An Introduction to Embedded Systems and Applications

Through the Raspberry Pi

An Introduction to Embedded Systems and Applications

Through the Raspberry Pi

Anthony Bradburn

April 12, 2022

Website: https://github.com/ajbradburn/FundOfMicroPLabManual

[1] http://creativecommons.org/licenses/by-sa/4.0/

Feedback about the text

This lab manual is currently a work in progress, and likely to contain errors. Additionally, it is likely to age poorly unless maintained. To that end, it is made open source, available for forking and editing.

If you find errors, please check the version of the product you're using, to make sure it is the newest. If the errors exist in the most recent version, please email the maintainer at ajbradburn@gmail.com[2] and include answers to the following:

1. A description of the error, with screenshots/photo if possible.
2. Is the problem in the online version or the PDF version or both?
3. What is the URL for the online page, or page number in the PDF that contains the errors?

This document was created using PreTeXt[3]. It is a very useful tool, but does require a bit of a learning to get it to work. Thankfully, documentation is pretty good, and the project is currently being maintained. Basic instructions for cloning the project, making edits, and exporting your changes to useful formats are available in an appendix to this document.

[2] ajbradburn@gmail.com
[3] https://pretextbook.org/

Contents

Chapter 1

Setting up for, and exploring assembly programming.

In order to interface with the processor, we are going to need to write programs. Writing programs can be a challenging process that requires a great deal of knowledge, and is prone to mistakes. A simple human error can result in all sorts of unanticipated behavior.

There are tools, called an Interactive Development Environment or **IDE** that make this process of writing code and working through errors much easier. We will install one such IDE that will be used for some of the labs in this book.

1.1 Installing Code::Blocks IDE in Raspberry Pi OS

1.1.1

For our purposes, we are going to use Code::Blocks IDE. To install it:

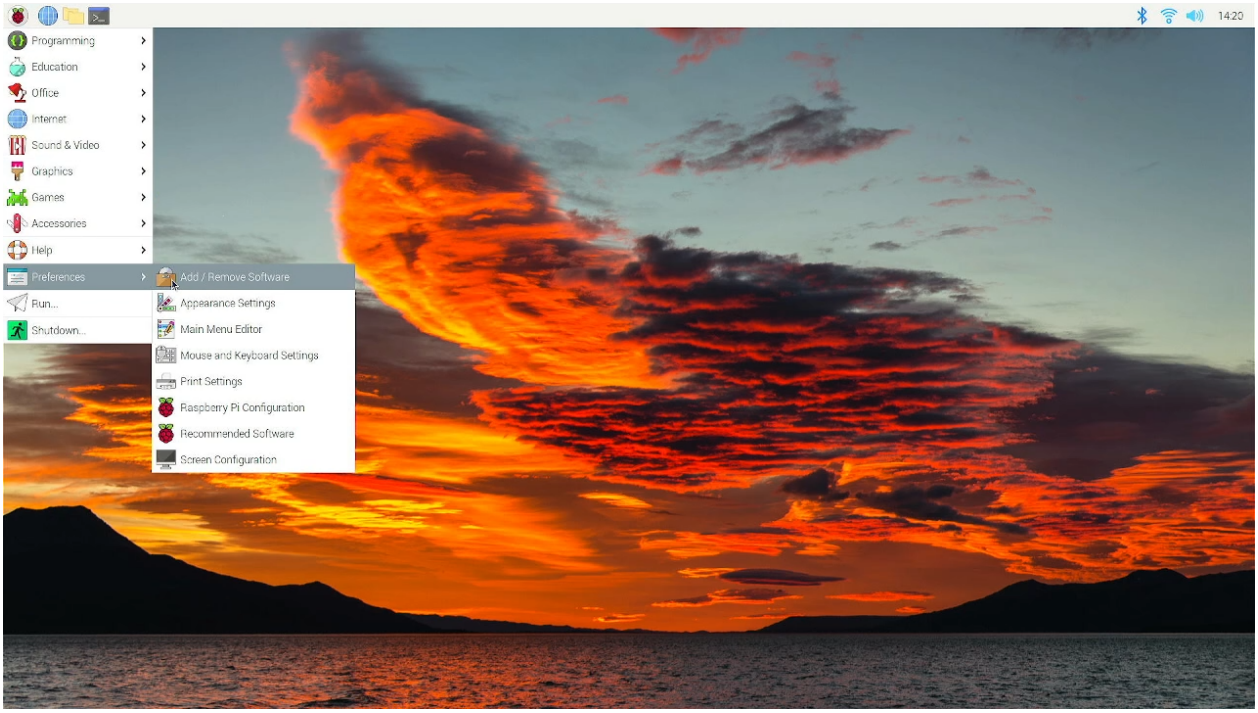

Figure 1.1.1 Opening Add / Remove Software

1. Click *main menu* (Raspberry Pi Icon on the Taskbar)
2. ... *Preferences*
3. ... *Add / Remove Software*

1.1.2

When the *Add / Remove Software* window opens.

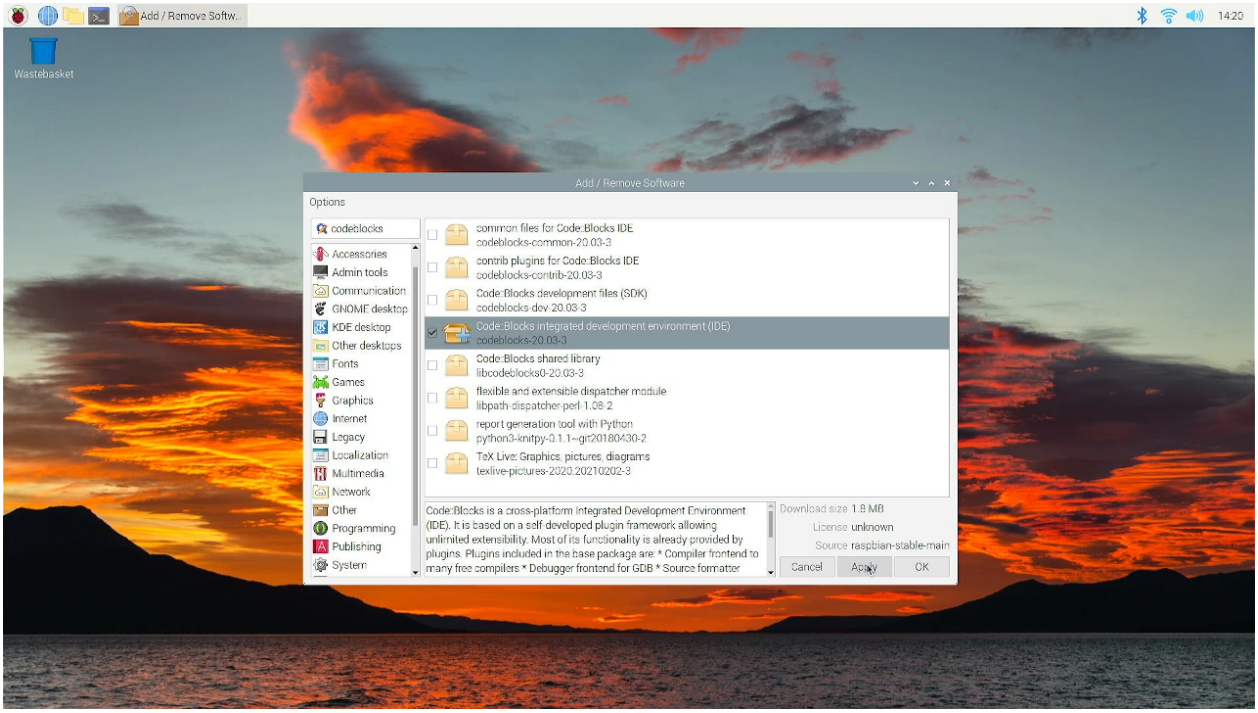

Figure 1.1.2 Selecting software in Add / Remove Software

1. Type ”codeblocks” in the search field at the top left
2. Wait for the list of software to load
3. Check the box for *Code::Blocks integrated development environment (IDE)*
4. Click *Apply*

1.1.3

Next you’ll be prompted with an *Authentication* window, where you’ll need to type in the password you’ve recently created, and click *OK*.

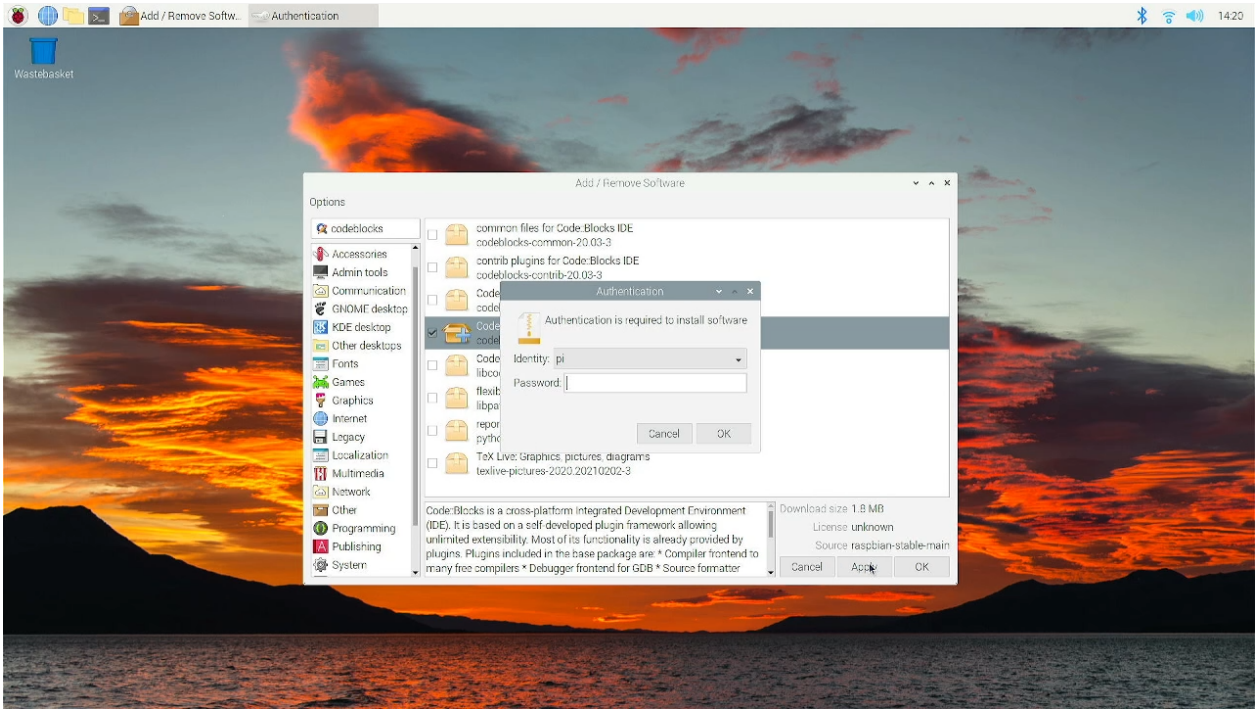

Figure 1.1.3 Authentication Window

Finally, wait a while while the software installs. When complete, click *OK* to close the *Add / Remove Software* window.

1.2 Writing and Debugging Programs

We will go into more detail about the process of programming in later exercises. For now, the expectation is that you will simply follow steps as you are guided through the process of using Code::Blocks IDE to write and debug a simple program. The goal with this exercise is to simply expose you to the process as a whole. To see, and do.

1.2.1

To begin, open *Code::Blocks IDE*.

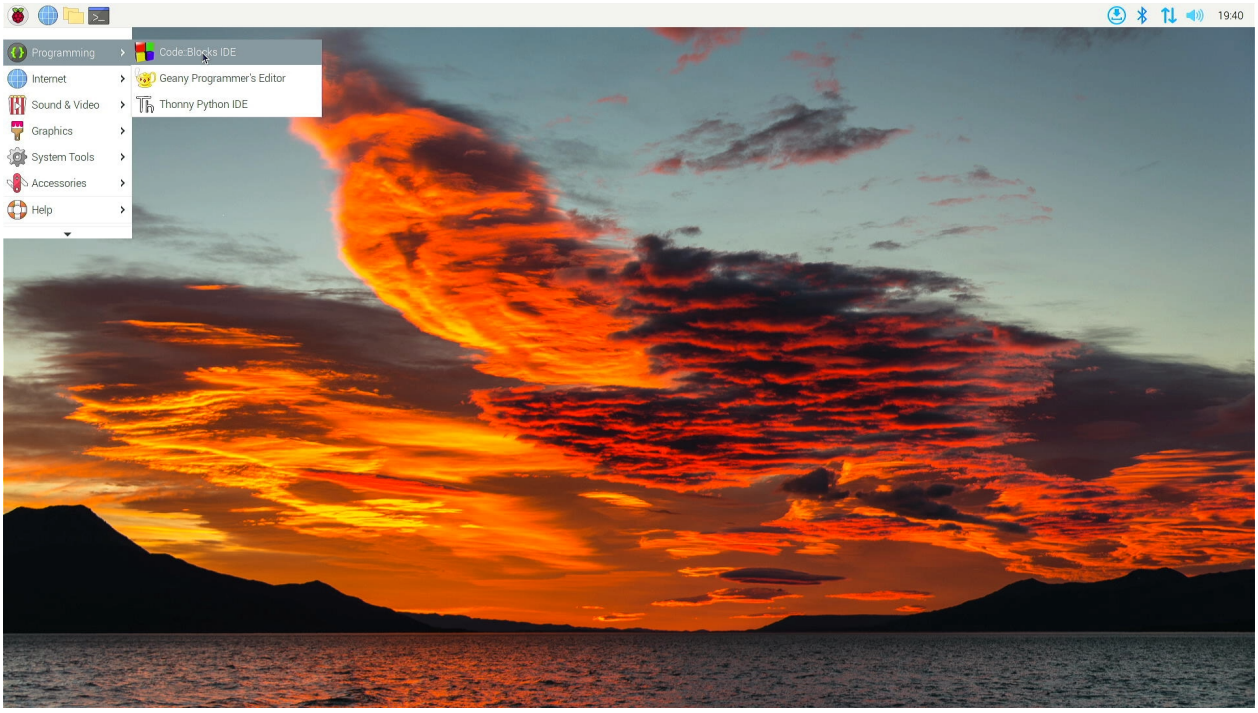

Figure 1.2.1 Opening Code::Blocks IDE

1. Click *main menu* (Raspberry Pi Icon on the Taskbar)
2. ... *Programming*
3. ... *Code::Blocks IDE*

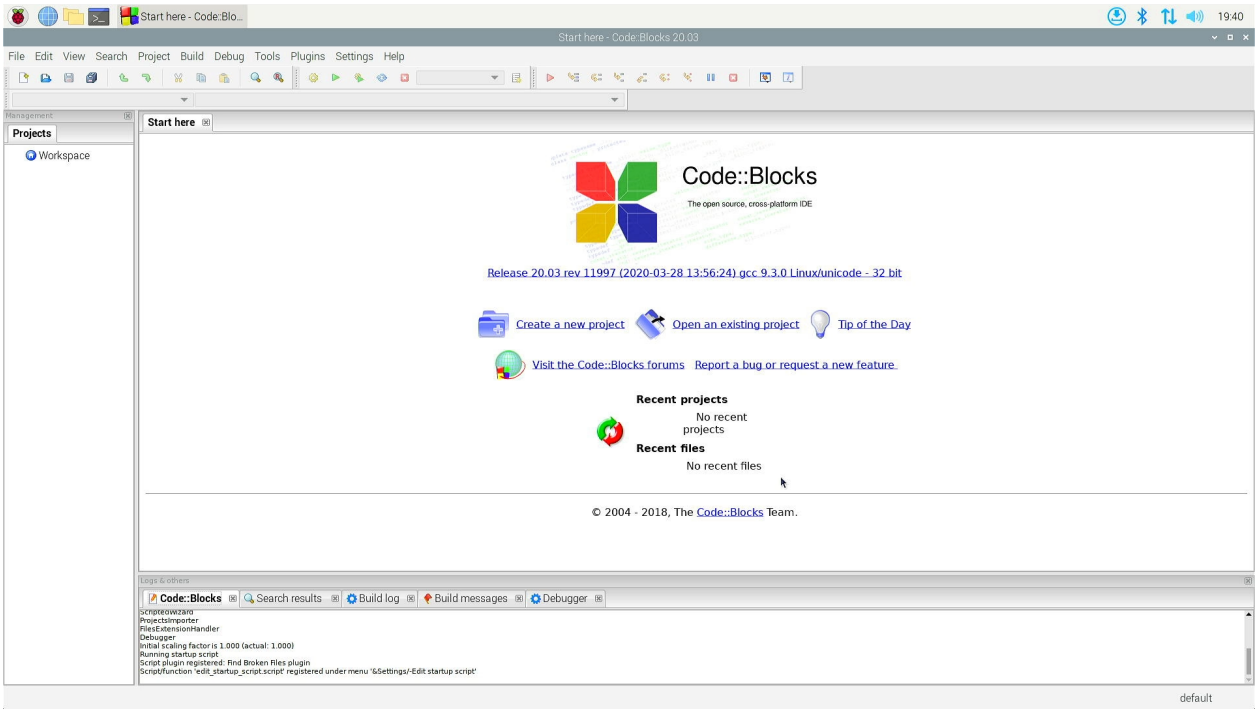

Figure 1.2.2 Code::Blocks IDE Layout

1.2.2

Create a new project.

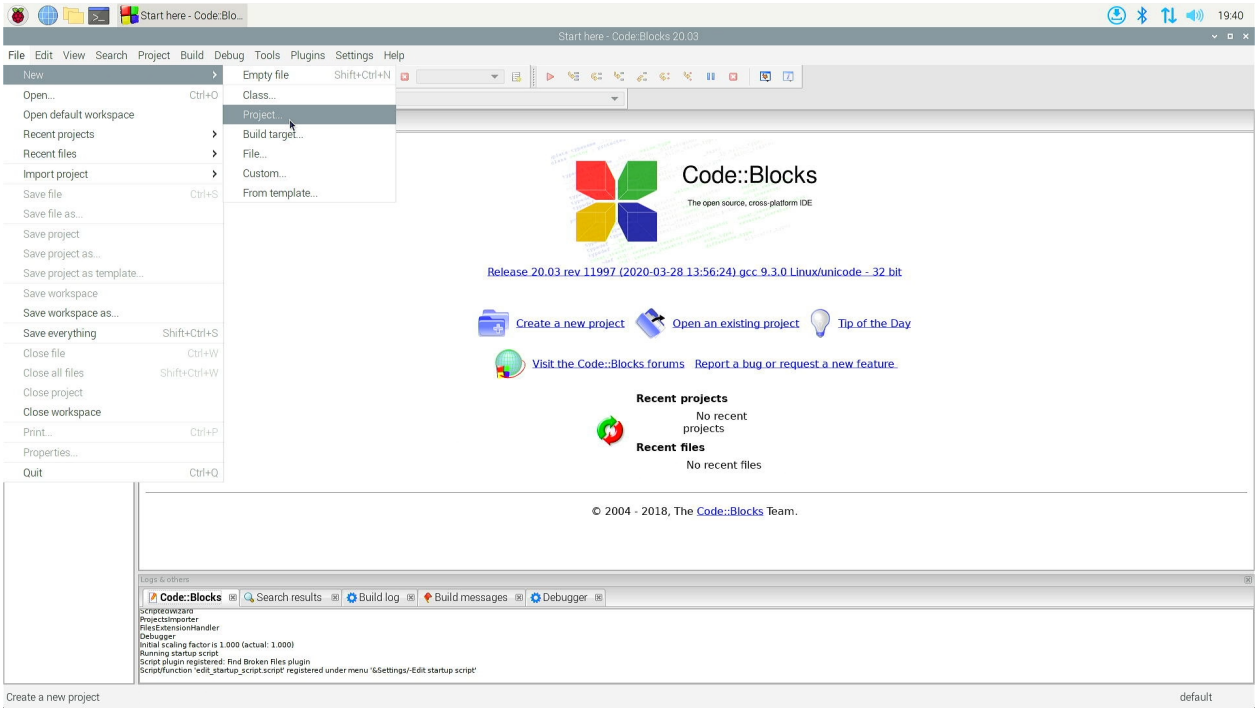

Figure 1.2.3 Creating a new project in Code::Blocks IDE

1. Click *File*
2. ... *New*
3. ... *Project...*

1.2.3

When the *New from template* window opens.

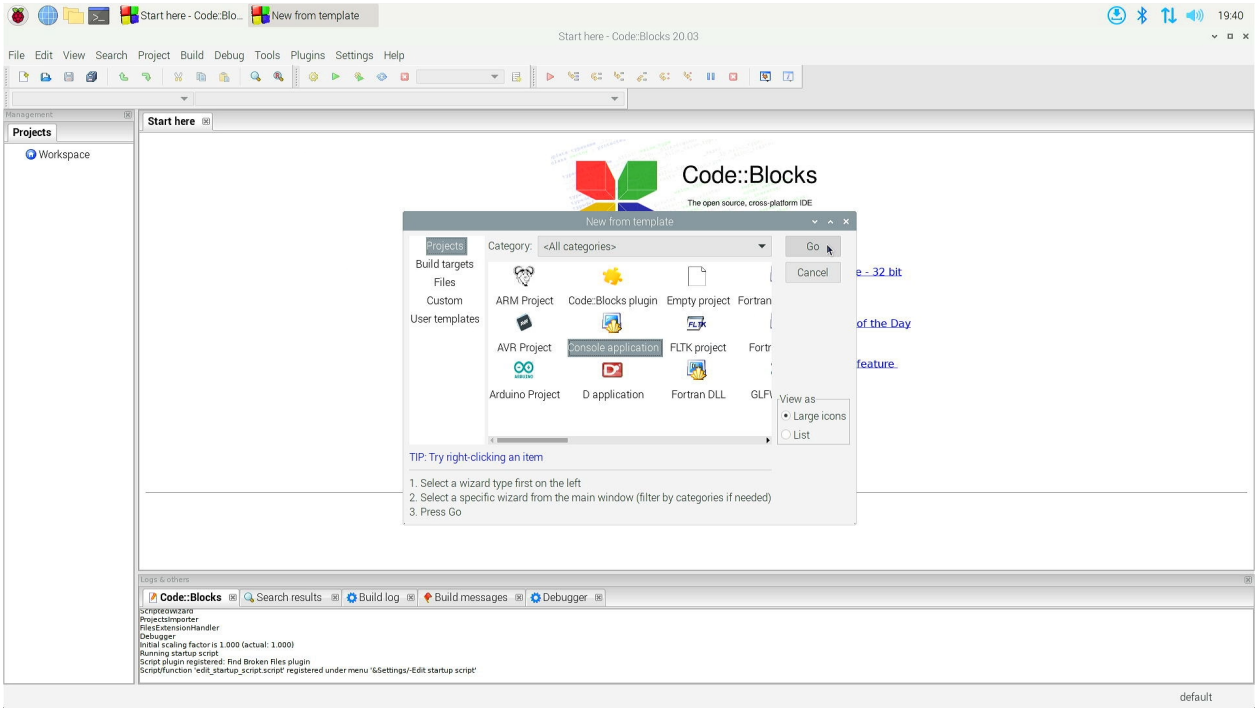

Figure 1.2.4 Select *Console application* for your new project.

1. Select *Console application*
2. Click *Go*

1.2.4

When the *Console application wizard* window opens

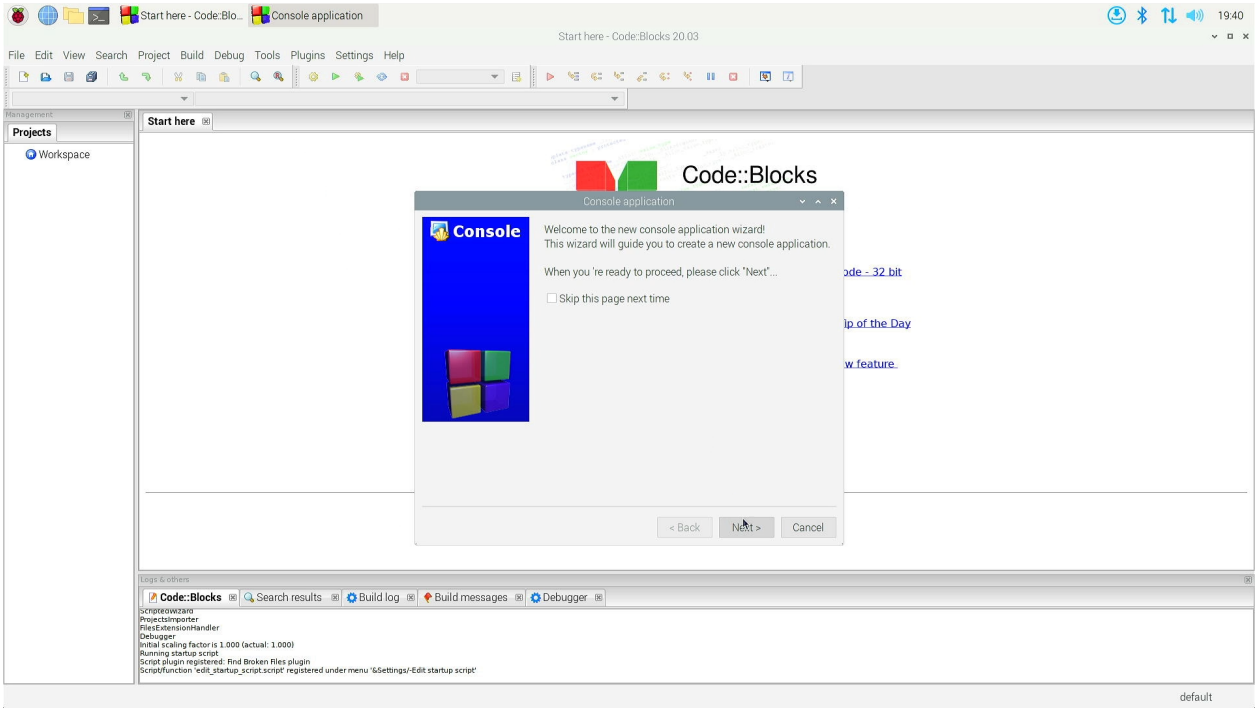

Figure 1.2.5 The console application wizard introduction screen.

1. Click *Next*

1.2.5

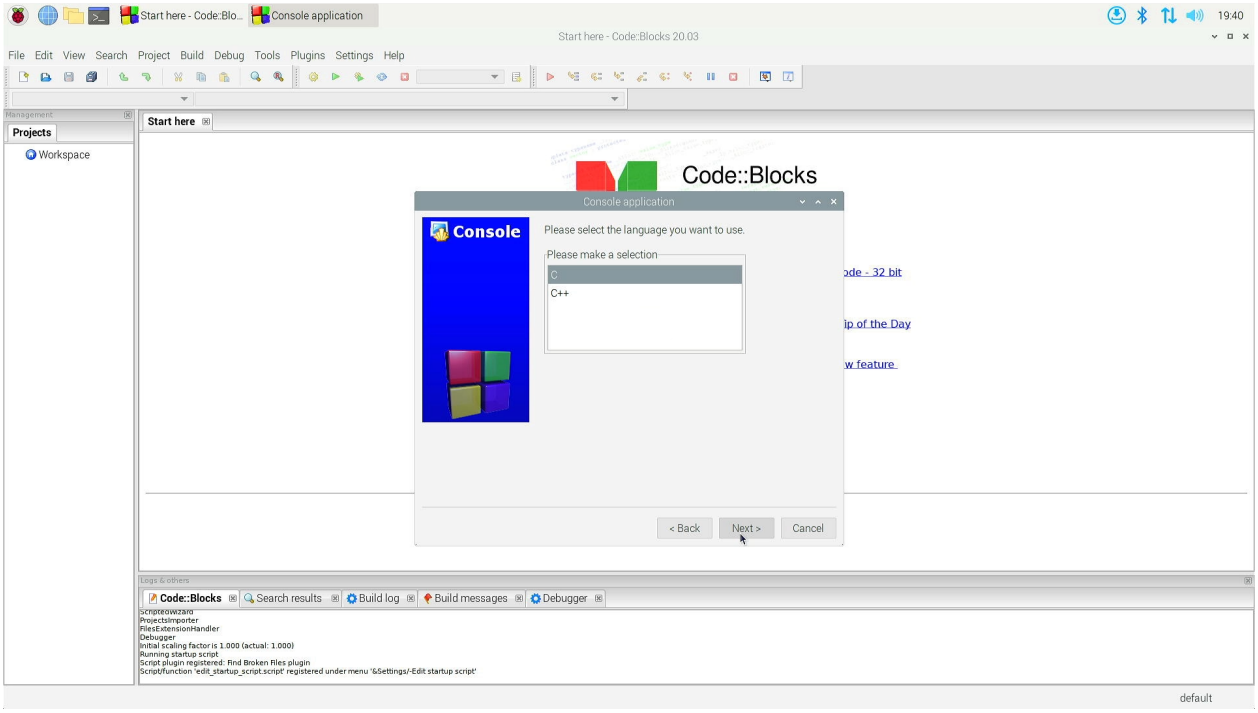

Figure 1.2.6 The console application wizard language selection window.

