Lab 3: Servo Control

By Alec Newport [acn55] and Eldor Bekpulatov [eb654]
Thursday Lab - Section 403
November 7, 2019

Introduction

This lab was designed to get us familiar with using GPIOs as outputs, driving a continuous rotation Parallax servos motors using Pulse Width Modulation (PWM) from a GPIO output.

The lab consisted of four parts. The first part of the lab primarily focused on configuring the external hardware and internal libraries to properly and safely use GPIO pins as outputs to drive a Parallax servo motor. The second part consisted of using 6 of the unused GPIO pins as inputs to select different modes of operation on the two servos. The third part of the lab consisted of creating a PyGame user interface that correctly displays the recent history of operating modes on two servos and also registered a stop signal from the piTFT's touch screen. The last part of the lab included constructing a robot and running a script at startup without any user input that drove the robot through a sequence of moves.

Design and Testing

Part 1

For the first part of the lab, we used the rpi.GPIO library to develop a control application for the servo. As in Lab 2, we included the rpi.GPIO library, but for this experiment, we selected a GPIO pin and configured it as an output. Figure 1 displays the code used to set up the pin 26 as an output.

```
1  import RPi.GPIO as GPIO   # Import the GPIO library.
2  import time               # Import time library
3
4
5  GPIO.setmode(GPIO.BCM)    # Set Pi to use pin number when referencing GPIO pins.
6                            # Can use GPIO.setmode(GPIO.BCM) instead to use
7                            # Broadcom SOC channel names.
8
9  pwmPin = 26                    # set pin number to 26
10 GPIO.setup(pwmPin, GPIO.OUT)    # Set GPIO pin 26 to output mode.
11
```

Figure 1. GPIO setup

Once we configured the pin to be an output, we moved on to implementing a PWM signal generation function. Luckily, GPIO module of RPi library includes a PWM class, which is used to generate the signals on a specified pin with parameters such as frequency and duty cycle. Frequency is a self explanatory term, and it defines the number of times the signal should toggle per a second. Duty cycle refers to the amount of time (in percent) the signal is high in a given period of the signal. These two parameters will be used to control the servos.

Before moving on to simulating PWM signals to control our servos, we implemented a simple test script, blinky.py, that checked if our GPIO was working correctly. In this script, we set the PWM output to a low frequency of about 10 Hz and duty cycle of about 50% to see if LED would toggle ten times a second with half of each period as on and the other half as off. We did this for 5 seconds, and changed the frequency to 5 Hz and duty cycle to 25% for another 5 seconds. In the second 5 seconds, the LED toggled less frequently and spent less time turned on. This confirmed we can

modify our PWM signals properly online for later parts of the lab. Figure 2 displays our script. We also connected the oscilloscope to see the actual PWM signal generated.

```
fq = 10                         # set frequency to 30 Hz
dc = 50                         # set dc variable to 50 for 50%
pwm = GPIO.PWM(pwmPin, fq)      # Initialize PWM on pwmPin 100Hz frequency


try:
    pwm.start(dc)                   # Start PWM with 0% duty cycle
    time.sleep(5)                   # sleep for 5 seconds

    fq = 5
    dc = 25
    pwm.ChangeDutyCycle(dc)         # where 0.0 <= dc <= 100.0
    pwm.ChangeFrequency(fq)         # where freq is the new frequency in Hz

    time.sleep(5)                   # sleep for 5 seconds
    pwm.stop()                      # stop PWM

except:
    pass

# clean up GPIO and quit pygame on normal exit
GPIO.cleanup()

```

Figure 2. LED test script

With a confirmation that our PWM pin was working, we moved on to implementing the actual PWM signals necessary to drive our Parallax servos. In order to gain information on what signals are required to drive the servos, we turned to the datasheet. Figures 3-6 are the diagrams from the Parallax servo datasheet. Figure 3 outlines the setup to power the servos. As seen in the diagram, Vservo and Vμ are two independent power sources. This is because servo will pull higher current than the microcontroller pin can provide. In addition, servos will put noise back onto its power rail. If this is the same power source as is being used by the Pi, it can cause abnormal operation or even damage the board. As indicated in Figure 3, the I/O pin from the microcontroller will provide the PWM signal. A sample signal to stop the parallax servo is provided in Figure 4.

