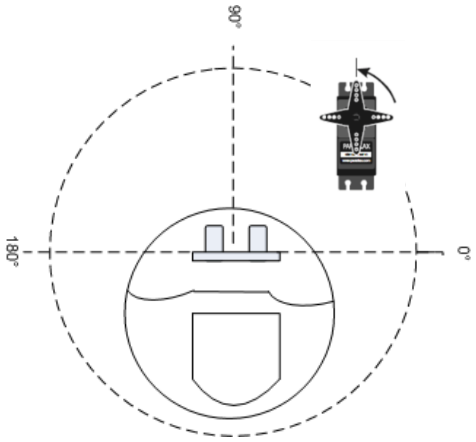
Introduction

In this lab you will be exploring the control of a servo motor using pulse-width modulation (PWM). By learning how to control the servo motor, you will be able to position the sensors already used in prior labs. This will be useful for analyzing the surrounding environment.

About the Servo Motor

A servo motor is an electro-mechanical device with built-in position feedback. The device is commonly used in remote control devices and robotics and comes in two configurations: (1) one has a mechanical stop that restricts the servo to 180 degrees of motion, and (2) another allows 360 degrees of rotation. The servo that we will be using in lab has 180 degrees of rotation.



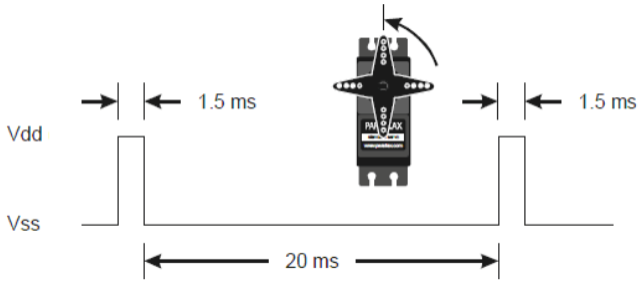
The main feature of a servo is the ability to position the device at an absolute position by signaling the control circuit with a pulse train (periodic digital waveform of pulses) consisting of a positive pulse having a specific width.

Optionally, the servo has feedback circuitry that keeps the servo in a specified position. This feedback is established by a pulse train with a constant pulse width. For example, if the servo were used to control a rudder or wing flap position, forces would push the controlled surface out of position unless there is feedback compensating for the forces. In our lab setup, the turret mounted on the servo will experience minimal forces that would move the turret out of position, so the feedback feature is not necessary.

Positioning the Servo Motor

The servo motor works by rotating to an absolute position using a control circuit driven by a pulse train that has a specified period and positive (high) pulse width. It typically expects to see a pulse every 20 milliseconds, and the length of the pulse determines how far and in which direction the motor turns.

Because the servo must be positioned by generating a waveform that has events at precise times, the microcontroller has special hardware in the timer module, called output compare or pulse-width modulation, to generate an output waveform having preset timing parameters. The output waveform is generated continuously.



If the pulse is high for **1 millisecond**, then the servo angle will be **0 degrees**.

If the pulse is high for **1.5 milliseconds**, then it will be at its **center position of 90 degrees**.

If the pulse is high for **2 milliseconds**, it will be at **180 degrees**.

Pulse widths between **1 ms and 1.5 ms** will position the motor to **between 0 to 90 degrees**, and pulse widths between **1.5 ms and 2 ms** will position the motor **between toward 90 and 180 degrees**.

A servo has a **linear relationship (y = mx+b) between timer count value and servo position** in units of degrees. By determining the timer count values for 0 degrees and 180 degrees, a conversion factor can be used to convert a desired degree position into its corresponding counter value. The conversion factor is the ratio of the difference in counts over the range of motion in degrees.

****Note:** Testing will show that pulse widths translate to different absolute positions for each servo. You should document which CyBot you are using and develop a method to make adjustments if a different device needs to be used.

