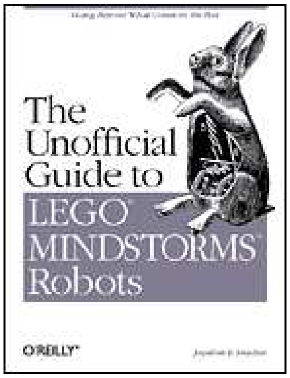
O'REILLY Online Catalog

PRODUCT INDEX
SEARCH THE CATALOG



LEGO®MINDSTORMS(TM) Robots

By Jonathan B. Knudsen
1st Edition, October 1999
1-56592-692-7, Order Number: 6927
250 pages, $24.95

Chapter 4: Not Quite C

In this chapter:

Once you've written a few programs in RCX Code, it gets a little constrictive. For one thing, there aren't any variables. It would be nice if your robot could remember things, like how many obstacles it's encountered or what the temperature was three minutes ago. In RCX Code, the only thing remotely resembling a variable is the counter. Back in *Chapter 3, Trusty, a Line Follower*, I used the counter to remember which way to turn. The counter was always a 0 or a 1, indicating whether the robot should turn left or right. But it's tough going, and if you try anything more complicated, the counter is not going to get the job done.

The lack of variables is only one of the limitations of RCX Code. Two other important limitations are:

- Although you can define subroutines (called "My Commands"), you can't call one subroutine from another. Likewise, a subroutine can't call itself.
- You can't control the RCX's display. It would be really nice, for debugging purposes, to be able to show values on the display.

The LEGO Group aimed the Robotics Invention System at people who had never programmed before. For this group, RCX Code is a gentle way to get started with programming mobile robots. But RIS is exceedingly popular with programmers and other technically savvy people, who are frustrated by RCX Code's limitations. If you're reading this chapter, this probably includes you.

Since RIS was released in the fall of 1998, the MINDSTORMS community has produced an amazing stream of clever, innovative software designed to overcome the limitations of RCX Code. Most of this software is available, free of charge, on the Internet. In this chapter, I'll describe one of the most popular packages, Not Quite C (NQC). NQC allows you to write programs for your RCX with a text-based language. I'll describe the syntax and commands of this language, with copious examples. If you've programmed in C, NQC will look familiar. If you have never programmed in C, don't worry; NQC is easy to learn.

This chapter presents NQC in four steps:

1. To get you started with NQC, this chapter begins with a simple example.
2. To understand how NQC works, you need to understand the software that's running on the

RCX. This chapter describes the important pieces of the RCX's software architecture.
3. A detailed listing of NQC's commands is also presented, with examples.
4. Finally, this chapter contains software for Trusty written in NQC.

A Quick Start

Let's get right to the good stuff with a working example. First, you'll need to download and install NQC. It's available for MacOS, Linux, and Windows. Navigate to the NQC web site (http://www.enteract.com/~dbaum/lego/nqc/) and follow the instructions to download and install the latest version. The examples in this book were written with the NQC version 2.0b1.

Once it's installed, enter the following program using a text editor. This program operates Hank, the robot from *Chapter 2, Hank, the Bumper Tank*. Save the program in a file called *Hank.nqc*.

```
#define BACK_TIME 50
#define TURN_TIME 80

task main() {
  SetSensor(SENSOR_1, SENSOR_TOUCH);
  SetSensor(SENSOR_3, SENSOR_TOUCH);
  OnFwd(OUT_A + OUT_C);
  while (true) {
    if (SENSOR_1 == 1) {
      PlayTone(440, 50);
      OnRev(OUT_A + OUT_C);
      Wait(BACK_TIME);
      OnFwd(OUT_A);
      Wait(TURN_TIME);
      OnFwd(OUT_C);
    }
    if (SENSOR_3 == 1) {
      PlayTone(880, 50);
      OnRev(OUT_A + OUT_C);
      Wait(BACK_TIME);
      OnFwd(OUT_C);
      Wait(TURN_TIME);
      OnFwd(OUT_A);
    }
  }
}
```

Now compile the source code using the `nqc` command:

```
C:\>nqc Hank.nqc

C:\>
```

If you made a mistake typing in the program, `nqc` gives you a list of errors. Otherwise, you're ready to download the program with the `-d` option:

```
C:\>nqc -d Hank.nqc
Downloading Program:.....complete

C:\>
```

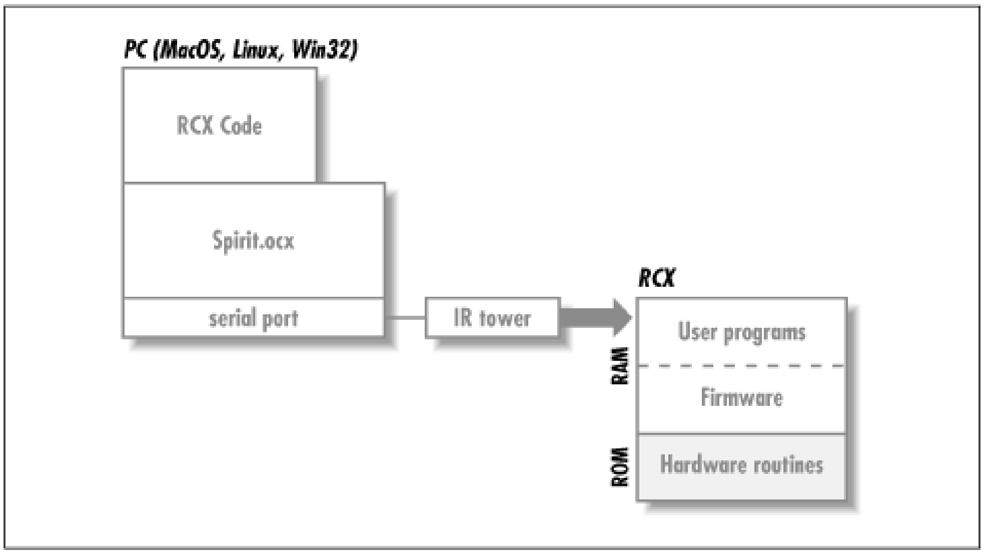
(If you need to specify a serial port different from the default, use the `-s` option.)

Go ahead and run the program. When you're done playing, come back and get some background on the software that runs the RCX.

