

# Autonomous Programming: Tank Drive

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

## Autonomous Programming:

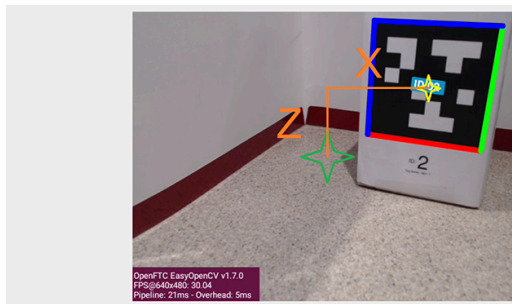
Every First Tech Challenge has an autonomous component. This is a period of time when the team selects a program, then the robot executes the program completely on its own without any driver inputs. The challenge changes every year, but it generally involves moving to specific locations and taking in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

## Power and Time:

In this example we don't have the green “repeat while call opModelsActive” block that we are used to using 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.

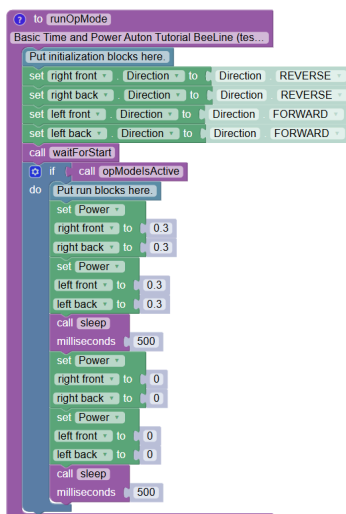
We can tell our robot where to move by telling it what power level to give each wheel (the negative or positive value determines direction). The power level will also determine how fast our robot will move. If you have a multi-motor configuration, remember to set all the right motors to go positive or negative together and all the left motors to go positive or negative together.

We then add the “call sleep milliseconds \_\_\_” button. This tells our program to pause execution of its commands for the amount of time indicated. 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: Without setting the wheel power to 0 at the end, the power level would not change from 0.3 until it was given new instructions about the power level of the motors or the OpMode ends. It might not look like the robot is stopping without setting the robot's power to 0 because this command comes at the end of our list of commands in our sequence. Our robot will stop at the end of its list of commands. However, if we put lots of time consuming commands after the “call sleep milliseconds \_\_\_” block without changing the motors' power levels, the motors will keep going at the 0.3 power that they were last instructed to use.

*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. The “call sleep milliseconds \_500” block at the end is optional.*

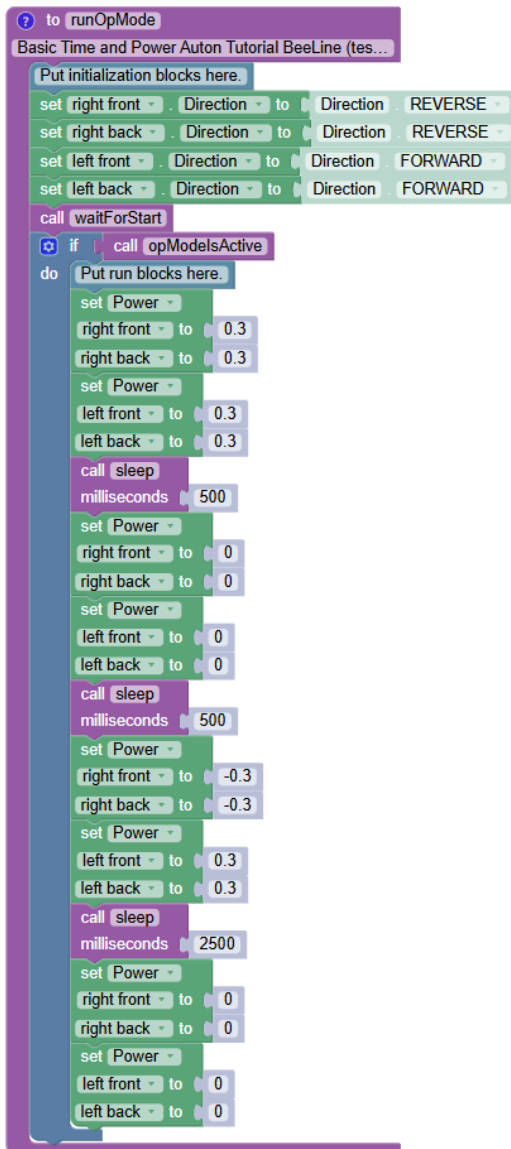


## Adding Extra Steps:

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 the way we want the wheels to move, 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 turn to the right for 2500 milliseconds, then stop.*

