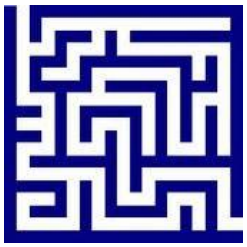
The Basics of Robot Mazes
Teacher Notes

Why do robots solve Mazes?

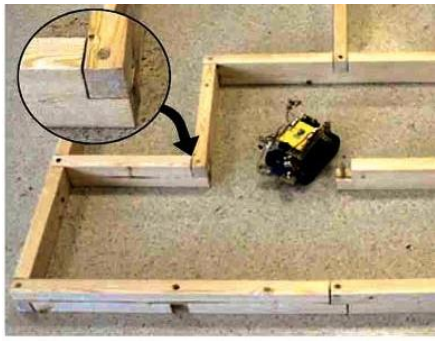


A maze is a simple environment with simple rules.

Solving it is a task that beginners can do successfully while learning the essentials of building and programming robots.


Maze at Hever Castle

Creating a Maze



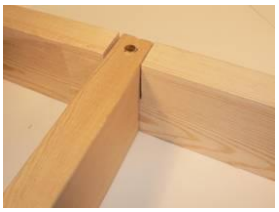
As the teacher, you have to provide a suitable maze. There are many ways you can do it. Any sturdy material can be used. The most common is wood, but bricks, blocks, and books can also work. What ever you use to make the maze, there are some general guidelines to keep in mind.

Scale the maze to the robots. Your maze should have dimensions that allow for a variety of robots to travel through it. The walls should be high enough for sensors to detect them. The travel corridors in the maze should be wide enough for the robots to turn in. We recommend a wall height of 2/3 to ¾ the height of a typical robot and 3 to 4 times a typical robot width. For BotBrain robots, a suitable maze is between 3 and 5 inches high, with a corridor width of 12 to 16 inches.

The maze should be built to be strong enough for the robots to hit the walls, and not push them around too much. If you are using light materials, you might consider taping them to the floor. However, pushing the maze around is another environmental variable that young robot engineers should really have to consider. Any bot that moves the maze is not well designed or programmed and should suffer the consequences. We have seen bots push the maze so far as to block their own paths. We have also seen them push their way out at the end (permitted by our rules). Securing the maze to the floor is not always practical. If it is not secured, we like to put tape marks on the floor and return it to its initial shape before each trial.

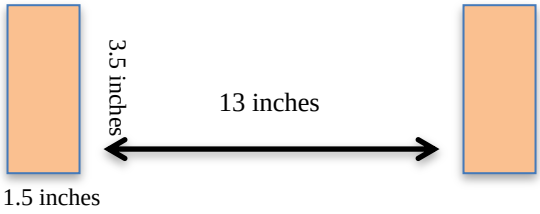
Because not everyone has the time, we have created an excellent Maze Environment, suitable for BotBrain, BoeBot and many other small





robots. You can purchase one from http://botbrain.com (shameless plug!). The BotBrain Maze uses 2x4 lumber that is held together with pegs and holes. It slides together and comes apart easily and is heavy enough to be used without taping down.

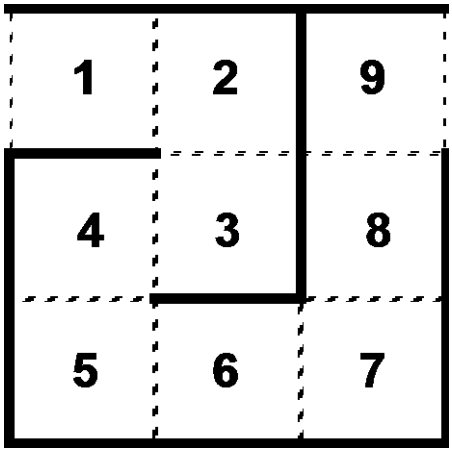
The dimensions of the BotBrain Maze are shown in the diagram below.



Rules

A maze competition is a race, a time trial actually. The goal is to solve the maze in the least amount of time. There are many ways to score it. These are rules that work for us.

