



linuxjabber.net



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index.html>

## Lego MindStorms: An Introduction

by [Jonathan Knudsen](#)  
 01/31/2000

The Lego MindStorms Robotics Invention System was clearly the coolest toy of 1999. A surprise bestseller, this set from Lego has captured the imaginations and wallets of people of all ages. It offers the chance to breathe life into Lego creations—to make them move and think. Lego robots are a fun, surprisingly easy way to explore the fascinating and complex field of mobile robotics.

What makes Lego robotics such a success?

- First, cheap electronics make small robots affordable.
- Next, Lego bricks have always been a good mechanical prototyping tool. Massachusetts Institute of Technology (MIT) researchers have been combining Lego bricks and small computers in interesting ways for years.

And the fast-growing Internet is an excellent medium for robot builders to gather and share ideas. In the rest of this article, I'll talk about what MindStorms means, what the Robotics Invention System really is, how MIT is involved, the Internet connection, and how all this has changed the Lego company itself.

### What is MindStorms?

MindStorms currently includes three main sets:

- The **Robotics Invention System** (US\$200) contains over 700 Lego bricks and a special robot brain called the RCX. It also includes software (Windows) for programming your robotic creations.
- The **Robotics Discovery System** (\$150) is a scaled-down version of the Robotics Invention System. It includes a robot brain, the Micro Scout, which is similar to the RCX. Robot programming is done through front panel of the Scout.
- The **Droid Developer Kit** (\$100) is a set for building very simple robots, including R2D2 from the Star Wars movies. It also comes with the Micro Scout and preprogrammed behaviors.

Of the three sets, the Robotics Invention System is the most powerful (and expensive). It has also been around the longest (since Fall 1998). It has been thoroughly examined and documented, and it supports MindStorms, a large body of software. For these reasons, when many people say MindStorms they mean the Robotics Invention System.

### What's in the box?

The Robotics Invention System consists of more than 700 Lego bricks and software for your desktop computer. Most of the bricks are standard pieces from Legos' Technic line, such as beams, plates, gears, axles, wheels, pulleys, and various other doodads. But some of the pieces are specific to robot building.

The most important special brick is the RCX, the robot brain, shown in Figure 1.

### Lego MindStorms

- MindStorms is a line of robotics kits from the Denmark-based Lego company which is famous for making plastic bricks.

- The kits include Lego brick hardware and a software development environment. You can write programs by snapping together visual program modules as you snap together bricks.

- Lego is a big enough player to create a standard in a hobby area (building toy robots) that has lacked one up until now.

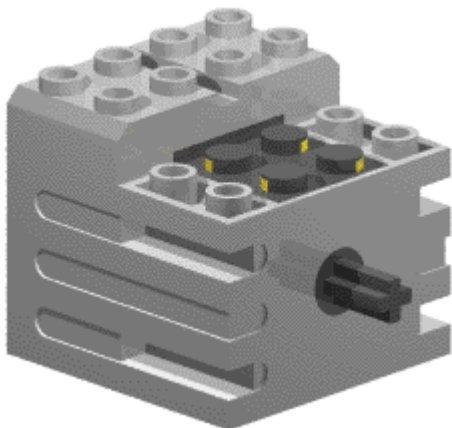
### In This Column

- [What is MindStorms?](#)
- [RCX Programming](#)
- [Lego and MIT](#)
- [Lego Glasnost](#)



**Figure 1. The RCX is the brain of the Robotics Invention.**

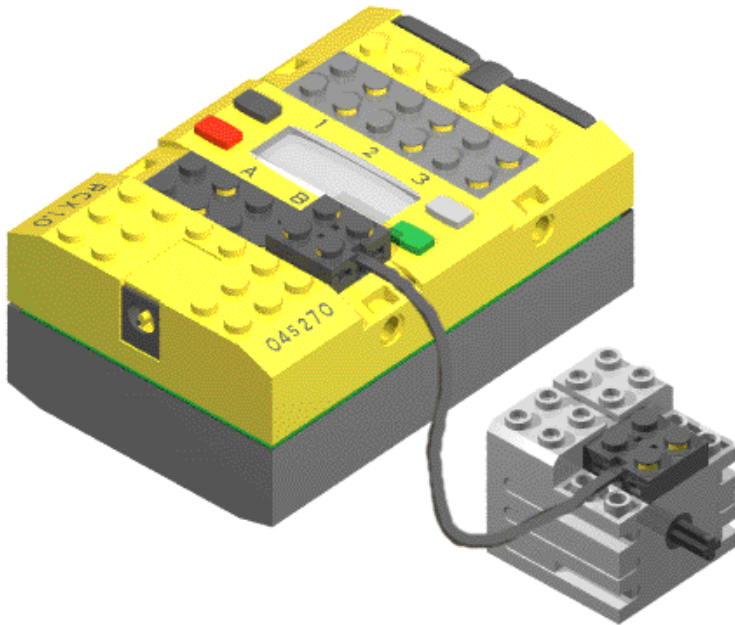
The RCX's outputs can drive motors or lights. The Robotics Invention System also includes two motors like the one shown in Figure 2.



**Figure 2. One of two motors included in the Robotics Invention System.**

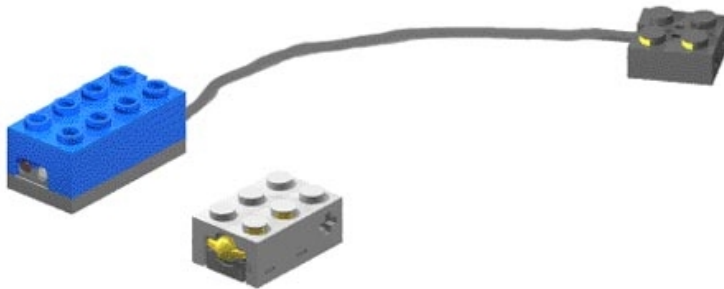
The motor is internally geared down, which means the output shaft moves at a useful speed and probably won't need much external gearing. An internal flywheel makes the output motion very smooth.

To connect a motor to one of the RCX's outputs, you use a special "wire brick." This is a wire with special Lego connectors on each end. You simply snap one end on the motor and the other end on one of the RCX's outputs (Figure 3). The connectors are cleverly designed so that you can attach wire bricks in any orientation.



**Figure 3. Connecting motors and sensors to the RCX is as easy as snapping Lego bricks together.**

The Robotics Invention System also includes sensors that can be attached to the RCX's inputs. The set comes with two touch sensors (switches, basically) and one light sensor (Figure 4).



**Figure 4. One light sensor and two touch sensors come with the Robotics Invention System.**

The light sensor includes a wire brick. The touch sensors are attached to the RCX's inputs just as the motors attach to the outputs. You can purchase other types of sensors (temperature and rotation sensors) from Lego. If you're handy with a soldering iron, you can even build your own sensors.

---

Jonathan Knudsen is an author at O'Reilly & Associates. His books include [The Unofficial Guide to Lego Mindstorms Robots](#), [Java 2D Graphics](#), and [Java Cryptography](#).

---

#### **Related Articles:**

##### [Lego MindStorms: RCX Programming](#)

In Part Two of this introduction, Jonathan Knudsen looks at the programming interface and scripting options for MindStorms.

##### [Lego MindStorms: Lego and MIT](#)

In Part Three of this introduction, Jonathan Knudsen looks at the connection between the Massachusetts Institute of Technology and Lego's research dollars.

##### [Lego MindStorms: Lego Glasnost](#)

In Part Four of this introduction, Jonathan Knudsen tells how Lego came to grips with its evolution from a plastic bricks company to one that also deals with a consumer base of programmers.

**oreillynet.com** Copyright © 2000 O'Reilly & Associates, Inc.



linuxjabber.net



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index1a.html>

## Lego MindStorms: RCX Programming

by [Jonathan Knudsen](#)  
01/31/2000

### RCX: A lingua franca for robot builders

Hobbyists have been building robots for decades. But the Robotics Invention System offers a standard.

If someone builds a robot with the pieces from the Robotics Invention System, anyone else with the same set can build it too. Robots with nonstandard parts are much harder to reproduce.

Also, programs you write for the RCX can run on anyone else's RCX. Homebrew robots tend to have software that's hard to reuse in other robots.