*\*In this particular robot's set-up, a 30% power turn for 2500 milliseconds produced a 90 degree turn to the right. We figured this out by trial and error. Adding a text note to ourselves to remember things like this could be very helpful.*



```
to runOpMode
  Put initialization blocks here.
  set right front . Direction to Direction REVERSE
  set right back . Direction to Direction REVERSE
  set left front . Direction to Direction FORWARD
  set left back . Direction to Direction FORWARD
  call waitForStart
  if call opModelsActive
  do
  Put run blocks here.
  set Power
  right front to 0.3
  right back to 0.3
  set Power
  left front to 0.3
  left back to 0.3
  call sleep
  milliseconds 500
  set Power
  right front to 0
  right back to 0
  set Power
  left front to 0
  left back to 0
  call sleep
  milliseconds 500
  set Power
  right front to -0.3
  right back to -0.3
  set Power
  left front to 0.3
  left back to 0.3
  call sleep
  milliseconds 2500
  set Power
  right front to 0
  right back to 0
  set Power
  left front to 0
  left back to 0
```

By adjusting the speed and time, you can control 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 right!

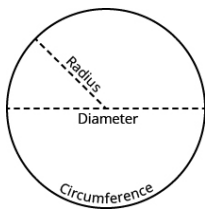
Technically speaking, we do not need the step in the middle where we set the power to 0 for 500 milliseconds. However, including this step will give our movements much greater precision. There is sometimes a tiny lag between the actions of the various wheels, and building in tiny pauses between movements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

\*There is a problem with using encoders when you have multiple motors linked to each other physically by a fixed gear or belt (as is the case with the GoBilda BeeLine chassis). When one of the motors on a side reaches its target number of clicks, it will stop. If the other motor has not reached its exact target clicks yet, it will keep trying to move to its target position, but it will be unable to get there due to the other motor physically stopping it from moving any farther. The second motor will probably be very close to its target, but won't actually get there. There are ways to get around this, but they are beyond the scope of this tutorial. We recommend NOT using motor encoders for pre-programmed motions if you are beginner using a BeeLine Chassis or another chassis where multiple motors are fixed to each other.

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](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>

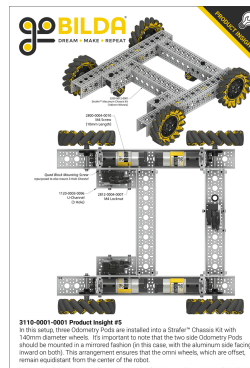
# Odometry

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 without programming your robot in Java. For those of you who are interested, get cracking on that Java! 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.

<https://vrobotsim.com/>

GoBilda Odometry Pod

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

Youtube video on Odometry for FTC: Aperture Science

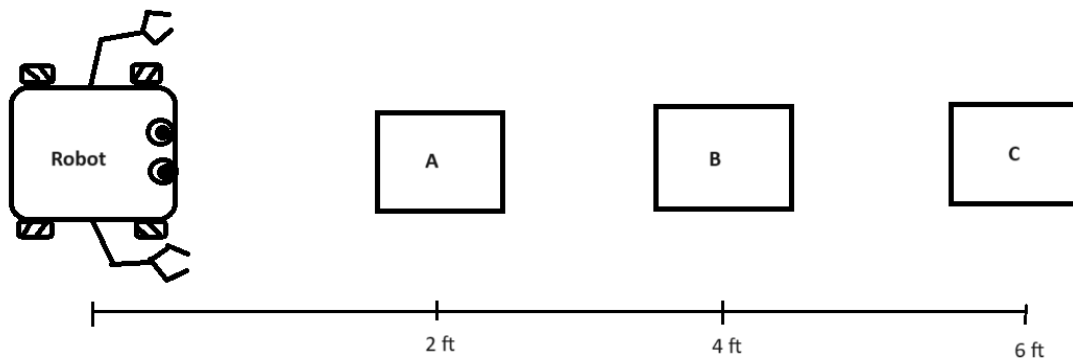
<https://www.youtube.com/watch?v=Av9ZMjS--gY>

