Definitive Guide to LEGO® MINDSTORMS™, Second Edition

DAVE BAUM

Apress™

Definitive Guide to LEGO® MINDSTORMS™, Second Edition

ISBN (pbk): 1-59059-063-5

Printed and bound in the United States of America 12345678910

Technical Reviewer: Rodd Zurcher

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Copy Editor: Ami Knox

Compositor: Diana Van Winkle

Illustrators: Rodd Zurcher, Tony Jonick

Indexer: Ann Rogers

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

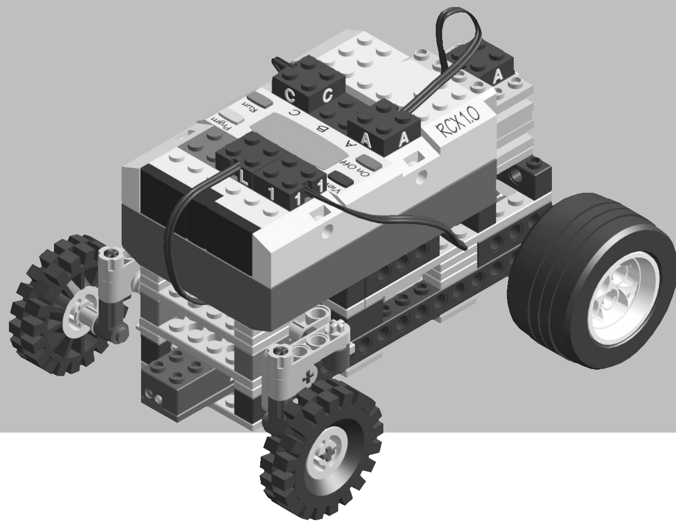
Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Chapter 13
Steerbot



Many of the previous robots have been based on the tread-driven chassis introduced with Tankbot. Although such designs are easy to build and program, they are limited by the size and traction of the treads themselves. This puts a practical limitation on the size and speed of robots that can be built with the standard LEGO treads. In this chapter, we will construct a faster robot using a new steering mechanism. In addition, proximity detection will be used (instead of physical bumpers) to avoid obstacles.

RACK AND PINION STEERING

In the real world a different steering mechanism, known as *rack and pinion steering*, is much more common. In fact, it is the same mechanism used in virtually all automobiles. Rack and pinion steering provides complete flexibility on the shape and size of tires, thus it is often a good choice for vehicles that do not perform adequately with tracks. In the case of Steerbot, the steering will be done by the front wheels, and the drive train will power the rear wheels. Other combinations are also possible—forklifts often steer the rear wheels, sport utility vehicles power all four wheels, and some cars even provide four-wheel steering—however, the basic principles are the same.

When driving straight, a vehicle's wheels are parallel to one another. In order to turn, the front wheels are angled slightly. In a sense, this angle aims the front of the vehicle to the right or left. As long as the wheels remain angled, the vehicle will continue to turn.

In order for the wheels not to slide, the outside wheels must turn slightly faster than the inside wheels. This is the same effect that we exploited in the track-based designs, but this time cause and effect are reversed. Whereas before we changed the speed of one side in order to cause a turn, now the turn (which is caused by the angle of the wheels) will cause a difference in speed to be required. Even if we were willing to use two motors (one for the right-rear wheel, and one for the left-rear wheel), the required difference in speed depends on the sharpness of the turn and is rather complicated.

The solution involves using a *differential*, which was described in Chapter 4. We can use a motor to turn the shell of the differential. The differential functions by distributing power to both wheels in such a way as to minimize the overall friction. When driving straight, this means that both wheels will turn at the same speed. When turning, however, the inside wheel will turn slightly slower than the differential shell, and the outside wheel will turn slightly faster. This variation in speed will happen without any direct intervention—the only thing we need to do is spin the differential itself.

CHASSIS CONSTRUCTION

Steerbot is mechanically the most complicated robot so far, but if you look carefully you will see the same familiar techniques employed on earlier robots. The only truly unique portion of Steerbot is the rack and pinion steering. LEGO manufactures several specialized pieces that can be used to easily build a compact rack and pinion mechanism. Unfortunately, the RIS set does not include all of the necessary pieces, so we will have to use a little creativity and build the mechanism from scratch.

Figures 13-1 and 13-2 show the initial construction of the chassis, including a differential and the special brackets used to mount a motor. The new pieces near the front of the frame (left side of the illustration) are something of a "hack." This is the area where the rack will need to slide back and forth; thus a smooth surface is required. Ideally, a couple of *tiles* (plates without studs on top) would be used. Since the Robotics Invention System doesn't include any regular tiles, these odd pieces are used instead.

