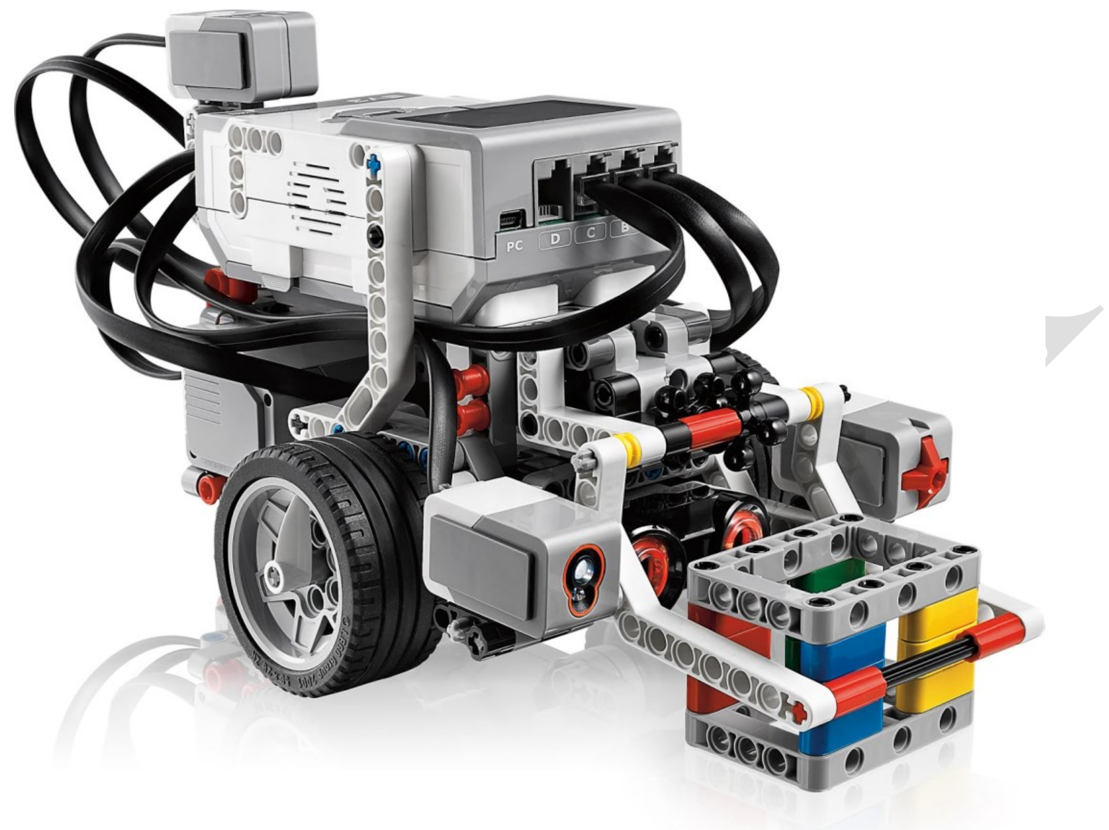


Programming Lego MINDSTORMS Robots using NI LabVIEW®

Author: Pooyan Nayyeri
Translation to English: Pooyan Nayyeri & Mahbod Tajdini

Contents

Preface

Robotics is a cool topic and Lego MINDSTORMS is the easiest way to program a real robot without getting your hands dirty with all electronics and mechanical challenges. I wrote this book on 2018 originally in Persian, while I was instructing a course on the same subject of this book at the robotics engineering center, located at the University of Tehran. Right now, Lego MINDSTORMS robots are no longer available for purchase (which is very disappointing since there is no proper alternative for them). However, I decided to translate this book to English to provide a good resource for programming MINDSTORMS robots using NI LabVIEW.

I hope you find it useful for your projects and in your learning journey. You can always contact me via email (pnnayyeri@gmail.com) to discuss anything related or unrelated to this book.

Pooyan

Spring 2022

Chapter 1
An Introduction to LabVIEW

AN INTRO TO LABVIEW

LabVIEW software is developed by **National Instruments**. This software is known with a graphic programming language (G[1]) allowing users, including engineers, to create complex programs. The name of the program stands for **Lab**oratory **V**irtual **I**nstrument **E**ngineering **W**orkbench. The primary purpose of developing this software is to build and design control and measurement systems by engineers and scientists.

Any program coded in **LabVIEW** is called a **VI** or **V**irtual **I**nstrument. Each **VI** consists of three parts: *Block Diagram*, *Front Panel*, and *Connector Panel*.

Front Panel or the user interface contains *controls* and *indicators* of the program. Controls are the inputs from the user to the program, and Indicators are the outputs displayed in the form of numbers or symbols.

Behind the scenes of this panel is the main program or **Block Diagram**, which is the central part of the VI. In this section, various functions and commands are connected in blocks to determine the process and purpose of the program. All controls and indicators in the Front Panel have links in the Block Diagram section, called **Terminals**, and thus communicate with the core of the program. The blocks in this section receive the input from the Front Panel, and after processing them, transfer the result to the Front Panel.

Finally, in the **Connector Panel** section, a VI can be used as a block in the Block Diagram of another VIs.

In fact, it can be said that a VI or virtual instrument is like a real instrument. Take a computer, for example. All the buttons on the keyboard and mouse, and any other input on this computer, are the Front Panel, which is provided directly to the user. Also, the screen, speakers and lights on the computer are part of the Front Panel because they show the computer's output to the user. Other parts of this computer are hidden from the user. The user does not deal directly with them and only performs mathematical and logical processing, and operations such as motherboard, GPU, RAM, CPU, etc., belong to the Block Diagram section.

Moreover, suppose we connect this computer to another device or computer. In that case, we have provided the input and output of this computer to other computers, which is the function of the **Connector Panel**. So just as all the physical instruments somehow include these three parts, every VI also includes these three parts.

Start Programming with LabVIEW

To start working with the LabVIEW program, after installing the software on your system, we can run the software from the shortcut created on the desktop or the Start menu:

[1] It differs from the g-code programming language

Figure 1 LabVIEW software icon

The first window that appears after running the software is the main page which is as follows:

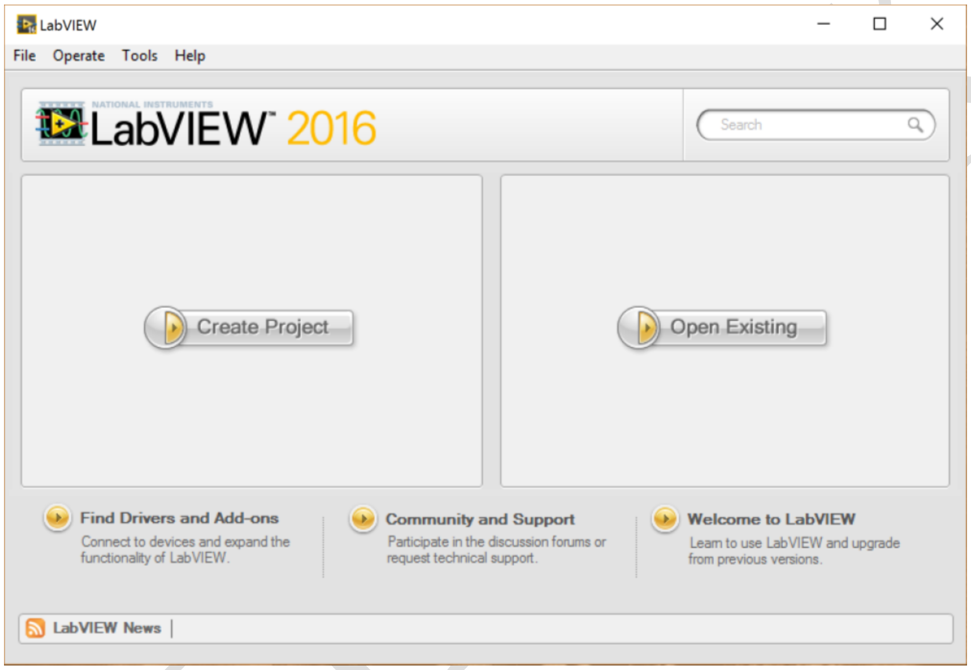

LabVIEW
File Operate Tools Help
LabVIEW 2016
Search
Create Project
Open Existing
Find Drivers and Add-ons
Connect to devices and expand the functionality of LabVIEW.
Community and Support
Participate in the discussion forums or request technical support.
Welcome to LabVIEW
Learn to use LabVIEW and upgrade from previous versions.
LabVIEW News

Figure 2 The main page of the software

In this window, you can define a new project or reopen previous projects. Each project can contain multiple VIs. To start building the program, go to **File → New NXT/EV3 → Targeted VI** from the menu on the top and click on it. Now the two windows *Front Panel* and *Block Diagram* related to the created VI are displayed.

Front Panel and Block Diagram Windows

The Graphical User Interface (**GUI**) is displayed in the Front Panel window. As the name implies, you can receive inputs from the user and display outputs to him/her using the built-in tool in this section. Inputs and outputs can be in a *numerical*, *binary*, *string*, and so on. In this software, the inputs that are taken from the user are called **Control,** and the outputs that are displayed to the user are called **Indicators**.

As mentioned, the Front Panel window is the only interface between the user and the program. Therefore, the Block Diagram window is used to develop a program.

The Block Diagram window is a space for determining the process and structure of the program. In this environment, by using the blocks provided to the user and arranged in different groups, the user designs his/her program structure. These groups include *numerical, array, Boolean, comparison*, *etc.* groups. Depending on the type of software used, these groupings change and provide the tools the user needs. For instance, this software shows the user-related groups for this purpose when using the MINDSTORMS module. On the other hand, due to the connection of this section with the Front Panel window, terminal blocks are created for using inputs and sending outputs, which is, in fact, a gateway for transferring information from the Front Panel to the Block Diagram and vice versa.

Intro to Virtual Instruments (VI)

As mentioned, any LabVIEW software program is called a virtual tool or VI, consisting of two main parts: **Front Panel** and **Block Diagram**. Here, we want to briefly get acquainted with the Front Panel and Block Diagram environments.

After creating a new VI in the software, two windows called Front Panel, and Block Diagram are displayed. You can enlarge any Front Panel or Block Diagram windows on the screen by pressing *Ctrl+/*. *Ctrl+E* is also used to move between the two windows. When you need to have both windows in front of you simultaneously, press *Ctrl+T*, in which case both windows will be next to each other.

The overall layout of the two windows is very similar, so here are the key/common parts of the two windows.

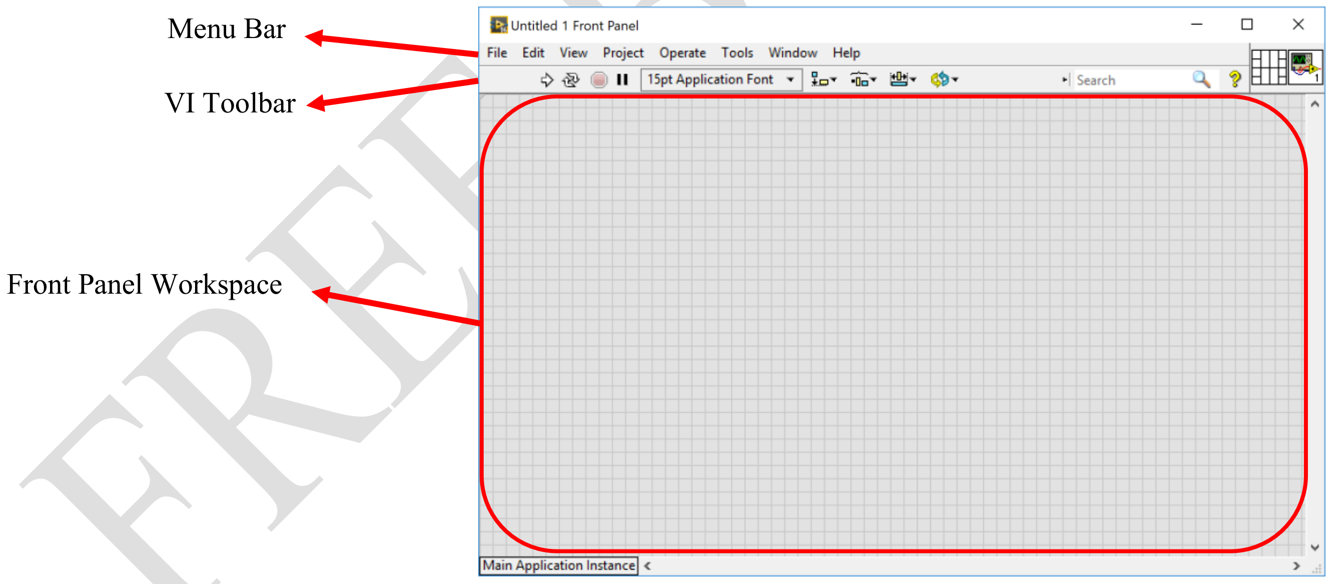


Figure 3 The main sections of the Front Panel window

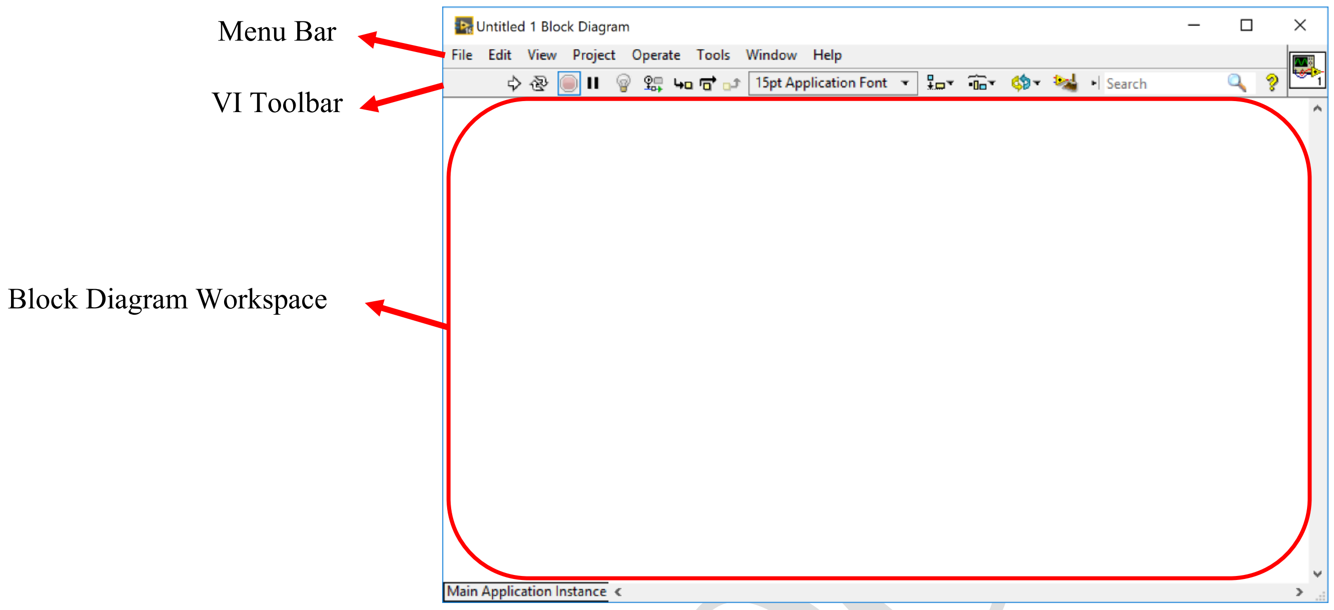


Figure 4 The main sections of the Block Diagram window

As you can see, this window consists of several parts. The top bar of this window, like most programs, is called the **Menu Bar**, which is commonly used to create, store, or load a VI.

The bar at the bottom of the menu bar is known as the **VI Toolbar**. This bar can be used to run and stop VI as well as arrange blocks.

A major part of these windows is the Front Panel or Block Diagram workspace. The workspace is the space available to the user that shows the types of inputs and outputs or blocks. Because of the importance of the VI toolbar, we introduce its buttons:

Run: By clicking on this button, the corresponding VI is executed, and processing is done. This option is also available via the *Ctrl+R* hotkey.

Run (running): This is how the Run button looks when the VI is running.

Run (broken): When the Run key looks like this, there is an error in VI; by clicking on this button, the list of errors is displayed.

Run Continuously: Selecting this button will run the program continuously until you stop or abort the program.

Abort Execution: This option stops the program.

Pause: It pauses the program. If the program is paused, by clicking on this button, the program will resume.

Making a Simple Program

To start with VI, let us look at a simple example.

After creating the first VI and displaying the *Front Panel* and *Block Diagram*, right-click anywhere in the Front Panel environment to display the **Control Palette**:

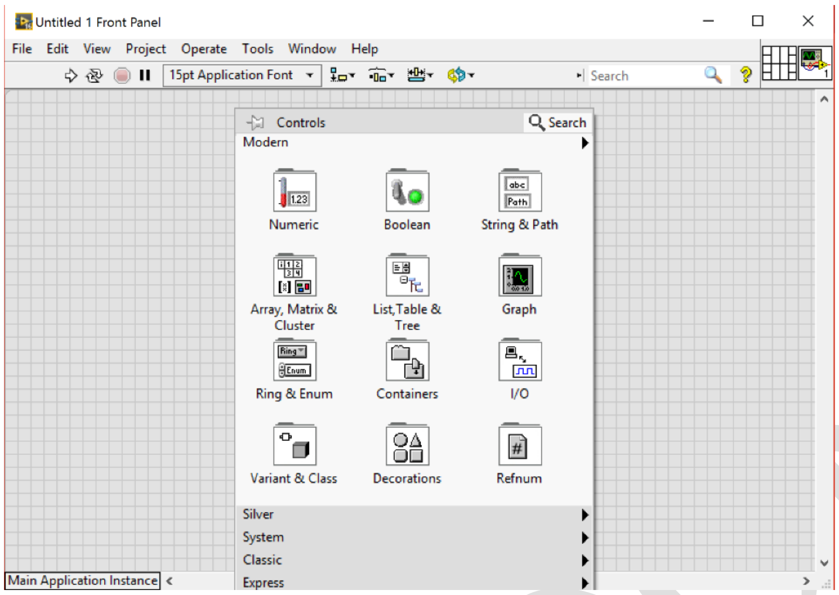


Figure 5 Displays the control panel by right-clicking in the Front Panel window

As you can see, different control and indicator groups are displayed. If you need to search for control or indicator, you can use the **Search** field on the control palette. If you do the same operation (right-click) in the Block Diagram environment, you will see that the ***Functions Palette*** opens for you, which includes different groups:

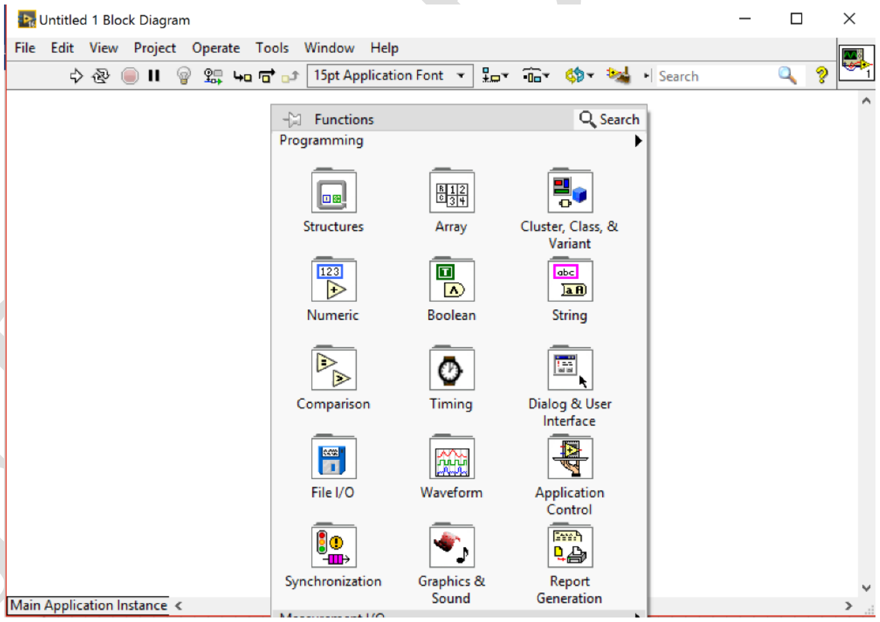


Figure 6 Displays the function palette by right-clicking in the Block Diagram window

Now we want to take a number as input from the user in the Front Panel window and display the same number as output in the same window. For this purpose, using the control palette, enter the **Numeric** group and select **Numeric Control**:

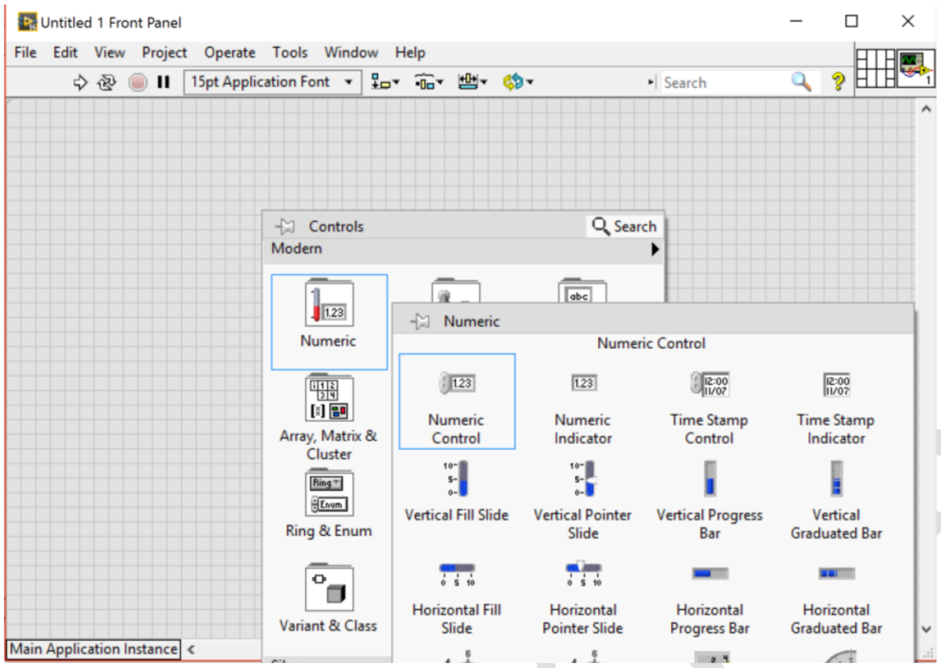


Figure 7 Selecting the Numeric Control block

By selecting this option, you can place the Numeric Control anywhere in the Front Panel. Left click the mouse to place this control:

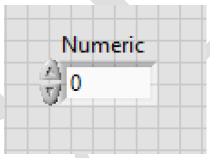


Figure 8 Placing the Numeric Control block in the Front Panel window

The number shown in this control field is the input that the user gives to the program. Clicking the small arrows to the left of this field changes the value of this number. You can also set this control by selecting the number and typing the desired value. Now, you can see that a block called **Numeric** has been created in the Block Diagram window, which is a terminal linked to the numeric control. Upon further examination of the Numeric block, this block has a node:

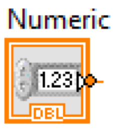


Figure 9 Numeric block associated with the Numeric Control block

This node is a representative for the number entered by the user, accessible in the Block Diagram space. In the next step, right-click on the Front Panel workspace and select the Numeric group and select the **Numeric Indicator**. By placing this block in the Front Panel environment, an indicator called **Numeric 2** is created. This indicator is responsible for displaying numbers to the user. This number can be anything and depends on the program and its block diagram.

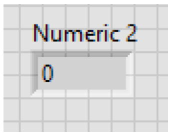


Figure 10 Numeric Indicator block in the Front Panel window

Again, as before, a block is created for this indicator in the Block Diagram window. This block also has a node that displays the outputs from the Block Diagram space to the user in the Front Panel area:

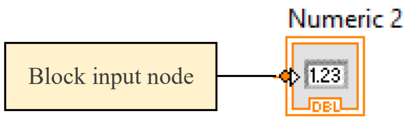


Figure 11 Numeric block associated with the Numeric Indicator block

However, if we look a little more closely, the two blocks, Numeric and Numeric 2, although each has a node, are different. In the Numeric block, which is related to the control or numeric input, the existing node is output, which can input other blocks. But in the Numeric 2 block, which is the indicator or numeric output terminal, the existing node is an input displayed to the user by connecting any number to this node.

To display the input number in the output, connect the Numeric block output node to the Numeric 2 input node in the Block Diagram space. To do this, click on the Numeric output node to create a connection from this block, then click on the Numeric 2 input node to connect the two blocks. By doing this, we have designed a program to receive a number from the user and display the same number to the user:

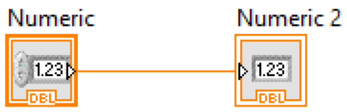


Figure 12 Connecting Numerical indicator and control blocks to each other

Now, we only need to **Run** the program. Since the designed program is extremely simple, the operation is performed instantly, and the output is displayed. By changing the input value and running the program, the output also shows the same input value:

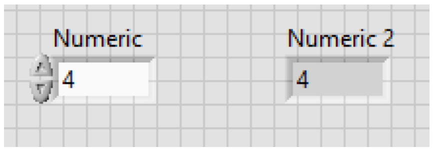


Figure 13 Displays the input number at the output in the Front Panel window

To run the program continuously and update the output instantly, you need to click the **Run Continuously** button so that your program always runs. In this case, by changing the input, the output also changes, and there is no need to re-run the program every time.

DATA TYPE

LabVIEW stores data in various forms in memory. The most important types of data include the following:

- **Boolean**: This type of data can only have two modes: *FALSE* (or zero) and *TRUE* (or any non-zero value). This type of data is stored as 8-bit values in memory.
- **Integer**: This data type is stored as *integers* (non-decimal numbers) that can be expressed with and without signs (signed and unsigned). Integer data are categorized according to the bits assigned to this data:
    - Byte Integer: 8 bits
    - Word Integer: 16 bits
    - Long Integer: 32 bites
    - Quad Integer: 64 bits
- **Floating-point**: This data is used to display *real numbers* using the scientific symbol method on a computer. Floating point data in LabVIEW can be stored with two precisions: single-precision and double-precision.

Numerical Data

In the next step, we must process the data entered into the program and display the desired output to the user. This process can be a numerical operation such as addition, multiplication, etc., or logical and comparative operations or any other operation. Also, program inputs can be from the user, sensors, or any other way.

Numerical Functions

Built-in functions can be used for numerical data processing in LabVIEW. In the following, we will introduce these functions to get familiar with them:

- **Add**: Adds two numeric values together and outputs them through the output node.
- **Subtract:** Subtracts two numeric values from each other and outputs them through the output node.
- **Multiply**: The product of multiplication of two numeric values are outputted through the output node.
- **Divide**: The result of division of two numeric values will be outputted through the output node.
- **Quotient & Remainder**: Outputs the quotient and remainder of two numeric values into two output nodes.
- **Increment**: Increases the input number by one unit.
- **Decrement**: Reduces the input number by one unit.
- **Absolute Value**: Produces the absolute value of the number.
- **Square root**: Outputs the square root of the input number.
- **Square**: Outputs the square (power of 2) of the input number.

For example, we want to write a program that takes two numbers from the user and displays the sum of those two numbers in the output. As in the previous section, we first place two numeric controls and a numeric indicator in the Front Panel window:

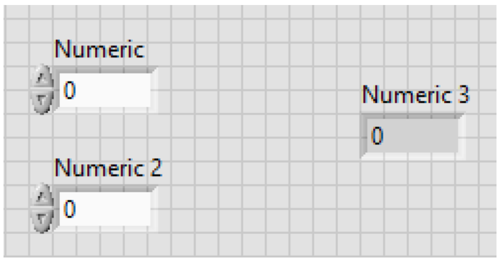


Figure 14 Creating two numeric control blocks and one numeric indicator block

Since we want to add two numbers together, in the Block Diagram space, we use the *Add* block in the Functions palette and the *Numeric* group:

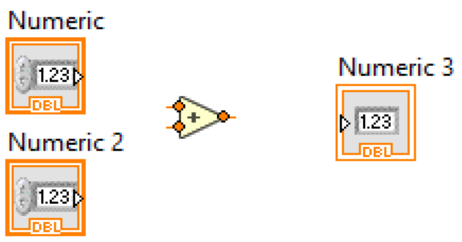


Figure 15 Adding the Add block in the Front Panel

As it can be seen, this block has three nodes, the nodes on the left are the input, and the node on the right is the output. By connecting two numeric values to the nodes on the left, the node on the right shows the sum of them. Then we connect the blocks as follows:

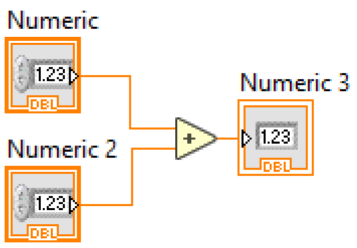


Figure 16 Connecting the numeric control and numeric indicator blocks to the Add block

By running the program continuously (Run Continuously), you can enter both numbers in the numerical control field to see the sum of the sum in the numeric indicator:

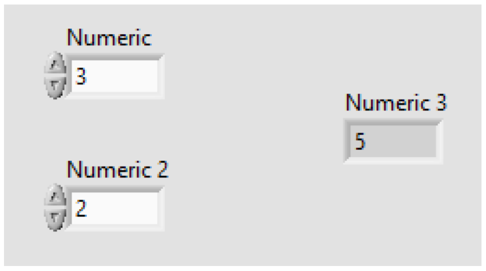


Figure 17 shows the sum of two inputs in the numeric index block

Binary Data

The example above is for the sum of two numeric values. Data operations can be performed logically, comparatively, and so on. To examine another type of data, we consider binary or binary data. This data can have only two values at a time: 0 or 1. These two values can be considered in different ways. For example, any binary data can be considered as a switch, with a value of 0, the switch is off, and with value 1, the switch is on.

Like numerical data for which mathematical operations such as addition and multiplication are defined, specific operations are defined for binary data as well. The most essential and original operations can be mentioned as follows:

- **And**: This operation is like the multiplication for binary data and is defined for the operation between two binary data. The answer to this operation is obtained by multiplying two binary data with each other. For example, if we consider operations 1 and 0, $0 \times 1 = 0$, then 0 *AND* 1 = 0. This operation has a response of 1 if both data are one and otherwise a value of 0 as the answer.
- **Or**: Equivalent to the summation in numerical data. The answer to this operation is equal to 1 if at least one of the data has a value of 1; otherwise, it is equal to 0.
- **Not:** With the help of this operation, the desired data can be negated. If our data is 1, it makes it 0, and if it is 0, it makes it 1.

Let's say, we want to have two radio buttons that turn on a light if both are on, so we can design a program like the one below:

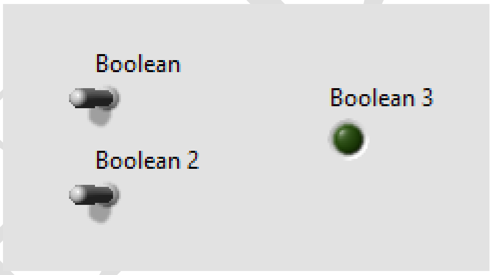


Figure 18 Creating two binary control blocks and one binary indicator block

The two switches are used as the program's input, and the light is used as the program's output. We use the *AND* block to turn on the light when **both** switches are on. To do this, right-click in the Block Diagram window and select the *AND* block from the Functions palette and the Boolean group:

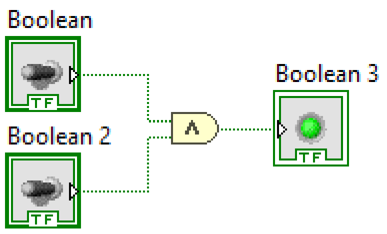


Figure 19 Connecting blocks to the AND blocks

By continuously running the program, the light turns on when both switches are on:

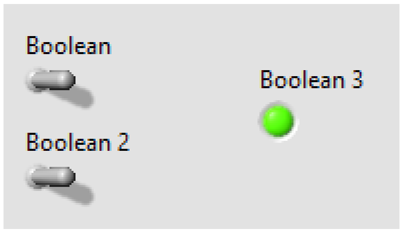


Figure 20 running the logical AND program

Array Data

Arrays represent vectors or matrices. Like a vector or matrix, an array contains several data of the same type. Each array has two parts: the *array* (or element) and the *dimensions* (array size). Arrays can include numeric, binary, string, and so on. The array can be used as both input (control) and output (indicator).

The main advantage of arrays is that a large amount of data of the same type is available, or we will do the same operation on several data of the same type.

Each element in an array has a specific index. An index is like the address of an element in an array.

* Indexing in arrays starts from zero, which means that the first array index is zero.

1-D Array

To create a one-dimensional array in LabVIEW, right-click on the Front Panel window and go to Array, Matrix & Cluster from the Controls palette and select Array:

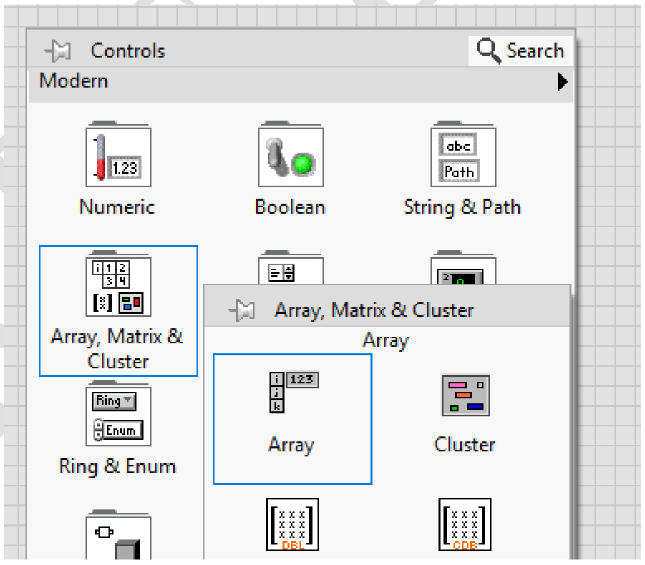


Figure 21 Selecting an array block in the control palette

By placing an *Array* anywhere on the page, you can create an array:

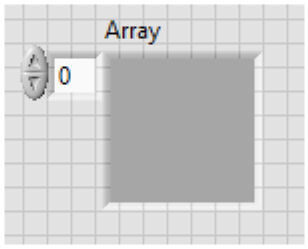


Figure 22 Creating an Array block in the Front Panel

As it can be seen, to the left of the created array is a numeric control that represents the index of the element in the left cell. In fact, by selecting any value in this numeric control, this array displays elements starting from this index.

As mentioned, an array can contain data such as numeric, binary, and so on. Initially, the inserted array is empty. In this section, depending on the type of data considered, a numeric constant, a binary variable, or etc. can be used to form an array of the same type. For example, here we want to create an array with numeric data; for this purpose, from the control palette, in the *Numeric* section, we add a *Numeric Control* to the composed array:

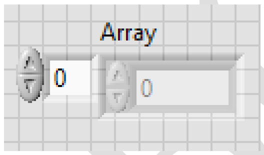


Figure 23 Inserting numeric control into an array

In this case, the created array is the input (control) type and contains numeric variables. To change the size or dimension of the array, you can use pressing and holding the cursor on the enclosing box of the numeric control and then moving the highlighted blue dots:

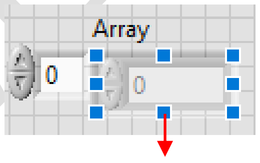


Figure 24 Resizing an Array

By moving the blue dots, the size of the array also changes. For creating a one-dimensional array, it can be transformed into a columnar vector:

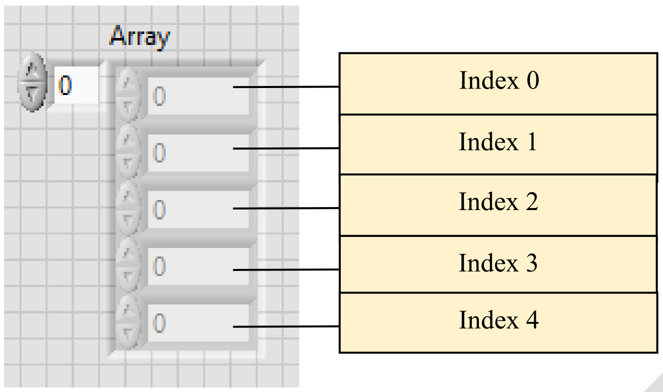


Figure 25 Creating a one-dimensional array with 5 elements

Note that the number is not set up until a number is assigned to the array. That is, if we input numbers into the only the first 4 elements of the above 5x1 matrix, this array will act like a 4x1 matrix:

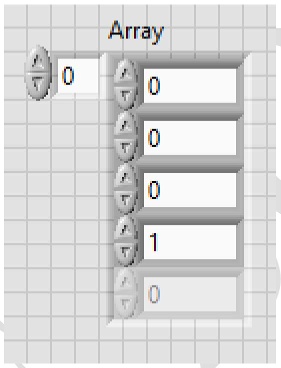


Figure 26 Setting up the elements of an array

By assigning the number 1 to the fourth layer of this array, all the layers before it will take the default value of zero, and the layers after it will not be set up.

Like when creating numeric or binary variables created a terminal in the Block Diagram space, a terminal is created for this array in the block diagram space. This terminal can be used in the programming process:

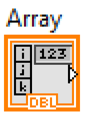


Figure 27 the terminal created for an array in Block Diagram

Constant Array

Because the user does not interfere in creating or changing the constant array, we start from the Block Diagram environment and create a constant array via the Functions palette → the Array section → Array Constant:

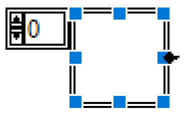


Figure 28 Creating a constant array

As before, we must specify the data type of constant values. For example, we want this array to have binary constants. To this end, go to the Boolean section of the Functions palette and select the *True Constant* block and place it inside the array box:

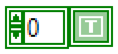


Figure 29 Formation of a constant array of binary type

As before, this array can be defined as a vector that has several constants:

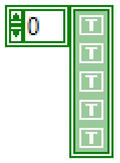


Figure 30 Resizing a constant binary array

Multi-dimensional Array

To build a multidimensional array or matrix, we follow the same procedure to build a one-dimensional array. For instance, to create a two-dimensional matrix, place an array in the Front Panel and then right-click on the array control (input number at the top left of the block) and select **Add Dimension**, to add another dimension to the array:

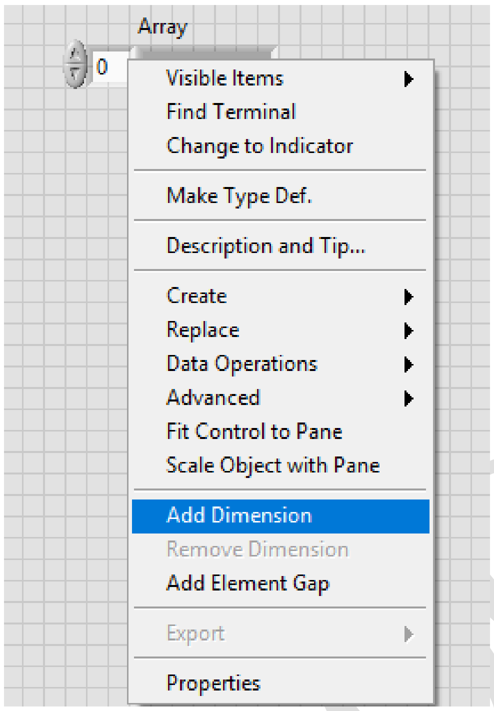


Figure 31 Adding dimension to array

In this case, another array control is added under the previous control. The upper control represents the matrix row index, and the lower control represents the matrix column index.