## 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*.

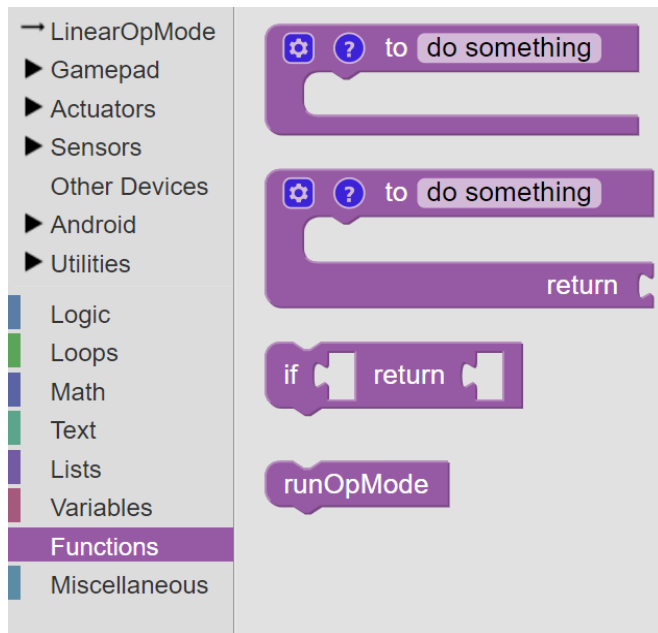
Now that I have defined my function, I can write my code like this:

### **Code Using “Dance” Function:**

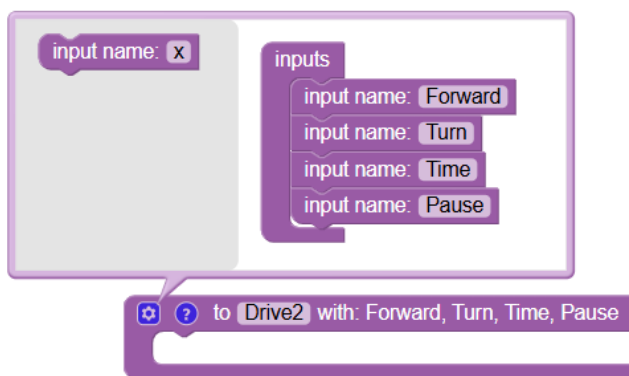
*Move forward 2 feet (location A)*→*Dance*→*Move forward 2 feet (location B)*→*Dance*, *Move forward 2 feet (location C)*→*Dance*

## Making Functions in Blocks:

- Open the “**function**” menu.
- Select the “**to something**” block (the one without the “return” slot).
- 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.”

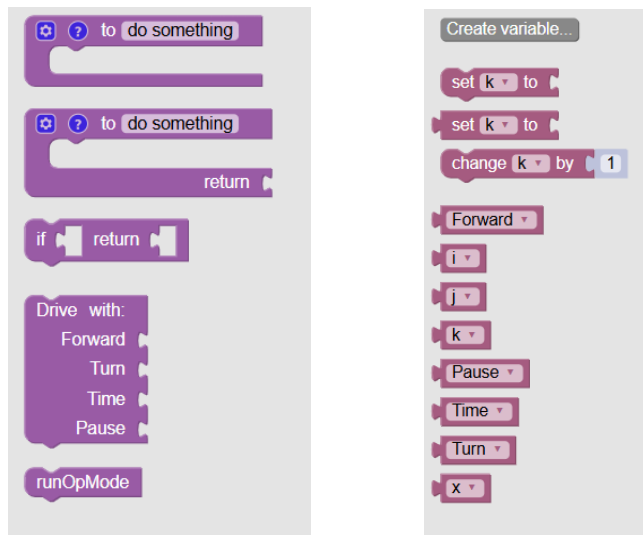


- Click on the gear icon to make your function box more complex.
- Drag the “**input name: \_\_**” block into the slot under the “**inputs**” block. You can add as many as you need to. *We added 4 in this example.*
- Close the pop up window by clicking the gear again.



