# Mine Sweeper

## Building the Mine Sweeper

You can build the robot with all the parts from the NXT retail set. After the robot building instructions, you'll see how to prepare four mines of the correct size and color for your robot,

using the few retail set parts left. You might also want to prepare some other suitable object to be collected, by using some black 2 - 4 bricks that you might have among your LEGO spares. You can make a mine from two piled bricks. After completing the grabber subassembly (Step 70), you can test the arm mechanism by rotating the bevel gear by hand.

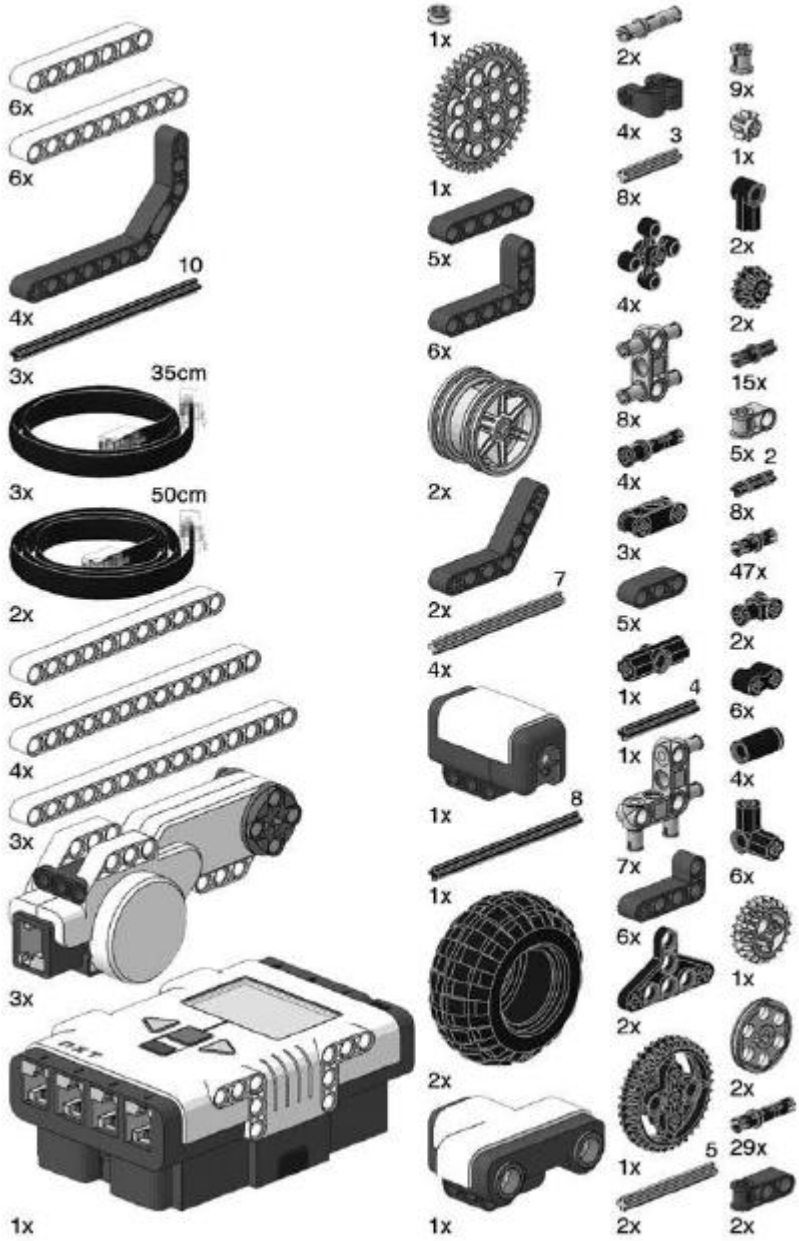



**Figure 7-8.** *Mine Sweeper bill of materials*

**Table 7-2.** *Mine Sweeper Bill of Materials*

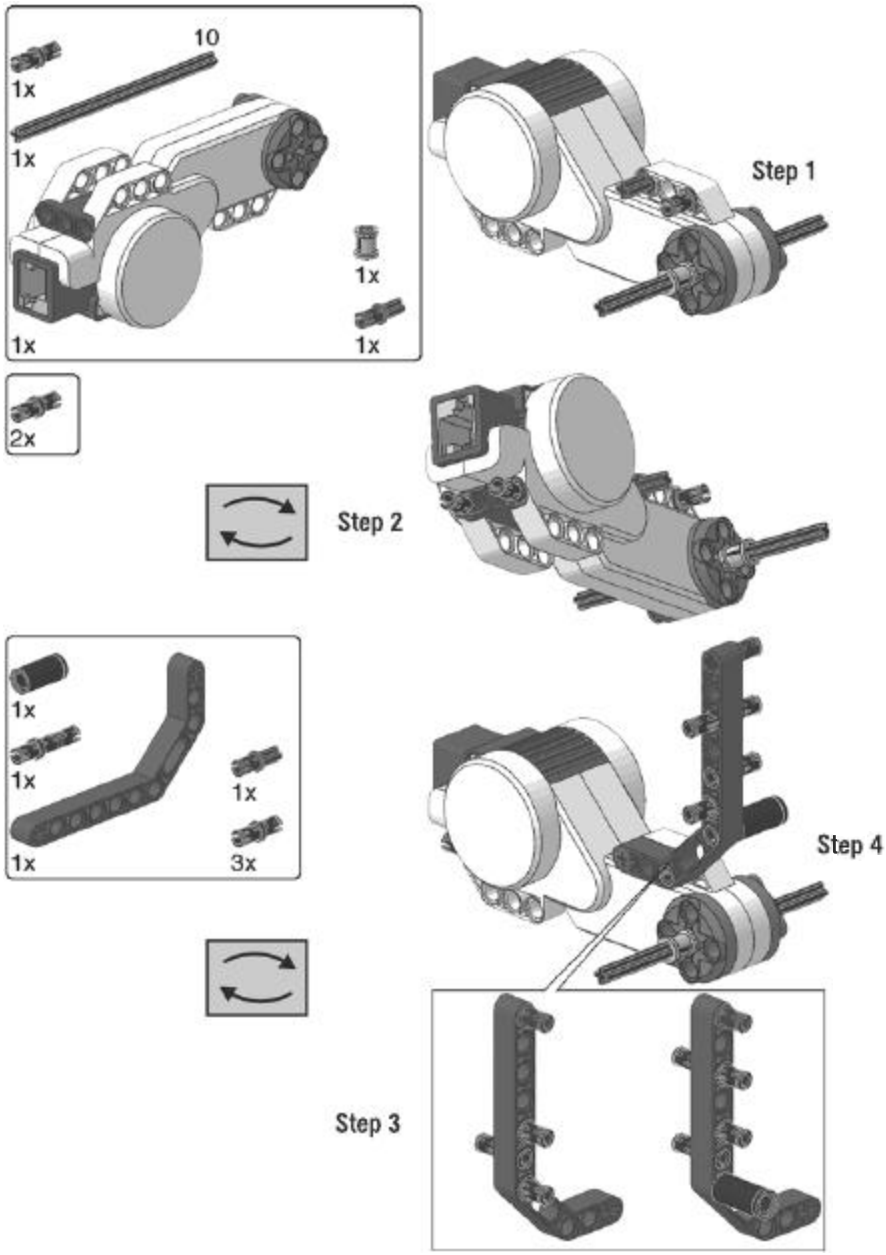
| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 6 | White | 32524.DAT | TECHNIC Beam 7 |
| 6 | White | 40490.DAT | TECHNIC Beam 9 |
| 4 | Dark gray | 32009.DAT | TECHNIC Beam 11.5 Liftarm Bent 45 Double |
| 3 | Black | 3737.DAT | TECHNIC Axle 10 |
| 3 | | 55805.DAT | Electric Cable NXT 35cm |
| 2 | | 55806.DAT | Electric Cable NXT 50cm |
| 6 | White | 32525.DAT | TECHNIC Beam 11 |
| 4 | White | 41239.DAT | TECHNIC Beam 13 |
| 3 | White | 32278.DAT | TECHNIC Beam 15 |
| 3 | | 53787.DAT | Electric MINDSTORMS NXT Motor |
| 1 | | 53788.DAT | Electric MINDSTORMS NXT |
| 1 | Light gray | 32123.DAT | TECHNIC Bush 1/2 Smooth |
| 1 | Light gray | 3649.DAT | TECHNIC Gear 40 Tooth |
| 5 | Dark gray | 32316.DAT | TECHNIC Beam 5 |
| 6 | Dark gray | 32526.DAT | TECHNIC Beam 7 Bent 90 (5:3) |
| 2 | Light gray | 54087.DAT | Wheel 43.2 - 22 Without Pinholes |
| 2 | Dark gray | 32348.DAT | TECHNIC Beam 7 Liftarm Bent (4:4) |
| 4 | Light gray | 44294.DAT | TECHNIC Axle 7 |
| 1 | White | 55969.DAT | Electric MINDSTORMS NXT Light Sensor |
| 1 | Black | 3707.DAT | TECHNIC Axle 8 |
| 2 | Black | 55976.DAT | Tire 56 - 26 Balloon |
| 1 | White | 53792.DAT | Electric MINDSTORMS NXT Ultrasonic Sensor |
| 2 | Light gray | X783.DAT | TECHNIC Pin Long |
| 4 | Dark gray | 41678.DAT | TECHNIC Axle Joiner Perpendicular Double Split |
| 8 | Light gray | 4519.DAT | TECHNIC Axle 3 |
| 4 | Black | 32072.DAT | TECHNIC Knob Wheel |
| 8 | Light gray | 48989.DAT | TECHNIC Axle Joiner Perpendicular 1-3-3 with 4 Pins |
| 4 | Black | 32054.DAT | TECHNIC Pin Long with Stop Bush |
| 3 | Black | 32184.DAT | TECHNIC Axle Joiner Perpendicular 3L |
| 5 | Dark gray | 32523.DAT | TECHNIC Beam 3 |
| 1 | Black | 32034.DAT | TECHNIC Angle Connector #2 |
| 1 | Black | 3705.DAT | TECHNIC Axle 4 |
| 7 | Light gray | 55615.DAT | TECHNIC Beam 5 Bent 90 (3:3) with 4 Pins |
| 6 | Dark gray | 32140.DAT | TECHNIC Beam 5 Liftarm Bent 90 (4:2) |

*Continued*

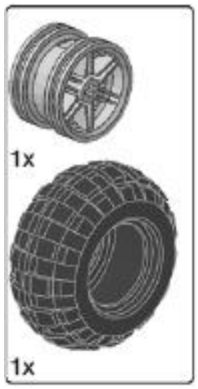
**Table 7-2.** *Continued*

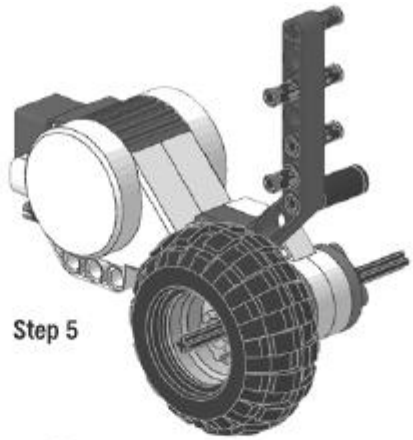
| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 2 | Black | 2905.DAT | TECHNIC Liftarm Triangle 5 - 3 - 0.5 |
| 1 | Black | X344.DAT | TECHNIC Gear 36 Tooth Double Bevel |
| 2 | Light gray | 32073.DAT | TECHNIC Axle 5 |
| 9 | Light gray | 3713.DAT | TECHNIC Bush |
| 1 | Light gray | 3647.DAT | TECHNIC Gear 8 Tooth |
| 2 | Black | 32013.DAT | TECHNIC Angle Connector #1 |
| 2 | Black | 32270.DAT | TECHNIC Gear 12 Tooth Double Bevel |
| 15 | Blue | 43093.DAT | TECHNIC Axle Pin with Friction |
| 5 | Light gray | 6536.DAT | TECHNIC Axle Joiner Perpendicular |
| 8 | Black | 32062.DAT | TECHNIC Axle 2 Notched |
| 47 | Black | 2780.DAT | TECHNIC Pin with Friction and Slots |
| 2 | Black | 32039.DAT | TECHNIC Connector with Axlehole |
| 6 | Black | 45590.DAT | TECHNIC Axle Joiner Double Flexible |
| 4 | Black | 75535.DAT | TECHNIC Pin Joiner Round |
| 6 | Black | 32014.DAT | TECHNIC Angle Connector #6 (90 degree) |
| 1 | Light gray | 32269.DAT | TECHNIC Gear 20 Tooth Double Bevel |
| 2 | Light gray | 4185.DAT | TECHNIC Wedge Belt Wheel |
| 29 | Black | 6558.DAT | TECHNIC Pin Long with Friction and Slot |
| 2 | Dark gray | 42003.DAT | TECHNIC Axle Joiner Perpendicular with 2 Holes |

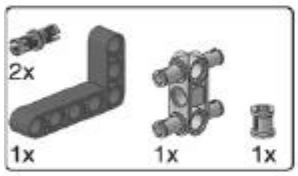
266 parts total (all included in NXT retail set)

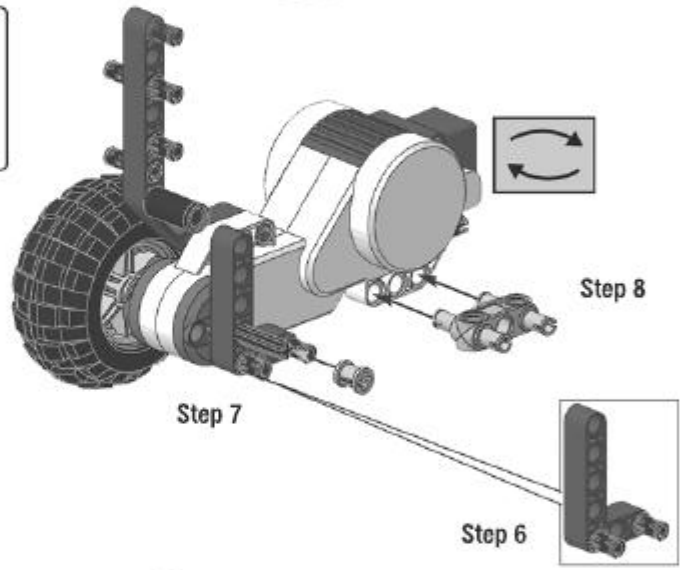


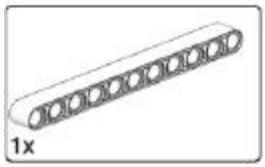

*Start building the right wheel assembly. In Step 2, insert two black pins in the motor's rear.*

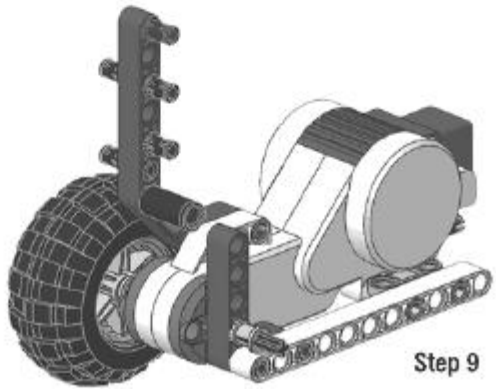


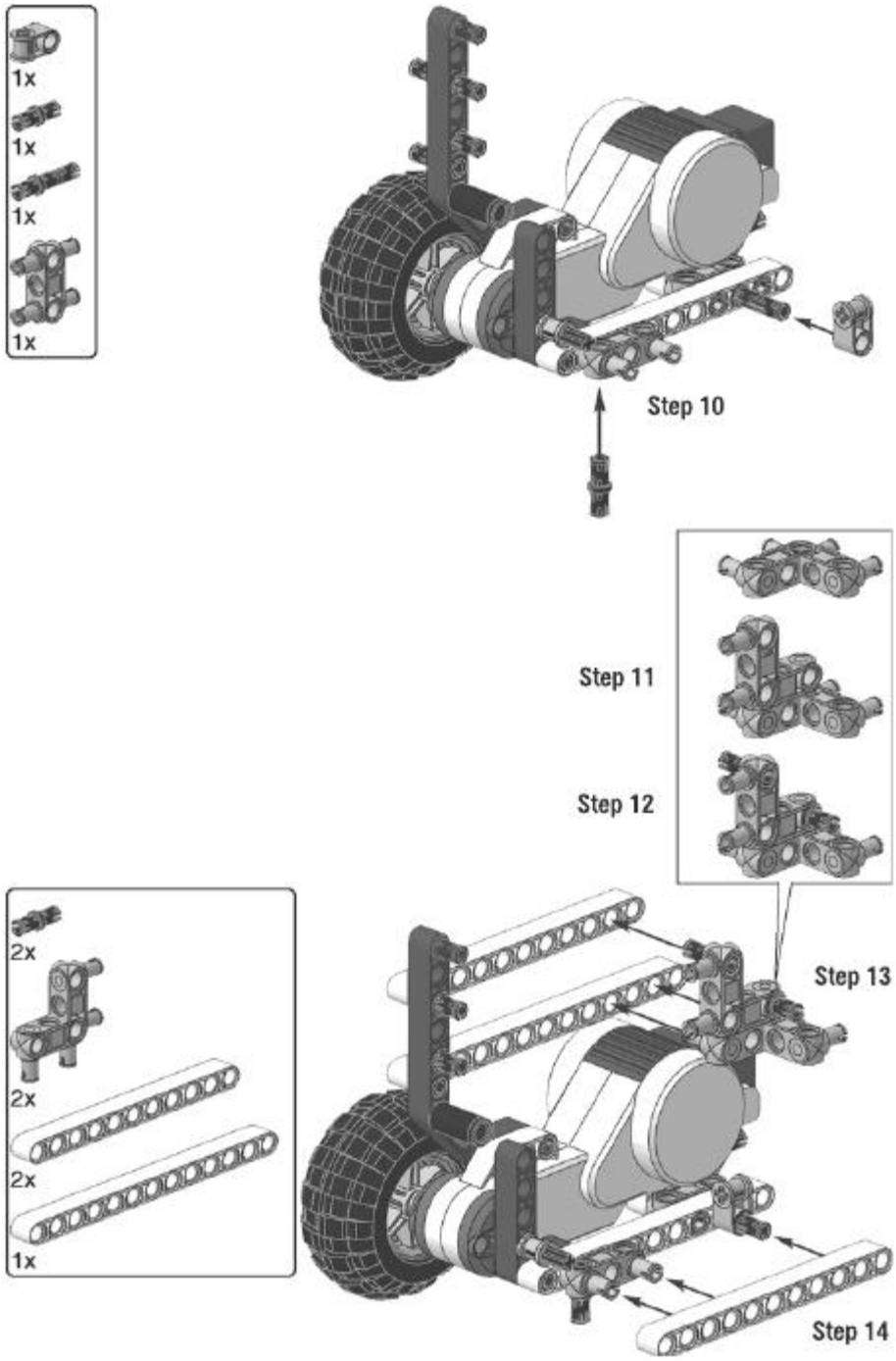



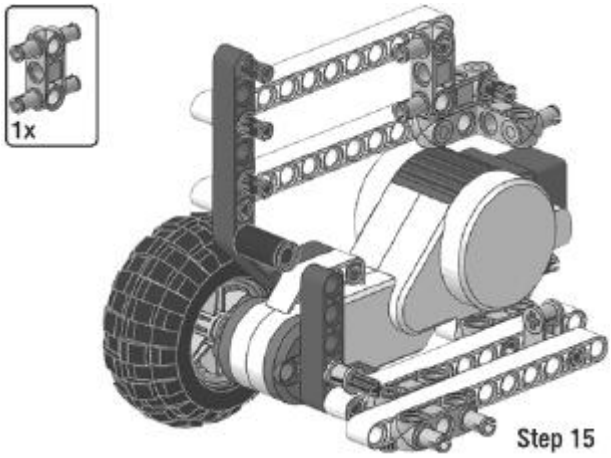



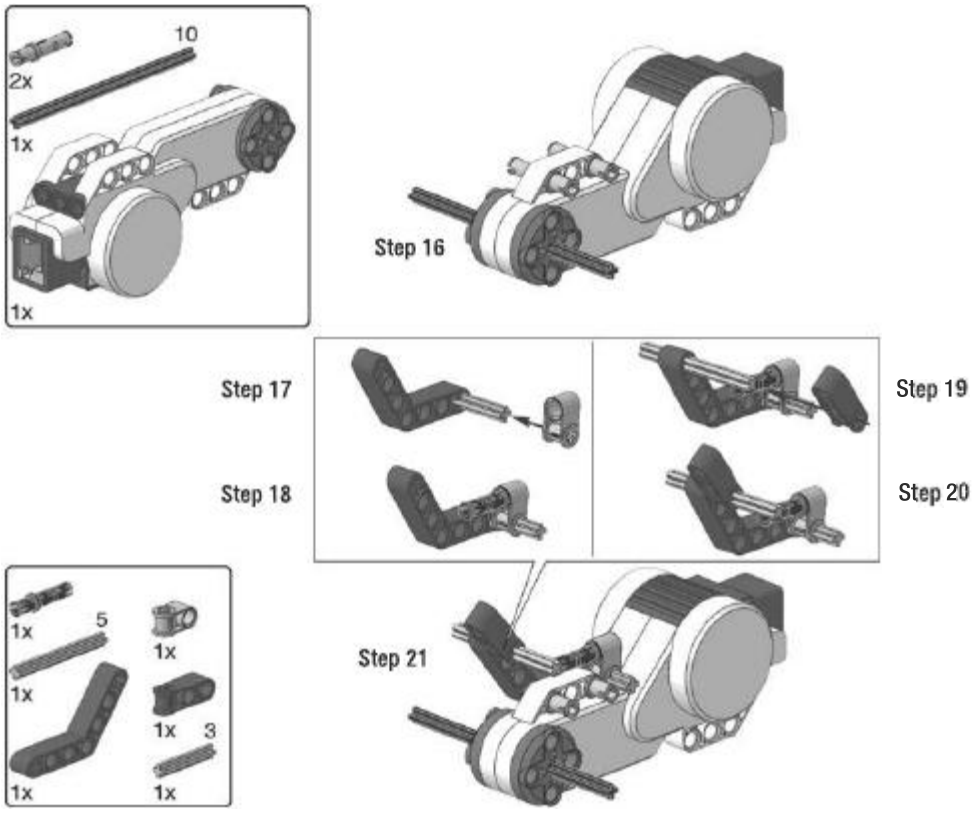

*Attach the wheel, then turn the model and insert the bent beam in the wheel axle, securing it with a bush. Finally, add an 11 holes–long beam.*



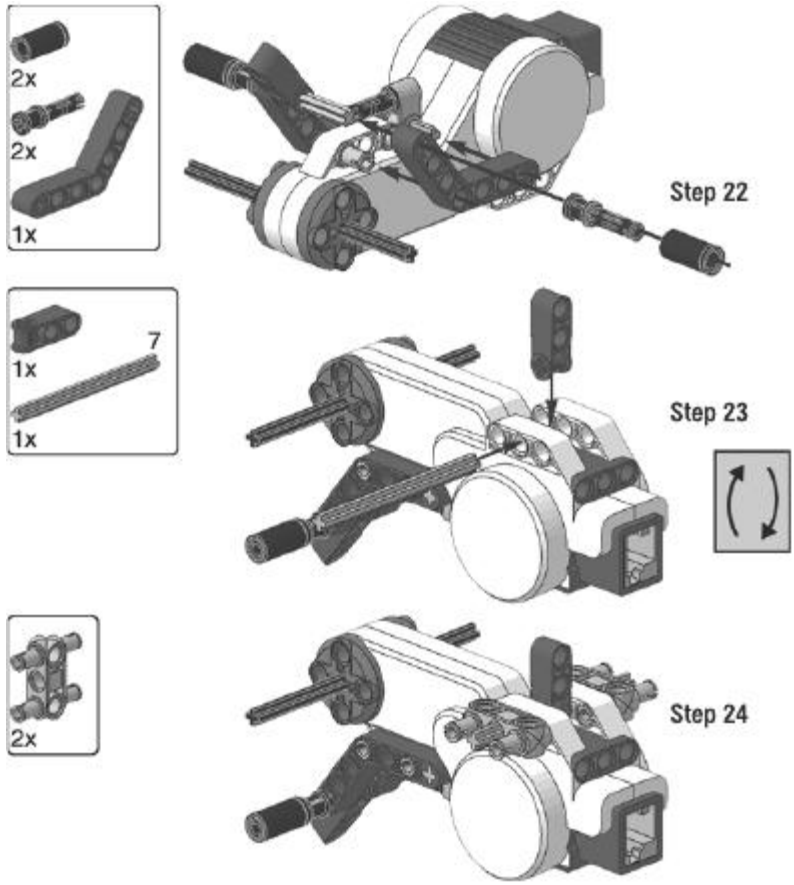

*Continue building the structure. In Step 14, place two 11 holes–long beams and a 13–long beam as shown.*



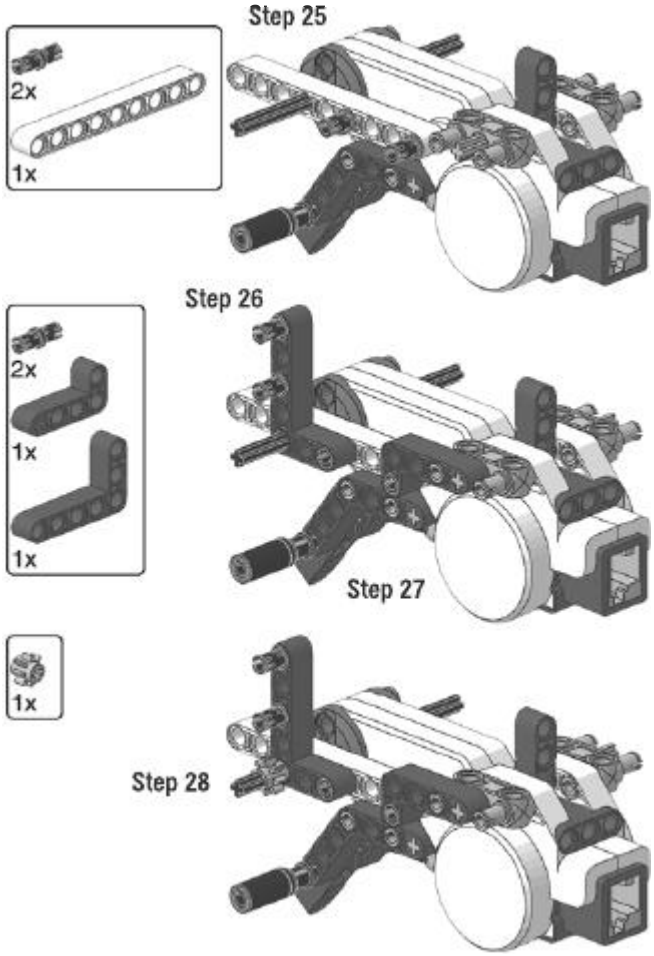

*The right wheel assembly is done.*



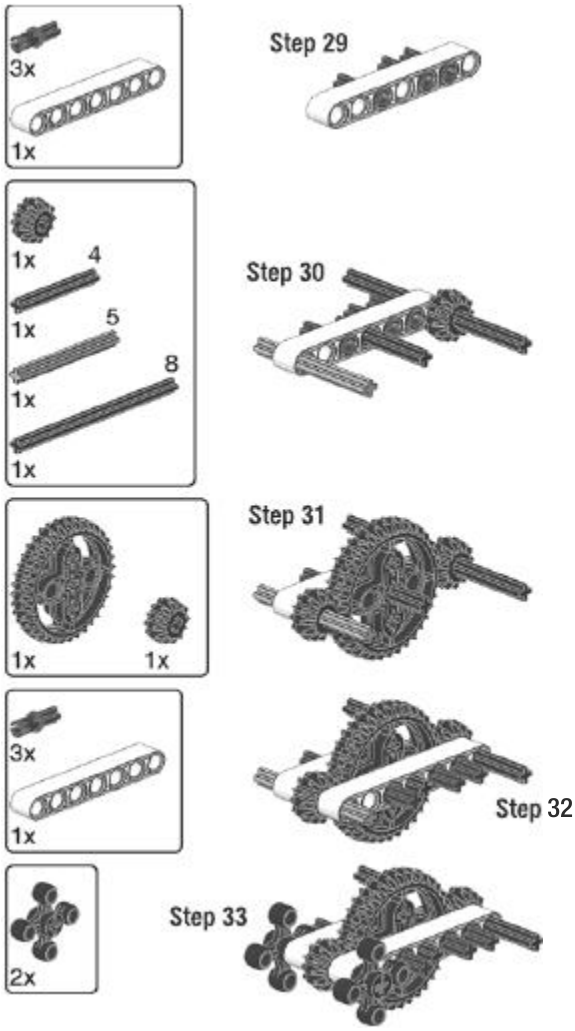

*Now you are building the grabber motor's assembly that forms the central part of the robot.*



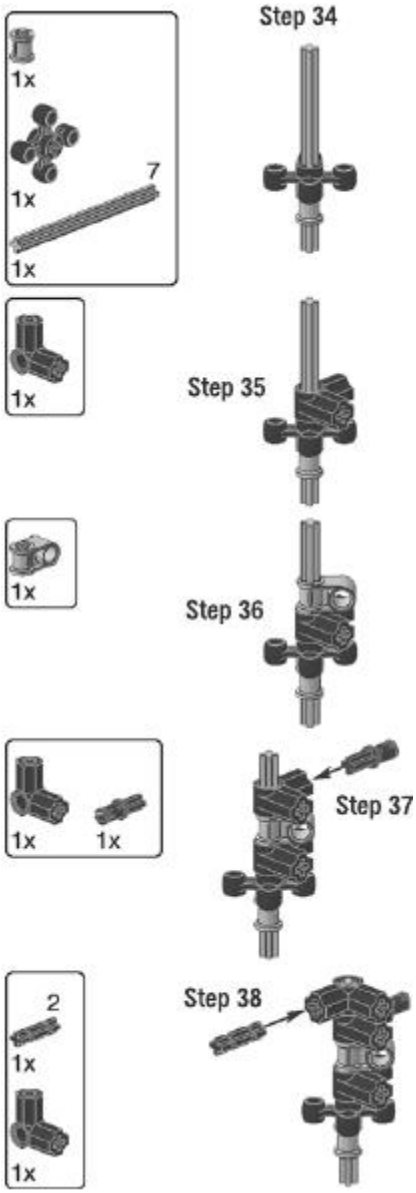

*The bent wedge on top of the motor helps the mines to slide down into the robot's hold, once the grabber arm has collected, raised, and released them.*



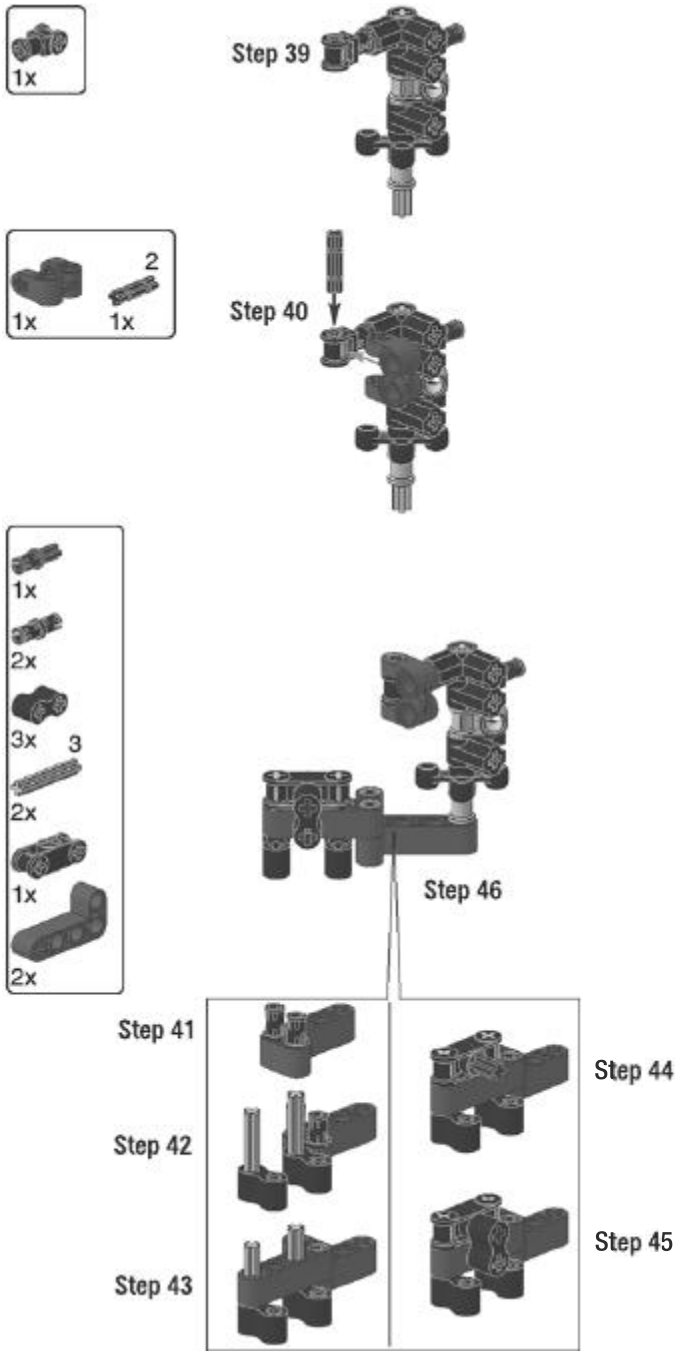

*The building goes on by placing a 9-long beam and two pins. Then add the bent beams to hold the white beam in place. Finally, add the 8-tooth gear.*



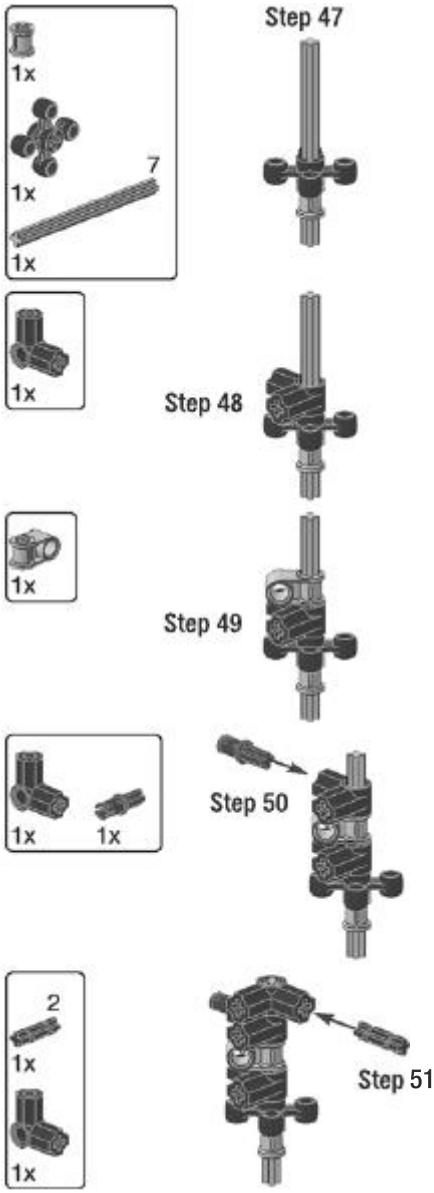

*Start building the grabber arm itself. In Step 29, place the blue axle pins in the 7-long beam as shown. Then add the gears and the other white beam. Lock the structure with two black knob wheels. They are used to trasmit movement at 90 degrees to the grabber fingers, as were gears with just 4 teeth.*



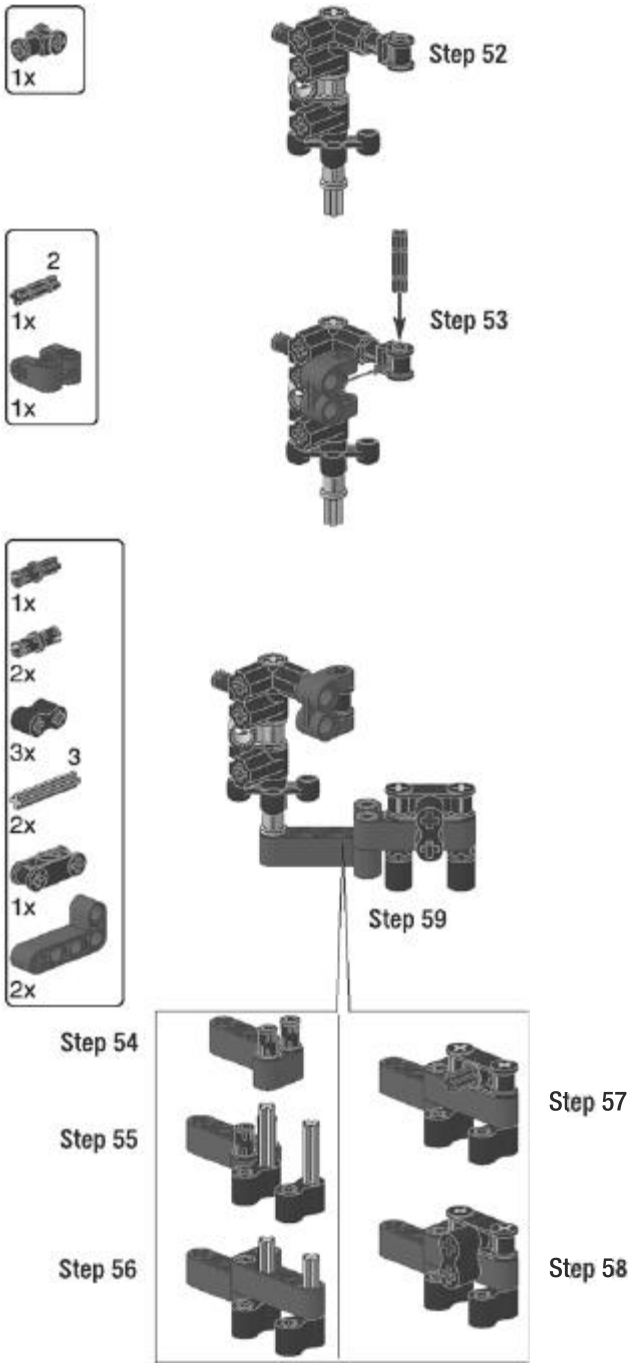

*Start building the right finger subassembly.*



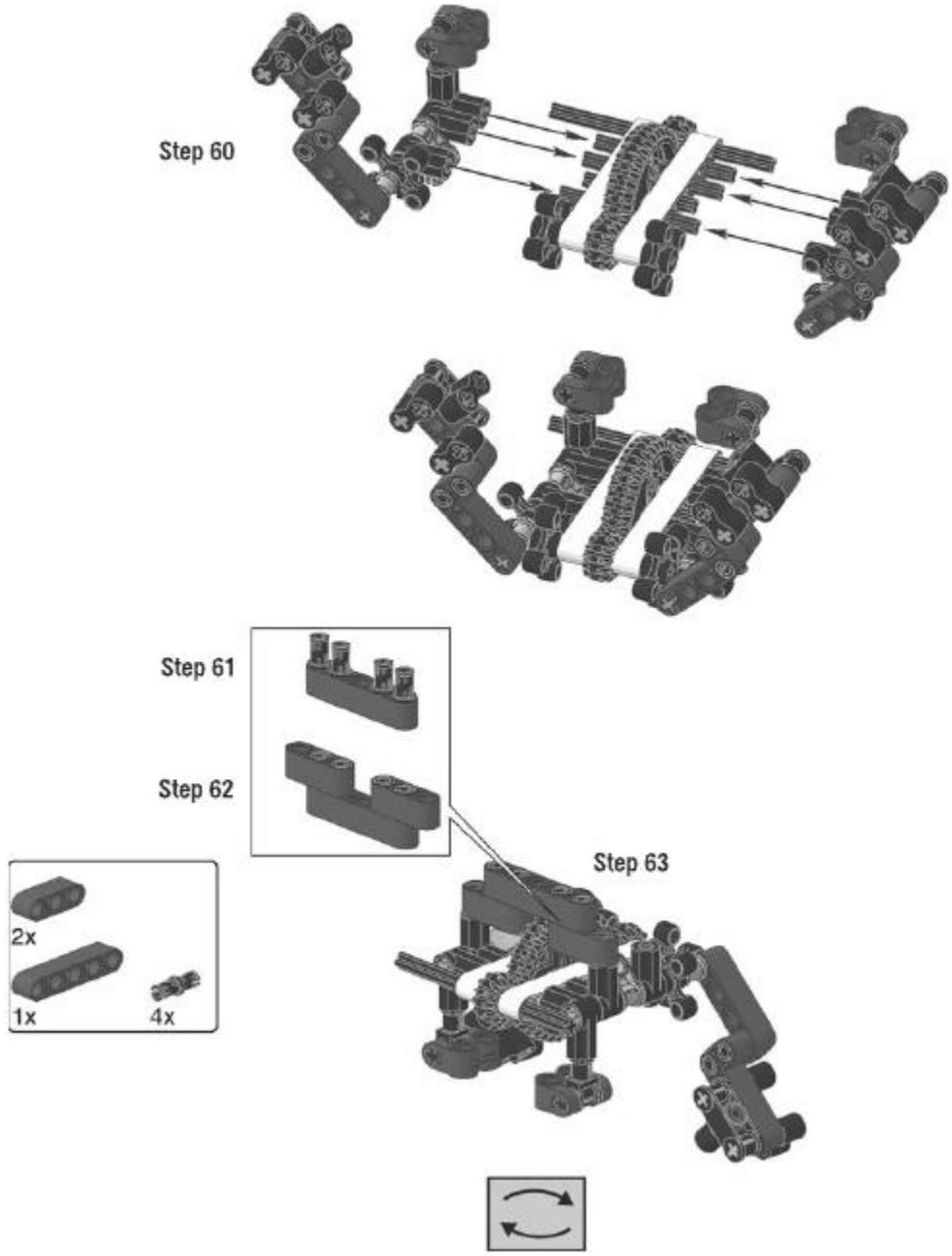

*The right finger subassembly is done.*



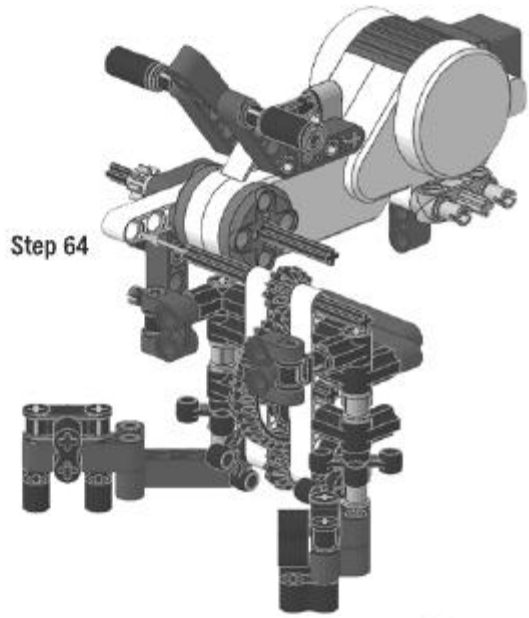

*Start building the left finger subassembly.*

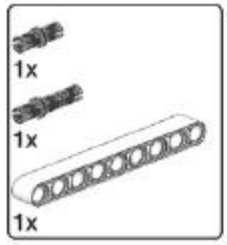


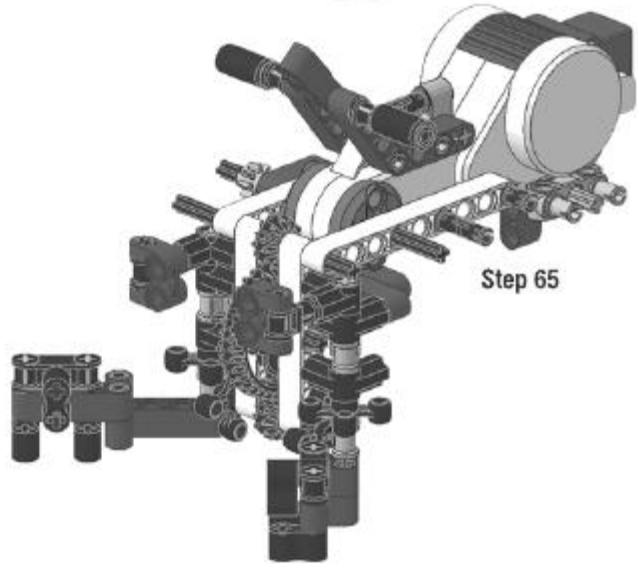

*The left finger subassembly is done.*



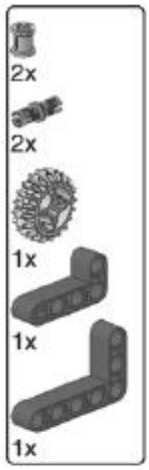

*Attach the fingers to the grabber arm, so that they open and close symmetrically. Turn the model and add a blocking beam. The grabber arm is done.*

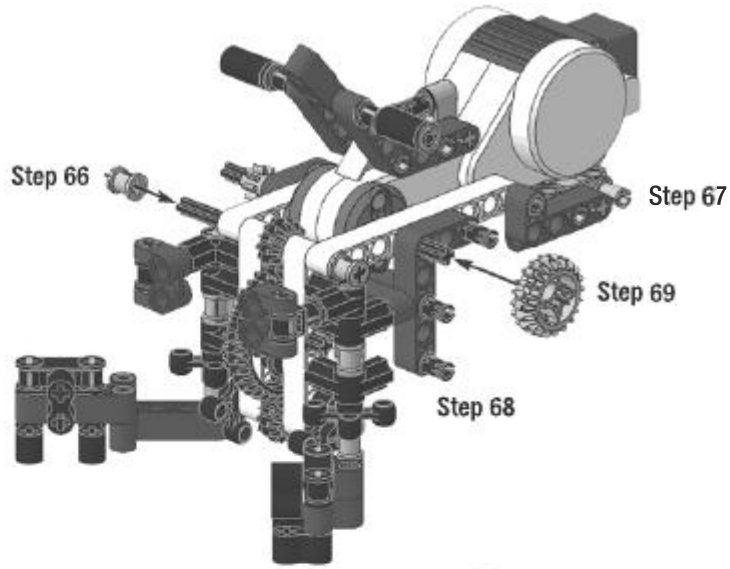


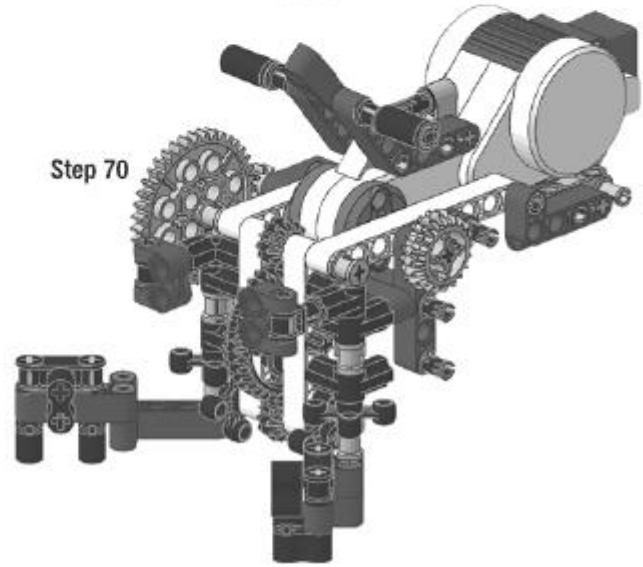




*Insert the grabber arm into its place in the grabber motor assembly. Add a 9-long beam, a black pin, and a long pin.*

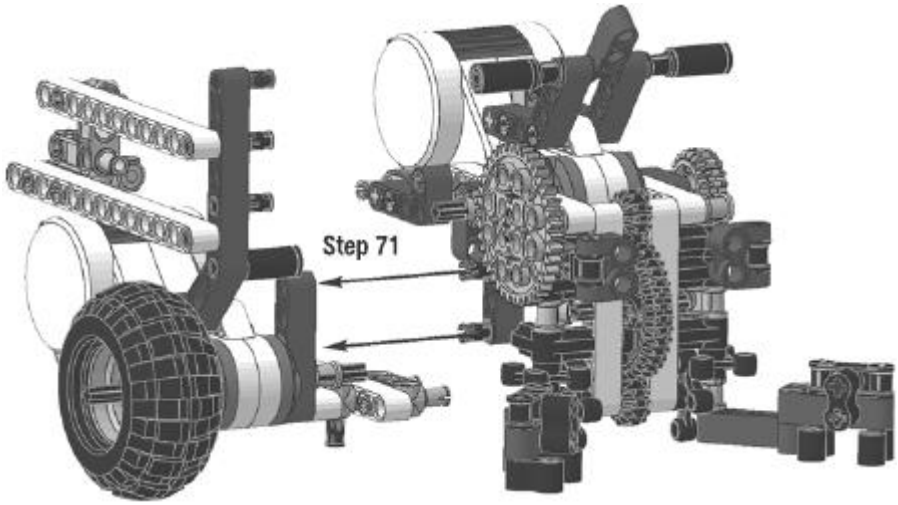


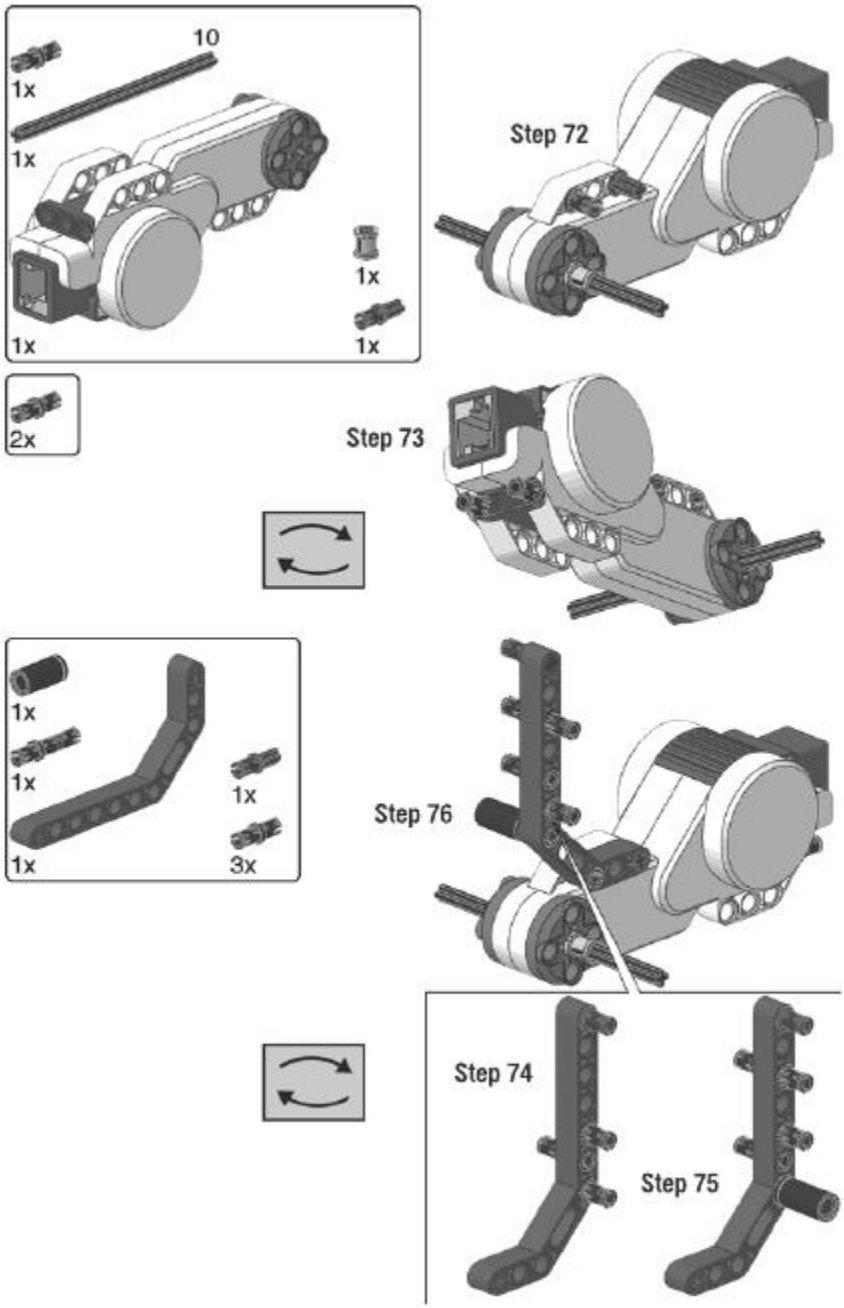




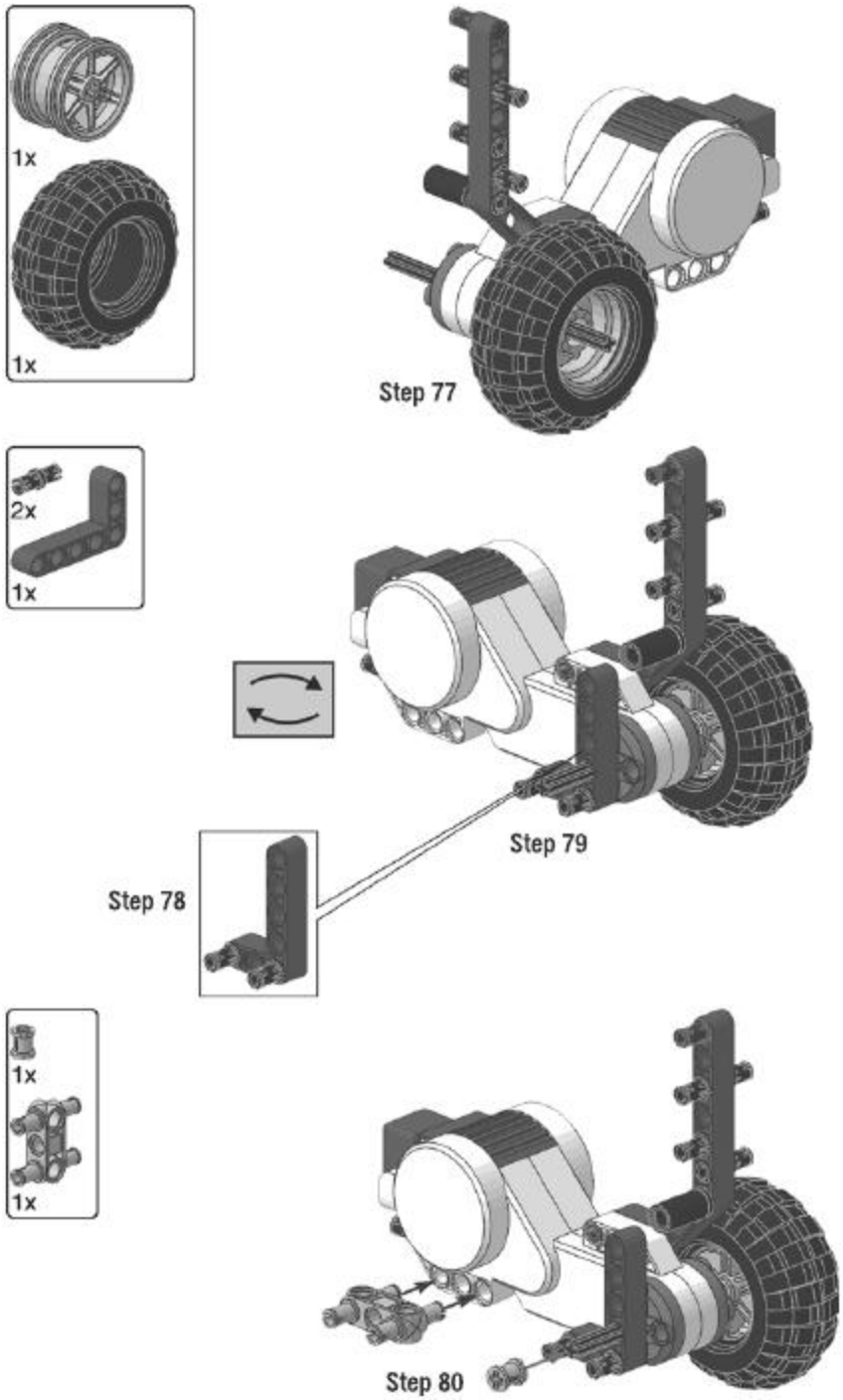

*Add two bent beams, then the 20-tooth gear and the bush. Attach two black pins to the longer bent liftarm. Add the 40-tooth gear, and the grabber assembly is finished. Now you can try the underactuated mechanism, turning the bevel gear by hand.*



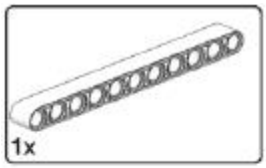

*Attach the grabber assembly to the right side of the robot.*

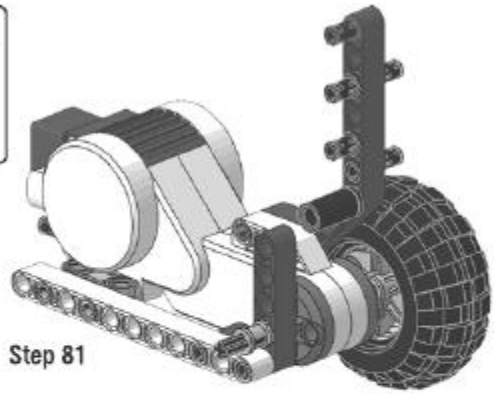


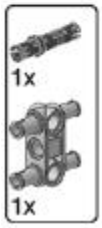

*Start building the left wheel assembly. In Step 73, insert two black pins in the motor's rear.*

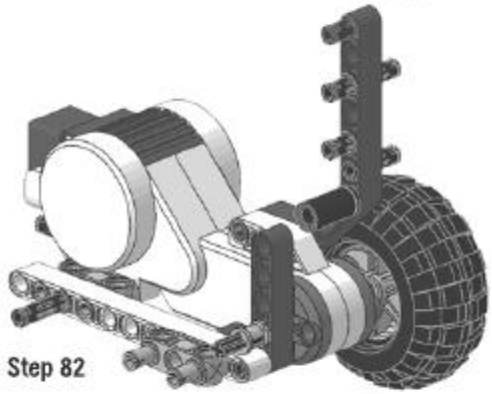



*Attach the wheel, then turn the model and insert the bent beam in the wheel axle, securing it with a bush.*

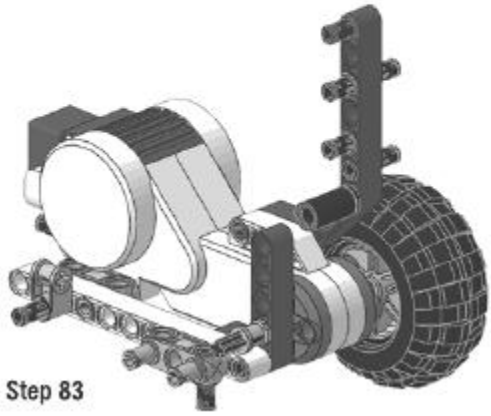


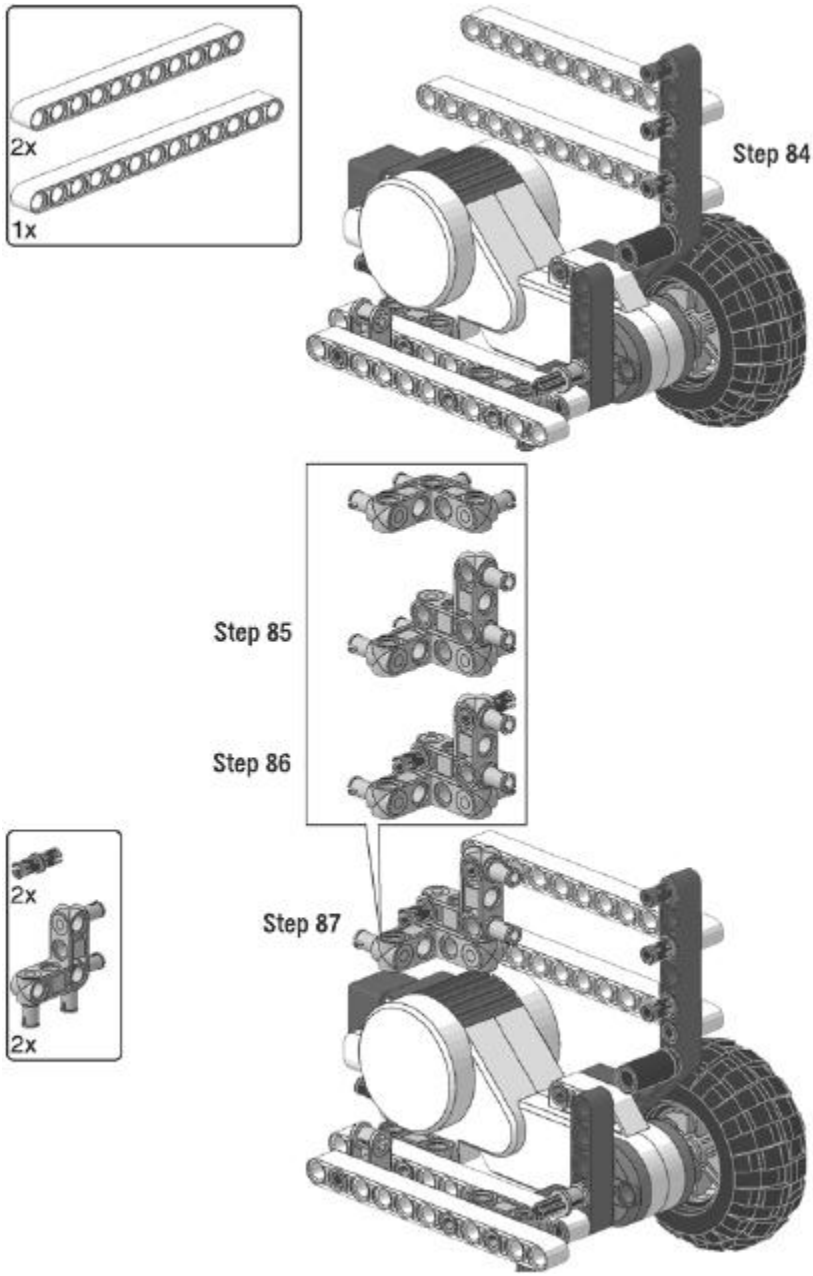



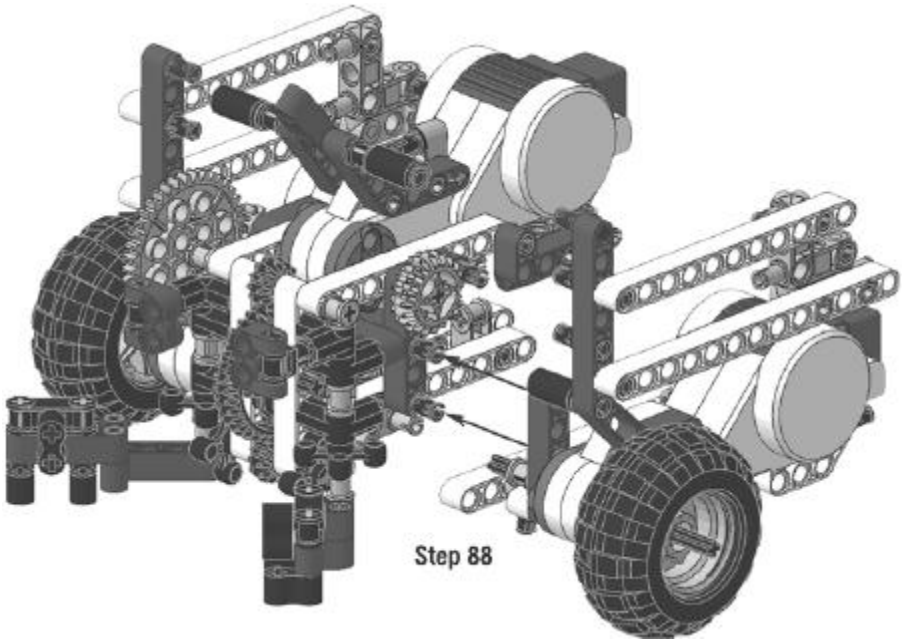




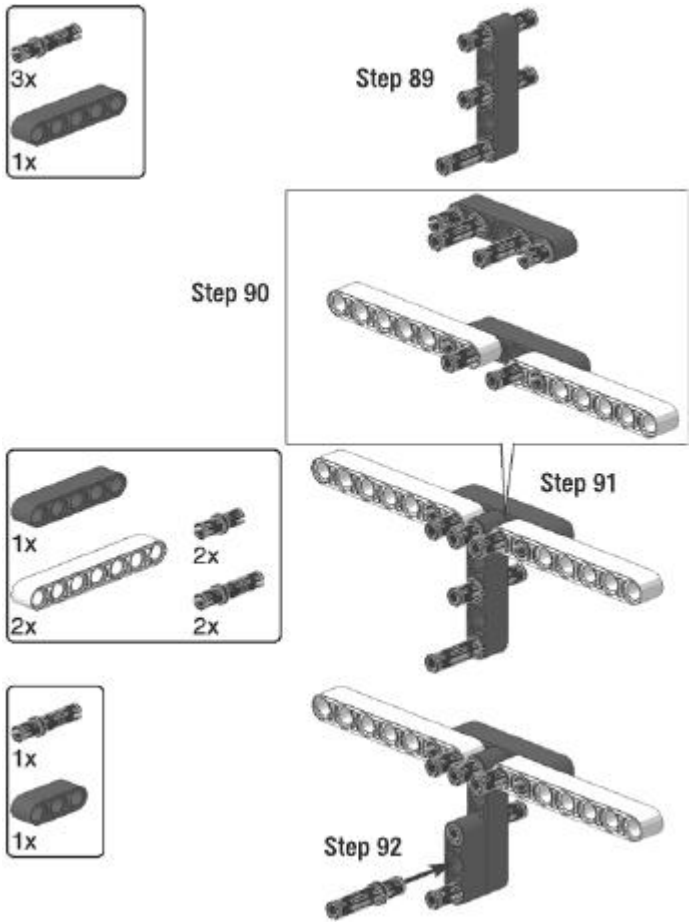

*Add an 11 holes–long beam in Step 81.*



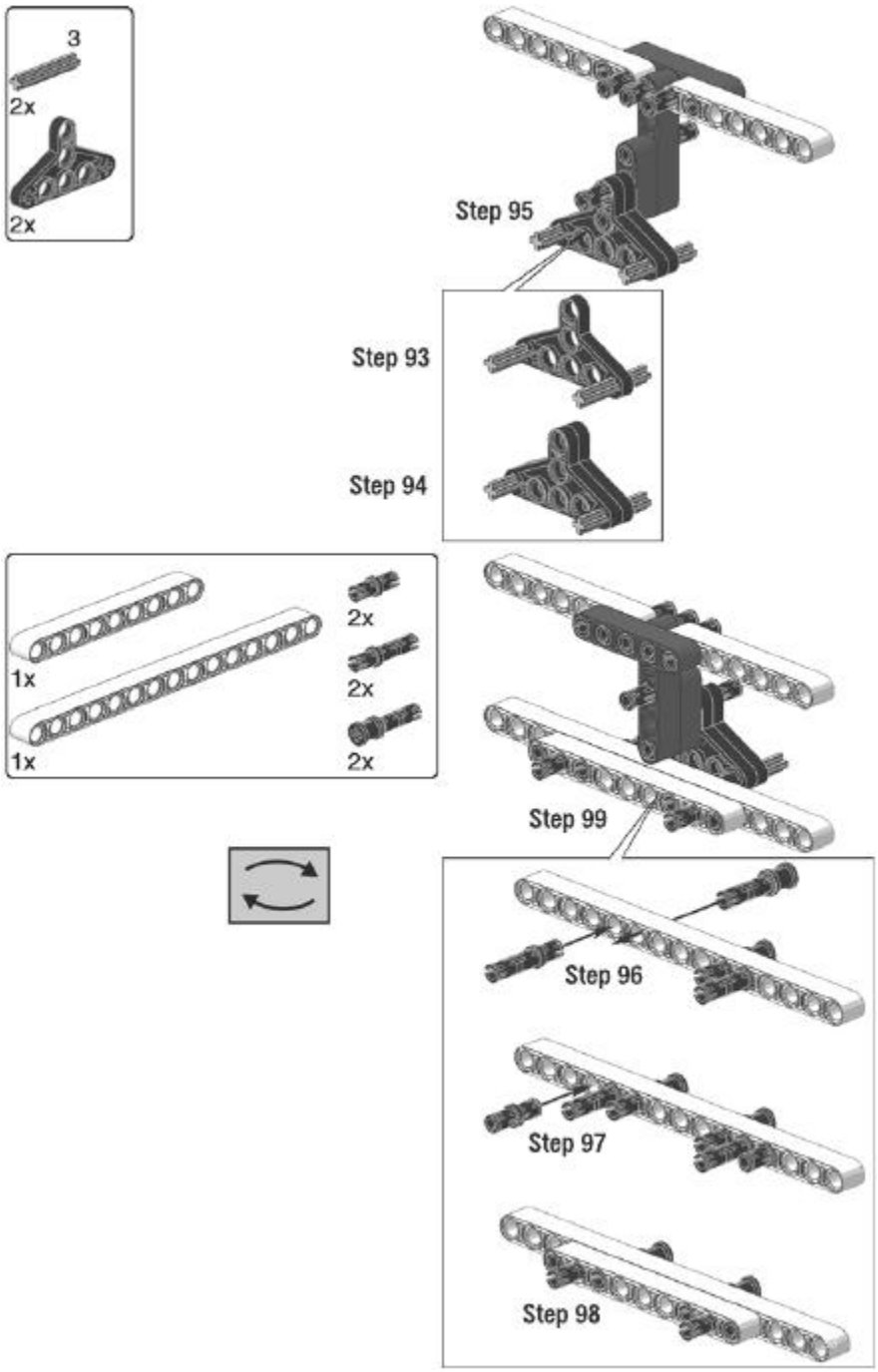

*In Step 84, place two 11 holes–long beams and a 13 holes–long beam as shown. The left wheel assembly is done.*



