Lab 06: Mid-Course Correction

When Little Boe Learns To Back It Up

INTRODUCTION

The last time we saw our little friend BOE, he was lost in a maze. Dead reckoning only gets you so far…like around the block and (hopefully) right back home. The ability to change course without having to be told what to do would absolutely be an advantage.

Without a pair of eyes, how does little BOE know where the walls are? Well, why do cats have whiskers? Why do insects have antennae? Two words: Tactile navigation.

OBJECTIVES

The specific goals for this exercise are:

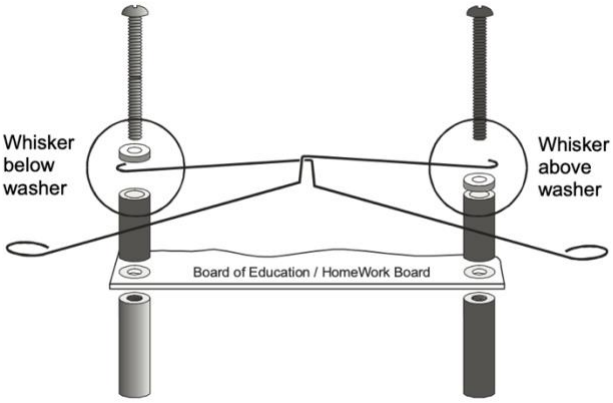
- Install a pair of sensory whiskers to your BOE-Bot
- Test the basic whisker function
- Develop an algorithm to allow tactile navigation
- Program a subroutine that course corrects
- Demonstrate your bot's ability to navigate independently

INSTALL THE HARDWARE

Collect the necessary parts. The parts case has everything you need, and plenty of them. However, you need to be extra-careful selecting your resistors; you have to use the proper resistances, so double-check the color bands!

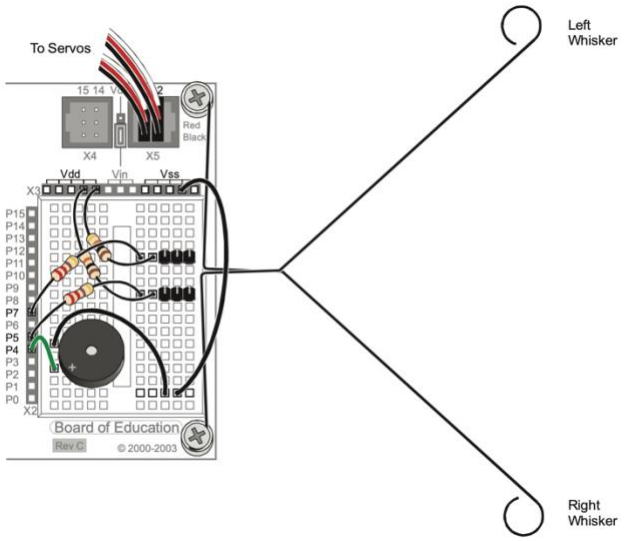
- Two whisker wires
- Two 3-mm pin headers
- Two 220Ω resistors (red–red–brown)
- Two 10kΩ resistors (brown–black–orange)

Mount the whiskers to the front of the bot as shown:



If you need the spacers or the nylon washers (you probably don't), go ahead and grab them from the case.

Once you've got the whiskers attached, use the wiring diagram below to complete the circuit. Notice that the pin headers don't have to go into any specific location, but you want them far enough apart that the left whisker won't hit the right pin or vice-versa. Pay careful attention to which resistor goes where, and where each end is attached!



For now, just ignore the piezo speaker (black disk) and the black and green wires (we'll get to that later).

DEVELOP SOME TEST CODE

First of all, notice that the right whisker touches the pin header wired to P7, and the left whisker touches the pin wired to P5. If the whisker touches the pin, it shorts the circuit. BOE's got a pretty tiny brain, but it can easily keep track of any individual pin's state: 1=on and 0=off. The default state for each pin is on, until a whisker makes contact and shorts the circuit. Here's a short program to just check this out:

```
Program: Is This Thing On?
'{$STAMP BS2}           'Stamp directive
'{$PBASIC 2.5}          'Language directive
' Set up a table to record the whisker states
DEBUG "WHISKER STATES", CR,
    "Left    Right", CR,
    "------  ------"
' Set up a loop to continuously check the
' state of each whisker
' BIN1 = display as binary (1 or 0)
' IN5  = the input register for P5
' IN7  = the input register for P7
DO
 DEBUG CRSRXY, 0, 3,
     "P5 = ", BIN1 IN5,
     "   P7 = ", BIN1 IN7
 PAUSE 50
LOOP
```

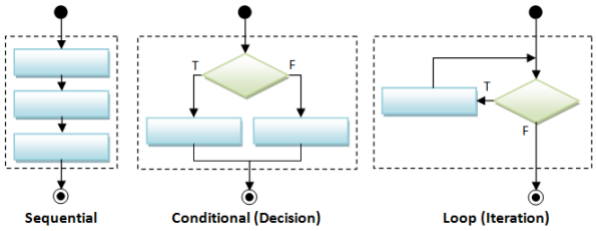
This is not great programming (an infinite loop is a terrible idea!), but all we need to do is make sure that the whiskers are operating as advertised. You are not handing this program in for credit, so you don't need all the comments—but you *do* need to run it.

CONSTRUCT THE ALGORITHM

Okay, now what? We want to program the BOE-Bot to run the maze, but we want it to be able to change course if it hits a wall. That's our most basic algorithm.

