M E M O R A N D U M

To: Dr. Charlie Refvem

From: Kyle Schumacher
Toby Sagi

Date: February 17, 2025

Subject: Line Following and Closed Loop Control

Memo

In this lab, we built upon the task state paradigm implemented in Lab 3 to include closed loop control of the Romi robot to follow a circular black line on a white background. To do this, we took data from the motor encoders and an IR sensor and created a nested control loop. The structure and implementation is described in the next section.

The inner loop uses the encoders to spin at constant velocity using a custom proportional, integral, and derivative loop PID() class. The class is modified to include feedforward gain and offset. For both motors, we used the feedforward gain and offset instead of a derivative gain. We can do this because, in Lab 2, we found the relationship between duty cycle and rpm, in addition to the minimum duty cycle to spin a motor. The feedforward “speeds up” the loop by immediately giving a starting duty cycle without requiring large integral error to accumulate. This function is implemented in the Task 2: Left Motor Controller and Task 3: Right Motor Controller. Attachment 1 shows the motor control loop.

The outer loop uses the IR sensor to add a bias to the left and right motor controller tasks. We purchased a Pololu QTR 8 mm x 4 Reflectance Sensor, and wrote an ir() class to update the four analog sensors, calibrate, and calculate the centroid of the black line. Task 4: Line Follower is responsible for implementing an IR object, calibrating, and following the line. To follow the line, the centroid is calculated, where the output is a value between -1 and 1 each corresponding to an edge of the sensor. Since a max velocity is determined in the scheduler, a gain is calculated and multiplied to the motors’ maximum velocity. The result is a sharp turn to keep the centroid of the line in the center of the IR sensor. The new values of max velocity and put into the share and sent to the motor controller tasks. This is effectively just an open loop proportional controller. Attachment 2 shows the IR control loop.

