

## BONUS CHAPTER 1

# Zip-Bam-Bot Version XK3

THE SMALL-AND-FAST sumo-bots presented in Part Two of *Competitive MINDSTORMS* can be built entirely out of the pieces available in the RIS. This actually has its advantages for the small-and-fast strategy, since fewer pieces help maintain a high speed. On the other hand, who says you can't add more mechanisms and pieces on a sumo-bot belonging to the small-and-fast strategy? A small sumo-bot *can* take advantage of additional pieces—maybe a third motor?

In this chapter, you'll add a third motor to enhance Zip-Bam-Bot's performance. Although this does stray from the RIS limits, the extra motor is the *only* piece in this sumo-bot not found in the RIS. What if you don't have a third motor? One way to obtain one is to buy the RoboSports or Ultimate Builders Expansion Pack, which both contain a motor. If you don't feel like buying an expansion pack just yet, you can do a search on eBay and easily find LEGO motors. And if you want to order motors from a more official source, you can buy them (and other LEGO motors) from the Pitsco LEGO Educational Division, at [www.plestore.com](http://www.plestore.com).

Figure 1 shows XK3, the version of Zip-Bam-Bot you're going to build in this chapter.

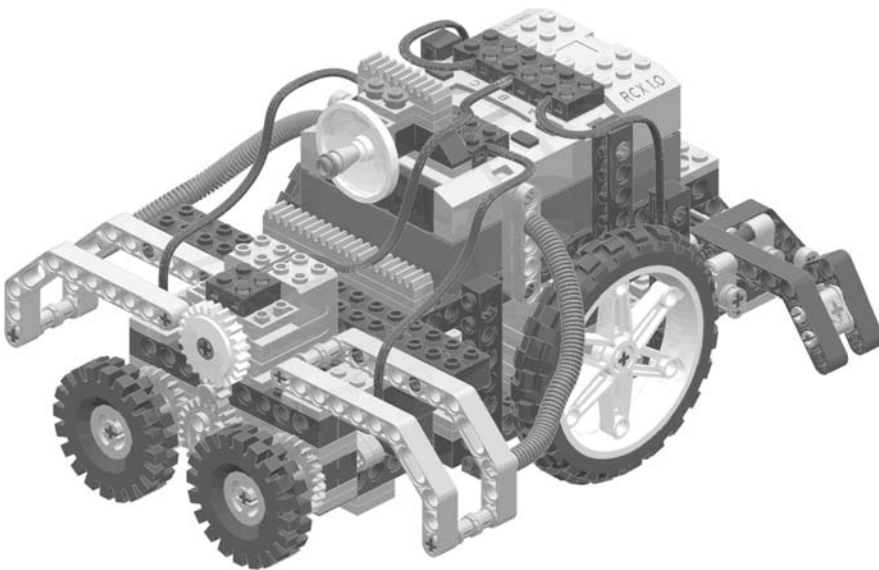


Figure 1. Completed Zip-Bam-Bot Version XK3

## What Does the Extra Motor Do?

What is the purpose of the extra motor? It is intended to power counter-rotating wheels (CRWs). These wheels are positioned “face-forward” on the robot and rely on the following tactic: rotating at a high speed, they attempt to reduce the effects of a front-end hit (by another sumo-bot) and also give XK3 the most effective hit upon impact.

**NOTE** In Chapter 7 of *Competitive MINDSTORMS, Brain-Bot Version ZR2* uses what can be considered “traditional” CRWs. The ones on XK3 are positioned in a different way and have a different purpose than the ones on ZR2.

Since small-and-fast sumo-bots can’t push, they always need a means of detecting the opponent. For XK3, the CRWs are the means. You are going to be using XK3’s CRWs as a type of bumper that can—in addition to everything else—detect the opponent. Using CRWs in this way produces some interesting issues on the building side, as well as on the programming side. You’ll see what these problems are and one possible solution for overcoming them in this chapter.

As for XK3’s substrategy, it is based around the same substrategies as XK1 and XK2 (built in Chapters 4 and 5 of *Competitive MINDSTORMS*), but it also adds one more factor to the equation: the CRWs. This means that XK3 will continually ram its opponent (as the other XKs do), but it will use the new wheels to make the approach more effective. XK3’s strategy essentially falls in between the first and second approach to the repeated ramming method. It relies mainly on its momentum and repeated ramming to push the opponent off the arena, but it also has a mechanism to help defeat the opponent.

Right now, you probably have two burning questions in mind, something like these:

- The CRWs are always supposed to be spinning at a high speed, so how are they going to hit something, stop, and then start up again without breaking the motor?
- How in the world is XK3 going to *know* when its CRWs have hit something (stopped spinning)?

The answers come in the form of a well-known mechanical trick and some programming you might recognize from a chapter in *Competitive MINDSTORMS*, but we’ll get to that later. We need to begin by building XK3 first!

## Constructing Subassemblies for XK3

As noted at the beginning of the chapter, XK3 requires an RIS and one additional motor. XK3’s bill of materials (which does not include the pieces for the chassis) is shown in Figure 2.

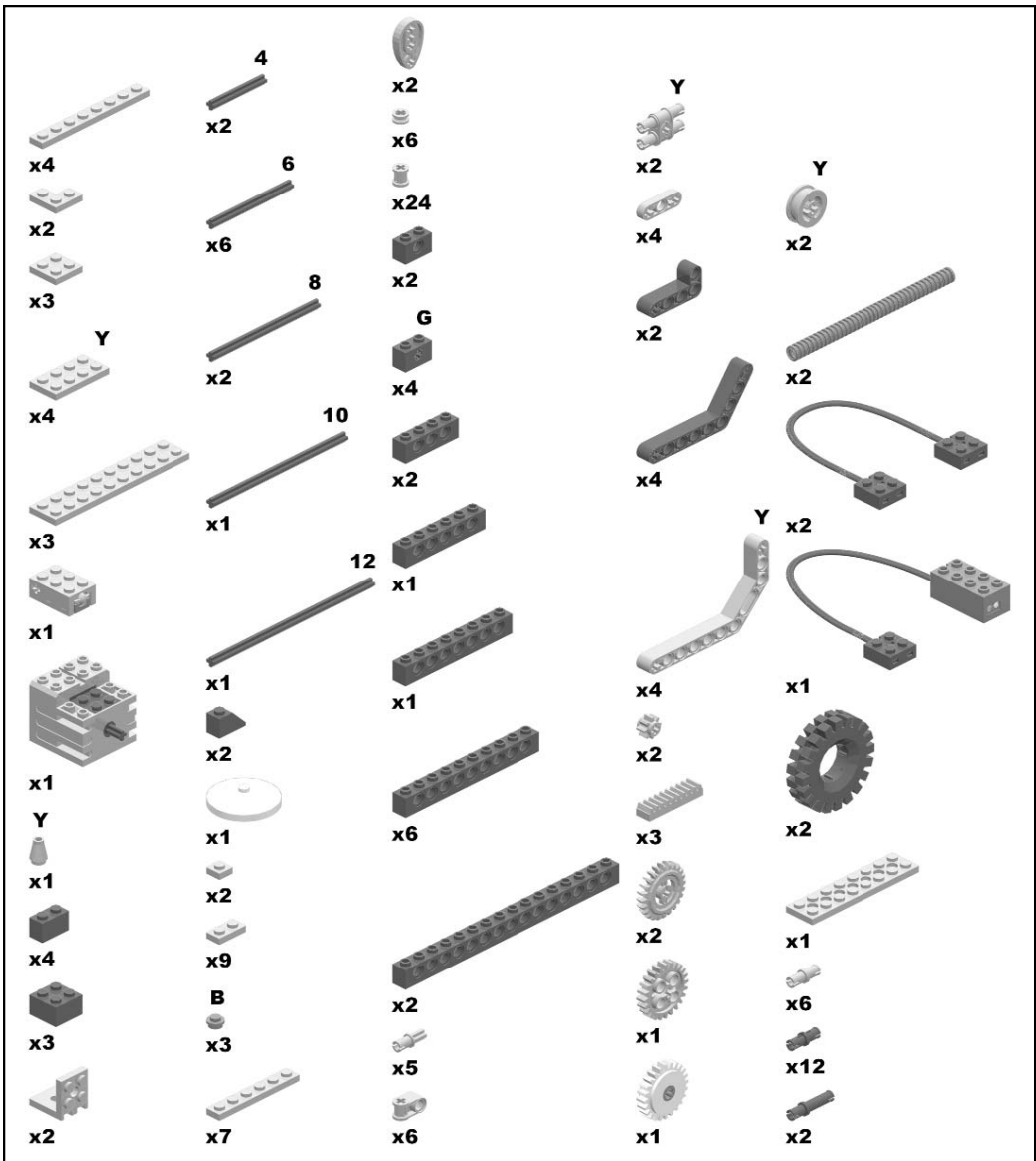


Figure 2. XK3's bill of materials

XK3 has only three subassemblies (but this does not reflect its size):

- Front subassembly
- Eye subassembly
- Back slope subassembly

XX2 (built in Chapter 5 of *Competitive MINDSTORMS*) had one large subassembly, and XK3 has one, too: the front subassembly. This subassembly is extremely large in small sumo-bot terms, and it's actually more like many subassemblies in one. Also, while you're building these subassemblies, make sure to pay attention to the descriptions about how they operate. Once you understand how some of these work, you can easily adapt the designs and ideas into your own sumo-bots.

## Front Subassembly

The front subassembly, shown in Figure 3, holds the CRWs, a light sensor, a touch sensor, and four yellow liftarms that are designed to protect the front end of the robot. This is indeed actually four or five subassemblies in one! But in the small-and-fast strategy, you sometimes need to scale down your designs to make things fit.

**TIP** What you must remember here is weight. Especially for this strategy, weight must be kept to a minimum. Always keep a sumo-bot you're designing using the small-and-fast strategy light; otherwise, it won't be small and fast!

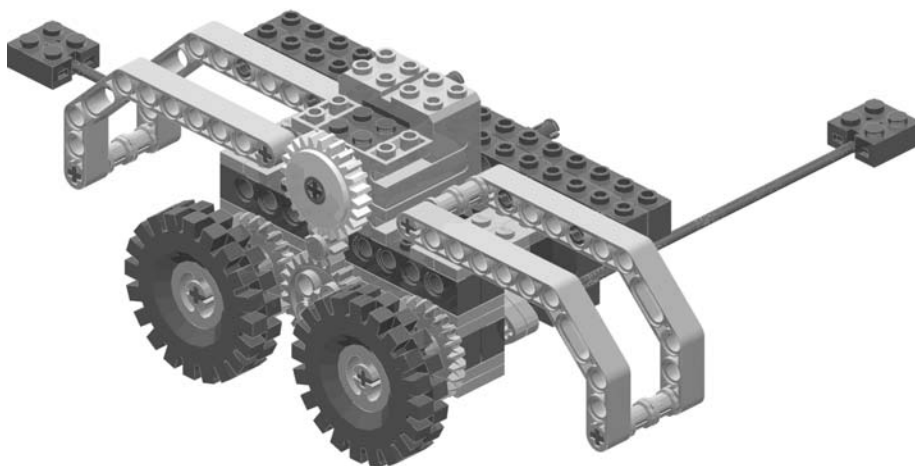
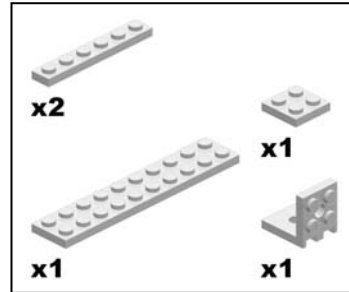
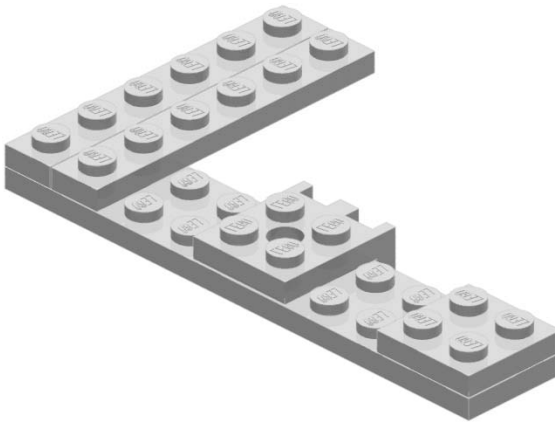


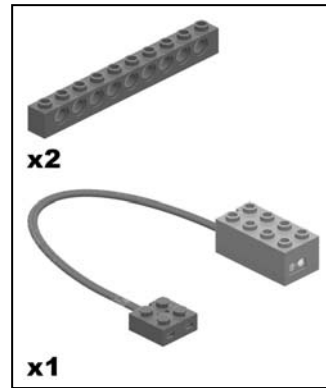
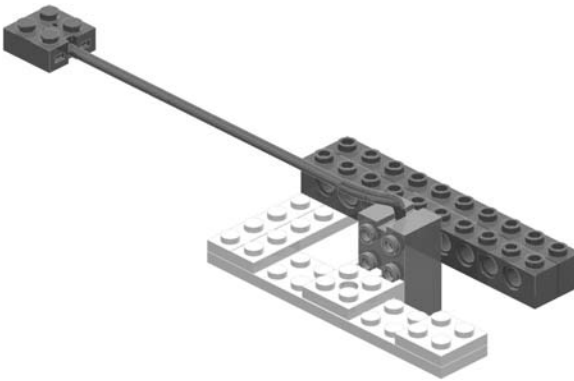
Figure 3. The finished front subassembly

To begin, take a 2x10 plate, 2x2 plate, angle plate, and two 1x6 plates, and position them as shown in step 1.



