

Using Motor Encoders in Autonomous OpModes

When using FTC Blocks, encoders are one of the most useful tools for programming motors in Autonomous code. What makes them so hard is that the math needed to convert encoder counts into something that is useful for reliable, consistent robot movement can be difficult for beginning programmers. Additionally, using encoders requires an understanding of how to properly sequence commands, and how to use the tools within the Blocks editor to ensure those commands are completed. This lesson, while focused on using FTC Blocks to create robot movement, is more about the math principles needed to effectively use encoders, and how to leverage math *within* FTC Blocks to create code. Using this lesson, you will learn how to create a set of FTC Blocks *functions* that will let you build simple sets of movement instructions to move a robot forward and backwards, AND to rotate a robot.

What is an encoder:

Motor encoders are electronic counters that measure the rotation of a motor's shaft. All an encoder does is count, either forward or backwards depending on the direction a motor is rotating. For every motor/ encoder combination, the encoder will count a specific number of counts for each motor shaft rotation. Since our team uses REV motors, I will provide the counts/ rotation for both the REV Ultrapanetary kit/ HD Hex motor, and the REV Core hex motor below:

REV Ultrapanetary kit/ HD Hex motor: 28 counts per revolution (without any planetary gear stacks added)

REV Core hex motor: 288 counts per revolution

When is the encoder used in FTC Blocks:

Motors in FTC blocks can be used in various *modes*. These modes are set using the `set.motorname.Mode` command as seen below.



There are 4 different modes a motor can be set to:

- 1) **Stop and reset encoder**- this mode stops the motor and resets the encoder position to a value of 0. This command is useful at the beginning of an OpMode to ensure that motors start from a known baseline condition and can also be used within an OpMode

Using Motor Encoders in Autonomous OpModes

to either restart your counting, or to help troubleshoot when you are experiencing unusual robot behaviors.

- 2) **Run Without Encoder**- this mode ignores the encoder completely. In this mode, the `set.motorname.Power` command scales power between -1.0 and 1.0 (full reverse to full forward power) based on the maximum available *power/ voltage* the robot controller is able to provide. This mode is easy to use, and is commonly used in Teleop OpModes. The disadvantage of this mode is that as the robot's battery voltage drops, the maximum speed/ available power drops as well, making motor speed less consistent than encoder driven modes.
- 3) **Run WITH encoder**- This mode unlocks the counter functionality of the encoder, as well as the ability to set a *velocity* of a motor (in counts/ second), and a maximum velocity that the motor is allowed to operate at. In this mode, the `set.motorname.Power` command scales from -1.0 to 1.0 to the defined maximum motor velocity/ speed, and adjusts motor current while running to maintain that speed. This allows for more consistent control over robot speed, and allows a programmer to use the encoder counts as variable inputs to other commands. This mode additionally applies PIDF coefficients to the motor's operation, which when properly defined and tuned can make motion changes smoother for better precision in robot movement.
- 4) **Run to Position**- This is a special mode that allows a programmer to define a specific *target* encoder position for the motor to run to. This mode is ideal for Autonomous coding, as you can use math to define targets that allow you to precisely position your robot without direct control. The major disadvantage of this mode is that it is more complex than Run Without Encoder, and the math required to determine proper encoder targets can be difficult for younger or less experienced programmers to understand. This is the mode that our main example today will use to create a fully functional.

Using Run to Position-

In order to effectively use the Run to Position mode, you need four specific commands:

- 1) You must first set a *target position*. This is the target (in encoder counts) that you want the motor to drive to. To do this, use the `set.motorname.TargetPosition` block as seen below.



Using Motor Encoders in Autonomous OpModes

When using this block, be sure to add a number block (or a reference to a variable) to the end. This is what defines your target encoder position.

- 2) The next step is to place a `set.motorname.Mode.Runmode.Run_to_Position` block as seen below. **Make sure you specify the correct motor.**



A screenshot of a LEGO Mindstorms block. The block is titled "set" and has a dropdown menu for "mRight". The "Mode" dropdown is set to "RunMode" and the "Runmode" dropdown is set to "RUN_TO_POSITION".

- 3) The next step is to set a power or velocity for the motor using either the `set.motorname.Power` or `set.motorname.velocity` command. In general, I've found that the most consistent results for beginners come with setting Power versus Velocity, though our main example program will use Velocity in order to take advantage of one of its special features (the ability to set a maximum rotational speed). To do this, place a `set.motorname.Power` block as seen below:

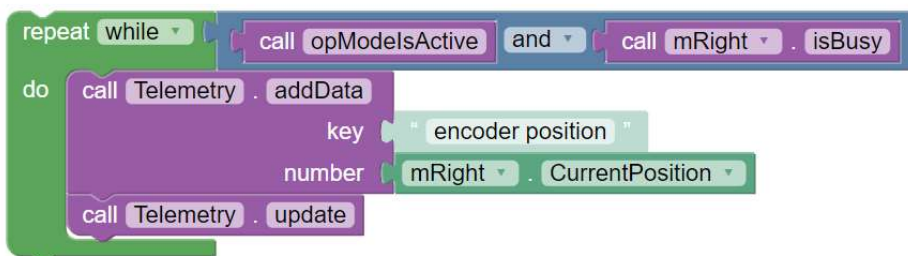


A screenshot of a LEGO Mindstorms block. The block is titled "set" and has a dropdown menu for "mRight". The "Power" dropdown is set to "0".

When using this block, ensure you attach a number or variable block with a value of somewhere between -1.0 and 1.0, depending on which direction you want the motor to drive. Keep in mind that if your target position is larger than your current position, you **MUST** drive the motor forward to increase encoder counts, and the opposite is true when it comes to going in reverse.

