Digital Design and Computer Architecture

Lab 6: C Programming

Introduction

In this lab, you will learn to program an ARM processor on the Raspberry Pi in C by following a tutorial and then writing several of your own programs.

At this point in your education, you have likely written a fair amount of code. Writing C code for microprocessors, you can apply the same algorithms and clever tricks to perform computational backflips that you can in any other language. The programs you will write in this lab are simple, but they are meant to show how easily your CS5, CS60, etc. concepts transfer to a portable processor like the Raspberry Pi. Suddenly your computations are no longer confined to your laptop—you can use the results to drive a motor, light up a display, build an automatic plant waterer… the possibilities are endless. This lab will give you a taste of embedded programming, introducing you to the power of microprocessors.

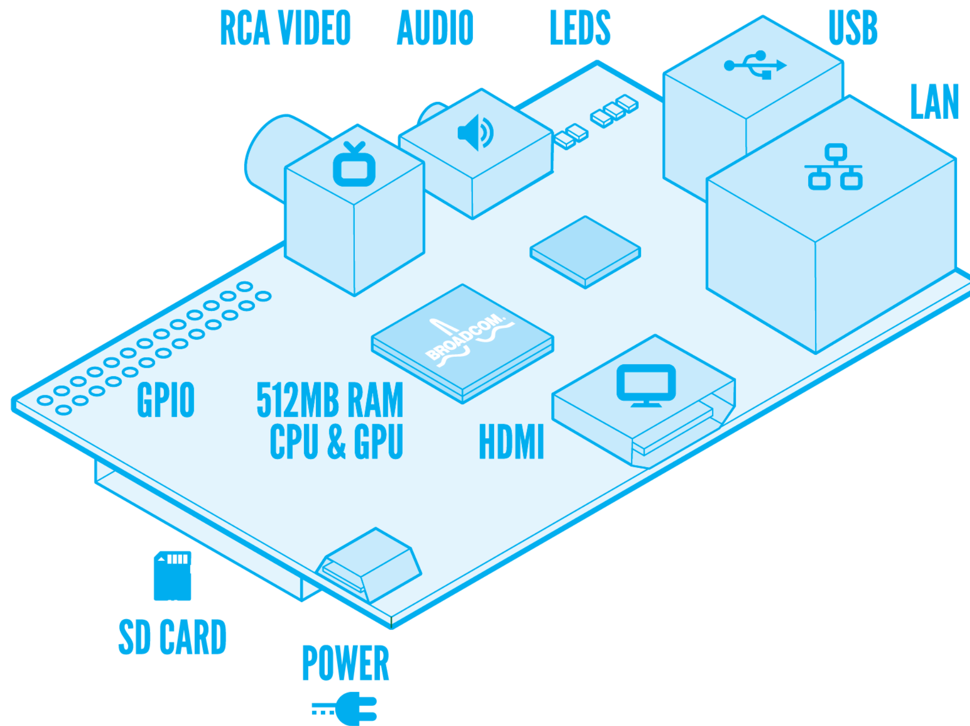
The Raspberry Pi

In this lab and the next two, you will be using the Raspberry Pi, a single-board computer developed by the Raspberry Pi Foundation and first released in early 2012. The Raspberry Pi contains a Broadcom BCM2835 system-on-a-chip (SoC) and is capable of running Linux with a correctly configured SD card. The processor is built with ARMv6 architecture, typically runs at 700MHz and has 512MB of RAM. The Raspberry Pi is available online for ~$35.

The Raspberry Pi is an extremely powerful, versatile tool. It was designed to inspire the next generation of programmers and hardware hackers by making cheap, accessible, programmable computers widely available. They have been adopted by the open-source hardware hacker community. To see all sorts of cool projects that people are doing, check out <http://hackaday.com/?s=raspberry+pi>.

Check out the "*Linux and Pi: Getting Acquainted with Linux and the Raspberry Pi*" guide on the class website for instructions on connecting to the Pi, and a brief introduction to using Linux.

The BCM2835 has many built-in peripheral functions. The Raspberry Pi board has hardware that takes advantage of the BCM2835's capabilities, including an HDMI connector, audio output, and USB ports.