Diagrams

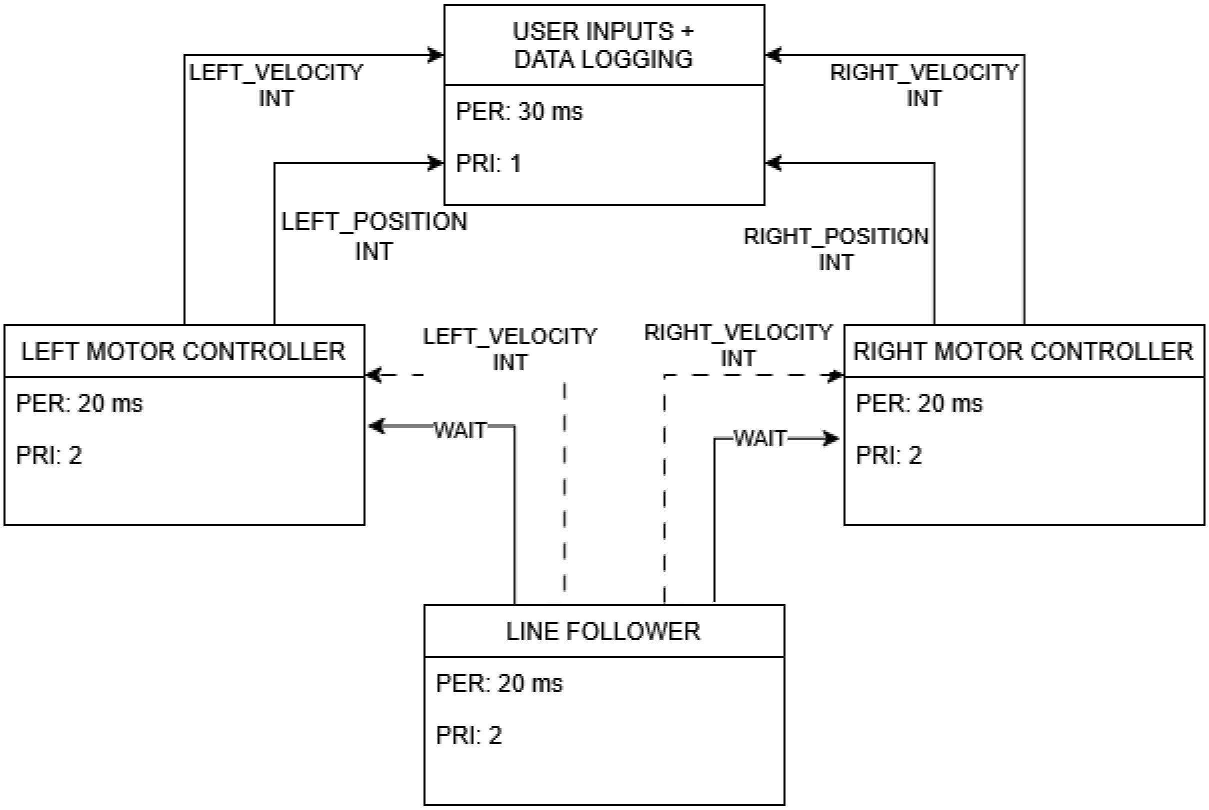


Figure 1. Task Structure

Table 1. Task descriptions

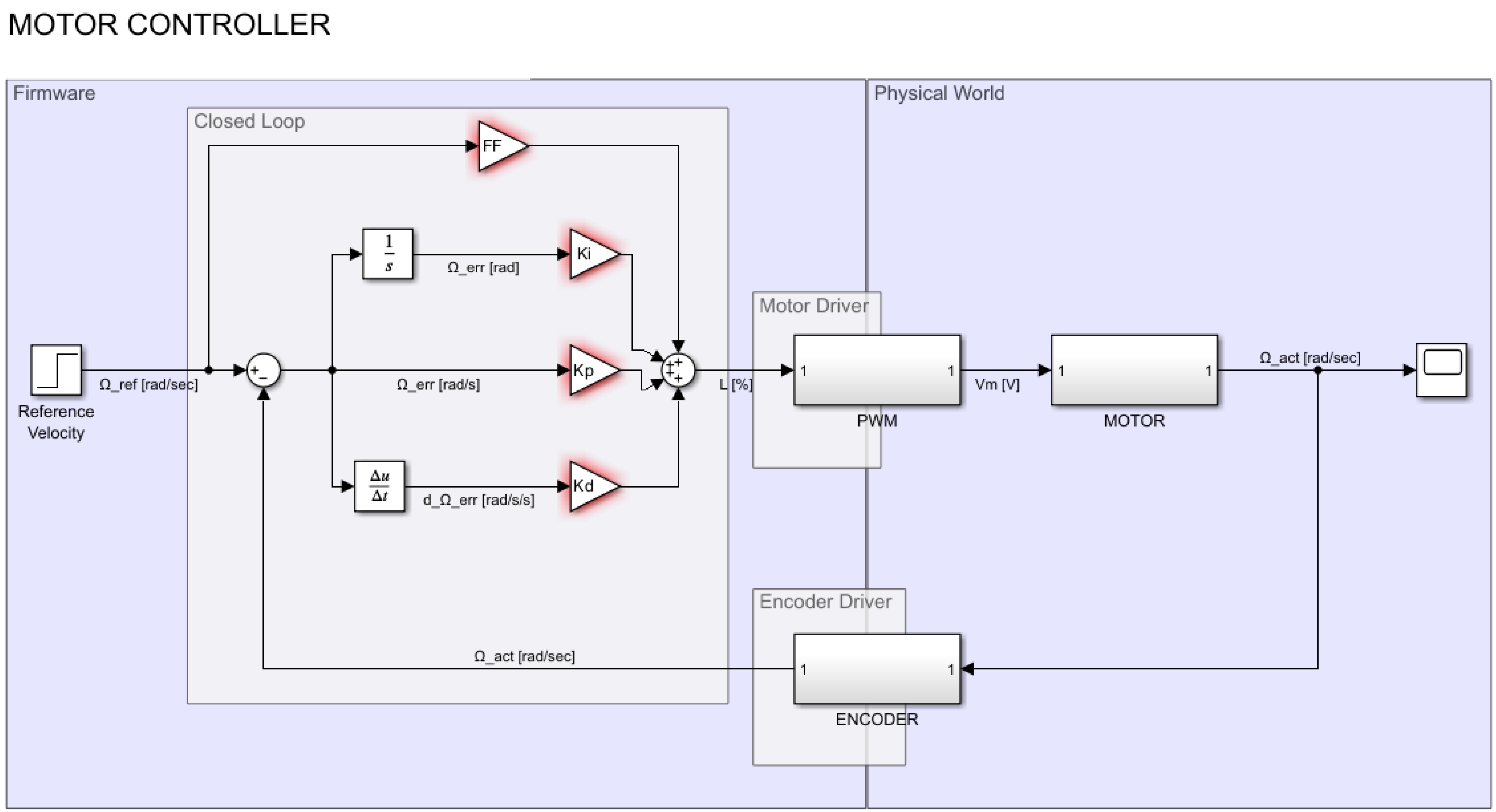
Task #	Task Name	Use Case
1	Data Logging and User Control	Manages user-defined motor control parameters such as effort and run duration. Stores and processes encoder position and velocity data using queues. Outputs collected data for analysis
2	Left Motor Control	Initializes motor and encoder objects. Enables motor operation based on the share of motor velocity. Implements a PID object with feedforward to control the motor.
3	Right Motor Control	Mirrors the left control task structure. Independently handles motor actuation and encoder feedback for the right motor
4	Line Follow	Tracks the centroid of the line to and sets the desired motor velocities. Changes the share variable of pause until the sensor is calibrated, and sets the stop variable to allow a user to reset the program.

