WPI

NSF Workshop
Where are we headed?

Last modification: January 4, 2022

Workshop Note:
As we build on the low-level functionality in previous activities, the project becomes more open ended as to how a student team might solve the problems herein. We provide room to explore ideas, but also solutions that work well for demonstrating basic functionality. We leave additional tasks as exercises, with ideas for how to solve them. We expect to take time in the workshop to discuss some of the solutions, and we provide pointers on how they can be implemented.

1 Introduction

By now, you have most of the functionality for *physically* delivering a bag for the final challenge: you can detect and lift a bag, follow a line, detect intersections and turn – so long as you have your remote control in your hand, you're all set! (OK, we haven't covered dropping off a bag, yet, but we'll handle that below.)

Alas, you are not allowed to use the IR remote to control your robot for the final demo. You have to program it to deliver bags *autonomously*. To make that happen, your Romi will need to know how to get from one point to another – i.e., how to *navigate* the course. In contrast to the low-level control for line following or speed control, navigation requires higher-level planning and it is that high-level planning that sets a robot apart from an "ordinary" electromechanical device.

1.1 Objectives

To successfully complete this lab, the student will:

- Enable the robot to drop off a bag, and
- Add autonomy to the robot by navigating a map.

2 Overview

2.1 Problem statement

Your goal for this lab is to program your robot to drive from the bag pick-up to a delivery platform, where the specific platform is indicated by a button on your IR remote (e.g., '1' for delivery zone

A, etc.). You will not simply program a chain of commands – you will represent the course using an *internal map* of the arena.

3 Resources

3.1 Materials

- Your Romi, with all sensors and actuators mounted,
- A "bag," and
- A test arena (see the Build Instructions for the layout).

4 Navigation

To actually deliver a bag, you'll need to add a function to physically drop off the bag. Before you fill in that detail, however, you will add *autonomy* to your robot – instead of responding to commands from the IR remote, the robot will keep track of its location so that every time it reaches an intersection, it can determine which direction to turn.

The general approach will be to keep track of which *segment* of the course the robot is on and then respond to intersections to decide what the next step should be. We'll then work out the logic of how to get from one point to another by figuring out the correct series of turns.

Figure 1 shows a labelled version the course.

Start by opening the `activity04.navigation` skeleton project. Note that this project includes an additional header file that defines a new structure, `Delivery`, which holds three variables to help with navigation. These variables will be defined as new datatypes, which will allow us to use more natural language to make the code more readable. Referring to the file, `delivery.h`, we've defined,

- `currLocation`, which is of type `Road`. `currLocation` keeps track of the current segment that the robot is on.
- `deliveryDest`, which is of type `Destination`. `deliveryDest` holds the location of where the current delivery is to be made.
- `currDest`, which is also of type `Destination`. `currDest` holds the immediate destination, which may be the pickup zone or a return to the start, in addition to the delivery location.

> **Workshop Note:** Depending on the level of programming skills of the students, you can have them develop more or less of the code. In our introductory course, many students do not have substantial programming experience, so we feed them lots of skeleton code here and 'backfill' coding skills in later classes.

The navigation algorithm will be implemented in the function, `handleIntersection()` in `main.cpp`. In the previous activities, the code in `handleIntersection()` simply commanded your robot to stop at an intersection, but now it will contain the logic for determining which way to go.

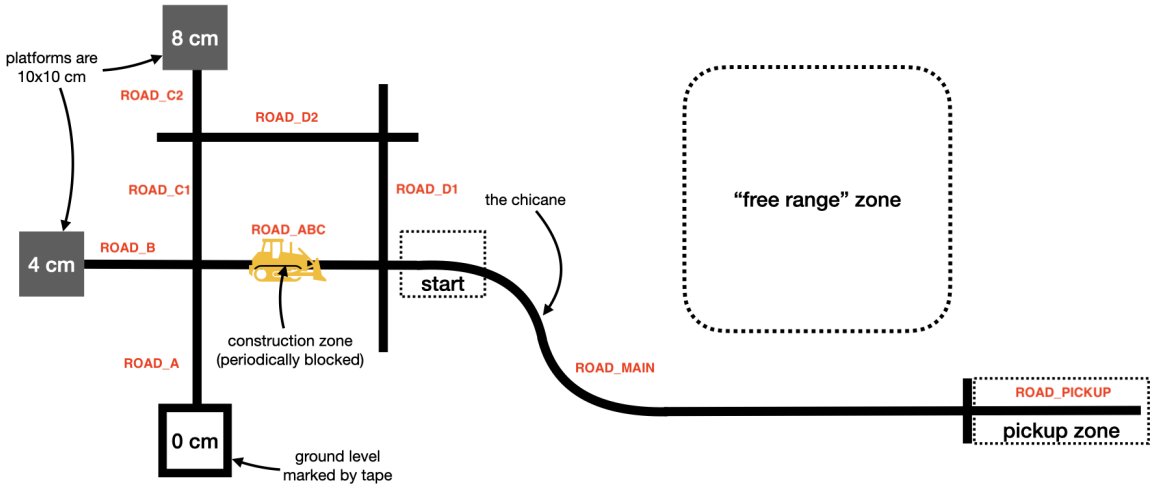

Figure 1: Arena with roads annotated.