RCX Software Architecture

Writing a program for the RCX involves a handful of software layers, both on the development PC and on the RCX itself. Figure 4-1 shows an overview of the important pieces.

Figure 4-1.RCX software architecture



ROM

The RCX is a small computer based on a Hitachi H8/3292 microcontroller. The RCX contains two kinds of memory: Read Only Memory (ROM) and Random Access Memory (RAM). As its name implies, ROM cannot be written. It is programmed at the factory and cannot be changed. RAM, on the other hand, can be written and read as many times as you want, with one catch: it needs power. If you take the batteries out of your RCX, the contents of the RAM are erased. Under normal circumstances, however, the batteries preserve the contents of the RAM.

When you first got your RCX, it had some stuff in ROM and an empty RAM. The RCX's ROM routines know a little bit about the RCX's hardware. These routines can run the motors or access the sensors. Most importantly, the ROM routines know how to receive code from the IR port and place it in RAM.

Firmware

One of the first things you had to do with your RCX was download the *firmware*. The firmware is, essentially, an operating system for your RCX. The routines in ROM know how to download a set of firmware from the IR port and store it in RAM.

The firmware is more capable than just the ROM routines. It shows a clock on the display of the RCX; it can recognize and respond to the **View** button. Most importantly, it can receive robot programs over the IR port and run them.

Although at first it sounds like the firmware and the robot programs are the same kind of animal, this is not the case. The firmware is actual Hitachi H8 machine code. Together with the H8 machine code in ROM, the firmware defines an operating system for the RCX. It provides access to the RCX's inputs and outputs. It also provides a way for programs to be downloaded, stored, started, and stopped.

The actual robot programs are not H8 machine code. They are defined at a higher level called *bytecode*. Whereas the H8 machine instructions are very rudimentary, like "move this value to register 1," bytecode instructions are more powerful, like "turn output 2 on with full power." The firmware interprets the bytecode and performs the appropriate action.

Spruce Up That Résumé

Writing programs for the RCX is an example of *cross-compiling* for an *embedded system*, a phrase that is bound to sound good on your résumé. Cross-compiling means that you are writing programs one one computer (your PC) that will run on another computer (the RCX). An embedded system is a computer that is part of some other device. Microwave ovens and mobile phones both contain embedded systems.

The usual way to develop software for a chip like the Hitachi H8 would be to use a cross-compiler running on a PC. You would write source code (probably in C or assembly language) on your PC and use the cross-compiler to create H8 machine code from the source. Then you would probably use a special PC peripheral, a *burner*, to place the machine code on some sort of programmable memory chip.

The final step would be to physically place the memory chip in the embedded system. When the embedded system boots up, the software you just wrote will be run.

The RIS software simplifies this process in two important ways. First, it provides a graphic programming environment that's very accessible, particularly for people who haven't programmed before. Second, programs are downloaded to the RCX over the IR link and stored in RAM. This eliminates the complexity of dealing with memory chips and burners yourself.

The cross-compilation step is a little different, too, because RCX programs are bytecode rather than machine code. But it's still cross-compilation: the end result is just bytecode rather than H8 machine code.

About Spirit.ocx

On the PC side, an ActiveX control called *Spirit.ocx* handles interaction with the RCX via the IR link. *Spirit.ocx* can execute bytecode commands on the RCX, download new programs to the RCX, download firmware to the RCX, and receive data from the RCX.

The RCX Code programming environment sits on top of *Spirit.ocx*. It provides the click-and-drag programming environment that you're already familiar with. RCX Code converts these graphic programs into bytecode and uses *Spirit.ocx* to download the programs to the RCX.

Spirit.ocx is a regular ActiveX control, which means its functions are accessible from programming languages like Visual Basic and Visual C++.

A Day in the Life of a Program

Let's examine the typical life of a robot program:

1. The program's life begins when you create something in RCX Code. RCX Code compiles your program to bytecode.
2. RCX Code uses *Spirit.ocx* to download the program to one of the RCX's five program slots. The compiled bytecode is transferred to the RCX via the IR link.
3. The program is now available in RAM. When you run it, the firmware interprets the bytecode in your program and performs the appropriate tasks.

NQC Overview

Where does NQC fit in? NQC is a replacement for the software on the PC, both RCX Code and *Spirit.ocx*. NQC source code is stored in simple text files, just like C, C++, or Java source code. NQC compiles these source files to bytecode and can download them to the RCX using the IR tower. NQC is a good way to overcome the limitations of RCX Code. But because it produces bytecode programs, it's still subject to the limitations of the firmware's bytecode interpreter.

Because NQC talks to the IR tower directly, without depending on *Spirit.ocx*, it is very portable. NQC runs on MacOS (using MPW), Linux, and of course Windows 95, 98, and NT. RCX Code, by contrast, only runs on Windows.

NQC was developed by Dave Baum, who maintains the official web site at http://www.enteract.com/~dbaum/lego/nqc/. His web site also includes pithy documentation for the language.

If you're using NQC on Windows, you might want to also use RCX Command Center (RcxCC). RcxCC is a Windows application that wraps around NQC. It provides a syntax-colored program editor, push-button compilation and downloading, real-time control of the RCX, and a host of other useful features. Although NQC is fairly easy to use all by itself, RcxCC gives you an even smoother ride. See the Online Resources at the end of this chapter for a URL and more information.

This chapter covers the important commands of NQC. If you have a background in C programming, the syntax and control structures will look familiar. If you don't have a background in C, don't worry: NQC is easy to learn. I've included lots of example programs to demonstrate how things work. I won't cover NQC exhaustively; several excellent web pages detail the entire language. See the Online Resources section at the end of this chapter for details.

main

NQC programs are organized into one or more *tasks*. A task is simply some set of instructions that execute in order. A task is analagous to a *thread* in other programming environments. A single program may consist of several tasks that execute at the same time.

Tasks have names. Every program should have a special task called `main`. When the **Run** button is pressed, the RCX begins the program by running `main`. If you define other tasks, you have to explicitly start and stop them. The `main` task is the only one that is automatically run by the RCX. I'll explain more about starting and stopping tasks later.

Output Commands

NQC includes several commands for controlling the outputs of the RCX. You've already seen one of these, `On`, in our simple *hello.nqc* example.

On(const outputs)
: This command turns on the specified outputs. The outputs should be some combination of the constant values `OUT_A`, `OUT_B`, and `OUT_C`. Multiple outputs can be specified by adding them together, as shown in the first example. When an output is turned on, its current power and direction are consulted to determine what actually happens.