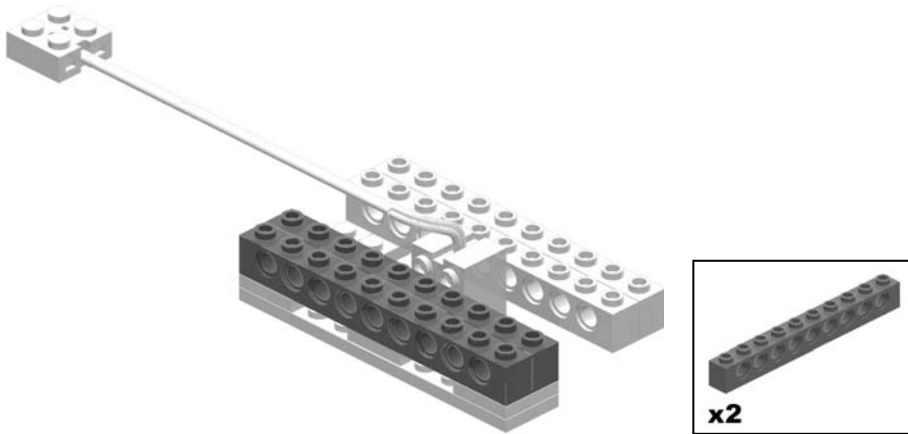
**Front Subassembly Step 1:** Lay down 2x10 plate, 2x2 plate, two 1x6 plates, and angle plate.

In step 2, attach the light sensor to the angle plate and add two 1x10 beams to the assembly. I know this looks (and is) wobbly right now, but it will turn into a very strong assembly by the time you're finished.



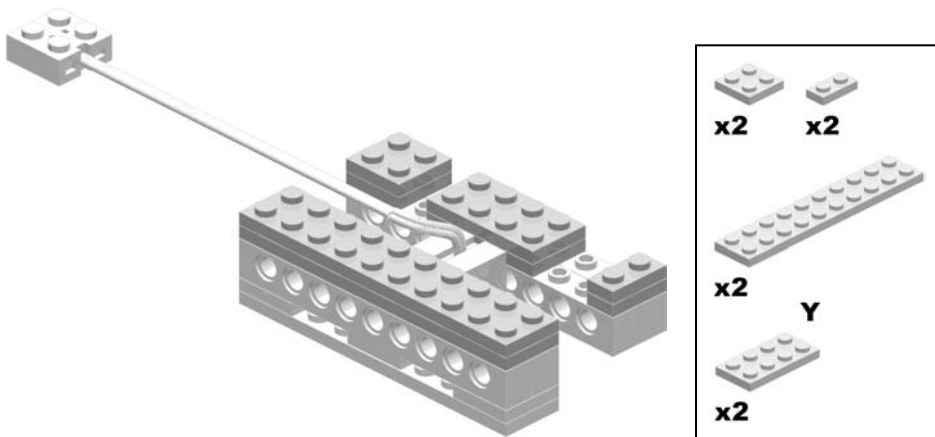
**Front Subassembly Step 2:** Add light sensor to angle plate and two 1x10 beams.

Next, add two more 1x10 beams to the assembly, as shown in step 3.



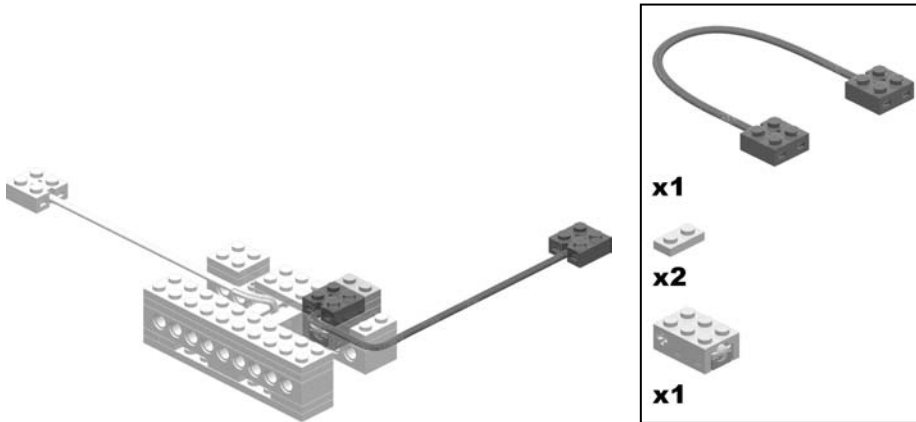
**Front Subassembly Step 3:** Add two more 1x10 beams.

In step 4, you add various plates; instead of using just one type of plate, you use four types of plates! The multiple types of plates are required due to the unique design of the subassembly. With touch and light sensors built in—and many more pieces as well—there is some cramming going on.



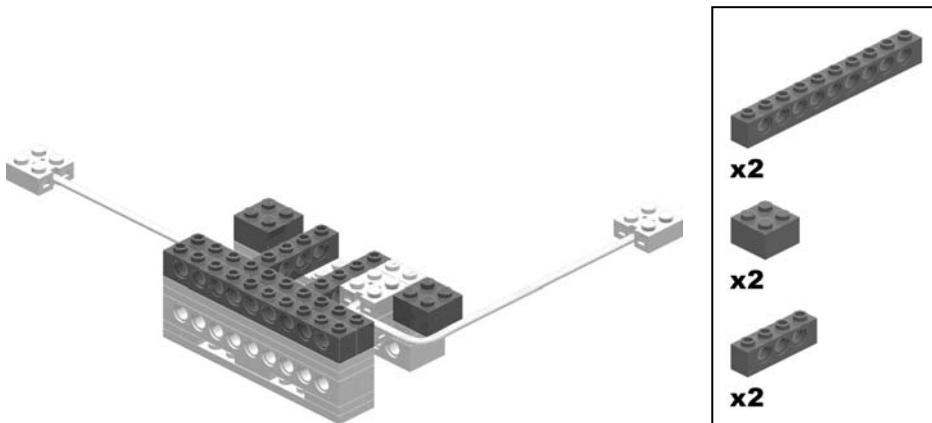
**Front Subassembly Step 4:** Build up assembly with 2x10, 2x4, 2x2, and 1x2 plates.

Add a touch sensor and two 1x2 plates, as shown in step 5. Notice the particular position of the electrical wire on top of the touch sensor. At first, this orientation of the wire seems cumbersome and even irritating, but it's extremely important. Because of the cramped design, this is the *only* direction the wire can face without causing problems. What is the touch sensor for? It has something to do with recognizing when the CRWs have stopped, but we'll get into the details later when we do the programming.



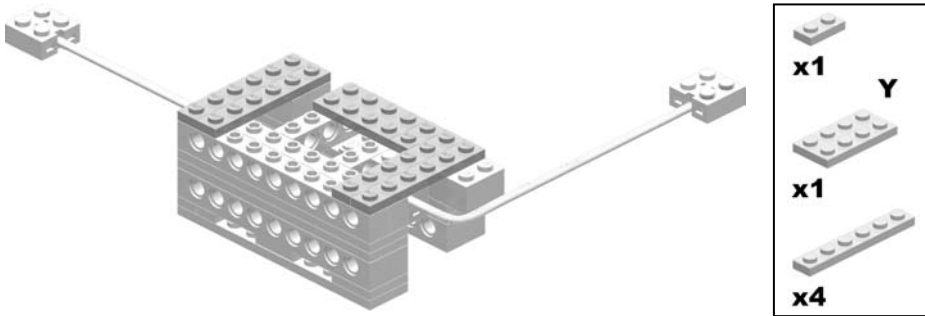
**Front Subassembly Step 5:** Add two 1x2 plates, touch sensor, and electrical wire.

Step 6 builds upward with bricks and beams—the core building elements of the LEGO universe.



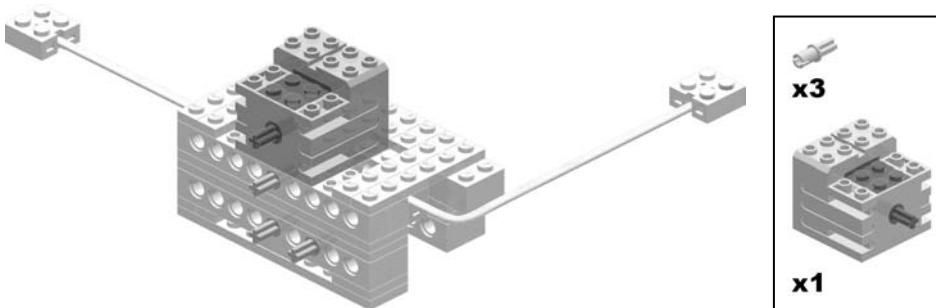
**Front Subassembly Step 6:** Build up assembly with 1x10 and 1x4 beams and two 2x2 bricks.

Step 7 positions plates in several places to strengthen the model.



**Front Subassembly Step 7:** Strengthen assembly with 2x4, 1x6, and 1x2 plates.

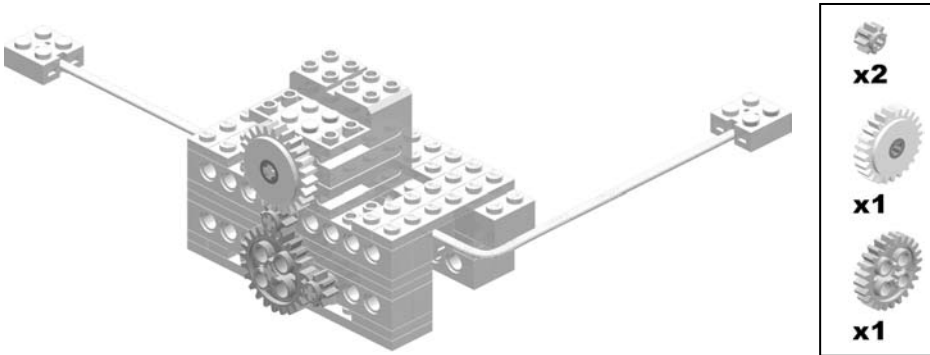
In steps 8 through 10, you will be constructing the gear train for the CRWs. To start, place a motor and three axle pins on the assembly.



**Front Subassembly Step 8:** Attach motor and three axle pins.

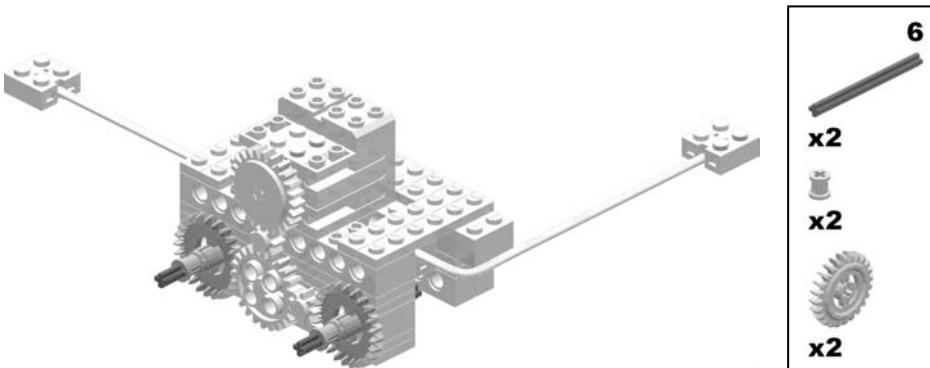
In step 9, position a *white clutch gear*, a 24t gear, and two 8t gears on the front. The clutch gear is *very* important. This special gear (it must be special—there is only one in the RIS) serves the very important task of *slipping*. Once a specified amount of force is exerted on the gear—an amount designated in Newton centimeters on the front of the gear—either the gear itself or the axle in the gear will slip. This property is important because it allows the CRWs on XK3 to stop on impact, while letting the motor continue running, instead of burning out. This is how the CRWs can stop and then start again later, without breaking anything.





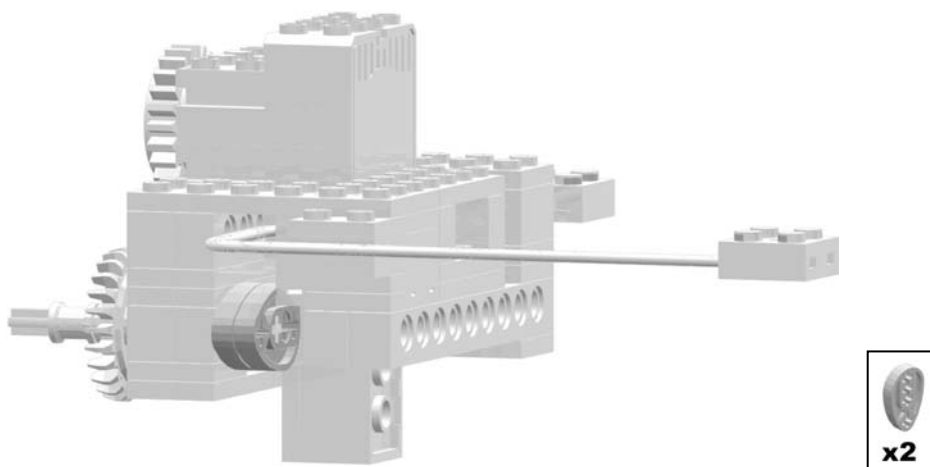
**Front Subassembly Step 9:** Connect 8t gears, 24t gear, and clutch gear.

Next, in step 10, add two #6 axles, two 24t crown gears, and two bushings. Don't worry about the back side of these axles; you will add something there to hold it in place in just a second. Notice that there are more gears on one side of the subassembly than on the other. This is intentional and essential to XK3's CRWs. A common property of gears is that meshing gears turn in opposite directions. This means you need to have a different number of gears for one side than the other to get a different direction of rotation. In this setup, there are four gears leading to the right side and five gears leading to the left side. This is why there is an "extra" 8t gear on the left side.



