Part Three

The Second Approach: The M-Class Strategy

CHAPTER 6

The Brain-Bot Chassis

Be prepared to throw off every last remnant of RIS-only restrictions and delve into a large and fascinating section of robotic sumo: the M-class strategy. No longer will we build simple and small chassis and sumo-bots; no longer do rule sets tie us down to single-set restrictions; no longer is robotic sumo a "little game" of tiny bots and little pushes. In this strategy, just about every definable factor and variable are quite different from those contained in the small-and-fast strategy; you could almost say it's a different game. In the midst of all these changes, one of them is extremely important: the plan of attack. Speed is still important, but having a powerful gear train for pushing is essential, too.

In the M-class strategy, many different kinds of gear trains become viable options, and methods that help to find the opposing sumo-bot abound. Complex and ingeniously designed sumo-bots are commonplace. However, more simple sumo-bots—which can be just as effective—use this strategy as well. The bottom line is that the diversity in the M-class strategy is simply amazing.

Why does the M-class strategy have these characteristics? The answer is simple: it is *in between* the small-and-fast and big sumo strategies. Small sumo-bots rely on speed and almost no push; big sumo-bots rely on push and almost no speed. Medium sumo-bots can cover a wide spectrum of designs and configurations, and they even have the potential to possess more than one aspect or ability. For example, how about a sumo-bot that can go both fast *and* slow?

In this chapter, you'll meet the Brain-Bot chassis, discover the meaning behind its name, and learn how to make a sumo-bot that can change its speed. Figure 6-1 shows the completed model.

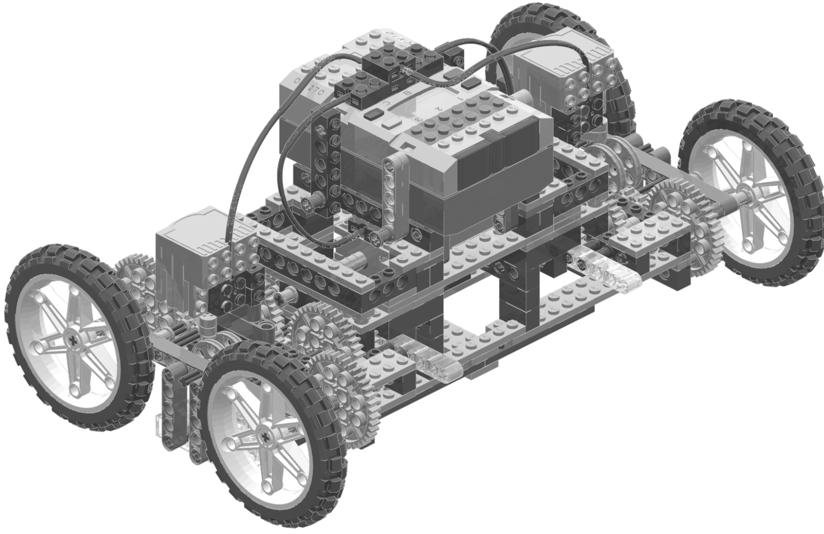


Figure 6-1. The finished Brain-Bot chassis

Building the Brain-Bot Chassis

Brain-Bot may seem kind of large (which it is), but this doesn't mean you'll need to build two dozen subassemblies to construct it. Instead, the majority of the construction happens in the drive subassemblies. These hold the motors that power the wheels, additional motors (as you will see in the final assembly), and, of course, the wheels. Fortunately, the designs for the left and right sides are identical; this means that you can build two drive subassemblies, and they will make up the core base of the robot. As for the rest of the robot, there is an RCX subassembly, two middle bulk subassemblies, bottom bracer subassembly, and two gear-switch subassemblies. In the final assembly, described in the "Putting the Brain-Bot Chassis Together" section, you will also add some additional pieces to the sumo-bot.

Brain-Bot is constructed mainly out of pieces from the RIS 2.0 and the Ultimate Builders Expansion Pack (UBEP). In addition to these sets, you will need one more motor, some extra #4 and #6 axles, and extra gears. Fortunately, the model doesn't have any rare or hard-to-find gears. However, you'll need quite a few common gears to complete Brain-Bot. Among these are more than a dozen 8t gears and eight 40t gears. Figure 6-2 shows Brain-Bot's bill of materials.

At this point, you might be wondering how Brain-Bot can use four motors, since there are only three output ports on the RCX. Four (or more) motors *can* be added to the RCX; you'll see how this works when you put the chassis together later in this chapter.

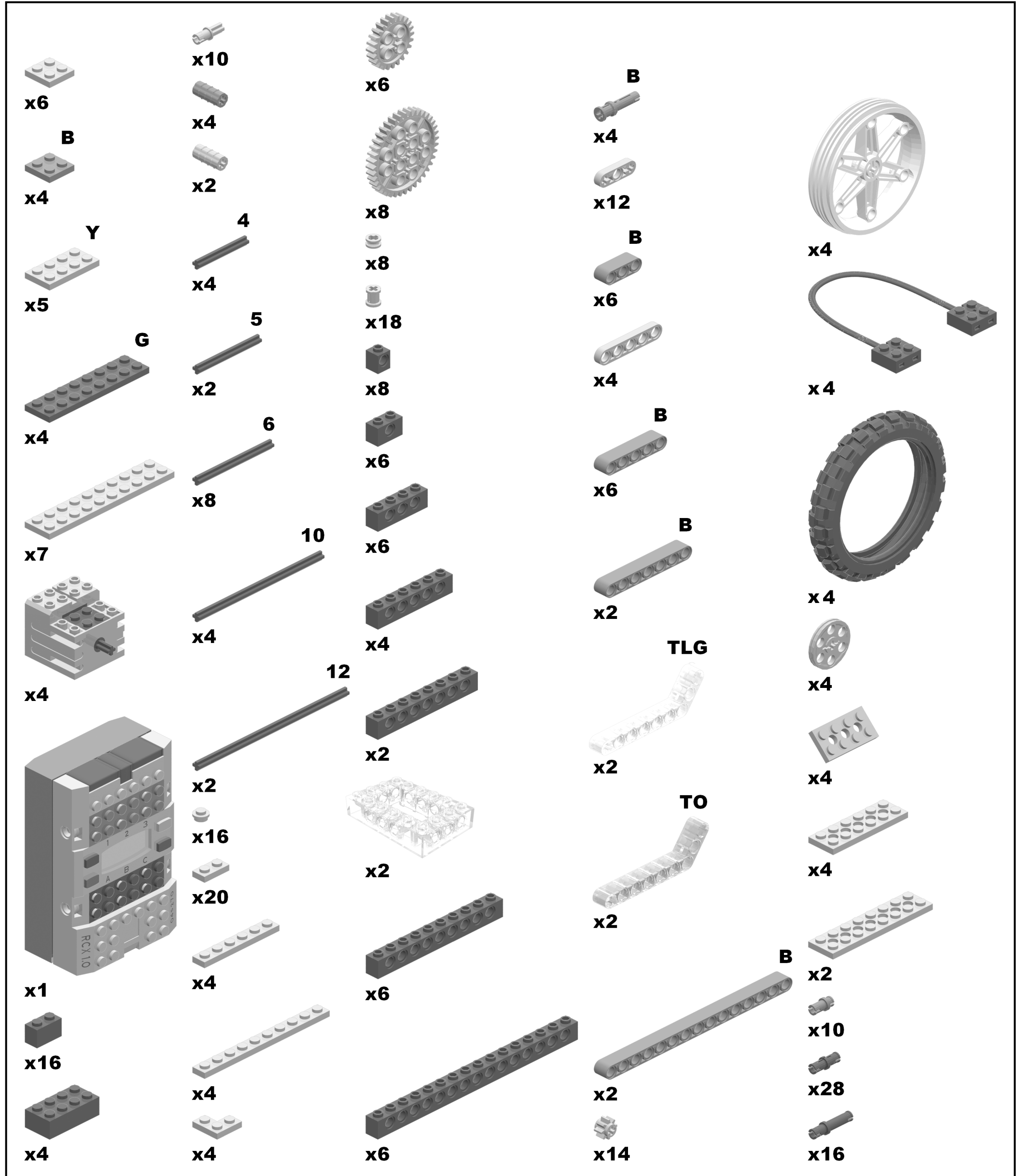


Figure 6-2. Brain-Bot's bill of materials

The Drive Subassembly

The drive subassembly is shown in Figure 6-3. The gearing in this assembly might leave you scratching your head, but once you have actually built it and seen it in action, it will make a lot more sense than it does at first glance.

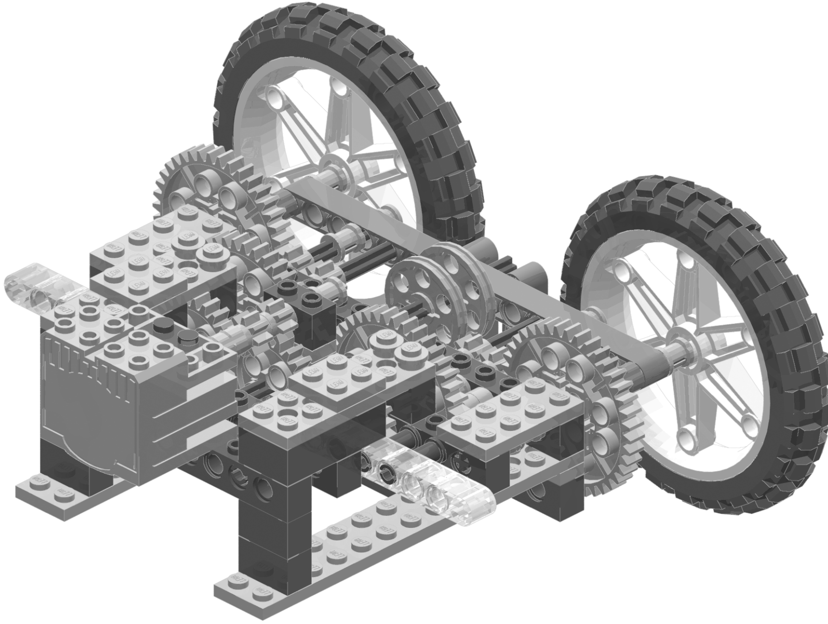


Figure 6-3. The completed drive subassembly

The primary purpose of this drive subassembly is mobility: it makes the robot go (a pretty simple concept). It can also switch to two different speeds—when another assembly is added later—but *that* isn't such a simple concept. The speed-control system as a whole is known as a *transmission*, and the system that actually changes the speed is known as a *gear switch*. And the gear switch does just that: switch gears.

Brain-Bot is designed to go a relatively fast speed in one mode, but slow (meaning a lot of torque, or pushing power) in another mode. This is so the sumo-bot can go fast while searching for the opponent and go slow while pushing the opponent. However, I encountered a difficulty in designing this. The problem is that gear switches don't work that way, or so it seems. Most gear switches give only a small change in speeds as they use 8t, 24t, or 16t gears to do the switching (giving only 1:3 or 3:1 ratio boosts, for instance). I wanted something quite different: a *slow* speed and a *relatively faster* speed. Tiny changes in the speed and torque just wouldn't do.

Ratios and Gears

What's the deal with ratios and how do they work? A ratio—for example, 1:3—will ultimately represent the rotation of the gears. I say *ultimately* because there are two ways to do your ratios, which we will examine in a moment. What's important to realize when making ratios and observing ratios is *which gear is turning which*. The ratio will always depend on which is the *powered gear* and which gear is being turned by the powered gear, or, in a setup involving multiple gears, which gear is the final output gear.

