

Chapter #9: A Mazing Things

A Mazing Things

Mobile robots can do many things but one of the most basic is to move between two points while avoiding obstacles. The programs used with the J-Bot thus far have

been relatively simple. They use the various sensors to avoid an obstacle but they will not traverse a complex set of obstacles such as walls that make up a maze.

In this chapter we examine some basic maze traversal algorithms. The J-Bot will use the infrared detectors configured and tested in prior chapters to help it traverse a maze without bumping into a wall. For our purposes, a wall will be considered a boundary that the J-Bot should not cross. We do this because the J-Bot can be setup to detect either a vertical wall in front of it or a virtual wall created by some black tape like that used for the line follower experiment. In the latter case, the J-Bot's infrared sensors will be aimed at the floor. The floor should be a color that reflects the infrared light. Typically a white floor will be suitable. The J-Bot can detect the difference between the white floor and the black tape designed to be the wall.

Two sensor classes are already defined to handle the two types of maze walls. The `IrRangeSensor` is used to detect walls while the `IrRangeDropOffSensor` is used to detect virtual walls implemented using black tape on a white floor. Either sensor can be used with the applications presented in this chapter. The one you use will depend upon the type of maze that is constructed for the J-Bot.

Maze exploration can get very elaborate. It is possible to have start and stop points but this requires additional recognition by the J-Bot. The J-Bot can easily handle additional sensors to detect when it exits a maze. For example, an IR sensor could be aimed up to detect when the J-Bot passes through an arch. In this chapter we forego the stop point detection and simply let the J-Bot run forever. If a maze contest is being run then the J-Bot can simply be picked up once it exits the maze or crosses the stop point.

There are a number of different ways to navigate through a maze. Three are presented in this chapter including:

- Random Walk
- Right Hand Rule
- Basic Backtracking

The Random Walk is similar to the normal obstacle avoidance programs used earlier in this book with a minor change so that the J-Bot will turn in a random direction if an obstacle is found in front of it. In theory, the J-Bot should eventually find

its way out of a maze but it may take a very long time since it will cover the same area many times.

The Right Hand Rule approach uses a simple mechanism for finding a way out of a maze. It is called the right hand rule because the a person could find their way out of a maze by walking so their right hand could touch the wall. This method always works if the maze does not have any cycles. A cycle occurs when there is a wall does not come in contact with a wall that forms the maze exterior. A maze without a cycle is often called a simple maze. A maze with one or more cycles is called a complex maze. The right hand rule will get the J-Bot out of any simple maze. It will get the J-Bot out of a complex maze if it starts next to the exterior wall. If the J-Bot starts next to a wall that is not the exterior wall it will follow the wall forever. The Right Hand Rule is simple to implement and it will normally get the J-Bot out of a simple maze faster than the random walk.

The Random Walk and the Right Hand Rule approach do not keep track of where the J-Bot has been. The J-Bot does not know if it winds up in the same spot more than once because the algorithms do not incorporate any form of memory. The J-Bot simply responds to immediate feedback.

The Basic Backtracking approach brings memory into play. It keeps track of where the J-Bot has been and where it should look next if a particular path does not lead out of the maze. The Random Walk and Right Hand Rule do not require precise movements but the Basic Backtracking approach does because the J-Bot will need to backup to a point where it can continue to explore if a particular path results in a dead end.

Activity #1: Random Walk

The Random Walk approach utilizes a task to control movement and a sensor object to detect obstacles. As mentioned earlier, the `IrRangeSensor` and `IrRangeDropOffSensor` classes can be used. The sample program in this activity utilizes the `IrRangeSensor`.

The theory behind the Random Walk maze exploration approach is that random movements when an obstacle is encountered

The `RandomWalkTask` class file below is very similar to the `AvoidObstacleTask` class used earlier in the book.