Off(const outputs)
: This command turns off the specified outputs, putting them in brake mode. For motors, this means that the motor shaft will be hard to turn.

Float(const outputs)
: `Float()` is really a variation on `Off()`. No power is sent to the output, but the shaft of an attached motor will turn freely. This is a useful option if you want your robot to coast to a stop.

You can set the direction of outputs with the following three commands:

Fwd(const outputs)
: Use this command to set the direction of the specified outputs to forward.

Rev(const outputs)
: This command sets the direction of the specified outputs to reverse.

Toggle(const outputs)
: To switch the direction of one or more outputs, use this command.

To determine the output power, use the following command:

SetPower(const outputs, expression speed)
: This command sets the power of the given outputs. Any expression that evaluates to a value from one to seven can be used as the `speed`. You may use the constant values `OUT_LOW` (1), `OUT_HALF` (4), and `OUT_FULL` (7) if you desire.

To fully determine an output's actions, you should set its mode, direction, and power explicitly. By default, all three outputs are set to full power and the forward direction. In *hello.nqc*, therefore, calling `On()` was enough to get the motors running.

NQC provides two handy "combination" commands:

OnFwd(const outputs)
: This command turns on the specified outputs in the forward direction.

OnRev(const outputs)
: This command turns on the specified outputs in the reverse direction.

These commands set the mode and direction of the outputs in one fell swoop, but you should still set the power level explicitly with a call to `SetPower()`.

For timed actions, the following command will come in handy:

OnFor(const outputs, expression time)
: This command turns on the specified outputs for the given time, measured in hundreths of a second. Then the given outputs are turned off (in brake mode, not in float mode).

The following example runs outputs A and C forward, waits one second, then reverses outputs A and C. After another second, the outputs are turned off. (I'm jumping ahead a little by using the `Sleep()` command. Just take my word that it pauses the program for the given number of hundreths of a second.)

```
task main() {
  SetPower(OUT_A + OUT_C, OUT_HALF);
  Fwd(OUT_A + OUT_C);
  OnFor(OUT_A + OUT_C, 100);
  Rev(OUT_A + OUT_C);
  OnFor(OUT_A + OUT_C, 100);
}
```

The `On()`, `Off()`, `Float()`, `Fwd()`, `Rev()`, and `Toggle()` commands are really shorthand for these lower level output commands:

SetOutput(const outputs, const mode)
: This command sets the mode for the given outputs. The outputs are specified in the same way as in the `Fwd()` and `Rev()` commands. The value of `mode` should be one of the constants `OUT_ON`, `OUT_OFF`, and `OUT_FLOAT`.

SetDirection(const outputs, const direction)
: This command determines the direction of the supplied outputs. The `direction` parameter should be `OUT_FWD`, `OUT_REV`, or `OUT_TOGGLE`. `OUT_TOGGLE` is a special value that sets the direction of the output to the opposite of its current value.

In general, I recommend you don't call `SetDirection()` with the `OUT_TOGGLE` value. If you explicitly set the directions of your outputs, your program will be clearer. Furthermore, in programs with more than one task, your program is more likely to behave as you expect.

Input Commands

Before you can read a value from one of the RCX's inputs, you need to tell the RCX what type of sensor is attached to the input. NQC provides a command that does just this:

SetSensor(expression sensor, const configuration)
: This command tells the RCX how to configure the given input. Valid values for `sensor` are `SENSOR_1`, `SENSOR_2`, and `SENSOR_3`, which represent the three inputs of the RCX. The sensor configurations are detailed in Table 4-1.

Table 4-1:NQC Sensor Modes

Configuration	Sensor Type	Input Value	ClearSensor()
`SENSOR_TOUCH`	Touch	1 (pressed) or 0 (not pressed)	-
`SENSOR_LIGHT`	Light	0 (dark) to 100 (bright)	-
`SENSOR_ROTATION`	Rotation[1]	16 units per full rotation	yes
`SENSOR_CELSIUS`	Temperature	Celsius degrees times 10	-
`SENSOR_FAHRENHEIT`	Temperature	Fahrenheit degrees times 10	-
`SENSOR_PULSE`	Touch	Count of presses	yes
`SENSOR_EDGE`	Touch	Count of state transitions	yes

[1] See *Appendix A, Parts and Sources*, which lists the sensors that are available for the RCX.

The actual sensor value can be read using `SENSOR_1`, `SENSOR_2`, and `SENSOR_3`. These are shorthand for the following command:

SensorValue(const input)

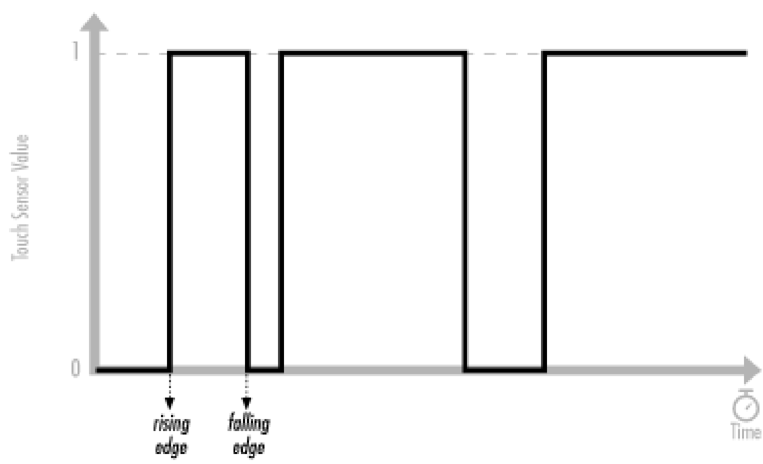
This command returns the current value of the given input, which should be 0, 1, or 2, for Input 1, Input 2, and Input 3 respectively. The values returned from an input depend on the input's configuration and are described in Table 4-1.

TIP: `SENSOR_1`, `SENSOR_2`, and `SENSOR_3` actually have a dual purpose in life. Their first purpose is to identify the inputs on the RCX to commands like `SetSensor()`. The second purpose is to retrieve values from the inputs. Thus, there are two distinct uses for `SENSOR_1`, `SENSOR_2`, and `SENSOR_3`.

