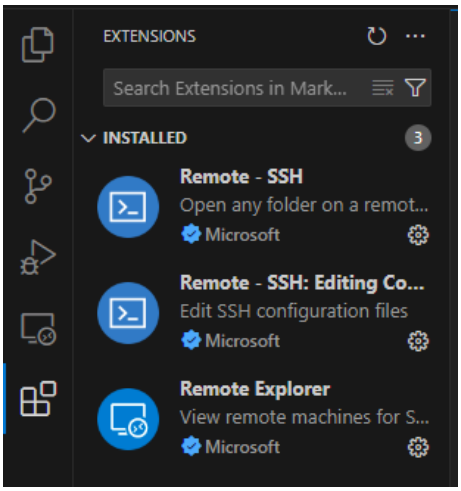
Contents

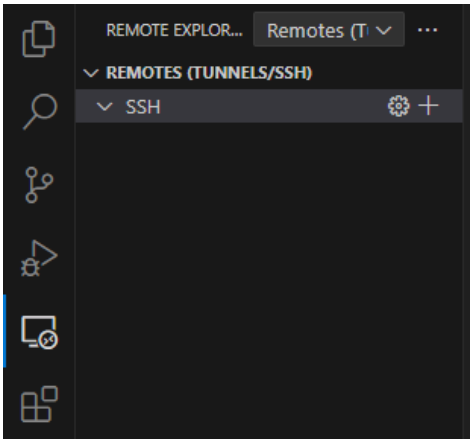
Raspberry Pi Startup Instructions

Connecting to the RPi

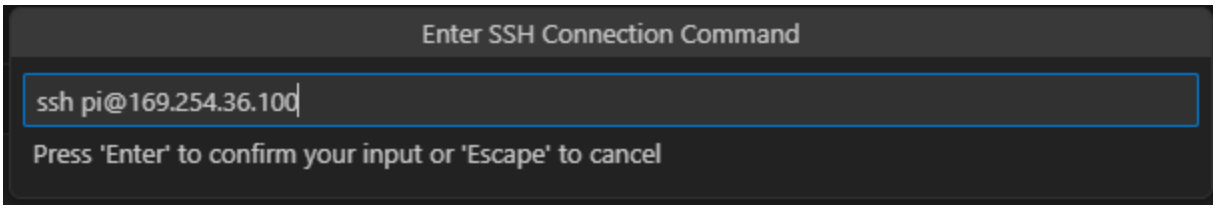
1. On the Windows computer, open Visual Studio Code (VSCode).
2. Plug in Raspberry Pi
    a. Ethernet cable to Raspberry Pi and USB->Ethernet adapter to computer
    b. Power supply to the USB-C port on the pi
3. Install the following extension in VSCode. If you just install the Remote – SSH extension, the others will be installed automatically.



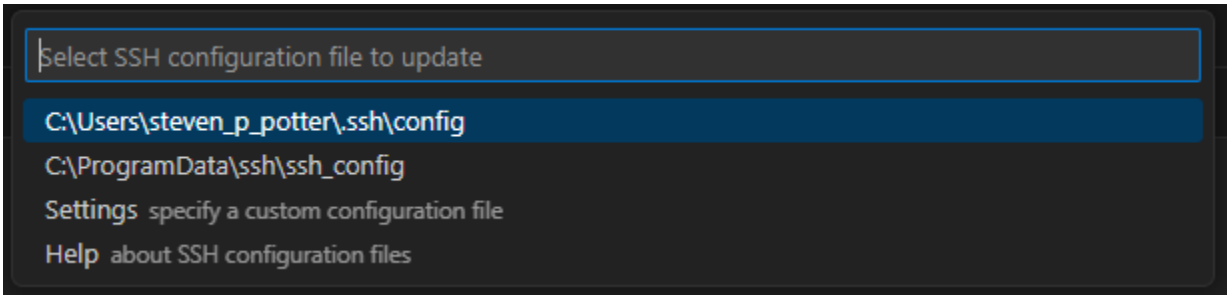
4. Go to the Remote Explorer and click the + next to SSH



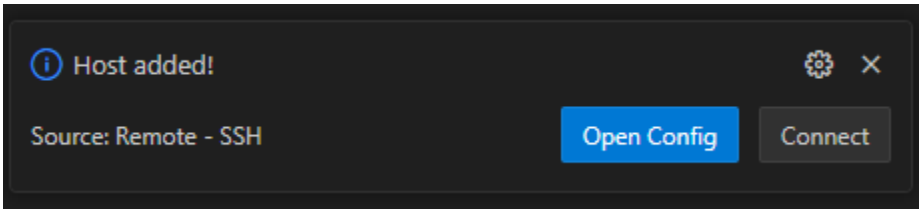
5. In the window that pops up:



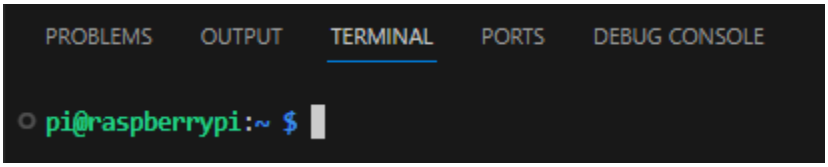
6. At the next window:



7. At the bottom right, click ‘Connect’



8. When it asks you to select an operating system, select ‘Linux’.
9. When it asks to Continue, say ‘Continue’.
10. When it asks for a password, the default password is:
    a. Thx2EcsSupp0rt
        i. The 3rd to last character is zero
11. Once it connects, select the ‘Terminal’ in the bottom pane and you should see the following. If there is not a terminal in the bottom pane (or there is no bottom pane), use the menu at the top to select Terminal->New Terminal.



RPi Configuration

1. Check to see if you are connected to the ECS-IoT network. To do so, run the following command:
    a. ifconfig
        i. This will list all of the interfaces on the RPi. The wlan0 interface should have an IP address starting with 192.168.6. If not, please let me know immediately.

2. The MicroSD cards were made based off of an 8GB image, but they are 32GB. I did this to make the process of creating MicroSD cards 4x faster. But we would like to be able to use the entire 32GB card if necessary. To do this, run the following command to expand from 8GB to 32GB.
    a. sudo raspi-config --expand-rootfs
3. I made a mistake on the image file that you will need to fix. From the terminal, run the following command:
    a. sudo nano /boot/config.txt
    b. Edit the last line in the file to change gpio-pin=16 to gpio_pin=16.
    c. Ctrl-o, enter
    d. Ctrl-x
    e. sudo reboot
4. Wait for the system to reboot (about 30 seconds), then reload the window in VSCode. It will likely ask for your password.

Network Time Protocol (NTP) Setup

For a variety of reasons, it is important for the RPi to have an accurate clock time. However, since the RPi does not have a battery to keep time while it is powered off, you get a different (and inaccurate) time each time you start it up. To resolve this, we will connect to a time server at Baylor (time.baylor.edu) and the RPi will get its clock time from that server.

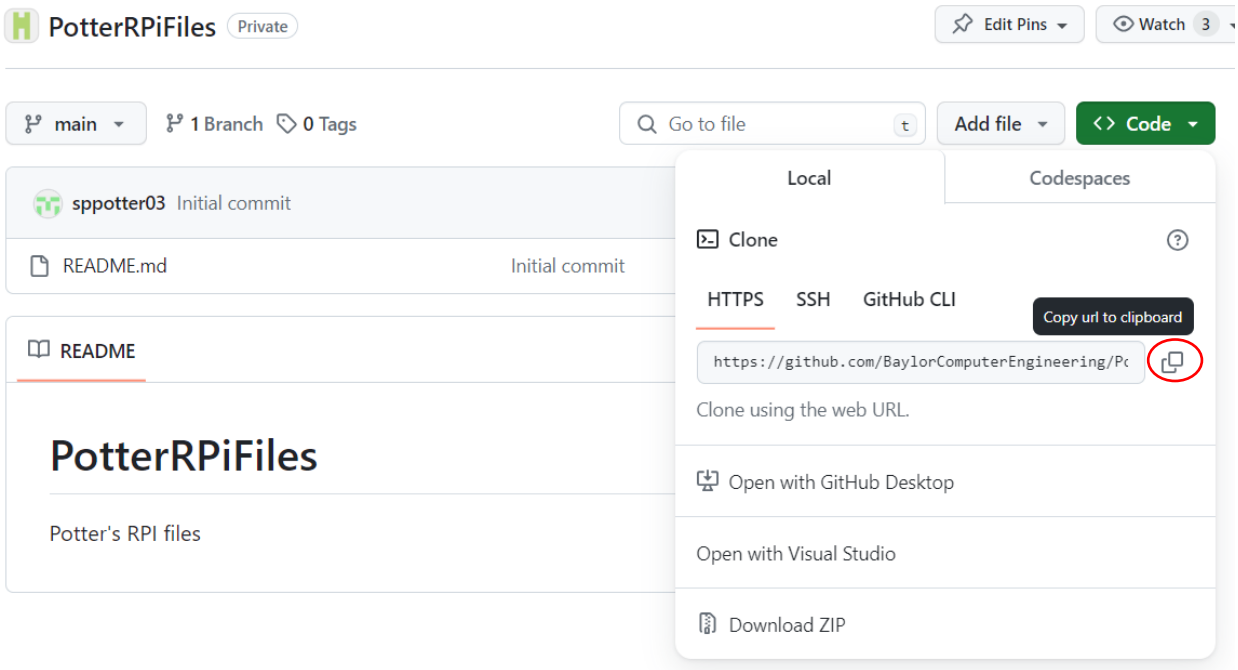
1. We need to edit the timesync config file to look for time.baylor.edu. Use the command below to edit the file and update the [Time] section to look like the screenshot below.
    a. sudo nano /etc/systemd/timesyncd.conf

```
[Time]
NTP=time.baylor.edu
#FallbackNTP=0.debian.pool.ntp.org 1.debian.pool.ntp.org 2.debian.pool.ntp.org 3.debian.pool.ntp.org
RootDistanceMaxSec=5
PollIntervalMinSec=32
PollIntervalMaxSec=2048
ConnectionRetrySec=30
SaveIntervalSec=60
```