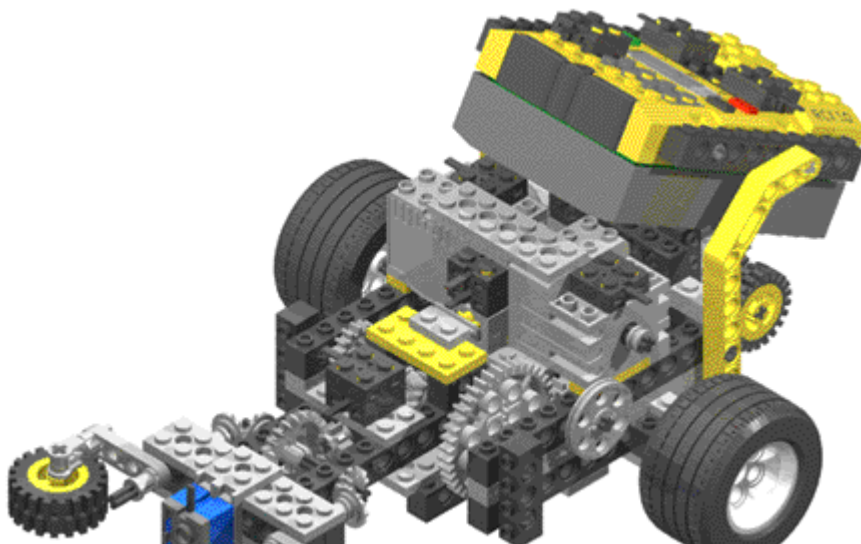
The time was ripe for a product like the Robotics Invention System. Lego bricks are a good, standard, mechanical prototyping toolkit. And some standard robot brains have been made, such as MIT's [HandyBoard](#) or the [BASIC Stamp](#) from [Parallax](#). The beauty of the Robotics Invention System is that it combines a standard mechanical toolkit with a standard software platform. If you build something cool, someone else can easily build it too. This makes it easy to exchange ideas and programs with other Lego robot builders.

### The forgotten challenge of mechanical design

One of the joys of creating Lego robots is the challenge of mechanical design. Mechanical design is something kids generally understand better than adults; after all, they get to play with wooden blocks, Lego bricks, and other physical toys. Many adults spend their workdays in front of a computer and have forgotten the rewards of mechanical design. Building robots with the Robotics Invention System is a great way to reacquaint yourself with the physical world.

Lego bricks are a great prototyping tool because they are so versatile. It's easy to take something apart and rebuild it as many times as you like.

Here's one of my robots, [Minerva 1.5](#). She can drive around and pick things up with her hand.



### RCX Programming

- Lego is a big enough player to define a standard for hobby robotics.

- You can program the MindStorms RCX brain with Legos' visual development environment or with newly developed programming languages like [NQC](#), [legOS](#), and [pbFORTH](#).

- It's a challenge working with software and hardware together.

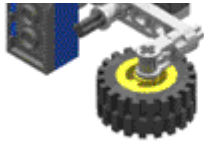


Figure 5. Minerva 1.5, a robot with an arm and gripper.

Other people have built tank-style robots, [walking robots](#) (with two, four, six, or even eight legs), [a model of a VCR](#), [models of natural disasters](#), robots that slither, robots that play games, [robots that play tic-tac-toe](#), copying machines, and almost anything else you can imagine.

## Developing with RCX

How do programs get from a computer to the RCX? The answer is an infrared light link. The Robotics Invention System includes an infrared tower that attaches to a serial port on your computer. The RCX itself has an infrared port. When you're ready to download a program to the RCX, you just point the infrared tower at the RCX, just like pointing a remote control at a television.

Special software on the RCX, called the *firmware*, acts as a kind of RCX operating system. It can receive programs from the infrared port and respond to the front panel buttons to start and stop programs.

The development software that comes with the Robotics Invention System runs on Windows platforms. Programs are created by snapping together different functional blocks of RCX code, just as you create a robot by snapping together Lego bricks. (See Figure 6.)

### In This Column

- [What is MindStorms?](#)
- [RCX Programming](#)
- [Lego and MIT](#)
- [Lego Glasnost](#)



Figure 6. The RCX code graphic programming environment.

Palettes of program blocks are on the left. Here, I've created a simple program (the green blocks) that turns outputs A and C on for 4 seconds, then turns them off and plays a sound.

## Development Alternatives

Development Alternatives include [NQC](#), [legOS](#), and [pbFORTH](#). Although RCX code is good for people who haven't programmed before, it has some serious limitations. The biggest problem is a lack of variables. In response to this, dedicated and innovative fans have created a dozen alternate programming environments for the RCX. The most important of these are NQC, legOS, and pbFORTH.

NQC is based on text source code files with a C-like syntax. It's the most popular and widespread alternate development tool for the RCX, but it is still subject to the limitations imposed by the default RCX firmware. To give you an idea what NQC source

code looks like, here's a program that does the same thing as the RCX code program shown above:

```
task main() {
    SetPower(OUT_A + OUT_C, OUT_FULL);
    Fwd(OUT_A + OUT_C);
    On(OUT_A + OUT_C);
    Wait(400);
    Off(OUT_A + OUT_C);
    PlaySound(SOUND_CLICK);
}
```

With legOS, you can write robot programs in real C using GNU compiler tools. And pbFORTH is a Forth language interpreter that runs on the RCX. Both legOS and pbFORTH both offer more power than NQC, but involve replacing the firmware on the RCX, which make them harder to use.

---

### Related Articles:

#### [Lego MindStorms: RCX Programming](#)

In Part Two of this introduction, Jonathan Knudsen looks at the programming interface and scripting options for MindStorms.

#### [Lego MindStorms: Lego and MIT](#)

In Part Three of this introduction, Jonathan Knudsen looks at the connection between the Massachusetts Institute of Technology and Lego's research dollars.

#### [Lego MindStorms: Lego Glasnost](#)

In Part Four of this introduction, Jonathan Knudsen tells how Lego came to grips with its evolution from a plastic bricks company to one that also deals with a consumer base of programmers.

**Return to:** [Lego MindStorms: An Introduction](#)

oreillynet.com Copyright © 2000 O'Reilly & Associates, Inc.



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index1b.html>

## Lego MindStorms: Lego and MIT

by [Jonathan Knudsen](#)  
 01/31/2000

### From MIT with love

MIT and Lego have been in bed for quite a while. The history of this relationship, in many ways, is a road that leads to the MindStorms product line. Lego gives MIT money so they can investigate how children learn, how robots think, and other heavy topics. In return, Lego gets to take ideas developed by MIT and make products out of them. In fact, MindStorms is a borrowed name, originally used in a 1980 book by [Seymour Papert](#) titled *MindStorms: Children, Computers, and Powerful Ideas*. Papert cofounded MIT's Artificial Intelligence Laboratory back in the 1960s and is now an instrumental figure at the MIT Media Lab. The book is about the creation of LOGO, a computer language for children.

Later work at the Media Lab led directly to the Robotics Invention System. Researchers at MIT have used Legos for mechanical prototyping for many years. A recent project headed up by [Fred Martin](#) is the [MIT Programmable Brick](#), which inspired the design of the RCX.

MIT, of course, has its own firmware that they run on the RCX, and its own LOGO-like programming environment. But this software is not publicly available.

The MIT Media Lab hosted an extraordinary event in October 1999, about a year after the release of the Robotics Invention System. The event was [MindFest](#), subtitled "A Gathering of Playful Inventors." Several hundred people attended, including MIT staff, Lego employees, enthusiasts, educators, and kids. The tutorials and panel discussions included everything from "Artistic Machines" and "Virtual Tinkering" to "Young Inventors" and "Robotics in the Classroom." It was a fascinating weekend, and a good chance to see the recreational and educational aspects of Lego robotics.

### It's a Net thing

As if a distinguished pedigree from MIT wasn't enough, the MindStorms phenomenon has a heavy Internet element as well.

The [Lego's MindStorms web site](#) serves as the official home of Lego robotics on the Web. If you own a Robotics Invention System set, you have an automatic membership at this site. You can use the site to upload robot pictures and programs, and browse through the creations of other fans.

The widely acknowledged center of the Lego online universe is the fan-created Lego Users Group Network ([LUGNet](#)). LUGNet is a searchable collection of discussion forums. Every forum is available as a regular NNTP newsgroup, on the Web, or as an e-mail list. There are groups and subgroups for every corner of the Lego universe. If you've got a question, it's probably already been answered here. If not, this is the place to ask.

In [lugnet.robotics](#) and its subgroups, for example, you'll find discussions about using a power adapter on the infrared tower, exotic drives like Killough's platform, and the tri-star design, techniques for using the infrared port on the RCX as a proximity detector, and almost anything else you can imagine. Here you'll also find the hackers who reverse engineered the RCX and created NQC, legOS, and pbFORTH.

### MIT and Lego

- Lego gives money to MIT to research how children learn.
- MIT gives product ideas to Lego.
- MIT hosted the [MindFest](#) conference.
- Key sites include [Lego's MindStorms](#) and [LUGNet](#).