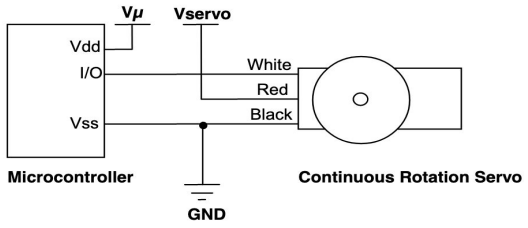

Figure 3. Powering the servos

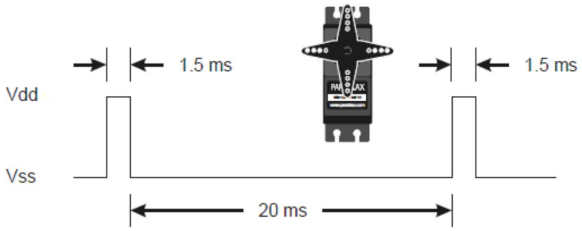

Figure 4. Calibration PWM signal

The data sheet tells us that our calibration pulse must be 1.5 ms wide with 20 ms delay in-between pulses to stop the servo from rotating. If it rotates in either direction with this given pulse, there is a small potentiometer accessible from the case to help us calibrate it, and we can play with it until the servo is almost static.

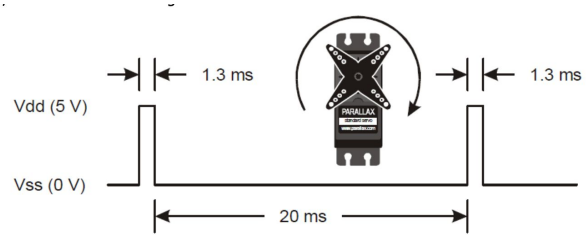

Figure 5. Clockwise full speed

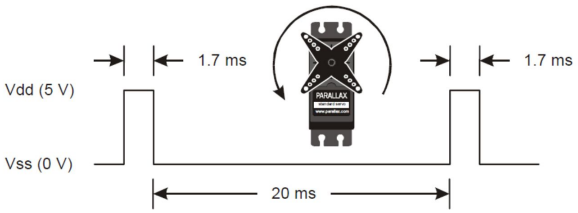

Figure 6. Counter-clockwise full speed

Now that we know how to make the servo stop, we needed to make it rotate. The datasheet tells us that the pulse width can be increased to make the servo rotate counter-clockwise, and decreased to make it rotate clockwise. Figure 5 shows that 1.3 ms pulse is used to make the servo rotate at full speed clockwise. Figure 6 shows that 1.7 ms pulse is used to make it rotate counter-clockwise at full speed. To achieve half speed clockwise, we would make the pulse width halfway between 1.3 and 1.5ms, i.e. 1.4ms, and for half-speed counter-clockwise, we would make it halfway between 1.5 and 1.7ms, i.e. 1.6ms.

Given the above information, we made a function to calculate the frequency and duty cycle for the PWM signal. Figure 7 shows that function. The argument `step` is the change in width of the pulse in ms from 1.5 and is bounded between -0.2 and +0.2.

```
5   def calculateFreqDC(step):
6       """
7       Returns the (frequency, dutyCycle) as tuple given a step.
8
9       Param: step (float) bounded by -0.2 <= step <= +0.2
10      Return: (freq, dutyCycle) tuple of floats
11      """
12      w = (1.5 + step)
13      d = (20+w)
14      return (1000/d, w/d*100)
```

Figure 7. Helper function to calculate frequency and duty-cycle

With the groundwork done to control the servo, we wrote another function to simply take a PWM pin and update its mode of operation called `setMode`. Figure 8 shows this function which references the `calculateFreqDC` function from above.

```
16  def setMode(step,pwm):
17      (fq, dc) = calculateFreqDC(step)
18      pwm.ChangeDutyCycle(dc)
19      pwm.ChangeFrequency(fq)
```

Figure 8. Helper function to update PWM pin

With the PWM control logic implemented, we performed a sanity check by hooking up the oscilloscope to check if our PWM signal is correct for different modes of operation. Figure 9-11 shows the PWM signals for full speed counter-clockwise, full speed clockwise, and stopped modes respectively.

