Virtual Robot Simulator
Official Guidebook
1st Edition

Introduction

Hello! This book is an all-in-one guide for your first steps in robotics. Here you will learn the essentials to controlling, piloting, and navigating your robot towards your future STEM aspirations. I understand that starting out in robotics, learning all these new terms and phrases can be confusing. FIRST Tech Challenge (FTC) requires a lot of space and funds that not alot of people necessarily have. Using the VRS will let you get a good feel of the actual robot experience! You will be able to program, pilot, and do so much more by the end of this book. We cover everything in an easy and understandable way. Don't let the simple wording make you think this book is for only beginners though, I cover a lot of advanced concepts as well. By the end of this book, you will be as, if not more, experienced than the average FTC player as well as having a helpful reference guide to aid you on your robotics endeavors.

I have been teaching robotics to up and coming learners like you for several years now. The VRS has been one of my primary teaching tools and I know it inside out. I have even contributed to parts of its development! I am certain that you will be as well taught as my students, many of whom are part of successful FTC teams.

FTC

So, what is FIRST Tech Challenge (FTC) anyways? Well at its core, FTC is a robotics competition for grades 7-12 where students design, build, and program robots to compete in head-to-head challenges. Teams of up to 15 students learn STEM skills, engineering principles, and teamwork while working with adult mentors. The program emphasizes innovation, hard work, and community outreach. FTC’s core value is Gracious Professionalism. It's about creating a positive and supportive community where everyone feels valued, regardless of the outcome of the competition. It's about winning with grace and losing with dignity.

Centerstage Game and Basic Rules

Centerstage was the FTC (FIRST Tech Challenge) game for the 2023-2024 season. It was a high-energy competition that challenged teams to design, build, and program robots to perform various tasks on a themed stage.

Official centerstage introduction:
https://www.youtube.com/watch?v=6e-5Uo1dRic

"Welcome to the FIRST Tech Challenge 2023 - 2024 Season Game CenterStage presented by RTX. A team consists of up to two driver operators, a human player, a coach, and a robot. Each match is played with four randomly selected teams, two per alliance. Each alliance is allowed one human player. Your opponent for one match may be your partner for another. Robots must be built from maQterials specified in the game manual and fit within an 18 inch sizing tool and may expand after the match begins. The primary scoring element is a plastic hexagonal shaped pixel three inches across by one half inch thick. There are 64 white pixels and ten each yellow, green and purple pixels. Teams may construct

custom game and scoring elements, including paper drones and team props. The game is played on a 12 foot square playing field with a foam tile floor and one foot high walls. Two trusses are located mid-field. Between the trusses is the hinged stage door. In the back of the field are the backdrops, one for each alliance. Beneath the backdrops are taped off backstage areas. In the front corners are taped off wings. In each quadrant of the field are three separate spike marks. Just inside of the front wall are white pixel locations stripes and outside of the front wall there are three taped off landing zones. AprilTags are located in the field wall and both backdrops to aid navigation. Blue and Red Alliance stations are on the left and right sides of the field. And in front of those are the red and blue human player stations. Before each match, pixels are stacked next to each player station and on the inside of the front wall. Four pixels are placed on the spike marks. However, team props may be substituted for the spike mark pixels. Teams place their robots on the field, touching the sidewalls. Each team may pre-load one yellow and or one purple pixel onto their robot. Teams may also pre-load one drone onto their robot. The spike mark pixels and team props are then randomized. The field and players are now ready.

The match begins with a 30 second autonomous period. During this time, teams may attempt to score using pre programmed instructions and sensor inputs. A purple pixel placed on the spike mark tape earns ten points. However, if the spike mark tape has a team prop, placing a purple pixel will earn 20 points. Each pixel in the backstage earns three points and each pixel placed on the backdrop earns five points. If a yellow pixel is placed on the backdrop in the location indicated by the spike mark pixel, it earns ten bonus points or 20 points if a team prop was used as an indicator. Robots parked in the back stage earn five points. Pixels scored in the autonomous period will also earn points at the end of the driver controlled period.

Following the autonomous period is the two minute driver control period. During this time, human players may introduce new pixels into the wings. Each pixel that is placed backstage earns one point and each pixel on the backdrop earns three points. Each mosaic of three identical or three different pixels earns ten points. If an alliance can build pixels above the set lines, they earn a ten point bonus for each line they cross.

The last 30 seconds of the driver controlled period is the end game. Robots may continue scoring pixels and there are also ways to earn bonus points. Robots may launch drones over the truss or stage door and into the landing zones. Depending on where they're parked, they can earn 10, 20 or 30 points. Robots parked in the backstage area earn five points. And a robot suspended by the rigging earns 20 points. There are many ways to score in CenterStage, but there are also rules that, if not followed, will deduct points from your alliance. Robots may not limit the upward motion of the stage door. Robots may not descore pixels from the opposing alliance's backdrop. Intentionally damaging another robot is not allowed. A robot may not affect the flight of an opposing alliance's drone. A robot may not make contact with an opponent suspended from the truss."

This has been a summary of CenterStage presented by RTX.

For complete rules, please read both game manuals and check the Q&A forum and always remember the most important rule of FIRST Tech Challenge: Gracious Professionalism.
https://drive.google.com/file/d/1RR5HBRvqcS67P0QPw_A9H3rHwxnqotO6/view?usp=sharing
https://drive.google.com/file/d/1FvtJtaHsUxVZrevNHrTQlAkM7E7zMbcN/view?usp=sharing

Powerplay Game and Basic Rules

POWERPLAY was the FTC (FIRST Tech Challenge) game for the 2022-2023 season. It was themed around energy and sustainability, with teams competing to power their innovations forward.

Official POWERPLAY introduction:
https://www.youtube.com/watch?v=HsitvZ0JaDc.

"Welcome to the first tech challenge 2022-2023 season game POWERPLAY presented by Raytheon Technologies. A team consists of up to two driver operators, a human player, a coach, and a robot. Each match is played with four randomly selected teams two per alliance but each alliance is only allowed one human player. Your opponent for one match may be your partner for another. Robots must be built from materials specified in the game manual and fit within an 18 inch sizing tool but may expand after the match begins. The game element is a four inch diameter by five inch tall plastic cone. There are 60 cones, 30 red and 30 blue. Teams may also use their custom designed beacons in gameplay. The game is played on a 12 foot square playing field with a foam tile floor and one foot high walls. Various sized junctions are placed across the field including nine ground junctions. In addition, there are eight low, four medium, and four high junctions. These junctions are mounted on flexible springs and might not be perfectly vertical. Taped off substations are centered on both sides. In each corner are taped off terminals. In the front and back of the field are taped stripes to help robots find cone stacks.

There are four signals placed on the field. Four unique navigation images are positioned on the field wall. Outside of the field are the red and blue alliance stations. Before each match cones are stacked in the substation storage areas

and inside the playing fields. Custom beacons are also placed in the substation storage areas. Teams place their robots on the field touching the wall between the substations and terminals. Each team may preload one cone into their robot.

The three-sided signal is then randomized. Each image references a specific parking target during match play. If a team designs their own signal sleeve it can be used to gain additional points. The field and players are now ready.

The match begins with a 30-second autonomous period. During this period there are a number of ways for teams to score using only pre-programmed instructions and sensor inputs. Each cone secured on a junction earns points: two points for the ground junction, three points for the low junction, four points for the medium junction, and five points for the high junction. A robot parked in their alliance substation or terminal earns two points. If a robot sensor correctly reads the randomized signal, it can park in the corresponding signal zone to earn 10 points, however parking in the proper signal zone shown by a custom signal sleeve will earn 20 points.

Following the autonomous period is the two-minute driver controlled period. During this period human players may introduce new cones into the substations. Each cone that is placed in its matching color terminal earns one point. Securing cones in junctions earns the same points as during the autonomous period: two points for the ground junction, three points for the low, four points for medium, and five points for the high junction.

The last 30 seconds of the driver controlled period is the end game. Robots may continue scoring cones but there are also ways to earn bonus points. Alliances earn points for owning a junction. This can be accomplished in two ways. An alliance owns the junction if their colored cone is scored on top. This earns an

additional 3 points. The junction can also be owned by capping it with a beacon. This earns 10 additional points and prevents the opposing alliance from recapturing that junction. An alliance earns 20 points by completing a circuit. This is done by owning a continuous path of junctions from one terminal to the other. If a robot is parked in either of their alliance terminals it earns two points. All cones scored during the autonomous period count again if they remain in place at the end of the match.

There are many ways to score in PowerPlay but there are also rules that if not followed will deduct points from your alliance. For example, robots may not score opposing alliance cones or beacons. A robot may not block or interfere with an opposing alliance attempting to score. A robot must be completely outside of a substation in order to score a cone or a beacon. Robots may control or possess only one cone and one beacon at a time. Scoring elements may not be launched and robots may not deliberately remove game elements from the playing field."

This has been a brief summary of this season's game PowerPlay presented by Raytheon Technologies. For complete rules please read both game manuals and check the Q and A forum and always remember the most important rule in FIRST Tech Challenge: Gracious Professionalism.
https://drive.google.com/file/d/1Fy1_RipNoozW6UZRPr8-d8LM6CoY2FG_/view?usp=sharing
https://drive.google.com/file/d/1JyxnCzRrwgNgClJ-9uKfrDMIiByirxpf/view?usp=sharing

VRS

What is VRS? VRS, or Virtual Robot Simulator, is a student developed simulator for FTC. It’s meant to let users program and test robots virtually, removing

physical and financial constraints. Furthermore, it can be a learning tool to aid newcomers in robotics. The VRS is still very much in development right now and we hope to add more features in the future!

https://centerstage.vrobotsim.online/homepage.html

Programming

Programming is the mind of your robot. It tells your robot how to think and move. Without programming, your robot would just be a hunk of metal and wires. Furthermore, programming is an increasingly essential skill in our high tech world. Experience and knowledge in programming can be a major boost towards obtaining a high-paying job.

With respect to robotics, two main types of programs are present: autonomous and manual. Autonomous programs are designed to control the robot without the help of human intervention. This is useful when performing predictable and repetitive activities. With enough refinement and testing, automated programs can prove to be far more efficient and precise than manually controlled robots. But, what if the activity isn't predictable? What if there is an object in the way or your partner's movement intersects with yours?

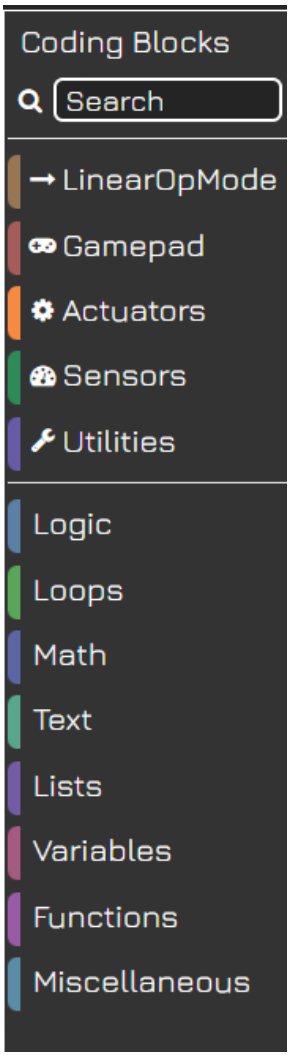
Well, this is where manually controlled robots excel. Unlike autonomously controlled robots, manually controlled robots are directly affected by the pilot's (that's you!) actions. Like playing a video game, a button press or a joystick flick will correlate to the corresponding movement of the robot. This permits rapid changes in strategy and movement, adjusting for changes in the game. Here the program largely defines what buttons do what and, in some cases, active series of predefined commands, making controlling the robot easier.

To master FTC, you will need to learn both types of programming in order to succeed.

You can program online on https://centerstage.vrobotsim.online/programpage.html. The website provides a platform for users to learn and practice programming robots using either Blocks or Java. It provides a virtual environment to test and experiment with your code.

User Interface

Coding Blocks



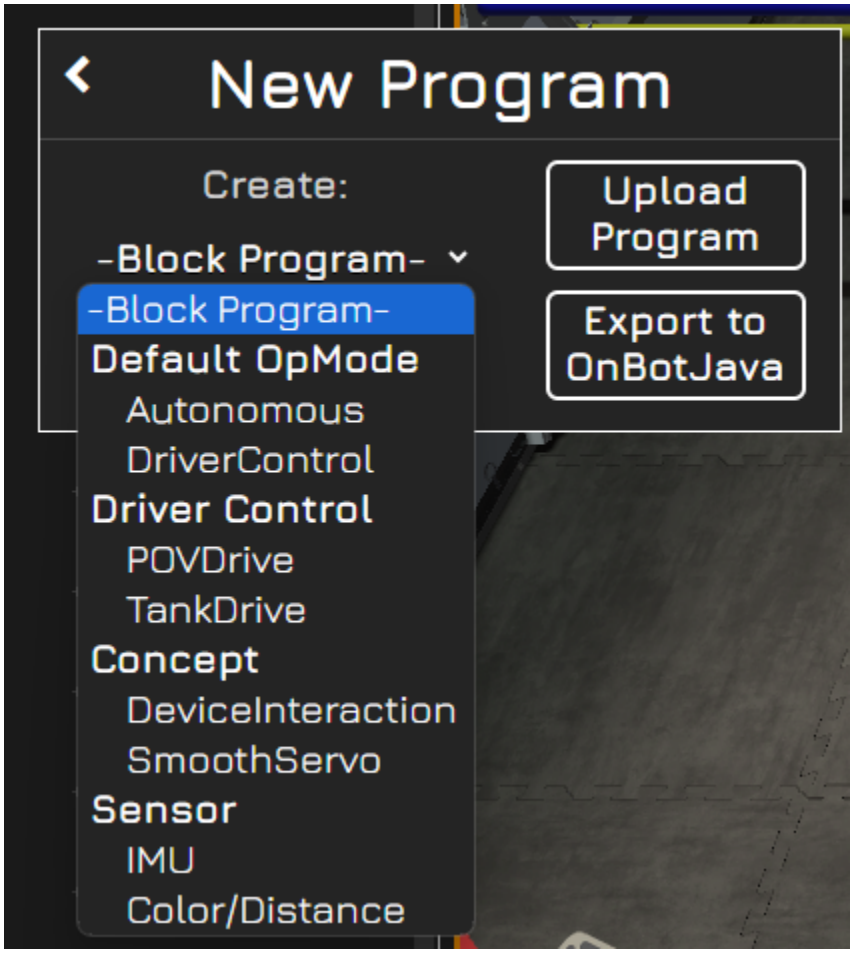
This where all the available blocks can be accessed from.

Blocks/OnBotJava

This will let you switch between coding in Blocks and OnBotJava.

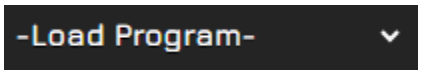
New Program

Pressing this will let you create a new blocks or OnBotJava program. For Blocks, several premade programs will be available to give you a headstart in your coding.



Saving As

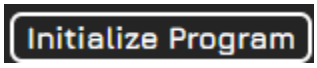
When you create a program you want to keep click this button to save the program.
If you do you can select the program with



and you will have the option to

save, delete, and download the program using the respective buttons.

Initialize Program



Lets you initialize the program, usually starting your autonomous program as well. After pressing it, you will have the option to start the program, typically beginning your manual control, and stop the program.

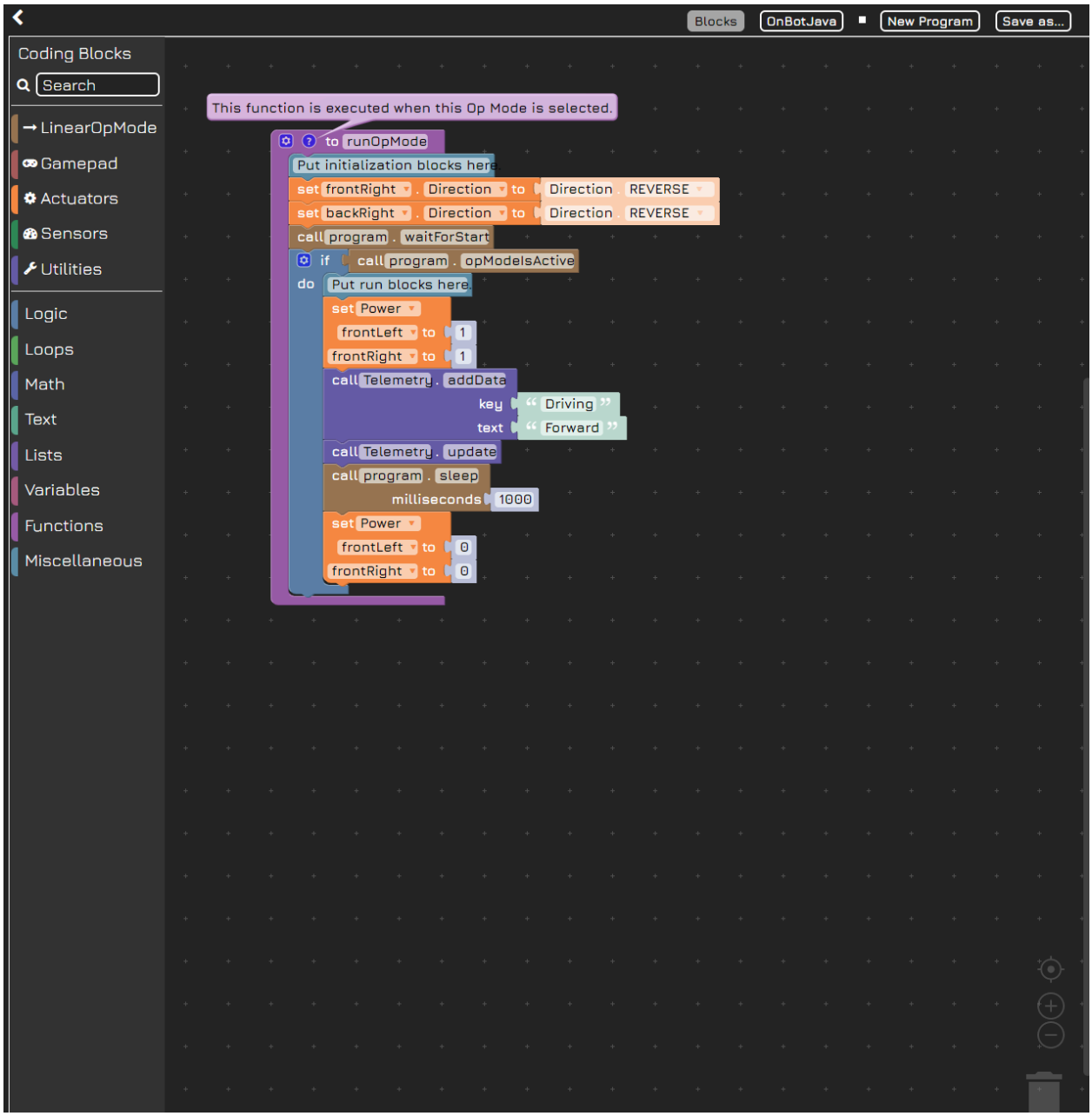
Reset Field

resorts the game field

Blocks

Block programming is a visual programming method that uses graphical blocks to represent code. Instead of typing lines of code, users drag and drop these blocks together to create programs. It's a beginner-friendly introduction to programming, as it simplifies the learning process.

