EV3 Scratch

Release 2020

Apr 11, 2020

Contents:

CHAPTER 1

Introduction

This tutorial shows how to program the LEGO MINDSTORMS EV3 robot with the **EV3 Classroom** software.

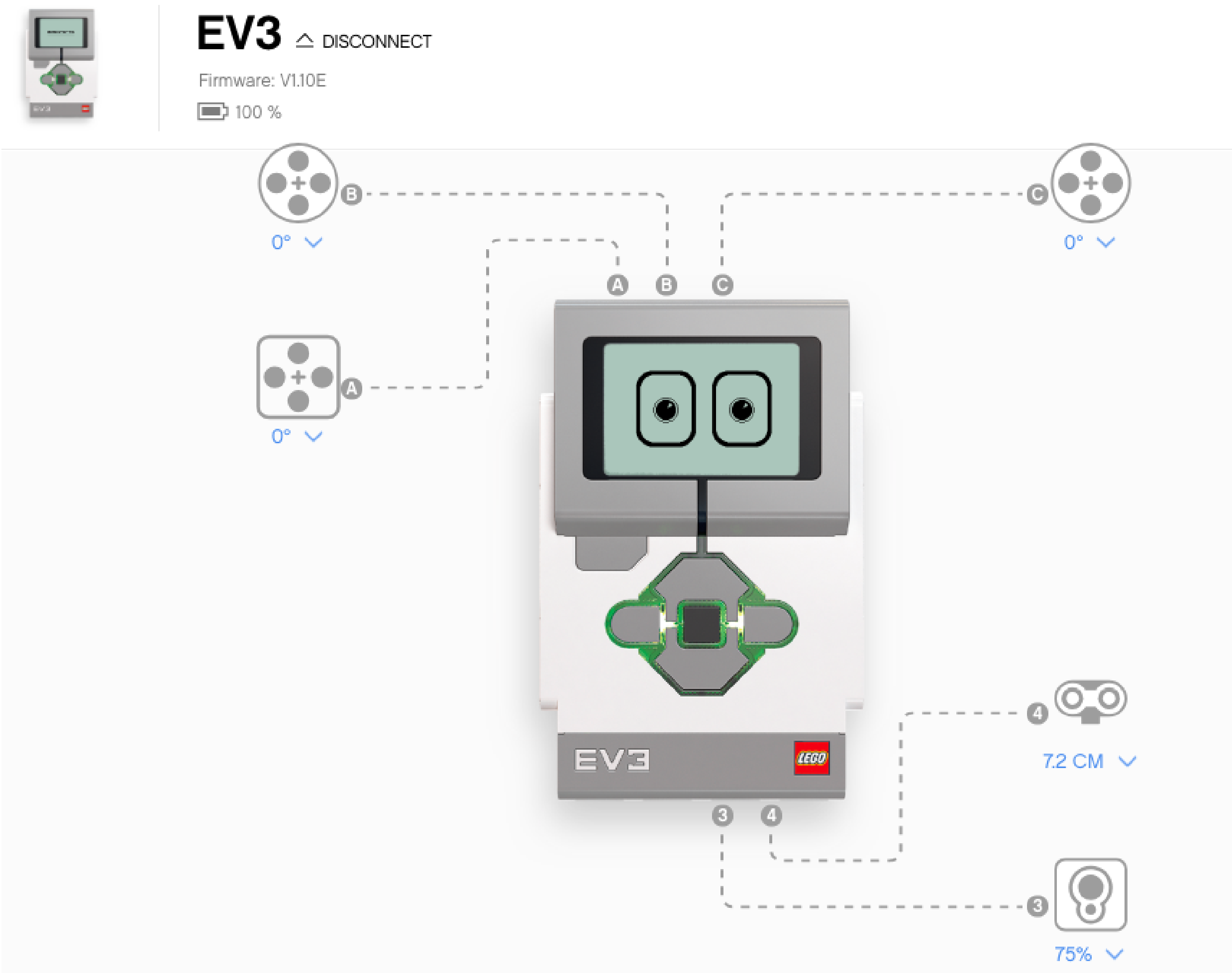
1.1 Connect the EV3

In order to download programs, your robot needs to be connected via USB cable or Bluetooth. When your EV3 is connected to your computer, the red dot next to the EV3 brick icon turns green, and all the attached motors and sensors are shown.

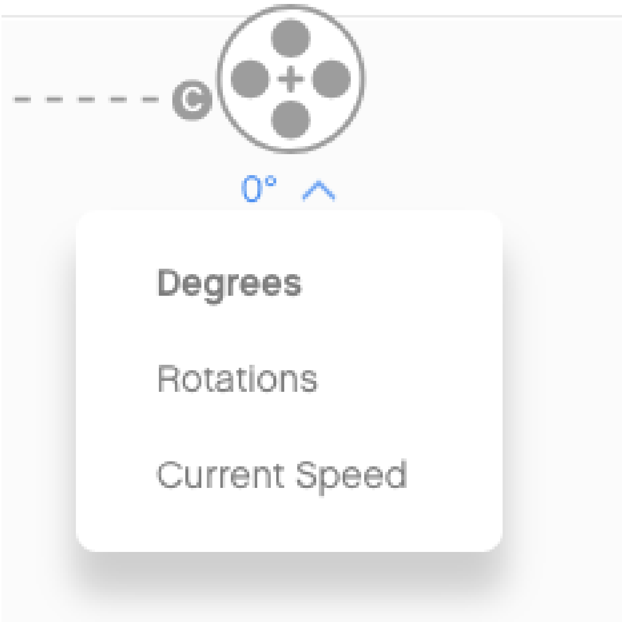
1.2 The dashboard

When your EV3 is connected you can click the brick icon to open the dashboard. The dashboard provides useful information about:

- EV3 name
- firmware version
- battery level
- motors and sensors
- real-time values

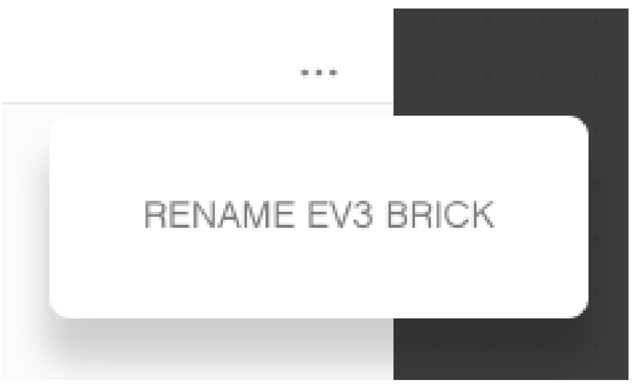


The dashboard displays real-time values of sensors and motors. You can choose which value you want display.



An **Update** button will appear when new firmware is available.

You can rename the brick by clicking on the `...` menu.



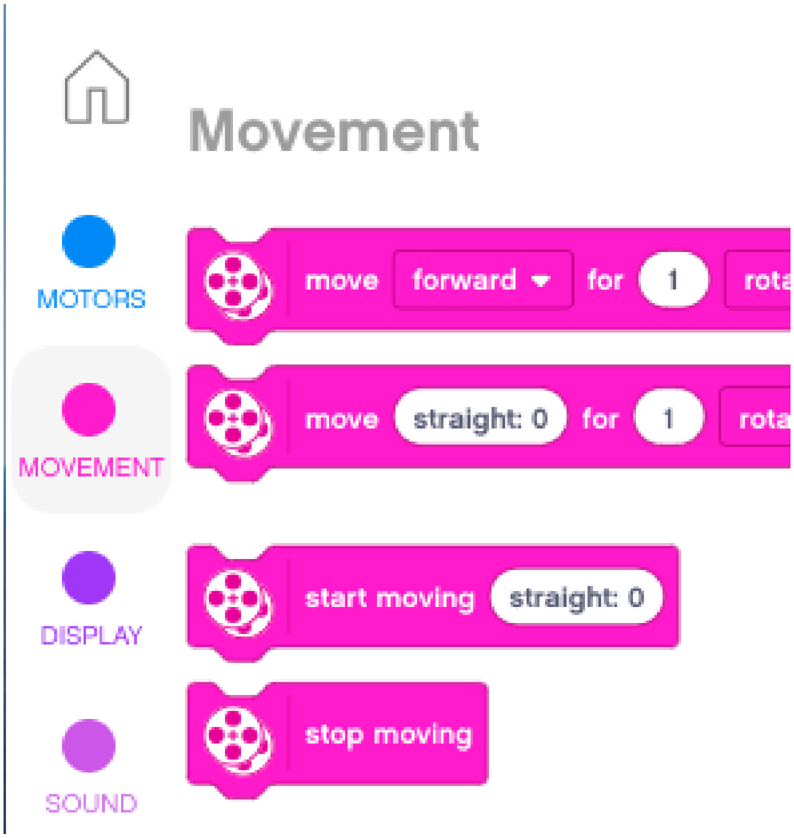
1.3 The programming canvas

The **programming canvas** is where you will create programs. It consists of:

- block palette
- programming area
- tab bar with open projects
- dashboard overview
- controls to zoom, redo, undo, download, etc.

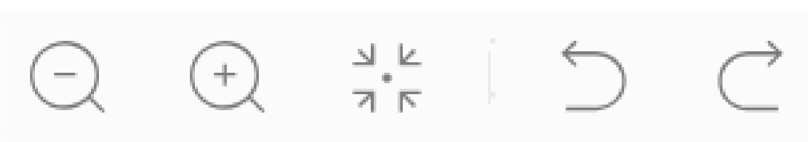
1.4 The block palette

The **block palette** contains the available blocks grouped by functionality.



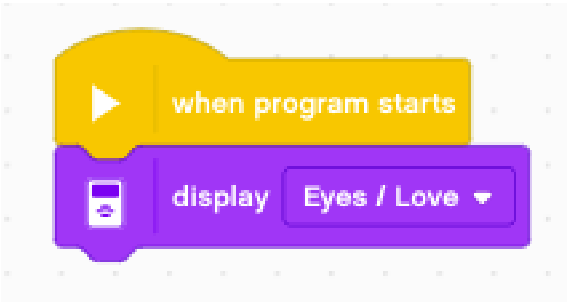
- to use a block, drag it to the canvas.
- to delete a block, drag it back to the palette

To zoom, redo and undo use these 5 buttons

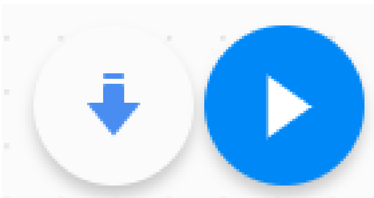


1.5 Display eyes

In our first program we are going to display an emotion on the EV3 screen.

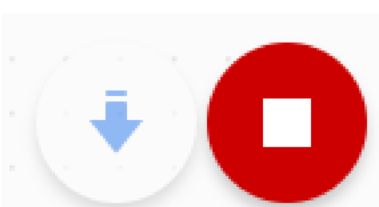


To download and execute the program click on the blue button.

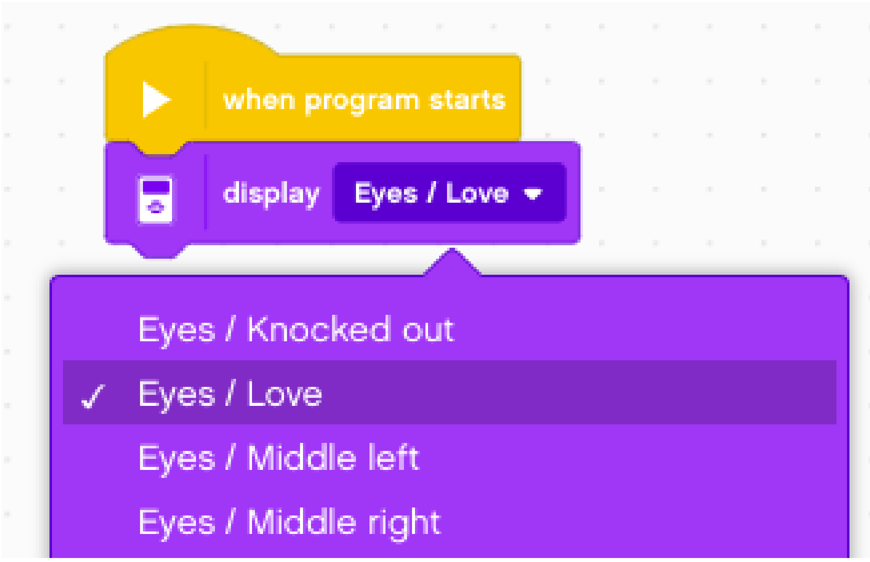


When you download and execute the program, the robot displays this

The program continues to display this image until you quit the program with the red **stop** button.

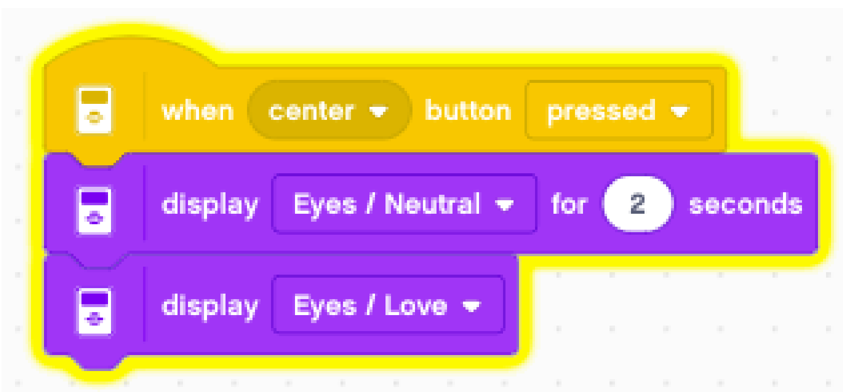


You can select a different image and try again.



1.6 Press a button

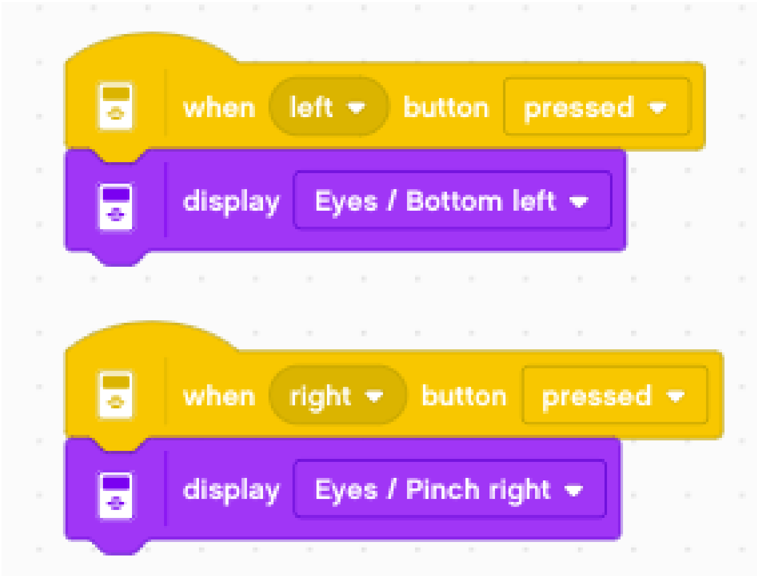
Use the **center** button to change the image on the EV3. When pressing that botton, we show a different image (Eyes/Neutral) but just for 2 seconds. After that we come back to the original image.



When you download and execute the program you can observe, your program get's feedback from the EV3. Every time you press the center button, the part of the code activated will have a **yellow outline** for 2 seconds.

1.7 Press left/right

You can add more buttons to your program. For example change the image shown when pressing **left/right**.



1.8 Press up/down

You can add even more buttons to your program. For example change the image shown when pressing **up/down**.

You can download this file:

`intro.lmsp`

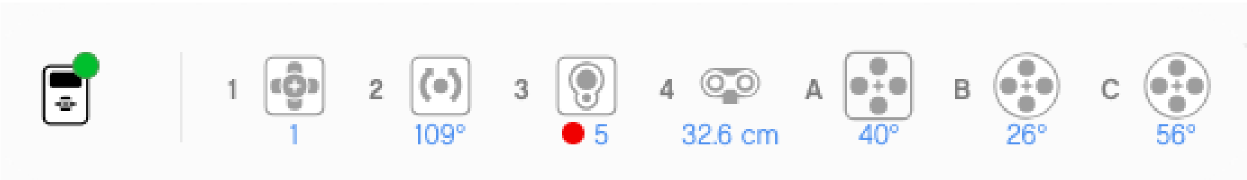
CHAPTER 2

Sensor

Sensors perceive the environment and send data to the robot.

2.1 Real-time dasboard

The sensors are connected to the robot via ports 1 to 4. Small icons at the top of the program show the current values.



- the touch sensor is pressed (value=1)
- the gyros sensor shows 109°
- the color sensor sees the color red (value=5)
- the ultrasound sensor measures a distance of 32.6 cm

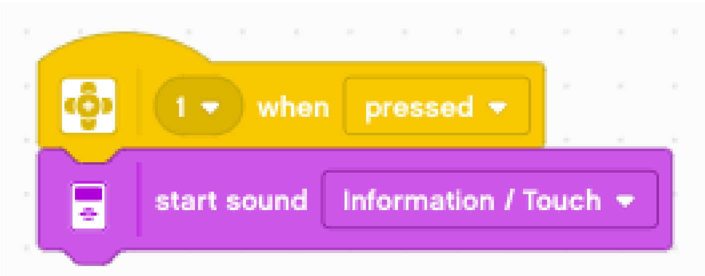
The motors are connected to ports A to D. They contain rotation sensors and display the current angular position:

- the medium-size motor on port A is at 40°
- the large motor on port B is at 26°
- the large motor on port C is at 56°

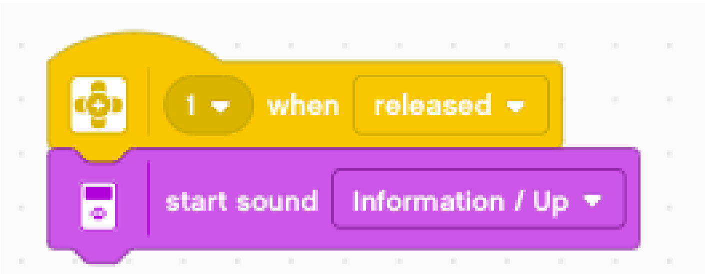
2.2 Touch sensor

The touch sensor can be used on robot to detect physical touch. It can be mounted as a bumper or an antenna.

We program it to say something when the touch sensor is pressed.

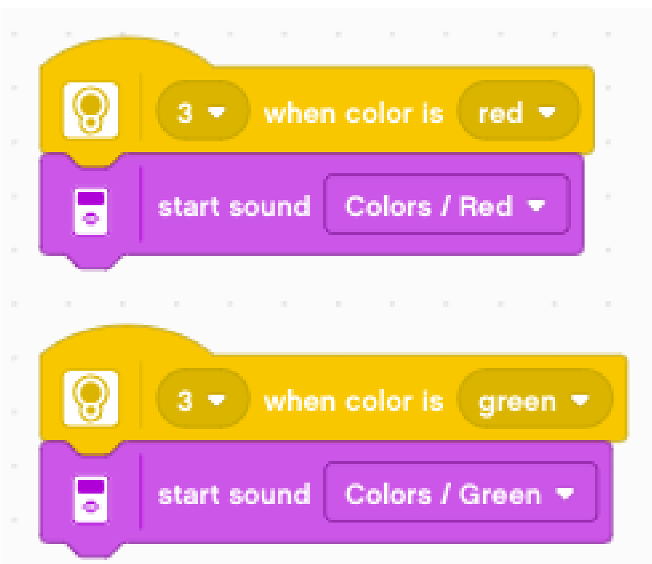


But we can also program it to do something when it is released.



2.3 Color sensor

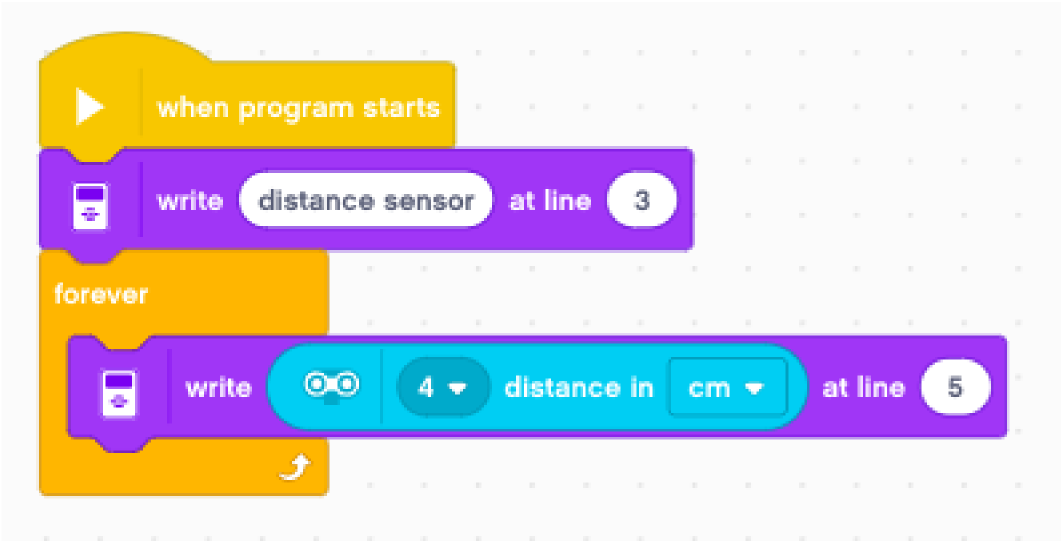
When the color sensors sees red or green it pronounces these colors



2.4 Distance sensor

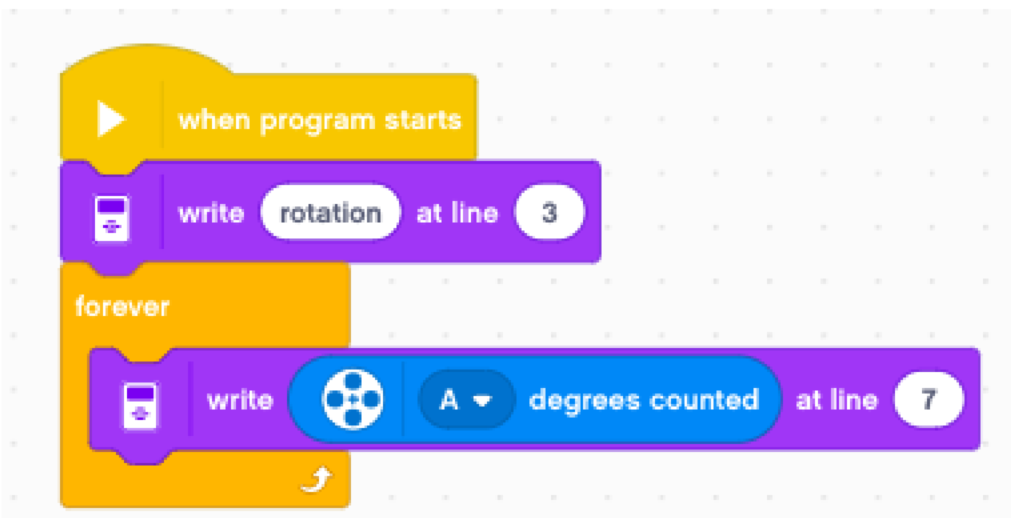
Now let's use a different method. We will continously measure the distance and display it. For this we will use the **forever** loop.

On line 3 we write once the explanation **distance sensor**. Inside the loop, we write continously the value measured with the disance sensor.



2.5 Rotation sensor

As we have seen in the beginning, all the motors have a rotation sensor built-in. We can use the wheels as input knobs and display the values.



If you look carefully, you notice the values are positive to one side, negative to the other. At the start of the program, the value is always 0.

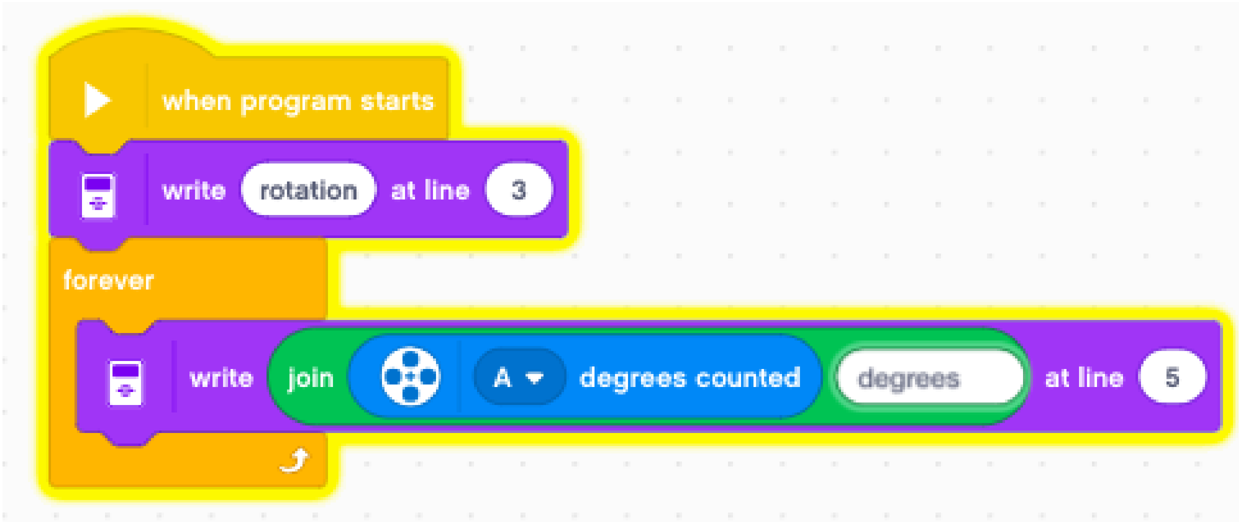
Do you notice a problem? Let's say you go up to:

```
100
```

This occupies 3 characters. Once written, the caracters are not erased when the number becomes smaller. If you return back to `99` the display will now show `990`. And when you're back at `9` the display will now show:

```
900
```

There is a trick to correct this. The green operator **join** allows you to attach a couple of empty spaces after the number. These will erase any extra digits.



We even can do better. We can add a unit after the number. Take care to add 1 space before and 4-5 spaces afterwards. Your string shoud look like this: _degrees_____.

CHAPTER 3

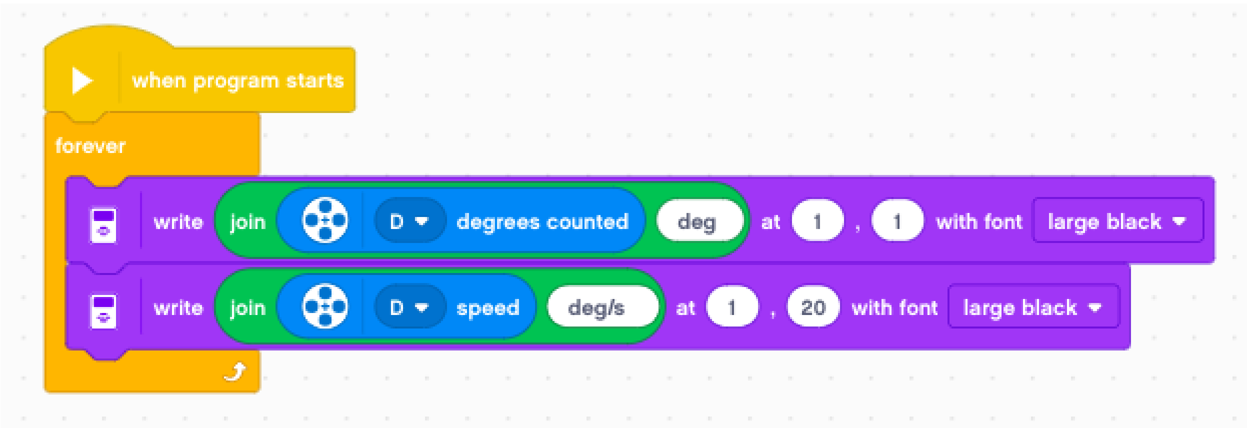
Motor

With the motors the robot can move around.

3.1 Display speed and position

Each motor has a rotational sensor. You can read:

- angular position (degrees)
- angular speed (degrees/sec)



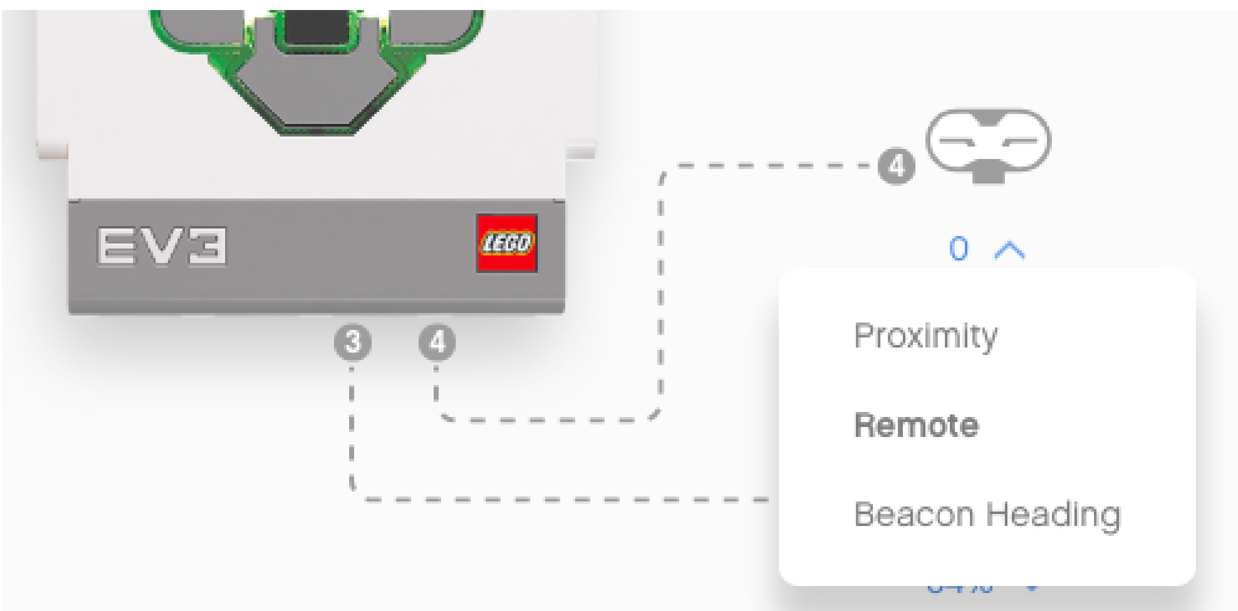
CHAPTER 4

Remote

The EV3 has an infrared sensor. We connect it to port 4.

The sensor has three functions:

- proximity
- remote
- beacon heading

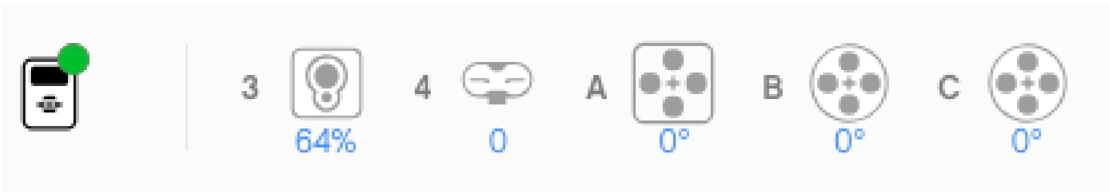


Here we select **remote**.

In order to be able to control multiple robots separately, the remote control has 4 different channels.

4.1 The role of the buttons

In the top icon view you can see the sensor state.



When pushing the buttons on the remote control you will get:

- 0 : no button
- 1 : left-top
- 2 : left-bottom
- 3 : right-top
- 4 : right-bottom
- 9 : activate beacon button

You can also press **two buttons** at the same time:

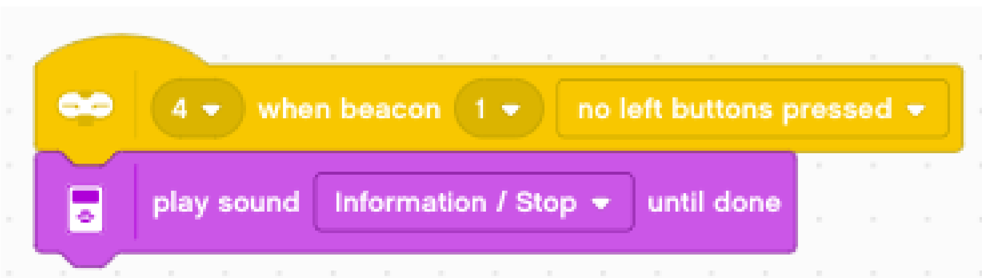
- 5 : top two
- 6 : diagonal down
- 7 : diagonal up
- 8 : bottom two
- 10 : left two
- 11 : right two

4.2 Detect a button press

When a button is pressed on the remote control, we can play a sound. For exemple for the left side buttons

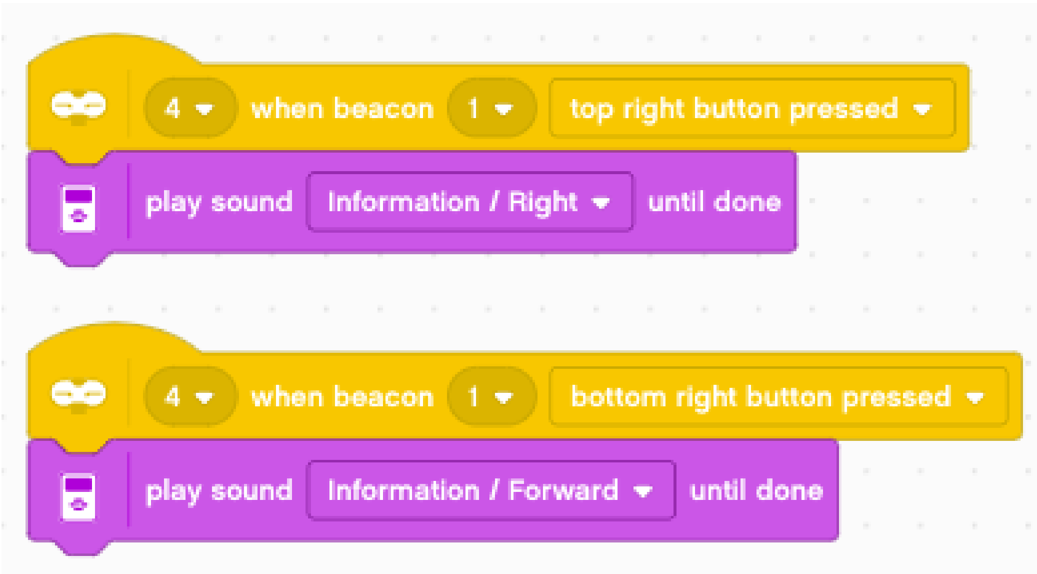
- top left : play **left**
- bottom left : play **backwards**

Push a button and hold it for 2-3 seconds. Then release it. This will activate the **no left button pressed** event, which should rather be called *left button released* event.

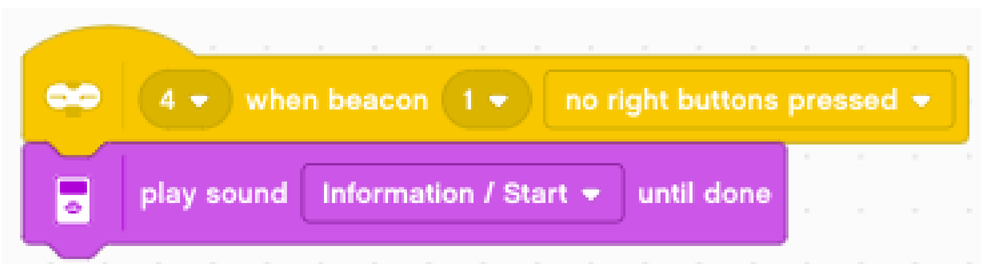


We can program the right side as well.

