roduction to LegOS

LegOS, like NQC and pbForth, is an alternative programming environment for the RCX. Unlike the other environments, legOS is designed around the standard and popular C language, which has been used and refined for nearly thirty years. Markus L. Noga created legOS, and although there have been a number of other contributors, Markus wrote most of the code. Markus maintains the original legOS Web site at http://noga.de/legOS. However, legOS development has migrated to SourceForge (http://LegOS.sourceforge.net/) in order to support a larger community of developers. The SourceForge site is the central collection point for all of the newest legOS files, documents, and development work.

So Why LegOS?

Programming in C on the RCX has its tradeoffs. On the one hand, if you already know C, the combination of C and legOS offers you a great deal of flexibility, power, and efficiency that can't be matched by the other MINDSTORMS languages. For example, unlike the limited number of variables offered by the standard firmware, C provides an essentially unlimited number of variables. The programmer also has access to useful data types like matrices and arrays, along with the ability to reuse code that was originally written for other platforms.

In addition to these features of the C language, legOS supports your code by providing a number of library functions that allow direct access to the hardware, which gives the programmer a level of control that can't be matched by any of the other environments. When you aren't accessing the hardware directly, legOS provides many traditional operating system features, like preemptive multi-tasking and a networking layer. You can even generate random numbers so that your robot can be truly unpredictable. With the power provided by legOS, and the flexibility offered by C, you can develop virtually anything you dream of writing on a computer.

So What's the Catch?

There are a few catches to using legOS. First and foremost, you need to know or learn C. While the basics of the language are pretty straightforward, it can be difficult to take full advantage of because it is a complex and sophisticated language with many intricacies. If you don't know C, there are a couple of things you can try to help you learn it. Many programmers find that the best way to learn a language is to study code that someone else has written. Hopefully, the example code in this book is simple and clear enough to at least help serve that purpose. There are also a great deal of C tutorials online. I strongly suggest going to the Google directory listing at `http://directory.google.com/Top/Computers/Programming/Languages/C/Tutorials` and trying the newest links there.

If you prefer the more thorough treatment that a book can provide, *Practical C Programming*, by Steve Qualline, has been a favorite of mine. *Programming C: A Modern Approach*, by K.N. King, is also highly recommended by a variety of sources on the Web. Of course, once you've started, nothing beats writing lots of code and seeing what works.

Secondly, even if you know C, you'll still have to install and configure the tools necessary to compile and download your programs. This is getting easier (particularly on the Windows platform) but isn't seamless yet.

Unfortunately, if you aren't on Windows or Linux, it is not currently possible to run legOS at all. MacOS does not have the necessary command line environment, and complications with I/0 have made it difficult to port the download tools to other Unix platforms like Solaris. On the bright side, legOS is known to work on non-Intel versions of Linux (as a result of the standardized IO.)

So, How Does it Work?

Unfortunately, with power comes complexity. As a result, legOS is a system of many parts. It can be boiled down to three main pieces:

- OS
- Compiler
- Program loader

LegOS itself is basically a set of libraries that is compiled using gcc and downloaded to the robot with firmdl before you download your own programs. The compiler is a recompiled version of the GNU C Compiler (gcc.) This is used to compile the legOS and your programs into H8 assembly that the RCX can execute. Finally, there

are the firmdl3 and dll programs. Once you have used the compiler to create an executable out of your code, these two programs download legOS and your executable (respectively) from your computer to the robot.

Let's explore the interplay of these four tools (gcc, firmdl3, dll, and legOS) in more detail, by getting the robot ready to run with a sample program. This section assumes that you've already installed the necessary tools correctly and built the kernel, as described in Appendix B.

The first step is to use firmdl3 to download the kernel to the RCX. You can do that by executing the following command in your legOS home directory:

```
util/firmdl3 boot/legOS.srec
```

This should install the legOS kernel in your RCX. When it's done, you'll see the LCD man, and a dash (–) in the rightmost spot on the LCD. To make sure the kernel has been loaded correctly by the RCX, press the **Prgm** button. You should see "NONE" in the LCD. If not, it's possible that the download failed. If that is the case, try it again while adding `–s` after the firmdl3 command. This will be slower, but more reliable.

You can also use the following options with firmdl3 to change its behavior:

`--tty=TTY`: Set TTY to the correct port if you connect the tower to a different port than the default. The default is `/dev/ttyS0` under Linux and `COM1` under Windows.

`--fast`: Download quickly. This is the default.

`--slow`: Download slowly. If you Sare having problems with downloading, you may want to try this, as it is more reliable under a wider range of lighting conditions.

Table 7-1 specifies some of the most common error messages that firmdl3 will return in case of failure. Note that a few of these start with `/dev/ttyS0`. On your system, this will show whatever port firmdl3 was actually trying to contact: by default on Linux, this is `/dev/ttyS0`, but on Windows systems it will usually be `COM1`. Of course, if you use `–tty` to specify a different port, the error message will reflect this.

Table 7-1. Common Error Messages for firmdl3

ERROR	DIAGNOSIS
`/dev/ttyS0: Permission denied`	You do not have sufficient permission to write to the port. This may be the default under many Linux distributions. You will have to become root, and either use firmdl3 as root or execute the following command: "`chmod +222 /dev/ttyS0`".

