

Autonomous Programming: Mecanum

Please complete the Control System Overview, Intro to FTC Programming and Mecanum Drive: Principles and Programming tutorials before proceeding.

Autonomous Programming:

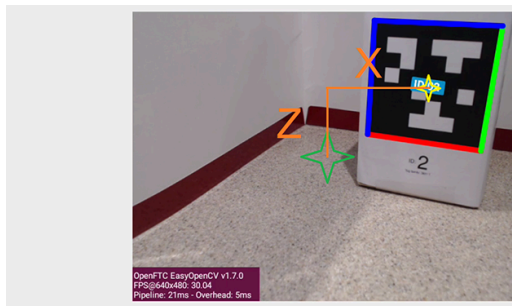
Every First Tech Challenge has an autonomous component. This is a period of time when the team selects a program, then the robot executes the program completely on its own without any driver inputs. The challenge changes every year, but it generally involves moving to specific locations and/or taking in camera data or sensor data to react to the environment.

Tools you can use to complete the tasks:

1. Pre-programmed movements. Give the robot a set of instructions for where to move.
2. Sensor inputs. Scan and react to the environment using a variety of approved sensors: distance sensors, touch sensors, color sensors, magnetic limit switch, IMU, potentiometer.
3. Camera information (webcams). The main 2 ways to use the cameras are to read April Tags or use TensorFlow Lite.

-April Tags are black and white patterns that look like QR codes. They tell you detailed information about your robot's distance and angle from the April tag.

-TensorFlow can be trained to recognize images. Some images might be pre-programmed on the hub, but you can also train it to recognize an image yourself.



Getting "Pose" data from an April Tag.



Recognizes an object with TensorFlow

Learn more/sources:

Approved Sensors:

https://ftc-docs.firstinspires.org/en/latest/control_hard_compon/rc_components/sensors/sensors.html

Approved Webcams:

https://ftc-docs.firstinspires.org/en/latest/control_hard_compon/rc_components/uvc/uvc.html

April Tag Information:

https://ftc-docs.firstinspires.org/en/latest/apriltag/vision_portal/apriltag_intro/apriltag-intro.html

General Information on Autonomous Programming (2023-2024 competition focus)

<https://firstroboticsbc.org/ftc/ftc-team-resources/centerstage-autonomous-programs/>

Programming Autonomous Movement With Power and Time:

In this example we don't have the green "repeat while call opModelsActive" block that we are used to in our TeleOp programs. Any commands that come after the "do" part of the "if call OpModelsActive" block will take place in the order that they are laid out without repeating.

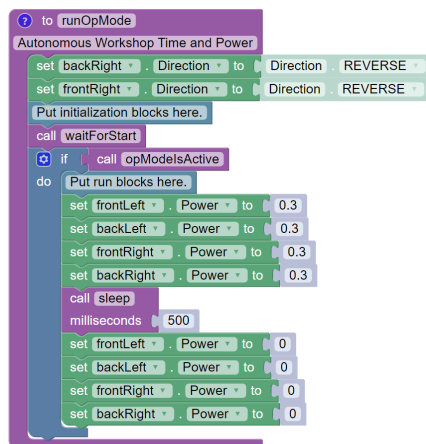
We can tell our robot where to move by telling it what power level to give each wheel (the negative or positive value determines direction according to mecanum wheel logic). The power level will also determine how fast our robot will move.

We then add the "call sleep milliseconds ___" button. This tells our program to pause execution of its next command for the amount of time indicated. So, it will continue to do whatever the last command was until the amount of time indicated has elapsed. Remember that 1000 milliseconds= 1 second. So, 500 milliseconds= ½ second. This block is found in the "Linear OpMode" menu.

Lastly, we then set the power of all 4 wheels to 0. It is important to always set your robot to stop at the end of its movement sequence.

*Note: The robot will stop when it comes to the end of its list of commands or when its autonomous timer runs out (30 seconds). So, if you forgot to add the step that sets all the wheels' power to 0 at the end of this sequence, it would still stop. However, it is a good habit to always manually set the power of each motor to 0 when you want it to stop. The robot's motors will keep doing whatever they were told to do last until they are given a new instruction (like being set to 0), or it comes to the end of its command sequence or the timer ends. So, if you had the robot do a bunch of time-consuming, non-motion steps after your final movement step without setting the motors to 0, the robot would just keep moving.

The following program will make our robot move forward at a power of 0.3 (30% of max power) for 500 milliseconds (½ second) and then stop.

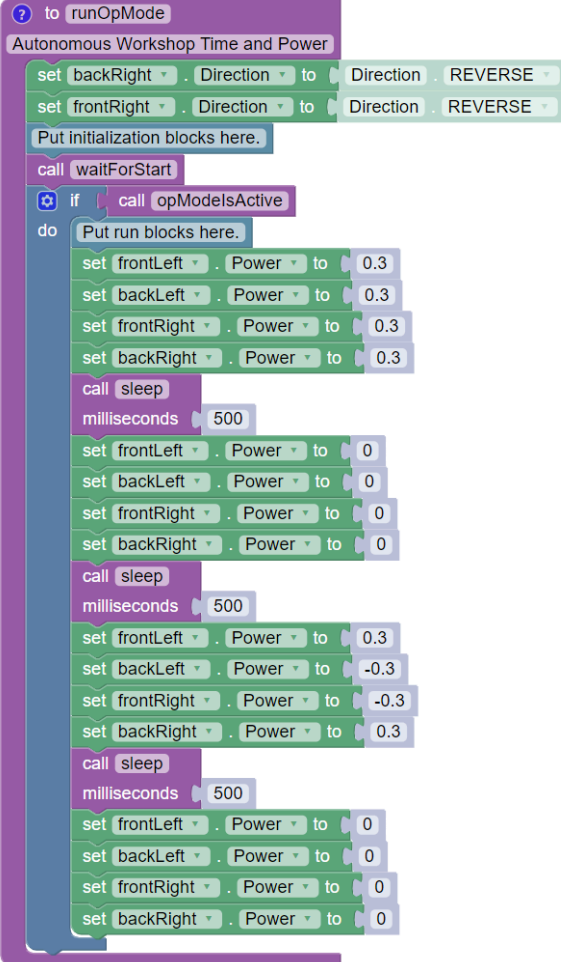


```
to runOpMode
  Autonomous Workshop Time and Power
  set backRight . Direction to Direction REVERSE
  set frontRight . Direction to Direction REVERSE
  Put initialization blocks here.
  call waitForStart
  if call opModelsActive
  do
  Put run blocks here.
  set frontLeft . Power to 0.3
  set backLeft . Power to 0.3
  set frontRight . Power to 0.3
  set backRight . Power to 0.3
  call sleep
  milliseconds 500
  set frontLeft . Power to 0
  set backLeft . Power to 0
  set frontRight . Power to 0
  set backRight . Power to 0
```

Adding Extra Steps:

In order to add additional movements to our sequence, we just have to repeat all the same steps in the process below our first command, but change the **power** or **time** as needed. By making our power positive or negative in accordance with mecanum logic, we will be able to pick our robot's direction of motion.

The following program will move forward at a power of 0.3 for 500 milliseconds, then stop for 500 milliseconds, then strafe to the right for 500 milliseconds, then stop.



```
to runOpMode
  Autonomous Workshop Time and Power
  set backRight . Direction to Direction . REVERSE
  set frontRight . Direction to Direction . REVERSE
  Put initialization blocks here.
  call waitForStart
  if opModelsActive
  do
    Put run blocks here.
    set frontLeft . Power to 0.3
    set backLeft . Power to 0.3
    set frontRight . Power to 0.3
    set backRight . Power to 0.3
    call sleep
    milliseconds 500
    set frontLeft . Power to 0
    set backLeft . Power to 0
    set frontRight . Power to 0
    set backRight . Power to 0
    call sleep
    milliseconds 500
    set frontLeft . Power to 0.3
    set backLeft . Power to -0.3
    set frontRight . Power to -0.3
    set backRight . Power to 0.3
    call sleep
    milliseconds 500
    set frontLeft . Power to 0
    set backLeft . Power to 0
    set frontRight . Power to 0
    set backRight . Power to 0
```

By adjusting the speed and time, you can control exactly how far your robot will move with each step. You can accomplish a lot with pre-measuring, but you will have to do a lot of trial and error to get it exactly right!