The first thing to notice is that we added a couple of lines near the top of the function that center your robot on the intersection. That is, whenever it reaches the intersection, the first thing the robot does is drive forward a short distance (8 cm in our example, but you can adjust this, if needed). That way, the center of the robot will be roughly over the center of the intersection, which will make turning much easier. For better or worse, these lines of code are *blocking*, in that they wait for the robot to center itself before moving on. You could make centering a state of its own, but the code becomes significantly more complicated, so we've gone with the simpler approach here.

Next, the function uses a `switch` statement to determine which way to turn. the `switch` statement only runs the code that corresponds to the current location of the robot. Within each `case`, the code checks the current destination using `if..else` statements to determine what the next step should be.

4.1 A single delivery

The skeleton version has the code needed to make a pickup, but not a delivery. Here, we'll add code to deliver to House B.

Workshop Note: We provide programming resources for students to learn about the `switch` statement. If you need a refresher, here is a refresher on the syntax.

Procedure

1. Add code to `handleKeyPress()` so that when you press Button 2 on the IR remote, it will start the robot on its delivery. Your code should:

   - Set the delivery destination to `HOUSE_B`,
   - Set the current destination to `PICKUP`, and
   - Start the robot delivering by calling `beginLineFollowing()`.

   Remember that these variables are all part of the global structure, `delivery`, which needs to prefix the data member, e.g.,

   ```
   delivery.currDest = PICKUP;
   ```

   Solution: A good solution looks like:

   ```
   if(keyPress == NUM_2)
   {
       delivery.deliveryDest = HOUSE_B;
       delivery.currDest = PICKUP;
       beginLineFollowing();
   }
   ```

2. Test your robot by placing it near the start and pressing the number 2 on your remote. Your robot should drive to the pickup stripe, execute a pickup, do a U-term and then stop when it gets to the pickup stripe.

   Examining the code, we see that `handleIntersection()` only handles one case, namely when the robot detects an intersection while it is driving on `MAIN`. In this case it initiates the pickup sequence. After the pickup is done, when it drives back towards the start, it is on `PICKUP` – which is unhandled. Let's remedy that.

3. Add another `case` to handle an intersection when the current location is `ROAD_PICKUP`. In this case, the robot should simply continue onward, so we need to do two things:

   (a) Update `currLocation` to `ROAD_MAIN`, and

   (b) Add a statement to begin line following.

   Warning! You must close every `case` with a `break;` statement, otherwise, the program will continue to the next case, which is generally bad.

   If you run the code now, the robot will continue line following, intersection after intersection. The reason is because `case ROAD_MAIN` only does anything if `currDestination == PICKUP`. But if you glance at `pickupBag()`, you'll see that the current destination is updated to the location of the delivery. So, let's handle the scenario where it is headed to a house.

4. Add an `else` statement to `case ROAD_MAIN` to handle the intersection that leads to the houses. (You could use a bunch of `else..if` statements for each house, but in this case, a single `else` will do the trick. You will need to update `currLocation` and command the robot to line follow again.

Solution: A good solution is:

```
case ROAD_MAIN:
    if(delivery.currDest == PICKUP)
    {
        delivery.currLocation = ROAD_PICKUP;
        beginBagging();
    }

    else
    {
        delivery.currLocation = ROAD_ABC;
        beginLineFollowing();
    }

    break;
```

5. Add a `case` to handle the next intersection. In this instance, the robot's next action will depend on the destination: turn left to get to A; go straight to get to B; turn right to get to C. For now, just implement B by updating `currLocation` and commanding the robot to `beginDropping()`. We've added that function for you, but all it does at the moment is call `idle()` – not what we want in the long run, but valuable for testing what you have so far.

Solution: A good solution is:

```
        case ROAD_ABC:
            if(delivery.currDest == HOUSE_A) {} //filled in later

            else if(delivery.currDest == HOUSE_B)
            {
                delivery.currLocation = ROAD_B;
                beginDropping();
            }

            else if(delivery.currDest == HOUSE_C) {} //filled in later

           break;
```

6. Test your system. Set it near the start and press the 2 on your IR remote. Does it drive to the correct place? If not, do some troubleshooting – we have plenty of pointers if you need them!

5 Dropping off

Before you can deliver a bag, there is one last function that needs to be completed, namely dropping off a bag. Dropping off a bag is not terribly different from picking one up, but it is complicated by the fact that there are three delivery zones, each at a different height.

Workshop Note: The complication can be removed, of course, if you choose to simplify the problem for an introductory class. One way to scale the class is to have all of the delivery options the same to start and then challenge the advanced students with more complicated arrangements.
Another simplification is to put a piece of tape a short distance in front of each drop zone. Once the tape is detected (with `checkIntersection()`), a function you already have, then dead reckoning can be used to align with the drop zone.

