```
int main(int argc, char **argv)
{
  /* turn it on */
  ds_active(&SENSOR_2);
  ds_rotation_on(&SENSOR_2);

  /* calibrate it to 0 */
  ds_rotation_set(&SENSOR_2,0);
  msleep(100);

  while(1)
    {
      lcd_int(ROTATION_2);
      msleep(20);
    }
}
```

Remembering that the rotation sensor is an active sensor, the first thing this code (and any rotation sensor code) must do is turn on the sensor itself with `ds_active(&SENSOR_X)`. Once that is done, `ds_rotation_on(&SENSOR_X)` initializes the rotation sensor. Both of these steps are required. After that, we can at any time calibrate the sensor to any starting point we want, by calling `ds_rotation_set(&SENSOR_X, value)`. It is helpful to remember that while this value is usually 0 (because we typically only want to count rotations), it can be set to other values as well.

Once the sensor has been set up with these three functions, we can use `ROTATION_X` just as we've used `LIGHT_X` and `TOUCH_X` previously. Rotation sensors have a resolution of 16 ticks per revolution, so two full rotations would be read as a value of 32. Generally speaking, rotation sensors are quite reliable. However, the rotation sensor will not always update correctly if control of the program stays with another thread for too long. This isn't usually a problem. Multi-tasking is much improved in legOS 0.2.x, and the processing of interrupts has also improved, so the kernel should count all rotation sensor changes with no problems. The second potential problem is that rotation sensors don't function well if the RCX batteries get too

Advanced LegOS

THE PREVIOUS CHAPTER COVERED the vast bulk of the functionality of legOS. If you've read it, you should be able to write some interesting legOS programs. However, there are still a few gaps in your legOS knowledge. In this chapter, we'll look at a more advanced application for legOS, and the legOS functionality needed to support it and other advanced legOS programs. Once you've completed this chapter, you will be ready to explore the full potential of legOS.

Rotation Sensors

Before delving deeper into the mysteries of legOS, we should tackle a more mundane topic—rotation sensors. Long the bane of legOS users, the rotation sensor code in legOS finally reached production quality with legOS 0.2.3. They are now quite reliable, and reasonably straightforward to use. They are also versatile and provide an easy way for robots to measure angles, rotations, distance, or speed. The sample program in Listing 8-1 is a simple one that counts the number of times the rotation sensor has been turned.

Listing 8-1. rotation.c

```
/* rotation.c */

#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
/*
 * main
 */
```

weak. If it looks like you're losing counts—in other words, if `ROTATION_X` doesn't appear to change when the rotation sensor moves, or doesn't change enough—try replacing the batteries and your results should markedly improve.

Sound

One of the more entertaining features of legOS is the sound driver, although it is not incredibly useful (especially when compared to the rotation sensor). The RCX speaker is the last piece of hardware that this chapter will cover. It's not your home stereo system, but it does have a surprising range of tones and can produce some recognizable music.

NOTE *To access the speaker, you must include the file `dsound.h`.*

To play a sound or series of notes with legOS, you must first characterize the sound or notes as a series of `note_t` structures. Each `note_t` contains a pitch and a duration. Pitches are specified from a list of tones in the `dsound.h file`, each of which has the form `PITCH_XY`, where X is the note (A, C, D, etc.) and Y is the octave (1-8). Durations can be specified in two ways: either as multiples of sixteenth notes (for example, a half note would be given as 8) or, more conveniently, through the use of some `#defines`, which are defined as the multiples for common notes like `WHOLE`, `HALF`, `QUARTER`, and `EIGHTH` notes.

Once the notes have been defined, they need to be put into a vector of `note_ts`. That vector is then passed to the function `dsound_play()`, which plays the vector. A number of auxiliary functions are available and defined in `dsound.h`; for example, `dsound_set_duration()` sets the length of a sixteenth note in milliseconds, which allows you to control the pace of music played by the RCX. Another important function is `dsound_playing()` . It returns true if music is playing, and false otherwise. This is important because `dsound_play()` returns as soon as it is called, therefore multiple calls to `dsound_play()` might overlap each other if `dsound_playing()` is not used first as a test.

The example shown in Listing 8-2 uses a snippet from the Mitch Ryder song "Devil with a Blue Dress on" to demonstrate how all of these functions work.

Listing 8-2. sound.c

```
/*sound.c*/

#include <dsound.h>

/*array of notes that make up the refrain*/
/*of Devil with a Blue Dress*/

static const note_t devil[] = {
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },

  { PITCH_G4, HALF },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },

  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, HALF },
  { PITCH_F4, HALF },

  { PITCH_F4, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },

  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_F4, HALF },
  { PITCH_F4, HALF },

  { PITCH_E4, HALF },
  { PITCH_E4, HALF },
  { PITCH_F4, HALF },
  { PITCH_F4, HALF },
```

```
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_F4, HALF },
  { PITCH_F4, HALF },

  { PITCH_E4, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_END, 0 }
};


int main(int argc,char *argv[]) {

  /*The default makes this a really, really slow song*/
  /*So, we speed it up a little bit.*/
  dsound_set_duration(40);

  /*now, we play it*/
  while(1) {
    dsound_play(devil);
    wait_event(dsound_finished,0);
    sleep(1);
  }

  return 0;
}
```

"Devil" is a very fast-paced song, so we have to use `dsound_set_duration()` to shorten the length of the notes. While 40 milliseconds (ms) doesn't sound long, remember that this is for a sixteenth note—the quarter notes in the tune are 160 ms and the half notes are 320 ms, which are considerably longer periods of time. The `dsound_finished()` function used in the call to `wait_event` is essentially a wrapper function (also found in `dsound.h`) around `dsound_playing()`. It ensures the previous sound snippet has finished before we start playing it again.

> NOTE: If you have the Ultimate Accessory Kit for MINDSTORMS, you have one other piece of hardware: the LEGO Remote Control. While the remote is not yet officially supported by legOS, at least a couple of people have reported success using it, and it seems likely that it will be supported by the time this book reaches the shelves. Always check http://legos.sourceforge.net/ for the latest news on this and other legOS updates.

Math in LegOS

One of the big advantages of using legOS over the standard firmware is the ability to do a lot of math. The next two sections will explain two wrinkles that make math in legOS slightly more complex than just "x = y + z."

Floating Point

As you may have already noticed if you have tried to do math in NQC, the RCX is normally limited to integer math (for example, math without decimal points.) This is because the Hitachi chip the RCX is based on has no Floating Point Unit (FPU), the section of a processor dedicated to floating point math.

