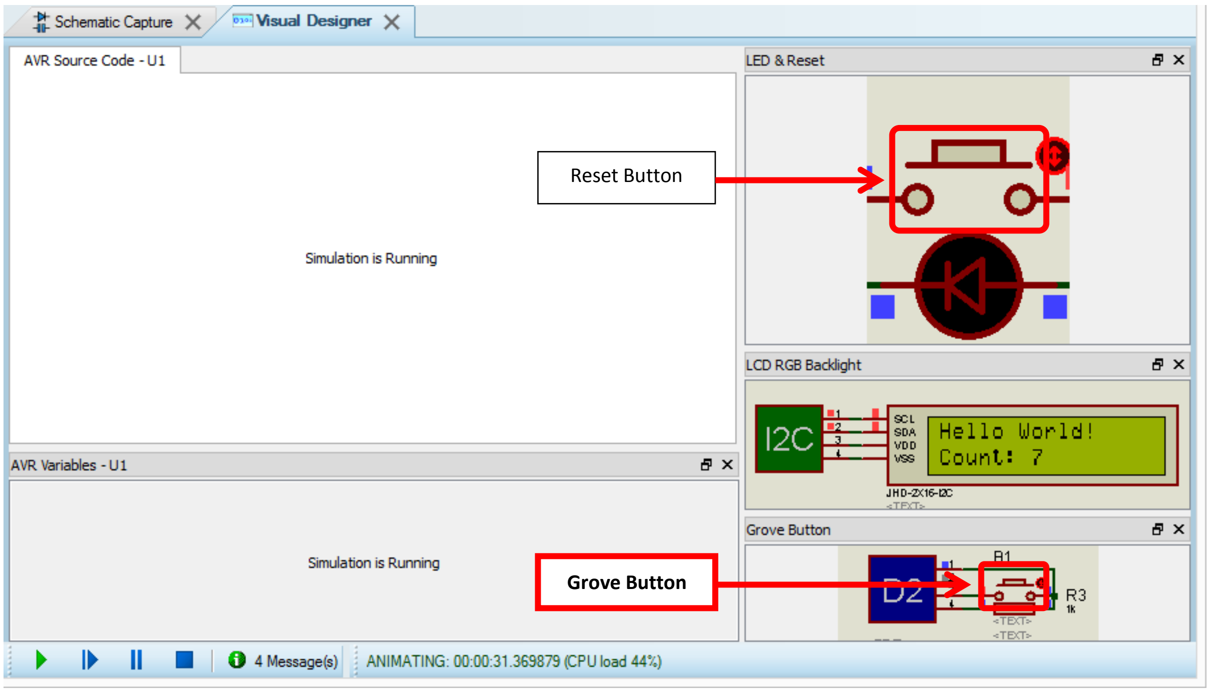


Figure 4-22: Grove button vs. Reset button

4.11. The Reset Button

The Reset button (highlighted in Figure 2-27, Figure 2-28 and Figure 4-22) "restarts" the microcontroller. If you press it whilst the simulation is running then you should notice that the counter starts again from 1; this is because your program has started over from the beginning again (running the Setup routine once, and then moving onto the repeating Loop routine). Be careful not to confuse the Reset button with any other button(s) which you may be using in your project.

4.12. Sleep on it

Variables are a big new concept. Besides for that, we have also introduced LCD displays in this chapter. Unless you have some previous programming experience, it may take a while to "sink in". You may want to sleep on it, and then re-read this chapter to go over all the new concepts again and make sure that they are well understood.

5. Basic Electric Circuit Theory and Simulation

Whilst the focus of this book is largely on software design, it is inevitably necessary to have at least some knowledge of electrical principles in order to be able to build electronic machines. Anybody living in a modern society has come into contact with, and has at least some knowledge or understanding of electricity, due to everyday encounters with it (switches, batteries, electronic devices such as phones, etc). This section however goes over some basic concepts to ensure that the reader is familiar with them.

As you by now know, you can design and simulate microcontroller software flowcharts in Proteus. Proteus also however has the ability to simulate electrical circuits. This makes Proteus a great tool for learning about and experimenting with electrical circuits (as well as electronic components) in general. Proteus also includes a number of sample projects which demonstrate basic electrical principles. In order to ensure familiarity with some basic electrical concepts, we will use a few of the sample projects to demonstrate these principles.

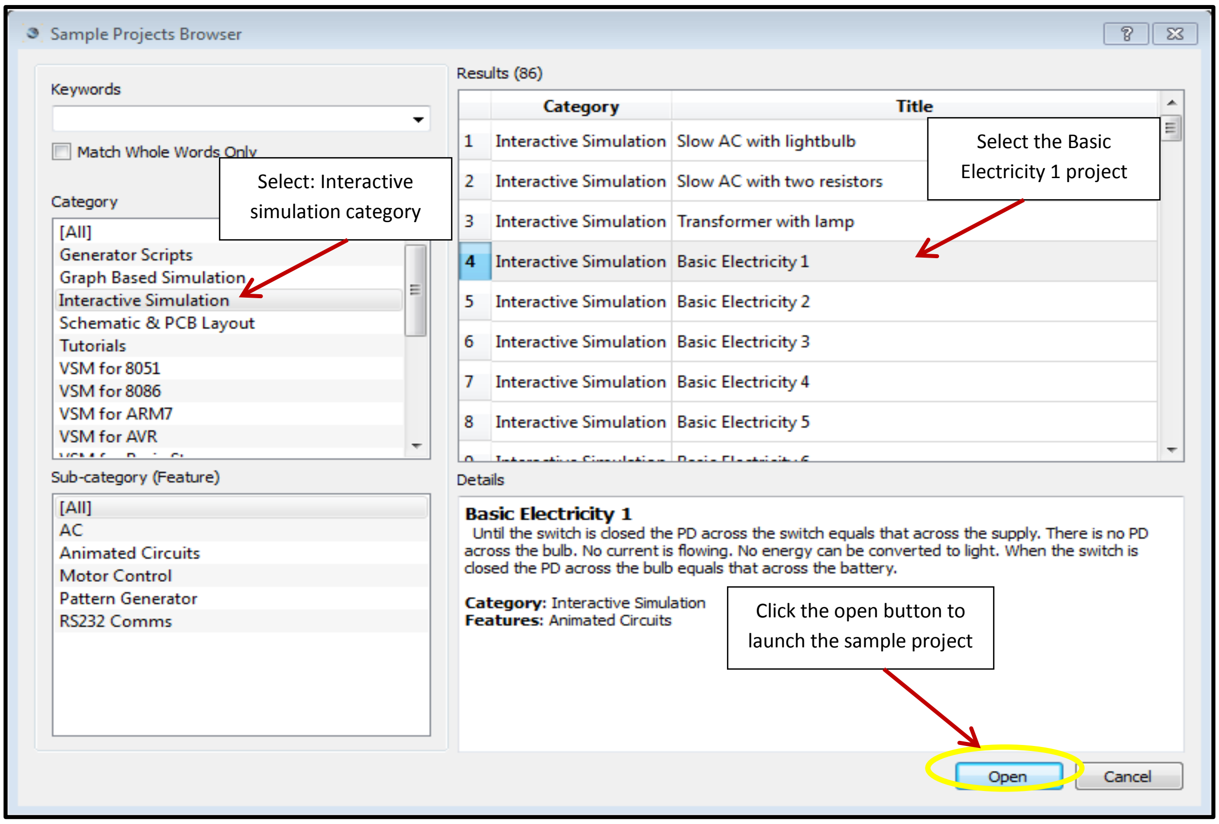
5.1. The Schematic Capture Environment

Up until now we have worked in the Visual Designer (flowchart designer) tab of Proteus, but for this chapter (as well as some of the following chapters) we will be using the Schematic Capture tab.

The easiest way to gain some familiarity with the Schematic Capture tab is to open a sample project. Let’s start by opening the “Basic Electricity 1” sample project:

1. Save ( ) any existing project which you may have open (if you do not want to lose it). This can be done using the “Save Project” command available from the File menu.
2. Click on the File menu and then select **“Open Sample Project”**, or alternatively click on the **“Open Sample”** button on the Proteus Home Screen ( ). The Sample Projects Browser window should open (see Figure 5-1).
3. Select the **“Interactive Simulation”** category.
4. Select the **“Interactive Simulation: Basic Electricity 1”** project.
5. Click **“Open”** button to launch sample project.

The following figure illustrates the procedure outlined above for opening the “Interactive Simulation: Basic Electricity 1” sample project.


Sample Projects Browser
Select: Interactive simulation category
Select the Basic Electricity 1 project
Click the open button to launch the sample project
Open
Cancel

Keywords

Match Whole Words Only

Category

[All]
Generator Scripts
Graph Based Simulation
Interactive Simulation
Schematic & PCB Layout
Tutorials
VSM for 8051
VSM for 8086
VSM for ARM7
VSM for AVR

Sub-category (Feature)

[All]
AC
Animated Circuits
Motor Control
Pattern Generator
RS232 Comms

Results (86)

	Category	Title
1	Interactive Simulation	Slow AC with lightbulb
2	Interactive Simulation	Slow AC with two resistors
3	Interactive Simulation	Transformer with lamp
4	Interactive Simulation	Basic Electricity 1
5	Interactive Simulation	Basic Electricity 2
6	Interactive Simulation	Basic Electricity 3
7	Interactive Simulation	Basic Electricity 4
8	Interactive Simulation	Basic Electricity 5

Details

Basic Electricity 1
Until the switch is closed the PD across the switch equals that across the supply. There is no PD across the bulb. No current is flowing. No energy can be converted to light. When the switch is closed the PD across the bulb equals that across the battery.

Category: Interactive Simulation
Features: Animated Circuits

Figure 5-1: Procedure for opening the Interactive simulation - Basic Electricity 1 sample project

Once the project opens, it should look like the following figure.

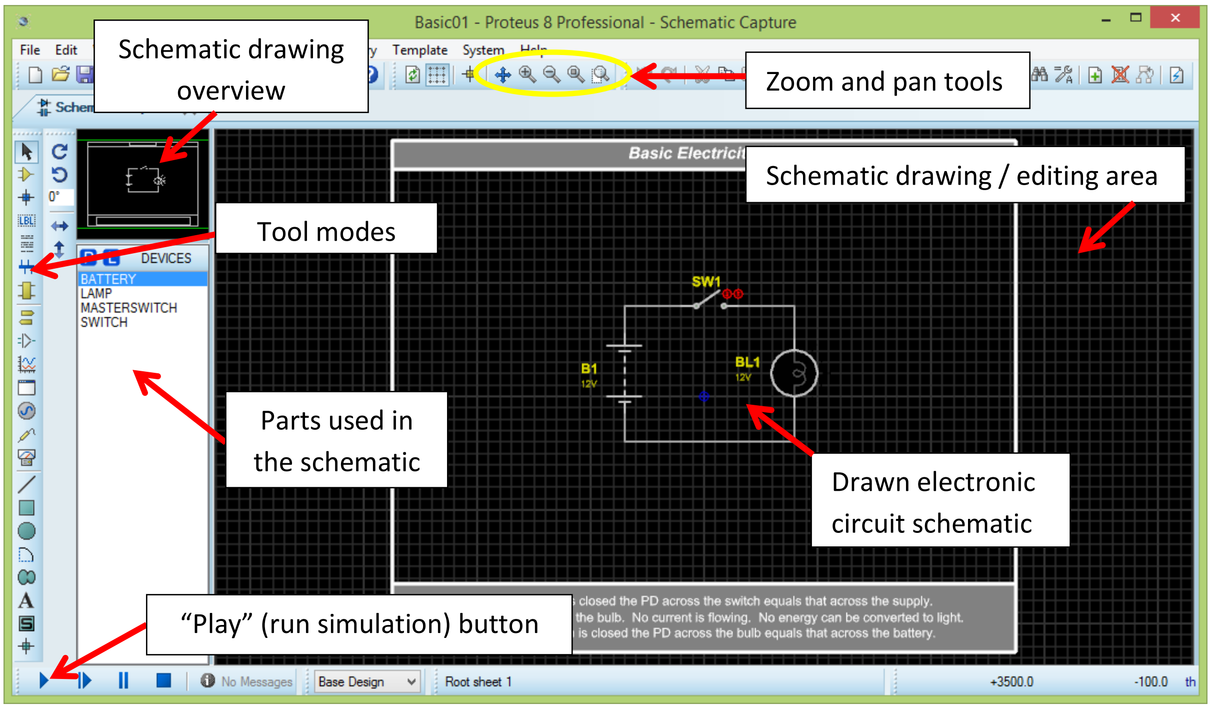
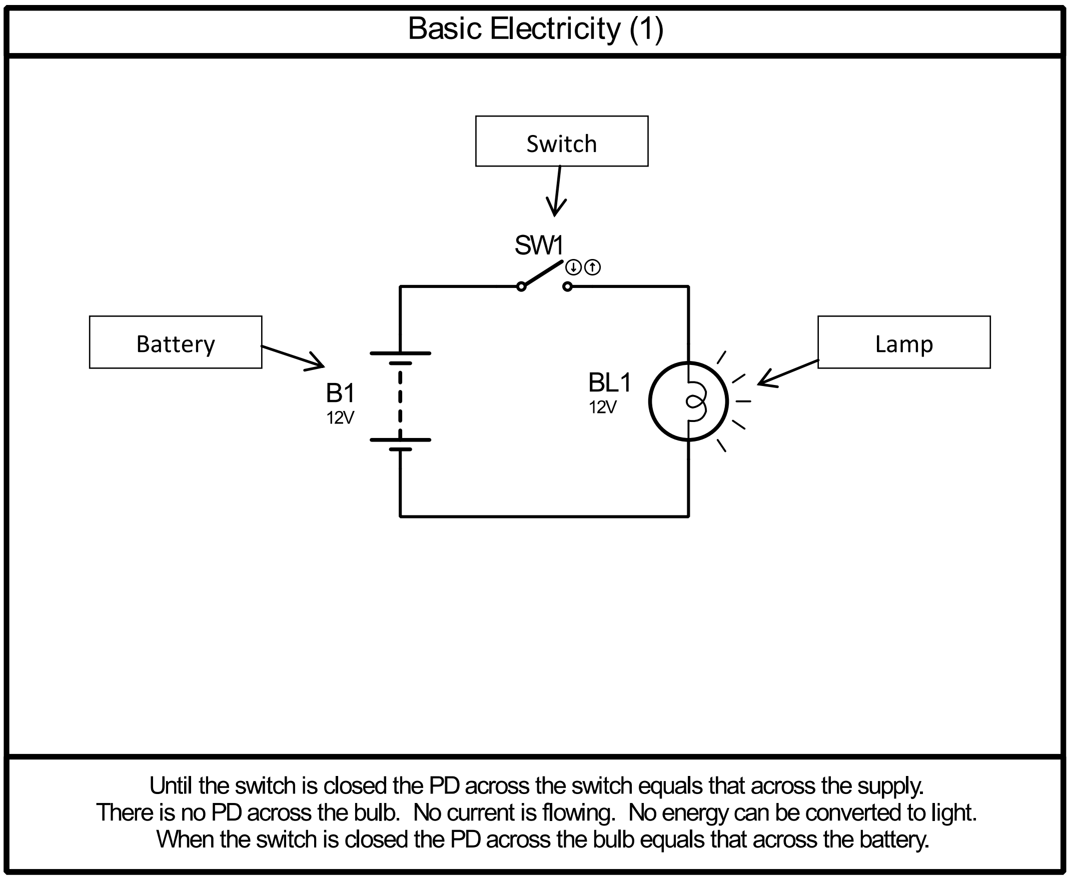

Basic01 - Proteus 8 Professional - Schematic Capture
File
Edit
Template
System
Help
Schematic drawing overview
Zoom and pan tools
Schematic drawing / editing area
Tool modes
DEVICES
BATTERY
LAMP
MASTERSWITCH
SWITCH
Basic Electricity
SW1
B1
12V
BL1
12V
Parts used in the schematic
Drawn electronic circuit schematic
"Play" (run simulation) button
No Messages
Base Design
Root sheet 1
+3500.0
-100.0

Figure 5-2: Schematic capture tab with "Basic Electricity 1" sample project open

The Schematic Capture tab, and how to draw your own schematics, will be covered in more detail in a following chapter. For now we only need to know how to open a sample project, navigate around it if necessary (zoom and pan), and run the simulation. Zooming and panning works the same as in the Visual Designer (flowchart) environment – in case you would like a refresher on this, please see section 2.3.5 in the “Creating Your First Project” chapter.

5.2. Simple Electrical Circuit

This section assumes that the project opened in the previous section is still open (if not, then please open it again).



Basic Electricity (1)
Switch
SW1
Battery
B1
12V
BL1
12V
Lamp
Until the switch is closed the PD across the switch equals that across the supply.
There is no PD across the bulb. No current is flowing. No energy can be converted to light.
When the switch is closed the PD across the bulb equals that across the battery.

Figure 5-3: “Basic Electricity 1” circuit schematic diagram

Figure 5-3 illustrates the project’s circuit. This circuit essentially consists of a battery, a switch and a lamp. The battery stores electrical energy that can be used to power the lamp. The purpose of the switch is to complete or break the circuit, thus turning the lamp ON or OFF. This circuit can be simulated in Proteus and can be interacted with during the simulation – i.e. the switch can be closed or opened by clicking on it. To run the simulation, click the “Play” button (blue triangle) in the bottom-left corner of the Proteus Schematic Capture window.

When running the simulation, it can be observed that when the switch is in the closed (ON) state, this allows for electricity to flow from the battery through the lamp. The battery is said to have the “potential” to supply electricity.

The positive side of the battery is said to have a “high potential” and the negative side of the battery a “low potential”.

An analogy for this is water in a high dam (positive side of the battery), which can flow to water in a low dam (negative side of the battery), via a water-wheel ("load", in this case a lamp). Note that it doesn't matter *how big* the dams are. The water flows because of the difference in height of the dams, not because of how big or small they are. The flow of water can be started or stopped by a sluice gate (switch). Whilst the water is in the high dam it has the *potential* (high potential) to "do work", by turning the water-wheel as it flows down to the lower dam. Once it has flowed down to the lower dam however, it does not have that potential anymore (low potential). In a similar way, electricity is said to flow from a point of high potential (positive side of the battery) to a point of low potential (negative side of the battery), when the two points are connected by a circuit.

The difference in potential (between high and low) is called "potential difference" (PD).

For electricity, the unit used to measure potential difference is called Voltage.

As a consequence of this potential difference or "pressure" exerted by the battery, electrical energy flows through the lamp and is converted into light (when the switch is ON). When the switch is turned OFF, it can be observed that the lamp remains OFF as the electricity is not able to flow through it.

5.3. Ohms Law (Voltage, Current and Resistance)

Consider the following experiment, involving a bucket with a pipe connected to it, and a tap at the end of the pipe.

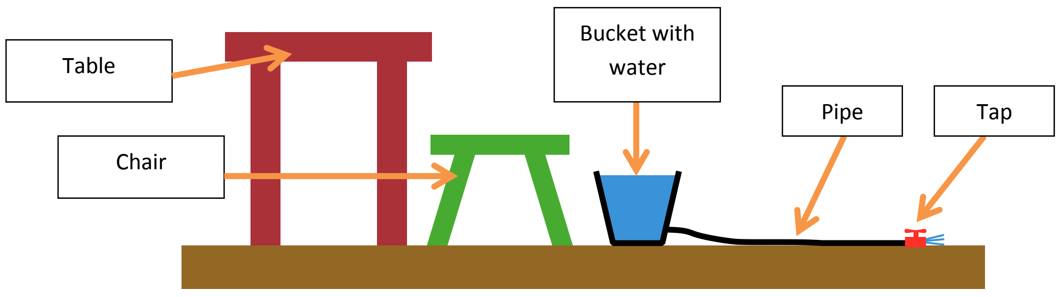


Figure 5-4: Bucket on floor (low voltage)

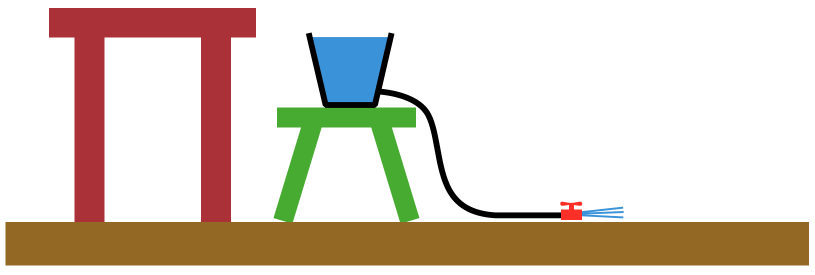


Figure 5-5: Bucket on chair (medium voltage)

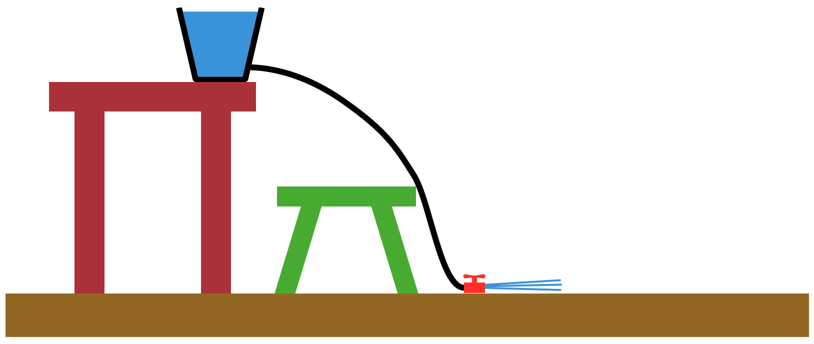


Figure 5-6: Bucket on table (high voltage)

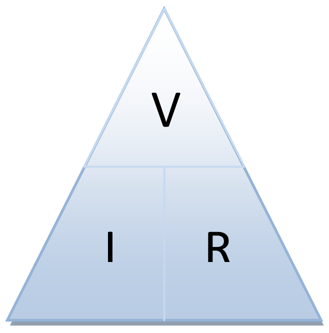
If we open the tap and leave it open (don't change it), but then move the bucket around, then we will find that the water from the bucket will come out faster when the bucket is on the table than when the bucket is on the floor. If you are in doubt regarding this, try it yourself! So long as the tap stays open and on the floor, the higher you lift the bucket the faster the water will come out of the tap!

It is also obvious that, if we leave the bucket in one place (e.g. on the chair), then closing the tap a little will make the water come out slower, and opening the tap more make the water come out faster.

If we relate this experiment to electricity, then the following correlation applies:

- **Voltage** is the height of the bucket. The higher we lift the bucket, the greater the "potential difference" (height difference between the bucket and the outlet tap). Note that it doesn't matter how big or small the bucket is, it is the height of the bucket which matters.
- **Resistance** is the tap. If we close the tap then it presents a greater resistance to the flow of water, and if we open the tap then it presents less resistance to the flow of water.
- **Current** is the amount of water coming out of the tap outlet – how much is coming out (millilitres per second, for example). The current would determine how long it would take to fill a glass, for example. Lower current means that it would take longer to fill a glass, and higher current means that it would take shorter to fill a glass.

The mathematical relationship between these 3 terms can be illustrated using the following diagram.



Current (I) = Voltage (V) / Resistance (R)

Voltage (V) = Current (I) x Resistance (R)

Resistance (R) = Voltage (V) / Current (I)

If Voltage remains the same, and the resistance is decreased, then the current increases (and vice versa).
If Resistance remains the same, and the voltage is increased, then the current increases (and vice versa).

You may be wondering about things like what the amount of water in the bucket represents. In electricity, the total amount of water in the bucket can be measured in "Coulombs" or "Amp-Hours". "Amps" are the unit of measurement for current, and will be discussed in the next section. "Amp-Hours" refers to how many hours a power source can supply 1 "Amp" of current for. Referring back to the water example, if we can supply 2 litres of water per hour for 3 hours, then the total "amount" of water will be 2 x 3 = 6 litres. You may have been batteries which have a milliAmp-Hour (mAh) rating before.

To proceed to a Proteus simulation demonstrating Ohm's Law in action, open the **"Interactive Simulation: Basic Electricity 2"** sample project (using the procedure described previously in section 5.1).

The Basic Electricity 2 sample project is another interactive simulation, which demonstrates the effect of introducing "variable resistance" into the lamp circuit from section 5.2. Whilst a switch is either "on" or "off", a variable resistor can be thought of as a "tap" or "valve" in the circuit, which can be adjusted to control the flow through the circuit. Opening the tap allows for more or quicker flow, whilst closing the tap results in less or slower flow.

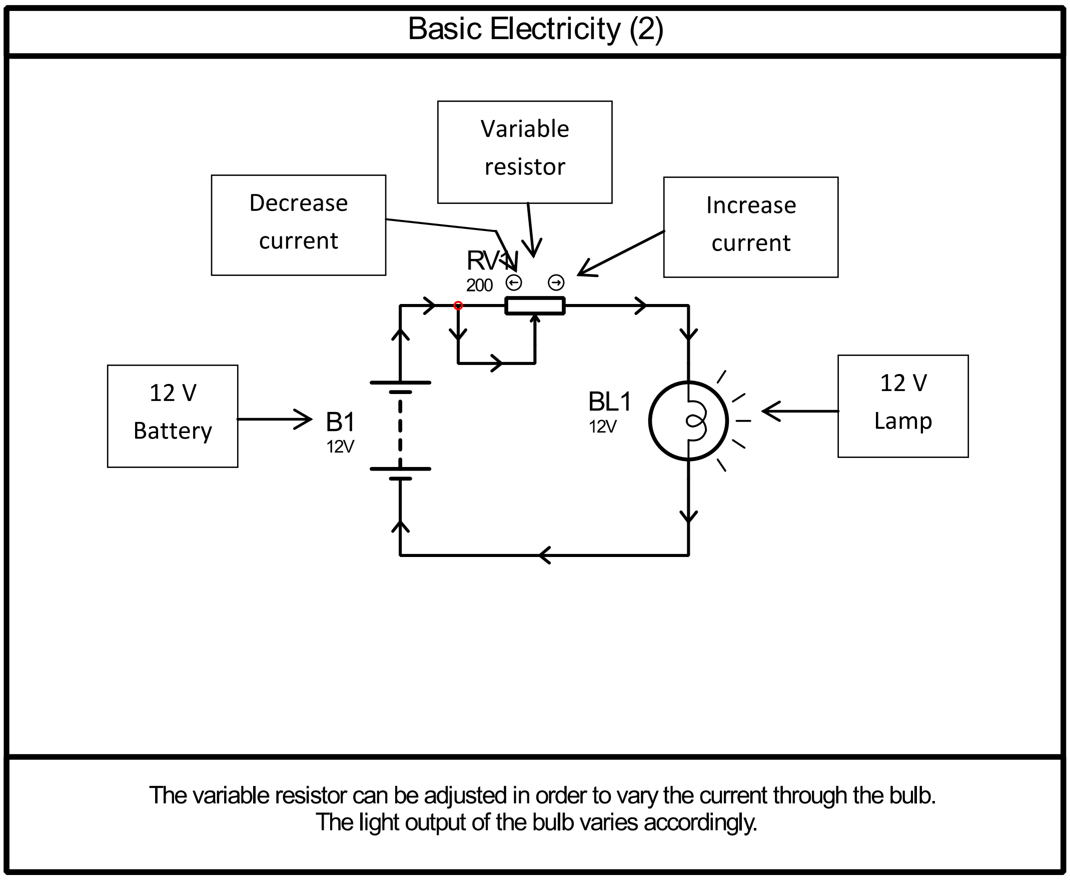

Basic Electricity (2)
Variable resistor
Decrease current
Increase current
RV1
200
12 V Battery
B1
12V
BL1
12V
12 V Lamp
The variable resistor can be adjusted in order to vary the current through the bulb.
The light output of the bulb varies accordingly.

Figure 5-7: "Basic Electricity 2" circuit schematic diagram

This circuit consists of a 12 V battery, variable resistor and a 12 V Lamp. According to Ohm's law, "Resistance (R) is *inversely proportional*[14] to the electrical current (I) flowing through the resistor, provided that the potential difference (V) across the resistor remains constant".

When running the simulation, the variable resistor is again an interactive element and can be adjusted to observe the effect of a change in resistance to the circuit. The electrical current flowing through the lamp can be varied by using the increase ("+") and decrease ("–") buttons on the variable resistor (as highlighted in Figure 5-7). It can be observed that as the current is increased, the lamp brightness increases. It can also be observed that as the current level decreases, this consequently results in the lamp brightness decreasing. From the results of the simulation it can be concluded that the lamp brightness is proportional to the electrical current flowing through the lamp, in accordance with Ohms law as described previously.

5.4. Electrical Measurements

Just like weight can be measured in kilograms and distance in kilometres, there are special units used for measuring electricity. These are:

Quantity Measured	Unit	Symbol
Voltage (V)	Volts	V
Current (I)	Amperes (or "Amps")	A
Resistance (R)	Ohms	R or Ω

Just like other units, these can be prefixed with a multiplier such as "kilo", "milli", etc – for example 100 milli-amps (100mA) or 10 kilo-ohms (10kΩ, often just written as 10k). The "Measuring Voltage, Current and Resistance" sample projects listed below include voltmeters and ammeters (for measuring voltage and current respectively).

5.5. Further Circuits

A number of further sample projects are available, which can be explored at your leisure if you are interested. The circuits listed below illustrate the effects of resistors placed in series or in parallel, as well as elaborate on the principles already discussed.

Sample Project	Principles Demonstrated
Basic Electricity 3	Series resistance.
Basic Electricity 4	Parallel resistance.
Basic Electricity 5	Making and breaking circuits using two switches.
Basic Electricity 6	Controlling a motors speed and direction using switches and a variable resistor.
Basic Electricity 7	Blowing a fuse by running too much current through it.
Measuring Voltage, Current and Resistance 1	Voltage over a battery and light-bulb are measured.
Measuring Voltage, Current and Resistance 2	The function of series resistors as voltage dividers is demonstrated.
Measuring Voltage, Current and Resistance 3	The use of an ammeter is demonstrated.
Measuring Voltage, Current and Resistance 4	The use of a voltmeter and ammeter are demonstrated.
Measuring Voltage, Current and Resistance 5	The function of parallel resistors as current dividers

[14] "Inversely Proportional" means that as one value increases, the other decreases. In the case of Ohm's law, as resistance increases then current decreases.

	is demonstrated.
Measuring Voltage, Current and Resistance 6	The way series voltages are added is demonstrated.
Measuring Voltage, Current and Resistance 7	Ohm's Law is demonstrated.
Measuring Voltage, Current and Resistance 8	The way resistors of different values divide current is demonstrated.
Measuring Voltage, Current and Resistance 9	A potentiometer (variable resistor) is added to the previous circuit to further demonstrate the concept.

If you should feel so inclined then there are also further sample projects available demonstrating electronic components and concepts such as capacitors, diodes, transistors, OpAmps, Logic Gates, Sequential Logic Circuits, Oscillators and more. For the purposes of this course however, a grasp of the basic electrical concepts just covered is sufficient.

6. Drawing and Simulating Circuits

Whilst we have been using sample designs for circuit simulation up till now, this chapter aims to teach how to draw your own circuits (schematics). Besides for being a useful skill on its own, this will also allow us to draw custom circuits to interact with our microcontroller and flowcharts.

6.1. For the Curious, an Overview of the Electronic Design Process

As mentioned in the introduction, a schematic is an electronic wiring diagram. It shows what is connected to what, but not exactly how.

In the electronic design process, after designing a schematic the circuit is generally converted into a PCB (printed-circuit-board) layout, which is a physical circuit implementing the "theoretical" connections specified in the schematic. Note that it is obviously possible to create multiple different PCB layouts from the same schematic. Below is an example of a schematic as well as a possible PCB layout thereof.

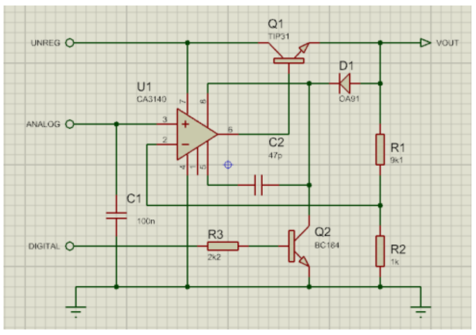


Figure 6-1: Example schematic

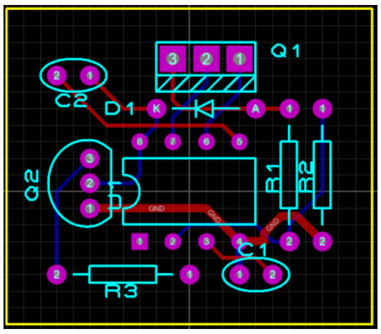


Figure 6-2: Example PCB Layout

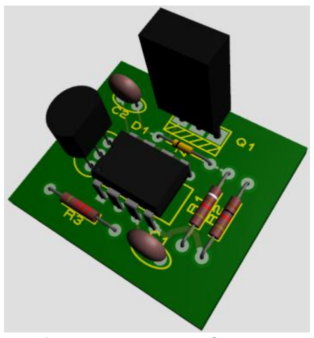


Figure 6-3: Example PCB

Once a schematic has been drawn, it is converted into at least one PCB Layout. The PCB design is then sent to a PCB manufacturer, who manufactures the printed-circuit-board. At this stage, the PCB contains only the "wires" on the schematic, but none of the components. The PCB is then "populated" by soldering components onto it.

Proteus can be used to design schematics as well as layout PCBs, however PCB layout is beyond the scope of this book as we will be using ready-made PCBs such as the Arduino UNO board.

(Tip: Besides for the instructions listed in this chapter, a Schematic Capture Tutorial and Help File are also available from the Help menu in Proteus when the Schematic Capture tab is open.)

6.2. Schematic Capture Environment Overview

To start with, please ensure that you have any project open which contains a schematic (if you're unsure of what to do then simply open the "Blinking LED for Arduino Uno" sample project by selecting "Open Sample Project" from the File menu, and then selecting it from the "Visual Designer for Arduino" category).

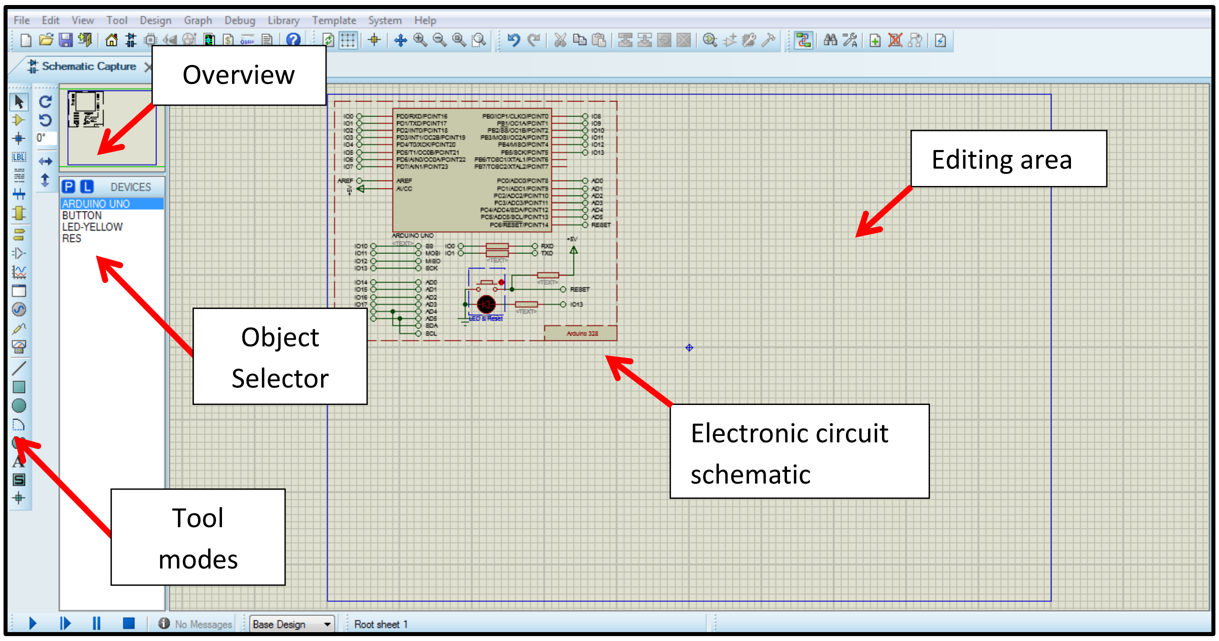


Figure 6-4: Proteus schematic capture tab

The schematic capture tab screen essentially consists of three main areas; the Overview area, Object Selector area and Editing area.

The editing area is where components are placed and connected ("wired-up"), in order to construct the electrical circuit. When a new Visual Designer project is created, Proteus automatically adds the microcontroller board (Arduino UNO in this case) to the schematic capture editing area (illustrated in the previous figure).

