NXT AT-ST

Also known as "chicken walker," because of its shape and walking motion, the All Terrain Scout Transport (AT-ST) is a bipedal war craft employed by the Galactic Imperial Forces in the *Star Wars* saga.

In this chapter, you'll build the AT-ST biped shown in Figure 4-1, guided by detailed building instructions. You'll program it to walk around, and by the end of this chapter, you'll have at your command one of the most famous battle robots in the history of cinema.

Figure 4-1. *The impressive-looking NXT AT-ST*

Design Thoughts

AT-STs were seen in the *Star Wars* movies in the Battle of Hoth in *The Empire Strikes Back* and the Battle of Endor in *Return of the Jedi*. The AT-ST has chin-mounted double laser cannons, a concussion grenade launcher on the right side of its head, and a blaster cannon on the left. The bipedal propulsion system is the strength of the *Star Wars* AT-ST, allowing it to move its weaponry across uneven terrain that a wheeled unit would not be able to traverse. This craft can carry one pilot and one gunner, with a maximum speed of 90km/h. Even though it's not as imposing as its larger All Terrain Armored Transport (AT-AT) quadruped walker cousin, the AT-ST serves as a sort of robotic cavalry to the Imperial side on the battlefields of the *Star Wars* films.

The NXT AT-ST walker shown in Figures 4-1 and 4-2 is mainly built with NXT retail set parts, but includes a few extra parts, needed just to improve the design. These additional parts aren't structural, so don't worry if you don't have them in your LEGO spare reserves. You'll be guided in how to build the alternative retail-set-only version in the building section of this chapter.

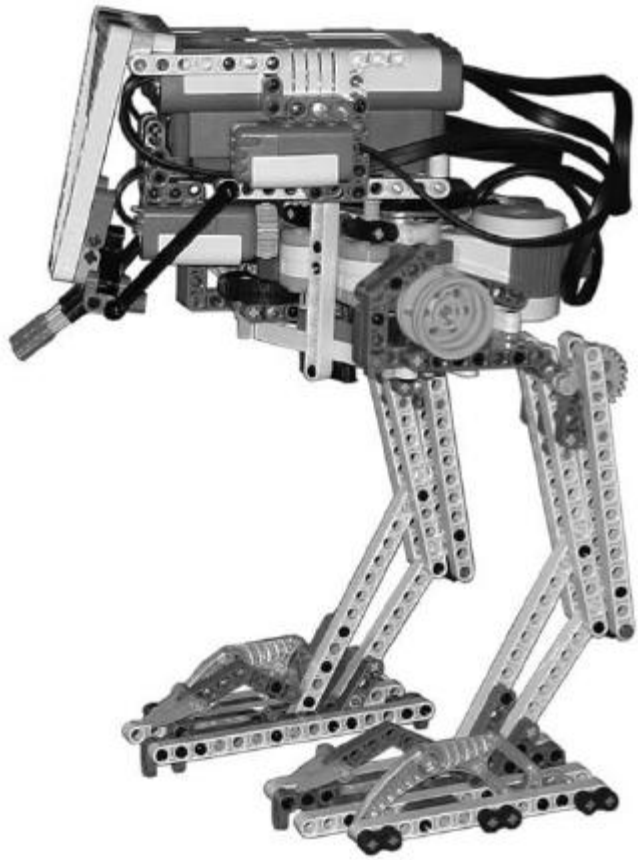


Figure 4-2. *Another view of the NXT AT-ST*

I tried hard to reproduce all the AT-ST features in a well-proportioned way, from the particular leg shape to the head profile. As you might guess, the robot you're going to build cannot move across uneven terrain; on the contrary, the surface to walk on must be smooth and plain. Also, the robot cannot carry humans and won't hurt anybody, because the weapons have been replaced by the Sound Sensor and the Ultrasonic Sensor.

This model wasn't designed in a day! It was difficult to get to the final shape. In Figure 4-3, you can see the AT-ST in one of its early stages of development. The legs were not at all similar to the final ones, and the head was disproportionate. On both feet, I used a Touch Sensor to know which side the robot was leaning on; this feature proved to be useless in the final robot version due to a new, timed approach. Also, notice the tendon made with the ball joint steering link that prevents the hip from bending (as in Quasimodo in Chapter 2). Although a bit raw, this old prototype already featured all the key ideas that brought me to the final AT-ST presented here.

Figure 4-3. *An early prototype of the NXT AT-ST*

This LEGO MINDSTORMS biped perfectly fits in the jerky COG shifting category introduced in Chapter 1. It uses only two motors to accomplish the needed movements, while other robots fitting in this category, which you might have seen on the Web, generally use three motors. Here, one motor shifts the weight (mostly concentrated in the head) by turning the neck turntable, while the other motor rotates the legs in sync, as shown in the plan back in Figure 1-4*b*. Because these movements are done one after another, the gait of this biped is jerky.

Using only two motors allowed me to lighten the whole structure a bit, so that the legs of the AT-ST, accurately reproduced in all their slimness, could support the upper body weight. You can understand what I mean here by "accurately" by looking at Figure 1-5*c* in Chapter 1. The AT-ST I made with the RCX had squat, boxy legs, not at all similar to the real leg shape or the elegance of the legs of the actual NXT version.

I tried to keep the feet as small as possible, always bearing in mind, however, that small feet yield poor stability. On the internal side of both feet, I placed two wedges that are the only touch point of the feet when the head is perfectly centered. When the head turns to the side, only one of the feet will touch the ground with the wedges and with the external rubber edge. If the feet were totally flat (entirely touching the ground), the AT-ST would have needed another mechanism to bend the ankles. With this solution, the legs can be made strong and rigid, because they have no moving joints. The robot walks straight by shifting the head weight aside and suddenly stepping forward, when the loaded foot is touching the ground with wedges and rubber elements, and the other is off the ground.

Turning is a little more complicated, and the performance might vary according to the nature of the surface the robot is walking on. Carpets are the worst surface you can imagine, while flat, smooth surfaces such as tables or parquet are perfect. For example, let me explain how the robot turns right. First, it rotates the legs while they are both on the ground (the head is centered), so that the right foot is in front of the left one. Then, the weight is shifted to the right, and the left foot is suddenly brought forward to get aligned with the right foot again. Repeating this many times, the AT-ST can turn right; and, doing the opposite, it can turn left. This can sound complicated, but don't worry if you don't have a clear understanding of what is going on here. Once you see the robot walking, driven by the program you'll learn later, all the confusion will melt away.

The head contains all the sensors: the side weapons are the Sound Sensor and the Ultrasonic Sensor. A Touch Sensor is used to detect if the head has reached its turning limits; this sensor is hidden (not so well, actually!) under the AT-ST face.

Programming the AT-ST

The program discussed in this section gives your robot the ability to walk everywhere, by avoiding obstacles. Download the sound files provided, together with the source code, to the NXT using BricxCC, so that the NXT can play them.

■ **Tip** You can find the files in the Source Code/Download area on the Apress web site at http://www.apress.com. Use Appendix A as a guide for downloading files to the NXT.

When you start the AT-ST program, it aligns the legs and the head in the center. Then the AT-ST starts walking straight as described earlier, until it sees an obstacle. At this point, it produces a twin-laser sound, it chooses a random turning direction, and starts steering. It should turn changing its direction by about 90 degrees, although this depends on the slipperiness of the surface it is walking on. Then it starts walking straight, as before.

"Again? Another walker with obstacle avoidance?" you say. Well, again: having well-performing hardware doesn't mean that making it work well is easy. Therefore, here I'll focus attention on the routines that manage the mechanical parts to allow the robot to walk correctly. This is not at all a trivial issue, especially concerning the right timing of the motor movements. Also, the legs and the head orientation are regulated by FSMs. So, this is the perfect occasion to see in practice what you read about in the previous chapter. Once you have this core software ready to work, you can adapt the program to do anything you want—even control the robot remotely.

As with any NXC program, the declaration of all the constants and macros used in the program appears at the top of the code, shown in Listing 4-1.

■ **Note** A macro is an operation defined inside a `#define` directive. Before the NXC compiler starts to translate your code into instructions readable by the NXT processor, the NXC preprocessor expands the macro as follows. If the macro has no arguments, such as `#define NEAR 20`, the preprocessor will replace `NEAR` with the constant 20 every time it encounters the word `NEAR`. If the macro has arguments, such as `#define TWO_TIMES(x) 2*(x)`, then the argument `x` will be replaced with the argument inside the brackets of a call, such as `y = TWO_TIMES(3)`. So, the preceding will be expanded into `y = 2*(3)`, storing the result 6 inside the variable `y`. Notice that you could also call `y = TWO_TIMES(3+4)`, which would be expanded into `y = 2*(3+4)`, yielding 14. If you omitted the parentheses in the macro, the expression would be wrongly translated as `y = 2*3+4`, yielding 10 as a wrong result. Writing macros in all capital letters isn't strictly required, but it is a useful convention. When you see a word with all capital letters, it is probably defined as a macro.

Listing 4-1. *The AT-ST Program Definitions*

```
#define TOUCH IN_3 //short cable
#define SONAR  IN_1 //mid cable
#define MIC IN_4 //mid_cable
#define LEGS OUT_C
#define HEAD OUT_A

#define NEAR 20
#define LEFT 0
#define CENTER 1
#define RIGHT 2
#define TURN_RIGHT 1
#define TURN_LEFT -1
#define WALK 0
#define STOP 5

#define OBSTACLE (SensorUS(SONAR)<NEAR)
```

The main task code is shown in Listing 4-2.

Listing 4-2. *The main Task Code of the AT-ST Program, After Declaring Global Variables*

```
// global variables
int weightState, legsState, runState;
[...]
task main ()
{
  ATST_init();
  int action;

  while (true) {

    if(OBSTACLE)
      runState = 1-2*(Random(2)); // 1, -1
    else
      runState = WALK;

    switch(runState) {
      case TURN_RIGHT:
        TurnRight(9);
        break;
      case TURN_LEFT:
        TurnLeft(9);
        break;
      case WALK:
        GoStraight();
        break;
    }
    CenterHead();
    CenterLegs();
  }
}
```

Outside every task, function, or subroutine, you see three *global* variables declared; every function in the program can access these variables. By contrast, a variable declared inside a function (for example the action variable inside the main task) is called *local* and can be used only by the function inside which it is declared. If any other function tries to use that local variable inside its body, it would cause a compiler error. In fact, the compiler would complain about the presence of an Undefined `Identifier` action, because it would not know what action is.

At the start, the ATST_init() function is called to perform hardware initialization; in this case, to tell the NXT where each sensor is attached and to reset the head and the legs to their zero position. These operations are not trivial, as you will see later. Next, the program enters an infinite loop, as was the case for Quasimodo (see Chapter 2). Inside this loop, the whole basic AT-ST behavior is expressed. Let's analyze the code by breaking it into smaller chunks:

```
if(OBSTACLE)
    runState = 1-2*(Random(2)); // 1, -1
else
    runState = WALK;
```

With if(OBSTACLE), we're checking if the Ultrasonic Sensor is detecting an obstacle. Well, this could be clear already, but how is it done? OBSTACLE is a macro defined previously as follows:

```
#define OBSTACLE (SensorUS(SONAR)<NEAR)
```

This is a common handy way to have a piece of code replaced by an easier-to-remember macro. Every time the compiler meets this constant OBSTACLE later in the program, it will replace it completely with (SensorUS(SONAR)<NEAR).

■ **Note** SONAR is another alias that stands for IN_1: the input port constant to which the Ultrasonic Sensor is attached.

So, if(OBSTACLE) becomes if(SensorUS(SONAR)<NEAR) for the compiler, after the preprocessor has finished its work. If the Ultrasonic Sensor reading is less than NEAR (another constant with the value of 20), then the runState variable will be assigned a random value that can be 1 or -1 (corresponding to the right or left direction). Otherwise, it will be assigned the constant value 0 (WALK). The function Random(2) returns a random value that can be 0 or 1. So, runState = 1-2*(Random(2)) assigns the runState variable a value that can be 1-2*(0) = 1 or 1-2*(1) = -1.

■ **Tip** The Random(n) function returns a random number between 0 and n-1.

The switch statement then uses the runState variable to decide what to do: to turn right or left, whether runState has the TURN_RIGHT or TURN_LEFT value, or to go straight if it corresponds to WALK. In the turning functions, the number inside the brackets indicates how many times the turning movement pattern must be repeated. If the AT-ST decides to walk straight, it will stop only when it detects an object. After the walk, in whichever direction, the CenterWeight() and CenterLegs() subroutines realign the head and legs.

Now that you know all about the basic behavior of the AT-ST, it's time to see what's inside the subroutines to realign the head and the legs. Let's dissect them by looking at Listing 4-3, for the leg-centering subroutine.

Listing 4-3. *The CenterLegs Subroutine*

```
sub CenterLegs()
{
    int t;
    t = MotorRotationCount(LEGS); //save actual position
    OnFwd(LEGS,65);
```

```
    Wait(100);
    // if position does not change more than specified angle in specified time
    while( abs(t-MotorRotationCount(LEGS))>10 )
    {
      t = MotorRotationCount(LEGS);
      Wait(100);
    }
    Off(LEGS);
    RotateMotor(LEGS,50,-120);
    Wait(200);
    legsState = CENTER;
}
```

The legs have no evident sensor to let the NXT know which direction they are oriented in. Don't forget, we aren't working with mere motors. The NXT motors contain optical encoders to measure the shaft's relative angle. Thus, all the sensors we need are already inside the motors. The motor's actual angle is saved into the variable `t` and the motor is started to orient the legs to the left. How can you know if the legs have reached their limit position? Here, I adopted a trick to measure the motor shaft's speed: after the motor has started, the NXT continuously checks if the motor shaft is rotating to a minimum number of degrees (10) in a certain period of time (100ms). The small angle and the time interval were chosen appropriately for the application. What I said in words can be translated into this code:

```
    while( abs(t-MotorRotationCount(LEGS))>10 )
    {
      t = MotorRotationCount(LEGS);  //update starting angle
      Wait(100);  //wait 100ms
    }
```

While the angle of the shaft varies more than 10 degrees in the time interval of 100ms, the program doesn't stop the motor. However, if the motor cannot accomplish this angle in this little bit of time, it means that the legs are stopped by something and the motor is stalling, so it is turned off. The shaft speed is measured every 100ms as the difference between the angle stored in the variable `t` and the angle measured time by time with the MotorRotationCount() function.

■ **Note** Measuring an increment of a shaft angle inside a time window means measuring the shaft's speed; in fact <speed> = <angle increment>/<time>, s = ⊗a/t. This effective technique is frequently used in robotics but has no common name. Let me give it a pompous name: servomotors automagic built-in limit switch!

Now that the legs are posed in a known direction, you can bring them to their center position with RotateMotor(LEGS,50,-120), where 120 is the measured number of degrees to rotate the legs from the left direction to the center. The legs are now realigned! Let's see how to center the head, where we have a Touch Sensor, by looking at Listing 4-4.

Listing 4-4. *The Subroutine to Center the Head*

```
sub CenterHead ()
{
    #define CNT_SPEED    50
        OnFwd(HEAD,CNT_SPEED);  //bring the head to the right
        Wait(400);
        Off(HEAD);

        if (Sensor(TOUCH)) //head was already at right or center
        {
            OnRev(HEAD,CNT_SPEED);
            while (Sensor(TOUCH));
            Off(HEAD);
            RotateMotor(HEAD,CNT_SPEED,-30);
        }

        else if (!Sensor(touch)) //head was already at left
        {
            OnRev(HEAD,CNT_SPEED);
            until (Sensor(TOUCH));
            Off(HEAD);
            RotateMotor(HEAD,CNT_SPEED,60);
        }

    weightState = CENTER;
}
```

Reading the Touch Sensor only tells you if the head is turned completely to the right or to the left (sensor closed), but gives you no information about the actual head direction. This is not at all a subtle difference; we only know if the head is turned, but not in which direction! Here, only clever programming can get us out of the trouble. Read on carefully to see how, remembering Listing 4-4.

At the beginning, the head is rotated to the right by the motor, without checking the sensor yet. Only after that can the program check whether the Touch Sensor is closed or not. If it is closed, that means the head is now at the right (the initial turning brought it here). In this case, the motor turns the head to the left until the sensor opens again; it stops and turns a little more to compensate for the remaining constant offset to get to the exact center.

If the Touch Sensor is open after the first "blind" turn, we assume (being right!) that the head was initially pointed left. In this case, the motor turns the head to the left until the sensor is closed and the head has reached the full left side, and stops. The last RotateMotor(HEAD,CNT_SPEED,60) brings the head to the center from the leftmost known position. Whew, we did it!

Only the functions to make the AT-ST step and lean aside remain to be covered. ("Only" is just a manner of speaking, of course.) These apparently simple routines are worth a lot of attention and explanation. The subroutine to rotate the head implements an FSM using the `if-then-else` statements, while the other subroutine that moves the legs adopts a decision table (explained in Chapter 3) to make the code cleaner and more elegant. Let's deal with the first subroutine right now. Take a look at Listing 4-5, which enables the AT-ST to lean aside, turning the head left and right.

Listing 4-5. *The Code to Lean the Head*

```
sub Lean (int newState)
{
  if ( weightState != newState )
  {

    if (weightState==CENTER) //head is at center
    {
      if (newState==RIGHT) OnFwd (HEAD,LEAN_SPEED);
      if (newState==LEFT) OnRev (HEAD,LEAN_SPEED);
      until (Sensor(TOUCH));
      Off (HEAD);
    }

    if (weightState==LEFT)  //head is at left
    {
      if (newState==CENTER)
      {
        OnFwd(HEAD,LEAN_SPEED);
        while (Sensor(TOUCH));
        Off(HEAD);
        RotateMotor(HEAD,LEAN_SPEED,40);
      }
      if (newState==RIGHT)
      {
        OnFwd(HEAD,LEAN_SPEED);
        while (Sensor(TOUCH));
        until (Sensor(TOUCH));
        Off(HEAD);
      }
    }

    if (weightState==RIGHT) //head is at right
    {
      if (newState==CENTER)
      {
        OnRev(HEAD,LEAN_SPEED);
        while (Sensor(touch));
        Off(HEAD);
        RotateMotor(HEAD,LEAN_SPEED,-30);
      }
```

```
            if (newState==LEFT)
            {
                OnRev(HEAD,LEAN_SPEED);
                while (Sensor(TOUCH));
                until (Sensor(TOUCH));
                Off(HEAD);
            }
        }
        weightState = newState;
    }
}
```

The new desired position for the head is passed on as the newState argument to the Lean subroutine. If newState is equal to weightState (the state variable that keeps track of the head position), nothing has to be done because the head is already in the desired position. If not equal, different actions are performed according to the new state passed to the subroutine. To get the center from a known side, the motor turns until the Touch Sensor is cleared. To lean aside (say to the left), the program checks if the head is actually right or centered. If it is to the right, the Touch Sensor is pressed and the motor turns until the sensor is cleared, and then is pressed again. You use a similar procedure to get to the right from the left.

■ **Tip** Using the `weightState` variable (and state variables in general) allows you to know the direction of the head without reading any sensor, by reading (and trusting) this variable instead.

Listing 4-6 shows the second subroutine to make your imposing AT-ST take steps.

Listing 4-6. *Making the AT-ST Take Steps*

```
sub Step(int newState, bool slow)
{
    int speed = 65;
    if(newState!=legsState)
    {
        if (slow) speed = 45;
        RotateMotorPID(LEGS,speed,legsAngles[newState+3*legsState],30,20,80);
        legsState = newState;
    }
}
```

■ **Note** RotateMotorPID is an advanced version of the NXC function RotateMotor. The basic function is the same, except you can specify three additional parameters: P, I, and D. Leave them for now; we'll come back to them briefly in Chapter 7.

Here you can see a good example of implementing a decision table: a technique associated with FSMs, which you saw in theory in Chapter 3. In fact, the Step subroutine accepts as an argument the variable newState, which is used together with legsState (the variable that keeps track of the legs' state) to index the table legsAngles. Inside this table you find the appropriate angles to rotate legs from legsState to newState. A table can be viewed as an array with two dimensions—in other words, as a matrix. In NXC, you can declare multidimensional arrays. However, working with them is tricky and not worth the effort for this simple application, so it is preferable to implement the decision table as a single dimensional array. The legsAngles decision table and corresponding array declaration are shown in Table 4-1 and Listing 4-7.

Table 4-1. *legsAngles Decision Table with the Angles to Move the Legs*

	newState	LEFT	CENTER	RIGHT
oldState				
LEFT		0	-S	-2*S
CENTER		S	0	-S
RIGHT		2*S	S	0

Listing 4-7. *The Array Implementing Decision Table 4-1*

```
const int legsAngles[] = {      0      , -STEP_TURN, -2*STEP_TURN,
                           STEP_TURN,      0      , -STEP_TURN,
                         2*STEP_TURN,  STEP_TURN,      0      };
```

For example, if you want to go from LEFT (which oldState is equal to) to RIGHT (passed as the newState argument), you'd use this code:

```
RotateMotorPID(LEGS,speed,legsAngles[newState+3*legsState],30,20,80);
```

The preceding code rotates the legs' motor by the angle -2*STEP_TURN, which is found in the legsAngles table at row 0 (LEFT is defined as 0; see the program definitions at the beginning of this section) and column 2 (RIGHT is an alias for 2). To be clear, legsAngles is a one-dimensional array, indexed as follows:

legsAngles[column+3*row]

row and column can be 0, 1, or 2. In the case of this example, the index value is 2+3*0 = 2, and shows where to find the needed angle. You can see other example cases in Table 4-2.

Table 4-2. *Examples of How the Stepping Decision Table Works*

Old State	New State	Index Value	Corresponding Angle
LEFT(0)	CENTER(1)	index = 1+3*0 = 1	(-STEP_TURN)
CENTER(1)	RIGHT(2)	index = 2+3*1 = 5	(-STEP_TURN)
RIGHT(2)	LEFT(0)	index = 0+3*2 = 6	(2*STEP_TURN)
CENTER(1)	CENTER(1)	index = 1+3*1 = 4	(0), as one would expect

■ **Caution** Array indexes are 0-based; for example, the first element of `array` is `array[0]`, and the last element is at an index that equals `ArrayLen(array)-1`.

I find this solution cleaner and more elegant than using a nested **if** or a switch statement that would result in much messier code. Now you have an idea of how low-level routines (to manage AT-ST mechanics) work. This software represents the skeleton for any program you would write for your AT-ST, without having to worry about low-level mechanical management.

Building Your Personal AT-ST

Before you throw yourself into building a robot, you should read the following brief notes. To get the best robot appearance, I used some extra parts besides the NXT retail set parts. They aren't hard-to-find elements, and they're used just to embellish the model.