ERROR	DIAGNOSIS
`/dev/ttyS0: Input/output error`	Most likely, the tower is not plugged into the specified port. May also indicate that the tower is low on batteries. To fix, either plug it into the specified port or use `–tty=TTY` to change the specified port (as mentioned earlier in this section).
`no response from RCX`	Either the RCX is still turned off, or possibly the tower battery level is low.
`unlock firmware failed`	Indicates slight corruption in the download. Use firmdl3 to download the firmware again.
`delete firmware failed`	Indicates that legOS is still present on the RCX. To remove it so that the download can succeed, either take out the batteries (in case of a hard freeze) or press the **Prgm** button while holding down the **On-Off** button.

Now that you have a kernel in the robot, let's test it by compiling a simple demonstration program:

1. Create a directory (I'll call it ch7) within your legOS directory.
2. Copy the Makefile from legOS/demo/ to your new legOS/ch7 directory.
3. Type the following listing into a file named simple.c within the legOS/ch7 directory:

```
/* simple.c */

int main(int argc, char *argv[])
{
  return 0;
}
```

4. Enter the ch7 directory, and type "`make simple.lx`". By default, this will look for the simple.c file and compile it into legOS's binary format. Once it is done, your ch7 directory should contain simple.o and simple.lx.

The .o file isn't particularly useful most of the time, but the .lx file is legOS's binary format—the file that actually gets downloaded to the RCX. When you are compiling

your own programs, all you need to do is create the program yourfilename.c, copy the Makefile into your directory, and the command "`make yourfilename.lx`" will compile it. It is important to note that this directory must be a subdirectory of legOS/. Otherwise, your compile will fail because of Makefile problems.

> **NOTE** *All of the sample programs in this book may be downloaded from the book's Web site at* http://www.apress.com [illegible]

If you are running Linux, you may want to put symbolic links to dll and firmdl3 in more convenient places than util/ . I find it useful to put a link to dll in the directory in which I write and compile my programs (ch7/ in this case) and a link to firmdl3 into my legOS/ directory to make it quicker to download a new kernel when I need one. If you have followed the build instructions for Windows, these should already be in your path.

Now that we've compiled a .lx file, it's reasonably straightforward to get the code into the robot. First, turn on the RCX and go to your legOS directory and type

```
util/dll ch7/simple.lx
```

This will load simple.lx into the first program slot. LegOS has eight program slots, and by adding `-pX` after dll, you can download your program into the desired slot. Once you've downloaded the program, hit the **Prgm** button until the digit on the right hand side of the LCD shows the number of the slot that you've downloaded the program into. (By default, this is slot zero.) Once you've selected it, hit the **Run** button, and the program will execute.

> **NOTE** *There is only one significant dll error: error deleting program. This is unfortunately a catch-all error, meaning that when you get it, it could be any of a large number of problems. The most important things to check are that your IR tower is correctly connected, the RCX is actually on (don't laugh—I've done it several times), and no program is currently running on the RCX.*

Now that we've compiled a sample program and downloaded it to the robot, let's look at the structure of the program itself.

This snippet is the simplest possible legOS code that will compile. As you can see, there isn't much to it. Like most C programs on a Unix system, every legOS program must start with a `main()` function that takes two arguments and returns an integer. Because the programs are called from within legOS (instead of the command line), the two arguments to main are actually never used. However, they

are still required, for internal compatibility purposes. Beyond this `main()` function, no other code is actually required in a legOS program. Nevertheless, you'll probably want to add a little bit more, so that the program will actually do something. The rest of this chapter will take a look at the various operating system functions you can use within your programs.

Basic LegOS Functions

The core of legOS is, of course, the library of functions that provide access to the various features of the RCX and the OS. For convenience, I'll divide these functions into three main categories:

- Output functions
- Input functions
- Program control

Outputs

Let's start by looking at code that will allow us to interact with the outside world. These are primarily the motors and the LCD. While the motors are probably essential to just about every robot you'll build with the RCX, it is also extremely important to learn how to use the LCD. It is basically your only tool for debugging legOS programs. As a result, if you want to write sophisticated programs, you'll have to understand the LCD and use it well.

Motors

Motor control is straightforward. There are two sets of functions: one set that controls the direction of the motors and another set that controls their speed. The direction functions are of the form `motor_x_dir(MotorDirection dir)`, where x is the letter of the motor (a, b, or c) and "dir" is one of four strings: off, fwd, rev, and brake. This is slightly different than what you may be used to, because legOS collapses *direction* and *mode* (in the language of Chapter 2) into the single concept of direction. In this notation, brake is the same as what Chapter 2 refers to as "off," while off is what that same section refers to as "floating." This mixing also means that, unlike the normal firmware, when a motor is switched to a different mode, the motor will forget its previous direction.