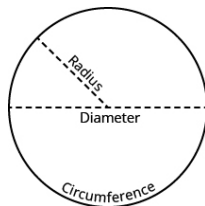
Technically speaking, we do not need the step in the middle where we set the power to 0 for 500 milliseconds. However, including this step will give our movements much greater precision. There is sometimes a tiny lag between the actions of the various wheels, and building in tiny pauses between movements will allow everything to catch up and work together better.

Motor Encoders

An Encoder is a sensor that detects rotation of the motor axel. This can be done with a fairly high degree of precision. Different types of motors will each have a different number of “ticks” or “counts” associated with a single complete revolution. The number of ticks per revolution ranges from hundreds to a few thousand.

Many motors have built-in encoders. You can look this up on the specification chart from the company that produces the motor you are using. A motor with an encoder built in will have 2 cords to plug into your control hub. If you don't have an encoder built-in to your motor, you can add an external encoder that sits around the shaft of the axel and provides the same function.

*If you know the diameter of your wheels and the number of ticks per revolution your motor is measuring, you can calculate the distance that any given number of ticks will take you. You can also just figure it out by trial and error.



$$\text{Circumference} = \pi \text{ times diameter } (c = \pi d)$$



Built in encoder in Rev HD Hex Motor



Rev Through-Bore Encoder

Encoder Based Movements: First Robotics British Columbia

<https://firstroboticsbc.org/ftc/ftc-team-resources/autonomous-encoder-based-movement/>

FTC Approved Motor Encoders:

https://ftc-docs.firstinspires.org/en/latest/control_hard_compon/rc_components/encoders/encoders.html

Oregon Robotics Youtube video on motor encoders

<https://www.youtube.com/watch?v=OMBfgO-AntY>

Programming Autonomous Movement With Encoders:

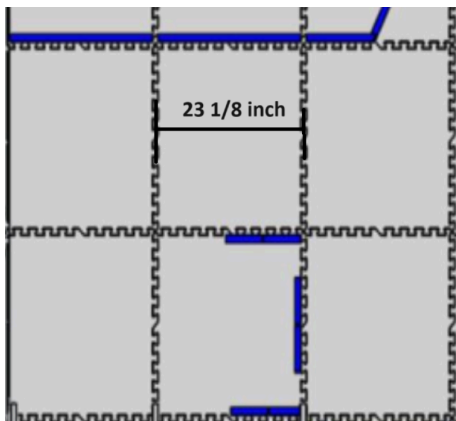
When programming a movement with encoders, you have 6 steps for each movement.

- 1) Set the counter to 0 for all 4 motors. If you already knew what the counter was at, you could just continue adding to it, but it is easier to keep track and avoid errors if you reset the counter before each movement. Do this by selecting the “**RunMode**” block under the “**Actuators**” “**DcMotor**” menu. Then select “**STOP_AND_RESET_ENCODER**” from the drop-down options available.
- 2) Set the **Target Position** for all 4 motors. This block is also available in the “**Actuators**” “**DcMotor**” menu. This is the counter number you want it to go to (how much you want the axle to rotate). This corresponds with the distance you want your robot to travel. As long as you have remembered to reverse the direction of either your right or left two wheels at initialization AND you have set your motor encoders to zero, then a positive target position will make the wheel move forward and a negative target position will make the wheel move backward. *We used 800 in this example- this is the distance that corresponds to a linear motion of ~23 1/8 inches with THIS robot's motors and wheels.*
- 3) Set the target **power** for all 4 motors. *We set it to 0.3 in this example so that it goes slow enough for us to observe the nuances of how our robot is moving in this demo.*
- 4) Tell all 4 motors to “**RUN_TO_POSITION**” from the same drop-down menu. This command will not do anything if you have not already set target position and power.
- 5) Now you need to tell your program to WAIT until all 4 motors have stopped moving before going on to the next step. This is very important. Your control hub can read your program much faster than any motor could spin its wheels. You do not want the program to try to move on to executing the next step until all the wheels have finished moving to their target positions. Building this command is a little bit fussy (luckily, you can copy and paste after you build it once).
 - a) Grab the “**repeat while do**” block from the “**loops**” menu. Change “while” to “until” in the drop down menu.
 - b) Grab 3 “**__and__**” blocks from the “**logic**” menu. Nest them inside each other until you have 4 blanks with three “ands” in between them.
 - c) Grab 4 “**call motor name is busy**” blocks from the “**Actuators**” “**DcMotor**” menu. Plug them into your 4 blanks spaces
 - d) Grab 4 “**not**” blocks from the “**logic**” menu. Place them just in front of your “**call motor name is busy**” blocks.
 - e) Leave the “do” part of the loop block blank. You can add telemetry here if you want. But, it is intentionally empty of any motor commands.
 - f) Yay- now you have created a command that essentially says “now you will continue to do nothing new until all the motors are done doing stuff.”
- 6) Lastly, set all 4 motor's power to 0. We always need to give our robots a true stop at the end of their autonomous motions.

Example Code:

This code will tell our robot to move forward 23 1/8 inches (800 ticks) at a power of 0.3 (30% of maximum). It will then stop.

```
to runOpMode
  Autonomous Workshop Encoders
  Put initialization blocks here.
  set backRight . Direction to Direction REVERSE
  set frontRight . Direction to Direction REVERSE
  call waitForStart
  if call OpModelsActive
  do
  Put run blocks here.
  set frontRight . Mode to RunMode STOP_AND_RESET_ENCODER
  set backRight . Mode to RunMode STOP_AND_RESET_ENCODER
  set frontLeft . Mode to RunMode STOP_AND_RESET_ENCODER
  set backLeft . Mode to RunMode STOP_AND_RESET_ENCODER
  set TargetPosition
    backLeft to 800
    backRight to 800
  set TargetPosition
    frontLeft to 800
    frontRight to 800
  set Power
    backLeft to 0.3
    backRight to 0.3
  set Power
    frontLeft to 0.3
    frontRight to 0.3
  set frontRight . Mode to RunMode RUN_TO_POSITION
  set frontLeft . Mode to RunMode RUN_TO_POSITION
  set backRight . Mode to RunMode RUN_TO_POSITION
  set backLeft . Mode to RunMode RUN_TO_POSITION
  repeat Until
    not call backLeft . isBusy and
    not call backRight . isBusy and
    not call frontLeft . isBusy and
    not call frontRight . isBusy
  do
  We do not need to put anything in this "do" section.
  set Power
    backLeft to 0
    backRight to 0
  set Power
    frontLeft to 0
    frontRight to 0
```



23 1/8 inches is the distance between the center point of the interlocking line between each of the gray foam tiles that are usually used to line the FTC Arena. So a distance of 23 1/8 inches= "1 foam floor tile."

Adding Extra Steps:

In order to add a second movement in our sequence, we just have to repeat all the same steps in the same process below our first command, but change the **Target Position** as we see fit.