(source: raspberrypi.org)

Note: This diagram is the Model B, but we will be using the Model B+

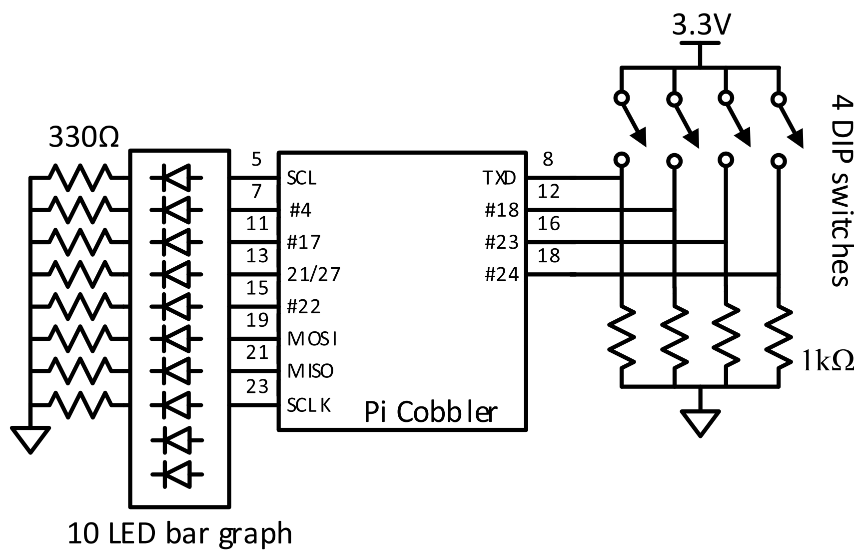
In addition to peripherals with specific functions, the Raspberry Pi board breaks out 17 pins for GPIO (general-purpose input/output). These pins can be configured for a variety of functions, but we will be using them to read inputs from switches and write outputs to LEDs. All of the pins operate at 3.3V levels, so **you should be very careful not to put 5V on the GPIOs**. Additionally, current from the GPIO pins is limited to 16mA per pin. The 3.3V pins can source a maximum of 50mA. The 5V pins draw current directly from the mini-USB power source, with a maximum 1A – board power consumption (typically 500-700mA) = 500-300mA.

For the tutorial and your first assigned program, you will be using the DIP switches and LEDs to interact with the processor, which you will need to connect to your Raspberry Pi. Pin numbering gets confusing. The actual processor, the BCM2835, has 53 possible PIO pins. On the Raspberry Pi board, most of these are used for the built-in peripherals, but 17 of them are routed to the GPIO header in the top left corner of the board. The pin numbers here are consistent with the BCM2835 numbers. Some of these are marked as having specific functions (such as the MOSI and MISO pins for SPI communication), but can still be used as general PIO pins. The C library you will be using to access the GPIO (called *wiringPi*) uses it's own pin numbering scheme that is listed in the table below. If you get confused, ask a lab proctor!

For outputs, you will be using light-emitting diodes (LEDs). To prevent breaking the LEDs or the Pi, you need to limit the amount of current going through the LED using resistors. The Pi outputs 3.3V, so you will add a 330Ω resistor between the LED and ground to limit the current to 10mA. For inputs, you will be using dual inline pin (DIP) switches. When the switch is ON, the pins on either side of the switch are electrically connected. Once again, you don't want a large amount of current, so you will add a 1kΩ resistor between each switch and ground.

Make sure to test that your circuit works like you expect it to before plugging in the Pi. Apply power and test your switches using a multimeter; test your LEDs by connecting the 3.3V power.

Make sure to connect your Pi ground to the breadboard ground, and your Pi 3.3V to your breadboard power bus. Be very careful that ground and power are never directly connected; there should always be a resistor in between.



WiringPi pin	BCM GPIO	Name on Pi Cobbler	Header		Name on Pi Cobbler	BCM GPIO	WiringPi pin
		3v3	1	2	5v0		
8	Rv1: 0, Rv2: 2	SDA	3	4	5v0		
9	Rv1: 1, Rv2: 3	SCL	5	6	GND		
7	4	#4	7	8	TXD	14	15
		GND	9	10	RXD	15	16
0	17	#17	11	12	#18	18	1
2	Rv1: 21, Rv2: 27	21/27	13	14	GND		
3	22	#22	15	16	#23	23	4
		3v3	17	18	#24	24	5
12	10	MOSI	19	20	GND		
13	9	MISO	21	22	#25	25	6
14	11	SCLK	23	24	CE0	8	10
		GND	25	26	CE1	7	11