There are three major differences between the `RandomWalkTask` class and the `AvoidObstacleTask` class definitions. First, the `RandomWalkTask` does not check the distance to an obstacle. This

is done so either type of the aforementioned sensors can be used. Remember, the floor looking sensors do not provide a good distance result so an obstacle distance will always be reported as zero.

The second difference is that most movements are chosen randomly when an obstacle is encountered. The exception is when an obstacle is detected only to the left or right. This is where the randomness comes into play. The flipCoin method uses a Random object to generate a result. The Random.next method returns a number between 0 and MAX_RAND. If the object generates a truly random sequence of numbers then half will be above MAX_RAND/2 and half will be below this value. This is essentially the same as flipping a coin to see if it lands showing heads or tails.

The third difference is minor. In the initialState in the execute method, there is only one call to the nextState method. This is because the extra calls in the AvoidObstacleTask's execute method were not really needed since the state remains the same unless changed. Changing it to the same state has no affect on the operation of the method.

The test program, RandomWalkTest1, is relatively simple since all the work is done by the sensor object and RandomWalkTask.

The TaskManager method should not terminate since the RandomWalkTask task will never terminate. If the RandomWalkTask class is altered so it can check when the J-Bot exits the maze then the TaskManager method to return.

The program begins running as soon as power is applied. Watch the J-Bot as it moves through the maze. It should not come in contact with the walls. If it does, try turning the IR LED and sensor on each side towards the outside so it will detect obstacles to the side sooner.

Keep in mind that the J-Bot may collide with a wall when moving backwards since it has no sensors there. The J-Bot will not have a problem if the maze provides enough room within a corridor. The J-Bot will not be able to navigate corridors that are only slightly larger than its width. In general, obstacles should be at least one foot apart if the J-Bot is to pass between the obstacles.

Your Turn

- ❑ The range sensor has not been optimized. It always checks all 16 distances. Is this necessary for this activity or is it possible to check for the maximum distance first? Do closer

distances need to be checked if a greater distance is detected?

- ❑ The distance result from the range sensor is not used. What would the difference be in the J-Bot's movement be if the distance result was checked to allow the J-Bot to get closer to a wall? If the downward looking IR range sensor were used instead, how could distance be judged?

Activity #2: Right Hand Rule

A simple maze actually consists of a single line that has been bent at various points. Getting out the J-Bot of a maze of this type is then a simple matter of following the line until the J-Bot exits the maze.

The `RightHandRuleTask` class defined next handles the movement of the J-Bot based on feedback from the sensor.

The `waitNextState` and `waitSensorNextState` methods are included because these short sequences of method calls to change the state variable and suspend the task are used often within the `execute` method. Essentially the task runs, starts the sensor, suspends, is resumed by the sensor when data is ready and the `execute` method then determines what to do based on the sensor results. The same process is used for movement that is initiated by the sensor results. In this approach, only one task will actually be running at any time. In general the execution sequence that is done repeatedly is this task, the sensor task, this task, and then the J-Bot movement task.

FYI

The Task resume method does not change the current state. The stop method changes the state to stopped. This is why the resume method must be used in the `waitNextState` method definition.

The task cycling does lead to a start-stop movement of the J-Bot because the sensors take a noticeable amount of time to get the range of an obstacle. It is only a fraction of a second but enough so that the movement of the J-Bot is not smooth. There are tradeoffs compared to the approach taken in Activity #1. In Activity #1, the J-Bot moves continuously but its sensor readings are not as accurate because the J-Bot is moving so its movement control is not that accurate. In this activity, the movement is very accurate but the movement is not continuous.

The execute method is where the task control is handled. The task starts in the initialState where it starts the sensor using waitSensorNextState. The execute method will be called again when the sensor has obtained its results and notified this task so it will enter the checkForObstacle clause of the switch statement. The results of the sensor are tested and the J-Bot movement control is initiated via the jbot object. The next state will be initialState or doneWallToRight depending upon the sensor results. If the next state is initialState then the J-Bot will have moved forward one inch. This distance can be greater depending upon the distance to an obstacle obtained from the sensor but this is not checked in this version of the program. If it were then the J-Bot could be instructed to move forward a greater distance without colliding with an obstacle.