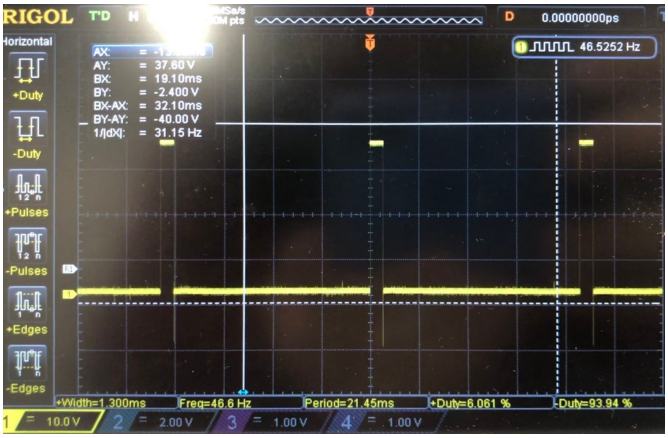

Figure 9. Full speed counter-clockwise mode PWM signal

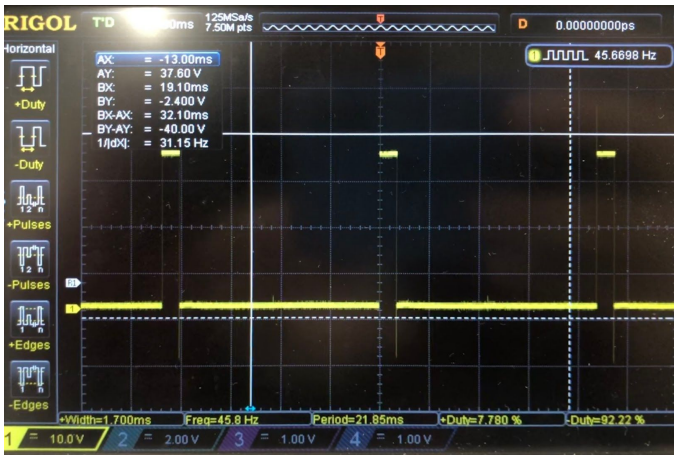

Figure 10. Full speed clockwise mode PWM signal

Figure 11. Full speed stopped mode PWM signal

We had confirmed our GPIO is sending the correct signals for the extreme modes of the operation, meaning full speed modes in either direction. Now we had to make sure the intermediate modes were reachable as well. With a simple for-loop we incremented the step value of the servos in either direction to make the servo rotation speed accelerate linearly from rest to full speed in 10 steps while waiting 3 seconds in each step. So from rest, it would take 30 seconds to reach full speed in either direction. Figure 12 shows our code to check this, servo_control.py.

```
32      (fq, dc) = calculateFreqDC(0)
33      pwm = GPIO.PWM(pwmPin, fq)      # Initialize PWM on pwmPin
34      pwm.start(dc)                   # Start PWM
35
36      # speed up Clockwise
37      for step in range(0, 21, 2):
38          step = step/100.0
39          setMode(step,pwm)
40          time.sleep(3)
41
42      #speed up ounter-clockwise
43      for step in range(0, -21, -2):
44          step = step/100.0
45          setMode(step,pwm)
46          time.sleep(3)
47
48      pwm.stop()                      # stop PWM
```

Figure 12. For-loop to test all modes of operation in range

Part 2

This part of the lab consisted of using 6 of the unused GPIO pins as inputs in order to select different modes of operation on the two Parallax servo motors. Since we had dealt with handling GPIO inputs from external buttons using callback functions before, this section was not too difficult. We replicated some steps from the previous lab, for example configuring the pull-up button setup. Since we had already talked about the button configuration in previous labs, we'll just show the wiring diagram for each button and a template script for setting up a callback function for the GPIO pin (Figures 13 - 14).