Think of it like building with Lego bricks: you snap different blocks together to create a structure. In block programming, you snap together code blocks to create a program.

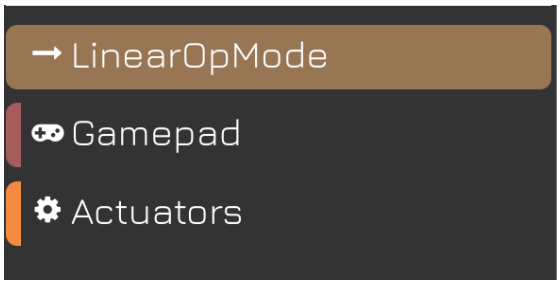


Once you've assembled your blocks into a program, the computer executes each function in order, top to bottom. It follows the logic you've established by

connecting the blocks. Some blocks, like logic and loop blocks can alter the order or rate the blocks are executed, helping to create more complex commands.

LinearOpMode

This set of blocks primarily relate to the program itself. It doesn't affect how the code affects the robot, but instead how the code affects the logic or running of itself.



waitForStart

Used to pause the program execution until the "Start" button is pressed. This ensures that your robot's actions begin at the same time as the match starts.

Key points:

- Purpose: To synchronize the start of your robot's program with the match start.
- Behavior: Pauses the program until the "Start" button is pressed.
- Placement: Typically placed at the beginning of an OpMode.

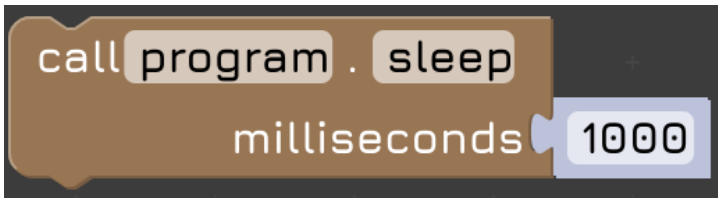
idle

Used to pause the current thread execution for a short time. This allows the other threads to run.

Key points:

- Purpose: To enhance program performance by reducing the amount of active threads running.
- Behavior: Pauses the program for 1 millisecond
- Usage: Typically used for a thread not performing any active work. This puts it into a state of readiness, helping to conserve energy and optimize CPU utilization.

sleep



Used to pause the program execution for a specified duration.

Key points:

- Purpose: To introduce a delay in the program's execution.
- Units: The duration is measured in milliseconds.
- Usage: Commonly used for timing actions, synchronizing movements, or creating pauses between executions.

opModeIsActive

A conditional block that checks whether the current OpMode is still running. It returns a boolean value: true if the OpMode is active, false otherwise.

Key points:

- Purpose: To determine if the OpMode is currently running.
- Return value: A boolean value indicating the OpMode's active state.
- Usage: Often used within loops to continuously check the OpMode's status.

isStarted

A conditional block that checks if the Opmode has been started. It returns a boolean value: true if the OpMode has started, false otherwise.

Key points:

- Purpose: To determine if the Opmode has been started.
- Return value: True if the Opmode has been started, false otherwise.
- Usage: Typically used within a loop to check if the start has begun so that it can execute subsequent code.

isStopRequested

A conditional block that checks if a stop has been requested for the OpMode. This is typically initiated by pressing the “Stop” button.

Key points:

- Purpose: To determine if the OpMode should be terminated.
- Return value: True if a stop has been requested, false otherwise.

- Usage: Typically used within a loop to check for a stop request and perform necessary cleanup actions.

getRuntime

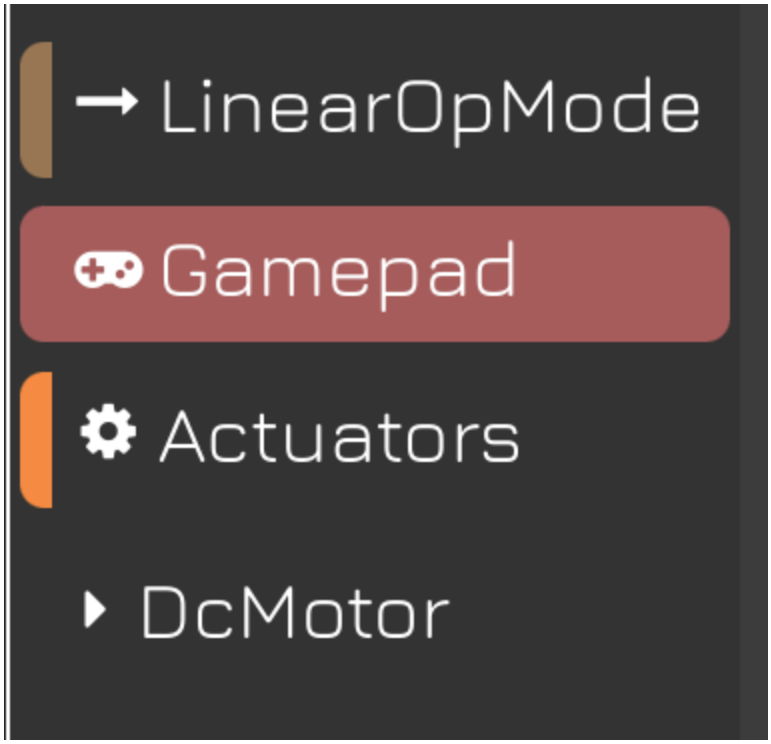
Provides the elapsed time since the OpMode was initialized. This means it starts counting as soon as the OpMode is selected, even before the waitForStart block is reached.

Key points:

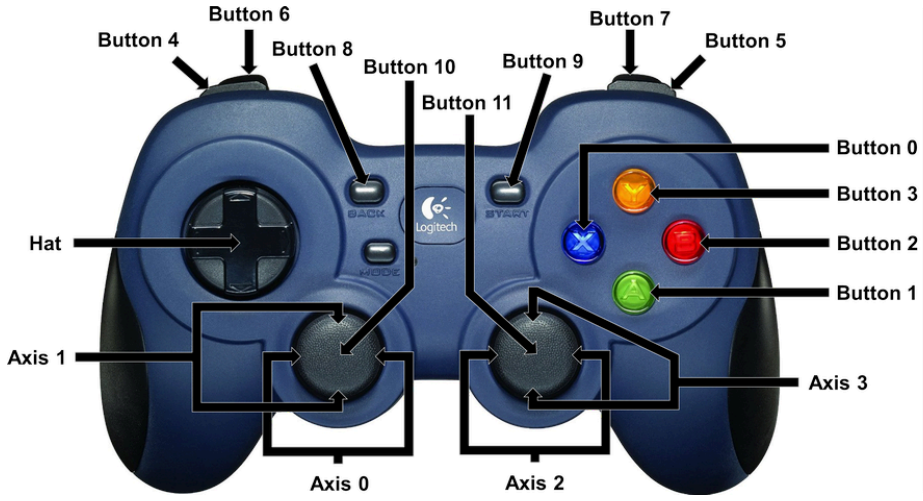
- Start time: The timer begins when the OpMode is initialized, not when the match starts.
- Return value: Returns a double value measured in seconds.
- Usage: You can use this block to measure time-based events, implement delays, or calculate rates.

Gamepad

A series of blocks that allow you to read gamepad button presses, joystick positions, and trigger values to command your robot's movements and actions.



Here is a diagram laying out what each button is mapped to:



Axis 0	Left Joystick X
Axis 1	Left Joystick Y (inv)
Axis 2	Right Joystick X
Axis 3	Right Joystick Y (inv)
Button 0	X
Button 1	A
Button 2	B
Button 3	Y
Button 4	Left Bumper
Button 5	Right Bumper
Button 6	Left Trigger
Button 7	Right Trigger
Button 8	Back
Button 9	Start
Button 10	Left Joystick (click)
Button 11	Right Joystick (click)
Hat	Directional Pad (X, Y)

There are four main types of gamepad blocks:

- Button blocks: These blocks represent the various buttons on the gamepad (A, B, X, Y, D-pad, bumpers, triggers, etc.). They return a boolean value (true or false) indicating whether the button is pressed or not.

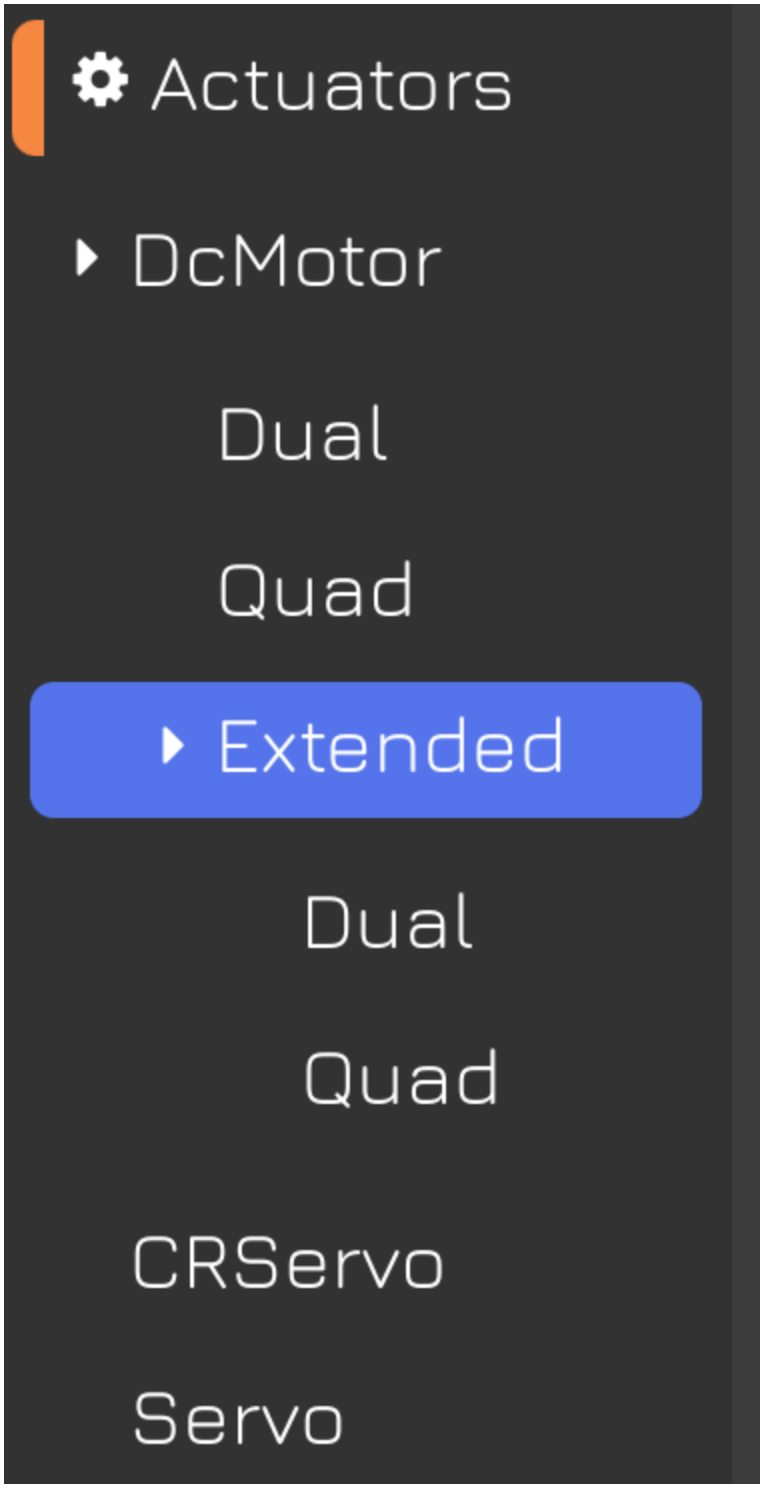
- Joystick blocks: These blocks provide the X and Y coordinates of the left and right joysticks. The values range from -1 to 1, representing the full range of motion of the joystick.

- Trigger blocks: These blocks provide a value between 0 and 1, representing the trigger's position.

- The status of the gamepad. There is only one block of this type. AtRest block returns true if all analog sticks and triggers are in their rest position.

Actuators

Actuators convert electrical energy into mechanical motion, enabling your robot to perform various tasks. They are mainly categorized as DCMotor, CRServo, and Servo.


Actuators
DcMotor
Dual
Quad
Extended
Dual
Quad
CRServo
Servo

DcMotor

DcMotor is a fundamental component in FTC (FIRST Tech Challenge) robots, providing the essential power for various movements and actions. It's a direct current electric motor that converts electrical energy into mechanical energy.

CurrentPosition

Provides the current rotational position of a motor.

Key points:

- Return Value: A floating point value in ticks.
- Usage: Crucial for precise motor control, closed-loop control, and position-based actions.

Mode

Gets the mode of a motor.

Key points:

- Return Value: RunMode. The four possible modes are RUN_WITHOUT_ENCODER, RUN_USING_ENCODER, RUN_TO_POSITION, and STOP_AND_RESET_ENCODER.
- Usage: Crucial for precise motor control, closed-loop control, and position-based actions.

Power

Gets the power of a motor.

Key points:

- Return value: A floating point value from -1.0 to 1.0
- Usage: Used to check the power of a motor. Crucial for precise motor control.

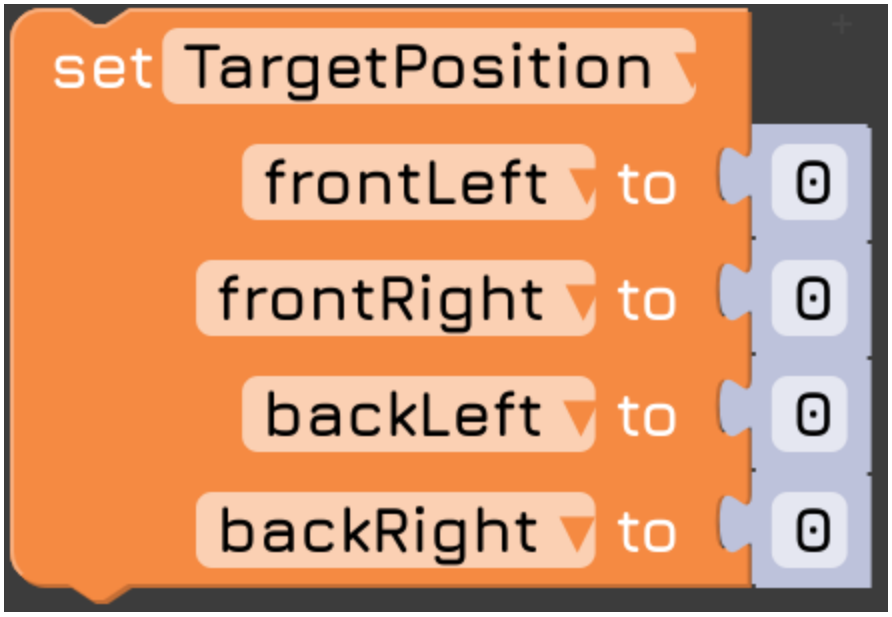
PowerFloat

Determines if the power of a motor is a float number.

Key points:

- Return value: True if motor power is not an integer, false otherwise.
- Usage: Used to check if the power of a motor is a whole number. Crucial for precise motor control.

TargetPosition



Sets the target position of four motors.

Key points:

- Value: A floating-point value in ticks.
- Usage: A more precise method to control the position of the motors.

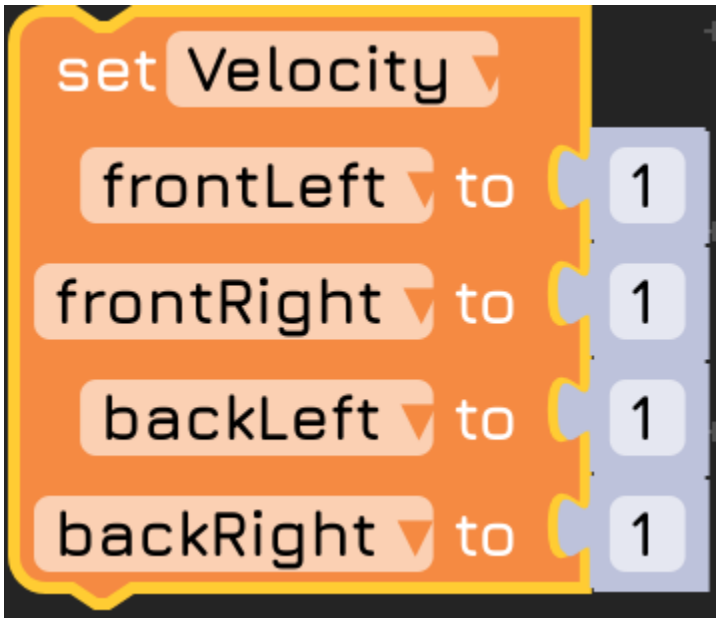
TargetPositionTolerance

Sets the target position tolerance of four motors.