1. Select *C*

2. Click *Next*

1.2.6

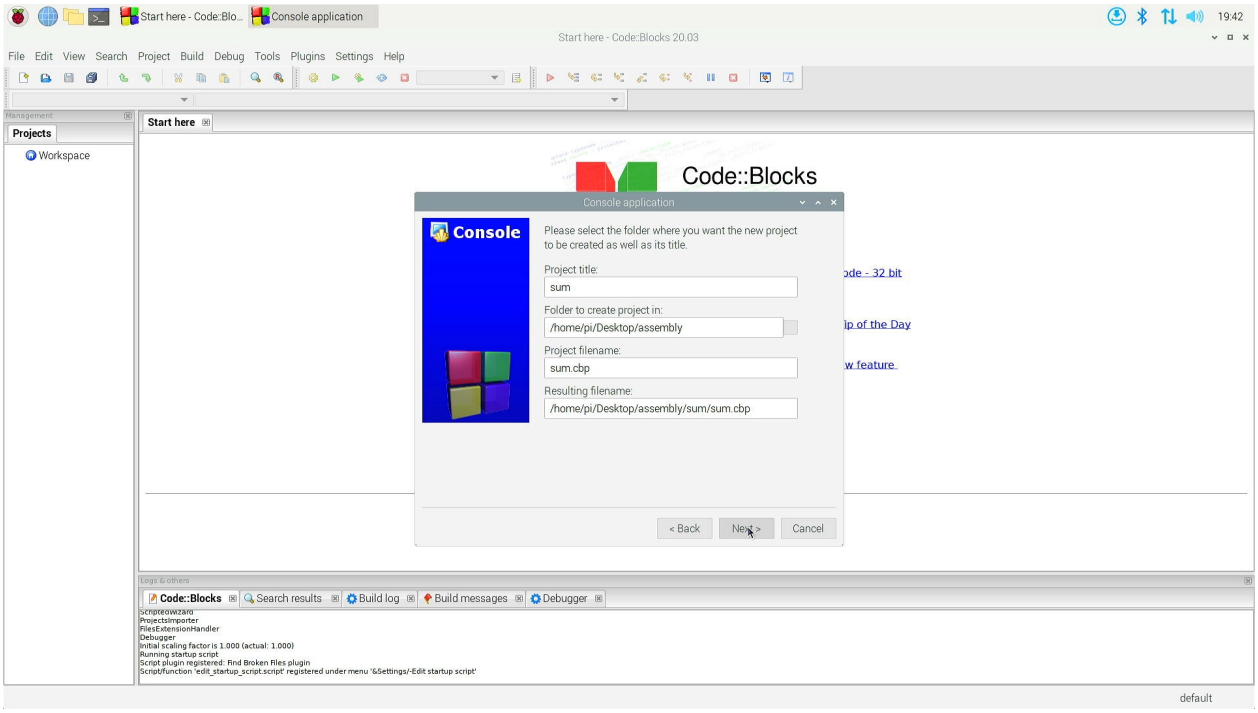

Figure 1.2.7 The console application wizard project title and save location window.

1. Type the project name, or title in the *Project title:* field. In this case, *sum*

2. Enter the path to where your project will be saved in the *Folder to create project in:* field. You can type this out, or click the *button* to the right of the field to select it using the file browser.

1.2.7

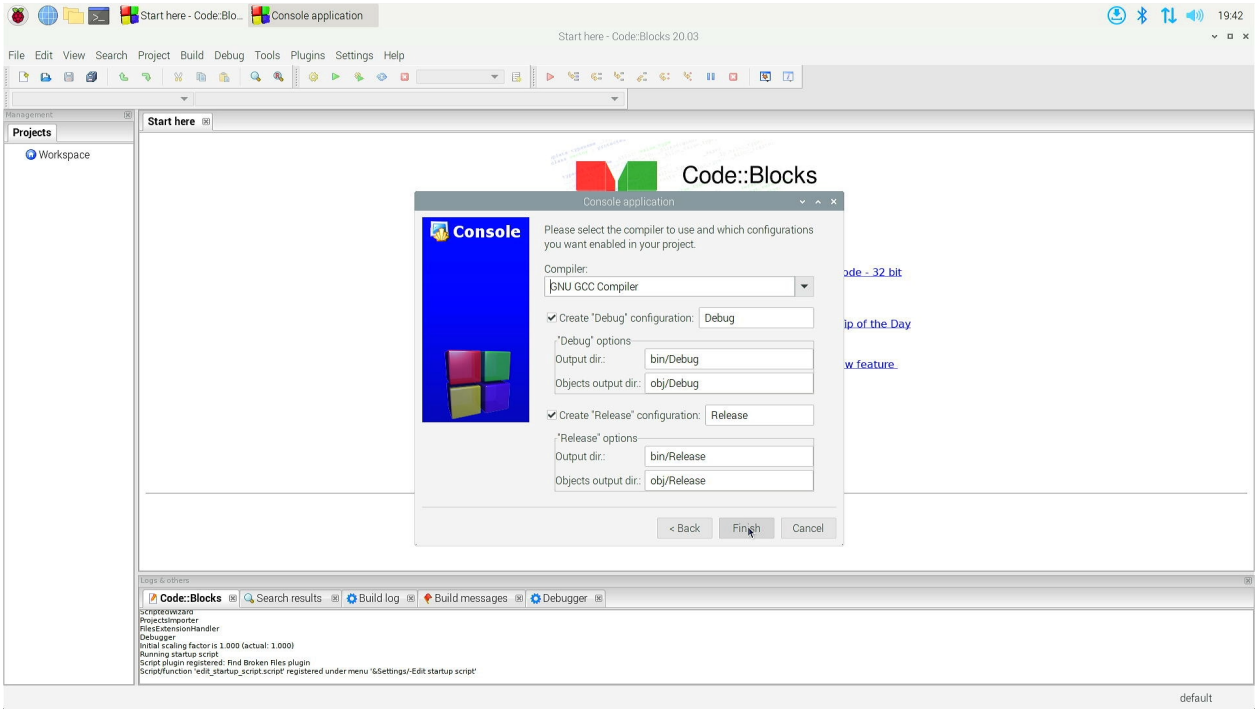

Figure 1.2.8 The console application wizard compiler configuration window.

1. Click *Finish*

1.2.8

Now that your project is established, we need to make a few customizations for it to work well with our assembly code.

The new project was setup with a main.c file that includes some framework code. We don't need that, so we are going to remove the file from the project, and create a new blank file for our program.

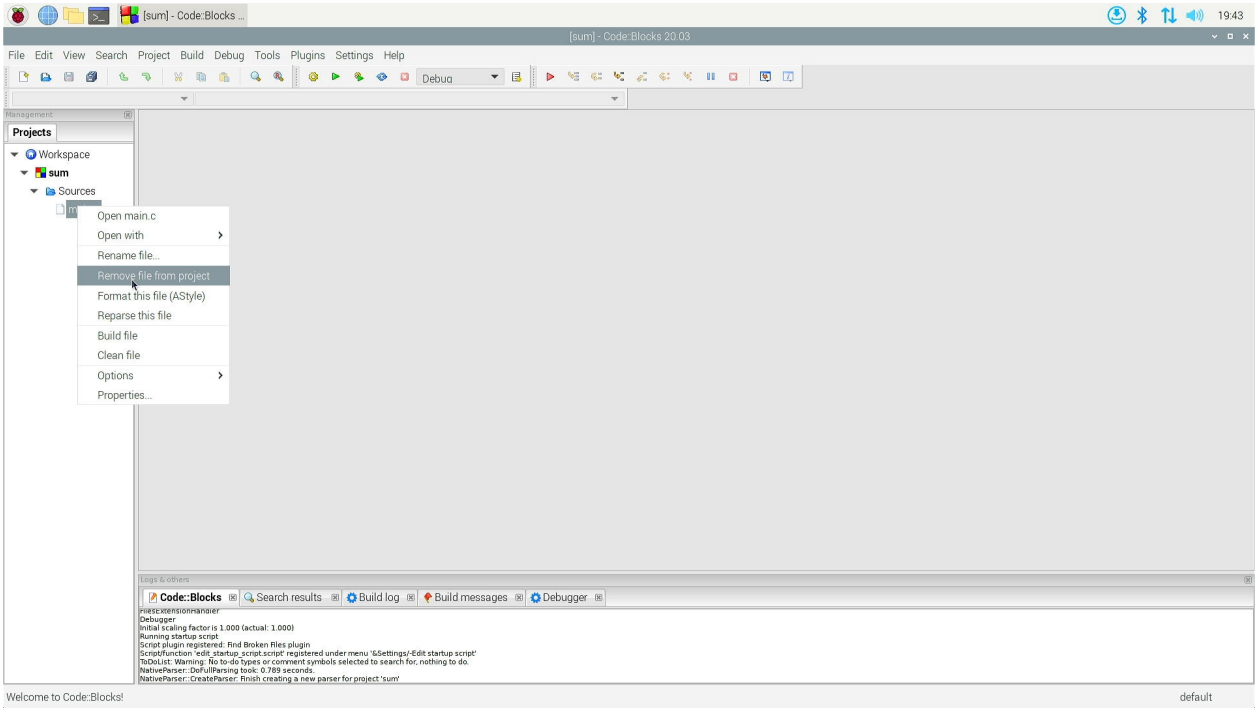

Figure 1.2.9

1. Click the triangle next to *Sources* to list its contents
2. Right-click *main.c*
3. Click *Remove file from project*

1.2.9

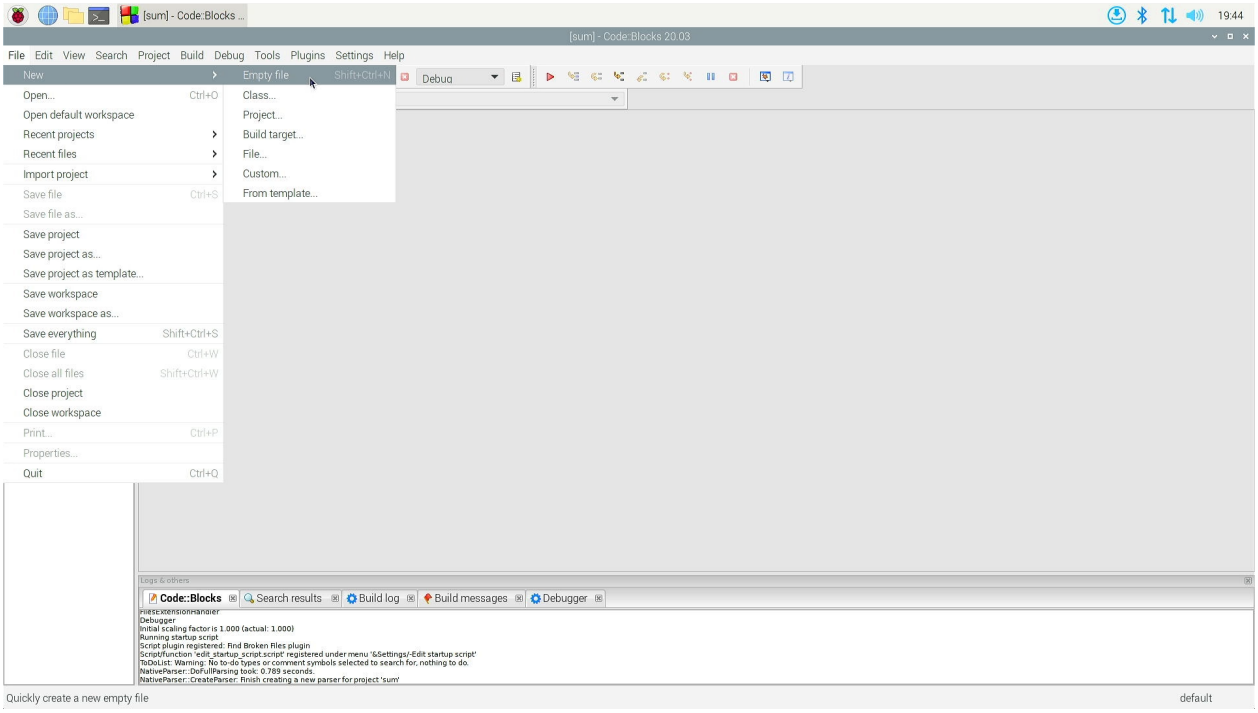

Figure 1.2.10 The navigation path for adding a new empty file using the file menu.

1. Click *File*
2. ... *New*
3. ... *Empty File*

1.2.10

When prompted to add file to the active project.

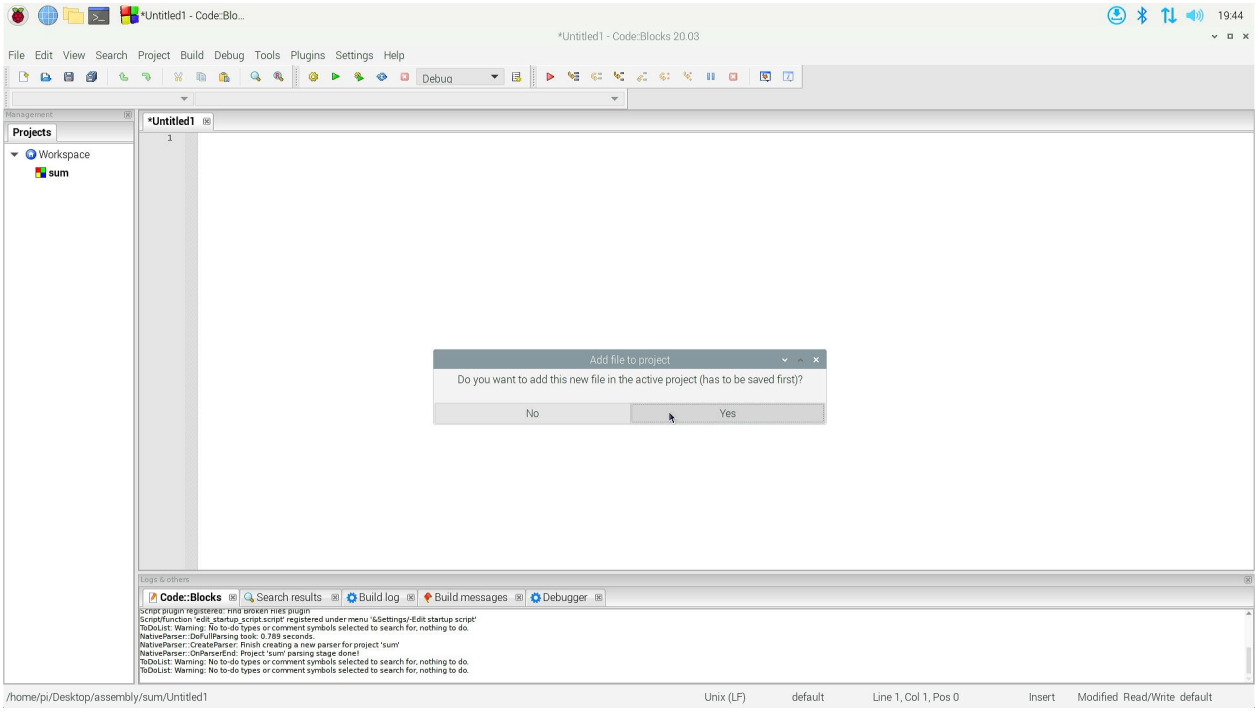

Figure 1.2.11 The *Add file to project* window.

1. Click *Yes*

1.2.11

When presented with the *Save file* window.

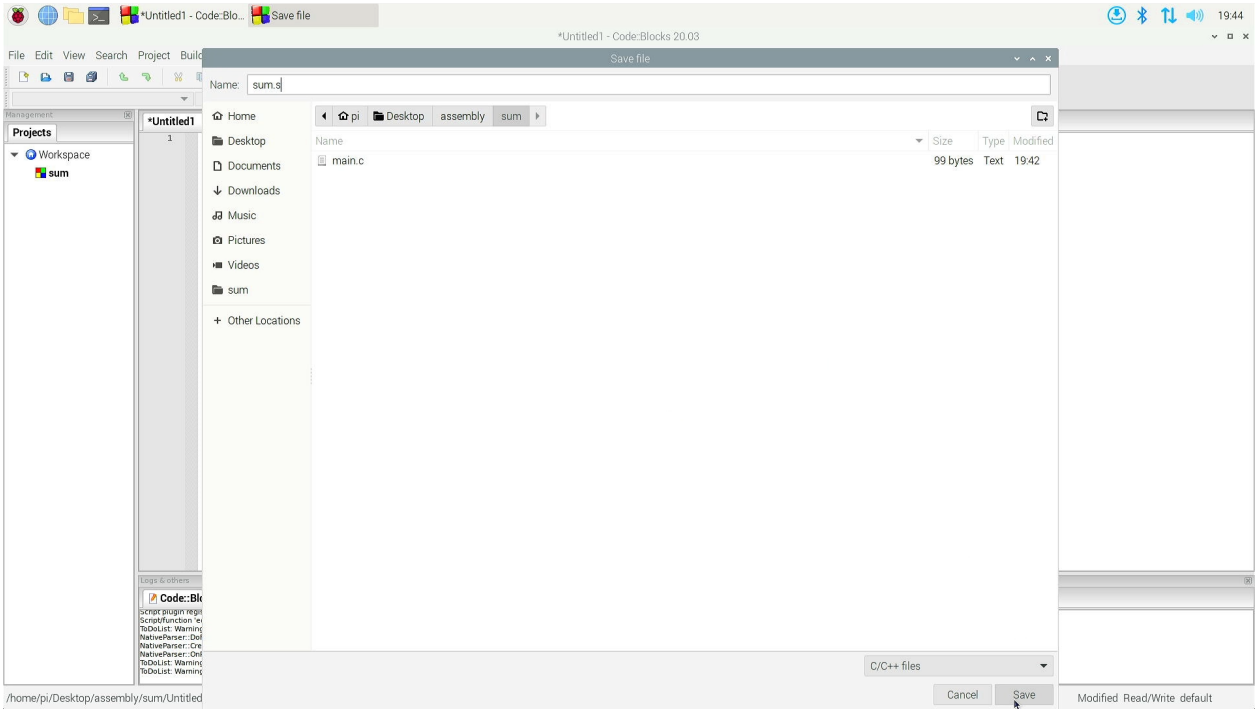

Figure 1.2.12 The *Save file* window.

1. Type the name of the file, in this case *sum.s*
2. Click *Save*

1.2.12

When presented with the *Multiple selection* window.

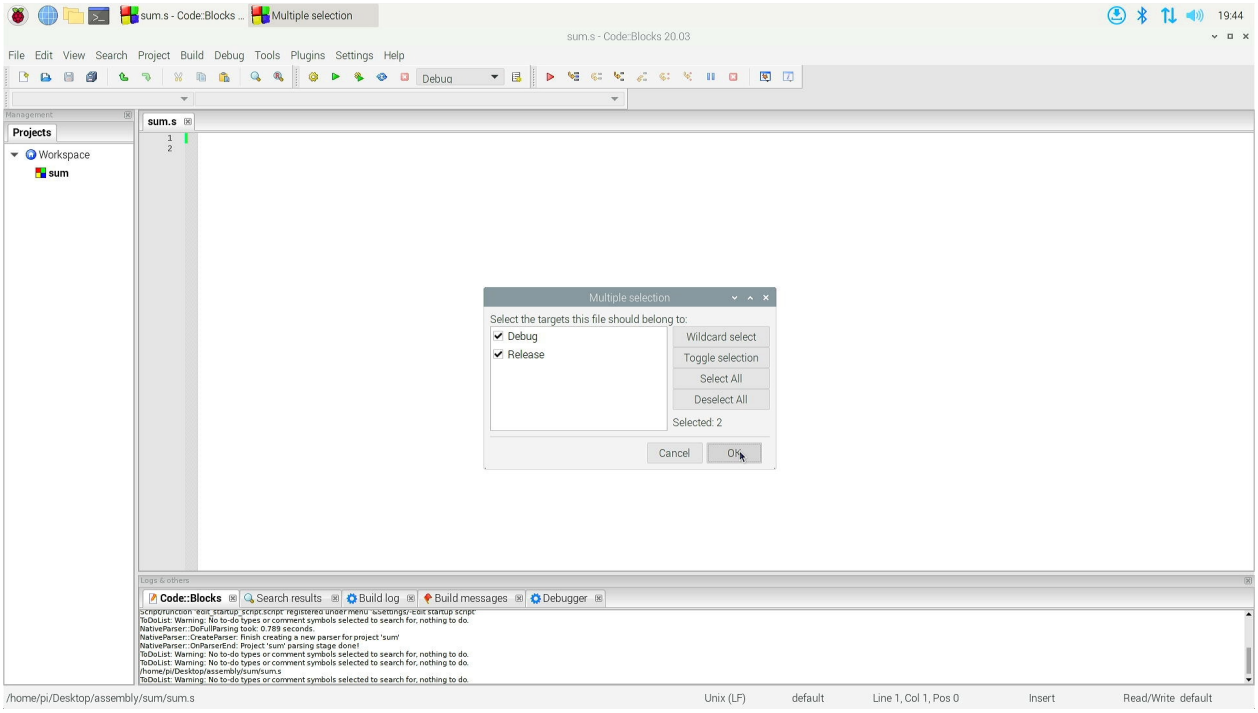

Figure 1.2.13 The *Multiple selection* window.

1. Click *OK*

1.2.13

In *sum.s* document, type out the following code. When done, save.

```
.global main
.func main

main:
  mov r1, #3
  mov r2, #4
  add r0, r1, r2
  bx lr
```

Listing 1.2.14 A simple introductory program.

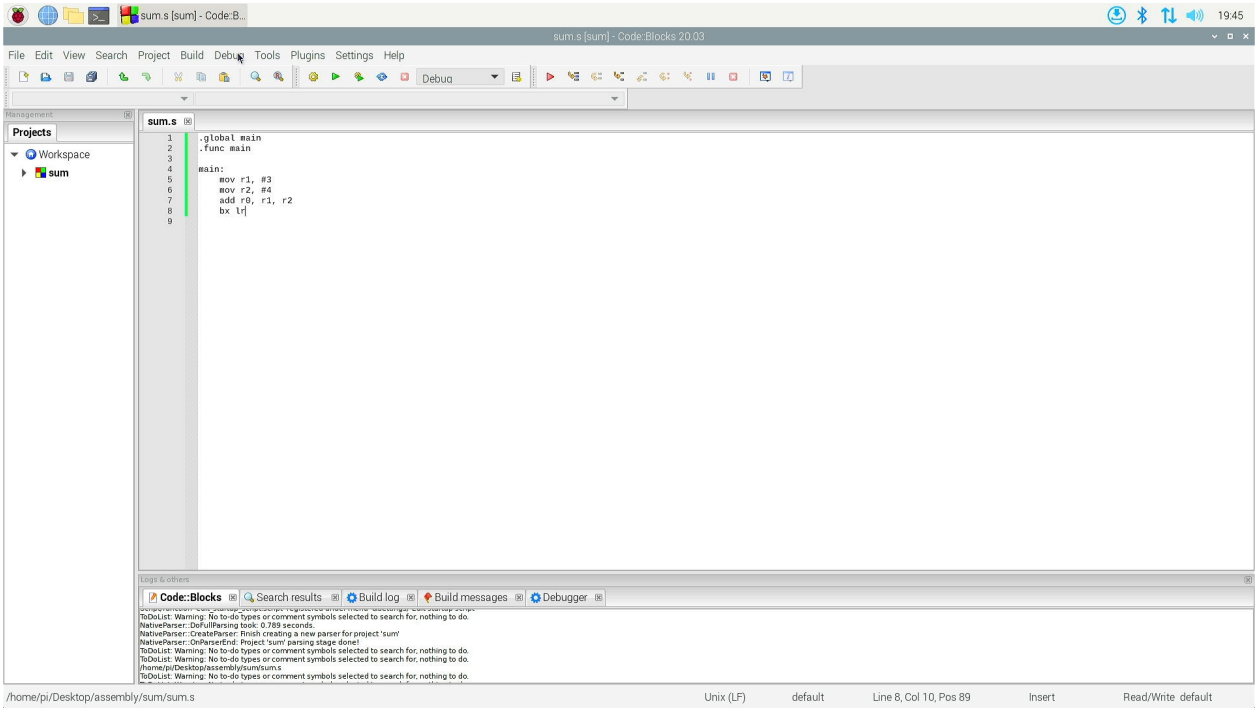

Figure 1.2.15 The *sum* program.

1.2.14

Now that our program is written, we can see it in action by:

1. Click *Build* in the menu bar.
2. ... *Run*

OR

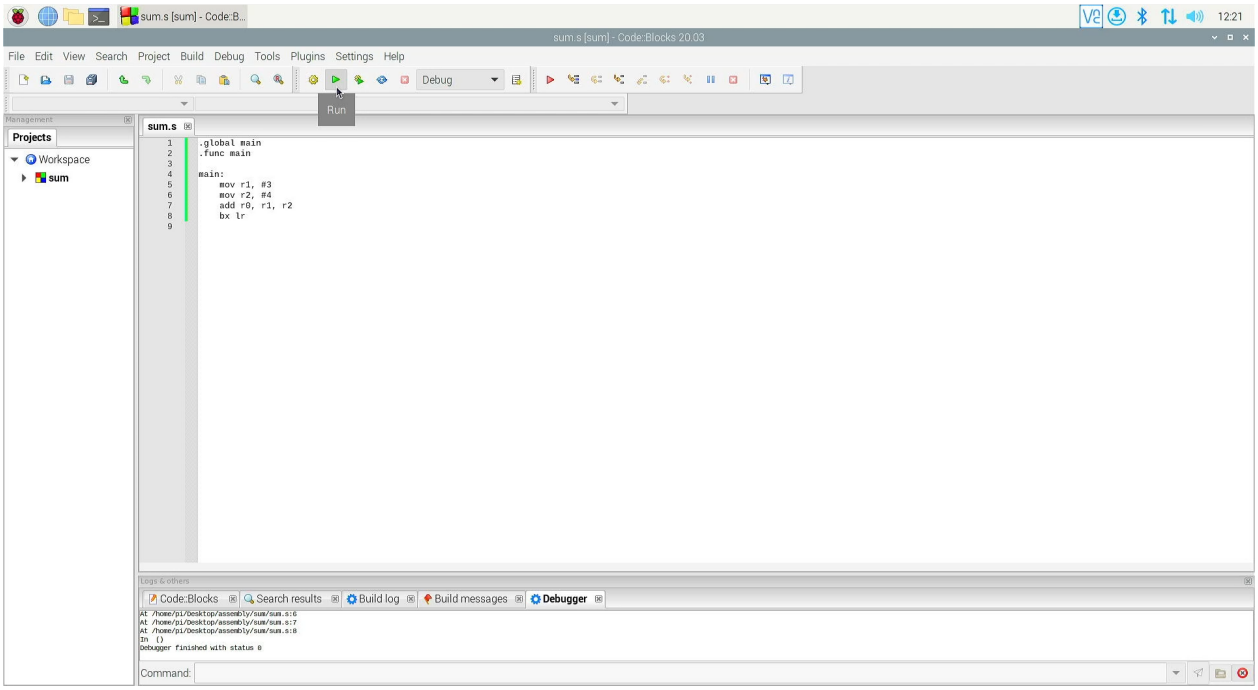

Figure 1.2.16 Select the active debugger for your application.

1. Click the *Run* (green arrow) shortcut in the toolbar.

If everything was typed correctly, you should see a little window pop up and present to you the result of running your program. However, if there are errors, you're going to need to double check your syntax and do some debugging.

Because the program functions by executing one step at a time, you can follow the progress of the application by adding *break points* to the program in the IDE. These function as stop-and-wait points in the program and enable you to take time and review expected states for things.

In our case, we are moving data into registers of the processor, and then adding up the values from two different registers into a third. As the program advances through the steps, we should be able to see the values of these registers change.

To see this debugging process in action, do the following:

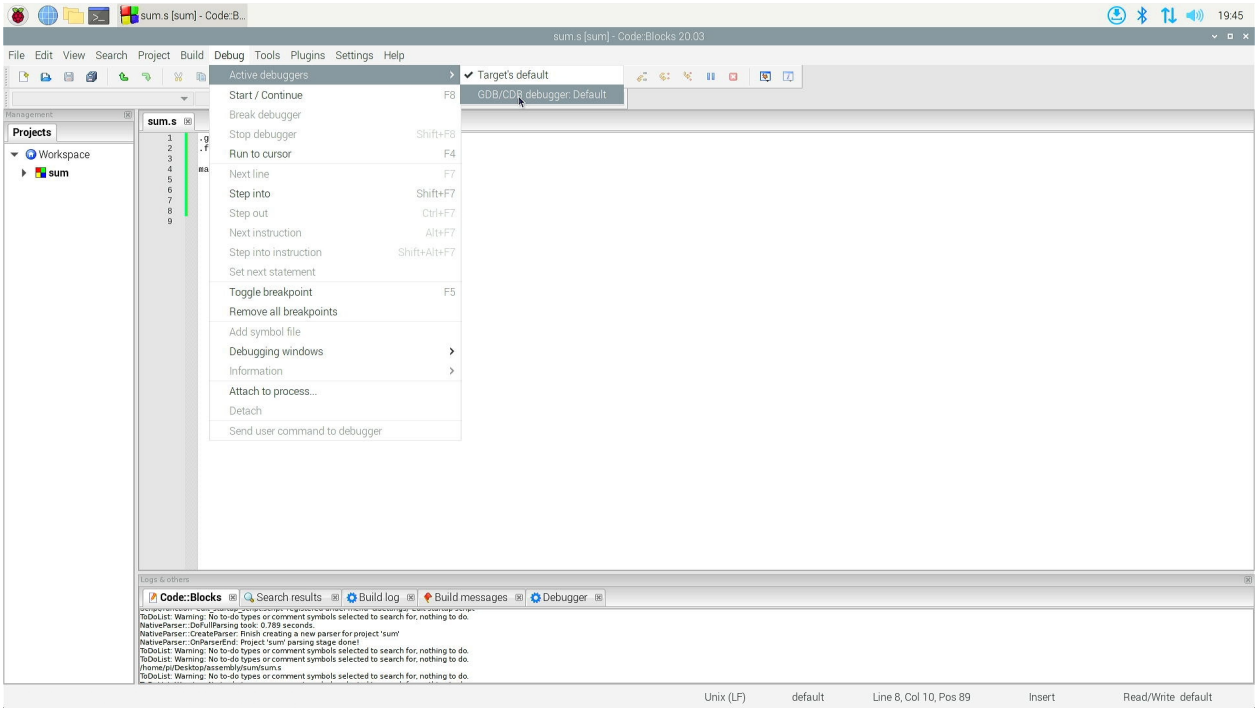

Figure 1.2.17 Select the active debugger for your application.

1. Click *Debug* in the menu bar.
2. ... *Active debuggers*
3. ... *GDB/CDB debugger: Default*

There are other debugging options, but this one works well for us now. In time, you may need a different one.

1.2.15

Add break points to the program.