Don't worry; the robot's functionality is not compromised if you don't have these elements. Figure 4-4 shows the parts needed, and Table 4-3 points out which parts are not included in the retail set. Follow the instructions carefully to know which steps to skip and which parts to replace.

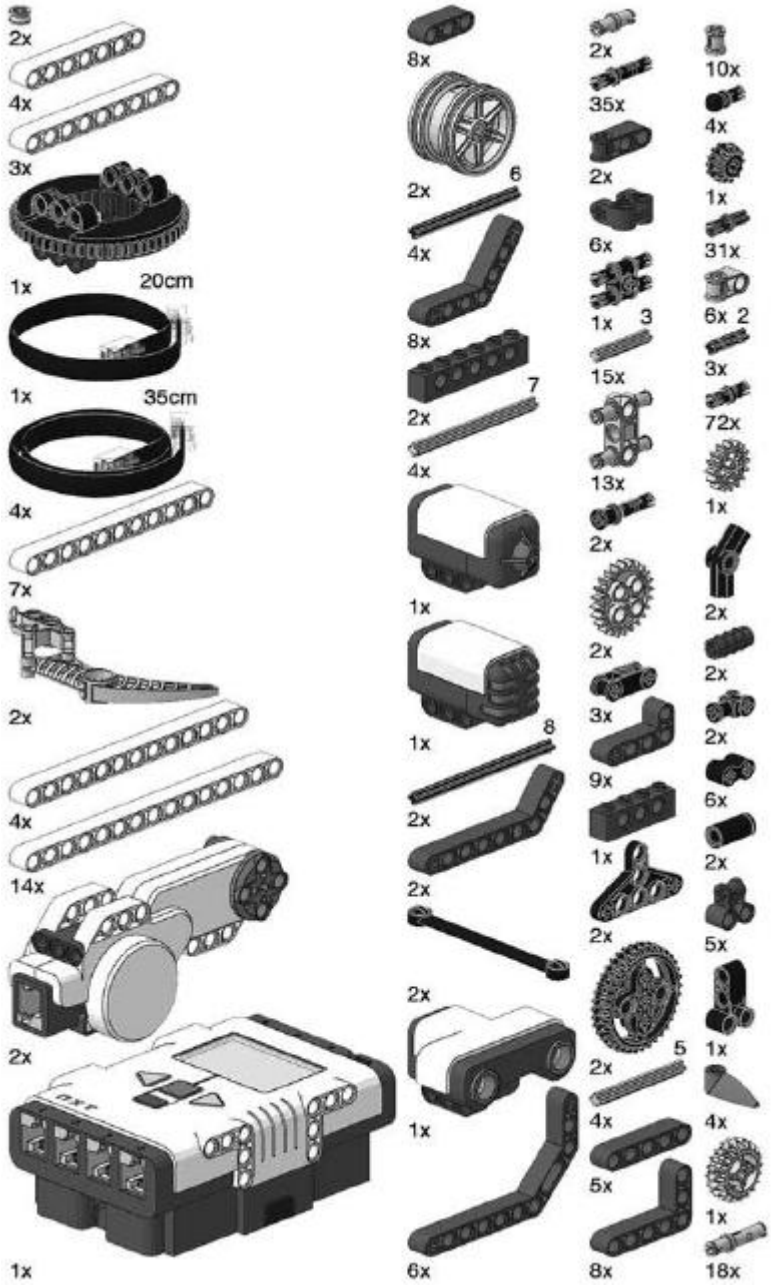


Figure 4-4. *NXT AT-ST bill of materials*

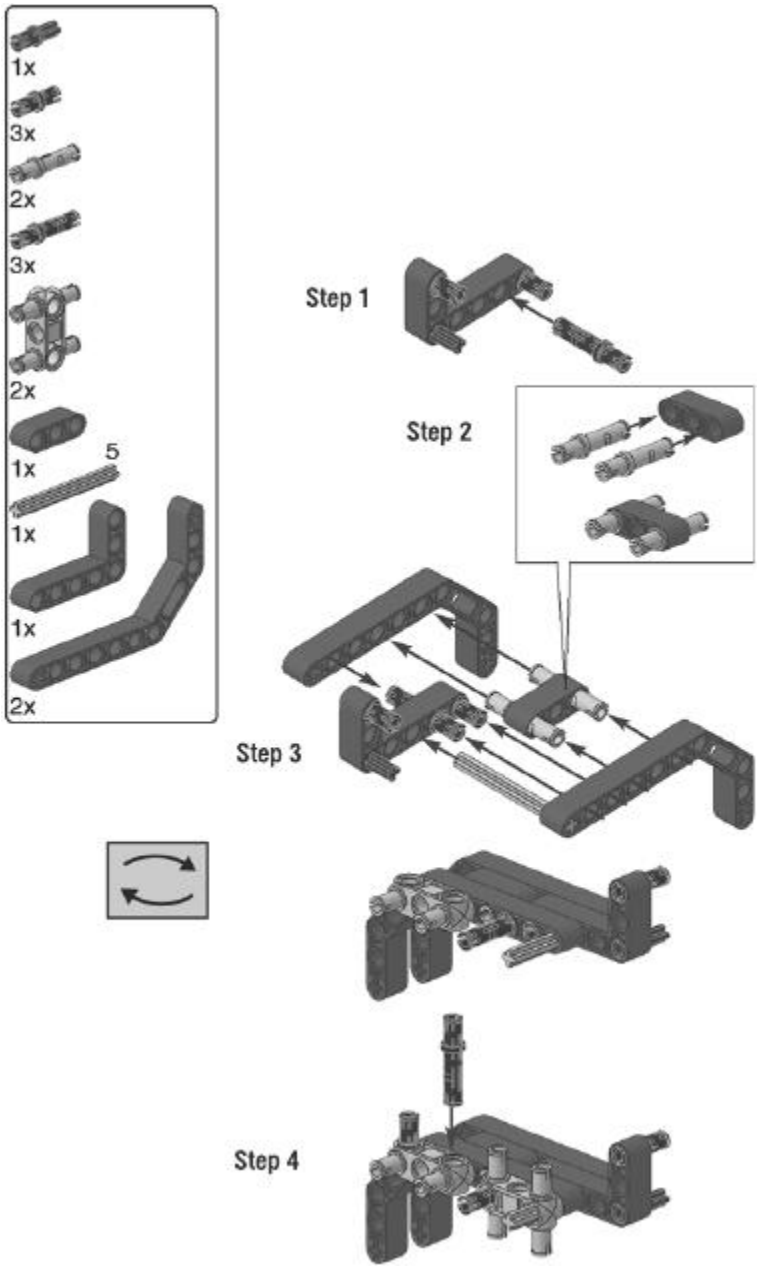
Table 4-3. *NXT AT-ST Bill of Materials*

Quantity	Color	Part Number	Part Name
4	White	32524.DAT	TECHNIC Beam 7
3	White	40490.DAT	TECHNIC Beam 9
1		4845x.DAT	TECHNIC Turntable New
1	Black	55804.DAT	Electric Cable NXT 20cm
4	Black	55805.DAT	Electric Cable NXT 35cm
7	White	32525.DAT	TECHNIC Beam 11
2	Light gray	50914.DAT	TECHNIC Bionicle Weapon Pincer Suukorak
4	White	41239.DAT	TECHNIC Beam 13
11	White	32278.DAT	TECHNIC Beam 15
2		53787.DAT	Electric MINDSTORMS NXT Motor
1		53788.DAT	Electric MINDSTORMS NXT
8	Dark gray	32523.DAT	TECHNIC Beam 3
4	Black	3706.DAT	TECHNIC Axle 6
8	Dark gray	32348.DAT	TECHNIC Beam 7 Liftarm Bent 53.5 (4:4)
2	Dark gray	3894.DAT	TECHNIC Brick 1 - 6 with Holes
4	Light gray	44294.DAT	TECHNIC Axle 7
1		55963.DAT	Electric MINDSTORMS NXT Sound Sensor
1-		53793.DAT	Electric MINDSTORMS NXT Touch Sensor
2	Black	3707.DAT	TECHNIC Axle 8
2	Dark gray	32271.DAT	TECHNIC Beam 9 Liftarm Bent 53.5 (7:3)
2	Black	32293.DAT	TECHNIC Steering Link 9L
1		56467.DAT	Electric MINDSTORMS NXT Ultrasonic Sensor
6	Dark gray	32009.DAT	TECHNIC Beam 11.5 Liftarm Bent 45 Double
2	Light gray	3673.DAT	TECHNIC Pin
33	Black	6558.DAT	TECHNIC Pin Long with Friction and Slot
2	Dark gray	42003.DAT	TECHNIC Axle Joiner Perpendicular with 2 Holes
6	Dark gray	41678.DAT	TECHNIC Axle Joiner Perpendicular Double Split
1	Black	32136.DAT	TECHNIC Pin 3L Double
11	Light gray	4519.DAT	TECHNIC Axle 3
13	Light gray	48989.DAT	TECHNIC Axle Joiner Perpendicular 1 - 3 - 3 with 4 Pins
2	Black	32054.DAT	TECHNIC Pin Long with Stop Bush
2	Light gray	3648.DAT	TECHNIC Gear 24 Tooth
3	Black	32184.DAT	TECHNIC Axle Joiner Perpendicular 3L
9	Dark gray	32140.DAT	TECHNIC Beam 5 Liftarm Bent 90 (4:2)

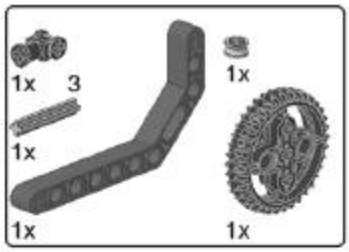
Continued

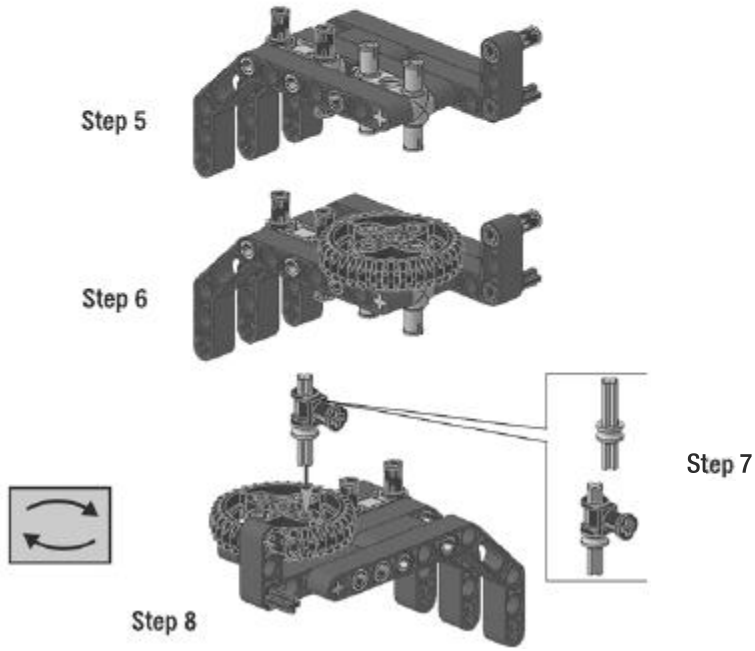
Table 4-3. *(Continued)*

Quantity	Color	Part Number	Part Name
1	Dark gray	3701.DAT	TECHNIC Brick 1 - 4 with Holes
2	Black	2905.DAT	TECHNIC Liftarm Triangle 5 - 3 - 0.5
1	Black	X344.DAT	TECHNIC Gear 36 Tooth Double Bevel
4	Light gray	32073.DAT	TECHNIC Axle 5
5	Dark gray	32316.DAT	TECHNIC Beam 5
1	Dark gray	32526.DAT	TECHNIC Beam 7 Bent 90 (5:3)
10	Light gray	3713.DAT	TECHNIC Bush
4	Black	6628.DAT	TECHNIC Friction Pin with Towball
1	Black	32270.DAT	TECHNIC Gear 12 Tooth Double Bevel
31	Blue	43093.DAT	TECHNIC Axle Pin with Friction
6	Light gray	6536.DAT	TECHNIC Axle Joiner Perpendicular
3	Black	32062.DAT	TECHNIC Axle 2 Notched
72	Black	2780.DAT	TECHNIC Pin with Friction and Slots
1	Light gray	4019.DAT	TECHNIC Gear 16 Tooth
2	Black	32192.DAT	TECHNIC Angle Connector #4 (135 degree)
2	Dark gray	6538B.DAT	TECHNIC Axle Joiner Offset
6	Black	45590.DAT	TECHNIC Axle Joiner Double Flexible
2	Black	75535.DAT	TECHNIC Pin Joiner Round
5	Dark gray	32291.DAT	TECHNIC Axle Joiner Perpendicular Double
1	Black	32557.DAT	TECHNIC Pin Joiner Dual Perpendicular
1	Light gray	32269.DAT	TECHNIC Gear 20 Tooth Double Bevel
10	Light gray	32556.DAT	TECHNIC Pin Long
2	Light gray	4185.DAT	TECHNIC Wedge Belt Wheel (replaces TECHNIC Gear 36 Tooth Double Bevel)
338 parts total (all included in NXT retail set)			
PARTS FOR AESTHETIC ADD-ONS			
2	Light gray	32123.DAT	TECHNIC Bush 1/2 Smooth
4	White	32278.DAT	TECHNIC Beam 15
2	Light gray	54087.DAT	Wheel 43.2 - 22 Without Pinholes
2	Black	6558.DAT	TECHNIC Pin Long with Friction and Slot
4	Light gray	4519.DAT	TECHNIC Axle 3
1	Black	X344.DAT	TECHNIC Gear 36 Tooth Double Bevel
2	Black	32039.DAT	TECHNIC Connector with Axlehole
8	Light gray	32556.DAT	TECHNIC Pin Long

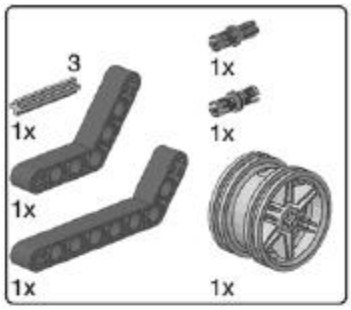


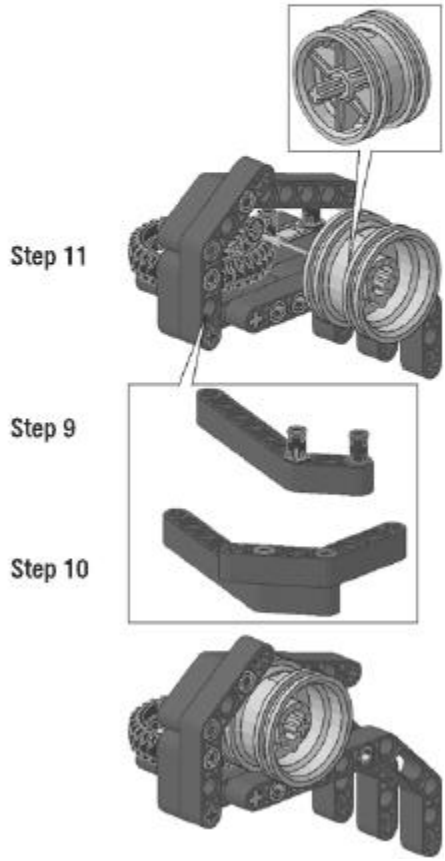
Start building the left hip.



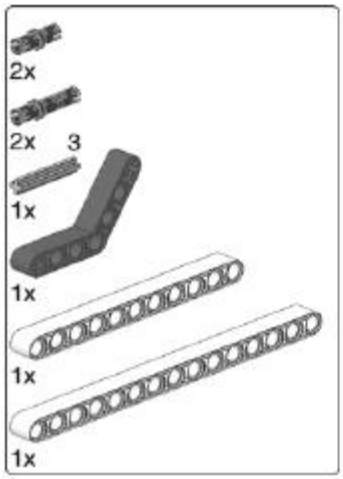


Skip Step 8 if you do not have two black Gears 36 Tooth Double Bevel. Do this to achieve symmetry. In fact, you can't mount the large decorative wheel in the other leg, because that black gear is replaced by two gray belt wheels.

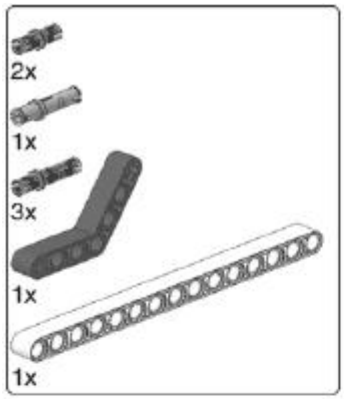


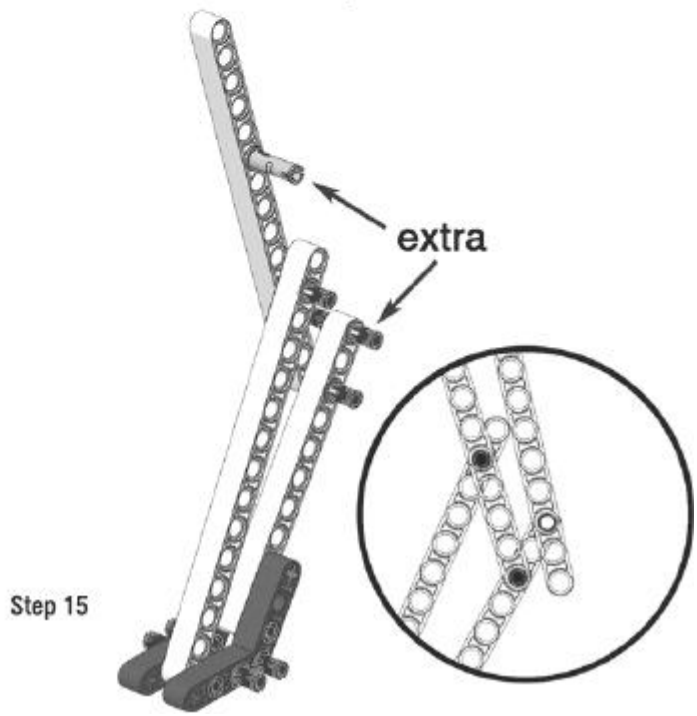


Build the decorative parts of the hip. If you don't have two black gears, don't attach the large decorative wheel for the same reason as before.

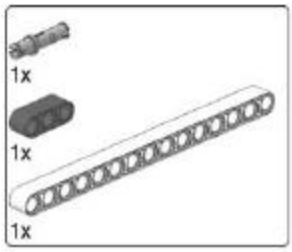


Start building the part of the leg common to both sides.

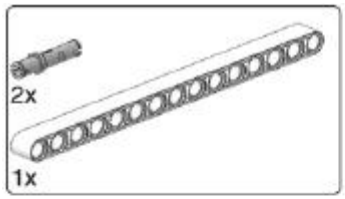




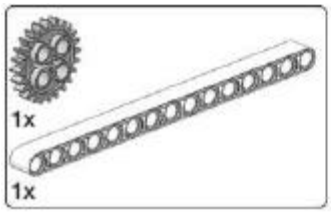
In Step 15, do not insert the marked pins. In the circle you can see the correct holes in which to attach the upper 15-long beam.



From here on, you build the decorative part of the leg. If you do not have extra parts, skip Step 16. In Step 17, add just the 15-long beam.



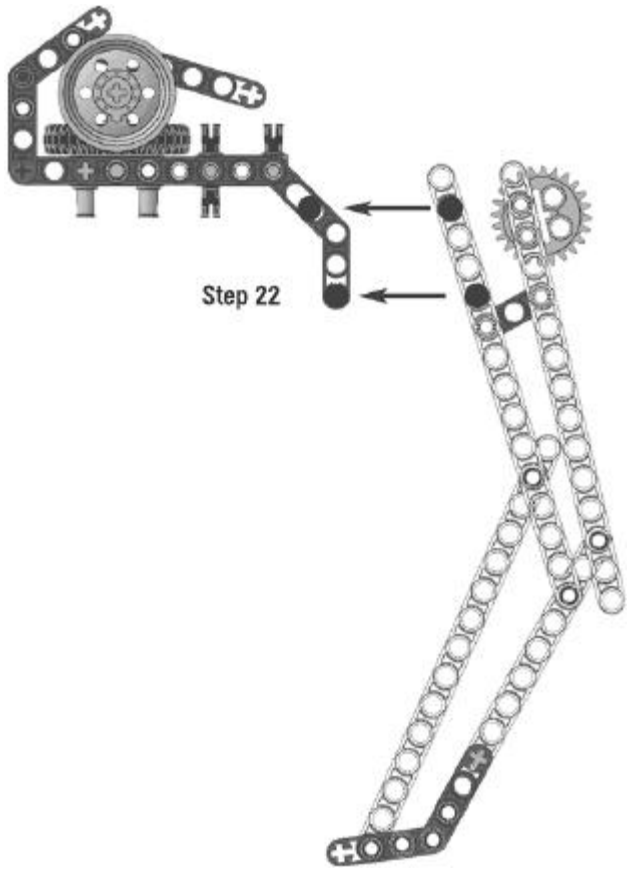
Continue skipping these steps if you don't have the extra parts.



Step 20

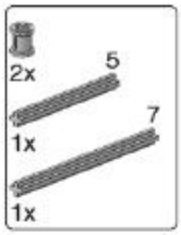
Step 21

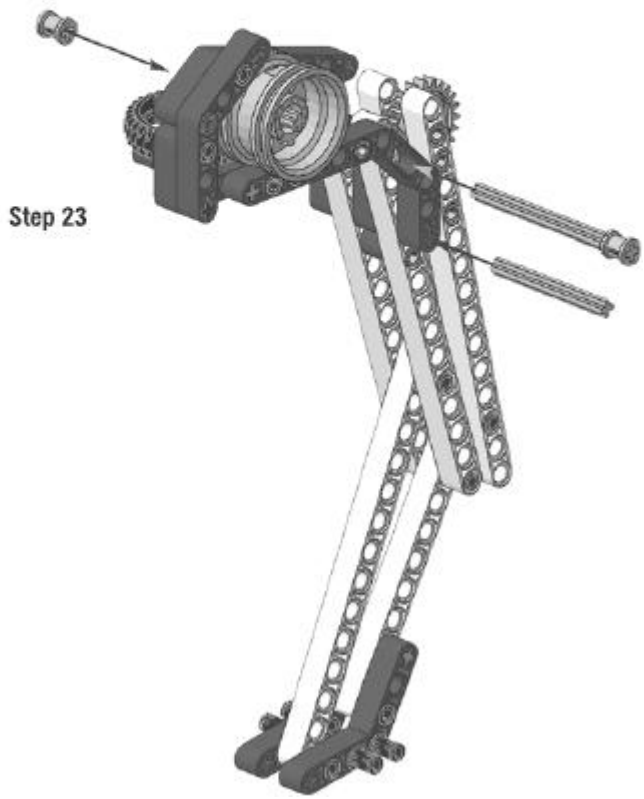
Continue skipping these steps if you don't have the extra parts. The leg is done.