Array Functions

As we know, arrays are like variables that store large amounts of data of the same type. In most cases, it is necessary to perform operations and processing on the existing data to obtain the desired result or use it somehow in our programming process. Like numeric or binary variables, functions are defined for arrays, which we will introduce in the following.

Array Size

One of the most critical and basic operations that we will deal with in arrays is to get the dimensions of an array. There is a built-in block in LabVIEW for this purpose. This block outputs a number or a one-dimensional array (depending on its dimensions) for the number of indexes in each dimension.

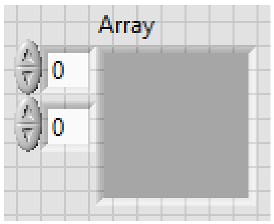


Figure 32 An array with two dimensions

For example, consider the following two-dimensional matrix that has the dimension of 5x4 (5 rows and 4 columns):

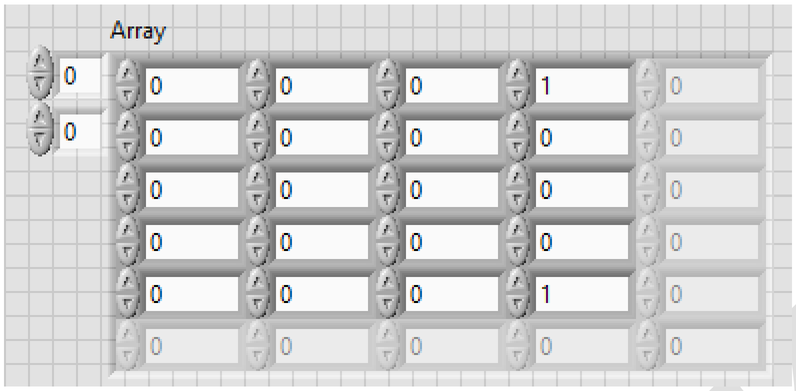


Figure 33 Creating an array with two dimensions

To get the dimensions of this matrix in the Block Diagram environment, select and place the *Array Size* block from the Functions palette, in the Array section. By connecting the above matrix to the input node of this block and connecting a one-dimensional array with at least two elements to the output node (because each element is supposed to represent the number of elements in each dimension of the input matrix), we have the following program:

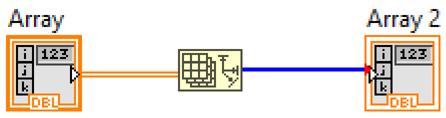


Figure 34 Connecting an Array Size block to a populated array

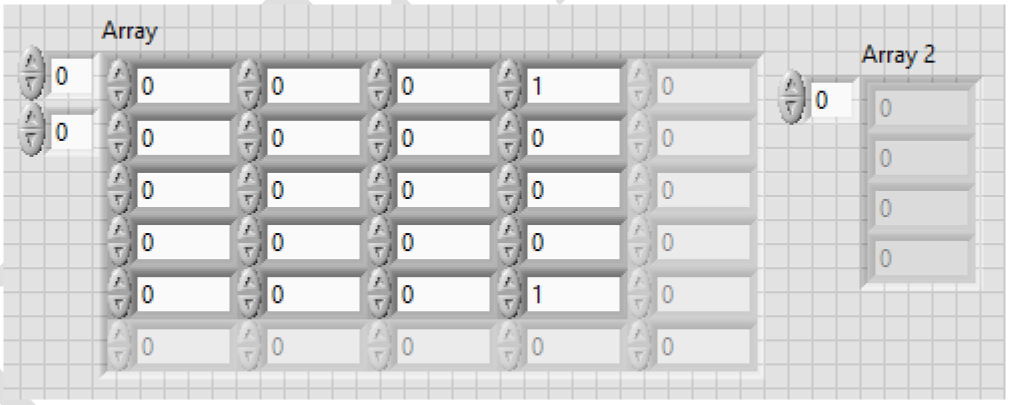


Figure 35 Displaying the array size in the Array 2 block

As expected, by running the program, values of 5 and 4 will appear in the first two elements of the size array (Array 2), respectively:

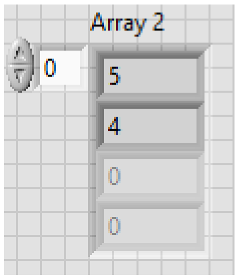


Figure 36 Showing the size of the array

As you can see, only the first two components are used in the dimension matrix, and the other components are not defined and set up.

Index Array

Sometimes it is necessary to have access to one or more specific elements. The prerequisite for this action is knowing the index of the desired array(s). The *Index Array* block can be used for this purpose. This block is located in the Array section of the Function palette. By connecting the array and the index number(s) to the inputs of this block, the corresponding elements can be extracted at the output.

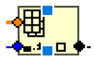


Figure 37 Index Array block

To call multiple indices, resize it using the two blue squares at the top and bottom of the block to increase the number of indices/nodes:

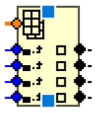


Figure 38 Resizing the Index Array block

Replace Subset

Replace [Array] Subset block is used to replace an element, row, or column in an array. For this purpose, you need to connect to the input nodes of this block to the input array, the indices, and the desired element(s) to output an array with the replaced element(s).

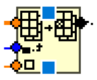


Figure 39 Replace Subset block

You can change the size of this block depending on your needs to increase the number of indices.

Other Functions

The following is a brief introduction to other useful functions for arrays:

- **Insert into Array:** Adds an element/array in the specified index of an n-dimensional array.

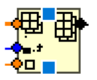


Figure 40 Insert into Array block

- **Delete From Array**: Removes an element/array from the specified index and both the modified array and the deleted subset.

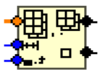


Figure 41 Delete from Array block

- **Build Array:** Creates an n-dimensional array with the given elements.

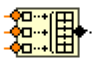


Figure 42 Build Array block

- **Initialize Array:** Integrates several arrays or adds one or more elements to the original array.

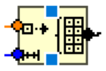


Figure 43 Initialize Array block

- **Array Subset:** Extracts part of the array using the specified index and the given length.

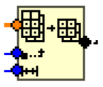


Figure 44 Array subset block

- **Array Max & Min**: Gives the maximum and minimum values in the array as well as their indices.

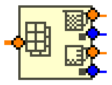


Figure 45 Array Max & Min block

STRUCTURES AND LOOPS

For loop

The For loop repeats a process for a certain number of times. To access this loop in the Block Diagram environment, go to the *Structures* section and select *For Loop*:

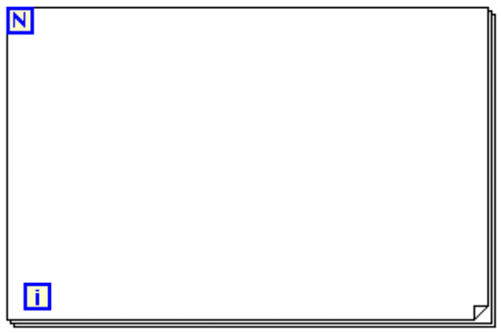


Figure 46 For loop in the Block Diagram environment

Each program designed inside this loop will be repeated N times, and then the program will exit the loop. To determine the frequency of repetition of this loop, we need to connect the node of the N block to a numeric value. This numeric value can be obtained from a constant number, numeric control, or any other method.

Block i inside this loop is the *iteration* or the loop counter that can be used as an output node in the program. This value always starts from 0; meaning that i is equal to zero when the program starts.

The For loop holds for the condition N>i and will be repeated. The value of i is incremented each time the loop is repeated. So, if we set the value of N to zero or a negative number, the For loop will never run. Also, variables i and N are of Integer type. If float values are connected to these variables, they are automatically rendered to the nearest integer number.

For Loop with Conditional Terminal

An important feature of the For loop is that you can also exit the loop for a specific condition in addition to repeating the loop several times. By right-clicking on the For loop and selecting the *Conditional Terminal*, a condition block will be created for the For loop. If this condition is satisfied, the program will exit the For loop:

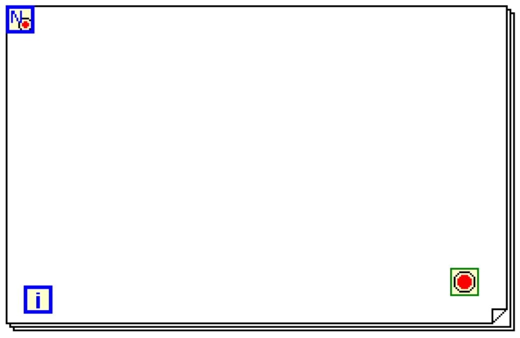


Figure 47 For loop with stop condition block

The created condition acts like the ***while*** loop, which we will talk about in the next subsection.

While Loop

The While loop will be repeated until the specified condition is satisfied. The structure of this loop is as follows:

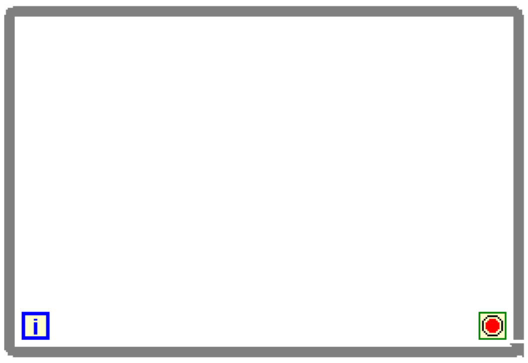


Figure 48 While loop in Block Diagram

Block i, like the For loop, gives the number of repetitions of the program. The condition block is located on the right-bottom of the loop by default. This block has an input node that must be connected to a binary variable. The condition block can be considered a condition in two ways: 1) Stopping the loop when the input node is True; 2) Continuation of the loop when the input node is True.

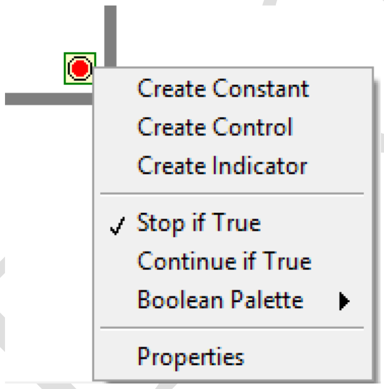


Figure 49 Changing the stop condition

By default, the condition block assumes the first case. In this case, we need to connect a constant value of zero or *False* to the condition block to create an infinite loop (endless loop repetition). By right-clicking on the condition block and selecting Create Constant, a constant value of zero can be attached to this block. In this case, the program will never exit when entering the While loop.

Shift register

The register shift can be used to transfer the values obtained in each iteration of the loop (For and While loops) to future iterations. Right-clicking on the loop and selecting *Add Shift Register* will create two blocks on both side of the loop, as shown below:

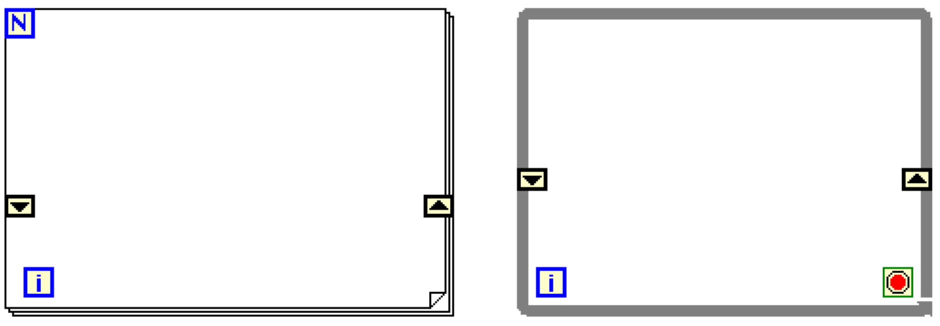


Figure 50 Creating a Shift Register in the While and For loops

The block on the right is marked with an up arrow that stores the values in each iteration. In this case, the software automatically transfers this value to the next iteration. The shift register can receive any type of data, and of course, both data connected to the two register shift blocks must be of the same type.

* The number of shift registers is not limited, and they can be added to the loop as many times as needed. You can also resize the left block of the shift register to access the values in previous iterations.

Case structure

This structure has two or more cases that if the condition of each item is met, the program of that section will be executed. This structure executes only one item at a time. The value given to the input of this structure determines which case will be executed.

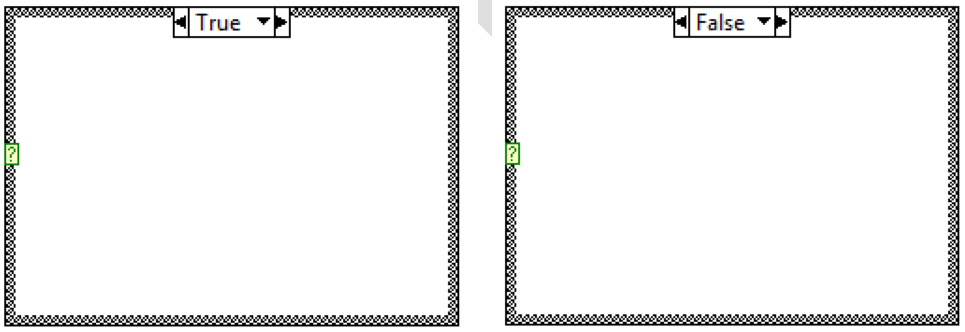


Figure 51 Two different cases in the Case Structure

The block on the left side of the wall of this structure is designated by "?" and is the condition of the structure. If this value matches any of the items, the program for that item will run. By default, this structure has two items: one (True) and zero (False). However, this structure can also be used for other datatypes.

Items of this structure are located in the drop-down menu above it. You can change the value of each item by selecting it. If the input to this structure is not binary, one case must be specified as the default case; so that the program can be executed if other cases do not match. A case can be chosen as default by and right-clicking on the desired case and selecting the *Make This the Default Case* option.

Flat Sequence

This structure is used to create an order in the execution process of different parts of the program. As mentioned, LabVIEW does not necessarily run from left to right or vice-versa, and the program will run whenever values are available for input nodes.

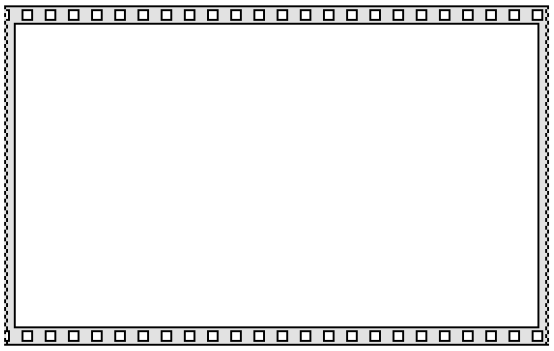


Figure 52 Flat Sequence structure

Each part of a Flat Sequence structure is called a frame, and each Flat Sequence structure can contain one or more frames. Since this structure is executed from left to right and when all the required data in a frame is available, this structure can be utilized to accurately plan the desired process of program execution. To add a frame to this structure, right-click on it and click *Add Case Before* or *Add Case After*.

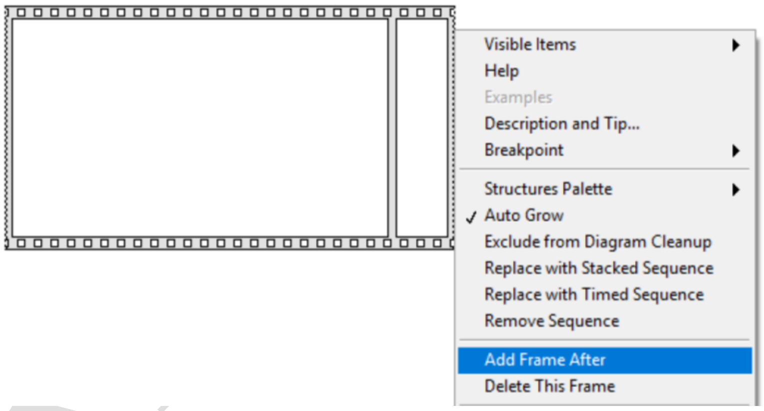


Figure 53 Creating a new frame in the Flat Sequence structure

Data can be outputted once each frame is completely executed, so the input of one frame may depend on the output of other prior frame(s).

No data will be outputted from this structure until the last frame's operation is completed.

CHAPTER 1 – PROBLEM SET

1. Design a program to receive the temperature in Celsius from the user and display it in Fahrenheit (use °F = 1.8×°C+32 to convert Celsius to Fahrenheit).

2. Using the For loop and without using the Multiply block, design a program to take two numbers from the user and display the product of their multiplication.

3. Use a For loop to create a timer to display the elapsed time when the program starts.

4. Create a simple calculator that takes two numbers from the user and determines one of the four operators (addition, subtraction, multiplication and division) to perform this operation on those two numbers and display the result.

Chapter 2
Programming Mindstorms

An Intro to the Mindstorms Module

LabVIEW is designed to connect to various hardwares. Thus, it is necessary to install driver(s) to connect LabVIEW to the hardware. The provided MINDSTORMS module is used to connect LabVIEW to the Lego MINDSTORMS package.

Using this module, the designed VI can be implemented in two ways: 1) running VI on a computer and exchanging commands and information with EV3/NXT brick, 2) running VI independently on EV3 / NXT brick. This module can also be used to control TETRIX motors and servos connected to EV3/NXT brick. In this section, we will implement a VI using both ways.

To get started with this module, go to the File section of the main screen of the program and select the New NXT/EV3 → Robot Project option.

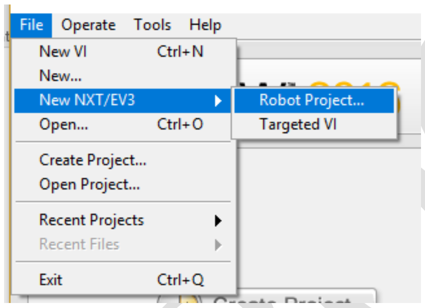
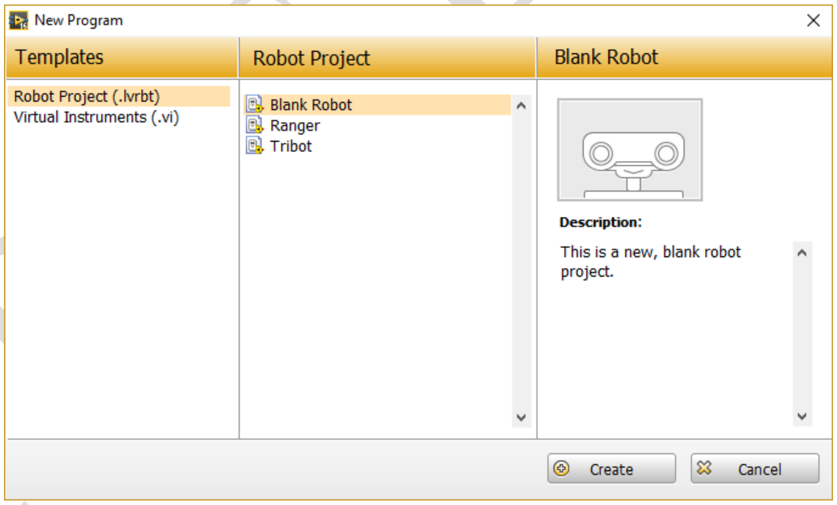


Figure 1 Creating a new EV3 project

By selecting this option, the following window will be opened:

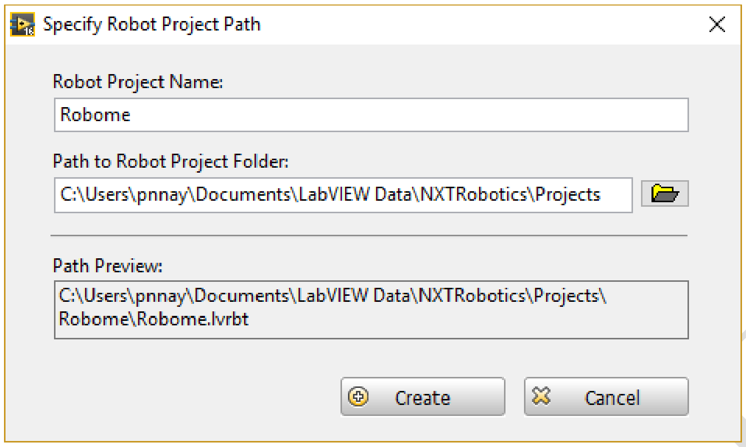


Description:

This is a new, blank robot project.

Figure 2 EV3 project creation window

Click the *Create* button and choose a name for your project and then click the *Create* button again:


Specify Robot Project Path

Robot Project Name:
Robome

Path to Robot Project Folder:
C:\Users\pnnay\Documents\LabVIEW Data\NXTRobotics\Projects

Path Preview:
C:\Users\pnnay\Documents\LabVIEW Data\NXTRobotics\Projects\Robome\Robome.lvrbt

Figure 3 Selecting a name and path for the robot project

This defines a new robotic project that can be controlled through the following window:

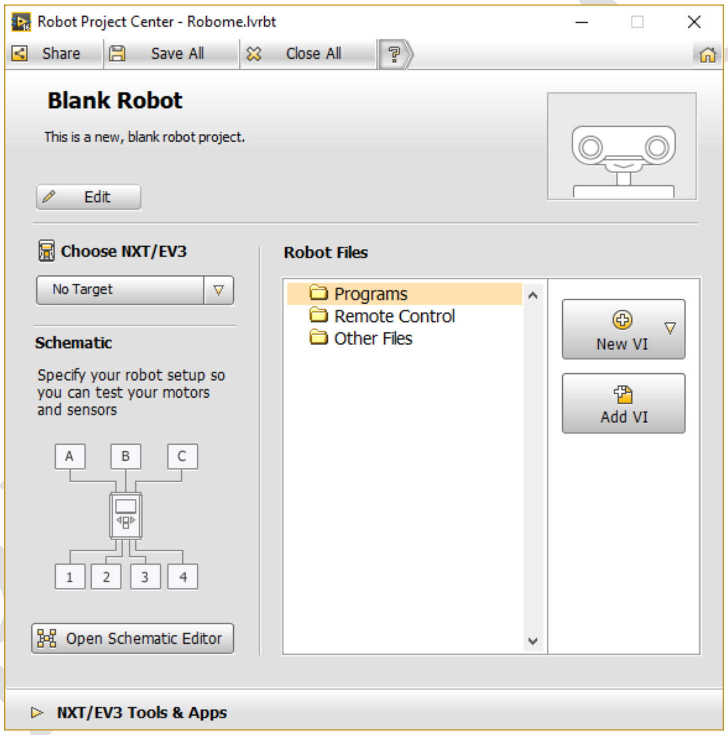

Robot Project Center - Robome.lvrbt

Figure 4 Project Center window

At the top of this window is the robot's name, descriptions, and image; you can edit it by clicking the *Edit* button.

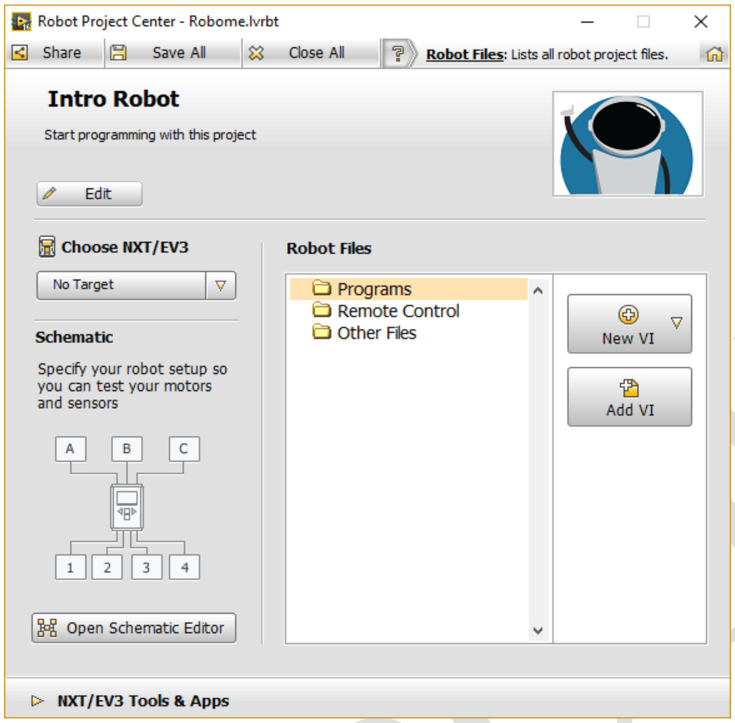

Robot Project Center - Robome.lvrbt
Share
Save All
Close All
Robot Files: Lists all robot project files.
Intro Robot
Start programming with this project
Edit
Choose NXT/EV3
No Target
Schematic
Specify your robot setup so you can test your motors and sensors
A
B
C
1
2
3
4
Open Schematic Editor
Robot Files
Programs
Remote Control
Other Files
New VI
Add VI
NXT/EV3 Tools & Apps

Figure 5 Enter the robot's name, description, and image

In the *Choose NXT/EV3* section, select the desired brick that is connected to the computer. To do so, connect your brick to the computer with a USB cable and after the brick is identified, select it from the drop-down menu in this section:

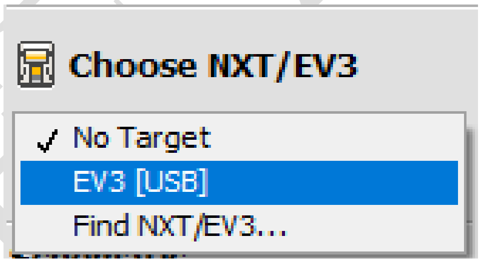
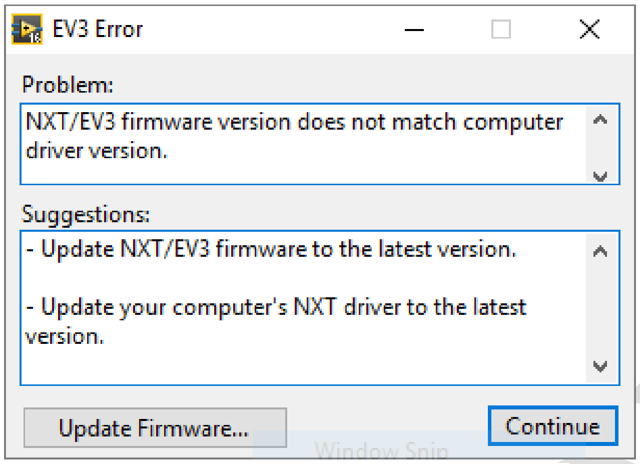


Figure 6 Selecting an EV3 brick connected to the computer

If you get an EV3 error message after selecting the brick, click the *Update Firmware…* button to update your brick to the appropriate version of LabVIEW:


EV3 Error

Problem:

NXT/EV3 firmware version does not match computer driver version.

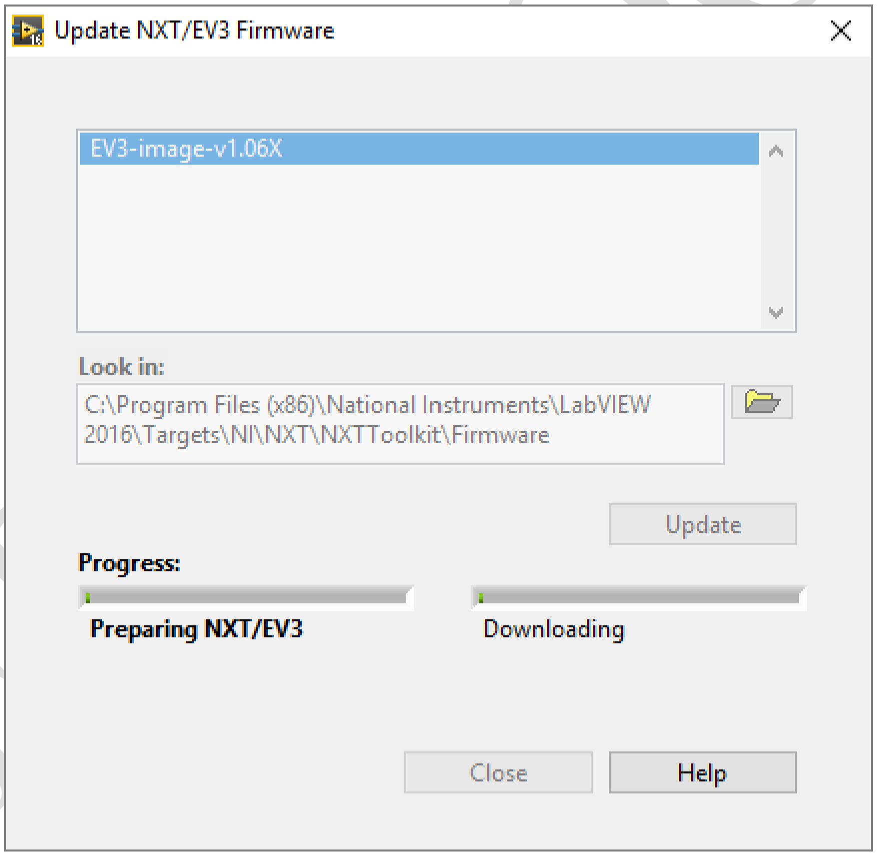
Suggestions:

- Update NXT/EV3 firmware to the latest version.

- Update your computer's NXT driver to the latest version.

Figure 7 Updating the brick

After selecting this option, in the Update NXT/EV3 Firmware window, click the *Update* button and wait for the update process to complete:


Update NXT/EV3 Firmware

EV3-image-v1.06X

Look in:

C:\Program Files (x86)\National Instruments\LabVIEW 2016\Targets\NI\NXT\NXTToolkit\Firmware

Progress:

Preparing NXT/EV3

Downloading

Figure 8 Brick’s update process

After displaying the message “*Successfully Downloaded Firmware!*”, Click the *Close* button.

Another part of the Robot Project Center window is Schematic. By selecting *Open Schematic Editor*, you will enter the schematic editor environment:

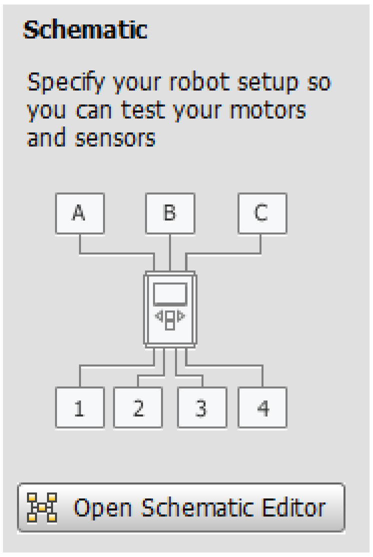

Schematic
Specify your robot setup so you can test your motors and sensors
A
B
C
1
2
3
4
Open Schematic Editor

Figure 10 Schematic of a brick in the project center panel

As you can see on the left side of this environment, general information of the brick, including battery percentage, connection type, free space, and the firmware version are displayed. Also, if you are connected to several bricks at the same time, you can identify the desired brick by clicking the *Go* button in front of the *Play Tone* to play a beep sound from the brick.

In the right space of this window, you can see a schematic of the brick and its ports. By clicking on the small arrow next to each port, you can select the motor or sensor connected to that port to get help from this layout in the future and while designing the program process. Moreover, to perform various tests on motors or sensors connected to the brick, a panel will be displayed on the left by selecting any of them, which provides you with several options.

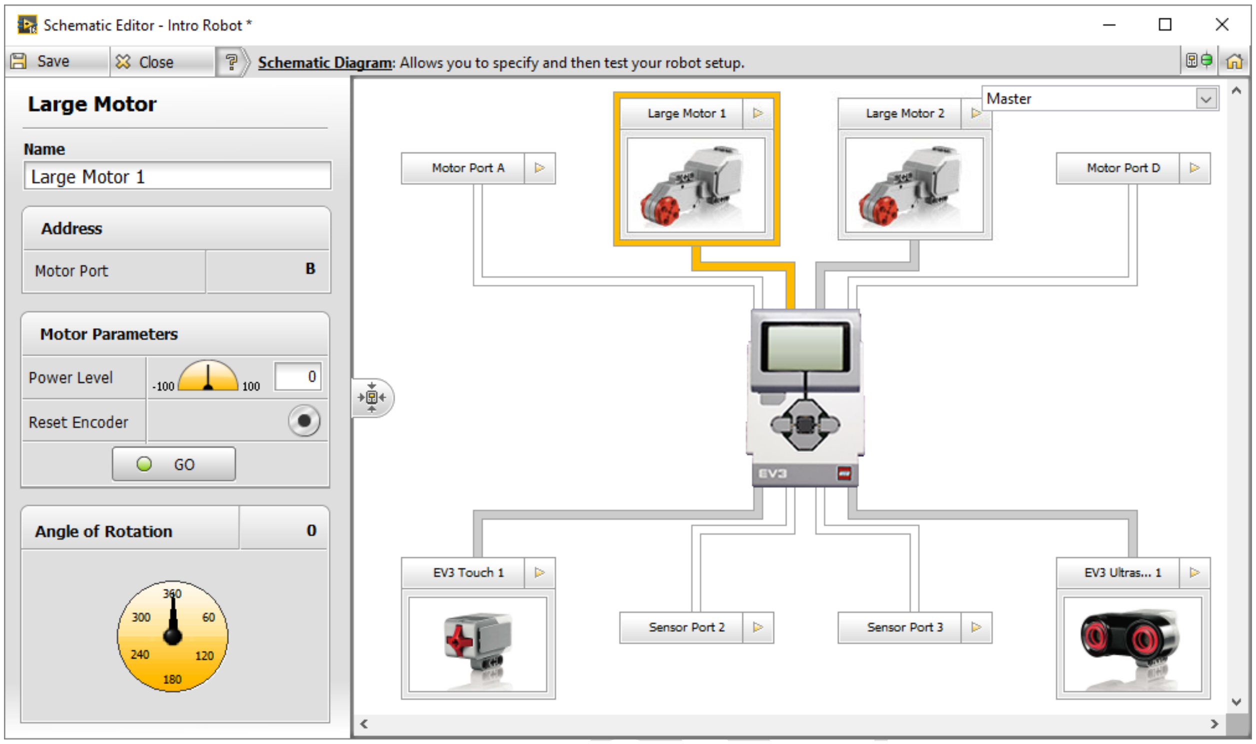


Figure 11 Shows the motors and sensors connected to the brick

In the *Robot Files* section of *the Robot Project Center window*, VI files and other projects are displayed. At the bottom of the Robot Project Center window, click on *NXT/EV3 Tools & Apps* to get the following applications and tools:

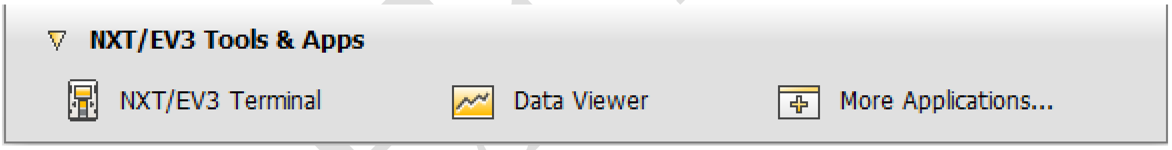


Figure 12 Other tools included in the Project Center window

The *NXT/EV3 Terminal* tool opens a window where it provides you with the brick's information. You can also use this tool to rename the brick, send, receive, and edit files from the brick. The *Data Viewer* tool displays the information received from the sensors connected to the brick. The *More Applications* option gives you extra add-ons which we will not discuss in this book.

Now that we have reviewed all the components of *Robot Project Center*, we are ready to start programming for the brick. To build a VI and start programming, click the *New VI* button in the Robot *Files* section:

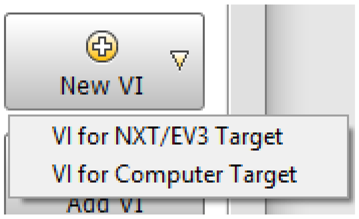


Figure 13 Creating a program for the intended destination

Clicking this button will bring up two options. In fact, you can specify that the program you want to design will be run independently on the brick or a computer connected to the brick.

We use the first option, *VI for NXT/EV3 Target*, where the designed program does not require a computer, and the brick independently controls the motors and sensors. In this case, the destination is the brick.

But the second option, *VI for Computer Target*, is a program designed to be run by a computer, which receives information from EV3 sensors or sends commands to EV3 motors if needed. This type of programs can be used in various cases, for example, when it is necessary to display the data on the computer or in cases where the user changes variables in the Front Panel space. In this case, the destination of the VI is a computer.

By selecting any of the options, after selecting the name, a new VI is created, and two windows, *Front Panel* and *Block Diagram*, are displayed:

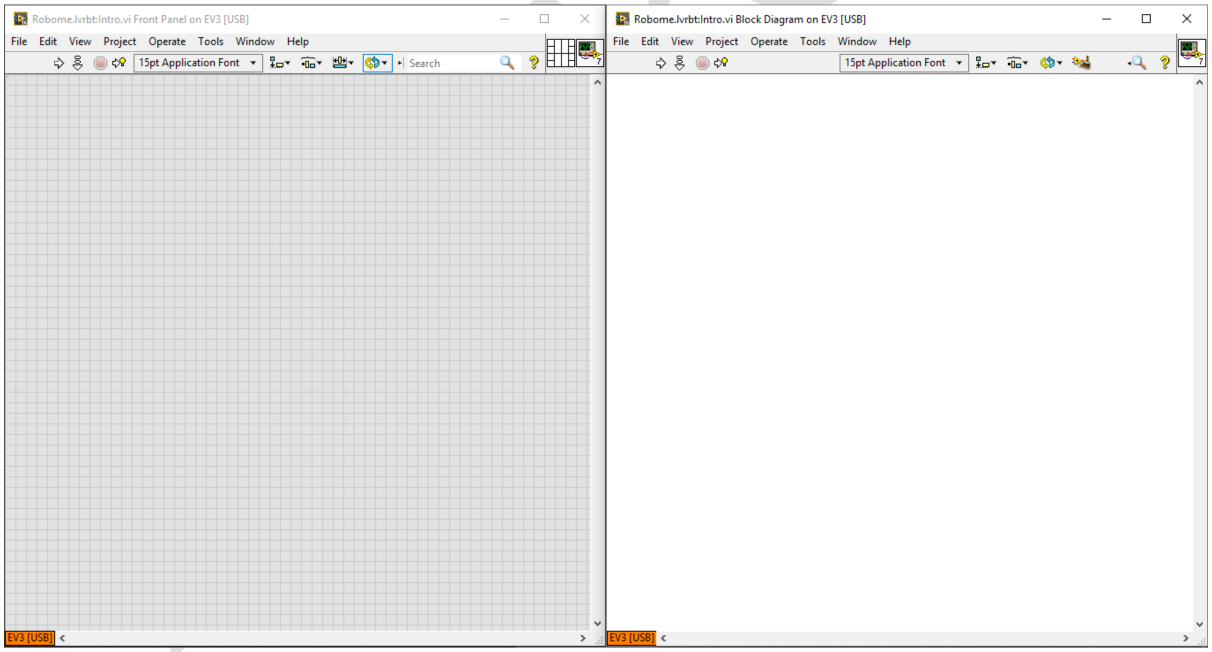

Robome.lvrbt:Intro.vi Front Panel on EV3 [USB]
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
EV3 [USB]
Robome.lvrbt:Intro.vi Block Diagram on EV3 [USB]
File Edit View Project Operate Tools Window Help
15pt Application Font
EV3 [USB]

Figure 14 Displays the Front Panel and Block Diagram

There is a field at the bottom left of each window that specifies destination VI. By right-clicking on this field, you can change the destination of the VI:

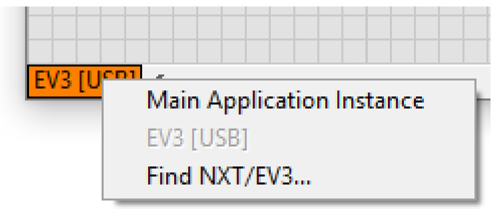


Figure 15 Destination selection of VI

The *Main Application Instance* option changes destination VI to the computer. If the software has not yet detected your brick, you can click on *Find NXT/EV3* to find your brick.