Shared Variables

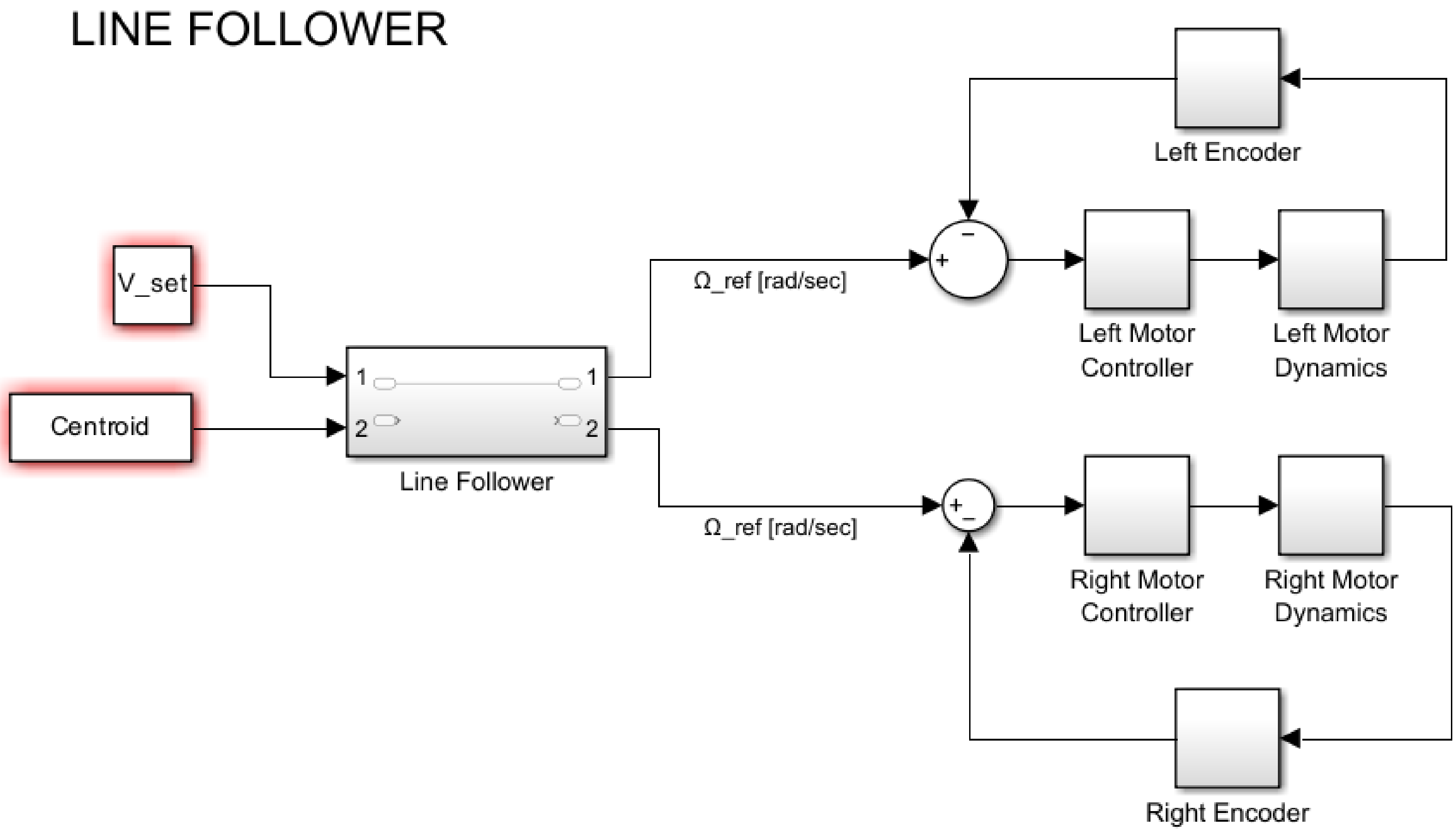
Table 2. Shared Variables and Queues

Variable Name	Type	Purpose
share_motor_controller_period	Int Share	Stores the period for the interval of motor controller updates.
share_left_effort & share_right_effort	Int Share	Stores motor effort percentages for each motor.
queue_left_position & queue_right_position	Long Int Queue	Logs motor position readings over time.
queue_left_velocity & queue_right_velocity	Flat Queue	Tracks velocity data for motor behavior analysis.
Share_max_vel	Int Share	Stores the maximum velocity setpoint
Share_right_vel & share_left_vel	Int Share	Stores the setpoint for the motor velocities from the line follower task
Share_cur_left_vel & share_cur_right_vel	Int Share	Shares the current velocities for both the right and left motors
Share_wait	Int Share	Allows the line follower to keep the other tasks in standby mode while the sensor is being calibrated
Share_stop	Int Share	Allows the line follower to stop the current run once the button is pressed again