Here are the two ways to do your ratios, or, better put, the two views on ratios:

- **Rotation view:** This view has the ratios corresponding exactly to the rotations of the gears, which means an 8t gear turning a 24t gear would result in a 3:1 ratio; that is, the 8t turns three times for every one turn of the 24t. Likewise, a 24t turning an 8t would result in a 1:3 ratio, as the 24t turns once for every three turns of the 8t.
- **Gear teeth view:** This view goes by the number of teeth on the gears, which means an 8t gear turning a 24t gear would result in a 1:3 ratio. This ratio is achieved by counting the gear's teeth, putting them together as a ratio—8:24—and then simplifying them: 1:3. Likewise, a 24t gear turning an 8t gear would result in a 3:1 ratio, as the ratio would be 24:8, which would then simplify to 3:1.

To truly explain gears and ratios, and to comprehend the actual physical forces a ratio represents, you must consider the rotations of the gears. However, the gear teeth view can also effectively explain ratios and is easier to understand and use. Although gear ratios have plenty to do with LEGO MINDSTORMS robotic sumo, they aren't what it's all about. We'll be concentrating more on robotic sumo itself in this book, so we'll take the easier-to-understand gear teeth view.

Which ratio view should you use in your MINDSTORMS career? Really, it's nothing more than a matter of personal opinion. If you choose the rotation view, you'll be using true ratios that refer to the actual rotation of the gears. If you choose the gear teeth view, the numbers in the ratios can be easier to comprehend and use.

After a great amount of experimentation and work, the end result is what you see: the drive subassembly. This assembly uses 40t gears to attain greater ratio boosts instead of using only 24t and 8t gears. However, it also uses other gears, which combine to make up a complex gear train that gives the final (and desired) result. That result is a 1:15 ratio in fast speed, obtained by multiplying these ratios: 1:3 × 1:5 = 1:15. This would be considered slow by the small-and-fast strategy's standard, but for the M-class strategy, it's a quite acceptable ratio, since the sumo-bots are bigger and heavier. Figure 6-4 highlights the gears responsible for the fast speed in the drive subassembly.

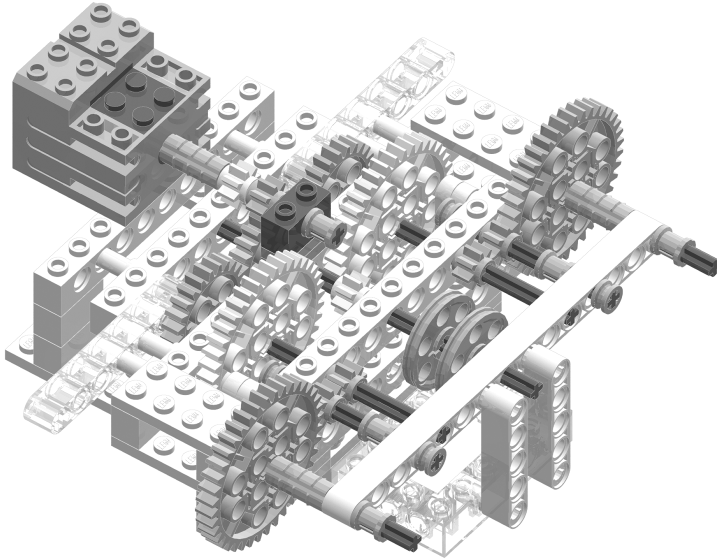


Figure 6-4. The drive subassembly's fast speed

For the slow speed, Brain-Bot has only one additional set of gears that the power is transferred through, but this makes all the difference. Figure 6-5 highlights the gears responsible for the slow speed. Now let's do the math: 1:3 × 1:5 × 1:5 = 1:75! That's a lot of torque! And these are just the kinds of numbers I wanted to see.

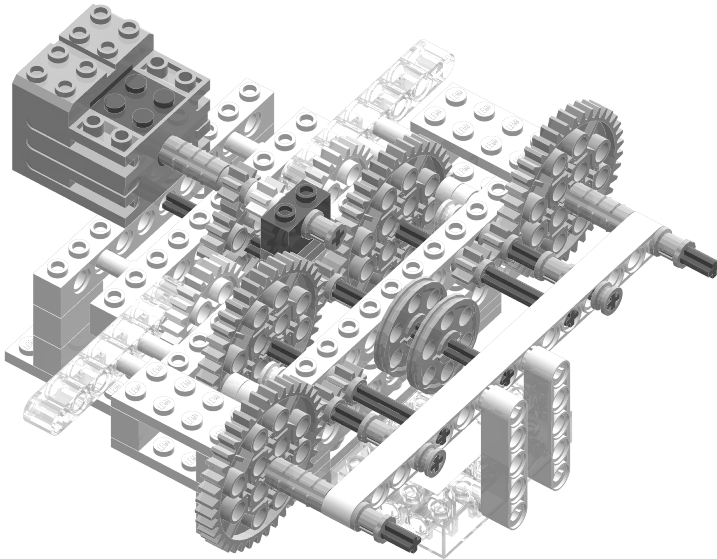
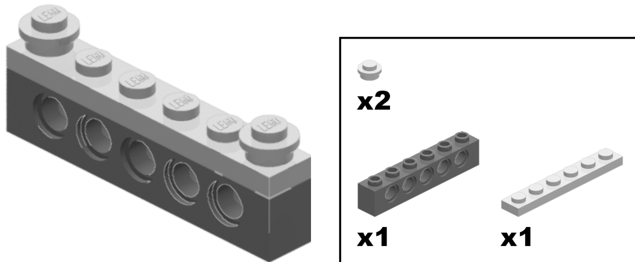


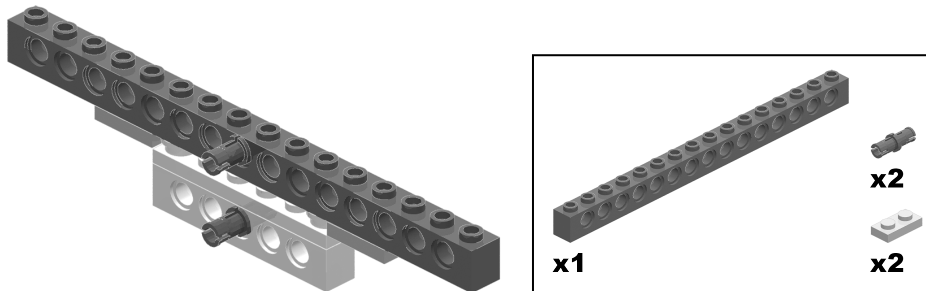
Figure 6-5. The drive subassembly's slow speed

Now that you know what it does, let's build the drive subassembly—the first (and the most important) assembly. Note that you will build *two* drive subassemblies. In the final assembly, one goes on the right side and one goes on the left side; together, they make up the majority of the sumo-bot and the core chassis itself.

Steps 1 and 2 do some simple and basic work with beams and plates.

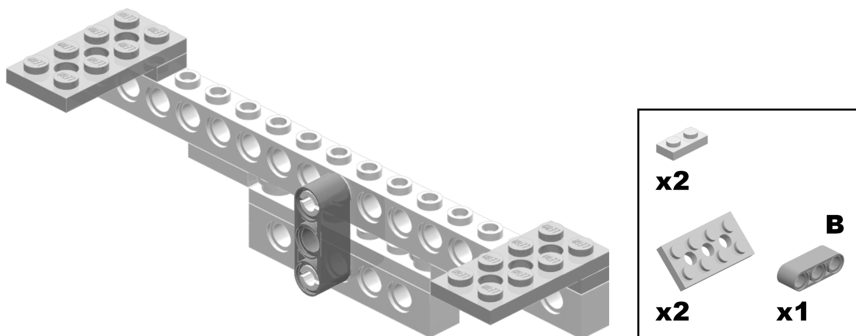


Drive Subassembly Step 1



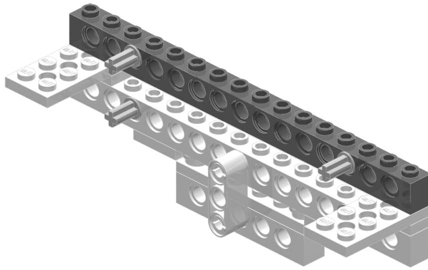
Drive Subassembly Step 2

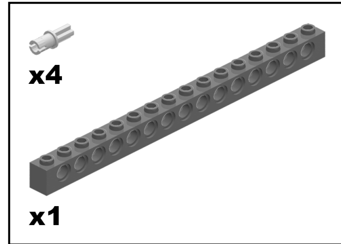
Step 3 braces the beams with a piece from the UBEP—the 1x3 blue liftarm—and adds more plates.



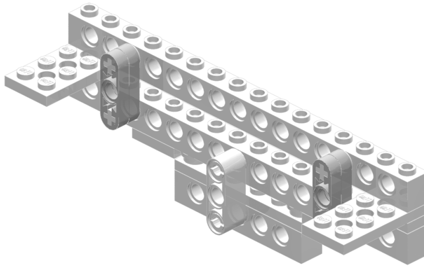
Drive Subassembly Step 3

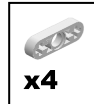
Steps 4 and 5 build the assembly up another layer, and then brace that as well; this time with axle pins and the 1x3 gray half-liftarms that are easily found in the RIS.





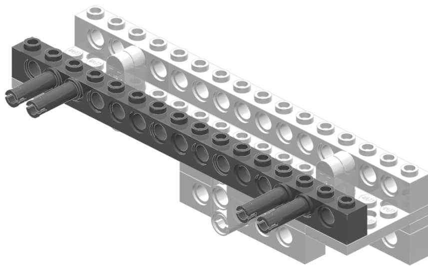
Drive Subassembly Step 4

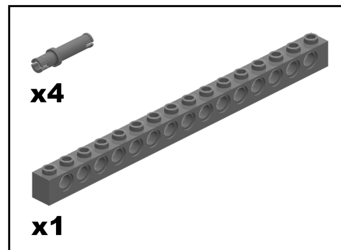




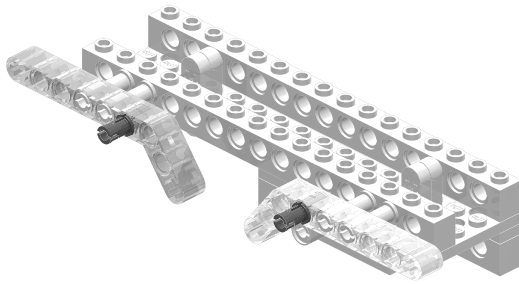
Drive Subassembly Step 5

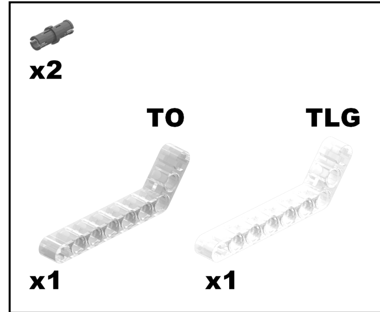
Now things begin to get a little more interesting. The assembly needs to be widened—no problem! Steps 6 through 9 stretch out the structure a good amount. This is accomplished with a special method: connect long friction pins into a beam or liftarm, attach another beam/liftarm, put more long pins in that, and keep going! This climaxes in step 9 with one last beam—you also add four 1x2 black bricks and two 2x10 plates to the bottom of this beam.



