BONUS CHAPTER 2

Brain-Bot Version ZR1

IT'S WIDE, POWERFUL, has two awesome bumpers on the front, uses pieces from the Extreme Creatures Expansion Pack to look like some crazy creature, and holds more sensors than your next door neighbor's robot. What is it? It's ZR1, shown in Figure 1, and it possesses something very amazing and special: a sumo-bot searching ability.

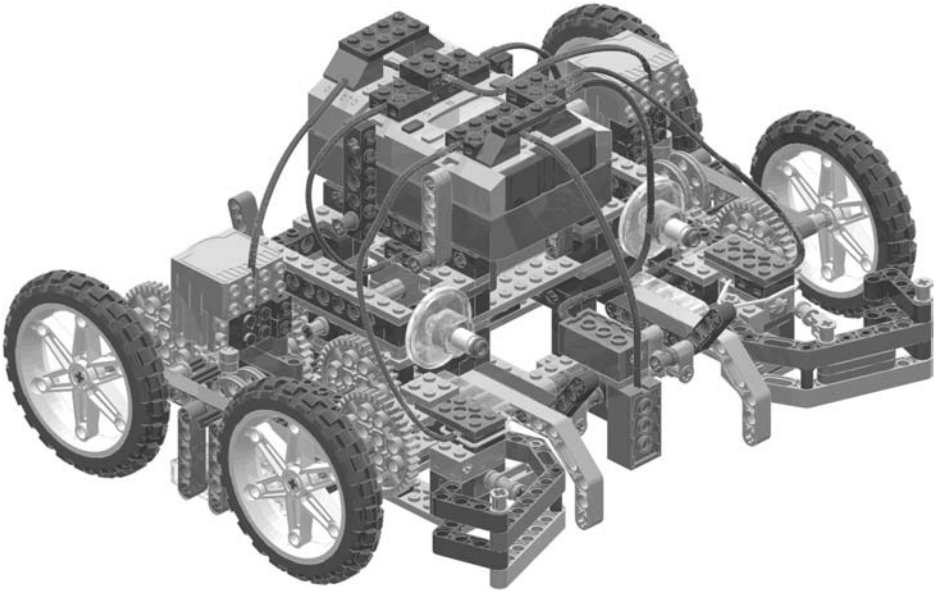


Figure 1. Completed Brain-Bot Version ZR1

One of the intriguing, useful, and exciting features of robotic sumo and the M-class strategy is the task of searching for the opponent. There are many ways to do this. Some electronics fans will create custom sensors that flood the arena with IR rays to find the opponent. LEGO purists may devise LEGO-only schemes that use electronics (LEGO sensors) to find the opponent. The nonelectronic searching approach is also common. This can include super-long "antennae" on the sumo-bot, as done in Chapter 9 of *Competitive MINDSTORMS* with Gargantuan-Bot Version BL58. In this chapter, we will be looking at the LEGO-only approach that uses electronics.

How Does ZR1 Search for the Opponent?

How will ZR1 be able to search for the opponent? ZR1 will be using a light sensor to search for the opponent—a *second* light sensor, since the first one is dedicated to watching for the line.

> **NOTE** *Although you could use a flipping light sensor (a sensor that searches for the line in one phase and the opponent in the other), using two light sensors fits ZR1's design better and is more efficient, since one light sensor can do only one thing at a time. Flipping light sensors are useful in certain situations, such as when the competition has a restrictive rule set that allows only one light sensor, but for ZR1, two light sensors are a better choice.*

"Okay," you might be thinking, "but *how* will the *light sensor* find the other sumo-bot?" I'll answer that question as I explain the part of ZR1's substrategy that is related to the searching light sensor. ZR1's one primary purpose (other than finding the opponent) is to position its searching light sensor directly at the opponent. ZR1 has two bumpers, and if they are activated, the program turns ZR1 in a direction that will (we hope) line up its searching light sensor with the opponent. This alignment serves two purposes:

- The middle section of ZR1's front end, where we station the searching light sensor, is perfect for pushing; ZR1 has specially designed subassemblies mounted there for that very purpose.

- It will be harder for the other sumo-bot to get away. If the opponent was, say, next to the right bumper, it could escape easily. But if the opponent is at the front, we *know* it is there because of our screaming cool, all-LEGO, electronic-distance sensor—the light sensor, that is.

It's commonly known among MINDSTORMS fans that you can use the RCX's IR message sender and the light sensor together as a distance sensor. However, you can also use the light sensor *by itself*, without the RCX IR message sender, as a distance sensor. Does this really work? Absolutely! When the opponent moves in front of ZR1's second, searching light sensor, it returns a much lower value than it normally would give. When we get the lower values, we know we have the opposition in sight. "Yes, but how far can the light sensor actually 'see'?" You might be surprised: the light sensor can detect sumo-bots at more than one foot! For this sumo-bot, we won't be basing our program around sumo-bots that are more than one foot away, but this shows you how sensitive the light sensor really is.

Now that you know how the searching light sensor can search, I will quickly summarize ZR1's substrategy as a whole. To help ZR1 search for the opponent, it uses a maneuvering task (which I'll discuss later), the searching light sensor, and its bumpers to get the opponent in sight. After both bumpers have been pressed at the same time or the searching light sensor has sighted the opponent, ZR1 will switch to slow speed.

It will stay on slow speed until it has either lost sight of the opponent or pushed the other sumo-bot out of the arena. Of course, there are a lot more technical details, but we will get to them later.

Constructing Subassemblies for ZR1

As noted at the beginning of the chapter, ZR1 uses parts from the Extreme Creatures Expansion Pack (ECEP); it also takes parts from the RIS and a few from the UBEP as well. Figure 2 shows the bill of materials for ZR1.

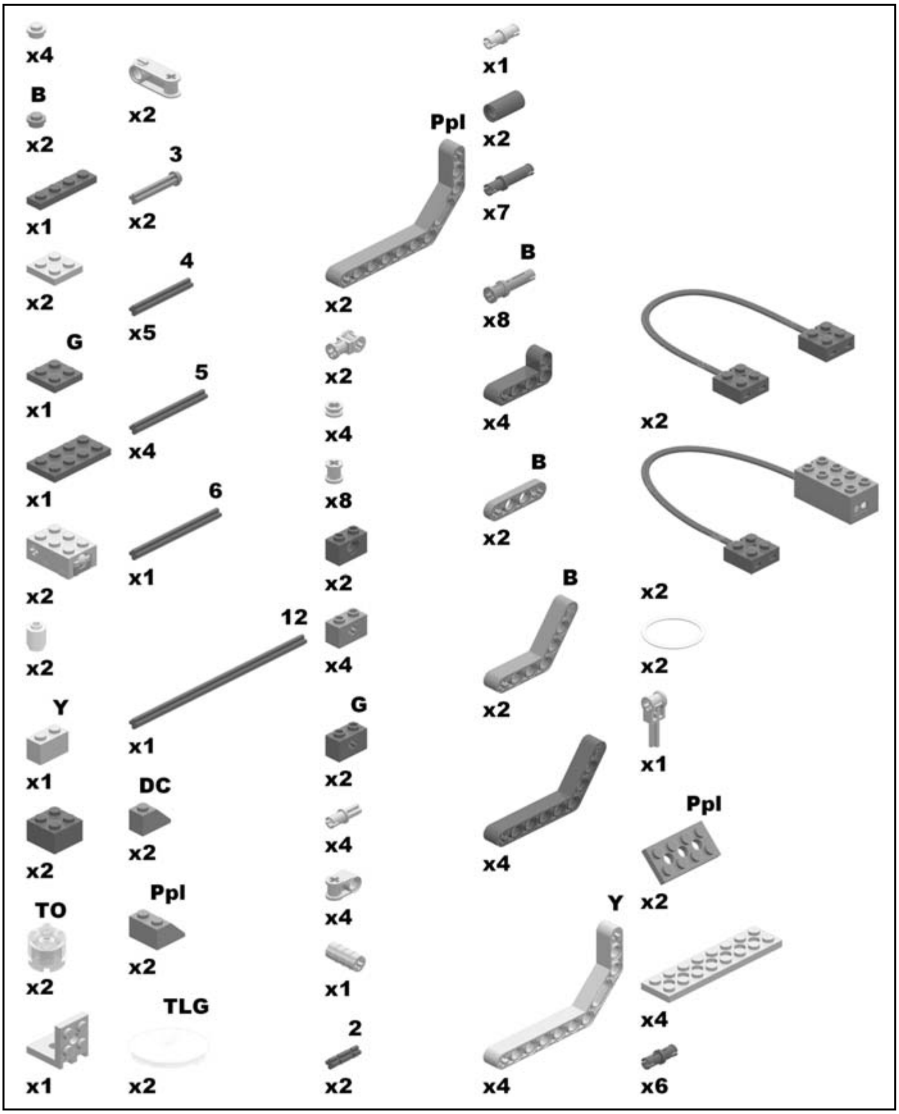


Figure 2. ZR1's bill of materials

NOTE *If you don't have the ECEP, you can still build ZR1 by substituting the ECEP parts with parts from the RIS. ZR1 was not built with many parts from the ECEP, and those parts are only for decoration, characteristics, and color schemes. As of this writing, the ECEP has long been out of production (it was released in 1998). If you would like to obtain the ECEP, a great place to look for it—and at a great price—is eBay.*

Notice that ZR1 uses four sensors. As you know from sumo-bots presented in *Competitive MINDSTORMS*, the "problem" of having more sensors than the number of sensor input ports on the RCX can be overcome in many ways. You'll see just how you can fit four sensors on ZR1 as you go through this chapter.

Although you'll construct a number of subassemblies for ZR1, the actual quantity of pieces involved is relatively small because most of the subassemblies are relatively small. The Brain-Bot chassis (see Chapter 6 of *Competitive MINDSTORMS*) is quite full featured, so that, in itself, is enough in the mechanical sense. The subassemblies for ZR1 include two pusher subassemblies, the light sensor mount subassembly, two bumper subassemblies, two eye subassemblies, and a back protector subassembly. These are what make up ZR1, along with a handful of other pieces you'll add in the final assembly.

NOTE *If you don't have a second light sensor, you're probably wondering where you can get one. There aren't any expansion packs that contain one, so it boils down to a couple choices. As of this writing, you can buy a second RIS, get one full price ($16) at* `www.plestore.com`*, or look for one on eBay or BrickLink (*`www.bricklink.com`*).*

The Right Pusher Subassembly

The right pusher subassembly, shown in Figure 3, has a perfect design for firmly pressing against the opponent's structure, as well as for not allowing any slopes to wreak havoc on ZR1. It may be a little difficult to visualize this very small subassembly accomplishing all that, but you'll be able to see firsthand how it works when ZR1 is complete. When combined with the other pusher assembly and strengthened with the light sensor mount subassembly, these pushers are very useful and effective indeed.

Steps 1 and 2—the only steps—consist of two different liftarms and a variety of pins. The pins that are "remaining" at the end of step 2 will connect to the chassis in the final assembly. The steps are self-explanatory and easy to follow.

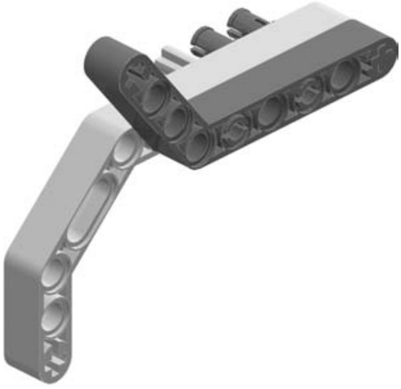
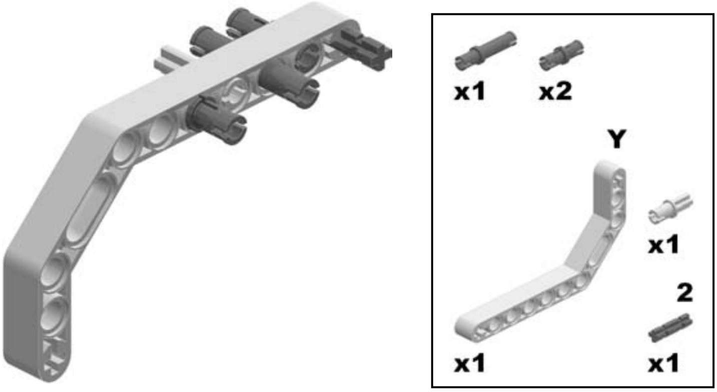
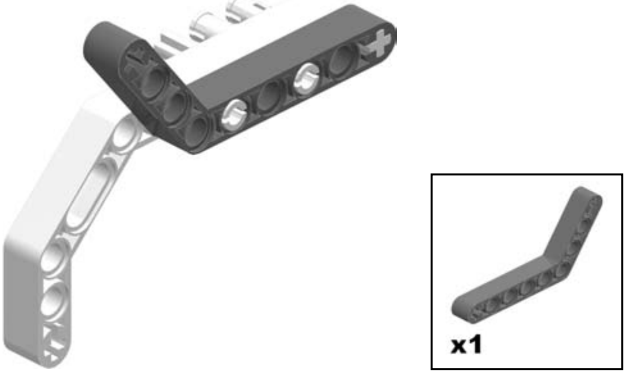


Figure 3. The completed right pusher subassembly



Right Pusher Subassembly Step 1



Right Pusher Subassembly Step 2

The Left Pusher Subassembly

The left pusher subassembly, shown in Figure 4, is the counterpart, and mirror image, of the right pusher subassembly.