The overview area gives (as its name suggests) an overview of the entire schematic. When a new object is selected from the Object Selector however, then the Overview area is used to preview the selected object (this will be described in more detail later).

The Object Selector area normally contains a list of components (e.g. buttons, LEDs, buzzers) which have been selected (or "picked") for use in the project / schematic (the procedure for picking parts will be described shortly). If a tool mode other than Selection Mode or Component Mode (more on this shortly) is selected however, then the Object Selector will change depending on which tool is selected.

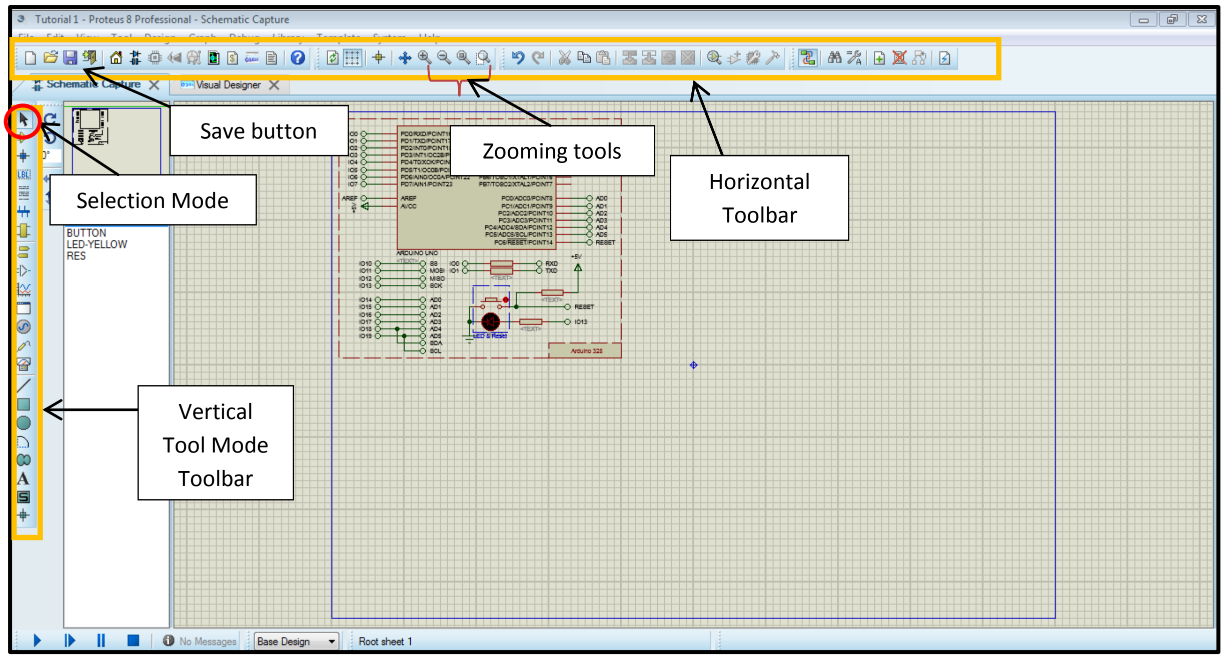


Figure 6-5: Schematic capture toolbars

The schematic capture tab has essentially two main toolbars, a horizontal toolbar and a vertical toolbar (Tool Mode Toolbar). The horizontal toolbar contains buttons pertaining to project file management (e.g. opening a new project, saving the project) as well as tools such as zooming, etc. The vertical toolbar (Tool Mode Toolbar) contains all the tools which can be used to draw the schematic.

6.3. Picking, Placing and Wiring Up Components

6.3.1. Picking Components

If not already available in the Object Selector list, then new components can be picked for use in the schematic from the Parts Library. With Selection tool mode ( ) or Component Mode ( ) selected; click on the P button above the Object Selector area (see the following figure) or alternatively use the keyboard shortcut (i.e. “P” key).

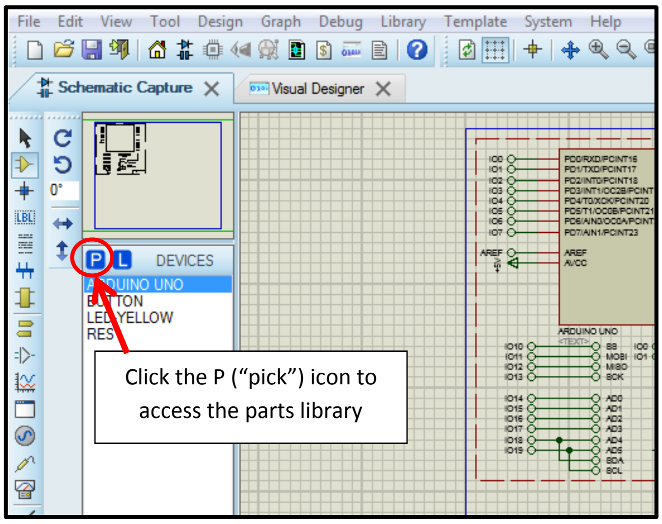

File Edit View Tool Design Graph Debug Library Template System Help
Schematic Capture
Visual Designer
P L DEVICES
ARDUINO UNO
BUTTON
LED-YELLOW
RES
Click the P (“pick”) icon to access the parts library

Figure 6-6: Accessing the parts library in Proteus

Components (e.g. LEDs, resistors, lamps, etc) can then be added to the Object Selector from the Parts library, so that they are available for use in the schematic.

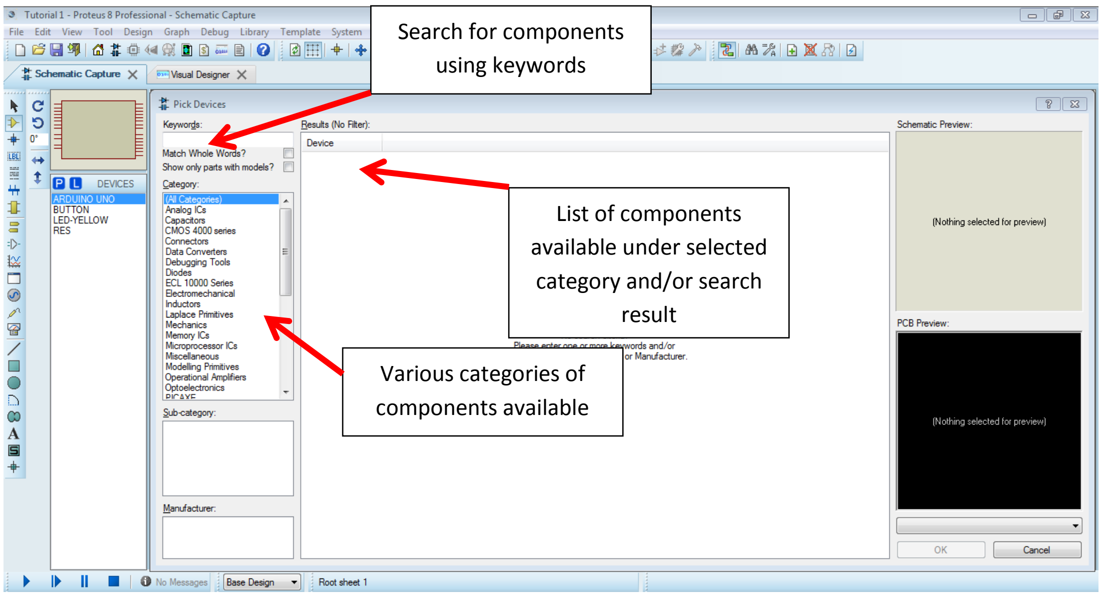

Search for components using keywords
List of components available under selected category and/or search result
Various categories of components available
Pick Devices
Keywords:
Category:
Results (No Filter):
Schematic Preview:
PCB Preview:
OK
Cancel

Figure 6-7: Overview of the Proteus Parts Library

A component can be found by either searching for it using keywords, or by selecting it from a category. Once found, it can be added to the Object Selector by double-clicking it, or by selecting it and then clicking "OK".

6.3.2. Placing Components

Once a component has been added to the Object Selector then it can be placed on the schematic using the following sequence:

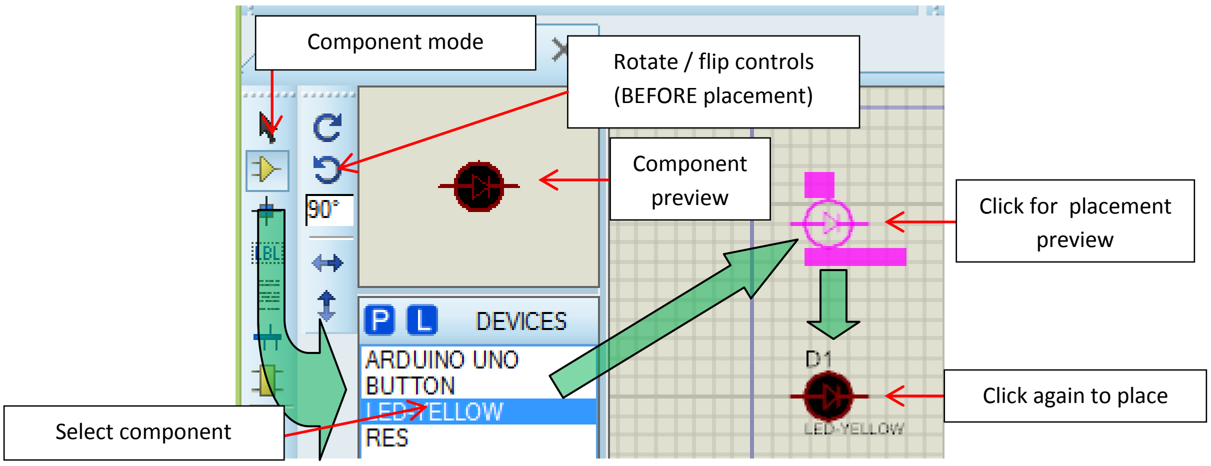


Figure 6-8: Component placement

1. Make sure that Selection Mode or Component Mode is active.
2. Select the component to be added to the schematic in the Object Selector.
3. Click once in the schematic editing area. A magenta preview of the component should now appear.
4. Move the preview to where you would like to place the component, and then click again to place it.

6.3.3. Component "Reference", "Value" and "Body"

You may have already noticed that components are automatically labelled with names such as "R1" / "D1" / etc. These are "**references**" (or "IDs") which help to uniquely identify components (both on the schematic and on a PCB). Components also have **values**, for example a resistor may have a value of "10k" (10 kilo-ohms). The component "drawing" is called the "component **body**".

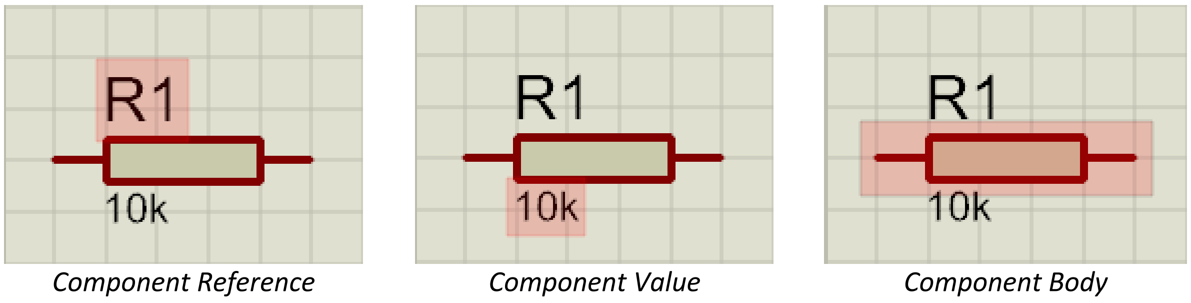


Figure 6-9: Component Reference, Value and Body

6.3.4. Moving, Rotating and Mirroring Components

Components can be **rotated** or "**flipped**" ("mirrored") *before* placement by using the controls on the vertical toolbar; a preview of the current rotation / mirroring will be displayed above the object selector (Figure 6-8). *After* placement rotation / mirroring can be achieved by right-clicking on the component and then selecting the desired operation from the popup menu. An option to **move** components is also available from the same popup menu ("Drag Object").

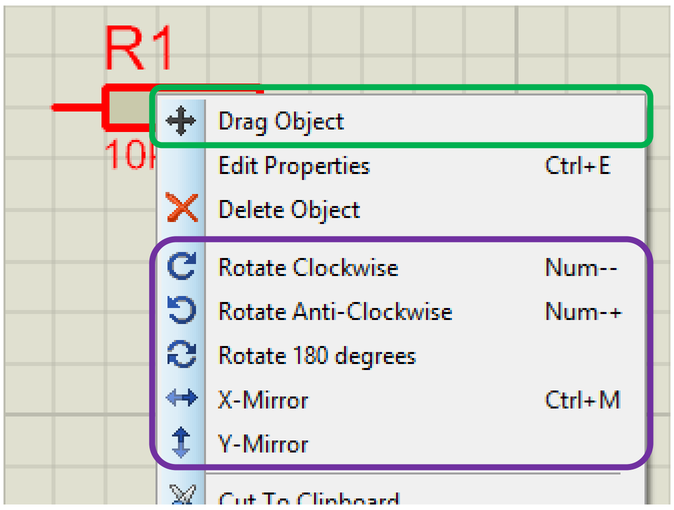


Figure 6-10: Component right-click popup menu

Importantly, note that it is possible to move the component reference and value labels <u>independently</u> of the component, which can be a source of great frustration if you are intending to move the whole component rather than just the label! To move the whole component, ensure that you click on the Component Body (Figure 6-9).

Another method for moving components, when Selection Mode is active, is to first "tag" them by clicking on them once or drawing a selection box around them (tagged objects will become red – see Figure 6-11), and then clicking and dragging them to a new location.

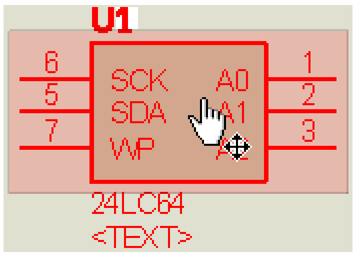


Figure 6-11: Tagged object ready to be moved

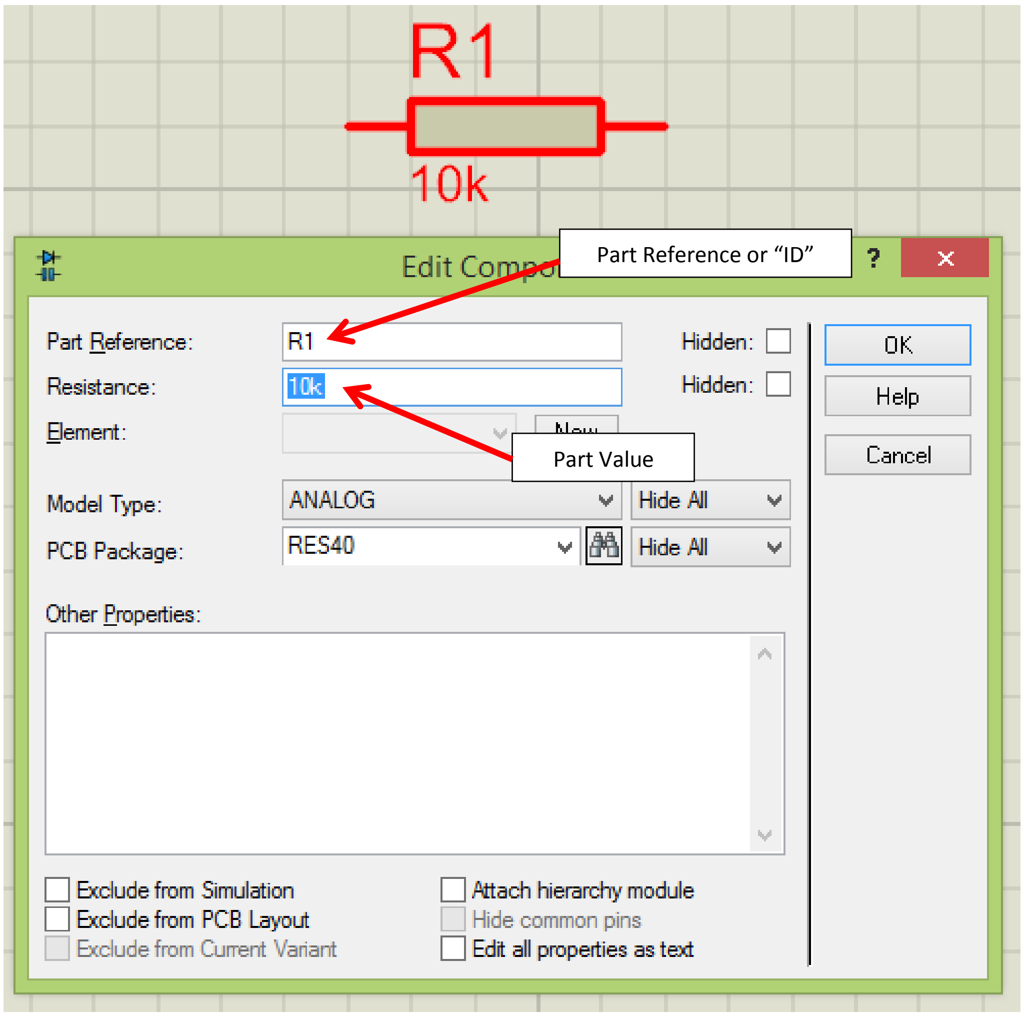
Components can be deleted or removed from the schematic by using the right-click popup menu (Figure 6-10), or by right-clicking on them twice (which is a convenient, but potentially also confusing if unexpected, shortcut).

For practice, it is recommended to try placing, moving, deleting and rotating/flipping (before and after placement) a few components now.

(Tip: Some components, such as “BUTTON”, have *interactive* regions; as we have seen in previous chapters, we can “press” the button by clicking on it. This can make these components tricky to “get hold of” however, since clicking on them to “select” them will instead result in an interaction (e.g. “pressing” the button). The solution is to either right-click on the component and select the desired action from the popup menu, or drag a box around the component.)

6.3.5. Editing Component References and Values

Both the **reference** and **value** of a component can be edited by right-clicking the component and then selecting “Edit Properties” from the pop-up menu; which will bring up the “Edit Component dialog” (Figure 6-12).



Part Reference: R1 Hidden: ☐
Resistance: 10k Hidden: ☐
Element:
Model Type: ANALOG Hide All
PCB Package: RES40 Hide All
Other Properties:

☐ Exclude from Simulation ☐ Attach hierarchy module
☐ Exclude from PCB Layout ☐ Hide common pins
☐ Exclude from Current Variant ☐ Edit all properties as text

Figure 6-12: Edit Component dialogue

Note that you must click on the component "drawing" or "body" (Figure 6-9), rather than on the reference text or value text, in order to properly bring up the "Edit Component" dialog. Clicking on the reference or value texts will instead bring up an "Edit Label" dialog for that particular text, which allows for editing of just the specific label which was selected (reference label or value label). The full "Edit Component" dialog, which is brought up by rather clicking on the component body, allows for editing of the component reference and value (as well as other advanced parameters) at the same time.

Left-clicking on a component once it has already been tagged (is red) will have the same effect as the "Edit Component" command available from the right-click popup menu (will also bring up the "Edit Component" dialog). Thus, left-clicking twice on an untagged component is a shortcut to the "Edit Component" dialog.

Proteus automatically increments the component references of newly placed components to ensure that they are unique, so there is normally no need to edit these manually. You may however wish to edit the value of, for example, a resistor (in order to change its resistance).

6.3.6. Making Electrical Connections between Components ("Wiring Up")

Once the required components are all added to (placed on) the schematic, these components can then be electrically connected together – or "wired up" – as required.

6.3.6.1. "Follow-Me" Wire Routing

The "Follow-Me" Wire Router is available in all tool modes. Components contain "pins", which are normally fairly obviously visible but which can be verified by hovering the mouse pointer over them. When the mouse is hovered over a component pin, then a green pencil ( ) will appear, indicating that a wire can be drawn from/to this point. Wiring is started by clicking on the start point for a wire and then clicking again on the end point for the wire. The following figure illustrates this procedure.

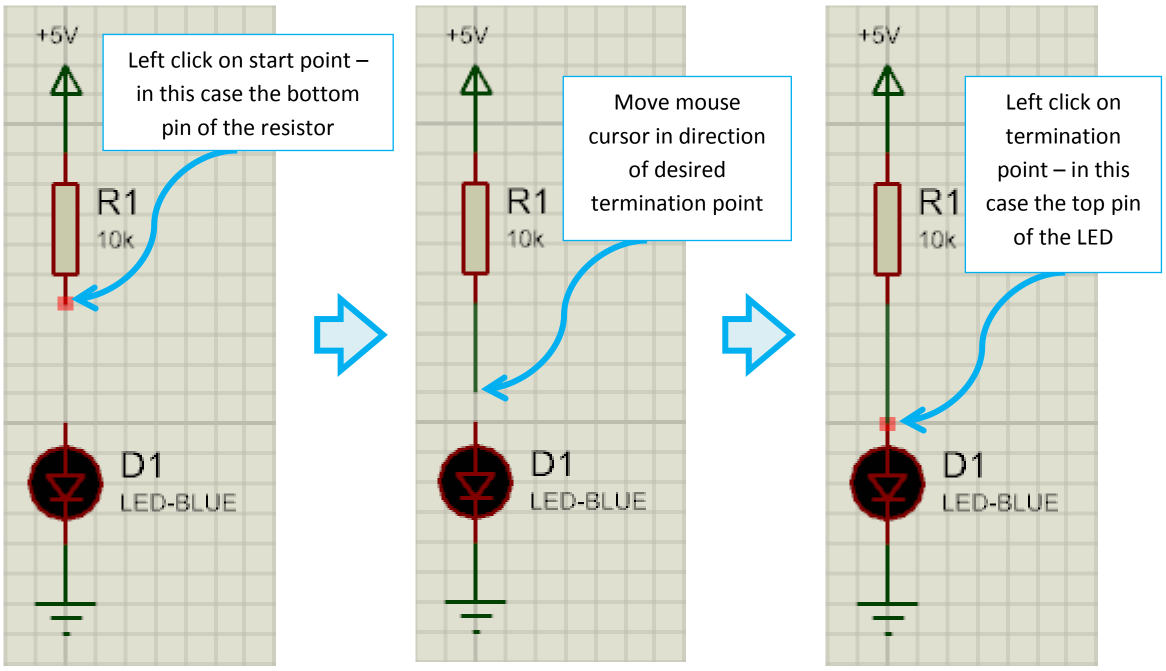

+5V
Left click on start point – in this case the bottom pin of the resistor
R1
10k
D1
LED-BLUE
+5V
Move mouse cursor in direction of desired termination point
R1
10k
D1
LED-BLUE
+5V
Left click on termination point – in this case the top pin of the LED
R1
10k
D1
LED-BLUE

Figure 6-13: Making an electrical connection with a wire – from start to finish

(Tip: Clicking along the route whilst tracing the wire will anchor it at that specific point. After the wire has been drawn, it can be moved by first selecting it (after which it will turn red to indicate that it has been "tagged"), and then dragging it – or by right-clicking on the wire and then selecting "Drag Wire" from the popup menu. A wire can also be deleted by using the applicable command available from the right-click popup menu, or by right-clicking on it twice.)

6.3.6.2. Making Connections with Terminals

Because having many wires running all over a schematic can get messy and confusing, an alternative method for making a connection between two pins also exists, called "Terminals".

Two or more terminals will be considered electrically "connected" if they have the same name. This allows for an electrical connection to be made between two pins without having to draw a wire between them. Figure 6-14 illustrates a connection made via terminals between the PB5/SCK/PCINT4 pin on the Arduino Uno microcontroller, and the resistor next to the LED (with the terminals bearing the common name of "IO13"). This connection enabled us to blink the LED in our first project.

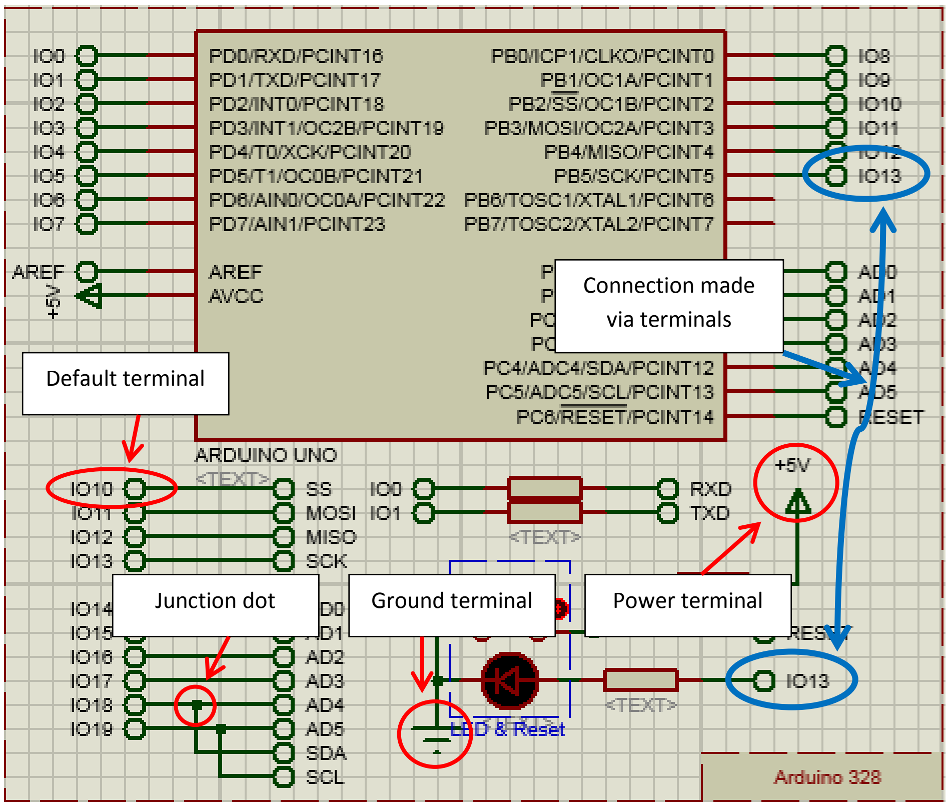


Figure 6-14: Pin connections using terminals

A terminal point can be placed on the schematic using the “Terminal Mode” ( ) available in the vertical toolbar (Figure 6-15).

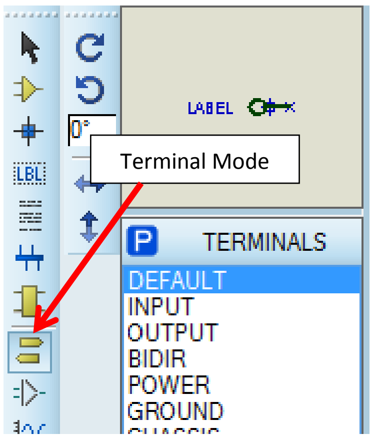


Figure 6-15: Terminal Mode

Placing terminals using Terminal Mode is very similar to placing components using Component Mode (except that it is not necessary to pick the terminals from a library first).

Various types of terminals are available; however the ones which we will be using most are **Default**, **Power** and **Ground**.

Power (⏚) and Ground (⏚) terminals are automatically recognised by Proteus as being power sources. Note that we don't need to tell Proteus exactly where the power comes from (it could come from a battery / plug / generator / etc – the exact source normally does not matter for design and simulation purposes); it is enough simply to specify that the power source is there. Power terminals are connected to +5V (5 volts) by default, and ground terminals are connected to 0V (zero volts) by default. The term "Ground" originates from the practice of placing a copper rod into the ground in for example building wiring, the exact reason for which is beyond the scope of this book. Suffice to say, "Ground" means 0V (zero volts). The voltage of a power terminal can be changed in the same way as a components reference or value can be changed – by selecting "Edit Properties" from the terminal right-click popup menu, or by left-clicking on the terminal twice.

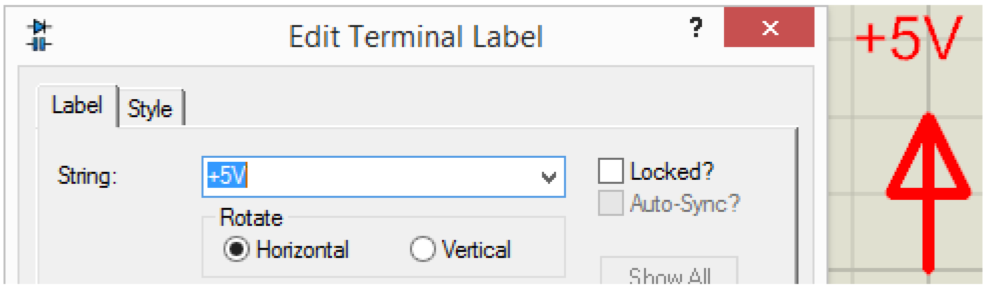


Figure 6-16: Edit Terminal Label dialog

Terminals should be wired up in the normal way (section 6.3.6.1) to the pins of the components which they should be connected to. Note that simply placing two components or terminals so that their pins "look" connected (overlap) does **not** mean that they are actually connected – they must be joined by a wire!

Note that an alternative method for placing a terminal (as well as other components) on the schematic is to right-click on an empty spot in the editing area, and then select **Place – Terminal** from the popup menu (Figure 6-17).

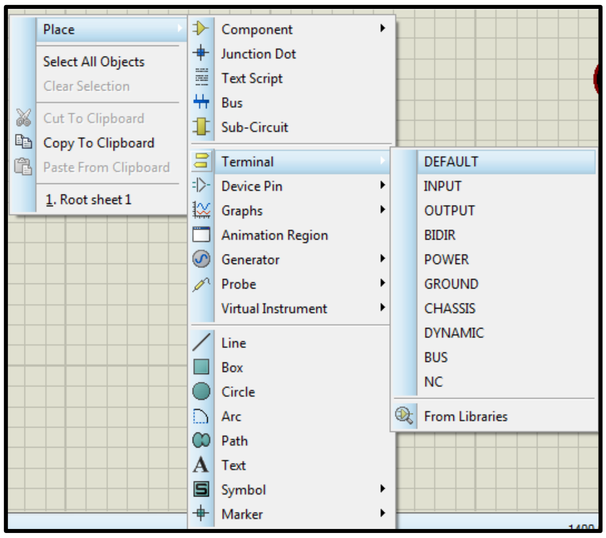


Figure 6-17: Adding a terminal point to the schematic from the right-click popup menu

6.3.6.3. *Junction Dots*

A junction dot indicates a point where two wires are joined. Wires which cross over each other without a junction dot are not considered joined.

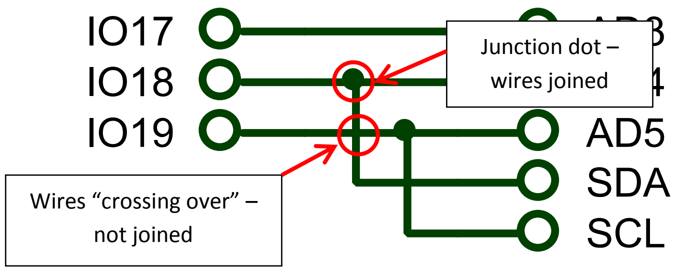

IO17
IO18
IO19
Junction dot – wires joined
Wires "crossing over" – not joined
AD5
SDA
SCL

Figure 6-18: Junction dots

Junction dots can be created automatically as part of the Follow-Me Wire Routing process (by routing a wire to another wire). They can also be explicitly placed using the right-click popup menu, or by using the Junction Dot tool mode ( ) available from the vertical toolbar.

6.4. Drawing and Simulating Your Own Schematic

Next, we will draw and simulate our own circuit schematic.

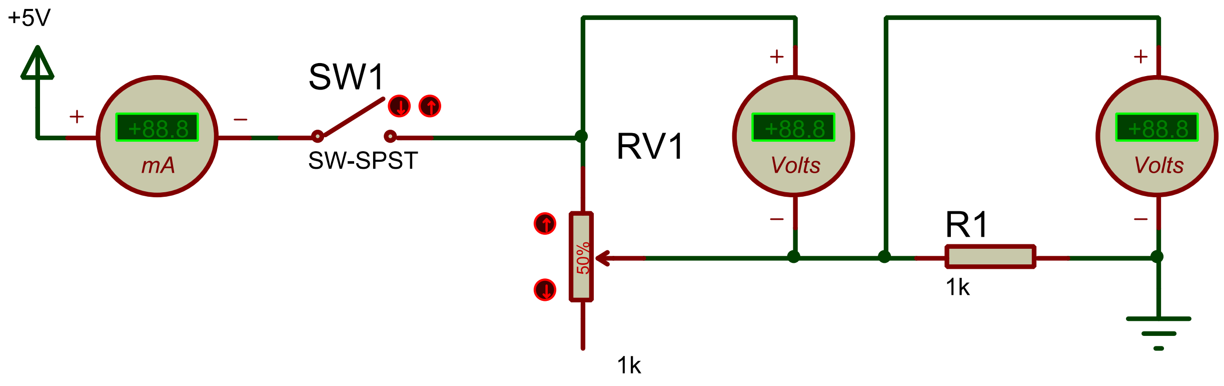


Figure 6-19: Schematic circuit to draw

Drawing this schematic will test all the skills learned in this chapter, and will also allow us to simulate the principles learned in the previous chapter (Ohm's law).

(Tip: Besides for the instructions listed in this chapter, a Schematic Capture Tutorial and Help File are also available from the Help menu in Proteus when the Schematic Capture tab is open.)

To draw the schematic illustrated in Figure 6-19, follow the below steps.

1. Save ( ) any work which you may have open and don't want to lose, and then close ( ) any open project (File menu and then Close Project).
2. Click on the Schematic Capture icon ( ) to open the Schematic Capture tab (you may also start a new project if you wish, however the aforementioned method is a quick shortcut – the project can be called "SimpleCircuit" if using the New Project wizard, and could also be saved under this name by using the Save command if using the former method).
3. Pick the following components from the Component Library (if you need a reminder on how to pick components then please section 6.3.1 – when in Component Mode ( ), click the

button above the Object Selector). The easiest method for finding the components in the library is to type their names into the “Keywords” box.

    a. RES (“Resistor”)[15].
    b. POT-HG (“Potentiometer”, another name for a variable resistor).
    c. SW-SPST (a switch – technically a “single-throw single-pole” switch, the explanation for which is beyond the scope of this book).

4. Place the components and terminals (one power terminal and one ground terminal) on the schematic.
5. Also select “Virtual Instrument” tool mode ( ) on the vertical toolbar, and place a DC Ammeter and 2 x DC Voltmeters on the schematic (these will be listed in the Object Selector when “Virtual Instrument” tool mode is selected).
6. Change the following properties of the components / terminals / instruments:
    a. Set the Power Terminal for +5V.
    b. Change the resistance of the resistor to 1k.
    c. Change the Display Range of the DC Ammeter to “Milliamps” (this can be done from the Edit Component dialogue for the DC Ammeter, by the “Display Range” option which is below the Part Reference and Part Value settings).
7. Move, rotate and flip components as necessary, and wire everything up, till your schematic looks like the previous figure. Note that some components, such as the SW-SPST, can be difficult to “get hold of” for moving due to having interactive elements (which “interact” when clicked on rather than allowing for moving) – in this case it can be easier to right-click on the component and then select “Drag Object” from the popup menu, or drag a box around the component and then move it.
8. Save your work! (As mentioned, the project can be called “SimpleCircuit”, if it has not already been assigned a filename.)

We are now ready to simulate the circuit. This circuit allows us to experiment with the principles of voltage, current and resistance described in the previous chapter.

(Tip: If you would like your circuit to have arrows on it indicating current flow, like the circuits in the Interactive Simulation sample projects do, then do the following: With the simulation stopped, click the System Menu, select Animation Options, tick the “Show Wire Voltage by Colour?” and “Show Wire Current with Arrows?” options, and then click ok – when you run the simulation again, then the arrows will appear.)

[15] Note that you may see a large number of results when searching for “RES”. If you however type exactly “RES” (without quotes) under “Keywords”, then the RES component will be selected by default; it should show up in the preview on the right of the dialog, and can be added by simply clicking “OK” (or pressing the “Enter” key twice).

7. Drawing Custom Peripherals

Up till now, we have been using ready-made circuits in our flowchart projects. Proteus automatically inserts the microcontroller circuitry onto the schematic when starting a new project, and automatically inserts the circuitry for a peripheral (button, LED, LCD, etc) when it is added to the project from the peripheral gallery in Visual Designer. This mimics the way in which rapid-prototyping boards (such as the Arduino Uno board itself, and the Grove system of peripherals or add-on boards) can be used to construct a project in the "real-world". It is also obviously possible however, both in simulation and in the real-world, to construct custom circuits to interact with the microcontroller.