### In This Column

- [What is MindStorms?](#)
- [RCX Programming](#)
- Lego and MIT
- [Lego Glasnost](#)

### Related Articles:

#### [Lego MindStorms: RCX Programming](#)

In Part Two of this introduction, Jonathan Knudsen looks at the programming interface and scripting options for MindStorms.



[Lego MindStorms: Lego and MIT](#)

In Part Three of this introduction, Jonathatn Knudsen looks at the connection between the Massachusetts Institute of Technology and Lego's research dollars.

[Lego MindStorms: Lego Glasnost](#)

In Part Four of this introduction, Jonathan Knudsen tells how Lego came to grips with its evolution from a plastic bricks company to one that also deals with a consumer base of programmers.

**Return to:** [Lego MindStorms: An Introduction](#)

oreillynet.com Copyright © 2000 O'Reilly & Associates, Inc.



linuxjabber.net



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/01/31/mindstorms/index1c.html>

## Lego MindStorms: Lego Glasnost

by [Jonathan Knudsen](#)  
 01/31/2000

### Lego Glasnost

It would be easy to congratulate Lego on its brilliant product strategy. The truth is, however, that like many brilliant products, this one happened almost by accident. The Robotics Invention System was created by a renegade group, with a high-ranking champion, within the Lego company.

If you take a look at the history of Lego, you can appreciate what an upheaval MindStorms represents. Lego has been molding plastic for the last 50 years. The Robotics Invention System, on the other hand, involves circuit design, embedded systems programming, and desktop application programming. All of these are pretty crazy fields for a plastics company to jump into.

But jump they did, and the results are stunning. Lego managed to transfer its high quality in plastic molding into high-quality technology. Lego has been a slow-moving, conservative company for many years. Now, like it or not, it is also a high-tech company.

Lego is trying to respond to the unexpected success of the Robotics Invention System. It is also trying to respond to the unexpected dedication of RCX hackers who reverse engineered important parts of the RCX within a few weeks of its release. Many Lego fans, particularly MindStorms fans, are adults—a group that has never been part of Lego's target audience. So far, Lego's response in the aftermath of MindStorms consists of three important steps:

1. [In November 1999](#), the official MindStorms web site recognized NQC source files as a valid file type for uploading to member's project areas. This was an implicit endorsement of NQC as a valid RCX development environment, and an important recognition of the online community.
2. [Later that month](#), Lego published juicy details of the [internal workings of the Scout](#), the robot brain from the Robotics Discovery Set.
3. In December 1999, a vice president from Lego made [an unprecedented appearance at LUGNet](#), promising a new "direct-to-you communications and commerce channel." "We are listening," he said—encouraging words indeed.

### Looking forward

It's been a wild ride for the last 18 months or so. Lego is running to catch up. Meanwhile, people all over the world are having a blast building and programming robots.

In the months to come, I'll guide you on a tour of the most important landmarks in the Lego robot universe. The contents of these articles are, to some degree, up to you. If you have ideas, [I'd love to hear about them](#). I will introduce NQC programming and talk about the curious problem of moving a robot in a straight line.

### Lego Glasnost

- Lego's been shipping plastic bricks for 50 years. It took them a few months to figure out the personality of their new, programmer consumer.

- Lego's been moving to gradually accept user innovations like NCQ.

- After an initial chilliness toward the open source attitude of the hackers who reverse-engineered some RCX code, Lego now promises to listen to and work with its new customers.

### In This Column

- [What is MindStorms?](#)
- [RCX Programming](#)
- [Lego and MIT](#)
- Lego Glasnost

### Related Articles:

#### [Lego MindStorms: RCX Programming](#)

In Part Two of this introduction, Jonathan Knudsen looks at the programming interface and scripting options for MindStorms.

#### [Lego MindStorms: Lego and MIT](#)

In Part Three of this introduction, Jonathan Knudsen looks at the connection between the Massachusetts Institute of Technology

and Lego's research dollars.

[Lego MindStorms: Lego Glasnost](#)

In Part Four of this introduction, Jonathan Knudsen tells how Lego came to grips with its evolution from a plastic bricks company to one that also deals with a consumer base of programmers.

**Return to:** [Lego MindStorms: An Introduction](#)

oreillynet.com Copyright © 2000 O'Reilly & Associates, Inc.



linuxjabber.net



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/02/25/mindstorms/NQC.html>

## Lego MindStorms: Programming with NQC

by [Jonathan Knudsen](#)  
02/25/2000

*In the second column in his series on Lego MindStorms robots, O'Reilly & Associates author Jonathan Knudsen introduces us to an alternative programming environment for the robots, NQC. On the first page, he introduces NQC and its properties and tells how to get it. On page two, he explains more details, including program flow, inputs, and variables. On the final page, he lists the code for a more complex program that has the robot seeking out a dark place to hide.*

[Last month](#) I told you about the [Lego MindStorms](#) Robotics Invention System and why it's such an interesting product. This month, you'll get right to the fun stuff: building and programming your own robot. You'll learn how to program using [Not Quite C \(NQC\)](#). Created by Dave Baum, NQC is one of the alternate development environments I talked about in last month's column. NQC is a great piece of software for several reasons:

- NQC is freely distributed under the Mozilla Public License (MPL).
- It runs on Linux and Mac OS as well as Windows.
- It has a syntax that looks like C, making it easy to learn for C programmers.
- NQC uses the default firmware that comes with the Robotics Invention System. This makes it easier to deal with than some of the other, more powerful development environments that use alternate firmware.

### The RoboTag robot, a programming testbed

Before you start writing programs, you'll need to build a robot for testing. I suggest the RoboTag robot from [The Unofficial Guide to Lego MindStorms Robots](#). RoboTag is a simple, sturdy, tank-style robot. It has a front bumper and a downward-pointing light sensor.



Figure 1. RoboTag, from [The Unofficial Guide to Lego MindStorms Robots](#)

RoboTag's building instructions are online. To get the most out of this tutorial, [click here for the building instructions](#) and then return to get into the NQC programs. If you're only interested in NQC's capabilities, keep reading.

### A first look at NQC

Although you could certainly program your robot using RCX code (the programming environment that comes with the Robotics Invention System), there are compelling reasons to

[Previous Lego Mindstorm](#)

make the jump to NQC. Variables are probably the most important thing -- NQC has them and RCX code doesn't. Using variables can make your robot a lot smarter. Later on, for example, you'll see how using a variable makes a program more robust, allowing us to determine a light sensor threshold on the fly rather than hard coding the value. To get started, you'll need to [download NQC](#). There are versions for Mac OS, Linux, and Windows.

You can use any text editor to create NQC source files. If you're running on Windows, consider Mark Overmars' [RCX Command Center \(RcxCC\)](#), a smooth graphic interface that runs over NQC.

RcxCC provides a nice, syntax-colored editor. MacNQC also includes an editor. On Linux or Windows, just use the text editor of your choice. Save the following as "Hello.nqc":

```
task main() {
  OnFor(OUT_A + OUT_C, 200);
}
```

To compile this program, just use the `nqc` command, like this:

```
C:\nqc Hello.nqc
```

(With MacNQC, you'll use a menu item instead. With RcxCC, there's a toolbar button that compiles a program.)

If you get any errors, check your typing and try again. Otherwise, you're ready to download the program to the RCX. Make sure your RCX is turned on and use the `nqc` command with the `-d` option:

```
C:\>nqc -d Hello.nqc
```

(If you need to use a serial port different from the default, use the `-s` option.)

Go ahead and run the program by pressing the Run button on the RCX. Your robot should move forward for 2 seconds, then stop.

## Controlling outputs

NQC includes a suite of commands for controlling the outputs. The three output properties that can be independently controlled are

- mode,
- power, and
- direction.

You might, for example, want an output to be on (mode) with full power (power) going forward (direction).

In the Hello.nqc program, we took advantage of the fact that outputs are full power and forward by default. All we had to do was turn on outputs A and C to make the robot move forward.

The `OnFor()` command turns one or more outputs on for a certain amount of time, measured in hundredths of a second. Outputs are specified with some combination of the constants `OUT_A`, `OUT_B`, and `OUT_C`. Multiple outputs can be specified by adding the constants together. If you want to control more than one output, you can just add the constants together (equivalent to a boolean "OR" here). In our simple program, we used `OUT_A + OUT_C` to specify outputs A and C.

Let's look at the modes first. You can turn outputs on and off with the `On()` and `Off()` commands. There are two flavors of off: braking and floating. The default, `Off()`, puts the motors in a braking mode, so they resist turning. The other flavor, `Float()`, turns off power to the motors but does not resist turning.