Key points:

- Value: A floating-point value in ticks.
- Usage: Controls the margin of error of the motors
- Can’t find it? You will need to choose the set TargetPosition block, then choose from the dropdown list.

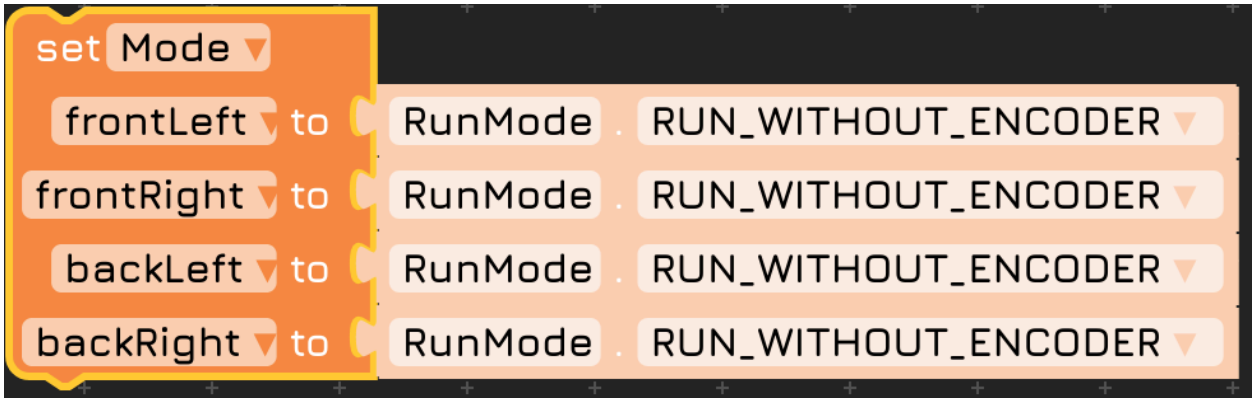
Velocity



Sets the velocity of four motors. Not all motors support this feature.
Key points:

- Value: A floating-point value in ticks per second
- Usage: Controls the speed of the motors more precisely.
- Can’t find it? You will need to choose the set TargetPosition block, then choose from the dropdown list.

Mode

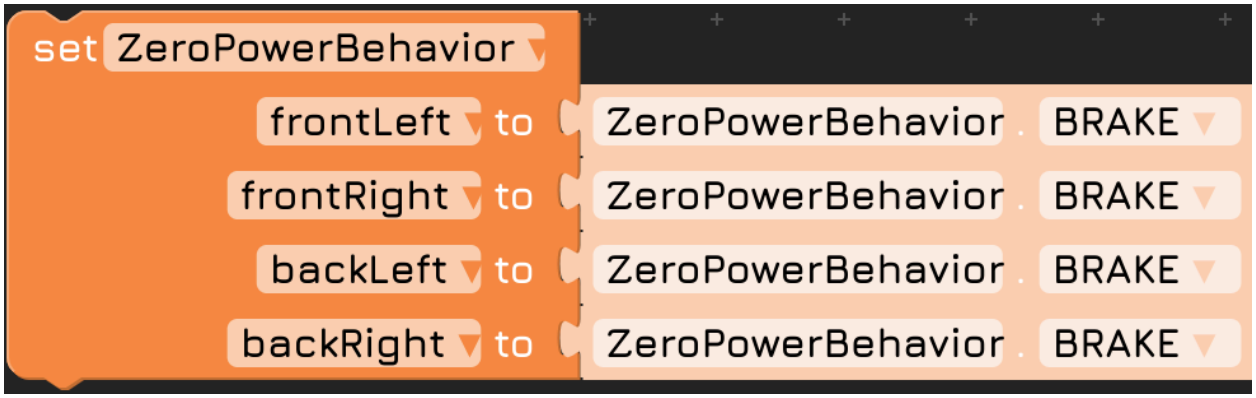


Sets the mode of four motors. The four modes are RUN_WITHOUT_ENCODER, RUN_USING_ENCODER, RUN_TO_POSITION, and STOP_AND_RESET_ENCODER.

Key points:

- DcMotor.RunMode.RUN_WITHOUT_ENCODER: This is the default mode. The motor runs at the specified power level without using encoder feedback. Suitable for basic open-loop control.
- DcMotor.RunMode.RUN_USING_ENCODER: Uses encoder feedback to control motor speed. Allows for more precise speed control. Often used in conjunction with PID control.
- DcMotor.RunMode.RUN_TO_POSITION: Runs the motor to a specified target position using encoder feedback. Ideal for precise positioning tasks.
- DcMotor.RunMode.STOP_AND_RESET_ENCODER: Stops the motor and resets the encoder count to zero. Used to initialize the motor's position before starting a new movement.

ZeroPowerBehavior



Sets the state of four motors when their power is set to zero. The two possible values are BRAKE or FLOAT.

Key points:

- Brake: When the motor's power is set to zero, the motor is actively braked, bringing it to a quick stop. Ideal for situations where you want the motor to stop quickly and hold its position. This is commonly used for mechanisms like arms or lifts.
- Float: When the motor's power is set to zero, the motor coasts freely, allowing the motor's axle to rotate without generating torque. Suitable when you want the motor to coast freely after setting power to zero, allowing momentum to carry the mechanism. This can be useful for wheels or mechanisms where controlled deceleration is not required.

Extended

More blocks for motors that are less commonly used. Nothing in this folder is implemented. DO NOT USE.

CRServo

A Continuous Rotation Servo (CRServo) is a type of servo motor designed to rotate continuously in both directions. Unlike standard servos, which have a limited range of motion, CRServos can spin indefinitely.

Key Characteristics:

- Continuous rotation: Can spin in both clockwise and counterclockwise directions.
- Speed control: Typically controlled by setting a power level between -1.0 (full speed reverse) and 1.0 (full speed forward).
- No positional feedback: Unlike standard servos, CRServos generally don't provide feedback on their position.

Direction

Provides the current direction of a servo.

Key points:

- Return value: A string value of “FORWARD” or “REVERSE”.
- FORWARD: The power the servo receives is unaltered.
- REVERSE: The power the servo receives is multiplied by -1, reversing its sign.
- Usage: Used to check the current direction of a servo. Crucial for precise motor control and ease of programming.

Power

Provides the power of a servo.

Key points:

- Return value: A floating point value from -1.0 to 1.0
- Usage: Used to check the power of a servo. Crucial for precise motor control.

set fullservo.Direction to

Controls the direction of the servo. The only directions are “FORWARD” or “REVERSE”

Key points:

- FORWARD: The power the servo receives is unaltered.
- REVERSE: The power the servo receives is multiplied by -1, reversing its sign
- Usage: To make setting and controlling the power to servo easier while programming and driving.

Set fullservo.Power to

Controls the power the servo receives, determining its speed and direction.

Key points:

- Value range: Accepts a floating-point value between -1.0 and 1.0.
- Direction: Positive values move the servo forward, negative values move it backward.
- Magnitude: The absolute value of the power determines the servo's speed.

Servo

A type of motor that can be controlled to move to a specific angle. It includes a built-in feedback mechanism that allows it to accurately position itself. This makes servos ideal for applications requiring precise control, such as robotics, RC models, and automation.

set servo.Direction to

Controls the direction of the servo. The only directions are "FORWARD" or "REVERSE".

Key points:

- FORWARD: The power the servo receives is unaltered.
- REVERSE: The power the servo receives is multiplied by -1, reversing its sign.
- Usage: To make setting and controlling the power to servo easier while programming and driving.

set servo.Position to

Sets the target position of the servo.

Key points:

- Value: A floating-point value in ticks.
- Usage: A more precise method to control the position of the servo.

Direction

Provides the current direction of a servo.

Key points:

- Return value: A string value of “FORWARD” or “REVERSE”.
- FORWARD: The power the servo receives is unaltered.
- REVERSE: The power the servo receives is multiplied by -1, reversing its sign.
- Usage: Used to check the current direction of a servo. Crucial for precise servo control and ease of programming.

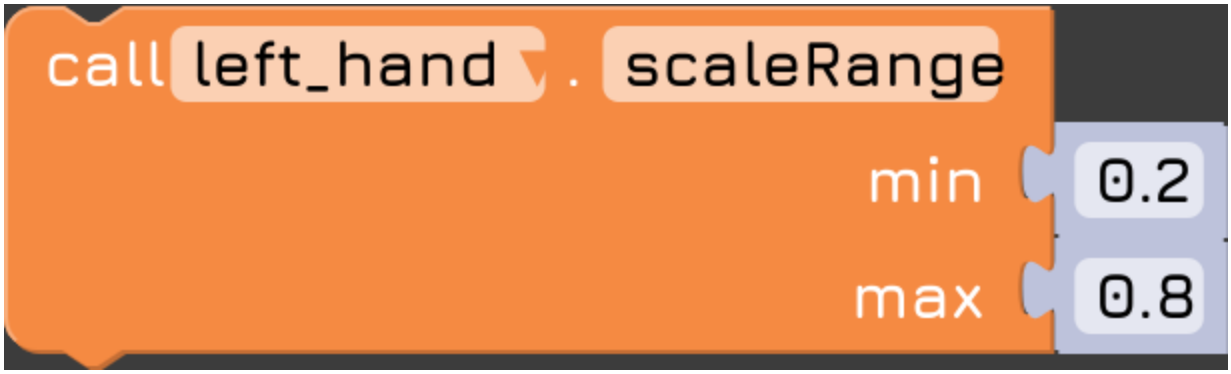
Position

Provides the current position of a servo.

Key points:

- Return Value: A floating point value in ticks.
- Usage: Crucial for precise servo control, closed-loop control, and position-based actions.

scaleRange

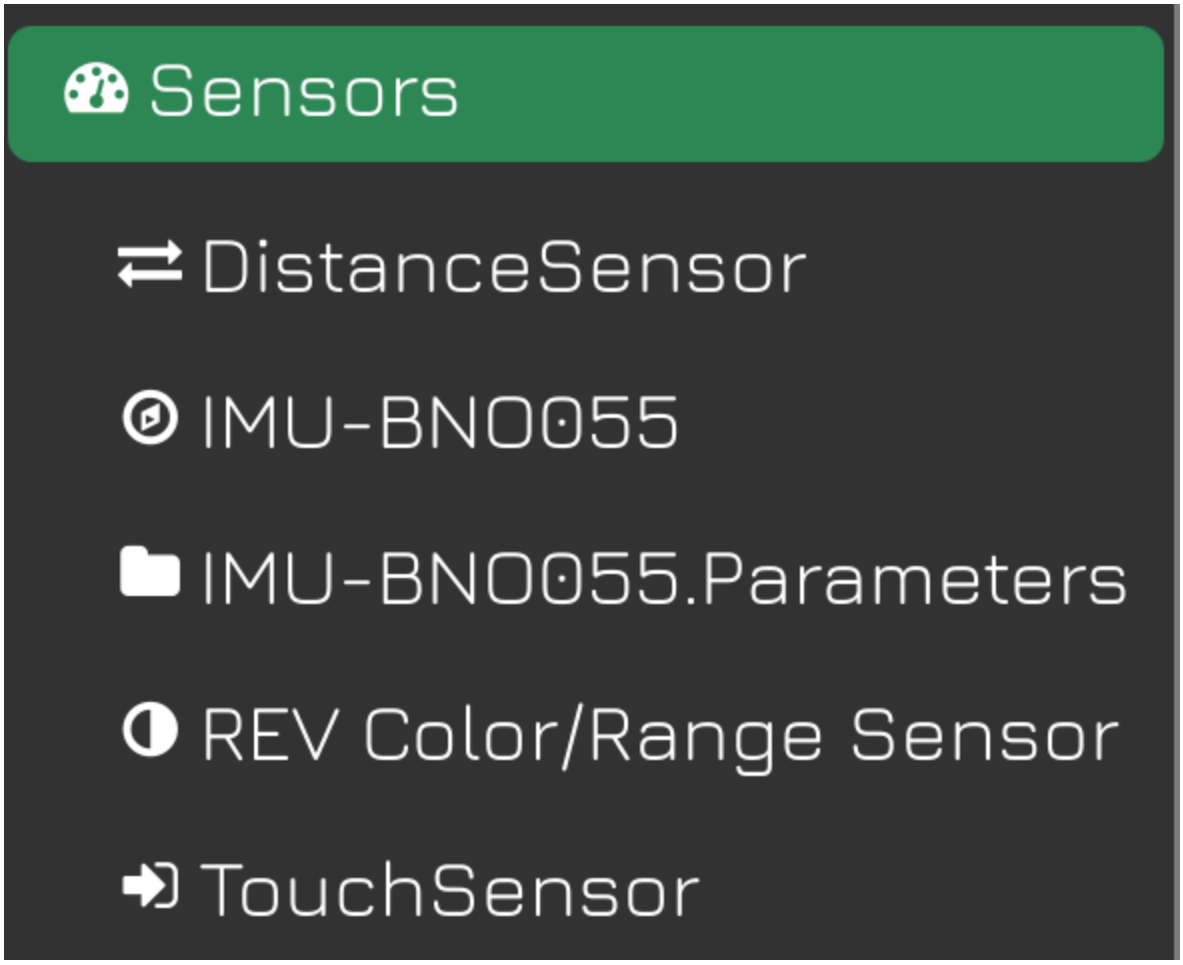


Defines a specific range of motion for a servo within its overall potential range.
Key points:

- Restricting movement: Prevents the servo from moving beyond a desired range, protecting mechanical components.
- Simplifying control: Maps desired values to a scaled range, making calculations easier.
- Optimizing performance: Focuses servo control on a specific area of its movement for better precision.
- Usage: Takes two parameters: a minimum and maximum value as a fraction of the full range (0.0 to 1.0). Subsequent set position calls will be scaled to fit within the specified range.

Sensors

Sensors are the vital components that enable robots to perceive and interact with their environment. These electronic devices provide data about the robot's surroundings, such as distance, color, light, touch, and more. By processing this information, robots can make intelligent decisions, navigate autonomously, and complete complex tasks.



DistanceSensor

An electronic device that measures the distance between itself and an object.

getDistance

Provides the distance measured by a distance sensor.

Key points:

- DistanceUnit: You can specify the desired unit for the distance measurement (CM, INCH, METER, MM).

- Return Value: The getDistance method returns a double value representing the measured distance.
- Sensor Type: The specific distance sensor used will influence the measurement range and accuracy. The simulation has consistent distance sensors.

IMU

An Inertial Measurement Unit (IMU) is a device that measures and reports a body's specific force, angular rate, and sometimes orientation. It typically consists of accelerometers and gyroscopes, and sometimes magnetometers. IMUs are essential for navigation, stabilization, and control in various applications like drones, robots, and vehicles.

To interact with IMU, you must use an I2C address. These addresses are 8-bit values ranging from 16 to 254 in decimal (10 to FE in hexadecimal) and must be even. By default, a Modern Robotics color sensor has an I2C address of 3C hexadecimal. This address is assumed by an op mode written with Blocks Programming unless otherwise specified.

If you change the address of a color sensor, you must inform the Blocks Programming environment of the new address. You can do this using the "Set i2cAddress7Bit" or "Set i2cAddres8Bit" block. If you use blocks that refer to "7Bit," only the high 7 of the 8 bits are used, and the numbers range from 8 to 7F in hexadecimal (8 to 127 in decimal). For example, if you have an 8-bit address of 22 hexadecimal (34 decimal), its 7-bit address would be half of these values: 11 hexadecimal (17 decimal).

You can use the blocks in this section to measure the acceleration, gravity, angular velocity, etc. If a block returns an object (e.g. Acceleration), you can use blocks under Utilities to process those projects.

Acceleration

Returns an Acceleration object representing the last observed acceleration of the sensor. Note that this does not communicate with the sensor, but rather returns the most recent value reported to the acceleration integration algorithm.

Key points:

- Units: can be set to m/s^2 or milligals (mGal).
- Values: returns an Acceleration object containing the x axis acceleration, y axis acceleration, z axis acceleration, and units in that order.
- Usage: to help autonomously guide the robot.

Gravity

Returns an Acceleration object representing the direction of the force of gravity relative to the sensor.

Key points:

- Units: can be set to g.
- Values: returns a Gravity object containing the force of gravity.

- Usage: to help autonomously guide the robot.

OverallAcceleration

Returns an Acceleration object representing the overall acceleration detected by the sensor. This is composed of a component due to the movement of the sensor and a component due to the force of gravity.

AngularOrientation

Returns an Orientation object representing the absolute orientation of the sensor as a set of three angles.

AngularOrientationAxes

Returns a List of the axes on which the sensor measures angular velocity. Some sensors measure angular velocity on all three axes (X, Y, & Z) while others measure on only a subset, typically the Z axis. This block allows you to determine what information is usefully returned through the get AngularVelocity block.

AngularVelocity

Returns an AngularVelocity object representing the rate of change of the absolute orientation of the sensor.

AngularVelocityAxes

Returns a List of the axes on which the sensor measures angular velocity. Some sensors measure angular velocity on all three axes (X, Y, & Z) while others measure on only a subset, typically the Z axis. This block allows you to determine what information is usefully returned through the get AngularVelocity block.

CalibrationStatus

Returns a text string giving the calibration status of the sensor.