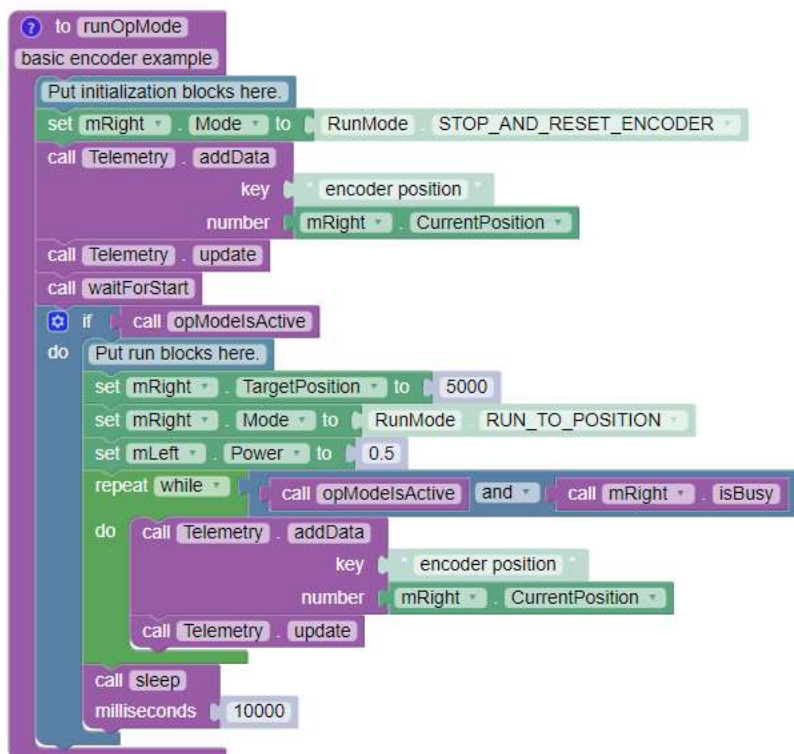
- 4) The final step while not required, is almost always the easiest way to ensure that you actually drive the motor all the way to the target. The problem many teams run into is that as soon as the `set.motorname.Power` command is executed, the program will automatically step to the next command. In many cases, the next set of commands will either end the program, or tell the motor to do something else entirely. To prevent this you can insert a *while* loop, that looks for 2 things, verify that the OpMode is still running, and verify that the motor is *Busy*. While a motor is running to a target, it considers itself *busy*. With the *while* loop in place, the program will check if that motor is still running to its target, over and over until the motor has stopped at its target. Generally, I place a telemetry block into this while loop that shows the encoder value for the motor as it moves to its target position. Here is an example of the *while* loop below:

Using Motor Encoders in Autonomous OpModes



```
repeat while [ ] {
  call opModelsActive and call mRight . isBusy
do {
  call Telemetry . addData
  key encoder position
  number mRight . CurrentPosition
  call Telemetry . update
}
```

This combination of blocks, when placed in an OpMode, will set a target encoder position, place the motor in the Run_to_Position mode, and drive the motor until the target position is met. Below is an example of an OpMode that will do just that. I've also added an additional set of telemetry blocks to provide feedback to the driver's station after initially resetting the encoder, as well as a time delay at the end of the OpMode to give the operator time to verify that the motor has driven to the correct position.

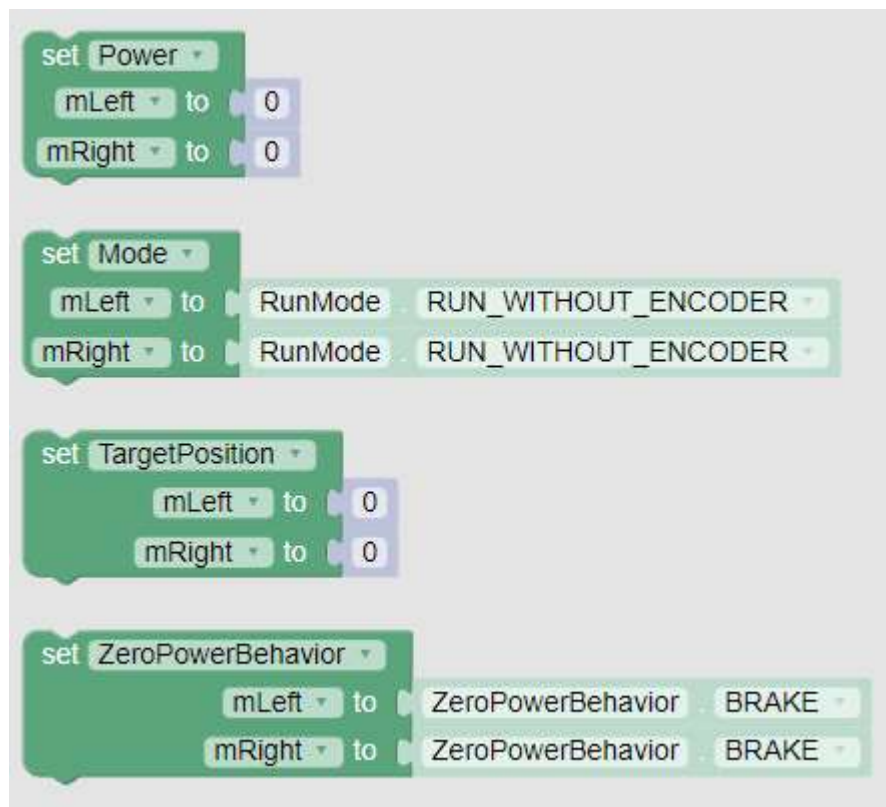


```
to runOpMode {
  basic encoder example
  Put initialization blocks here.
  set mRight . Mode to RunMode STOP_AND_RESET_ENCODER
  call Telemetry . addData
  key encoder position
  number mRight . CurrentPosition
  call Telemetry . update
  call waitForStart
  if call opModelsActive {
  do {
    Put run blocks here.
    set mRight . TargetPosition to 5000
    set mRight . Mode to RunMode RUN_TO_POSITION
    set mLeft . Power to 0.5
    repeat while [ ] {
      call opModelsActive and call mRight . isBusy
    do {
      call Telemetry . addData
      key encoder position
      number mRight . CurrentPosition
      call Telemetry . update
    }
    call sleep
    milliseconds 10000
  }
}
```

Using Motor Encoders in Autonomous OpModes

Using Dual Motors:

Generally speaking, if you want to drive a robot you need at least 2 motors, one on each side of the robot. To make it easier to use 2 motors at once, Blocks provides a set of 'Dual' motor commands, which let you set Mode, Target Position, Zero Power Behavior, and Power for both motors in a single block. As a note, the dual Power block also has a drop down that will allow you to set Velocity. Below are examples of the dual commands available in Blocks:



Understanding and converting Gear Ratios in Blocks:

We now know how to set a motor's mode and rotate a motor until its encoder hits a target position, but most of the time our drive motors don't have enough torque to move a robot without help. As a rule, teams use either a planetary gear motor (such as the REV ultraplanetary HD Hex) or a spur gear motor to create enough torque to move larger objects. While I'm not going to go deeply into gear theory in this lesson, it's important to understand one significant concept. A gear assembly converts rotational speed into torque. In effect, a gear assembly torque at the output shaft by reducing speed. As an example, if I have a gear assembly with a 10:1 ratio, the motor will spin 10 times for each rotation of the output shaft. This lowers our maximum shaft speed but allows for more force to be applied at the wheels. Below is a chart of

Using Motor Encoders in Autonomous OpModes

the various combinations of gear ratios that the REV ultrapanetary kit can obtain by stacking two the available gear stacks on the motor shaft (note that the empty blocks are there for a reason, you never want to place a higher gear ratio stack on top of a lower gear ratio stack, as this increases the chance of damaging either the gear stacks or the motor):

Stage 1	Stage 2		
	3:1	4:1	5:1
3:1	8.4	*	*
4:1	10.5	13.1	*
5:1	15.2	18.9	27.4

If you notice that these values seem uneven, it's because of the type of gearing that the REV motors use. More important is to understand how gear ratios impact encoders. Once again using the REV kit as an example, if each rotation of the motor results in 28 encoder counts, and assuming you have a 5:1 & an 4:1 gear stack on your kit, the total number of encoder counts for a single rotation of the output shaft (and correspondingly the wheel) is **28 X 18.1 (or 506.8)**, because the motor rotates 18.1 times for **each** single rotation of the output shaft. The reason I am using this particular combination of gear stacks is that this is a common gear ratio to use, as it provides adequate torque for drive motors in most situations while still providing enough speed to be useful.

