BLOCKS by Kardia Robotics

A BLOCKS Programming Manual
created by Kardia Robotics FTC Team 11208

~As a service to fellow FTC teams

http://kardiarobotics.weebly.com/

BLOCKS BY KARDIA ROBOTICS

A Blocks programming manual created by Kardia Robotics 11208

As a service to fellow FTC teams

Table of Contents

Basic Programming

The intent of this handbook is to provide a simplified explanation of the more complex and useful facets of Blocks programming. It will cover topics that have been found most useful to FTC Team 11208 in our competition seasons. We hope that it will be useful to you, too!

HELPFUL RESOURCES

Many resources have already been provided for those interested in Blocks programming. A selection of theses are listed below.

-The FTC Blocks Programming Manual: this is a resource provided by FIRST which covers the basic configuration and setup for Blocks programming:

https://www.firstinspires.org/sites/default/files/uploads/resource_library/ftc/blocks-programming-manual.pdf

-Bruce Schafer of Oregon Robotics has also provided many short and helpful tutorials on YouTube:

https://www.youtube.com/playlist?list=PLq-SuSZcm5veW1kiK8JenKZoQgtrHi1on

- The ortop.org reference manual provides a detailed, text-based look at blocks programming:

http://www.ortop.org/ftc/BlocksProgramming/BlocksProgrammingReferenceManual.pdf

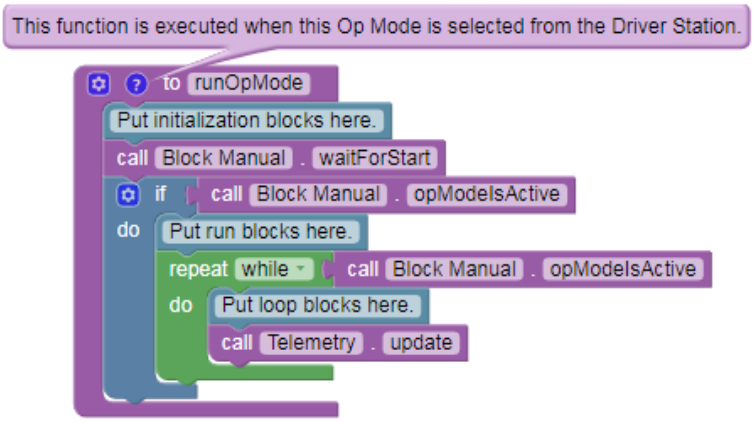
-Pennsylvania FTA Jim Rumbaugh's Blocks programming webinars (a great complementary resource for this manual, covering many of the same topics):

http://www.ftcpenn.org/ftc-events/2018-2019-season/blocks-programming-webinar

INTRODUCTION TO PROGRAMMING

Each Blocks program begins with the below template.

Figure 1.1

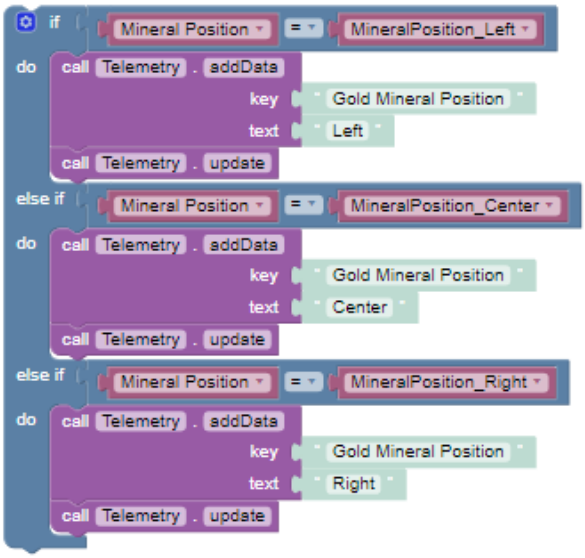


The purple "runOpMode" is the actual program; all blocks within this bracket will be used. Blocks outside it (such as functions) are available within the program but are not used unless called in the runOpMode block. The blue comment block ("Put initialization blocks here.") is merely a note to the programmer. It does not tell the robot anything; it is a bookmark, or commentary. In the template program, the comment block is put before the "waitForStart". This is as far as the program will go before the Start button is pressed on the driver station. This is where we usually place our SetUp functions (see page 24), set variables, and scan field elements (see page 17 and following). The template program also includes an example "if statement" and "while loop."

In an if statement, you program statements and, if they are true, the program runs the subsequent code. "Else if" blocks are used if the "if statement" is false and the "else if"

statement is true. “Else” blocks are used if neither the “if statement” or the “else if” statement is true.

Figure 1.2



Using a series of “if statements” from the Rover Ruckus game, for example:

-If the mineral position is left, send telemetry: Gold Mineral Position: Left

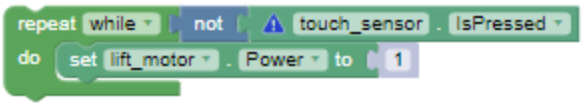
-If the mineral position is not left and is center, send telemetry: Gold Mineral Position: Center

-If the mineral position is neither left nor center, send telemetry: Gold Mineral Position: Right

In summary, the program will continue to search until it finds something true. If the first “if” is found true, the program will go no farther.

While loops are more simple. A qualification is set (in the example above, “while opMode is active”) and the blocks inside the while loop will repeat, endlessly and eternally, until the while qualification ceases to be true.

Figure 1.3



In the example above, the lift_motor's power will be continually set to one as long as the touch sensor is not pressed. When the touch sensor is pressed, it will break out of the loop.

Another simple (but incredibly important) kind of block is a telemetry block. These blocks are customizable; the words filled in for "key" and "text" will appear on the screen of the Driver Station (see Figure 1.2). For more on telemetry and its purposes, see page 32.

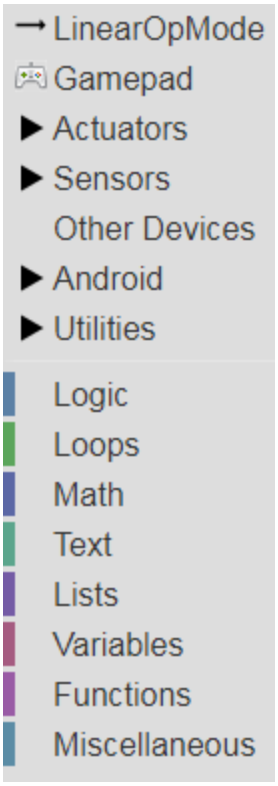
NOTE: In Figure 1.3, a warning (the small blue triangle with the exclamation point in the center) appears. If this warning appears in your program, it means that the device named in that block is not in the robot's configuration file; this can only happen if the device was in a configuration at one time and since been removed. This can also be caused by having a configuration file for more than one robot, such as when a sensor is on one robot and not the other. This is not necessarily something to be concerned about if the active configuration is not the one for the program you are editing. To amend this, click "configure robot" on either of the phones. There is an option to scan; this should create a basic configuration file for all the devices that the phone can detect on the robot. The FTC Blocks Programming Training Manual (link on page 2) provides a more detail look at configuration files and phone setup.

A MAP OF BLOCKS

The blocks are categorized on the left of the programming screen. A brief sketch of this organization scheme is included below. It is important to know that the blocks adapt to match

the needs of the robot. For example, if something is added to the configuration file, new blocks for that element will appear in the correct category.

Figure 1.4



In the LinearOpMode category blocks can be found which call the OpMode and give it instructions (waitForStart, idle, Sleep, opModeIsActive, isStarted, isStopRequested, and getRuntime).

The Gamepad category is for all the blocks which refer to the actual gamepad buttons, bumpers, triggers, joysticks, and Dpad. (These are the key to programming TeleOp).
The Triggers are a 0-1 analog input, and the joysticks are a -1 to 1 analog input. The triggers can be utilized as buttons by using an if statement; an example of this can be found on page 35 and following.

The Actuators tab is divided into subcategories for motors and servos (as indicated in the configuration file). Under the motor block tab there is also dual motor blocks, which set two different motors' powers, positions, etc. at once.

The Sensor category contains blocks for any sensor in the configuration file (i.e., touch or voltage sensors). For more on programming sensors, see page 14.

Other Devices is a catch-all category which will contain blocks for any "extra" devices added to the configuration file which cannot be organized in any of the other categories.

Android contains blocks for the Accelerometer, Gyroscope, Orientation, SoundPool, and TextToSpeech (see page 32 and following).

Utilities houses a variety of blocks subcategories, including those for TensorFlow (see page 17), Telemetry (page 32), Vuforia (page 14), Color (page 19), and Time, as well as a variety of other specific block categories, mostly related to positioning.

Beneath these are the eight programming categories, organized by color. *Logic* is the category which contains the blocks for "if statements", "true" blocks, and "=" blocks, among others. *Loops* contains blocks used for "while loops" or "until loops", "break out of loop" blocks, blocks for repeating a string of commands a certain number of times, etc. *Math* contains arithmetic blocks (adding, subtracting, etc.), empty blocks for numbers, pi, and trigonometric functions, as well as other math blocks. *Text* is the category which holds blocks which create and refer to text values. *Lists* blocks creates lists (used with variables) and provides blocks to sort, get, or set variable lists. *Variables* contains blocks for creating and setting variables. (Variables are data items the programmer creates that may change during the course of the program. These are very useful and there are numerous examples of variable use in the program examples

throughout this manual.) *Functions* contains blocks for creating and using functions and their returned information. (Functions are specific named sections of programming which focus on a specific task or procedure.) *Miscellaneous* contains a random section of uncategorizable blocks, including the comment block.

TIP: Because of Block's commitment to organization, a "clean up" is programmed in the system. If your blocks and functions have become lost in the vastness of the screen, right click and select "clean up blocks" and all your blocks will be collected in an ordered line in the center of the screen.

The best way to become comfortable with the blocks environment is to experiment with it. Feel free to explore!

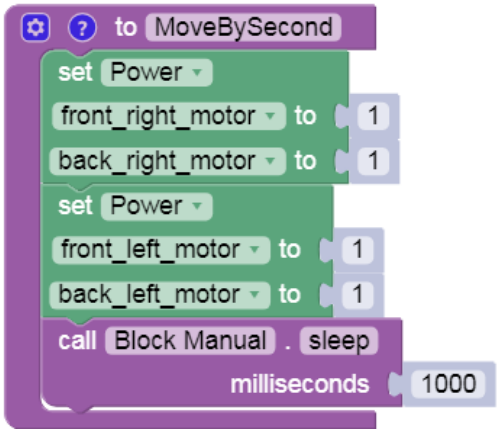
Movement

Our team has used two different movement methods: move by second (a simpler, albeit less reliable option) and move by inch (a more complicated but infinitely more precise method). Move by inch can only be achieved with motor encoders.

MOVE BY SECOND

Move by second is a simple movement system. However, programmers must be cautious when using this method. It is not very precise, fluctuating its path in response to changing factors such as remaining battery life and wheel tightness. For best results, check battery regularly and account for a margin of error in robot performance.

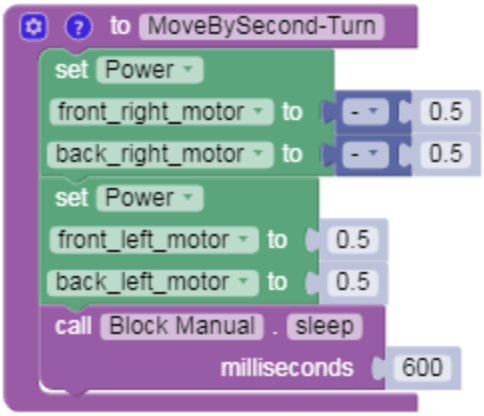
Figure 2.1



In the above function (for more on function use, see pages 24-31), move by second is demonstrated. All the motor powers are set to the same speed and will "sleep" for 1000 milliseconds. This results in the robot going forward for one second. Distances may be adjusted by changing the number of milliseconds the robot sleeps; speed may be adjusted by changing the

motor power (e.g., changing the motor power to 0.25 will result in the robot moving forward at one quarter of the current speed).

Figure 2.2



To turn with this method, reverse the motors on the side of the turn (e.g., for a right turn, the right motors must move backward). Adjusting the speed and sleep time will adjust the angle and duration of the turn. Correct adjustments are discovered by trial and error.

MOVE BY INCH

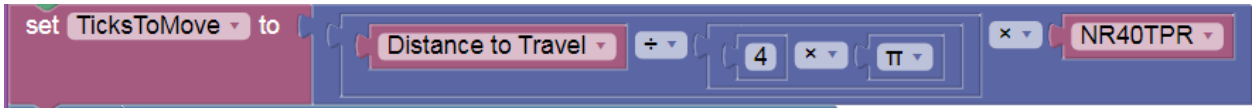
If one desires greater control over robot movement, programming by inch is an option which allows great precision and reliability. Note that this movement method requires motor encoders for maximum results. In order to program by inch, a simple computation must be done.

Figure 2.3

$$Ticks = \left(\frac{\frac{Distance\ to\ Travel}{Wheel\ Diameter * \pi}}{Gearing\ Ratio} \right) * Ticks\ per\ Revolution$$

In order to utilize the above formula, you must find your gearing ratio (drive teeth divided by wheel teeth), wheel circumference (diameter times pi), and ticks per revolution (for more information and some standard motor tick statistics, see the Appendix beginning on page 37).

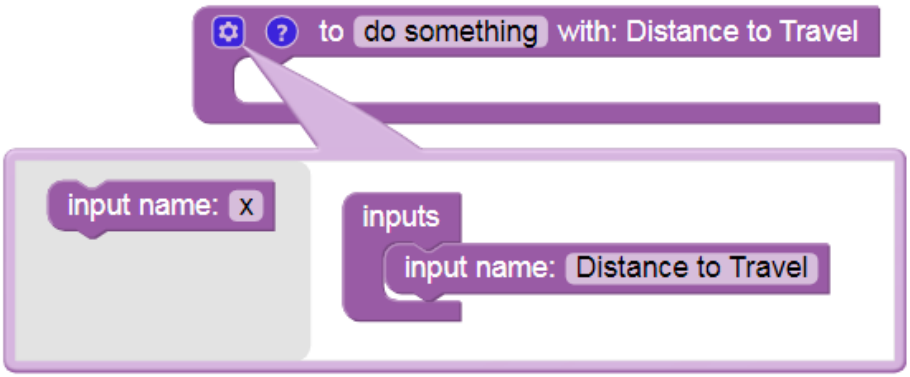
Figure 2.4



The whole equation can be created in the blocks program, where TicksToMove is a variable set to the output of the equation, Distance to Travel is a variable set and reset for each new movement, and NR40TPR is a variable created to represent a NeverRest 40 motor's ticks per revolution.