This program will move forward 23 1/8 inches, then strafe right 23 1/8 inches, then stop.

```
to runOpMode
  Autonomous Workshop Encoders
  Put initialization blocks here.
  set backRight . Direction to Direction REVERSE
  set frontRight . Direction to Direction REVERSE
  call waitForStart
  if call opModelsActive
  do
    Put run blocks here.
    set frontRight . Mode to RunMode STOP_AND_RESET_ENCODER
    set backRight . Mode to RunMode STOP_AND_RESET_ENCODER
    set frontLeft . Mode to RunMode STOP_AND_RESET_ENCODER
    set backLeft . Mode to RunMode STOP_AND_RESET_ENCODER
    set TargetPosition
      backLeft to 800
      backRight to 800
    set TargetPosition
      frontLeft to 800
      frontRight to 800
    set Power
      backLeft to 0.3
      backRight to 0.3
    set Power
      frontLeft to 0.3
      frontRight to 0.3
    set frontRight . Mode to RunMode RUN_TO_POSITION
    set frontLeft . Mode to RunMode RUN_TO_POSITION
    set backRight . Mode to RunMode RUN_TO_POSITION
    set backLeft . Mode to RunMode RUN_TO_POSITION
    repeat until not call backLeft . isBusy and not call backRight . isBusy and not call frontLeft . isBusy and not call frontRight . isBusy
    do We do not need to put anything in this "do" section.
    set Power
      backLeft to 0
      backRight to 0
    set Power
      frontLeft to 0
      frontRight to 0
    set frontRight . Mode to RunMode STOP_AND_RESET_ENCODER
    set backRight . Mode to RunMode STOP_AND_RESET_ENCODER
    set frontLeft . Mode to RunMode STOP_AND_RESET_ENCODER
    set backLeft . Mode to RunMode STOP_AND_RESET_ENCODER
    set TargetPosition
      backLeft to -800
      backRight to 800
    set TargetPosition
      frontLeft to 800
      frontRight to -800
    set Power
      backLeft to 0.3
      backRight to 0.3
    set Power
      frontLeft to 0.3
      frontRight to 0.3
    set frontRight . Mode to RunMode RUN_TO_POSITION
    set frontLeft . Mode to RunMode RUN_TO_POSITION
    set backRight . Mode to RunMode RUN_TO_POSITION
    set backLeft . Mode to RunMode RUN_TO_POSITION
    repeat until not call backLeft . isBusy and not call backRight . isBusy and not call frontLeft . isBusy and not call frontRight . isBusy
    do We do not need to put anything in this "do" section.
    set Power
      backLeft to 0
      backRight to 0
    set Power
      frontLeft to 0
      frontRight to 0
```

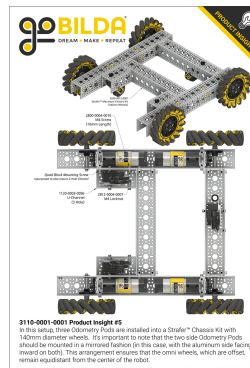
Odometry

The third way to give your robot pre-programmed movement instructions is with odometry. Some of the names used to describe this method are “odometry” “3-wheel odometry” or “dead-wheel odometry” or just “dead-wheel.”

How it works:

You install 3 wheels to your robot spread out in a triangular configuration (2 facing in the same direction and 1 facing perpendicular to the other 2). These wheels contact the ground, but they do not have motors attached and they do not provide any propulsion to the robot. This is why they are referred to as “dead wheels.” As the robot moves along the ground, it moves the wheels. The wheels each have sensors that tell them how much they have moved in any given direction. The program uses this information to decide whether or not to keep moving in any given direction. If you are using odometry, you set a target distance as your stop cue (not power and time and not axel rotations). *2-wheel version also exists, but needs an “odometry computer” installed on the robot to be effective.

The HUGE advantage that this provides, is that it removes a large and common error from your code. Imagine that your robot catches its wheel on the edge of a pole that is anchored to the ground. Now your robot’s wheels are still spinning, but the robot isn’t moving. The robot just keeps trying to run into the pole and doesn’t go anywhere, but the program is still trying to carry out its commands as if nothing has happened. A robot with an odometer would be able to tell that it isn’t actually moving, because the dead wheel wouldn’t move while the robot was stuck on the pole. A clever programmer could even build in a clause for what to do if the robot stops unexpectedly (something like: back up 1 inch and then move left 1 inch and try again).



Odometers are pretty much impossible to use with blocks programming. For those of you who are interested, get cracking on that Java or Python! We have a good PDF tutorial on how to program in Java for FTC. It’s 215 pages, but it is meant for beginners. Let us know and we will send it to you! You can practice your code on the FTC virtual robot simulator. You can also start learning Java with Code Academy, which is a free online course.

<https://vrobotsim.com/> [Code Academy](#)

GoBilda Odometry Pod

<https://www.gobilda.com/odometry-pod-43mm-width-48mm-wheel/>

Youtube video on Odometry for FTC: Aperture Science

<https://www.youtube.com/watch?v=Av9ZMjS--gY>

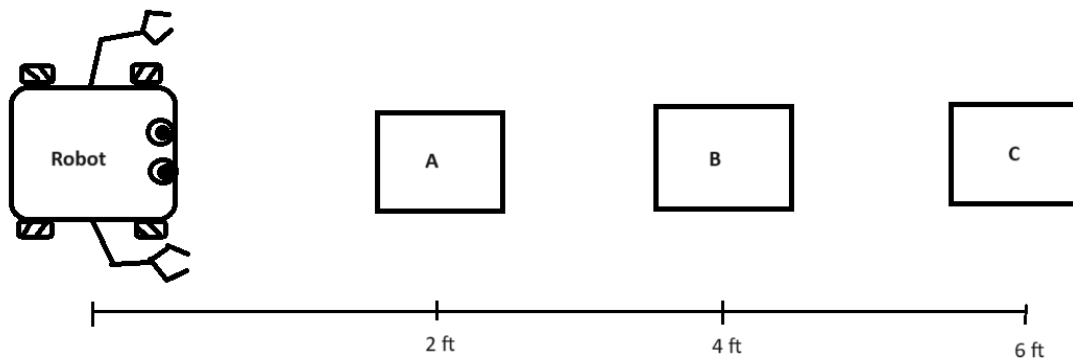
What is a Function?

When programming, it is helpful to use functions to stream-line your code.

A function is a piece of code you can call on to do several steps with one command.

Example:

Imagine that I had a challenge that said go to location A, B and C and perform a little dance in each location. If I create a function that describes what my “dance” looks like, I will save myself a lot of time and space when I write my code.



Function: “Dance”

My code might include commands to do the following: *Move right arm up and left arm down*→*then move right arm down and left arm up*→*then turn right 10 degrees*→*then turn left 10 degrees*→*then put both arms up*→*then do a full circle to the right*→*then put both arms down*.

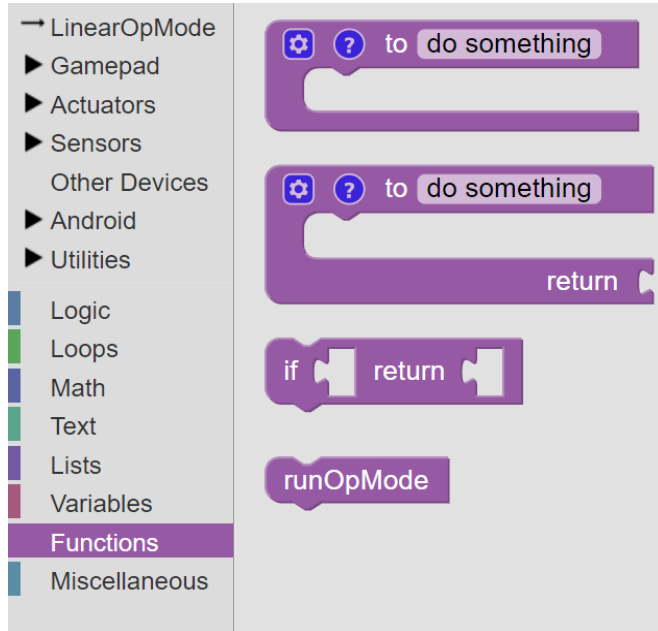
Now that I have defined my function, I can write my code like this:

Code Using “Dance” Function:

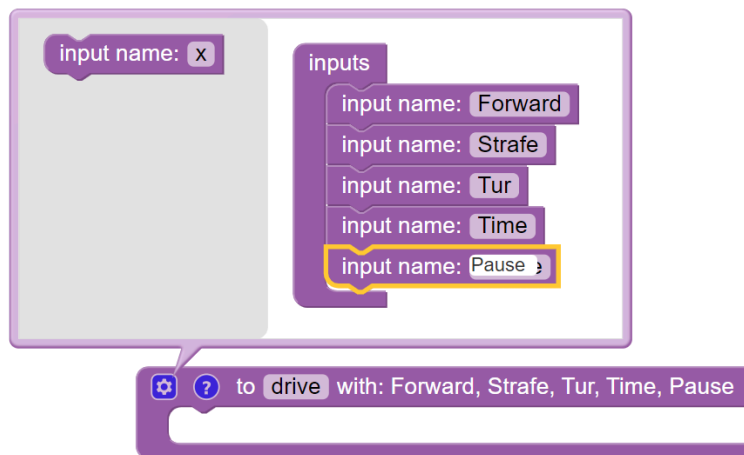
Move forward 2 feet (location A)→*Dance*→*Move forward 2 feet (location B)*→*Dance*, *Move forward 2 feet (location C)*→*Dance*

Making Functions in Blocks:

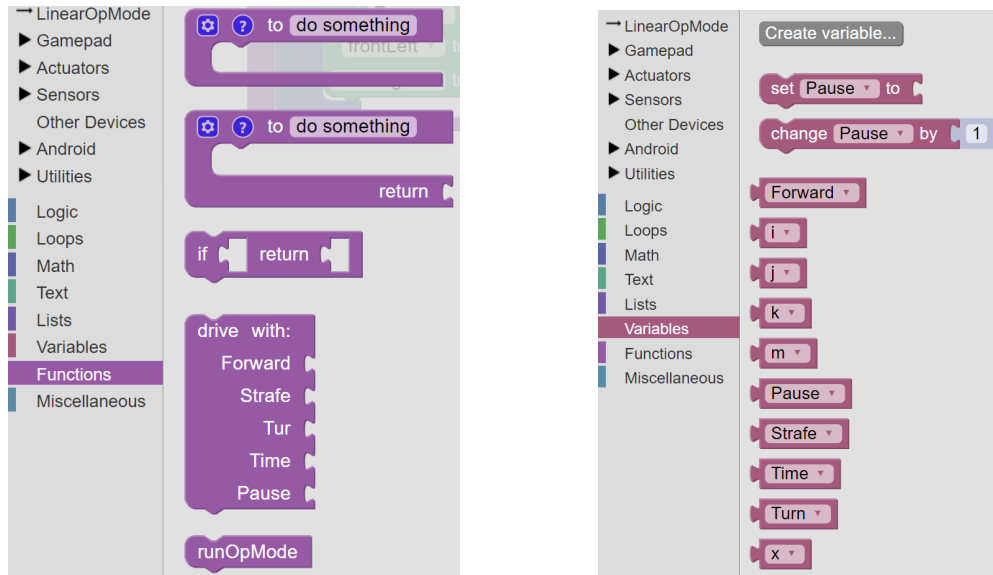
- Open the “**function**” menu.
- Select the “**to something**” block (the one without the “return” slot for now).
- Once you have selected it, you can type in a new word where it currently says “do something.”
- Let’s change it to say “drive.”



- Click on the gear icon to make your function box more complex.
- Drag the “input name: __” block into the slot under the “inputs” block. You can add as many as you need to. *We added 5 in this example.*
- Close the pop up window by clicking the gear again.



- Now that you have created this function, you will find a new block available under the **“function”** menu that corresponds to the function you just created.
- You will also find new variables in the **“variables”** drop-down menu that correspond with the inputs you named in your function.



Robotic Programming Ethics: A quick aside

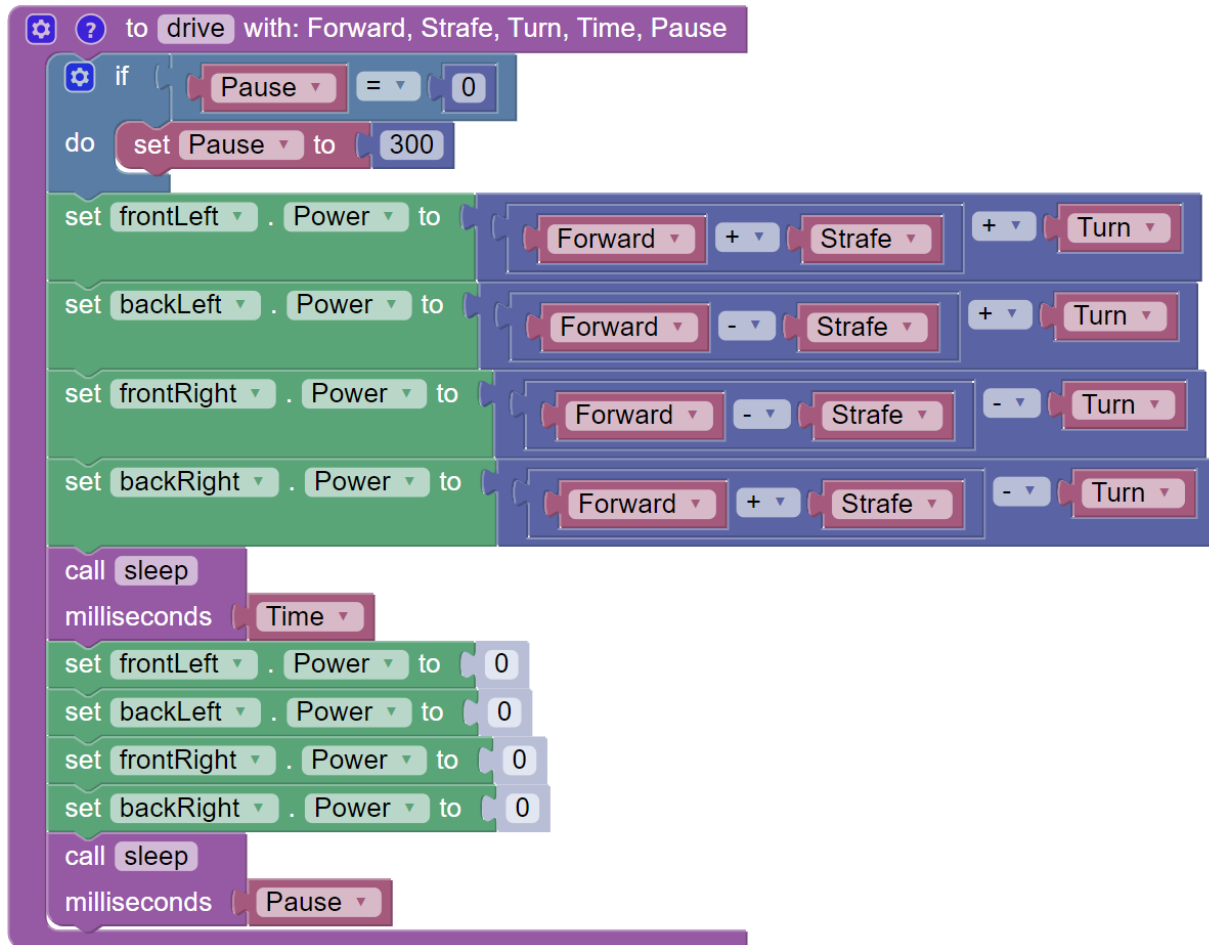
*Note: There are LOTS of ways to create functions. There are variations on every one of these steps. If someone teaches you another way to do it, that is perfectly fine. But, always make sure you are understanding it. Feel free to share code with each other, but always make sure to TEACH your code to others if you share it. Professional programmers use code written by other people all the time. But, if they don't understand how it works, it would be impossible for them to fix it if it doesn't work perfectly (and it probably won't).