Unfortunately, Blocks doesn't simply know what your gear ratio is, so you must create code to calculate how many counts will result in a single rotation of the wheel (this will be **very** important later!!!!). The simple way to do this is to create a variable (in this case I'm calling it *shaft counts per rev*) and take it's result as motor counts per revolution multiplied by the gear ratio. In our case we can do this in one of two ways:

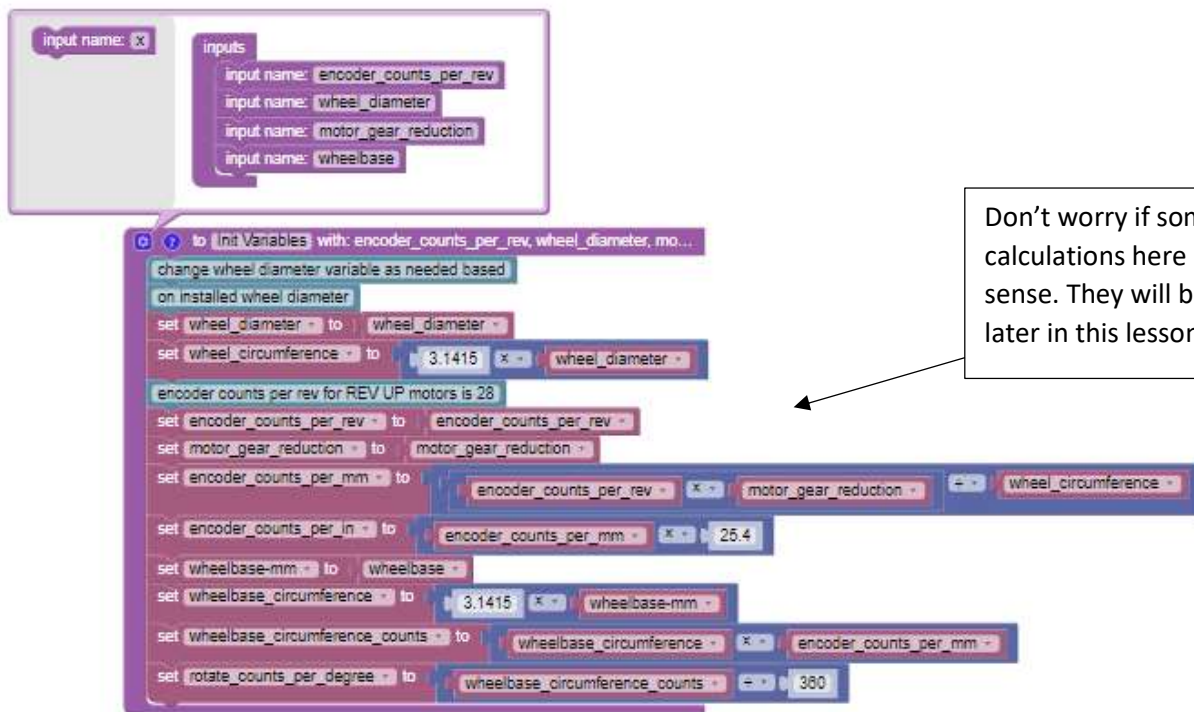


OR

Using Motor Encoders in Autonomous OpModes

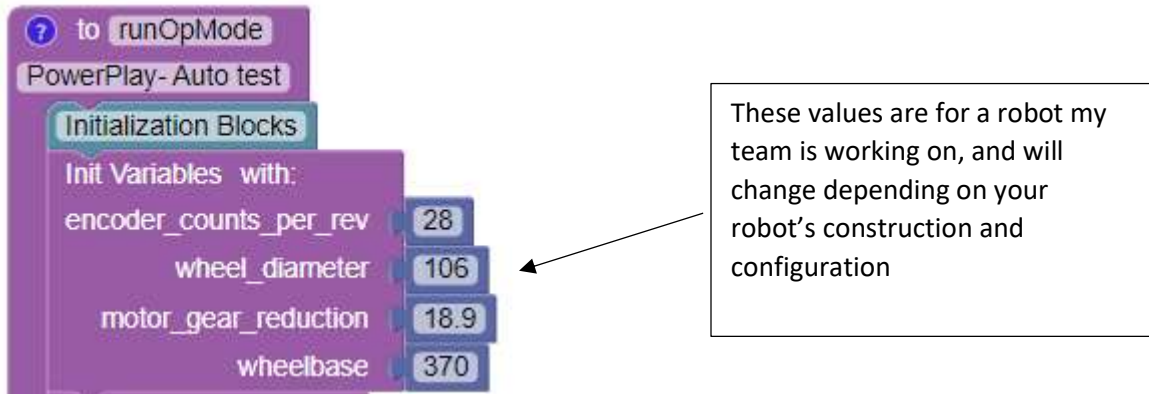


In the second example, we create 2 new variables, *encoder_counts_per_rev* and *motor_gear_reduction*. The reason we might want to do it this way is if there is any possibility that we might change our hardware (different motor with a different encoder count, or changing our gear ratio), we only need to change the values of our two variables, which are labeled easily and are therefore easier to find in our code. One thing I prefer to do is to set up all of my variables when I initialize the robot, and to place all of them into a special function I call *initVariables* (or *initialize variables*). This puts every possible variable I may need to change in one place that is easy to find. For variables that I might need to change as a part of my robot build process, I can also place these variables as *inputs* to my *initVariables* function. This provides an even simpler way identify and tune my variables. Here is one example of the *initVariables* function that I am using for this lesson, as well as what it looks like on the Main code string:



This function takes my inputs (counts per rev, wheel diameter, gear ratio, and wheelbase width), and uses those values to mathematically derive a number of other variables that I will need to drive the robot in an easy and consistent manner.