One of the significant differences between a VI with a brick destination and a computer destination is the VI play or upload buttons. If the destination is brick, the VI toolbar buttons are displayed as *Run* and *Deploy*. However, if the destination is a computer, the *Upload* button is changed to the *Run Continuously* button.

In fact, if the destination is a brick, by clicking the *Run* button, the program will run only once and will not even be saved on the brick's memory, but by clicking the deploy button, the program will be moved to the brick, stored in its memory, and then can be independently run by the brick.

Another difference between the two types of programs is the toolbars and functions they provide for the user by default. If the destination is brick, the palettes display the blocks associated with the MINDSTORMS group by default. However, if the destination is a computer, the primary program palettes are displayed. Nevertheless, in both cases, all groups can be accessed by selecting the group name in both palettes.

In the next section, we will discuss programming for and working with motors and sensors.

MINDSTORMS EV3 COMPONENTS

To start programming the EV3, we first review the electronic components, including the motors and sensors in the MINDSTORMS EV3 package:

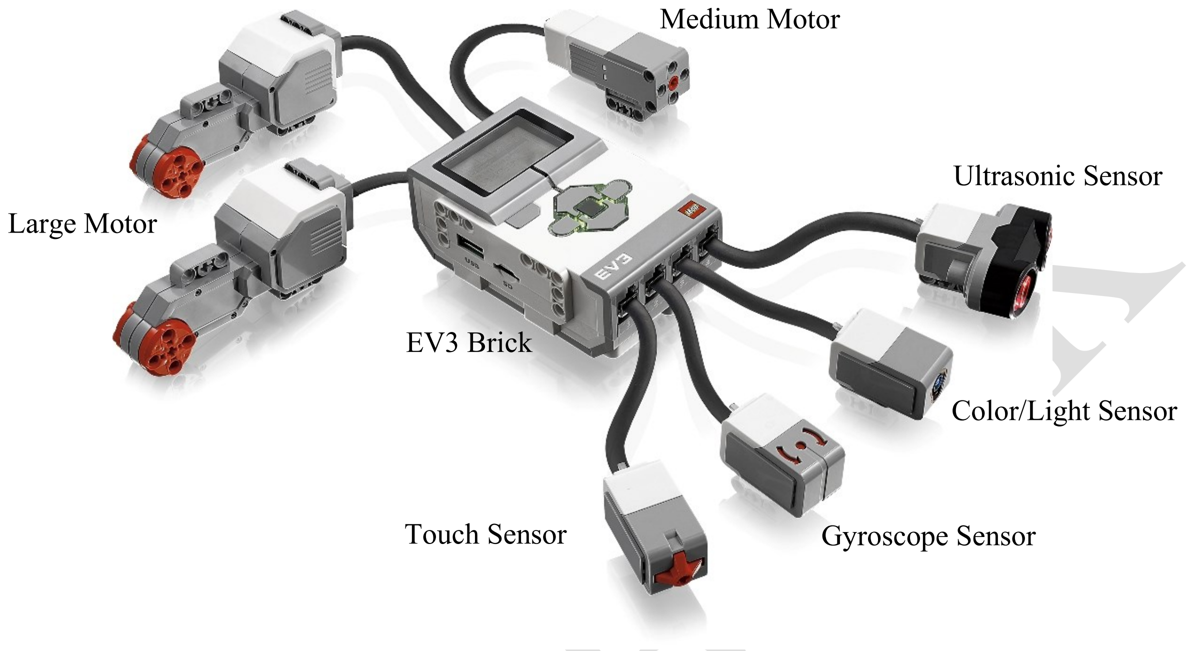


Figure 16 Different components of the MINDSTORM EV3 package

EV3 Brick

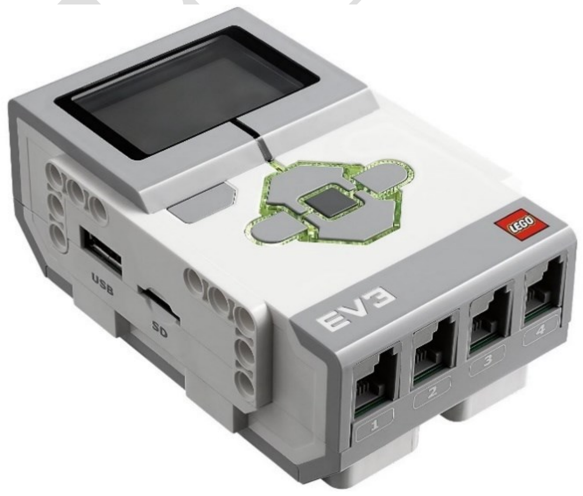


Figure 17 EV3 brick

Brick is the brain of MINDSTROMS robots. Written programs are executed in the brick. The information extracted from the sensors are transmitted to the brick and the desired commands are sent to the motors by the brick.

There are 5 buttons and 8 input ports on each brick. The following figure shows the different parts of the brick:

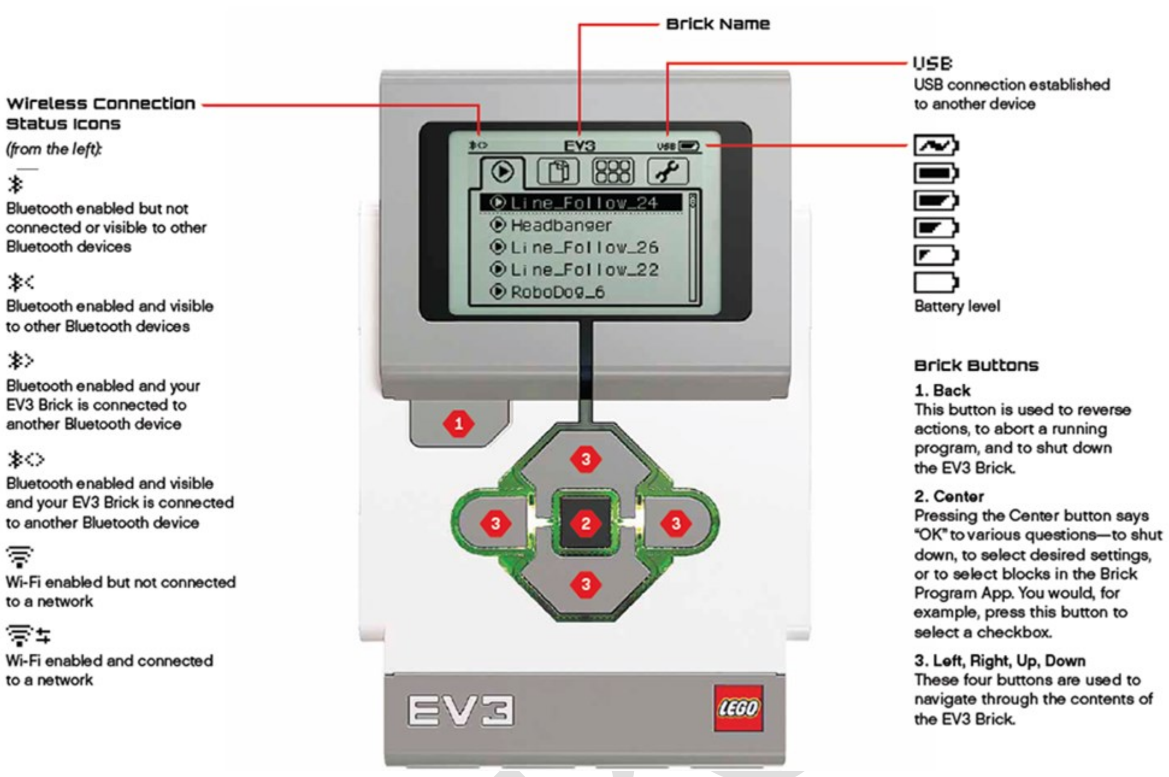

Brick Name
USB
USB connection established to another device
Wireless Connection Status Icons
(from the left):
Bluetooth enabled but not connected or visible to other Bluetooth devices
Bluetooth enabled and visible to other Bluetooth devices
Bluetooth enabled and your EV3 Brick is connected to another Bluetooth device
Bluetooth enabled and visible and your EV3 Brick is connected to another Bluetooth device
Wi-Fi enabled but not connected to a network
Wi-Fi enabled and connected to a network
Battery level
Brick Buttons
1. Back
This button is used to reverse actions, to abort a running program, and to shut down the EV3 Brick.
2. Center
Pressing the Center button says "OK" to various questions—to shut down, to select desired settings, or to select blocks in the Brick Program App. You would, for example, press this button to select a checkbox.
3. Left, Right, Up, Down
These four buttons are used to navigate through the contents of the EV3 Brick.
EV3
Line_Follow_24
Headbanger
Line_Follow_26
Line_Follow_22
RoboDog_6
EV3

Figure 18 Different sections of a brick

1. ***Back* button**: This button is used to return to the previous menu, exit the running program, or turn off the brick. To turn off the brick, you need to return to the main menu of the brick, and after pressing the return button, select the option.
2. ***Center* button**: This button can be used to select the desired option or run programs.
3. ***Directional* button**: These keys are four main directions (up, down, left and right). These keys can be used to navigate through the brick menus as well as input applications.

As you can see, after turning on the brick (i.e., holding down the center button), four tabs and the main menu are displayed. Use the left and right buttons to navigate through the tabs.

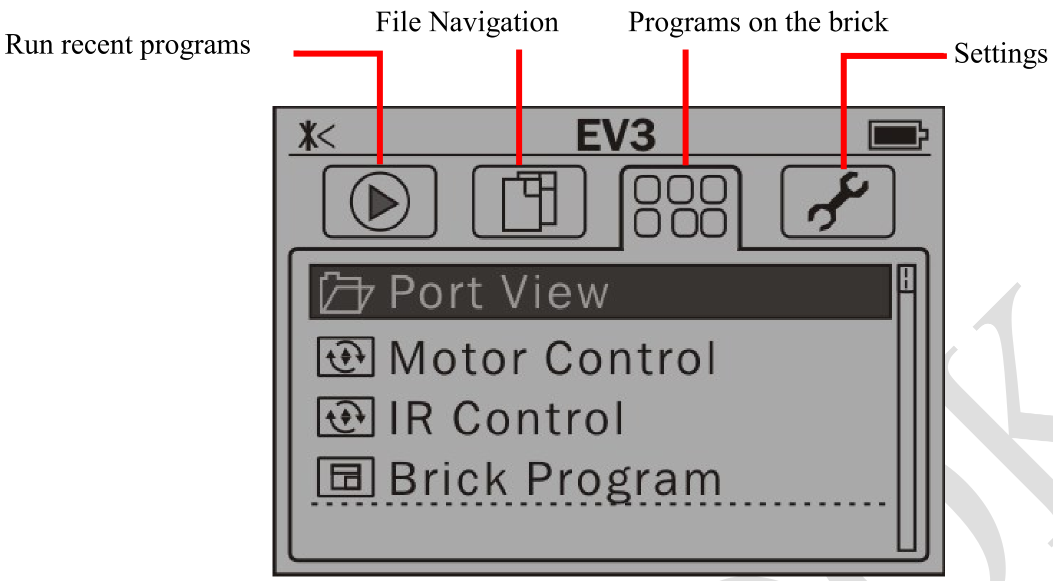


Figure 19 Brick's tabs

1. **Run Recent menu**: The latest programs run by the block are displayed in this menu. New applications uploaded within the block are also displayed.
2. **File Navigation menu**: Using this menu, you can access the files in the brick's memory.
3. **Programs menu**: You can access the various applications on the brick in this menu. In this section, we will briefly describe each of the programs:
    - **Port View**: This program can be used to view the sensor or motor connected to each port. You can see the sensor or motor connected to each port with the arrow buttons in this case. This section can be used to troubleshoot or calibrate sensors and motors.
    - **Motor Control**: This program can be used to control motors manually. In this case, by pressing the center button, the pair of motors connected to the brick can be specified, and with the directional buttons, each motor can be moved in two directions. This program can be used to check the motors and their performance.
    - **IR Control**: This program can be used to set the infrared remote-control options. In this book, we will not describe these options because the remote-control is not included in the core sets of EV3 robots.
    - **Brick Program**: This program can be used to program a robot. This program can be used for simple applications, but it cannot be used for complex applications due to its simple environment and limitations.
    - **Brick Data log**: This program can be used to store and display data sent from sensors.
4. **Settings**: In this menu, you can access the brick's settings such as volume, timer, Bluetooth, Wi-Fi, and other options.

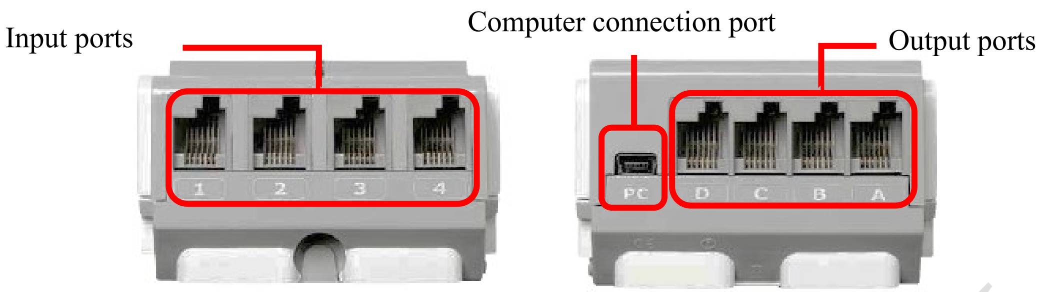


Figure 20 Brick's input and output ports

There are 8 input and output ports on each brick. There are 4 ports at the top with the letters A, B, C and D, which are output ports for connecting the motor to the brick. The other 4 ports are located at the bottom of the block and are labeled 1, 2, 3 and 4. These ports are inputs and used to connect the sensors to the brick. There is a Mini-USB port for connecting the brick to a computer with a USB cable. Wireless connections can also be used to connect the brick to the computer.

Motors

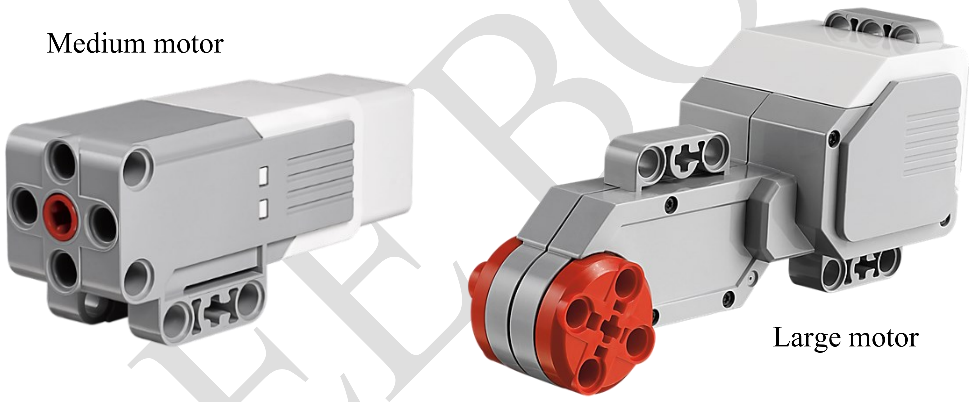


Figure 21 EV3 motors

There are two types of engines in the EV3 series, known as large motor and medium motor. These two types of motors are servo motors that can measure rotation, speed, and power of the motor. Servo motors use encoders to measure rotation and its derivatives (velocity and acceleration). The accuracy of measuring the rotation of this motor is 1 degree. The sensors in these motors can also be used to extract information in experiments.

The difference between a large motor and a medium motor is in their power, speed and spatial configuration. Large motors are used for more power but lower speeds, as their maximum speed is 160-170 rpm, and their maximum torque is 20 N.cm in motion and 40 N.cm at rest. On the other hand, for the medium motors, their maximum speed is 240-250 rpm, torque is 8 N.cm in motion and12 N.cm at rest.

In fact, these motors are the actuators of our robot which can be used as wheel drivers, grippers, belt actuators, and so on. In the following, we will discuss the basics of working with motors and some examples of their applications.

Commanding Motors

There are several ways to command a motor in LabVIEW. The simplest type of command is to specify motor power/speed. Another way is to determine the amount of motor rotation. There are other ways to move a motor, which we will discuss later.

Move Motors Block

Connect a large motor to port A of the brick to start. Then, connect the brick to your computer with a USB cable. Create a new project in LabVIEW. In the block diagram window, right-click and navigate to the *I/O* category in the *MINDSTORM Robotics* section, and from there, select the *Move Motors* block and place it in the block diagram space:

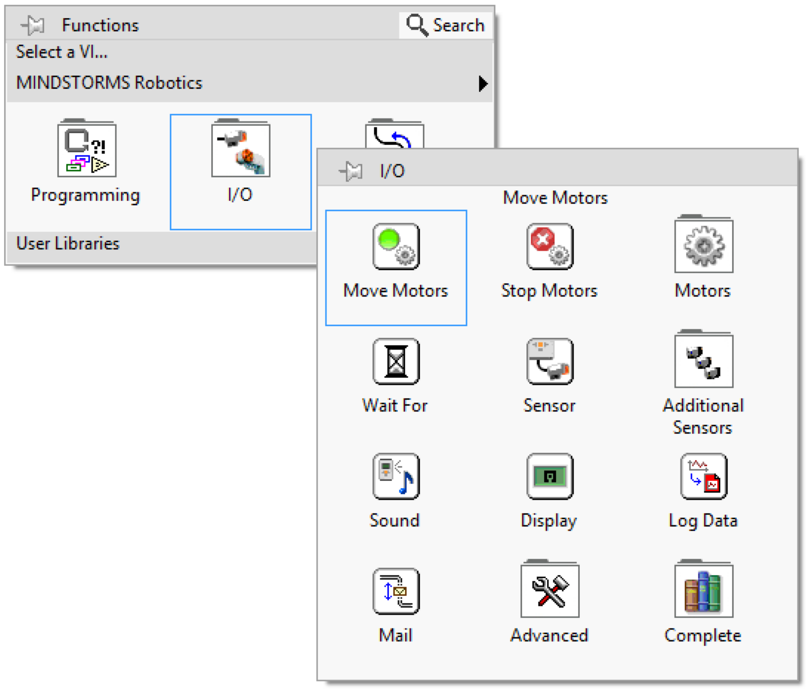


Figure 22 Selecting the Move Motors block in the block diagram window

After placing the Move Motors block, this block will appear:

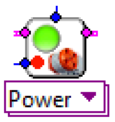


Figure 23 Move Motors block

Right-click on the top node of the block (Motors) and select *Create Constant* to specify the port to which the motor is connected. Here we want to apply this command to the motor connected to port A. So, after selecting *Create Constant* from the drop-down menu, select Lego Port A:

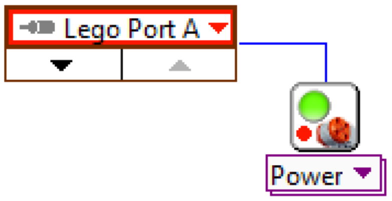


Figure 24 Connecting the desired motor to the Motors node

Note that if you do not specify a motor in this section, the program will apply this command to all motors connected to the brick.

In the next step, it is time to determine the magnitude of power/speed. As shown in the block, you can specify the type of command from the drop-down menu at the bottom of the block. This block has two types of commands called *Constant Power* and *Constant Speed*. Using the first option, we specify the motor's power, which will change its rotational speed according to the amount of load on the motor. But in the second option, the motor speed is specified, in this case the motor adjusts its power to meet and maintain the commanded speed.

To determine the power/speed of the motor in this block, we must connect a fixed number block to its input node (known in the program as *Power/Speed 1*), which indicates the power/speed of rotation of the motor. For simplicity, you can right-click on its input node and select *Create* → *Constant*.

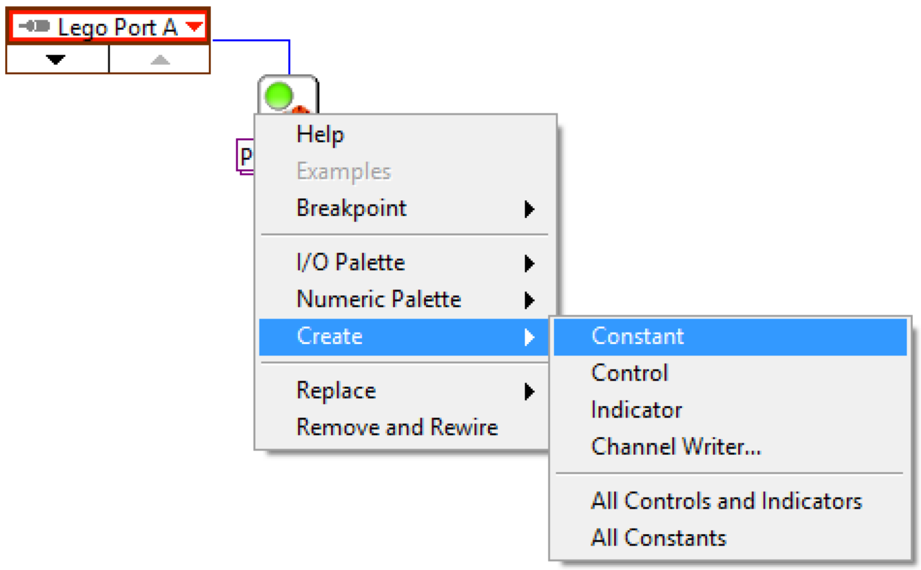


Figure 25 Creating a fixed number on the Power 1 node

By selecting a fixed number block as shown below, a fixed number of 75 is created by default:

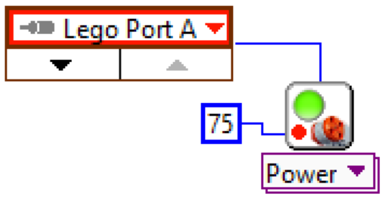


Figure 26 Assigning a constant value of 75 for the power of the motor

The number given as power/speed to this block must be between -100 and 100. Negative numbers mean rotation in the opposite direction. If numbers greater than 100 or less than -100 are given to the block, they are considered as 100 and -100. In fact, this number indicates the percentage of power/speed of the motor, which is 75% or $^3/_4$, by default. Now by uploading and running the program, the motor connected to port A will start with 75% of power.

By running this program, the motors are activated in an instant, but they stop quickly and the program ends. This is because we only executed the command once, and after executing this command, the program is over. To move the motors consistently, this command must be sent to the robot continuously. So, it is necessary to execute this command all the time. To do this, an infinite While loop can be used. As explained in the previous chapters, by creating an infinite *While* loop, one or more blocks can be executed infinitely and permanently. So, after adding an infinite loop, we will have:

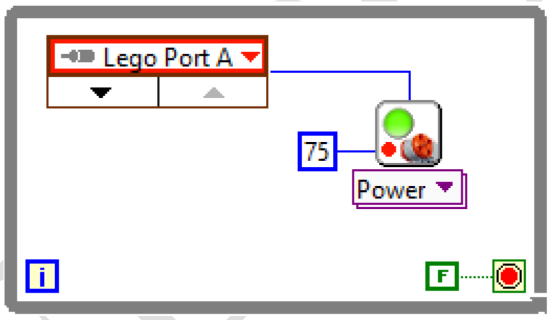


Figure 27 Assigning the value 75 as the motor power connected to port A

We can use only one *Move Motors* block to command several motors simultaneously. This is possible by clicking on the arrow below Lego Port A and selecting another port from the newly created drop-down menu. For example, if one motor is on port A and the other one is on port C:

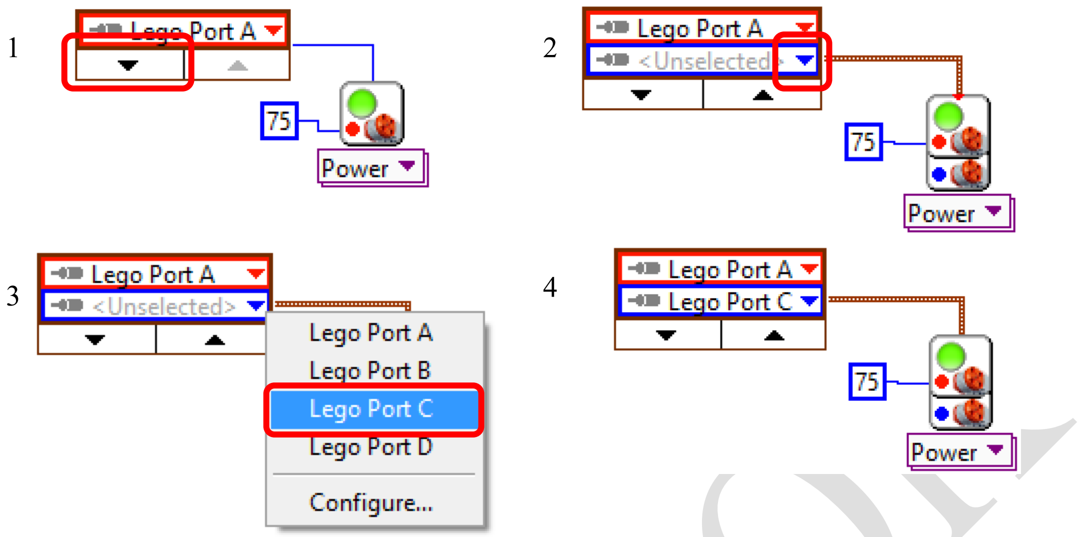


Figure 28 Method of adding a motor to the Motors node

In this case, we will be able to determine the power/speed of the two motors separately. To do the same as before, we connect a fixed number block to the *Power/Speed 2* node:

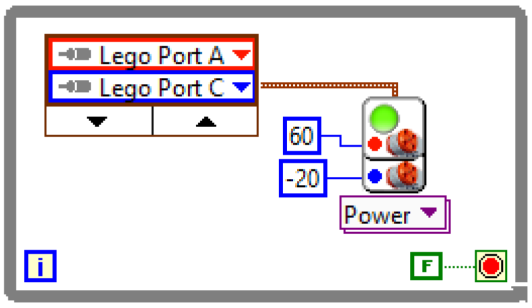


Figure 29 Movement of two motors in an infinite loop

By uploading and running this program, the motor connected to port A with 60% power and the motor connected to port C with 20% power will move in the opposite direction.

Command using Fixed Distance block

This block is used to create a fixed distance movement. This block is used in cases where we already know the amount of robot movement or the amount of engine rotation, or we can calculate. To add this block to your application, go to the MINDSTORM Robotics section and select I / O → Motors → Fixed Distance from the section:

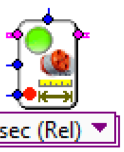


Figure 30 Fixed Distance block

First, in this block, like the Move Motors block, we specify the port (s) connected to the desired motor (s):

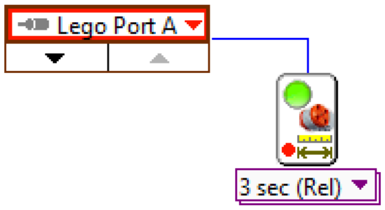


Figure 31 Connecting the Motors node to the desired port

In the next step, we must specify the amount of rotation and the speed of the motor. For the engines in the EV3 set, we will be able to move them with an accuracy of 1 degree. So, the number that goes to the *Distance 1* node indicates the amount of motor rotation in degrees. The rotation value is 360 degrees by default. By creating a fixed number block on this node, we can specify the amount of motor rotation:

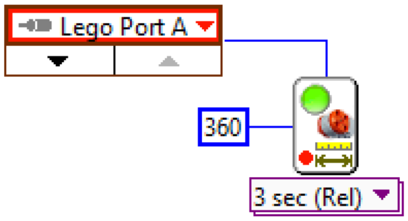


Figure 32 Assigning the value of the motor rotation angle

The Speed 1 node is used to determine the speed of the motor, which is determined like the *Move Motors* block. By applying for a fixed number between -100 and 100 to the Speed 1 node, the engine rotation speed can be defined:

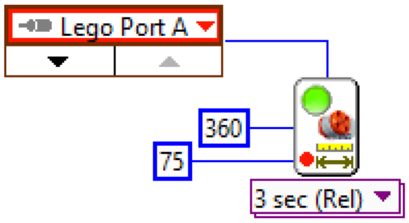


Figure 33 Assigning the amount of motor power

Another difference between this block and the Move Motors block is in the drop-down menu at the bottom of the block. In this block, by opening the drop-down menu, two options, *Relative* and *Absolute*, appear.

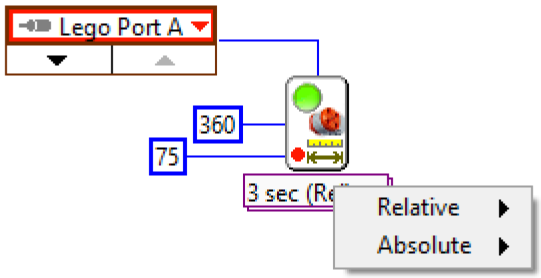


Figure 34 Specifying the relative or absolute motion of the motor in the drop-down menu at the bottom of the block

In relative motion, the motor rotates a certain amount from its current position. However, in absolute motion, the motor moves the motor to the said position according to the value read by its encoder. For example, suppose we specify a value of 200 in the Distance 1 node and the read value of the motor encoder is 100 in the initial state, in the relative motion mode. In that case, the motor eventually reaches the 300 positions, but in the absolute motion mode, the motor moves to the same position 200... In fact, it can be said that in relative and absolute terms, the definition of a block from the *Distance 1* node changes.

It should be noted that for the relative movement of the motor, the amount of displacement and speed must be both positive and negative for the block to function correctly.

Next, after determining the type of relative or absolute motion, we must specify the amount of program pause to execute this command. By default, this block partially moves the engine, giving the command 3 seconds to run. If the engine speed is low or the amount of rotation is high, more than 3 seconds is required for the specified speed, so this time should be increased. If the specified period value is performed earlier than the specified time, the program will automatically run and continue.

Lego Steering Block

One of the most important commands that can be given to motors is to follow curved and circular paths. You can use Move Motors or Fixed Distance blocks to do this, but for more simplicity and speed in robot programming, the *LEGO Steering* block is designed. This block is available in the Functions → MINDSTORM Robotics → I / O → Motors section called LEGO Steering:

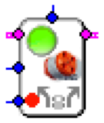


Figure 35 LEGO Steering block

In this block, by specifying two motors in the Motors node, the command operation can be performed with different motor powers. In the Power 1 node, select the value of the motor power, which is a number between -100 and 100, and in the Steering 1 node, select a number between -100 and 100, in which case the first and second motor power (respectively, in the Motors node Are specified) is a percentage of the power specified in the Power 1 node. This percentage is determined according to the following chart:

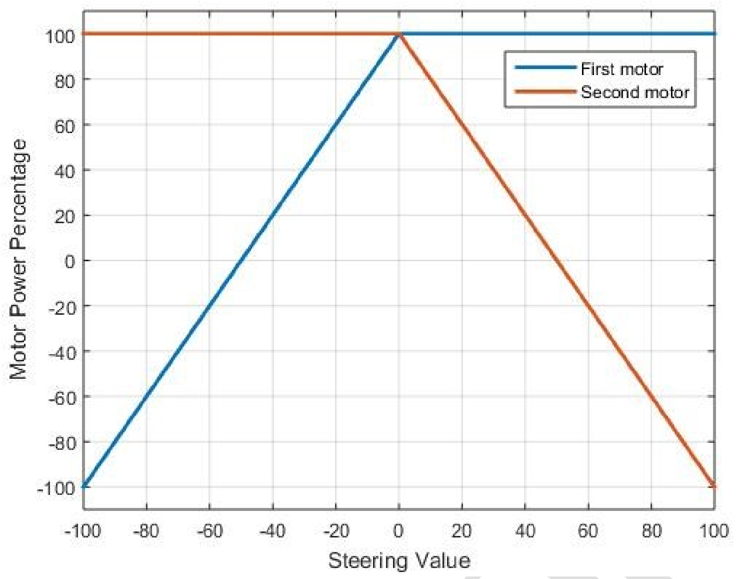


Figure 36 Diagram of the power of each motor relative to the input value of the steering node

In fact, in positive values for the Steering 1 node, the first motor has a power equal to the value given in the Power 1 node, and the second motor has a percentage of this power. For negative values, this is the case for two photo engines. In the next section, we will discuss in detail the application of this block in robot programming.

Stop Motors Block

The two blocks introduced in the previous sections are used to command and move the motors, one of which can be used depending on the type of movement and the purpose of that movement. But the point to note is that when we command the motors to move, we must also issue a stop command to the motors after the movement is completed. Because when we use several motors to move the motor, if the Stop Motors block is not used, the commands will overlap and usually, the movement predetermined by the robot will not be executed.

The Stop Motors block is accessible from Functions → MINDSTORM Robotics → I/O → *Stop Motors*:

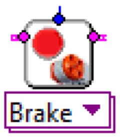


Figure 37 Stop Motors block

To use this block, all you must do is specify the port (s) connected to the motor (s) and then connect the NXT / EV3 input and output nodes to the previous and following blocks, respectively:

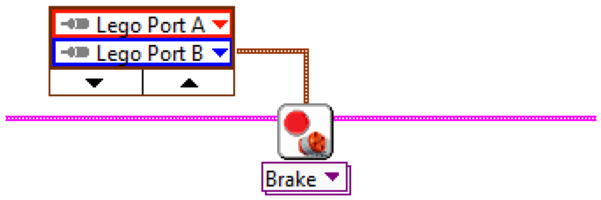


Figure 38 Connecting the nodes of the Stop Motors block

Note that this block will issue a stop command to all motors connected to the break if we do not specify the Motors node.

Another option for this block is available from the drop-down menu at the bottom of the block. This block has two modes, Brake and Coast. In Brake mode, the motors stop quickly, but in Coast mode, the motors slow down slowly and then stop, which can be selected depending on our goal.

Motor Applications

As mentioned, motors can be used for different parts of a robot, including grippers, wheels, arms, launchers, belt actuators, etc. In fact, our actuators in the EV3 series are DC motors, but in the industry, different actuators are used besides DC motors, such as hydraulic and pneumatic cylinders, each of which has its own characteristics.

One of the simplest and most widely used application of motors in robots is to drive a wheel. In this case, as the motor rotates, the wheel also rotates and causes the robot to move on the ground.

Build the basic Educator robot according to the instructions given in the appendix.

In this robot, we have two wheels that are connected to two motors. One of the motors is connected to port B and the other to port C of the brick. By activating these two motors, different paths can be traveled by the robot. In the following, we will review the planning for some main routes.

Moving in a straight path

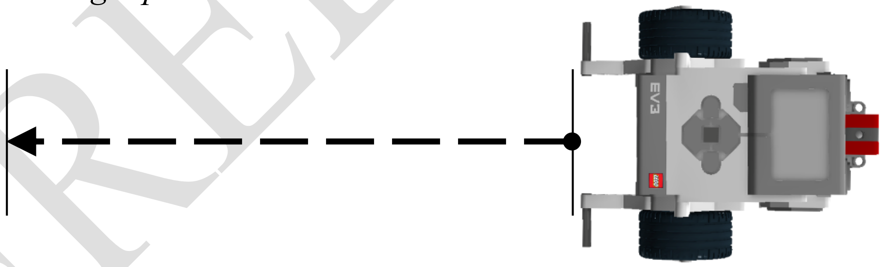


Figure 39 Moving the robot in a straight line

As we know, to move in a straight line, it is necessary for both wheels to rotate at the same speed, in which case all points of the robot have the same speed. Move Motors block can be used for this purpose. As described in the previous section, add the Move Motors block to the system and apply the command to the two motors B and C:

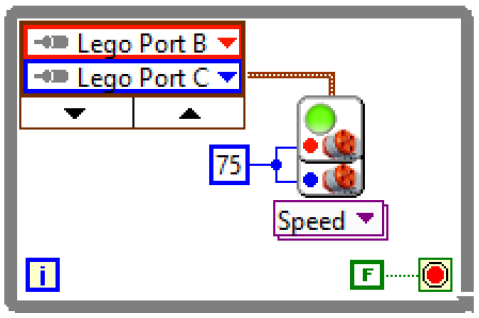


Figure 40 Continuous movement in a straight line with the Move Motors block

Note that in the Move Motors block, select the Constant Speed drop-down menu from the drop-down menu and set the speeds of both motors (Speed 1 and Speed 2) equal to 75. We have also placed it inside an infinite While loop to execute this command continuously.

By changing the fixed number 75 between -100 and 100, the movement speed on the straight path can be increased or decreased.

Try this program with LEGO Steering block as well. The LEGO Steering block is commonly used to rotate the robot and navigate curved paths, but direct movement can also be commanded.

Rotating about a wheel

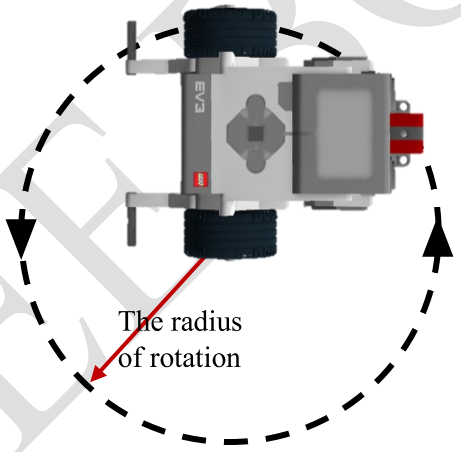


Figure 41 Rotation of a robot around a circle

If we want to rotate around wheel C, we need to apply the Move Motors command only to motor B.

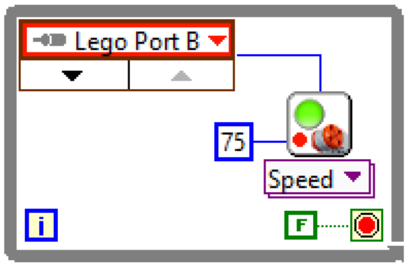


Figure 42 Continuous rotation around a circle with a Move Motors block

If we want to rotate around a wheel with the LEGO Steering block, according to the diagram of this block, we can program the exact same operation by setting Power 1 equal to 75 and Steering 1 equal to 50:

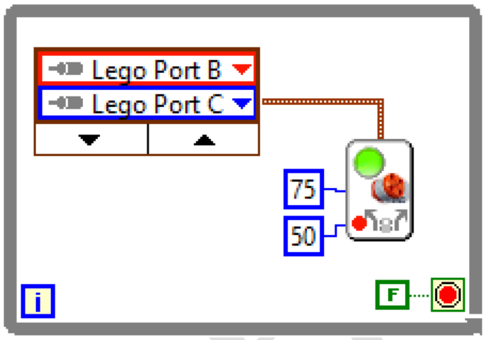


Figure 43 Continuous rotation around a circle with LEGO Steering block

The robot starts spinning around the C wheel by running the above program and repeats this movement indefinitely. But in many applications, it is necessary for our robot to rotate a certain amount and then follow another operation, so it is necessary to rotate around a wheel a certain amount. To do this, we use the Fixed Distance block to determine the change of the robot angle by rotating a wheel to a certain value. As you can see from the previous experiment, rotation around a wheel is like moving a robot on a circle with a radius of the distance between the two wheels and the center of the wheel, so to determine the angle of rotation, we need to calculate the length of the path we walk on this circle. On the other hand, with the Fixed Distance command, we control the angle of rotation of the engine and the wheel. If we consider the wheel's radius (which can be obtained by measurement) equal to r and take the angle of rotation of the motor θ °, the path traveled by this wheel is equal to rθ.