In the world of Arduino, custom circuits are sometimes built using a technique called "breadboarding".

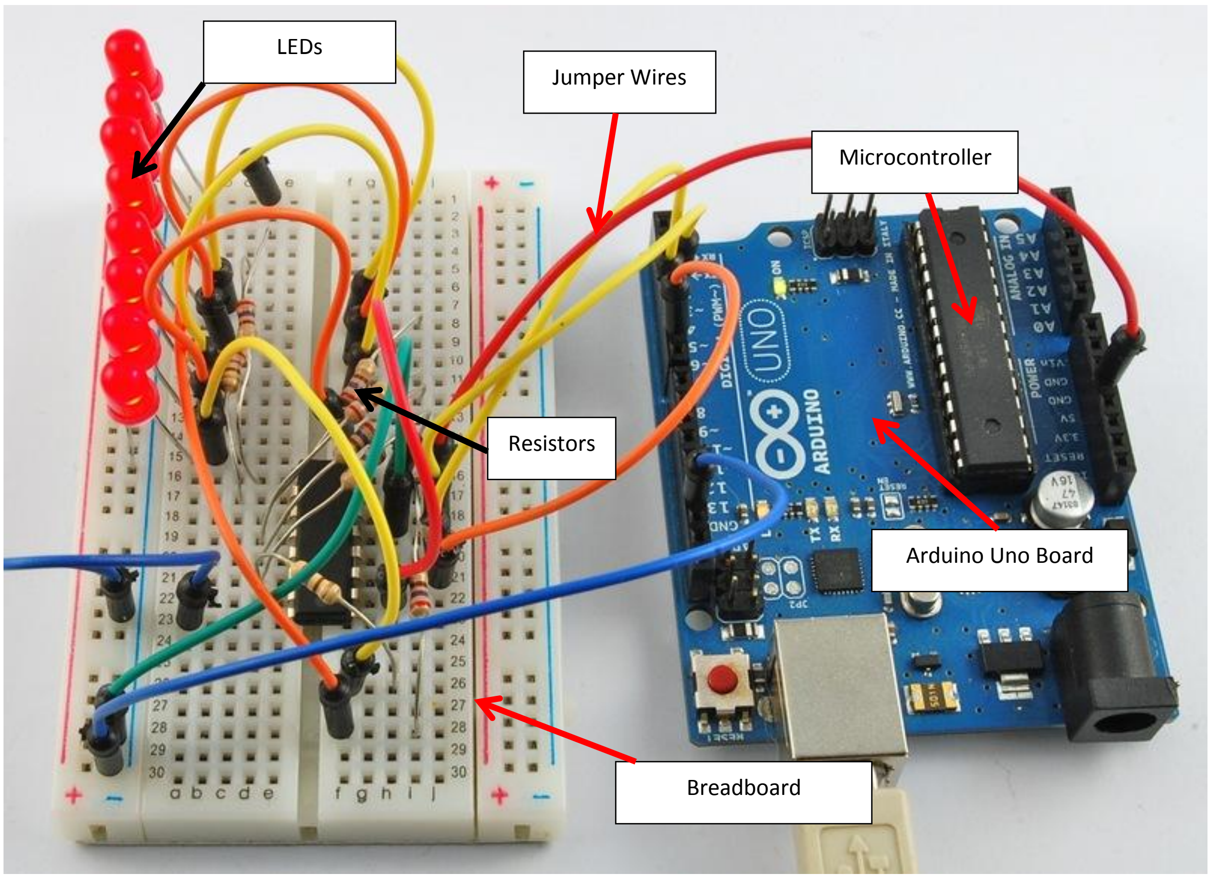


Figure 7-1: Breadboarding[16]

[16] Breadboarding image from http://learn.adafruit.com/adafruit-arduino-lesson-4-eight-leds/overview, Attribution Creative Commons license, unmodified except for labels added.

The previous figure may look somewhat complicated, but don't worry – drawing custom circuits in Proteus is a lot less confusing, less prone to errors (such as incorrect or loose connections), and easier to debug in the case of a problem.

A breadboard is a plastic board with holes in it, which are electrically connected in a certain pattern. Components and jumper-wires (jumper-wires are simply wires with pins on the ends) can be inserted into the breadboard holes in order to build a circuit. Other options for building a real-life custom circuit include stripboard such as Veroboard (which is similar to a breadboard, but not reusable), or designing your own PCB (as mentioned in a previous chapter). Since we can and will be building and simulating our circuit in Proteus, and because the focus of this course is mostly circuit and software design, it is not necessary to cover breadboarding in depth. If you would however like to construct a "real-life" version of the circuit we will be building in this tutorial, then there are plenty of resources regarding how to breadboard online.

7.1. Project Summary: Sequencing LEDs

For this chapter, we will be drawing circuitry and constructing a flowchart to sequence some LEDs (flash them one after another). We will also control the speed of sequencing using a potentiometer (variable resistor – this will be described in more detail shortly). The final circuitry is illustrated in the following figure (note how much tidier this looks than the breadboard situation!):

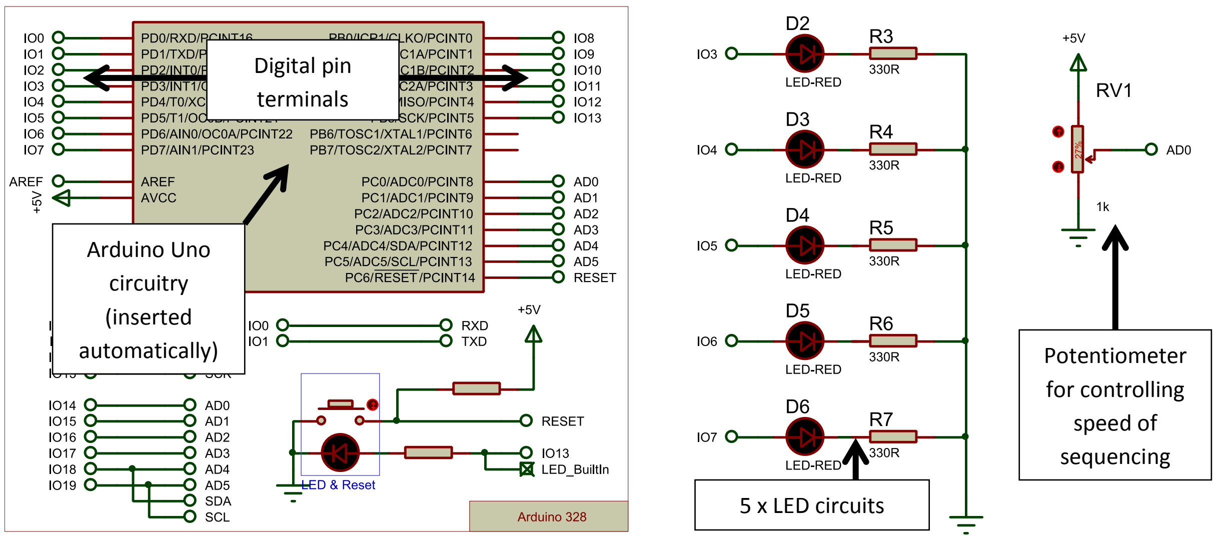


Figure 7-2: Sequencing LEDs circuitry

7.2. Drawing the Circuitry

(Tip: If you get stuck whilst trying to draw the schematic, or for a refresher on how to do so, please refer to Chapter 6.)

1. To begin, please start a new Arduino Uno Flowchart Project. If you need a reminder on how to do so, then please see section 2.2, 3.3 or 4.1 (remember to save ( ) any work which you may have open and don't want to lose, close ( ) any open project, and then select "New Flowchart" from the Home ( ) screen). The project can be called "SequenceLEDs".
2. Switch to the Schematic Capture ( ) tab.

3. Pick the following parts from the Component Library (the other components needed should already be in the Object Selector). If you need a reminder then please see section 6.3.1 (when in Component Mode ( ), click the button above the Object Selector; then use the "Keywords" box to search the library).
    a. LED-RED (Red LED)
    b. POT-HG (Potentiometer / Variable Resistor)
4. Place components and terminals as indicated by Figure 7-2; wiring-up, and setting any custom values needed. This will involve doing the following:
    a. Place 5 x red LEDs (LED-Red).
    b. Place 5 x resistors (RES).
    c. Place 1 x potentiometer / variable resistor (POT-HG).
    d. Place 6 x DEFAULT terminals.
    e. Place 1 x POWER terminal and 2 x GROUND terminals.
    f. Wire up, as depicted in Figure 7-2.
    g. Name the 5 x DEFAULT terminals which are connected to the LEDs from IO3 to IO7 (IO3, IO4, IO5, IO6, IO7). Note that the names consist of the letters "i" and "o" (for input/output), and then the numbers 3-7. Please see section 6.3.6.2 / Figure 6-16 if you need a reminder (right-click on the terminal and then select Edit Properties from the popup menu – enter the terminal label / name and click "OK"). By naming these terminals from IO3 to IO7 we are connecting them to the corresponding terminals at the Arduino Uno microcontroller pins.
    h. Name the DEFAULT terminal which is connected to the potentiometer "AD0". Note that the name consists of the letters "a" and "d" (for analogue/digital, this will be covered shortly) and then the number 0 (zero).
    i. Set the POWER terminal to +5V (please see section 6.3.6.2 / Figure 6-16 if you need a reminder).
    j. Change the **values** (resistance) of the resistors to 330R (which stands for 330Ω, but it is easier to type "R" since most keyboards don't have a "Ω" key). If you need a reminder on how to change a component value then please see section 6.3.5 / Figure 6-12. If you are wondering whether there are easier ways to change the values of the resistors (rather than doing so manually one-by-one) then the answer is yes; the LED and resistor circuit could be first constructed / set and then duplicated (as described in the next section), or a tool called the "Property Assignment Tool" could be used to more quickly set the values of all of the resistors. Use of the "Property Assignment Tool" ("PAT") is beyond the scope of this book, however information is available in the Proteus Schematic Capture help file. If in doubt, then the simplest solution for now is to just edit the values one-by-one (there are only 5 of them).

7.2.1. Copy and Paste Facilities

Note that Proteus has copy-and-paste methods available, which can ease the repetitive task of drawing the Terminal-LED-Resistor circuit each time. Notably, the "Block Copy" ( ) facility on the horizontal toolbar can be used to do this. In order to use the Block Copy facility, first highlight ("tag") the circuitry which you would like to copy. This can be done by first ensuring that Selection Tool Mode is selected ( ), and then drawing a box around the circuitry to be copied. The highlighted ("tagged") circuitry will turn red. Next click the Block Copy button on the horizontal toolbar (or right-click on the tagged circuitry and select Block Copy from the context menu). A magenta preview of the new copied circuitry will appear – move it to where you would like it to be placed, and then click to place it.

7.3. LED Circuits

You may be wondering exactly how the LED circuits work, and why we have resistors in them.

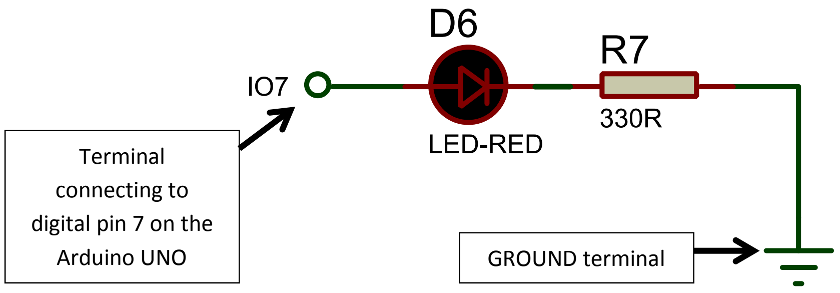


Figure 7-3: LED circuit

The basic principle is that when the Arduino Uno sets an output pin "high" ("on" or "true") in the software (flowchart), then it generates an output voltage of 5 volts on that pin. Because of the potential difference between the output pin (5 volts) and ground (0V), electricity flows from that pin, through the LED and resistor (lighting up the LED in the process), and down to ground (0V). When the pin is set "low" ("off" or "false"), then it generates an output voltage of zero volts (0V) on that pin – because there is no potential difference between zero volts and ground (both 0V), no electricity flows (and the LED turns "off"). LEDs themselves are diodes (LED stands for "light-emitting-diode"). We won't go in-depth into what diodes are here, but by way of explaining the need for the resistor we should know that diodes do not have any resistance as such. Without an "extra" resistor in the circuit, there is not much resistance to the flow of electricity in the circuit, and this can result in "too much" electric current flowing through the circuit. Too much electric current flowing through the LED can damage it. Thus, the LED needs a resistor in its circuit ("in series" with it), in order to limit the amount of current which will flow through it (in accordance with Ohm's law). This resistor is called a "current-limiting resistor", due to its application in the circuit. Does it matter if the resistor is "before" or "after" the LED? The answer is "no". Whether you place a tap at one end of a pipe or the other, you can still control/limit the amount of water flowing through the pipe using the tap. In mathematical terms, the current flowing through the circuit is I = V / R (see section 5.3). Since the LED does not have any resistance as such, the amount of current (I) flowing through the circuit is determined by the voltage (V, 5V from the Arduino pin) and the resistance in the circuit (330R). Without the resistor the resistance in the circuit would be very small, resulting in a small R value and thus large I value in the equation – meaning a large current (which could cause the "magic smoke"[17] inside the LED to escape).

7.4. Analogue and Digital Electronics

You may have noticed in the previous section that we have connected the LEDs to "IO3"-"IO7" (standing for input/output 3 to 7), and that we have connected the potentiometer (variable resistor) to "AD0" (standing for analogue/digital zero).

[17] Section 1.2.6.

Digital electronics refers to electronics which think of everything in ones and zeroes – on or off, true or false, high or low. Inside a microprocessor, this is how things work. For the Arduino, this corresponds to five (5) volts ("on" / "true" / "high") and zero (0) volts ("off" / "false" / "low").

Analogue electronics refers to electronics which looks at something in-between – it could be 1 volt, 2.5 volts, 3 volts, etc – much more specific.

Technically (for the Arduino Uno at least), anything below 1.5V would be considered a "zero", and anything above 3V would be considered a "one" – in-between would be "undefined" (could go either way), and outside of those ranges could damage the microcontroller.

7.4.1. Analogue-to-Digital Converters

Because the microprocessor itself only understands ones and zeroes, there is a peripheral built into the microcontroller called an "analogue to digital converter" (ADC). The analogue to digital converter takes an analogue voltage, and converts it into ones and zeroes which the microprocessor can understand (please see section 1.2.3 if you need a reminder on the difference between a microprocessor and a microcontroller).

Without going in-depth into the binary number system (in which numbers are represented using only ones and zeroes), it is sufficient to understand that the ADC converts the analogue voltage into a number inside the microcontroller.

The number doesn't exactly correspond to the voltage (i.e. it is not "1" for 1 volt or "5" for 5 volts); but rather represents a "scale" of where the voltage is between two reference points. Because the Arduino Uno works on 5 volts, the top of the scale is 5V and the bottom of the scale is 0V.

The biggest number which the ADC can produce is 1,023, so this is at the top of the scale and (in this case) means 5 volts (this is called the "resolution" of the ADC – how precisely it can measure the analogue voltage). So for an analogue voltage of 5 volts the ADC would output a number of 1023. For zero volts (0V) it would be zero (0). For 1 volt it would be about 205. 2.5V would be about 512, 3V would be about 614, and so on.

This can be represented using the following equation: ADC_Output = (Input_Voltage / 5) x 1023.

Different ADC resolutions and reference voltages are possible in the electronics world, but for our purposes with the Arduino Uno the aforementioned numbers and equations are applicable.

7.4.2. Using the Potentiometer to Generate an Analogue Input Voltage

The potentiometer (variable resistor) in our circuit is used to generate an analogue input voltage which we will use to determine the speed of sequencing of our LEDs.

In real-life, a potentiometer is usually a knob, like a volume knob.

Figure 7-4: Potentiometer with knob

Figure 7-5: Potentiometer knobs on electronic device[18]

Technically, the potentiometer is a "voltage divider" circuit. According to the laws of series resistance (resistors placed one after another), a voltage which passes over two resistors in series will be divided proportionally between them. This is illustrated by some of the sample projects listed in the "Basic Electric Circuit Theory and Simulation" chapter, as well as by the circuit which we constructed in the "Drawing and Simulating Circuits" chapter.

Another way of putting this is that the voltage over the potentiometer is used up "smoothly" within it, and thus by tapping into the potentiometer at a certain point we can "generate" an analogue voltage somewhere in-between the voltage at the top and bottom of the potentiometer.

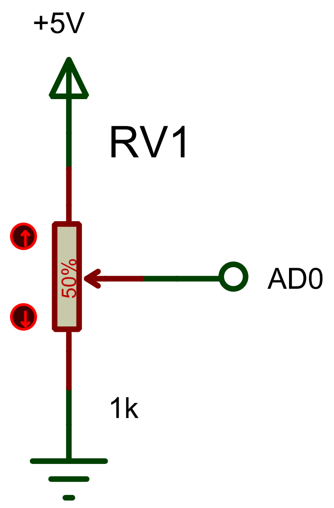


Figure 7-6: Potentiometer circuit

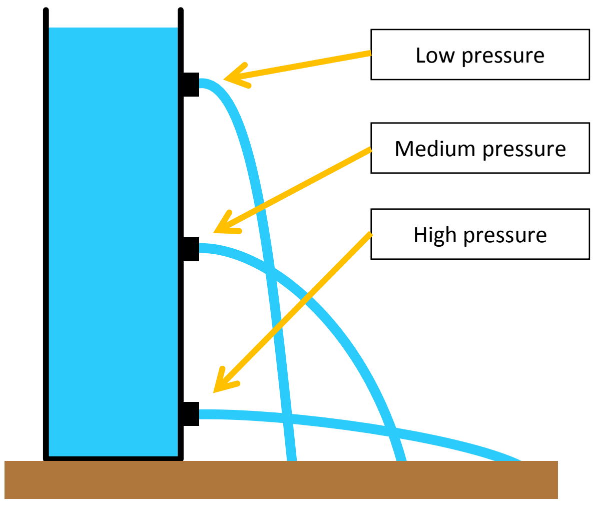


Figure 7-7: Water pressure experiment

Figure 7-7 depicts an experiment which can be conducted with water which can help to act as an illustration. Water is placed in a container (such as a tall bottle), with equal-sized holes cut into it at different heights along the side of the container. Water pressure at the bottom of the container is higher than the water pressure at the top of the container (this is because the water at the bottom of the container is being "squeezed" or "pressed on" by the weight of all of the water on top of it). This can be seen visually because the high-pressure water at the bottom of the container (which is being "squeezed" or "pressed on" more) shoots out further than the low-pressure water at the top of the container, which does not shoot out as far. In-between the top and bottom of the container,

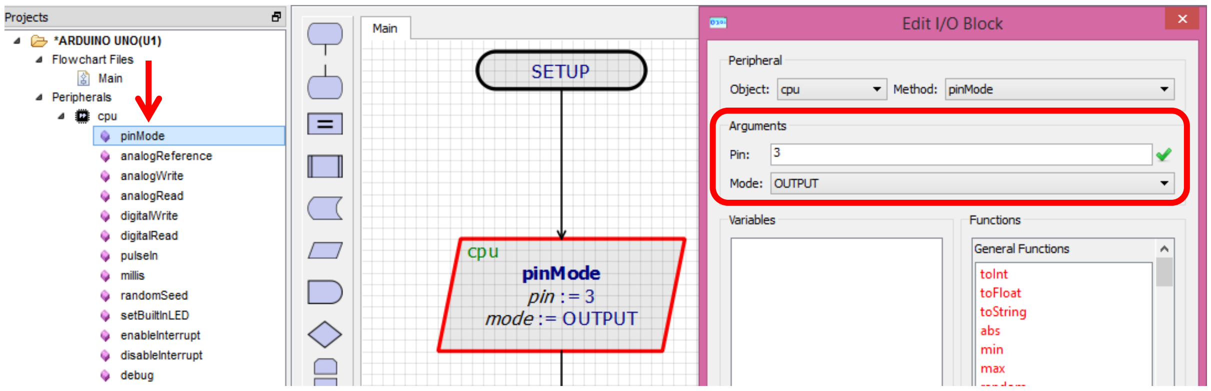
[18] https://commons.wikimedia.org/wiki/File:Audio_SwitchBoxTwo_(With_Volume_Knobs)_(2011-10-08_by_Kevin_B_3).jpg

the water pressure is also somewhere in-between what it is at the top and bottom of the container. In electrical terms, the “potential difference” (electrical “pressure”) is greater when measured between the top and bottom, than it is when measured between (for example) the top and middle. The “pressure” (“voltage”) is distributed evenly along the height of the container. This ties in with what we learnt in section 5.3 (“Ohm’s Law”). Note that whilst water pressure in the experiment is determined by gravity, and thus pressure will always be higher at the *bottom* of the container, electrical “pressure” is determined by voltage, which could be drawn any way around in our circuit. In Figure 7-6, the “high pressure” (+5 volts) is at the top, whilst the “low pressure” (ground, or 0 volts) is at the bottom (opposite to our water experiment).

Suffice to say, when connected up as it is in our present circuit, the potentiometer will “output” an analogue voltage which is somewhere between zero and 5 volts (which is then “input” into our Arduino Uno microcontroller).

7.5. Inputs and Outputs

The pins on a microcontroller can be either “listening” (input), or “speaking” (output). By default, the pins are “listening” (inputs). If we would like to change some of the pins to be outputs however, such as to control our LEDs, then we need to manually tell the microcontroller to set these pins as such; this is done using the “pinMode” method available under the cpu peripheral.

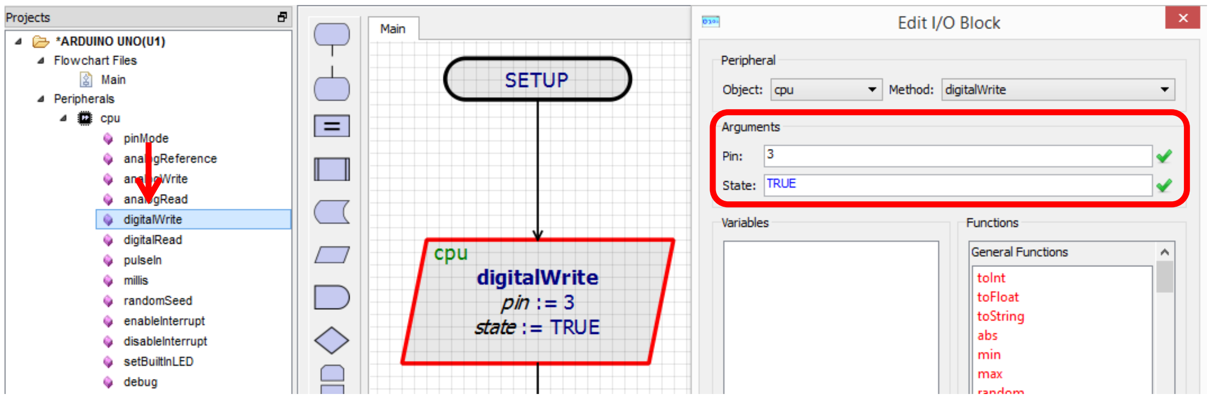


cpu
pinMode
pin := 3
mode := OUTPUT

Peripheral
Object: cpu Method: pinMode
Arguments
Pin: 3
Mode: OUTPUT

Figure 7-8: "pinMode" method of the cpu peripheral

Once a pin has been set as an output (using the pinMode method), we can switch it on or off using the “digitalWrite” method available under the cpu peripheral.



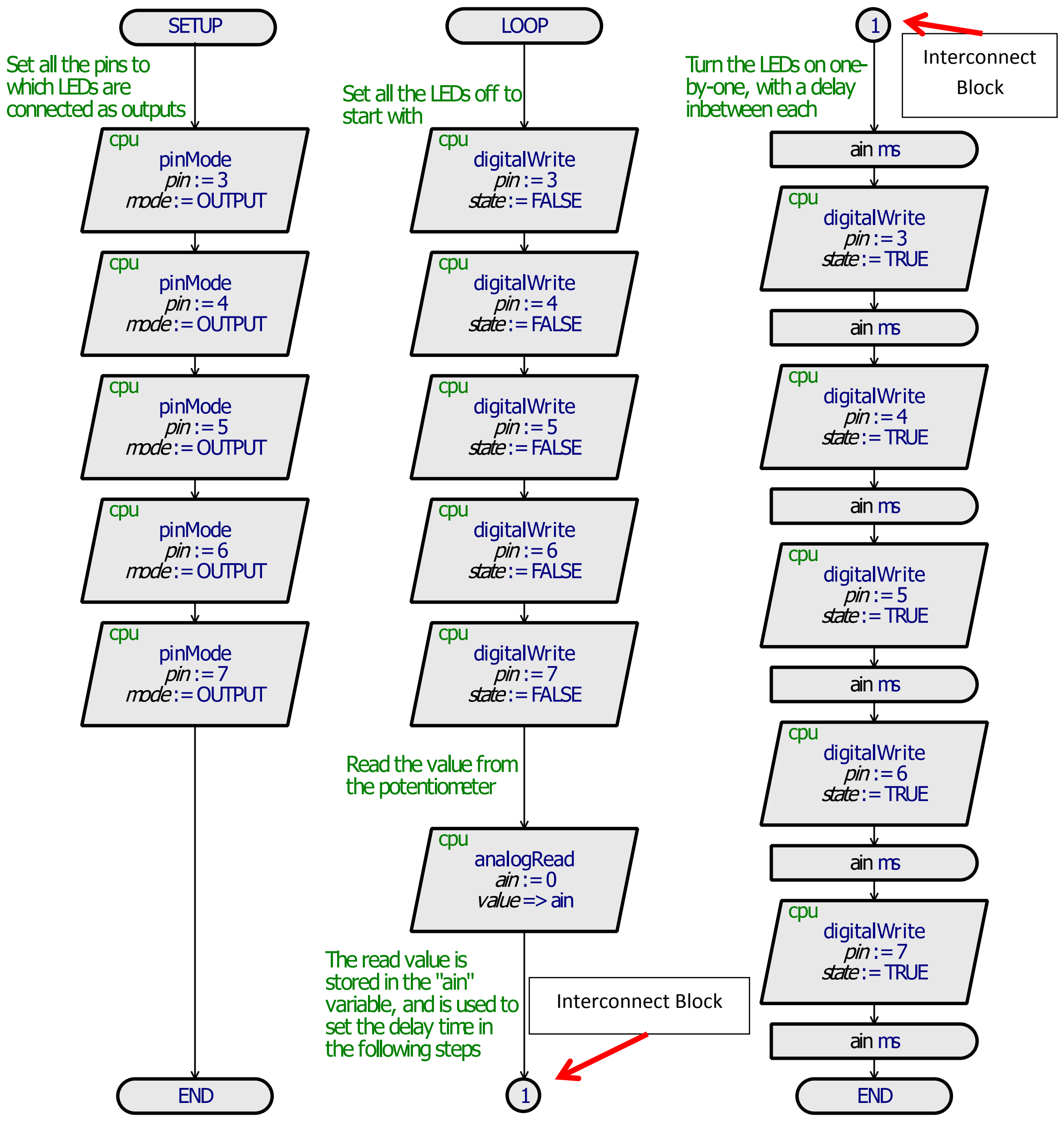
cpu
digitalWrite
pin := 3
state := TRUE

Peripheral
Object: cpu Method: digitalWrite
Arguments
Pin: 3
State: TRUE

Figure 7-9: "digitalWrite" method of the cpu peripheral

Remember that, to the microcontroller, “TRUE” means the same thing as “On” (or “High”) – in this case “output 5V”; and “FALSE” means the same thing as “Off” (or “Low”) – in this case “output 0V”.

7.6. The Flowchart



SETUP
Set all the pins to which LEDs are connected as outputs
cpu pinMode pin := 3 mode := OUTPUT
cpu pinMode pin := 4 mode := OUTPUT
cpu pinMode pin := 5 mode := OUTPUT
cpu pinMode pin := 6 mode := OUTPUT
cpu pinMode pin := 7 mode := OUTPUT
END
LOOP
Set all the LEDs off to start with
cpu digitalWrite pin := 3 state := FALSE
cpu digitalWrite pin := 4 state := FALSE
cpu digitalWrite pin := 5 state := FALSE
cpu digitalWrite pin := 6 state := FALSE
cpu digitalWrite pin := 7 state := FALSE
Read the value from the potentiometer
cpu analogRead ain := 0 value => ain
The read value is stored in the "ain" variable, and is used to set the delay time in the following steps
Interconnect Block
1
1
Interconnect Block
Turn the LEDs on one-by-one, with a delay inbetween each
ain ms
cpu digitalWrite pin := 3 state := TRUE
ain ms
cpu digitalWrite pin := 4 state := TRUE
ain ms
cpu digitalWrite pin := 5 state := TRUE
ain ms
cpu digitalWrite pin := 6 state := TRUE
ain ms
cpu digitalWrite pin := 7 state := TRUE
ain ms
END

Figure 7-10: Sequencing LEDS flowchart

Figure 7-10 depicts the flowchart program for this project. Don’t worry about drawing the flowchart just yet, we will get around to that shortly; for now, let’s just study it to see how it works:

The steps in the programs are as follows.

1. Set all of the pins which the LEDs are connected to as outputs. This is done using the “pinMode” method (section 7.5) – one block for each pin which we would like to set as an output. The blocks are placed in the **Setup** routine, since this action only needs to be done once (at startup). Note that it is not necessary to explicitly set the pin which the

potentiometer is connected to as an input, as all pins are inputs by default (until you set them as an output).

2. In our **Loop**, we start by having all the LEDs switched off. This is done by using digitalWrite blocks to set the state of each pin to "FALSE" ("Off").
3. Next we read the analogue value from the pin connected to the potentiometer (analogue pin 0, or AD0), and store it in a variable called "ain". This is done using an "analogRead" block (available under the cpu peripheral).
4. After jumping through the "Interconnect" (to be discussed shortly), the next step is to delay for a certain amount of time; the amount of time to delay being specified by the "ain" value which we read from the potentiometer. After the delay, switch the first LED on (by setting the pin which it is connected to with a state of "TRUE", meaning "On", using the digitalWrite method again). This "delay-then-switch-on" process is then repeated for each LED (from pin 3 to pin 7).
5. The program then automatically goes back to the start of the loop, and the LEDs are switched on again from the start (step 2) – this process repeats indefinitely (forever).

Some things to note about the flowchart / program:

- Comment blocks have been added, to help explain to the reader what is happening in the program. These comment blocks do not affect the program at all, but are simply there as "notes" for a human reader.
- The flowchart features an "interconnect block". Because the flowchart routine is too long to fit in the page / sheet (without changing the sheet size), we can use an interconnect block to "split it up". The procedure for doing so is illustrated in the following figure (taken from the Visual Designer help file, note that it illustrates a different flowchart to the one which we are busy with).

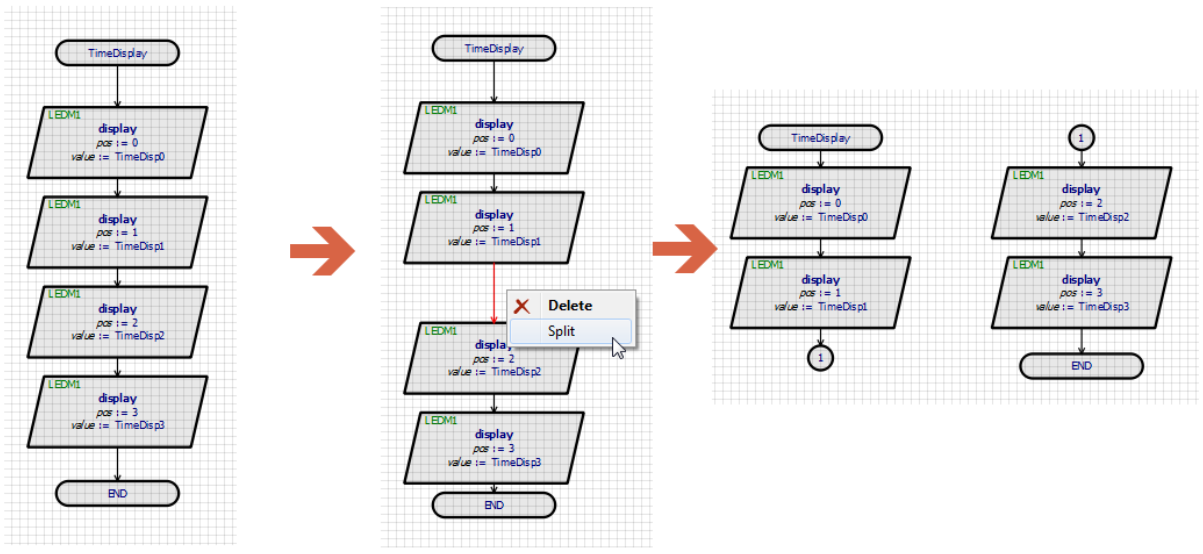


Figure 7-11: Splitting into two separate lines with interconnect blocks

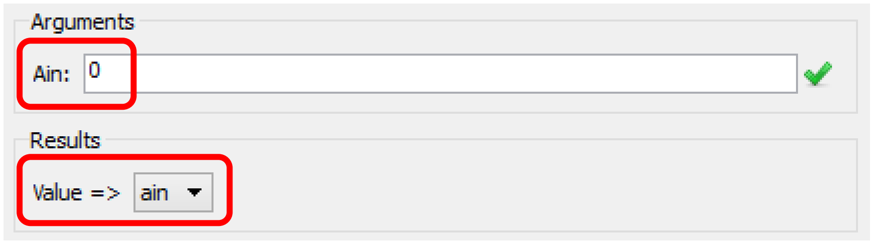
- Interconnect blocks with the same number are considered to be joined. Interconnect blocks can be inserted by right-clicking on a flowline and then clicking "split", or by manually dragging the interconnect blocks onto the flowchart (from the list of available blocks to the left of the flowchart editing area).

- The program only reads the potentiometer, and thus only sets the sequencing speed, at the start of each sequencing cycle. Thus, if you change the potentiometer whilst the sequencing is in progress, then the speed change will only take effect at the beginning of the next cycle (when the potentiometer is read again). The flowchart could be changed so that the potentiometer is read more frequently; so that the speed change could be implemented mid-cycle (you are welcome to try implementing this if you like, by inserting the relevant analogRead block before each delay).

Note that, even with interconnect blocks, it is possible to run out of space on the sheet. This can be overcome however by adding additional sheets (click the Project menu and then select New Sheet) – interconnect blocks can then be used to connect between sheets. There are also further methods available for better organising larger flowcharts, such as sub-routines, and these will be covered in the next chapter.

We will now proceed to constructing our flowchart so that it matches the one in Figure 7-10.

1. Drag 5 x pinMode methods / blocks from the cpu peripheral into the flowchart, placing them in the Setup routine. Set the "pin" and "mode" arguments of the blocks so that they set pins 3 to 7 as outputs (as per Figure 7-10). Remember that the arguments of a block can be changed by right-clicking on the block, and then selecting "Edit" from the popup menu.
2. Drag 5 x digitalWrite methods / blocks from the cpu peripheral into the flowchart Loop routine. Set the arguments of these so that they will switch all of the pins (3 to 7) off ("FALSE").
3. Drag an analogRead method / block from the cpu peripheral into the flowchart Loop routine, placing it after the digitalWrite blocks which were added in the previous step.
4. Edit the analogRead block. Set the "Ain" argument to 0 – this is the analogue pin which we would like to read from ("AN0"). Create a new variable called "ain" (this may have been automatically created already), and select for the result of the analogRead to be stored in this variable (as per the following figure).

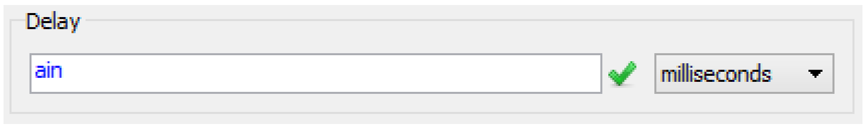
   Arguments

   Ain: 0

   Results

   Value => ain

   Note that the similarity between the argument name "Ain" and the variable name "ain" is purely coincidental – we could name the variable "potentiometer_reading" for example.
5. Split the flowchart by right-clicking on the flowline below the analogRead routine, and selecting "Split" from the popup menu.
6. Add a delay block as the next step in the Loop routine (inserting it after the "interconnect"), setting its delay to "ain" milliseconds.



This will delay for the amount of milliseconds specified by the “ain” variable.

7. Add a digitalWrite method / block as the next step in the Loop routine, in order to switch the first LED on. Set it’s pin and state arguments to 3 and TRUE respectively.
8. Add another delay and digitalWrite method / block for each of the remaining 4 LEDs, including a final additional delay block at the end. We now have a sequence of blocks which will switch the LEDs on one-by-one, with a delay inbetween (the time of which is determined by the “ain” variable, which was set by reading the potentiometer voltage).