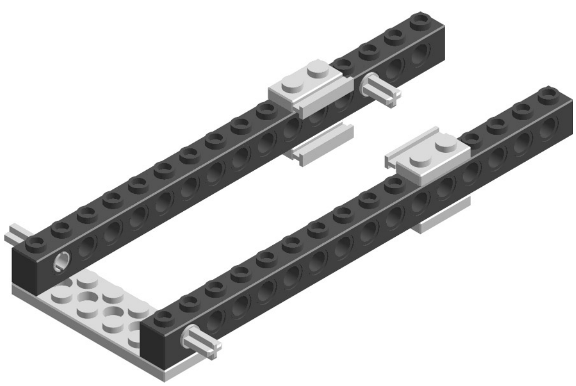


Figure 13-1.
Steerbot step 1

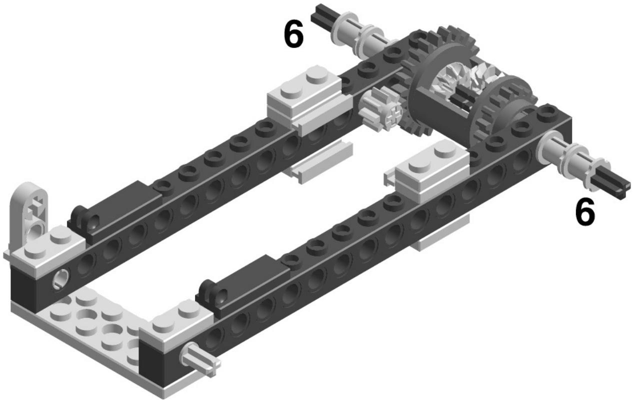


Figure 13-2.
Steerbot step 2

The motor (with a crown gear attached) is shown in Figure 13-3, slid into position in Figure 13-4, then locked in place in Figure 13-5. The crown gear is a 24-tooth gear with curved teeth that allows it to mesh at a right angle with another gear (see Chapter 4 for a more detailed explanation of the crown gear). The gearing between the motor and differential may appear a bit odd. The motor turns a crown gear (24 teeth), which meshes with an 8-tooth gear for a 3:1 reduction. This 8-tooth gear then meshes with the 24-tooth side of the differential for a 1:3 gear ratio. The resulting ratio is simply 1:1. So why not skip the 8-tooth gear and mesh the crown gear directly with the differential? In order to do this, the crown gear would need to be moved closer to the differential. Unfortunately, at such close proximity, the center of the crown gear would collide with the shell of the differential. By introducing the 8-tooth gear, space is added between the crown gear and the differential without altering the overall speed and power (aside from a minor amount of friction).

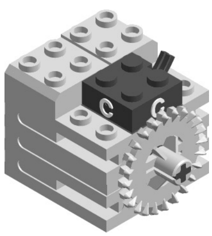


Figure 13-3.
Steerbot step 3

A light sensor, which will be used for proximity detection, is also added in Figure 13-4. The touch sensor added in Figure 13-5 will be used to detect when the steering is centered.

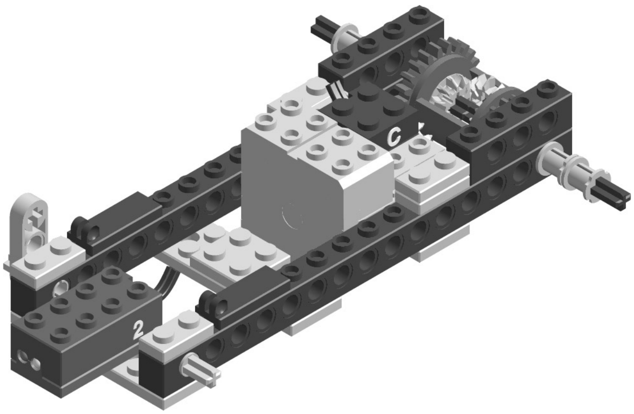


Figure 13-4.
Steerbot step 4

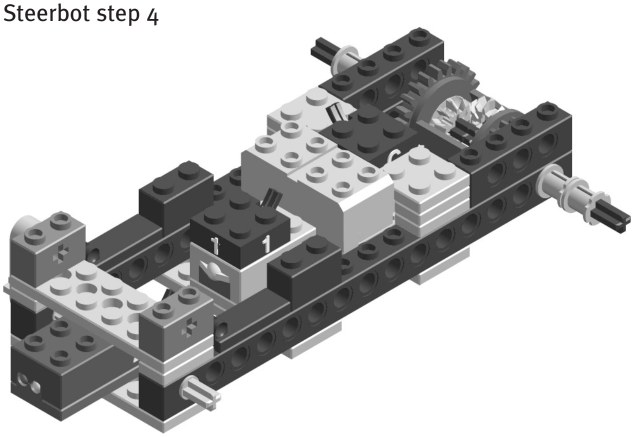


Figure 13-5.
Steerbot step 5

Figures 13-6, 13-7, and 13-8 show the construction of the steering rack. The small #2 axle underneath the rack will press against the touch sensor when the rack is centered.