In most cases, there are simple workarounds for this problem. For example, in seeker.c (and in NQC programs), operations like multiplying by .85 (for the threshold calculation) are done first by multiplying by 85 and then dividing by 100. For most applications, this approach is sufficient. In fact, experienced embedded systems designers such as Ralph Hempel (author of the pbForth chapters in Part Two of this book), argue that most math, even in complicated programs, should be done without using floating point. However, for programmers used to the flexibility of a desktop, this approach may seem a little backward. Furthermore, it makes it easy to make mistakes (one early version of seeker.c multiplied by 10 and divided by eight in an attempt to get 80%).

Luckily, Kekoa Proudfoot has written a small floating point library that can be linked into legOS for those times when floating point math is either simpler or more necessary. In fact, as of the 0.2 series, legOS is set up to automatically include the library whenever you use a `double` or `float` in your code. Even though it is now automatic (and as a result, simple to use) there are still a few reasons to consider avoiding using these numeric types when you write large programs.

First, the entire library is linked to your .lx file, not the legOS kernel. While this makes the kernel more flexible, it means, for example, that a program which counts from 1 to 2^{16} using an `int` requires about 92 bytes of memory, while the same program using a `double` and doing no math except addition uses 980 bytes. As mentioned previously, this isn't usually a problem because the 32KB accessible by legOS is a

surprisingly large amount. However, if you want to use legOS for something sophisticated, the linkage of the library into your .lx file is definitely something to keep in mind.

The other detriment to floating point math is the speed of the operation. While this difference isn't noticeable most of the time, in math intensive applications, floating point emulation will slow down your program. For example, calculating x*0.85 in floating point is roughly fifty percent slower than calculating (x*85)/100 in integer math. In most cases, this difference won't be noticeable, but in some uses it can be significant.

Type Sizes

Perhaps the most insidious bugs I've ever had to track in a legOS program stemmed from a simple source: misunderstanding the size of variable types when using gcc and legOS. For instance, the example in the preceding section (where `int x` is multiplied by 85) can easily cause an error if x is too large. Why? Well, because an `int` in legOS is only 16 bits, it has an upper bound of only about 32,000. If you try to fill an `int` with something larger than that (say, 85*400, or 0xffff in the case of sensor raw values), you'll get no error messages and many garbage numbers. The solution is, of course, very simple—just use a `long int` instead of an `int`. Despite the simple solution, this can be an irritating problem to debug, so make this trick one of the first things you try if numbers mysteriously change to values you don't expect.

Because the normal source for these values (`limits.h`, a compiler file) is incorrect in the cross-compiler source, I've included a table of type sizes.

TYPE	SIZE IN BYTES	UPPER BOUND	LOWER BOUND
`int`	2	32,767	-32,768
`unsigned int`	2	65,535	0
`long int`	4	2,147,483,647	-2,147,483,648
`float`	4	Depends on amount of accuracy necessary.	
`double`	4	Depends on amount of accuracy necessary.	

LegOS Network Protocol (LNP)

While legOS generally exceeds the standard firmware in flexibility and power, there has traditionally been one gaping hole—the ability to communicate back

and forth with the PC. This period is coming to a close. LegOS has its own networking protocol: LegOS Network Protocol, or LNP. If you look closely at the source, you'll note that LNP is already used in firmdl3 and dll to make downloads faster and more reliable. More importantly, you can now access LNP-enabled RCXs from computers running Linux, thanks to the work of Martin Cornelius. He has written two pieces of code (LibLNP and the LNP Daemon, or LNPD), which allow programmers to write Linux programs that access the IR tower and the robots.

> **NOTE** *LibLNP and LNPD Daemon are Linux-specific tools. However shortly before this book went to print, Yarius Jeanpierre released a similar set of tools for Windows called WinLNP, enabling robots to be accessed from Visual C++, MS Java, and other Windows programming environments. You can find links to these new tools (called appropriately enough WinLNP) at* `http://legos.sourceforge.net/#Files`.

The model for these tools is reasonably straightforward. The LNP daemon is a continually running process that prevents the IR tower from shutting off and brokers requests between a program running on the PC and the program on the RCX. LibLNP is the set of interfaces that allow the Linux program to contact the daemon and send messages to the RCX. This is a very flexible model: any number of programs can access the daemon while running on the PC, and because every legOS kernel can have its own LNP address, you can communicate with up to eight different RCXs at once. For further details, download the package from `http://legos.sourceforge.net/files/linux/LNPD/` and read the README and .h files.

LegOS Debugging

Unfortunately, debugging legOS programs is often the low point of the process of coding for legOS. This is usually the case for any serious programming exercise, of course, but you haven't really known pain until you've tried to debug a large program on a five character LCD screen using an OS that can be a bit unstable. With those problems in mind, here are a few things from my experience that may help with debugging:

- **Frequently output variables**. Because there are no formal debugging tools for legOS (such as gdb), there is no way to trace the values of variables except by dumping them to the screen as often as possible. By this I mean as often as every other line, even when the intervening lines are something simple like "`x = y * z`." Remember, even simple operations like this can bite when you're used to working with 32-bit integers that are suddenly 16 bit. Frequently dumping variables can help you figure out exactly which line of "perfect" code is actually the culprit.

- **Label variables as you dump them.** Integers flashing by on the screen are too easy to lose track of. To help, I often use a function that takes a label and a variable value as arguments, then outputs them to the screen with `msleep()` calls of appropriate length between them. A simple function like this can condense four lines of code (`cputs`, `sleep`, `lcd_int`, `msleep`) into one line, making your code more readable, while still getting all the necessary information onscreen .

- **Make variables global.** While generally considered bad programming form for larger programs, it can help keep things simple, which should be a primary goal when writing shorter programs (as most of your legOS programs will probably be.)

- **Download the kernel again.** It has been my experience that starting from scratch by using firmdl3 to download a new kernel is sometimes the answer to obscure or difficult problems. Why this is the case is not clear, but it is possible that certain kernel operations are not thread-safe and may cause data corruption within the kernel. Also, you can't mix and match kernels and .lx files: if you've compiled an executable against one kernel and then rebuilt the kernel, you may be able to download the program but it will probably die mysteriously.

- **When in doubt, post your code and your symptoms to** `lugnet.robotics.rcx.legos@lugnet.com`. This is the online gathering place for most of the legOS developers and serves as a good place to ask questions and get answers. There are some very knowledgeable folks there—they've helped me a ton.

Above all, don't give up. While trying to create a sophisticated program with legOS can be frustrating, the results can be very rewarding.

Trailerbot

As noted, the strengths of legOS lie in the ability to do math, store large numbers of variables, and use all the features of the C language. Trailerbot uses all of these features. As result, it not only pushes the limits of what the RCX is physically capable of, but it serves as a great demonstration of the kind of complex work that can be done with legOS, given time and skill.