To see how this works, let's try a simple modification of our first program. Save the following as "Coaster.nqc":

```
task main() {
  OnFor(OUT_A + OUT_C, 200);
  Float(OUT_A + OUT_C);
}
```

After this program exits, you'll notice that RoboTag coasts to a stop, rather than stopping abruptly. Also, you can freely turn the treads with your fingers.

## Columns:

- [The Straight and Narrow](#)
- [Tools to Save Your MindStorm Models](#)
- [Lego MindStorms: An Introduction](#)
- [Lego MindStorms: RCX Programming](#)

## NQC Resources

- [The Unofficial Guide to Lego MindStorms Robots](#) contains a chapter about NQC. It's [available online](#).
- The [NQC web site](#) includes documentation.
- The RcxCC web site also includes a [tutorial about NQC](#).

The `OnFor()` command is an interesting variation that turns outputs on for a certain amount of time, then off (in brake mode).

The outputs have variable power, from 1 to 7. NQC defines some handy constants, like `OUT_LOW` (1), `OUT_HALF` (4), and `OUT_FULL` (7). You can set the power of one or more outputs with the `SetPower()` command.

The direction of the outputs is either forward or reverse. You can control this with the `Fwd()` and `Rev()` commands. Two combination commands, `OnFwd()` and `OnRev()`, allow you to specify a mode and a direction at the same time.

By default, the outputs have full power in the forward direction. Instead of relying on the defaults, we could rewrite the first program as follows:

```
task main() {
  SetPower(OUT_A + OUT_C, OUT_FULL);
  Fwd(OUT_A + OUT_C);
  OnFor(OUT_A + OUT_C, 200);
}
```

*On the next page, I'll go into more details of program flow, reading inputs, and variables.*

[Next Page](#) 

## Lego MindStorms: Programming with NQC

by [Jonathan Knudsen](#) |

### Program flow and waiting

If you've programmed in C, you'll be very comfortable with NQC. You get the standard `for` and `while` loops. NQC also supports `if` and `if else`. The syntax is just like C. If you haven't programmed with C, it's not hard to learn.

NQC supports two commands that allow you to delay execution of a program. The `Wait()` command is a simple delay, where you specify the time in hundredths of a second. If you want to wait for a certain condition to become true, use `until()` instead. This is actually a kind of loop, with this form:

```
until([condition])
  [statements]
```

The condition is evaluated; as long as it is false, the statements are executed repeatedly. The `until()` loop follows C syntax, such that multiple statements can be enclosed in brackets. In the next section, we'll write a program with a simple `until()` loop.

### Reading inputs

You can make your robot dance now, but to program interesting behavior you'll need to respond to RoboTag's sensors. Here's another program that allows RoboTag to avoid obstacles. Save the following program as "Bounce.nqc":

```
task main() {
  SetSensor(SENSOR_1, SENSOR_TOUCH);
  while (true) {
    // Move forward.
    OnFwd(OUT_A + OUT_C);
    // Wait for a bump.
    until (SENSOR_1 == 1)
    ;
    // Spin left.
    Rev(OUT_A);
    Fwd(OUT_C);
    Wait(50);
  }
}
```

There are several new things in this program. Let's look at the sensor-reading stuff first. The RCX needs to know what kind of sensors are attached to its inputs in order to interpret their signals correctly. In this program, we use `SetSensor()` to tell the RCX that a touch sensor is attached to input 1:

```
SetSensor(SENSOR_1, SENSOR_TOUCH);
```

NQC includes constants that represent each input: `SENSOR_1`, `SENSOR_2`, and `SENSOR_3`. There are also constants representing all the basic sensor types, like `SENSOR_LIGHT`, `SENSOR_ROTATION`, `SENSOR_CELSIUS` for temperature sensors, and others. If you build your own sensors, there are more flexible commands for configuring the inputs, but I won't get into them here.

Once you configure an input, you can read its value by using `SENSOR_1`, `SENSOR_2`, and `SENSOR_3`. In this program, we configured input 1 to have a touch sensor. It will have a value of either 0 or 1.

The algorithm in this program is simple: move forward until the bumper is hit. Then spin left for half a second. The whole thing is enclosed in an infinite `while()` loop.

## Variables

NQC supports 31 integer variables. To use a variable, just declare it as in C. Here's a simple example:

```
int x;

task main() {
    x = 14;
    x++;
}
```

## Multitasking

It gets even better. The default RCX firmware supports multitasking programs, i.e. programs that do more than one thing at a time. Up until now, all of the sample programs have had one task, `main`. Every program must have a `main`; it's the task that is executed when you press the Run button on the RCX. However, you're free to start other tasks from `main`:

```
task main() {
    start secondTask;
    start thirdTask;
}

task secondTask() {
}

task thirdTask() {
}
```

You can stop tasks, too, using `stop`. Programs can have up to ten tasks. To communicate information between tasks, you'll have to use variables. NQC supports several flavors of subroutines too, but I'm not going to cover them here.

*To close, I'll detail a program that instructs RoboTag to roam around avoiding obstacles, until it finds a nice dark place to hide.*

[Next Page](#) ►

## Lego MindStorms: Programming with NQC

by [Jonathan Knudsen](#) |

### Going out with a bang

As a final example, let's develop a dark seeking program for RoboTag. We'll make the robot roam around on the floor, avoiding obstacles. When it gets to some place dark, like under a piece of furniture, it will stop.

The light sensor returns a value from 0 (dark) to 100 (light). In theory, the whole range of values can be returned. In practice, you'll probably never see readings outside the 30-70 range. We could hardcode a threshold into our program so the robot knows when it's in a dark place. But this is not very reliable; the same program might not work at different times of day. Instead, we'll take a baseline reading of the light sensor when the program starts. We'll stop the robot when the light sensor value falls significantly below this baseline.

The obstacle avoidance is similar to what you saw before, with a twist. This time, the robot picks a random direction to turn when it bumps into something. It turns for a random amount of time, too, from 0.5 s to 2.5 s. The key to this is the `Random()` command, as you'll see.

The program is split across three different tasks for clarity. The `main` task initializes the inputs and starts up the other tasks. The

`avoid` task takes care of obstacle avoidance, responding to pushes on the bumper. The `dark` task shuts down the robot's motors, and the program, when the robot moves into a dark place.

Here's the whole program:

```
int baseline;

task main() {
    // Initialize.
    SetSensor(SENSOR_1, SENSOR_TOUCH);
    SetSensor(SENSOR_2, SENSOR_LIGHT);
    baseline = SENSOR_2;

    // Start tasks.
    start avoid;
    start dark;
}

task avoid() {
    On(OUT_A + OUT_C);
    while (true) {
        if (SENSOR_1 == 1) {
            // Back away.
            Rev(OUT_A + OUT_C);
            Wait(50);
            // Turn a random direction.
            if (Random(1) == 0) {
                Fwd(OUT_A);
                Rev(OUT_C);
            }
            else {
                Rev(OUT_A);
                Fwd(OUT_C);
            }
            // Turn for a random duration.
            Wait(50 + Random(200));
            // Go forward again.
            Fwd(OUT_A + OUT_C);
        }
    }
}

task dark() {
    until (SENSOR_2 <= baseline - 5)
        ;
    Off(OUT_A + OUT_C);
    stop avoid;
}
```

## New directions

Now that you've got a feel for NQC, there are all sorts of interesting things you can try. NQC lets you do things like send and receive data with the infrared port (to communicate with other robots or PCs), play sounds, control the display, use timers, and store data in a datalog. Here are three suggestions:

- **Write a wall-following program.** The algorithm is straightforward: drive forward in a gently turning arc. When you hit the wall, turn away from it a little. Then start over with the forward arc. You'll need to modify RoboTag's bumper a little to make this work -- the current design isn't very good at detecting collisions at the front corners of the robot.
- **Write a line-following program.** This behavior causes the robot to move along a path marked on the floor. (If you don't want to mark up your carpets with electrical tape, you can just use the test pad that comes with the Robotics Invention System.) Line following is surprisingly hard. For two different approaches, see Chapter 3 of [my book](#) and Chapter 8 of [Dave Baum's book](#).
- **Add sound to your program.** This is useful for debugging; you can play different sounds to signify different parts of your program.

I hope you've enjoyed this quick tour through NQC. It's a great way to get involved with robot programming.