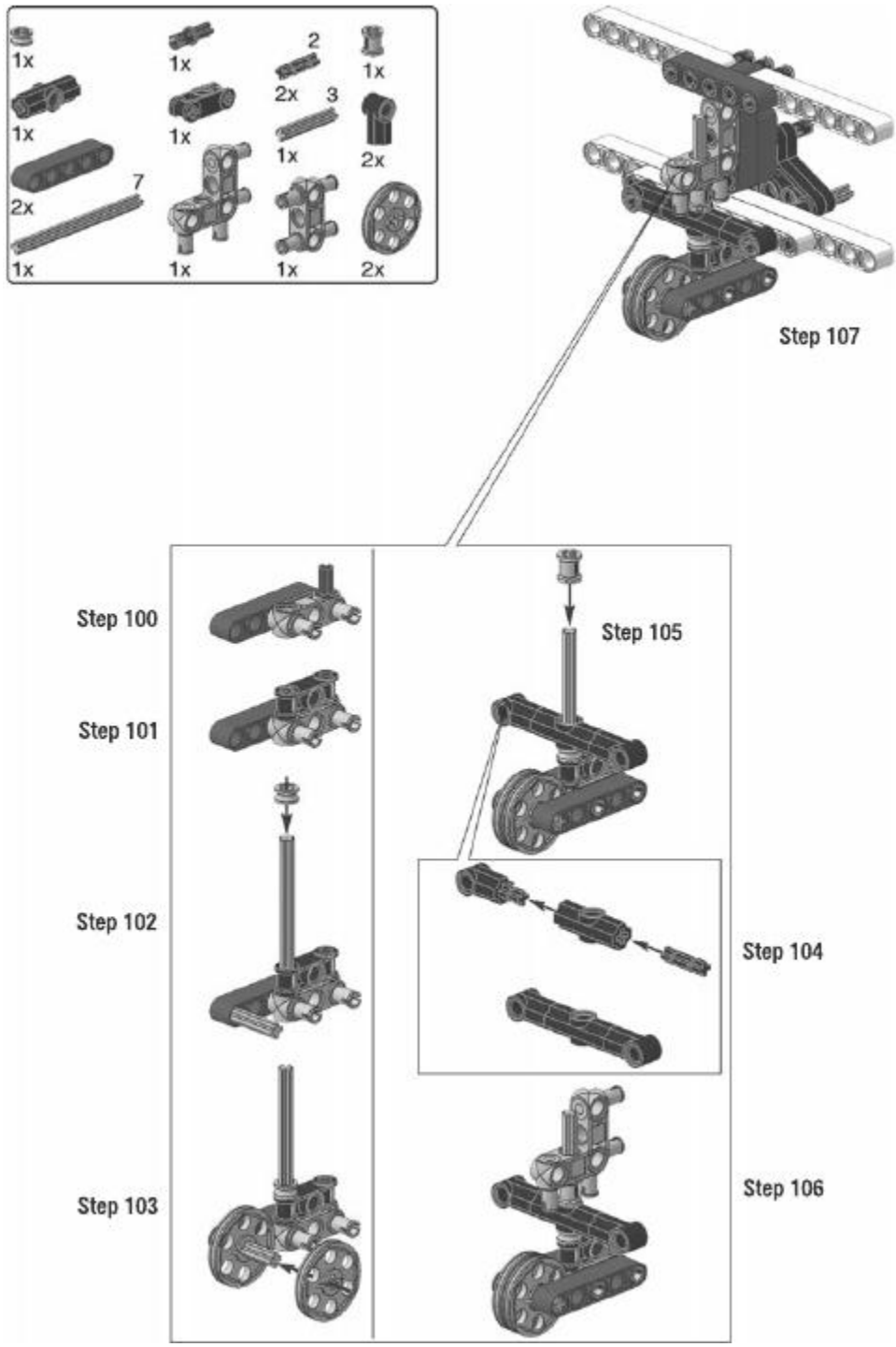

*Attach the left wheel assembly to the rest of the robot.*



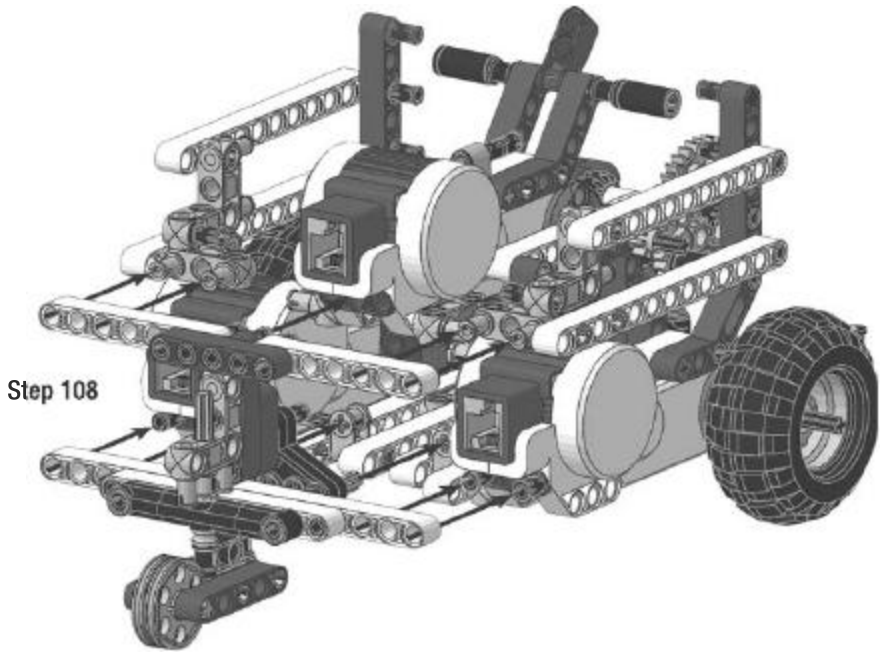

*Start building the frame that will be placed in the back of the robot to hold together the three main parts you just built.*



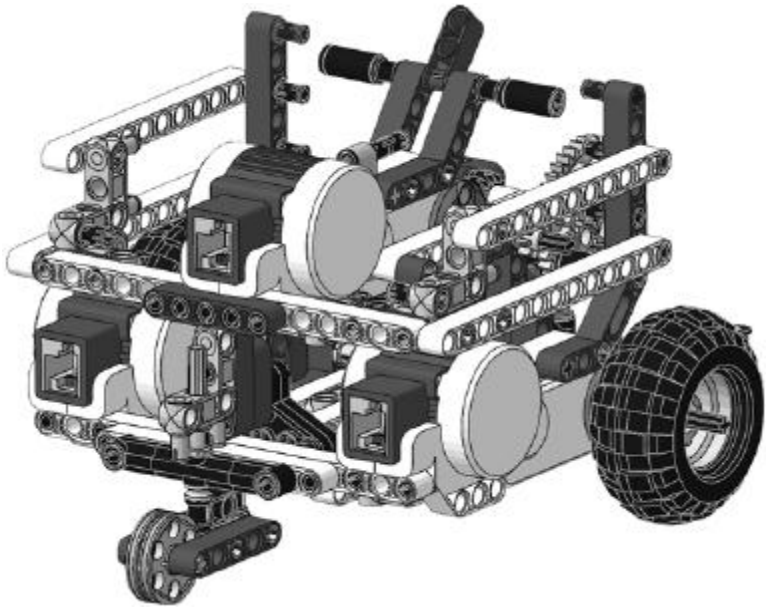

*Continue building the frame as shown.*



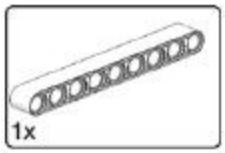

*This frame also holds the rear passive wheel.*

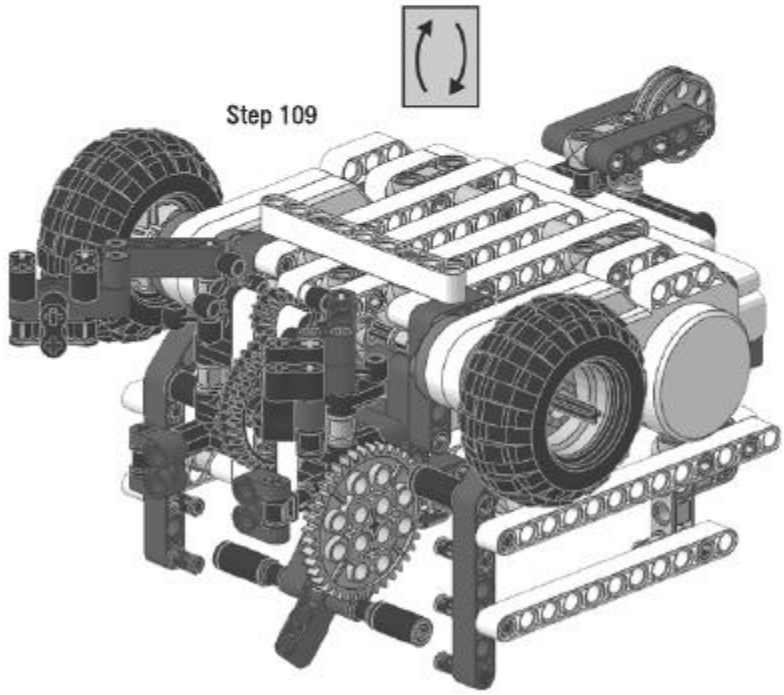



*Attach the frame on the back of the robot. Step 108 is tricky, because you have to insert 14 pins in their correct holes.*

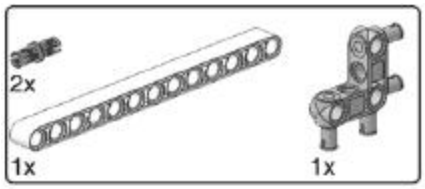


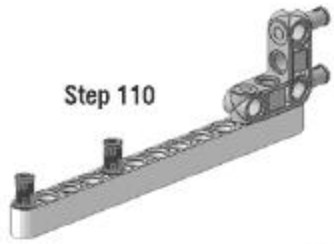
*Here you can see how the back frame looks when attached.*



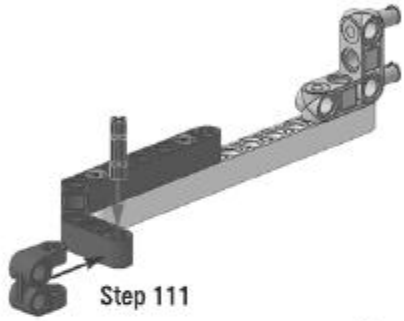



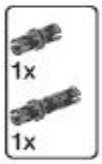

*Rotate the robot and insert a 9-long beam.*

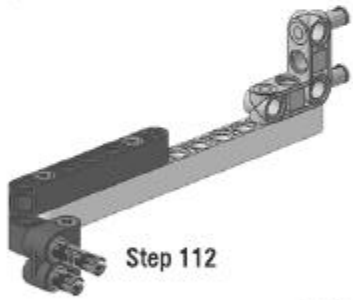


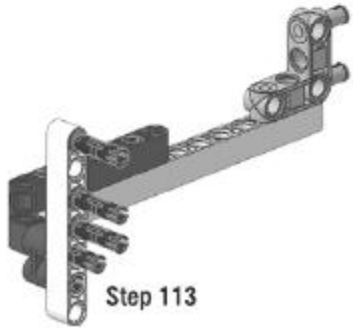




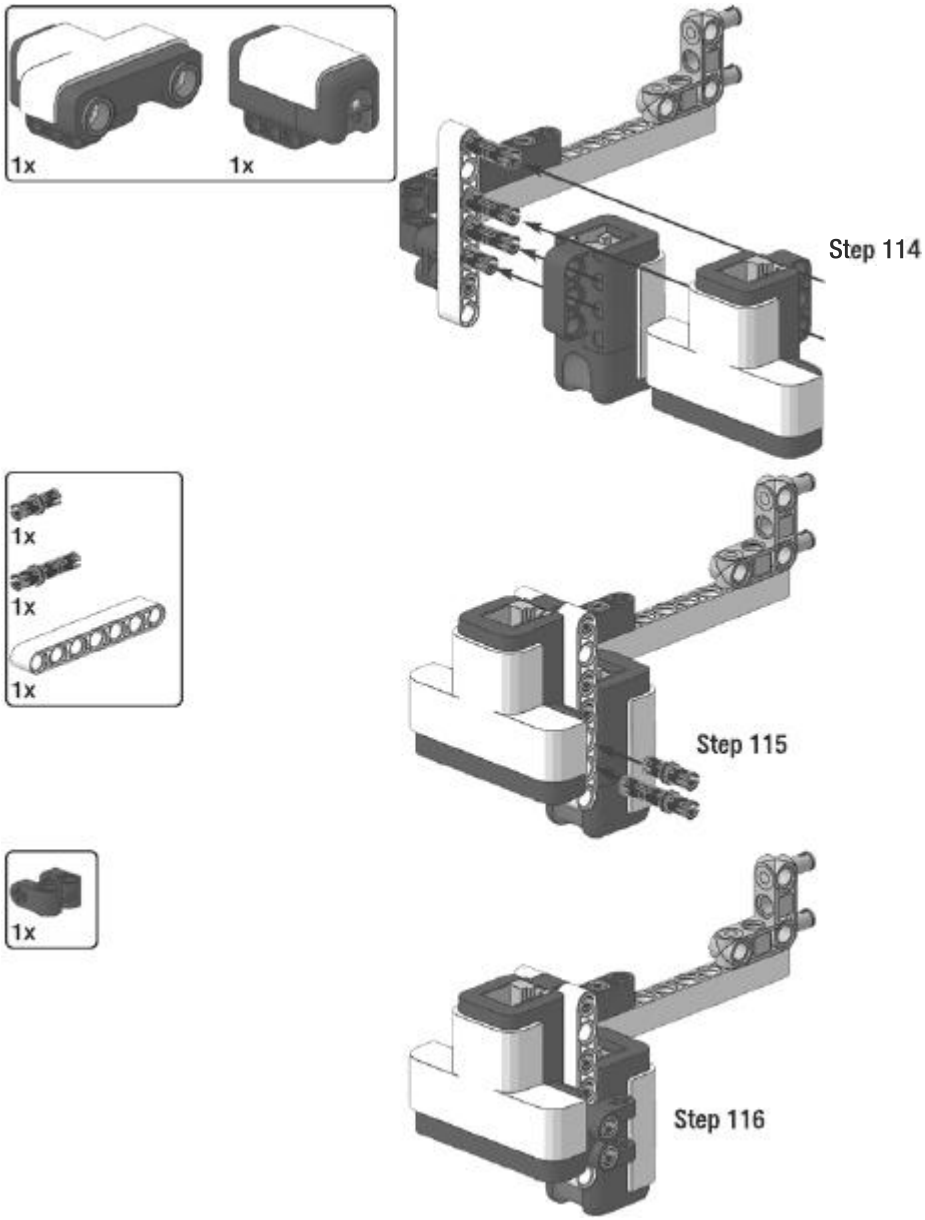



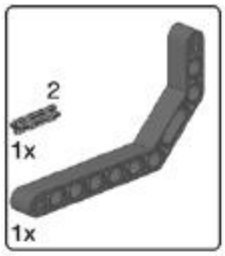


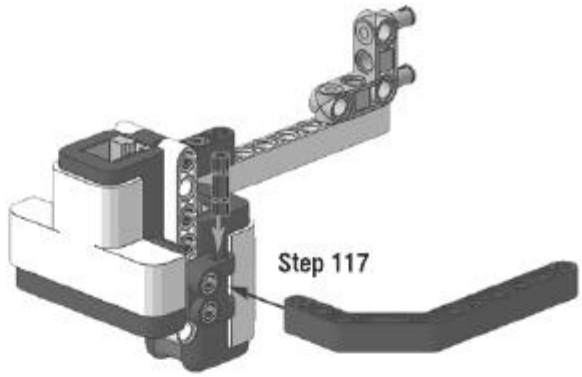



*Start building the front scanner of the robot.*

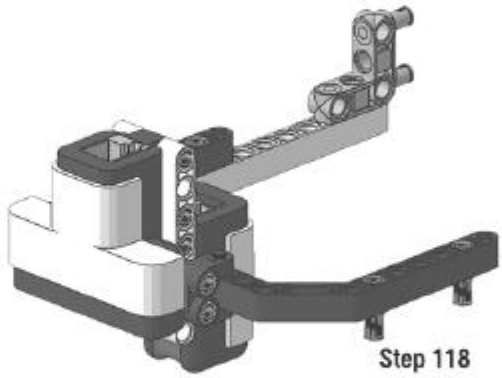


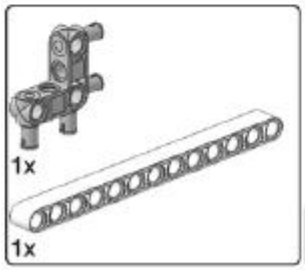

*Add the Light Sensor and the Ultrasonic Sensor.*

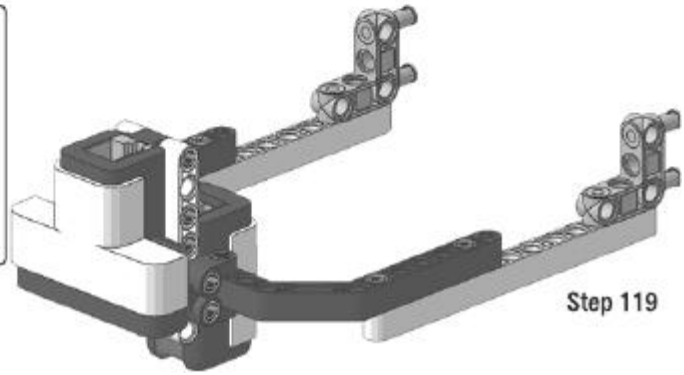


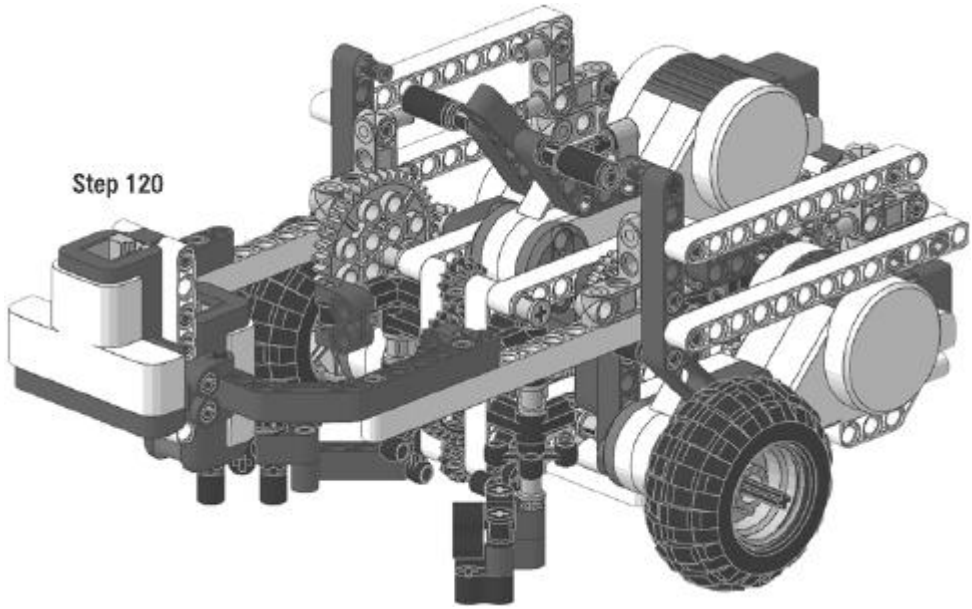




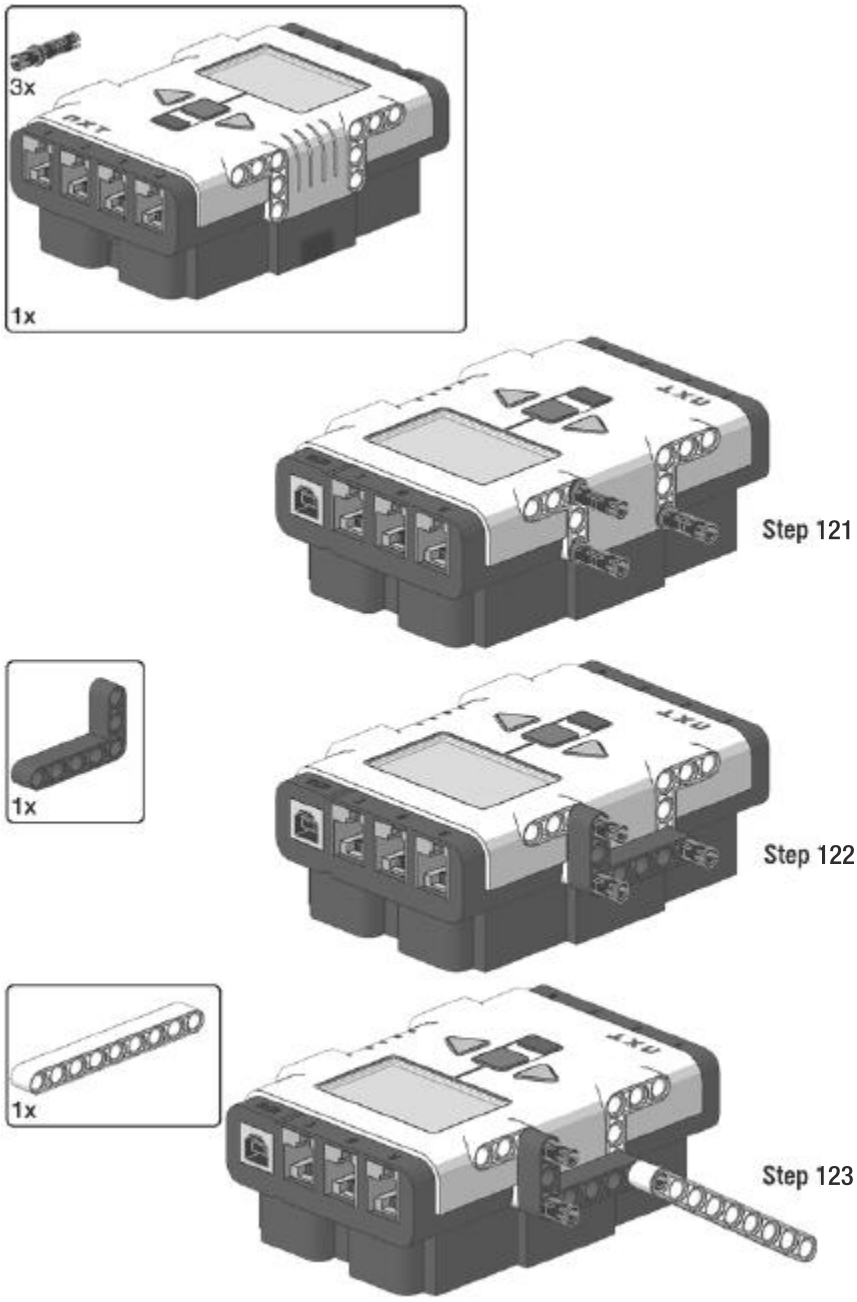



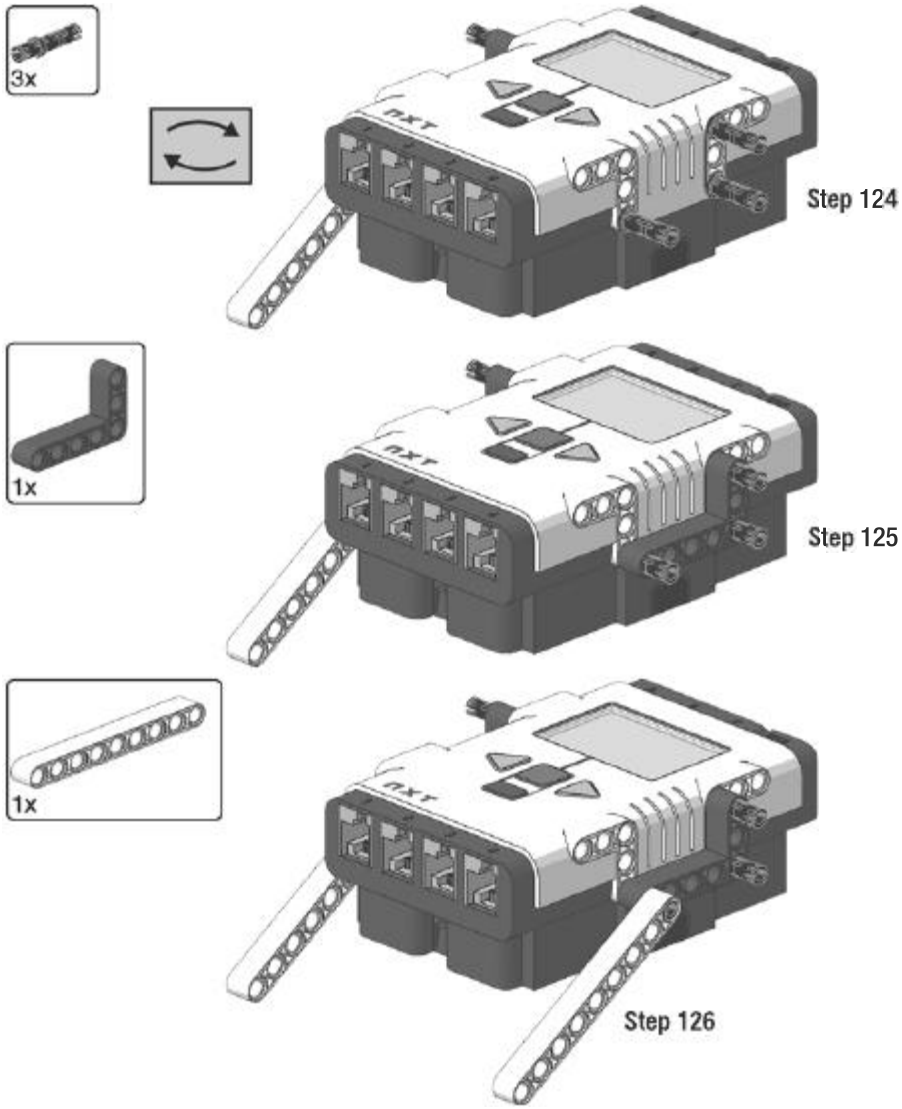

*Complete the front scanner.*



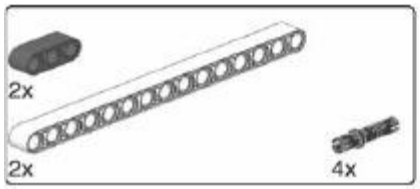

*Here's how the robot looks when the front scanner is attached.*

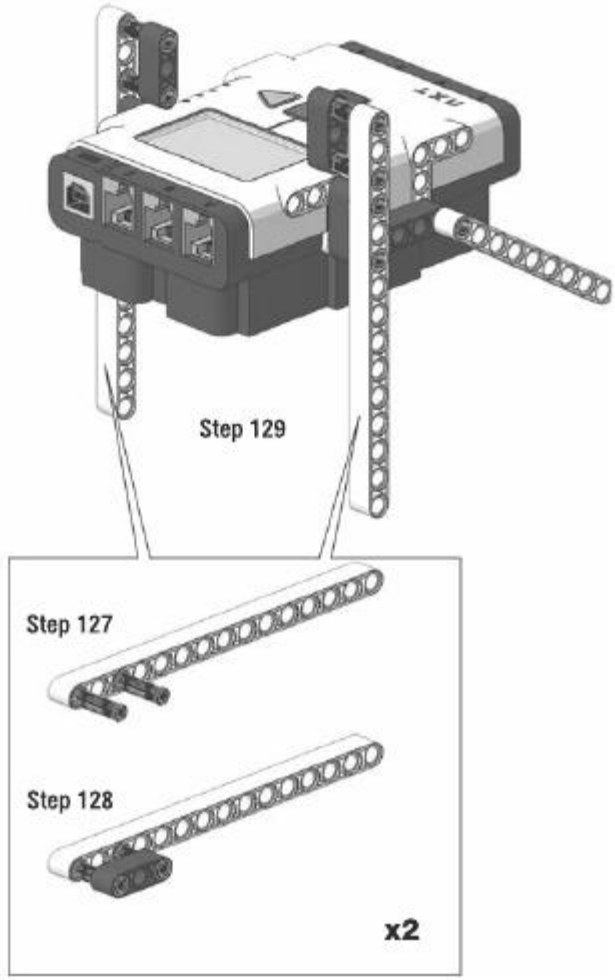



*Build the NXT holder frame.*

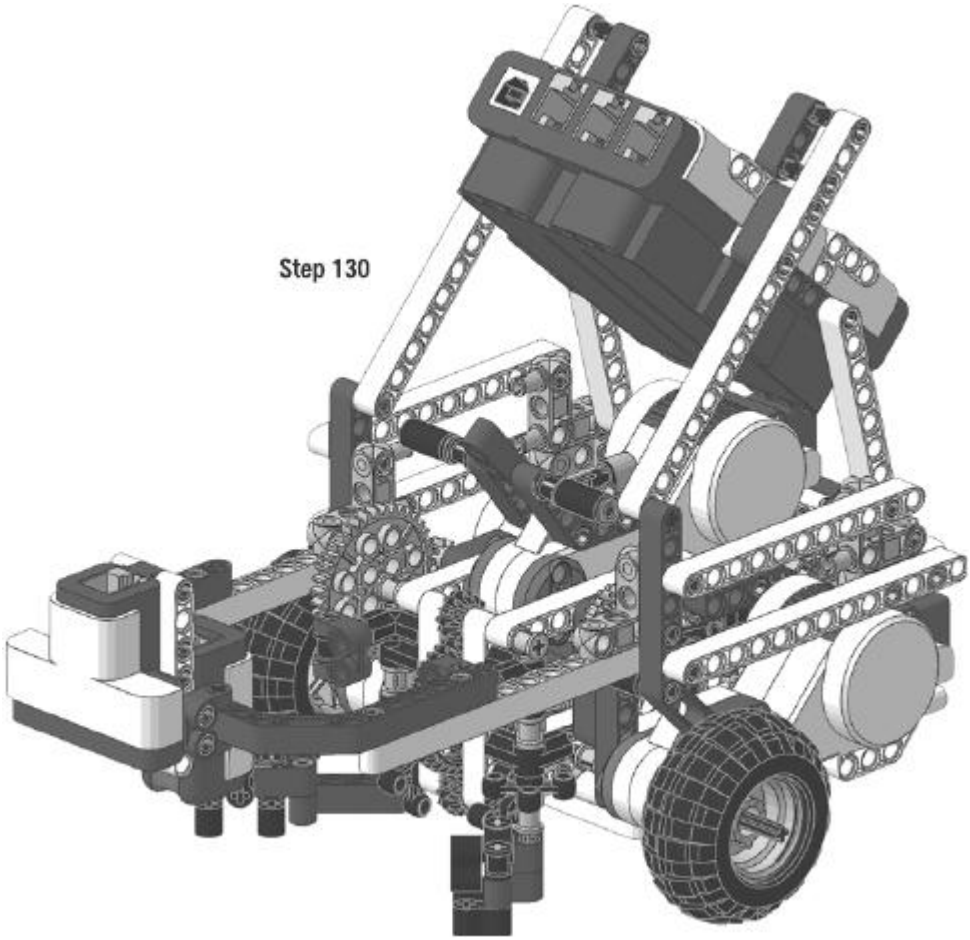



*The NXT frame will stand on the top of the robot.*

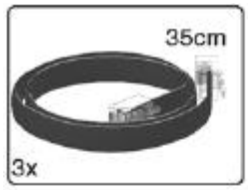


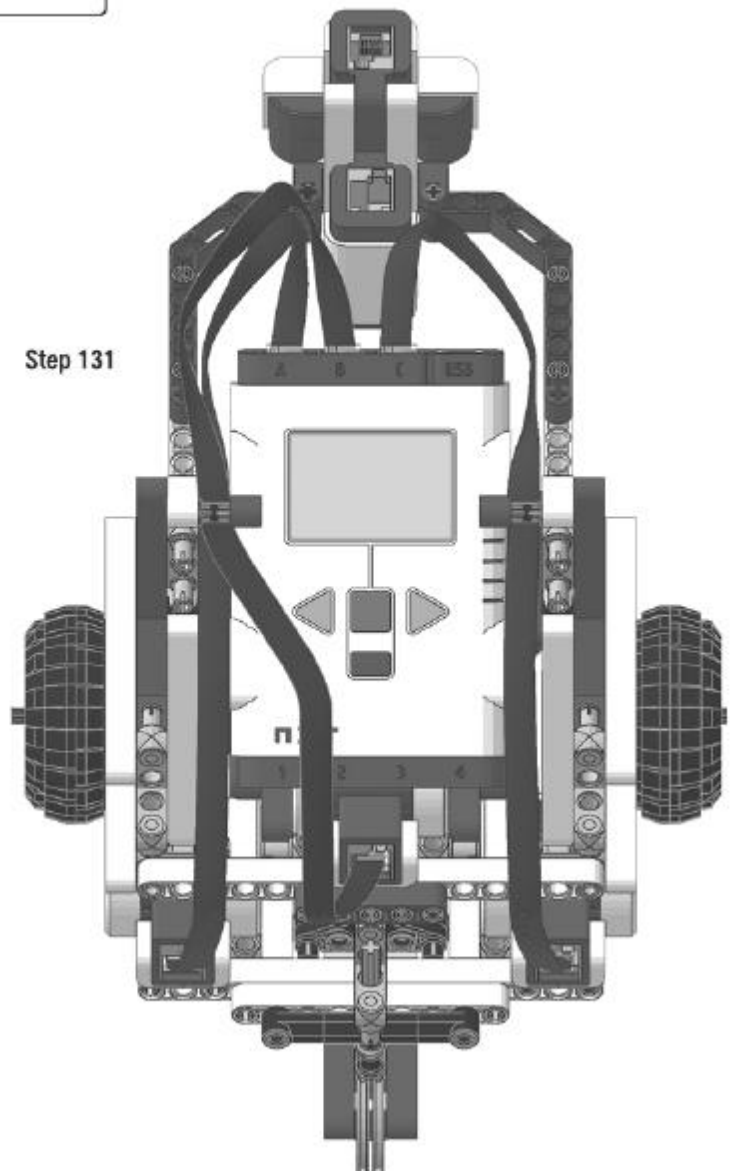




*Complete the NXT frame assembly.*

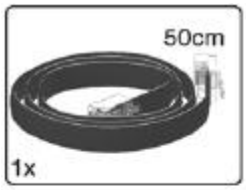


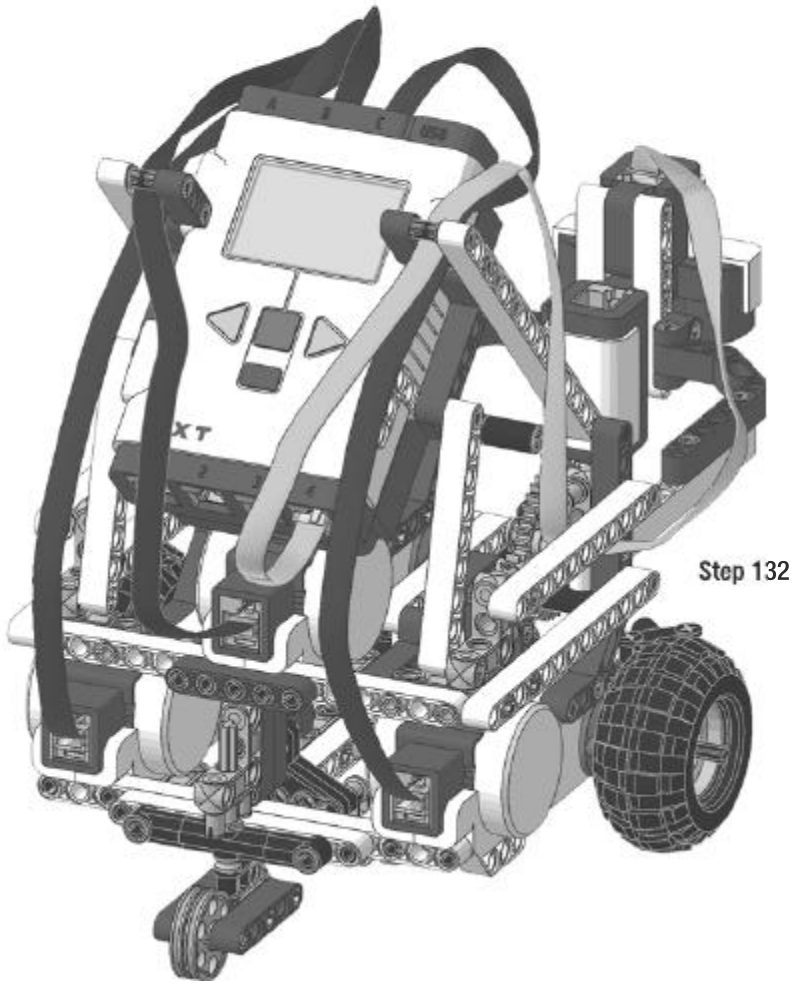

*Attach the NXT frame on the robot as shown.*



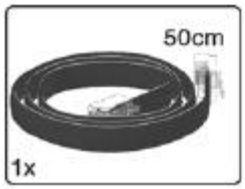



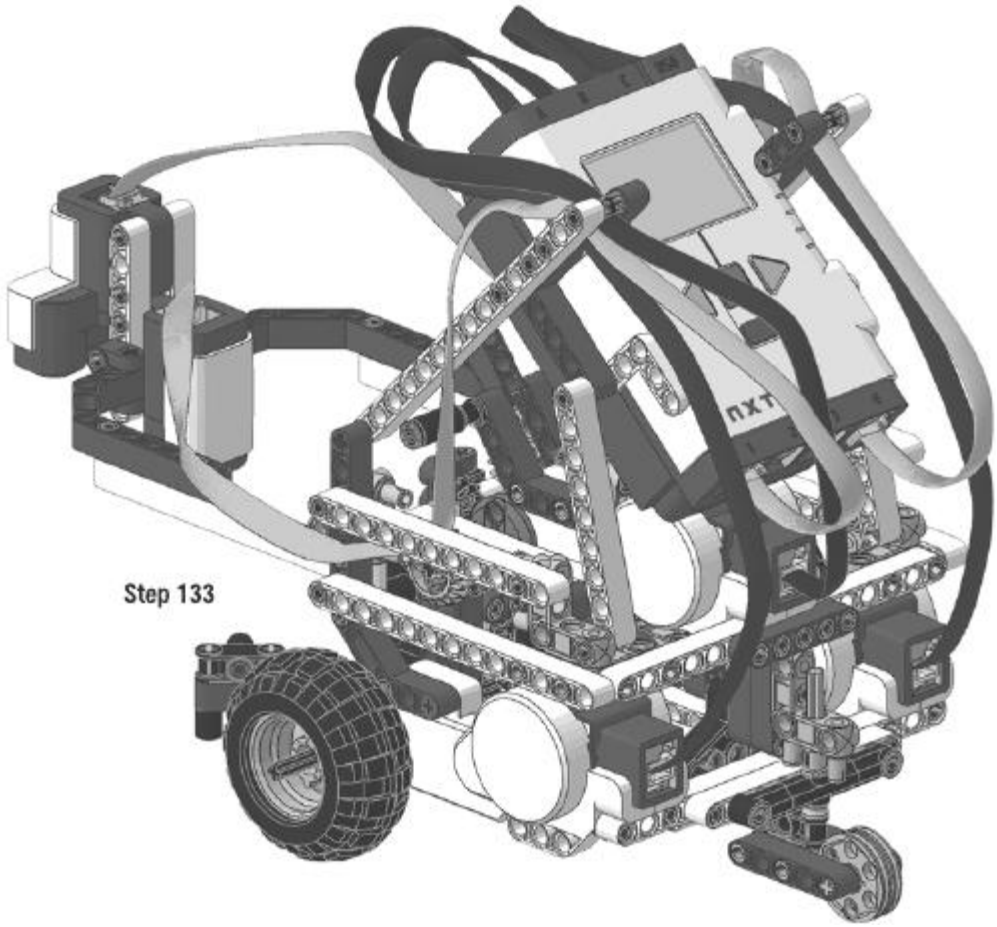

*Attach the left wheel motor to NXT output port A, the grabber motor to port B, and the right wheel motor to port C. Use three 35cm (14 inch) cables. Pass these cables through the NXT frame's top beams as shown here and in the next figure.*



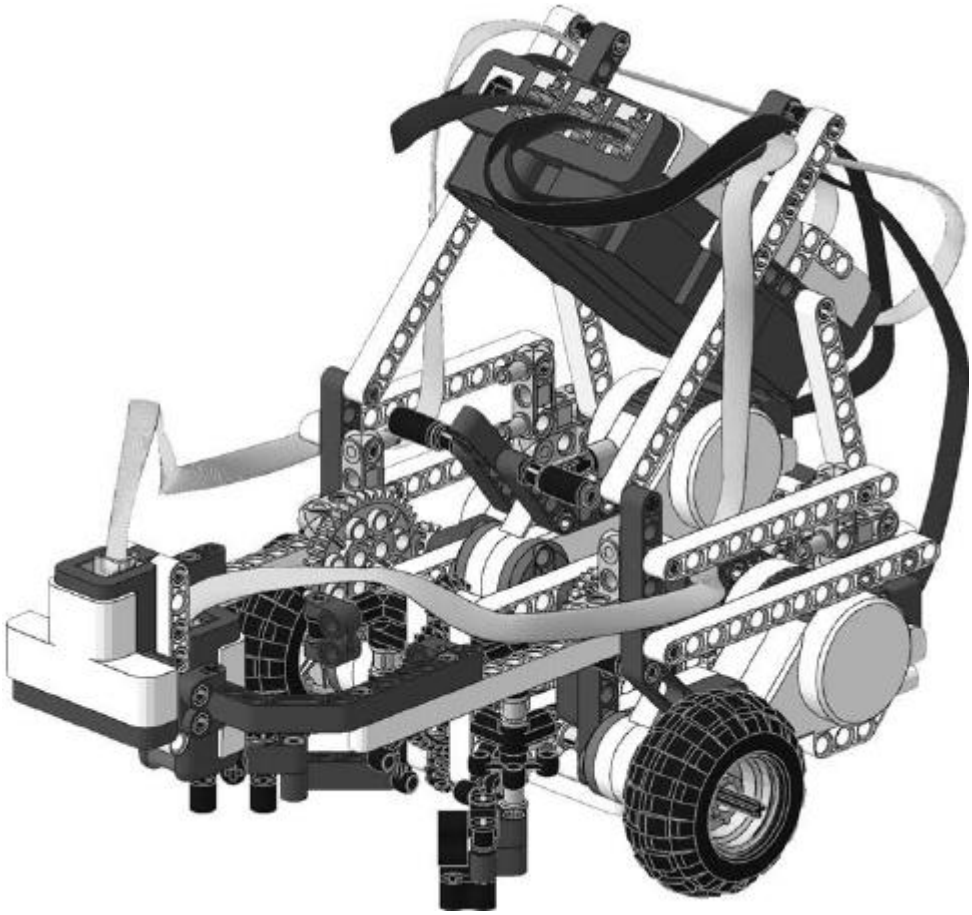




*Attach the Ultrasonic Sensor to NXT input port 4 using a 50cm (20 inch) cable.*




*Attach the Light Sensor (mine scanner) to NXT input port 1 using another 50cm (20 inch) cable.*



*The Mine Sweeper is completed.*

# Mines Building Instructions

In the NXT retail set there are no 2 - 4 black bricks, which are ideal to build the mines for our robot. So, you'll now see how to build four kinds of mines using the NXT set parts left. These mines are of the right size and color, so that the Mine Sweeper will be able to detect and handle them.

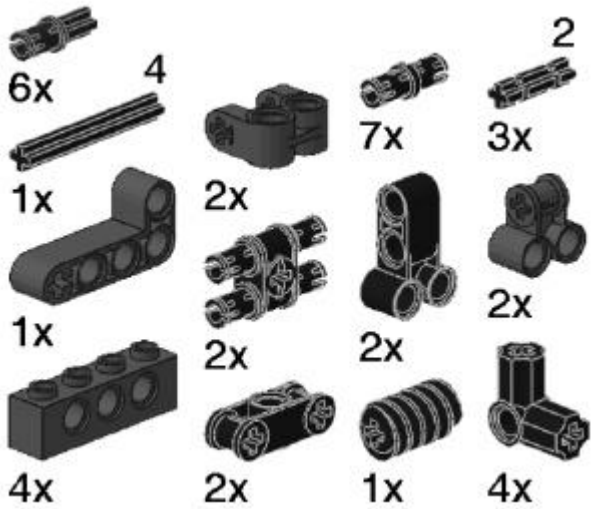
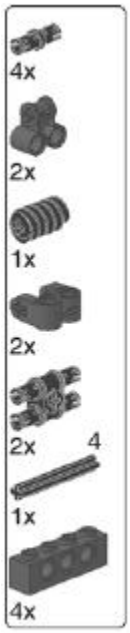
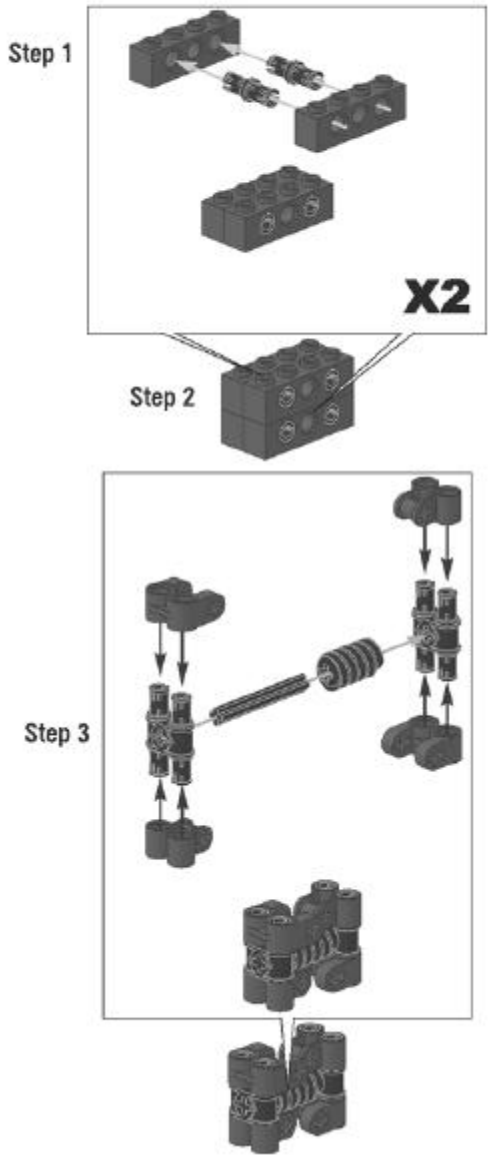



**Figure 7-9.** *Mines bill of materials*

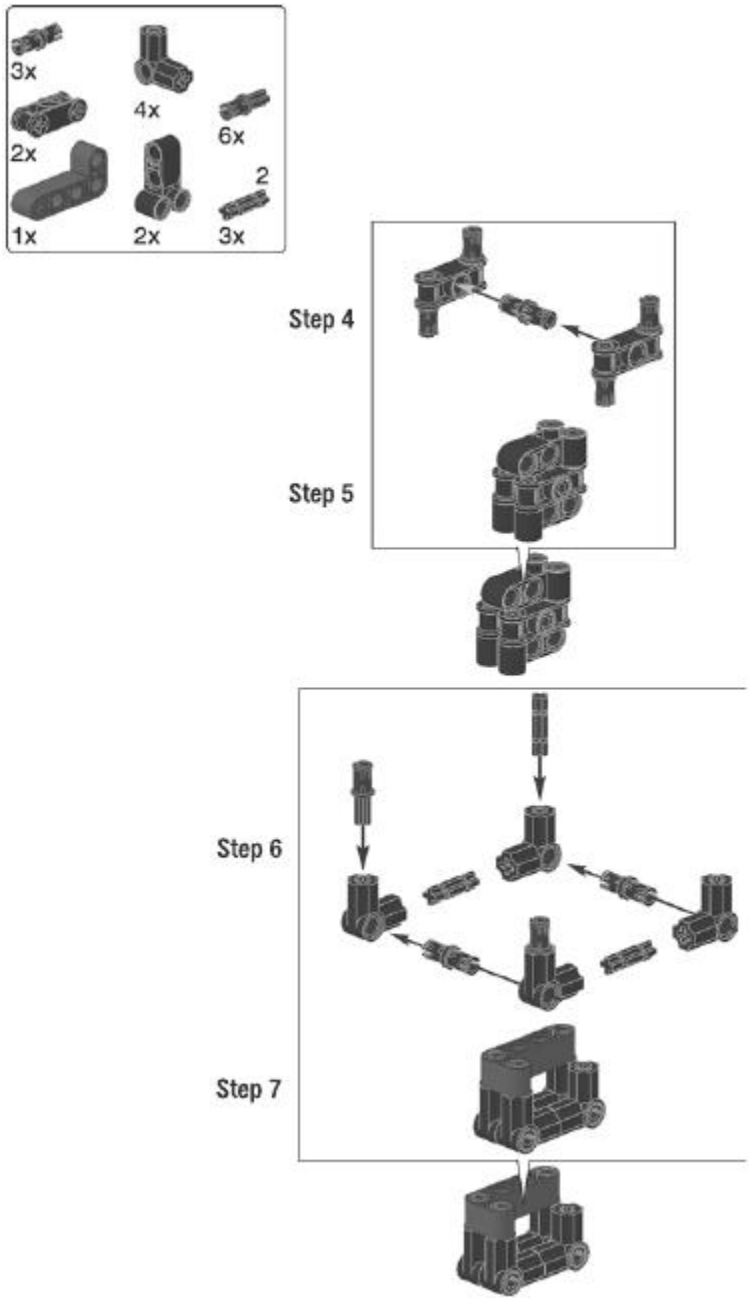
**Table 7-3.** *Mines Bill of Materials*

| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 6 | Blue | 43093.DAT | TECHNIC Axle Pin with Friction |
| 1 | Black | 3705.DAT | TECHNIC Axle 4 |
| 1 | Dark gray | 32140.DAT | TECHNIC Beam 5 Liftarm Bent 90 (4:2) |
| 4 | Dark gray | 3701.DAT | TECHNIC Brick 1 - 4 with Holes |
| 2 | Dark gray | 41678.DAT | TECHNIC Axle Joiner Perpendicular Double Split |
| 2 | Black | 32136.DAT | TECHNIC Pin 3L Double |
| 2 | Black | 32184.DAT | TECHNIC Axle Joiner Perpendicular 3L |
| 7 | Black | 2780.DAT | TECHNIC Pin with Friction and Slots |
| 2 | Black | 32557.DAT | TECHNIC Pin Joiner Dual Perpendicular |
| 1 | Black | 4716.DAT | TECHNIC Worm Screw |
| 3 | Black | 32062.DAT | TECHNIC Axle 2 Notched |
| 2 | Dark gray | 32291.DAT | TECHNIC Axle Joiner Perpendicular Double |
| 37 parts total (all included in NXT retail set) | | | |




*Here you are building both the first and second kind of mine.*




*Now you are building the third and fourth kind of mine. Notice that all these mines are made in a color and shape such that the robot can detect and handle them.*

# Summary

Though it's a wheeled robot, the Mine Sweeper is unique. Its peculiarity is the ability to detect, grab, and store "mines" (two piled 2 - 4 black bricks are ideal) scattered around its working area. This model has shown you an interesting hardware underactuated mechanism to accomplish two different tasks with the same motor (grabbing and storing the objects into its capacious hold). On the software side, you learned how to drive a wheeled robot with precision, when needed to accomplish refined tasks—in this case, to align the robot with the mine.

The treaded robot you will see in the next chapter is really something else. JohnNXT is a stunning robot, and you'll have the opportunity to see why.

## Exercise 7-1. Further Ideas

1. Improve the mine-centering algorithm by adding the mine width measurement. As a suggestion, use the following code:

```
int FindMineWidth( short edge )
{
  int x;
  byte ev;
  unsigned long time = CurrentTick();
  // save start position
  x = MotorRotationCount(LEFT_WHEEL);

  Spin(sign(edge));

  if (edge == LEFT)
  {
    // ignore the first edge found (the right one)
    WaitEdge(APPROACHING);
  }
  // wait for edge, dismissing with timeout constraint
  ev = WaitEdge(DISMISSING);
  Off(WHEELS);
  if (ev == EDGE_EV)
  {
    // if the edge was found, save position
    x = MotorRotationCount(LEFT_WHEEL);
  }
  return x;
}
```

Also, change the CenterMine() subroutine accordingly, adding the code proposed after the first centering procedure:

```
TextOut(0,LCD_LINE6,"    RIGHT EDGE    ");
width = FindMineWidth(RIGHT);
TextOut(0,LCD_LINE6,"    LEFT EDGE    ");
width = width - FindMineWidth(LEFT);
// and then center the mine
RotateMotorEx(WHEELS,80,width/2,-100,true,true);
```

2. It might happen that a mine is not detected even if it falls under the Light Sensor. If this unfortunate case occurs, the mine could end up in the space between the Light Sensor of the protruding scanner and the grabber fingers. Could you plan how to avoid this situation—both by using an additional sensor or thinking about some software security procedure, as a periodic maneuver to free these trapped mines?

3. As is, the Mine Sweeper goes around quite randomly. After any mine collection, you should design a solution to let it find its way again. For example, you can use a third-party digital compass sensor to keep the right direction.

CHAPTER 8

# JohnNXT Is Alive!

Here comes the top model of the book. By huge public demand, I'm proud to present JohnNXT! For those who don't know this robot yet, JohnNXT is an accurate desktop-scale–sized replica of the famous robot Johnny 5 from the *Short Circuit* 1980s movies. This is a fairly complex robot: you'll need more parts than the ones included in the two NXT retail sets to build it.

## Johnnicle: My LEGO Johnny 5 Chronicle

Johnny 5 is the robot star of the 1986 movie *Short Circuit*. The film became a cult classic, along with its sequel *Short Circuit 2*, which followed in 1988. If you were a kid in the 80s or 90s (like me), you surely know those films. In *Short Circuit,* the robot called Number 5, one of five prototypes built by Nova Robotics for military purposes, is struck by lightning while being recharged and becomes alive, achieving self-consciousness, a sense of humor, and an understanding of the value of life. At the end of the film, he renames himself Johnny 5. I think Johnny 5 is one of the most engaging robots in film history.

Inspired by these movies, I've been trying to build a LEGO Johnny 5 (J5) replica since age 11. I started with plain LEGO bricks. Then, I abandoned the project during my "dark age of LEGO" (a period of time when real-life interests got the better of LEGO play). I came back to this project when I first got my Robotics Invention System MINDSTORMS set, after I found on the Web the amazing two-RCX robot called Cinque (meaning "five" in Italian), a J5 replica built by Mario Ferrari in 2001. You can see this robot at http://www.marioferrari.org/cinque/cinque.html. I got inspired both by the general concept of that robot and by the functions it featured. After four draft prototypes, I got to the final shape of the RCX-based J5, which you can see in Figure 8-1.

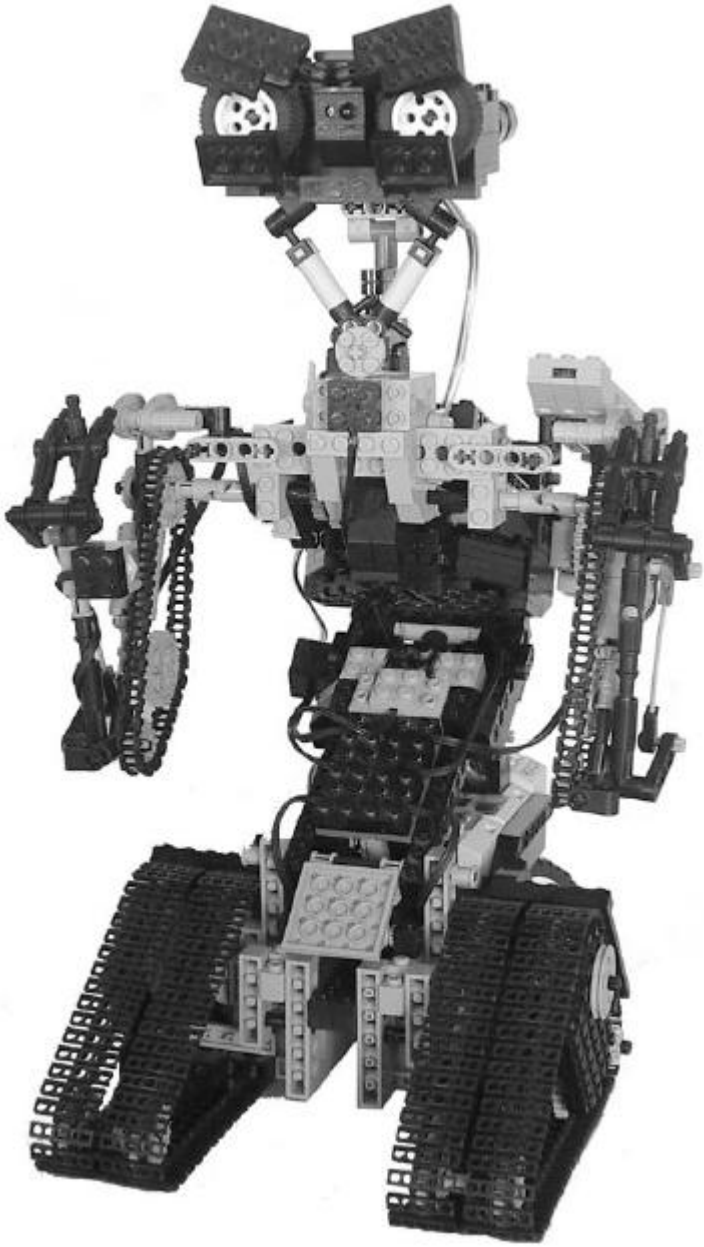


**Figure 8-1.** *RCX-based version of LEGO Johnny 5*

This LEGO model resembles the real J5, and already features all the functions that will also be in the NXT version you'll build: moving treads, rotating head, raising torso, and grabbing hands. To drive the six motors with just one RCX, I invented a motor multiplexer to allow the RCX to pilot up to six motors with its three output ports. I also built a homebrew infrared proximity detector in J5's head to let him follow my hand. This detector was built with the kind help of Philippe Hurbain, better known in the LEGO MINDSTORMS community as Philo.

The head is a bit over-dimensioned to feature the expressive poseable eyelids and neck. However, the entire model was *not* designed with regard to the proportions and the body's relative sizes. The head is too big, the base is too short, the shoulders are too large, the tread hubs are too small, and so on. Nonetheless, it remains a good functional J5 replica. I knew myself that I could do better.

The NXT system provided the perfect material to improve the project. As you can guess, the NXT version inherits a lot from the RCX version. In the early design stage, I started studying J5's real dimensions and proportions from all the photos that I found on the Web. From the many captured movie frames, I began to put a 2D CAD project together. I took the best measurements that I could to reproduce the J5 structure. After finishing this CAD drawing, I went on looking for the right LEGO elements to match the scaled-size J5 parts.

As a reference size to choose the scale for the whole robot, I used the dimensions of the NXT Ultrasonic Sensor, which fits perfectly as a J5 head. Since MINDSTORMS NXT's first sightings on the Web, many people have noticed the resemblance between the shape of the Ultrasonic Sensor and the J5 head, so they thought, "Johnny 5 is alive!" I surfed this popular wave of thinking. In Figure 8-2, you can see an early stage of JohnNXT's body development.

**Figure 8-2.** *An early stage of JohnNXT's body development*

The Ultrasonic Sensor alone was not big enough to shape the head, so I added some LEGO fairing panels. Other J5 parts that immediately followed in the size-matching process were the treads' hubs and passive wheel. The first complete JohnNXT (version 1) still had many defects, mainly regarding the passive wheel and the upper body structure. In Figure 8-3, you can get an idea of what JohnNXT version 1 looked like.

**Figure 8-3.** *JohnNXT version 1, still missing some parts*

The top tread hubs were made with large black turntables, which are in fact dimensionally perfect when coupled with the 40-tooth gears used as ground hubs. The laser shape was primitive, but already functional. It did not just rotate about a pivot—its movement was *eccentric*.

a combination of a rotation and a translation. This rough model was the starting point that resulted in the making of the second version of JohnNXT, shown in Figure 8-4.

**Figure 8-4.** *JohnNXT version 2, with one of the NXT microcomputers removed*

The head shape was already the final one, with eyelids and positionable neck pistons. It has not been modified further in successive versions. The fingertips were made with half-cut rubber axle joiners (horror for the purists!) to get more grabbing friction. The main problems raised in this version were the laser shape, the unreliable head-driving mechanism (a slippery rubber band), and the shoulders' shape. The worst problem was the fact that the treads' chains escaped from the largest hubs—the ones made with turntables.

For the third version (see Figure 8-5), the laser was completely redesigned. The lever mechanisms allow it to lower and move toward the shoulder, or to lift and move away from it. The tread hubs were made using 40-tooth gears, sacrificing the design a bit, but eliminating the escaping chain problem. Also, the passive wheel was improved, making it as smooth as possible, to avoid influencing the treads' movements with its friction on the ground.

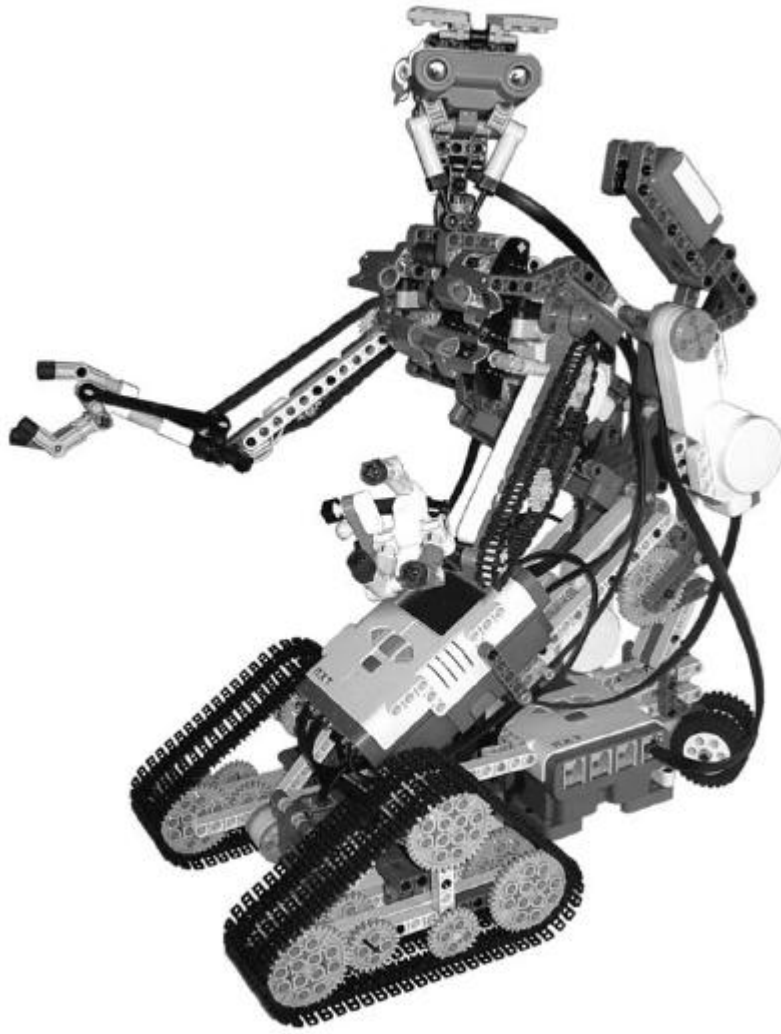


**Figure 8-5.** *JohnNXT version 3, featuring new treads and laser. Notice the rubber bands added on both elbows.*

A rubber band was added on both elbows to improve the underactuated two-DOF mechanism that allows the arms and hands to be driven by just one motor. As explained in Chapter 7, with regard to how to get more functions from a single actuator, you need one movement to be done before the other. Here, you need the arms to unfold before the hands open; the rubber band comes into play to help you. In the Johnny RCX version (similar to the Mine Sweeper grabber), the rubber band was not needed. This was because the forearm fell down under the force of gravity, because the whole arm assemblies were roughly vertical. In JohnNXT, the arms are inclined with respect to the ground, so I needed another force to make the forearms move before the hands. You can read the discussion about underactuation back in Chapter 7.

The rubber band used as the transmission to drive the head was replaced with a more reliable gear train. The head motor was built inside the upper body, while the arms' motor is on the side made invisible in the photo, shaping Johnny's typical toolbox. I was still not satisfied with the shoulder shape and the upper body in general. The end result is the final version 4, shown in Figure 8-6.

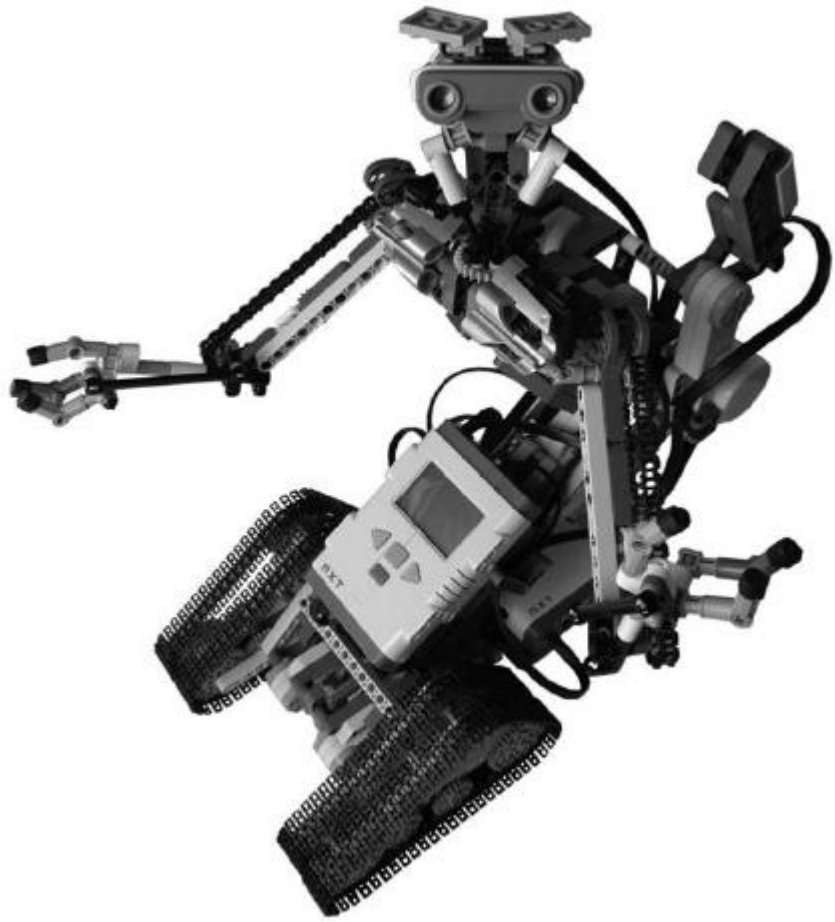


**Figure 8-6.** *JohnNXT version 4, with new shoulder shape. The NXTs communicate using Bluetooth. The final version you'll build is identical, but uses the NXT high-speed serial ports to let the two bricks communicate.*

Although many things might seem to appear the same as before, there are significant differences from the previous version. The shoulders are completely redesigned to look like a cylinder. The parallelogram frame of the upper body, used to keep the shoulders and head vertical while JohnNXT raises the torso, is now dimensionally more precise. The head tended to lean backward when the torso was raised, while now the neck axis always remains vertical, whatever inclination the upper body has. The arms' motor now has a structural function in the upper body. The arms' motor and head motor are swapped in position. Also, the head's and arms' geartrains are now different, to fit in the new frame. Finally, the treads have been structurally improved and lightened, by using a triangular frame. Could this version be the last one? Could I stop with version 4? Of course not.

The main reason is that this fourth version of JohnNXT has two NXTs that communicate using Bluetooth. What's wrong with it? Nothing, at first glance. But, wanting to control the whole model remotely using Bluetooth (see Chapter 9), I needed to free the connection and so had to plan a different manner of communication. So, I used the mysterious high-speed serial communication implemented behind port 4 of the NXT. The external shape of JohnNXT remains the same, while a new communication protocol had to be developed to use the serial ports.

So, the final version of JohnNXT, the one you'll build, is version number 5, where the version change is primarily due to the software update. It seems this number was assigned by fate! Our beloved JohnNXT can be controlled remotely via Bluetooth with a mobile phone, a PDA, or the remote control shown in Chapter 9. This concludes the development chronicle for the robot you'll build in this chapter.

# JohnNXT Features

Let me try to summarize all the JohnNXT features. From now on, I'll refer to version 5, the final one. This complex robot is a differential drive mobile robot, meaning that it has two independent treads to move around. You can vary its angular speed (rotational speed about its turning center) by driving the treads at different speeds; to drive it straight, the treads must turn at the same speed.

## Sensors and Actuators

Check out the following features in Figure 8-7. Two NXT bricks are used to drive six motors. The master NXT, placed in the abdomen, drives the two motors for the treads, as well as the third motor, used to turn the head. The slave NXT, built in the base, controls the other three motors. The fourth motor is used to raise the whole robot's body, both the lower column and the torso; the fifth motor is used to move the arms; and finally the sixth motor is used to arm and disarm the laser.

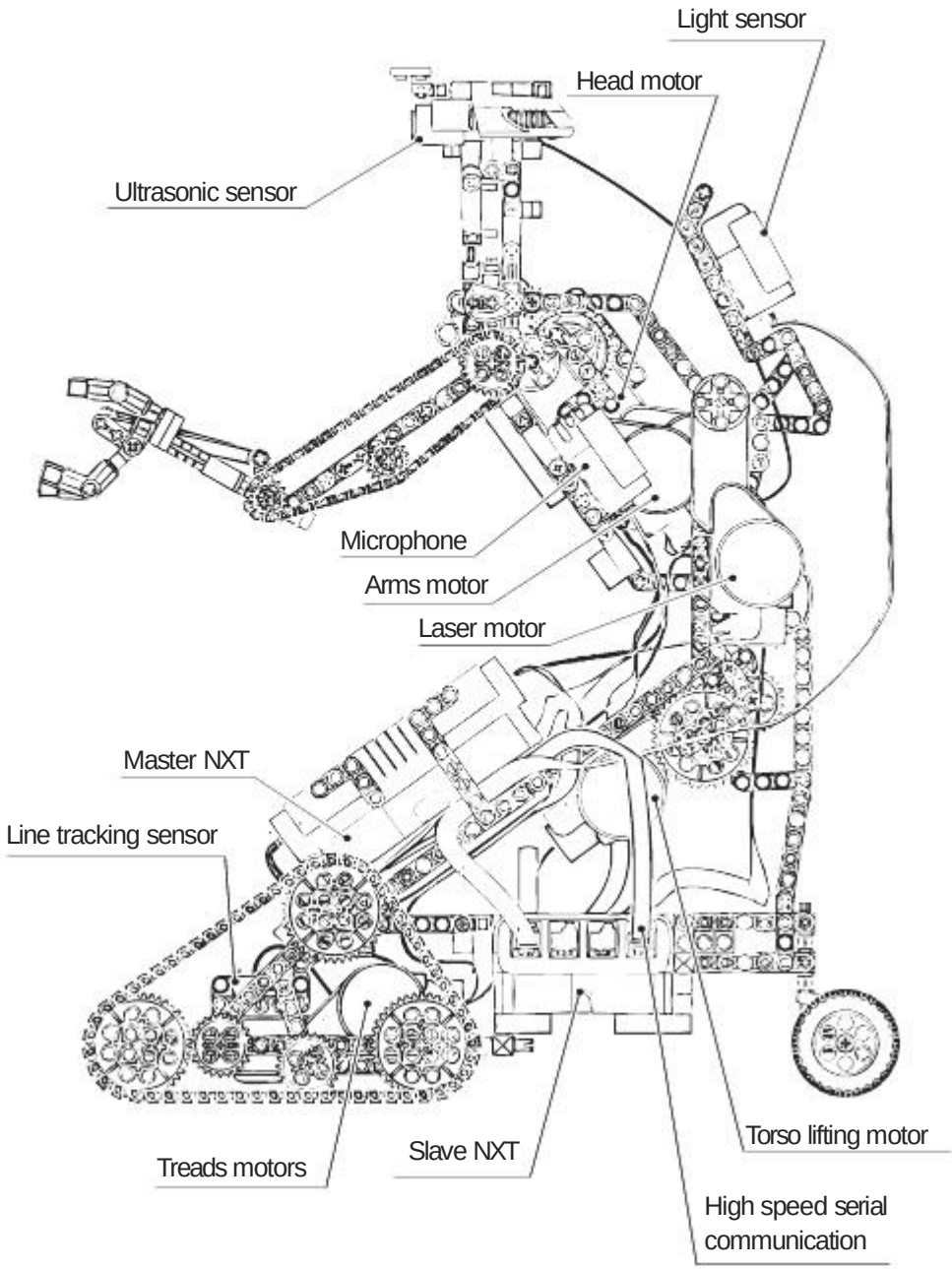



**Figure 8-7.** *Panoramic view of JohnNXT's features*

JohnNXT senses the world through a down-facing Light Sensor, the Ultrasonic Sensor in the head, and a microphone. It also uses the master NXT buttons to get commands from you. This equipment allows JohnNXT to follow a black line on the ground, to follow your hand, or to react to sounds. He can grab and lift small objects, commanded by a sound-counting FSM.

The moving parts that have a limited moving range—such as the head, the arms, the torso, and the laser (practically everything except the treads)—use the servomotors automagic built-in limit switch technique (see Chapter 4 for details) to reset every part to its zero position when the programs are started. The head's motor feels when the neck has reached its leftmost position, the arms' motor feels when the arms are completely folded, the torso's motor feels the downmost torso limit, and the laser's motor feels when the laser is completely raised or lowered, all using the same method. Now, you should realize how powerful this technique is. To do the same thing without servomotors, I would use many Touch Sensors, sometimes putting them in awkward places. To manage the intermediate positions of these mechanisms—to know by which angle a certain part must rotate to get into another state—I use decision tables, as described in detail in Chapter 3.

## JohnNXT's Behavior and Menu

JohnNXT features a simple autonomous behavior, schematized in Figure 8-8. When nothing interesting happens around him, he fools around, by performing some action and playing sounds from his repertoire. If the environment becomes crowded and noisy (the Ultrasonic Sensor sees something near and the microphone measures a loud continuous sound), JohnNXT becomes angry, and enters into attack mode: he aims his laser and remains there, threatening, until everything gets calm again. If he sees someone getting near and the noise level is reasonably low, he greets and then asks for input. It enters into the menu as soon as you press the master NXT orange button.