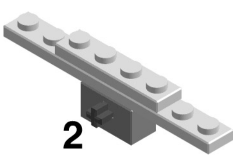


Figure 13-6.
Steerbot step 6

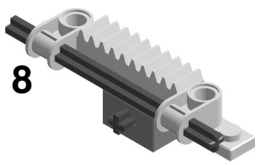


Figure 13-7.
Steerbot step 7

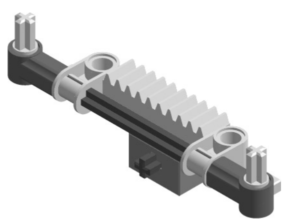


Figure 13-8.
Steerbot step 8

The steering rack is added to the chassis in Figure 13-9. It should rest lightly on the smooth pieces and slide back and forth easily. The #10 axle added in this step will eventually be the point of support between Steerbot and its front wheels. Because of this, it is important that it be properly braced, hence the 1×3 liftarms that anchor it firmly to the 1×16 beams of the chassis itself. The axle and gear added in Figure 13-10 will be used to control the steering.

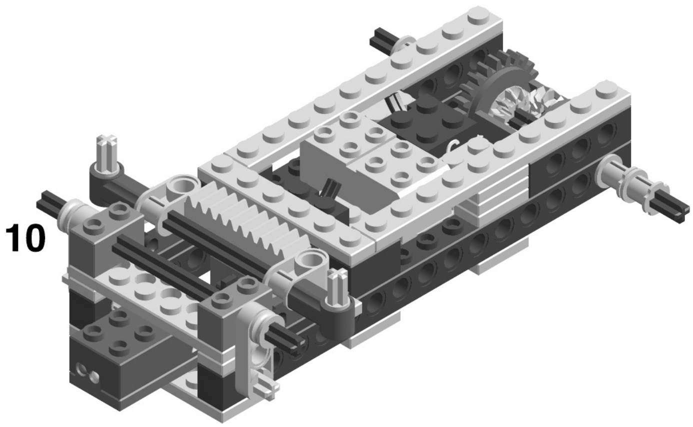


Figure 13-9.
Steerbot step 9

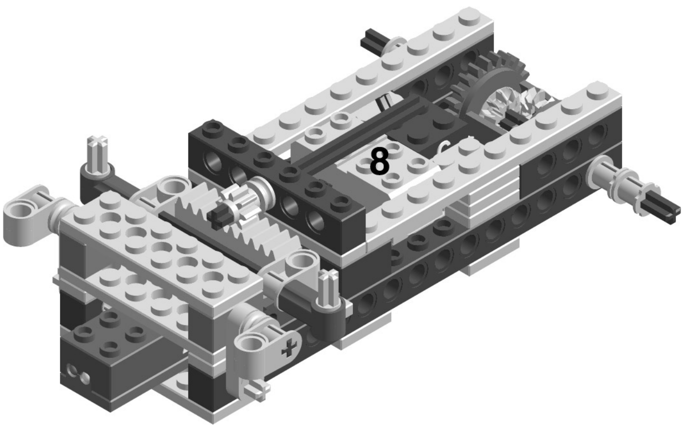


Figure 13-10.
Steerbot step 10

The axles for the front wheels are constructed in Figures 13-11 and 13-12. The wheels need two degrees of freedom—they must be able to both spin and pivot. This is similar to building the swivel wheel for Tribot, except that this time it is all right if the wheel is a little off center from the pivot point.

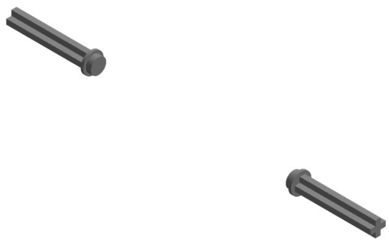


Figure 13-11.
Steerbot step 11

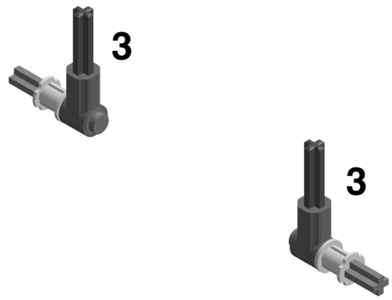


Figure 13-12.
Steerbot step 12

Once the two pivot assemblies are completed, they are added to the rest of the model. The rack and pinion mechanism is completed by the addition of two 1×4 liftarms that link the steering rack to the wheel pivots as shown in Figure 13-13. Be sure to leave a small gap when attaching these liftarms to the axle pegs at the ends of the steering rack. This gap is necessary to make the liftarms level and allow the steering mechanism to move freely. Observe how the wheels rotate right and left as the rack is moved back and forth and how the touch sensor gets pressed only when the wheels are centered (or very close to center). If the touch sensor is not getting pressed in firmly enough, you may have to adjust the #2 axle.