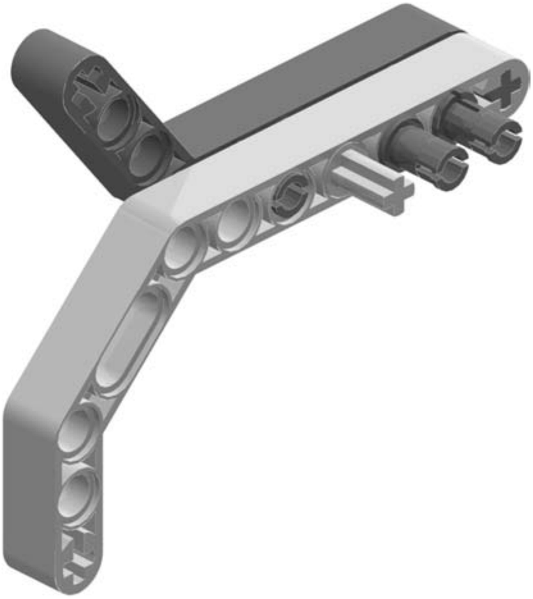
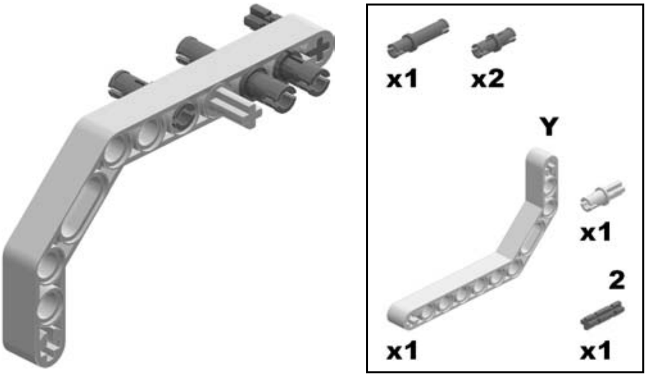
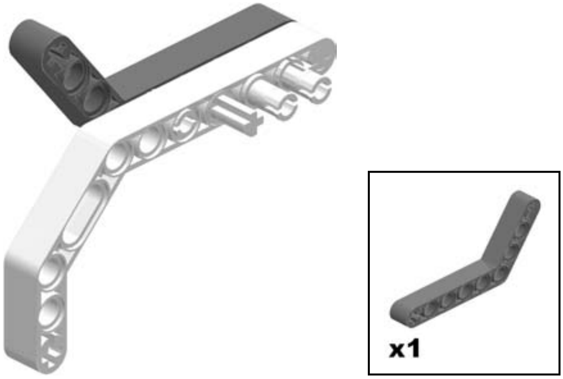


Figure 4. The completed left pusher subassembly

Two steps, two liftarms, and a couple of pins are all it takes to build this subassembly.



Left Pusher Subassembly Step 1



Left Pusher Subassembly Step 2

The Light Sensor Mount Subassembly

Notice this subassembly, shown in Figure 5, isn't called the light sensor subassembly; it's called the light sensor *mount* subassembly. This is due to the fact that there is more than one light sensor on this assembly. Like most of the other assemblies for ZR1, this one is modular. It includes four blue pins that are pushed into the chassis in the final assembly.

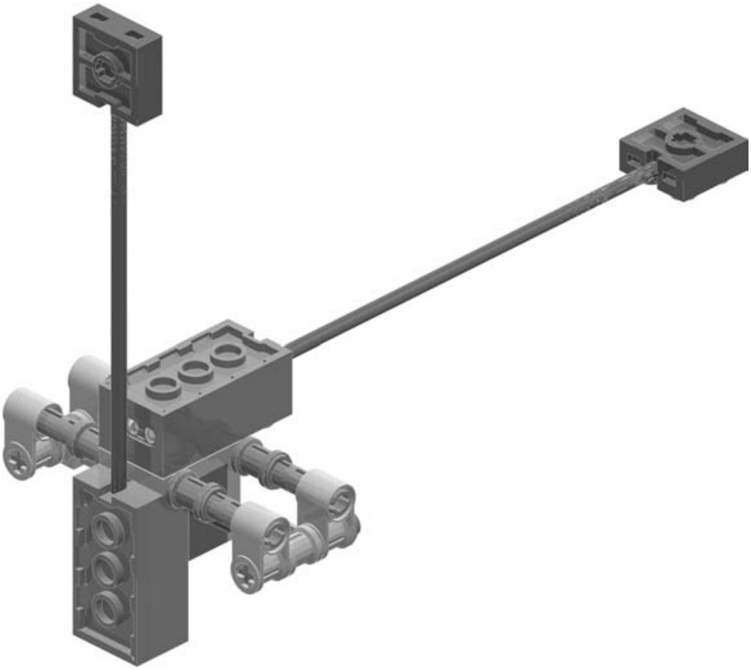
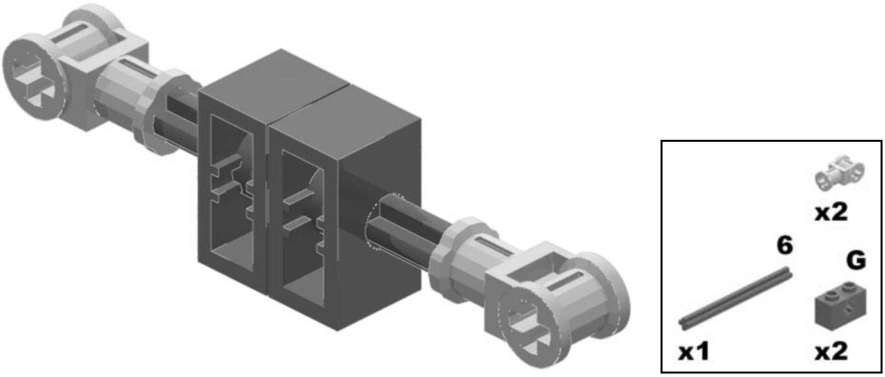


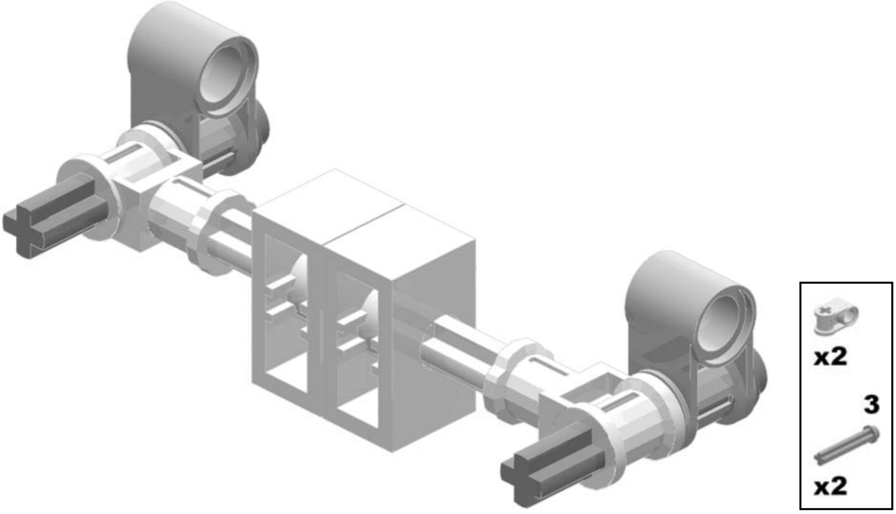
Figure 5. The completed light sensor mount subassembly

In step 1, you add those ever-present green axle bricks, a #6 axle, and two connectors with an axle hole. You'll notice the orientation of the two bricks: their studs are facing away from the front. Don't worry; this is intentional and essential, and it will become obvious why in just a few steps.



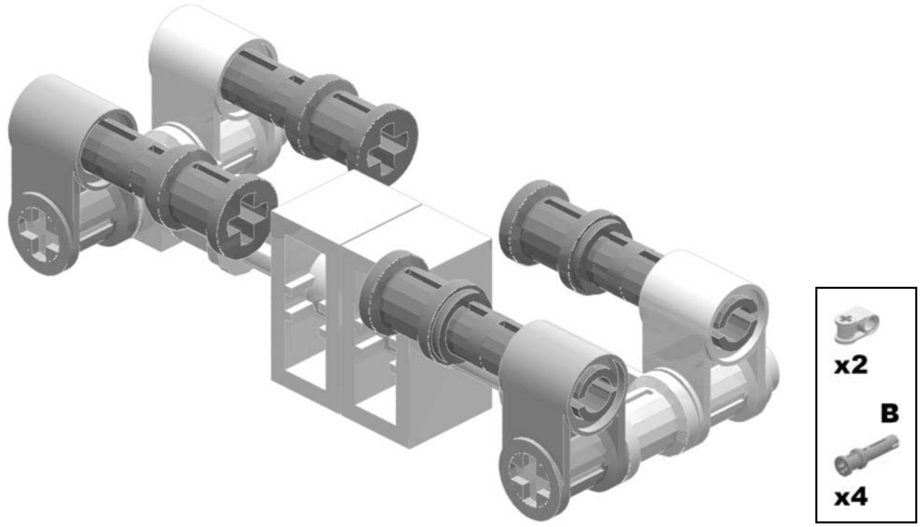
Light Sensor Mount Subassembly Step 1

In step 2, you add two #3 axles with a stud and two crossblocks. These parts, and the ones in the next step, will provide the means of attaching to the chassis.



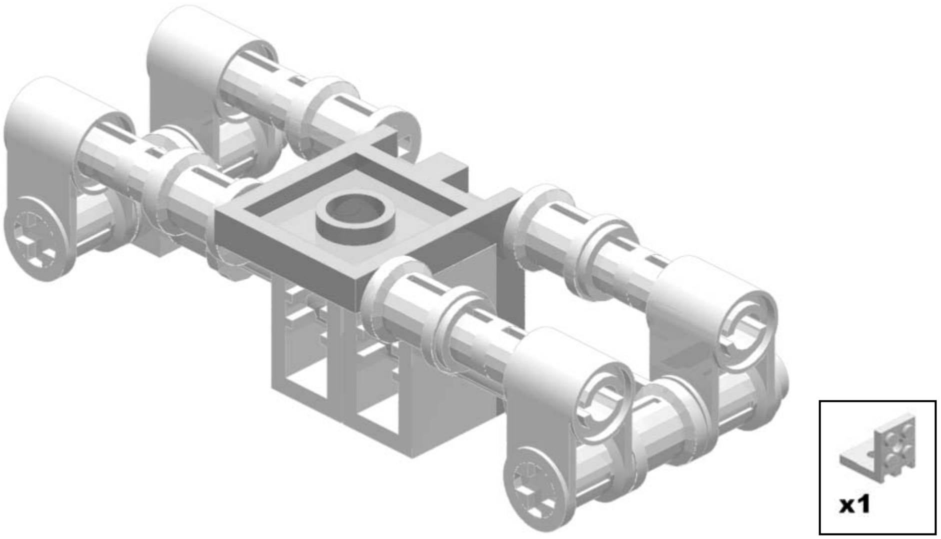
Light Sensor Mount Subassembly Step 2

Step 3 adds two more crossblocks and the blue pins to finish the little section started in step 2.



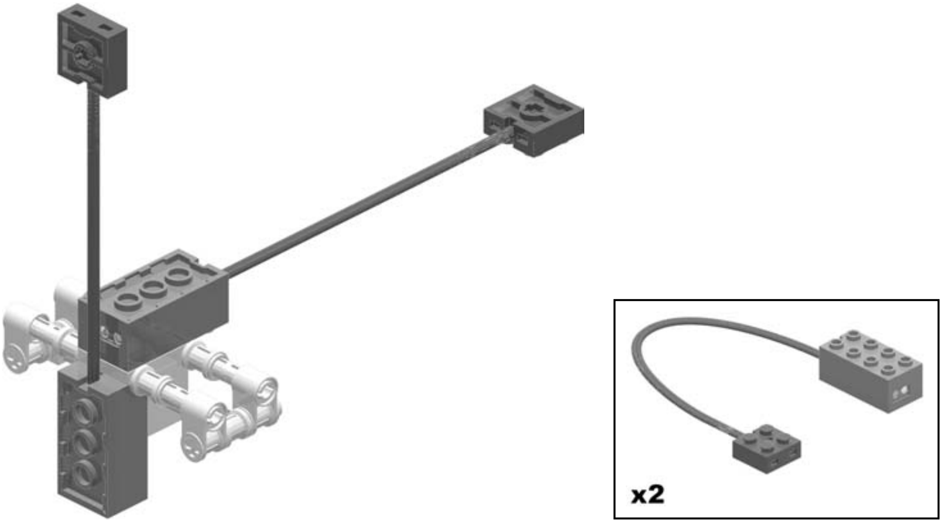
Light Sensor Mount Subassembly Step 3

Step 4 places the angle plate on the green axle bricks. This plate is how the light sensors will (properly) attach to the assembly.