The speed functions are similar. They look like `motor_x_speed(speed)`, where x is again either a, b, or c, and the speed is a number between 0 and the constant `MAX_SPEED`. `MAX_SPEED` is 255 by default. So, unlike the standard firmware's 8 power

levels, legOS has access to 255 (a speed of 0 leaves the motor turned off). Generally speaking, this is overkill—on most surfaces, the 1/8 power increments that the normal software uses are completely sufficient and provide as much control as is necessary. However, this level of detailed control can be very useful as it allows you to linearly scale your power output to smoothly link math to outputs.

Let's look at a simple piece of example code in Listing 7-1.

Listing 7-1. motors.c

```
/* motors.c */
#include <unistd.h>
#include <dmotor.h>

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

  int k;

  /*start the motor*/
  motor_a_dir(fwd);
  motor_a_speed(MAX_SPEED);

  /*slow down the motor gradually*/
  for(k=MAX_SPEED;k>=0;k--)
    {
      /*slow the motor down a notch*/
      motor_a_speed(k);
      /*this function makes the robot wait for 20 milliseconds.*/
      /*more details on it later*/
      msleep(20);
    }

  motor_a_dir(off);
  return 0;
}
```

You'll note that this code steps the motor speed down in very small increments, which can't be done in the standard RCX environment. It also turns off the motor at the end of the program. This is unnecessary in this case, because the OS will

shut off the motors after the OS retakes control of the robot. However, it is a good habit to get into, because it is the end of the program (and not the function!) that turns off the motors. If you don't pay attention to this detail, your motors are likely to run until your robot goes into a wall.

NOTE: Many legOS functions are made available to your program through various header (.h) files. In order to use the functions you'll have to #include the correct file. Each section will have a note like this one near the top, noting which file you should include to get that functionality. To access the motors, for example, you should #include <dmotor.h>.

The LCD

Unlike the standard firmware, legOS allows direct control of the LCD. Every individual segment down to the arms of the running man can be turned on and off individually. In most cases, this level of control is unnecessary and higher level functions can be called from your program to take care of the details. However, the flexibility is there if you ever need it. The most important LCD functions are listed next, with the most frequently used functions first.

- `cputs(char *string)` is perhaps the most commonly used LCD function. Send it a string (no more than five letters, of course) and it will push the characters to the LCD. This is left justified—a single character printed with `cputs` will appear on the left side of the LCD.

- `lcd_int(int x)` pushes the int x to the screen. This can be very useful for debugging your code—for example, printing out the values of sensors. This is right justified. Note that this takes an int—if you pass it an unsigned int that is larger than the range of an int, you'll get strange behaviors.

- `lcd_clear()` Clears the LCD. While not always necessary, it can be a good idea to call this function before attempting to write something to the screen, since it doesn't necessarily overwrite the characters that were previously on the screen.

- `cls()` is similar to `lcd_clear()`, but clears only the LCD letters and not the walking man and other symbols.

- `cputw(unsigned int x)` outputs a value in hexadecimal notation. If you are comfortable with hex, then this can be useful, since it can represent values between 0 and 65,535. Whereas `lcd_int()` is effectively limited to four digits (-9,999 to 9,999) by the size of the LCD, and limited by the size of an int to roughly ±32KB. (More on this in the Advanced legOS chapter.)

- `dlcd_show(segment)` and `dlcd_hide(segment)` are very low-level functions that take as arguments the names of specific pieces of the screen, like LCD_ARMS, and turn those locations on and off. Each of the previous calls (`lcd_int()` and `cputs()`) are actually just functions that call variants of these two functions, which turn a large number of specific segments on and off for you. The complete list of LCD segment names is in the dlcd.h include file.

- `lcd_refresh()` is an older function that is still useful under certain circumstances. Technically speaking, the functions I've already described do not actually write to the LCD, but rather to a buffer in memory. In its default configuration, legOS automatically flushes this buffer to the LCD, but if you want more control, you can turn this feature off and control the flushing of the buffer to the screen yourself with `lcd_refresh()`. For more on how to configure the OS in this way, check out the advanced legOS chapter of this book.

The sample program lcd.c shown in Listing 7-2 contains examples of a couple of these features. First, we access a specific segment of the LCD—in this case, the arms of the walking man. Then we clear the screen and use a higher level call (`cputs`) to put two similar (but not quite the same) messages on the screen. Keep a close eye on the second two messages: you'll notice that H and O are the only letters in this group that actually change from lower to upper case. This isn't a bug, it's just a limitation of using text in a small LCD screen. Keep this limitation in mind if you have problems decoding the LCD later.

Listing 7-2. lcd.c

```
/* lcd.c */
#include <unistd.h>
#include <conio.h>

int main(int argc, char **argv)
{
  int k;
  /*make our man wave hello*/
  for(k=0;k<=5;k++)
```



```
  {
    cls();
    dlcd_show(LCD_ARMS);
    msleep(200);
    dlcd_hide(LCD_ARMS);
    msleep(200);
  }

 /*now he says hello*/
 cputs("hello");
 msleep(1000);

 /*now he says it differently*/
 /*notice: not too differently*/
 cputs("HELLO");
 msleep(1000);
 cls();

 return 0;

}
```

One thing to keep in mind as you write code is that in certain applications, if you aren't careful, successive calls to the LCD will quickly overwrite each other making it impossible to read any of them. For example, in a for loop, if you expect useable data to be outputted, it is probably insufficient to call "`cputs`" repeatedly. The RCX is fast enough that each loop will finish quicker than you can read the output of the previous loop. In fact, if you write and erase something fast enough (less than 10 ms) the item may never make it to the screen at all. To get around this, just use the `msleep()` call that I used in the motor example. As you'll see in slightly more detail later, that call puts your function to "sleep" for the specified number milliseconds. I find that it takes about 200 ms for a person to read a two-digit integer, but your experiences may vary.

