Extreme MINDSTORMS™: An Advanced Guide to LEGO® MINDSTORMS™

DAVE BAUM, MICHAEL GASPERI, RALPH HEMPEL, AND LUIS VILLA

Apress™

Access Control

At times, a situation will arise where two separate tasks would like control over the same resource, and without some degree of coordination, the resulting behavior is likely to be surprising. For example, in the last chapter the bump and seek behaviors both needed control over the motors at certain times. Coordination was implemented in the main task (bump behavior) by having it explicitly stop and later restart the seek task. In some situations, this approach of starting and stopping tasks is a bit too severe.

The new firmware provides another option: *access control*. In general terms, access control allows a task to request ownership of a *resource*. If the resource is not owned by any task, the request is successful. If the resource is owned by a task with higher priority than the requestor, then the request fails. If, however, the resource is owned by a task with the same or lower priority than the requestor, then the request succeeds with the added consequence that the original owner loses their access to the resource. Besides requesting ownership, a task may also release ownership of resources when it is finished with them.

These basic abilities—requesting and releasing ownership—can be used to implement a kind of if/then logic for resources:

```
if ownership request succeeds then
    do something with the resource
    release resource
else
    do something to recover from failure or loss
```

Note that the RCX treats losing a resource the same as a failed request. In other words, any recovery code must be able to deal with two situations: the initial request failed, or the initial request succeeded, but the task later lost ownership (to an equal or higher priority task) before releasing the resource.

So far the discussion has been a bit abstract—it's time for some details. The firmware defines four physical resources:

- Motor A
- Motor B
- Motor C
- Sound

In NQC, these are represented by the constants ACQUIRE_OUT_A, ACQUIRE_OUT_B, ACQUIRE_OUT_C, and ACQUIRE_SOUND. There are also four user-defined resources that can be used:

- ACQUIRE_USER_1
- ACQUIRE_USER_2
- ACQUIRE_USER_3
- ACQUIRE_USER_4

The only difference between physical and user-defined resources is that when ownership of a physical resource is lost from one task to another, the firmware takes some default action (in addition to any recovery defined by the task that lost the resource). The default action for outputs is to turn them off, and the default action for the sound resource is to stop the currently playing sound and remove any pending sounds from the sound queue.

Each task has a priority, which ranges from 0 to 255 with 0 being the highest priority. This can be a bit counter-intuitive because lower numbers are actually higher in priority. I find it convenient to think of priority 1 as the first priority, 2 as the second priority, and so on. In NQC, a task can set its own priority with the SetPriority function:

```
SetPriority(0); // the highest priority
SetPriority(255); // the lowest priority
```

NOTE *It is imperative that any task using access control set its priority, otherwise, the initial priority of a task will be undefined and the results may be surprising.*

An NQC task may request a resource using the acquire statement, whose syntax contains the following :

```
acquire(resources) body [catch handler]
```

In this statement, acquire and catch are keywords, resources is a list of resources to acquire, and *body* and *handler* are statements. The catch and handler portion is optional. The acquire statement requests access to the resources, then executes the body, then implicitly releases the resources. If the request fails or if the resources

are lost to another task, the handler (if present) is executed. For example, the following code acquires outputs A and C and then waits 10 seconds.

```
acquire(ACQUIRE_OUT_A + ACQUIRE_OUT_C)
{
    Wait(1000);
}
ClearTimer(0);
```

In the preceding example, `ClearTimer` will always get called, regardless of whether the request fails, succeeds, or if ownership is lost during `Wait`. If the program needs to differentiate between normal completion of the acquire and a failure/loss, then a handler should be used as shown in the following example.

```
acquire(ACQUIRE_OUT_A + ACQUIRE_OUT_C)
{
    Wait(1000);
}
catch
{
    PlaySound(SOUND_CLICK);
}
ClearTimer(0);
```

At this point, if the request fails or ownership is lost, control will transfer to the handler and `PlaySound` will be called. In all cases, control will still eventually drop through to the `ClearTimer` call.

Access control provides a convenient way to coordinate Seeker's bump and seek behaviors from the previous chapter. Using the same functions and definitions from the original seekbump.nqc program, new main and seek tasks can be written that use access control rather than starting/stopping a task to coordinate the behaviors. These tasks are shown in Listing 4-2.