Trailerbot is a variation on a class project from CPS 196, a legOS-based course taught at Duke University in the Fall of 1999 It was intended at that time to demonstrate a strategy in artificial intelligence (AI) called reinforcement learning. Despite its new home in the RCX, the Trailerbot is a variation of a traditional experiment in

learning robots: the pole balancing robot or inverted pendulum problem. In traditional versions of the problem, a pole mounted on a robot must be kept balanced straight up in the air. Because the RCX really can't compute that quickly, we lay the pole on its side, and it becomes a trailer for the RCX to attempt to push in a straight line without jackknifing. What makes the task interesting is that instead of explicitly telling the robot what to do, it is programmed to learn on its own, beginning with a limited set of information about itself and the outside world.

Using the analogy of a trailer introduces some inevitable terminology problems. Normally, you pull a trailer instead of pushing it; you only push it when you go backwards. To clarify, when I say the robot or trailer is "going forward," I mean that the motors are pushing towards the trailer, and not pulling it, which is how trailers normally "go forward." The same "flipping" of terms applies to backing up—when I say backing up, I mean that the robot is moving away from the trailer. Also, because "forward" is toward the motor ports, the "right" side of the robot is the side with the On-Off and View buttons, and a right turn rotates the robot in that direction.

Building Trailerbot

Trailerbot uses a rotation sensor not included as part of RIS 1.0 (#9719) or RIS 1.5 (#9747). You may find a rotation sensor in the Ultimate Accessory Set (#3801) or you may purchase it individually from LEGO Shop-At-Home (1-800-453-4652). Aside from the rotation sensor, the rest of Trailerbot can be built with the parts contained in either RIS 1.0 or RIS 1.5.

Trailerbot consists of two parts: the main body and a trailer. Actually, the term "trailer" is a misnomer because the goal of Trailerbot is to push (rather than pull) its trailer, thus the trailer is in front of the main body. Two wheels at the front end power the main body and a skid supports the rear end.

First, construct the body's frame as shown in Figures 8-1 and 8-2.

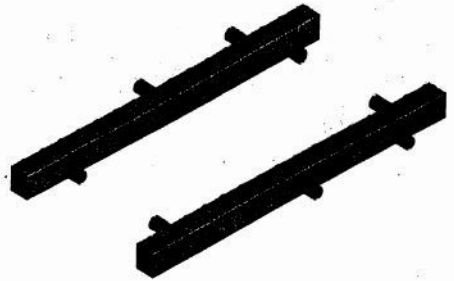

Figure 8-1. Step 1

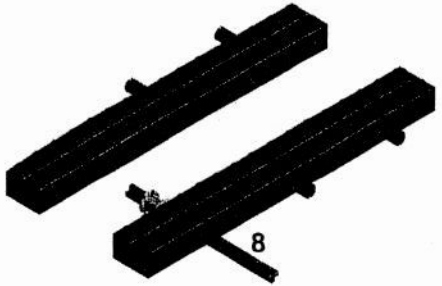

Figure 8-2. Step 2

Next, build the skid, shown in Figures 8-3 to 8-5, and the hitch, shown in Figure 8-6.

Figure 8-3. Step 3

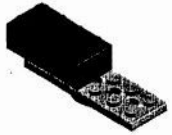

Figure 8-4. Step 4

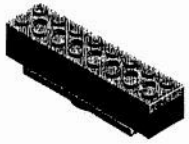

Figure 8-5. Step 5

Figure 8-6. Step 6

Continue by adding the skid and hitch to the frame, along with the support for the motors as depicted in Figures 8-7 to 8-9.

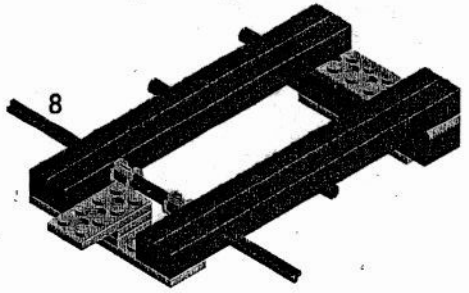

Figure 8-7. Step 7

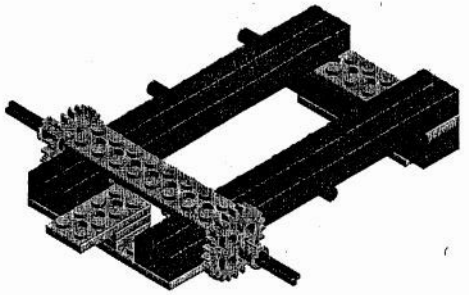

Figure 8-8. Step 8

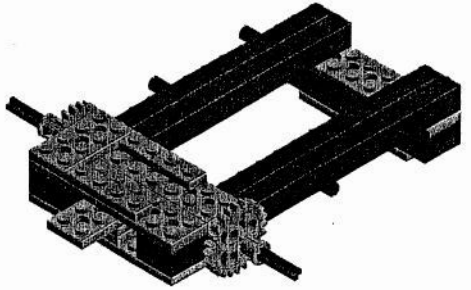

Figure 8-9. Step 9

The motors, the RCX, and most of the wiring is added next, as shown in Figures 8-10 and 8-11. Be sure to use 24-tooth gears on the motors so they can mesh with the gears on the axles below. Pay careful attention to the orientation of the motor wires—if not placed properly, Trailerbot may not move forwards when the RCX turns on both motors in the forward direction. Although not critical to Trailerbot's operation, a light sensor is added so the random number generator can be seeded with an external value. The completed body is shown in Figure 8-12 with the addition of tires and several 1×6 beams to hold everything together.