One other important thing to remember is that under certain circumstances the OS will take over the LCD and print out its own messages. For example, the OS generally keeps the LCD stick figure walking in order to indicate OS activity. In addition, if you interrupt the program with the **Run** button, the OS will clear the screen. However, at other times when you might expect the OS to clear the screen

(like when the program exits!) it won't. That's why I use the `cls()` function at the end of this program to remove the "HELLO" from the screen and make it more obvious that control has moved to the OS.

Inputs

Now that we can interact with the outside world using the motors and gain insight into our robot with the LCD, we can also take a look at the input functions that allow a program to get data and extract meaning from the outside world and from the user. These functions control both the sensors that other programs have been using to get information and the buttons on the front of the RCX, which are customizable under legOS.

NOTE: To access the sensors, you should #include <dsensor.h>

Raw Sensors

Recall that each sensor returns a raw value that is then interpreted into something "meaningful." In legOS, the raw values are accessed directly by way of three constants: `SENSOR_1`, `SENSOR _ 2`, and `SENSOR_3`. The legOS kernel updates all three of these virtually continuously. This is different from the normal firmware, which samples the sensors at a fixed rate and can therefore miss changes that occur more quickly than the sampling interval. This allows programs that use legOS to respond more quickly to small changes in the environment. For example, legOS can read rotation sensors with reasonable accuracy at nearly 5000 revolutions per minute (rpm). The standard firmware can only do 1250 rpm, because it samples less frequently. Similarly, legOS users have found that they can build extremely fast line-following robots, because they can read and adjust to inputs at speeds at which an RCX Code-controlled robot would run off the line.

NOTE: When you use `SENSOR_x`, the value can be a decimal number that can have a value from 0 to 1023. 0x3ff is the hexadecimal notation of the decimal value 65 535. Because 0x3ff is very large, if you want to display the value of a sensor to the screen, use `sensor()` instead of `SENSOR_x`.

Touch Sensors

The constants `TOUCH_1`, `TOUCH_2`, and `TOUCH_3` each read 1 if the sensor is pressed and 0 if it is not pressed. These are just simple macros that act on the raw sensor values. Like the three raw sensor constants, these constants are continuously and automatically updated. It is important to remember that these variables are handled differently from touch sensors in the standard firmware. First, the OS doesn't need to be "informed" what type of sensor is attached to the sensor port. The touch sensor variables will return a simple boolean value (1 or 0, for true and false, respectively), no matter what is attached to the port—touch sensor, light sensor, or regular plastic brick. The second thing to remember is that unlike the complex sampling done by the standard firmware, legOS simply decides whether or not the raw value is above a certain threshold, and then reports that as the answer. In my experience, this works fine, because in most applications the touch sensor gets firmly depressed—when your robot runs into a wall, for example. However, should you wish to have finer control, you may want to use the raw values directly.

Using `TOUCH_1` in code is pretty straightforward. The touch.c program shown in Listing 7-3 uses it in a simple if statement. If the sensor is pressed, the value of `TOUCH_1` is 1, thus the if statement makes the man wave. Otherwise, his arms are hidden.

Listing 7-3. touch.c

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

/*make our man wave hello*/

int main(int argc, char **argv)
{
 while(1)
   {
     /*when we are touching the button, wave*/
     if(TOUCH_1)

       {
         dlcd_show(LCD_ARMS);
       }
```

```
    /*if we aren't, don't*/
    else
      {
        dlcd_hide(LCD_ARMS);
      }
  }
 return 0;
}
```

Light Sensors

The light sensor constants are (unsurprisingly) `LIGHT_1`, `LIGHT_2`, and `LIGHT_3`. Like the raw and touch sensor values, these are constantly updated values. They are scaled so that you'll get small integer values roughly between 50 (for dark) and 300 (for light).

Aside from the values of the lights, it is important to mention the functions that switch the light sensor between active and passive modes. To set the mode, use the functions `ds_active(&SENSOR_X)` and `ds_passive(&SENSOR_X)`, where X is the number of the light sensor. Make sure to use the &–if you don't, no error message will be generated and you'll wonder why your light is still on or off.

Listing 7-4 is a simple program that uses these functions to do a simple test on the brightness of the light source it points towards. If the light falls below the prescribed level, the program will output "dark," if it is brighter, it will show the actual light reading.

Listing 7-4. light.c

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

int main(int argc, char **argv)
{
 /*Turn on the sensor*/
 ds_active(&SENSOR_2);

 while(1)
   {
    /*test to see if it is dark out*/
    if(LIGHT_2<150)
```



```
      {
        cputs("dark");
        msleep(10);
      }
    /*if not, say exactly how bright it is*/
    else
      {
        lcd_int(LIGHT_2);
        msleep(10);
      }
  }

 /*go back to the OS*/
 cls();
 return 0;
}
```

As you can see, using the values is straightforward. Just treat the `LIGHT_2` value as you would any other variable and use the equal (=), less than (<), and greater than (>) operators to compare it to other values. The only difference from a normal variable, of course, is that instead of setting the value yourself, the OS sets it.