Once done ensure that your flowchart looks functionally like Figure 7-10 (remembering that comment blocks, for example, do not affect the function of the flowchart), but do not run the simulation just yet.

7.7. Active Popups

When running a flowchart simulation, the components which the program uses (buttons, LEDs, etc) can be seen and interacted with on either the Visual Designer tab or the Schematic Capture tab. When components are seen on the Visual Designer tab during simulation, Proteus is in fact just “copying” certain parts of the schematic into the Visual Designer tab. The way in which Proteus knows which parts of the schematic to copy over, is by using something called “Active Popups”. Active Popup areas can be seen on the schematic as blue labelled rectangles, as illustrated in the following figure.

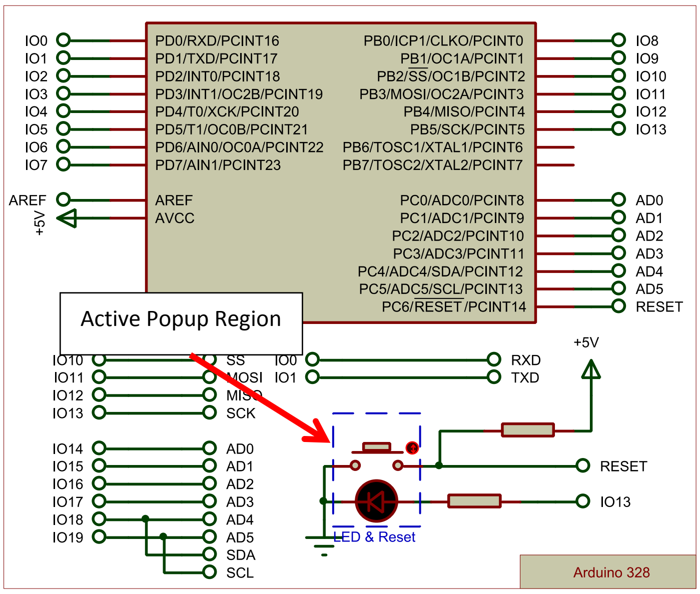


Figure 7-12: Active Popup

You can also add your own Active Popups to your schematic using the Active Popup tool mode (), which is available from the vertical toolbar. To add an Active Popup, simply select Active Popup mode and then draw a rectangle over the area which you would like to create an Active Popup over. You can then also give the Active Popup a unique label (name) to identify it (by right-clicking on it and then clicking Edit Properties). If you like, then you could try adding Active Popups to your schematic, as illustrated in the following figure.

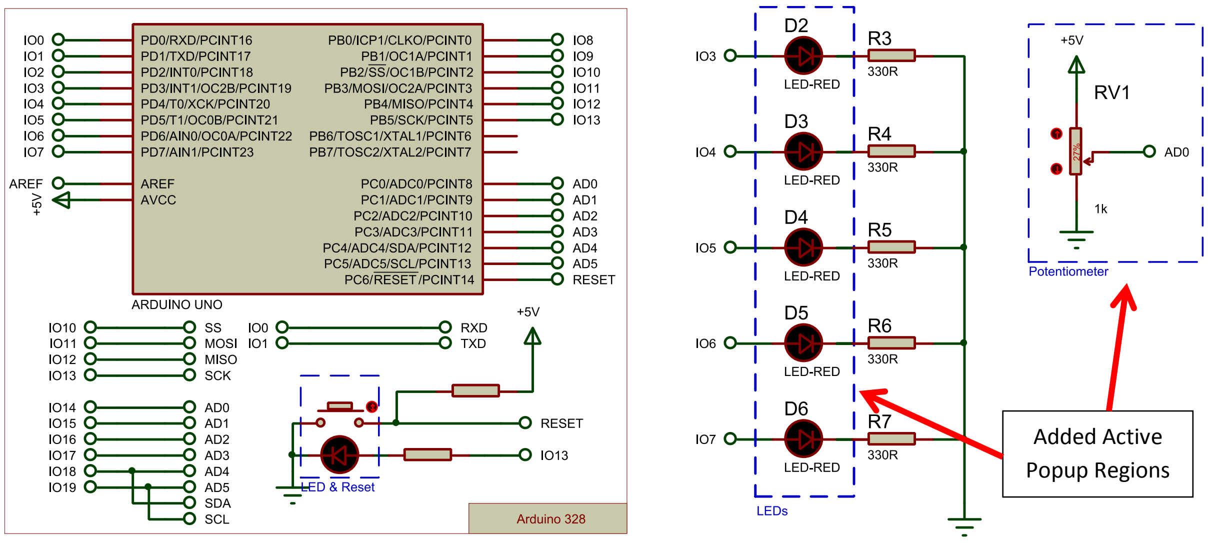


Figure 7-13: Schematic with Active Popup regions added

If the Active Popup regions are added to your schematic then you will be able to view and interact with these regions from the Visual Designer tab (when the simulation is running). Either way, you will be able to view and interact with the components from the Schematic Capture tab.

(Tip: If an Active Popup region is not appearing in the Visual Designer tab when the simulation is running, then try clicking the Debug menu – if it is listed there, then clicking on it in the menu will make it visible again. Note that Active Popups can also be moved, resized and detached (into separate windows) within the Visual Designer tab (when the simulation is running).)

7.8. Testing the Project

Once the flowchart has been completed as per Figure 7-10, and optionally Active Popup regions added, the "Play" button (blue triangle in the bottom-left corner of the screen) can be pressed to test the simulation. You should see the LEDs flashing in sequence, and changing the setting on the potentiometer (using the arrows, or by clicking on it) should change the speed (rate) of the flashing. If anything doesn't work as expected, then it is recommended to go through this chapter again to check if anything has been missed or set incorrectly.

7.9. Chapter Summary

Using the skills taught in this chapter, it is possible to draw exact custom circuits to interact with our microcontroller (as opposed to being limited to the selection of peripherals available from the Peripheral Gallery).

8. Sub-Routines and Conditional Loops

Congratulations! If you have completed and understood the previous flowchart design chapters, then you now have all the tools needed to solve any programming problem which can be solved with a flowchart in Visual Designer[19]! This chapter introduces two new (but separate) concepts: sub-routines, and conditional loops. Whilst these two new concepts do not enable us to do anything which we could not do before, they do make it easier to do so.

We will be using both sub-routines and conditional loops to better organise more complex flowcharts. They will enable us to make the flowcharts shorter, simpler, and easier to read / understand.

(Note: Sub-routines and counting loops are separate concepts (they are not related). We will however be taking the opportunity to introduce both of them in just one chapter, instead of having a separate chapter for each.)

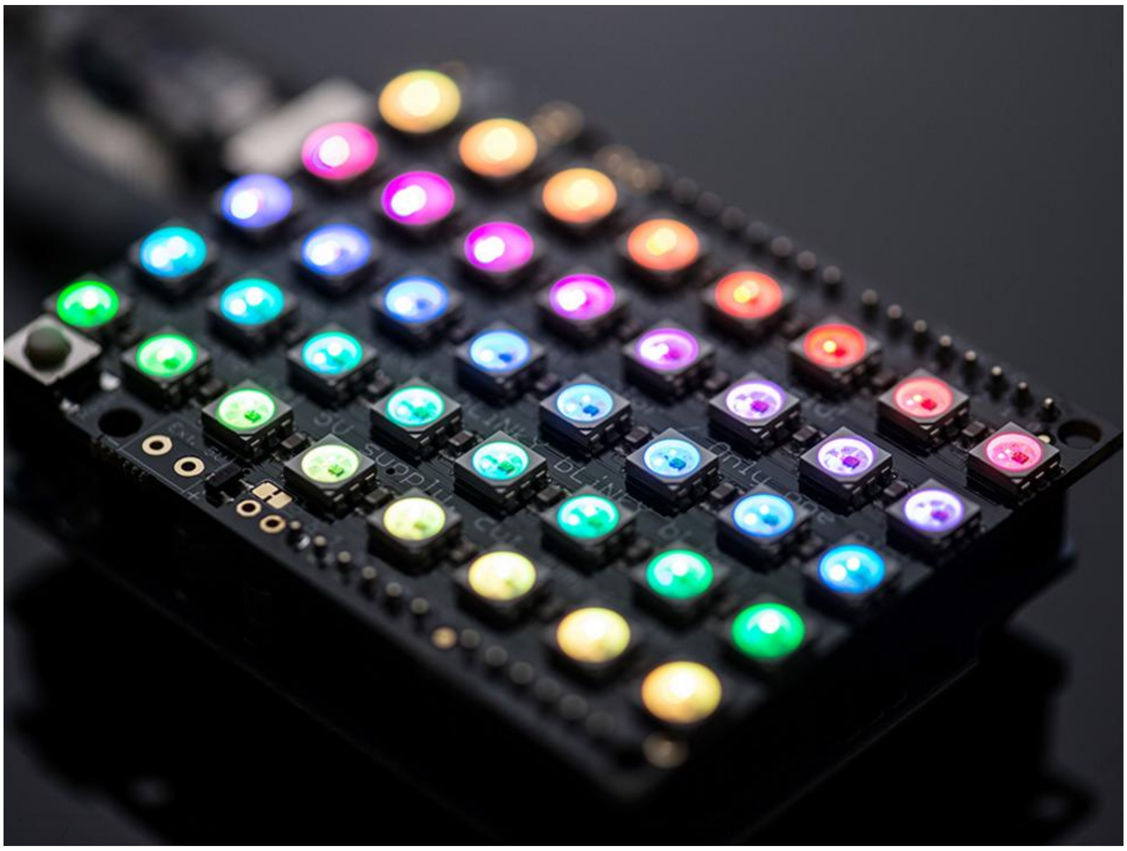


Figure 8-1: Adafruit NeoPixel shield for Arduino[20]

We will be using the “Adafruit NeoPixel shield for Arduino” in this chapter. A NeoPixel is a fancy LED, the colour of which can be precisely controlled. The NeoPixel shield features 40 of these LEDs, in 8 columns and 5 rows (or 5 columns and 8 rows, depending on how you look at it – this setup can also be called an “array” or “matrix”). The “shield” (a name used to describe some Arduino add-on peripheral boards) plugs in on top of the Arduino Uno. We will start off with a small project which

[19] Technically, the advanced topic of “Interrupts” has not yet been covered. Suggestions for further learning on this subject are made in section 12.1.
[20] https://www.adafruit.com/product/1430

switches just one of the NeoPixels on, and then build on that project till we have all of the NeoPixels on and changing colours. In the process we will introduce and explain sub-routines and conditional loops. You may wish to save the project under a different filename at each completed "step", in case you would like to keep a copy of each of these "steps".

8.1. Setting up the Starter Project

1. Start a new flowchart project. If you need a reminder on how to do so, then please see section 2.2, 3.3 or 4.1 (remember to save ( ) any work which you may have open and don't want to lose, close ( ) any open project, and then select "New Flowchart" from the Home ( ) screen). The project filename can be "NeoPixel1".
2. Add an "Adafruit NeoPixel Shield" to the project from the Peripheral Gallery in Visual Designer. If you need a reminder on how to add a peripheral to the project, then please see section 3.4. (The Peripheral Gallery can be accessed by right-clicking the Project menu and then selecting Add Peripheral. The "Adafruit NeoPixel Shield" can be found under the Adafruit category.)

After adding the "Adafruit NeoPixel Shield" peripheral from the Peripheral Gallery, the Visual Designer Projects Area should contain a peripheral with a number of methods available for controlling the NeoPixels (as illustrated in the below figure).

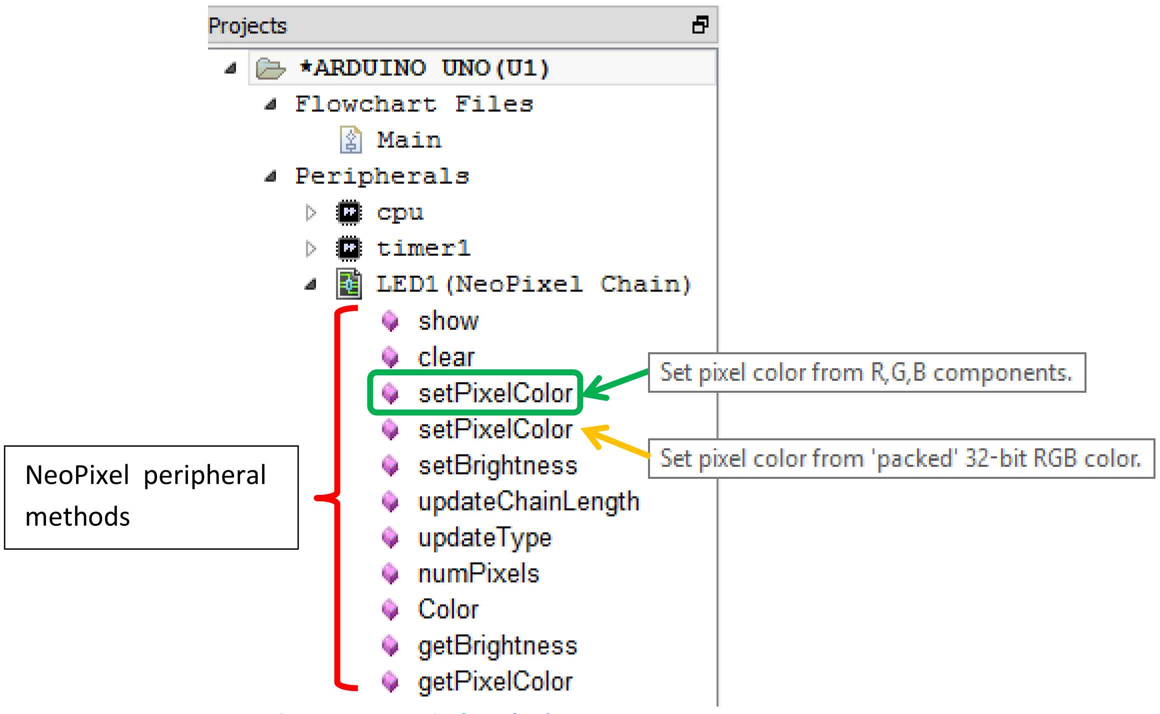


Figure 8-2: NeoPixel methods

8.2. Constructing Colours

The colour of each of the 40 NeoPixels on the NeoPixel shield can be exactly controlled by specifying that colour as a combination of red, green and blue. The combination of colours to form another colour is a principle of light, and is similar to the way in which you can mix different colours of paint together to make new colours (e.g. red and blue paint mixed together makes purple paint, etc).

Inside each NeoPixel are actually three LEDs – a red, green and blue LED. When the coloured light generated by each of these 3 separate LEDs is combined (or "mixed"), different colours of light can be made. To start with, we will be keeping things simple and using only completely red, completely green or completely blue colours. Similar to the way in which the analogue-to-digital converter (ADC) in the previous chapter had a maximum "scale" of 1023, the colour settings for the LEDs have a maximum scale of 255, and so each colour can be set in a range from zero to 255 (0-255), where 0 is fully off and 255 is fully on.

8.3. Setting a Single Pixel

For the first of our project steps, we will set the colour of a single NeoPixel. This can be done using the "setPixelColor" method of the NeoPixel peripheral (Figure 8-2). Note that there are *two* "setPixelColor" methods available, each offering a different way of specifying the colour to be set – we will be using the top method, which sets the colour by specifying RGB components (hovering your mouse over each method should result in a tool-tip being displayed which provides more information regarding it). Drag the top setPixelColor method from the list of NeoPixel methods (Figure 8-2) into the Loop routine of the flowchart. Also drag the "show" method onto the flowchart, inserting it after the setPixelColor method. You will know that you have used the correct setPixelColor method if the resulting flowchart block looks like the one in the following figure.

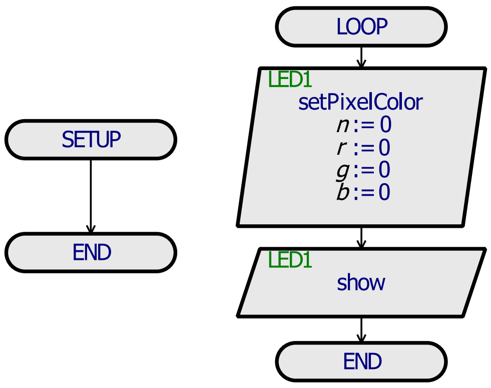
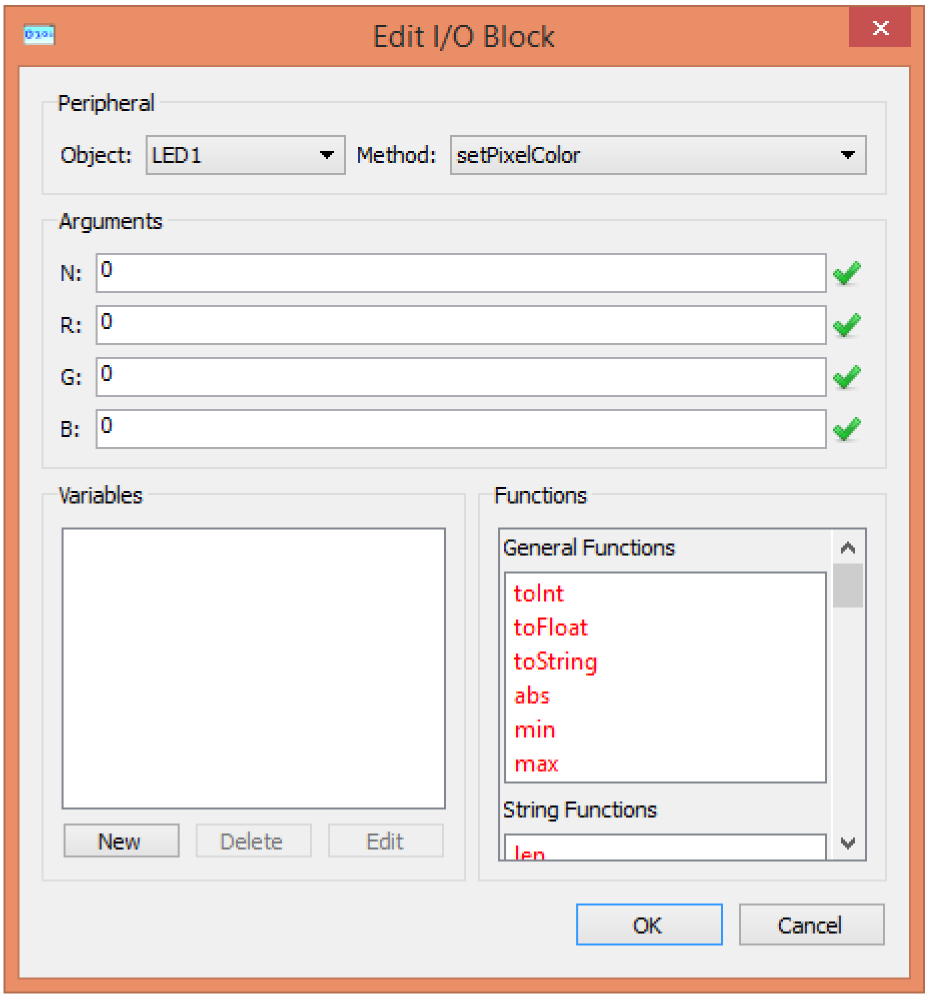


Figure 8-3: setPixelColor method inserted into flowchart Loop routine

The setPixelColor method (obviously) sets the colour of a NeoPixel. Note that this does not "take effect" however until the "**show**" method is called (used). The NeoPixel peripheral works this way because we may want to "prepare" our display by setting the colours of a number of pixels (requiring multiple setPixelColor flowchart blocks) first, and then only send those updates to the display once they're all ready; this way the updates all "take effect" at the same time (rather than one-by-one). In programming-speak this technique is called "buffering"; the setPixelColor method updates the buffer – which is the "plan" of what the pixels will all be set to next – and the show method sends that buffer to the NeoPixels.

To specify which NeoPixel we would like to update, and to what colour, right-click on the newly added "setPixelColor" flowchart block and select "Edit" from the popup menu (or left-click on the flowchart block twice). The Edit I/O Block dialogue for the setPixelColor block appears.


Edit I/O Block

Peripheral

Object: LED1 Method: setPixelColor

Arguments

N: 0

R: 0

G: 0

B: 0

Figure 8-4: Edit I/O block dialogue for the setPixelColor method

Four arguments (or parameters) are available: “N”, “R”, “G” and “B”.

- “N” is the number of the pixel, from zero (0) to 39. Computers often use something called a “zero-based-index”, which means that they start counting something from zero (0) instead of from 1. Because of the way in which computers work inside, using this technique helps them to be more efficient. There are 40 NeoPixels. If we started counting them from 1, then the first pixel would be pixel # 1 and the last pixel would be # 40. Because the computer starts counting them from zero however, the first pixel is called pixel # 0 and the last pixel is called pixel # 39. The NeoPixels on our shield in Proteus are arranged in rows, in a zig-zag fashion which starts at the top-left and ends at the bottom right (this is illustrated in the following figure).
- R is the red component of the light, specified as a number from zero (0) to 255.
- G is the green component of the light, specified as a number from zero (0) to 255.
- B is the blue component of the light, specified as a number from zero (0) to 255.

Remember that the red, green and blue colours combine to form a single colour. Red and green combined (at the same intensity), for example, would produce yellow.

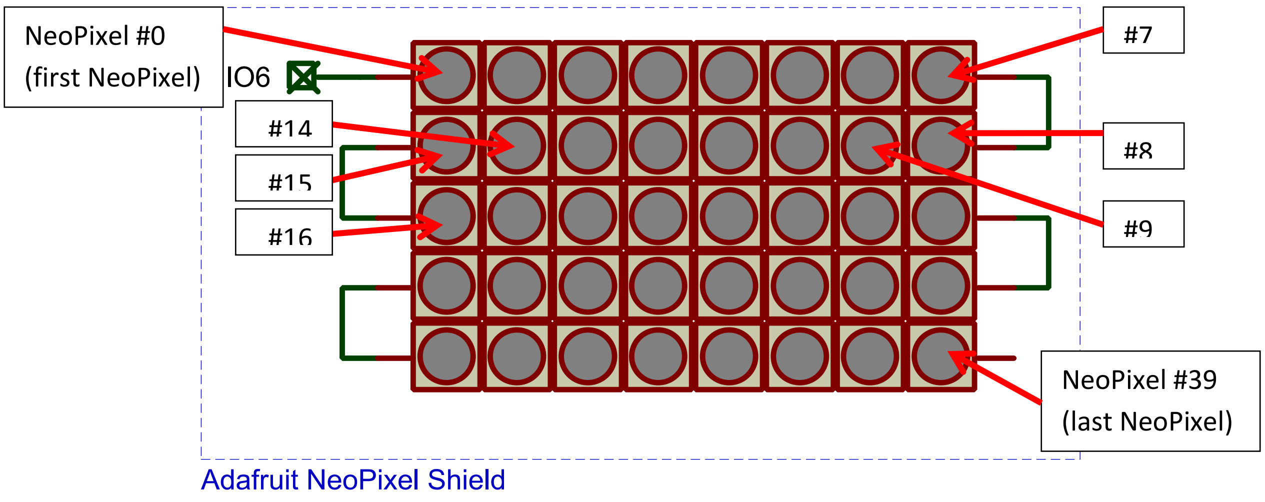


Figure 8-5: NeoPixel shield schematic component

Leave the number of the pixel ("N") as 0. Set the intensity of the red colour ("R") to 255 (full intensity), and leave the intensities of the green and blue colours ("G" and "B") as 0 (off). This should produce a red colour on the first NeoPixel.

Once ready, run the simulation (click "OK" to accept the settings in the setPixelColor dialogue before trying to run the simulation, if you have not done so already). NeoPixel #0 should turn on with a red colour.

8.4. Setting 3 Pixels

Next, let's set the colour of NeoPixels #1 and #2 as well, so that the first 3 pixels will all be red.

If you would like to save each completed step of this project under a new filename, then click the File menu and then select "Save Project As". The new filename "NeoPixel2" can be used.

Drag two more setPixelMethods onto the flowchart, above the "show" method. Set the parameters for these methods so that they set pixels 2 and 3 to a red colour. The following figure illustrates the flowchart.

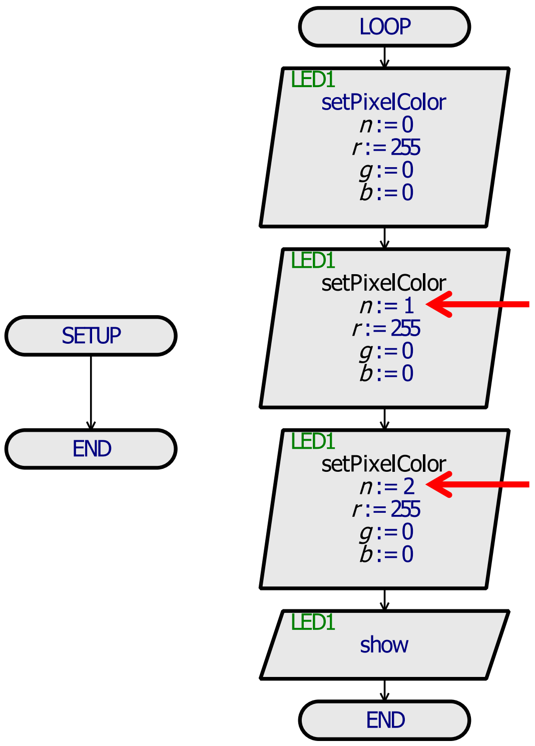


Figure 8-6: Flowchart to set 3 pixels

8.5. Setting 40 Pixels

What if we would like to set all 40 of the NeoPixels as red? Do we need to drag 40 setPixelColor blocks onto the flowchart? We could drag 40 setPixelColor blocks onto the flowchart, set each of them, and it would work. There is a much easier way to set all 40 pixels though.

If you would like to save each completed step of this project under a new filename, then click the File menu and then select “Save Project As”. The new filename “NeoPixel3” can be used.

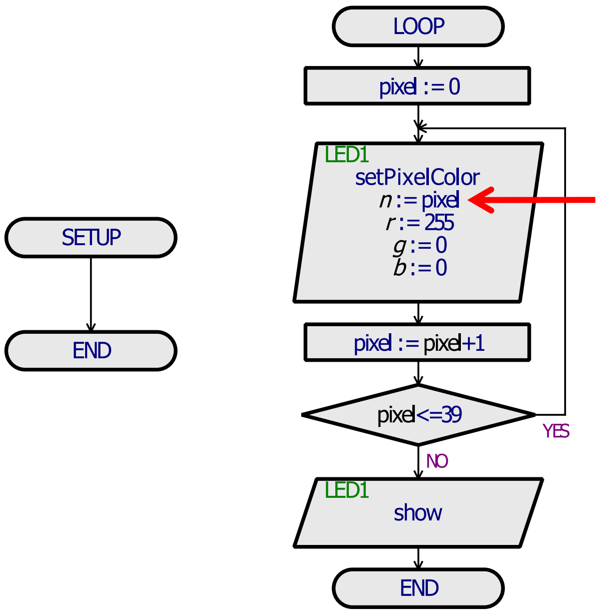


Figure 8-7: Flowchart to set 40 pixels

The previous figure illustrates a flowchart which sets all 40 pixels, using only 1 setPixelColor block (rather than 40). The flowchart works by using a variable to specify which pixel should be set, and looping that variable from 0 to 39 in order to set all 40 pixels. The program works by first setting the pixel number to zero (0), and then setting the colour of the pixel specified by the pixel number variable (which is currently set to 0). Next, it adds one to the value of the pixel number variable, so that the variable now equals 1. The program then checks if the pixel number variable is less than or equal to 39 – if it is ("YES") then it routes back to setting the colour of the pixel specified by the pixel number variable (which is now set to 1). This process continues until all of the pixels from 0-39 have been set, at which point the value of the pixel number variable will become 39 + 1 = 40. 40 is not less than or equal to 39, so at the decision block the program then routes in the "NO" direction and proceeds to the "show" block so that the set NeoPixel colours will all take effect. Once you have set your flowchart up as per the previous figure, run the simulation in order to check that the program works as expected (all 40 NeoPixels set to a red colour). If you need a reminder on how to add a variable to your program, please see section 4.2.

8.6. "For" Conditional Loops

Because the need to count a variable through a sequence of numbers occurs quite often in programming, there is a "shortcut" for it – called a "For" loop (or "For-Next" loop).

If you would like to save each completed step of this project under a new filename, then click the File menu and then select "Save Project As". The new filename "NeoPixel4" can be used.

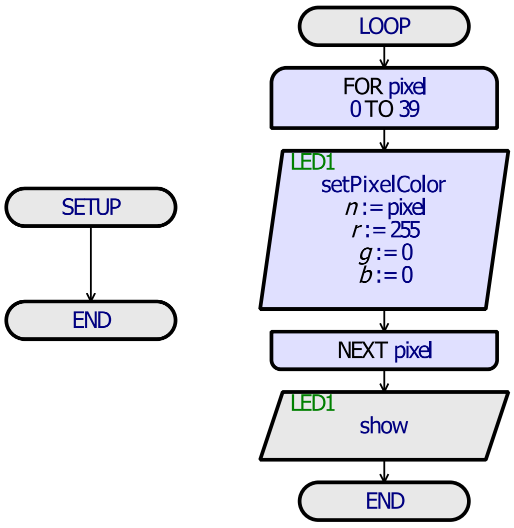


Figure 8-8: Setting 40 pixels using a "For" loop

The flowchart illustrated in the previous figure (Figure 8-8) does exactly the same thing as the flowchart illustrated in the previous section (Figure 8-7), however it is quicker and easier to design, and is tidier and easier to understand.

We will now update your flowchart to use a "For" loop, as per the previous figure.

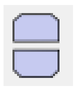
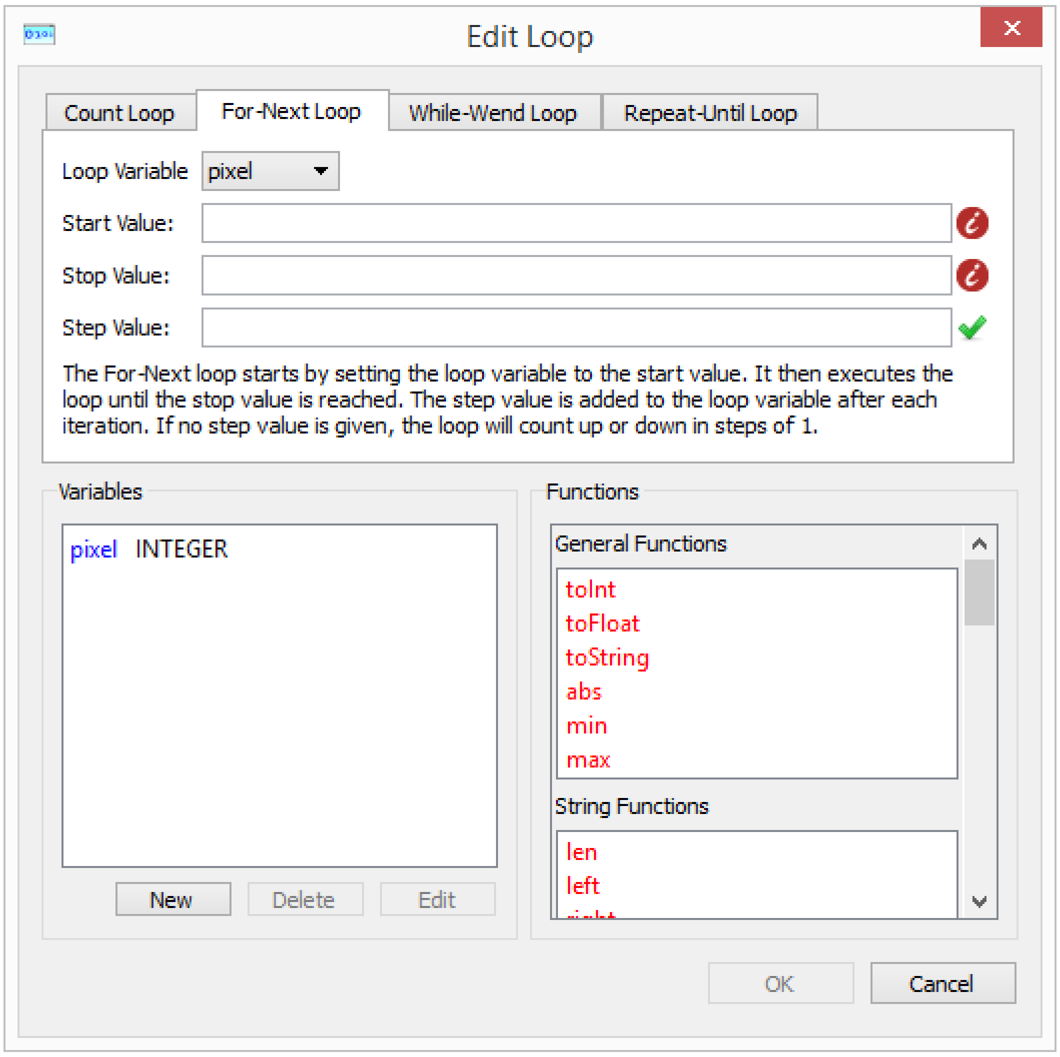


Figure 8-9: Loop construct

To start with, drag a "Loop Construct" onto your flowchart. Next, right-click on the newly added "LOOP" blocks and select "Edit" from the popup menu (alternatively click twice on the "LOOP" blocks). The following figure depicts the Edit Loop dialogue.


Edit Loop

Count Loop | For-Next Loop | While-Wend Loop | Repeat-Until Loop

Loop Variable: pixel

Start Value:

Stop Value:

Step Value:

The For-Next loop starts by setting the loop variable to the start value. It then executes the loop until the stop value is reached. The step value is added to the loop variable after each iteration. If no step value is given, the loop will count up or down in steps of 1.

Variables

pixel INTEGER

Functions

General Functions: toInt, toFloat, toString, abs, min, max

String Functions: len, left

Figure 8-10: "Edit Loop" dialogue box

From the tabs at the top we can see that there are a number of different types of loops available. All of the different types of loops will be discussed shortly (they are all similar), however for now we will be using the For-Next loop (which is the default option).

The Loop Variable is the variable which will be counted (in our case, the pixel number variable).

The Start and Stop values are self-explanatory – in our case they will be 0-39.

The Step value determines how the Loop Variable is adjusted on each loop iteration. In our case we would like the pixel number variable to increase by 1 each time, so the step value would be 1. 1 is also the default if nothing is specified, so if we leave the step value blank then that will also work. If we specified a step value of, for example, 2 then the pixel number variable would increase by 2 on each loop iteration (0, 2, 4, 6, 8, etc).

Make sure that “pixel” is selected as the loop variable, set the Start Value to 0, End Value to 39, and Step Value to 1 (or leave the Step Value blank). Click “OK” to close the dialogue.

Manipulate the flowchart so that it looks like the one for this section (as per Figure 8-8). Remember that you can “Tear Off” flowchart blocks, in order to reposition them within the flowchart, by right-clicking on them and then selecting “Tear Off” (or by holding down the Ctrl key on the keyboard and then clicking on and dragging them).

Once done, run the simulation and verify that all 40 NeoPixels still light up red, as per the previous program.

8.7. Types of Conditional Loops

8.7.1. “While” Loops

“While” loops (or “While-Wend” loops, “Wend” being short for “While End” – meaning the end of the while loop section) continue looping so long as a certain condition is met. For example, “While Button1 is pressed, repeat these actions” (another way to say this would be “So long as Button1 is pressed, repeat these actions”). The following figure illustrates two equivalent methods of accomplishing the same task, by using a decision block or by using a while loop. The task is, “While Button1 is still pressed, add 1 to the count variable”.

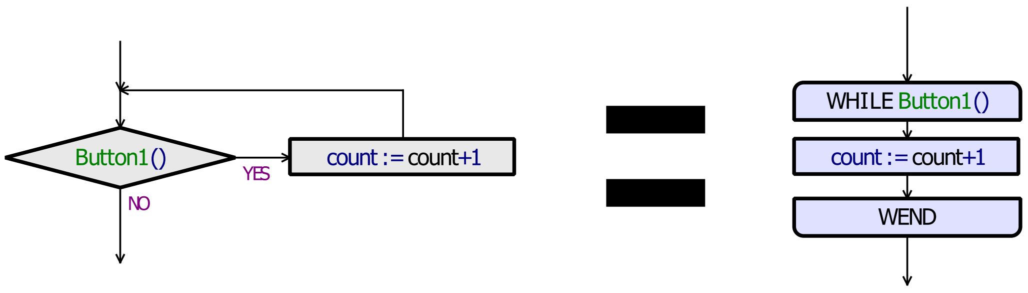


Figure 8-11: While-Wend loop

8.7.2. “Repeat” Loops

“Repeat” loops (or “Repeat-Until” loops) are very similar to While loops, except that the condition is checked *after* the actions to be repeated, rather than before them. For example, “Repeat these actions, until Button1 is not pressed anymore”.