1. Each team (robot) has three trials
2. Each trial has a maximum time of three minutes.
3. A judge will time each trial. The timing will start when any part of the robot crosses the start line and end when the entire robot has exited the maze at the other end. The time to finish is the robot's score.
4. In the event that the robot is unable to finish the maze, it will be given a score based on how far it made it into the maze. This is done by counting the number of squares it made it into (all parts of the robot entered). The farthest point is counted. In the maze at right, if a bot made it into square 6, it would receive six points, square 2, 2 points.



5. If a member of the team touches the robot during its trial, then the turn ends at that point.
6. If a member of another team touches a robot or disturbs the maze…**???**
7. All decisions from the judge(s) are final.

Scoring If all or nearly all robots finish, just award places based on time. Average the times or use the best of three. Any bot that finishes beats any one that does not. If there is a tie from one trial, then the result from their second best run can be used to break the tie.
If only a few finish, you can combine distance and times as follows. For each trial, for each robot that finishes, rank the times. The slowest bot that actually finishes gets total maze points plus 1, 10 in the maze shown. If six bots finish, the fastest gets 9+6 = 15. Second place gets 14, etc. Most maze competitions allow groups to modify the robot and/or the program between trials. This emphasizes the iterative, trial-and-error nature of engineering and allows them to recover from simple mistakes or bad luck. Still, the teams that do best have usually done a lot of testing and practicing before the competition begins.

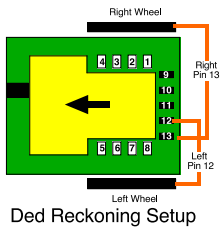
Maze Solving Strategies

One early decision is whether or not to publish the maze layout ahead of time. There are pros and cons to this. If it is known, then there are more strategies that can be used to solve the maze. (These strategies are described later.) Not publishing the exact layout forces teams to make their robots adaptable to any maze layout or program and test quickly. This makes for a more "intelligent" robot and the students learn more. Engineering is a discipline of trade-offs.

There are at least four strategies for solving the maze;

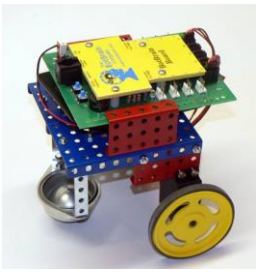
Strategy	Maze Known Ahead of Time	Maze Layout a Surprise
Ded Reckoning	Very fast. Reliability changes with battery voltage.	Must be programmed quickly, may not be time for adequate testing.
Bump and Turn	Very Fast. Reliability changes with battery voltage.	Must be programmed quickly, may not be time for adequate testing.
Dumb Luck	Fairly fast but not always reliable.	Fairly fast but not always reliable.
Right (Left) Hand Rule.	Extremely reliable but slower.	Extremely reliable but slower.

Ded[1] Reckoning



Ded Reckoning Setup

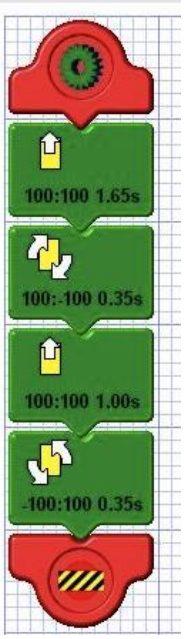
The term DedReckoning comes from how ships and aircraft navigate using only a compass and a clock. They chart a course on a map, sail a certain direction for a given distance and then turn a new heading and sail some distance.
Your robot can solve a maze much the same way, if you know the layout of the maze



[1] Short for Deductive. Often spelled "Dead Reckoning"

ahead of time. You simply program the robot to travel a certain time and turn whichever direction is needed. No sensors are required.
How good is it? It can be very fast. When it works, the bot travels maximum speed the precise distance required and makes all turns the precise angle required. However, the speed of rotation of the servos changes with voltage. So, if the voltage drops as the batteries wear down (which it does) the speed of the robot will change and the distance it travels in the programmed time interval will decrease. The tendency will be to change the time interval to increase the distance and to change the angle of the turns. Installing fresh batteries will make it go farther and turn farther than before. If you choose this strategy, you will have to monitor the voltage very closely.