Using Motor Encoders in Autonomous OpModes



This is what that function looks like when I call it in my main OpMode code. If you notice, I can now very easily adjust all those inputs, making my code simple to adjust if I either change my hardware, or decide to use the same code for a different robot.

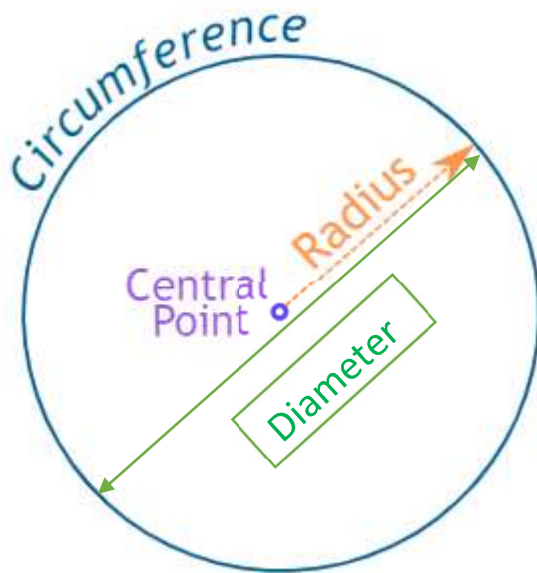
Geometry concepts important to robotics (AKA Circle Math 101):

While it's certainly possible to simply identify the correct encoder counts needed to move your robot to a desired location through trial and error, it's better from an engineering standpoint to use the tools available to you to get from point 'A' to point 'B' correctly the first time. To do this, you first need to understand some basic math terms and equations.

If you take a circle, there are 3 important characteristics that matter to you for programming purposes.

- 1) Radius- The radius of a circle is the distance from the center point to the outside edge.
- 2) Diameter- the Diameter of a circle is the distance across the circle in a straight line going through the center point. The Diameter will always be twice the length of the radius.
($D=2r$)
- 3) Circumference- The Circumference of a circle is the distance around the outer edge. Mathematically, you can find the circumference of any circle by multiplying pi (3.1415.....) by the diameter.

Using Motor Encoders in Autonomous OpModes



From this information, there are several math equations that are useful:

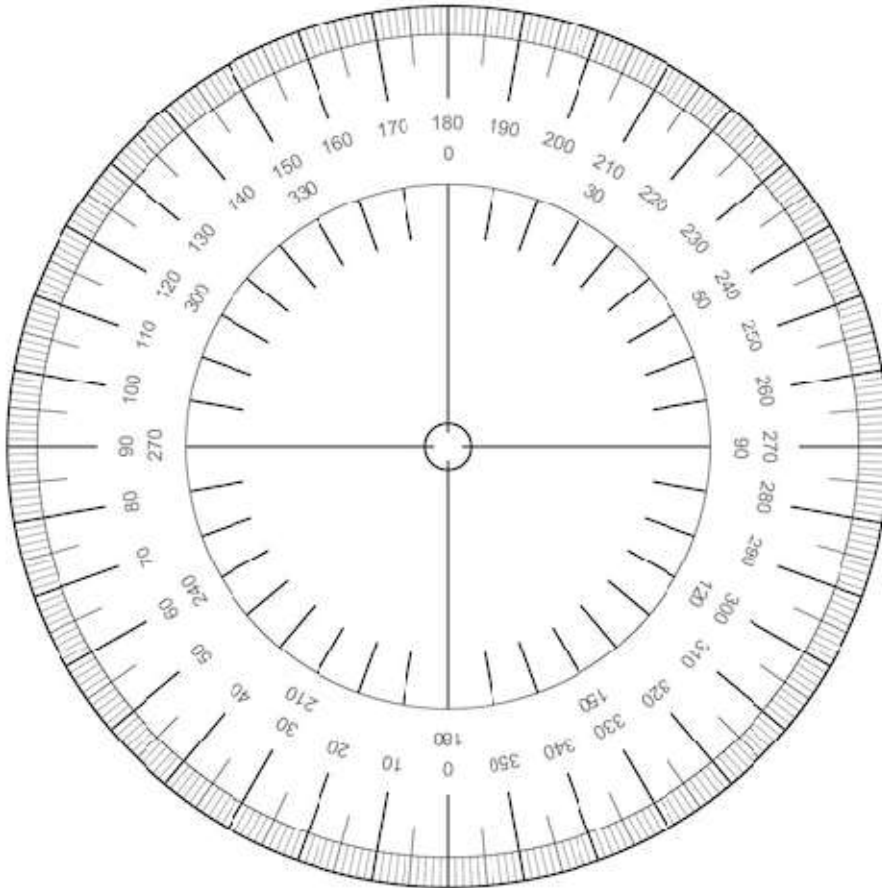
$$\pi (pi) = 3.1415 \dots$$

$$diameter = 2 \times radius$$

$$circumference = \pi \times diameter$$

There is one additional concept that you need to know about circles that will prove important to utilizing math to simplify robot programming. For any given circle, you can divide it into exactly 360 slices (think of it like a pizza with a LOT of slices). This is one of the core concepts of modern geometry.

Using Motor Encoders in Autonomous OpModes



Converting encoder counts into distance travelled:

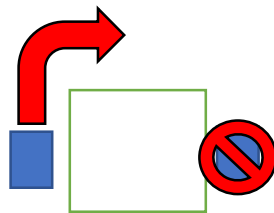
We now have all of the tools needed to create a `run_to_position` OpMode that can be told how 'far' to go versus how 'many encoder counts'. To do this, first we need some information about our robot. We will need the diameter of our wheels (can be found in the vendor data sheet for the specific wheels), our encoder counts per revolution (also available in the vendor data sheet), and the width of our wheelbase (the distance between the center of our two drive wheels). Once we have that information, we can now derive everything we need within our `initVariables` Function as follows:

- 1) Assign a value to our `wheel_diameter` variable.

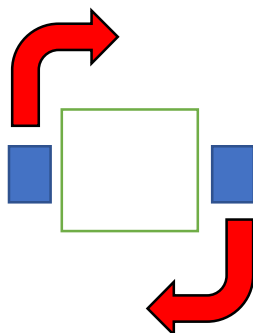
Using Motor Encoders in Autonomous OpModes

- 2) Calculate our `wheel_circumference` variable by multiplying the `wheel_diameter` by pi. This tells us how far the wheel will roll for one complete rotation (very important concept).
- 3) Assign values to our `motor_gear_reduction` and `encoder_counts_per_rev` variables.
- 4) The next step is tricky. We want to determine how many encoder counts will tick for each mm of travel (we are using mm because that is the unit of measure that the wheel diameter is in. Technically we can use any unit of measure we want, but all of the following steps would need to be modified.) To calculate this we will multiply our `motor_gear_reduction` and `encoder_counts_per_rev` together, then divide the result by our wheel circumference. The final result is then assigned to the `encoder_counts_per_mm` variable.
- 5) Next, we will perform a conversion to allow us to use inches as our unit of travel (the play field is 12 feet to a side, and determining travel by mm doesn't really make sense). To do this, we will multiply our `encoder_counts_per_mm` variable by 25.4 (there are 25.4 mm in an inch). The result is assigned to the `encoder_counts_per_in` variable. This is the variable that will be used for forward and backward movement.
- 6) We will now begin calculating our variables needed for robot rotation. For this lesson, we will use the terms turning and rotating to mean different methods of changing a robot's direction. Rotating is using both wheels turning in opposite directions to change the robot's heading. Turning is using a single wheel moving either forward or backward to change the robot's heading, while the other drive wheel remains stationary.