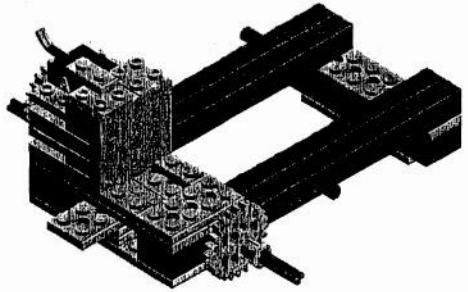

Figure 8-10. Step 10

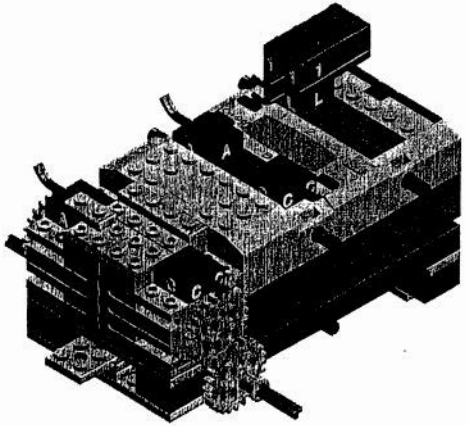

Figure 8-11. Step 11

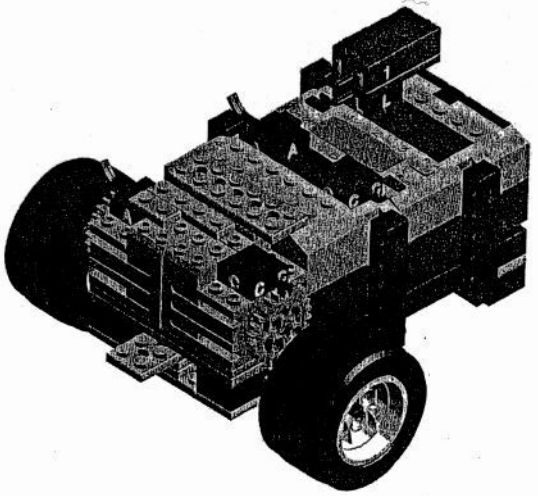

Figure 8-12. Step 12

Construction of the trailer begins is shown in Figure 8-13 with a #8 axle and a bushing. One end of the bushing is circular while the other end has a small notch in it. Most of the time it doesn't matter which way a bushing is put onto an axle, but in this particular case it is important that the notched side faces up. A few more pieces are added to the axle as shown in Figure 8-14. Don't worry about the spacing between pieces—they will be adjusted after the trailer is attached to the body.

Figure 8-13. Step 13

Figure 8-14. Step 14.

The trailer is completed in Figures 8-15 to 8-17. Be sure to use the smallest wheel—larger wheels won't be able to spin freely.

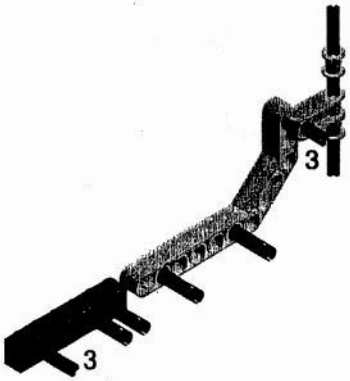

Figure 8-15. Step 15

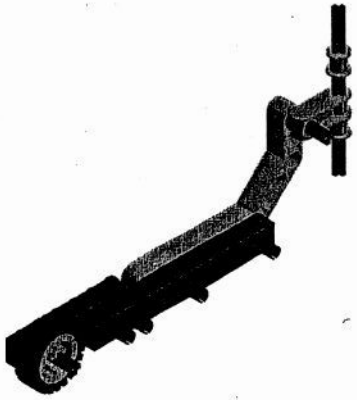

Figure 8-16. Step 16

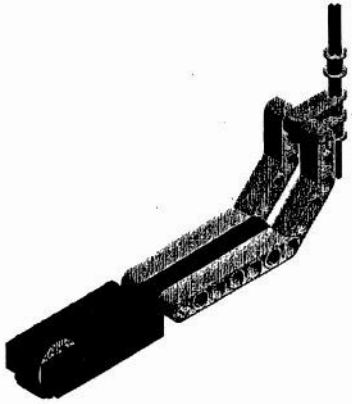

Figure 8-17. Step 17

The trailer is attached to the main body (as shown in Figure 8-18) by feeding the bottom of the trailer axle through the hole in the body's hitch. A 2×4 TECHNIC plate (a plate with holes in it) should be added to secure the top end of the axle to the body. This may require some minor adjustment to the bushings on the axle. Once completed, the bottom bushing should rest on the hitch, and the top bushing should be just below the newly added 2×4 TECHNIC plate. The trailer itself should be able to pivot easily left and right. If it does not, the top bushing is probably pressed too tightly against the top plate—slide it down a bit.

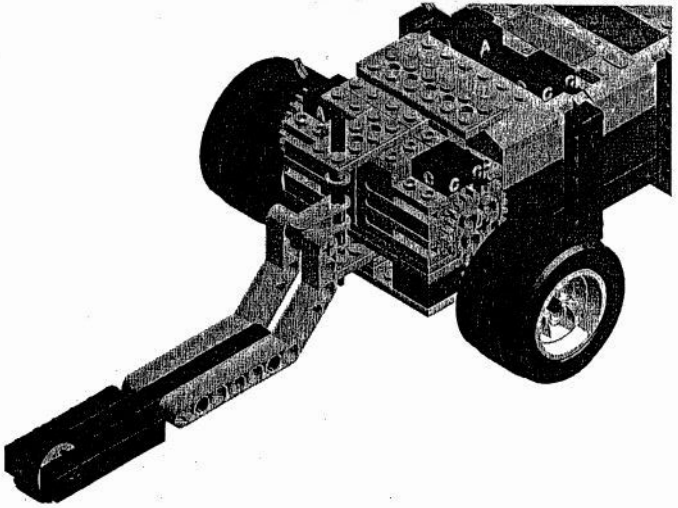

Figure 8-18. Step 18

Trailerbot uses a rotation sensor to determine the position of its trailer. This is done by attaching an 8-tooth gear to the rotation sensor, then meshing this gear with a 24-tooth crown gear placed at the top of the trailer axle. As the trailer swings left or right, the crown gear will turn, which will then spin the rotation sensor. The rotation sensor and crown gear are added in Figures 8-19 to 8-21. The crown gear might need slight adjustment to mesh smoothly with the rotation sensor's gear.

Figure 8-19. Step 19

Figure 8-20. Step 20

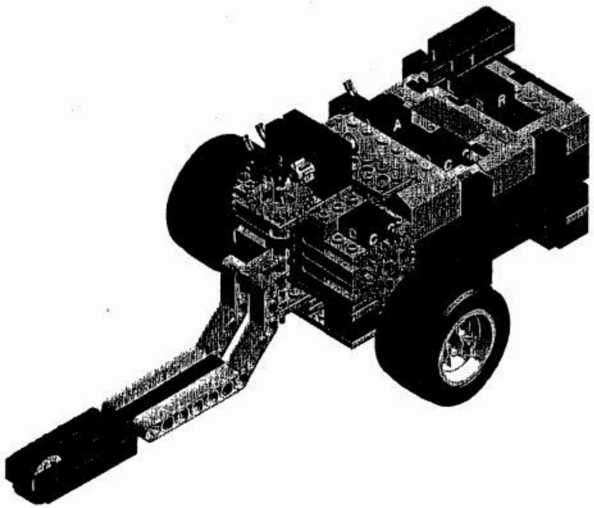

Figure 8-21. Step 21

The rotation sensor has a resolution of 16 steps per rotation, or 22.5 degrees per step. However, the meshing of the crown gear and 8-tooth gear provides a gear ratio of 3:1, thus the rotation sensor will measure the position of the trailer with a resolution of 7.5 degrees.

