ROBOLAB™ 2.5 Software Training Lab

By Joel Stone
Revised for ROBOLAB by Doug Frevert
September 12, 2002
Version 0.8.2

Version 9/6/2002

Copyright and Trademark Notice

© 2002 INSciTE in agreement with, and permission from FIRST and the LEGO Group. This document is developed by INSciTE and is not an official FLL document from FIRST and the LEGO Group. This document may be freely copied and distributed, electronically or otherwise, in its entirety only, and only if used in conjunction with FIRST™ LEGO® League. Any use, reproduction, or duplication of this manual for purposes other than directly related to FIRST LEGO League is strictly prohibited without specific written permission from INSciTE.

LEGO®, ROBOLAB, and MINDSTORMS™ are trademarks of the LEGO Group used here with special permission. FIRST™ LEGO® League is a trademark owned by FIRST (For Inspiration and Recognition of Science and Technology) and the LEGO Group used here with special permission. INSciTE™ is a trademark of Innovations in Science and Technology Education.

INSciTE
PO Box 41221
Plymouth, MN 55441

Table of Contents

ROBOLAB 2.5 Software Training Lab 1
Copyright and Trademark Notice 1
Table of Contents 2
Introduction to ROBOLAB Programming 1
Requirements to use this lesson document: 1
The Hardware 1
The Software 1
Preparation 2
Loading Firmware into RCX 2
The Test RCX Communication screen 4
The RCX Settings screen 4
To Reinstall Firmware 4
To Set the COM Port 5
Back to the ROBOLAB Main Menu 5
Programmer 6
Take some time to examine the programming area. 7
Building your practice robot 9
Coaching Suggestions 9
Programming task 1: Move forward for 5 seconds & return. 11
Requirements 11
Command selection 11
Creating a New Program 11
Saving Your Program for the First Time 13
Opening a Saved Program 14
Download Your Program 14
Run, Test, Debug 14
Comments 17
Repeat the run, test, debug steps as required. 17
Requirements changes 17
Algorithm creation 17
Topics learned in task 1: 18
Programming task 2: Move forward for 5 seconds & turn right. 19
Requirements 19
Command selection 19
Start a New Program 19
Programming the turn 21
Rewrite the MOTOR A and C REVERSE functions 21
Turn: MOTOR A and MOTOR C in opposite directions 21
Testing the new program 22
Program slots – pitfall to avoid!! 22
Topics learned in task 2: 22
Programming task 3: Subroutines (make a square) 23
Requirements 23
Command selection 23
Copy/Paste 23
Problems with our current program 24

How to dramatically improve the program ... 24
Loops ... 24
Local subroutines. ... 26
Shared subroutines or VIs. ... 27
A SubVI Walkthrough ... 28
Naming Subroutines ... 30
Saving your program ... 30
Topics learned in task 3: ... 31
Programming task 4: Rotation sensor ... 33
Requirements ... 33
Structure of programming ... 33
Rotation sensor introduction ... 33
Manually measuring distance with the RCX ... 33
Activating a robot's rotation sensor ... 34
Measuring by moving the robot ... 35
Programming with the rotation sensor ... 35
Resetting the rotation sensor ... 36
Build a Forward_50 program ... 36
Using variables to expand the robot's capabilities ... 36
Turning using the rotation sensor ... 38
Caveat of using rotation sensor ... 38
Topics learned in task 4: ... 38
Programming task 5: Forward to a black line, stop, turn right ... 39
Requirements ... 39
Light sensor ... 39
Light sensor programming ... 39
Write a new program named " Fwd2Line" ... 39
Calling a subroutine ... 41
Changing light conditions ... 42
Additional notes ... 43
Which commands are used most often? ... 43
Which commands should be avoided? ... 43
Why do sensors and containers need to be reset to zero? ... 43
Limiting the commands available in ROBOLAB ... 44
Waiting for several sensors at the same time. ... 44
Wait for light sensor event OR touch sensor event ... 45
Debugging a program ... 46
Step through the program ... 46
Peer Review ... 46
Add Debugging Code ... 46
Interrogate the RCX ... 46
Last Resorts ... 47
Backing up your kid's program code ... 47
Robot Construction ... 49
Parts List: ... 49
Instructions: ... 49

Introduction to ROBOLAB Programming

Requirements to use this lesson document:

1. LEGO Robotics Invention System (The large box with the 700+ Lego parts – any version will work fine).
2. One LEGO rotation sensor.
3. ROBOLAB 2.5 software (2.0 will also work.)
4. PC. Either a Windows PC or a MAC. (A notebook is best, but a desktop PC will do.)
5. White-topped table

The Hardware

The brain of the Robotics Invention System is the RCX, which stands for Robotics Command System. The RCX is a microcomputer, built into a yellow LEGO brick. The RCX uses sensors (such as touch and light) to take input from its environment. It can then process the data to make motors turn on and off.

To perform the lessons in this class, you will build a simple robot using a minimum number of parts.

The Software

FLL allows participating teams to use one of two programming languages, RIS (Robotics Invention System) and ROBOLAB. This class uses ROBOLAB.

For teams using MACs, ROBOLAB is the only choice. ROBOLAB is based on LabVIEW, which allows programming capabilities well beyond those needed for success in FLL competitions. In contrast, many people think that RIS is easier to learn than ROBOLAB. Neither language seems to have an edge in the competition.

Your kids may use ROBOLAB 2.0. Version 2.5 adds some capabilities, including the ability to connect the USB IR tower that became standard with the Mindstorm 2.0 kits.

When **ROBOLAB** or RIS programs are downloaded to the RCX, the programs are a type called “embedded code”. Embedded water in a rock means the water is “stuck” inside the rock. Embedded code means that the program is “stuck” inside the RCX – there is no way to change the code after it is downloaded. Programs can only be modified on the PC!

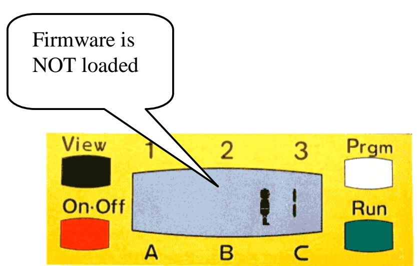
When kids program on the PC, they are entering commands in ROBOLAB or RIS. ROBOLAB or RIS use human readable commands. When downloading the program to the RCX, the PC converts the RIS or ROBOLAB commands to bytecode. Firmware, in the yellow brick, converts bytecode into machine code. which a microprocessor in the yellow brick understands. Machine code is easy for machines but does not make much sense to humans.

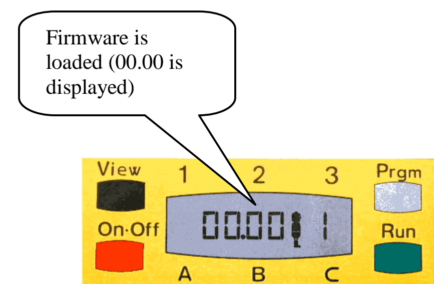
Preparation

Firmware is a special program that must be installed onto your RCX so that you can program the RCX brick.

This preparation step is required when the RCX is new. If the RCX loses its firmware in the future for some reason (such as changing batteries too slowly), or starts behaving badly, you may want to reload the firmware.

The RCX brick must be set up properly prior to downloading your team's programs into it. The RCX brick must contain the proper version of firmware in order to proceed. The following photos show the RCX screen without and with firmware loaded.



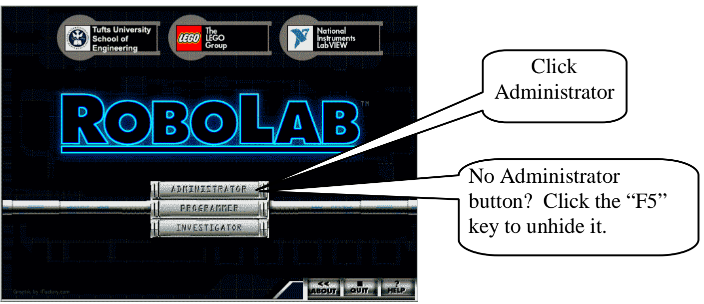


Unfortunately, even if you see that firmware is loaded, it still may be the wrong version of firmware.

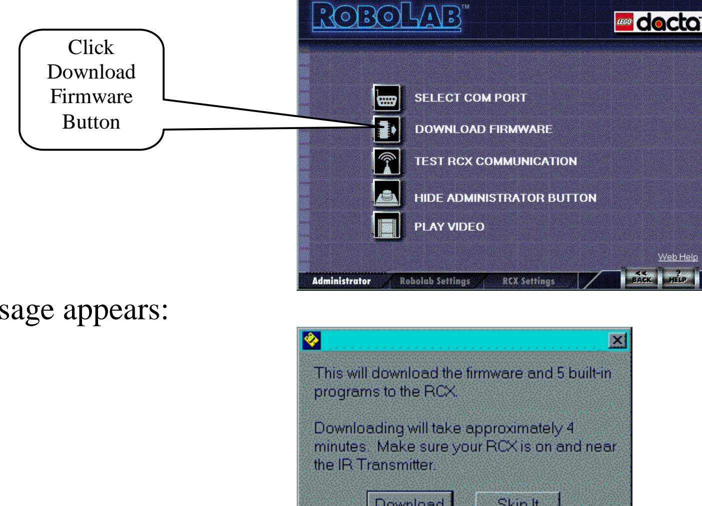
Loading Firmware into RCX

To load firmware into the RCX or verify the firmware version, start the **R45oboLab** program and go to the Administrator screen.

1. Start up **ROBOLAB**.
2. Select **ADMINISTRATOR.**



3. Select **DOWNLOAD FIRMWARE.**



This warning message appears:

Connect the IR tower to your computer and turn on the RCX yellow brick. Make sure the switch on the front of the IR Tower is moved to the left. That is the low power setting.

Point the RCX towards the tower – the black lens on the RCX should be facing the tower.

4. Select **Download.**
The IR Tower's green light should turn on. The RCX should start counting up from 1 to about 2400 (1620 for ROBOLAB 2.0 firmware).

Communication Successful

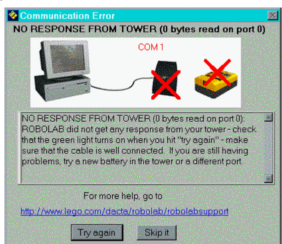
OK

If successful this message appears:

5. Select **OK.** You're done.

Troubleshooting

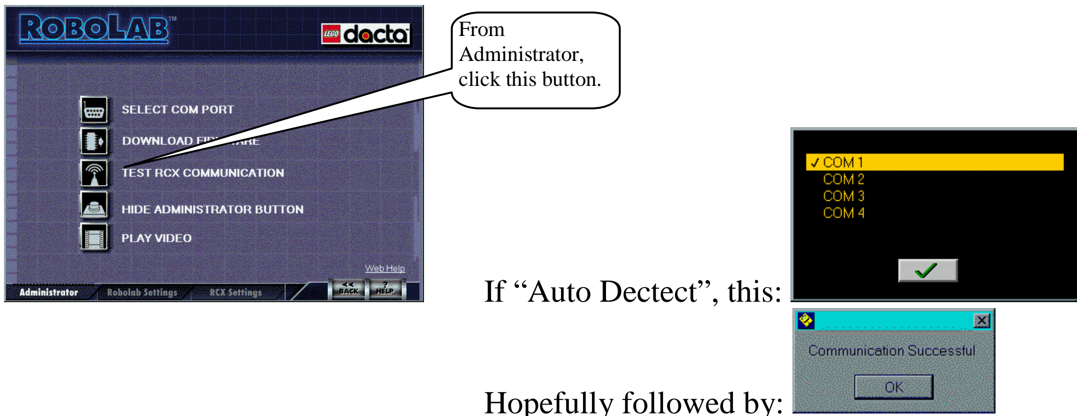
a. The IR Tower's green light doesn't appear. This usually indicates that the PC is not sending data to the IR Tower. Check the cable. Check the COM or USB port settings. This window often appears to indicate a loose cable:



b. During the download this window appears:

IR Tower is connected, but can't find the RCX.
Turn on the RCX.
Place the RCX about 6 inches from the IR Tower.
Check the IR Tower battery (9 volt).

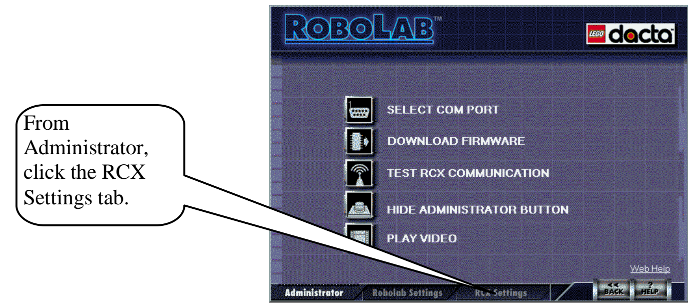
The Test RCX Communication screen



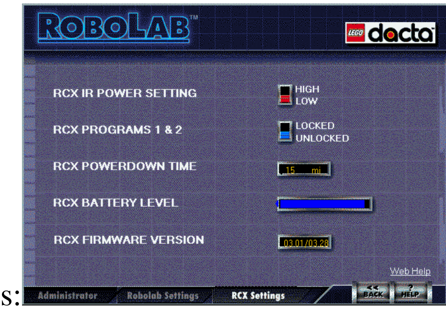
If "Auto Dectect", this:

Hopefully followed by:

The RCX Settings screen



To get this:



The "RCX Settings" menu options should be set as shown.

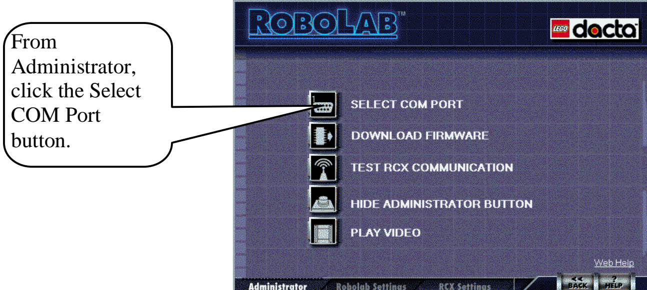
RCX IR Power Setting	**Low**
RCX Programs 1 & 2	**Unlocked**
RCX Powerdown Time	15 minutes *(5 min. to save batteries, 30 min. to lower aggravation)*
RCX Battery Level	*should be about 9 volts*
RCX Firmware Version	*03.01/03.09 (ROBOLAB 2.0) 03.01/03.28 (ROBOLAB 2.5)*

To Reinstall Firmware

1. Place the RCX 6 inches from the tower.
2. Turn on the RCX and point the RCX black window towards the tower.

3. From the **ADMINISTRATOR screen,** click "**Download Firmware**"
4. Once firmware is installed, you are ready to start programming!

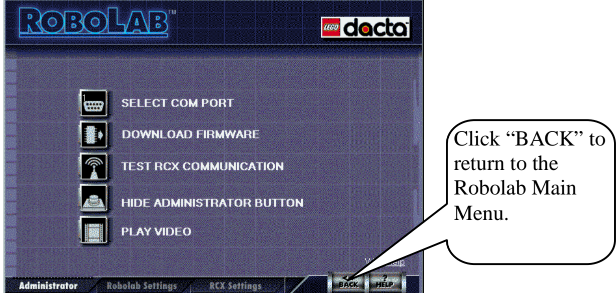
To Set the COM Port



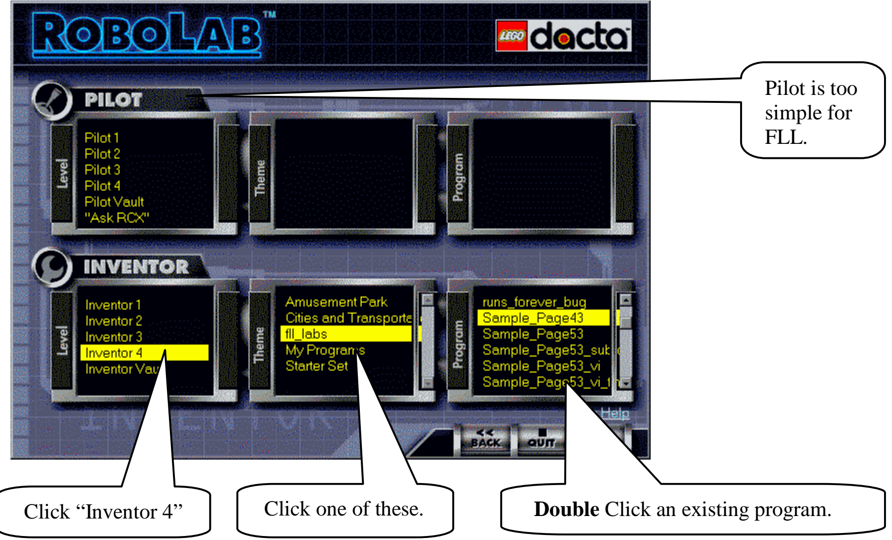
Pick AUTODETECT

and click the green checkmark. Now click the TEST RCX COMMUNICATION button.