**Front Subassembly Step 10:** Add two #6 axles with 24t crown gears and bushings.

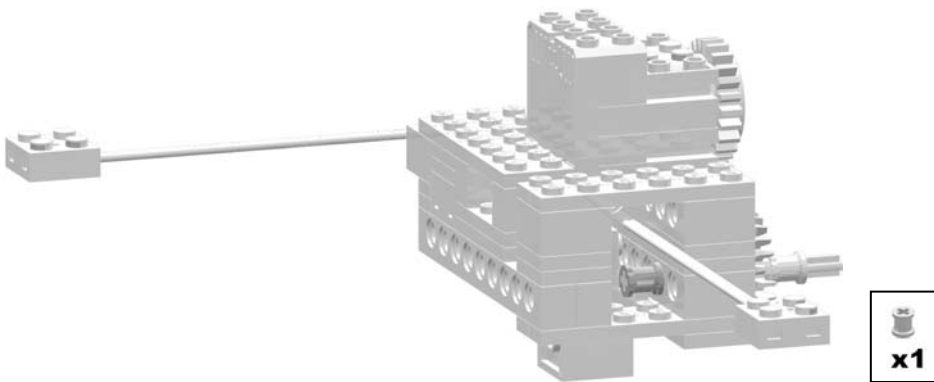
For step 11, rotate the model to where you are looking at the gap in the left side. You are now going to add two pieces called *cams*. These pieces are commonly used to “click” touch sensors—they have a smooth shape that works perfectly for the task. Also, you can insert an axle into a cam in four different places. This makes cams ideal for pressing (or clicking) touch sensors, because you can adapt them to different situations where the sensor is closer, farther away, and so on. For XK3’s purposes, insert the axle in the second hole away from the pointed ends of the cams.



**Front Subassembly Step 11:** Add two cams to back of axle.

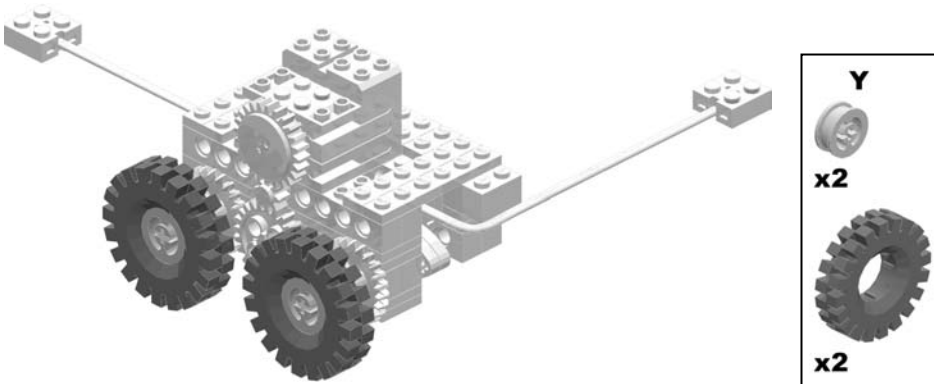
Next, rotate the model to the gap in the right side for step 12, and then add a bushing to the back of the #6 axle you inserted earlier.

**TIP** To insert both the cams and the bushing on the #6 axles, pull out the axle a little bit first, position the pieces with your fingers, and then push the axles into the pieces. It is much more difficult to squeeze the pieces onto the axles instead.



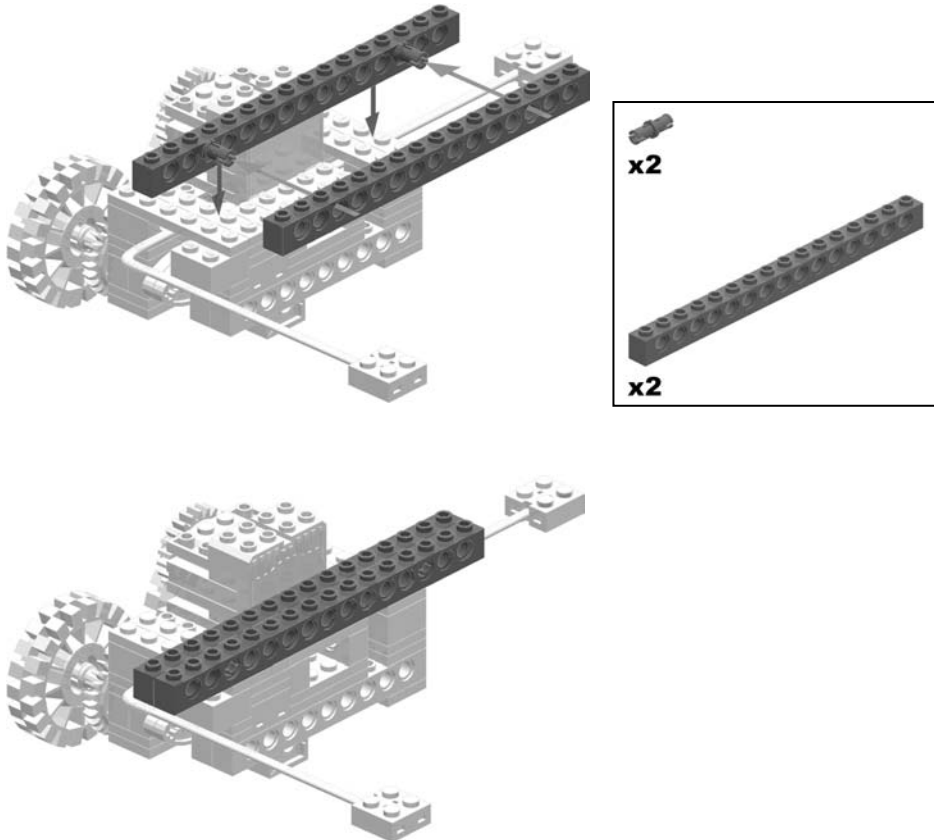
**Front Subassembly Step 12:** Add bushing to back of axle.

In step 13, rotate the model back to the front and add two of the medium-size, chunky wheels included in the RIS to the last remaining space of the #6 axles. I chose this type of wheel because of its light weight yet relatively large size.



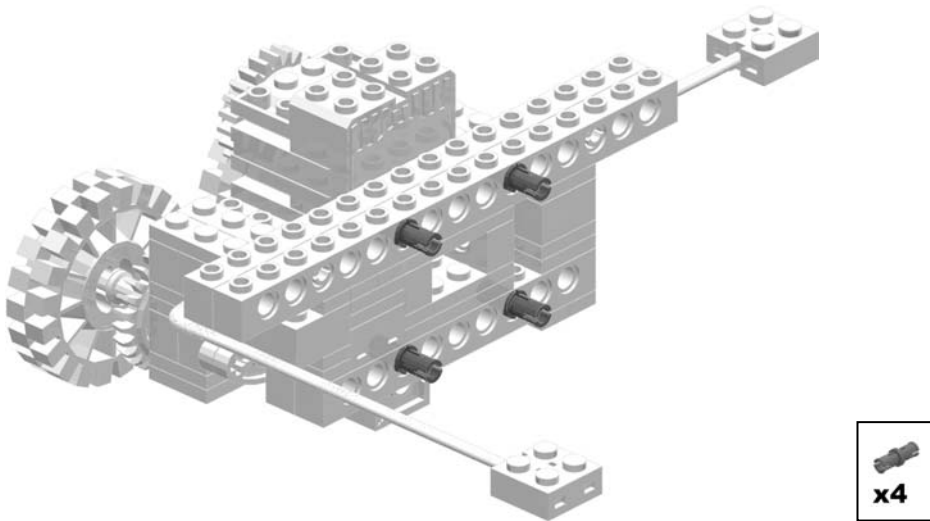
**Front Subassembly Step 13:** Connect two wheels to front of assembly.

After you've added the wheels, rotate the model to the back. You're going to finish the back of the assembly in steps 14 and 15. Pick out a 1x16 beam (the longest type) and add two black friction pins into it as shown in step 14, snap another 1x16 beam onto the pins, and then place everything onto the subassembly.



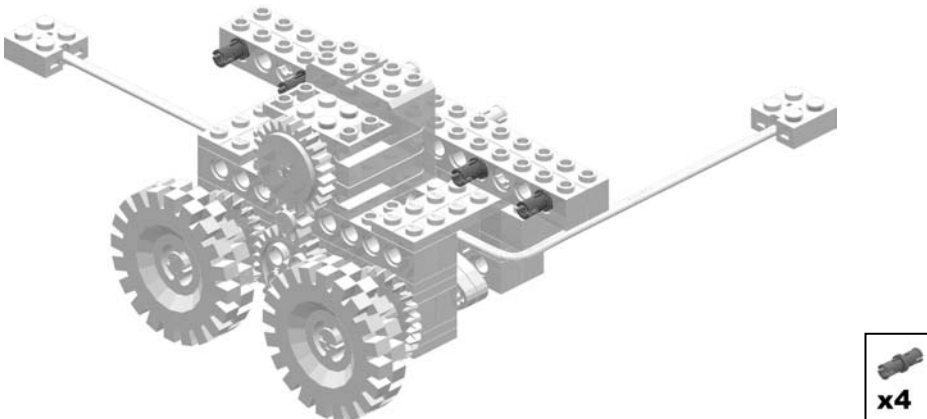
**Front Subassembly Step 14:** Snap two 1x16 beams together with friction pins, and add them to assembly.

Add four friction pins to the back of the subassembly as shown in step 15. This is what makes the subassembly modular. With these pins, you can simply plug it into or unplug it from the chassis.



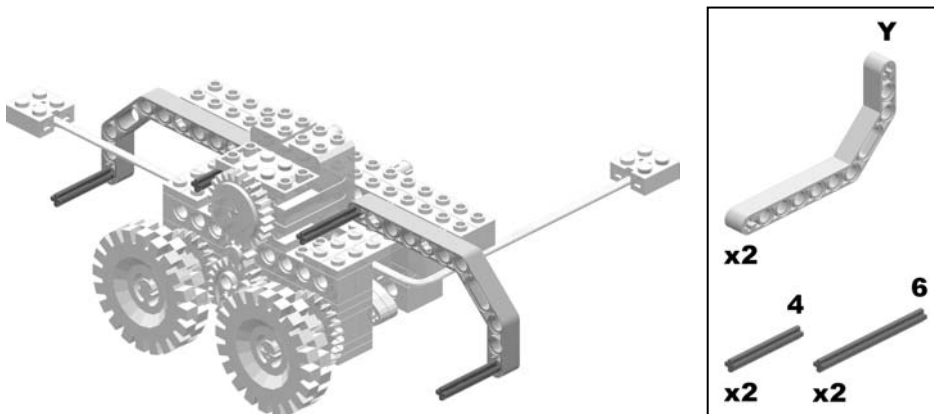
**Front Subassembly Step 15:** Snap four friction pins into back of assembly.

Rotate the model one last time to the front. You'll be working on the arms in the last few steps of this assembly—steps 16 through 19. First, in step 16 add four more friction pins; the arms will connect to these pins.

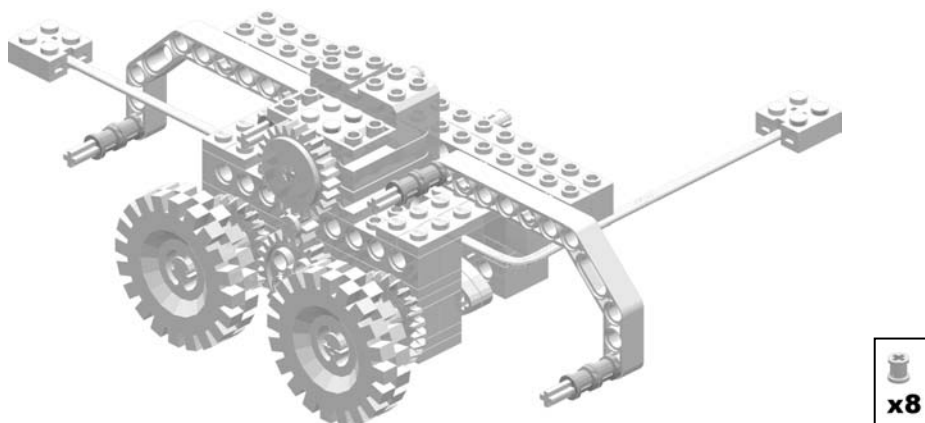


**Front Subassembly Step 16:** Snap four friction pins into front of assembly.

Steps 17 and 18 attach the actual arms and begin extending them with axles. More specifically, you use two #4 axles and two #6 axles. The #4 axles go on the ends of the yellow liftarms, and the #6 axles are the ones closest to the center. The #6 axles are positioned a distance of two studs into the beams behind the yellow liftarms.

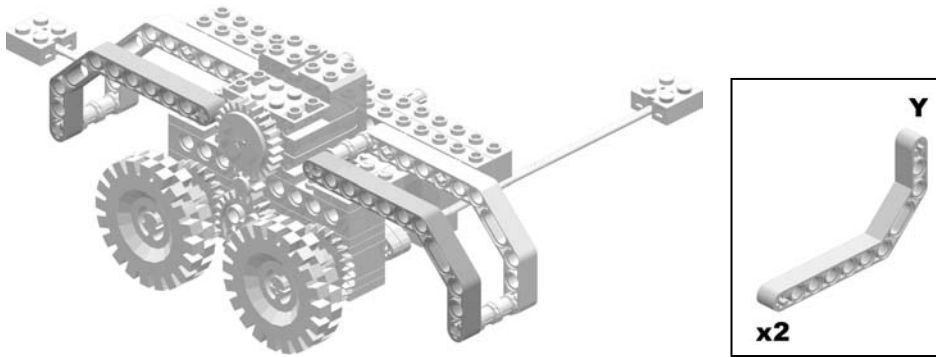


**Front Subassembly Step 17:** Add two yellow liftarms and #4 and #6 axles.



**Front Subassembly Step 18:** Slide eight bushings onto #4 and #6 axles.

Finish the front subassembly by adding two more yellow liftarms to the last remaining space of the axles, as shown in step 19.

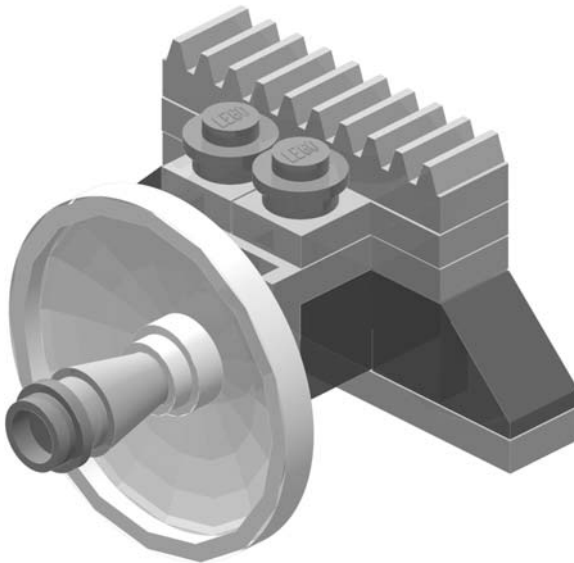


**Front Subassembly Step 19:** Attach two more yellow liftarms.

Once you've finished step 19, your front subassembly is complete, and it should look like the one shown in Figure 3, at the beginning of this section. Now, you're ready to move on to the eye subassembly.