You'll note that the function `ds_passive()` is not used at the end of the program. LegOS takes care of this bookkeeping detail and does similar work for motors.

Buttons

Unlike the standard firmware, under legOS you can check the status of certain buttons on your RCX and use them for whatever purposes you'd like.

BUTTON NAME	LEGOS NAME
View	BUTTON_VIEW
Prgm	BUTTON_PROGRAM

With these two names (`BUTTON_VIEW` and `BUTTON_PROGRAM`), use `RELEASED(dbutton(), variable)` and `PRESSED(dbutton(), variable)` as functions. They'll return 1 if they are in the correct state (`PRESSED` or `RELEASED`) and 0 if they are not. Because `PRESSED` is the same as "not `RELEASED`" only one of these functions is necessary, but they are both included to allow you to ensure the clarity of your code.

It is important to remember that legOS does not debounce buttons when they are accessed in this way. You'll have to make your program wait for the bounce on its own. Alternately, if you don't want raw access to the buttons, there is a `getchar()` function in the dkey.h file. This function will wait until a single button has been pressed and return an integer that represents the button that was pressed. The relevant values are:

BUTTON NAME	GETCHAR() VALUE
View	4
Prgm	8

The `getchar()` function can't handle multiple button presses at once, so if you want to be able to use key combinations, you'll have to handle those with the raw button information.

NOTE *Under other versions of LegOS all four buttons could be accessed in the same way as View and Prgm. However, according to the LegOS source code, you'll note that the On-Off and Run buttons still have values associated with PRESSED() and RELEASED(), and as such, if you access the OS uses these keys, they are no longer available directly to the programmer in the same manner.*

Listing 7-5 shows how you might use these functions in a program.

Listing 7-5. button.c

```
/*button.c*/

#include <unistd.h>
#include <conio.h>
#include <dlcd.h>
#include <dsensor.h>
#include <dbutton.h>
#include <dkey.h>

int main(int argc, char **argv)
{
  int temp;
```



```
  /*wait until program is pressed*/
  while(RELEASED(dbutton(),BUTTON_PROGRAM))
    {
      cputs("prog");
    }
  cls();
  sleep(1);

  /*now wait until view is pressed*/
  while(RELEASED(dbutton(),BUTTON_VIEW))
    {
      cputs("view");
    }
  cls();
  sleep(1);

  /*wait until any button is pressed*/
  temp = getchar();

  /*show us which one got pressed*/
  if(8==temp)
    {
      cputs("prog");
    }
  else
    {
      cputs("view");
    }
  sleep(1);
  cls();

  return 0;
}
```

Quite simply, the program prompts the user to press a specific button. `RELEASED()` is then used to tell whether or not a specific button has been pressed. While those two loops are ongoing, pressing the other button will have no result because `RELEASED()` and `PRESSED()` are tied to the specified buttons. Once that is done, the program uses `getchar()`. At that point, either button can be pressed and the results returned.

NOTE *To access the button-related functions, you should #include <dbutton.h> and #include <dkey.h>.*

Program Control

If you've read through the various demo programs for input and output, by now you've seen a few examples of functions, such as `msleep()`, which are used for program control in legOS. We'll explain them here and discuss certain other functions.

> **NOTE** *To access the legOS control functions you should #include <unistd.h>. This is true for all the various control functions listed next unless specifically noted.*

The sleep Functions

As you saw in the LCD example, a common and useful pair of functions is the set of time functions, `sleep()` and `msleep()`. These functions each take an integer, and when called, put the program to sleep for the specified number of seconds or milliseconds. In a multi-threaded program, only the thread calling the function will be put to sleep. Otherwise, the whole program will wait until the allotted time has passed. You saw this behavior, for example, in light.c (Listing 7-4), where a small `msleep(10)` was used to prevent the LCD from flickering during extremely fast rewrites, even though the robot wasn't doing anything else during those "sleeping" periods.

It is important to note that `msleep()` shouldn't be used as a timer if exact time is important, because the OS waits the specified number of seconds and then executes the next line of code only after the current task is finished. For example, if you ran a bumper thread and a light-sensing thread at the same time (as we will do in the seeker.c program later in this chapter), an `msleep(50)` in the bumper thread would sleep for 50 ms and then wait until the light-sensing thread finished its task. In most cases, such a delay should be negligible, but under certain circumstances it might be important. As a result, it is safe to use these functions liberally throughout your code, but you will need to examine them more closely if you experience strange timing problems with threaded programs.

The wait_event() Function

The second important time management function to consider is the `wait_event()` function. This function is used to make a program wait until a particular event has occurred. For example, in the light seeker that we'll see at the end of this chapter, the robot has to wait until the bumper is touched. A `wait_event()` call is used for this, much like the `until()` function in NQC.