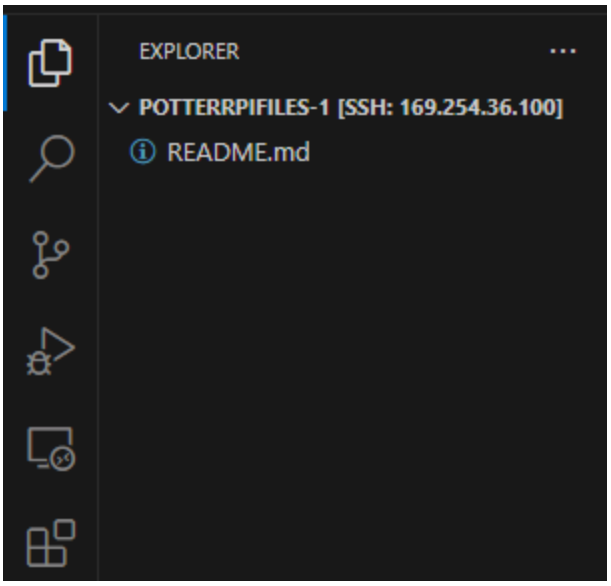
2. Before we can install the time sync package that we need, we have to manually set the time (ironic). If your time is extremely inaccurate, the package servers will not respond to you. To manually set the time, use the following command but substitute in the current time and date.
    a. sudo date --set="2 OCT 2006 18:00:00"
3. We need to install the systemd-timesyncd package to the RPi. Before we can do this, we must update the package list on the RPi, then we can install the package.
    a. sudo apt-get update
    b. sudo apt install systemd-timesyncd
4. Run the following command to ensure that NTP is enabled in the timesyncd service.
    a. sudo timedatectl set-ntp true
5. Run the following command and make sure that the system clock is synchronized, the NTP service is active, and the time and date are accurate.
    a. timedatectl

Setting up GitHub

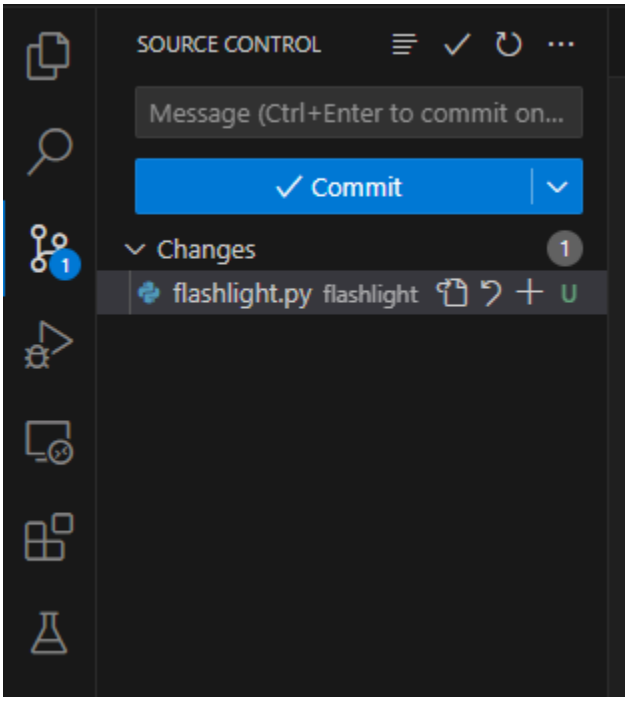
1. Login to GitHub in a web browser and create a new repository for your files. I called mine PotterRPIFiles, so I will use this as my example. When creating the repository, make select 'Private' rather than 'Public' so that others cannot see your code. Also, I recommend selecting the 'Add a README file' so that there is a file when you initially use your repository.
2. Once you have created the repository, click on 'Code' and then click the 'Copy URL to Clipboard' button.