Light Sensor Mount Subassembly Step 4

Step 5 adds the long-awaited light sensors and shows you just how to attach them.



Light Sensor Mount Subassembly Step 5

The Right Bumper Subassembly

Now that we've reached this point, let's take a moment to talk about the technical details regarding how we're going to fit four sensors on ZR1. To accomplish this, we'll put a light sensor and a touch sensor on the same input port, and configure the port as a light sensor. Using this method, the light sensor reads as it normally does, and when the touch sensor is pressed, the input port returns 100, thus allowing us to easily distinguish between the two sensor's readings. This is the same method used in Gargantuan-Bot Version BL58 in Chapter 9 of *Competitive MINDSTORMS*.

As was also discussed in Chapter 9 of *Competitive MINDSTORMS*, release-type bumpers are better than press-type bumpers, so it would be good if we could use release-type bumpers for our sumo-bot. However, we have the same problem here as we did with BL58: if we have *two* release-type bumpers, and if we use the special method to save the input port, the input port using the special method would always return 100 because a release-type bumper is hooked up to it. The solution is to use a release-type *and* a press-type bumper.

The right bumper, shown in Figure 6, is the better, release-type bumper. The left bumper for ZR1 is a press-type bumper and connects to the same port as the second light sensor. This means we can successfully hook up two sensors on one port. The price is that the pressing bumper on the left isn't as efficient as the release-type bumper on the right. But the alternative is no searching light sensor or only one bumper! We really want a searching ability, and only one bumper on a side of the robot would make people laugh, so this compromise is necessary, but not a bad one for the end results.

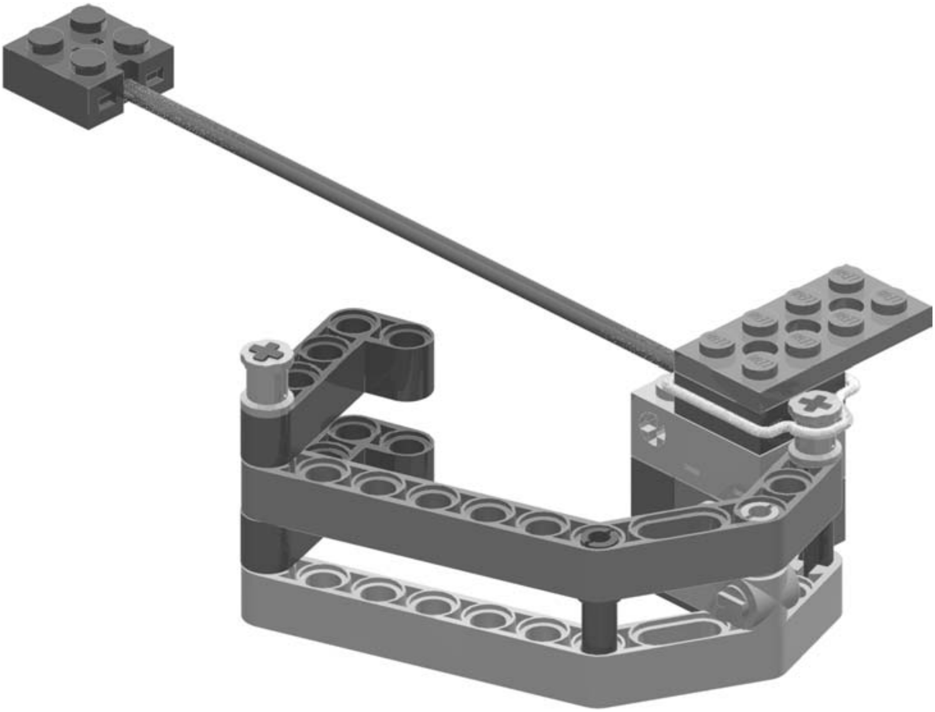
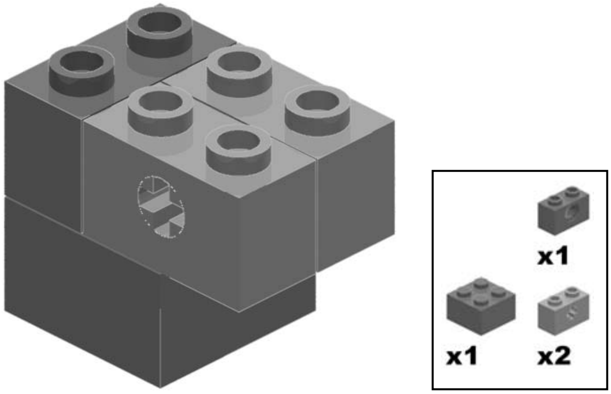


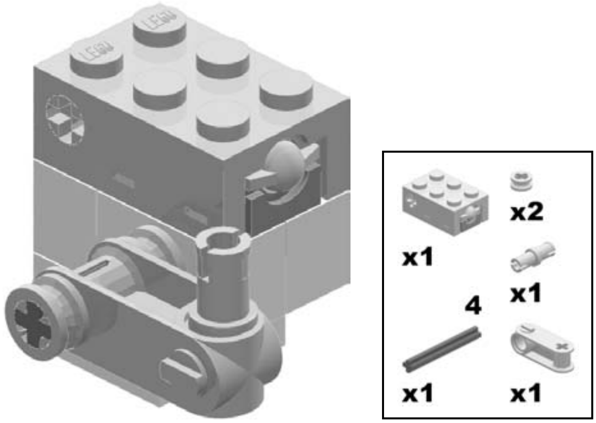
Figure 6. The completed right bumper subassembly

To prepare for building this subassembly, pull out your RIS and ECEP. The ECEP holds several interesting “characteristic pieces” in some nice colors. You’ll be using these to give ZR1 some color flair in not only this subassembly, but also in other parts of ZR1 as well. In step 1, you put several bricks together, including axle bricks from the ECEP.



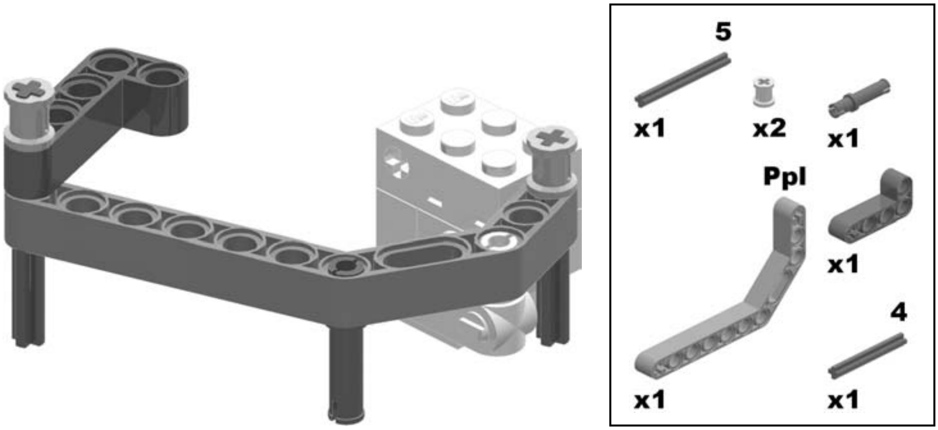
Right Bumper Subassembly Step 1

Step 2 adds a touch sensor, an axle, two half-bushings, and two pieces fundamental to many bumpers: the *long crossblock* and a TECHNIC pin. You'll see just how these pieces are fundamental in a moment.



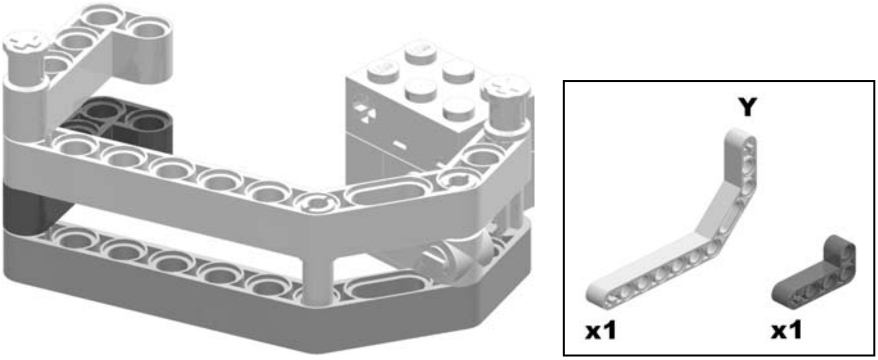
Right Bumper Subassembly Step 2

Construction begins to get a little more interesting in step 3. As you know, the RIS has a number of large, yellow liftarms. But liftarms come in other colors, too—like purple. Rummage around in your ECEP until you spot a purple liftarm that is the same size as the large, yellow ones in the RIS. Snap that onto the pin as shown and position all the other pieces for step 3 onto the liftarm.



Right Bumper Subassembly Step 3

For step 4, find one of the RIS's large, yellow liftarms this time and add that to the assembly, along with one more 2x4 L-shaped liftarm.



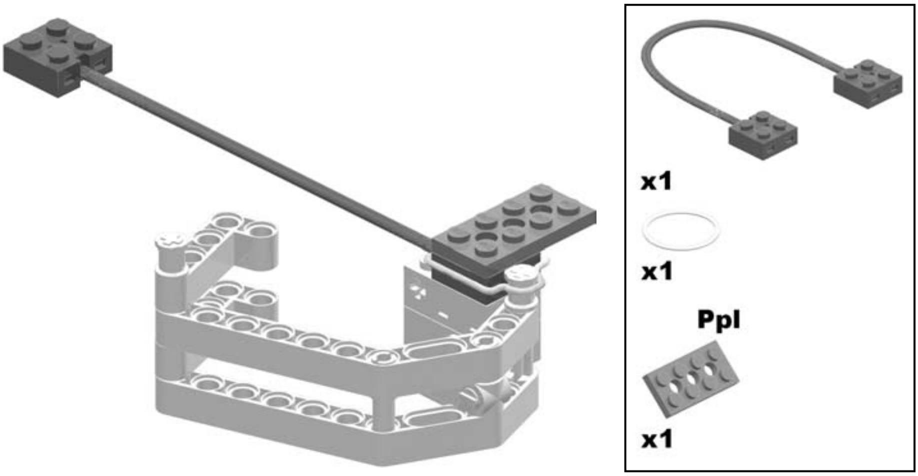
Right Bumper Subassembly Step 4

Step 5 is the final one. One look at that step, and you'll probably think, "Can't this guy just put rubber bands on like everybody else?" Yes, I admit, the rubber band is in a somewhat unusual position. When I make bumpers, I look at every possible solution, even if it looks crazy—because sometimes the solutions that appear crazy are the best ones. And this "crazy solution" was definitely one that worked, so I decided to use it. As you would guess, there is a certain way you must add the rubber band in step 5:

- Before placing the electrical wire onto the touch sensor, slip the rubber band over the 2x2 electrical brick.

- Proceed to place the electrical wire on the touch sensor.

- Finally, once the electrical brick is in place, grab an edge of the rubber band with your fingers and position it over the bushing as shown.



Right Bumper Subassembly Step 5

The Left Bumper Subassembly

As I explained in the previous section, ZR1's left bumper, shown in Figure 7, is the press-type. You would think the two bumpers on ZR1 would be composed of different pieces, since they are different types of bumpers. That's what you would *think*, but that's actually not the case. Rather than having two completely different bumpers on the front, you'll just rearrange the setup a bit, taking away and adding only a handful of pieces.