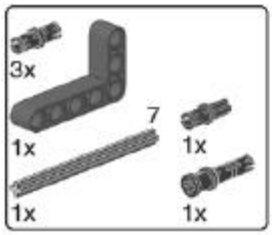
The black spots on the leg must meet the spots on the hip.

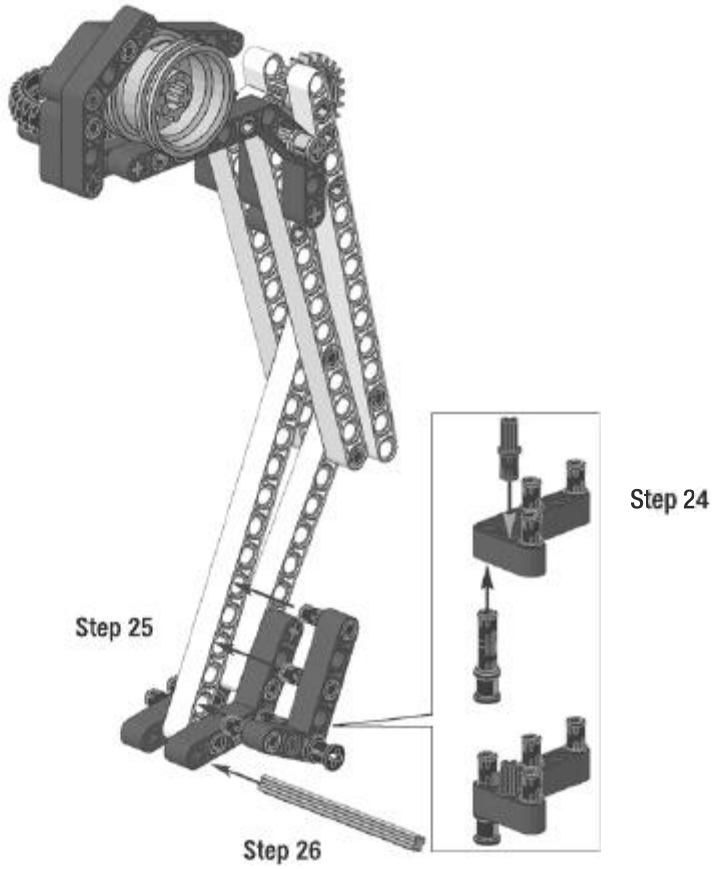
The leg beams must fit in the spaces between the three dark gray bent beams.



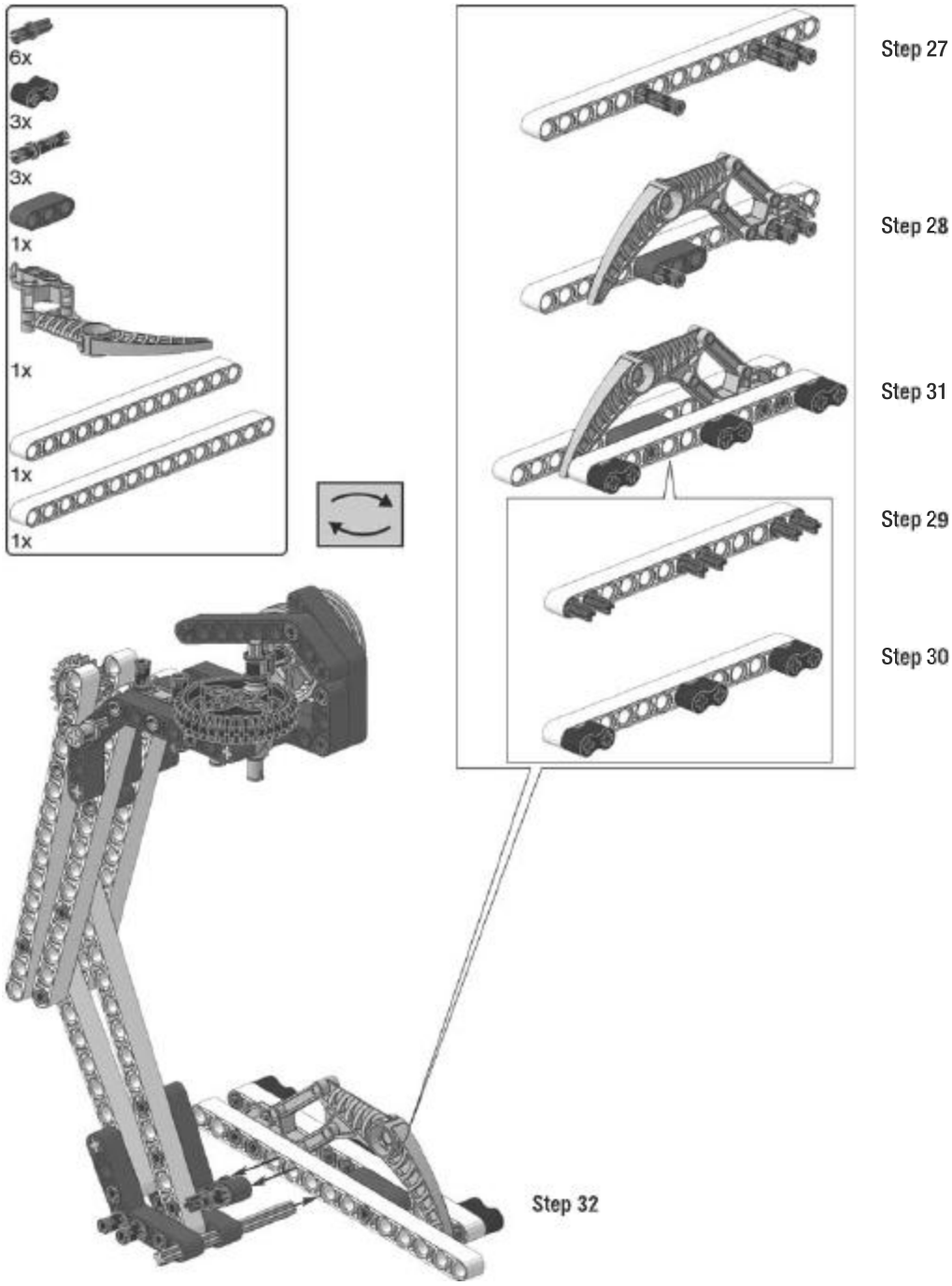


This picture shows how the leg should fit in the hip assembly. Insert the axles to hold the leg in place.

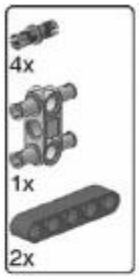


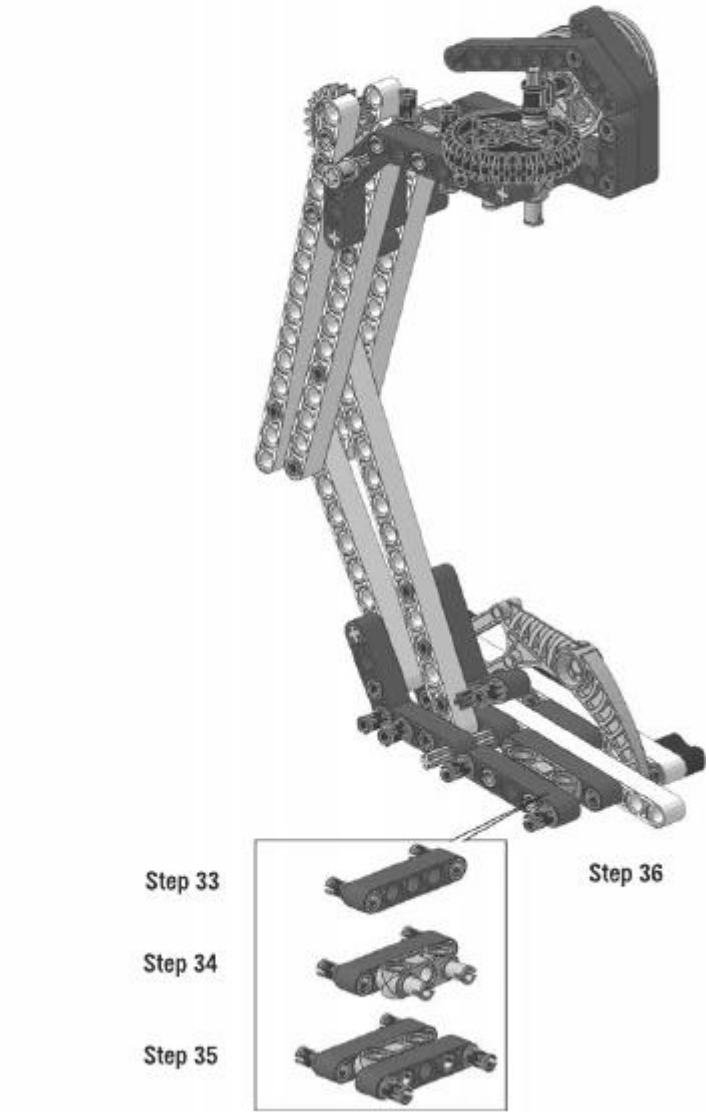


Build the reinforcer that prevents the ankle from bending to the outside too much during stepping. Insert the 7-long axle at the end of the leg.

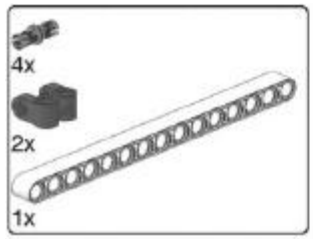


Rotate the assembly and build the external foot.

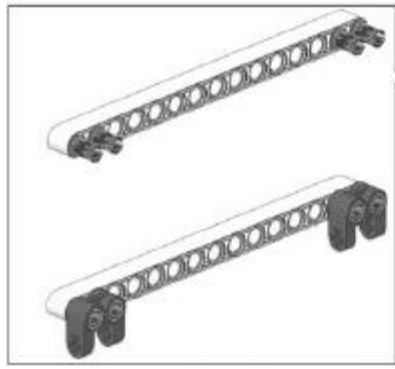




Insert the foot pad.



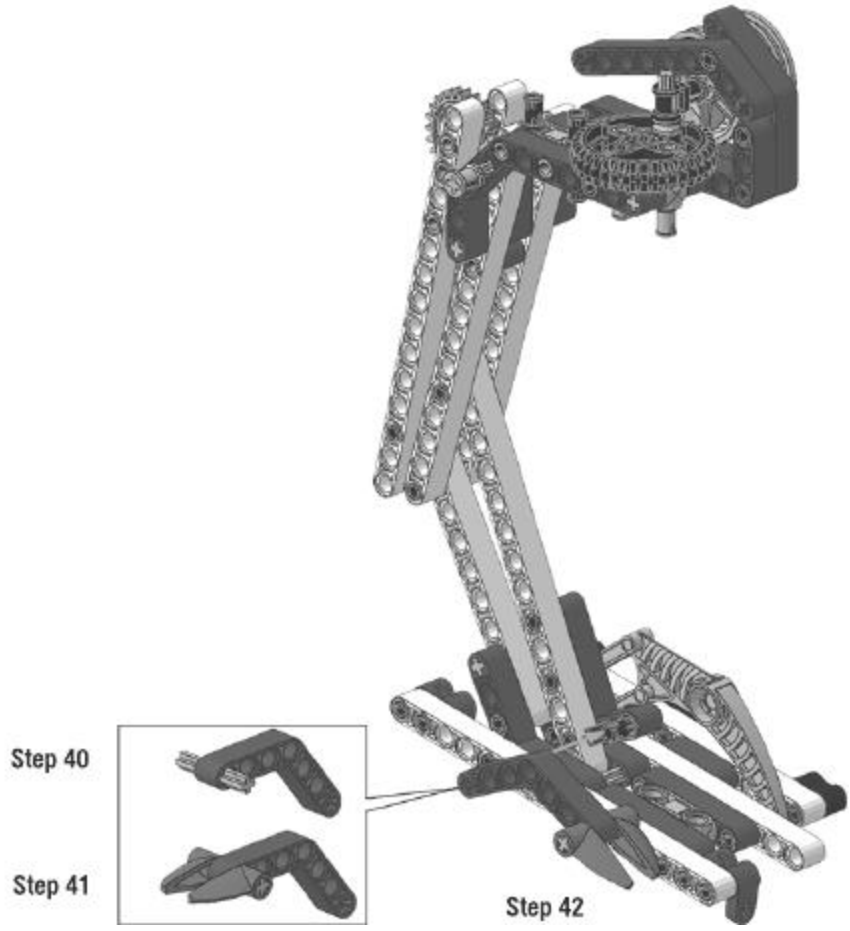
Step 39



Step 37

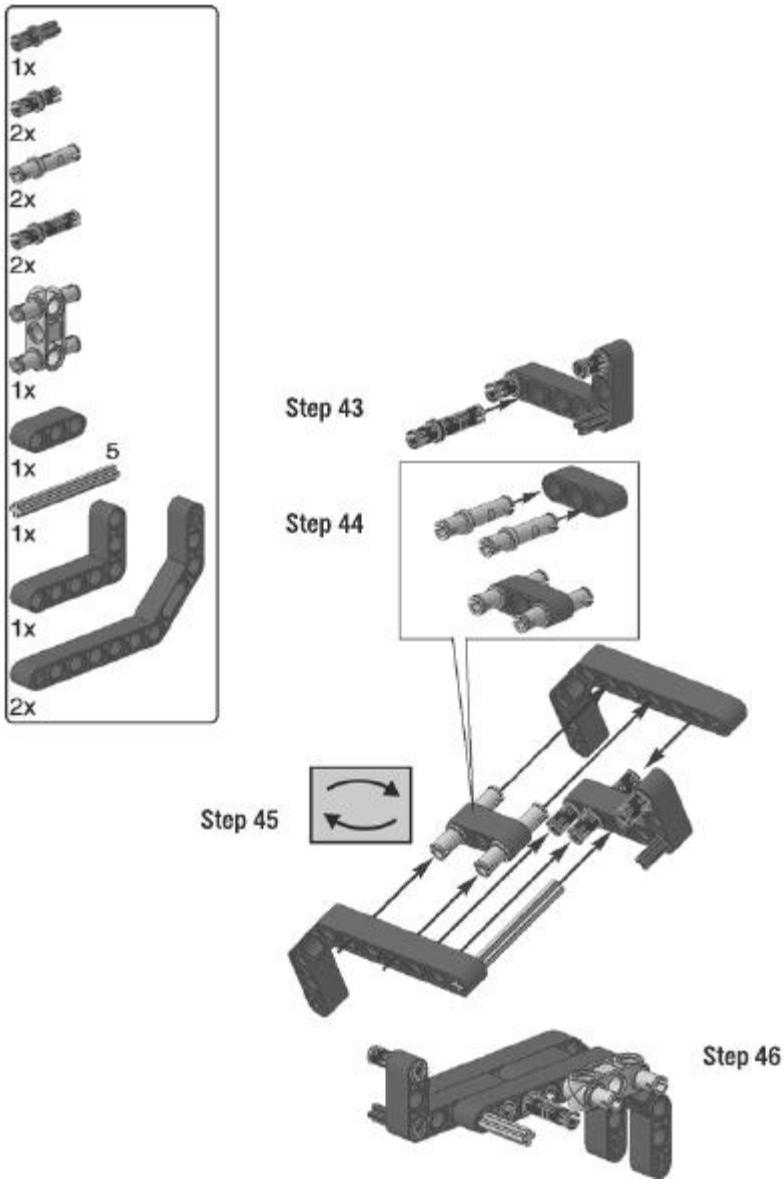
Step 38

Build the internal side of the foot with wedges.

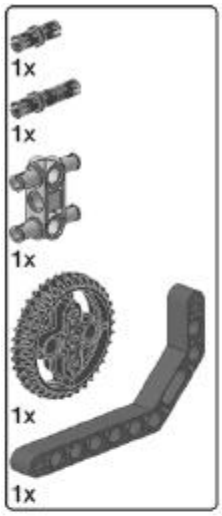


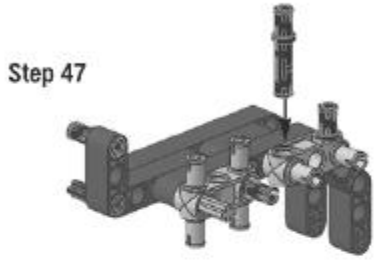
Attach the foot blades.

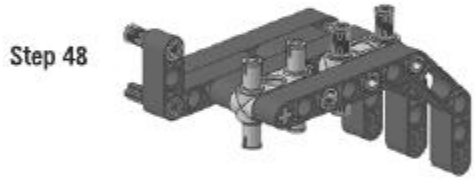
The left leg is completed.



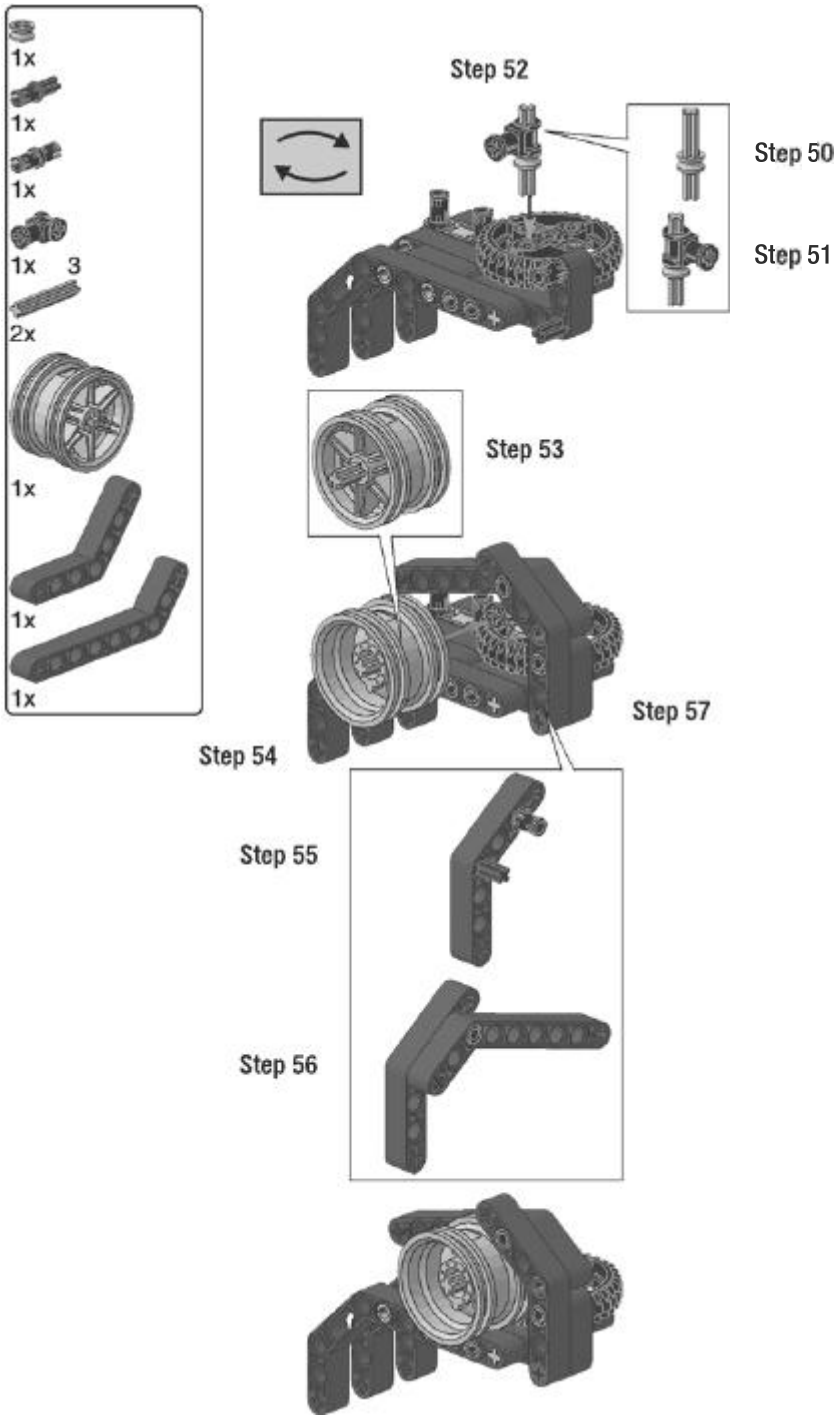
Start building the right hip.



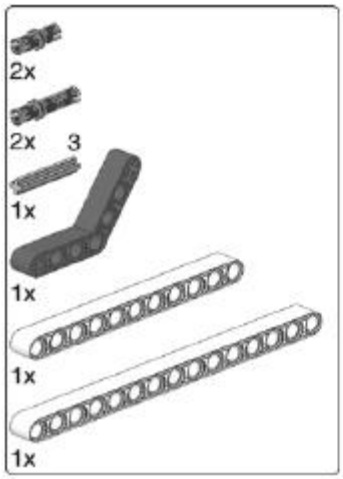




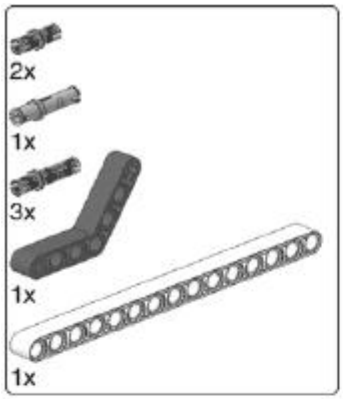
In Step 49, if you don't have the extra black gear, replace the black gear with two gray wheels, checking their position in the figure on page 111.

