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

ISBN (pbk): 1-893115-84-4

Printed and bound in the United States of America 12345678910

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson

Technical Reviewers: Dave Baum, Rodd Zurcher

Projects Manager: Grace Wong

Developmental Editor and Indexer: Valerie Perry

Copy Editor: Kiersten Burke

Production Editor: Janet Vail

Page Composition and water braiding: Susan Glinert

Artist and Cover and Part Opener Designer: Karl Miyajima

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010

and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER; orders@springer-ny.com; http://www.springer-ny.com

Outside the United States, contact orders@springer.de; http://www.springer.de; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710

Phone: 510-549-5931; Fax: 510-549-5939; info@apress.com; http://www.apress.com

CHAPTER 4

RCX 2.0 Firmware

IN THE FALL OF 1998, LEGO released their first MINDSTORMS set: the Robotics Invention System (RIS) 1.0. The following year, a new version of the set (RIS 1.5) was released. This new version featured a number of improvements to the RCX Code software, to the documentation included with the set, and it included a few new pieces. The RCX itself was slightly different physically (it had no external power connector), but the firmware remained unchanged.

LEGO is planning another update to the RIS. As of this writing, LEGO has not announced an official release date, but an educated guess is that RIS 2.0 will be available early in 2001. Most of the specifics of RIS 2.0 are also unknown, but one thing is certain: the RCX will get new firmware. Furthermore, this new firmware can be used with existing RCX's, which is good news for owners of RIS 1.0 and RIS 1.5. As an added bonus, this new firmware is available today (albeit in pre-release form). In March of 1999, LEGO made an early version of this firmware available to the public as part of the RIS 2.0 pre-alpha SDK (Software Development Kit). Among other things, the SDK contains the RCX 2.0 Firmware, which can be easily downloaded to any RCX.

The RCX 2.0 Firmware adds a number of new capabilities to the RCX. Presumably, the programming environment for RIS 2.0 will utilize these new capabilities. However, you can get started today by using either the tools contained in the SDK or by using NQC. This chapter will explain some of the major new features and illustrate using them in NQC.

Getting Started

In order to start working with the RCX 2.0 Firmware you will need two things: the firmware itself, and a version of NQC that supports the new features (version 2.2 or higher).

The firmware was initially made available as part of the RIS 2.0 SDK, which can be downloaded from the MINDSTORMS Web site at `http://www.legomindstorms.com`. However, Web sites change much faster than books, so the specifics of getting this software from the MINDSTORMS site may change. Future MINDSTORMS products will also include the new firmware, so if you are an RIS 2.0 user you already

have the new firmware and it has probably been downloaded to your RCX. The firmware is contained in a file named FIRM*xxxx*.LGO, where *xxxx* is the firmware's version number. For example, the original firmware was version 3.0.9 and thus named FIRM0309.LGO. As of this writing, the most recent version of the new firmware is FIRM0328.LGO (version 3.2.8).

The new firmware is compatible with the original firmware, so all of the existing tools and programs for the original firmware should also work with an RCX running the new firmware. However, to take advantage of the new features using NQC, you will need NQC 2.2 or higher. The latest version of NQC can be found at `http://www.enteract.com/~dbaum/nqc`. If you want to determine the version of NQC you are currently using, just type the following command:

```
nqc
```

In order to download firmware to the RCX you should use either the -firmfast or -firmware options for NQC. As its name implies, -firmfast will be faster, but it is also more susceptible to interference. In general, try firmfast. If that doesn't work, step back to -firmware. You will also need to specify the firmware file to be downloaded. For example, if the firmware file was named FIRM0328.LGO and located in your current directory, you can use either of the following commands to download it:

```
nqc -firmfast FIRM0328.LGO
nqc -firmware FIRM0328.LGO
```

After a successful firmware download, NQC will query the RCX and report the versions of the ROM and the firmware. For example, after downloading FIRM0328.LGO you may see something like this:

```
Current Version: 00030001/00030208
```

This indicates that the ROM version is 3.0.1 and the firmware version is 3.2.8. The features discussed in this chapter apply to firmware versions 3.2 and higher.