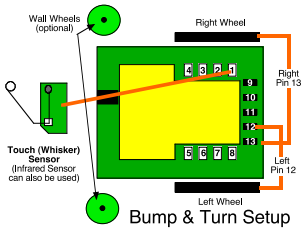
Programming Ded Reckoning: The program is fairly simple since it does not use any sensors. You will make a list of driving commands that will tell the robot how far to drive forward and which direction to turn.



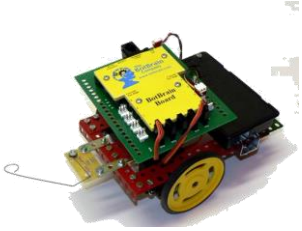
The picture to the right shows one possible start to a program for ded reckoning. You will need to set the times in each of the movement tiles to make the robot get through the maze. Remember to keep those batteries fresh so the timing doesn't change. You will also need to add more tiles for the rest of the maze. This program only has two turns.

When programming for ded reckoning, do it one step at a time. Get the first straight section done first, then add the first turn, then the next straight away, etc. If you try to do the whole thing at once, it will get frustrating and it will be difficult to tell where it went wrong.

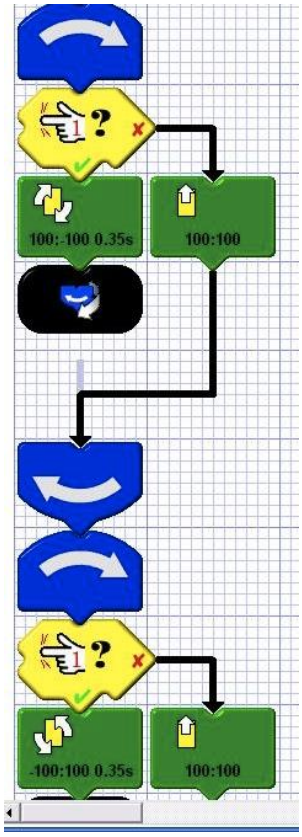
Bump and Turn



Bump and turn is similar to Ded Reckoning except that the distance is not set by the program. Instead, it uses one sensor in the front of the robot. The robot will drive forward until the sensors detect the wall and then turn the needed direction according to the program. A key element to this one is that the turns need to be a perfect 90^{O} so that only front walls are hit and not side ones. Turns are just as critical as inDed Reckoning: as the voltage drops, the amount of turn will decrease. Some teams use 'Wall Wheels' or sliders on the front corners or sides of the robot to guide it along a wall to help ensure that the touch sensor hits only the wall in front. Using Wall Wheels allows the bot to compensate for inaccurate turns.

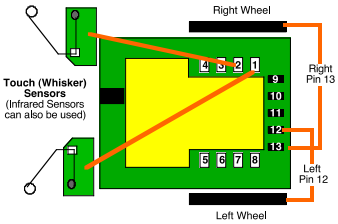


Programming Bump & Turn The program shown could be used as a start to the bump and turn approach. This program shows that the touch sensor in the front of the robot is plugged into pin 1 of the BotBrain Board. The logic is as follows: Has the touch sensors touched something? If not, drive forward and loop to check the sensor again. If the sensor does touch something, then turn (right in this case) and exit the loop. New Loop. Check the sensor. Did it hit something? If not, drive forward, if it did, turn, and exit the loop.



You would need to add additional loops with the needed turns to solve the maze.

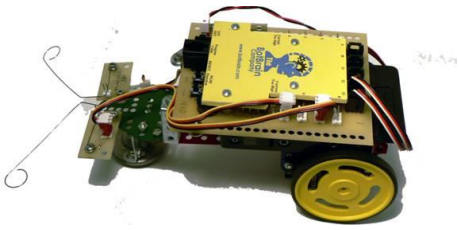
Dumb Luck (Random Turns)