From the menu, you have access to all JohnNXT functionalities: remote control, line following, hand following, arms' sound control, and show off. In the *Remote Controlled* mode (*R/C*), you can fully control JohnNXT with the remote device described in Chapter 9. In the *Line Following* mode, you're asked to calibrate the bottom Light Sensor on the light ground and on the dark line, similar to the Turtle in Chapter 6, and then JohnNXT follows the line. In the *Hand Following* mode, JohnNXT swings his head and moves toward your hand when he sees it near. If the head is centered, he will move straight; if the head is turned on the side, he will move steering on that side; and if the hand is too near, JohnNXT will back up. In the *Arms Control* mode, you can control JohnNXT's arms with repeated sound pulses. In the *Show Off* mode, JohnNXT exhibits all his functionalities, synchronized with a looped soundtrack. The JohnNXT complete user guide and the main programming topics to get him to work will be discussed in the following sections.

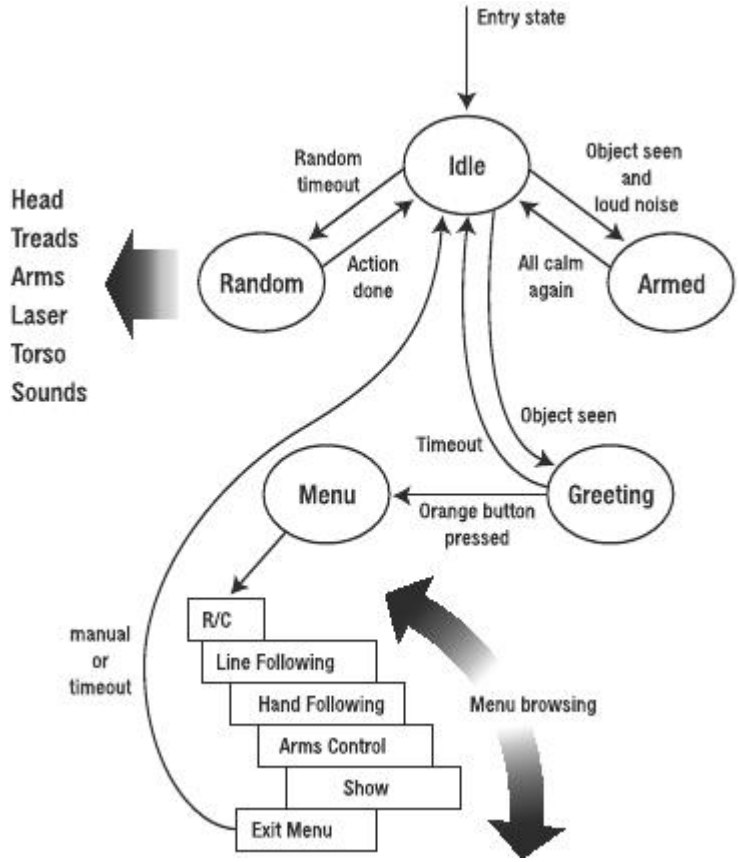



**Figure 8-8.** *The FSM describing JohnNXT's behavior*

# JohnNXT User Guide

Before we get into the programming details, a good user guide to exploit all JohnNXT's features is needed. This section will lead you through all JohnNXT's functions.

## Turning It On

Calm down, I'm not teasing you! I know that you know how to turn on two NXTs and how to start the programs! This section explains how to get JohnNXT working, seeing it as a system composed of three NXT bricks. Two of them are locally connected with a 6-wire cable, while the third NXT is the one for the remote control device.

Turn on both the master and the slave NXT in JohnNXT. Then, start the slave program. At the beginning, the slave performs the initialization of the hardware: it resets the torso to its downmost position, folds the arms, raises the laser, and then raises the torso to its middle position again. Only after that can it start receiving and executing the master's commands.

Now run the master program, which starts by telling the slave to reset all the moving parts, and then brings the head into the central position. If the master program was started before the slave, it would hang on, waiting for the slave to respond, and would continue to repeat the message until it received the slave acknowledgement (ack), meaning that the action has been completed. Using this kind of protocol, no command gets lost.

---

■ **Note** A message-exchanging communication protocol is termed synchronous, if the sender is blocked until the receiver responds with an ack, and the receiver is blocked until it receives a message. This protocol guarantees that the two sides are synchronized. On the contrary, if the sender puts the message into a buffer and continues its execution without waiting for the receiver's ack, the protocol is termed asynchronous. The protocol used by JohnNXT is synchronous. This means that if the slave program (receiver) is not responding, the master program (sender) would become stuck.

---

After that, you can start the program for the remote control (see Chapter 9); it automatically connects to the master NXT via Bluetooth. If you don't connect the remote, you'll get an error message saying "Remote offline" on the master NXT screen, when trying to activate the R/C item (*Remote Controlled* mode) in the JohnNXT menu. Remember that you cannot control JohnNXT remotely unless you select the *R/C* mode from the menu first.

## Autonomous Behavior

Once both master and slave programs are started, they reset all the moving parts of JohnNXT into their zero position. If you leave JohnNXT alone, in a quiet environment with nothing in front of him, he'll remain in the IDLE mode. He performs random actions, as shown in Figure 8-8. He also can randomly play some sound clips, such as "Need input," "Malfunction," and "Yeah! Johnny 5! That's cool!" If he sees an obstacle while in the presence of loud sounds, he'll lower his torso and aim the laser until the whole environment becomes calm again. When he simply sees something, he greets you and waits for you to press the orange button, to enter his menu.

## JohnNXT's Menu

From the menu, you can access the *R/C*, *Line Following*, *Hand Following*, *Arms Control*, and *Show Off* modes. From the menu, you can also choose to come back to the autonomous mode (IDLE). When you are in the menu, if you don't press any button for a while, JohnNXT will come back into autonomous mode automatically.

### R/C Mode

This functionality is enabled only if you've connected the remote control to the master NXT via Bluetooth before. In this mode, JohnNXT waits for commands from the remote and executes them. As you'll also see in Chapter 9, the remote device can access all JohnNXT's features.

The remote program for this particular robot works in two modes, toggled by pressing both remote buttons together. In the first mode, the joysticks control the treads and buttons that make the arms move to the open state (right trigger) or to the closed state (left trigger). In the second mode, the right joystick controls the head, the left joystick controls the torso, the

right trigger makes JohnNXT play a random sound from his repertoire, and the left trigger arms or disarms the laser.

### Line Following Mode

In this mode, you can make JohnNXT follow dark lines on light ground. The robot doesn't start going around at once: first, you are asked to sample the dark and light values from the bottom Light Sensor. The actual reading is shown on the NXT display. Place the JohnNXT line-tracking sensor on the line (dark) and press the left arrow button, then place it on the ground (light) and press the right arrow button. This way, the sampled values are stored and used to calibrate the line-following algorithm.

Press the orange button to start the line-following routine, and press it again to come back to the JohnNXT menu. If you don't calibrate the sensor readings, two default values for light and dark will be used.

### Hand Following Mode

In *Hand Following* mode, JohnNXT centers the head and then swings it left, center, right, center, and so on continuously, while staying still in place. When the Ultrasonic Sensor detects your hand, the robot will drive towards you, going straight or turning according to the actual position of the head. If the hand is really near, the robot will drive away from you.

### Arms Control Mode

In this mode, you can control the arms' position with sound pulses—for example, clapping your hands. When entering this mode, JohnNXT measures the ambient noise (so please keep silent for a short while) and folds its arms; then he waits for sharp sounds, which represent the commands you can give him:

- *One sound*: The arms are folded
- *Two sounds*: The arms are unfolded a bit, so they are vertical
- *Three sounds*: The arms are completely unfolded, but the hands are still closed
- *Four sounds*: The hands are open
- *Five sounds*: The wrists are rotated

Try clapping your hands and observe the text shown on the NXT screen: with every clap, a new arm state is chosen. If you stop clapping, the arms are moved into the state whose name is shown on the display. For example, consider starting with folded arms; clap two times and the arms are unfolded a bit; three claps unfold the arms, keeping the hands closed; and five claps (the maximum number) cause the hands to open and the wrists to rotate. A single clap brings the arms to their folded position again. So, you can go from every state to another.

Notice that you must produce the sound sequence with a constant timing, otherwise the robot won't behave as you would desire. For example, if you want to open the robot's hands starting from the state where the arms are folded, you must clap your hands five times. If you start clapping at a certain rate, but either the Sound Sensor fails to detect a clap, or you slow down the clapping rate too much, the robot will assume that the sequence is finished, and it

will move the arms according to the number of sounds detected up till that moment. The procedure is simple and reliable, but it needs a bit of practice!

### Show Off Mode

In this mode, JohnNXT performs a complete demonstration of its features. A sound loop is played, while JohnNXT moves right in time with it. It's a pity that I could not fit, in this small-scale model, a last feature that would be cool: making Johnny stand on his tread tips would make his dancing performance perfect!

# Programming JohnNXT

It's time to make JohnNXT come alive! A pedantic, detailed description of more than 1,600 lines of NXC source code that make up the JohnNXT programs (master and slave) would be at the least boring—for me to write, but more so for you to read. For this reason, I'll focus only on the programs' outlines and on their interesting aspects. Many solutions adopted for the JohnNXT programs have already been explained in detail in the preceding chapters. I can say for certain that this chapter's programs are a big summary of all the programming techniques shown in this book. I won't go back to basics to discuss the FSM that implements the autonomous behavior, the servomotors automagic built-in limit switch technique that was mentioned before, the line-following routine (like the one seen in Chapter 6), and other minor trivial aspects. In particular, I'll spend some time discussing the overall program's structure and the sound counting routine, used to control the arm movements.

You can download the complete code for JohnNXT from the Source Code/Download area of the Apress web site at http://www.apress.com.

## Panoramic View of the JohnNXT Software

When you are about to write a program for a complex project, the basic rule is to *divide et impera*, the Latin motto corresponding to "divide and conquer." To maintain the whole program easily, you should split it up into small parts; it is a good habit to split every program into many functions and subroutines. Of course, you already know that, but I mean that the program should also be divided into many files. So, you'll have an NXC file with the routines to manage head movements, another one devoted to remote control command reception, and so on. Then, in the main file, you just need to include the subfiles with the known preprocessor directive #include "filename.nxc".

The general working of JohnNXT is shown in Figure 8-8 and is described in the section "JohnNXT Behavior and Menu." Now, let's dive into the details. As you know, JohnNXT is controlled by two NXT bricks: the master, placed in the body, and the slave, the one built in the base. You need two different programs for these two NXTs. The FSM in Figure 8-8 is implemented in the master program, while the slave program's duty is just to receive and execute the commands from the master program. The whole slave NXT is just an interface to let the master control the three auxiliary motors attached to it: the torso, the arms, and the laser motors.

As said before, the two NXTs communicate through a standard 6-wire cable connecting their respective ports 4. So, I prepared a simple library to use this kind of communication, the High Speed Communication Library, implemented in the file HSlib.nxc. In the following section, you'll learn how to use this library, so you can also reuse it in your own two-NXT robots.

After that, I'll describe the two programs in an outline form, first starting from the slave, and then proceeding with the master, where we'll focus on the sound counting routine.

## High Speed Communication Library

Exploiting the serial communication behind port 4 is made simple by this NXC library.

- To use the library, simply include it at the top of your program:

  ```
  #include "HSlib.nxc"
  ```

---

■ **Caution** The HSlib.nxc file must be located in the same folder of your program, otherwise the compiler won't be able to find it.

---

- At the beginning of the program, you can initialize port 4 by calling

  ```
  SetHSPort();
  ```

  This is enough to tell the NXT firmware to enable the RS485 chip for the serial communication.

- To send a string, simply call

  ```
  SendHSString ( msg );
  ```

  where msg is the string you want to send.

- To send a number, call

  ```
  SendHSNumber ( num );
  ```

  where num is the integer you want to send.

- To receive a string, make the following call:

  ```
  result = ReceiveHSString ( msg );
  ```

  After the call, the Boolean result will be true if the buffer is not empty, and the msg string will contain the received data. If the buffer is empty, or the data is equal to zero, the result value will be false.

- To receive an integer number, call

  ```
  result = ReceiveHSNumber ( num );
  ```

  As before, after the call, the Boolean result will be true if the buffer is not empty, and the num variable will contain the received data. If the buffer is empty, or the data is equal to zero, the result value will be false. This function, as does the preceding, features a timeout mechanism to avoid remaining stuck if the high-speed buffer is empty.

Next I'll describe the two programs in outline form, starting from the slave—the simpler—and proceeding with the master, where I'll focus on the sound counting routine.

## Slave Program

To start dealing with the slave program, take a look at the program excerpt in Listing 8-1. Some parts, similar to the programs in the preceding chapters, are omitted. The purpose here is to give you an idea of what the program outline looks like.

**Listing 8-1.** *The JohnNXT Slave Program Code*

```
#include "J5Defs.nxc"
#include "HSlib.nxc"

//mechanical state variables
short torso_state, arms_state;

////////////////////////////////////////////////////////////////////////////////
//LASER//
////////////////////////////////////////////////////////////////////////////////

sub Laser (short new_state)
{
   [...]
}

////////////////////////////////////////////////////////////////////////////////
//ARMS//
////////////////////////////////////////////////////////////////////////////////

   [...]

sub Arms( short new_state )
{
   [...]
}

sub ArmsStepOpen()
{
   [...]
}

sub ArmsStepClose()
{
   [...]
}
```

```
////////////////////////////////////////////////////////////////////////////
//                                TORSO                                   //
////////////////////////////////////////////////////////////////////////////

   [...]

sub Torso( short new_state )
{
   [...]
}

sub TorsoStepUp()
{
   [...]
}

sub TorsoStepDown()
{
   [...]
}

////////////////////////////////////////////////////////////////////////////
//                                INIT                                    //
////////////////////////////////////////////////////////////////////////////

void J5_Init()
{
   SetHSPort();
   torso_state = UNKNOWN;
   arms_state = UNKNOWN;
   Torso(T_DOWN);
   Arms(A_FOLDED);
   Laser(L_UP);
}

////////////////////////////////////////////////////////////////////////////
//                                MAIN                                    //
////////////////////////////////////////////////////////////////////////////

// show a message to describe the command being executed
bool ShowRxCommand ( short s )
{
  [...]
   return error; //returns true if the command is unknown
}
```

```
// parse received command and execute it
void ExecuteCommand ( short command )
{
  if (abs(command)/10 == TORSO_ACTIONS)
  {
    if (command == T_STEPUP) TorsoStepUp();
    else if (command == T_STEPDOWN) TorsoStepDown();
    else Torso(command);
  }
  if (abs(command)/10 == ARMS_ACTIONS)
  {
    if (command == A_STEP_OPEN) ArmsStepOpen();
    else if (command == A_STEP_CLOSE) ArmsStepClose();
    else Arms(command);
  }
  if (abs(command)/10 == LASER_ACTIONS) Laser(command);
}

task main()
{
  KeepAliveType kaArgs;  // this structure is used to call SysKeepAlive
  int cmd;

  J5_Init();
  Torso(T_MID);
  // receive commands from master.
  // send an ack meaning "command received"
  // execute the command
  // send another ack meaning "command executed"
  SendHSNumber(ACK_RX);
  SendHSNumber(ACK_DONE);
  while(true)
  {
    if (ReceiveHSNumber(cmd))
    {
      TextOut(0,LCD_LINE3,"send rx ack  ");
      SendHSNumber(ACK_RX);
      ShowRxCommand(cmd);
      ExecuteCommand(cmd);
      TextOut(0,LCD_LINE3,"send exe ack  ");
      Wait(100);
      SendHSNumber(ACK_DONE);
    }
    else
    {
      TextOut(0,LCD_LINE3,"idle ");
      SendHSNumber(ACK_ERR);
    }
```

```
        // keep the NXT alive: this system call
        // resets the sleep timer, to avoid having the NXT
        // turn off automatically when the program is running.
        SysKeepAlive(kaArgs);
    }
}
```

In Listing 8-1, you find the subroutines to actuate the three motors (torso, arms, and laser) using the decision tables (see Chapter 3). The `J5_Init()` function initializes the state variables torso_state and arms_state, and performs the routines to bring the moving parts into their zero position. The communication between the NXT bricks is set up simply by calling the function SetHSPort(), defined inside the High Speed Communication Library HSlib.nxc. At the top of the code, you can read the following preprocessor directives:

```
#include "J5Defs.nxc"
#include "HSlib.nxc"
```

With the first line, you tell the compiler to include the JohnNXT header file J5Defs.nxc, where all the robot constant definitions are specified: the motor ports; the sensor ports; and the opcodes for the various commands written in capital letters, such as `A_FOLDED`, `T_DOWN`, or `UNKNOWN`, for example. The other inclusion is for the communication library described earlier. After the initial definitions, the code goes on with the low-level routines for the mechanics.

The program starts executing the main task; here the `J5_Init()` function is called. After that, an infinite loop is started, to receive and execute the commands for the master NXT.

The incoming commands are received using the ReceiveHSNumber(cmd) function. Note that the ReceiveHSNumber function is called inside the parentheses of the **if** structure, and so the value returned by that function affects the subsequent working. If the result returned is true, it means that no error occurred when receiving the command, and the `cmd` variable, passed as an argument, contains the command sent by the master NXT; so, the first branch of the **if** structure is taken. The slave sends a first ack `ACK_RX` to the master, meaning "Hey, I got the message correctly!" Then it displays information onscreen by calling the ShowRxCommand(cmd) function, it executes the command with ExecuteCommand(cmd), and after that sends another ack `ACK_DONE` to the master, to say "I've executed your command!" The reason for using two different acks is to inform the master about both reception and execution of the command. So, the master can choose to wait for the remote operation's completion (signaled by the `ACK_DONE` ack), or to go on with its own task after having sent the command. To tell the master to repeat the last message, the slave can send a third kind of ack, `ACK_ERR`, if the ReceiveHSNumber function returned `false`, meaning that some error occurred (`else` branch). The Remote function (inside the J5_comm.nxc file), used by the master program to send commands to the slave, can thus exploit the information carried by these different acks to implement a simple error detection and correction protocol.

The last thing done before closing the loop in main is to invoke the system call SysKeepAlive. With that, you reset the firmware's sleep timer, and you can avoid the annoying issue that causes the NXT to turn off right in the middle of play, even if the program is running! I once heard about a LEGO sumo competition lost by a robot that had this kind of problem. You can solve it as shown, or manually change the sleep timer settings using the NXT on-brick menu.

This is the overall structure of the slave program. It isn't complicated. The subroutines to move the auxiliary motors are not reported here, but you can read those in the source code of JohnNXT; they are simple to understand once you've learned the theory behind them. The FSMs are explained in Chapter 3 and the practical example is in Chapter 4 (Listings 4-6 and 4-7, Tables 4-1 and 4-2).

## Master Program

In Listing 8-2, I show the master JohnNXT program's outline. Even though the real code is replaced by explanatory comments, the listing is still very long. That's why I chose not to report the entire code.

**Listing 8-2.** *The Master Program Outline*

```
#include "J5Defs.nxc"
#include "J5_comm.nxc"
#include "J5_head.nxc"
#include "J5_sounds.nxc"
#include "J5_lineflw.nxc"
#include "J5_handflw.nxc"
#include "J5_show.nxc"
#include "J5_remote.nxc"

// global state variable for JohnNXT behavior FSM
short J5_state;

void J5_Init()
{
  TextOut(0,LCD_LINE1,"Initializing...");
  SetHSPort();
  SetSensorLowspeed(EYES);
  SetSensorSound(MIC);
  Remote(L_UP,ACK_DONE);
  Remote(T_MID,ACK_DONE);
  Remote(A_FOLDED,ACK_DONE);
  head_state = UNKNOWN;
  J5_state = IDLE;
  Head(CENTER,1);
}

////////////////////////////////////////////////////////////////////////////////
// MANIPULATION //
////////////////////////////////////////////////////////////////////////////////

// here is the code to control JohnNXT's arms using sound pulses

[...]
```

```
////////////////////////////////////////////////////////////////////////////////
//                    SUBROUTINES FOR BEHAVIOR STATES                         //
////////////////////////////////////////////////////////////////////////////////

sub J5_Armed()
{
  // lowers torso and laser
  // aiming it at whoever is present
  // when a loud noise occurs
}

sub J5_Greeting()
{
  // makes JohnNXT greet and then waits for someone
  // to click the orange button, to enter menu.
  // if nothing happens within 3 seconds
  // comes back to idle mode
}

sub Idle_treads()
{
  // called by J5_Random, moves treads
}

sub J5_Random()
{
  // performs random actions
}

sub J5_Idle()
{
  // Reads Ultrasonic Sensor and mic values:
  // if someone is near, enters GREETING mode,
  // but if there is also a loud noise,
  // enters ARMED mode.
  // Otherwise calls J5_Random,
  // then waits for a random amount of time
}

////////////////////////////////////////////////////////////////////////////////
//                                  MENU                                      //
////////////////////////////////////////////////////////////////////////////////

[...] //menu opcodes definitions

short DisplayMenuItems(short item)
{
```

```
    //displays the specified menu item
}

short MenuEngine()
{
    //calls DisplayMenuItems to browse the menu and returns
    //the chosen item
}

sub J5_Menu()
{
    //calls MenuEngine() and
    //changes J5_state according to menu choice
}

////////////////////////////////////////////////////////////////////////////////
//MAIN//
////////////////////////////////////////////////////////////////////////////////

task main()
{
    KeepAliveType kaArgs;
    J5_Init();
    J5_state = IDLE;
    // J5 behavior FSM
    while (true)
    {
        switch(J5_state)
        {
            case ARMED:
                J5_Armed();
                J5_state = IDLE;
                break;
            case GREETING:
                J5_Greeting();
                break;
            case HANDFOLLOW:
                J5_HandFollow();
                J5_state = MENU;
                break;
            case IDLE:
                J5_Idle();
                break;
            case LINEFOLLOW:
                J5_LineFollow();
                J5_state = MENU;
                break;
```

```
            case MANIPULATION:
                J5_Manipulation();
                J5_state = MENU;
                break;
            case MENU:
                J5_Menu();
                // next state is determined by user choice
                break;
            case SHOWOFF:
                J5_Show();
                Remote(T_MID,ACK_DONE);
                Remote(A_FOLDED,ACK_DONE);
                Remote(L_UP,ACK_DONE);
                J5_state = MENU;
                break;
            case REMOTE_CONTROL:
                if (BluetoothStatus(0)==NO_ERR)
                {
                // if the Bluetooth master NXT (remote control)
                // is connected, call J5_Remote_Control()
                    J5_Remote_Control();
                    Remote(T_MID,ACK_DONE);
                    Remote(A_FOLDED,ACK_DONE);
                    Remote(L_UP,ACK_DONE);
                }
                else
                {
                // show error message
                    ClearLine(3);
                    TextOut(0,LCD_LINE3,"Remote offline!");
                    Wait(1000);
                }
                J5_state = MENU;
                break;
        }
        // keep the NXT alive: this system call
        // resets the sleep timer, to avoid having the NXT
        // turn off automatically when the program is running.
        SysKeepAlive(kaArgs);
    }
}
```

The master program core is in the main task, where I implemented the FSM that regulates JohnNXT's behavior. The diagram of this FSM is shown in Figure 8-8. The structure of main is quite similar to the skeleton program in Listing 3-4, described in Chapter 3, in the section "FSM General Implementation." You can refer to that chapter to go over the FSM argument again, if you still have some doubts.

At a quick glance, the program looks modular: the working of every FSM state is implemented in a separate subroutine, to help the readability and maintainability of this big program. Also, notice the many NXC subfile inclusions at the top of the program.

The master program is not as complicated as you might think. It is long, but it uses many techniques and tricks that I have already presented throughout the book, with which you might feel familiar by now. Next, I'll discuss the part of the master program worthy of a detailed explanation: the FSM that allows you to control JohnNXT's arms with sounds.

## Sound Counting FSM

As described in the section "JohnNXT User Guide," when you enter the Arms Control mode in JohnNXT's menu, you can control the position of its arms with sound pulses. Now, let's analyze the mechanism that makes this possible: the sound counting FSM, illustrated in the diagram in Figure 8-9.

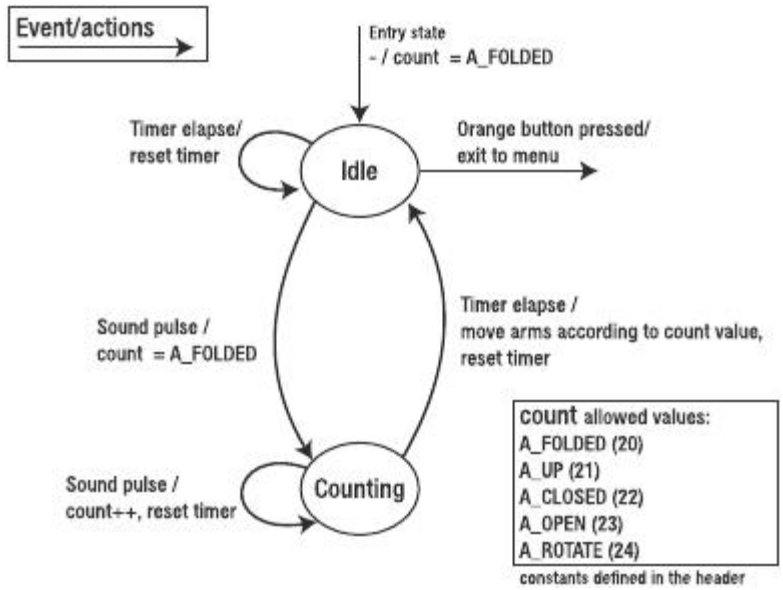



**Figure 8-9.** *The sound counting FSM diagram*

Its implementation code, whose outline was included in Listing 8-2, is reported in Listing 8-3.

**Listing 8-3.** *The Sound Counting FSM Implementation, to Control JohnNXT's Arms with Sounds*

```
////////////////////////////////////////////////////////////////////////////////
//                                MANIPULATION                                //
////////////////////////////////////////////////////////////////////////////////

#define E_EXIT 1
#define E_TRIGGER 2
#define E_ELAPSE 3
#define E_TIMEOUT 1300
#define FSM_IDLE 0
#define FSM_COUNTING 1

int MeasureNoise ()
{
  int n;
  //estimate ambient noise by averaging 10 readings
  n = 0;
  repeat(10)
  {
     n += Sensor(MIC);
     Wait(20);
  }
  n /= 10;
  return n;
}

// this function waits for one of three events:
//  - someone clicks the orange button
//  - the timer elapses
//  - a loud sound pulse occurs
// and returns a number to describe which event
// occurred first
short WaitEvent(short noise, unsigned long timer)
{
   short event = 0;
   while ( event==0 )
   {
      if (ButtonPressed(BTNCENTER,true)==1)
      {
         event = E_EXIT;
         while(ButtonPressed(BTNCENTER,true)==1);
         TextOut(5,LCD_LINE6,"Button");
      }
      else if (CurrentTick() > timer + E_TIMEOUT)
      {
         event = E_ELAPSE;
         TextOut(5,LCD_LINE6,"Elapse");
```

```
        }
        else if (Sensor(MIC)>noise+50)
        {
            event = E_TRIGGER;
            TextOut(5,LCD_LINE6,"*");
            until(Sensor(MIC)<noise+40);
            Wait(10);
        }
    }
    ClearLine(6);
    return event;
}
// this subroutine shows the actual state of the arms FSM
sub ShowArmsState(short state, short line)
{
    ClearLine(line);
    switch(state)
    {
        case A_FOLDED: TextOut(5,(8-line)*8,"Folded"); break;
        case A_UP: TextOut(5,(8-line)*8,"Up"); break;
        case A_CLOSED: TextOut(5,(8-line)*8,"Hands closed"); break;
        case A_OPEN: TextOut(5,(8-line)*8,"Hands open"); break;
        case A_ROTATE: TextOut(5,(8-line)*8,"Wrist rotate"); break;
    }
}

// this subroutine manages the arms FSM, to command the arms with
// sound pulses
sub J5_Manipulation()
{
    unsigned long soundFSMtimer;
    short count = A_FOLDED;
    short state = FSM_IDLE;
    short event;
    bool exit;
    short noise;
    ClearScreen();
    TextOut(0,LCD_LINE1,"Arms control");
    // measure background noise
    noise = MeasureNoise();
    Remote(A_FOLDED,ACK_DONE);
    // the sound counting FSM is implemented as follows
    until(exit)
    {
        if (state == FSM_IDLE)
        {
```

```
            // wait for an event
            event = WaitEvent(noise,soundFSMtimer);
            // perform the actions for this state
            if (event == E_TRIGGER)
            {
              state = FSM_COUNTING;
              count = A_FOLDED;
            }
            else if (event == E_ELAPSE)
            {
              // reset the timer
              soundFSMtimer = CurrentTick();
            }
            else if (event == E_EXIT) exit = true;
          }
          else if (state == FSM_COUNTING)
          {
            // displays information onscreen line 3
            ShowArmsState(count,3);
            // wait for an event
            event = WaitEvent(noise,soundFSMtimer);
            // perform the actions for this state
            if (event == E_TRIGGER)
            {
              count++;
              if (count>A_ROTATE) count = A_ROTATE;
              // reset the timer
              soundFSMtimer = CurrentTick();
            }
            else if (event == E_ELAPSE)
            {
              // reset the timer
              soundFSMtimer = CurrentTick();
              Remote(count,ACK_DONE);
              state = FSM_IDLE;
            }
            else if (event == E_EXIT) exit = true;
          }
        }
}
```

The J5_Manipulation() subroutine is called by the principal JohnNXT FSM from the main task, when you choose the Arms Control item from the JohnNXT menu. This subroutine starts measuring the average ambient noise by calling the MeasureNoise() function, which reads the Sound Sensor ten times and then computes the arithmetic average.

■ **Tip** Averaging many measurements over time is useful for lowering the uncertainty of the whole measurement. For example, imagine making a single measurement of the sound level, right when an unpredictable loud noise occurs. You would get a bad estimation of the real environment's background noise: in fact, you expected to get a low percentage value, but your single measurement has returned a high value, because of that sudden loud noise. In robotics, it's a bad habit to trust a single measurement. To avoid having a single measurement corrupt your estimation with its high uncertainty, it's common use to average many measurements over time. Every time you make a new measurement, it contributes to lowering the estimation's uncertainty.

To compute the average value, you simply sum these ten readings in the variable n (initially set to zero) and then divide it by ten. This noise measurement is then used as a threshold to detect loud sounds, such as hand claps or whistles.

As previously said, the FSM in Figure 8-9 is implemented in the J5_Manipulation() subroutine. The code's meaning is straightforward. In both the two states, the WaitEvent function detects the events. This function waits for one of three kinds of events, by returning the corresponding opcode to the FSM: the orange NXT button click that causes the whole routine to exit, the timer elapsing, or a loud sound's detection.

In the beginning, the FSM is in IDLE mode, and there it remains until a loud sound is detected. The count variable is initialized with the A_FOLDED value. The first sound detected is an event that causes the state to switch from IDLE to COUNTING, and the count variable to assume the value A_FOLDED, which is the first value among the constants that describe the arms' state.

While in the COUNTING state, a new sound detection is an event that causes the count variable to be incremented by one (limited by the maximum value of A_ROTATE) and the timer to be reset. If you stop clapping, the timer elapses, and the arms are actuated according to the count variable's value. The master actuates the arms by calling the Remote(count, ACK_DONE) function, which sends the count variable value as a command to the slave NXT. The count variable can assume one of the values of the possible commands for the arms: remember that the constants in capital letters are indeed aliases for numerical values, declared in JohnNXT's header file J5Defs.nxc. So, by passing count to the Remote function, you're just telling the FSM implemented in the slave NXT program to bring the arms into one of their possible states. After the arms' actuation, the state comes back to IDLE and the timer is reset. If the timer elapses while the FSM is in IDLE mode, it does not cause any action.

If you were paying attention, you would notice that the code that implements the timer is the same as I described in Chapter 6, using the soundFSMtimer variable and the CurrentTick() NXC API function. Now you know the internals of the sound counting FSM.

# JohnNXT Programming Guide

The programs provided are just a start! You can write your own custom programs for JohnNXT, to make him do whatever you want. You could develop an articulated master program to give JohnNXT a real autonomous behavior, or you could use this elegant hardware platform to attempt some new software experimentation. To help you in this task, I have provided here a short programming guide, to let you exploit all the ready-to-use functions to make JohnNXT's

parts move, so you won't have to worry about the low-level mechanical hassles. Whatever you write, keep in mind that you don't need to modify the slave program.

Your work can be focused on the master program by following some directions. First of all, you must have the files shown in Table 8-1 in the same folder; you must include them in the master program's code using the #include directive. You can compile and run some subfiles as stand-alone, because they contain a main task that is compiled only when the file is not included in the JohnNXT master program.

**Table 8-1.** *The Subfiles That Must Always Be Included in the Master NXT Program*

| File | Functions Provided | Can Run As Stand-Alone |
|---|---|---|
| J5Defs.nxc | Contains all the definitions and macros for JohnNXT | No |
| HSlib.nxc | This is the High Speed Communication Library to use high-speed wired communication | No |
| J5_head.nxc | Contains the FSM to manage head movements | Yes |
| J5_comm.nxc | Contains the Remote function to send commands to the slave NXT | No |
| J5_handflw.nxc | Contains the hand-following algorithm | Yes |
| J5_lineflw.nxc | Contains the line-following algorithm | Yes |
| J5_remote.nxc | Receives commands from the remote control | Yes |
| J5_show.nxc | Performs the show, featuring all the actions available | Yes |
| J5_sounds.nxc | Plays the sound files containing J5 phrases | No |

You can use Table 8-2 as a quick guide for the master NXT Input/Output devices' aliases defined in the JohnNXT header file (J5Defs.nxc). Table 8-4 contains the definitions for the slave NXT. If you have any doubts about constants or ports, please check the JohnNXT definitions header file.

**Table 8-2.** *Master NXT Constant Definitions for Motors and Sensors*

| Device | Port | Alias |
|---|---|---|
| Right tread motor | OUT_A | R_TREAD |
| Left tread motor | OUT_C | L_TREAD |
| Both treads | OUT_AC | TREADS |
| Head motor | OUT_B | HEAD |
| Sound Sensor | IN_1 | MIC |
| Line-tracking sensor | IN_2 | LIGHT |
| Ultrasonic Sensor | IN_3 | EYES |
| High-speed serial port | IN_4 | COMM |

While reading the following paragraphs, I recommend you have a look at the source code of the various subfiles. The code to control the treads is the master program; to know where to find the code that controls a particular feature, check Table 8-1.

## Moving the Treads

The tread motors are connected to the master NXT and are the simplest to use. You can control them by using the common OnRev and OnRevSync statements to let JohnNXT go forward and turn, or the OnFwd and OnFwdSync statements to go in reverse. The inversion of directions is due to the particular motor's position and gearing. To stop the treads, just use Off or Float, to brake or to stop them gently, respectively.

## Moving the Head

The head motor is connected to the master NXT. To move the head, I advise you against directly turning the motor on and off, but to use instead the FSM-regulated function that I wrote. In fact, if you moved the head with the basic OnFwd/OnRev functions, you would lose control of the head's position, and you wouldn't know if the neck had reached its turning limits. You can use the Head function after having included the file J5_head.nxc.

To move the head, use the following code, where direction can be CENTER, RIGHT, or LEFT, and the scale_factor can be a small integer number (normally 1):

```
Head ( direction, scale_factor)
```

The scale_factor is a number used as a divisor to turn the head less than the normal angle. For example, a scale_factor equal to 1 makes the head rotate by the total range, while a scale_factor equal to 2 makes the head rotate at an angle that is half the total excursion. Unless you need small movements, always keep this number equal to 1.

To reset the head to the center at the beginning of a program, write the following code:

```
head_state = UNKNOWN;
Head(CENTER,1);
```

The head_state variable is the one that is used to keep track of the position of the head. So, you must not assign it a value directly in your programs, as you would mess up the way the head FSM works. As the only exception to this rule, you need to change the head_state value explicitly when you want to center the head using the torque sensing limit switch. In fact, if the state is UNKNOWN when the Head(CENTER,1) is called, this function uses the procedure to center the head based on motor torque sensing. It turns the head to the right, until the motor feels blocked, and then rotates the head back to the center. After this initial reset, the Head subroutine itself changes the head_state variable, so you don't have to change it manually anymore. To turn the head left or right, simply use Head(LEFT,1) or Head(RIGHT,1), respectively. By calling Head(CENTER,1) again, the head is brought to the center without using the reset routine, because the head_state variable is assigned a value different from UNKNOWN. The head is rotated by a precise number of degrees, according to the head_angles decision table.

## Playing Sounds

You must download sound files for JohnNXT into the master NXT by using BricxCC (see Appendix A). To play sounds, use

```
Sound (sound, wait_completion)
```

where sound is one of the constants in Table 8-3 and where the wait_completion can be true (the program waits until the sound has been completely executed) or false (the program starts the sound and continues at once).

**Table 8-3.** *Opcodes to Play JohnNXT Sounds*

| Action | Constant |
|---|---|
| Play "Number Five is alive!" | S_ALIVE1 |
| Play "I am alive." | S_ALIVE2 |
| Play "Yeah! Johnny 5! That's cool!" | S_COOL |
| Play "Hello, bozos!" | S_HELLO |
| Play "Need input!" | S_INPUT |
| Play "Malfunction!" | S_MALFUNCTION |
| Play soundtrack loop | S_LOOP |
| Stop sounds | S_NONE |
| Play one of the preceding sounds randomly | S_RANDOM |

## Moving the Slave NXT Motors

The torso, arms, and laser motors are connected to the slave NXT on the ports specified in Table 8-4.

**Table 8-4.** *Slave NXT Constant Definitions for Motors and Sensors*

| Device | Port | Alias |
|---|---|---|
| Torso motor | OUT_A | TORSO |
| Arms' motor | OUT_B | ARMS |
| Laser treads | OUT_C | LASER |
| Laser-tip Light Sensor | IN_1 | LASER_TIP |

You can control those motors by calling the following function (implemented inside J5_comm.nxc) within JohnNXT's master program:

```
Remote (opcode, ack)
```

opcode is one of the constants in Table 8-5, and ack can be either ACK_RX or ACK_DONE. With ACK_RX, the master program just waits for the slave to receive the command, while with ACK_DONE, the master program hangs on until the slave has finished the action it was told to do. The constants listed in Table 8-5 are used in the slave program both as commands to be executed and as state descriptions.

**Table 8-5.** *Constant Definitions to Move Torso, Arms, and Laser*

| Output Device | Action | Constant |
|---|---|---|
| Torso motor | Lower torso to downmost position using torque sensing | T_DOWN |
| Torso motor | Bring torso to middle position | T_MID |
| Torso motor | Bring torso to up position | T_UP |
| Torso motor | Bring torso to upmost position | T_UPMOST |
| Torso motor | Move torso a step up | T_STEPUP |
| Torso motor | Move torso a step down | T_STEPDOWN |
| Arms motor | Fold the arms | A_FOLDED |
| Arms motor | Move forearms up | A_UP |
| Arms motor | Close hands | A_CLOSED |
| Arms motor | Open hands | A_OPEN |
| Arms motor | Rotate wrists | A_ROTATE |
| Arms motor | Step towards open hands state | A_STEP_OPEN |
| Arms motor | Step towards folded arms state | A_STEP_CLOSE |
| Laser motor | Raise laser | L_UP |
| Laser motor | Lower laser | L_DOWN |
| Laser light | Turn laser tip on | L_ON |
| Laser light | Turn laser tip off | L_OFF |
| Laser light | Blink laser tip | L_BLINK |

This concludes the programming guide, a sort of handy Software Development Kit (SDK) intended for those who want to write custom programs for JohnNXT. Now that you know what's running inside JohnNXT's brain, it's time to put him together.

# Building JohnNXT

Constructing JohnNXT takes quite a long time and more than a thousand LEGO elements. As stated at the beginning of this chapter, you need more than two NXT sets' parts, because many of them are not included in the standard set—just to name a few, the neck pistons, the tread links, and the arms' chain links. The complete bill of materials is shown in Figure 8-10, and the textual list in Table 8-6.

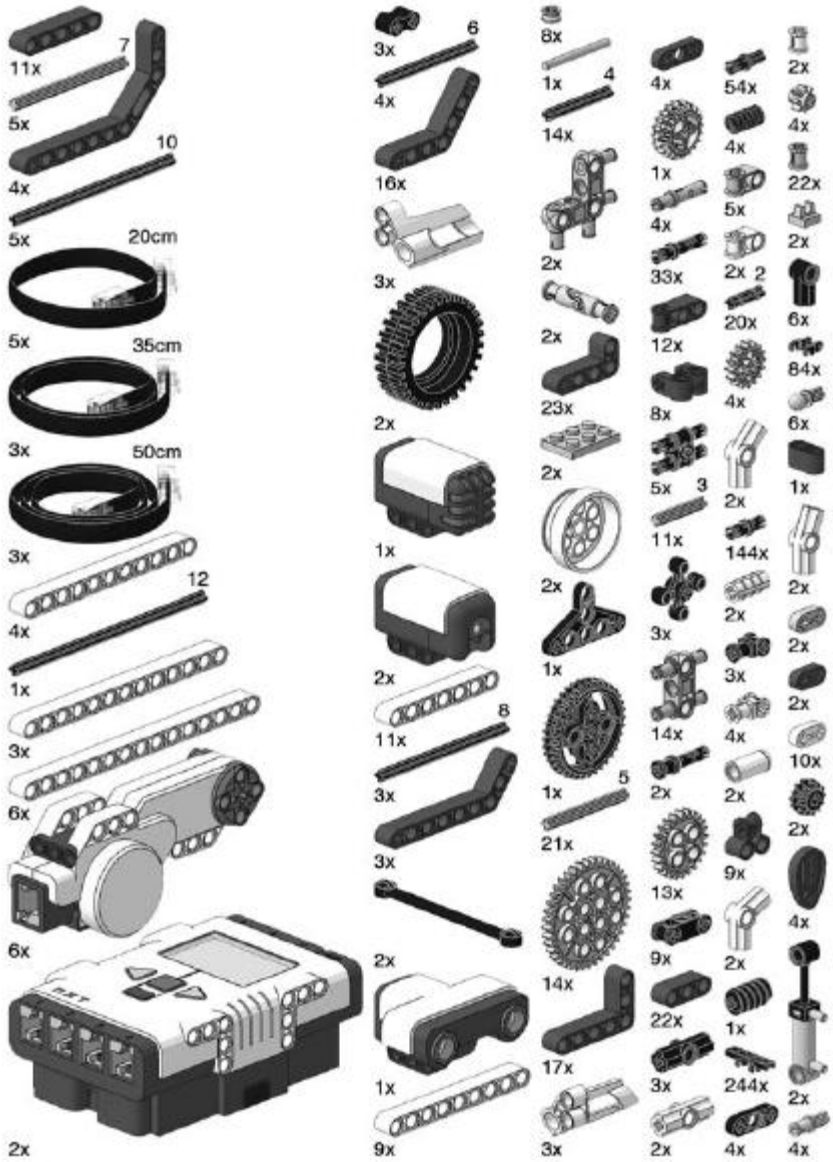



**Figure 8-10.** *The bill of materials for JohnNXT*

**Table 8-6.** *JohnNXT Bill of Materials*

| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 11 | Dark gray | 32316.DAT | TECHNIC Beam 5 |
| 5 | Light gray | 44294.DAT | TECHNIC Axle 7 |
| 4 | Dark gray | 32009.DAT | TECHNIC Beam Liftarm Bent 45 Double |
| 5 | Black | 3737.DAT | TECHNIC Axle 10 |
| 5 | | 55804.DAT | Electric Cable NXT 20cm |
| 3 | | 55805.DAT | Electric Cable NXT 35cm |
| 3 | | 55806.DAT | Electric Cable NXT 50cm |
| 11 | White | 32525.DAT | TECHNIC Beam 11 |
| 1 | Black | 3708.DAT | TECHNIC Axle 12 |
| 3 | White | 41239.DAT | TECHNIC Beam 13 |
| 6 | White | 32278.DAT | TECHNIC Beam 15 |
| 6 | | 53787.DAT | MINDSTORMS NXT Motor |
| 2 | | 53788.DAT | MINDSTORMS NXT |
| 3 | Black | 45590.DAT | TECHNIC Axle Joiner Double Flexible |
| 4 | Black | 3706.DAT | TECHNIC Axle 6 |
| 16 | Dark gray | 32348.DAT | TECHNIC Beam 7 Liftarm Bent 53.5 (4:4) |
| 3 | White | 32527.DAT | TECHNIC Panel Fairing #5 |
| 2 | Black | 2696.DAT | Tire Model Team |
| 1 | | 55963.DAT | Electric MINDSTORMS NXT Sound Sensor |
| 2 | | 55969.DAT | Electric MINDSTORMS NXT Light Sensor |
| 11 | White | 32524.DAT | TECHNIC Beam 7 |
| 3 | Black | 3707.DAT | TECHNIC Axle 8 |
| 1 | Dark gray | 32271.DAT | TECHNIC Beam 9 Liftarm Bent 53.5 (7:3) |
| 2 | Black | 32293.DAT | TECHNIC Steering Link 9L |
| 1 | | 53792.DAT | MINDSTORMS NXT Ultrasonic Sensor |
| 9 | White | 40490.DAT | TECHNIC Beam 9 |
| 8 | Light gray | 32123.DAT | TECHNIC Bush 1/2 Smooth |
| 1 | Light gray | 30374.DAT | Bar 4L Light Saber Blade |
| 14 | Black | 3705.DAT | TECHNIC Axle 4 |
| 2 | Light gray | 55615.DAT | TECHNIC Beam 5 Bent 90 with 4 Pins |
| 2 | Light gray | 9244.DAT | TECHNIC Universal Joint |
| 23 | Dark gray | 32140.DAT | TECHNIC Beam 5 Liftarm Bent 90 (4:2) |
| 2 | Light gray | 3021.DAT | Plate 2 - 3 |
| 2 | White | 2695.DAT | Wheel Model Team |
| 1 | Black | 2905.DAT | TECHNIC Liftarm Triangle |

| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 1 | Black | X344.DAT | TECHNIC Gear 36 Tooth Double Bevel |
| 21 | Light gray | 32073.DAT | TECHNIC Axle 5 |
| 14 | Light gray | 3649.DAT | TECHNIC Gear 40 Tooth |
| 17 | Dark gray | 32526.DAT | TECHNIC Beam 7 Bent 90 (5:3) |
| 3 | White | 32528.DAT | TECHNIC Panel Fairing #6 |
| 4 | Dark gray | 6632.DAT | TECHNIC Beam 3 - 0.5 Liftarm |
| 1 | Light gray | 32269.DAT | TECHNIC Gear 20 Tooth Double Bevel |
| 4 | Light gray | 32556.DAT | TECHNIC Pin Long |
| 33 | Black | 6558.DAT | TECHNIC Pin Long with Friction and Slot |
| 12 | Dark gray | 42003.DAT | TECHNIC Axle Joiner Perp. with 2 Holes |
| 8 | Dark gray | 41678.DAT | TECHNIC Axle Joiner Perp. Double Split |
| 5 | Black | 32136.DAT | TECHNIC Pin 3L Double |
| 11 | Light gray | 4519.DAT | TECHNIC Axle 3 |
| 3 | Black | 32072.DAT | TECHNIC Knob Wheel |
| 14 | Light gray | 48989.DAT | TECHNIC Axle Joiner Perp. with 4 Pins |
| 2 | Black | 32054.DAT | TECHNIC Pin Long with Stop Bush |
| 13 | Light gray | 3648.DAT | TECHNIC Gear 24 Tooth |
| 9 | Black | 32184.DAT | TECHNIC Axle Joiner Perpendicular 3L |
| 22 | Dark gray | 32523.DAT | TECHNIC Beam 3 |
| 3 | Black | 32034.DAT | TECHNIC Angle Connector #2 |
| 2 | Black | 32034.DAT | TECHNIC Angle Connector #2 |
| 1 | White | 6536.DAT | TECHNIC Axle Joiner Perpendicular |
| 54 | Blue | 43093.DAT | TECHNIC Axle Pin with Friction |
| 4 | Light gray | | TECHNIC Hose 2L |
| 5 | Light gray | 6536.DAT | TECHNIC Axle Joiner Perpendicular |
| 2 | White | 6536.DAT | TECHNIC Axle Joiner Perpendicular |
| 20 | Black | 32062.DAT | TECHNIC Axle 2 Notched |
| 4 | Light gray | 4019.DAT | TECHNIC Gear 16 Tooth |
| 2 | White | 32192.DAT | TECHNIC Angle Connector #4 (135 degree) |
| 144 | Black | 2780.DAT | TECHNIC Pin with Friction and Slots |
| 2 | White | 6538B.DAT | TECHNIC Axle Joiner Offset |
| 3 | Black | 32039.DAT | TECHNIC Connector with Axlehole |
| 4 | White | 32039.DAT | TECHNIC Connector with Axlehole |
| 2 | White | 75535.DAT | TECHNIC Pin Joiner Round |
| 9 | Dark gray | 32291.DAT | TECHNIC Axle Joiner Perp. Double |

*Continued*

**Table 8-6.** *Continued*

| Quantity | Color | Part Number | Part Name |
|---|---|---|---|
| 2 | White | 32015.DAT | TECHNIC Angle Conn. #5 (112.5 degree) |
| 1 | Black | 4716.DAT | TECHNIC Worm Screw |
| 244 | Black | 3873.DAT | TECHNIC Chain Tread |
| 4 | Dark gray | 6632.DAT | TECHNIC Beam 3 - 0.5 Liftarm |
| 2 | White | 3713.DAT | TECHNIC Bush |
| 4 | Light gray | 3647.DAT | TECHNIC Gear 8 Tooth |
| 22 | Light gray | 3713.DAT | TECHNIC Bush |
| 2 | Light gray | 2555.DAT | Tile 1 - 1 with Clip |
| 6 | Black | 32013.DAT | TECHNIC Angle Connector #1 |
| 84 | Black | 3711.DAT | TECHNIC Chain Link |
| 6 | Light gray | 2736.DAT | TECHNIC Axle Towball |
| 1 | Dark gray | 43857.DAT | TECHNIC Beam 2 |
| 2 | White | 32016.DAT | TECHNIC Angle Conn. #3 (157.5 degree) |
| 2 | Light gray | 41677.DAT | TECHNIC Beam 2 - 0.5 Liftarm |
| 2 | Dark gray | 41677.DAT | TECHNIC Beam 2 - 0.5 Liftarm |
| 10 | White | 41677.DAT | TECHNIC Beam 2 - 0.5 Liftarm |
| 2 | Black | 32270.DAT | TECHNIC Gear 12 Tooth Double Bevel |
| 4 | Dark gray | 6575.DAT | TECHNIC Cam |
| 2 | | x253.DAT | TECHNIC Pneumatic Cylinder Small |
| 4 | Tan | 3749.DAT | TECHNIC Axle Pin |

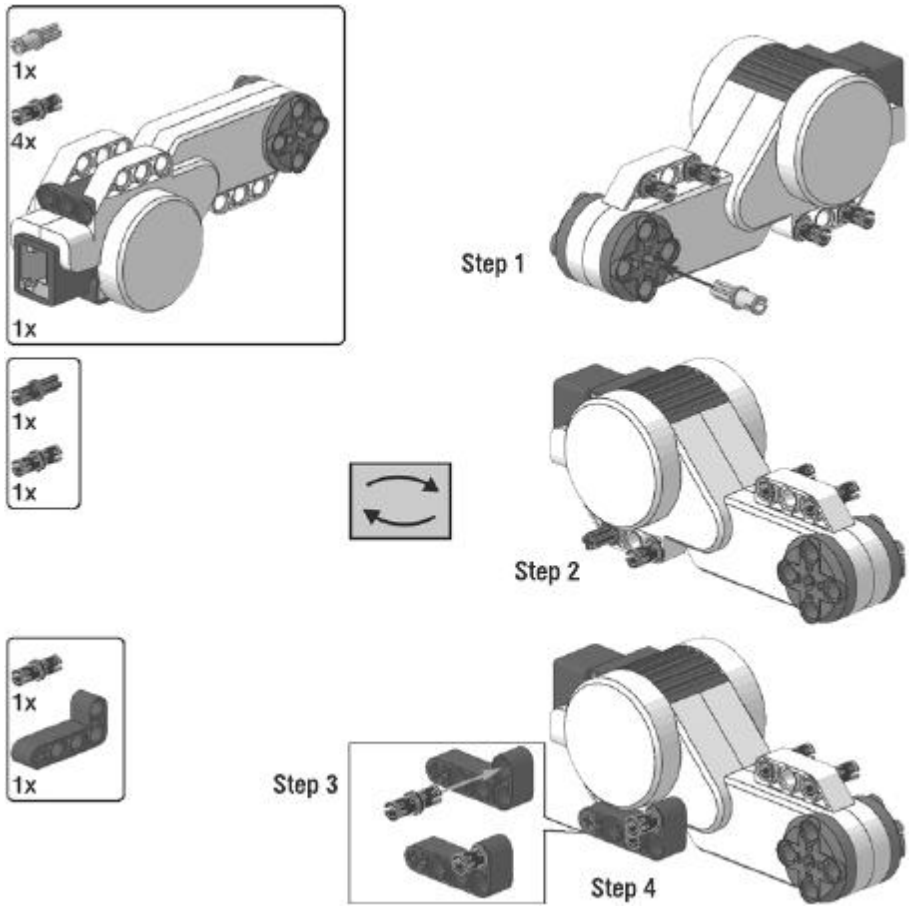
1,042 parts total (more than two NXT retail set parts are needed)

You can obtain LEGO spare parts from LEGO Education (http://www.legoeducation.com/global.aspx), BrickLink (http://www.bricklink.com/), and LEGO Factory Pick a Brick (http://us.factory.lego.com/), even though the TECHNIC section is not well furnished at the moment. Also, a good place to find LEGO spares remains eBay (http://www.ebay.com). For those who live in Italy, a good resource is CampuStore (http://www.campustore.it/lego).

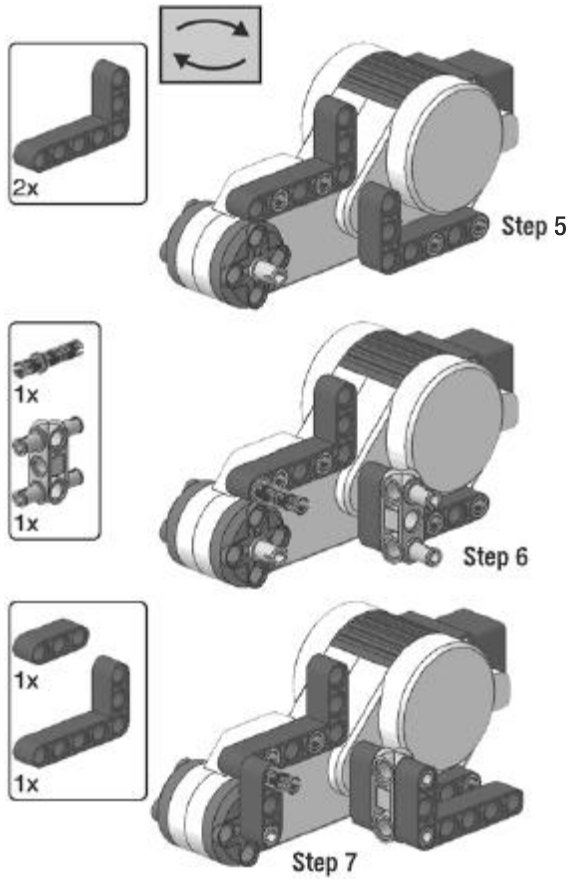
This time, the building phase is more challenging than for the other robots in this book, but once finished, it will give you great satisfaction. Every JohnNXT subassembly will reveal many interesting mechanical aspects for you, and each one will teach you something.

- The torso motor subassembly shows you how to get a liftarm to rotate with a huge torque, to raise heavy loads, such as JohnNXT's body. Here symmetry is important, because loading an axle (or a gear) on just one side would twist it and you could run the risk of breaking it. In this assembly, notice that the geartrain is repeated on both sides of the motor. Also, to avoid ungearings, all the axle-bearing gears that are subject to high torque are kept together by beams.
- The torso subassembly shows you how to use a servomotor as an integral part of a robot. Here, the motor shaft not only drives the arms' mechanism, but also is the pivot for the whole shoulders' frame. The lever system to raise the whole body shows you how to transmit a translational movement beyond a narrow bent structure (the lower body's top joint with the torso).
- The treads' frames teach you how to build extremely strong structures. Triangular frames are in fact crushproof, contrary to the rectangular ones.
- The arms feature the double-drive mechanism explained in the section "Johnnicle: My LEGO Johnny 5 Chronicle" to fold the forearm and grab objects. The poseable shoulder shows how you can use a LEGO universal joint to transmit motion through a bendable joint, on the condition that the hinge is aligned with the universal (cardan) core.
- The parallelogram frame of the upper body shows how to keep complex moving structures always vertical with respect to the ground.
- The laser levers system shows you how to obtain a combined rotational and translational movement.

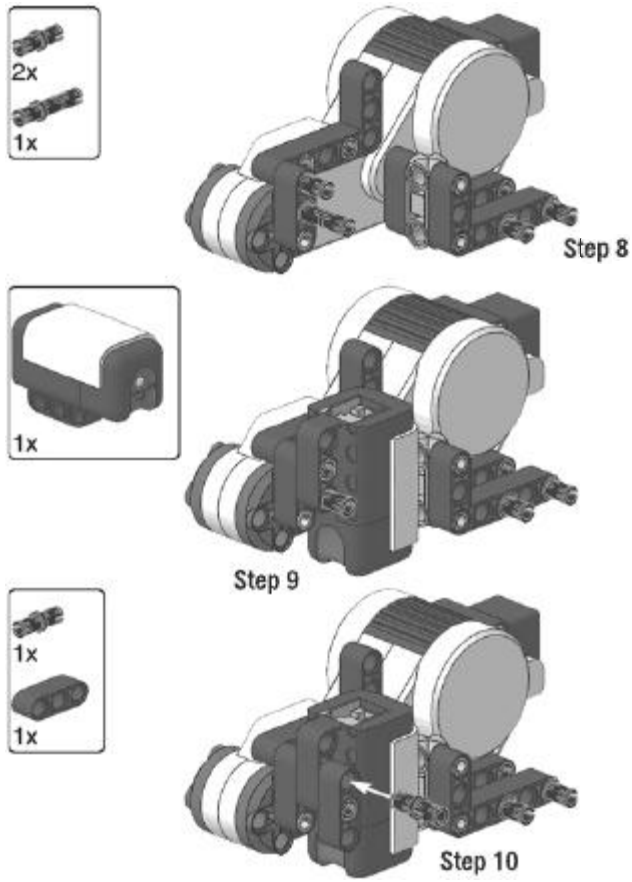
You must insert two LEGO yellow rubber bands in the elbows as shown in Figure 8-11 (steps *a* through *j*). There, you can see the left elbow montage; repeat the same procedure for the right elbow before attaching the arms to the shoulders. Make sure to align the forearms' levers before adding the chain to the gears. If their position is not correct, the final result will be asymmetrical once they are attached to the shoulders' driving axle.



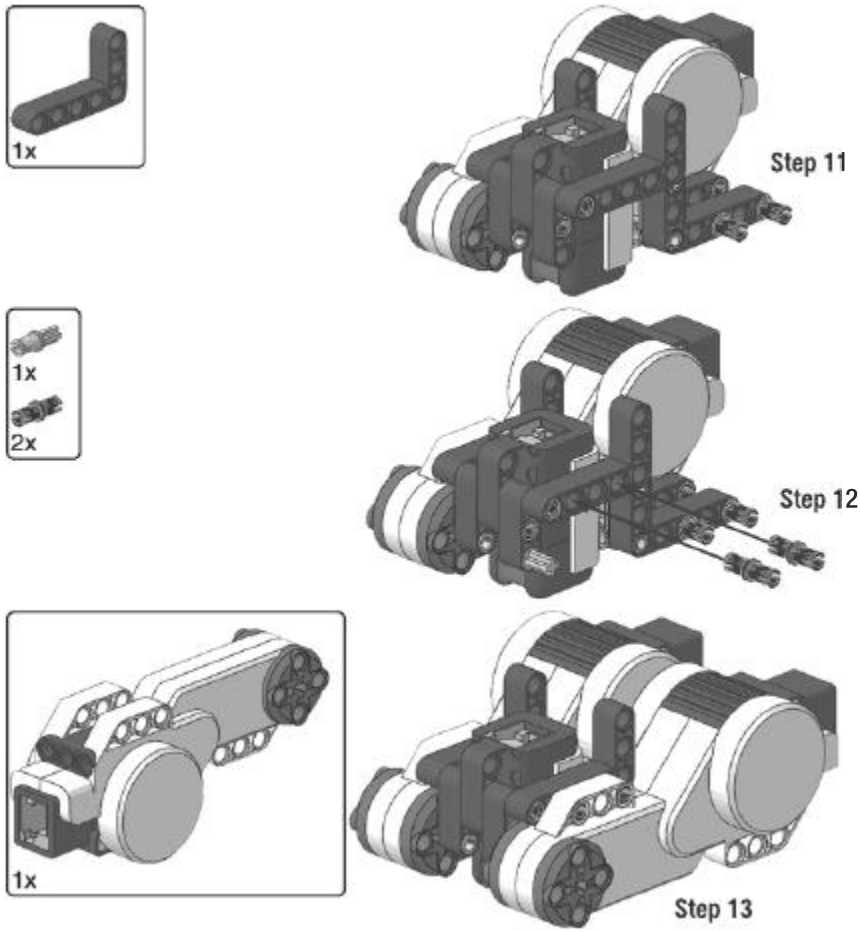

*Start building the treads' motors subassembly. In Step 1, insert a tan axle pin in the motor shaft.*



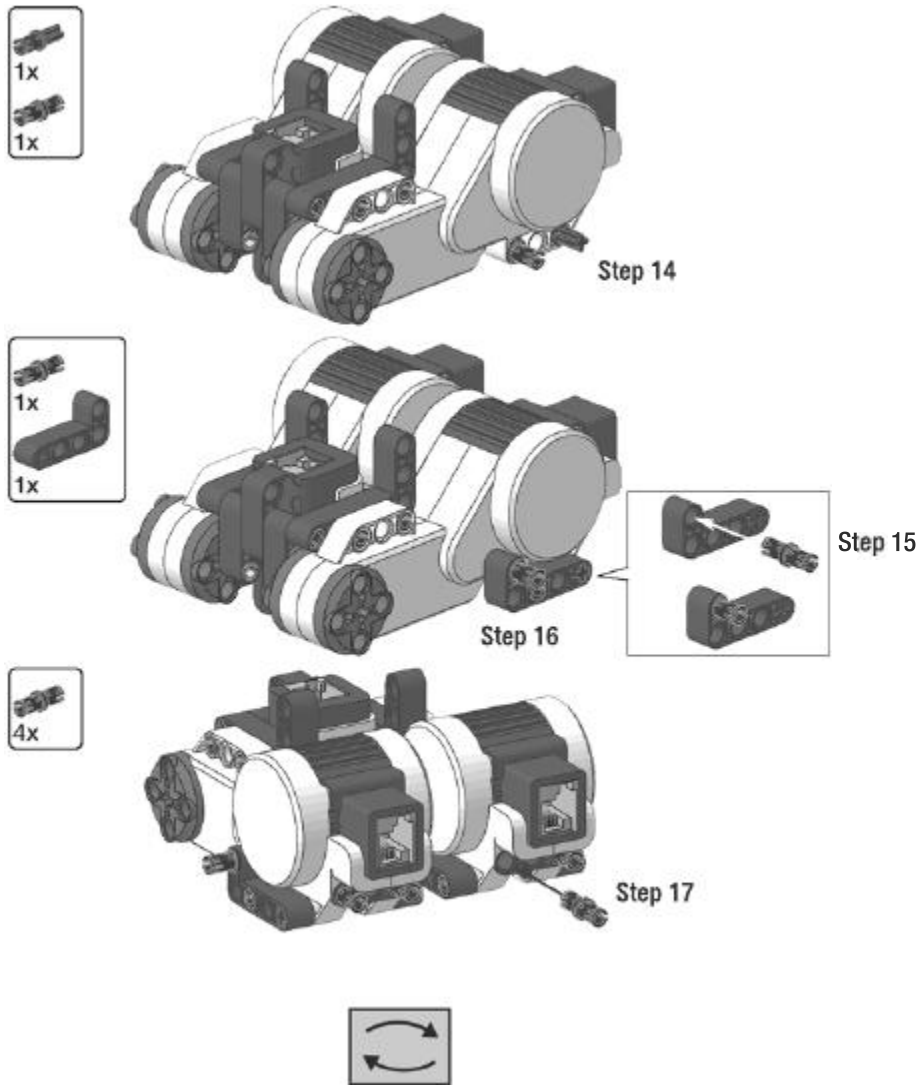

*Build the spacer between the motors.*



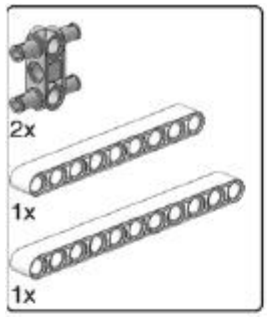

*Add the Light Sensor pointing downwards; it's used to follow lines on the ground.*

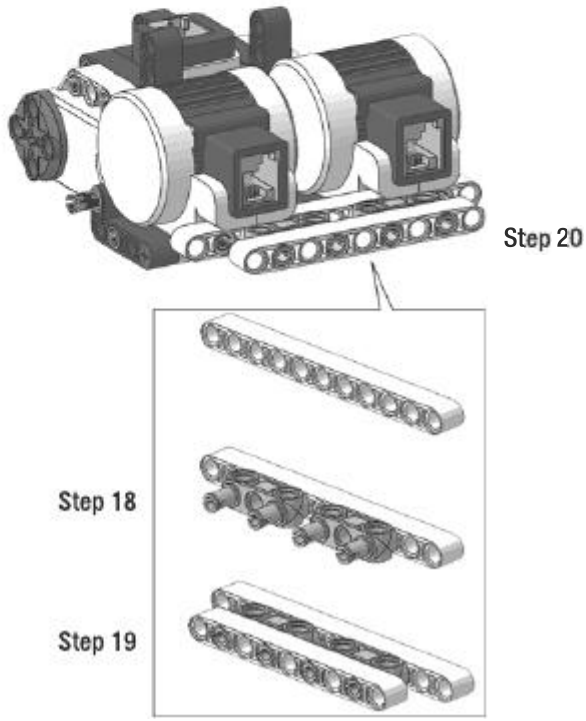



*Finish the sensor holder and add the left tread motor. In Step 12, add two black pins and a tan axle pin.*

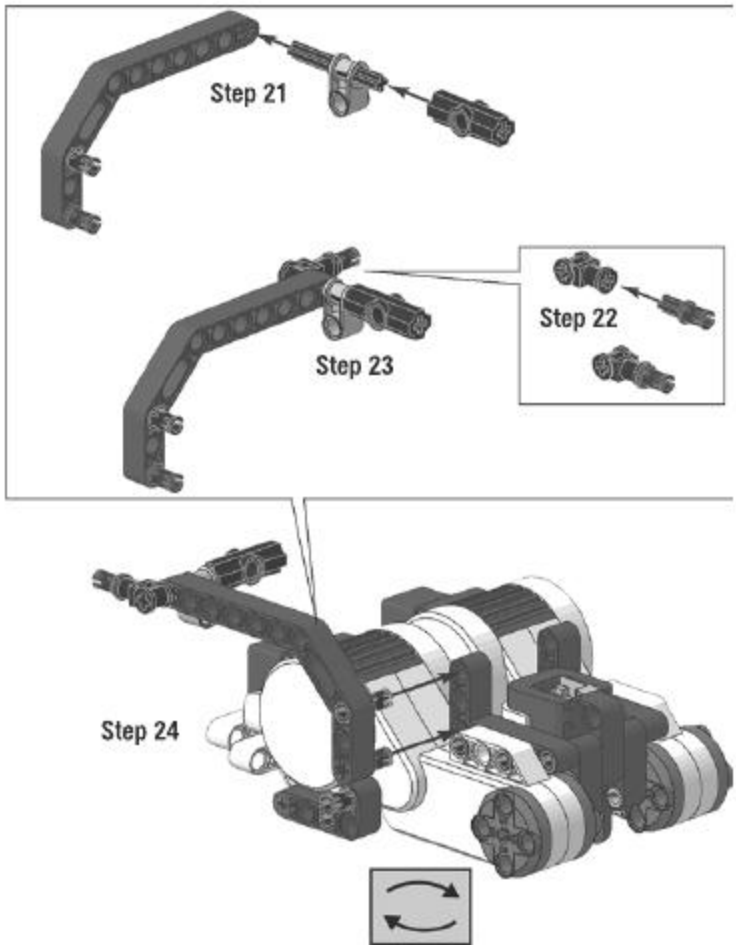



*In Step 17, add four black pins in the back of the motors.*

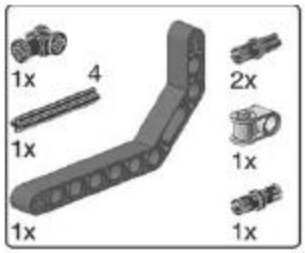


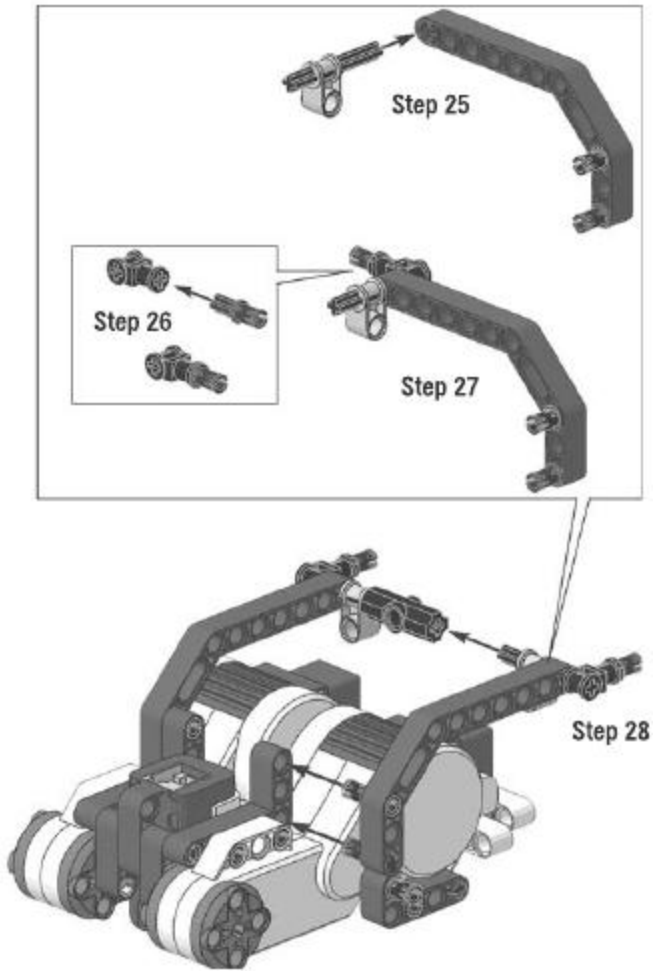




*Add an 11-hole beam and a 9-hole beam, attaching them with the gray Axle Joiners Perpendicular with 4 Pins.*

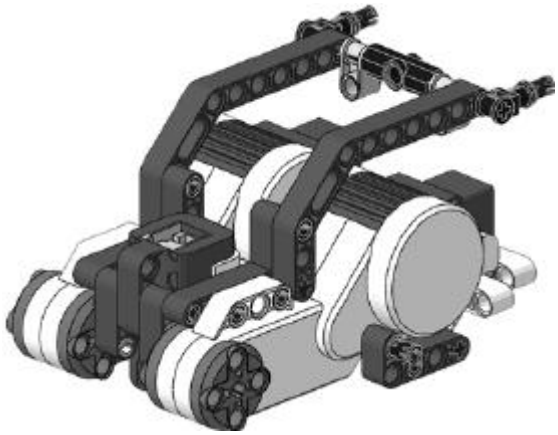




*Add the right part of the frame that will connect the tread motors to the slave NXT.*