The function takes two arguments. The first is the location of a function (of type `wakeup_t`), which returns true or false. And the second argument is a string

that can be passed to that function. Calling `wait_event(my_function, data)` will call `wakeup_t my_function(data)` repeatedly, until `wakeup_t my_function(data)` returns a "true" (non-zero) value. Until the function returns true, the thread won't do anything except the `wait_event()`. Once a true value is returned, the thread can continue on its merry way.

As an example, let's look at Listing 7-6. This program waits until the touch sensor has been touched, then takes control of the robot.

Listing 7-6. wait.c

```
/*wait.c*/
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dlcd.h>

/*
 * we must take the argument,
 * but we don't have to use it
 */

wakeup_t touch_wakeup(wakeup_t ignore)
{
  return(TOUCH_1);
}

int main(int argc, char *argv[])
{
  /*a message from the robot*/
  cputs("touch");
  msleep(500);
  cputs("me");

  /*the event itself*/
  wait_event(touch_wakeup, 0);

  /*we are done*/
  cputs("yay");
  sleep(1);
  /*return to the OS*/
  cls();
  return 0;
}
```

As you can see, the `wait_event` call in this code calls a simple function that merely returns the value of the touch sensor. Once the button has been pressed, the program will stop waiting and "yay" will appear on the screen. While this example is a simple function (with an obvious use), `wait_event()` calls can have many other uses. For example, with the rotation sensors, you can wait until your robot has traveled a certain distance. Or, if you wanted to expand on the seeker.c program at the end of this chapter, you could wait until the light sensor passed a certain threshold and have the robot do a small victory dance. It is important to remember that you can pass values to a wait event: for example, you can pass a threshold value to a light-sensing `wait_event`, or pass a specific count to a rotation-sensing `wait_event`.

Threading with execi() and Friends

Finally, let's cover threading. As you've already noticed in the NQC portion of Chapters 3 and 4, it is difficult to write an interesting program on the robot if your programs can't figuratively walk and chew gum at the same time. LegOS programs accomplish this by using threads. A *thread* is basically a separate function or set of functions, which run side by side with other threads. Under legOS, threads are created by using the `execi()` function call. For example, the Seeker program used later creates one of its threads as follows:

```
driving_thread = execi(&basic_driver,0,0,PRIO_NORMAL,
                       DEFAULT_STACK_SIZE);
```

As you might tell from the example code, `execi()` takes five parameters. In most cases, only the first is important. This first parameter is the location of the function that you'd like to use as the separate thread. In the example, this is `&basic_driver`. Generally speaking, this is an ampersand (&) and the name of the thread function. Like `main()`, any function that is used as a separate thread must take two arguments—`int argc` and `char *argv[]`. You can use these parameters to pass information to a new thread, or you can ignore them, but either way they must be included in the declaration of the function. One other important note: the functions that you start the thread with (`basic_driver()` in this example) must be of return type `int`. Otherwise you'll get compiler errors.

The second and third arguments are the values to be passed to the new thread. In this case, I had no information to pass to `basic_driver()`, so the second and third arguments were both zero. If you do need to pass information to your new threads, the information you pass as the second and third arguments of `execi()` will be in `argc` and `argv` when the new function is called.

The fourth argument to `execi()` is the priority of the task. There are three things to keep in mind when assigning this number. First, the OS is not as efficient when

multiple threads have the same priority. So, keep these unique—only one thread should have priority one, only one thread should have priority two, and so on. Second, threads with a lower priority will get executed after threads with a higher priority. Because (generally) all threads always get executed, this isn't terribly important. The third point is the default set of priorities: `PRIO_LOWEST`, `PRIO_NORMAL`, and `PRIO_HIGHEST`. These are defined to be 1, 10, and 20, respectively. I have used `PRIO_NORMAL` here, but as long as you use positive numbers less than 20, you should be fine. This last note is important, since priorities equal to or greater than `PRIO_HIGHEST` may not be properly killed by the OS.

The fifth and final argument is the stack size in bytes. Under most conditions, it is best to use `DEFAULT_STACK_SIZE` for this, unless you have a very good idea of how much stack the thread is going to use and are in extreme need of a few extra bytes. If you don't know what stack is, don't worry about it—`DEFAULT_STACK_SIZE` (which is 512) is fine for all but the most memory-starved threads.

Once you've used `execi()` to create a new thread, there are a couple of things to keep in mind. First, `execi()` will return a process ID number in the form of an argument of type `pid_t`, which you should save to a global variable (for example, a variable declared outside of a function so that it is accessible to every function.) When the time comes to end that thread (say, as the result of a button press or light sensor activation), use the `kill(pid_t threadid)` function to kill the thread, passing it the process ID that you got from `execi()`. For example, the bumper thread uses `kill(driving_thread)` to end the light-seeking behavior after the bumper has been hit.