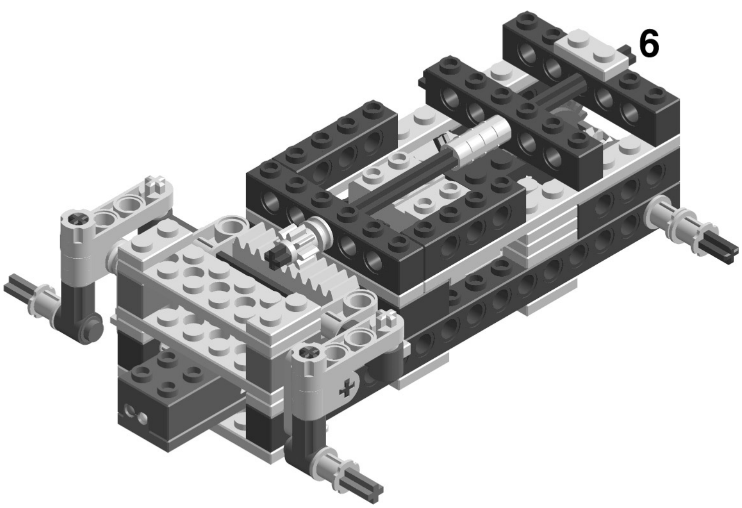


Figure 13-13.
Steerbot step 13

A few odds and ends are added in Figure 13-14, then in Figure 13-15 (which shows the view from the back of Steerbot), a pair of pulleys and a blue rubber belt are used to transfer power from a motor to the steering mechanism. The difference in size between the pulleys provides a reduction of approximately 7:2, which constitutes a good compromise between power and speed. A belt was intentionally used (as opposed to gears) in order to allow slippage when the steering has been turned completely to the left or right. As usual, support for the motor must take into account the rather unusual shape of its bottom.

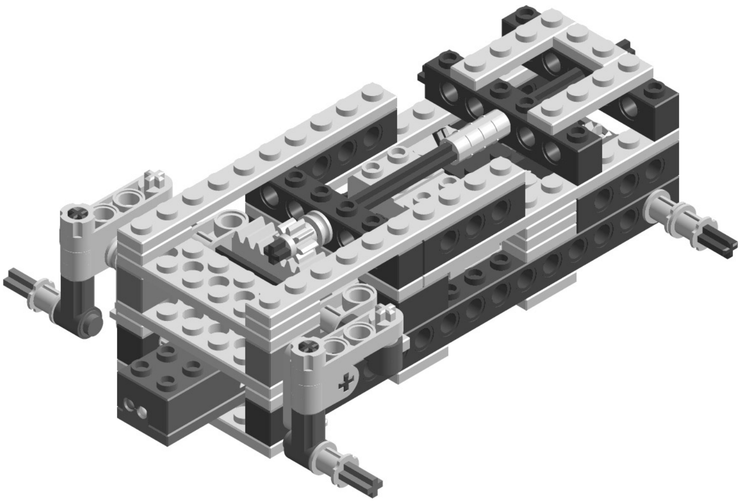


Figure 13-14.
Steerbot step 14

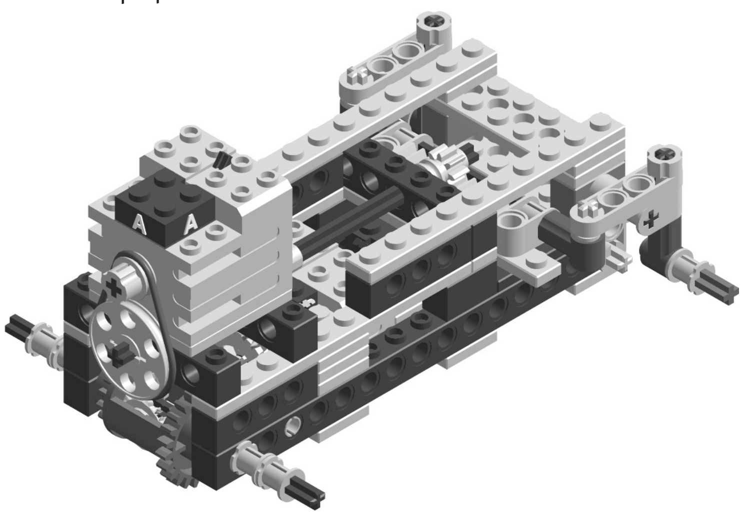


Figure 13-15.
Steerbot step 15

Steerbot is completed by adding the RCX and wheels to the chassis as shown in Figure 13-16.