- top right : play **right**
- bottom right : play **forward**



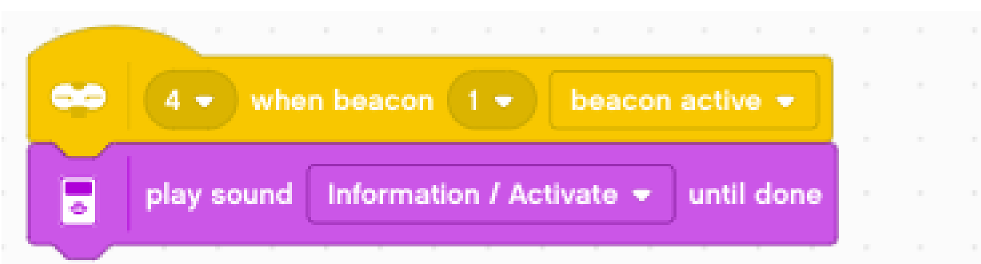
When one of the right buttons is released we do this:



There is one larger button at the top. It activates the beacon and has a toggle function:

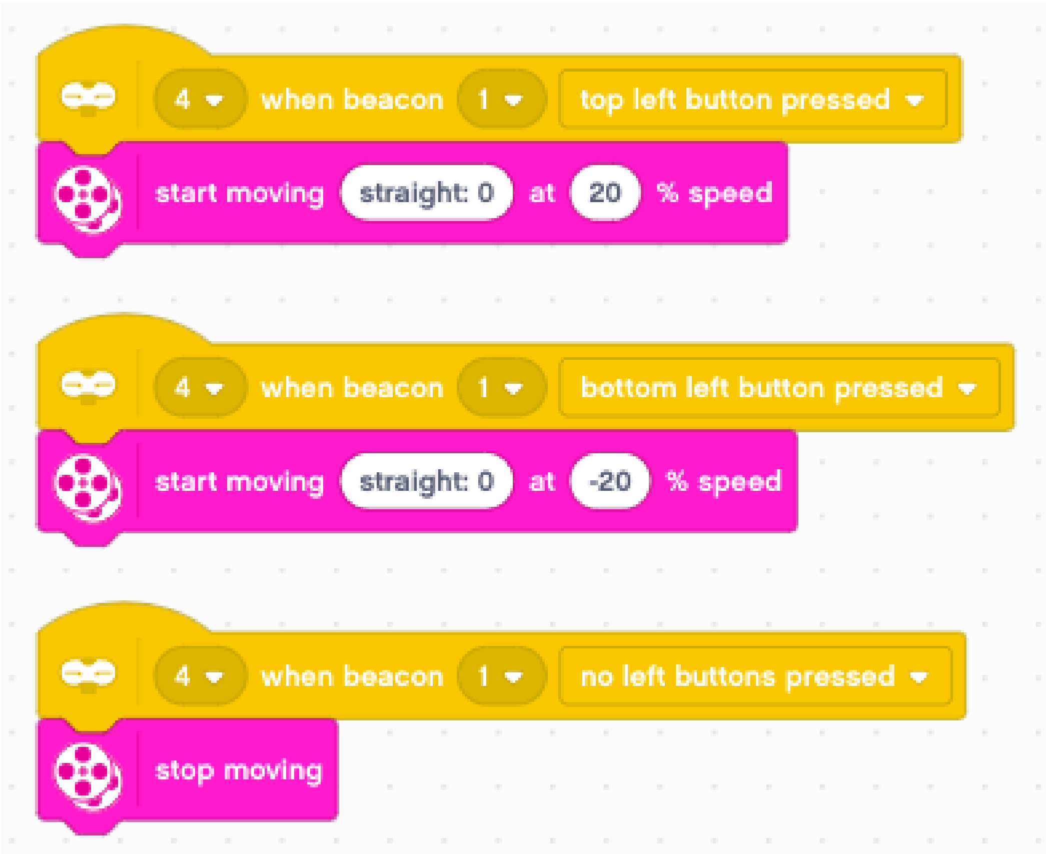
- pushing it once, turns on the green LED
- pushing it again, turns the LED off

When pressed, we play the sound **activate**

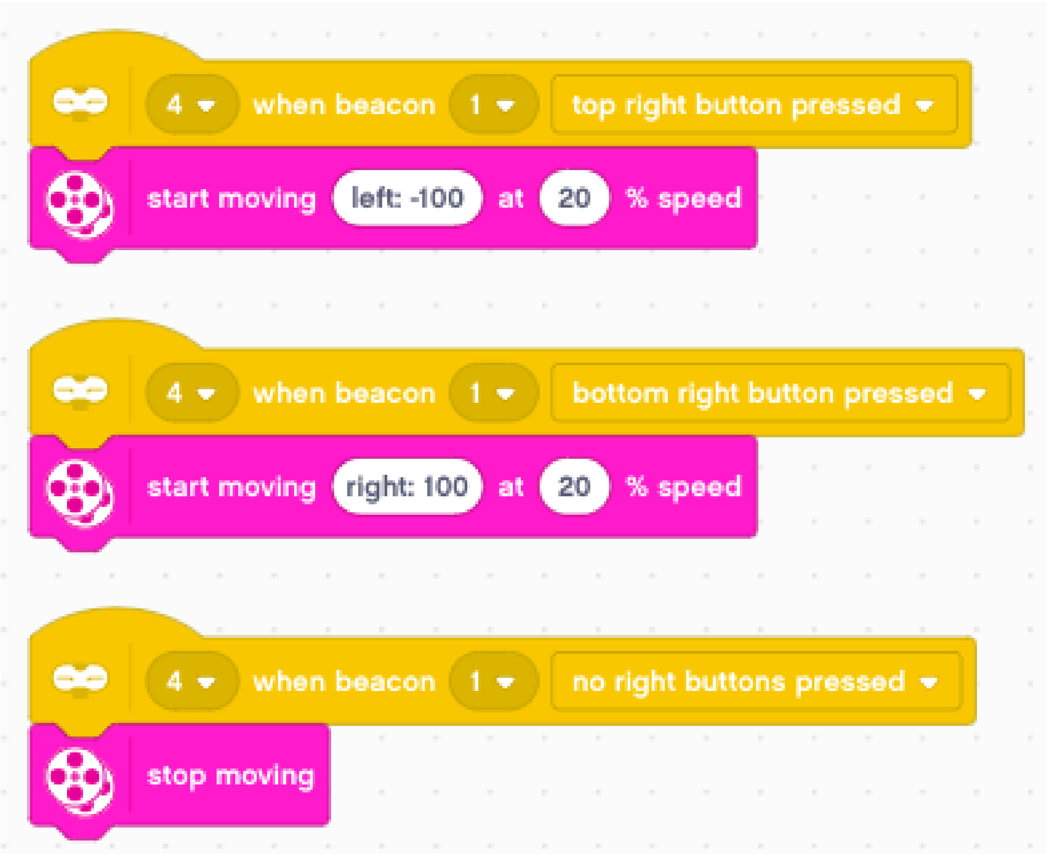


4.3 Controlling the robot

Now we can program the remote unit to control the movement of the robot. We use the left buttons to control the **forward/backward** movement.



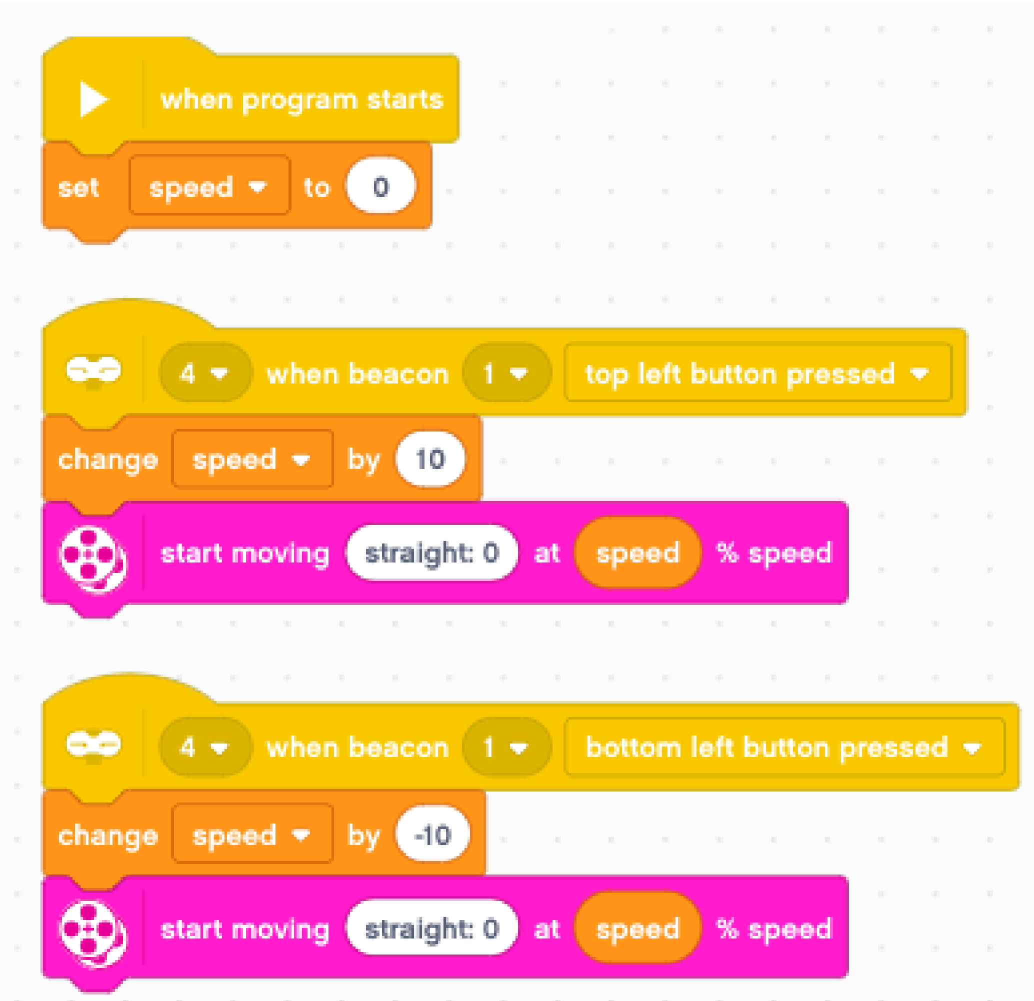
We use the right buttons to control the **left/right** movement.



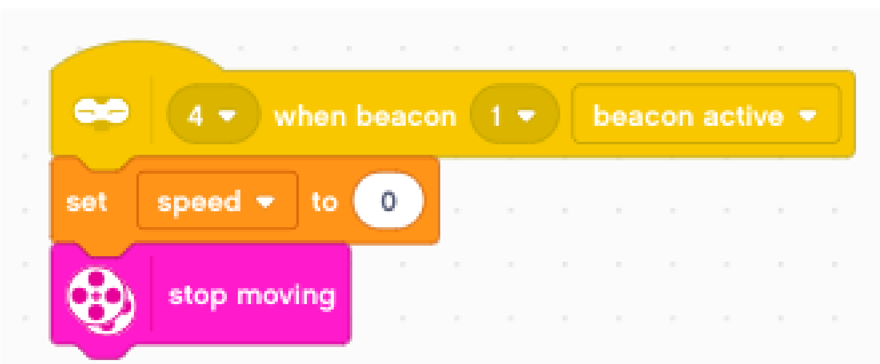
4.4 Controlling motor speed

A more flexible way would be if we could also control the speed. We createa variable **speed** and set it to 0 initially.

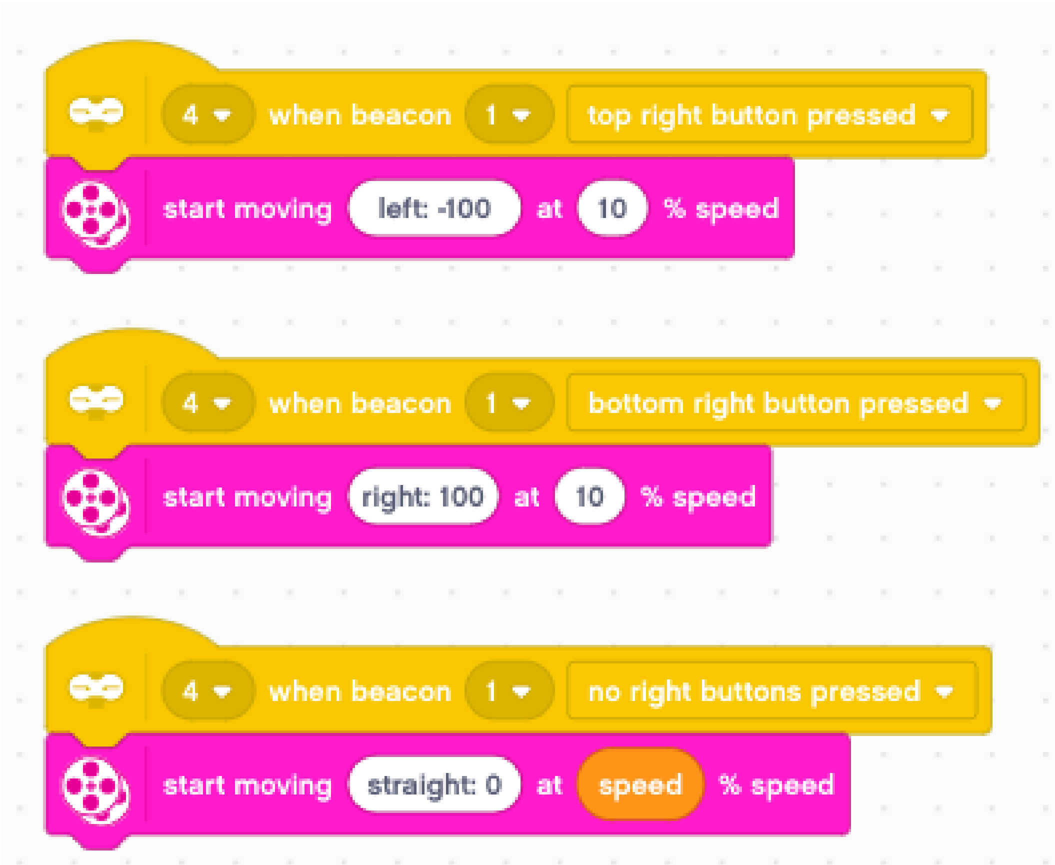
- the top button increases the speed by 10
- the bottom button decreases the speed by 10



We use the **beacon** button for the emergency brake.



And the right side buttons are used to pivot left and right, as long as the buttons are pressed.



4.5 Memorize a path

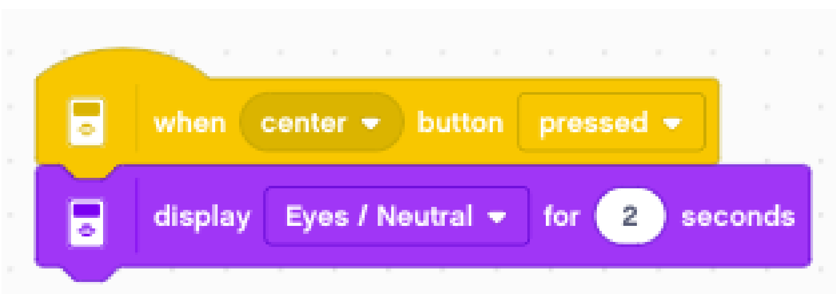
CHAPTER 5

Display

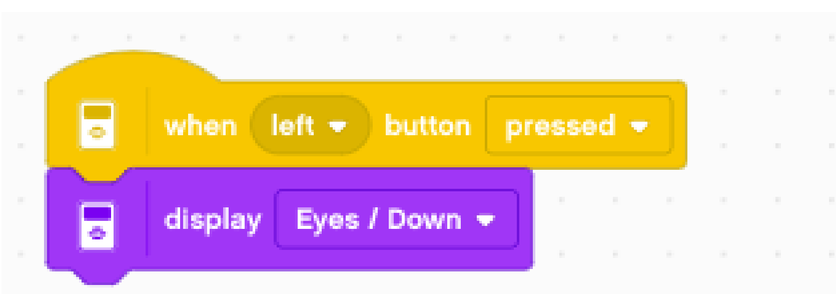
The EV3 can display images and write text.

5.1 Display an image

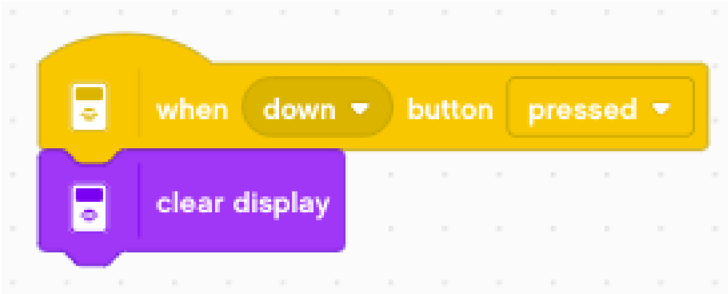
You can display an image for a specified time duration. The following program displays neutral eyes for 2 seconds.



After 2 seconds the screen is cleared and becomes white. There is also an option for displaying an image continously, without erasing.

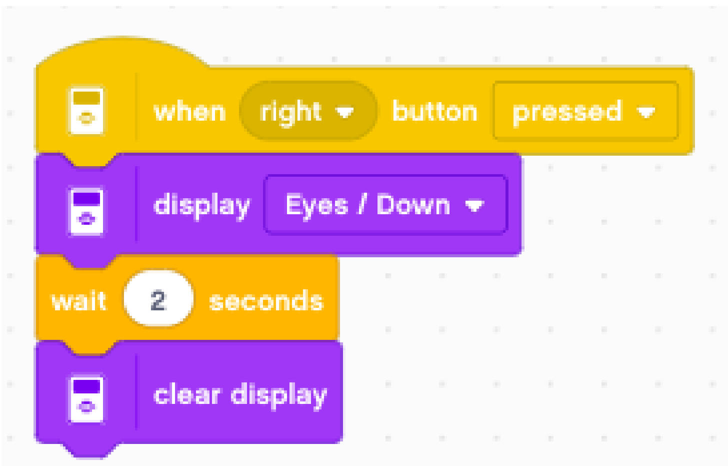


There is a command to clear the screen.



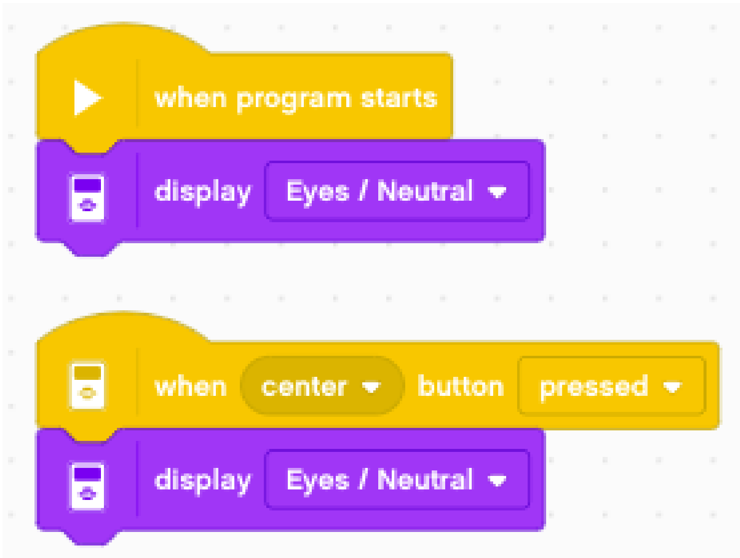
The command **display for X seconds** can be composed from:

- display image
- wait X seconds
- clear display

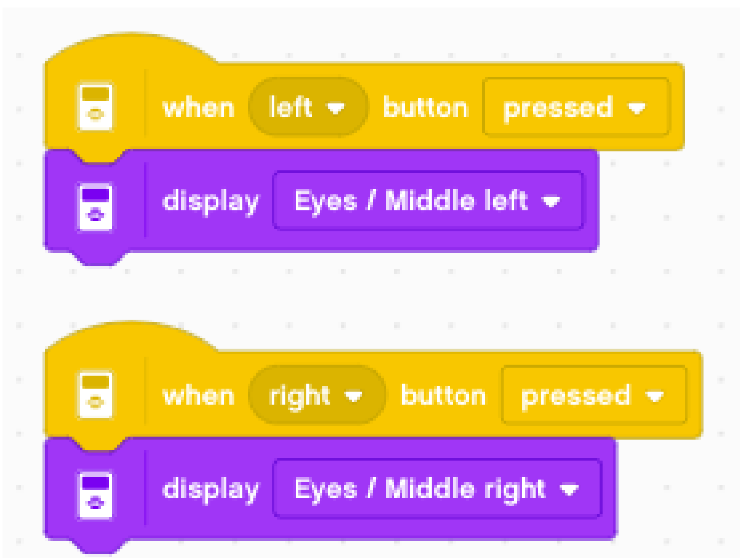


5.2 Move the eyes

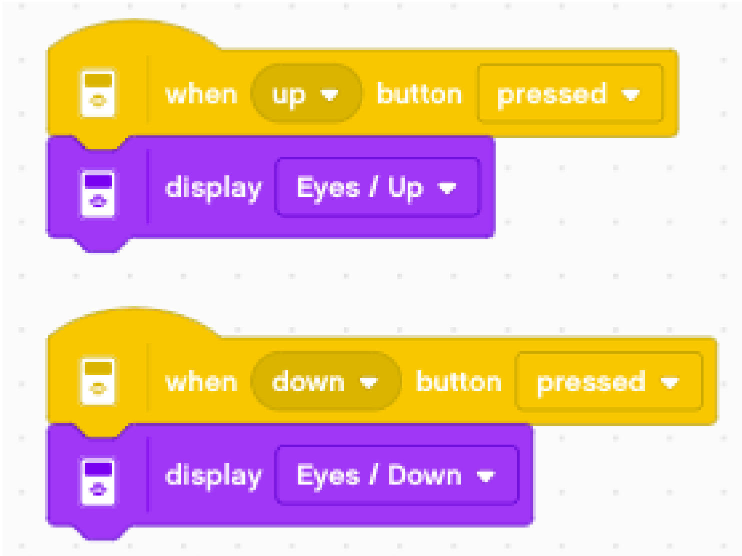
You can use the 5 buttons to display eyes which look into the direction of the button. We start with a neutral position, and can return to that position with the center button.



With the **left/right** buttons you can move the eyes to the left and to the right.



With the **up/down** buttons you can move the eyes up and down.

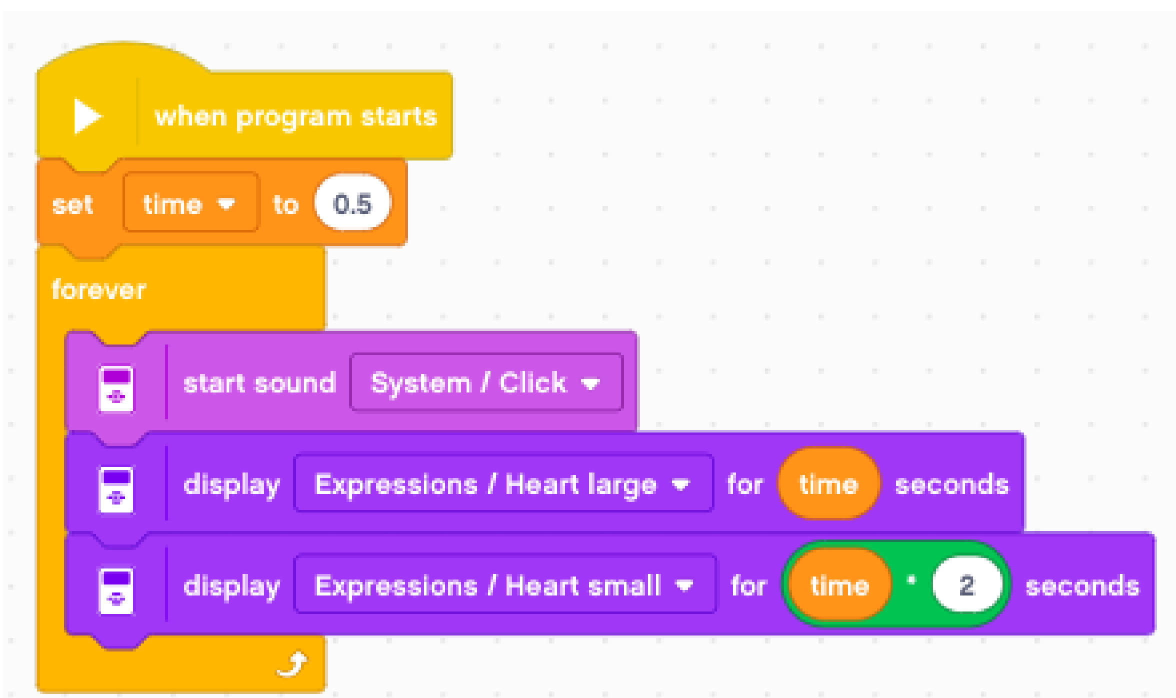


5.3 Show a beating heart

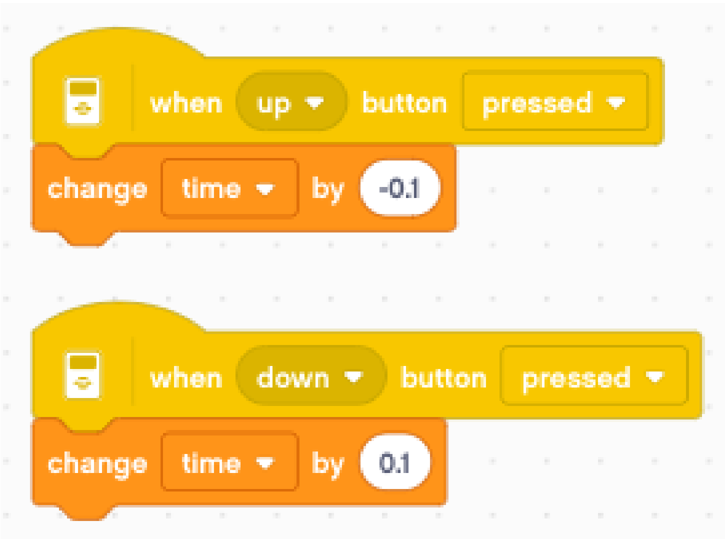
By displaying two images in repetition we can create a simple animation. The following loop displays two hearts, a small one and a larger one.

We define a variable **time** which we set to 0.5. Then we enter a **forever** loop where we:

- play a *click* sound
- display the *large heart* for `time` seconds
- display the *small heart* for `2*time` seconds

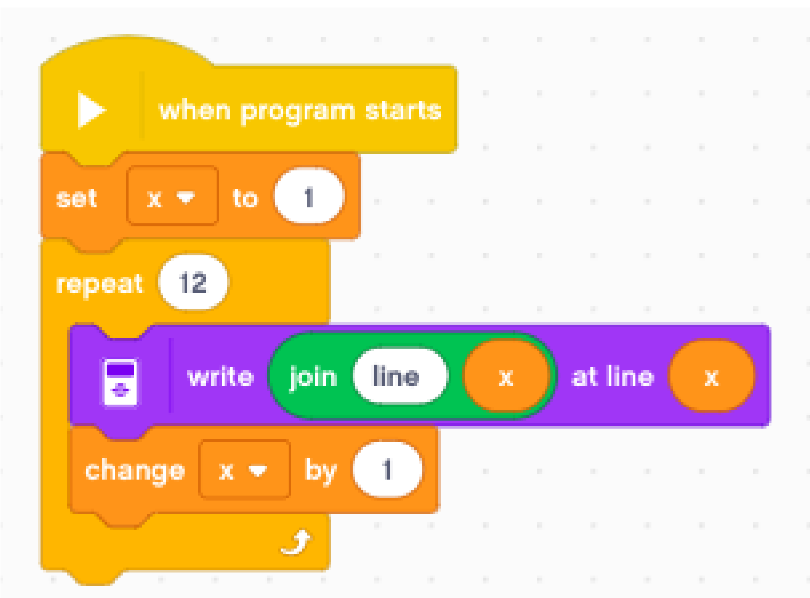


The **up/down** buttons serve to change the **time** variable by increments of 0.1.



5.4 Write lines of text

You can write text to one of 12 lines. The following program sets the variable **x** to 1 and increases it to 12 in a loop, in order to write text on each line.



This is the result:

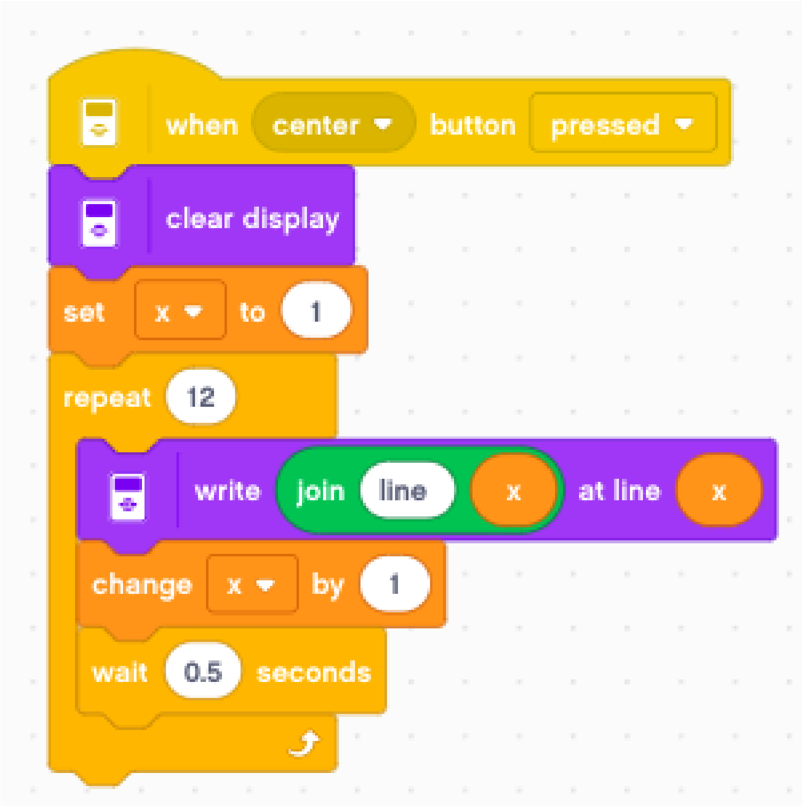
```
line 1
line 2
line 3
line 4
line 5
line 6
```

(continues on next page)

(continued from previous page)

```
line 7
line 8
line 9
line 10
line 11
line 12
```

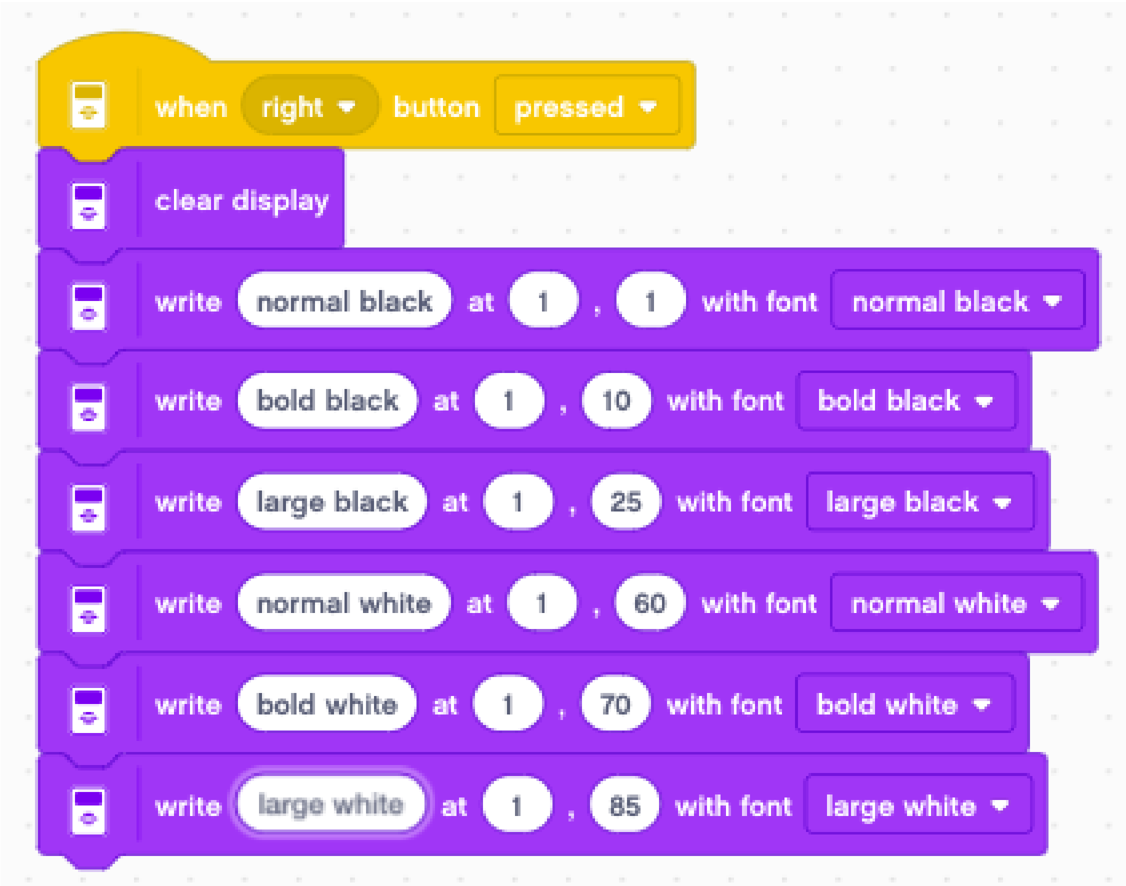
We also can slow it down and write line by line.



5.5 Write in different styles

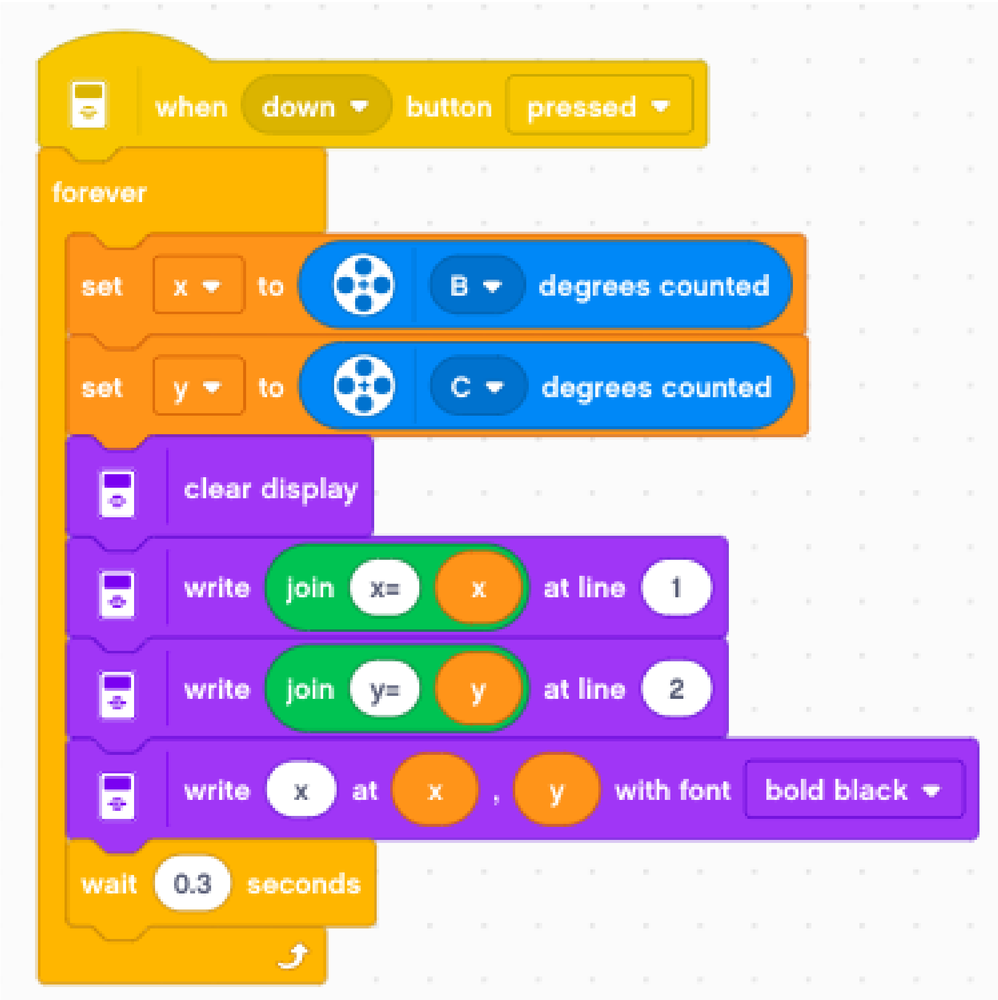
The second write instruction allows to write at any position (x, y) and to use one out of **6 styles**:

- normal black
- bold black
- large black
- normal white
- bold white
- large white



5.6 Write at position (x, y)

The following program uses two rotary encoders to write the letter **x** at position (x, y).