When compiling a program with NQC you need to tell it if you want to generate code for the original firmware or for the new RCX 2.0 Firmware. You can do this with the -T<*target*> option, where <*target*> is a name that specifies where you plan on running the program. NQC supports a number of different targets, but the most important ones (for RCX owners) are `rcx` and `rcx2`, which specify, respectively, an RCX running the original firmware and an RCX running the 2.0 firmware. If no target is specified, the default is the original RCX, although this may change in later versions of NQC.

> **NOTE** *If you are using an IDE (such as RcxCC or MacNQC), you may be able to specify the target using preferences within the IDE itself. If the IDE doesn't support the rcx2 target, you may still be able to use the NQC_OPTIONS environment variable to set a default target for NQC to use. Whenever NQC is launched, it first reads the NQC_OPTIONS environment variable and acts as if the options specified in this variable were also typed on the command line. By setting NQC_OPTIONS to "-Trcx2," NQC will default to targeting the new firmware.*

Here is a simple program that uses a couple of the new firmware features. First, it uses the `SetUserDisplay` function to write a value to the LCD. Second, it uses `FirmwareVersion` to determine the current firmware version.

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

```
// firmcheck.nqc

task main()
{
    SetUserDisplay(FirmwareVersion(), 2);
    while(true);
}
```

You can compile and download this program to the RCX using the following command:

```
nqc -Trcx2 -d firmcheck.nqc
```

If NQC reports an error during compilation, either you made a mistake typing in the program or you forgot to tell NQC to target the new firmware (-Trcx2). At this point, if you run the program you should see the firmware version displayed on the LCD. For example, when using firmware 3.2.8, the display shows "3.28.". Note that this program depends on new firmware features; if you try to run it on an RCX with the original firmware, the display will remain unchanged. We're now ready to explore some of the special features of the new firmware.

Local Variables

One of the most limiting aspects of the original firmware is that it only provides thirty-two storage locations that must be shared between all of the tasks in a program. The primary use for this storage is for variables in an NQC program, but they are also used whenever an NQC program needs a temporary variable to hold an intermediate result of a calculation. The new firmware provides sixteen additional storage locations, and unlike the initial thirty-two which are *global* (shared across all tasks), the sixteen new locations are *local* to a task. This means that each of the RCX's ten tasks has its own set of sixteen local locations (for a grand total of 32 + 16 x 10 = 192 storage locations).

The terms *global* and *local* have slightly different meanings when applied to RCX storage locations versus NQC variables. When discussing a storage location, the distinction between global and local locations is simply that global locations may be shared across multiple tasks, while local locations are private to each task. When discussing NQC variables, the meanings are somewhat different. Variables that are declared outside of a task, function, or subroutine are global variables, which means that they can be accessed from anywhere in the program. Variables declared within a task, function, or subroutine are local variables and can only be accessed within the block that declares them. The following code illustrates how global and local variables may be declared and where they can be accessed.

```
int g; // g is a global variable

task main()
{
    int x; // x is a local variable
    while(true)
    {
        int y; // y is another local variable
        // x, y, and g can all be accessed here
    }
    // x and g can be accessed here, y cannot
}

task foo()
{
    // only g can be accessed here
}
```

When NQC compiles a program it must assign each variable to a specific storage location. In general, global variables must be assigned to global locations, but local variables can use either global or local locations. It is important to note that NQC still enforces the restrictions for access to local variables even when they are

assigned to global locations. In the previous example, task `foo` would not be able to use variables `x` and `y` even if they were placed in global locations.

Because the firmware's new locations are local to a task, NQC is able to re-use the locations more efficiently and can even generate code for certain cases that were impossible when using the original firmware. Using these new storage locations couldn't be easier; NQC automatically uses local storage locations, as appropriate, when compiling a program for the RCX2.