The key word there is 'if.' We're going to need another control structure (we've already learned about using loops). Basically, we need a branching structure: if <condition A is true>, do <what needs to be done>. Otherwise <condition A is

false>, so do <a different thing>. Or maybe just continue what you were already doing. Depends, doesn't it?



Let's imagine a hypothetical maze path for the robot. Let's say that the bot needs to start on the launch square, go forward, turn left, go forward, turn right, forward, right, forward, left, and then finally forward until it hits the landing square.

Here's where it gets complicated. If the bot goes too far forward on the first segment, it'll hit a wall when it tries to execute the first turn. What does the bot need to do? Back up? Or does it need to undo the turn before it backs up? Will the right response be the same every time the bot hits a wall?

Also, when we ran the maze, we gave the bot an explicit number of steps to take on each segment. How often do we need to check the whisker states? With every step? Every five steps?

It might be a little too ambitious to try to run the entire maze. So what *can* we do? Let's not try complex navigation yet—let's just see if we can turn BOE loose and keep him from running into stuff. Let's map out what that would look like:

```
If both whiskers sense an obstacle:
  Back up and turn 180°
Else if right whisker only senses an obstacle:
  Back up and turn 90° left
Else if left whisker only senses an obstacle:
  Back up and turn 90° right
Else no whiskers are triggered:
  Roll forward
Loop back to the beginning and repeat
```

Notice the order in which we check the whiskers! Why is the forward roll the *last* thing we do and not the first?

WRITE THE CODE

1. **Add to the existing.** You can continue to add to and modify the dead-reckoning program you already have. Make a copy and rename, preserving the original motion program.
2. **Check your subroutines.** You already have forward, backward, left, and right subroutines. But are you going to use them differently now? Notice that your turns need to be precise (how many ticks make a 90° turn?). Also, how far do you want to back up? Should you be rolling forward more than one pulse at a time?
3. **Construct the conditional.** The main program relies on the conditional IF-THEN structure in the algorithm above to respond correctly to the whisker states. Luckily, the syntax is pretty much exactly what you would expect it to be! Here's a snippet to get you started:

```
' If both whiskers detect an obstacle: state=0
' Bot should roll back and turn 180°
  IF (IN5=0) AND (IN7=0) THEN
    GOSUB Backward    'Bot rolls backward
    GOSUB Left        '90° left turn
' If only right whisker detects an obstacle:
' right state = 0 but left state = 1
' Bot should step back and turn 90° left
  ELSEIF (IN5=0) THEN
    GOSUB Backward    'Bot steps backward
    GOSUB Left.       '90° left turn
' Add an ELSEIF for the case that only the
' left whisker detects an obstacle. Hint: what
' do you copy/paste?
' Then add one last ELSE for the case where
' neither whisker is triggered
' Also, replace this comment block with the
' appropriate annotations
  ENDIF             'Close conditional
```

Once you check for both whiskers being triggered, does it matter if you check the right one before the left one?

4. **Complete the loop.** Use the same Do-Loop structure that we used in the test code. It's not the best structure—an infinite loop with no way to get out is not optimum. However, we don't know anything about the path or course, so we can't pre-program a fixed number of iterations (using a FOR-NEXT loop), and we have no way (yet!) to communicate with the bot to tell it to stop!
5. **Compile and test.** When you have completed and compiled the code, you should have a program that lets the bot stagger around the room and bump into stuff without getting stuck.

MANAGE YOUR EXPECTATIONS

Staggering around the room is…well…*some*thing. Technically, it's called a Drunkard's Walk. But it's nowhere *near* our goal of running the maze with a self-correcting robot. Answer the following questions in a separate document.

6. **Is the existing maze suitable for our goal?** Was it realistic to expect the bot with whiskers to navigate that maze?
7. **Are our skills up to it?** Based on our previous programming experience, were we too optimistic to expect that we could achieve our self-driving goal?
8. **Did we fail?** Even though we did not reach our goal, did we get closer to being able to achieve it? Is it a failure to recognize that you need to take an intermediate step (or a few) before you can expect success?

SAVE AND SUBMIT

Be sure to save your source code frequently! Always save to your own UCA Google drive. If you are using the UCA computers, save the program on the Desktop—but always save it in a second location.

- **Demonstrate your success:** Let little BOE roam and shoot a video! It does not need to be long or fancy, just demonstrate that the bot can successfully avoid an obstacle. Blackboard Assignments supports the submission of multiple video file formats, so you should be able to upload your video with your source code.
- **Use the proper file names:** Whatever you have named your program in the source code, you must use the correct filename for code submission. Name your program lastnameLAB06, obviously using your own last name. It should already/automatically have the .bs2 file extension. Never submit word processor documents or .pdf files!
- **Submit electronically:** Your program, video, and questions are due no later than 6:00 PM on Tuesday, 05 March 2024. You must submit via Blackboard.

GRADING RUBRIC

Your performance will be evaluated using the rubric below:

ASSESSMENT	CRITERIA / VALUE		POINTS EARNED
Program 06: Drunkard's Walk! DUE: Tue 05 Mar 24 (filename: lastnameLAB06.bs2)	Compilation: Program compiles cleanly	4 points	
	Annotation: Program is sufficiently commented	5 points	
	Execution: Program executes correctly (yields correct results)	5 points	
	Questions: Appropriate responses to Questions 6–9	9 points	
	Video: Proper operation is completely documented via video	7 points	