On the other hand, if we assume the angle of rotation of the robot α and the distance between the two wheels is d, the value of dα is equal to the length of the circular path:

By equating these two paths (the path traveled by the wheel and the path traveled on the circle), the value of θ can be obtained:

$$r\theta = d\alpha \rightarrow \theta = \frac{d\alpha}{r}$$

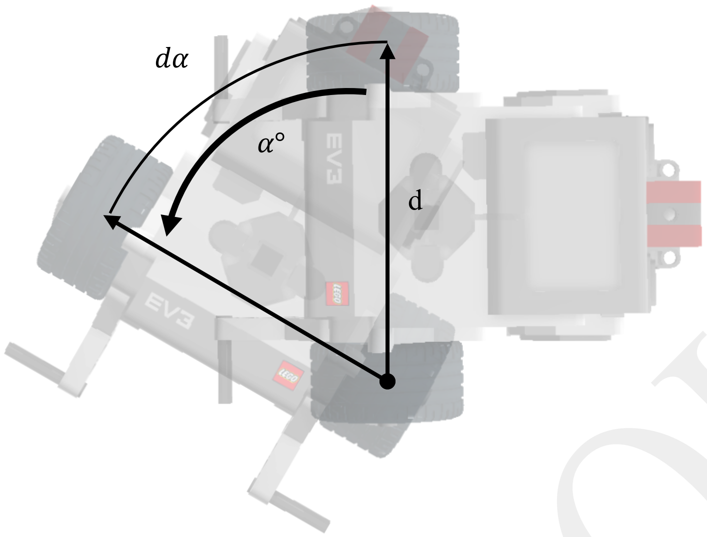


Figure 44 The path traveled by moving around a wheel

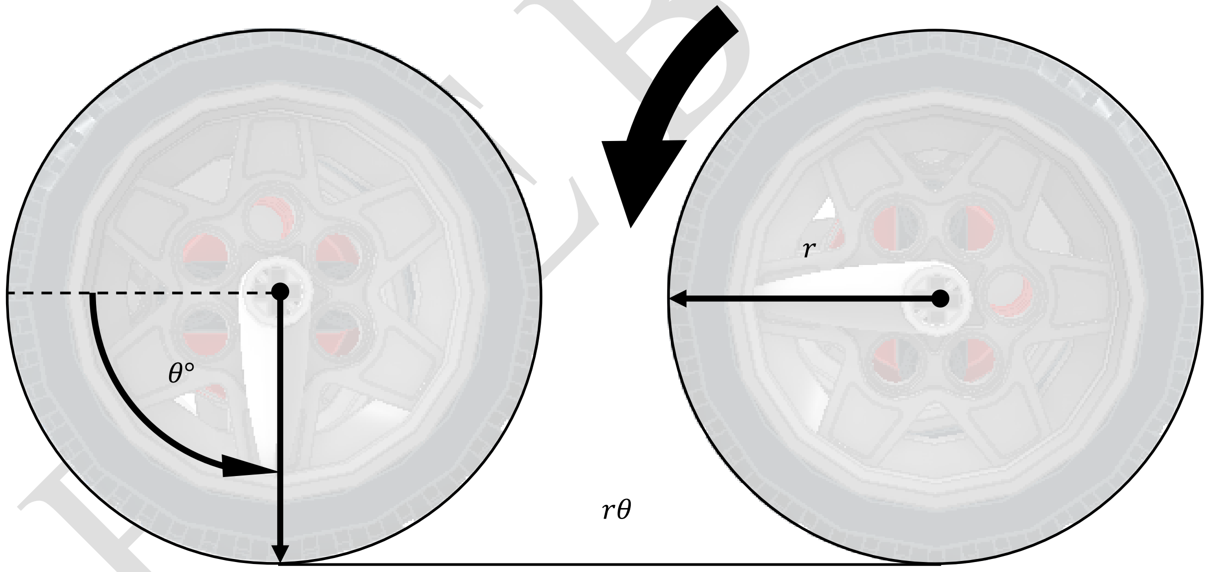


Figure 45 Path traveled by a wheel

In practice, to use this formula, the values of r and d can be obtained by measuring with a meter, ruler, or caliper, and we know the value of the robot's rotation angle (α) as we wish. To simplify the calculations, the said relation can be written in the program so that the wheel rotation angle can be calculated only by changing the rotation angle:

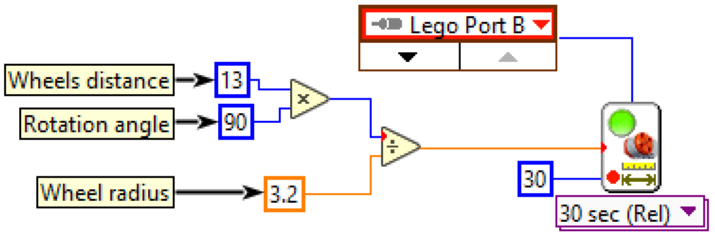


Figure 46 90-degree rotation around a wheel with a Fixed Distance block

Here our desired rotation value is 90 degrees. To change this angle, just set the Rotation angle to the desired angle. Note that from the drop-down menu at the bottom of the Fixed Distance block, we have selected the Relative → Wait Up To → 30-sec option, because first, we want the motor to move θ from its original position, and second, we give the motor a maximum time of 30 seconds to adjust its angle. Change. If the spin is completed in less than 30 seconds, the program continues and ends.

Compound Paths

In the previous two sections, we examined the movement in a straight line and a circular path and wrote a plan for each. In this part, which is a combination of the previous two parts, we want our robot to move in a path that includes a straight path and a circular path. For example, suppose our path is a square with rounded corners as follows:

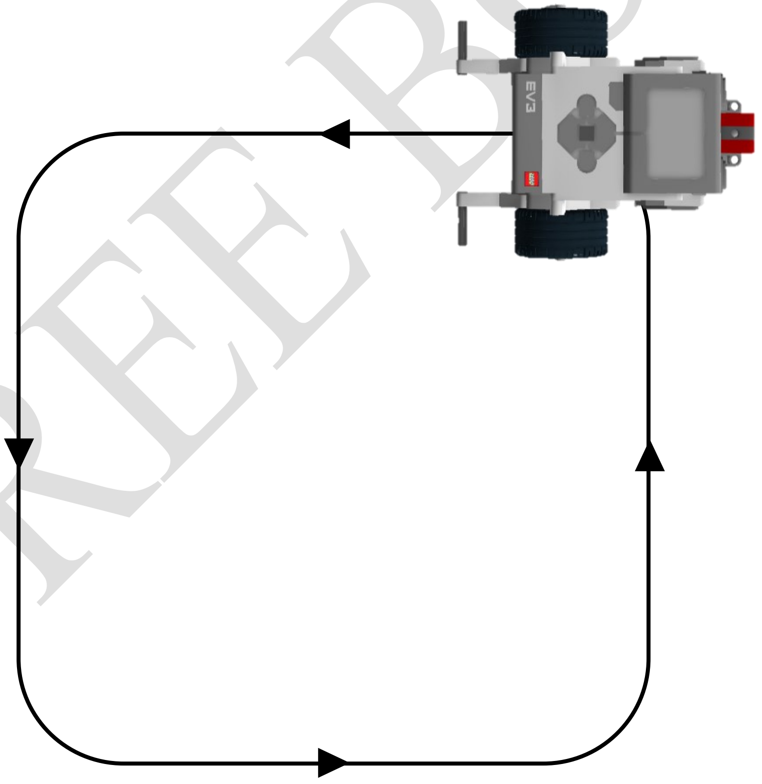


Figure 47 Robot movement in a compound path

As we know, this path includes four straight paths (square side) and four circular paths (square corners). Since all four sides and corners of this square are the same, it is enough to write the program for one side and corner of this path and repeat it infinitely. By combining the previous two parts, we will have:

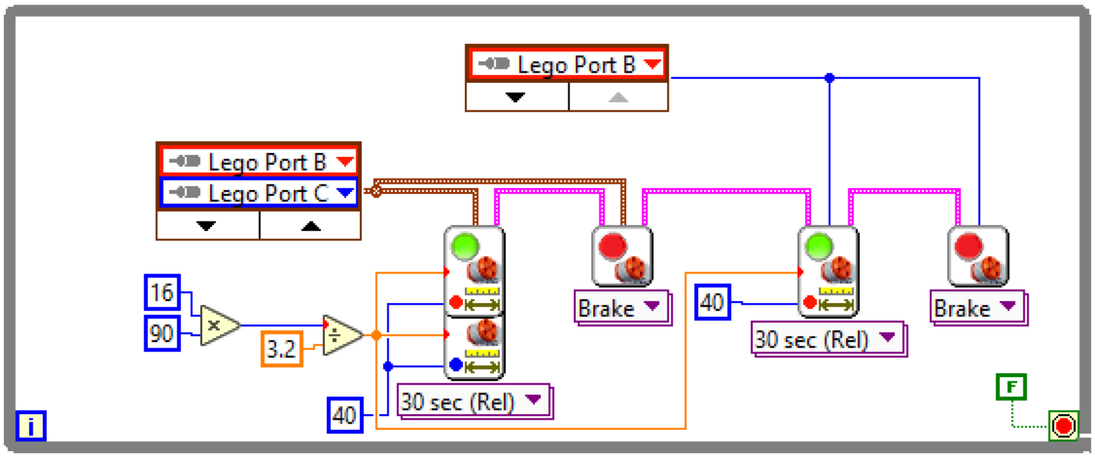


Figure 48 Continuous motion of the robot in a square path

This program consists of two blocks described in the previous two sections connected to the adjacent blocks by the NXT/EV3 output and input nodes. The Stop Motors block is used after each motor command to move the motors correctly and not interfere with each other.

TOUCH SENSOR

Figure 49 EV3 Touch sensor

The Touch Sensor is a simple button designed to detect the robot's collision with objects and obstacles or as a zero-in-one input. The sensor can detect when the button is pressed, released, tapped, and counts the number of times the single and multiple buttons are pressed. One of the most important applications of this sensor is for maze-solving robots.

Reading Touch Sensor

First, we add the Sensor block from the functions → MINDSTORMS Robotics → I/O section to our diagram block.

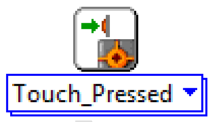


Figure 50 Touch sensor block

There are 4 modes or operating modes for this sensor. The output of the Sensor block in all 4 modes of this sensor will be a True/False logic variable.

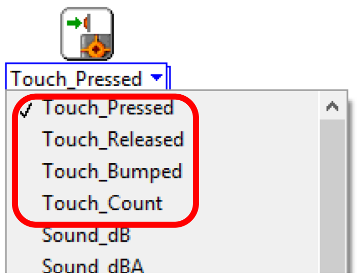


Figure 51 Four modes defined for the touch sensor

These 4 modes are *Pressed, Released, Bumped* and *Count,* respectively.

- In **Pressed** mode, when the button is pressed, it generates a True output node, otherwise False. By placing this block in the infinite loop and adding the Sound block, you can write the following program that creates a beep break by pressing the button.

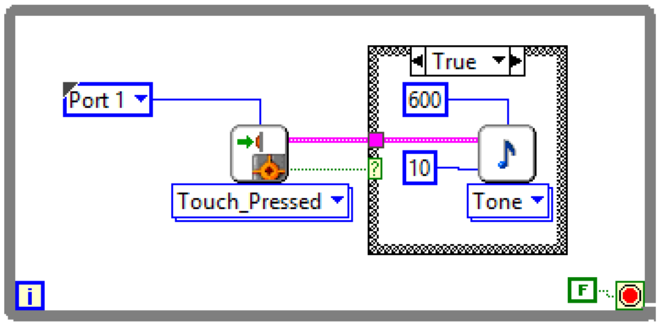


Figure 51 Creating a beep sound by pressing the sensor button

You can also write such a program with the *Wait For* block:

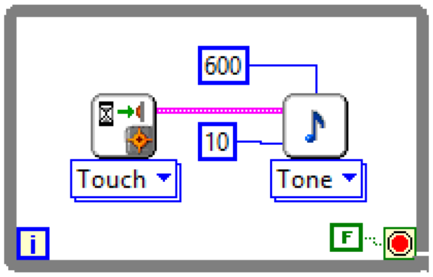


Figure 52 Beep when pressing the sensor button with the Wait For block

- **Released** mode is similar to Pressed mode, except that it works the other way around. That is, in Released mode, when the button is pressed, the output node is False, and in other cases, it is True.
  If we write a program similar to the one, we wrote for the previous fashion, then the break is constantly beeping unless the button is pressed:

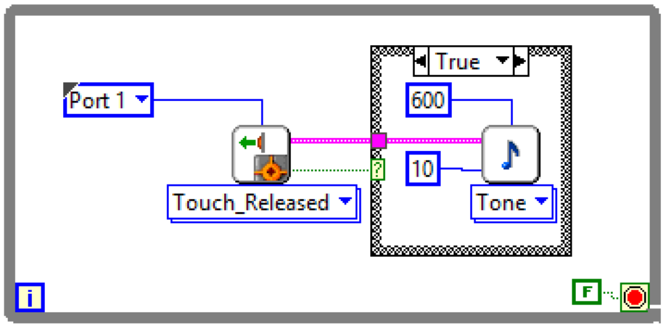


Figure 53 Mute the beep by pressing the sensor button

- In **Bumped** mode, the output node generates a True pulse each time you hit the button (at the moment of release). The difference between this case and the previous two cases is that in the previous cases, the output node stays True continuously, but it becomes True only for a moment, and in other cases, it is always False. In the following program, each time you hit the button, it produces a beep break.

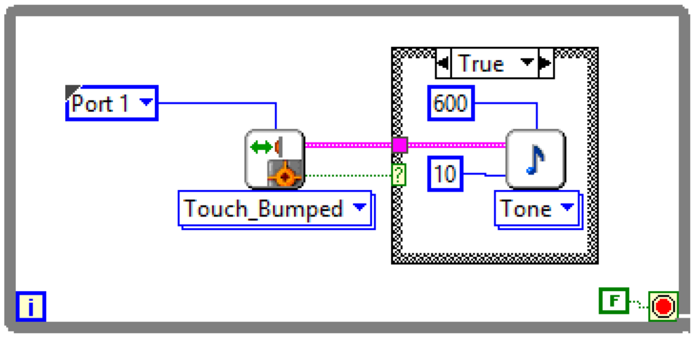


Figure 54 Generating a beep by tapping the sensor button

- The latest mode of this sensor is **Count**. In this mode, the output of the node is a number that the sensor counts each time the button is pressed. In fact, in this case, the number of times the button is counted. In the following program, each time the button is pressed, the number of times counted increases and is displayed on the brick's screen:

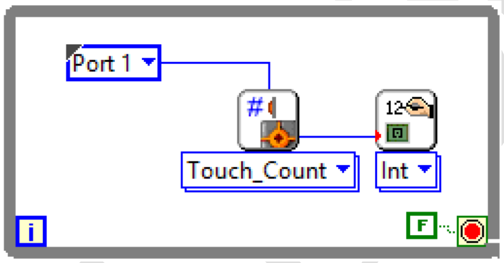


Figure 55 Displays the number of times the sensor button is pressed on the break screen

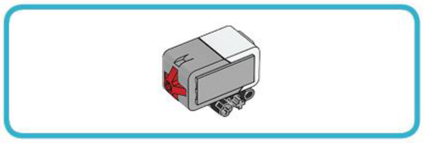
Table 1 Output node status in three modes of contact sensor

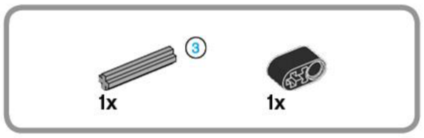
Position / Output	Pressed	Released	Bumped
Button is not touched	False	True	False
Button is touched	True	False	False
The moment the button is released	False	True	True

In a general summary and to clarify the difference between the three modes (*Pressed, Released and Bumped*), the following table shows the output of the block in different conditions:

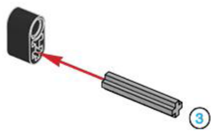
Touch Sensor Applications

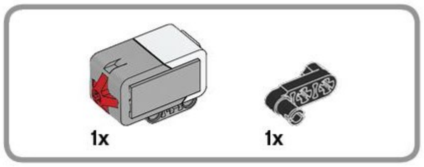
One of the simplest and most widely used uses of this sensor is to determine the robot's encounter with obstacles in front of it. For this purpose, we first connect the touch sensor to the EV3 Educator robot as follows:



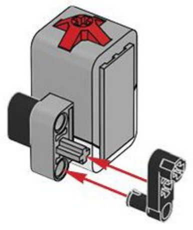


1

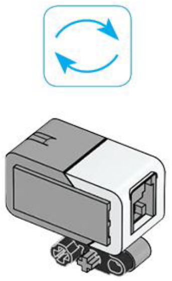




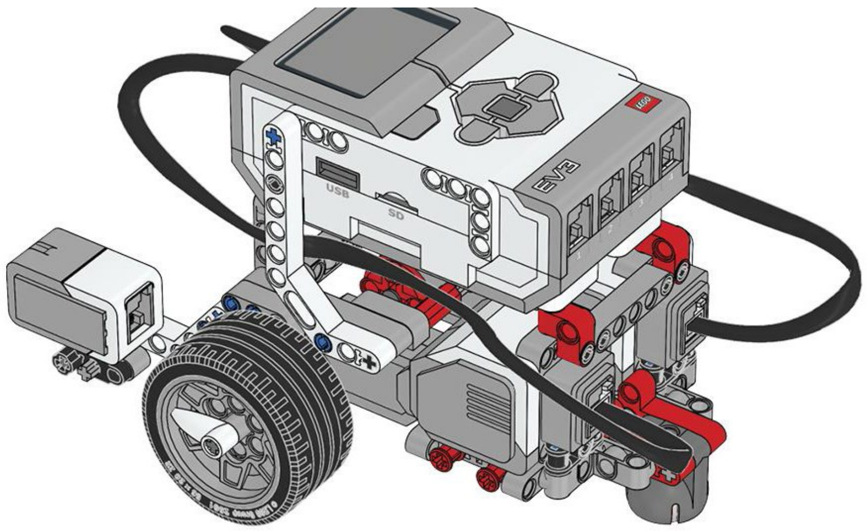
2

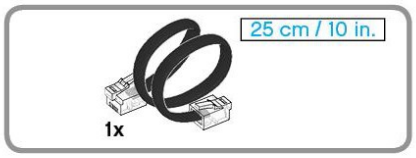


3

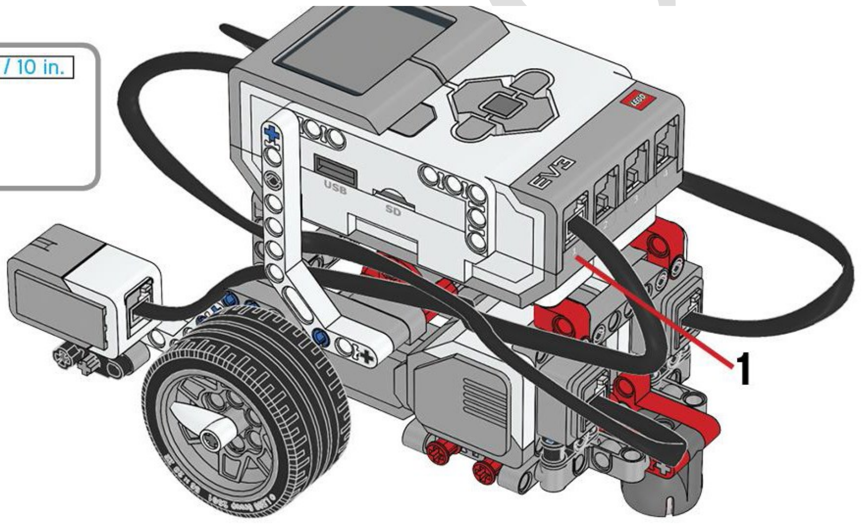


4

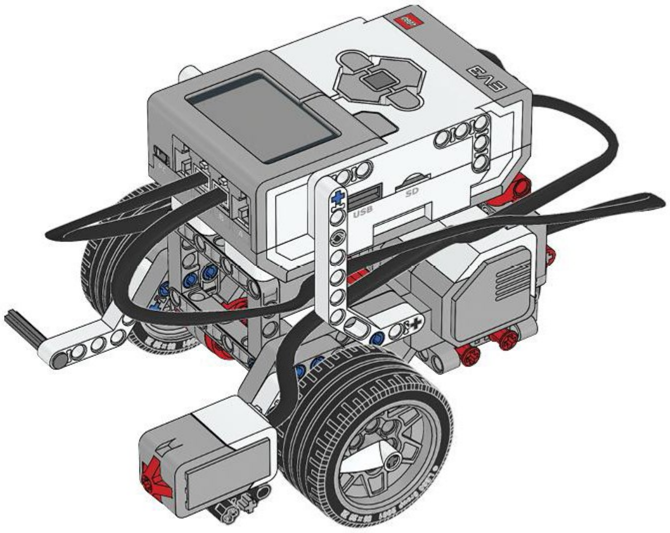




5



6



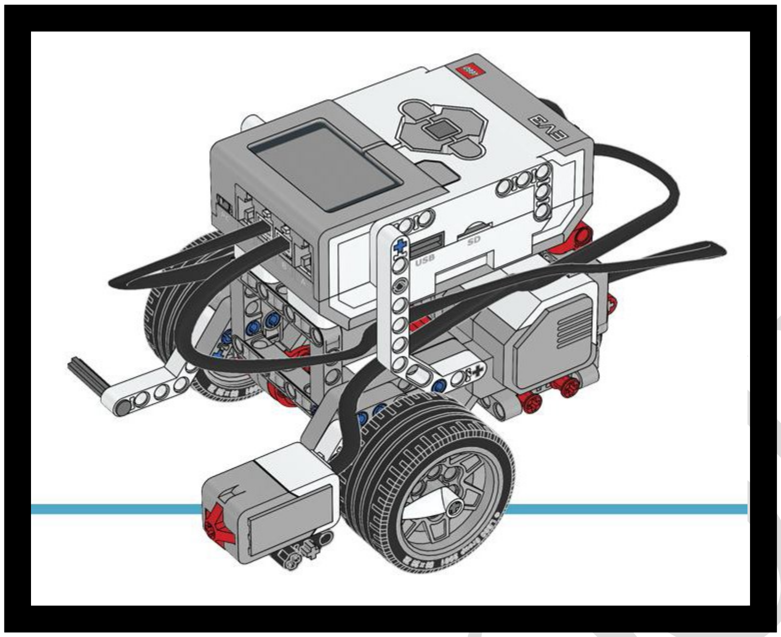


Figure 56 Installing the touch sensor on the Educator robot

Our goal is to program a robot that goes straight and spins when it encounters an obstacle and then continues its way again.

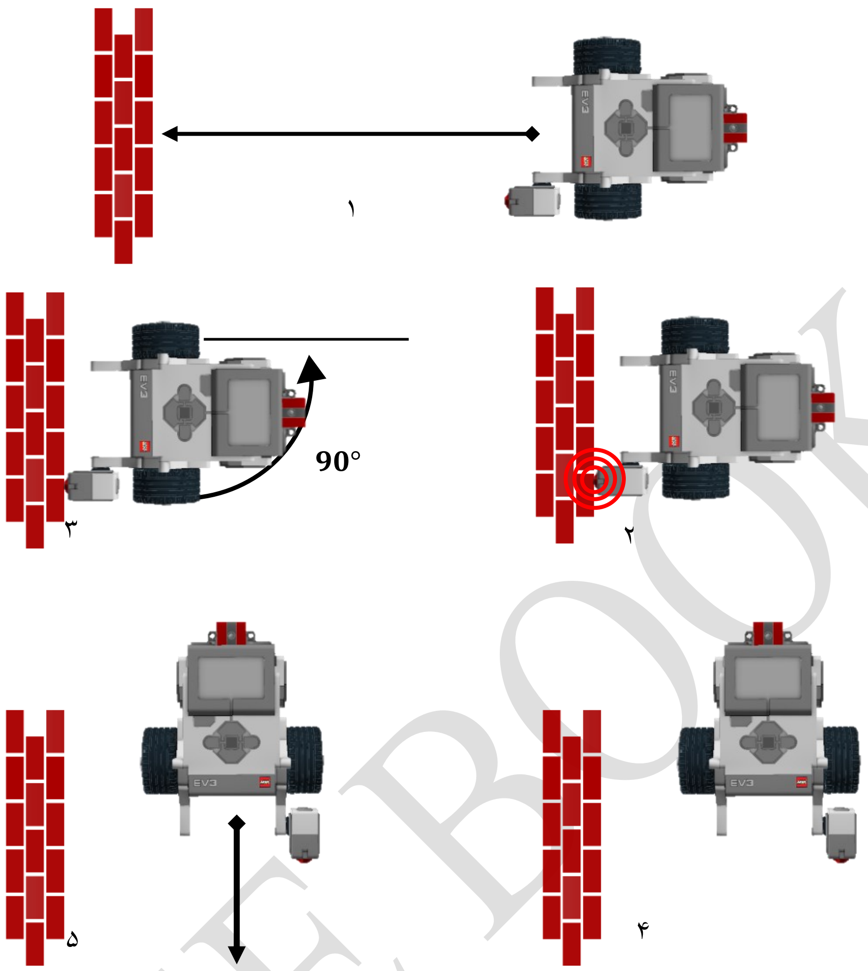


Figure 57 The robot redirects when it encounters obstacles

To do this, we first place the Move Motor block inside the infinite loop so that the robot can continue to move directly:

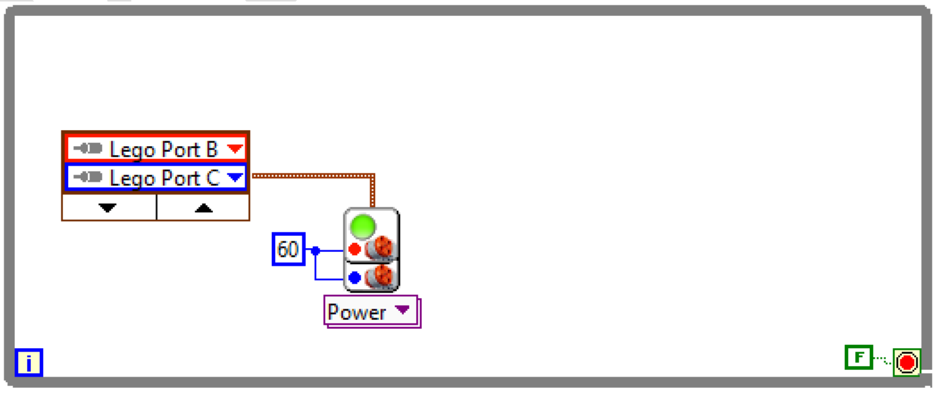


Figure 58 Creating continuous motion on a straight line

Next, if the touch sensor is pressed, we want the robot to stop and spin backward on one of its wheels (if it spins in place or forward, it certainly does not allow the front obstacle to rotate). Use the *Wait For* block to detect the button being pressed and set it to Wait for NXT/EV3 Touch → Pressed. Next, first, stop the motors and then rotate on a wheel:

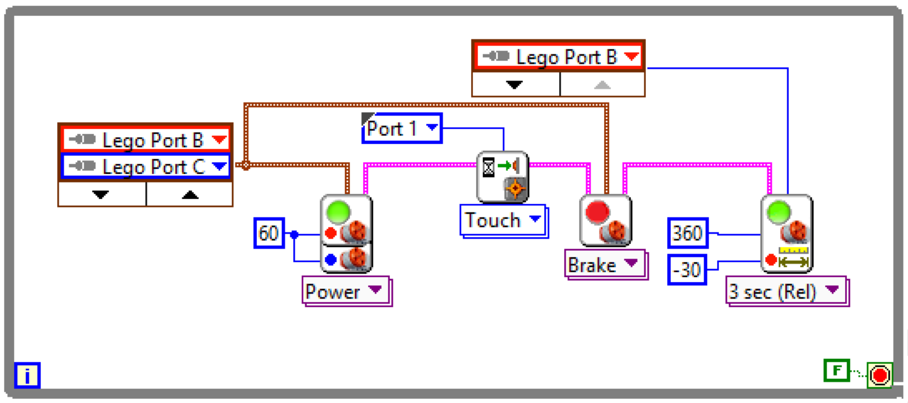


Figure 59 Detection of the sensor button colliding with the Wait For block and the robot turning backwards and repeating this process

In this way, the robot continues the straight path until it hits an obstacle. If the robot hits an obstacle, it rotates 90 degrees and then continues straight again.

ULTRASONIC SENSOR

Figure 60 EV3 Ultrasonic Sensor

An ultrasonic sensor has an ultrasonic transmitter and receiver that measures the distance to obstacles at front of it using ultrasound waves (electromagnetic waves with a frequency beyond the human hearing range). Specifically, the transmitter first sends an ultrasound pulse to the front. The transmitted pulse then returns to the receiver after reflecting from the obstacle. By calculating the time between sending the pulse and receiving the reflected pulse and knowing the speed of sound in the air (for dry air at 20 °C, this value equals 342.2 m/s), the distance to the obstacle can be calculated.

Reading Ultrasonic Sensor

To read the sensors, we generally use the *Sensor* block in the I/O section. There are several ways to get information from the ultrasonic sensor and display it in the software. First, connect the ultrasonic sensor to the EV3 socket by cable (to each port 1 to 4) and connect the socket to the computer's *USB* port. Then, to read the sensor information and display it in the lab, place the *Sensor* block in the infinite *while* loop to continuously update and display the read values from the sensor:

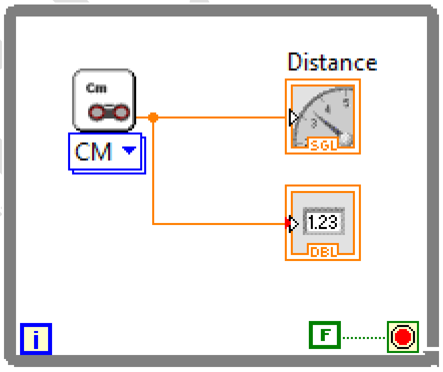


Figure 61 The method to read ultrasonic sensor information

In the Front Panel section, the read values in centimeters are displayed as follows:

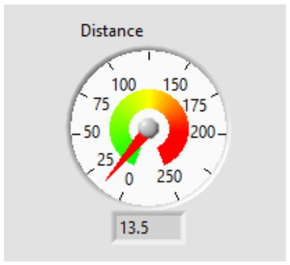


Figure 62 Read the value of the sensor in the Front Panel

Ultrasonic calibration

Experiments can be performed to understand the accuracy of the ultrasonic sensor and the environment covered by the sensor. Figure 64 shows the layout required for the experiment:

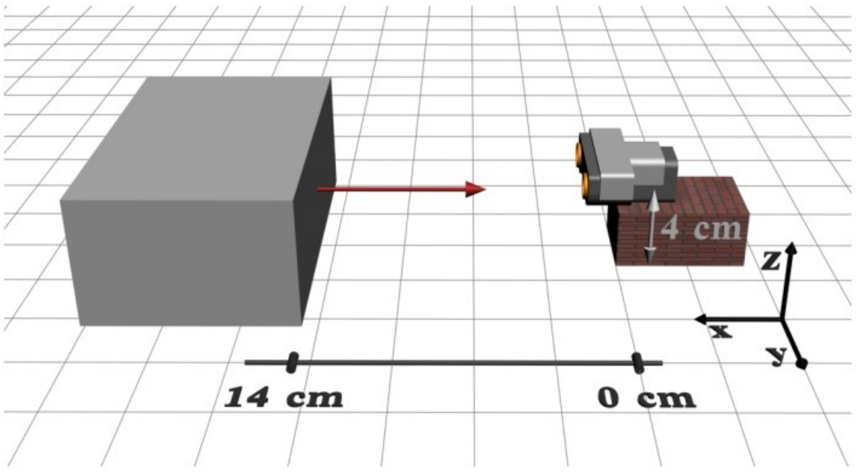


Figure 63 Position of sensor and obstacle in the first test

By considering the obstacle in front of the sensor and increasing its distance from the sensor, the diagram in Figure 65 can be drawn, which shows the distance measured by the sensor in terms of the actual distance. It is also possible to plot the average deviation from the actual value.

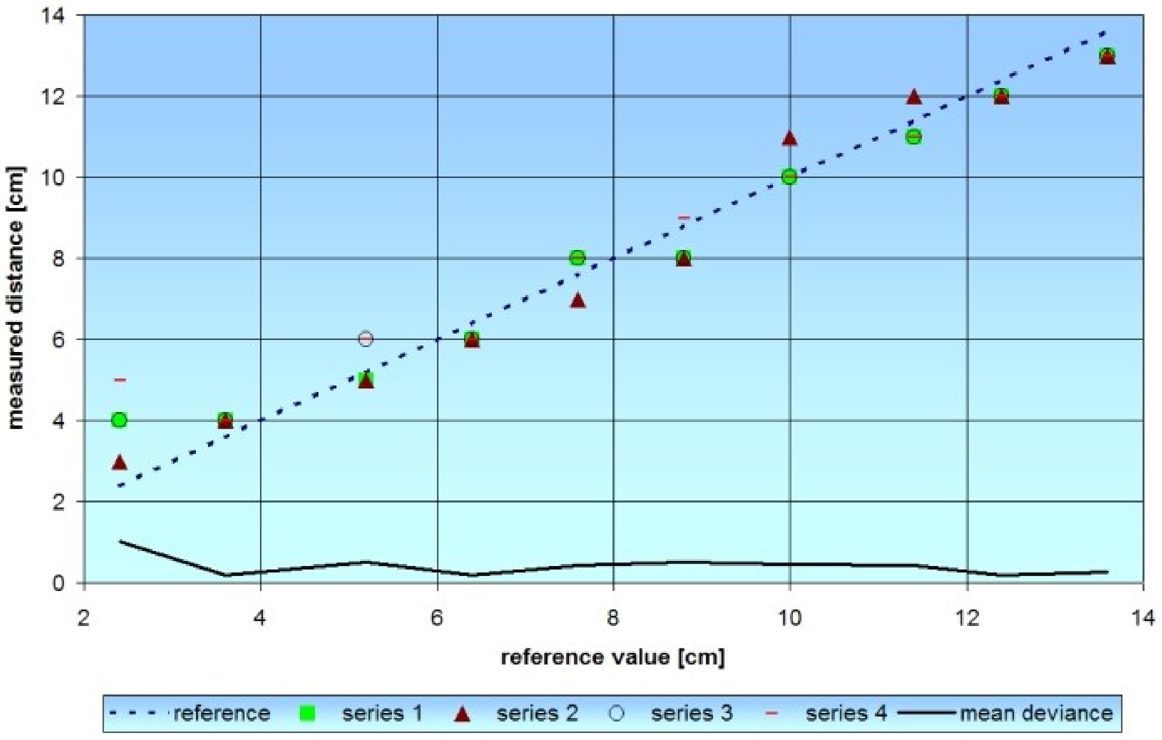


Figure 64 Graph of the distance measured by the sensor in terms of actual distance

Distances of less than 3 cm are not measurable by the sensor. The maximum deviation from the true value is 2.6 cm at a distance of 2.5 cm. The average deviation of this sensor is 0.408 cm (less than 0.5 cm), which indicates the correct behavior of the sensor.

In the second experiment, our goal is to get the field of view of this sensor. In this experiment, the same barrier was used before (14.5 by 9.5 by 6 cm). In this case, we move the obstacle at different distances and angles to the sensor and record the read distances. The sensor is placed in both horizontal and vertical positions. Figure 66 shows the layout with the horizontal position of the sensor.

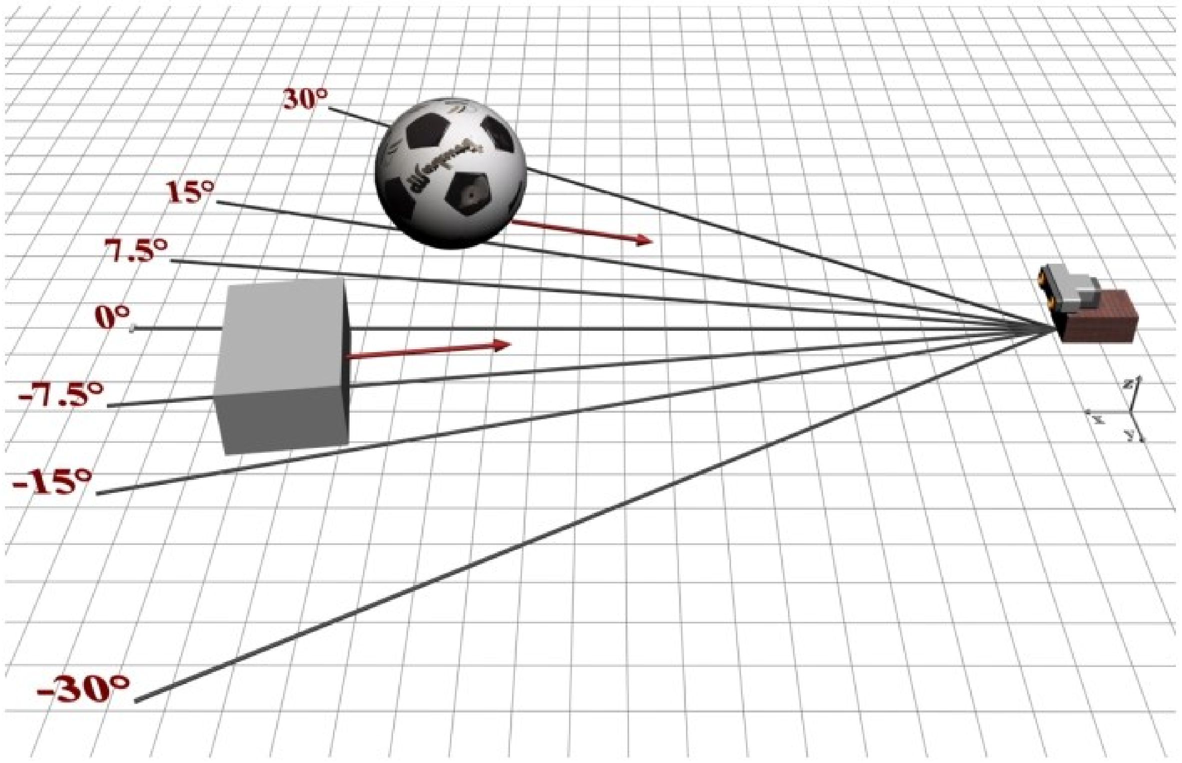


Figure 65 Obtaining the range of view in the second experiment

In Figure 67, a graphical representation of the field of view is drawn in the horizontal position.