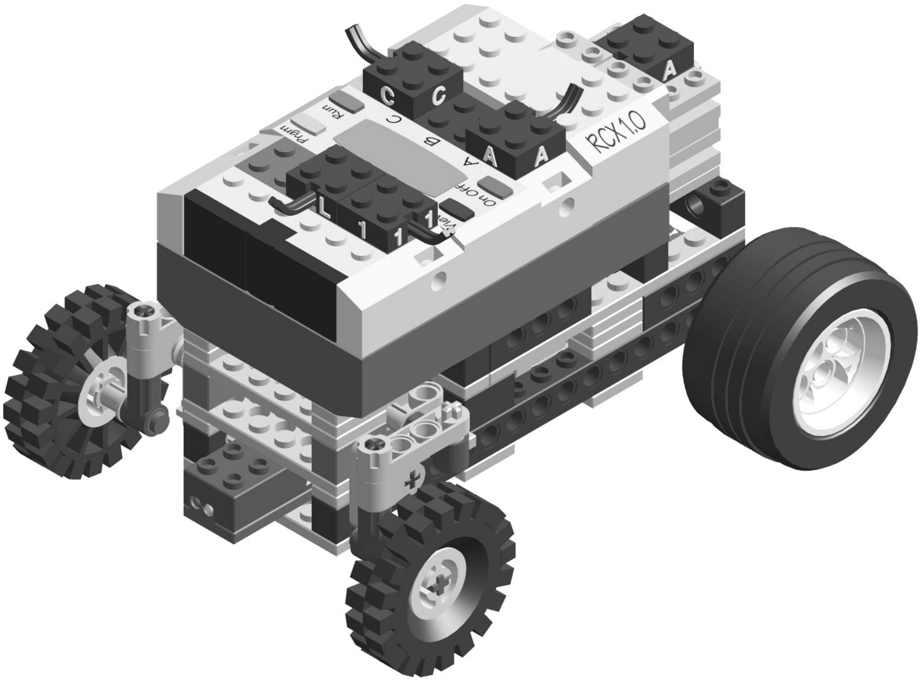


Figure 13-16.
Steerbot step 16

PROGRAMMING

We will first introduce three operations to manage the steering. The RCX Code versions of these functions are shown in Figure 13-17, and the NQC version appears in Listing 13-1. To turn the wheels left, we run the steering motor backward for a little while—$^1/_2$ second to be exact. This number was chosen because it provides ample time for the steering to be moved from the far right extreme to the far left extreme. Turning the wheels right is the same, except the motor is run in the forward direction.

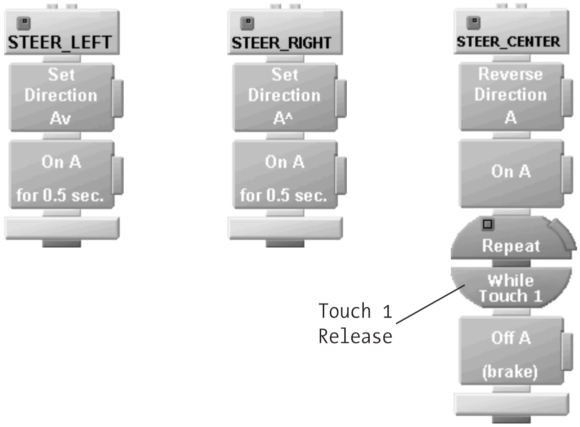


Figure 13-17.
Steering blocks in RCX Code

Listing 13-1. Steering Functions in NQC

```
#define CENTER   SENSOR_1
#define STEER    OUT_A
#define FULL_TURN_TIME   50

void right()
{
    Fwd(STEER);
    OnFor(STEER, FULL_TURN_TIME);
}


void left()
{
    Rev(STEER);
    OnFor(STEER, FULL_TURN_TIME);
}


void center()
{
    Toggle(STEER);
    On(STEER);
    until(CENTER==1);
    Off(STEER);
}
```

Centering the steering is a little different. The centering operation assumes that the steering had previously been moved to the right or left. As a result, switching the motor's direction will get the steering moving back toward center. The motor remains on until the centering touch sensor is triggered. If you find that the centering operation is turning either too little or too much, then it may be adjusted by either adding a delay after the sensor is triggered (turning too little), or reversing the motor's direction for a bit (turning too much). Examples of this in NQC are shown in Listing 13-2.

Listing 13-2. Adjusted Steering Functions in NQC

```
// version of center () that turns a little extra
void center()
{
    Toggle(STEER);
    On(STEER);
    until(CENTER==1);
    Wait(10);
    Off(STEER);
}

// version of center() that turns a little less
void center()
{
    Toggle(STEER);
    On(STEER);
    until(CENTER==1);
    Toggle(STEER);
    Wait(10);
    Off(STEER);
}
```

Steerbot also needs a way to detect obstacles. Unlike previous robots, which used a physical bumper, Steerbot will look for obstacles with a light sensor using a technique that is commonly called *proximity detection*. The light sensor detects incoming light, so all we need to do is shine a lot of light in front of Steerbot, and as it nears an obstacle, some portion of that light will bounce back and be detected by the sensor. Unfortunately, the red LED built into the sensor itself does not emit enough light to work at a distance. However, the RCX has its own infrared LED that is normally used for communication. Even though you can't see its light (because it is infrared), the light sensor has no trouble detecting it.