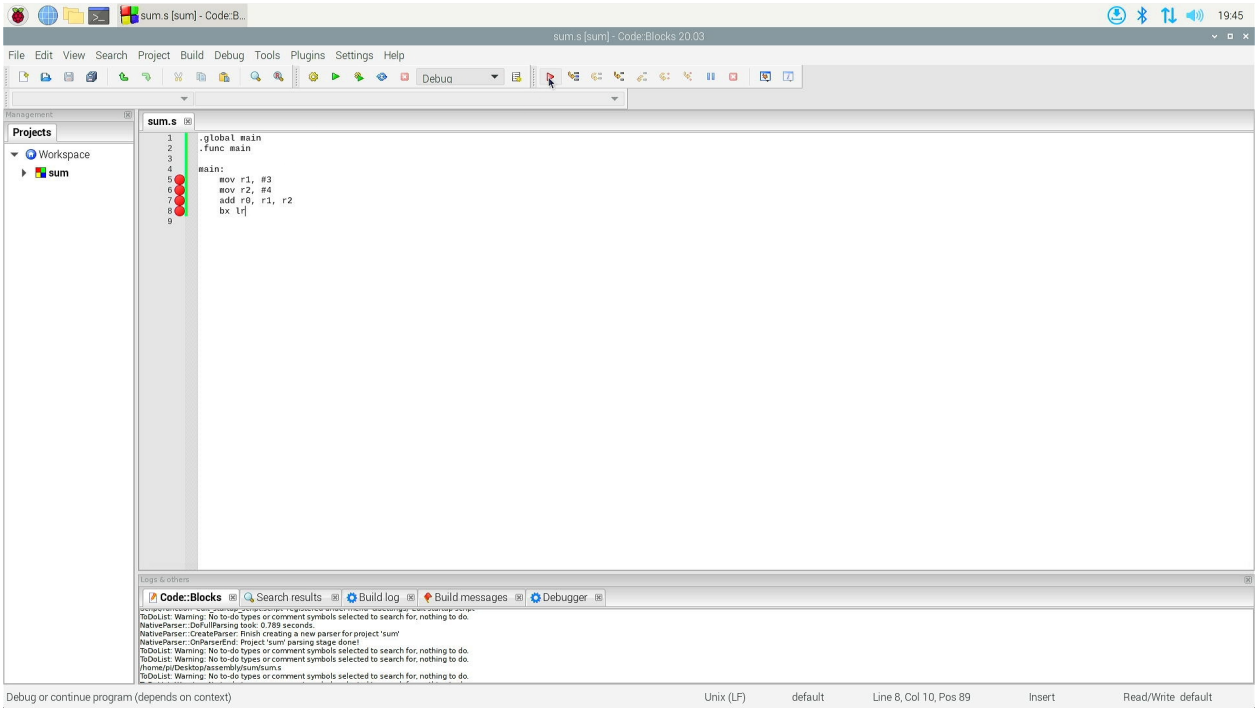

Figure 1.2.18 Adding break points to the program.

1. Click in the space to the right of the line number.

To remove a break point:

1. Click on the red dot that indicates a break point.

1.2.16

Run the program in debug mode.

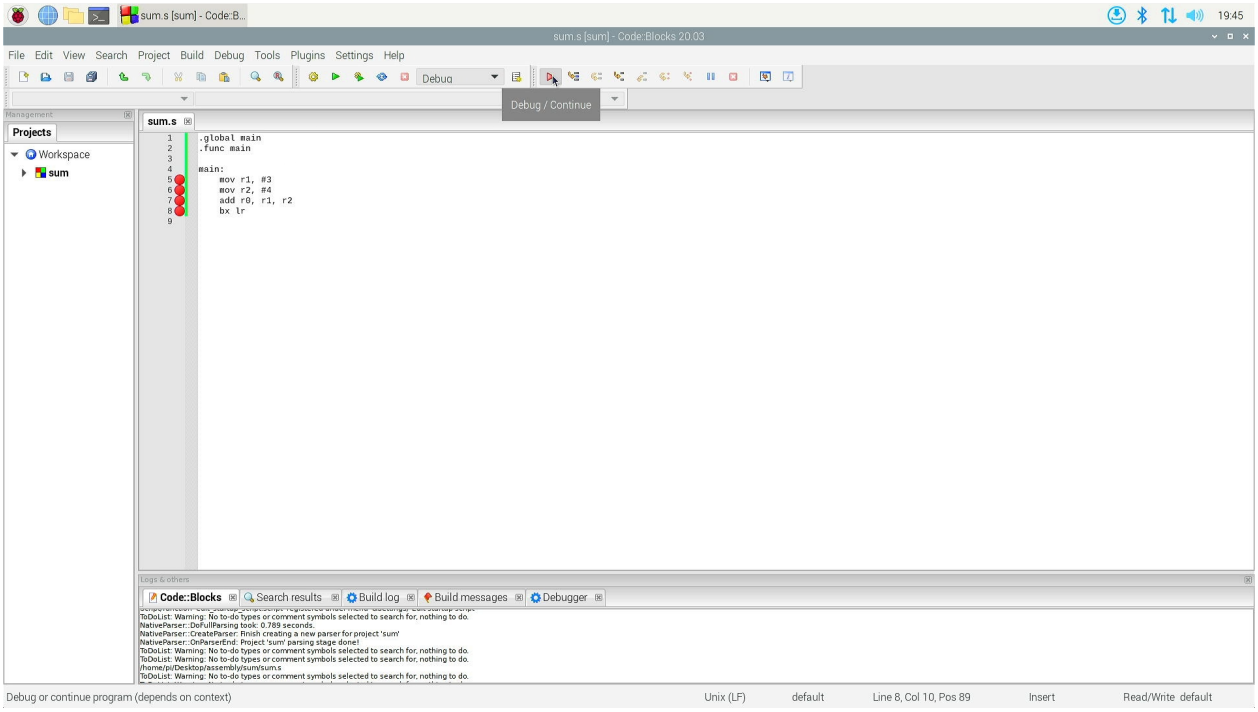

Figure 1.2.19 The *Debug / Continue* shortcut in the toolbar.

1. Click the *Debug / Continue* (red arrow) shortcut in the toolbar.

OR

1. Click *Debug*
2. ... *Start / Continue*

1.2.17

At this point, things will start happening on the screen.

- A program console will pop up on the screen, and maybe disappear.
- The layout will change slightly.

When asked to save the layout change:

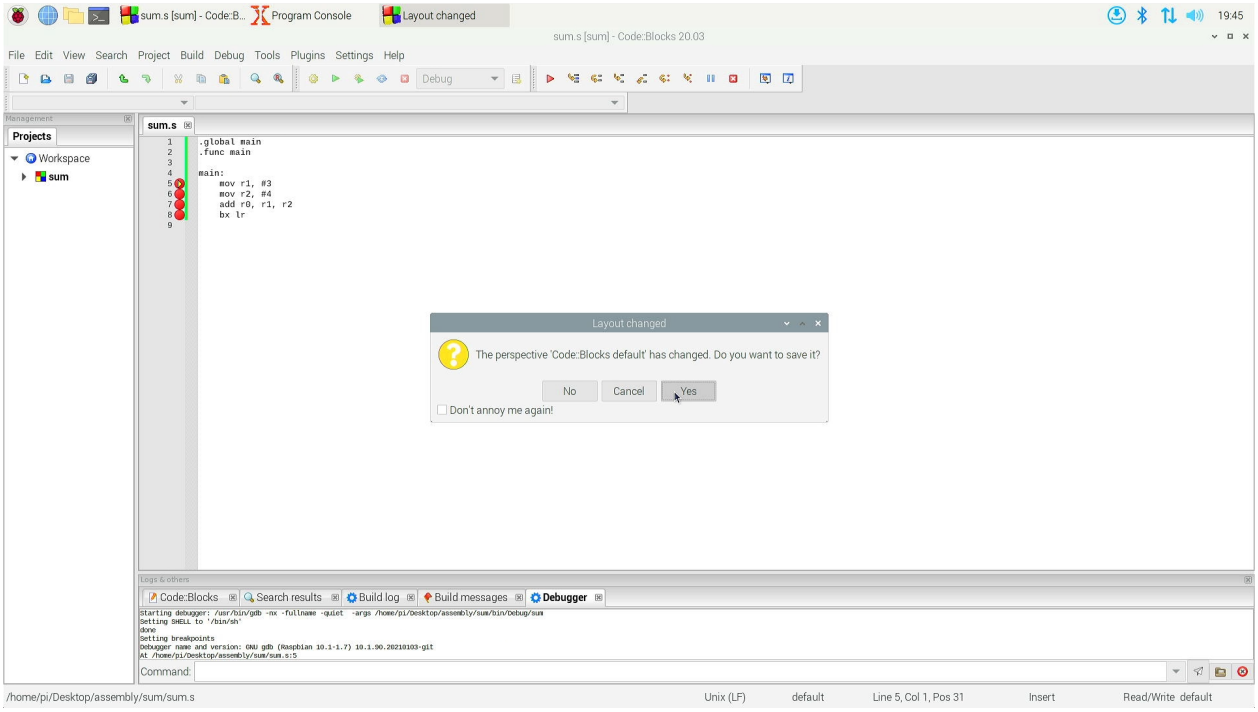

Figure 1.2.20 Notification that the layout has changed.

1. Click *Yes*

1.2.18

To see the register values as we advance through our application:

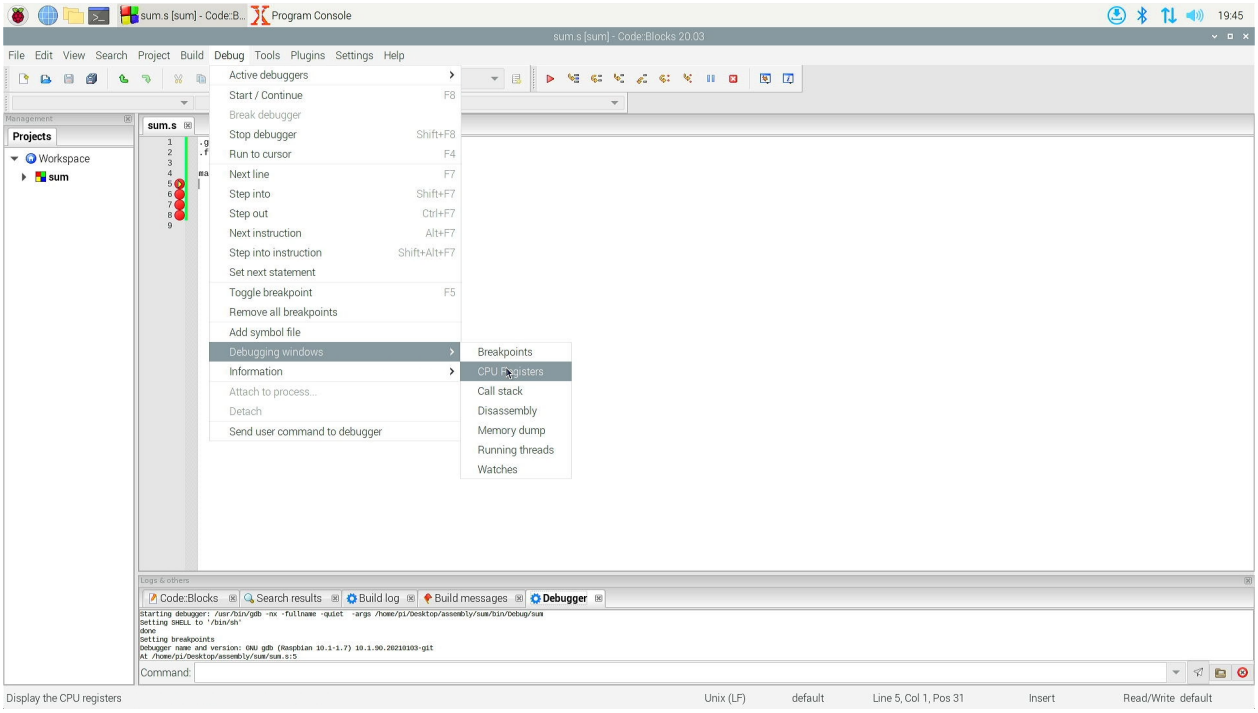

Figure 1.2.21 Navigation path for selecting *CPU Registers* from the *Debug* menu.

1. Click *Debug*
2. ... *Debugging windows*
3. ... *CPU Registers*

1.2.19

There should now be a window in the foreground that displays register names, and values in both hexadecimal, and decimal.

To see the values change as our program executes, step-by-step:

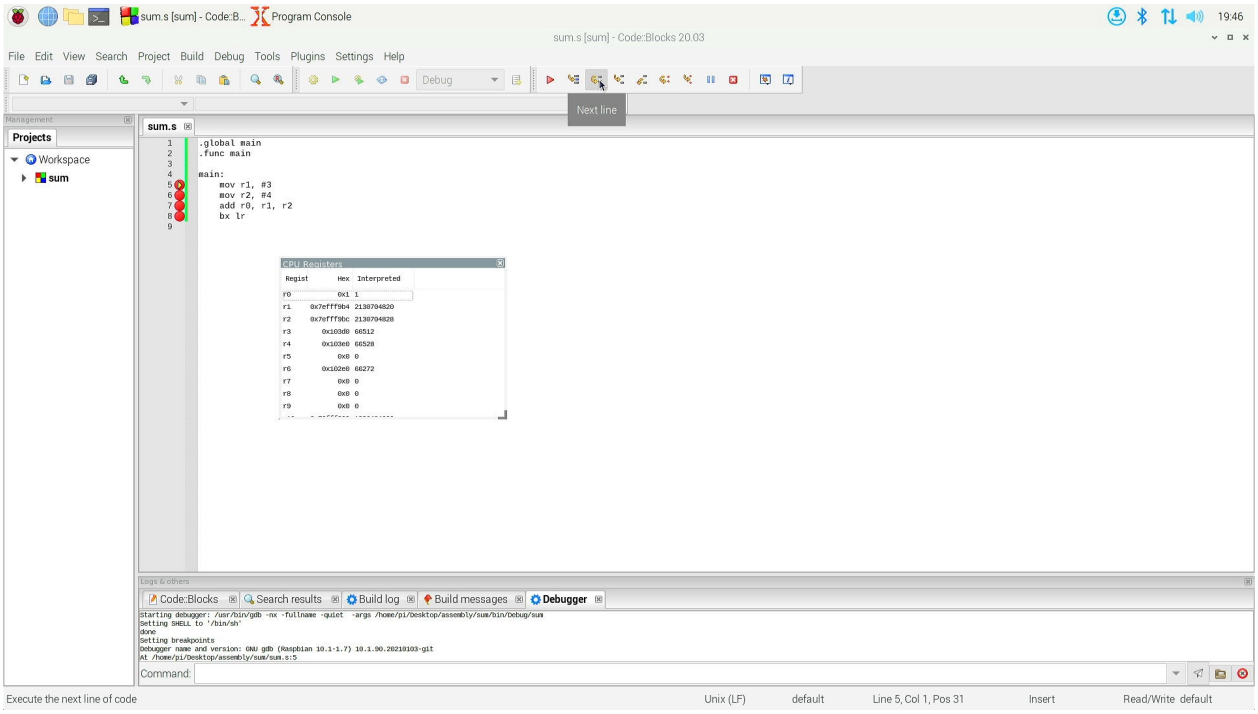

Figure 1.2.22

1. Click the *Next line* shortcut in the toolbar
2. View the register values
3. Repeat

As you advance through the program, you may notice a little yellow arrow overlaying the red break point symbols to indicate which one it is currently waiting at.

You may also notice messages in the *Debugger* logs in the bottom panel of the screen.

These can be useful references for later.

1.2.20

When you've stepped through the entire program, stop the debugger:

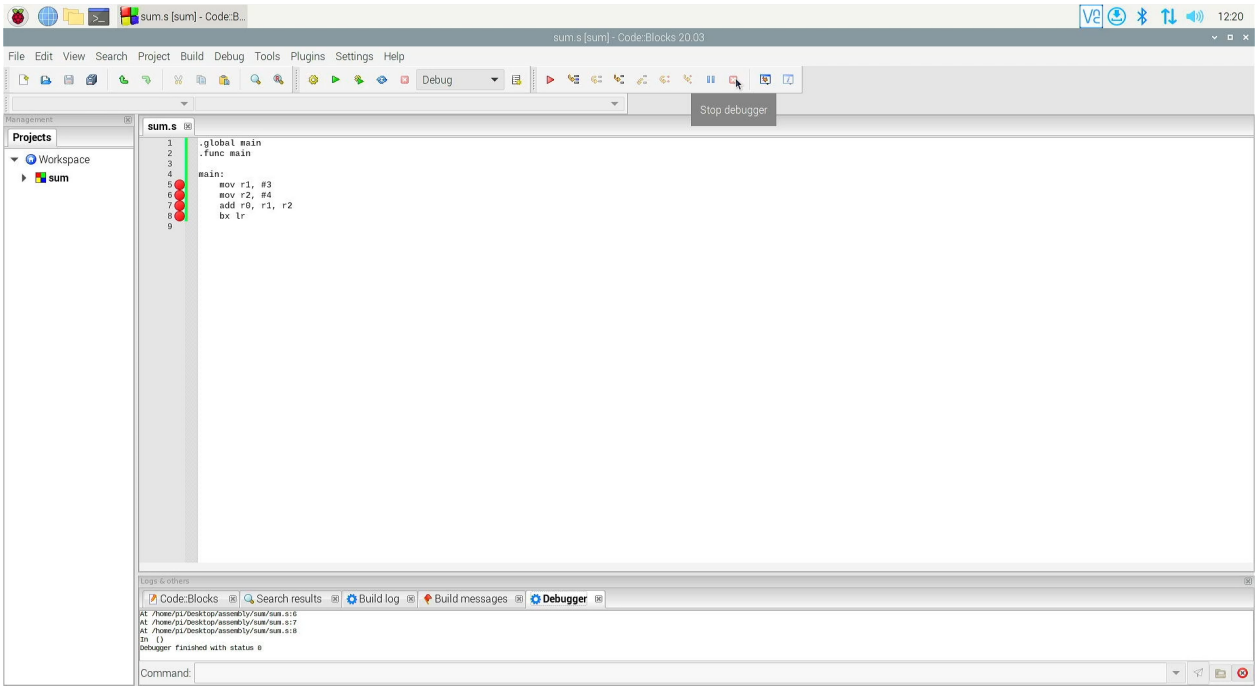

Figure 1.2.23

1. Click the *Stop debugger* shortcut in the toolbar.

Checkpoint 1.2.24 Reflections.

(a) What, if any, experience do you have with programming?

(b) Assuming you'll be shown what you need to type as you're taught the basics of the assembly programming language, how confident are you that you'll be able to write and successfully compile your programs?

(c) The Raspberry Pi uses an ARM processor. What are some other popular microprocessor designers?

(d) Is programming a concept that is limited to computers? Please explain.

(e) What aspects of your life require detailed istructions, precise communication?

Chapter 2

Basics of Programming

Programming, more than anything else, is about giving instructions. Programming is the source of control over microprocessors that are the core of embedded systems.

Before exploring the syntax of any programming language, we are going to explore the basic concepts that will be used to compose the code for a program.

We will explore the concepts of pseudo code, branching, looping, proceedures, and how these can be expressed in flowcharts, or comments to facilitate problem solving through programming.

Finally, we will explore some concepts like machine language, low-level programming languages, and high-level programming languages.

2.1 Where to begin?

We are going to begin this exploration with a really simple concept, and expand upon it to demonstrate it's importance when it comes to programming. We will start with this question:

How do you make a peanut butter and jelly sandwich?

A first draft of an answer to this question might go like:

1. Apply peanut butter to one slice of bread.
2. Apply jelly to a second slice of bread.
3. Lay second piece of bread on first.

It seems like a really simple question, and doesn't require a great deal of thought to answer. For many, in the United States, these instructions may be adequate enough. However, these instructions make very broad assumptions about the readers knowledge, and established ways of doing things. To provide useful instructions to a broader range of reader, you have to make fewer assumptions, and be more explicit with your instructions.

1. Acquire two pieces of bread with the following dimensions: 10cm wide, 10cm tall, and 1cm thick.
2. Apply 15g of peanut butter to one side of the first slice of bread.
3. Apply 7g of grape preserves to one side of the second slice of bread.
4. Lay the first slice of bread flat, with the peanut butter laying up.
5. Lay the second slice of bread on top of the first, with the jelly side down, so that the peanut butter and jelly are in contact.

With this second draft, the instructions are more explicit, but there are still many assumptions made. For example, does everyone know what peanut butter is? The answer is no. So, again, to provide useful instructions, you have to be still more explicit.

This process can seem tedious, but it is at the root of making computers do what you want them to do. Take a moment to reflect on just how difficult it would be to explain to a 2 year old the process of making a peanut butter and jelly sandwich. The explanation would require details for everything from the process of growing peanuts, making peanut butter, baking bread, and preparing jellies, to slicing bread, and the proper tools for applying the core ingredients. Now, take a second to reflect on just how difficult it is to explain the same process to a rock.

2.2 Defining the problem.

Now that the importance of being explicit is understood, we will move on to a topic that applies to programming, as well as any project you will work on. That is, defining the problem, and framing the solution.

The concept of defining the problem may seem silly, as silly as describing how to make a peanut butter and jelly sandwich, but it can honestly be quite tricky.

The problem may come from a client, that lacks the knowledge to properly explain what they need, or it may come from your own observations and desires. Whatever the source, it usually pays to step back, and study the problem before working on a solution.

Let us return to our peanut butter and jelly sandwich example, from a slightly different perspective for a moment.

Let's say a friend comes to you and says they are hungry. Our first assessment of the problem indicates that the real problem is a lack of food. Providing food will eliminate the hunger. We offer a solution of a PB&J. Our friend accepts that solution, and they eat the sandwich. As they make progress on the sandwich, they realize that the bread and the peanut butter are making it a bit difficult to eat the sandwich, and that they could really use something to wash it down.

While our solution of the sandwich was appropriate for the problem as it was originally outlined, in practice, we have learned that there were unintended consequences to our original solution that make it inadequate, so the solution has to be revised.

In our example, we revise the solution we offer our friend, by adding a glass of milk. Our friend enjoys a sip of milk occasionally as they eat the sandwich, and they are no longer hungry.

The point to this hypothetical scenario is that a problem often includes hidden, or obscured parameters that can be identified before you get started on a solution if you take a bit of time to study it. Even then, you may only be able to anticipate some of them once you've had experience with them before.

2.3 Framing the Solution.

Once you have a problem and solution defined in broad terms, it becomes necessary to start breaking the solution down into constituent steps. This first round of definition usually involves drawing connections between the parts of the solution, and the end product.

For example, when talking about making a PB&J, we'd be in a good position if we identified the components that go into making that sandwich, and organized them in a way that conveys relationships.

It can be convenient to start with a list of the elements involved, using levels to indicate subordinate points.

- Sandwich
    - Bread
        - Slice 1
            - Jelly
        - Slice 2
            - Peanut Butter

- Milk

Another option is a flowchart.

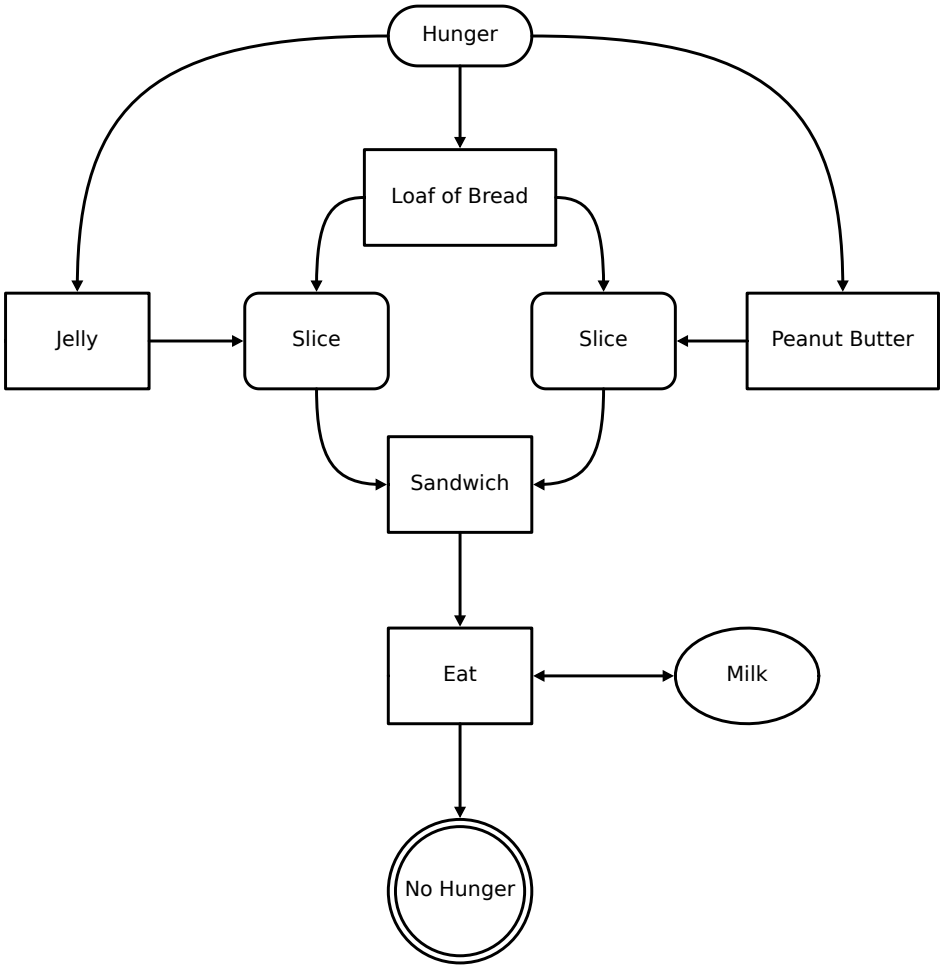

Figure 2.3.1 Illustration of solution for making a Peanut Butter and Jelly Sandwich.

Once the first round of definition is completed, it is very common for further definition to be necessary, resulting in one, or more rounds of increased granularity. Do any of your processes include loops, or decision branching?

For example, is it necessary to explain how jelly is made? Should we include a knife for cutting bread, or a plate to hold our sandwich? Should we include more definition of the Eat, Drink cycle in our solution?

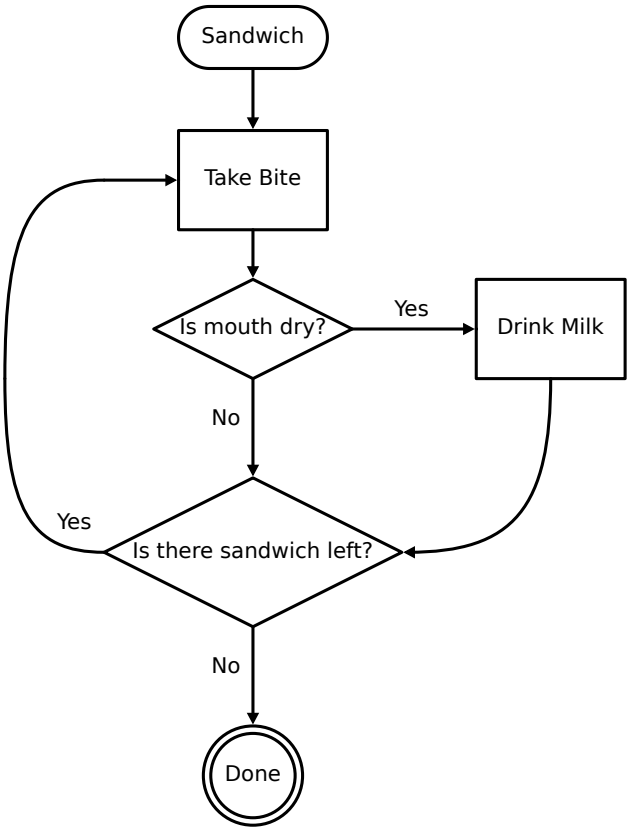

Figure 2.3.2 Illustration of the eat sandwich, drink milk cycle for our example.

Depending on the complexity of the project, what components are involved, etc. The documentation of the solution can become quite the production, but is typically even more valuable in the long run.

If you've never seen the repair manuals for old electrical equipment, I'd strongly recommend you take a look. The HP 9825[4] is just an example.

2.4 Documenting the process.

Lists are probably your best friend when it comes to planning but not in the obvious way. Lists really shine when you begin writing pseudocode. This is typically included as comments in a program, and is generally the starting point for new work.

These comments, lists of the steps that will be needed to make this section of code work are an excellent framework for programmers. First, because they are natural language descriptions that will remain in the code base, and will be readable for others who come later. Second, because they are syntax independent. You won't be able to remember how to type every command in the programming language, and looking that stuff up can be a distraction from the more important process of defining your objectives. Writing out the pseudocode list first, and adding the programming afterwards helps to keep you on track.

[4] `https://www.curiousmarc.com/computing/hp-9825-scientific-computer`

```
# Program Name: Make Peanut Butter and Jelly Sandwich

# import loaf
# slice_1 = get slice of bread with loaf
# slice_2 = get slice of bread with loaf

# import jelly
# call apply with (slice_1, jelly)

# import peanut butter
# call apply with (slice_2, peanut butter)

# sandwich = join (slice_1, slice_2)

# import milk

# call eat (sandwich, milk)


# function eat (sandwich, milk)
# A function to handle the sandwich and milk loop.
# while sandwich > 0
#   call bite on sandwich
#   if mouth == dry
#     call drink on milk

# function get slice of bread
# A function for getting a slice of bread from a loaf.
# loaf = loaf - slice
# return slice

# function apply (slice, topping)
# A function to handle the application of something to a slice of bread.
# slice = slice + topping
# return slice

# function join (slice_1, slice_2)
# A function for joining two slices of bread, with the expectation that they both
    hae a topping.
# return slice_1 + rotate(slice_2, 180)
```

Listing 2.4.1

Lists, in the form of pseudo code, have a limited scope of utility. Large programs can't easily be described with lists, because the relationships between sections of code, and the jumps they make to functions, or obscured bits of the program, aren't included in them. This is where the importance of flowcharts comes back to the forefront.