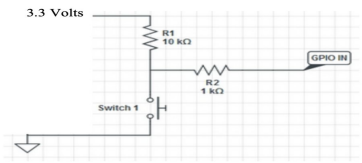

Figure 13. External GPIO setup

```
89  GPIO.setup(21, GPIO.IN)
90
91  def GPIO21_callback(channel):
92      print("falling edge detected on 21")
93      # function call for this button here
94
95  GPIO.add_event_detect(21, GPIO.FALLING, callback=GPIO21_callback, bouncetime=200)
```

Figure 14. Callback function setup

Using the templates above we connected six buttons on the unused GPIO pins and properly initialized them in our script with appropriate callback functions. We initially set up two GPIO pins as global variables to drive our servos. The callback functions of three of the six control the modes of the left servo, and those of the other three control the right one. Since the PWM objects were globally declared, we were able to reference them in our callback functions. In addition, having a modular `setMode` function from Part 1 made it convenient to program the functions to each button. For every button, we pass in its respective servo pin and the mode to which the servo is to be set. Finally, we simply started both servos and programmed the seventh button with an internal pull-up resistor as the quit button. Figure 15 shows our servo_control script excluding the GPIO callback functions.

```
####################SERVOS#####################
GPIO.setup(26, GPIO.OUT) # LEFT
GPIO.setup(19, GPIO.OUT) # RIGHT
GPIO.setup(27, GPIO.IN, pull_up_down=GPIO.PUD_UP) # QUIT

(fq, dc) = calculateFreqDC(0)
pwmL = GPIO.PWM(26, fq)          # Intialize PWM
pwmR = GPIO.PWM(19, fq)          # Intialize PWM
try:
    pwmL.start(dc)                  # Start PWM
    pwmR.start(dc)                  # Start PWM
    GPIO.wait_for_edge(27, GPIO.FALLING) # wait for quit
    pwmL.stop()
    pwmR.stop()

except:
    pass
GPIO.cleanup() # clean up GPIO on exit
```

Figure 15. Servo control script

Part 3

The third part of the lab involved creating a PyGame user interface for the robot in rolling_control.py consisting of a large stop/resume button, a quit button that terminates the program, and rolling histories of the operating directions and time stamps for each of the two wheels. Since we had already implemented the six control buttons and one physical quit button in Part 2, we simply added the PyGame logic on top of what was already written.

We started with the most complicated portion of this script: the rolling histories. We needed to track and display the three most recent operation modes for each wheel, meaning we needed a LIFO data structure. The most obvious choice for this is Python's built-in double-ended queue (deque). By removing old entries from the right and adding new ones to the left, we can keep track of the order of the events (most recent on the left, oldest on the right). Then, when it came time to display them, we could make the leftmost element appear highest on the list, and continue downwards. We maintain three elements in each history by always popping from the list immediately prior to adding a new element, both in the same function: `update_hist` (Figure 16). Each element in these deques is a tuple consisting of the direction of motion (either "Clkwise," "Counter-Clk," or "Stop") and the time the action was taken in seconds since the start of the program. We initially populated each deque with three ("Stop", 0) entries to maintain our invariant of always having three entries in each. The `update_hist` function takes the deque object and new motor speed as arguments. If the motor speed is less than 0, it's rotating clockwise; if it's greater than 0, it's rotating counter-clockwise; and if it's exactly 0, it's stopped.

```
93  def update_hist(hist, speed):
94      hist.pop()
95      sec = int(time.time() - start_time)
96      if speed < 0:
97          hist.appendleft(("Clkwise", sec))
98      elif speed > 0:
99          hist.appendleft(("Counter-Clk", sec))
100     else:
101         hist.appendleft(("Stop", sec))
```