To realize the greatest benefit from the new storage locations, it is a good idea to use local variables instead of global ones whenever possible. This gives the NQC compiler the freedom to put the variable in either global or local storage. It can thus try to maximize the total number of variables available to your program. When assigning locations for local variables, NQC will generally attempt to use local locations first and resort to global locations only after the local ones have been exhausted for a given task. In the following code, variable `g` must be assigned to a global location, while variable `x` is declared such that the compiler can use whatever storage space happens to be available (local or global).

```
int g; // g will have to use a global location

task main()
{
    int x; // x can use local or global location, NQC will pick
}
```

Display

With the original firmware, a program's control over the LCD was somewhat limited. The default mode was for the LCD to display the value of the RCX's watch (its internal clock). Other modes allowed the current value of a sensor or output to be displayed instead. However, the ability to display an arbitrary number or monitor a variable was missing.

The new firmware adds one more display mode: user display. In user display mode, the LCD can display the value of any RCX data source (for example, sensors, timers, and so on). The most obvious use for this mode is to display constant numbers. However, the mode can also be used to display the contents of dynamic sources, such as variables or timers. NQC supports this new feature with a single function: `SetUserDisplay(source, decimal_point)`. The first argument is the data source to display; the second argument specifies where the decimal point (if any) should be drawn.

```
SetUserDisplay(123, 0);  // displays "123"
SetUserDisplay(123, 2);  // displays "1.23"
SetUserDisplay(Timer(0), 0); // display the value of timer 0
```

The behavior of displaying a timer may surprise you. At first glance, it looks like `Timer(0)` will be evaluated, and its current value will then be passed to `SetUserDisplay()`, presumably to be written to the LCD. In NQC, however, things operate a little differently. The expression `Timer(0)` is passed by reference to `SetUserDisplay()`. When the display is in user mode, it continuously reads and displays the data source (in this case `Timer(0)`). Run the following program to see this effect in action.

```
// viewtimer.nqc

task main()
{
    ClearTimer(0);
    SetUserDisplay(Timer(0), 0);
    until(false);  // keep the program running
}
```

The RCX normally returns to the default display mode (the watch) whenever a program ends. In the preceding program, the `until` statement prevents the program from finishing, thus allowing the display to stay in user mode. Note that the RCX will only reset the display mode when a program ends normally—interrupting the program by pressing the Run button will leave the display mode unchanged. Thus the value of `Timer(0)` in this case would continue to be displayed even after the program was stopped.

The fact that the RCX will continuously update the display in user mode has some interesting consequences. First, it must be possible for the RCX to read the value to be displayed. Because the LCD does not belong to any specific task, local storage locations (which are only valid within their own task) cannot be used. If you want to display the value of a variable, you generally need to make the variable global (by declaring it outside a task, function, or subroutine).

Another consequence of displaying a variable is that you may occasionally see its value partway through a calculation because it is continuously being updated on the LCD. For example, the following program gives the impression that it will always be showing the value of 1 plus timer 0. However, if you run the program you will notice that although the timer continues to count upwards, every once in a while the value 1 is shown on the LCD. This is because sometimes the LCD grabs the value of x in the middle of the calculation `x = 1 + Timer(0)`.

```
// display_bad.nqc

int x = 0;

task main()
{
    ClearTimer(0);
    SetUserDisplay(x, 0);
    while(true)
    {
        x = 1 + Timer(0);
    }
}
```

This problem can be fixed by latching the calculated value into a separate global variable that actually gets displayed, as shown in the code that follows. Even though the value of x moves around during the calculation of `x = 1 + Timer(0)`, the LCD will always show the correct value since the update to `displayValue` happens in a single operation. Since x is no longer displayed directly, it can be moved to a local variable.

```
// display_ok.nqc

int displayValue = 0;

task main()
{
    int x;

    ClearTimer(0);
    SetUserDisplay(displayValue, 0);
    while(true)
    {
        x = 1 + Timer(0);
        displayValue = x;
    }
}
```

One of the best uses of the display is to assist in debugging a program. In cases where you want to inspect a variable's value, just insert the appropriate call to `SetUserDisplay` in your program. Another useful technique is to display constant values at various points in the program. Then by observing the value on the LCD you can tell what part of the program was last executed.