Flowcharts can be really helpful tools for visually orienting oneself with how a process will go. If the solution you're working on involves more than a couple components, it can be a worthwhile illustration to spend time on, as it can be used for your own reference as your work progresses, when discussing it with customers, and orienting new team members.

2.5 Writing Code

With documentation in place, as lists, flowcharts, and or pseudocode, it is time to start writing the actual program.

This set of exercises will introduce you to a couple of different programming paradigms. The first is

lower-level programming languages that covers very simple steps, and demands a great deal of knowledge of the underlying system. As a result, they don't get you very far very quickly, and thus, it is pretty difficult to go from idea to finished product. The second is high-level programming languages that have lots of inbuilt functionality, don't require a great deal of familiarity with the underlying system, and enables you to program a solution quickly. However, as a result, they do abstract away a lot of the underlying principles of the machine.

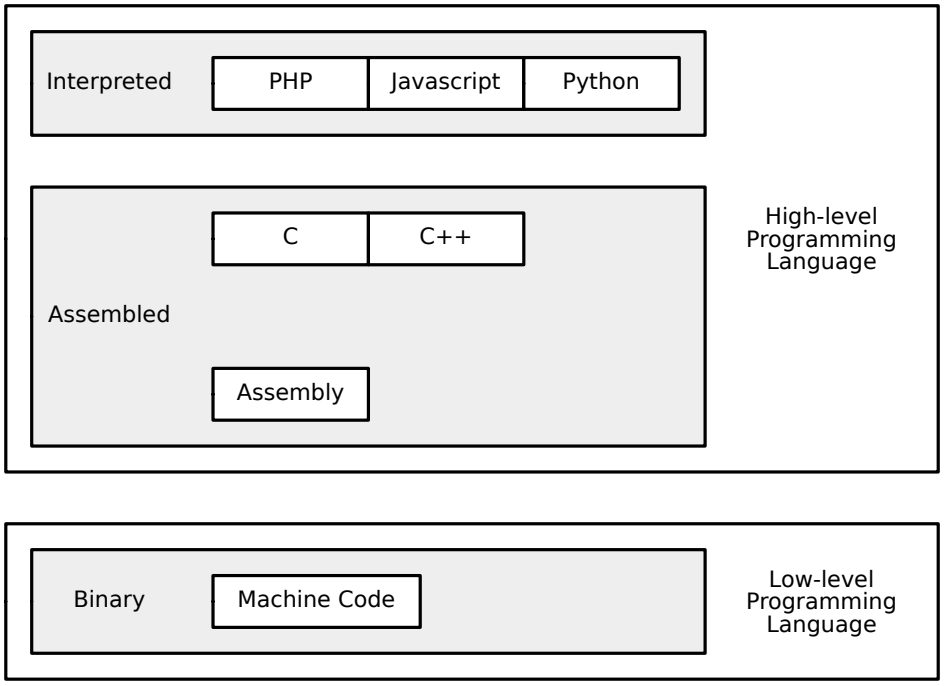

Figure 2.5.1 Illustration of the basic relationship between different programming languages.

The first programming language we will be exploring is called Assembly. Assembly is just one layer above machine code. It requires mind-numbing levels of detail to accomplish anything of great significance on a modern computer. However, it can be everything you need for smaller embedded microcontrollers in the right environment. A microwave isn't a complex electronic circuit. Assembly on a tiny 8-pin microcontroller interfacing with a few buttons, and controlling operation would be perfectly reasonable.

Once you start getting into meaningful human interactions, interfacing with data, or peripheral systems, a language like C, or C++ is much more approachable. While the level of detail for these languages is still pretty high, it includes a lot of abstraction in the base libraries. Processes like getting user input, or providing output are made significantly easier.

Finally, there are a whole slew of scripting, or interpreted languages that abstract away a lot of the detail from programming. In doing so, you loose some control, but for most general purpose applications, that control isn't needed. In exchange, they make processes like getting data from the internet, or displaying images on a screen a breeze.

Checkpoint 2.5.2 Reflections.

(a) Reflect on your daily routine. Pick one regular process that you'd like to automate to the point that you could completely forget about it. What is that process? What involved in that process, and what is the result?

(b) Using the process you've identified, write, in pseudocode, the series of steps and decisions that would be needed to automate it.

(c) Using the pseudocode you've created, draft a flow chart. Include inputs, decisions/branching, loops, etc.

(d) By your own estimate, how many other people would share your desire to automate this process? What

are some of the challenges to actually doing so?

Chapter 3

Basics of Assembly

Machine code is the actual data that a microprocessor uses to do the work you give it. However, machine code isn't human readable. Assembly was an early step in allowing humans to more easily provide instructions to the computer. It is translated at compile time, into machine code which is then executed by the microprocessor.

The approach to programming in assembly will change fairly significantly based upon the processor that you're coding for. The instruction set will alter the flow of many basic operations, and programs written for one processor will not work on a different one.

We will explore the basics of the ARMV7 Instruction Set.

While the syntax is different than that of the Atmel, and Intel microprocessors, the basic concepts are the same. These are instructions designed to interface with highly integrated digital circuits.

The basics will include registers, and memory addressing as well as arithmetic and logical data processing instructions.

3.1 Associated Reading

- Raspberry Pi Assembler[5]
  - Chapter 2: ARM Registers
  - Chapter 3: Memory
- ARMv7-A (32-bit) Programmer's Guide[6]
  - Chapter 5: ARM/Thumb Unified Assembly Language Instructions
- ARMv8-A (64-bit) Programmer's Guide[7]
  - Chapter 6: The A64 instruction set

3.2 Basic Operations and Composition

3.2.1 Dicsussion

As you will have learned through the reading, assembly is oriented around very basic operations. Moving data into a register, out of a register, performing some sort of calculation on the value of one or more registers, etc.

[5] https://personal.utdallas.edu/~pervin/RPiA/RPiA.pdf

[6] https://developer.arm.com/documentation/den0013/d/ARM-Thumb-Unified-Assembly-Language-Instructions?lang=en

[7] https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set

3.2.2 Code

Project 1 Transcribe, compile, and run the program.
Download the Source[8]

```
0|  .global main
1|  .func main
2|
3|  main:
4|    mov r0, #5
5|    mov r1, #5
6|    add r0, r0, r1
7|    bx lr
```

Listing 3.2.1

3.2.3 Exploration

Checkpoint 3.2.2 What is the return value of the program?

Answer. 10

The first program is very simple. It loads two values into register 0, and 1, and then adds the two values placing the sum into register 0. We will build off of this as we progress through the experiment.

Once you've got it running, proceed to the next section.

3.3 Memory Addressing

3.3.1 Discussion

The values that are moved into the registers `r0` and `r1` in the first example are immediate values, or values that are defined in the program code. More often than not, the values you'll be working with won't be defined in the code, but will instead be stored in memory, and will need to be retrieved before performing operations on them.

3.3.2 Code

Project 2 Transcribe, compile, and run the program.
Download the Source[9]

[8] `https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_basics/p1.s`

```
 0|  .data
 1|  .balign 4
 2|  numbers: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
 3|
 4|  .text
 5|  .balign 4
 6|  .global main
 7|  .func main
 8|
 9|  main:
10|    mov r0, #5
11|    mov r1, #5
12|      /* r0 + 5 */
13|    add r0, r0, r1
14|
15|    ldr r2, numbers_address
16|    ldr r1, [r2], #4
17|      /* r0 + 10 */
18|    add r0, r0, r1
19|    ldr r1, [r2], #4
20|      /* r0 + 9 */
21|    add r0, r0, r1
22|    ldr r1, [r2], #4
23|      /* r0 + 8 */
24|    add r0, r0, r1
25|
26|    bx lr
27|
28|  numbers_address: .word numbers
```

Listing 3.3.1

3.3.3 Exploration

Checkpoint 3.3.2 What is the return value of the program?

Answer. 37

This example addresses accessing values from the memory. In doing so, we first initialize an array, or list of numbers that we will use for our calculations. When doing so, we specify the byte length for each element (number) which is `.word` or 4 bytes each. We also provide an assembler directive of `.balign 4` which tells the assembler that our data should be arranged with addresses that are multiples of 4 bytes. You can read more about assembler directives[10] if you're interested.

While `numbers` identifies our variable for later reference, we can't use that to access the memory address directly, so we add the line `numbers_address: .word numbers` at the bottom that provides another variable that we can use. The explanation for this is found in the book.[11]

Finally, we start accessing our numbers from memory with the `ldr r2, numbers_address` command. This places the address of our first element. It is important to understand that even though our first element is 10, `r2` only contains the memory address to it, but not the actual value `10`.

With the next line `ldr r1, [r2], #4` we copy the value that is stored at the memory address stored in `r2`, and then change the memory address by increasing its value by 4.

Once our new value is copied into `r1` we can add it to the running total in `r0`.

[9] `https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_basics/p2.s`

[10] `http://web.mit.edu/gnu/doc/html/as_7.html`

[11] Raspberry Pi Assembly, page 12

Project 3 The sample addition of our list of numbers stops at 8, add the remaining lines of code so that all of the numbers of our list are added to our running total in `r0`.

Download the Source[12]

Checkpoint 3.3.3 What is the output of the program once all the numbers are added?

Answer. 65

3.4 Arithmetic Functions

3.4.1 Discussion

So far, we've made great use of the addition instruction. But, as you know, there are more arithmetic operations.

3.4.2 Code

Project 4 Add the following lines before `bx lr` in the previous example to calculate the average value for the numbers we've added up.

```
1|      /* Find the average */
2|    mov r1, #12
3|    udiv r0, r0, r1
```

Listing 3.4.1

Your final code should look something like this:

Download the Source[13]

[12] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_basics/p3.s

```
 0|  .data
 1|  .balign 4
 2|  numbers: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
 3|
 4|  .text
 5|  .balign 4
 6|  .global main
 7|  .func main
 8|
 9|  main:
10|    mov r0, #5
11|    mov r1, #5
12|      /* r0 + 5 */
13|    add r0, r0, r1
14|
15|    ldr r2, numbers_address
16|    ldr r1, [r2], #4
17|      /* r0 + 10 */
18|    add r0, r0, r1
19|    ldr r1, [r2], #4
20|      /* r0 + 9 */
21|    add r0, r0, r1
22|    ldr r1, [r2], #4
23|      /* r0 + 8 */
24|    add r0, r0, r1
25|    ldr r1, [r2], #4
26|      /* r0 + 7 */
27|    add r0, r0, r1
28|    ldr r1, [r2], #4
29|      /* r0 + 6 */
30|    add r0, r0, r1
31|    ldr r1, [r2], #4
32|      /* r0 + 5 */
33|    add r0, r0, r1
34|    ldr r1, [r2], #4
35|      /* r0 + 4 */
36|    add r0, r0, r1
37|    ldr r1, [r2], #4
38|      /* r0 + 3 */
39|    add r0, r0, r1
40|    ldr r1, [r2], #4
41|      /* r0 + 2 */
42|    add r0, r0, r1
43|    ldr r1, [r2], #4
44|      /* r0 + 1 */
45|    add r0, r0, r1
46|
47|      /* Find the average */
48|    mov r1, #12
49|    udiv r0, r0, r1
50|
51|    bx lr
52|
53|  numbers_address: .word numbers
```

Listing 3.4.2

Compile, and run the program.

[13] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_

3.4.3 Exploration

Checkpoint 3.4.3 What is the return value of the program?

Answer. 5

Checkpoint 3.4.4 Do the same calculations with a calculator, does the return value of the program match your own calculations?

Answer. No, the program returns 5, and the real answer is $5.41\overline{6}$.

Solution. $\frac{5+5+10+9+8+7+6+5+4+3+2+1}{12} = \frac{65}{12} = 5.41\overline{6}$

This addition to our program makes use of an unsigned division instruction, but as we can see, the returned value from our program doesn't include any decimal values. `r0` only contains the whole number portion of the quotient. When doing multiplication or division operations, there are special registers, and special rules, or tricks to make the process easier.

We won't get too far into the weeds with how this works, but if you want to know more you can read more about ARM assembly division on the Raspberry Pi[14] or ARM assembly floating point registers[15].

With assembly, and some other high-level programming languages, it is very important to understand that there are signed, and unsigned ways of encoding the numbers you're working with. When working with unsigned values, you can add, and subtract, but you can't go into negative numbers. Additionally, there are often specific instructions to use when working with unsigned vs signed numbers.

Checkpoint 3.4.5 What is the return value for the program if we remove the division instruction, and replace the last 10 `add` instructions with `sub` instructions? Download the Source[16]

Answer. 211

Checkpoint 3.4.6 What is happening to `r0` that produces the value returned by this modification to the program?

Answer. We are experiencing a register overflow.

Solution. The maximum value of the register is apparently 255, or the maximum value for an 8-bit number.
$5 + 5 - 10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 = -45$
$256 - 45 = 211$

3.5 Logical Operations

3.5.1 Discussion

For our last subject, we will explore a bit of logical operations.

These are operations like shifting, not, or, and and xor. These operations aren't as relatable as the arithmetic, or the `mov` commands you've explored so far, but they do have very important places in the computer world. Otherwise, they wouldn't have been implemented.

3.5.2 Code

Project 5 Add the following line before the `bx lr` line. Then compile and run the program.

```
1|      /* Clear register 0 */
2|    eor r0, r0, r0
```

Listing 3.5.1

Your code should look like this:

basics/p4.s

[14] https://thinkingeek.com/2013/08/11/arm-assembler-raspberry-pi-chapter-15/

[15] https://eclecticlight.co/2021/07/23/code-in-arm-assembly-floating-point-registers-and-conversions/

[16] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_basics/p4.s

Download the Source[17]

```
 0|  .data
 1|  .balign 4
 2|  numbers: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
 3|
 4|  .text
 5|  .balign 4
 6|  .global main
 7|  .func main
 8|
 9|  main:
10|    mov r0, #5
11|    mov r1, #5
12|      /* r0 + 5 */
13|    add r0, r0, r1
14|
15|    ldr r2, numbers_address
16|    ldr r1, [r2], #4
17|      /* r0 + 10 */
18|    sub r0, r0, r1
19|    ldr r1, [r2], #4
20|      /* r0 + 9 */
21|    sub r0, r0, r1
22|    ldr r1, [r2], #4
23|      /* r0 + 8 */
24|    sub r0, r0, r1
25|    ldr r1, [r2], #4
26|      /* r0 + 7 */
27|    sub r0, r0, r1
28|    ldr r1, [r2], #4
29|      /* r0 + 6 */
30|    sub r0, r0, r1
31|    ldr r1, [r2], #4
32|      /* r0 + 5 */
33|    sub r0, r0, r1
34|    ldr r1, [r2], #4
35|      /* r0 + 4 */
36|    sub r0, r0, r1
37|    ldr r1, [r2], #4
38|      /* r0 + 3 */
39|    sub r0, r0, r1
40|    ldr r1, [r2], #4
41|      /* r0 + 2 */
42|    sub r0, r0, r1
43|    ldr r1, [r2], #4
44|      /* r0 + 1 */
45|    sub r0, r0, r1
46|
47|      /* Clear register 0 */
48|    eor r0, r0, r0
49|
50|    bx lr
51|
52|  numbers_address: .word numbers
```

Listing 3.5.2

Compile, and run the program.

[17] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_

3.5.3 Exploration

Checkpoint 3.5.3 What is the return value of the program?

Answer. 0

This exclusive or `eor` takes the value that is in `r0` and xors it with the same value to produce an empty register, or a register with a value of 0.

If you'd like to learn more about the logical operators, check out section *A4.4.1 Standard data-processing instructions* from the ARM Architecture Reference Manual[18]

3.6 C

3.6.1 Discussion

When dealing with embedded systems, even when using high-level programming languages, it is likely that you're going to have to make use of some of this low level manipulation. It may be a situation where you need to take advantage of some feature in a chip that isn't already abstracted by a library, or it could be when communicating with some peripheral. Whatever the case, having at least a basic knowledge of some of this is very useful.

However, it is a lot easier to get many of these basic functions accomplished with a high-level programming language. A language like C is one where you're still very intimate with the size, and signing of variables, but there are also abstractions for many of the most common operations that make using it much more desirable.

Just as a quick example, I offer you this little program that adds up several values, and displays the output.

3.6.2 Code

Project 6 If you care to, you can type this up in a filename `extra.c`, then compile and run it.

Download the Source[19]

```
 0|  #include <stdio.h>
 1|
 2|  int main()
 3|  {
 4|   int numbers[] = {5, 5, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
 5|   int n = sizeof(numbers) / sizeof(numbers[0]);
 6|
 7|   float sum = 0;
 8|   for (int i = 0; i < n; i++)
 9|     sum += numbers[i];
10|
11|   float avg = sum / n;
12|
13|   printf("The total of the added numbers is: %f\n", sum);
14|   printf("The average value of the numbers added up is: %f\n", avg);
15|  }
```

Listing 3.6.1

The program can be compiled with the following command typed in a terminal:

```
gcc -o extra extra.c
```

Listing 3.6.2

basics/p6.s

[18] https://developer.arm.com/documentation/ddi0406/cd/?lang=en

And can then be run by typing the following command in a terminal:

```
./extra
```

Listing 3.6.3

3.6.3 Exploration

You can see that with a high level programming language like C, you can accomplish the same result, but with better output. In this case, we print out, onto the `stdout` a whole sentence that includes a decimal value of the calculations we made.

We will explore more of high-level programming languages in later experiments.

Checkpoint 3.6.4 Reflections.

(a) Please provide the assembly instructions needed to add 3 numbers with the total available in register 0

(b) What assembly instruction would be used to load a value from memory, and increment the pointer address by 1 byte?

(c) At this early stage in your assembly exposure, what are your thoughts on programming a computer in this way?

[19] `https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/assembly_basics/extra.c`

Chapter 4

Assembly Control Structures

Most applications for microprocessors will not be met by the basics of assembly covered so far. A program that can only address memory and do some math won't have many use cases.

To improve upon what we've learned, we will explore control structures in the form of branching, and loops. Additionally, we will explore some additional data processing instructions in the form of shift operations.

4.1 Associated Reading

- Raspberry Pi Assembler[20]
  - Chapter 5: Branching
  - Chapter 6: Control Structures

4.2 Loops

As you'll have learned from the reading branching, and control structures are closely related. The reality is that looping, as well as if, then, and else statements are entirely dependent upon decisions made based upon evaluations of values in our input.

4.2.1 Discussion

Loops are incredibly useful. Mostly because they allow us to re-use portions of code so that we aren't typing out millions of lines of instructions to handle the same process over and over again, but also because they can seriously simplify processes that would require repeated decision making, thus simplifying our code, and making it more human readable.

There are a couple of different approaches to loops, which you use may depend on the application.

The first, which will be the subject of our first project in this experiment, uses an index, and decrements that index as you iterate over the elements of the array. Once the index reaches a specific value, the loop exits. This method is generally most useful when you know exactly how much data you're working with. Either through a calculation, or through some predefined method.

The second looks for a specific value in the array, and when it is found, exits the loop. This approach is generally used for streams of data, or data with an undefined length.

[20] https://personal.utdallas.edu/~pervin/RPiA/RPiA.pdf

4.2.2 Code

Project 7 Transcribe, compile, and run the program.
Download the Source[21]

```
 1|  .data
 2|  .balign 4
 3|  numbers: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
 4|  .balign 4
 5|  number_count: .word 10
 6|
 7|  .text
 8|  .balign 4
 9|  .global main
10|  .func main
11|
12|  main:
13|    mov r0, #5
14|    mov r1, #5
15|      /* r0 + 5 */
16|    add r0, r0, r1
17|
18|      /* Create an itteration counter that starts with the number count */
19|    ldr r3, =number_count
20|    ldr r3, [r3]
21|
22|    ldr r2, =numbers
23|  add_more:
24|    ldr r1, [r2], #4
25|    add r0, r0, r1
26|      /* Decriment our itteration counter. */
27|    sub r3, r3, #1
28|      /* Is our itteration counter equal to 0? */
29|    cmp r3, #0
30|      /* If yes/true, jump to the end of the program. */
31|    beq end
32|
33|      /* If no/false, do it again. */
34|    b add_more
35|  end:
36|
37|    bx lr
```

Listing 4.2.1

4.2.3 Exploration

In this example, we define an index value `number_count` on line 5 and load it into register `r3` on lines 19 and 20. Then, as we load, and add each element of the `numbers` array, we decrement `r3` by 1 on line 27. Next, we compare the value in `r3` to 0 to set flags. The `beq` command on line 31 will check the flags to see if `r3` and 0 are equal, if so, it will jump to the `end:` label. If not, it will advance to the `b add_more` instruction on line 34, which sends our program back to the `add_more:` label on line 23.

[21] `https://github.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/raw/master/assembly_control_structures/p1.s`

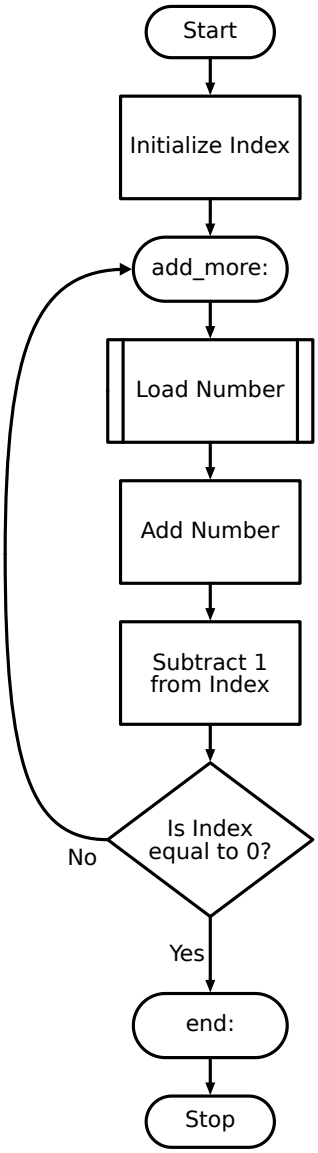

Figure 4.2.2 The logical flow of the program.

Checkpoint 4.2.3 What is the value returned by the program?

Answer. 65

Checkpoint 4.2.4 What would happen if the index were initialized at 11, or 0 instead of 10?

Answer. ?

4.2.4 Discussion

Our next program illustrates the process of looking for a special value in an array, and exiting the loop when it is found.

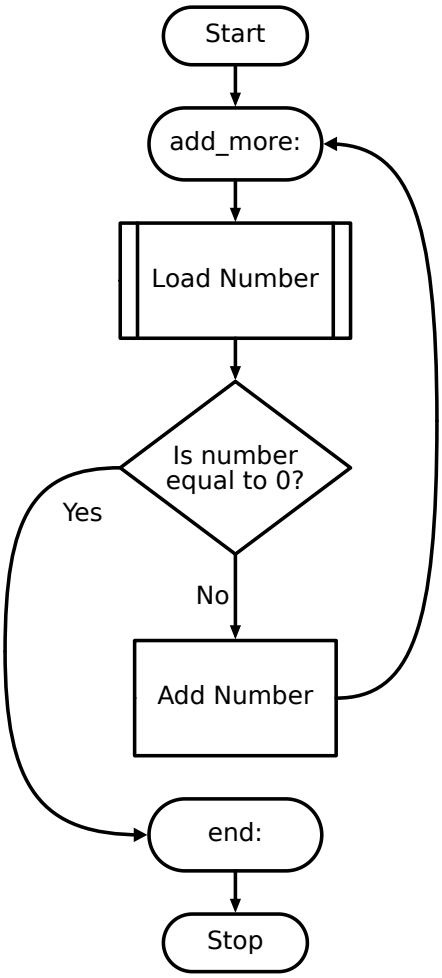

Figure 4.2.5 The logical flow of the program.

4.2.5 Code

Project 8 Transcribe, compile, and run the program.
Download the Source[22]

```
 1|  .data
 2|  .balign 4
 3|  numbers: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
 4|
 5|  .text
 6|  .balign 4
 7|  .global main
 8|  .func main
 9|
10|  main:
11|    mov r0, #5
12|    mov r1, #5
13|      /* r0 + 5 */
14|    add r0, r0, r1
15|
16|    ldr r2, =numbers
17|  add_more:
18|    ldr r1, [r2], #4
19|      /* Is the new number equal to 0? */
20|    cmp r1, #0
21|    beq end
22|      /* Add new number. */
23|    add r0, r0, r1
24|      /* Do it again. */
25|    b add_more
26|  end:
27|
28|    bx lr
```

Listing 4.2.6

4.2.6 Exploration

In this example, we simply add a special character (0) at the end of our numbers array, and as we load each number, we check to see if the newly loaded value is equal to our special character. On line 20, we compare `r1` and 0 to set flags. Then on line 21 the `beq end` instruction checks the flags, and if `r1` is equal to our special character, we jump to the `end:` label on line 26. Otherwise, the program advances, adds the new value to our running total in `r0` and then reaches `b add_more` on line 25 which sends the program back to the `add_more` label on line 17.

4.3 Branching

4.3.1 Discussion

As we continue with our exploration of control structures in assembly, we are going to shift our frame of reference a bit, and use a different example.

Let us imagine that we have a project that needs to control the temperature of an enclosure. We have a temperature sensor, two relays, and an alarm buzzer attached to a microcontroller.

One of the relays enables power for a heating element. The other enables power for an air conditioning unit.

Our objective is to write a program for our little micro controller that takes a temperature reading from the sensor, and turns on the heat, the air conditioner, or does nothing, depending on defined conditions.

[22] https://github.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/raw/master/assembly_control_structures/p2.s

Our design takes into consideration that something may go wrong with our temperature control equipment, and so it is important that we include functionality that will alert us in the event that there is some sort of failure.

Our failure condition is: if we go to enable either the heater, or air conditioner, and it is already enabled. In either case, we may have a broken sensor, a broken temperature control circuit, or a compromised enclosure.

The logic looks a bit like this:

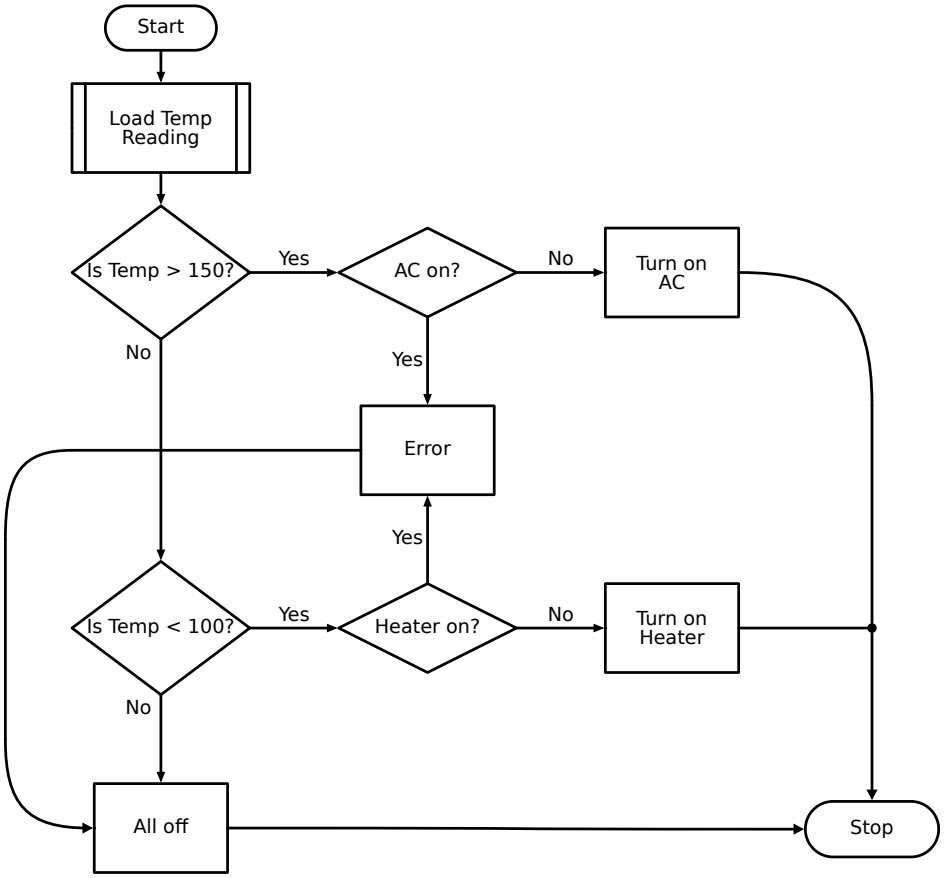

Figure 4.3.1 The logical flow of the program.

4.3.2 Code

Project 9 Transcribe, compile, and run the program.

Download the Source[23]

```
 1|  .data
 2|  temp_measurement: .word 130
 3|  temp_upper: .word 150
 4|  temp_lower: .word  100
 5|
 6|  .text
 7|
 8|  error:
 9|    /* Send a signal to a connected monitoring device. */
10|    /* Turn on a red LED. */
11|    mov r0, #0
12|    b end
13|
14|  all_off:
15|    and r3, r7, #6
16|    eor r7, r7, r7
17|    mov r0, #1
18|    b end
19|
20|  cool_on:
21|    /* Check if cool is already enabled. */
22|    and r3, r7, #2
23|    cmp r3, #2
24|    /* If yes, go to error. */
25|    beq error
26|    /* if no, enable. */
27|    and r7, r7, #2
28|    mov r0, #2
29|    b end
30|
31|  heat_on:
32|    /* Check if heat is already enabled. */
33|    and r3, r7, #4
34|    cmp r3, #4
35|    /* If yes, go to error. */
36|    beq error
37|    /* if no, enable. */
38|    and r7, r7, #4
39|    mov r0, #3
40|    b end
41|
```

Listing 4.3.2

```
42|  .global main
43|  main:
44|    /* Check temperature reading. */
45|    ldr r1, =temp_measurement
46|    ldr r1, [r1]
47|
48|    /* If equal or above temp_upper, call cool. */
49|    ldr r2, =temp_upper
50|    ldr r2, [r2]
51|    /* cmp subtracts r1 from r2 */
52|    cmp r1, r2
53|    bhs cool_on
54|
55|    /* If equal or lower temp_lower, call heat. */
56|    ldr r2, =temp_lower
57|    ldr r2, [r2]
58|    cmp r1, r2
59|    bls heat_on
60|
61|    /* If neither, continue. */
62|    b all_off
63|  end:
64|    bx lr
```

Listing 4.3.3

4.3.3 Exploration

Checkpoint 4.3.4 What is the return value for the program as it is written?