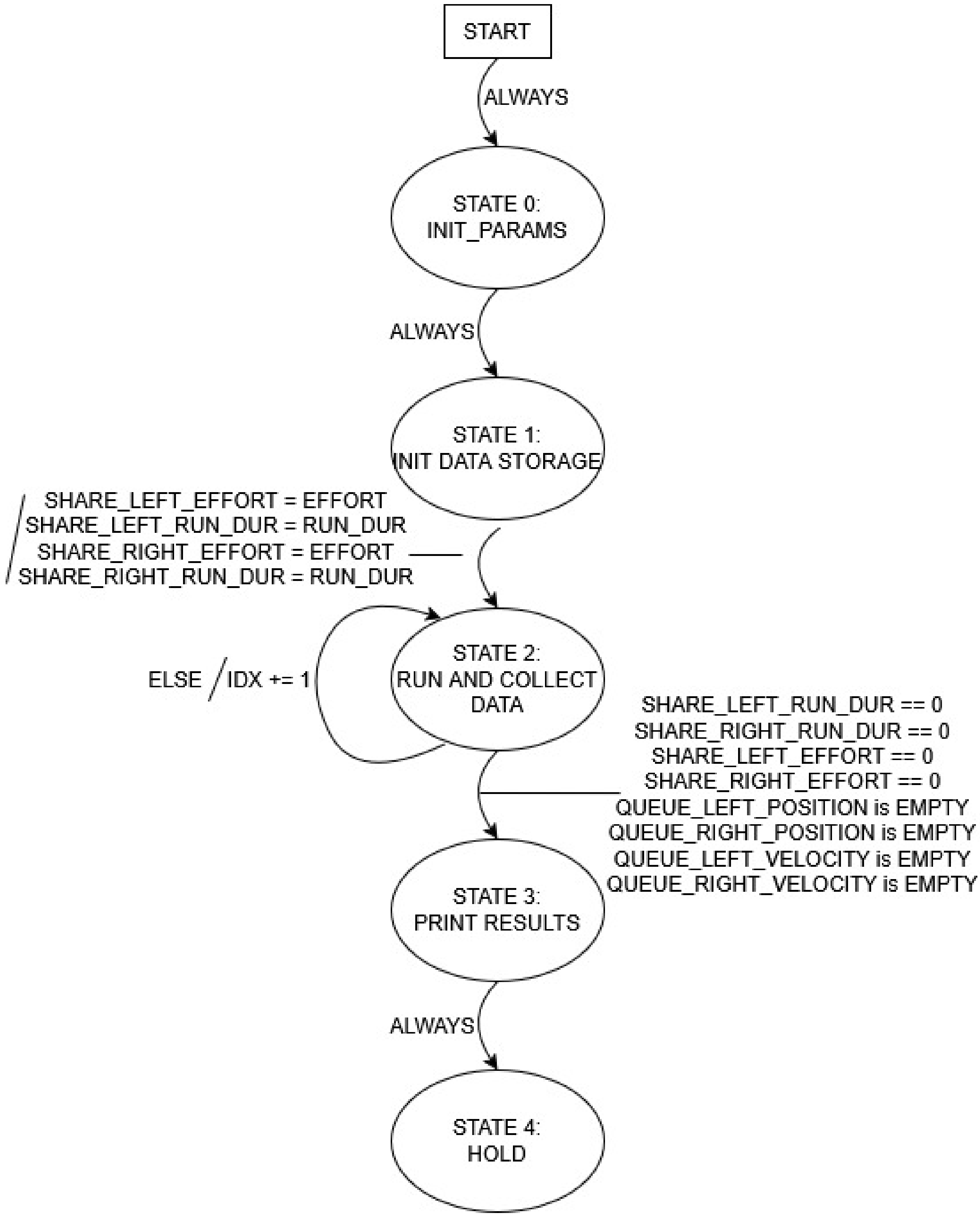
Attachment 1



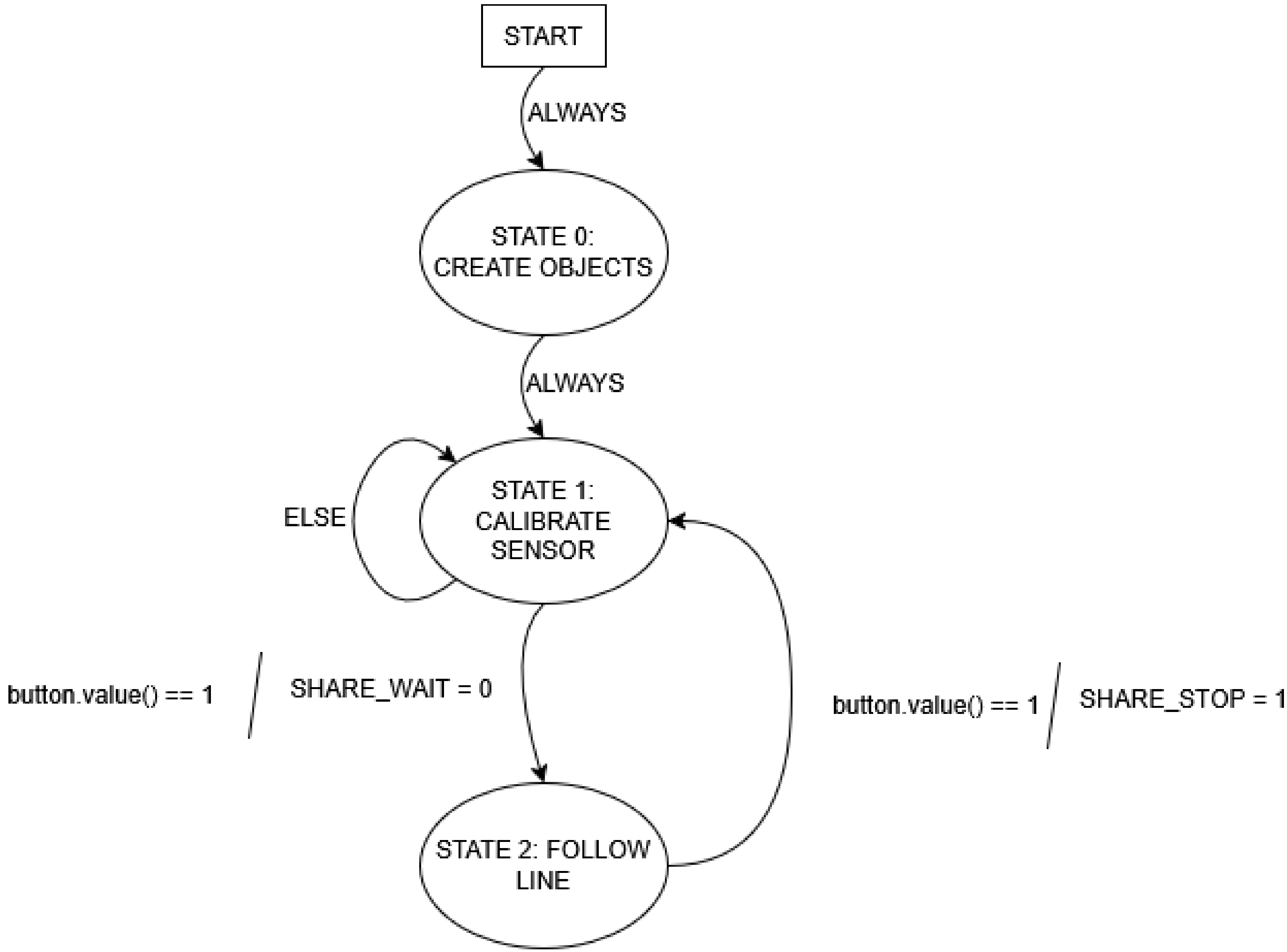
Attachment 2



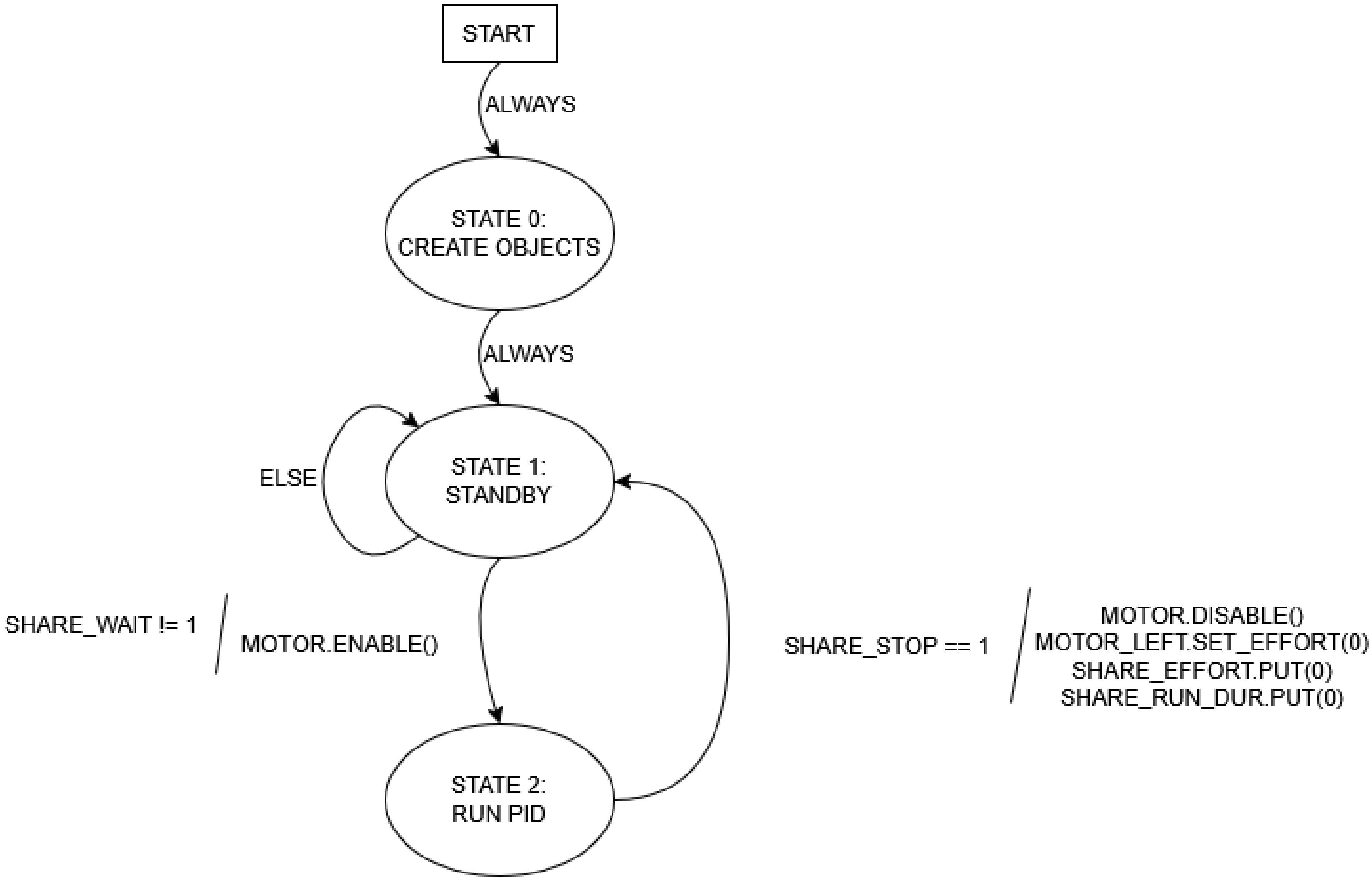
Attachment 3. Main



Attachment 4. Line Follow



Attachment 5. Motor Controller



Attachment 6. Source Code

The source code is only for the new classes (PID and ir) and "main" file.

```
from time import ticks_us, ticks_diff  # Use to get dt value in update
from pyb import Pin, Timer
import cqueue as cqueue

class PID:
  """
  A PID loop encapsulated in a Python class
  """

  def __init__(self, Kp, Ki, Kd, feedforward_gain=0, feedforward_offset=0):
    """
    Initializes gains and integral error
    """
    self.Kp = Kp
    self.Ki = Ki
    self.Kd = Kd
    self.ff_gain = feedforward_gain
    self.ff_offset = feedforward_offset
    self.integral_error = 0
    self.prev_error = 0

  def update(self, desire, actual):
    """
    Generates a reponse
    """
    error = desire - actual
    self.integral_error += error
    derivative = error - self.prev_error
    self.prev_error = error
    dir = 1 if desire > 0 else -1 if desire < 0 else 0
    return self.Kp * error + self.integral_error * self.Ki + self.Kd * derivative + self.ff_gain *
desire + (self.ff_offset * dir)

import pyb
from pyb import Pin, Timer, ADC
from array import array
```

```python
from pin_definitions import *
from time import sleep_ms

class IR:
  """
  A motor driver interface encapsulated in a Pytohn class. Works with
  motor drivers using separate PWM and direction inputs such as the DRV8838
  drivers present on the ROMI chassis from Polulu.
  """

  def __init__(self, ODD, PINS):
    """
    Initializes an IR object
    """
    self.ODD_pin = Pin(ODD, mode=Pin.OUT_PP)
    self.ODD_pin.high()

    if len(PINS) != 4:
        raise ValueError("PINS list must have exactly 4 elements")

    PIN1_pin = Pin(PINS[0], mode=Pin.IN)
    PIN3_pin = Pin(PINS[1], mode=Pin.IN)
    PIN5_pin = Pin(PINS[2], mode=Pin.IN)
    PIN7_pin = Pin(PINS[3], mode=Pin.IN)

    self.IR1 = pyb.ADC(PIN1_pin)  # ADC has max value 4095
    self.IR3 = pyb.ADC(PIN3_pin)
    self.IR5 = pyb.ADC(PIN5_pin)
    self.IR7 = pyb.ADC(PIN7_pin)

    self.values = array('f', len(PINS) * [0])
    self.min_val = 0
    self.max_val = 5.5

    self.pins = len(PINS)

  def update(self):
    """
    Updates the data in the array values
    """
```

```python
    self.values[0] = (self.IR1.read() - self.min_val) / (self.max_val - self.min_val)
    self.values[1] = (self.IR3.read() - self.min_val) / (self.max_val - self.min_val)
    self.values[2] = (self.IR5.read() - self.min_val) / (self.max_val - self.min_val)
    self.values[3] = (self.IR7.read() - self.min_val) / (self.max_val - self.min_val)

  def calc_centroid(self):
    """
    Enables the motor driver by taking it out of sleep mode into brake mode
    """
    self.update()
    sum_values = 0
    moment = 0
    for i in range(self.pins):
      moment += self.values[i] * i
      sum_values += self.values[i]

    # Returns a value from -1 to 1 depending on where the centroid of the line is
    if sum_values > 0:
      return moment / sum_values / ((self.pins-1) / 2) - 1
    return 0

  def calibrate_black(self):
    """
    Disables the motor driver by putting it into sleep mode
    """
    self.values[0] = self.IR1.read()
    self.values[1] = self.IR3.read()
    self.values[2] = self.IR5.read()
    self.values[3] = self.IR7.read()
    self.max_val = sum(self.values) / len(self.values)

  def calibrate_white(self):
    """
    Disables the motor driver by putting it into sleep mode
    """
    self.values[0] = self.IR1.read()
    self.values[1] = self.IR3.read()
    self.values[2] = self.IR5.read()
    self.values[3] = self.IR7.read()
    self.min_val = sum(self.values) / len(self.values)
```

```python
if __name__ == "__main__":
   my_IR = IR(ir_ctrl, [ir_1, ir_3, ir_5, ir_7])
   input("Hit enter to calibrate black")
   my_IR.calibrate_black()
   print(my_IR.max_val)