There are two logical, high level cases that this task can be in. The first is where the J-Bot does not know where the wall is. The second is where the J-Bot thinks the wall is to its right. In the first case, the J-Bot moves forward until a wall is detected. It then turns so the wall will be to its right that is the second case. In the second case, the J-Bot moves forward a short distance, turns right to check if the wall is still there. If it is, the J-Bot turns left and moves forward trying to move along the wall. If not, it assumes there is a turn and the J-Bot moves forward assuming the wall is now to the right. It will know if this assumption is correct after the initial movement.

The test program is relatively simple since most of the work is done by the RightHandRuleTask.

The try/catch/finally method was added to this test program because making changes in the tasks can result in some strange errors. This allows the system to shut down gracefully. The main method is not much different than in Activity #1. The range sensor is created and passed to the newly created task object that starts everything running. In theory, the Task.TaskManager should never terminate because there is no maze exit detection mechanism.

The J-Bot will require walls that are relatively far apart as in Activity #1. It should be able to go down a corridor that is at least one foot wide. In most cases, the wider the better.

The RightHandRuleTask has some deficiencies. It could allow the J-Bot to get closer to a wall. It only moves one inch forward at time, and it performs 90 degree turns when a 45 degree turn may be sufficient. Correcting these deficiencies will make the program more complicated but the new program will improve the way the J-Bot operates. In particular, allowing the J-Bot to get

closer to a wall will mean that it can traverse an area where the walls are closer together.

The range sensor has not been optimized as noted in Activity #1. Improving the response time for the sensor will minimize the jerky movements associated with this application.

Activity #3: Basic Backtracking

The prior activities move around a maze but they do not take advantage of the information to be gained by keeping track of where the robot has gone. This activity takes a different approach. Now the JBot maintains a map of the area it has traveled in and uses that information to determine where it should explore. The type of mapping and backtracking is just one way to implement a more intelligent maze exploration program.

The JBot needs to operate on batteries, unattached to the PC. Still, it is useful to let the JBot show you what it has been up to. Short of using a wireless link (a good alternative when extra hardware is added), the JBot can store the map it makes so it can be printed when the JBot is reattached to the PC. This will look something like the following captured from the Message window.

```
Stored map
Y range -1 2
X range -1 2
 2: . 0 . .
 1: 0 + 0 .
 0: 0[+] + 0
-1: . 0 0 .
```

In the example, there is a 4 x 4 element array. The unexplored areas are noted by periods (.) and the obstacles are noted by the letter O. The area traveled by the JBot is indicated by the plus sign (+) and the starting position is in brackets ([]).

To do this, we need a few things. First is the main exploration program, MazeMapTaskTest1. This will move the JBot through the maze and create the map. The map will then be stored in the JBot's EEPROM when exploration is complete. The second program, DumpMazeMapTaskTest1, is loaded after the JBot runs through the maze. It reads the EEPROM and displays the information shown in the Message window example just given. The EEPROM area is shared by the programs and this data but there is more than enough space for both. Downloading a new program does not overwrite the data area allowing this approach to work.

The MazeMapTaskTest1 program is tested in the following fashion. The program is downloaded from the PC to the JBot. The power and PC cables are then disconnected from the JBot. The program is stored in the EEPROM at this point. The JBot must have its

batteries installed. The JBot is then placed in the maze and battery power is applied by connecting the power cable to the JBot. The JBot then moves through the maze mapping as it goes. Once it thinks it has explored the entire maze the buzzer will sound. The map is stored in EEPROM at this point.

The JBot is then reattached to the PC and the DumpMazeMapTaskTest1 program is downloaded and run. The map results are displayed and can be compared to the actual maze. To repeat this scenario, the MazeMapTaskTest1 must be again downloaded since the JBot will have the DumpMazeMapTaskTest1 in memory at this time.