Listing 4-2. Tasks in seekbump_access.nqc

```
task main()
{
    setup();
    SetPriority(1);
    start seek;

    while(true)
    {
        until(BUMPER==0);
```

```
        acquire(ACQUIRE_USER_1)
        {
            avoid_obstacle();
        }
    }
}


task seek()
{
    SetPriority(2);

    while(true)
    {
        acquire(ACQUIRE_USER_1)
        {
            Wait(SEEK_DELAY);
            while(true)
            {
                until(EYE < threshold);
                find_target();
            }
        }
    }
}
```

NOTE *The program in its entirety is contained in the file seekbump_access.nqc, which you can download from the book's Web site at* `http://www.apress.com`.

Overall, the structure of both tasks is the same as before—the main task sets things up, then enters an infinite loop waiting for the bumper to be activated and then avoiding an obstacle. The seek task waits for the light sensor to drop below the threshold, then tries to find the target. Most of the time, the seek task owns the `ACQUIRE_USER_1` resource. However, once the bumper is hit, the main task requests ownership and because it is a higher priority task, the seek task loses ownership. The main task is then free to avoid the obstacle. Once the seek task loses ownership, it will continue in its `while` loop—each time requesting access of the resource. Only after the main task is finished avoiding the obstacle will it release the resource, thus letting the seek task re-acquire it and resume checking the light sensor and finding the target.

Events

All of the programs so far have explicitly checked their sensors using conditional statements such as `if`, `while`, or `until`. This is a relatively easy and straightforward way to construct a program, but it does have some limitations. For example, conditional statements begin to get unwieldy if the program needs to accommodate multiple stimuli (such as both a touch sensor and a light sensor). In the case of Seeker, introducing a second task so that each task would only have to watch a single sensor solved this problem. However, this approach had its own drawback since it added the complexity of coordinating the two tasks.

Conditional statements are also limited to testing the immediate state of a sensor (for example, "is the sensor pressed?"). In order to respond to more complicated stimuli (for instance, "has the sensor been held less than one second?" or "has the sensor been pressed twice?"), additional code would have to monitor the sensor and keep track of characteristics such as how long it was held or how many times it was pressed.

The new firmware provides another mechanism to react to stimuli: *events*. As their name implies, events represent something that can happen that is of interest to the program. For example, a light sensor that detects darkness, a touch sensor pressed for longer than one second, or a timer that exceeds the preset limit are all examples of potential events. The firmware can monitor up to 16 independent events. Each event is referred to by its *event number*, which is simply a number from 0 to 15. There is no implicit difference between any of the event numbers—a program may decide to use event 0 to watch a light sensor or event 7 to monitor a timer. The program is thus responsible for configuring the events—telling the firmware what condition should result in the triggering of the event.

Event Monitoring

Before digging into the details of how events are configured, it will be helpful to understand how they can be monitored from within a program. In many ways, event monitoring is similar to access control, but rather than requesting access to a specific resource, a task indicates that it wishes to have a certain set of events monitored. This request always succeeds. However, if a monitored event later becomes triggered, the program can have special handler code executed—this is similar to the handler code that executed when ownership of a resource was lost. In fact, event monitoring and access control are so similar that their syntax in NQC is nearly identical. The major difference is that instead of the keyword `acquire`, event monitoring uses the keyword `monitor`, and instead of a list of resources, it takes a list of events:

0< (n1 -- f) "zero-less-than"

Sets f true if n1 is less than 0, and false otherwise.

AND (n1 n2 -- n3) "and"

Sets n3 to the bitwise AND of n1 and n2.

OR (n1 n2 -- n3) "or"

Sets n3 to the bitwise OR of n1 and n2.

XOR (n1 n2 -- n3) "x-or"

Sets n3 to the bitwise XOR of n1 and n2.

INVERT (n1 -- n2) "invert"

Sets n2 to the bitwise inverse of n1.

Basic RCX Sensors and Motors

Now we have almost enough material to do something interesting with a touch sensor and a motor. This chapter is only an introduction, so we don't go into much detail describing the words to control the RCX and its systems. The next chapter describes the words in more detail because we go through the exercise of designing the Seeker robot there. To do that, we need to know a lot more about how things work inside the RCX. Here is a glossary of the RCX words we use in the script.