The `SENSOR_PULSE` and `SENSOR_EDGE` configurations are variations on `SENSOR_TOUCH`. The `SENSOR_PULSE` configuration counts the times the touch sensor has been pressed, while `SENSOR_EDGE` counts the transitions from on to off and from off to on. When you read the value of an input in one of these configurations, the input value is the accumulated count.

Edges and Pulses

If you examine the output of a touch sensor, over time, it looks something like this:



The transitions from 0 to 1 and from 1 to 0 are called *edges*. A transition from 0 to 1 is a rising edge, while a transition from 1 to 0 is a falling edge.

The `SENSOR_EDGE` configuration counts all edges, rising or falling. `SENSOR_PULSE` is a little more selective `--` it counts rising edges only.

The configurations that keep a count can be reset with a call to `ClearSensor()`, as shown in Table

4-1.

ClearSensor(expression sensor)
: This command resets the current count for the given input to 0.

The following example plays a sound after every four press on the touch sensor. It begins by configuring Input 1 to count touch presses with the `SENSOR_PULSE` configuration. Then it enters an endless loop, repeatedly testing the value of Input 1. If it is 4, a sound is played and the count for Input 1 is reset.

```
task main() {
  SetSensor(SENSOR_1, SENSOR_PULSE);
  while(true) {
    if (SENSOR_1 == 4) {
      PlaySound(SOUND_DOWN);
      ClearSensor(SENSOR_1);
    }
  }
}
```

The `SetSensor()` command actually configures an input's *type* and *mode* at the same time. The input type describes the electrical characteristics of the attached sensor, while the mode determines how the sensor values are interpreted. If you need finer control over the inputs than you can get from `SetSensor()`, use the `SetSensorType()` and `SetSensorMode()` commands.

SetSensorType(expression sensor, const type)
: This command specifies the type of sensor attached to the given input. Input types are listed in Table 4-2. This command specifies how the RCX should treat an input, electrically speaking. The `SENSOR_TYPE_LIGHT` type, for example, supplies power to the sensor. (I described this back in Chapter 3.)

Table 4-2:Input type constants

Type Constant	Sensor type
`SENSOR_TYPE_TOUCH`	Touch sensor
`SENSOR_TYPE_TEMPERATURE`	Temperature sensor
`SENSOR_TYPE_LIGHT`	Light sensor (powered)
`SENSOR_TYPE_ROTATION`	Rotation sensor (powered)

SetSensorMode(expression sensor, const mode)
: Use this command to set the mode of the given input. While the `SetSensorType()` command is used to specify the electrical characteristics of the input, the `SetSensorMode()` command specifies how the input value should be processed. The modes are listed in Table 4-3.

Table 4-3:Input mode constants

Mode Constant	Description
`SENSOR_MODE_RAW`	Raw sensor value from 0 to 1023
`SENSOR_MODE_BOOL`	Either 1 or 0
`SENSOR_MODE_EDGE`	Counts transitions from 1 to 0 and vice versa
`SENSOR_MODE_PULSE`	Counts transitions from 1 to 0
`SENSOR_MODE_PERCENT`	Percent from 0 to 100
`SENSOR_MODE_CELSIUS`	Celsius temperature
`SENSOR_MODE_FAHRENHEIT`	Fahrenheit temperature
`SENSOR_MODE_ROTATION`	Shaft angle, 16 counts per full revolution

Internally, input values initially have a raw value from 0 to 1023. Raw values are converted to the input values your program sees by a process that depends on the input mode. Three of the modes count events: `SENSOR_MODE_EDGE`, `SENSOR_MODE_PULSE`, and `SENSOR_MODE_ROTATION`. The other modes perform a mathematical scaling operation on the raw input value. For example, if the input mode is `SENSOR_MODE_PERCENT`, the RCX converts the raw value into a percent according to the following equation:.

If you wanted to attach a temperature sensor to Input 2 and measure Celsius values, you would do the following:

```
SetSensorType(SENSOR_2, SENSOR_TYPE_TEMPERATURE);
SetSensorMode(SENSOR_2, SENSOR_MODE_CELSIUS);
```

The `SetSensor()` command, which I described first, is a convenient way of specifying both an input type and an input mode. Table 4-4 shows what types and modes correspond to the configurations that `SetSensor()` recognizes.

Table 4-4:Input configurations, types, and modes

Input Configuration	Input Type	Input Mode
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSOR_MODE_EDGE
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_FAHRENHEIT	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_FAHRENHEIT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION

Timers

The RCX has four internal timers, numbered 0, 1, 2, and 3. They count in increments of 100 ms, or once every 1/10 second. NQC includes two commands for interacting with the timers:

Timer(const n)
: This returns the value of the specified timer, which should be 0, 1, 2, or 3. The number returned is the number of 1/10 second since the timer was cleared.

ClearTimer(const n)
: This command resets the value of the given timer to 0. The timer begins counting up again immediately.

Random Numbers

NQC has a simple command for creating random numbers. Random numbers are often useful in robot programming. For example, a robot that tries to drive around obstacles can easily get stuck in a corner if it always backs up and turns exactly the same way to get away from an obstacle. A robot that backs up for a random amount of time and turns for a random amount of time is less likely to get stuck in this way.

Random(const n)
: This command returns a random number between 0 and `n`.

Program Flow

You've seen how to control the RCX's outputs and inputs. But robot programs aren't very interesting unless they can make decisions and repeat actions. In this section, I'll sketch out NQC's program control commands. NQC supports a standard set of conditional branches and loops; if you've ever programmed in other languages (particularly C), this will look familiar.

Waiting

Although it might not seem important, NQC includes a command that tells the robot to do nothing for a certain amount of time. This is often useful if you need to allow some time for something to happen -- maybe the robot needs to move forward or turn for a little while, or you want to give a sound time to play.

Wait(expression ticks)
: This command causes the current task to pause for the supplied hundreths of a second; a call to `Wait(100)` will pause the task for a full second. Note that this only applies to the current task -- other tasks will continue to execute. I'll talk more about tasks a little later.