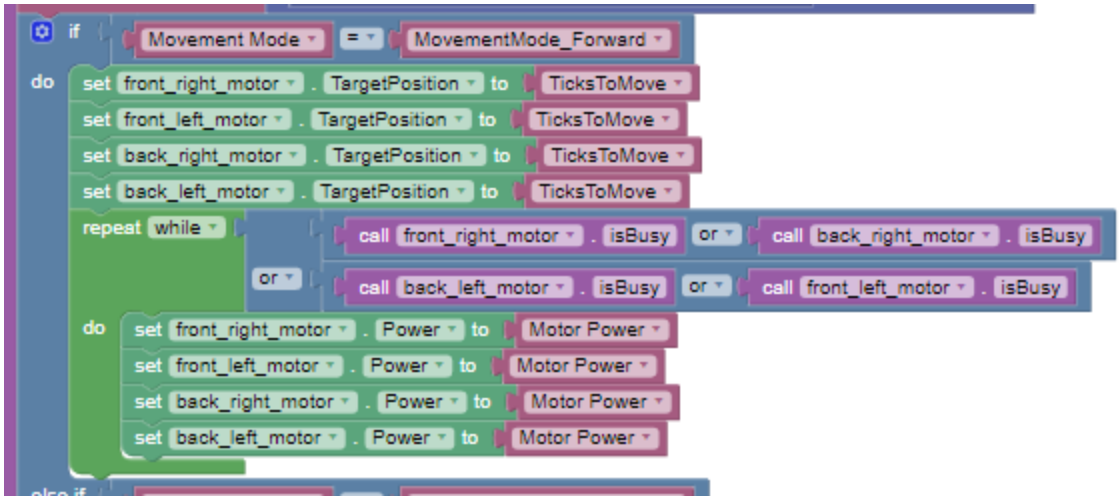
In order to create a function which can be repeatedly recycled throughout the program and used with new and varying travel distances, select the settings gear on the far left of the function block and begin adding inputs to match the needed variable.

Figure 2.5



Once this is accomplished, the function can be named and filled. Our standard EncoderMove functions generally include setting each motor's run mode to stop and reset encoder (so that at each new motion the robot is able to start fresh and track its progress from its current starting point) and then to run to position (so that the robot will be constantly running to meet a preprogrammed target position). After that, the function should contain the necessary math (figure 2.5). Our experimentations with programming encoder-based movements using mecanum wheels has led us to create Movement Mode variables for each direction and then calling them when needed.

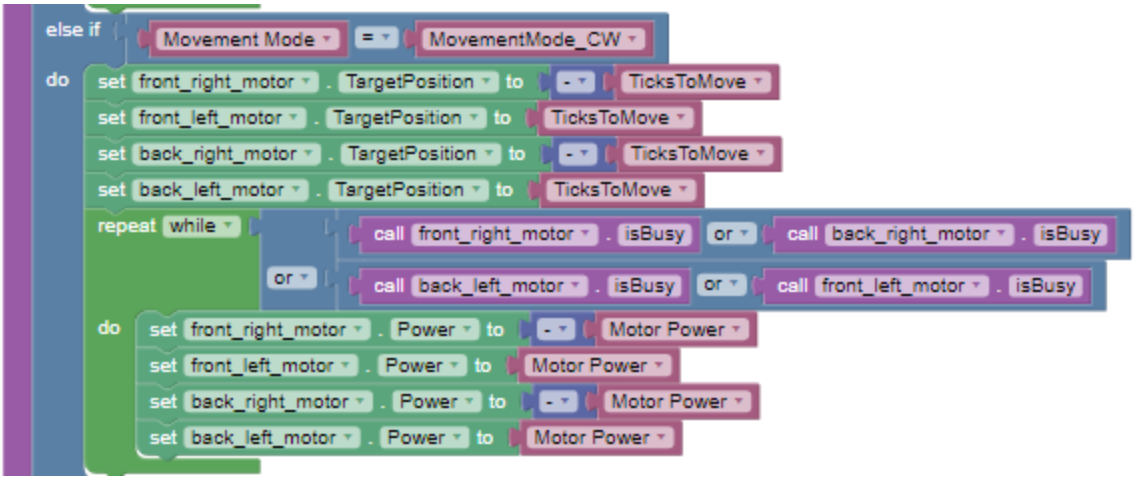
Figure 2.6



The rest of the function after the equation should be composed of if statements that checks to see which movement mode is requested. For example, if forward is selected (see above) each motor is set to the variable TicksToMove (the output of the equation before it) and told to repeat while any of the motors are still busy. Until every motor reaches the target position (as recorded by the encoder) the loop will continue. To change direction between movement modes, negative symbols (-, found in Math) can be added before each TicksToMove and

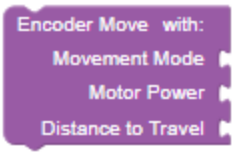
MotorPower variable. For example, to drive clockwise, the two right motors would need to be going backwards as the left motors went forward, and the program would appear like so:

Figure 2.7



The creation of a function such as the one pictured above (with added inputs for MovementMode, MotorPower, and Distance to Travel) will create the block pictured in Figure 2.8, which can be recycled for each new movement provided it is given the new movement mode (i.e., direction, where the wheel directions are set in the original EncoderMove function), motor power, and Distance to Travel (a numerical unit which will be inferred as inches).

Figure 2.8

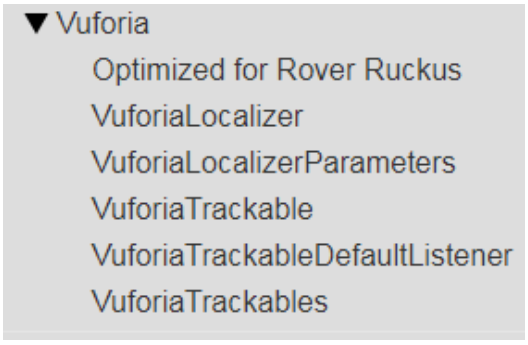


Sensors, Etc.

Most FTC games include sensors. This manual will discuss some of the most commonly used sensors and sensory aids (Vuforia, TensorFlow, color sensors, distance sensors, and touch sensors).

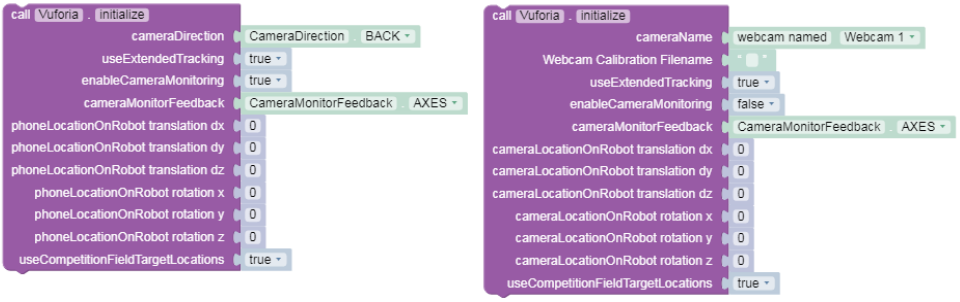
VUFORIA

Most recent FTC games have included Vuforia, for alignment purposes or acquiring game-relevant information. FIRST releases blocks customized for the new games at the beginning of each new game season, as seen below.



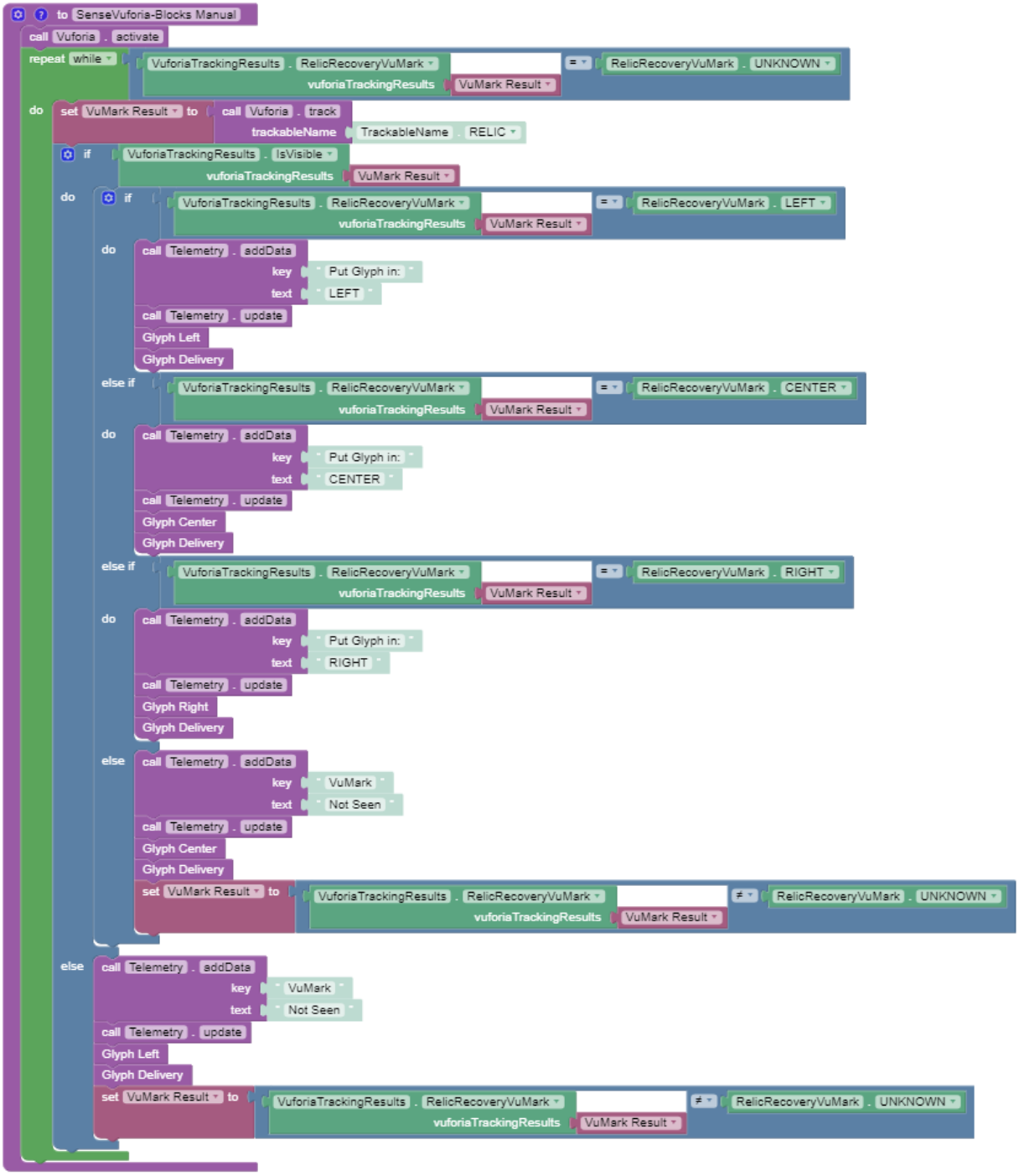
In order to use Vuforia, a camera must be placed on the robot with an unobstructed view of the VuMark. If this cannot be achieved with the phone camera, Vuforia can be used with an external webcam. The two different initialization blocks are pictured below.

Figure 3.1



Generally, the default settings do not need to be changed. This block can be placed with the other initialization blocks, in the setup function (see page 24).

Figure 3.2



In the above program sample, Vuforia is activated. The program will continue until the VuMark result (a variable set in the SetUp function, see page 24 and following) is known, or *not*

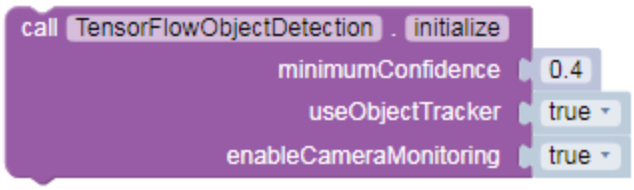
unknown. The trackable name is set to the current year's Vuforia package (this example was taken from a program run during Relic Recovery, the 2017-2018 FTC game). If the VuMark is visible, the program launches a series of if-statements to determine which VuMark pattern it is (and act accordingly by calling coordinating functions); if it is not visible, it returns to telemetry and continues on a "default path".

TIP: Remember to always set a default path, especially if your autonomous program is structured with Vuforia-sensing at an early step! If your program is written so that the robot is looking for VuMarks and moves *only* when they are found, your program will not move at all if, for some reason, the VuMark remains unseen.

TENSORFLOW

TensorFlow, created by Google, is programmed essentially the same as Vuforia and uses similar initialization. Unlike Vuforia, which is used to recognize preprogrammed images, TensorFlow is more shape-oriented and can be used to return numerical coordinates.

Figure 3.3



Like Vuforia, TensorFlow requires only one initialization block and one activation block. In the initialization block (figure 3.3), the minimum confidence is set. This gives the program parameters for how certain it must be to report its findings. If TensorFlow is reporting objects

inaccurately or too often, this number can be changed to anything from 0 to 1, 1 representing 100% confidence.