The RCX's communication LED can operate at two different power levels. The lower level is good for short range communication, while the higher level emits more light. Since we want as much light as possible, it is important to tell the RCX to use the high power mode. If you are using the RIS software, this is controlled from the Settings screen. From the Main Menu screen press the **Settings** button, and then select **high** in the **Communication** box. If you are using NQC, the program itself can specify that the RCX use the high power mode, as you will see a bit later in the chapter.

If we instruct the RCX to send a message, it will emit some pulses of infrared light. By repeatedly sending messages within a loop, we will ensure that pulses are continually being generated, and if we get close to an obstacle one of these pulses will bounce back and be detected by the light sensor. An RCX Code program that uses proximity detection along with the steering functions introduced earlier is shown in Figure 13-18.

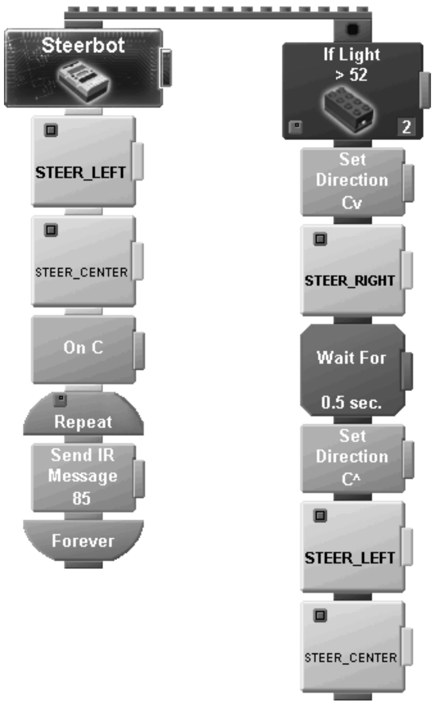


Figure 13-18.
Proximity detection in RCX Code

By placing the **Send IR Message** block in the program stack and using a sensor watcher for the light sensor, we ensure that the pulses of infrared light will be sent concurrently with checking the light sensor. If instead we had a single loop that sequentially sent the message and then checked the sensor, the pulses of light would have stopped by the time the sensor was read.

Listing 13-3 shows the new code that must be combined with Listing 13-1 to program Steerbot in NQC. While most of our previous NQC programs used condition-based tests, this one uses events. Since event checking is done "in the background," using events allows the light sensor to be monitored at the same time the messages are being sent. Another option would be to use two separate tasks, but then we would have a potential risk of missing one of the pulses being reflected back if the light sensor wasn't checked frequently enough. One other item of note is the `SetTxPower` function, which allows a program to control whether communication uses low or high power.

Listing 13-3. Proximity Detection in NQC

```
#define EYE       SENSOR_2
#define DRIVE     OUT_C
#define STRAIGHT_TIME    50

#define PROX_EVENT 1
#define PROX_THRESHOLD 52

task main()
{
    // set up sensors and event
    SetSensor(CENTER, SENSOR_TOUCH);
    SetSensor(EYE, SENSOR_LIGHT);
    SetEvent(PROX_EVENT, EYE, EVENT_TYPE_HIGH);
    SetUpperLimit(PROX_EVENT, PROX_THRESHOLD);

    // align wheels and drive forward
    left();
    center();
    On(DRIVE);

    // use high power communication
    SetTxPower(TX_POWER_HI);

    while(true)
    {
        monitor(EVENT_MASK(PROX_EVENT))
        {
            while(true)
            {
                SendMessage(85);
            }
        }
```

```
        catch
        {
            // back up while turning
            Rev(DRIVE);
            right();
            Wait(STRAIGHT_TIME);

            // forward while turning other direction
            Fwd(DRIVE);
            left();

            // center steer
            center();
        }
    }
}
```

The message number to send is somewhat arbitrary. I used 85 because it gets transmitted as alternating pulses of light. Another option is message 255, which will “clump” the pulses of light together more (longer periods of light, but also longer periods of dark), but generally doesn’t make much difference.

Both the RCX Code and NQC programs use a constant threshold to detect obstacles (the light sensor watcher in RCX Code, and `PROX_THRESHOLD` in NQC). This value must be adjusted for the environment that Steerbot will be driving around in. Run the program, use the **View** button to monitor sensor 2, and hold Steerbot about 1' from an obstacle. The values shown on the RCX will probably bounce around quite a bit. This is because sometimes the RCX reads the sensor when a pulse of light is being emitted, and sometimes it reads the sensor when no pulse is present. Take note of the highest value shown on the RCX’s display, and use that as a starting point for the light sensor threshold.