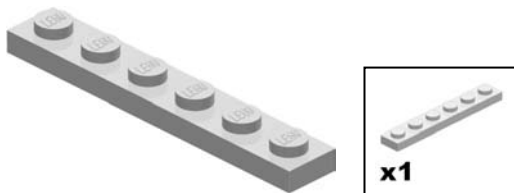
### *Eye Subassembly*

One word: cyclops. This subassembly, shown in Figure 4, is obviously for adding personality to XK3, but these kinds of assemblies are important aren't they? Just for fun, take this assembly off when XK3 is finished and see how it looks without it!



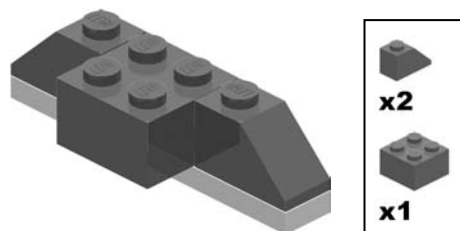
*Figure 4. The completed eye subassembly*

The eye subassembly is easy to build and consists of only a few pieces. In step 1, pick out and place in readiness a single 1x6 plate—think of it as a miniature base.



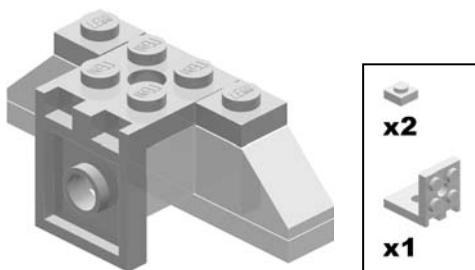
**Eye Subassembly Step 1:** Lay down 1x6 plate.

Next, add a black 2x2 brick and two black 1x2 sloped bricks onto the 1x6 plate, as shown in step 2.



**Eye Subassembly Step 2:** Connect a 2x2 brick and two 1x2 sloped bricks to 1x6 plate.

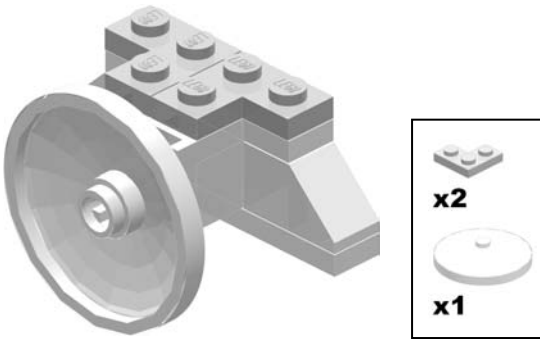
Add two 1x1 plates and an angle plate in step 3. The angle plate is an important piece for this assembly, as you'll see in the next step.



**Eye Subassembly Step 3:** Attach two 1x1 plates and angle plate.

Step 4 adds two corner plates and an “eye piece.” The (interesting) attachment of the “eye” to the angle plate deserves a little explanation. First, notice that there is a single stud on the back of the eye piece. Second, notice that on the front side of the angle plate there is a single hole in the middle—a hole that is just the right size for pushing single studs into. So, to connect the eye to the model, push the single stud into the angle plate’s hole. This attachment is perfect for eyes on a robot!



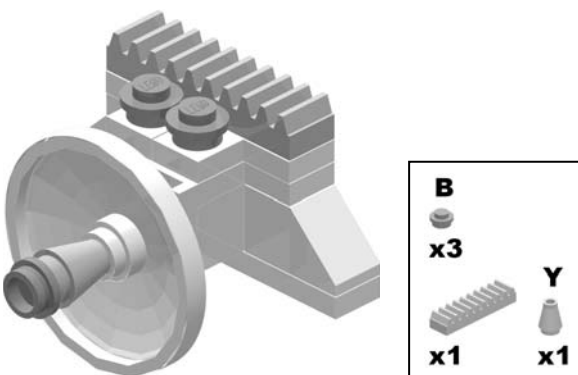


**Eye Subassembly Step 4:** Add two corner plates and white eye piece.

There are two parts to step 5. First, attach a *yellow cone* and a blue 1x1 round plate to the eye. To attach these to the eye, take the thinner end of the cone and push it into the eye's "hole" in the front, and then take the round plate and push its stud into the back end of the cone.

**TIP** You might think the type of connection the eye, cone, and round plate made is a rather odd one, but it really isn't. When dealing with "strange angles," connections like these abound. There are many more ways to attach many different types of LEGO pieces together than appear at first glance. This versatility should spark the spirit of creativity and experimentation in each of us. Always experiment and try new and interesting things!

The second part of step 5 adds two additional 1x1 round plates and a *gear rack*. The gear rack is definitely an interesting piece. Although here it is used for decoration, gears (such as 24t or 8t) can be used to make the rack move left and right, which generates a highly useful lateral movement.



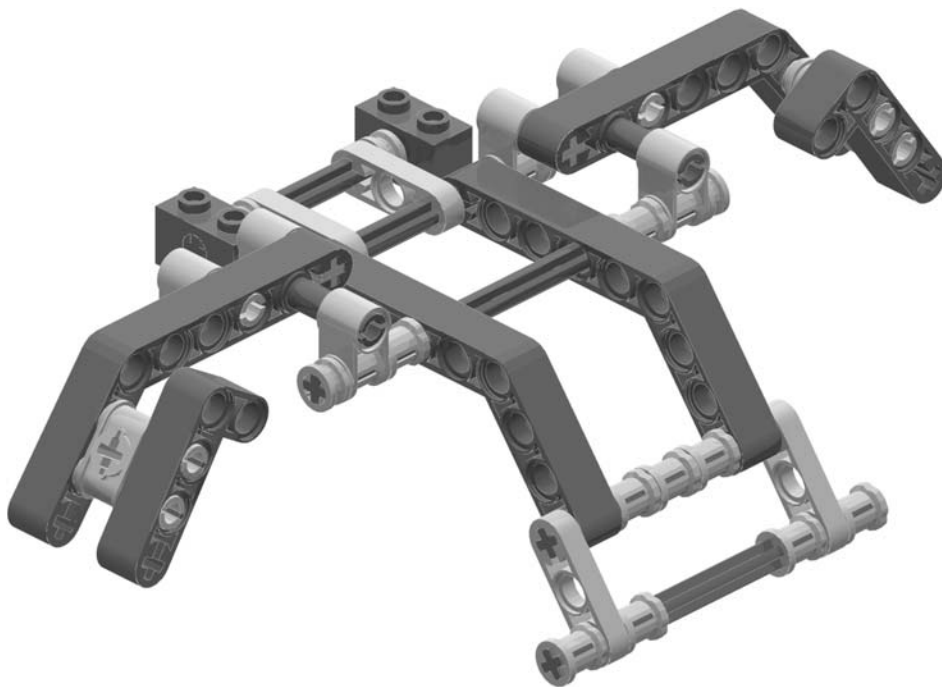
**Eye Subassembly Step 5:** Attach yellow cone, three blue 1x1 round plates, and gear rack.

Your eye subassembly should now look like the one shown in Figure 4, at the beginning of this section.

## *Back Slope Subassembly*

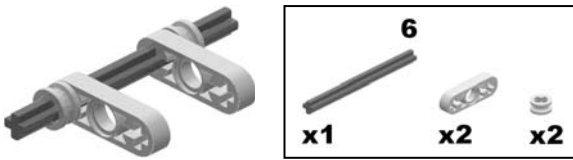
Zip-Bam-Bot Version XK1 (built in Chapter 4 of *Competitive MINDSTORMS*) uses a slope subassembly in the back. You are going to build something similar here. The design isn't the same, but the purpose is: if the sumo-bot backs up into the opposition, the opposition will go upwards on the slope.

Similar to the front subassembly, the back slope assembly is more like three assemblies in one, as shown in Figure 5. Included in this assembly are what I like to call *hazards*. These are the black parts protruding to the left and right. You might be wondering why you should use hazards. Well, there is nothing more hazardous than a hazard itself. In robotic sumo, axles, liftarms, and anything that simply points outward can pleasingly irritate opposing sumo-bots (or their creators). This is simply another timeless robotic sumo trick.



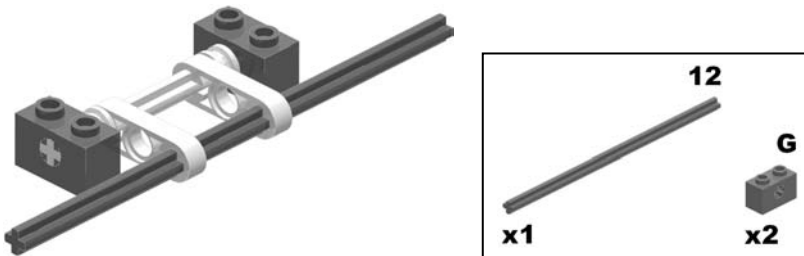
*Figure 5. The completed back slope subassembly*

Step 1 consists of an axle, 1x3 half-liftarms, and two half-bushings. Slide all the pieces together as shown.



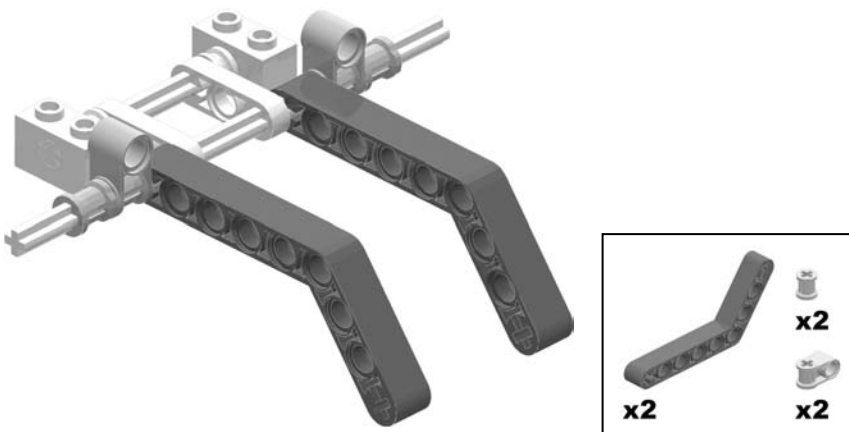
**Back Slope Subassembly Step 1:** Slide two half-bushings and 1x3 half-liftarms onto #6 axle.

Next, you'll add another axle—the largest size in the RIS, which is a #12—and some green axle bricks in step 2.

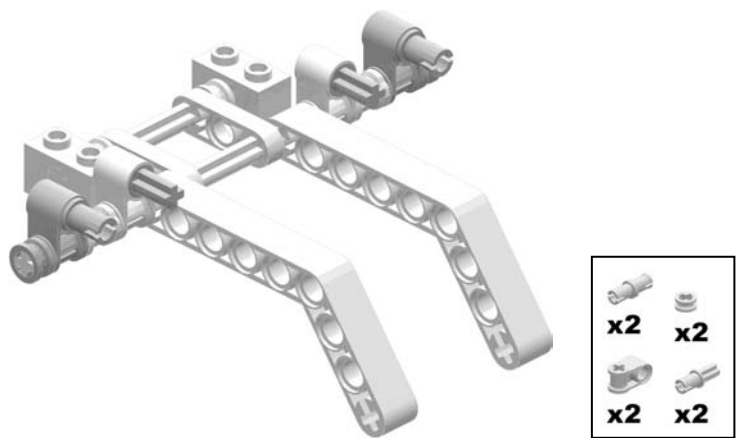


**Back Slope Subassembly Step 2:** Add #12 axle and two 1x2 green bricks with axle hole.

Step 3 constructs the actual slope, which is made out of liftarms, and introduces some of the very useful crossblocks. These crossblocks, and the crossblocks and pins in step 4, are what hold the hazards.

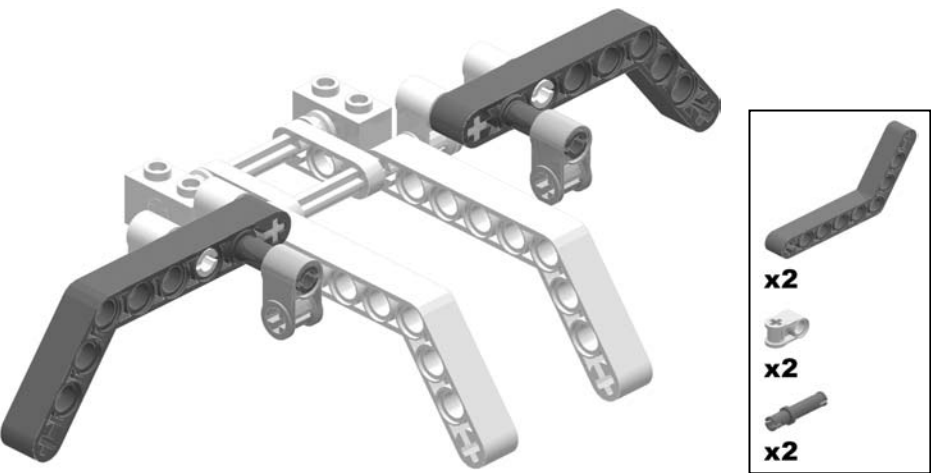



**Back Slope Subassembly Step 3:** Attach two black liftarms, crossblocks, and bushings.



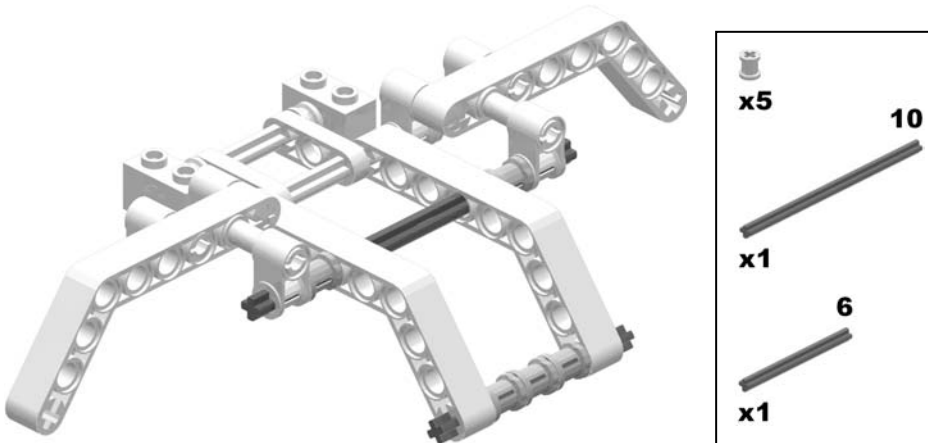
 **Back Slope Subassembly Step 4:** Attach two crossblocks, axle pins, regular pins, and half-bushings.