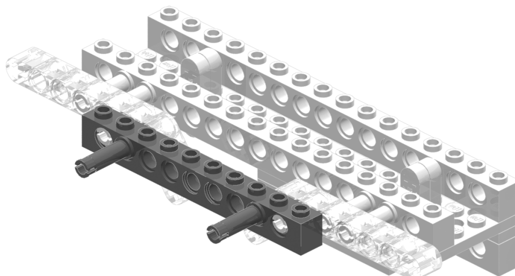


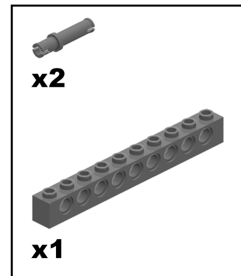
Drive Subassembly Step 6



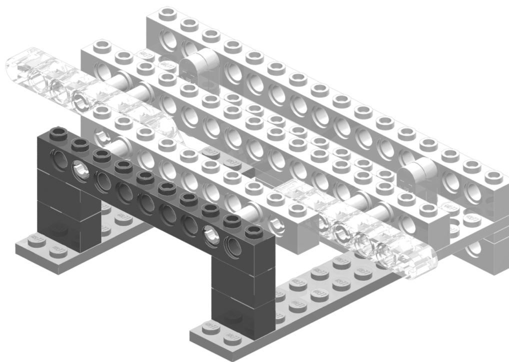


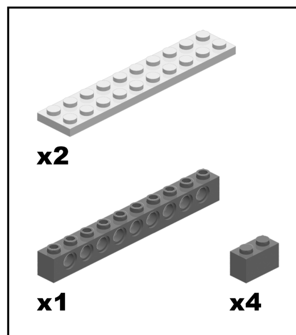
Drive Subassembly Step 7





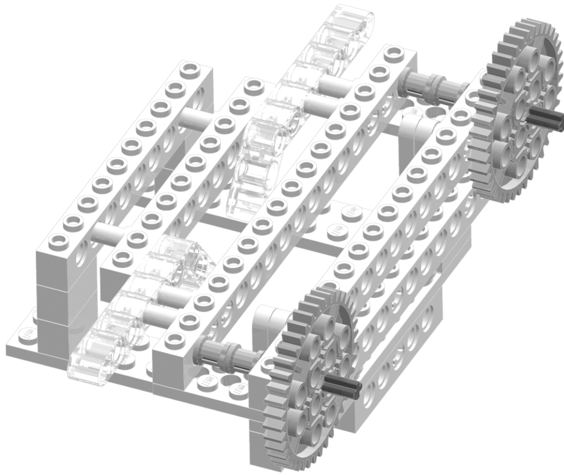
Drive Subassembly Step 8

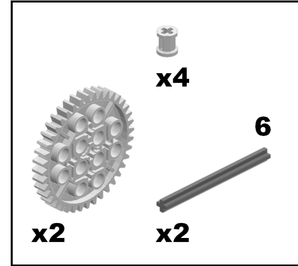




Drive Subassembly Step 9

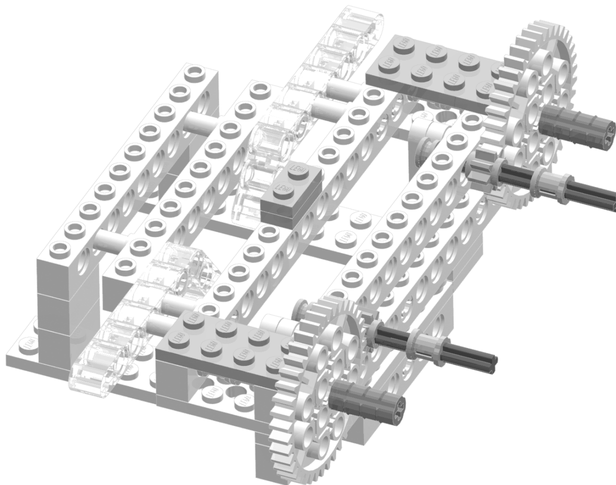
With the main bulk of the assembly done, you are now ready to work on the gearing. Turn the model so the other side is facing you. In step 10, add 40t gears along with bushings and #6 axles.

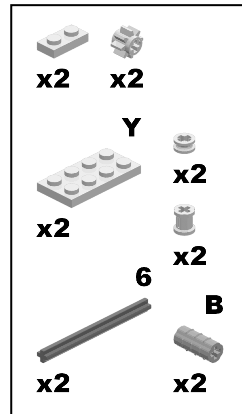




Drive Subassembly Step 10

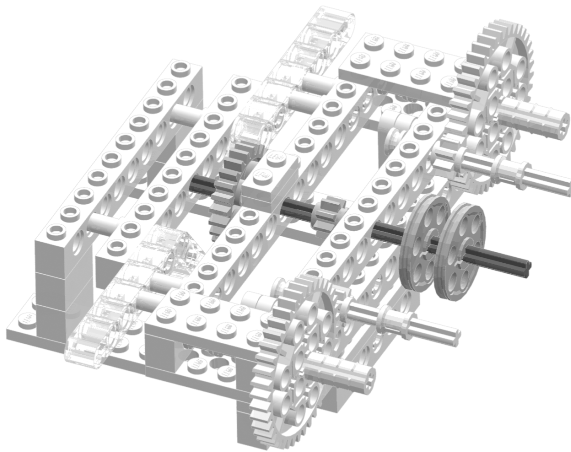
Step 11 adds plates, axle extenders, and more gears—this time, 8t gears.

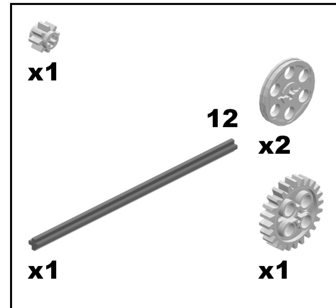




Drive Subassembly Step 11

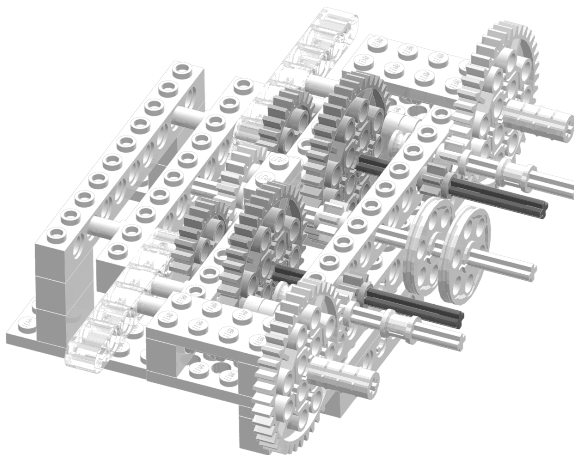
In step 12, build and add the main switching axle as shown. Notice the "breathing space" of one stud for the 24t and 8t gears added in this step. When you move the switching axle (to switch the speed), the gears on that axle can easily be moved as well.

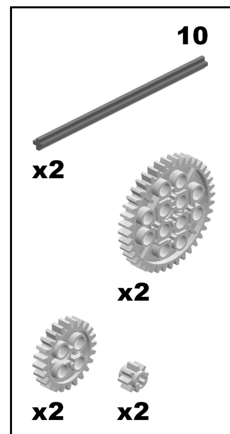




Drive Subassembly Step 12

Now, in step 13, add the final gears, which include two more 40t gears, 24t gears, and 8t gears.

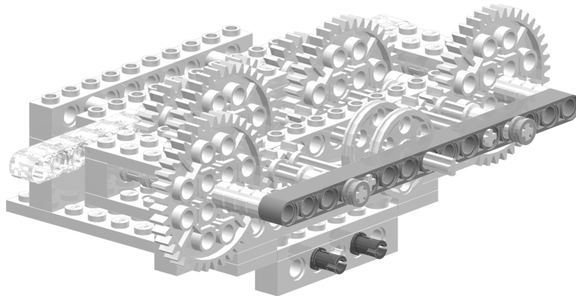


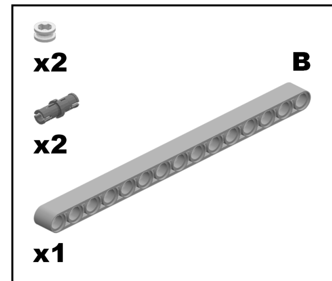


Drive Subassembly Step 13

As you have probably noticed, all those gears have their accompanying axles protruding out into space. You are now going to take care of this. As shown in step 14, take one of the *1x15 straight liftarms* from the UBEP, slide it onto the axles, and top off two of them with half-bushings.

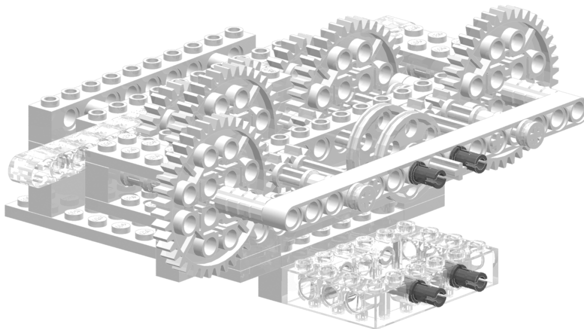
There is one other part to this step. Underneath all this building activity is a single 1x6 beam—the one you added back in step 1. It is now time for this beam to do something! There will be a number of pieces attached either directly or indirectly to this beam, and you start off by snapping two friction pins into it.

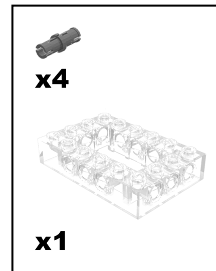




Drive Subassembly Step 14

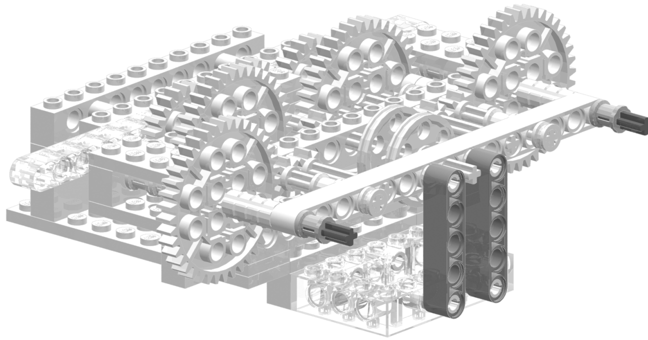
In step 15, connect one of the transparent TECHNIC 4x6 bricks with an open center onto the friction pins from the previous step, and add four *more* friction pins to the assembly.

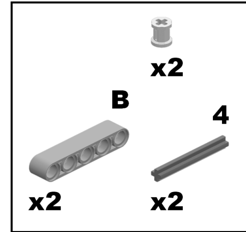




Drive Subassembly Step 15

Step 16 makes it easy to see the purpose behind all this: bracing! This step uses two of the 1x5 blue, straight liftarms from the UBEP for the bracing, and also adds some axles for the wheels.

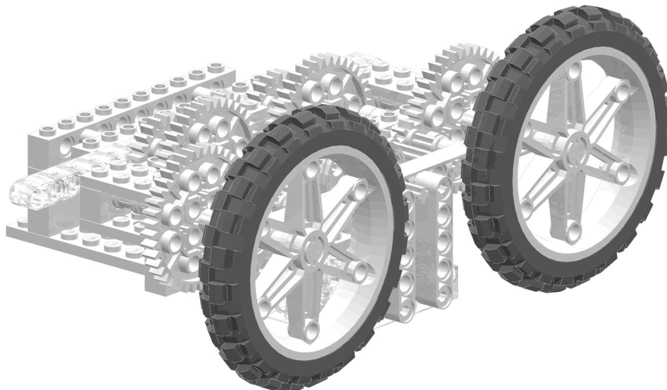




Drive Subassembly Step 16

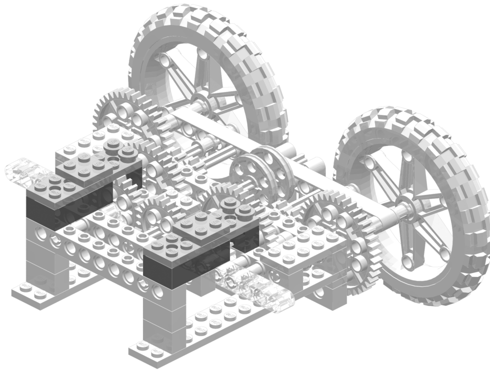
Step 17 adds the wheels for this sumo-bot.