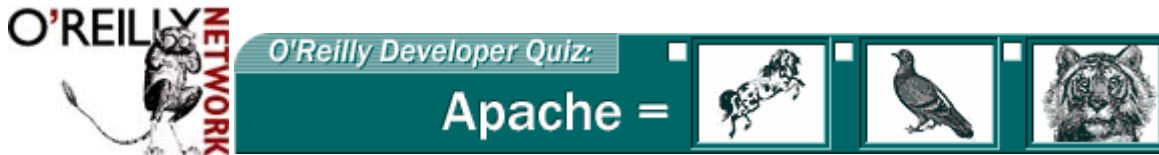
Building a robot of your own? Creating interesting programs for it? Tell us what you're up to in our [Mindstorms forums](#).

---

Jonathan Knudsen is an author at O'Reilly & Associates. His books include [The Unofficial Guide to Lego MindStorms Robots](#), [Java 2D Graphics](#), and [Java Cryptography](#).







Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/03/29/mindstorms/index.html>

## Tools to Save Your MindStorm Models

by [Jonathan Knudsen](#)  
 03/29/2000

If you've done much work with Lego bricks, you've probably built some models you felt were outstanding -- good enough to save for posterity. You might have wanted to describe how to build the model for other people, or simply to rebuild it yourself some time in the future.

Lego supplies excellent instructions with their sets. These are full color, carefully drawn instructions, usually showing the parts needed for each step and including handy arrows that describe where everything goes. Being an international company, Lego makes the pictures very explicit so they don't rely on a particular language. If you own Lego sets, take some time to appreciate the instructions that Lego provides. They're extremely clear.

What about the rest of us? How can we create Lego building instructions? There are suprisingly good tools available for free on the Internet. Lego enthusiasts form a very strong community on the Internet, and some of them are good programmers too. In this article, I'll describe how you can use several of the available tools to create high-quality building instructions. See the [list](#) at the end of this article for links to these tools.

### LDraw and LEdit

Any story about computer-generated building instructions has to begin with LDraw and LEdit. These DOS applications were created by the late James Jessiman in 1996. LEdit is a program for creating model instruction files, while LDraw is the program that displays them. Models are described as a series of steps, and LDraw is capable of showing the model being built, one step at a time.

The next screenshot shows the same model in LEdit. At the top of the screen you can see actual lines from the model file (more on this soon). Within LEdit, you can add parts to a model, remove parts, move and rotate parts, and change the colors of parts. You can also insert markers that signify a complete step.

LEdit and LDraw are good tools, but it takes some time to learn how to use them. LEdit, in particular, has a pretty steep learning curve. The important legacy of LEdit and LDraw is their file format, which we'll examine next.

### The DAT file format

LDraw and LEdit models are stored in "DAT files," with a `.dat` file extension. This file format is the center of most subsequent work in the Lego modeling software world. No matter what tools you use, you may very well find yourself staring at a DAT file in a text editor someday, so a basic understanding of the format is useful.

The file format encompasses two functions:

- part modeling and
- model building.

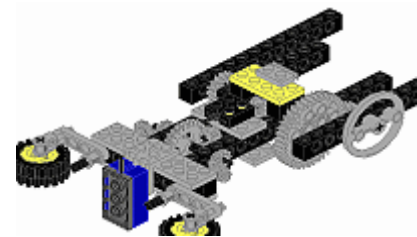


Figure 1. This screenshot of LDraw shows the arm subassembly from Minerva, a robot from [The Unofficial Guide to Lego MindStorms Robots](#). (click on image to view full size)

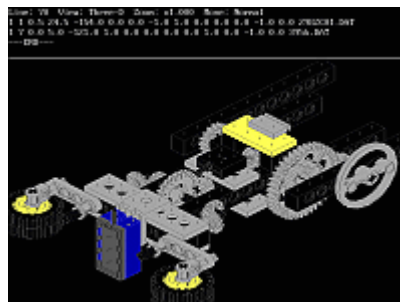


Figure 2. The same arm subassembly viewed in LEdit. (click on image to view full size)

Individual Lego bricks can be modeled as DAT files by assembling collections of lines and surfaces. DAT files can reference other DAT files, so a complex brick can be built up from simpler pieces. A regular brick DAT file, for example, might reference another DAT file that represents the studs, or bumps, on the brick.

A Lego model DAT file is built by referencing the individual brick DAT files that make up the model. Let's look at a simple example. The model in Figure 3 is built from two bricks

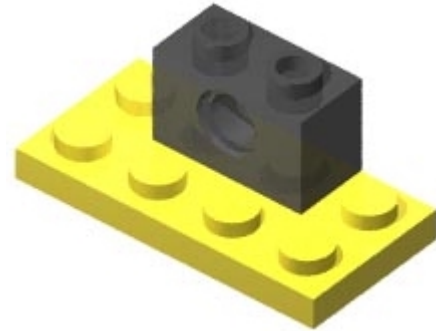


Figure 3. Image of a two-stud Technic brick and a 2 x 4 plate, the basis for the DAT file.

Here's the DAT file for this simple model:

```
0 Untitled
0 Name: simple.dat
0 Author: MLCad
0 Unofficial Model
1 14 0 0 0 1 0 0 0 1 0 0 0 1 3020.DAT
1 0 0 -24 10 1 0 0 0 1 0 0 0 1 3700.DAT
0
```

You can pretty much figure it out by looking at it. Every line has a type, which is just a number at the beginning of the line. A 0 indicates a comment line, something that adds information to the DAT file but doesn't add to the model itself. The two important lines begin with a 1, which imports another DAT file into the model. In this case we're using *3020.DAT*, the 2 x 4 plate, and *3700.DAT*, the two-stud Technic brick with the hole through it. What are all those other numbers on those lines? Those show where the imported part should be placed and how it should be rotated relative to the model.

Complex model files look a lot like this simple one, just with many more "1" lines.

The DAT files for bricks are named according to their part numbers. Most bricks have numbers molded into them; you can see these numbers if you look closely. Over the years, the online Lego community has modeled (that is, created DAT files for) more than 1,500 bricks. If you can't find a brick DAT file, sometimes friendly folks on the Internet will create it for you.

An important extension of the DAT format, the Multiple Part DAT (MPD) file, allows you to specify multiple subparts inside a single file. For example, the entire Minerva robot model is represented in a file called *Minerva.mpd*. Inside this file, a subassembly called *MinervaArm.dat* is defined. It is used as a component of the entire model.

[Next Page](#) ►

## Tools to Save Your MindStorm Models

Jonathan Knudsen |

### The next generation: MLCAD

Now let's look at the next step beyond LEdit. I like to use a Windows program called MLCAD. (There are other options, but this is the one I know about.) The bottom line is that MLCAD is a souped-up DAT or MPD file editor. But you can also think of it as a specialized computer-aided design (CAD) tool. MLCAD shows several different models at once and lets you move pieces around using the mouse. It also shows a portion of the DAT or MPD file you're working on and provides palettes of bricks to make it easy to find the right piece.

Using MLCAD, you can build complex models quickly. It gets a little sluggish after you've added a lot of pieces, but it's still usable.

### Important Glue: I3p

LDraw and MLCAD can display models, but the image quality is not great. To generate professional-looking images, like the ones in the official Lego instructions, you'll need to use POV-Ray, a *raytracer*. This is a program that can create highly realistic computer-generated scenes.

The gap between the world of DAT files and the world of POV-Ray is filled by a little DOS utility called I3p, developed by Lars Hassing. The usage is pretty simple: Just supply a DAT file and I3p creates a POV-Ray file for you. For example, the two-brick model we looked at before can be converted like this:

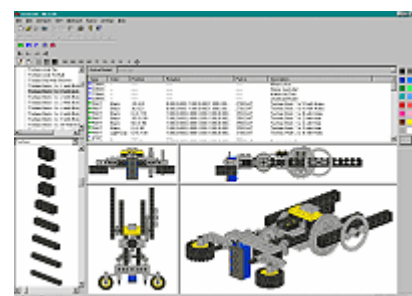


Figure 4. The same Minerva arm model in MLCAD, a Windows DAT/MPD file editor. (click on image to view full size)

```
l3p simple.dat
```

l3p will chug a little and create a `simple.pov` file which can be immediately rendered in POV-Ray.

There are a couple of things you should know about l3p. First, it can only deal with eight-character filenames. Second, there are a couple of command-line options that I find useful. The `-b` option allows you to specify a background color (as red, green, and blue components) for the model. I usually use white, like this:

```
l3p -b1,1,1 simple.dat
```

Finally, the `-o` option specifies that if the POV-Ray file already exists, it should be overwritten:

```
l3p -b1,1,1 -o simple.dat
```

## POV-Ray

To create a scene in POV-Ray, just load the `.pov` file. It will show up in POV-Ray's text editor. You can choose an image size from the drop-down list on the toolbar. Then click the Run button and sit back while your scene is drawn.