Here is a good general rule for using code that you didn't write: If you feel confident that you could recreate it yourself with a little time and thinking... then you can use it. This goes for using parts of your robot that were created by other teams or your own team in a previous year. If you feel you could build it on your own with some extra time... go ahead and use it. If you are not sure, then it's probably best to start fresh and learn how each step is done. You might not have as "good" a robot as if you used other people's builds and other people's code. But, you will LEARN A LOT MORE. And those skills will take you farther than any individual competition.

Building a Time and Power Function:

Inside our new function box, we will need to input our logic, similar to how we did in the mecanum TeleOp OpMode.

*We do not need to add the “regulator/denominator” step that we used before, because that was only needed to compensate for an issue that came up when using the joystick. There is no joystick in autonomous mode!



- 1) Ignore the “if do” block at the top. We will come back to it in a moment.
- 2) Insert 4 “Set **motor name**. Power to” blocks. Use the drop-down menu to make sure all 4 wheels are accounted for. These are in the “Actuators” “DcMotor” menu.
- 3) For each line after motor power, you will need to grab to “__ +/- __” blocks from the “Math” menu. Nest one of the blocks inside the other so that you have 3 blanks, separated by 2 +/- symbols.
- 4) Grab 4 copies of the **Forward, Strafe and Turn** variables from the “Variables” menu.
 - a) Plug these into the blank spaces in your “__ +/- __” blocks.

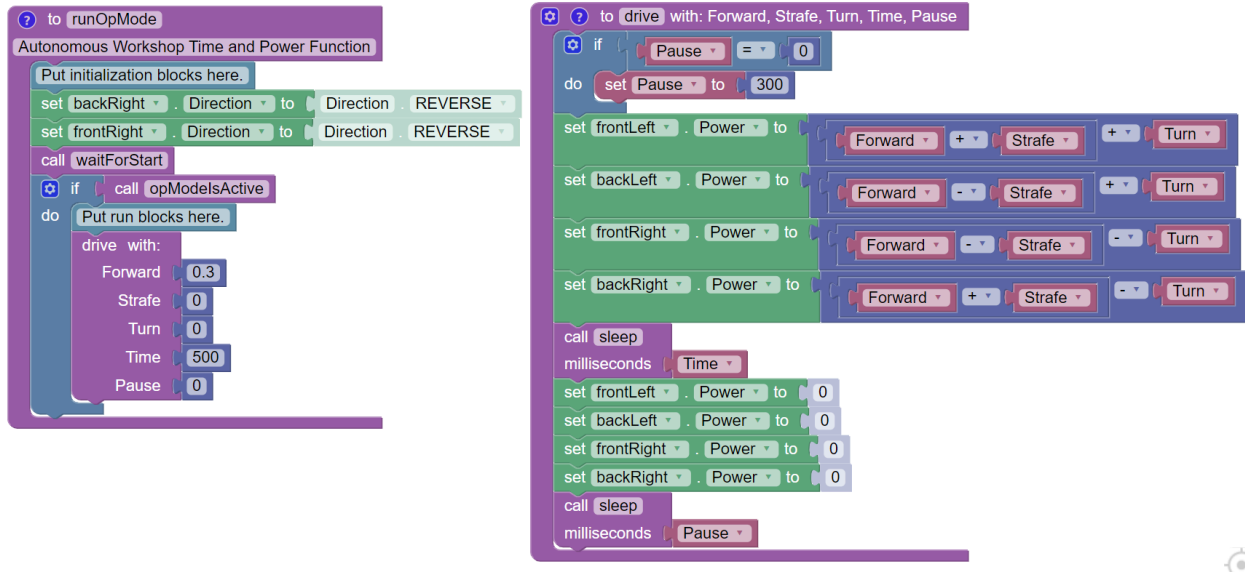
- b) Adjust the +/- symbol to create the mecanum logic we know and love.
- 5) Grab the “**call sleep milliseconds** ___” block from the “**LinearOpMode**” menu and place it after the 4 motor power levels you just set.
- 6) Grab the “**Time**” variable from the “**Variable**” menu. Plug it in after milliseconds.
- 7) Add 4 more “**Set motor name. Power to**” blocks after the sleep box.
 - a) Use the drop-down menu to assign one of these boxes to each of your motors.
- 8) Add another “**call sleep milliseconds** ___” block after you set all the motor’s power to 0.
 - a) Add the “**Pause**” variable to this block after milliseconds.
 - b) The point of this pause is to make sure that the robot stops for a quick break in between each of its movements. This will give it much more precision because the code won’t be moving on to the next set of instructions before it completes the first set. The pause doesn’t need to be very long.
- 9) At the top of your function, add an “**if do**” box from the “**logic**” menu.
 - a) After the “if,” add a “**___ = ___**” block from the “**Logic**” menu.
 - b) Add the “**Pause**” block from the “**Variable**” menu to the first blank.
 - c) Add a blank number (“**0**”) block from the “**Math**” menu to the second blank.
 - d) Add a “**set Variable to**” block from the “**Variables**” after the “do” command.
 - i) Select “**Pause**” from the drop-down menu.
 - e) Add a blank number (“**0**”) block from the “**Math**” menu after the “**set Pause to**” block.
 - f) Change the number to 300.
 - g) What we just said with this “**if do**” block in our code is the following: If 0 is entered into the space provided for the “**Pause**” line in our function box, then it will be automatically changed to 300. Our little pause at zero power will last 300 milliseconds (0.3 seconds) for every one of our motions unless we specifically give it a new number in our function box. This means we don’t really have to think about it any more for most of our movements.

Now you have created your Time/Power function. For each movement instruction you give your robot with this function, it will move with the power you input with its mecanum logic already figured out and it will stop appropriately after the time you tell it (in milliseconds). It will also take a 0.3 second pause between each movement to make sure it is running smoothly.

Using a Time and Power Function:

Now that you have your function defined, you can plug in each movement rapidly!

- 1) Grab your new “**drive with: Forward, Strafe, Turn, Time, Pause**” block from the “**Functions**” menu.
- 2) Place it in your code in the “**if call opModelsActive do**” section of your code.
- 3) Grab a blank number block (“**0**”) from the “**Math**” menu.
 - a) Place a blank number next to Forward, Strafe, Turn and Time and Pause



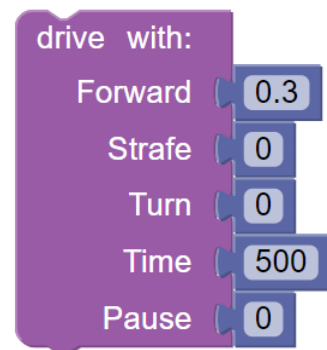
Now that you have your function block, you just need to put 2 numbers into each blocks' number boxes. You are only going to choose a value for either Forward, Strafe or Turn for each of your movements. Your robot will not work properly if you try to get it to do more than 1 of these at once.

The number you input is the power level that you will use to complete the movement you are planning. The direction of your movement is determined by putting in either a negative number or a positive number.

- A positive power level in the Forward box will produce a **FORWARD** motion.
- A negative power level in the Forward box will produce a **BACKWARD** motion.
- A positive power level in the Strafe box will produce a **RIGHT** lateral motion.
- A negative power level in the Strafe box will produce a **LEFT** lateral motion.
- A positive power level in the Turn box will produce a **CLOCKWISE** motion.
- A negative power level in the Turn box will produce a **COUNTERCLOCKWISE** motion.