In step 5, after you have the pins in place, pull out yet two more of those black liftarms, and snap them right onto the pins. Also be sure to add the long friction pins and crossblocks.



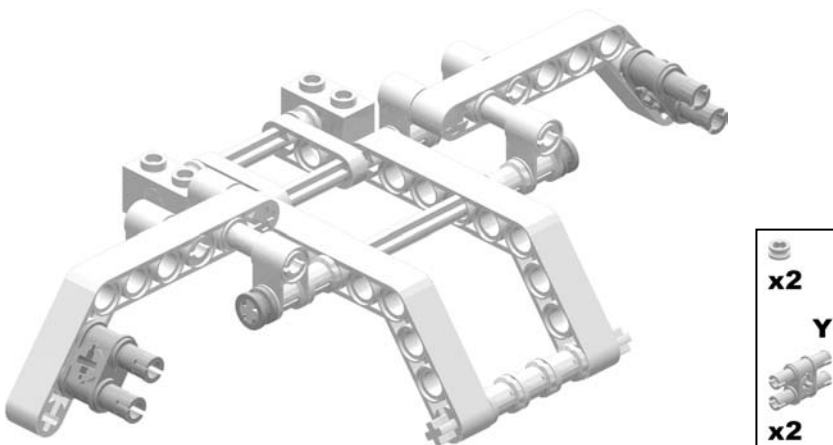
 **Back Slope Subassembly Step 5:** Connect two black liftarms, long friction pins, and crossblocks to assembly.

Step 6 inserts axles and bushings. Before you go any further, a little explanation is in order: the crossblocks you just pushed an axle through strengthen the hazards. Without these pieces and the long friction pins, there is the possibility that the hazards could get knocked off.



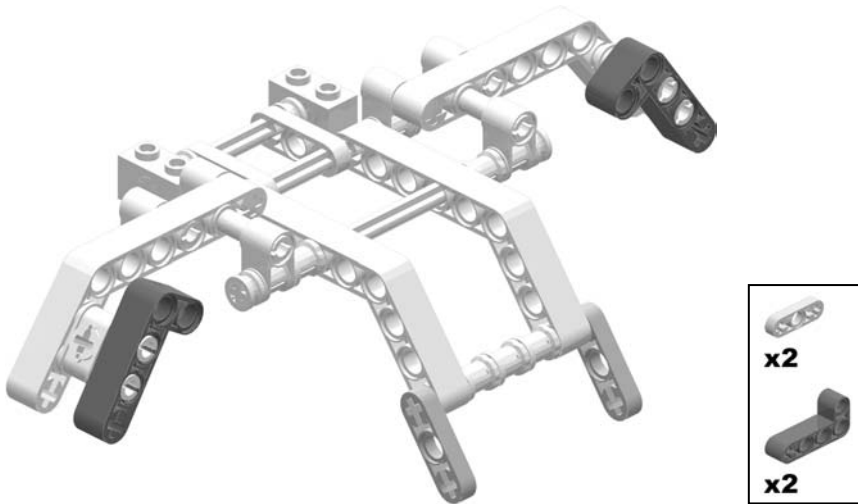
**Back Slope Subassembly Step 6:** Add #10 axle, #6 axle, and five bushings.

Step 7 continues construction of the hazards and tops off the ends of one of the axles with half-bushings.



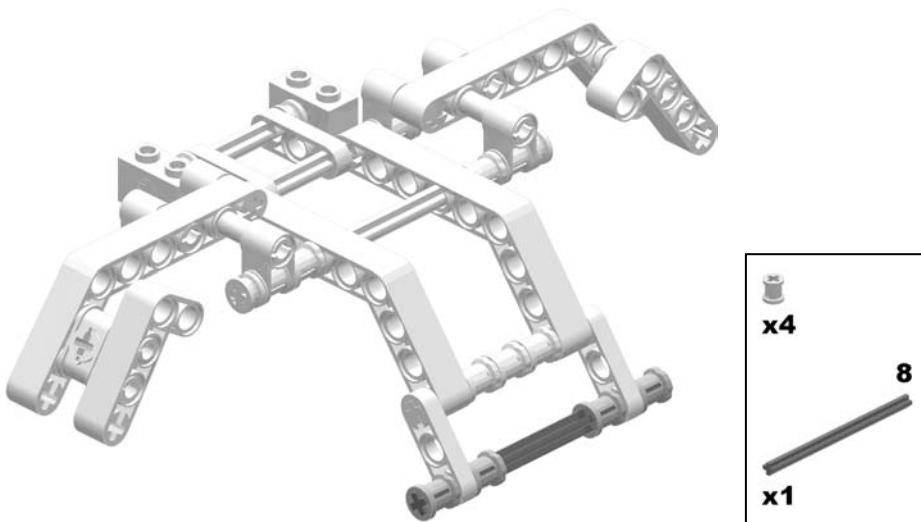
**Back Slope Subassembly Step 7:** Snap two 3L double pins into liftarms and top off axle with two half-bushings.

Step 8 finishes the hazards and begins bringing the slope out an additional distance of two studs.



**Back Slope Subassembly Step 8:** Add two 2x4 L-shaped liftarms and two 1x3 half-liftarms.

In step 9, finish off the slope, and when you finish the slope, you finish the sub-assembly! It should look like the one shown in Figure 5, at the beginning of this section.



**Back Slope Subassembly Step 9:** Add #8 axle and four bushings.

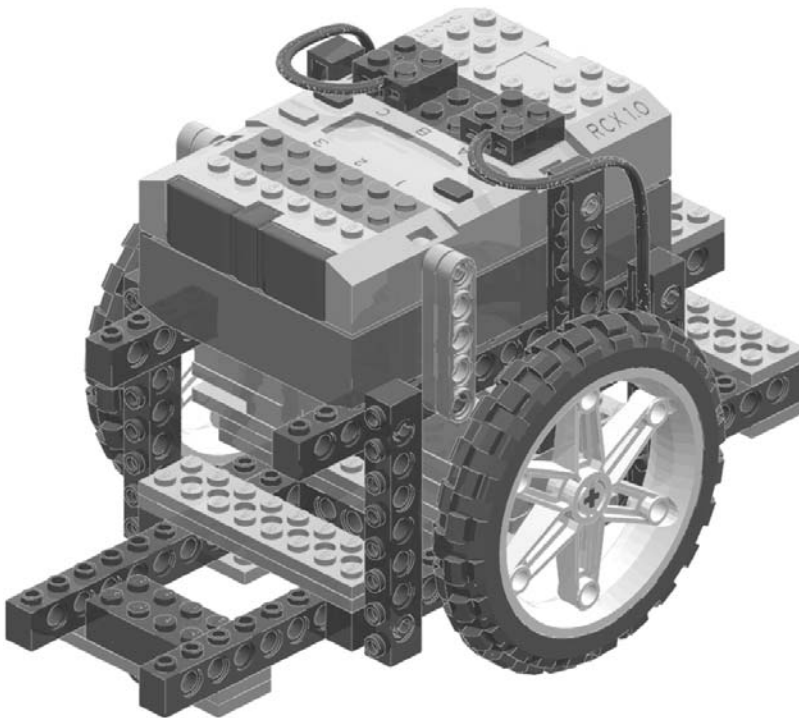
All the subassemblies have now been built, and you're ready to put XK3 together!

## Putting XK3 Together

Parts of this final assembly are easy to follow, but other sections are not quite as straightforward. So make sure to pay attention to the construction descriptions, because just looking at the images and trying to build XK3 from them might cause you some problems. Okay, let's start!

**NOTE** You will need the completed Zip-Bam-Bot chassis for the final assembly. See Chapter 3 of Competitive MINDSTORMS for the building instructions for the Zip-Bam-Bot chassis.

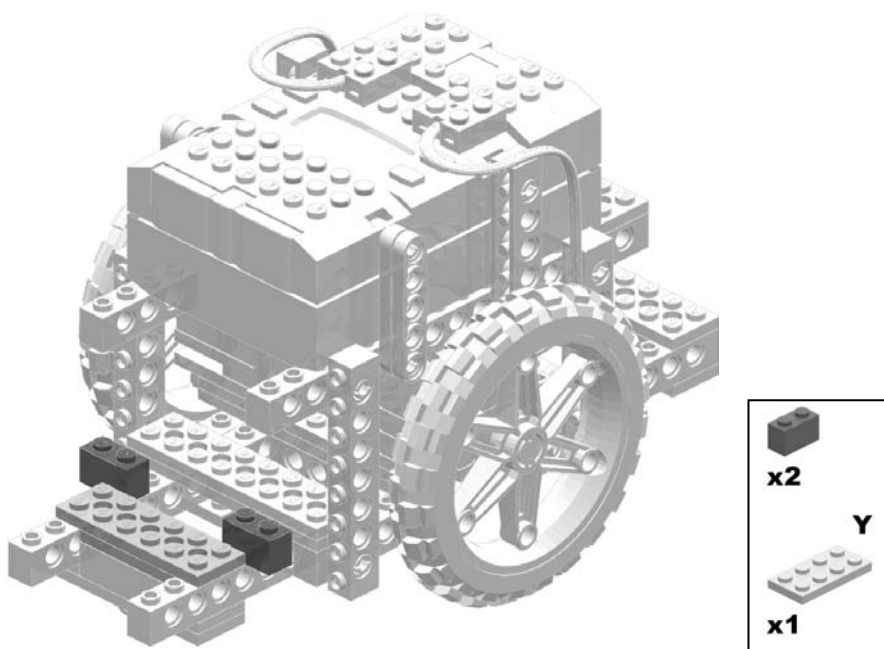
Step 1 introduces two 1x2 plates to the front end of the chassis. You need to first remove the pieces that are at the front end of the chassis, and then add the new 1x2 plates. Next, replace the pieces you just removed—with the exception of the 2x6 TECHNIC plate that went on top of the 2x4 brick—attaching them to the 1x2 plates. The 2x6 TECHNIC plate will be making its reappearance in the next step.



**XK3 Final Assembly Step 1:** Modify front end of chassis.

Step 2 strengthens the pieces added in the previous step with the addition of a few plates, including the 2x6 TECHNIC plate. But it also does something else: it starts the construction of a brick wall (or beam wall) that the front subassembly will connect to.

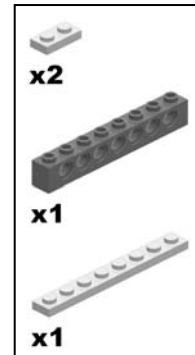
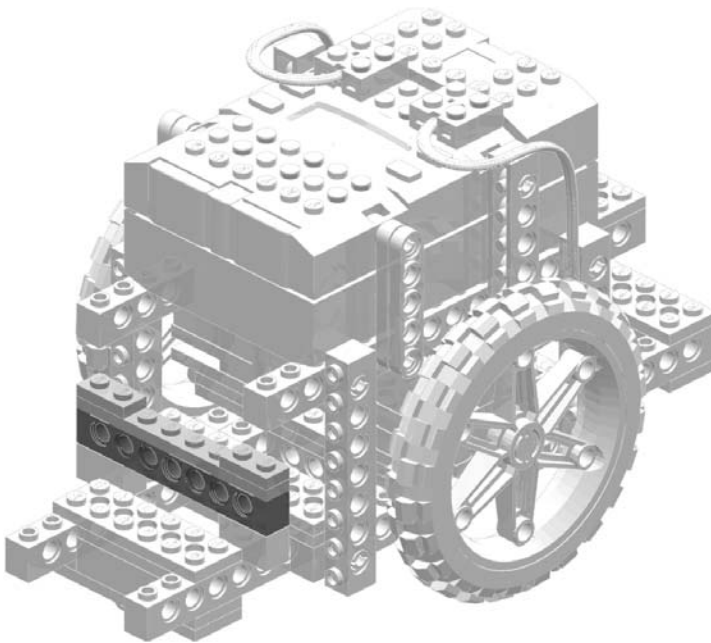
**NOTE** *The skid plate on the front is a little hard to see, but it is there. XK3 will be skidding, as all its predecessors did, so this skid plate is quite necessary. Also, you can remove the skid plate on the back, as it is now not needed; in this design, this portion never makes contact with the ground.*



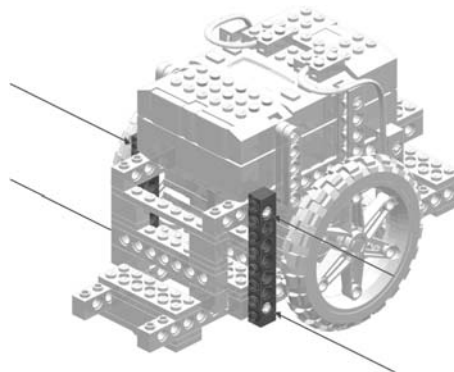
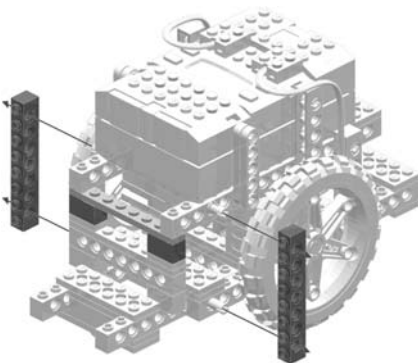
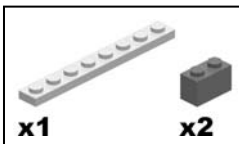
**XK3 Final Assembly Step 2:** Replace 2x6 plate, add a 2x4 plate, and add two 1x2 bricks.