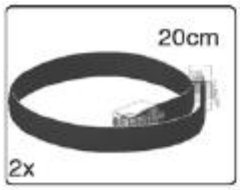


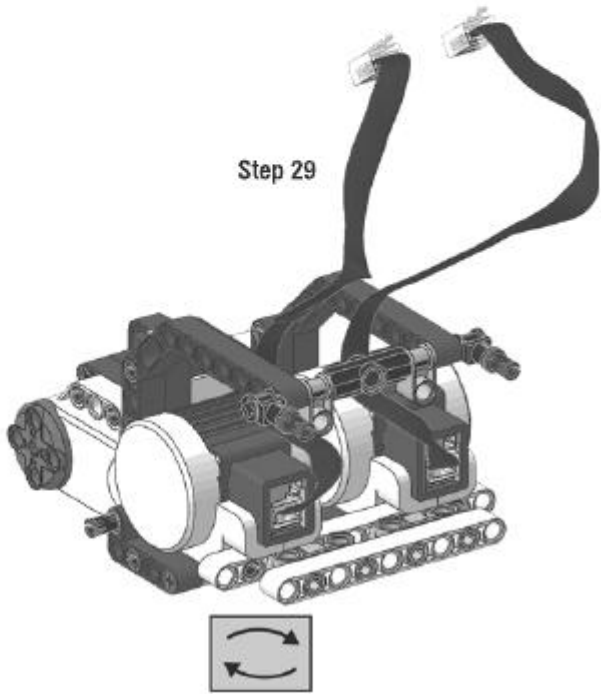




*Complete the frame with its left part.*

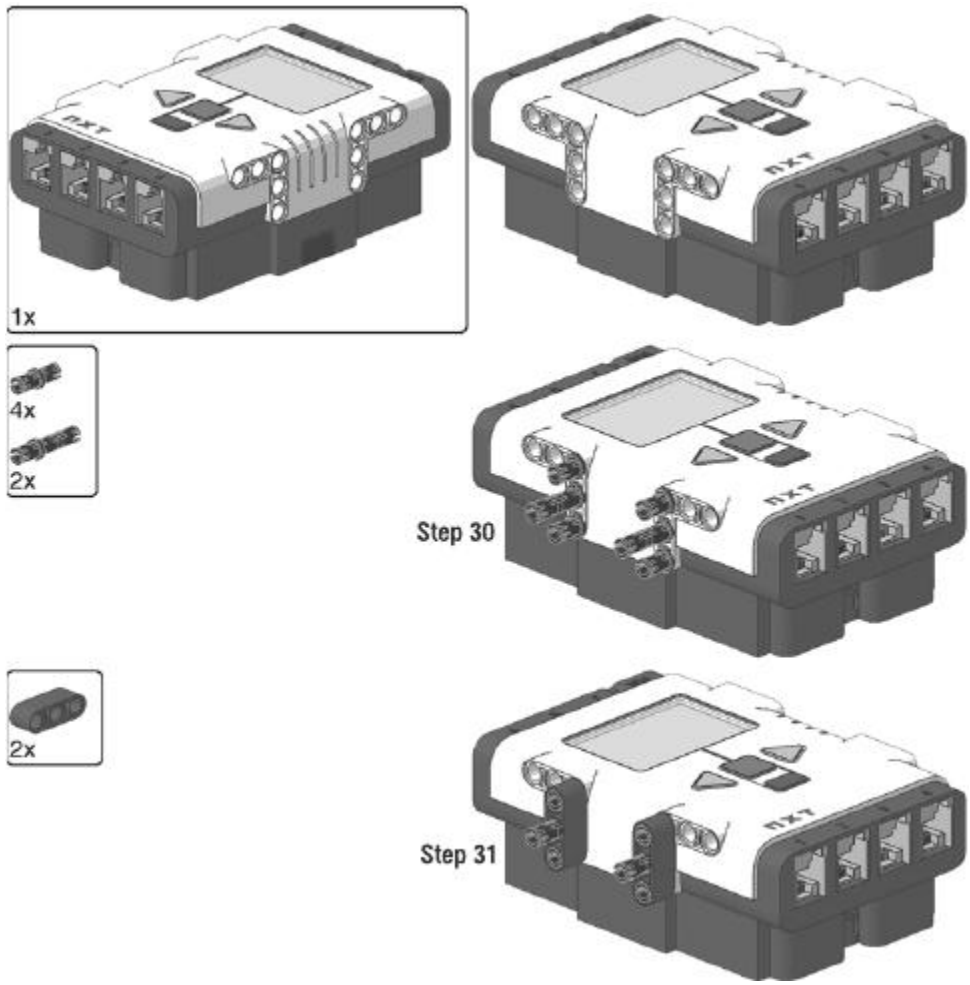


*The treads' motors' subassembly is complete.*

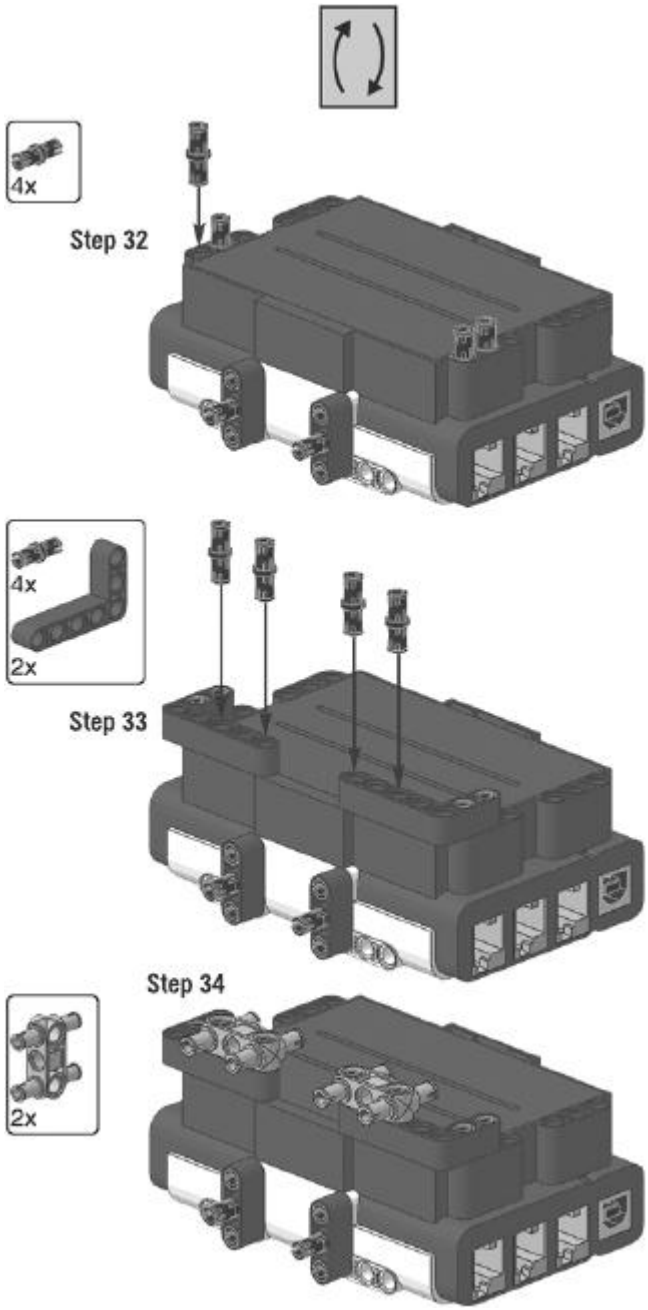


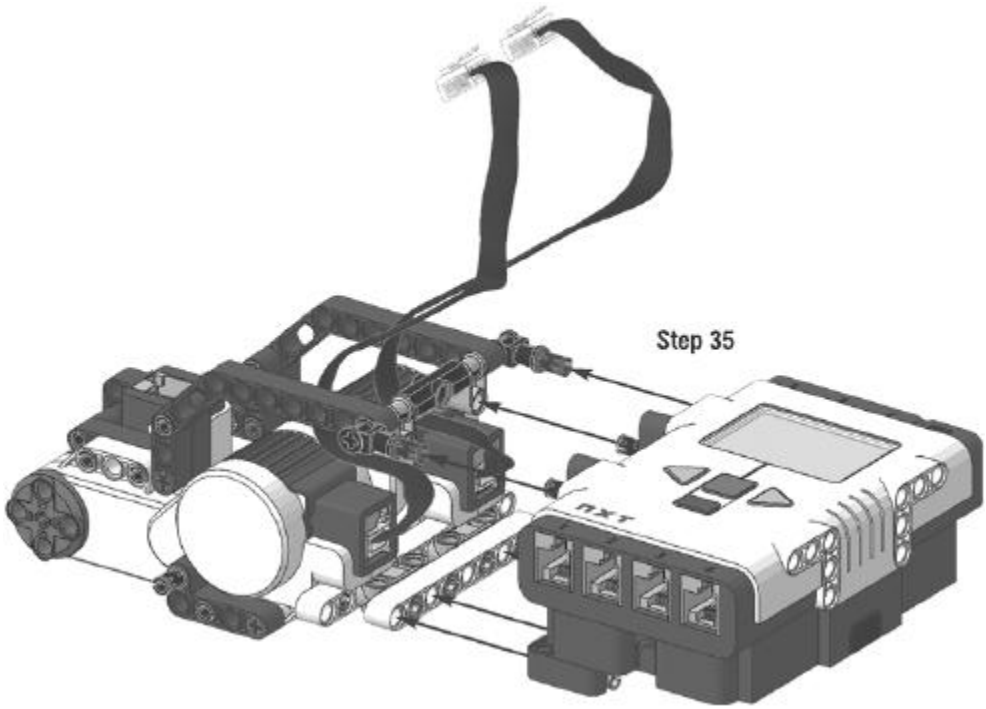

*Add two 20cm (8 inch) cables.*



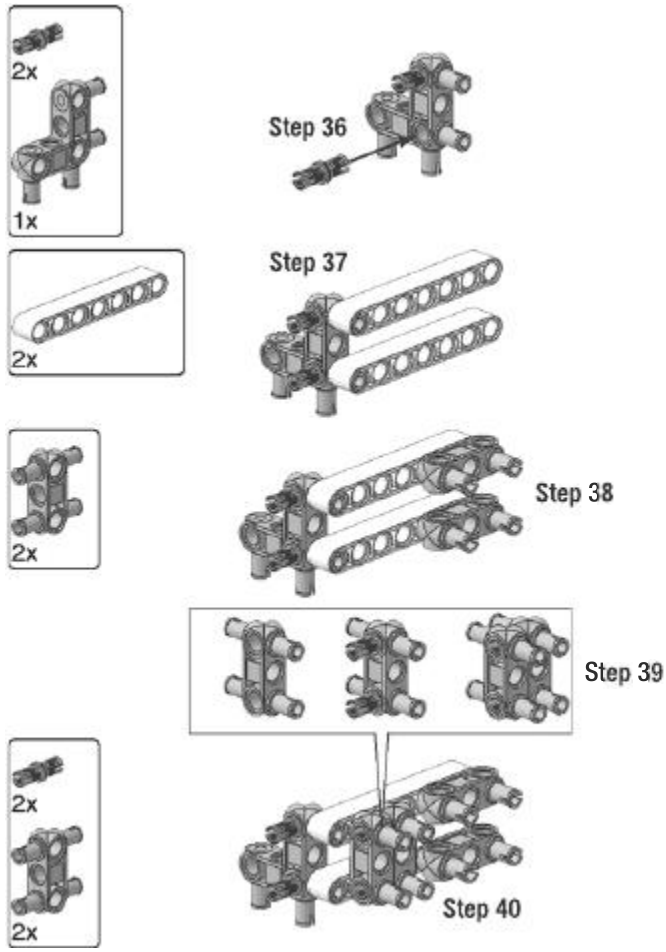

*Start building the slave NXT subassembly.*



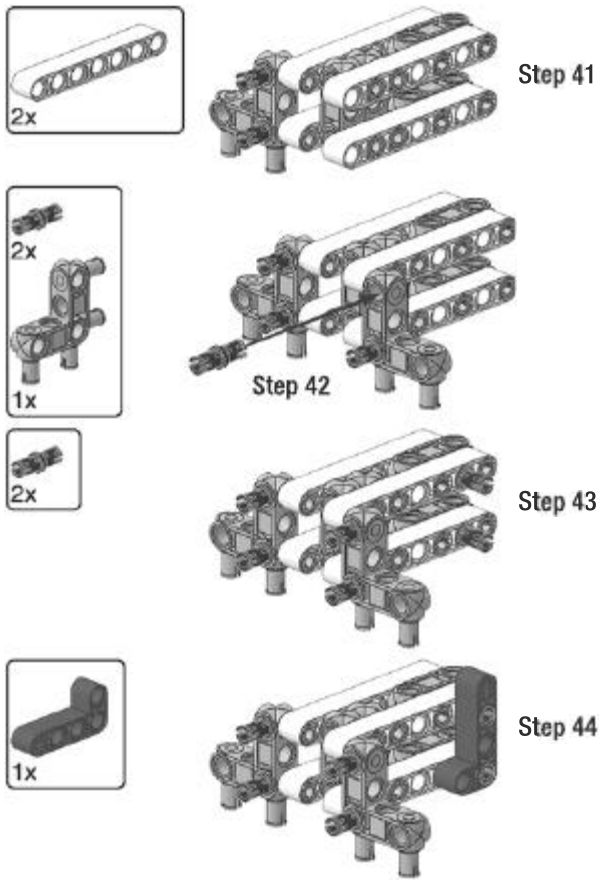

*Turn the NXT upside down and add the parts that will be connected to the rest of the base.*



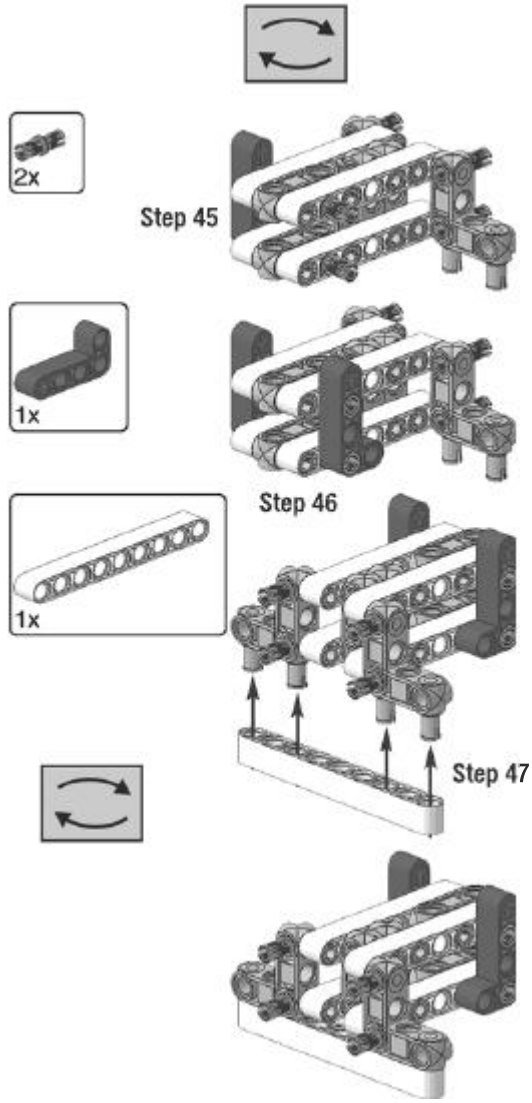

*Attach the NXT to the treads' motors' assembly.*



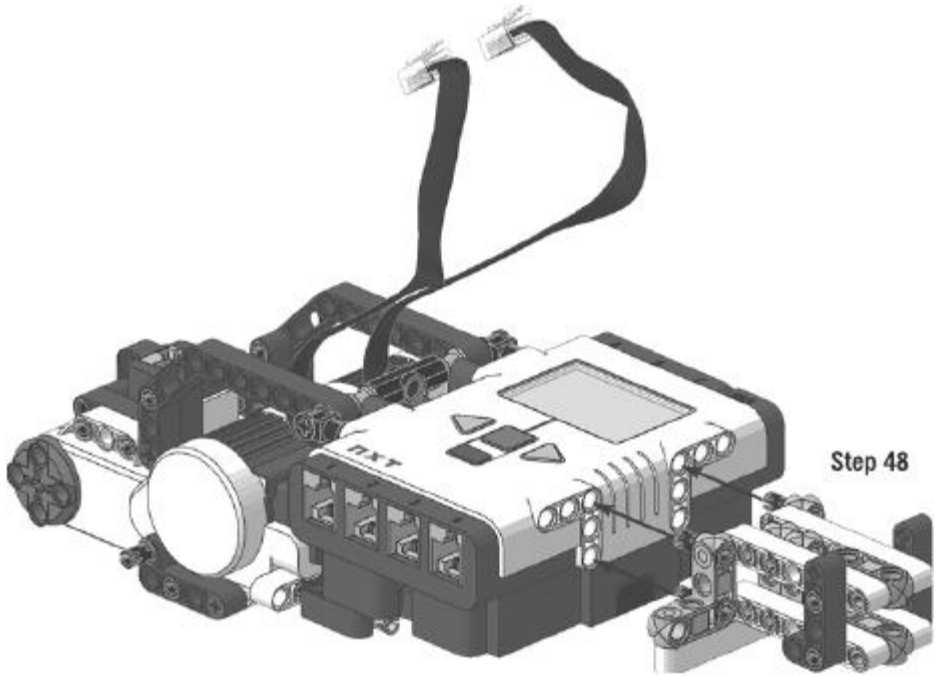

*Start building the passive wheel holder assembly.*



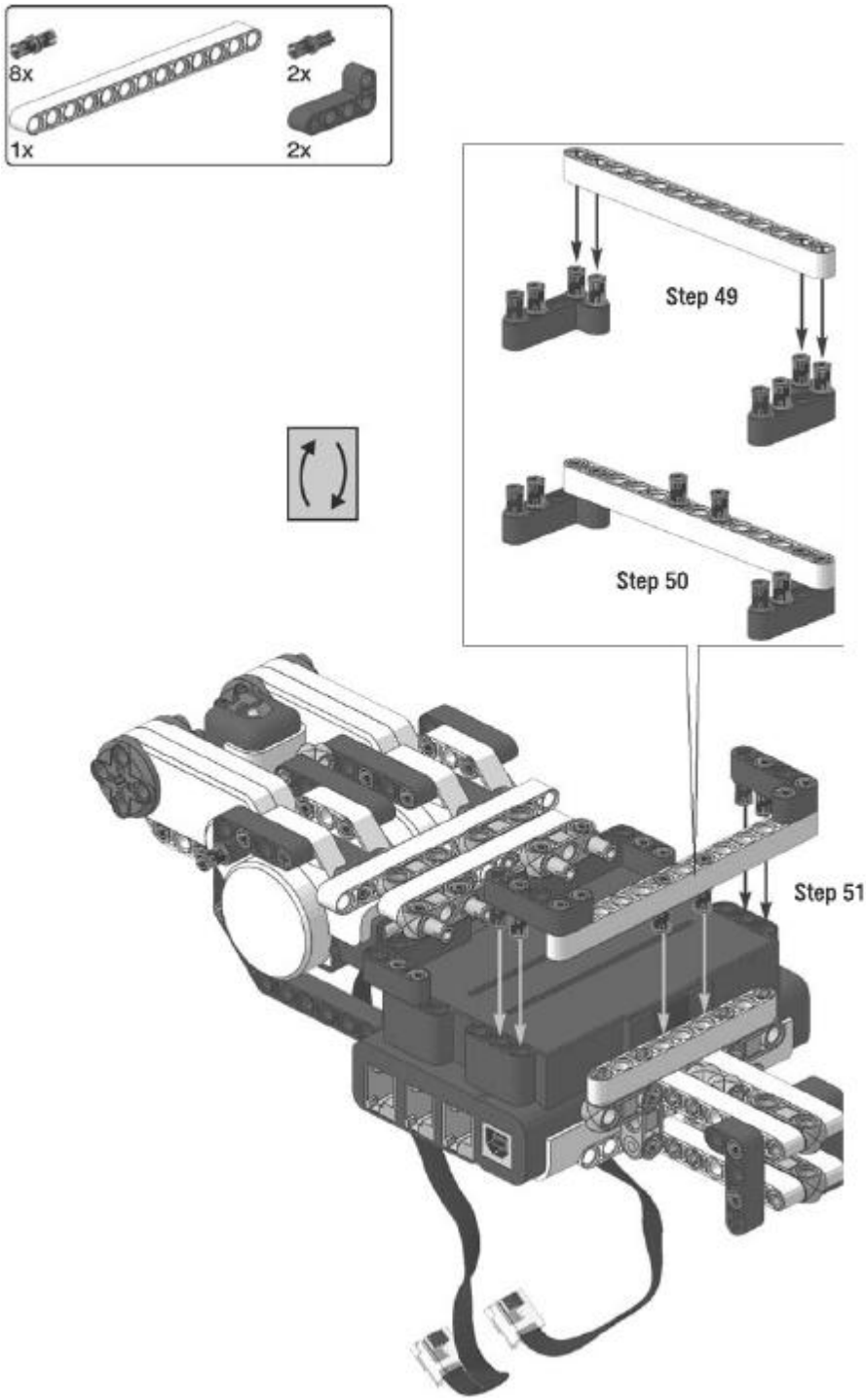

*In Step 44, add the first beam that will be used to support the lower body's lifter mechanism.*



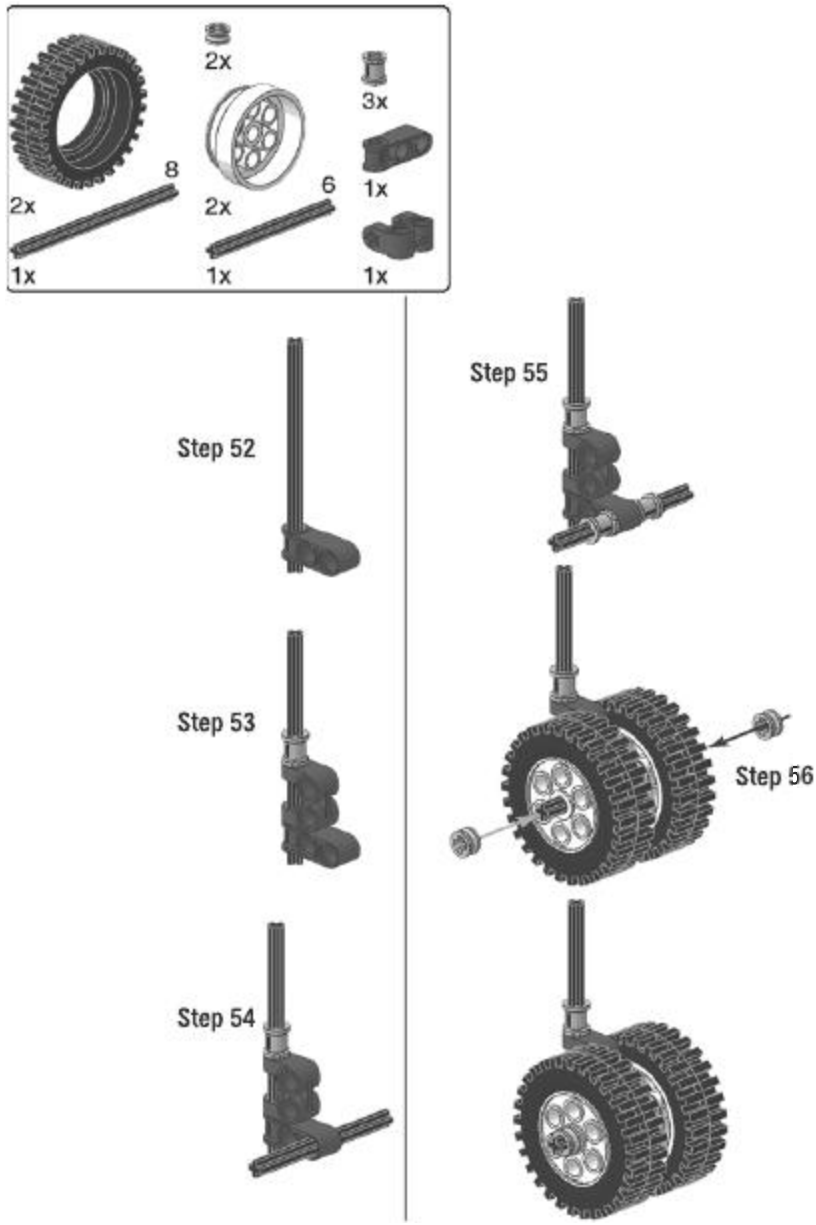

*Complete the passive wheel holder.*



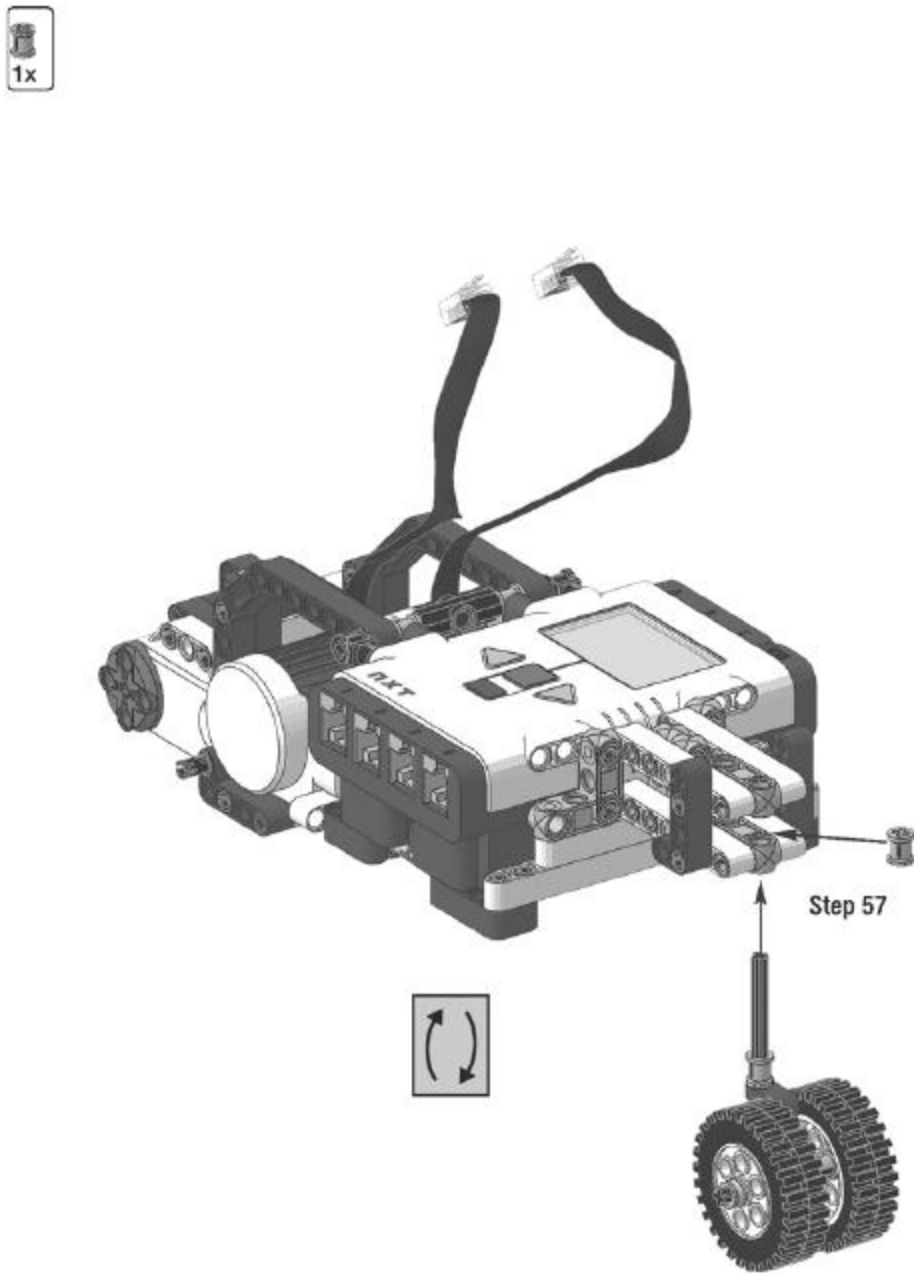

*Attach the passive wheel holder to the rest of the base.*




*Turn the model upside down and build the frame to hold the passive wheel holder firmly with the NXT. Use a 13-long beam.*

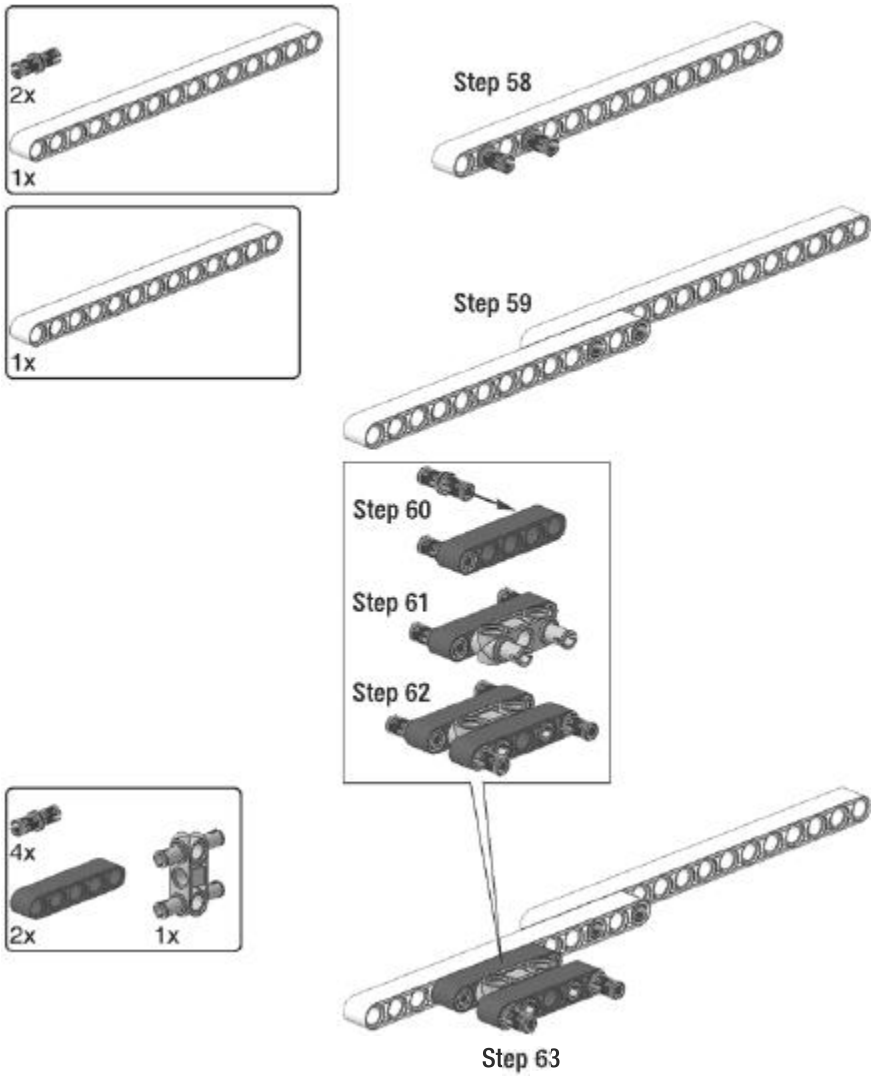



*Build the passive wheel.*

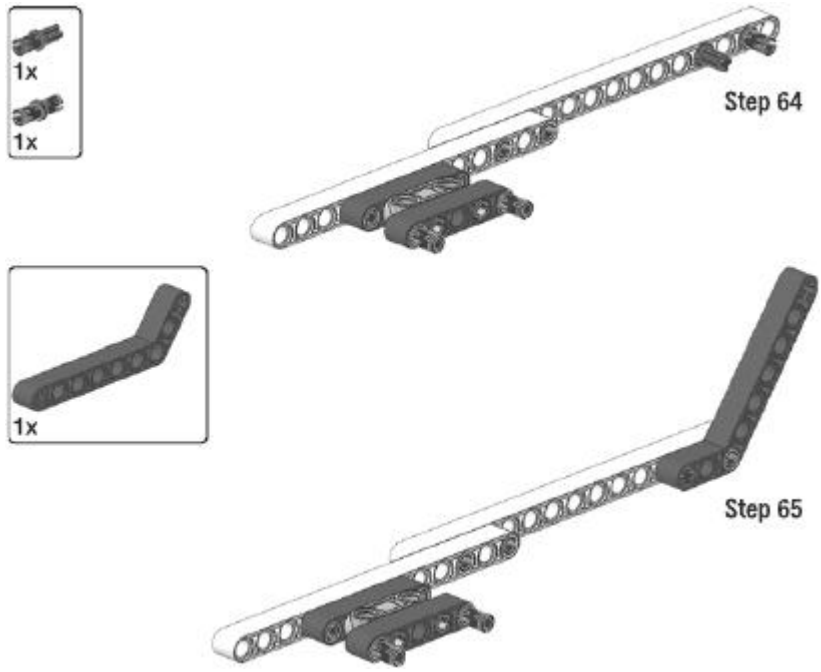



*Insert the passive wheel in place, holding it in place with a bush.*

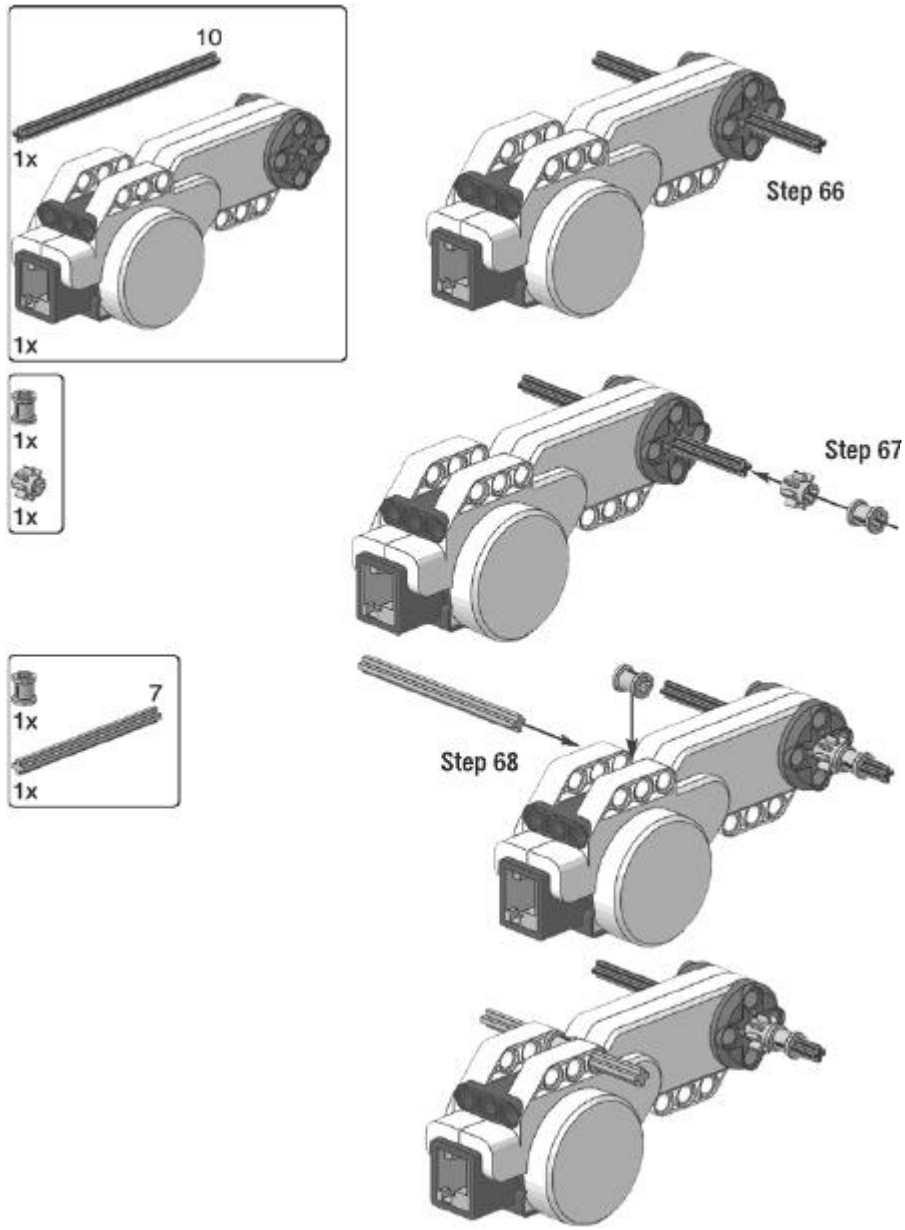

*The base assembly of JohnNXT is complete.*



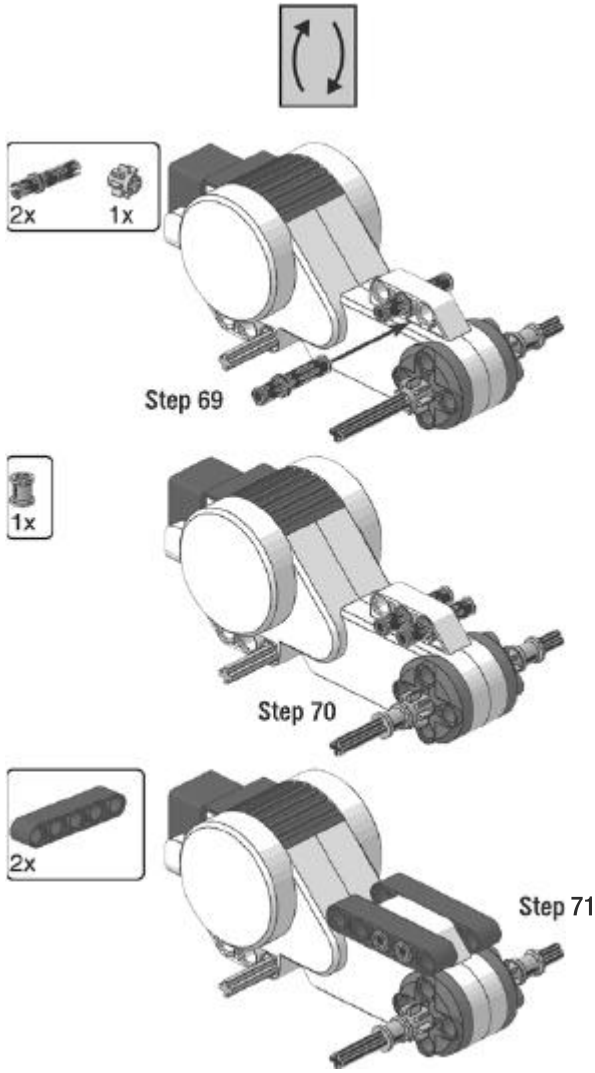

*Start building the right part of the lower body. Attach the 15-long beam to the 13-long beam as shown.*



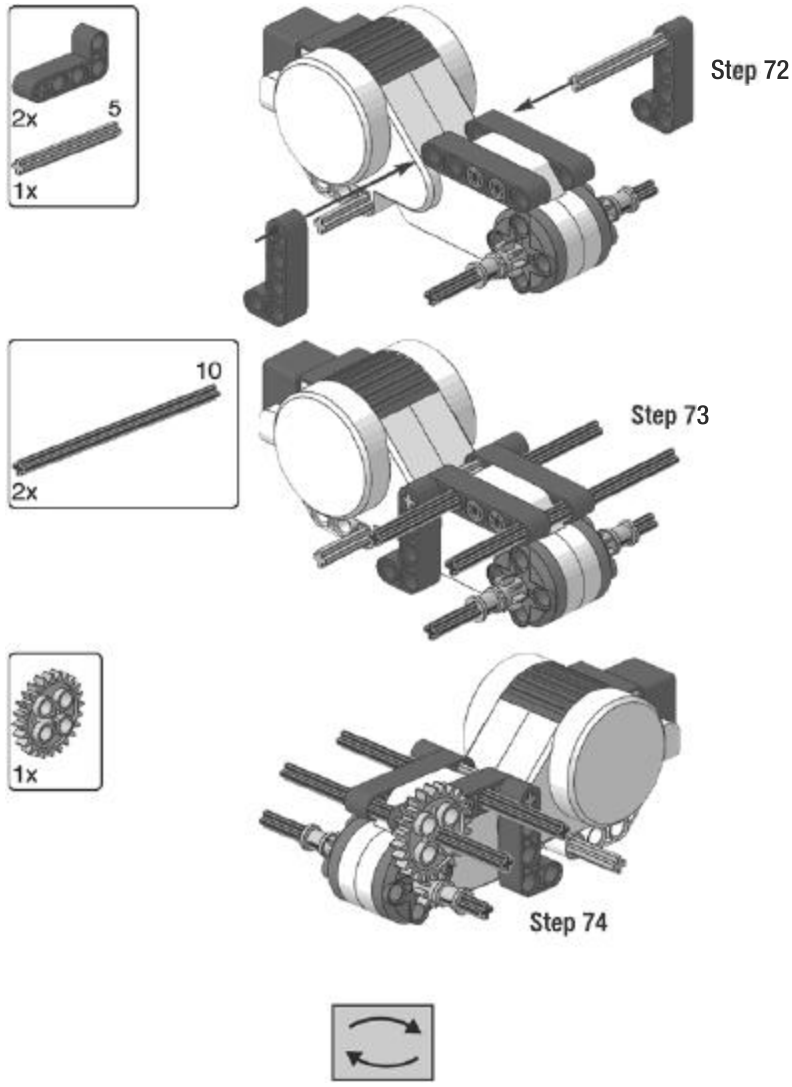

*Complete the right part of the lower body.*



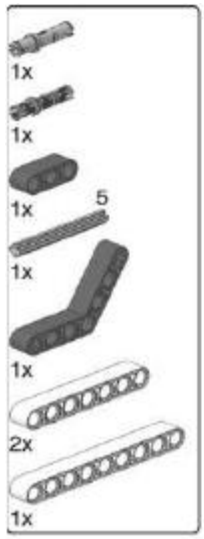

*Start building the torso's lifter mechanism.*

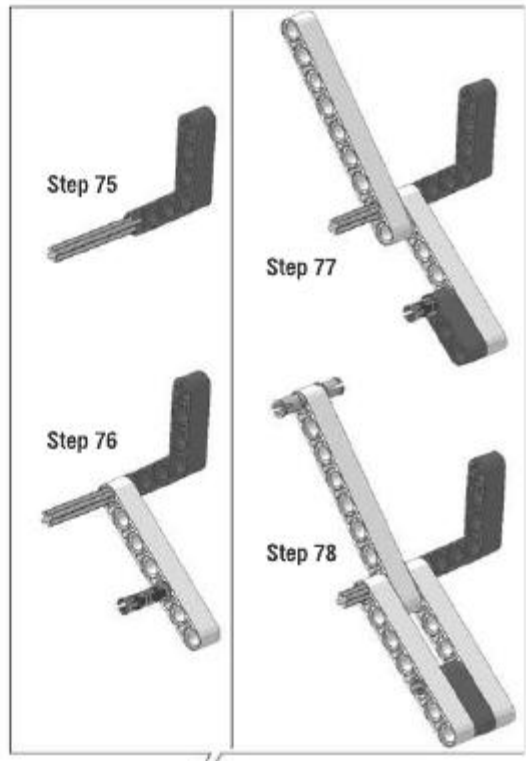


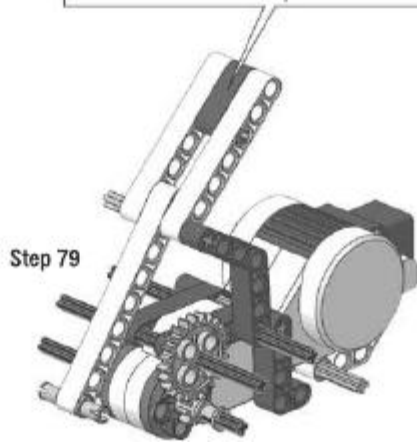

*Turn the motor as shown.*



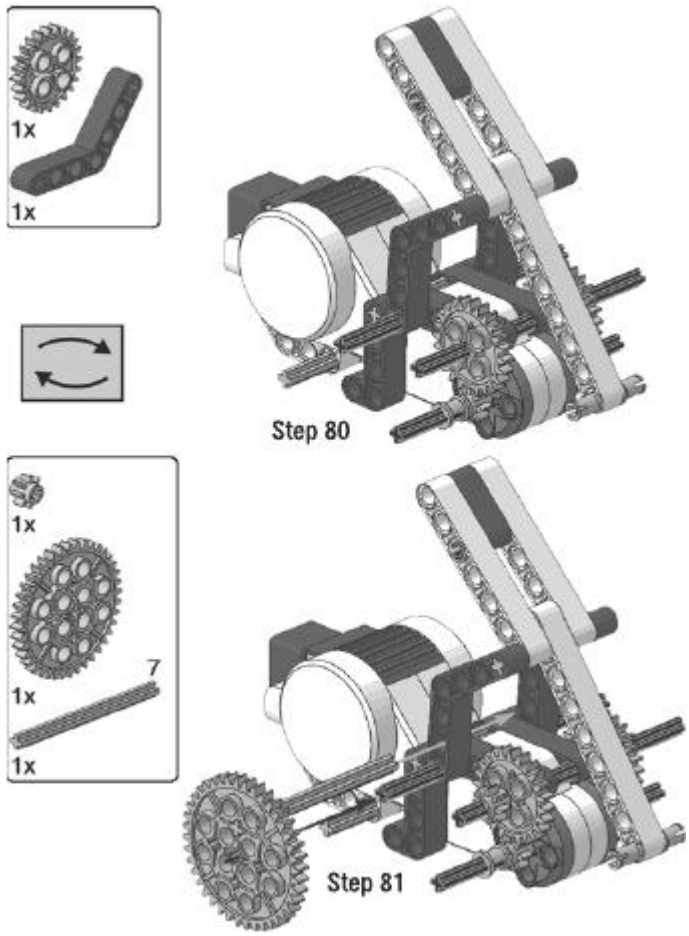

*Continue building the torso's lifter mechanism.*



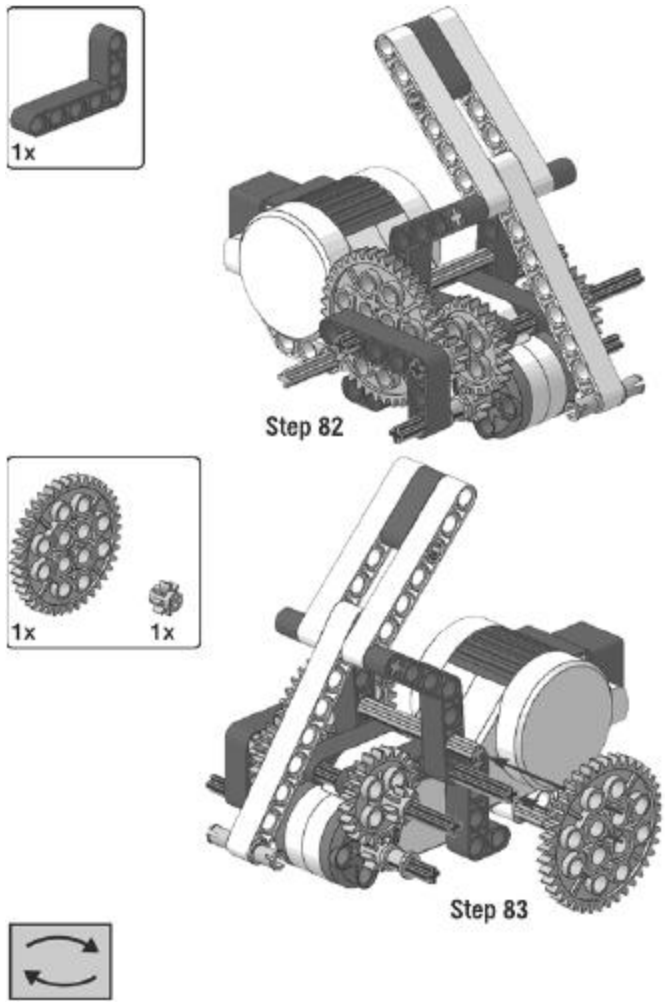



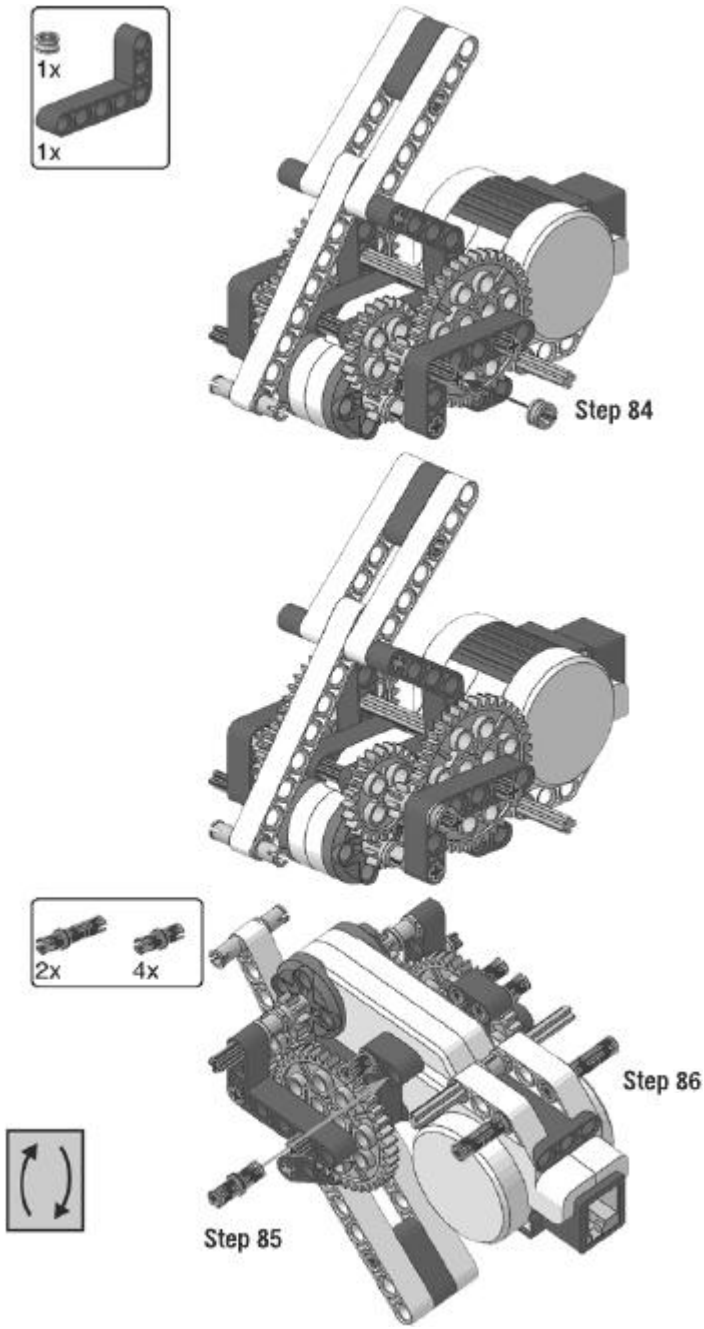

*Build the levers' structure that will lift the whole JohnNXT body. Use two 7-long beams and a 9-long beam.*



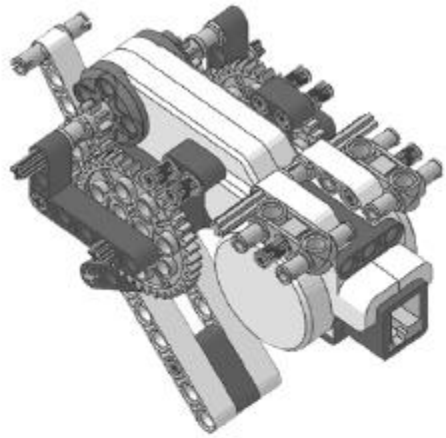

*When adding the 40-tooth gear, make sure that the 7-long axle is correctly inserted in one of the gear's axleholes.*



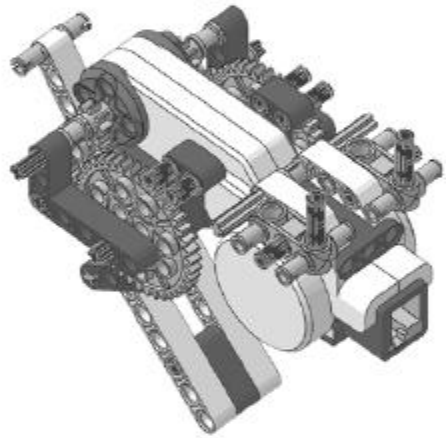

*Add the bent beam that cross-braces the geartrain axles, to avoid gear scratching. Add the symmetric 40-tooth gear and 8-tooth gear to complete the geartrain.*



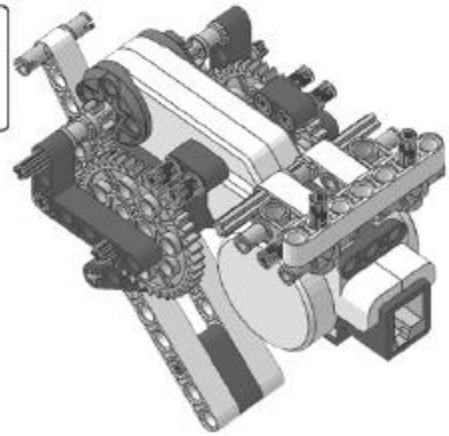

*Add the other cross-bracing beam. Then turn the model and add the pins where shown.*

Step 87

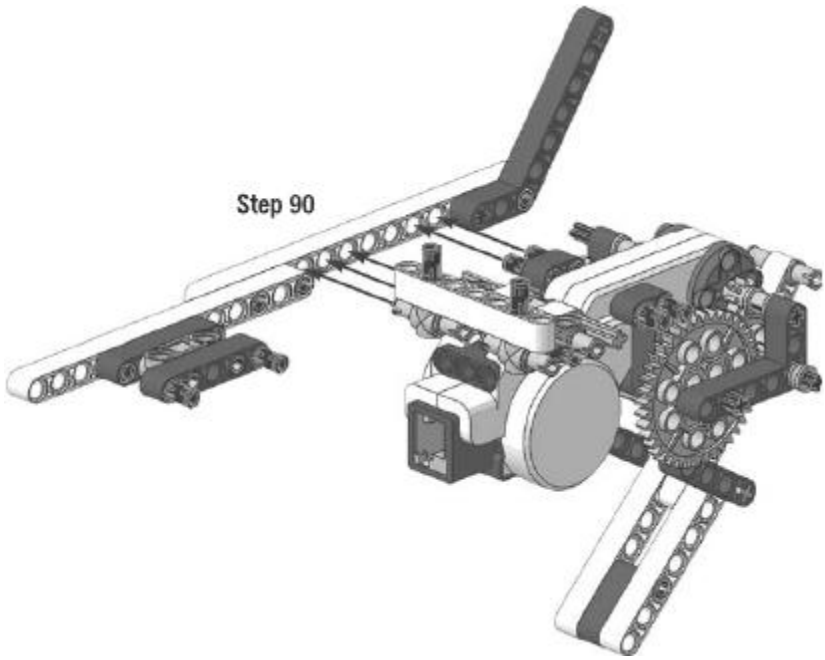



Step 88

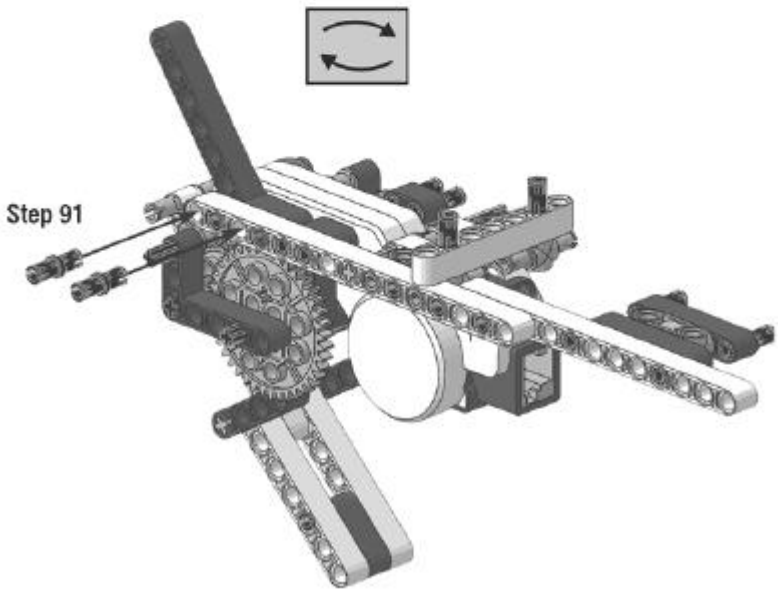



Step 89

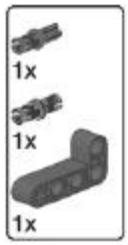
*Complete the torso's lifter mechanism.*

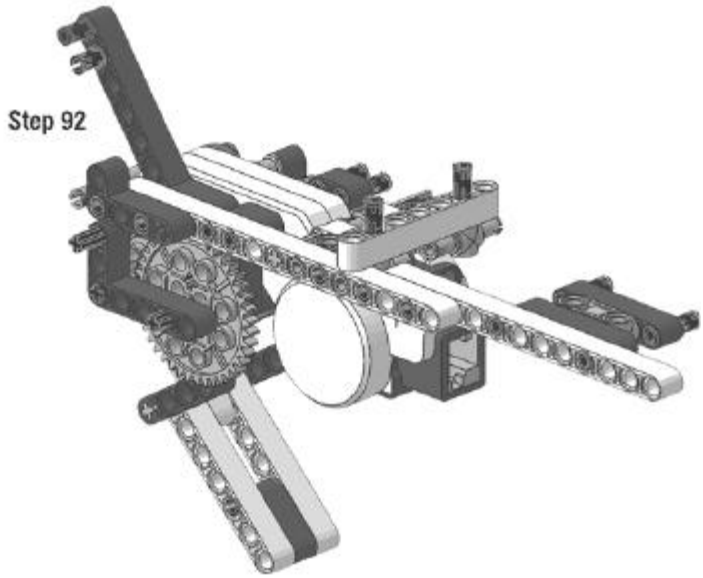



*Attach the subassembly you just finished to the right part of the lower body.*

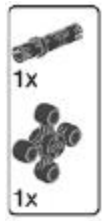



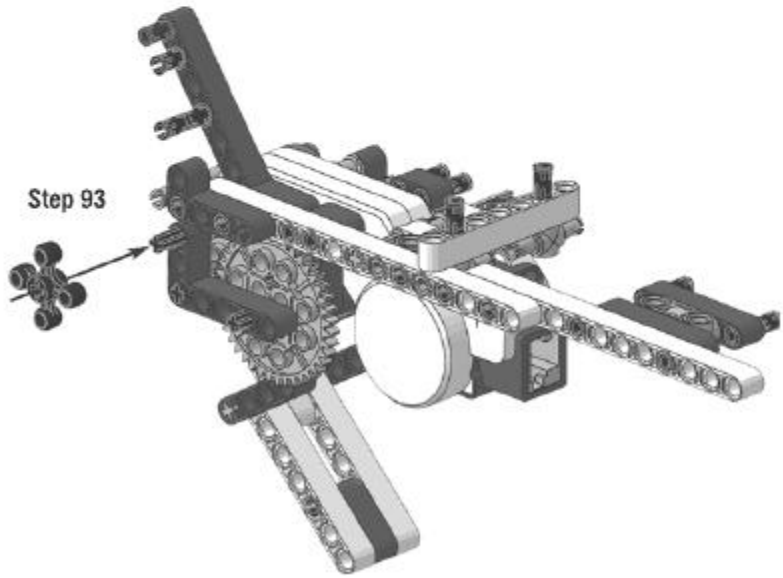

*Turn the model and add two black pins.*



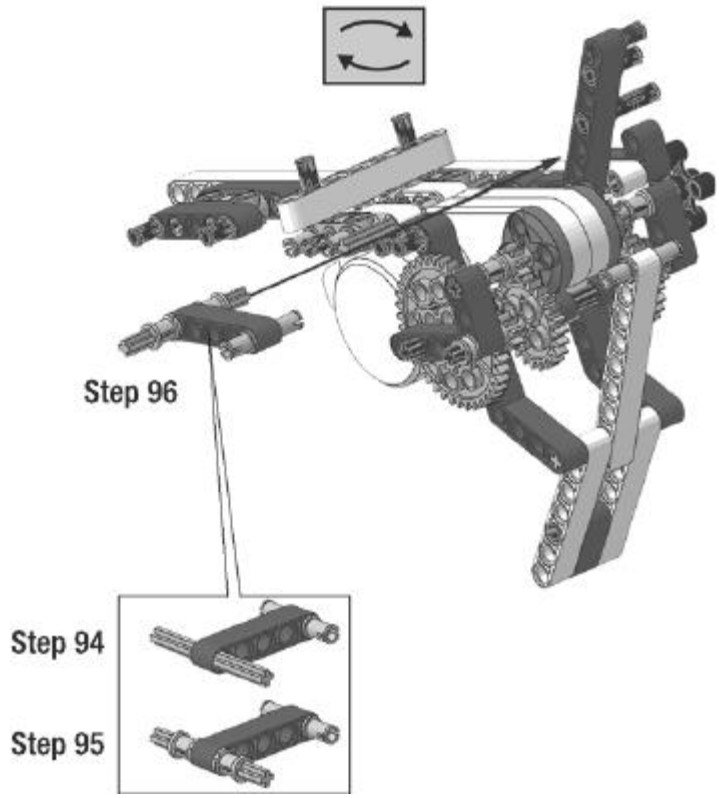




*Add a blue axle pin in the top hole of the bent beam, just over the black pin. Add the liftarm that will be used to hold the cable of the arms' motor.*