You will need to input time for every function box.
The time is measured in milliseconds.

In this example, the robot will move forward with a power of 0.3 (30% of maximum) for 500 milliseconds (½ second) and then stop for a 0.3 second pause before moving on to the next instruction.

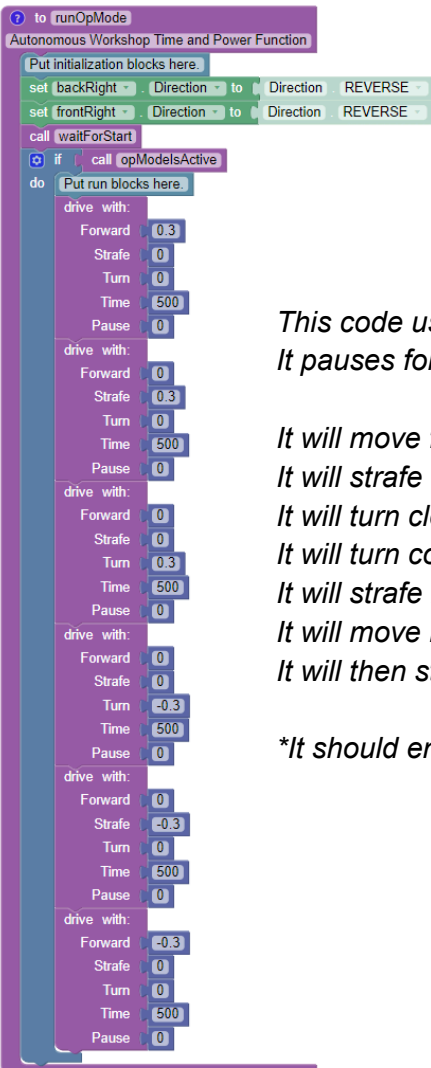


Building More Complex Instructions:

Now that we have built our function, we can add new steps to our movement instructions very easily. We simply copy and paste the “**drive with: Forward, Strafe, Turn, Time, Pause**” block as many times as we need and place them one after another. Then, all we have to do is go in and set our power level for either **Forward, Strafe** or **Turn** and set our **Time**. We will assign either a positive or a negative value, depending on which direction we are trying to go.

*Remember to set the 2 values you are not using (out of Forward, Strafe, Turn) to 0. If you copy and paste, you will probably have a value in your box you don't want from where you copied and you must remember to correct it.

*You will probably NOT have to change the pause at any time, but you may want to adjust it occasionally (maybe one part of your navigation requires more precision and you want to make your pauses longer during that section of your movements).



*This code uses a power of 0.3 (30% of maximum for all its movements
It pauses for 300 milliseconds (0.3 seconds) between each movement*

It will move forward for 500 milliseconds (½ second)

It will strafe right for 500 milliseconds

It will turn clockwise for 500 milliseconds

It will turn counterclockwise for 500 milliseconds

It will strafe left for 500 milliseconds

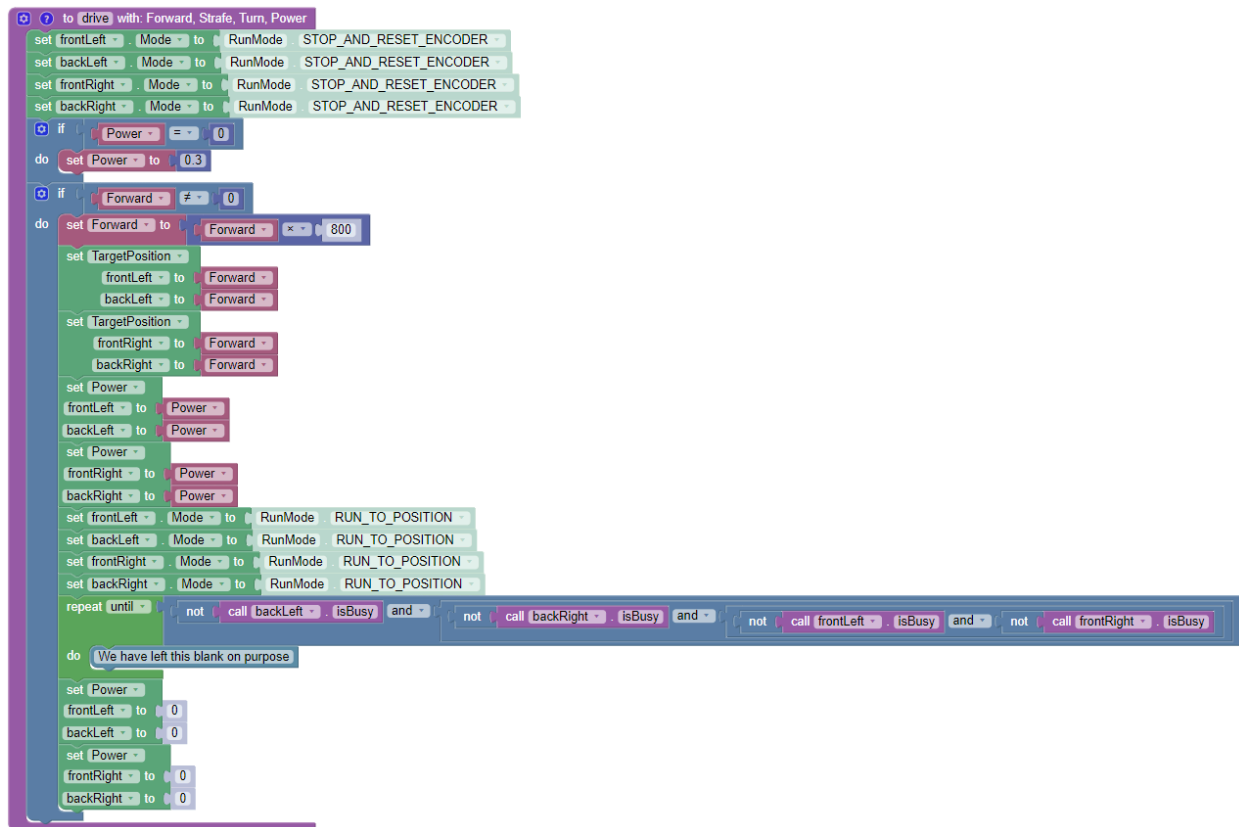
It will move backward for 500 milliseconds

It will then stop

**It should end up exactly back where it started.*

Building an Encoder Function:

Using a function for our encoder instructions will save us a LOT of time and effort.



- 1) Create a function using the “**to do something**” block from the “**Functions**” menu.
 - a) Use the gear to create the inputs **Forward, Strafe, Turn** and **Power**.
 - b) Change “do something” to “drive”
- 2) Set all the motor encoder counters to zero. Do this by selecting the “**RunMode**” block under the “**Actuators**” “**DcMotor**” menu. Select “**STOP_AND_RESET_ENCODER**” from the drop-down options available.
 - a) Insert 4 blocks this way and select each motor from the drop-down menu.
- 3) Grab an “**if do**” block from the “**Logic**” menu.
 - a) Add a “**__ = __**” block from the “**Logic**” menu after the if statement.
 - i) Add the “**Power**” variable from the “**Variable**” menu to the first blank.
 - ii) Add a blank number (“**0**”) from the “**Math**” menu to the second blank.
 - (1) Keep it at zero.
 - iii) Add a “**set variable to**” block from the “**Variable**” menu after the do command.
 - (1) Change it to “**power**” in the drop-down menu.
 - iv) Add a blank number (“**0**”) from the “**Math**” menu after the “**set Power to**” block.