A variation on this theme is the concept of waiting for an event, like a press on a touch sensor, or a certain time of day. The following command waits for a condition to become true:

until (boolean condition) [statements]
: Use this command to wait for the given condition to become true. You could, for example, wait for the value of input 1 to become 4 like this:

```
until (SENSOR_1 == 4);
```

: This particular `until` has an empty body, which means it won't do anything each time the condition is tested -- it simply waits until the condition is true. The following program beeps every half second until you press a touch sensor on Input 1 four times:

```
task main() { SetSensor(SENSOR_1, SENSOR_PULSE); until (SENSOR_1 == 4) {
PlaySound(SOUND_CLICK); Wait(50); } }
```

Loops

A loop is a series of commands that you want to be executed repeatedly. NQC offers three flavors of loop:

repeat (expression value) [statements]
: This command simply repeats the given statements `value` times.

while (boolean condition) [statements]
: This loop repeats the supplied statements until `condition` is no longer true.

do [statements] while (boolean condition)
: This loop is similar to `while` but the statements are executed before the `condition` is tested. The statements will always be executed at least once, which is not true for a `while` loop.

Let's look at an example. The following code plays a sound every half second while a light sensor attached to Input 3 sees dark:

```
while (SENSOR_3 < 35) {
  PlaySound(0);
  Wait(50);
}
```

Notice how curly braces are used to bracket the statements that belong to the `while` loop. If you have only one command in the body of the `while`, you can omit the braces like this:

```
while (SENSOR_3 < 35)
  Wait(50);
```

Conditionals

To test a condition, use the `if` command.

if (boolean condition) [statements]
: This command executes the given statements only if `condition` is true.

if (boolean condition) [statements] else [statements]
: This is a simple variation on the basic `if` command. If the condition is false, the statements after the `else` are executed.

The following example turns different directions depending on the value of Input 2:

```
SetPower(OUT_A + OUT_C, OUT_FULL);
if (SENSOR_2 < 50) {
  Fwd(OUT_A);
```

```
  Rev(OUT_C);
}
else {
  Rev(OUT_A);
  Fwd(OUT_C);
}
On(OUT_A + OUT_C);
```

Using #define for Constants and Macros

Constant values can be assigned meaningful names using `#define`. This is a idiom that will be familiar to C programmers. Here is an example:

```
#define POWER 5

task main() {
  SetPower(OUT_A + OUT_C, POWER);
  On(OUT_A + OUT_C);
}
```

NQC replaces every occurrence of `POWER` with `5` in your source code before compiling it. Although this may not seem like a big deal, it is; `#define` lets you create readable names for things that might otherwise be cryptic. It also lets you define things that might need to be adjusted throughout your program in one place. Your program, for example, might have multiple places where it set the outputs to power level 5. Instead of explicitly putting 5 all the way through your program, you can use the constant value `POWER`. If you later decide you want the power level to be 7, you just have to change the definition of `POWER`, instead of finding all the places in your program where the output power is set.

You can also create *macros* with `#define`. A macro is a kind of miniature program. Usually you'll define a macro for something you want to do frequently. The following program uses three macros.

```
#define forward(power) \
    SetPower(OUT_A + OUT_C, power); \
    OnFwd(OUT_A + OUT_C);
#define left(power) \
    SetPower(OUT_A + OUT_C, power); \
    OnRev(OUT_A); OnFwd(OUT_C);
#define right(power) \
    SetPower(OUT_A + OUT_C, power); \
    OnFwd(OUT_A); OnRev(OUT_C);

task main() {
  forward(OUT_FULL);
  Wait(100);
  left(OUT_HALF);
  Wait(100);
  right(OUT_HALF);
  Wait(100);
  Off(OUT_A + OUT_C);
}
```

The preceding example shows off two features of macros. First, each macro has a parameter, `power`, that is used in the body of the macro. Second, the macros are split out to multiple lines by using a backslash.

Sounds and Music

Your RCX can play various prepackaged sounds, using the following command:

PlaySound(const n)
: This command plays the sound represented by `n`. NQC includes constant names for each available sounds, as detailed in Table 4-5.

Table 4-5:RCX System Sounds

Sound Name	Description
SOUND_CLICK	Short beep
SOUND_DOUBLE_BEEP	Two medium beeps
SOUND_DOWN	Descending arpeggio
SOUND_UP	Ascending arpeggio
SOUND_LOW_BEEP	Long low note
SOUND_FAST_UP	Quick ascending arpeggio (same as 3 but faster)

If you'd prefer to make your own music, you can play individual notes with the `PlayTone()` command.

PlayTone(const frequency, const duration)
: This command plays a note with the given frequency for the specified duration. The frequency is in Hz, so 440 is the pitch of the A above middle C on a piano. The duration is in hundreths of a second. You can only specify integer values for the frequency, so don't expect the pitches to be exactly in tune. No one expects your little robot to sound like Pavorotti.

If you want to play a sequence of notes, you have to be a little tricky about it. Each time you call `PlayTone()`, the command returns almost immediately, without waiting for the sound you've requested to finish playing. The tone you've requested is put in a queue; the system plays it while the rest of your program executes. If you call `PlayTone()` repeatedly, the queue will fill up. Subsequent calls to `PlayTone()` will not fit on the queue and the tones you've requested will not be played. The queue is long enough to hold eight tones. If you want to play a sequence longer than this, you should insert calls to `Sleep()` in your program so that the queue has time to empty out as notes are played.

The following example demonstrates this technique; it plays part of *Quando men vo*, from Giacomo Puccini's *La Bohème* .

```
#define SIXTH 12
#define HALF 3*SIXTH
#define BEAT 2*HALF
#define GRACE 6

task main() {
  PlayTone(330, 2*BEAT);
  Wait(2*BEAT + 2*SIXTH);
  PlayTone(115, SIXTH);
  PlayTone(208, SIXTH);
  PlayTone(247, SIXTH);
  PlayTone(330, SIXTH);
  PlayTone(311, 2*BEAT);
  Wait(4*SIXTH + 2*BEAT + 2*SIXTH);
  PlayTone(115, SIXTH);
  PlayTone(208, SIXTH);
  PlayTone(247, SIXTH);
  PlayTone(311, SIXTH);
  PlayTone(277, 3*BEAT);
  Wait(4*SIXTH + 3*BEAT + HALF);
  PlayTone(277, HALF);
  PlayTone(311, HALF);
  PlayTone(370, GRACE);
  PlayTone(330, HALF);
  PlayTone(311, HALF); Wait (2*HALF);
  PlayTone(277, HALF);
  PlayTone(330, HALF);
  PlayTone(220, HALF);
  PlayTone(220, 2*BEAT);
  Wait(GRACE + 5*HALF + 2*BEAT + HALF);
  PlayTone(247, HALF);
  PlayTone(277, HALF);
  PlayTone(330, GRACE);
  PlayTone(311, HALF);
  PlayTone(277, HALF); Wait (2*HALF);
  PlayTone(247, HALF);
  PlayTone(311, HALF);
  PlayTone(208, HALF);
```

```
  PlayTone(208, 2*BEAT);
  Wait(GRACE + 5*HALF + 2*BEAT + HALF);
}
```

IR Communication

Your robot can send and receive data over its IR port. In NQC, three commands handle sending and receiving data:

SendMessage(expression m)
: This command sends the given byte of data out the IR port.