Figure 3.4

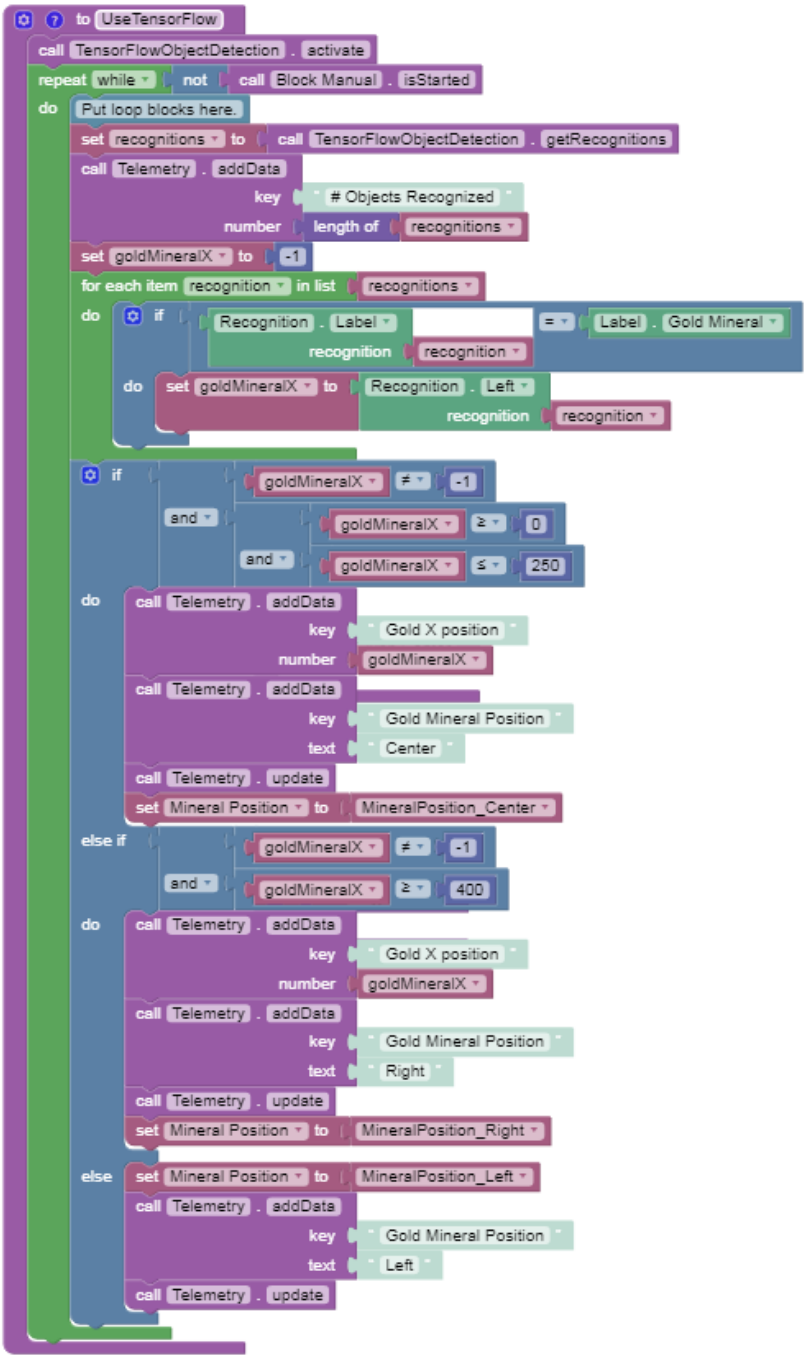


Figure 3.4 demonstrates a TensorFlow function (utilizing the blocks optimized for Rover Ruckus). After TensorFlow is activated, the program begins a while-loop which loops while the program is not started (that is, initialized).

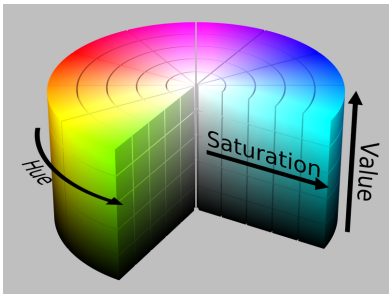
TIP: It saves a lot of time for the program to sense things in initialization, rather than taking up valuable time during the 30-second autonomous period. However, as the field is still being randomized during this time, it is important for the program to keep sensing and not fix on anything until the start button is pressed. The while loop is crucial for this; also, a "break out of loop" block must not be put in the program. To make sure the sensing is constant, add an "update telemetry" block.

The TensorFlow engine tracks objects from left to right, with 0 being far left and far right ending in the 500-600 range. To use TensorFlow, program the numerical range the object is to be found in. In figure 3.4, the program is sensing either the center mineral (0-250) and the right mineral (any object to the right of the 400 mark). The program deduces the mineral is on the the left if objects are not detected in either of these two ranges. (Programs can be made to sense all three; the decision to sense only two minerals in the example above was team-specific).

COLOR SENSORS

Most simple color sensor programs use the HSV (hue, saturation, value) scale to determine color, where hue refers to the actual pigment (i.e., red, green, blue), saturation refers to the intensity of the pigment (i.e., a highly-saturated blue object would be more blue than a lightly-saturated object), and value refers to the light or darkness of the object. Figure 3.5 below is a visual example of the difference between hue, saturation, and value.

Figure 3.5:

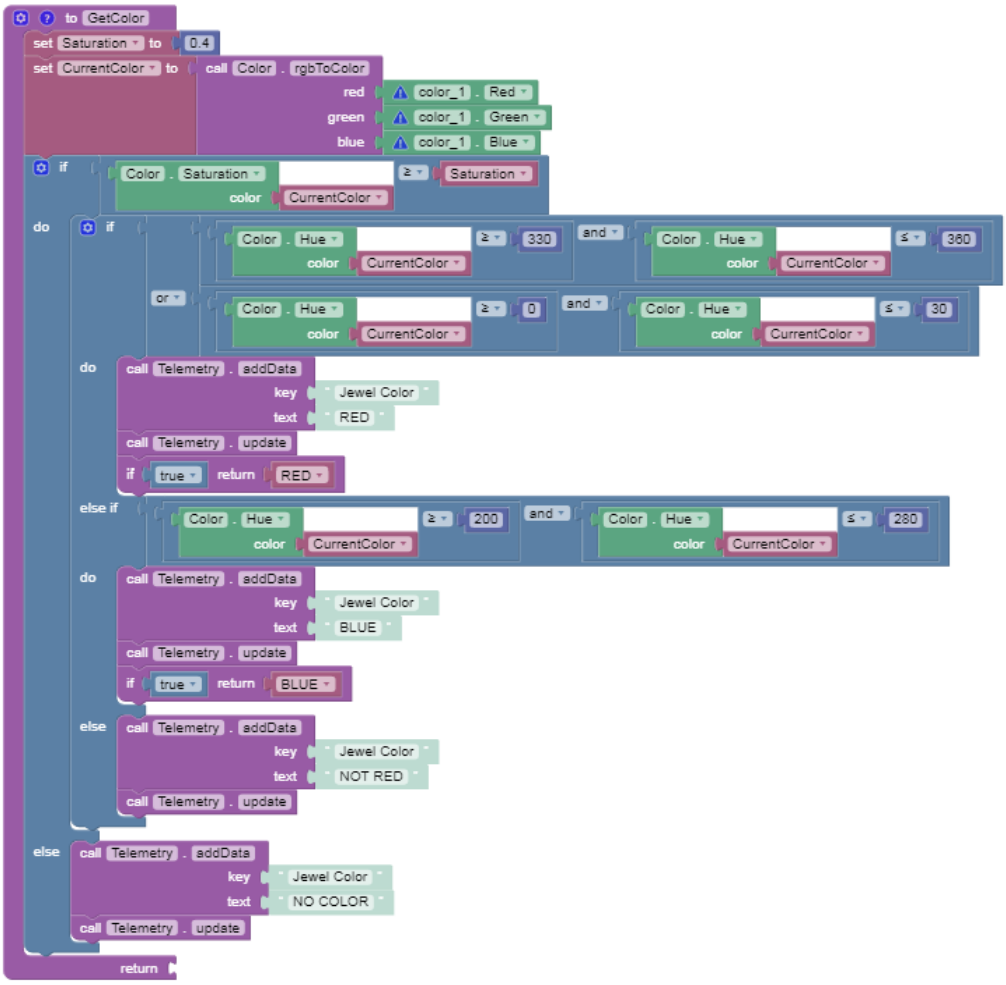


By HSV_color_solid_cylinder.png: SharkDderivative work: SharkD Talk - HSV_color_solid_cylinder.png, CC BY-SA 3.0,

https://commons.wikimedia.org/w/index.php?curid=9801673

In our color sensor programs, we created a variable CurrentColor and set it to rgb values, so that the hues are standard red/green/blue. We also set RED and BLUE (the colors we are looking for) as 1 and 2, respectively, in our SetUpAuto function.

Figure 3.6



As seen in the GetColor function (figure 3.6), saturation is set to a number between 0 and 1, and CurrentColor is set to the rgb standard color scale. The program then states that if the color currently being sensed is at least as saturation equal to or greater than the set saturation variable, the color is determined. If it does not, return no color.

For determining color, any standard HSV color wheel can be used for approximate values. Our suggested range for red is 0-30 and 330-360; our suggested range for blue is 200-280.

To use a function such as GetColor in a program, simply create in if statement based on the returned color variable and program subsequent paths, like the empty if-statement below. Note that in this template, the ReturnedColor variable will be newly set to whatever color the GetColor function finds.

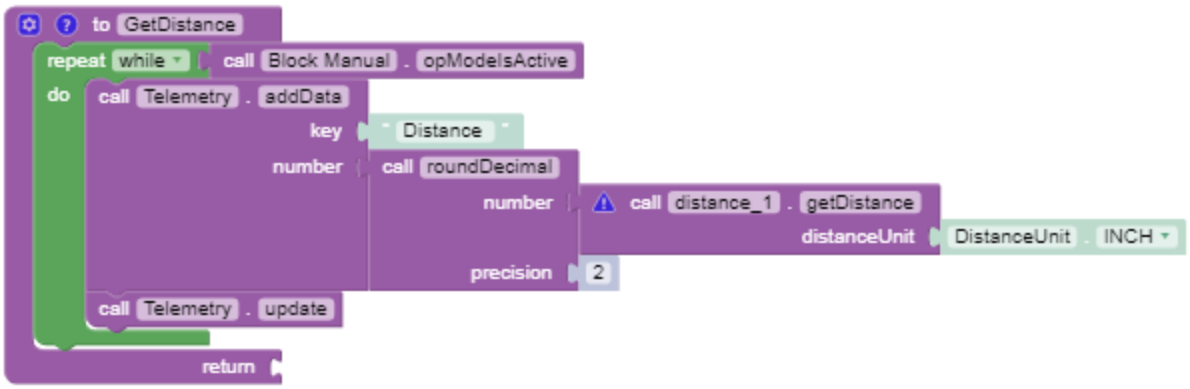
Figure 3.7

TIP: When using color sensors, it is helpful to create a "ColorSenseTime" variable so that the program knows when to give up looking for a color and move on to its next task.

DISTANCE SENSORS

Distance sensors are arguably the easiest sensors to program.

Figure 3.8



To use a distance sensor, attach a block calling the distance sensor (distance_1 in figure 3.8), specify the unit of measurement, and set the precision (i.e., number of digits allowed in the

reported answer. To use that distance later in the program, a block combination like that pictured in Figure 3.9 may work (although the possibilities are infinite):

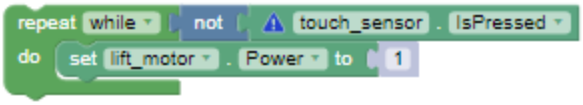
Figure 3.9

Distance sensors open the door to a whole realm of possibilities, specifically for using positional feedback to determine future movements. If the program is meant to sense continually, make certain the distance-sensing blocks are placed in a while loop with updating telemetry.

TOUCH SENSORS

Touch sensors are programmed much like TeleOp commands: dependent on a pressed button.

Figure 3.10



In the example above, a motor will be full power as long as the touch sensor is not pressed.

Functions

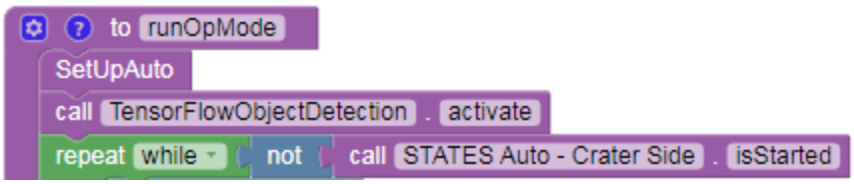
Functions, as previously stated, are specific named sections of programming which focus on a specific task or procedure. Functions can be used to organize and name specific tasks (especially repeated ones) outside the main body of the program. Our team has used these for movement functions, setup, and color sensing, among other things.

SETUP FUNCTIONS

Our team uses SetUp functions to initialize systems; set motor directions, modes, and zero power behaviors; set variables; and set servo positions. This function is called in the program before the start button is pressed, so that it is activated in the initialization phase. It is helpful to keep all the "factory settings" for each program in one place.