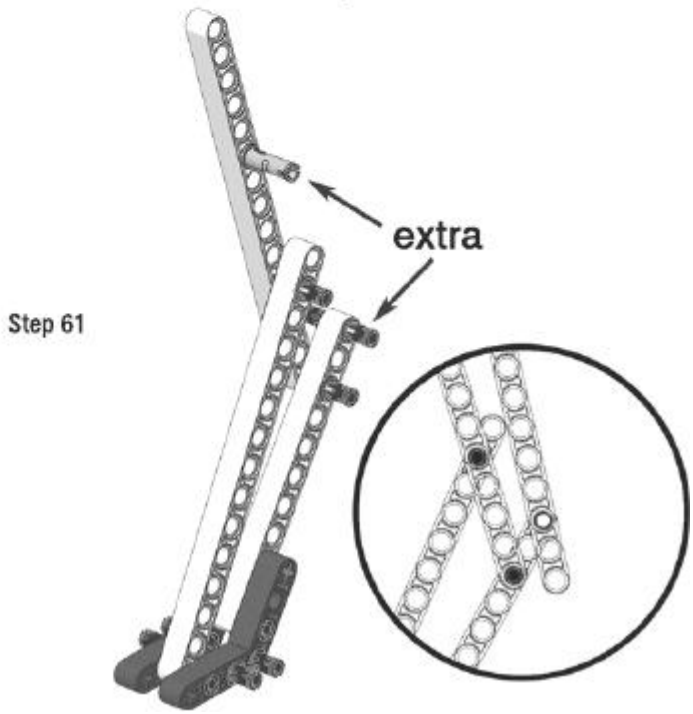


Build the decorative parts of the hip. Skip Steps 50 to 54 if you replaced the black gear with two gray wheels in Step 49.

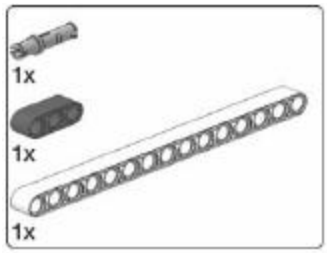


Start building the part of the leg common to both sides.

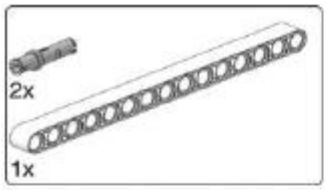




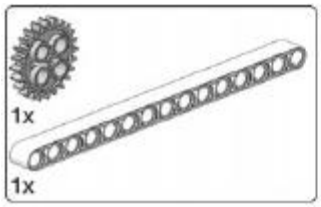
In Step 61, do not insert the marked pins. In the circle you can see the correct holes where you can attach the upper 15-long beam.



From now on, you'll build the decorative part of the leg. If you don't have the extra parts, skip Step 62, and in Step 64, add just the 15-long beam.

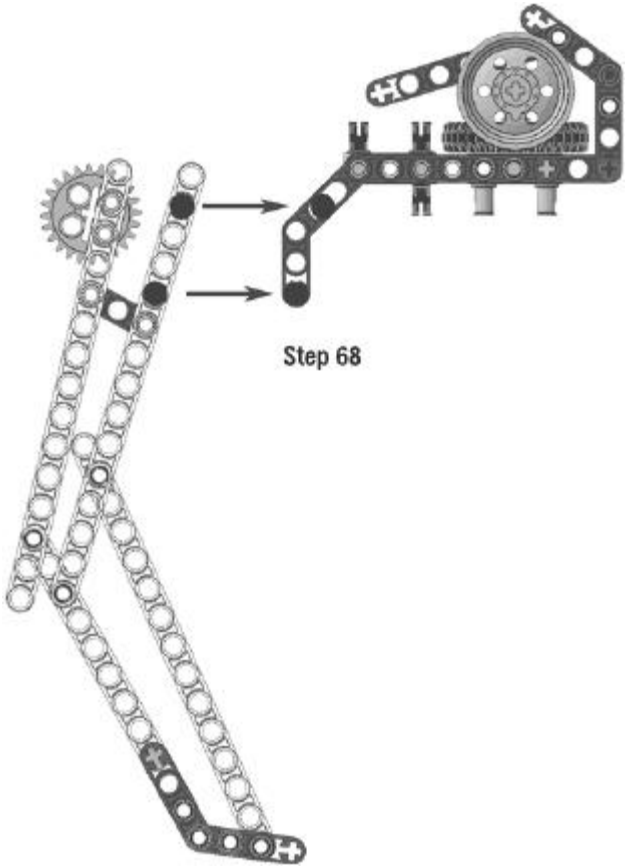


Continue skipping these steps if you don't have the extra parts.

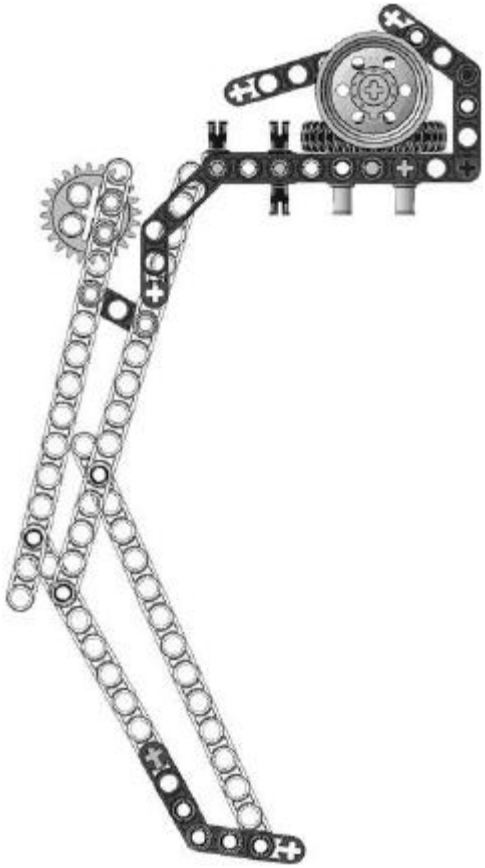


Step 67

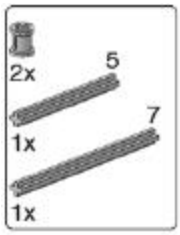
Continue skipping these steps if you don't have the extra parts. The leg is done.

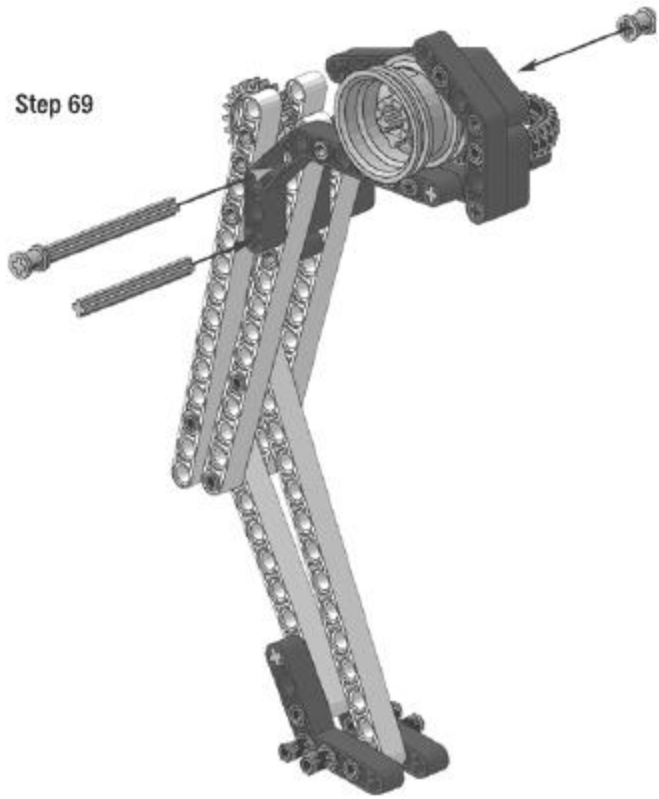


The black spots on the leg must meet the spots on the hip.

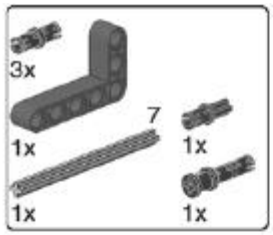


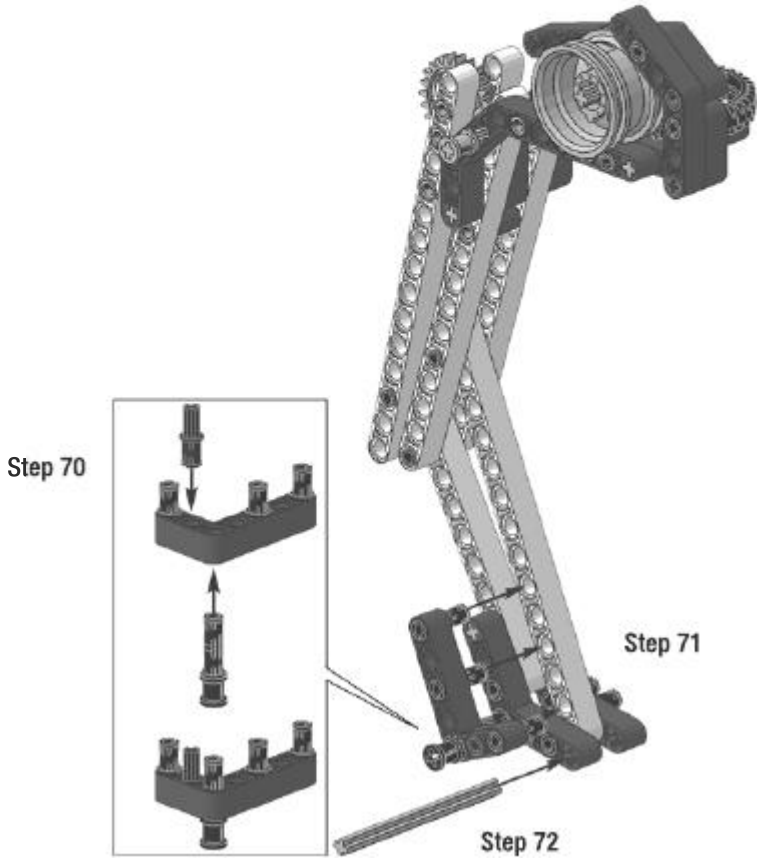
The leg beams must fit in the two spaces between the three dark gray bent beams.



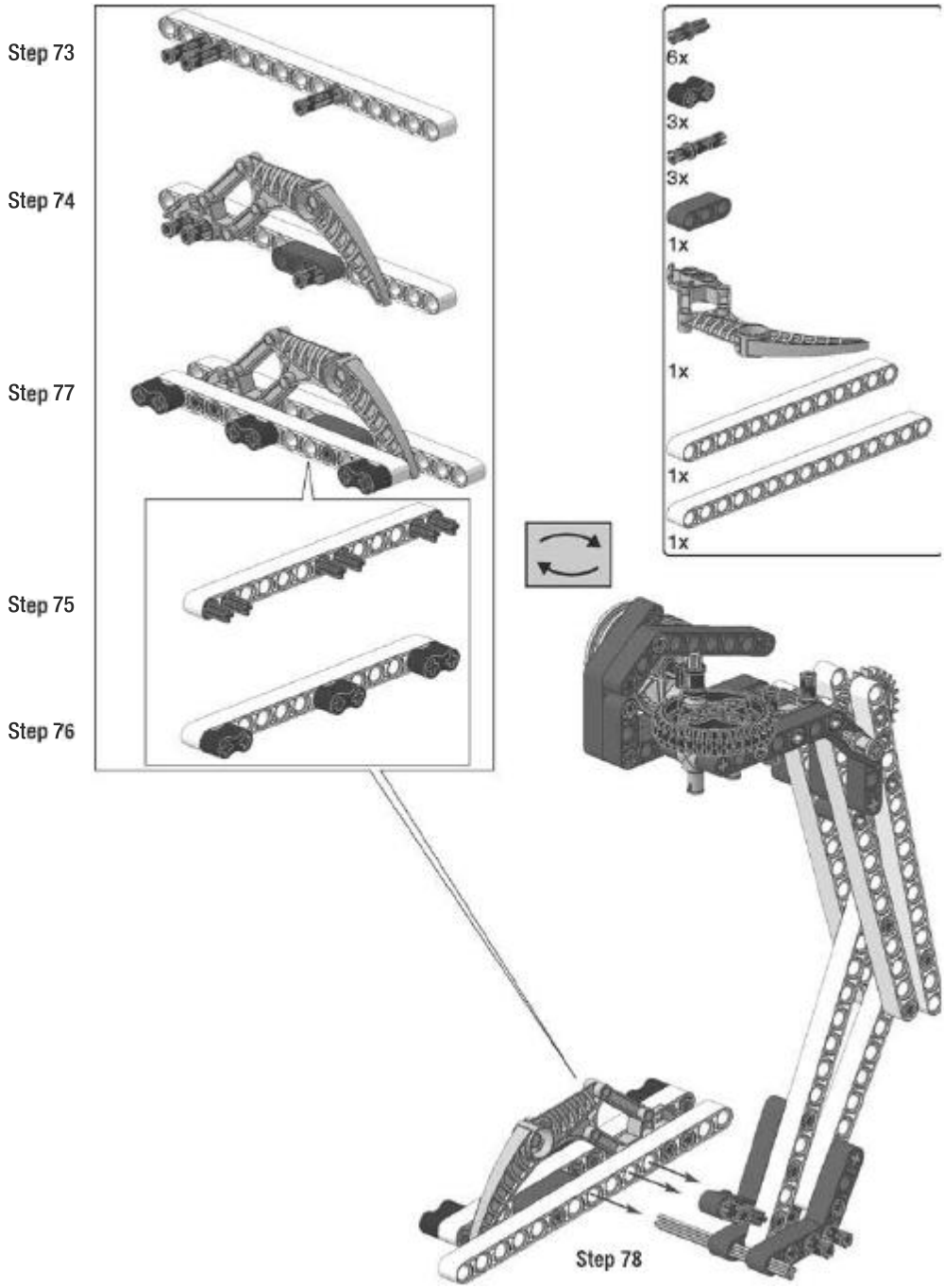


This picture shows how the leg should fit in the hip assembly. Insert the axles to hold the leg in place.

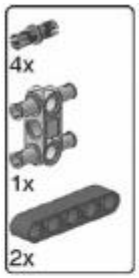




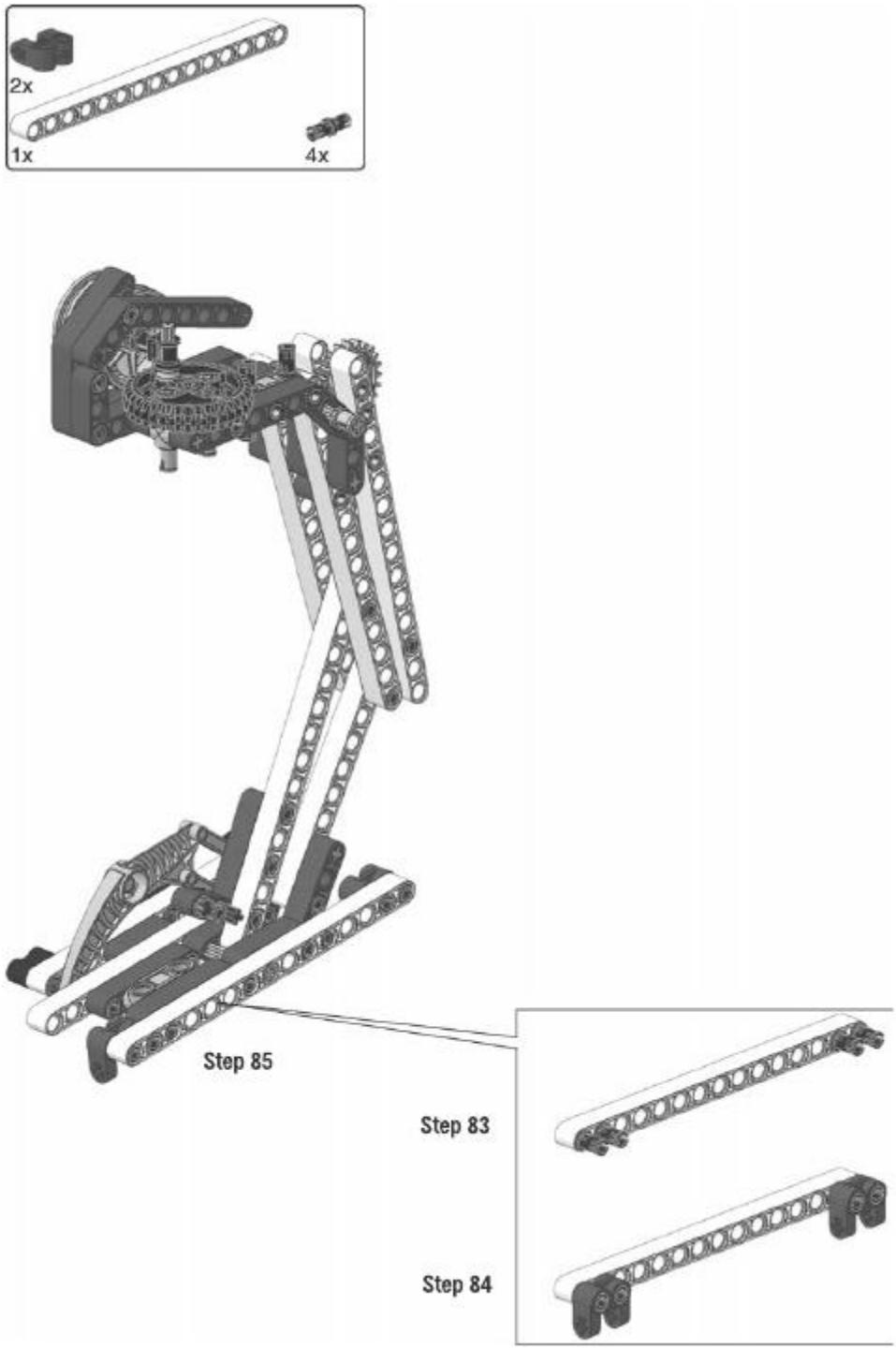
Build the reinforcer, which prevents the ankle from bending too much to the outside during stepping. Insert the 7-long axle at the end of the leg.



Rotate the assembly and build the external foot.



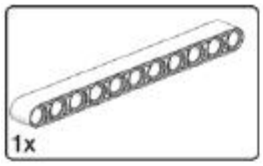
Insert the foot pad.

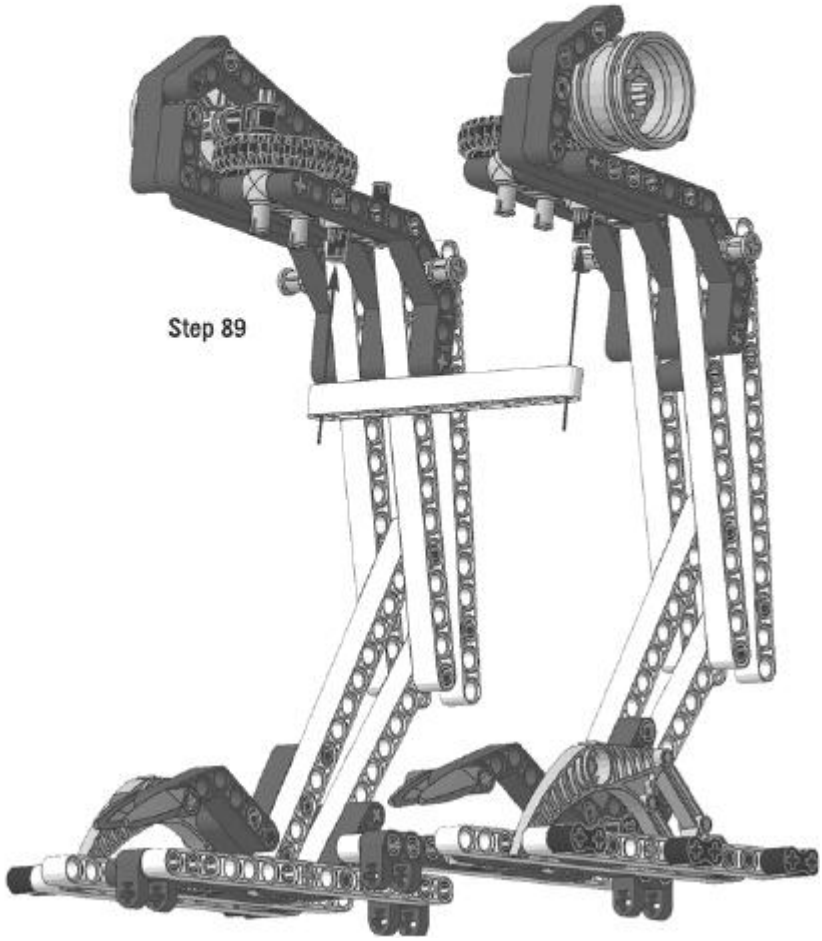


Build the internal side of the foot with wedges.

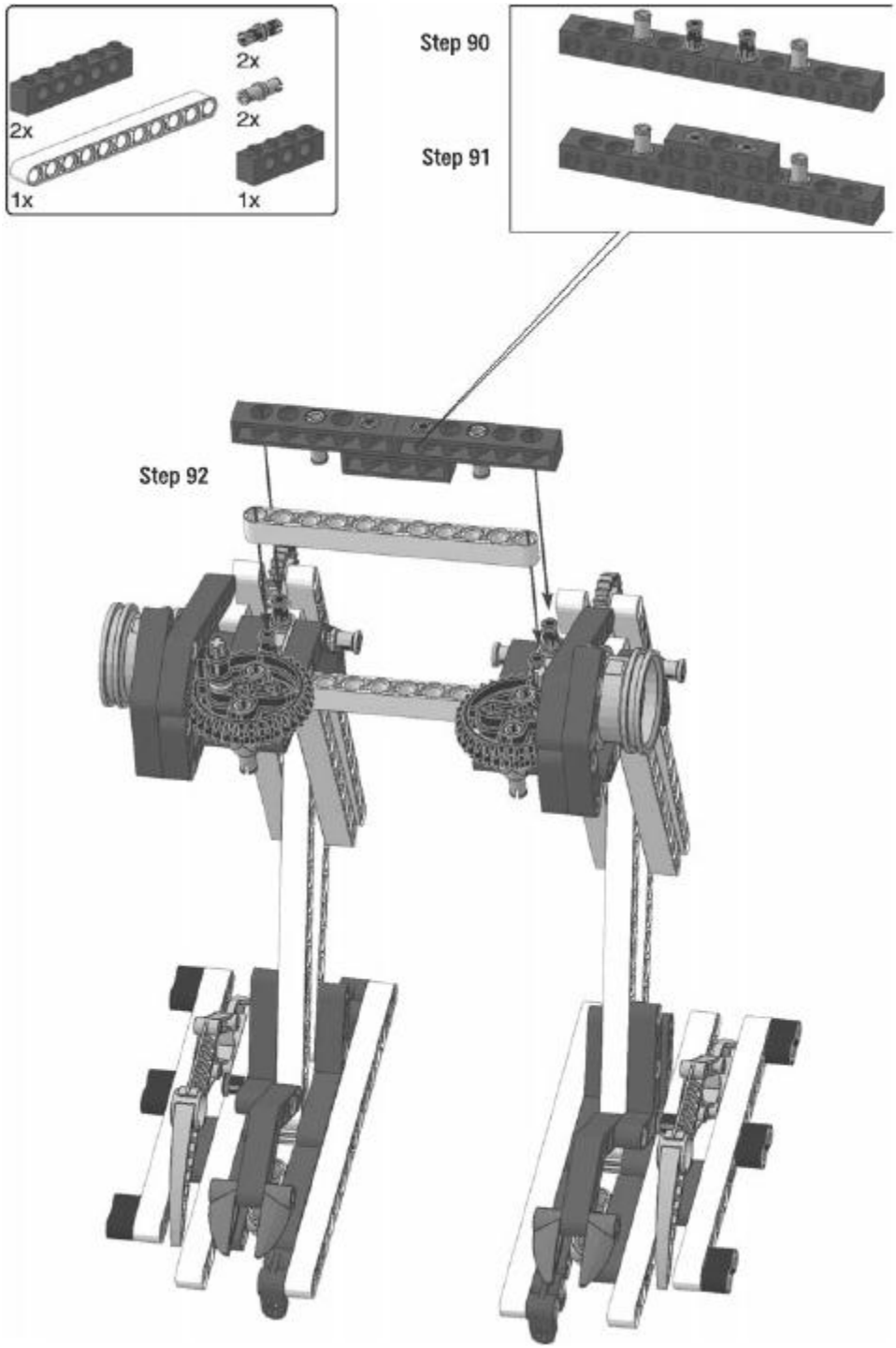
Attach the foot blades.