Message()
: Use this command to return the last byte of data received on the IR port.

ClearMessage()
: This command clears the incoming message. You may want to do this after responding to an incoming message in order to avoid responding more than once.

The Display

Although you can't control the display directly in NQC, you can configure it to some degree:

SelectDisplay(expression v)
: This command tells the RCX to show the data source represented by `v` on its display. The legal values for `v` are shown in Table 4-6. You can achieve the same results by pressing the **View** button on the front of the RCX to show the state of the inputs or outputs, but the `SelectDisplay()` command allow you to do this as part of a program.

Table 4-6:Display values

Value	Description
0	System clock
1	View Input 1
2	View Input 2
3	View Input 3
4	View Output A
5	View Output B
6	View Output C

You can set the clock in the RCX using the following macro:

SetWatch(const hours, const minutes)
: Use this macro to set the current time of the RCX's clock. Unfortunately, only constant values can be used.

The Datalog

With the default firmware, your RCX supports an interesting option called a *datalog*. The datalog is simply a list of numbers. You can create a new datalog and put numbers into it. Datalogs can be uploaded to your PC for analysis or display.

CreateDatalog(const size)
: This command tells the RCX to make space for the given number of elements. There is only one datalog, so this command will erase any previous datalog.

AddToDatalog(expression v)
: This command adds a value to the datalog. It's up to you to keep track of how many values are in the datalog. If you try to add values after the datalog is full, nothing happens.

The following example waits for a touch sensor on Input 1 to be pressed. For each press, the value of Timer 0 is stored in the datalog, which holds 20 values in this example.

```
int count;

task main() {
  CreateDatalog(20);
  ClearTimer(0);
  SetSensor(SENSOR_1, SENSOR_TOUCH);

  count = 0;
  until (count == 20) {
    until(SENSOR_1 == 1);
    AddToDatalog(Timer(0));
    count++;
    until(SENSOR_1 == 0);
  }
}
```

When you run this program, you'll notice the RCX shows the status of the datalog on the right side of the display. It looks kind of like a pie; as you add values to the datalog the pie fills up.

To upload a datalog to the PC, you can use `nqc`'s `-datalog` option, which simply dumps the values to the screen:

```
C:\>nqc -datalog
8
12
16
19
23
25
27
29
31
33
39
47
52
56
59
62
65
68
71
75

C:\>
```

The datalog actually stores the source of every value. If you use a tool like RCX Command Center, it can show you the source of each value in the datalog. In *Chapter 8, Using Spirit.ocx with Visual Basic*, I'll show you how to write your own program in Visual Basic to retrieve the contents of the datalog.

Tasks

NQC gives you powerful control over *tasks* and *subroutines*. Each of the RCX's five programs is made up of one or more tasks. These tasks can execute at the same time, which is another way of saying that the RCX is *multitasking*.

Tasks are defined using the `task` command. Every program must have a `main` task which is executed when the program is first started. Other tasks must be started and stopped explicitly.

start taskname
: This command stops the named task.

stop taskname
: Use this command to stop the named task.

The following program controls its outputs from `main` and uses another task, `sing`, to play some music. The `sing` task has to be started from `main`; otherwise, its commands will never be executed.

```
task main() {
  start sing;
  while (true) {
    OnFwd(OUT_A);
    OnRev(OUT_C);
    Wait(100);
    OnFwd(OUT_C);
    OnRev(OUT_A);
    Wait(100);
  }
}

#define SIXTH 12
#define HALF 3*SIXTH
#define BEAT 2*HALF
#define GRACE 6

task sing() {
  PlayTone(330, 2*BEAT);
  Wait(2*BEAT + 2*SIXTH);
  PlayTone(115, SIXTH);
  PlayTone(208, SIXTH);
  PlayTone(247, SIXTH);
  PlayTone(330, SIXTH);
  PlayTone(311, 2*BEAT);
  Wait(4*SIXTH + 2*BEAT + 2*SIXTH);
  PlayTone(115, SIXTH);
  PlayTone(208, SIXTH);
  PlayTone(247, SIXTH);
  PlayTone(311, SIXTH);
  PlayTone(277, 3*BEAT);
  Wait(4*SIXTH + 3*BEAT + HALF);
  PlayTone(277, HALF);
  PlayTone(311, HALF);
  PlayTone(370, GRACE);
  PlayTone(330, HALF);
  PlayTone(311, HALF); Wait (2*HALF);
  PlayTone(277, HALF);
  PlayTone(330, HALF);
  PlayTone(220, HALF);
  PlayTone(220, 2*BEAT);
  Wait(GRACE + 5*HALF + 2*BEAT + HALF);
  PlayTone(247, HALF);
  PlayTone(277, HALF);
  PlayTone(330, GRACE);
  PlayTone(311, HALF);
  PlayTone(277, HALF); Wait (2*HALF);
  PlayTone(247, HALF);
  PlayTone(311, HALF);
  PlayTone(208, HALF);
  PlayTone(208, 2*BEAT);
  Wait(GRACE + 5*HALF + 2*BEAT + HALF);
  stop main;
  Float(OUT_A + OUT_C);
}
```

When the `sing` task is done playing music, it stops the `main` task with the `stop` command. Then it turns the motors off. The order is critical. If we turned off the motors and then stopped the `main` task, it's possible that `main` would turn on the motors again before it was stopped. Multithreaded programming is powerful but tricky.

Each RCX program can have up to 10 tasks.