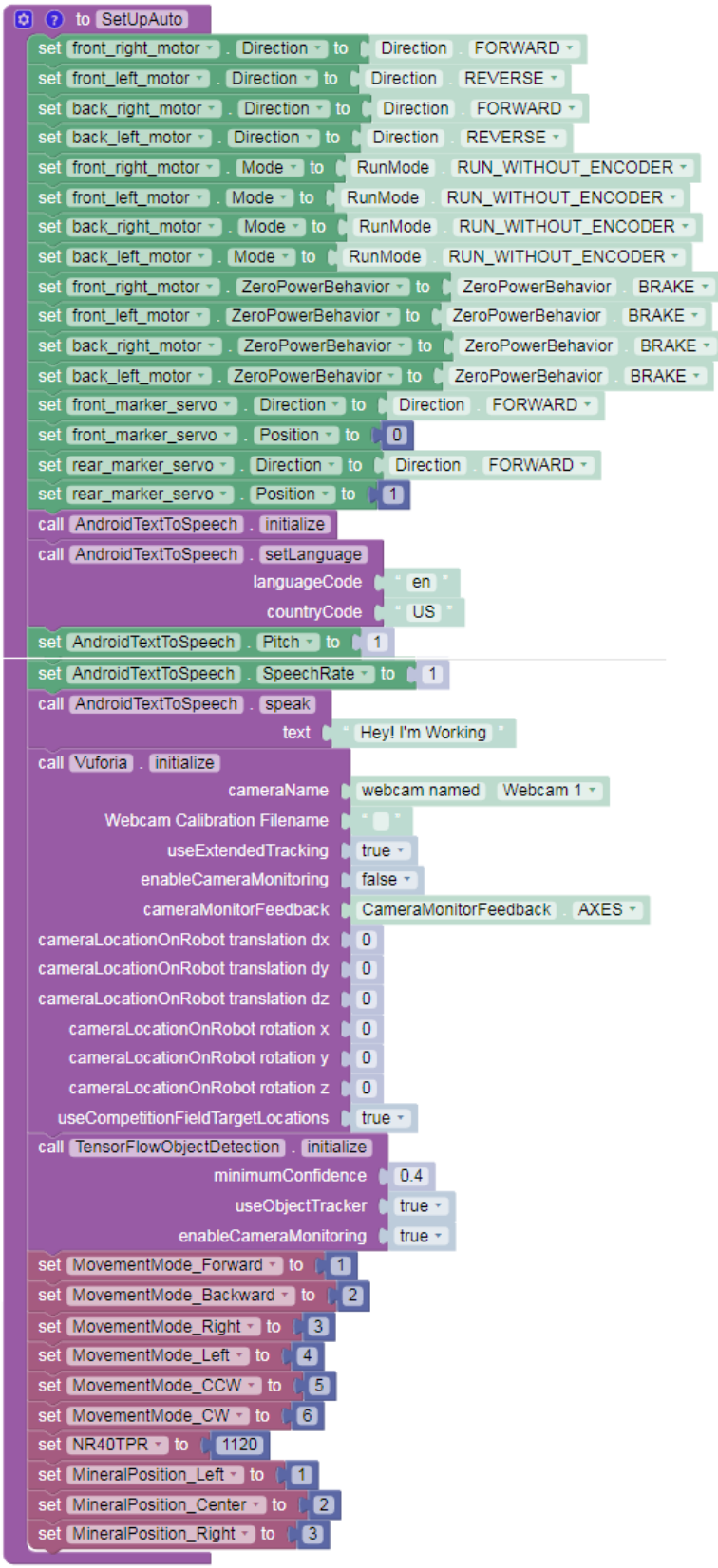
For instance, for the past two years, our program has called SetUpAuto first thing, and then launched into whatever sensing is necessary (Vuforia, TensorFlow), all before Start. Seconds are precious in autonomous, but the time before it begins is cheap and ought to be utilized!

Figure 4.1



Our full end-of-season setup program is one of our largest functions.

Figure 4.2


to SetUpAuto
set front_right_motor . Direction to Direction . FORWARD
set front_left_motor . Direction to Direction . REVERSE
set back_right_motor . Direction to Direction . FORWARD
set back_left_motor . Direction to Direction . REVERSE
set front_right_motor . Mode to RunMode . RUN_WITHOUT_ENCODER
set front_left_motor . Mode to RunMode . RUN_WITHOUT_ENCODER
set back_right_motor . Mode to RunMode . RUN_WITHOUT_ENCODER
set back_left_motor . Mode to RunMode . RUN_WITHOUT_ENCODER
set front_right_motor . ZeroPowerBehavior to ZeroPowerBehavior . BRAKE
set front_left_motor . ZeroPowerBehavior to ZeroPowerBehavior . BRAKE
set back_right_motor . ZeroPowerBehavior to ZeroPowerBehavior . BRAKE
set back_left_motor . ZeroPowerBehavior to ZeroPowerBehavior . BRAKE
set front_marker_servo . Direction to Direction . FORWARD
set front_marker_servo . Position to 0
set rear_marker_servo . Direction to Direction . FORWARD
set rear_marker_servo . Position to 1
call AndroidTextToSpeech . initialize
call AndroidTextToSpeech . setLanguage
languageCode en
countryCode US
set AndroidTextToSpeech . Pitch to 1
set AndroidTextToSpeech . SpeechRate to 1
call AndroidTextToSpeech . speak
text Hey! I'm Working
call Vuforia . initialize
cameraName webcam named Webcam 1
Webcam Calibration Filename
useExtendedTracking true
enableCameraMonitoring false
cameraMonitorFeedback CameraMonitorFeedback . AXES
cameraLocationOnRobot translation dx 0
cameraLocationOnRobot translation dy 0
cameraLocationOnRobot translation dz 0
cameraLocationOnRobot rotation x 0
cameraLocationOnRobot rotation y 0
cameraLocationOnRobot rotation z 0
useCompetitionFieldTargetLocations true
call TensorFlowObjectDetection . initialize
minimumConfidence 0.4
useObjectTracker true
enableCameraMonitoring true
set MovementMode_Forward to 1
set MovementMode_Backward to 2
set MovementMode_Right to 3
set MovementMode_Left to 4
set MovementMode_CCW to 5
set MovementMode_CW to 6
set NR40TPR to 1120
set MineralPosition_Left to 1
set MineralPosition_Center to 2
set MineralPosition_Right to 3

MOVEMENT AND TURN FUNCTIONS

Our Movement functions follow a consistent pattern. Figure 4.2 demonstrates a move-by-inch function (see page 9) used with two omniwheels with encoders; figure 4.3 is an example of our most recent move by inch movement function, used with four mecanum wheels with encoders.

Figure 4.2

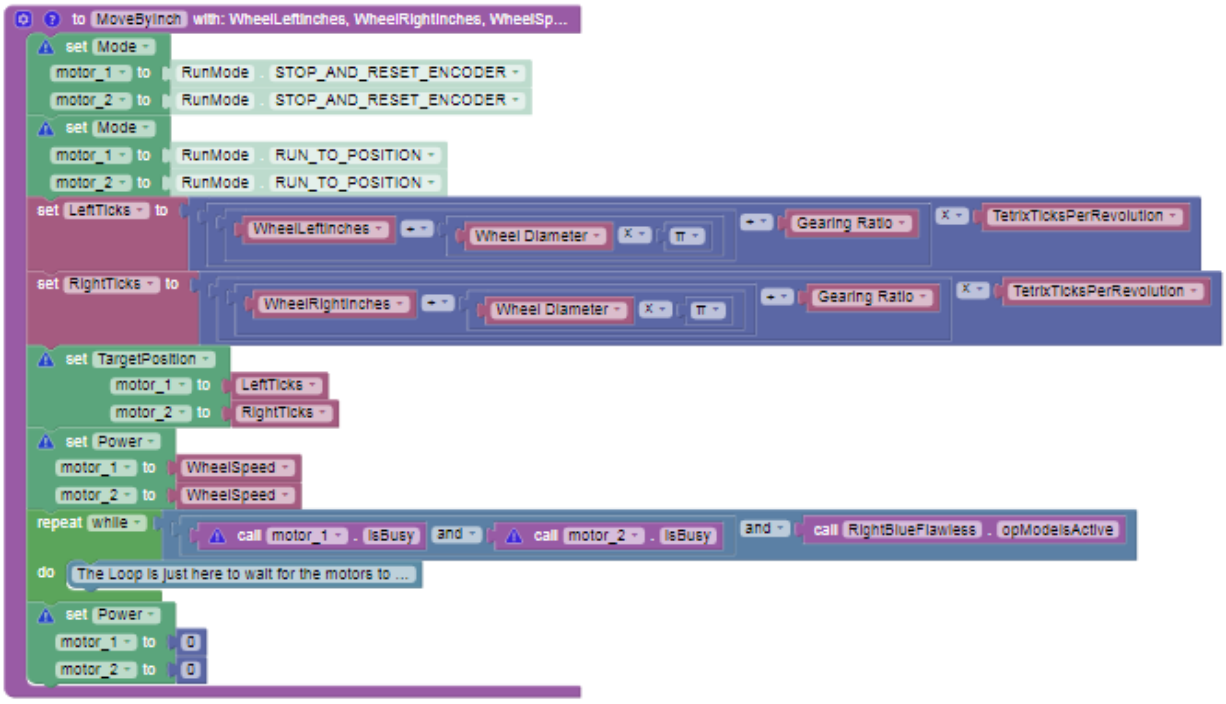
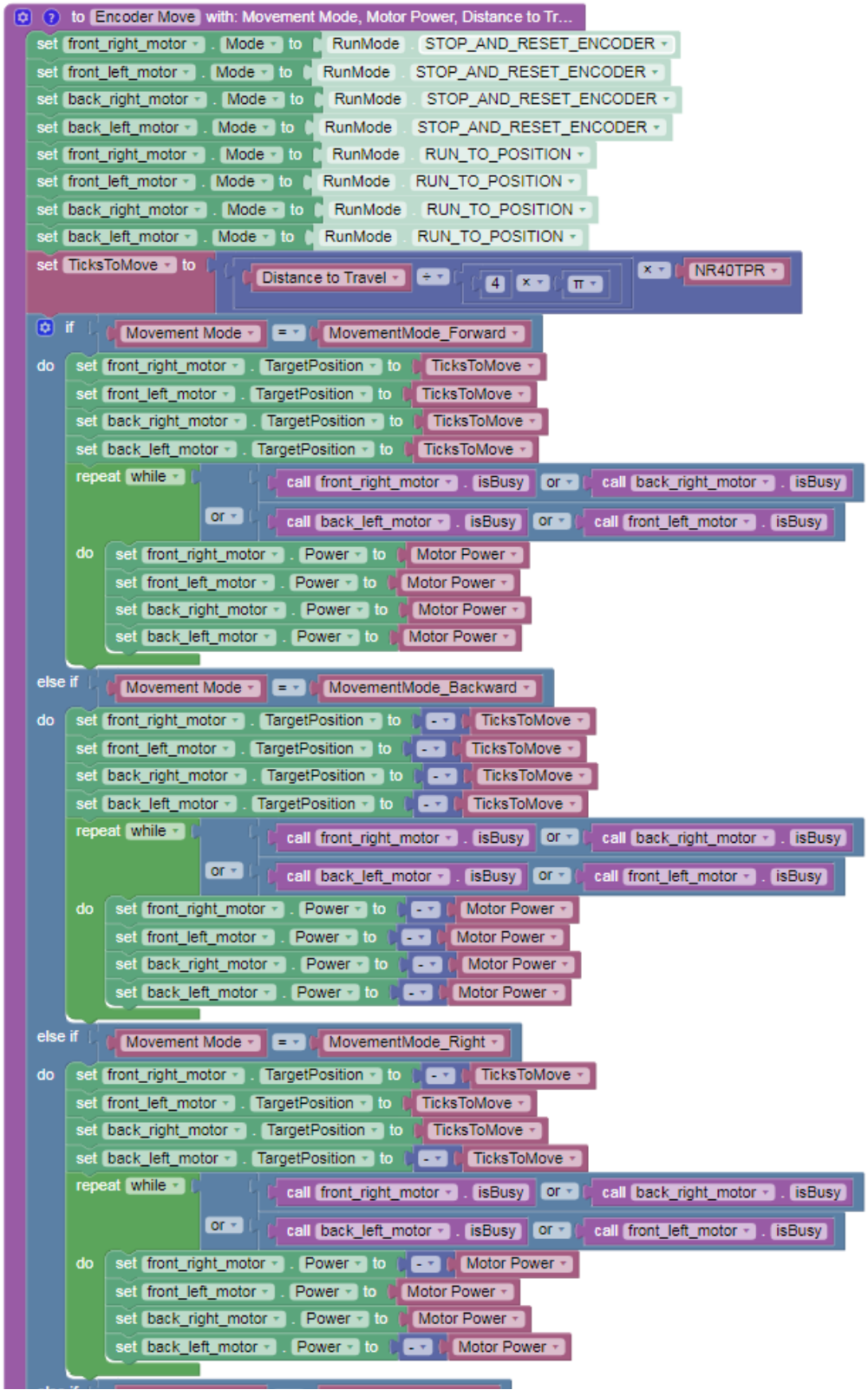
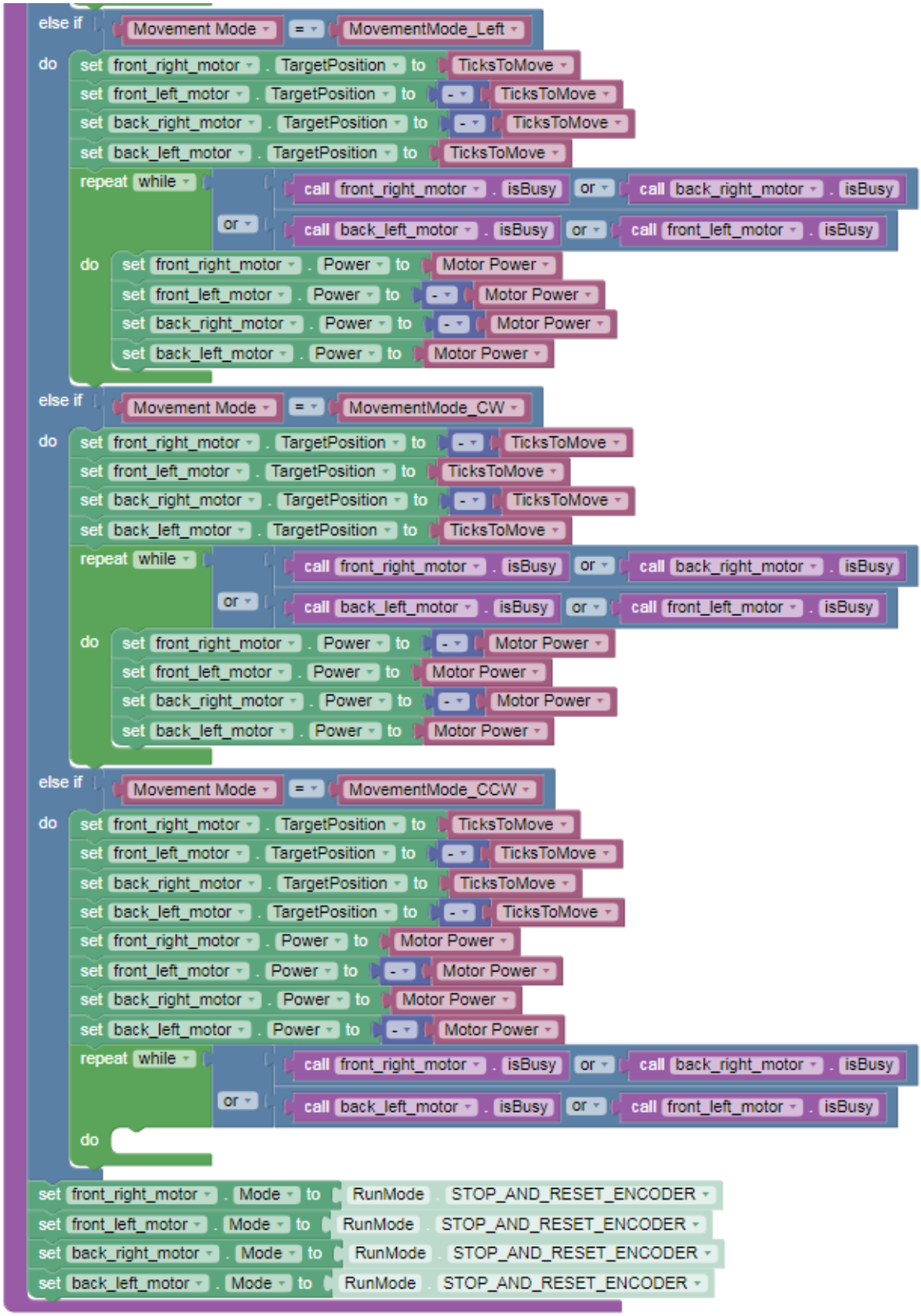