POV-Ray has its own language for describing scenes and how they should be drawn. Without knowing the entire language, there are some things you can tweak. The camera angle and the lighting are the most important adjustments.

The camera angle is defined in a block near the end of the `.pov` file generated by l3p. Here's a typical example:

```
camera {
#declare PCT = 0; // Percentage further away
#declare STEREO = 0; // Normal view
//#declare STEREO = degrees(atan2(1,12))/2; // Left view
//#declare STEREO = -degrees(atan2(1,12))/2; // Right view
location vaxis_rotate(<55.3377,-33.905,-43.7907> +
    PCT/100.0*<44.4925,-36.328,-44.4925>,
    <-1616.32,-3959.17,1616.32>,STEREO)
    sky      -y
    right    -4/3*x
    look_at  <10.8452,2.423,0.701839>
    angle    67.3801
    //orthographic
}
```

## Perspective

The default view is a perspective view, which ends up looking like this:

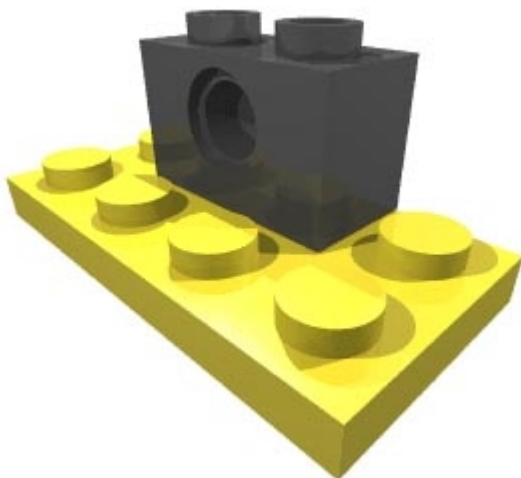


Figure 5. Default, perspective view of the two bricks shown in Figure 3.

This is not really what we want for the building instructions. Instead, I use something called an *orthographic* view. It's not as realistic, but it shows the same piece the same size in different places in the model. In general, it's easier to interpret for building models than a perspective view. (Official Lego instructions use an orthographic view.) To see the orthographic view, just uncomment the `orthographic` keyword by removing the two slashes from its front:

```
camera {
  #declare PCT = 0; // Percentage further away
  #declare STEREO = 0; // Normal view
  //#declare STEREO = degrees(atan2(1,12))/2; // Left view
  //#declare STEREO = -degrees(atan2(1,12))/2; // Right view
  location vaxis_rotate(<55.3377,-33.905,-43.7907> +
    PCT/100.0*<44.4925,-36.328,-44.4925>,
    <-1616.32,-3959.17,1616.32>,STEREO)
  sky      -y
  right    -4/3*x
  look_at  <10.8452,2.423,0.701839>
  angle    67.3801
  orthographic
}
```

Now our model looks like this:

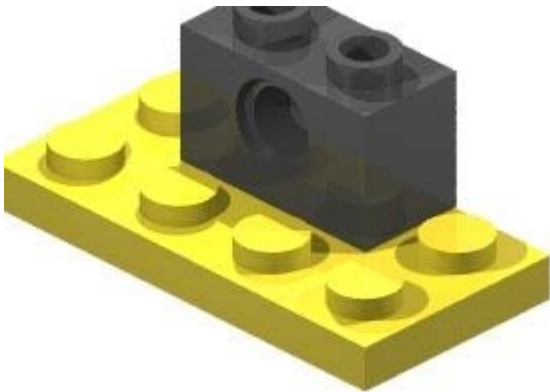


Figure 6. Orthographic view of the same two bricks.

### Camera position

Now we need to monkey with the camera angle so we can see the whole model. The two things we want to change are the `location` and `look_at` keywords. `l3p` automatically sets up `look_at` to point at the center of your model, so you don't usually need to adjust it drastically. I don't really understand the `vaxis_rotate` stuff that `l3p` puts in for `location`, so I generally take it out and substitute straight coordinates, like this:

```
camera {
  location <100, -100, -100>
  sky      -y
  right    -4/3*x
  look_at  <10, 0, 0>
  angle    67.3801
  orthographic
}
```



This produces the output in Figure 7.

Usually I adjust the `location` and `look_at` numbers by trial and error until the model is framed well and displayed at an angle I like. While you're adjusting these things, you should use a small image size setting in POV-Ray so it doesn't take forever to render each frame. To really speed things up while you're fiddling with the camera angle, you might consider changing the `QUAL` setting at the very beginning of the `.pov` file you're working on. Setting `QUAL` to 0 displays every brick as its bounding box, which makes rendering very fast.

Figure 7. Same model after substituting straight coordinates for `vaxis` data. Bricks appear too far away (small).

The above model is too far away, so I will adjust the camera location vector by multiplying it with a scalar. This moves the camera closer or farther from the model without changing any of the angles.

```
camera {
  location .6 * <100, -100, -100>
  sky      -y
  right    -4/3*x
  look_at  <10, 0, 0>
  angle    67.3801
  orthographic
}
```

The results (Figure 8) are satisfactory.

## Lighting

Once you've got the camera angle just the way you want it, it's time to start playing with the lights. In the image above, the lighting is not very good. The forward face of the black brick is not lighted well; it's hard to see the hole. Worse than that, however, there are too many shadows. For building instructions, we want the lighting to be very simple.

l3p creates three lights in your .pov file. These are specified just after the camera section, and look something like this:

```
light_source {
  <0,-61.3755,-51.3755> // Latitude,Longitude,Radius: 45,0,72.656
  color rgb <1,1,1>
}
light_source {
  <54.492,-46.328,31.461> // Latitude,Longitude,Radius: 30,120,72.656
  color rgb <1,1,1>
}
light_source {
  <-31.461,-72.9219,18.164> // Latitude,Longitude,Radius: 60,-120,72.656
  color rgb <1,1,1>
}
```

Basically all you need to do is move the lights until your model is lighted the way you want. Here I've moved them so there's one in front of the model, one on the front side, and one behind and above the hole in the black brick. The `shadowless` keyword is used to specify a light source that illuminates but casts no shadows.

```
light_source {
  <25,-100,-50>
  color rgb <1,1,1>
}
light_source {
  <-50,-50,-50>
  color rgb <1,1,1>
  shadowless
}
light_source {
  <10,-25,25>
  color rgb <1,1,1>
  shadowless
}
```

Figure 9 shows the result of the new lighting.

Lighting is tricky. You will need to practice for a while until you get things the way you want them.

[Next Page](#) ►

**Tools to Save Your MindStorm Models**  
Jonathan Knudsen |

**Creating building instructions**



Figure 8. Same model after changing camera location to bring model "closer."

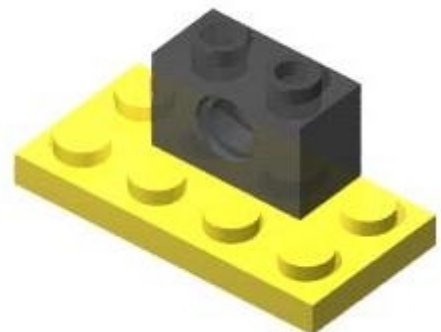


Figure 9. Same model after repositioning lights.

Now you understand how to create a high-quality image of a Lego model. Creating a whole series of instructions is just a tedious repetition of these basic steps. To create the online instructions for the robots from my book (<http://www.oreilly.com/catalog/lmstorms/building/>), for example, I followed these steps:

1. I use MLCAD to create an MPD file for the entire model.
2. Using a text editor, I cut and paste from the original MPD to create a DAT or MPD file representing each step in building the model.
3. I also use a text editor to pull out the parts needed for each step. Then I edit the parts files in MLCAD to lay them out nicely.
4. The step files and parts files are converted to .pov files using l3p.
5. Inside POV-Ray, I edit the camera angle and lights for the completed model until I am happy. Then I copy the camera and lights settings to each step file. I modify the camera and lights slightly and apply the new settings to each of the parts files. For complex models, some steps have to show different views of the model, so these have to be adjusted separately.
6. For certain pieces, like tires or treads, the default materials don't look right. By default, l3p makes all the bricks look shiny, but tires and treads need to look like rubber. For this, I manually create a new material and substitute it in for the tires and treads.
7. I use POV-Ray's file queue to render all of the step and parts files while I go to have lunch.
8. Then I crop each image using Adobe PhotoDeluxe. I would use a real tool like Photoshop or GIMP, but I don't have either one.
9. I add text to the parts images using PhotoDeluxe.
10. I save all the images as JPEGs.
11. Then I use Netscape Composer to create the HTML pages containing the images.

It's a tedious process, but I'm very pleased with the results.