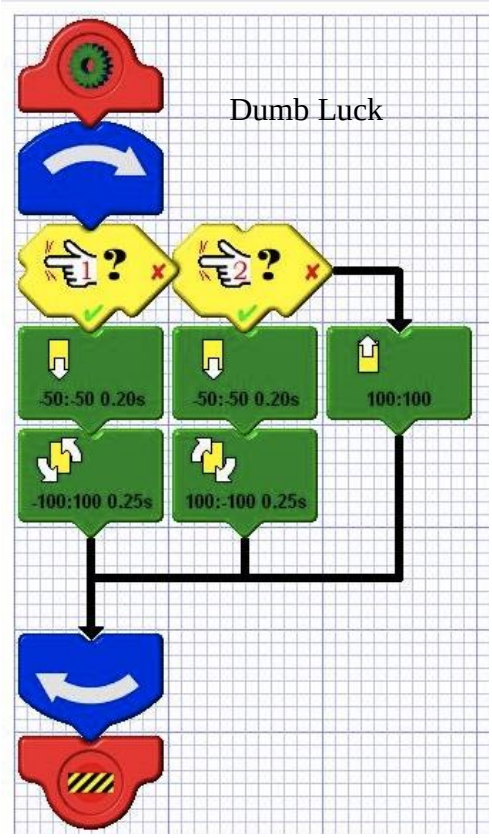
Dumb Luck Setup

The Dumb Luck method is not as bad as it sounds. We call it that because it seems a bit random sometimes. It can be very fast when it works right. Sometimes it will solve the maze, but it can also get stuck in a corner or turn around.
For this method, you will need two whiskers on the front of the robot; one on the left and the other on the right. If the left sensor hits, the robot turns right. If the right sensor hits, the robot turns left.

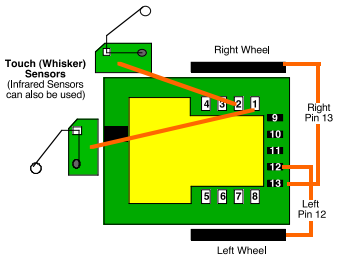
Sometimes a Dumb Luck robot will get stuck in a corner, turning left, touching the left sensor which makes it turn right, touching the right sensor which makes it turn left and on and on. This makes it good to insert some random elements into the program to help break out of this infinite loop.



Programming Dumb Luck This program uses two test condition tiles, one for each sensor. This particular program assumes that the right sensor is in 2, and the left is in pin 1. The logic here is to check the pin one sensor, if it hit something, back up and turn right. If not, check the pin 2 sensor. If it hit something, backup and turn left. If not, drive forward. The trick here is to adjust the time of the turns until the robot makes it through the maze