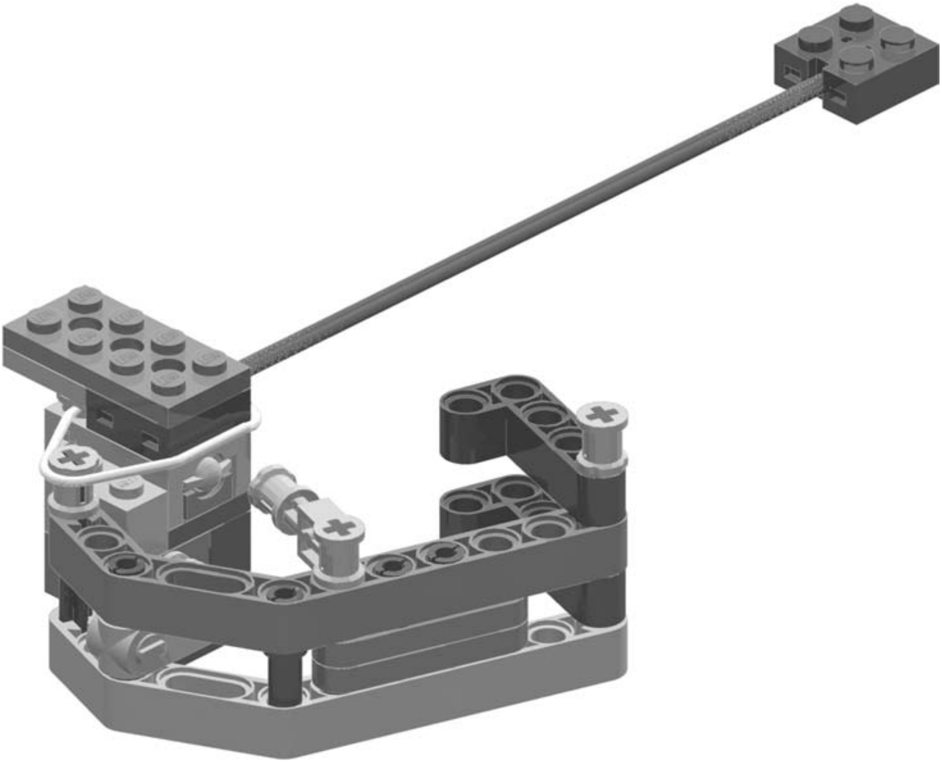
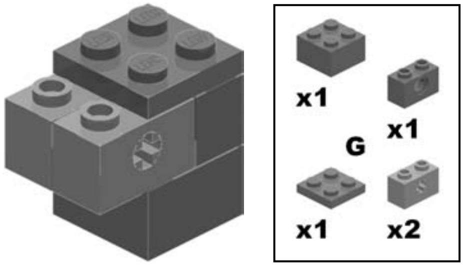
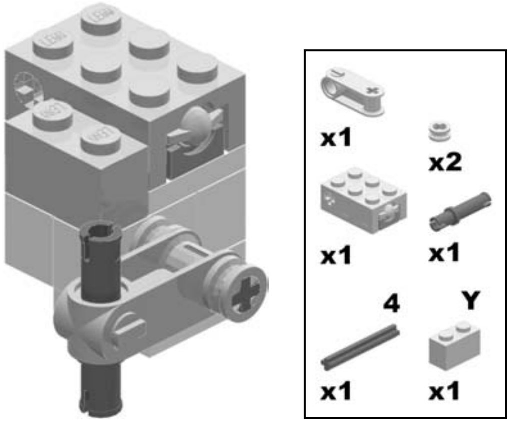


Figure 7. The completed left bumper subassembly

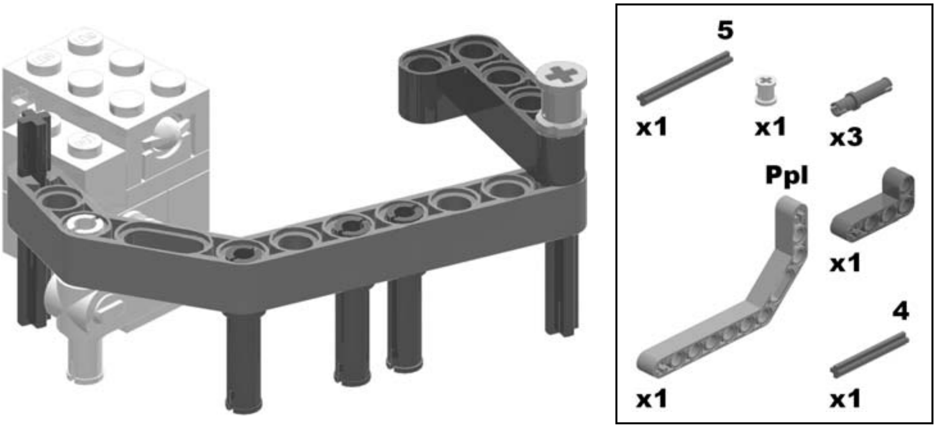
Steps 1 through 3 build up a few bricks, place a touch sensor on them—as with the previous bumper—and also add a long crossblock, purple liftarm, quite a few pins, and more. You'll notice that instead of adding a normal TECHNIC pin to the long crossblock, you are adding a long friction pin, which provides more strength (the right bumper assembly couldn't use this type of pin as it generated too much friction for it to handle when opening and closing the bumper). The 1x2 yellow brick you add in step 2 acts as a sort of a stopper for the purple liftarm, which you add in step 3.



Left Bumper Subassembly Step 1

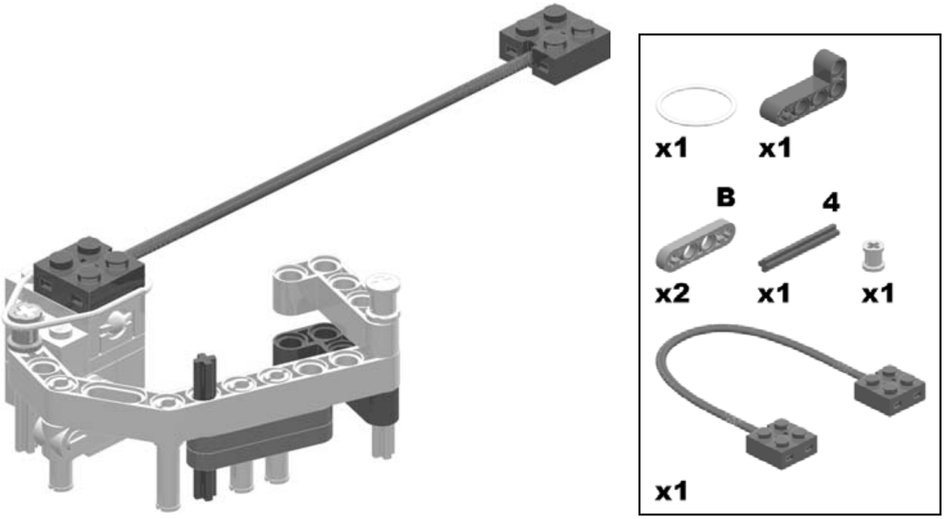


Left Bumper Subassembly Step 2



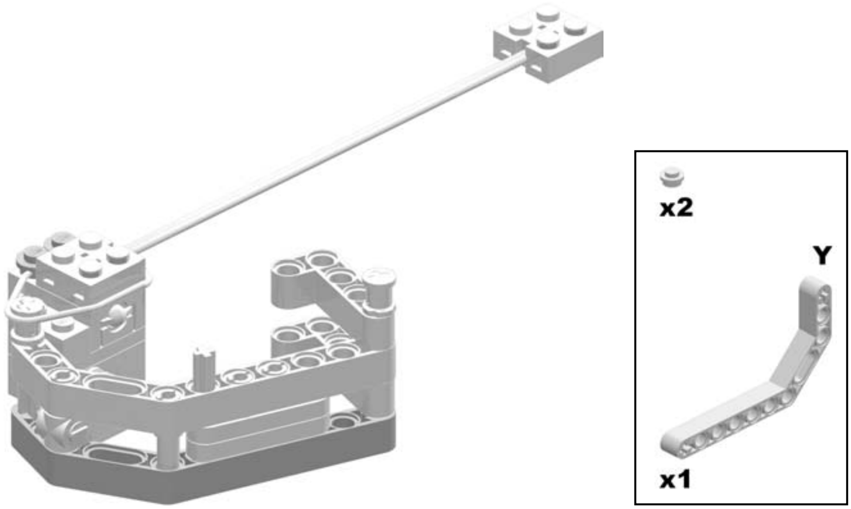
Left Bumper Subassembly Step 3

Remember how the rubber band was positioned for the previous bumper (in step 5)? The positioning of the rubber band in this bumper is about the same. In step 4, add an electrical wire to the touch sensor and a rubber band around that wire and a bushing. Also, add two *1x4 x 0.5 blue TECHNIC liftarms* (found in the UBEP), the other 2x4 L-shaped liftarm, and a #4 axle. As you'll see when you're finished putting together this subassembly, the 1x4 liftarms play an important role in making this a press-type bumper.



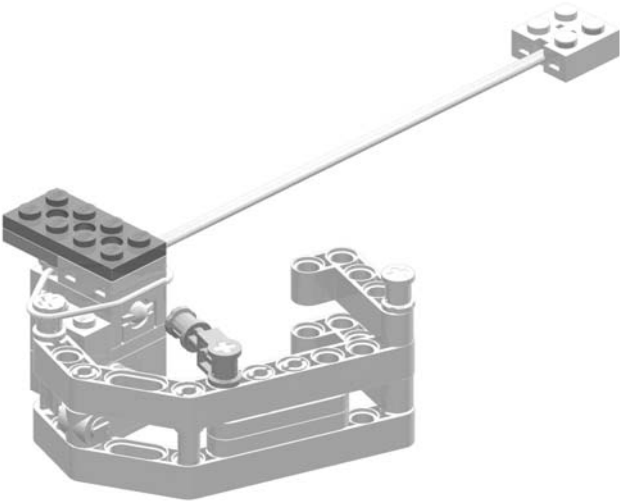
Left Bumper Subassembly Step 4

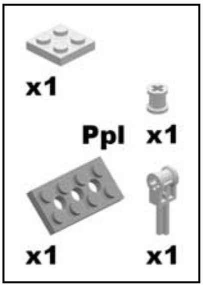
Step 5 adds the bottom, yellow liftarm and two 1x1 round plates. As you can see, the 1x1 plates are "squishing" the rubber band, but this has no adverse effects on the rubber band's performance.



Left Bumper Subassembly Step 5

In step 6, you add the pieces that truly make this a press-type bumper and several more plates that will attach to the chassis.





Left Bumper Subassembly Step 6

With the subassembly finished, the purpose of those 1x4 blue liftarms becomes very clear: they allow an axle to pass through them. The axle is held firmly in place, and the pieces on that axle that press the touch sensor are, in turn, held firmly in place. Therefore, you can depend on the pieces that are responsible for making contact with the touch sensor to stay in position.

The Eye Subassembly

The transparent colors in the eyes give ZR1 an electrified appearance, making this sumo-bot more of a standout. Some of my favorite decoration/characteristic pieces are the transparent ones used in this eye subassembly, which is shown in Figure 8. These transparent pieces can be found in the ECEP.

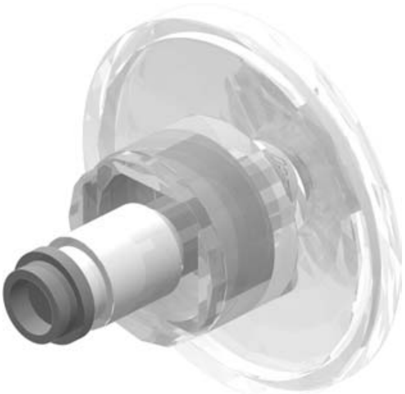
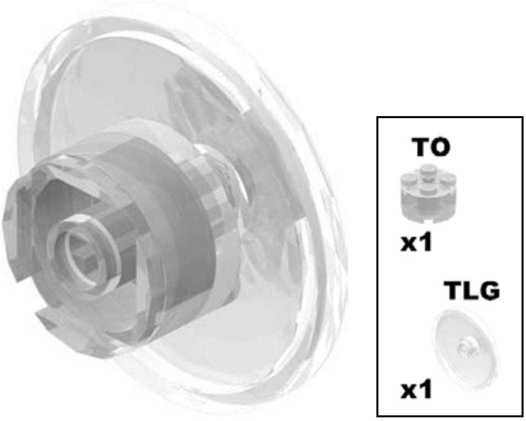
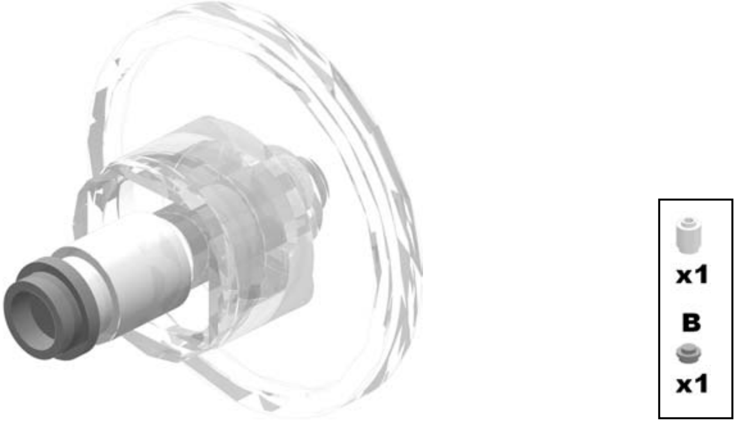


Figure 8. The completed eye subassembly

Steps 1 and 2 construct this subassembly. Remember to build *two* eye subassemblies.



Eye Subassembly Step 1



Eye Subassembly Step 2

The Back Protector Subassembly

All of the sumo-bots in *Competitive MINDSTORMS* had some sort of mechanism or subassembly on the back, and this sumo-bot is no exception. Even if it's small, some protection is always better than no protection. You'll add a sort of protector/slope subassembly to the back end of ZR1, which is shown in Figure 9.