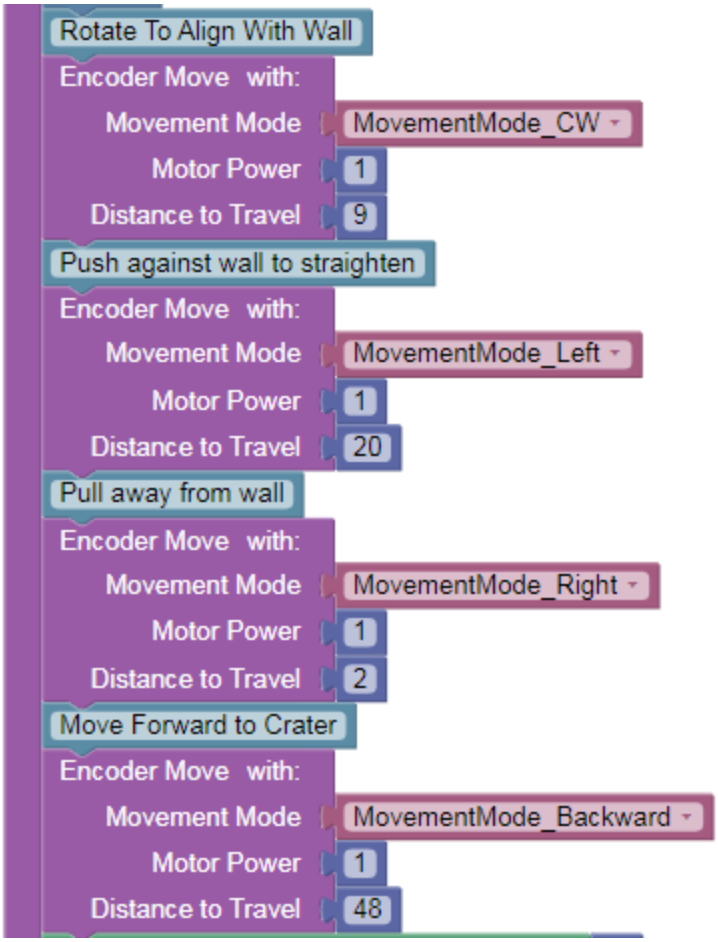
Figure 4.3


to Encoder Move with: Movement Mode, Motor Power, Distance to Tr...
set front_right_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set front_left_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set back_right_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set back_left_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set front_right_motor . Mode to RunMode . RUN_TO_POSITION
set front_left_motor . Mode to RunMode . RUN_TO_POSITION
set back_right_motor . Mode to RunMode . RUN_TO_POSITION
set back_left_motor . Mode to RunMode . RUN_TO_POSITION
set TicksToMove to Distance to Travel ÷ 4 × π × NR40TPR
if Movement Mode = MovementMode_Forward
do set front_right_motor . TargetPosition to TicksToMove
set front_left_motor . TargetPosition to TicksToMove
set back_right_motor . TargetPosition to TicksToMove
set back_left_motor . TargetPosition to TicksToMove
repeat while call front_right_motor . isBusy or call back_right_motor . isBusy
or call back_left_motor . isBusy or call front_left_motor . isBusy
do set front_right_motor . Power to Motor Power
set front_left_motor . Power to Motor Power
set back_right_motor . Power to Motor Power
set back_left_motor . Power to Motor Power
else if Movement Mode = MovementMode_Backward
do set front_right_motor . TargetPosition to - TicksToMove
set front_left_motor . TargetPosition to - TicksToMove
set back_right_motor . TargetPosition to - TicksToMove
set back_left_motor . TargetPosition to - TicksToMove
repeat while call front_right_motor . isBusy or call back_right_motor . isBusy
or call back_left_motor . isBusy or call front_left_motor . isBusy
do set front_right_motor . Power to - Motor Power
set front_left_motor . Power to - Motor Power
set back_right_motor . Power to - Motor Power
set back_left_motor . Power to - Motor Power
else if Movement Mode = MovementMode_Right
do set front_right_motor . TargetPosition to - TicksToMove
set front_left_motor . TargetPosition to TicksToMove
set back_right_motor . TargetPosition to TicksToMove
set back_left_motor . TargetPosition to - TicksToMove
repeat while call front_right_motor . isBusy or call back_right_motor . isBusy
or call back_left_motor . isBusy or call front_left_motor . isBusy
do set front_right_motor . Power to - Motor Power
set front_left_motor . Power to Motor Power
set back_right_motor . Power to Motor Power
set back_left_motor . Power to - Motor Power


else if
Movement Mode = MovementMode_Left
do
set front_right_motor . TargetPosition to TicksToMove
set front_left_motor . TargetPosition to - TicksToMove
set back_right_motor . TargetPosition to - TicksToMove
set back_left_motor . TargetPosition to TicksToMove
repeat while
call front_right_motor . isBusy or call back_right_motor . isBusy
or
call back_left_motor . isBusy or call front_left_motor . isBusy
do
set front_right_motor . Power to Motor Power
set front_left_motor . Power to - Motor Power
set back_right_motor . Power to - Motor Power
set back_left_motor . Power to Motor Power
else if
Movement Mode = MovementMode_CW
do
set front_right_motor . TargetPosition to - TicksToMove
set front_left_motor . TargetPosition to TicksToMove
set back_right_motor . TargetPosition to - TicksToMove
set back_left_motor . TargetPosition to TicksToMove
repeat while
call front_right_motor . isBusy or call back_right_motor . isBusy
or
call back_left_motor . isBusy or call front_left_motor . isBusy
do
set front_right_motor . Power to - Motor Power
set front_left_motor . Power to Motor Power
set back_right_motor . Power to - Motor Power
set back_left_motor . Power to Motor Power
else if
Movement Mode = MovementMode_CCW
do
set front_right_motor . TargetPosition to TicksToMove
set front_left_motor . TargetPosition to - TicksToMove
set back_right_motor . TargetPosition to TicksToMove
set back_left_motor . TargetPosition to - TicksToMove
set front_right_motor . Power to Motor Power
set front_left_motor . Power to - Motor Power
set back_right_motor . Power to Motor Power
set back_left_motor . Power to - Motor Power
repeat while
call front_right_motor . isBusy or call back_right_motor . isBusy
or
call back_left_motor . isBusy or call front_left_motor . isBusy
do
set front_right_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set front_left_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set back_right_motor . Mode to RunMode . STOP_AND_RESET_ENCODER
set back_left_motor . Mode to RunMode . STOP_AND_RESET_ENCODER

As demonstrated in pages 10-13, a move function like the one above can be used with great versatility within a program for nearly any directional movement.

Figure 4.4



In use of functions, remember that functions can be made to return values and/or require inputs. Figure 4.5 is an example of what a return function looks like, and an input function is pictured in Figure 4.6.

Figure 4.5

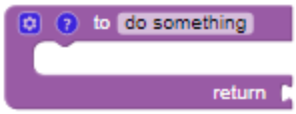
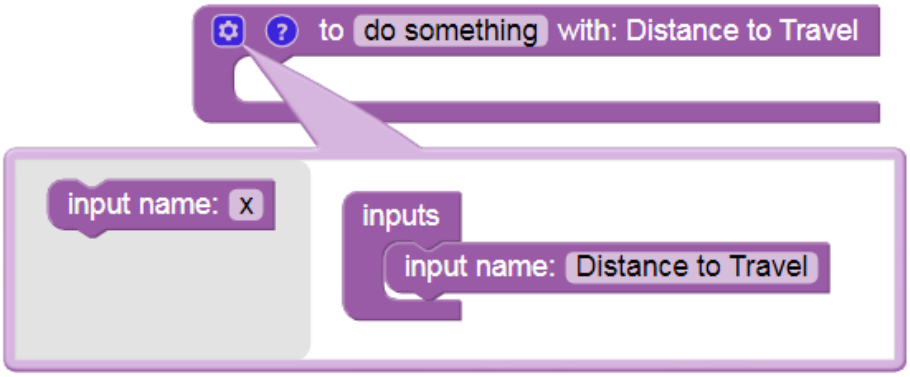


Figure 4.6


to do something with: Distance to Travel
input name: x
inputs
input name: Distance to Travel

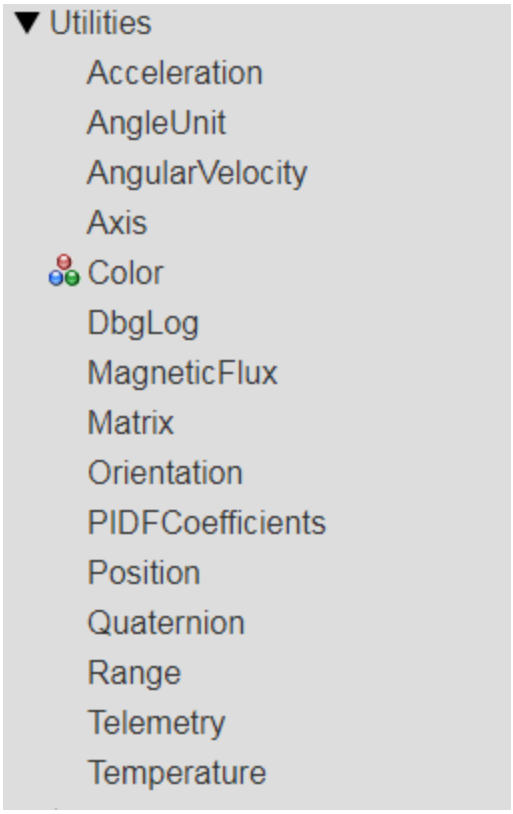
Troubleshooting

In every program, there are a variety of challenges and issues that must be met and remediated. To better understand what the robot is doing and what the robot errors are, there are several steps that can be taken.

TELEMETRY

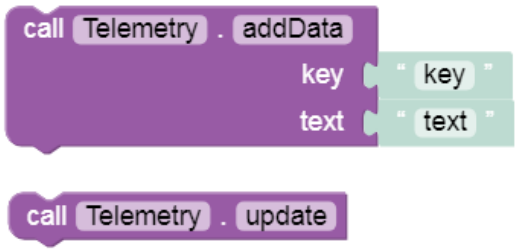
Kardia Robotics has experimented with two types of telemetry: written telemetry that appears on the screen of the driver station and oral telemetry using Android Text-to-Speech. Regular telemetry is found on the left sidebar under Utilities.

Figure 5.1



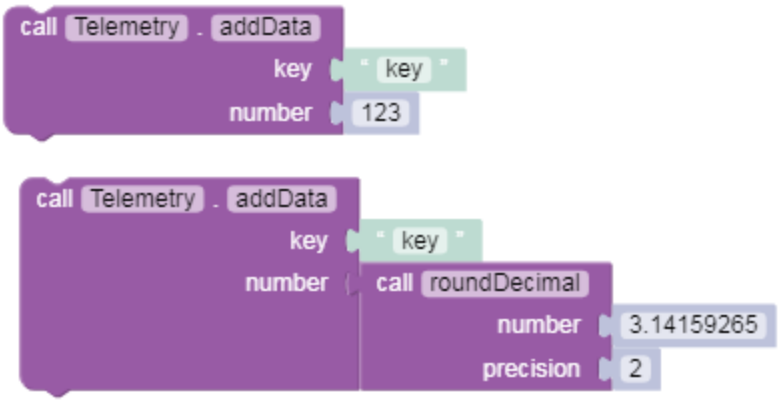
To use telemetry within the program, fill in the key and text (these will appear on the driver station screen beneath the play button as key: text). Remember to add an update telemetry block, especially if it is a sensory output which may change or need constant review (such as a VuMark result).

Figure 5.2



The upper blocks pictured in Figure 5.2 are those used for textual telemetry (such as Mineral Position: Center). However, telemetry can also be used to report numbers. Blocks even provides numerical telemetry blocks which round off the number at a predetermined digit. Examples of these are demonstrated below. To round the number, simply edit the precision for how many decimals ought to be displayed on the driver station phone.

Figure 5.3

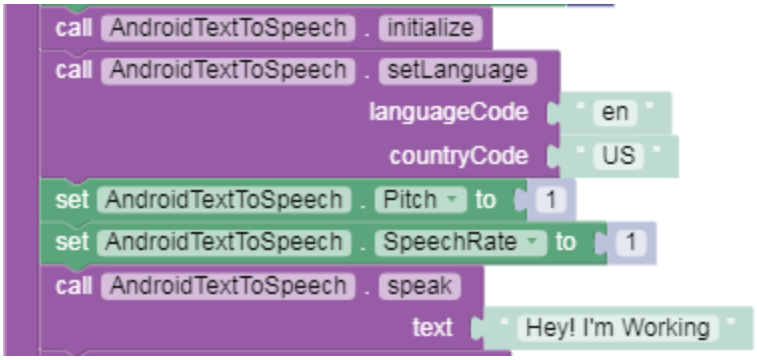


Telemetry can aid in troubleshooting. If the programmer is uncertain regarding the robot's status (i.e., where the problem is) insert telemetry blocks after each successive motion and read on the phone screen what the robot believes it is doing. By this, the gap between the robot's understanding of the program and the programmer's understanding of the program can be more easily discerned.