The right leg is completed.

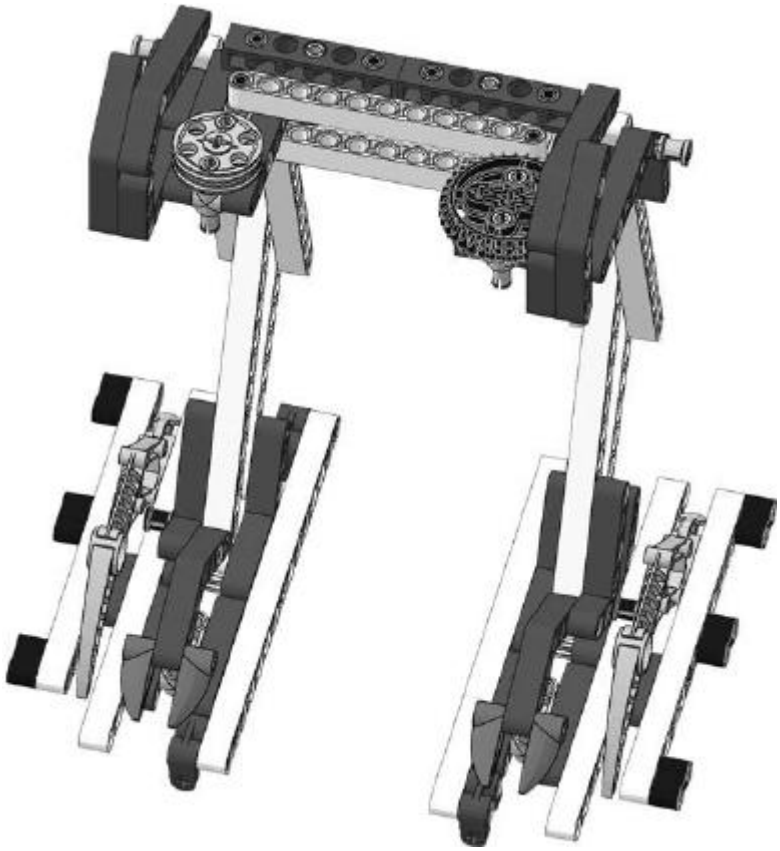




Join the completed legs with an 11–long beam.

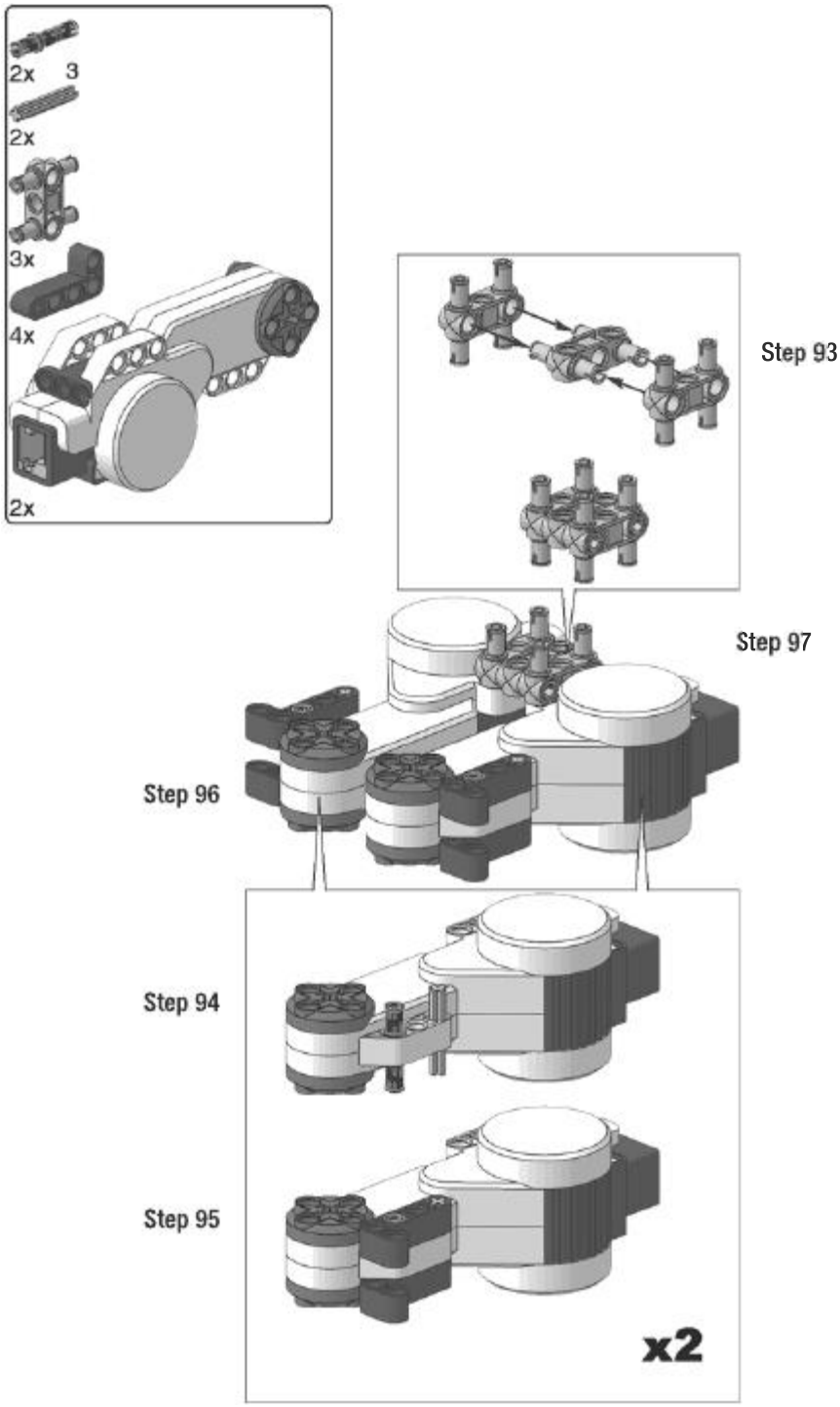


To make the legs move parallel to each other, insert another 11-long beam and the brick assembly.

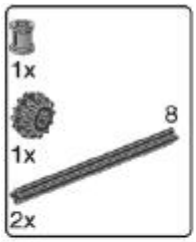


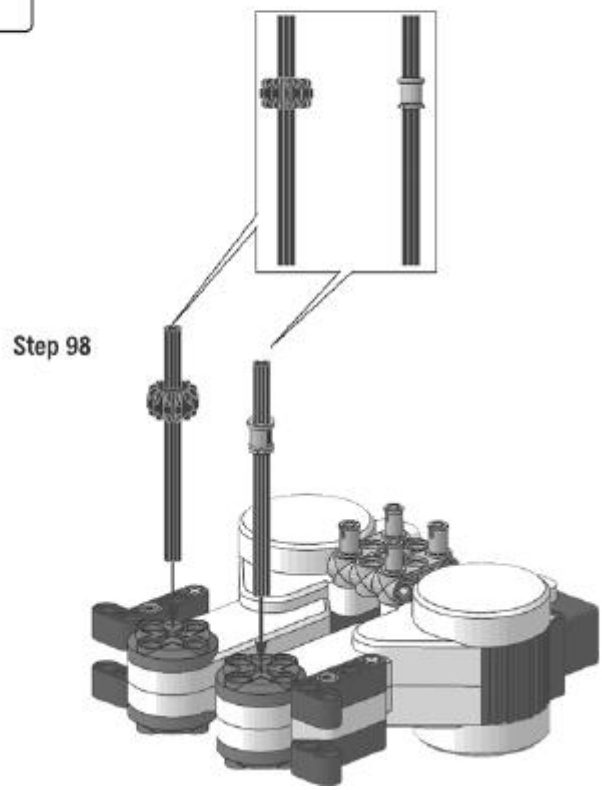
This picture shows how the AT-ST looks when assembled with retail set parts only. Notice the two gray wheels on the right hip assembly and the black gear on the left hip.

This picture shows the AT-ST legs without the additional 15-long beams. You can remove those beams safely, because they aren't structural.

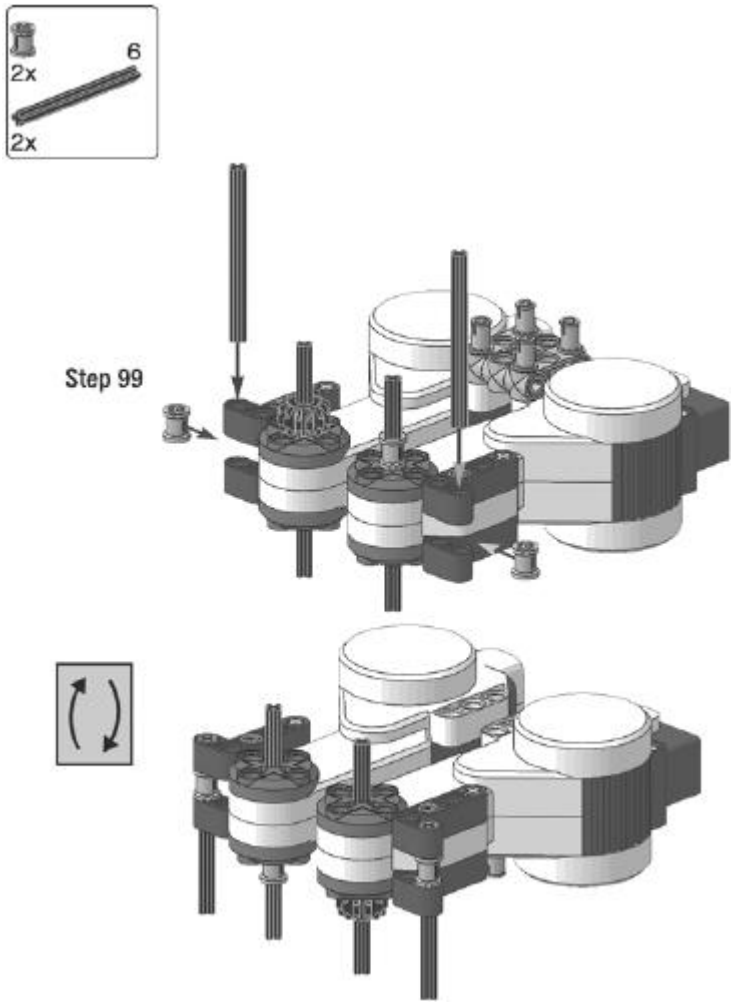


Now build the subassembly for the motors.

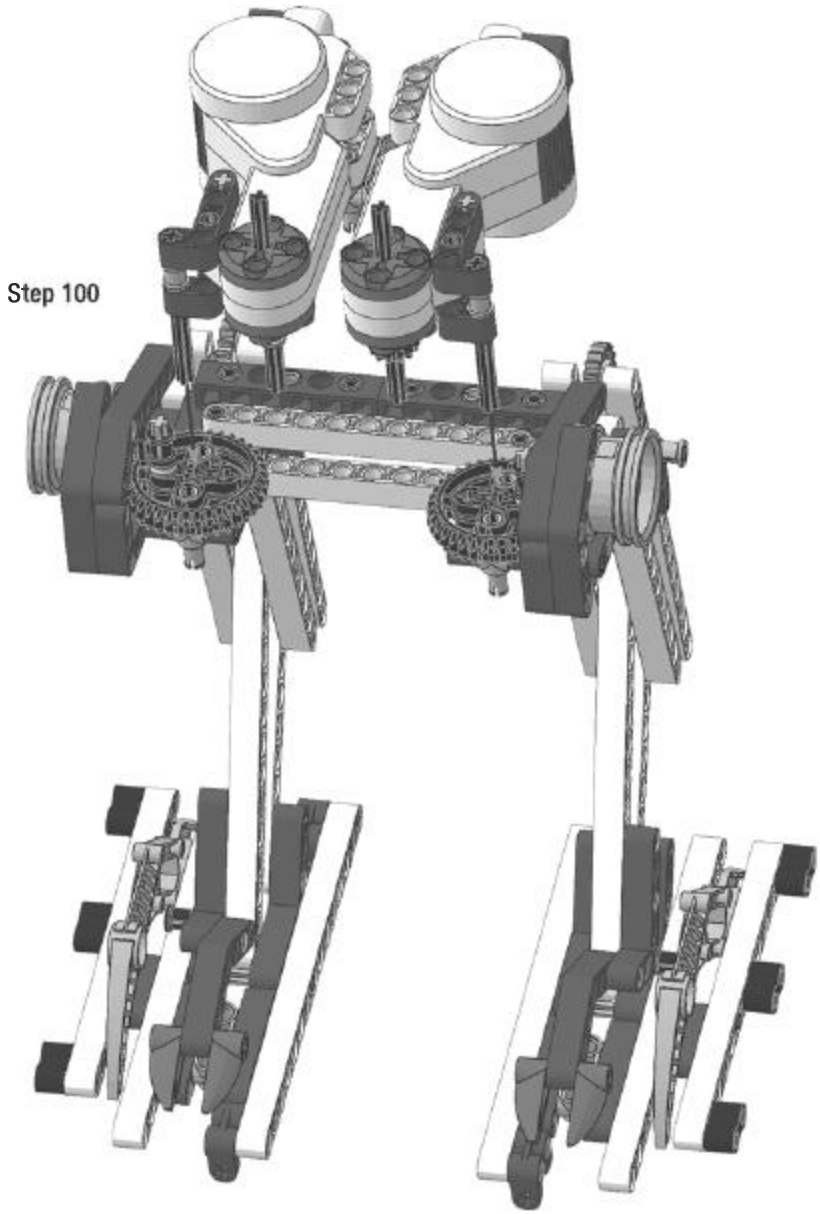




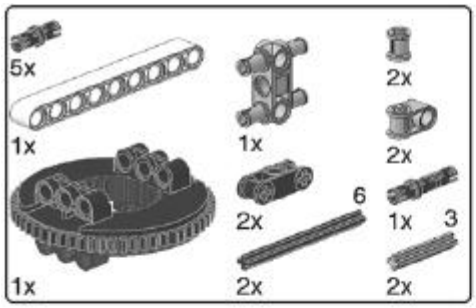
Insert two 8-long axles in the motor shafts. The callout shows where to place the bush and the gear.

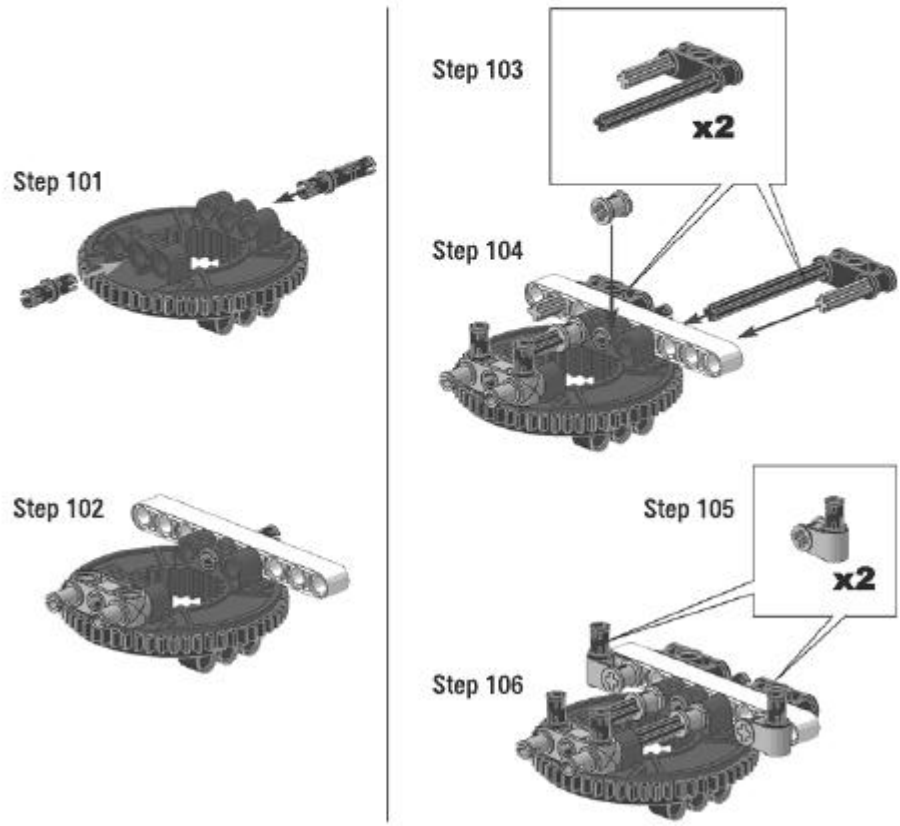


Insert the 6-long axles, fixing them with the bushes. Rotate the model and check if you inserted the axles correctly.

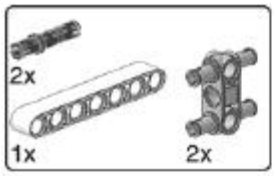


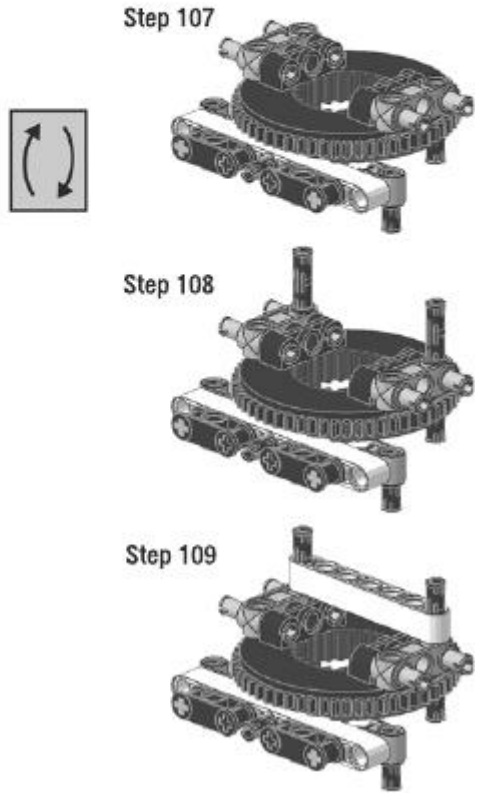
Attach the motors' subassembly on the legs. Insert the 6-long axles in the black gear's central hole (left leg) and in the gray wheel's central hole (right leg). However, if you have two black gears, the model looks as in this picture.



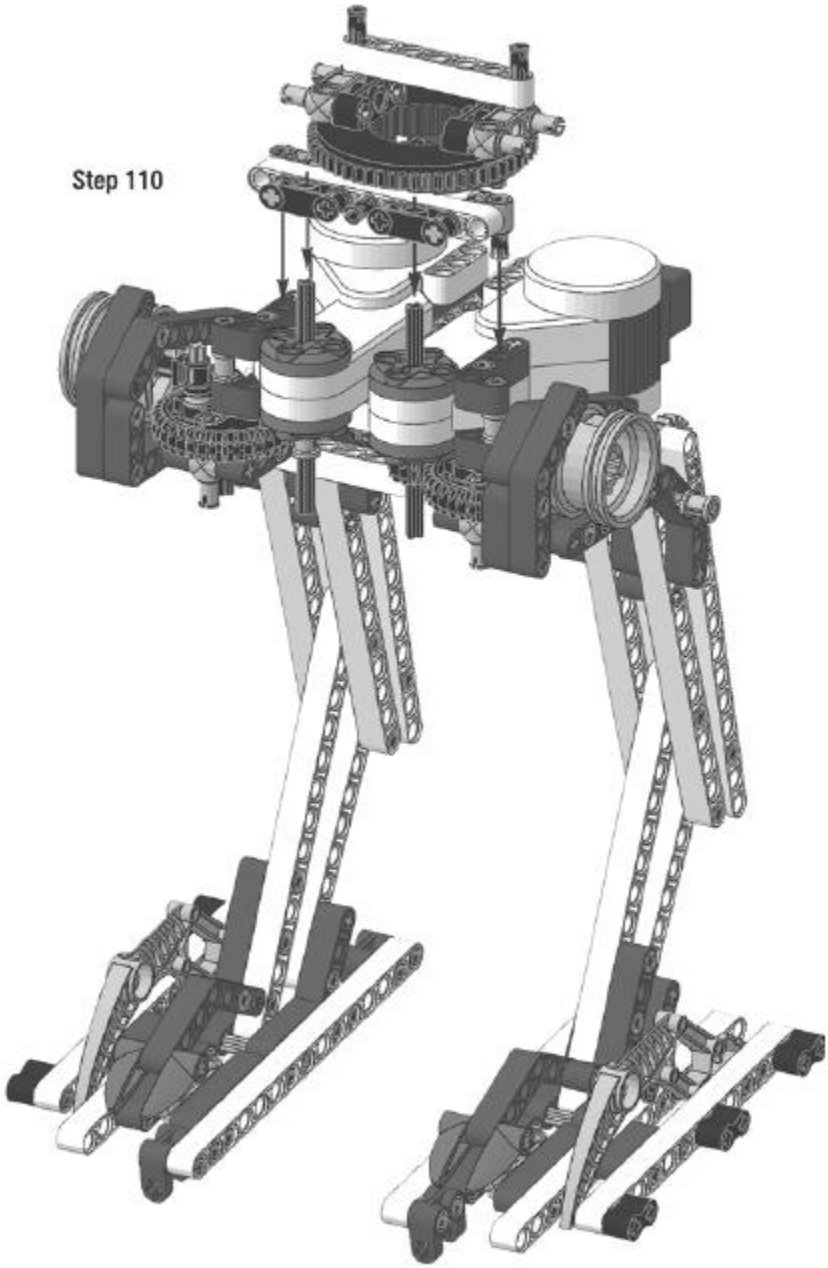


Build the rotating neck.