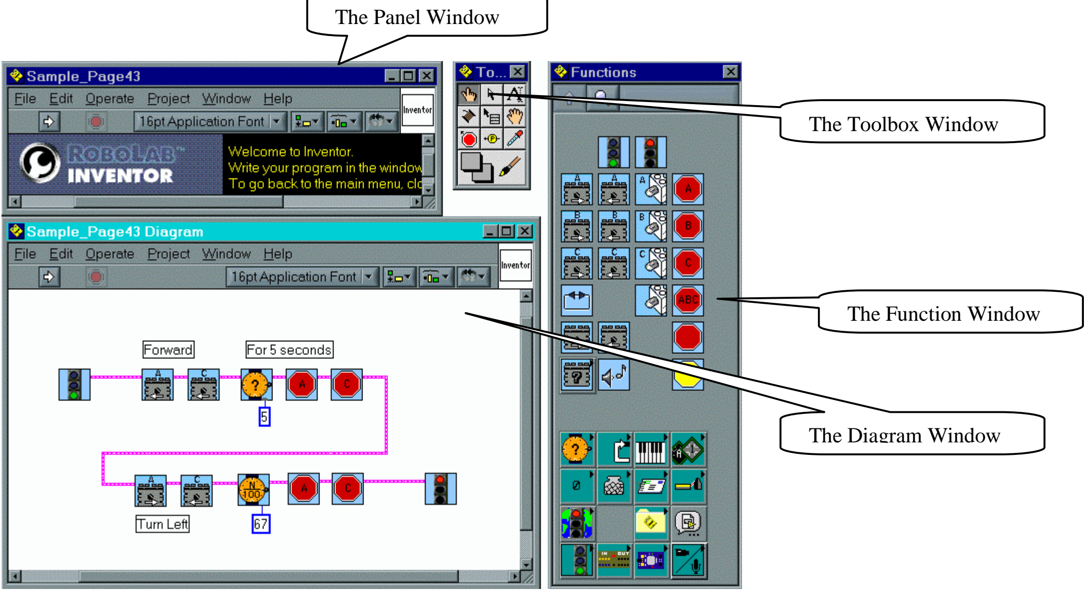
Back to the ROBOLAB Main Menu



Programmer



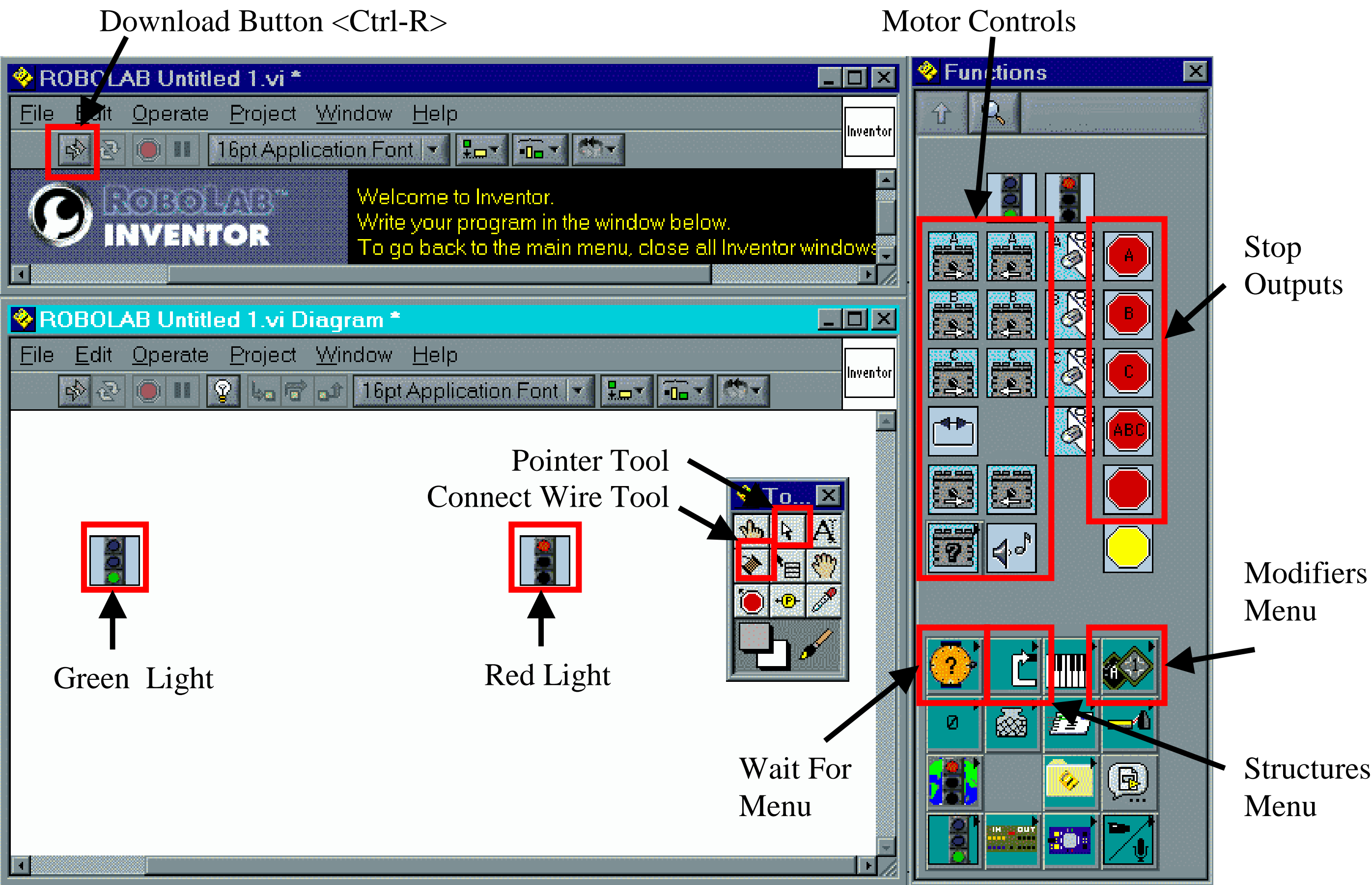
Select **Inventor 4**, any Theme, then double click a program and jump to programming:



You are now ready to edit the selected program.

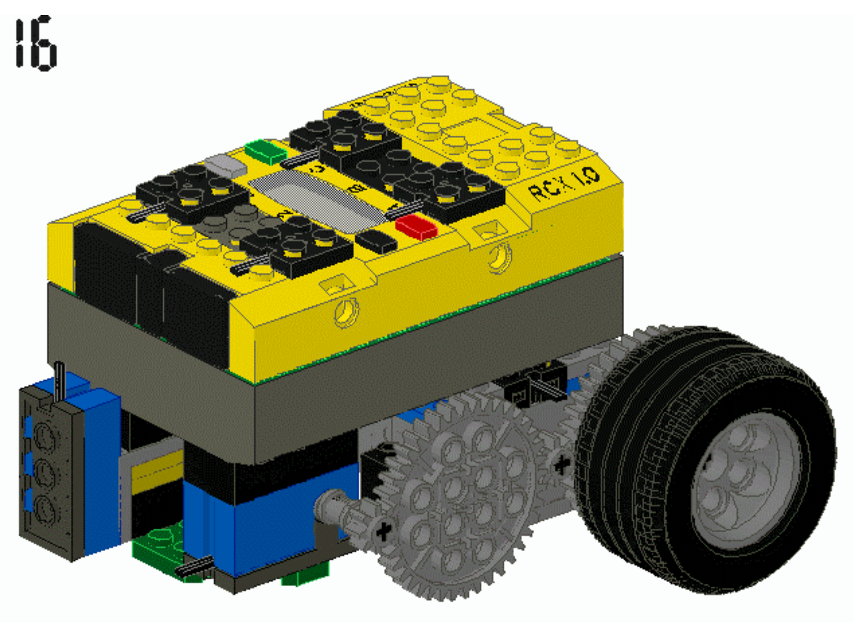
Take some time to examine the programming area.

Make sure that both the function palette and the tool palette are open. If not select **show tool palette** and **show function palette** under the **window** menu in the program window.



Take a moment in your group to review each of the icons as described above. Feel free to click the menu on the right side so that you can see the different commands that you have access to. Try using Context Help to investigate the various commands.

Building your practice robot



Before you can start programming, you must build a robot. We have created a simple-to-build robot to use for this class. The following requirements were met when designing this robot:

1. Build time less than 10 minutes
2. Simple design (too simple to be of much use in the competition)
3. Minimum possible LEGO parts used, but still support use of rotation and light sensors for learning.

The parts list and build diagram are located at the end of this document.

Coaching Suggestions

This section belongs in the material for "Coaching a Problem Solving Team."

Divide your team into groups of builders, programmers, and possibly testers. If you have testers, they will be responsible for running the programs and making observations about the performance of the robots. However, everyone should get a chance to observe the 2 robots move around while executing the program. They can then see how the exact same program will run differently depending on the engineering design and the environment. (If you only have one RCX programmable brick, then one robot will have to be built to completion, tested, then taken apart slightly to use the RCX brick for the other robot.)

You'll have to figure out how to divide up the groups appropriately. You probably want everyone to get an opportunity to try programming and building. However, the building (especially if a group is only to make 1 robot) will probably go much faster than the programming. Therefore, you might have to come up with other

activities for the builders to work on while the programmers are finishing up. If you have a testing group, then they will have to wait until both groups are finished in order to start their task.

Coaches in the past have found that assigning specific roles to each group member works best. For the builders, one can find the pieces and the other can build the robot. In programming, one can read the directions and one can operate the computer. For the testers, one can operate the robot while the other writes down observations. The kids should be trading roles so that they can have as much exposure to each aspect of building and programming as possible. For the building, roles can be swapped when building the second robot. For programming, roles can be swapped after the program has been saved for the first time. For testing, roles can be swapped between each of the two robots.

Here are some questions that you can ask the kids as the robots are being tested:

- Did the robot go in a perfect square? (It most likely won't.)
- Why didn't the robot go in a perfect square?
  A: The turns were not perfect 90 degree turns and/or the robot did not travel straight.
- What are the two things that you can change in the program to change the degrees of the turn?
  A: You can change the motor speed or the amount of time that the robot turns for.
- What could you change in the robot design to change the degrees of the turn?
  A: wheel size.
- How did the performance of the same robot differ when on different surfaces?
- How did the performance of the two robots differ when on the same surface?
- Why do you think there was a difference?

Programming task 1: Move forward for 5 seconds & return.

Time: 30 minutes (20 minutes to create and run program + 10 minutes Q&A)

Requirements

The first step in programming involves stating what needs to be accomplished. For this mission, we want to move forward for 5 seconds and then return. Given these requirements, the kids need to think of how to accomplish the task. With the robot that we are using, simply turning on both motors for 5 seconds will get us half way. Run the motors in the opposite direction to finish the objective.

Programmers break requirements written in English into a “not quite English” psuedocode. It’s not English and it’s not programming code, it’s somewhere in between. For example:

English: Move forward for 5 seconds and return.

Pseudocode: Motor A on, forward.
Motor C on, forward.
Wait 5 seconds
Flip motor directions.
Wait 5 seconds.
Stop Motor A.
Stop Motor C.

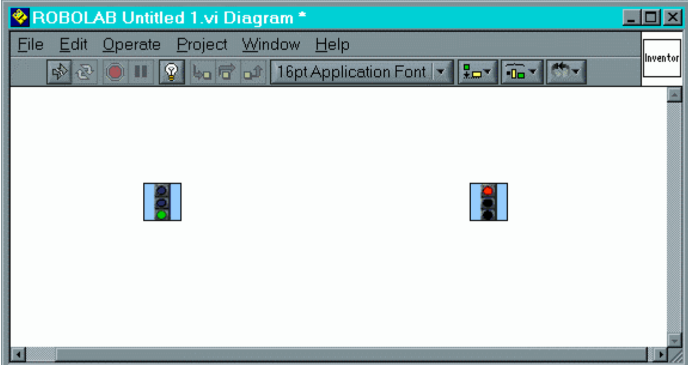
Together, these steps are called an *algorithm*. An algorithm is a step-by-step plan of what the program should do, but it does not indicate how exactly to perform those steps. It is important that you come up with an algorithm *before* you start creating your program on the computer.

Command selection

Creating a New Program

First, start a new program. From the programming area’s menu, <File><New> or from the keyboard <Ctrl-N>. You should now have an “Untitled 1.vi” program to edit. It should have a “green light” and a “red light”.

For example:



The "Untitled 1.vi" program's diagram window.

Now, each line of pseudocode needs to be translated into a ROBOLAB function. The functions appear in the "Functions" window, or a sub-window it. The MOTOR A FORWARD, MOTOR C FORWARD, FLIP DIRECTION, STOP A, and STOP C functions are all on the "Functions" window. Use the pointer tool to select each function and then move your cursor to the diagram window and click to plop it down.

Which leaves the "WAIT FOR TIME" function, which is on a sub-window. Find the watch face with a question mark. Click it to reveal the "WAIT FOR" window.

Functions

Motor A Forward

Stop A

Flip Direction

Reveals the "WAIT FOR" window.

Wait For

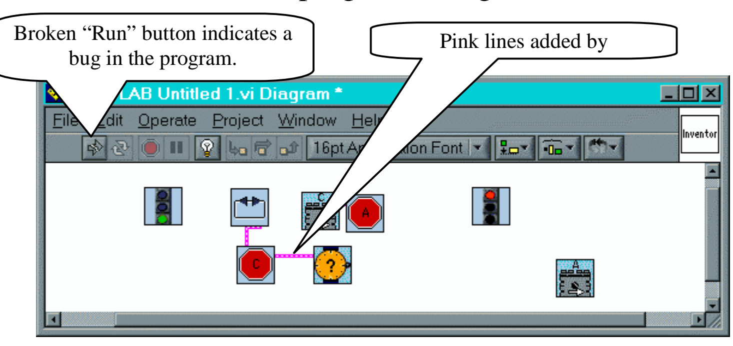
Wait for 1 sec

Wait for seconds.

Pick the WAIT FOR TIME function and plop it into the "Untitled 1.vi" program's diagram window.

By now, you've noticed that ROBOLAB is adding pink lines between the functions. This is a new feature of version 2.5. When it works, it's great.

So now, "Untitled 1.vi" program's diagram window looks like:

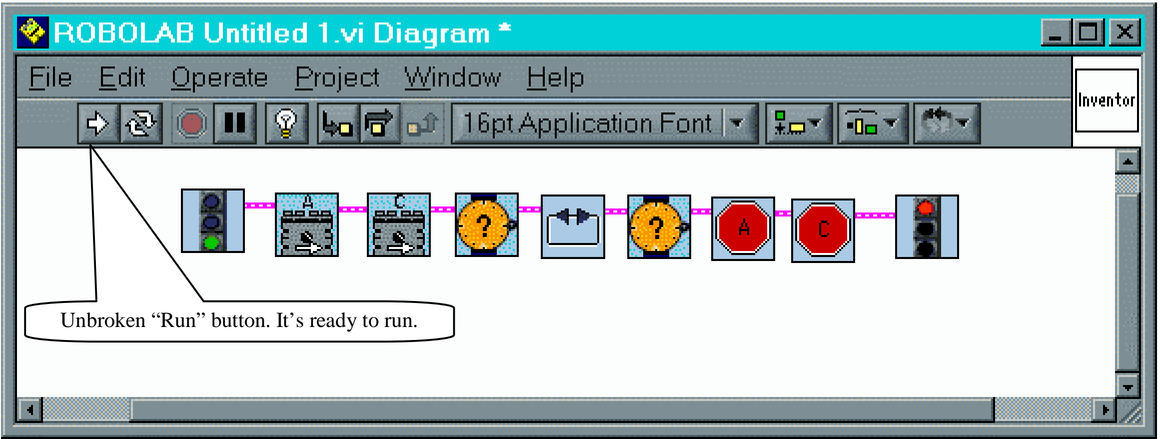


Use the pointer tool to select and drag the functions into position. Use the pseudocode as a guide. Place functions in order left to right and top down.

Some wires added by ROBOLAB will not make sense. Select and delete them.

Use the connect wire tool to connect the functions. The "End" of one function is wired to the "Begin" of the next function.

Does it look like this?



Saving Your Program for the First Time

1. Select the **File** menu and select **Save As...**

2. When the dialog box opens, select and delete the name '**ROBOLAB Untitled 1'.**

3. Type in the box an appropriate name for this program such as '**Forward_And_Return**'.

4. Press the Save button.

Your Program is now saved

5. Click the **X** in the upper right hand corner of the window to close the programming environment and exit the program by pressing the quit button in the lower left hand corner of the program.

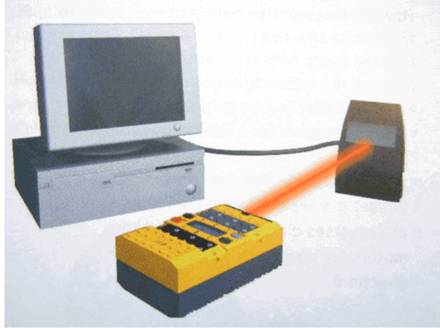
Opening a Saved Program

1. Open **ROBOLAB**.

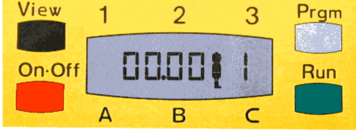
2. Click on **Inventor Level 4** and then on **My Programs**. After that, on the far right you should see your program. Double click on the file to open it.

Download Your Program

1. Place your RCX about 6 inches in front of the transmitter so that the infrared window is facing the transmitter.



2. On the RCX, press the “Prgm” button to select the program “slot” that you want the program downloaded to.



3. To send your program to the robot (also known as downloading), do one of the following:

   Type <Ctrl-R> (<Command-R> on a MAC);
   click the “Run” button on the command bar;
   or from the Menu, select <Operate><Run>.

   If this is the first time downloading, the program will search for the port connected to the IR Tower. In addition, if your RCX requires firmware, it will be downloaded at this time.

4. When your program finishes downloading, the RCX will play the “Fast Rising Sweep” sound.

Run, Test, Debug

Recall that our algorithm required that we go straight for 5 seconds and return. Try it. Press the green “Run” button on the RCX (the yellow brick).