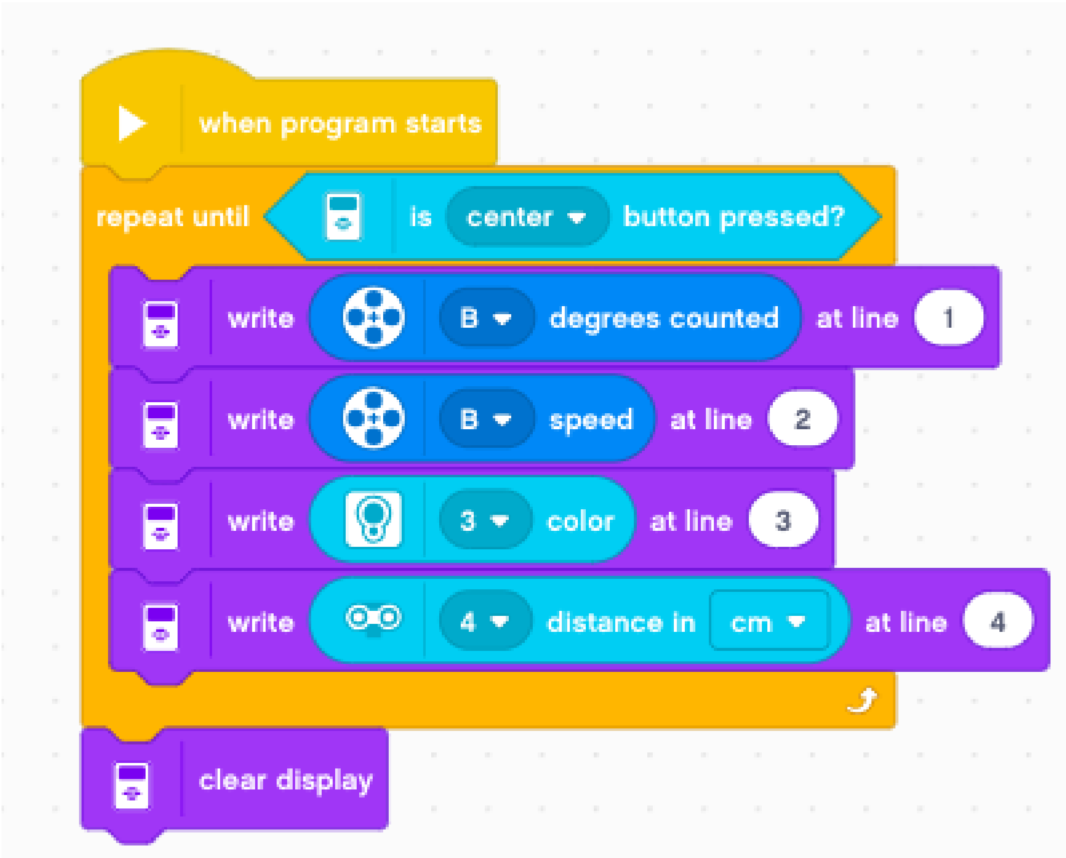


It produces output like this:

```
x=80
y=20
                              x
```

5.7 Display sensor values

Sometimes it is useful to display multiple sensor values on the display. This program displays 4 sensor values on the first 4 lines.

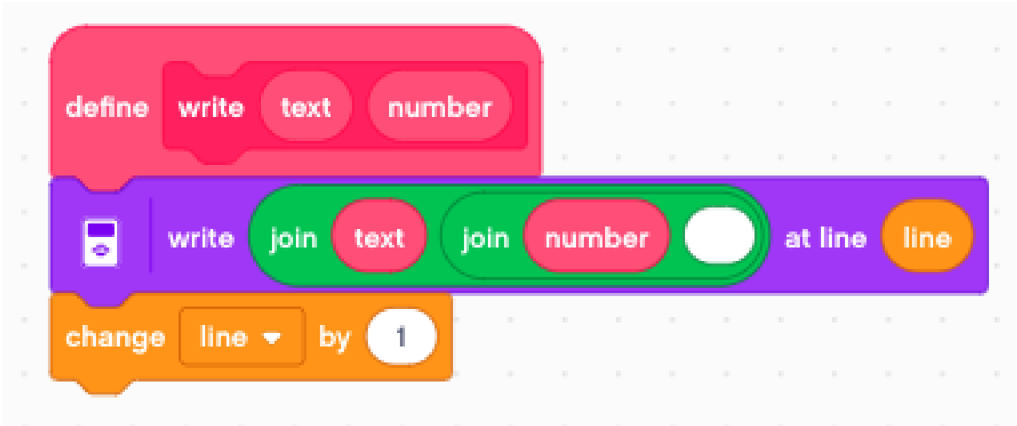


It produces output like this:

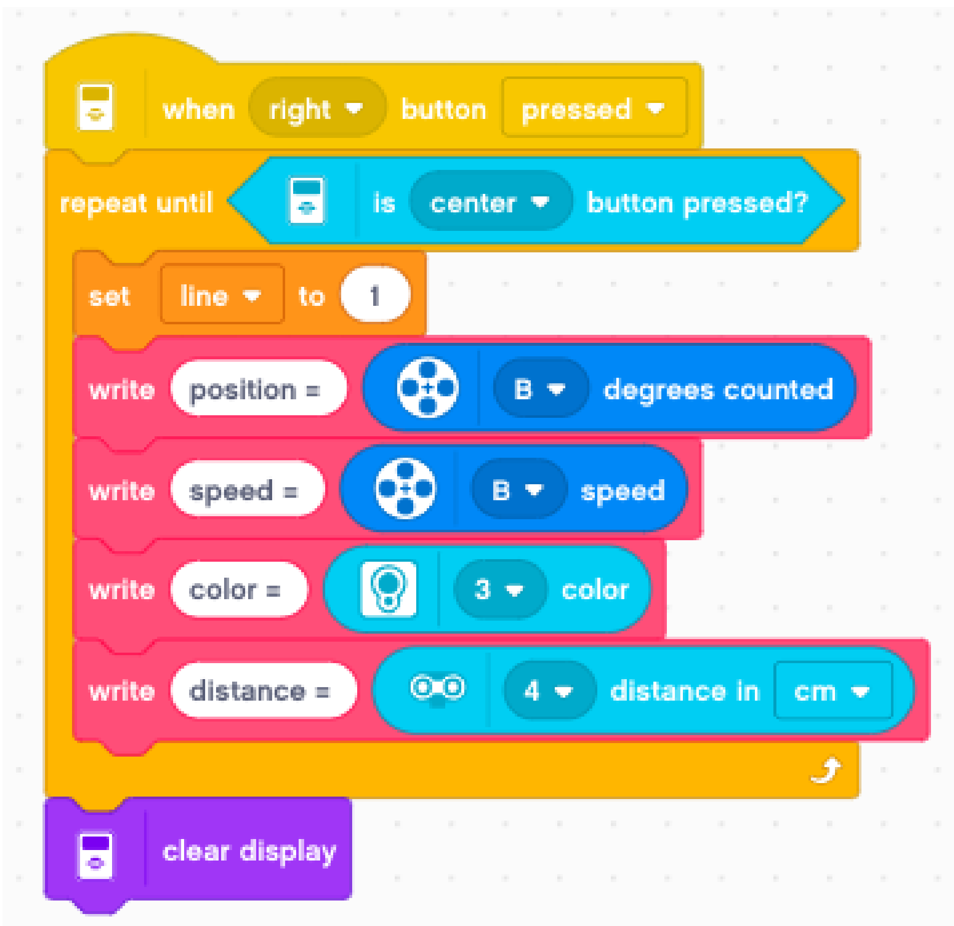
```
123
0
2
34.5
```

To better write this line of information we can define a function which:

- adds a text
- writes the number
- adds extra space after it (to erase erroneous digits)
- increments the line number



Now we can display these values with an explanatory text (position, speed, etc.)



It produces output like this:

```
position = 123
speed = 0
color = 2
distance = 34.5
```

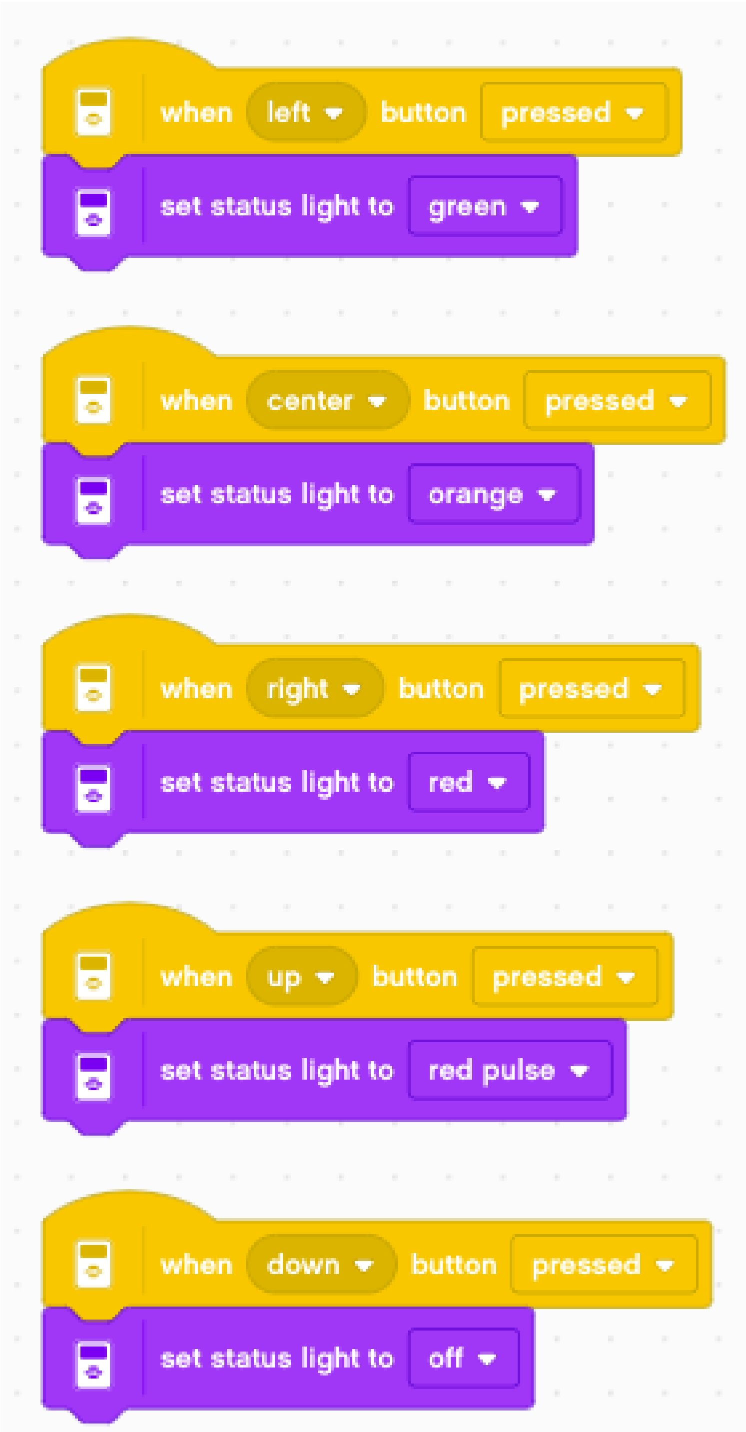
5.8 Set the status light

The status light around the buttons can be set to:

- green
- orange
- red

It also can be set to flashing mode called:

- green pulse
- orange pulse
- red pulse


when left button pressed
set status light to green
when center button pressed
set status light to orange
when right button pressed
set status light to red
when up button pressed
set status light to red pulse
when down button pressed
set status light to off

CHAPTER 6

Oscilloscope

You will be surprised that the EV3 text display can be tweaked to create an oscilloscope.

6.1 The EV3 display

The EV3 has a 178 × 128 pixel Monochrome display. The corners have these coordinates:

- top-left - (1, 1)
- bottom-left - (1, 128)
- top-right - (178, 1)
- buttom-right - (178, 128)

It can display:

- 12.8 lines of small text
- 22 characters long

The small character occupy 10x8 pixels. One character is

- 9 pixels high
- 7 pixels wide

For exemple an A is composed of these pixels:

```
1 2 3 4 5 6 7
      x
      x
    x   x
    x   x
  x       x
  x       x
  x x x x x
```

(continues on next page)

(continued from previous page)

```
x           x
x           x
```

6.2 Characters used

For the oscilloscope we are going to use the **vertical bar**:

```
1 2 3 4 5 6 7
      x
      x
      x
      x
      x
      x
      x
      x
      x
```

the **horizontal bar** (underscore):

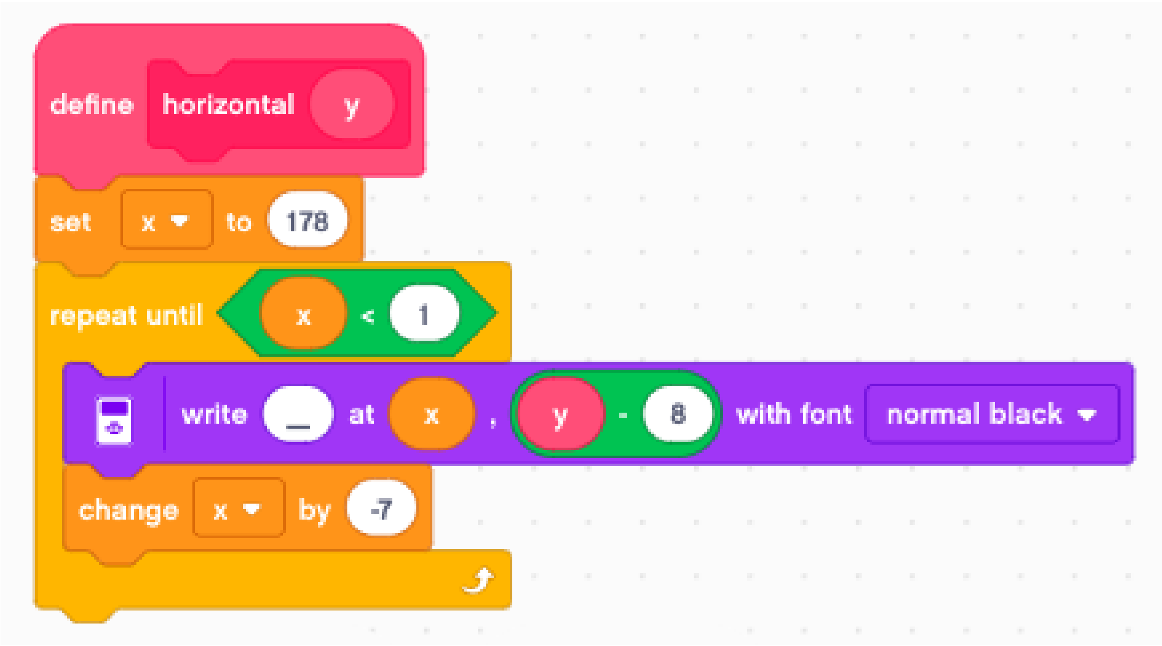
```
1 2 3 4 5 6 7

x x x x x x x
```

6.3 Display a horizontal line

With this information we are ready to display a horizontal line at position **y**. We define the variables **x** and **y**.

We write backwards. The reason for this is that the 7x9 pixel character is printed on a 8x10 pixel field. In fact the left and the top 1-pixel-wide line is erased.

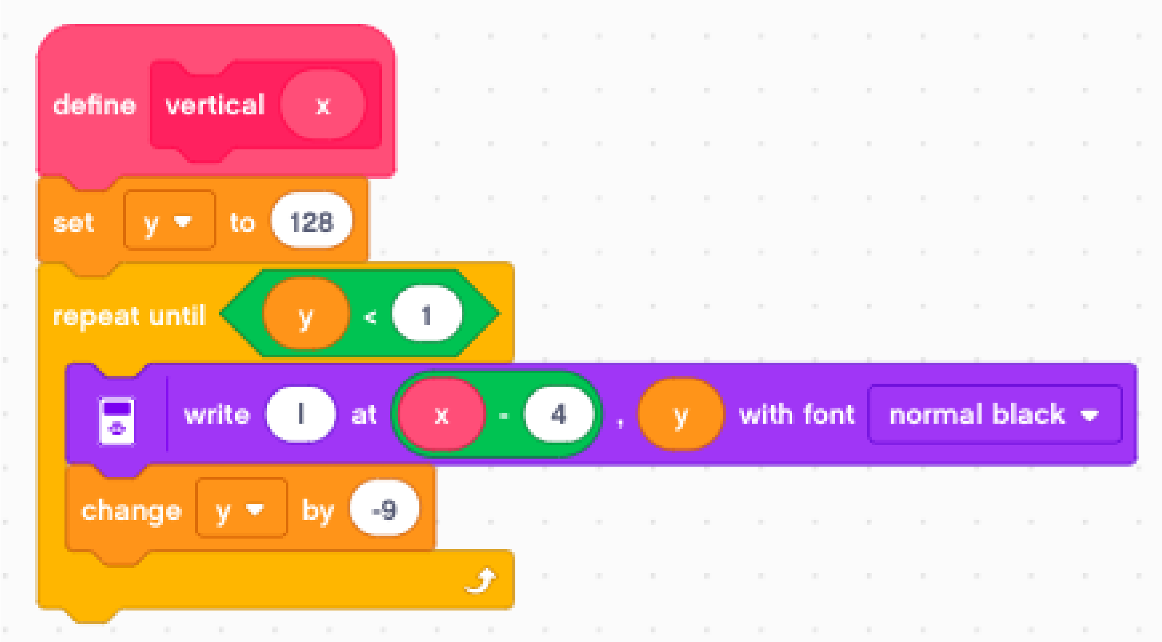
So we initialize x to 178, and decrement by the caracter width of 7. With regards to y there is a -8 pixel offset.



6.4 Display a vertical line

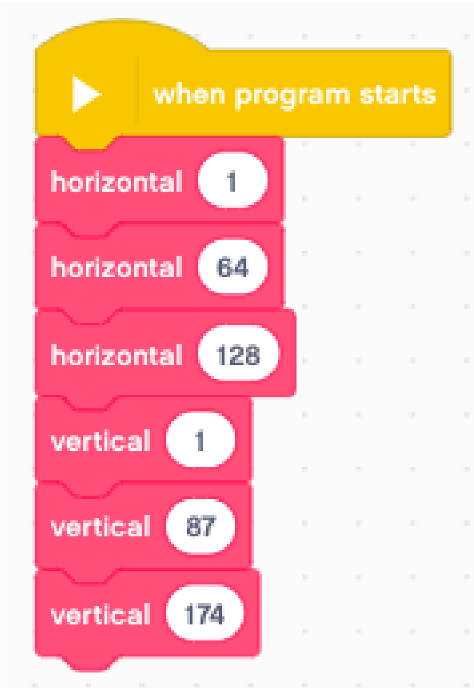
Then we define the **vertical(x)** function.

So we initialize y to 128, and decrement by the caracter height of 9. With regards to x there is a -4 pixel offset.



6.5 Display a grid

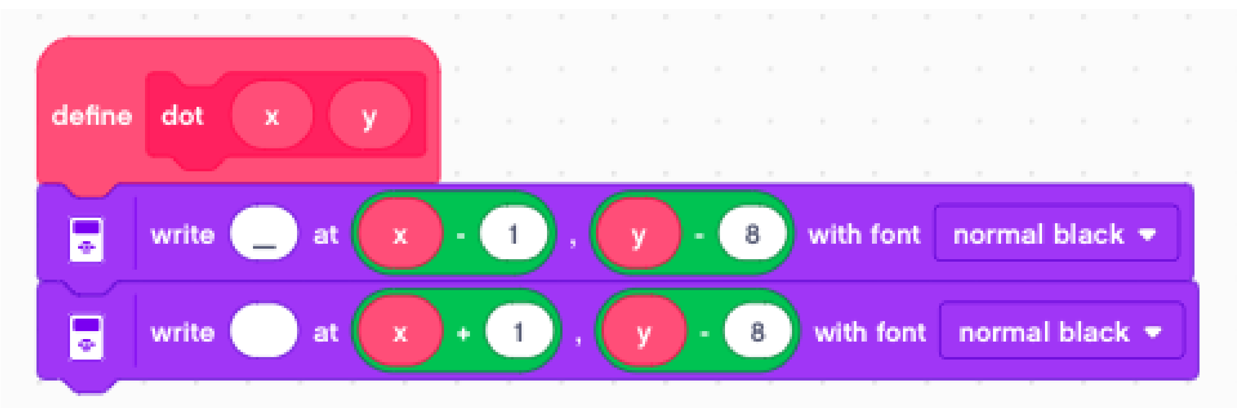
Finally we can draw a grid.



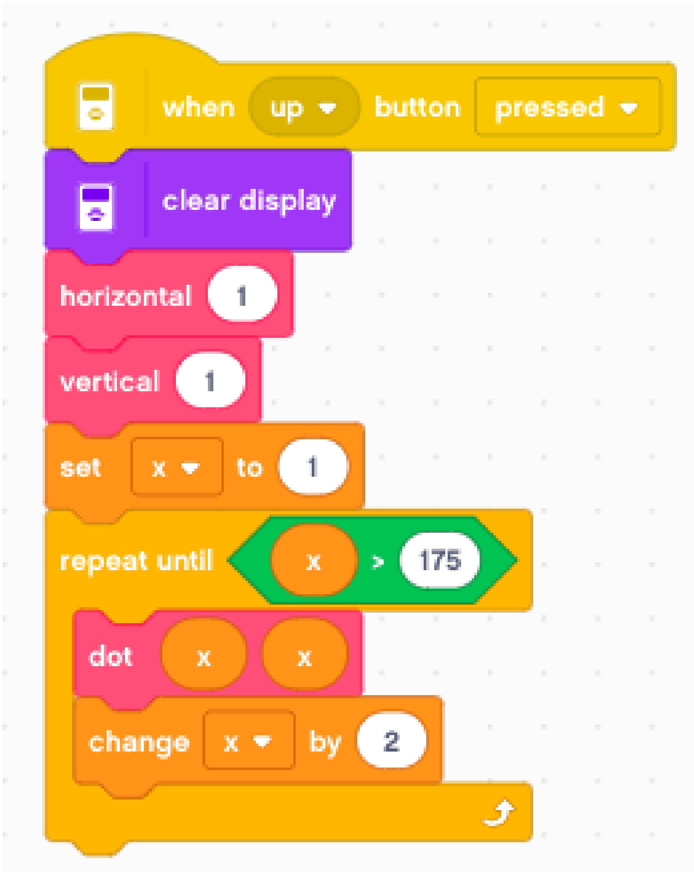
6.6 Draw a dot

The function **write text at (x, y)** can place a character at the position (x, y). To draw an arbitrary curve using dots, we could use the dot (**.**) or the hyphen (−).

But the best way is to use an underscore (_) followed by a space character. The space character is offset by 2 pixels and erases the 6 unused pixels of the underscore. This allows us to draw a dot every 2nd pixel.



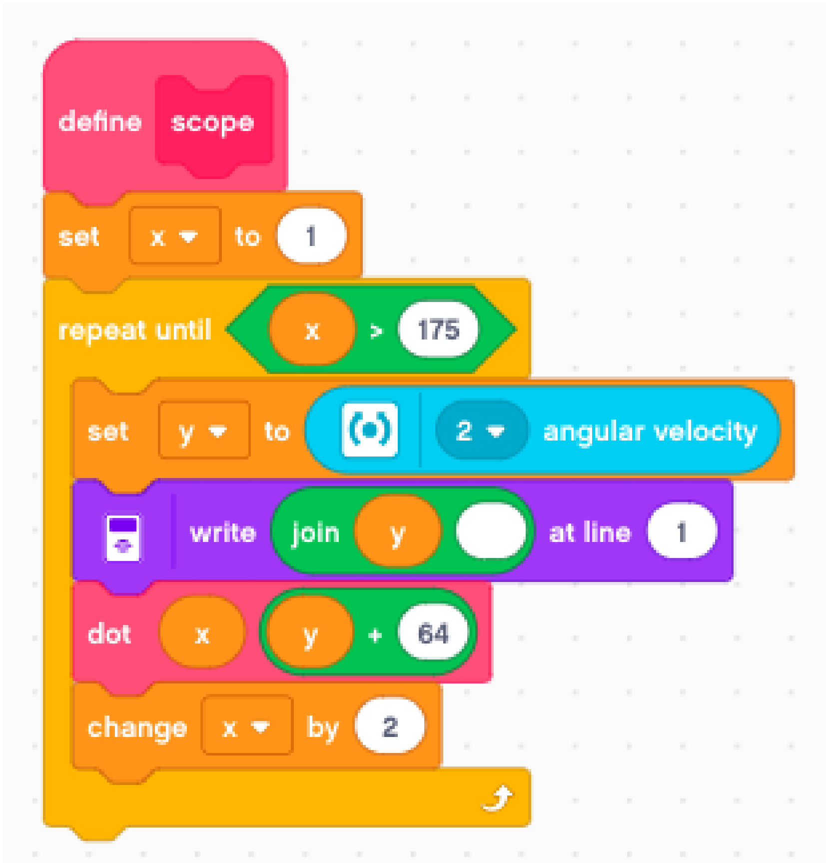
The example below draws a diagonal line starting at (1, 1).



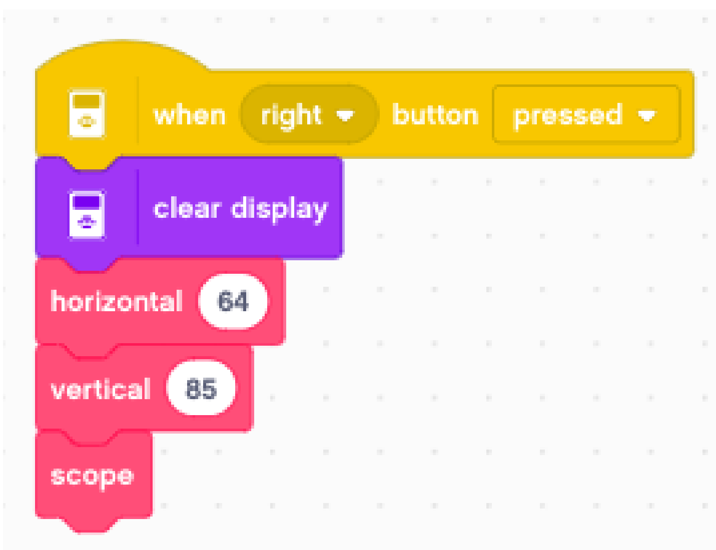
6.7 Display a scope trace

Now we have all the elements to program an oscilloscope. We start at x=1 and loop until x>175. At each iteration we increase x by 2.

The y value is the angular gyro velocity. We display it numerically on line 1. And we plot it with an offset of 64 to the screen.

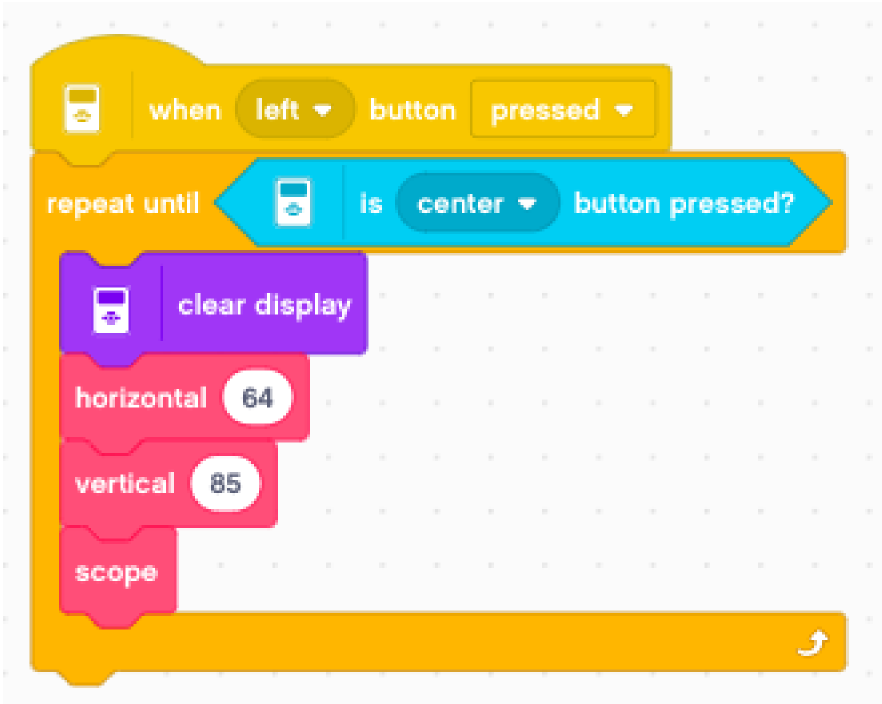


With a button press we can acquire a single trace of 88 samples.



6.8 Measure continously

We can place it inside a loop and measure continously.



You can download the programs so far: `scope.lmsp`

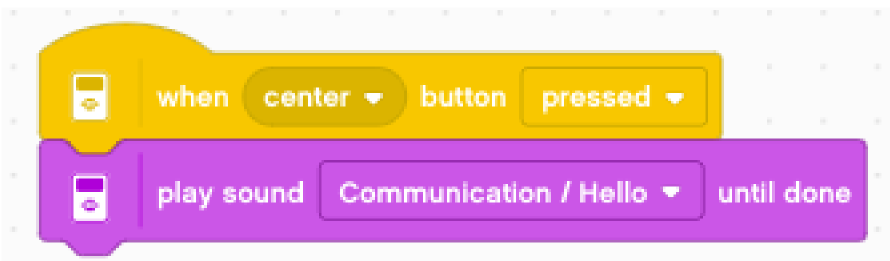
CHAPTER 7

Sound

The EV3 can also play sounds and music.

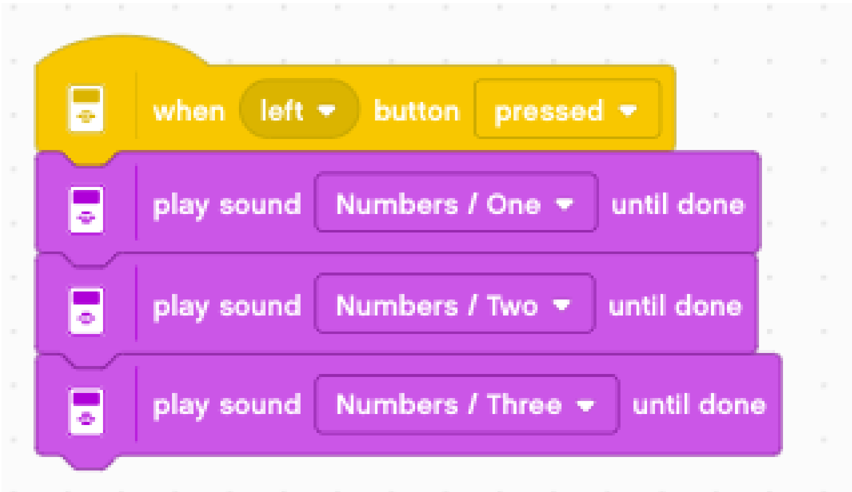
7.1 Say hello

We start with the simple program to say **hello** when pressing the center button.



7.2 Count to three

Next we create a program which counts to three.

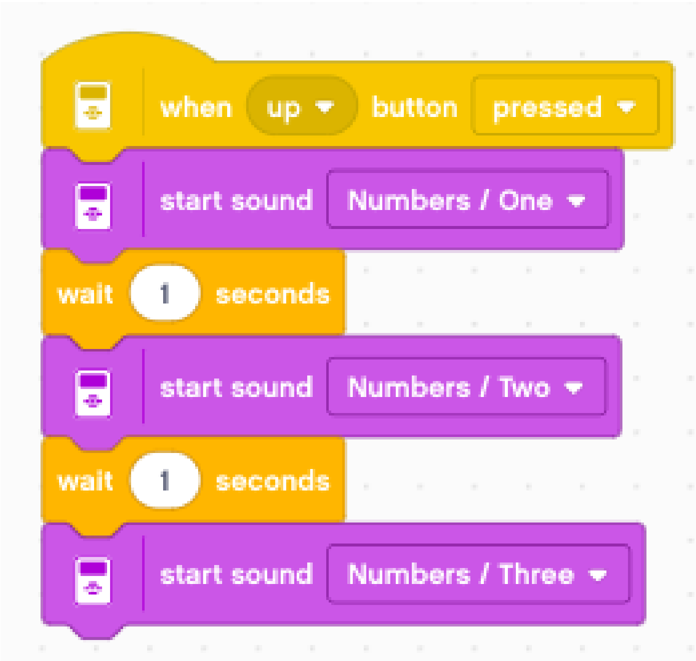


There is another block called **start sound**. What is the difference?

Try to count like this.

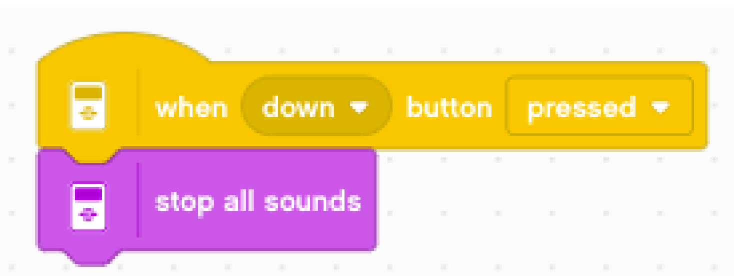
This block does not wait for the sound to finish. It starts immediately the next block and replaces the previous sound which barely has started with the new sound. Thus this program only plays the last sound (*three*).

To give the program time for the sound, we have to insert a **wait** block. This allows to play a sound precisely every second.



7.3 Stop all sounds

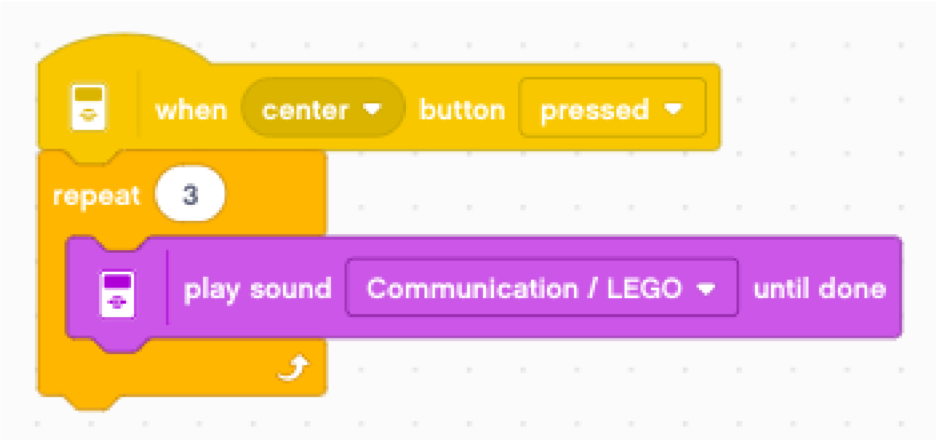
The **stop all sounds** block stops the currently running sound. If you press it while running *one, two, three*, it stops immeditly the current sound and plays the next one in the sequence.



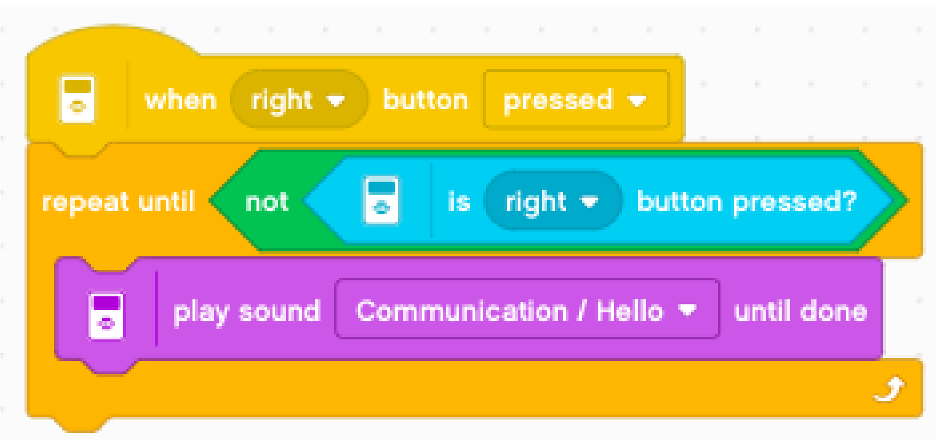
Download: `sound1.lmsp`

7.4 Repeat a sound

With a loop we can repeat a sound a given number of times. For example we can repeat the sound *LEGO* three times.