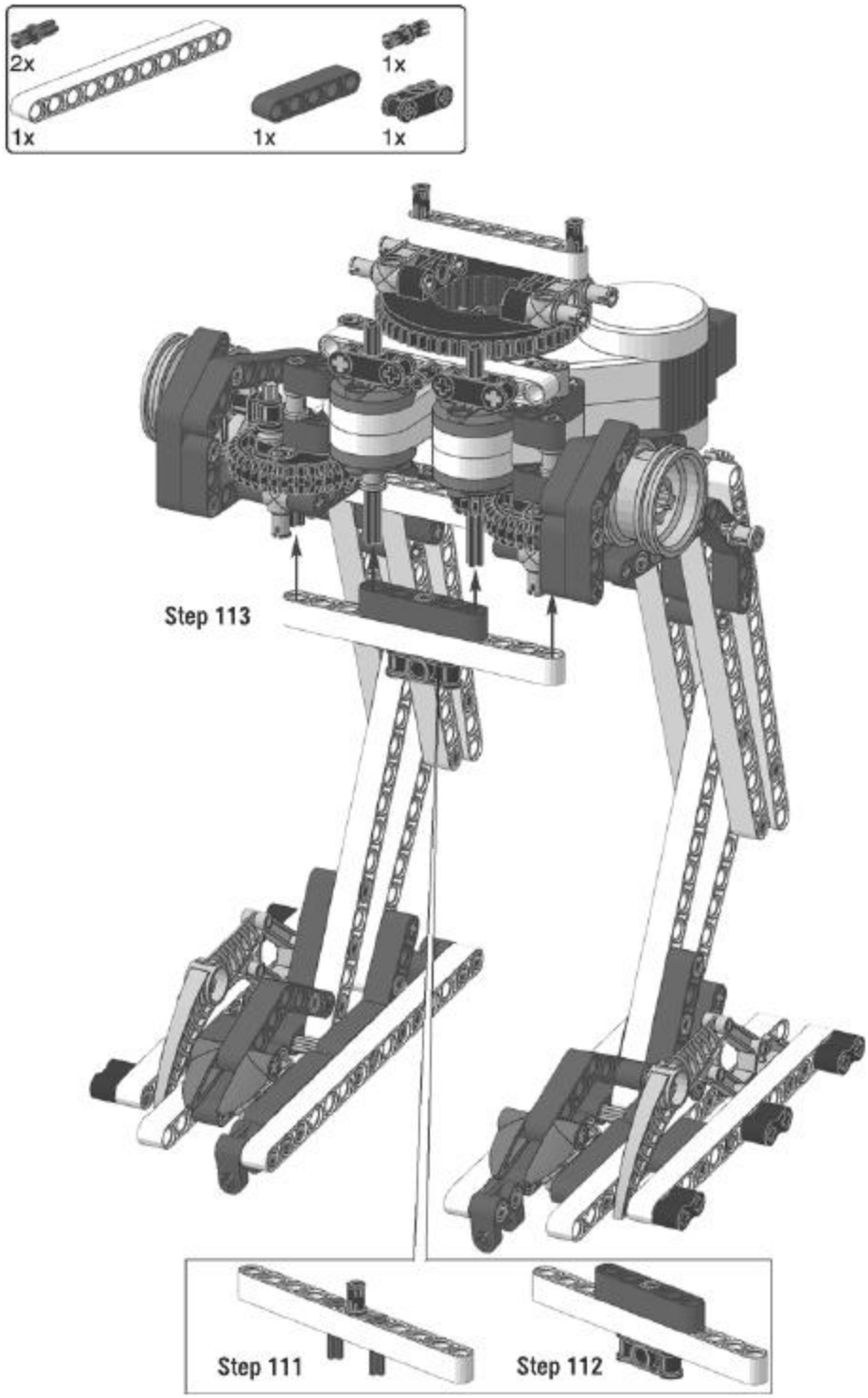




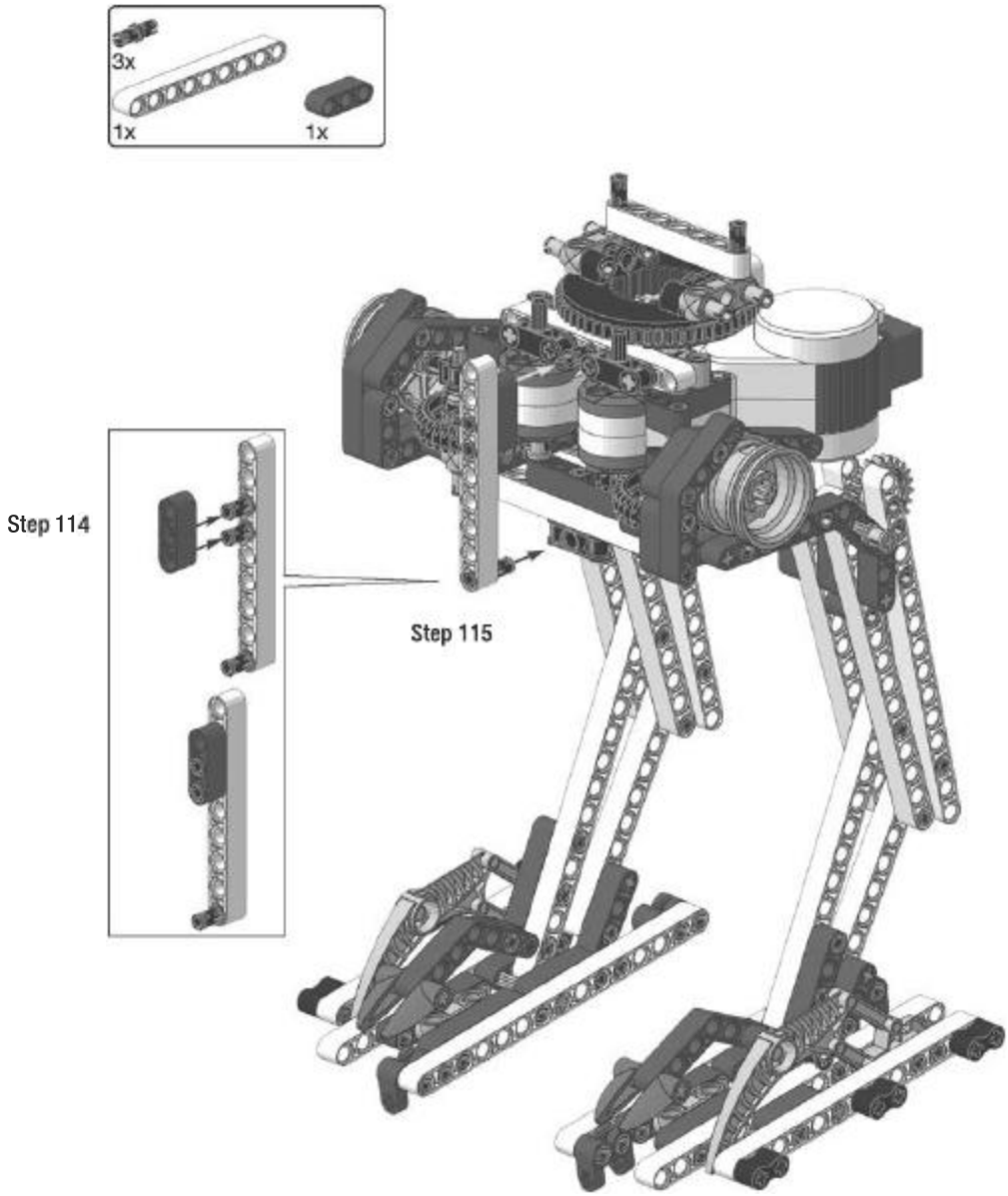
Rotate the model upside down and finish the neck assembly.



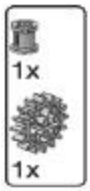
This graphic shows the motors attached to the legs, and also how to insert the neck in place.

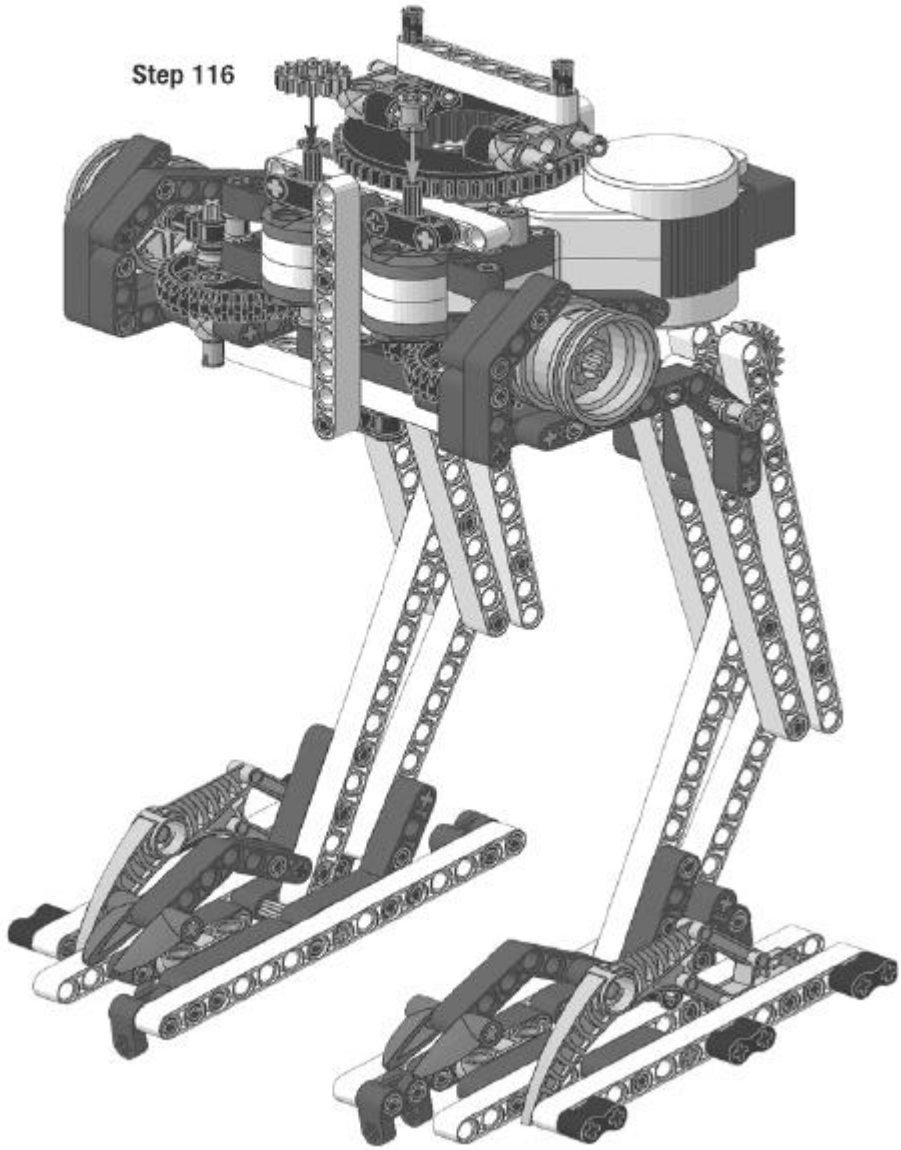


Build the beam that will hold the legs firmly.

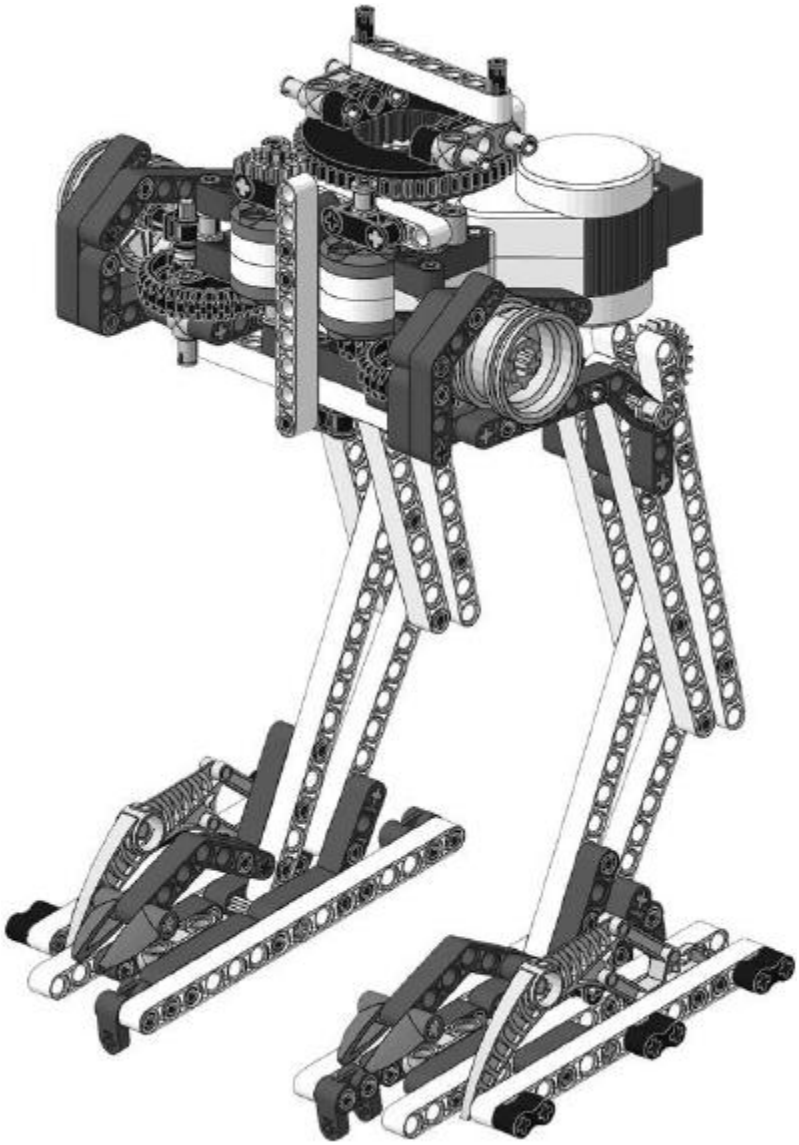


Insert the cross-bracing beam that holds the legs. Notice that you can't detach the legs from the AT-ST if this beam is in place.

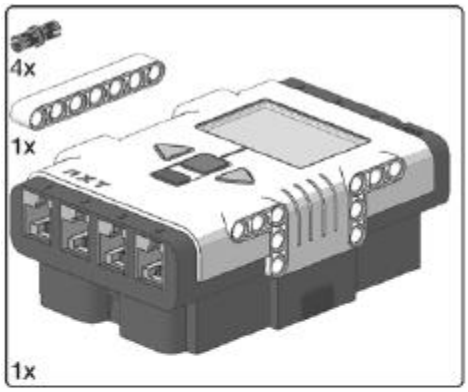




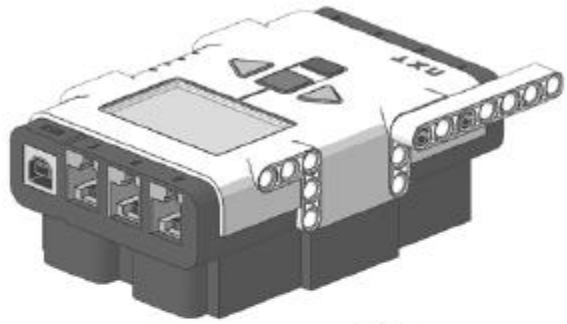
Insert the 16-tooth gear and the bush in their places.



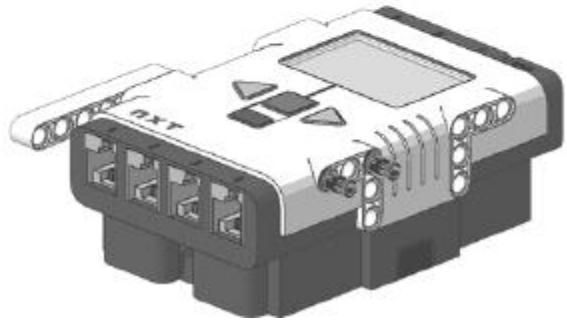
Notice that the right motor gear (on top) engages the neck turntable, while the left motor 12-tooth gear (on bottom) engages the left leg's black gear.



Step 117

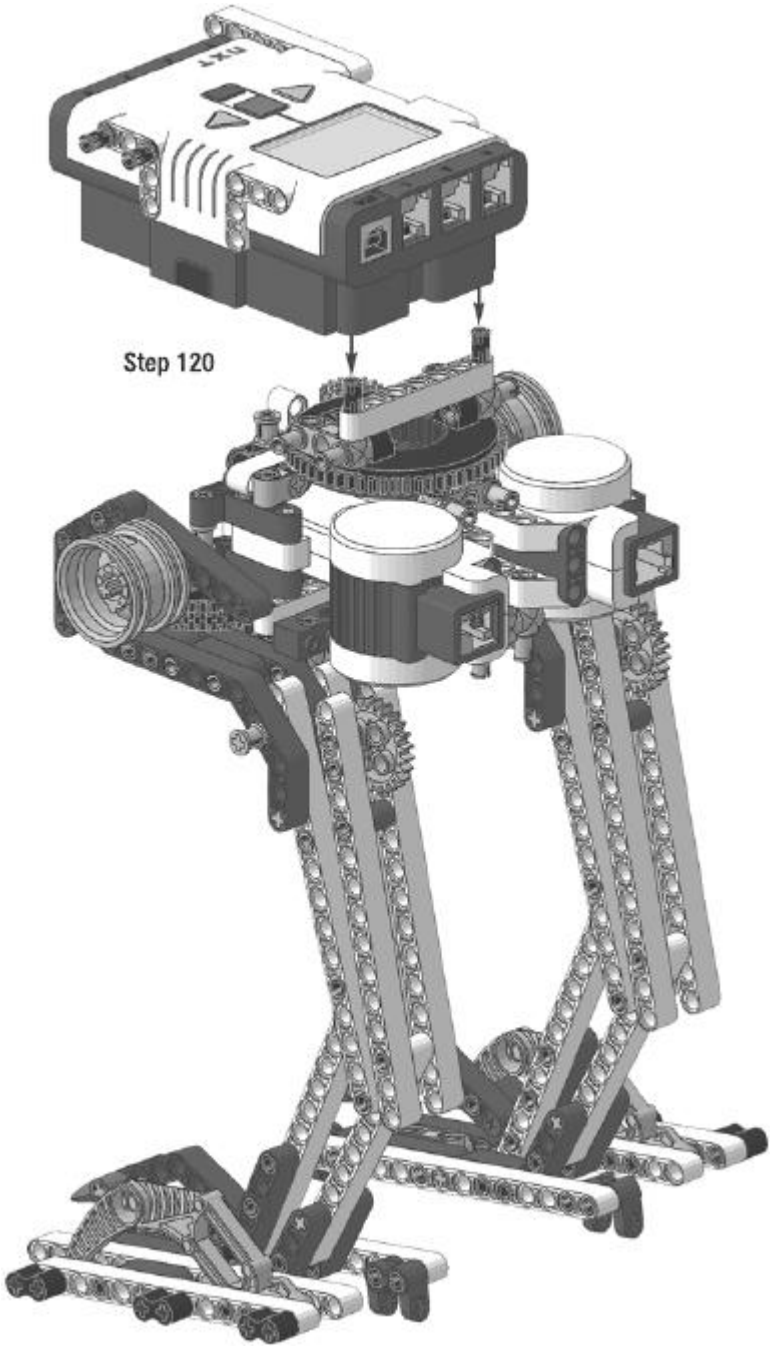


Step 118

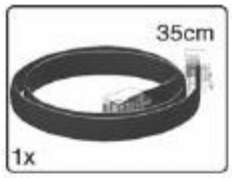


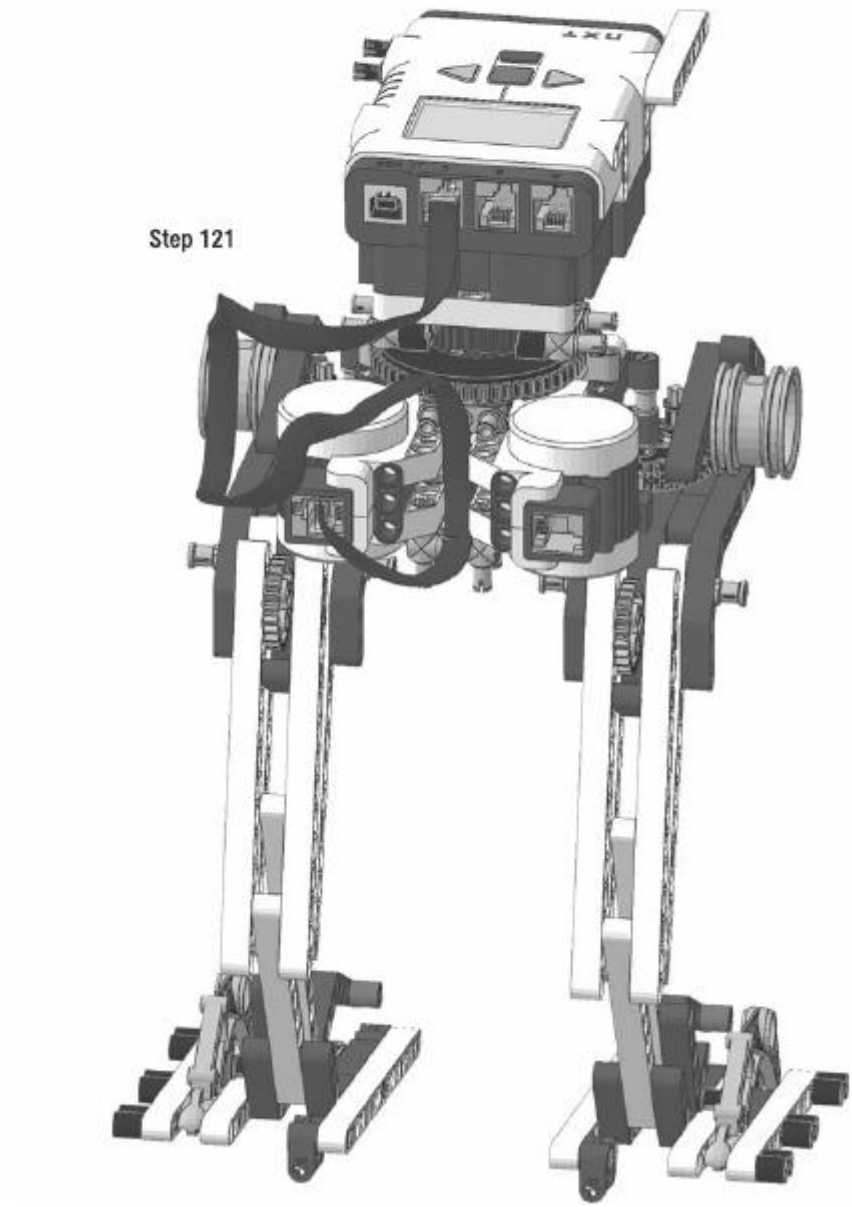
Step 119

The NXT is used as the AT-ST head.