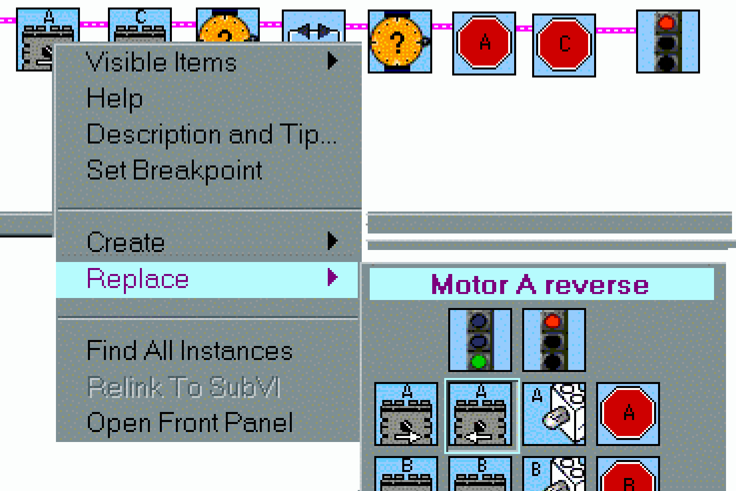
It didn't work? What went wrong? Finding out is called "debugging." Debugging is an important part of programming. Luckily, while you might not see the problem by looking at the code, you will see the problem by running the robot.

Two problems or "bugs" are expected.

1. The robot moved backward instead of forward.
2. The robot only moved forward and backward for one second, not five.

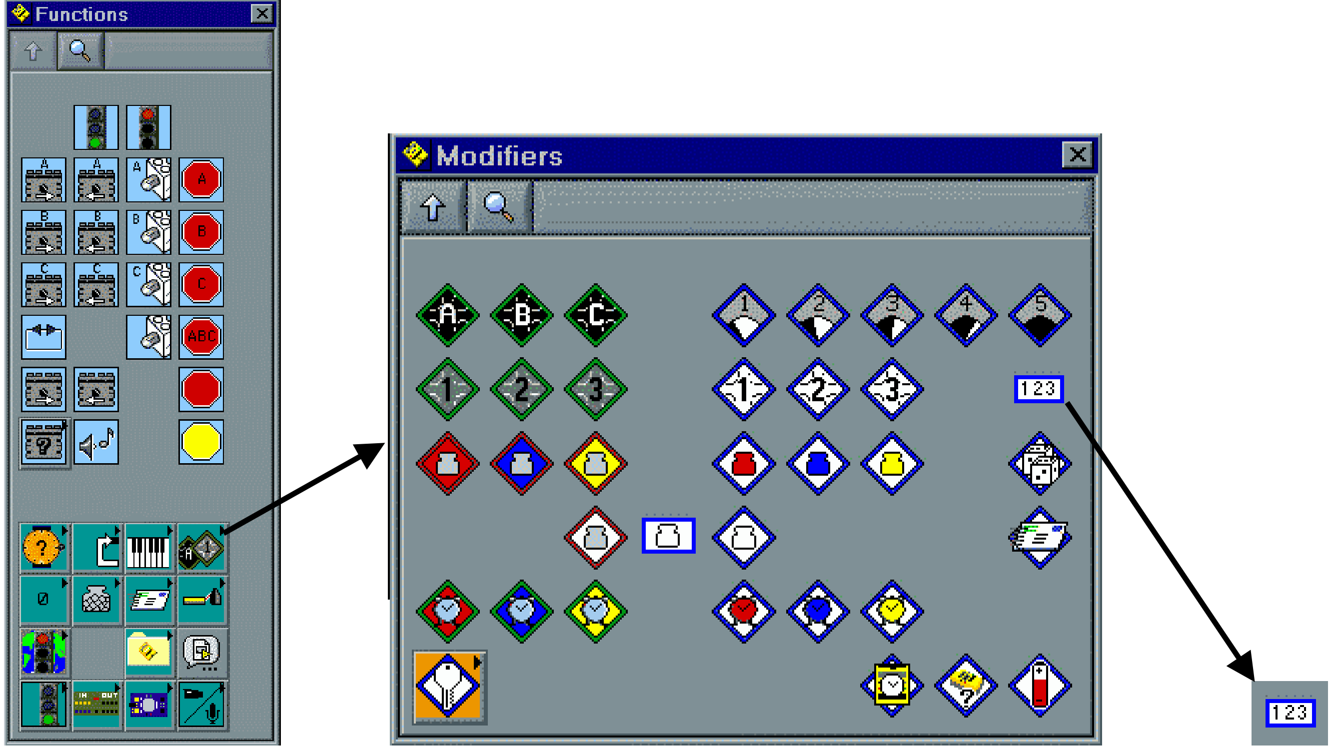
To fix #1, either rewire the robot or rewrite the program. This is a programming lab, so let's fix the program. To change the MOTOR A FORWARD function into a MOTOR A REVERSE function:

1. Select the "Object Shortcut Menu" tool from the toolbox.
2. Use the tool to select the MOTOR A FORWARD function. From the menu that appears, select "Replace".
3. The functions window appears. Pick the "MOTOR A REVERSE" function.



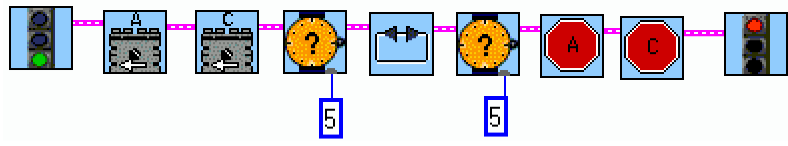
4. Repeat for the motor C function.

To fix bug #2, add a modifier to the “WAIT FOR TIME” function. Most functions have a default value that can be modified by connecting the function to a modifier. The modifiers are a special type of function. The modifiers are on a sub-window of the “Functions” window.



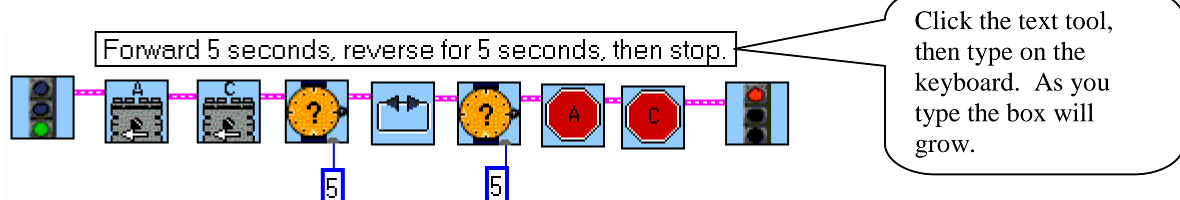
1. Use the pointer tool.
2. Select the “Numeric Constant” modifier.
3. Drop the “Numeric Constant” modifier near one of the “WAIT FOR TIME” functions of the “Forward_And_Return.vi” program. Do it carefully and ROBOLAB will autowire it.
4. Type “5”. If you clicked away from the “Numeric Constant” modifier, use the text tool to change it.
5. Repeat for the other “WAIT FOR TIME” function.

For example:



Comments

One of the most important tasks in the creation of a program is to document what the program does. Use the text tool to add comments.



In addition, comments can be entered from the menu <File><VI Properties . . .> select the “Documentation” category and enter a “VI Description”. These comments appear in the context sensitive help for the program. While not vital for all programs, this type of comment is very important for Sub VIs (see subprograms).

Repeat the run, test, debug steps as required.

Requirements changes

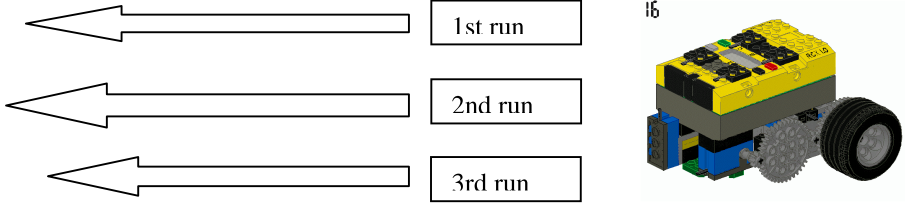
After running the program and watching it perform, your kids will undoubtedly want to change either the requirements or the method they used to meet the requirements.

Try changing your program to go forward for 0.5 seconds, or 20 seconds. Make it go forward and **not** return.

Algorithm creation

When you made decisions about how to approach the requirements, you created an algorithm, or a series of steps to meet the goals of the task. These steps were turned into program commands.

Try running your robot using a “Go Forward and Stop” program three times starting from the same spot.



When your robot moved forward for 5 seconds, did it cover the same distance each time you ran the program?

As the batteries wear down, will the robot travel the same distance each time the program is run?

Think about the FLL challenge: Is it important to move the same distance each time the program is run?

When is it important to move the exact same distance each time the program is run?

- During a turn would be a good example – you don't want an overturn or an underturn.
- Another example would be when approaching an object to be retrieved. You don't want to knock over the object, and you don't want to come up short!

Try to think about other methods that you could use to travel a distance more accurately and consistently.

Topics learned in task 1:

Congratulations! You have successfully programmed your robot, and have learned the following topics in programming task 1:

- programming editor navigation
- selection of commands
- variety of commands available
- how to add comments
- thought processes required to move your robot
- housekeeping tasks required in programming
- sequencing of commands
- shortcomings inherent with program as designed
- distance traveled using time is not accurate

Programming task 2: Move forward for 5 seconds & turn right.

Time: 30 minutes (20 minutes to create and run program + 10 minutes Q&A)

Requirements

This lesson builds upon the previous program. After moving forward for 5 seconds, we now want the robot to perform a 90 degree right turn. To perform these requirements, we will simply turn on both of the motors, but one motor will be run in reverse. This will cause a rotation of the robot, resulting in a turn.

Command selection

To determine what a robot must do to accomplish a task, it is often useful to develop analogies to share with your team. Compare something that they don't understand to something they already understand.

For example, lets create an analogy about starting our robot moving forward. The analogy will be with a person walking. Before starting to walk, what must a person think about? Several "set-up" items should come to mind.

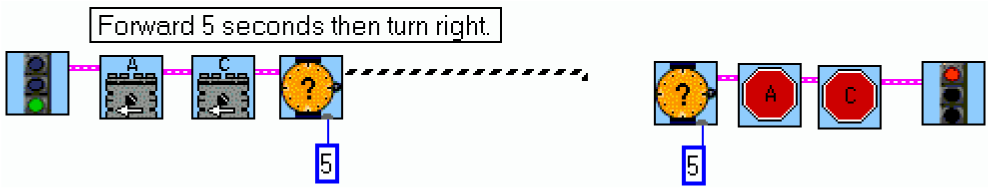
1. What direction am I going to walk?
2. How much power am I going to exert while walking?
3. When will I begin my walk?
4. What will cause me to stop walking?

These steps translate directly into robot commands. The first lesson was over-simplified, but worked OK. In this lesson we will be more thorough in describing the robot's travels.

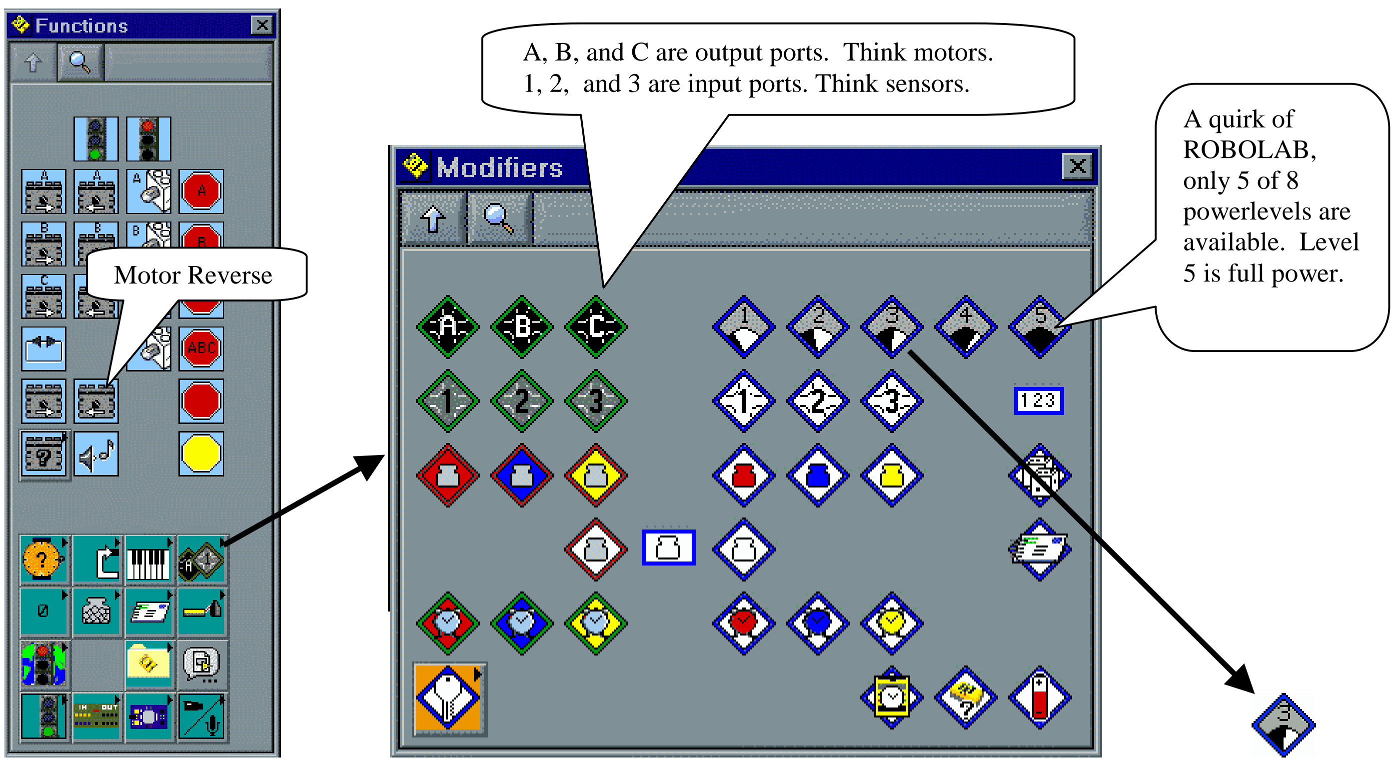
Start a New Program

To start a new program there are, as usual, several options.
1. Rename and then edit an existing program.
2. Start with a blank slate, a new program. (See "Creating a New Program", or just type <Ctrl-N>.)

To Rename and then edit an existing program, start by opening an existing program. After opening an existing file, from the menu <File><Save As> and enter the name of the new program, like "Forward_5sec_TurnRight". To remove functions from the previous program, use the pointer tool to select and then delete them. For example:

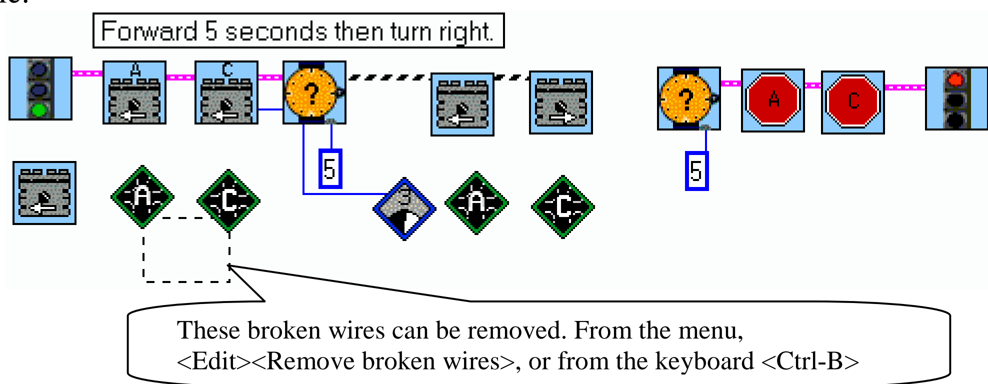


Following the human analogy above, we must first set the POWER and the DIRECTION of each motor. In Lab #1, we used functions like MOTOR A FORWARD that only worked for a motor wired to output A. This time, a generic MOTOR FORWARD function will be modified with a OUTPUT A and POWER LEVEL 3 modifier function. The modifiers are found on the modifiers sub-window of the functions window.



1. Use the pointer tool.
2. Select and drop into the program the “Output A”, “Output C”, “Power Level 3” modifiers.
3. Select and drop “Motor Reverse” from the “Functions” window.

For example:



Programming the turn

To program the robot to turn, we want the motors to spin in the opposite direction for a short time. Can you think of a way to do this?

The only function that needs to change is the direction of one motor. The time that the motors are turned on will need to be changed. For that, edit the modifier. In addition, set the power level of the motors to 3. Some teams find that turns are more accurate if the motors don't run at full power.