### For more information

This article skims over the top of several deep subjects: part modeling, DAT files, and POV-Ray. If you'd like to learn more, there's plenty of information out there.

- [ldraw.org](http://ldraw.org) is an outstanding collection of Lego model building information. You can find official parts updates, information about the DAT file format (in the FAQ), links to all of the known DAT file editors and utilities, and information about upcoming developments in the Lego CAD community.
- [LUGNET](http://LUGNET) is the center of the online Lego community. It has an entire hierarchy of discussion groups devoted to [Lego CAD issues](http://LUGNET).
- Although ldraw.org has MLCAD available for download, Michael Lachmann maintains the [official MLCAD page](http://www.michael-lachmann.com/mlcad/)
- [l3p has its own homepage](http://www.l3p.com), with helpful tips and examples.
- [POV-Ray has a web site](http://www.povray.com) with lots of information and documentation.

---

[Jonathan Knudsen](http://www.oreilly.com) is an author at O'Reilly & Associates. His books include [The Unofficial Guide to Lego MindStorms Robots](http://www.oreilly.com/catalog/linform/), [Java 2D Graphics](http://www.oreilly.com/catalog/linform/), and [Java Cryptography](http://www.oreilly.com/catalog/linform/).

---

Discuss this article in the O'Reilly Network [MindStorms Forum](http://www.oreilly.com).

Return to the O'Reilly Network [Hub](http://www.oreilly.com).

oreillynet.com Copyright © 2000 O'Reilly & Associates, Inc.



O'Reilly Developer Quiz:

XML =



Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)  
<http://www.oreillynet.com/pub/a/network/2000/05/22/LegoMindstorms.html>

## The Straight and Narrow

by [Jonathan Knudsen](#)  
05/23/2000

When I was a kid, my brother and I used to make up treasure maps. You'd have to start in a particular place in the backyard, then walk ten steps forward, turn left, walk twenty steps, and so forth. This wasn't a very reliable way to find treasure -- one person's steps are a different size than another person's steps, and it's hard to tell if you're moving exactly in a straight line or if you've turned exactly at right angles to your original path.

It turns out robots have a lot of the same trouble that my brother and I had. In this article, we'll examine one of these problems, moving in a straight line, in detail. Afterwards, I'll take a step back and look at the bigger picture of robot navigation.

### Differential Drive

One of the easiest ways to build a robot is using a technique called *differential drive*. This means that the robot has two motors, each of which controls a wheel or tread on one side of the robot. The tank-style robot is a classic example. If you run both motors forward, the whole robot moves forward. Run both motors in reverse, and the robot backs up. If you run the motors in opposite directions, the robot spins in place, just like a tank or a bulldozer. The following picture shows such a robot:



A simple variation on this idea uses wheels instead of treads, as shown here:



Differential drive robots with wheels are a little more difficult to create than tank-style robots, simply because the robot needs



some way to keep its balance. The robot shown above has an *idler wheel* at the back for support. The idler wheel has an unrewarding job, which is to keep the robot from falling over while not interfering with the movement dictated by the drive wheels. It swivels freely on a vertical axis, like the wheels on the front of a shopping cart.

Some wheeled differential drive robots have two idler wheels, one at the front and one in the back. Some don't have idler wheels at all, but a couple of supports that slide over the ground as the robot moves.

An aside: Don't confuse *differential drive* with a *differential*. Differential drive is a style of locomotion; a differential is a clever mechanical device that allows movement to be shared between two shafts. As you'll see, a differential is crucial in the construction of car-style robots.

## The Bad News

Differential drive is great: easy to construct, easy to understand, easy to program. But the one thing your robot can't do is drive in a straight line. Sure, if you run both motors forward, the robot moves forward, sort of. But if you observe your robot, you'll notice it moves in a gentle arc. Why is this? You've used the same motors and the same parts on both sides of your robot, right?

In the real world of robotics, there are hardly ever two things that are exactly the same. Let's look at just a few of the reasons your robot won't go straight:

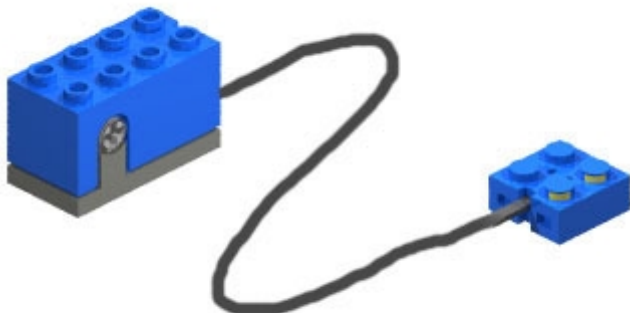
1. When the robot moves forward, the motors are actually running in opposite directions. To prove this to yourself, look at each motor by facing its shaft. When the robot moves forward, one of the motors is spinning clockwise and one of them is spinning counterclockwise. Although this might not seem like a big deal, most electric motors are optimized for one direction or the other. Because the motors run in opposite directions, one of them is actually running a little more efficiently than the other, which can cause a curve in the robot's forward motion.
2. The treads or tires may be slightly different sizes. Even if the motors ran at exactly the same speed, variations in the treads or tires would cause deviations from an absolutely straight course.
3. The robot may not be driving on a completely uniform (or even flat) surface. If one tread of your tank-style robot is running on a hardwood floor, and one on a carpet, the robot will not move in a straight line.

What's a robot designer to do? One way to deal with the straight line problem is to examine the progress of the robot and make corrections. In engineering this is called *feedback control*.

## Using Feedback

The next logical step would be to use a sensor to tell if the robot is turning or moving straight. One way to do this is to measure how far each wheel or tread has turned, which is a way of measuring how far each side of the robot has traveled. (I'm ignoring some assumptions here that will come back and bite us in the butt later.) The simplest way to measure the rotation of the robot's wheels or treads is to add a rotation sensor to the drive shaft of each side. The rotation sensor does not come with the Robotics Invention System or any of the other Mindstorms kits. However, you can purchase these sensors for \$16.50 each (postage included!) from the Lego Shop-At-Home service at 800-453-4652. The item number is 9756.

The sensor itself looks like a Lego brick with a shaft hole, as shown below. You can use this sensor to measure how far a shaft in the hole has rotated. The programming environment that comes with the Robotics Invention System, the RCX, measures rotations in the clockwise or counterclockwise directions, with 16 units per full revolution. For example, rotate the shaft one way by a full turn and the sensor will read 16. Rotate it the other way by two full turns and the sensor will read -16.



After attaching two rotation sensors to your robot, one for each side, you could use the following algorithm to keep the robot moving mostly straight:

```

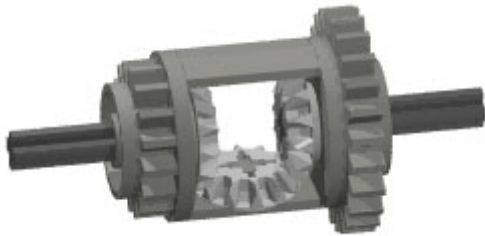
while (true) {
  if (left rotation sensor is greater than right rotation sensor)
    turn slightly left
  else if (left rotation sensor is less than right rotation sensor)
    turn slightly right
  else if (left rotation sensor equals right rotation sensor)
    full power to left and right motors
}

```

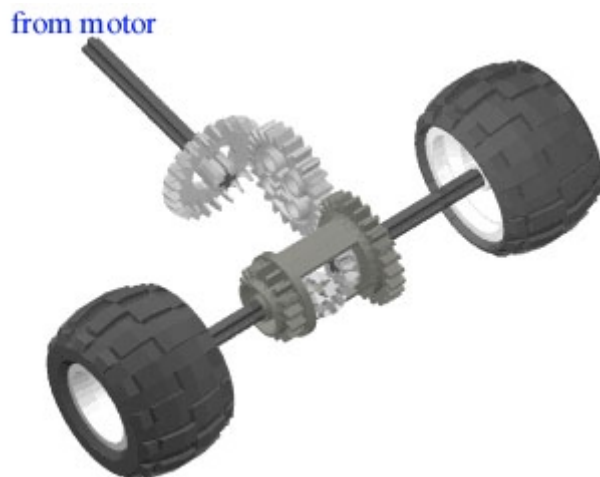
## Introducing the Differential

While the above solution would work, it costs \$33.00 and it sucks up two of the inputs on the RCX. What if we could just use one rotation sensor to measure the difference between the right and left shaft rotations? Then we'd only have to buy one rotation sensor and use up one of our precious inputs on the RCX.