C Compiler Tutorial

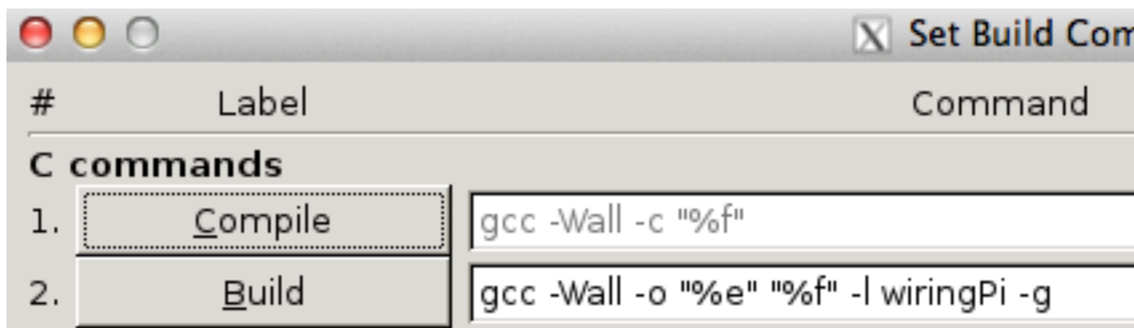
You will be using **ddd** to debug/test your programs, but to view and edit you will need to use a text editor such as **vim.tiny, leafpad,** or **geany**. To start any of these programs, simply type the name into the terminal when connected to the Raspberry Pi. By far, **geany** has the nicest interface, and will be easiest to use.

In this tutorial, you will learn to write and compile a simple program that uses the DIP switches, LEDs, and input and output from the terminal. You'll also learn to step through a program and debug it if you have errors.

On the Pi, create a directory called *lab6_xx*, where xx is your initials. Copy the *lab6tut.c* file from the class website into this directory. In the terminal, type **geany** to open the visual text editor. Click the "Open" button, navigate to your directory, and open the file. Click the "Compile" button. You should get a message like "Compilation finished successfully" in the compiler tab at the bottom of the window.

Now, try deleting a semicolon from a line and then click "Compile." You should get error messages. Fix the semicolon, save and compile once more.

Compiling will check for syntax errors. Click the "Build" button to link all of the libraries that your code depends on and create an executable. This should not work the first time you try it! It will tell you that there are undefined references to the *wiringPi* functions. Click the arrow next to the "Build" button and click "Set Build Commands." Type `–l wiringPi -g` at the end of the string next to Build, as shown:



The –l command (that is an L, not a #1) tells the program to include the functions in the listed non-standard library when compiling. The –g command is necessary for debugging. Click OK and build again.

When your build is successful, close the **geany** window.

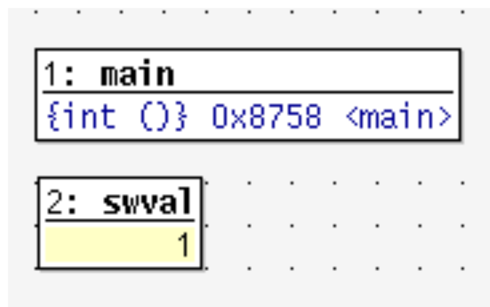
In the terminal, type **sudo ddd** to open the Data Display Debugger with root permissions (which are necessary to access the GPIO). This lightweight tool can be used to debug C and assembly code (which you will have the opportunity to try in Lab 8). You must compile and build your code before you can run it in **ddd**!

Click File->Open Program…, navigate to your compiled lab6tut file, and open it. If your code does not show up in the **ddd** window, go back and rebuild in **geany** with the `–g` flag.

The Command Tool window should be open, but if it isn't click View->Command Tool. Click on the "Run" button.

In the console at the bottom of the window, you should see the program print the number read from the switches. It will prompt you to enter a number. Type your number. The program will display the number from the DIP switches in the least significant nibble of the LEDs, and the number you entered in the most significant nibble. It will then pause for 3 seconds (3000 ms), then repeat. Try entering different numbers and changing the value on the DIP switches. Check that everything works as you expect. **Click "Interrupt" when you are done to stop the program.**

The next step is to learn to use **ddd** to trace through your program. Click Source->Display Line Numbers to turn on the line numbers, and right click on line 35 and place a breakpoint. Click the "Run" button in the Command Tool window and wait until the program stops at your breakpoint. Click on `swval` so that it is highlighted, and click the "Watch" button at the top of the window. You should see something like this:



Click "Cont" to run until you reach the breakpoint again, watching the `swval` change (depending on the state of your switches). Also watch the index variable `i`.