Key points:

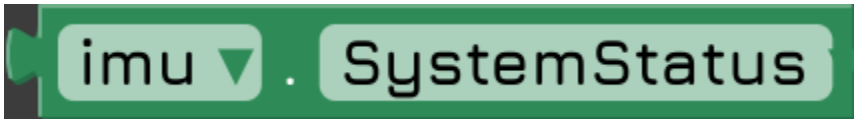
- The returned string is in the format, “IMU Calibration Status : sx gx ax mx”, where s stands for system, g for gyro, a for accelerometer and m for magnetometer. The x values are 0, 1, 2 or 3, where 0 means uncalibrated, 3 means fully calibrated and 1 and 2 mean partially calibrated. For example, “IMU Calibration Status : s0 g3 a0 m1” the system is not calibrated, the gyro is fully calibrated, the accelerometer is not calibrated

and the magnetometer is partially calibrated.

SystemError

If SystemStatus is “SYSTEM_ERROR”, returns more detail about the error. Otherwise, the value is undefined.

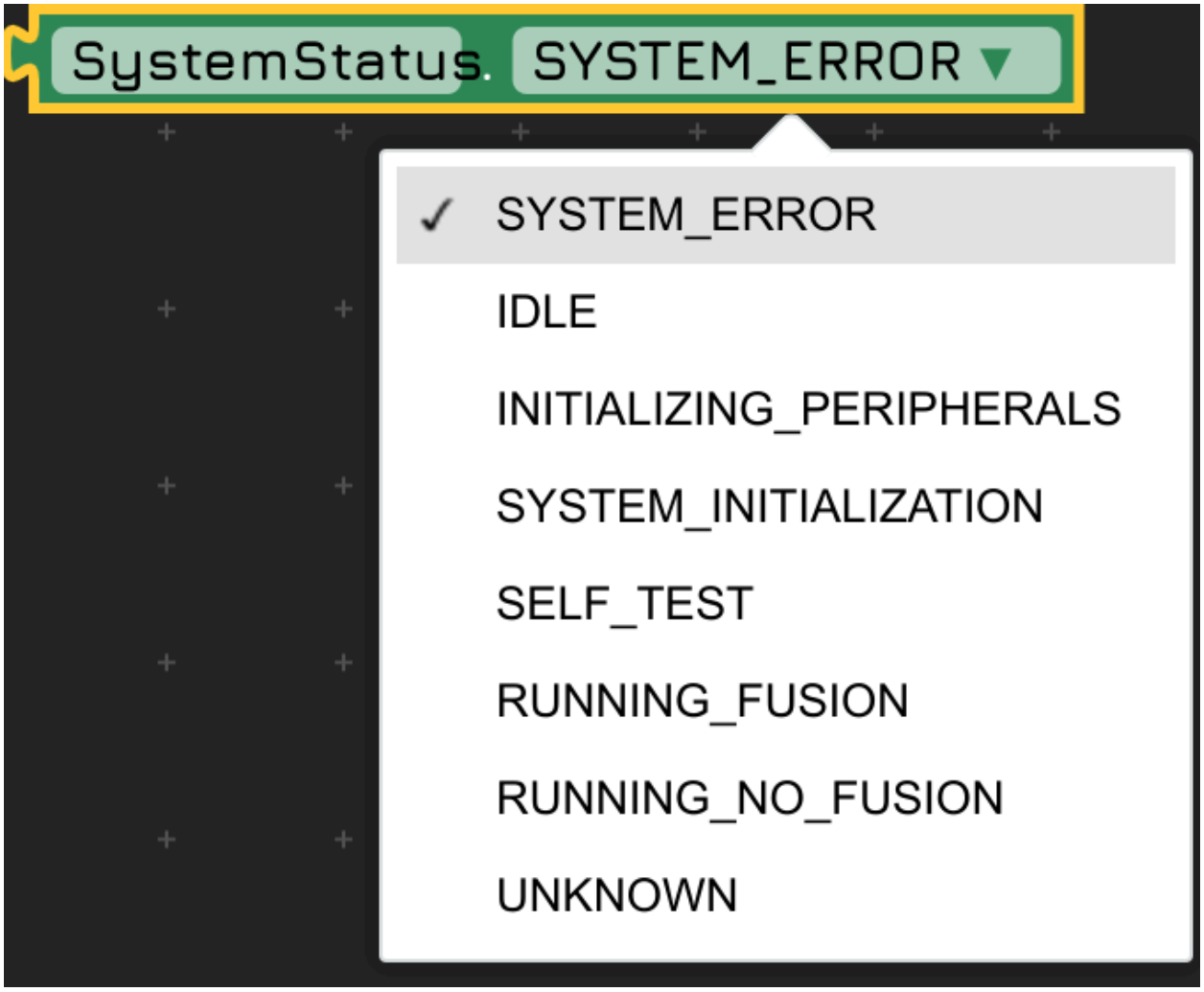
SystemStatus



Returns a text value representing the current status of the system.

SystemStatus.SYSTEM_ERROR

This indicates that there is a system error. You can use IMU.SystemError block to get the detailed error. Since we are running in a simulated environment, we should not expect SystemError.



set IMU.I2cAddress7Bit to

Sets the 7bit I2C address to the specified number.

Key points:

- Purpose. Change the default 7 bit I2C address to differentiate sensors.
- The address must be an even number from 8 to 127 included.

I2cAddress8Bit

Return the 8 bi I2C address of the IMU.

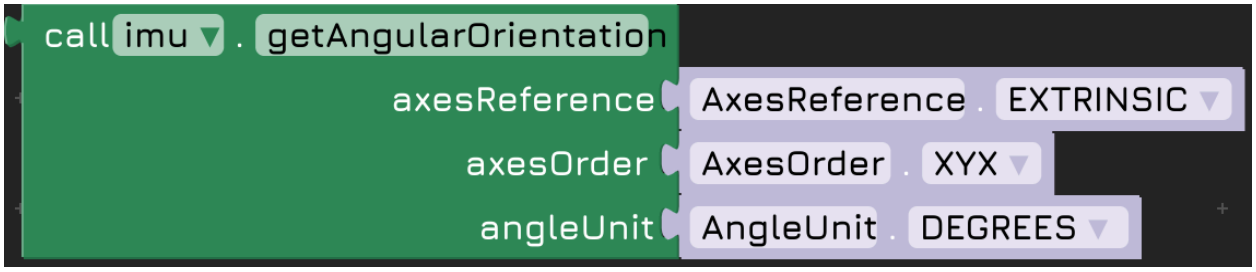
Position

Returns a Position object representing the current position of the sensor as calculated by doubly integrating the observed sensor accelerations.

Velocity

Returns a Velocity object representing the current velocity of the sensor as calculated by integrating the observed sensor accelerations.

getAngularOrientation

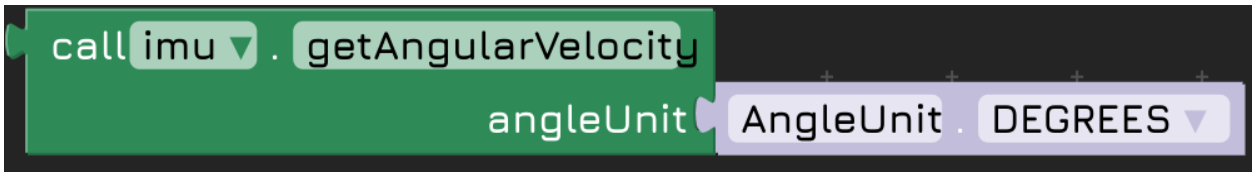


Returns an Orientation object representing the absolute orientation of the sensor as a set of three angles. Axes on which absolute orientation is not measured are reported as zero.

Key points:

- axesReference: you can choose EXTRINSIC or INTRINSIC.
- axesOrder: You can choose any of the 12 combinations.
- angleUnit: You can choose DEGREES or RADIANS.

getAngularVelocity



Returns an AngularVelocity object representing the angular rotation rate across all the axes measured by the sensor. Axes on which angular velocity is not measured are reported as zero.

Key points:

- angleUnit: You can choose DEGREES or RADIANS.

initialize

Initializes the IMU based on an IMU parameters object.

isAccelerometerCalibrated

Returns true if the Accelerometer has completed calibration; false otherwise.

isGyroCalibrated

Returns true if the gyroscope has completed calibration; false otherwise.

isMagnetomemterCalibrated

Returns true if the magnetometer has completed calibration; false otherwise.

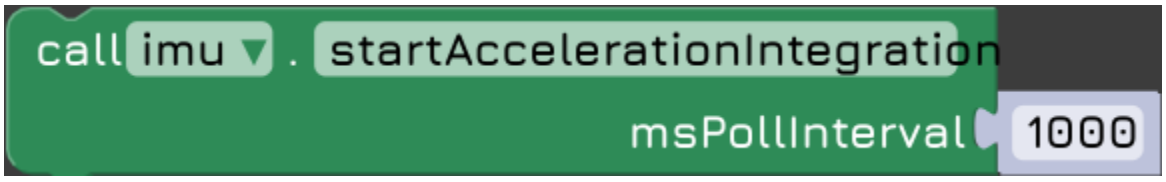
isSystemCalibrated

Returns true if the system is fully calibrated. The system is fully calibrated if the gyro, accelerometer, and magnetometer are fully calibrated.

saveCalibrationData

Saves the current calibration information in the file specified by the given text. Such files usually end with the “json” extension.

startAccelerationIntegration

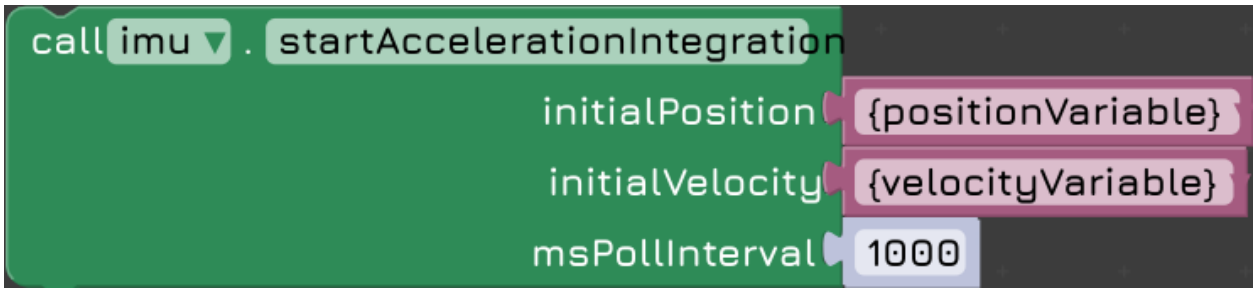


Starts (or re-start) polling, at the given interval, the current linear acceleration of the sensor and integrates it to provide velocity and position information.

Key points:

- The poll interval must be set in milliseconds.

startAccelerationIntegration (initialPosition and initialVelocity)



Starts (or re-start) polling, at the given interval, the current linear acceleration of the sensor and integrates it to provide velocity and position information.

stopAccelerationIntegration

Stops the integration thread if it is currently running.

IMU-BNO055.Parameters

Updates the IMU parameters object by setting the 7 bit I2C address.

Key points:

- Purpose. Change the default 7 bit I2C address to differentiate sensors.

- The address must be an even number from 8 to 127 included.

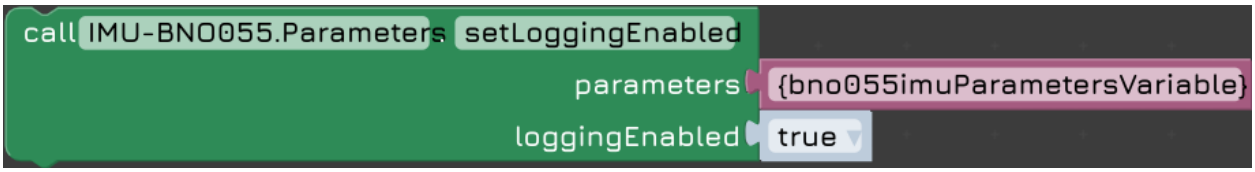
setI2cAddess8Bit

Updates the IMU parameters object by setting the 8 bit I2C address.

Key points:

- Purpose. Change the default 8 bit I2C address to differentiate sensors.
- The address must be an even number from 16 to 254 included.

setLoggingEnabled



Updates the IMU parameters object by setting the logging enabled state to true or false.

Key points:

- If the state is set to true the IMU sensor activities will be added to the log.
- If the state is set to false, there will be no log of the IMU sensor activities.

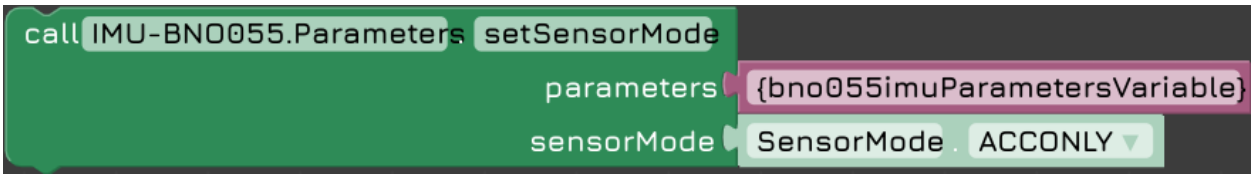
setLoggingTag

Updates the IMU parameters object by setting the logging tag to a specific text. This tag will appear at the beginning of each IMU log entry.

Key points:

- This tag makes it easier to visually scan and interpret log files, especially when multiple components or sensors are generating log entries.

setSensorMode



Updates the IMU parameters object by setting the senor's mode to one of 11 sensor modes:

Individual Sensor Modes

- ACCONLY: Only the accelerometer is active. Use this when you just need linear acceleration data (e.g., detecting if the robot is being bumped or tilted).
- MAGONLY: Only the magnetometer is active. Acts like a compass, providing information about the Earth's magnetic field.

- GYROONLY: Only the gyroscope is active. Measures how fast the sensor is rotating around its axes.

Combined Sensor Modes

- ACCMAG: Accelerometer and magnetometer are both active. Useful when you need both linear acceleration and compass-like heading information.
- ACCGYRO: Accelerometer and gyroscope are both active. Common for basic orientation tracking and motion control.
- MAGGYRO: Magnetometer and gyroscope are both active. Can be used for orientation tracking with lower power consumption than using the gyroscope alone.

Fusion Modes (Combining all Sensors)

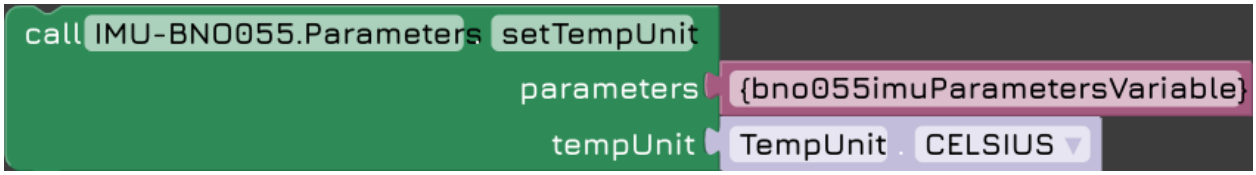
- AMG (ACC-MAG-GYRO): All three sensors are active. Provides the most complete information about the sensor's movement and orientation.
- IMU: Fuses accelerometer and gyroscope data to calculate orientation. Fast and suitable for many applications.
- COMPASS: Uses all sensors to calculate geographic direction (like a compass). Requires calibration due to magnetic field variations.
- M4G: Similar to IMU mode but uses the magnetometer instead of the gyroscope to detect rotation. Lower power consumption but sensitive to magnetic fields.
- NDOF: Advanced fusion mode that provides highly accurate orientation data, even in the presence of magnetic interference.
- NDOF_FMC_OFF: Same as NDOF but without fast magnetometer calibration. Slightly lower power consumption but may require longer calibration times.

Key points:

How to choose the right mode?

- Consider your application's needs: Do you need just acceleration, just orientation, or a combination?
- Power consumption: Some modes consume less power than others.
- Accuracy: Fusion modes generally provide the most accurate orientation data.
- Calibration: Some modes, like COMPASS, require calibration to function properly.

setTempUnit



Updates the IMU parameters object by setting the temperature unit to either CELSIUS, FARENHEIT, or KELVIN.

Key points:

- CELSIUS: The most widely used temperature scale globally, especially for everyday measurements, weather reports, and scientific applications.
- FARENHEIT: Primarily used in the United States for everyday temperature measurements.
- KELVIN: Used mainly in scientific fields, especially those involving very low or very high temperatures (like in thermodynamics or astronomy).

AccelUnit

Returns the acceleration unit of the IMU parameters object.

AccelerationIntegrationAlgorithm

Returns the acceleration integration algorithm of the IMU parameters object.

AngleUnit

Returns the angle unit of the IMU parameters object.

CalibrationDataFile

Returns the calibration data file of the IMU parameters object.

I2cAddess7Bit

Returns the 7 bit I2C address of the IMU parameters object.

I2cAddess8Bit

Returns the 8 bit I2C address of the IMU parameters object.

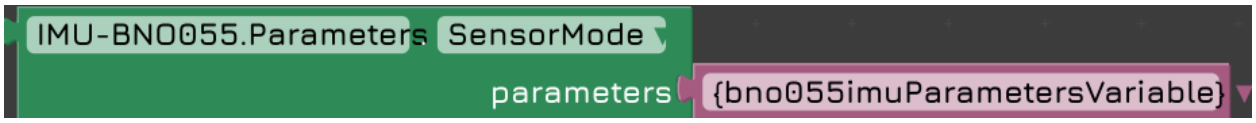
LoggingEnabled

Returns whether logging is enabled or not.

LoggingTag

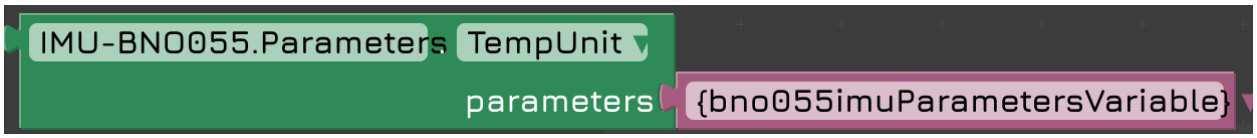
Returns the logging tag of the IMU parameters object.