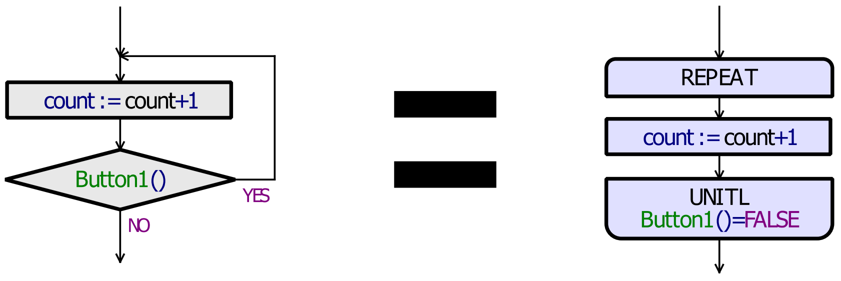


Figure 8-12: Repeat-Until loop

The previous figure illustrates two equivalent methods of accomplishing the same task, by using a decision block or by using a repeat-until loop. The task is, “Add 1 to the count variable, until Button1 is not pressed anymore”. Note that “Repeat-Until” loops use the opposite logical check from a

"While" loop ("Repeat-Until" loops check for a FALSE condition, whilst "While" loops check for a TRUE condition). Thus we "repeat until the button is **not** pressed anymore", as opposed to "repeat while the button is pressed". The word "not" inverts logic, converting true into false and vice versa (for example, "the tea is hot" – "tea hot = TRUE", or "the tea is not hot", "tea hot = FALSE").

8.7.3. "For" Loops

The "For" loop (or "For-Next" loop) has been covered in the previous project steps. A variable is counted from a start value to an end value, with the option to change the "step" size of the counting (e.g. counting from 0-10 with a step size of 1 will go [0,1,2,3,4,5,6,7,8,9,10], counting from 0-10 with a step size of 2 will go [0,2,4,6,8,10]. Although not necessary to understand, you may be interested to know that internally (inside the computer), the For loop is actually converted into a while loop. So "For count = 0 to 10 step 2" works internally as "Start count at 0. Add 2 to it each time and continue looping so long as it is less than or equal to 10". The equivalence can be seen in the previous project steps.

8.7.4. "Count" Loops

"Count" loops are a simplified form of a "For" loop, which assume that the Step size is 1 and that the counting starts at zero (0). So "For count = 0 to 39 step 1" would be the same as "Count 40". If in doubt, use a "For" loop instead.

8.8. Adding More Colours

Next we will add the displaying of green and blue colours to our program. After setting all NeoPixels as red, we will then set them green (after a short delay), and then set them blue (again after a short delay). The process will then repeat (once again, after a short delay).

If you would like to save each completed step of this project under a new filename, then click the File menu and then select "Save Project As". The new filename "NeoPixel5" can be used.

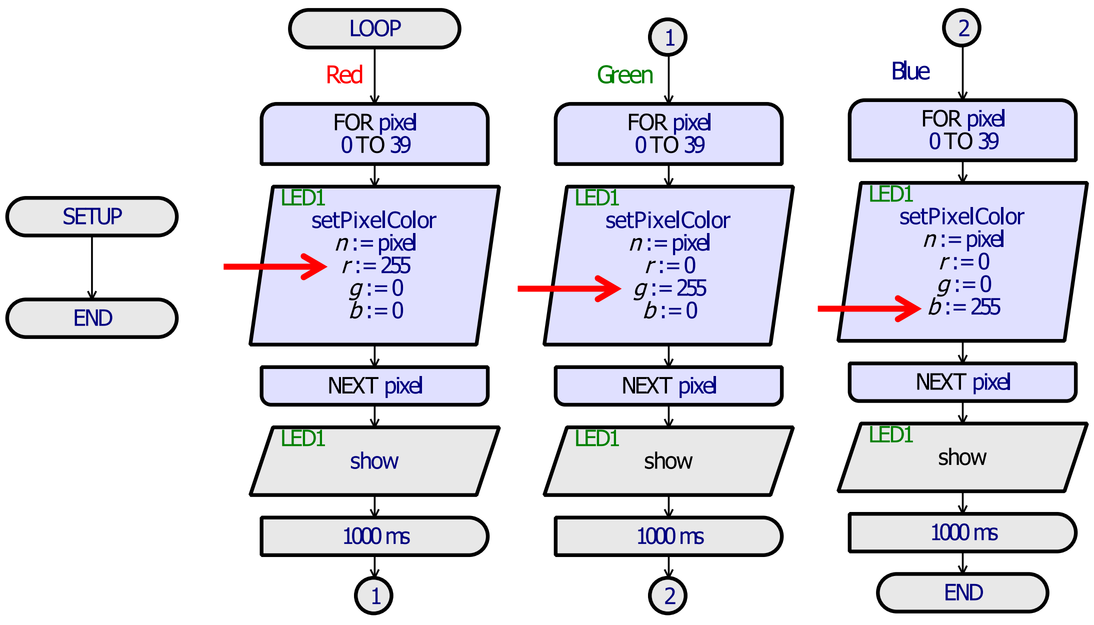


Figure 8-13: Flowchart for setting all NeoPixels to red, green and then blue

The previous figure illustrates the flowchart which we will build for this project step. We essentially need to make two more copies of our existing flowchart routine (note that Visual Designer does include copy-and-paste facilities, available from the right-click popup menu or from the main Edit menu once one or more flowchart blocks have been selected). We also need to adjust the parameters in the copied setPixelColor methods in order to set the different colours, and add a delay in-between each colour change. In order to get the flowchart to fit within the space available, it is also necessary to add some Interconnect Blocks (please see the previous chapter covering these if you need a reminder on how to use them). Note that the “Red”, “Green” and “Blue” comments above the For loops are for human readers only and do not affect the operation of the program. Once you have updated your flowchart so that it looks like the one in the previous figure (as mentioned, the comment blocks are optional), run the simulation and check that it behaves as expected. The NeoPixels should first all turn red, then after 1 second they should all turn green, and after another 1 second they should all turn blue. After a further 1 second delay the process repeats, and continues indefinitely.

8.9. Sub-Routines

Notice how we had to reproduce, or copy-and-paste, a section of flowchart blocks in the previous project step. Whenever you find yourself reproducing, or copying-and-pasting, sections of a program, then you should know that it may be better to use a sub-routine instead. As mentioned, sub-routines, like conditional loops, are completely optional and you can complete a program without them. They do however make it *easier* to build more complex programs. The section of flowchart which we have reproduced in the previous project step is relatively short (4 or 5 blocks, depending on whether you count the “next” in the For loop as a separate block), but as the size of the section which is being reproduced becomes bigger (perhaps 10, or 20 or 50 blocks), sub-routines start to present more and more of an advantage. For this project step, we will place the reproduced sections of the flowchart (which update the pixels) into a separate sub-routine.

If you would like to save each completed step of this project under a new filename, then click the File menu and then select “Save Project As”. The new filename “NeoPixel6” can be used.

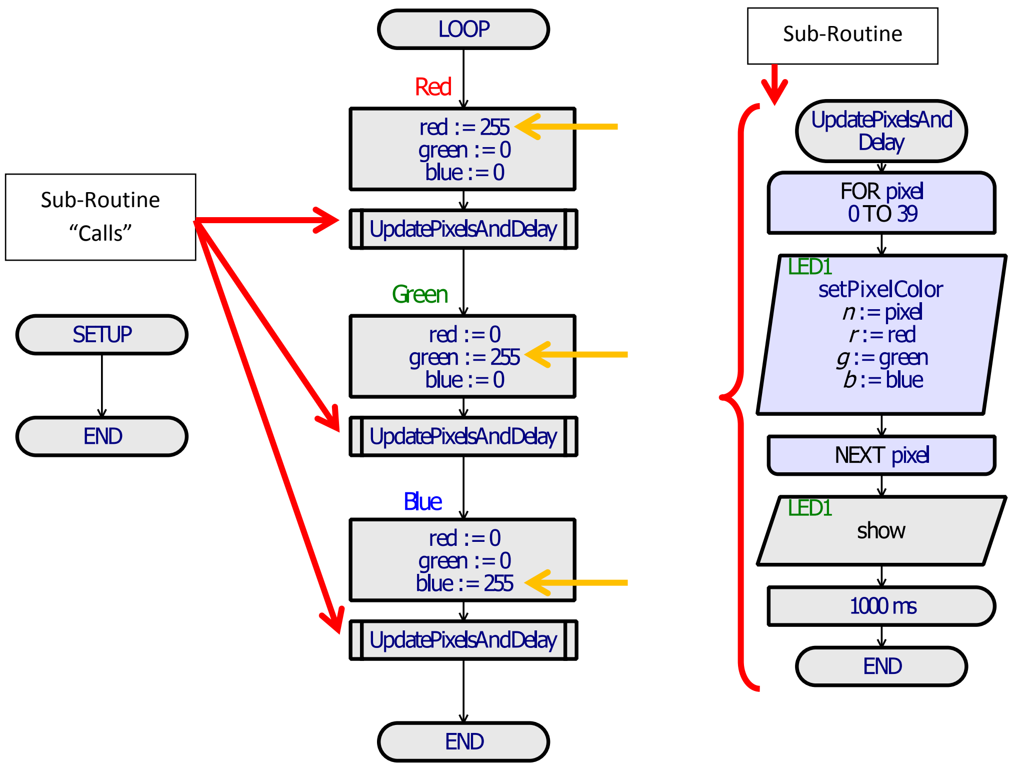


Figure 8-14: Flowchart with sub-routine

The previous figure illustrates the flowchart for this project step. Notice how the reproduced sections of the flowchart from the previous project step have been copied out and replaced by a single copy in the sub-routine called “UpdatePixelsAndDelay”. The next thing to notice in the flowchart is that we have added three new variables, called “red”, “green” and “blue”. Because we now only have one copy of the For loop which set all the NeoPixels to a particular colour, we need a way to tell it which colour to use. We can do this using variables (the colour “varies” each time, hence “variables”).

In order to update your project so that it looks like the previous figure, you will need to add the “UpdatePixelsAndDelay” subroutine. To do so, drag an Event Block onto the flowchart.

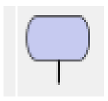


Figure 8-15: Event (sub-routine) block

Note that an “End” block is also automatically added. Change the name of the event block to “UpdatePixelsAndDelay”, by right-clicking on it and selecting “Edit” from the popup menu (or by clicking on it twice), then changing the name appropriately.

Move one of the flowchart sections which updates all the pixels from the main loop into the new subroutine (remember the “Tear Off” function from section 2.7.3), as per the previous flowchart figure. Delete the other two copies of the flowchart section (as well as the interconnect blocks).

Add a new Assignment Block into the now-empty main loop. Edit the assignment block and create 3 new variables; “red”, “green” and “blue”. Assign values to these variables as illustrated in Figure

8-14 (red:=255, green:=0, blue:=0). If you need help remembering how to create variables and assign values to them, then please see section 4.2.

Drag a "Sub-Routine Call" block into your main loop, after the variable assignment block.

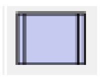


Figure 8-16: Sub-Routine Call block

Edit the properties of the Sub-Routine call block, and select the "UpdatePixelsAndDelay" method as the sub-routine to call.

Add two more variable assignment blocks and sub-routine calls, as illustrated in Figure 8-14. Note that the comments in the figure are optional.

Update the setPixelColor flowchart block so that it uses the newly created "red", "green" and "blue" variables for the "r", "g", and "b" values (i.e. instead of using "255" for the "R" argument, use "red" (<u>without</u> quotes)).

The flowchart has now been adapted so that there is a sub-routine which can be used to update all of the NeoPixels (and delay for 1 second after doing so), using the "red", "green" and "blue" variables to set the colours of the NeoPixels. From the main loop, we simply set the colour we want using the variables, and then call the "UpdatePixelsAndDelay" sub-routine.

Once ready, run the simulation and check that it behaves as expected. If not, then carefully compare your flowchart to the previous flowchart figure.

Sub-routines can be difficult to understand, but just remember that whenever you see a sub-routine call in a flowchart, it is as-if you copied-and-pasted the flowchart blocks contained in the subroutine into the flowchart where the sub-routine call is (in other words, replaced the sub-routine call block with the blocks contained in that sub-routine).

Note that many of the peripheral methods which we have been using till now, including the setPixelColor method, are in fact sub-routine calls themselves! The only difference is that the details of those sub-routines are hidden from us within Proteus, so that we don't need to worry about the details of them.

From an understandability (conceptual) point of view, it is generally easier to "forget" about the exact details of how a subroutine works, including sub-routines which we make ourselves. The sub-routine should be named well, so that we can understand from its name what it does, without looking at the details of it. This is another big advantage of sub-routines, even if they are not used to replace duplicated code – they can help to break a program up into smaller bits and pieces, which are much easier for the human brain to understand. For example, it is much easier to say "please make some tomato soup, bread, salad and chocolate cake" than it is to specify the exact steps needed to perform each of those tasks. Each of those tasks have their own recipe, or "sub-routine", which we can summarise by using their name. Imagine if we had to try and make the same request, specifying the exact steps needed! "Please take some tomatoes, chop them up, put some water in a pan", etc, etc – it would take forever and become really confusing for the human brain to understand.

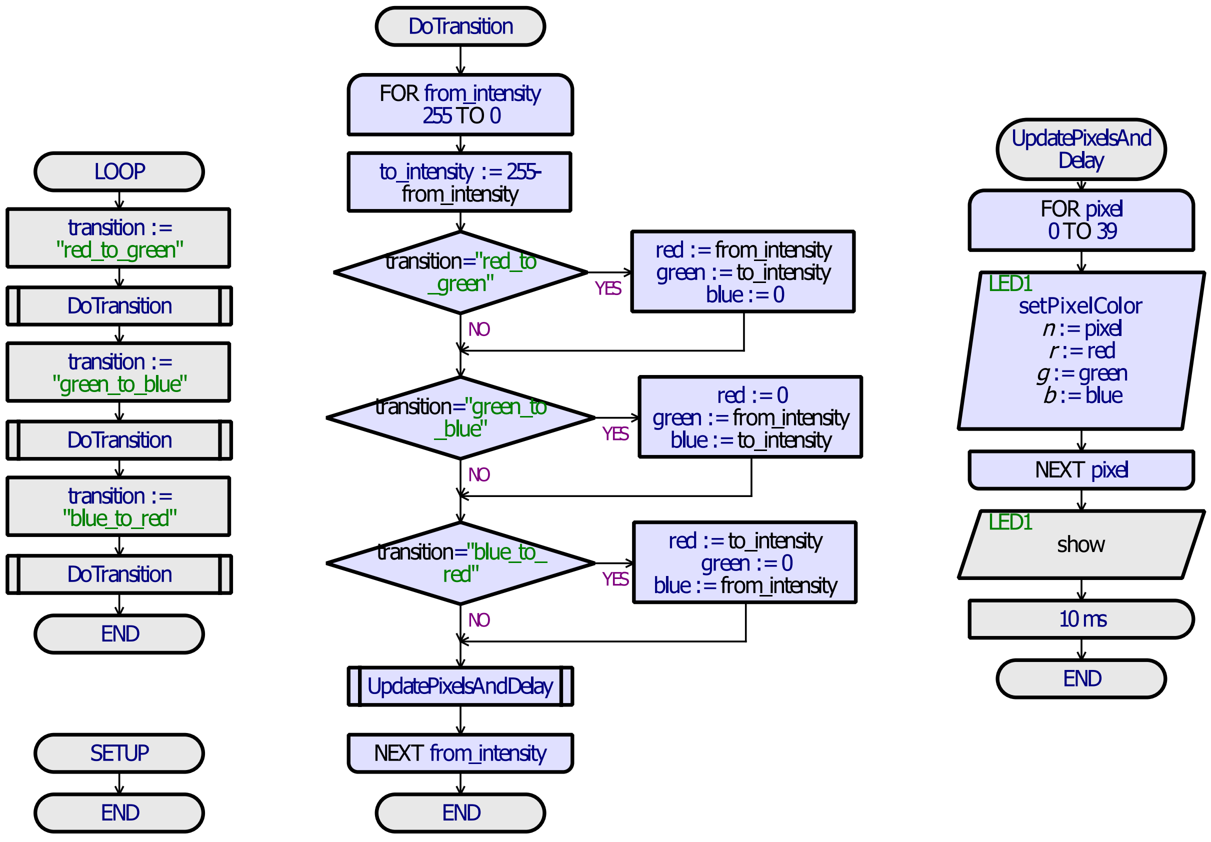
Sub-Routines thus serve two purposes:

1. Replacing repetitive sections of a program.
2. Breaking a program up into smaller bits, which makes it easier for a human to understand.

8.10. Getting Fancy

Using the new skills of sub-routines and conditional loops which we have just learned, we will now construct a flowchart which cycles the NeoPixels smoothly between all the colours of the rainbow!

If you would like to save each completed step of this project under a new filename, then click the File menu and then select "Save Project As". The new filename "NeoPixel7" can be used.



LOOP
transition := "red_to_green"
DoTransition
transition := "green_to_blue"
DoTransition
transition := "blue_to_red"
DoTransition
END
SETUP
END
DoTransition
FOR from_intensity 255 TO 0
to_intensity := 255- from_intensity
transition="red_to _green"
YES
red := from_intensity green := to_intensity blue := 0
NO
transition="green_to _blue"
YES
red := 0 green := from_intensity blue := to_intensity
NO
transition="blue_to_ red"
YES
red := to_intensity green := 0 blue := from_intensity
NO
UpdatePixelsAndDelay
NEXT from_intensity
END
UpdatePixelsAnd Delay
FOR pixel 0 TO 39
LED1 setPixelColor n := pixel r := red g := green b := blue
NEXT pixel
LED1 show
10 ms
END

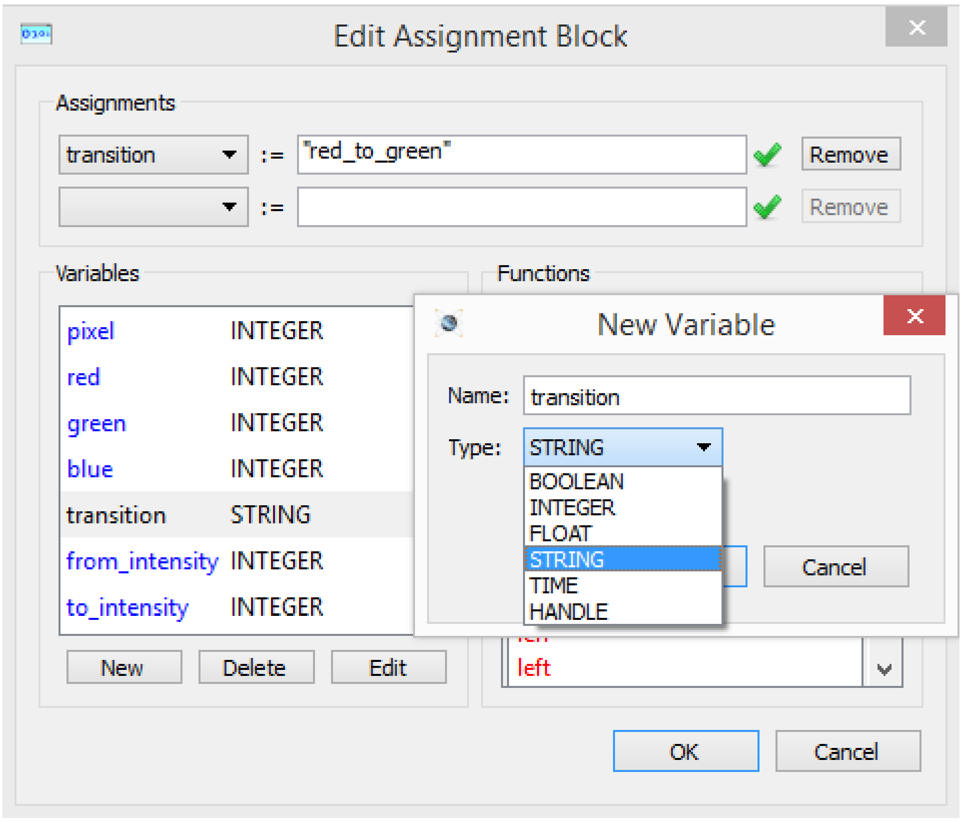
Figure 8-17: Rainbow flowchart

The previous figure depicts the flowchart for this final project step. A new routine called "DoTransition" has been added, which handles the transition from red to green, green to blue, and blue back to red. In the process, all the colours of the rainbow are displayed. Three new variables have been added – two new Integer (number) type variables called "from_intensity" and "to_intensity", and a string (text) variable called "transition".

The "DoTransition" sub-routine counts the "from_intensity" from 255 (full strength intensity) down to 0 (no intensity, or "off"). The "to_intensity" variable is set to the inverse (opposite) of the "from_intensity" variable, using the equation "to_intensity = 255 - from_intensity". Thus if “from_intensity” is 255, then to_intensity will be “255 minus (-) 255”, which equals zero. If “from_intensity” is 0, then to_intensity will be “255 minus (-) 0”, which equals 255. The result is that as from_intensity is counted from 255 down to zero (0), to_intensity will simultaneously be counted from 0 up to 255. You can write down and calculate each of the values using the equation if you are not yet convinced of this. Intensity means the same thing as “brightness”, so the brightness of the “from” colour is slowly decreased whilst the brightness of the “to” colour is slowly increased. In order to know which colour to assign the “from_intensity” to and which colour to assign the “to_intensity” to, the sub-routine looks at the “transition” variable, which can be one of 3 cases – “red_to_green”, “green_to_blue” or “blue_to_red”. Depending on the case, the sub-routine will choose which way to assign the intensities. Once the intensities have been assigned, the “UpdatePixelsAndDelay” sub-routine is called in order to update all of the NeoPixels with the new colour. The delay in the “UpdatePixelsAndDelay” subroutine has been decreased to 10ms (10 milliseconds), so that the transition can take place at a reasonable speed. If the delay were left at 1000ms (1000 milliseconds, or 1 second) then it would take 255 x 1 second, or 4 minutes and 15 seconds, to complete a single transition. By setting the delay to 10ms it takes 255 x 10ms, which equals 2550ms or 2.55 seconds, to complete a transition.

The main loop has been updated to simply set the “transition” variable to the name of the transition which should be performed, and then call the “DoTransition” sub-routine.

Update your flowchart to match the one from the previous figure. If you get stuck, then please check the relevant previous sections of this book. The following diagram illustrates the detail regarding the “transition” STRING type variable.



transition := "red_to_green"

pixel	INTEGER
red	INTEGER
green	INTEGER
blue	INTEGER
transition	STRING
from_intensity	INTEGER
to_intensity	INTEGER

Name: transition
Type: STRING

Figure 8-18: String variable creation and assignment

Once ready, run the simulation. The NeoPixels should smooth transition between all the colours of the rainbow. If the program does not work as expected, then carefully compare it to the previous flowchart figure (Figure 8-17).

Hopefully you are impressed by the result achieved with this final flowchart program step! This result could have been achieved using only setPixelColor blocks, but without the use of conditional loops or sub-routines it would take 3 x 255 x 40 = 30,600 blocks to achieve this. That is thirty-thousand-six-hundred blocks. We have managed to achieve this with only 19 blocks.

There are many more possibilities for things which could be displayed on the NeoPixel shield. For example, we have been setting the whole shield to a single colour at the same time, but each pixel could in fact be set / animated individually. With some imagination and ingenuity, many more projects could potentially be created.

9. Introduction to C

Note: Depending on your preferences, you may wish to complete “Bonus Chapter: Robotics” before completing this chapter.

Congratulations! If you have successfully completed and understood the preceding chapters, then by now you have acquired a new skill: the ability to write a computer software program. “C” is a programming language which can be used to build software programs, just like Visual Designer can be used to build software programs. This is a matter of syntax however: you could instruct somebody how to bake a cake in English, or you could instruct somebody how to bake a cake in French, but the actions required would still be the same. In much the same way, C is simply a different “language” for constructing computer programs. Whereas Visual Designer uses flowchart blocks to construct a program, C programs are written purely in text (letters, numbers, words, etc). As you will see however (as C is introduced), it is easier and less intimidating to get started with programming using flowcharts than it is with C!

C is an “industry standard” programming language, which means that it is very commonly used within the software development industry. The advantages of learning C thus include:

- You will be able to find C compilers for many more devices besides for Arduino.
- There are many C resources available, such as “libraries”. A library is a set of sub-routines, which somebody else has already written, and which can help you to accomplish a task (without “reinventing the wheel”). Visual Designer includes libraries itself, in the form of peripherals which can be picked from the Peripheral Gallery (section 3.4).

C is also a more powerful programming language than flowchart programming. In much the same way as loops and sub-routines made it easier to construct more complex programs in chapter 8, C includes features which can help to better organise more complex programs. C++, a more advanced version of C, includes even more features.

You may be interested to know that Proteus, including Visual Designer, is written mostly written in C++. You may also be interested to know that Proteus converts all Visual Designer flowcharts into C++ code during the compilation process.

This chapter aims to introduce you to the C language. Please note that this chapter does not attempt to be a thorough “C” course, as that would be a book in itself, but simply introduces “C” and points you in the right direction for further C learning.

Note that the intention of this chapter is not to “replace” the flowchart programming skills which have been learned in the previous chapter. The reasons for considering C are as just listed, and depending on your circumstances these may or may not apply to you; you can potentially carry on using flowcharts to design programs indefinitely.

9.1. Comparing C and Flowcharts

To introduce the C language, we will start by creating a simple flowchart project, and then creating the equivalent project in C.

To get started, please begin a new flowchart project (you should be familiar with the process by now –remember to save any work which you may have open and don’t want to lose). The filename

“C_BlinkLED_Flowchart” can be used for the project. Next, use methods from the CPU peripheral to construct the following flowchart (setting parameters for the flowchart blocks as necessary).

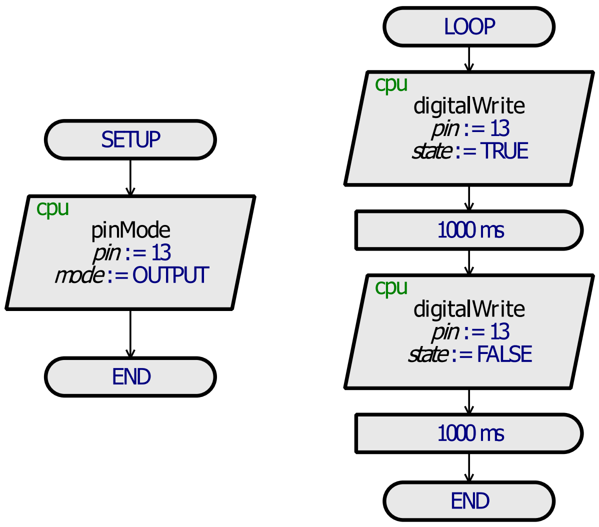
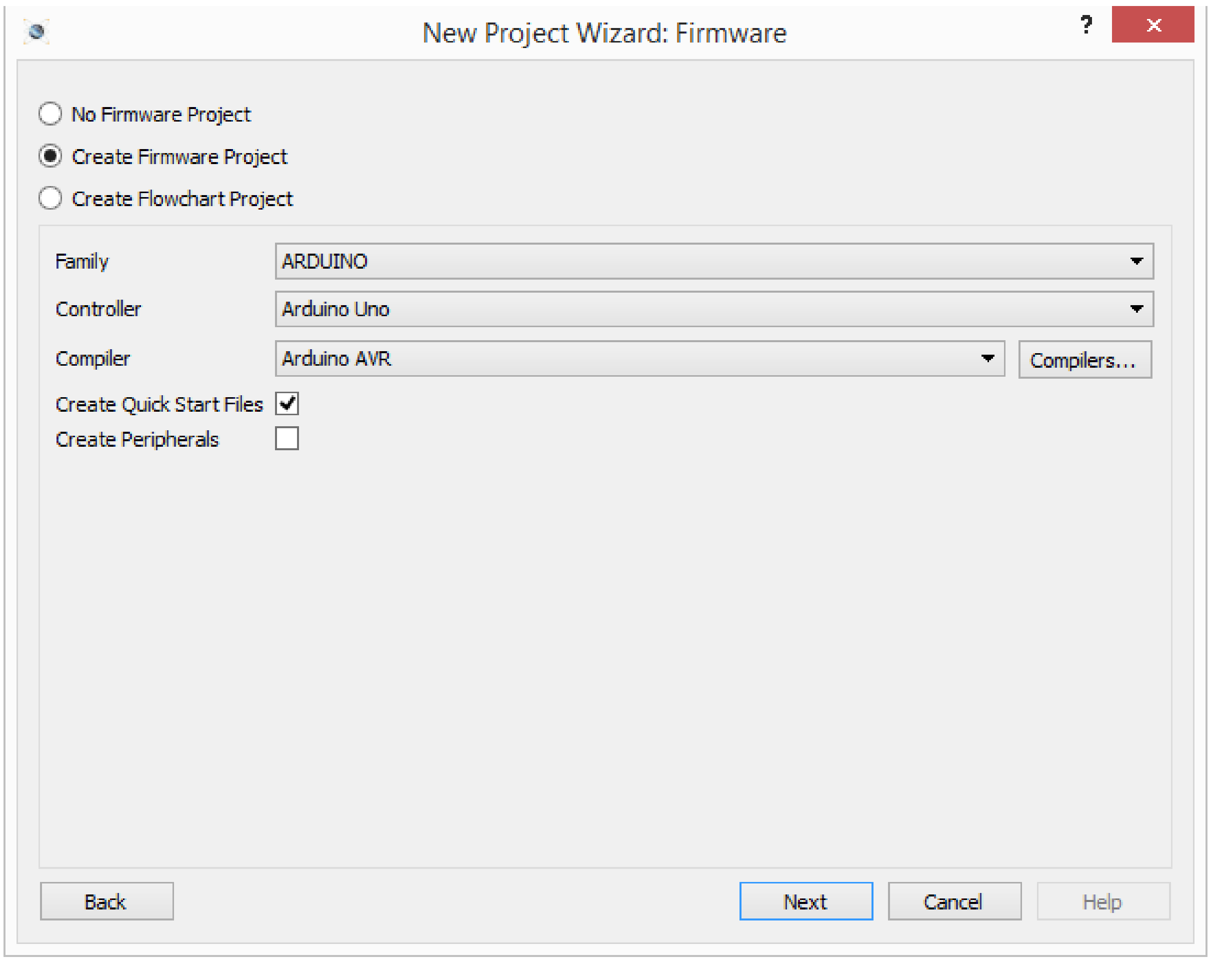


Figure 9-1: Blink LED flowchart

This flowchart sets pin 13 of the microcontroller as an output, and then proceeds to alternating that output between high (true / on) and low (false / off), with a 1 second (1000 millisecond) delay in-between. Since the built-in LED on the Arduino UNO is connected to pin 13, this flowchart will flash the onboard LED on and off with a 1 second delay in-between. Run the simulation to check that the flowchart works as expected.

Save the current flowchart project, and then click the File menu and select “New Project” in order to start a new project. Complete the New Project Wizard as per the following steps:

1. Project name “C_BlinkLED_Code”. (Click “Next”.)
2. Specify to create a schematic from the DEFAULT template. (Click “Next”.)
3. Do not create a PCB layout. (Click “Next”.)
4. Specify to create a new Firmware project. Select ARDUINO as the family (Arduino Uno as the controller and Arduino AVR as the compiler should then be set by default). Select to “Create Quick Start Files”, but do not “Create Peripherals”. This is illustrated by the following figure. (Click “Next”, and then “Finish” when done.)



No Firmware Project
[✓ Create Firmware Project]
Create Flowchart Project

Family	ARDUINO
Controller	Arduino Uno
Compiler	Arduino AVR
Create Quick Start Files	✓
Create Peripherals	☐

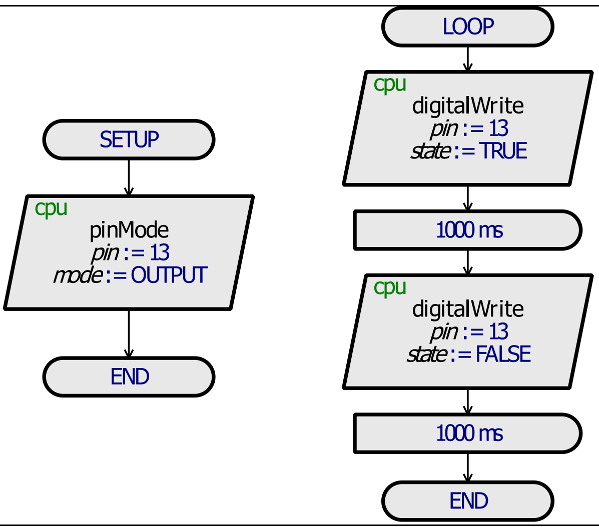
Figure 9-2: New firmware project

Double click on the “main.ino” file on the left in order to open it, if it is not already open. You should now have a text file open, containing code similar to the following:

```
/* Main.ino file generated by New Project wizard
 *
 * Created:   Sun Jan 14 2018
 * Processor: Arduino Uno
 * Compiler:  Arduino AVR
 */

void setup () {

// TODO: put your setup code here, to run once:
}

void loop() {

// TODO: put your main code here, to run repeatedly:
}
```

All of the code in green is comments which do not affect the operation of the program, and can be safely ignored. That leaves us with “void setup”, “void loop” and some brackets. “Setup” and “Loop” correspond to our flowchart routines of the same name. The following code illustrates the equivalent C code for our flowchart (the flowchart is listed again alongside for comparison).