Another telemetry option available on Blocks is Android Text to Speech. This is found under Utilities just like ordinary telemetry blocks. Android Text to Speech utilizes the phone to give oral telemetry (i.e., to read the telemetry that is programmed in it). This can be especially helpful in debugging and troubleshooting because it enables the programmer to watch the robot's actions and hear the robot's "reports" without having to choose between looking at the robot's path or the phone's telemetry updates.

Initialization for Android Text to Speech is fairly simple, requiring only blocks.

Figure 5.4



Android Text to Speech must be initialized. Then, the language must be set (with predetermined language and country codes), along with the pitch and speech rate. 1 is the standard for both pitch and speech rate. AndroidTextToSpeech.speak blocks function identically

to telemetry blocks; the programmer programs the desired speech in the text block and the robot will read them off when that stage of the program is reached.

POWERNAPS

Another sophisticated debugging technique which has proved valuable to our team is the PowerNap. This is a short function that causes the robot to pause for a moment, so that each action is separate and easier to track. Isolating movement allows for not only easier understanding of which action the robot is completing, but it also increases precision, especially if the robot is using the move-by-second movement method (see page 9 and following). The PowerNap function for our two-motor omniwheel drivetrain is pictured below.

Figure 5.5

TeleOp Programming

In many ways, TeleOp programming is significantly simpler than autonomous programming. It focuses on if-statements connected to gamepad commands (see pages 3 and 6).

TELEOP PROGRAMMING METHODS

TeleOp programming varies from autonomous programming in one key element: TeleOp programs are live programs. When programming an autonomous, all the functions are run in a straight program flow and are only executed once each (save for commands in for or while loops). When programming a TeleOp program, the entire program is contained in a while(OpModeIsActive) command, this will run the program over and over while the OpMode is active.

When programming motor commands, the motor power is a range of -1 (full power reverse) to +1 (full power forward), this is usually accomplished by using inputs from the joysticks because they also have a range of -1 to +1. An example of a two wheel tank drive TeleOp can be seen in figure 6.4. An example of a mecanum drive train can be seen in figure 6.3, this TeleOp is set up so that there is a tank drive system with the two joysticks, and the strafe command for the mecanum wheels is on the left/right direction of both sticks.

An addition to the drivetrain control is a motor power value, when the motors are set to the power of a joystick, they are also multiplied by the motor power value, these values are set in the beginning of the TeleOp and are used to give the driver four different speed options to use when driving.

Another key way of programming TeleOp functions is using a simple if statement to control motor power, figure 6.3 uses an if statement to control our winch motor that says if the DpadUp button is pressed, the motor is set to +1 power, and else if (else if means that it only is true if the if is false and the else if is true) sets the motor power to -1, there is then an else section to set the motor power to 0 if nothing is pressed, this stops the motor after one of the buttons is released.

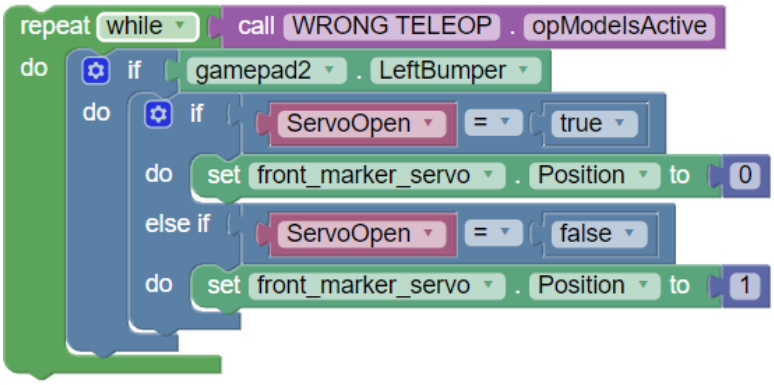
The final key element to a well functioning autonomous is a latching function, we used this type of function to latch our intake mechanism, we had the bumpers* activate the intake mode, the triggers** activate the output mode, and pushing the bumper again after it was in intake would activate the off mode (see page 38-46).

An if statement lower down would then evaluate which mode the system is in, then output motor movement based on that, for the off mode, the motor would be off, for the intake mode, the motor would be constantly taking in, and for the output mode the motor will only be powered when the output button is also pressed (this allowed our drive teams to deposit minerals one at a time).

A common problem with TeleOp commands can be solved by latching the buttons. Programmers must remember that the TeleOp program is running very quickly. If a program is set up to check "if the button is pressed and the servo is open, close the servo OR if the button is pressed and the servo is closed, open the servo," the program will cycle through the code, changing too quickly to register the correct value change. Even if the driver's finger is on the button for a short amount of time, it is enough for the program to say "the button is pressed and the servo is open, close it" and then while the button is still depressed, "the button is pressed and

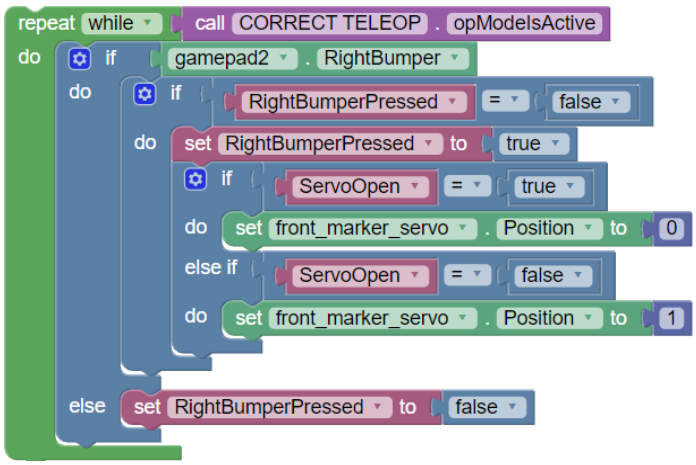
the servo is closed, open it." An example of this type of function (using two nested if statements) is shown below.

Figure 6.1



The problem can be solved by nesting three different if statements. The latch function that we used is made so that a press of a button will not just turn something on, it will toggle a value without spamming that value (this problem is shown in figure 6.1 and the paragraph proceeding). The latch function on the other hand uses nested if statements to make sure that once the button is pressed once, it won't register as having it's value changed until been released.

Figure 6.2



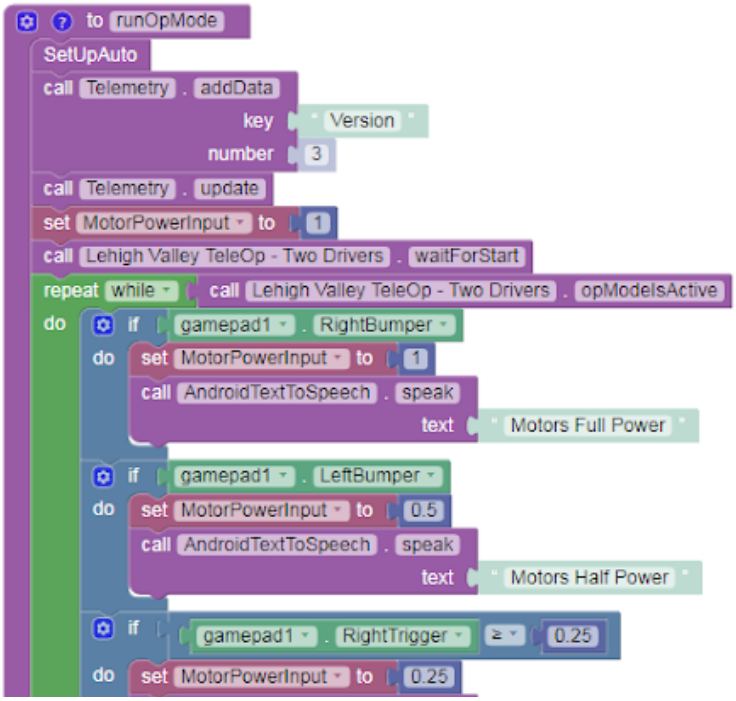
*bumpers: we have the program set up so that if either bumper is pressed it will still intake minerals (one of our operators is left handed and this was a simple solution without having two TeleOp programs)

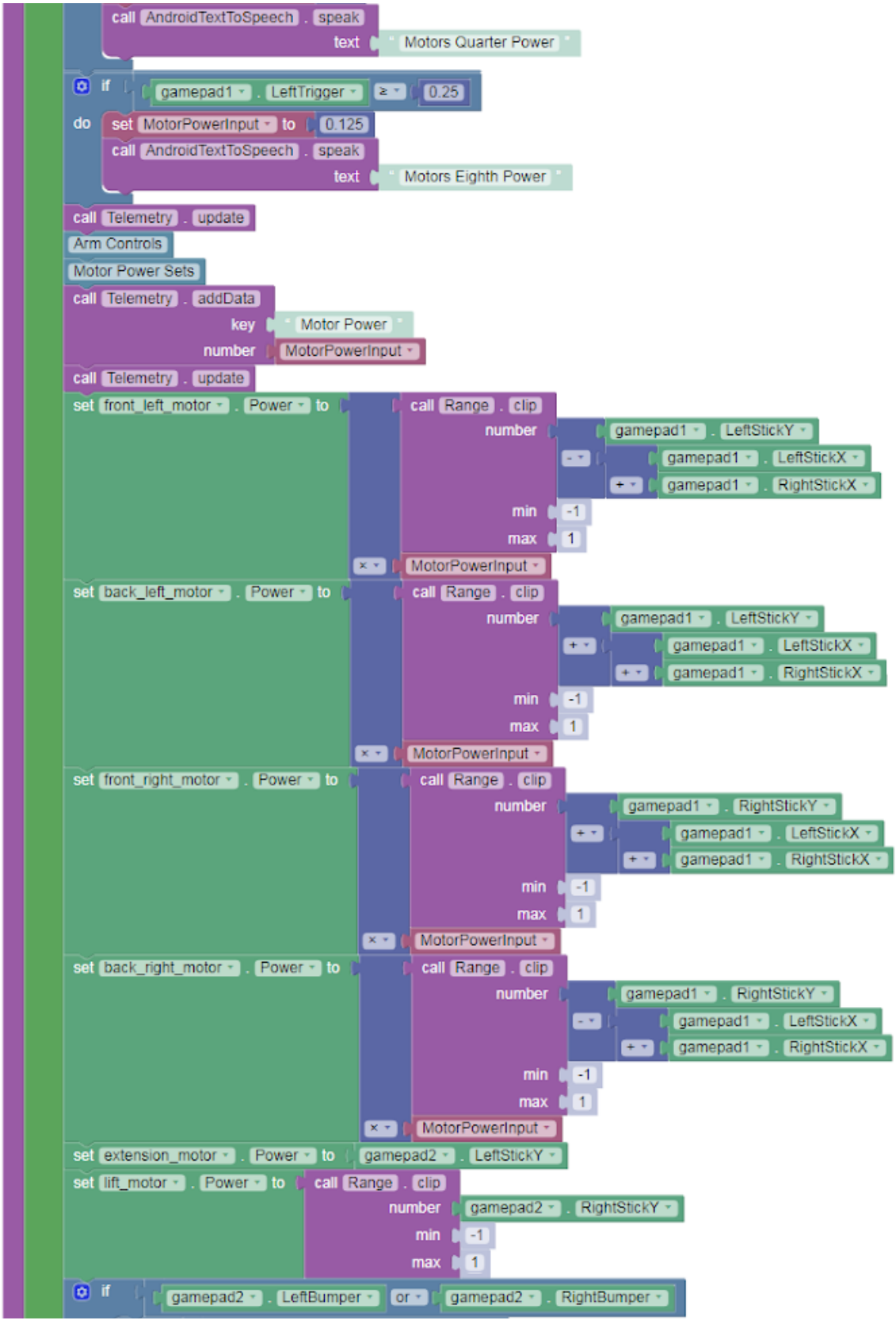
**triggers: we once again used both triggers to do the same function, but since the triggers are an analog input, we use an if statement that says that if they are pushed past 0.25 (about a quarter pressed) they act as true. Doing this allows us to use the triggers as buttons.