SensorMode



Returns the sensor mode of the IMU parameters object.

TempUnit



Returns the temperature unit of the IMU parameters object.

REV Color/Range Sensor

Key points:

- Return value: an integer indicating the 7 bit I2C address. The value ranges from 8 to 127.

I2cAddess8Bit

Returns the 8 bit I2C address of the sensor.

Key points:

- Return value: an integer indicating the 8 bit I2C address. The value ranges from 16 to 254.

LightDetected

Gets the amount of light detected from the named range sensor.

Key points:

- Purpose: To get the amount of light detected.
- Return value: A float value ranging from 0.0 to 1.0.

RawLightDetected

Gets a value of the sensor, which is proportional to the amount of raw light detected.

Key points:

- Purpose: To get the amount of light detected in proportional to raw amounts.

- Return value: A positive float value.

RawLightDetectedMax

Gets the maximum raw light value that can be returned by the sensor.

Key points:

- Purpose: To get the maximum possible raw light detected.
- Return value: A positive float value.

Red

Gets the current red component from the sensor.

Key points:

- Purpose: To get the red value of the sensor.
- Return value: an integer value ranging from 0 to 255.

set ColorSensor Gain to

Sets the gain for the sensor. "gain" refers to the sensitivity or amplification factor of the sensor. It represents how much the sensor's output signal is increased relative to the input it's measuring.

Key points:

- Purpose: To set the gain value for the sensor to magnify the sensor's output signal.
- Parameter: Gain can be any positive float number.

set ColorSensor 12cAddess7Bit to

Sets the 7 bit I2C address to the sensor.

Key points:

- Purpose. Change the default 7 bit I2C address to differentiate sensors. The address must be an even number from 8 to 127 included.

set ColorSensor 12cAddess8Bit to

Sets the 8 bit I2C address to the sensor.

Key points:

- Purpose. Change the default 8 bit I2C address to differentiate sensors. The address must be an even number from 16 to 254 included.

getDistance

Gets the current distance from the named range sensor using the requested distance unit. Valid values are DistanceUnit.METER, DistanceUnit.CM, DistanceUnit.MM, DistanceUnit.INCH.

Key points:

- DistanceUnit: You can specify the desired unit for the distance measurement (CM, INCH, METER, MM).
- Return Value: The getDistance method returns a double value representing the measured distance.

getNormalizedColors

Gets the normalized alpha-RGB color object detected by the sensor.

Key points:

- Return Value: A normalized color object with values ranging from 0 to 1.

TouchSensor

This sensor is connected as a digital device and not connected via an I2C bus so it does not have an I2C address.

isPressed

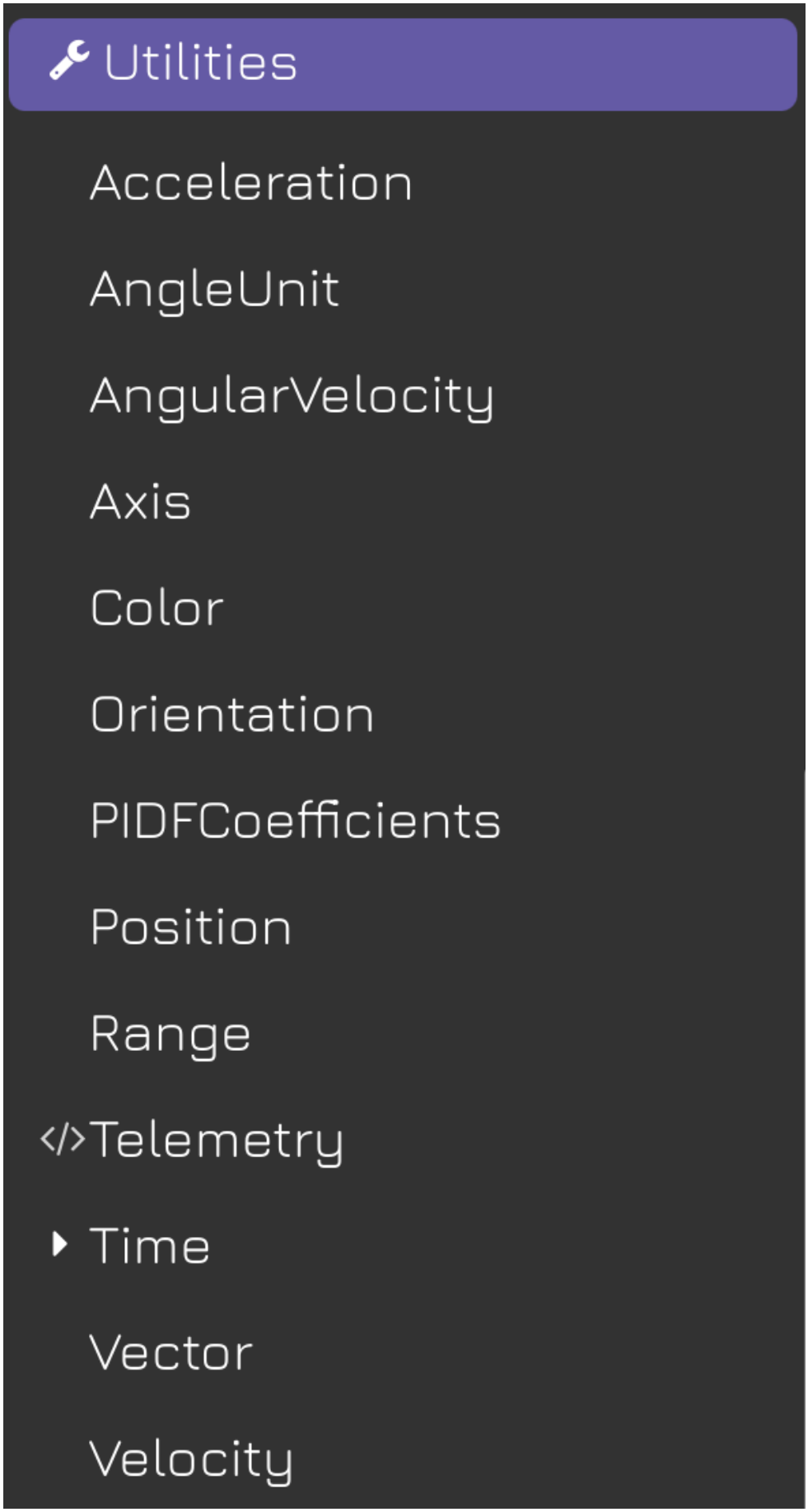
Return true if the named touch sensor has been pressed, false otherwise.

Key points:

- Purpose: To determine whether the sensor is pressed or not.
- Return value: True if the sensor is pressed.
- Usage: Typically used within a loop to check whether the robot hits an object or not.

Utilities

The "Utilities" section in your block programming environment is like a toolbox filled with handy gadgets and gizmos. These special blocks aren't about making your robot move or react directly, but they supercharge your code with extra capabilities.

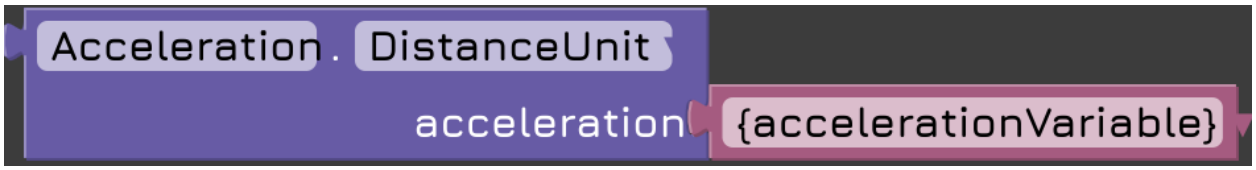

Utilities
Acceleration
AngleUnit
AngularVelocity
Axis
Color
Orientation
PIDFCoefficients
Position
Range
Telemetry
Time
Vector
Velocity

Acceleration

Acceleration

Acceleration refers to the rate at which the robot's velocity changes over time. It is a crucial aspect of controlling the robot's movement and ensuring smooth, precise actions. The blocks in this section allow you to create acceleration objects, manipulate these objects, and obtain information from these objects.

DistanceUnit



Returns the distance unit attribute of the acceleration object. Valid values are CM, INCH, METER and MM.

XAccel

Returns the acceleration along the X-axis of the acceleration object.

YAccel

Returns the acceleration along the Y-axis of the acceleration object.

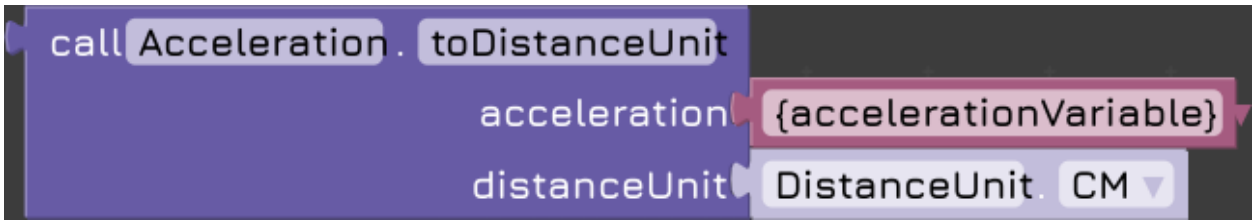
ZAccel

Returns the acceleration along the Z-axis of the acceleration object.

AcquisitionTime

Returns the acquisition time of the acceleration object.

toDistanceUnit



Returns a new Acceleration object based on another acceleration object after converting its x, y and z acceleration values to the specified distant unit.

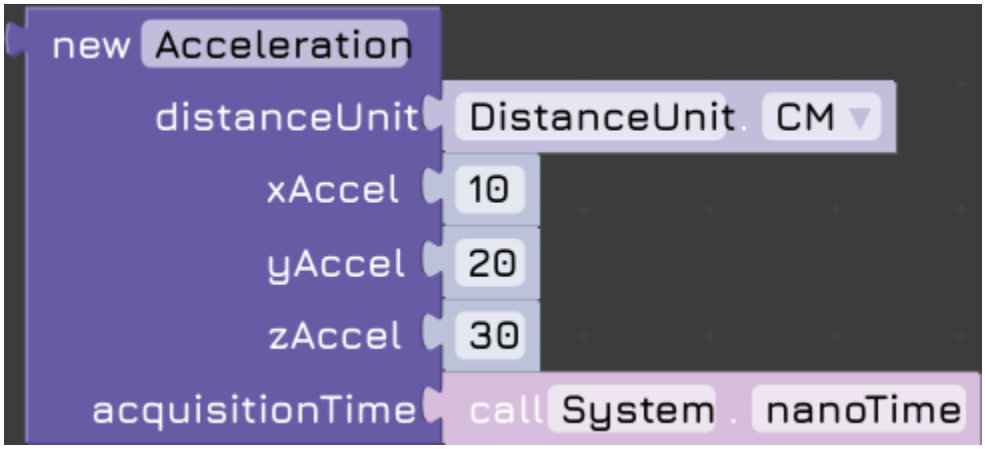
toText

Returns text string that is a formatted version of the acceleration object.

new Acceleration

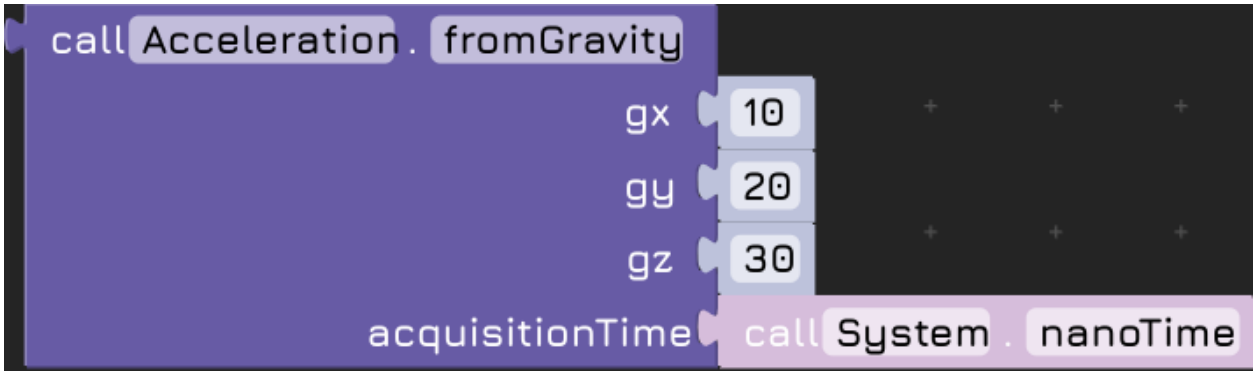
Returns a new Acceleration object.

new Acceleration (expanded)



Returns a new Acceleration object using the specified distance unit, x, y and z acceleration values in those units and the current system time as the acquisition time.

fromGravity



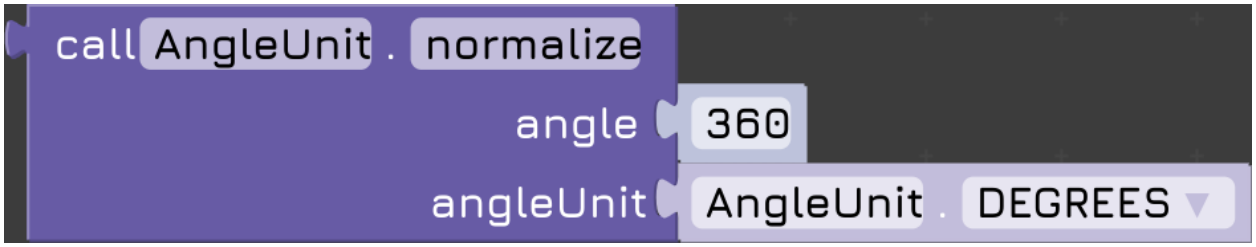
Returns a new Acceleration object with x, y, and z components expressed as multiples of the gravitational acceleration constant (g) and the current system time recorded as the acquisition time.

AngleUnit

AngleUnit refers to the unit of measurement used to represent angles. It defines how angular values are interpreted and used within the FTC programming environment. Common angle units are

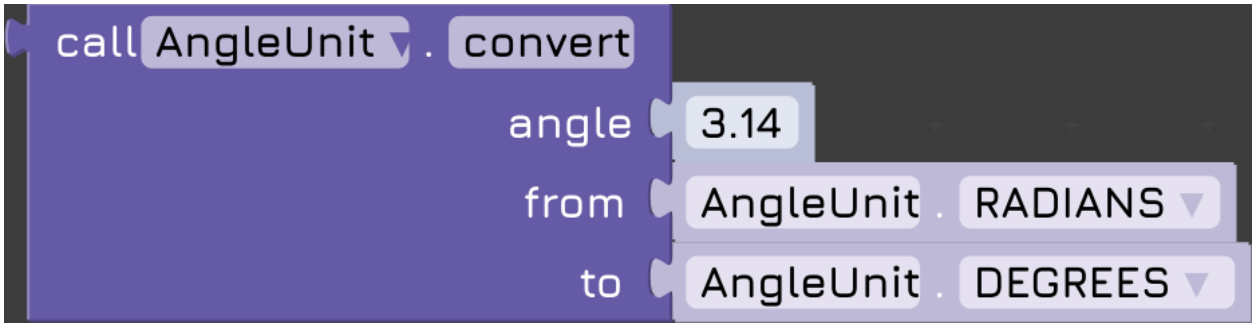
- DEGREES: This is the most common and intuitive unit for representing angles. A full circle is 360 degrees, a right angle is 90 degrees, and so on.
- RADIANS: This is the standard unit of angular measurement in mathematics and physics. A full circle is 2π radians, a right angle is π/2 radians.

call AngleUnit.normalize



Normalizes the given angle to the range of [-180, 180) degrees, or [-π, π) radians.

call AngleUnit.convert



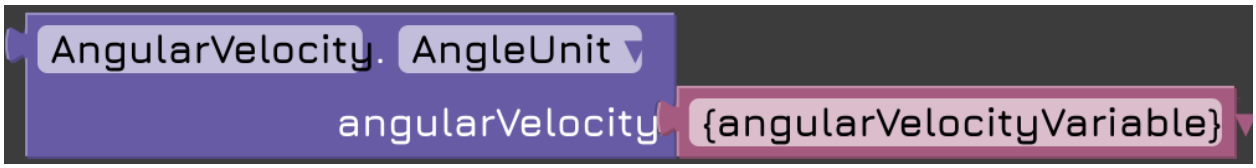
Converts the given angle to the range of [-180, 180) degrees, or [-π, π) radians, depending on the “to” unit.

AngularVelocity

AngularVelocity refers to the rate at which an object or robot rotates around an axis. It's a crucial concept for understanding and controlling the rotational motion of various components in your robot, such as wheels, arms, or sensors. The

blocks in this section allow you to create angular velocity objects, manipulate these objects, and obtain information from these objects.

AngularVelocity.AngleUnit



Returns the angle unit (DEGREES or RADIANS) of the given angular velocity object.

AngularVelocity.XRotationRate

Returns the rotational rate along the X-axis of the given angular velocity object.

AngularVelocity.YRotationRate

Returns the rotational rate along the Y-axis of the given angular velocity object.

AngularVelocity.ZRotationRate

Returns the rotational rate along the 2-axis of the given angular velocity object.

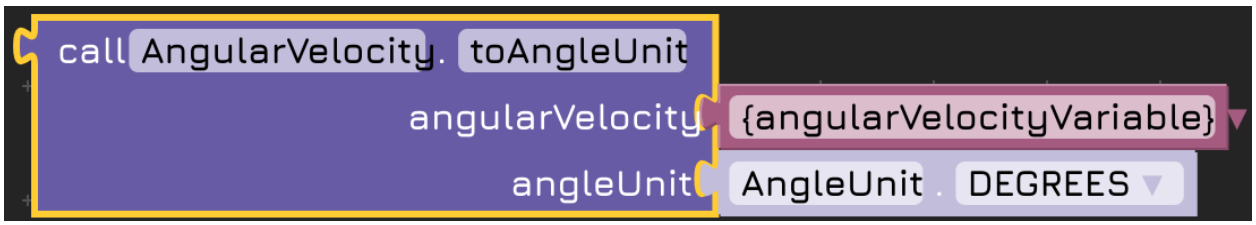
call AngularVelocity.getRotationRate

Returns the rotational rate of the given angular velocity object along the given axis (X, Y, or Z).