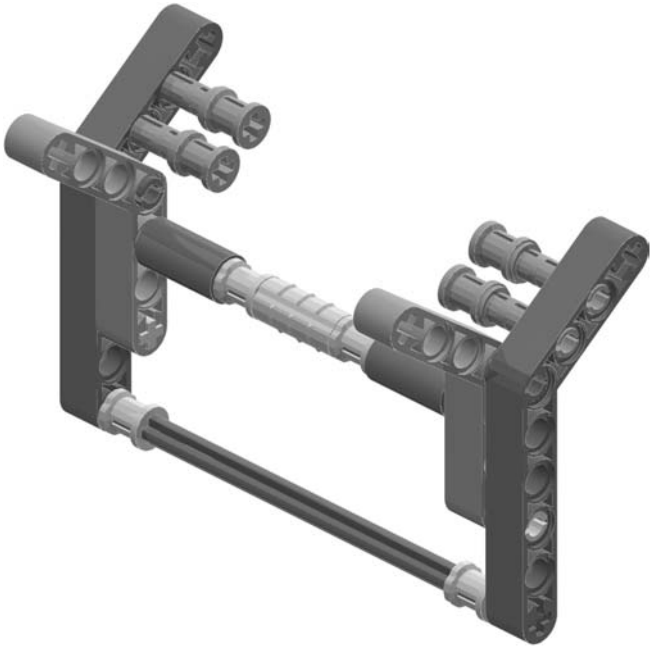
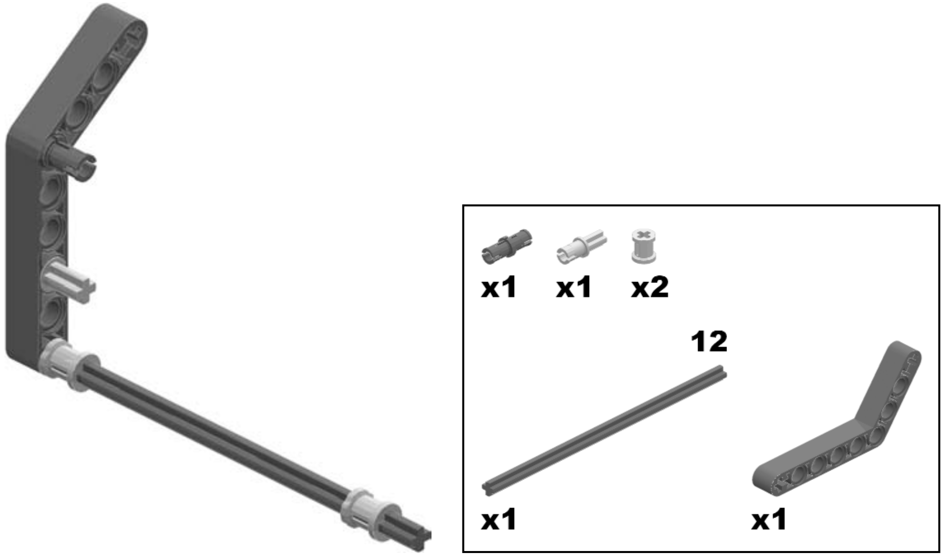


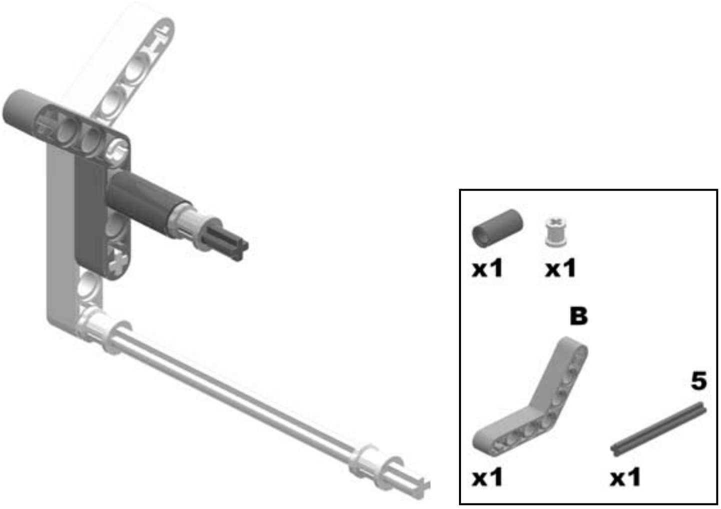
Figure 9. The completed back protector subassembly

Step 1 begins with a #12 axle, two bushings, a black liftarm from the RIS, and a few pins.



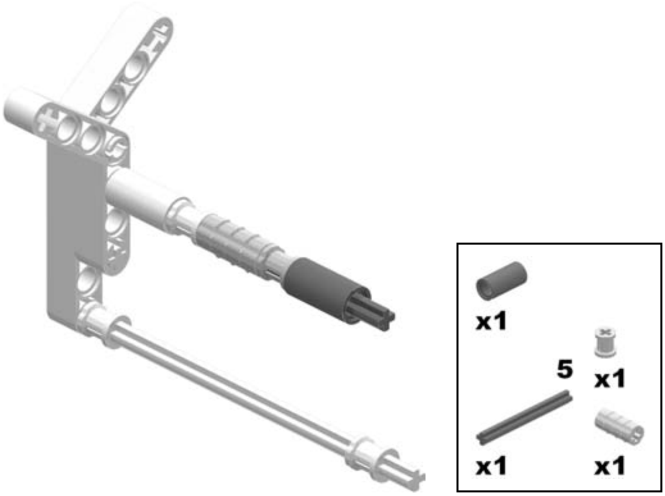
Back Protector Subassembly Step 1

In step 2, take a *1x7 blue bent liftarm* from the UBEP and snap it onto the pins added in the previous step. Also, pull out a #5 axle and push it one stud's space into the bent liftarm, and then slide a TECHNIC pin joiner (found in the UBEP) and a bushing onto the other end of the axle.



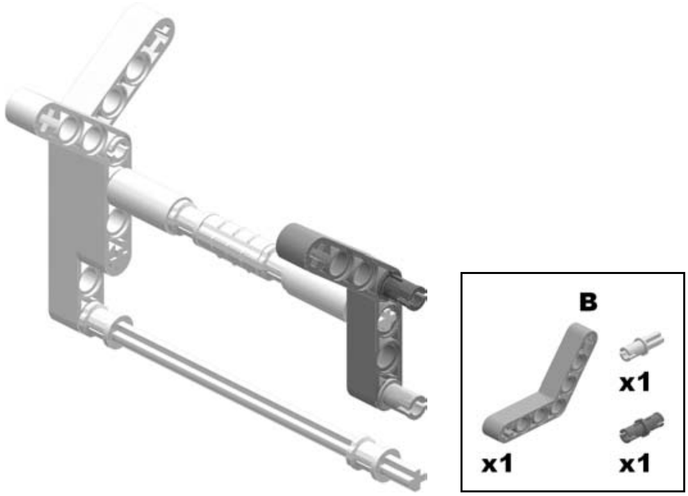
Back Protector Subassembly Step 2

Step 3 takes an axle extender, places that on the remaining part of the #5 axle, and adds another #5 axle into the axle extender. The pieces that were added on the first #5 axle are also added on this second #5 axle.



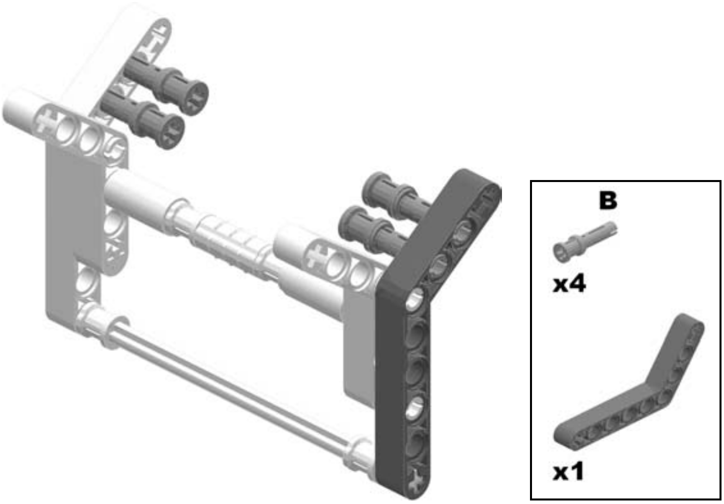
Back Protector Subassembly Step 3

Take another blue, bent liftarm and slide that onto the remaining space of the second #5 axle as shown in step 4. Add the pins shown as well.



Back Protector Subassembly Step 4

In step 5, take one more black liftarm from the RIS and attach it to the #12 axle and pins on the bent liftarm. Add four blue pins with a stop bushing to the black liftarms, and you're finished!



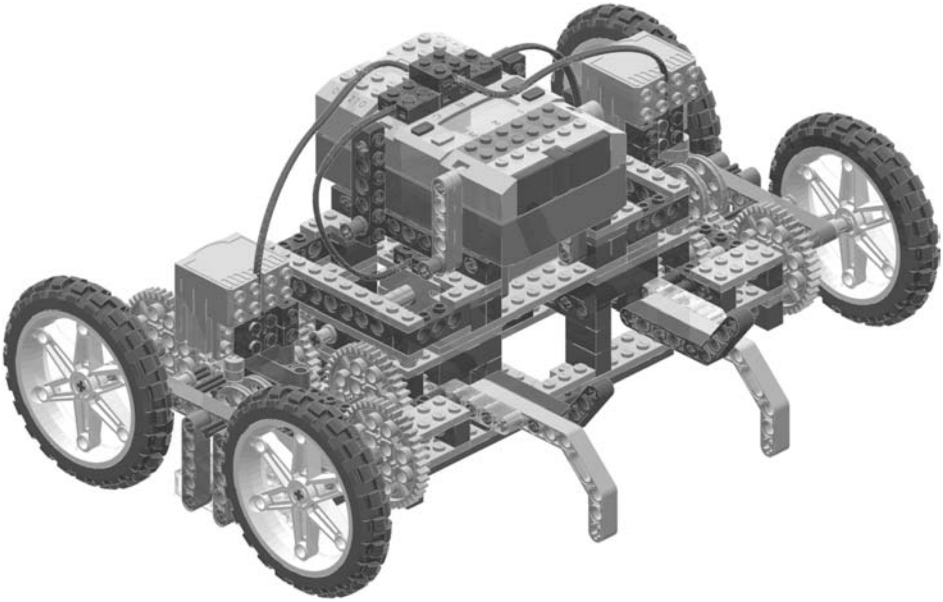
Back Protector Subassembly Step 5

Putting ZR1 Together

You're finally ready for the grand finale! First, bring out the **Brain-Bot chassis**. You don't need to modify the chassis for ZR1.

NOTE *For building instructions for the Brain-Bot chassis, see Chapter 6 of* Competitive MINDSTORMS.

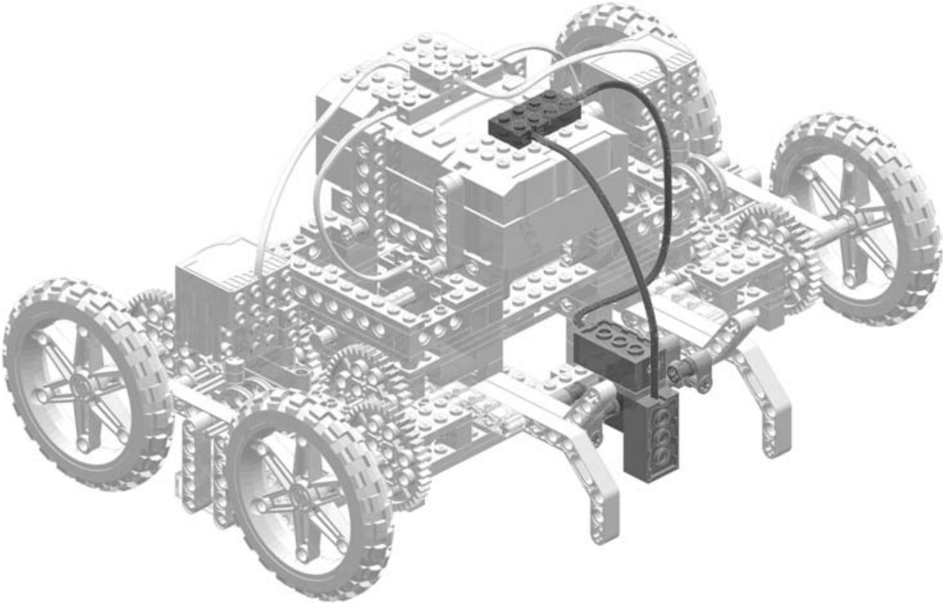
You begin construction in step 1 by pulling out the **right pusher subassembly** and **left pusher subassembly** and attaching them to the chassis as shown.