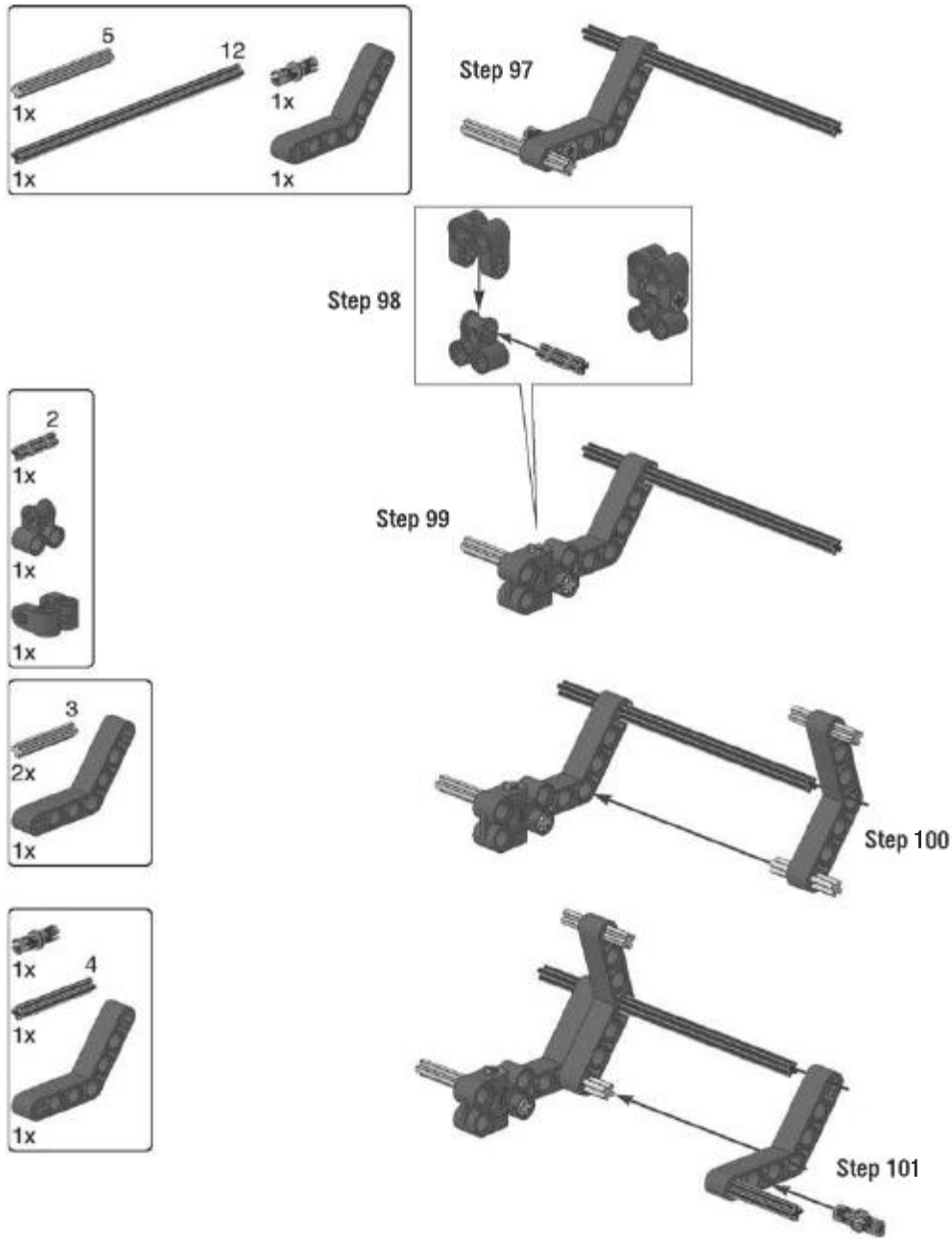


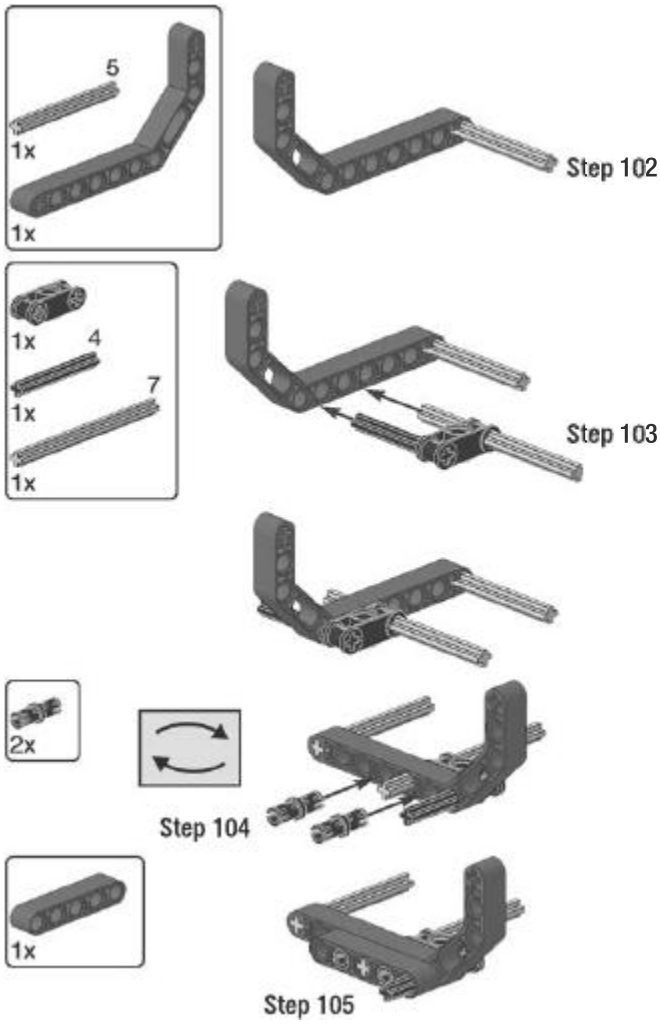

*The knob wheel is used to move the torso manually.*

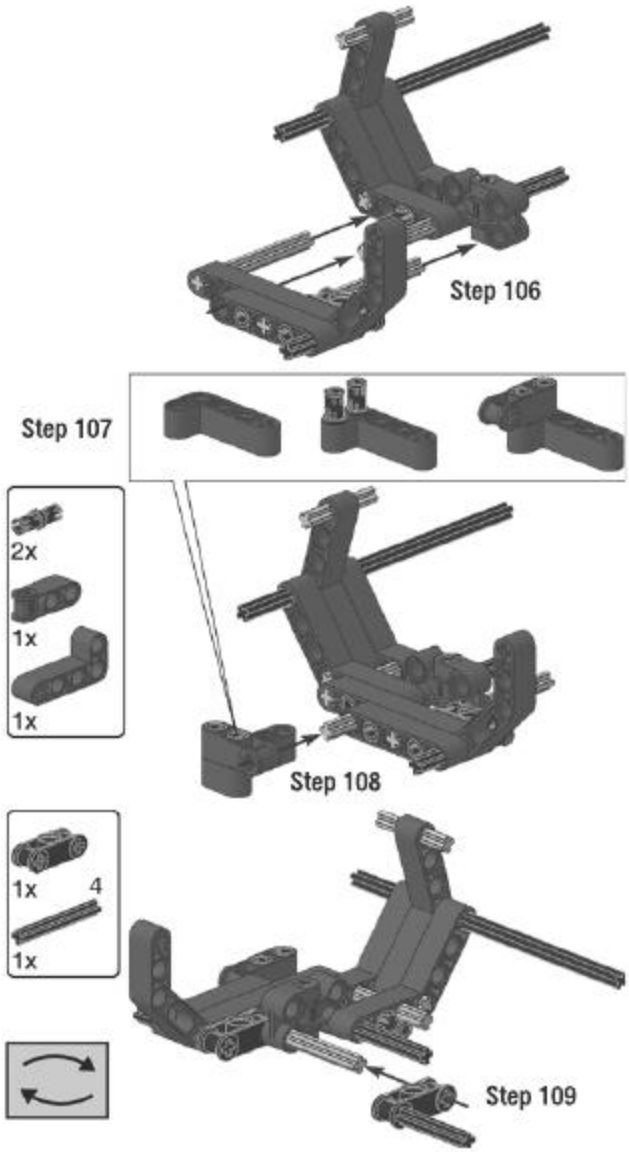

*Turn the model and add the auxiliary lever. Use a long gray pin.*



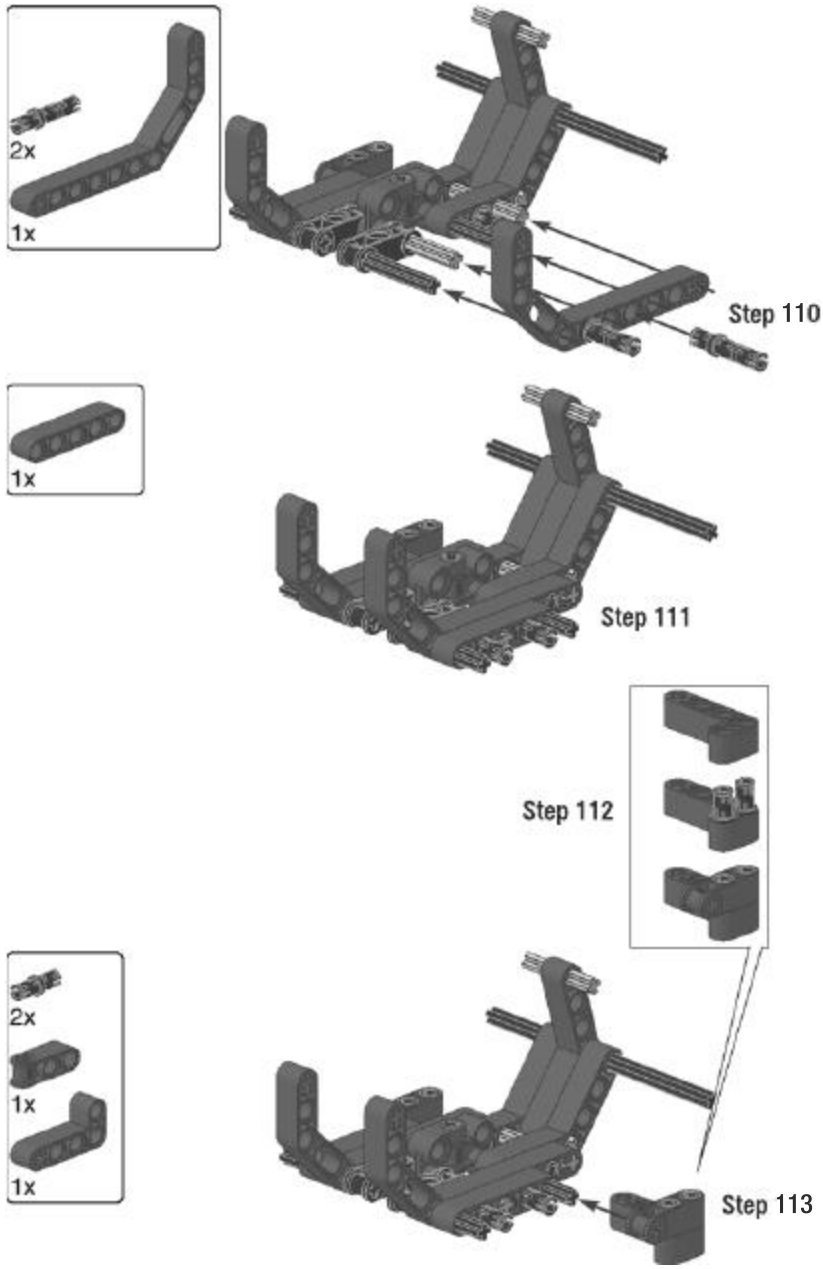

*Start building the upper body.*



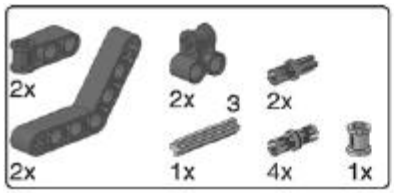

*Build the right part of the upper body.*

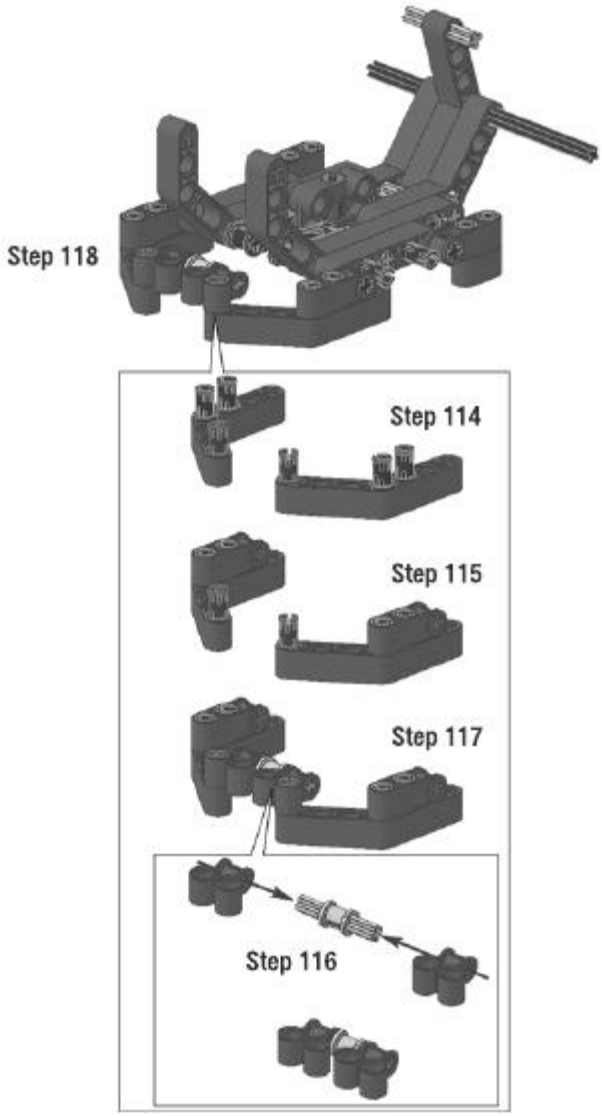



*Attach the two submodels together.*

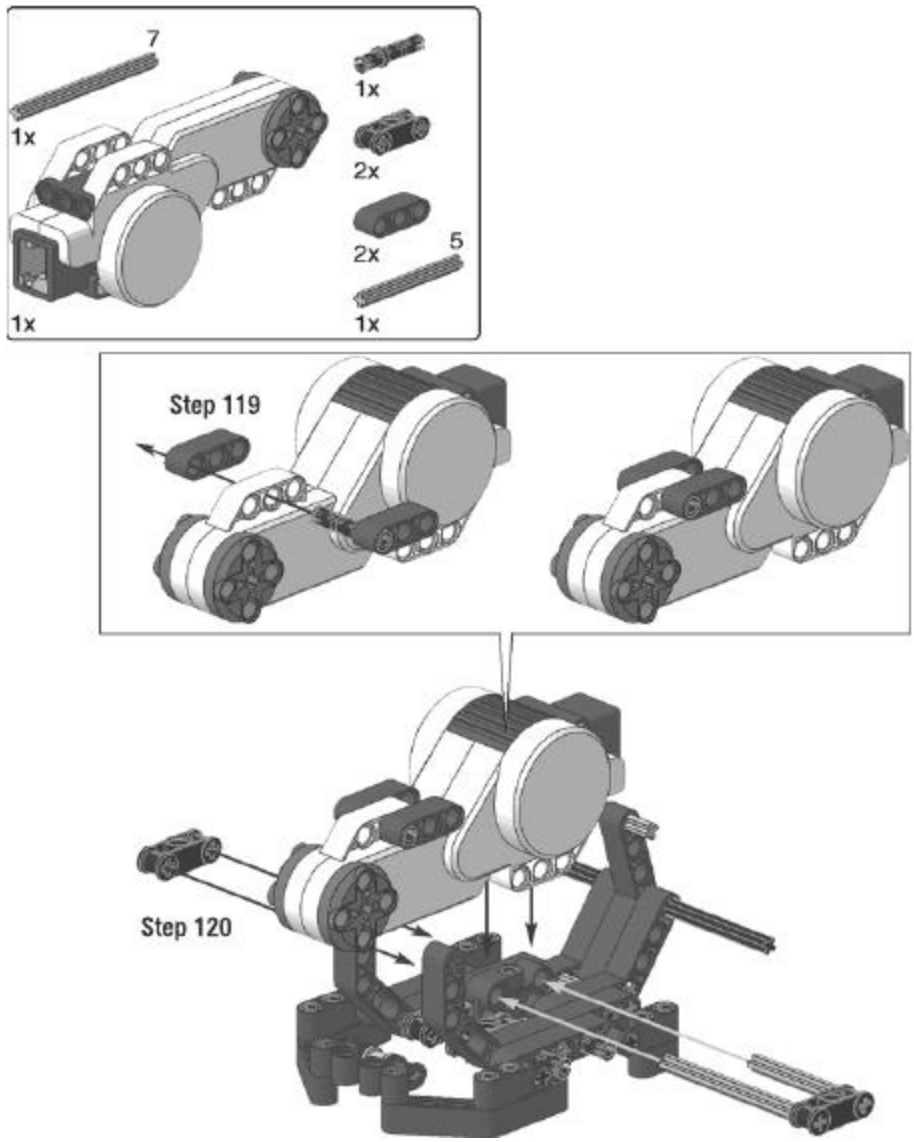



*Build the left part of the upper body.*

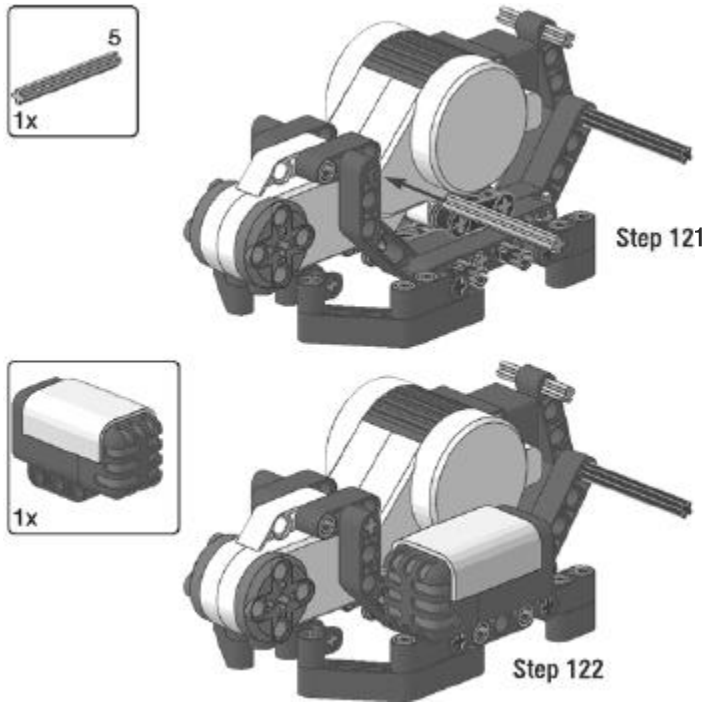


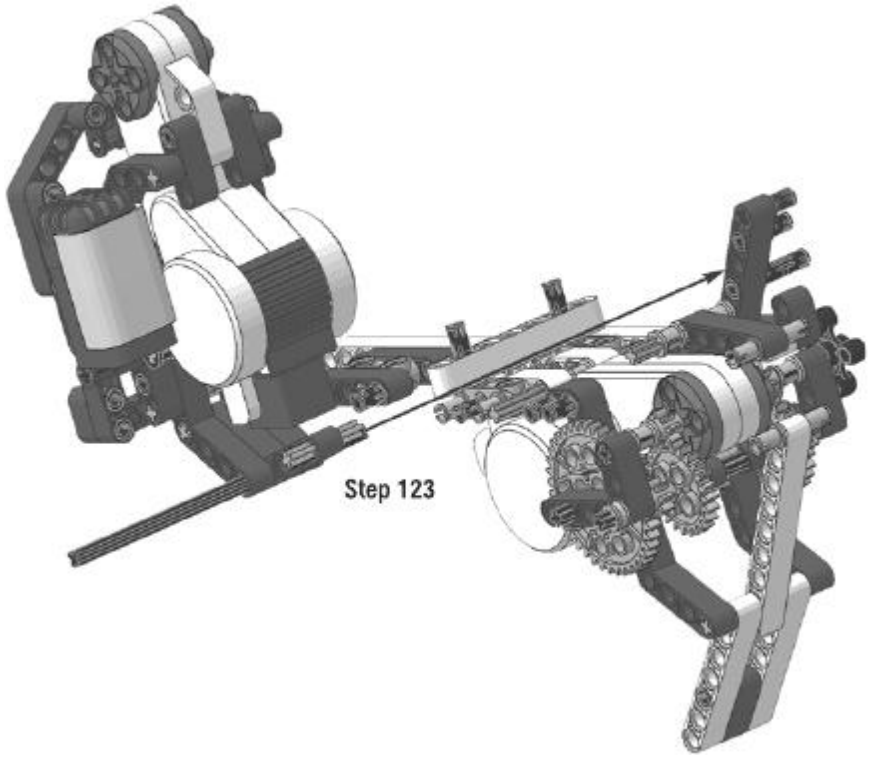

*Complete the upper body's lid shape.*



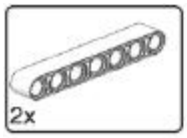

*Add the arms' motor.*

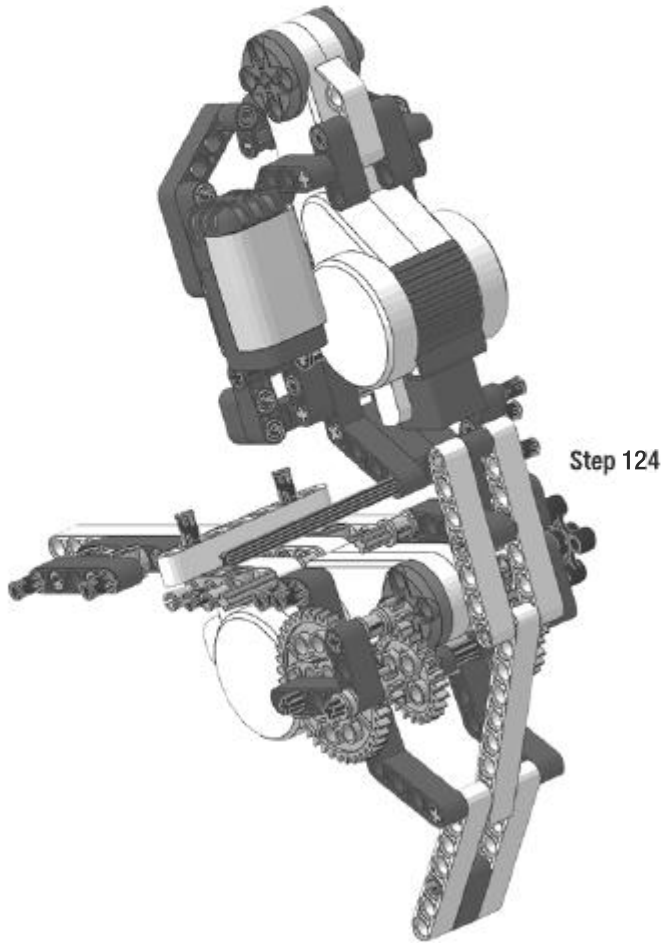



*Hold the motor in place with the 5-long axle and add the Sound Sensor.*

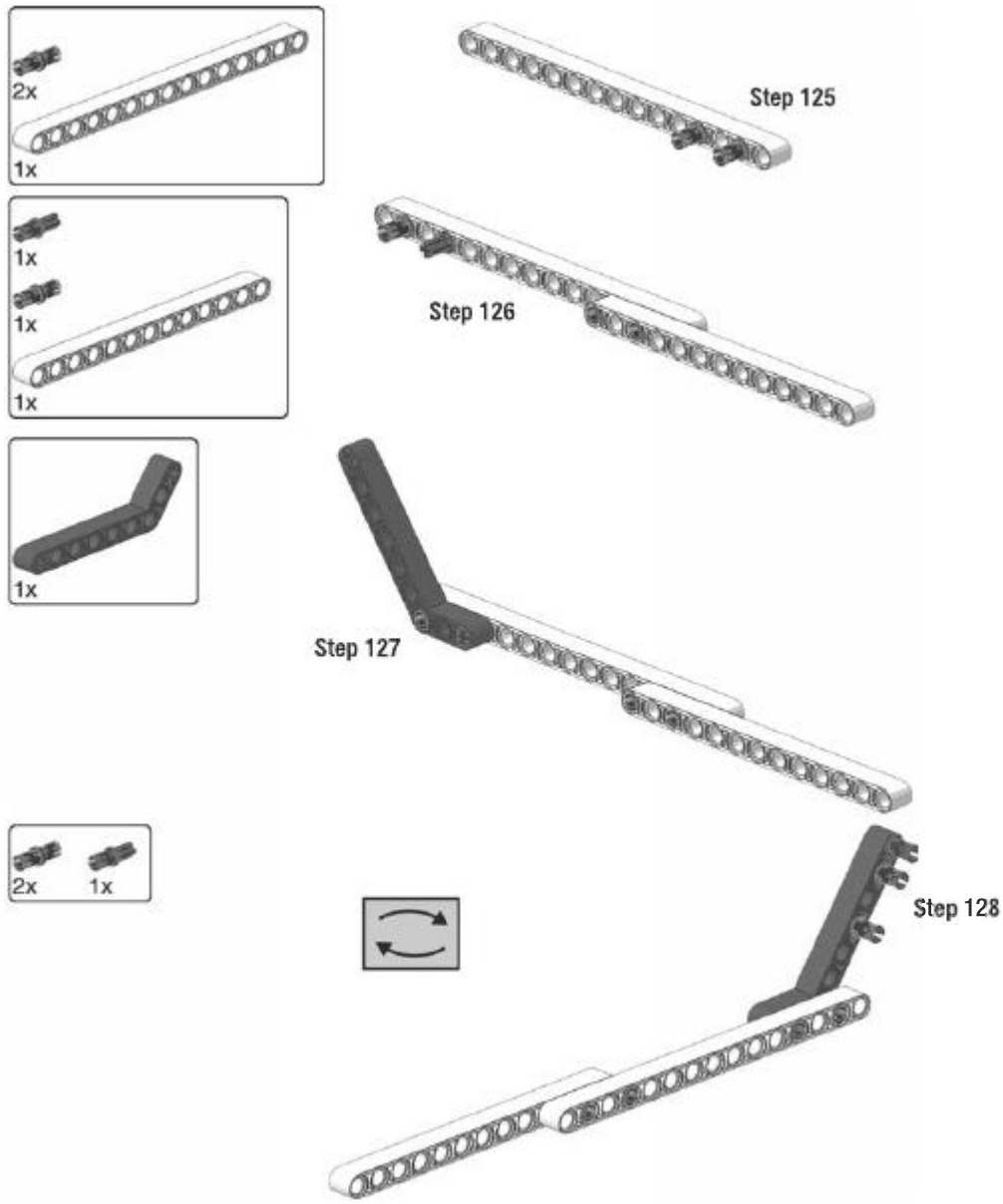



*Rotate the body so you can see it from behind. Insert the upper body's assembly onto the rest of the body.*

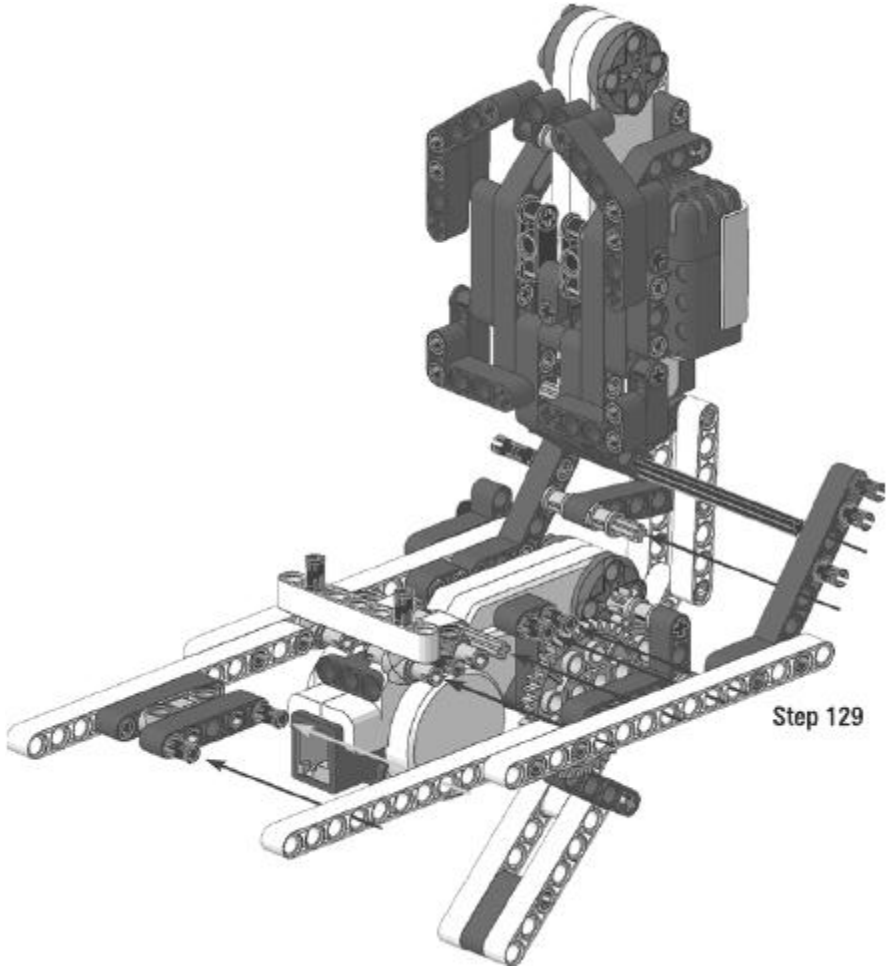


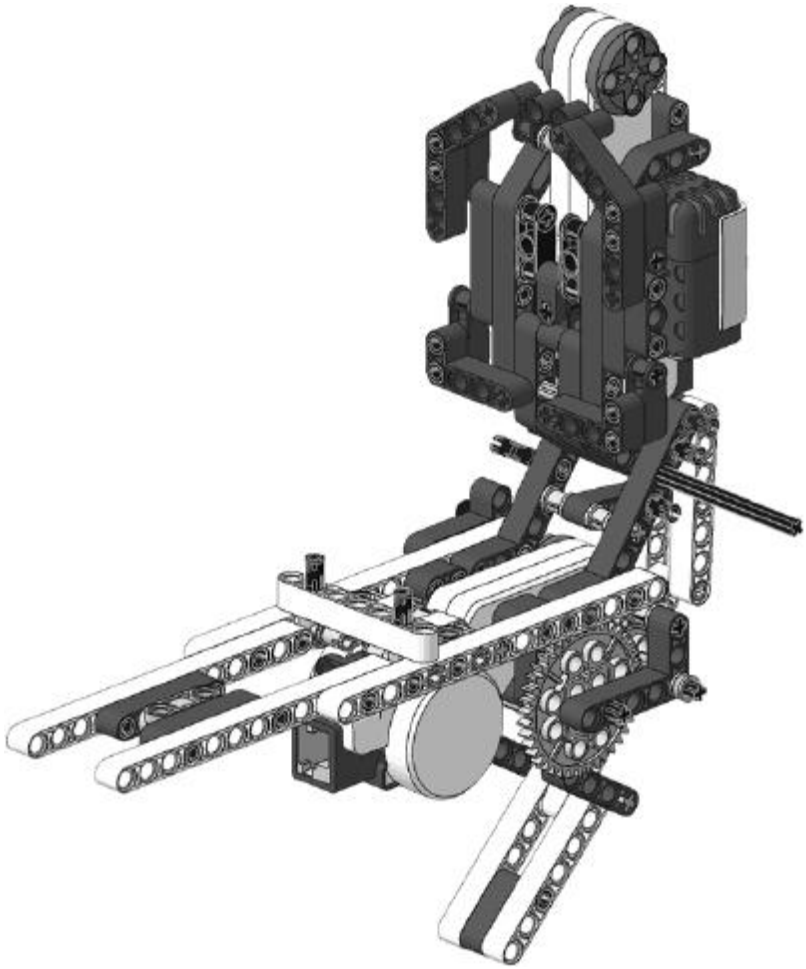

*Add two 7-long beams.*



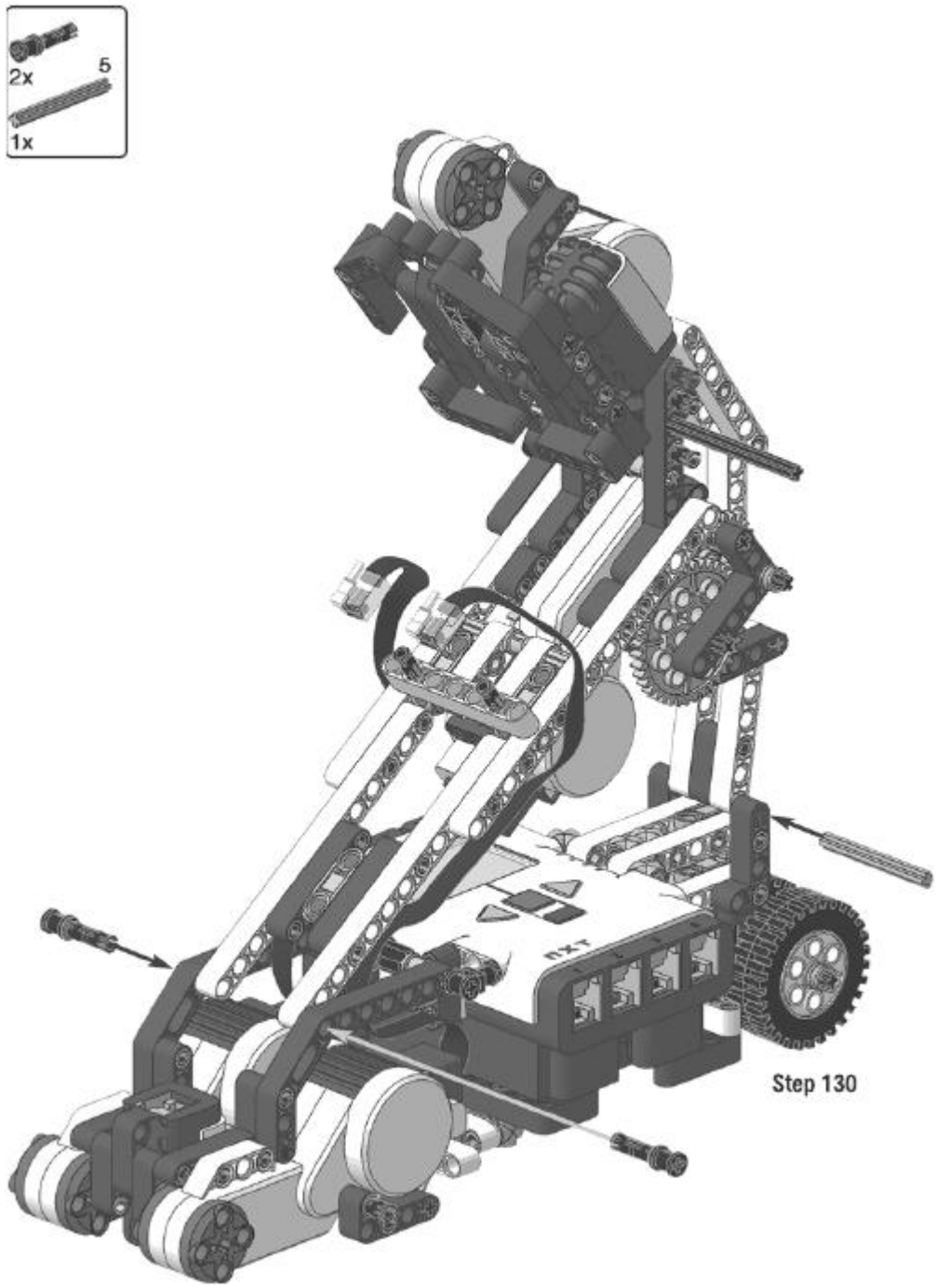

*Build the left part of the lower body. In Step 128, add the blue axle pin in the top axlehole of the bent beam.*




*Add the left side of the body to the rest of the body.*

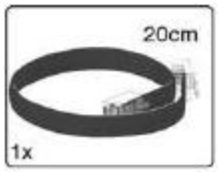


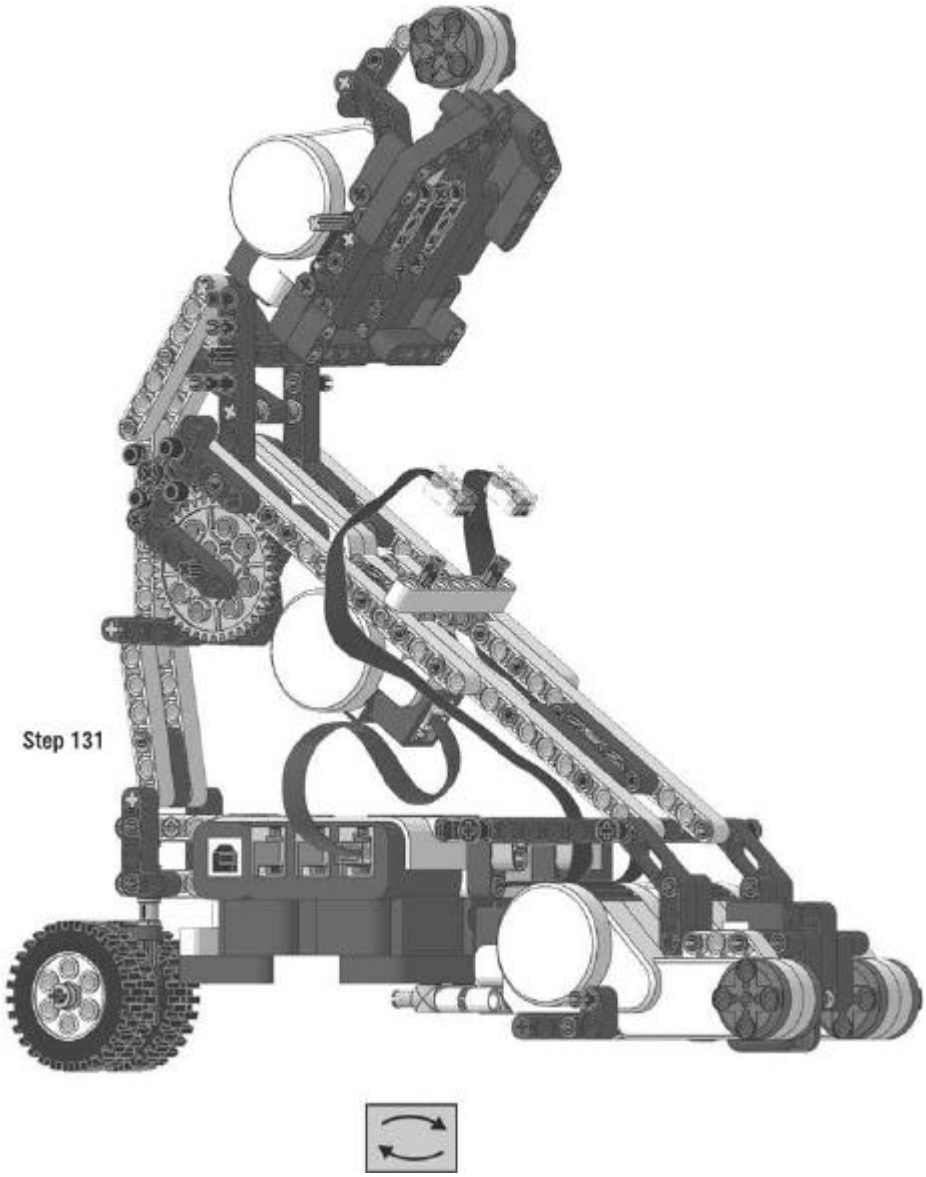
*The body is complete.*



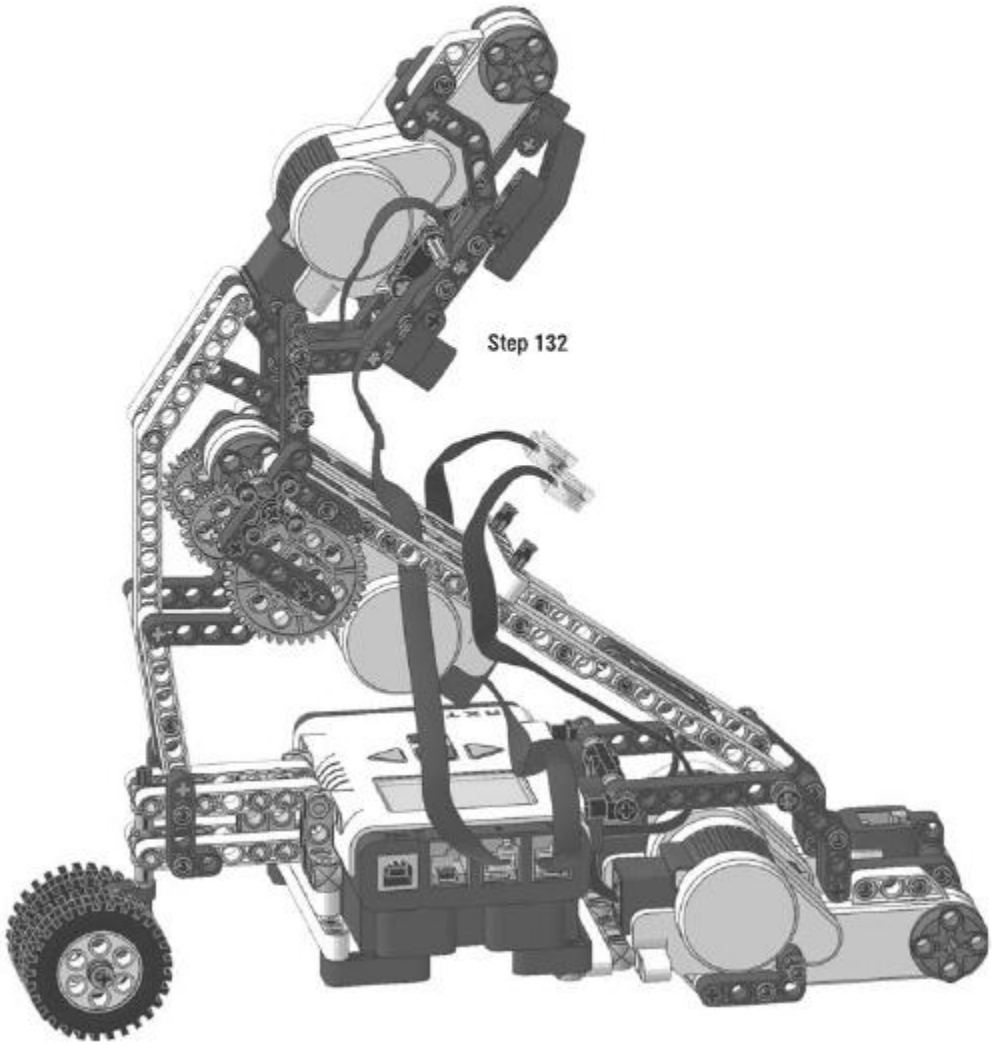

*Attach the body to the base and block it with two long pins with a stop bush and a 5-long axle in the back.*


*Try turning the knob wheel on the lifter's motor shaft to test whether the body's lifter mechanism is working.*

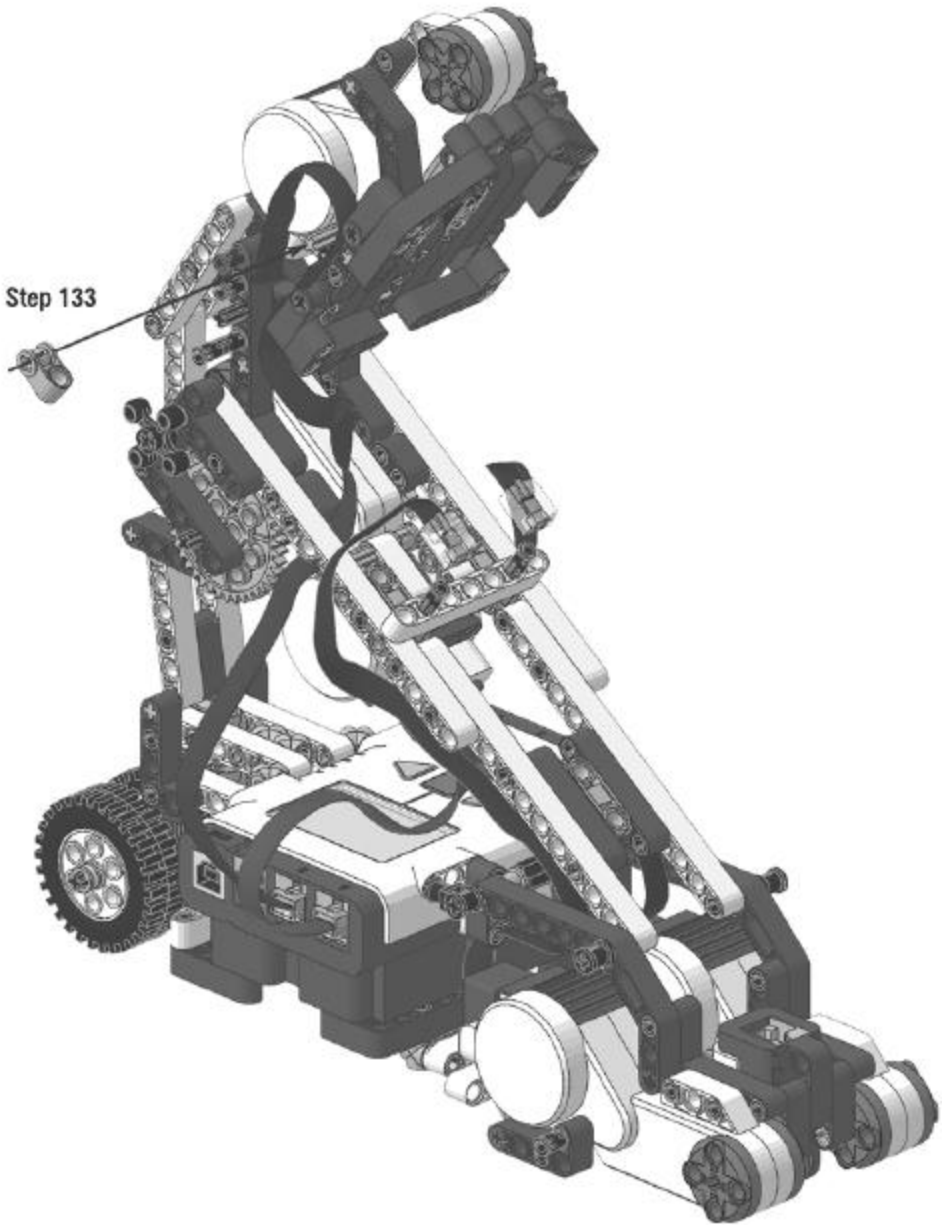


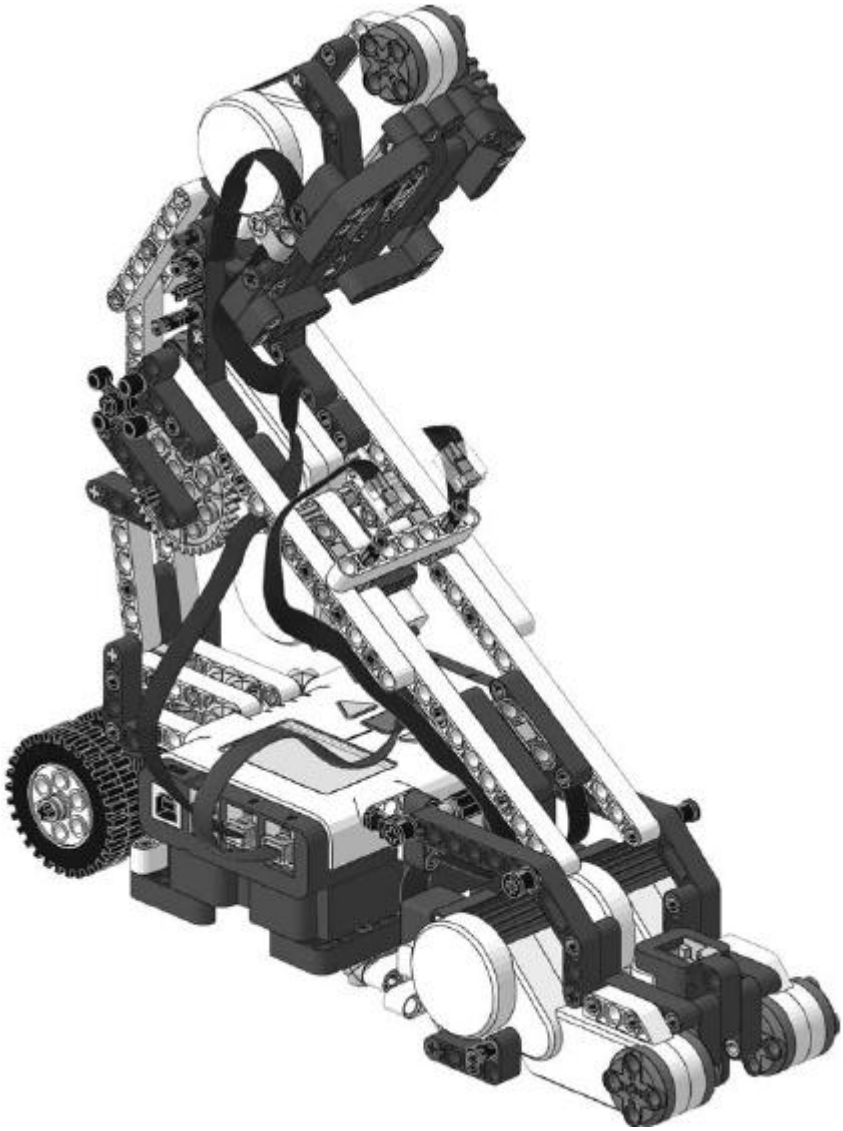

*Turn the model and attach the torso's lifter motor to slave NXT port A using a 20cm (8 inch) cable.*

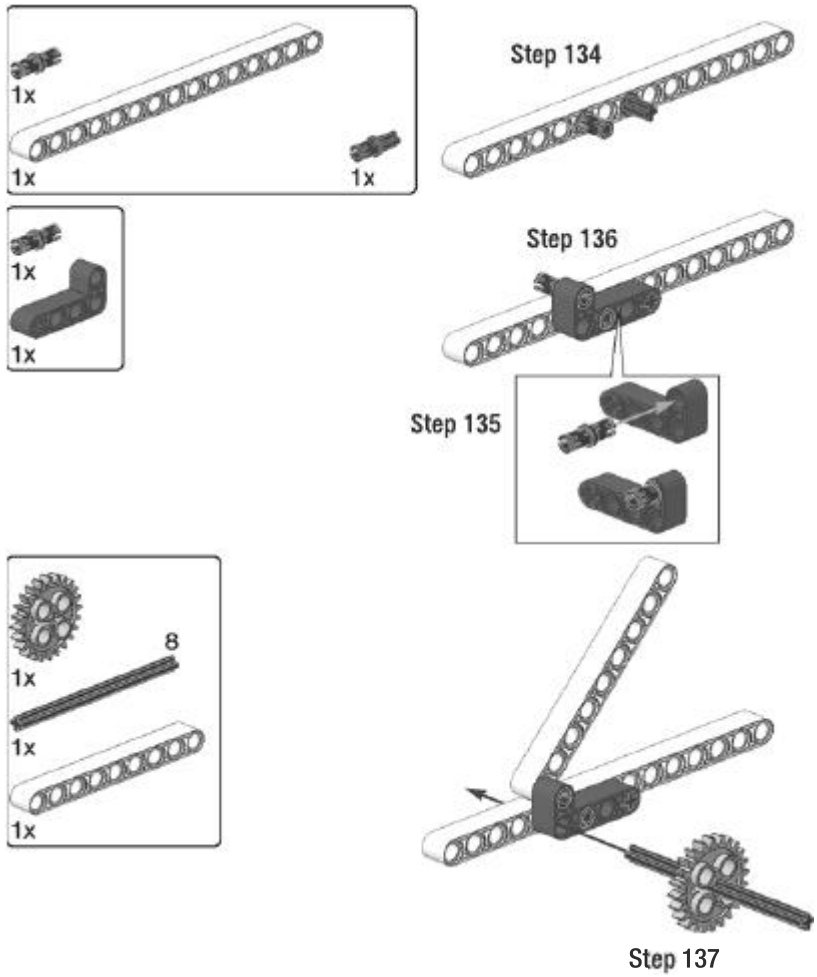

*Attach the arms' motor to slave NXT port B using a 35cm (14 inch) cable. Make sure to pass the cable where shown.*

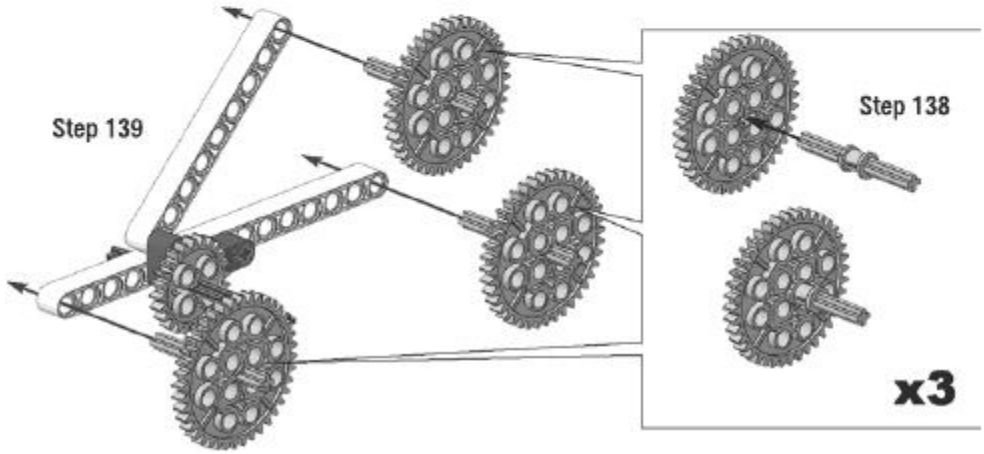

*Add the axle joiner to hold the cable in place.*



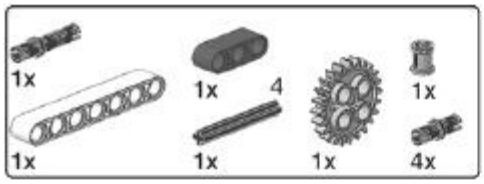
*Check if all the cables are placed correctly.*

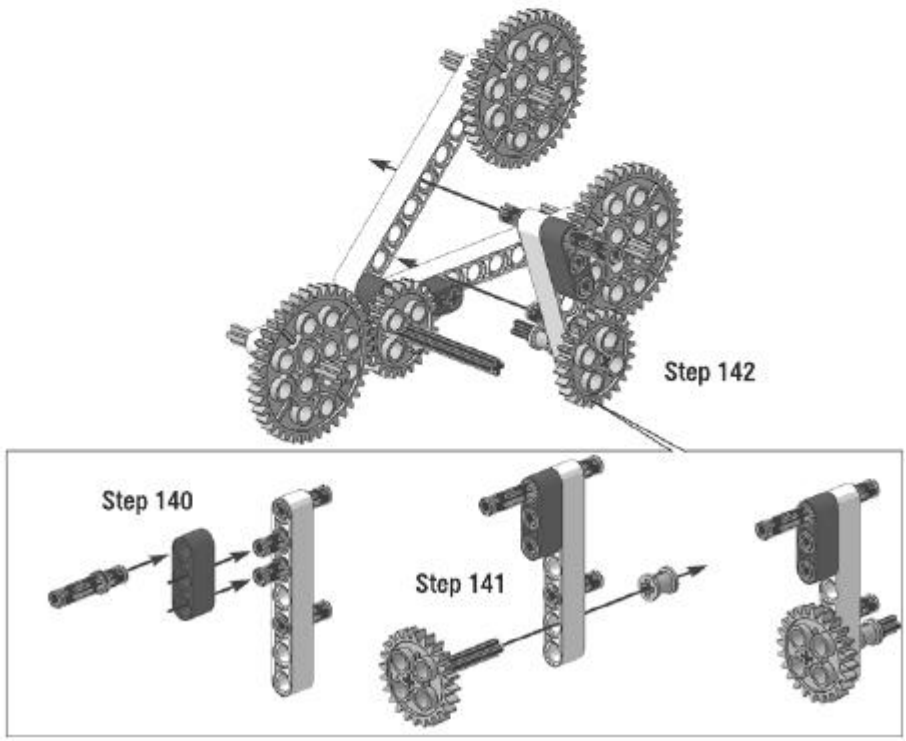



*Start building the right tread's assembly. Use a 15-long beam and a 9-long beam.*

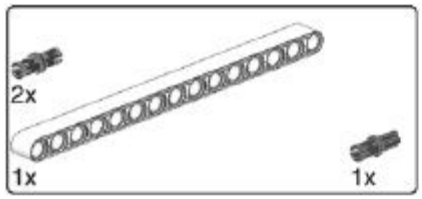



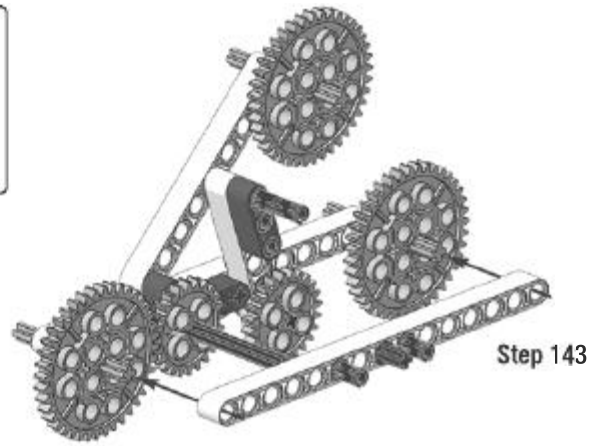

*Insert the three 40-tooth gears, used as hubs.*

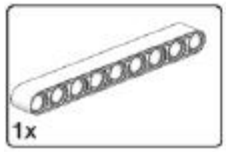


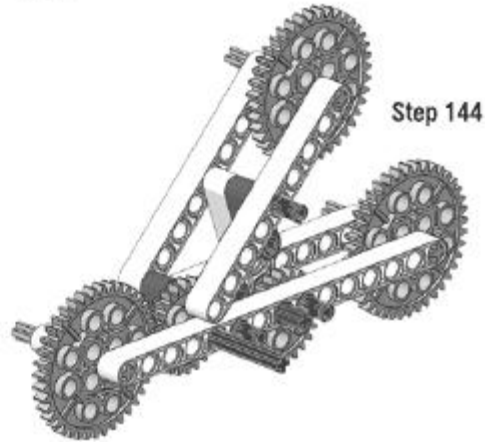




*Build the assembly that helps to form the strong triangular shape of the tread.*

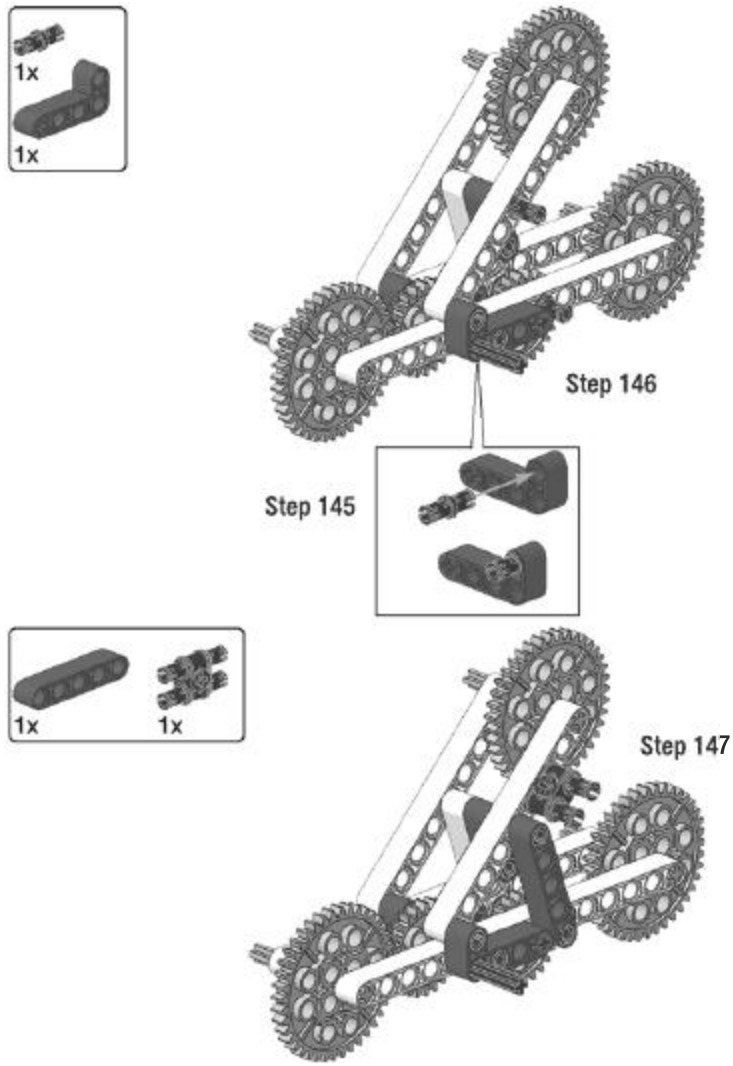


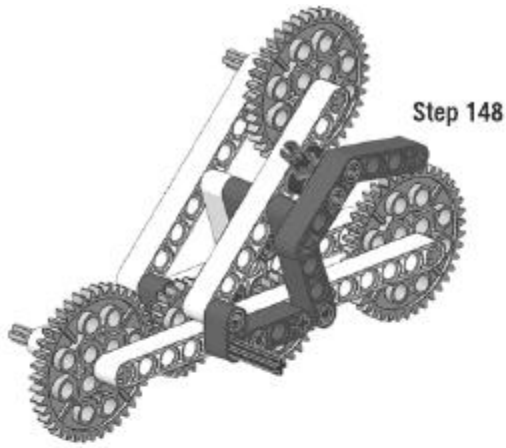



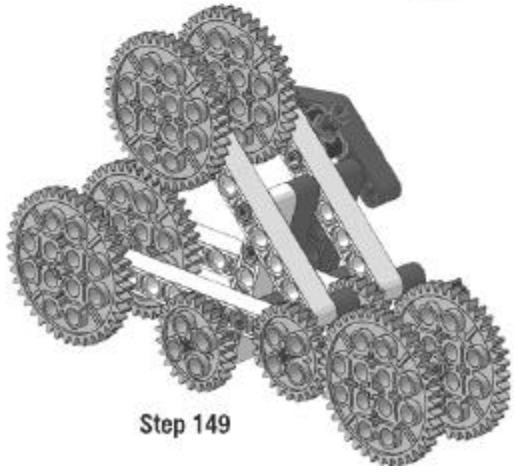

*Add a 15-long beam and a 9-long beam.*



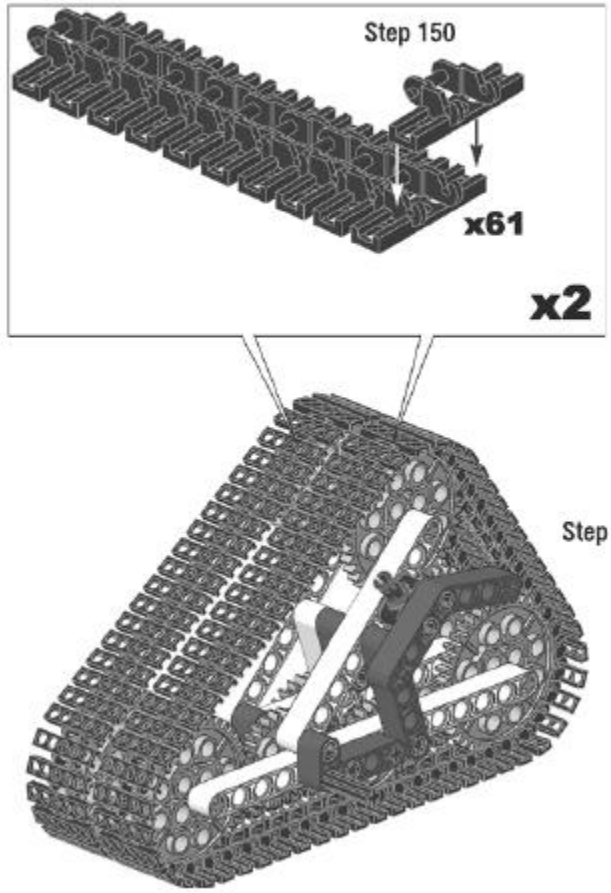

*Complete the triangular frame.*

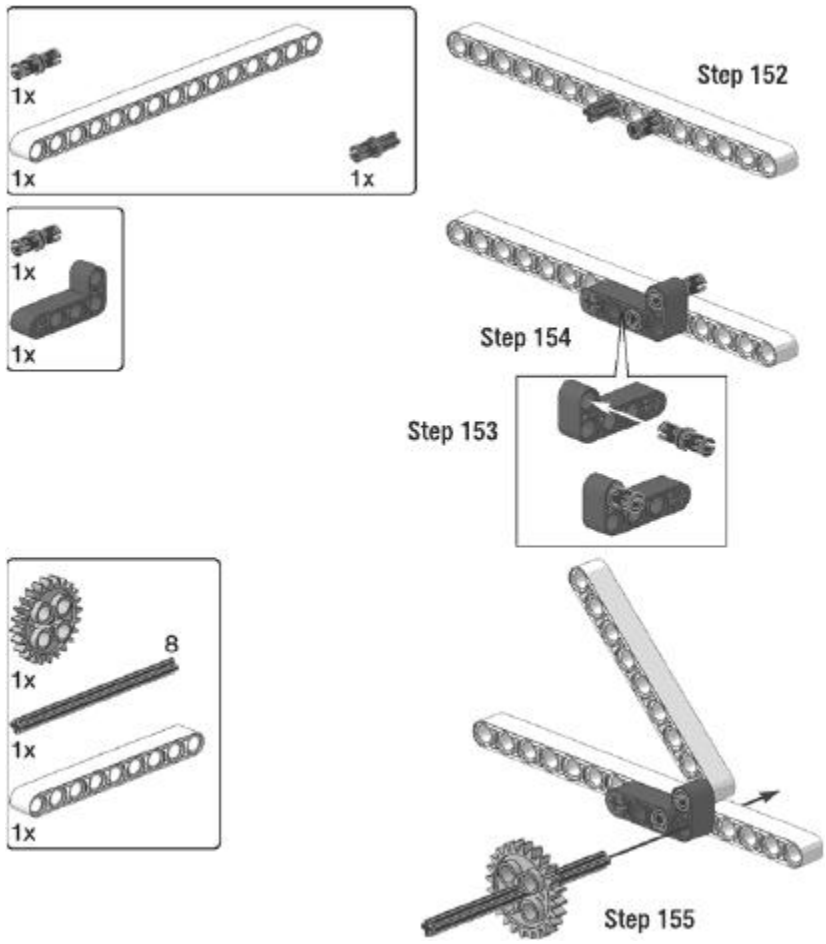




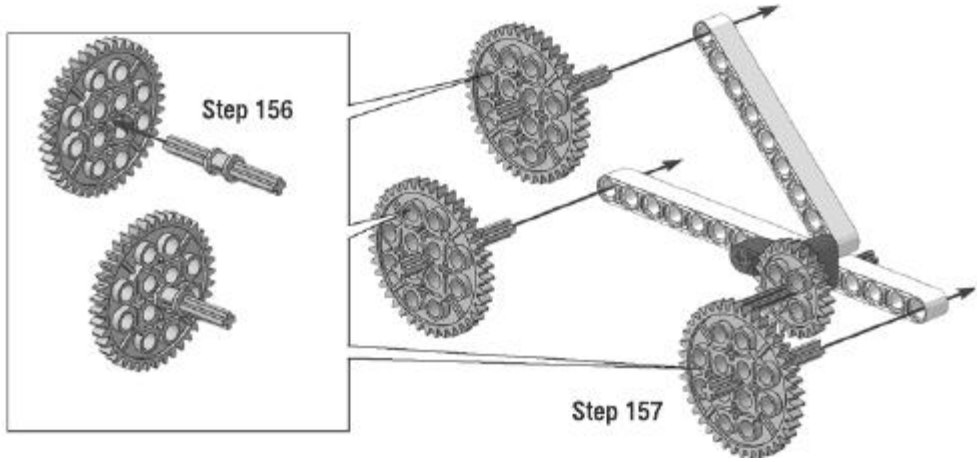

*Add a blue axle pin in the black joiner. Turn the model and add three 40-tooth gears and two 24-tooth gears.*

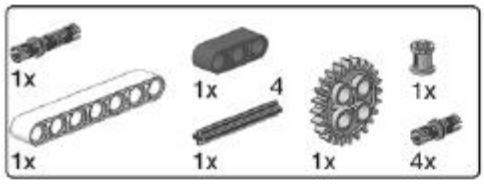

*Build 2 treads of 61 links each. Close them around the large gears. The right tread assembly is complete.*

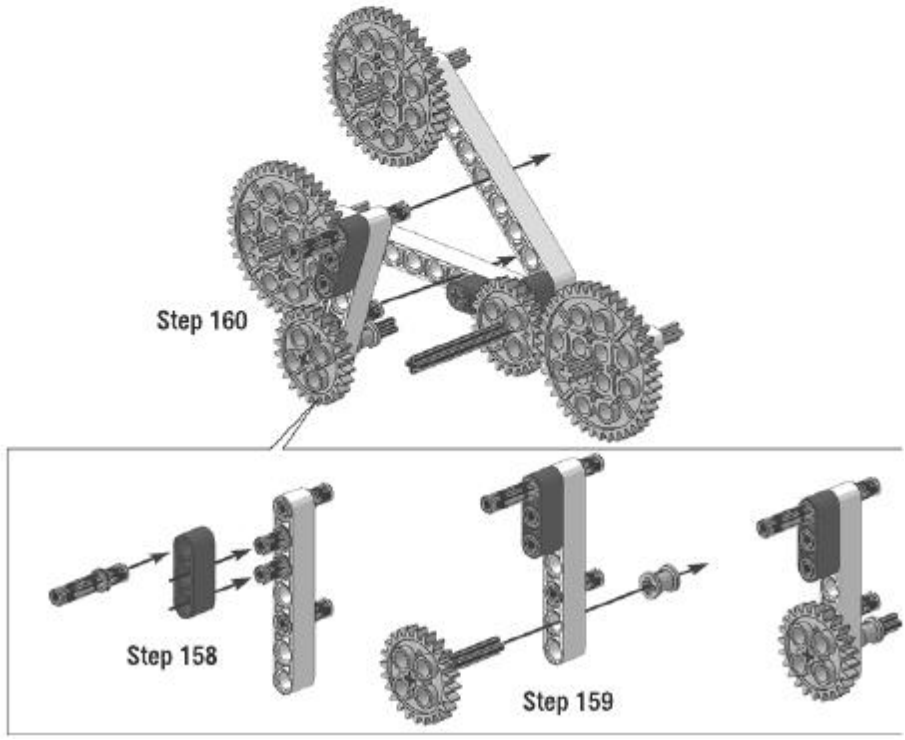



*Start building the left tread assembly. Use a 15-long beam and a 9-long beam.*

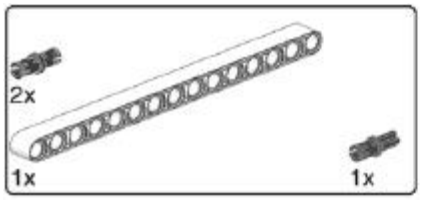



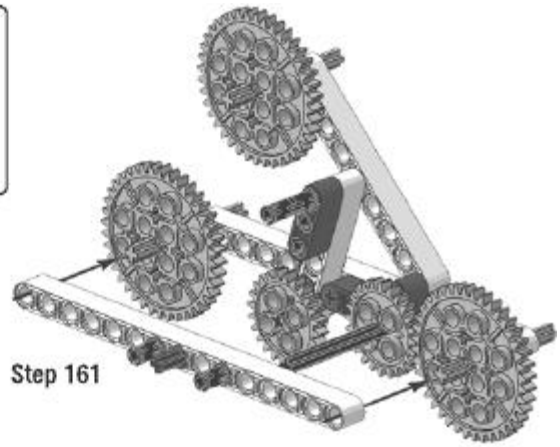

*Insert the three 40-tooth gears, used as hubs.*

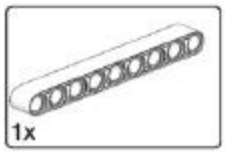


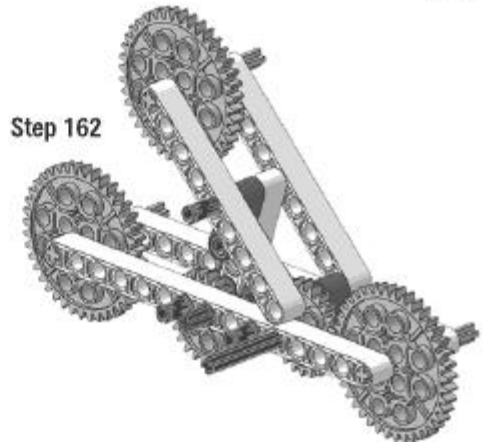




*Build the assembly that helps form the strong triangular shape of the tread.*

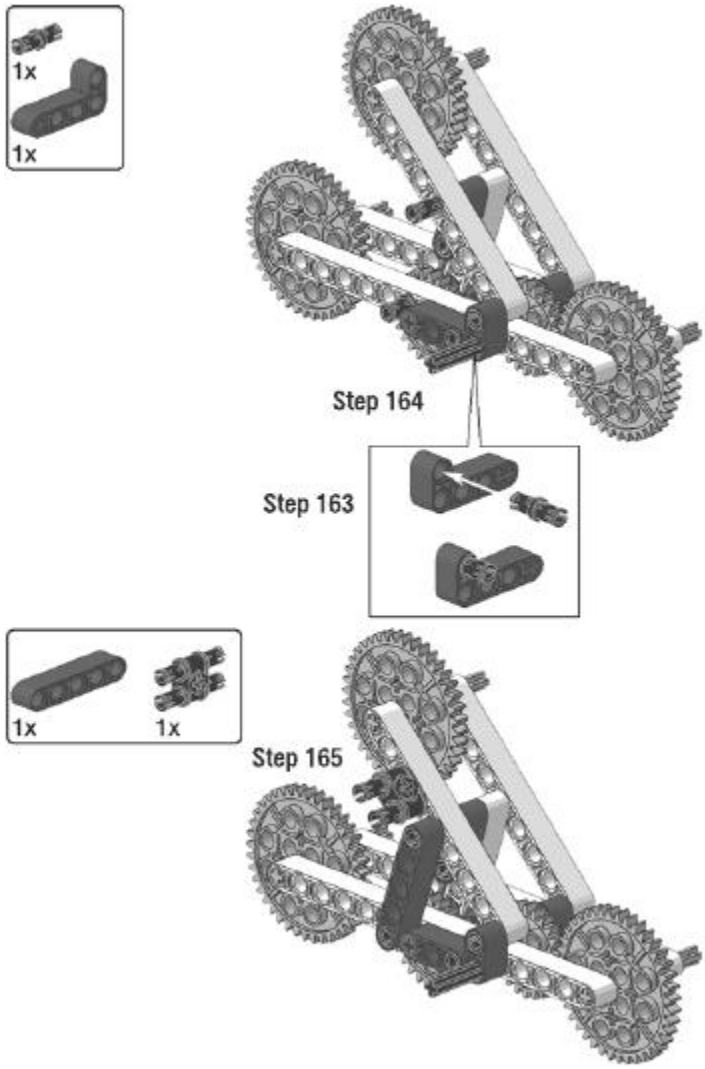


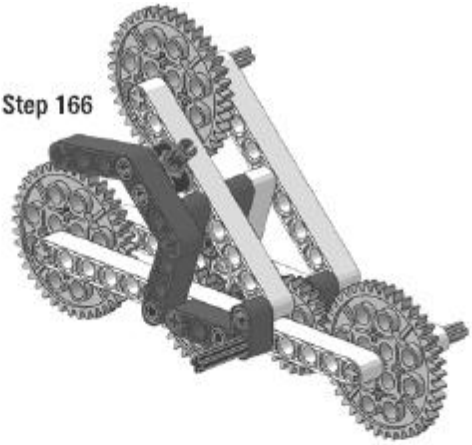




*Add a 15-long beam and a 9-long beam.*

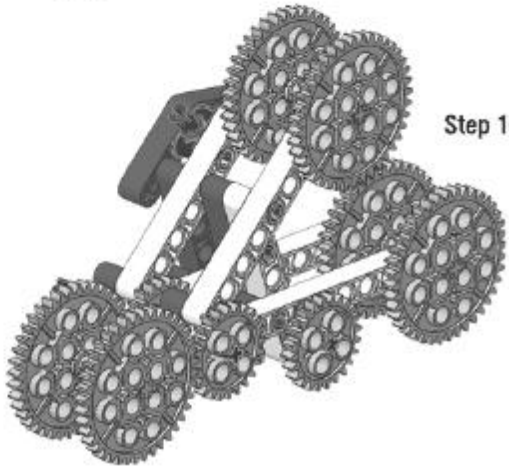



*Complete the triangular frame.*

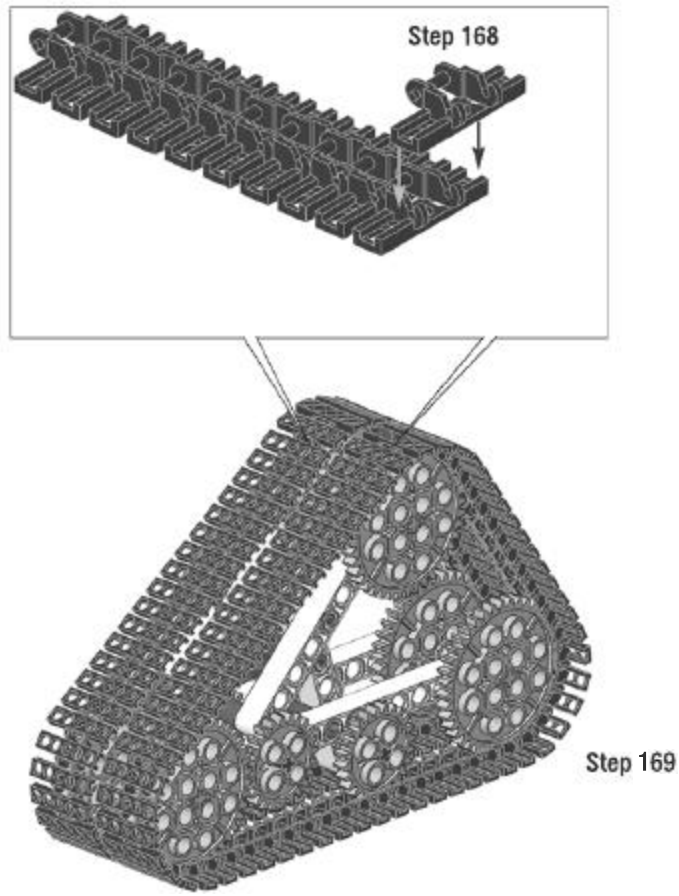

Step 166



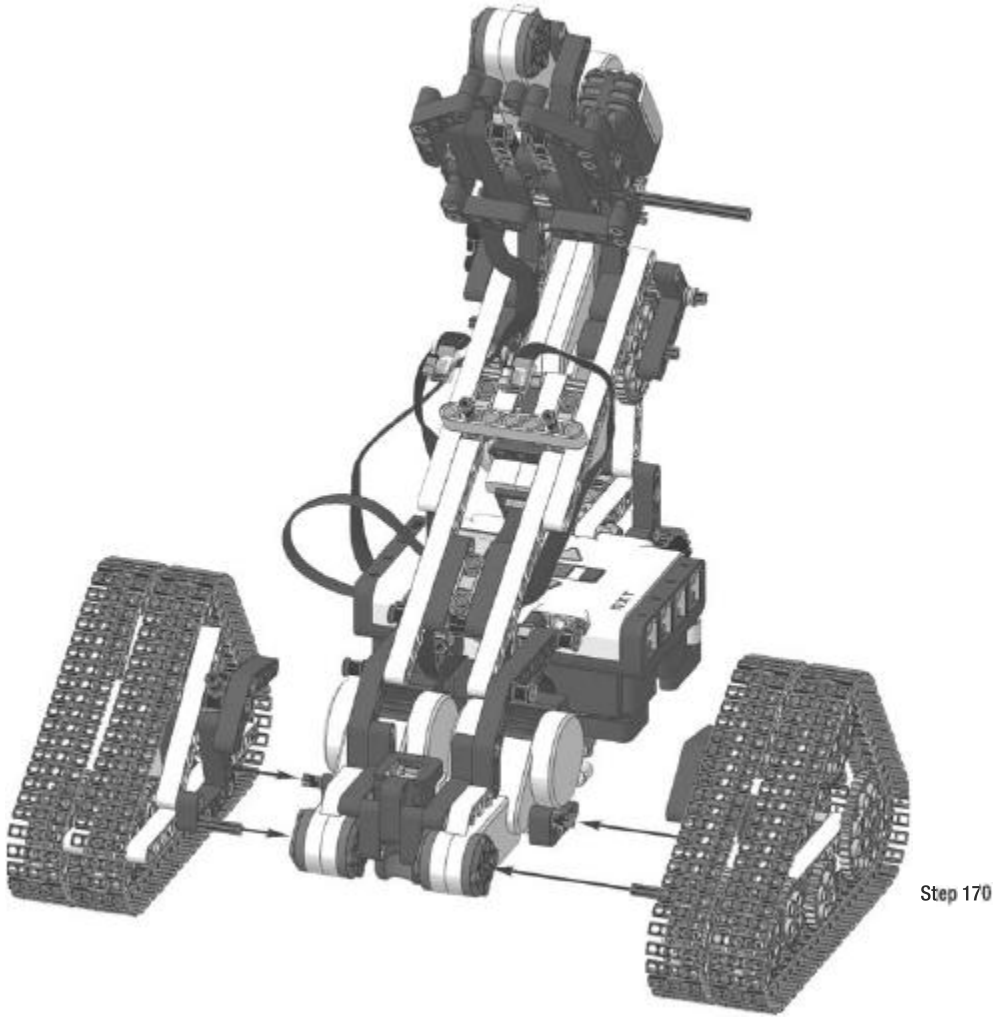

Step 167



*Add a blue axle pin in the black joiner. Turn the model and add three 40-tooth gears and two 24-tooth gears.*

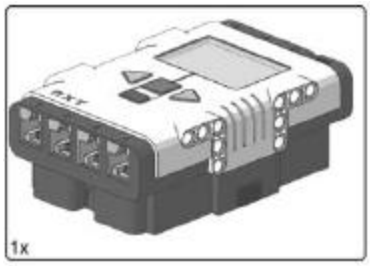

*Build 2 treads of 61 links each. Close them around the large gears. The left tread assembly is complete.*