- (1) Enter a number here. *We used 0.3 (30% of max power) in this example because it is nice and slow so that we can see what is happening with each step of our movements. Set it higher if you want your robot to go faster as a default.*
- b) We just created a short-cut. Now, if we leave the power level of each of our function blocks set to 0, it will automatically give it a power of 0.3 (*or whatever you chose*). This means we will have 1 less thing to think about with each of our instructions. However, we can still add in a value for power if we want it to go faster or slower than the default value we have set here.
- 4) Grab another “**if do**” block from the “**Logic**” menu and place it AFTER the one we just created.
- a) Grab the “**__ = __**” block from the “**Logic**” menu and place it after the “if”
- i) Add the “**Forward**” block from the “**Variable**” menu to the first blank.
 - ii) Add a blank number (“**0**”) block from the “**Math**” menu to the second blank.
 - iii) Change the “=” symbol to \neq in the drop-down menu.
- b) Everything we place in the “do” portion of this “**if do**” block will now happen in sequence if we enter anything other than 0 into our function box for the “**Forward**” value.
- c) Grab a “**set variable to**” block from the “**Variable**” menu and place it after the “do” command.
- i) Grab a “**__ x __**” block from the “**Math**” menu and place it after the “**set variable to**” block.
 - ii) Grab the “**Forward**” block from the “**Variable**” menu and place it in the first blank space (pre-populated with a number in this case).
 - iii) Set the second number to 800 (*or a number of your choosing*).
 - iv) *This is another shortcut. We previously discovered that 800 ticks caused our robot to move $23 \frac{1}{8}$ inch, which was exactly the length of 1 gray floor tile. Now, if we put in a value of “1” in our function box, it will be multiplied by 800. This will be easier for us to think about when we are trying to work out our robot’s movements. We will now know that a value of 1 translates to “move 1 floor tile length.”
- (1) We might want to put in a text note to remind ourselves why we chose these numbers.
 - (2) The number of ticks that equals 1 floor tile will be different for every robot based on its motor type, reducers and wheel size.
- d) The next thing we will put in this “**if do**” block is our “**set Target Position**” blocks for all 4 motors. You can find the blocks that contain 2 motors per block under the “**Actuators**” “**DcMotor**” “**Dual**” menu.
- i) Make sure all 4 motors have been selected from the drop down menu.
 - ii) Grab the “**Forward**” block from the “**Variables**” menu and place it after the “**set Target Position**” blocks for all 4 motors.

- iii) *Now, whatever number we put in our function box for **“Forward”** will be multiplied by 800 and used as our Target Position.
- e) Next we need to set power for all 4 motors. Grab the **“Set Power”** blocks under the **“Actuators” “DcMotor” “Dual”** menu.
 - i) Make sure all 4 motors have been selected from the drop down menu.
 - ii) Grab the **“Power”** block from the **“Variables”** menu and place it after the **“set Power”** blocks for all 4 motors.
 - iii) *Now, whatever number we put in the function box for **“Power”** will be used as the power level for our movement. However, if we leave it blank, it will be automatically set to the number we assigned as our default power level (*0.3 in this example*).
- f) Next, tell all 4 motors to **“RUN_TO_POSITION”** from the **“RunMode”** block under the **“Actuators” “DcMotor”** menu.
 - i) Select all 4 motors from the drop-down menus.
- g) Now you need to tell your program to WAIT until all 4 motors have stopped moving before moving on to the next step.
 - i) Grab the **“repeat while do”** block from the **“loops”** menu,. Change “while” to “until” in the drop down menu.
 - ii) Grab 3 **“__and__”** blocks from the **“logic”** menu. Nest them inside each other until you have 4 blanks with three “ands” in between them.
 - iii) Grab 4 **“call motor name is busy”** blocks from the **“Actuators” “DcMotor”** menu. Plug them into your 4 blanks spaces
 - iv) Grab 4 **“not”** blocks from the **“logic”** menu. Place them just in front of your **“call motor name is busy”** blocks.
 - v) Leave the “do” part of the loop block blank.
- h) Lastly, set all 4 motor’s power to 0 to make sure it stops at the end of each movement. Use the **“set Power”** blocks again, but this time just give each motor a value of 0.

*So far, we have only created the command for Forward. Yikes. But, luckily, we can copy and paste and make just a few changes for Strafe and Turn.

Copy the entire “if do” block that describes the **Forward** entry on the function box and paste it.

You will need to make a few changes to the code:

- 1) Change every “**Forward**” variable to “**Strafe**” using the drop-down menus.
- 2) Reverse the back left and front right motors numbers. This is how we achieve the mecanum logic that we know and love.
 - a) Grab the “___ x ___” block from the “**Math**” menu and place it after the back left and front right motors in the “**set TargetPosition to**” blocks (replaces the variable box).
 - i) Put the “**Strafe**” block into the first blank of “___ x ___” block (covering the number that was pre-populated).
 - ii) Change the second number to “-1”
 - b) Now, whatever number we put in for our **Strafe** value on our function box will be the our Target Position for the two of our motors (front left and back right motors). But the two motors we just modified (back left and front right motors) will move the same amount in the opposite direction. This will create a rightward strafing motion if we put in a positive number and a leftward strafing motion if we put in a negative number.

```
if (Strafe != 0)
do
  set Strafe to 800
  set TargetPosition
  frontLeft to Strafe
  backLeft to (Strafe * -1)
  set TargetPosition
  frontRight to (Strafe * -1)
  backRight to Strafe
  set Power
  frontLeft to Power
  backLeft to Power
  set Power
  frontRight to Power
  backRight to Power
  set frontLeft Mode to RunMode RUN_TO_POSITION
  set backLeft Mode to RunMode RUN_TO_POSITION
  set frontRight Mode to RunMode RUN_TO_POSITION
  set backRight Mode to RunMode RUN_TO_POSITION
  repeat until (not call backLeft isBusy and not call backRight isBusy and not call frontLeft isBusy and not call frontRight isBusy)
do
  We have left this blank on purpose
  set Power
  frontLeft to 0
  backLeft to 0
  set Power
  frontRight to 0
  backRight to 0
```

Copy the entire “if do” block that describes the **Strafe** entry we just made and paste it.

You will need to make a few changes to the code:

- 1) Change every “**Strafe**” variable to “**Turn**” using the drop-down menus.
- 2) Changes which motors get the “**Turn x -1**” blocks instead of just a “**Turn**” block.
 - a) We want the front right and back right motors to get the “**Turn x -1**” blocks this time.
 - b) *Now, if we put in a positive value in the **Turn** portion of our function block, it will turn our robot clockwise by the target amount we indicate. If we put in a negative value, it will produce a counterclockwise turn.

The image shows a Scratch script for a turn function. It starts with an 'if' block where the condition is 'Turn \neq 0'. Inside the 'do' block, the following steps are performed: 1. 'set Turn to Turn x 800'. 2. 'set TargetPosition frontLeft to Turn' and 'set TargetPosition backLeft to Turn'. 3. 'set TargetPosition frontRight to Turn x -1' and 'set TargetPosition backRight to Turn x -1'. 4. 'set Power frontLeft to Power', 'set Power backLeft to Power', 'set Power frontRight to Power', and 'set Power backRight to Power'. 5. 'set frontLeft Mode to RunMode RUN_TO_POSITION', 'set backLeft Mode to RunMode RUN_TO_POSITION', 'set frontRight Mode to RunMode RUN_TO_POSITION', and 'set backRight Mode to RunMode RUN_TO_POSITION'. 6. A 'repeat until' block with the condition 'not call backLeft isBusy and not call backRight isBusy and not call frontLeft isBusy and not call frontRight isBusy'. 7. A 'do' block with the comment 'We have left this blank on purpose'. 8. 'set Power frontLeft to 0', 'set Power backLeft to 0', 'set Power frontRight to 0', and 'set Power backRight to 0'.

The last step is to put these two new “if do” blocks into our function. Remember to stack the “if do” blocks for Forward, Strafe and Turn on top of each other in sequence (they should not be nested inside each other).