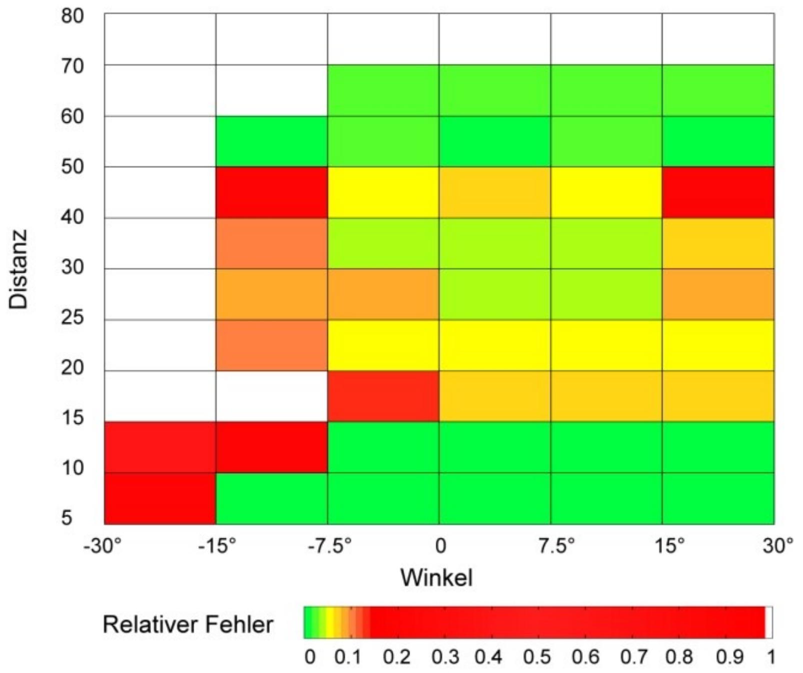


Figure 66 Graph of the horizontal viewing angle of the sensor in terms of obstruction distance

The results show that the ultrasonic sensor should always be in the horizontal position because the range and depth of vision are reduced in other positions. The sensor in the left eye has some visual impairment because the left eye of this sensor is the receiver of ultrasonic waves, and the right eye is the transmitter of those waves.

After the above tests were performed for the statistical behaviors of the ultrasonic sensor, we proceed to dynamic tests. Figure 68 shows the distances read by the ultrasonic sensor as it approaches the wall. The information in this diagram is plotted by programs in the LEGO software under LabView, in which the ultrasonic current values and the angle values of one of the moving motors are stored in a file in the break. This file is then downloaded from the break.

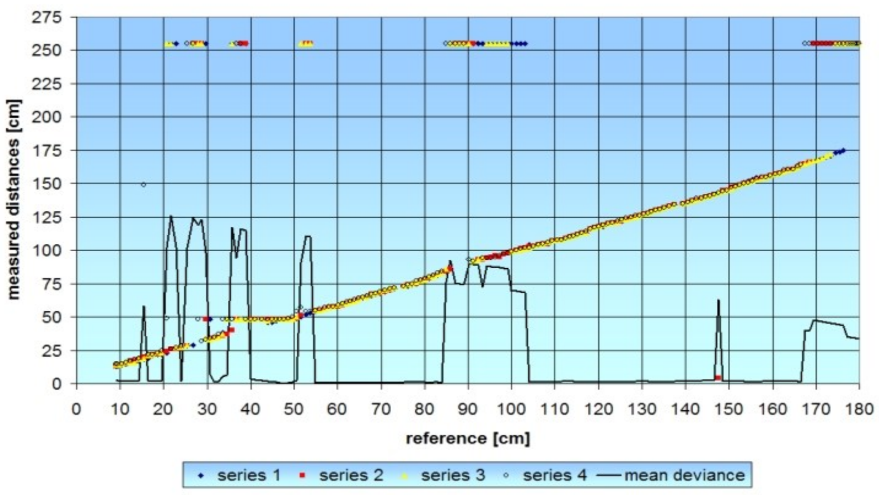


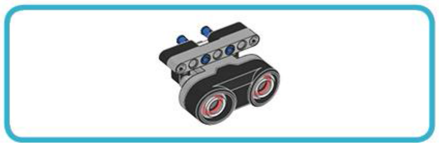
Figure 67 Graph of the ultrasonic sensor dynamic test result

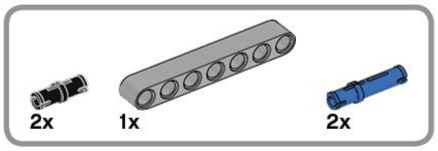
The dynamic test reveals two weaknesses of the ultrasonic sensor. The first is that the sensor displays 255 cm in some areas instead of the actual distance. The second case, which is even more important, is the critical area between 25 cm and 50 cm, in which case the sensor in most cases displays the wrong value of 48 cm.

Ultrasonic Sensor Applications

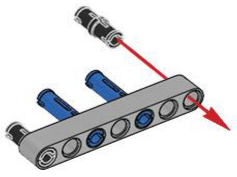
This sensor can be used for different applications. One of its applications is to prevent the robot from colliding with obstacles. Next, we modify the educator robot so that it detects the obstacles in front of it and avoids them by changing its course.

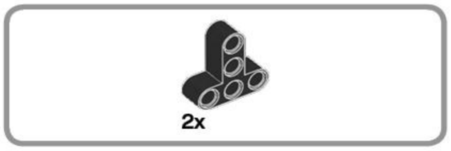
Make the following corrections on the educator robot:



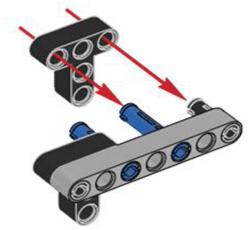


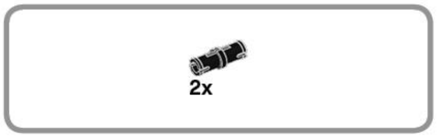
1



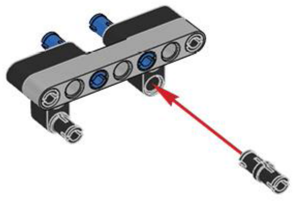


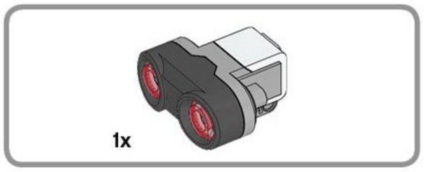
2



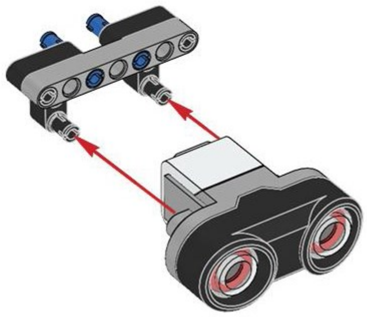


3

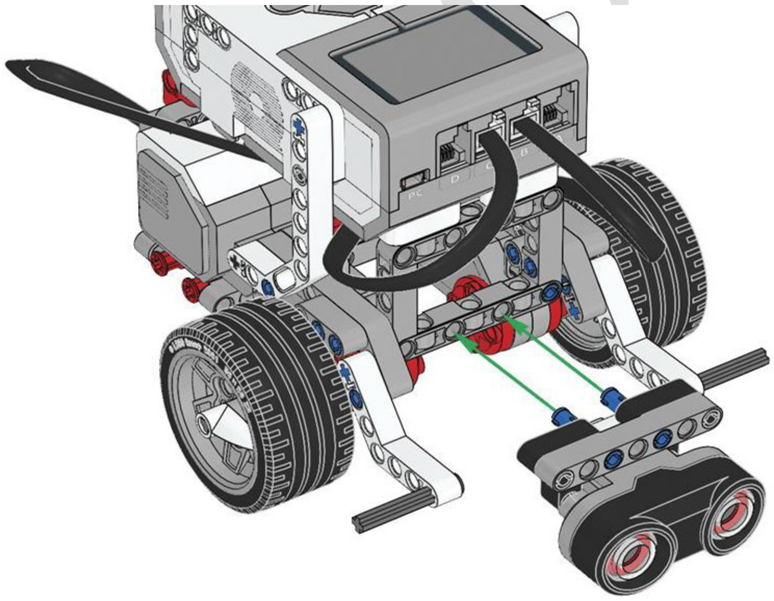


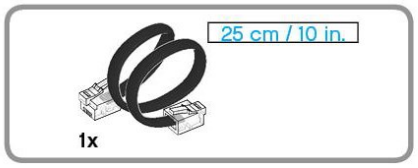


4

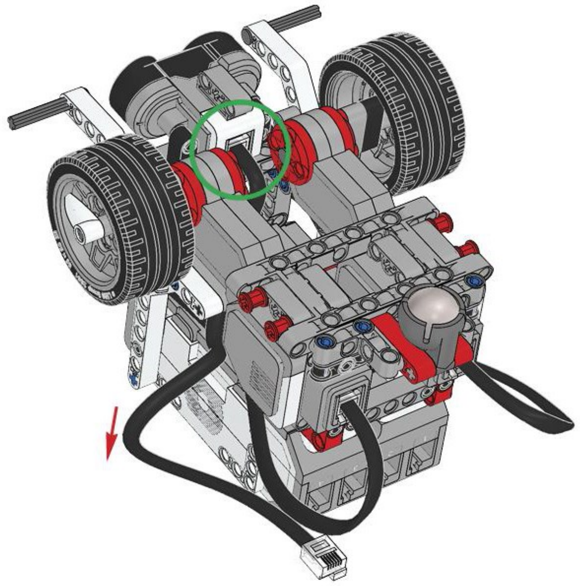


5

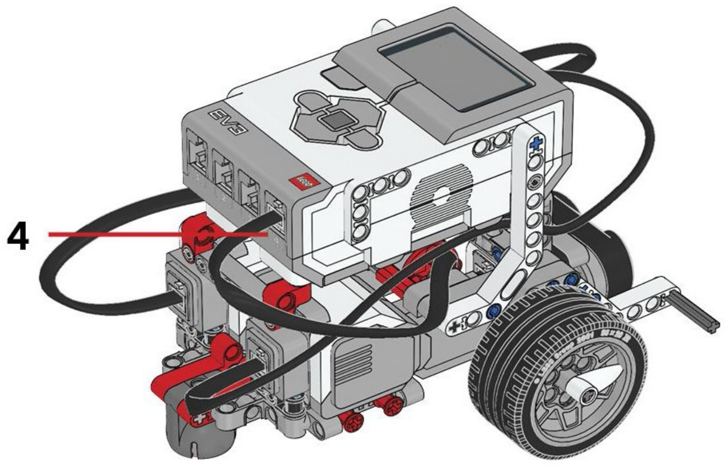




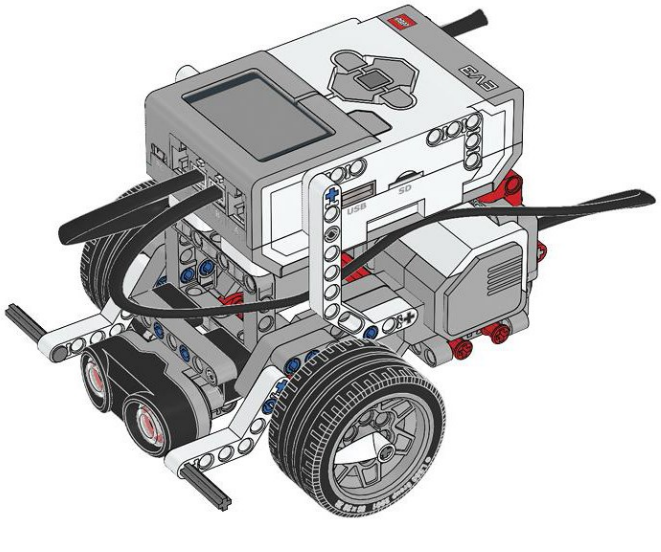
6



7



8



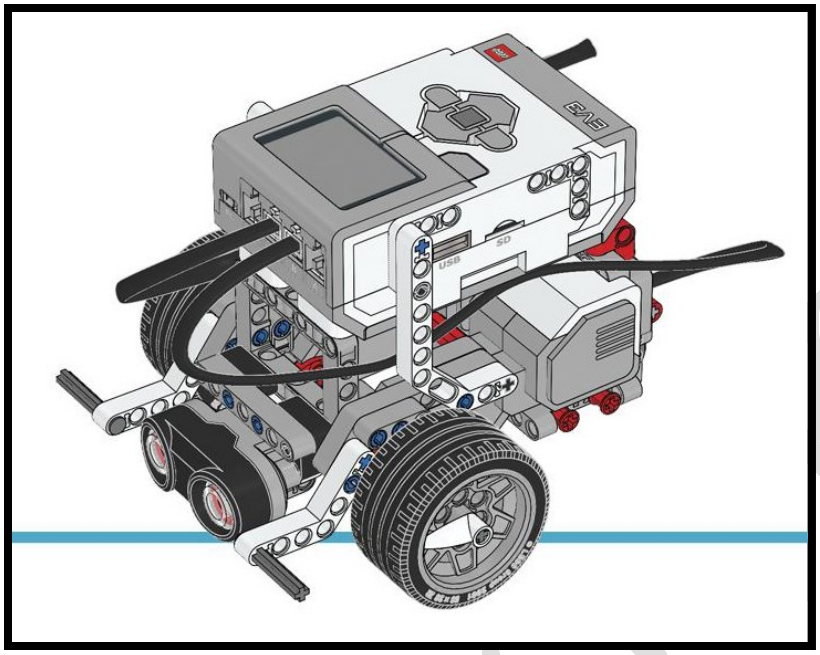


Figure 68 Installation of an ultrasonic sensor on the educator robot

We want to program the robot in a way that moves in a straight line by default and when it gets close to an obstacle, it rotates and continues its direct movement again. In this way, the robot will be able to constantly move and avoid hitting obstacles.

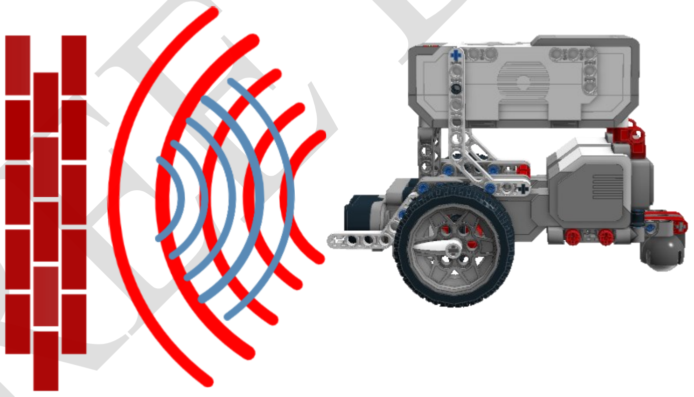


Figure 69 Obstacle detection with ultrasonic sensor

So first, like the following, we place *an infinite loop* always to repeat this process, in which we program the direct movement of the robot:

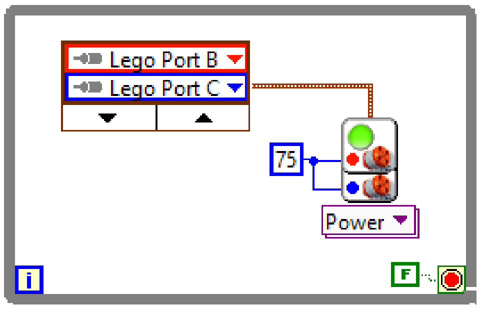


Figure 71 Movement of motors in the infinite while loop

In the next step, we must use the Wait For block to stop the engines and thus prevent collisions with obstacles. This block has arbitrary conditions that specify that when the condition is met, the robot stops working and performs the next activity. Here we want the robot to continue to move straight until an obstacle less than 10 cm away is detected, and if an obstacle less than 10 cm away is detected, the robot stops and rotates 90 degrees clockwise. Then set the *Wait For* block condition to *Ultrasonic* → *less than (cm)* and give the fixed number 10 as input to the block:

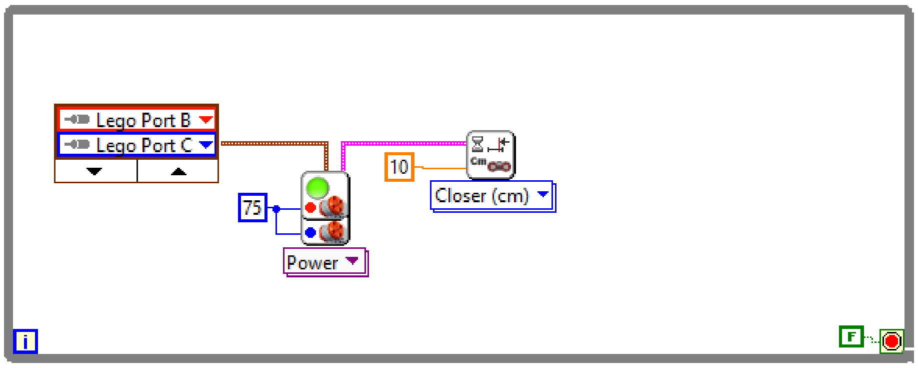


Figure 70 Adding the wait for block

What to do after the *Wait For* is to stop the wheels first and then move one of the wheels to the right size for the robot to rotate 90 degrees:

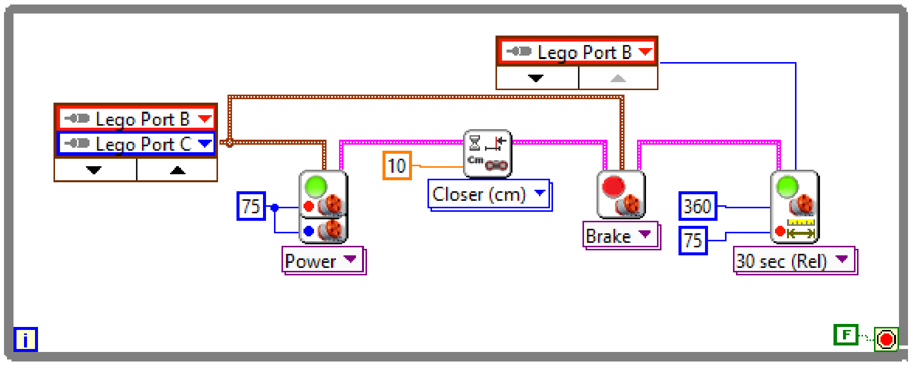


Figure 71 Adding Stop Motors block and rotating the robot

GYROSCOPE SENSOR

Figure 72 EV3 gyroscope sensor

The gyroscope sensor is used to measure the amount of rotation or to measure the speed of rotation. The gyroscope sensor in this set can only measure rotation around an axis. The EV3 gyroscope sensor is made using *MEMS (Micro Electro-Mechanical System)* technology. The operation of the gyroscope sensor is based on the force due to Coriolis acceleration. Consider a small mass of crystal inside a metal enclosure. This small mass of crystal shrinks slightly when an electrical voltage is applied and returns to its original size when the voltage is removed. Now, if we connect a spring to this mass and constantly disconnect and connect the electric current, the mass oscillates along an axis. As we know from the law of energy conservation, moving objects tend to keep moving in the direction of their motion unless an external force is applied to them in the other direction. When the sensor rotates, the crystal mass maintains its linear motion in the original direction of oscillation. However, because the sensor's body is also rotated, this crystal mass no longer oscillates in the same direction as the sensor body. The gyroscope sensor measures the amount of this deviation and calculates the amount of rotation and thus can measure the amount of rotation and the rate of rotation (angular velocity).

Gyroscope sensor is used in many industries, from cell phones to satellites and massive industrial robots.

Reading Gyro Sensor

First, we add the Sensor block from the *functions* → *MINDSTORMS Robotics* → *I/O section* to our diagram block.

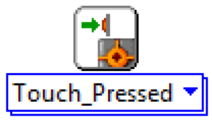


Figure 73 Sensor block

Two modes of operation are defined for this sensor. One is for reading the Read Angle, and the other is for reading the Read Rate. First, we place this block on the Read Angle and place it inside the infinite loop:

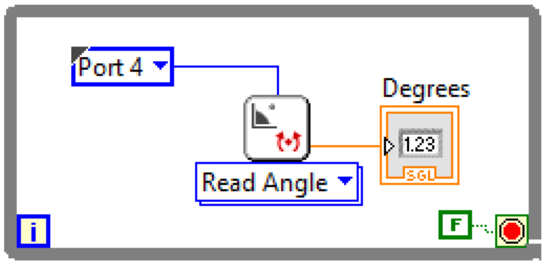


Figure 74 Reading the angle value from the reference

In this case, the value of the angle measured by the gyroscope sensor is read at any time and displayed in the *Front Panel*. Note that in this case, the read angle is measured relative to the default sensor reference. In the next section, called the calibration of the gyroscope sensor, we will change this reference.

In another mode of this sensor, we can read the rate of change of angle. As before, place the sensor inside the infinite loop and then select the *Read Rate* option. Now, by running the program, you can see the amount of sensor rotation in the *Front Panel* at any time:

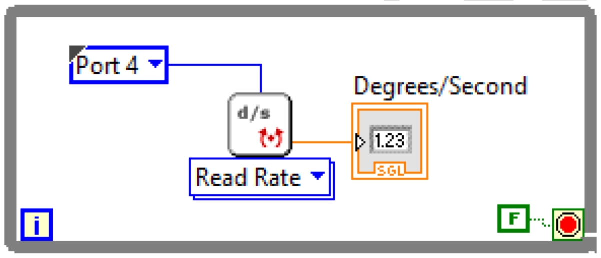


Figure 75 Reading the angle change rate

Gyro Sensor Calibration

As mentioned in the previous section, the angle read by the sensor relative to the sensor's default reference is announced. To change this reference, you can use the Sensor block and the Reset Gyro option, so that every time we run the program, the reference is the same as the original angle, and the sensor reads other angles relative to that angle:

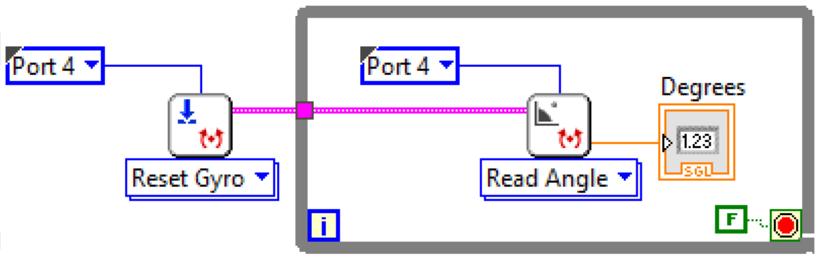
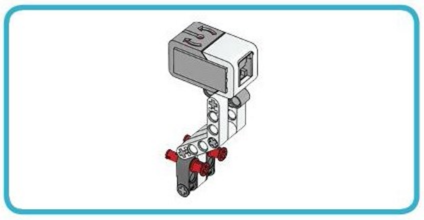


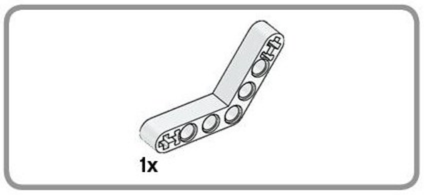
Figure 76 Resetting the reference angle and reading the angles relative to the initial sensor angle

By executing the above program, first by resetting the gyroscope sensor, the reference angle is changed to the initial angle of program execution, and then the other read angles are declared relative to that initial angle.

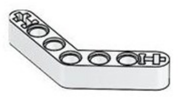
Gyro Sensor Applications

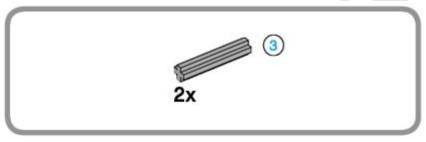
The gyroscope sensor is used in a wide range of robots. First, to get better acquainted with this sensor, using the basic robot, we write a program that always keeps the direction of the robot to one side, and the robot resists the change of direction. As we became acquainted with the proportional controller in the line-tracking robot, here we use this type of control to implement this goal. For this purpose, we first connect the gyroscope sensor to the *Educator* robot:



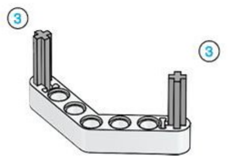


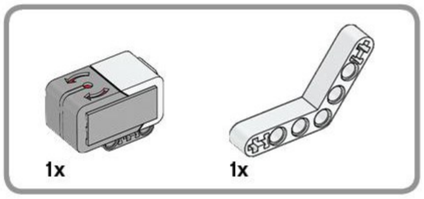
1



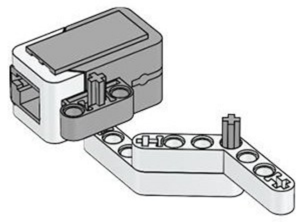


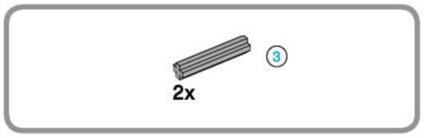
2



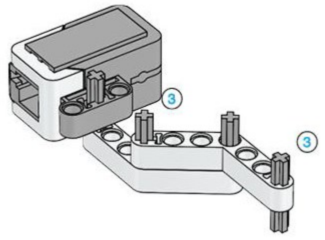


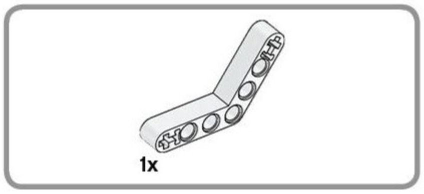
3



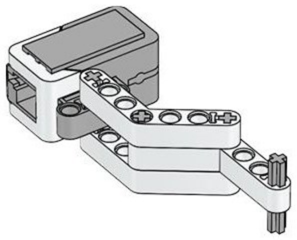


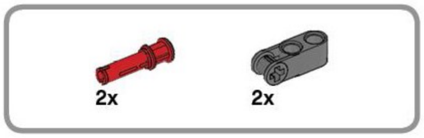
4



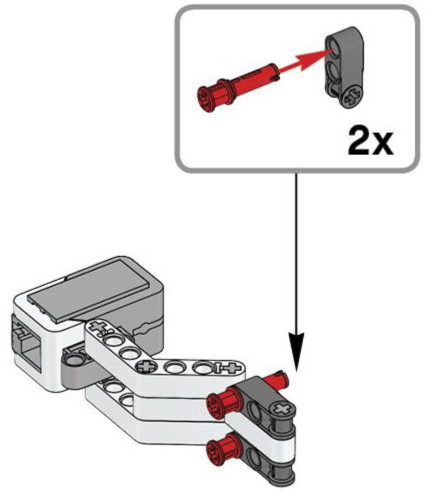


5

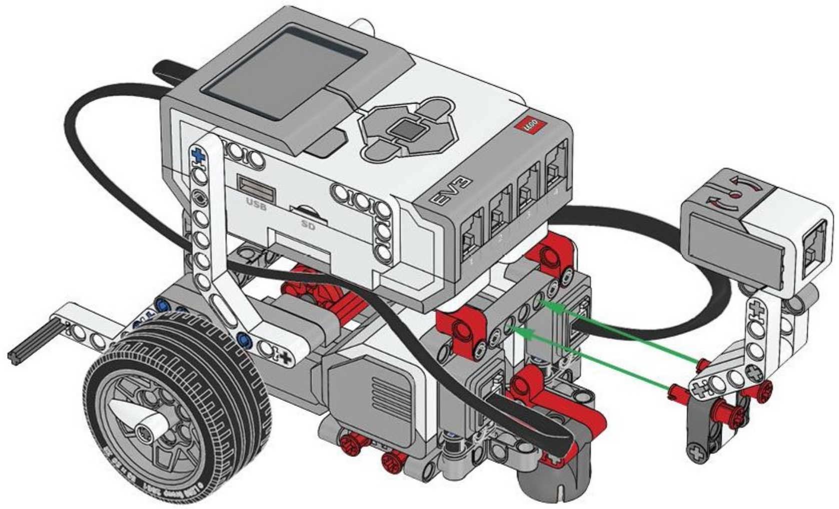


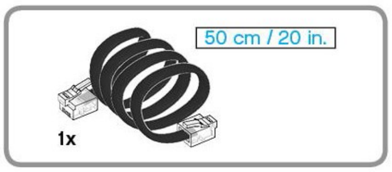


6

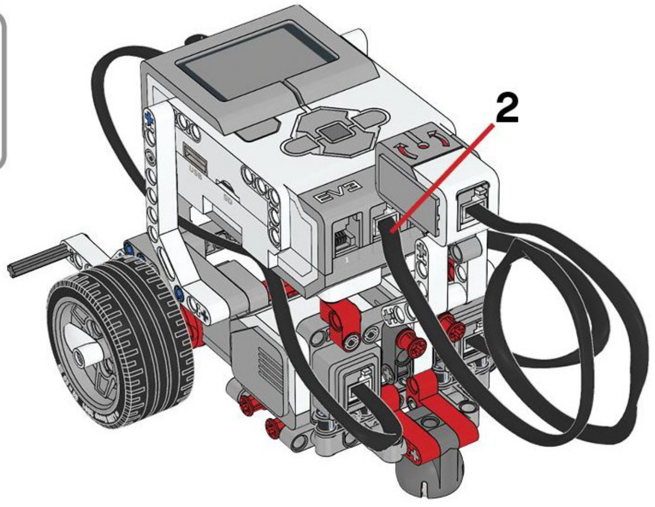


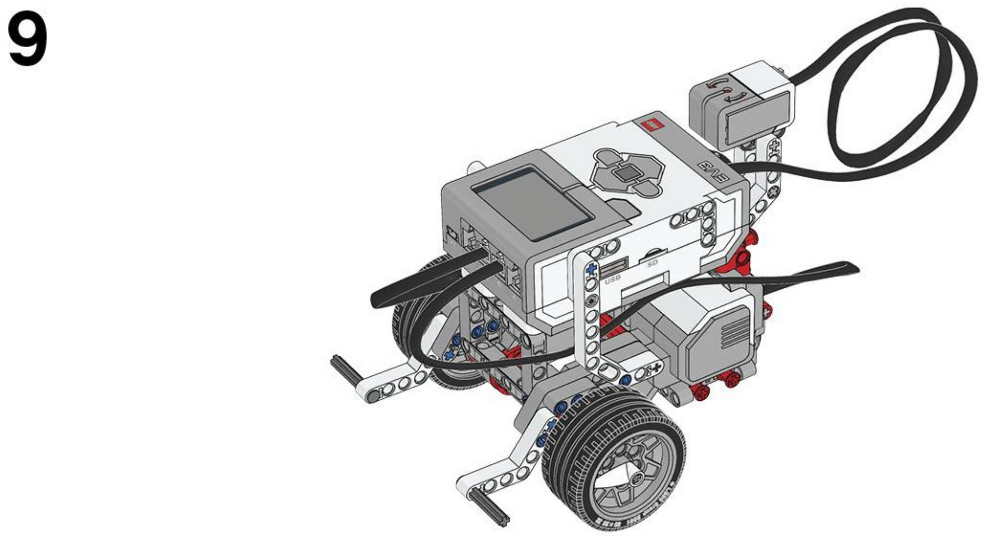
7





8





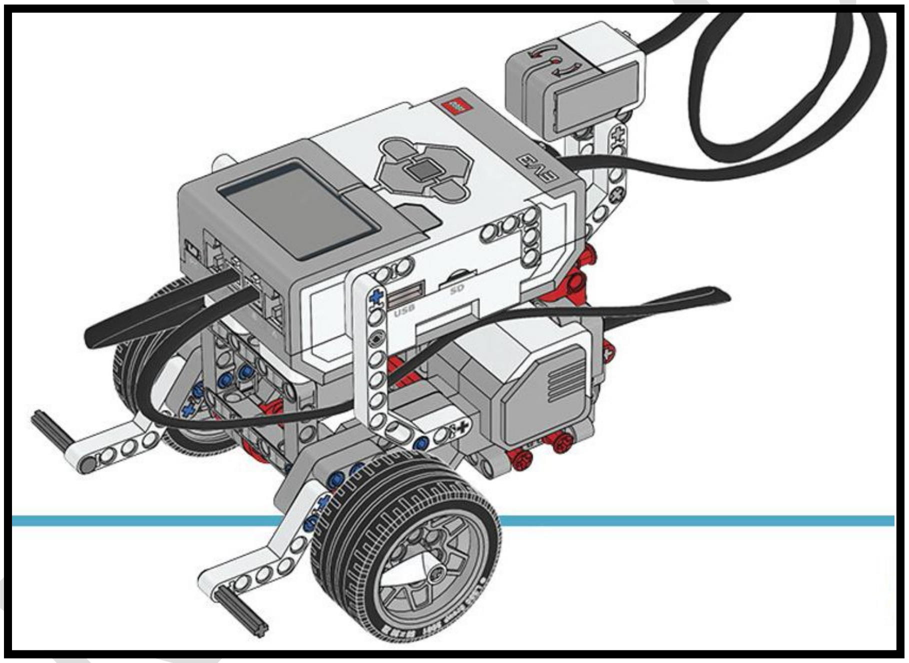


Figure 77 Connecting the gyroscope sensor to the Educator robot

Suppose we want the robot to always be at a zero-degree angle, so our reference value is zero.

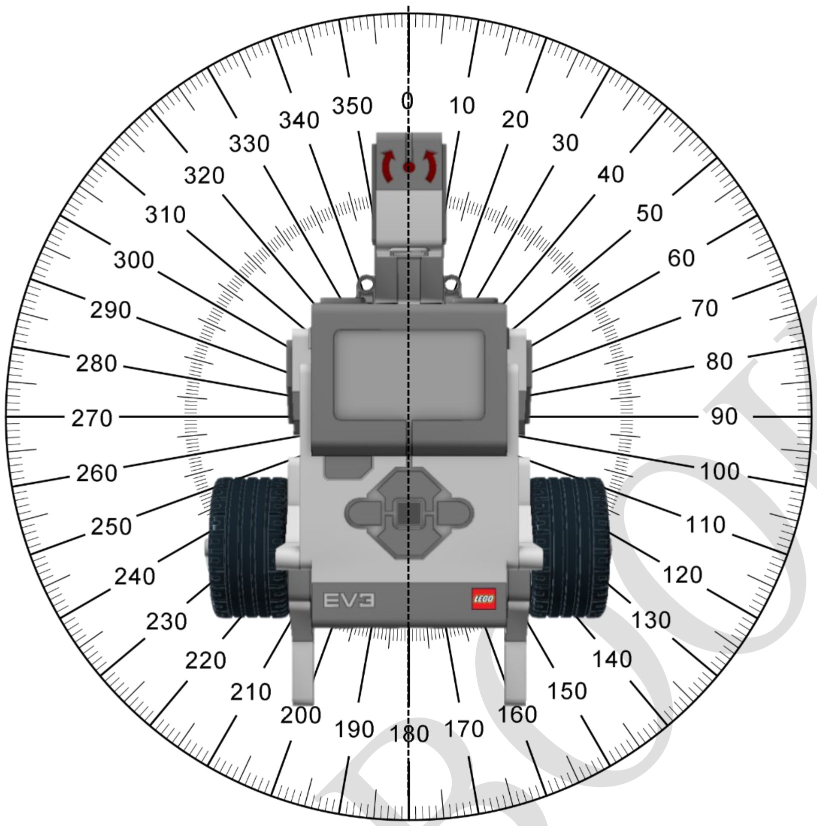


Figure 78 Angle measured by a gyroscope sensor

Now we have to generate the error signal and, after multiplying a gain factor in it, apply the amplified signal to the motors. To generate an error signal, we do the following:

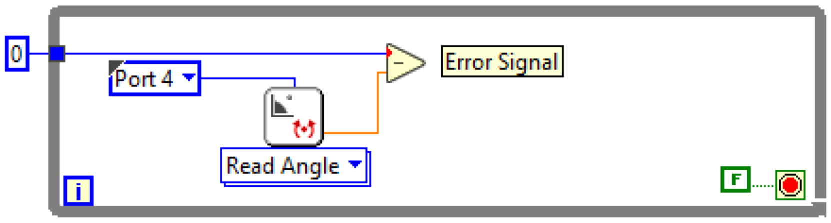


Figure 79 Error signal formation

Then we multiply this signal by a fixed number and apply it to the motors. Note that our goal here is to rotate the robot around its own axis, so the direction of rotation of the motors should be opposite to each other, so the amplified signal should be applied to one motor, and the negative signal should be applied to the other motor.

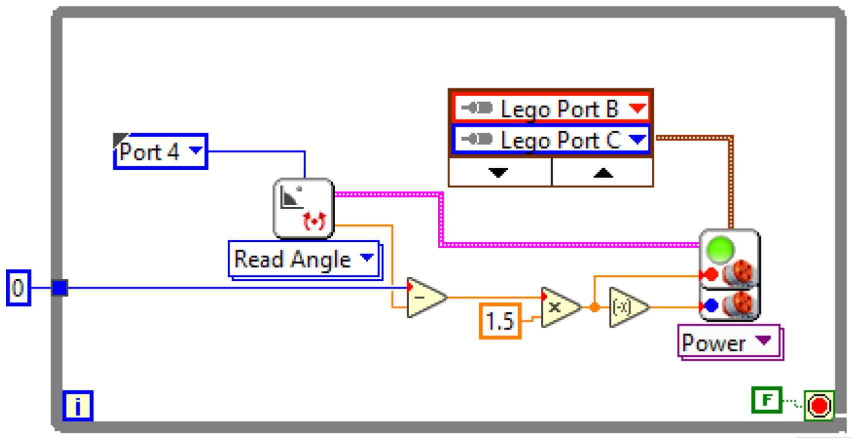


Figure 80 Adjusting the motor to a zero-degree angle by rotating the wheels

Finally, the desired program is shown above. Now, by running this program, the robot always maintains its direction towards the zero angles, and if it changes direction, it rotates in proportion to the difference it has from the zero angle and brings itself to the zero angle.

COLOR/LIGHT SENSOR

Figure 81 EV3 color sensor

Visible waves comprise a small part of the electromagnetic spectrum from a wavelength of 400 to 700 nm; by changing the wavelength in the visible waves, the color of the reflected spectrum changes. The use of color sensors has many applications in robotics and industry. In the simplest case, photoresistors can identify the color of the received spectrum, which applies different electrical resistances by changing the color. But the most widely used color sensors are made by semiconductor technology, in which case some LEDs create a different electric current according to each color. LEDs are usually used to illuminate white objects more accurately to identify the color of each object. Then, by applying specific filters on the received wavelength, it can be interpreted in three colors: red, green and blue. These three colors are the main colors that can be used to produce other colors. The basis of the work of the human eye is the same.

The sensor in the EV3 set, in addition to being able to detect the values of the three primary colors red, green and blue, can detect the absence of color (clear) and light intensity. In the following, we will examine the method of using this sensor for different purposes.

Reading Color/Light Sensor

Just as we used the Sensor block in *functions* → *MINDSTORMS Robotics* → *I/O* to read other sensors' values, we add this block to our program diagram block to read the values of this sensor in a similar way.