   input("Hit enter to calibrate white")
   my_IR.calibrate_white()
   print(my_IR.min_val)
   print(f"HIGH: {my_IR.max_val}, LOW: {my_IR.min_val}")
   sleep_ms(5000)
   while True:
      my_IR.update()
      print(f"1: {my_IR.values[0]}, 2: {my_IR.values[1]}, 3: {my_IR.values[2]}, 4: {my_IR.values[3]}")
      centroid = my_IR.calc_centroid()
      print(f"Centroid: {centroid}")
      sleep_ms(500)

import gc
import pyb
from array import array
from time import ticks_ms, ticks_diff, ticks_add

import lib.cotask as cotask
import lib.task_share as task_share

from lib.pin_definitions import *
from lib.encoder import Encoder
from lib.motor import Motor
from lib.pid import PID
from lib.ir import IR

def task1_main(shares):
  """!
  Task which controls user inputs and date logging
  @param shares A list holding the share(s) and queue(s) used by this task
  """
  # Get references to the share(s) and queue(s) which have been passed to this task
```

```
  share_motor_controller_period, share_left_effort, share_right_effort, share_left_run_dur,
share_left_run_dur, queue_left_position, queue_right_position, queue_left_velocity,
queue_right_velocity, share_cur_left_vel, share_cur_right_vel = shares

  # State machine variables
  state = 0
  S0_INIT_PARAMS = 0
  S1_INIT_DATA_STORAGE = 1
  S2_RUN_AND_COLLECT_DATA = 2
  S3_PRINT_RESULTS = 3
  S4_HOLD = 4

  effort = 0
  run_dur = 0

  while True:
    if state == S0_INIT_PARAMS:
      print("T1 S0")
      effort = 25
      run_dur = 2_000
      state = 1
      yield (state)

    elif state == S1_INIT_DATA_STORAGE:
      print("T1 S1")
      points = int(run_dur / share_motor_controller_period.get())+1
      data_L_pos = array('H', points * [0])
      data_R_pos = array('H', points * [0])
      data_L_vel = array('f', points * [0])
      data_R_vel = array('f', points * [0])
      idx = 0
      share_left_effort.put(effort)
      share_right_effort.put(effort)
      share_left_run_dur.put(run_dur)
      share_right_run_dur.put(run_dur)
      state = 2
      yield (state)

    elif state == S2_RUN_AND_COLLECT_DATA:
      # print("T1 S2")
```

```
      if (share_left_run_dur.get() == 0 and share_right_run_dur.get() == 0 and
          share_left_effort.get() == 0 and share_right_effort.get() == 0 and
          queue_left_position.empty() and queue_right_position.empty() and
          queue_left_velocity.empty() and queue_right_velocity.empty()):
        # Run completed and data collected
        state = 3
        yield (state)
      else:  # Motor(s) running and/or data needs processing
        # print(f"idx: {idx}")
        # data_L_pos[idx] = queue_left_position.get()
        # data_R_pos[idx] = queue_right_position.get()
        # data_L_vel[idx] = queue_left_velocity.get()
        # data_R_vel[idx] = queue_right_velocity.get()
        idx += 1
        state = 2
        yield (state)

    elif state == S3_PRINT_RESULTS:
      print("T1 S3")
      for i in range(len(data_L_vel)):
        print(f"{i / points * (run_dur / 1000)}, {data_L_pos[i]}, {data_R_pos[i]}, {data_L_vel[i]},
{data_R_vel[i]}")
      state = 4
      yield (state)

    elif state == S4_HOLD:
      yield (state)

    else:  # Error state
      state = 1
      raise RuntimeError("State machine in task2_motor_controller is in an illegal state. Resetting to
state 1.")

def task2_left_motor_controller(shares):
    """!
    Task which controls the left motor using PID velocity control
    @param shares - a tuple of shares and queues for motor control
    """
    # Get references to the share(s) and queue(s)
    share_run_dur, share_left_vel, share_max_eff, share_cur_left_vel, share_wait, share_stop = shares
```

```
    # State machine variables
    state = 0
    off_time = 0
    S0_CREATE_OBJECTS = 0
    S1_STANDBY = 1
    S2_RUN = 2

    while True:
        if state == S0_CREATE_OBJECTS:
            Motor_Left = Motor(motor_left_enable, motor_left_dir, motor_left_timer, motor_left_channel,
motor_left_pwm)