This is a turn:



This is a rotation:



Using Motor Encoders in Autonomous OpModes

Both turns and rotations can be useful, but for our sample code we will only set up rotation. To begin setting our robot up for rotation, we will assign a value to our *wheelbase_mm* variable that is equal to the width of the robot between the centerpoint of both drive wheels.

- 7) We will now calculate our wheelbase circumference (imagine if your two drive wheels were moving in opposite directions, this is the circumference of the circle that would be created by their paths.) To do this (same as when calculating the *wheel_circumference* variable), we will multiply the *wheelbase_mm* variable by pi (3.1415...) and assign the result to the *wheelbase_circumference* variable.
- 8) We will now calculate how many encoder counts are equivalent to our wheelbase circumference. Do this by multiplying our *wheelbase_circumference* variable with our *encoder_counts_per_mm* variable that we calculated in step 4. Assign the result to the *wheelbase_circumference_counts* variable.
- 9) Our last step in setting up our initVar function is to determine how many counts of rotation are needed to rotate the robot by 1 degree. Remember that a full circle has 360 degrees. This value is obtained by dividing our *wheelbase_circumference_counts* variable by 360, and assigning the result to the *rotate_counts_per_degree* variable.

At the end of this process, you should have a function that looks something like this:

(There might be some differences, depending on if you decide to use function inputs for some of your variable values, or if you choose to use different variable names.)

```
to [Init Variables] with: encoder_counts_per_rev, wheel_diameter, mo...
change wheel diameter variable as needed based
on installed wheel diameter
set wheel_diameter to wheel_diameter
set wheel_circumference to 3.1415 x wheel_diameter
encoder counts per rev for REV UP motors is 28
set encoder_counts_per_rev to encoder_counts_per_rev
set motor_gear_reduction to motor_gear_reduction
set encoder_counts_per_mm to encoder_counts_per_rev x motor_gear_reduction ÷ wheel_circumference
set encoder_counts_per_in to encoder_counts_per_mm x 25.4
set wheelbase-mm to wheelbase
set wheelbase_circumference to 3.1415 x wheelbase-mm
set wheelbase_circumference_counts to wheelbase_circumference x encoder_counts_per_mm
set rotate_counts_per_degree to wheelbase_circumference_counts ÷ 360
```

Using Motor Encoders in Autonomous OpModes

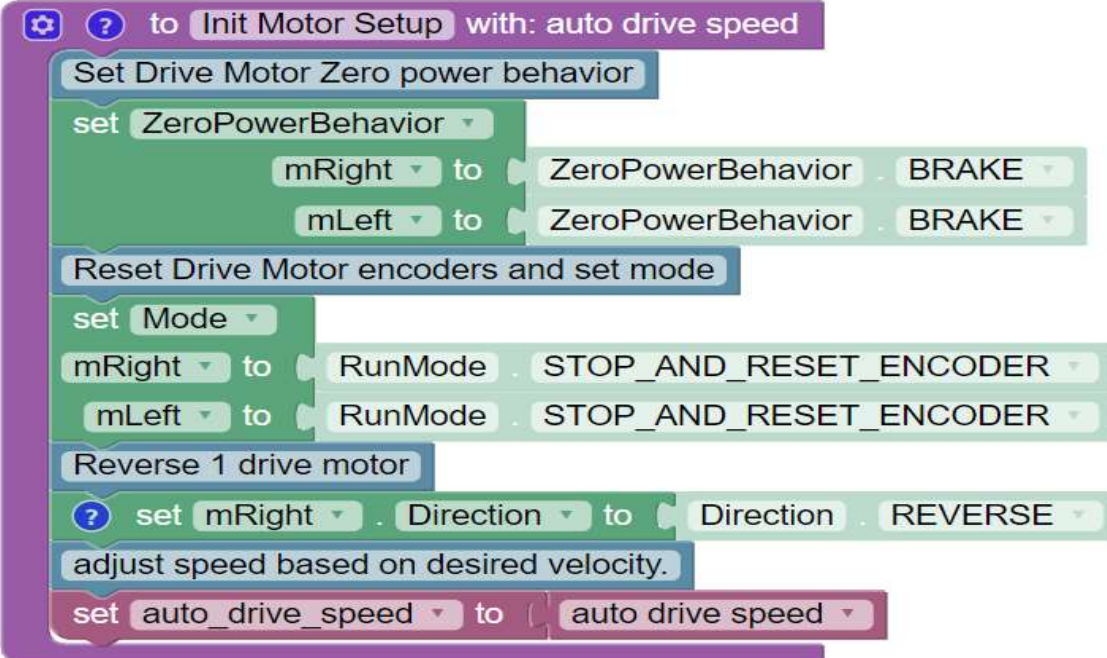
Setting up our motors:

Our next task is to set up our motors. Once again, I am going to use a function for this, as I like to keep my main code as clean as possible. This function will be named *Init Motor Setup*. This function will be much less complicated than our Init Variables Function.

To setup our motors we need to do several things. We need to assign our zero power behavior, reset our encoders to an initial value of 0, and we need to reverse the direction of one of our motors. The reason we reverse one of our motors is that in most robots, the 2 drive motors are pointed in opposite directions. Depending on the motor type and wiring, you may need to experiment to determine which motor needs to be reversed. In the example below, we have to reverse our Right side motor.

The last line of code, takes an input variable, and assigns it's value to a variable called `auto_drive_speed`. This program uses the motor's velocity instead of setting the Power value. Either method will work, but in this case, I'm setting up a very specific value that I can change to adjust my robot's maximum speed while keeping both the scaling of robot speed linear, AND minimizing the number of variables I need to change if I want to slow down (or speed up) my robot. If I later choose to change my program to use Power instead of velocity, the motor setup function will not need to be changed, but the value I put into my auto drive speed variable in the function input will.