MOTOR_SET (power mode idx --) "motor-set"

Sets the motor specified by an index of 0-2 to a power between 0-7. The allowed modes are:

```
1 forward
2 backward
3 stop
4 float
```

SENSOR_INIT (--) "sensor-init"

Initializes the sensor subsystem.

SENSOR_PASSIVE (idx --) "sensor-passive"

Sets the specified sensor to be a passive type. The touch and temperature sensors are passive.

SENSOR_TYPE (type idx --) "sensor-type"

Sets the sensor type to one of the following:

```
0 Raw sensor (useful for custom devices)
1 Touch sensor
2 Temperature sensor
3 Light sensor
4 Rotation sensor
```

SENSOR_MODE (mode idx --) "sensor-mode"

Sets the specified sensor to one of the following modes. Use common sense when combining different modes and types according to the following list:

```
0x0000 Raw mode
0x0020 Boolean mode
0x0040 Edge detection - every transition counts
0x0060 Pulse detection - only negative transitions count
0x0080 Percent of scale
0x00A0 Degrees Celsius
0x00C0 Degrees Fahrenheit
0x00E0 Angle detection
```

SENSOR_READ (idx -- code) "sensor-read"

Reads the specified sensor and returns a flag indicating success (0) or busy (other values).

SENSOR_BOOL (idx -- bool) "sensor-bool"

Retrieves the sensors current boolean data field. Note that this item will not change unless you actually call SENSOR_READ on the sensor first.

Now we'll use these words in the simple little script shown in Listing 5-1, which changes the motor's direction based on which buttons are pressed. It assumes we have a touch sensor on Port 1 (which we index as 0) that controls whether the motor is on or off. The other touch sensor is on Port 2 (which we index as 1) and it determines whether the motor turns forwards or backwards. The motor is on port A.

NOTE *Sample programs used in the book can be downloaded from the book's page on the Apress Web site at* `http://www.apress.com`.

Listing 5-1. MOTORDIR.TXT–Controlling a motor with a touch sensor

```
\ MOTORDIR.TXT
HEX

: HANDLE_SENSORS_INIT
  RCX_INIT
  SENSOR_INIT
     0 SENSOR_PASSIVE
   1 0 SENSOR_TYPE
  20 0 SENSOR_MODE
     1 SENSOR_PASSIVE
   1 1 SENSOR_TYPE
  20 1 SENSOR_MODE ;

: HANDLE_SENSORS
  7 ( default power level )
  0 DUP SENSOR_READ DROP SENSOR_BOOL ( on or off? )
  IF   1 DUP SENSOR_READ DROP SENSOR_BOOL ( fwd or rev? )
       IF 1 ELSE 2 THEN
  ELSE 3 THEN   ( brake the motor )
  0 MOTOR_SET ;

HANDLE_SENSORS_INIT
HANDLE_SENSORS
```

This is a good example of how Forth encourages use of the stack for parameter passing instead of storing transient values in variables. The `MOTOR_SET` word requires the power level, mode, and motor index on the stack. The `HANDLE_SENSORS` word computes the parameters in that order. The state of the sensor at index 0 determines whether the motor is on or off (mode 3). If it is on, the state of sensor 1 determines whether the motor should turn forwards (1) or reverse (2). Finally, we can just put the motor index of 0 on the stack before calling `MOTOR_SET`. To test the word, type `HANDLE_SENSORS` while leaving the touch sensors in their proper state. This is a bit awkward, so type the word, then press the touch sensors with the fingers of one hand and hit the return key on the computer with the other.