If Steerbot is having trouble detecting obstacles, try lowering the threshold value. If you are using RCX Code, you should check the Settings screen and make sure that the RCX is configured for high power. Proximity detection will work better in darker rooms since there will be more range between the ambient light of the room and the pulses of light emitted from the RCX.

Another common problem is that the #2 axle underneath the rack isn’t sticking out far enough to activate the touch sensor. If this is case, then the program will get stuck during its initial centering operation and Steerbot won’t go anywhere. To correct this, adjust the #2 axle such that it makes good contact with the touch sensor.

ASYNCHRONOUS STEERING

This section presents a more advanced programming technique that can be used with Steerbot. Although it is possible to apply this technique with RCX Code, the resulting program becomes quite cumbersome. Since NQC lends itself more easily to this style of programming, all of the sample code will be in NQC.

In the previous example, when the program wanted to turn right it called a function or My Block, which didn't return until the wheels had turned completely to the right. This sort of behavior, where an operation does not return until some external activity is complete, is known as *synchronous* behavior. Sometimes it is desirable to be able to start an operation, then continue doing other processing while the operation completes. This execution model is called *asynchronous* behavior.

Consider the process of cooking a "bacon and eggs" breakfast. One way to do this would be first to cook the bacon, and only when it was finished move on to preparing the eggs. In this case the bacon is being cooked synchronously. Another option would be to start cooking the bacon, then begin preparing the eggs while the bacon was still cooking. In this case the bacon is being cooked asynchronously. Preparing the eggs while still keeping an eye on the bacon is a bit more difficult than simply cooking one then the other, but overall it takes less time. This tradeoff often applies to software as well. Synchronous behavior is often the simplest to implement, but in some cases asynchronous behavior is worth the extra effort.

Since turning the wheels completely to the right or left can take up to $^1/_2$ second, steering control is a good candidate for asynchronous operation.

The key to asynchronous behavior (at least in the RCX) is to use multiple tasks. We will create a separate task whose sole function is to turn the wheels to a desired position. Of course, other tasks (known as *clients*) will need some way of interacting with the steering task in order to specify which position the steering mechanism should be set to. It is also desirable for these clients to be able to determine when the steering task has completed its work.

This is starting to sound pretty complicated, and one of the techniques programmers use to make complicated systems easier to understand is called *abstraction*. The key to abstraction is to take some feature of the system and separate "how to use it" from "how it works." The "how to use it" portion is called the *interface* to the feature, and "how it works" is the *implementation*.

We make such abstractions in our everyday lives as well. We constantly use devices—televisions, microwave ovens, automobiles—without completely understanding how they actually work. In many cases the internal operation of these devices has changed considerably over the years, yet the way they are used remains relatively unchanged. Of course, new features are added over time, but typically in a way that is compatible with the old features. Abstracting and designing good programming interfaces, which can grow to accommodate new features while still remaining compatible with existing clients, is one of the most challenging tasks a programmer must face.

Returning to asynchronous steering, the way a client interacts with the steering task is the *interface* to the steering feature. As mentioned previously, the interface must allow a client to specify which position to set the steering to and then determine when the operation has been completed. In NQC we build this interface using several constants and a pair of variables as shown in Listing 13-4.

Listing 13-4. Interface to Steering Task

```
// interface for steering task
#define STEER_LEFT      -1
#define STEER_CENTER     0
#define STEER_RIGHT      1

int steer_goal; // set this to steer
int steer_current = -999; // check this
```

A client may indicate its desired steering position by setting the value of `steer_goal` to one of the three position constants (such as `STEER_LEFT`). The task may check to see what the current steering position is by reading the value of `steer_current`. Initially, `steer_current` is set to an invalid number (-999) to indicate that the steering task has not been initialized. Once the steering task completes its initialization, this variable will contain a valid steering position.

Note that a client does not need to know anything about how the steering task works (i.e., its implementation). It is sufficient to understand only the interface to the task.

The implementation of our steering task is a bit more complicated than its interface. A complete listing is shown in Listing 13-5.

Listing 13-5. Implementation of Steering Task

```
// implementation of steering task
#define CENTER   SENSOR_1
#define STEER    OUT_A
#define FULL_TURN_TIME    50

task steer_task()
{
    SetSensor(CENTER, SENSOR_TOUCH);
```

```
        // align
        OnFwd(STEER);
        Wait(FULL_TURN_TIME);
        Toggle(STEER);
        until(CENTER==1);
        Off(STEER);

        steer_goal = 0;
        steer_current = 0;

        while(true)
        {
            int goal;

            goal = steer_goal;
            if (goal != steer_current)
            {
                if (goal == STEER_LEFT)
                {
                    // turn left
                    Fwd(STEER);
                    OnFor(STEER, FULL_TURN_TIME);
                }
                else if (goal == STEER_RIGHT)
                {
                    // turn right
                    Rev(STEER);
                    OnFor(STEER, FULL_TURN_TIME);
                }
                else
                {
                    Toggle(STEER);
                    On(STEER);
                    until(CENTER==1);
                    Off(STEER);
                }

                steer_current = goal;
            }
        }
    }
```

As usual, the code begins with some defines for the motor, sensor, and constants. The task itself starts by configuring the sensor, then centering the wheels using code that is almost identical to that from our earlier synchronous example. After the wheels are centered, the two interface variables, `steer_goal` and `steer_current`, are set to indicate that the wheels are centered.