Here is my completed Init Motor Setup function:



```
to Init Motor Setup with: auto drive speed
  Set Drive Motor Zero power behavior
  set ZeroPowerBehavior
    mRight to ZeroPowerBehavior BRAKE
    mLeft to ZeroPowerBehavior BRAKE
  Reset Drive Motor encoders and set mode
  set Mode
  mRight to RunMode STOP_AND_RESET_ENCODER
  mLeft to RunMode STOP_AND_RESET_ENCODER
  Reverse 1 drive motor
  set mRight . Direction to Direction REVERSE
  adjust speed based on desired velocity.
  set auto_drive_speed to auto drive speed
```

Using Motor Encoders in Autonomous OpModes

Just a couple of notes on this sample:

- 1) If you see the 4 blue blocks with writing in them, these are comment blocks used to describe what each command is intended to do.
- 2) This function is also where you can place PIDF velocity and position coefficients, if you so choose. These are an advanced method of controlling motor behavior that can be used to smooth out changes in motor speed (this minimizes the chance of overshooting a target position or damaging a motor) that we will not discuss in this lesson.

Creating a forward/ reverse drive function:

Now that we have all of our variables initialized, and our motors set up, it is time to create a function that will actually move the robot. This function has been designed in a way that it will move the robot ANY number of inches either forward or backward, depending on the distance you input when you place the function call within your main code. This results in a much more versatile program, with less need for changes when a code improvement is identified and fewer code blocks to troubleshoot if you find a problem.

If you remember earlier in this lesson, every `run_to_position` sequence has 3 required, and 1 optional command blocks needed to work:

- 1) A call to set a target position
- 2) A command to set the mode on the motor(s) to `run_to_position`.
- 3) A command to set either the power or the velocity of the motor
- 4) (optional, but recommended) A While loop to pause program flow until the desired movement has been completed.

I've also included a number of non-required commands like telemetry and a specific command block to ensure the motors have stopped prior to exiting the function. Let's look at the sample function, and walk through how it works:

Using Motor Encoders in Autonomous OpModes

```
to drive with: Distance
  set target_right to mRight.CurrentPosition + Distance * encoder_counts_per_in
  set target_left to mLeft.CurrentPosition + Distance * encoder_counts_per_in
  set TargetPosition
    mRight to target_right
    mLeft to target_left
  set Mode
  mRight to RunMode RUN_TO_POSITION
  mLeft to RunMode RUN_TO_POSITION
  set Velocity
  mRight to auto_drive_speed
  mLeft to auto_drive_speed
  repeat while call opModelsActive and call mRight.IsBusy or call mLeft.IsBusy
  do call Telemetry.addData
    key left encoder pos
    number mLeft.CurrentPosition
  call Telemetry.addData
    key right encoder pos
    number mRight.CurrentPosition
  call Telemetry.update
  set Velocity
  mLeft to 0
  mRight to 0
  call Telemetry.addData
    key left encoder pos
    number mLeft.CurrentPosition
  call Telemetry.addData
    key right encoder pos
    number mRight.CurrentPosition
  call Telemetry.update
```

- 1) The function includes a single input parameter: Distance. This parameter is the distance (in inches) that you want the robot to move. A positive number will move the robot forward, while a negative number will move the robot backward.
- 2) Two variable assignments for target_right and target_left.

```
set target_right to mRight.CurrentPosition + Distance * encoder_counts_per_in
set target_left to mLeft.CurrentPosition + Distance * encoder_counts_per_in
```

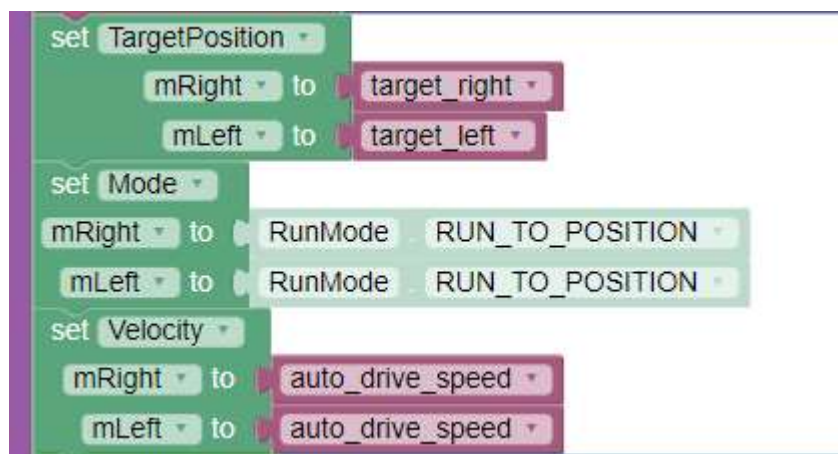
These variable assignments DO NOT set the target position, they only calculate a value, and then store that value in the two variables. We determine the value by first polling the encoder for the *motorname.CurrentPosition* value. This is the value of the encoder PRIOR TO moving the motor. We use this, instead of a value of 0 because for many OpModes, we will use multiple drive and rotate functions to perform movement, and

Using Motor Encoders in Autonomous OpModes

we can't assume that the encoder will start at zero every time. While it might be possible to simply reset the encoders after every movement, it's neither necessary nor efficient. We then multiply our desired movement distance with the value of the *encoder_counts_per_in* variable we calculated to give us a total number of counts by which we want to move. This value can be either positive OR negative, depending on which direction we want to go.

We then add together our current position and the desired change in position, and then store the two values in our target variables.

- 3) Our next 3 sets of blocks should be familiar.
 - a. A dual set.TargetPosition block (with the value being the two calculated target_right and target_left variables we just calculated).
 - b. A dual set.Mode.run_to_position block to set the motors into the correct mode.
 - c. A dual set.Velocity block that starts the motors at the speed designated by the auto_drive_speed setting we defined when we set up the motors.

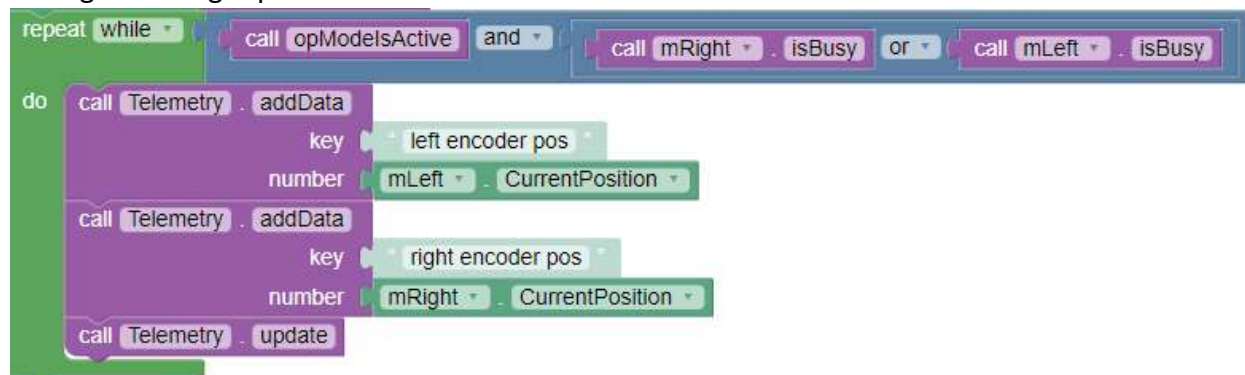


```
set TargetPosition
  mRight to target_right
  mLeft to target_left

set Mode
  mRight to RunMode RUN_TO_POSITION
  mLeft to RunMode RUN_TO_POSITION

set Velocity
  mRight to auto_drive_speed
  mLeft to auto_drive_speed
```

- 4) Next we have our *while* loop that we run to prevent the motors from stopping prior to hitting their target positions.