ZR1 Final Assembly Step 1

Next, attach the **light sensor mount subassembly** in step 2. This can be a little tricky, but it's far from impossible. Position the assembly *exactly* where it should be, and then—one at a time—push the blue pins in. Don't forget to attach the electrical wires to the RCX:

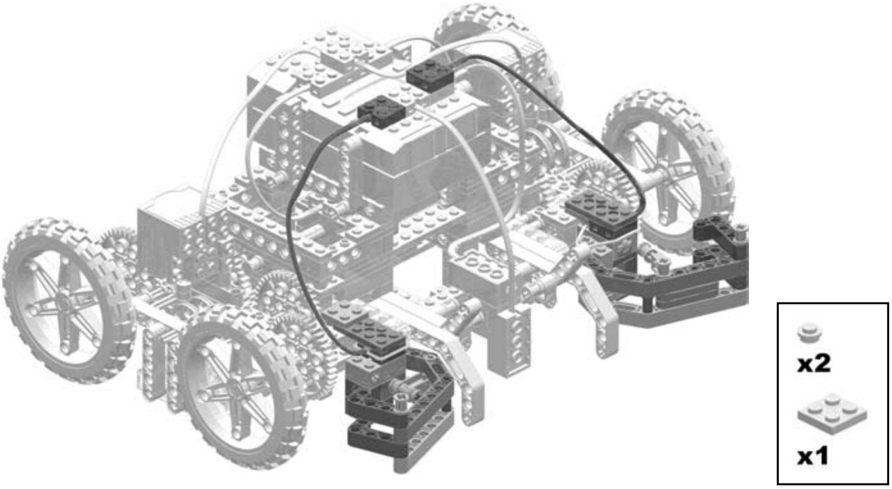
- The **line-detecting light sensor** goes on **input port 2**.
- The **sumo-bot-detecting light sensor** goes on **input port 1**.



ZR1 Final Assembly Step 2

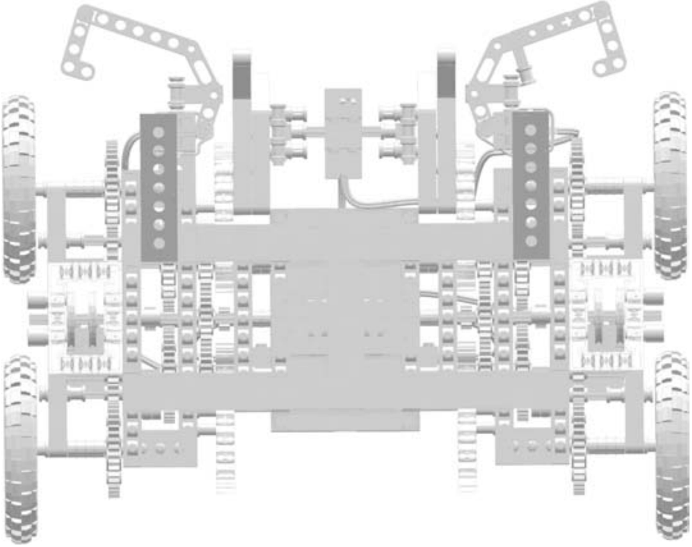
Step 3 adds the **left bumper subassembly** and the **right bumper subassembly**, two 1x1 round plates, and a 2x2 plate. To attach the bumpers, push their plates (which are on top of the touch sensors) onto the 2x4 yellow plates positioned on the chassis. For the right bumper, place the 1x1 plates directly behind the purple 2x4 plate, and then place a 2x2 plate on top of the 1x1 plates and the purple 2x4 plate. Also attach the wires:

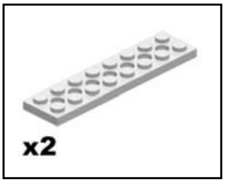
- The **right bumper touch sensor** goes on **input port 3**.
- The **left bumper touch sensor** goes on **input port 1**.



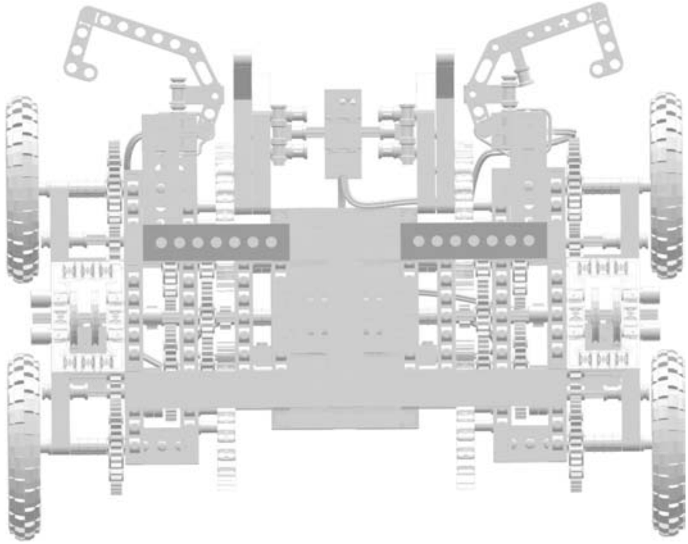
ZR1 Final Assembly Step 3

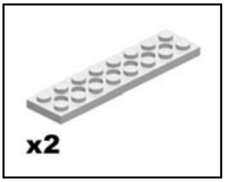
But this is far from sturdy! You need to do much more strengthening for these bumpers. In steps 4 and 5, you flip the sumo-bot over and attach plates to the bottom, which strengthen the bumper's attachment to the chassis. You first attach two plates that run across from the chassis to the bumper subassemblies, and then you add two more plates to strengthen these.





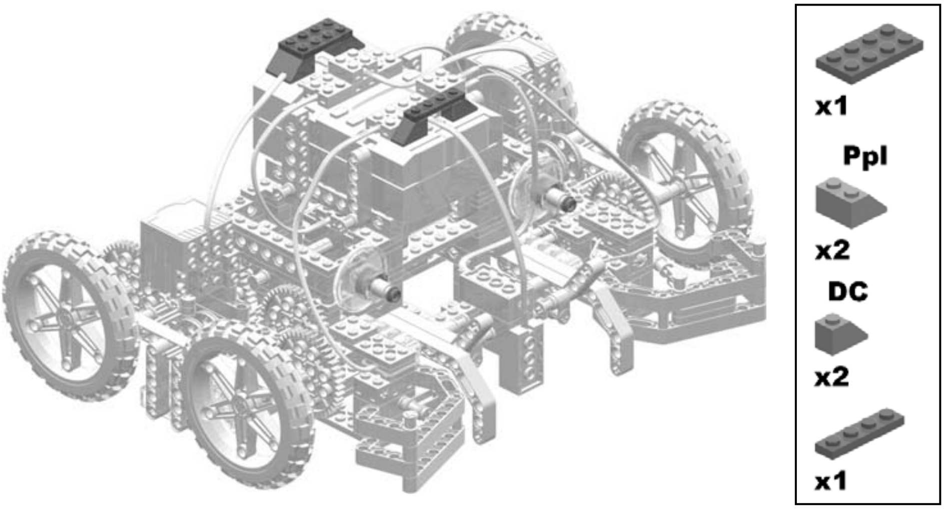
ZR1 Final Assembly Step 4





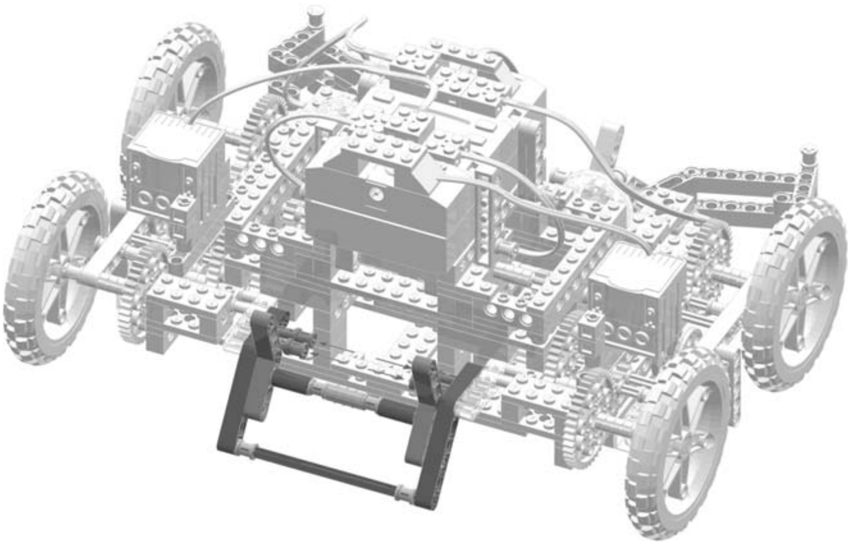
ZR1 Final Assembly Step 5

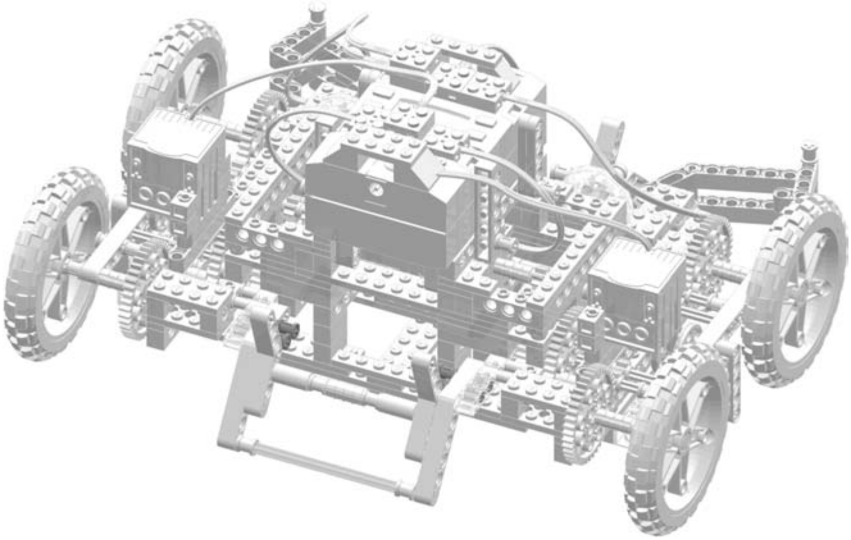
Flip the sumo-bot right side up and add the two **eye subassemblies** as shown in step 6. To attach each eye subassembly, simply push the single stud in the back of it into the middle hole of the 1x4 beam at the front of the chassis. We also add some decorative pieces in this step: attach 1x2 dark-cyan sloped bricks and 2x2 purple sloped bricks to the RCX. Then attach a black 1x4 and 2x4 plate to the sloped bricks. All these pieces come from the ECEP; however, you can easily substitute pieces from the RIS or other sets if you would like.



ZR1 Final Assembly Step 6

Turn the sumo-bot to the back and add the **back protector subassembly** in step 7. To do this, take the blue pins on the subassembly and push them into the chassis after properly positioning them.





ZR1 Final Assembly Step 7

You have just completed Brain-Bot Version ZR1 (see Figure 1, shown at the beginning of this chapter). On to the programming now!

Formulating a Solid Programming Solution for ZR1

ZR1's program will make use of NQC's access control feature and also speed theory (discussed in Chapter 7 of *Competitive MINDSTORMS*). To get a good grasp of the entire program, you should observe all the tasks we will be using. Here they are, in order of their priority:

- **Line-detection task:** Like ZR2's line-detection task (presented in Chapter 7 of *Competitive MINDSTORMS*), this one is somewhat complex. No boring, simple line-detection task here!

- **Sumo-bot searching task:** Here is the interesting task—interesting because of *what* it does and *how* it does it. You'll learn the *how* a little later, but here is a quick summary of *what* it does: once the light sensor has detected the other sumo-bot, slow speed will be engaged, and ZR1 will go forward until it loses sight of the opponent. After losing sight of the opponent, fast speed will be resumed.

- **Dual-bumper task:** If both bumpers are pressed at the same time, this task will take over and keep ZR1 moving forward until both bumpers have been deactivated.

- **Right bumper task:** This task watches the right bumper. If the right bumper is pressed, the program tells ZR1 to go to the right just a little.

- **Left bumper task:** This task watches the left bumper. If the left bumper is pressed, the program tells ZR1 to go to the left just a little.

- **Maneuvering task:** Also like ZR2's maneuvering task, this maneuvering task actually does maneuver the robot. In an infinite loop, ZR1 is told to first go straight for a designated amount of time, execute a sweeping turn, and then do it all over again.

- **NQC header:** We're going to use an NQC header in ZR1's program (as in the programs presented in Chapters 6 through 9 of *Competitive MINDSTORMS*).

As you can see, this is going to be a relatively large program. With six tasks and a header, we've got a lot of programming to do, so let's get started!

Programming ZR1 with the Solution

The last item on the summary of tasks list is the NQC header, but we're going to be looking at it *first*. It holds vital functions that will be used throughout the main program. Once we've finished with the header, we'll work on the main program.

Creating the NQC Header

Take a look at the NQC header for this program in Listing 1.