Steps 3 through 5 continue building up the “beam wall” and add some gear racks. These racks give the sumo-bot personality and look like a mouth when the model is finished. Most of these steps are pretty easy, with the exception of step 4. Before you do this step, you must do something else first: take off the two front beams that brace the chassis. As shown in step 4, they are to the side. Removing them (temporarily) loosens the front end and gives you the room to put the pieces in place. Once the bracing beams are removed, lift up the top section of the chassis a bit, add your pieces, let the top section back down, attach it to the new pieces, and then simply reattach the beams that are responsible for the bracing.

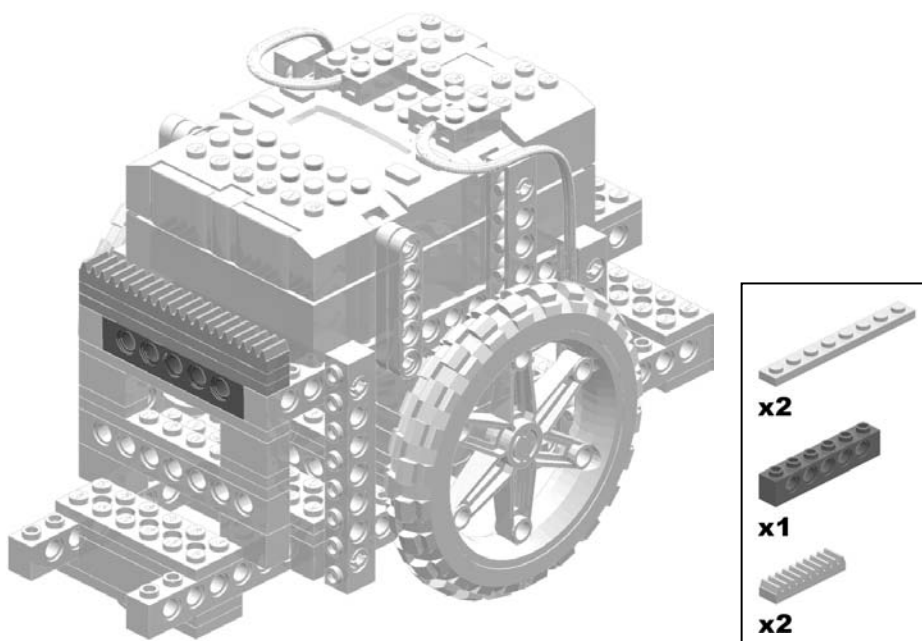




**XK3 Final Assembly Step 3:** Build up beam wall with 1x8 beam, 1x8 plate, and two 1x2 plates.



**XK3 Final Assembly Step 4:** Remove front bracing beams, add two 1x2 bricks and 1x8 plate, and then replace bracing beams.

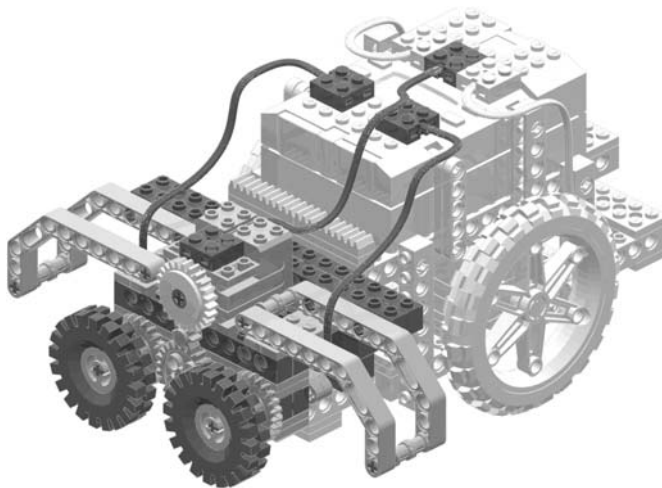
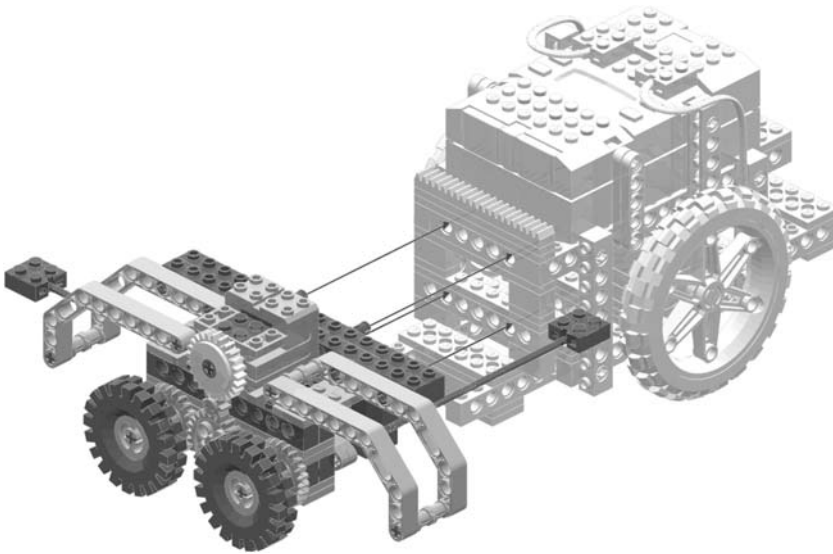
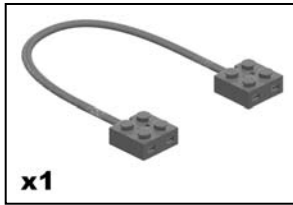


***XX3 Final Assembly Step 5:*** Finish beam wall with two 1x8 plates, 1x6 beam, and two gear racks.

Step 6 adds the **front subassembly** and shows you just how to attach it. There are four friction pins in this subassembly, and they snap right into the beam wall you just made. Follow the annotating arrows closely, as they point to which holes the pins should snap into. Once you have the front subassembly in place, connect a wire to its motor and attach all the front subassembly's wires to the RCX as indicated here:

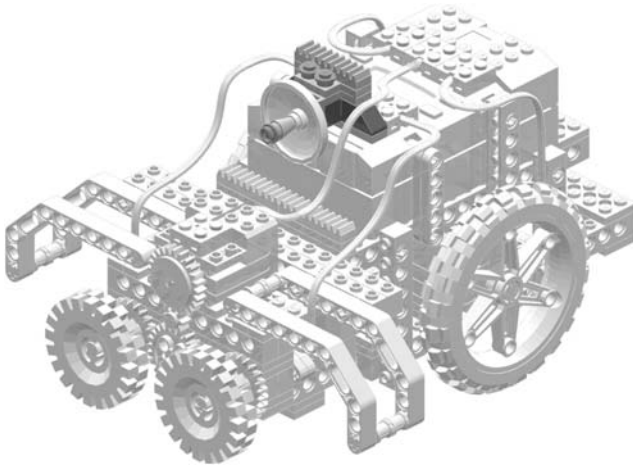
- The **touch sensor** goes on **input port 1**.
- The **light sensor** goes on **input port 3**.
- The **front subassembly's motor** goes on **output port B**.

**CAUTION** *Connect the wires to the RCX exactly as shown. If you do not, the sumo-bot will act strangely later on when you run the program, and that is something you definitely don't want to happen!*



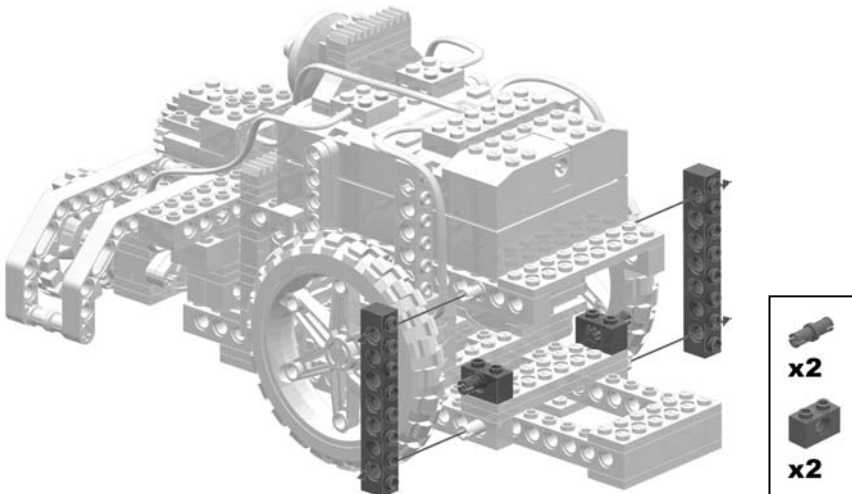
**XK3 Final Assembly Step 6:** Attach front subassembly to chassis.

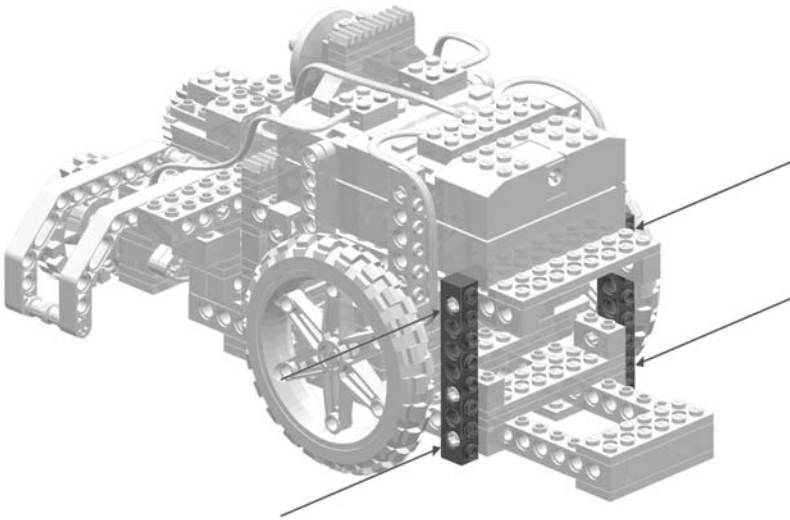
Add the **eye subassembly** in step 7. Simply connect it directly onto the RCX.



**XX3 Final Assembly Step 7:** *Attach eye subassembly to chassis.*

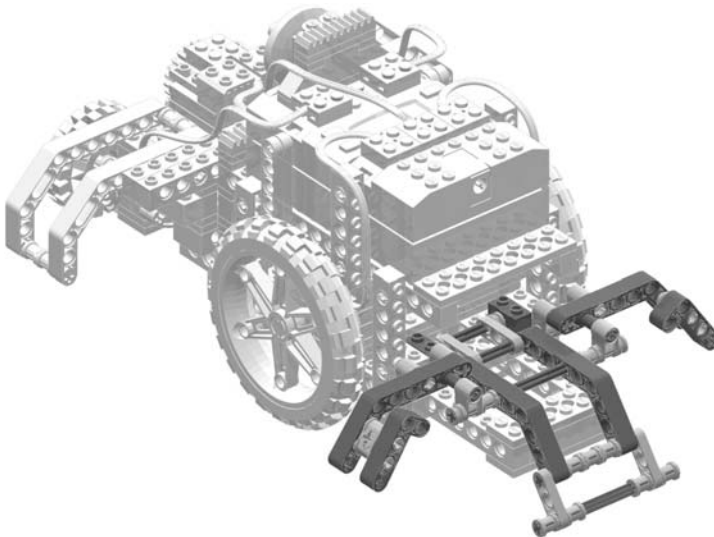
Step 8 provides another challenge: attaching the back slope subassembly. Actually, you're not adding this assembly just yet; you're preparing the chassis first. You'll do what you did just a few steps earlier: take off the bracing beams (you can also remove the wheels if necessary). This time, however, you'll take off the back bracing beams that go to the chassis. After you've removed them, place two 1x2 beams with two friction pins onto the chassis as shown. Reattach the bracing beams when you're finished, and notice that there is now a third friction pin that they snap into—the ones you just added.





**XK3 Final Assembly Step 8:** Remove back bracing beams, add two 1x2 beams and friction pins, and then replace bracing beams.

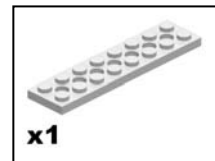
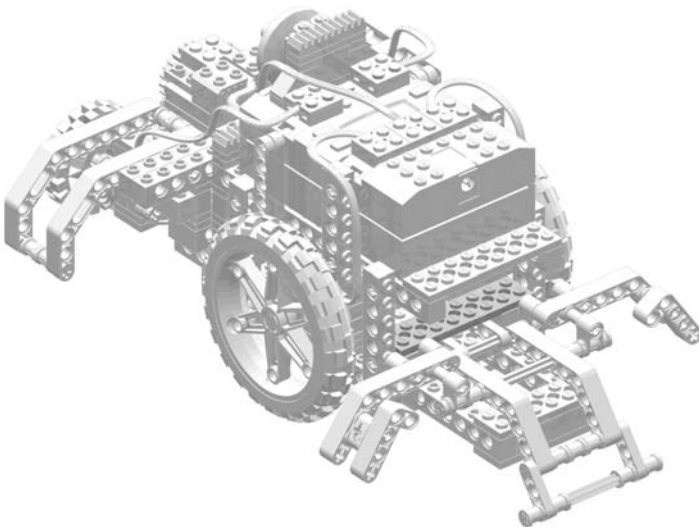
Place the **back slope subassembly** onto the chassis as shown in step 9. To do this, push the back slope subassembly's green axle bricks directly onto the chassis, next to the pieces you added in step 8.



**XK3 Final Assembly Step 9:** Attach back slope assembly to chassis.

Step 10 places a TECHNIC plate onto the back slope subassembly's green axle bricks. This plate goes across and onto the 1x2 beams.