How Trailerbot Learns

Trailerbot uses a specific type of reinforcement learning algorithm, called Q learning, to understand how to push around the trailer. In a nutshell, this means that the code does two things. First, the robot uses trial and error to learn the probability

that an action (such as turning or backing up) will lead to a specific state (for example, jackknifed or straight). Second, it tries to figure out which action will give it the greatest probability of staying straight. This is complicated, but it will become more clear as we step through the code.

> **NOTE** *Trailerbot's algorithm is not strictly Q learning, but it is pretty close to the real thing. To get the lowdown on Q learning complete with lots of actual math, check out* http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node25.html *or* http://envy.cs.umass.edu/~rich/book/6/node6.html.

Q learning is a good choice for this application because it learns best when there aren't many options. In fact, as we'll see, the code takes special steps to refine and distill what the robot knows about the world.

Seeing the World

As you may have noticed from building the robot, it really doesn't have much information about what is going on around it, except for the lone rotation sensor. This sensor is geared up to give us about 20 "clicks" of resolution between the left jackknifed position and the right jackknife. We won't use all of this resolution because the robot would be overwhelmed. Instead, we'll use the function shown in Listing 8-3 to "water down" what the robot sees into information it can deal with more easily.

> **NOTE** *Because the actual code for Trailerbot is long (nearly 600 lines), I won't show it in its entirety in the text. Additionally, because the comments are long (frequently longer than the code itself), I've removed them in many cases in order to make the code presented here more readable. To run the program, download the sample programs from the book's page on the Apress Web site:* http://www.apress.com. *The program for Trailerbot is called* trailerbot.c.

Listing 8-3. Creating manageable rotation.

```
/*
 * Function to convert rotation from 0-19
 * to a more manageable 0-4
 */
```

```
int norm_rotation(int real_value)
{
  switch(real_value)
    {
      /*right jackknife*/
    case -1:
    case 0:
    case 1:
    case 2:
      return 0;
      /*leaning toward the right*/
    case 3:
    case 4:
    case 5:
    case 6:
      return 1;
      /*centered!*/
    case 7:
    case 8:
    case 9:
    case 10:
    case 11:
    case 12:
      return 2;
      /*leaning towards the left*/
    case 13:
    case 14:
    case 15:
    case 16:
      return 3;       /*left jackknife*/
    case 17:
    case 18:
    case 19:
    case 20:
      return 4;
      /*if none of these, big error*/
    default:
      /*to be easily visible on LCD*/
      cputs("ERROR");
      sleep(1);
      return -1;
    }
  /*to get rid of compiler warning*/
  return 0;
}
```


The switch statement is a handy construct in C. In this case, it gets used to make a simple converter from what the rotation sensor "sees" to more meaningful information for the robot.

Controlling Actions

One of the strengths of being human is that we learn things from one situation and apply that learning to others. MINDSTORMS robots don't have that property (even with legOS, oddly enough). So, like the rotation sensor being filtered down to five inputs, we have to limit the robot to a choice of six outputs:

- Front/back
- Hard right/hard left
- Soft right/soft left

We control these movements with the block of motor control command code shown in Listing 8-4:

Listing 8-4. Controlling movements with motor control commands.

```
/*
 * Big block of motor commands
 * provides simple functions to cleanup later code.
 */

void run_motors()
{
  motor_a_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED);
}

void go_forward()
{
  motor_a_dir(fwd);
  motor_c_dir(fwd);
  run_motors();
  msleep(100*TURN_MULTIPLIER);
}
```

```
void go_back()
{
  motor_a_dir(rev);
  motor_c_dir(rev);
  run_motors();
  msleep(150*TURN_MULTIPLIER);
}

void soft_left()
{
  motor_a_dir(fwd);
  motor_c_dir(fwd);
  motor_a_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED/2);
  msleep(75*TURN_MULTIPLIER);
}

void soft_right()
{
  motor_a_dir(fwd);
  motor_c_dir(fwd);
  motor_a_speed(MAX_SPEED/2);
  motor_c_speed(MAX_SPEED);
  msleep(75*TURN_MULTIPLIER);
}

void hard_right()
{
  motor_a_dir(rev);
  motor_c_dir(fwd);
  motor_a_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED);
  msleep(100*TURN_MULTIPLIER);
}

void hard_left()
{
  motor_a_dir(fwd);
  motor_c_dir(rev);
  motor_a_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED);
  msleep(100*TURN_MULTIPLIER);
}
```



```
void stop_motors()
{
  motor_a_speed(0);
  motor_c_speed(0);
  motor_a_dir(brake);
  motor_c_dir(brake);

  /*to conserve batteries*/
  msleep(500);
  motor_a_dir(off);
  motor_c_dir(off);
}
```

Unfortunately, these commands don't cross over between floor surfaces very well. You may have to adjust them to be more appropriate functions for your floor. Luckily, adjusting them is pretty simple—find `TURN_MULTIPLIER` (defined near the top of the code) and fiddle with it. A value of seven works pretty well on my carpet, and I've used a four on my hardwood floors. Having said this, don't worry about adjusting them too much—one of the benefits of having your robot learn (instead of having you teach) is that if something doesn't work well, the robot will learn and stop doing it. So, if `hard_right()` turns your robot 180 degrees because your floor is smoother than carpet, the robot will learn that `hard_right()` always jackknifes and should stop avoid such a turn in the future.

Defining the World with Constants and Arrays

Now that we've figured out how the robot interacts with the world, we should try to understand how the robot keeps track of the world. Generally speaking, all the important information is stored in global arrays or `#define` statements at the beginning of the code, which appears in Listing 8-5 (without the comments).