            Encoder_Left = Encoder(encoder_left_timer, encoder_left_A, encoder_left_B)
            # Create PID controller with tuned gains
            PID_Motor_Left = PID(.2, 0.05, 0, 100 / 34.8 / 7.2, 0.44 / 7.2 * 100)
            state = 1
            yield (state)

        elif state == S1_STANDBY:
            if share_wait.get() == 0:
                print("The wait is over!")
                Motor_Left.enable()
                state = 2
                yield (state)
            else:
                state = 1  # Stay in current state
                yield (state)

        elif state == S2_RUN:
            # Update encoder and store data
            # print("T2 S2")
            Encoder_Left.update()
            current_velocity = Encoder_Left.get_velocity()
            share_cur_left_vel.put(current_velocity)


            # Calculate control effort using PID
            effort = PID_Motor_Left.update(share_left_vel.get(), current_velocity)
            print(f"LEFT: Desire: {share_left_vel.get()}, Actual: {current_velocity}, Effort: {effort},
Error: {PID_Motor_Left.prev_error}, Integral_Error: {PID_Motor_Left.integral_error}")
```

```
            # print(f"Left Vel Des: {share_left_vel.get()}")
            # Constrain effort to valid range
            effort = max(min(effort, share_max_eff.get()), 0)
            Motor_Left.set_effort(effort)

            if share_stop.get() == 1:
                Motor_Left.disable()
                Motor_Left.set_effort(0)
                share_run_dur.put(0)
                share_left_vel.put(0)
                # state = 1
                yield (state)
            else:
                state = 2  # Stay in current state
                yield (state)

        else:  # Error state
            state = 1
            print("T2 Error")
            raise RuntimeError("State machine in task2_left_motor_controller is in an illegal state. 
Resetting to state 1.")

def task3_right_motor_controller(shares):
    """!
    Task which controls the right motor using PID velocity control
    @param shares - a tuple of shares and queues for motor control
    """
    # Get references to the share(s) and queue(s)
    share_run_dur, share_right_vel, share_cur_right_vel, share_wait, share_stop = shares

    # State machine variables
    state = 0
    off_time = 0
    S0_CREATE_OBJECTS = 0
    S1_STANDBY = 1
    S2_RUN = 2

    while True:
        if state == S0_CREATE_OBJECTS:
```

```
            Motor_Right = Motor(motor_right_enable, motor_right_dir, motor_right_timer,
motor_right_channel, motor_right_pwm)
            Encoder_Right = Encoder(encoder_right_timer, encoder_right_A, encoder_right_B)
            # Create PID controller with tuned gains
            PID_Motor_Right = PID(.2, 0.05, 0, 100 / 33.7 / 7.2, 0.49 / 7.2 * 100)
            state = 1
            yield (state)

        elif state == S1_STANDBY:
            if share_wait.get() == 0:
                print("The wait is over! but for the right motor")
                Motor_Right.enable()
                state = 2
                yield (state)
            else:
                state = 1  # Stay in current state
                yield (state)

        elif state == S2_RUN:
            # Update encoder and store data
            Encoder_Right.update()
            current_velocity = Encoder_Right.get_velocity()
            share_cur_right_vel.put(current_velocity)
            # queue_position.put(Encoder_Right.get_position())
            # queue_velocity.put(current_velocity)

            # Calculate control effort using PID
            effort = PID_Motor_Right.update(share_right_vel.get(), current_velocity)
            print(f"RIGHT: Desire: {share_right_vel.get()}, Actual: {current_velocity}, Effort:
{effort}, Error: {PID_Motor_Right.prev_error}, Integral_Error: {PID_Motor_Right.integral_error}")
            # print(f"Right Vel Des: {share_right_vel.get()}")
            # Constrain effort to valid range
            effort = max(min(effort, 100), -100)