Listing 1. Brain-Bot-ZR1.nqh

```
/* Brain-Bot-ZR1.nqh - A file holding important
* instructions for Brain-Bot version ZR1.
* To be included in ZR1's main program */

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

// constants
#define CHANGE 35
#define CALIBRATION 50
#define SEARCH  100
#define TURN_SMALL 110
#define SPIN 400
#define GO 500
#define TURN 575
#define AMOUNT 5

// sensors
#define See SENSOR_2
#define LBump SENSOR_1
#define RBump SENSOR_3

// go forward
void Forward()
{
   OnFwd(Left+Right);
}

// go in reverse
void Reverse()
{
   OnRev(Left+Right);
}

// switch gears to fast speed
void SwitchF()
{
   OnRev(Switch);
   Wait(CHANGE);
   Off(Switch);
}
```

```
// switch gears to slow speed
void SwitchS()
{
   OnFwd(Switch);
   Wait(CHANGE);
   Off(Switch);
}

/* special function for switching gears
*  to slow speed if robot spinning in place*/
void SwitchSTurn()
{
   OnFwd(Left+Right);
   Wait(CHANGE);
   OnFwd(Switch);
   Wait(CHANGE);
   Off(Switch);
}

// turn right
void TurnR()
{
   Fwd(Left);
   Rev(Right);
   Wait(TURN);
   Fwd(Left+Right);
}

// turn left
void TurnL()
{
   Fwd(Right);
   Rev(Left);
   Wait(TURN);
   Fwd(Left+Right);
}

// turn right a little
void TurnRL()
{
   Rev(Right);
   Wait(TURN_SMALL);
   Fwd(Left+Right);
}

// turn left a little
void TurnLL()
{
   Rev(Left);
   Wait(TURN_SMALL);
   Fwd(Left+Right);
}

// spin in place
void HalfSpin()
{
   Rev(Left);
   Fwd(Right);
   Wait(SPIN);
}
```

Within this header are extremely simple functions (one line of code), medium-sized functions, and even one function that is five lines of code long. They do everything from making the sumo-bot go forward, go in reverse, turn left, turn to the right just a little, and switch speeds. We also have one function, `SwitchSTurn()`, that switches to slow speed under a certain circumstance.

> **NOTE** *All the following code makes use of the constants presented in the header. Instead of adding all the definitions of the constants again in each code example, you can go back to the header here to determine each constant's value.*

Now we can work on the main program. What is the very first step? We need to program the calibration functions.

Programming the Calibration Functions

For ZR1, we need to create not one, but *two* calibration functions: one for the line-detecting sensor and one for the searching light sensor. Here is the calibration function for ZR1's line-detecting light sensor:

```
// variables
int threshold, line;

/* This function calibrates the
*  "line searching" light sensor */
void Calibrate()
{
    until(RBump==0);
    line=See;
    threshold=line+AMOUNT;
    until(RBump==1);
    PlaySound(SOUND_CLICK);
}
```

The function waits until the right bumper—the release-type bumper—is “open” (activated or pressed), takes a reading, and then waits until it is “closed” (back in place with the touch sensor reading 1, meaning it has been released). Since we’re not finished with all of the calibrations, this function doesn’t have a slight waiting time at the end; the other calibration function will possess that code.

Here is the other function, to calibrate a light sensor that searches for sumo-bots:

```
//variable
int thresh;

/* This function calibrates the
*  "sumo-bot searching" light sensor */
void Calibrate2()
{
    until(RBump==0);
    thresh=LBump;
    until(RBump==1);
    PlaySound(SOUND_CLICK);
    until(RBump==0);
    until(RBump==1);
    PlaySound(SOUND_CLICK);
    Wait(CALIBRATION);
}
```

As with the previous function, we wait until the release-type bumper is open, and then we take a reading and wait until the bumper is closed. We have a new variable here: `thresh`. Think of it as the `threshold` variable’s little brother. The `thresh` variable is obviously being stored with the value of the searching light sensor, but what’s going on here?

Let’s back up a bit. For the line-detecting light sensor, something physical was involved in the calibration, and that physical action was placing the light sensor over the line. For this calibration—the searching light sensor’s calibration—physical action is involved as well. That physical action is putting something in front of the searching light sensor: your hand! It might come as a surprise, but your hand is a very good emulator of a sumo-bot when it comes to calibrating the light sensor. You simply put your hand in front of the searching light sensor, press and release the right bumper, and your searching light sensor has been calibrated! (I’ll explain in more detail how to properly calibrate the searching light sensor in the “Testing ZR1” section later in this chapter.)

The small amount of waiting time I mentioned earlier—which allows you to back off before the sumo-bot starts up—is at the end of this function. After waiting for the bumper to be pressed and released one more time, the function waits for `CALIBRATION` (0.50 second), and then gets the main program underway.

Finally, you’ll notice all the little “clicks” in these two calibrating functions. As in programs presented in *Competitive MINDSTORMS*, the clicks let you know that portions of code have completed successfully: calibration has been completed or the bumper has been pressed and released.

TIP *Instead of guessing the numbers for the searching light sensor's threshold or figuring it out every time I start up the robot in a room with different lighting, I use calibration. When attending a robotic sumo event, leave nothing to chance. Using calibration—even more than one type as you can see—is* the *way to work with light sensors. Not only that, it is considered good practice.*

Programming the Line-Detection Task

Now that we've completed the calibration, we'll work on the task that watches for the line encircling the arena; this task makes use of the speed theory. Take a look:

```
// variable
int speed = 0;

// Task that 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)  // get control
         {
           if(speed == 1)   // if in slow speed..
           {
             SwitchF();   // switch to fast
             speed = 0;   // tell program we're on fast speed
             Reverse();   // avoid the line!!!
             Wait(SEARCH);
             TurnR();
             BL = 0;
             BR = 0;
             PlaySound(SOUND_UP);   // end of this task...
           }

           else    // this is if we're already on fast speed
           {
             Reverse();       // avoid the line!
             Wait(SEARCH);
             TurnR();
             BL = 0;
             BR = 0;
             PlaySound(SOUND_UP);  // end of this task...
           }
         }
   }
}
```

This task has the highest priority (0). There are two basic sections to this task once the line has been detected; that is, there are two basic sections in all the code after the acquire statement. These sections are outlined with the if statement and the else statement. Take a look at the code for the if statement:

```
if(speed == 1)   // if in slow speed..
{
  SwitchF();
  speed = 0;
  Reverse();
  Wait(SEARCH);
  TurnR();
  BL = 0;
  BR = 0;
  PlaySound(SOUND_UP);
}
```

If the sumo-bot is on slow speed, which it can tell by the speed variable, the code that follows the if statement tells ZR1 to immediately switch to fast speed, update speed, reverse a little, and turn right (the function for turning right also makes ZR1 go forward again). But what are BL and BR? These are variables that have almost nothing to do with the line-detecting function, but have a lot to do with the tasks for the bumpers. We'll look at these variables when we talk about the bumper tasks' code in the "Programming the Bumper Tasks" section later in this chapter.

The code within the else statement is nearly identical to the code within the if statement, the exception being the first two lines of code. The first two lines of code that were in the if statement are not necessary here; the else part of the code is for when ZR1 is already on fast speed, so ZR1 doesn't need to switch to fast speed.

Programming the Searching Task

The amazing searching task is next. This task is given second highest priority (1). We don't give it an equal priority with the line-detecting task because line detecting is always more important. Here is the searching task:

```
/* This task searchs for the other
*  sumo-bot with a second light sensor*/
task Search()
{
   // second highest priority
   SetPriority(1);

   while(true)  // infinite loop - always check for sumo-bots
   {
      until(LBump<=thresh); // until sensor detects sumo-bot

      acquire(ACQUIRE_USER_1)  // get control
      {
         Wait(CALIBRATION); // wait a little

         /* if Input Port 1 is less than "thresh" or
         * equal to 100*/
         if(LBump<=thresh | LBump==100)
         {
          BL = 1;
          BR = 1;
          SwitchSTurn(); // if true, go on slow speed and...
          speed = 1;
          Forward();   // ...attack!
          until(LBump==thresh+5); // until sumo-bot goes away
          SwitchF();  // stop attacking
          speed = 0;
          BL = 0;
          BR = 0;
          PlaySound(SOUND_UP);
         }
      }
   }
}
```

The code that triggers this task is `until(LBump<=thresh);`. Why then, you might ask, is this repeated—in a slightly different form—after control has been acquired? We already know that `LBump<=thresh`, so why ask it again? Let's look at this interesting line of code:

```
if(LBump<=thresh | LBump==100)
```

This line of code is asking *two* questions. The second question is `| LBump==100`. This means "or if LBump is equal to 100." This is necessary because there is a little problem associated with the method we are using to put two sensors on one port.

When you attach a light and touch sensor to the same input port, the light sensor won't always be able to take a reading. If the touch sensor, which is on that same port, gets pressed, the value equals 100. If this happens, that sensor port won't be `<=thresh` (asked at the beginning of the searching task), and authority will transfer to the bumper, even though the searching light sensor has the enemy in sight! Although putting a light sensor and touch sensor on the same port is an awesome trick, this little problem above goes along with it. So what do we do? We look very hard for a solution. And the solution is the line of code we are talking about here.

Once the searching light sensor has sighted the opponent, it asks another question with this line of code: if the light sensor is less than the threshold, *or* if the same input port equals 100, attack the enemy. Now if the input port equals 100 because the left bumper has been pressed, ZR1 keeps on going straight, which is exactly what we want.

Looking at the searching task again you'll see those `BL` and `BR` variables once more. This time they do have something to do with this task, but I'll put off explaining until we discuss the bumper tasks (in the "Programming the Bumper Tasks" section, coming up soon).

Continuing in the searching task, you'll find another interesting statement:

```
until(LBump==thresh+5);
```

This is for exiting the searching task. If input port 1 is equal to `thresh + 5`, exit the task. But why do this? Well, we have ourselves another problem at this point (which this line of code solves). If we simply tell the task to wait until `LBump` is more than `thresh`, what happens if the left bumper gets pressed? Then `LBump` *really is* more than `thresh`—100 to be precise—and the program exits the task, but at the wrong time, since ZR1 hasn't necessarily lost sight of the opponent. We get around this by simply using a limitation: if `LBump` is equal to `thresh` plus 5, exit the task.

When ZR1 finally loses sight of the opponent (maybe the other sumo-bot fell off the side of the arena?), the searching light sensor's value will soar past `thresh` plus 5. However, it *will* pass, and that is all the program needs. Although we can't see the value `thresh + 5` because it happens so fast, the program sees the value of `thresh + 5`, even if just for a fraction of a second. So when the opponent goes out of sight, `LBump` eventually reaches (and then passes) `thresh + 5`, the program realizes that the specified value has been met, and exits the task. On the other hand, if the left bumper gets pressed, the searching light sensor's value doesn't soar past `thresh + 5` all the way to 100; instead, it immediately becomes 100. So using the line `until(LBump==thresh+5);`, we can prevent incorrectly exiting the searching task when the left bumper is pressed.

Programming the Dual-Bumper Task

Next is the task that takes over when both bumpers are activated. You might be wondering just how the searching light sensor could *not* see the opponent if *both* bumpers are pressed. Believe it or not, this can happen: both bumpers may be pressed, and yet ZR1 has not yet detected the opponent with its searching light sensor (this often depends on the geometry of the opponent's structure). If this happens, ZR1 will use the dual-bumper task to switch to slow speed and "attack" the opponent. We also use this task to avoid erratic behavior. If there were no dual-bumper task, and if both bumpers were pressed, ZR1 would go in reverse.

Take a look at task `BPressed()`:

```
/* Task Both Pressed - for when both
* bumpers are being pressed */
task BPressed()
{
  SetPriority(2);  // third highest prioity

  while(true) // infinite loop - always check for this situation
  {
      // until both bumpers are pressed
      until(LBump==100&RBump==0);

      acquire(ACQUIRE_USER_1) // get control
      {
              SwitchS();    // get on slow speed
              speed = 1;
              Forward();    // and attack!
              until(LBump<100&RBump==1); // until we don't detect anymore
              SwitchF();  // get back on fast speed
              speed = 0;
              PlaySound(SOUND_UP);
      }
  }
}
```

NOTE *Notice the ampersand (&) in this task. This is the* AND *operator in NQC. Here is another example of how it can be used:* `if(4+3==7&3+4==7)`*. There are two types of* AND*:* & *and* &&*. The former, the one we use in this program, is called* bitwise AND *and has no restraints on the types of arguments that can be used with it. The latter example is called logical* AND *and is constrained to constants only (such as 5, 27, OUT_A). So, the example* `if(4+3==7&3+4==7)` *should actually be* `if(4+3==7&&3+4==7)`*, because it uses constants.*

This is actually pretty simple. If both bumpers are activated, ZR1 switches to slow speed and stays that way until both bumpers are released. "Wait a second! What if the searching task kicks in and ZR1 is in slow speed? That means it will switch to slow speed although it's already on slow speed!" That's entirely correct. But it's also not a big problem, for the following reasons:

- This situation—both bumpers getting activated before the searching light sensor sees the opponent—is not common.
- If the left bumper remains pressed, the searching task can't take over since the input port reads 100.
- If ZR1 has such a good grip on the opponent, the left bumper may not get deactivated until it has run the opponent off the arena; when this happens ZR1 will detect the line and can properly switch back to fast speed without any harm.
- There is more to be gained in this situation by switching to slow speed than not to switch to slow speed.
- An occasional switch to a speed the sumo-bot is already on won't harm the motors.