How It Works

There are a couple of classes that are needed to make things work. These are covered in more detail later in the activity. The main task is covered in **The Mapping Task** section that comes next. The other classes are found in the **Supporting Classes** section. If you are interested in the details then check out all the sections. If you want to forego some of the details and concentrate on what makes things move then you can leave the **Supporting Classes** section till some later date.

The two main programs, MazeMapTaskTest1 and DumpMazeMapTaskTest1, are covered in this program. They are relatively short since they utilize existing classes or ones covered in this activity. The MazeMapTaskTest1 is modular like the prior examples so it is possible to replace the sample objects with almost any compatible object. For example, the MazeMapTaskTest1 uses the IR detector sensor class, IrRangeSensor. This can be replaced by any compatible sensor like the whisker sensor class covered earlier in the book. Likewise, the JBot movement support can be replaced as well. The MazeMapTaskTest1 uses the wheel encoder support covered in the last chapter to keep a more accurate track of the JBot's movements.

The MazeMapTaskTest1 makes a couple of assumptions. First, the sensors range is at least two inches. If this is not true, as with the whisker sensors, then a minor adjustment to the mapping task needs to be made to prevent the JBot from running into an obstacle. Second, the JBot moves in fixed increments. This makes control and mapping easier. It is possible to run the JBot so its movements are continuous but this is a more difficult problem to tackle. Finally, the program assumes that the areas that the JBot will be able to explore are within its limitations. For example, the corridors of the maze should be about 12 inches wide. The JBot is assumed to occupy a 6 x 6 inch area that easily fits within a 12-inch corridor. It will assume that an opening that is too narrow for it is an obstacle. This of course depends upon the sensors employed but it is best if the maze is setup so the JBot can navigate it easily. This implies that the edge of a narrow

wall should not be exposed since the JBot's sensors may miss it or improperly interpret the sensor results. In general, it is best to make a maze where all exposed walls are at least 12-inches.

The best way to make a maze for the IR detectors is to use a bunch of white boxes that are at least 4-inches high, at least 6-inches wide and 12-inches long. These can be placed next to each other to create a maze that can be easily reconfigured. They also move if the JBot accidentally runs into a box.

Now onto MazeMapTaskTest1. It is shown in the following listing:

```
The main method creates a printableMazeMap and a MapMazeTask. The latter uses the map along with a JBotInterface that is a new RampingJBot. The 2nd parameter of the constructor specifies the range value that indicates an obstacle is within 1- to 2-inches in front of the JBot.
```

The printableMazeMap constructor takes two parameters. These are the y and x map dimensions. In this case there will be a map with 400 elements created. Each element corresponds to a 6-inch square that the JBot can occupy. This means the map can represent an area that is 10 feet on a side.

Before you go out and create a giant maze, keep a couple of things in mind. First, it is a lot of work. Second, the JBot will not be able to accurately navigate this area as you might think.

Remember, when the JBot moves or rotates, its movement is not exactly what you may desire. For example, if the JBot turns to the right the actual amount of rotation may be anywhere from 85 or 95 degrees instead of exactly 90 degrees. This may appear to be a minor difference and may not even be noticeable but this error can be a problem. This is because cumulative error results from the combination of multiple movements with a small amount of error. Let the JBot turn half a dozen times and it can be off by more than 45 degrees.

For this reason, the size of the test maze should not be too large. A few feet on a side is more than enough for experimentation. The JBot can navigate through a large maze but the JBot's map may not give the desired results.

<<ed note: it would help to reference Laura Wong's 2002 ISEF paper here>>

The MapMazeTask runs with the sensor and JBot tasks when the Task.TaskManager method is called. These tasks move the JBot around the maze. All tasks stop when the program determines that the maze has been explored to the best of its ability. The Task.TaskManager method then returns and the map.save method is called. This stores the current map in EEPROM. A FREQOUT object is created to sound a one second tone indicating that exploration is complete and the map has been save. The power can then be disconnected so the JBot can be reconnected to the PC.