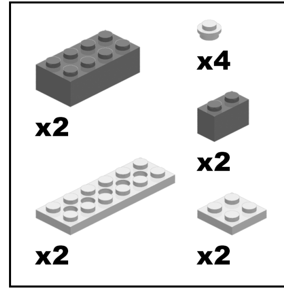
NOTE *You might be wondering, "Why these wheels again? Why not different wheels?" We're using these wheels again because they provide good traction and speed. Also, they are readily available; every version of the RIS includes four of them. Smaller wheels don't give as good of a performance, and, actually, using smaller wheels won't work in this model. Try putting some on and see what happens.*



Drive Subassembly Step 17

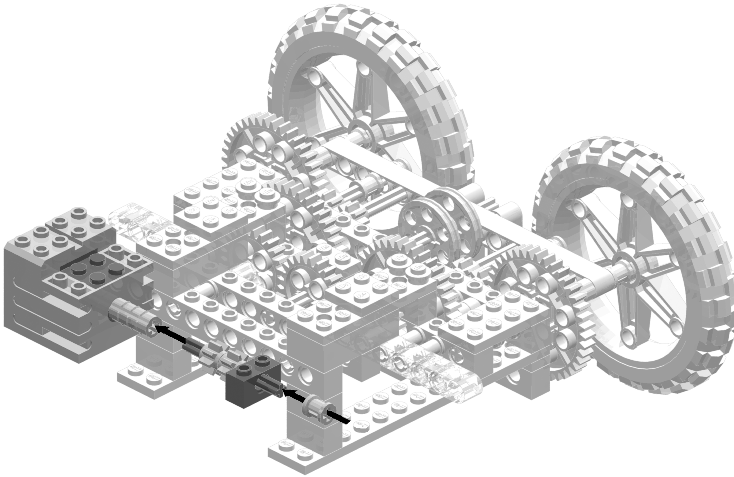
Step 18 builds up some bricks and plates for later use.

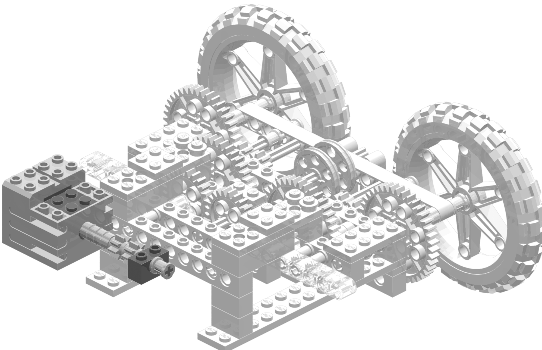


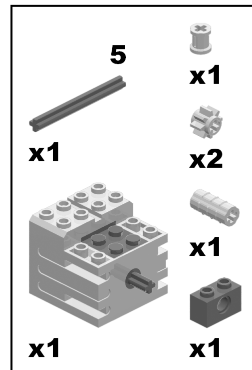


Drive Subassembly Step 18

In step 19, pull out a motor and attach the various pieces to it as shown.

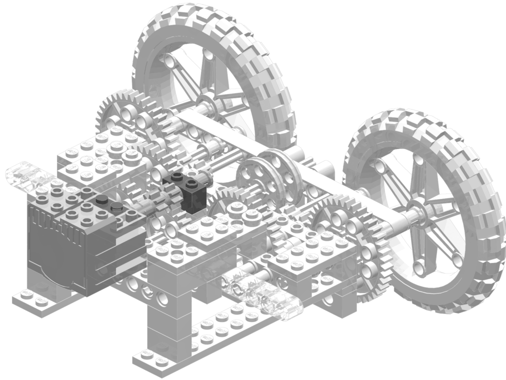






Drive Subassembly Step 19

Lastly, in step 20, after you've added the pieces to the motor, add the motor itself to the assembly.



Drive Subassembly Step 20

You've completed the last building step for the drive subassembly, but right now there's a potential problem in the assembly. Do you see the axle on the motor's shaft, which holds two 8t gears and goes into a 1x2 beam at the end? Well, on that axle is a bit of "extra breathing space", and this will cause a gap—the gap could appear in between the two 8t gears, in front of the 8t gears, or in other places along the axle. Although this gap is very small, it will eventually cause problems when trying to switch speeds by snagging the moving gears. You need to eliminate this little gap, but how? You can't add anything else to the axle—is there any solution?

MindStormers (like you and me) sometimes come up against a problem like this one. Often, the answer to the problem is to redesign the problem spot; but redesigning really wouldn't work here because of the nature of the problem. A major redesign might solve the problem, but then the sumo-bot wouldn't be Brain-Bot anymore. Instead, here are the two simple steps you can perform to remove the little gap:

1. Take the axle extender on the motor and push it forward against the gap until it disappears. Once you do this, the gap is transferred to the other side of the 1x2 beam.

2. While firmly holding the axle extender so that it can't move backwards, push the bushing at the end of the axle all the way towards the 1x2 beam.

This solves the problem and properly seals the axle on that end. There is a bit of extra axle coming out of the bushing, but it can be safely disregarded. This little fix works excellently; sometimes the best solutions are the simplest ones.

> **CAUTION** *Fixing the gap is a small but crucial task. If you don't do it, the whole sumo-bot might not work properly.*

After fixing the gap on your drive subassembly, you are finished with your first subassembly. Your drive subassembly should look like Figure 6-3, shown at the beginning of this section.

Now, follow the same steps to build another drive subassembly (remember that you need *two* drive subassemblies). Don't forget to fix the gap problem on both of them!

The Left Switch Subassembly

The left switch subassembly, shown in Figure 6-6, is responsible for switching the gears that will change the output speed of the robot. How does it work? First, the RCX sends a short burst of power—at a certain power level—to the motor. Then 1x3 liftarms connected directly to the motor move thin *wedge wheels* (located on the drive subassembly); these wedge wheels are on the main switching axle, so the switching axle is moved. This also means the gears on the switching axle are moved, and that is how you change the speeds.

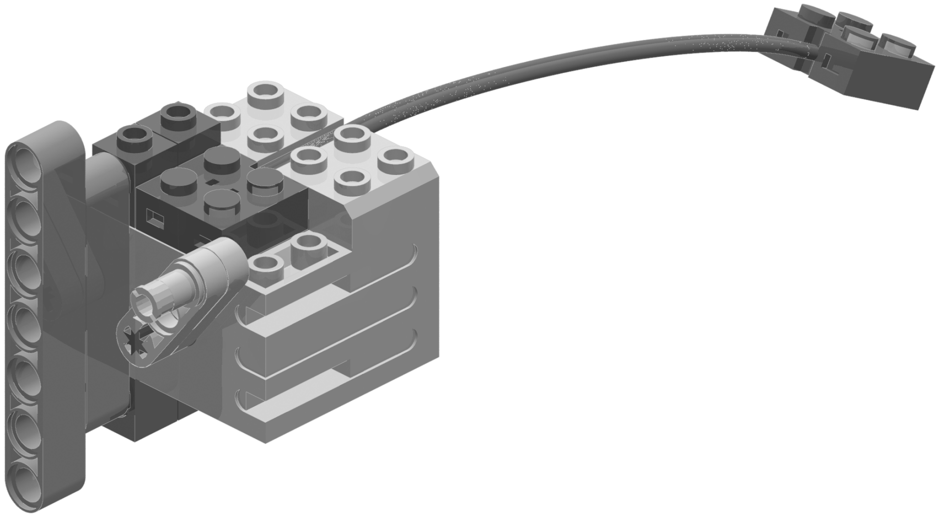
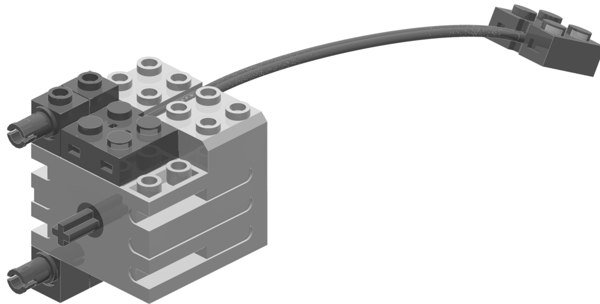
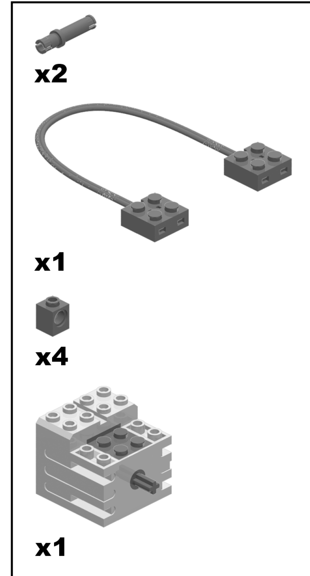


Figure 6-6. The completed left switch subassembly

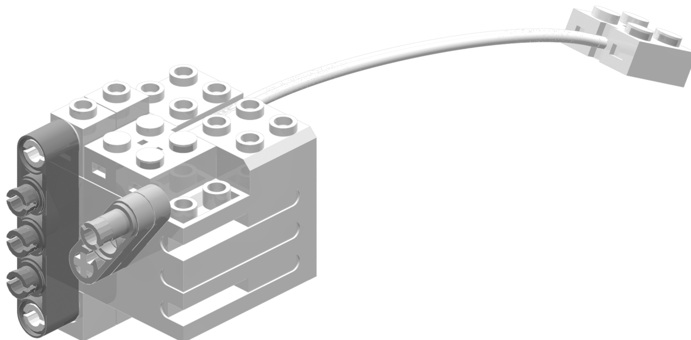
In step 1, you add an electrical wire, facing towards the back, on a motor, and some pieces you haven't used yet in this book: *1x1 bricks with a hole*. Put four of these on, as shown, with a long friction pin running through them.

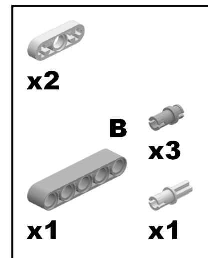




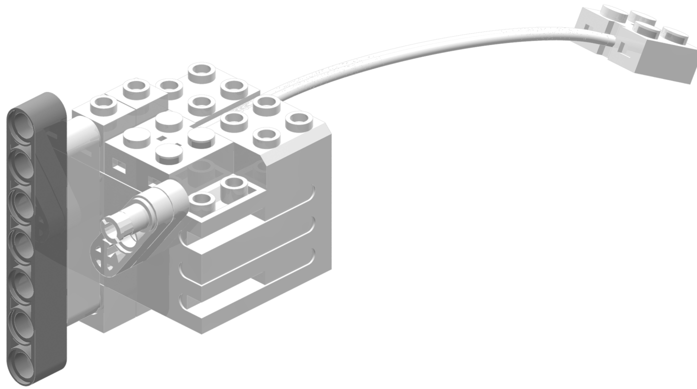
Left Switch Subassembly Step 1

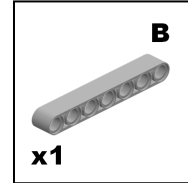
Steps 2 and 3 add the actual switching mechanism to the motor and construct the section that will connect to the chassis (you'll see exactly how it connects in the final assembly).