You can also step through the program one line at a time using "Step" or "Next." If there is a function, you can use "Stepi" or "Nexti" to step into the subroutine rather than moving on over it. Hover over any of the buttons in **ddd** to get a description of what they do.

Note that `swval` and `i` will disappear outside of the loop because they are out of scope.

To start the program again, click "Interrupt," then "Run."

Finally, learn to modify the program. You will have to close **ddd** and re-open **geany**. Comment out the line with the delay. Recompile and build your code. You can click the "Execute" button in **geany** to run your program, or re-open it in **ddd** to run it. Test it again and observe that there is no delay.

You now know your way around the **geany** text editor and **ddd** debugger. In the next sections, you can write some programs of your own.

Pocket Hypnotizer

You have been dispatched to the Atacama desert to obtain secret information from a rebel leader. You'll have to get past the border guards to reach your destination. For this mission, you need to build a pocket hypnotizer.

Create a new file called *lab6ph_xx.c* for your program. Remember to `#include <wiringPi.h>` in your file and to call `wiringPiSetup()` before running the rest of your program.

Write a program that causes a pattern on the LED bar to zip back and forth. When your program starts, it should turn on one of the LEDs. Then it should turn off that LED and turn on the next. Continue until reaching the end of the LED bar, then go back, then repeat indefinitely. You'll need to choose a suitable delay between steps to get the desired effect.

You can use any text editor to write your program, but you should use **ddd** to compile and test it on the Pi. If you have difficulties, step through your code and compare the values of the variables against your expectations. Tune the delay until it looks mesmerizingly good. Stare into the blinking lights as you repeat to yourself "I will ace E85."

Fibonacci Numbers

Your next goal is to calculate and print the first 16 Fibonacci numbers to the screen. Recall that each number in the Fibonacci series is the sum of the previous two numbers. Table 1 lists the first few numbers in the series.

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**…**
fib(n)	1	1	2	3	5	8	13	21	34	55	89	…

Table 1: Fibonacci Series

We can also define the fib function for negative values of n. To be consistent with the definition of the Fibonacci series, what would the following values be?

fib(0) = ____

fib(-1) = ____

These values are useful when writing a loop to compute fib(n) for all non-negative values of n.

Create a new file called lab6fib_xx.c and write your program. Compute and print the Fibonacci numbers for n = 1…16. Remember to `#include <stdio.h>` in your file so that you can print text to the console with `printf("your text here!\n");` statements.

Number Guessing Game

Your final project is to write a game to guess a random number between 1 and 100. The game should play something like the one below, with bold indicating user input.

```
I'm thinking of a number between 1 and 100.

Your guess: 40

Too high.  Try again.

Your guess: 20

Too low.  Try again.
```

```
Your guess: 33
You got it in only 3 guesses!
```

Call your file *lab6guess_xx.c*.

You'll find the `rand()` function to be helpful. It returns a pseudorandom integer. You'll need to `#include <stdlib.h>` to use the function.

Unfortunately, it uses the same random number seed each time your program starts, making the game rather boring to play more than once. The `srand()` function changes the random number seed. For extra credit, develop a way to make your game less predictable.

What to Turn In

Include each of the following items in your submission **in the following order**. Clearly label each part by number.

1. Please indicate how many hours you spent on this lab**.** This will be helpful for calibrating the workload for next time the course is taught
2. Your neatly written, commented *lab6ph_xx.c file*. Did it work?
3. Your *lab6fib_xx.c* file. What is the 16th Fibonacci number that it calculates?
4. Your *lab6guess_xx.c* file. Does it work?
5. What were the most difficult bugs you encountered and how did you fix them?
6. Do you have any suggestions for improving this lab prompt or the lab setup?

Hints

If you are having trouble, check for the following common problems:

[Add common Pi problems here as they arise] Report any you find!

- Never connect 5V to any GPIO pin. If you suspect your board may be fried, use a multimeter to the voltage on the 3.3V and 5V pins. If you do this, be VERY CAREFUL not to short the 5V to 3.3V, or any pins to ground.
- Make sure you are connecting the ribbon cable correctly! The red stripe should align with P1 on the Pi and the left side of the T-shaped Pi Cobbler.
- The SCL, SDA, and RXD pins have pull-up resistors (when driven as inputs, they default to 3.3V rather than 0V), and should not be used for the DIP switches.
- If the Raspberry Pi crashes, your work may not autosave. Make sure to save frequently and keep a copy of your code on the lab computer as well.