Listing 8-5. Tracking the world in global arrays or #define statements.

```
#define ANGLES 6
#define MOVEMENTS 6

#define FAR_RIGHT 0
#define FAR_LEFT 4

#define TURN_MULTIPLIER 5
```

```
#define EPSILON_MAX .60
#define EPSILON_MIN .10
#define EPSILON_DECAY .02
double epsilon = EPSILON_MAX;

#define ALPHA .20
#define GAMMA .90
#define KAPPA 10

#define HEAVEN 5
#define HEAVEN_REWARD 20

enum movement
{
  HARD_LEFT, SOFT_LEFT, FORWARD, SOFT_RIGHT, HARD_RIGHT, REVERSE
};

double steering_results[ANGLES][MOVEMENTS][ANGLES];
double q_score[ANGLES][MOVEMENTS];
```

The first set of #defines and the last set of arrays are probably the most important information here. ANGLES is the number of values the robot can get from the rotation sensor. Remember, we're limiting the number to five (the range 0–4) with the norm_rotation() function. However, the code makes ANGLES equal to these five states plus one more. As you can see, the rotation sensor will never read 5, so the robot will never make it to this "extra" state. For now, all you need to know about this extra state is that it exists to trick the robot—by inserting it into ANGLES, we tell the robot that this state exists, and it never knows that it can't get there. The reason for this trickery will be explained later, in the Q Score section. The two special constants are also related to this state—HEAVEN indicates that the state is in the "fifth" rotation sensor state (that can't be reached) and HEAVEN_REWARD tells the robot that, should it ever get to HEAVEN, it will be well rewarded.

Similar to the mapping of trailer orientations to ANGLES, each of the six basic motions (such as hard_right) is mapped to MOVEMENTS. The two arrays, steering_results and q_score, are the "memory" of the algorithm. Think of steering_results as a table where the robot can look up its current angle (the first index), ask about a specific movement (the second index), and find out the probability that this movement will get it to other angles (the third index). Basically, this information is stored so that once the robot knows where it wants to go, it can figure out the best way to get there from where is currently located.

The q_score array is similar. In it, the robot will look up the angle it is currently in (the first index), ask about possible movements (the second index), and get back the Q score of each movement. Simply put, the higher the Q score, the more desirable

the associated pairing of angle and movement is. The q_score array then stores the information about that desirability.

> NOTE steering_results and q_score are initialized at the beginning of the code in the array_initialization() function. Because steering_results is a probability, all of its values are set to be equal to each other. Q score is initialized to zero because the robot has no prior knowledge of which actions are good or bad.

The three Greek letters (ALPHA, GAMMA, and KAPPA) and the epsilon definitions are parameters that control the learning process itself. We'll look at them in more detail once we plunge into the algorithm in the Q Score section.

Most of the other constants defined in this section of code are pretty straightforward. The movement enum and FAR_LEFT and FAR_RIGHT are ways to link the world to specific numeric values that can be used as indexes for the arrays we have already defined.

Moving Trailerbot

Once the robot has been set up and initialized, it must begin to explore the world and interpret what it learns. In Trailerbot, this occurs in the move() function. Every call of the move() function results in one movement and one update of steering_results, as shown in Listing 8-6.

> NOTE The norm_random() function called in this code is just a wrapper for random() that returns a random number between 0 and 1.

Listing 8-6. Calling move() *functions and updating* steering_results.

```
void move()
{
  int i;

  /*variable to use in figuring out the "best" option*/
  int max_q_score = 0;
```

```
/*what do we do next? store it here*/
/*we init to -1 as an error*/
int next_movement = -1;

/*Where we started.*/
/*We don't use ROTATION_2 all the way through in case it changes.*/
int initial_angle = norm_rotation(ROTATION_2);

/*Where we ended up.*/
int new_angle;

/*Show the current angle*/
cputs("ANGLE");
msleep(200);
lcd_int(initial_angle);
msleep(500);

/*
 * Most of the time, we do the "correct" thing
 * by finding the best q_score of our possible options.
 * On the off chance that norm_random() is low (or EPSILON is high ;)
 * we then "explore" by choosing a random movement.
 */

if(norm_random() > epsilon)
  {
    /*We are doing what the table tells us to.*/
    cputs("real ");
    msleep(500);

    for(i=0; i<MOVEMENTS; i++)
      {
        if(q_score[initial_angle][i] > max_q_score)
          {
            max_q_score = q_score[initial_angle][i];
            next_movement = i;
          }
      }
  }
else
  {
    double temp;
    /*We are just picking something at random.*/
    cputs("rand ");
    msleep(500);
```



```
    /*pick one. Any one.*/

    temp = norm_random();
    next_movement = temp*MOVEMENTS;

    /*show what we do next*/
    lcd_int(next_movement);
    sleep(1);
  }

/*what happens if next_movement never gets changed?*/
/*we'd hate to do HARD_LEFT over and over again*/
/*so we choose randomly*/

if(-1==next_movement)
  {
    double temp;
    temp = norm_random();
    next_movement = temp*MOVEMENTS;
  }

/*having chosen a movement, lets do it*/
switch(next_movement)
  {
  case HARD_LEFT:
    cputs("HL   ");
    hard_left();
    break;
  case SOFT_LEFT:
    cputs("SL   ");
    soft_left();
    break;
  case FORWARD:
    cputs("FWD  ");
    go_forward();
    break;
  case SOFT_RIGHT:
    cputs("SR   ");
    soft_right();
    break;
  case HARD_RIGHT:
    cputs("HR   ");
    hard_right();
    break;
```

```
  case REVERSE:
    cputs("REV  ");
    go_back();
    break;
  default:
    /*this is an error and should never be reached*/
    cputs("ERROR");
    sleep(1);
    stop_motors();
    break;
  }

/*Once we've started, we'd better stop*/
stop_motors();

/*Allows us to read direction*/
msleep(500);

/*This is here just to make the next function cleaner*/
new_angle = norm_rotation(ROTATION_2);

/*Where are we now?*/
cputs("NEW  ");
msleep(200);
lcd_int(new_angle);
msleep(500);

/*
 * Since we know that "next_movement" took us from "initial_angle"
 * to new_angle (ROTATION_2), we store that increased probability.
 */

steering_results[initial_angle][next_movement][new_angle] += ALPHA;

/*here we re-norm so that the sum of the probabilities is still 1*/
for(i=0; i<ANGLES; i++)
  {
    steering_results[initial_angle][next_movement][i] /= (1+ALPHA);
  }
```



```
  /*The last thing we do is reduce Epsilon*/
  if(epsilon > EPSILON_MIN)
    {
      epsilon-=EPSILON_DECAY;
    }

}
```

For the robot, exploration is always a mix of doing what it thinks is "right" and (occasionally) trying something random to see what happens. As you can see in the section of code that uses `epsilon`, the percentage of time the robot chooses a random direction is set in `epsilon`. If the random number generated is more than `epsilon`, the robot chooses the "best" option by looking it up in the Q table. It determines this best movement by looking up where it is (`initial_angle`) in the `q_score` array and then going through the array until it finds the `MOVEMENT` with the highest Q value. This is, of course, not necessarily the best option; the first time around, all Q values are still zero, thus the robot defaults to a random choice.