Left Switch Subassembly Step 2





Left Switch Subassembly Step 3

Your left switch subassembly is now complete. It should look like Figure 6-6, shown at the beginning of this section.

The Right Switch Subassembly

The construction for the right switch subassembly, shown in Figure 6-7, is exactly the same as for the left switch subassembly, except that everything is mirrored. And as you would expect, this assembly will be going on the right side of the sumo-bot. To build a right switch subassembly, follow the instructions given for the left switch subassembly, but change the orientation of the pieces while you're building so that the completed subassembly looks like Figure 6-7.

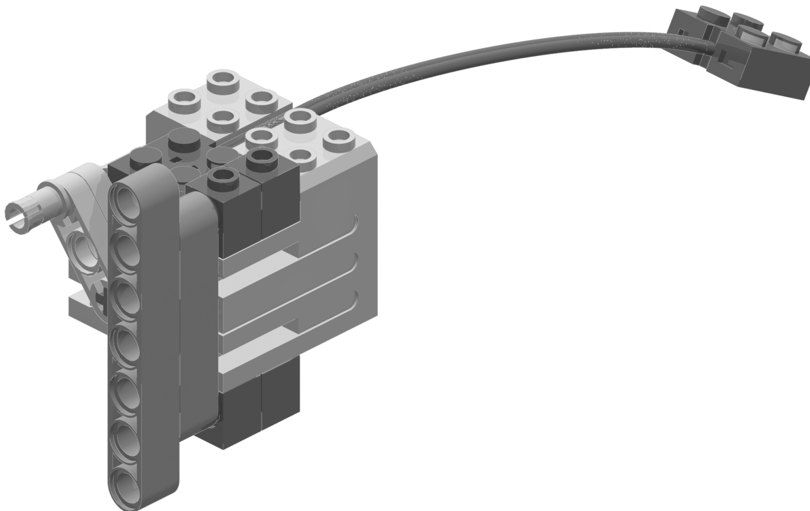


Figure 6-7. The completed right switch subassembly

The Middle Bulk Subassembly

With a name like "middle bulk subassembly," you can't help but wonder, "What exactly is this?" It is just what its name implies: bulk that goes in the middle. Once the basic layout of Brain-Bot is put together, there seems to be an "unfinished" spot in the middle—a spot that needs something to be there to make the sumo-bot complete. This is where the middle bulk subassembly, shown in Figure 6-8, comes in. Not only does it make the robot complete, the middle bulk subassembly is a great place to attach other pieces. In fact, the spot in this model where the RCX will rest is on top of the middle bulk subassemblies. This brings up another point: you will need to build *two* of these subassemblies. The left and right sides of Brain-Bot are identical, so two middle bulk subassemblies are necessary.

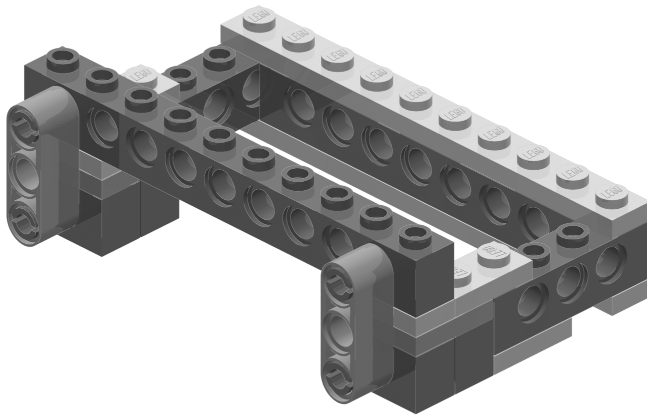
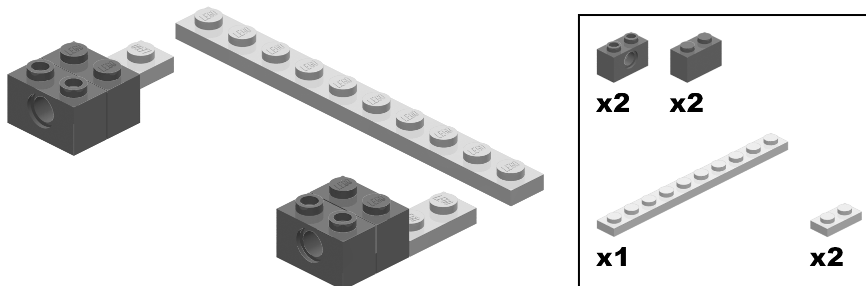
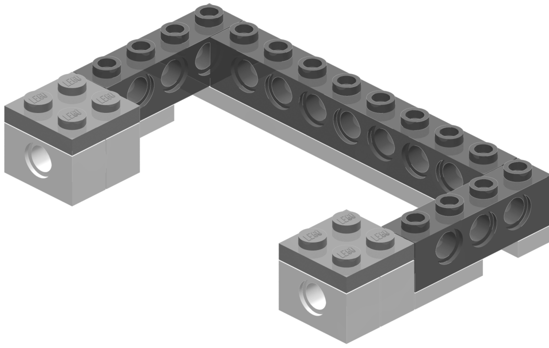


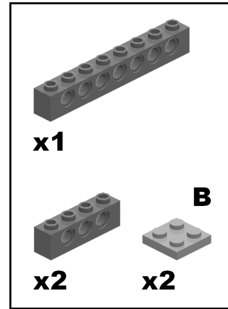
Figure 6-8. The completed middle bulk subassembly

In steps 1 and 2, you add a series of plates and beams, which are stacked up, to form the base.



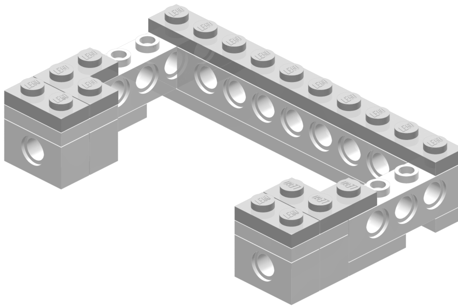
Middle Bulk Subassembly Step 1

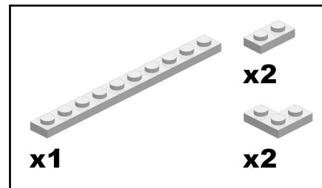




Middle Bulk Subassembly Step 2

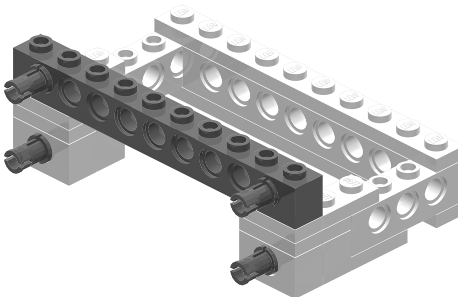
Step 3 adds more plates; these are mostly for reinforcement.

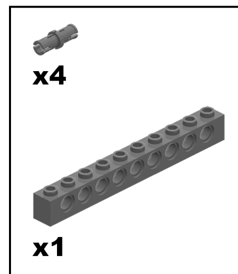




Middle Bulk Subassembly Step 3

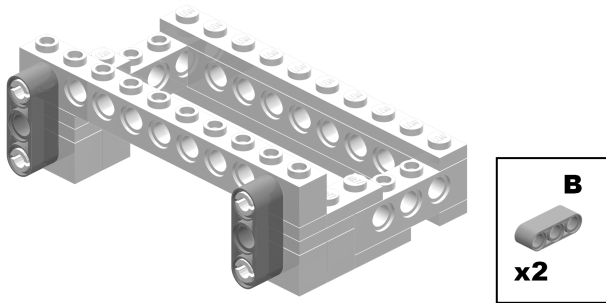
Step 4 places another beam on top of the plates closest to the front and four friction pins in the beams.





Middle Bulk Subassembly Step 4

Bracing time again! In step 5, take two 1x3 blue liftarms, found in the UBEP, and snap them directly onto the friction pins.



Middle Bulk Subassembly Step 5

That completes the middle bulk subassembly construction, and it should look like Figure 6-8, shown at the beginning of this section. Remember to build *two* of these.

The RCX Subassembly

The RCX subassembly, shown in Figure 6-9, will make your life easier when you do the final assembly. Instead of needing to build this directly onto the chassis in the final assembly, you can build it here and then easily slip it onto the chassis. Once again, in the final assembly, the RCX will not be directly attached or pushed onto bricks. Just as in the Zip-Bam-Bot chassis, only the edges of the RCX, which cannot attach to anything, are placed on studs. Therefore, this assembly—or should I say the RCX—is attached to the chassis by means of four of those ever-useful blue pins.

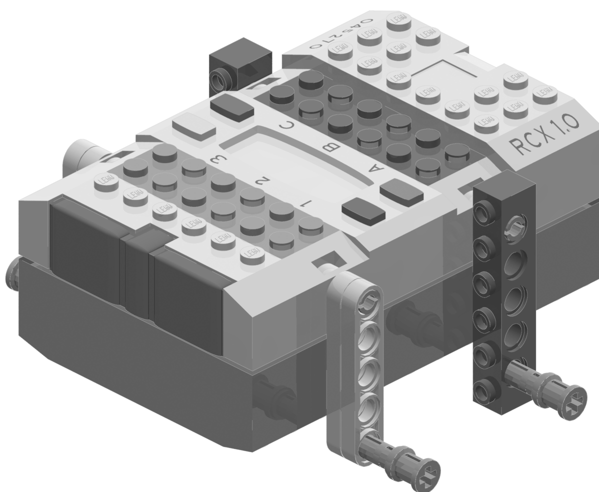
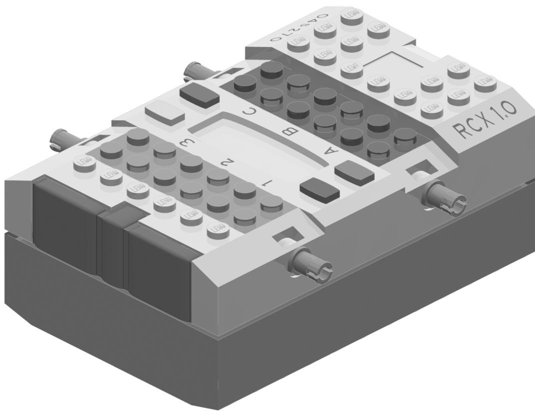
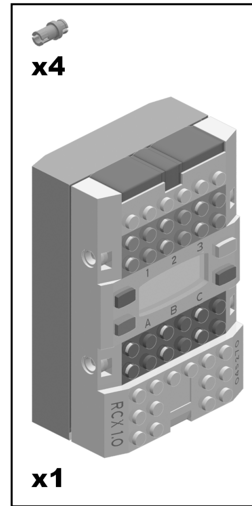


Figure 6-9. The completed RCX subassembly

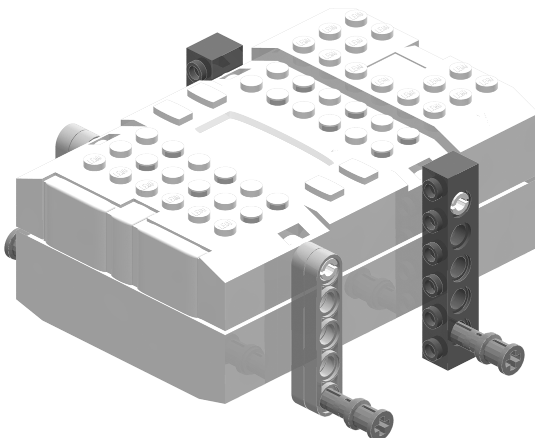
NOTE *A bit of LEGO MINDSTORMS history: If you have only owned the RIS 2.0, you might not know that the blue pins used in this RCX subassembly did not make their appearance until the RIS 1.5. The RIS 1.0 didn't have quite as many axle accessories, and instead was more "brick-oriented." When the RIS 1.5 came out, some of the bricks were removed and several more "axle-oriented" pieces were introduced, including the blue pins with stop bushings.*

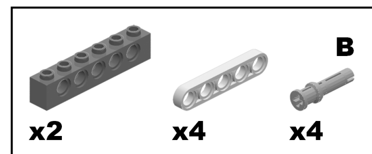
Construction is quite simple. In step 1, push four 3/4 pins into the RCX. In step 2, attach 1x6 beams and double layers of 1x5 half-liftarms onto the pins, and then push four blue, long pins with stop bushings into the liftarms and beams.