Rewrite the MOTOR A and C REVERSE functions

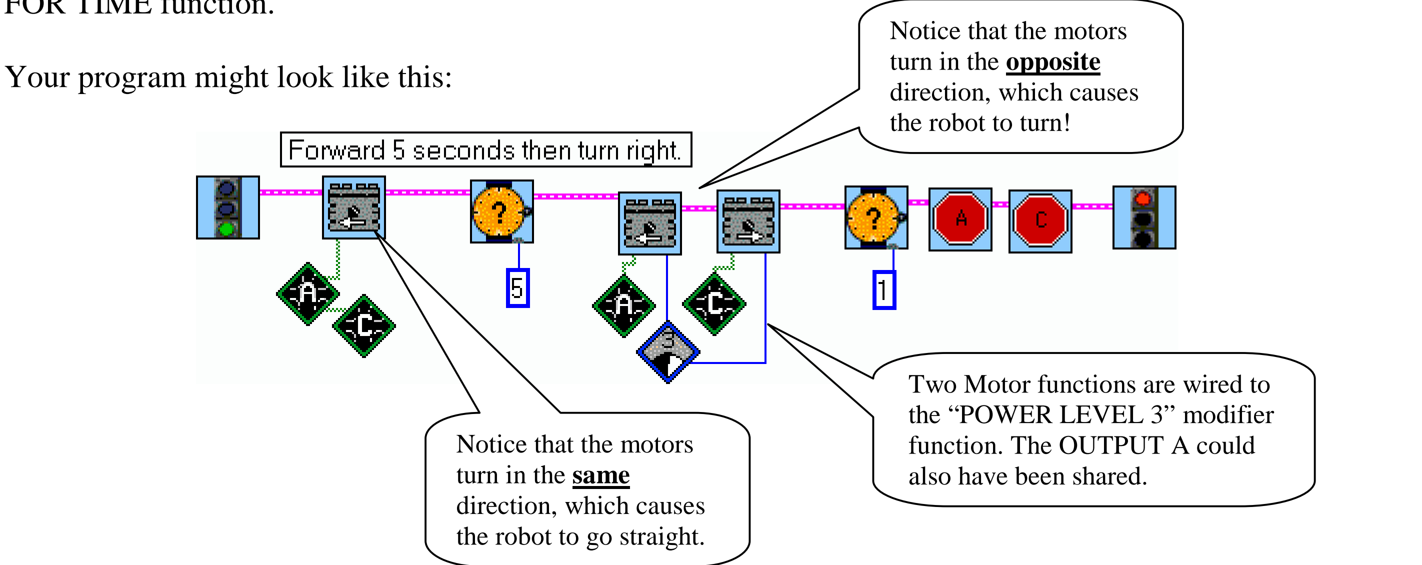
Change MOTOR A REVERSE to a MOTOR REVERSE function wired to modifier OUTPUT A. Now delete the MOTOR C REVERSE function and wire OUTPUT A to OUTPUT C. What's the difference? Not much. You might think that the new form starts the motors at exactly the same time, while the first form starts them one after the other. Don't assume it works that way (see Firmware Note). Test it. In most cases it is more important to use the form that makes sense to the programmer.

Firmware Note: Remember the firmware? Downloaded programs are bytecodes that the firmware converts into machine code that is understood by the processor chip in the RCX. It turns out that the bytecode pattern that starts a motor requires a port, a direction, and a power setting. There isn't a bytecode to turn on both motor A and motor C. ROBOLAB converts both forms into two sets of bytecode. **Lesson 1:** don't assume that one function is faster or more reliable than another. Test it. **Lesson 2:** the firmware often pauses for 1/500th sec. between bytecodes to read the sensors and update the LCD. Don't worry, the mechanical and logic problems will overwhelm this subtle problem.

Turn: MOTOR A and MOTOR C in opposite directions.

Go ahead and wire OUTPUT A, OUTPUT C, MOTOR REVERSE, MOTOR FORWARD and POWER LEVEL 3 functions to turn the robot to the right. Edit the NUMERICAL CONSTANT modifier of the WAIT FOR TIME function.

Your program might look like this:



Testing the new program

Now it is time to download the program and run it.

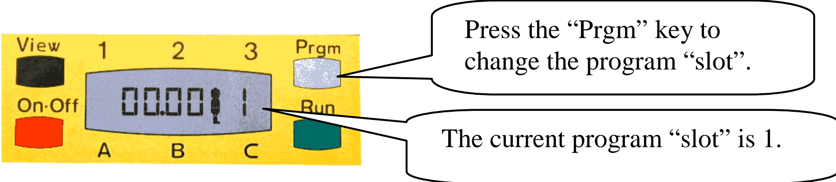
Does the program perform as expected? Does the robot make a 90 degree right turn? Probably not.

What can be done to make the turn closer to 90 degrees?

Did the robot turn right, or did it turn left by mistake? How can you correct this problem?

Program slots – pitfall to avoid!!

The RCX is capable of holding 5 programs at a time. Each program is downloaded to a program "Slot". Prior to downloading a program, always check the RCX to see which program slot is selected. The program will overwrite the existing program. On the RCX by push the "Prgm" button to change the "Slot".



If your kids are making an emergency change during a tournament to program 5, and your kids forget to change the slot from the prior download of program 1, guess what? Program 1 no longer does what it should – it now contains program 5. The kids will be very disappointed when they run the mission, and it doesn't work.

Best bet is to simply check the settings prior to each download. This will get the kids in the habit of verifying which slot has which program!

Topics learned in task 2:

Congratulations! You have learned more about the following topics:

- difficulty with turns based on motor run time
- demonstrate with people – must set power and direction before walking
- use metaphors to help kids see concepts
- copy, cut, and paste programming elements
- downloading to various program slots

Programming task 3: Subroutines (make a square)

Time: 30 minutes (20 minutes to create and run program + 10 minutes Q&A)

Requirements

This lesson builds upon the previous program. After moving forward for 5 seconds and turning right, lets **run the robot in a pattern to complete a square**. To perform this task, duplicate the code from the last lesson three more times!

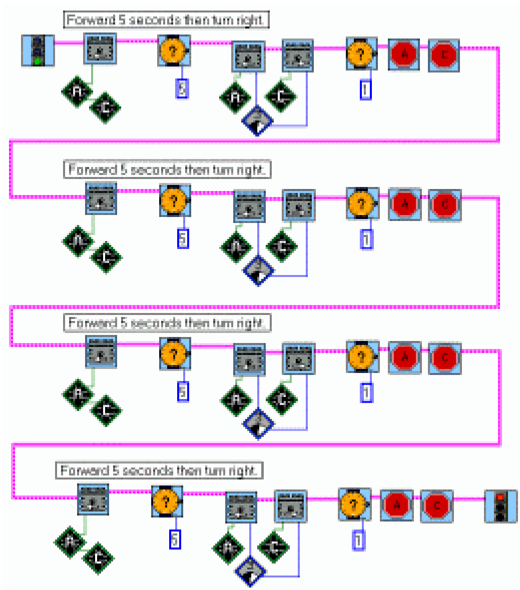
Command selection

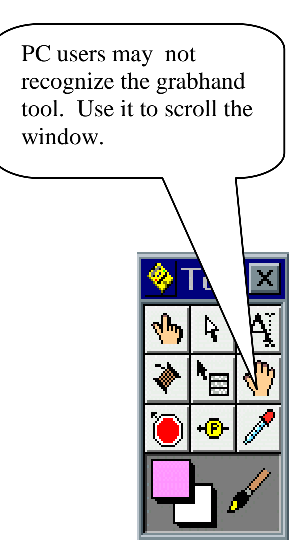
You should be able to complete the square pattern on your own. What is the easiest way to do this?

1. Copy/Paste
2. Loops
3. Subroutines

Copy/Paste

1. Open the "Forward_5sec_TurnRight" program from before.
2. Create a new file, "Square_copy_paste".
3. In the "Forward_5sec_TurnRight" diagram window, use the pointer tool to drag a selection box around all the functions except the "green light" and "red light."
4. Copy the selection. (<Ctrl-C> from the keyboard, or <Edit><Copy> from the menu.)
5. Now select the "Square_copy_paste" diagram window. (Just click in the window.)
6. Paste. (<Ctrl-V from the keyboard, or <Edit><Paste> from the menu.)
7. Drag the selection into position, then click somewhere else in the window to de-select the paste.
8. Paste, drag, deselect; three more times.
   Did you run out of space? No. The window has plenty of room, but you may not be able to see it all at once. Use the scrollbars, if you need to; or the grabhand tool.
9. Finish up, by wiring the clumps of pasted code together.





So what do you think of it?
A big mess! This is where so many teams get themselves into trouble – the programs get so long and complex so quickly, that no one can see the overall picture!

Problems with our current program

- Imagine hundreds commands filling up the programming screen as the robot travels around the mission board. How does one keep a sense of sanity with such complexity?
- What if you want to modify the right turn portion of each leg, for example add a beep after each right-turn. How many duplicate changes must you make?
- If you want to move a portion of code around, it is easy to grab the wrong chunk of code.

How to dramatically improve the program

We need to simplify, but how? It was easy to copy/paste, but it leads to a complex program. Programmers use several techniques to handle copy/paste situations. We'll consider two: loops and subroutines.

A simple rule of thumb: if you cut and paste a third copy of a set of program elements, go back and simplify it. Try converting it into a loop or a subroutine. We'll do a little of both.

Loops

Use loops to do the same thing a number of times. In this case, "Go forward and turn right" four times. While ROBOLAB has many specialized loops, for this case use the basic loop functions START OF LOOP and END OF LOOP . Add a NUMERIC CONSTANT modifier to the START OF LOOP to hardcode the number of repeats, "4". The default number of repeats is 2.

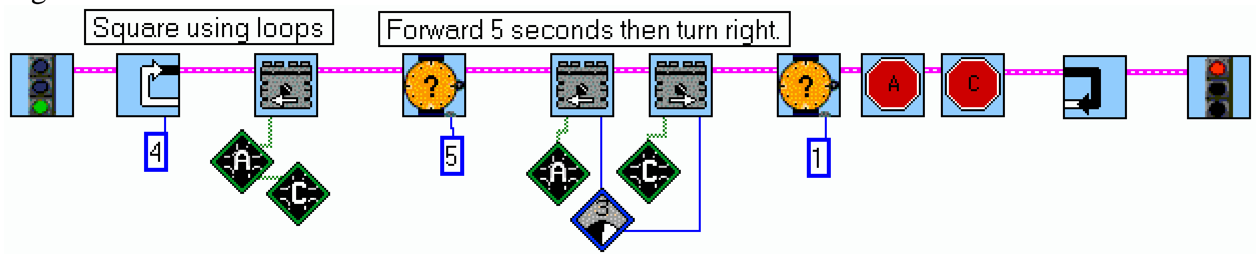
Between the START OF LOOP and END OF LOOP functions, place one copy of the copy/paste code.

Select and remove (delete) three copies of the copy/paste code that you'll no longer need.

Remove the bad wiring (<Ctrl-B> from the keyboard, <Edit><Remove Bad Wires> from the menu).

Rewire as necessary.
Constant "4" to modify the "Start loop"
End of "Green Light" to begin of "Start Loop"
End of "Start Loop" to begin of copy/paste code
End of copy/paste code to begin of "End Loop"
End of "End Loop" to begin of "Red Light"

Something like this:



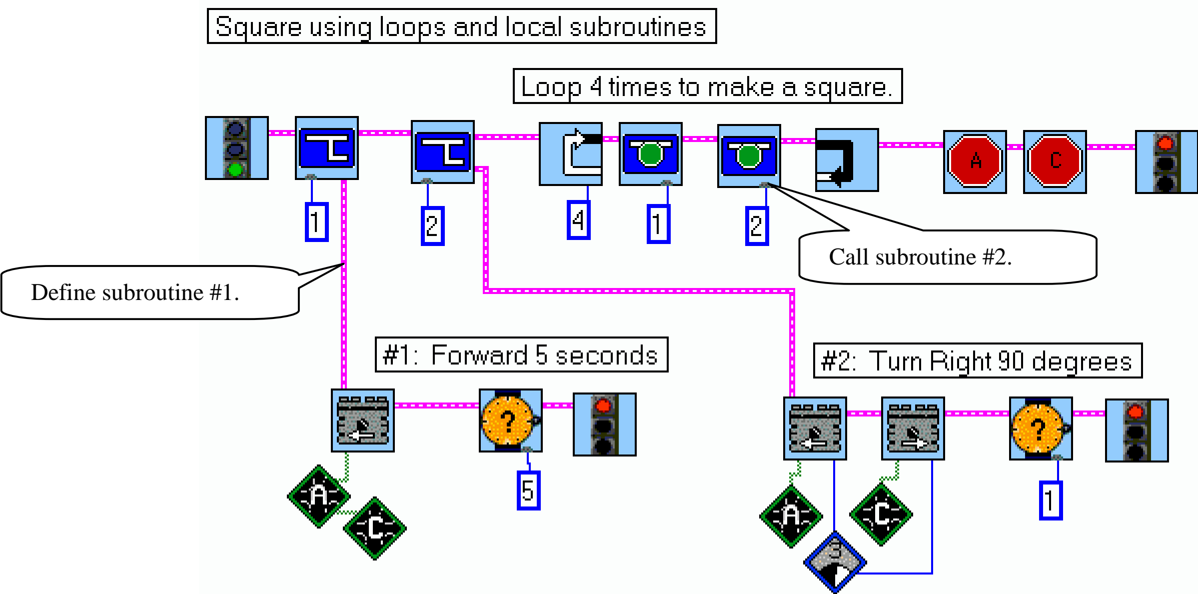
To locate the loop functions, FUNCTIONS → STRUCTURES → LOOPS → START LOOP, END LOOP.

So now, what do you think of the program? Pretty good, but lets go for extremely simple. After all, a square is a simple shape.

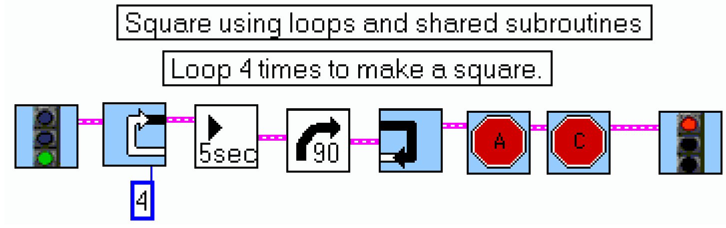
Consider the “Forward 5 seconds and Turn Right” section. See the “and”? How likely is it that you’ll need to “Turn Right” or “Forward 5 seconds” separately? Very likely. Let’s convert that section of code into two subroutines. Why two? Because robots will need to “Turn Right” and “Forward 5 seconds” often, but not always together.

Once the subroutines are written, the program will “call” the subroutine. Basically, the program will say “Turn Right” and let the subroutine do it.

Here is our “Square” program revised to use local subroutines:



And here, revised to use ROBOLAB’s version of a shared subroutine, a SubVI:

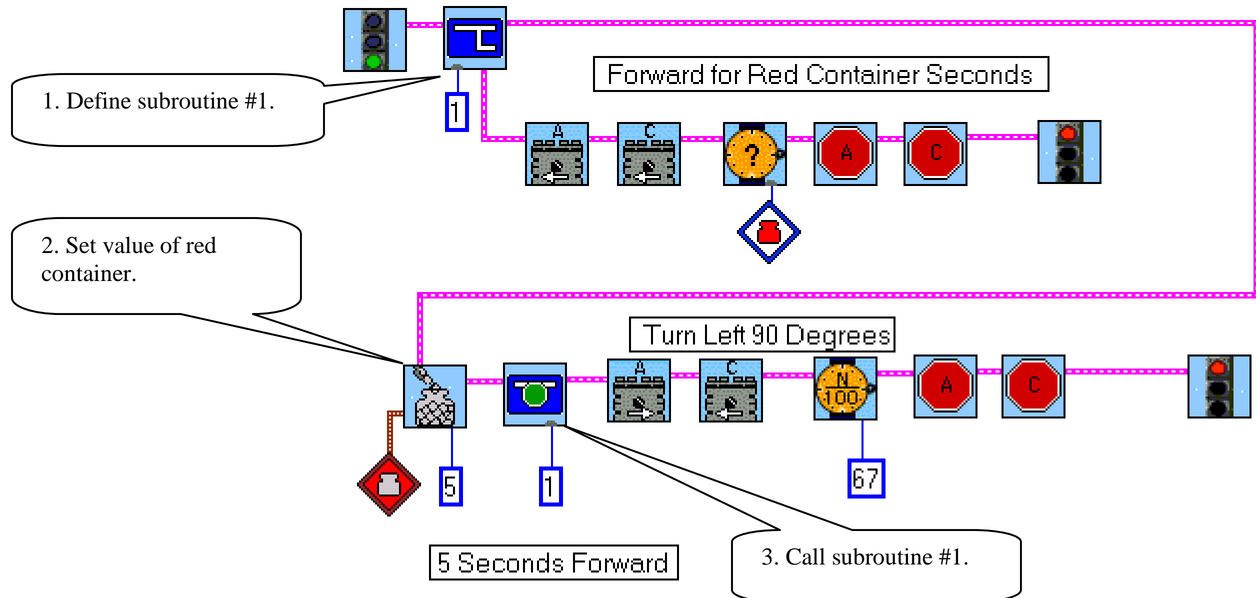


The SubVI version is much easier to understand. It almost doesn’t need comments to explain it. Subroutines simplify by hiding the details of sections of the program. Unfortunately, SubVIs are more difficult to create.

Local subroutines.

A local subroutine exists inside a program. It is saved with that program and can’t be used by other programs. In other programming languages this might be called a “private” subroutine. Local subroutines must be declared before they are called. Local subroutines are identified and called by number (1-8). In the example below, it makes sense to talk about the program’s subroutine #1.

Here is another example of a ROBOLAB local subroutine:



There are several rules that must be followed.

a. The local subroutine must be defined before it is used.
b. The subroutine definition must end with a "red" light (an END function).

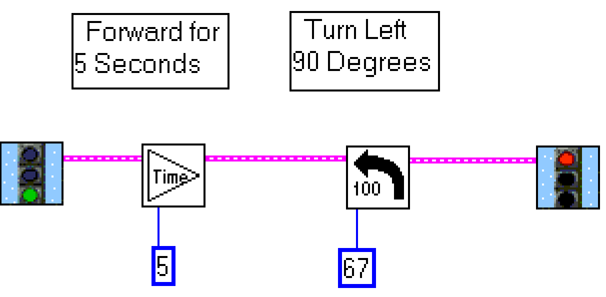
The example has one subroutine, which goes forward for "red container" seconds. Instead, create two subroutines: one subroutine to go forward for 5 seconds, and a second one to turn right. The idea is to build a subroutine for each robot function.