AngularVelocity.AcquisitionTime

Returns the acquisition time of the angular velocity object.

call AngularVelocity.toAngleUnit

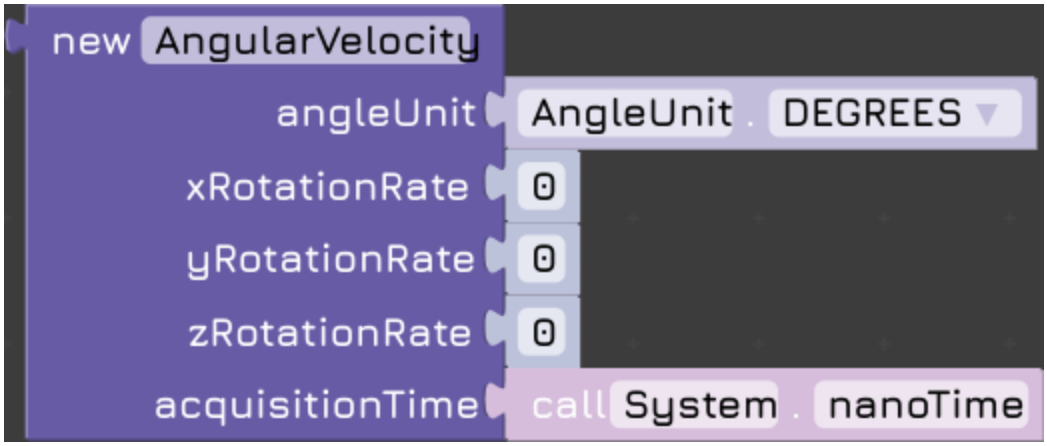


Returns a new angular velocity object based on another object after converting its x, y and z rotation rates to the specified units.

new AngularVelocity

Returns a new angular velocity object.

new AngularVelocity (expanded)



Returns a new angular velocity object using the provided unit and x, y and z rotation rates. The new angular velocity object sets the current system time as the acquisition time.

Axis

Axis refers to a reference line or direction used to describe the movement and orientation of the robot and its components. It can be one of X, Y, or Z axes. The blocks in this section allow you to obtain information from axis objects.

X

Returns the value on the X axis.

Y

Returns the value on the Y axis.

Z

Returns the value on the Z axis.

Color

Returns the blue component of the given color object. The returned value ranges from 0 to 255.

Color.Alpha

Returns the alpha component of the given color object. The returned value ranges from 0 to 255.

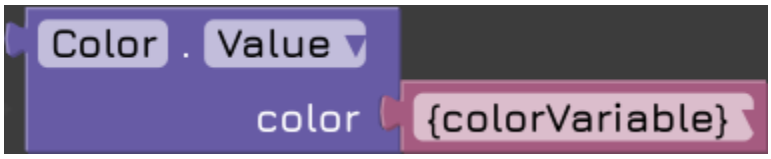
Color.Hue

Returns the hue component of the given color object. The returned value ranges from 0 to 255.

Color.Saturation

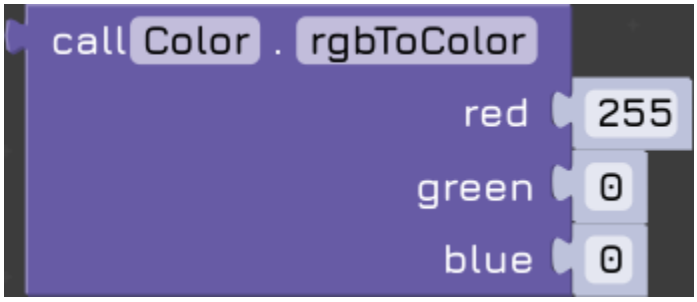
Returns the saturation component of the given color object. The returned value ranges from 0 to 255.

Color.Value



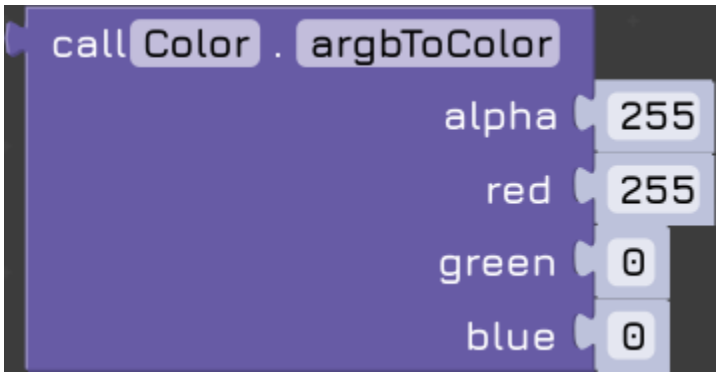
Returns the value of the given color object. The returned value is a 32-bit integer value that represents a color using the ARGB (Alpha, Red, Green, Blue) color model. Each of the four components (A, R, G, B) is allocated 8 bits, resulting in a total of 32 bits.

call Color.rgbToColor



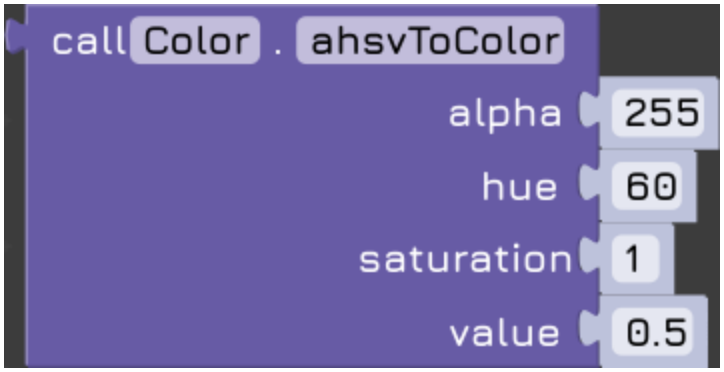
Returns a new color object using the given values for red, green, and blue.

call Color.argbToColor



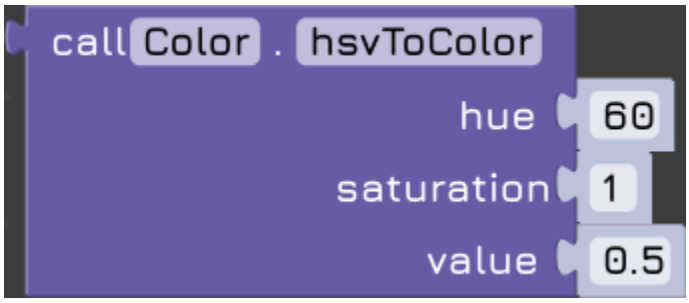
Returns a new color object using the given values of alpha, red, green, and blue.

call Color.hsvToColor



Returns a new color object using the given values of alpha, hue, saturation, and value.

call Color.hsvToColor



Returns a new color object using the given values of hue, saturation, and value

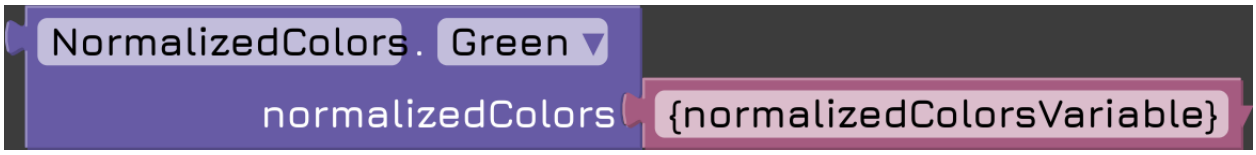
call Color.textToColor

Returns a new color object using the given text value assuming it has the format of “#RRGGBB” or “#AARRGGBB”.

NormalizedColors.Red

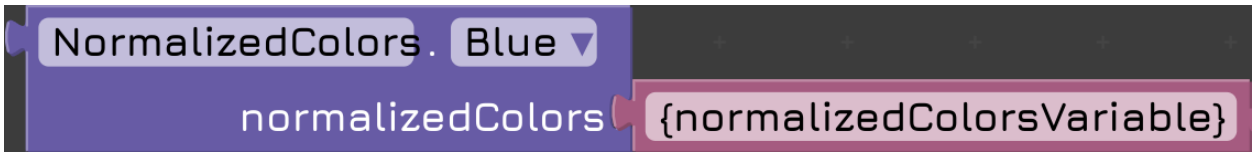
Returns the red value of the given normalized Color object. The value ranges from 0 to 1.

NormalizedColors.Green



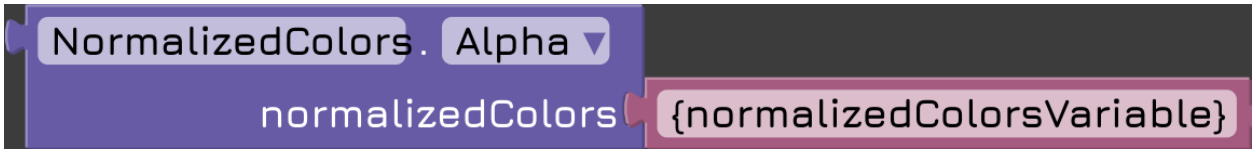
Returns the green value of the given normalized Color object. The value ranges from 0 to 1.

NormalizedColors.Blue



Returns the blue value of the given normalized Color object. The value ranges from 0 to 1.

NormalizedColors.Alpha



Returns the alpha value of the given normalized Color object. The value ranges from 0 to 1.

NormalizedColors.Color

Returns the Android color integer representation of the normalized color. The

returned value is a 32-bit integer value that represents a color using the ARGB (Alpha, Red, Green, Blue) color model. Each of the four components (A, R, G, B) is allocated 8 bits, resulting in a total of 32 bits.

Orientation

FTC robots operate in three-dimensional space. Orientation describes how the robot is rotated or tilted within this space. These blocks in this section allow you to create orientation objects, manipulate these objects, and obtain information from these objects. For background on orientation calculations see Euler angles in Wikipedia.

call Orientation.AxesReference

Returns the axes reference of the given orientation object.

call Orientation.AxesOrder

Returns the axes order of the given orientation object. For example, “XYX”.

call Orientation.AngleUnit

Returns the angle unit value of the given orientation object.

call Orientation.FirstAngle

Returns the first angle of the given orientation object.

call Orientation.SecondAngle

Returns the second angle of the given orientation object.

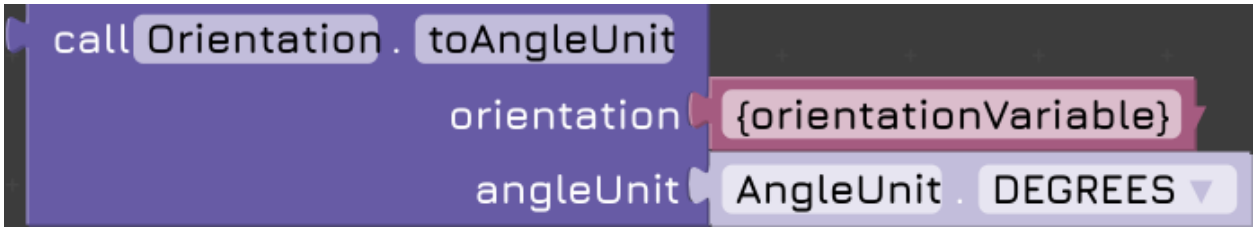
call Orientation.ThirdAngle

Returns the third angle of the given orientation object.

call Orientation.AcquisitionTime

Returns the acquisition time of the given orientation object

call Orientation.toAngleUnit



Returns a new orientation object based on the given orientation object, converts the angle to a new unit, either “DEGREES” or “RADIANS.”

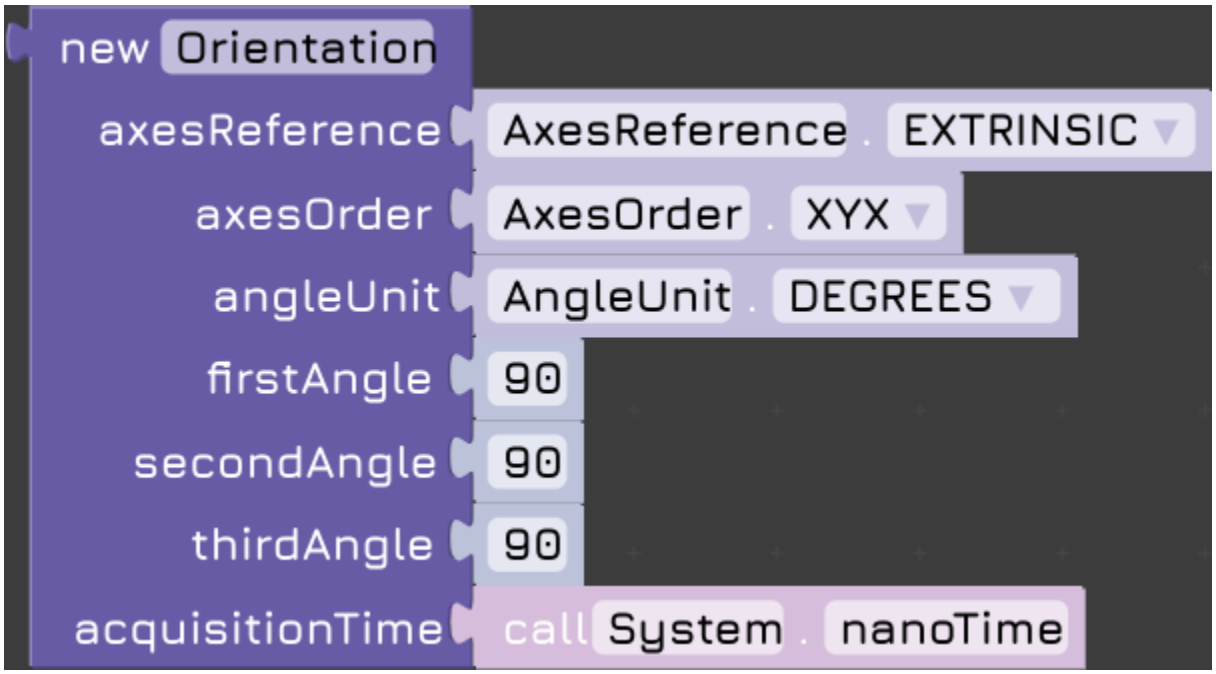
call Orientation.toText

Returns text representation of the given orientation object.

new Orientation

Return a new orientation object.

new Orientation (Expanded)



Returns a new orientation object based on the information provided. The new orientation object uses the current system time as the acquisition time.

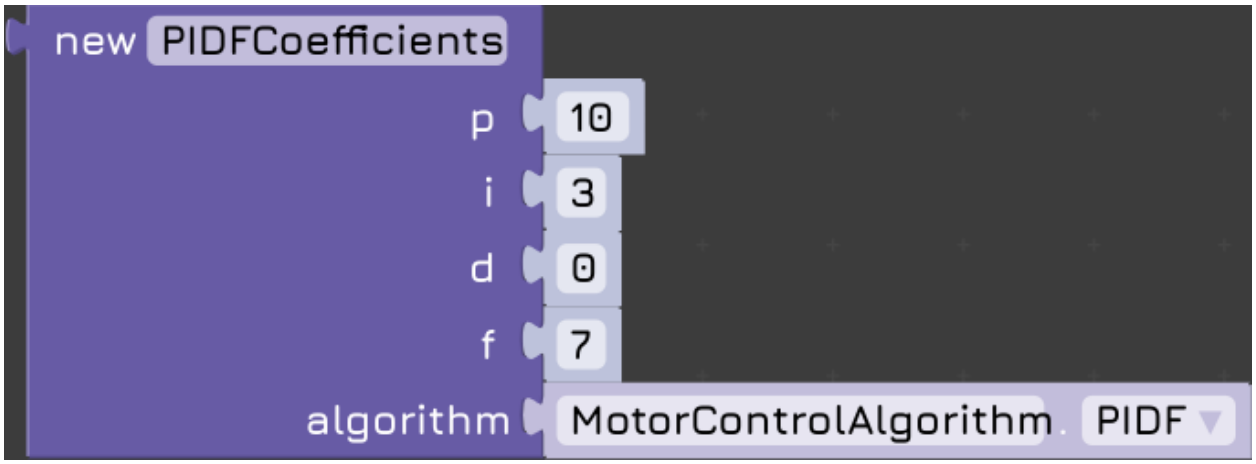
PIDFCoefficients

PIDFCoefficients represents a set of tuning parameters used in PIDF control algorithms, specifically for controlling motors. PIDF stands for Proportional, Integral, Derivative, and Feedforward. The blocks in this section allow you to create PIDFCoefficients objects, manipulate these objects, and obtain information from these objects.

new PIDFCoefficients

Returns a new PIDFCoefficients object with default values: p=0, i=0, d=0, f=0. The default algorithm is PIDF.

new PIDFCoefficients (expanded with algorithm)

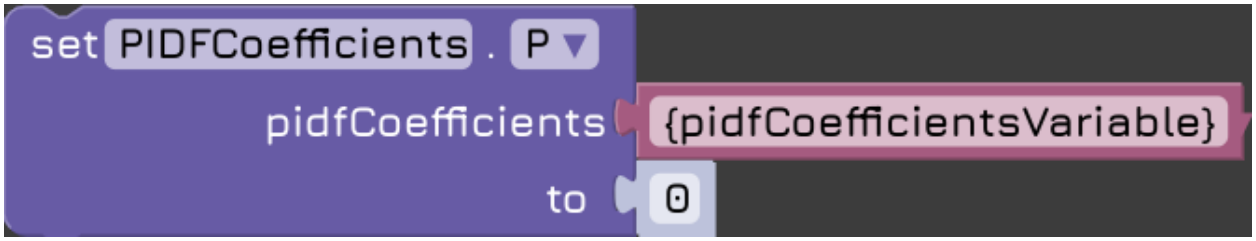


Returns a new PIDFCoefficients object with the specified PIDF values and algorithm.

new PIDFCoefficients (expanded)

Returns a copy of the given PIDFCoefficients object.