RCX Subassembly Step 1





RCX Subassembly Step 2

The Bottom Bracer Subassembly

The bottom bracer subassembly, shown in Figure 6-10, is nothing more than plates connected together in a certain pattern—a pattern that will conform perfectly to the bottom of the robot and strengthen the chassis. Without this subassembly, Brain-Bot would be much weaker. In a sense, the bottom bracer subassembly welds the robot together by attaching to both of the drive subassemblies.

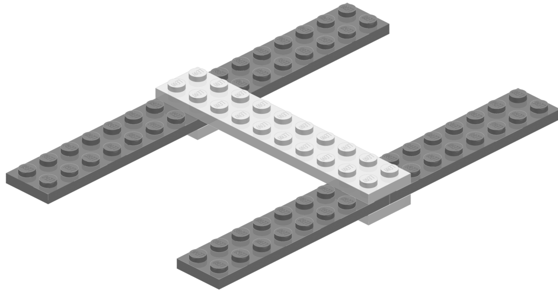
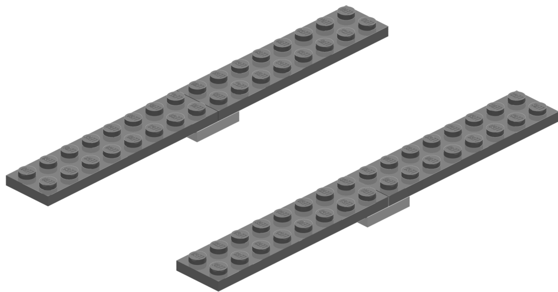
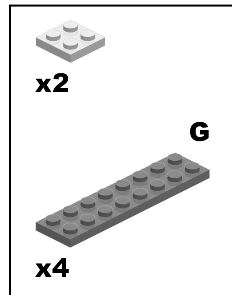


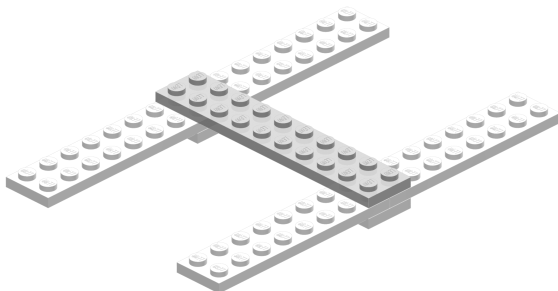
Figure 6-10. The completed bottom bracer subassembly

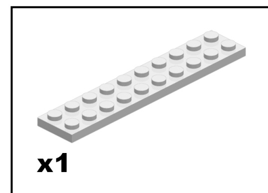
The construction involves only two steps.





Bottom Bracer Subassembly Step 1



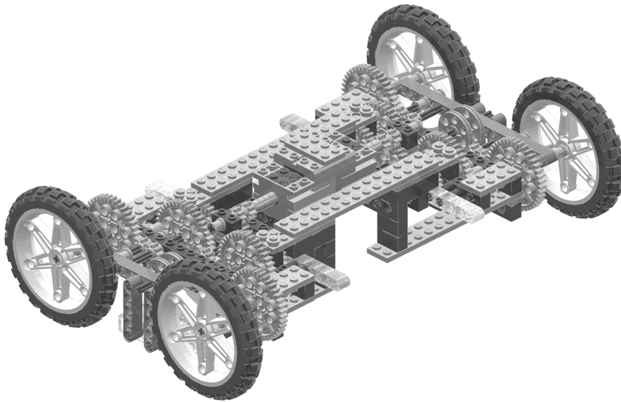


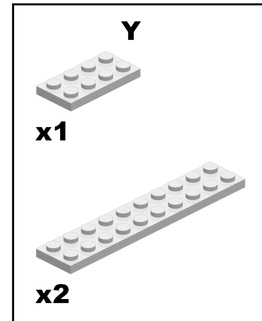
Bottom Bracer Subassembly Step 2

Putting the Brain-Bot Chassis Together

With all the subassemblies finally completed, the long-awaited moment has come: the final assembly of the Brain-Bot chassis.

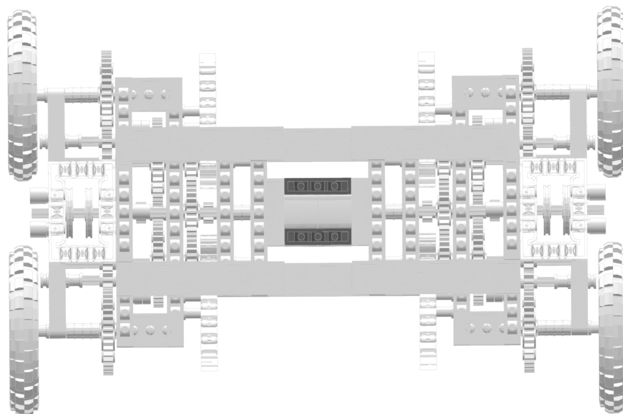
To begin, get both **drive subassemblies** you constructed at the start of this chapter. Place them with their wheels facing away from each other and their motors touching together. Attach a 2x4 yellow plate to the two motors, and two 2x10 plates to the two assemblies, as shown in step 1.

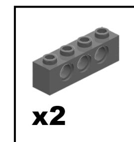




Brain-Bot Chassis Step 1

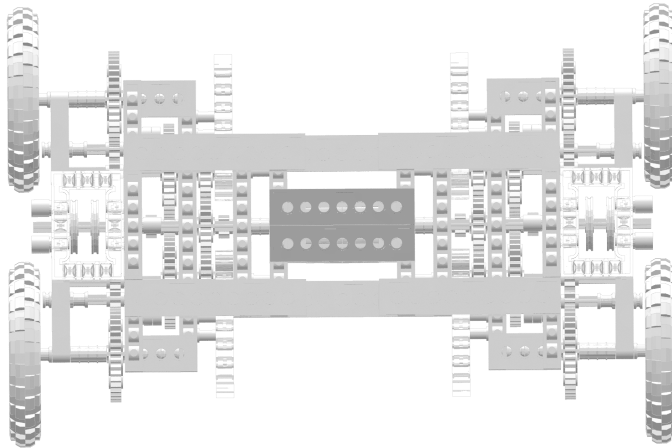
Before you start putting on vital pieces such as the motors or the RCX, you need to strengthen the chassis quite a bit more. You will do this by firmly bringing the two drive subassemblies together in steps 2 through 4. In step 2, turn the model on its top (carefully!) and put two 1x4 beams onto the bottom of the motors. These beams further connect the two motors together.

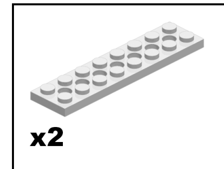




Brain-Bot Chassis Step 2

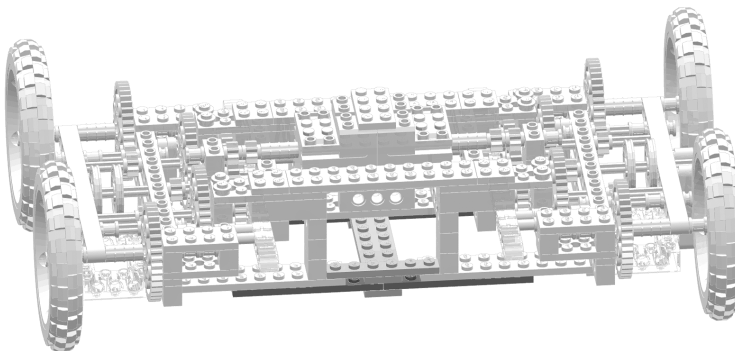
Now attach two 2x8 TECHNIC plates, as shown in step 3. These plates further strengthen the motors and also the whole chassis.

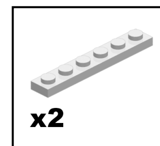




Brain-Bot Chassis Step 3

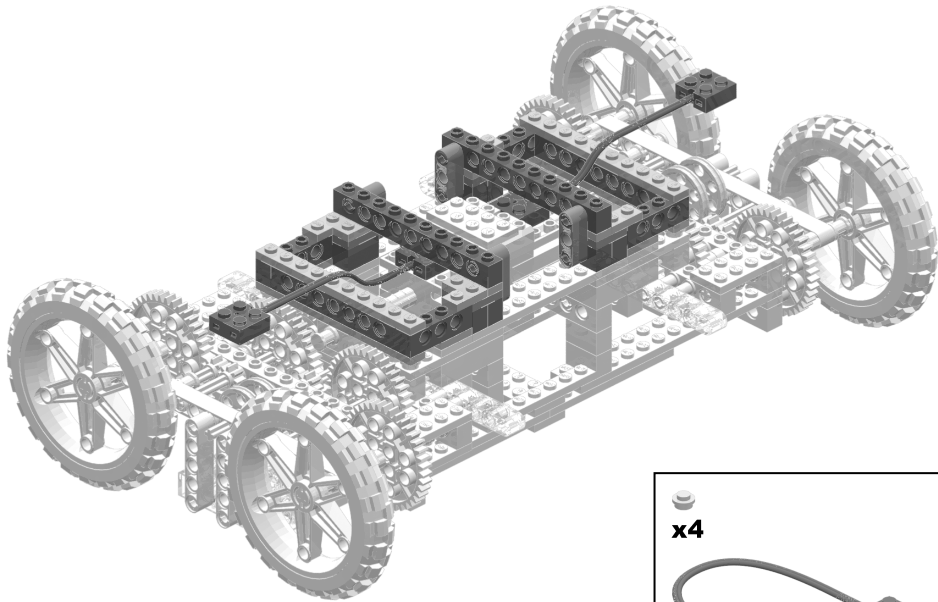
However, the chassis is still not strong enough. One more series of plates is necessary. How can you do this? It's easy: pull out the **bottom bracer subassembly** and attach that! While you're at it, also add two 1x6 plates to the front and back of the sumo-bot. These plates attach to the bottom bracer subassembly and both sides of the robot, as shown in step 4 (one of the 1x6 plates is not visible in the image).



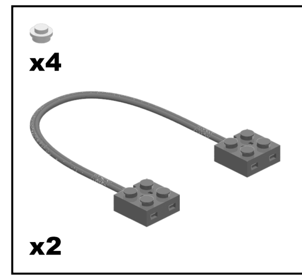


Brain-Bot Chassis Step 4

Now you can whip out your two **middle bulk subassemblies**, and attach them as shown in step 5. Actually, first you attach an electrical wire to each motor, then two 1x1 round plates to each electrical wire's 2x2x2/3 plates which are connected to the motors, and then you place the middle bulk subassemblies on top.