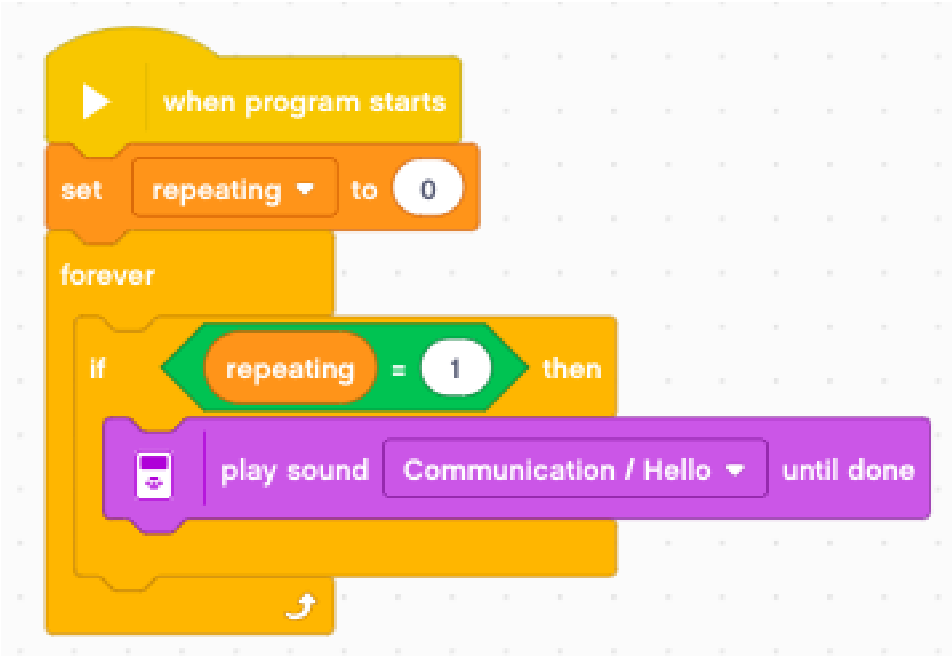
We can also repeat as long a button is pressed.



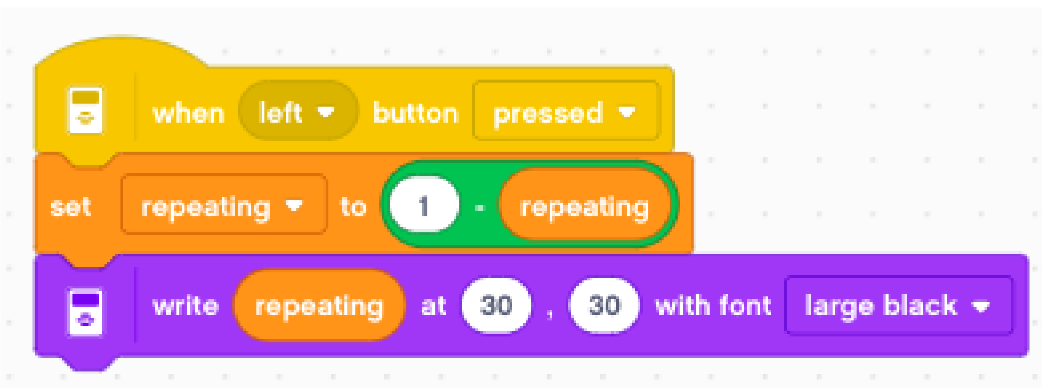
Finally it's a bit more complicated to start repetition with a first button press and stop repetition with a second button press.

We need to define a variable **repeating** which we initialize to 0. Then we enter a **forever** loop. Inside the loop we have an **if** block.

If **repeating = 1** then we play the sound.



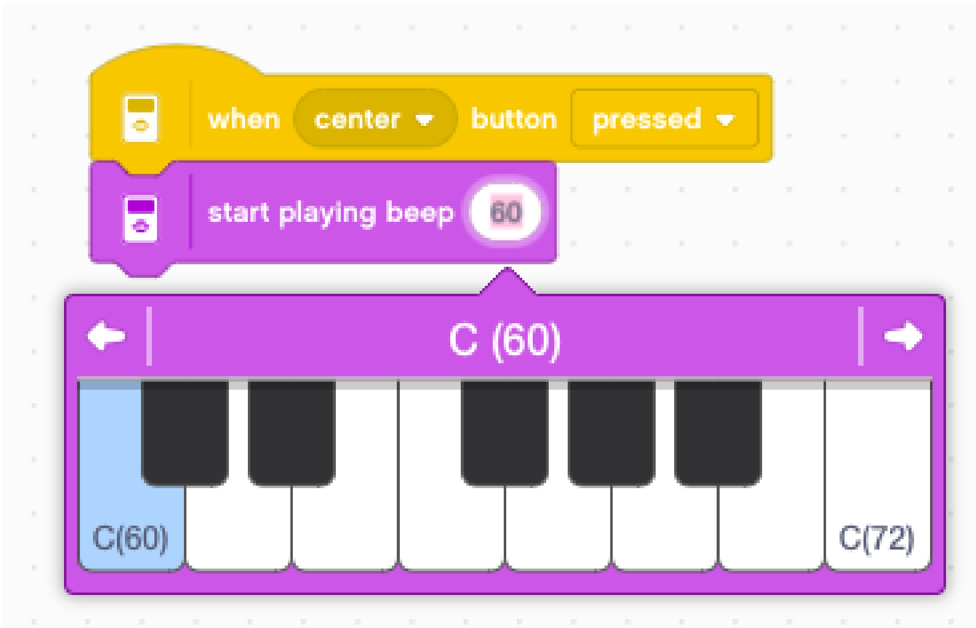
Next we program the button to toggle the variable **repeating** between the values 0 and 1. For our feedback, we also print this value to the screen.



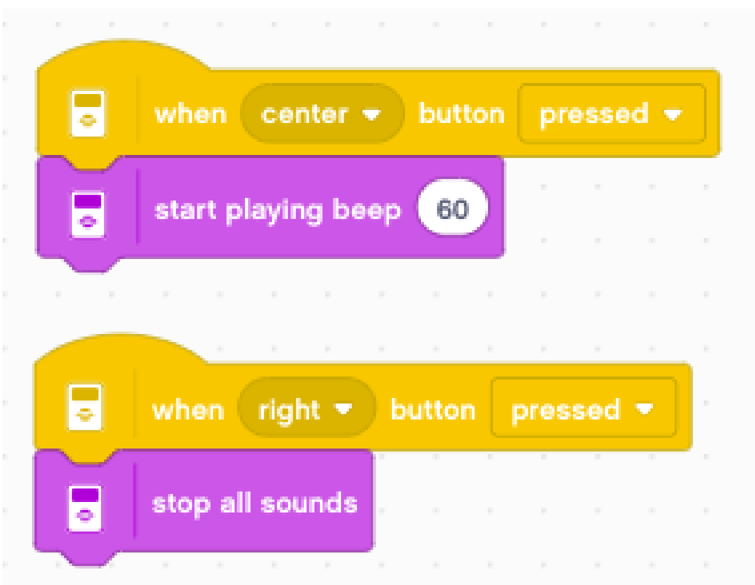
Download: `sound2.lmsp`

7.5 Start playing a beep

The **start playing beep** block starts a beep. With a keybord we can chose the pitch.



The sound will be playing continously. We can use a second button to stop the sound.



There are 3 different ways to play a beep with only 1 button:

- play a **timed** beep
- play beep **while** button is pressed
- **toggle** beep when button is pressed

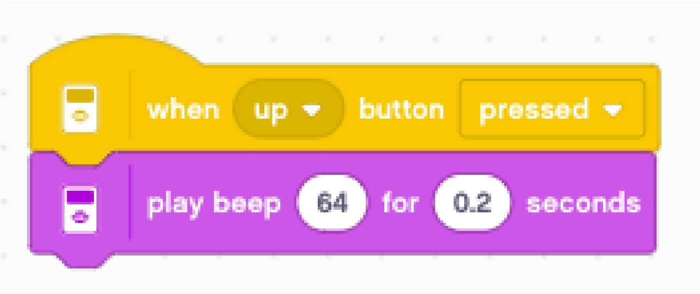
7.6 Play a timed beep

The **play bee** function has two arguments:

- pitch

- duration

It allows to give a duration to the sound. In the followign example we play the sound for 0.2 seconds.

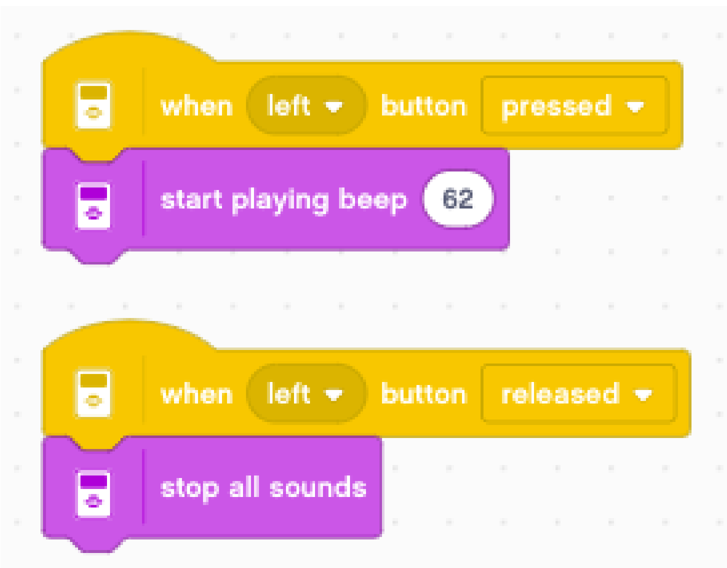


7.7 Play beep while pressed

Buttons have two associated events:

- pressed
- released

We can use these two events to program a button which plays a sound only while the button is being pressed.

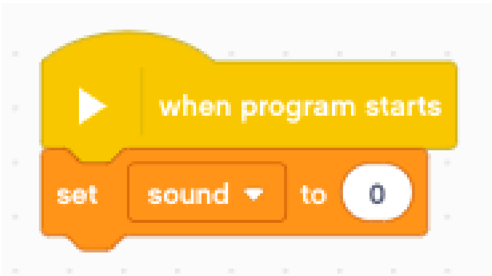


7.8 Toggle beep when pressed

The last way is an on/off toggle button. This method needs a variable **sound** which is going to store the state of the sound:

- 0 = sound is on
- 1 = sound is off

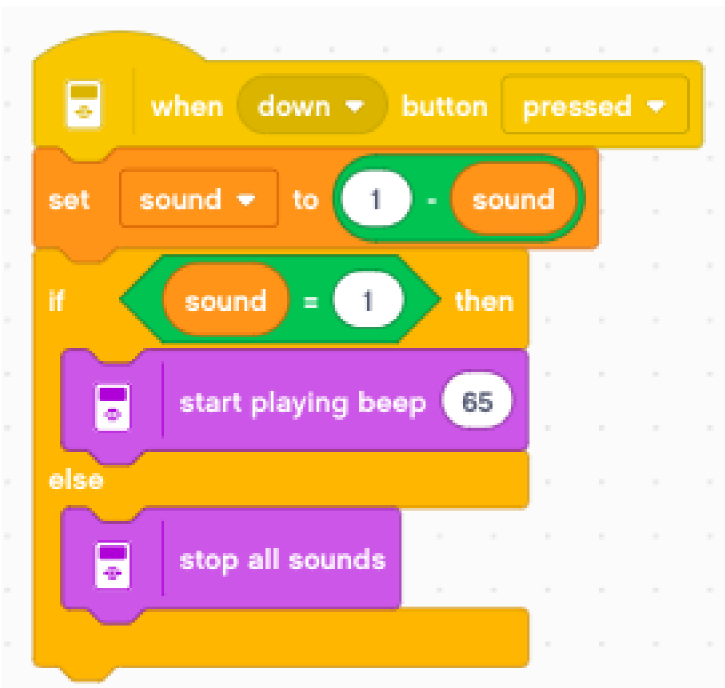
We initialize the variable **sound** to 0 (off) at the start.



When the button is pressed, we toggle the variable **sound** by using the expression **sound = 1-sound**

Then we enter an **if-else** block:

- if sound = 1 (off) we start playing
- if sound = 0 (on) we stop playing

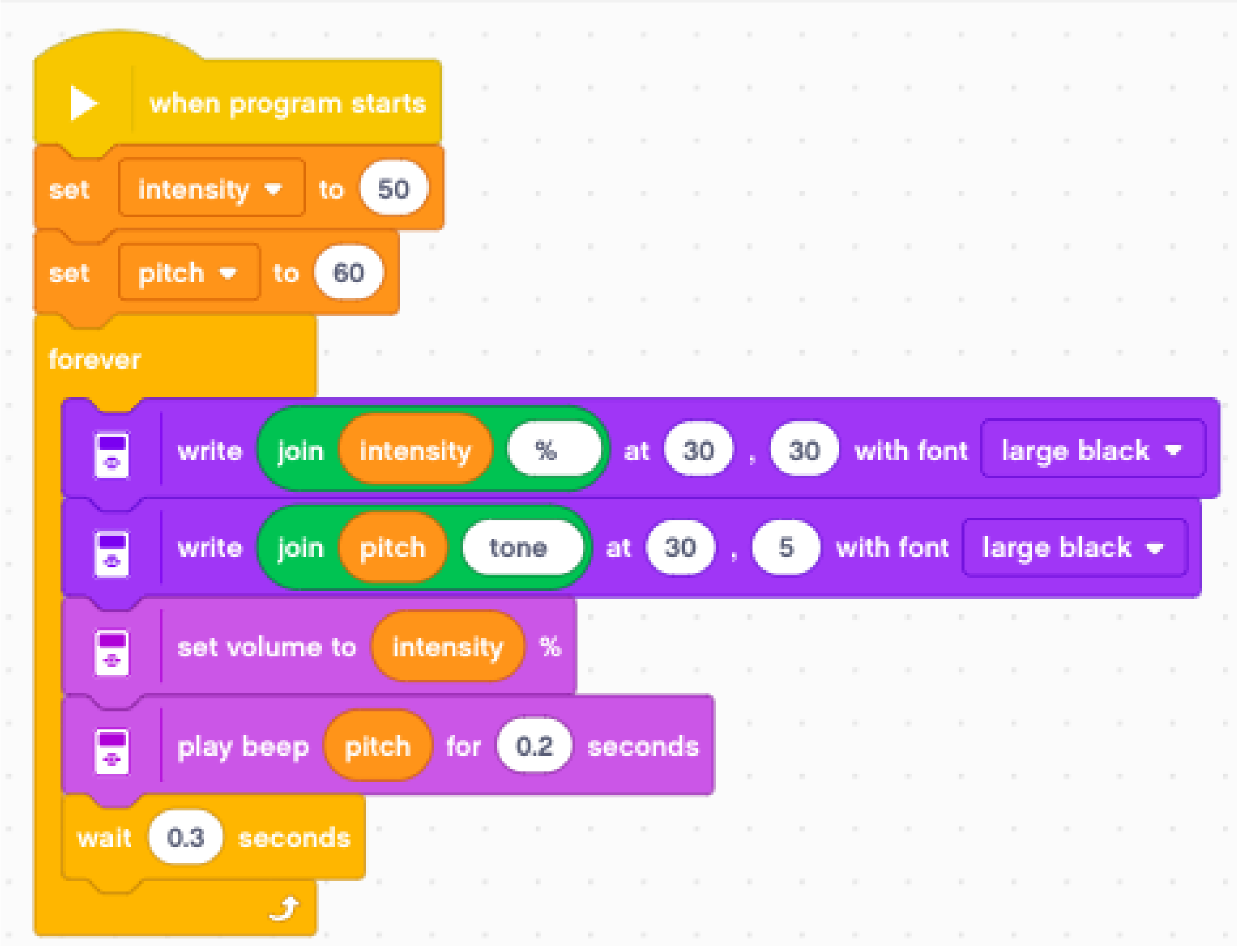


Download: `sound3.lmsp`

7.9 Change volume and pitch

We can control the volume and pitch of of a sound. First we start by creating two variables called **intensity** and **pitch**.

We set intensity to 50 and pitch to 60. Then we enter a loop where we first display these two values to teh screen. Then we produce a short beep repeating every 0.5 second.



Now we can use the 4 buttons to change the two variables **pitch** and **intensity**.

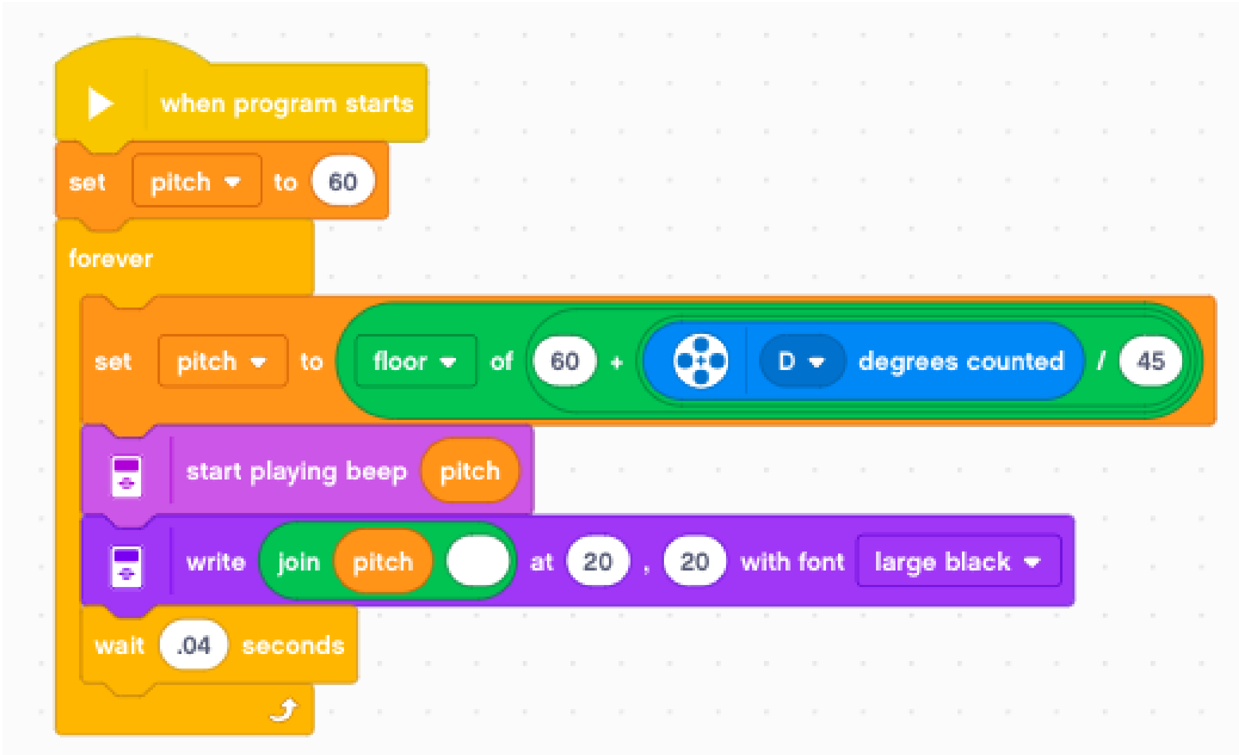
The pitch has been initalized to the value 60. These numbers correspond to:

```
60 C
61 C#
62 D
63 D#
64 E
```

7.10 Use the rotary encoder

We can use the rotary encoder to change pitch. In get the pitch in half-tone steps we:

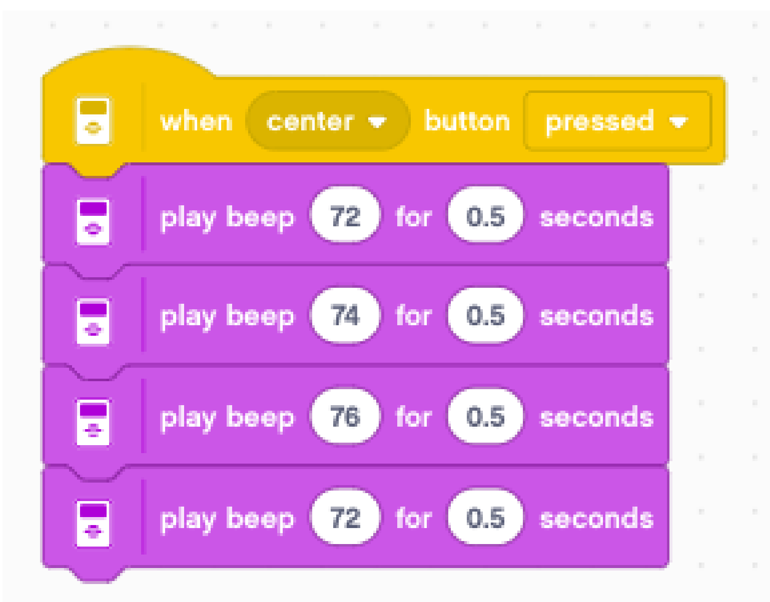
- divide by 45 to adjust sensitivity to 45° steps
- offset by 60 to start with the C
- take the floor to get integers (half tones)



7.11 Play a melody

We can play beeps in sequence to play a melody. For example to play the music of this famous French folk song **Frère Jacques**

It is quite straightforward to program the first measure. If we want 120 beats per minute (120 bpm) each beat must be 0.5 seconds.



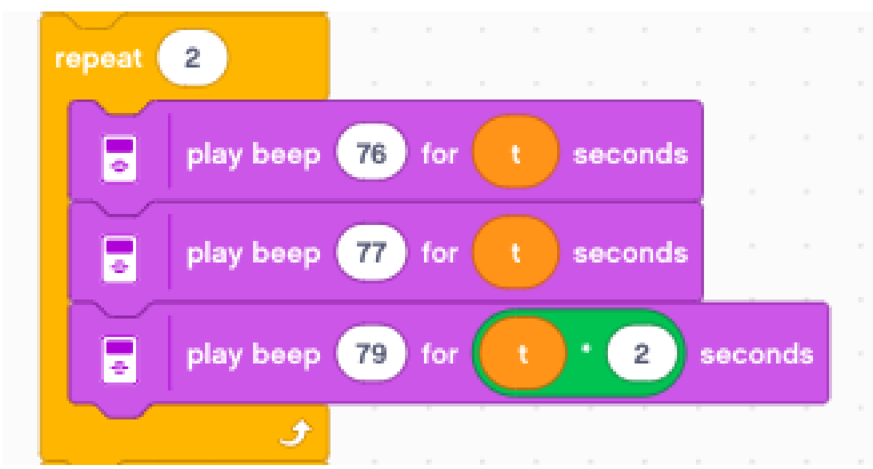
7.12 Change the tempo

If we want to change the tempo, then it would be better to code the duration of the beep with a variable. We create the variable **t** (time) and initialize it to 0.5 seconds. Also, we repeat the first 4 notes in a loop.

With the **up/down** buttons we can select the tempo.

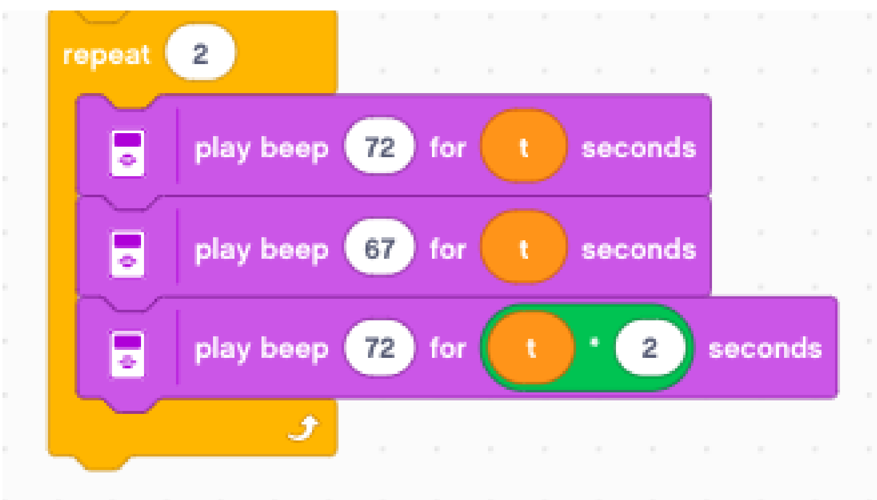
7.13 Short and long notes

Not all the notes have the same duration. The white ones are twice as long. We use the expression `t*2` as the duration.



On the other hand some other notes only have half the lenght. We use the expression `t/2` for their duration.

And this is the final part.



Download: `music.lmsp`

CHAPTER 8

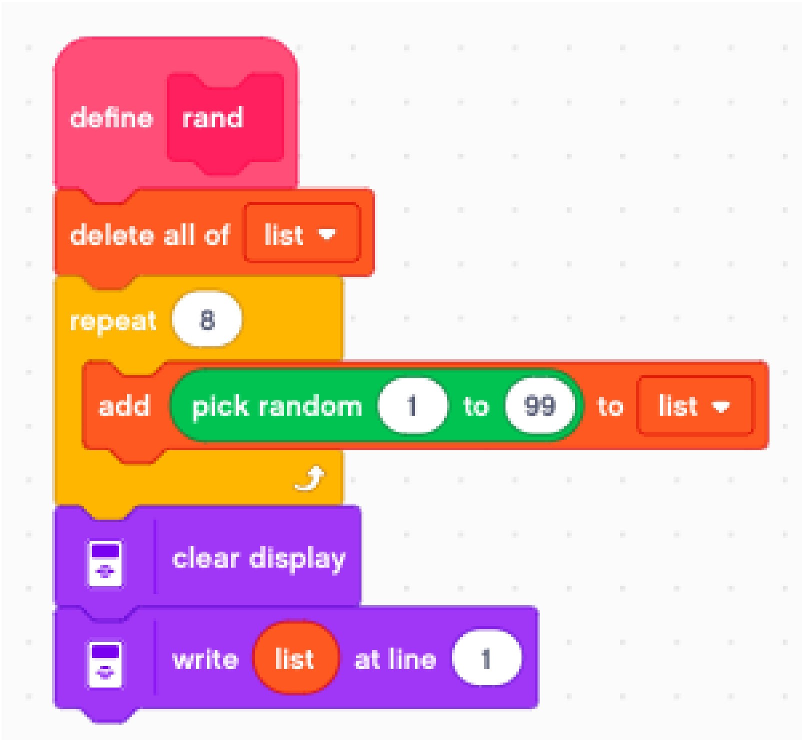
Statistics

In this section we look at a list of numbers and calculate

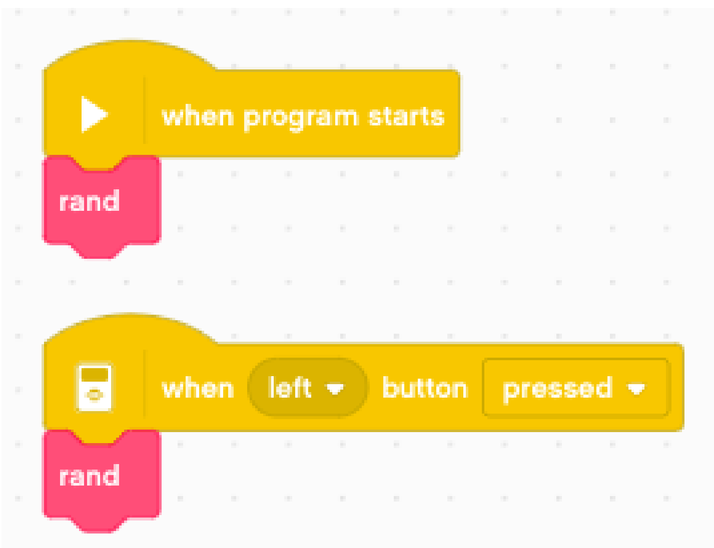
- minimum value and its position
- maximum value and its position
- sum
- average

8.1 Random list

For this exercice we use a list with 8 random numbers between 1 and 99. This way we can print them on the first line of the display.



We call this function in the **start** event and also with the **left** button.



The result of this function looks like this:

```
86 35 49 54 37 6 93 62
```

8.2 Calculate the minimum

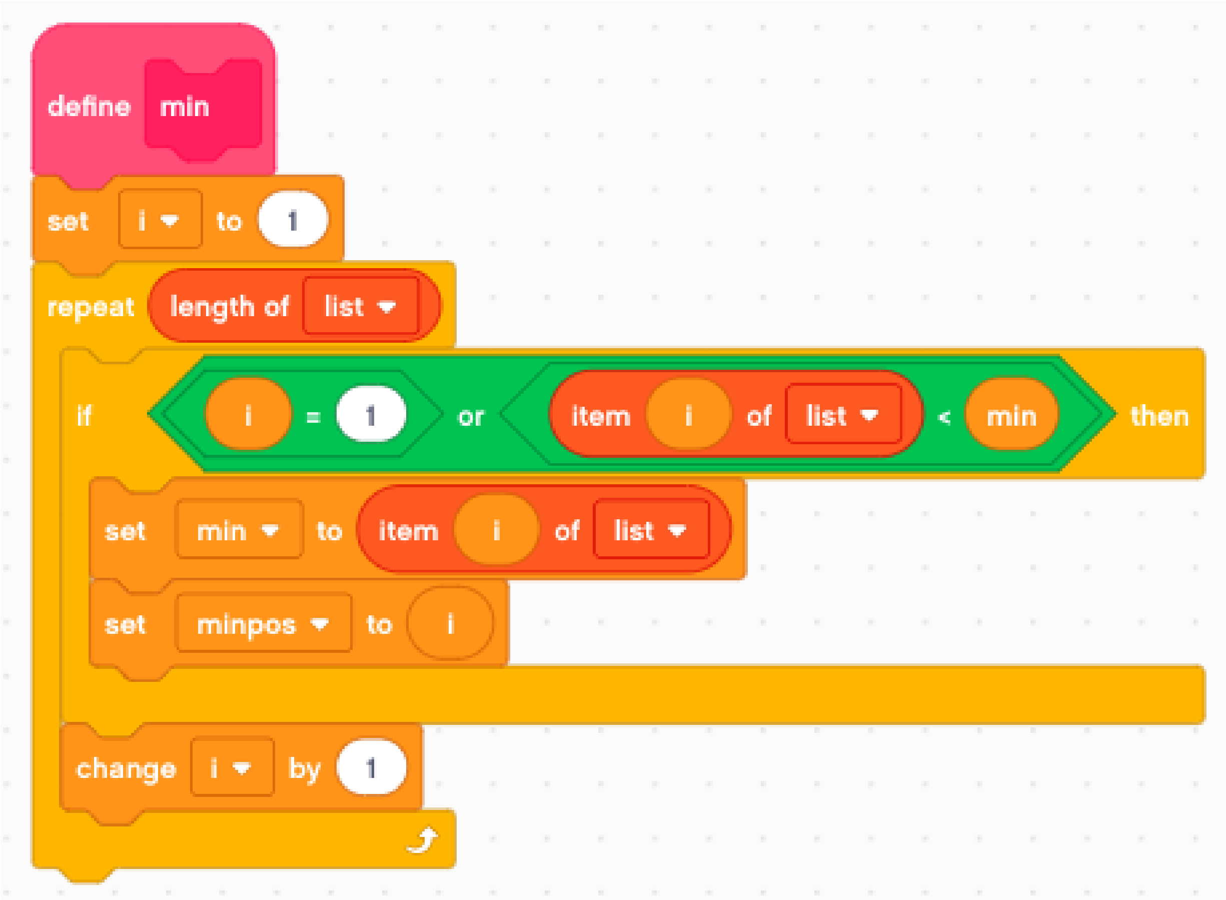
At the first iteration **i=1** we set

- min = list[1]

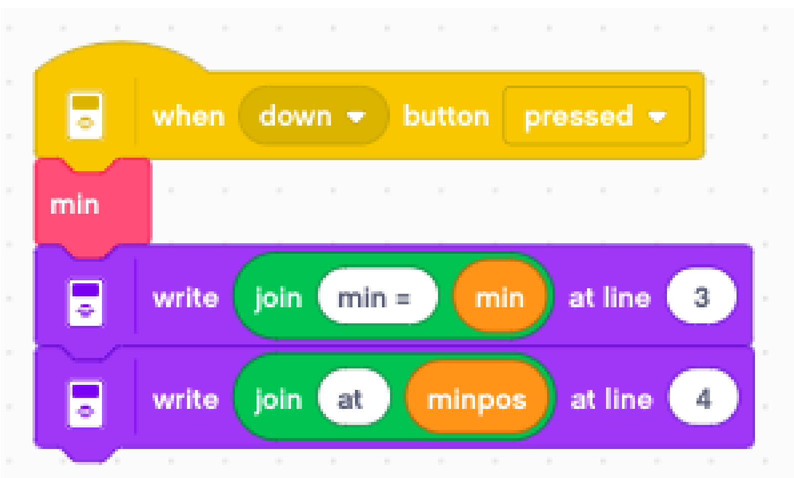
- minpos = 1

Then we iterate through the rest of the list. If we find a number which is smaller, we take it as the new minimum.

- min = list[i]
- minpos = i



We call this function with the **down** button.



The result of this function looks like this:

```
86 35 49 54 37 6 93 62

min = 6
at 6
```

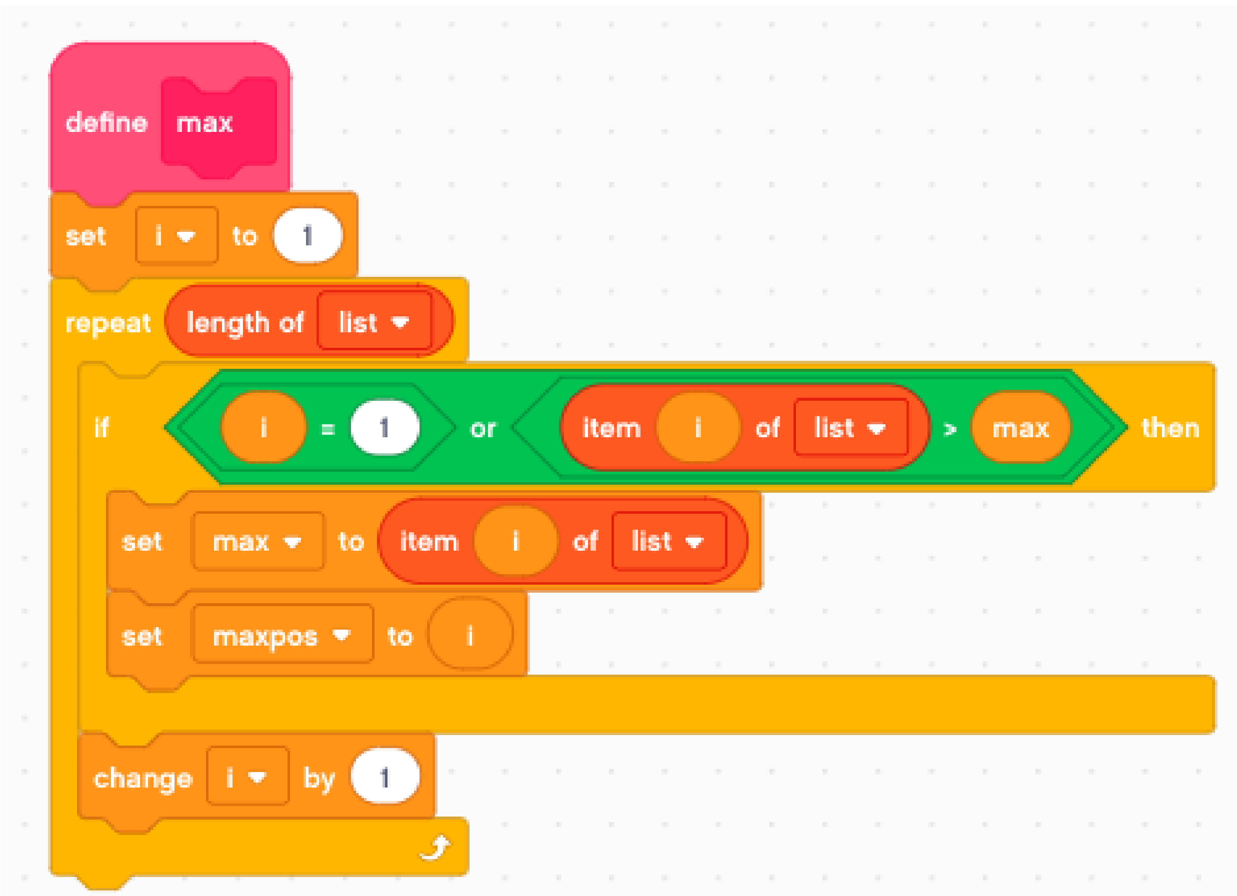
8.3 Calculate the maximum

Again, at the first iteration **i=1** we set

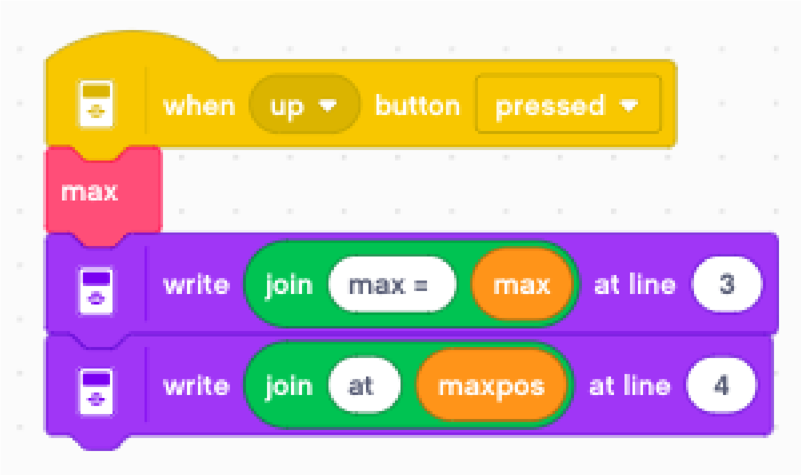
- max = list[1]
- maxpos = 1

Then we iterate through the rest of the list. If we find a number which is larger, we take it as the new maximum.