Arrays

Not only does the new firmware provide more variable storage, but it also provides a new way to access the storage: *arrays*. An array is a set of storage locations that share a common name. The individual locations are then accessed by their position—called the *index*—within the array. In NQC, arrays must be of a fixed size. In theory, the maximum size for an array is thirty-two. However, such a large array would use up all of the global storage locations, leaving no room for additional global variables. In general, arrays should be kept as small as possible because storage locations are such a precious resource. Declaring an array is similar to declaring a normal variable except that you must also specify the size of the array. For example, the following statement declares an array named `my_array` that can hold four values:

```
int my_array[4];
```

The size of the array can involve a calculation, but must be constant at compile time, as shown by the two declarations (one legal, the other an error) that follow.

```
int my_array[2 * 3]; // ok - compiler knows that 2 * 3 = 6
int x;
int bad_array[x]; // error - x is not constant!
```

The values inside an array are called *elements* and are accessed by specifying the array name and the index of the element. In NQC, the first element has an index of 0. At first, this may seem a bit strange—shouldn't the first item be number 1? However, arrays in C are zero-based (first element has an index of 0), and NQC uses this same model. Don't worry, after a while you'll get used to asking for element 0. The following code illustrates setting and reading elements of an array.

```
my_array[0] = 123; // set first element to 123
my_array[1] = my_array[2]; // copy third element into second
```

Unlike the size of an array, which must be constant, indexes do not have to be constant. This is where the true power of an array comes in because it allows a set of statements to operate on any (or all) elements in the array. For example, the following code declares an array of five elements and sets them all to 0:

```
int my_array[5];
int i;
for(i=0; i<5; ++i)
    my_array[i] = 0;
```

Note that NQC does not check to see if an index is within the bounds of the array. In the previous example, if the array were declared with a size of 4, the code would still compile without any errors. The result of running the program, however, would be undefined since it would be attempting to set element 5 of an array that only contained 4 elements.

The `for` loop that sets the elements to 0 is certainly more compact than writing five separate assignment statements. At first arrays may just seem like a fancy way to save a little typing, but they also allow functionality that would be impossible (or at least difficult and obscure) with ordinary variables.

As a more complex example we'll write a program that starts by taking several readings from a light sensor and then computes their average. Then as new samples are taken, the oldest one will be discarded and a new average will be computed. The key to this entire program is to store the samples in an array. The complete program is shown in Listing 4-1.

Listing 4-1. average.nqc

```
// average.nqc

// sensors
#define BUTTON    SENSOR_1
#define EYE        SENSOR_2

// number of samples
#define N    3

// average needs to be global for display
int average = 0;


// this function waits for a press and
// release of the touch sensor

void wait_for_touch()
{
    until(BUTTON == 1);
    until(BUTTON == 0);
    PlaySound(SOUND_CLICK);
}
```

```
task main()
{
    int index;
    int current;
    int samples[N];
    int total = 0;

    // setup sensors and display
    SetSensor(BUTTON, SENSOR_TOUCH);
    SetSensor(EYE, SENSOR_LIGHT);
    SetUserDisplay(average, 1);

    // start with N samples
    for(index=0; index<N; ++index)
    {
        wait_for_touch();

        current = EYE;
        samples[index] = current;
        total += current;
    }

    PlaySound(SOUND_UP);

    index = 0;
    while(true)
    {
        // compute the average
        average = total * 10 / N;

        // get new sample
        wait_for_touch();
        current = EYE;

        // replace sample
        total -= samples[index];
        total += current;
        samples[index] = current;

        // adjust index
        ++index;
        if (index==N) index = 0;
    }

}
```

The program assumes that a touch sensor is attached to port 1 and a light sensor is attached to port 2 (the same sensor configuration used in Seeker, discussed in Chapter 3). The `wait_for_touch` function waits for the touch sensor to be pressed and released, then emits a brief sound. The main task starts by declaring its variables and configuring the sensors and display. Note that the variable `average` was intentionally kept as a global variable so that it could be displayed.