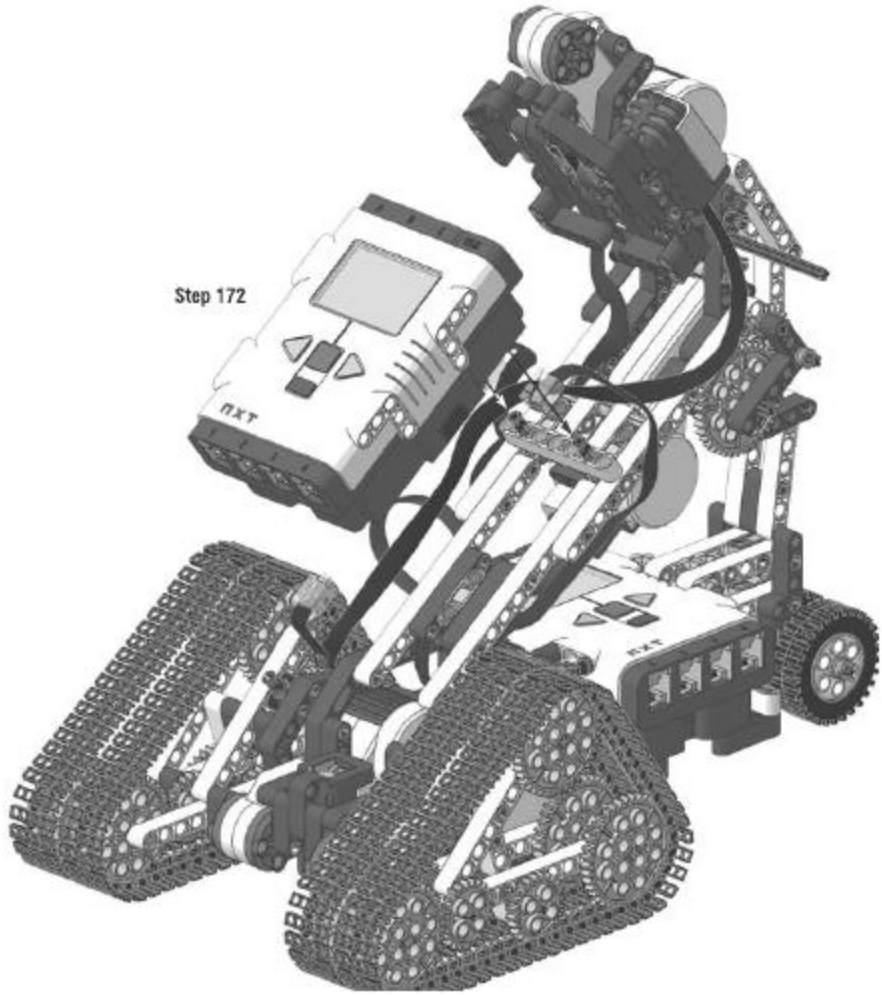



*Attach both treads in place. The driving axles go in the motor shafts, while the black pins go in the corresponding holes in the 15-long beams.*

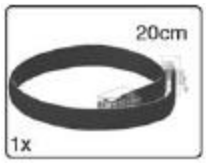


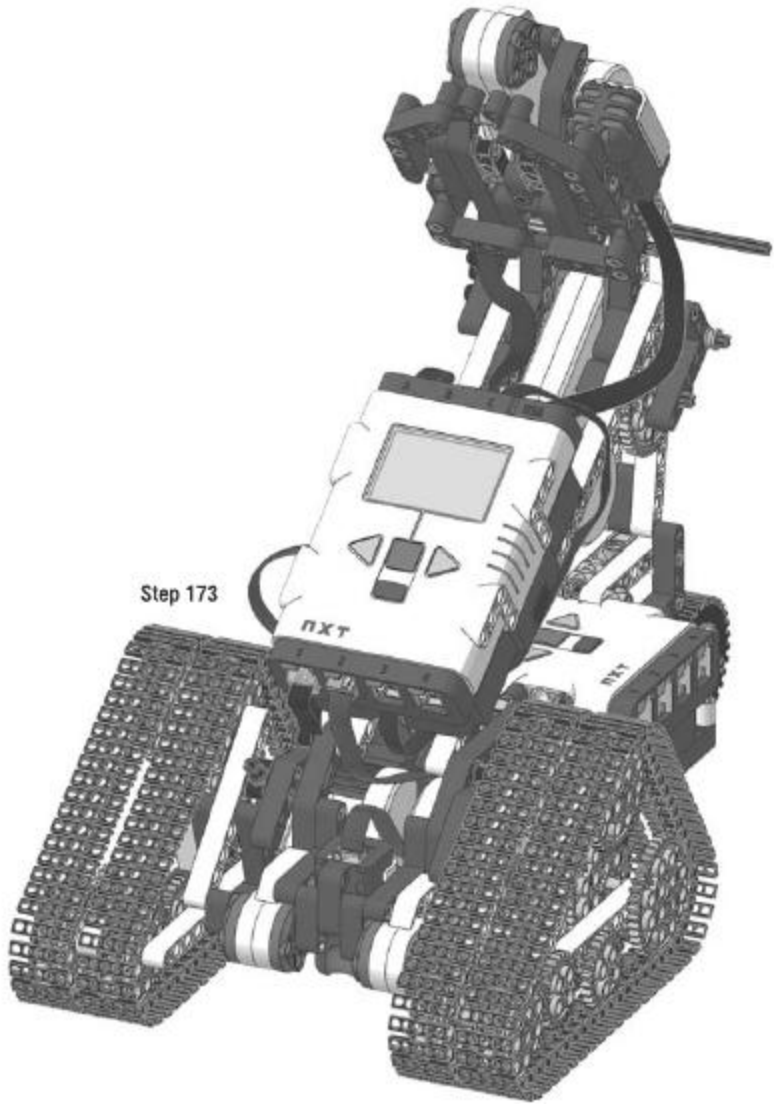
*Attach a 50cm (20 inch) cable to the Sound Sensor. Leave the other end floating; you'll attach it in the next steps.*



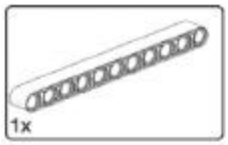




*Insert the master NXT brick in place. Notice that the two long pins go in the holes on the NXT's back. Attach the Sound Sensor cable to port 1. Attach the left tread's motor to port C and the right tread's motor to port A.*



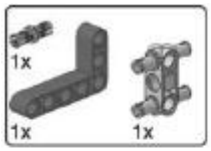



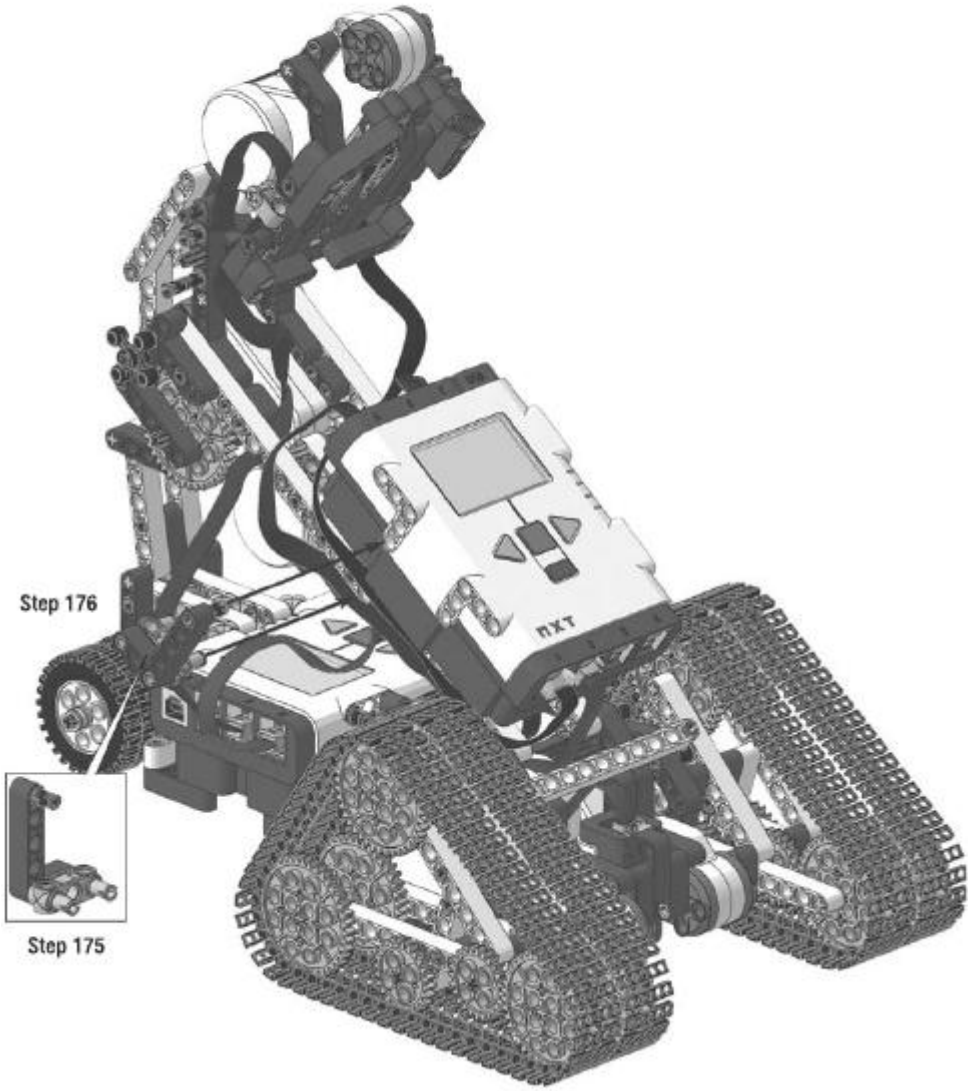

*Attach the line-tracking Light Sensor to master NXT input port 2, using a 20cm (8 inch) cable.*





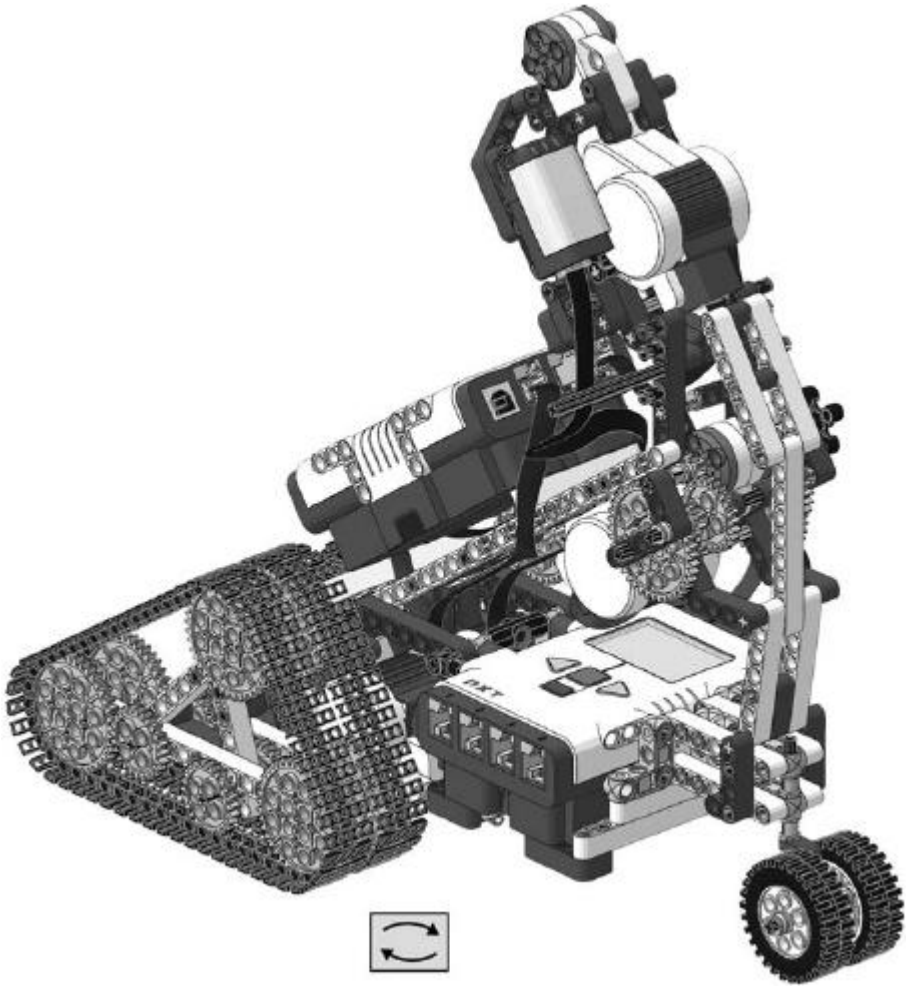
*Add a 13-long beam to hold the treads' assemblies together.*



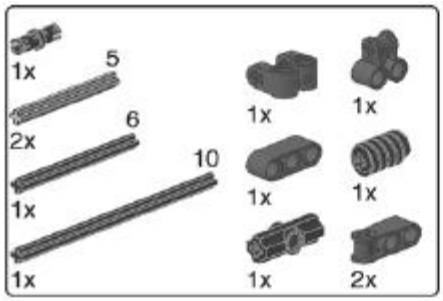



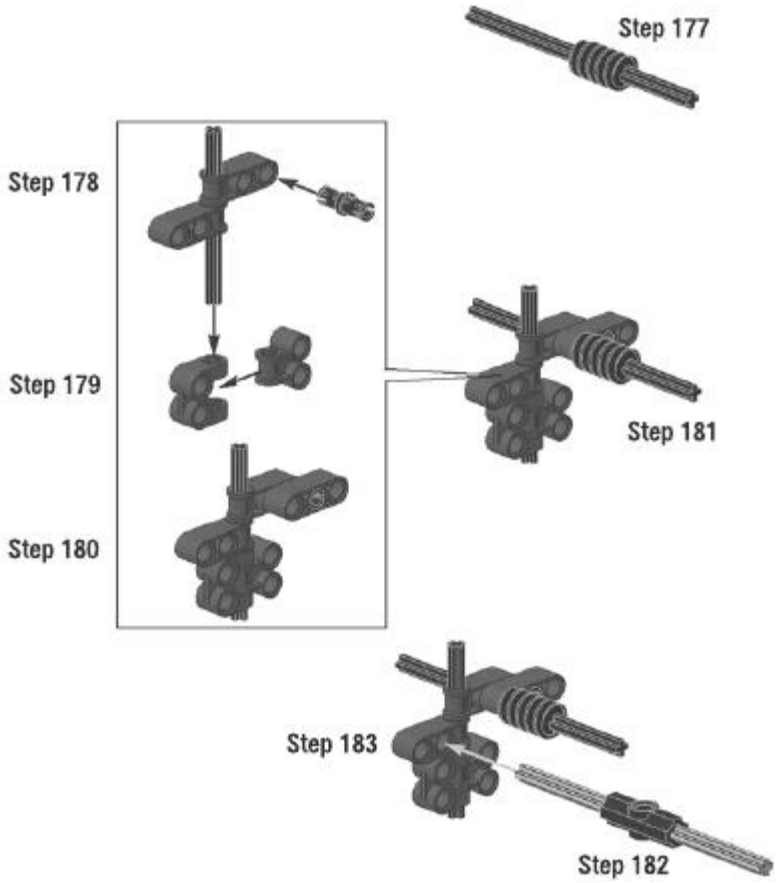

*Turn the model and block the NXT with the assembly shown.*


*The Sound Sensor cable must pass between the NXT and the block just added.*

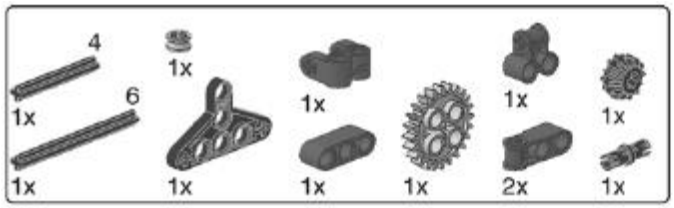


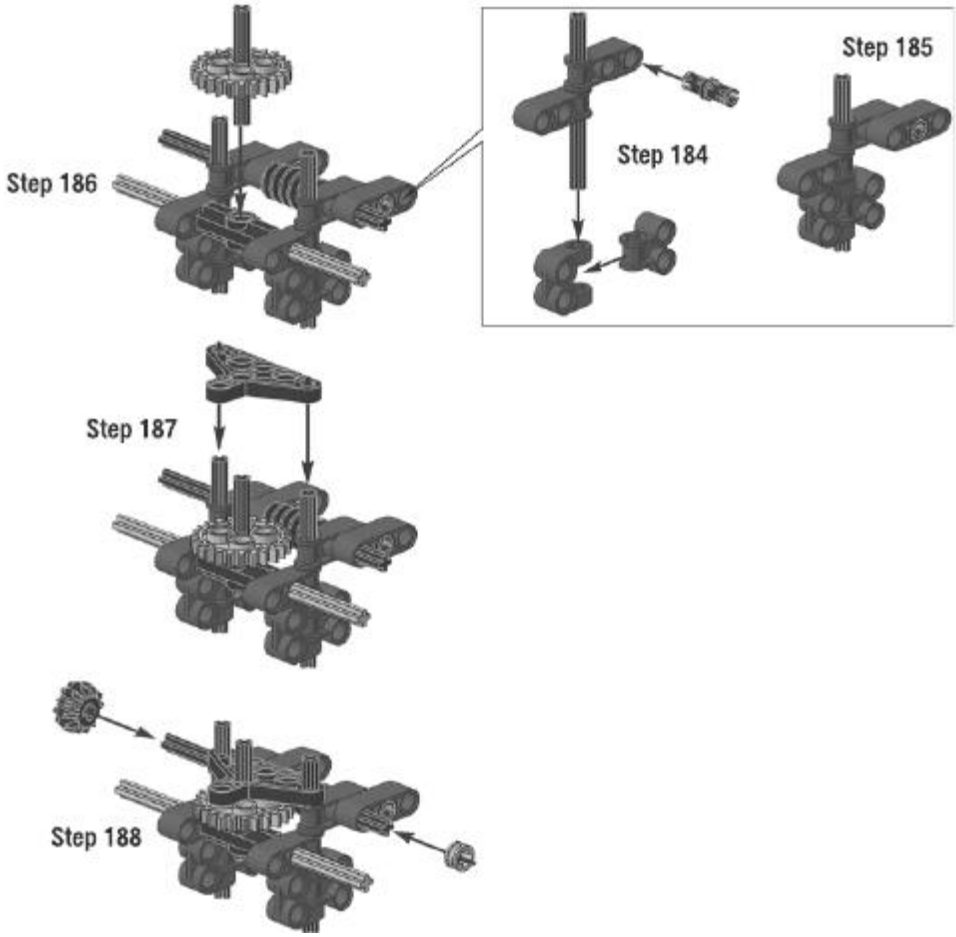
*The base and body assemblies are complete. The master NXT is attached to the body, leaving the space necessary for the Li-Ion battery pack.*



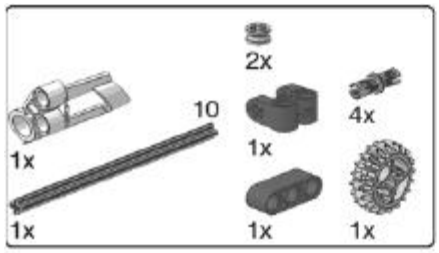



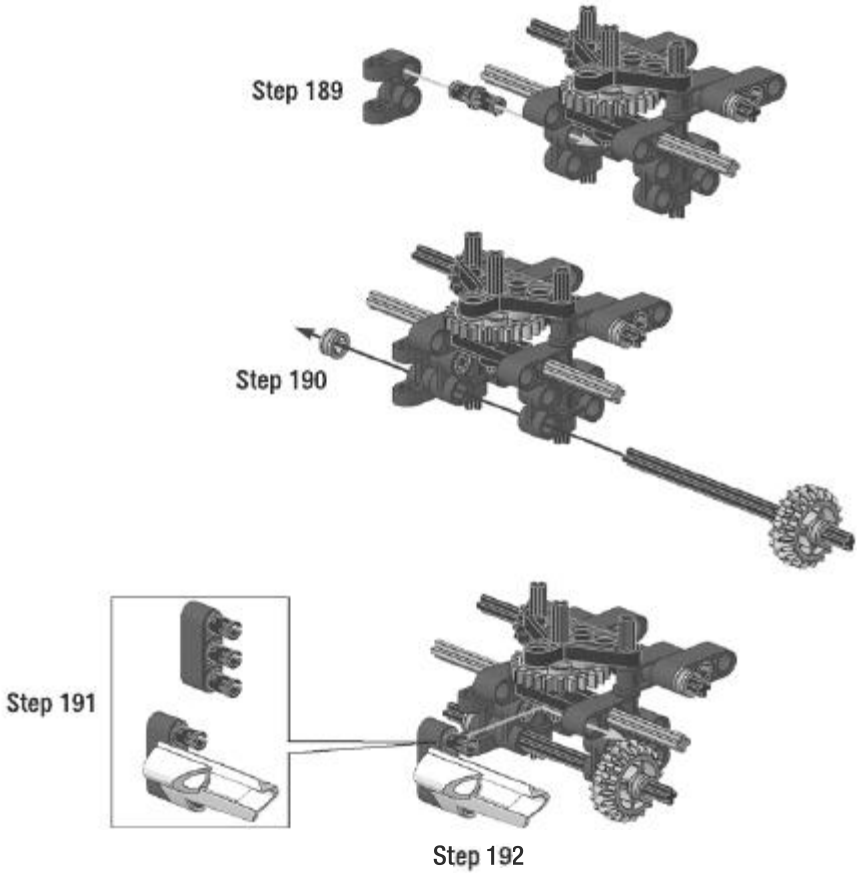

*Start building the shoulders.*



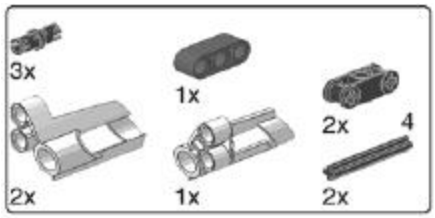



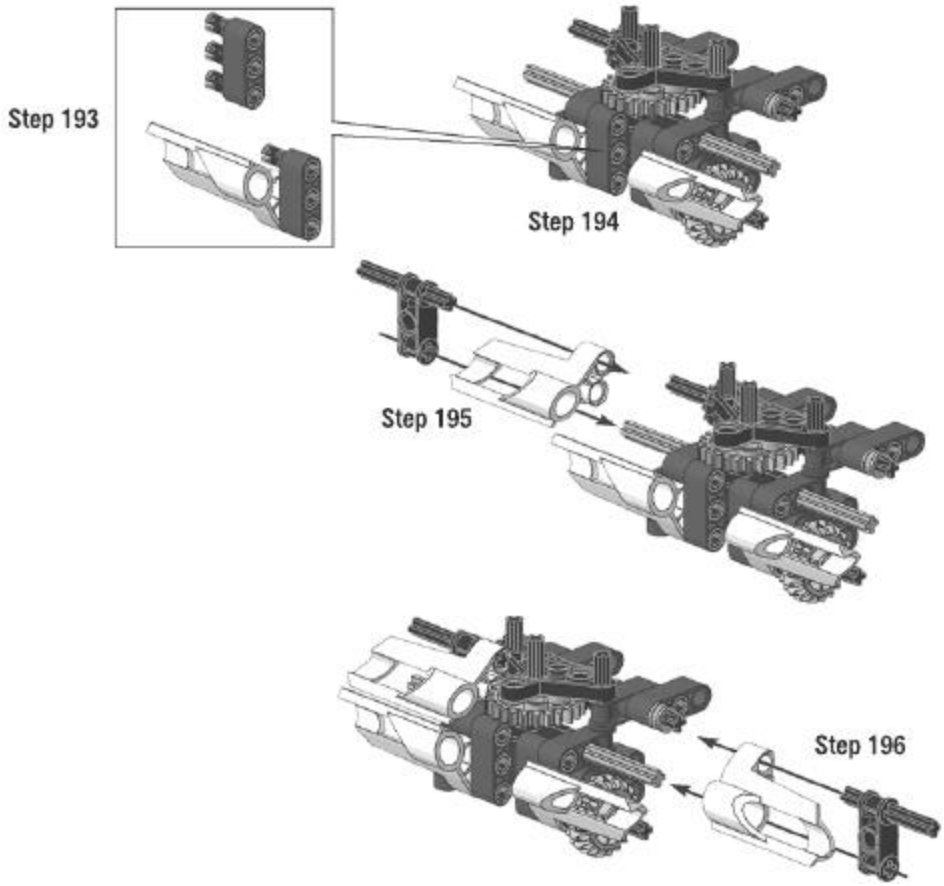

*You're building the mechanism to turn the neck.*



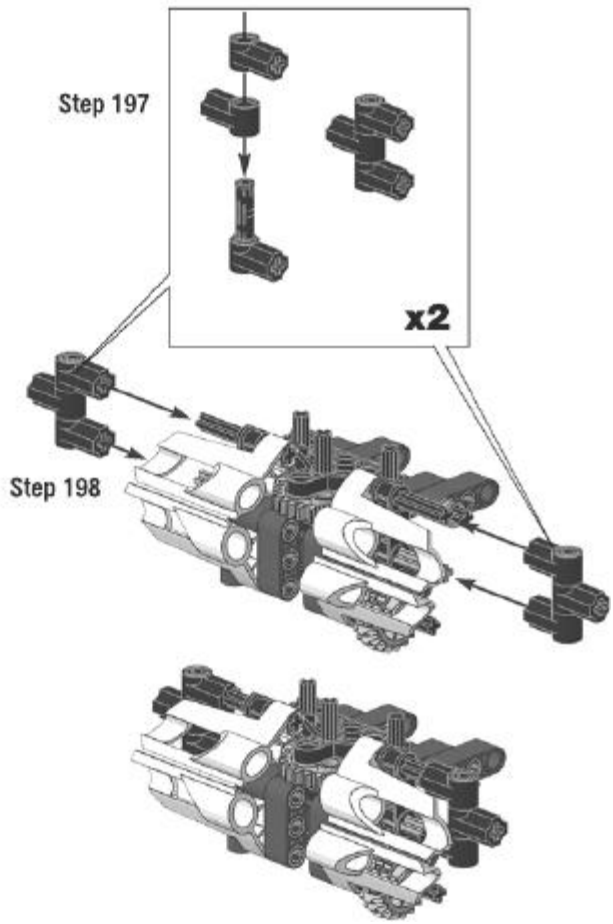




*Add the axle and the gear for moving the arms. Add the fairing panel #6.*

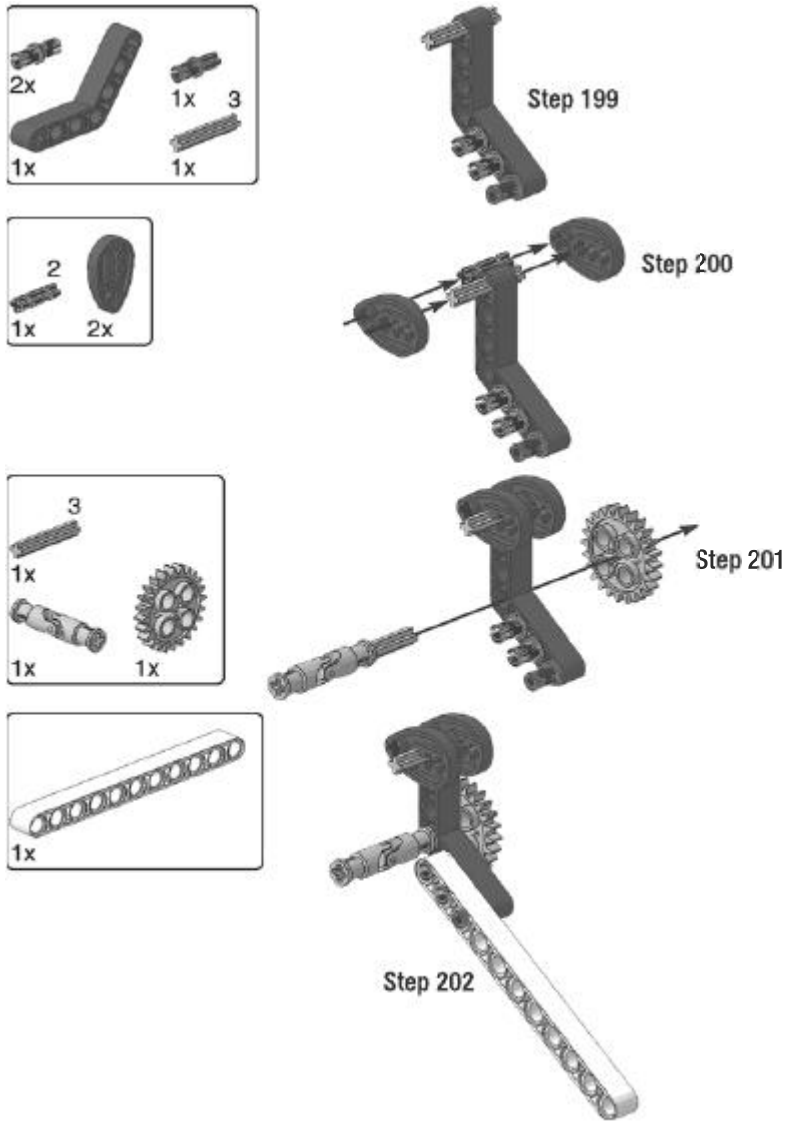



*Add the symmetric panel #5 and another couple panels #5 and #6 to shape the round shoulders of JohnNXT.*

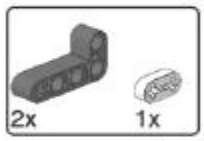



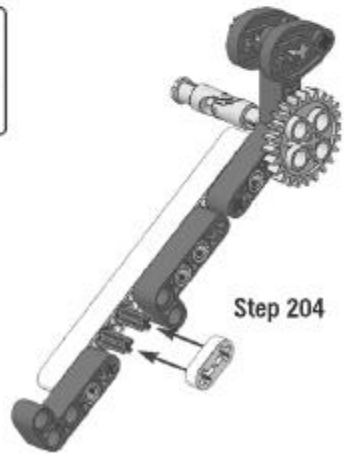

*Build the hinges for the arms.*



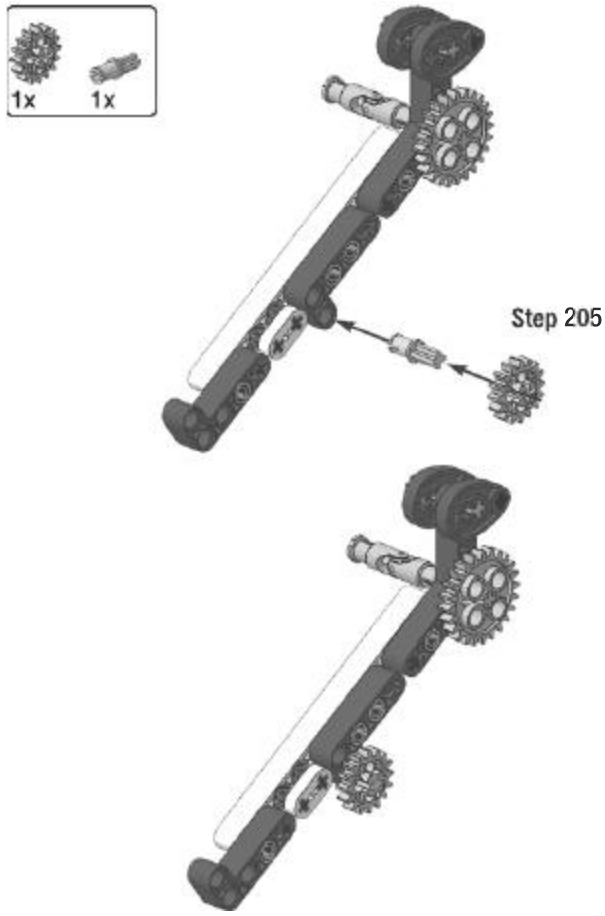

*Start building the left arm. The universal joint brings the movement over the shoulder hinge. Add an 11-long beam.*



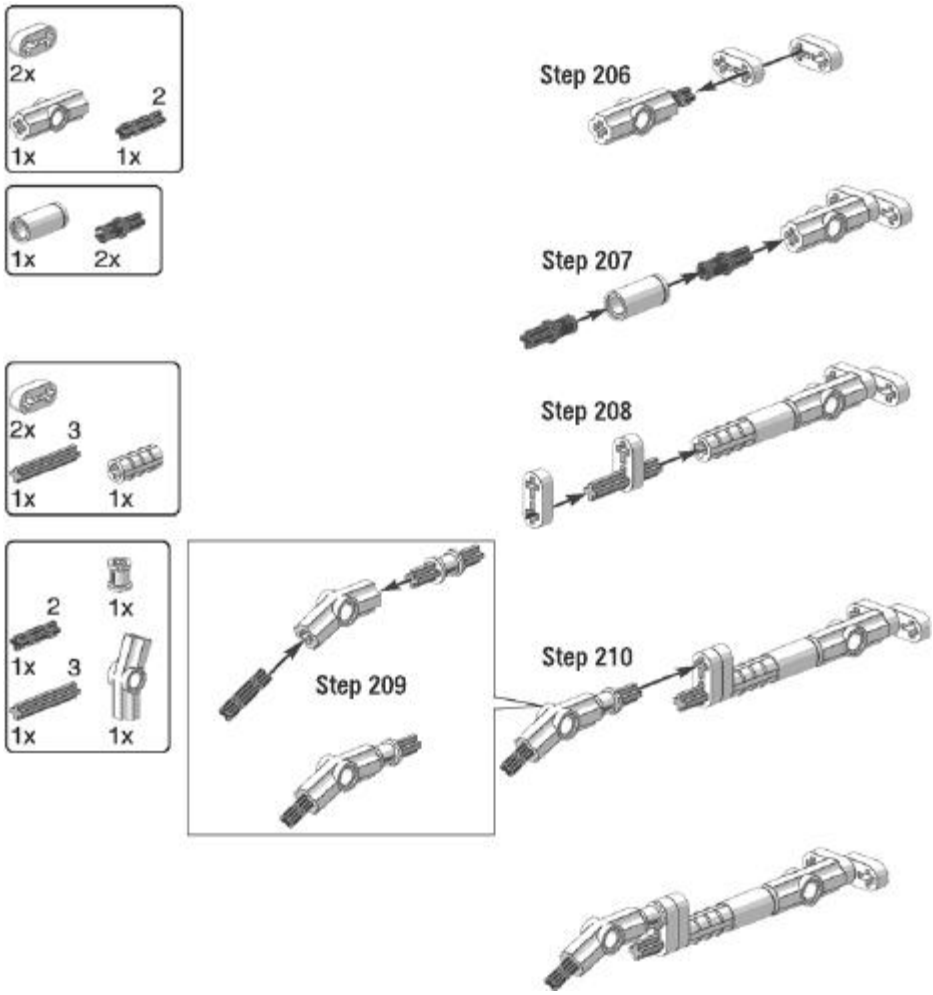


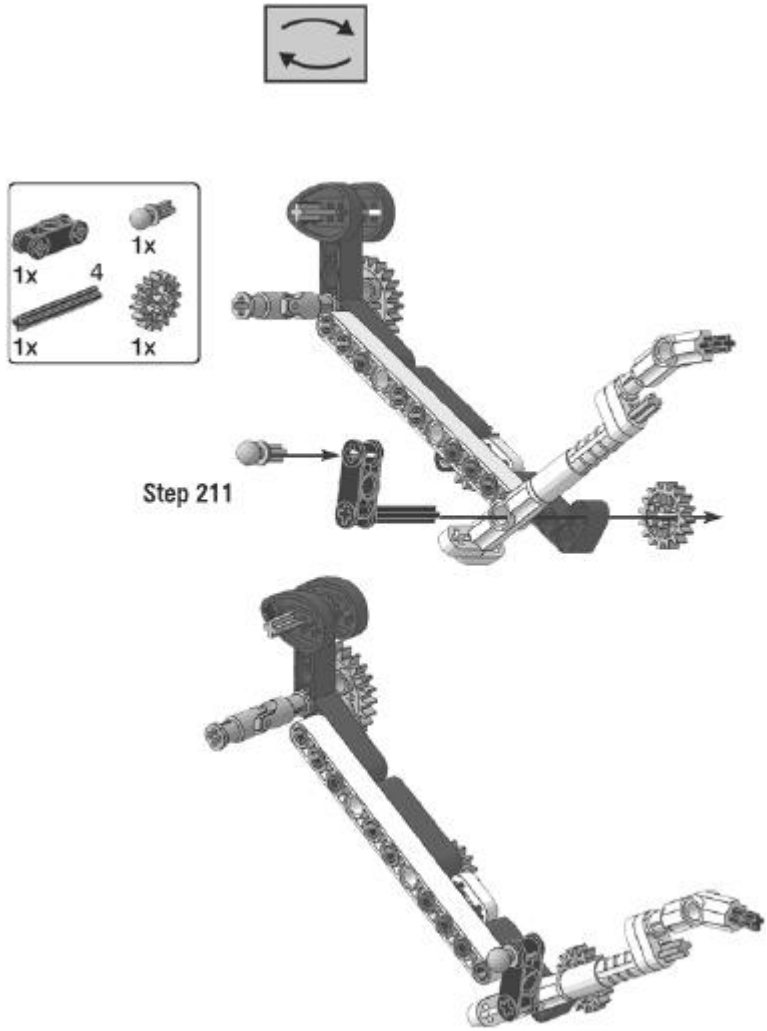

*Add three blue axle pins and the elbow liftarm. The small white 1 - 2 beam holds the rubber band on the elbow.*



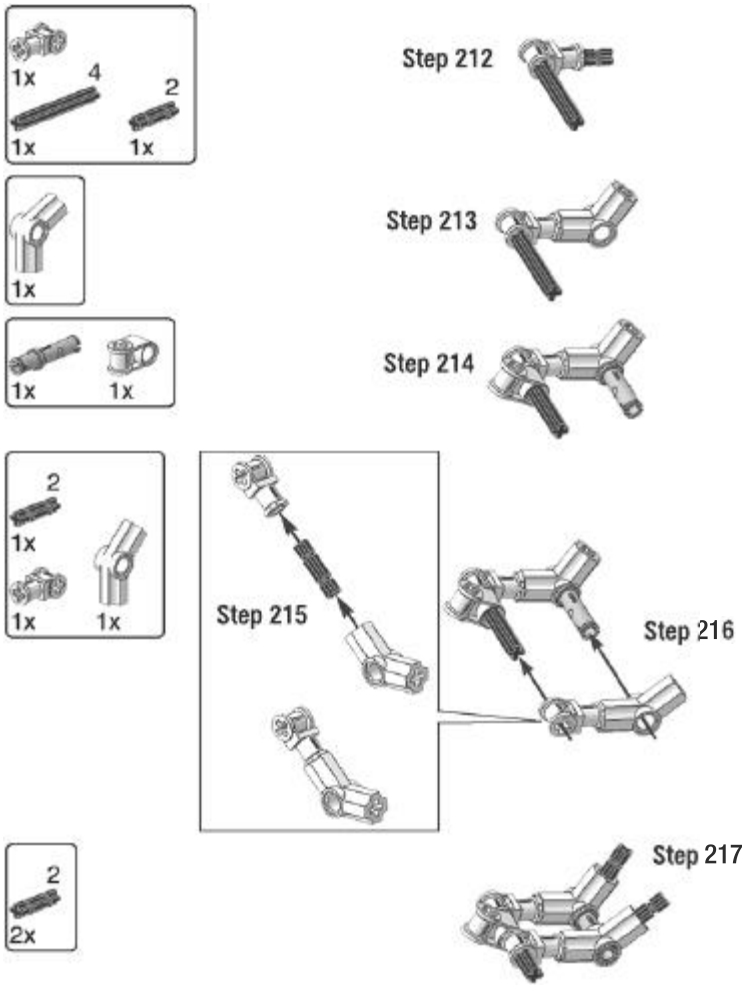

*Add the 16-tooth gear that keeps the arm chain in tension.*



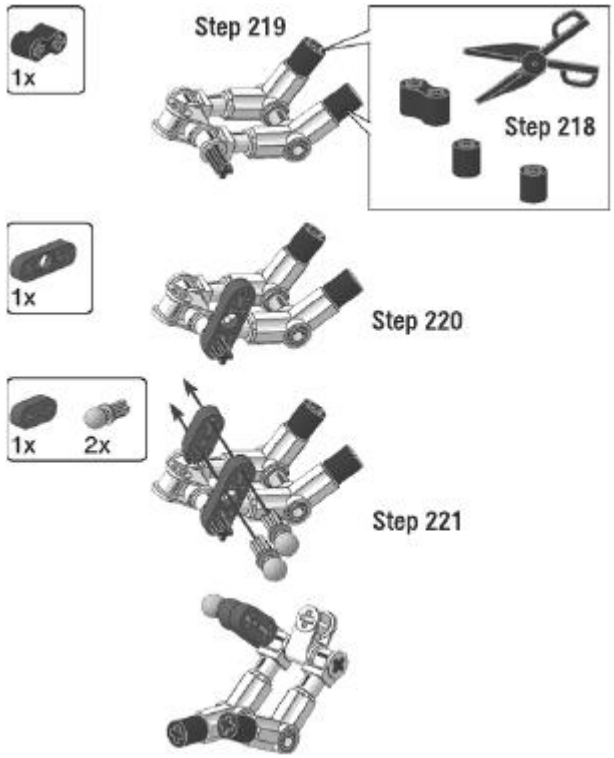

*Build the left forearm. In step 207, add two blue axle pins.*



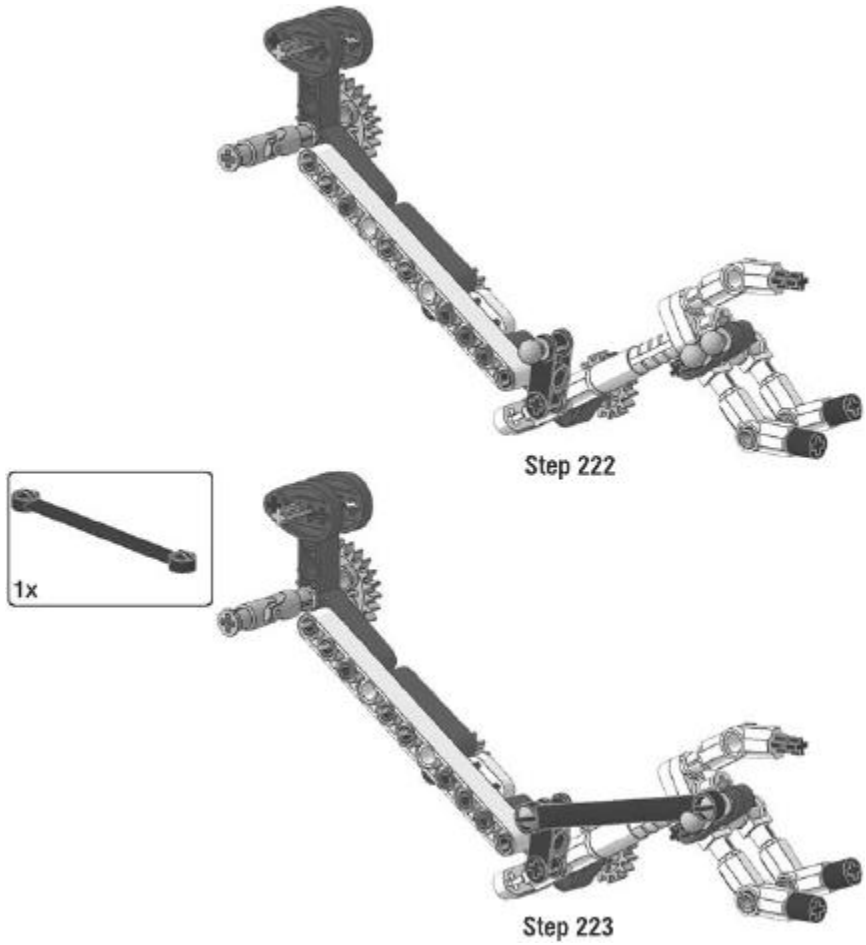

*Attach the forearm to the arm.*



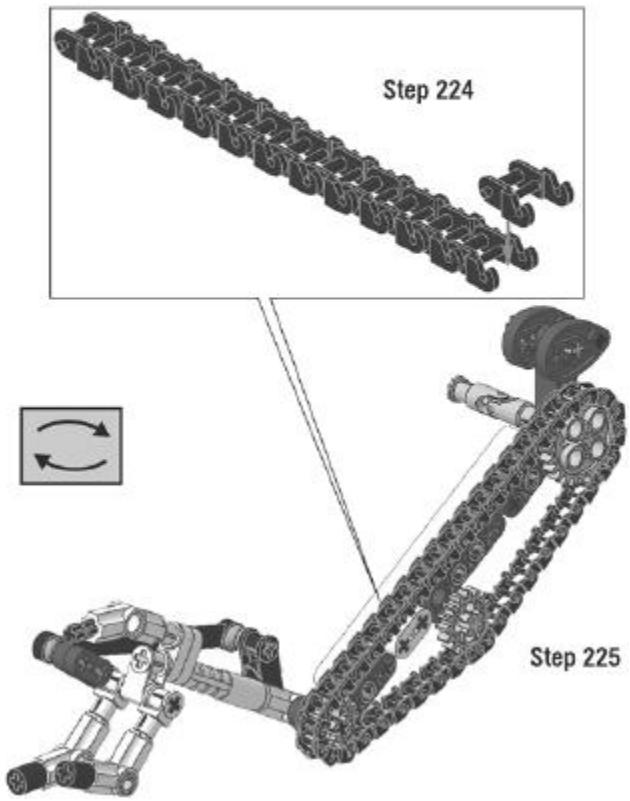

*Build the left hand.*



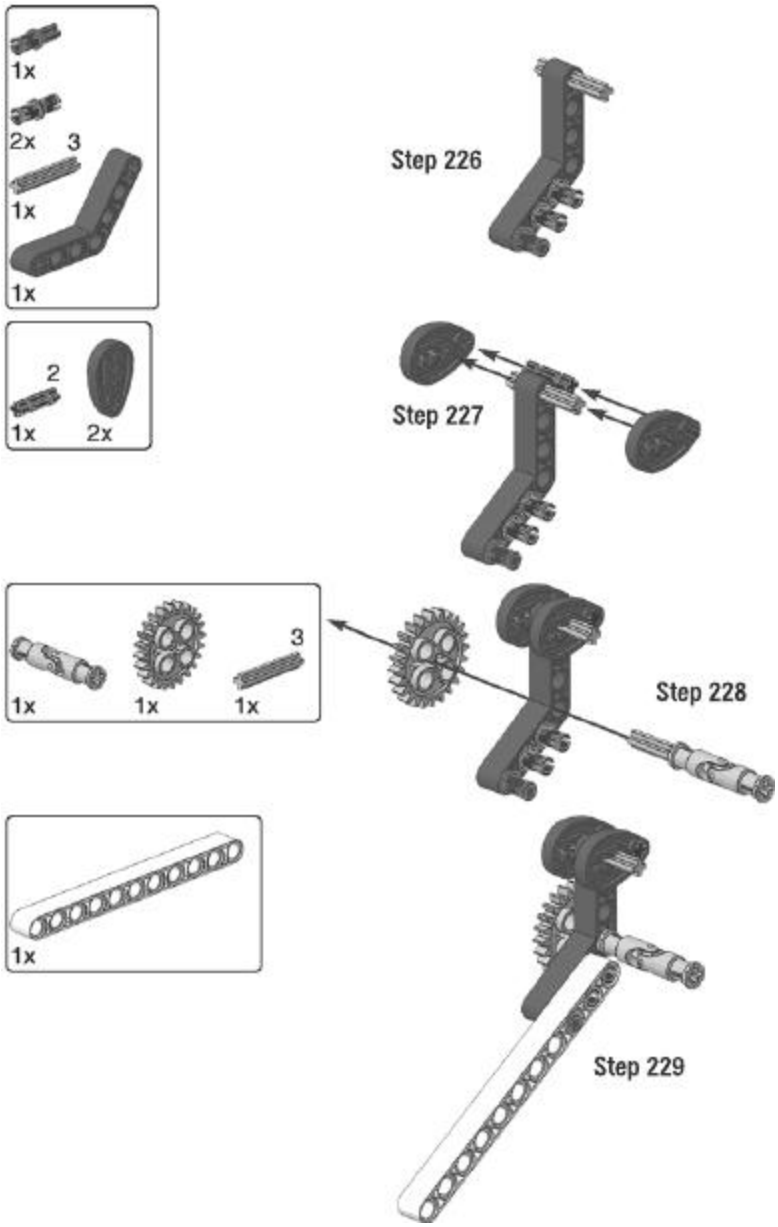

*If splitting a rubber joiner in two does not bother you, prepare the finger grippers as shown. Otherwise, skip this step; the hand won't have friction on grasped objects.*



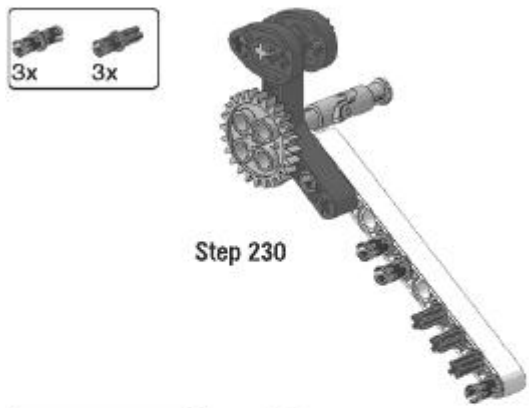

*Attach the hand to the forearm and add a long steering link.*

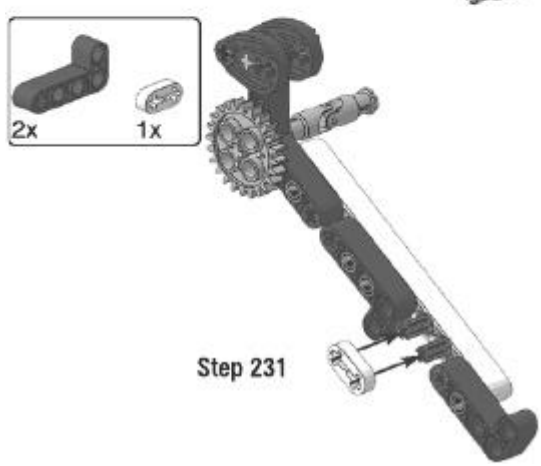




*Attach 42 chain links and wrap them on the arms' gears as shown. The left arm is complete.*

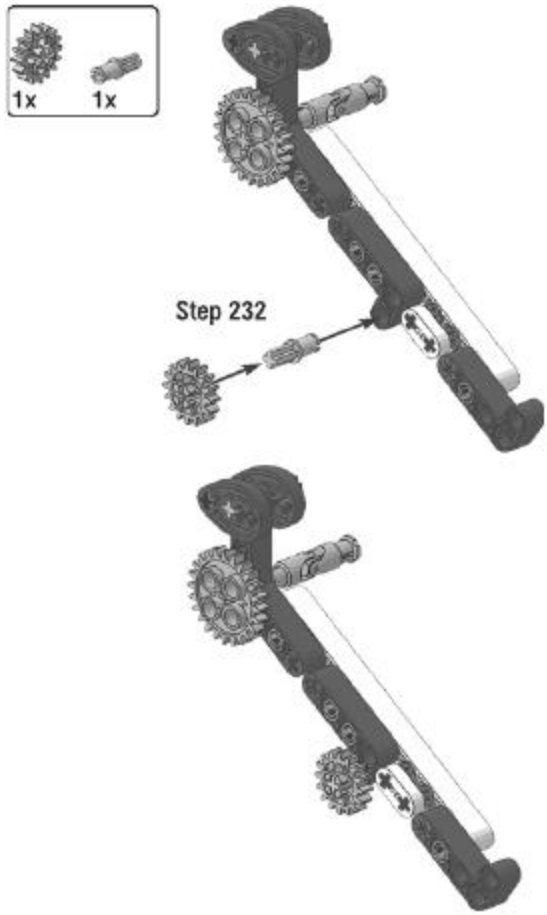



*Start building the right arm. The cardan joint brings the movement over the shoulder hinge. Add an 11-long beam.*

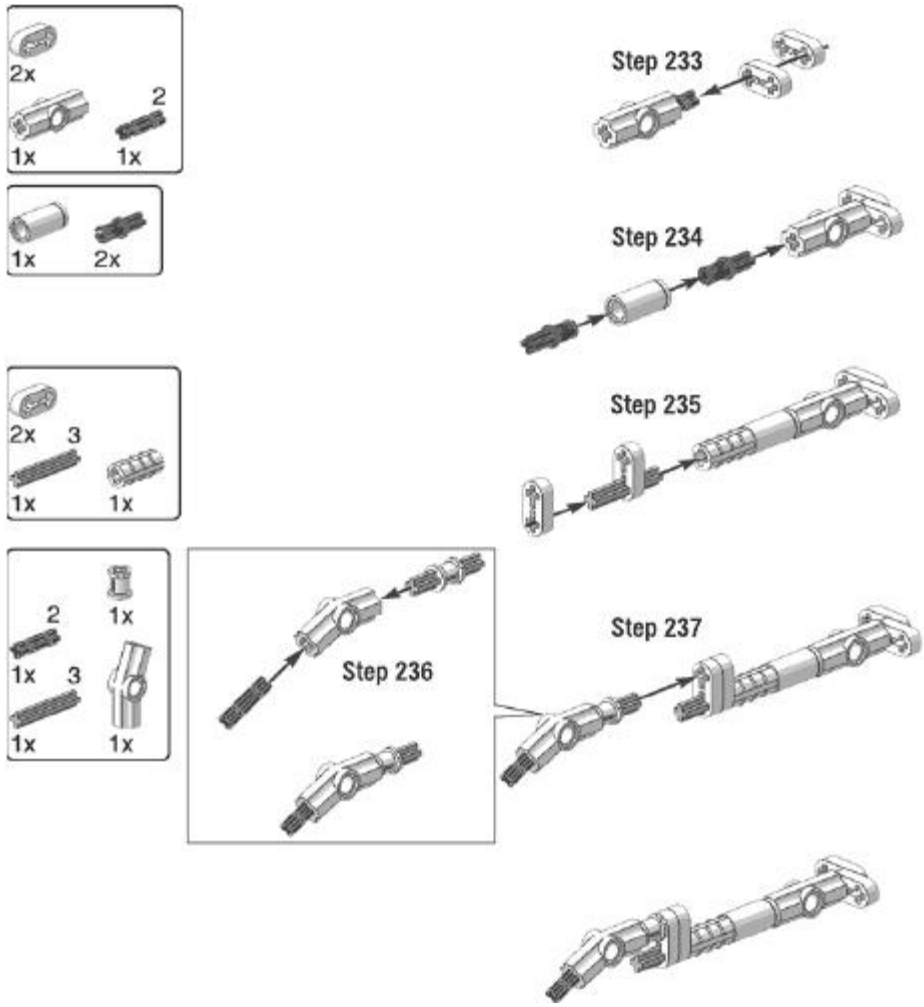


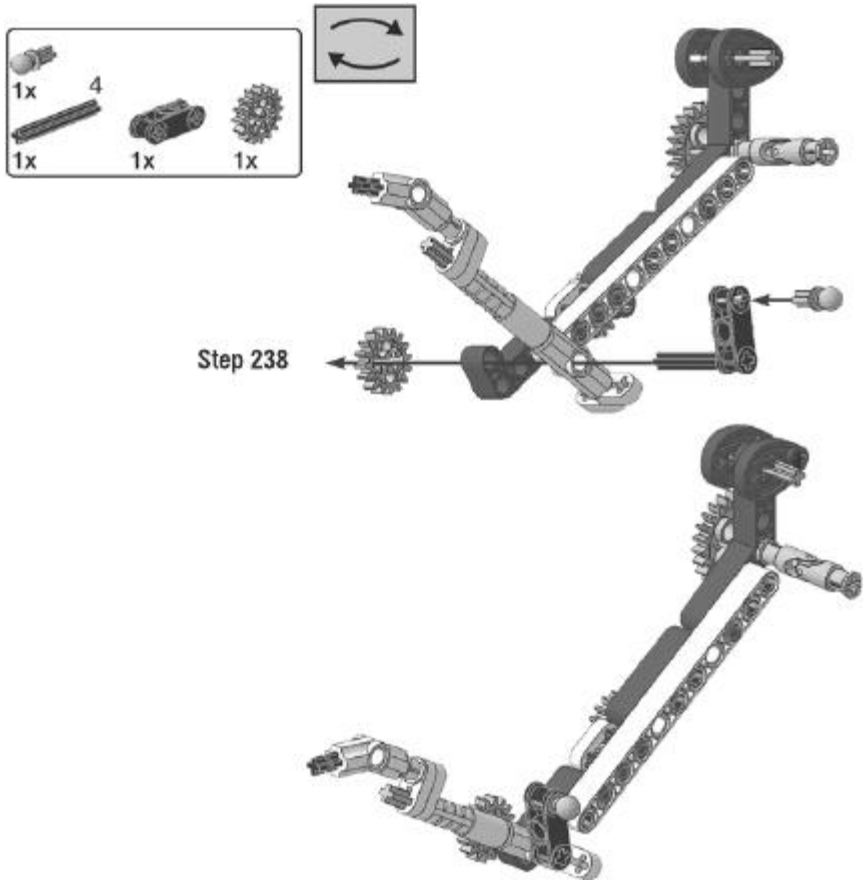

*Add three blue axle pins and the elbow liftarm. The small white 1 - 2 beam holds the rubber band on the elbow.*



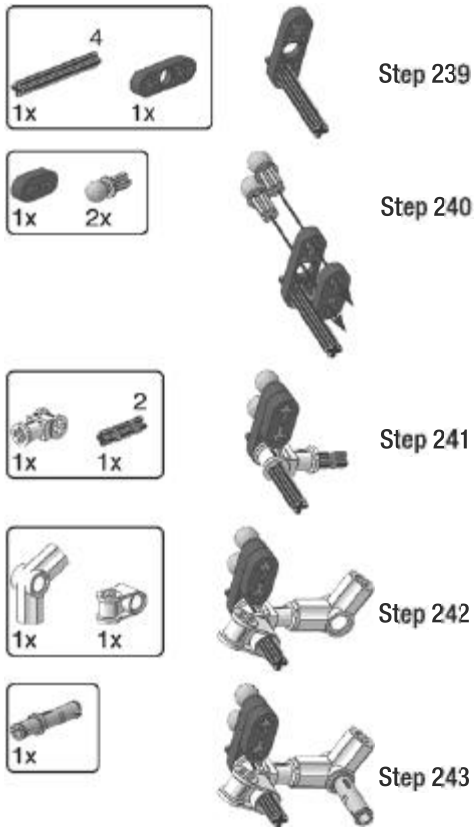

*Add the 16-tooth gear that keeps the arm chain in tension.*



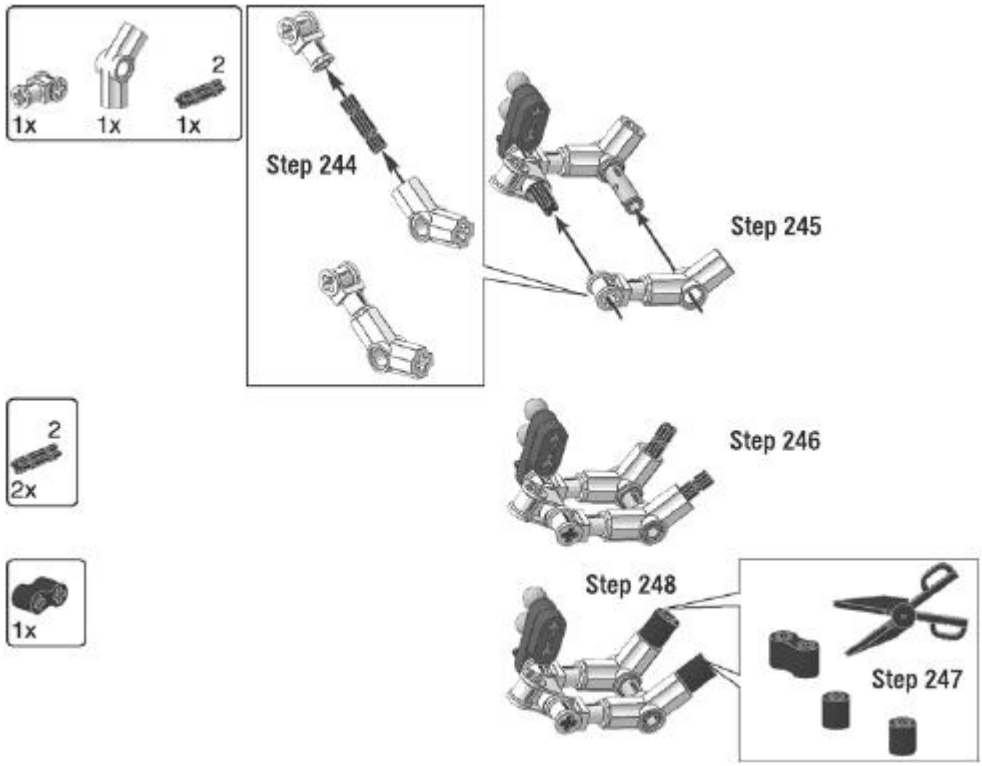

*Build the right forearm. In Step 234, add two blue axle pins.*



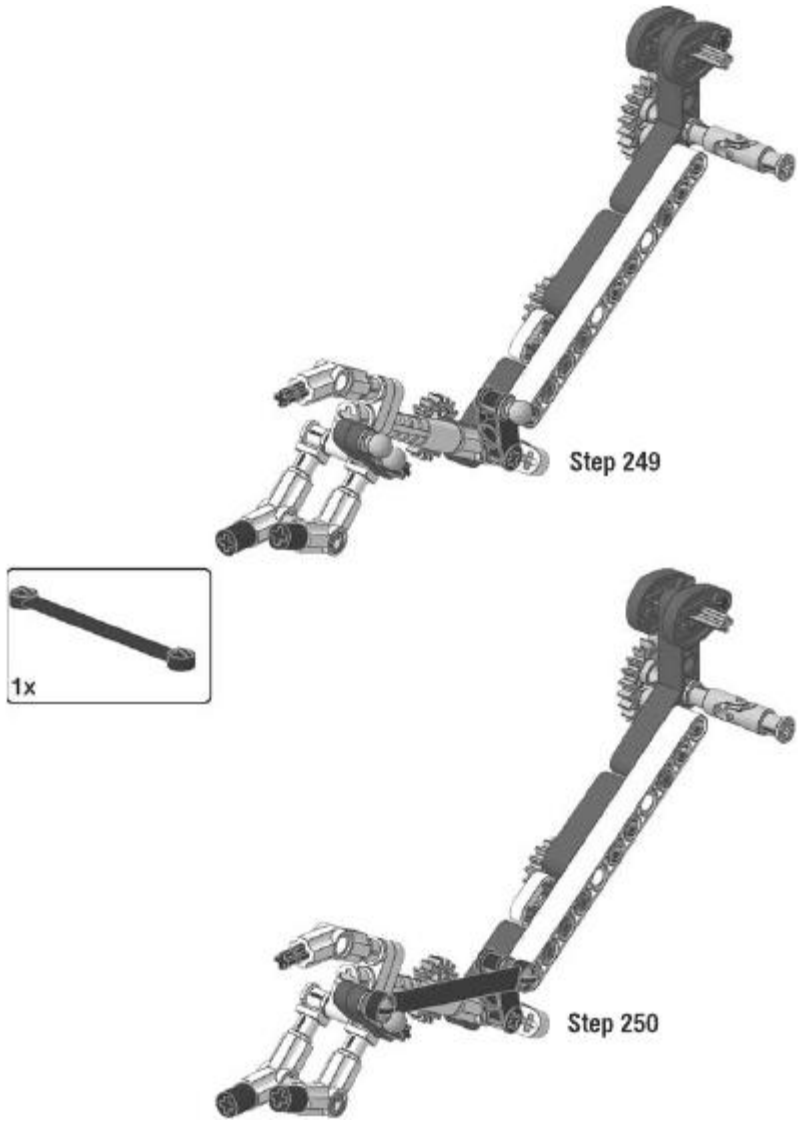

*Attach the forearm to the arm.*



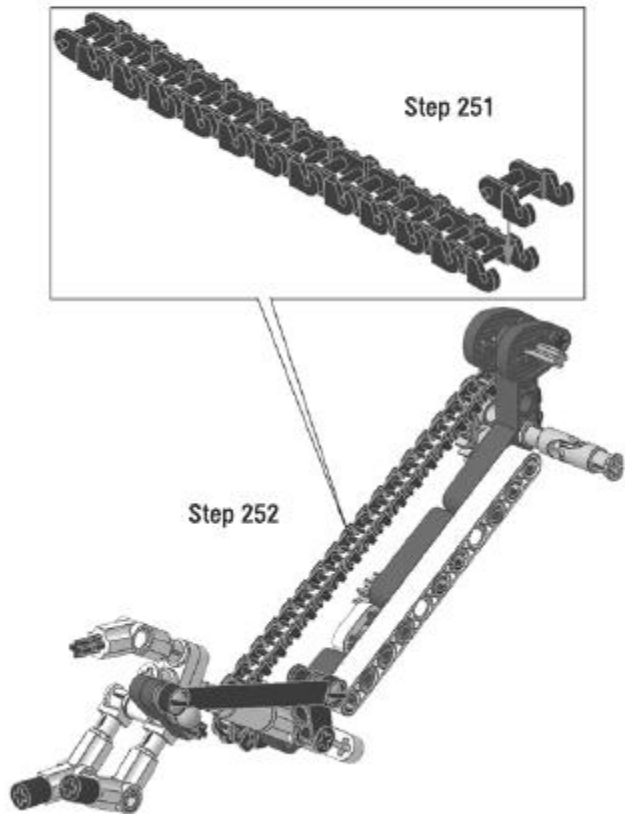

*Start building the right hand.*




*Again, if cutting LEGO parts does not bother you, prepare the finger grippers, splitting the rubber joiner as shown. Otherwise, skip this step; the hand won't have friction on grasped objects.*




*Attach the hand to the forearm and add a long steering link.*


*Join 42 chain links and wrap them on the arms' gears as shown. Make sure to align the forearm to the left one before adding the chain to the gears. The right arm is complete.*

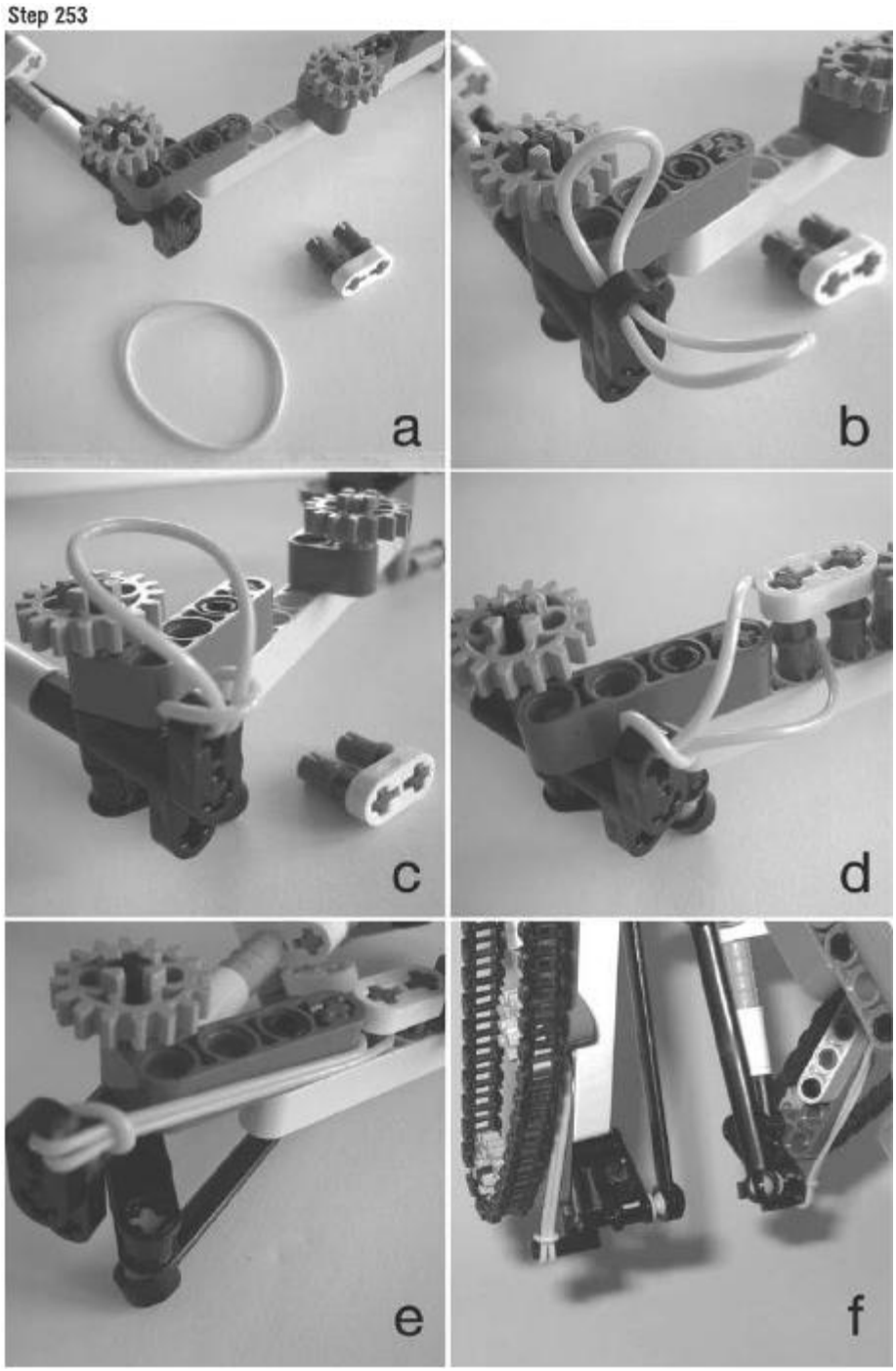
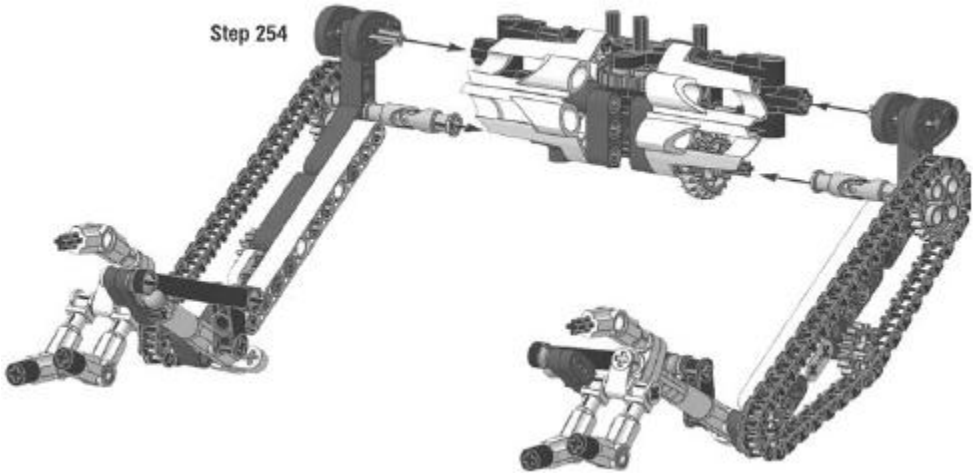
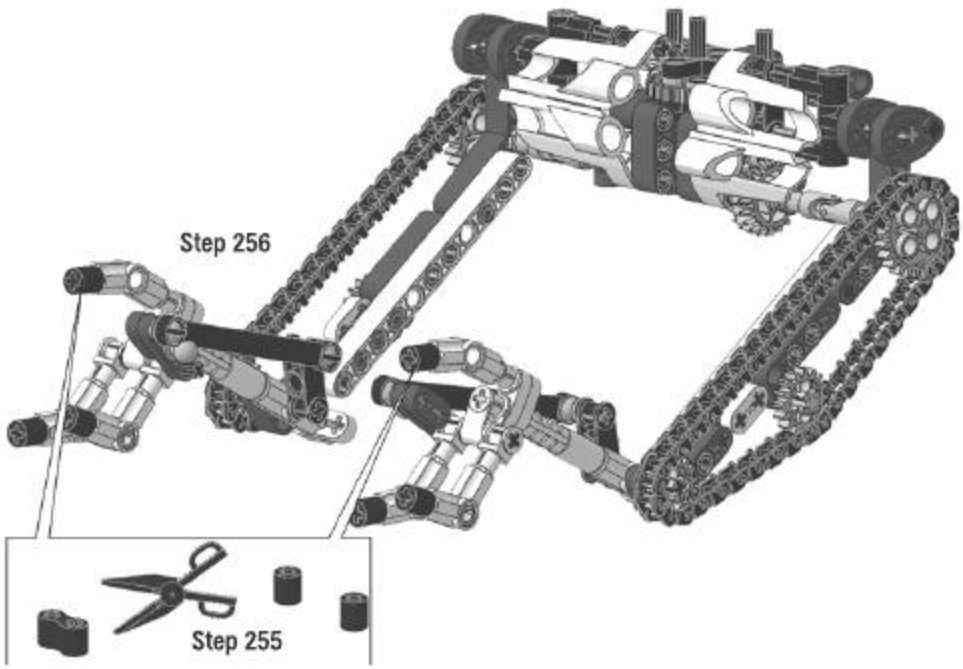