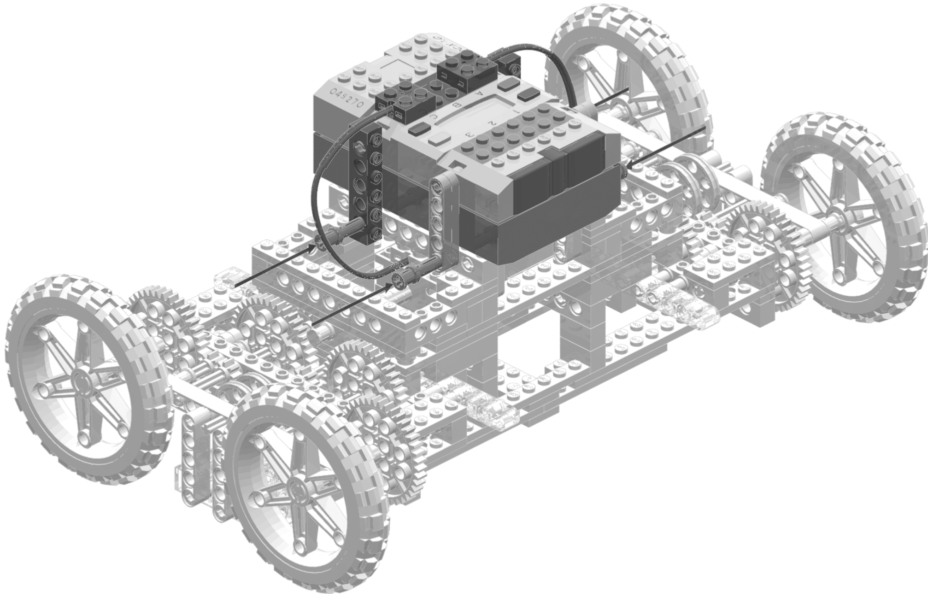
Brain-Bot Chassis Step 5



Step 6 adds the **RCX subassembly**, uses the four blue stop pins to firmly attach it to the chassis, and connects the "free" electrical wires to the RCX. Remember that all you need to do to remove the RCX is pull out those blue pins. There are a number of reasons why this is a great feature to possess, but here is one of them: changing the batteries is painless! Connect the motor's electrical wires to the RCX like this:

- The **right drive subassembly's motor** goes on **output port C.**
- The **left drive subassembly's motor** goes on **output port A.**

TIP *Among MINDSTORMS fans (and LEGO fans as a whole) there are no rules, but there are a few guidelines that are recognized and generally followed. Among these guidelines is the RCX rule: make access to the RCX easy, prevent the infrared port from being obstructed, and allow easy detachment and attachment of the RCX. As you can see, Brain-Bot abides by these guidelines. However, these guidelines are only suggestions, and not all models have the capability to use them. So, if you use them, that's great! If you don't, that's just fine, too.*

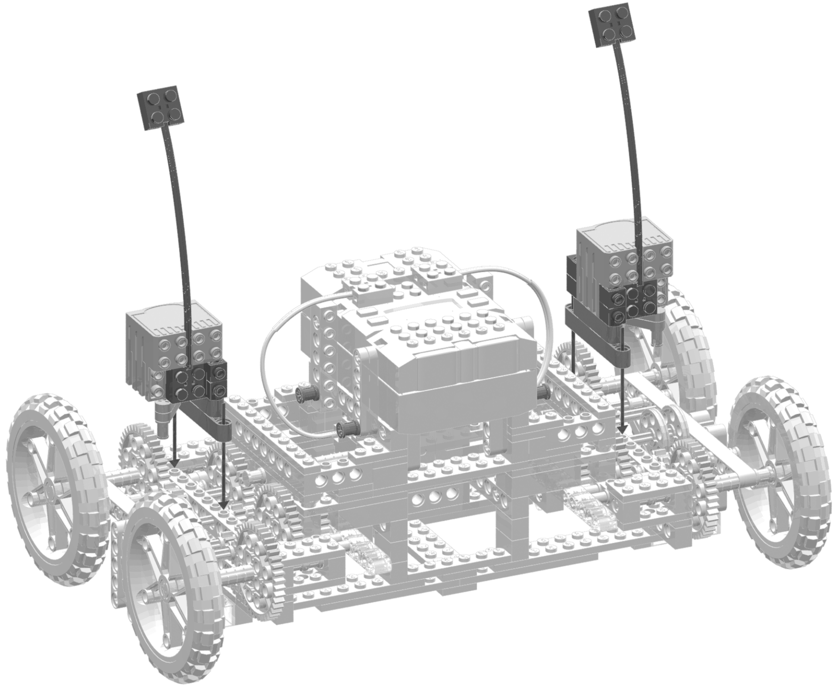


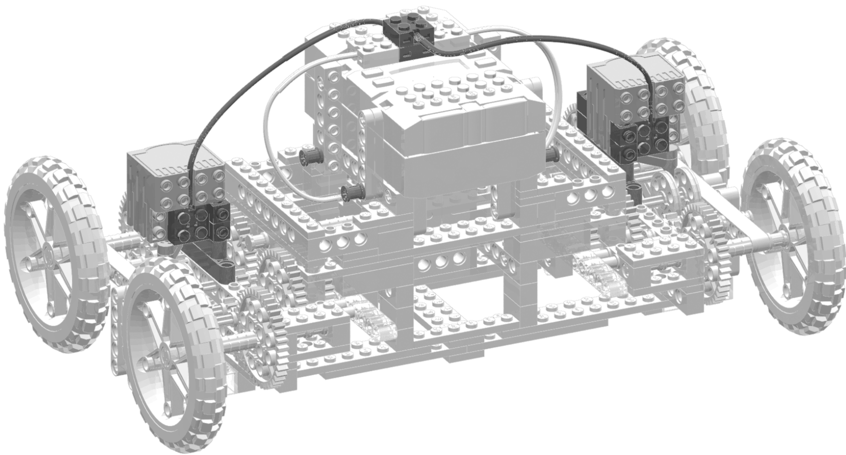
Brain-Bot Chassis Step 6

Only one step left! In step 7, add the **left** and **right switch subassemblies**. As you can see, they are pushed down onto the drive subassemblies. The blue liftarms on the switch assemblies actually go down onto the beam's studs and make a connection. Do you remember this type of connection from Chapter 4? It's a slightly unusual connection, but it's a useful one!

CAUTION *When pushing the switch subassembly's 1x7 blue liftarms onto the chassis, make sure you push them* all the way *onto the beam's studs. If the connection isn't solid, the switch subassemblies might work themselves off the chassis. One way to ensure you have a firm connection is to take the 1x7 blue liftarms off the switch subassemblies, attach those to the chassis first, and then snap on the rest of the assemblies.*

As mentioned at the beginning of this chapter, having more motors than there are motor outputs isn't a problem—there is an easy way to work with more than three motors. And you can see how to do it in step 7: both of the switch subassembly's wires connect to *one* port (**output port B**) on the RCX! This is completely "legal" and will not harm the RCX or motors in any way. The important thing to remember is the orientation of the wires. The motor on the right should have its wire on the bottom and going out the back; the motor on the left should have its wire on top and going out the front towards the LCD.





Brain-Bot Chassis Step 7

Once you've completed step 7, you are finished building the Brain-Bot chassis. Your model should look like the completed chassis shown in Figure 6-1, at the beginning of the chapter.

Programming the Brain-Bot Chassis

After seeing cool gear switches and impressive gear trains, you are probably ready to see Brain-Bot really work. Besides that though, it's a good idea to test the basics of a chassis before doing some serious sumo-bot programming on it (although, in this case, the chassis has been rigorously tested).

But how did Brain-Bot get its name? *Brain* signifies, as you probably guessed, thinking power or being smart. Not only does this allude to the fact that Brain-Bot is smart, it also means Brain-Bot has the *capabilities* to be smart; the M-class strategy allows this to happen. With all the amazing features it has to offer, the M-class strategy holds endless "smart" possibilities. Most of this smartness goes on in the programming; in other words, the program that runs the sumo-bot is what will make it truly smart.

In the programs for Brain-Bot, we will add something new: a *header*. A header is a separate, different file from your .nqc program and has the filename extension .nqh.

Programming Brain-Bot with a Header

Before we go any further, we need to take a closer look at headers and why they are useful.

In the C or C++ language, which NQC is based on, most programs have code at the very top that serves the purpose of including certain files for that program, so it can access specific functionality. As an example, let's say we are writing a program to create textures for a racecar game in C or C++ (however out of place that example may be). To get the textures for the cars, we need to include a file in the program with a line of code that says:

```
#include <cartexture.h>
```

Notice that it is of a file type .h instead of the typical .c in the C language. NQC has a file type of .nqc, which mimics .c, and it also has a file type of .nqh, which mimics the .h files of the C language. To include files in your NQC programs, you use the following line of code (the filename is robot.nqh in this example):

```
#include "robot.nqh"
```

Along with the different filename extension, an important difference you should note between the C and NQC code is that NQC uses *double quotes* (") around the filename. C uses *angle brackets* (< and >). Official NQC documentation states that enclosing a filename in angle brackets is forbidden.

So, why use these files called headers? If you are using a chassis and making several robots out of it, those robots are likely to have something in common. Instead of needing to add the same basic functions in each of the programs for those robots, we can put all those functions in one .nqh file, and then include them in whatever program we want with just one line of code! To execute those functions, all we need to do is call their name within the program as is ordinarily done. This whole process makes our life easier and our programs much cleaner.

> **NOTE** *Including more than one header in a NQC program is legal, but it isn't always practical. If you were making a header for a specific type of robot, it would be a good idea to put all the code in one header instead of multiple ones. However, if the set of commands you're using are distinct and separate, including more than one header in a NQC program is fine.*

Now let's look at the NQC header we'll use for our Brain-Bot chassis, shown in Listing 6-1. As you can see, it has some basic actions and also some constants, which can be included and used in an NQC program.

Listing 6-1. Brain-Bot.nqh

```
/* Brain-Bot.nqh - A file holding important instructions
* for the Brain-Bot chassis. To be included in any Brain-Bot
* programs */

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

// constants
#define Change 25
#define Position 200
#define Straight 300

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

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

// switch gears to fast speed
void SwitchF()
{
   On(Left+Right);
   OnRev(Switch);
   Wait(Change);
   Off(Switch);
}
```

```
// Switch gears to slow speed
void SwitchS()
{
   On(Left+Right);
   OnFwd(Switch);
   Wait(Change);
   Off(Switch);
}

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

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

// stop
void Stop()
{
   Off(Left+Right);
}
```

> **NOTE** *You will notice that there is no main task in this program (or actually header). That is correct. In headers, there are no main tasks. If you try to compile this header, you will get, "Error: task 'main' not defined." This is okay. BricxCC can't quite tell the difference between an .nqh file and an .nqc file, which does need a main task. Just ignore this error message, or even better, don't compile the header at all.*

Now that the header is set up, the next logical step is to create a full-blown NQC program and include this header in that program. As with the Zip-Bam-Bot chassis program in Chapter 3, we are not going to create a "real" sumo-bot program—just think of this as a test drive. We want to try out Brain-Bot's different features and get a feel for how it works. For this test drive, I wrote a relatively simple program that moves the robot around for a while and then stops. The program is called Brain-Bot-One.nqc and is shown in Listing 6-2.