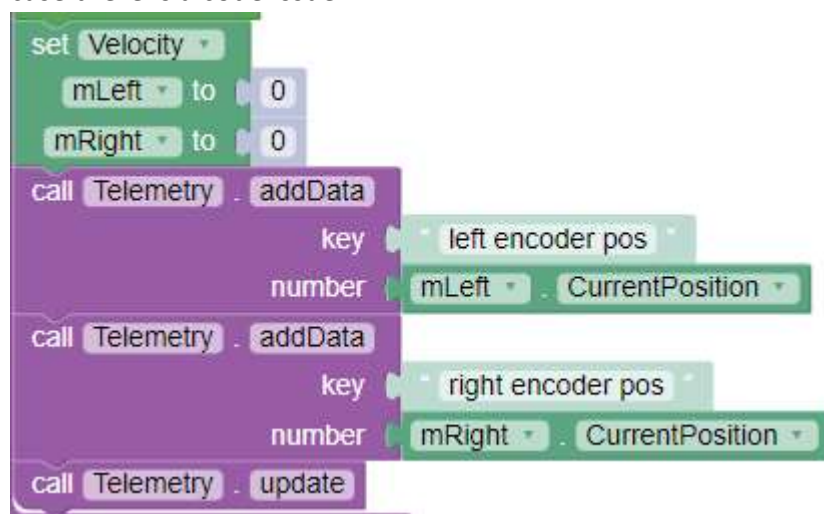
```
repeat while
  call opModelsActive and call mRight isBusy or call mLeft isBusy
do
  call Telemetry.addData
    key left encoder pos
    number mLeft.CurrentPosition
  call Telemetry.addData
    key right encoder pos
    number mRight.CurrentPosition
  call Telemetry.update
```

Using Motor Encoders in Autonomous OpModes

If you look at the logic for the while loop, you can see that it's looking for 3 conditions: The OpMode is still running (call `opModelsActive`), and if either of the two motors is busy. The important factor here is that the loop ensures that BOTH motors have hit their target positions (the motor is considered busy until it hits its target) prior to exiting the loop. This ensures that if you have a situation where one of the motors is running a bit slower than the other, you will still have even movement at the end of the loop.

Within the *while* loop, I've placed additional encoder telemetry blocks, as well as a telemetry update. This lets the user verify that the motors are progressing to their target (useful for troubleshooting either code issues or hardware problems.)

- 5) The last few blocks in our program aren't required but have been put in in order to ensure the motor has stopped successfully, and to provide additional telemetry. Remember, once the *run_to_position* command sequence has completed, the motors should have stopped on their own, this final *set.Velocity* to 0, is just a safety feature in case there is a code issue.



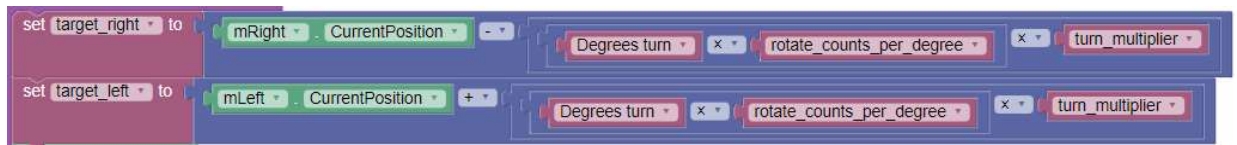
We have now completed our basic forward/ backward drive function.

Rotating the robot:

Now that we have a good template for forward and backward motion, we need to build a function that will allow us to rotate the robot (most of the time, the game doesn't just want you moving in a straight line.) This function will be VERY similar to our normal drive function, but with a few key changes. For simplicity sake, we will only go into detail on the block that is different from the drive Function.

- 1) Our first blocks (much like the drive function) assign values to our target variables for both motors.

Using Motor Encoders in Autonomous OpModes



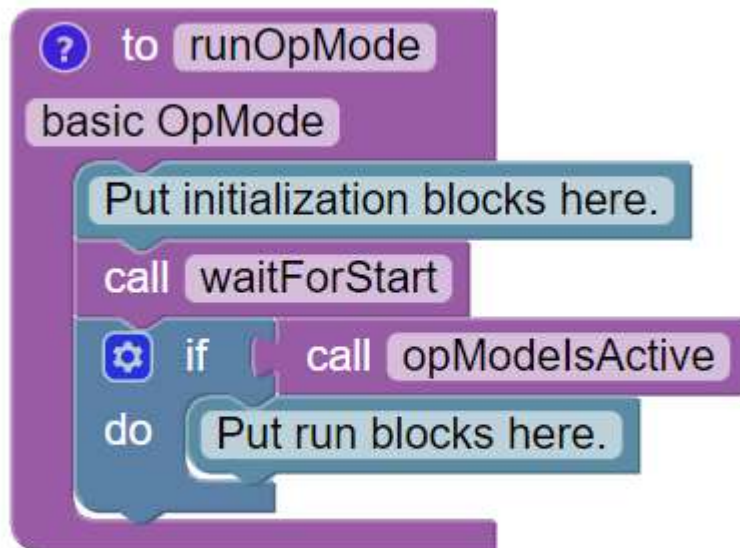
This is the only block that is different between the drive and rotate functions, but those differences are important.