If the random number generator picks a value less than or equal to `epsilon`, a random choice is made. Balancing the random and "learned" is difficult: if the robot is random too often, it will often make poor decisions even after it has learned what to do, but if it is not random enough, it will learn very slowly. In order to balance this, we use the values `EPSILON_MAX`, `EPSILON_MIN`, and `EPSILON_DECAY`. By setting `EPSILON_MAX` high, the robot starts randomly, learning as it goes. In code not shown here, the robot reduces `epsilon` by `EPSILON_DECAY` every time the code is called. In this way, the robot becomes less random as it learns more. With any luck, by the time `epsilon` equals `EPSILON_MIN`, the robot will have learned some reasonably intelligent behavior and won't have to make as many mistakes as it continues to learn.

After the movement has been chosen, the `move()` function continues by using a `switch` statement to issue the command to move. Once it has moved, the robot has to figure out where it went! That's not so tough (it simply calls `ROTATION_2` again). However, the robot can't just throw away that information. It has to store it to better understand the results of its actions. It does this by adjusting the probabilities in `steering_results` to reflect the fact that it started in `initial_angle`, performed `next_movement`, and ended up in `new_angle`. The constant `ALPHA` controls the rate of this update: a higher alpha makes the table update faster, but it also means that one wacky result (say, a bump in the floor, or an angle directly on the boundary between two angle measurements) can have a disproportionate impact on Trailerbot's learning.

As this cycle is repeated over and over again, the table will gradually grow to reflect reality more accurately. For example, the first time the robot is jackknifed and goes backward, it will begin to understand that "backing up when I am jackknifed leaves me straight again," and the more often that occurs, the more the

interaction will be strengthened. Similarly, the robot should also lea[illegible]at "going forward when I am pointed forward leaves me going forward." Ho[illegible]r, statements like these are only half the battle—the robot has to learn that it wants to go forward and avoid jack-knifing. This is where the Q scores come in.

Q scores

Technically speaking, Q scores are the "expected value" of an angle/movement pair. In other words, if the robot makes a movement from an angle, can it expect to be rewarded where it ends up? And what can it expect from the movement after that? This is a cool algorithm because it is farsighted: the robot looks not only at the immediate reward, but also attempts to understand what options it will have one movement into the future. Furthermore, because the algorithm is run over and over again, it is effectively recursive—it looks not only one turn into the future, but many turns. Maintaining the Q scores is the responsibility of `q_score_update()`, which is shown in Listing 8-7.

Listing 8-7. Maintaining Q Scores

```
void q_score_update()
{
  /*loop variables. Lots of them.*/
  int i, j, k, l;

  /*three variables for later*/
  float reward;
  float q_sum;
  float max_q_score;

  for(i=0; i<ANGLES; i++)
    {
      for(j=0; j<MOVEMENTS; j++)
        {
          /*are we doing a bad thing?*/
          if((i>=FAR_LEFT)||(i<=FAR_RIGHT)||(REVERSE==j))
            {
              reward = 0;
            }
          /*are we in "heaven?"*/
          else if(HEAVEN==i)
            {
              reward = HEAVEN_REWARD;
            }
```



```
          /*if not, we get rewarded normally*/
          else
            {
              reward = 1;
            }

          /*
           * This code "looks ahead" to see two things:
           * 1) What possibility do we have of getting to
           *    all possible angles?
           * 2) Once we get to those angles, what is the best
           *    possible outcome?
           * These two pieces of information are combined and
           * stored in q_sum.
           */

          q_sum = 0;

          for(k=0; k<ANGLES; k++)
            {
              max_q_score = 0;
              for(l=0; l<MOVEMENTS; l++)
                {
                  if(q_score[k][l] > max_q_score)
                    {
                      max_q_score = q_score[k][l];
                    }
                }
              q_sum += (steering_results[i][j][k]*max_q_score);
            }

          /*store the new expected q-score*/
          q_score[i][j] = reward+(GAMMA*q_sum);
        }
    }
}
```

Basically, this code has two parts. Each of these parts is called for every member of the `q_scores` array. Each member of the array represents a potential future combination of angle and movement. It doesn't matter whether or not the robot has been in that combination, we have to study it in case the robot needs to know about it in the future. The first part is quite straightforward: if the robot were to find itself in this position, would it be jackknifed or backing up? If either one is true, it doesn't get rewarded. Now we check to see if this position is in "heaven."

If so, we reward it with a very high value. If it is neither jackknifed nor in Heaven, it would have to be going forward, so it would get a reward of one. Once this is done the real meat of the algorithm occurs.

The reasoning behind this second part is tricky but effective. In the outer loop, the algorithm looks at all possible outcomes (the five angles). The inner loop finds the best possible Q score that could occur from each of those angles. The best possible score is then multiplied by the probability that the robot would end up in that particular angle. This way, if an angle would be impossible to get to in one move (say, the probability of reaching the left side jackknife from the far right side) then the robot more or less ignores the `q_score` of that location. On the other hand, `q_scores` that are very likely to occur (no matter how poor the reward) get weighted more heavily.

The weighted scores are then added to the reward to create the new `q_score` for that particular angle/movement pair. This is moderated by `GAMMA`: if `GAMMA` is large, the future becomes more important. If the future is very important, then the robot can be tempted to do interesting things. For example, one robot I watched while testing this program decided to jackknife itself repeatedly because it learned that it could reliably get from jackknifed to almost jackknifed. Because the future was so important, the fact that it didn't get a reward when it was jackknifed was mitigated by the high probability that it could get rewarded at the next turn.

We use the concept of "heaven" here because initially when the probabilities of going to all locations are equal, the Q score of "heaven" will be high and the robot will try very hard to reach if. To do this, it will explore and try new things; in the process it will gain more information about its environment. As it continues to explore, not only will it benefit from the new information it stores, it will also slowly realize that it will never get to "heaven." As it learns this fact (in other words, as the probabilities for reaching heaven that are stored in `steering_results` drop to zero), the robot will stop randomly looking for heaven and focus on what it now knows are "correct" behaviors with a high probability of a good outcome.

The `q_score_update()` function needs to be called repeatedly because each time it is, the algorithm looks farther into the future. This results in a more accurate depiction of reality (at least, as far as can be said of a model like this). `KAPPA` controls how many times it is called after each move(). `KAPPA` should be set to at least 10; higher values will create better results, though the effects may not always be clearly visible.

Running Trailerbot

When Trailerbot first runs, it prompts the owner to help calibrate the rotation sensor. This is very important: if the sensor is miscalibrated, the rest of the program will get bad data and function poorly. Begin with the trailer jackknifed to the right by turning it clockwise until it contacts Trailerbot's body, then start the program. The message "start at right" should appear on the LCD. After a slight pause, the

phrase "now center" will be displayed, at which point you should move the trailer to a centered position. After another brief pause the LCD will display what position it thinks the trailer is in. If the display shows the value "2" (the code for "centered") then calibration was successful, otherwise the program will have to be stopped and re-started.