The full function is really long:

```
0 to (Drive) with: Forward, Strafe, Turn, Power
set frontLeft . Mode to RunMode STOP_AND_RESET_ENCODER
set backLeft . Mode to RunMode STOP_AND_RESET_ENCODER
set frontRight . Mode to RunMode STOP_AND_RESET_ENCODER
set backRight . Mode to RunMode STOP_AND_RESET_ENCODER
do if Power == 0
do set Power to 0.3
do if Forward == 0
do set Forward to Forward * 1 800
set TargetPosition to
frontLeft to Forward
backLeft to Forward
set TargetPosition to
frontRight to Forward
backRight to Forward
set Power to
frontLeft to Power
backLeft to Power
set Power to
frontRight to Power
backRight to Power
set frontLeft . Mode to RunMode RUN_TO_POSITION
set backLeft . Mode to RunMode RUN_TO_POSITION
set frontRight . Mode to RunMode RUN_TO_POSITION
set backRight . Mode to RunMode RUN_TO_POSITION
repeat until not call backLeft . isBusy and not call backRight . isBusy and not call frontLeft . isBusy and not call frontRight . isBusy
do We have left this blank on purpose
set Power to
frontLeft to 0
backLeft to 0
set Power to
frontRight to 0
backRight to 0
do if Strafe == 0
do set Strafe to Strafe * 1 800
set TargetPosition to
frontLeft to Strafe
backLeft to Strafe * 1 -1
set TargetPosition to
frontRight to Strafe
backRight to Strafe
set Power to
frontLeft to Power
backLeft to Power
set Power to
frontRight to Power
backRight to Power
set frontLeft . Mode to RunMode RUN_TO_POSITION
set backLeft . Mode to RunMode RUN_TO_POSITION
set frontRight . Mode to RunMode RUN_TO_POSITION
set backRight . Mode to RunMode RUN_TO_POSITION
repeat until not call backLeft . isBusy and not call backRight . isBusy and not call frontLeft . isBusy and not call frontRight . isBusy
do We have left this blank on purpose
set Power to
frontLeft to 0
backLeft to 0
set Power to
frontRight to 0
backRight to 0
do if Turn == 0
do set Turn to Turn * 1 800
set TargetPosition to
frontLeft to Turn
backLeft to Turn
set TargetPosition to
frontRight to Turn * 1 -1
backRight to Turn * 1 -1
set Power to
frontLeft to Power
backLeft to Power
set Power to
frontRight to Power
backRight to Power
set frontLeft . Mode to RunMode RUN_TO_POSITION
set backLeft . Mode to RunMode RUN_TO_POSITION
set frontRight . Mode to RunMode RUN_TO_POSITION
set backRight . Mode to RunMode RUN_TO_POSITION
repeat until not call backLeft . isBusy and not call backRight . isBusy and not call frontLeft . isBusy and not call frontRight . isBusy
do We have left this blank on purpose
set Power to
frontLeft to 0
backLeft to 0
set Power to
frontRight to 0
backRight to 0
```

Using an Encoder Function:

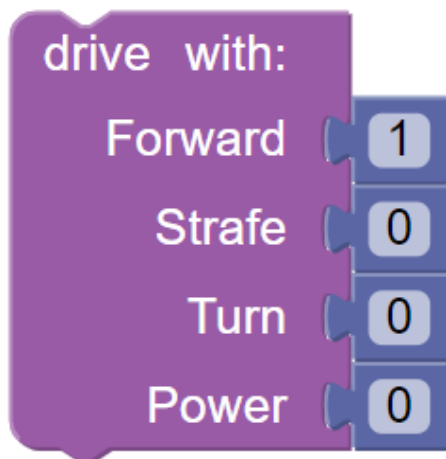
Now that we have built our encoder function, we can program each of our movements with a function box. Each box corresponds to a movement in our sequence. The “**drive with Forward, Strafe, Turn, Power**” block is in the “**Functions**” menu. You will also have to add the blank number box (“**0**”) from the “**Math**” menu to each of the variables in our function.

You will put in a number to either **Forward, Strafe,** or **Turn**. Do NOT put a number in more than one of these three boxes- it will confuse the program.

The number you enter is the “**Target Position,**” that your robot will go to. This corresponds to the distance your robot will move.

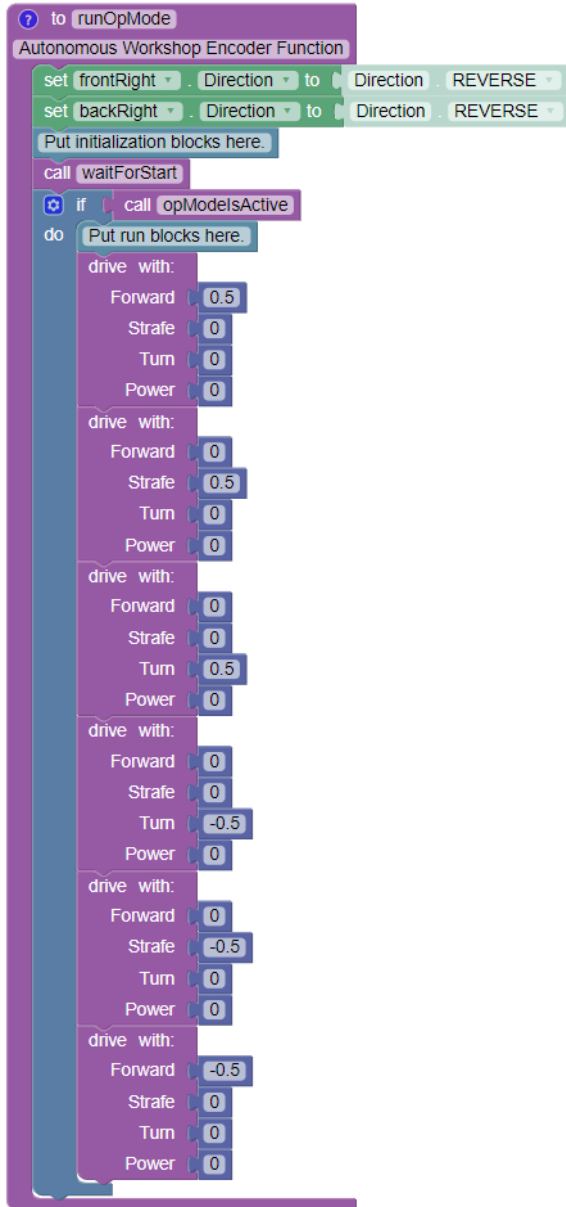
Each movement must also have a Power level. We set our Power level to automatically run at 0.3 (30% of maximum) if we have a 0 in this box, so we don't need to put a number other than zero here if we don't want to change it. If we want the step to go faster or slower, we can manually put in the power.

In this example, we added a 800 x multiplier to our number. So, by putting “1” in the Forward slot, we are telling all 4 motors to move forward 800 ticks. On THIS robot's build (specific to its motors, reducers and wheel size), 800 ticks will move the robot forward 23 1/8 inches, which is equivalent to 1 gray tile. We have left the Power level at 0. This means that our program will use a power of 0.3, as we instructed it to do in our function box. In summary: by putting “1” in the Forward space, THIS robot will move forward 1 tile's length at 30% of max power level.



Building More Complicated Instructions:

To add extra movements, all we have to do is copy and paste our “drive with: Forward, Strafe, Turn, Power” function block and stack them up in sequence within our main “if call opModelsActive do” block. Remember to set the number you want in the correct box and then set the other two boxes to 0 (out of Forward, Strafe and Turn).



This program will make our robot move forward ~1 foot (400 ticks), then strafe right 1 ft, then turn clockwise 400 ticks (about 45 degrees on this robot), then turn counterclockwise 45 degrees, then strafe left 1 ft, then move backward 1 foot. We should end up back where we started.