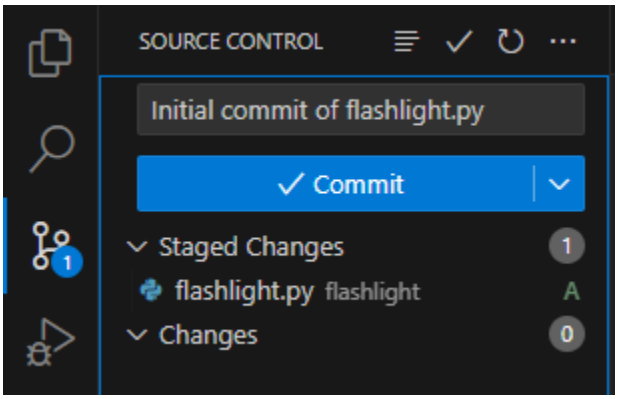
3. Go back to VSCode and run the following command from the terminal:
    a. git config --global http.sslverify false
4. In the Explorer window on the left side of VSCode, click 'Clone Repository'. If that option is not available, go to the Help menu at the top and select 'Welcome'. The Welcome page should offer the 'Clone Repository' option.
    a. When it asks you for a URL, paste the URL from GitHub.
    b. When it asks you where to clone the repository, choose:
        i. /home/pi/Documents
    c. When it asks you whether you want to open the repository, select 'Open'.
5. If it worked properly, you should see something similar to this in the Explorer window:



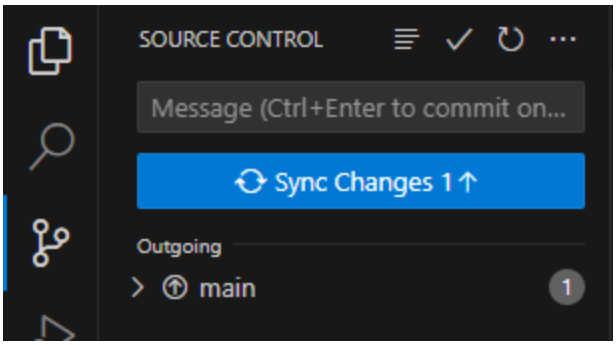
6. In the Explorer window, add a folder called ‘flashlight’
7. Click on the flashlight folder and add a new file called ‘flashlight.py’
8. Click on the ‘Source Control’ window, and you should see something like this:



9. Click the + next to flashlight.py to ‘Stage’ the changes. Also enter a commit message like ‘Initial commit of flashlight.py’. If you do forget to enter a message, it will appear as if it is trying forever because it is waiting for a commit message. After doing this, you should see something like:



10. If there is no terminal at the bottom, select Terminal -> New Terminal. In the terminal, run the following commands (using your credentials rather than mine)
    a. git config --global user.email steven_p_potter@baylor.edu
    b. git config --global user.name "sppotter03"
11. Add a Commit Message and click 'Commit'. This commits your flashlight.py file to the local repository on the RPi
12. To push the file to the GitHub repository on github.com, click 'Sync Changes'.



13. Now go back to the browser with GitHub open, refresh the browser and you should see the flashlight directory and flashlight file.

Important Reference Information

Shut down methods for the Raspberry Pi

In order of desirability:

1. From the command line:
    a. sudo shutdown -h now
    b. If you instead want to restart the Pi:
        i. sudo reboot
2. Connect a push button between pins 34 and 36. Press the button to shut down the device.
3. With either method, you will know the device is shut down when you no longer hear the fan.

How to run a python script

1. Navigate to the directory where the Python script is located.
2. python flashlight.py (or whatever the script is called).