Answer. ?

Checkpoint 4.3.5 What is the return value if you change the `temp_measurement` definition to 151?

Answer. ?

Checkpoint 4.3.6 What is the return value if you change the `temp_measurement` definition to 99?

Answer. ?

Checkpoint 4.3.7 What is the return value of the program if you insert `mov r7, #2` between lines 43, and 44, and change the `temp_measurement` definition to 200?

Answer. ?

Checkpoint 4.3.8 Reflections.

(a) Write down the assembly instructions for comparing registers 1 and 0, and then jumping to a label "repeat" if the value is less than equal.

(b) Consider grocery shopping. When it comes to buying more toilet paper, or milk, you likely go through some sort of evaluation before doing so. Pick 4 such evaluations, and write them in assembly. You can use variable names, or made up numbers to represent values, but the comparison and branch instructions should be proper assembly insructions. Please comment your code.

[23] `https://github.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/raw/master/assembly_control_structures/p3.s`

Chapter 5

High-level Programming Languages

Giving a microprocessor instructions through assembly is tedious, as it requires seemingly infinite explicit instructions that quickly become repetative, and difficult to understand. Even with procedures, loops, and branching.

Early on, computer users started re-using code and working to make it more human readable, from that emerged the concept of high-level programming languages. These programming languages make complex but common tasks such as comparing numbers, working with text, interfacing with humans, or external data sources as simple as typing a phrase, and providing easily understandable instructions.

While all programming languages require that you provide instructions, the instructions that are provided with high-level languages are much more general, and obscure impressive amounts of low-level work.

We will build upon our assembly experience with a more complex example of previous exercises, and then we will review how the same process might be handled with a high-evel programming language called Python.

This introduction to Python will touch on the variables, expressions, and statments, and introduce functions.

5.1 Associated Reading

- Think Python: How to Think Like a Computer Scientist. 2nd Edition[24]
  - Chapter 1: The way of the program
  - Chapter 2: Variable, expressions and statements

5.2 High Level Programming Languages

5.2.1 Discussion

High-level programming languages exist to make programming easier. They are ubiquitous, and when it comes to programming for personal computers, or embedded systems, learning to make use of them will be a fantastic boon to your project.

With the Raspberry Pi, you can make use of any number of programming-languages, but we are going to focus on Python. It is an interpreted language, this means that it isn't necessary to compile the program you write. Additionally, Python has comparatively liberal data typing, which makes it a bit easier for a beginner to deal with strings of letters, integers, and floats. While data typing is very important, and has it's place in Python, it is less of a learning hurdle with Python than with a language like C, or C++.

[24] https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist

Before we get into the weeds of writing our own programs, I am going to introduce you to some of the basics of the Python programming language by showing you a program, and then describing some of the functions of the code.

Program: dice.py

5.2.2 Code

Project 10 Download the Source[25]

```
#!/usr/bin/python3

import sys
import getopt
from os.path import exists
import json
import random

def calc_combinations(roll_totals, remaining_die, cumulative_rolls):
  for r in range(1, 6 + 1):
    if remaining_die > 1:
      die_left = remaining_die - 1
      calc_combinations(roll_totals, die_left, cumulative_rolls + r)
    else:
      roll_totals.append(cumulative_rolls + r)


def main(argv):
  # Check if configuration file exists.
  file_name = 'die_stats.json'
  if(exists(file_name)):
    file_handle = open(file_name, 'r')
    stats = json.load(file_handle)
    file_handle.close()
    file_handle = open(file_name, 'w')
  else:
  # Initialize stats.
    file_handle = open(file_name, 'w')
    stats = {
          '1':{
            'run_count': 0,
            '1':{"rolls":0, "percentage":16},
            '2':{"rolls":0, "percentage":16},
            '3':{"rolls":0, "percentage":16},
            '4':{"rolls":0, "percentage":16},
            '5':{"rolls":0, "percentage":16},
            '6':{"rolls":0, "percentage":16},
          }
        }

  # Look for command line input of N die.
  help_string = 'output_with_format.py -i <file_name>'
  try:
    opts, args = getopt.getopt(argv, "hn:")
  except getopt.GetoptError:
    print(help_string)
    sys.exit(2)

  number_of_die = None

  for opt, arg in opts:
    if opt == '-h':
      print(help_string)
    elif opt in ("-n"):
      number_of_die = arg

```

Listing 5.2.1

```
  57|    # If NO, ask for user input of N Die.
  58|    if number_of_die == None:
  59|      number_of_die = input("Enter the number of die you'd like to roll: ")
  60|
  61|    # Make sure that any input is an int.
  62|    number_of_die = int(number_of_die)
  63|    str_number_of_die = str(number_of_die)
  64|
  65|    # Check the user input for a reasonable value.
  66|    if number_of_die < 1:
  67|      print('You must enter a number greater than 1.')
  68|      sys.exit(2)
  69|
  70|    # Initialize stats for N die if it doesn't already exist.
  71|    if not str_number_of_die in stats:
  72|      die_combinations = []
  73|      calc_combinations(die_combinations, number_of_die, 0)
  74|      roll_values = (list(set(die_combinations)))
  75|
  76|      stats[str_number_of_die] = {}
  77|      stats[str_number_of_die]['run_count'] = 0
  78|      for rv in roll_values:
  79|        roll_possibilities = len(die_combinations)
  80|        roll_frequency = die_combinations.count(rv)
  81|        roll_percentage = int(roll_frequency / roll_possibilities * 100)
  82|        stats[str_number_of_die][str(rv)] = {"rolls":0,
    "percentage":roll_percentage}
  83|
  84|    # Roll Die.
  85|    print("Rolling {} die.".format(number_of_die))
  86|
  87|    rolls = {}
  88|    roll_total = 0
  89|
  90|    for d in range(1, number_of_die + 1):
  91|      rolls[d] = random.randint(1, 6)
  92|      str_roll = str(rolls[d])
  93|      stats['1'][str_roll]['rolls'] = stats['1'][str_roll]['rolls'] + 1
  94|      stats['1']['run_count'] = stats['1']['run_count'] + 1
  95|      roll_total = roll_total + rolls[d]
  96|    str_roll_total = str(roll_total)
  97|    stats[str_number_of_die][str_roll_total]['rolls'] =
    stats[str_number_of_die][str_roll_total]['rolls'] + 1
  98|    stats[str_number_of_die]['run_count'] =
    stats[str_number_of_die]['run_count'] + 1
  99|
 100|    # Output die results.
 101|    for d in rolls:
 102|      print("Die {} rolled a {}.".format(d, rolls[d]))
 103|
```

Listing 5.2.2

```
104|    # Output die statistics.
105|    str_actual_roll_percent =
    int((stats[str_number_of_die][str_roll_total]['rolls'] /
    stats[str_number_of_die]['run_count']) * 100)
106|    print("Out of the {} times that this program has been run {} has been
    rolled {} times, a percentage of {}. A roll of {} should occur {}% of the
    time.".format(stats[str_number_of_die]['run_count'], str_roll_total,
    stats[str_number_of_die][str_roll_total]['rolls'], str_actual_roll_percent,
    str_roll_total, stats[str_number_of_die][str_roll_total]['percentage']))
107|
108|
109|    # Save config and statistics to a file.
110|    stats_string = json.dumps(stats, indent=4)
111|    file_handle.truncate()
112|    file_handle.write(stats_string)
113|
114|  if __name__ == "__main__":
115|    main(sys.argv[1:])
```

Listing 5.2.3

5.2.3 Exploration

This program will effectively roll virtual dice. It takes user input in the form of the number of die that the user wants to roll, and randomly generates numbers in the range of 1, to 6 for each die. For a lark, it will track the number of times each total is rolled, and save those statistics in a file. At its conclusion, the program will output the roll total, the roll for each die, as well as some information about the statistics for that total.

```
user@server:~/$ python3 dice.py -n 2
Rolling 2 die.
Die 1 rolled a 5.
Die 2 rolled a 4.
Out of the 56 times that this program has been run 9 has been rolled 6 times, a
    percentage of 10. A roll of 9 should occur 11% of the time.
```

Listing 5.2.4

While this program isn't exactly short, it abstracts away tens of thousands of machine code instructions, and represents hundreds of hours of work from minds better than my own, as well as thousands of hours of my own time saved due to the reuse of code.

If you'd like to play with the program, you can find the code in the Intro to Embedded Systems GitHub Repo[26].

5.3 Libraries

5.3.1 Discussion

This program will effectively roll dice. It takes user input in the form of the number of die that the user wants to roll, and randomly generates numbers in the range of 1, to 6 for each die. For a lark, it will track the number of times each total is rolled, and save those statistics in a file. At its conclusion, the program will output the roll total, the roll for each die, as well as some information about the statistics for that total.

[25] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/python_high-level_programming/dice.py

[26] https://github.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/blob/master/python_high-level_programming/dice.py

The first part of the application, in lines 3 to 7 import libraries that include functions that we will make use of in our application.

5.3.2 Code

```
3|  import sys
4|  import getopt
5|  from os.path import exists
6|  import json
7|  import random
```

Listing 5.3.1

5.3.3 Exploration

Line 6 imports a library that will read, or write our statistics from our file, doing so in a data format that is widely used, and human readable. It will eliminate the need for us to design our own method for saving the statistics for our rolls.

Line 7 imports a library that will generate ”random” numbers for us. We specify a range, and it returns a number. While it is beyond the scope of this material, the idea of truly random numbers is a whole subject of it’s own, and I just encourage you to appreciate the amount of work that this library saves you, and to take a moment to enjoy the fact that someone has already gone through the trouble of writing the code to accomplish it.

5.4 Functions

5.4.1

Next, we define two functions. The first that you see occurs on line 9, the second on line 18. When discussing Assembly, we touched on the idea of reusing code. Functions allow us to group instructions, and to re-use instructions in a way that saves us convoluted code, and improves the readability of our work.

The function `calc_combinations` on line 9 makes use of a concept called recursion to calculate all of the possible rolls for any number of die. It is called later to generate some stats, like the possible roll values, and how often those values may occur.

5.4.2

```
 9|  def calc_combinations(roll_totals, remaining_die, cumulative_rolls):
10|    for r in range(1, 6 + 1):
11|      if remaining_die > 1:
12|        die_left = remaining_die - 1
13|        calc_combinations(roll_totals, die_left, cumulative_rolls + r)
14|      else:
15|        roll_totals.append(cumulative_rolls + r)
```

Listing 5.4.1

5.4.3

The function `main` is our program function, and will be called when the program is run. You’ll notice that the full definition `def main(argv):` has parenthesis with the word ”argv” between them. If you look at the `def calc_combinations(roll_totals, remaining_die, cumulative_rolls):` function definition, you’ll see the same pattern, with different words, separated by commas. These words between the parenthesis are

called parameters, and are ”local” variable names for data that needs to be handed off, or pointed to, for the function to do its work.

The parameters that are defined with the function are names that you give to data that you, or someone else will provide when the function is called. You use those local names to do the work you need to do in your function. When the function is called, the data that is passed into the function can have any variable name, it doesn't have to match the names of the parameters. On line 73, `calc_combinations` is called with `calc_combinations(die_combinations, number_of_die, 0)`. None of the names passed match the names used in the function definition, in fact, one of them is just an integer. The function gives each of them it's own name, and does the work it needs to do.

Variable scope and the way that the data passed into a function are referenced are their own subject for study and are beyond the scope of this exploration. Know that variables defined in functions as parameters or otherwise, are only usable/visible within the same function. If a variable that is defined in your program has the same name as a variable in a function, they don't share data unless you pass that variable to the function when you call it.

5.5 Variables

5.5.1

When discussing Assembly, we touched on the concept of variables as names given to data that we could use later. In Assembly, it was necessary to decide the type of data that was going to be referenced by the variable at the time that it was declared. That is the case with several high-level programming languages also. However, in Python, PHP, JavaScript, as well as others, that requirement doesn't exist. However, it is still necessary to pay attention to data types, and we will touch on that more later.

In Python, to set up a variable, you simply write a name for it, and assign it a value. The type of data that it can reference can be an int, float, character, or an object such as a string, list, dictionary, file, etc. You can also change the type of data that a variable references by simply reassigning it. This approach is quite different from C based languages, but that discussion is beyond the scope of this exploration.

On line 20 of our program, we declare a variable `file_name = 'die_stats.json'`. This creates a string object, and makes it available to use with the name file_name. On the next line, we pass this variable to a function `exists(file_name)` which looks at the file system of the computer, and checks to see if a file with that name already exists. If it does, we declare a new variable, that will point to another object that represents a connection to the file for reading its contents with `file_handle = open(file_name, 'r')`

If the file doesn't exist, we go ahead and create a new file with `file_handle = open(file_name, 'w')`, and initialize another variable `stats` that will point to a data structure called a dictionary. The dictionary itself is composed of many different variables: Characters '1' through '6', 'rolls', and 'percentage'.

Variables will be everywhere in your program. Learning how to use them well will be key to good programming. Additionally, giving them proper names will be key to great programming. Giving a variable a long, descriptive name doesn't effect the speed of your code, but does dramatically improve the readability of your code. Conversely, a long descriptive variable name can be irritating to type repeatedly, so you'll need to find your own balance.

Here are a few examples of common case types used in programming:

Table 5.5.1

Case Name	Example
Camel Case	thisIsMyVariableName
Snake Case	this_is_my_variable_name
Kebab Case	this-is-my-variable-name
Pascal Case	ThisIsMyVariableName

5.6 Control Structures

5.6.1

There are a couple of control structures available, and they are pretty much universal to all programming languages. The first that we will discuss, and that we encounter in our program is the `if` statement, and it's variations.

On line 11 we have `if remaining_die > 1:`. This statement is composed of a call to `if` with a test condition. At its simplest implementation, this is all you need to have some code executed based upon the truth of the test condition. There are situations where you want code to execute only when a condition is true, and then there are situations where you will want to choose a path for your program based upon the test condition, and maybe subsequent test conditions. In those cases, you'd add an `else if:` or `else:` statement like the one seen on line 14.

5.6.2

```
11|      if remaining_die > 1:
12|        die_left = remaining_die - 1
13|        calc_combinations(roll_totals, die_left, cumulative_rolls + r)
14|      else:
15|        roll_totals.append(cumulative_rolls + r)
```

Listing 5.6.1

5.6.3

The next control structure type is the loop, of which there are two variations. The first is a while loop, the second a for loop.

The while loop will execute forever, as long as a condition exists. There are times where you won't know when you'll want to exit at the time that you start you loop. The while is most useful for these situations. As an example, if you wanted to monitor a stream of data, you could use a while loop that looks for some value in the stream, and exits when the condition is met.

The for loop is useful for situations where you know how many elements you're going to be dealing with ahead of time. As an example, if you have a list of items, and you want to perform an operations for each item in the list, using a for loop is very convenient, as it will execute the code in the loop once for each item in the list, and make that item available to you for use. On line 78 a for loop is used to iterate over a list of every possible roll of N die. With each iteration of the loop, the item in the list is made available in the variable `rv`. That variable is then used to do some calculations related to probability, and the data is saved to the stats data structure for later use.

5.7 Structure

5.7.1

You may notice, while looking over our program, that most of our lines are indented. All lines are indented by some multiple of 2. Python uses indentation to group lines of code into blocks. This indentation level indicates if a line of code belongs to a function, an `if`, `else`, or `for` statement, or some other program structure.

When writing Python programs, it is essential that you pay attention to the indentation level of the line you're writing. Failure to do so will result in code that doesn't work the way you expect it to.

It is notable that Python is unique in the use of white space for this purpose. Most programming languages use curly brackets to group lines of code.

5.7.2

```
function sum(array numbers)
{
  for($i = 0; $i < len(numbers); $i++)
  {
    # Do some stuff.
  }
}
```

Listing 5.7.1

Checkpoint 5.7.2 Reflections.

(a) How would you write an if, else if, else statement in Python? How would you do the same in C++?

(b) Which case convention makes the most sense to you? Write 5 variable names in your chosen convention.

(c) With your knowledge of Assembly, and now Python, which is your preferred programming tool? Why?

Chapter 6

Setting up for, and exploring python programming.

Different integrated development environments bring different strengths and weaknesses to the process of solving problems with programming.

The Code::Blocks IDE that has been used for assembly programming so far isn't as well suited for Python as the Thonny Python IDE. We will go through the process of writing our first simple Python program in this new IDE.

6.1 Raspi Config

In order to use the GPIO pins that we will need for our next couple of experiments we need to enable some features of the Raspberry Pi.

6.1.1

To perform the next few steps, we will need to open the *Terminal* application.

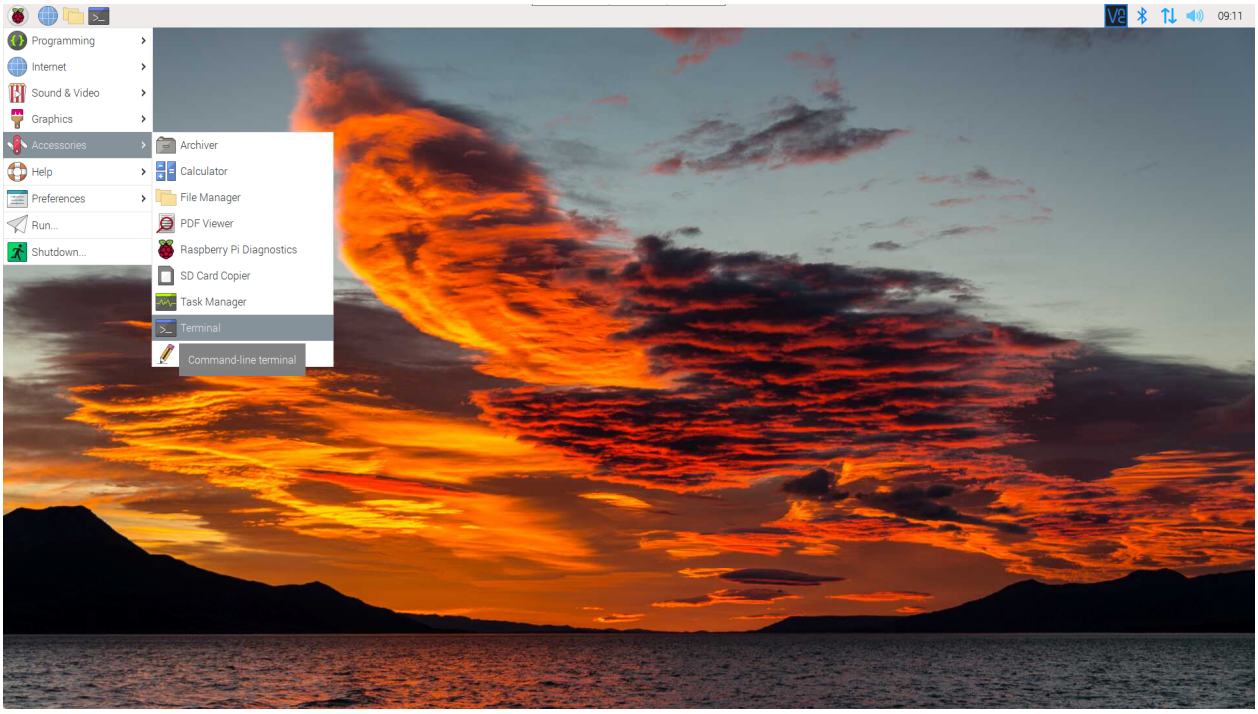

Figure 6.1.1 Opening the Terminal application.

1. Click *main menu* (Raspberry Pi Icon on the Taskbar)
2. ... *Accessories*
3. ... *Terminal*

6.1.2

Next, launch the Raspberry Pi Software Configuration Tool by typing the following:

```
sudo raspi-config
```

Listing 6.1.2

6.1.3

When the configuration tool is displayed, use your arrow, and enter keys to execute the following commands.

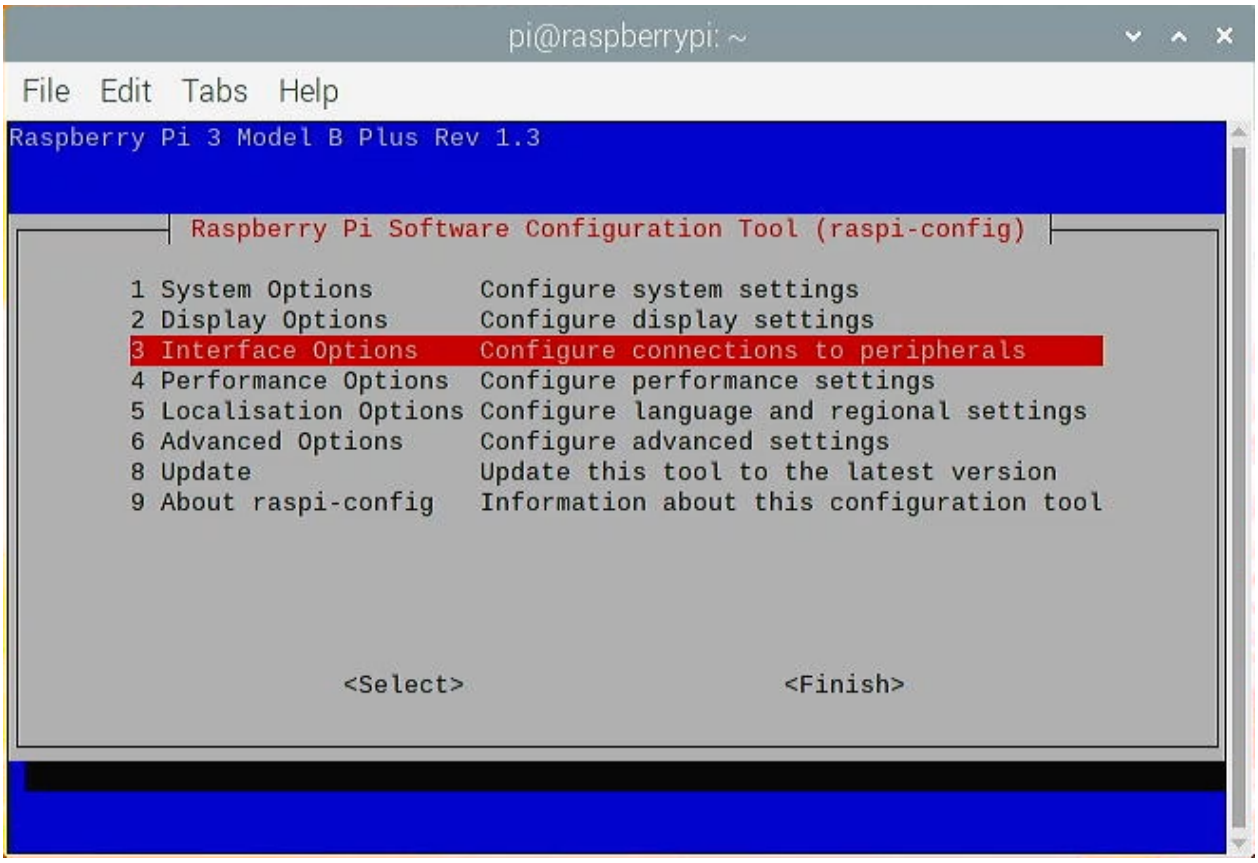

Figure 6.1.3 Opening the Terminal application.

1. Select *Interface Options*.

6.1.4

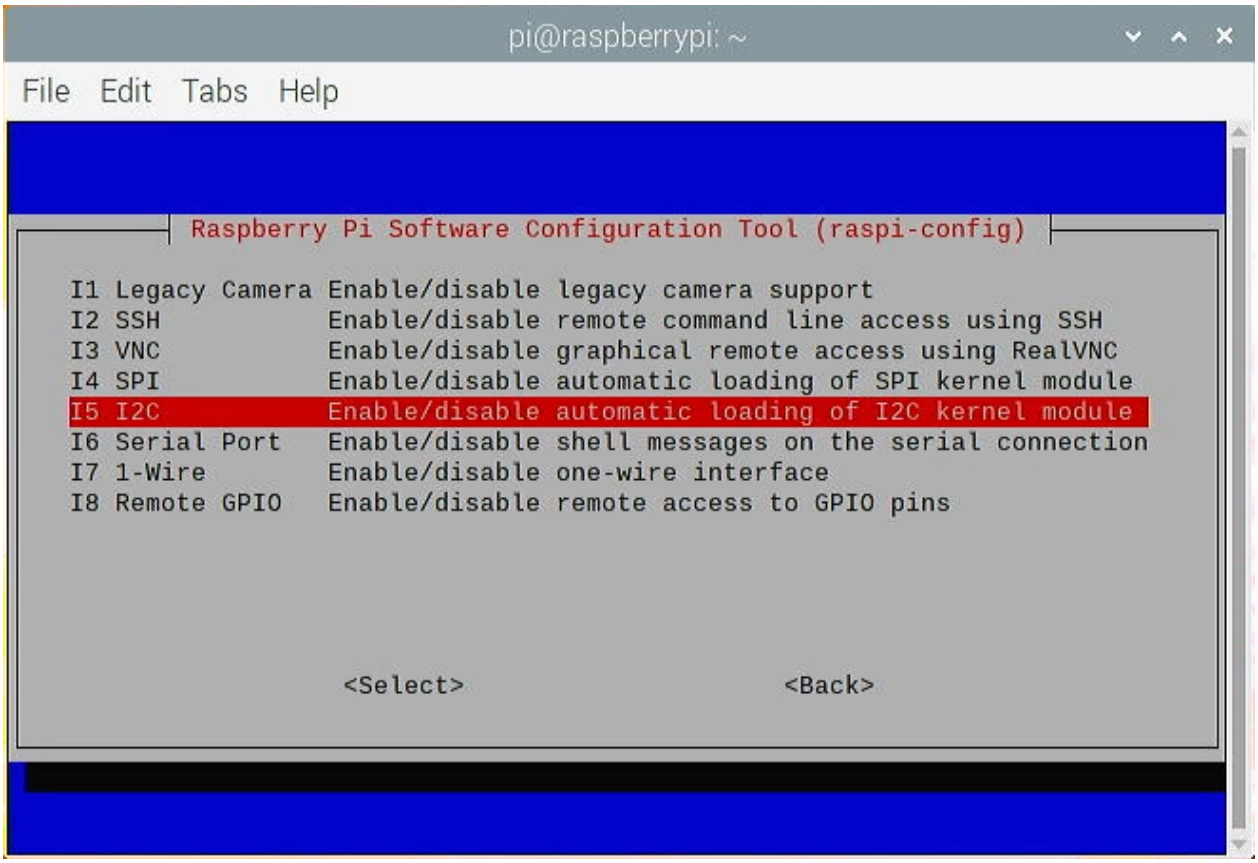

Figure 6.1.4 Opening the Terminal application.

1. Select *Enable/disable automatic loading of I2C kernel module.*
2. On the following screen, select *Yes.*
3. Finally, select *Ok.*

After going through those steps, you should be returned to the main menu of the Raspberry Pi Software Configuration Tool. You can either press *Esc* on the keyboard to return to the terminal prompt, or close out the window.

6.2 Installing Python Libraries

6.2.1

In order to interface with the GPIO of the Raspberry Pi with Python, as well as to communicate with some peripheral devices in the later lab, we are going to need to install some Python libraries.

Open the *Terminal* application, and type the following line:

```
pip3 install adafruit-circuitpython-ssd1306 adafruit_ssd1306 adafruit-circuitpython-ahtx0
    adafruit-blinka
```

Listing 6.2.1

The Raspberry Pi will find the libraries and download them. When done, you can close out the *Terminal* window.

6.3 The Thonny Python IDE

6.3.1

The Thonny Python IDE is already included with a fresh installation of the Raspberry Pi OS. To get started, you only need to launch the application from the main menu.

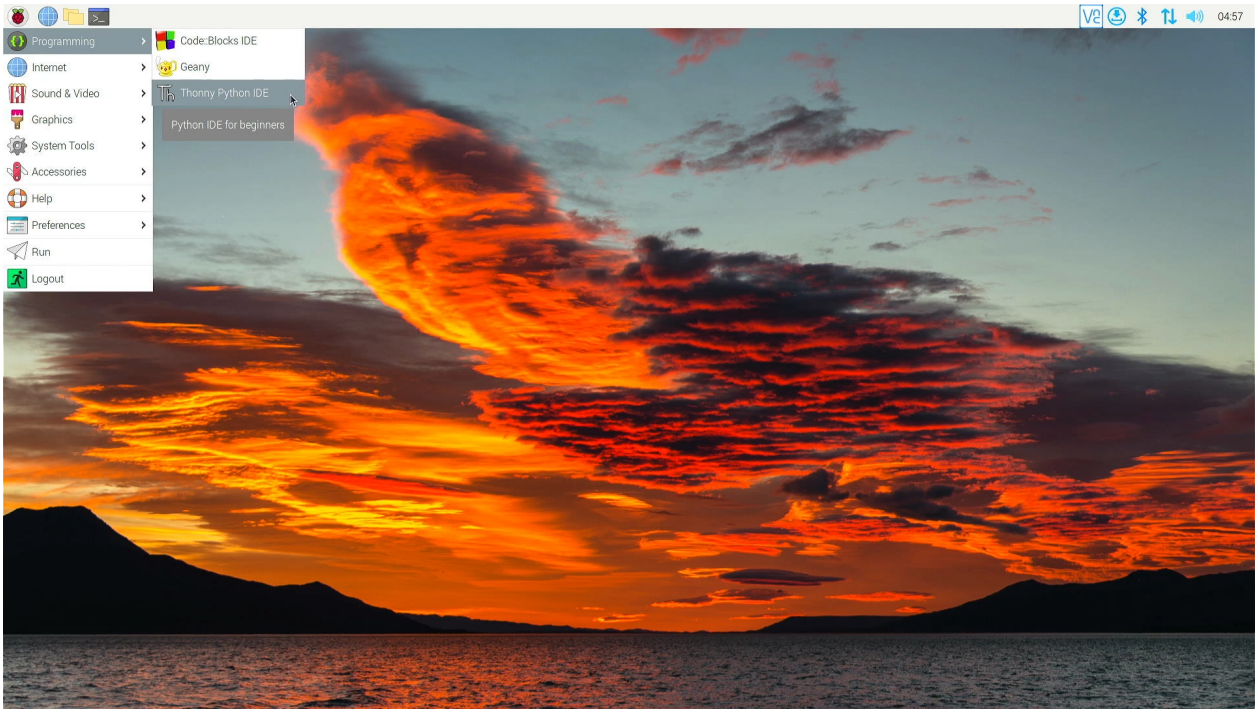

Figure 6.3.1 Selecting the Thonny Python IDE in the Raspberry Pi OS main menu.