“C” Code	**Flowchart**

```
void setup () {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```
	 SETUP
	cpu
	pinMode
	pin := 13
	mode := OUTPUT
	END
	LOOP
	cpu
	digitalWrite
	pin := 13
	state := TRUE
	1000 ms
	cpu
	digitalWrite
	pin := 13
	state := FALSE
	1000 ms
	END

The similarities between the C code and the flowchart should be obvious, remembering that “HIGH” means the same thing as “TRUE” and “LOW” means the same thing as “FALSE”. Note that the comments have been removed from the C code. Copy the code into your program, and once ready run the simulation (which will now simulate the C code); check that the results are the same as those obtained with the flowchart project (LED toggling on and off with a 1 second delay). If any problems are encountered, then please ensure that your code exactly matches the code listed here.

9.2. C Language Syntax Introduction

C code requires correct syntax in order for code to compile correctly (otherwise, the compiler will report an error). A flowchart program and the equivalent C code are listed below – the C code provides examples of the syntax rules which will be covered next. For interests' sake, the program flashes an LED whenever a button is pressed, incrementing the number of flashes each time.

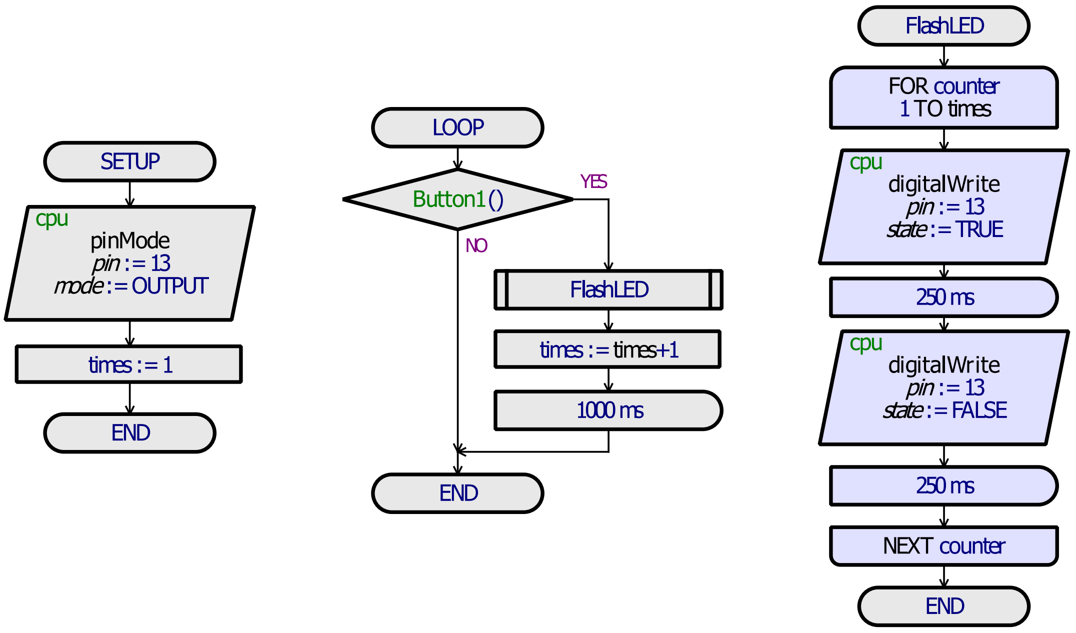
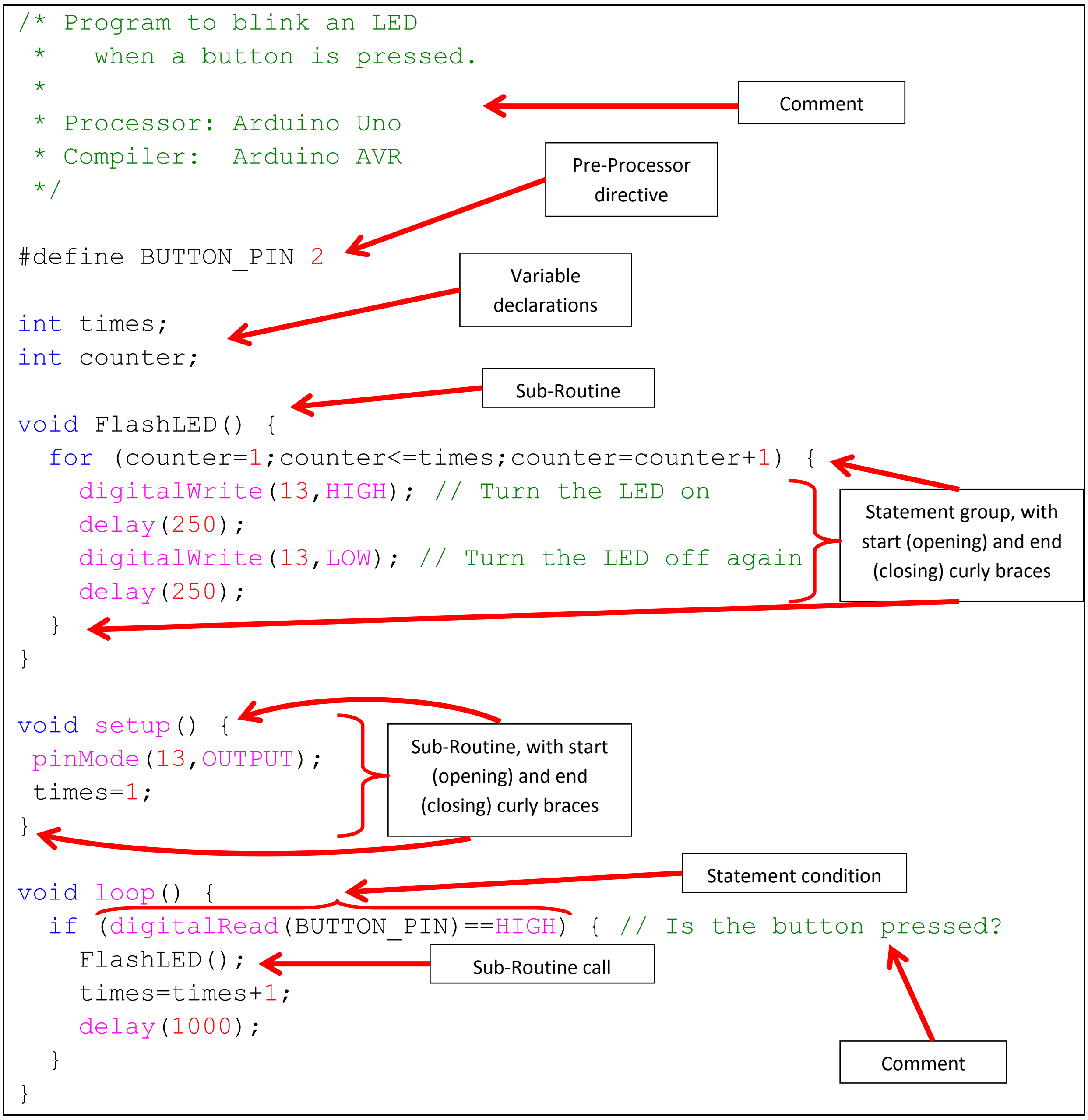


Figure 9-3: "C" language syntax introduction flowchart



```
/* Program to blink an LED
 *   when a button is pressed.
 *
 * Processor: Arduino Uno
 * Compiler:  Arduino AVR
 */

#define BUTTON_PIN 2

int times;
int counter;

void FlashLED() {
  for (counter=1;counter<=times;counter=counter+1) {
    digitalWrite(13,HIGH); // Turn the LED on
    delay(250);
    digitalWrite(13,LOW); // Turn the LED off again
    delay(250);
  }
}

void setup() {
 pinMode(13,OUTPUT);
 times=1;
}

void loop() {
  if (digitalRead(BUTTON_PIN)==HIGH) { // Is the button pressed?
    FlashLED();
    times=times+1;
    delay(1000);
  }
}
```

Comment
Pre-Processor directive
Variable declarations
Sub-Routine
Statement group, with start (opening) and end (closing) curly braces
Sub-Routine, with start (opening) and end (closing) curly braces
Statement condition
Sub-Routine call
Comment

C Code 1

The C syntax rules demonstrated in this program are discussed below. Please note that this program by no means demonstrates *all* C features, but it does demonstrate the fundamentals.

9.2.1. Comments

Comments are information which is added to a program for human readers only, and do not affect the operation of the program (they are ignored by the computer). Comments in C can be specified in two ways.

9.2.1.1. Bounded Comments

Bounded comments are bounded by a start and an end marker. The start marker is "/*" (forward-slash, star) and the end marker is "*/" (star, forward-slash). Any text in-between the start and end marker is considered to be a comment, and is ignored by the computer. Bounded comments can span multiple lines, or even be inserted in-between code on a single line.

```
/* This is
   a multiline
   comment */
```

9.2.1.2. *Rest-of-the-line Comments*

"Rest-of-the-line" comments are started by "//" (forward-slash, forward-slash), and continue to the end of the line. No end marker is needed.

```
delay(1000); // Delay for 1,000 milliseconds = 1 second
```

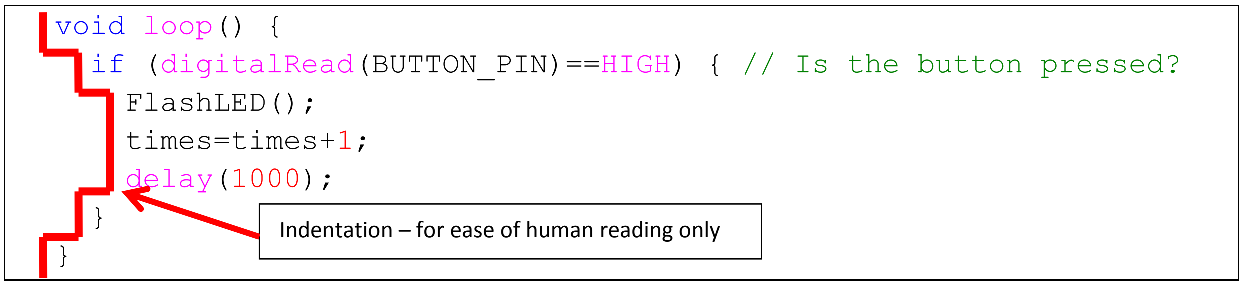
9.2.2. Pre-Processor Directives

Pre-Processor directives begin with a #, such as the #define in the example code, must be placed on their own line, and continue till the end of the line (it is possible to extend them to multiple lines, however they typically span one line). In the example, "#define BUTTON_PIN 2" tells the compiler that "BUTTON_PIN" and "2" mean the same thing in this program (are interchangeable).

```
#define BUTTON_PIN 2 //"BUTTON_PIN" means the same thing as "2"
```

9.2.3. Indentation

Indentation, or the practice of inserting spaces at the start of a line, is done only to make the code easier for humans to read - indentation is completely ignored by the computer.



```
void loop() {
  if (digitalRead(BUTTON_PIN)==HIGH) { // Is the button pressed?
    FlashLED();
    times=times+1;
    delay(1000);
  }
}
```

9.2.4. Semicolons

All commands and declarations must be followed by a semicolon (";"). For example:

```
int times; // Variable declaration
times=times+1; // Command
delay(1000); // Command
```

In some other programming languages commands and declarations are separated by placing them on separate lines, but in C they are separated by semicolons. The following code is the same as the former:

```
int times; times=times+1; delay(1000); // All on one line
```

C generally does not look at lines, except in some cases such as with "Rest-of-the-line" comments and pre-processor directives. Instead, C looks at semicolons to separate commands.

9.2.5. Variable Declarations

Before a variable can be used in a program, it must first be "declared" by specifying what kind of variable it is as well as its name. Variable declarations are done by first specifying the type of variable – for example "int" for integer – and then the name of the variable (followed by a semicolon).

```
int times;
```

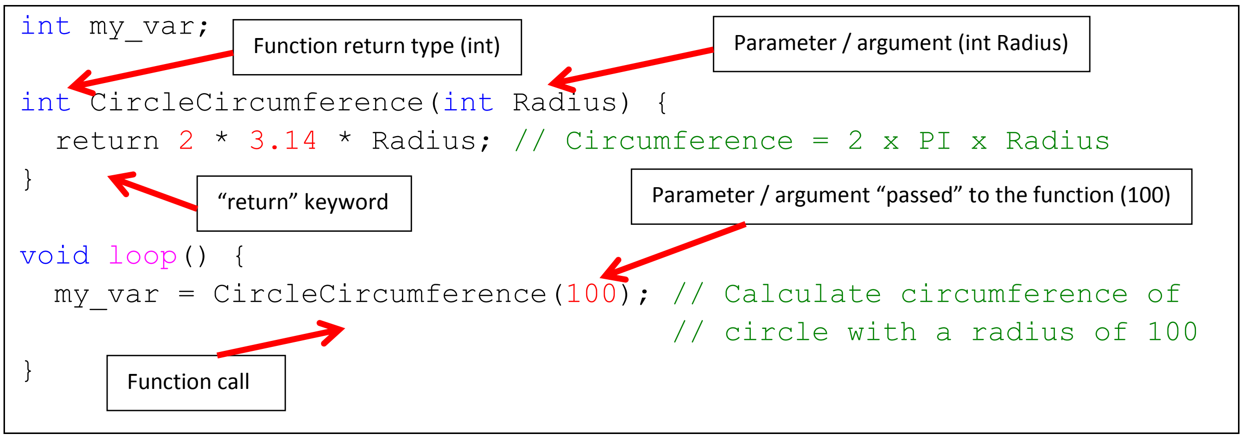
There are numerous types of variables available to choose from – more information regarding these can be found in the compiler documentation (for Arduino, this is at https://www.arduino.cc/en/Reference/VariableDeclaration).

9.2.6. Sub-Routines

Sub-routines are inserted into a program by first specifying the type of variable value they return – for example "void" if they do not return anything – and then the name of the function followed by opening and closing brackets "()". The instructions which the sub-routine should execute are then inserted between curly braces ("{" and "}").

```
void MySubRoutine() {
  // Sub-routine instructions go here
}
```

"Parameters" (also called "arguments") can optionally be inserted between the opening and closing brackets. The following example of a function which both returns a value and accepts a parameter / argument is discussed in the following sections.



```
int my_var;

int CircleCircumference(int Radius) {
  return 2 * 3.14 * Radius; // Circumference = 2 x PI x Radius
}

void loop() {
  my_var = CircleCircumference(100); // Calculate circumference of
                                     // circle with a radius of 100
}
```

Function return type (int)
Parameter / argument (int Radius)
"return" keyword
Parameter / argument "passed" to the function (100)
Function call

9.2.6.1. Sub-Routines which Return a Value ("Functions")

Up till now, we have built sub-routines which behave simply as "procedures" – they only execute a list of commands. Sub-routines in C can also work as *functions* however, in that they can *return* a value. For example, we could have a function which calculates the circumference of a circle from its radius. We could then use (call) that function like this:

```
my_var = CircleCircumference(100);
```

This code would copy the *return* value from the CircleCircumference function into the variable called "my_var".

If the function does not return anything (i.e. if it is simply a procedure), then the type "void" can be used. For example, from the example program ("C Code 1"):

```
void FlashLED()
```

The example program ("C Code 1") does demonstrate *using* (calling) a function which returns a value however:

```
if (digitalRead(BUTTON_PIN)==HIGH)
```

"digitalRead" is a function which returns a value of either "HIGH" or "LOW".

It is possible to write any program without building your own functions which return a value, however doing so can often make things easier. An exhaustive explanation regarding writing functions which return a value is beyond the scope of this book, however for the curious there are many resources available online – such as at https://www.arduino.cc/en/Reference/FunctionDeclaration.

9.2.6.2. Sub-Routine Parameters

Sub-routines can optionally require *parameters* to be "passed" to them when they are called. For example, to use the example of a function which calculates the area of a circle again:

```
int CircleCircumference(int Radius)
```

This function requires an "int" (number) parameter to be passed to it when it is called. Inside that function, the value passed can be referred to by the name "Radius".

Whilst the example program ("C Code 1") does not demonstrate how to declare (insert) a function which requires parameters, it does demonstrate *using* (calling) functions which require arguments – for example:

```
delay(1000);
```

"delay" is the name of the sub-routine (function), and "1000" is the value passed as a parameter to that function. Another example is:

```
digitalWrite(13,HIGH);
```

The "digitalWrite" function requires two parameters – a pin number and a state ("HIGH" or "LOW"). The parameters are separated by a comma (",").

It is again possible to write any program without building your own functions which require parameters, however doing so can often make things easier. An exhaustive explanation regarding writing functions which require parameters is beyond the scope of this book, however for the curious there are many resources available online – such as at https://www.arduino.cc/en/Reference/FunctionDeclaration.

9.2.7. Statements

Statements include "if" decisions, "while" loops and "for" loops.

9.2.7.1. Statement Conditions

The condition(s) for a statement must be placed within brackets. For example (statement brackets highlighted):

```
if (digitalRead(2)==HIGH)
```

The various statements are further discussed in a following section.

Note that, when asking a *question* regarding equality in C (such as in a statement condition), two equals (==) signs should be used. One equals sign (=) is used in instructions ("telling") and two equals signs (==) are used in conditions ("asking"). It is **very important** to get this correct, as otherwise the statement will *always* evaluate to true, and you will be scratching your head wondering why it isn't working as expected! This is a nuance of C which has uses for advanced users, but for most users should simply be remembered.

9.2.7.2. *Statement Groups*

Statement groups are demarcated using curly braces ("{" and "}"). By default, a statement will only affect the instruction immediately following it. For example, in the following code only "FlashLED();" forms part of the while loop.

```
while (digitalRead(2)==HIGH)
  FlashLED();  // This instruction is included in the loop
  delay(1000); // This instruction is not included in the loop
```

In order to incorporate the delay into the while loop as well, the following code is needed (curly braces highlighted):

```
while (digitalRead(2)==HIGH) {
  FlashLED();  // Both instructions
  delay(1000); // are now included in the loop
}
```

9.3. C Statements

The two fundamental types of statements in C are the "if" statement and the "while" loop. All other types of statements can be constructed using these.

In the statement syntaxes listed below, square brackets denote optional code (meaning that the code between the square brackets, including the square brackets themselves, is optional and can be left out if not needed). Curly braces for statement grouping have been inserted, even when not strictly needed, for clarity.

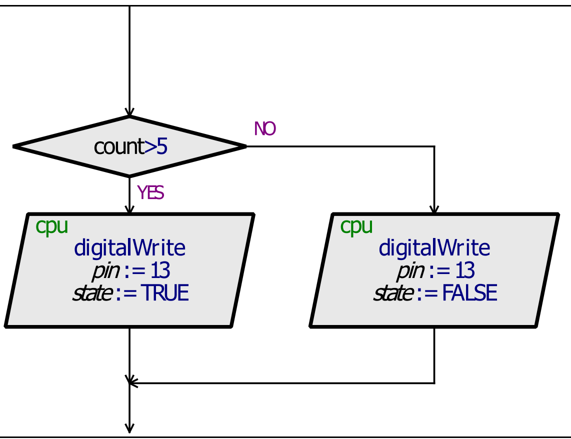
9.3.1. "If" Statement

Syntax

```
if (expression) instruction1 [else instruction2]
```

Example

```
if (count > 5) {
  digitalWrite(13, HIGH);
} else {
  digitalWrite(13, LOW);
}
```



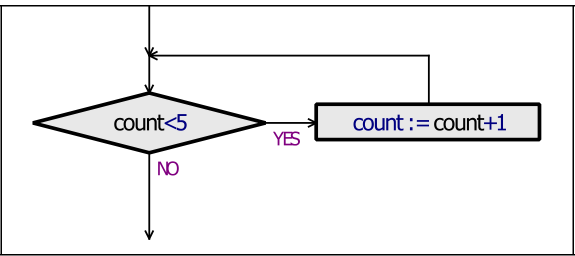
9.3.2. “While” Statement

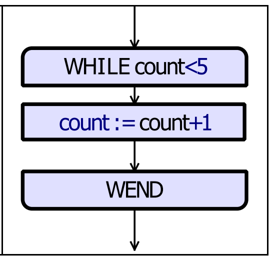
Syntax

```
while (expression) instruction
```

Example

```
while (count < 5) {
   count=count+1;
}
```





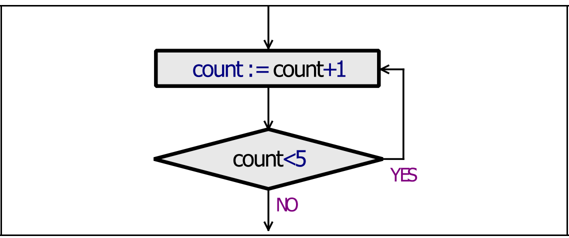
9.3.3. “Do” Statement

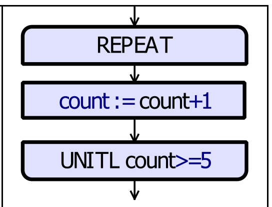
Syntax

```
do instruction while (expression);
```

Example

```
do {
   count=count+1;
} while (count < 5);
```





Note: The “Repeat” flowchart loop in the example has inverted logic (“count >= 5” rather than “count < 5”, as it loops “until” rather than “while”). There is no direct equivalent in C.

9.3.4. “For” Statement

Syntax

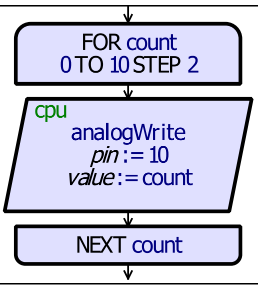
```
for ([init-expr]; [cond-expr]; [inc-expr]) instruction
```

- “init-expr” is the “initialisation expression”, which is an instruction executed before the loop begins.
- “cond-expre” is the “condition expression”, which is checked (evaluated) at the start of each loop pass to decide whether the looping should continue.
- “incr-expr” is the “increment expression”, which is an instruction executed after each loop pass. Note that this instruction does not necessarily need to “increment” something, and could potentially multiply etc.

You may wish to review section 8.7.3 for a reminder on how “For” loops in flowcharts can also be built using decision and action blocks.

Example

```
for (count=0; count<=10; count=count+2) {
  analogWrite(10, count);
}
```



9.3.5. “Switch” Statement

The “Switch” statement is like a decision block which has more possible outcomes than just “yes” and “no”. There is currently no equivalent block for this in Visual Designer (note however that the same effect can also obviously be achieved using multiple decision blocks / “if” statements).

Syntax

```
switch (expression) {
  case constant-expression_1:
    statement1;
    [break;]
  case constant-expression_2:
    statement2;
    [break;]
  [default:
    statement;]
}
```

Example

```
switch (count) {
  case 1:
    analogWrite(10, 12);
    break;
  case 2:
    analogWrite(10, 34);
    break;
  case 3:
    analogWrite(10, 56);
    break;
  default:
    analogWrite(10, 0);
}
```

Note that if the “break” instruction is omitted, then the code will continue executing the “cases” below the specific case which was matched, until a break instruction is encountered, or the end of the switch statement is reached.

An exhaustive discussion regarding the “switch” statement is beyond the scope of this book, however for the curious there are many resources available online – such as at https://www.arduino.cc/reference/en/language/structure/control-structure/switchcase/.

9.4. Next Steps for Further Learning

As C is an industry standard programming language (commonly used), there are many learning resources available for it.

A suggested initial step would be an internet search like "Getting Started with C". To be Arduino specific, try "Arduino Getting Started" – results from the official Arduino website itself (https://www.arduino.cc/en/Guide/HomePage), as well as from Adafruit (https://learn.adafruit.com/lesson-0-getting-started/overview), should be near the top of the list.

Proteus also includes a number of Arduino sample projects using C code, as well as numerous other sample projects using C code on other devices. Opening the Sample Projects Browser (File, Open Sample Project), typing "Arduino" as the keyword, and then selecting the VSM for AVR category, is a suggested start. The example projects can be studied, modified and simulated.

The official Arduino IDE (see https://www.arduino.cc/en/Main/Software) includes numerous sample projects demonstrating various concepts.

Note that Proteus can simulate microcontroller code even if that code is not developed within the Proteus environment! For example, you could develop code in the Arduino IDE, and then still simulate it in Proteus. For more information on this, please see the "Direct Simulation" topic under "Working with Microprocessors" in the Proteus Simulation help file. The Proteus Simulation help file is available from the Help menu in Proteus when the Simulation tab is selected / active.

C concepts which have not been covered (or not fully covered) in this chapter include:

- Arrays
- Pointers
- Jump Statements ("break" and "continue")
- Strings
- Structures
- Enumerations
- Libraries

In addition, no C++ specific concepts have been covered. Note that the Arduino compiler is a C/C++ compiler, meaning that it can compile programs written in just C code as well as programs written in C++ code.

In general, a good way to learn is to study and modify working examples. That way, you have something which already works, and you can start modifying it for your case. At each modification step you should test your modifications – if they don't work as expected, or at all, then you can focus on fixing a relatively small step, or doing things a different way; this is in contrast to trying to write a large program from scratch, finding it doesn't work as expected, and then trying to pinpoint where exactly the problem might be. Note that you also don't necessarily need to understand exactly how each part of an example project works in order to modify a certain smaller part of it to suit your needs.

Remember that, if you have successfully completed and understood the previous chapters, you already know how to think like a programmer. Learning C is just a matter of learning how to communicate the instructions to the computer in a different way.

9.5. C Operator Reference

Arithmetic Operators

+	add
-	subtract
*	multiply
/	divide
%	modulus (remainder after division)

Relational Operators

==	equal
!=	not equal
>	greater than
<	smaller than
>=	greater or equal
<=	smaller or equal

Assignment Operators

```
a  = b; // Assignment
a += b; // a = a + b;
a *= b; // etc...
```

Logical Operators

&&	logical AND
||	logical OR
!	logical NOT

Bitwise Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise Complement
<<	bitwise Shift Left
>>	bitwise Shift Right

Unary Operators

```
i++; // i = i + 1;
i--; // i = i – 1;
a = ++b; // b = b + 1;
         // a = b;
a = b++; // a = b;
         // b = b + 1;
```

9.6. C Variable Type Reference (Arduino)

Type	Size in Bytes	Range
byte	1	0 .. 255
[unsigned] char	1	0 .. 255
signed char	1	- 128 .. 127
[signed] short [int]	1	- 128 .. 127
[signed] int	2	-32768 .. 32767
unsigned [int]	2	0 .. 65535
[signed] long [int]	4	-2147483648 .. 2147483647
unsigned long [int]	4	0 .. 4294967295
float	4	$-3.4 * 10^{38}$ .. $+3.4 * 10^{38}$

9.7. C Pointers Reference

Pointer Declaration

```
char* p; // Declares "p" as a character pointer
         // (pointer to a character)
```

"Value At" a Pointed Address

```
v = *p; // v is assigned the value of the variable
        // (at the address) pointed to by p
```

"Address Of" a Variable

```
p = &v; // p is assigned the address of the variable v
        // (p points to v)
```

9.8. Converting a Flowchart to C/C++ Code

All Visual Designer flowcharts can be converted to C/C++ code with a few clicks. You may find it useful, whilst learning C, to first design a program as a flowchart and then convert it to C, and/or to convert projects from earlier chapters in this book into C.

To convert a flowchart into C/C++ code, simply right-click on the project folder in Visual Designer, and select “Convert to Source Code Project” in Visual Designer.

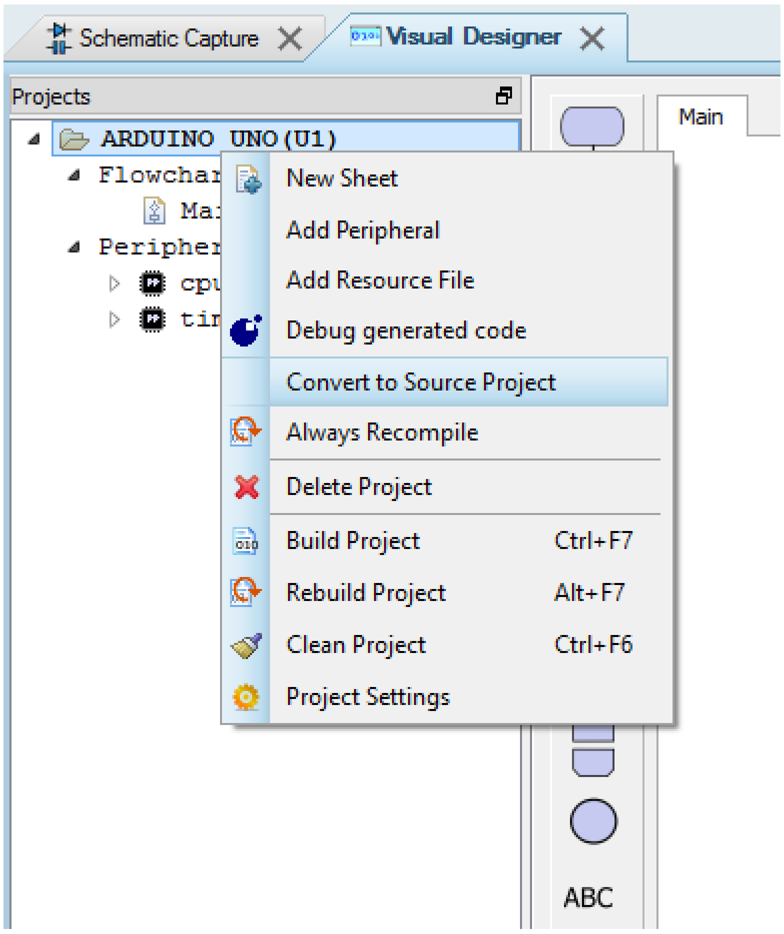
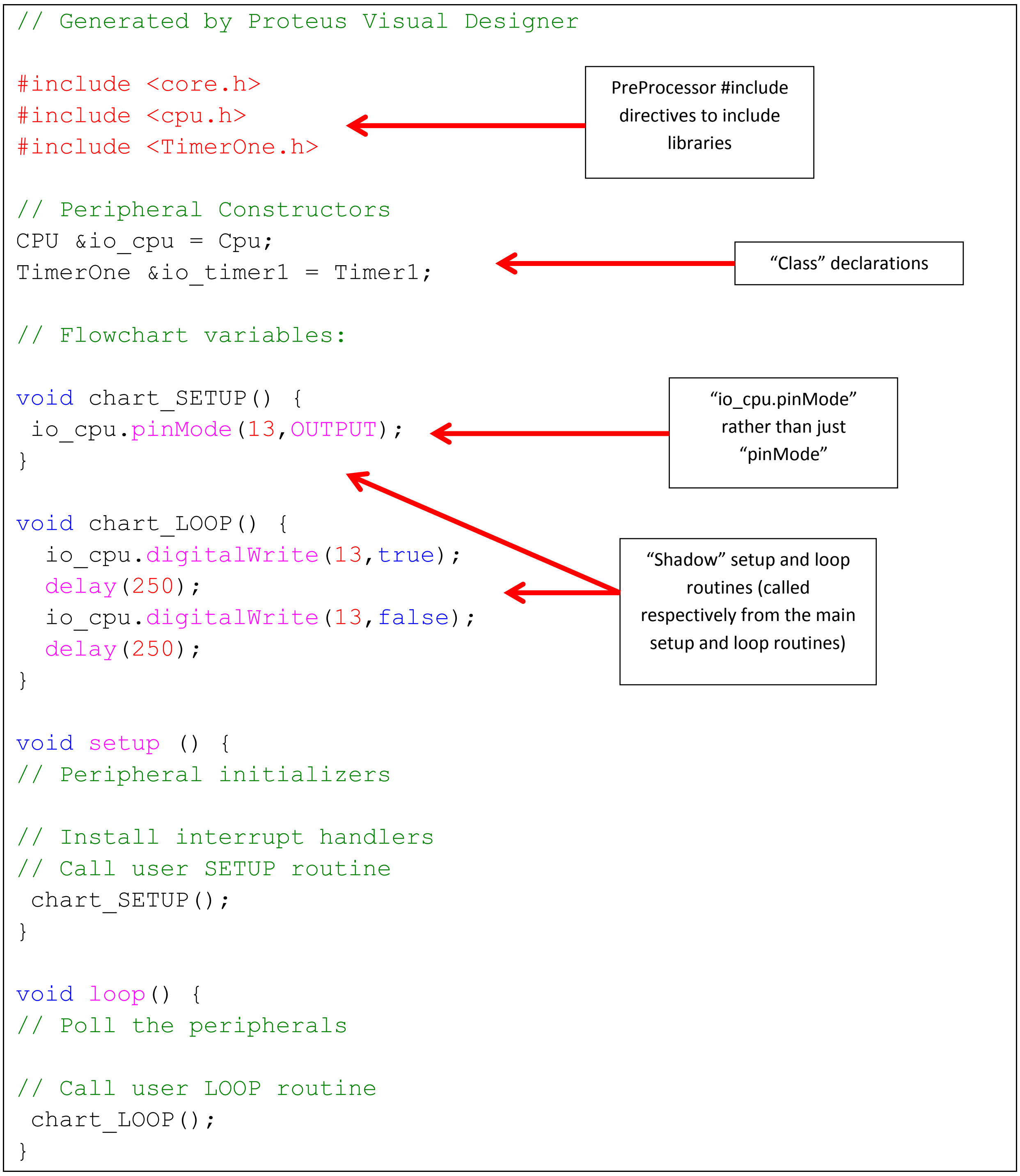


Figure 9-4: Convert to Source Project

As an example, let’s convert the project from section 9.1 into C/C++ code.

1. Either recreate or reopen the flowchart project from section 9.1 (“C_BlinkLED_Flowchart”). Remember to save any work which you may have open and don’t want to lose. The filename “C_BlinkLED_Converted” can be used for the project, either during the “New Project Wizard” if recreating the project, or by using the “Save Project As” command under the file menu if reopening the project.
2. Convert the project to C/C++ code, using the procedure detailed in Figure 9-4 (right click on the project folder and then select “Convert to Source Project” from the popup menu).
3. If not already open, double-click on the “main.ino” file on the left in order to open it.

The resulting code should look like the following:



```
// Generated by Proteus Visual Designer

#include <core.h>
#include <cpu.h>
#include <TimerOne.h>

// Peripheral Constructors
CPU &io_cpu = Cpu;
TimerOne &io_timer1 = Timer1;

// Flowchart variables:

void chart_SETUP() {
 io_cpu.pinMode(13,OUTPUT);
}

void chart_LOOP() {
  io_cpu.digitalWrite(13,true);
  delay(250);
  io_cpu.digitalWrite(13,false);
  delay(250);
}

void setup () {
// Peripheral initializers

// Install interrupt handlers
// Call user SETUP routine
 chart_SETUP();
}

void loop() {
// Poll the peripherals

// Call user LOOP routine
 chart_LOOP();
}
```

PreProcessor #include directives to include libraries
“Class” declarations
“io_cpu.pinMode” rather than just “pinMode”
“Shadow” setup and loop routines (called respectively from the main setup and loop routines)

C Code 2: Flowchart converted to C code

You will notice that this code looks a little more complicated than the code from section 9.1 (copied again on the next page for convenience), however it accomplishes exactly the same thing. In fact, this is the code which the flowchart is converted into each time it is compiled (such as before simulation).

```
void setup () {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

C Code 3: Simplified version of "C Code 2" program

The code which has been automatically generated by converting the flowchart is a bit more complicated than strictly necessary, due to the way in which Visual Designer works inside. This additional "complication" however does grant Visual Designer the capability to design programs using flowcharts, which is well worth it.

Note that, as mentioned in the previous section, you don't necessarily need to understand exactly how each part of the code works in order to modify a smaller part of it to suit your needs, or even study a smaller part of it to see how Proteus converted a certain section of flowchart into code.

Some of the nuances are nonetheless discussed below.

9.8.1. PreProcessor Include Directives

The pre-processor include directives are used to reference libraries which are used in the program. The libraries are contained in separate files, which may themselves, in turn, reference even more files.

9.8.2. Class Declarations

"Classes" are a C++ feature. They are a way of grouping together sections of code, and are used in something called "object-oriented programming". For example, there could be a class called "screen", with *properties* "width" and "height" ("screen.width" and "screen.height"), and a *method* called "clear" ("screen.clear"). Properties are variables which are used in a class, and methods are sub-routines which are used in a class. Class properties and methods are referenced by using the class name (e.g. io_cpu) followed by a dot "." and then the variable or method name (e.g. digitalWrite).

```
  io_cpu.digitalWrite(13,true);
```

Each of the peripherals in Visual Designer are implemented as a class.

10. Summary

Through this course, we have looked at why electronics and software are important to us, and also at how easy it can be to get started with these disciplines.

We have learned how to instruct computers what to do by developing software programs for them, and also learned some fundamental principles of electronics. Starting with programming simple instructions, then learning how to make decisions, then adding variables for remembering things, and then even learning about sub-routines and conditional loops; insight was gained into how computers "think".

Using a bucket, a pipe and a tap we learned about Ohm's law; the relationship between Voltage, Resistance and Current. We also learned how to draw and simulate our own electronic circuits.

If you have completed this book, then you now have the basic tools needed to start building your own smart machines; and also know where to look to continue building on your skill-set. There are many advanced topics which can be studied. The electronics industry is also a fast moving one, and learning is thus a continual process as new technologies are invented and older ones are improved. The basic principles however, as covered in this book, remain the same.

It is up to you to find applications for the tools you should now have! The next time you encounter a problem, or a repetitive task, then perhaps consider whether you could build a machine to help with it – to make your and other people's lives easier.

Happy developing!

11. Bonus Chapter: Robotics

In case you did not already know this; electricity can be used to make things move!

Traditionally, motion from electricity is often achieved using electric motors (which exploit the relationship between electricity and magnetism, or “Electromagnetism”[21]). Electric motors are extremely common, and you will no doubt have encountered them many times in everyday life (air-conditioners, escalators, fridges, gate-motors, quadcopters, drills, hair-dryers, etc).

Electricity can also be used to make things move in other ways; such as with piezoelectricity, where an electric charge results in a deformation or “bending” of a material[21].

Besides for directly causing movement, electric signals can also be used to control other motion systems; petrol- and diesel-engines are typically controlled by an ECU (Electronic Control Unit), and even the muscles in your own body are controlled by electric signals (transmitted via your nerves)! When programming is applied in order to intelligently control a moving machine, then this is typically called “robotics”; robotics is thus not really much different from what you have already learned, just with moving parts involved!