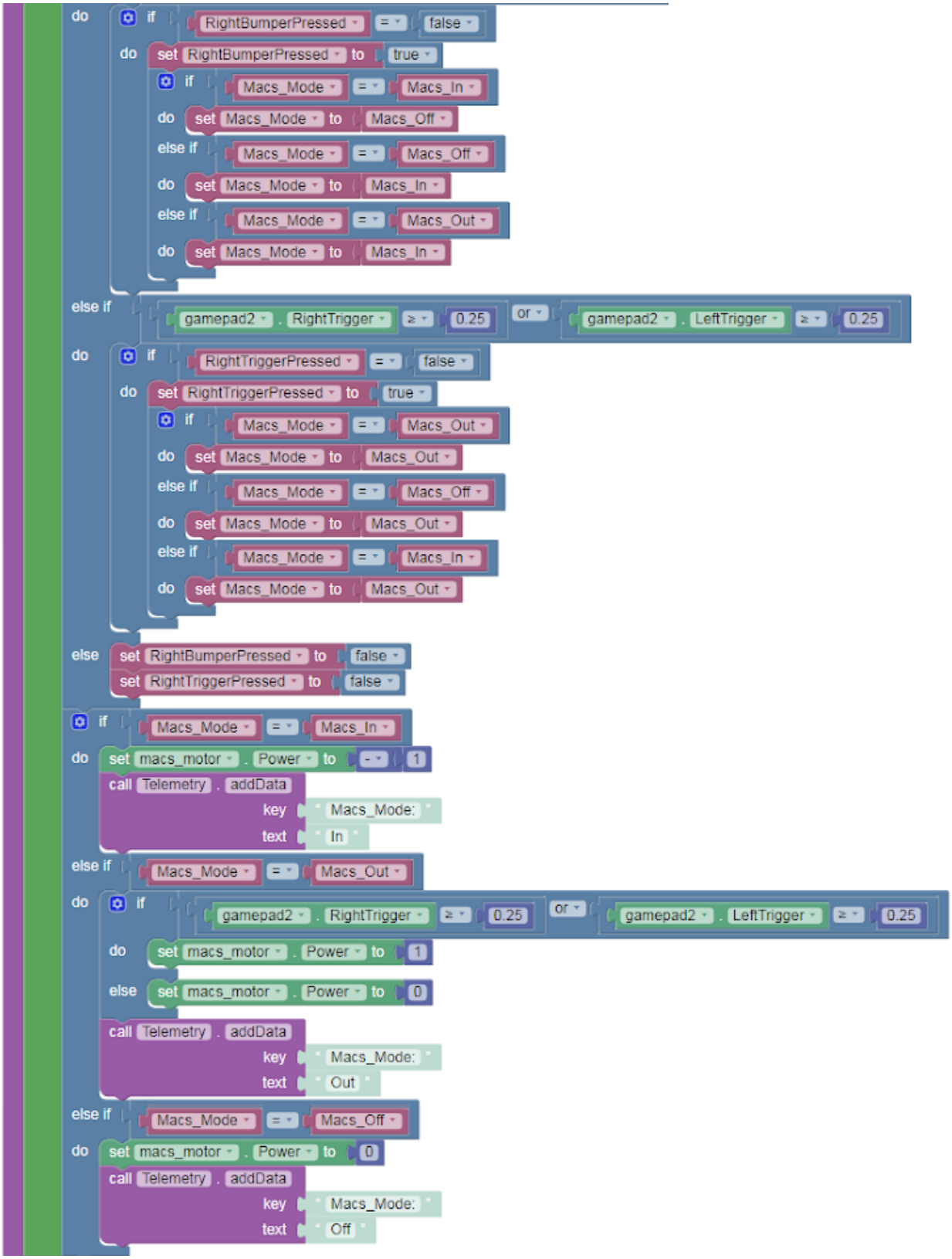
TELEOP EXAMPLES

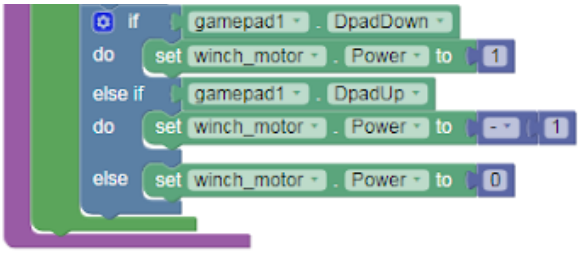
Figure 6.3 is our full TeleOp program, designed to work with a simple SetUp function which sets motor directions, modes, and zero power behaviors (and initializes Android Text to Speech, see page 33), as well as the variable MotorPowerInput, which controls robot speed throughout the program. The various TeleOp controls are designed to work with an arm, a robot-lifting mechanism, and four mecanum wheels with encoders.

Figure 6.3



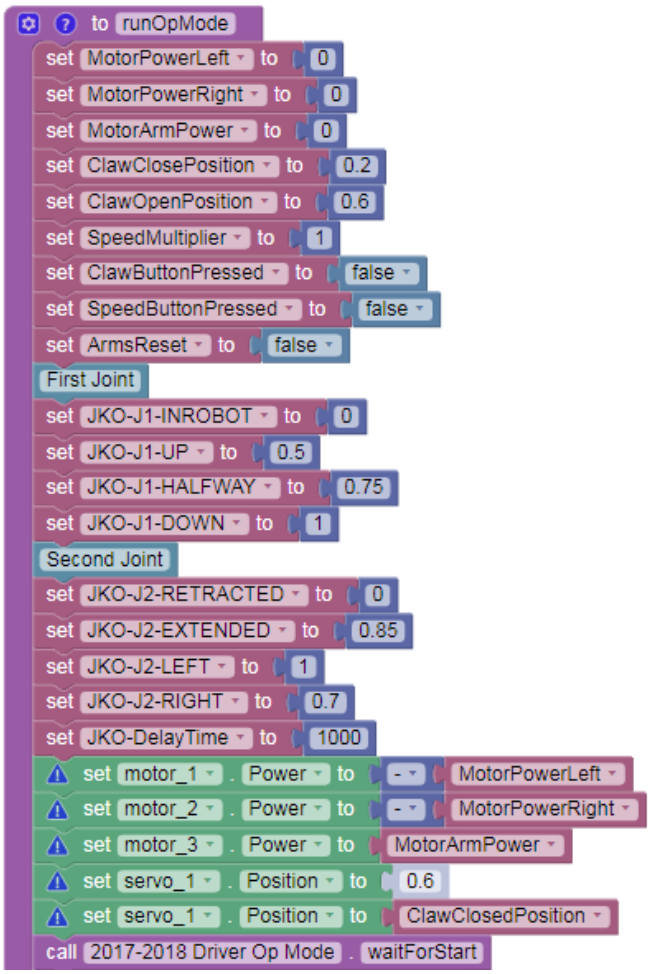

call AndroidTextToSpeech . speak
text " Motors Quarter Power "
if gamepad1 . LeftTrigger ≥ 0.25
do set MotorPowerInput to 0.125
call AndroidTextToSpeech . speak
text " Motors Eighth Power "
call Telemetry . update
Arm Controls
Motor Power Sets
call Telemetry . addData
key " Motor Power "
number MotorPowerInput
call Telemetry . update
set front_left_motor . Power to
call Range . clip
number gamepad1 . LeftStickY - gamepad1 . LeftStickX + gamepad1 . RightStickX
min -1
max 1
× MotorPowerInput
set back_left_motor . Power to
call Range . clip
number gamepad1 . LeftStickY + gamepad1 . LeftStickX + gamepad1 . RightStickX
min -1
max 1
× MotorPowerInput
set front_right_motor . Power to
call Range . clip
number gamepad1 . RightStickY + gamepad1 . LeftStickX + gamepad1 . RightStickX
min -1
max 1
× MotorPowerInput
set back_right_motor . Power to
call Range . clip
number gamepad1 . RightStickY - gamepad1 . LeftStickX + gamepad1 . RightStickX
min -1
max 1
× MotorPowerInput
set extension_motor . Power to gamepad2 . LeftStickY
set lift_motor . Power to
call Range . clip
number gamepad2 . RightStickY
min -1
max 1
if gamepad2 . LeftBumper or gamepad2 . RightBumper

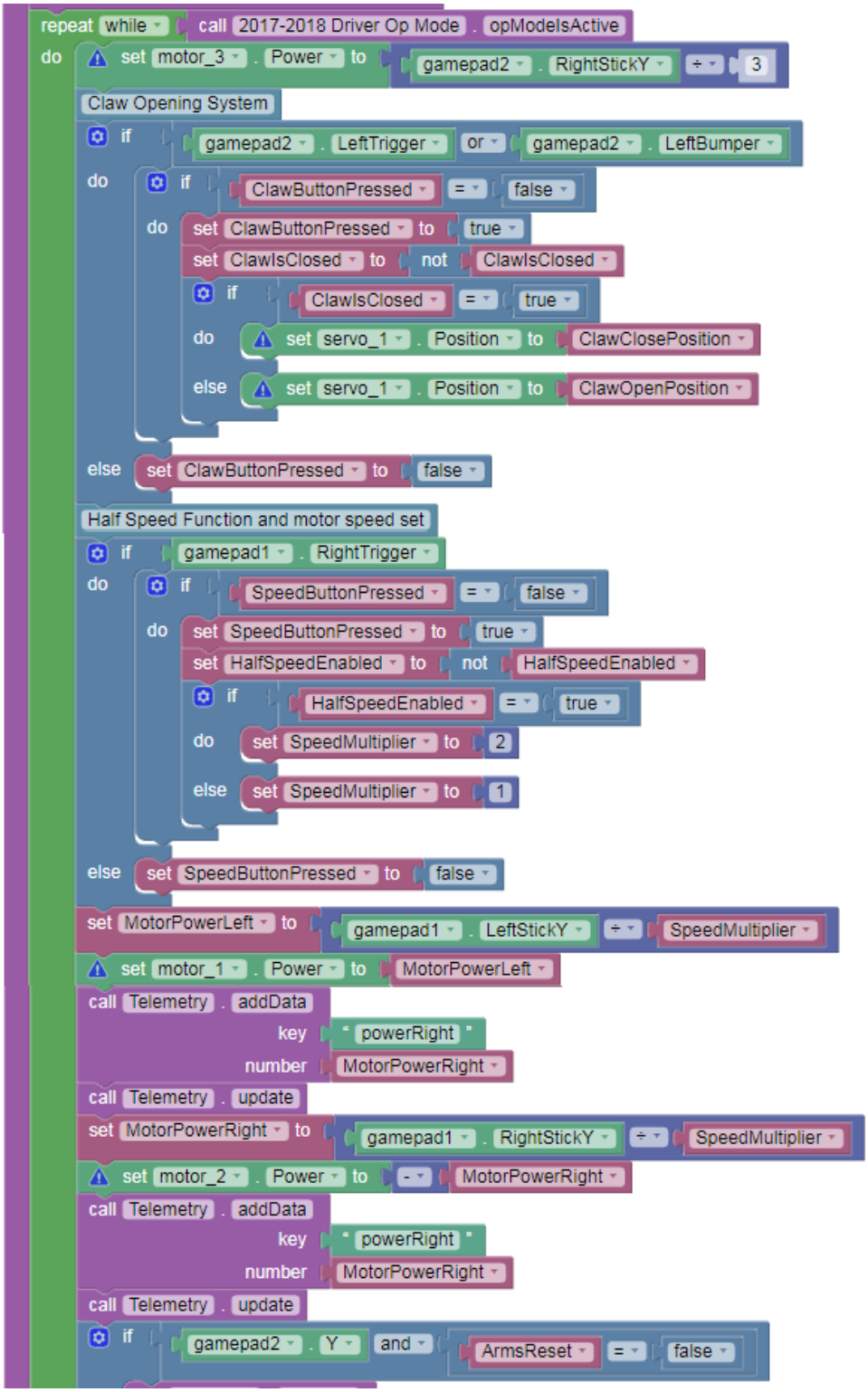

do
if RightBumperPressed = false
do set RightBumperPressed to true
if Macs_Mode = Macs_In
do set Macs_Mode to Macs_Off
else if Macs_Mode = Macs_Off
do set Macs_Mode to Macs_In
else if Macs_Mode = Macs_Out
do set Macs_Mode to Macs_In
else if gamepad2 . RightTrigger ≥ 0.25 or gamepad2 . LeftTrigger ≥ 0.25
do if RightTriggerPressed = false
do set RightTriggerPressed to true
if Macs_Mode = Macs_Out
do set Macs_Mode to Macs_Out
else if Macs_Mode = Macs_Off
do set Macs_Mode to Macs_Out
else if Macs_Mode = Macs_In
do set Macs_Mode to Macs_Out
else set RightBumperPressed to false
set RightTriggerPressed to false
if Macs_Mode = Macs_In
do set macs_motor . Power to - 1
call Telemetry . addData
key " Macs_Mode: "
text " In "
else if Macs_Mode = Macs_Out
do if gamepad2 . RightTrigger ≥ 0.25 or gamepad2 . LeftTrigger ≥ 0.25
do set macs_motor . Power to 1
else set macs_motor . Power to 0
call Telemetry . addData
key " Macs_Mode: "
text " Out "
else if Macs_Mode = Macs_Off
do set macs_motor . Power to 0
call Telemetry . addData
key " Macs_Mode: "
text " Off "

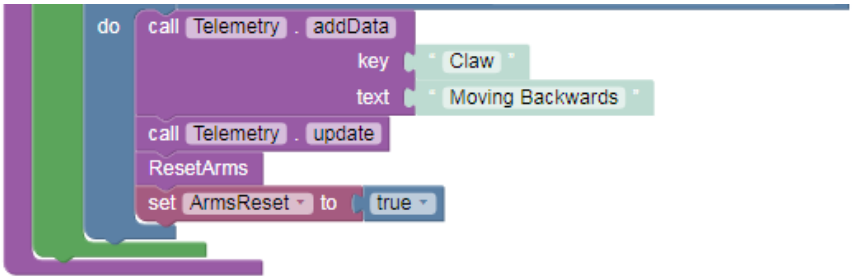


Our TeleOp has expanded over the years. For a more detailed look at some of the facets of this program refer to the TeleOp programming methods on page 35. A previous year's TeleOp was slightly simpler, made for a robot with omniwheels.

Figure 6.4




repeat while call 2017-2018 Driver Op Mode . opModeIsActive
do set motor_3 . Power to gamepad2 . RightStickY ÷ 3
Claw Opening System
if gamepad2 . LeftTrigger or gamepad2 . LeftBumper
do if ClawButtonPressed = false
do set ClawButtonPressed to true
set ClawIsClosed to not ClawIsClosed
if ClawIsClosed = true
do set servo_1 . Position to ClawClosePosition
else set servo_1 . Position to ClawOpenPosition
else set ClawButtonPressed to false
Half Speed Function and motor speed set
if gamepad1 . RightTrigger
do if SpeedButtonPressed = false
do set SpeedButtonPressed to true
set HalfSpeedEnabled to not HalfSpeedEnabled
if HalfSpeedEnabled = true
do set SpeedMultiplier to 2
else set SpeedMultiplier to 1
else set SpeedButtonPressed to false
set MotorPowerLeft to gamepad1 . LeftStickY ÷ SpeedMultiplier
set motor_1 . Power to MotorPowerLeft
call Telemetry . addData
key " powerRight "
number MotorPowerRight
call Telemetry . update
set MotorPowerRight to gamepad1 . RightStickY ÷ SpeedMultiplier
set motor_2 . Power to - MotorPowerRight
call Telemetry . addData
key " powerRight "
number MotorPowerRight
call Telemetry . update
if gamepad2 . Y and ArmsReset = false


do
call Telemetry . addData
key
Claw
text
Moving Backwards
call Telemetry . update
ResetArms
set ArmsReset to
true

APPENDIX: Ticks Per Inch Tutorial

Using Motor Encoders to Control FTC Robots

Kardia Robotics 11208

Introduction:

When programming robot movements during the Autonomous Period it is useful to be able to move the motors on the robot precisely. This can be accomplished by using motor shaft encoders, these will provide an electrical signal as the motor shaft turns. In Blocks and Java programming you can control your motors based on how many motor rotations you want to turn the shaft. The following pages include the math theory behind the process of programming the motor to move by revolutions, inch measurement, and degrees of rotation.