- When does it make sense to divide a task into two subroutines?
- Why are STOP A and STOP C functions in the subroutine? What happens if a motor is stopped and immediately started?

Shared subroutines or VIs.

In contrast to local subroutines, shared subroutines (a.k.a. global or public subroutines) can be used by other programs. In fact, they aren't even stored with the program. No program "owns" a shared subroutine. Instead, shared subroutines are stored in their own files. They often have the file extension ".vi" which stands for "Virtual Instrument."

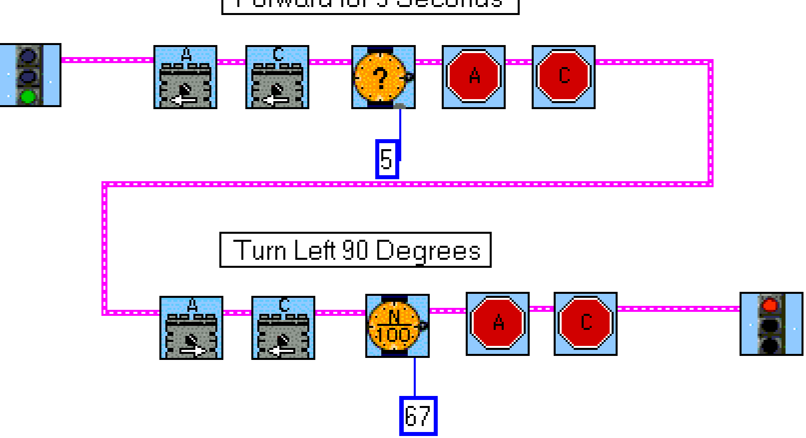
Here is an example:



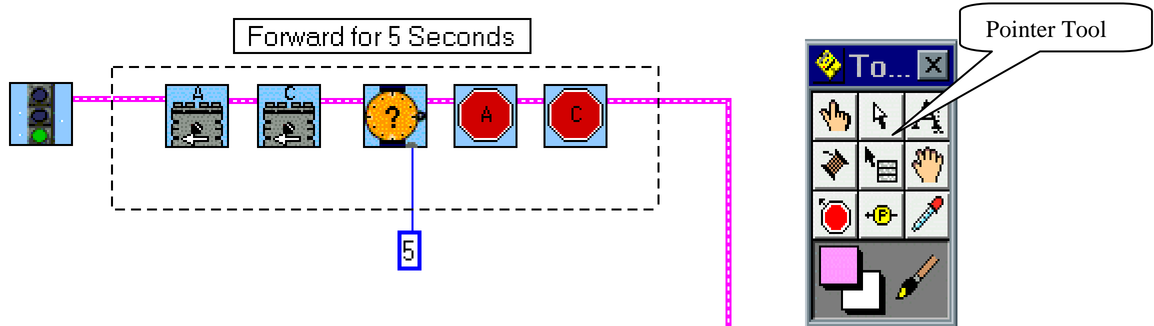
Welcome to the power of ROBOLAB.

A SubVI Walkthrough

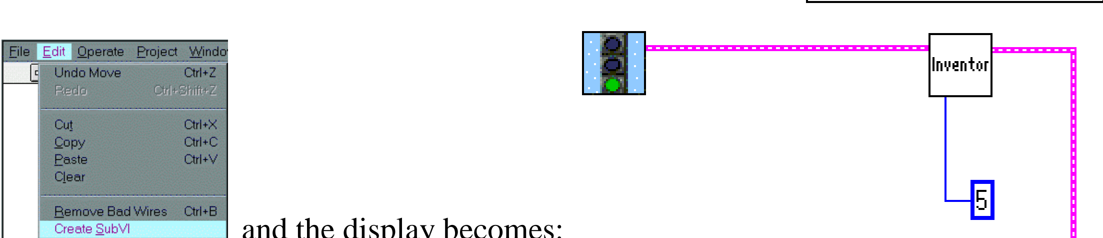
1. Start with a well-tested program. Don't even think about using buggy code.



2. Use the pointer tool to **select the code** that will become the VI (subroutine).



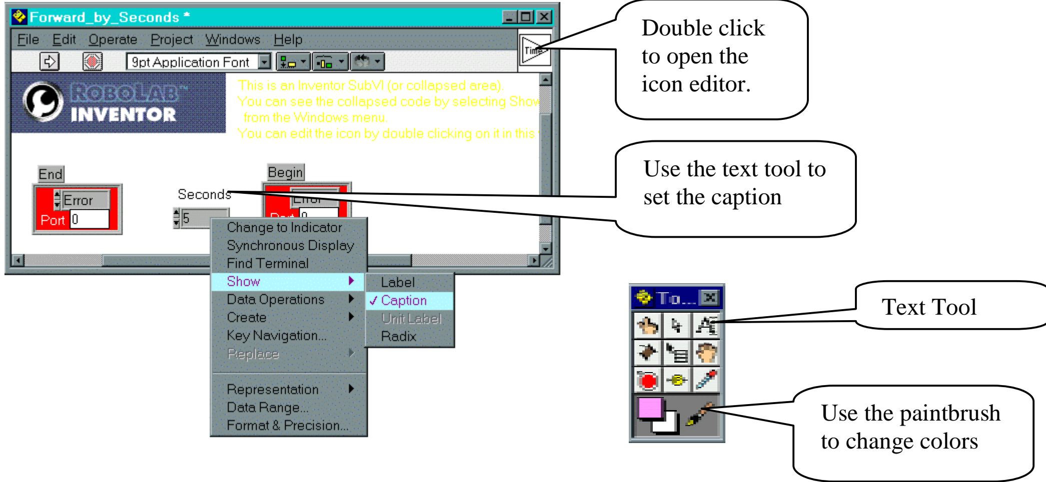
3. From the menu, select **Edit, Create SubVI**.



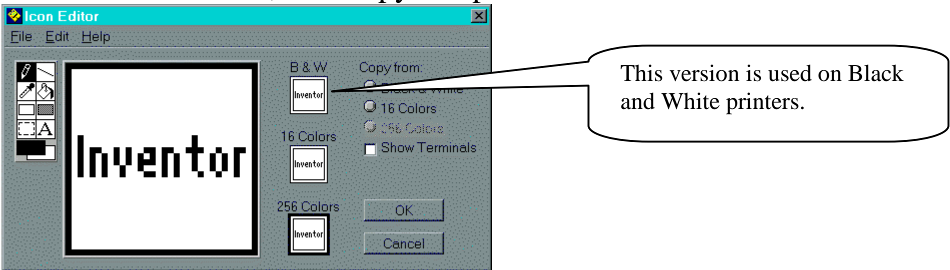
and the display becomes:
Note that the "Inventor" icon has three lines entering it.

4. Double click the "Inventor" icon and a new window will open. Drag a corner to expand the window. The three lines have become three terminals. The "End" and "Begin" terminals are ok. The "numeric" terminal needs to be edited to give users an idea what is expected. In this case, "numeric" is the number of seconds the robot will travel forward before stopping.

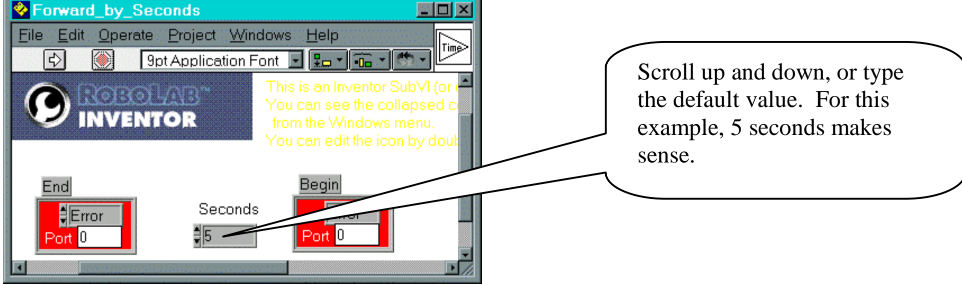
   To change "numeric" to "Seconds", use the text tool to select the "numeric" caption and enter "Seconds".



5. Open the icon editor by double clicking the icon in the top right corner. Start with 256 colors, then copy and paste into B & W.

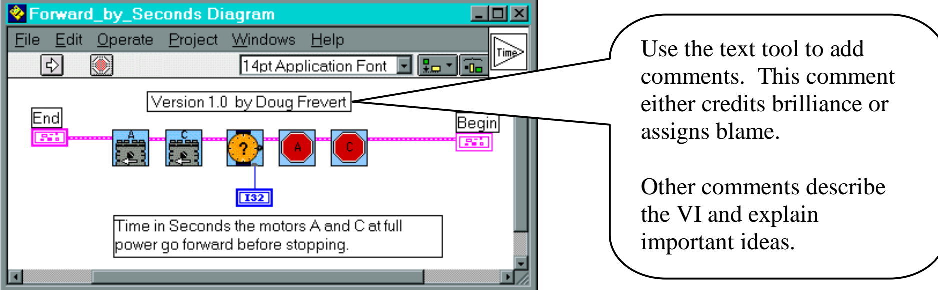


6. Set the default value of seconds.



7. Save the VI as a file. Use a descriptive name, here "Forward_by_Seconds" will do.

8. So where did the programming go? From the menu, select **Windows, Show Diagram** to see:



Naming Subroutines.

Naming is very important. If done properly, it will help your kids tell at a glance where they are and what the program is up to. Choosing poor names makes programming much more difficult.

Think of some of the basic motion functions that the FLL robot may need to do for the challenge:

1. move the robot forward until it hits a wall, then stop;
2. move forward and stop at a black line
3. move the robot forward a certain distance
4. turn right 90 degrees

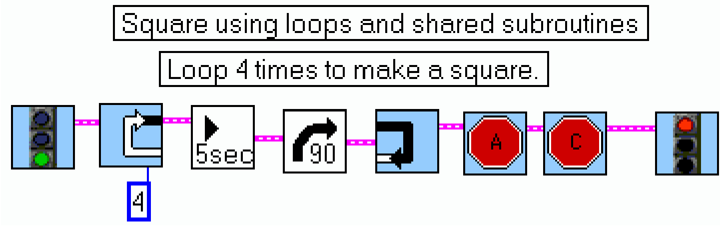
I suggest using "action + to + target", but this can be varied. To name the examples above, I might use the following names:

1. Fwd2Wall or Forward_To_Wall
2. Fwd2Line or Forward_To_Line
3. FwdDist or Forward_by_Inches
4. TurnRight or Turn_Right_90_degrees

As you can see, you have to be creative. But some naming consistency makes sense.

Some poor examples might include:

- "Fwd5sec" (when changing to 6 seconds, you don't want to have to change the name);
- "GoToTallTower" (too long, and probably can't re-use such a specific subroutine).
- "GoStraight" (no sense of direction – can't tell if the robot will move forward or backward).



Untitled

This is about as easy to read as the program can be. Can you see the defference between this program and the versions above that perform the same routine? See the simplicity of the code? It is very easy to read. The complexity of the code is hidden well, but can be viewed easily if desired by opening the subVI commands.

Saving your program

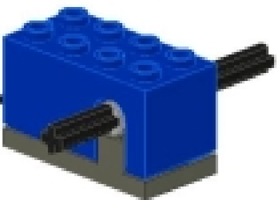
Go ahead and save your program using the File – Save menu option.

Topics learned in task 3:

Congratulations! You have learned more about the following topics:

- Using a loop.
- Why use subroutines:
  1. Most important reason: reduce a program's complexity
  2. Create subroutines to hide information so you wont have to think about the details
  3. Think about the subroutine when you are writing it, but use it later without knowing about its internal workings
  4. Avoiding duplicate code
  5. Promotes code re-use
  6. Isolates complex operations
- importance of names
- importance of hiding information
- difficulty with turn based on motor run time
- webs of wired functions are difficult to follow

Programming task 4: Rotation sensor


Rotation sensor

Time: 30 minutes (20 minutes to create and run program + 10 minutes Q&A)

Requirements

In this lesson, we will use the rotation sensor to measure distance traveled more accurately. This should result in a robot that gives a more repeatable performance run after run.

Structure of programming

You should now have a good idea of how to structure a program. We want to use well-named subroutines to create building blocks of code fragments that perform specific tasks.

As you have probably noticed, locomotion using motor run time results in limited accuracy. What we have to cover now is building some really useful routines. Turning motors on for a specific amount of time is of limited value in FLL. It will get the kids started but isn't going to help with most of the missions.

Rotation sensor introduction

Why use a rotation sensor? As you have probably noticed, positioning a robot based on motor run time leaves a lot to be desired. Any of the following can cause the robot to alter its path based on motor run time:

- Squeeze the Lego parts together. Friction between the gears increases.
- Battery power fades with every robot run.
- The surface friction changed by warming up, cooling down, etc;

These uncontrollable issues make time a poor way for the kids to navigate their robot.

Think of giving a friend driving directions to Chicago. Would you say "drive you car for 25,395 seconds, turn right for 10 seconds, then drive for 250 seconds to a restaurant"?

They would not get near the restaurant that you had in mind! It probably would land them in a different city, and it is likely that after the first turn, they turned off the road and onto a farm!

A more accurate instruction set would be "drive 314 miles, then exit at the water tower. Turn right, and go 2.4 miles to the restaurant". Your friend would most likely end up at the same place that you had planned on.

A rotation sensor can behave as an odometer for the Lego robot. It can measure the number of rotations that an axle turns. If the axle is somehow connected to a wheel, it will quite accurately measure distance traveled.

Manually measuring distance with the RCX

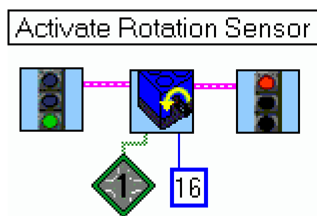
It makes a good demonstration to measure a distance with the robot using the rotation sensor. And, this method will be used often to measure distances. Why bother converting from inches to rotations and back many times? Simply use the robot to take the measurements, then your kids will know exactly how far it must travel!

Before we can use the robot to measure, the RCX computer needs to have the rotation sensor activated.

Activating a robot's rotation sensor

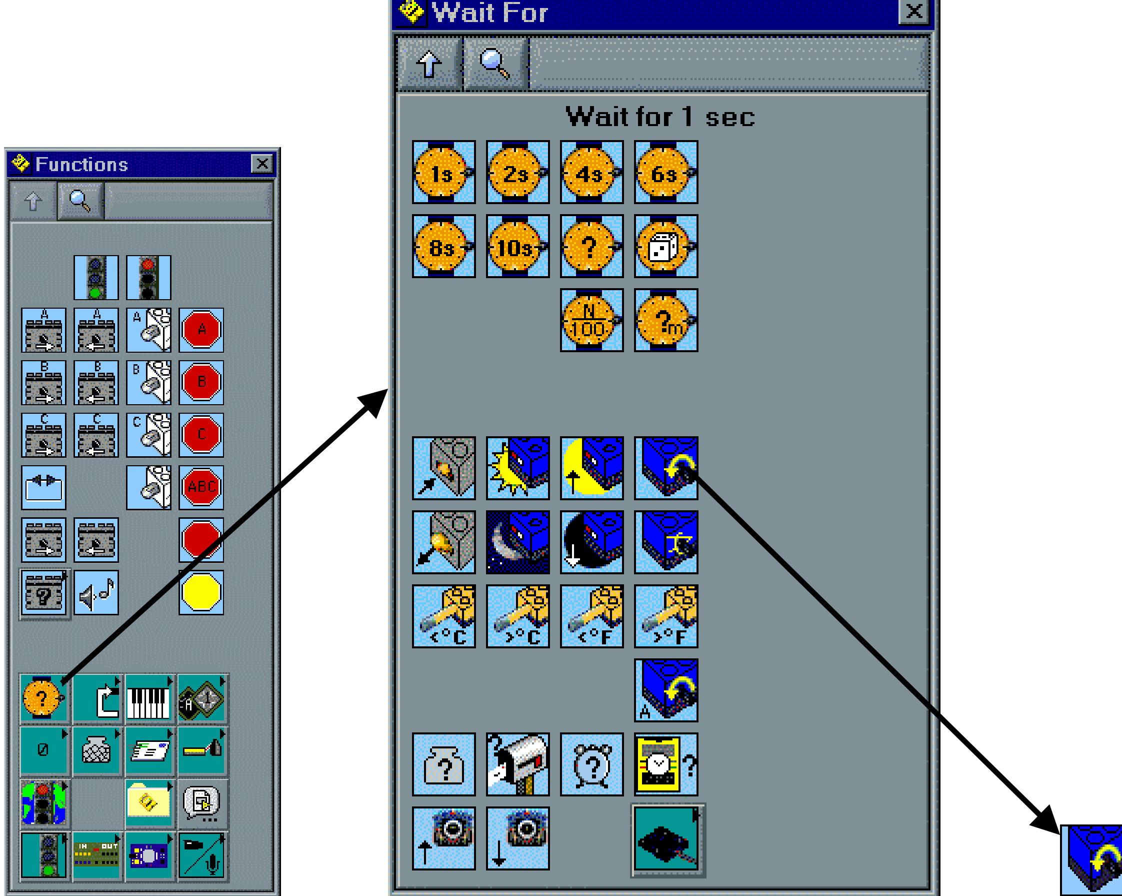
To activate the rotation sensor on the robot, we must build a program that uses the rotation sensor. This can be a very simple program, such as one command to wait for the rotation sensor to reach a certain value.

To add a rotation sensor, first drag a WAIT FOR ROTATION command into a new program:



The WAIT FOR ROTATION defaults to a rotation sensor attached to the RCX port 1. Setting the port to "1" is not required, since it is the default value. Likewise, connecting the NUMERICAL CONSTANT modifier "16" is not required, because 16 is the default. The rotation counter counts 16 for each turn of the axle.