1. Click *main menu* (Raspberry Pi Icon on the Taskbar)
2. ... *Programming*
3. ... *Thonny Python IDE*

6.3.2

Once Thonny is open, save the new empty file.

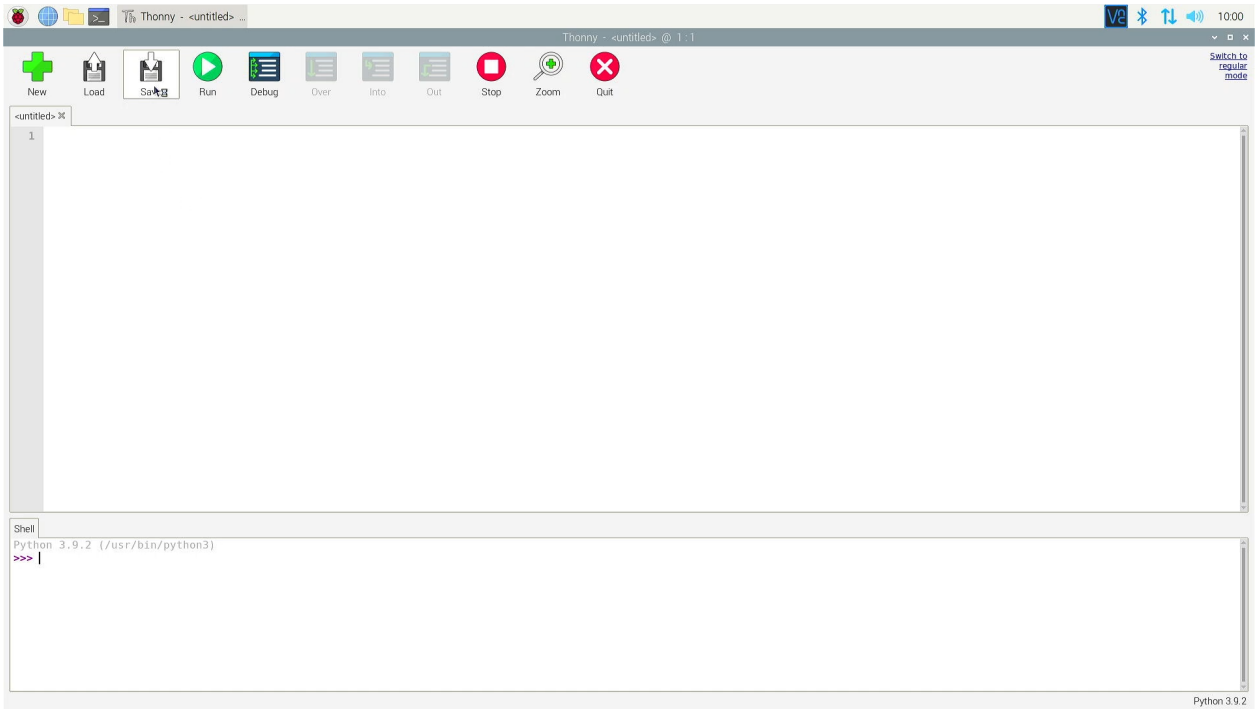

Figure 6.3.2 Saving the new program.

1. Click *Save* in the shortcut bar.

When the *Save as* window opens:

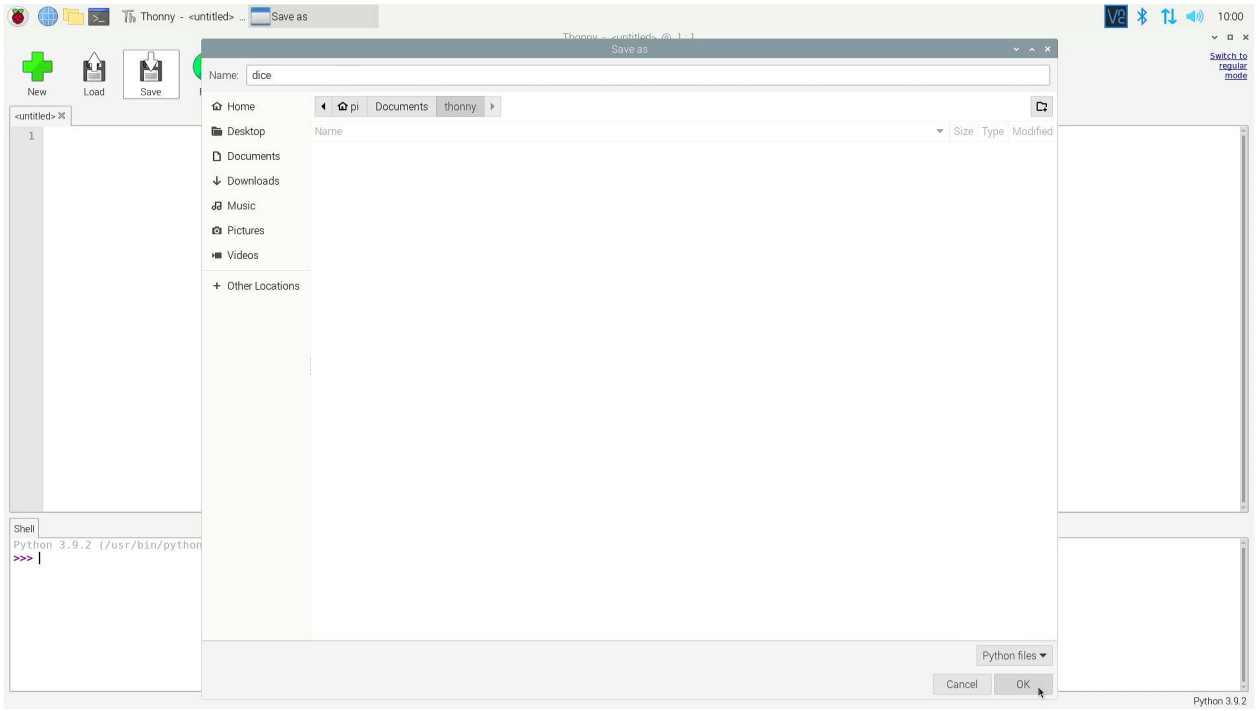

Figure 6.3.3 The *Save as* window.

1. Navigate to an appropriate place for saving your files.

2. Type *dice* in the *Name:* field.
3. Click *OK*.

6.3.3

With your new empty program saved, type out the following program, and save your progress.
Download the Source[27]

```
 1|  # Import the random library.
 2|  import random
 3|
 4|  # Ask the user for the number of die they want to roll.
 5|  number_of_die = input("Enter the number of die you'd like to roll: ")
 6|  number_of_die = int(number_of_die)
 7|
 8|  # Initialize some variables that will store our roll data.
 9|  rolls = {}
10|  roll_total = 0
11|
12|  # Using a loop, roll each die, and save the roll results to the variables.
13|  for d in range(1, number_of_die + 1):
14|      rolls[d] = random.randint(1, 6)
15|      roll_total = roll_total + rolls[d]
16|
17|  # Output die results.
18|  # Output individual results.
19|  for d in rolls:
20|      print("Die {} rolled a {}.".format(d, rolls[d]))
21|  # Output the combined results.
22|  print("The roll total was: {}.".format(roll_total))
```

Listing 6.3.4

6.3.4 Running a Python Program

Now that the program is written, you can simply click the *Run* button in the shortcut bar.

[27] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/python_environment_setup/dice.py

Figure 6.3.5 Running your python program.

In this case, you should see a prompt in the *Shell* pane below the code editor pane.

Enter a number, like 2, and press *Enter*.

The program will generate some random numbers, and output the results.

You may find that there are occasions where you need to manually quit the program. If you find yourself accidentally stuck in an infinite loop, or waiting for an input that never comes, you can click the *Stop* button in the shortcut bar, and your program will exit.

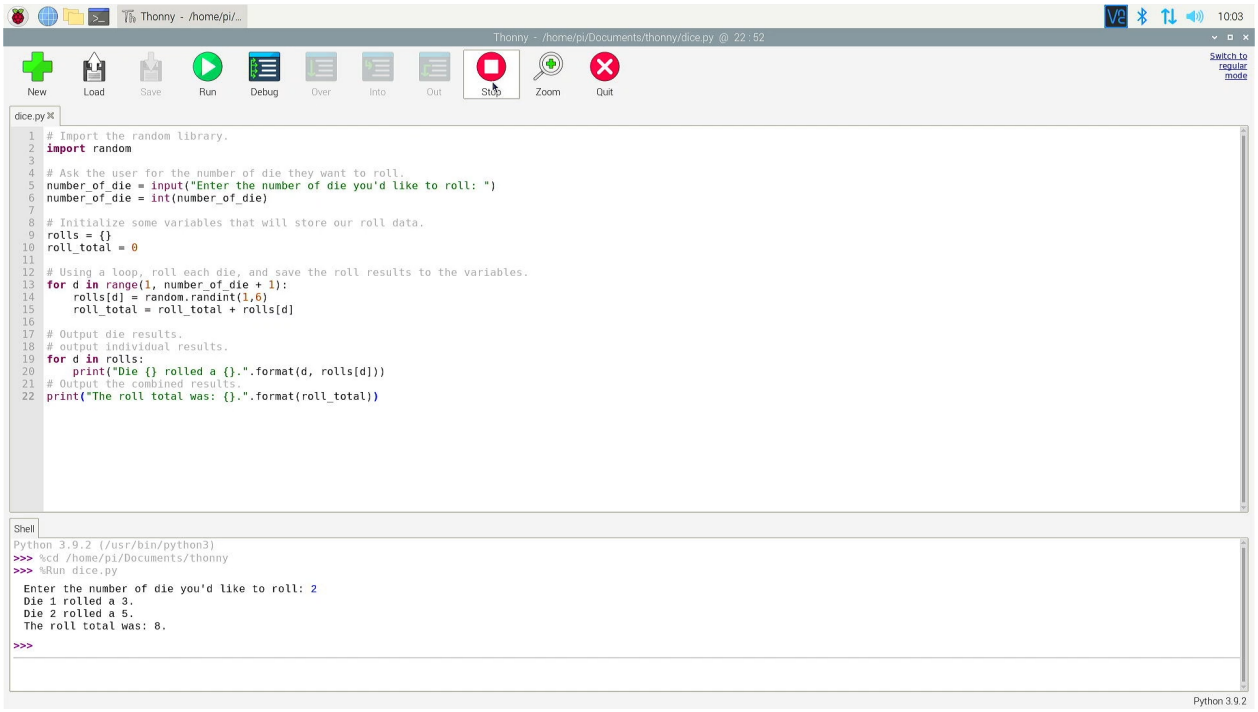

Figure 6.3.6 Stopping your python program.

6.3.5 Debugging with Thonny

As with Assembly, there is a near certainty that you're going to need to debug your program. Thonny has a very handy set of features for exactly this.

To debug, you can simply click the *Debug* button in the shortcut bar. When you do, your program will start, but will wait on the first instruction line for your instruction. Additionally, a new *Variables* pane will open to the right of the code editor pane.

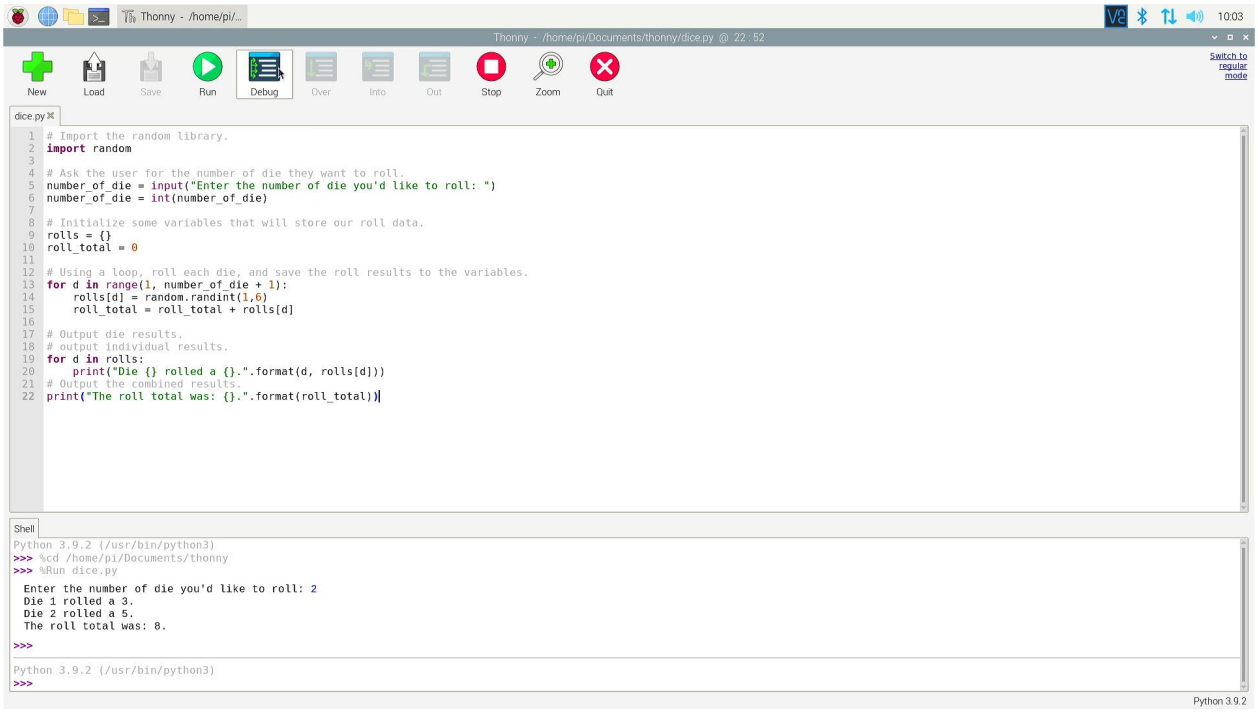

Figure 6.3.7 Debugging your python program.

To advance to the next instruction, click the *Over* button in the shortcut bar.

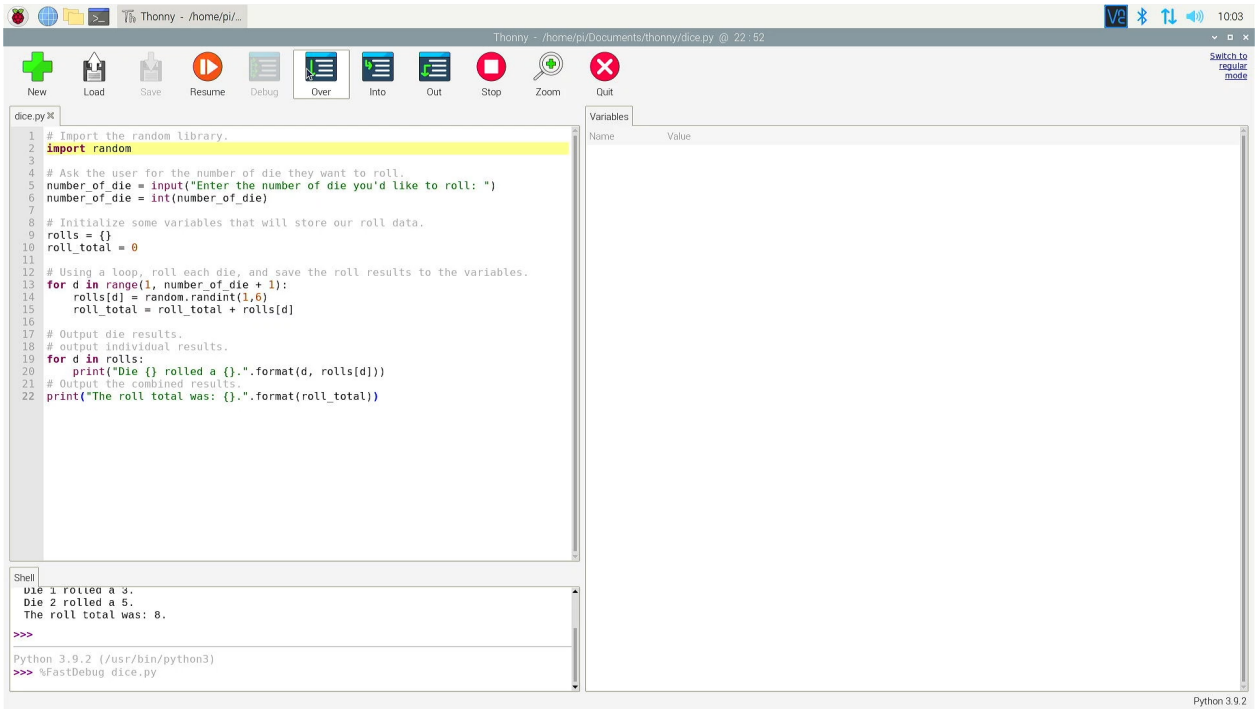

Figure 6.3.8 Advancing the debugger to the next instruction.

As you click through the lines of the program, you will see variables be added to the list in the *Variables* pane. This helps you keep track of what variables have been declared, and what their values are.

You'll notice that the `number_of_die` variable is initially loaded with a string value of `'2'`, and then changed to an int value of `2`. The difference is subtle, but very important.

If you're dealing with a larger program, or just want to focus on debugging a specific part of the program, you can add a break point by double-clicking the line number in the index to the left of the code. When you click *Debug* again, the debugger will run through the program up until the break point, then wait for your instruction.

When you're ready to return to editing your code, you will need to click the *Stop* button to exit the debugger.

6.4 Debugging with Python

While the Thonny Python IDE has some very powerful features that make debugging easy, there are some tricks that you should be aware of that will help you as you explore programming with Python.

The first thing to understand is that simple concepts like a sentence are in fact, incredibly complex structure that we often take for granted. When it comes to working with text, or any other data structure in python, there are typically a myriad of operations that are built in to make working with them easier.

```
1|  text = "This is a STRING."
2|
3|  print(text)
```

Listing 6.4.1

For example, if you wrote the above program, we would see "This is a STRING." in the *Shell* output.

Now, if we wanted to search for a word or character within, break it up into multiple sections, change the case from upper to lower, or many other common operations, there are functions built into the Python `string` object that can help us.

```
1|  text = "This is a STRING."
2|
3|  text = text.upper()
4|  print(text)
5|
6|  print(text.lower())
```

Listing 6.4.2

If you run the code above, you'll see the same sentence in different cases.

```
THIS IS A STRING.
this is a string.
```

Listing 6.4.3

Now, to find these functions, you can, and should feel empowered to search online. However, you can also use some built-in Python tools.

```
1|  from pprint import pprint
2|
3|  text = "This is a STRING."
4|
5|  pprint(dir(text))
```

Listing 6.4.4

The `dir` function that is used above is the star of the program. This function will list all of the functions that the object has for you to use. In the output of the program, you'll see the functions `upper` and `lower` that were used in the previous example.

There is a similar function `vars` that will also output the variables in the object. But it will only work on Python objects that have been setup to be compatible with it.

```
1|  from pprint import pprint
2|  import random
3|
4|  pprint(vars(random))
```

Listing 6.4.5

The above program will output pages of information about the random Python object.

Between these two functions, you can explore and learn quite a bit about how to use, or the state of the data in your program.

Checkpoint 6.4.6 Reflections.

(a) When would you use the `dir` funciton to debug? Run the code in Listing 6.4.4 and pick 3 functions. Provide a description of what each function does.

(b) When is the `vars` function useful? Run the code in Listing 6.4.5. There are 2 constants defined in the `random` class. What are their variable names, and what are their significance in math?

(c) Write a simple Python program that uses the random library to generate a random number and print it out. Include a screenshot that captures the code and the output.

Chapter 7

Python Basics

High-level programming languages like python make very complicated tasks like interfacing with humans incredible simple.

What is one or two lines in Python is likely thousands of lines in assembly, because, as we've explored, seemingly simple tasks are in fact, made up of many elementary steps.

We are going to explore some basic Python programming by getting user input, and providing natural language responses. This will be done through extensive use of functions.

7.1 Associated Reading

- Think Python: How to Think Like a Computer Scientist. 2nd Edition[28]
    - Chapter 3: Functions

7.2 Basic Python Programs

7.2.1 Discussion

When programming with Assembly, we used labels and branching to change the flow of our program based upon conditions. With high-level programming languages, the equivalent operation is achieved with something called a function. Like jumps and labels in Assembly, you define a name, call it to execute some instructions, and then are returned to the original position to continue with your program.

With high-level programming languages, so much of the basic computer operation is abstracted into functions that they are nearly all that you'll work with. In this lab, we will explore this subject with a simple Python program.

7.2.2

Project 11 Code. Transcribe, compile, and run the program.

Download the Source[29]

[28] https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist

```
 1|  import sys
 2|  import random
 3|
 4|  # Ask the user to tell us how many die they wish to roll.
 5|  number_of_die = input("Enter the number of die you'd like to roll: ")
 6|  # Make sure that any input is an int.
 7|  number_of_die = int(number_of_die)
 8|  # Check the user input for a reasonable value.
 9|  if number_of_die < 1:
10|    print('You must enter a number greater than 1.')
11|    sys.exit(2)
12|
13|  # Initialize a list of possible faces.
14|  die_faces = [4, 6, 8, 10, 12, 20]
15|  # Ask the user how many sides they want for the die that are rolled.
16|  sides_of_die = input("How many sides for each die? [{}]: ".format(',
    '.join(str(item) for item in die_faces)))
17|  # Make sure that any input is an int.
18|  sides_of_die = int(sides_of_die)
19|  # Check that the number provided was in our list of possible options.
20|  if sides_of_die not in die_faces:
21|    print('You must choose one of the options listed.')
22|    sys.exit(2)
23|
24|  # Roll Die.
25|  print("Rolling {} die with {} faces each.".format(number_of_die, sides_of_die))
26|
27|  # Create a dictionary that will store the rolls for each die.
28|  rolls = {}
29|  # Initialize a variable that will store the cumulative rolls.
30|  roll_total = 0
31|
32|  # For each die indicated, make a roll.
33|  for d in range(1, number_of_die + 1):
34|    # Select a number within the range possible for the number of sides.
35|    rolls[d] = random.randint(1, sides_of_die)
36|    # Add this new roll to the roll total.
37|    roll_total = roll_total + rolls[d]
38|
39|  # Output die results.
40|  for d in rolls:
41|    print("Die {} rolled a {}.".format(d, rolls[d]))
42|  print("The roll total is {}.".format(roll_total))
```

Listing 7.2.1

7.2.3 Exploration

The first this this program does is import two different libraries that contain the programming instructions for accessing the system functions, and generating random numbers.

Next, on line 5 we call a function called `input`. You'll notice that there are parenthesis, and a string following the function name. The parenthesis accompany every function, and enclose parameters that are passed to the function for use by the program instructions they contain. In this case, the `input` function has one parameter defined, called *prompt* that is expected to contain a string of text from you that it will display to the user of the program. In this call of the `input` function, we are passing an instruction to the

[29] `https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/python_basic_programs/dice.py`

program user asking them for a number.

This particular function will output the text we've provided, and wait for the user to respond by typing something, and pressing enter. It will then take that input from the user, and return it to our program for our use. In our case, we want to store that returned value, and do so in a variable `number_of_die`.

If you debug this program, and step through it until line 7, you'll be presented by the prompt in the *Shell* pane of the Thonny IDE. After you type a number, and press *Enter* you will see a new variable appear in the *Variables* pane. That new variable should have the value that you typed, however, you may notice that the number you typed has single quotes around it.

7.3 Data Types

Variables in Python are incredibly dynamic. They can switch between a float, an int, a char, or an object and not even complain. In our case, the `input` function returns a string, and we stored that string in our variable. However, a string can't be used by some of the function we are going to call later. Additionally, a string doesn't even have to be a number.

For the sake of our dice rolling program, we have to convert this string into a number. And then we have to check this number value, to ensure that the value we have makes sense. For example, we can't roll a negative number of die.

The conversion from string to number is done in a lazy way, by simply calling the `int` function which does quite a bit of background work to determine what integer value it should return for the string value we provided. If you advance the debugger one more time, you'll notice that the *'22'* changes to *22*.

While it may not seem like much, the change from a string to a int has quite a bit of background work being done for us, and has a very significant impact on how we can use that data in later steps.

Checkpoint 7.3.1 What happens to the program if you remove or comment out line 11?

Answer. ?

7.4 Checking User Input

Whenever dealing with external data, it is best practice to double check it for incorrect values. This is important because it will go a long way to prevent crashes, or unintended program behavior. It is also important because there are many cases where unchecked input can result in someone gaining access to the computer, and using it for malicious purposes.

For our simple program, we may not need to worry about someone hacking our computer with carefully crafted input, but we do have to worry about someone entering characters that don't correspond to numbers, or negative numbers. If that happened, our program wouldn't perform the way we expect, and we want to guide our users to a successful experience.

```
9|  if number_of_die < 1:
10|      print('You must enter a number greater than 1.')
11|      sys.exit(2)
```

Listing 7.4.1

On line 9 of our program, we use a conditional control structure called an *if* statement to check if the value that we've stored in the `number_of_die` variable is less than the integer 1. If so, we execute special code that sends the user a message about how to use the program, and exits the program.

```
20|  if sides_of_die not in die_faces:
21|      print('You must choose one of the options listed.')
22|      sys.exit(2)
```

Listing 7.4.2

We do something similar later in the program, on line 20. In this case, we check if the value that the user entered for the sides of the die is in a list of preappreoved values. If the value stored in `sides_of_die` is `not in` the list `die_faces`, the program will execute some code that again provides information to the user, and exits.

Checkpoint 7.4.3 Remove lines 9-11, then run the program. What is the result of the program if you enter 0 for the number of die? How about -1? What happens to the program if you type letters instead of numbers?

Answer. ?

7.5 Providing Output

Many embedded applications may have no need to interact with a user through text. However, it will not be uncommon for such applications to provide output. The output may come in the form of debugging logs that are streamed over a serial output, hidden deep inside the enclosure, or it may be in the form of data streamed to a web server somewhere on the internet for later analysis. Whatever the reason, it is advantageous to understand the basics of providing meaningful output.

```
25|  print("Rolling {} die with {} faces each.".format(number_of_die, sides_of_die))
```

Listing 7.5.1

On line 25 we make use of a print function that is designed to output the provided value to the *stdout*. In other cases, you may use functions designed to write the data to a file, or send it to a server on the internet. However, when calling the function, we create, and pass a string of text in a way that is worth closer examination.

In Python, string objects have a built in function called `format` that takes any number of parameters, and will sequentially replace and `{}` curly bracket pairs that it finds in the string. Special formatting instructions can be included in the curly brackets, and extensive documentation for this can be found in the Python documentation and other sources.

In our case, we have two curly bracket pairs, and we've provided two parameters when we called the `format` function. When the string is formatted, and printed out, we see a nice, human-readable sentence that lets us know what the program is doing.

Checkpoint 7.5.2 Reflections.

(a) In Listing 7.2.1 on line 16, we prompt the user for information. There is a lot going on there. Deconstruct the line, and describe what is happening.

(b) In Listing 7.2.1 on line 14, we create a list. Use the `dir` function on it. While not a function, it is demonstrated to be a more complex data object than a simple grouping of numbers. How many functions does this `list` class make available to you? List 3 of them, and what they do.

(c) Why is it important to check user input? What is an appropriate way to handle invalid input?

Chapter 8

Interfacing Basics

While interfacing with humans at some level is pretty much the entire purpose of computers, embedded systems aren't generally focused as much on general purpose uses. They aren't typically desktop computers with keyboards and mice attached.

Most embedded systems are interfacing with dozens of sensors, and executing control over other systems based upon their inputs. The Raspberry Pi is exceptional for the access that it offers curious people like yourself to this crossover.

We are going to explore the use of the Raspberry Pi GPIO header for interfacing with electronic circuits. This will include reading input, and controlling output based upon it.

We will explore the intersection of hardware and software, as well as make use of itteration structures

8.1 Associated Reading

- Think Python: How to Think Like a Computer Scientist. 2nd Edition[30]
  - Chapter 7: Itteration
  - Chapter 8: Strings

8.2 Connecting the Pi to Peripherials

8.2.1 Discussion

Before we can start programming, and seeing the fruits of our labor on something other than a computer screen, it is necessary to attach our Raspberry Pi to peripherals. As you know, when working with electronics, you have to be aware of the power limitations of the components you're working with. You have to design your circuits with this in mind.

While the circuitry of this lab isn't complicated, doing something incorrectly can result in damage to your Raspberry Pi. It is recommended that you check, and double check your work before powering up your Raspberry Pi.

If you do any of your own design, make sure that you read up on the voltage, and current limits for each of the Raspberry Pi GPIO pins. Allowing too much current or voltage can easily cause damage.

8.2.2 Required Equipment

These are the required components.

[30] https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist

Table 8.2.1

Amount	Part Type
1	Breadboard
1	100nF Capacitor
6	Red LED
6	470Ω Resistor
1	10KΩ Resistor
1	Raspberry Pi 3
1	Push button
Handful	Male to Female Jumper Wires
Handful	Solid Core Jumper Wires

8.2.3 Preview

Our objective is to create a dice rolling circuit. We will provide user input through a momentary switch, and we will get output through LEDs. Our little computer will run a program that responds to a button push, generates a random number between 1 and 6, and then displays the value generated by illuminating a corresponding number of LEDs.

I recommend that care be taken with the placement of the components, as the following lab will build upon this initial setup. Setting it up correctly the first time will save you the trouble of rebuilding things.