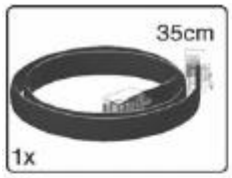


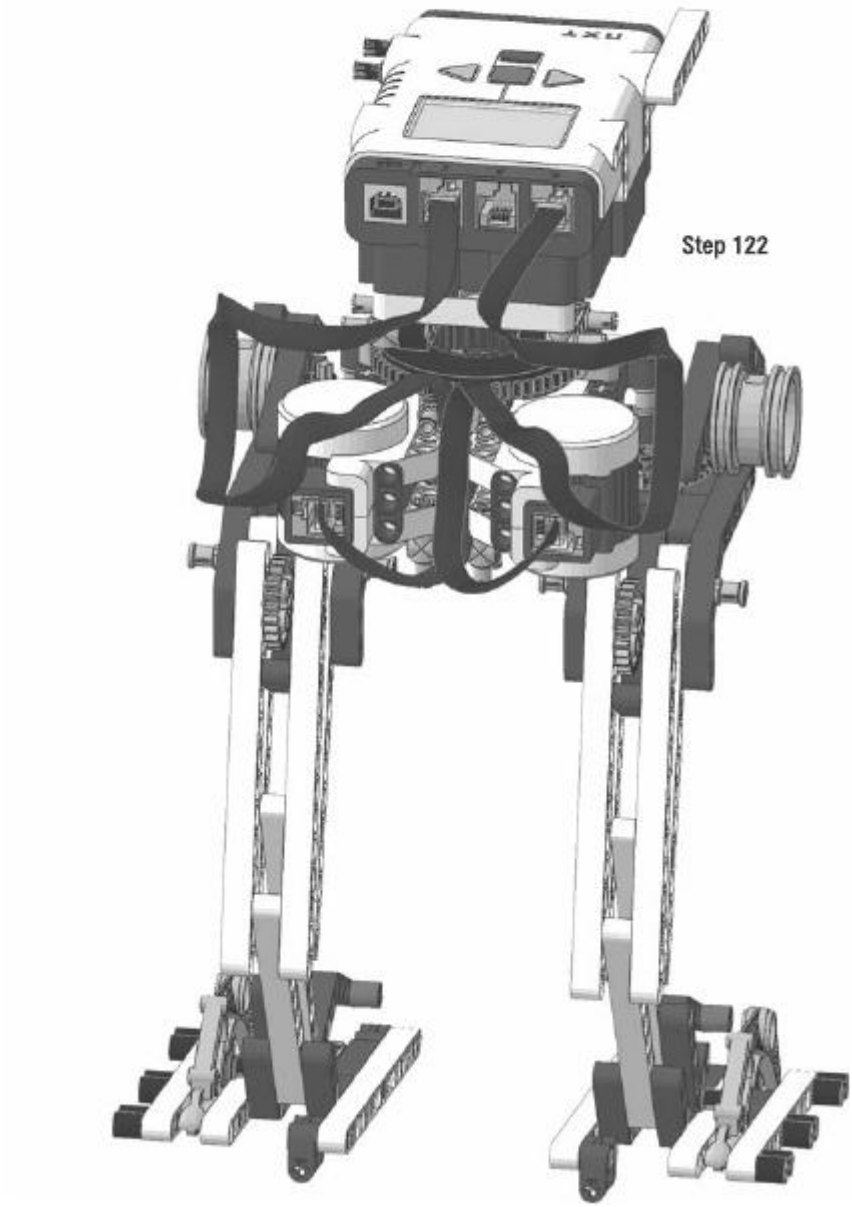
Attach the head to the neck. Be careful; the NXT is not secured to the neck yet.



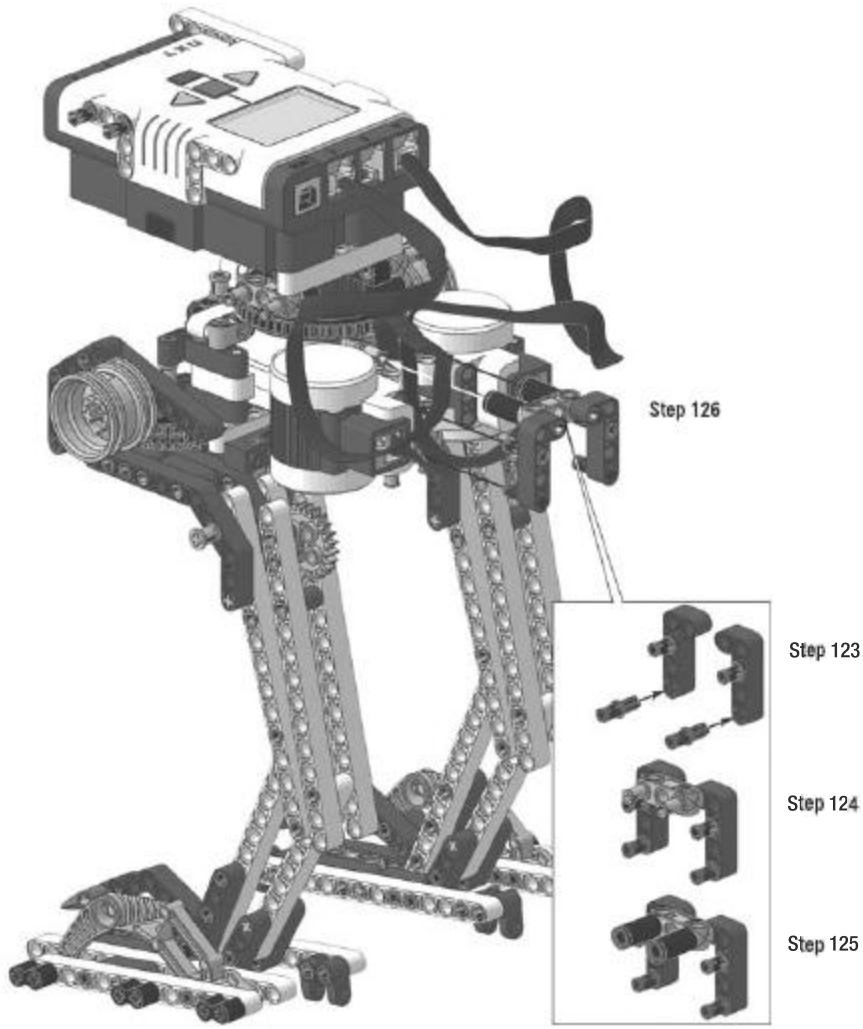


Connect the left motor to NXT output port C using a 35cm (14 inch) cable.

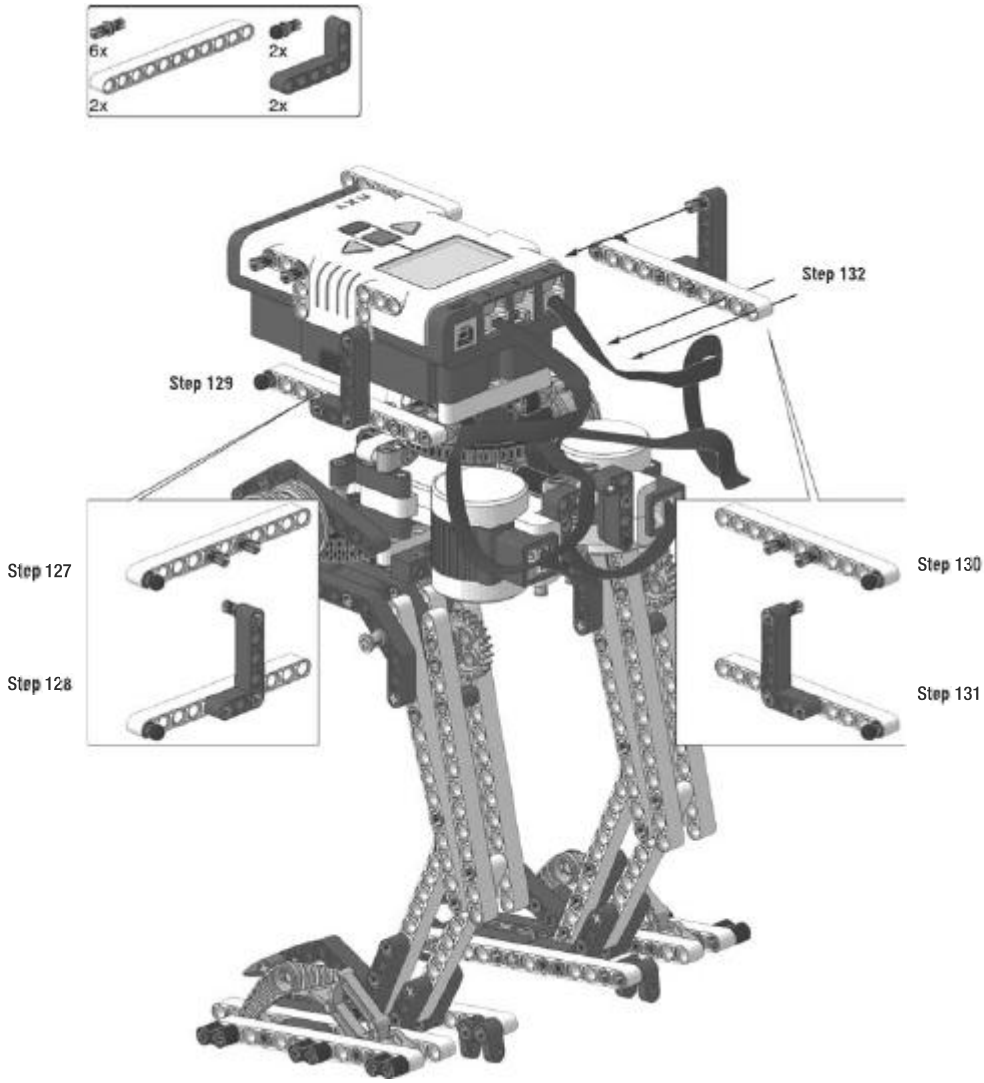




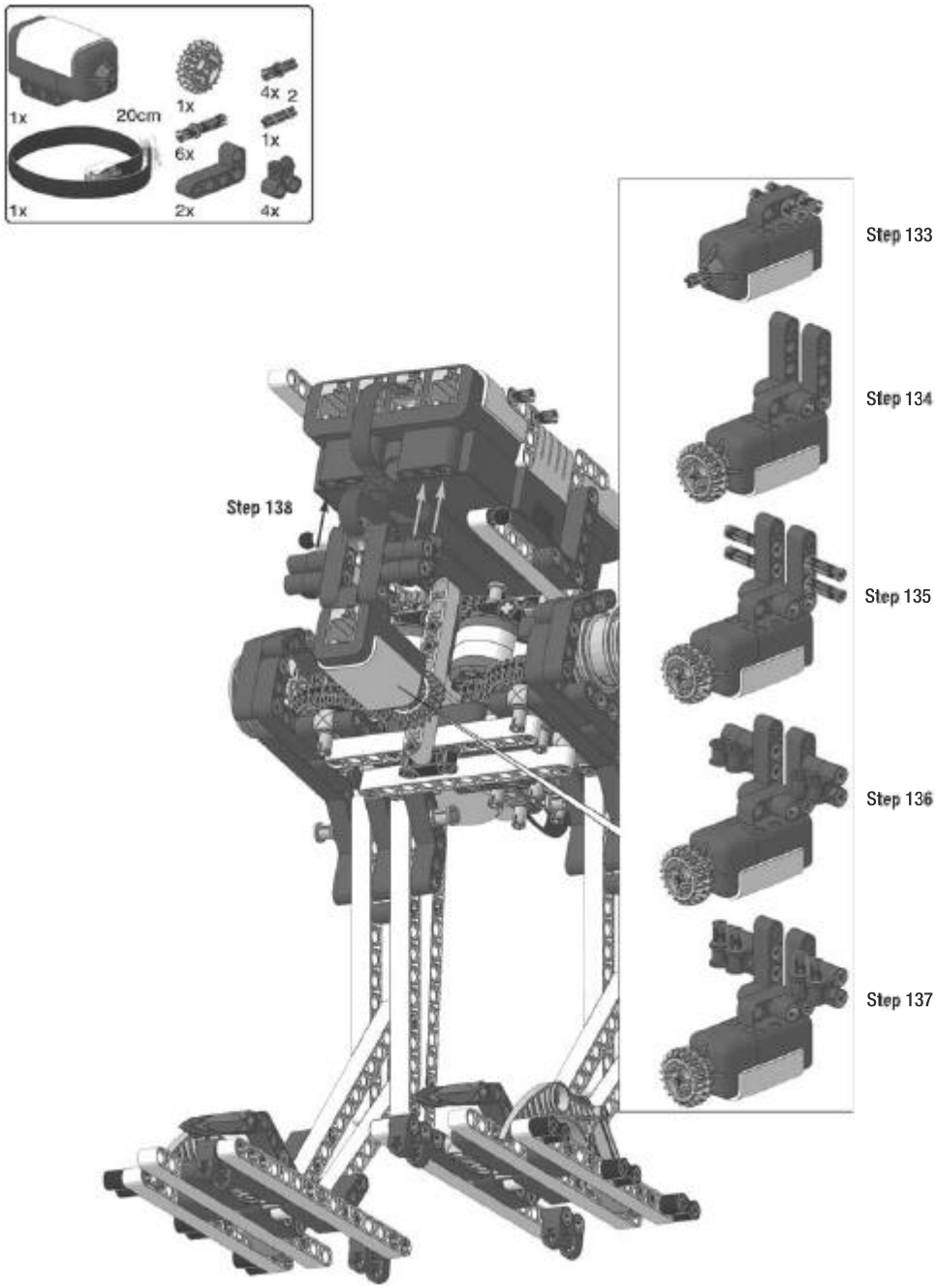
Connect the right motor to NXT output port A using a 35cm (14 inch) cable.



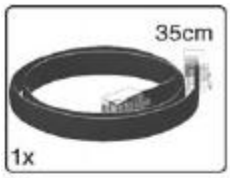
This frame will firmly connect the neck to the motors. Also, this time use the cross-bracing technique.

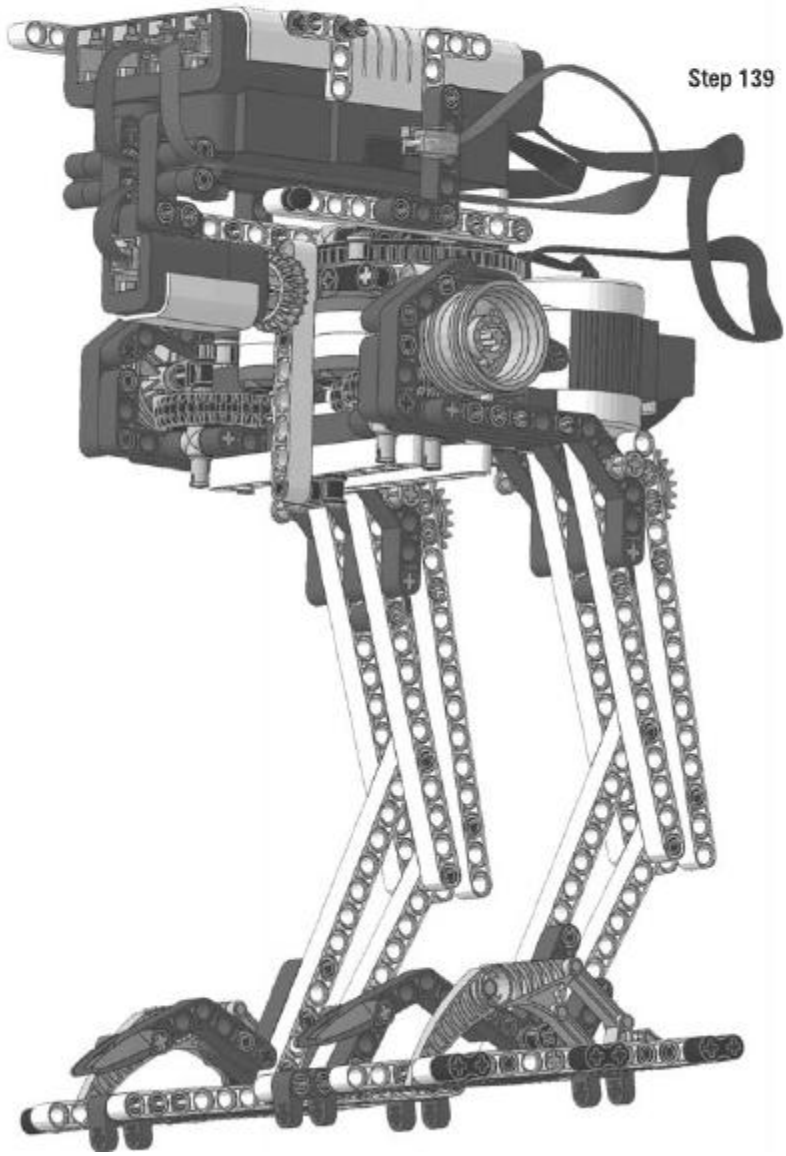


Attaching these side assemblies will secure the NXT to the neck, and so to the rest of the robot.

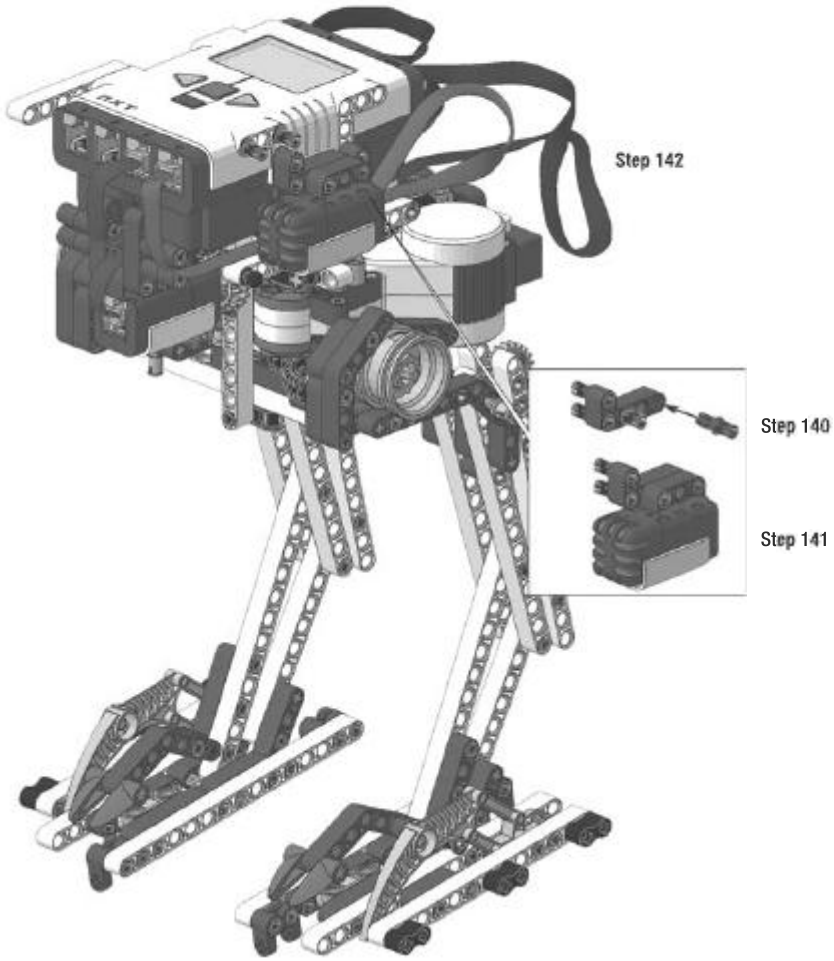


Build the Touch Sensor assembly and attach it under the head. Connect the Touch Sensor to NXT input port 3 using a 20cm (8 inch) cable.

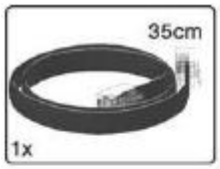


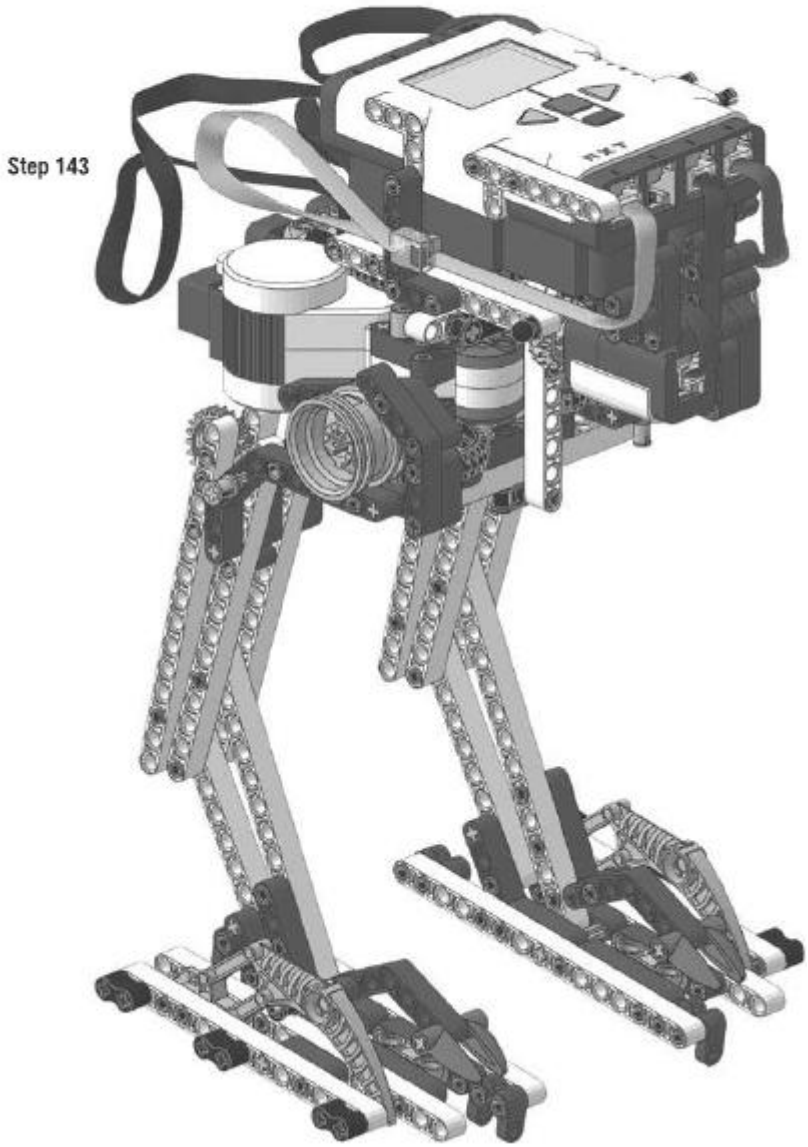


Attach a 35cm (14 inch) cable to NXT input port 4 and pass it under the bent beam on the left side of the head.