As of legOS 0.2.x, the kernel is preemptive in its multi-tasking. This means that the scheduler should automatically and regularly switch back and forth between the threads you have created in this fashion. If you want more precise control of your threads by explicitly giving control back to the kernel: `yield()`, `sleep()`, and `msleep()` explicitly return program control to the scheduler so that it can wake up the next thread. However, using these commands for this purpose should be unnecessary.

> **NOTE** *legOS also has support for semaphores, which allow communication between threads. If you are experienced with them, take a look through legOS/include/semaphores.h to see the interface that legOS uses for semaphores. It should be quite familiar to most experienced Unix programmers.*

The LegOS Seeker

Now that we've seen the basic building blocks of a legOS program, it's time to combine them into a complete program for the Seeker robot presented in Chapter 3. This robot won't be exactly the same as that Seeker bot, because it will use a couple

of legOS's features so to make it more interesting and capable. LegOS's superior memory handling will give our robot a sense of history, and the random function will make it slightly less predictable. However, integrating these things will make the robot behave slightly differently (particularly when it first starts), so don't be surprised if it doesn't work exactly like the NQC version does.

When you look at seeker.c (shown as Listing 7-7) one of the first things you may notice is the use of random() in the bumper code. The two calls to random() are used to generate random numbers, which allows a legOS-powered robot to actually surprise you. In this case, when the robot bumps into a wall, it "guesses" the better way to back up. Though it's not really that important in this application, for other uses, like genetic algorithms, random numbers are absolutely necessary. Used wisely, the element of unpredictability can make any robot more interesting.

Listing 7-7. seeker.c

```
/* seeker.c*/

#include <conio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>
#include <dlcd.h>
#include <time.h>

#define BUMPER !TOUCH_1
#define EYE LIGHT_2
#define HISTORY_SIZE 100
#define MAX_COMMAND 2
#define LOCATE_COMMAND 1

/*global process IDs*/
pid_t driving_thread;
pid_t bumper_thread;
pid_t light_thread;

/*A small array to record history*/
int local_history[10];

/*big history*/
int room_history[HISTORY_SIZE];
```



```
/*other useful universal variables*/
/*this is poor form but more reliable*/
long int threshhold = 0;
long int local_average = 0;

/*next turn direction*/
char next_direction;

/*dummy variable*/
wakeup_t dummy = 0;

/*
 *   bumper press functions
 */

wakeup_t bumper_hit_wakeup(wakeup_t data)
{
  return BUMPER;
}

wakeup_t bumper_release_wakeup(wakeup_t data)
{
  return BUMPER;
}

wakeup_t threshhold_wakeup(wakeup_t data)
{
  return(EYE>threshhold);
}

/*
 * 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();
}

void go_back()
{
  motor_a_dir(rev);
  motor_c_dir(rev);
  run_motors();
}

void spin_left()
{
  next_direction = 'r';
  motor_a_dir(fwd);
  motor_c_dir(rev);
  run_motors();
}

void spin_right()
{
  next_direction = 'l';
  motor_a_dir(rev);
  motor_c_dir(fwd);
  run_motors();
}

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

void change_direction()
{
  /*judging from next direction, turn*/
  if('l'==next_direction)
    {
      spin_left();
    }
```



```
  else
    {
      spin_right();
    }
}

/*this finds the index of the largest integer in the array*/
int query_array(int an_array[], int query_type)
{
  int k = 0;
  int max_value = 0;
  int max_index = 0;
  for(k=0;k<HISTORY_SIZE;k++)
    {
      if(an_array[k]>max_value)
        {
          max_value = an_array[k];
          max_index = k;
        }
    }

  /*if we want to get the location of the max*/
  if(LOCATE_COMMAND==query_type)
    {
      return max_index;
    }

  /*if we want to get the value of the max*/
  else
    {
      return max_value;
    }
}

/*move old values down, insert old value at 0*/
void local_history_update(int new_entry)
{
  int k = 0;
  for(k=9;k>0;k--)
    {
      local_history[k]=local_history[k-1];
    }
  local_history[0]=new_entry;
  return;
}
```

```
/*get the average of the array*/
int local_history_average()
{
  int total = 0;
  int k = 0;
  for(k=0;k<10;k++)
    {
      total+=local_history[k];
    }
  return (total/10);
}

/*
 * Here we record history and find the brightest spot.
 * Once found, we try to return to it..
 */

void history_calibration()
{
  int initial_max = 0;
  int k = 0;

  /*status report*/
  cputs("calib");
  msleep(100);

  /*here we spin for 3 seconds, taking light readings.*/
  spin_left();
  for(k=0;k<HISTORY_SIZE;k++)
    {
      room_history[k] = EYE;
      msleep(30);
    }
  stop_motors();

  /*find roughly where we found the brightest spot*/

  initial_max = query_array(room_history, MAX_COMMAND);

  lcd_int(initial_max);
  msleep(100);
```



```
  /*set the threshhold to a reasonable fraction of the max value*/
  threshhold = (initial_max*85)/100;

  spin_right();

  /*test to see if the local values are ~= max value*/

  wait_event(&threshhold_wakeup, dummy);

  /*We've found it, so let's announce that.*/
  stop_motors();
  cputs("found");
  sleep(1);

  /*what exactly did we find, anyway?*/
  lcd_int(threshhold);
  sleep(1);

  return;
}


/*
 * This function drives the motor by default.
 * Should be killed when another thread wants to take control.
 * Simple strategy:
 * If we are getting brighter or staying the same, we are OK.
 * Otherwise, we should sweep back and forth until we find a bright
 * spot again.
 */

int basic_driver(int argc, char *argv[])
{
  time_t sweep_start_time, current_sweep_length;
  int bright_spot_not_found;

  go_forward();

  while(1)
    {
      /*check the rolling average of readings*/
      local_average = local_history_average();

      /*are we getting brighter?*/
      if(EYE>=local_average)
```

```
  {
    lcd_int(local_average);
    local_history_update(EYE);
  }

/*uh-oh, we just got darker*/
/*lets sweep back and forth*/
else
  {
    stop_motors();

    /*get a new threshhold*/
    threshhold = local_average;

    /*we haven't found anything yet*/
    bright_spot_not_found = 1;

    /*initialize the length of the sweep to 1/10 second*/
    /*remember, it doubles each time, so it grows quickly*/
    current_sweep_length = 100;

    while(bright_spot_not_found)
      {
        /*prepare to do a scan*/
        cputs("scan");

        /*what time did we start at?*/
        sweep_start_time = sys_time;

        /*that last try didn't work, let's sweep the other way*/
        change_direction();

        /*sweep left or right for current_sweep_length ms*/
        while(sys_time < (sweep_start_time + current_sweep_length))
          {
            /*test to see if we are bright enough*/
            if(threshhold_wakeup(dummy))
              {
                /*we found it!*/
                cputs("found");
                bright_spot_not_found = 0;
                break;
              }
          }
```



```
            /*haven't found it, double the length of time we go back*/
            current_sweep_length *= 2;
            /*also, we'll reduce the threshhold by 10%*/
            threshhold *= 90;
            threshhold /= 100;
          }
        /*ok, so we've found the spot. Let's go.*/
        go_forward();
      }
    /*wait some time before sampling again*/
    msleep(100);
  }

  /*so the compiler doesn't complain*/
  return 0;
}

/*
 * This thread tests for the bumper and
 * takes control of the robot when that occurs.
 */

int bumper_driver(int argc, char *argv[])
{
  while(1)
    {
      wait_event(&bumper_hit_wakeup, dummy);

      /*we've hit something. Better stop moving.*/
      kill(driving_thread);
      stop_motors();

      lcd_clear();
      cputs("BUMP");

      /*let's back up a little bit*/
      go_back();
      msleep(500);

      /*pick a direction*/
      if(random()&(0x1))
        {
          spin_left();
        }
```

```
      else
        {
          spin_right();
        }
      /*wait to spin a reasonable distance*/
      msleep(250);

      /*now lets press forward again*/
      go_forward();
      msleep(250);
      stop_motors();

      /*restart the light seeking.*/
      driving_thread = execi(&basic_driver,0,0,PRIO_NORMAL,
                             DEFAULT_STACK_SIZE);
    }

  /*so the compiler doesn't complain*/
  return 0;
}

int main(int argc, char *argv[])
{
  /*initialize the light sensor*/
  ds_active(&SENSOR_2);

  /*initialize the random number generator*/
  srandom(LIGHT_2);

  /*initialize history and try to orient ourselves*/
  history_calibration();

  /*start the control threads*/

  driving_thread = execi(&basic_driver,0,0,PRIO_NORMAL,
                         DEFAULT_STACK_SIZE);
  bumper_thread = execi(&bumper_driver,0,0,PRIO_NORMAL+1,
                         DEFAULT_STACK_SIZE);

  return 0;
}
```