Reference files

The following reference files will be used in successful completion of this lab:

1. lcd.c, program file containing various LCD functions
2. lcd.h, header file for lcd.c
3. Timer.c, program file containing various wait commands
4. Timer.h, header file for timer.c
5. TM4C123GH6PM Datasheet
6. Servo Datasheet

In addition to the files that have already been included for you, you will need to write your own **servo.c** file and **servo.h** file and the associated functions for setting up and using PWM mode on the GPTM module. Separate functionalities should be in separate functions for good coding quality and reusability purposes. This means that in your servo.c file, you should write separate functions for initializing/configuring the servo motor and positioning it. Remember to use good naming conventions for function names and variables. For example, you may want to name your servo initialization function **servo_init** and name your function to move the servo **servo_move** given other initialization functions that will eventually have to be used together. Minimally, we recommend defining the following functions:

```
void servo_init(void);
int servo_move(float degrees);
```

Recommend Readings: You are encouraged to read about the servo motor and browse through the reference and resource files. You are also encouraged to review the GPTM registers needed for this lab. You have previously used most of these registers in Lab 8, but you will set up PWM mode in this lab. A new register for this lab is the GPTMTnMATCHR match register. Refer to the Tiva datasheet, in particular, Figure 11-5 and section 11.4.5. See also sections 9.2.3.7 and 9.2.8.3 in the textbook. Figure 9.5 in the textbook is the same as Figure 11-5 in the datasheet.

Part 1: Moving the Servo Using PWM

Develop an API that positions the servo to 0 degrees using a counter value. The position may not be true 0 degrees, however, that is okay for this part of the lab.

There is no way to know programmatically whether the servo has reached its new position. Positioning the servo 120 degrees away from its current position will take longer than positioning 10 degrees away. Your function should return a value indicating the servo has reached its new position. This will require you to judge how long to safely wait prior to returning from this function. Keep in mind being overly conservative will slow down servo movement sequences and the speed at which you can take sensor readings in new directions. However, being overly optimistic may result in sensor measurement errors, since you may be getting sensor data before the sensor has reached its new position.

In this lab, the PWM control signal for the servo is obtained using **Timer 1B**, which is connected to **GPIO Port B pin 5 (PB5)** and the "SERVO" header on the baseboard of the CyBot. **Timer1B is a 16-bit timer**, and in **PWM mode**, the **8-bit prescaler** register is used as an **extension of the timer**, creating a **24-bit timer** (most-significant bits are in the prescaler register). **The timer is configured as a 24-bit count-down counter in PWM mode. You will use it in periodic mode.**

1. The counter outputs a high PWM signal when it starts from a 24-bit start value loaded into the **GPTMTnILR** and **GPTMTnPR** load registers (most-significant bits are in the prescaler register).

2. It stays high until the current counter value equals a **match value** that is stored in the **GPTMTnMATCHR** and **GPTMTnPMR** match registers.

3. It then goes low and stays low until the counter reaches 0.

4. Once the counter reaches 0, it reloads the start value and repeats the cycle from step 1.

The **start value** can be considered to be the **period** of the PWM signal (**in timer tick units**). The **match value** can be considered to be the **falling edge time** of the PWM signal, where the high pulse stops and the low pulse starts. So, **the match value is the width of the low pulse** for the PWM signal. The **duty cycle** of the PWM signal is the **width of the high pulse divided by the period**. Formulas and an example of these calculations using the timer registers are given in section 9.2.8.3 of the textbook.

Refer to **Figure 9.15** of the textbook for a functional diagram. You should determine appropriate start and match values for the counter. **Section 11.4.5 PWM Mode in the Tiva Datasheet** outlines the steps necessary for configuring the timer module in PWM mode. See also sections 9.2.3.7 and 9.2.8.3 in the textbook. Figure 9.5 in the textbook is the same as Figure 11-5 in the datasheet.

Draw your own version of datasheet Figure 11-5 (textbook Figure 9.5) for this part of the lab, in which you are generating a PWM signal for a servo position of 0 degrees. What are the start and match values? Remember that the TM4C has a 16 MHz system clock. Display this waveform using the PicoScope as well.

Checkpoint: Demonstrate that you can move the servo to 0 degrees. It is not required that the servo is positioned exactly at 0 degrees as you will calibrate the servo later. Also explain your drawing to your mentor, and the PicoScope waveform.

Part 2: Calibration and position Control

Expand your API for full position control and calibration of the servo motor.

Your program must:

1. Dynamically adjust the match value, which consequently changes the widths of the low and high pulses. The servo will then be positioned relative to the new duty cycle. Let's refer to the match value as the servo position counter value.

2. Display the position counter value on the LCD.

3. Use the pushbuttons to adjust the position counter value to move the servo, as follows:
    a. SW1: Move the servo 1 degree
    b. SW2: Move the servo 5 degrees
    c. SW3: Switch between clockwise and counterclockwise movement of the servo
    d. SW4: Move servo to 0 degrees, when in clockwise mode (to 180, when in counter clockwise mode).