Next the following DumpMazeMapTaskTest1 program is loaded.

There are actually more comment lines than anything else. The class method, load, is used to read EEPROM memory and create a printableMazeMap that is identical to the one used by the MazeMapTaskTest1 program. The print method will display the map's contents in the Message window.

A simple way to test these programs while leaving the JBot connected to the PC is to put the JBot on a small box or object so the wheels do not touch. Note, detaching the servo cables will not work because the wheel encoders only work if the wheels actually move. An obstacle is placed in front of the JBot. Running the MazeMapTaskTest1 program should result in a map with one traveled cell, where the JBot starts, with obstacles all around it. You can watch the wheels move while the program runs and you can add System.out.println method calls to print status information. Of course, things get more interesting when the JBot is running unencumbered.

The next section takes a look at the mapping task that actually handles most of the work.

The Mapping Task

The MapMazeTask is the main task that controls the JBot and records its movements in a printableMazeMap covered in the next section. The MapMazeTask extends the Task class. Its execute method is called repeatedly by the Task.TaskManager in the main program, MazeMapTaskTest1, discussed in the prior section. The following is the MapMazeTask class file.

The MapMazeTask consists of two methods, the constructor and the execute method. The constructor saves off the parameters. It

creates a character array for the backtracking path and sets the direction of the JBot to north. Now this direction is strictly for mapping purposes and does not correspond to magnetic north. This would require the use of a compass like the Parallax Compass Appmod.

The MapMazeTask only moves the JBot along straight lines and only pivots the JBot at 90 degree angles. This greatly simplifies the mapping and backtracking process. It does mean that movement along a corridor that is not at a 0 or 90 degree angle with respect to the initial JBot position will be explored in a stair step fashion.

The execute method is divided into three states:

```
initialState
computePath
followPath
```

The initialState checks the sensors to see if an obstacle is in front of JBot. It does not matter whether it is directly in front or only to one side. Because of the granularity of the system, an obstacle that partially blocks the JBot is assumed to be the same width as the mapping cell's logical size that is 6-inches.

The JBot moves forward in short steps of 2-inches since this is assumed to be the maximum distance the sensors can detect. Some sensors may provide a longer range and the movement can be adjusted by changing the stepDistance value used in the jbot.move method call. The checkingOffset variable is used to keep track of how far JBot inches forward. If the JBot gets partway into the next 6-inch square and detects an obstacle then the JBot backs out, using the checkingOffset value, of the logical map cell and marks it as an obstacle. This means that a JBot moving down the middle of a 12-inch corridor will mark it as a 6-inch corridor, a one cell map width, even though there is actually more space than the map indicates.

The area traveled by the JBot is saved in the map calling map.markTraveled. This occurs when the checkingOffset hits 6-inches. The logical position in the map is adjusted using the map.move method. The direction variable, facing, is used to keep track of the logical direction the JBot is facing.

The fuzziness of the map is actually for the JBot's benefit. It allows the JBot to move through an area that it has traveled before without having to worry how close it is to an obstacle. It will be at least one inch away if not more.

The computePath state marks the explored area as an obstacle using map.markObstacle. It then obtains a path to a new exploration area by calling map.computeBacktrackPath. The path is

placed in the `backtrackPath` array. Each element of the array is the direction the JBot needs to move, starting from its current location, to get to an unexplored area. The result of this call is the number of directions placed in the array. If the number is 0 then all areas accessible by the JBot have been explored. The task is and servos are stopped and the `Task.TaskManager` method, discussed in the prior section, will return.

The `followPath` state is used to move the JBot along the backtrack path. The last step in the path is used to position the JBot but the JBot will not move into the area. Instead, the `initialState` will be entered at this point and the JBot will use the sensor to see if the unexplored area is open and can be traveled in or if it contains an obstacle.

