PS03: Multi-Threaded Robot Programming in C

Due: March 26, 2014

Robot systems need to sense, process, and act on information in real-time, with strict timing guarantees. For this assignment, you will experiment with the multi-threaded kernel on the rone robots, and then build two fun behaviors in C: Avoid obstacles, and wall-follow.

1 Threads, Mutexes, and Messages

Make a two-thread program based on `tbd.c`. Each thread should print a unique string 10 times. Something like, “Hippopotamus” and “Platypus”. Do not use any yield or delay functions from the FreeRTOS API. Use `cprintf()` to print over the serial port.

1. Make thread 1 a higher priority than thread 2. Capture the output and hand in.
2. Make the two threads the same priority. Capture the output and hand in.
3. Make a new function, `serial_send_string_mutex()`. Use a mutex to ensure that only one thread can print at a time. Capture the output and hand in.

Make a three-thread program and a message queue. Thread 1 and 2 should put 10 total messages on the queue, one every 0.5 second. The messages should be pointers to the strings from above, and use a different string for each thread. Thread 3 should read the queue and print the message. You will need to read about how to implement periodic threads in the FreeRTOS book.

1. Make thread 1, 2, and 3 the same priority. Capture the output and hand in.

2 Obstacle Detection and Wall Following

Make a new program based on `tbd.c`. Use a background thread to read the obstacle detector with the `irRangeGetBits()` function.

1. Make a `obstacleAngleCompute()` function that takes the obstacle bits and computes the direction of the obstacle. Refer to the `process_nbr_message()` function for inspiration on computing direction from bits. Note that you will potentially need to deal with obstacles on many different sides of the robot.
2. Make a `avoidObstacles()` function that takes the obstacle angle and steers the robot away from obstacles. Put this function into a program to make the robot wander around the environment.
3. Make a `followWall()` function that takes the obstacle angle and drives the robot along a wall. Print the turning angle around corners to the console.

3 Orbit

Make a new program that creates an orbit task. Select a leader out of the neighboring robots using the robot id. The leader stays in-place while the other robots rotate around the leader. The orbiting robot should follow a circle centered at the leader with the distance from the robot to the leader as the radius. The turning angle is then the bearing of the robot turned 90 degrees clockwise. $turning\_angle = bearing - \frac{\pi}{2}$

4 Tree Navigation

Create a self-stabilizing tree that allows the leaf nodes to navigate to the root node.

4.1 Write self_stabilizing_tree function

Select the robot with the lowest id as the root node. The remaining robots should select the neighbor with the smallest number of hops to the root as their parent. Break ties between each robot by choosing the robot with the lowest id.

Use the led lights to mark the number of hops away from the root node. Turn on all R,G,B leds for the root node. Use the circle red led pattern when the robot does not have any neighbors. Use the following binary encoding to mark number of hops away from the root.

1. Blue - 1 hop
2. Green - 2 hops
3. Green, Blue - 3 hops
4. Red - 4 hops
5. Red, Blue - 5 hops
6. Red, Green - 6 hops
7. Red, Green, Blue - Root Node

4.2 Write tree_navigation function

Select a robot in the tree using a button push. The robot will then use the tree to navigate to the root. The robots should update their hop count and parent nodes, allowing the tree to self-stabilize.

5 Hand-In / Check-Off

1. **Hand-in:** Your traces from Section 1.
2. **Check-off:** Your `avoidObstacles()` and `followWall()` function in operation.
3. **Check-off:** Your `orbit()` function in operation
4. **Check-off:** Your `self_stabilizing_tree()` and `tree_navigation()` functions in operation