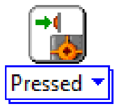


Figure 82 Sensor block

After placing the sensor block, click on the bar at the bottom of the block to go to the Read EV3 Color directory. Here are 4 options as shown below:

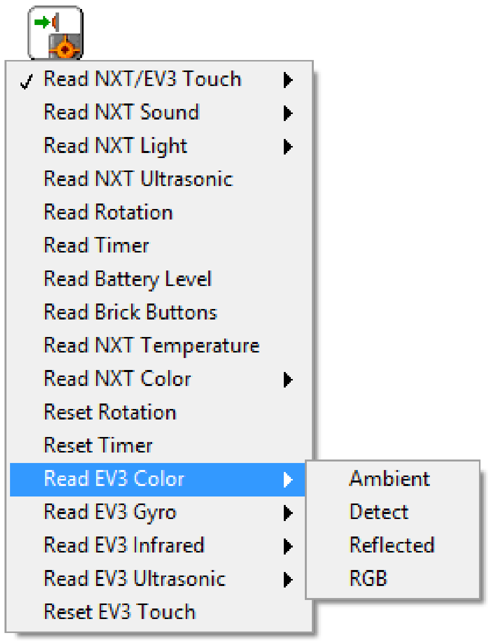


Figure 83 Different modes for the color sensor

- The first mode is **Ambient**, in which case the sensor is in the mode of measuring the intensity of ambient light. Selecting this mode creates an input node and an output node in the block:

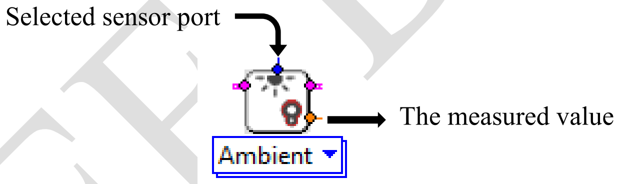


Figure 84 Color sensor inputs and outputs

To specify the port to which the sensor is connected, right-click on the node at the top of the block, select *Create* → *Constant* and select the appropriate port from the created bar. In the next step, to show the amount received, which indicates the amount of light in the environment, right-click on the node to the right of the block and click *Create* → *Indicator*. In this case, a window called Scaled Value will be created in the *front panel*, in which the read value of the sensor will be displayed:

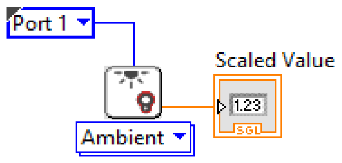


Figure 85 Color sensor input and output connection

In order for this value to be repeated continuously, we place it in an *infinite while loop*:

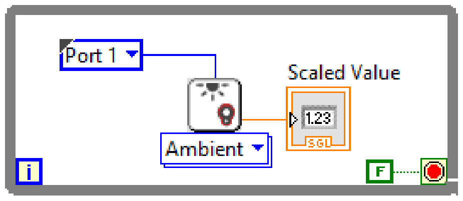


Figure 86 Putting the measurement process in an infinite loop

In this case, by uploading the program on the break and running the program continuously, you can see the read value of the sensor.

- The next mode is the **Detect** sensor. In this mode, the sensor detects 7 colors and identifies each color with a number. According to the table below, each number is assigned to one color:

Table 2 corresponds to the code for each color

Red	5	Black	1
White	6	Blue	2
Brown	7	Green	3
No Color	0	Yellow	4

This mode, like the previous mode, has a node to specify the port connected to the sensor and another node to identify the color.

To read the color sensor and identify the colors, we can place the sensor in an infinite loop as before and display its output according to the table above.

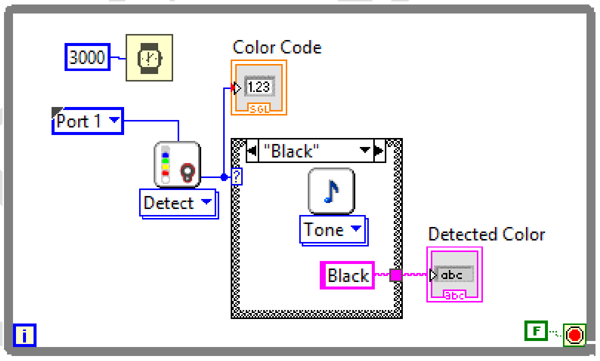


Figure 87 Reading and color recognition by the color sensor

By adding a *Case Structure*, the detected color can be viewed on the Front Panel. The Wait block (ms) is placed to create a time interval between each color detection, which causes the sensor to detect color every 3 seconds (3000 milliseconds).

- The third mode of the sensor is called **Reflected**. In this case, the sensor measures the light reflected from the surface. This mode has many applications, especially for the design of line-following robots. In this way, the values read from the reflection of the black line are different from the values read from the ground. As in the previous cases, in this case, we specify the port connected to the sensor and display the read value on the *Front Panel*.

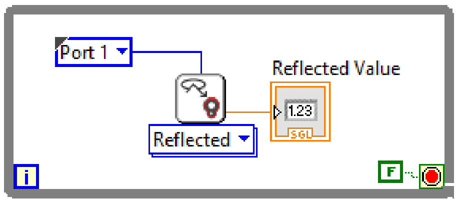


Figure 88 Measuring and displaying the reflected value from the surface

In the next section, we will calibrate this state of the sensor to design the line tracking robot and other similar robots properly.

- The last mode of color/light sensor is **RGB**. RGB stands for red, green and blue, which are known as the principal colors. In this case, the sensor shows the detected color as a combination of these three colors. By specifying the port connected to the sensor and creating the indicator (Indicator) by right-clicking on the RGB Color node and selecting the *Create* → *Indicator* option, the *RGB Color* block can be added to the set.

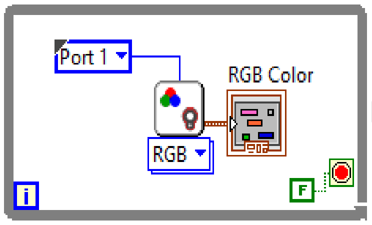


Figure 89 Measuring and displaying RGB values

In this case, three RGB values are displayed in the Front Panel.

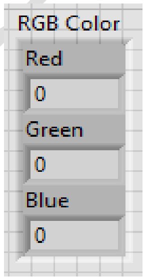


Figure 90 Displays RGB values in the Front Panel

Color/Light Sensor Calibration

As mentioned, the color/light sensor can measure light intensity, color detection, RGB values and reflection. In the first three cases, the sensor works according to factory settings, and in most cases, calibration is not required for specific applications. However, in the case of reflectance measurement, it is necessary to adjust it according to the conditions and working space of the robot and to determine the correct readings and accuracy of the robot, the values read on the black line and the background as a criterion.

For this purpose, we create a program as follows so that the appropriate values for the black line and the background can be extracted and measured by performing its steps.

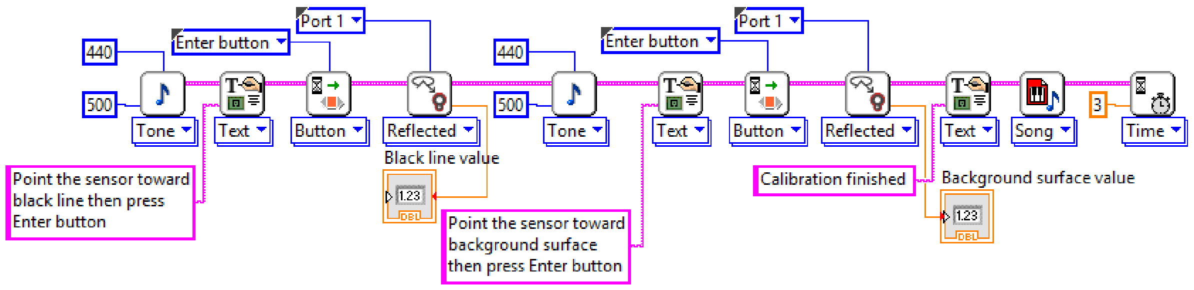
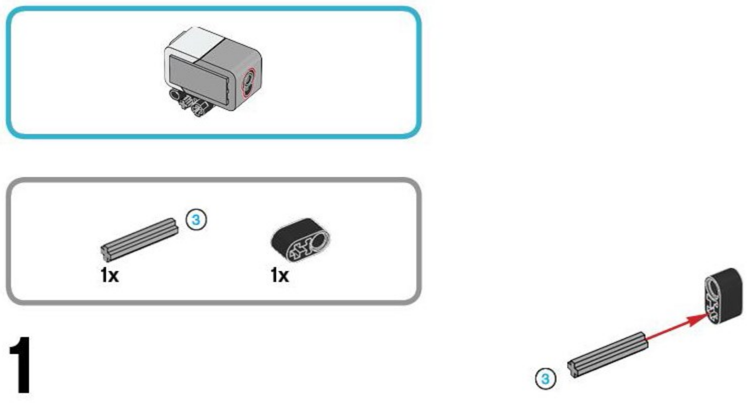


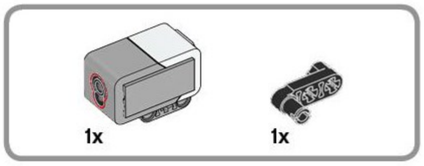
Figure 91 Color/light sensor calibration process

First, a short beep sounds to announce the start of calibration, and then a message is generated on the brick's screen based on placing the sensor towards the black line, and the device waits for the Enter key to be pressed on the break. It indicates the Blackline value in the Front Panel. The next step is to read the background value, which is similar to the previous state of beep and the message is created on the brick, and the result is displayed as Background surface value. Finally, the message "Complete Calibration" is played, and a song is played announcing the end of the calibration. The read values for the black line and the background can be used in the design of the line tracking robot, etc. In the next section, we will use these values to build a line tracking robot.

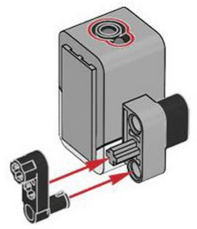
Color/Light Sensor Applications

One of the most widely used light sensors is the line tracking robot. These types of robots have been widely used in industry and production lines. To start designing such a robot, you first need to make the following changes to the Basic Educator robot:

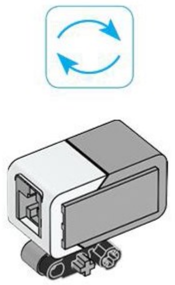




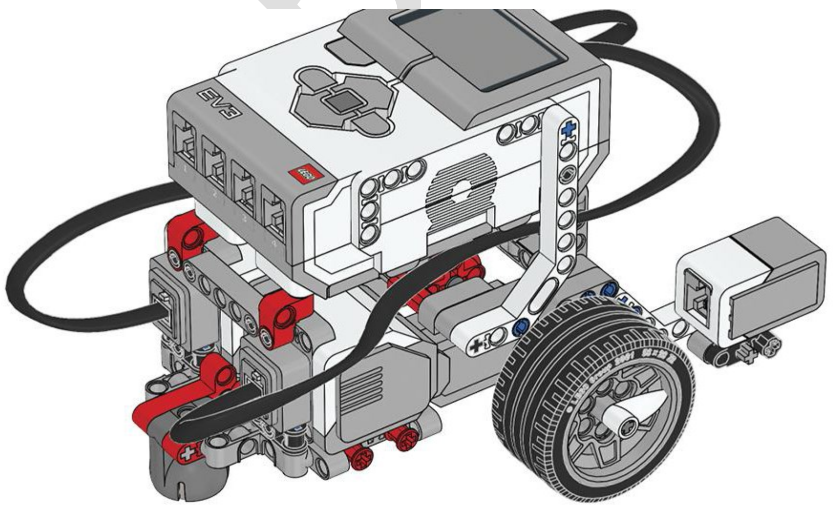
2

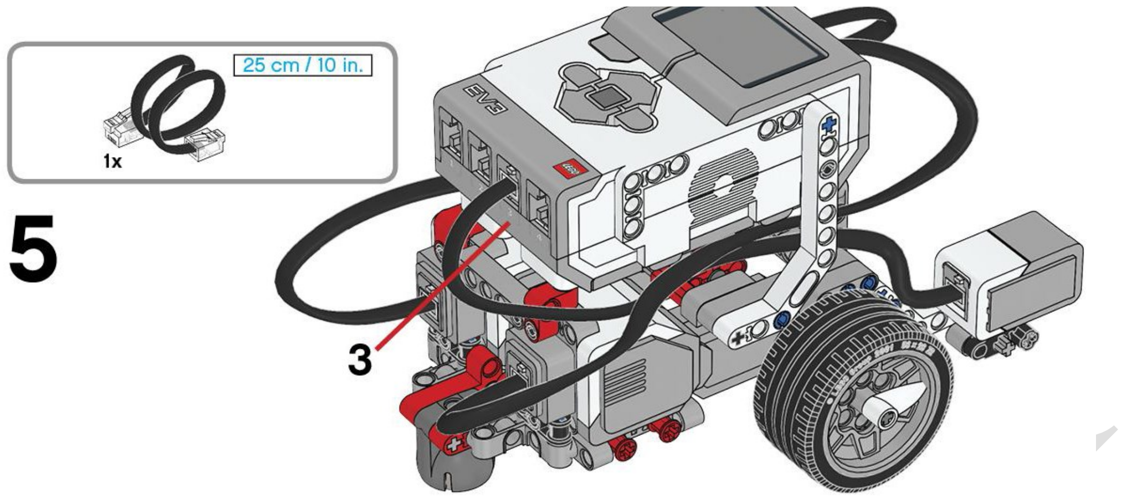


3



4





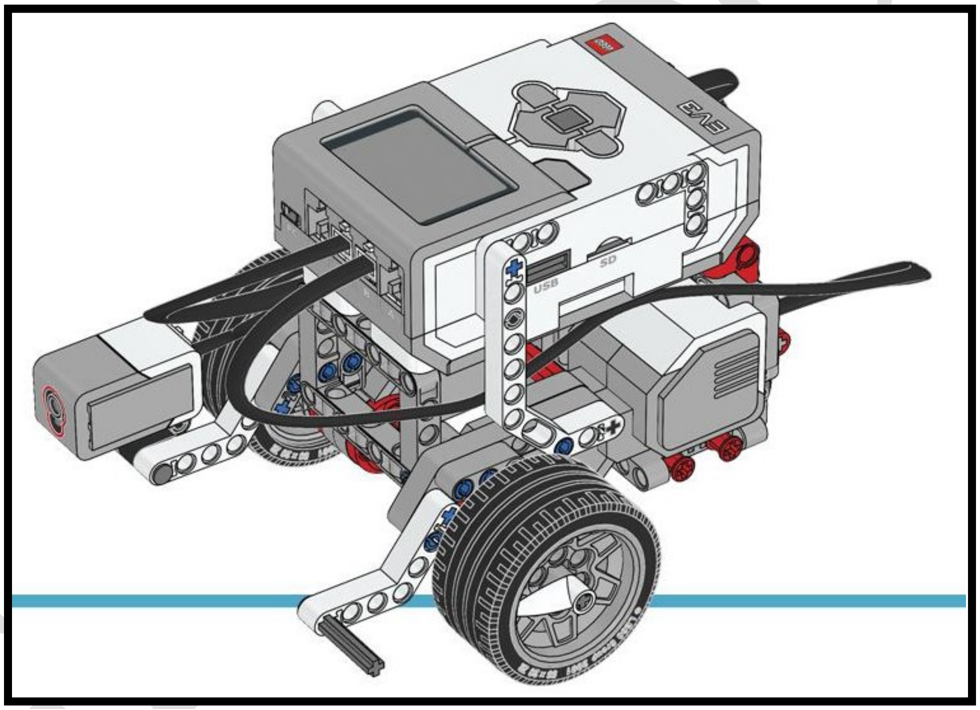


Figure 92 Installing the color sensor on the educator robot

To build a line-following robot, we first need to define the problem clearly. One way to determine the path of a robot is to plan the robot and measure the desired path. Nevertheless, if the route changes, the robot program must be updated and modified. In addition, there are usually errors in measuring and walking the distances determined by the robot. If one path is repeated several times, the other robot is not in the desired path or does not have the necessary accuracy.

Another way to determine the route and navigate it for the robot is to follow the line embedded in the desired path. In this case, you can be sure that the robot is always moving in the desired direction and will never make a mistake or reduce accuracy.

The line tracking robot must be able to distinguish the line drawn on the ground from the background surface. Here we design a robot that detects the black line from the surface on which the robot moves and follows it. Although this robot is a line-following robot, it actually follows the line.

Instead, the robot aims to follow the edge of the line (the boundary between the line and the background). Because we are only going to use a light sensor here, if we follow the line by twisting the path, the robot will not be able to detect the direction of the twist and, therefore, will not be able to follow the line. However, if our goal is to follow the edge of the line, we can detect the direction of rotation of the path and rotate the robot accordingly.

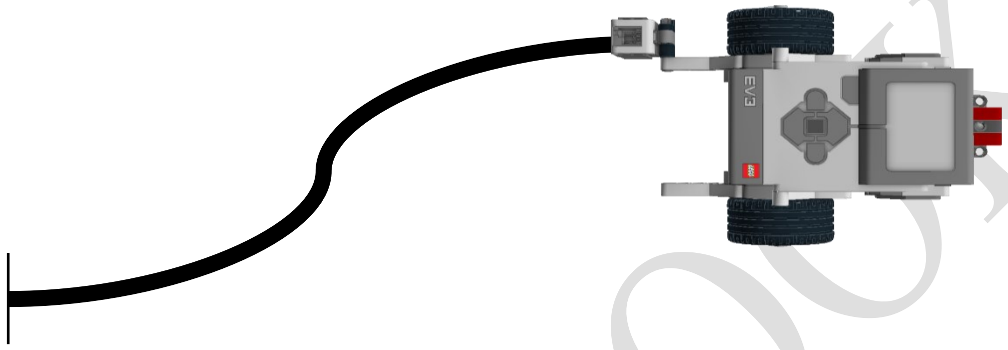


Figure 93 Line tracking by robot light/color sensor

To distinguish the black line and the background, according to the previous section called Calibration, we obtain the read values of the sensor for reflection from the black line and the background. The intensity of light reflected from the black line is much less than the bright background surface. As we get closer to the edge of the black line, the amount of reflection increases, and when we reach the background level, this value reaches its maximum.

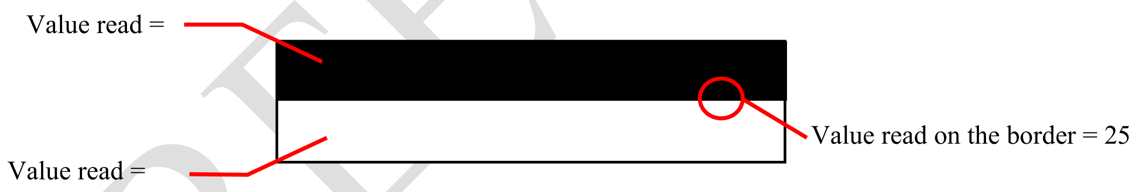


Figure 94 Values read on the line and background

In the following algorithms, we use these values to design and program the robot.

Simple Algorithm

Different algorithms can be considered for this purpose. The most straightforward algorithm to achieve this goal is to use a two-state algorithm. In this way, at any moment, the robot reads the measured amount of reflection from the sensor and by comparing this value with the amount of reflection at the border of the black line (the middle of the value read on the black line and on the background), the direction of rotation of the robot We specify. In fact, our robot is either turning to the right or turning to the left in this algorithm. In this way, the robot follows the line step by step.

First, we will read the sensor's output value and compare it with the reflection value at the line boundary. To calculate the amount of reflection at the line boundary, the following equation can be used:

$$border\ reflection = line\ reflection\ + \left(\frac{line\ reflection\ +\ background\ reflection}{2}\right)$$

The block diagram so far works as follows:

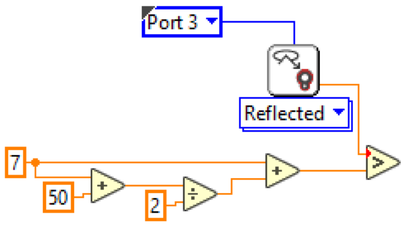


Figure 95 Obtaining the value of reflection on the boundary and comparing the output value of the sensor with it

Next, if this value is more than the value on the border, the robot will turn to the black line. If we always want to follow the left edge of the path, this rotation will be to the right, so hold the right wheel steady and move the left wheel. In order for this comparison process to always be repeated, we must place all of these blocks within the Infinite *While* loop:

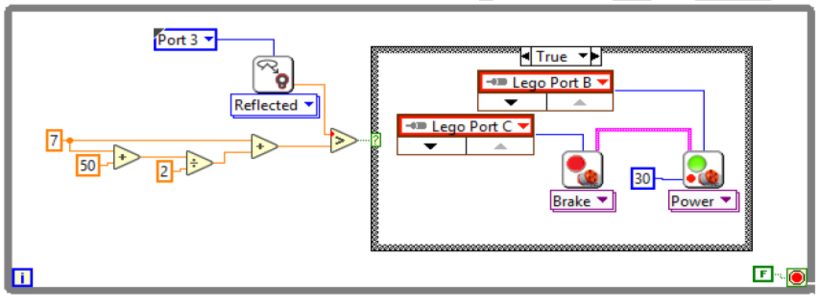


Figure 96 Rotate right to approach the black line

To reverse the said state, the motors must be commanded in reverse. That is, when the value read is less than the value on the border (= not more), i.e., the robot has moved into the black line, then the left wheel must remain fixed, and the right wheel must move so that the robot is tilted to the left.

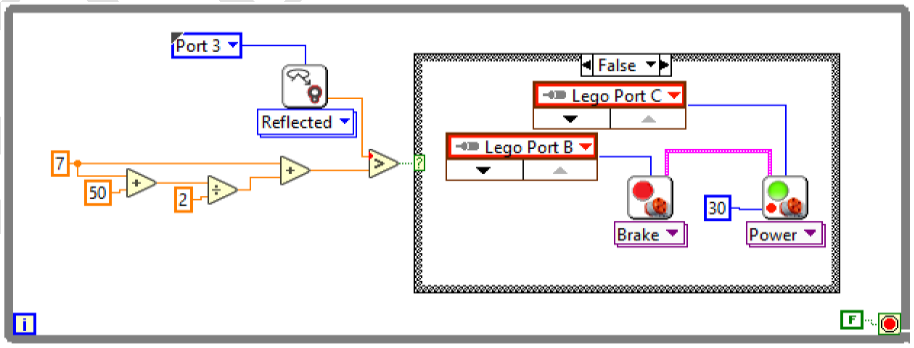


Figure 97 Rotate left to distance from the black line

Be sure always to use the Stop Motors block to hold the motors that is supposed to be stationary; otherwise, the motor will repeat the previous command.

Proportional Algorithm

In this case, we want to make the robot's movement following the line smoother and more uniform. Instead of the robot constantly rotating left or right, it follows the line and constantly changes its direction.

For this case, we define a concept called feedback. Feedback means considering the value of the system output (here, the value read by the sensor) and comparing it with the desired input value (here, the amount of reflection on the line boundary). By doing this, we are aware of the robot's status at virtually any time and can control it properly.

Let us first consider a robot that only matches itself on the line boundary. To design this robot, we need to read the measured value of the sensor at any time and compare it with our desired value. In general, instead of comparing these two values, their difference is used, which is also called the error signal (this signal indicates how much error our system currently has compared to the desired state). So, with the difference between the desired value and the measured value, we will have an understanding of the robot deviation from the line boundary.

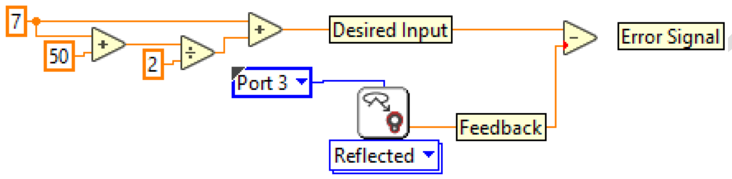


Figure 98 Generating an error signal with the desired input and sensor output

Now we want to apply this error signal to the robot motors so that by rotating the wheels, it adjusts itself to the line boundary. Our goal is to set the robot on the left border of the line. If the left wheel is connected to port B and the right wheel to port C, the robot can be adjusted to the left border of the line by placing the steering wheel on the motors, as shown below:

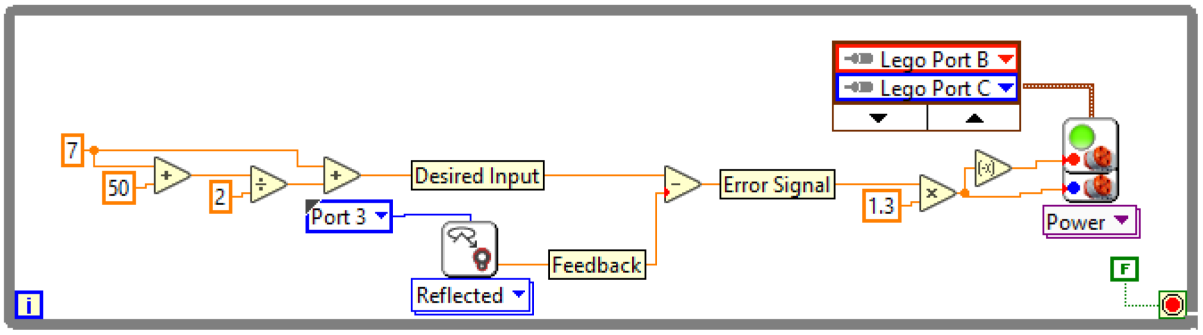


Figure 99 Rotation of wheels in proportion to the amount of error signal

In the above loop, first, according to what has been said before, the amount of reflection is calculated on the line boundary, then the difference with the reading value of the sensor is obtained, which forms the error signal. Here, because the error signal is not too large for the input to the motors, we use a coefficient (1.3) to magnify it, which in the control literature is called the proportional coefficient and is usually known by the symbol Kp. Next, we apply this magnified value to the motor on the right and its negative value to the motor on the left. The proportionality coefficient can be changed to get the best system behavior. But as the proportionality coefficient increases, the stability of the system decreases and the likelihood that the robot will be able to adapt to the boundary decreases. On the other hand, with this coefficient being low, the matching speed of the robot also decreases. As a result, the median for this coefficient must be obtained.

Next, we want the robot to follow the line. To do this, an interval around the desired value must be considered as the desired interval, that as long as the value read is in this interval, the robot will continue to move directly, and when it exits this interval, the robot will adjust itself. And then return to the interval.

If we take this desired interval at a distance of ± ten from the read value, it can be implemented in LabVIEW that if more than ten units are more than the desired value or less than 10 units is less than the desired value, and the robot will adjust itself. On the slow line, otherwise, the robot is in the desired range and can continue to move straight. So, in case, the robot is not in the desired range, we have to set the same robot as before:

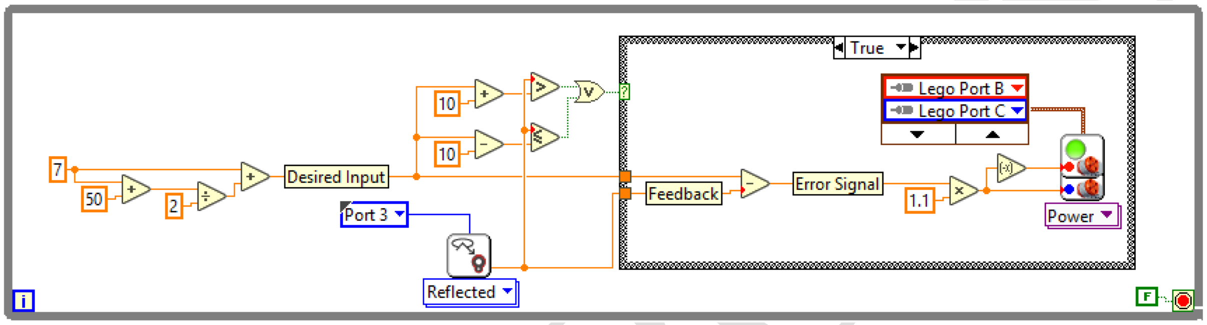


Figure 100 Adjusting the robot on the edge of the line with proportional control

And for the case that the robot is in the desired range, it is enough for the robot to continue to move directly:

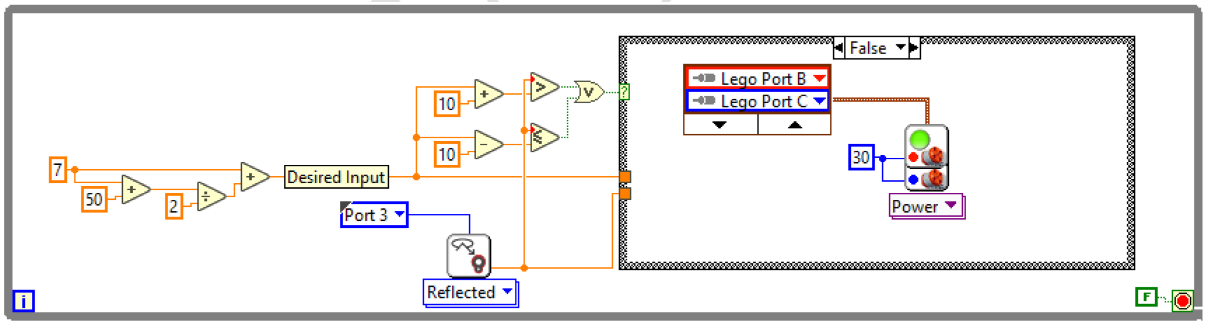
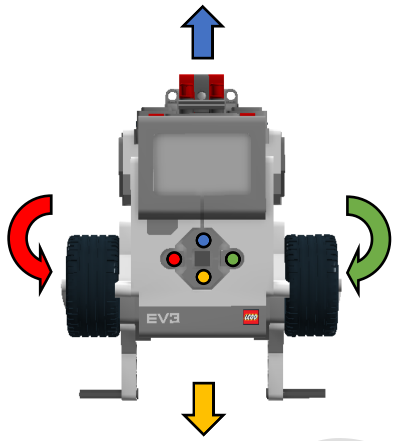


Figure 101 Direct movement of the robot when the robot is on the line boundary

This program is one of the programs that can be implemented for the line tracking robot. If several light sensors are used to detect the line, the robot will follow the line more accurately and faster.

CHAPTER 2 – PROBLEM SET

1. Design a program for the base robot that uses the buttons on the brick to move forward, backward, and left and right by pressing the up, down, left, and right keys, respectively.



Exercise 1 Move the robot with the keys on the brick

2. On a white surface, draw a complex path using black tape (detectable by a light sensor). Now design a linear chase robot to follow the path well. Find the best controller coefficient by trial and error so that the robot can follow the line best.

Exercise 2 Follow the line by adjusting the proportional coefficient

3. Build a combination of line tracking and wall tracking robots (install both light and ultrasonic sensors on your robot). Design a hybrid path that includes black lines on a white surface and a wall so that the robot can follow the path using line and wall pursuits. Now write a plan so that the robot can follow the combined path well (the robot must check whether there is a line or a wall, then follow the path using the written algorithm).

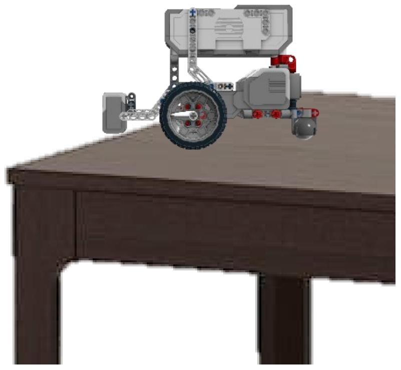
Exercise 3 Produce a beep sound proportional to the obstacle distance

4. Build and program a robot that measures the distance from the obstacle and emits a beep according to that distance.

Exercise 4 robot routing with the help of line and wall tracking

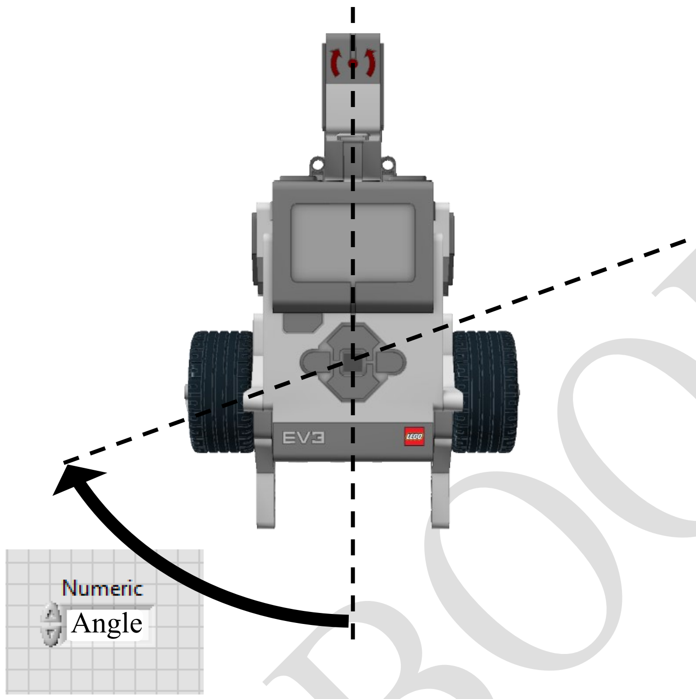
By changing the obstacle distance, the frequency of the generated horn changes accordingly.

5. Design a precipice detection robot that sits on a wall less table and, when it reaches the edge of the table, detects the precipice and changes its path so that it does not fall.



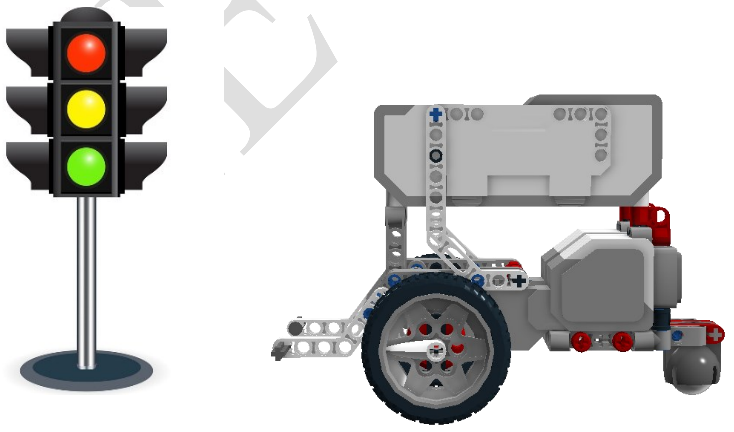
Exercise 5 Detect the abyss and prevent the robot from falling

6. Design a robot that rotates and is oriented in that direction by determining the position of the robot from the Front Panel.



Exercise 6 rotate the robot with the command value

7. Using the color sensor, design a robot that moves by displaying green and stops by displaying red.



Exercise 7 Move the robot with the green light and stop with the red light

Chapter 3
Advanced MINDSTORMS Programming

INTRODUCTION

In this chapter of this book, we intend to improve our programming skills by building and programming more advanced robots and bring the issues and challenges closer to the real goals. In each of the robots mentioned below, first, the purpose and task of the robot are mentioned, and then the process of making the robot is introduced and described and finally, its programming will be done.

Different methods can be used to program robots. As a result, the program described in this book is not the only program that can be designed for these robots. After building the robot structure, it is recommended to carefully determine the programming process and start designing the program, and then, refer to the book to review and validate your program to get all the points in this and take the complete advantage of the section.

Color Sorter Robot

In this section, our goal is to build a robot that identifies colored blocks and categorizes each block according to its color. First, we will build this robot according to the instructions in the appendix at the end of the book.

After completing the robot construction process, it is time to program it.

To get more familiar with this robot, we will introduce the parts and components of this robot:

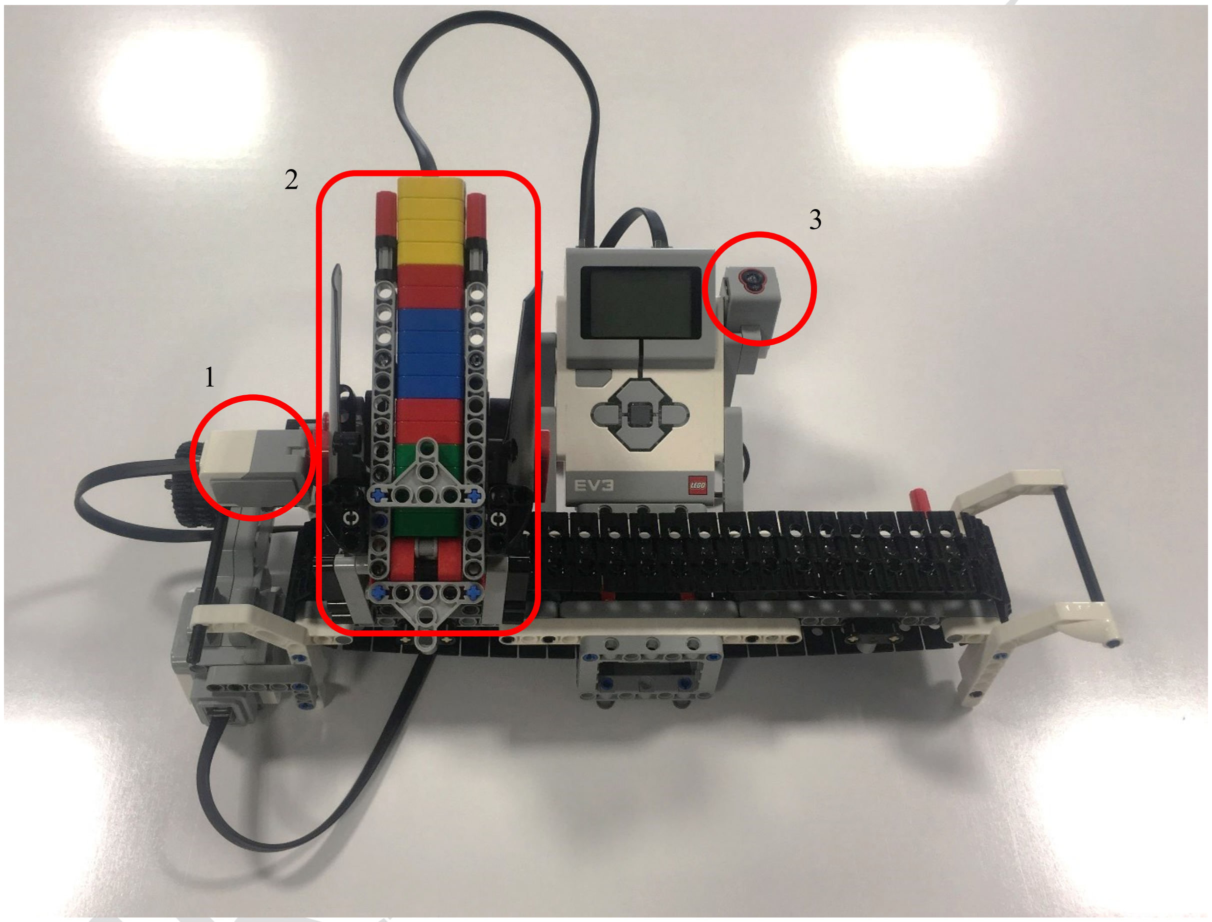


Figure 1 Overview and main parts of the color sorter robot

1. Contact sensor: Used to return the car to its original position.
2. Cartridge: Used to transport and unload blocks, unloaded by the motor connected to port A. The cartridge is moved on the belt by the motor connected to port D.
3. Light / Color Sensor: Used to detect the color of blocks. It is necessary to divide the steps of robot operations into several parts: identification, decision making, movement. First, the blocks need to be identified by the robot. As a result, the robot must recognize and store each color block before placing it inside the robot's tank. A light/color sensor is used to detect color. So, the way the robot works so far is that each block is first held in front of the robot color sensor to detect the color, then this block is placed inside the robot tank. In

this way, the colored blocks are identified in order and placed in the robot tank in the same way.

To start programming this robot, we do the identification step programming. For this purpose, we use arrays to store the color of each block. Since the number of blocks may change, we also receive the number of blocks from the user in the Front Panel window. For this purpose, we use Numeric Control:

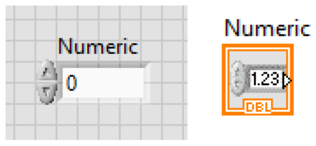


Figure 2 Creating a numeric input

The value entered in this numeric input will indicate the number of blocks.