Proteus includes some robotic simulation features. One of the robots which can be simulated is the Pololu Zumo:

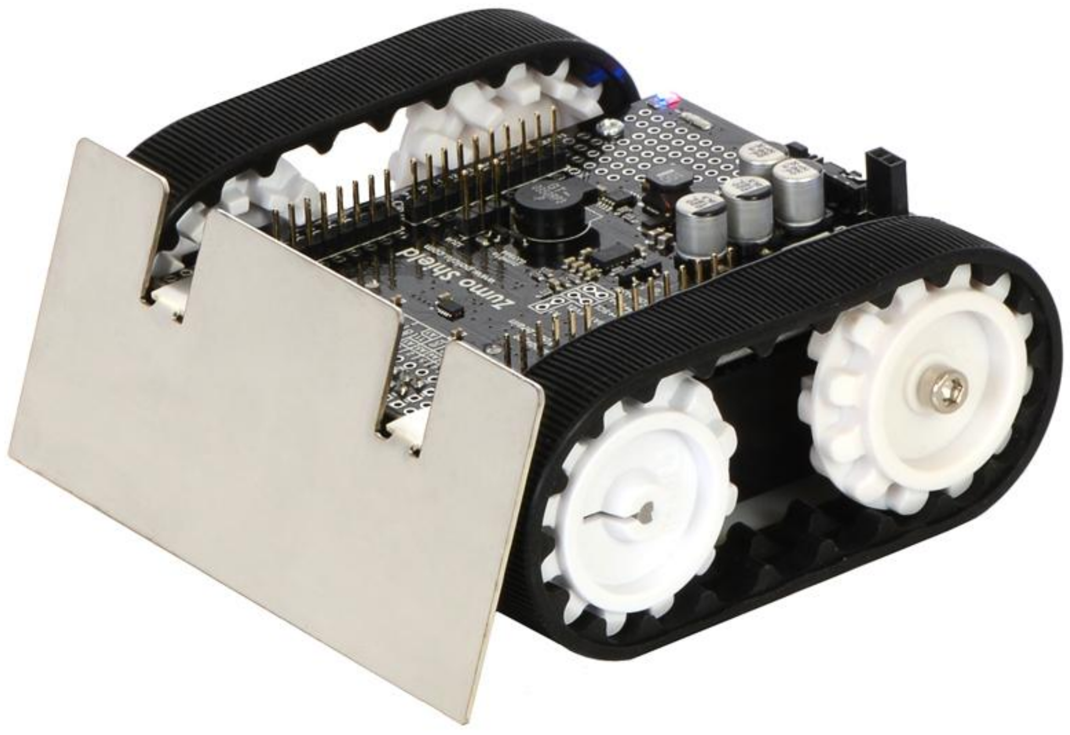


Figure 11-1: Pololu Zumo Robot

[21] In this chapter we simply accept that electricity can be used to create motion (such as with an electric motor). Details regarding exactly *how* electricity is used to create motion are beyond the scope of this book, however there are many other resources available on these topics (for example, conduct web searches for “Electromagnetism” / “Electromagnet” / “DC Motor” / “Piezoelectricity” etc).

The Pololu Zumo features:

- Two electric motors (one for each driving track).
- An array of 6 infra-red reflectance sensors, which can be used for following lines or detecting edges (more on this below).
- A piezo buzzer, for playing sounds (as briefly mentioned previously, piezoelectricity is another way of creating motion with electricity – in this case piezoelectricity is used to generate vibration which results in sound waves).
- Navigation sensors:
  - 3-axis accelerometer (can be used to detect which way is "up", or how fast the robot is accelerating/decelerating).
  - 3-axis magnetometer ("compass").
  - 3-axis gyroscope (for detecting rotation, such as if the robot turns 90° left or right).

A number of convenient methods are included in Visual Designer for working with the robot:

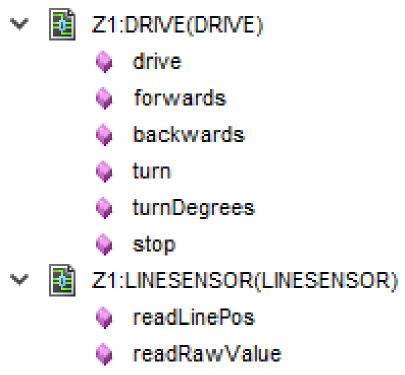


Figure 11-2: Some of the Zumo Peripheral Methods

Specific information on each of these methods is included in the Visual Designer help file (available from the Help menu when the Visual Designer tab is selected):

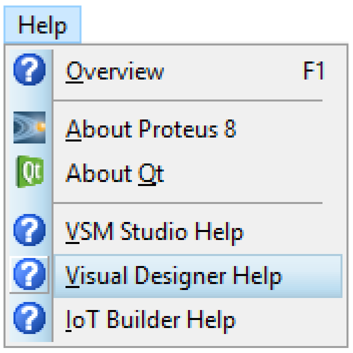


Figure 11-3: Visual Designer Help Menu

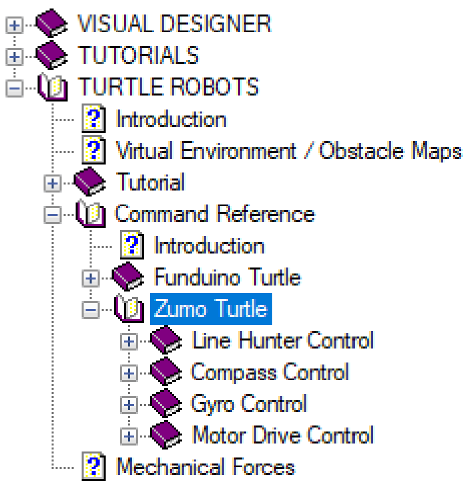


Figure 11-4: Visual Designer Help Topics

Proteus can simulate the robots driving around an obstacle course / track; besides for this being useful in the case where physical hardware is not available, it can also be very helpful whilst developing and debugging the code / algorithms which should control the robots:

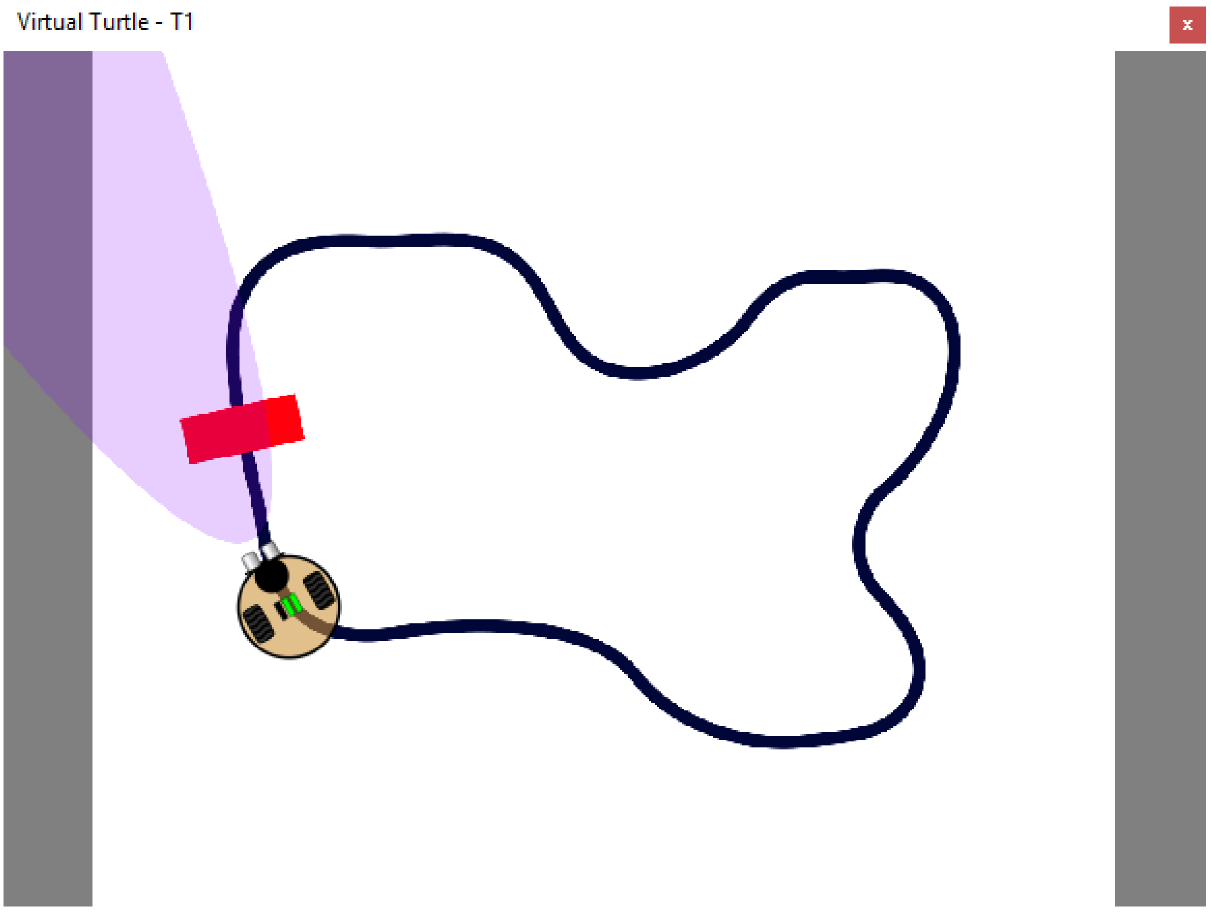


Figure 11-5: Proteus Robot Simulation

You can draw any track you like, using a simple graphics editor such as MS Paint (for details on exactly how to do this please see the "Virtual Environments / Obstacle Maps" topic in the previously-mentioned help file – for this chapter however, we will simply use a sample map rather than drawing our own).

In this chapter, we will develop a simple program to help the Zumo robot follow a line. You will be able to try the program out in the simulation, and if you have a physical Zumo robot available then you can try it out with the "real" hardware as well (by, for example, constructing a track on the floor using black insulation-tape).

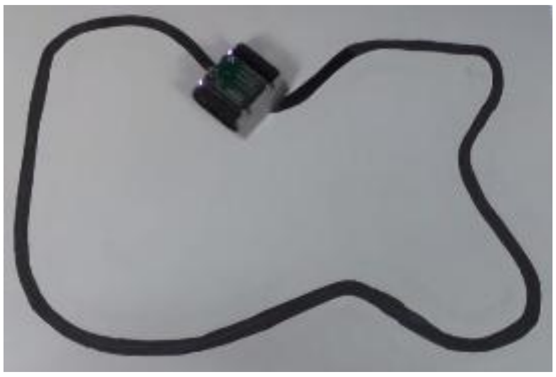


Figure 11-6: "Real Life" Zumo and Track

11.1. Line-Following Theory

Underneath the front of the Zumo robot are 6 infra-red reflectance sensors.

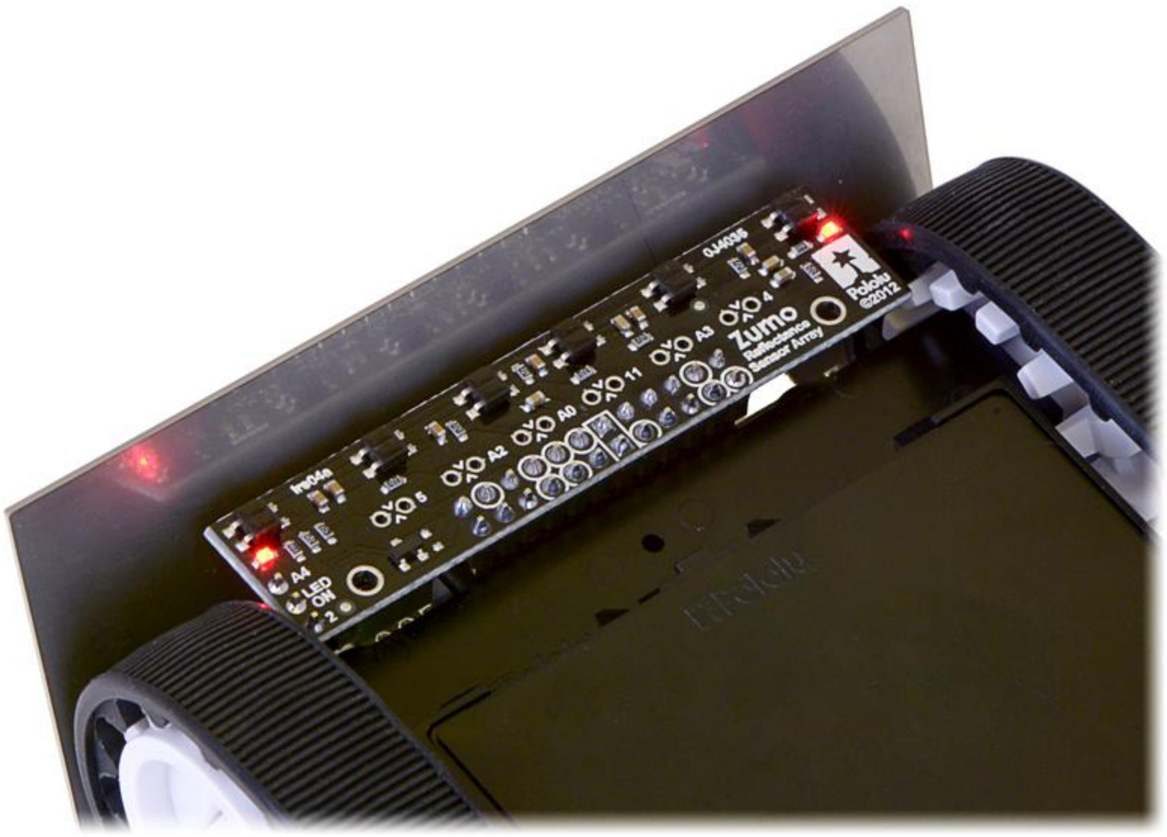


Figure 11-7: Zumo Reflectance Sensors

The reflectance sensors work by emitting ("shining") a light, and then detecting how much of the light is reflected back (note that the red lights visible in the picture are not the light which is being emitted/detected by the sensors – the sensors use infra-red light, which is invisible to humans). Darker objects reflect less light and lighter objects reflect more light. The sensors can thus be used to, for example, detect a black line against a white background: sensors currently over the line will see less reflectance (indicating that a "dark" object is currently underneath the sensor), whilst sensors currently over the background will see more reflectance (indicating that a "light" object is currently underneath the sensor). Using this information, we can tell whether the line is currently under the middle of the robot, under the left of the robot, or under the right of the robot (as well as how far to the left or right the line is). In order to follow the line, the robot should try to keep the line under the middle sensor(s):

Line Position	**Action Required (to get "Back on Track")**
Far Left	Turn left a lot
Left	Turn left a little bit
Middle	*Drive straight ahead*
Right	Turn right a little bit
Far Right	Turn right a lot

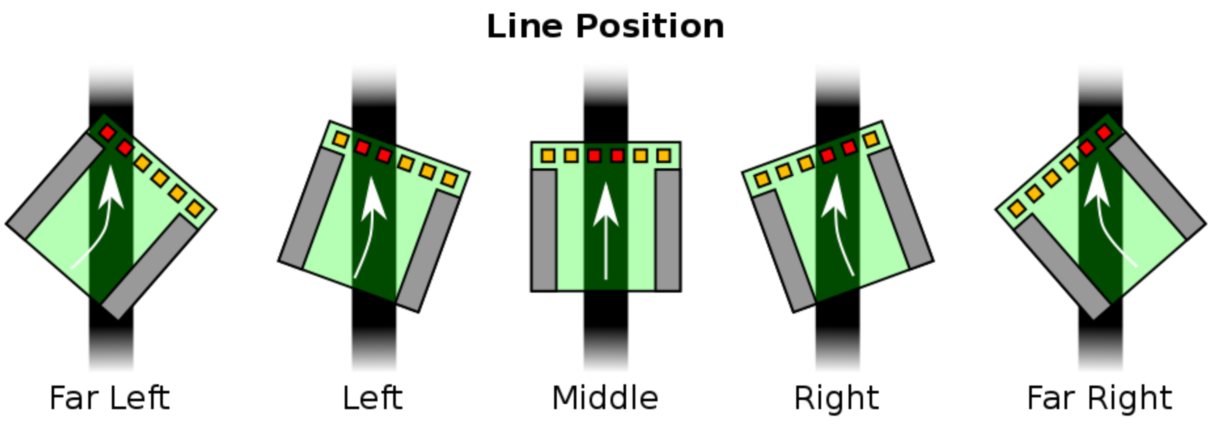


Figure 11-8: Corrective action required according to line position

If the line is currently under the left sensor(s), and the robot turns left, then as the robot continues to drive the line will "move" towards the middle sensor(s) – and vice versa.
If the robot can "keep" the line under the middle sensors, as it drives forwards, then it will succeed in "following" the line! (It may help to look at the previous picture of a line-track which the robot should follow, and think about how the position of the line under the robot will change as the robot drives forwards around the track).

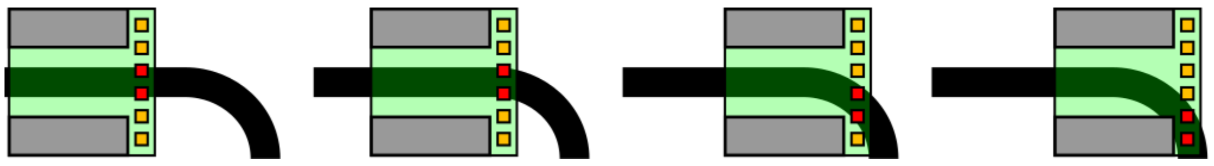


Figure 11-9: Changing line position as robot drives forwards

This sort of control system is called "proportional"; the amount which the robot should turn by is proportional to how far the line is from the middle sensor(s) – the further the line is from the middle, the more sharply the robot should turn in order to get “back on track”. Proportional control by itself can be subject to overshoot / “wobble” / “fish-tailing”, so more advanced control methods are also available (such as “PID”, which stands for Proportional-Integral-Derivative).

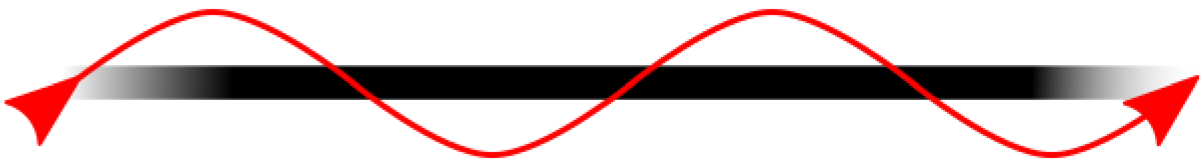


Figure 11-10: "Fishtailing"

For our project however, “proportional” control alone will work just fine.

11.2. Constructing the Program: The Plan

For our program, we will use two of the Zumo’s methods: “drive” and “readLinePos”.

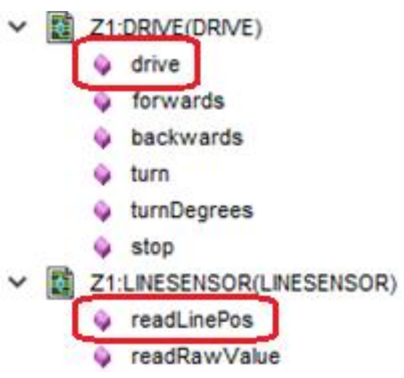


Figure 11-11: Zumo peripheral methods to be used

11.2.1. “drive” Method

Parameter	Possible Values	Description
Wheel	- Both - Left - Right	Specify which track(s) (“wheels”) to set the speed/direction for.
Dir	- Forwards - Backwards	Specify whether the track (“wheel”) should drive forwards or backwards.
Speed	0-255	Specify how fast the track (“wheel”) should drive; where 0 is “stopped” and 255 is “full speed”.

- To drive straight ahead, both tracks (“wheels”) should obviously be set to the same speed.
- To turn left, the right track should be set to drive faster than the left track.
- To turn right, the left track should be set to drive faster than the right track.
- When turning, the difference in speed between the two tracks will determine how “sharply” the robot will turn.

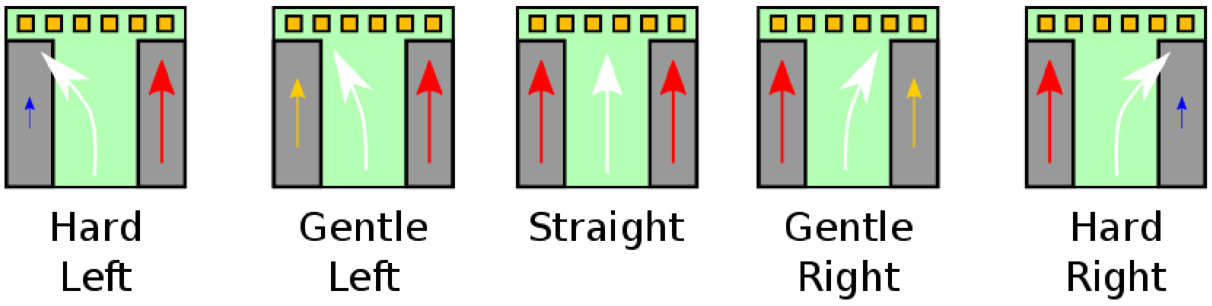


Figure 11-12: Left and right track speed differences, and resulting turn

To turn, it is necessary to use the “drive” method twice – once for each wheel.
To drive straight ahead, the “drive” method could be used once – with the “Wheel” parameter set to “Both”; but we could also drive straight ahead by using the “drive” method twice (once for each wheel), so long as the speed and direction were the same for each wheel.

11.2.2. “readLinePos” Method

The “readLinePos” method returns a value between 0 and 5000, which represents the position of the line underneath the sensors – “0” meaning that the line is under the far left sensor and “5000” meaning that the line is under the far right sensor. If the line is directly underneath the middle sensors, then the reading will be 2500.

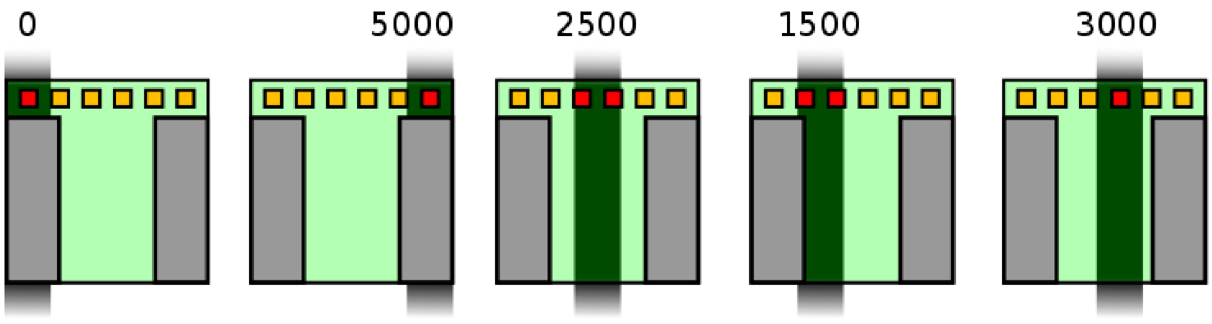


Figure 11-13: Example "readLinePos" result for different line positions

11.2.3. The Plan

The basic structure of our program will be as follows:

1. Read the position of the line.
2. Calculate how far “off track” the robot currently is (and thus how much we need to turn, and in what direction, in order to get “back on track”) – we will call this the “deviation”.
3. Adjust the speed of the motors, so that the robot turns in the correct direction in order to get “back on track”.

One way in which we can do this is to set the normal “straight forwards” motor speeds to 50% each, and then add or subtract speed from each motor depending on how much turning is required. This means that we could add or subtract up to 50% speed from each motor (giving a maximum speed of 100% and a minimum speed of 0% on each motor). We will thus need to interpret the “deviation” as a value from -50% to +50%, which should be added-to/subtracted-from each motor speed. The mathematics for this are below:

- Since “full speed” on each motor is represented by the number 255, 50% speed will be approximately 127.
- We thus need to convert the deviation to a number between -127 and +127 (meaning -50% to +50%).
    - Because the line being under the middle sensors is represented by the number 2500, we will need to subtract this number from the reading in order to get a deviation of 0 when the line is under the middle sensors.
    - After subtracting 2500 from the reading, we will have a number between -2500 and +2500 which represents the deviation. We now need to scale this into a range of -127 to +127, which can be done by dividing the number by approximately 20 (-2500/20=-125, and +2500/20=+125). This does not give exactly 127 (only 125), but it will work just fine.
    - Our formula for calculating the deviation is thus:
      `deviation=(reading-2500)/20`
      Some quick tests of our formula reveal that it works as intended:
        - A reading of 2500 (line under middle sensors) gives a deviation of 0.
        - A reading of 0 (line under far left sensor) gives a deviation of -125.
        - A reading of 5000 (line under far right sensor) gives a deviation of +125.
- We now need to adjust the motor speeds using the deviation. If the line is on the right-hand-side of the robot then the deviation will be positive. In order to turn right (in order to get the robot “back on track”, with the line under the middle sensors), we will need to increase the left “wheel” speed and decrease the right “wheel” speed, giving the formulas:
  ```
  leftWheelSpeed=127+deviation
  rightWheelSpeed=127-deviation
  ```

That’s it! However complicated you may have thought it would be to put together a program for a line-following robot, with a little clever maths we have done it in only a few steps. Of course this is not the best or most complex control algorithm (the sample project which comes with Proteus is more advanced), but as an introduction it is nice and simple and should work well (we will find out in a minute)!

11.3. Constructing the Program: Putting “The Plan” Into Action

By now it is assumed that you know how to start a new project, add peripherals to it, etc. If you need a refresher on these points then please consult the relevant prior chapters of this book.

To put together the line-follower project, we need to do the following:

1. Start a new Flowchart project (using Arduino Uno; the project could be called “ZumoLineFollower”).
2. Add the “Arduino Zumo Robot” peripheral to the project (under the “Motor Control” category).
3. Construct the flowchart (details to follow).
4. Specify the map (track) with the line for our robot to follow (details to follow).

11.3.1. Constructing the Flowchart

The flowchart for our program will look something like the following:

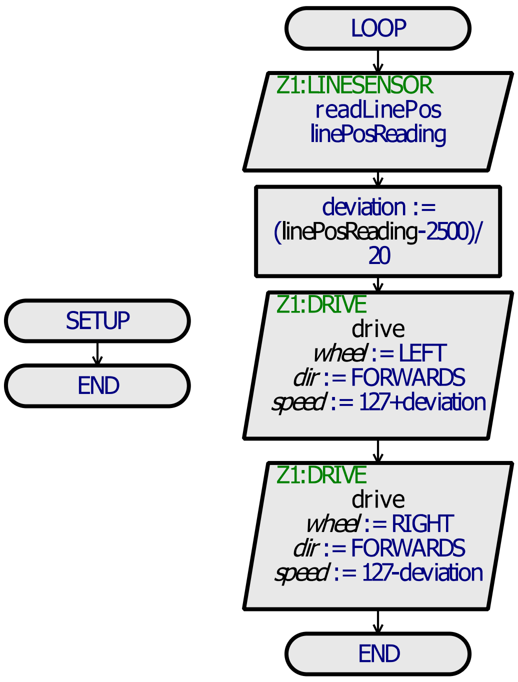
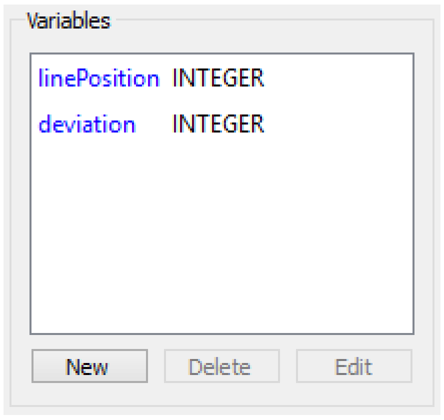


Figure 11-14: Simple line-follower flowchart program

This follows the steps we put together in “The Plan” stage. Note that two variables are needed:


Variables

linePosition	INTEGER
deviation	INTEGER

Figure 11-15: Line-follower flowchart program variables

If you have any trouble constructing the flowchart (or adding the variables) then please consult the relevant prior chapters which cover these topics.

11.3.2. Specifying the Map

To specify the map (track) for the robot:

On the Schematic sheet, go to the properties of the Zumo robot.

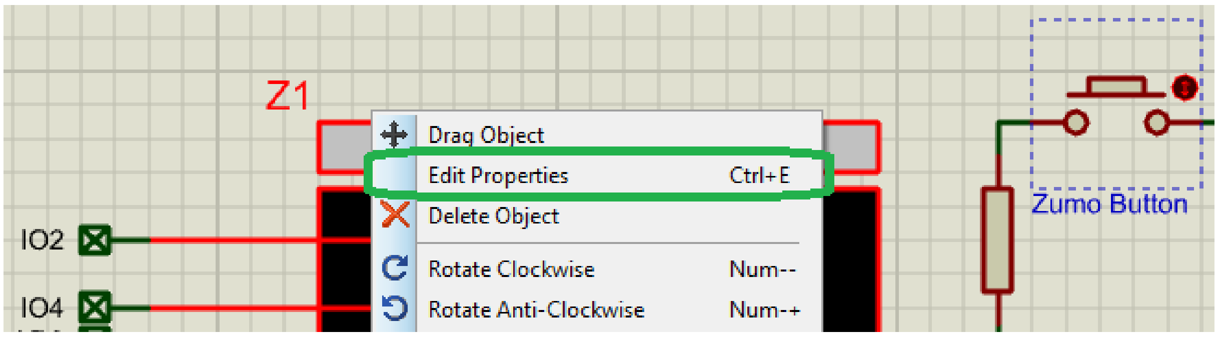
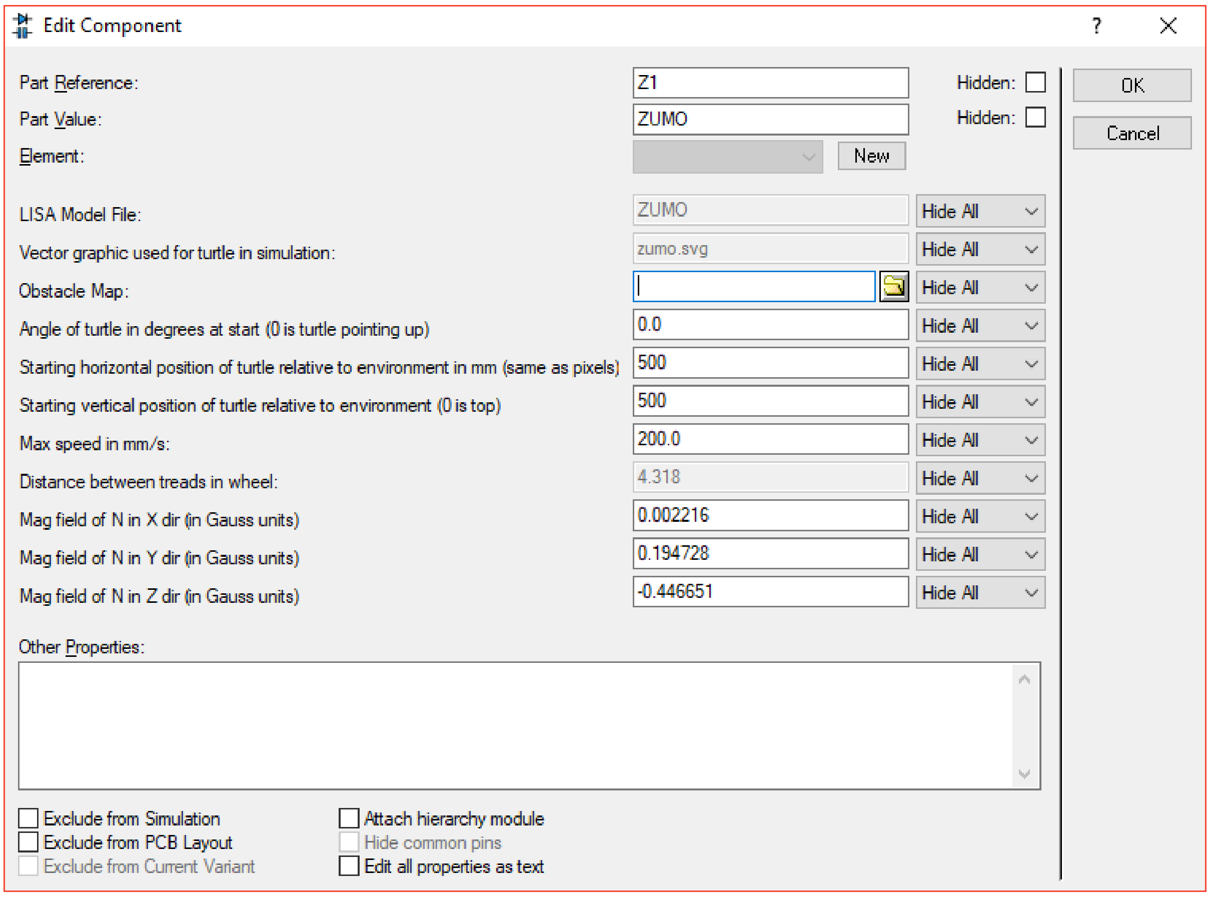


Figure 11-16: Accessing Zumo schematic part properties dialogue


Edit Component

Part Reference:	Z1	Hidden: ☐
Part Value:	ZUMO	Hidden: ☐
Element:		New
LISA Model File:	ZUMO	Hide All
Vector graphic used for turtle in simulation:	zumo.svg	Hide All
Obstacle Map:		Hide All
Angle of turtle in degrees at start (0 is turtle pointing up)	0.0	Hide All
Starting horizontal position of turtle relative to environment in mm (same as pixels)	500	Hide All
Starting vertical position of turtle relative to environment (0 is top)	500	Hide All
Max speed in mm/s:	200.0	Hide All
Distance between treads in wheel:	4.318	Hide All
Mag field of N in X dir (in Gauss units)	0.002216	Hide All
Mag field of N in Y dir (in Gauss units)	0.194728	Hide All
Mag field of N in Z dir (in Gauss units)	-0.446651	Hide All

Other Properties:

☐ Exclude from Simulation
☐ Exclude from PCB Layout
☐ Exclude from Current Variant
☐ Attach hierarchy module
☐ Hide common pins
☐ Edit all properties as text

Figure 11-17: Zumo schematic part properties

As mentioned previously, you can draw any track you like for the robot in a simple graphics program such as MS Paint (exact details regarding how to do this are in the help file). For this chapter however we will use a sample track which comes with Proteus.

The sample track can be found under the Proteus "Samples" directory (usually "C:\ProgramData\Labcenter Electronics\Proteus 8 Professional\SAMPLES\Visual Designer for Arduino\Virtual Turtle\") and is called "Line Follow.png".

For convenience, the sample track is also available for download from www.dizzy.co.za/zumo-st .

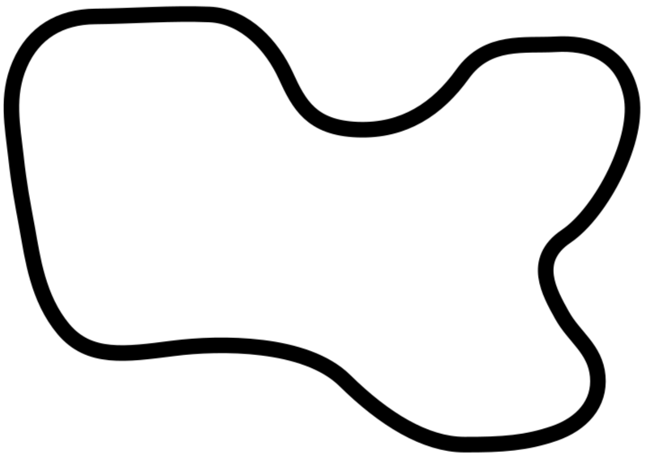
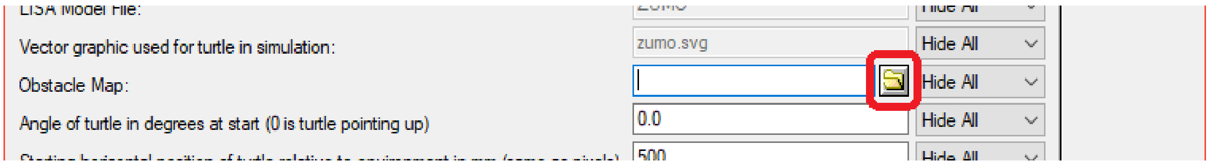


Figure 11-18: "Line Follow" sample track

To load the sample track into Proteus, click on the "Browse" button by the "Obstacle Map" field:



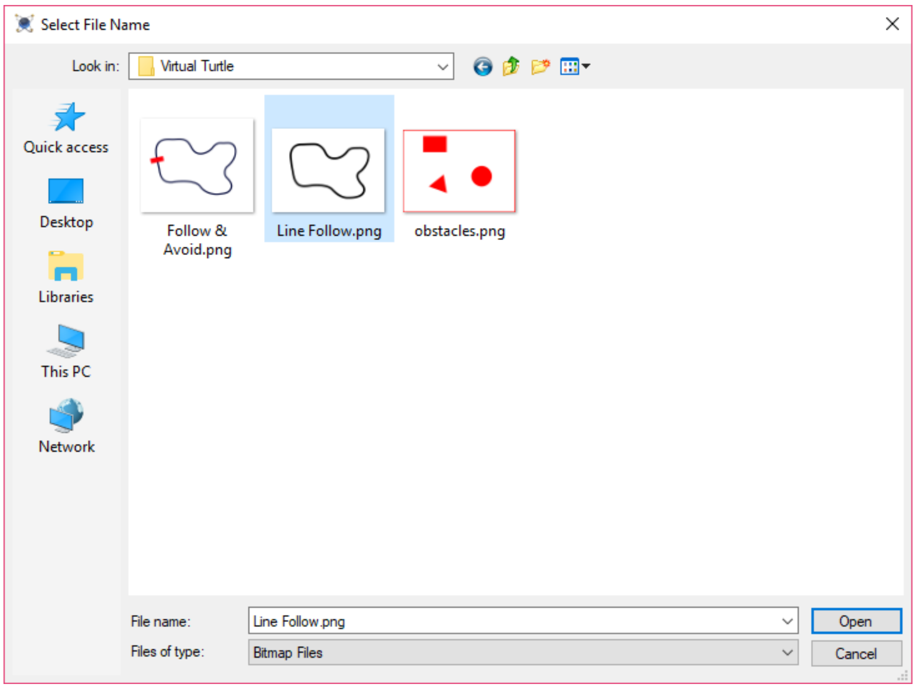
Vector graphic used for turtle in simulation: zumo.svg Hide All

Obstacle Map: Hide All

Angle of turtle in degrees at start (0 is turtle pointing up) 0.0 Hide All

Figure 11-19: Obstacle map field and "Browse" button

Navigate to the "Line Follow.png" file and click "Open".