            Motor_Right.set_effort(effort)

            if share_stop.get() == 1:
                Motor_Right.disable()
                Motor_Right.set_effort(0)
                share_run_dur.put(0)
                share_right_vel.put(0)
```

```
                # state = 1
                yield (state)
            else:
                state = 2  # Stay in current state
                yield (state)

        else:  # Error state
            state = 1
            raise RuntimeError("State machine in task3_right_motor_controller is in an illegal state.
Resetting to state 1.")

def task4_line_follower(shares):
    """!
    Task which reads IR sensors and controls motor velocities for line following
    @param shares A list holding the share(s) used by this task
    """
    # Get references to the share(s)
    share_max_vel, share_left_vel, share_right_vel, share_cur_left_vel, share_cur_right_vel, share_wait
= shares

    # State machine variables
    state = 0
    S0_INIT = 0
    S1_CALIBRATE = 1
    S2_FOLLOW_LINE = 2
    button = pyb.Pin(blue_button, pyb.Pin.IN, pyb.Pin.PULL_UP)


    while True:
        if state == S0_INIT:
          # Create IR sensor object
          ir_sensor = IR(ir_ctrl, [ir_1, ir_3, ir_5, ir_7])
          state = S1_CALIBRATE
          share_wait.put(1)
          yield(state)

        elif state == S1_CALIBRATE:
            # Calibrate IR sensors
            for i in range(100):
                yield(state)
```

```
        print("Press the blue button to calibrate black")
        while button.value() == 1:
            yield(state)
        ir_sensor.calibrate_black()
        for i in range(100):
            yield(state)
        print("Press the blue button to calibrate white")
        while button.value() == 1:
           yield(state)
        ir_sensor.calibrate_white()

        for i in range(100):
          yield(state)

        state = S2_FOLLOW_LINE
        share_wait.put(0)
        yield(state)

    elif state == S2_FOLLOW_LINE:
        # Get centroid position (-1 to 1)
        centroid = ir_sensor.calc_centroid()

        # Get max velocity from share
        max_vel = share_max_vel.get()

        # Calculate motor velocities based on centroid
        # When centroid is 0, both motors run at max_vel (straight)
        # When centroid is -1, right motor runs at max_vel, left at -max_vel (sharp left turn)
        # When centroid is 1, left motor runs at max_vel, right at -max_vel (sharp right turn)
        # print(f"Centroid: {centroid}")
        left_vel = max_vel * (1 + centroid)
        right_vel = max_vel * (1 - centroid)

        if left_vel > max_vel:
            left_vel = max_vel
        if right_vel > max_vel:
            right_vel = max_vel

        if left_vel < 0:
            left_vel = 0
```

```
            if right_vel < 0:
                right_vel = 0

            # Update velocity shares
            share_left_vel.put(left_vel)
            share_right_vel.put(right_vel)
            print(f"Left Vel: {left_vel}, Right Vel: {right_vel}")

            if button.value() == 0:
               share_stop.put(1)

            state = S2_FOLLOW_LINE
            yield(state)

        else:  # Error state
            state = S0_INIT
            raise RuntimeError("Invalid state in task4_line_follower")

# This code creates a share, a queue, and two tasks, then starts the tasks. The
# tasks run until somebody presses ENTER, at which time the scheduler stops and
# printouts show diagnostic information about the tasks, share, and queue.
if __name__ == "__main__":
  print("Testing Lab_0x03 Motor Controllers with Multitasking using cotask.py and task_share.py\r\n"
        "Press Ctrl-C to stop and show diagnostics.")

  # Create a share and a queue to test function and diagnostic printouts
  share_motor_controller_period = task_share.Share('i', thread_protect=False,
name="Share_Left_Motor_Controller_Period")
  share_motor_controller_period.put(20)
  share_left_effort = task_share.Share('i', thread_protect=False, name="Share_Left_Effort")