Figure 16. Update history logic

The `update_hist` function is only called when one of the six physical buttons (not including the physical quit button) is pressed. Therefore, we modified our previously-defined callback functions to include a call to `update_hist`. We also decided that since the screen isn't doing any kind of animation, it only needs to update the display when something changes, such as when the histories are updated. So we added a new function, `update_screen`, that redraws the screen, and we call it from each callback as well. Figure 17 shows an example of the updated callback for one of the physical buttons.

```
179 def GPIO16_callback(channel):
180     print("falling edge detected on 16")
181     speed = 0
182     update_hist(l_hist, speed)
183     update_screen()
184     setMode(speed, pwmL)
```

Figure 17. Modified callback function

There are three major elements of the screen in this program: the big button in the center, the two servo histories, and the quit button. We wrote individual functions to draw each of these, called `draw_big_button`, `draw_histories`, and `draw_quit_button` respectively, to better compartmentalize our code. Therefore, the complete `update_screen` function, shown in Figure 18, wipes the previous screen, calls the three draw functions, and flips the display.

```
164 def update_screen():
165     screen.fill(WHITE)
166     draw_big_button()
167     draw_histories()
168     draw_quit_button()
169     pygame.display.flip()
```

Figure 18. Update screen function

The first draw function, `draw_big_button`, either draws a red "STOP" button or a green "RESUME" button. We have a global variable called `motor_en` that tracks the status of this button. If `motor_en` is True, the motors are currently allowed to run, so we display the red button, allowing the user to disable them. Otherwise, we display the green button, allowing the user to restart them. We had some trouble getting the circle to line up correctly in the center of the screen. For some reason, even though `pygame.draw.circle` supposedly takes coordinates for the center of the circle as one of its arguments, and the screen is 320x240, the coordinates that center the circle on the

screen are about (95,75), while those for the text rect are the expected (160, 120). We think maybe the documentation is incorrect, and the circle function actually takes the upper-left corner of the rect.

The second draw function, `draw_histories`, takes the entries from the two deque objects and renders the text on either side of the big button. Because of the way we ordered these objects, we are able to simply iterate through each in order from left to right and display the data one line at a time. We have variables called `line_num` and `line_spacing` that allow us to space the text nicely on the screen. We aligned each operating direction text to the left and each time stamp to the right, and then used the formula `line_num*line_spacing+80` for the vertical placement. As we iterate through each deque, we draw the action text, then the time stamp text, then we increment `line_num` so that each line of text is `line_spaceing` apart (a value we set to 20 through trial and error).

The final draw function, `draw_quit_button`, is the simplest of the three. The quit button never needs to change, so we simply need to print the text “QUIT” to the lower-right corner of the screen. Even though it doesn’t change, we still need to redraw it with everything else because we wipe the screen each time something changes. We could have written the code so that only the portion of the screen being updated is erased, but that would be significantly more complicated than simply redrawing the quit button. If performance were an issue, this might be something to consider, however.

The main loop of this script mirrors those of previously-implemented PyGame UIs. On startup, we enable the PWM signals to the motors and draw the first screen. Then, we enter a while loop based on a global variable called `running` that keeps track of when the user wants to end execution of the script. While that variable is `True`, we continuously check for mouse click events and `pygame.QUIT` events. In the latter case, we simply set running to `False`, breaking out of the loop and terminating the script. In the former, we check if the click occured on either the STOP/RESUME button or the QUIT button. If it was on the former, we toggle `motor_en` and either stop or restart the PWM signals to the motors based on the new value of `motor_en` (we do this rather than writing the stop PWM signal because it’s meant to act as an emergency stop, cutting power to the motors). We then call `update_screen` to change the big button. This is the only time `update_screen` is called outside of one of the physical button callbacks. If the QUIT button was pressed, like with the `pygame.QUIT` event, we set running to `False`, which breaks out of the outer loop and terminates the script. Figure 19 shows the complete main loop of this script.

```
224     pwmL.start(dc)                  # Start PWM
225     pwmR.start(dc)                  # Start PWM
226
227     # Draw screen
228     update_screen()
229
230     # ------main loop-------
231     while running:
232         for event in pygame.event.get():
233             if event.type == pygame.QUIT:
234                 running = False
235                 break
236             elif(event.type is MOUSEBUTTONUP):
237                 x,y = pygame.mouse.get_pos()
238                 if y < 150 and y > 90 and x < 190 and x > 130:
239                     # STOP/RESUME button
240                     motor_en = not motor_en
241                     if motor_en:
242                         # Motors resuming
243                         pwmL.start(dc)
244                         pwmR.start(dc)
245                     else:
246                         # Motors stopping
247                         pwmL.stop()
248                         pwmR.stop()
249
250                     update_screen()
251
252
253                 elif y > 200 and x < 300 and x > 260:
254                     # Quit button
255                     running = False
256
257         # Limit frames per second
258         clock.tick(frame_rate)
```