Set PIDFCoefficients.P

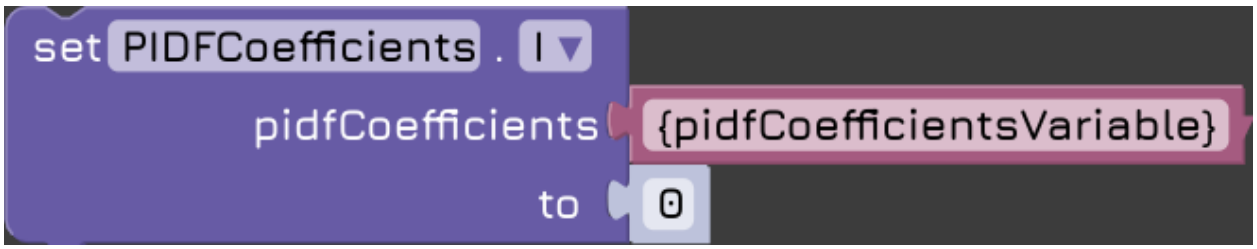


Sets the proportional term P of the given PIDFCoefficients object.

new PIDFCoefficients.P

Returns the proportional term P of the given PIDFCoefficients object.

Set PIDFCoefficients.I

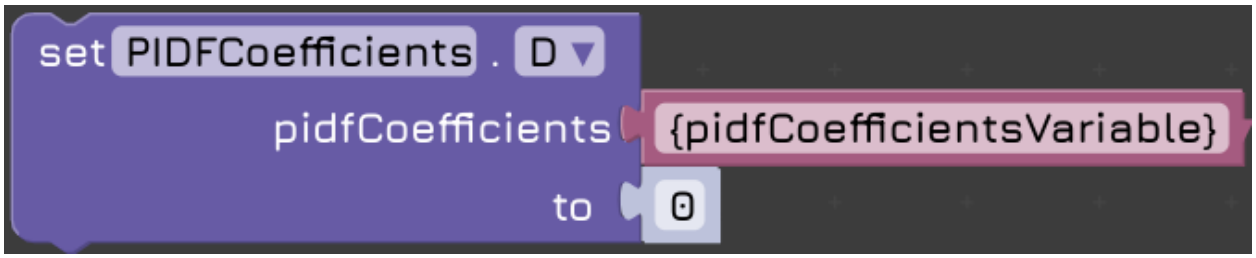


Sets the integral term I of the given PIDFCoefficients object.

new PIDFCoefficients.I

Returns the integral term I of the given PIDFCoefficients object.

Set PIDFCoefficients.D

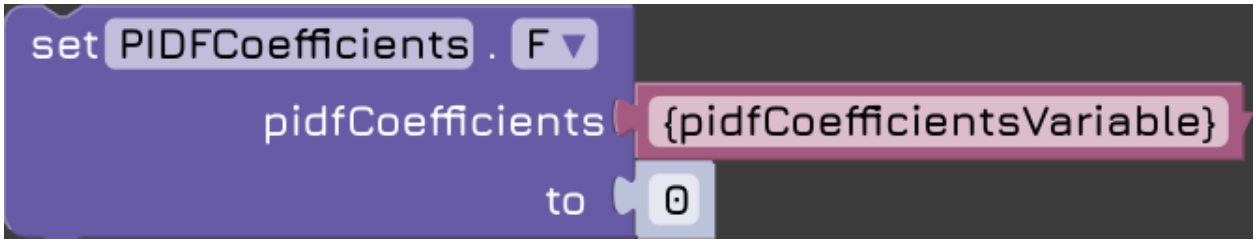


Sets the derivative term D of the given PIDFCoefficients object.

new PIDFCoefficients.D

Returns the derivative term D of the given PIDFCoefficients object.

Set PIDFCoefficients.F

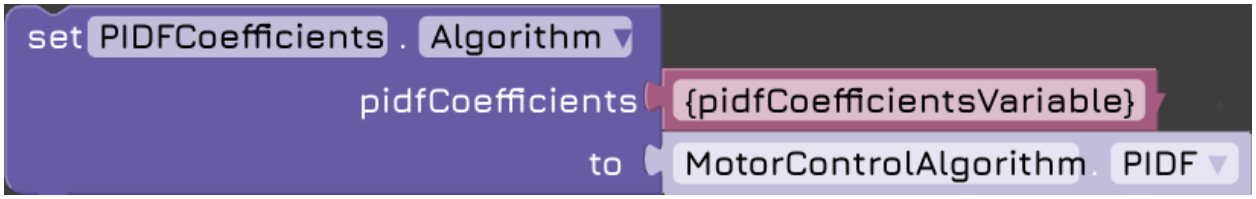


Sets the feedforward term F of the given PIDFCoefficients object.

new PIDFCoefficients.F

Returns the feedforward term F of the given PIDFCoefficients object.

set PIDFCoefficients.Algorithm



Sets the algorithm for the given PIDFCoefficients object. The system only supports two algorithms: “PIDF”, or “LegacyPID.”

PIDFCoefficients.Algorithm

Returns the algorithm of the given PIDFCoefficients object.

call PIDFCoefficients.toText

Returns a text presentation of the given PIDFCoefficients object.

Position

Position refers to the location or placement of the robot or its components within the playing field or its own coordinate system. It's a fundamental concept for understanding and controlling the robot's movements and interactions with its environment. The blocks in this section allow you to create position objects, manipulate these objects, and obtain information from these objects.

Position.DistanceUnit

Returns the distance unit used by the given position object.

Position.X

Returns X value of the position represented by a given position object.

Position.Y

Returns Y value of the position represented by a given position object.

Position.Z

Returns Z value of the position represented by a given position object.

Position.AcquisitionTime

Returns acquisition time of a given position object.

Position.toDistanceUnit

Returns a new position object based on a given position object and a new distance unit.

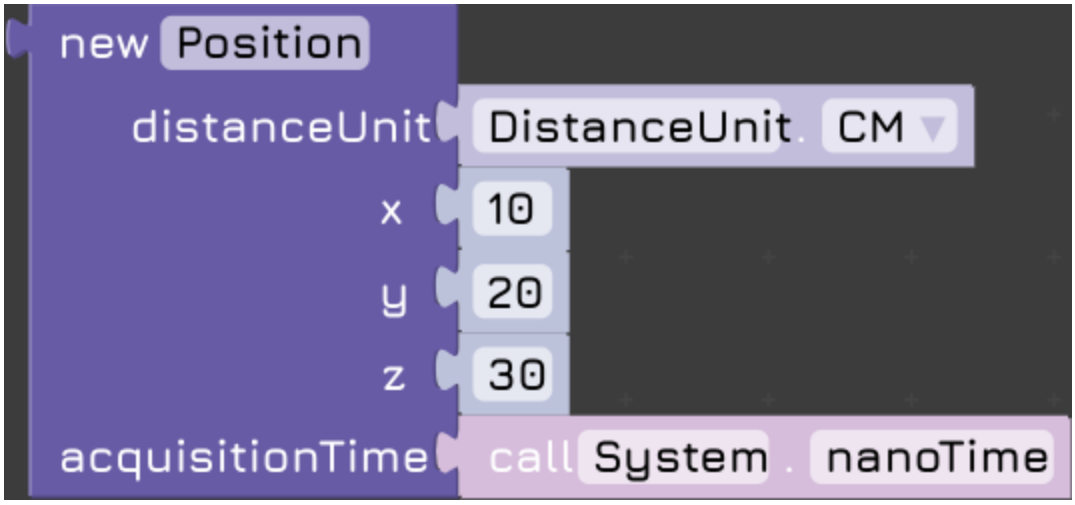
toText

Returns a text representation of a given position object.

new Position

Returns a new position object with all X, Y, Z values being 0.

new Position (expanded)

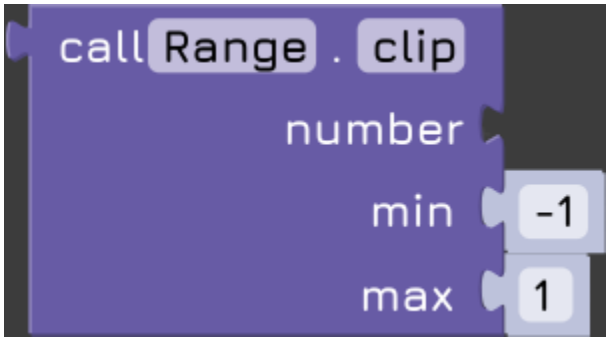


Returns a new position object using the given distance using x, y and z values. It uses the current system time as the acquisition time.

Range

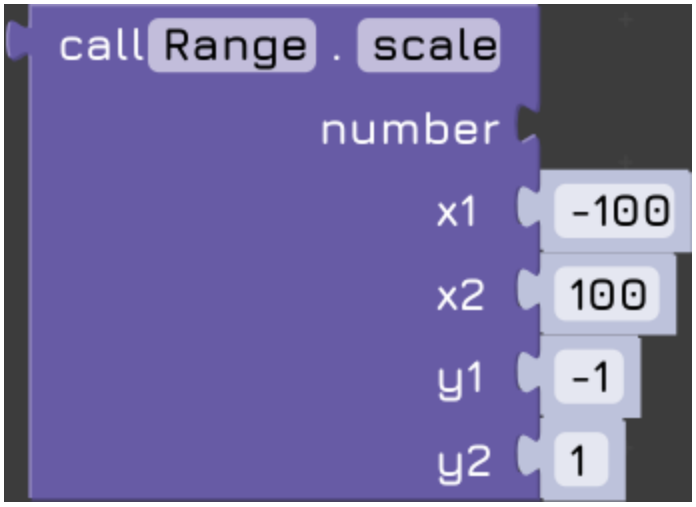
A range is defined by a minimum value and a maximum value. It is often used to limit the sensor or servo range. The blocks in this section allow you to adjust a number to a range.

clip



Clips a number if it is smaller than the min value or larger than the max value. For example, an original value of -5 becomes 0 after clip to a range of [0, 100].

scale

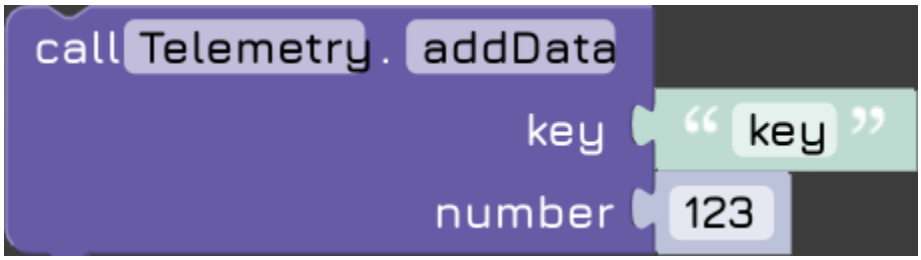


Scales a number in the range of [x1, x2] to the range of [y1, y2]. For example, an original value of 2 in range [0, 10] scales to 20 in the range of [0, 100].

Telemetry

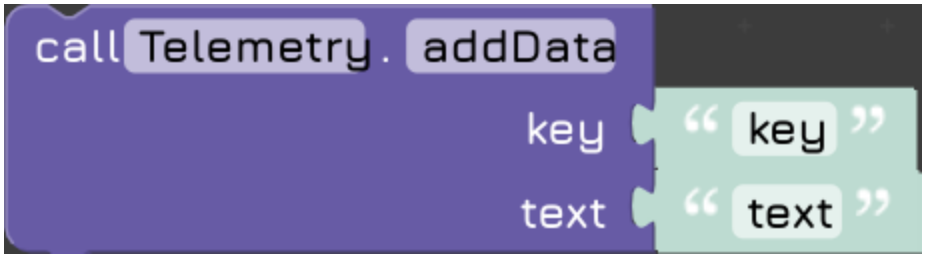
Telemetry allows you to monitor the robot's status, sensor readings, and other critical information in real-time during operation.It is useful for debugging, troubleshooting, performance monitoring, data logging and more. The blocks here allow you to show telemetry data.

addData (key and number)



Sends key and numeric value to Driver Station for display. The telemetry shows “key: 123” in the given example.

addData (key and text)



Sends key and text value to Driver Station for display. The telemetry shows “key: text” in the given example.

call Telemetry.update

Sends accumulated telemetry data and causes the driver station to display it. Before calling this function, the telemetry information is buffered and not visible.

Time

Time plays a critical role in various aspects of robot design, programming, and competition strategy. The blocks in this section allow you to get high precision system time and calculate the elapsed time.

nanoTime

Return the current system time in nanoseconds.

ElapsedTime

Elapsed time simply refers to the amount of time that has passed between a starting point and an ending point. It's the duration of an event or the time it takes for something to happen.

new ElapsedTime

Returns an ElapsedTime object presenting a timer with SECONDS resolution, initialized with the current system time.

new ElapsedTime (startTime)

Returns an ElapsedTime object presenting a timer with SECONDS resolution, initialized with a given start time.

new ElapsedTime (resolution)

Create a new elapsed time object with a resolution of SECONDS or MILLISECONDS, initialized with the current system time.

StartTime

Returns the start time of the specified elapsed time object using the current resolution of the elapsed time object. Please note that the start time may be reset.

Time

Returns the duration that has elapsed since the last reset of the timer. The duration is a float number measured with the elapsed time object's resolution - either SECONDS or MILLISECONDS.

Seconds

Returns the elapsed time in seconds since the last reset of the elapsed time object.

Milliseconds

Returns the elapsed time in milliseconds since the last reset of the elapsed time object.

Resolution

Returns the current resolution of the elapsed time object – either SECONDS or MILLISECONDS.

reset

Resets the specified elapsed time object. After reset, the start time is set to the current system time.

toText

Returns a text string giving the elapsed time of the elapsed time object. For example, “10.2500 milliseconds.”

Vector

A vector is a mathematical tool used to represent quantities that have both magnitude (size or value) and direction. Vectors are essential for describing and manipulating various aspects of a robot's motion and behavior. The blocks in this section allow you to create vector objects, manipulate these objects, and obtain information from these objects.

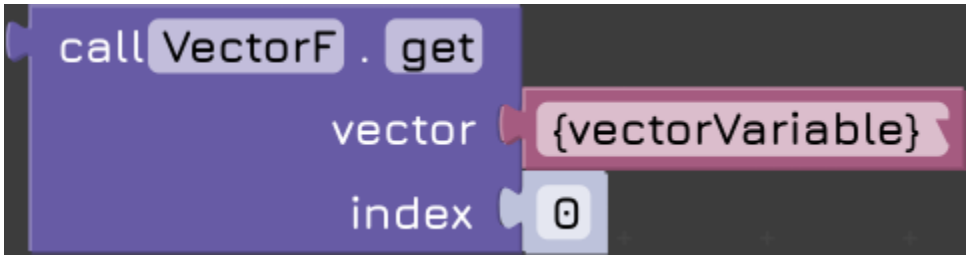
length

Returns the length of a given vector object.

Magnitude

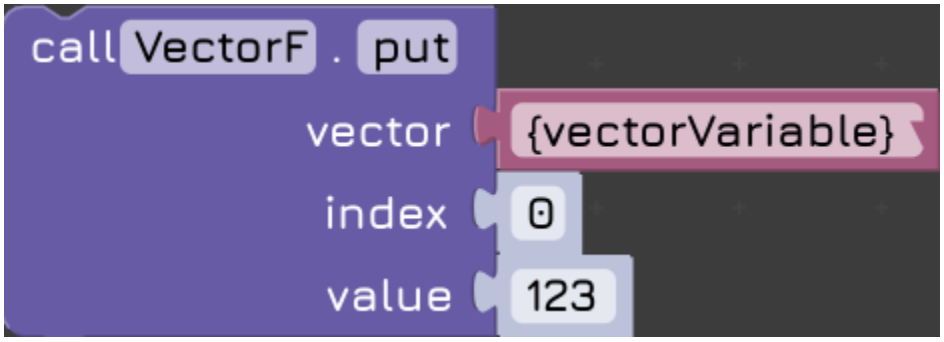
Returns the magnitude of a given vector object. Magnitude represents the vector’s length or size, regardless of its direction. It's calculated using the Pythagorean theorem.

get



Returns a particular of a given vector specified by the index value. Index starts from 0.

put



Update the given vector by updating the component specified by the index value and the given value. The other values in the vector are left intact.