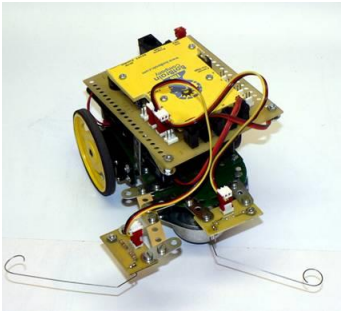
Right (or Left) Hand Rule

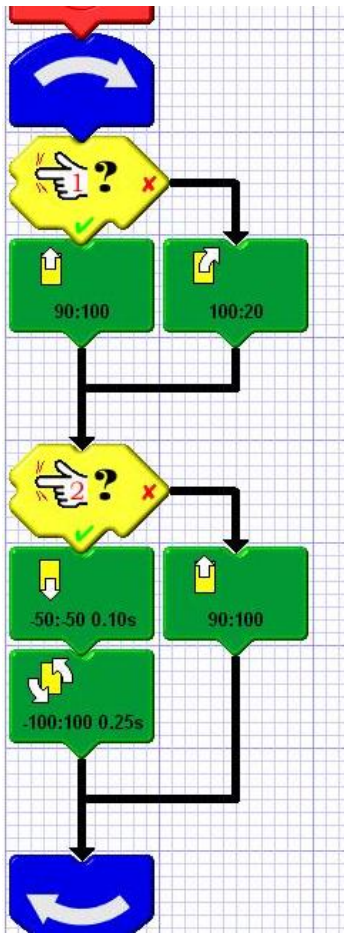


Right Hand Rule Setup

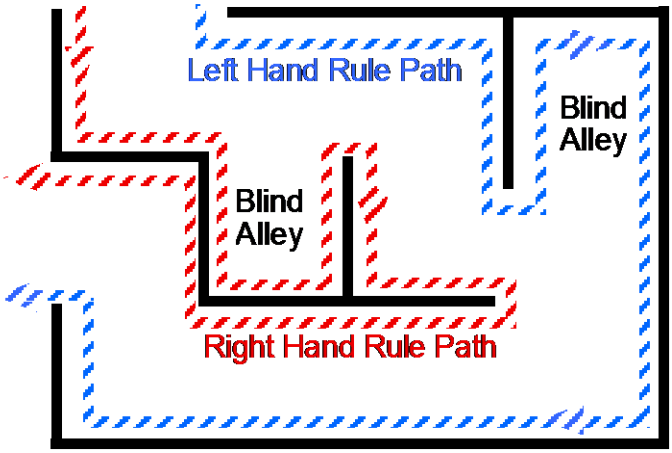
Picture yourself going through a maze. If all you did were keep your right hand on the wall, you would eventually find your way out of most mazes. The robot can do the same thing with two touch (or Infrared) sensors; one on the front, one on the right side.It will follow the right hand wall through the maze.

The goal is to keep the sensor on the right side in contact with the sidewalland for the one in front to sense a wall it is about to run into. As long as the side sensor is depressed, the robot will drive forward (actually very slightly to the left). If the side sensor is released, then the bot will move back to the right until the sensor finds the wall again. If the front sensor is pressed, the bot will back up and turn 90 degrees to the left and pick up the right wall again. This strategy is a bit slower, but will solve any maze every time. One of the tricks to getting this method to work is the placement of the sensors. You will need to keep the sensors pretty far forward and find just the right angle.





Programming Right Hand Rule: This is a sample of a program for using the right hand rule. This assumes that the right whisker is in pin 1, and the front whisker is in pin 2. The program checks the status of pin 1. If it is touching the right wall, it will drive forward (with a very slight left turn). If it is not touching the wall, it will turn right. Then it checks the front sensor. If it hits the wall in front, then the robot will back up and turn left. Otherwise, it drives forward (with the same slight left). The keys to making this work are the amount of slight left in the forward motion tile, the right turn, and the left turn. You want the slight left to pull the robot away from the wall if it gets too close. The right turn must get the robot around the end of a maze piece. The left turn must get the right whisker close enough to the wall that it touches and can follow that wall.

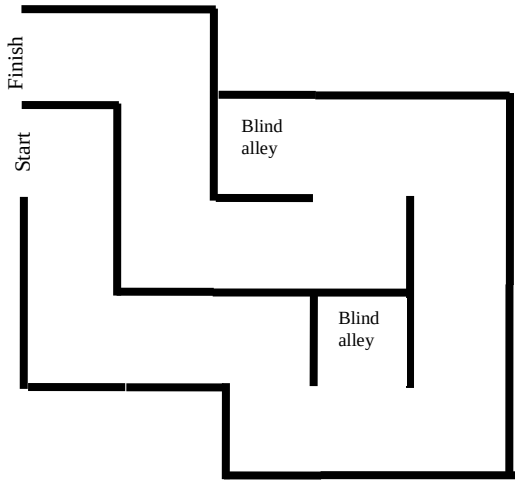


Left & Right-Hand Rule Paths in a Maze
Note the two blind alleys.
For the maze to be fair, there should be the same number of blind alleys on the left and right walls

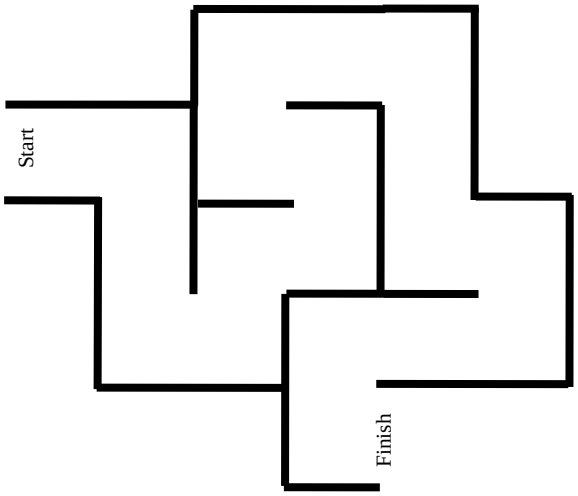
Sample Maze Layouts

Below are some possible maze layouts to consider. The maze can be done in any number of configurations, but there are some things to consider when designing the maze. Try to make an equal number of left and right turns. If you make a maze with blind alleys, make one on each wall so that those doing the right hand rule do not have an advantage (or disadvantage).