- max = list[i]
- maxpos = i



We call this function with the **up** button.

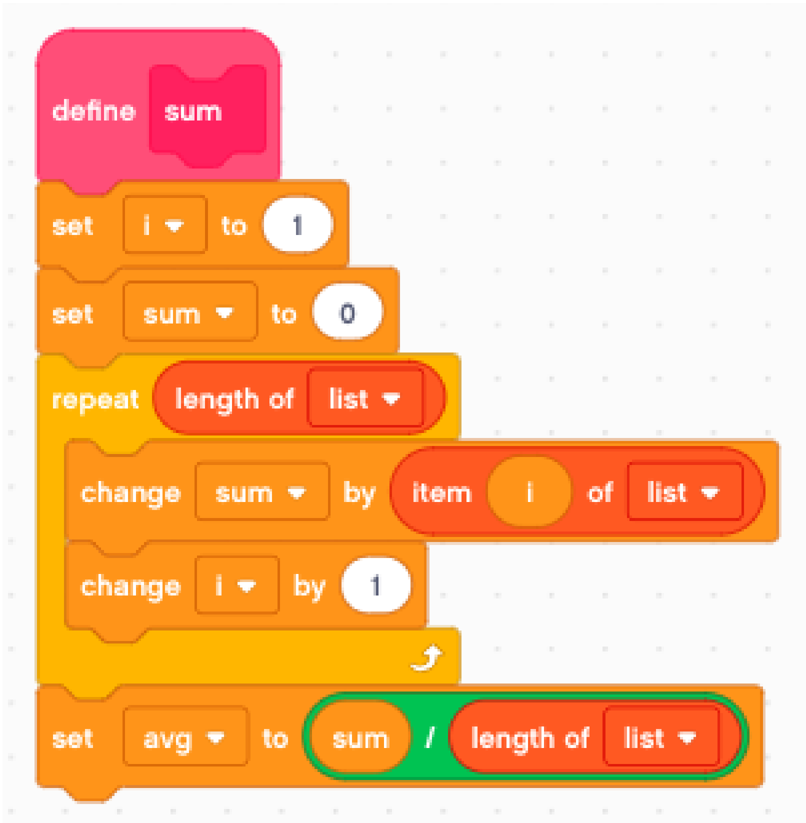


The result of this function looks like this:

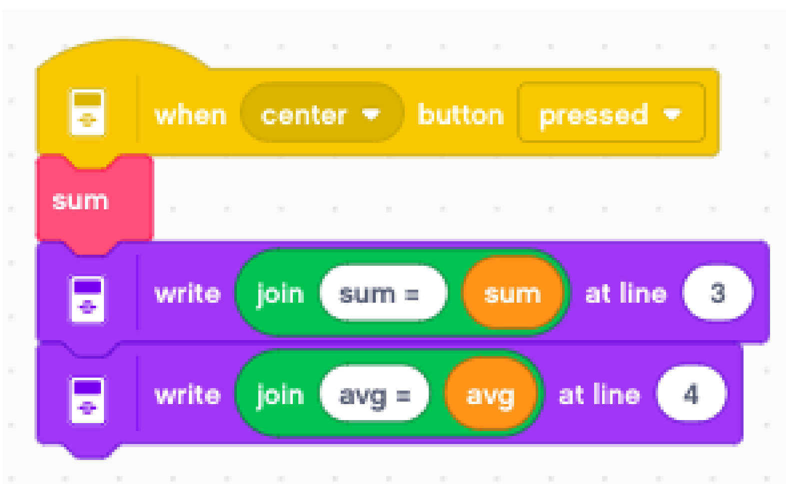
```
86 35 49 54 37 6 93 62

max = 93
at 7
```

8.4 Calculate sum and average

To get the sum we add all elements of the list together. The average is obtained by dividing the sum by the number of elements.



We call this function with the **center** button.



The result of this function looks like this:

```
86 35 49 54 37 6 93 62

max = 422
avg = 52.75
```

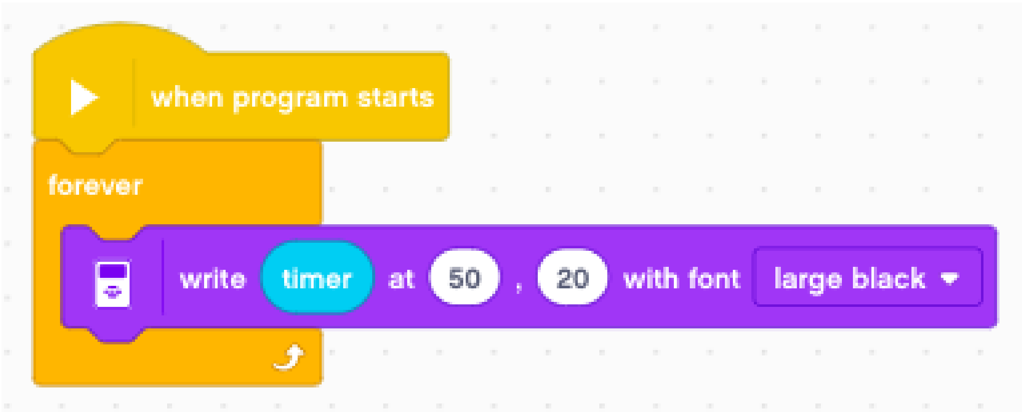
CHAPTER 9

Timer

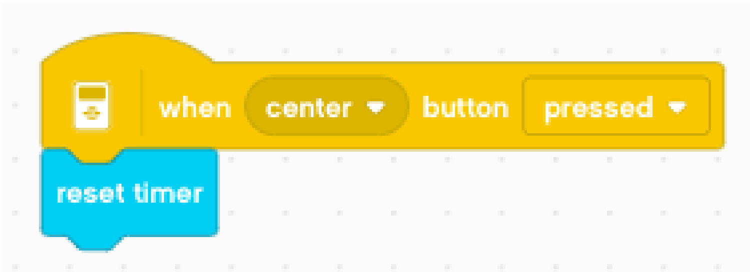
The EV3 has a timer which starts counting when the program starts.

9.1 Display the timer

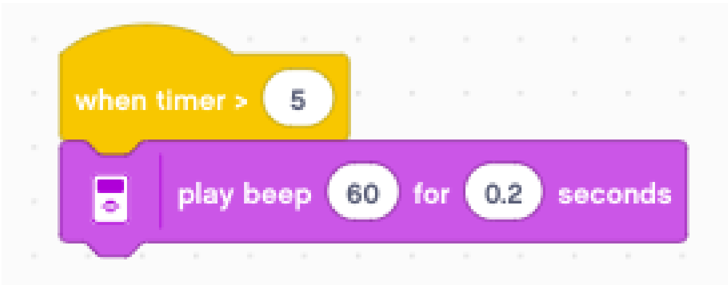
The timer is a varible which is incremented by the micro-processor. It tells the time in seconds since start-up or the last timer reset. It has milli-second precision.



The **reset timer** function sets the timer back to 0.



The **when timer** event activates a single event when the timer crosses the given threshold.



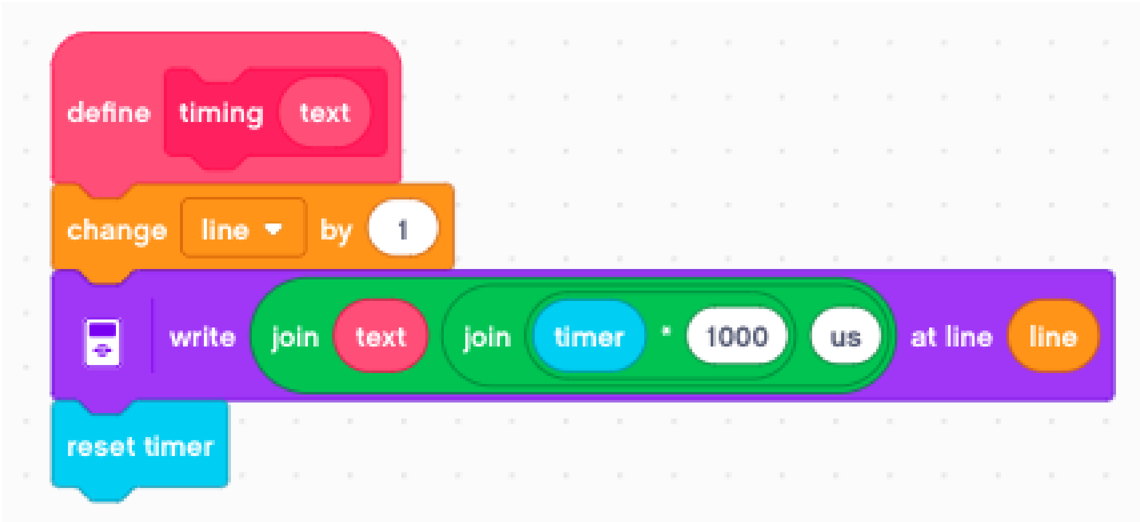
9.2 Record intermediate times

We can record intermediate times and write them to the screen.

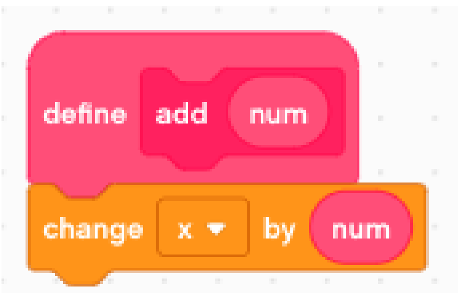
9.3 Measure EV3 speed

Now we can measure how much it takes for the EV3 to execute its operations. The idea is to repeat a function 1000 times in a loop to have a good precision. Let's define a function **timing** which:

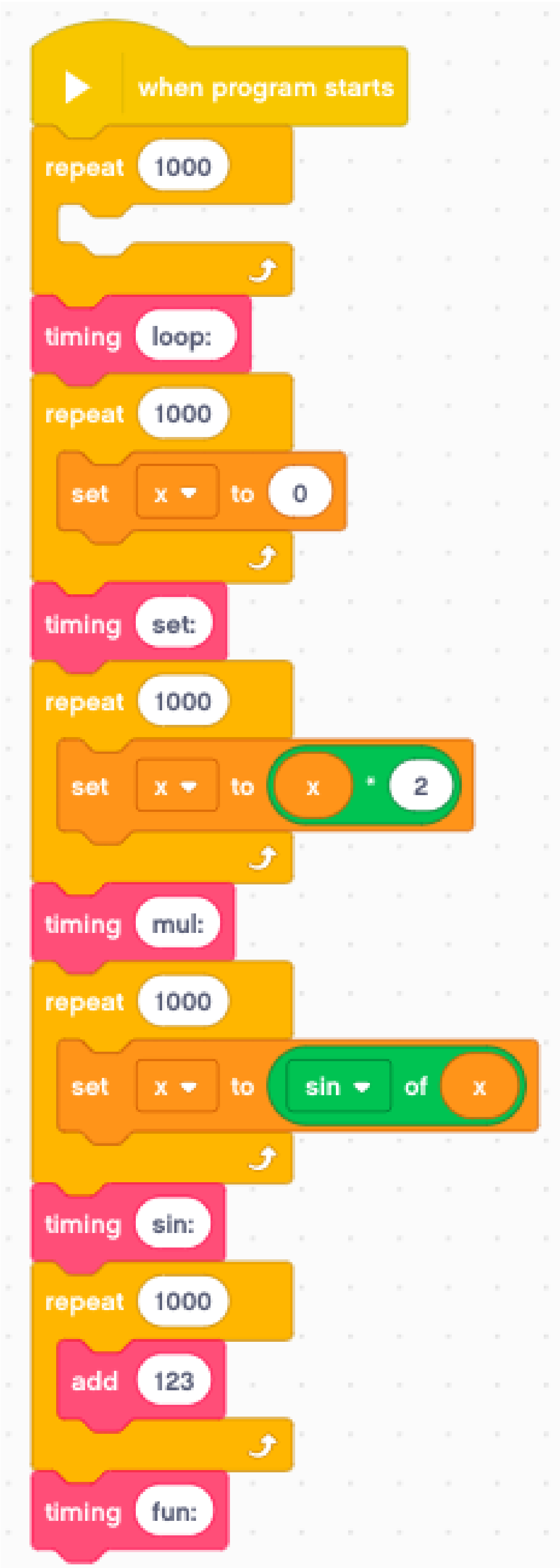
- increments the line number
- writes a text
- multiplies the **timer** with 1000 (to obtain microsecons)
- add *us*
- reset the timer for the next mesasurement



We will also use a self-defined function **add**.



These are several loops used to make the timings.


when program starts
repeat 1000
timing loop:
repeat 1000
set x to 0
timing set:
repeat 1000
set x to x * 2
timing mul:
repeat 1000
set x to sin of x
timing sin:
repeat 1000
add 123
timing fun:

This is the result:

```
loop: 43 us
set: 62 us
mul: 92 us
sin: 101 us
fun: 927 us
```

This gives us a rough idea how long different blocks take to execute:

- 43 us for a loop
- 20 us for a set (variable assignment)
- 30-40 us for a math operation (add, mul, sin, etc.)
- 900 us for a user-defined function (My block)

While basic operations take 50-100 us, the user-defined functions have a 20-times overhead.

9.4 Kitchen timer

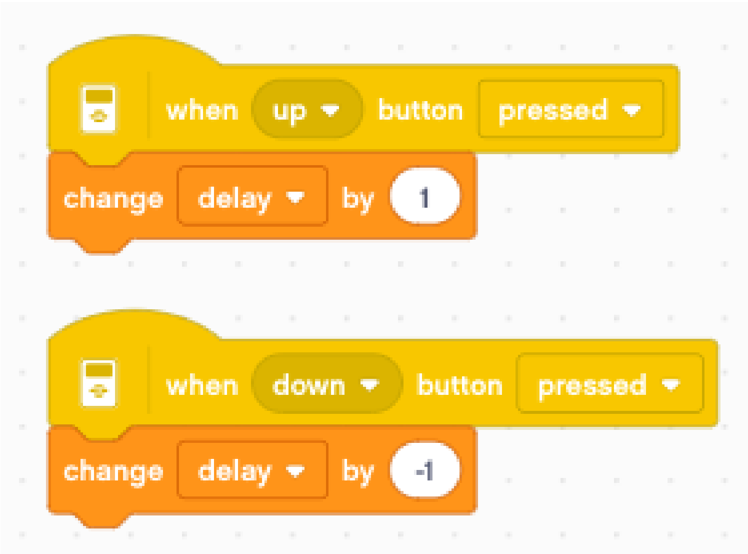
To program this timer we will use the technique of the **state machine**. In our case we have 3 states:

- reset
- count
- alarm

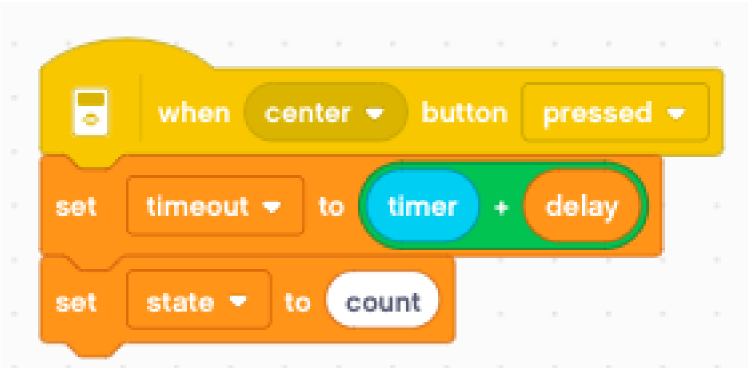
We define 3 variables:

- **delay** is the duration of the count-down in seconds
- **state** is one of the strings *reset, count, alarm*
- **timeout** is the point in time of the alarm

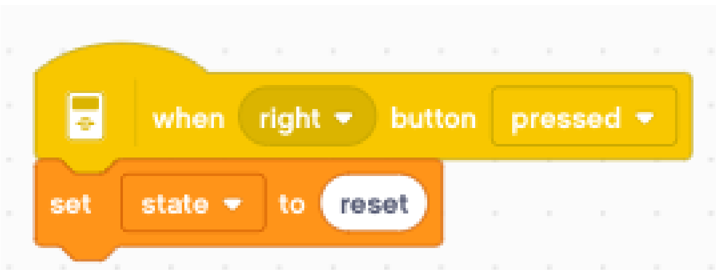
In the **reset** state we set the delay with the **up/down** buttons.



With the **center** button we set the **timeout** to **timer + delay** and switch to the **count** state.

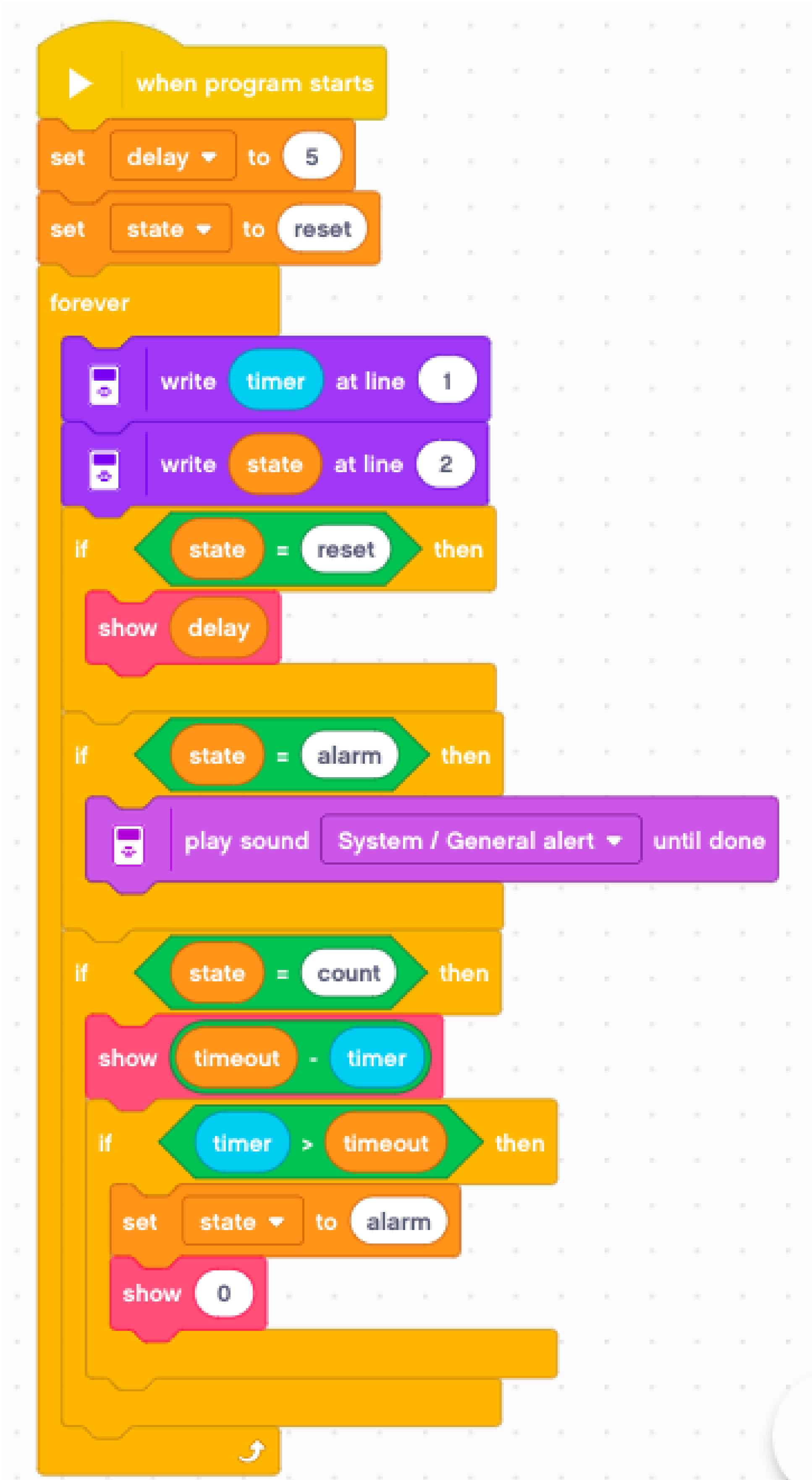


With the **left** button we stop the alarm sound and return to the **reset** state.



The state machine consists of a **forever** loop with 3 *if** blocks which check for the 3 states.

- in **reset** state, we just show the delay
- in **count** state, we show the count-down
- in **alarm** state, we repeat the alarm sound


when program starts
set delay to 5
set state to reset
forever
write timer at line 1
write state at line 2
if state = reset then
show delay
if state = alarm then
play sound System / General alert until done
if state = count then
show timeout - timer
if timer > timeout then
set state to alarm
show 0

Notice here, that we never need to reset the timer. This can be important in timing applications for not losing precision. The timer is displayed in line 1 and the state in line 2.

CHAPTER 10

Clock

In this section we build and program a coockoo clock with 2 hands.

CHAPTER 11

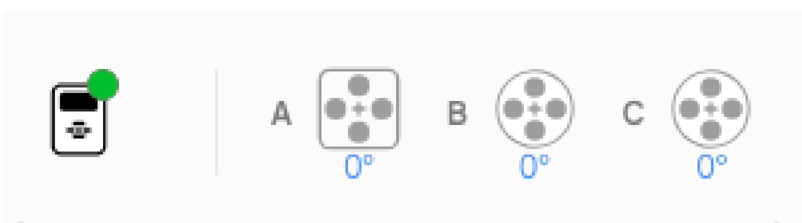
Drawing robot

In this chapter we build a drawing robot.

This robot uses three motors:

- the small one to lift the pen
- the large ones to move

To have a higher precision, it uses the small wheels. The pen is place right in the center between the two wheels.

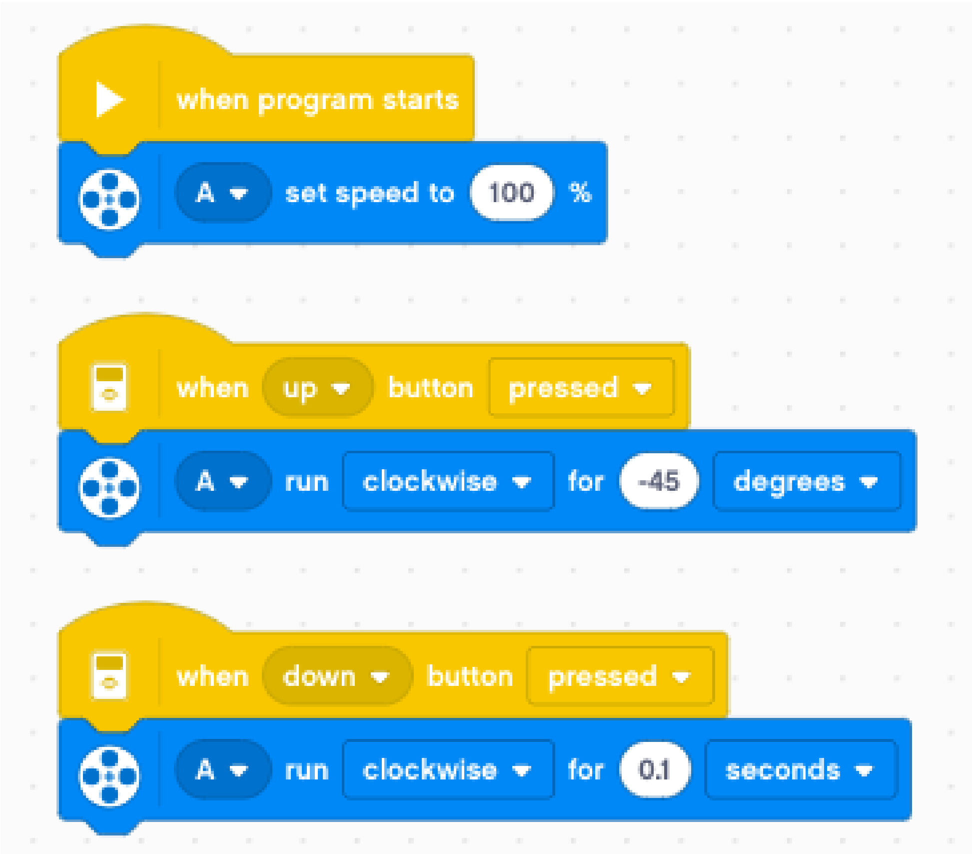


11.1 Lift the pen

Try to turn up the small motor lever to the verticl. If the horizontal postion was 0°, it will be -90°.

The angle decreseases as we lift the pen. We can now program the **up/down** buttons to move the pen. As we want to make this movement as quick as possible, we set the speed to 100%.

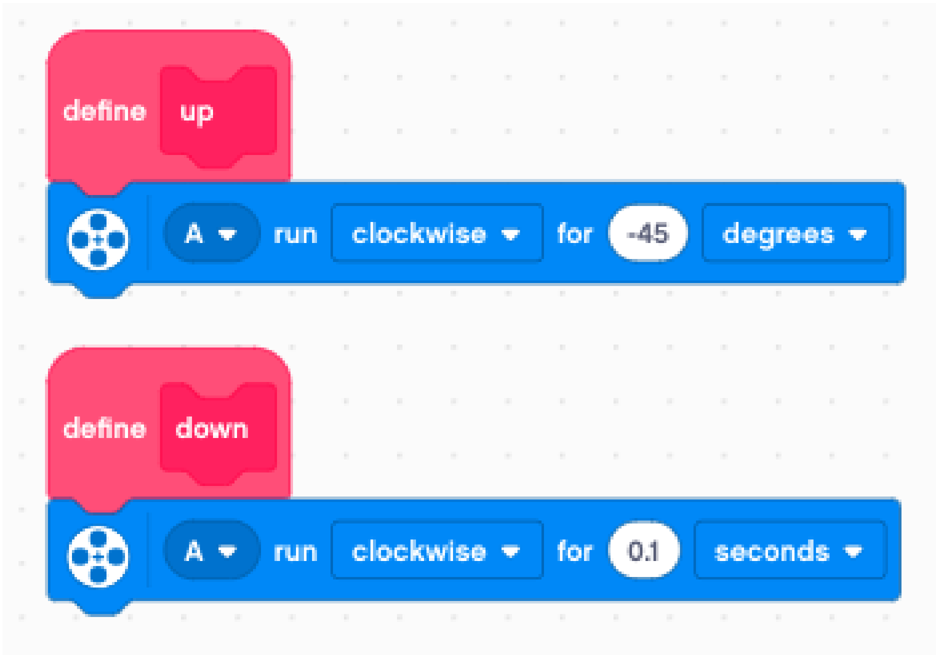
For the down movement we set a time. This is necessary, as we let the motor hit a mechanical limit. This trick is a calibration without using a sensor. We move the motor to a known position.



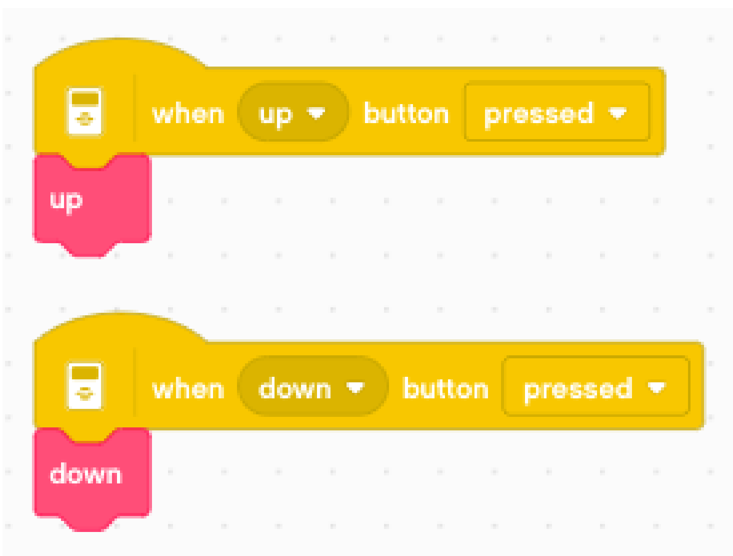
- up: move by 45°
- down: move during 0.1 seconds

11.2 Define functions

A program becomes much readable and versatile when using function. Let's define two functions **up** and **down**



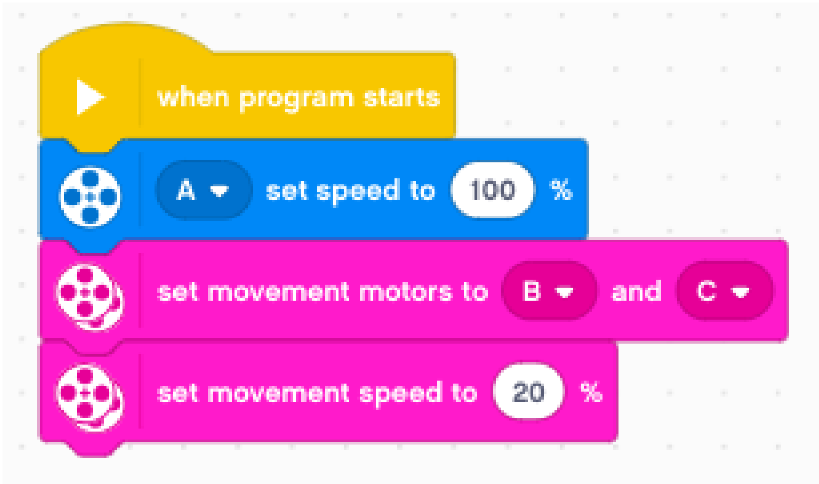
Now we can use this two functions and associate them with the buttons. The code is much more readable.



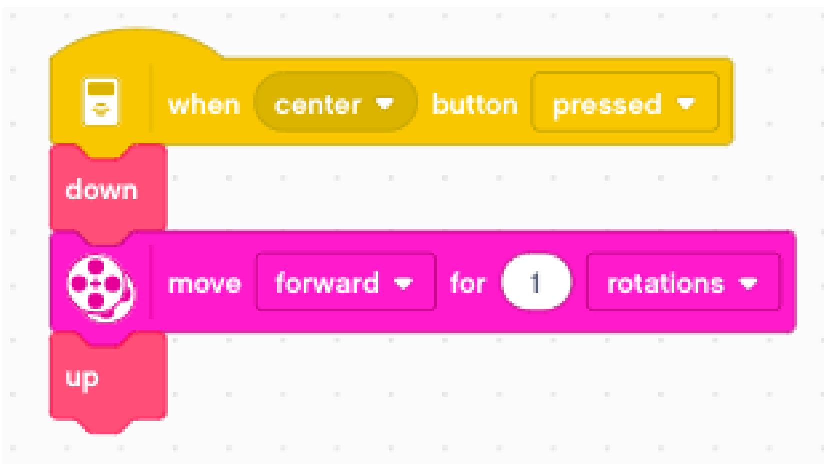
But there is another advantage. In a large program, we may use the pen in many places. If we change the pen mechanics, or correct a bug with the pen up/down movement, there is one single place to make such a correction

11.3 Move the robot

We go no to moving the robot. We are going to use motors B and C for movement. In order to obtain precise drawing results, we set the speed to 20%.



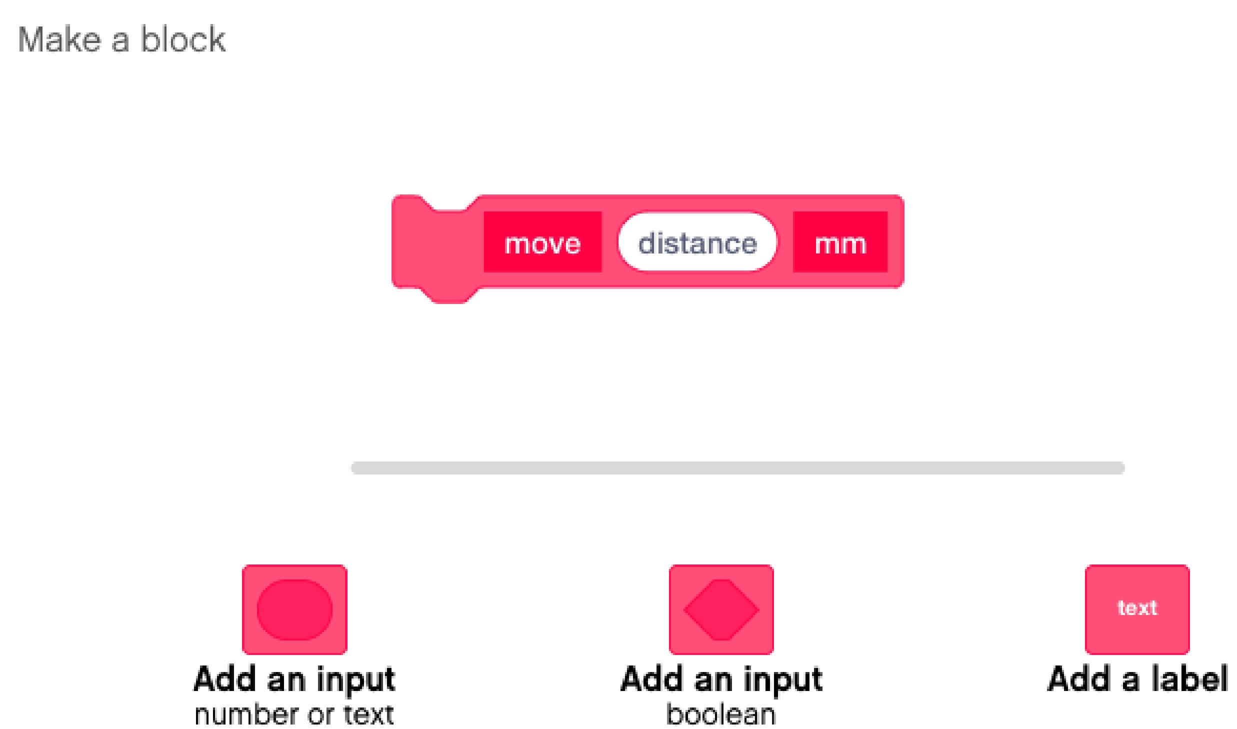
So how much does the robot advance with 1 rotation ? It is difficult to measure from the robot, but it becomes easy if the robot is going to draw a line.



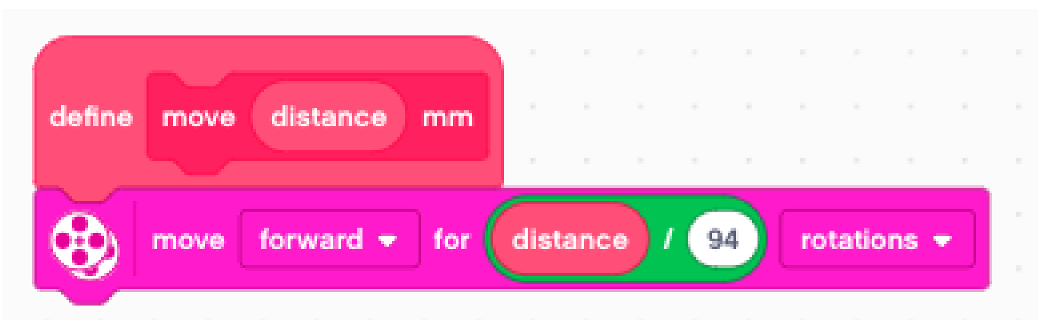
Now you can measure the line. It is about 94 mm long.

11.4 Create a move function

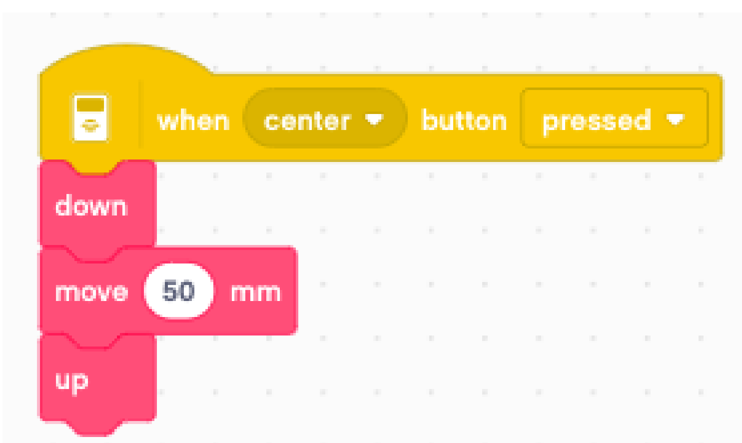
Now we have all the information to create a **move** function with an argument. So go ahead and create a new function with one parameters and these labels.



With the rule of three we can calculate the number of rotations for any distance. The number of rotations is **distance/94**.

