

RIS

Install RIS software

Install the RIS software on the computer you will be using to do your Java programming exercises. The computer needs a minimum of roughly a 400 MHz Pentium class processor, 256 MB RAM, 200 MB of free disk space, CD ROM and Windows 98 operating system. These are minimum requirements. More memory, a faster processor or a later version of Windows, e.g. XP, are all “good things”.

Installing the RIS software is simple. Insert the RIS disk into the CD ROM drive and the installation should start immediately. At every opportunity click on the “OK” or “Next” button (unless you have a reason to specify something other than default). Besides the actual RIS software, DirectX and QuickTime may also be installed. If you are asked, install both. At the end of the installation, your computer will want to re-boot. Click “OK”. After the re-boot, RIS will be started. At this point, you will need to install batteries in the RCX and the IR tower.

If the installation does not immediately start when the RIS disk is inserted, you probably have autorun turned off. To start the installation manually, double click on the “my computer” icon on the desktop and then double click on the CD ROM drive. This should bring up a directory of the RIS disk. Find and double click on “setup” to start the installation. Then follow the instructions above.

Start Robotics Invention System (RIS)



When you start the RIS software, the screen on the right appears. Clicking on run causes the login screen to appear (note, the introduction screen may not appear and you may be presented with the login screen immediately). If you have used the software before, your login name will appear in a list underneath the text input field. If so, double click on your name. If not, type your name into the text field and click on the “new user” button.



Do the training missions



Once you are logged in, the screen at left appears. You should explore a little by clicking on the tour button and then look at some of the features of the RCX by clicking on the settings button. After you are done exploring, click on the missions button and the mission command screen appears. Click on the training missions button and the training missions screen appears. Complete all the training missions which should take approximately two hours. You can stop at any time and the RIS software will remember where you stopped. Doing all the training missions is important. Later on, we will be



creating and using programming models of the various Mindstorms



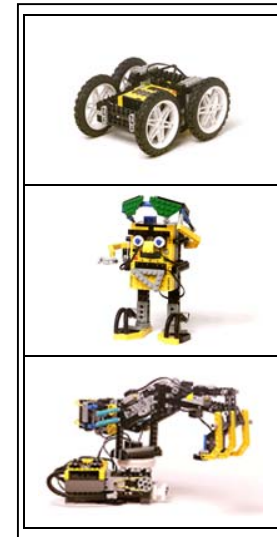
components. The training missions provide a good

introduction to each component and being very familiar with these components aids in using them in programs. We will also be covering the topics contained in the training missions in greater detail. The visual nature of RIS programming will give you a deeper understanding of programming that is directly applicable to learning how to program in Java.

Build the Roverbot



The RIS software provides a variety of well-designed robots, examples at right, with clear directions on how to build them in the *Constructipedia* and pictures in the RIS software. The Roverbot, pictured at left, is one of these robots. It is engineered to be rugged; dropping off a table onto the floor usually results in no damage. It can support a variety of interchangeable sensor assemblies including a one-touch sensor assembly, a two-touch sensor assembly and a light sensor assembly. It also can be fitted with small wheels, large wheels, tracks or legs. It is easy to change gear ratios to make it slow and powerful or fast and weak. And it has a zero turning radius, making it useful for robotic navigation exercises. You should build the Roverbot just to practice building



robots and to gain experience on what constitutes good construction instructions.

Once you have built a Roverbot, you can choose to build your own robot. The essential feature of the Roverbot that we will be using is that it has a zero turning radius, ZTR. That is, it steers by having one motor turn off while the other stays on, “turn” or having one motor reverse while the other remains going forward, “spin”. This is the method used by a tank or bulldozer. The other method of steering normally uses a rack and pinion. Cars use this method of steering. When we start using navigation, ZTR makes programming much simpler and is more typical of commercial robots.

Programming tasks

At this point, you should have a working robot. If you haven't already, download the firmware and try running the built-in programs. If you don't know how to run the programs, go back and do the training missions.

Before we start talking about programming concepts, you need to do some programming (yes, this is putting the cart before the horse). There are two programming tasks that follow, maze runner and line follower. They are both easy to describe. A maze runner navigates (goes through) a maze. A line follower follows a line on the floor. For a person, these tasks would be very easy. For a robot, you have to program them.

There is one very important thing to remember as you start programming your robot. This is supposed to be fun. Make sure that you smile and you laugh as your robot does really stupid things because you screwed up the program.

Maze runner



You just have to trying.

As a hint, one problem that is often encountered is oscillating in a corner. The robot heads towards a corner. It touches on one side, backs up, goes forward and then touches on the other side. It then proceeds to do a mirror image movement and arrives back at its starting point. It then bounces off one wall and then the other for a very long time. Recognizing that the robot is oscillating is hard to do in RIS code but possible.