> **NOTE** *Make sure that your headers are always in the same directory as the program that includes them. For instance, if you have an NQC program in C:\Robot that includes a header, the header must also be in C:\Robot. Otherwise, the compiling or downloading procedure will produce an error message.*

Listing 6-2. Brain-Bot-One.nqc

```
// Brain-Bot-One.nqc - a program for the Brain-Bot chassis.

// let’s include the header for this program
#include "Brain-Bot.nqh"

task main()
{
  SetPower(Switch,3);   // set power for switch motors to 3

  // let’s do this twice
  repeat(2)
  {
  Forward();
  Wait(Straight);
  TurnR();
  PlaySound(SOUND_CLICK);
  }

  // and this twice
  repeat(2)
  {
  Reverse();
  Wait(Straight);
  TurnL();
  PlaySound(SOUND_CLICK);
  }

  SwitchS();     // let’s try out the switching mechanisms

  // we'll do this twice in slow mode
  repeat(2)
  {
  Wait(Straight);
  TurnR();
  PlaySound(SOUND_UP);
  }

  SwitchF();      // switch back to fast

  // then do this twice
  repeat(2)
```

```
  {
  Wait(Straight);
  TurnL();
  PlaySound(SOUND_UP);
  }

  Stop();       // we're done, turn off motors
  PlaySound(SOUND_DOWN);   // play a sound so we know we're done
}
```

A Quick Test of Brain-Bot-One.nqc

Just as in Chapter 3, we are going to do some quick tests of each of the programs, and then do more thorough testing later. Download Brain-Bot-One.nqc to program slot 1 on the RCX using BricxCC, following the procedure outlined in Chapter 3, in the “Downloading Programs to the RCX” section. Then press the Run button on the RCX. Brain-Bot should begin the program by moving forward. After waiting a specified amount of time, Brain-Bot should turn right, play a sound, and then repeat that set of actions one more time.

> **CAUTION** *Brain-Bot should* always *be on fast speed when starting a program. The fast speed is the basic, or usual, driving speed, and the slow speed is the secondary driving speed. When you start the program, it assumes that the fast speed is the current one, and Brain-Bot’s program will eventually turn it to slow speed. But if it’s already on slow speed, its switching motors won’t be able to move! This won’t burn out your motors, but it is most assuredly not desirable.*

Now Brain-Bot should do something else: instead of going forwards, it should go backwards for a certain amount of time, and then turn left. This set of actions should be repeated twice as well.

At this point, Brain-Bot should switch gears to slow speed. Then the program should make Brain-Bot do another set of actions in slow speed, switch back to fast speed, do another set of actions, and finally stop, sounding the end of the program with `SOUND_DOWN`.

Understanding Brain-Bot-One.nqc

Brain-Bot-One.nqc doesn’t contain anything outstanding or earth-shattering, but it does have two things you haven’t seen in previous programs. One is the `#include` command, which as discussed earlier, includes header files:

```
#include "Brain-Bot.nqh"
```

The other new code is the repeat() statement. As you can guess, it's used to repeat something. It takes one argument, an *expression*, and repeats anything within its braces that number of times.

```
repeat(2)
{
Wait(Straight);
TurnL();
PlaySound(SOUND_UP);
}
```

The number 2 is the expression, and the repeat() statement will execute anything within its braces two times. It's pretty easy to use repeat()!

The rest of the program doesn't require much explanation. The individual commands included from Brain-Bot.nqh are put to good use (like calling SwitchS();), there are a few sounds played here and there to confirm when something is finished, and several Wait(); commands are used to have the robot move or do an action for a certain period of time.

Creating a More Complex Header-Based Program for Brain-Bot

The previous example just executed little chunks of code a repeated number of times. Couldn't we do something a bit more interesting? Of course! Because Brain-Bot doesn't have any sensors right now, our programming is somewhat limited, but there is still a lot more we can do. Although this next example doesn't boast of being super-complex, it executes commands based on the values of a timer. Therefore, the sumo-bot depends on an external source—time. This should definitely be more interesting, especially since you can change the waiting time periods, resulting in infinite possibilities.

Listing 6-3 shows Brain-Bot-Two.nqc. This program makes use of timers, until, and a special command—introduced with the RCX 2.0 firmware and parallel NQC 2.0 release—that controls the LCD: SetUserDisplay().

Listing 6-3. Brain-Bot-Two.nqc

```
// Brain-Bot-Two.nqc - a program for the Brain-Bot chassis

#include "Brain-Bot.nqh"

task main()
{
  SetUserDisplay(Timer(0),0);   // let's set the RCX's LCD to Timer 0

  SetPower(Switch,3);     // set power for switch motors to 3

  ClearTimer(0);   // clear the timer
```

```
  Forward();       // go forward

  until(Timer(0) >= 25);  // until timer is >= 2.5 seconds
  PlaySound(SOUND_CLICK);
  ClearTimer(0);           // clear the timer

  SwitchS();      // switch to slow mode
  Reverse();      // then reverse

  until(Timer(0) >= 35);   // until timer is >= 3.5 seconds
  PlaySound(SOUND_CLICK);

  TurnR();               // turn right
  Forward();             // go forward
  SwitchF();             // switch back to fast
  TurnL();               // now turn left

  ClearTimer(0);         // then and only then clear the timer

  Reverse();             // now put in reverse

  until(Timer(0) >= 45);   // until timer is >= 4.5 seconds

  Stop();   // stop!!!

  PlaySound(SOUND_DOUBLE_BEEP); // we're done
}
```

A Quick Test of Brain-Bot-Two.nqc

Once again, download the program to the RCX. You can place it in slot 1 (which erases any previous program in slot 1), slot 2, or any other slot. Position Brain-Bot somewhere with a little moving space, make sure it is on fast mode, and press the Run button on the RCX.

The first thing you should notice is that the display on the RCX changes from its normal state (system clock) to something different: a timer. This is not just any old timer, but the timer *we are using*; that is, `Timer(0)`. We set this up at the beginning of the program to be able to visually observe when the robot makes its decisions (the program clears the timer about the time a set of different commands executes).

After 2.5 seconds, Brain-Bot should play a sound, clear the timer, change to slow speed, and go in reverse. After reversing for 3.5 seconds, it should play another sound, turn right, put the motors in the forward direction, switch back to fast, and *then* turn left. Once that burst of activity is finished, Brain-Bot will once again clear the timer in use, go in reverse for 4.5 seconds, and finally stop, playing `DOUBLE_BEEP` to let you know it's finished.

Understanding Brain-Bot-Two.nqc

As noted before Listing 6-3, this program uses the `SetUserDisplay()` command to control the LCD:

```
SetUserDisplay(Timer(0),0);
```

The format for using this command is `SetUserDisplay(`*`value, precision`*`)`. *Value* is the source to be displayed. This could be a constant (for example, 54), a variable, a timer, a sensor, and even the message buffer. *Precision* deals with a decimal point. The number entered here determines the position of the decimal point. If we have 1 for precision, the decimal point would move one space to the left (starting from the right). A precision of 0, as in our example, means that there won't be a decimal point displayed.

What's the purpose of using this in our case? There isn't any particular reason really, but it does show the potential usefulness of `SetUserDisplay()`. When you are using something frequently in a program, like a timer, it can be extremely helpful to actually monitor its progress. With `SetUserDisplay()`, you can determine where bugs are in your programs and do many other useful and interesting things. The program here shows just one example of controlling the LCD.

> **TIP** *To find out more about setting the LCD, look in the* NQC Programmer's Guide, *which comes with BricxCC. For more thorough coverage, check out* Extreme MINDSTORMS: An Advanced Guide to LEGO MINDSTORMS, *by Dave Baum et al (Apress, 2000). This book also covers other features of the RCX 2.0 firmware.*

In the rest of the program, the sumo-bot executes an action or series of actions until the allotted amount of time has run out. It is done with a line of code like this:

```
until(Timer(0) >= 45);
```

The program then clears the timer and executes another set of actions until another predefined period of time runs out. There is one exception in this program where it runs some actions without worrying about time, but the main part of the structure operates this way.

> **TIP** *Make your own program for Brain-Bot and see what you can come up with. String together a few commands from the header file and add some statements such as* `repeat` *and* `until` *(or both) to control the structure of the program. Also see if you can figure out how to use* `Random()` *to give Brain-Bot some unpredictability.*

Testing the Brain-Bot Chassis

Now that you have seen these programs, let's retest them, but in some "sumo-like" situations. First, make sure that Brain-Bot-One.nqc is in program slot 1 on the RCX and Brain-Bot-Two.nqc is in program slot 2. Place Brain-Bot in your arena and put some relatively small and heavy objects—as you did in Chapter 3—in the general area where Brain-Bot will be going.

> **NOTE** *Remember that you don't need to retype the programs in this book into BricxCC. All of the programs for this book are available from the Downloads section of the Apress web site* (`www.apress.com`).

Press the Run button on the RCX, for Brain-Bot-One.nqc, and watch carefully. How does Brain-Bot manage hitting an object on fast speed? How about on slow speed? What about the same object on different speeds? Set up different situations to see the different reactions.

Now change the program to Brain-Bot-Two.nqc. Run the program and, once again, watch the reactions with the objects. This time, however, let's make things a little more interesting. Grab an object and position it in front of Brain-Bot, without letting go. Use your hands to increase or decrease the resistance of the object once Brain-Bot has run into it. Try holding an object somewhat firmly in front of Brain-Bot when it is on slow speed—the reaction you see is similar to Brain-Bot shoving another sumo-bot.

Now try this: while Brain-Bot is running along—whether in slow or fast mode—place your finger, or even hand, in front of one of the tires. What happens? Brain-Bot will go right over it! But it's not the fact that Brain-Bot can go over your finger or hand that is interesting; it is *why* Brain-Bot can go over it that is interesting. It's because of the type of wheels—believe me, big wheels are a good thing!

> **CAUTION** *The switch subassemblies can slightly separate the wedge wheels on the drive subassemblies over time; this could possibly lead to problems when attempting to change the speed. Therefore, you'll need to properly reposition the wedge wheels if they get out place. To do this, put the switching axle in fast speed with the outer wedge wheel touching the 1x15 blue, straight liftarm (as it should always be on fast speed), and push the other wedge wheel towards it until there are no more gaps between the switching motor's axle pin and the wedge wheels. You don't need to check to see if the wedge wheels have come apart every time you start up Brain-Bot, but be sure to check this occasionally.*

Conclusion

Brain-Bot shows you just how different things can be when switching from one strategy to another. Right from the start in the M-class strategy we are getting into some complex sumo-bots—and this is just the chassis! However, this chassis is so full-featured it could almost be called a sumo-bot in itself; almost, but not quite. It's missing something quite vital: sensors. In the next chapter, you will meet ZR2. This M-class sumo-bot adds more fun and interesting subassemblies, quite a few sensors, and utilizes the features of this chassis and some nice programming to become a formidable foe.

However, before continuing, keep in mind that the examples in this book are not intended to be a comprehensive representation of the M-class strategy, because there are just too many possibilities (it would be like trying to make a book conclusive on all the things you can make with LEGO). The purpose of the examples in Part Three of this book is to generalize the M-class strategy, and to help you to understand and appreciate this strategy. Because of the way Brain-Bot is designed, it is limited in one way or another. In other words, Brain-Bot can't be everything!

But Brain-Bot *can* show you many M-class tricks, substrategies, mechanisms, and more. Learn from these examples; adapt the mechanisms into your own designs; take the information, tips, and tricks presented and store them in your brain; and most of all, remember to have fun while you're doing it!