How to stop a program

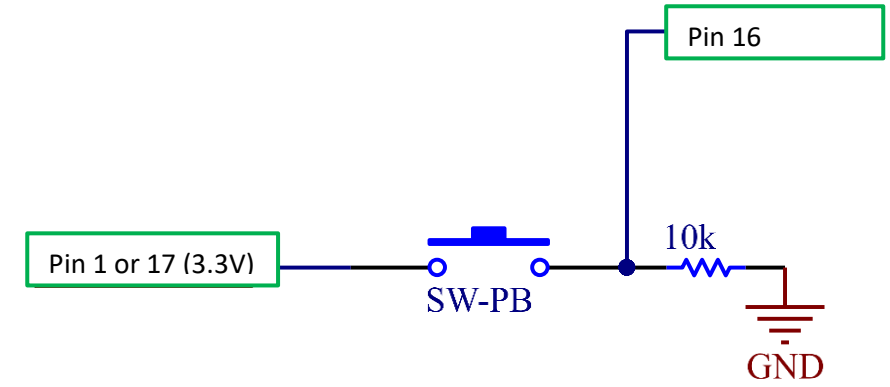
1. In the terminal window with the program running, press Ctrl-C

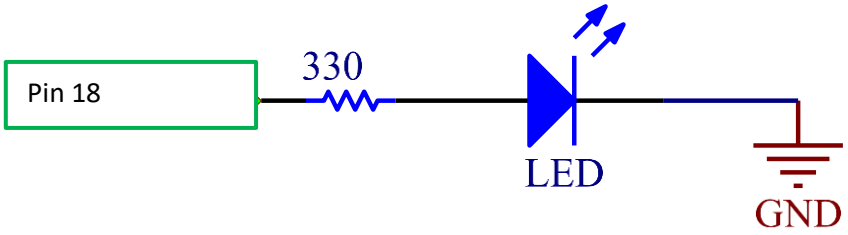
RPi Lab 1 – Flashlight

Objective

Create a flashlight that works in the same basic fashion as the flashlight created in Lab A1. To do this, you will need to create the following circuits.

Hardware





Software

To write software to control the I/O pins (pins 16 and 18), you will import and use the RPi.GPIO module in flashlight.py. Examples for how to use this module can be found here. NOTE: I recommend installing the Python Extension in VSCode to improve your Python development experience.

Part 1: Basic Polling Flashlight

1. Like with our Atmel labs, it is good to write code that is readable and maintainable. Therefore, rather than just using 16 and 18 for the pin numbers, you should create variables called BUTTON_0_PIN and LED_0_PIN and assign them with the pin numbers associated with the button and LED.
2. Using the examples in the RPi.GPIO documentation, update the software to make a basic edge-triggered flashlight like we created in Lab A1, where the flashlight turns on when the button is pressed, and it turns off the next time it is pressed.

a. To run flashlight.py, navigate to the correct directory in the terminal, then run 'python flashlight.py' from the terminal.
b. To stop the program, press 'Ctrl-C'.
c. You should have a time.sleep(0.01) at the end of the loop for the following reasons:
    i. We need to have a sleep to yield the processor so other applications can run.
    ii. Since we do not have a smoothing capacitor, the 10ms delay serves as a simple form of digital filtering. After detecting an edge, it waits for 10ms before checking again. 10ms is much more than the mechanical bounce time of a push button.
d. When running, check to make sure it works at least 20 out of 20 times with different lengths of button presses.

3. Right now, unless you added more functionality, if you stop the program with the LED on, the LED will stay on. Also, since you are in an infinite while loop, you never get to run GPIO.cleanup() as recommended in the documentation. To add the capability to clean up after ourselves, we will utilize the try/except capabilities in Python.

   Try/Except exists in other languages like C++ with fairly similar concepts and syntax (C++ uses try/catch). The general idea is that if you put code inside of a 'try' block, it will try to run it. If there is an exception, it will go to an exception handler that you get to write. There are many types of exceptions that could occur including a divide by zero, memory allocation error, and many more. But in addition to programming exceptions, there are also system-level exceptions that can be caught. One of those system-level exceptions is a keyboard interrupt. As stated in lecture, when you are running linux, you do not directly handle interrupts. However, for certain interrupts, linux will inform you that the interrupt occurred and let you take action. So when you press Ctrl-C to stop the program, linux will tell your program that a keyboard interrupt occurred….you just have to be listening. Below is some example code using Try/Except. My 'while True' loop doesn't do anything…you will need to update it your logic. Also, you will need to import the 'sys' module so that you can call sys.exit().

```
try:
    while True:
        time.sleep(.01)
except KeyboardInterrupt:
    print('Got Keyboard Interrupt. Cleaning up and exiting')
    GPIO.output(LED_0_PIN, GPIO.LOW)
    GPIO.cleanup()
    sys.exit()
```

Part 2: Waiting for an Edge

The RPi.GPIO module offers a wait_for_edge function that seems perfect for what we are trying to do with the flashlight. Create a new file called flashlight_edge_detect.py and use the wait_for_edge function to implement the flashlight. You should not use a sleep in your 'while' loop because wait_for_edge is non-blocking and does not consume CPU time while waiting. Make sure that your

solution works 20 out of 20 times. If you are like me, it will not work every time and require further investigation.