One of the classic tasks of robots is to maneuver through a maze. A maze runner is a ZTR robot with two touch sensors mounted on the front ([see the picture of the Roverbot above](#)). Its mode of operation is to go forward until one of the sensors registers contact with a wall. It then backs up, a little bit, and turns, a little bit, away from the wall. It then proceeds forward until it hits the next wall. This is a very simple task to program in RIS, at least for first cut.

It is easy to build a maze. You can use bricks, boards, books, or anything else that is relatively flat and heavier than your robot. Arrange the objects in a pattern that has a space for the robot to go through. Then try having your robot run the maze with this program. If your robot does not make it through the maze, and with this program making it through the maze is highly unlikely, analyze why it does not. Then try and fix it. I have written maze runner code in RIS that does work, so you can be sure that it is possible to do it.

Line follower



robot to follow the boundary between the black and the white. Since the sensor averages the area that it is pointed at, we want to maintain the reading in the range of 35 to 45. So long as the reading is between 35 and 45, we want our robot to go forward. If the reading goes below 35, the robot is getting into a region that is too black and needs to turn away from the black. If the reading goes above 45, the robot needs to turn in the opposite direction.

Get out the test pad from the RIS kit and see what values work. Does this program do what you want or do need to change it? You can select the threshold values by clicking on the sensor. May sure that there is a space between the values where the robot goes forward (this difference is called *hysteresis*). The panel on the right allows you specify values. You can see the value of the sensor reading by viewing it on the LCD of the RCX after you select it using the view button of the RCX (the training missions covered this too).

Line follower is very similar to maze runner. As the robot moves, a light sensor is pointed at the floor. Over dark places, the light sensor has a low reading. Over light places, the light sensor's reading is high. What a line follower attempts to do run along the edge of a line, the place where the black and the white come together. The sensor effectively averages the readings from the black and the white sides of the edge of the line. The objective of a line follower is to go forward while turning away from areas that are too white and turning away from areas that are too black.

Putting this into concrete terms, assume that black has a sensor reading of 30 and white has a reading of 50. We want our



Programming in RIS

What is a program?

The last two sections had you writing and debugging (fixing) programs for a robot. Now, it is time to start thinking about the nature of programs.

The old definition of a computer program is an *ordered-list of computer-instructions*. While this definition is less accurate than it used to be, it is still a good place to start.



You should be familiar with the concept of lists. Most people make up a list of items to buy before they go to the grocery store. This is an *unordered-list* because it doesn't really matter which item you put in your cart first. On the other hand, a list of errands is usually an *ordered-list* because it is important that you “work out at gym” before you “buy ice cream to take home”. Doing those errands in reverse order would be bad, unless the temperature outside is below freezing.

Consider the *code fragments* on the left. They both have the same blocks. But they are not the same. Your robot will wind up at a different spot when you run one versus the other (try it). For computer programs, **the order of instructions is important.**

Computer-instructions are simply the specific things you want the computer to do in terms that the computer can understand. At right is a simple RIS *program*. When you download the program and run it, your robot goes forward for 1 second.

Two terms have just been used: code fragment and program. Both are ordered-lists of computer-instructions. The difference is that a program is a complete list while a fragment is not. You can download and run a program, you can't run a fragment. Besides a list of instructions, a program needs a name. The program at right is name “untitled”. That is the name for programs before they are saved (you can, in fact, save a program with the name “untitled” but that would be a bad idea). In RIS code, as in Java, the name of a program is also the name of the file that the program is saved in.



Algorithms

An *algorithm* is simply detailed instructions to accomplish a task. Note that we did not say computer instructions. An algorithm is written in a natural language like English. To put that into context, having a robot go through a maze is a task. In the previous section, a program was presented that was intended to maneuver a robot through a maze. That program is the implementation of an algorithm. So, what is the algorithm?

The objective of the maze runner is to have the robot move from the start to the end of the maze. So, a key part of the algorithm is for the robot to continually move forward. However, simply going forward won't work (or, it violates the rule of mazes that climbing over a fence is not allowed). Whenever the robot runs into a fence, it cannot go forward anymore. So it has to change its state, blocked by a fence, to a state where it can continue to move forward. The robot needs to back up and turn away from the fence so it is unblocked and then continue forward. The algorithm is to go forward when unblocked and to turn away from the blocking fence when blocked.

If you look at the RIS code for [Maze Runner](#), you see that this is exactly what happens. The robot starts by going forward. When it touches a fence on the right side, it backs up, turns left (away from the fence), and goes forward. Likewise, when it touches a fence on the left side, it does a mirror image maneuver, backing up, turning right (away from the fence), and proceeding forward.

There are two important considerations in implementing algorithms in programs: *robustness* and *calibration*. Robustness is how well an algorithm works in different situations. Being more robust is, in general, better. Calibration deals with adapting an algorithm to the real world. In Maze Runner, the times are wrong. The time of 1.0 second for moving forward is probably too short while the backing up and turning times are probably too long. The word "probably" is used because, until you test, you cannot be sure. After you try running your program, it is quite likely that you will find that you need a better algorithm.