To locate the function's icon, FUNCTIONS → WAITFOR → WAIT FOR ROTATION.

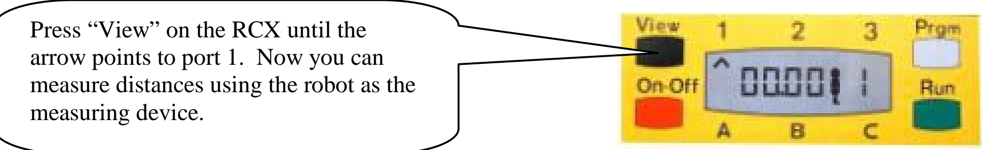


Now download the "Activate_Rotation_Sensor" program to the RCX and we can measure distances with it!

Measuring by moving the robot

Any sensor can have its current value displayed on the RCX window. After downloading a program containing a rotation sensor command, you are ready to begin measuring.

Press the view button until the arrow is pointing to the sensor port that you want to monitor. For your robot, press "view" until the arrow pointing to port 1.



Give it a try! Press the RUN button on the robot to execute the program and activate the rotation sensor. As you move the robot around, does the RCX window display the rotation counter value?

Programming with the rotation sensor

Now lets turn our attention back to programming. Lets say that we hand-measured a distance of 50 that we would like the robot to move.

How can we program this using ROBOLAB?

First, think about how to walk 50 feet. Here are the first few commands to walk 50 feet:

1. Set the zero mark at where you currently are standing.
2. Point your legs in a certain direction.
3. Set the power that you legs will use.
4. Walk forward.

Now the tricky part: what is the next command that you would give your legs?

If you think about it, the next command is "WAIT". "Wait" seems like it would indicate stopping, but it does not. "Wait" means to continue doing whatever you are doing, until an event occurs.

If your goal is to walk 50 feet, the event would be your eyes sensing that you have reached 50 feet. At that moment, you would issue a command to your legs to stop walking.

Here is a series of commands that might complete the above sequence for the Lego robot:
1. Reset the rotation sensor to zero
2. Set motor direction
3. Set motor power

4. Turn on both motors
5. WAIT until the rotation sensor = 50
6. Turn off the motors

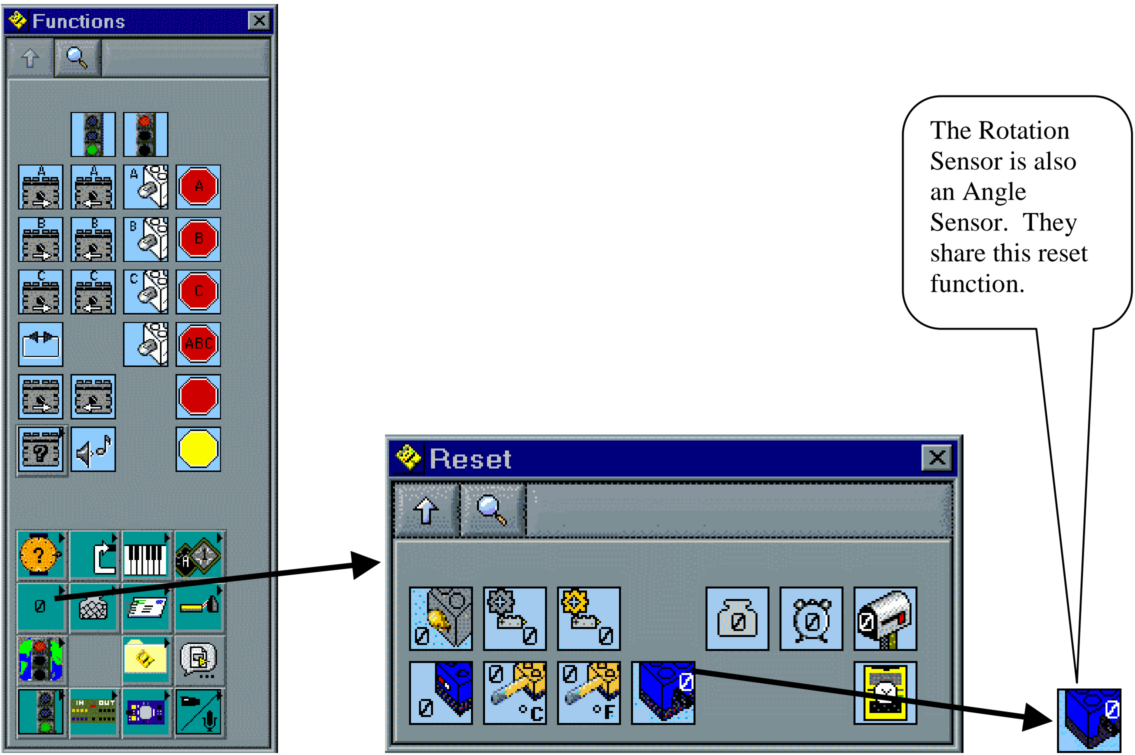
Resetting the rotation sensor

When should the rotation sensor be reset?

Again, think of driving directions to a distant place. Distances are usually given as measurements between fixed objects.

The same idea works for the rotation sensor. It is highly recommended that the rotation sensor be reset to zero whenever the robot is in a known location on the board!

Use the ZERO ANGLE SENSOR function to accomplish the reset.
To locate the function: FUNCTIONS → RESET → ZERO ANGLE SENSOR



Build a Forward_50 program

Try putting the concepts discussed above into a program that moves the robot 50 rotation counts.

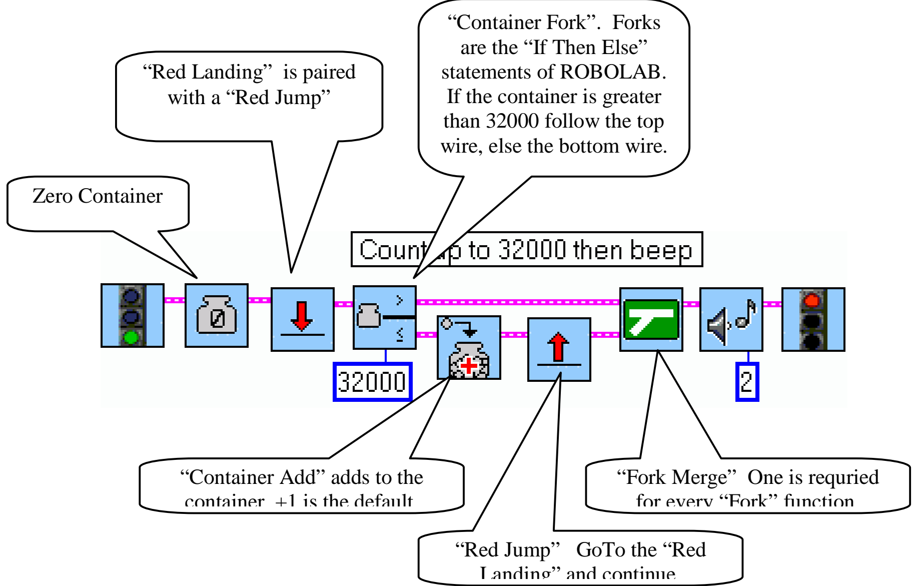
Using variables to expand the robot's capabilities

Variables are probably one of the most useful tools your kids can use to make useful functions.

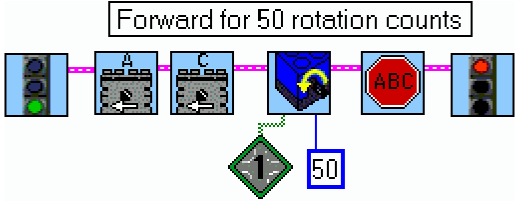
Now that you have a Forward_50 program, your robot can travel 50 rotation counts quite accurately. What if the mission requires that the robot travel 58 rotation counts? Or 125 counts?

This is where variables are useful. Set a variable (x) to the number of counts. Rewrite the program to wait for (x) rotations.

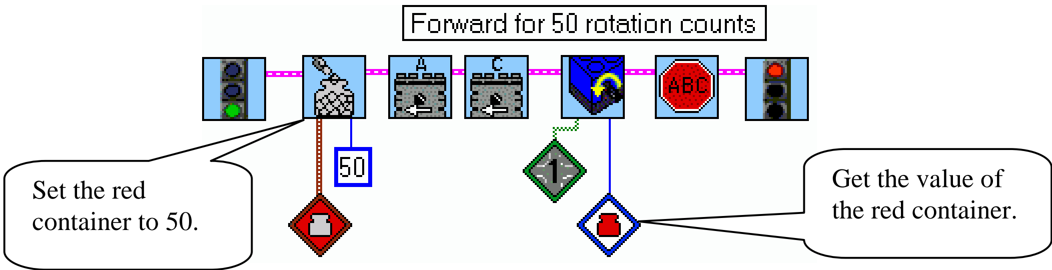
Every programming language uses variables, but using variables in *ROBOLAB* is unusual. First, variables are known as “containers.” Containers can hold only hold 16 bit signed integers. (That is, the numbers 0, 1, 2, 3, …, 32767 and -1, -2, -3, …, -32768. Here is a silly example that shows the use of the default container (red). The red container is zeroed, then incremented until 32001. After a couple of minutes, it beeps.



So, back to the “Forward_50” program:



Rewritten to use a red container instead of a hardcoded “50”, the program might look like this:

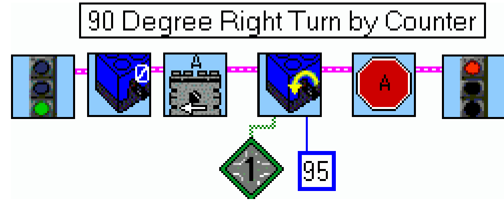


Not exactly worth the trouble, but it does show how to set and use a variable (container).
Note: The modifiers to set a container value and get a container value are different. Also note, we have to set the value before getting it (before the WAIT FOR ROTATION function is executed).

Turning using the rotation sensor

The rotation sensor can also be used to make more accurate turns. Try making a "RightTurnUsingRotation" program that uses the rotation sensor instead of the timing commands to make a more accurate turn. If the rotation sensor is geared to the robot's left-side tire, should the right-side tire turn? Can the right-side tire be locked, so it doesn't turn? If only the left-side tire turns, does the robot turn as sharply?

A "RightTurnUsingRotation" program might look like this:



Caveat of using rotation sensor

Suppose your kids want their robot to travel until the rotation sensor reaches 100, but the robot hits a wall at 90. What happens now?

At first, you might think that the robot would pause at the wall for a few seconds and then proceed with its program.

But think about the program – the WAIT FOR ROTATION function is waiting until the rotation counter goes past 100. Since this will never occur, the robot will wait FOREVER at the wall!!

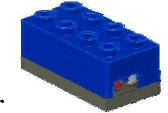
How can your kids program so that the robot will not wait forever at an object if it reaches the object earlier than expected?

Topics learned in task 4:

Congratulations! You have learned more about the following topics:

- using a rotation sensor for measuring mileage during straight runs and turns
- add a rotation sensor to the robot
- Introduce containers (variables).
- When and why to use containers, how to set them, how to change and read them
- how to make use of rotation sensor
- measuring mileage with the rotation sensor using the view mode
- how to enable rotation sensor to measure using view (must load program with rotation sensor)
- when to reset rotation sensor (every known fixed location – reset)
- comparison of accuracy using rotation sensor vs motor run time

Programming task 5: Forward to a black line, stop, turn right

Light Sensor 

Requirements

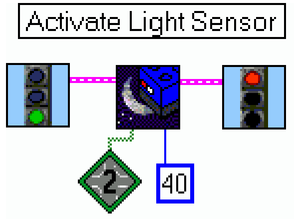
In this lesson we will program the robot to move forward until it reaches a black line. The robot should then stop, and turn right.

Light sensor

The Mindstorms light sensor is programmed like the rotation sensor. You can view the value of the light sensor through the RCX window exactly as you did with the rotation sensor.

The light sensor interprets the amount of light from a reading of zero (black) through 100 (white).

Try downloading any light sensor command to the RCX, and VIEWing the light sensor reading. Run a program containing a light sensor to activate this feature. Like this:



Light sensor programming

To use the light sensor in a program, we will follow the same technique that was used with the rotation sensor.

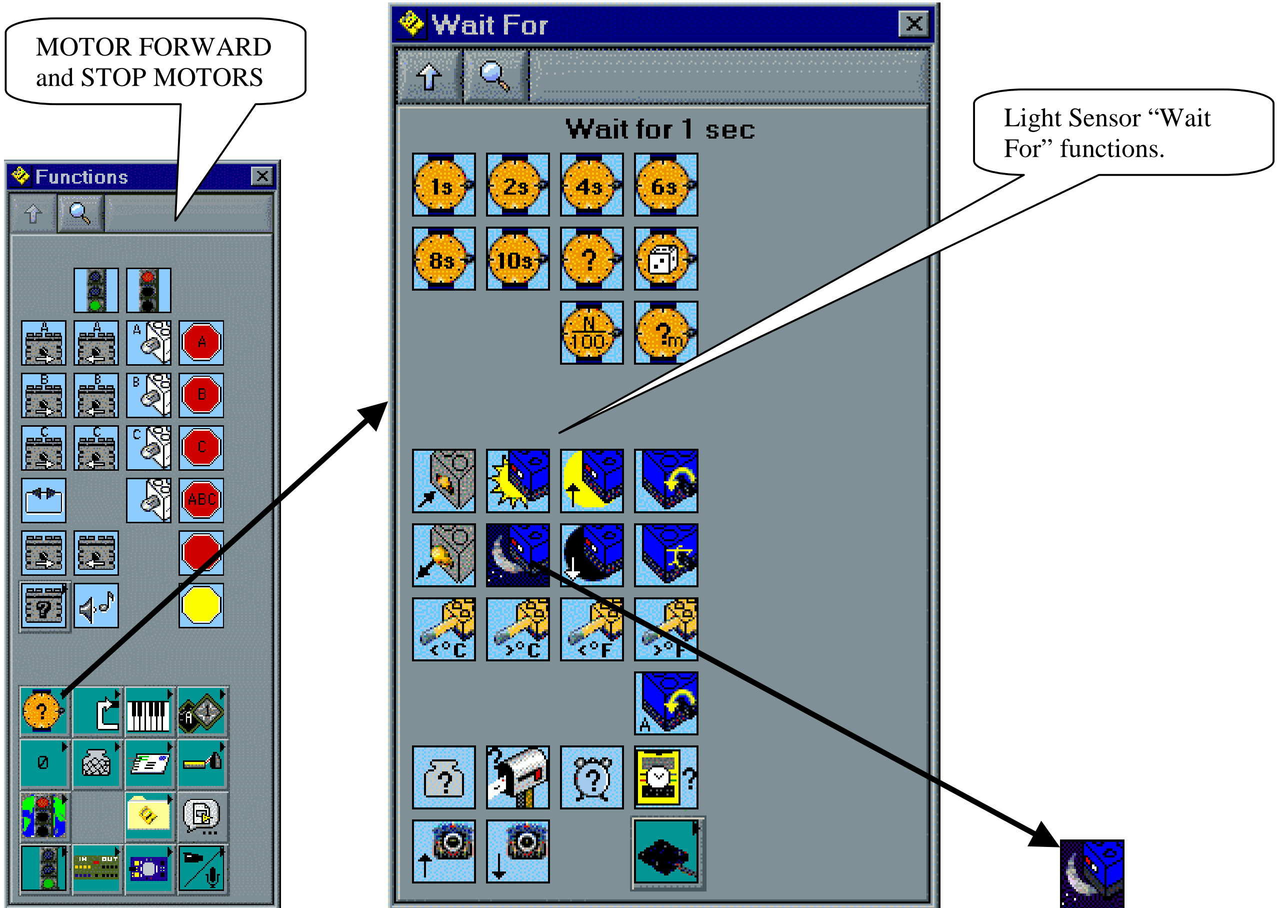
In English: Move up to the black line and turn right.
In pseudocode:

1. set motor direction
2. set motor power
3. turn on both motors
4. WAIT until the light sensor reading is dark
5. turn off the motors
6. turn right

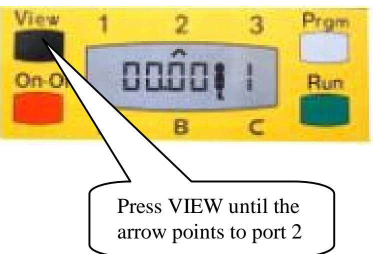
Write a new program named " Fwd2Line"

See if you can do this yourself from the above pseudocode.

First, collect the functions required: The MOTOR FORWARD, POWER LEVEL 3, OUTPUT A, OUTPUT C, WAIT FOR DARK, STOP MOTORS ABC.



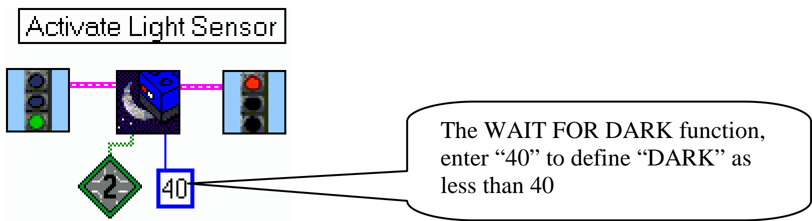
The trick here is determining what the RCX will read as "Dark". To measure with the RCX, remember that you must load and run a program on the RCX that uses the light sensor.