The `followPath` state compares the current direction of the JBot with the desired direction. It turns the JBot to the desired direction if necessary. The use of the `dir` variable is an optimization. The initial turn computation can result in a 270 degree turn that is the same as a 90 degree turn. The latter is more efficient and accurate. This code means the JBot will only be executing left and right 90 degree turns or 180 degree turns.

It then moves the JBot 6-inches in the desired direction assuming that the path step is not the last one. The `map.move` method call changes the logical position within the maze map. The `map.markTraveled` method call is actually redundant because the path will always be over a traveled area.

That's it. The logic is relatively simple but getting to this design takes a good bit of thought. This is one of the simpler methods for using a map. As you can see, making things more complex is no easy chore but it is a good challenge.

Supporting Classes

The `printableMazeMap` is used by the `MapMazeTask`. It is based on the following class hierarchy.

```
charArray
  charMap
    mazeMap
      printableMazeMap
```

The classes serve two purposes. The `charArray` class is needed because the Javelin implements only one dimensional arrays and our map is a two dimensional array. The `charMap` is used to provide a more dynamic use of the array since arrays are normally accessed from 0 to N where there are N+1 elements. The `charMap` allows negative indices to be used. The map grows as it is accessed to the maximum size used to create it. This is very

handy for our mapping because the JBot can be logically started at 0,0 and moved in any direction based upon its exploration. A conventional map would have to be four times the size to provide similar coverage.

The `mazeMap` adds backtracking capabilities as well as constant definitions for the values that are used and stored in the map. The `printableMazeMap` extends this class by providing EEPROM storage methods and `System.out.println` status reports. It does not provide any additional mapping or backtracking capabilities.

The following is the `charArray` class definition. It maps a 2-dimensional array onto a 1-dimensional character array. The same approach can be used for arrays of any type. The methods like `xLow` provide a consistent way to determine the size of an array that can take into account the dynamic nature of the subclasses. The class throws the `IndexOutOfBoundsException` exception if the array is addressed improperly.

The `charMap` class presented below maps a logical array with bounds that can be negative on a conventional array that uses only positive array indices. This is handled by redefining the `getIndex` used by the `charArray`. The class maintains the high and low map ranges and provides public methods for their access.

The `mazeMap` class, shown below, builds on the `charMap`. It populates it with values like `obstacle` and `traveled` and uses the `path` value when searching for a backtracking path. It maintains a logical position in the `y` and `x` variables. These are changed using the `move` method. It assumes logical directions based on north, east, south and west that match the movements the JBot can perform. The `markTraveled` and `markObstacle` methods are used to populate the map based upon the information the JBot can garner regarding its surroundings.

The `computeBacktrackPath` searches for a path to an unexplored area using and modifying the current map contents. The map was initialized to `unknown` (or `unexplored`) and the method looks for a path from the current logical position (`y,x`) to a cell marked `unknown`. The search proceeds by changing `traveled` cells to `path`

cells as a path is created. Cells marked as part of the path are left in that state until the method returns at which point these points are changed back to traveled leaving the map in its original position. This is done so the method can determine whether it has already looked at a cell for a path. If this is not done then a circular search can result in an infinite loop.

The method does not try to find the closest unexplored area. It simply tries to find one using a simple algorithm. It starts looking north. If it cannot find a path in that direction it moves around the cell to the right ending in a westerly direction. This means it will find the first unexplored area to the north 10 cells away even if there is an unexplored cell immediately to the west.

Note how the direction of the search is changed as the path is cut back. The path consists of a set of directions from the logical position (y,x) to an unexplored cell.

There are many ways to search a map. This is simply an approach that is easy to implement. It also uses no additional space. This can be critical in tight memory environments like the Javelin.

The `printableMazeMap`, shown below, adds a EEPROM methods and `System.out.println` oriented methods that provide a way to save, restore and display the contents of a `mazeMap`. The `load` method is a static class method that returns a new `printableMazeMap` object. The contents of the map are stored at the beginning of the EEPROM area. A more sophisticated version would allow the map to be positioned elsewhere in the EEPROM area. The `save` method stores an existing map object in EEPROM.