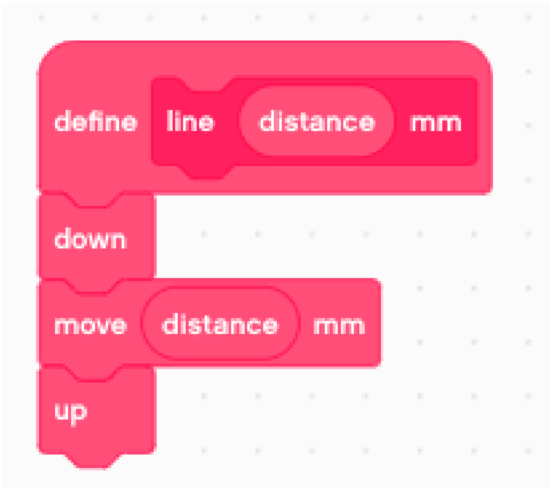


Now we can call this function with a specfic argument. For example 50 mm. Try it and measure the length of the line.

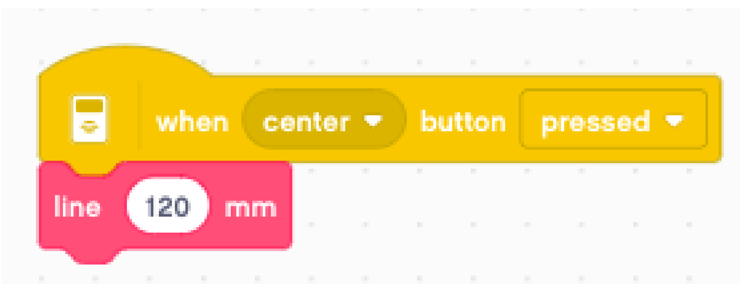


11.5 Create a line function

We can go one step further, and directly create a **line** function.

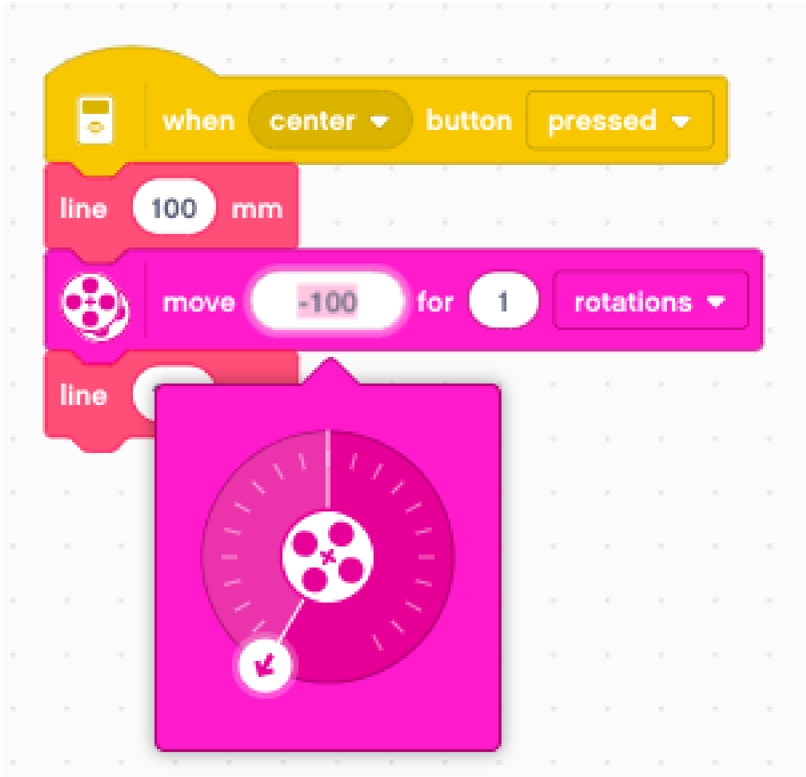


We can now call the **line** function to draw a line of for example 120 mm.



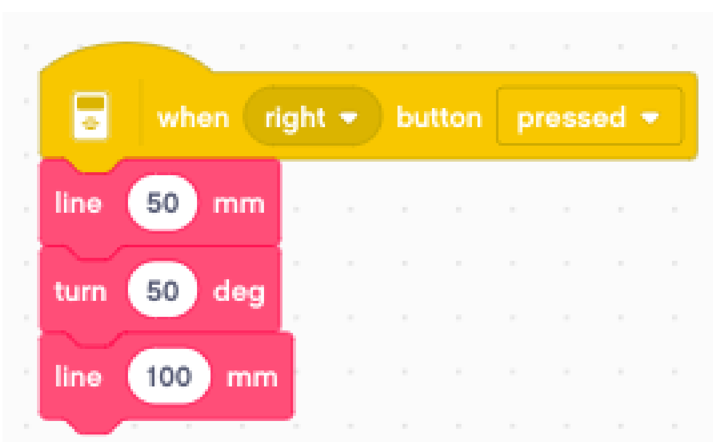
11.6 Turn the robot

Now let's turn the robot on place. First we draw a line 100 mm. Then we pivot by 1 wheel rotation to the left. And finally we draw a second line of 100 mm.



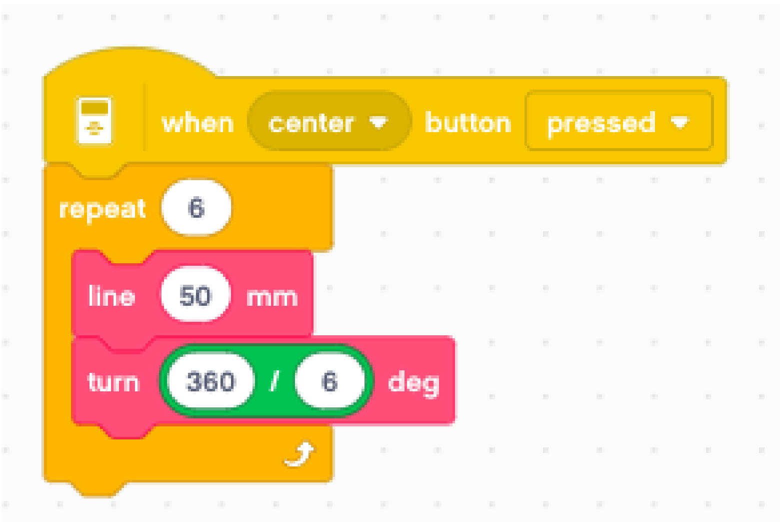
We find that the robot turns by 82 degrees. This allows us to create a **turn** function.

Now we can call this function with a 90° angle.

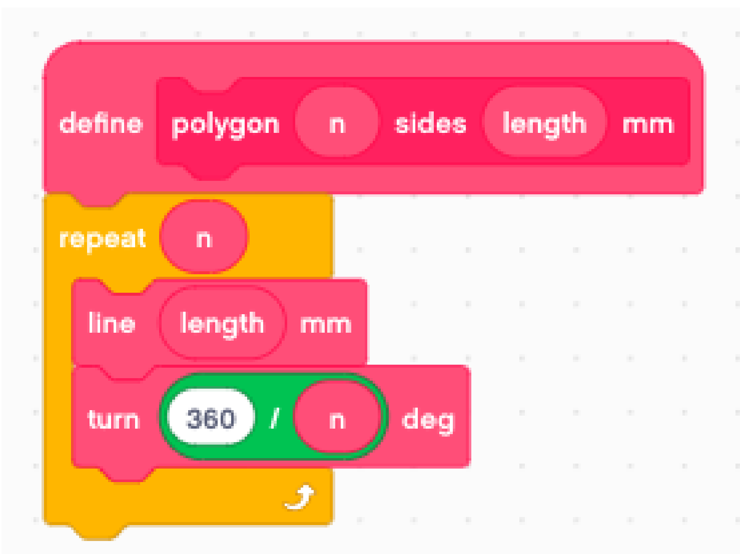


11.7 Draw a polygon

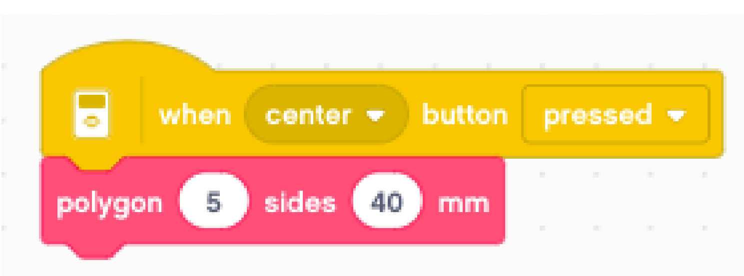
We have now everything needed to draw a regular polygon. We just use a loop to repeat **n** sides of a regular polygon. Then we turn an angle of **360/n** degrees. For example we can draw a hexagon with a side length of 50 mm.



Now we can turn this into a function.



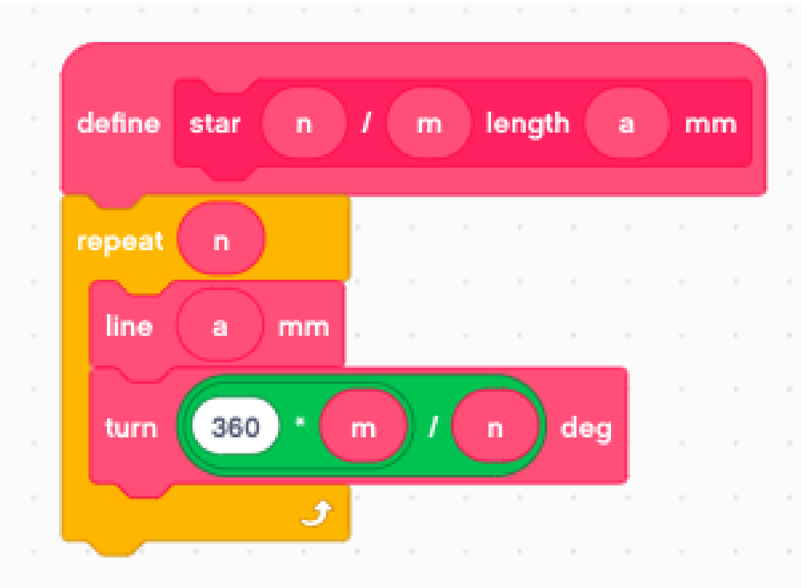
Now we can use the **polygon** function to draw a pentagon with a side length of 40 mm.



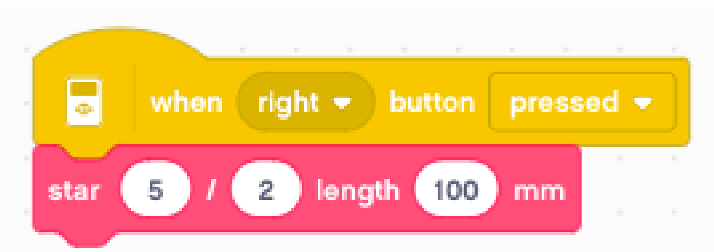
11.8 Draw a star

The the star polygon is drawn exactly as the polygon, but the turning angles are multiples of the normal polygon angle. For example, turning **360/5** results in a pentagon. However turning twice that angle, **2x360/5**, creates a 5-pointed start.

We create a **star** function which allows as to draw **n/m** star polygons.



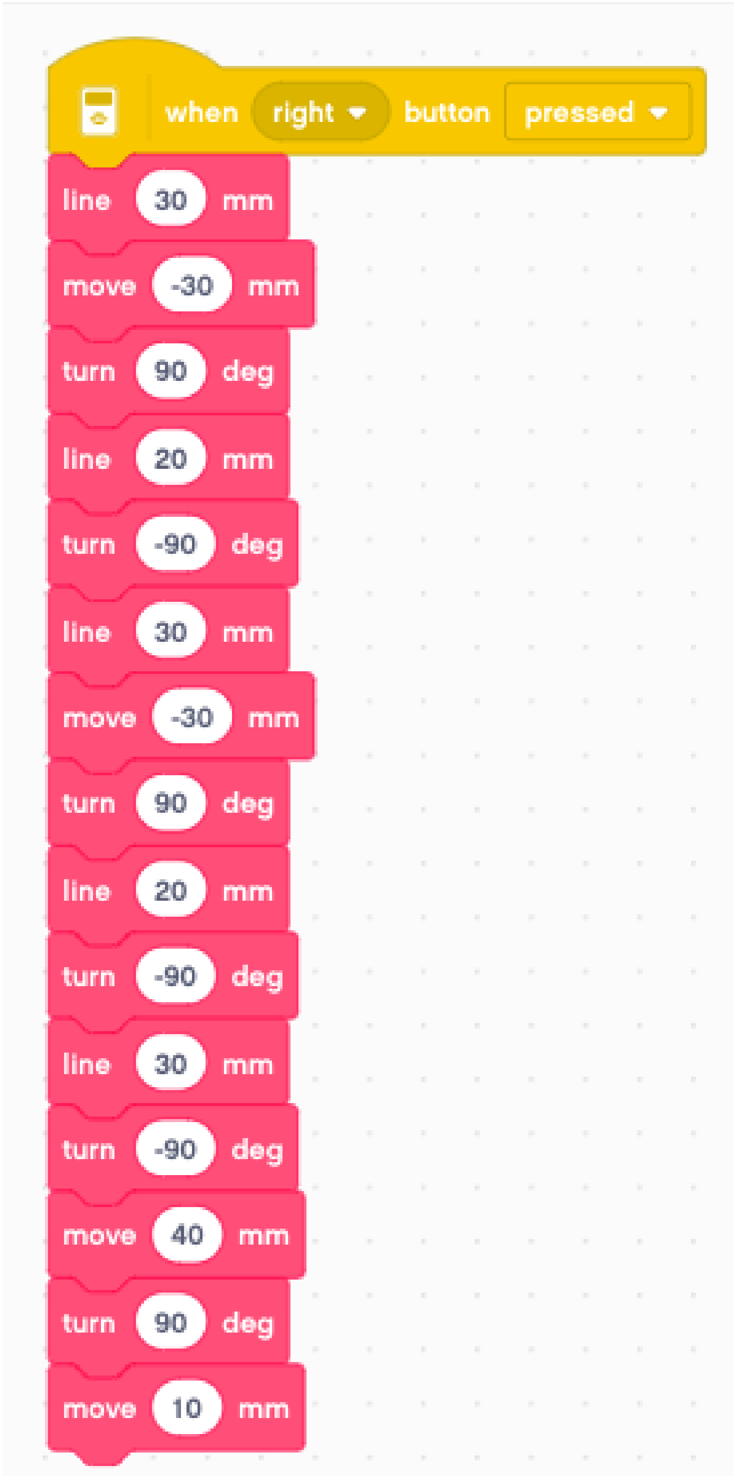
Now lets draw such a 5-pointed star



You can download the programs so far: `draw1.lmsp`

11.9 Draw a letter

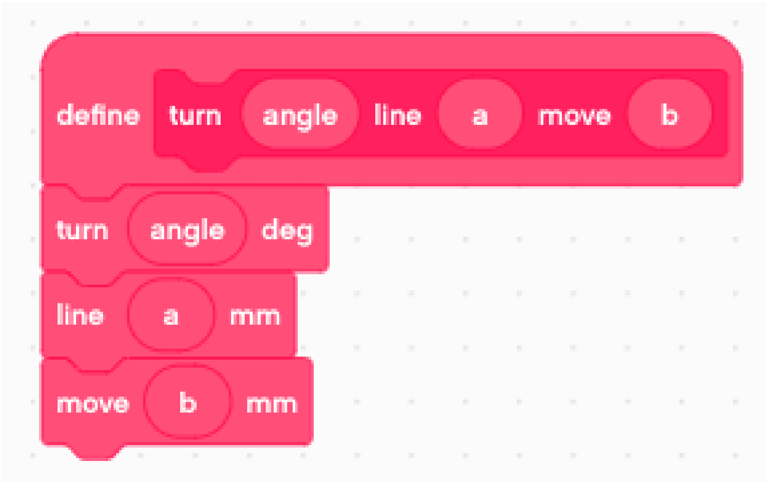
We have everything in place to draw a letter. For exemple to draw the letter **E** inside a rectangle of 30 x 40 mm we do this:



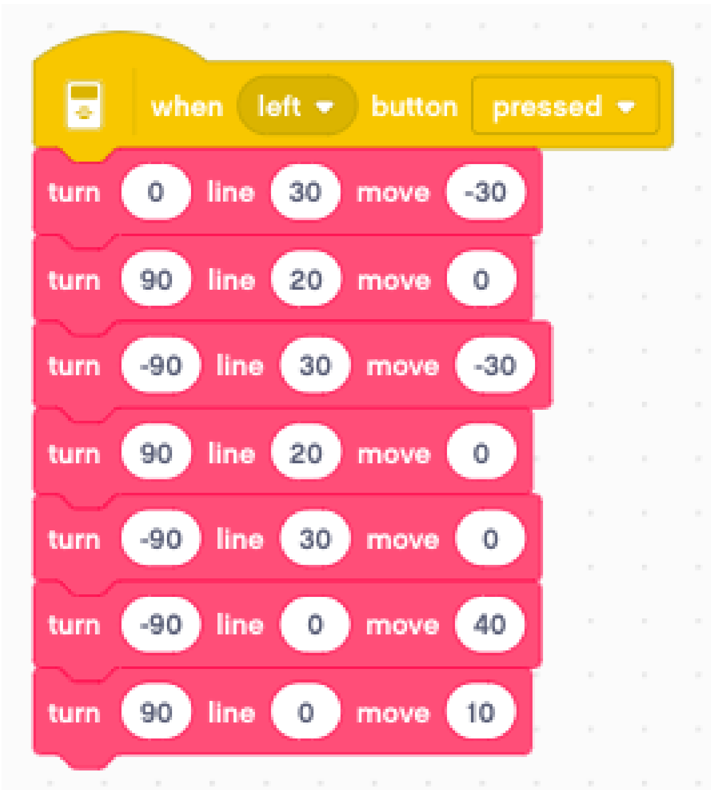
At the end we place the robot to the beginng of the next letter.

11.10 A function with 3 arguments

If you look at the previous program, you notice it's pretty long. But it consists of a sequence of *line, move* and *turn* functions. We could combine these three functions in one. Let's make this function with 3 arguments:

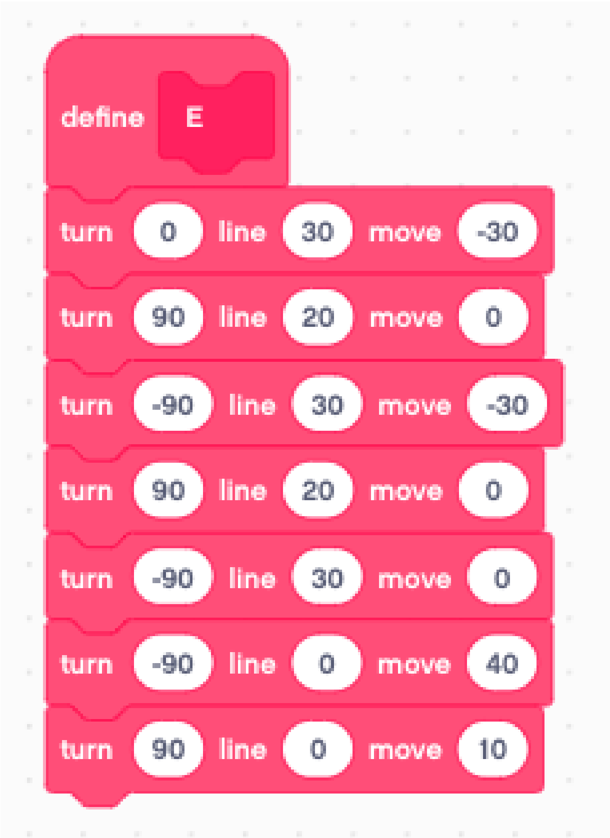


Using this new function, we can reduce the number of function calls from 15 to 7. It is easier to understand, as each line corresponds now to a segment of the letter.

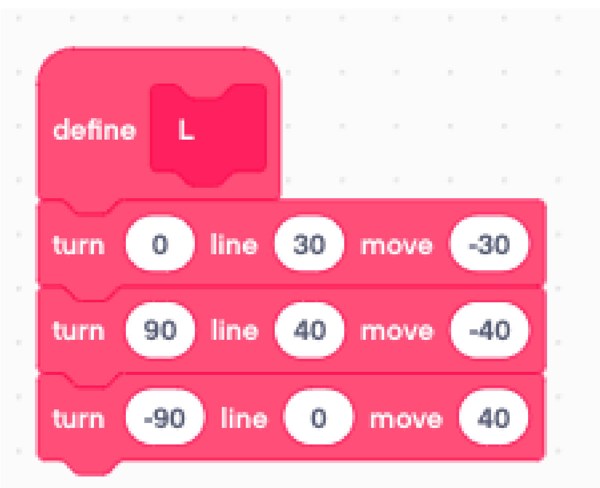


11.11 Define letters as functions

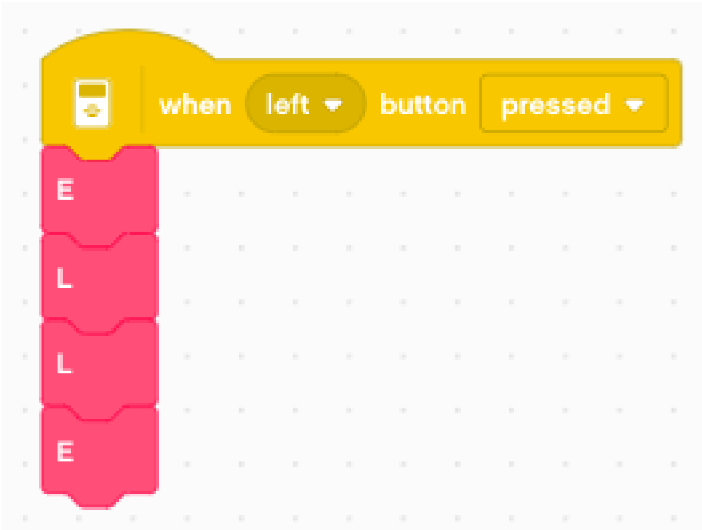
The next step is to define a function for each letter. We define the letter **E**



We define the letter **L**



And now we can write the word **ELLE**



11.12 Draw numbers in 7-segment style

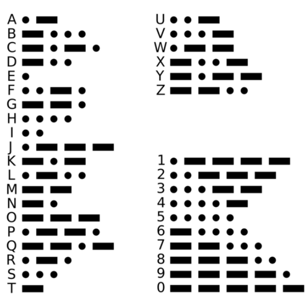
CHAPTER 12

Morse code

In this section we program the robot to create Morse code.

Samuel Morse invented this code which uses dots and dashes to encode letters. He looked at the frequency of letters and assigned the shortes codes to the most frequent letters.

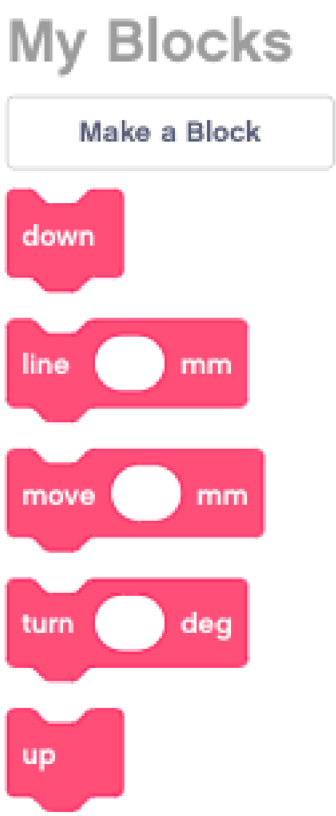
Here is the code



Create a dot

12.1 Drawbot

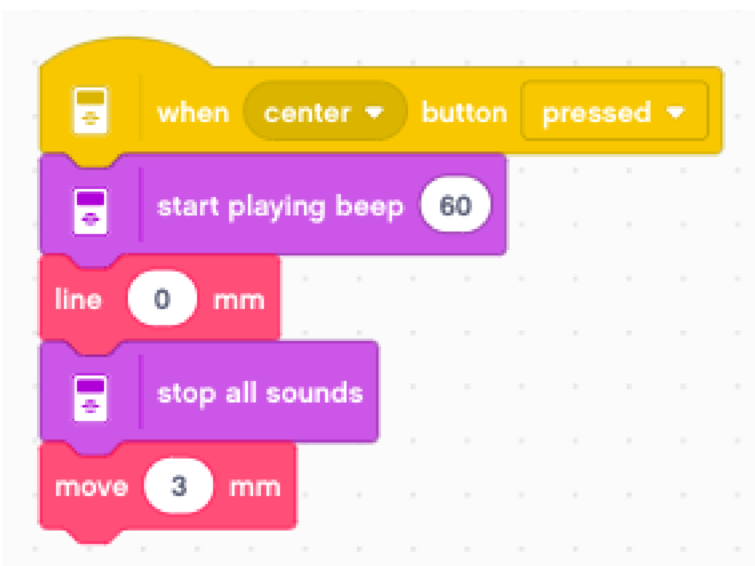
This section uses the Drawbot from the previous section. You will use these functios



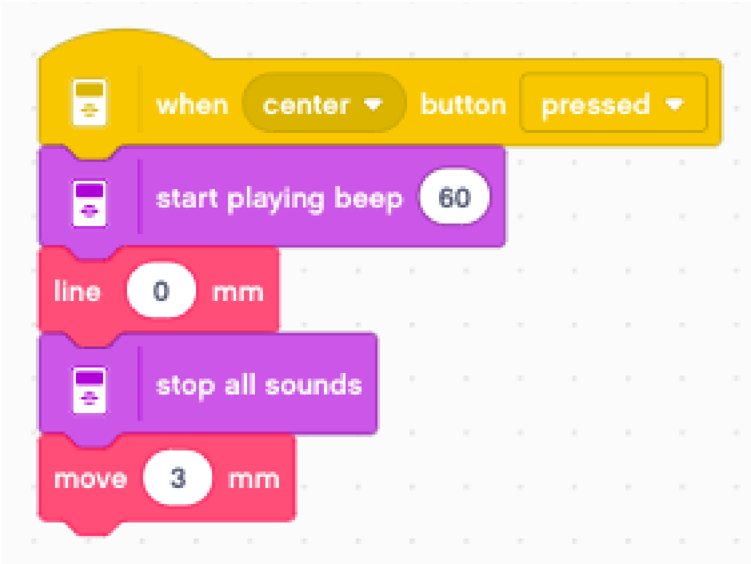
You can download this program which contains already this functions as your starting point: `draw.lmsp`

12.2 Play a dot or dash

Let's start with drawing a dot. That's just a line of lenght 0 mm. We play a sound which is defined by the duration it takes to draw that dot.



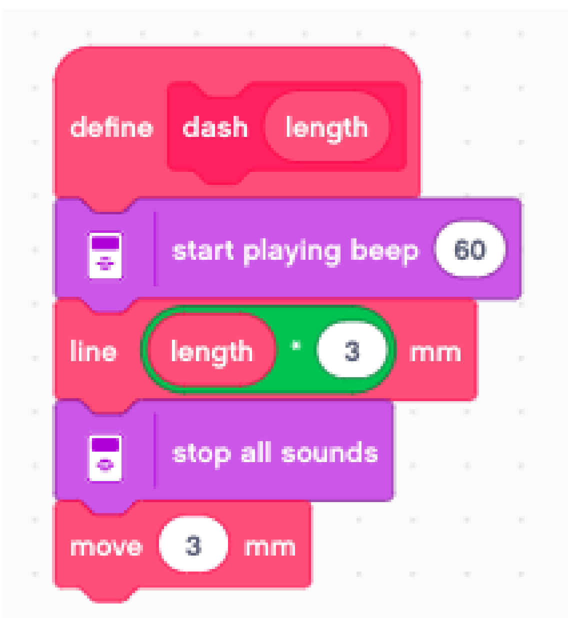
The dash is the same, just 3 mm long.



The basic symbols are separated by 3 mm of distance.

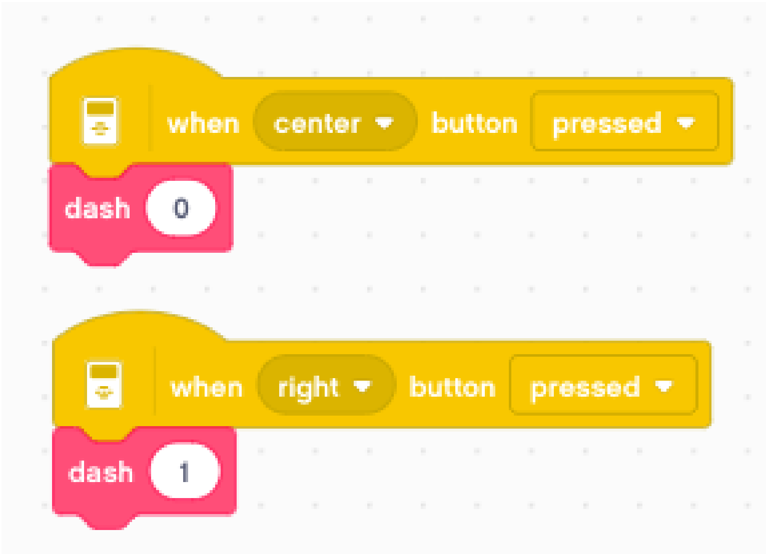
12.3 Make this a function

We could now make two functions **dash** and **dot** and place the 4 lines of code inside. However since they are very similar, we will make just **dash** and give it a parameter.



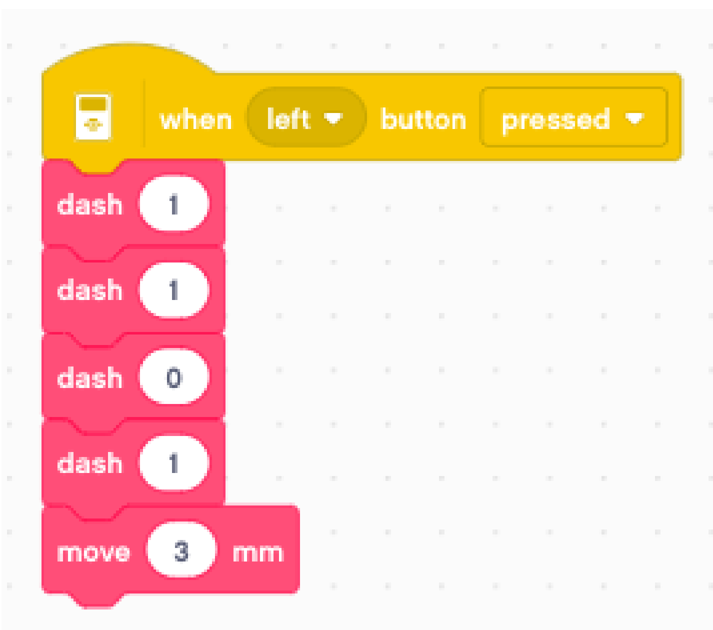
Now we can call it with these two parameters

- 0 to draw a dot
- 1 to draw a dash (3 mm long)



12.4 Draw the Morse code for Q

Now we can program for example the code for the letter Q. It's dash-dashes-dot-dash. We have to add an extra 3 mm space to separate it from the next letter.



We could define a fonction for all 26 letters. But there is a better way

12.5 Decompose a sequence with modulo

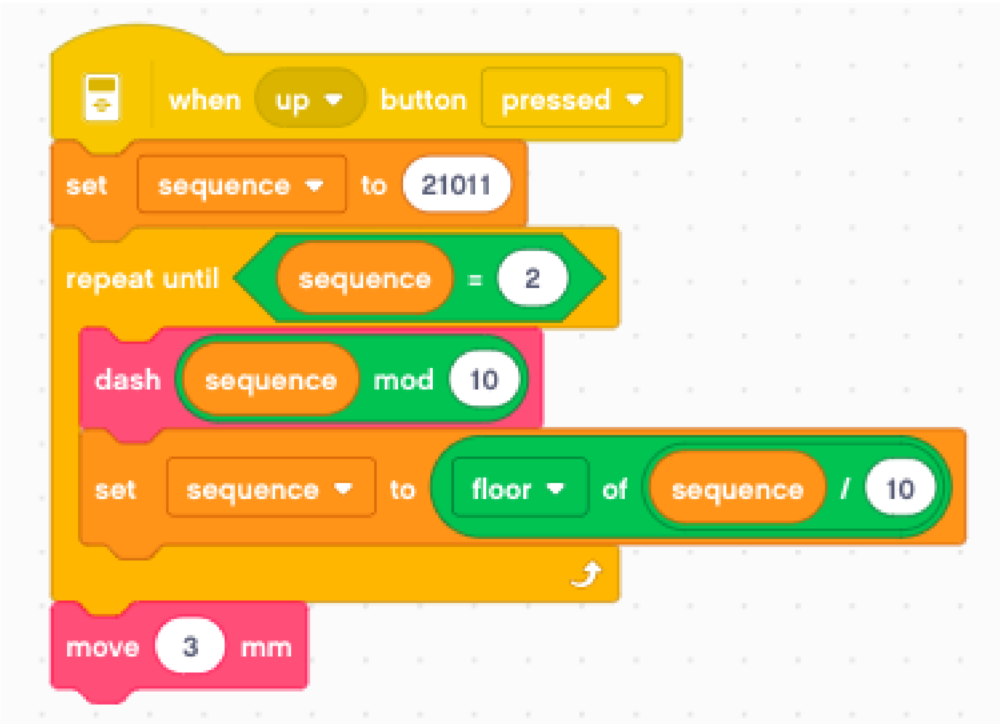
The **modulo** function or **mod** function returns the result of a divison. In our case we will use **mod 10** which will give us the reminder of a divison by 10, or in other words it gives us the last digit of the number:

```
21011 mod 10 = 1
2101 mod 10 = 1
210 mod 10 = 0
21 mod 10 = 1
2 mod 10 = 2
```

The function **floor** returns the integer part of a number we get from the division by 10:

```
floor 2101.1 = 2101
floor 210.1 = 210
floor 2.1 = 2
```

It is still the letter Q, but the sequence is read from the back.