Subroutines

A *subroutine* is a group of commands that you will execute frequently. Subroutines offer a way to clean up your source code and reduce the size of compiled programs. Subroutines in NQC are defined in much the same way as tasks. The following program has one subroutine, called `wiggle()`. The `main` task shows how this subroutine is called.

```
task main() {
  wiggle();
  Wait(200);
  wiggle();
}
```

```
sub wiggle() {
  OnFwd(OUT_A);
  OnRev(OUT_C);
  Wait(20);
  OnFwd(OUT_C);
  OnRev(OUT_A);
  Wait(20);
  Off(OUT_A + OUT_C);
}
```

Subroutines execute as part of the task from which they are called. It works just as if the call to `wiggle()` was replaced with the commands it contains. The nice thing about subroutines is that their code is defined once, but you can call it as many times as you like from other places in your program. You could accomplish the same sorts of things with subroutines and macros, but subroutines are more efficient because the code is just compiled once. With a macro, the entire macro body would be placed at each point where it was called.

The RCX imposes three crippling restrictions on subroutines. First, you can't call another subroutine from within a subroutine. As a consequence, a subroutine also cannot call itself. Second, you can't pass parameters to a subroutine or get a return value. Third, no more than eight subroutines can be defined for a single program. These limitations are imposed by the RCX's bytecode interpreter, which is defined in the firmware. To get around restrictions like these, you'll need to use a different firmware, like legOS or pbForth.

Inlines

NQC does offer another interesting option, the *inline* subroutine. In source code, it looks a lot like a subroutine except with with a C-style return type (always `void`):

```
void wiggle() {
  OnFwd(OUT_A);
  OnRev(OUT_C);
  Wait(20);
  OnFwd(OUT_C);
  OnRev(OUT_A);
  Wait(20);
  Off(OUT_A + OUT_C);
}
```

Inlines are called the same way as subroutines. The compiler actually places the code of the inline wherever it is called, almost like a macro or constant definition. This actually makes inlines appear a little more capable than subroutines: they can call other inlines or even subroutines. In NQC version 2.0, for example, you can define inlines with a parameter, like this:

```
void wiggleTime(int waitTime) {
  OnFwd(OUT_A);
  OnRev(OUT_C);
  Wait(waitTime);
  OnFwd(OUT_C);
  OnRev(OUT_A);
  Wait(waitTime);
  Off(OUT_A + OUT_C);
}
```

You can have more than one argument, if you wish. Just remember that inlines are really an example of *syntactic sugar*, something that makes your source code look pretty but doens't necessarily result in better efficiency.

Arguments to inlines can be passed in four different ways. Table 4-7 summarizes the options.

Table 4-7:Argument passing for inlines

Type	By value or by reference?	Temporary variable used?
`int`	by value	yes
`const int`	by value	no
`int&`	by reference	no
`const int&`	by reference	no

If you pass `int` by value, the parameter's value is copied into a temporary variable (from the pool of 31) and used in the inline. `const int` passes by value, but the value must be a constant at compile time.

If you pass by reference, the variable that is passed in can actually be modified in the inline. In this code, for example, a `count` variable is incremented in the body of an inline:

```
task main() {
  int count = 0;
  while (count <= 5) {
    PlaySound(SOUND_CLICK);
    Wait(count * 20);
    increment(count);
  }
}

void increment(int& n) {
  n++;
}
```

The last option, `const int &`, is used when you want to pass a value that should not be changed. This is great for things like `Sensor()` and `Timer()`. For example, you might have an inline like this:

```
void forward(const int& power) {
  SetPower(OUT_A + OUT_C, power);
  OnFwd(OUT_A + OUT_C);
}
```

With this inline, you can do normal things, like passing a variable or constant:

```
  int power = 6;
  forward(power);

  forward(OUT_HALF);
```

But you can also do trickier stuff, like this:

```
  forward(Message());
```

You can basically accomplish the same stuff with `int` parameters and `const int&` parameters. The advantage of `const int&` is that no temporary variables are used.

Variables!

To use a variable, you simply need to declare its name. Only integer variables are supported. Once a variable is declared, you can assign the variable values and test it in the body of the program. Here's a simple example:

```
int i;

task main() {
  i = 0;
  while (i < 10) {
    PlaySound(0);
    Wait(5 * i);
    i += 1;
  }
}
```

This example beeps at successively longer intervals. The variable, `i`, is declared in the very first line:

```
int i;
```

Values are assigned to the variable using the `=` operator:

```
  i = 0;
```

You can also assign input values to variables, like this (not part of the example):

```
  i = SENSOR_2;
```

In the following line, one is added to the value in variable `i`.

```
    i++;
```

This is really shorthand for the following:

```
    i += 1;
```

The `+=` operator, in turn, is shorthand for this:

```
    i = i + 1;
```

Trusty Revisited

You've seen some small examples of NQC code. Now I'll show you how Trusty can be programmed using NQC. You'll be able to compare the NQC programs to the RCX Code programs from Chapter 3.

New Brains For Trusty

As you may recall, we used a counter to keep track of Trusty's state. The counter value was used to decide if Trusty would turn left or right the next time the light sensor left the black line. In NQC, we can store Trusty's state in a real variable. Plus, we'll use symbolic constants to represent state values.

```
int state;

#define LEFT 0
#define RIGHT 1
```

Trusty's program has two tasks. The first task (`main`) tests the value of the light sensor. If it is over the black line, the robot is set to move forward:

```
  while (true) {
    if (SENSOR_2 < DARK2)
      OnFwd(OUT_A + OUT_C);
  }
```

The `DARK2` and `POWER` constants are determined using `#define`s; this means it's easy to fiddle with their values, and our program is easy to read.

The second task takes care of things when the light sensor leaves the black line. Whenever the robot leaves the line, the `toggle()` subroutine is called. `toggle()` starts the robot turning. Then we wait a little while; if the robot is still not on the black line, we call `toggle()` again to turn back the other way.

```
task lightWatcher() {
  while (true) {
    if (SENSOR_2 > LIGHT2) {
      toggle();
      Wait(TIMEOUT);
      if (SENSOR_2 > LIGHT2) {
```

```
        toggle();
        Wait(TIMEOUT * 2);
      }
    }
  }
}
```

The `toggle()` subroutine performs two important functions. First, it makes Trusty turn, based on the value of the `state` variable. Second, it updates the value of `state`; if it was `RIGHT`, it will be `LEFT`, and vice versa.