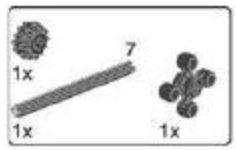

**Figure 8-11.** *This composite figure shows how to insert a LEGO yellow rubber band in the left elbow. Repeat this procedure also for the right elbow, before attaching the arms to the shoulders.*

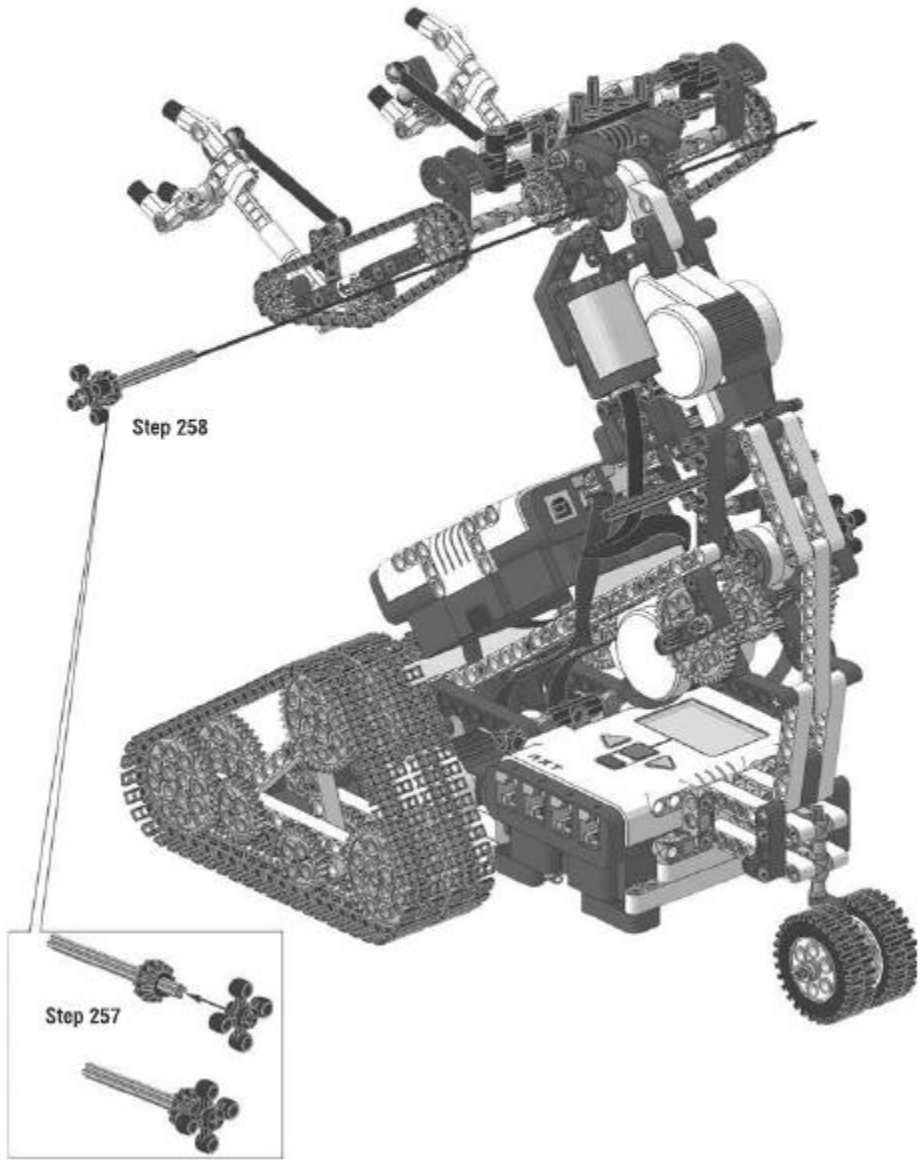



*Attach both arms to the shoulders.*

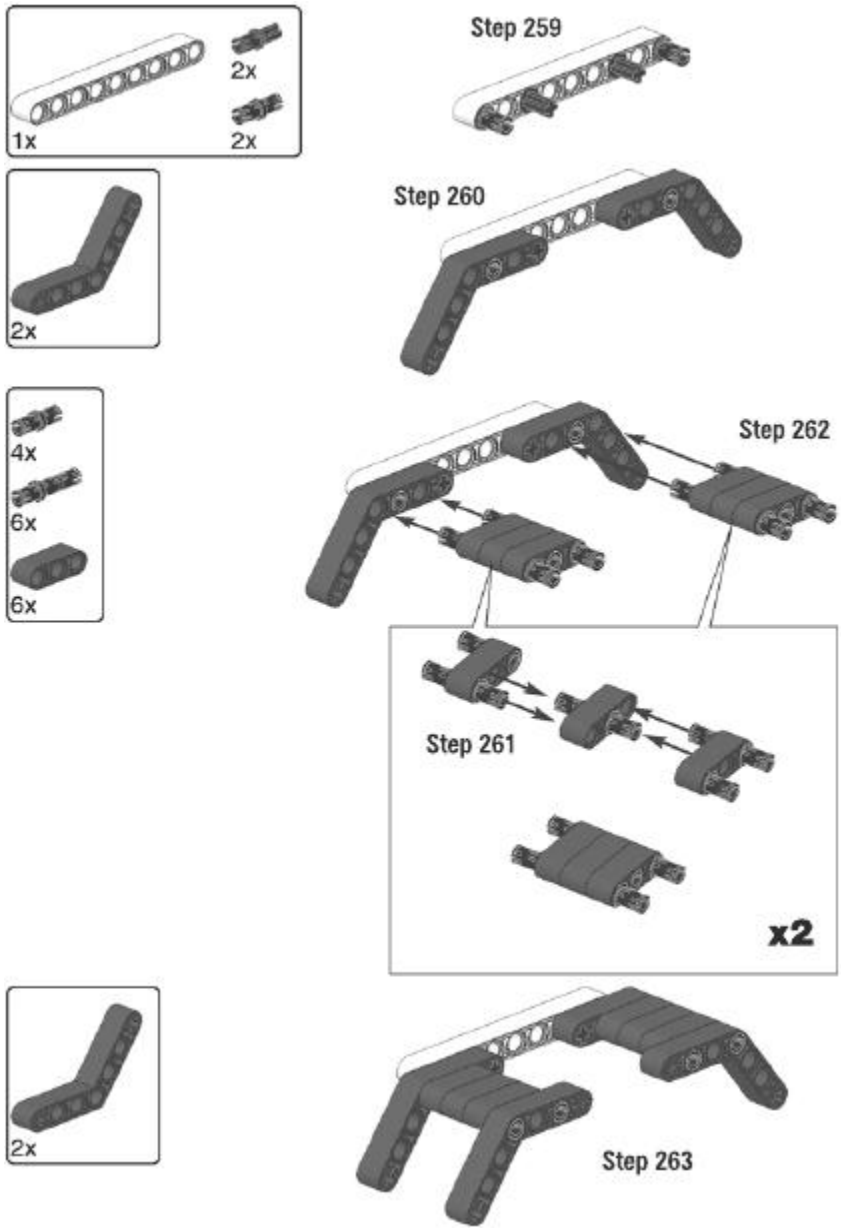




*This is the last nerve-wracking rubber joiner cutting. Again, you can avoid this step.*

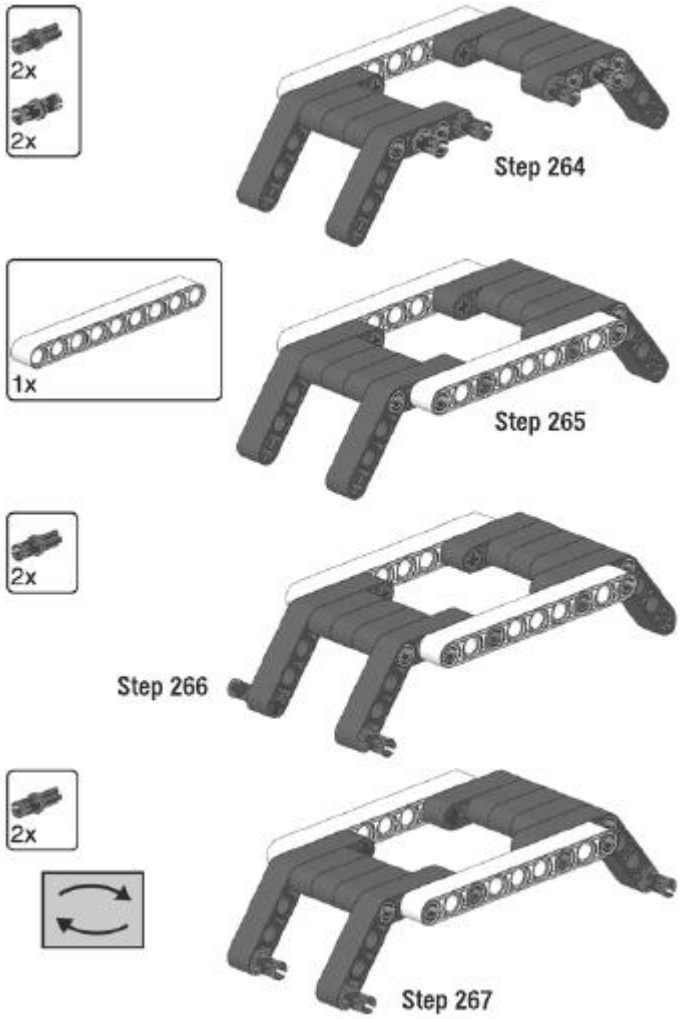


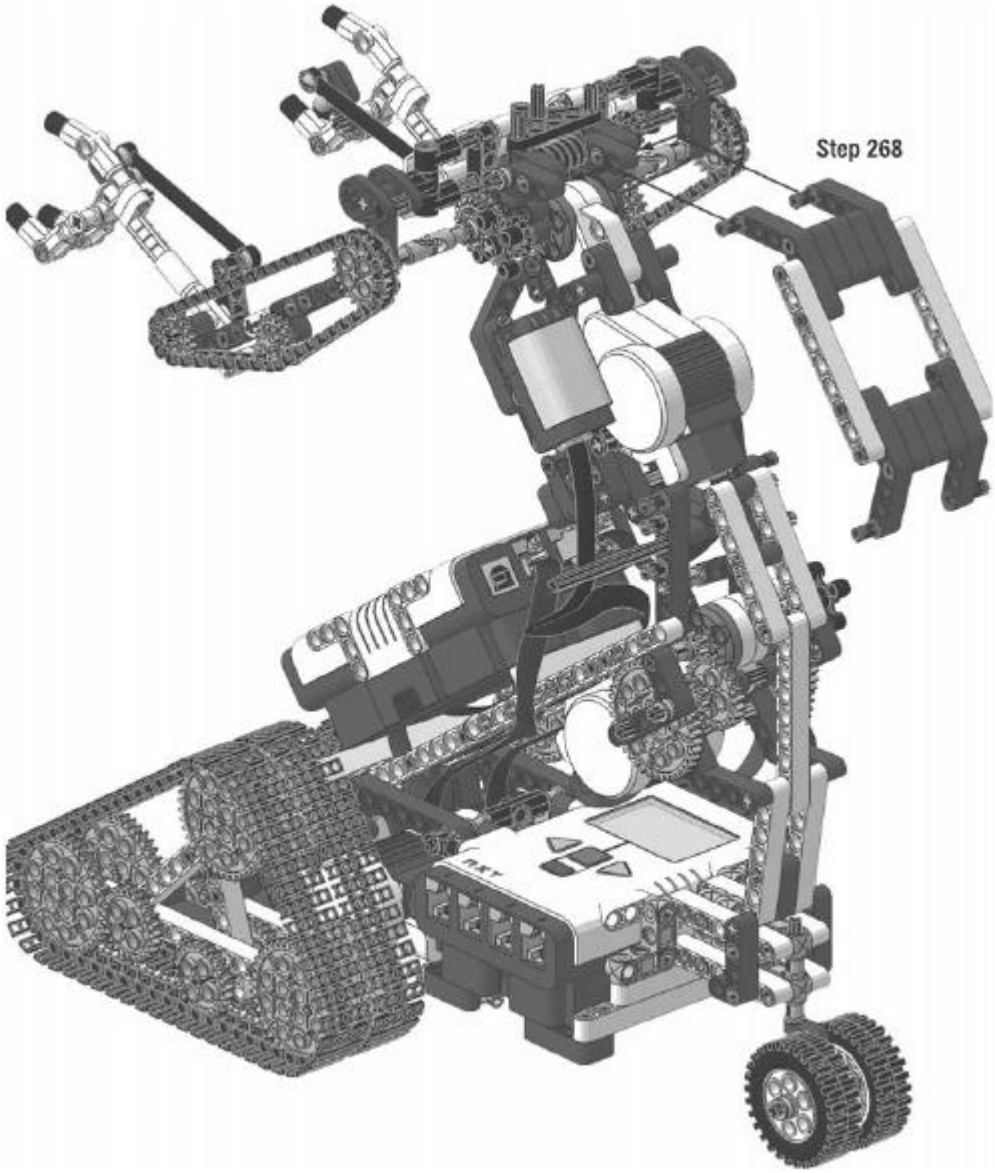

*Attach the shoulders to the whole model, aligning the bottom holes with the motor shaft. Insert the axle that brings the movement to the arms and blocks the shoulders in place. You can turn the knob to move the arms manually.*



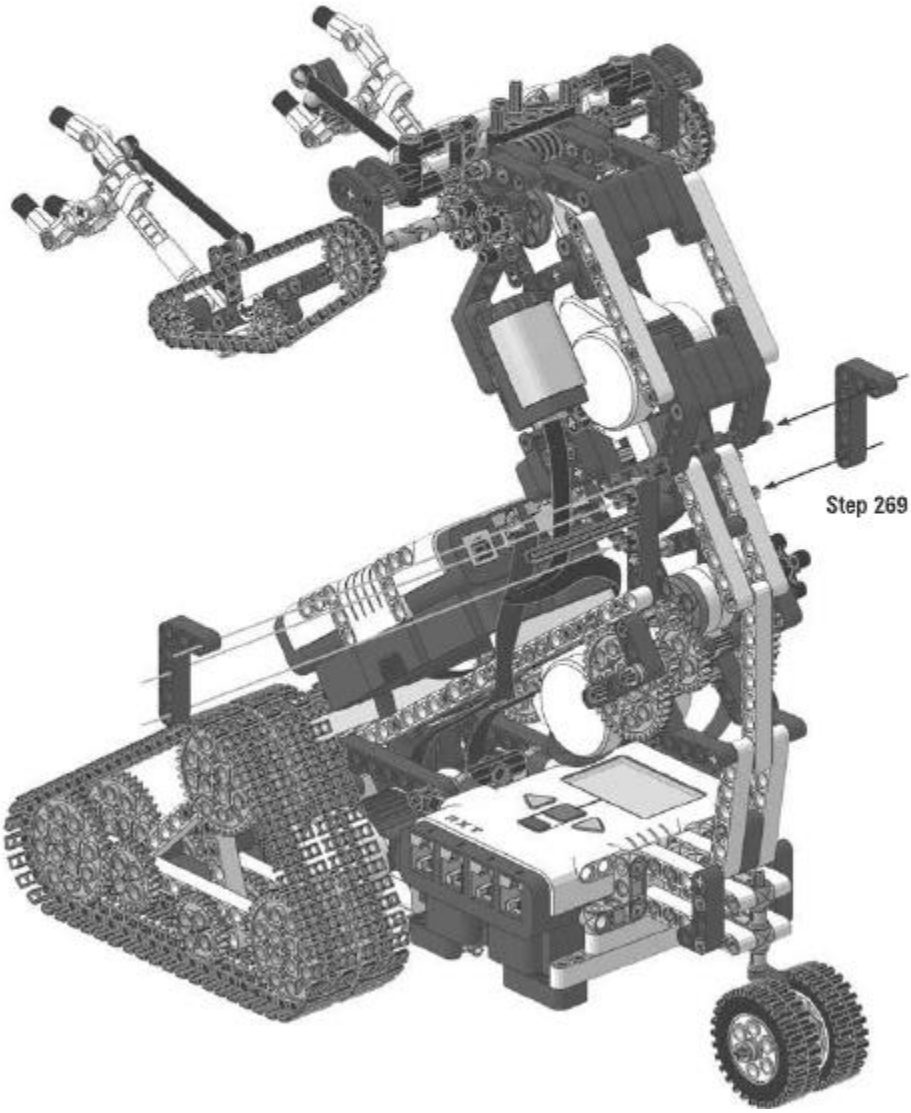

*Start building the back of JohnNXT.*



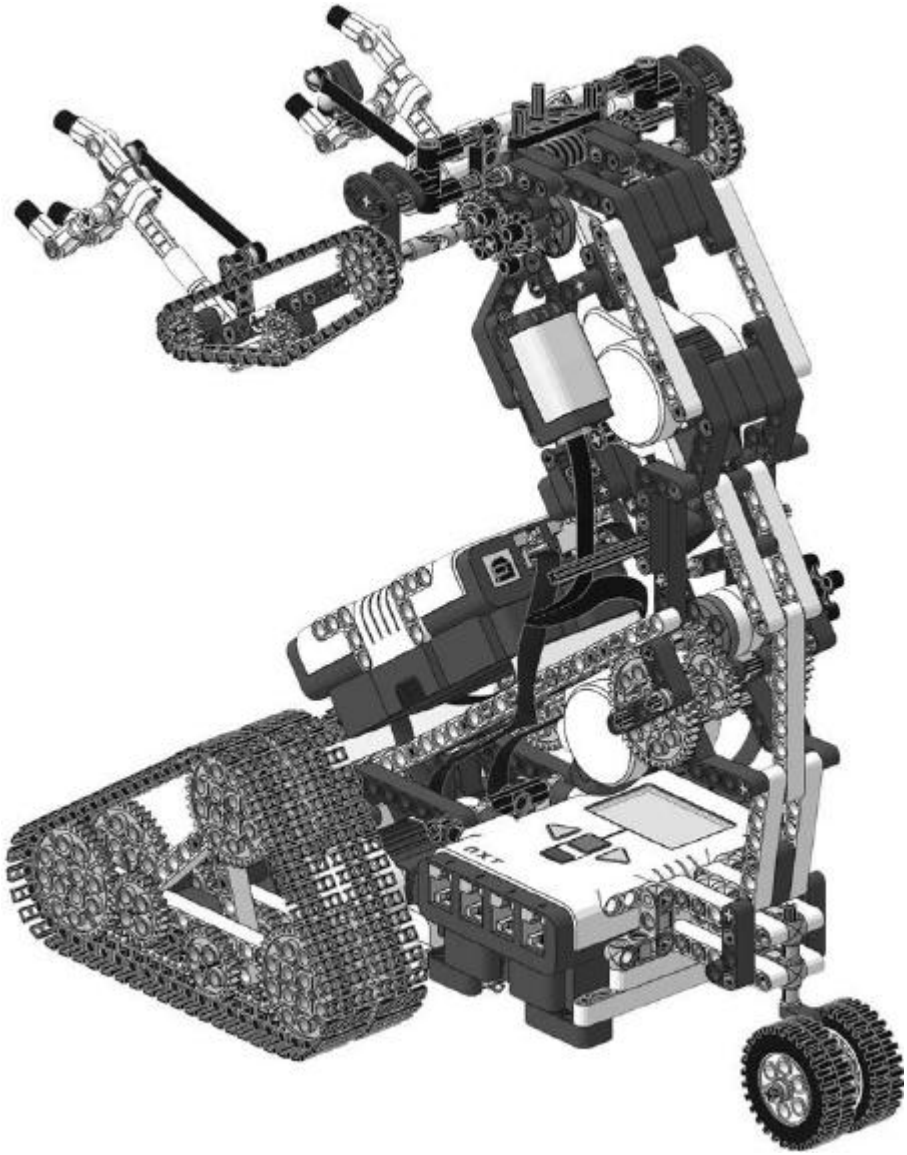

*Complete the back of the robot.*



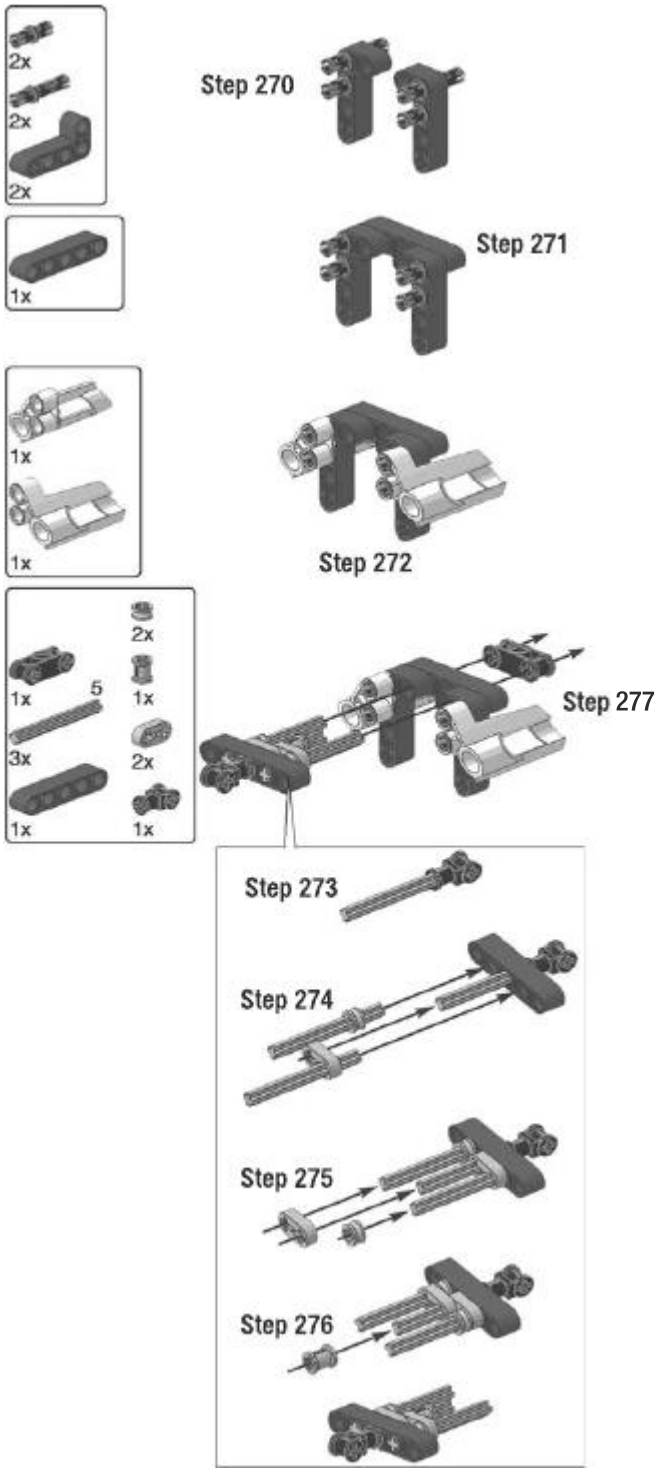

*Attach the back assembly to the shoulders.*

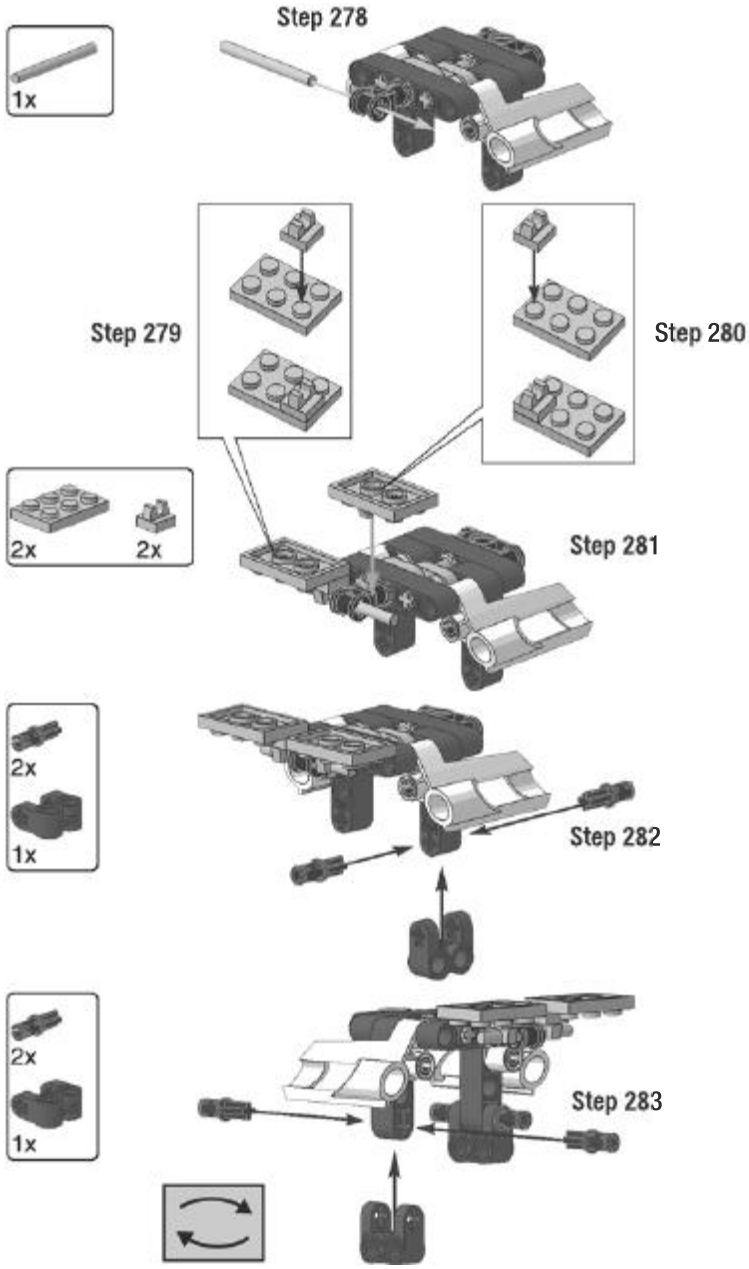

*These two bent beams complete the upper body's structure.*



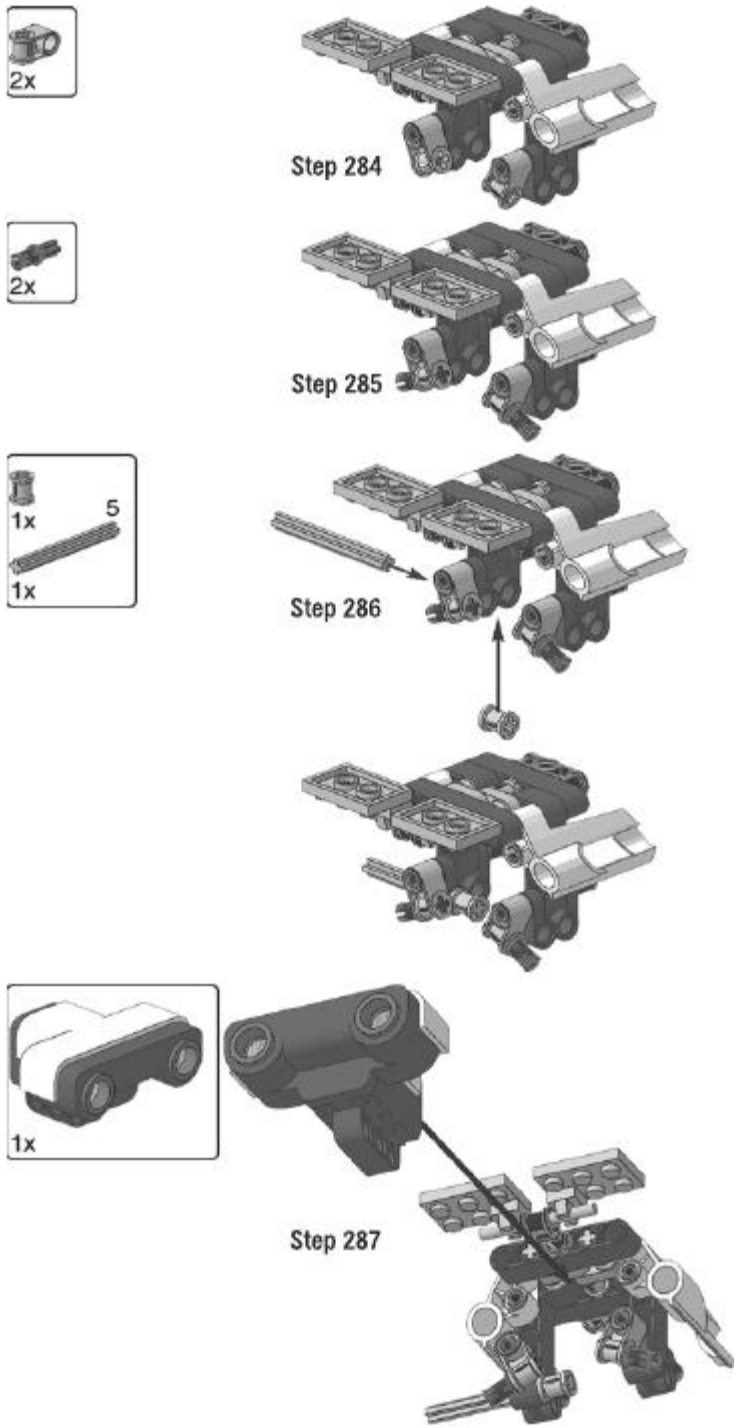
*The upper body is a parallelogram structure. This way, the shoulders can move up and down, keeping the same inclination with respect to the ground.*



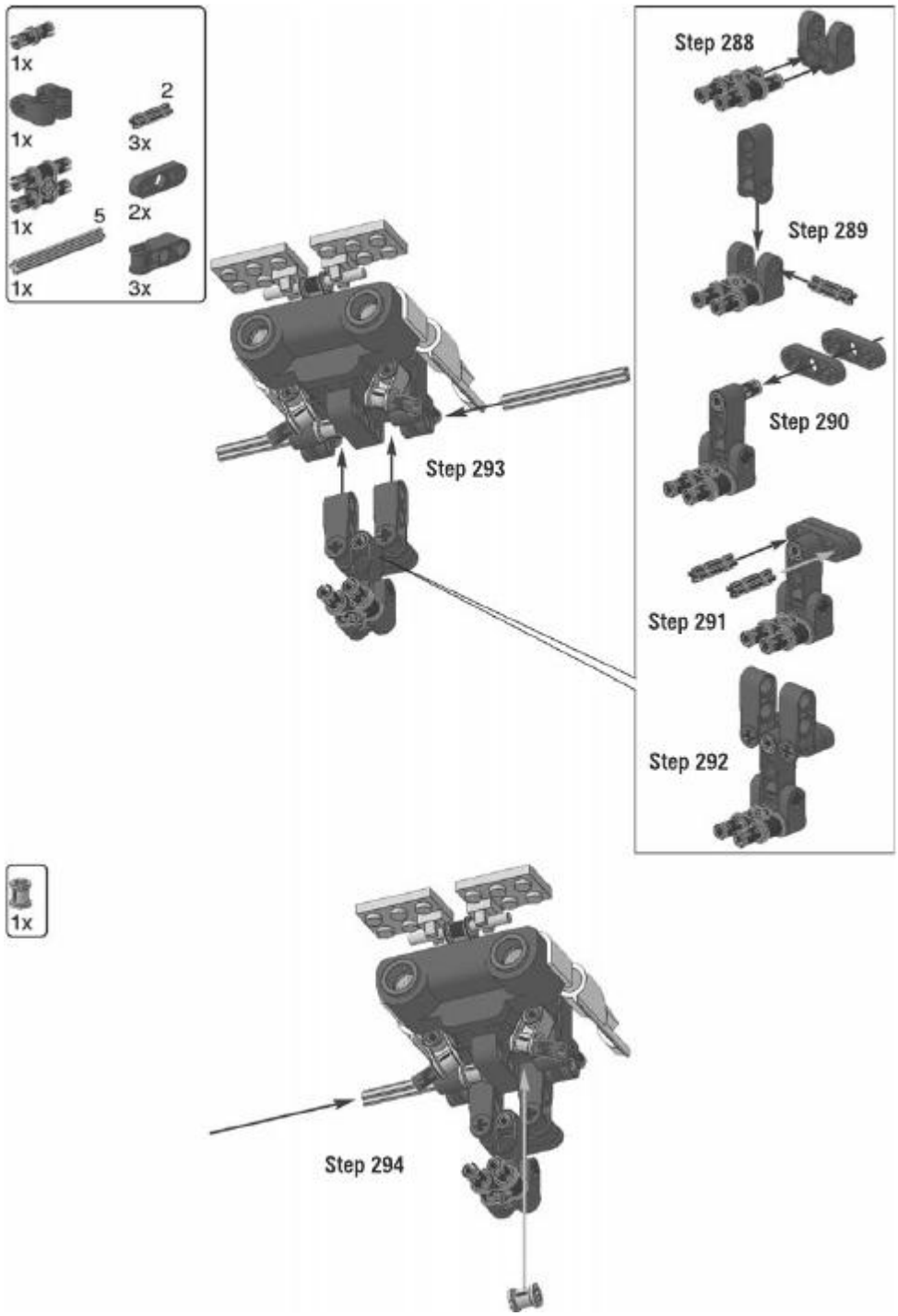

*Start building JohnNXT's head. Add fairing panels #5 and #6.*



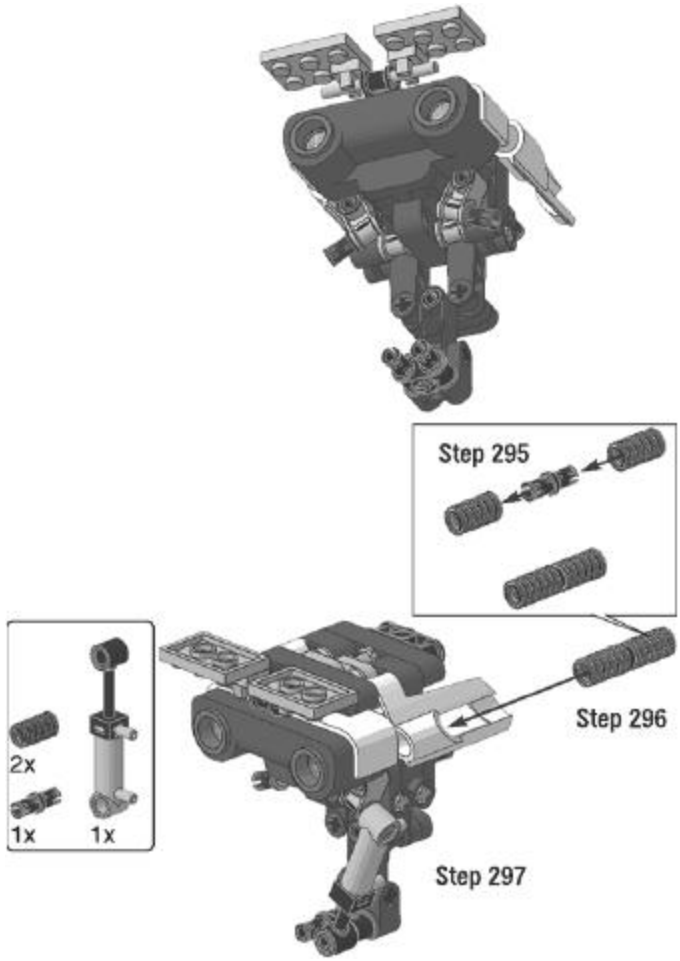

*Add a 4-long bar (like the minifigure's saber blades). Build the eyelids.*



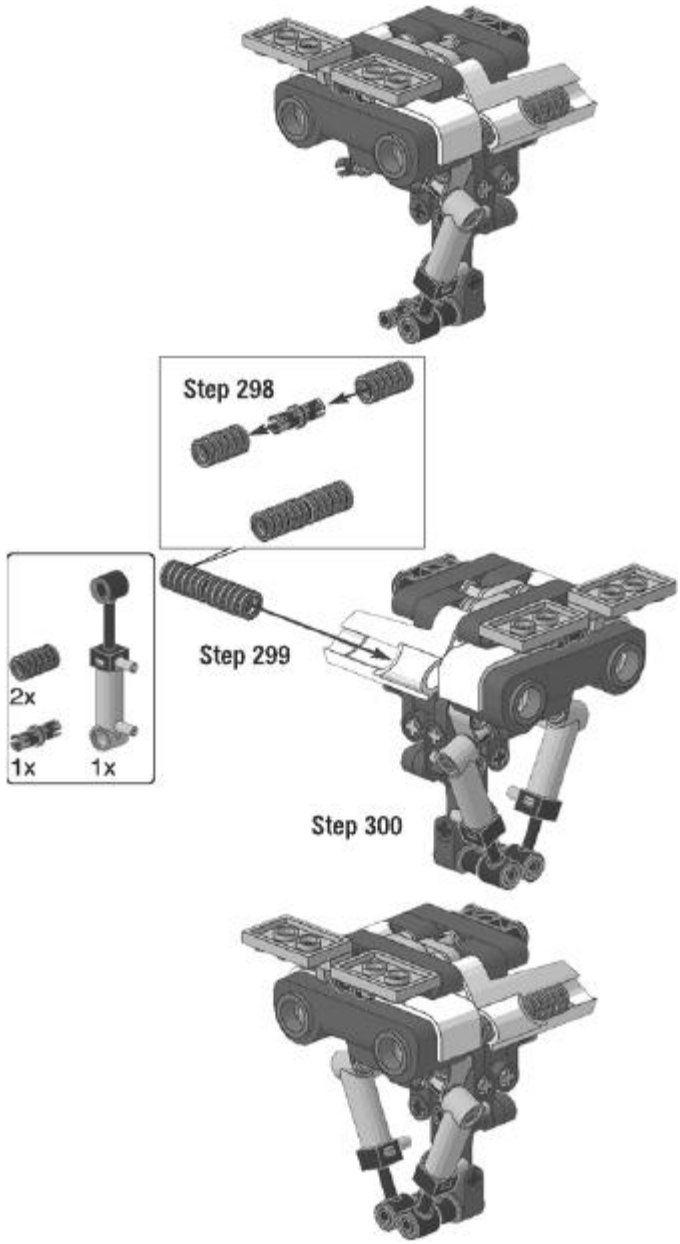

*Continue building the head, adding the Ultrasonic Sensor. This one is similar to J5's own head!*



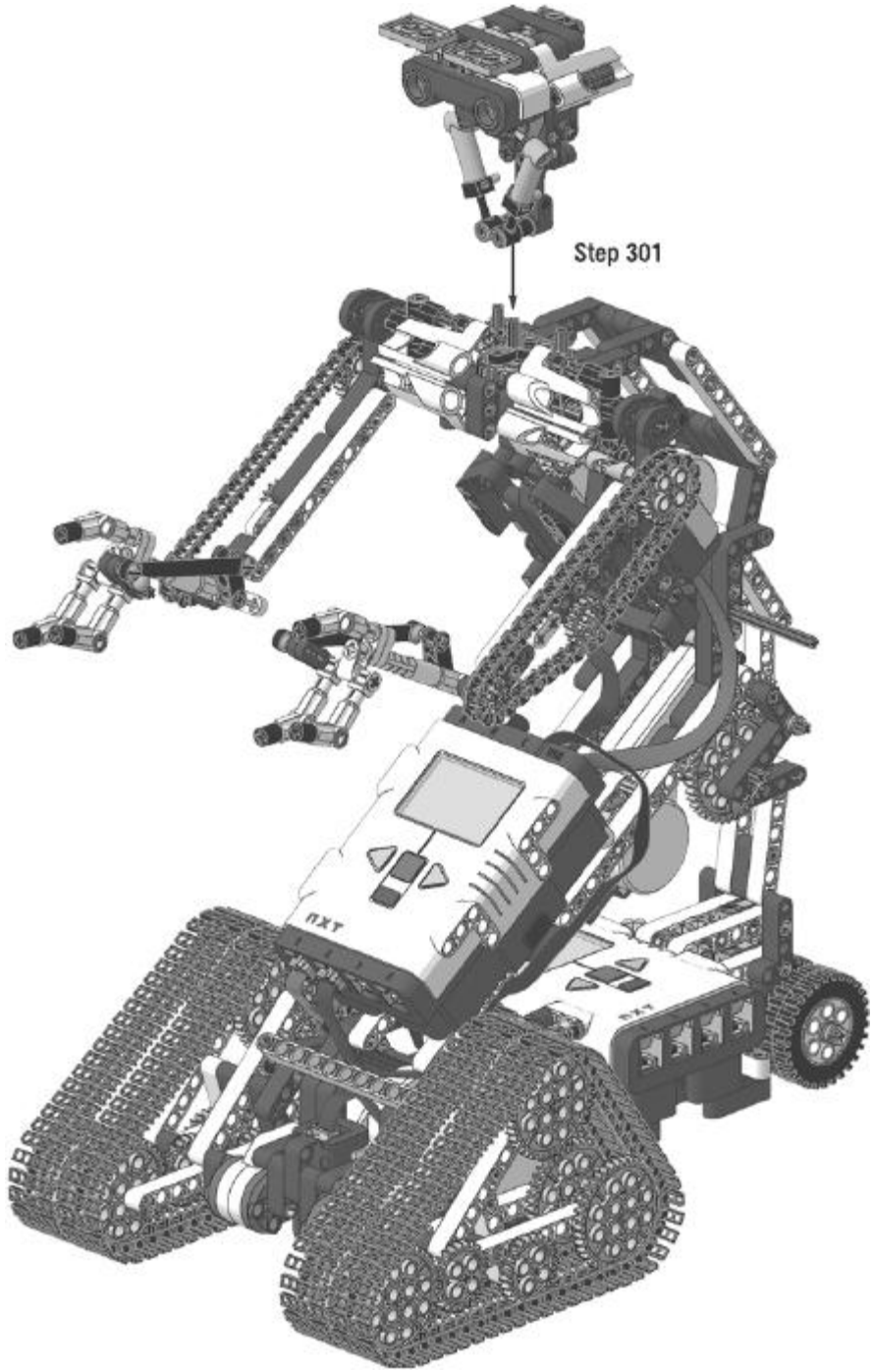

*Build the neck. Add a bush and push the axle to lock the sensor to the rest of the head.*



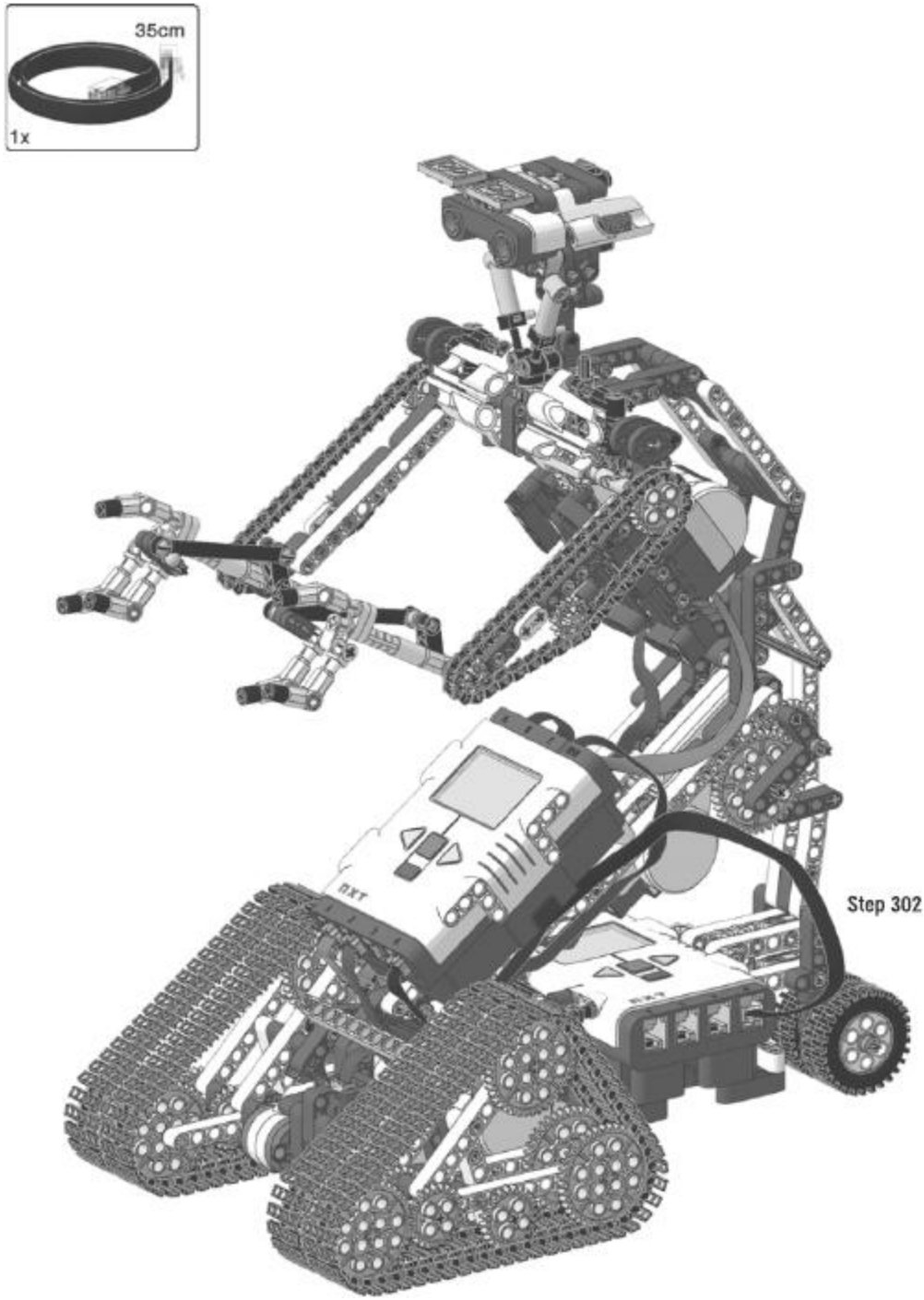

*Add a piston and the decorative hoses.*



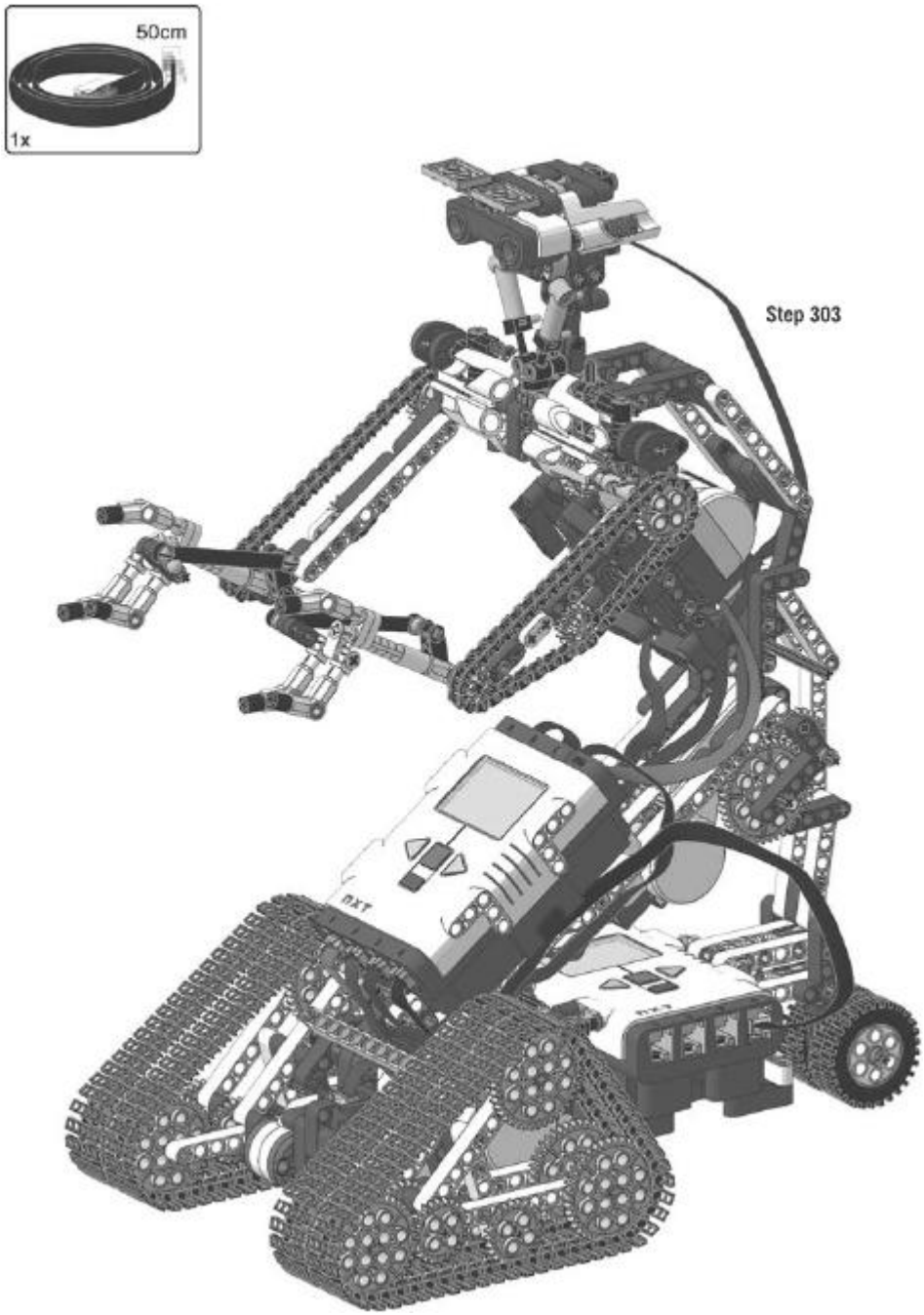

*Complete the head, adding the other piston and hoses.*



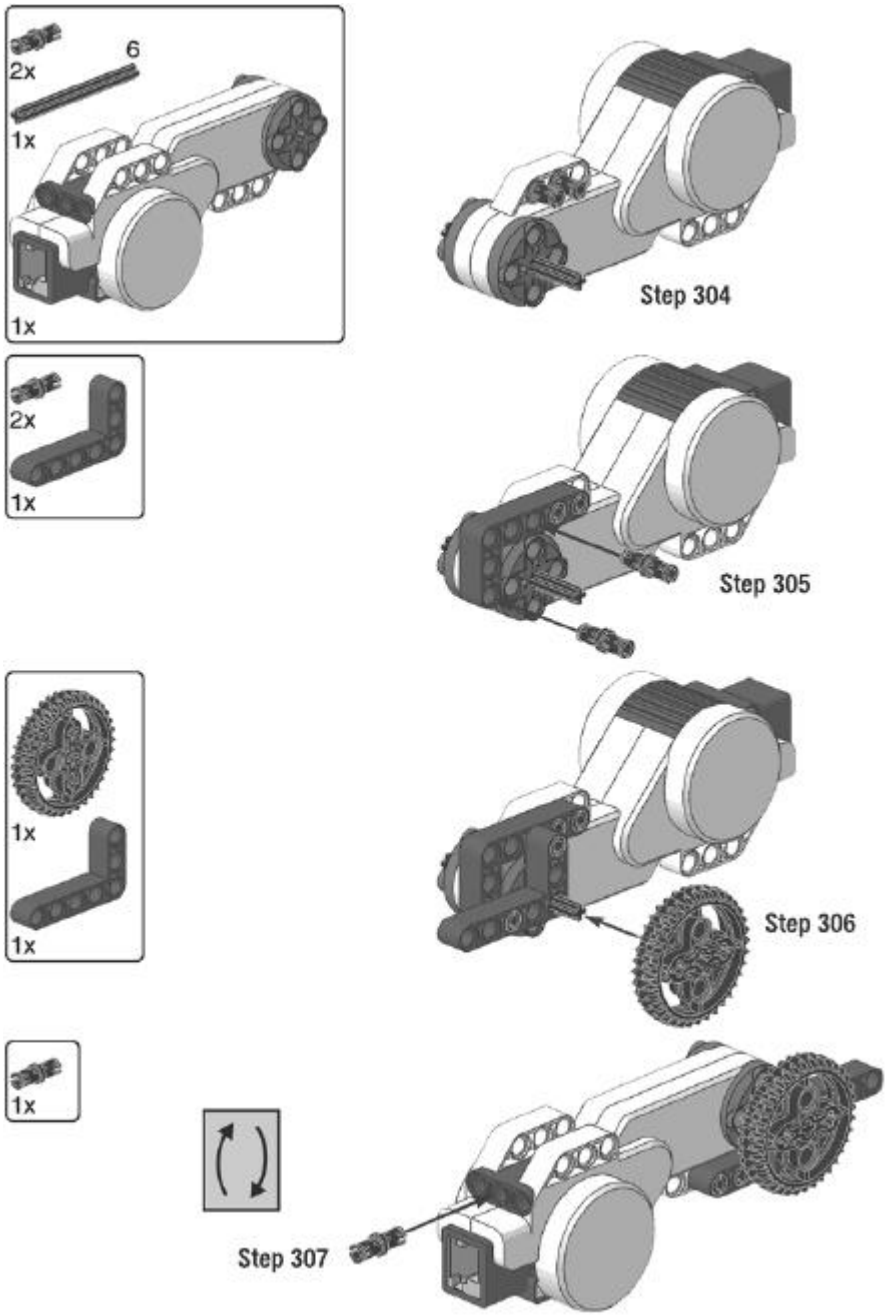

*Attach the head to the axle.*



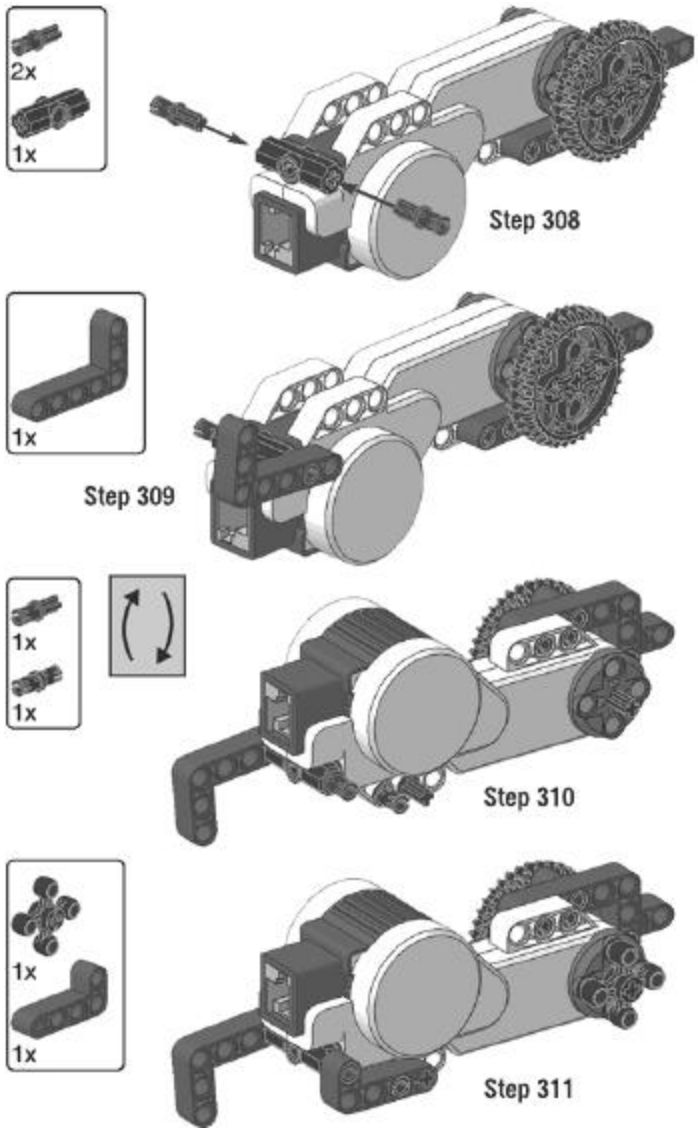

*Attach a 35cm (14 inch) cable between the respective ports 4 of both NXTs. This cable is used for high-speed serial communication.*



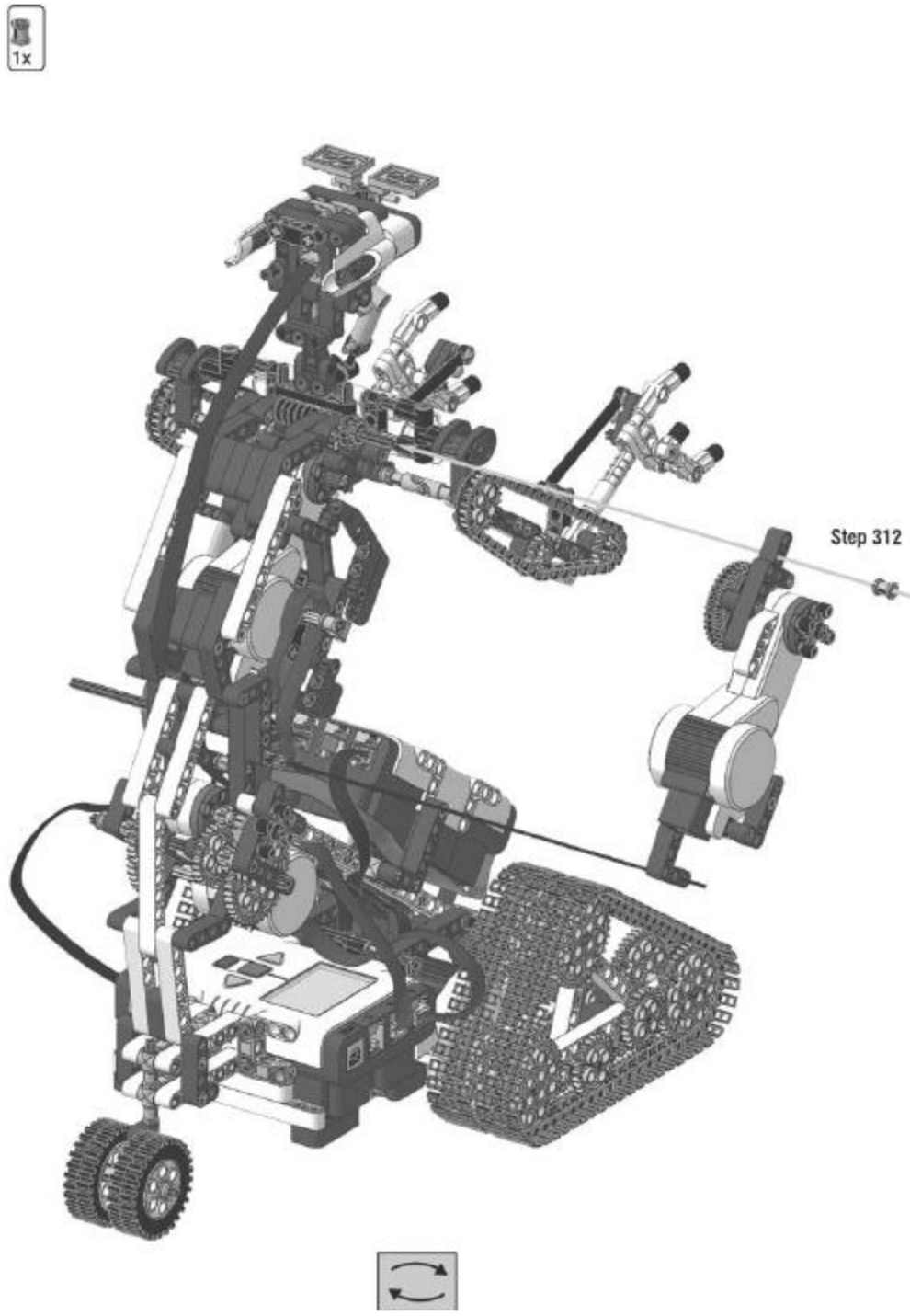

*Attach the Ultrasonic Sensor to master NXT input port 3 using a 50cm (20 inch) cable. Pass the cable where shown under the white 7-long beam to which the NXT is attached. Try to keep the cable central. Pull it to get the maximum length sticking out the back, so that it is free to move together with the head.*