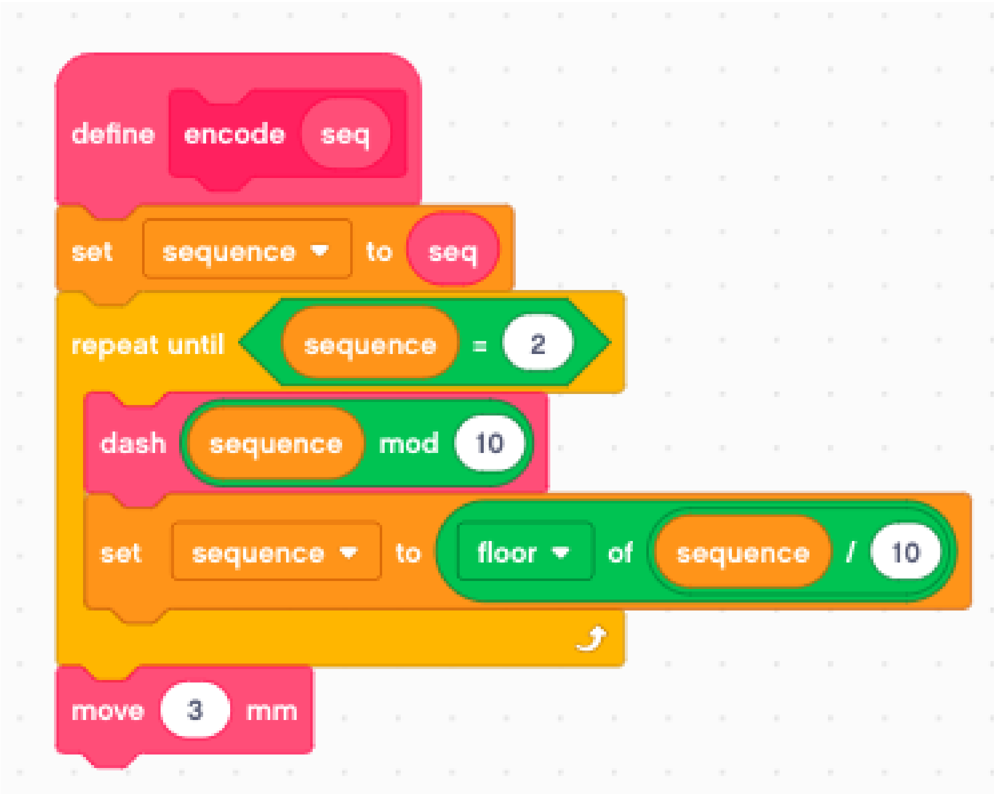


12.6 Create a function

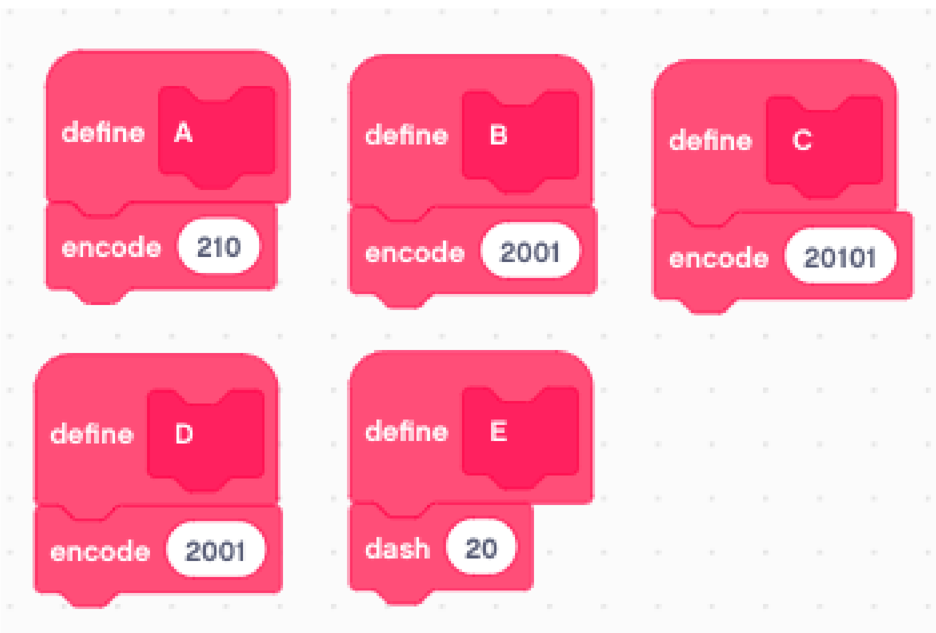
We can now create a function which encodes a number sequence into Morse code. The number sequence has to be defined in reverse order. The code is composed of 3 digits:

- 0 to indicate a dot
- 1 to indicate a dash
- 2 to indicate the end of the sequence

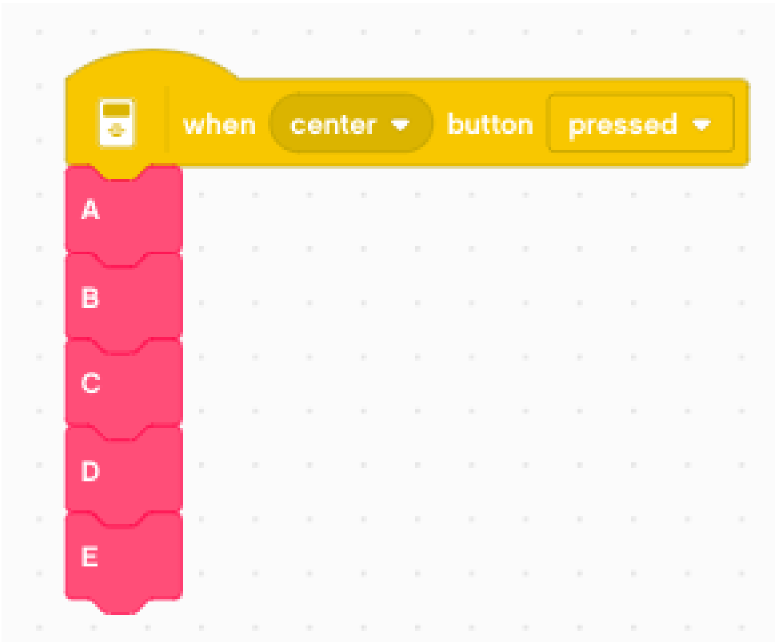
The end marker is necessary because in numbers preceding zeros are ignored.



The **encode** function is used to define a function for each letter (A, B, C, etc.)



We can compose words by using these functions. As en example we print the letters ABCDE.



You can download the programs so far: `morse.lmsp`

CHAPTER 13

Robot Arm

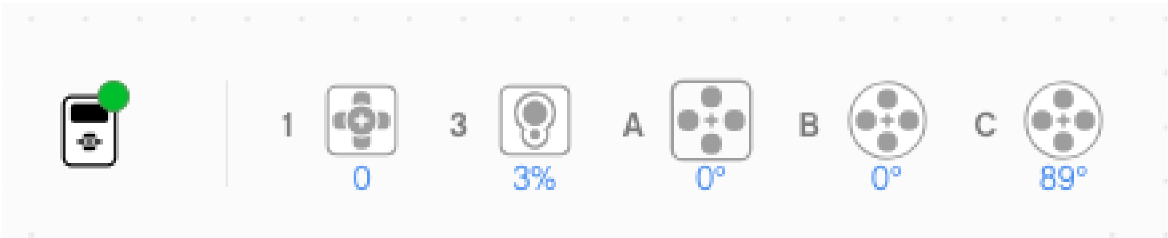
In this chapter we control a robot arm. You can:

- lift the arm
- rotate the arm
- open and close the hand

You can find the construction guide to build the robot arm here.

13.1 Motors and sensors

When you connect the robot you will see these sensors and motors:

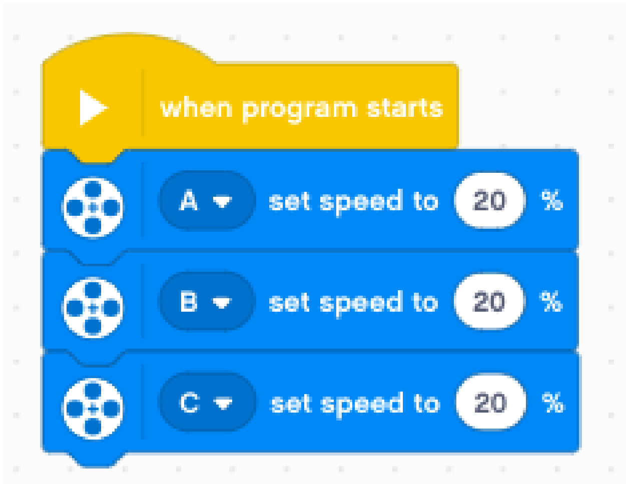


A rotation in the clockwise direction has this effect:

- motor A : opens the hand
- motor B : lowers the arm
- motor C : turns the arm clock-wise

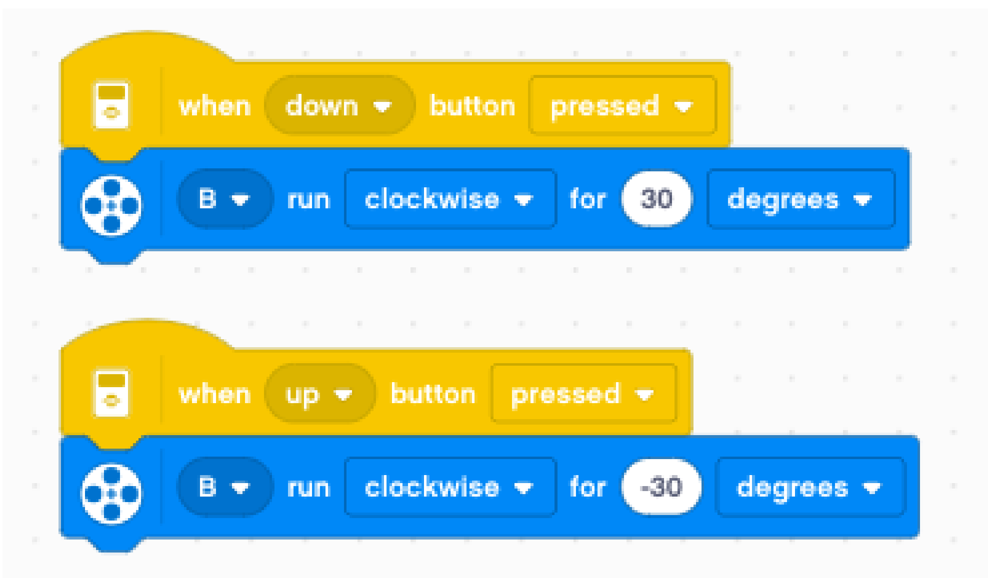
The two sensors serve to detect the range limit of the arm.

The first thing we will do is to set the speed of the 3 motors to 20%.



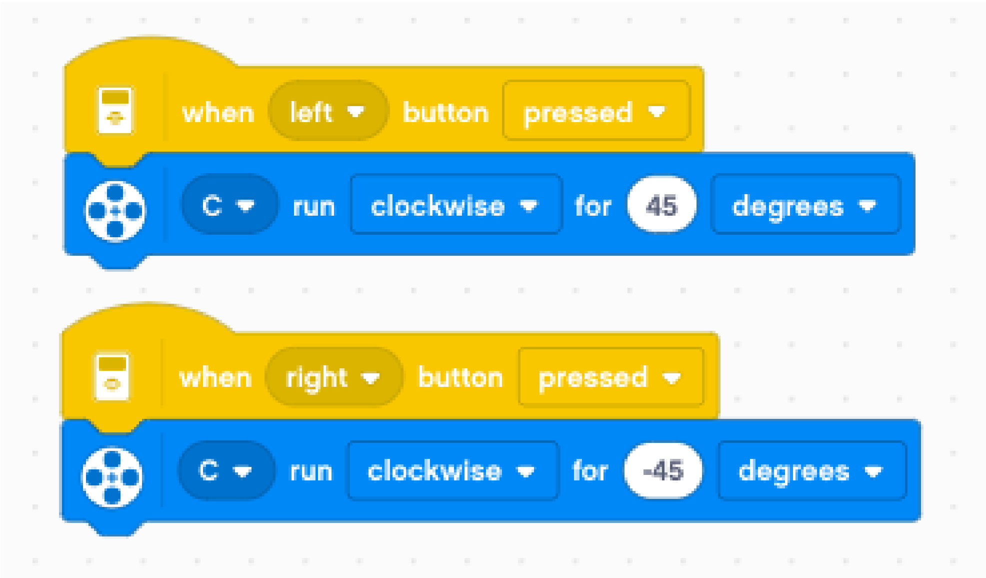
13.2 Lift the arm

We program the **up/down** buttons to move the motor B by 30° increments.



13.3 Rotate the arm

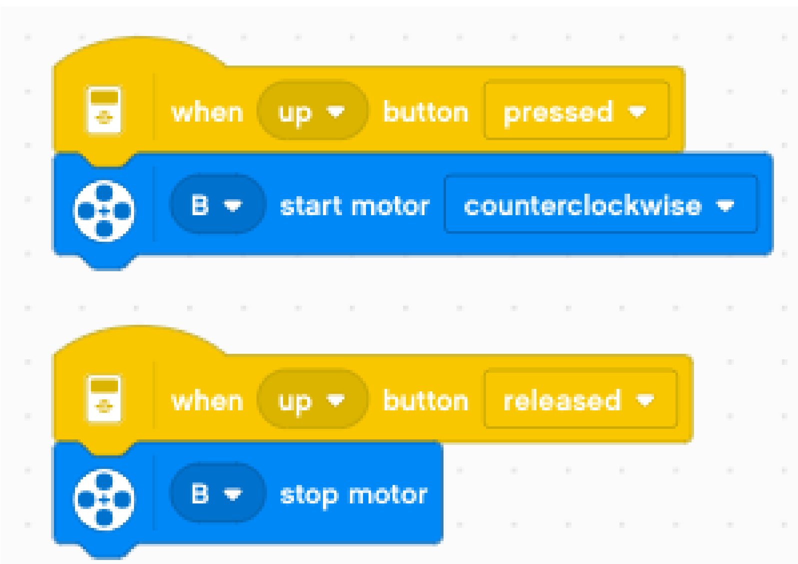
We program the **left/right** buttons to move the motor C by 45° increments.



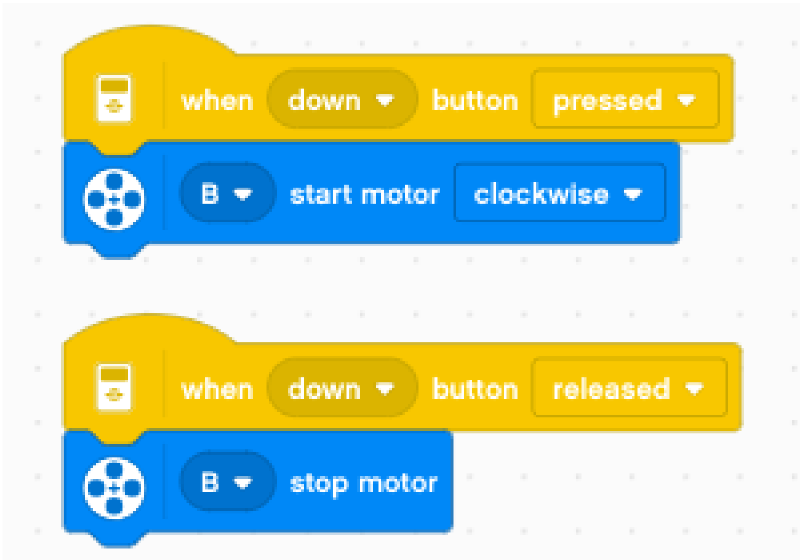
13.4 Move continously

Now we change the program. We move the motor as long as we press the button. For this we use two events:

- the **pressed** event
- the **released** event



And for the other direction



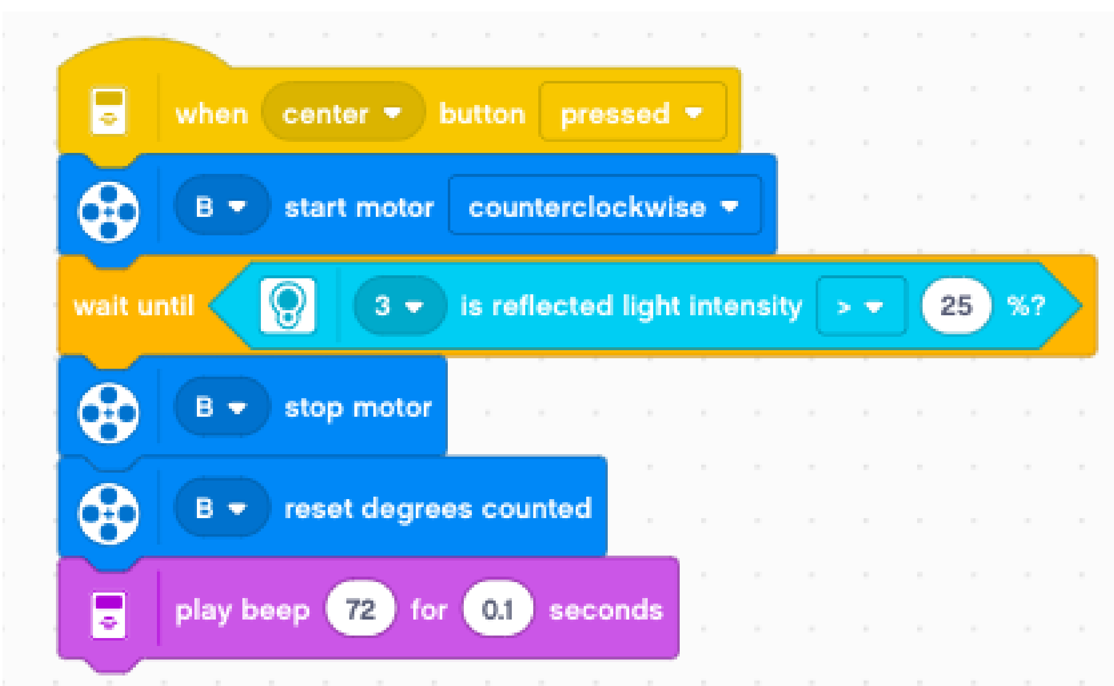
13.5 Limit the lift

In order to calibrate the robot, two sensors are attached to the robot:

- the light sensor detects the upper limit of the arm,
- the touch sensor detects the rotational limit of the arm.

Move the motor upwards and look at the light value. It will go from 5% up to 30%.

Let's program the **center** button to start the calibaration movement. We move the arm up until the reflected light intensitiy is larger then 25%. Then we stop the motor and play a short beep.



We also reset the rotation sensor, so that this upper value becomes 0°. So when the robot is calibrated, we always can know its absolute angles.

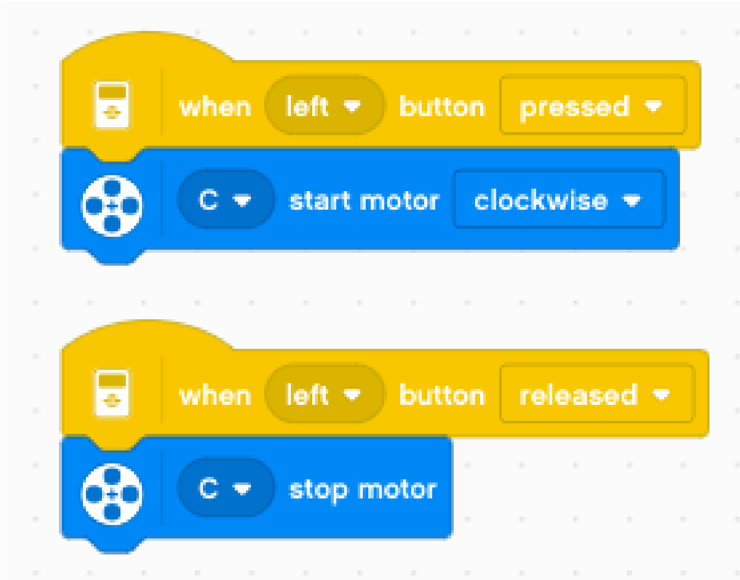
You can lower the arm now and find the angular postion when the arm touches the table. It should be close to 280°.

You can download this program: `arm2.lmsp`

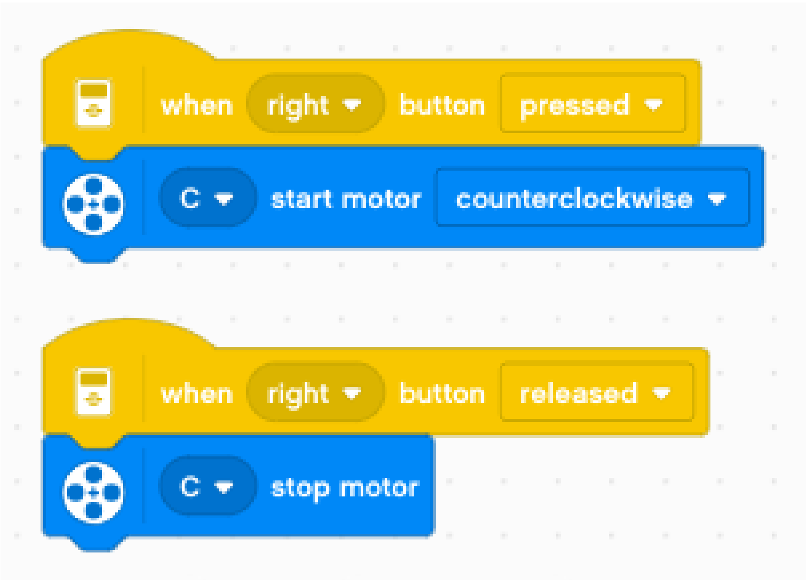
13.6 Limit the rotation

Again, let's program the arm that it moves as long a button is pressed. For this we use these two events:

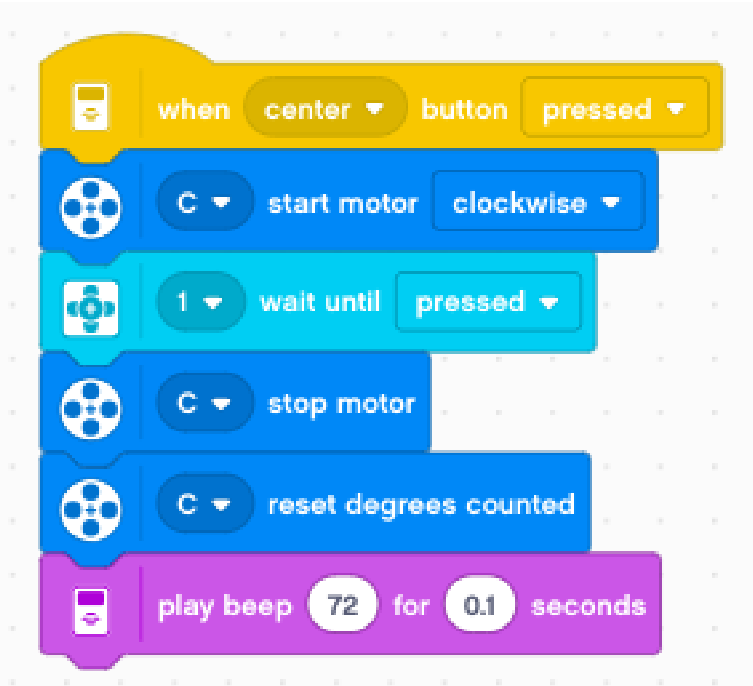
- the **button pressed** event
- the **button released** event



And do it also the other way.



Move the arm slowly to the left and try to find the position where the touch sensor detects the limit position. It's when the arm faces completely backwards. Like before, we program the **center** to go automatically to that position.



When the limit position is detected, we sound a short beep and reset the rotation sensor. This position becomes the new zero.

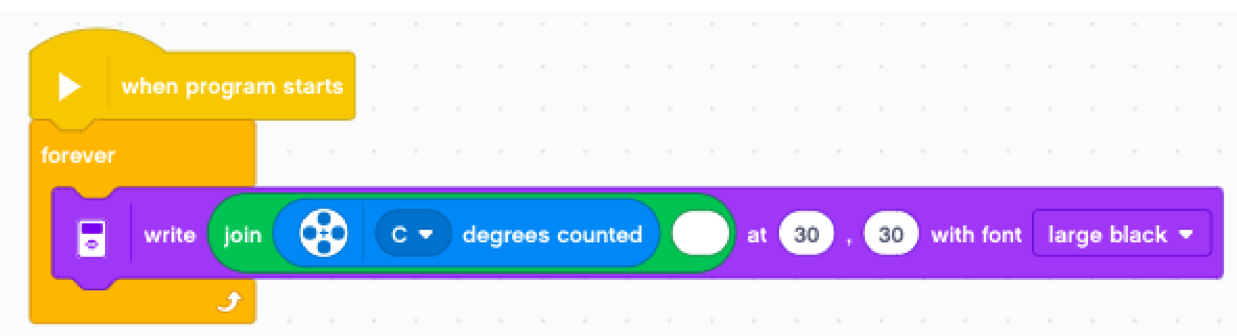
Now move the arm manually to the front position. The sensor position should be about -600°.

13.7 Display current position

It's quite useful to display sensor information directly on the brick. Of course now we can see the sensor values on the computer, but the brick can also run without a computer being attached.

In order to display the current sensor values, we use the **when program starts** event to start a **forever** loop. Inside this loop we write the sensor value to the screen.

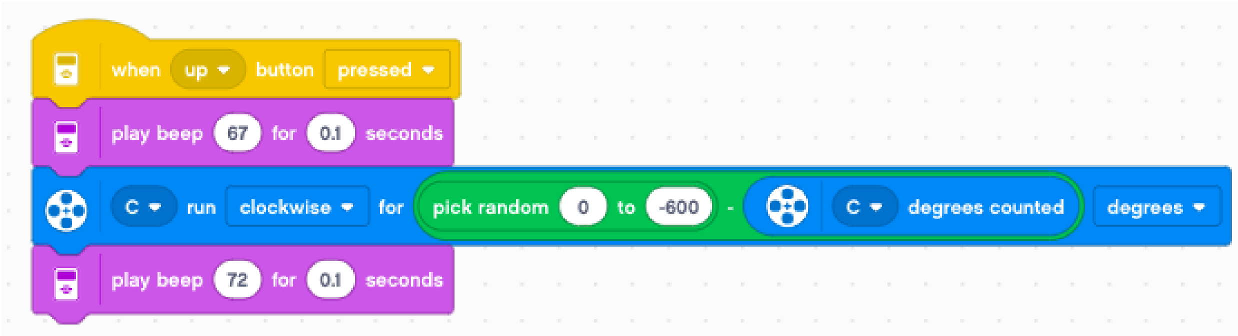
In order to see it well, we place it at position (30, 30) and we select a large black font.



13.8 Go to a random position

Now that we have calibrated the rotation and established it's allowd range, we can control the arm. Any value in the range [0 .. -600] is allowd.

We can use the **pick random** function to get such a value. In order to know the distance the motor has to move we calculate the difference **target - current**.

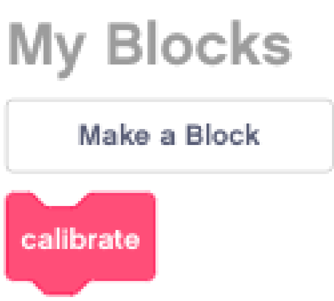


We mark the beginning and end of this random move with two different sounds.

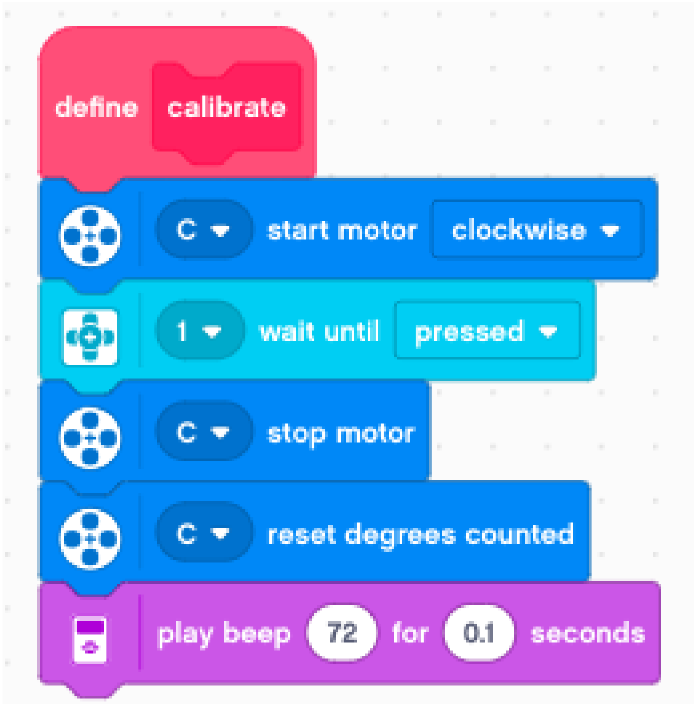
You can download this program: `arm3.lmsp`

13.9 Create a calibrate function

Now it's time to define our first function. The calibration needs to be done each time at the beginning of the progam. Let's define a function and execute it automatically at start.



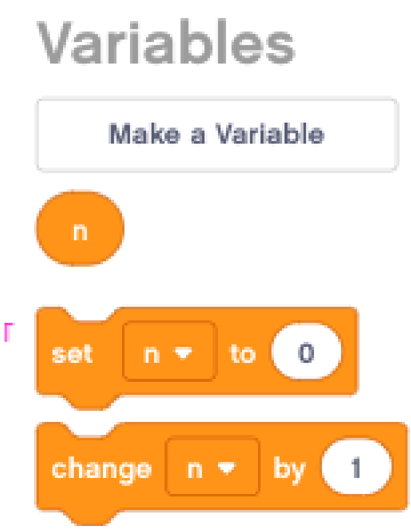
Create a new **My Block** and define it like this:



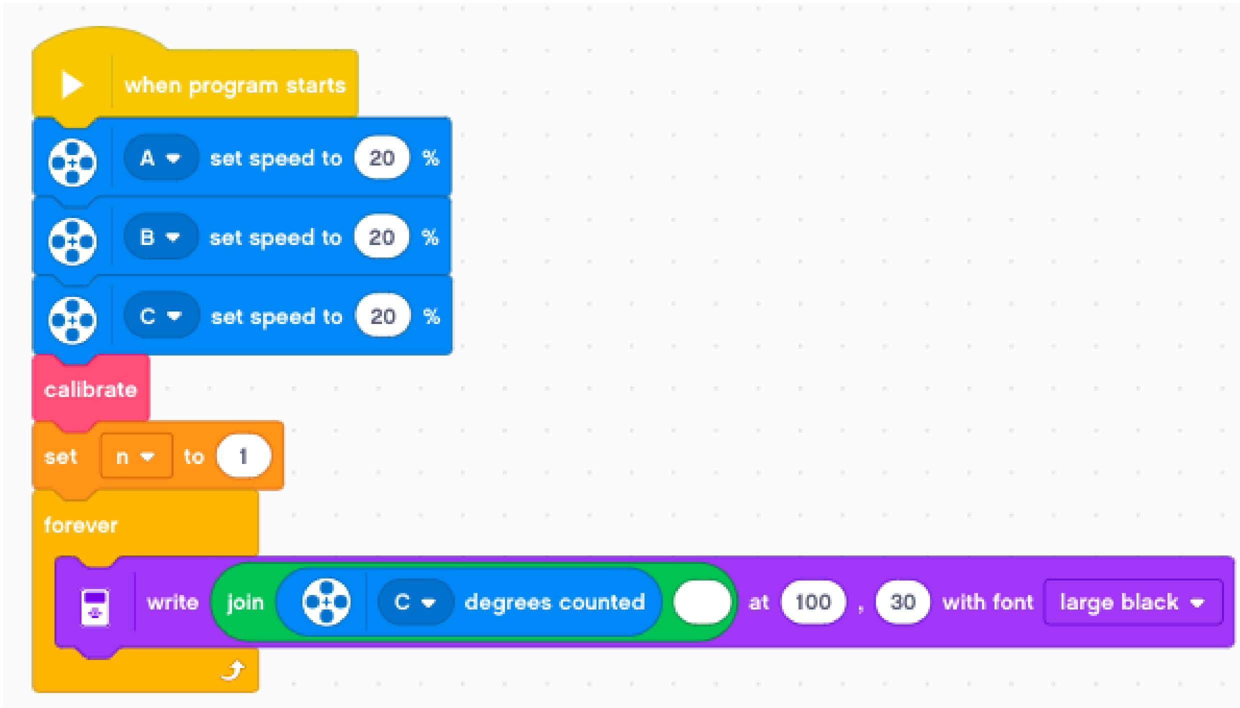
13.10 Record arm positions

An industrial robot needs to go to specific positions. It must remember these positions. We are going to program the arm so that it can memorize positions.

We make a new variable **n**



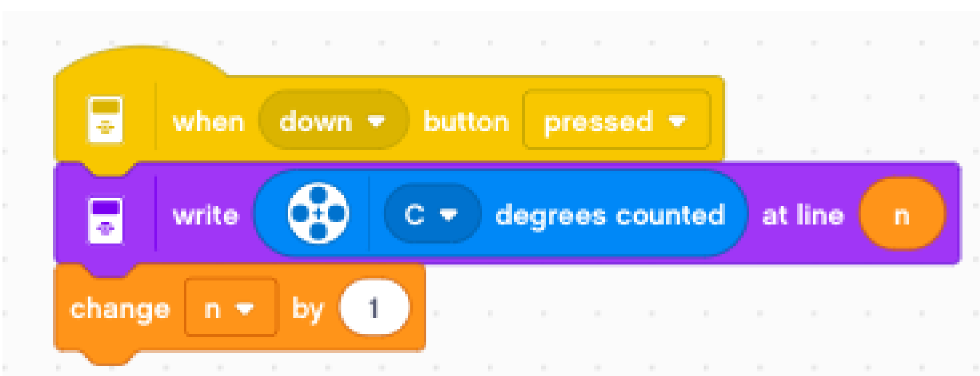
This variable will be used to count each memorized position. In the **start** event we set the variable n to 0.



We place all the commands which need to be done once at the beginning. Then we enter a **forever** loop to repeatedly display the current sensor position.

Then we create a **button down** event which does:

- write the current sensor value to line **n** on the screen
- increment the variable **n** by 1



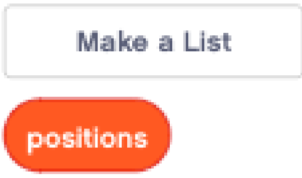
After calibration has finished, move the arm with the **left/right** button. Then press the **down** button to write the current position to the screen. You will get something like this:

```
-234
-345
-435
-534
```

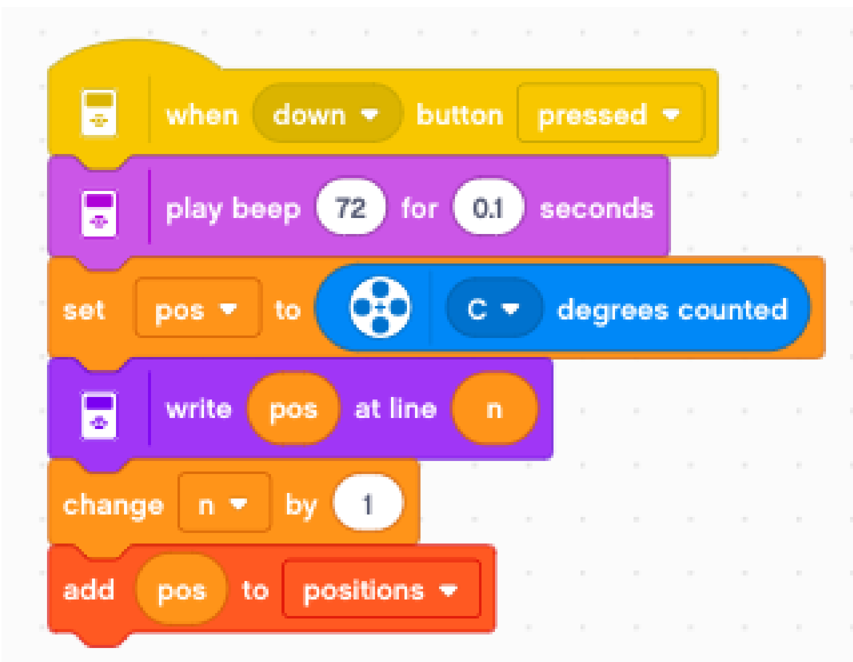
13.11 Saving values in a list

So far these values have just been written to the screen. They are not registered in any list.

Create a new list called **positions**.

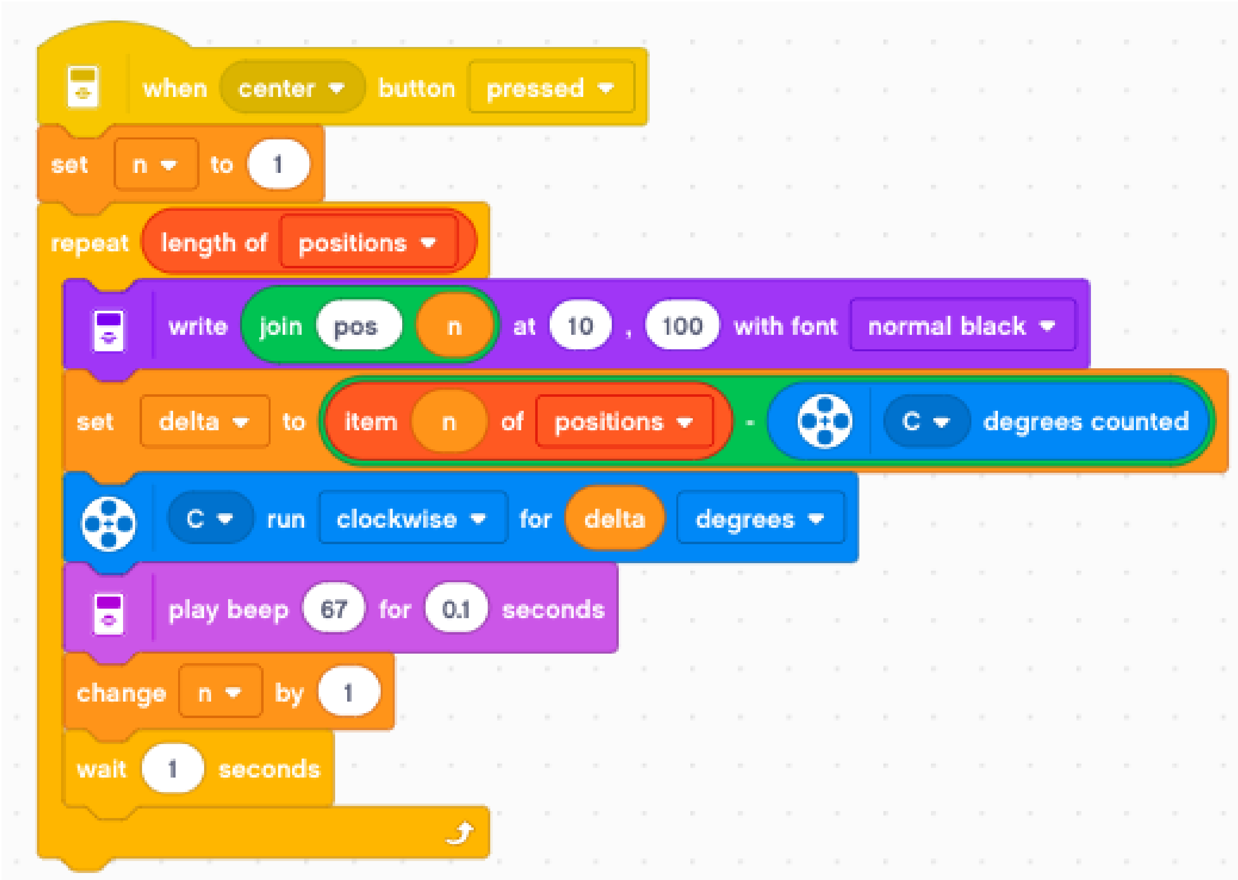


Now we have to change the **button down** event to save the current position in the list.