Figure 19. Program flow

Part 4

The first component of this section involved constructing a small robot exactly as we were instructed in the lab handout. A picture of our completed robot running run_test.py, as described later, is shown in Figure 20.

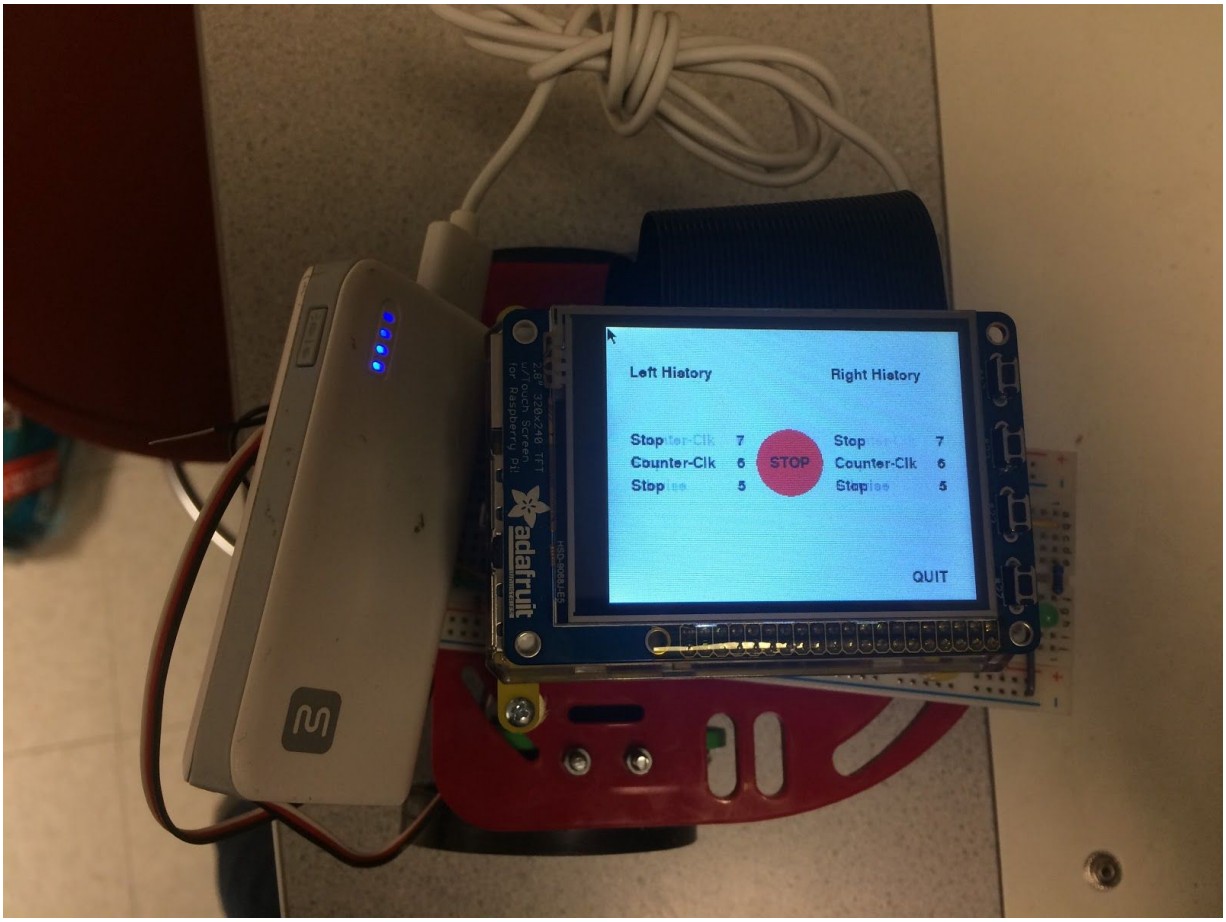

Figure 20. Robot running run_test.py

The way we tested the robot, as mentioned, was with a script called run_test.py. It is very similar to rolling_contol.py, with the exception that rather than waiting for a user to press buttons to drive the servos, the script autonomously progresses through a series of states that drive the robot in specific ways. The state diagram is shown in Figure 21. To summarize, the robot repeats the following sequence: forward for one second, stop for half a second, back for one second, stop for half a second, pivot left for one second, stop for half a second, pivot right for one second, stop for half a second. We were able to condense all of the times when the robot is supposed to stop into a single state by tracking the state it was in prior to transitioning to the stop state in the variable `prev_state`. The current state is held in the variable `state`, the state to which we are transitioning in `next_state`, the time at which we entered the current state in `state_time`, and the time to remain in the current state in `trans_time`.