Perform the following steps to measure the light sensor readings:

- download the Fwd2Line command to the RCX,
- run the program by pressing the green RUN button,
- stop the program by pressing RUN again
- press the VIEW button on the RCX until the arrow points to port 2.
- to determine what a LIGHT reading is, place the light sensor over a WHITE area
- to determine what a DARK reading is, place the light sensor over a BLACK area
- write down the light and dark readings
- calculate the approximate average of the light and dark readings (WHITE + BLACK) / 2
- enter this average in the WAIT UNTIL command to differentiate light from dark.

For example, the white reading is 50, and the dark reading is 30. The average of 40 would be coded in the WAIT UNTIL command as shown:



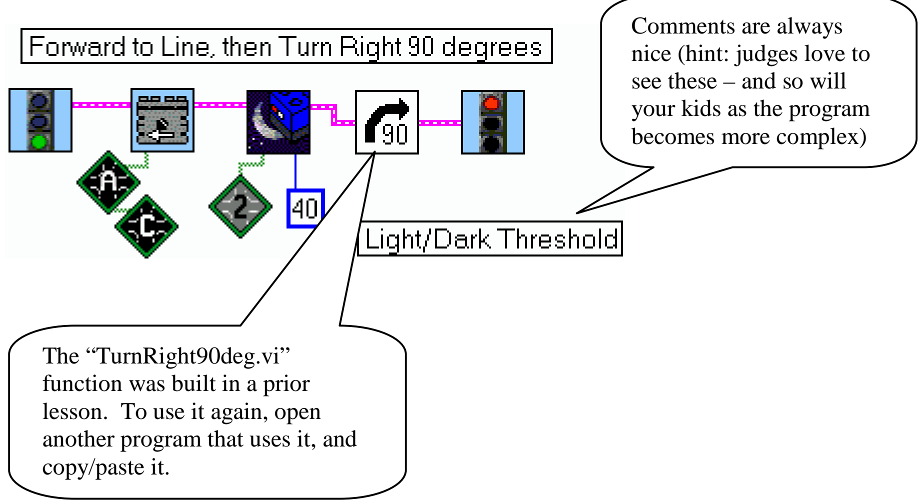
So you've reached the black line, now what? Turn Right, remember. Didn't we already do that? Yes.

Calling a subroutine

If you created a "TurnRight" SubVI as part of Lab #3, you should reuse that code. Open the your final version of the Lab #3 program. (From the menu, <file><open> then pick the file from the list.) Use the pointer tool to select the "RightTurn" icon. Copy it. Locate and select the "Fwd2Line" program window. Paste it. The "RightTurn" icon will be plopped somewhere in the window.

If you used a local subroutine, instead of a subVI, more work is required. Again, open the final version of the Lab #3 program. Use the pointer tool to select all of the "RightTurn" subroutine. It is best to drag a box around the entire subroutine to select it. Usually, the icons must be moved around so that they can be neatly selected. It may be easier to copy everything and delete what isn't needed after the paste. In any case, after selecting, copy it. Switch to the new program's window. Paste it. A second copy and paste will be needed for the icons that call the local subroutine.

Then again, recoding a right turn is pretty easy. Maybe easy is better. It depends.



Changing light conditions

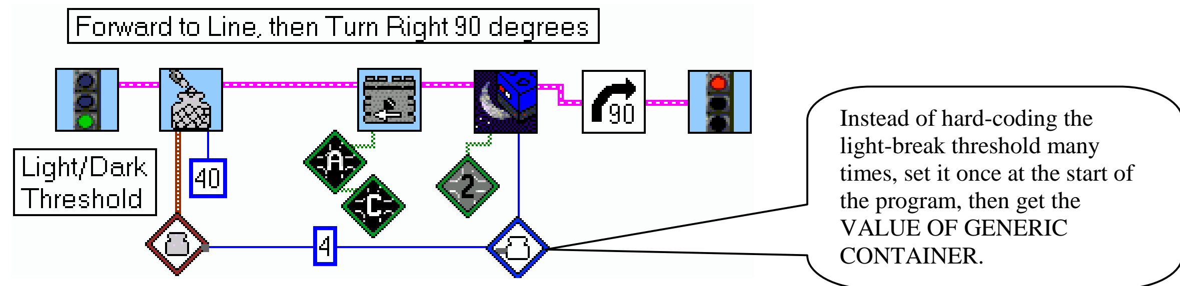
Lets say that your kids found that 40 is the value that they want the light sensor to read as “Dark”. If their program has several different light sensor commands in a program, they could “hard-code” 40 as the “dark” reading. Hard coding means to enter a specific value into a command.

If the kids take their robot to the competition and the ambient lights are much brighter, it could cause the light sensors to not see “dark” when they should, because the black line never reads below 62, for example.

Can you think of a method to address this problem? Can you think of a second method? Remember, if you only think of one solution, you might end up using the worst possible solution.

One solution is to use the WAIT FOR DARKER function instead of the WAIT FOR DARK function. The problem with the WAIT FOR DARKER light sensor function is that it seems to be a bit unpredictable. If you start the program on a dark area, it behaves differently than if you start it on a light area.

A second method is to use a container to store the light sensor threshold. If you want to sense “dark” as below 60, you can use the following command:



A third method is to always start with the light sensor on WHITE. Take an initial reading and set the container to WHITE + *n*, where *n* is a good guess of the difference between WHITE and BLACK is various lighting conditions.

A fourth method watches values of the light sensor and waits for an out-of-range value. If the out-of-range is darker, assume it is BLACK the other values must be WHITE. Of course this requires the hardcoding of the idea “out-of-range”.

A fifth method saves the “out-of-range” threshold to a container.

And so on.

Which is best? It depends. By the way, there are more possibilities. Where else could you store the threshold containers? Is it possible to store thresholds without using containers?

What if someone uses a flash to take a picture of a running robot?

Additional notes

Which commands are used most often?

Motor n Forward. (n = A, B, or C; Forward or Backward). Some teams spend a lot of time with power settings, others just run the motors at the default (full power).

Forks: Touch Sensor Pushed, Light Sensor Darker/Lighter, Rotation Sensor Greater Than.

Wait Fors: Wait for Dark, Wait for Light, Wait for Pushed In, Wait for Rotation

Resets: Timer Reset, Rotation Sensor Reset.

Loops: Loop Start, Loop End

Jumps:

Containers: Fill Container, Add to Container, Reset Container, Wait For, Fork, and Loop.

Which commands should be avoided?

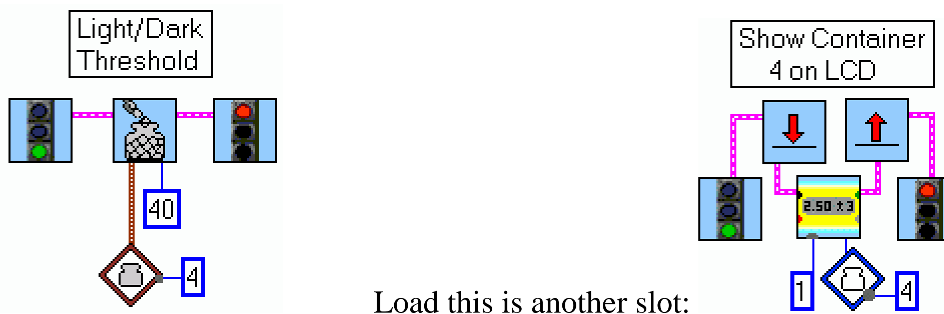
Task Split: The command works very well as the last set of commands, but as an intermediate step it can be difficult. For example: a line following program can be run as two tasks: 1) if too white, power motor A. 2) if too dart, power motor B. Great, it works. Now add it to code that follows a line until a rotation sensor is greater than 500. How do you put the two together? If a “rotation sensor” task is added, it will serve to detect the rotation, but how do you stop the tasks running the motors? “Stop Tasks” does just that, including the task that was watching the rotation sensor. Essentially, the program stops.

There has to be a way to program around this, but until shown a working counter-example: task splits should be avoided.

Why do sensors and containers need to be reset to zero?

Just like the firmware, the values in containers don’t disappear unless the battery power fails. There are containers for most sensors. While this is a problem that forces programmers to reset container values when programs first start, it can be useful.

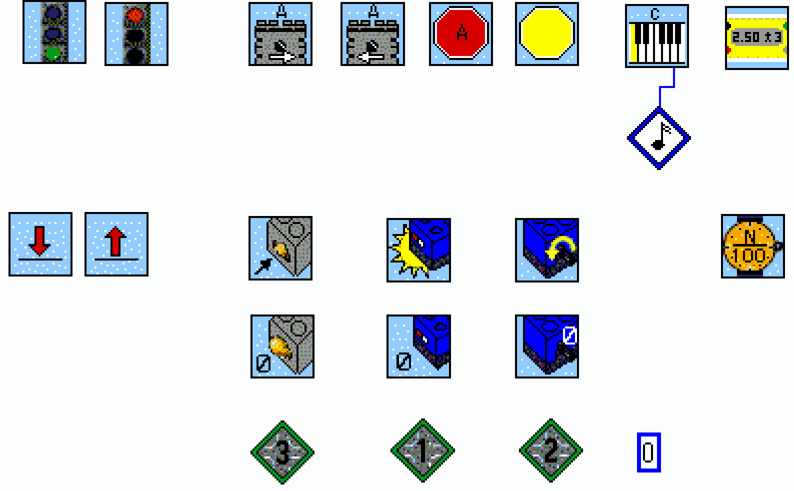
Consider this program: Load this is another slot:



A value saved into a container can be used by another program.

Limiting the commands available in ROBOLAB.

You may find that a limiting the supply of ROBOLAB command icons helps your programmers. Write a program so that programmers can copy command icons and paste them into their programs. This program will not run, but that doesn't mean it isn't useful. Here is an example:



Waiting for several sensors at the same time.

Consider the following requirements:

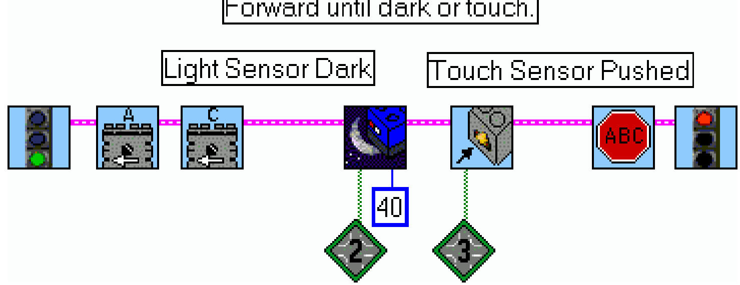
1. Move forward until the robot reaches a black line **or** hits a wall (detected by a touch sensor);

2. Move the robot's arm up while traveling forward;

3. Drive forward until the left bumper **and** the right bumper hits a wall.

Without giving away all of the secrets, the tasks described above can be accomplished using ROBOLAB. The trick is that more than one sensor must be monitored simultaneously.

For example, lets explore #1 above. Your kids build a program as shown:



Notice that the robot will turn on both motors to drive forward, and then WAITs until an event is detected by the light sensor on port 2.

But the requirement states that when the robot finds a black line OR if a wall is bumped, the robot must stop moving.

Will the robot stop at a black line? Yes.

Will the robot stop at a wall if it never saw a black line? NO! (Actually the robot WILL stop at a wall, of course, because it can't drive though a 2x4; but with this program, the robot will be stuck at the wall, forever spinning the wheels forward looking for that black line!)

How can the kids create a program to stop at a black line OR at a wall?

The trick is to have the kids build their own WAIT command. Think of what a WAIT command does.

- Start
- Test a sensor
- Did the sensor detect a change?
- If not, go back to "Start"
- Do something, because the sensor changed.
- Stop

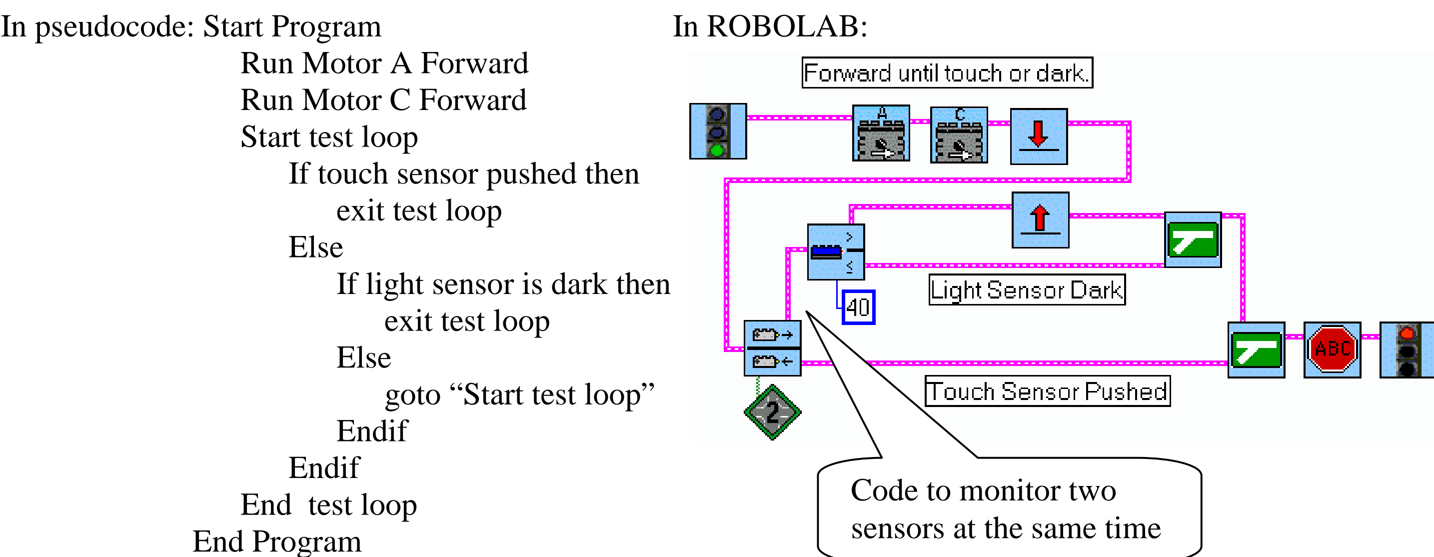
Wait for light sensor event OR touch sensor event

In English: Go forward, until either the touch sensor is pushed or the light sensor sees dark.

In pseudocode:

```
Start Program
     Run Motor A Forward
     Run Motor C Forward
     Start test loop
          If touch sensor pushed then
               exit test loop
          Else
               If light sensor is dark then
                    exit test loop
               Else
                    goto "Start test loop"
               Endif
          Endif
     End  test loop
End Program
```

In ROBOLAB:



Notice that one If, Then, Else fork is nested inside another. There is no limit to the number that can be nested together. Just remember, for every fork there must be a fork merge.

This code is meant to give a hint at some of the more advanced programming concepts available with ROBOLAB.

Many FLL programming problems can be solved by constructing similar programs. If your kids can understand and make use of this type of code construction, they will be ahead of the curve!

Debugging a program

Step through the program

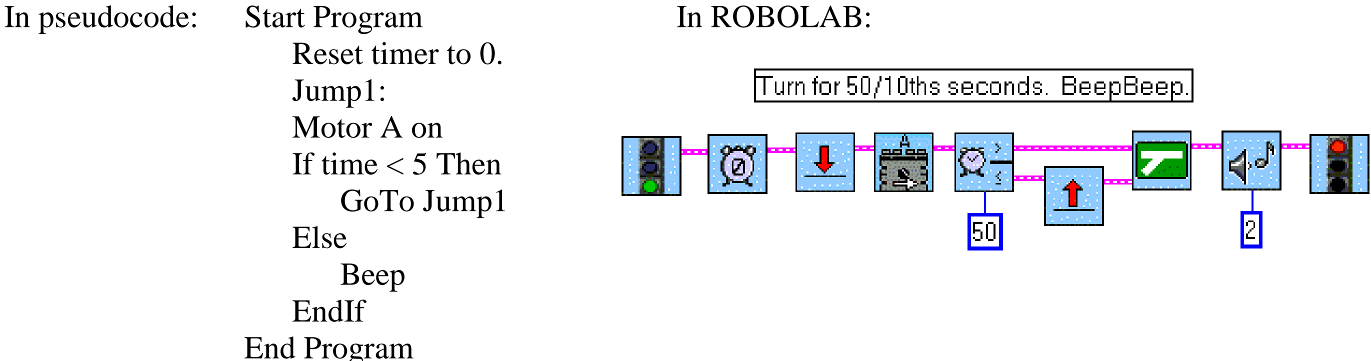
One of the simplest methods of debugging is visually stepping through a program. Here is a snippet of code with a bug.

In English: Run Motor A for 5 seconds then beep.

In pseudocode:

```
Start Program
    Reset timer to 0.
    Jump1:
    Motor A on
    If time < 5 Then
        GoTo Jump1
    Else
        Beep
    EndIf
End Program
```

In ROBOLAB:



To visually step through the program, start at the first ROBOLAB function (Start Program) and follow the wires to the next function. Pretend you are the computer.

Do you see the problem? Maybe there isn't one. When you say, "Run for 5 seconds." do you mean "Run for 5 seconds, then stop."? If so, the bug is that Motor A doesn't stop.

Do you see a second problem? Does Motor A need to be turned on more than once?

Peer Review

"Two minds are better than one." What is it about programming that keeps programmers from seeing the obvious? Imagine building a three-legged stool with only two legs. A test clearly reveals a bug. Now imagine that you see the third leg even though it doesn't exist. Once in a while, every programmer writes code with a missing third leg. As hard as they try, they see a leg that isn't there. In fact, the harder they look, the more clearly they see the mirage. Programmers need friends who understand this. The friend will see what's missing and understand the mirage. A true friend will not tease.

Programmers call this "Peer Review."