Maze with Blind Alleys

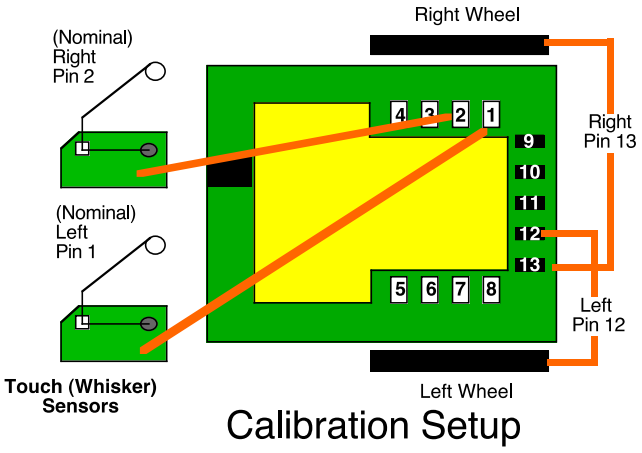


Maze w/o Blind Alleys



Calibration

The first step will be to calibrate the servos. If they are not properly calibrated, the robot will not drive straight, which makes maze solving difficult. The calibration program assumes that the left touch sensor is in pin 1, the right touch sensor in pin 2, the left wheel servo in pin 12 and the right in pin 13 as shown.