The task then enters an infinite loop which checks to see if `steer_goal` differs from `steer_current`. The actual value of `steer_goal` is copied into the local variable `goal`, prior to this test. This is because the steering task must make a number of decisions, and a change of goal in the midst of these decisions could lead to unpredictable behavior. Since a client may change `steer_goal` at any time, the steering task must read it once, then make all decisions from that single reading. Each time through the loop, this copy is refreshed with the latest value from `steer_goal`.

When `steer_goal` does not match `steer_current`, the task then takes the appropriate action (turning the wheels to the left, right, or center), then updates `steer_current` with the new location. Although the steering happens asynchronously with respect to the clients, each steering operation must run to completion before another one can be acted upon. For example, if `steer_goal` is set to `STEER_RIGHT`, the steering task will begin to move the wheels to the right. This may not complete for another 1/2 second. Meanwhile a client may set `steer_goal` to `STEER_LEFT`. The steering task will continue moving the wheels all the way to the right before checking `steer_goal` again and determining that the wheels must be moved back to the left. In short, once a steering operation has started, it cannot be canceled.

Before the steering task may be used, the client program must start it. In addition, it is often desirable for the client to wait until the steering task has finished its initialization before moving on to other things. These two steps can be done with the following code:

```
start steer_task;
until(steer_current == 0);
```

Using the asynchronous steering task, we can write a new version of our bumper car program. In this version we will add a new behavior: the robot will beep as it backs up. The following code snippet will spend 1 1/2 seconds beeping every 1/2 second.

```
repeat (3)
{
    PlayTone(880, 30);
    Wait(50);
}
```

If we used this code with the previous synchronous steering functions, then the beeps wouldn't start until the steering had been completed. Now that we have asynchronous steering, the beeps can start immediately. The main task for such a program is shown in Listing 13-6 (combine with Listings 13-4 and 13-5 for the complete program).

Listing 13-6. Main Program for Asynchronous Steering

```
#define EYE     SENSOR_2
#define DRIVE     OUT_C
#define STRAIGHT_TIME     50

#define PROX_EVENT 1
#define PROX_THRESHOLD 52

task main()
{
    SetSensor(EYE, SENSOR_LIGHT);
    SetEvent(PROX_EVENT, EYE, EVENT_TYPE_HIGH);
    SetUpperLimit(PROX_EVENT, PROX_THRESHOLD);

    start steer_task;
    until(steer_current == 0);

    On(DRIVE);

    // use "far" messages
    SetTxPower(TX_POWER_HI);

    while(true)
    {
        monitor(EVENT_MASK(PROX_EVENT))
        {
            while(true)
            {
                SendMessage(85);
            }
        }
        catch
        {
            Rev(DRIVE);
            steer_goal = STEER_RIGHT;
            repeat (3)
            {
                PlayTone(880, 30);
```

```
                Wait(50);
            }

            steer_goal = STEER_CENTER;
            Fwd(DRIVE);
        }
    }
}
```

If synchronous behavior is desired while using the steering task, then the client needs to wait for `steer_current` to equal the desired setting before moving on. For example, in the program shown in Listing 13-6, the operation of centering the wheels before driving forward could be made synchronous by replacing

```
steer_goal = STEER_CENTER;
Fwd(DRIVE);
```

with

```
steer_goal = STEER_CENTER;
until(steer_current == STEER_CENTER);
Fwd(DRIVE);
```

CONCLUSION

Rack and pinion steering is not limited to bumper cars. In general, it can be used with the sensor mechanisms from the previous robots to make faster versions of Bugbot, Scanbot, and others. Proximity detection is also applicable to a wide variety of robots; it is much more friendly than ramming into everything with a bumper.

The touch sensor isn't a very reliable way to determine when the wheels are centered, and as a result Steerbot tends to drift a little to the left or right. A more precise steering mechanism can be constructed by using a rotation sensor. For improved accuracy, there should be a gear reduction between the rotation sensor and the 8-tooth gear that turns the rack. That way the rotation sensor will be able to measure smaller amounts of movement of the rack. A belt drive, if used, should be between the motor and the rotation sensor, and not between the rotation sensor and the rack. If a rotation sensor is used, then the value of the sensor itself can replace the `steer_current` variable in the asynchronous steering task.

Even with its challenges, rack and pinion steering can be an interesting choice for almost any robot.