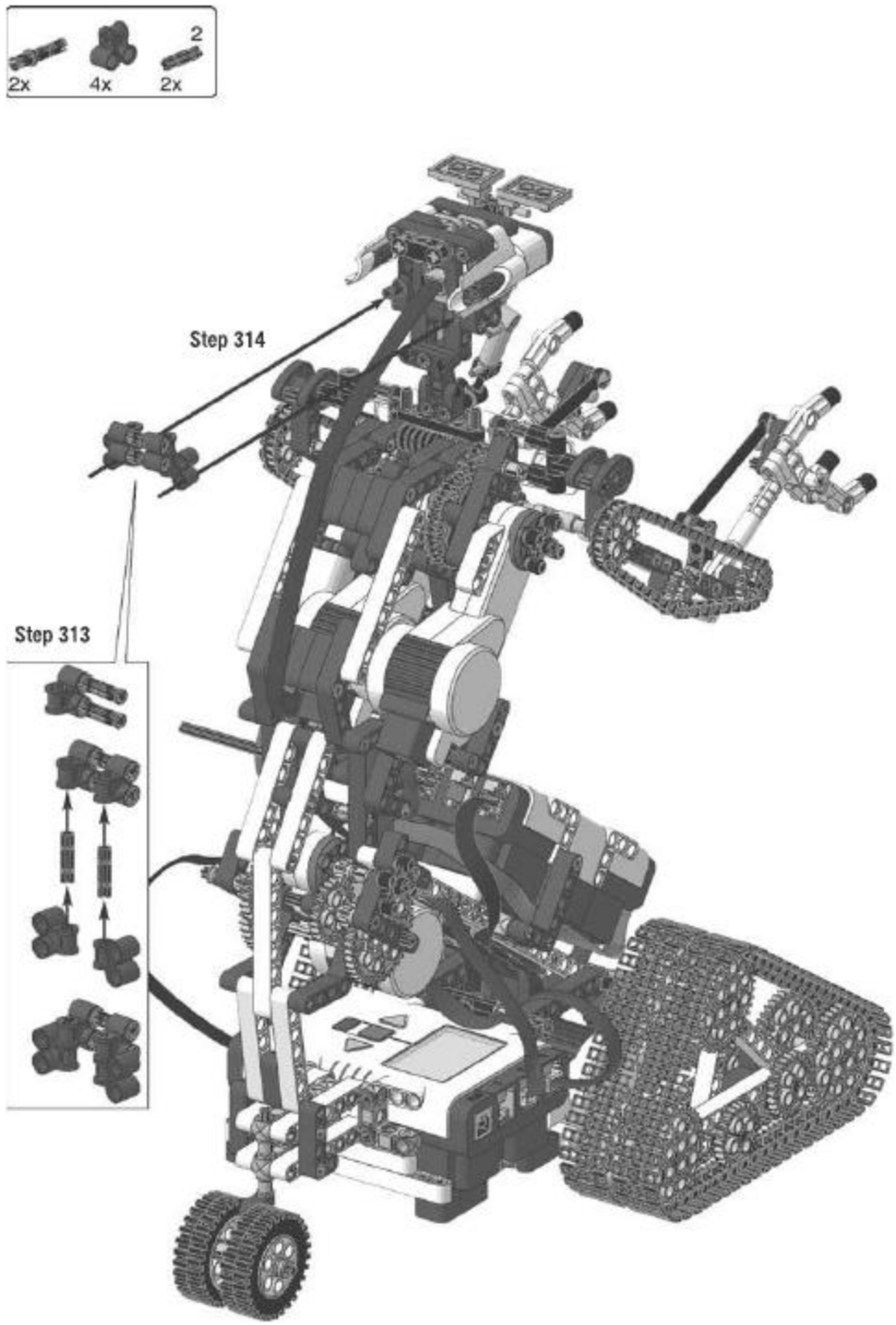

*Start building the head's motor assembly.*



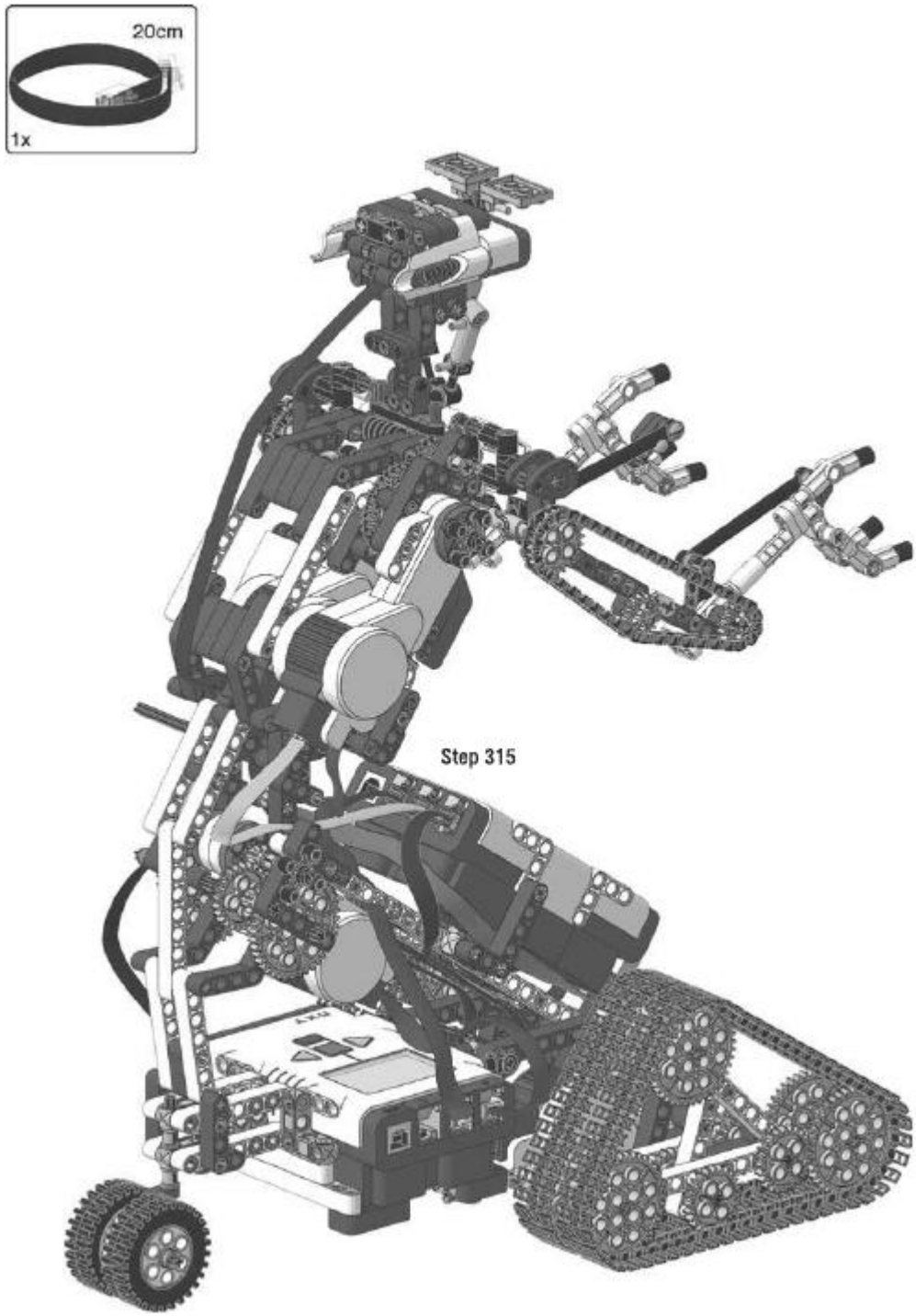

*Complete the head's motor assembly.*



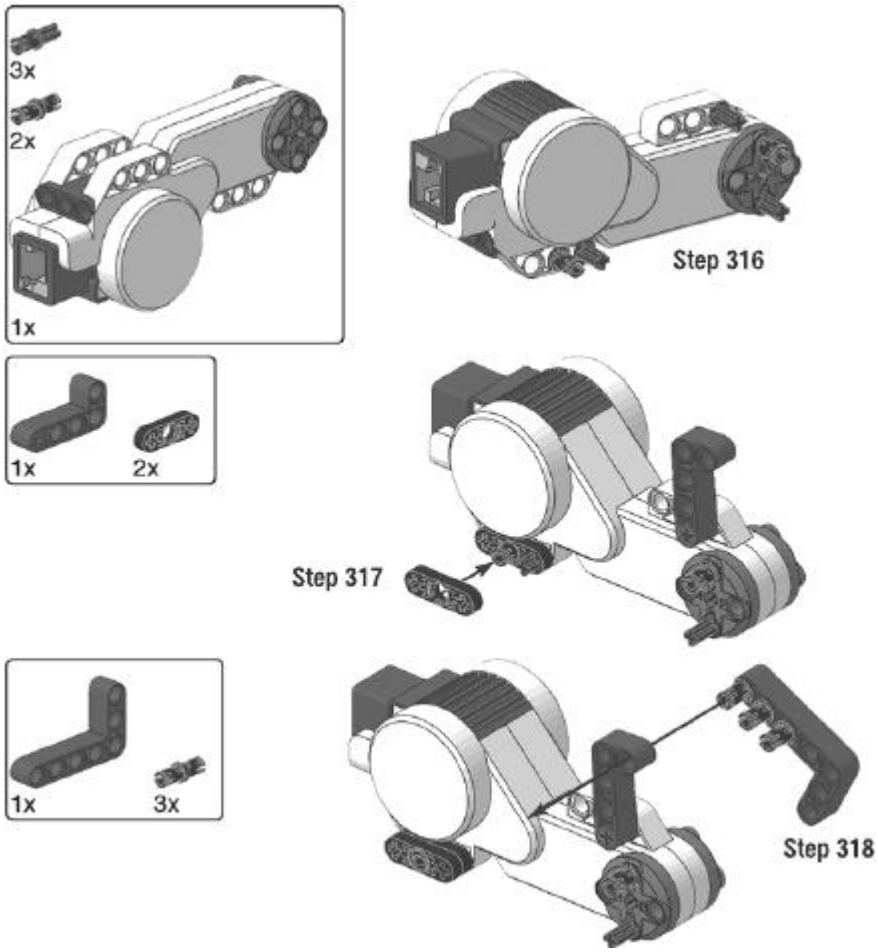

*Attach this motor to the rest of the robot, blocking it in place with a bush.*



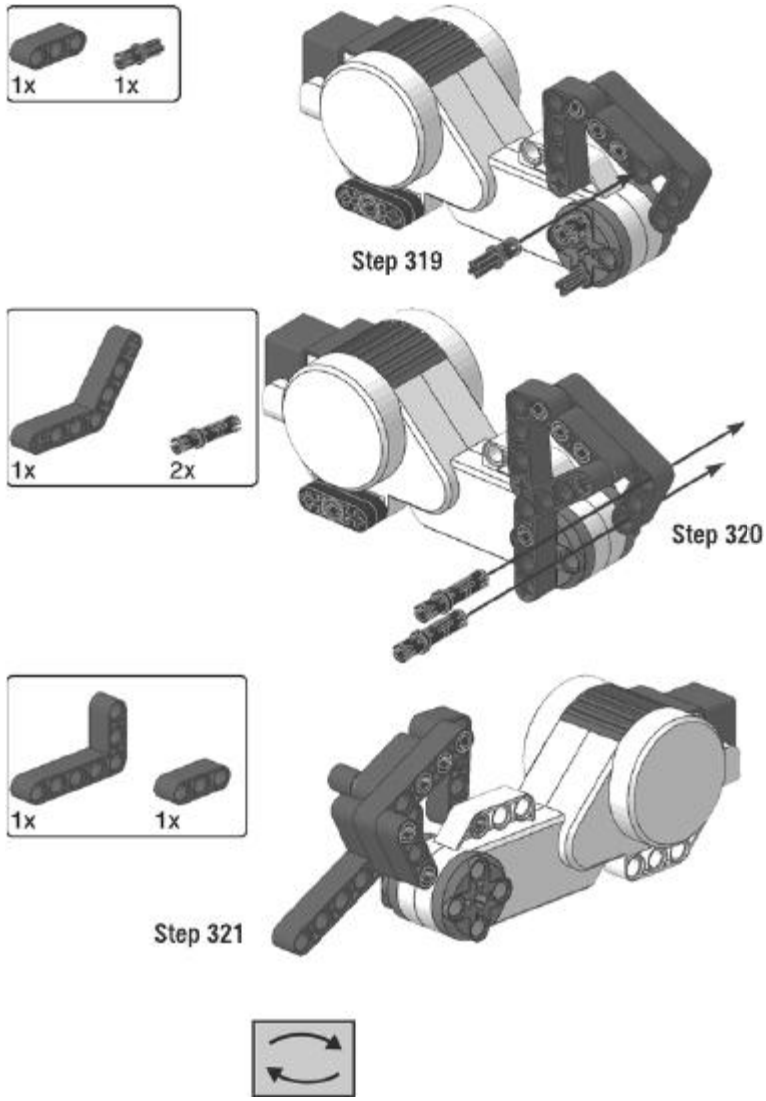

*Build the back of the head.*



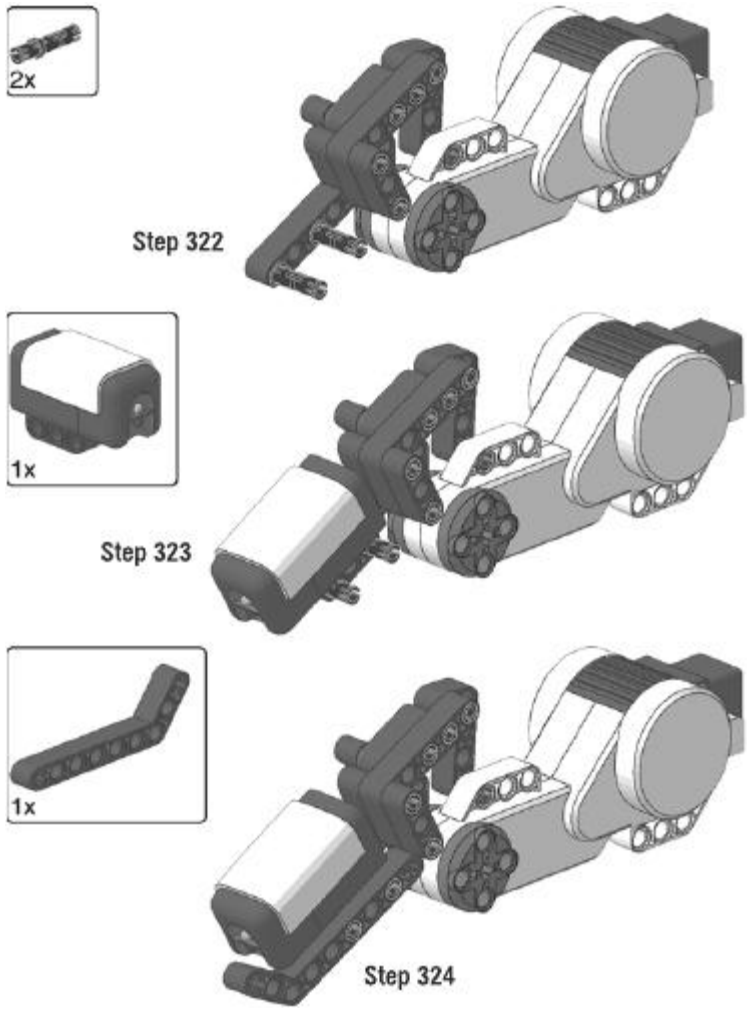

*Attach the head's motor to master NXT port B using a 20cm (8 inch) cable.*



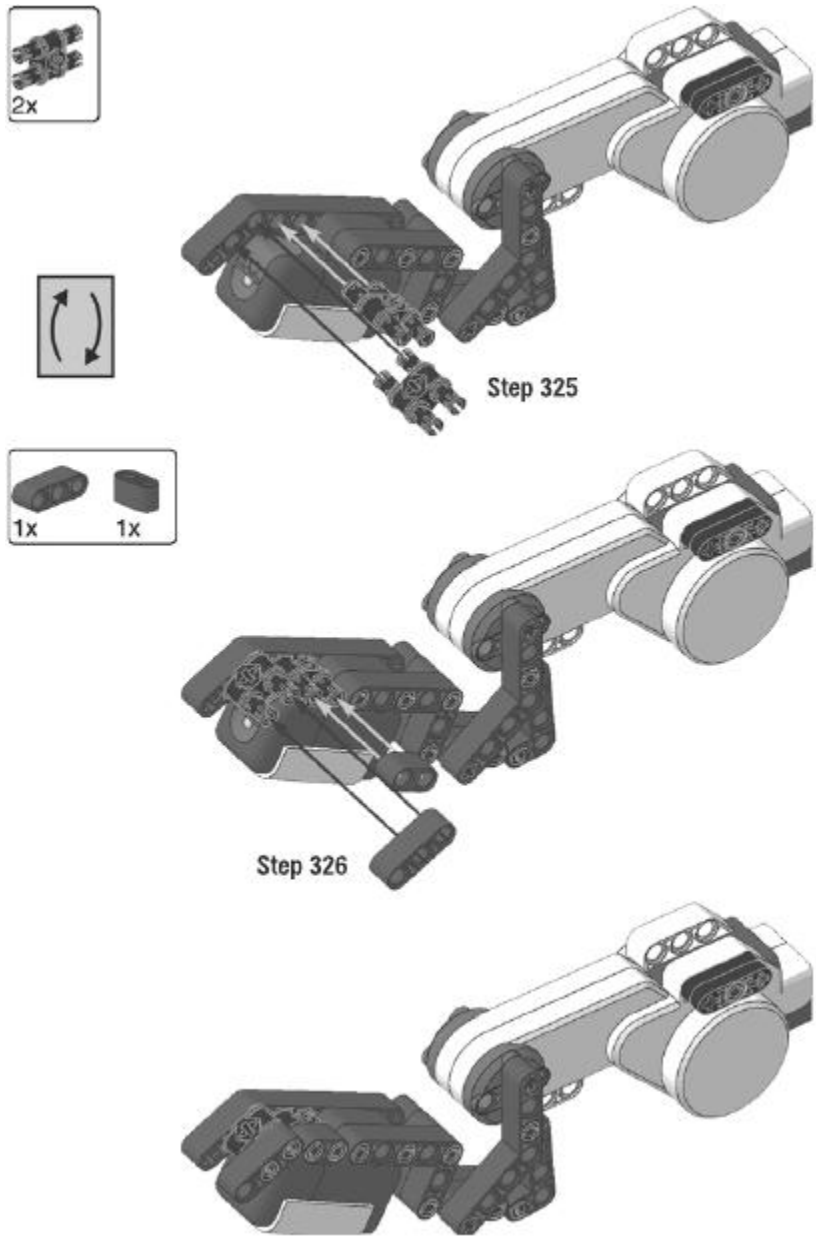

*Start building the laser's assembly.*



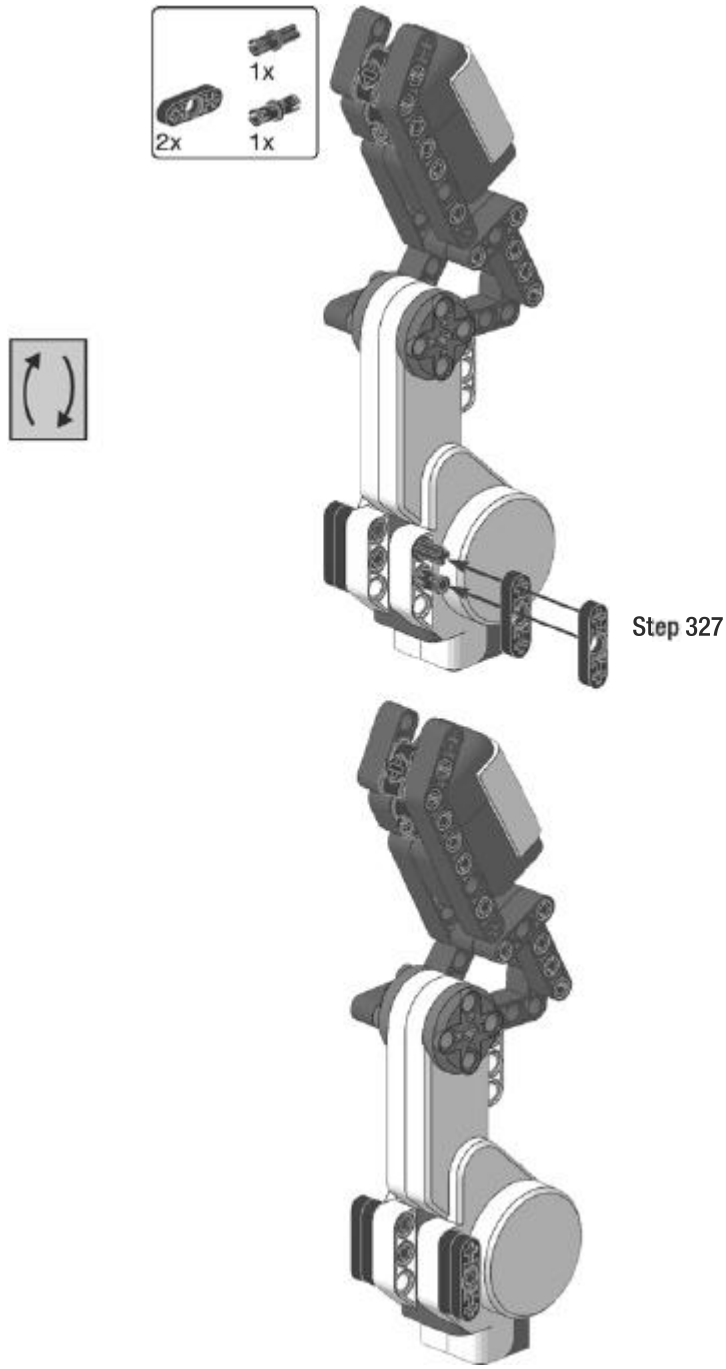

*The levers' system is complete.*



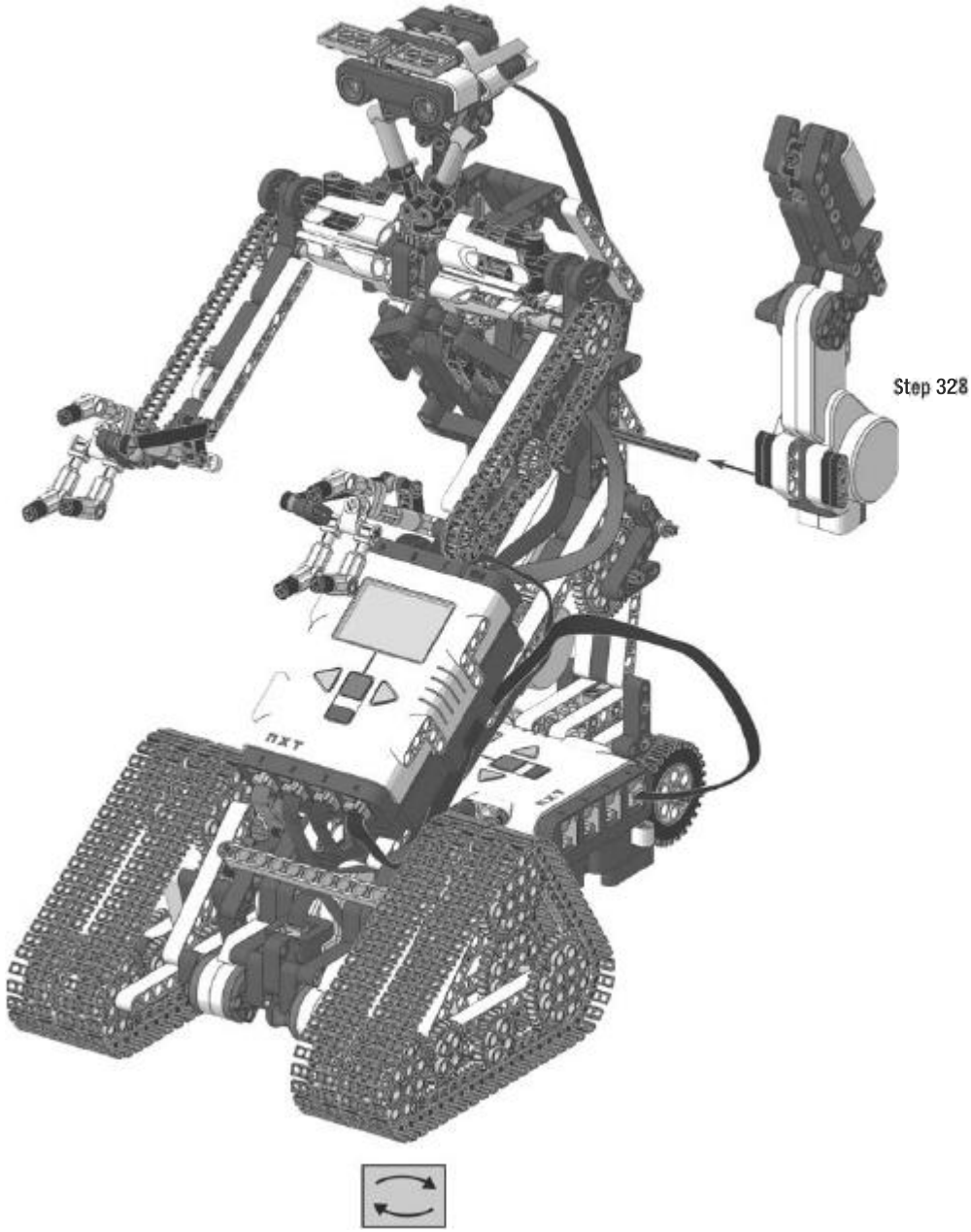

*Add the Light Sensor that simulates the laser blinking.*



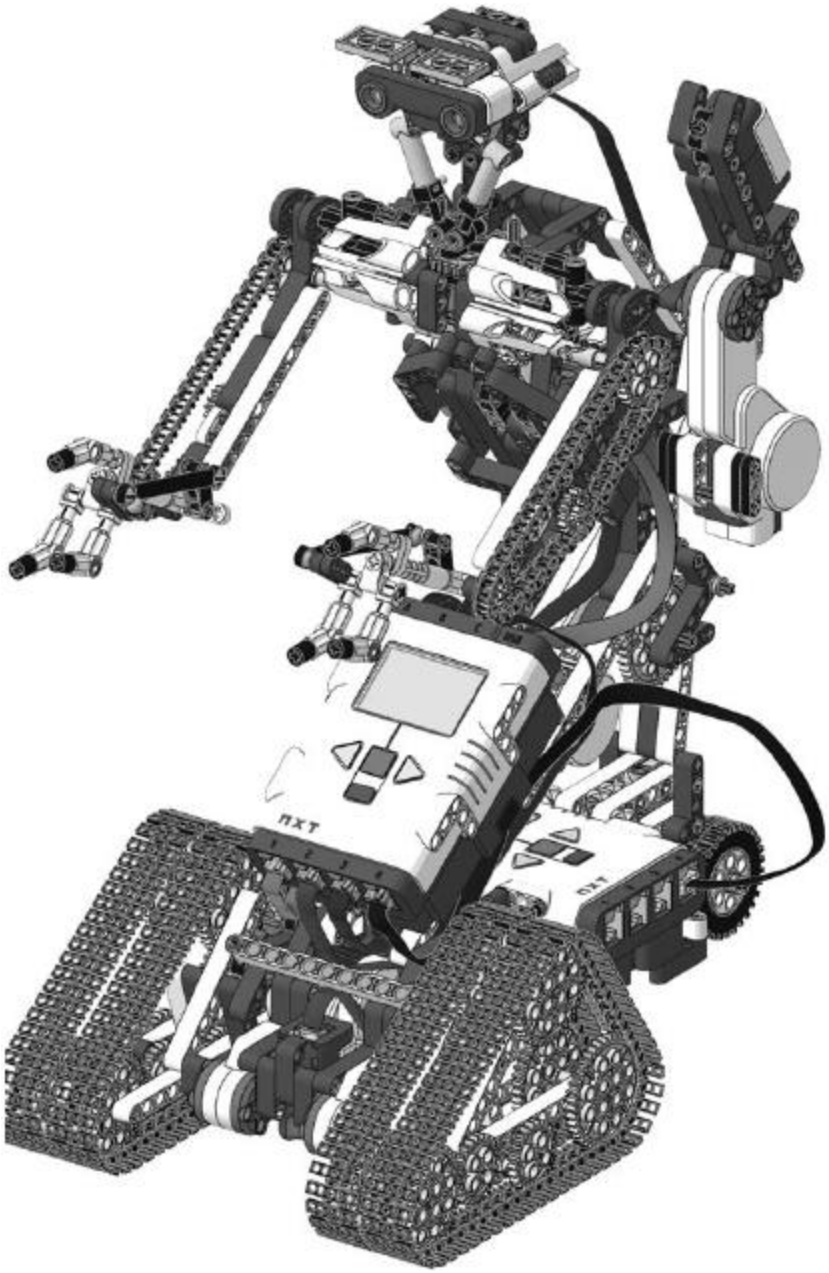

*Turn the model and complete the laser weapon.*



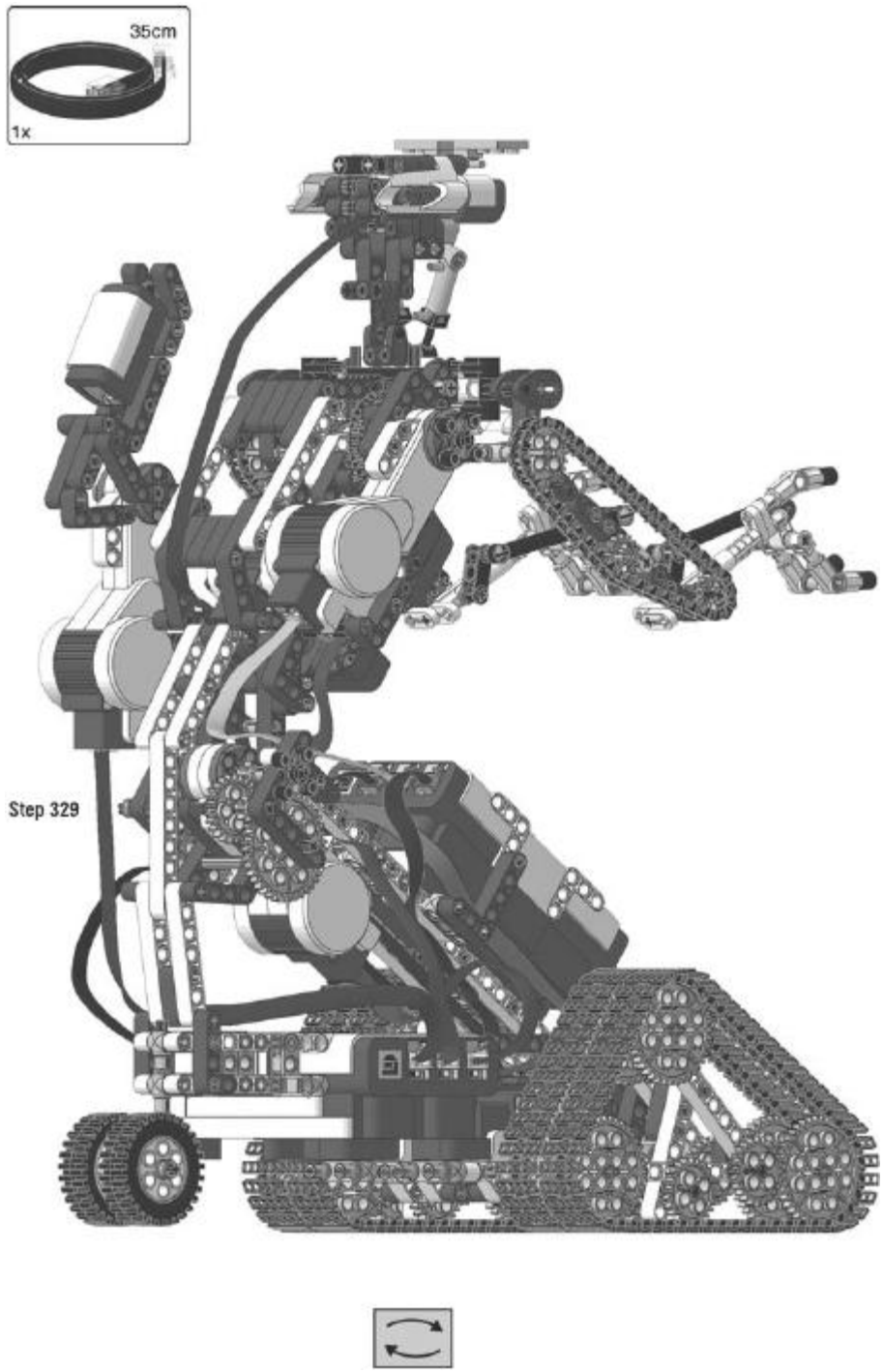

*Turn the laser and complete the assembly.*



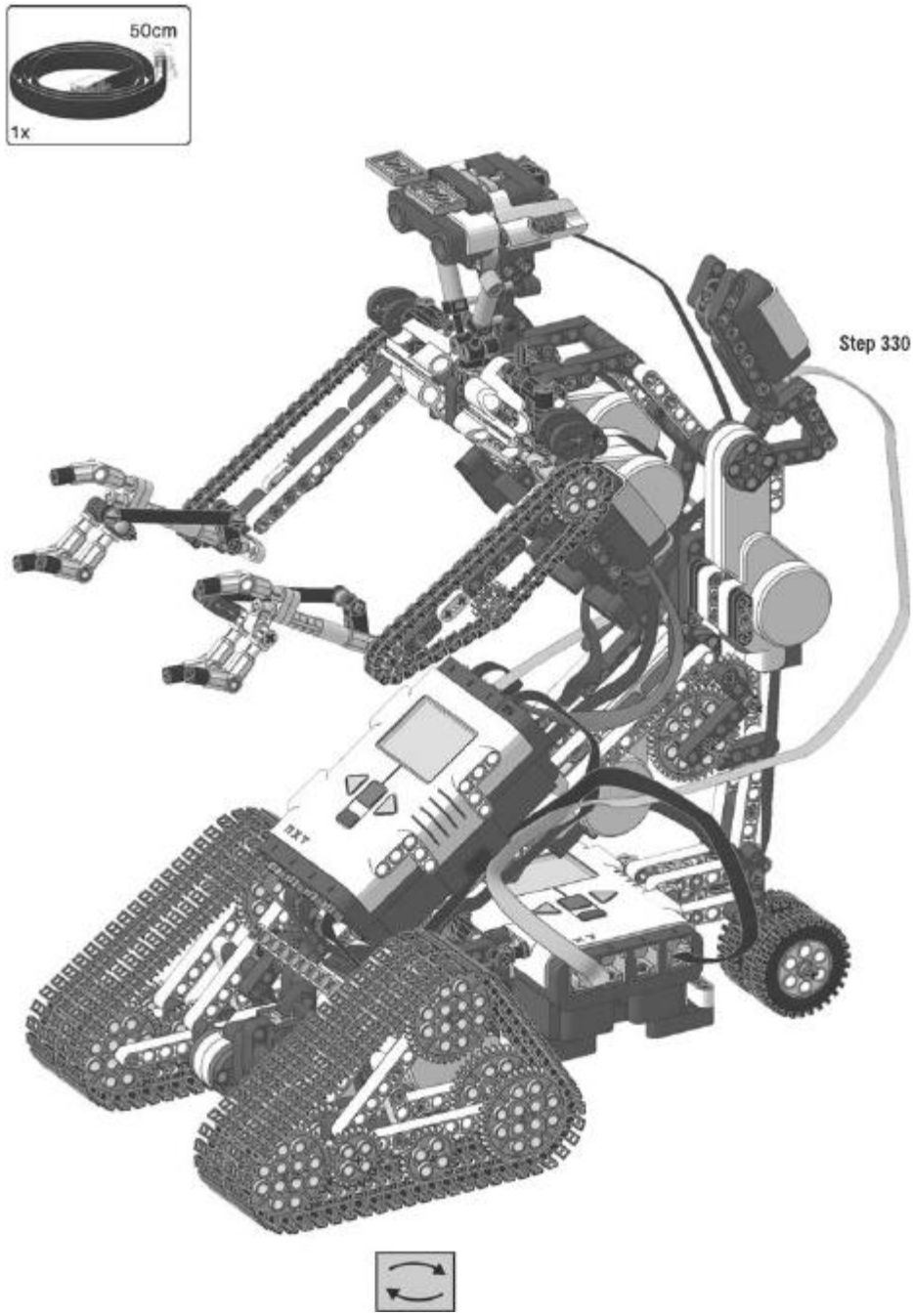

*Attach the laser in place, on the 12-long axle. The laser moves with the torso.*



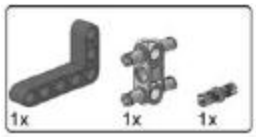

*Here's how the model looks once the laser is attached.*

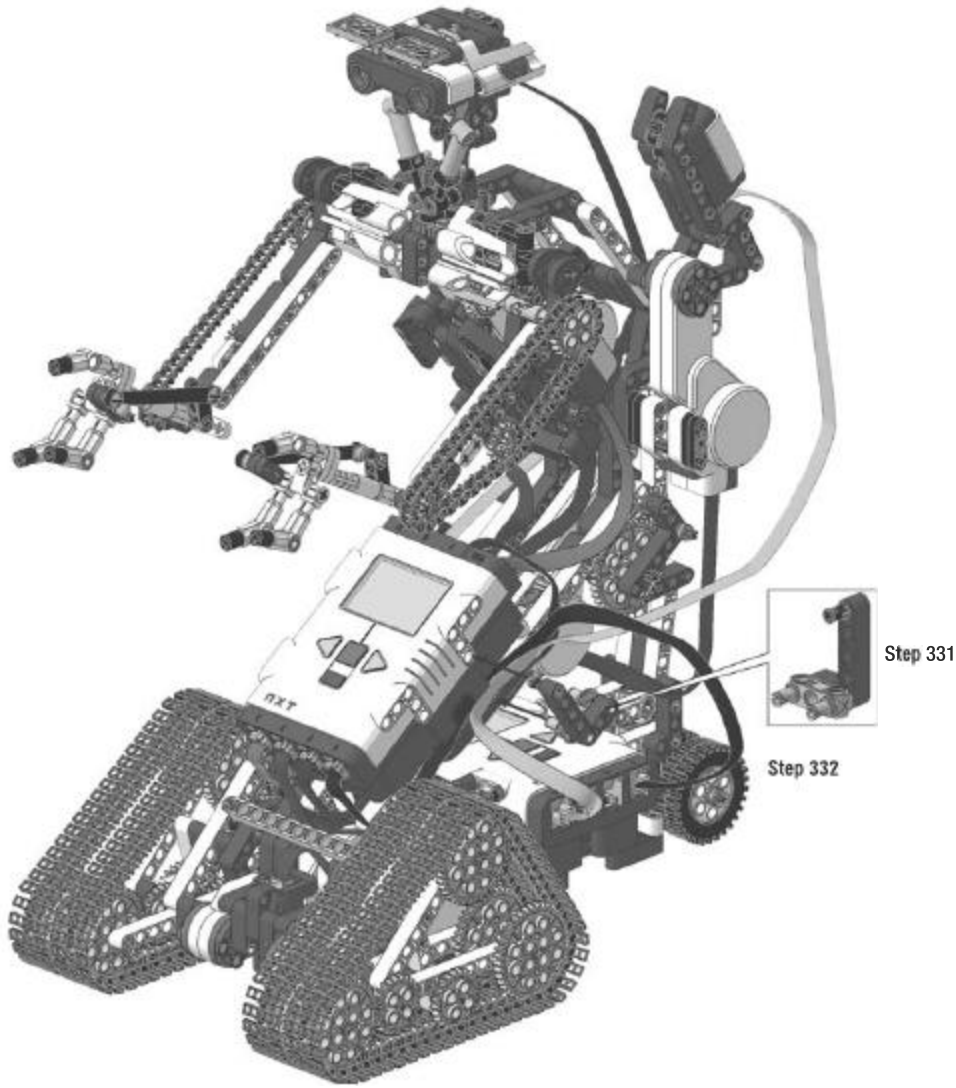



*Turn the model and attach the laser's motor to slave NXT port C using a 35cm (14 inch) cable.*

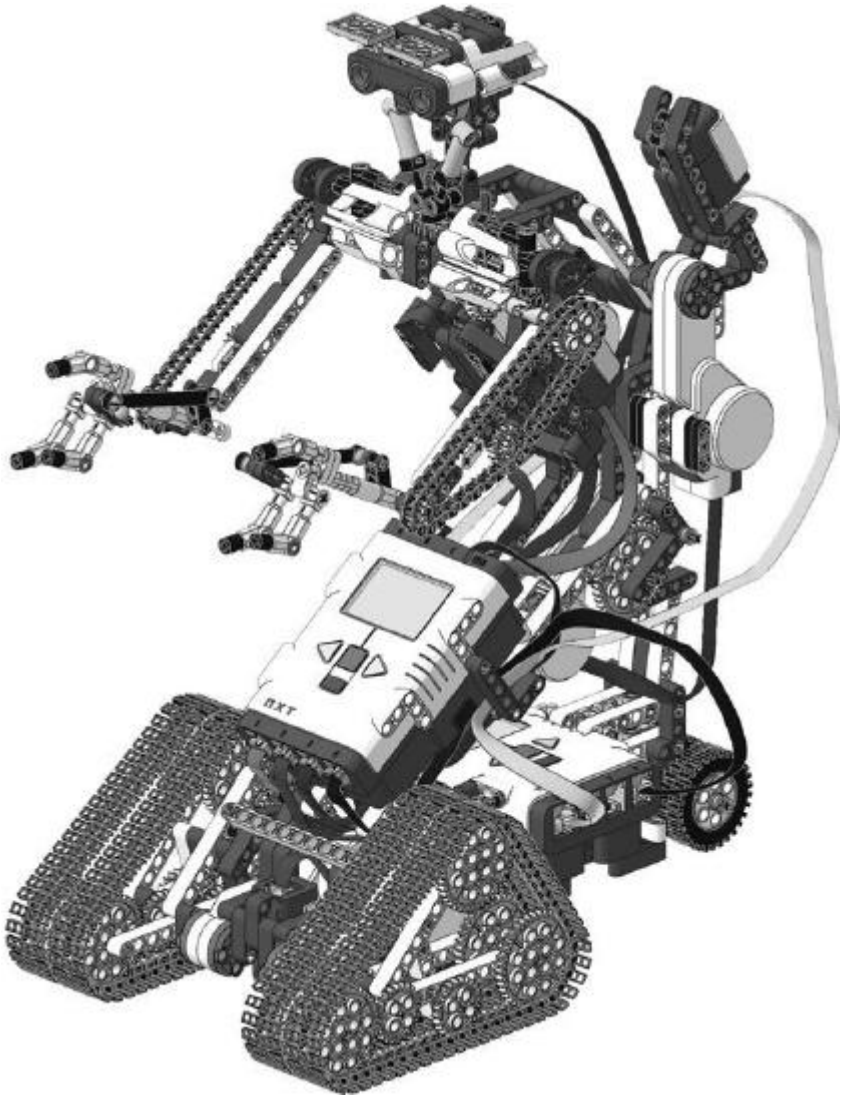



*Turn the model again and attach the Light Sensor of the laser to slave NXT port 1 using a 50cm (20 inch) cable.*




*Build the assembly that blocks the NXT in place. This block must hold the cable for the laser sensor and for the high-speed communication.*


NXT

The protagonist of this chapter is a compact wheeled robot equipped with a frontal double-sensor scanner and a grabbing arm. Its design will show you how a single motor can accomplish more than a single function. In Figure 7-1, you can see the Mine Sweeper with the abyss-avoidance sensor mounted. In the frontal scanner a Light Sensor is used to detect dark LEGO bricks on white ground. In addition, the Ultrasonic Sensor is mounted downwards as an abyss detector, so if the robot is going to work on upland planes (tables), it won't fall down.

**Figure 7-1.** *The Mine Sweeper is equipped with a ravine-avoidance sensor. The mines shown here are built using two black 2 - 4 LEGO bricks.*

The robot you are going to build could have been given many other different names: garbage collector, floor sweeper, object collecting contraption, and so on. If "Mine Sweeper" recalls the sadness of some human invention, call the robot whatever else you want, and I won't take offense.

The robot can collect only small objects with a regular shape, distinguished by their dark color on a light ground; it is not able to collect objects of any color and any shape. Because it is specialized for collecting only a precise kind of object, it came to mind to call it a mine sweeper. The real bomb-disposal robots use particular metal detectors to find mines on the ground and collect them using skilled robotic arms; our LEGO Mine Sweeper has a frontal sensor and a grabber arm, and this gave me the idea for the name.

# Getting More Actions from a Single Motor

The arm mechanism is designed to grab objects, lift them, and store them into the robot's internal hold, performing all these actions with only one motor. This is particularly interesting, because usually one motor corresponds to one degree of freedom (DOF).

### UNDERACTUATION

We talk about underactuation, in robotics, when dealing with mechanical devices that have a lower number of motors than degrees of freedom.

The DOF of a mechanical system is defined as the number of independent parameters needed to characterize its state. In other words, if a motor can drive only a mechanism, a robot must have a motor for each action it can do. For example, in a wheeled robot a motor controls each wheel; in a steering vehicle the motor that drives the wheels cannot also steer; in the official NXT robotic arm, the motor used to grab the balls does not move the arm up and down. However, having an actuator for every DOF can become a problem! In a robotic grasping hand, using an actuator for every phalanx leads to a huge number of actuators: the device's versatility would increase, but its cost, complexity, and weight would become unmanageable.

In the Mine Sweeper's case, its arm can both grab and lift objects. The grabbing is an underactuated mechanism, because there is not a specific motor to close the fingers; the actuation is done by the same motor that lifts the whole arm. This solution saves space—where to fit another motor?, reduces cost—simply, we do not have a fourth motor, and lowers the overall weight. For these reasons, underactuated devices can be more efficient, simpler, and more reliable than their fully actuated alternatives. Of course, for a motor to perform more actions, you must devise a clever mechanism.

LEGO itself produced some official models that use an underactuated mechanism to grab an object first, and then lift it. Among many others, some examples are the yellow submarine 8250/8299 (released in 1997), the barcode truck 8479 (1997), and the alternative model of bulldozer 8275 (2007). How can a single actuator decide in which order to perform such different tasks—to grab and then to lift?

The submarine has a pneumatic hand that grabs and then lifts a barrel. Talking informally, in the submarine detail shown in Figure 7-2, the grasping is a lighter operation than the lifting. When the pneumatic piston shortens, it runs into the mechanical opponent force of the spring (the LEGO shock absorber) and then closes the grabber. Once the grabber is fully closed, the movement is blocked so the piston can't help but raise the arm. When the piston lengthens, the arm is lowered first and then the grabber is opened. Also at this time, the spring that forces the arm down does the lowering. Something similar happens both in the barcode truck and in the bulldozer alternative model, but this time, the opposing force is gravity.

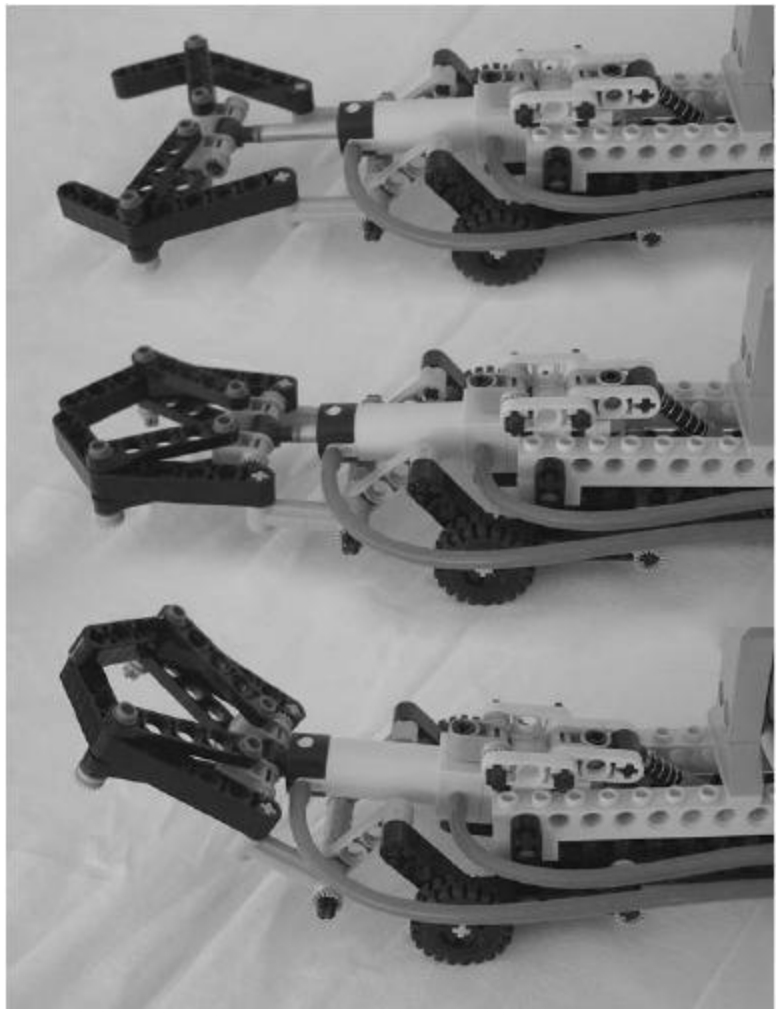


**Figure 7-2.** *The underactuated grabber mechanism of the LEGO submarine 8299. The piston first closes the grabber and then raises it.*

Now you know the state-of-the-art in underactuated LEGO grabbers. Among the many unofficial LEGO robots featuring underactuated grabbers are Ben Williamson's FetchBot (1998), Jonathan Knudsen's Minerva (1999), and Philippe Hurbain's Barrel Collector Robot (2003), all based on the RCX system.

The Mine Sweeper becomes part of that unofficial LEGO robots rank—it uses the same principle as the barcode truck. The arm grabs, lifts, and brings the mine up to the opening of the hold if the motor is turning forward. It then releases the mine into the hold and comes back down if the motor's turning direction is reversed. The easiest and most direct way to understand how this double action is achieved is to build the robot and observe it in action; see the photos in Figure 7-3.

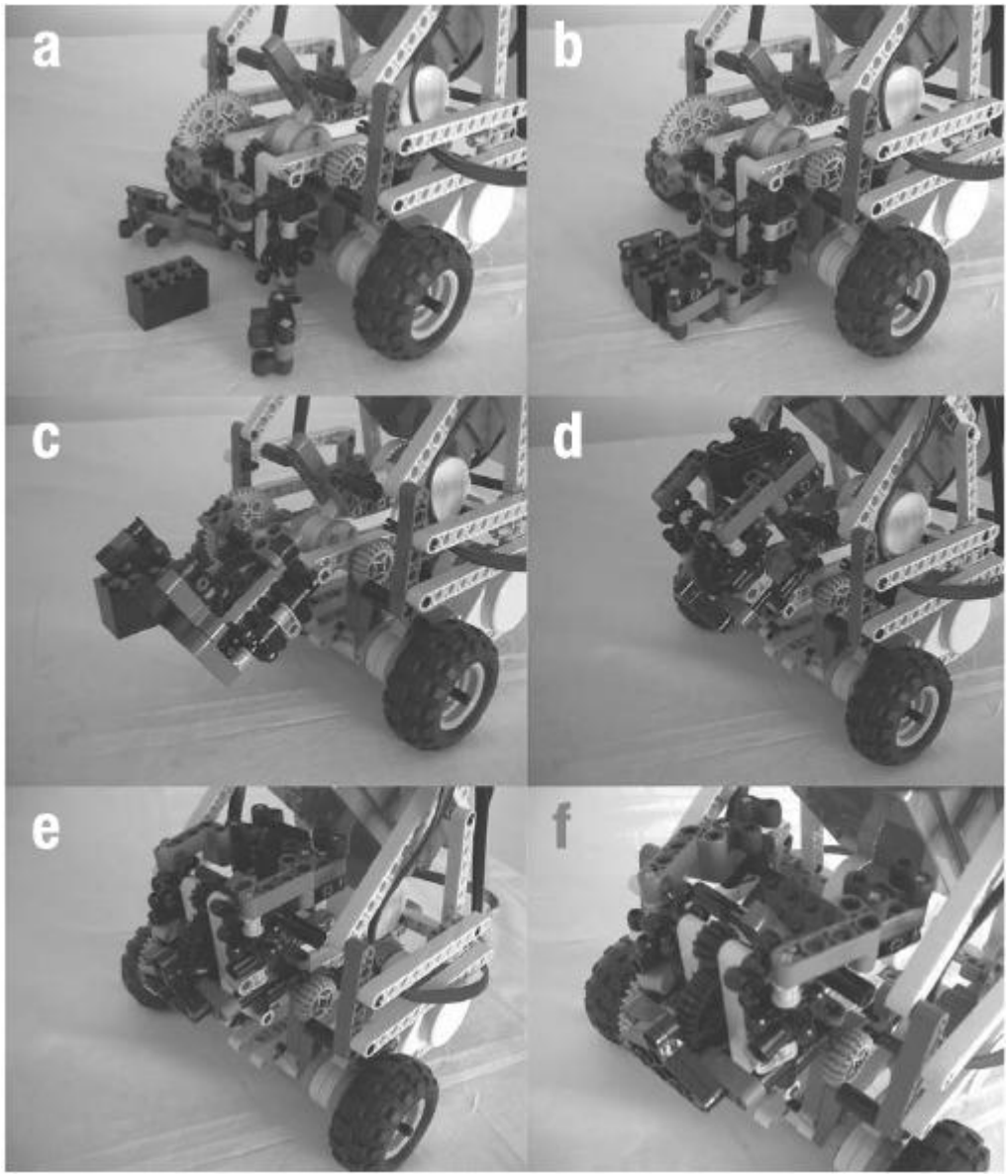



**Figure 7-3.** *The Mine Sweeper's grabbing sequence (the frontal scanner is removed)*

Now, take a look at Figure 7-4. Here you can see the arm mechanism extracted from the robot context. At the base of the actions' switching stands an opposing force. Here the force is gravity, while in the submarine, the force was produced by the compressed spring.

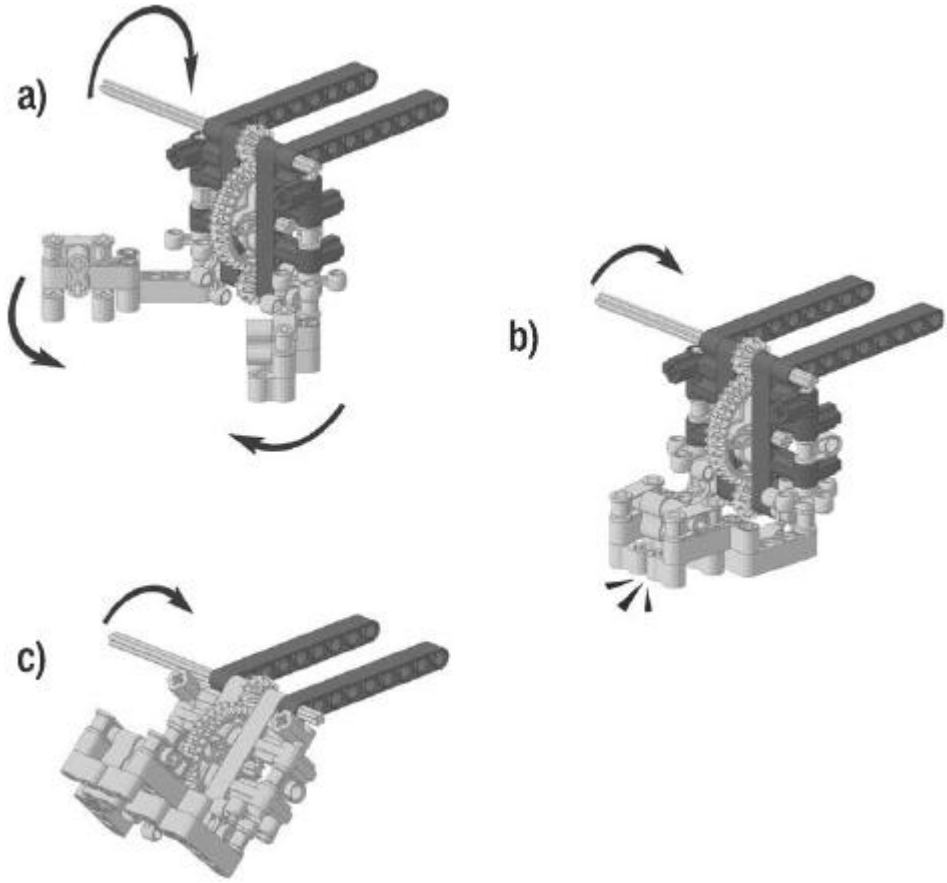



**Figure 7-4.** *The Mine Sweeper's grabber arm is extracted from the robot's context. The sequence of actions is determined by the force of gravity and by the limited run of the fingers.*

In Figure 7-4*a*, the arm is hanging vertically and the motor starts to move in the direction specified by the arrow; the light-colored parts are the ones that can move freely. The axle that transmits the driving torque is not integral with the 7-long white beams that form the arm frame; the axle rotates freely in the beam holes, and the geartrain brings its movement to the fingers. The arm is prevented from lifting by the force of gravity, which has no influence on the fingers' movement. So, the driving torque flows towards the fingers, because they are completely free to move. In Figure 7-4*b*, the fingers are completely closed, and the driving torque is thus redirected to raise the arm. In Figure 7-4*c*, the axle has become integral with the arm frame, because the geartrain is blocked by the closed fingers. The axle can't help but lift the whole structure. This sequence *a*–*b*–*c* in Figure 7-4 is matched with the photos *a*–*b*–*c* of Figure 7-3. If the sequence ends here, the robot has collected an object: reversing the motor direction, the object will be lowered and released. To store the mines into the hold, the sequence must be completed, as shown in Figure 7-3, photos *d*, *e*, and *f*.

The dark objects collected are stored into a space found in the depths of the robot. Considering a standard "mine," one built with two 2 - 4 black bricks, the robot can collect more than ten of them. Not bad at all for our purposes!

## The Double Scanner

Now, you know all about how the robot collects and stores the objects. But, how does it find them? The easiest way is to use a Light Sensor to detect dark objects on lighter ground by measuring the amount of light reflected by the objects. The Light Sensor is equipped with a red Light Emitting Diode (LED) that illuminates objects. The Light Sensor also has a detector (a phototransistor) that can measure the light reflected by the surface of the objects: the lighter the color of the object, the higher the reading returned by the Light Sensor, expressed in percent.

Using a third-party color sensor, you can expand the robot's abilities. For example, the robot could pick up bricks of a certain color without storing them (the short sequence of Figure 7-4) and accumulate them in a pile, as a moving brick sorter. On the other hand, it could work on uneven colored ground, overcoming the actual dark-and-light recognition restriction.

As anticipated at the beginning of this chapter, the frontal scanner includes two sensors: the Light Sensor, used to detect the mines, and the ultrasonic radar pointed downwards, to give the robot the ability to avoid the ravines. The robot interprets as a ravine an Ultrasonic Sensor reading of more than 35cm. It would not be a big deal if our expensive robot fell down from a table!

## Programming the Mine Sweeper

The program presented gives the robot the ability to clean the ground of the bricks, proceeding straight and scanning the ground as shown in Figure 7-5. The robot does not know about the already explored area. It simply goes straight, unless a ravine avoidance maneuver changes its direction. As said before, here we assume that the robot is on a light surface, searching for dark mines.

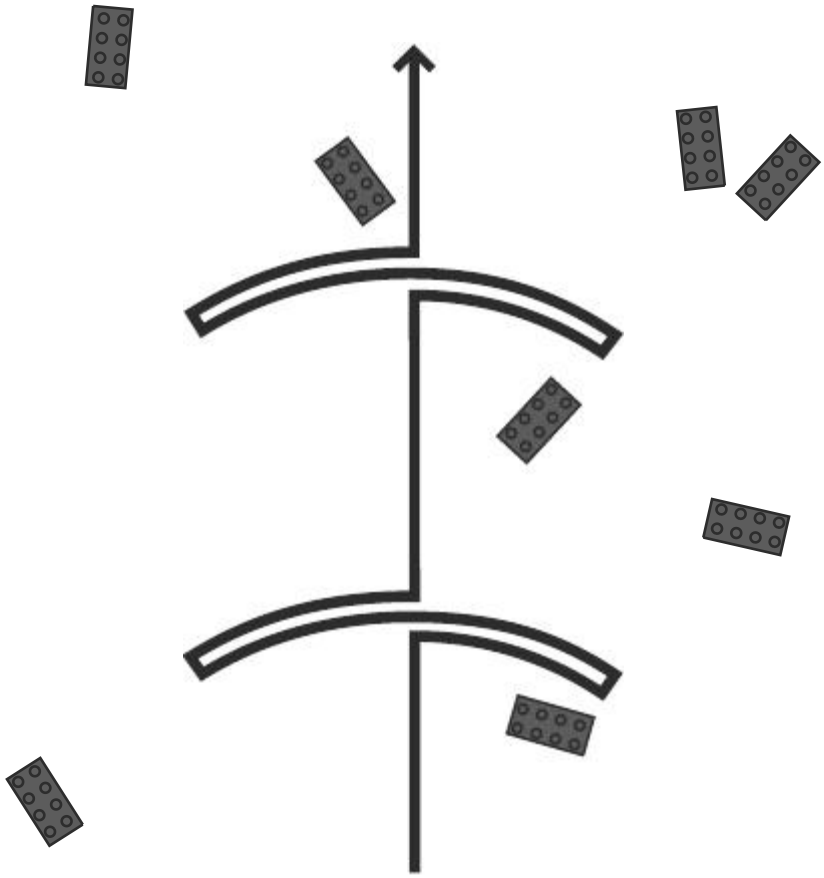


**Figure 7-5.** *The path taken to search for mines*

How do we obtain that particular searching path? Take a look at the flow chart in Figure 7-6. The overall working is given by the actions' sequence: search the mine, estimate the mine center, grab the mine. The ravine avoidance maneuver is activated only in case of emergency, during the search.

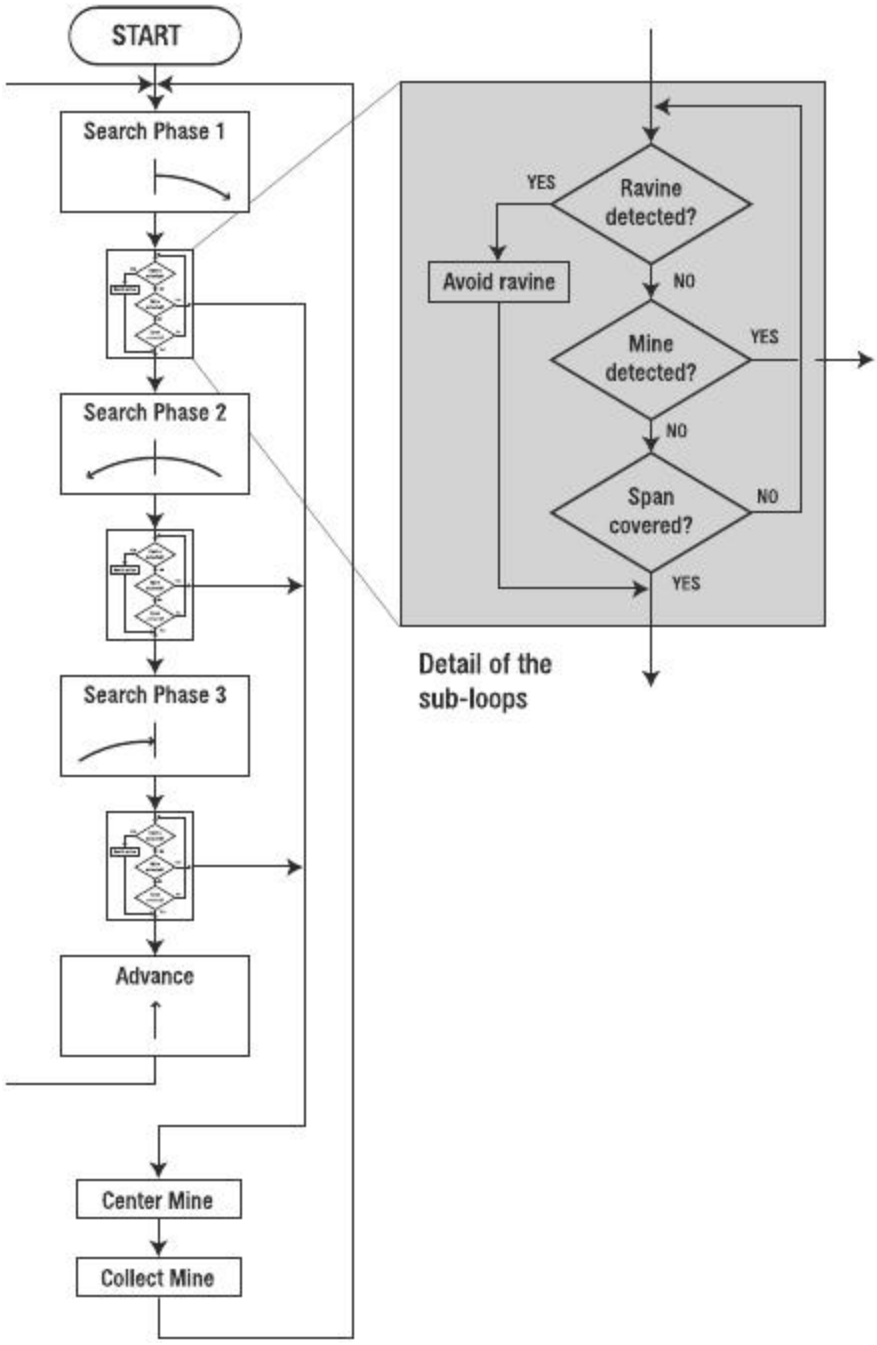



**Figure 7-6.** *The flow chart of the Mine Sweeper program*

The ground scanning is divided into three steps. First, the robot spins in place clockwise, until a ravine or a mine is detected, or a limited span has been covered (phase 1); then, the robot spins counterclockwise to reach the symmetric position, under the same conditions as earlier (phase 2); finally, it spins back to its original heading (phase 3) and advances a bit. The waiting for the three events that can stop the spinning (ravine detection, mine detection, and angle limit) is represented inside the subloop inserted between the "search phase N" blocks, whose detailed structure is shown in Figure 7-6, inside the gray rectangle.

If a ravine is detected when the Ultrasonic Sensor measures a big distance, the searching is suspended, the robot spins to change its heading, and then the searching is resumed. If a mine is detected, the search loop is interrupted, the mine size is measured to find its center of mass, and then it is collected. After the mine is stored inside the robot's hold, the searching loop restarts from the beginning (phase 1).

Now you are ready to be introduced to the NXC program that implements the working just described. The program is divided into many subsections, labeled by commented headings; the various parts will be discussed separately. The program for the Mine Sweeper is in Listing 7-1.

**Listing 7-1.** *The Complete Mine Sweeper Program*

```
// NXT ports aliases
#define LIGHT           IN_1
#define USONIC          IN_4
#define WHEELS          OUT_AC
#define GRABBER         OUT_B
#define LEFT_WHEEL      OUT_A
#define RIGHT_WHEEL     OUT_C

#define SEARCH_WIDTH    100
#define SEARCH_SPEED    25
#define SEARCH_TIMEOUT  2000
#define CW  1     // clockwise
#define CCW -1    // counterclockwise

// detection events definitions
#define NONE            0
#define MINE_EV         1
#define RAVINE_EV       2
#define TIMEOUT_EV      3
#define EDGE_EV         4

// mine centering algorithm definitions
#define TOP             1
#define BOTTOM         -1
#define RIGHT           1
#define LEFT           -1
#define APPROACHING     1
#define DISMISSING      0
```

```
// macros
#define RAVINE_DETECTED (SensorUS(USONIC)>35)
#define MINE_DETECTED (Sensor(LIGHT)<threshold-8)
#define MINE_LOST (Sensor(LIGHT)>threshold-4)

// global variables
int threshold;
bool found;

//===============================================================//
// ROBOT INITIALIZATION //
//===============================================================//
sub GrabberZeroPosition()
{
  // bring the grabber to its zero position,
  // with the arm down and the grabber opened.
  int t;
  t = MotorRotationCount(GRABBER);
  // run the grabber motor
  OnRev(GRABBER,40);
  Wait(400);
  // while it is not stalled
  while( abs(t-MotorRotationCount(GRABBER))>14 )
  {
    t = MotorRotationCount(GRABBER);
    Wait(50);
  }
  // and stop it
  Off(GRABBER);
  Wait(50);
  Float(GRABBER);
}

sub MineSweeper_init()
{
  // initialize sensor ports
  SetSensorLight(LIGHT);
  SetSensorLowspeed(USONIC);
  // measure the ground color:
  // the robot must be started with no mine
  // under the detector
  threshold = Sensor(LIGHT);
  // show the threshold, given by the light color
  // of the ground, according to which dark mines are detected
  TextOut(5,LCD_LINE1,"Threshold: ");
  NumOut(70,LCD_LINE1,threshold);
  // close the grabber a bit to avoid forcing
  // the geartrain
```

```
  RotateMotorExPID(GRABBER,60,180,0,false,false,20,20,50);
  // and then reset its position
  GrabberZeroPosition();
  TextOut(5,LCD_LINE1,"                                    ");
}

//===========================================================//
//                    SUBROUTINES FOR SPINNING               //
//===========================================================//
sub AvoidRavine()
{
  // avoid the ravine by spinning,
  // to change the robot heading
  // to the opposite direction
  Off(WHEELS);
  RotateMotorEx(WHEELS,80,550,-100,true,true);
  OffEx(WHEELS,RESET_ALL);
}

sub Spin( short dir )
{
  // this is called during the search phases
  // to start the robot spinning
  // the parameter dir indicates the
  // spinning direction (CW or CCW)
  OnFwdReg(LEFT_WHEEL, sign(dir)*SEARCH_SPEED, OUT_REGMODE_SPEED);
  OnFwdReg(RIGHT_WHEEL, -sign(dir)*SEARCH_SPEED, OUT_REGMODE_SPEED);
  Wait(50);
}

//===========================================================//
//                      SEARCHING PHASES                     //
//===========================================================//
bool SearchPhase ( byte phase )
{
  // the research is divided into 3 phases:
  // 1 - the robot scans the ground spinning clockwise
  // 2 - the robot scans the ground spinning counterclockwise
  // 3 - the robot spins back to get the initial heading
  // this function returns true if the mine has been found,
  // and false otherwise
  byte detection;
  bool result;
  short span;

  detection = NONE;
  result = false;
```

```
// start spinning according to the search phase
if (phase == 1)
{
    Spin(CW);
    span = SEARCH_WIDTH;
}
if (phase == 2)
{
    Spin(CCW);
    span = 2*SEARCH_WIDTH;
}
if (phase == 3)
{
    Spin(CW);
    span = SEARCH_WIDTH;
}
// the following loop is interrupted when
// a ravine is detected
// a mine is found
// the robot has covered the current search width
while ( detection == NONE )
{
    if (RAVINE_DETECTED)
    {
    // if a ravine is detected, avoid it
    // and break out of this loop
        detection = RAVINE_EV;
        AvoidRavine();
    }
    else if (MINE_DETECTED)
    {
    // if a mine is detected, set result to true
    // and break out of this loop
        detection = MINE_EV;
        result = true;
    }
    else if (abs(MotorTachoCount(LEFT_WHEEL))>span)
    {
    // if the span has been covered,
    // break out of this loop
        detection = TIMEOUT_EV;
    }
}
```

```
    OffEx(WHEELS, RESET_NONE);

    return result;
}

//=============================================================//
//                        SEARCH THE MINE                      //
//=============================================================//
sub SearchMine()
{
    byte detection;
    // initialize "mine found" flag to false
    found = false;
    TextOut(0,LCD_LINE5,"SEARCHING MINE      ");
    // repeat the procedure until a mine is found
    until (found)
    {
        // phase 1 : search the mine spinning clockwise
        found = SearchPhase(1);

        if (!found) // do this if the mine has not been found yet
        {
            // phase 2 : search the mine spinning counterclockwise
            found = SearchPhase(2);
        }

        if (!found) // do this if the mine has not been found yet
        {
            // phase 3 : spin back to center
            found = SearchPhase(3);
        }

        if (!found) // do this if the mine has not been found yet
        {
            // advance a bit
            OffEx(WHEELS, RESET_ALL);
            RotateMotorEx(WHEELS,50,40,0,true,true);
        }
    }
}
```

```
//=============================================================//
// DETECT MINE EDGE, MEASURE MINE LENGTH //
//=============================================================//

/*
        top
        ____
       /O)O)|.              ^
       /O)O)// right       /
  left /O)O)//            length
       /O)O)//             /
       |____|/            v

        bottom

      <-width->
*/

byte WaitEdge( byte mode )
{
  // wait for the sensor reading to change
  byte event = NONE;
  unsigned long time = CurrentTick();

  while(event == NONE)
  {
    if ( (CurrentTick()-time) > SEARCH_TIMEOUT )
    {
      event = TIMEOUT_EV;
    }
    else if ( mode == DISMISSING )
    {
    // detect the transition from black to white
      if (MINE_LOST) event = EDGE_EV;
    }
    else if ( mode == APPROACHING )
    {
    // detect the transition from white to black
      if (MINE_DETECTED) event = EDGE_EV;
    }
  }

  return event;
}
```

```
int FindMineLength( short edge )
{
  // this function returns the left wheel rotation count (angle)
  // used to measure the mine length as the
  // distance between top and bottom edge
  int y;
  byte ev;
  // save start position
  y = MotorRotationCount(LEFT_WHEEL);

  OnFwdSync(WHEELS,sign(edge)*30,0);

  if (edge == BOTTOM)
  {
    // ignore the first edge found (the top one)
    WaitEdge(APPROACHING);
  }

  // wait for mine dismissing with timeout constraint
  ev = WaitEdge(DISMISSING);
  Off(WHEELS);
  if (ev == EDGE_EV)
  {
    // if the edge was found, save position
    y = MotorRotationCount(LEFT_WHEEL);
  }
  return y;
}

//===========================================================//
//                      CENTER THE MINE                      //
//===========================================================//
sub CenterMine()
{
  int length;
  // this file should be present in the NXT memory
  // if you have downloaded the latest complete firmware
  PlayFile ("! Attention.rso");
  TextOut(0,LCD_LINE5,"CENTERING MINE      ");

  // find top and top and bottom edges of the mine
  // to measure its length
  TextOut(0,LCD_LINE6,"    TOP EDGE      ");
  length = FindMineLength(TOP);
  TextOut(0,LCD_LINE6,"    BOTTOM EDGE   ");
  length = length - FindMineLength(BOTTOM);
  // and then center the mine
```

```
    RotateMotor(WHEELS,40,length/2);
    //
    // HERE YOU CAN ADD A SIMILAR PROCEDURE TO
    // MEASURE THE MINE WIDTH
    //
    // clear line 6 of the screen
    TextOut(0,LCD_LINE6," ");
}

//===============================================================//
//COLLECT THE MINE//
//===============================================================//
sub CollectMine()
{
    TextOut(0,LCD_LINE5,"COLLECTING MINE");
    // move forward to get the mine between the claws
    RotateMotorEx(WHEELS, 50, 115, 0, true, true);
    ResetAllTachoCounts(WHEELS);
    StopSound();
    // grab the mine and lift it
    RotateMotorPID(GRABBER,90,1460,30,30,60);
    // release the mine into the hold, and lower the arm again
    RotateMotorExPID(GRABBER,-100,1000,0,false,false,30,20,50);
    GrabberZeroPosition();
}

task main ()
{
    // call the initialization subroutine
    MineSweeper_init();

    // execute the sequence of actions forever
    // using the preceding subroutines
    while ( true )
    {
        SearchMine();
        CenterMine();
        CollectMine();
    }
}
```

The program starts, as always, by executing the main task, which is the only task running in this program. In main, the MineSweeper_init() subroutine is called to perform robot initialization. After the sensor ports are configured to read data from the Light Sensor and the Ultrasonic Sensor, the color of the ground is acquired to set the threshold that will be used in the rest of the program to distinguish the dark mines on the ground. Then, the grabber arm is brought into its zero position by the GrabberZeroPosition() subroutine, using the useful motor stall detection algorithm you have seen throughout the book.

After initialization, the program flow comes back to the main task, where a perpetual loop begins: here inside the mine, searching, centering, and collecting procedures are called, one after another. You may begin to compare these parts of the code with the corresponding abstract blocks of the flow chart in Figure 7-6.

Let's dive into the detailed description of these three procedures, proceeding in order. The SearchMine() subroutine contains a loop that sequentially activates phases 1, 2, and 3 of the search, calling the SearchPhase(byte phase) function, passing the phase number as an argument. The program flow breaks out the SearchMine() loop only if a mine is found during one of those phases. Also, notice that the SearchPhase function calls after the first are performed only if the mine has not already been found (and the found variable is false). In fact, when the mine is right under the sensor, the robot must not continue with the successive scanning phases: the SearchMine() subroutine returns, and the main task can call the other subroutines.

The SearchPhase function implements the subloops shown in gray in Figure 7-6, returning a Boolean value that is true if a mine has been detected during the scan, false otherwise. Inside this function, the ravine avoidance maneuver can be eventually triggered. According to the phase argument passed to this function, the robot spins clockwise or counterclockwise, and the span covered is determined by the value of the span variable. Every time that the motors are started, their Tacho Count register is reset. So, the condition abs(MotorTachoCount(LEFT_WHEEL))>span allows you to check if the left wheel has turned by the number of degrees specified by the span variable.

---

■ **Caution** The motors' Tacho Count registers are always reset by the standard functions such as OnFwd, OnRev, Off, and other functions, unless you use their counterparts with the Ex postfix, as OnFwdRegEx, OnFwdSyncEx, OffEx, and so on. Among the arguments accepted by these extended functions, you can specify which motor-related registers you want to reset, or none of them. For details about all the motors' control registers, consult the NXC Programming Guide.

---

The Spin(short dir) subroutine simply runs the motors in opposite directions, calling the OnFwdReg NXC function. Using this function, the NXT firmware turns the motors on and regulates their speed precisely.

Once a mine is found, the program flow goes back to main task. Here, the CenterMine() and CollectMine() subroutines are called sequentially. The CenterMine() subroutine attempts to find the mine's center of mass, measuring the mine's length. The program is left open to the development of a more refined centering procedure; for example, also measuring the mine's width. However, given that the size of the collectable objects is almost known, the grabber rarely fails in collecting them, even if they are not precisely aligned with respect to the robot's direction.

You measure the mine length by calling the FindMineLength(short edge) function, passing as an argument the edge constant values TOP and BOTTOM. This function returns the left wheel's motor rotation count at the moment of the edge detection. The information about the other wheel angle is not important, because the motors are running synchronized. You can determine the center of the mine by subtracting the value returned by the second call from the wheel angle that's returned by the first FindMineLength call, and dividing the result by 2. The CenterMine subroutine calculates and uses this value to move the robot forward. Use Figure 7-7 as a reference.

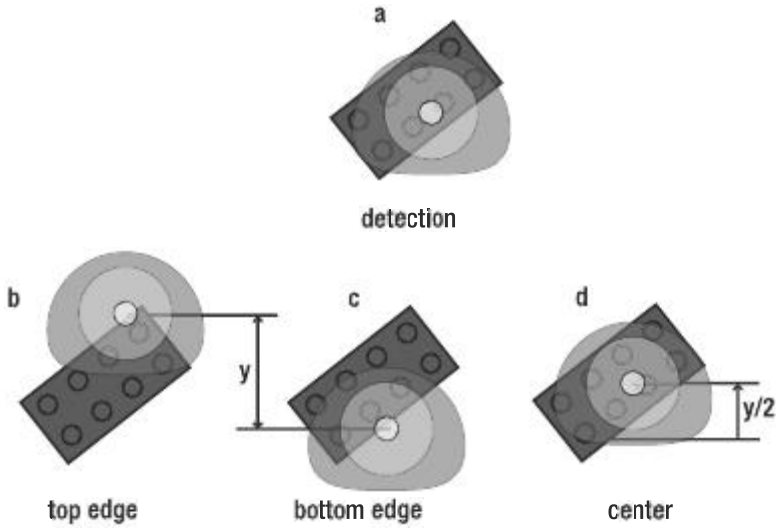



**Figure 7-7.** *This scheme shows the phases to align the mine before collecting it.*

Once the mine is detected (*a*), its precise position under the sensor is unknown; the only known thing is that the Light Sensor is reading a value well below the threshold value. The function FindMineLength saves the actual left wheel rotation count and, receiving the constant TOP as an argument, moves the robot forward until a transition from black to white occurs (*b*: top edge). This waiting is done by calling the WaitEdge(byte mode) function: the argument mode allows the caller to specify WaitEdge to wait for a black-to-white transition, or for a white-to-black transition, implemented by the two macros MINE_LOST and MINE_DETECTED, respectively. The first waiting mode is associated with the constant DISMISSING, meaning that you want to wait for the mine to go away from the sensor halo; the second mode is given the name APPROACHING, meaning that you want to wait for the mine to approach, until it comes under the sensor halo. To avoid unpredictable color measurement problems making this waiting become infinite, you can use a timeout mechanism. The timeout is detected using the same technique you have encountered in Chapter 6. The constants and macros the WaitEdge function uses are summarized in Table 7-1.

**Table 7-1.** *Constants Used in the Edge Detection Functions*

| Waiting Mode | Macro Used | Expanded Code | Color Transition |
|---|---|---|---|
| APPROACHING | MINE_DETECTED | Sensor(LIGHT)<threshold-8 | White to black |
| DISMISSING | MINE_LOST | Sensor(LIGHT)>threshold-4 | Black to white |

After an edge is detected or the timeout has elapsed, the WaitEdge function returns a value describing which of the two events has occurred. If the edge was found before the timeout, FindMineLength returns the actual left wheel rotation count, otherwise it returns the value saved before the edge detection.

Once FindMineLength(TOP) returns, the CenterMine() subroutine calls FindMineLength(BOTTOM), to detect the bottom edge (*c*). The only difference, with respect to before, is that receiving

BOTTOM as an argument, FindMineLength() first waits for a transition from white to black—WaitEdge(APPROACHING) —and then for another transition from black to white—WaitEdge (DISMISSING). This is necessary, because the top edge detection implies that the dark mine is not under the sensor anymore: the robot was going forward, and when it stopped after the top edge detection, it surpassed the mine a bit. So, when going backwards, the robot must wait for the mine to be detected again in correspondence with the top edge, and then wait for the bottom edge to be detected. Once the bottom edge is found, in correspondence with a black-to-white transition, the second measurement of the wheel angle is taken.

The CenterMine() subroutine subtracts the two measurements, halves the result, and moves the robot forward (⁄). The mine should be now under the sensor. The CenterMine() subroutine returns, and the next subroutine called by main is the one to grab and store the mine.

The distance from the sensor to the grabber is fixed, determined by the structure of the robot. So, the CollectMine() subroutine makes the robot advance by the amount of space needed to bring the mine between the grabber claws; the grabber motor is rotated by a precise number of degrees, and our good ol' underactuated mechanism does the rest! The mine should be released correctly into the hold opening, shaped to help the mines to fall down, under the central motor. Note that if the mine was placed in an unfavorable way on the ground, an aspect of the whole procedure could go wrong: the mine could not be picked up at all, or could become stuck with the frontal scanner during the lifting, or could remain in the hold opening instead of falling down into the hold.

After the collection, the grabber motor is reversed, to bring the arm back down; the appropriate subroutine brings the grabber into its zero position.

---

■ **Note** RotateMotorEx and RotateMotorExPID are extended versions of the basic RotateMotor NXC function, whose arguments are motor Port, Power, and Angle. With RotateMotorEx, you can rotate two motors together using the Sync feature, and tell the motor to come to a stop after having turned the specified number of degrees. The other function RotateMotorExPID rotates the motor like the preceding functions, but also allows you to change the internal Proportional, Derivative, and Integrative gains of the PID controller that is run by the NXT firmware. The Proportional contribution gives promptness to the motor response, the Integrative contribution eliminates the steady state error (deviation from the desired position after the motor is stopped), while the Derivative contribution compensates for the oscillations generated by the Integrative and Proportional combined contribution, to reach the steady state faster. Consider, however, that the PID control is an argument well worth dedicating entire books to. For details about those NXC functions, check the Programming Guide.

---

Once the mine has been collected, the loop inside the main task restarts from the beginning, searching for the next mine on the robot's path.