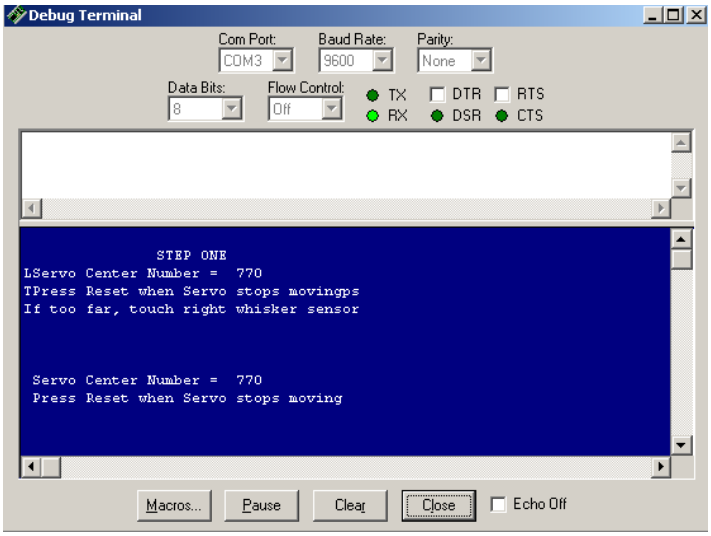


Calibration Setup

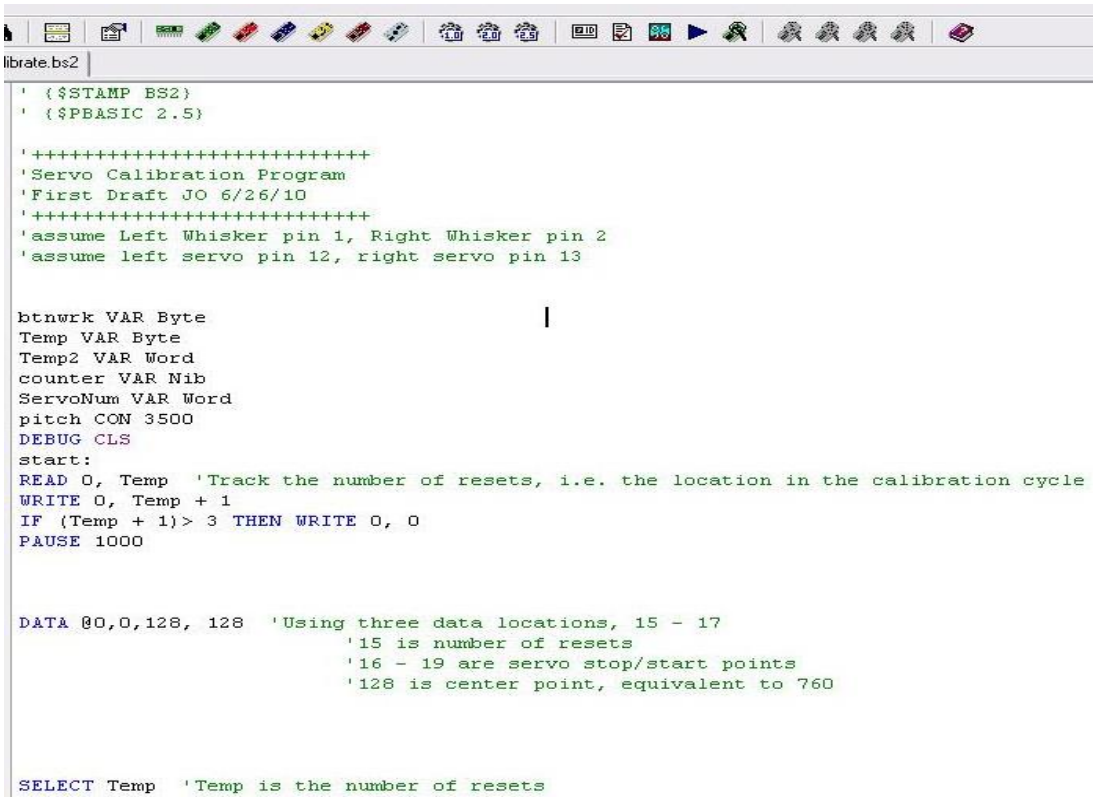
Click the **Calibrate** button in **Program Maker.**

Program Maker will disappear, and a new piece of software, **Basic Stamp Editor**, will open. (Don't worry, your program in **Program Maker** is still there, it is just hidden.) Connect the robot to the computer, and press the run button, or F9 key in Basic Stamp Editor. The calibrate program will download to your robot, and a blue "debug" window will open. In that window, you will find instructions to follow.


Run



1. The left wheel should be turning slowly. If it is not or if the right wheel is turning, check your setup. Press the left whisker as indicated until it the wheel just stops turning.
2. Press the reset button on the BotBrain Board, and the right wheel will start turning. Press the whiskers again as indicated until the wheel stops.
3. **(ScreenShot)** Press the reset button again. The robot will drive forward and backward. (Make sure that there is enough room for the robot to drive, and not drive off the table. If your programming cable is long enough, do this on the floor.)
4. If the robot is drifting right, press the left whisker and vice versa.
5. **(ScreenShot)** Once it is driving straight, press the reset button again. The values for the servos will now be stored in memory on the BotBrain Board.
6. Close the blue debug window.
7. Close the Basic Stamp Editor. When you do, Program Maker will be visible again. To run your maze program, load it into the robot, and your ready to go.


librate.bs2

```
' {$STAMP BS2}
' {$PBASIC 2.5}

'++++++++++++++++++++++++++++
'Servo Calibration Program
'First Draft JO 6/26/10
'++++++++++++++++++++++++++++
'assume Left Whisker pin 1, Right Whisker pin 2
'assume left servo pin 12, right servo pin 13


btnwrk VAR Byte
Temp VAR Byte
Temp2 VAR Word
counter VAR Nib
ServoNum VAR Word
pitch CON 3500
DEBUG CLS
start:
READ 0, Temp  'Track the number of resets, i.e. the location in the calibration cycle
WRITE 0, Temp + 1
IF (Temp + 1)> 3 THEN WRITE 0, 0
PAUSE 1000



DATA @0,0,128, 128  'Using three data locations, 15 - 17
                          '15 is number of resets
                          '16 - 19 are servo stop/start points
                          '128 is center point, equivalent to 760



SELECT Temp  'Temp is the number of resets
```

The Basic Stamp Editor Window showing the Calibrate Program