13.12 Replaying the list

Now we are ready to program the **replay** function. We use the **center** button event.



We reset the variable n to 1, to point to the first element in the list. Then we enter a loop which will repeat the number of times there are elements in the list.

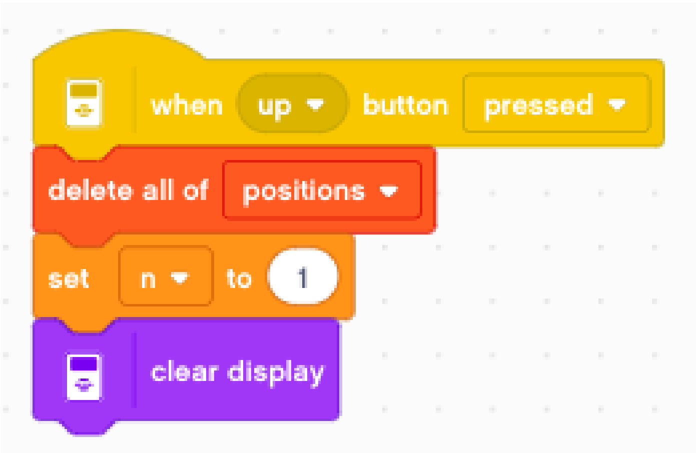
Inside the loop we:

- display the current item number (pos 1, 2, 3, . . .)
- calculate the **delta** value the arm has to move
- move the arm to the new position
- play a beep
- increment the variable **n** by 1
- wait for 1 second

13.13 Reset the list

At the end of the replay the variable **n** will be pointing at the next possible position. It is possible to add more values to the list. At any time you can replay the list.

In order to reset the list we use the **up** button.



It does:

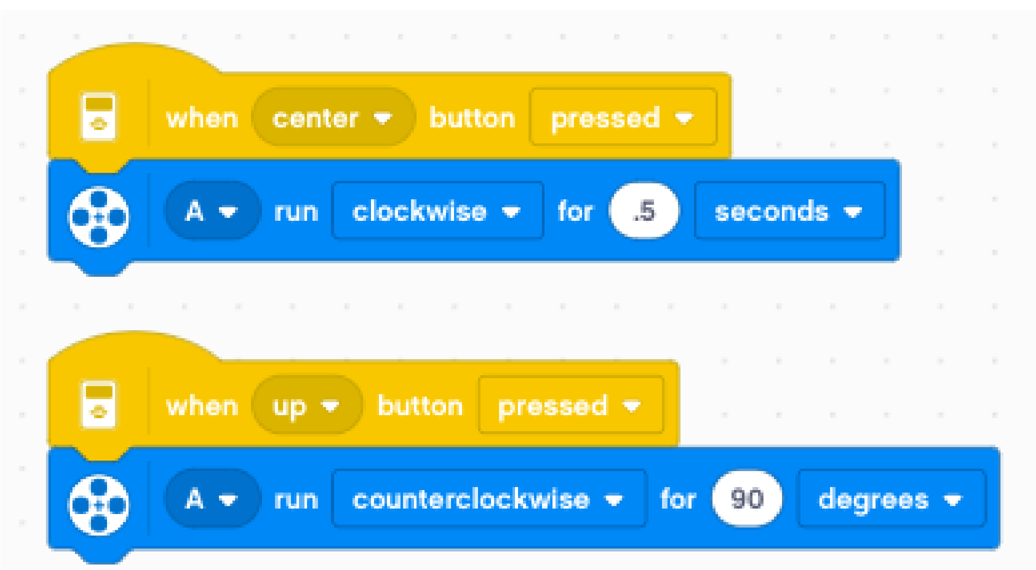
- delete the list
- reset the variable n to position 1
- clear the display

You can download this program: `arm4.lmsp`

13.14 Open and close the hand

To operate the hand we control motor A. This motor does not have a limit sensor. We use a little trick to find the end position. We close the hand for about half a second. Once the hand is closed, the motor cannot move any further and it will stop shortly after. This gives a defined state. From there we can go 90° the other way, to open the hand.

- closing for 0.5 secondes (time mode)
- opening 90° (rotation mode)



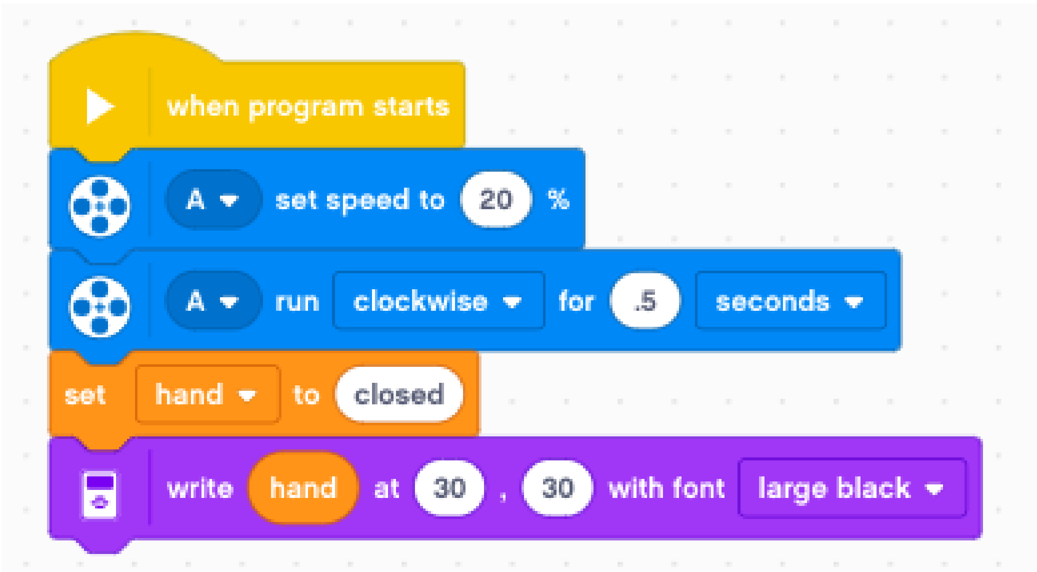
You can try to put an object between the claws and close the hand. The object will be firmly held.

13.15 Remember the state

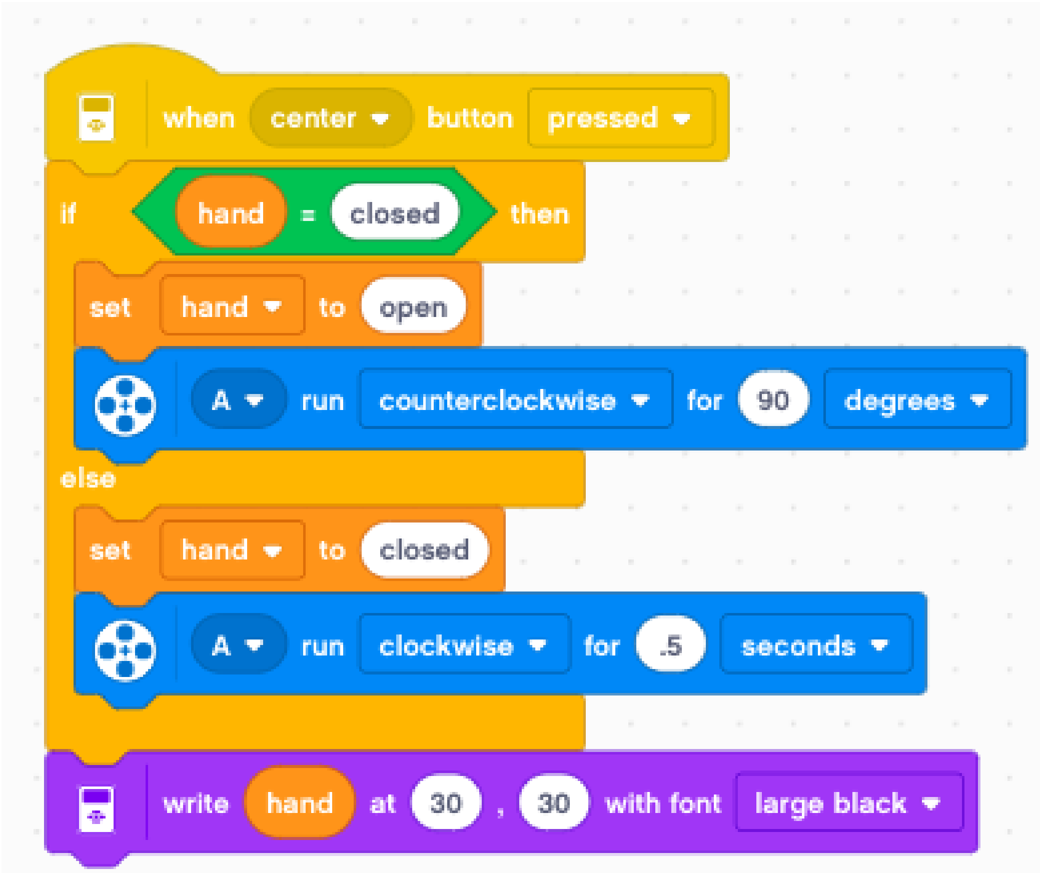
Since we only have 5 buttons, it would be convenient to use just one button to operate the hand. This button could be used to toggle between the two states:

- open
- closed

We use a variable **hand** to keep this two states as a string. We define the hand to be closed at the start of the program.



Based on the state of the variable **hand** we open the hand if it's closed and close it if its open.



Inside the **if-else** block the state of the variable is inverted. At the end, the current state of the hand is printed to the screen in large letters.

You can download this program: `arm5.lmsp`

CHAPTER 14

Gyro Boy

Gyro Boy is a self-balancing robot.

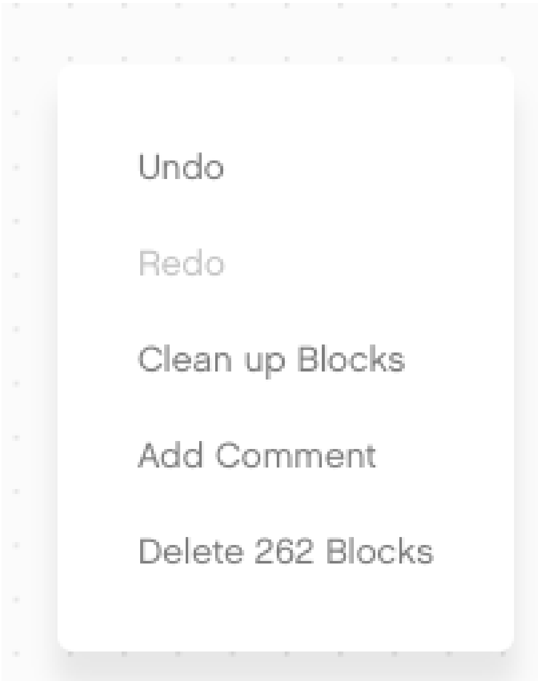
14.1 Reverse engineering

In this section we will do a bit of reverse engineering. The code for a balancing robot is quite complex.

You can find the official project in the EV3 Classroom under **Home > Core Set Models > Gyro Boy**.

Or you can download the file here: `Gyro Boy 1.lmsp`

The initial file has a total of 262 blocks. You can see this in the contextual menu.



Let’s try to simplify this program and keep only the essential blocks.

- remove the user interface start loop (down to 202)
- remove calibrate gyro offset (169)
- simplify timing dt = t-t0 (163)
- simplfiy angular speed and angle (152)
- remove averaging of motor position (137)
- delete clear screens, integrate control loop timing (132)

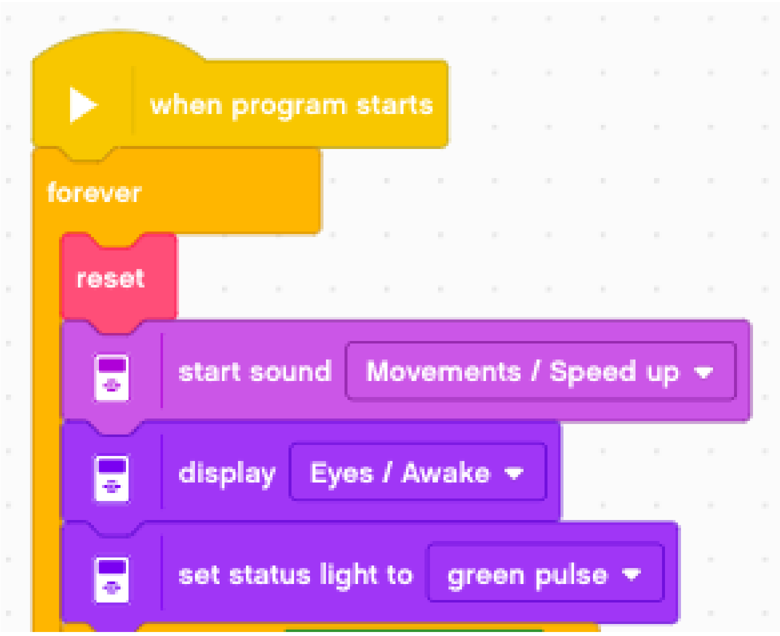
Finally we manage to go down to 98 blocks.

You can download the file here: `Gyro Boy 2.lmsp`

14.2 Startup

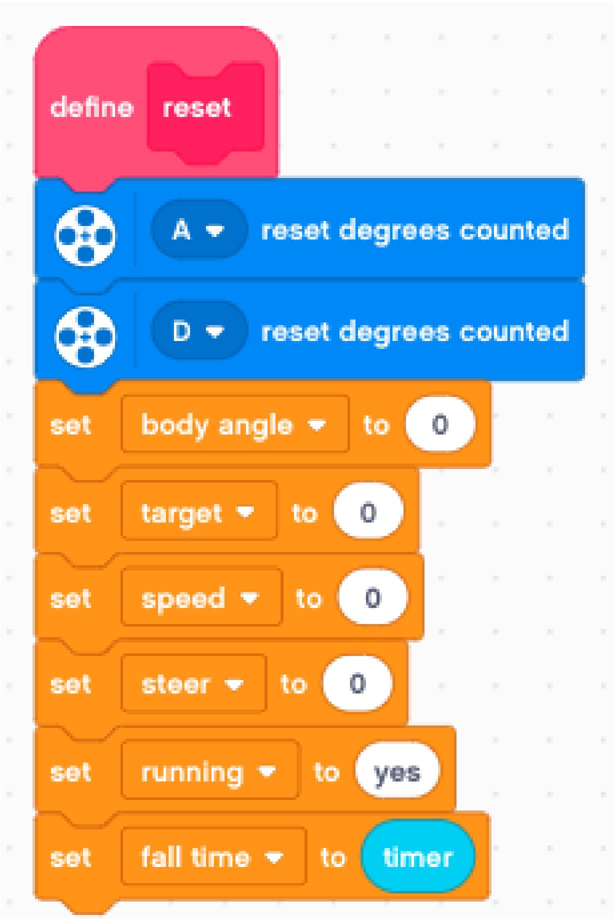
At startup we:

- call the **reset** function
- do a startup sound
- display awake eyes
- set the lights to green pulsing



This is the reset function which:

- resets both motor encoders to 0
- sets the body angle to 0
- sets speed and steer to 0 (remote control)



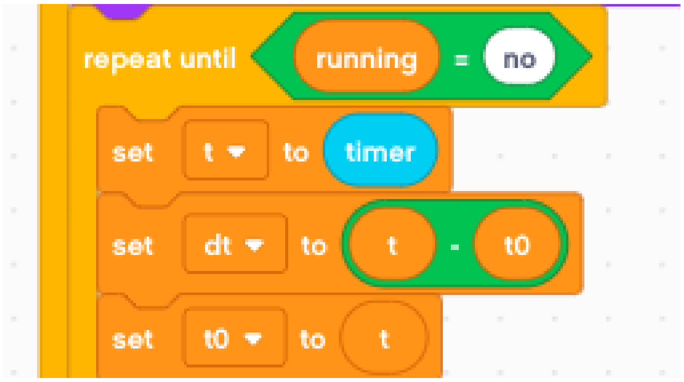
14.3 Get the loop time

The first part is about timing. We need to know the time **dt** it takes for the feedback loop. The loop time dt is used to calculate :

- angle from speed : a = (s1-s0) * dt
- speed from angle : s = (a1-a0) / dt

This is a classic 3-line algorithm.

- read the new timer value **t**
- calculate the increment **dt = t - t0**
- set the old timer value to the new one



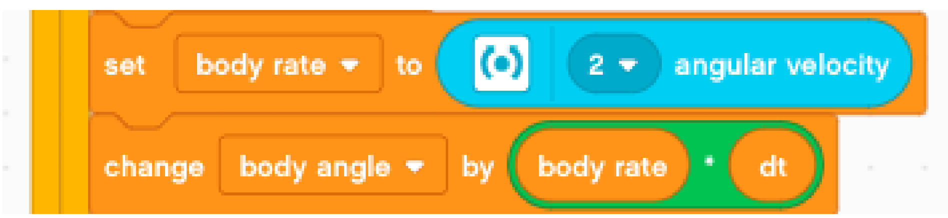
14.4 Angle from rate

Then we calculte the angle from the angular velocity. Why not reading the light-blue built-in **gyro angle** variable ?

Well, it's less precise and it does not seem to be possible to read both, angular velocity and angle at the same time.

Mathematically speaking we obtain the angle by integrating the angular speed. That's what we are doing here. The formula for discrete integration is:

```
angle += rate * dt
```



14.5 Rate from angle

For the wheel we are in the opposite position. We measure an angle, and calculate the rate (speed) eed with a derivative.

First we measure the wheel angle, which is the sum of both rotation sensors. If the robot moves in a straight line, they add up. If the robot pivots and stays in place, they cancel each other.

Again, why not use the dark blue **motor speed** variable ?

Well, it's less precise, because we don't know exactly what dt is used internally.

The formula for discrete derivation is:

```
rate = (angle - angle0) / dt
```

14.6 The PD controller

The robot tries to stay at the target position. The target position is the path lenght when speed is integrated over time:

```
target += speed * dt
```

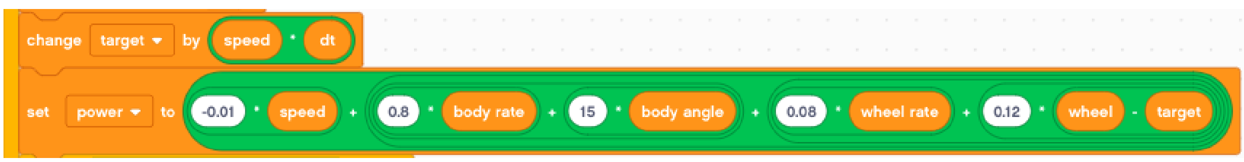
In technical terms we have a PD controller: P for proporional control and D for Derivative control.

The magic of the control system happens here. The power is a sum of 5 weighted quantities which describe the state.

Concerning the signs we have this situation:

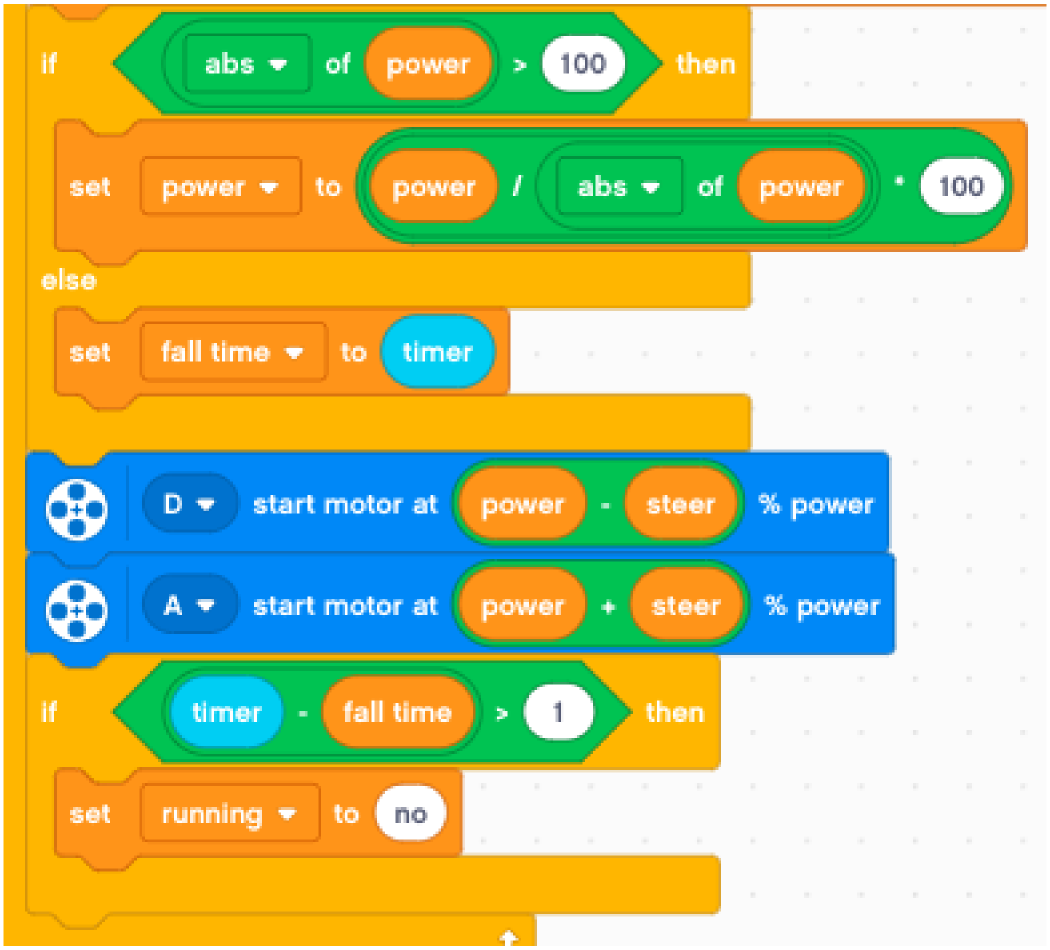
- positive body angle - robot leans forward
- positive body rate - robot falls forward
- positive wheel angle - wheels are ahead of target
- positive wheel rate - wheels move forward

In all this cases the power must be positive, to counter the tendency of the robot to fall.



14.7 Motor control

The last part controls the motors.



First we limit the power to the range [-100 .. 100]. We use a little trick to get the sign:

```
sign(power) = power/abs(power)
```

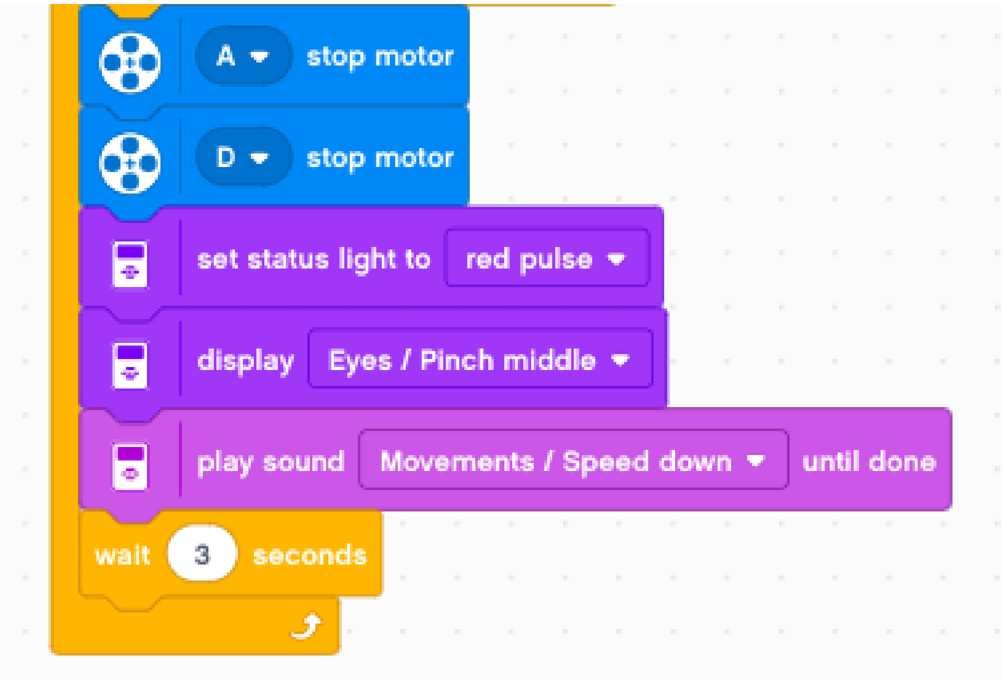
Each time we are in normal regime (not in saturation) we reset the **fall time**.

We use differential steering for the two motors. If the saturation regime lasts more then 1 second, we stop running.

14.8 Shutdown when falling

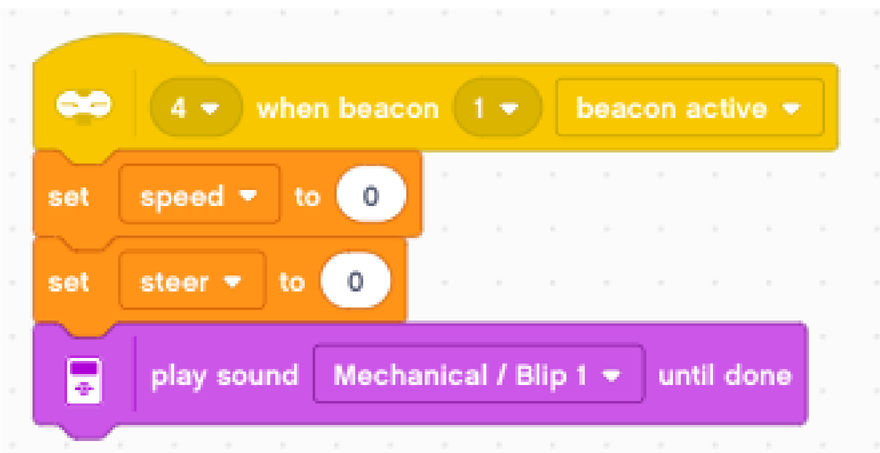
If the robot falls and runs in saturation for more than 1 second, we:

- stop both motors
- set the status light to red pulses
- show pinched eyes
- play the spead down sound
- wait for 3 seconds

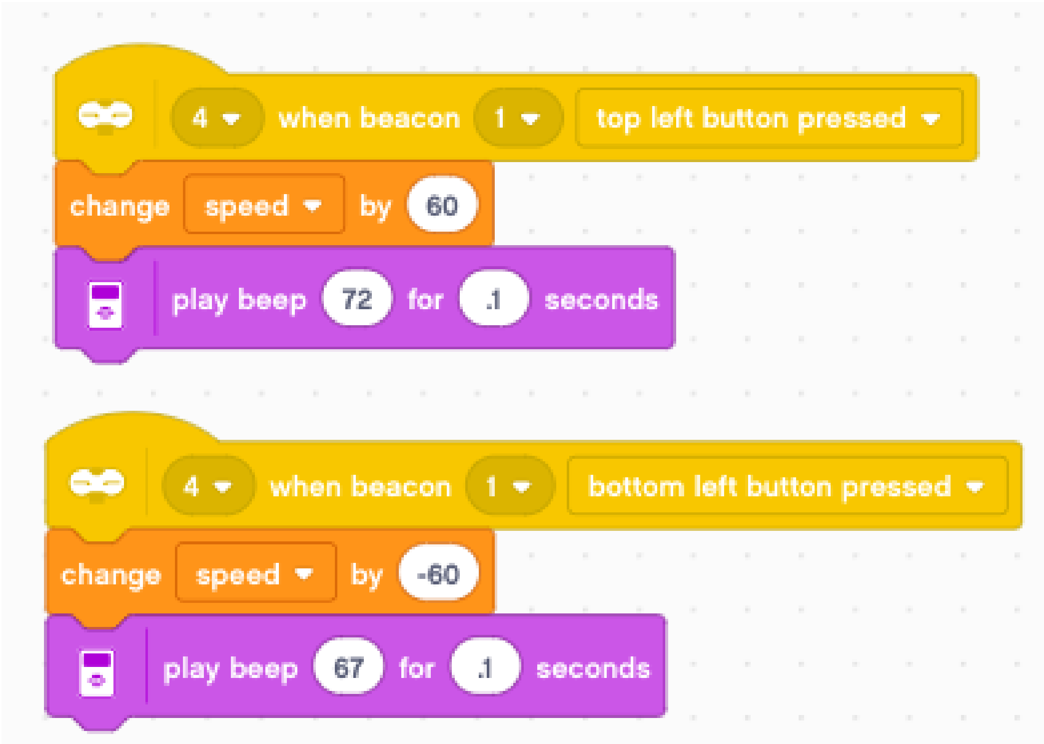


14.9 Driving with the remote control

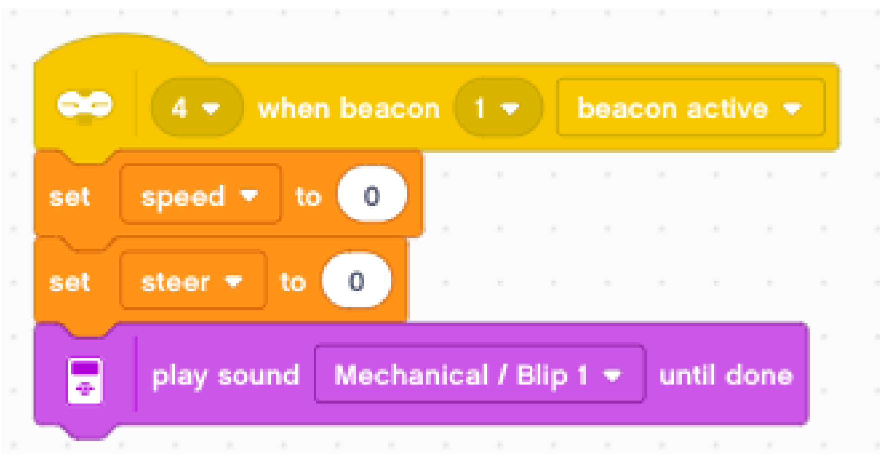
It's now very easy to add remote control functionalty. First we use the large button to implement an emergency stop.



We use the left side buttons to control speed.



and we use the right side buttons to steer.



CHAPTER 15

Puppy

Interact with this charming robot. Pet it, feed it, and experience its reactions.

CHAPTER 16

Color Sorter

Scan and load colord objects and let the Color sorter place the in the right area.

CHAPTER 17

Annex

In this annexe we look at the file format and other technical stuff.

17.1 File format

The **EV3 Classroom** application stores projects with a **.lmsp** extension. You can remember this as *LEGO MINDSTORMS Scratch Program*.

When you create your first project, it is called **Project 1.lmsp**

The file inspector shows:

- Type: EV3 Project Archive
- Size: 40985 bytes (41 KB)

17.2 Open the .lmsp file

The **Project 1.lmsp** is in fact a ZIP file. You can:

- make a copy of it,
- change the extension from .lmsp to .zip
- decompress the ZIP file

You will get a **Project 1** folder with 3 files:

- icon.svg
- manifest.jsn
- scratch.sb3

17.3 The icon.svg file

This file contains an image of the program, probably to be displayed in **Home > Recent projects**.

17.4 The manifest.jsn file

This file contains information about the connection, zoom level, position, etc.


```
{
    "autoDelete": true,
    "created": "2020-02-07T15:58:32.427Z",
    "hardware": {
        "=h{q!08=bSKjnG!;0#eZ": {
            "address": "IOService:/AppleACPIPlatformExpert/PCI0@0/AppleACPIPCI/
↪XHC1@14/XHC1@14000000/PRT2@14200000/EV3@14200000/Xfer data to and from EV3 brick@0/
↪AppleUserUSBHostHIDDevice",
            "description": "",
            "connection": "usb-hid",
            "name": "EV3",
            "type": "ev3",
            "serial": "0016533d0a6c",
            "hubState": {
                "programRunning": false
            },
            "lastConnectedSerial": "0016533d0a6c"
        }
    },
    "id": "lKDY_8BjjFjI",
    "lastsaved": "2020-04-11T16:00:46.620Z",
    "size": 0,
    "name": "Project 1",
    "slotIndex": 0,
    "showAllBlocks": false,
    "state": {
        "playMode": "download",
        "canvasDrawerTab": "monitorTab"
    },
    "version": 1,
    "zoomLevel": 0.8250000000000002,
    "workspaceX": 120.00000000000034,
    "workspaceY": 220.00000000000006,
    "extensions": [
        "ev3events",
        "ev3display"
    ]
}
```


17.5 The scratch.sb3 file

This is a **SB3 file**, based on the MIT Scratch 3.0 format.

You can again:

- replace the .sb3 extension with .zip
- decompress the archive

You will get a folder called **scratch** which contains:

- svg file
- wav file (Meow)
- png file
- project.json file

This JSON file contains:


```
{
    "targets": [
        {
            "isStage": true,
            "name": "Stage",
            "variables": {},
            "lists": {},
            "broadcasts": {},
            "blocks": {},
            "comments": {},
            "currentCostume": 0,
            "costumes": [
                {
                    "assetId": "14d134f088239ac481523b3c2c6ecd8c",
                    "name": "backdrop1",
                    "bitmapResolution": 1,
                    "md5ext": "14d134f088239ac481523b3c2c6ecd8c.svg",
                    "dataFormat": "svg",
                    "rotationCenterX": 47,
                    "rotationCenterY": 55
                }
            ],
            "sounds": [
                {
                    "assetId": "83c36d806dc92327b9e7049a565c6bff",
                    "name": "Meow",
                    "dataFormat": "wav",
                    "format": "",
                    "rate": 44100,
                    "sampleCount": 37376,
                    "md5ext": "83c36d806dc92327b9e7049a565c6bff.wav"
                }
            ],
            "volume": 0,
            "tempo": 60,
            "videoTransparency": 50,
            "videoState": "on",
            "textToSpeechLanguage": null
        },
        {
            "isStage": false,
            "name": "7OQe8zk4TyyTWFib4vkW",
            "variables": {},
            "lists": {},
            "broadcasts": {},
```


(continues on next page)

(continued from previous page)

```
            "blocks": {
                "xYhpfMLy1ynSQzb93O1W": {
                    "opcode": "ev3events_whenProgramStarts",
                    "next": "]}sBQp1+Wg:TSU[;Qw?S",
                    "parent": null,
                    "inputs": {},
                    "fields": {},
                    "shadow": false,
                    "topLevel": true,
                    "x": 34,
                    "y": -19
                },
                "]}sBQp1+Wg:TSU[;Qw?S": {
                    "opcode": "ev3display_displayImageForTime",
                    "next": null,
                    "parent": "xYhpfMLy1ynSQzb93O1W",
                    "inputs": {
                        "DURATION": [
                            1,
                            [
                                4,
                                "2"
                            ]
                        ]
                    },
                    "fields": {
                        "IMAGE": [
                            "Neutral",
                            null
                        ]
                    },
                    "shadow": false,
                    "topLevel": false
                }
            },
            "comments": {},
            "currentCostume": 0,
            "costumes": [
                {
                    "assetId": "93ca32a536da1698ea979f183679af29",
                    "name": "8my-jkz-3zNRywbMFwa-",
                    "bitmapResolution": 1,
                    "md5ext": "93ca32a536da1698ea979f183679af29.png",
                    "dataFormat": "png",
                    "rotationCenterX": 240,
                    "rotationCenterY": 180
                }
            ],
            "sounds": [
                {
                    "assetId": "83c36d806dc92327b9e7049a565c6bff",
                    "name": "Meow",
                    "dataFormat": "wav",
                    "format": "",
                    "rate": 44100,
                    "sampleCount": 37376,
                    "md5ext": "83c36d806dc92327b9e7049a565c6bff.wav"
```

(continues on next page)

(continued from previous page)


```
                }
            ],
            "volume": 100,
            "visible": true,
            "x": 0,
            "y": 0,
            "size": 100,
            "direction": 90,
            "draggable": false,
            "rotationStyle": "all around"
        }
    ],
    "monitors": [],
    "extensions": [
        "ev3events",
        "ev3display"
    ],
    "meta": {
        "semver": "3.0.0",
        "vm": "0.2.0-prerelease.20190619042313",
        "agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36␣
↪(KHTML, like Gecko) EV3Classroom/1.0.0 Chrome/69.0.3497.106 Electron/4.0.4 Safari/
↪537.36"
    }
}
```

CHAPTER 18

Indices and tables

- genindex
- modindex
- search