When we move on to programming in Java, Line Follower and Maze Runner are programs that you will write. The algorithms, at least at first, are the same for a program in Java and a program in RIS code. But you will find that it is easier to implement more complex algorithms in Java than it is in RIS code.

Program flow



RIS uses color to distinguish between different types of program elements. Green blocks are used for regular instructions and blue blocks start programs or *tasks* (sub-programs). At left is an updated version of MazeRunner. This one includes purple blocks for conditional execution, if-then, and light green blocks for manipulating variables. This version of MazeRunner has an improved algorithm that deals with corner oscillation. RIS allows you to have several variables. Here I created 2, x and xx¹.

x is used to keep track of which side was last touched. A negative number indicates that the sensor on 3 was last touched while a positive number indicates that sensor 1 was last touched. In Java, I would have used a boolean variable, a variable that can have only the values true or false, and named the variable side1LastTouched. However, RIS code does not have boolean variables so we use what we have.

xx is used to keep track of number of consecutive, alternating touches. Again, this is hard to put into words and watching your robot do it makes it clear what is going on. Put simply, if we continue to touch on alternate sides, we are probably trapped in a corner. If we are trapped, we need to something really different to

get back to moving forward through the maze.

The purple blocks are if, or conditional, statements. Examine the code for sensor 3. The first block tests variable x. If x is positive, that means that sensor 3 was the last sensor pressed and that we are alternating pressing sensors. So, we add one to the alternating pressed counter and continue in the normal way. If however x is negative then we have not had alternating touches. In this case, we set the counter back to zero and continue, knowing that we are not oscillating in a corner. The code for counting touches for sensor 1 mirrors

¹ The reason for the names is an apparent bug in RIS that allowed me to only create variables with those names. Ordinarily, I would use something like sideTouched and touchCount for these variable names.

the sensor 3 code just like the back up and turn away from the wall code for each sensor mirrors the other. Finally, the alternating pressed counter has a watcher. When the count gets up to five, it fires and turns the robot much more than usual. After we have alternated five touches, we assume we are stuck. Hopefully, the spin maneuver will get us out of the corner.

The new algorithm builds on the original MazeRunner. We enhance the old algorithm by counting the alternative touches. If we have 5 alternative touches in a row, we assume we are trapped in a corner. When that happens, instead of backing up, turning a little and going forward, we back up, spin a lot more than usual and then starting going forward. By spinning for a longer time, we hope to break the oscillating cycle in the corner. This algorithm is more robust than the old algorithm. But it will need to be calibrated to determine good parameters for the “get out of corner” code fragment.

Experiment with this new MazeRunner, varying the back up time and spin time to see if you can make them work. Then, think about a better way of determining that you are stuck in a corner and how to get out. If you have a rotation sensor, think about how it might help you create a much more robust MazeRunner.

Interrupts and polling



At left is the maze runner program that was introduced in a previous section. It is a program because it has a blue program block on the far left. It also has two additional blue blocks. These are sensor blocks; when the condition specified on the block is met the code fragment attached to it is executed. This is a very sophisticated programming technique, one that most beginning programmers don't start using until after many years of study.

Each blue block is a separate *task* and each task appears to run on a separate *thread*. A task is an independent sub-program that is started whenever its entry conditions are met. Maze runner is composed of three tasks: main, right-button pressed and left-button pressed. This is called *interrupt driven* programming. Main is the first task started when the program is run. When one of the sensors is pressed, the main task is interrupted and the code list attached to the sensor starts running. When the interrupt code completes, the main task resumes running at the point where it was interrupted. In this case, the main task has completed, it takes only 2 seconds, so the program simply waits for the sensors to be pressed again.

As we said previously, the old fashioned definition of a program is an ordered-list of computer-instructions. The program at right is MazeRunner2 and is written in the old fashioned style. The program starts with the same "Forward 2.0 sec." block as in MazeRunner. Then it starts an orange loop, "Repeat Forever". In the loop is a purple "If" statement. If sensor 1 is pressed, the same program fragment is executed as above. If sensor 1 is not pressed then sensor 3 is tested.



If it is pressed then the sensor 3 program fragment is executed. If neither sensor is pressed, then the *branch* of the if statement with no instructions is executed. In all three cases, the next statement executed is the forever statement. This puts the instruction pointer back to the repeat statement and sensor 1 is tested again. This process repeats forever or until you turn off your RCX, whichever comes first. This programming technique is called *polling*. Note that MazeRunner and MazeRunner2 are similar but not the same. In the interrupt driven environment, a pressed sensor interrupts the currently running task even if it is in an interrupt service routine. In the polling technique, the fragment runs to completion before the sensors are checked again.