Add Debugging Code

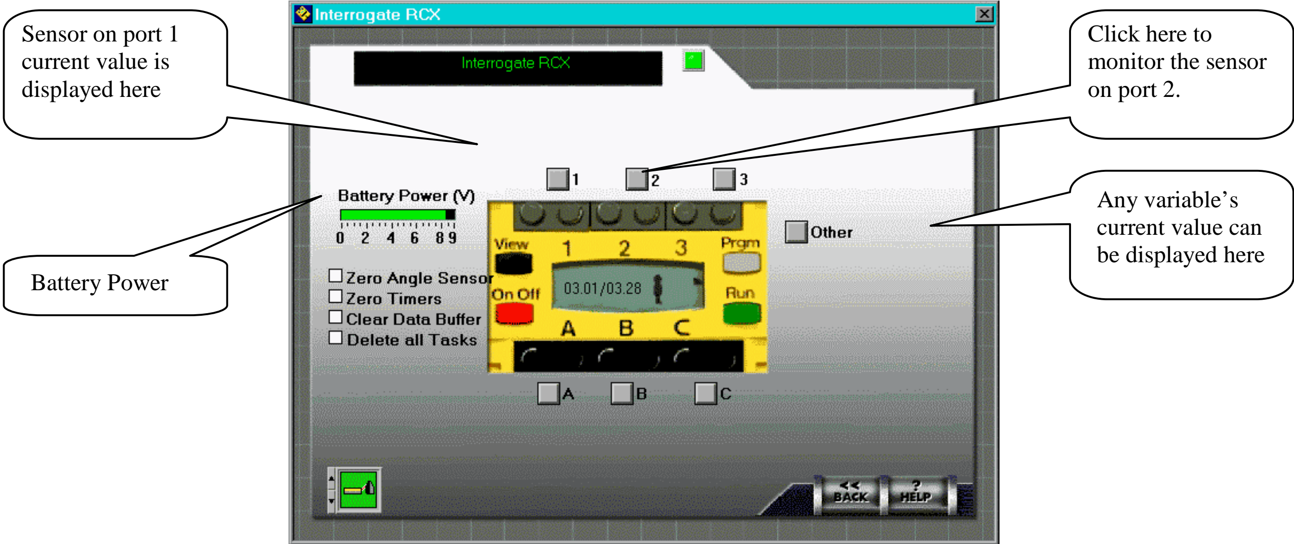
Add a beep, a wait for timer, or set the LCD display at strategic points in the program. It is often surprising to see which section of code is running. Remember to remove the debugging code after the bug is found.

Interrogate the RCX

When this doesn't reveal any problem areas, ROBOLAB includes an "Interrogate RCX" menu option that can be useful as a debugging tool. With the "Interrogate RCX" screen, kids can monitor current variable values and current sensor values as they run their program.

This debug screen is quite sophisticated – it reads the real-time sensor and variable values from the RCX through the IR tower and displays the values on the PC screen.

To fire up the debug screen, select "Interrogate RCX" from the "Project" menu.



Last Resorts

If your kids still can't figure out why the robot is behaving badly, you can try as a last resort the following:

- Divide and Conquer.
  Split the code into two halves. Run each half in its own program. If only one half fails, repeat the process with the failing half. Eventually, you'll have a small enough section of code that you'll see the bug.

- Add timer delays.
  Add "WAIT FOR 0.1 seconds" prior to "sensor wait" or "motor on" commands. This seems to allow time for the RCX to stabilize. [The firmware pauses about 500 times a second to read all the sensors (inputs) and RCX buttons, reset the LCD panel, and switch the direction and power of the motors (outputs). The firmware then goes back to decoding the bytecodes of your program. This fix guarantees that changes to settings have been seen by the firmware.]

- Reload the firmware.

Backing up your kid's program code

Be sure to save your team's programs after each meeting, preferably to diskette or on a different computer. The files holding the team's programs are fairly small. ROBOLAB allows the administrator to set the location for storing programs. If not set, ROBOLAB defaults to:

PC   C:\Program Files\ROBOLAB\Program Vault\Inventor\My Programs\Level4\
MAC   HDriveName:ROBOLAB:Program Vault:Inventor:My Programs:Level4:

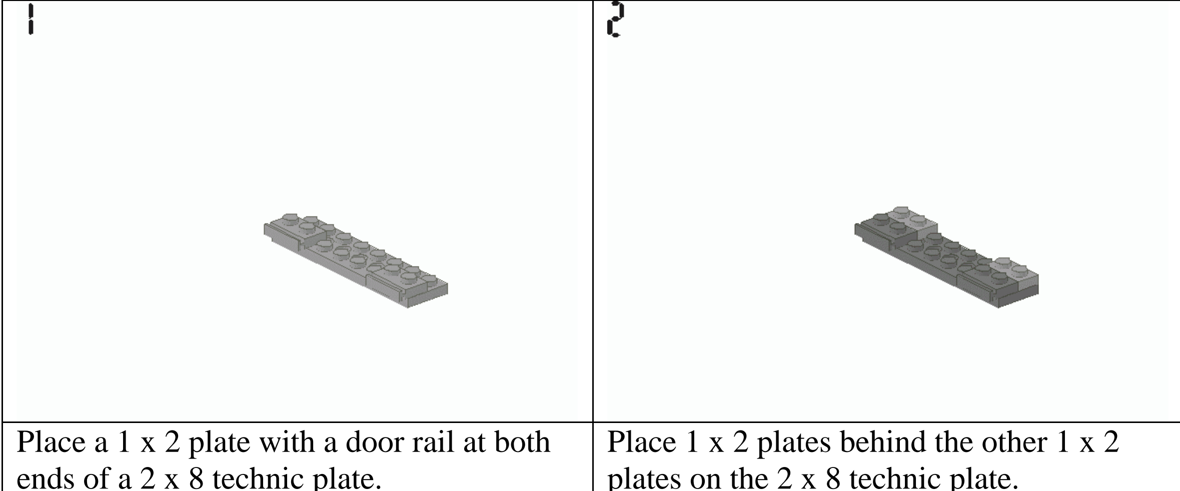
Copy the entire Level4 directory (folder) to diskette. Keep three backup diskettes and cycle them, overwriting the oldest backup. This is overkill, but an excellent habit to learn. Use it.

Robot Construction

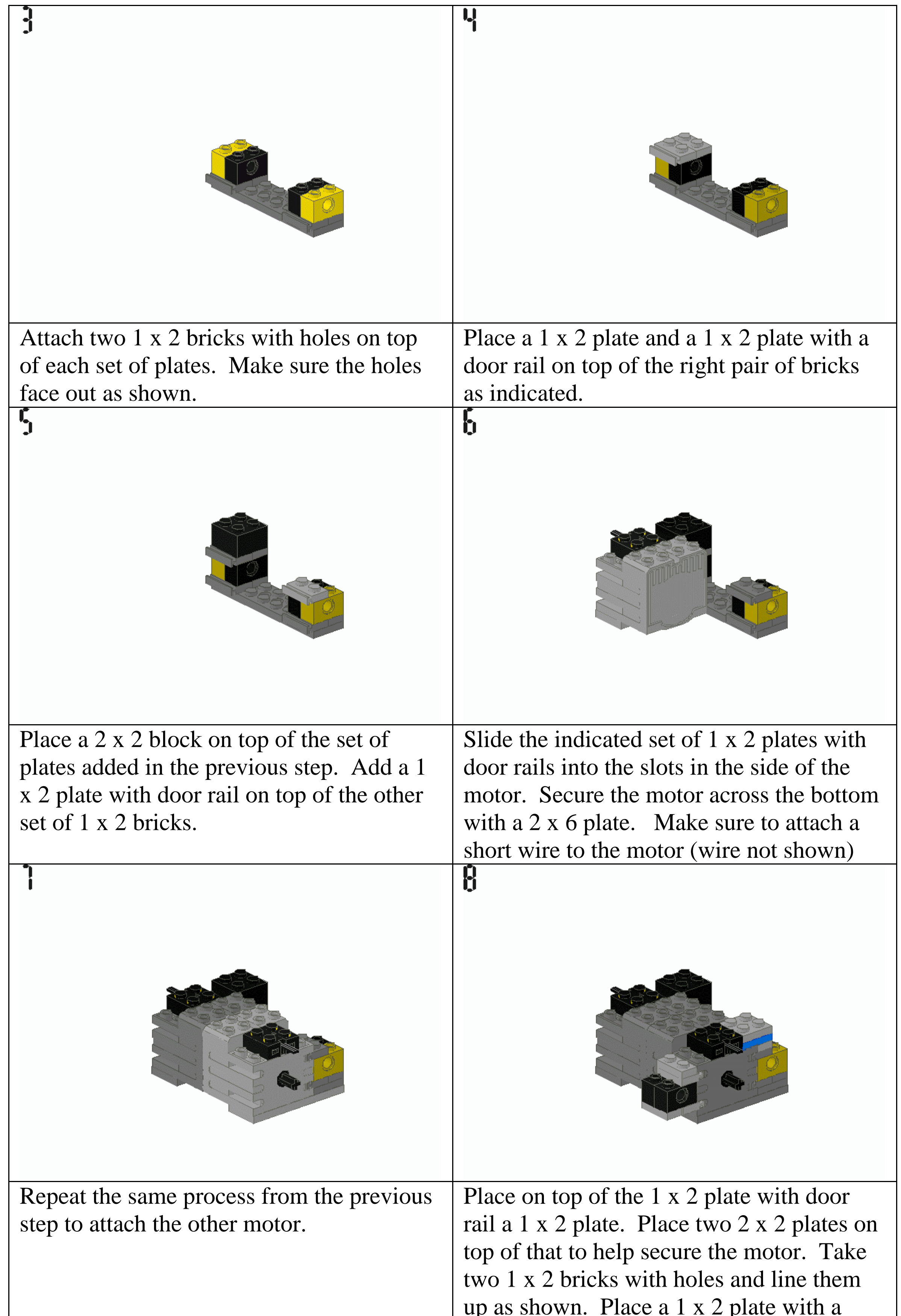
Parts List:

1: 2 x 2 Boat Stud
1: 2 x 2 - 2 x 2 Bracket
1: 2 x 2 Brick
3: 2 x 4 Bricks
1: Light Sensor
1: Mindstorms RCX
1: Rotation Sensor
2: Short Wires
2: Technic Mini-Motors
7: 1 x 2 Plates
6: 1 x 2 Plates with Door Rails
4: 2 x 2 Plates
2: 2 x 4 Plates
1: 3 x 6 Plate without Corners
3: Technic Axles length 6
1: Technic Axle Pin
6: 1 x 2 Technic Bricks with Holes
6: Technic Bushs
3: Technic 8 Tooth Gears
3: Technic 40 Tooth Gears
2: 2 x 6 Technic Plates with Holes
1: 2 x 8 Technic Plate with Holes
2: Wheels

Instructions:

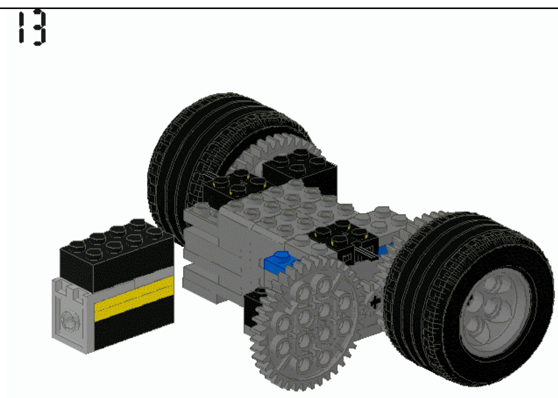
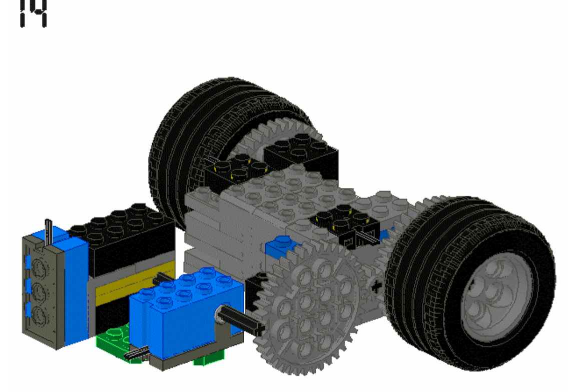
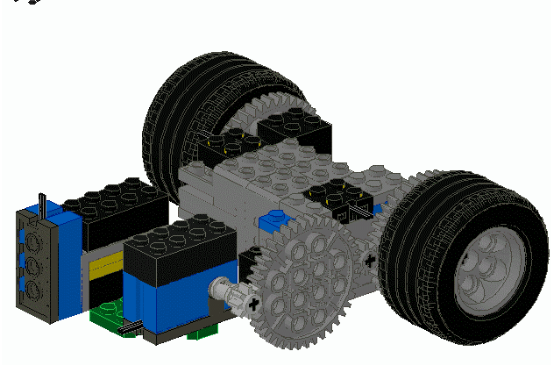
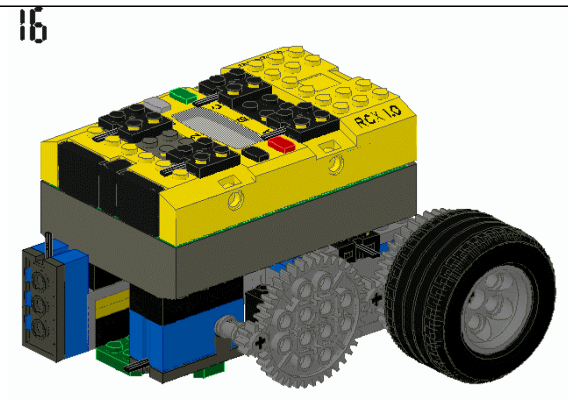
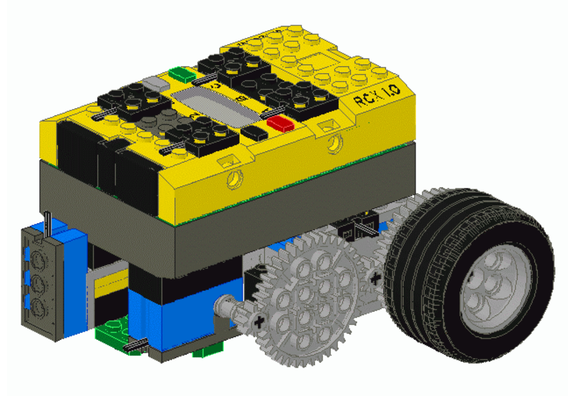
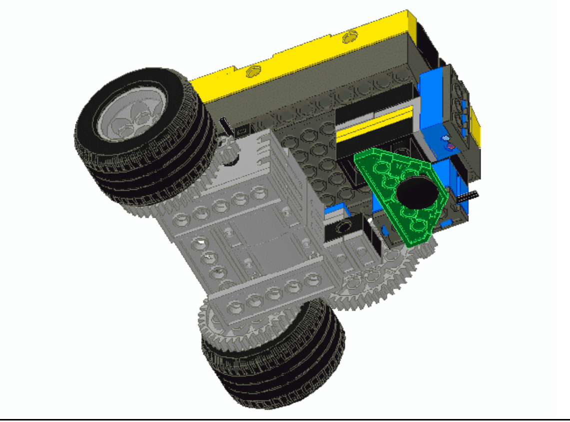


Place a 1 x 2 plate with a door rail at both ends of a 2 x 8 technic plate.	Place 1 x 2 plates behind the other 1 x 2 plates on the 2 x 8 technic plate.



3	4
Attach two 1 x 2 bricks with holes on top of each set of plates. Make sure the holes face out as shown.	Place a 1 x 2 plate and a 1 x 2 plate with a door rail on top of the right pair of bricks as indicated.
5	**6**
Place a 2 x 2 block on top of the set of plates added in the previous step. Add a 1 x 2 plate with door rail on top of the other set of 1 x 2 bricks.	Slide the indicated set of 1 x 2 plates with door rails into the slots in the side of the motor. Secure the motor across the bottom with a 2 x 6 plate. Make sure to attach a short wire to the motor (wire not shown)
7	**8**
Repeat the same process from the previous step to attach the other motor.	Place on top of the 1 x 2 plate with door rail a 1 x 2 plate. Place two 2 x 2 plates on top of that to help secure the motor. Take two 1 x 2 bricks with holes and line them up as shown. Place a 1 x 2 plate with a

	door rail on top and another on the bottom. Attach two more 1 x 2 plates as shown and slide the assembly into the motor.
9	10
Place a 2 x 2 plate to secure the front set of bricks. Place a 1 x 2 plate as shown.	Attach an 8 tooth gear to both motors. Also connect a technic axle pin into the front set of bricks.
11	12
Insert a length 6 axle through the rear set of blocks on either side. Secure it on the inside with a technic brush. Connect a 40 tooth gear on the outside that meshes with the gear on the motor. Secure that with another technic brush. Repeat on the other side. Attach another 40 tooth gear to the pin in the front.	Attach two wheels to the back two axles.

	
Place a 2 x 4 brick in front of the motors. Attach on top of that two 2 x 4 plates. On top of that place a 2 x 2 – 2 x 2 bracket and a 2 x 2 plate. Attach to that another 2 x 4 brick.	Place below those bricks a 3 x 6 plate without corners. Attach a light sensor as shown to the bracket. Attach a rotation sensor to the plate. Insert through the rotation sensor a length 6 axle. Place a 2 x 2 boat stud on the bottom to serve as a skid.
	
Attach to the length 6 axle a hub on both sides of the rotation sensor to secure it in place. Attach an 8-tooth gear on the end. Place a 2 x 4 brick on the rotation sensor.	Place the RCX on the motors and sensors. Attach the wires from the motors and sensors to the RCX as indicated.
The Finished Model	
	

End of document