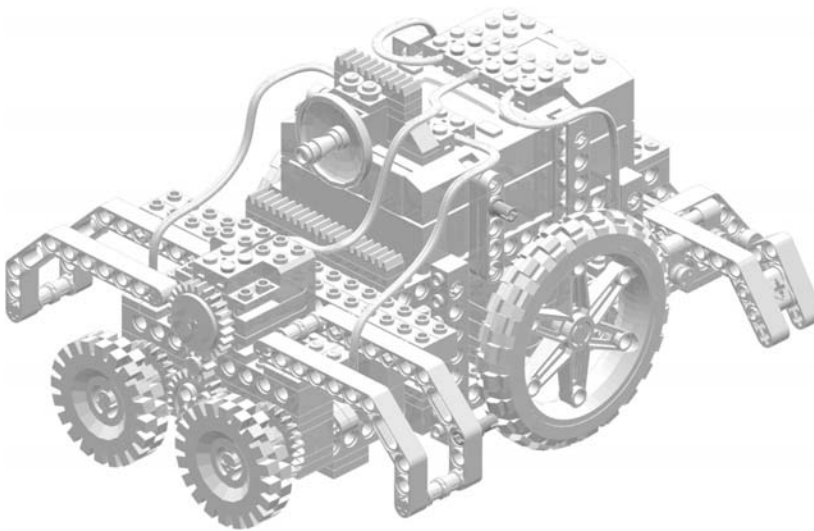
**NOTE** Do you see what happens in step 10? The back slope subassembly is indirectly reinforced with bracing! The 1x2 beams that you braced are “latched together” with the back slope subassembly, providing the assembly with some of their strength. I like to call this “class 2 bracing.” Although this isn't as strong as if the assembly were directly reinforced with bracing, it does provide much strength and is a great alternative for when you can't directly brace something.



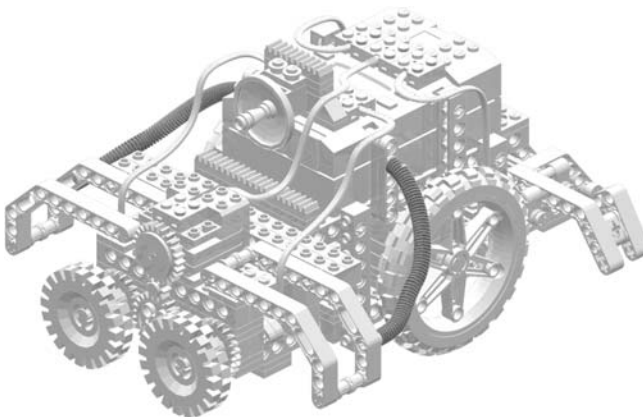
**XK3 Final Assembly Step 10:** Add TECHNIC plate to back slope subassembly's green axle bricks.

Steps 11 and 12 add pins and magenta ribbed hosing. A bit of colorful hosing often helps the appearance of a robot, and XK3 is no exception.

**TIP** The ribbed hosing used here is the commonly found magenta type in the RIS. However, feel free to use different colors and lengths; just realize that you'll need to slightly adjust the positioning of the pieces if you substitute other lengths of ribbed hosing.



**XK3 Final Assembly Step 11:** Snap four TECHNIC pins into front of model.



**XK3 Final Assembly Step 12:** Push ribbed hosing into the TECHNIC pins.

Congratulations and a job well done! You have now completed XK3. Your completed model should look like the one shown in Figure 1, at the beginning of the chapter. Now let's move on to the wonderful world of programming.



## Formulating a Solid Programming Solution for XK3

For XK3, we will obviously need the basic line-detecting and sumo-detecting tasks. But what isn't quite so obvious is how we should program XK3 to know when its CRWs have stopped spinning. Sure, you now know how the mechanical part of it works, but how is it going to work in programming?

Before we consider how to handle detecting when the CRWs stop, let's look at the whole behavior we have for XK3. Remember that XK3 will use the ram-and-ram-again tactics, as did its predecessors XK1 and XK2, so some of this programming is already thought out. Here are all the tasks we will be using:

**Line-detection task:** As usual, this task uses a light sensor to detect when XK3 has crossed over the line. However, in a second version of this program, we will use a second light sensor to cover the possibility of backing over the line.

**Impact-detection task:** This task is responsible for reacting to the other sumo-bot's interactions or XK3's interactions with the other sumo-bot. This task does not have something to do with the touch sensor in the front subassembly, but it doesn't monitor or keep track of the touch sensor's value. Instead, another task (the next one) monitors the touch sensor, and this task makes its decisions based on the information the other task passes to it.

**Touch-sensor-registry task:** This is the task that monitors the touch sensor and holds the key to knowing when the CRWs have hit something. This task does not have a priority, because it is always running and updating information.

**Maneuvering task:** As with XK2, we will use a maneuvering task that keeps some randomness in the sumo-bot's pattern of movements. It, too, could be useful for getting out of a bad situation.

This is how the program works: first, light sensor calibration is done. Second, all the motor outputs are turned on in the forward direction within a second after calibration is finished. If an enemy sumo-bot is detected by means of the CRWs, XK3 will continually ram it. Last, but not least, there is the line-detecting task (self-explanatory) and a maneuvering task that executes a random turn after a certain amount of inactivity.

After this first project, we'll tackle another variation: using a second light sensor for detecting the line. This addition can greatly benefit any small-and-fast sumo-bot. We will add the second sensor to the back of the robot and create new code in the "Programming XK3 with a Second Light Sensor" section.

## Programming XK3 with the Solution

XK3's overall approach is similar to that used in XK1's and XK2's programs (presented in Chapters 4 and 5 of *Competitive MINDSTORMS*), but the new equipment offers plenty of interesting opportunities. We will start off by working on the touch-sensor-registry and impact-detection tasks.



## Programming the Touch-Sensor-Registry and Impact-Detection Tasks

How can the program tell when the CRWs have been stopped? By using a special method, which is also used in Brain-Bot Version ZR2 (built in Chapter 7 of *Competitive MINDSTORMS*). ZR2 doesn't use bumpers, yet it can detect the opponent. It does this through touch sensors that are right up against the gears on its chassis. The touch sensors monitor the speed of those gears with the help of some programming. When the gears slow down, ZR2 can tell that it has run into the opponent. We're going to use a nearly identical setup with XK3.

Consider that the CRWs on XK3 will always be rotating—like the gears on ZR2's chassis—except for when they have made contact with the opponent. The cams you added in the front subassembly tap into the CRWs gearing, and thus are constantly pressing and releasing the touch sensor in the front subassembly, *except* for when the CRWs have made contact with the opponent. Therefore, using the touch sensor and a task from ZR2's program, `Spin()`, XK3 can detect when the CRWs have stopped or slowed down, and thus can detect the opponent. Observe the task `Spin()`:

```
// this task watches the touch sensor
task Spin()
{
    while(true)      // infinite loop - continually check
    {
        until(Clicker==1); // until it hits sensor...
        until(Clicker==0); //...until it goes away from sensor...
        ClearTimer(1); // ...clear timer one
    }
}
```

Let's quickly review what this task does. After the touch sensor has been pressed and released, a timer is cleared. Now, a different task—the impact-detection task—is monitoring this same timer. After the timer has not been cleared for a specified amount of time, meaning the touch sensor has not been pressed and released in that same amount of time, the task knows that XK3 has run into something and proceeds to put it into “attack” mode.

Now take a look at XK3's impact-detection task:

```
// constant
#define TOUCH 2

// this task takes care of the readings the touch sensor provides
task Crash()
{
    // second highest priority
    SetPriority(1);

    while(true)
    {
```

```

OnFwd(Front); // turn on the counter-rotating wheels
start Spin;   // start the task that keeps track of touch sensor

until(Timer(1) > TOUCH); // until timer one > than .2 seconds

    acquire(ACQUIRE_USER_1)
    {
        Win();           // attack!!!
        PlaySound(SOUND_CLICK); // end of this task...
    }
}
}

```

That “specified amount of time” I mentioned earlier is stored in the TOUCH constant and holds the value of 0.2 second. So once the CRWs have stopped spinning, the touch sensor stops getting clicked—and when this has happened for more than 0.2 second, “attack” mode will commence.

When I was developing the first version of the Spin() task, I took a slightly different approach. After testing and thinking it through, I decided that I needed to change it because of a bug in the code. I will show you the previous “buggy” Spin() task here and explain what was wrong:

```

// this task watches the touch sensor
task Spin()
{
    while(true)    // infinite loop; continually check
    {
        if(Clicker==1) // when the cam piece hits sensor...
        {
            ClearTimer(1); // ...clear timer one
        }
    }
}
}

```

The culprit is `if(Clicker==1)`. If XK3 rams its opponent right at the moment when the cam piece is pressing the touch sensor, the timer gets continually cleared! If that happens, XK3 will never even know it has hit something. This is important—it’s not just some minor detail. Consider this scenario: XK3 has hit the opponent, doesn’t detect it, continues giving full power to its motors, and yet stays still because it can’t push the other sumo-bot. Not only does this leave XK3 unproductive and prone to attack, it might damage the motors if it goes on for an extended period of time. The point is that you need to pay attention to details no matter how small they seem.

**NOTE** *Remember that spending a lot of time building and programming a sumo-bot is well worth the results.*

## Putting the Program Together

Now let's look at the complete program, shown in Listing 1.

*Listing 1. XK3\_Access\_Control.nqc*

```
// XK3_Access_Control.nqc
// A sumo-bot program for Zip-Bam-Bot Version XK3

// motors
#define Left OUT_A
#define Right OUT_C
#define Front OUT_B

// sensors
#define Clicker SENSOR_1
#define See SENSOR_3

// constants
#define BACK_TIME 80
#define WAITER 50
#define TURN 65
#define AMOUNT 3
#define LIMIT 25
#define LIGHT_BACK 60
#define TOUCH 2

// variables for calibration
int line=0,threshold=0;

// main task
task main()
{
    // initialize sensors
    SetSensor(Clicker,SENSOR_TOUCH);
    SetSensor(See,SENSOR_LIGHT);

    // calibrate light sensor
    Calibrate();

    // start motors
    OnFwd(Left+Right);

    // clear the timers we are going to use
    ClearTimer(0);
    ClearTimer(1);

    // start tasks
    start LWatch;
    start Crash;
```

```
    start Maneuver;
}

// this task watches for the line
task LWatch()
{
    // highest priority
    SetPriority(0);

    while(true) // infinite loop - always check for line
    {
        until(See<=threshold); // until light sensor detects line

        acquire(ACQUIRE_USER_1)
        {
            Avoid(); // avoid the line!!!
            PlaySound(SOUND_UP); // end of this task...
        }
    }
}

// this task is the "attacking" task
task Crash()
{
    // second highest priority
    SetPriority(1);

    while(true) // infinite loop - continually check
    {
        OnFwd(Front); // turn on the counter-rotating wheels
        start Spin; // start the task that keeps track of touch sensor

        until(Timer(1) > TOUCH); // until timer one > than .2 seconds

        acquire(ACQUIRE_USER_1)
        {
            Win(); // attack!!!
            PlaySound(SOUND_CLICK); // end of this task...
        }
    }
}

// this task adds random to XK3's movement patterns
task Maneuver()
{
    // third highest priority
    SetPriority(2);

    while(true) // infinite loop - keep doing this!
    {
        until(Timer(0)>=LIMIT); // until timer 0 is >= 2.5 seconds
```

```

        acquire(ACQUIRE_USER_1)
        {
            Change();           // change direction
            PlaySound(SOUND_DOWN); // end of this task...
        }
    }

}

// this task watches the touch sensor
task Spin()
{
    while(true)           // infinite loop - continually check
    {
        until(Clicker==1); // until it hits sensor...
        until(Clicker==0); //...until it goes away from sensor...
        ClearTimer(1); // ...clear timer one
    }
}

// this function sets the robot up for another crash
void Win()
{
    Rev(Left+Right);
    Wait(BACK_TIME);
    OnFwd(Left+Right);
    ClearTimer(0);    // remember to clear the timers!
    ClearTimer(1);
}

// this function avoids the line
void Avoid()
{
    OnRev(Left+Right);
    Wait(LIGHT_BACK);
    Fwd(Left);
    Wait(TURN);
    OnFwd(Left+Right);
    ClearTimer(0);
    ClearTimer(1);
}

// this function changes the direction of the robot
void Change()
{
    Rev(Left);
    Wait(TURN+Random(40));
    Fwd(Left);
    ClearTimer(0);
    ClearTimer(1);
}

```

```
// this is the calibrating function
void Calibrate()
{
    until(Clicker==1);
    line=See;
    threshold=line+AMOUNT;
    until(Clicker==0);
    PlaySound(SOUND_CLICK);
    until(Clicker==1);
    until(Clicker==0);
    PlaySound(SOUND_CLICK);
    Wait(WAITER);
}
```

Although this program is relatively large, it runs very smoothly. With a program like this one, you could very well be satisfied with the results and performance. However, when you're trying to make the *ultimate* sumo-bot, you want to go a step further and *optimize*.

Optimization, in MINDSTORMS terms, is taking a specific type of robot—whether it's a line follower, wall follower, or sumo-bot—and doing everything possible so it will give the very best performance. Although our goal isn't to make XK3 the *ultimate* sumo-bot (that task would be a book in itself), what could we do to improve XK3's performance? A second light sensor perhaps?

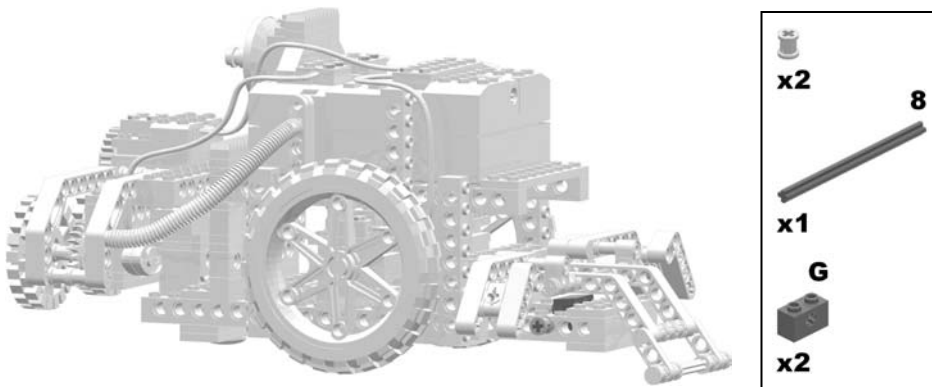
## *Programming XK3 with a Second Light Sensor*

For those of you who don't have a second light sensor, and don't intend to get one—at least for a little while—you can skip this section, or you can read it so you know what to do when you do obtain another light sensor. However, if you do have a second light sensor, get ready to use it! With it, you can solve that irritating problem of running backwards over the line. With the second light sensor positioned on the very back end of the chassis, XK3 knows when it is backing over the line; now you won't have to worry as much about backing into death! Before we program the second light sensor though, we need to build it into XK3.