  share_left_effort.put(0)
  share_left_run_dur = task_share.Share('i', thread_protect=False, name="Share_Left_Run_Dur")
  share_left_run_dur.put(5000000)
  share_right_effort = task_share.Share('i', thread_protect=False, name="Share_Right_Effort")
  share_right_effort.put(0)
  share_right_run_dur = task_share.Share('i', thread_protect=False, name="Share_Right_Run_Dur")
  share_right_run_dur.put(5000000)
  q_len = 500
  queue_left_position = task_share.Queue('l', q_len, thread_protect=False, overwrite=False,
name="Queue_Left_Position")
```

```
  queue_left_velocity = task_share.Queue('f', q_len, thread_protect=False, overwrite=False,
name="Queue_Left_Velocity")
  queue_right_position = task_share.Queue('l', q_len, thread_protect=False, overwrite=False,
name="Queue_Right_Position")
  queue_right_velocity = task_share.Queue('f', q_len, thread_protect=False, overwrite=False,
name="Queue_Right_Velocity")

  share_max_vel = task_share.Share('f', thread_protect=False, name="Share_Max_Vel")
  share_max_vel.put(50)

  share_max_eff = task_share.Share('f', thread_protect=False, name="Share_Max_Eff")
  share_max_eff.put(70)

  share_right_vel = task_share.Share('f', thread_protect=False, name="Share_Right_Vel")
  share_right_vel.put(0)
  share_left_vel = task_share.Share('f', thread_protect=False, name="Share_Left_Vel")
  share_left_vel.put(0)
  share_dir = task_share.Share('i', thread_protect=False, name="Share_Dir")
  share_dir.put(0)

  share_cur_left_vel = task_share.Share('f', thread_protect=False, name="Share_Cur_Left_Vel")
  share_cur_left_vel.put(0)
  share_cur_right_vel = task_share.Share('f', thread_protect=False, name="Share_Cur_Right_Vel")
  share_cur_right_vel.put(0)

  share_wait = task_share.Share('i', thread_protect=False, name="Share_Wait")
  share_wait.put(1)

  share_stop = task_share.Share('i', thread_protect=False, name="Share_Stop")
  share_stop.put(0)

  # Create the tasks. If trace is enabled for any task, memory will be
  # allocated for state transition tracing, and the application will run out
  task1 = cotask.Task(task1_main, name="Task_1", priority=1, period=30,
                      profile=True, trace=False,
                      shares=(share_motor_controller_period, share_left_effort, share_right_effort,
share_left_run_dur, share_left_run_dur,
                              queue_left_position, queue_right_position, queue_left_velocity,
queue_right_velocity, share_cur_left_vel, share_cur_right_vel))
```

```
  task2 = cotask.Task(task2_left_motor_controller, name="Task_2", priority=2,
period=share_motor_controller_period.get(),
                      profile=True, trace=False,
                      shares=(share_left_run_dur, share_left_vel, share_max_eff, share_cur_left_vel,
share_wait, share_stop))
  task3 = cotask.Task(task3_right_motor_controller, name="Task_3", priority=2,
                    period=share_motor_controller_period.get(),
                    profile=True, trace=False,
                    shares=(share_right_run_dur, share_right_vel, share_cur_right_vel, share_wait,
share_stop))
  task4 = cotask.Task(task4_line_follower, name="Task_4", priority=2, period=20,
                    profile=True, trace=False,
                    shares=(share_max_vel, share_left_vel, share_right_vel, share_cur_left_vel,
share_cur_right_vel, share_wait))
  cotask.task_list.append(task1)
  cotask.task_list.append(task2)
  cotask.task_list.append(task3)
  cotask.task_list.append(task4)

  # Run the memory garbage collector to ensure memory is as defragmented as
  # possible before the real-time scheduler is started
  gc.collect()

  # Run the scheduler with the chosen scheduling algorithm. Quit if ^C pressed
  while True:
    try:
      cotask.task_list.pri_sched()
    except KeyboardInterrupt:
      break

  # Print a table of task data and a table of shared information data
  print('\n' + str(cotask.task_list))
  print(task_share.show_all())
  print('')
```