- a) The first change is that instead of multiplying a distance variable by our *counts_per_in* variable value, we will multiply our desired turn value (in degrees) by the *rotate_counts_per_degree* variable we calculated in our *InitVariables* function. This provides us with a number of counts that we need to rotate each wheel.
 - b) The next change, if you look closely is that we have added an additional *turn_multiplier* factor, that adjust our number of counts we want to turn by. This isn't a requirement, but I've discovered that depending on your hardware configuration (number of wheels, type and placement), sometimes a rotation will either consistently under or overshoot its target. This value is obtained from a manually set variable at the beginning of the program (you *could* set it in the *initVariables* function, but for some reason it wasn't working for me), and is determined through trial and error, but will generally be between 0.8 and 1.2 if the rest of your programming is correct. **This value is only needed if you ALWAYS undershoot or overshoot your turns!!!! If your forward/ backward movement is inaccurate as well as your turns, or if you just sometimes overshoot or undershoot, this multiplier will not help you, check the rest of your code for errors!!!!** If you DO consistently under or overshoot your turns, experiment with different values for the *turn_multiplier* variable until your turns are consistently correct.
 - c) Our final change is in how we apply our calculated movement against the *motorname.CurrentPosition* value. Remember, if I want to rotate I need to drive one motor forward and the other backward. This means that one of your values will be added to the current position (resulting in forward motor rotation), and the other subtracted (resulting in reverse motor rotation). Which motor to reverse will depend on how your robot is set up, and you may need to experiment to determine which motor to reverse in this block.
- 2) The rest of the function is identical to the Drive function.

Putting it all together in the main OpMode:

Now that we have all of our needed functions written, we need to use them in our main OpMode. At this point let's review the basic flow of an OpMode:

Using Motor Encoders in Autonomous OpModes



In this sample, I've removed everything except the basic code flow, including the *while* loop that Teleop code uses to iterate through inputs and outputs. In this basic flow you have two main periods, the Init phase, and the Run phase. The init phase code runs after the Init button is pressed on the driver's station, and then the code stops when the code hits the `call.waitForStop` command block. Once the operator presses the start button on the driver's station, the rest of the code in the program runs sequentially.

Now let's look at our completed program. For this example, I'm including 3 movements. A forward movement of 24 inches, a 90 degree right turn and another forward movement of 24 inches. At the end of the program, I've placed a 10 second pause in order to give the operator an opportunity to verify that the encoders have moved correctly.

Using Motor Encoders in Autonomous OpModes



If you will notice, all our functions with inputs give you the opportunity to provide input values using number (or text, if the variable is a text type) blocks within the main line code. This makes it simple to make quick adjustments to your code or troubleshoot problems. For this example, I had my *turn_multiplier* variable set to 1.2 (as previously stated, this variable needs to be tuned to your application, so don't assume YOU will need this value).

Using Motor Encoders in Autonomous OpModes

We now have a fully functional autonomous program that can be set by any user to move a robot to any location on the play field. The rest of this guide will be our full program, including all its relevant functions.

```
to Init Variables with: encoder_counts_per_rev, wheel_diameter, mo...
change wheel diameter variable as needed based
on installed wheel diameter
set wheel_diameter to wheel_diameter
set wheel_circumference to 3.1415 x wheel_diameter
encoder counts per rev for REV UP motors is 28
set encoder_counts_per_rev to encoder_counts_per_rev
set motor_gear_reduction to motor_gear_reduction
set encoder_counts_per_mm to encoder_counts_per_rev x motor_gear_reduction + wheel_circumference
set encoder_counts_per_in to encoder_counts_per_mm x 25.4
set wheelbase-mm to wheelbase
set wheelbase_circumference to 3.1415 x wheelbase-mm
set wheelbase_circumference_counts to wheelbase_circumference x encoder_counts_per_mm
set rotate_counts_per_degree to wheelbase_circumference_counts + 360
```



```
to Init Motor Setup with: auto drive speed
Set Drive Motor Zero power behavior
set ZeroPowerBehavior
mRight to ZeroPowerBehavior BRAKE
mLeft to ZeroPowerBehavior BRAKE
Reset Drive Motor encoders and set mode
set Mode
mRight to RunMode STOP_AND_RESET_ENCODER
mLeft to RunMode STOP_AND_RESET_ENCODER
Reverse 1 drive motor
set mRight Direction to Direction REVERSE
adjust speed based on desired velocity
set auto_drive_speed to auto drive speed
```

Using Motor Encoders in Autonomous OpModes

```
to drive with: Distance
  set target_right to mRight.CurrentPosition + Distance * encoder_counts_per_in
  set target_left to mLeft.CurrentPosition + Distance * encoder_counts_per_in

  set TargetPosition
    mRight to target_right
    mLeft to target_left

  set Mode
  mRight to RunMode RUN_TO_POSITION
  mLeft to RunMode RUN_TO_POSITION

  set Velocity
  mRight to auto_drive_speed
  mLeft to auto_drive_speed

  repeat while call opModelsActive and call mRight.isBusy or call mLeft.isBusy
  do
    call Telemetry.addData
      key left encoder pos
      number mLeft.CurrentPosition
    call Telemetry.addData
      key right encoder pos
      number mRight.CurrentPosition
    call Telemetry.update

  set Velocity
  mLeft to 0
  mRight to 0

  call Telemetry.addData
    key left encoder pos
    number mLeft.CurrentPosition
  call Telemetry.addData
    key right encoder pos
    number mRight.CurrentPosition

  call Telemetry.update
```

Using Motor Encoders in Autonomous OpModes

```
to rotate with: Degrees turn
  set target_right to mRight.CurrentPosition -- Degrees turn * rotate_counts_per_degree * turn_multiplier
  set target_left to mLeft.CurrentPosition ++ Degrees turn * rotate_counts_per_degree * turn_multiplier
  set TargetPosition
    mRight to target_right
    mLeft to target_left
  set Mode
    mRight to RunMode RUN_TO_POSITION
    mLeft to RunMode RUN_TO_POSITION
  set Velocity
    mRight to auto_drive_speed
    mLeft to auto_drive_speed
  repeat while call opModelsActive and call mRight.isBusy or call mLeft.isBusy
  do
    call Telemetry.addData
      key left encoder pos
      number mLeft.CurrentPosition
    call Telemetry.addData
      key right encoder pos
      number mRight.CurrentPosition
    call Telemetry.update
  set Velocity
    mLeft to 0
    mRight to 0
  call Telemetry.addData
    key left encoder pos
    number mLeft.CurrentPosition
  call Telemetry.addData
    key right encoder pos
    number mRight.CurrentPosition
  call Telemetry.update
```

Using Motor Encoders in Autonomous OpModes

```
to runOpMode
  PowerPlay-Auto test
  Initialization Blocks
  Init Variables with:
    encoder_counts_per_rev 28
    wheel_diameter 106
    motor_gear_reduction 18.8
    wheelbase 370
  Init Motor Setup with:
    auto drive speed 750
  set turn_multiplier to 1.2
  call waitForStart
  if call opModelsActive
  do This is where all the magic happens
    drive with:
      Distance 24
    rotate with:
      Degrees turn 90
    drive with:
      Distance 24
    call sleep
      milliseconds 10000
```