- 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.



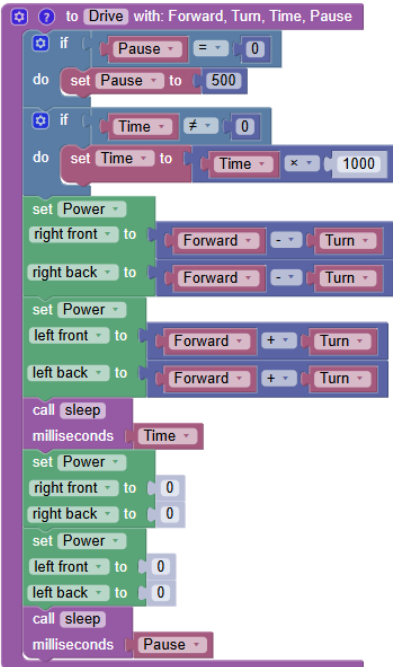
## Robotic Programming Ethics: A quick aside

\*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 will 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 TeleOp OpMode.



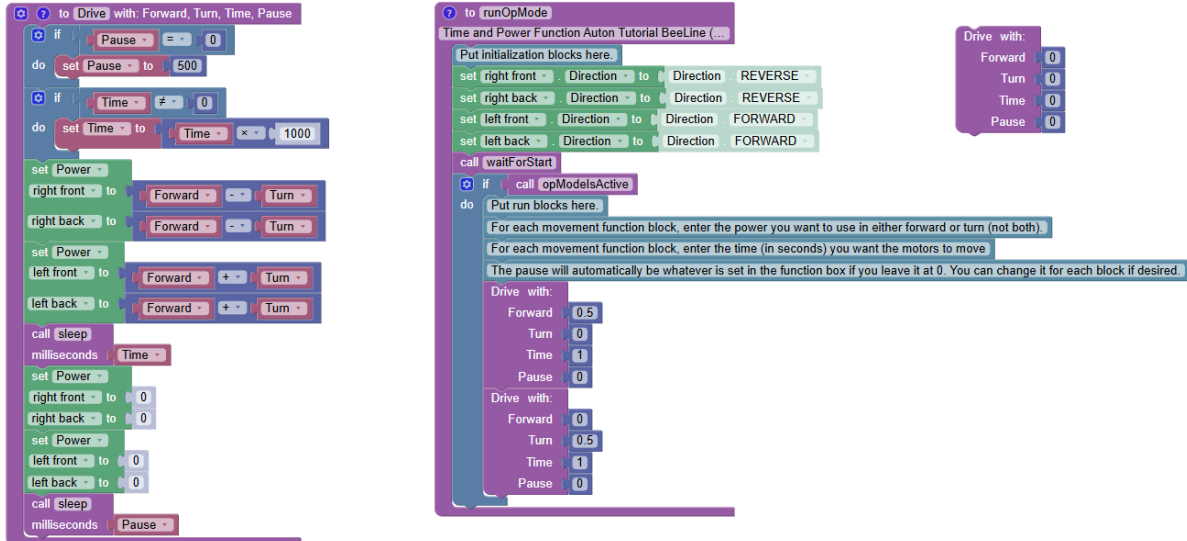
- 1) Ignore the 2 “if do” block at the top. We will come back to them in a moment.
- 2) Insert 4 “**Set motor name. Power to**” blocks. Use the drop-down menu to make sure all 4 wheels are accounted for. These are in the “**Actuators**” “**DcMotor**” menu. The double motor blocks are under the “**Dual**” submenu.
- 3) For each line after motor power, you will need to grab to “**\_\_ +/- \_\_**” blocks from the “**Math**” menu.
- 4) Grab 4 copies of the **Forward and Turn** variables from the “**Variables**” menu.
  - a) Plug these into the blank spaces in your “**\_\_ +/- \_\_**” blocks.
  - b) Adjust the +/- symbol to create the tank drive logic as seen above.
- 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 500.
  - 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 500. Our little pause at zero power will last 500 milliseconds (0.5 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.
- 10) At the top of your function, add another **“if do”** box from the **“logic”** menu.
  - a) After the “if,” add a **“\_\_ = \_\_”** block from the **“Logic”** menu.
  - b) Use the drop-down menu in the box to change the + sign to a  $\neq$  sign.
  - c) Add the **“Time”** block from the **“Variable”** menu to the first blank.
  - d) Add a blank number (**“0”**) block from the **“Math”** menu to the second blank.
  - e) Add a **“set Variable to”** block from the **“Variables”** after the “do” command.
    - i) Select **“Time”** from the drop-down menu.
    - ii) Add a **“\_\_ x \_\_”** block from the **“Math”** menu.
    - iii) In the first blank at a **“Time”** variable from the **“Variable”** menu.
    - iv) In the second blank, add a **“0”** block from the **“Math”** menu, then change it to 1000.
  - f) What we just did with this block is converted milliseconds to seconds. This block says that whenever we have a number other than “0” in the “Time” part of one of our function commands, we will multiply the number we enter times 1000. So, we can put in 1 to equal 1 second instead of 1000 to equal 1 second. The program thinks in milliseconds, but most kids don’t think in milliseconds. This is an optional step, but it might be helpful for you.