The program gets the initial `N` readings and stores them in the `samples` array. It also keeps a running `total` of the readings. After the initial samples have been read the `SOUND_UP` sound will be played and the program will be ready to update its samples.

The update process is a little more complicated than reading in the initial samples. That's because each time a new sample is read, the oldest sample must be discarded. The `index` variable is used to keep track of the oldest sample, which, initially, is element 0.

The `while` loop is where most of the work gets done. First, it computes the average from the current total. Normally, an average would be computed by dividing the total by the number of samples, but in this case we are also multiplying by ten. By doing this, `average` will actually contain a value that is ten times the actual average. However, the display was configured to display values in tenths, so the overall result is that the LCD will display the real average including a single decimal digit of precision (for example, "45.3" instead of just "45").

Next, the program waits for the touch sensor to be activated, then it reads the light sensor. Replacing the old sample with a new one requires three operations—subtracting the old sample from `total`, adding the new sample to `total`, and saving the current sample in the `samples` array. Now that the oldest sample has been replaced with the newest, `index` must be adjusted to point to the next oldest sample. In most cases, this will be the next higher index, but we also have to take into account the case where `index` needs to *wrap-around* to the start of the array.

Download the program to the RCX and give it a try. When you first run the program it should display "0.0.". Now press and release the touch sensor three times (if you are using Seeker, then just hit the bumper three times). You should hear the SOUND_UP sound and the display will now show the average of those first three light sensor readings. Aim the sensor at a bright light or into a dark corner and take more samples (press the touch sensor or bumper). If you move the RCX from a light environment to a dark one and then take several samples, you should see the average gradually change from the old value to a new one during the first three samples. It will then tend to stabilize (at least until the lighting changes again). This method of averaging is called *finite response* because any given sample (such as the initial bright sample) will only affect the average for a finite number of new samples before the given sample is discarded. Finite response can be a very useful way to keep a running average and is just one example of the many things you can do with arrays.

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:

```
monitor(MY_EVENTS)
{
    // normal code to execute
}
catch
{
    // code that handles the events
}
```

Events are numbered 0 to 15, and are each represented by a single bit in the event list (also called an event mask). The EVENT_MASK macro creates the appropriate mask from a single event number, and these masks can be added together in order to monitor multiple events. For example, to monitor events 2 and 5 the following line could be used:

```
monitor(EVENT_MASK(2) + EVENT_MASK(5))
```

Event Configuration

Each event has a number of parameters that determine how and when it will be triggered. The first parameter is the *event source*, which determines which stimulus the event should be monitoring. The source can be a sensor, timer, counter, or the message buffer (`SENSOR_2`, `Timer(1)`, and so on).

The second parameter is the *event type*, which defines what condition the event should be looking for. There are eleven different event types as shown in Table 4-1.

Table 4-1. Event Types

EVENT TYPE	CONDITION	EVENT SOURCE
EVENT_TYPE_PRESSED	value becomes *on*	sensors only
EVENT_TYPE_RELEASED	value becomes *off*	sensors only
EVENT_TYPE_PULSE	value goes from *off* to *on* to *off*	sensors only
EVENT_TYPE_EDGE	value goes from *on* to *off* or vice versa	sensors only
EVENT_TYPE_FASTCHANGE	value changes rapidly	sensors only
EVENT_TYPE_LOW	value becomes *low*	any
EVENT_TYPE_NORMAL	value becomes *normal*	any
EVENT_TYPE_HIGH	value becomes *high*	any
EVENT_TYPE_CLICK	value from *low* to *high* back to *low*	any
EVENT_TYPE_DOUBLECLICK	two clicks within a certain time	any
EVENT_TYPE_MESSAGE	new message received	Message() only

Boolean Value Events

The first four event types (pressed, released, pulse, and edge) are most useful with touch sensors because the sensors have obvious *on* and *off* conditions. However, these event types can also be used with other sensors provided that the default boolean conversion (described in Chapter 2) is acceptable.