The issue that you will come across is that there is mechanical bounce in the switch. This means that when you press it or release it, even though it feels like it cleanly makes contact once, there are actually several oscillations between the switch making contact and not making contact. This is more common when you press the button, but it still exists to a smaller extent when you release the button. The wait_for_edge function has a bouncetime parameter where it will ignore edges for a certain amount of time after the first edge is detected. The documentation has a section about debouncing at the end of the 'Inputs' page. It talks about the bouncetime parameter in the 'add_event_detect' function, but it also exists for the 'wait_for_edge' function. Based on the Digilent screenshots from Atmel lectures and Lab A1, we can use a bouncetime of 10ms. Our switch bounce will last less than 10ms, and we know that even a quick tap of the button will take 50ms or more.

Part 3: Detecting Event and Using Callbacks

The RPi.GPIO library also offers something that resembles interrupt handling like we had with EIC interrupts on the Atmel board. In Atmel, when an interrupt occurred, an interrupt handler function was called. An interrupt handler function could also be described as a callback function.

In RPi.GPIO, you can call add_event_detect() to add an event for the system to be watching for. When calling the function, you can add a callback function name. When the event occurs, the callback function will automatically run. This is very similar to the way the EIC handler worked except that this callback function runs in a different thread of execution, meaning it can run in parallel with your main program by running on a different processing core (remember, our processor is quad core). So you can continue operating the main program like usual while also doing other stuff in response to the button press.

To experience this, please create a new file called flashlight_event_detect.py. The goal of this part is not to create a flashlight that works the same as the previous parts. Instead, the LED should turn on and stay on while the button is being held, and it should turn off once the button is released. Use the following code as your "main" loop:

```
cnt = 0
try:
    while True:
        time.sleep(1)
        print(cnt)
        cnt = cnt + 1

except KeyboardInterrupt:
    print('Got Keyboard Interrupt. Cleaning up and exiting')
    GPIO.output(LED_0_PIN, GPIO.LOW)
    GPIO.cleanup()
    sys.exit()
```

This will print out an incremented number every 1 second. The button press should be detected by adding it as an event. In the callback, you will need to wait for the button to be released. While waiting,

the callback should be printing ‘abc’ every 100ms. Because it is running in a different thread on a different processor, you should see these prints interleaved with the counter prints from main. Example shown below has a long button press and then a short button press.

```
6
7
8
In rising_edge_callback
abc
9
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
10
abc
abc
abc
abc
Leaving callback
11
12
13
14
15
16
In rising_edge_callback
abc
abc
17
abc
Leaving callback
18
19
```

Submitting the Lab

1. Please submit the following files in Canvas:
    a. flashlight.py
    b. flashlight_edge_detect.py
    c. flashlight_event_detect.py

Lab 2 – ADC and PWM

Part 1 – Analog to Digital Conversion

The goal of this part of the lab is to develop the ability to read an analog input signal and calculate the voltage of the signal for use in future logic and calculations. Since the RPi does not have an analog to digital converter built into the device, we will use and ADC chip and communicate with it using the SPI protocol.

RPi Configuration

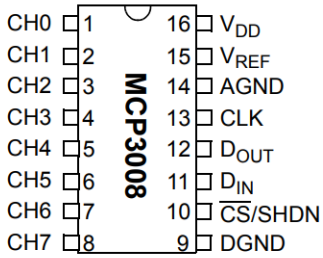
The MicroSD cards that were provided already have SPI enabled in /boot/config.txt. As an FYI, SPI is not enabled by default, but I changed a setting in /boot/config.txt to enable it. So nothing needs to be done to enable SPI on the RPi. However, we need help interacting with the SPI hardware on the RPi, so we need to install the spidev library package. In the most recent versions of Raspberry Pi OS, you cannot install python packages using 'pip install' directly on the RPi. You must first create a virtual machine on the RPi, then install the package on the virtual machine. Below are the steps to create the virtual machine, start it, and install the spidev package.

1. sudo apt install python3-venv
    a. This is likely already the newest version and will just inform you that it does not need to be updated.
2. Navigate to the /home/pi/Documents directory
    a. IMPORTANT: You do NOT want to run the next command inside of your GitHub directory where you created the flashlight. This will add thousands of files that are frequently changing to your GitHub repository.
3. python3 -m venv potter --system-site-packages
    a. The 'potter' argument is the name of the virtual machine you are creating. You can name it whatever you want (except for potter, that's my name not yours).
4. source potter/bin/activate
    a. This starts the virtual machine
    b. If you want to stop using the virtual machine at some point, you can just run the 'deactivate' command and the virtual machine will exit.
5. pip install spidev
    a. This might already be installed and up to date.