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/semaphore.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];
```

skills such as reading schematics, soldering, and packaging. We will go step-by-step in detail through the building process so that hopefully everyone can build the projects with success.

As with the passive sensors, if you have trouble making or understanding the sensor projects, there are many places to get help. You should be able to find help from teachers or people that work in electronics stores and repair shops. Your local library and bookstores should have beginner's books on electricity too.

We will use a two-step process in building the projects. The first step is to build and test the complete circuit on an electronic breadboard. The breadboard allows you to construct electronic circuits without soldering, which makes correcting mistakes easy. The second step is to transfer the parts from the breadboard to a printed circuit board with the same layout as the breadboard and solder them in place to make the project permanent.

RCX Powered Interface

The RCX powered interface must supply power to the sensor and read its value using the same pair of wires. It accomplishes this by rapidly alternating between these two functions. First, the RCX applies about 8 volts to the sensor port, much like it applies voltage to the motor outputs. It does this for a short 3 milliseconds or a 0.003-second time period. It then measures the sensor value the same way it does for a passive sensor —for an even shorter 0.1 millisecond or 0.0001-second time period. After that, it repeats the process over again. The challenge in building a powered sensor is to separate these two functions. On top of that, it would be nice if it didn't matter which direction the LEGO connector wire was attached to the RCX.

Sensor Power Circuit

The circuit shown in Figure 10-1 gets power to the sensor. The parts labeled D1 to D4 are called *switching diodes.* They act as one-way valves for electricity and their industry standard part number is 1N4148. The diode symbol looks like an arrow pointing at a line, and the electric current can only flow through the diode in the direction of the arrow. The real diode has one end with a painted stripe, which corresponds to the side with the line in the diode's symbol. The 1N4148 is similar to another diode called the 1N914 and either can be used for these projects.

NOTE *On a circuit diagram, dots represent electrical connections. For example, the vertical wire connected to both D1 and D2 is NOT connected to the horizontal wire that goes to the RCX.*

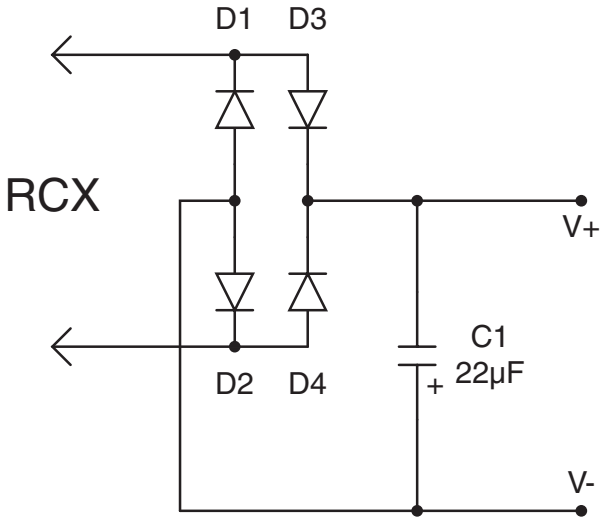

Figure 10-1. A power circuit diagram