The program you see in Listing 4-3 illustrates how to use the EVENT_TYPE_PRESSED event with a touch sensor on port 1. Whenever the touch sensor is pressed the program will emit a short beep.

Listing 4-3. event_pressed.nqc

```
// event_pressed.nqc

#define MY_EVENT 0  // we'll use event #0

task main()
{
    SetSensor(SENSOR_1, SENSOR_TOUCH);
    SetEvent(MY_EVENT, SENSOR_1, EVENT_TYPE_PRESSED);

    while(true)
    {
        monitor(EVENT_MASK(MY_EVENT))
        {
            until(false);
        }
        catch
        {
            PlaySound(SOUND_CLICK);
        }
    }
}
```

The SetEvent function configures an event with the specified source and type. The event can then be used in a monitor statement (after computing its EVENT_MASK).

If the default boolean conversion is not acceptable, then a slope parameter may be used (see Chapter 2). In this case, an EVENT_TYPE_FASTCHANGE will be triggered whenever the sensor's value changes more rapidly than the slope parameter.

Range Events

Many event sources don't have such well-defined boolean values. Sensors other than the touch sensor generally work better as a range of values, and sources such as timers and counters have no boolean equivalent. For these cases, the firmware provides the *range event* types. Two additional parameters—the *lower limit* and *upper limit* are used to decide which range a sensor's value falls into. If the value is less than the lower limit, then it is considered to be *low*. If the value is greater than the upper limit, then it is *high*. When the value is between the lower and upper limits (or equal to either limit) it is considered *normal*. The low, normal, and high event types are triggered whenever the event source first enters that range. For example, if a source goes from normal to low then a low event will be triggered, but while it remains low no additional low events will be triggered.

There's another wrinkle in the conversion from a source's value to the low/normal/high ranges. In the real world, sensors don't instantly change value from one extreme to another—sometimes they may waver a bit. If they waver near one of the limit values, this can cause many events to happen, even though the value isn't changing in a significant way. For example, consider a light sensor (values from 0 to 100) where the range 0–59 is considered "dark" and 81–100 is "bright". To use events with this sensor it would make sense to set the lower limit to 60 and the upper limit to 80. Now consider what would happen if the sensor's value was wavering between 80 and 81. Each move to 80 would cause it trigger a normal event, and each move to 81 would trigger a high event. This is probably not very helpful to the program—it would be much better if the program considered this wavering value to be either normal or high and not continue to generate new events.

One way to overcome this problem is to use a slightly lower cutoff for exiting the high range than was used to enter it. For example, the cutoffs 75 and 80 could be used so that a value of 75 or below would be in the normal range and anything above 80 would be in the high range. Values from 76 to 80 would leave the range unchanged. Let's say the initial value starts at 50 (normal state) and begins to increase. As soon as the value reaches 81 or above it would be considered high and an appropriate event would be generated. Now if the value wavers a bit—perhaps decreasing to 78 then back up to 81—the value would still be considered in the high range. The value would have to drop all the way to 75 or lower before the range would transition back to normal. This difference between the two cutoffs is called hysteresis and is a very effective way of minimizing the jittery effects of real world sensors. An event's hysteresis parameter affects both the lower limit and upper limit as shown in Figure 4-1.