When I tried to add more code to the searching task to determine what speed the sumo-bot was in, that only caused problems. Of course, feel free to experiment to see if you can think of another way to handle this task.

Programming the Bumper Tasks

Next, we have the bumper tasks. We'll be examining just one of them (the left bumper task) since they are very similar:

```
task BLWatch()
{
   // fourth highest priority
   SetPriority(3);

   while(true)   // infinite loop - continually check sensor
   {
      // until LBump is released and "BL" = 0
      until(LBump==100&BL==0);
         acquire(ACQUIRE_USER_1)  // get control
         {
              TurnLL();         // and turn little left
              Forward();        // go forwards
              until(LBump<100); // until not pressed
              PlaySound(SOUND_UP);
      }
   }
}
```

The basic meaning of the task is simple: if pressed, turn a little in that direction and go forwards until the bumper isn't pressed any more.

Notice that we have one of those mysterious variables here: BL. As promised, I will now explain the purpose of the BL (for Bumper Left) and BR (for Bumper Right) variables.

When the searching task is running, and one of the bumpers is activated, that bumper will fight for control—and get it. So when ZR1 should be going forward, it turns because its bumper got pressed; and that bumper has a lower priority than the searching task! The special sensor method we're using appears to be part of the reason for this occurrence. After a couple of seconds have passed with the left bumper pressed, the program gets confused because of input port 1's constant 100 value and hands control to the bumper task. We need to combat this, which is the purpose of the BL and BR variables. This is the line of code that allows this bumper task to take control:

```
until(LBump==100&BL==0);
```

If the bumper is pressed *and* BL equals 0, the task can take control. Look back to the searching task. When that task takes control, it makes BL and BR equal 1. Now, even though the bumpers might be activated, their tasks can't take control because these variables *must* equal 0. We have beaten those troublemaking bumper tasks!

Now look back to the line-detecting task; you should notice that it sets the BL and BR variables back to 0. If ZR1 ran over the line while running the searching task, these variables need to be set back to 0 to let the bumper tasks take control if necessary.

Programming the Maneuvering Task

The final task to discuss is the maneuvering task. This one simply lets the sumo-bot go forward for a predefined amount of time, and then make a little turn; after that, do it all over again.

```
// Our maneuvering task
task Maneuver()
{
   // fifth highest priority
   SetPriority(4);

   while(true)   // infinite loop - continually check
   {
         acquire(ACQUIRE_USER_1) // get control
         {
           // another infinite loop - always do this
           while(true)
           {
           OnFwd(Left+Right);
           Wait(GO);
           HalfSpin();
           }
        }
   }
}
```

Putting the Program Together

We're finally ready to look at the program in its entirety, which is almost completely composed of the tasks we just observed. ZR1_Access_Control.nqc is shown in Listing 2.

Listing 2. ZR1_Access_Control.nqc

```
// ZR1_Access_Control.nqc
// A sumo-bot program for Brain-Bot Version ZR1

#include "Brain-Bot-ZR1.nqh"   // include our header

// set some variables
int line,threshold,thresh,BL = 0,BR = 0,speed = 0;

task main()
{
  SetSensor(See,SENSOR_LIGHT);     // let's not forget to
  SetSensor(LBump,SENSOR_LIGHT);   // initialize our
  SetSensor(RBump,SENSOR_TOUCH);   // sensors!

  SetPower(Switch,3);  // putting OUT_B at the correct power level
  Off(Switch);  // make sure these motors are in brake mode

  Calibrate();    // calibrating light sensor...

  Calibrate2();   // calibrating second light sensor...

  OnFwd(Left+Right);    // get moving forward...

  start LWatch;   // turn on all six tasks!
  start Search;
  start BPressed;
  start BRWatch;
  start BLWatch;
  start Maneuver;
}

// Task that 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)  // get control
         {
           if(speed == 1)   // if in slow speed..
           {
             SwitchF();   // switch to fast
             speed = 0;   // tell program we're on fast speed
             Reverse();   // avoid the line!!!
             Wait(SEARCH);
             TurnR();
             BL = 0;
             BR = 0;
             PlaySound(SOUND_UP);
           }

           else    // this is if we're already on fast speed
           {
             Reverse();       // avoid the line!
             Wait(SEARCH);
             TurnR();
             BL = 0;
             BR = 0;
             PlaySound(SOUND_UP);  // end of this task...
           }
         }
   }
}
```

```
/* This task searches for the other
*  sumo-bot with a second light sensor*/
task Search()
{
   // second highest priority
   SetPriority(1);

   while(true)  // infinite loop - always check for sumo-bots
   {
      until(LBump<=thresh); // until sensor detects sumo-bot

      acquire(ACQUIRE_USER_1)  // get control
      {
         Wait(CALIBRATION); // wait a little

         /* if Input Port 1 is less than "thresh" or
         * equal to 100*/
         if(LBump<=thresh | LBump ==100)
         {
          BL = 1;
          BR = 1;
          SwitchSTurn(); // if true, go on slow speed and...
          speed = 1;
          Forward();   // ...attack!
          until(LBump>thresh&LBump<100); // until sumo-bot goes away
          SwitchF();  // stop attacking
          speed = 0;
          BL = 0;
          BR = 0;
          PlaySound(SOUND_UP);
         }
      }
   }
}

/* Task Both Pressed - for when both
* bumpers are being pressed */
task BPressed()
{
  SetPriority(2);  // third highest prioity

  while(true) // infinite loop - always check for this situation
  {
      // until both bumpers are pressed
      until(LBump==100&RBump==0);

      acquire(ACQUIRE_USER_1) // get control
      {
              SwitchS();    // get on slow speed
              speed = 1;
              Forward();    // and attack!
              until(LBump<100&RBump==1); // until we don't detect anymore
              SwitchF();  // get back on fast speed
              speed = 0;
              PlaySound(SOUND_UP);
      }
  }
}

// Task Bump Left Watch
task BLWatch()
{
   // fourth highest priority
   SetPriority(3);

   while(true)   // infinite loop - continually check sensor
   {
      // until LBump is released and "BL" = 0
      until(LBump==100&BL==0);
         acquire(ACQUIRE_USER_1)  // get control
         {
              TurnLL();         // and turn little left
              Forward();        // go forwards
              until(LBump<100); // until not pressed
              PlaySound(SOUND_UP);
      }
   }
}

// Task Bump Right Watch
task BRWatch()
{
   // fourth highest priority
   SetPriority(3);

   while(true)   // infinite loop - continually check sensor
   {
      until(RBump==0&BR==0); // until RBump is released

         acquire(ACQUIRE_USER_1)  // get control
         {
              TurnRL();        // turn right a little
              Forward();       // go forwards
              until(RBump==1); // until not released
              PlaySound(SOUND_UP);
       }
   }
}

// Our maneuvering task
task Maneuver()
{
   // fifth highest priority
   SetPriority(4);

   while(true)   // infinite loop - continually check
   {
         acquire(ACQUIRE_USER_1) // get control
         {
           // another infinite loop - always do this
           while(true)
           {
           OnFwd(Left+Right);
           Wait(GO);
           HalfSpin();
           }
        }
   }
}

/* This function calibrates the
*  "line searching" light sensor */
void Calibrate()
{
    until(RBump==0);
    line=See;
    threshold=line+AMOUNT;
    until(RBump==1);
    PlaySound(SOUND_CLICK);
}

/* This function calibrates the
*  "sumo-bot searching" light sensor */
void Calibrate2()
{
    until(RBump==0);
    thresh=LBump;
    until(RBump==1);
    PlaySound(SOUND_CLICK);
    until(RBump==0);
    until(RBump==1);
    PlaySound(SOUND_CLICK);
    Wait(CALIBRATION);
}
```

Feel free to modify this program and experiment with other programming techniques. For example, you could use timers (which are absent in this program), other types of programming structures, or different responses to sensor readings. You can use this program as a base on which to build, or you can start from scratch and create a whole new program for ZR1. The possibilities are endless!

Testing ZR1

After all that programming, you are probably more than ready to try out ZR1. Follow these steps to get started:

1. Download ZR1_Access_Control.nqc to slot 1 on your RCX.

NOTE *The entire arena should be well lit when using ZR1. Although this should be the case anyway, for ZR1, it is especially important. If one section of the arena is darker than another section, ZR1 might actually think it is seeing the opponent and switch to slow speed when it gets to that area. The* entire *arena must be uniformly and brightly lit.*

2. Put ZR1 in your arena and position its line-detecting light sensor on the line. Press the Run button on the RCX; you are now in the first calibration phase.

3. Find ZR1's right bumper—the release-type one—press it, and then let the rubber bands draw it back—you should hear a click. The line-detecting sensor has been properly calibrated.

4. Now you need to calibrate the second light sensor. This one is a little trickier. For this calibration, you use your hand to emulate the other sumo-bot. Depending on the exact position of your hand, the light sensor will either be extremely sensitive or barely sensitive. The best place to position your hand is about 2 inches away from the light sensor. Do this, press and release the right bumper (with your other hand!), and you should hear a click. The second light sensor has now been calibrated.

TIP *It is a great deal of fun to try out different levels of sensitivity for the searching light sensor. Hold out your hand farther when calibrating, and ZR1 will be very sensitive. Put your hand very close when calibrating, and ZR1 is not nearly as sensitive. Have fun with this feature!*

5. Place ZR1 in the middle of the arena and press the right bumper one more time. With one last click, ZR1 should shoot forward.

Once you've gotten a little experience, it shouldn't take more than a few seconds to calibrate both light sensors and start the robot.

After ZR1 has started moving, observe its response to the line. It should back up only a small distance, but make a very large turn. If it made only small turns upon seeing the line, it would keep running into the line.

Next, observe the maneuvering task. After five seconds of inactivity, ZR1 should start spinning in place; it will spin for at least four seconds. After that, it will begin the process all over again. *But*, if ZR1 sees something with its searching light sensor while spinning, it will stop spinning and instead go forward, and it will switch to slow speed (attack mode). That is the purpose of spinning in the maneuvering task; although I call it *spinning*, it's more like scanning! ZR1 scans its surroundings with its searching light sensor to see if anyone is nearby.

Let's start something a little more interactive. While ZR1 is happily going along, press one of its bumpers. The sumo-bot should turn in the direction of the bumper you pressed. Now press both bumpers at the same time; ZR1 should switch to slow speed and go forward. Deactivate one of the bumpers, reactivate it after a second or two, and then deactivate the other one. ZR1 should still be going forward. Only after *both* of the bumpers are deactivated does ZR1 switch back to fast speed and exit the `BPressed()` task.

Next, try out that searching light sensor. Take a relatively dark object that ZR1 can push—possibly another sumo-bot (if you have one), canned food, or a tall cup—and place it in ZR1's path. Once ZR1 has gotten close enough, it should switch to slow speed and start pushing. While this is happening, try pressing its bumpers. What happens? Hopefully nothing! If ZR1 is in its "searching phase," the bumpers shouldn't be allowed to take control. Now observe what happens once ZR1 reaches the edge of the arena. The object should get pushed off, and when ZR1 sees the line, it should switch to fast speed. Because of the program's `speed` variable, ZR1 knows it is time to start moving fast again. After switching to fast speed, it will back up and execute a large turn to escape from the line.

After performing these tests, you have seen the basic, overall idea of what Brain-Bot version ZR1 can do. However, you can continue and experiment if you would like—there are multitudes of situations and tests you can set up.

Considering the Pros and Cons of ZR1's Design

Let's review the results of all our building, programming, and testing. We'll do this by summarizing the pros and cons of ZR1's design.

The Pros

Here are the main advantages of ZR1's design:

- The two-speed system makes the sumo-bot very efficient; it can go fast while searching and slow while pushing.
- The searching light sensor allows ZR1 to detect opponents without needing to make physical contact.
- ZR1's entire front end can detect the opponent.
- ZR1 excels at pushing from the front.
- The back protector subassembly protects the sumo-bot's back end from the opponent.

The Cons

The following are the disadvantages of ZR1's design:

- The switching subassemblies can get knocked off chassis.
- ZR1's extra wide structure causes it to sag a little in the middle.
- ZR1 might accidentally back over the line.
- The wheels are vulnerable to a hit.

The first, second, and fourth cons listed are discussed in Chapter 7 of *Competitive MINDSTORMS*. So, let's look at the third one here.

Since ZR1 doesn't have any sensors on the back end, and since it goes in reverse at times, backing over the line is a possibility (albeit a small one). You can fix this by adding a *third* light sensor to the back of ZR1, and using the light-sensor-and-touch-sensor-on-same-input-port trick. This will come at the cost of turning the release-type bumper into a press-type bumper—to accommodate the sensor trick—but it might prove even more beneficial than having a release-type bumper. You might want to try it and see how it works.

Conclusion

You have just seen how to implement a searching feature into an M-class sumo-bot. But, as noted at the beginning of the chapter, there are *many* different approaches you can take when giving your M-class sumo-bots (and other sumo-bots) a searching ability. What you saw here—using a light sensor to detect the opponent—is just one technique. I encourage you to experiment with other sumo-bot searching techniques on your sumo-bots!