The arrangement of four diodes is called a *bridge rectifier,* which is normally used to convert alternating current (AC) into direct current (DC). This may seem confusing because the RCX does not produce AC. The circuit is used here only so that the sensor will work regardless of how the LEGO connector wire is attached to the RCX. Imagine the positive terminal of the RCX attached to the top connector on the circuit diagram and the negative terminal attached to the lower, as shown on the left in Figure 10-2. Electrical current cannot flow into D1 because it is not oriented correctly, but it can flow into D3. The current then flows into whatever sensor circuitry you have and out through D2. If the RCX polarity is reversed, D4 and D1 pass the current.

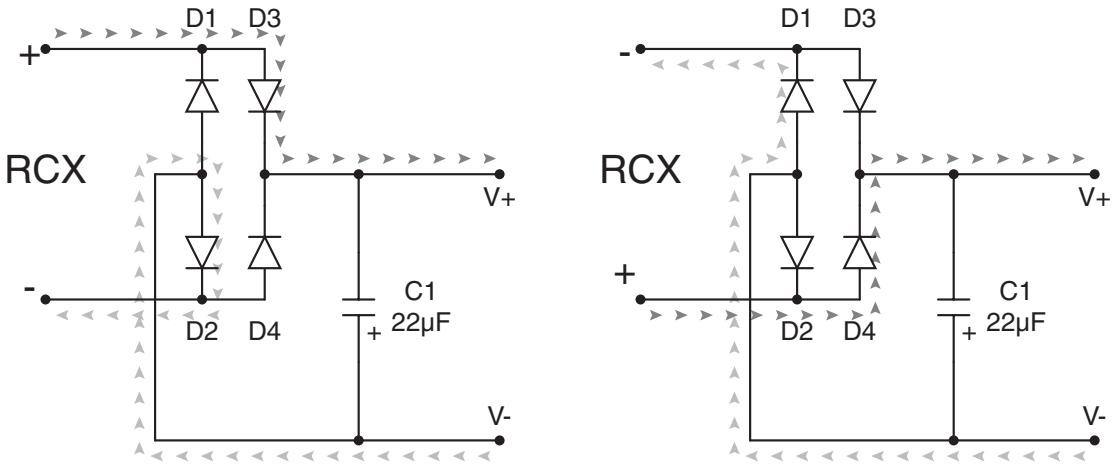

Figure 10-2. Current flow through the power circuit

The part labeled C1 is an aluminum electrolytic capacitor. It acts like a small rechargeable battery and, just like a battery, it has positive and negative terminals. On the real capacitor only the negative end is usually marked. Two things are needed to specify a capacitor like this. Its capacity of 22 microfarads (abbreviated µF) defines the size of the capacitor, much like AA, C, and D define the size of a battery. The rated voltage of 35 V defines the maximum voltage the capacitor can be charged without damage. For this circuit, the capacitor will never be charged over 10 V, however, 35 V is the lowest value Radio Shack retail stores carry. You can always use a part with a higher voltage than you need.

Now let's put these parts on the electronic breadboard. As you can see in Figure 10-3, the breadboard has several vertical columns of five electrically connected holes (labeled A-E and F-J), along with two long rows of connected holes (labeled X and Y) at the top and bottom. If your breadboard is not labeled like this, make your own labels and stick them to the edge of the board. The columns are numbered starting from 1, but we skip column 1 to make the eventual layout on the printed circuit board easier. Inside the holes are spring contacts that pinch the component leads or wires to make the electrical connections.

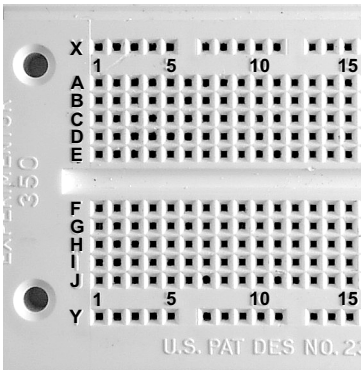

Figure 10-3. Labels on the electronic breadboard

IPoke the part leads into the breadboard until they hit bottom. The lead will extend about 1/4 inch or 6 mm below the surface of the board. Look at Figure 10-4 and Table 10-1, which gives the hole locations to determine the placement of each part. It helps to trim the length of the part leads so they don't stand too high off the breadboard. The RCX wires need to be attached to short pieces of solid #22 gauge hookup wire so they can be plugged into the breadboard too. Strip enough insulation from the wire so that it can be inserted fully into the breadboard.

NOTE *Occasionally, parts are so close that their leads bump together and create a short circuit. To prevent this, strip insulation from hookup wire and slip the insulation over the part leads. For example, in Figure 10-4 insulation is used on the negative lead of C1.*

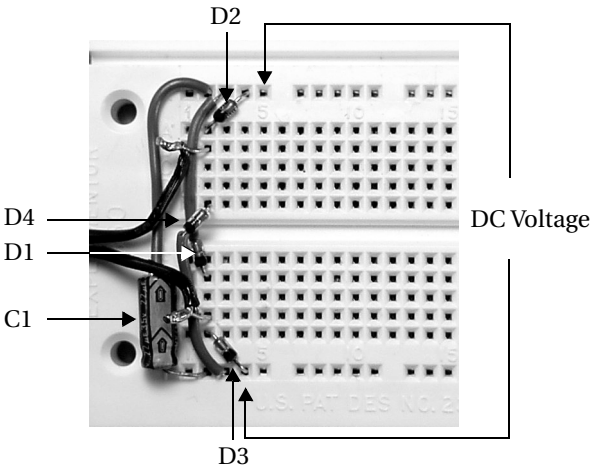

Figure 10-4. The power circuit on the breadboard

Table 10-1. Power Circuit Part Placement Table

PART	COLUMN	ROW
D1 stripe 1N4148	2	G
D1 plain	3	X
D2 stripe1N4148	2	A
D2 plain	4	X
D3 stripe1N4148	4	Y
D3 plain	2	J
D4 stripe1N4148	3	Y
D4 plain	2	E
C1 + 22uF	2	Y
C1 -	2	X
RCX 1	2	B
RCX 2	2	I