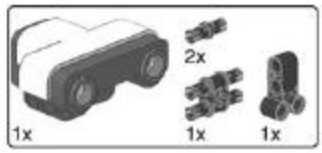


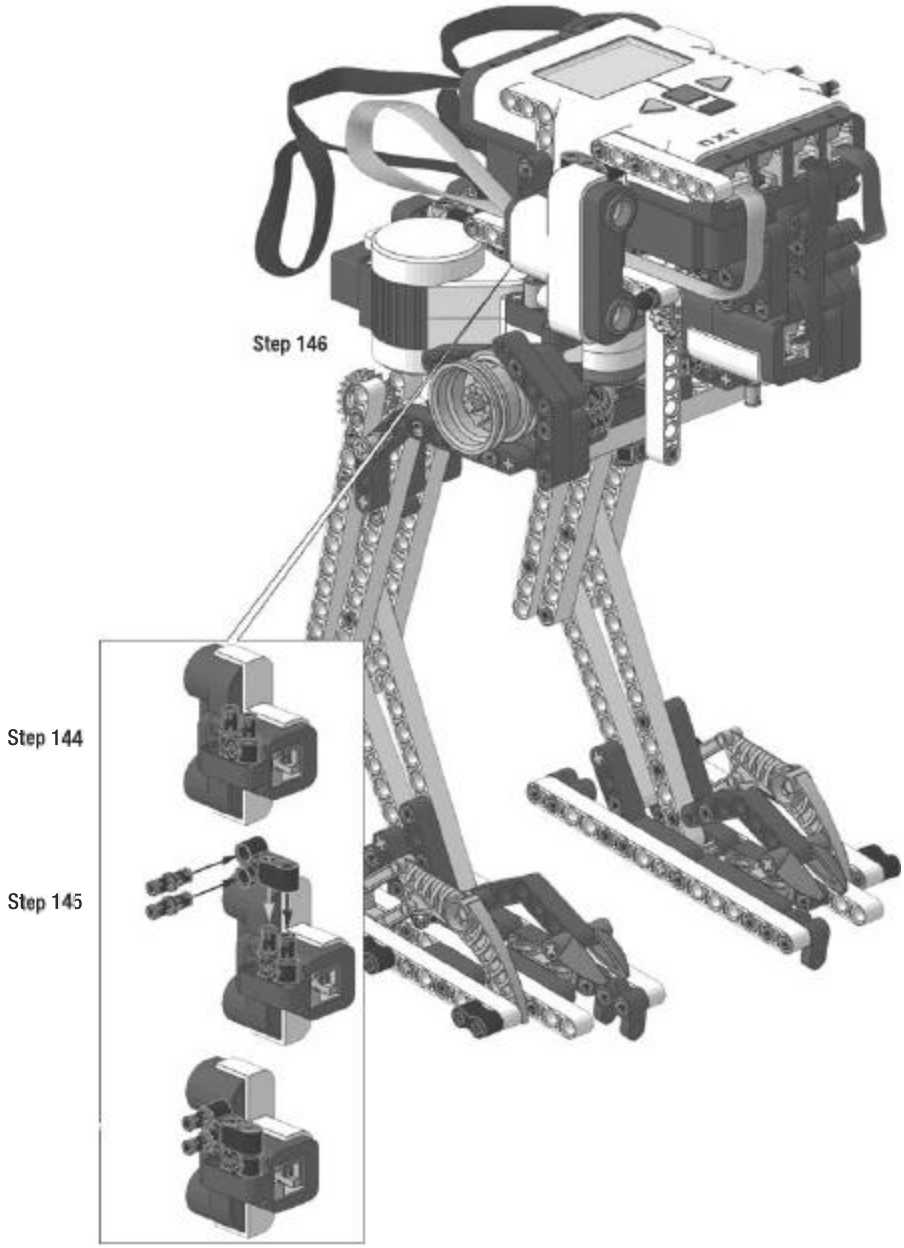
Build the Sound Sensor assembly and attach it to the NXT and to the cable left free in the preceding step.



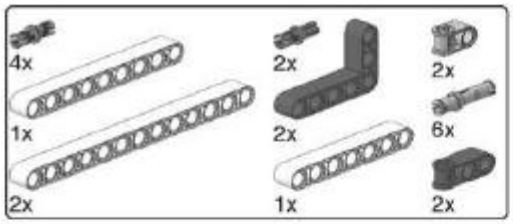


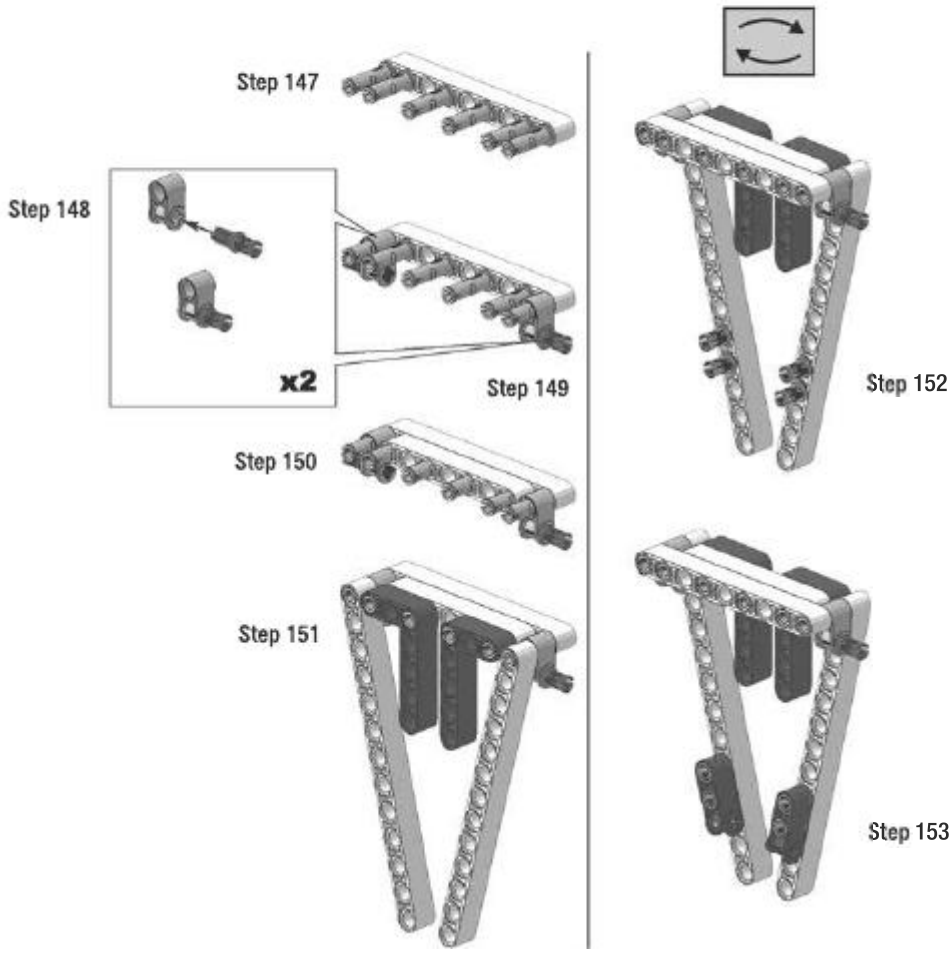
Turn the model and attach another 35cm (14 inch) cable to NXT input port 1, passing it under the bent beam on the right side of the head.



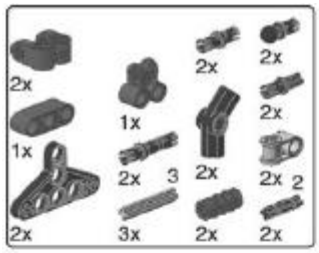


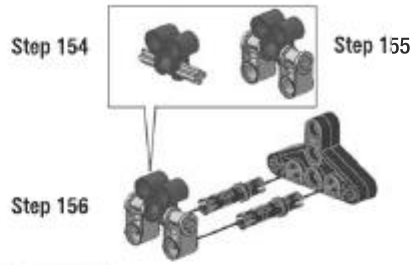
Build the Ultrasonic Sensor assembly and attach it to the NXT and to the cable left free in the preceding step.

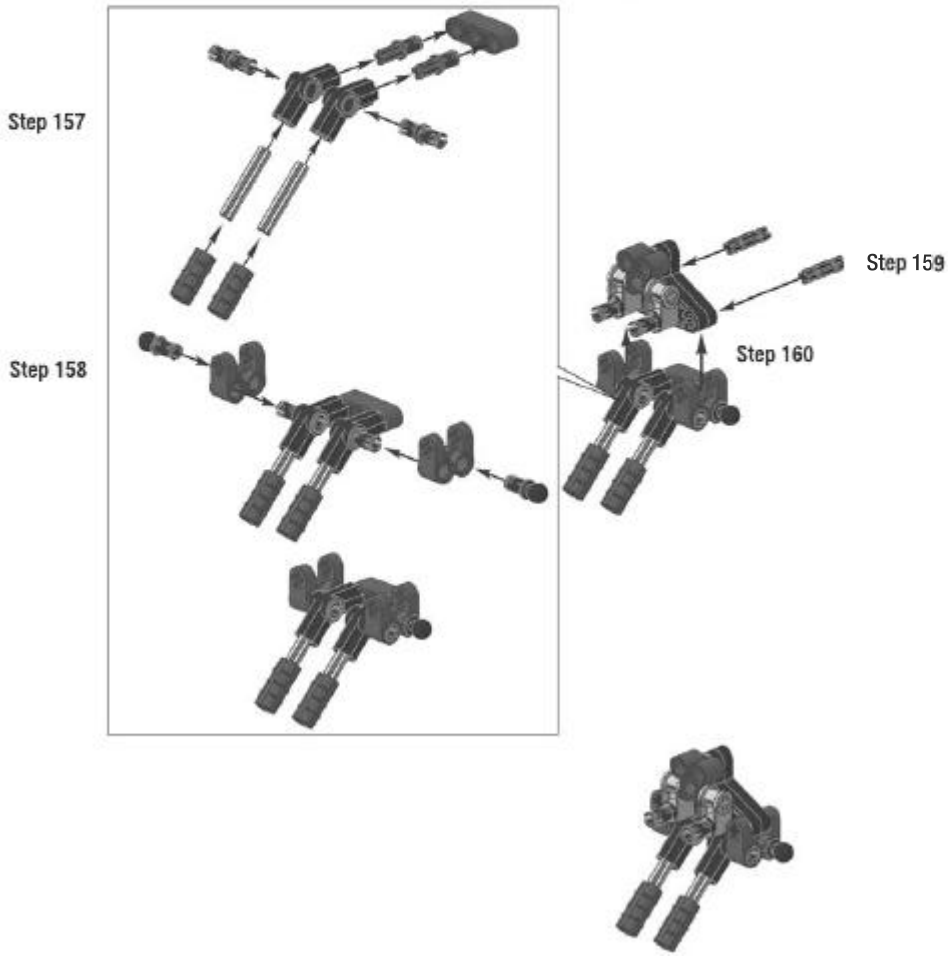




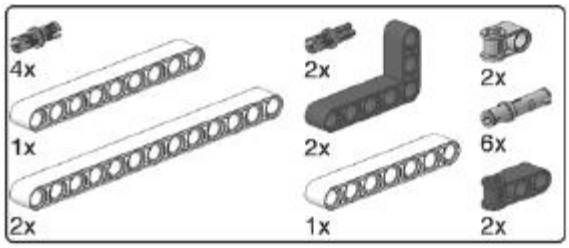
Start building the face of the robot.



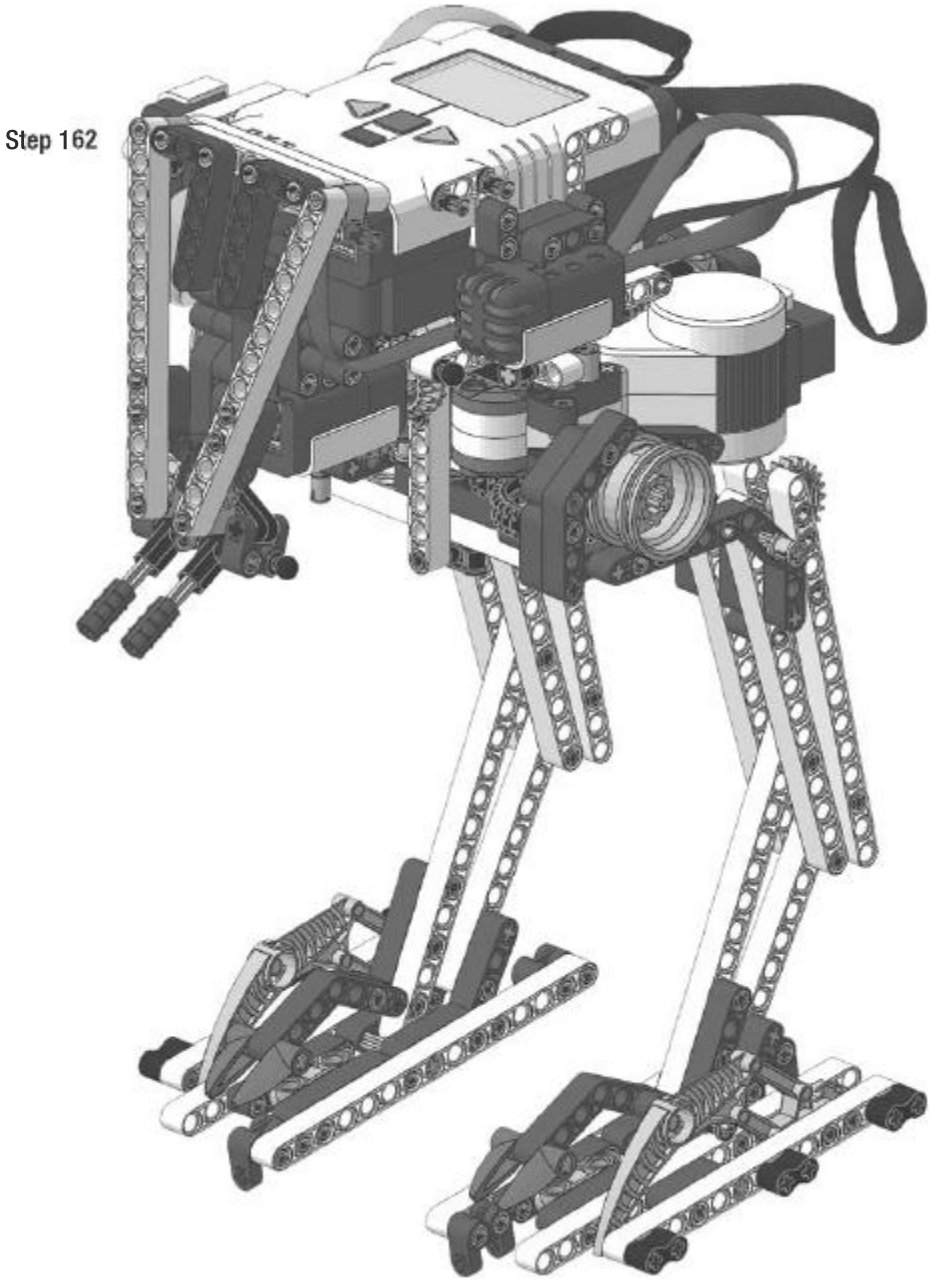




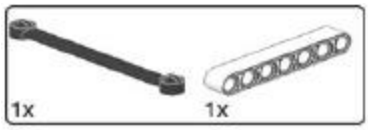
Build the chin-mounted laser cannons.

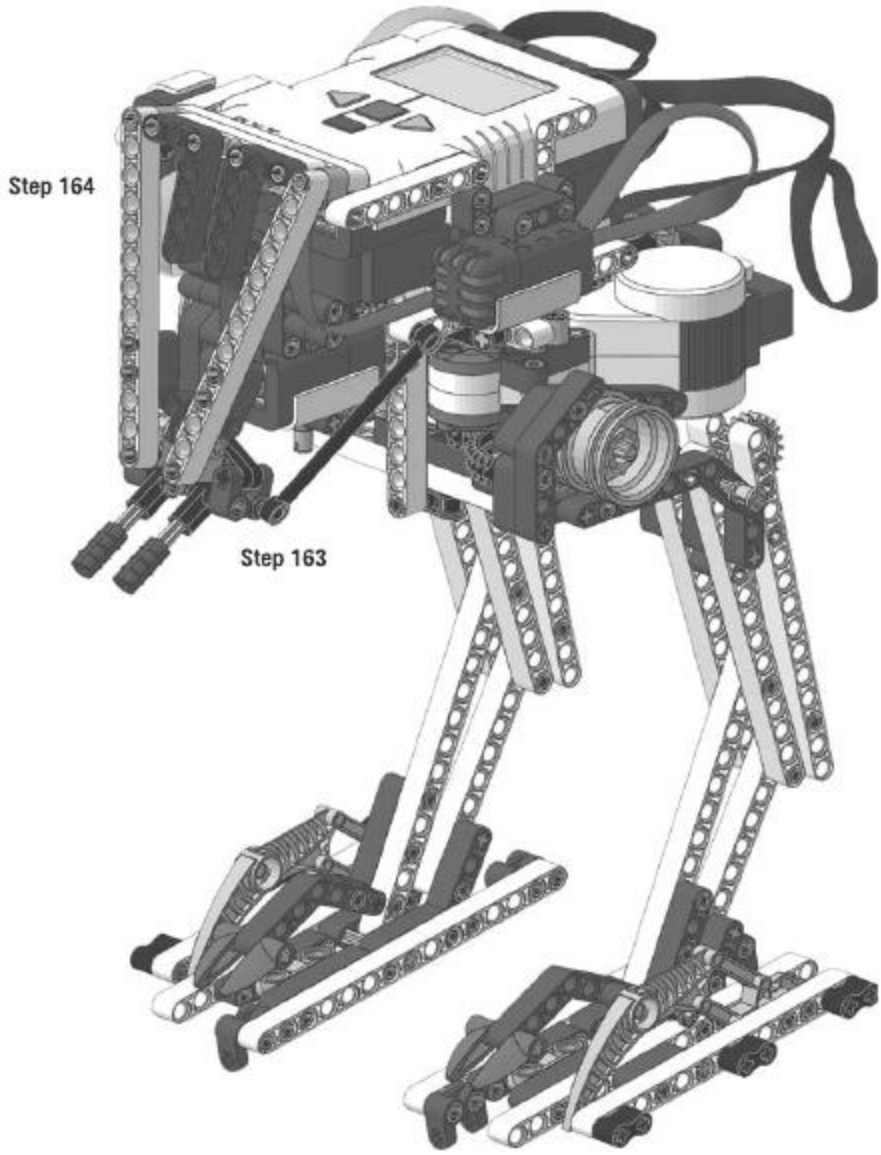


Complete the AT-ST face assembly.



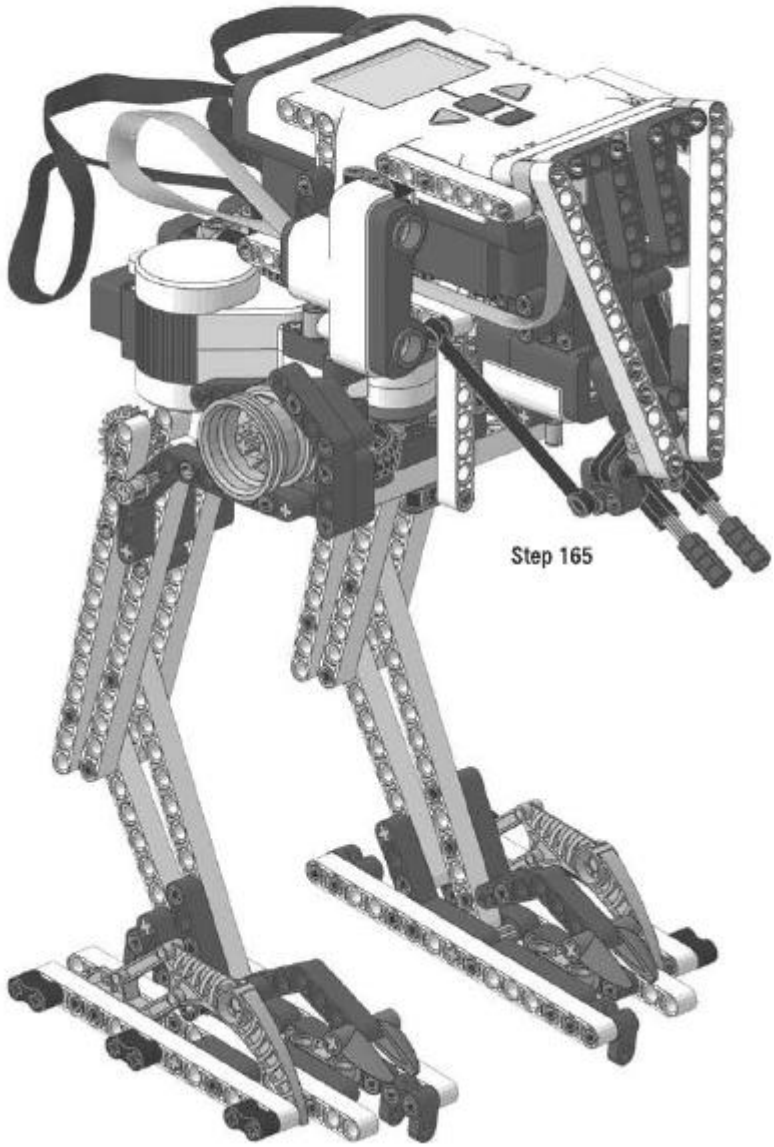
Attach the face to the rest of the head.





Place a 7–long beam and a black steering link to hold the head.


1x


Step 165