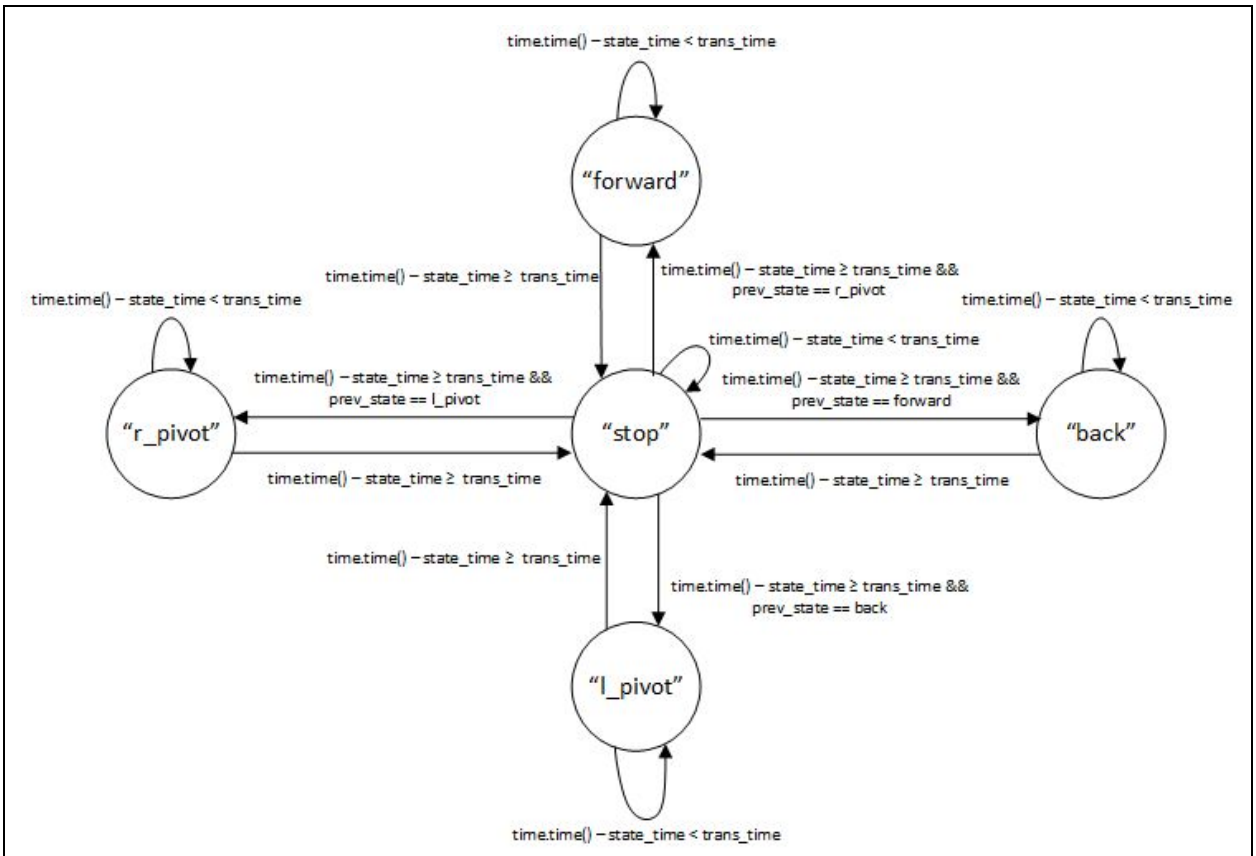

Figure 21. State diagram for run_test.py

The function that is responsible for setting `next_state` is named `update_state`. Based on the current state, the time that has passed, and, if the current state is `"stop,"` the previous state, `update_state` sets `next_state` to continue in the loop. Then, once `update_state` has finished, we execute the state transition using the line `state = next_state`. We only update the state when `motor_en` is True, since that's the only time the robot can execute the movement associated with the state. When the user pauses execution by pressing the red STOP button and motor_en is set to false, we want the state timer to pause as well. We do this by subtracting the time that has passed in the state from trans_time. Then, when execution is resumed, we set state_time to be the current time and restart the motors. That way, the robot should actually execute the movement in each state for the expected amount of time, no matter how many times motion is paused, even within a single state.

The final change we made for run_test.py from rolling_control.py was to move the motor controls into their own functions separate from the button callbacks. We left the callbacks in, but they now call these other functions. We do this because a callback function cannot be called anywhere in the code itself, but we still want to have that functionality available in multiple locations. These functions are aptly named `left_forward()`, `left_back()`, `left_stop()`, `right_forward()`, `right_backward()`, and `right_stop()`. To move the robot forward, we call `left_forward()` and `right_forward()`; to go backward, `left_back()` and `right_back()`; to pivot left, `left_back()` and `right_forward()`; to pivot right, `left_forward()` and `right_back()`; and to stop, `left_stop()` and `right_stop()`.

All of these are called on the transitions rather than continuously within the states because the PWM signals only need to be written once. We also call these functions when restarting the PWM signals when the RESUME button is pressed.