*Now you have created your Time/Power function. For each movement instruction you give your robot with this function, it will move forward, backward, turn right or turn left with the power you input and the +/- you input before the power 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, 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 Turn box will produce a **CLOCKWISE** motion.
- A negative power level in the Turn box will produce a **COUNTERCLOCKWISE** motion.

\*You will also need to input time for every function box.

The time is measured in seconds in our code now because we added a multiplier for time in our function.

*In this example, the robot will move forward with a power of 0.3 (30% of maximum) for 0.5 seconds and then stop for a 0.5 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, 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** 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 value you are not using (out of Forward and 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. It's a good idea to keep a function box with all 0 values nearby for easy copy and paste.

\*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).

```

to runOpMode
  Time and Power Function Auton Tutorial BeeLine (...)
  Put initialization blocks here.
  set right front . Direction to Direction REVERSE
  set right back . Direction to Direction REVERSE
  set left front . Direction to Direction FORWARD
  set left back . Direction to Direction FORWARD
  call waitForStart
  if call opModelsActive
  do Put run blocks here.
    For each movement function block, enter the power you want to use in either forward or turn (not both).
    For each movement function block, enter the time (in seconds) you want the motors to move
    The pause will automatically be whatever is set in the function box if you leave it at 0. You can change it for each block if desired.
    Drive with:
      Forward 0.5
      Turn 0
      Time 1
      Pause 0
    Drive with:
      Forward 0
      Turn 0.5
      Time 1
      Pause 0
    Drive with:
      Forward 0.5
      Turn 0
      Time 2
      Pause 0
    Drive with:
      Forward 0
      Turn 0.5
      Time 1
      Pause 0
    Drive with:
      Forward 0.5
      Turn 0
      Time 1
      Pause 0
    Drive with:
      Forward 0
      Turn 0.5
      Time 1
      Pause 0
    Drive with:
      Forward 0.5
      Turn 0
      Time 2
      Pause 0
    Drive with:
      Forward 0
      Turn 0.5
      Time 1
      Pause 0
  
```

*This code uses a power of 0.5 (50% of maximum for all its movements  
It pauses for 500 milliseconds (0.5 seconds) between each movement*

- It will move forward for 1 second (1000 milliseconds)*
- It will turn right/clockwise for 1 seconds*
- It will move forward for 2 seconds*
- It will turn right/clockwise for 1 second*
- It will move forward for 1 second*
- It will turn right/clockwise for 1 second*
- It will move forward for 2 second*
- It will turn right/clockwise for 1 second*
- It will then stop*

*\*It should end up exactly back where it started. Though, accuracy is often a problem when using time and power.*