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**

**A diagram of a computer program

AI-generated content may be incorrect.**

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**

**A computer screen shot of a diagram

AI-generated content may be incorrect.**

**Attachment 2**

**A diagram of a computer flowchart

AI-generated content may be incorrect.**

**Attachment 3.** Main

**A diagram of a flowchart

AI-generated content may be incorrect.**

**Attachment 4.** Line Follow

**A diagram of a diagram

AI-generated content may be incorrect.**

**Attachment 5.** Motor Controller

**A diagram of a process

AI-generated content may be incorrect.**

**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

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

    """

    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)

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('')