We had already enabled WiFi on the Raspberry Pi when we initially set up the system, so the final step of this lab was to configure the Pi to launch run_test.py at boot. We did this by adding the line `python /home/pi/Documents/ECE-5725-Labs/Lab3/run_Test.py` to the /home/pi/.bashrc file. We also set the Pi to automatically log in to the Pi user without entering the username and password by changing the corresponding setting in the configuration settings using sudo `raspi-config`. That way, as soon as the Pi is powered on and boots, run_test.py immediately runs with no user input at all. The results are shown back in Figure 20 above.

Conclusion

Much of this lab was straightforward. We were given the step-by-step instructions on building the robot and everything, so that wasn't difficult (although it might have been fun to build our own robot, we realize that's not what this lab is about). We did have some trouble understanding the exact requirements for the final demo, however. During the demo, one of the TAs informed us that we needed to have the state timer pause while the robot's wheels were disabled with the STOP button. While we didn't have the state transitions continuing, we did have the timer still running, so the state would immediately transition to the next one (since more than a second or half-second had typically passed while paused). This was a fairly quick change, but we still would have appreciated that clarification in the lab handout. We also had some trouble lining up the circle in the center of the screen for the STOP/RESUME button. We think the PyGame documentation for that function might be wrong or there's something else we're not understanding. Overall, this was a good lab, though. We enjoyed it and were able to learn more about PyGame and practice Python programming with different data structures.

Code

Code stored on server at /home/Lab3/acn55_eb654_Lab3