ADC Hardware

Using the MCP3008 datasheet and the figure below, wire the MCP3008 to the RPi using the following guidance:

PDIP, SOIC



1. VDD = RPi 3.3V
2. VREF = RPi 3.3V
3. AGND = RPi GND
4. CLK = RPi SCLK
5. DOUT = RPi MISO
6. DIN = RPi MOSI
7. CS = RPi Pin 24 (GPIO 8, CE0)
8. DGND = RPi GND
9. CH0 = Analog voltage input

ADC Software

Now we need to write software to interface with the ADC chip. The goal is to be able to apply a voltage to the Channel 0 input of the ADC chip and print an accurate reading of the voltage to the terminal. This requires using SPI to communicate between the ADC chip and the RPi. Please complete the following tasks while using your activated virtual machine. You will know you are using the virtual machine if your terminal prompt has the name of the virtual machine in parentheses on the left side as shown below:

```
(potter) pi@raspberrypi:~/Documents/PotterRPiFiles-1/flashlight $
```

1. In VSCode, create a directory called adc_test within the directory where you cloned your GitHub repository.
2. Within the adc_test directory, create a file called adc_test.py
3. Utilizing the spidev library (spidev documentation), write software to read channel 0 and print the raw adc value and voltage to the screen every 1 second.
    a. Settings:
        i. Bus = 0
        ii. Device = 0
        iii. Max Speed Hz = 1000000
        iv. Please do not modify any of the other settings
    b. For data transmission, you will use the xfer function. The xfer function:
        i. Takes a list of values to be sent by the RPi
        ii. Returns a list of values that were returned by the ADC chip
        iii. Figure 6-2 in the MCP3008 datasheet shows the data to be sent and returned
    c. Use the return value to calculate the raw ADC value
        i. The range of this number should be 0-1023
    d. Use the raw ADC value to calculate the voltage

4. Use the Digilent Supplies functionality to provide different voltages (0-3.3V) and verify that the voltage reported by the ADC is correct.

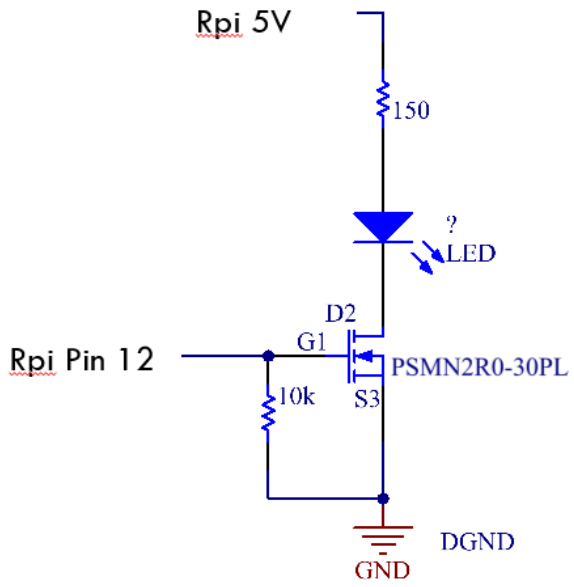
Photoresistor Hardware

1. Use your multimeter to determine the resistance of the photoresistor in ambient lighting, very dark, and very bright. Use your whole hand to cover the sensor for dark and use a flashlight for light.
2. Now design a circuit with a resistor and the photoresistor in series. The circuit will be a voltage divider where you will measure between the resistor and photoresistor. The goal is that you will apply 3.3V to the circuit and the circuit will produce a low voltage (< 0.5V) when a flashlight shines on it and high voltage (> 2.8V) when you cover the sensor. Design the circuit, build it, and test it with your multimeter.
    a. The resistor choices are:
        i. 150
        ii. 330
        iii. 660
        iv. 1k
        v. 10k
        vi. 1M
3. Now connect the circuit to the ADC module and verify that the correct voltages are being read by your adc_test software.

Part 2: Photodimmer

The goal of this part is to dim and brighten an LED such that the LED is off when a flashlight is shined on the photoresistor and the LED is at full brightness when the photoresistor is covered (dark). For all light levels between full darkness and full brightness, the LED should smoothly dim and brighten according to the light level.

LED Circuit



Command Line PWM Interface

Before we try to write software to control the LED with PWM, we will try it from the command line. These commands will be used in the software as well, but it is best to verify from the command line first.