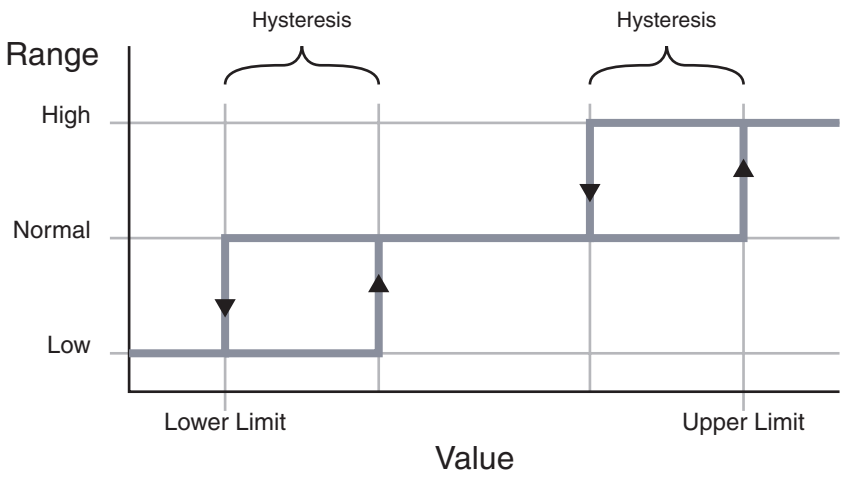

Figure 4-1. Hysteresis

By default, the lower limit and upper limit are set to the minimum and maximum values of the source, and hysteresis is set to 0. The current values of these parameters can be read with `LowerLimit(event)`, `UpperLimit(event)`, and `Hysteresis(event)`, where `event` is the event number. The values can be set with `SetLowerLimit(event, value)`, `SetUpperLimit(event, value)`, and `SetHysteresis(event, value)`, where `event` is the event number and `value` is the desired setting. The program shown in Listing 4-4 monitors the light sensor using an `EVENT_TYPE_HIGH` event. For completeness the lower limit is also set, although this isn't strictly necessary because the program is only concerned with the normal and high ranges.

Listing 4-4. event_high.nqc

```
// event_high.nqc

#define MY_EVENT 0  // we'll use event #0

task main()
{
    SetSensor(SENSOR_2, SENSOR_LIGHT);
    SetUserDisplay(SENSOR_2, 0);

    SetEvent(MY_EVENT, SENSOR_2, EVENT_TYPE_HIGH);
    SetUpperLimit(MY_EVENT, 80);
    SetLowerLimit(MY_EVENT, 50);
    SetHysteresis(MY_EVENT, 5);
```

```
    while(true)
    {
        monitor(EVENT_MASK(MY_EVENT))
        {
            until(false);
        }
        catch
        {
            PlaySound(SOUND_CLICK);
        }
    }
}
```

The program displays the current value of the light sensor (which is assumed to be on sensor port 2) on the LCD. By slowly turning the light sensor towards a bright light you should be able to gradually increase the sensor value. As soon as the value exceeds 80, the event will be triggered and a sound will be played. The sensor's value may then be lowered all the way to 76, then back above 80 without triggering an additional event. This difference of cutoff values (above 80 to enter the high level, but less than or equal to 75 to remain) is due to the hysteresis parameter.

NOTE *In the previous example, it will sometimes appear that the event is being triggered at 80 rather than 81. This is because event monitoring is a bit faster than the display and a momentary value of 81 may be enough to trigger the event, even though the display may still register a value of 80.*

Click Events

The firmware also has the ability to watch for a transition from low to high and back to low. If both transitions happen within a certain time—called the *click time*—then the entire sequence will trigger an `EVENT_TYPE_CLICK` event. If a second click occurs, and the time between the end of the first click and the start of the second is also less than the click time, then an `EVENT_TYPE_DOUBLECLICK` event will also be triggered. The click time is specified in units of 10 milliseconds (ms). It defaults to 15, which is equivalent to 150 ms. It can be read using `ClickTime(event)` and set with `SetClickTime(event, value)`.

The program you see in Listing 4-5 watches for the touch sensor on port 1 to be clicked. Note that the upper limit and lower limit are set to 0 and 1 respectively so that when the sensor is pressed (value of 1) it will be high and when it is released (value of 0) it will be low. The click time has been set to half a second (500 ms). If the

touch sensor (attached to sensor port 1) is pressed then released within a half second, the event will be triggered and a sound will be played. Pressing and holding the sensor for longer than a half second will not trigger an event.