In the first step, we are going to identify the blocks and store their color in the identified order in a variable. Since the identification and storage process is to be repeated in the number of blocks, a *for* loop must be used to do this. According to what is said in the topic of color / light sensor, we form the *for* loop as follows:

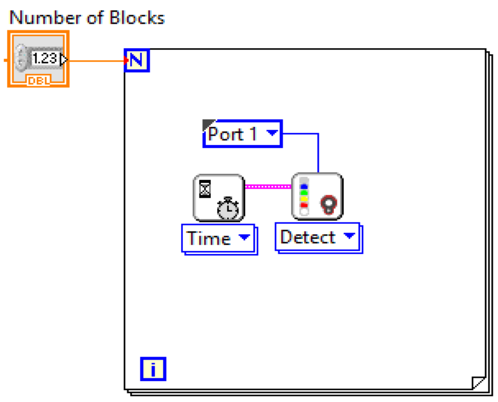


Figure 3 Read the color of the blocks in the specified number

In the next step, the values read by the color/light sensor must be stored in a variable so that these values can be accessed later. As a result, by using an array in the Front Panel window and placing a *Numeric Indicator* in it, the sensor output can be stored in this array:

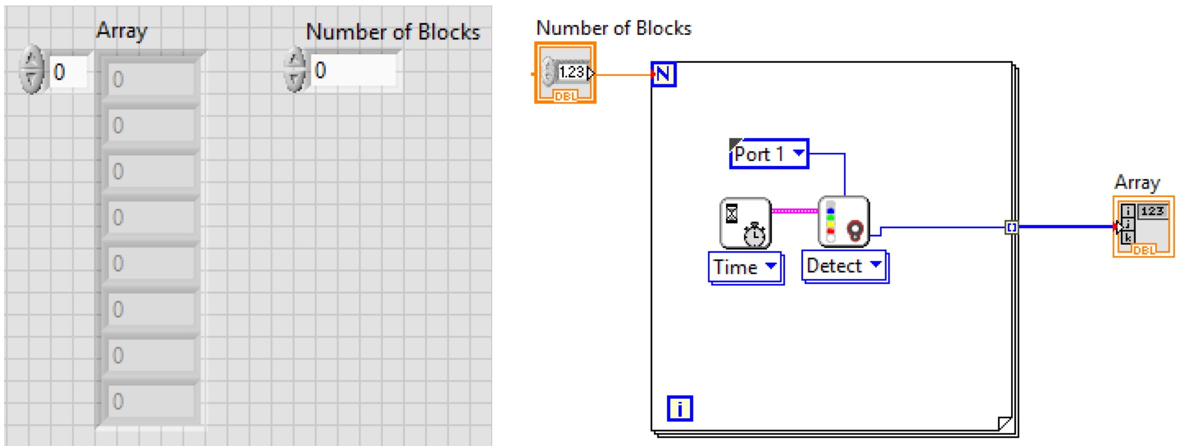


Figure 4 Storing the color sensor output in an array

The identification step is complete. However, since the identification process is performed without interruption, the written program is not practical because the program performs the identification and completes the number of blocks simultaneously. For the program to be useful, we must use the screen and speakers to show signs and symbols to the user so that the user and the program can perform the identification process, respectively. With a little patience and attention to what has been said, the program can be upgraded as follows:

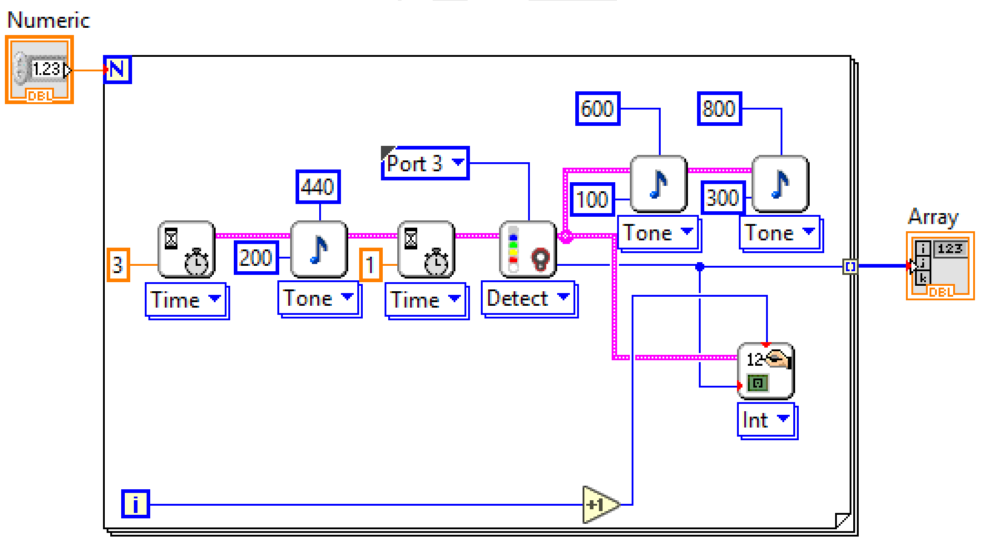


Figure 5 Completing the program for reading and saving the color of blocks

In this case, first, the break stops for 3 seconds, then plays the beep for 200 milliseconds and stops again for 1 second to place the user in front of the color sensor. After recognizing the color of the block, the break emits a sound that means the recognition is completed. At the same time as the detection, the color code of the detected block is displayed on the brick's screen. This operation is performed until the number of blocks is identified and the color code is stored in an array.

In the next step, we must place the blocks in the appropriate tanks in the identified order. To do this, we must identify two items. The first case is the amount of movement of the cartridge with the help of belt movement. Considering that the length of the cartridge path is limited and the number of colors of the blocks is 4, the cartridge's movement can be determined. The second case is motor A's rotation to empty a block from inside the cartridge tank.

To determine the length of the cartridge, the D motor can be rotated a few complete turns with a simple program to determine how many laps the cartridge travels. If we use this method, we find that by rotating 540 degrees (1.5 rpm), the cartridge travels all the way to the belt. As a result, we have to divide this path into 4 parts so that when separating the colored blocks, each of them is emptied in a specific position.

To determine the amount of rotation of motor A to empty each block, if we pay attention to the mechanism of this robot, we will notice that with each complete rotation of this motor, one block is emptied from the cartridge tank. So, for each unloading of the block, it is necessary to move motor A a complete revolution (360 degrees).

In the separation step, it is necessary first to read the color stored in the constructed array. Then, according to the color code, call the cartridge at a suitable distance from the sensor, empty the colored block, and return to the original location to empty the block. Prepare the next color.

The important point here is that the program must first identify and store the color and then move on to the separation process. For this purpose, as mentioned, we use the Flat Sequence structure. In this case, first, we put the process of color recognition and storage in the first frame and then we put the separation process in the next frame so that these two processes can be performed sequentially and without interfering with each other. So, we put the program that has been designed to identify and store color in a *Flat Sequence* frame:

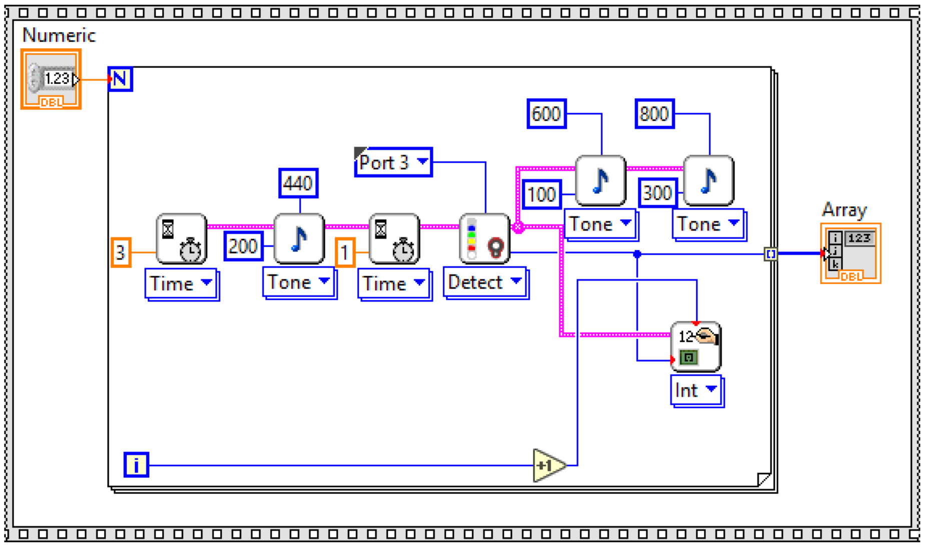


Figure 6 Placing the program in the first frame of the Case Structure

To build the separation process, a frame must be added to the Flat Sequence structure according to what has been said, and this process must be designed in the new frame:

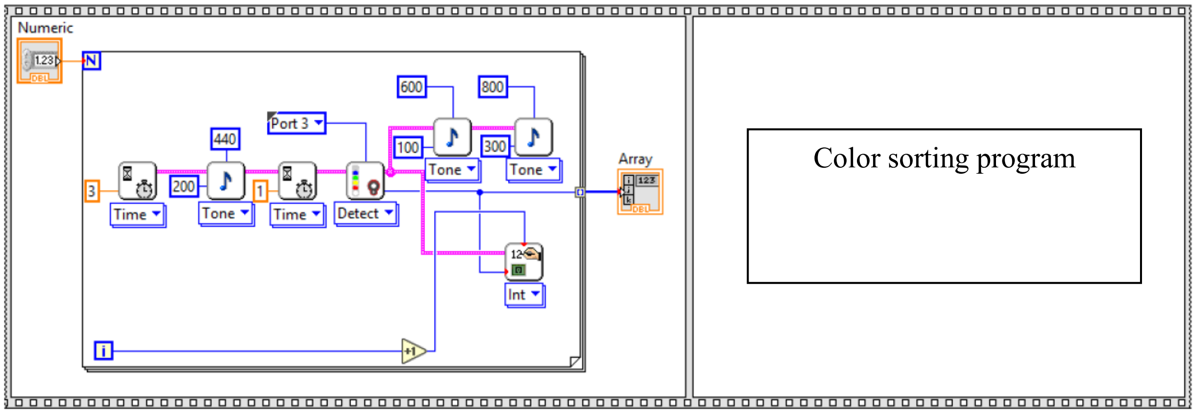


Figure 7 Creating a new frame to design a block separation program

As we know, the separation process is an iterative process that must be repeated in the number of blocks read. As a result, the best way is to use the For loop to repeat the separation operation according to the number of blocks read:

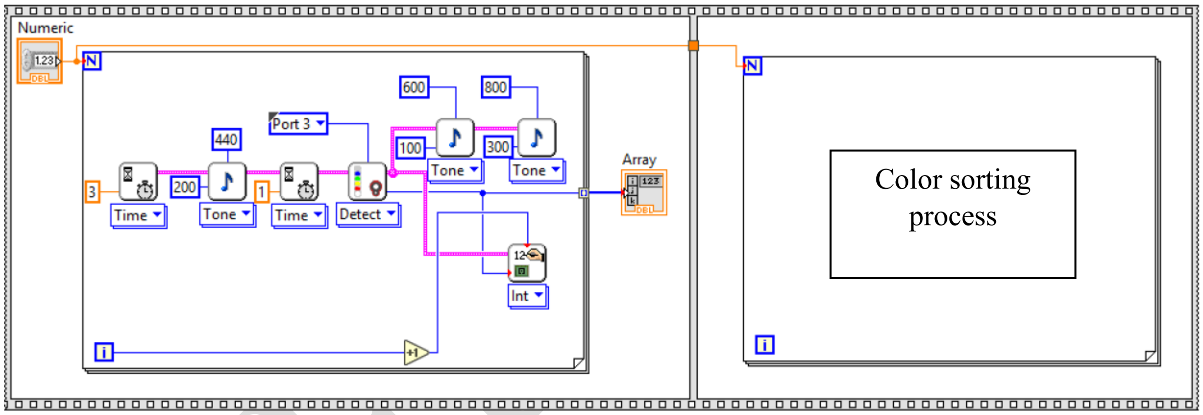


Figure 8 Repeat the separation operation in the second frame

In most automated and mechanized systems, microswitches are often used to return the machine or robot to its original state (the reference and known mode). Although most robots or systems use stepper motors (motors that have a certain rotation angle) and servo motors (motors that have an encoder or potentiometer to measure angle and speed), because physical systems mostly have small errors Microswitches may or may not be performed as expected when the machine is operating. The microswitch is a simple switch that sits on the frame and fixed parts of the device and sends a signal to the microcontroller when it comes in contact with the device's moving parts. In the color separation robot, the contact sensor is used as a microswitch.

In order to be able to perform the subsequent movements of the cartridge correctly and accurately after each movement of the cartridge, it is necessary to move it to the initial position by moving the cartridge towards the contact sensor until the contact sensor is activated. By doing this, the small errors that exist in the measurement or movement of the engine are eliminated, and our robot shows flawless operation.

So, the separation process is that first the color code read for the desired block is read from the stored array. According to the color of the block, move the cart to the appropriate value, and after reaching the appropriate location, the block from the inside the tank of the cartridge is emptied. Finally, we move the cartridge to its original position until it touches the contact sensor and stops it.

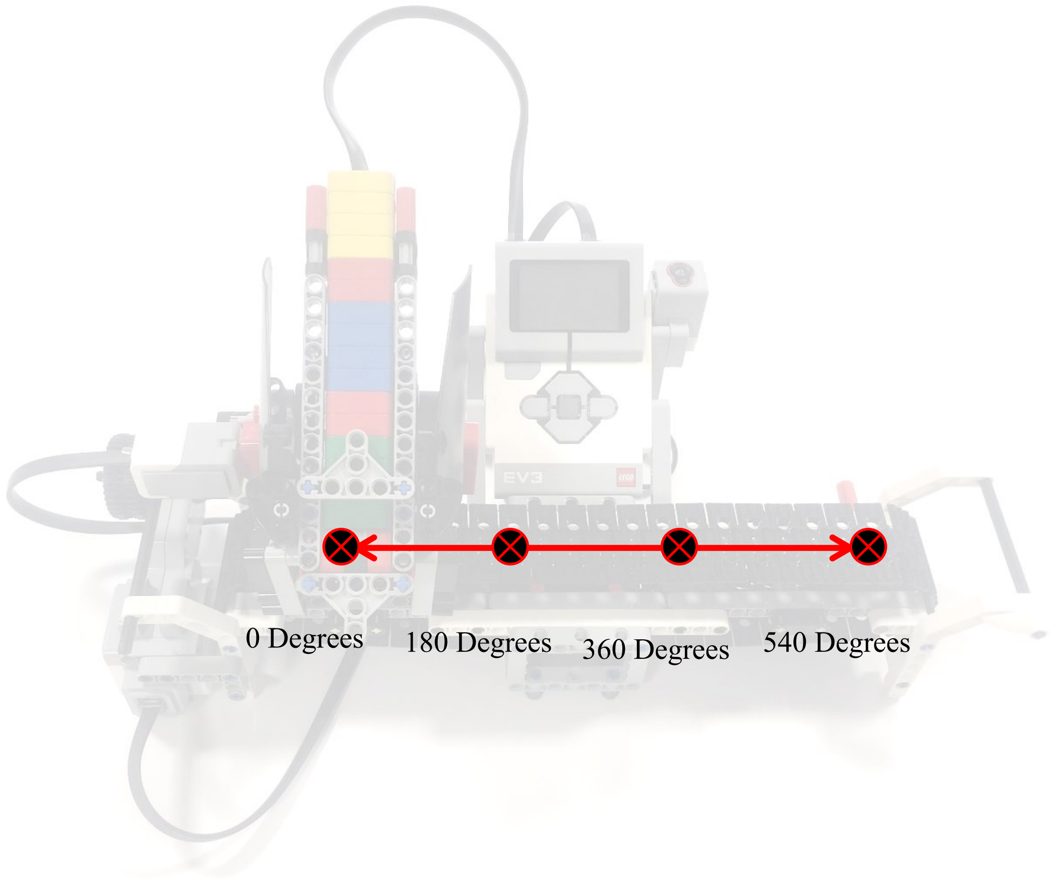


Figure 9 The amount of rotation required to separate the blocks into 4 different containers

The Local Variable block can be used to access data stored in a variable or array. To create a local variable block for a variable or array, right-click on the variable or array, select *Create* → *Local Variable,* and then place the created block in the appropriate space. Note that the created block is set to input mode. To make this block output mode, right-click on it and select *Change to Read*. In this case, by connecting this block to the desired nodes, the data inside the variable or array can be used:

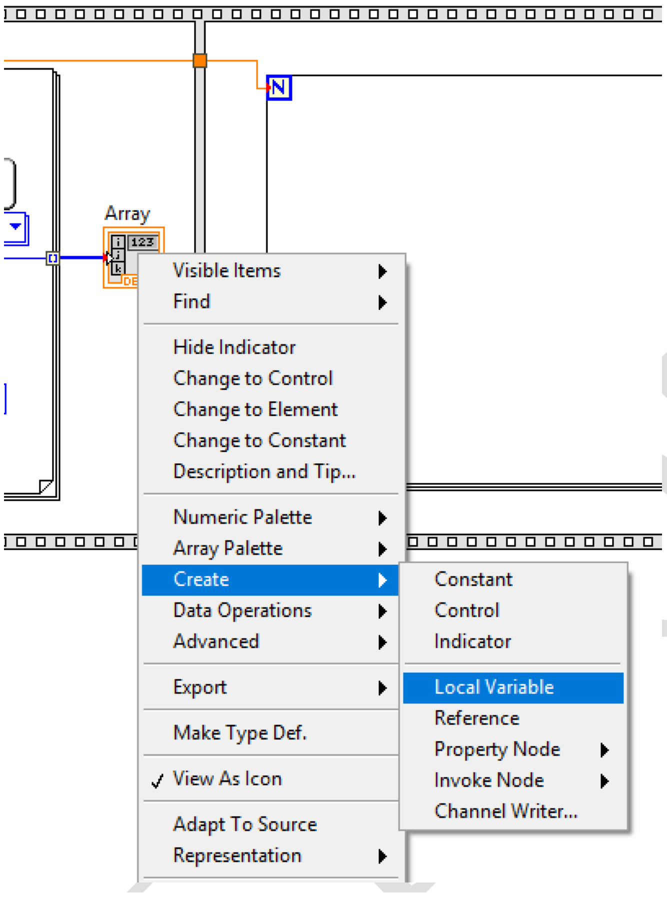


Figure 10 Creating a local variable

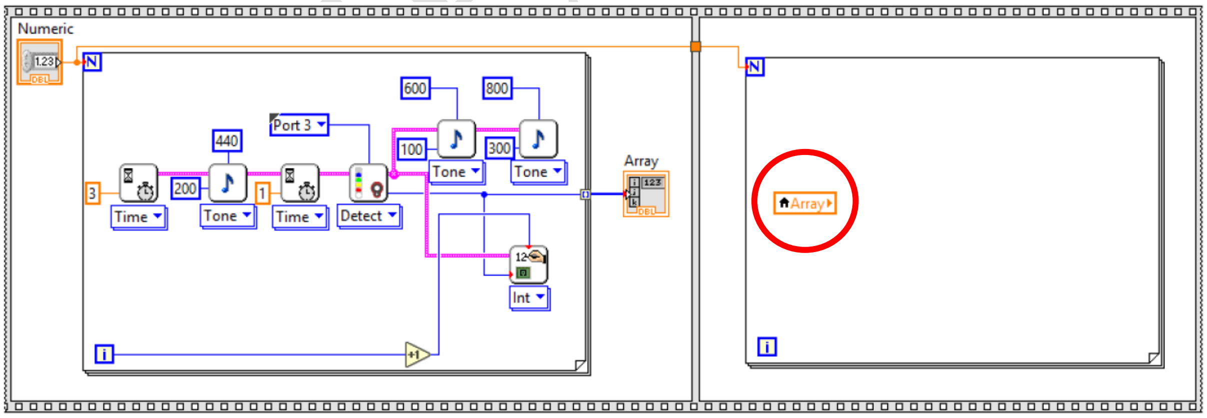


Figure 11 Creating a local variable terminal

Now that we have access to the values inside the array, we can read the order of the colored blocks by reading them in order. As mentioned in the previous sections, using the *Index Array* block, the value of each drive can be extracted by specifying the index. So, to access the different values stored in the array, we use this block as follows:

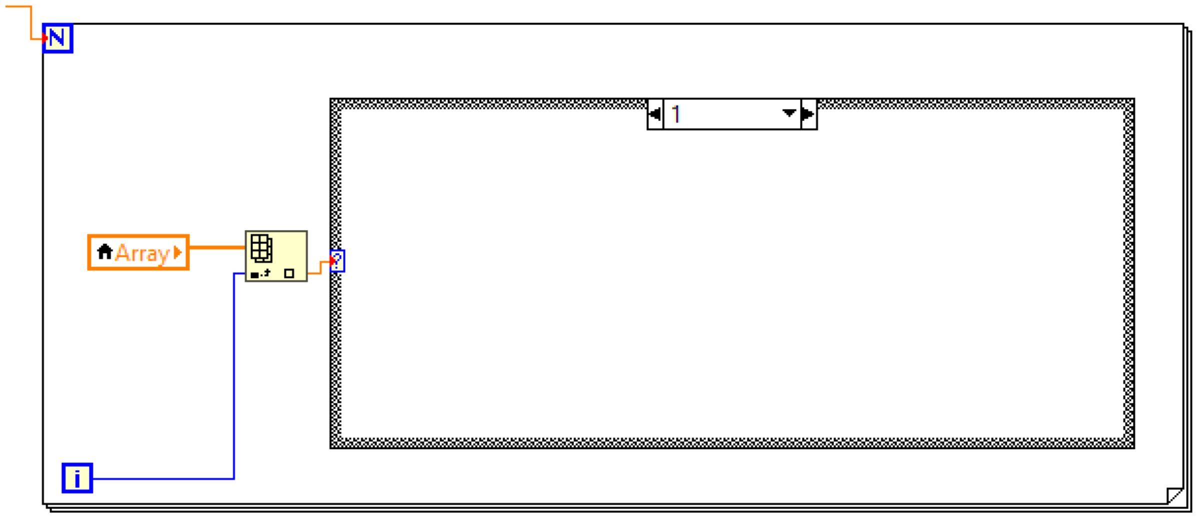


Figure 12 Read the color codes stored in the array

Since each code represents a specific color, we have four colors, blue, green, yellow and red, which have codes 2, 3, 4 and 5, respectively. So, we put the items in the *Case Structure* in proportion to these numbers as follows:

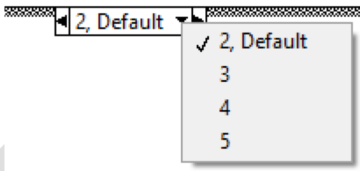


Figure 13 Creating different cases for recognizing and separating colored blocks

Inside each case, the operation of moving the cart to the desired point, emptying the block and returning the cart to its original location must be performed. The only difference between these operations for different colors (different cases) is the D motor's rotation to move the cartridge, which was described in the previous section. As a result, the program in the *Case Structure* section is designed as follows:

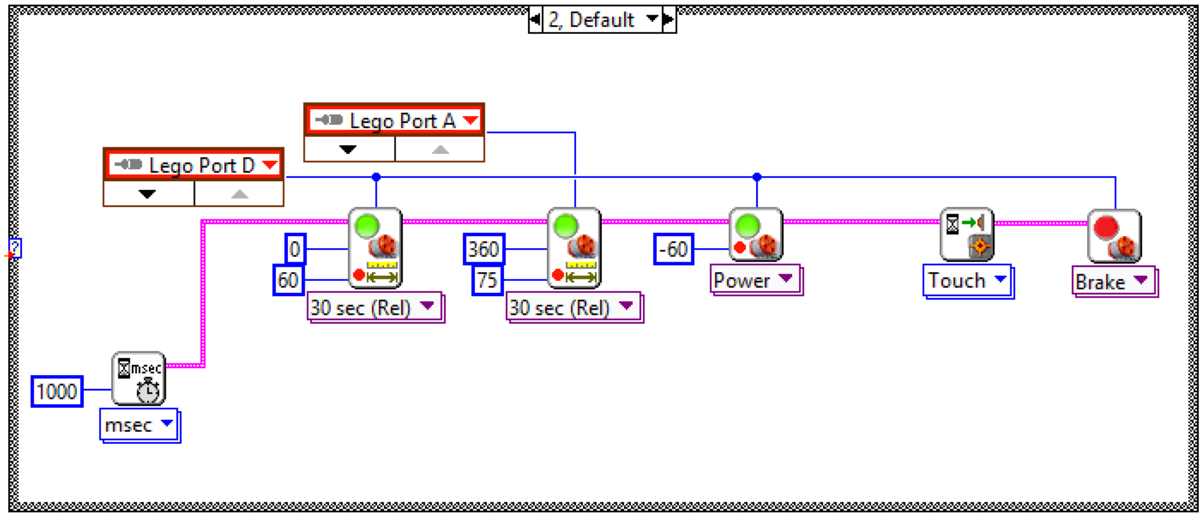


Figure 14 Program design for moving the cartridge to the desired point

For better performance of this robot, we have set a delay of 1 second at the beginning of each separation operation. For other cases, only the amount of motor D rotation changes. For others, the program changes as follows:

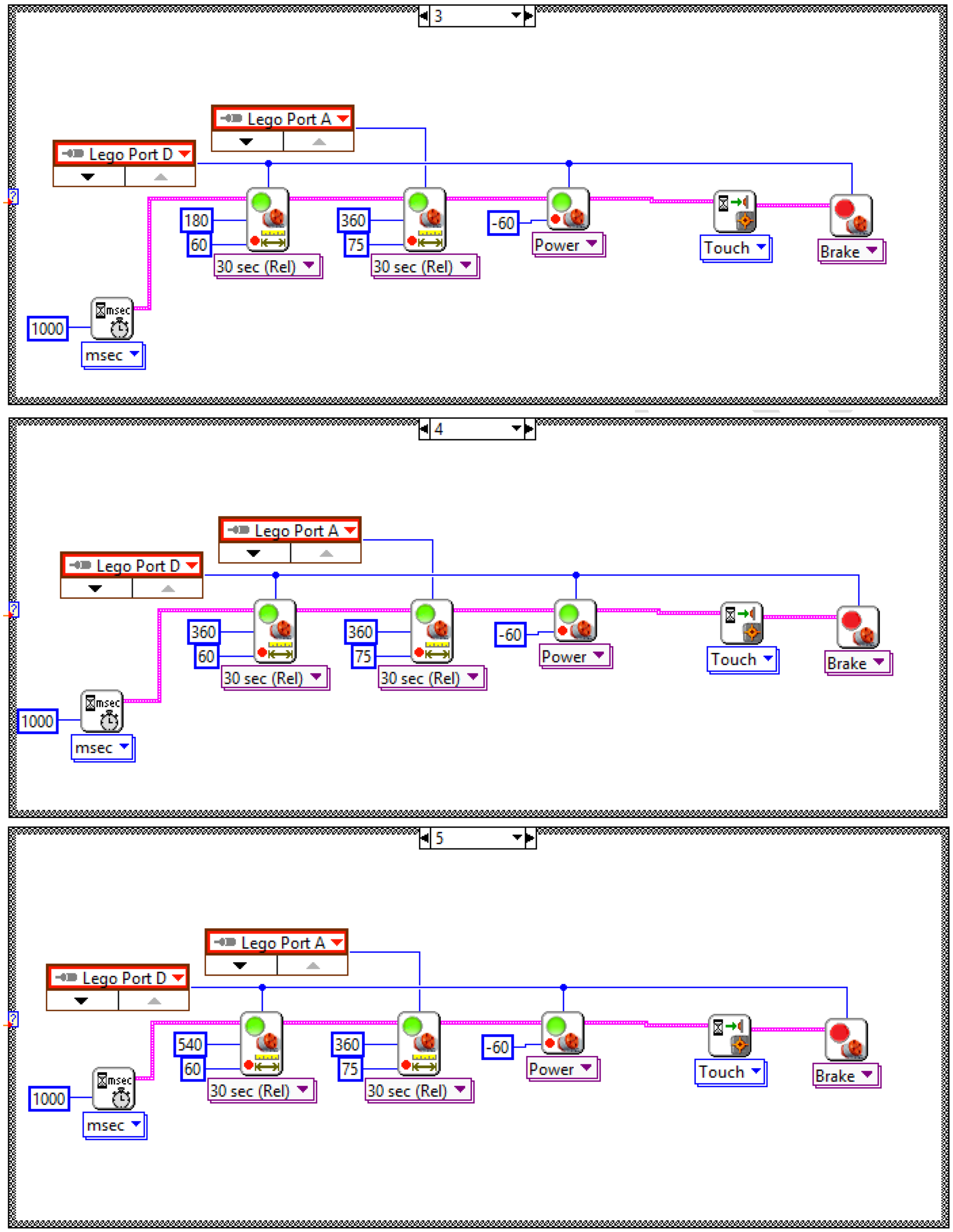


Figure 15 Changing the amount of rotation of the D motor according to the color read in each case

Now the design of the robot program is completed, and the robot first recognizes the colored blocks and then separates them in their proper place. The final program is as follows:

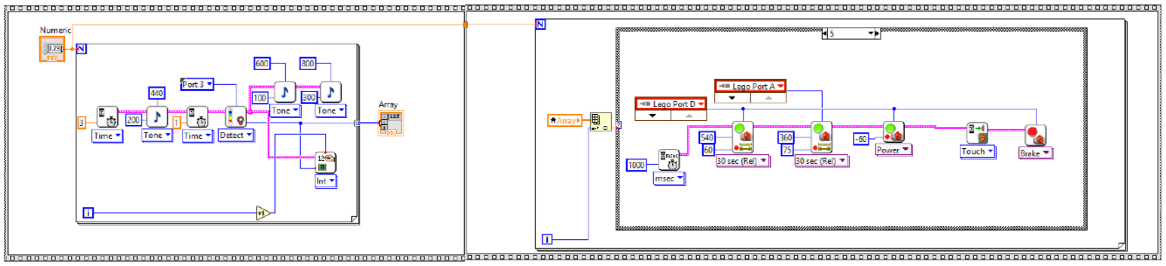


Figure 16 Complete program of color sorter robot

Note that each of the steps mentioned above can be done in different ways, and it is not necessary to be done as described, and on the other hand, by changing the program and even the structure of the robot, its performance can be improved.

Color Sorter Robot – Problem Set

1. To further practice programming, change the robot so that the number of blocks is not known in advance, and with the push of a button, the detection operation is stopped, and the separation operation begins.
2. Design the robot program so that when separating the blocks, it does not return to its original position each time and goes to the next block position. Is the robot still accurate enough to perform the separation operation?
3. Change the structure of the robot so that it first puts all the blocks in the tank and then reads the color of the last block before separating the blocks and empties it in the appropriate position.

ROBOT ARM

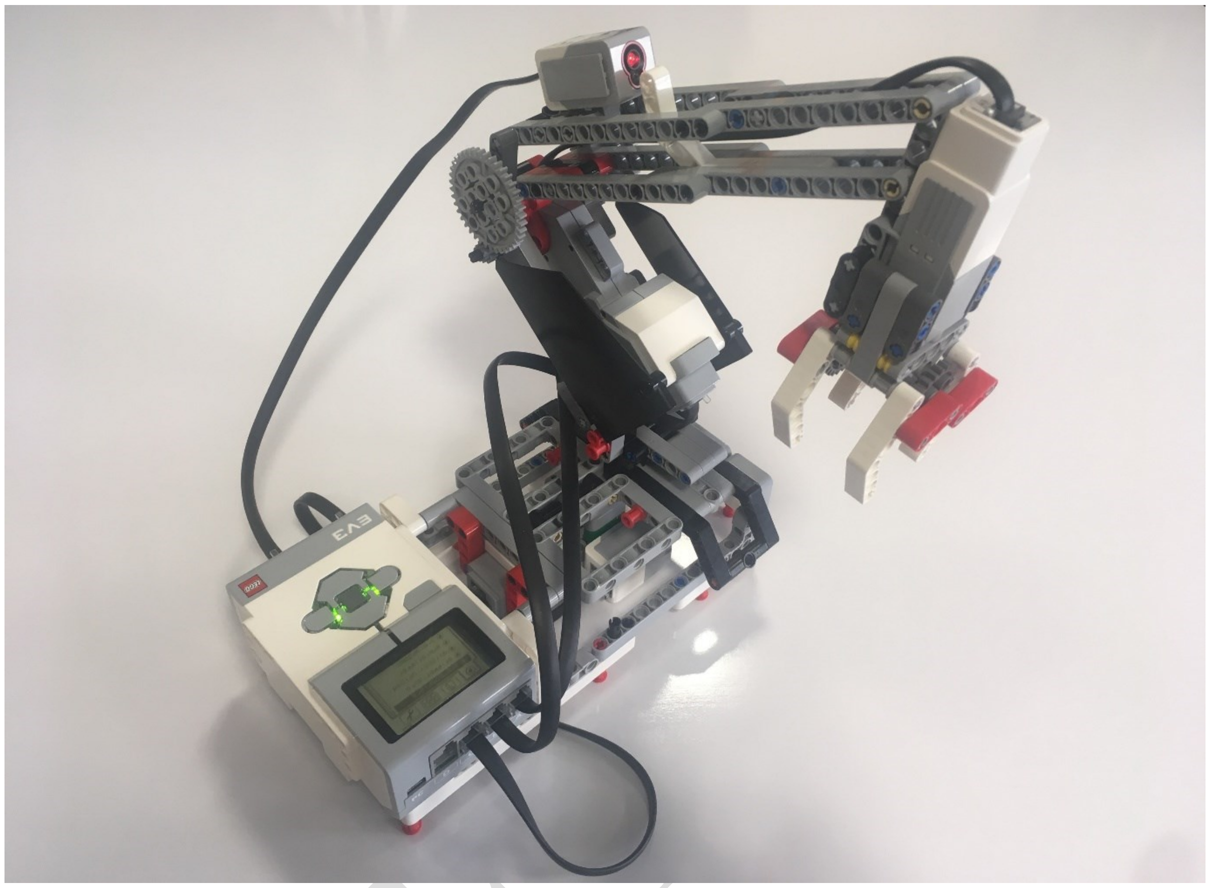


Figure 17 Robotic arm

One of the most famous and widely used machines in the robotics industry and the world is the robotic arm. Robotic arms are made up of several limbs, which are moved and controlled by hinged or sliding joints relative to each other using various actuators. Also, various sensors are used to control the movement of the arms, and their members do not collide with each other and cause damage.

Robotic arms can consist of two types of series or parallel mechanisms. The series mechanism is a mechanism in which the endpoint of the robot (the toe of the robotic arm) is connected to the ground from only one path. But a parallel mechanism controls the robot's endpoint from multiple paths.

In this section, we want to build and program a robotic arm with two degrees of freedom. The degree of freedom of a mechanism is the number of independent coordinates to describe the position of all points in that mechanism. In other words, the degree of freedom refers to the number of different movements that a mechanism performs along with different directions and around different axes.

The arm discussed in this section is a series of mechanisms and has two degrees of freedom. There is also a *gripper* at the end that is used to lift and place objects. As a result, 3 motors or actuators are used in this robot. Also, two microswitches are used to control and stop each member.

As mentioned in the section on the color separation robot, a microswitch is used to return the robot to its original position. In this robotic arm, we use two sensors as microswitches: a contact sensor (such as a color separation robot) and a light sensor. In the light sensor, the amount of proximity to the initial position can be obtained according to the light received from the sensor. It is also possible to detect approaching the initial position by color recognition.

First, build the robotic arm according to the instructions in the appendix at the end of the book.

As you can see during the construction of this robot, the two contact and light sensors are placed on it in such a way as to detect the position of the robot in a certain position and prevent it from moving further. In the programming process, we use these two sensors to restrict the robot's movements, and at the beginning, we return the robot to its original position.

The first step is to determine how each member of the robot will reach the initial conditions. If we show the coordinate axis corresponding to the arm as shown below, our robot rotates around the Z and Y directions. If we want to be more precise, motor C rotates around the Z-axis, and motor B rotates around the Y-axis.

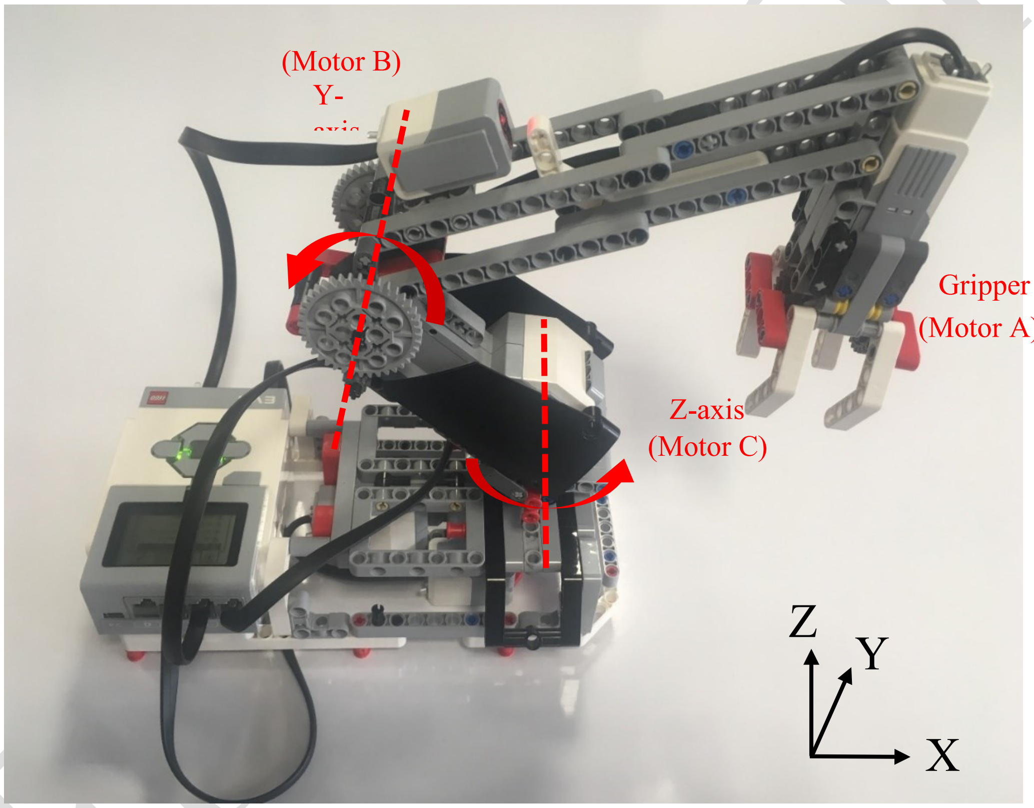


Figure 18 Robotic arm movement on axes

By paying attention to the robot's structure, by rotating the C motor in a particular position, the contact sensor is compressed, which indicates that the motor has reached the initial position. Also, by rotating motor A, the link connected to it approaches the light sensor, which can be used to determine the initial position of this motor by using the sensor output.

So, in the first step, we design a program to return the robot to its original position and restrict its movements by sensors.

Controlling Z-axis Rotation

To start programming this robot, we start with engine C. We know that with the rotation of the C motor, the second link and the robot gripper rotate around the z-axis. In this case, by rotating robot C, the tab in the path of the contact sensor will cause the sensor button to be pressed, and the motor will stop moving. For the robot to always start from this position, we use the information obtained from the contact sensor. Also, if the robot continues to move after pressing the contact sensor, it will cause damage to the engine and other parts. Therefore, it is necessary to stop the engine if the contact sensor is compressed and not to allow further movement.