This is what we are aiming for:

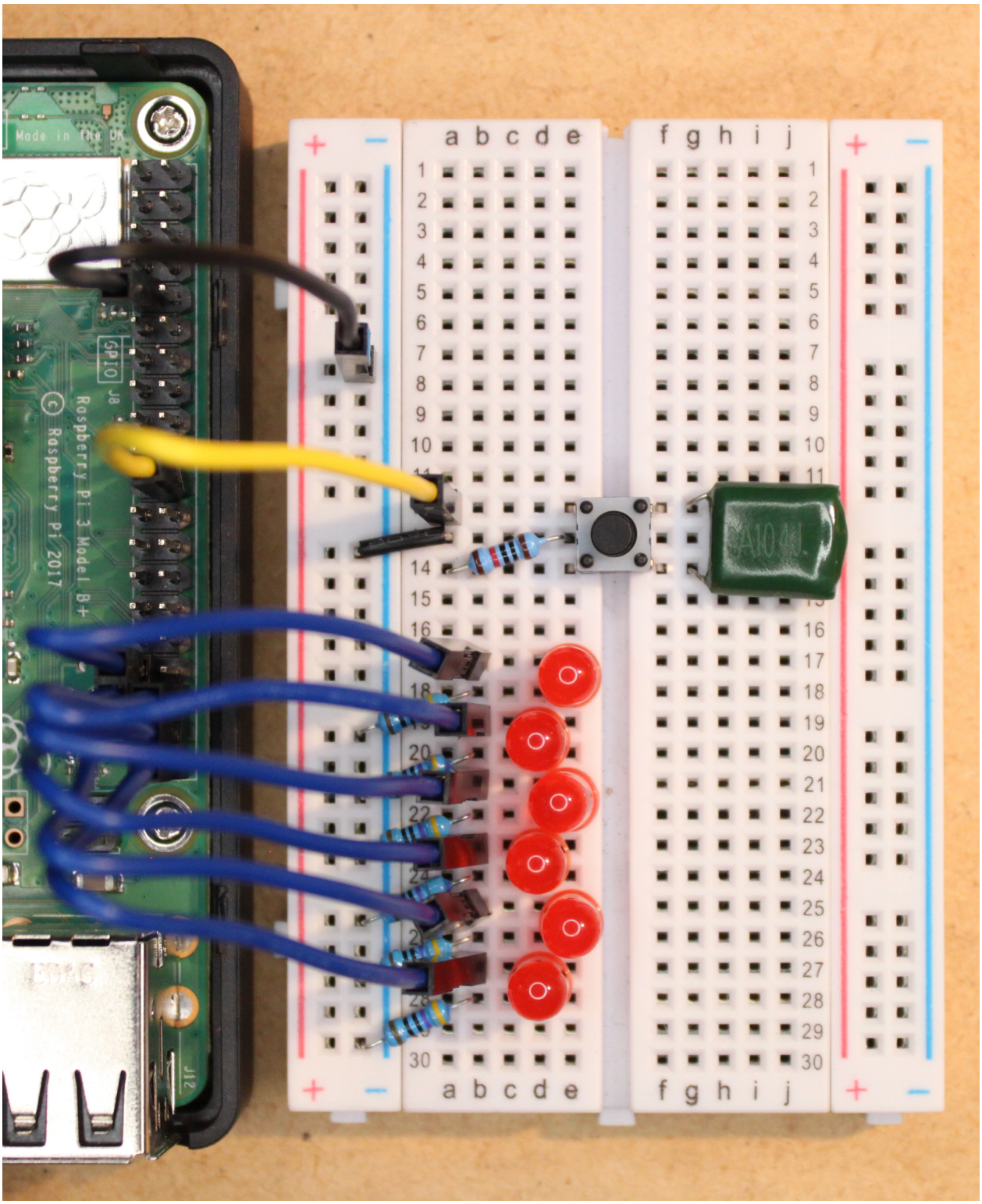

Figure 8.2.2 The Completed Circuit

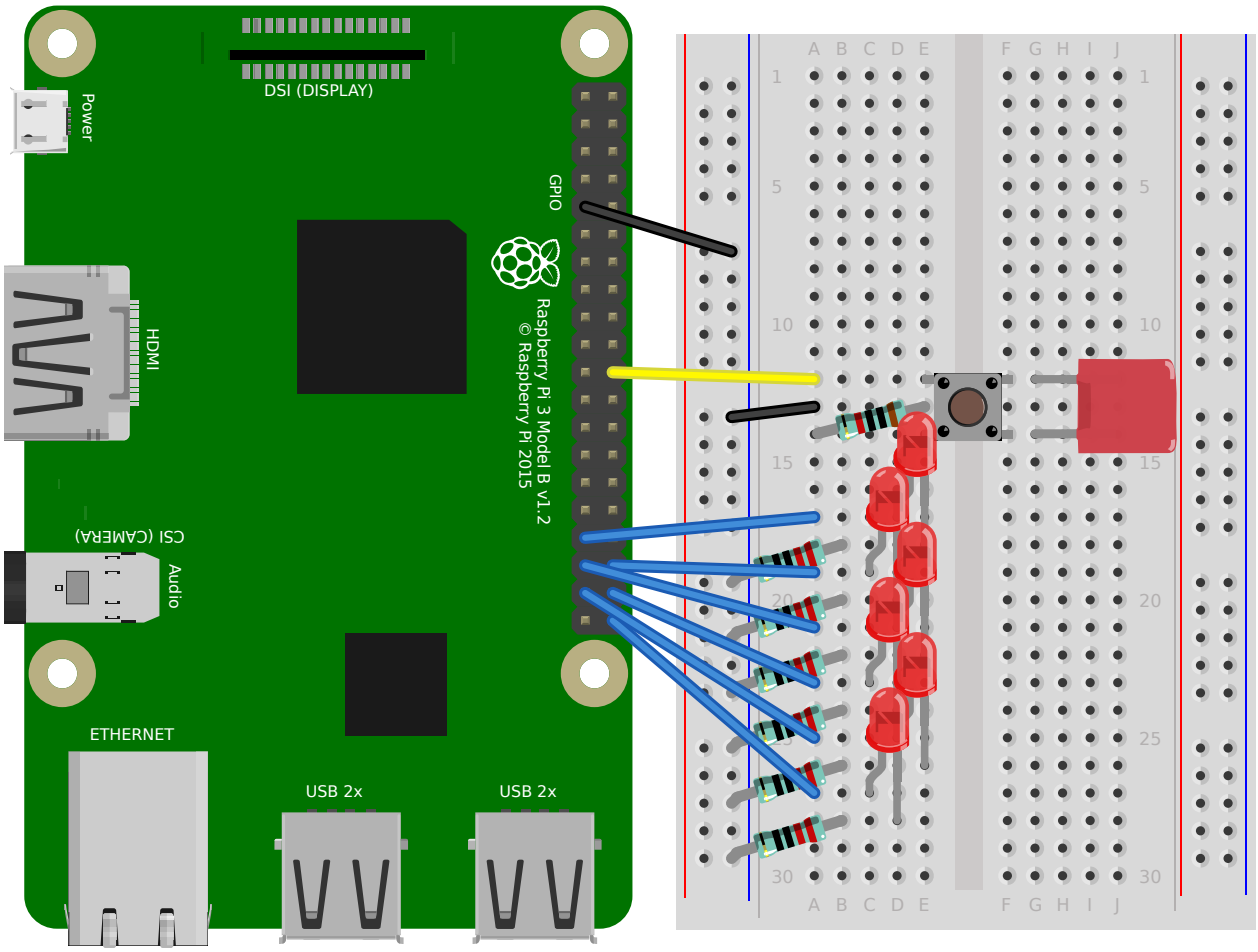

Figure 8.2.3 The Completed Circuit

8.2.4 Wiring

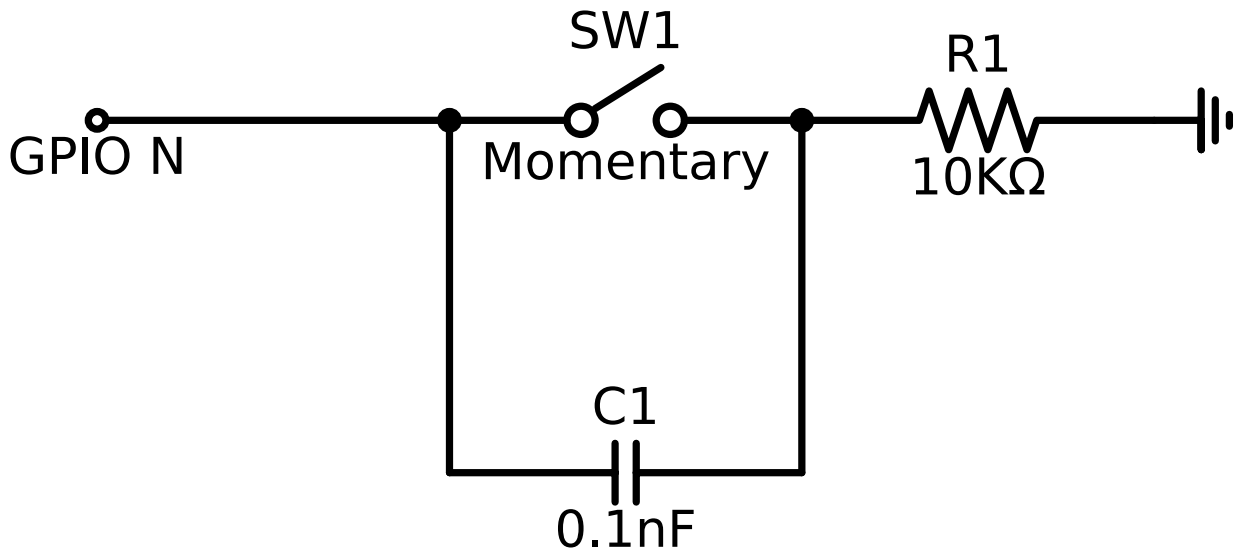

Figure 8.2.4 Circuit Schematic for Momentary Switch

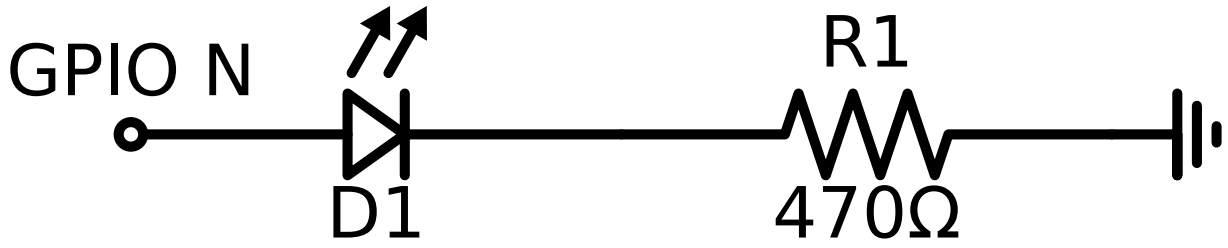

Figure 8.2.5 Circuit Schematic for LEDs

8.2.5 Raspberry Pi GPIO Connections

In order to match the code that will follow, special care will be needed with matching the correct GPIO to the proper components.

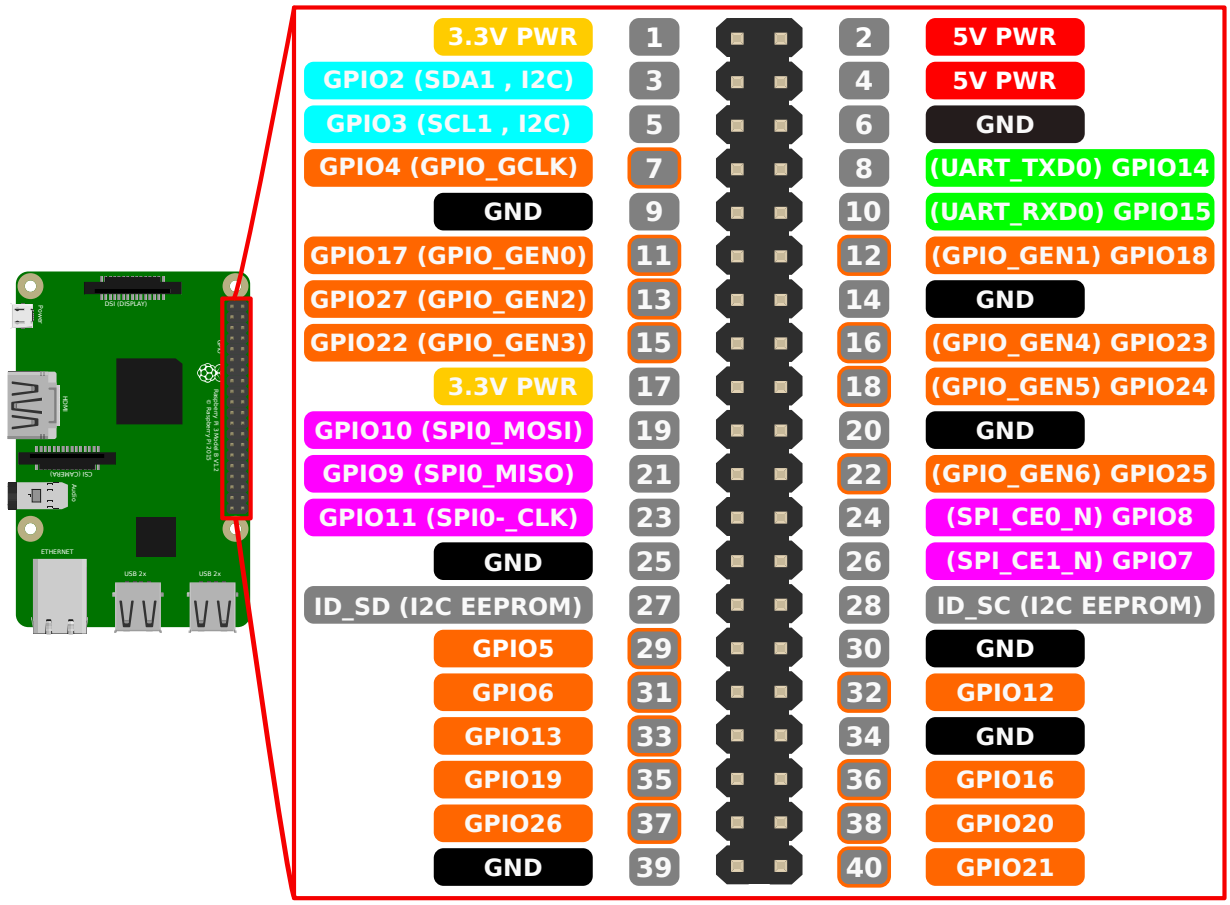

Figure 8.2.6 Raspberry Pi 3 GPIO Pinout

Table 8.2.7

GPIO Designation	Header Pin	Component
GND	Pin 9	GND (Shared)
GPIO25	Pin 22	Switch
GPIO13	Pin 33	LED 1
GPIO16	Pin 36	LED 2
GPIO19	Pin 35	LED 3
GPIO20	Pin 38	LED 4
GPIO26	Pin 37	LED 5
GPIO21	Pin 40	LED 6

8.2.6

Using the schematics, and details provided, wire up the components to the Raspberry Pi. When done, boot it up, and proceed to the next section.

8.2.7 Code

With everything wired up, transcribe the following program, and run it.

Download the Source[31]

[31] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/interfacing_basics/dice.py

```
 1|  import signal
 2|  import sys
 3|  import RPi.GPIO as GPIO
 4|  from time import sleep
 5|  import random
 6|
 7|  # Define a class to handle displaying a number on the LED array.
 8|  class led_display:
 9|    # Define LED pins. There are a number of ways to do this
10|    # I am doing it this way because I want it to be readable, and meaningful.
11|    lp1 = 13 # pin for LED 1
12|    lp2 = 16
13|    lp3 = 19
14|    lp4 = 20
15|    lp5 = 26
16|    lp6 = 21
17|    lp = [lp1, lp2, lp3, lp4, lp5, lp6] # An array of all LED pins in LED order.
18|
19|    # Class constructor function.
20|    # Initialize essential settings.
21|    def __init__(self):
22|      GPIO.setmode(GPIO.BCM)
23|      for l in self.lp:
24|        GPIO.setup(l, GPIO.OUT)
25|
26|    # Class Destructor function.
27|    # Do things that should be done with stopping.
28|    def __del__(self):
29|      GPIO.cleanup()
30|      return
31|
32|    # Display function: Illuminate n LEDs based upon a number provided.
33|    def display(self, number):
34|      if number < 0:
35|        return False
36|
37|      for i in range(0, len(self.lp)):
38|        n = i + 1
39|        if n <= number:
40|          GPIO.output(self.lp[i], 1)
41|        else:
42|          GPIO.output(self.lp[i], 0)
43|
```

Listing 8.2.8

```
44|    # An animation to indicate something is happening.
45|    def interlude(self):
46|      # Clear all of the LEDs quickly.
47|      for l in self.lp:
48|        GPIO.output(l, 0)
49|      sleep(0.100)
50|      # Turn on each LED with a small pause between each.
51|      for l in self.lp:
52|        GPIO.output(l, 1)
53|        sleep(0.020)
54|      # Turn off each LED with a small pause between each.
55|      for l in self.lp:
56|        GPIO.output(l, 0)
57|        sleep(0.020)
58|
59|  button_pin = 25
60|
61|  # Upon Ctrl-C, exit the application.
62|  def signal_handler(sig, frame):
63|    global led_display
64|    del led_display
65|    sys.exit(0)
66|
67|  def roll_dice(channel):
68|    led_display.interlude()
69|    roll = random.randint(1, 6)
70|    led_display.display(roll)
71|    print("New roll of: {}.".format(roll))
72|
73|  # Start monitoring for, and responding to, a button press.
74|  def start(button_pin, led_display):
75|    GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
76|    # Monitor the button pin, and when it is pressed for 100ms, call the
    function roll_dice.
77|    GPIO.add_event_detect(button_pin, GPIO.FALLING, callback=roll_dice,
    bouncetime=100)
78|    signal.signal(signal.SIGINT, signal_handler)
79|    signal.pause()
80|
81|  led_display = led_display()
82|  start(button_pin, led_display)
```

Listing 8.2.9

This program, unlike the rest we've done so far, will not end until you stop it manually. While it is running, you can press the button on the bread board, and then view the numerical output.

8.2.8 Exploration

The `RPi.GPIO` library offers two different ways to refer to the pins that you want to use. The `GPIO.BCM` mode that is used in our example program on line 22 sets up the program to use the pins as they are known by the Broadcom chip on the Raspberry Pi. The other option, `GPIO.BOARD` allows you to refer to the pins by their header number. In a later experiment, we will make use of some more advanced features that require the `GPIO.BCM` mode. For that reason, we are using it here.

Next, on line 17, you'll observe that we created a list that contains all 6 pin numbers attached to our LEDs. On line 23, we use a for loop to iterate over that list, and call `GPIO.setup([pin number], GPIO.OUT)`. This function configures the pin to operate in output mode. When your intent is to control something with the pin, output mode is what you want. However, if you're looking to read a value, or detect a button press,

input mode is needed.

On line 75, we use the `GPIO.setup()` function again, but for the button. In this case we use the input mode, and add a parameter that also configures a pull-up resistor. If you're not familiar with it, a pull-up resistor is a high value resistor connected to a voltage source and causes the signal line to rest at the high value instead of floating when not used. The alternative is a pull-down resistor which is just a high value resistor connected to GND and causes the line to stay at a low value when not used.

There are a number of possible approaches to determining when the button is pressed, but the most reliable way will generally always be one that relies upon hardware interrupts. On line 77, we make use of the `GPIO.add_event_detect()` function to monitor the button pin for a transition from high to low. When that happens, the function `roll_dice` will be called.

Upon being called, the `roll_dice` function will roll a random number, and display the number on the LED display we've created.

8.3 Troubleshooting

8.3.1 Switch

Double check the orientation of the switch. Pins 1 and 2 are connected to each other. Pins 3 and 4 are connected to each other. If the switch is oriented 90° in the wrong direction, you'll be short circuiting the switch, and pressing it will no produce any results.

8.3.2 LEDs

Double check the orientation of the LED. Because of their diode characteristics, having them reversed will prevent current from flowing through them, at the voltages we are using.

Checkpoint 8.3.1 Reflections.

(a) If we wanted to add a second button, which pin would be a good candidate? What code would be used to configure the pin to read the button state?

(b) If we wanted to add another LED, which pin would make a good candidate? What code would be used to configure the pin to drive the LED?

(c) Write a program that uses the LEDs as outputs that mimick a ring counter. Print out the position of the active output. Include a screenshot that captures the code as well as the output.

Chapter 9

Interfacing with increased versatility, and complexity.

Basic inputs and outputs of the digital or analog variety that make use of a single pin are very useful, but encounter a problem when we starting looking to work with more complicated systems.

This is essentially the reason for all networking protocols. There is a need for a structured communication between independent systems that allow for the exchange of much more complex information.

In the embedded world, there are several protocols that are designed for communication between independent systems. Of these, I^2C and SPI are very common protocols.

We will explore the basic underlying concepts, and use one or both to interface with a sensor. We will also continue our exploration of Python by making use of some Python data structures for managing data.

9.1 Associated Reading

- Think Python: How to Think Like a Computer Scientist. 2nd Edition[32]
  - Chapter 10: Lists
  - Chapter 12: Tuples
- Understanding the I^2C Bus[33]
- Inter-Integrated Circuit[34]

9.2 Connecting Using Data Exchange Protocols

9.2.1 Discussion

This lab builds upon the last by adding some components and making changes to the code.

When it comes to interfacing with external micro-controllers it is necessary to communicate with very specific sequences 1s and 0s. The methods for accomplishing this are defined by the communication protocol, and the micro controller itself. This data can be found in the data sheet for said micro controller.

It may eventually be necessary for you to write your own code for interfacing with such a device, but right now we are going to gloss over the tedious details of doing so, and rely upon the work of other skilled people. We will import libraries for an OLED display as well as an AHT20 Temperature and Humidity sensor. These libraries take care of sending and receiving the bytes of data that are necessary to make these useful to us. We are going to focus on the process of wiring up these new sensors on the I^2C bus, and making use of them for our own purposes.

[32] https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist
[33] https://www.ti.com/lit/an/slva704/slva704.pdf
[34] https://www.egr.msu.edu/classes/ece480/capstone/fall09/group03/AN_hemmanur.pdf

9.2.2 Required Equipment

These are the required components.

Table 9.2.1

Amount	Part Type
1	Breadboard
1	128x32 I2C Monochrome OLED Display
1	AHT20 Temperature and Humidity Sensor
1	100nF Capacitor
6	Red LED
6	470Ω Resistor
1	10KΩ Resistor
1	Raspberry Pi 3
1	Push button
Handful	Male to Female Jumper Wires
Handful	Solid Core Jumper Wires

9.2.3 Preview

We have two objectives. The first is to make use of the OLED display to display the roll results for our dice program. The second is to read the temperature and humidity from our AHT20 sensor, and then display that data on the OLED display.

This is what we are aiming for:

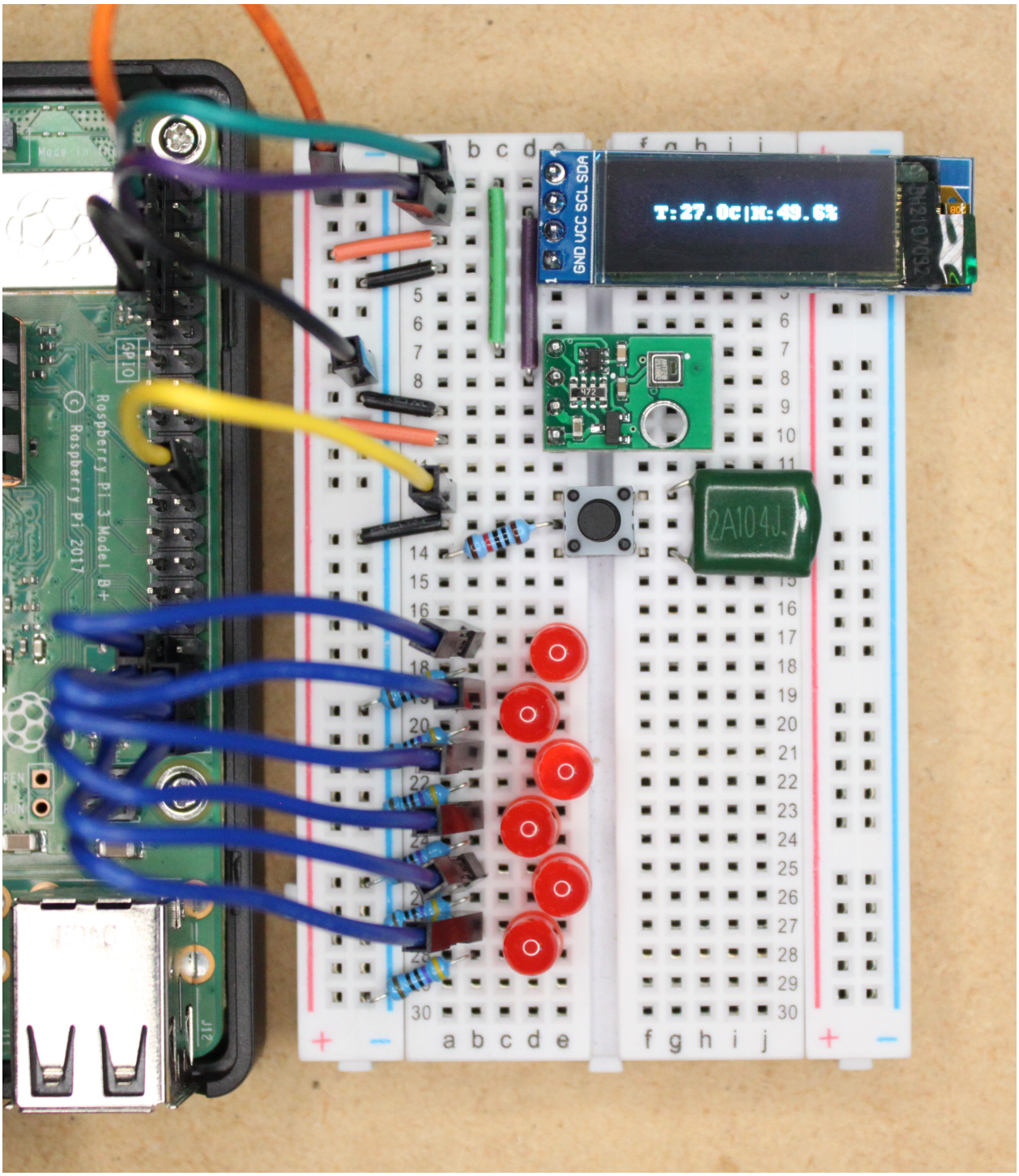

Figure 9.2.2 The Completed Circuit

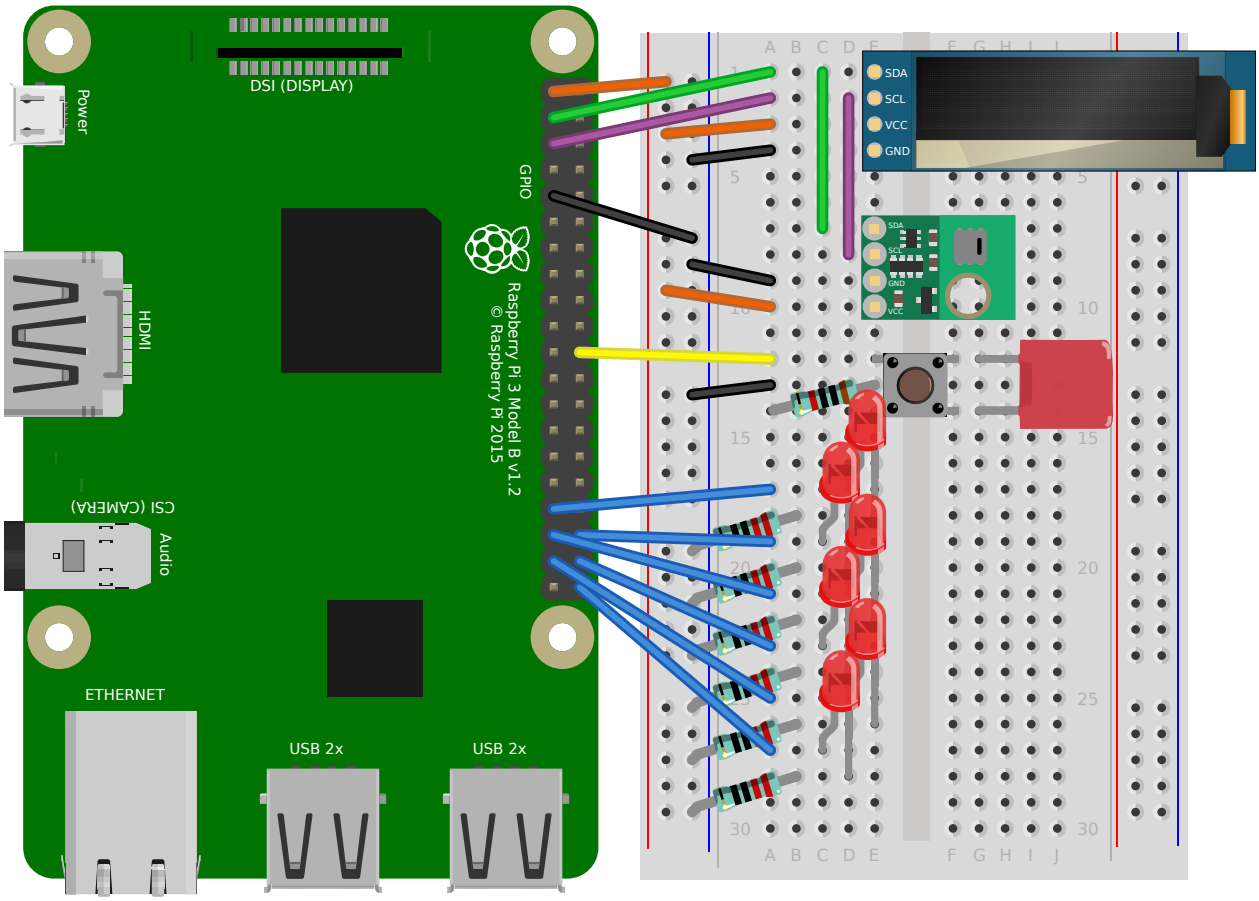

Figure 9.2.3 The Completed Circuit

9.2.4 Wiring

In order to match the code that will follow, special care will be needed with matching the correct GPIO to the proper components.