4. Initially position the servo to 90 degrees, and initialize its movement direction toward higher degrees.

5. Indicate on the LCD whether the program is rotating toward higher or lower degree positions.

6. Operate within the bounds of 0 – 180 degrees.

7. Identify position counter values corresponding to 0, 45, 90, 135, and 180 degrees.

Use a protractor (provided or printed) in order to determine actual position counter values for each angle.

****Note:** Testing will show that pulse widths translate to different absolute positions for each servo. You should document which CyBot you are using, and develop an approach for quickly calibrating a new CyBot's servo.

Checkpoint: Demonstrate to your mentor that you have successfully met all of the requirements listed above.

Part 3: Integrate PING, and Servo functionality into your Extended Simple Mission

Congratulation! With the completion of this lab you will have implemented all the primary features of the cyBot_Scan library. In previous labs, you updated your Extended Simple Mission lab to use your own UART, and ADC functions. In this part of the lab, you will additionally use your own **PING**, and **Servo** functionality.

Completion of this part of the lab will result in your Extended Simple Mission lab only using of your own developed code (i.e. you are not using the cyBot_Scan, or the UART library we have provided in the past).

Checkpoint: Demonstrate to your mentor that your Extended Simple Mission lab works using your own UART, ADC, PING, and Servo functions (i.e. no provided UART or cyBot_Scan library functions used).

Part 4: Combining Simple GUI with starter Client Socket code

Motivation: The motivation for this part of the lab is to provide you with a solid framework that you can extend to create a user interface that is customized for the problem your team is solving for your class project.

You will combine aspects of your previous labs where you created and/or used programs to 1) move the Cybot, 2) request scan data from the Cybot, 3) graphically display scan data, and 4) created a simple Graphical User Interface (GUI).

The result will be a framework you can use as a **starting point** in your class project for creating a user interface that is customized to the problem you are solving, and corresponding Users you have identified for your project.

Review the starter GUI Socket program:
i) The code: Review the provided starter programs (**simple-GUI-sensor-Socket-or-UART-client.py** & **simple-sensor-mock-Cybot-Socket-server.py** - found on the resource section of the course website) with your lab partner. **Give a brief summary** of the steps the starter programs are taking.
ii) Network Sockets: For a reminder on sockets, review the following Network Socket overview link given within the program (https://realpython.com/python-sockets/) with your lab partner.

Running the GUI socket program without a CyBot:
i) Server: Start the Server Program (**simple-sensor-mock-Cybot-Socket-server.py**). Pretends to be the Cybot.
ii) GUI Client: Start the GUI Client Program (**simple-GUI-sensor-Socket-or-UART-client.py**).
iii) Scan: The "Scan" button sends a request to the Server (i.e., mock Cybot) for sensor data.

Modify the GUI Client socket code:
i) Buttons for driving: Add four buttons to the GUI to drive the Cybot: forward, backward, turn left, turn right. Pressing a button should send a command to the Cybot (**hint**: see how the Scan button was implemented).
ii) Scan: Clicking the "Scan" button should send a scan request to the Server, and result in the Client receiving mock sensor data. Modify the code to graphically display the received sensor data.
ii) Key Press: Allow keypresses to also be used to drive the Cybot. See the following links to get started:

- Very direct keypress example: https://www.youtube.com/watch?v=D9YVHm8DIys
- An example with a bit more explanation: https://www.youtube.com/watch?v=8mZ9lZlsDHY

Use the GUI Client socket code to control the Cybot: Instead of having the Client communicate with the given Server program, instead have it communicate with your CyBot.
i) Client: Update your client to use the **IP address** and **Port number** that you have used for PuTTy in the past.
ii) Cybot: Use your Cybot program to receive commands from the Client program for driving the Cybot, and collecting scan data

- **Note:** Your Cybot code needs to be compatible with the Client (or you need to modify the Client). So first review, and use the **Simple-CyBot-echo.c** C program (found on the resource section of the course website) to communicate between the Client and Cybot. Then modify your Cybot code based on this example code to make it compatible with the Client program.

iii) Drive the Cybot: Use your GUI to manually drive your Cybot (using GUI buttons, and Keypresses).
iii) Graphically display scan data: Use your GUI to display the scan data received from the Cybot when the Scan button is pressed.

Checkpoint: Give a brief explanation of your GUI/Client socket code to your Mentor. Demonstrate using your GUI to drive the CyBot (using Buttons, using Keypresses), collect scan data, and graphically display scan data.