To return the robot to its original position, we must move the C motor in the right direction to compress the contact sensor. After the contact sensor is compressed, we stop the engine. By running this part of the program at the beginning of the robot start-up process, we return the robot to its original position from any position.

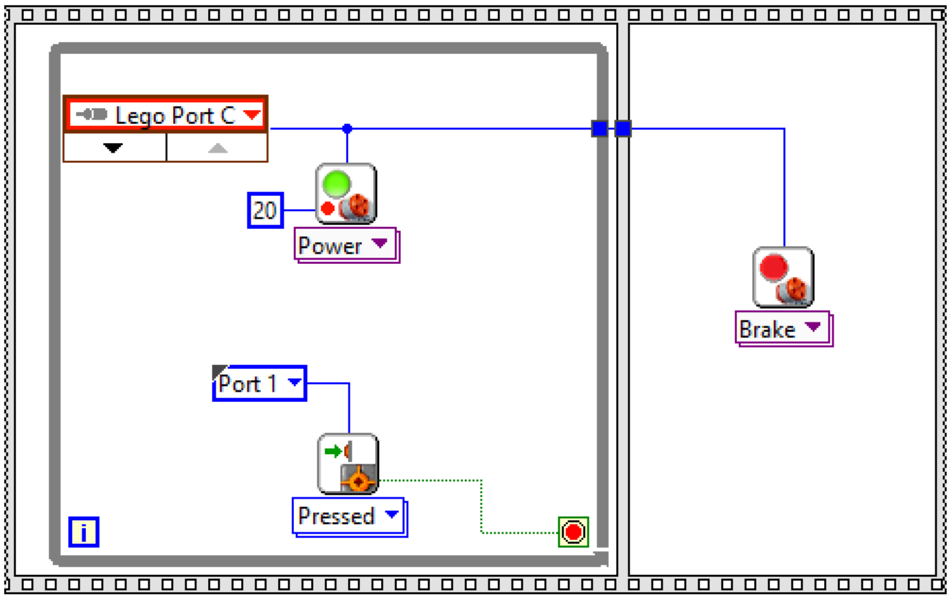


Figure 19 Returning motor C to its original position

Now we come to the part of the C motor control. Suppose we use the keys on the break to move the robot. In this case, to control the C motor, two keys are needed to move it in two directions. For example, we use the left and right keys to move this engine. In this case, according to the programs written in the previous sections, the following two keys can be used to move the C motor as follows.

If none of the left and right buttons are pressed, the C motor will remain stationary:

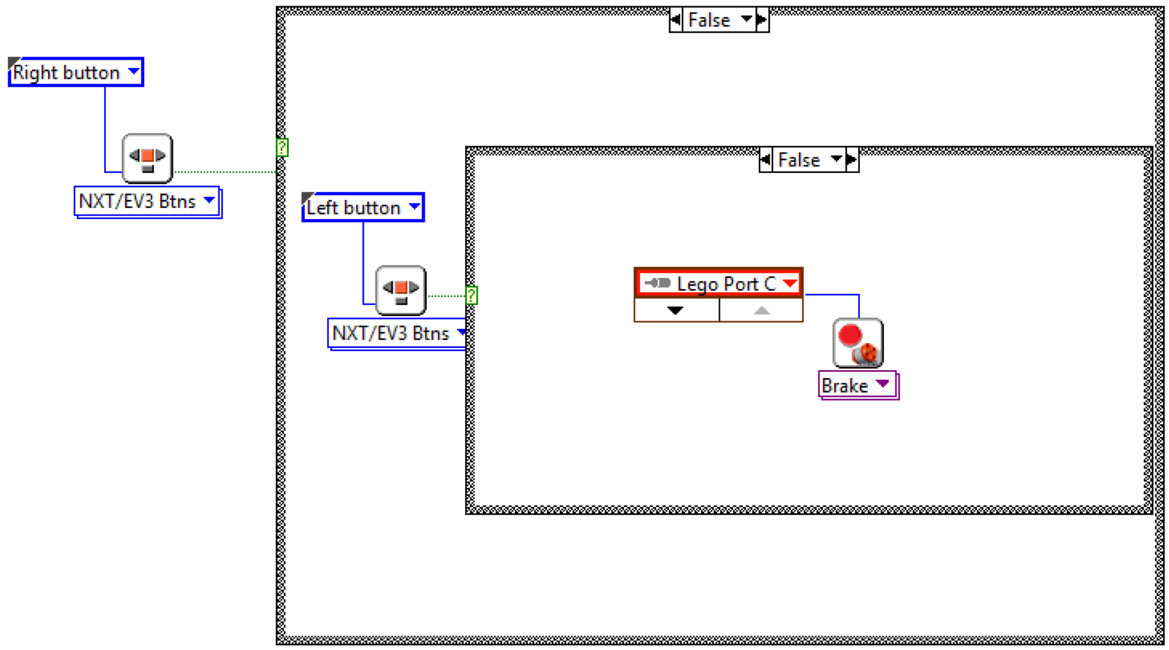


Figure 20 Moving the C motor with the keys on the brick

If the right key is pressed, if the call sensor is not activated (motor C has not reached the end of its working space), the motor will move clockwise:

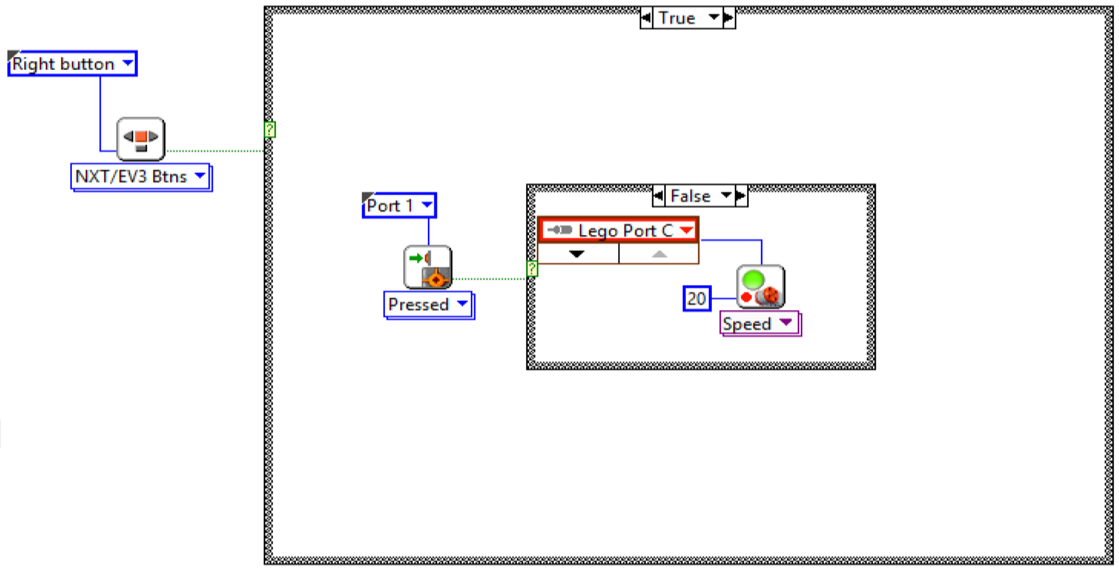


Figure 21 Moving the C motor towards the touch sensor by pressing the right key

And if the contact sensor is pressed, the C motor will stop and will no longer move in this direction:

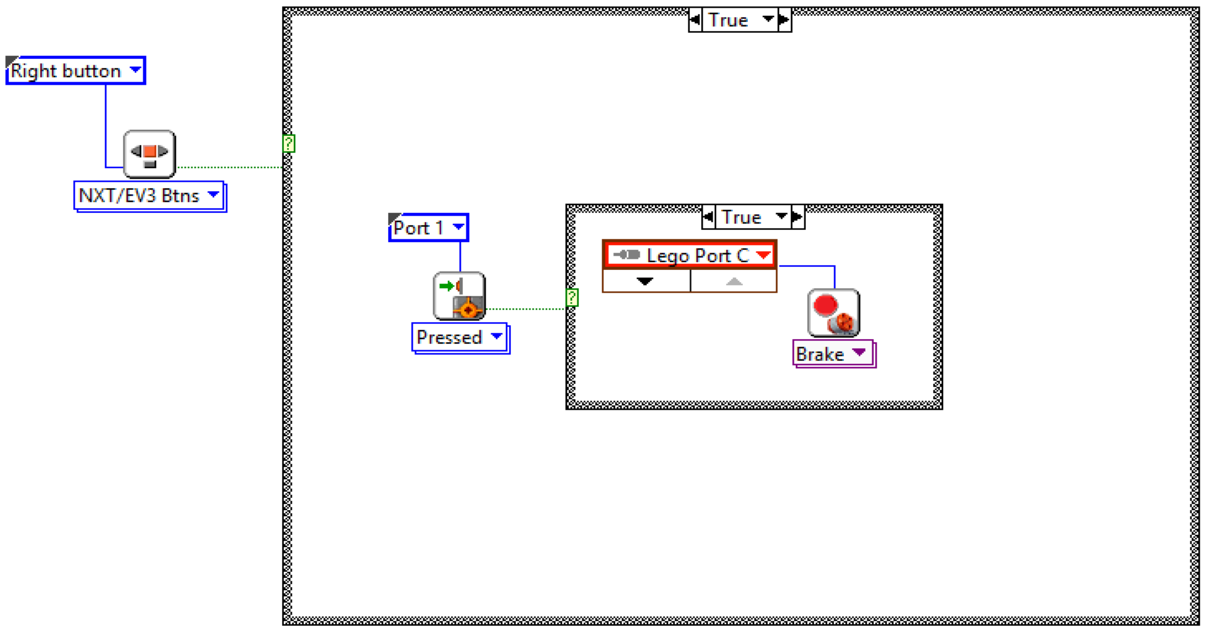


Figure 22 Motor C stopped when the touch sensor is pressed

However, if the left button is pressed, the C motor will start moving counterclockwise. There is a point in this that needs to be examined better than carefully. As is clear from the robot's structure, the C motor can move away from the contact sensor to some extent, so if the motor rotates too much, it may engage and damage the robot components and connections. If we assume that the absolute limit of the robot's rotation is equal to half the rotation around the z-axis, we must find the amount of rotation of the C motor to reach this position. Using the encoder in each engine, its position can be obtained. For this purpose, we use the Motor Status block. Also, in order to always refer to the initial position of the C motor (the position in which the contact sensor is pressed) as a reference, after bringing the C motor to the initial position, reset the encoder using the Reset Encoders block to zero this position. Show. In this case, by rotating motor C, we can get the distance from the initial position by reading the value of the encoder.

If we consider the final limit of rotation of motor C in the counterclockwise direction to be 600 degrees (approximately half a turn of the movement space of the robot arm), by pressing the left button until the number read from the encoder exceeds 600 degrees, motor C in the direction Moves counterclockwise:

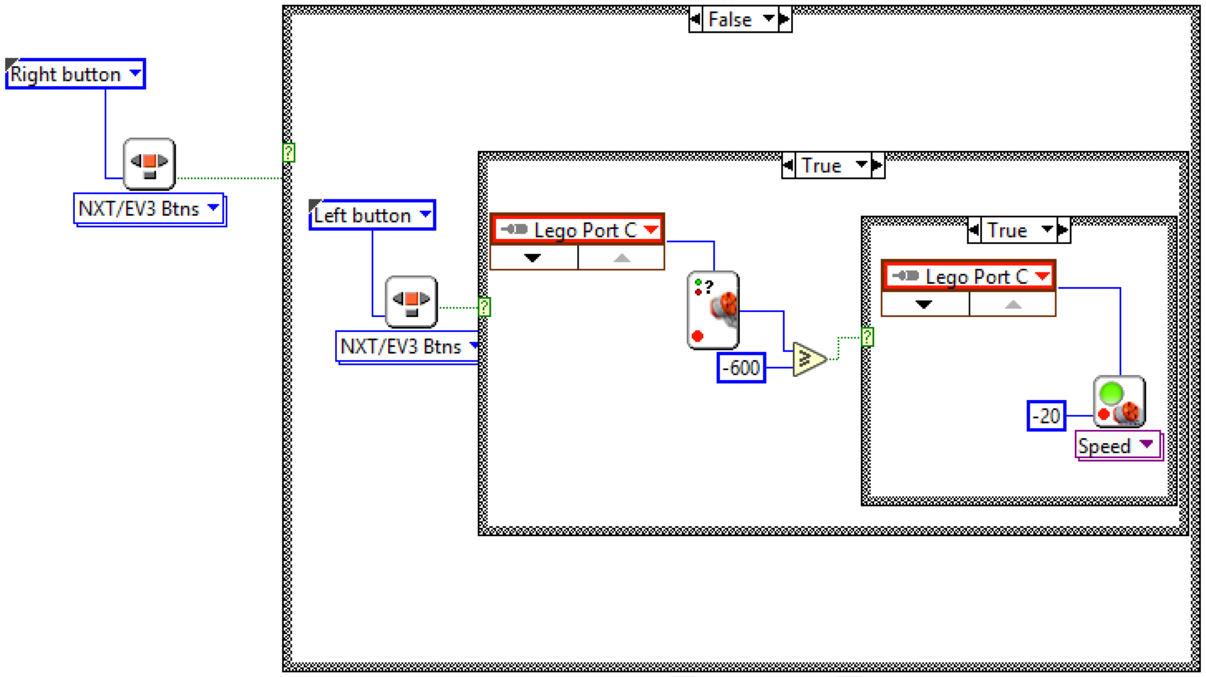


Figure 23 Moving the C motor to 600 degrees away from the touch sensor

If the value of the engine encoder exceeds 600 degrees, the engine will stop:

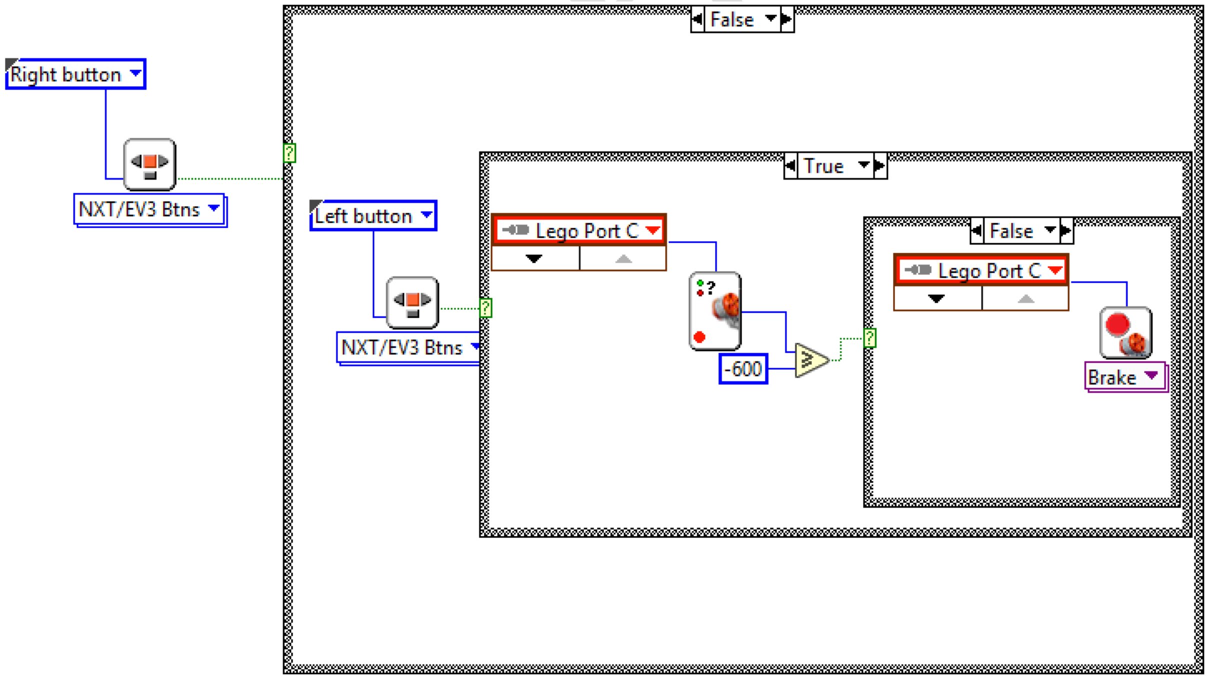


Figure 24 Stopping the C motor if it reaches 600 degrees

Now that the program control section is complete, we need to put it in an infinite loop and continue with the previous program, which returns the C engine to its original position. Also do not forget that after reaching the initial position, it is necessary to reset the value of the engine encoder:

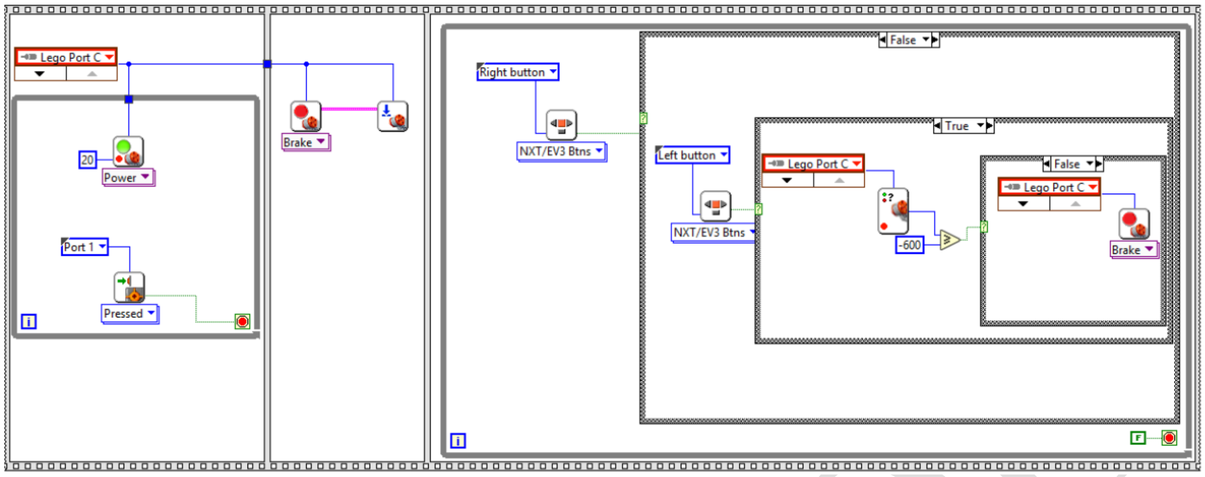


Figure 25 Completing the program by placing it in the infinite loop

If the above conditions are observed by running this program, the motor is first returned to the initial position and then by pressing the left and right buttons, and the C motor can be controlled.

Do not forget to save this program with the appropriate name and in the specified path to use this program in the future. For example, here, we save this program as *arm-z-axi.vi* in a specific place.

Controlling Y-axis Rotation

The rotation around the Y-axis is somewhat similar to the rotation around the Z-axis, except that in this axis, instead of the contact sensor, the light sensor is used as a microswitch. As before, we first design the Y-axis motor motion program, i.e., the B-motor to the initial position, and then control it.

To bring the B motor to the initial position, the motor must be moved in the appropriate direction until the tab in front of the light sensor approaches the sensor eye, and the sensor can see it. To do this, use the Reflected mode in the light sensor to read the proximity of the tab to the eye. With a bit of research, it can be seen that whenever the tongue is close to the eye, the value read in Reflected mode will be approximately more significant than 35. As a result, we have to bring the engine close to the sensor so that the real value exceeds 35.

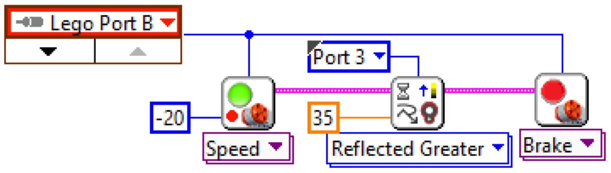


Figure 26 Reading the amount of light sensor to stop the motor B

By running this program, motor B moves the arm upwards and stops it when it reaches its maximum (the tab approaches the light sensor). Note that if the engine speed is high, it may cause damage or malfunction of this program, so the engine speed should not be selected as a large number. Here we have considered the engine speed to be 20 and in the negative direction. Also, note that the light sensor is connected to port 3, and the port number connected to the light sensor must be specified in the Sensor block.

In the next step, we want to write a program that can control the movement of motor B with the up and down keys on the break. The physical limitations of the arm must also be considered. Depending on the orientation of the break relative to the robot, we select the button at the bottom of the block to move motor B towards the light sensor and the button at the top of the break to move the reverse. With two Case Structures together, we can create this behavior. For the case where the down button is pressed, and the arm is not yet close to the light sensor, we have:

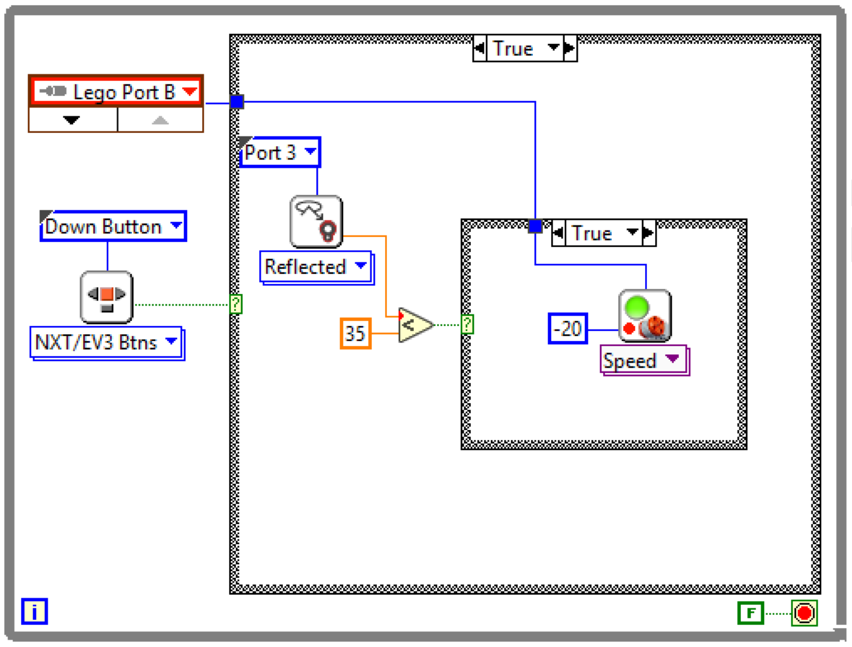


Figure 27 Moving motor B towards the light sensor

If the arm reaches the end of its movement (the tab approaches the light sensor), then motor B must be stopped:

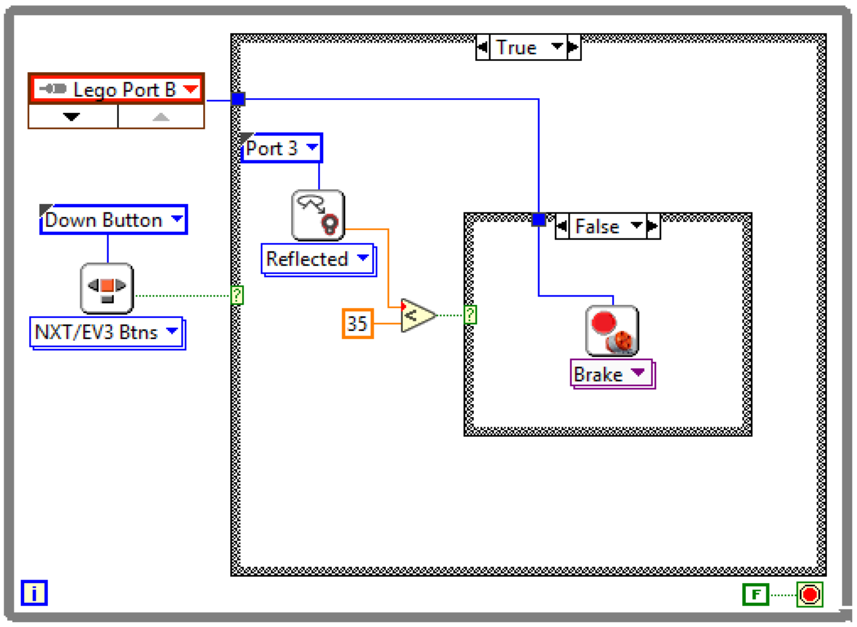


Figure 28 Engine B stops when it has reached the light sensor

For cases where the up button is pressed (as a result, the down button is not pressed), the engine moves in the positive direction:

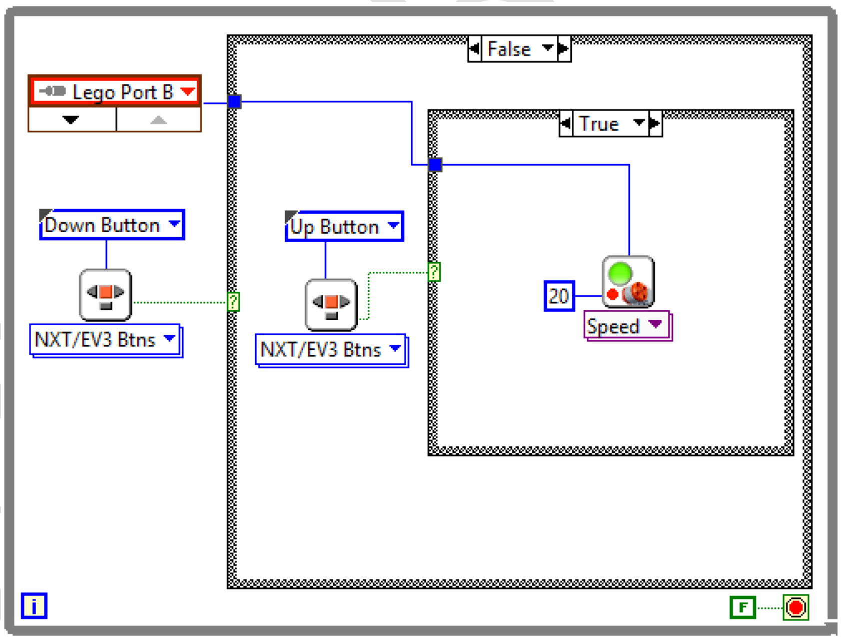


Figure 29 Motor B moving away from the light sensor

The engine must be stopped if none of the up and down buttons are pressed:

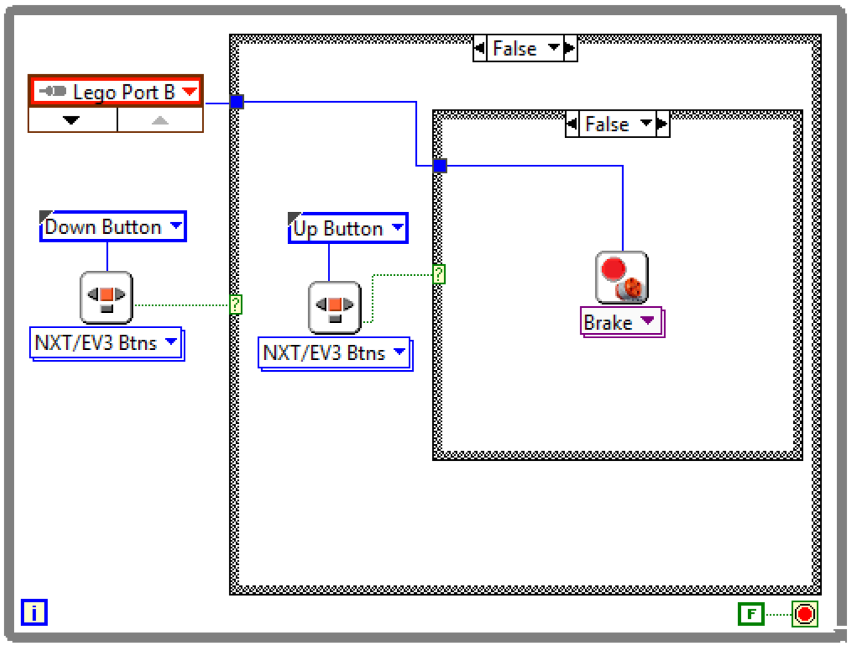


Figure 30 Motor B stopped if the up and down keys are not pressed

Finally, to control and move the Y-axis, we use *Flat Sequence* and put the first and second parts of the program together, respectively:

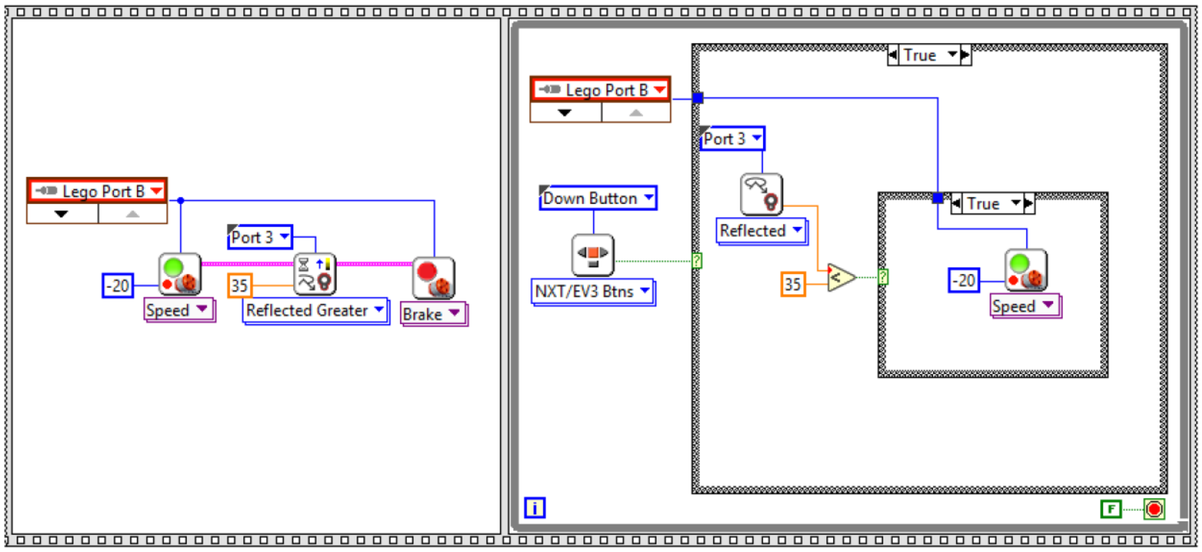


Figure 31 Completion of motor movement B related to the movement of the Y axis

Do not forget to save this file with the appropriate name (for example, arm-y-axis) for later use.

Gripper Motion

Now that the arm movements are designed around the two axes, Y and Z, it is necessary for the gripper to act in such a way that it performs the task of *picking and placing* objects well. Grippers or similar mechanisms are widely used in industrial robotic arms in factories and production lines. The gripper mechanism can take many forms, but the gripper used here rotates a motor to bring the two gripper jaws closer together and grasp the body in between.

By paying close attention to the gripper mechanism, we will understand that it is necessary to change the direction of the engine movement for each time it is opened and closed. That is, assuming the gripper is closed in the initial state, with the motor rotating approximately 180 degrees, the gripper will first open and then close again. Also, since we can not specify the position for the gripper encoder to be open or closed (because this position will change according to the different dimensions of the captured objects), so it is difficult to find a number as the gripper open or closed position. The best way is to rotate the physical gripper when necessary to rotate it for a certain amount of time (for example, 1 second) in a certain direction (change the direction of rotation with each movement of the gripper) and with a particular force.

Suppose we want to open and close the gripper every time we press the middle key on the break, and as a result, take the body. So, in a new VI, we create a Case Structure with the middle key condition:

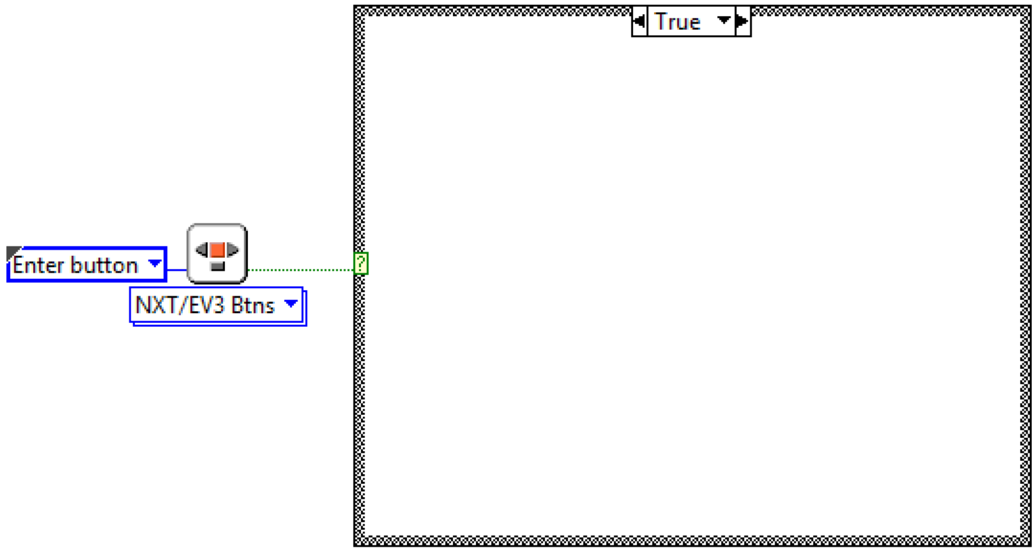


Figure 32 Reading the value of the middle key of the break

As mentioned, each time you press the middle key, motor A must move in the right direction for 1 second.

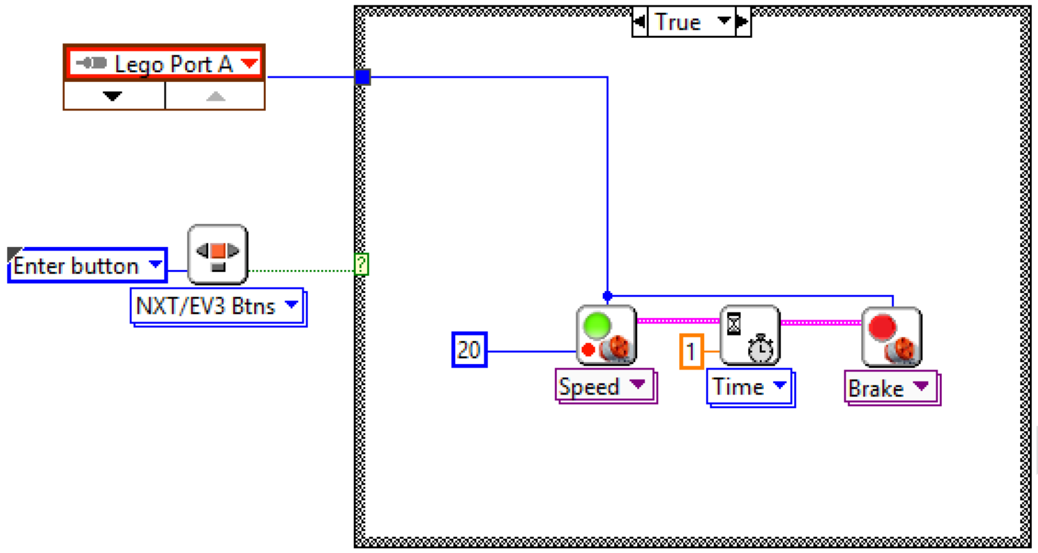


Figure 33 Gripper’s motor moving in the right direction

However, as the saying goes, the motor cannot always move in one direction and open and close the gripper. Therefore, it is necessary to change the direction of the engine movement after each time the middle button is pressed so that the motor movement can be opened and closed the next time. To do this, we use a variable called Dir. First, we put the initial value of 1 in this variable and multiply a negative sign each time the engine moves. As a result, if we use this variable to determine the direction of the motor's rotation, the motor's direction will change each time the button is pressed.

To define a variable, as used in the previous sections, right-click in the Front Panel window and place a Numeric Indicator in your program from the Numeric section. Then rename this numeric index to "Dir." To write in this variable in the Block Diagram window, right-click on the block corresponding to the numeric index and select *Create* → *Local Variable* from the menu that appears. If you want to read the value inside this variable, right-click on the created Local Variable block and select Change to read.

So, to change the direction of the motor each time it rotates, after the middle key is pressed and we move motor B, we change the variable sign to "Dir." As a result, our program is like the one below.

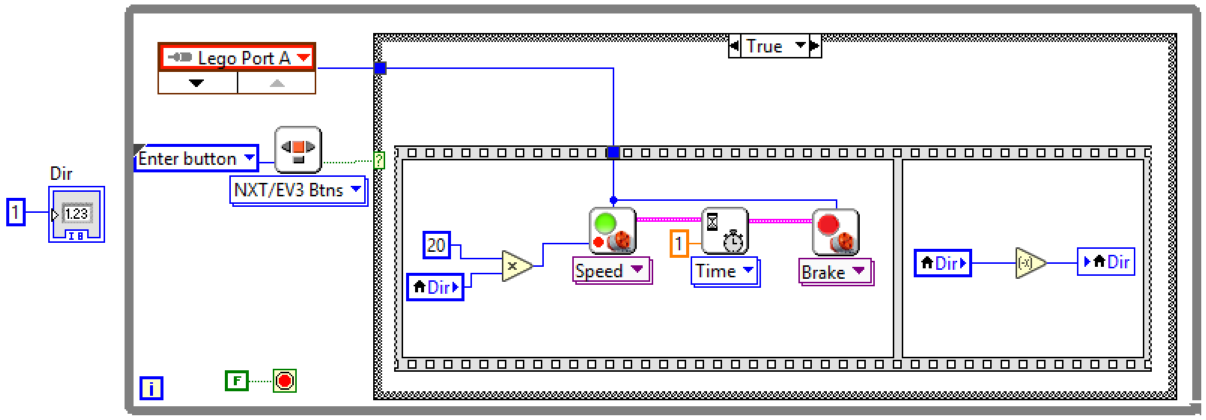


Figure 34 Changing the rotation of the gripper’s movement with each press of the middle key

Note that before the program enters the while loop, we assigned a value of 1 to the Dir variable.

Finally, we save this program with the appropriate name (here as "arm-gripper").

Final Program of Robot Arm

So far, we have programmed the three degrees of freedom of the robotic arm, but each of the robot's movements is programmed separately and independently. This means that different movements cannot be commanded to the arm at the same time.

Since each of the 3 programs designed in the recent episodes are independent and complete, we only need to run them simultaneously. Because Break is only able to run one program at a time, another program must be designed in the lab to cover all three robot movements. To do this, create a new VI (press Ctrl + N in the application window). To import a VI into your application, right-click in the Block Diagram window and select *Select a VI* from the Functions menu. In the window that opens, you need to select the desired VI file. We add to our new program the 3 programs created in the previous sections, which we saved as "arm-y-axis", "arm-z-axis," and "arm-gripper", respectively.

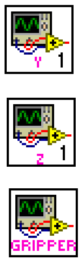


Figure 34 Adding built-in VIs to the new VI

Because there is an infinite while loop in each VIs, there is no need for a while loop in the final program. Now by uploading and running this program on your robot, first, the robot axes return to their original position (so-called home), and then you can use the left and right keys to rotate the robot around the y axis and the up and down keys to rotate the robot around the z-axis. Give. Also, by pressing the middle key of the break, the gripper opens and closes, and if an object is in that position, the gripper takes it.

Robot Arm – Problem Set

1. Program the robotic arm to pick up objects at a specific point and move them to another point in a specific direction.
2. Modify programs designed to be able to control the robotic arm using the *Front Panel*.