Listing 4-5. event_click.nqc

```
// event_click.nqc

#define MY_EVENT 0  // we'll use event #0

task main()
{
    SetSensor(SENSOR_1, SENSOR_TOUCH);
    SetEvent(MY_EVENT, SENSOR_1, EVENT_TYPE_CLICK);

    SetUpperLimit(MY_EVENT, 0);
    SetLowerLimit(MY_EVENT, 1);
    SetClickTime(MY_EVENT, 50);

    while(true)
    {
        monitor(EVENT_MASK(MY_EVENT))
        {
            until(false);
        }
        catch
        {
            PlaySound(SOUND_CLICK);
        }
    }
}
```

Message Event

The last event type, `EVENT_TYPE_MESSAGE`, only applies to events using `Message()` as their event source. This event type is triggered whenever a new message arrives, regardless of whether the value of `Message()` itself changes.

Using Events with Seeker

Now that the basics of event monitoring have been presented, we can look at how events can help with the bump and seek program. As discussed before, the robot will normally be engaged in seeking. Whenever a bump occurs, however, seeking

must be interrupted while the robot attempts to avoid the obstacle. Previous approaches used two separate tasks for the seek and bump behaviors and then sought to coordinate those tasks. With events, however, both the seek and bump actions can be implemented within a single task. We'll accomplish this by using an event to detect bumping. Because monitored events interrupt normal program flow, bumping into an obstacle can then interrupt seeking and the obstacle can be avoided. Using the functions defined in the original seekbump.nqc, the main task shown in Listing 4-6 both seeks and bumps (no seek task is required).

Listing 4-6. Main task for seekbump_event.nqc

```
#define BUMP_EVENT 0

task main()
{
    setup();

    SetEvent(BUMP_EVENT, BUMPER, EVENT_TYPE_RELEASED);

    while(true)
    {
        monitor(EVENT_MASK(BUMP_EVENT))
        {
            // seek
            Wait(SEEK_DELAY);
            while(true)
            {
                until(EYE < threshold);
                find_target();
            }
        }
        catch
        {
            // bump
            avoid_obstacle();
        }
    }
}
```

NOTE *The program in its entirety is contained in the file seekbump_event.nqc, which can be downloaded from the book's page on the Apress Web site at* `http://www.apress.com`.

This is perhaps the most elegant of the Seeker programs and is an excellent example of how event monitoring can be used to simplify a problem.

Bits and Pieces

Although this chapter explored some of the major new features of the RCX 2.0 Firmware, it is not by any means a comprehensive guide to its capabilities. Several of the features have additional calls or options that weren't discussed. There are numerous other features that are smaller in scope but can be just as important if they solve a specific problem you are having. Here is a partial list of some of these improvements:

- The capabilities for playing sounds have been enhanced to allow a program to mute and unmute sound, clear pending sounds, and play a note whose frequency is determined by a variable (previously the frequency had to be a constant).
- The timers have been enhanced to provide 10 ms measurement in addition to the original 100 ms resolution timers.
- Programs can now cause the RCX to switch to another program and start it.
- Programs now have direct access to the IR port. They can send arbitrary bytes of data rather than just the simple message capability previously available.
- Programs can read the present battery level.

For more information about this firmware, your first stop should be the official documentation provided by LEGO. The RIS 2.0 SDK contains a PDF document titled "LEGO MINDSTORMS RCX 2.0 Firmware Command Overview," which describes all of the commands and bytecodes supported by the firmware.

The documentation for NQC also contains some pertinent information about the new firmware, although, in general, its focus is more on how the features are called from within NQC rather than the internal workings of those features.

Bear in mind that this is a relatively new field—NQC only recently added support for the new firmware, and programmers are just starting to understand how the features can be used to build better programs. Hopefully this chapter has piqued your interest—perhaps you will even help contribute to our knowledge of how to use the RCX 2.0 Firmware effectively!