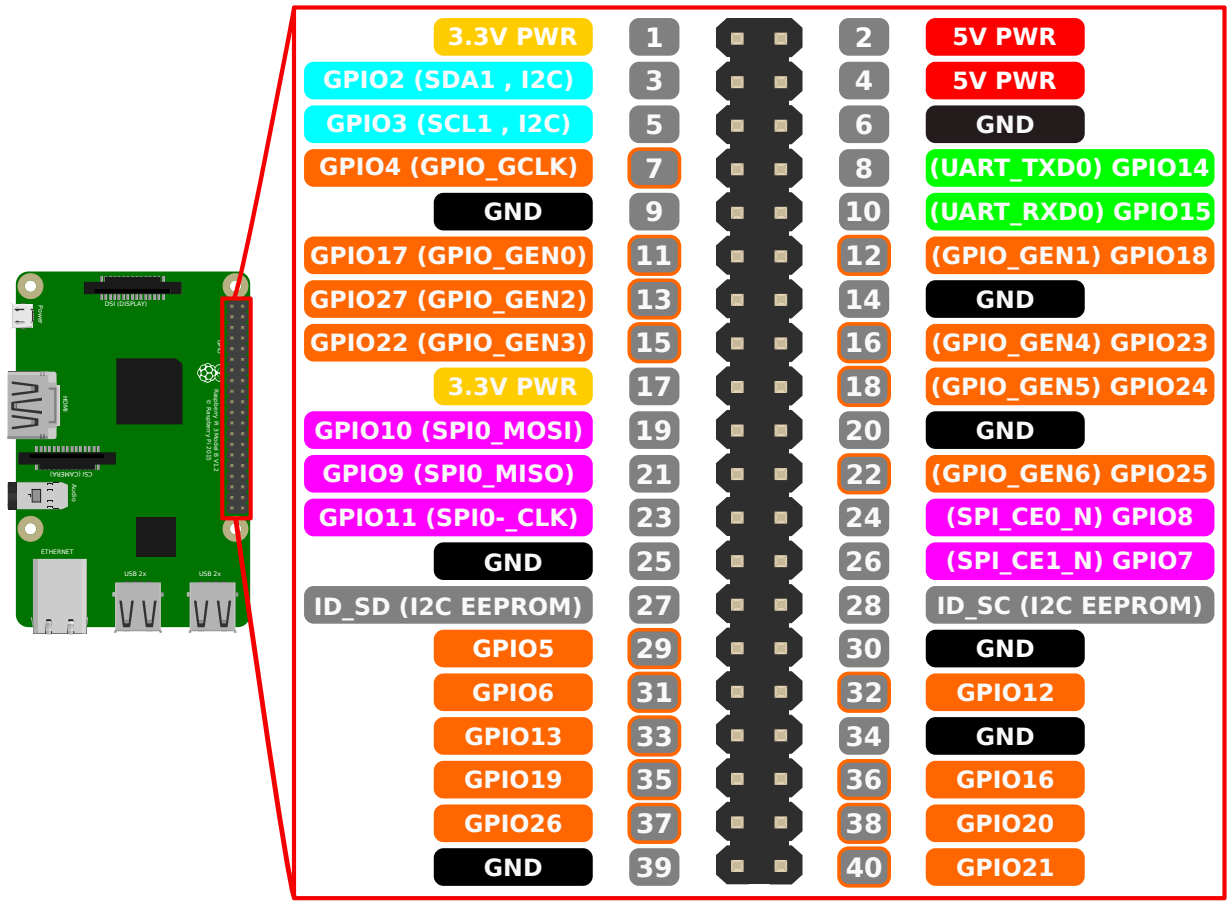

Figure 9.2.4 Raspberry Pi 3 GPIO Pinout

Table 9.2.5

GPIO Designation	Header Pin	Component
GPIO2/SDA1	Pin 3	SDA (OLED/AHT20)
GPIO3/SCL1	Pin 5	SCL (OLED/AHT20)
3.3V	1	VCC/VIN (OLED/AHT20)
GND	Pin 9	GND (Shared)
GPIO25	Pin 22	Switch
GPIO13	Pin 33	LED 1
GPIO16	Pin 36	LED 2
GPIO19	Pin 35	LED 3
GPIO20	Pin 38	LED 4
GPIO26	Pin 37	LED 5
GPIO21	Pin 40	LED 6

Using the schematics, and details provided, wire up the components to the Raspberry Pi. When done, boot it up, and proceed to the next section.

9.2.5 Code

With everything wired up, transcribe the following program and run it.

The majority of this code is a replication of the previous experiment. This includes some new library imports, a new class definition, and a couple of lines in the roll_dice function.

Download the Source[35]

```
 1|  import signal
 2|  import sys
 3|  import RPi.GPIO as GPIO
 4|  from time import sleep
 5|  import random
 6|  import board
 7|  from PIL import Image, ImageDraw, ImageFont
 8|  import adafruit_ssd1306
 9|
10|  # Define a class to handle displaying a number on the LED array.
11|  class led_display:
12|    # Define LED pins. There are a number of ways to do this
13|    # I am doing it this way because I want it to be readable, and meaningful.
14|    lp1 = 13 # pin for LED 1
15|    lp2 = 16
16|    lp3 = 19
17|    lp4 = 20
18|    lp5 = 26
19|    lp6 = 21
20|    lp = [lp1, lp2, lp3, lp4, lp5, lp6] # An array of all LED pins in LED order.
21|
22|    # Class constructor function.
23|    # Initialize essential settings.
24|    def __init__(self):
25|      GPIO.setmode(GPIO.BCM)
26|      for l in self.lp:
27|        GPIO.setup(l, GPIO.OUT)
28|
29|    # Class Destructor function.
30|    # Do things that should be done with stopping.
31|    def __del__(self):
32|      GPIO.cleanup()
33|      return
34|
35|    # Display function: Illuminate n LEDs based upon a number provided.
36|    def display(self, number):
37|      if number < 0:
38|        return False
39|
40|      for i in range(0, len(self.lp)):
41|        n = i + 1
42|        if n <= number:
43|          GPIO.output(self.lp[i], 1)
44|        else:
45|          GPIO.output(self.lp[i], 0)
46|
```

Listing 9.2.6

[35]https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/interfacing_bus_protocols/dice.py

```
47|    # An animation to indicate something is happening.
48|    def interlude(self):
49|      # Clear all of the LEDs quickly.
50|      for l in self.lp:
51|        GPIO.output(l, 0)
52|      sleep(0.100)
53|      # Turn on each LED with a small pause between each.
54|      for l in self.lp:
55|        GPIO.output(l, 1)
56|        sleep(0.020)
57|      # Turn off each LED with a small pause between each.
58|      for l in self.lp:
59|        GPIO.output(l, 0)
60|        sleep(0.020)
61|
62|  class oled_display():
63|    # Dimensions for OLED Display.
64|    width = 128
65|    height = 32
66|    border = 0
67|
68|    i2c = None
69|    oled = None
70|
71|    def __init__(self):
72|      # Initialize the I1iC bus.
73|      self.i2c = board.I2C()
74|      self.oled = adafruit_ssd1306.SSD1306_I2C(self.width, self.height,
    self.i2c, addr=0x3C)
75|
76|    def display_none(self):
77|      # Reset Display/Clear
78|      self.oled.fill(0)
79|      self.oled.show()
80|
```

Listing 9.2.7

```
  81|    def display_string(self, text):
  82|      # Reset display.
  83|      self.display_none()
  84|
  85|      # Create blank image with 0-bit color.
  86|      image = Image.new("1", (self.oled.width, self.oled.height))
  87|
  88|      # Get drawing object to draw on image.
  89|      draw = ImageDraw.Draw(image)
  90|
  91|      if self.border > 0:
  92|        # Draw a white background
  93|        draw.rectangle((0, 0, self.oled.width, self.oled.height), outline=255,
      fill=255)
  94|
  95|        # Draw a smaller inner rectangle
  96|        draw.rectangle(
  97|          (border, border, self.oled.width - self.border - 1, self.oled.height
      - self.border - 1),
  98|          outline=0,
  99|          fill=0,
 100|          )
 101|
 102|      # Load default font.
 103|      font = ImageFont.truetype('DejaVuSansMono.ttf', 16)
 104|
 105|      # Draw Some Text
 106|      (font_width, font_height) = font.getsize(text)
 107|      draw.text(
 108|        (self.oled.width // 2 - font_width // 2, self.oled.height // 2 -
     font_height // 2),
 109|        text,
 110|        font=font,
 111|        fill=10,
 112|        )
 113|
 114|      # Display image
 115|      self.oled.image(image)
 116|      self.oled.show()
 117|
```

Listing 9.2.8

```
118|  button_pin = 25
119|
120|  # Upon Ctrl-C, exit the application
121|  def signal_handler(sig, frame):
122|    global led_display
123|    del led_display
124|    sys.exit(0)
125|
126|  def roll_dice(channel):
127|    led_display.interlude()
128|    roll = random.randint(1, 6)
129|    text = "Rolled a {}.".format(roll)
130|    oled_display.display_string(text)
131|    led_display.display(roll)
132|    print("New roll of: {}.".format(roll))
133|
134|  # Start monitoring for, and responding to, a button press.
135|  def start(button_pin, led_display, oled_display):
136|    GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
137|    # Monitor the button pin, and when it is pressed for 100ms, call the
    function roll_dice.
138|    GPIO.add_event_detect(button_pin, GPIO.FALLING, callback=roll_dice,
    bouncetime=100)
139|    signal.signal(signal.SIGINT, signal_handler)
140|    signal.pause()
141|
142|  led_display = led_display()
143|  oled_display = oled_display()
144|  start(button_pin, led_display, oled_display)
```

Listing 9.2.9

9.2.6 Exploration

Lines 6, 7, and 8 import libraries needed for interfacing with the OLED display. `board` provides the I^2C protocol support, `PIL` provides libraries for creating images that will be sent to the OLED display, and `adafruit_ssd1306` provides the handlers for sending data to the OLED display over I^2C.

Next, we declare a class that will handle sending our roll results to the display. The main function is `display_string` which will take a preformatted sentence, incorporate it into an image, and send that image to the OLED display.

Classes are very important structures, if you continue on with programming, you'll explore them more. If you're interested, chapters 15 through 19 of Think Python: How to Think Like a Computer Scientist. 2nd Edition[36] cover the subject matter in more detail.

In the `roll_dice` function we've added two lines. The first formats a string of text that includes our rolled number. The next passes that text to the `display_string` function discussed previously.

Finally, to interact with the OLED display, we've modified the `start` function to allow an `oled_display` object to be passed to it, and we've initialized a variable that points to an instance of our `oled_display` class we defined earlier.

With this itteration of our project, when the button is pressed, the output is displayed on the LED display as before, but also on the OLED display in plain text.

[36] `https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist`

9.3 Reading and Using Data

9.3.1 Discussion

Outputting data is very handy, but being able to retrieve data from sensors is probably one of the greatest benefits of interfacing with peripherials. We will explore that with a second example.

With everything wired up, transcribe the following program, and run it.

Download the Source[37]

9.3.2 Code

```
 1|  import board
 2|  from PIL import Image, ImageDraw, ImageFont
 3|  import adafruit_ssd1306
 4|  import time
 5|  import adafruit_ahtx0
 6|
 7|  # Dimensions for OLED Display.
 8|  WIDTH = 128
 9|  HEIGHT = 32
10|  BORDER = 0
11|
12|  # Use for I2C.
13|  i2c = board.I2C()
14|  oled = adafruit_ssd1306.SSD1306_I2C(WIDTH, HEIGHT, i2c, addr=0x3C)
15|  sensor = adafruit_ahtx0.AHTx0(i2c)
16|
17|  def display_none():
18|    # Reset Display/Clear
19|    oled.fill(0)
20|    oled.show()
21|
```

Listing 9.3.1

[37] https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/interfacing_bus_protocols/temperature.py

```
22|  def display_temp():
23|    # Create blank image with 1-bit color.
24|    image = Image.new("1", (oled.width, oled.height))
25|
26|    # Get drawing object to draw on image.
27|    draw = ImageDraw.Draw(image)
28|
29|    # Draw a white background
30|    draw.rectangle((0, 0, oled.width, oled.height), outline=255, fill=255)
31|
32|    # Draw a smaller inner rectangle
33|    draw.rectangle(
34|      (BORDER, BORDER, oled.width - BORDER - 1, oled.height - BORDER - 1),
35|      outline=0,
36|      fill=0,
37|      )
38|
39|    # Read the temperature.
40|    temp = sensor.temperature
41|    # Read the humidity.
42|    hum = sensor.relative_humidity
43|
44|    text = " T:{:.1f}C|H:{:.1f}% ".format(temp, hum)
45|
46|    # Load default font.
47|    font = ImageFont.load_default()
48|
49|    # Draw Some Text
50|    (font_width, font_height) = font.getsize(text)
51|    draw.text(
52|      (oled.width // 2 - font_width // 2, oled.height // 2 - font_height // 2),
53|      text,
54|      font=font,
55|      fill=255,
56|      )
57|
58|    # Display image
59|    oled.image(image)
60|    oled.show()
61|
62|  while(True):
63|    display_temp()
64|    time.sleep(1)
```

Listing 9.3.2

9.3.3 Exploration

This little program re-uses the class we created for the OLED display in the last experiment.

On line 5 we import a new library that will allow us to read from the AHT20 sensor.

On line 62 we establish a continuous loop that reads the temperature, and relative humidity from the sensor, then formats the values in a text string, and sends it to the OLED display.

Checkpoint 9.3.3 Reflections.

(a) How many I^2C devices can share a bus?

(b) What are some advantages to using bus protocols over direct GPIO manipulation like what we used for the button and LEDs?

(c) Searching online, can you find an I^2C or SPI peripherial that you'd be interested in using in a person electronics project? Pick one, and determine if there is a pre-existing python, or C based library for it.

(d) Modify the code provided in Subsection 9.3.2 so that in displays the temperature in Fahrenheit. Print the results. Include a screenshot that captures the code and the output.

Chapter 10

Data Management

In this age of Internet of Things, embedded systems have come to rely heavily upon the exchange of data.

Handling this sort of exchange still presents challenges, but thanks to high-level programming languages, it is accessible to even the beginning programmer.

We will explore reading from, and writing to files that are local to our device, as well as making requests to the internet to fetch, or submit data.

While doing so, we will continue to explore data structures that make doing so more manageable.

10.1 Associated Reading

- Think Python: How to Think Like a Computer Scientist. 2nd Edition[38]
  - Chapter 11: Dictionaries
  - Chapter 14: Files

10.2 Connecting to the Internet

10.2.1 Discussion

The complexity of an embedded system grows dramatically when you begin to integrate communication and human interaction. However, it is an important part of the modern infrastructure, and so it is better to get a basic understanding of how such a thing is achieved.

10.2.2 Code

Transcribe the program, and run it.

Download the Source[39]

---

[38] `https://open.umn.edu/opentextbooks/textbooks/think-python-how-to-think-like-a-computer-scientist`

[39] `https://raw.githubusercontent.com/ajbradburn/IntroToEmbeddedSystemsLabExercises/master/data_management/weather.py`

```
 1|  import math
 2|  import requests
 3|  import json
 4|  import board
 5|  from PIL import Image, ImageDraw, ImageFont
 6|  import adafruit_ssd1306
 7|  from time import sleep
 8|  from datetime import datetime
 9|  import adafruit_ahtx0
10|  import threading
11|
```

Listing 10.2.1

```
12|  class nws_forecast():
13|    # Get the National Weather Service Forecast based upon the Geolocation.
14|    # Geolocation is found by using the public IP address for the network that
    this device is attached to.
15|    def get(self):
16|      # Get the geolocation for your IP Address.
17|      # API Documentation: https://ip-api.com/docs/api:json
18|      URL = 'http://ip-api.com/json'
19|
20|      r = requests.get(url = URL)
21|      data = r.json()
22|      # For debuggin purposes.
23|      #print(json.dumps(data, indent=2))
24|      latitude = data['lat']
25|      longitude = data['lon']
26|
27|      # Get the National Weather Service Office, and grid for your location.
28|      # API Documentation:
    https://www.weather.gov/documentation/services-web-api#
29|      HEADERS = {'user-agent': '(IntroductionToEmbeddedSystems,
    ajbradburn@gmail.com)'}
30|
31|      URL = 'https://api.weather.gov/points/{},{}'
32|      URL = URL.format(latitude, longitude)
33|
34|      r = requests.get(url = URL, headers=HEADERS)
35|
36|      data = r.json()
37|      # For debugging purposes.
38|      #print(json.dumps(data, indent=2))
39|
40|      # Using the data from the last request, get the forcast for your area.
41|      office = data['properties']['gridId']
42|      loc_x = data['properties']['gridX']
43|      loc_y = data['properties']['gridY']
44|
45|      URL = 'https://api.weather.gov/gridpoints/{}/{},{}/forecast'
46|      URL = URL.format(office, loc_x, loc_y)
47|
48|      r = requests.get(url = URL, headers=HEADERS)
49|
50|      data = r.json()
51|      # For debugging purposes.
52|      #print(json.dumps(data, indent=2))
53|
54|      # Return only the current, or most immediate forecast.
55|      return data['properties']['periods'][0]
56|
```

Listing 10.2.2

```
57|  class oled_display():
58|    width = 128
59|    height = 32
60|
61|    oled = None
62|
63|    def __init__(self):
64|      self.oled = adafruit_ssd1306.SSD1306_I2C(self.width, self.height,
    board.I2C(), addr=0x3C)
65|
66|    def display_none(self):
67|      self.oled.fill(0)
68|      self.oled.show()
69|
70|    def display_string(self, text, font = None, size = 10, height = None, width
    = None):
71|      image = Image.new("1", (self.oled.width, self.oled.height))
72|
73|      draw = ImageDraw.Draw(image)
74|
75|      if height == None or width == None:
76|        dimensions = self.text_size(text, font, size)
77|
78|        if width == None:
79|          width = dimensions[0]
80|
81|        if height == None:
82|          height = dimensions[1]
83|
84|      if font == None:
85|        font = ImageFont.load_default()
86|      else:
87|        font = ImageFont.truetype(font, size)
88|
89|      draw.text(
90|        (self.oled.width // 2 - width // 2, self.oled.height // 2 - height // 2),
91|        text,
92|        font=font,
93|        fill=10,
94|        )
95|
96|      self.oled.image(image)
97|      self.oled.show()
98|
```

Listing 10.2.3

```
 99|    def text_size(self, text, font = None, size = 10):
100|      if font == None:
101|        font = ImageFont.load_default()
102|      else:
103|        font = ImageFont.truetype(font, size)
104|      return font.getsize(text)
105|
106|    def scroll_string(self, text, font = None, size = 10, delay = 0.01):
107|      text_len = len(text)
108|      dimensions = self.text_size(text, font, size)
109|      window_length = math.floor(self.oled.width / (dimensions[0] / text_len))
110|      start = 0
111|      height = dimensions[1]
112|      width = window_length * (dimensions[0] / text_len)
113|      padding = ' ' * window_length
114|      text = padding + text + padding
115|      text_len = len(text)
116|      while start + window_length <= text_len:
117|        sub_string = text[start:start + window_length]
118|        start = start + 1
119|        self.display_string(sub_string, font, size, width=width, height=height)
120|        sleep(delay)
121|      return
122|
123|    def display_with_intro(self, intro, text, font = None, size = 10):
124|      self.display_string(intro, font, size)
125|      sleep(1)
126|      self.scroll_string(text, font, size)
127|
```

Listing 10.2.4

```
128|  default_font = 'DejaVuSansMono.ttf'
129|
130|  # Initialize objects.
131|  oled_display = oled_display()
132|  sensor = adafruit_ahtx0.AHTx0(board.I2C())
133|
134|  def start_log_timer():
135|    log_timer = threading.Timer(30, log_measurements).start()
136|
137|  def start_forecast_timer():
138|    forecast_timer = threading.Timer(60, get_weather).start()
139|
140|  def start_temp_timer():
141|    global temp_timer
142|    temp_timer = threading.Timer(10, display_temp)
143|    temp_timer.start()
144|
145|  def get_weather():
146|    # Start a new timer.
147|    start_forecast_timer()
148|
149|    # Cancel the temp_timer so that it doesn't interfere with our forecast
    display.
150|    global temp_timer
151|    temp_timer.cancel()
152|
153|    # Get the forecast and display it.
154|    global oled_display
155|    global sensor
156|    weather = nws_forecast()
157|    forecast = weather.get()
158|    oled_display.display_string('High: {}°F'.format(forecast['temperature']),
    default_font, 16)
159|    sleep(2)
160|    oled_display.display_string('Inside: {}°F'.format(int(sensor.temperature *
    9 / 5 + 32)), default_font, 16)
161|    sleep(2)
162|    oled_display.display_with_intro('Forecast:', forecast['detailedForecast'],
    default_font, 16)
163|
164|    # Resume the display of temperature, and start a new temp_timer.
165|    display_temp()
166|
```

Listing 10.2.5

```
167|  def display_temp():
168|    # Start a new timer.
169|    start_temp_timer()
170|
171|    # Read the temperature and display it.
172|    global default_font
173|    temperature = sensor.temperature
174|    humidity = sensor.relative_humidity
175|    oled_display.display_string('Inside: {:.2f}°F'.format(temperature * 9 / 5 +
    32), default_font, 12)
176|
177|  def log_measurements():
178|    # Start a new timer.
179|    start_log_timer()
180|
181|    # Read temperature and humidity. Log it with the date/time to a file in csv
    format.
182|    global sensor
183|    temperature = sensor.temperature
184|    humidity = sensor.relative_humidity
185|    string = '\"{}\",{:.2f},{:.2f}\n'.format(datetime.now().strftime("%d/%m/%Y
    %H:%M:%S"), temperature, humidity)
186|    file = open('log.csv', 'a')
187|    file.write(string)
188|    file.close()
189|
190|  # A variable to hold the temp_timer so that we can reference it later, and
    cancel the timer when needed.
191|  temp_timer = None
192|
193|  start_log_timer()
194|  display_temp()
195|  start_forecast_timer()
```

Listing 10.2.6

10.2.3 Exploration

10.2.3.1

There are a myriad ways to communicate over the internet. One of the more common methods is through what is called a REST API. Starting on line 12 we create a class with a single method that will get the forecasts for our area from the national weather service through a series of HTTP requests to different APIs.

The first request is to a IP geolocation service as identified in the URL on line 18. This service will take the public IP address that is making the request, and find the best estimate of the geographic location in terms of latitude and longitude. It will return this data in the form of a JSON object. If you uncomment line 23, and run the program, you can see what this response looks like.

When we get the geolocation response, we extract the latitude and longitude as seen on lines 24 and 25. We then use this information to make another request to the National Weather Service (as seen on line 31) so that we can get the office that is responsible for providing the forecasts, as well as the grid coordinates for our location. This request will return another JSON formatted data object that will contain the information we need. If you uncomment line 38 and run the application, you can see what the response looks like.

Next, we extract the office id, and grid coordinates as seen on lines 41, 42 and 43. We then use this data to make our final request as seen on line 45. The response for this will contain the data for a 7 day forecast. However, since we are only interested in the immediate forecast, we extract the first forecast in the list (at index 0) as seen on line 55, and return that to the requester.

10.2.3.2

The objective of this application is to display the inside temp, log the inside temp and humidity to a file, and also occasionally show the forecast high temperature along with the forecast. Measuring and displaying the inside temperature occurs every 10 seconds, except when the forecast is being displayed. The logging is done every 30 seconds. The display of the forecast is done every minute. In order to accurately control the timing of all of these functions we make extensive use of timers, and the functions on lines 134, 137, and 145 help us with that.

The threading library provides the functions for declaring timers. The declaration includes the function name that you want to be called, and the number of seconds until the timer executes the function call. Each timer is executed only once, so when the functions are called, we create a new timer so that the process is repeated.

When the `get_weather` function is called, it cancels the `temp_timer` so that the display of the forecast isn't interrupted. This is only necessary because both of them make use of the OLED display. The `log_timer` isn't canceled because there is no conflict in the resources used.

With the `log_measurements` function, the inside temperature and humidity are combined with the current date and time into a CSV (Comma Separated Values) string, and appended to a file. This is a common data exchange format, and it can be read by a number of different applications. A common one is Microsoft Excel. If you open this file in Excel, or LibreOffice Calc you can easily use it to create a graph that will show the changes in temperature and humidity in your immediate environment over time.

Checkpoint 10.2.7 Reflections.

(a) With access to the internet, what is a way that your project might be able to provide you with updates? What are some ways that you normally get updates? Could those same methods be used by a project to communicate with you? Is there a Python library to make it easier?

(b) Can the process of communication with your project be reversed? Can your project get updates from you remotely? Would the method you identified previously work?

(c) Modify the code for this project so that it records the temperature in Fahrenheit as well as celcius. This should add another column to the output data.

(d) Run the program for an hour, or overnight. Then, using Google Sheets, Microsoft Excel, Open Office Calc, or some other application graph the results. Include a screenshot of the results.

Appendix A

References

References

Materials for which this lab manual references, or relies upon.

[**1**] *Raspberry Pi Assembly*, Ibanez, R. F., (2018)
URL: https://personal.utdallas.edu/~pervin/RPiA/RPiA.pdf

[**2**] *ARM Programmer's Guide for ARMv8-A* URL: https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set

[**3**] Downey, Allen B. *Think Python 2nd Edition* URL: https://greenteapress.com/wp/think-python-2e/

[**4**] *Understanding the I^2C Bus* URL: https://www.ti.com/lit/an/slva704/slva704.pdf

[**5**] *Inter-Integrated Circuit* URL: https://www.egr.msu.edu/classes/ece480/capstone/fall09/group03/AN_hemmanur.pdf

Appendix B

Sourcing Suggestions

B.1 Raspberry Pi

RaspberryPi.com Hardware[40]

Note: For the purposes of this class, the Raspberry Pi 3 Model B+, or Raspberry Pi 4 Model B are recommended. The Raspberry Pi 400 will work, but you'll have to check the documentation for the pinout when it comes to integrating external circuitry.

While the other, older models of the Raspberry Pi∗ will technically work for the subject matter in this class, there will be additional challenges, and for the sake of your sanity, they are not recommended.

The Raspberry Pi Pico will not work for this class.

B.2 Sensors and Misc Components

As time goes by, the availability of these components may vary. I will attempt to remedy this by including links to online searches, rather than specific vendor listings.

B.2.1 OLED Display

The important part is the SSD1306 with I2C. The dimensions of the OLED display are less important.

Keywords: 128x32 I2C SSD1306 OLED Display

128x32 SSD1306 I^2C OLED Display @ Amazon[41]

128x32 SSD1306 I^2C OLED Display @ ebay[42]

128x32 SSD1306 I^2C OLED Display @ Adafruit[43]

B.2.2 AHT20 Temperature and Humidity Sensor

Keywords: AHT20 I2C

AHT20 I^2C Sensor Breakout @ Amazon[44]

AHT20 I^2C Sensor Breakout @ ebay[45]

AHT20 I^2C Sensor Breakout @ Adafruit[46]

[40] https://www.raspberrypi.com/products/

[41] https://www.amazon.com/s?k=128x32+SSD1306+i2c+oled+display+-128x64&i=electronics&crid=43Y2OYEWS9MQ&sprefix=128x32+ssd1306+i2c+oled+display+-128x64%2Celectronics%2C74&ref=nb_sb_noss

[42] https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2380057.m570.l1313&_nkw=128x32+SSD1306+i2c+oled+display&_sacat=0

[43] https://www.adafruit.com/product/931

[44] https://www.amazon.com/s?k=AHT20+i2c&crid=127F2S6VISW7H&sprefix=aht20+i2c+%2Caps%2C70&ref=nb_sb_noss

[45] https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2380057.m570.l1313&_nkw=AHT20+I2C&_sacat=0

[46] https://www.adafruit.com/product/4566?gclid=Cj0KCQjw3IqSBhCoARIsAMBkTb0juJQMoREL8sAM-v1LLEulkdTQxBiNlCdbKiyI_eVOksU9dTm2LO8aApqtEALw_wcB

B.2.3 LEDs, Resistors, Etc.

It would seem, at the time of writing, that Amazon is saturated with electronics kits that contain plenty of LEDs, resistors, switches and hookup wires for these labs. I am looking at one that is under $20.00, and has everything needed.

Keywords: electronics kit

Required Components

Table B.2.1

Amount	Part Type
1	Breadboard
1	100nF Capacitor
6	Red LED
6	470Ω Resistor
1	10KΩ Resistor
1	Push button
Handful	Male to Female Jumper Wires
Handful	Solid Core Jumper Wires

General Purpose Electronics Kit @ Amazon[47]

General Purpose Electronics Kit @ ebay[48]

[47] https://www.amazon.com/s?k=electronics+kit&crid=P1BONNXKBI7B&sprefix=electronics+kit%2Caps%2C90&ref=nb_sb_noss

[48] https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2380057.m570.l1313&_nkw=electronics+kit&_sacat=0

Appendix C

Using Pretext

C.1 Editing and Building Custom Versions

I used PreTeXt to create and export this work.

The project page outlines everything you need to know to get going.

PreTeXt Project Homepage[49]

C.2 Using PreTeXt CLI

I did most of my work with the PreTeXt CLI.

The GitHub page for the project outlines the steps of installing, and using PreTeXt CLI.

PreTeXt CLI GitHub Page[50]

Once it is installed, and you've got a copy of this source code, you can build your own copies with the following command.

```
pretext build pdf -i source/IntroductionToEmbeddedSystemsLabManual.ptx
```

Listing C.2.1

[49] https://pretextbook.org/

[50] https://github.com/PreTeXtBook/pretext-cli

Colophon

This book was authored in PreTeXt.