Here is the whole program:

```
int state;

#define LEFT 0
#define RIGHT 1

#define DARK2 35
#define LIGHT2 40

#define POWER 7

#define TIMEOUT 50

task main() {
  state = LEFT;
  SetSensor(SENSOR_2, SENSOR_LIGHT);
  SetPower(OUT_A + OUT_C, POWER);
  start lightWatcher;

  while (true) {
    if (SENSOR_2 < DARK2)
      OnFwd(OUT_A + OUT_C);
  }
}

task lightWatcher() {
  while (true) {
    if (SENSOR_2 > LIGHT2) {
      toggle();
      Wait(TIMEOUT);
      if (SENSOR_2 > LIGHT2) {
        toggle();
        Wait(TIMEOUT * 2);
      }
    }
  }
}

sub toggle() {
  if (state == LEFT) {
    OnRev(OUT_A);
    OnFwd(OUT_C);
    state = RIGHT;
  }
  else {
    OnFwd(OUT_A);
    OnRev(OUT_C);
    state = LEFT;
  }
}
```

The `main` task performs three important initializations which I haven't mentioned yet. First, `main` initializes the value of the `state` variable. It just uses `LEFT` arbitrarily. Next, `main` configures Input 2 for a light sensor. Finally, it starts the `lightWatcher` task.

Using Two Light Sensors

In this section, I'll present an NQC program that works with the two light sensor version of Trusty. As you may recall, programming this robot in RCX Code was cumbersome.

The programming is a lot cleaner in NQC. It's pretty straightforward to translate Table 3-1 into source code. The basic strategy is to use a state variable to represent the four states of the robot, represented by the four lines of Table 3-1. Then one task examines the sensors and updates the state variable. Another task examines the state variable and sets the motors appropriately.

The four possible states are represented by constant values. A fifth value, `INDETERMINATE`, is used when one or both of the light sensor values is not in the dark or light range.

```
#define BOTH_ON 3
#define LEFT_ON 1
#define RIGHT_ON 2
#define BOTH_OFF 0
#define INDETERMINATE 255
```

The `main` task simply tests the value of the `state` variable and sets the motors accordingly. No action is taken for `BOTH_OFF` and `INDETERMINATE`.

```
  while (true) {
    if (state == BOTH_ON)
      OnFwd(OUT_A + OUT_C);
    else if (state == LEFT_ON) {
      Off(OUT_A);
      OnFwd(OUT_C);
    }
    else if (state == RIGHT_ON) {
      Off(OUT_C);
      OnFwd(OUT_A);
    }
  }
```

A separate task, `watcher`, examines the light sensor values and sets the `state` variable. Here is the entire source code for the two sensor version of Trusty.

```
int state;

// "ON" refers to whether the light
//   sensor is on the line. If it is,
//   the light sensor is seeing black.
#define BOTH_ON 3
#define LEFT_ON 1
#define RIGHT_ON 2
#define BOTH_OFF 0
#define INDETERMINATE 255

// Thresholds for light and dark.
#define DARK2 35
#define LIGHT2 40
#define DARK3 40
#define LIGHT3 45

#define POWER 4

task main() {
  initialize();
  while (true) {
    if (state == BOTH_ON)
      OnFwd(OUT_A + OUT_C);
    else if (state == LEFT_ON) {
      Off(OUT_A);
      OnFwd(OUT_C);
    }
    else if (state == RIGHT_ON) {
      Off(OUT_C);
      OnFwd(OUT_A);
    }
  }
}

sub initialize() {
  SetSensor(SENSOR_2, SENSOR_LIGHT);
  SetSensor(SENSOR_3, SENSOR_LIGHT);
  SetPower(OUT_A + OUT_C, POWER);
  OnFwd(OUT_A + OUT_C);
  start watcher;
}

task watcher() {
  while (true) {
    if (SENSOR_2 < DARK2) {
```

```
      if (SENSOR_3 < DARK3) state = BOTH_ON;
      else if (SENSOR_3 > LIGHT3) state = LEFT_ON;
      else state = INDETERMINATE;
    }
    else if (SENSOR_2 > LIGHT2) {
      if (SENSOR_3 < DARK3) state = RIGHT_ON;
      else if (SENSOR_3 > LIGHT3) state = BOTH_OFF;
      else state = INDETERMINATE;
    }
    else state = INDETERMINATE;
  }
}
```

Online Resources

NQC - Not Quite C
http://www.enteract.com/~dbaum/lego/nqc/
: This is the official site for NQC. You can download the current release, read the documentation, or browse a FAQ. Dave Baum has packed a lot of useful information into this site, including such gems as how to create a cable to connect your Macintosh to the IR tower. This site also includes the definitive NQC documentation.

Lego Robots: RCX Command Center
http://www.cs.uu.nl/people/markov/lego/rcxcc/
: RCX Command Center (RcxCC), developed by Mark Overmars, is built on top of NQC. It's a Windows application that provides a friendly graphic interface to the features of NQC. It includes a syntax-colored program editor, real-time control of the RCX, utilities for making your RCX play music, and useful help files. I highly recommend this application.

Lego Robot Pages [NQC Tutorial]
http://www.cs.uu.nl/people/markov/lego/
: Mark Overmars, creator of RcxCC (the previous entry), has written a detailed introduction to NQC. It's available off his main web page as PDF, Word97, PostScript, or RTF. This document is a gentle and thorough introduction to NQC.

Kevin Saddi's NQC Reference Page
http://home1.gte.net/ksaddi/mindstorms/nqc-reference.html
: This page provides a distilled view of NQC. It's very handy when you can't remember the input type constants, or you'd like to see NQC's commands organized by function. Single-line code samples are also included.

Hitachi Single-Chip Microcomputer H8/3297 Series...
http://semiconductor.hitachi.com/products/h_micon/3_h8300/3_h8_300/H33TH014D2/html/h3314frm.cfm
: This page has all the crufty details on the Hitachi H8 that forms the heart of the RCX. The specific model is the H8/3292, which is covered in the manual. This information is not for casual browsing -- you probably won't need to look here unless you start writing your own firmware. (Hitachi's web site is a little flakey. If you're having trouble with this URL, try starting at http://semiconductor.hitachi.com/h8/ and searching for the H8/3292.)

Back to: LEGO® MINDSTORMS(TM) Robots

---

oreilly.com Home | O'Reilly Bookstores | How to Order | O'Reilly Contacts
International | About O'Reilly | Affiliated Companies | Privacy Policy