5.1 Dropping off a bag

The general plan will be to align the robot with the drop zone, lower the arm to the appropriate level, back up a few cm (to avoid getting the arm caught on the handle), and then spin 180 degrees.

Procedure

1. Define a new state, `ROBOT_DROPPING`.

   Solution: For example,

   ```
   enum ROBOT_STATE {ROBOT_IDLE, ROBOT_DRIVE_FOR, ROBOT_LINE_FOLLOWING,
                     ROBOT_BAGGING, ROBOT_DROPPING};
   ```

2. Edit `beginDropping()` to mimic the `beginPickup()` functionality. You will also need to comment out the call to `idle()`.

   Solution: For example,

   ```
   void beginDropping(void)
   {
       robotState = ROBOT_DROPPING;
       baseSpeed = 5;
   }
   ```

3. Add a code to the `loop()` to mimic the actions when in the `ROBOT_BAGGING` state. You can reuse `checkBagEvent()`, but you'll want your code to call `dropOffBag()`.

Solution: For example,

```
case ROBOT_DROPPING:
  handleLineFollowing(baseSpeed); //crawl towards bag
  if(checkBagEvent(8)) {idle();}//{dropOffBag();}
  break;
```

4. Create a new function, `dropOffBag()`. Because the drop platforms are at different heights, it will need to have a set of conditionals to determine where the drop is being made. See the `pickupBag()` function for pointers and ideas.

   You will need to define a target position for the servo that corresponds to the height of the platform at delivery zone B.

   At the end of the function, you'll need to set the current destination to return to the start.

Solution: For example,

```
void dropOffBag(void)
{
    Serial.print("Dropping...");

    if(delivery.deliveryDest == HOUSE_A) {} //to be filled in later

    // For B and C, we need to drive forward a bit
    else if(delivery.deliveryDest == HOUSE_B)
    {
    Serial.println("Crawling forward.");
    chassis.driveFor(2, 2);
    while(!chassis.checkMotionComplete()) {delay(1);} // blocking

    // Release the bag
    Serial.println("Dropping.");
    servo.writeMicroseconds(SERVO_B);
    delay(500); //blocking, but we need to make sure servo has moved
    }

    else if(delivery.deliveryDest == HOUSE_C) {} // to fill in later

    // Back up a little so the hook clears the handle
    Serial.println("Backing up.");
    chassis.driveFor(-5, 5);
    while(!chassis.checkMotionComplete()) {delay(1);} // blocking
    // Now command a U-turn (needed for all deliveries)
    Serial.println("U-turn");
    turn(180, 45);
 }
```

5. Test your system and adjust the parameters/distances so you can make a reliable delivery. You may want to add a new button press that commands the robot to just do a delivery. How would you do that?

Solution: A good idea is to make button 8 (defined as `NUM_8`) set the location to `ROAD_ABC` and the destinations to `HOUSE_B`. Then you can avoid all of the wait for picking up a bag.

5.2 Return to start

The only thing left to be able to do repeated deliveries (at least to House B) is to return to start.

Procedure

1. Add code to `dropOffBag()` that updates the current destination to `START`. There is already code for resuming line following when the U-turn is done.
2. Add code to `handleIntersection()` that handles the cases needed to get back to the start. Don't forget to call `idle()` when you get back to start.
3. Test the ability to deliver to House B. You may need to do some additional adjustment of the parameters to make it reliable. Once you have a good solution for House B, it's time to add more deliveries.

6 Next steps

Time permitting, we will discuss the remaining tasks:

- Delivering to House C requires mimicking much of the functionality of delivering to House B. You will need to fill in the missing details in `handleIntersection()` and `dropOffBag()`. You will need to call `turn()` instead of `beginLineFollowing()` in a couple of instances, but we hope that is straightforward.
- Delivering to House A is a little more involved. Because there is no platform, you can't use the rangefinder to find the drop zone. Instead, you will probably want to add code to `handleIntersection()` to detect the tape at the front of the box.
- Avoiding the "Construction Zone" requires a test when you reach the intersection between Main Road and Road ABC. If the rangefinder detects an object in the way, turn right onto Road C1.
- Finding the Free Range Bag provides an excellent opportunity for students to generate ideas and show creativity. the best way to initiate the FRB search is to test for an object when reaching the pickup zone – if no bag is detected, go into a new state. From there, we have seen several solutions to the problem, including,
  - Spin slowly and use the rangefinder to "find" the edges of the bag, then center the robot between the two and drive forward. This method benefits from pausing occasionally and reassessing the location of the bag.
  - Spin until the bag is detected and then turn an additional amount (calibrated through testing) to center on the bag.
  - Using a little trigonometry, some teams will keep track of the robot's heading, which allows them to find the line again. Other teams will simply have their robot spin 180 degrees after grabbing the bag. In either event the line sensors can be used to find the line again, though getting aligned with it can be tricky.