1. Before we can use hardware PWM, we need to enable it by doing the following:
    a. sudo nano /boot/config.txt
    b. At the end of the file, at the following line:
        i. dtoverlay=pwm,pin=18,func=2
    c. Ctrl-O, then enter to save
    d. Ctrl-X to exit
    e. sudo reboot
        i. Wait for the RPi to reboot
        ii. These changes only take effect when booting up
2. ls /sys/class/pwm/pwmchip0
    a. This command begins with a lowercase L, not a capital i
    b. You should see some files and directories, but there should not be a directory called 'pwm0'
3. echo 0 > /sys/class/pwm/pwmchip0/export
    a. This step only needs to be done the first time you run this after booting up. This command goes to pwmchip0 (there are multiple pwm chips on the SoC), and by writing 0 to the file, it enables pwm0 and therefore creates a pwm0 directory.
4. ls /sys/class/pwm/pwmchip0
    a. This time, you should also see a directory called 'pwm0'
5. echo 500000 > /sys/class/pwm/pwmchip0/pwm0/period
    a. This sets the period to 500000. Since period is expressed in ns, the period is 500 microseconds, therefore making the frequency 2kHz, which is good for dimming an LED.
6. echo 300000 > /sys/class/pwm/pwmchip0/pwm0/duty_cycle
    a. This sets the duty cycle to 300000. Since 500000 is the max value, this should set the LED at approximately 60% brightness.
7. echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable
    a. This turns pwm0 on. Echoing 0 to this file will turn it off.
8. Use the oscilloscope mode on the Digilent Analog Discovery 2 device to verify that the signal that is generate by RPi Pin 12 is 2kHz and that the duty cycle is varying properly with your duty_cycle entries. Try other values to see the brightness of the LED and the scope output change.
    a. Capture screenshots of the signal at 25%, 50%, and 75% duty cycles. On the 25% duty cycle screenshot, show the measured frequency to verify that the signal is 2kHz.

PWM Software

In order to use hardware PWM, we need to run the same basic commands that we ran in the previous section, but we need to run them from a python script. The 'os.system("whatever command I want to run")' allows us to run command line commands from a python script, and we will utilize this to create a utility to help us with hardware PWM.

1. In your GitHub repository, create a new directory called ‘utilities’. Within this directory, create a new file called ‘utilities.py’.
2. In Canvas, I have provided a nearly complete ‘utlities.py’. Download the file in Canvas, and copy/paste the contents into your ‘utilities.py’
3. The code in ‘utilities.py’ creates a class called HW_PWM. The constructor (__init__) is complete. Please read this code and understand it.
4. After reading and understanding the constructor, implement the set_duty_cycle function. The function takes a duty cycle percentage (0-100). You need to convert it to the kind of value you would’ve used from the command line, then write that value to the duty_cycle file (as done in the command line).
    a. If you receive a duty_cycle_percent parameter value above 100, you should detect that and set the percent to 100. If you receive a duty_cycle_percent parameter value below 0, you should detect that and set the percent to 0.
5. To test this function, we will run a python script (that I provide) that gradually dims and brightens the LED.
    a. In your GitHub repository (not in any subdirectory like utilities), create a directory called led_dimmer. Within this directory, create a file called led_dimmer.py. I have provided the complete code for this in Canvas. Please copy and paste that code into led_dimmer.py.
    b. Run led_dimmer.py and verify that your LED dims and brightens.
        i. Please note that you can ignore the error on the third line of the output. It is complaining because we have already run that command before and it doesn’t need to be done.

Photodimmer Software

In your GitHub repository, create a new directory called photodimmer and create photodimmer.py inside of this directory. In photodimmer.py, use a 2kHz PWM signal to control the LED brightness. For consistency among the class, please use a 1k resistor in series with the photoresistor. The program should start by:

1. Printing ‘Cover the photosensor and press the push button’ and waiting for the user to complete this step.
2. Printing ‘Shine a flashlight on the photosensor and press the push button’ and waiting for the user to complete this step.
3. From here on, the maximum and minimum values captured at the beginning of the program should be used to calibrate the PWM output such that:
    a. The LED is off when a flashlight is shined on the photoresistor
    b. The LED is at full brightness when the photoresistor is covered (dark)
    c. For all light levels between full darkness and full brightness, the LED should smoothly dim and brighten according to the light level.
        i. I recommend a delay of 1ms in your loop to make the transitions very smooth
4. When the program is stopped from the keyboard, the LED should turn off.

Submitting the Lab

1. Please submit the following files in Canvas:

a. utilities.py
b. photodimmer.py