Ticks Per Revolution:

The first step in calculating how many ticks the motor will need in order to turn the correct number of rotations is to determine how many ticks per revolution the motor's encoder is designed for. This will vary from one encoder to another. Here is a short list of some of the common numbers:

- Modern Robotics (Tetrix) Motor Encoder_____1440 ticks per revolution
- AndyMark NeverRest 40 Encoder_____1120 ticks per revolution
- Rev Hex Motor Encoder_____2240 ticks per revolution

(these numbers are from Bruce Schafer's video on encoders)

These numbers will be used whenever programming your motors as they will control whether or not your motor will turn one rotation when you tell it to. If you tell a Tetrix motor to move 2240 ticks and expect it to move one revolution, you're going to have problems, because the Tetrix motor requires less ticks per revolution it will turn almost 2 revolutions. Contrariwise if you fed 1440 ticks to a Rev Hex Motor, then you will have the problem of the motor not turning a full revolution, since it's encoder is designed to work off of more ticks per revolution.

Gearing Ratio

The next thing to determine is the gearing ratio of your drive train; if it's 1:1 or direct drive, you can skip this section. To determine the gearing ratio, you just take the tooth count on your drive gear or sprocket and divide it by the gear or sprocket on your wheel. If you are driving your wheel faster than your motor shaft, you will have a gearing ratio above 1, if you are driving your wheel slower than your motor shaft, then you will have a gearing ratio below 1.

$$Gearing\ Ratio = \frac{Drive\ Teeth}{Wheel\ Teeth}$$

The reason that you even need to determine your gearing ratio is that if you have a system that isn't direct drive or 1:1 and you have your motor turn one revolution, then your wheel will either turn less or more than you intended it to. Using the gearing ratio can help solve that problem in a simple way that only needs to be determined once.

This same concept can also be applied to the use of a chain drive system, in the instance the number of teeth on the driven and driver sprocket will be used in the place of the numbers used for gear teeth.

Wheel Rotations Based on Gearing Ratio

The number of rotations your motor shaft will need to turn is going to be the number of wheel rotations you want divided by the gearing ratio. If you are driving your wheel faster than the motor shaft, your greater than 1 ratio will cause the motor shaft to rotate less than once per full wheel rotation. If you are driving your wheel slower than your drive shaft, then your ratio below 1 will cause your motor shaft to turn more than one rotation for every wheel rotation.

$$Motor\ Shaft\ Rotations = \frac{Wheel\ Rotations}{Gearing\ Ratio}$$

Now you can take these formulas to determine how many ticks you need to drive your motor to be able to turn the correct amount at the wheel. This is accomplished by taking the motor shaft rotations multiplied by the number of ticks per revolution.

This formula is represented below in two forms: One with it fully simplified, and one using the gearing ratio to put two steps together for determining shaft rotations and ticks in the same equation.

I have chosen not to include a derivation that fully picks apart the formula because the gearing ratio should only ever have to be determined once, since that won't change unless the motors have different gears attached to them, and thus wouldn't need to be included in a formula used in a program since it would make the program needlessly

complicated. It's far easier to simply set the gearing ratio as a constant in the setup of your program and call it in your move by inch function.

$$Ticks = \left(\frac{Wheel\ Rotations}{Gearing\ Ratio}\right) * Ticks\ per\ Revolution$$

$$Ticks = Motor\ Shaft\ Rotations * Ticks\ per\ Revolution$$

Moving Your Robot by Inch

Now that we've determined how to make the robot move by wheel rotations, we can try to make it move by telling the program how many inches to make the robot move. To do this we will first have to find the circumference of the wheel, this will be accomplished using pi and the diameter.

$$Circumference = Diameter * \pi$$

Now that we have the circumference we can determine how many wheel rotations we will need to move the robot by the measurement specified. To do this, we will take the distance to travel and divide it by the wheel circumference, this will tell you how many rotations we need the wheel to turn.

$$Wheel\ Rotations = \frac{Distance\ to\ Travel}{Wheel\ Circumference}$$

Now that you know how many wheel rotations you need, you can use the earlier ticks formula to determine how many ticks to feed to the motor.

This information can now be combined to produce a comprehensive formula that can be converted into a function for your program, which can handle all the calculations required into one step.

$$Ticks = \left(\frac{\frac{Distance\ to\ Travel}{Wheel\ Diameter * \pi}}{Gearing\ Ratio} \right) * Ticks\ per\ Revolution$$

Moving a Motor Shaft by Degree

If encoders are going to be used for moving an arm on the robot you can very easily use the formula from earlier to move the arm by revolutions, but if you want to move the arm by degrees then you can use the formula below. This formula was made using the fact that every revolution is 360 degrees, then taking this knowledge and simply using it in the same way that the last section uses the circumference of the wheel to determine shaft revolutions.

$$Ticks = \left(\frac{Degrees\ to\ Move}{360} \right) * Ticks\ Per\ Revolution$$

If you are using gears or a chain drive for your arm movement, then you will need to incorporate the gearing ratio for that system, this can be done by using the formula below.

$$Ticks = \left(\frac{\frac{Degrees\ to\ Move}{360}}{Gearing\ Ratio} \right) * Ticks\ Per\ Revolution$$

Programming Samples

Here are some programming samples from our autonomous program, the above formulas were used to create this program for moving by inch.

Sample 1a: Two Wheel Tank Drive system - Move By Inch

For Reference, we are using Tetrix motors for our drivetrain. We have a 32:24 drive to wheel gearing for the drive train and this gives us a gearing ratio of 1.33.

This first function allows you to tell the robot how many inches to move, and it will move both the left and right motor the specified amount.

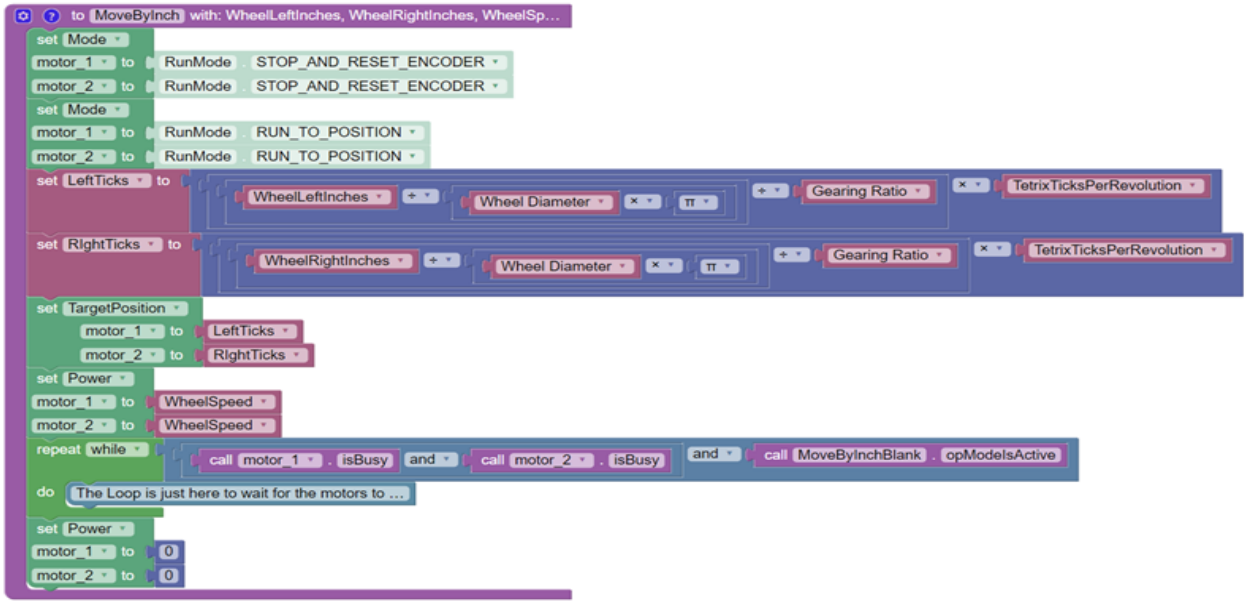
The first two blocks are motor mode set blocks (the dual motor blocks are under Actuators: DC Motor: Dual), the first block stops and resets the motor encoders. The reason you need to do this is that motor encoder tick values are cumulative, meaning that if you tell the motor to go to 2880 ticks and then tell it to go to 1440 it will go forward by two revolutions then backwards by one. The stop and reset encoder mode will stop the motor and set the value to zero, so that your robot will always move forward. The second block sets the motors' mode to run to position, this lets it use encoder values to move the motor shaft.

The Left Ticks and Right Ticks variable blocks are being set based off the formula from earlier.

The motor target position is then set to the value that was just set, and the motor speed was set to the variable WheelSpeed that was set in the values of the function inputs.

The while loop is then going to do nothing until the motors have reached their target position. The .isBusy block will return true when the motor is running to the target position and the OpModeIsActive block is there so that the motor will stop when the stop button is pressed on the driver station (Bruce Schaefer has a video explaining this). Once the motors have reached the target position the motor speeds

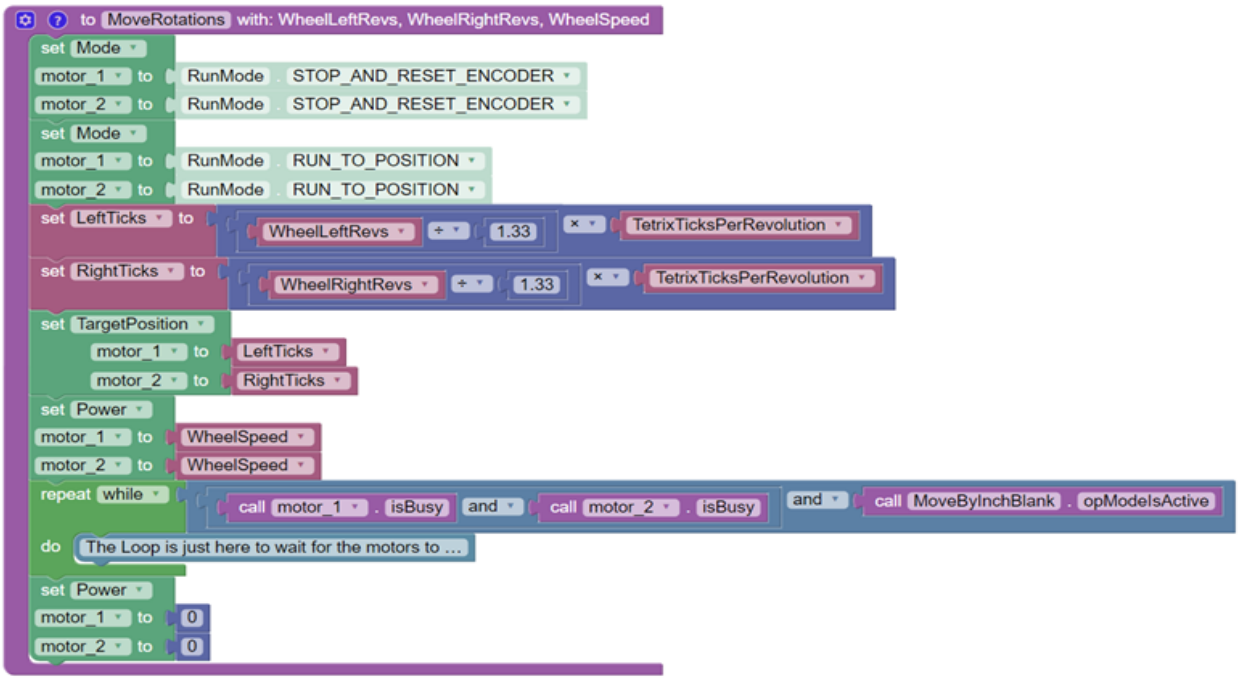
are set to zero and the function completes.



Sample 1a: Two Wheel Tank Drive system - Move Revolution

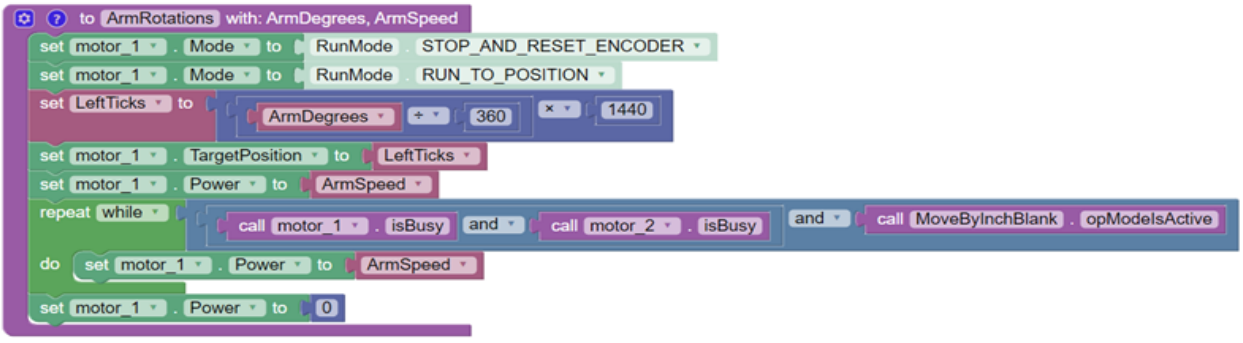
This Function uses all the same motor control blocks, the only difference is that the ticks formula is much less complicated and simply calculates how many ticks to send to the motor based on revolutions

instead of how many inches.



Sample 2: Single Motor - Move by Degree

The next function is one that just moves one motor by degree, this is useful for the arm that is on our robot to move down in between the two jewels.



Final Note

Kardia Robotics wishes you great success in your continued study of programming! We hope that this manual was helpful to you and your team, and that it inspires you on to greater heights in your programming career.

For more information about Kardia Robotics, FIRST Tech Challenge, Blocks programming, or any of the topics discussed or mentioned in this manual, please contact us at kardiastemclub@gmail.com, or visit our website at kardiarobotics.weebly.com. We would love to connect with you and answer any questions you may have.

Happy programming!

~Kardia Robotics

FTC Team 11208

3.02.21