Once that is done, the program does nothing other than call `move()` and `q_score_update()` repeatedly. At this point, the user really can't do much except sit back, watch, and perhaps occasionally move things out of the robot's way. This can be an exercise in patience: each set of calls to `q_score_update()` takes about ten seconds and learning usually isn't visible for a couple of minutes. Furthermore, this is not a completely reliable process. Because there is so much inherent randomness in the learning process, Trailerbot (and other robots that use reinforcement learning) often "learns" strange and interesting patterns. For example, sometimes Trailerbot will learn that traveling in a circle is a safe way to keep scoring points, especially if the mechanism and terrain make it difficult to travel in a straight line. Other times Trailerbot may get jammed in a jackknife for quite some time because it needs to randomly "guess" that backing up is good before it knows that it is a "good" thing in the long term. Because Trailerbot works by guessing randomly (at least initially), the robot may occasionally be unlucky and choose a series of movements that work poorly and don't allow it to learn. If this happens, the robot can look awfully silly.

Learning from Learning

Even the worst failures will often demonstrate some learning. Failures often repeat in the same patterns because they haven't yet stumbled into better opportunities. As you become more familiar with the algorithm, you'll spot these patterns more easily. For example, many times the Trailerbot will learn that going in a circle is a very stable way not to jackknife. Because it is rarely, if ever, punished when traveling in a circle, the circle gets strongly reinforced and Trailerbot never bothers to "discover" the joys of a straight line. Similarly, if `TURN_MULTIPLIER` is too low, or if the batteries are weak, soft turns may not move the robot at all. This may seem like a failure. However, to the robot this is a success: no movement means no jackknifing, thus the robot will probably quickly learn that anything other than a soft turn is a waste of time.

Furthermore, many of the robots will be "lucky" enough to learn the right patterns and learn how to correct themselves just as a real driver would. Watching these robots (particularly as they learn) is fascinating and will make your patience worthwhile. This is not a perfect program—there isn't a perfect AI anywhere yet, let alone on a system with only 32KB of RAM. Despite its imperfections, Trailerbot offers a glimpse into the possibilities that legOS alone can offer on the RCX platform through the extensive use of math and large arrays. Even if you don't write a

program as ambitious as the one used for Trailerbot, these strengths can be used with virtually any RCX robot to make it more flexible and powerful.

Going Further with LegOS—"Use the Source, Luke!"

I hope that these two chapters and the examples I have presented cover everything the average user might want to know about legOS. If you still want to learn more—perhaps to find the one feature that you know "must be there" or to add one that you'd really like to use—there is one last resource you already have at your fingertips: the source code to legOS itself.

Because legOS is an Open Source program, when you download legOS you get all the source code for it. Similarly, because the project began more than two years ago, thousands of other people have downloaded and used the legOS source. Many of them have written patches that have been incorporated into the source you are now using. More have likely been written between the writing of this book and the time you start to use legOS. Because of this constant change and breadth of features, one of the best things you can do to enrich your knowledge of legOS (after reading this book, of course) is to read the source code of the OS itself.

While "read the source" may sound intimidating for a beginning C programmer (or someone who hasn't had an Operating Systems course in a while) the legOS source code has many virtues that makes it fairly easy to read. Chief among these is brevity—there really isn't much source to look through. In this sense, legOS is quite efficient. If you are an experienced programmer and want to dedicate an evening or two, you can basically understand the entire OS from top to bottom. More importantly, for less experienced programmers, once you find the file you are looking for, legOS source is generally well commented and cleanly laid out. With the help of these comments you can quickly become a more proficient legOS programmer, even if you are just starting to learn C and are unfamiliar with the internals of operating systems.

NOTE *I use the phrase Open Source here as a reasonably unambiguous way to define the terms under which legOS developers and distributors allow you to use it. The phrase I prefer to use is Free Software—meaning you have the freedom to use, modify, and share the source of legOS without cost or restriction. To learn more about Free Software, a type of software that includes Linux and the GNU compiler legOS depends on, check out the homepage of the Free Software Foundation at http://www.fsf.org or their definition of Free Software at http://www.fsf.org/philosophy/free-sw.*

The "include" Directory

The `include` directory is the logical place to start for legOS newbies. This is where the various .h files live that you've been #including. It is also home to several interesting files I haven't yet covered. For example, one particularly useful file is `time.h`. This file includes the value `sys_time`, which is essentially an interface to the system clock. It starts at zero and is incremented in milliseconds to let you know how long a program (or more usefully, a function in a program) has been running. For example, I used it to time the efficiency of the floating point emulation mentioned in the "Math in LegOS" section earlier in this chapter. Similarly, the OS uses this value for many things, including `sleep()` and `msleep()`.

There are also interesting sub-directories in include. For example, the directory `include/c++` contains the start of a C++ interface for sensors. `c++.cpp` in the `demo` directory uses this interface and demonstrates how you can write and build a C++ program for your RCX. `include/rom` and `include/sys` contain some interesting low-level information and many hardware-related functions, such as ones that can turn off your RCX and check the RCX's batteries.

The LegOS Kernel

Once you've browsed through `include`, don't shy away from the kernel itself. While legOS is not a "real" operating system, poking around in its innards might help you better understand the behavior of your legOS programs. If you feel this bold, `kernel/kmain.c` is the logical place to start because it is what is first called by the hardware and must be the starting point for any attempt to trace the complete behavior of the OS.

Perhaps more relevant to most users is `boot/config.h`. Editing `config.h` allows you to directly control what functions are included in the kernel through the use of `#define` statements, which control `#ifdef` switches in the heart of the kernel. If you don't want a feature, just comment it out, type `make` in the `legOS` directory, and the system will attempt to build a kernel without those features. Be aware that "attempt" is the key word here—commenting out some of these options will prevent the kernel from being built properly. However, some other options are quite useful. For example, I find the little running man pretty irritating, so commenting out `CONF_VIS` gets rid of him. More seriously, the standard legOS kernel is 18KB, and judicious removal of options (particularly sound, which uses a large table of notes) can remove 3 or 4 KB. This could conceivably help a project squeezed for memory. Additionally, if you want to use LNP with more than one RCX, you'll want to look here because each robot's LNP address must be configured with `CONF_LNP_HOSTADDR`.

What about the Rest?

This section was not intended to be comprehensive of the legOS code base. Hopefully, though, it has covered the starting points for exploring the source and has made legOS easier to use for you. Who knows? You might be the next person to figure out something useful that legOS can do. You might even be the one who makes it more useful to others.