The `print` and `printPath` methods display the map and the current search path respectively. They convert the binary values used in the map and path array to more friendly output.

The `printableMazeMap` class is used in the mapping application but it was developed using the `MazeMapTest1` program shown below. This allows a map to be populated and the path creation method to be tested without having the JBot traveling all around a maze. This class is not necessary for the exploration and result applications already defined but it is invaluable when trying to develop new path search algorithms.

Your Turn

- ❑ The `computeBacktrackPath` method does a depth first search. Try implementing other search algorithms. For example, a breadth first search is more complicated but will provide different results. A greedy search should find the closest unexplored area.



Summary and Applications

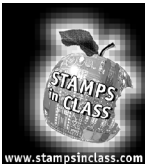
This chapter addresses navigation through a maze. It utilizes the J-Bot control and obstacle classes defined in previous chapters. Three maze navigation methods were presented: Random Walk, Right Hand Rule and Basic Backtracking. The Random Walk presented in Activity #1 used a continuous movement mechanism while sacrificing sensor accuracy. The other methods alternated between sensor and movement actions that were more accurate but more jerky in the J-Bot's movement. The basic backtracking method improved upon the Right Hand Rule method allowing the J-Bot to keep track of the areas already explored thereby allowing it to navigate through a complex maze.

Real World Example

Mobile robots that avoid obstacles and explore are used in a variety of areas. The most notable is the Sojourner robot that is still up on Mars. Although it was similar to a remote control vehicle, Sojourner was a true robot. It needed the ability to navigate by itself because the time needed to send a signal from Mars and back again is measured in minutes. This delay is too long for safe control of the robot. The robot could easily run into a rock before the human controller on Earth signaled the robot to stop.

Instead, Sojourner was programmed to move from one point to another. The robot was responsible for moving between the two points while avoiding obstacles that were in the way.

The maze exploration programs presented in this chapter are slightly different from the programs used with Sojourner because the J-Bot had no final fixed destination. Instead, the J-Bot would explore the maze.



Questions and Projects

Questions

1. What is a cycle within a maze? This is the same thing as a circular path.
2. What is a simple and complex maze?
3. Why would the J-Bot fail to exit a maze when using the Right Hand Rule?

4. How does sensor detection delays impact algorithms and movement?

Exercises

1. Change the Right Hand Rule in Activity #2 to the Left Hand Rule approach.
2. Modify the maze exploration programs so they record in EEPROM the path followed by the J-Bot. Write a program that displays this information in the Message window. Remember, the J-Bot needs to be connected to the PC for the latter.

Projects

1. The RandomWalkTask in Activity #1 polls the sensor to see when an obstacle range is available. Polling adds overhead so other tasks like the sensor task and servo task have less time to run. Use the setEvent support within the BasicSensor class so that the RandomWalkTask can suspend itself while the sensor task does its job. Does this change improve the J-Bot's response time when detecting an obstacle? It should.
2. The IR sensor classes return an obstacle distance from 1 to 16. The default implementation checks the sensor 16 times regardless of the distance detected. If the maximum distance to an obstacle before an action is taken is greater than 1 then the sensor class does not have to check any values between 1 and the maximum distance - 1. Modify the sensor class so it only checks between a range of values. Does this change improve the J-Bot's response time when detecting an obstacle? It should.
3. Having the JBot detect walls is easy but accurately turning the JBot can be a problem. One way around this difficulty is to simplify the problem and place the accuracy of movements on the maze instead. This can be done by using line following and clearly marking the intersections. An obstacle is considered to be a line that simply ends. Create the maze using lines of tape making sure the intersections are marked every six inches. The JBot should be programmed to follow each line until it ends as noted above. The Jbot does not have to worry about accurate rotation because the line following program should make the minor adjustments to follow the line.