This is indeed possible -- all we have to do is find a way to mechanically combine the rotations from the left and right motors. I already mentioned the solution, the *differential*. The best way to understand a differential is to hold it in your hands and play with it. Lego's differential piece looks like a cutout cylinder with a gear at either end. To fully assemble it, use three 12t gears and a couple of shafts, as shown here:



The differential can split or combine motion. In a rear-wheel-drive car, the differential takes the motion from the motor and shares it between the two back wheels. In this scenario, the engine would be driving the differential itself (the dark gray piece) on one of the sides where it has gear teeth. The wheels of the car would be attached to the shafts coming out of the sides of the differential, as shown:



When a car turns corners, the rear wheels don't move at the same speed. To understand this, think of the distances the wheels have to travel. The outside wheel has to travel through a greater arc in the same time the inner wheel travels through a smaller arc. Thus, the outer wheel must be moving faster than the inner wheel.

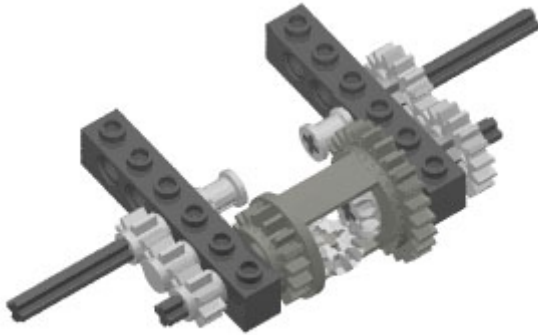
The differential takes care of splitting the engine's driving power between the two wheels. If you just had a straight axle connecting the two rear wheels, you'd have trouble when the car was turning -- either one of the wheels would have to slip, or the axle would have to break.

## A Smarter Solution

To make our robot drive straight, we'll use the differential in a slightly different way. We'll take the motion from both motors

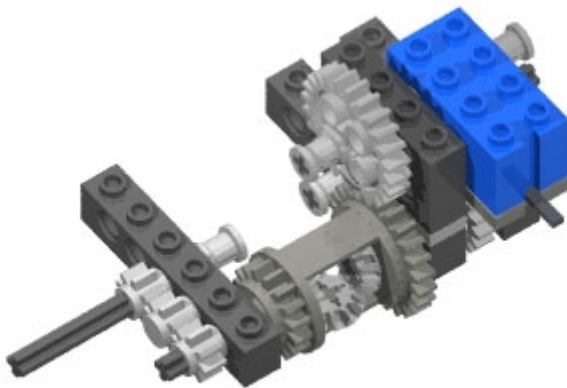
and feed it into the sides of the differential. If the side axles are rotating at the same speed, in opposite directions, the main gears of the differential (the dark gray piece) will not move at all. You can try this with the assembled differential. Just rotate the side axles at about the same speed; the differential itself won't rotate at all. We can attach the differential to a rotation sensor. If the rotation sensor registers any change, we'll know that we're off course.

The gearing is a little tricky, but not impossible. First, we need to invert the direction of one of the motor outputs so the outputs are moving in opposite directions when they enter the differential assembly. To do this, we'll use two 16t gears on one side and three 8t gears on the other. These two combinations of gears take up the same amount of space, but invert the rotation direction on one side. This is what it looks like:

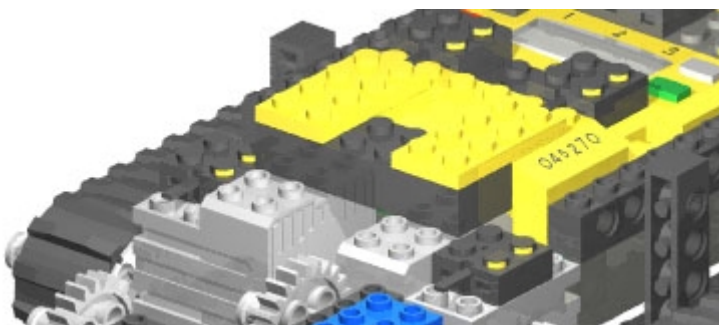


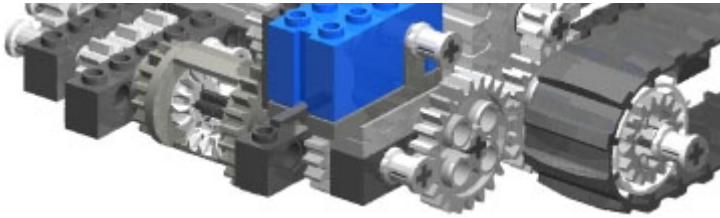
The shafts sticking out on the sides will be linked to the drive motors.

To measure the difference between the rotation of the two drive shafts, we need to attach the rotation sensor to the differential, as shown here:



Finally, we'll tack the whole thing on the back of a tank-style robot, adding a couple of gears to link things to the drive motors. The following illustration shows how the rotation sensor and differential can be added to the back of RoboTag, one of the projects from [The Unofficial Guide to LEGO MINDSTORMS Robots](#). Building instructions for RoboTag are [available online](#), although you'll need to deviate from the instructions to add the differential and rotation sensor assembly.





## Programming

Once we've got the rotation sensor mechanically attached to the two drive shafts, the programming is pretty straightforward. Here is a program in NQC (Not Quite C) that examines the rotation sensor as the robot drives forward. If it deviates from 0, the robot turns left or right to adjust its course.

```
task main() {
  // Configure input 3 as a rotation sensor.
  SetSensor(SENSOR_3, SENSOR_ROTATION);
  // Start moving forward.
  OnFwd(OUT_A + OUT_C);
  // Reset the rotation counter.
  ClearSensor(SENSOR_3);
  // Adjust course if necessary.
  int angle;
  while (true) {
    angle = SENSOR_3;
    if (angle < 0) {
      // Turn right.
      On(OUT_A);
      Off(OUT_C);
    }
    else if (angle > 0) {
      // Turn left.
      Off(OUT_A);
      On(OUT_C);
    }
    else On(OUT_A + OUT_C);
  }
}
```

The call to `ClearSensor()` sets the rotation count to 0. This should be done every time the robot starts moving forward.

An interesting side effect of our differential and rotation sensor setup is that we can measure turns with a fair degree of accuracy. When the motors turn in opposite directions to spin the robot, the rotation sensor will measure how far the robot is turning (although it cannot account for the differences in the motors or treads, which is the purpose when the robot moves forward).

## About Dead Reckoning

Having gone through all the trouble of attaching a differential and a rotation sensor to our robot, it's now time to step back and look at the larger question. Why would you want to move in a straight line to begin with? Moving in a straight line is important in a navigational technique called *dead reckoning*. In dead reckoning, you start from a known point. Then you move in a certain direction, and, based on how fast you're moving and in what direction, you can calculate a new location. When using dead reckoning in ships, it's prudent to periodically use celestial navigation to more accurately determine your position, then use dead reckoning for the next leg of your journey. Or, more likely, you'll probably just bring along a GPS receiver to pinpoint your position.

Lego robots, however, are unlikely to use celestial navigation (or GPS receivers) and thus would have to rely on dead reckoning alone. Dead reckoning is impractical, over time, because there are so many things that can go wrong. One of the treads or wheels of the robot might slip over the surface on which the robot is driving. Even using our rotation sensor feedback, we have no way of knowing whether the robot wheels or treads are actually driving or just slipping. If one side is gripping and the other is slipping, the robot's direction can change radically without the robot being aware of it. If both treads are slipping, the robot will believe it is moving when it really isn't. With dead reckoning, small errors become large errors over time, which is why navigators on ships synchronize with the stars now and then.

If you can use an alternative to dead reckoning, it's definitely worth considering. Can your robot locate a light beacon, or follow a wall until it bumps into its destination? Can the robot follow a line on the floor to get where it's going? Or perhaps dead reckoning is a reasonable choice, as long as the robot has a chance to accurately determine its location periodically.

## Bring It on Home

The best approach is to combine navigational techniques. Your robot might have a home base, for example, that could act as a navigational reference point. Starting from this known location, it could seek out other destinations using dead reckoning based on the feedback from the rotation sensor. The home base might be a bright light or a mechanical dock of some sort. To go back home, you could again use dead reckoning, then home in on the light at the dock to correct for navigational errors. After all, we only had to use one of the RCX's input ports to make the robot go straight. We can certainly spare another input port for a light sensor, to find the home base.

Sensor feedback is the key to success in robot navigation. Appropriately placed sensors can tell your robot if it's where it wants to be or not. With feedback control, a robot can find treasure much more accurately than my brother or I.

---

Read more from [Lego Mindstorms](#).

Discuss this article in the Mindstorms [forum](#).

Return to the [O'Reilly Network](#).

**oreillynet.com** Copyright © 2000 O'Reilly & Associates, Inc.