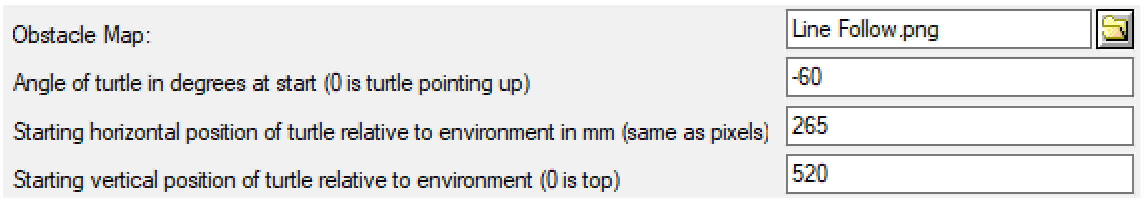
File name: Line Follow.png

Files of type: Bitmap Files

Figure 11-20: "Browse" dialogue

Next – because our program is still relatively simple and doesn't have code to handle the case where the robot is not over any line at all – we need to make sure that our robot is positioned over the track when it starts. To do this; we enter the starting angle, horizontal position and vertical position of the robot on the track:

Angle	-60
Horizontal Position	265
Vertical Position	520



Obstacle Map:	Line Follow.png
Angle of turtle in degrees at start (0 is turtle pointing up)	-60
Starting horizontal position of turtle relative to environment in mm (same as pixels)	265
Starting vertical position of turtle relative to environment (0 is top)	520

Figure 11-21: Zumo "start position" fields

(These values have been pre-calculated and are known to work, but there are many other values which could also work and you are free to experiment.)

This positions the robot near the bottom-left of the track:

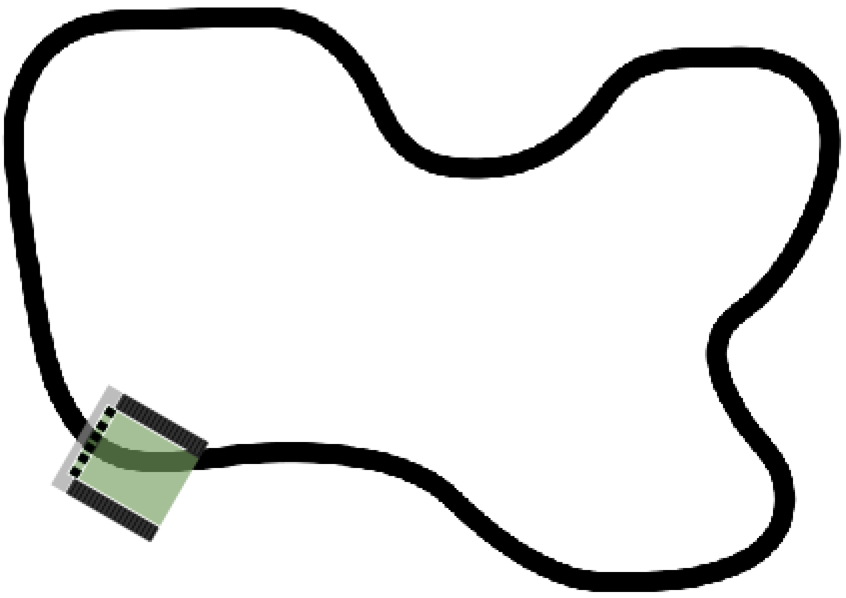


Figure 11-22: Zumo starting position on map

Once done, click "OK" to apply the changes to the Zumo Robot peripheral.

11.4. The Moment of Truth

We're all ready, so hit the "Play" button to start the simulation; and let's see if our robot can follow the line, or if it wonders off into oblivion!

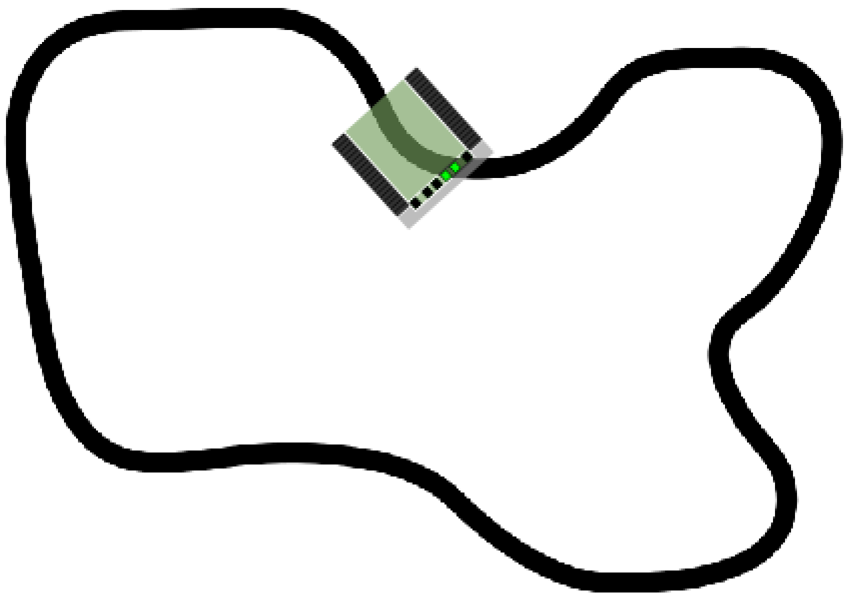


Figure 11-23: Line Follower simulation

If all has been done correctly then our robot should follow the track, without losing it.

11.5. Summary

In terms of programming, robotics is not really much more than what we have already learned – it just has motion involved. We have created a very simple program to help a robot follow a line as an introduction; you are free to experiment with better (faster/tighter) algorithms, different tracks, different applications, to study the robotics sample designs which are included with Proteus, to try out the other type(s) of robot(s) in Proteus, and so on. Happy experimenting and learning!

12. Author's Notes

12.1. Topics Not Yet Covered

The following topics have not yet been covered in this book (but may be included in a future revision). You may wish to investigate them yourself at your leisure, as subjects of further learning.

Topic	Resources for Self-Learning
Debugging	- "Simulation and Debug Skills" sub-topic in the Proteus Visual Designer help file[22] (under the "VISUAL DESIGNER" main topic).
Pulse-Width Modulation	- "Tutorial 4: Motor Control (Shield)" tutorial, under the "TUTORIALS" main topic in the Proteus Visual Designer help file[22]. - https://learn.adafruit.com/adafruit-arduino-lesson-3-rgb-leds
Interrupts	- "External and Timer Interrupts Demo" sample project. - https://learn.adafruit.com/multi-tasking-the-arduino-part-2 - https://www.sparkfun.com/tutorials/326 - https://learn.adafruit.com/multi-tasking-the-arduino-part-3
State Machines	- https://learn.adafruit.com/multi-tasking-the-arduino-part-1
File Access (SD Cards)	- "Tutorial 3: SD Card and TFT (Shield)" tutorial, under the "TUTORIALS" main topic in the Proteus Visual Designer help file[22].

12.2. Implementation of Technology

Technology is a tool. Like any tool, its implementation depends on the wielder. As such, technology in itself will not make the world a better place – it is up to us to use it wisely!

[22] The Proteus Visual Designer help file can be opened from the Help menu when the Visual Designer tab is open and selected, or from the "Help" area on Proteus Home Page tab.

Appendix A: Challenges

Challenges are included to practice with, and to prove that concepts have been *understood* (as opposed to just copied down). You will likely find that you will need to re-read certain sections of a chapter in order to complete a challenge; this will further assist with learning and understanding. Note that challenges are not a pre-requisite to progress through the book (which is why they are included as an appendix rather than in-line with the book chapters), so if you can't complete a challenge then don't worry or let it hold back your progress through the book – you can always come back to it later.

Possible solutions to the challenges can be found in Appendix B. Remember that there are potentially many different ways to solve a challenge in programming; so if your solution does not look exactly like the one listed then do not worry! The important thing is whether the program works or not (within reason; if a chapter covered an advanced technique for achieving something then you should obviously try use that technique rather than a simpler solution).

General Procedure for Challenges

1. Save ( ) any existing project which you may have open (if you do not want to lose it). This can be done using the "Save Project" command available from the File menu.
2. Close ( ) any existing project which you may have open. This can be done using the "Close Project" command available from the File menu. You should now see the Proteus Home Page ( ) tab (as illustrated in Figure 2-1).
3. Unless indicated otherwise (some challenges may not be flowchart related, but most are), start a new Flowchart project (section 2.2) – the challenge name can be used as the project name.

Other procedures which may be necessary will have been covered in the chapter (and/or previous chapters) to which the project relates.

Chapter 2 (Creating Your First Project)

Heartbeat on LED

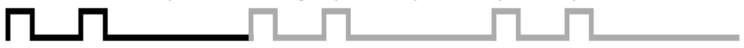
Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard

Challenge

Using the built-in LED on the Arduino Uno, design a flowchart which blinks the LED twice quickly, then delays before repeating again (looping). This is meant to visually mimic the sound of a human heart beating.

This can be represented graphically as (repeat cycles shown in gray):



The timings don't need to be exact – you can use whatever seems good to you. Suggested values are 250 milliseconds on, 500 milliseconds off, 2 second delay.

Chapter 3 (Reading Inputs and Making Decisions)

Blink on Demand

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Grove Momentary Action Push Button	Peripheral Gallery (Grove Category)
1	Grove LED (Green)	Peripheral Gallery (Grove Category)

Challenge

Design a flowchart which blinks an LED, but only when a button is pressed (the LED should be off otherwise).

The timing of the blinking is not terribly important, but a suggestion is 1 second on and 1 second off.

Blink Frequency Change

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Grove Momentary Action Push Button	Peripheral Gallery (Grove Category)
1	Grove LED (Green)	Peripheral Gallery (Grove Category)

Challenge

Design a flowchart which blinks an LED slowly when a button is not pressed, and faster when it is pressed.

Suggested timings are 1 second on / 1 second off for the slow blink, and 250 milliseconds on / 250 milliseconds off for the fast blink.

Tips

The “toggle” method of the Grove LED can be used to switch it on if it is off, and off if it is on. You don’t *need* to use this method, however it can potentially make the flowchart shorter and simpler.

Chapter 4 (Variables)

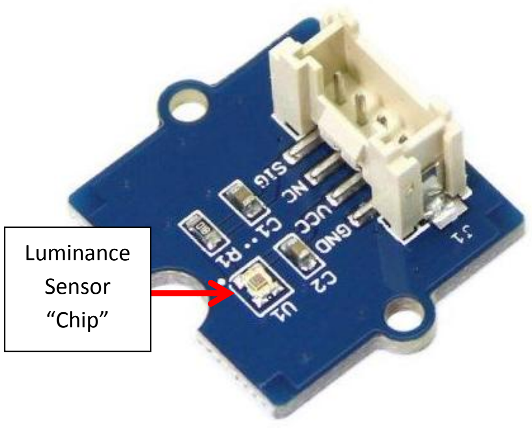
Nightlight

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Grove Luminance Sensor	Peripheral Gallery (Grove Category)
1	Grove LED (Green)	Peripheral Gallery (Grove Category)

Challenge

Design an automatic “nightlight” – a light which automatically comes on when it gets dark. The Grove Luminance Sensor can be used to measure whether it is light or dark; for this challenge a luminance reading from the sensor of less than 300 can be considered “dark”, otherwise consider it to be “light”. Our nightlight will be an LED (i.e. switch the LED on when the luminance sensor measures that it is dark, and switch the LED off when the luminance sensor measures that it is light).

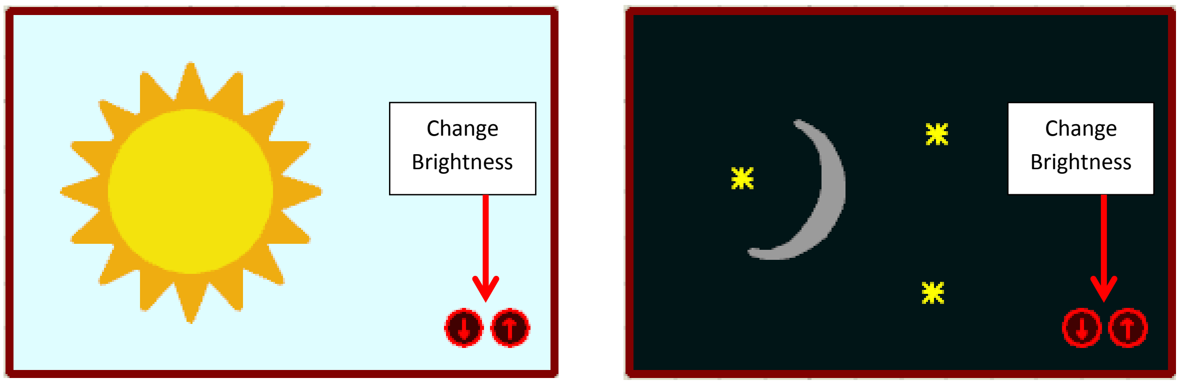


A-1: Grove Luminance Sensor

Tips

The "readLuminance" method of the Grove Luminance Sensor can be used to read the current luminance (light level). This method will by default automatically place the reading into a new variable called "Lux", which you can use elsewhere in your flowchart (such as in a decision block).

The luminance sensor is represented by a window, which shows a sun when it is light and stars when it is dark. To change the brightness, use the up and down arrows at the bottom-right of the window.



Countdown to Blastoff

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Grove RGB LCD Module	Peripheral Gallery (Grove Category)
1	Grove Momentary Action Push Button (Optional Bonus)	Peripheral Gallery (Grove Category)

Challenge

Design a flowchart which counts down from 10 to 0 (with 1 second delays), displaying the count on the top line of an LCD. When 0 is reached, then display the text "Blastoff!" on the bottom line of the LCD. Optional Bonus: Have a button which can be pressed to add 3 to the count.

Tips

When you print something to an LCD, it only prints as many characters as are strictly necessary, and leaves the rest of the display unchanged. This is a very useful feature if you only want to update a specific part of the LCD (without changing the rest of it), but it can also result in some confusing scenarios! For example; if you first print “10” (two characters) to the LCD, and then try to print “9” (one character) at the same position, then the resulting display will be “90”! This is because in order to print the “9” the LCD only strictly needed to change one character, and it left the rest of the display (including the “0” 2nd character from “10”) unchanged. There are two possible solutions to this:

The first possible solution is to wipe the whole LCD clean before displaying anything on it. This can be done using the LCD “clear” method (normally at the top of your loop).

The second possible solution is to print blank characters (i.e. spaces; " ") over anything which you would like cleared. So, for example, you could first print two blank characters over the “10” in order to clear it, and then print the “9” in the now blank space. Note that the printing of the clearing characters and the printing of the “9” would be done in two separate flowchart blocks, with a “setCursor” or “home” block in-between.

For the optional bonus, note that it is expected that the button will need to be “held in” for up to 1 second in order for it to “take effect”. There are ways to work around this (such as breaking the delay up into smaller chunks), however they are not required for this challenge.

Chapter 6 (Drawing and Simulating Circuits)

Ohm’s Law

Starting the Project

Note that this project does not require a flowchart. For the procedure for starting a project like this please see section 6.4.

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	RES (Resistor), value set to 5R (5 ohms)	Schematic Components Tool Mode (Pick from Library)
1	POWER Terminal, set to +10V (10 volts)	Schematic Terminals Tool Mode
1	GROUND Terminal	Schematic Terminals Tool Mode
1	DC Ammeter	Schematic Virtual Instruments Tool Mode

Challenge

Given a voltage of 10 volts, and a resistance of 5 ohms, calculate the resulting current in amps (according to Ohm’s Law). Draw a circuit in Proteus to verify your calculation by simulation.

Chapter 7 (Drawing Custom Peripherals)

Two LEDs, One Pin

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard

2	RES (Resistor), value set to 330R	Schematic Components Tool Mode (Should already be in Object Selector)
1	LED-RED	Schematic Components Tool Mode (Pick from Library)
1	LED-GREEN	Schematic Components Tool Mode (Pick from Library)
2	DEFAULT Terminal	Schematic Terminals Tool Mode
1	GROUND Terminal	Schematic Terminals Tool Mode

Challenge

Draw a custom circuit to interact with your Arduino Uno. Connect two LEDs (one LED-RED and one LED-GREEN), with current-limiting resistors (330R), to pin "IO7" on your Arduino Uno (both LEDs connected to the same pin). Construct a flowchart which switches pin IO7 on the Arduino Uno on and off, thus blinking the LEDs (1 second delays). Admittedly, this project is not very practical in "real-life"; it is however good for practice!

Chapter 8 (Sub-Routines and Conditional Loops)

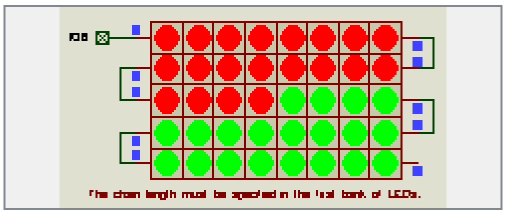
TwoTone

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Adafruit NeoPixel Shield	Peripheral Gallery (Adafruit Category)

Challenge

Set the first 20 pixels on the NeoPixel shield as "red", and the remaining 20 pixels as "green".


IO8
The chain length must be specified in the last bank of LEDs.

Tips

The first 20 pixels are number 0-19, and the remaining 20 pixels are numbered 20-39.

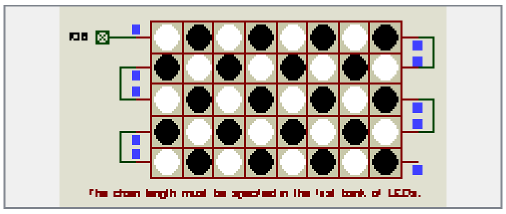
Checkerboard

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Adafruit NeoPixel Shield	Peripheral Gallery (Adafruit Category)

Challenge

Display a checkerboard (alternating white and black pixels) on the NeoPixel shield.


The chain length must be specified in the last bank of LEDs.

Tips

A white pixel is generated when all components colours (red, green and blue) are set to full intensity (255). A black pixel is generated when all component colours (red, green and blue) are set to zero intensity (0).

As with most programs, there are multiple ways to solve this challenge (one of the below techniques could be used, not both):

a) A for loop with a “Step” value of 2 can be used to count “0, 2, 4, etc”.
b) A mathematical equation like “intensity=255-intensity” can be used to toggle “intensity” between 255 and 0.

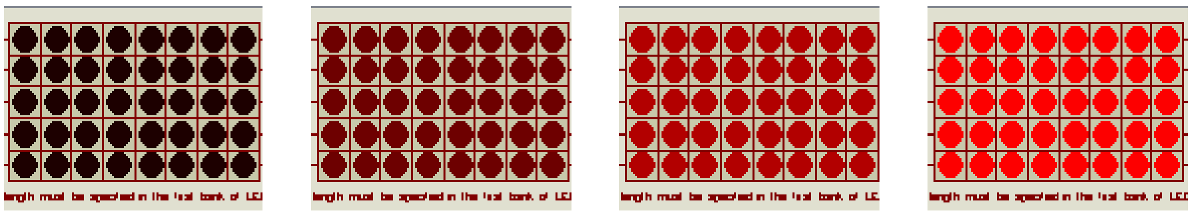
Pulsing Glow

Parts Needed

Qty	Part	Added to Proteus Design By / From
1	Arduino Uno	New Flowchart Wizard
1	Adafruit NeoPixel Shield	Peripheral Gallery (Adafruit Category)

Challenge

Using a red colour, take all of the pixels on the NeoPixel shield smoothly from zero intensity (red=0) to full intensity (red=255), and then smoothly back down again (and repeat). Optional Bonus: Instead of pulsing just a red glow, pulse a red then a green and then a blue glow (and repeat).



Tips

A delay of 5ms between display updates (“show” method) is recommended.

For the Optional Bonus:

The intensity of a pixel can be scaled using an equation like “r=red*intensity/255”, where “intensity” is a variable somewhere between 0 and 255. Looking at this equation:

- If intensity is 255 then there will be no effect on the “red” variable (255 / 255 = 1, and anything multiplied by 1 stays as it was).

- If intensity is 0 then “r” will become 0 (0 / 255 = 0, and anything multiplied by 0 becomes 0).
- If intensity is 127 then “r” will be about half of “red” (127 / 255 = approximately 0.5, and multiplying by 0.5 is the same as dividing by 2).

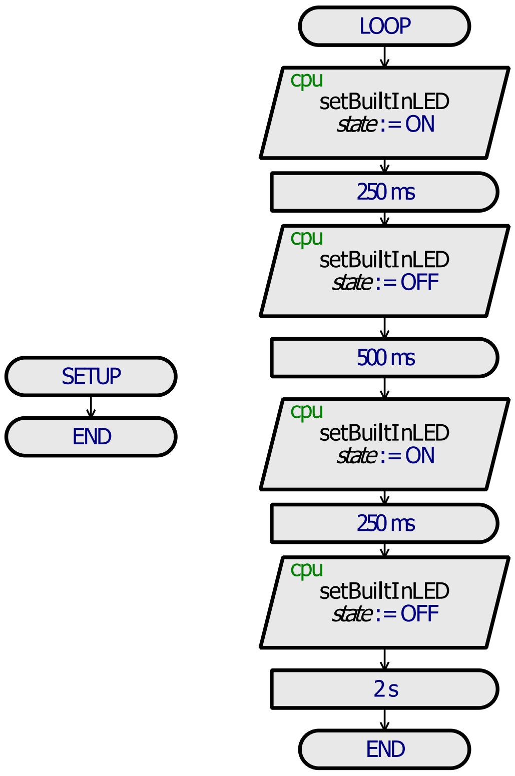
Note that “red” does not need to be 255 in this equation – any value will “work” (so more colours could potentially be “glowed” than just red, green and blue).

Appendix B: Solutions to Challenges

This appendix contains possible solutions to the challenges posed in Appendix A. Remember that there are potentially many different ways to solve a challenge in programming; so if your solution does not look exactly like the one listed then do not worry! The important thing is whether the program works or not (within reason; if a chapter covered an advanced technique for achieving something then you should obviously try use that technique rather than a simpler solution).

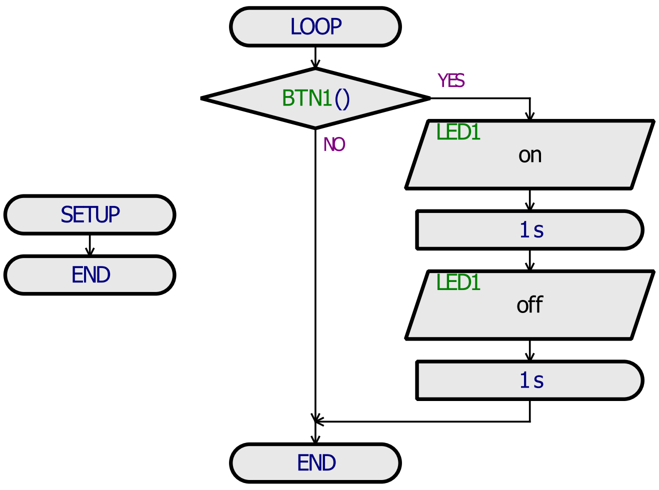
Chapter 2 (Creating Your First Project)

Heartbeat on LED

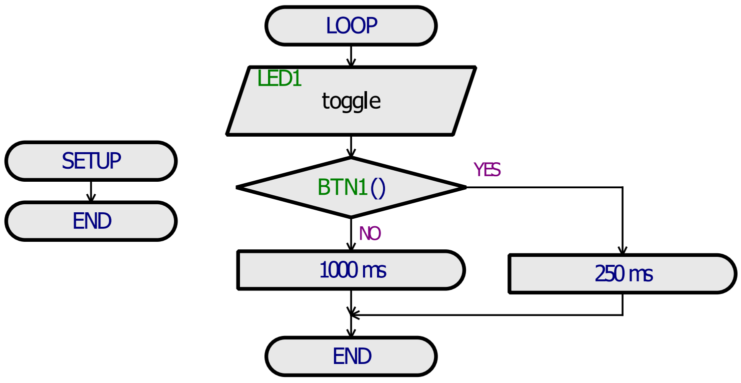


Chapter 3 (Reading Inputs and Making Decisions)

Blink on Demand

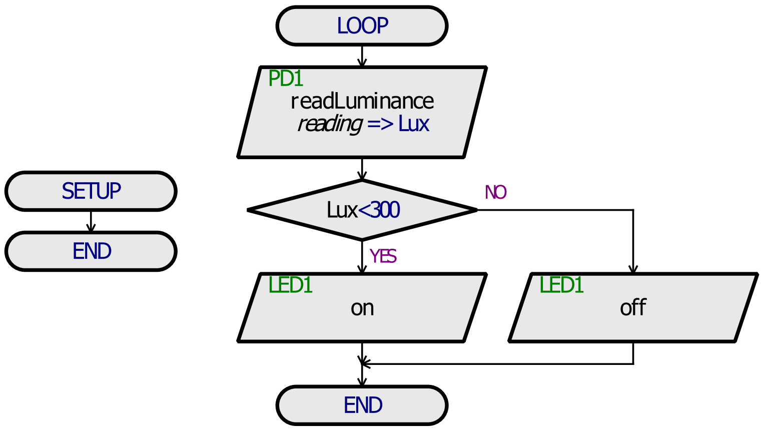


Blink Frequency Change

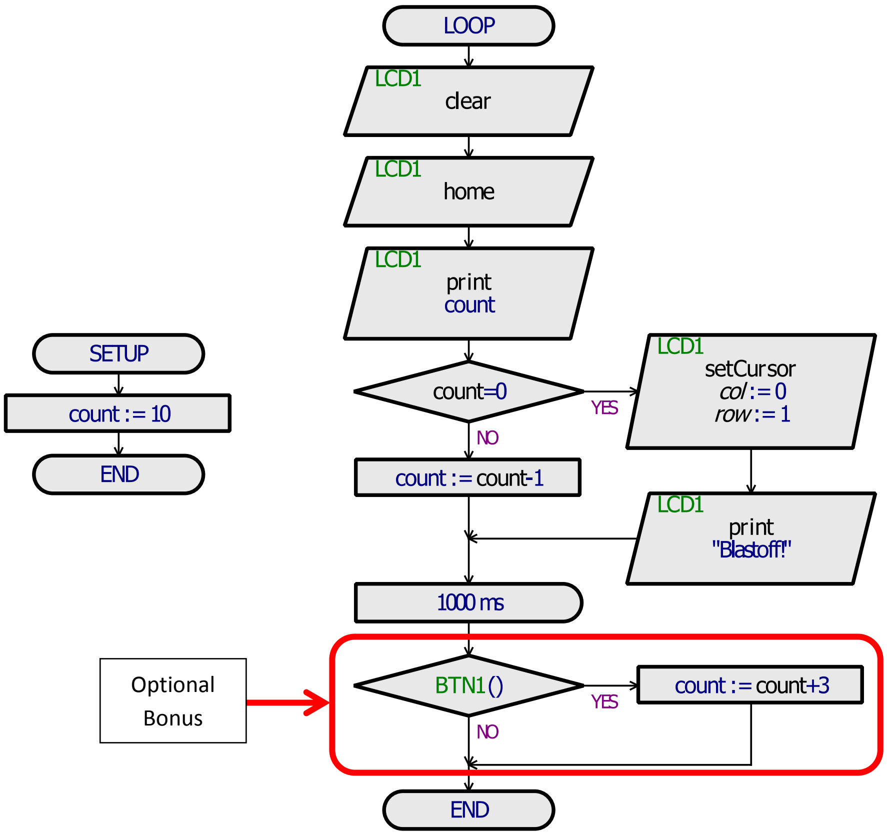


Chapter 4 (Variables)

Nightlight

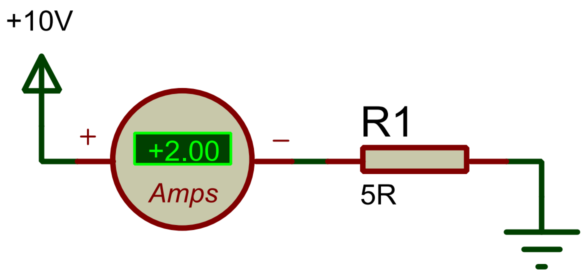


Countdown to Blastoff



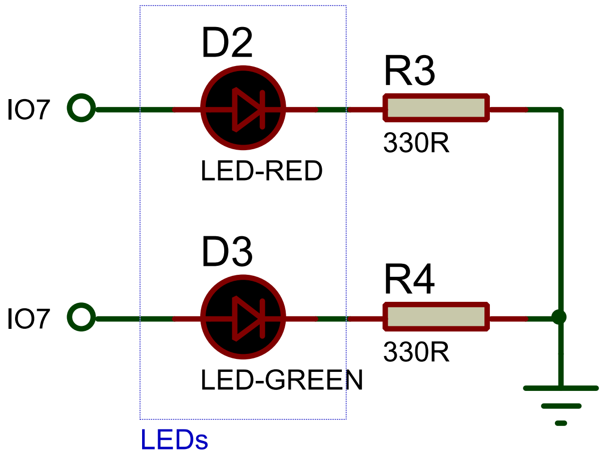
Chapter 6 (Drawing and Simulating Circuits)

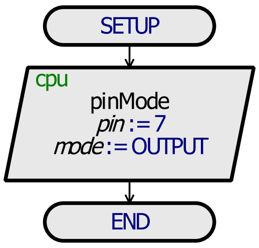
Ohm's Law

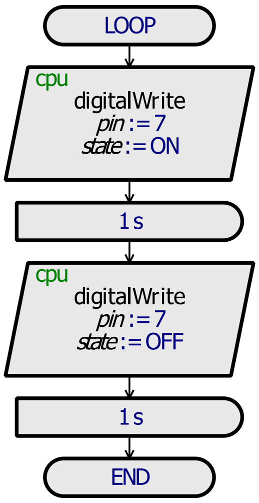


Chapter 7 (Drawing Custom Peripherals)

Two LEDs, One Pin

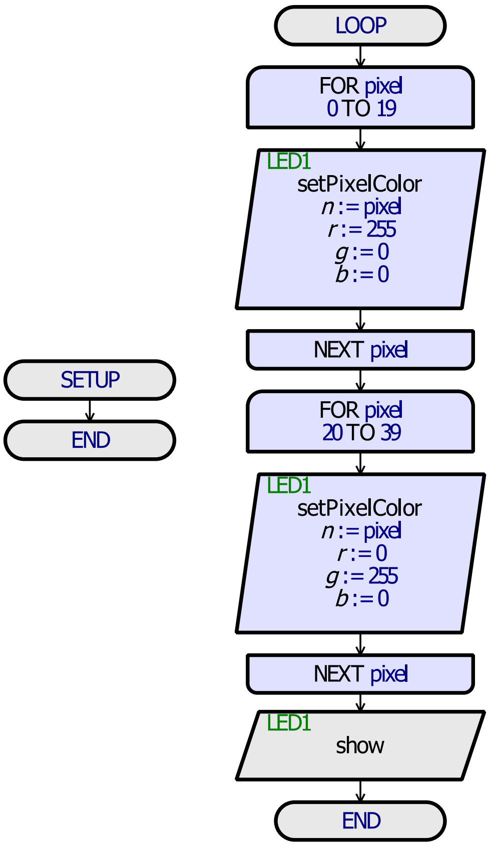




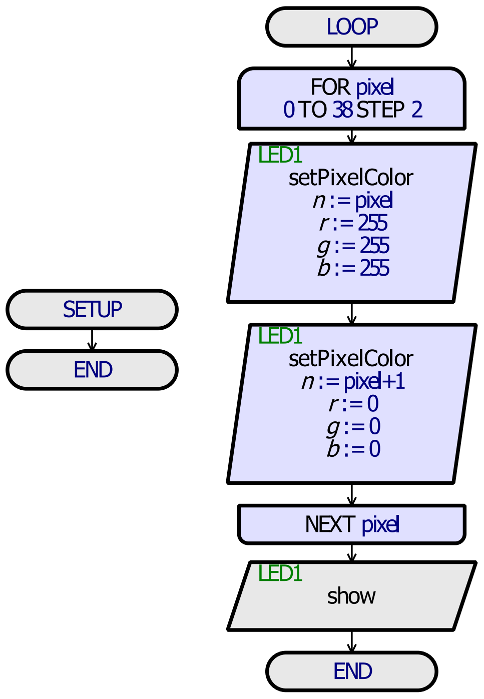


Chapter 8 (Sub-Routines and Conditional Loops)

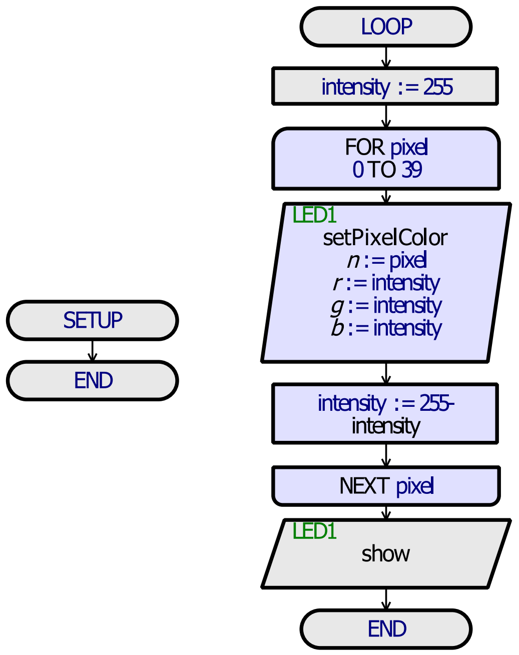
TwoTone



Checkerboard

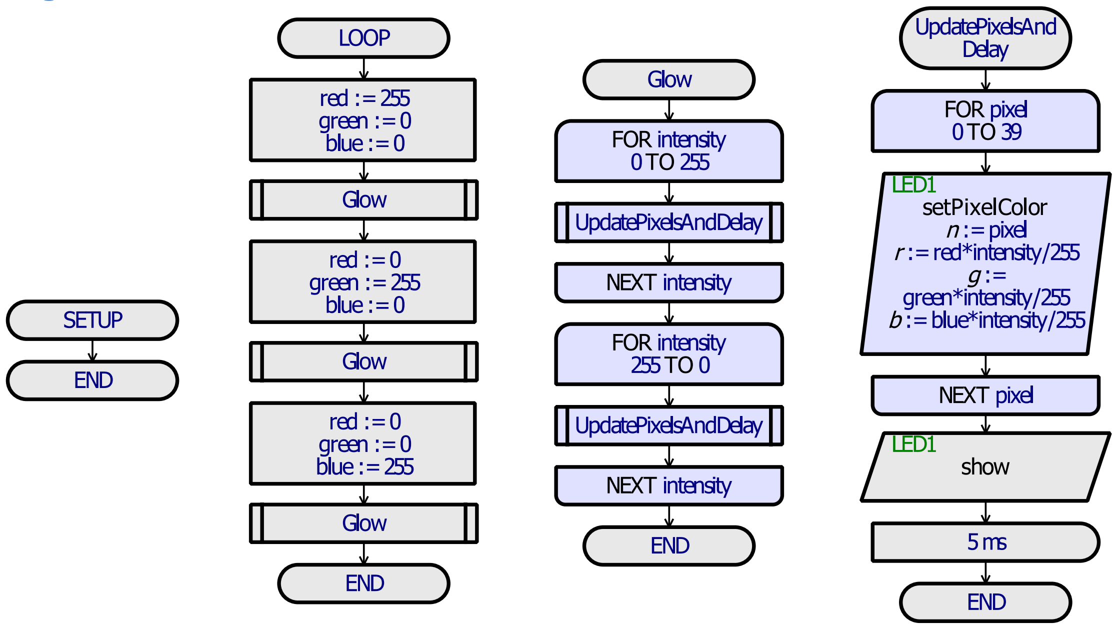


Possible Solution 1



Possible Solution 2

Pulsing Glow



Note: The flowchart depicted includes the Optional Bonus.