The difference between the behavior of this program and the behavior of the NQC-based Seeker at the beginning of the program can be attributed to legOS's superior handling of variables. Because we can store and access a lot of data instead

of making the user calibrate the robot at the beginning, the program causes the robot to circle for roughly three seconds and stores one hundred light readings into an array. It then backs up to find the location of the brightest spot it encountered. While not perfect, this gives the legOS robot a much better start than the pseudo-random walk of the NQC robot. Once it gets going, the robot keeps track of its last ten light readings. If the current reading dips below the average, the robot knows it is going into a darker spot and attempts to change direction by sweeping back and forth. Each time, it will look twice as far, doubling back upon itself to prevent turning all the way to one side or the other. These "doubling backs" (and in fact, all movement in the program) were calibrated on carpet, so if your robot moves too fast, simply find the `run_motors()` function and adjust the speed.

The two "history" arrays demonstrate two key powers of legOS: the use of multiple variables and flexible data structures. All robots need a sense of "history" if they want to move around well, and it is difficult to establish that without the data structures to organize information. This flexibility will become even more important as you add more sensors to your robot, and is very helpful for applications that need to do complex math, such as neural networks.

Conclusion

Hopefully, this chapter has served to show the basics of legOS. You should now be able to write solid legOS programs that use the basic functionality of the language. In the next chapter, we'll cover some advanced topics in legOS, such as using the rotation sensors and floating point math. Using the basics from this chapter plus the new ideas, we'll explore another interesting application for legOS—a learning robot.