### *Adding Another Light Sensor to XK3*

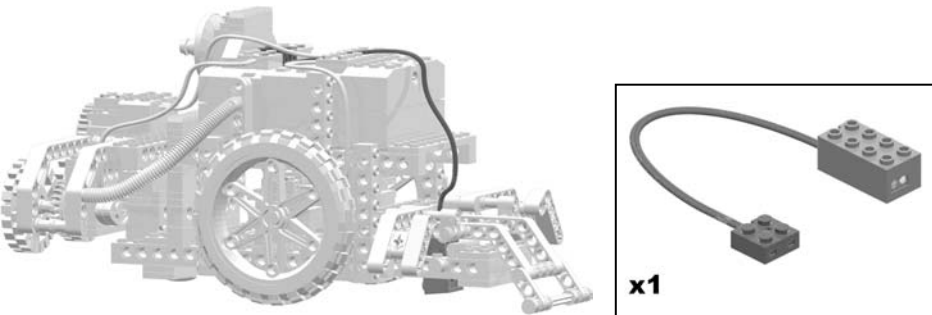
Before you gasp in horror at the sacrilegious act of building in a programming section, remember that you need to do this before going any further. Adding the second light sensor will take only a couple steps, a few pieces, and a minute or two.

To begin, on the back end of the chassis—the “brickish part” that juts out—add a #8 axle. Also add on the axle, in the middle, yet two more of those green axle bricks, and on the ends, two of those ever-present bushings. Step 1 illustrates the additions.



**Second Light Sensor Step 1:** Add #8 axle with two bushings and 1x2 bricks with axle hole to back of chassis.

Step 2 can be a little difficult because of the combination of cramped conditions and the light sensor's wire. However, it's quite doable. First, turn XK3 on its top. Then take the second light sensor and weave its wire all the way through the space directly in front of the green axle bricks you added in step 1. Then attach the light sensor to the green axle bricks, with one stud's space hanging over in the back and front, and connect its wire to **sensor input port 2**. With that, you're finished!



**Second Light Sensor Step 2:** Attach light sensor to green axle bricks and its wire to RCX.

**NOTE** On ZR2 (built in Chapter 7 of Competitive MINDSTORMS) both line-detecting light sensors needed to go on the same input port of the RCX, because of sensor port limitations. On XK3, we don't have those limitations, so we can give each light sensor its own port and its own task in the program.

## Adding Programming Support for the Second Light Sensor

With that little bit of construction completed, we need to think about the programming side (again). What is the best thing to do once the robot has detected the line with the back light sensor? Go forward! Going forward should move XK3 *away* from the line.

Therefore, we will have a new function that is executed whenever the line is detected with the second light sensor, which tells XK3 to immediately go forward. The function is called `Avoider()` and looks like this:

```
void Avoider()
{
    OnFwd(Left+Right);
    Wait(BACK_TIME); // 0.8 seconds
    ClearTimer(0);
    ClearTimer(1);
}
```

After the first action of going forward, the program is told to wait for 0.8 second, and then the timers that are being used in the program are cleared.

Now we need to create another task for watching the back light sensor. But what priority do we give the second line-detecting task? Line detecting is the most important type of task, which is why we gave the first line-detecting task the highest priority (0). Could we do this with the second line-detecting task as well? The answer lies in the official documentation of access control, which says that you can give two tasks the same priority. So we will give the new task a priority of 0.

**NOTE** *Don't worry about the situation where two line-detecting tasks are fighting for superiority. In all probability, the sumo-bot won't be seeing the line with both light sensors at the same time.*

Take a look at the new task:

```
// this task watches the back light sensor
task BLight()
{
    // highest priority
    SetPriority(0);

    while(true)    // infinite loop
    {
        until(See2 <= threshold);

        acquire(ACQUIRE_USER_1)
        {
            Avoider();           // change direction
            PlaySound(SOUND_CLICK); // end of this task...
        }
    }
}
```



With these chunks of code, we now have complete support for a second light sensor. Now all we need to do is add this code into our main program, which is easy to accomplish in an access control program. One of the benefits of this kind of program is that when you want to add another behavior, all you need to do is insert that new behavior, and you're finished. Another great feature is the ability to test just one of the behaviors to see if it works. You can do this by deleting or commenting out the other tasks.

The entire new program with support for a second light sensor is shown below in Listing 2.

*Listing 2. XK3\_Access\_Control\_Two.nqc*

```

/* XK3_Access_Control_Two.nqc - A sumo-bot program for Zip-Bam-Bot
   Version XK3 with support for a second light sensor */

// motors
#define Left OUT_A
#define Right OUT_C
#define Front OUT_B

// sensors
#define Clicker SENSOR_1
#define See SENSOR_3
#define See2 SENSOR_2

// constants
#define BACK_TIME 80
#define WAITER 50
#define TURN 65
#define AMOUNT 3
#define LIMIT 25
#define LIGHT_BACK 60
#define TOUCH 2

// variables for calibration
int line=0,threshold=0;

task main()
{
    // initialize sensors
    SetSensor(Clicker,SENSOR_TOUCH);
    SetSensor(See,SENSOR_LIGHT);
    SetSensor(See2,SENSOR_LIGHT);

    // calibrate light sensors
    Calibrate();

    // start motors and tasks
    OnFwd(Left+Right);

```

```
// let's clear the timers we are going to use
ClearTimer(0);
ClearTimer(1);

// start tasks
start LWatch;
start Crash;
start Maneuver;
start BLight;
}

// this task watches for the line
task LWatch()
{
    // highest priority
    SetPriority(0);

    while(true) // infinite loop - always check for line
    {
        until(See<=threshold); // until light sensor detects line

        acquire(ACQUIRE_USER_1)
        {
            Avoid(); // avoid the line!!!
            PlaySound(SOUND_UP); // end of this task...
        }
    }
}

// this task watches the back light sensor
task BLight()
{
    // highest priority
    SetPriority(0);

    while(true) // infinite loop - always check for line
    {
        until(See2 <= threshold);

        acquire(ACQUIRE_USER_1)
        {
            Avoider(); // change direction
            PlaySound(SOUND_CLICK); // end of this task...
        }
    }
}

// this task takes care of the readings the touch sensor provides
task Crash()
{
    // second highest priority
```

```

SetPriority(1);

while(true)  // infinite loop - continually check
{
    OnFwd(Front); // turn on the counter-rotating wheels
    start Spin;  // start the task that keeps track of touch sensor

    until(Timer(1) > TOUCH); // until timer one > than .2 seconds

    acquire(ACQUIRE_USER_1)
    {
        Win();           // attack!!!
        PlaySound(SOUND_UP); // end of this task...
    }
}

// this task adds random to the robot's movement patterns
task Maneuver()
{
    // third highest priority
    SetPriority(2);

    while(true)  // infinite loop - keep doing this!
    {
        until(Timer(0)>=LIMIT); // until timer 0 is >= 2.5 seconds

        acquire(ACQUIRE_USER_1)
        {
            Change();           // change direction
            PlaySound(SOUND_DOWN); // end of this task...
        }
    }
}

// this task watches the touch sensor
task Spin()
{
    while(true)  // infinite loop - continually check
    {
        until(Clicker==1); // until it hits sensor...
        until(Clicker==0); //...until it goes away from sensor...
        ClearTimer(1); // ...clear timer one
    }
}

// this function sets the robot up for another crash
void Win()
{
    Rev(Left+Right);
    Wait(BACK_TIME);
}

```

```
    OnFwd(Left+Right);
    ClearTimer(0);    // remember to clear the timers!
    ClearTimer(1);
}

// this function avoids the line
void Avoid()
{
    OnRev(Left+Right);
    Wait(LIGHT_BACK);
    Fwd(Left);
    Wait(TURN);
    OnFwd(Left+Right);
    ClearTimer(0);
    ClearTimer(1);
}

// this function is used by the second line-detecting task to avoid the line
void Avoider()
{
    OnFwd(Left+Right);
    Wait(BACK_TIME);
    ClearTimer(0);
    ClearTimer(1);
}

// this function changes the direction of the robot
void Change()
{
    Rev(Left);
    Wait(TURN+Random(40));
    Fwd(Left);
    ClearTimer(0);
    ClearTimer(1);
}

// this is the calibrating function
void Calibrate()
{
    until(Clicker==1);
    line=See;
    threshold=line+AMOUNT;
    until(Clicker==0);
    PlaySound(SOUND_CLICK);
    until(Clicker==1);
    until(Clicker==0);
    PlaySound(SOUND_CLICK);
    Wait(WAITER);
}
```

I think you'll be very pleased with the performance of the second light sensor. With this back light sensor, the chances of XK3 backing over the line are greatly decreased.

## Testing XK3

Download both programs to the RCX, and then position XK3 in your arena with its light sensor in the front subassembly viewing the line. Also, place objects of varying weights and sizes in your arena.

Start the first program (XK3\_Access\_Control.nqc) and begin by calibrating the light sensor. Make sure XK3's light sensor in the front subassembly is still viewing the black, outer line on the arena. Then twist XK3's CRWs until you hear a click from the RCX; the proper readings have now been taken. Twisting the CRWs causes the touch sensor inside the front subassembly to be pressed and released. The calibration function is programmed to watch for that touch sensor to be pressed and released, and then take the reading.

Now set XK3 in the middle of the arena. Twist its CRWs once again, until XK3 emits another click, and in 0.50 second, XK3 should shoot forward. Watch XK3's movements. Once it hits something with its CRWs, it should detect the hit quite easily. The proper reaction should be backing up, followed by going forward. When it sees the line, it should turn around.

**NOTE** *In the time that elapses between XK3 hitting something and recognizing it is in that state, the motors are going full power, but the wheels, and ultimately XK3, are staying still (this occurs only if the object is a heavy one). Don't worry about this damaging your motors, as they will not be in this situation for very long. However, if your battery power is low, this could trigger the low battery power segment on your RCX's LCD. If this happens, just put in some fresh batteries. The segment will go away, and XK3's performance will improve as well.*

Let's manipulate some different situations. Get an object and, while XK3 is backing up, run the object against its back slope, just as another sumo-bot might do. If you press the object firmly on the ground, the slope probably won't work, but sumo-bots aren't machines that go along on the ground with an airtight seal. If you give the object a little more freedom, you'll get much better—and probably more realistic—results. Don't forget to test XK3's CRWs. Use a wide variety of objects for this test; believe me, you never know what your sumo-bots might run into!

After testing the first program to your satisfaction, you can move on to the second program (XK3\_Access\_Control\_Two.nqc)—the program that supports a second light sensor. However, before starting, you need a reality check: the chance of XK3 running backward over the line in a situation like this is about zero. But, having the true MIND-STORM spirit, this won't faze us in the least. We'll deal with that in just a moment. First, set up the program. Make sure the RCX is on the second program, and then press the Run button. Once again, do the light sensor calibration, in the same way as you did the calibration for the first program.

So how do you get XK3 to run backwards over the line? One way is to *drag* XK3 backwards over the line yourself, which is what you should do. But its wheels are almost always going forwards, so how will you know if it really has detected the line? That's simple: in the program, we used a `PlaySound(SOUND_CLICK);` command at the end of the task for the second light sensor. This means you should hear a little click as XK3 is running away from the line. So go ahead and drag XK3 across the line, perhaps by pulling on its back slope. Once you've heard the click, you know the task has been executed.

In reality, there is one situation that would most commonly cause XK3 to go over the line. This is if XK3 has rammed the opposing sumo-bot right by the line, and then proceeds to back over the line. You can simulate this situation with the opposing sumo-bot as your *hand*. Set up everything as described, get XK3 moving, and give XK3's CRWs a good long slap. After that, XK3 will start to back over the line; however, the second light sensor will detect the line and prevent it from backing over the line. Use your imagination to come up with other ways to test XK3's second light sensor.

## Considering the Pros and Cons of XK3's Design

After seeing programming, gears, and concepts, we're definitely ready to sum it all up. It's always important to look at the big picture and form vital conclusions about your sumo-bots.

### *The Pros*

Here are the advantages of XK3's design:

- CRWs help to repel enemies.
- Large, yellow claws protect its front end from aggressive sumo-bots.
- The slope in the back prevents back-ramming sumo-bots from doing much harm.
- The design of the front subassembly allows for effective, efficient, and protective placement of the light and touch sensors.
- XK3's drivetrain provides maximum speed and minimum friction.

## The Cons

The following are the disadvantages of XK3's design:

- Extra weight slows XK3 down more than the other Zip-Bam-Bot versions.
- XK3's wheels are unprotected from a direct hit.
- Weight on the front end causes XK3 to tip forwards; this causes friction and a less well-placed center of gravity (COG).

Compared with XK2, XK3's top speed (and average speed) is lower. However, with some fresh batteries, XK3 goes a nice, fast speed and is just as crazy on the arena as any other small-and-fast sumo-bot. The COG is less well placed since most of the extra weight is on the front, but this is not a big concern. Why is this? XK3 doesn't *push*, it *rams*. A well-placed COG is important for pushing, but for ramming, it's not quite as important.

## Conclusion

In this chapter, you built, programmed, and tested XK3: a small-and-fast sumo-bot that relies on the combination of using the classic, repeated ramming method and also a mechanism to defeat the opponent. You also saw how to properly outfit and program a small-and-fast sumo-bot with two line-detecting light sensors.

The features that XK3 possesses can be applied to many other sumo-bots. Feel free to implement the features and mechanisms you've seen here in your own sumo-bots. And remember that putting *two* line-detecting light sensors in a small-and-fast sumo-bot is one of the best things you can do. Not only does it greatly boost survivability, it gives your sumo-bot an advantage against opponents who have only one line-detecting light sensor.