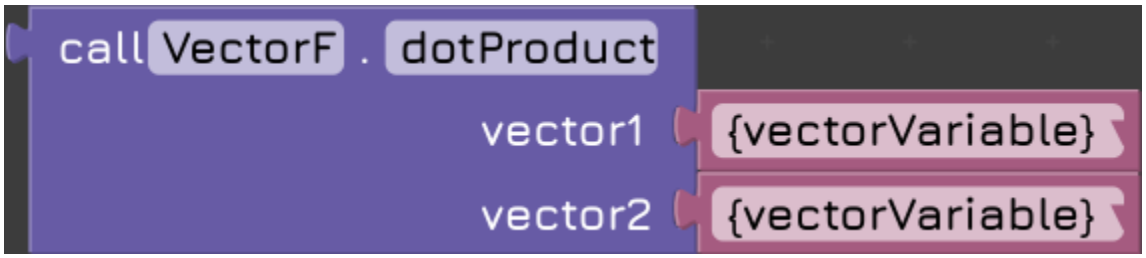
toText

Returns a text representation of the given vector object.

normalized3D

Given a vector that represents either a 3D coordinate or a 3D homogeneous coordinate, the function will return its normalized form. The result will always be a vector of length three, containing the coordinate values for x, y, and z at indices 0, 1, and 2, respectively.

dotProduct

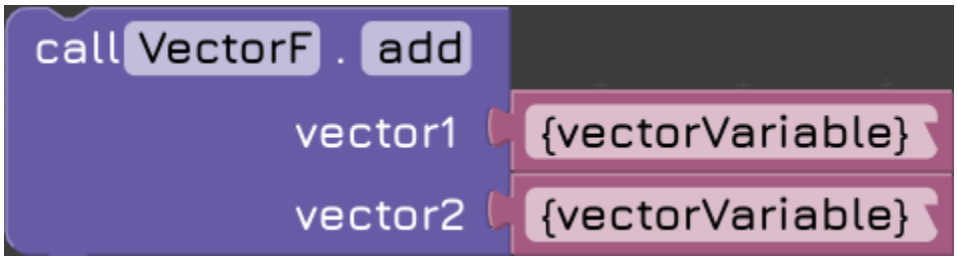


Returns a number representing the dot product of vector1 and vector2. The two vectors must be of the same length. The dot product can be expressed as $A \cdot B = |A|\,|B| \cos\theta$, where $|A|$ and $|B|$ are the magnitudes of vectors A and B, respectively and θ is the angle between the two vectors.

added

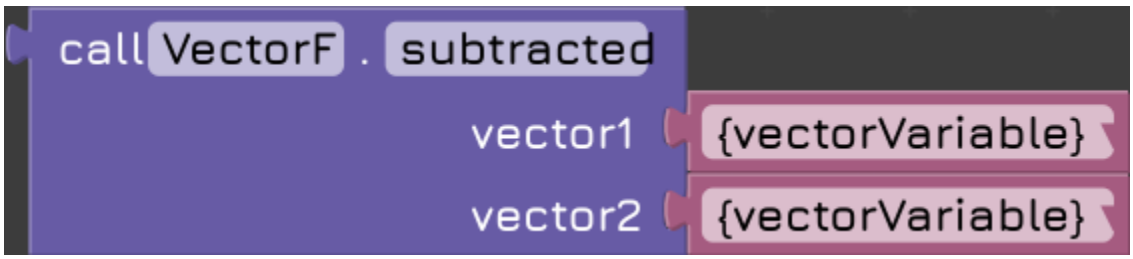
Returns a new vector representing the result of adding vector1 and vector2. Vector1 and vector2's values are not changed.

add



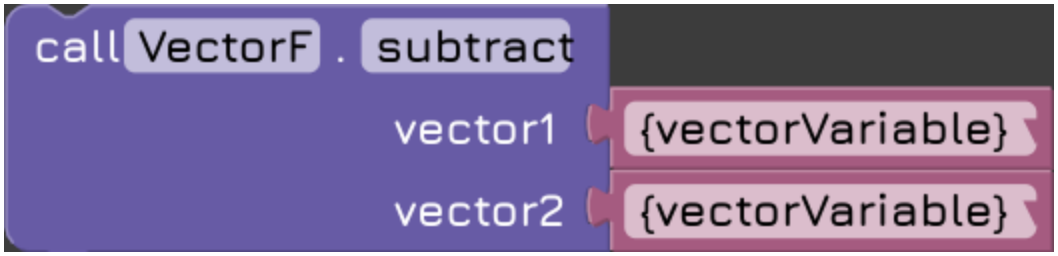
Updates vector1 to be the sum of vector1 and vector2. vector2 is left intact.

subtracted



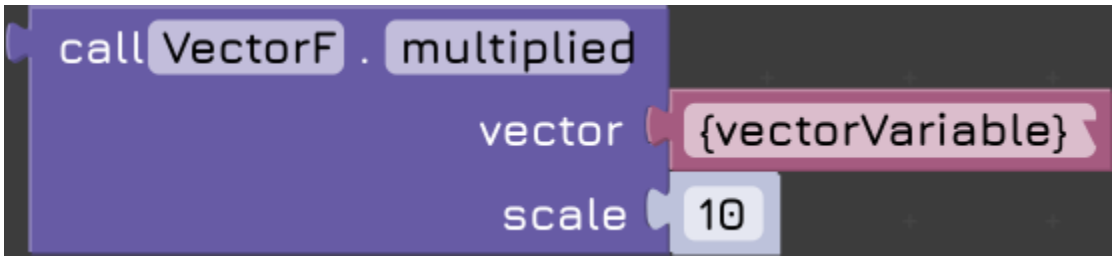
Returns a new vector representing the result of subtracting vector2 from vector1. Vector1 and vector2's values are not changed.

subtract



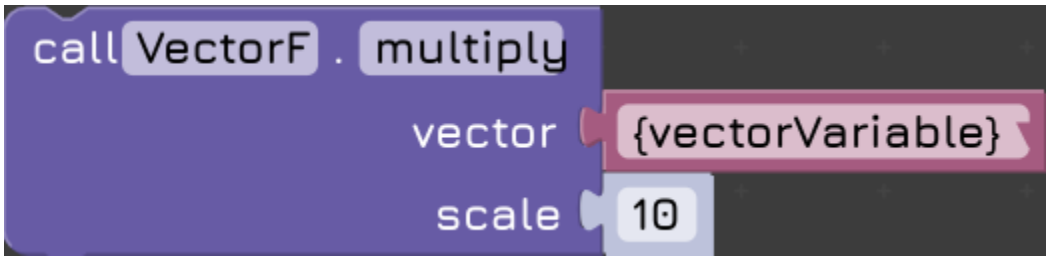
Updates vector1 to be vector1 subtracting vector2. vector2's value doesn't change.

multiplied



Returns a new vector containing the elements of a given vector scaled by the given scale value.

multiply



Updates a given vector, with each value scaled by the given scale.

new VectorF

Creates a new vector of the given length. Each value is default to float 0.

Velocity

Velocity describes the rate at which the robot's position changes over time, including both its speed and direction of motion. It's a crucial concept for understanding and controlling the robot's movement during matches. The blocks in this section allow you to create velocity objects, change distance units, and obtain information from these objects.

DistanceUnit

Returns the distance unit used in the velocity object. Values can be CM, INCH, METER, MM.

XVeloc

Returns the X attribute of a given velocity object.

YVeloc

Returns the Y attribute of a given velocity object.

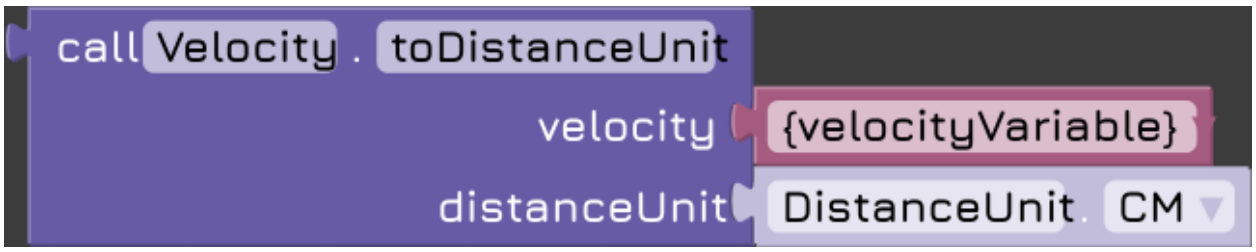
ZVeloc

Returns the Z attribute of a given velocity object.

AcquisitionTime

Returns the acquisition time of a given velocity object.

toDistanceUnit



Returns a velocity object based on the given velocity object after converting to the indicated units.

toText

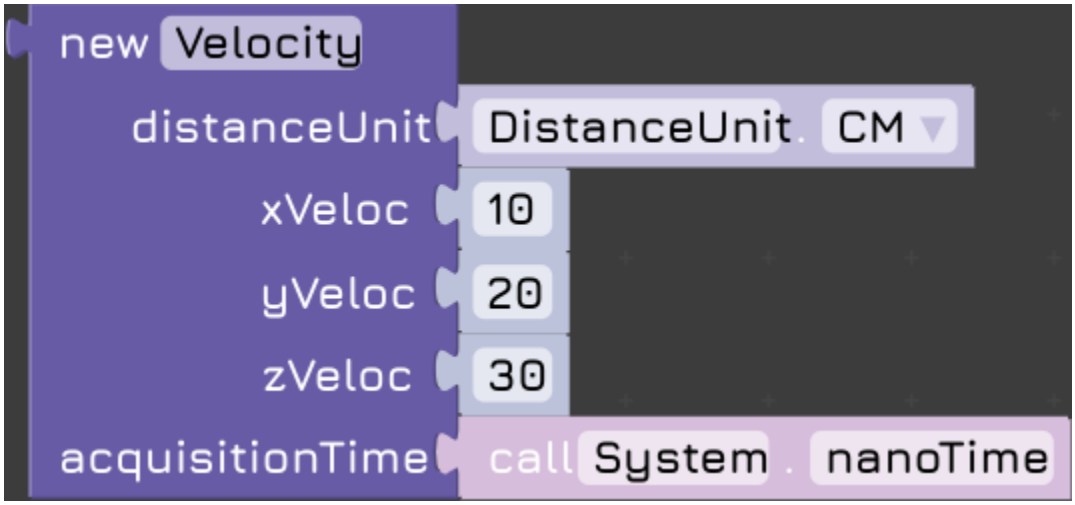
Returns the velocity attributes of the velocity object as a formatted string. The format of the string is “Velocity: (X Y Z)units/s” where X, Y, and Z are the

attributes of the velocity in the indicated units per second. For example, when X=1.2cm/s, Y=3.4 cm/s, and Z=5.6 cm/s, the string is “(1.200 3.400 5.600)cm/s”.

new Velocity

Returns a new velocity object with X=Y=Z=0. The default unit is CM.

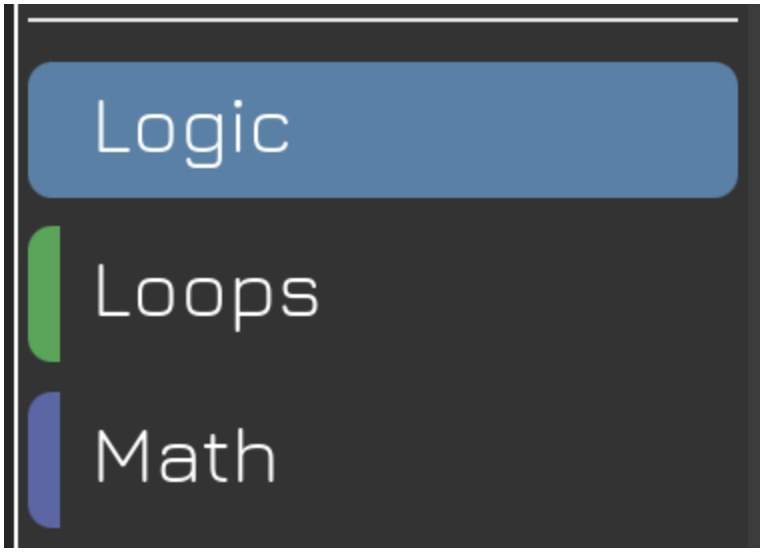
new Velocity (expanded)



Returns a new velocity using the x, y and z values measured in the given units and the current system time as the acquisition time.

Logic

The Logic section is your gateway to creating intelligent and adaptable robot behaviors. It houses a collection of blocks that enable decision-making, conditional execution, and control flow in your programs.



if do

Allows you to execute specific code blocks if a certain condition is true.

Key points:

- Condition: This is the part of the block where you specify the condition to be evaluated. It can be a comparison, a boolean value, or the result of a sensor reading.
- Do block: This block contains the code that will be executed if the condition is true.

if do else

A fundamental control flow structure that allows you to execute different code sections based on whether a condition is true or false.

Key points:

- If condition: Specifies the condition to be evaluated.
- Do block: Contains code to be executed if the condition is true.
- Else block: Contains code to be executed if the condition is false.

if do else if do else

Control flow structure in FTC that allows you to execute different code sections based on multiple conditions.

Key points:

- If condition: Specifies the first condition to be evaluated.
- The first do block: Contains code to be executed if the first condition is true.
- Else if condition: Specifies the second condition to be evaluated if the first condition is false.
- The seconds do block: Contains code to be executed if the second condition is true.
- Else block: Contains code to be executed if none of the previous conditions are true.

comparison operators

Returns true if both inputs are equal to each other.

Key points:

- The two inputs must be of the same type.
- There are more comparison operators that you can choose from the drop down list shown below.

and

Returns true if both inputs are true.

Key points:

- Used to combine two conditions.
- Returns true only if both conditions are true.
- Returns false if any of the conditions are false.

or

Returns true if one of the inputs is true.

Key points:

- Combines two conditions.
- Returns true if at least one condition is true.
- Returns false only if all conditions are false.
- Can’t find it? You need to first choose ‘and” box, then choose “or” from the drop down list.

not

A logical operator used to negate a condition.

Key points:

- Reverses the logical value of a condition.
- Returns true if the condition is false.
- Returns false if the condition is true.

true/false

Often referred to as Boolean values. Default is true. You can choose false from the dropdown list.

- True: Represents a logical condition that is satisfied or correct.
- False: Represents a logical condition that is not satisfied or incorrect.

Key points:

- Conditional statements: Determine the flow of a program based on whether a condition is true or false.
- Logical operations: Combined with operators like AND, OR, and NOT to create complex conditions.
- Boolean data types: You could define a variable to store and manipulate true/false values.

null

Represents an empty or non-existent value. It's often used to indicate the absence of data or an undefined state.

Key points:

- Represents absence: Indicates that a variable or object has no value assigned.
- Error handling: Can be used to handle situations where a value might be missing or invalid.

- Placeholder: Often used as a default value before data is available.

test if true if false

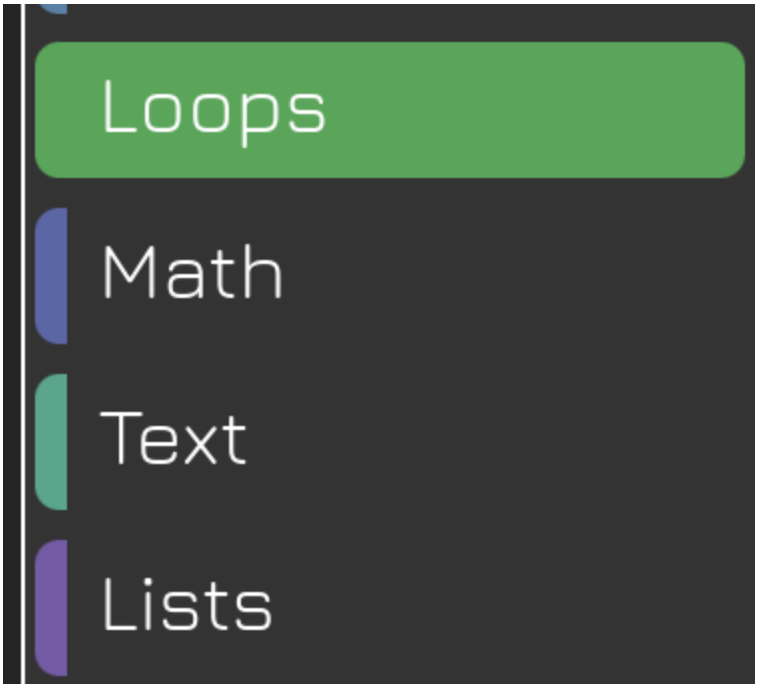
Check the condition in “test”. If the condition is true, return “if true” value. Otherwise, returns the “if false” value.

Key points:

- “if true" and "if false" values can be of any time.

Loops

Loops empower you to execute a sequence of code blocks multiple times, either a fixed number of times or until a certain condition is met. They streamline your code, making it more efficient and adaptable.



repeat times do

A programming construct that executes a specific block of code for a predetermined number of times.

Key points:

- Repeat _ Times: Indicates the number of times the code block should be executed.

- The do block won’t be executed if the given number of times is smaller or equal to 0.
- Do: Specifies the action being repeated.
- It is commonly used for
    - Iteration: Performing an action multiple times.
    - Data processing: Processing elements of an array or list.

repeat while do

Often referred to as a while loop, is a programming construct that repeatedly executes a block of code as long as a specified condition is true.

Key points:

- The condition is checked before each iteration.
- If the condition is true, the code within the loop is executed.
- If the condition is false, the loop terminates.
- There's a risk of infinite loops if the condition never becomes false.

repeat until do

A programming construct that repeatedly executes a block of code until a specified condition becomes true.

Key points:

- The condition is checked at the end of each iteration.
- The code within the loop is executed at least once.
- The loop continues as long as the condition is false.
- There's a risk of infinite loops if the condition never becomes false.

count with from to by do

A programming construct used to repeatedly execute a block of code a specific number of times.

Key points:

1. count variable (i in the example): Sets up the loop counter variable. The variable is automatically created.
2. from: Specify the start value of the variable.
3. to: Specify the end value. The end value itself is included. For example, in the example, the do statement will be executed 10 times.
4. by (Increment/Decrement): Updates the loop counter after each iteration. If the to value is larger than or equal to the from value, the variable is

increased during each iteration, no matter whether the by value is positive or negative. If the to value is smaller than the from value, the variable is decreased during each iteration, no matter whether the by value is positive or negative.

for each item in list do

A programming construct used to iterate over each element in a collection (like an array or list).

Key points:

- Iterates through each item in a collection.
- Simplifies iteration compared to traditional for loops.
- Often used with collections like arrays, lists, sets, or dictionaries.
- The loop variable takes on the value of each element in the collection.

break out of loop

Used to prematurely terminate a loop.

Key points:

- Immediately exits the current loop.

- Skips the remaining iterations of the loop.

Common Use Cases:

- Finding a specific value in a list or array.
- Handling unexpected conditions that require immediate loop termination.
- Optimizing loop performance by exiting early when a result is found.

continue with next iteration of loop

Used to skip the rest of the current iteration of a loop and proceed to the next iteration. How it works:

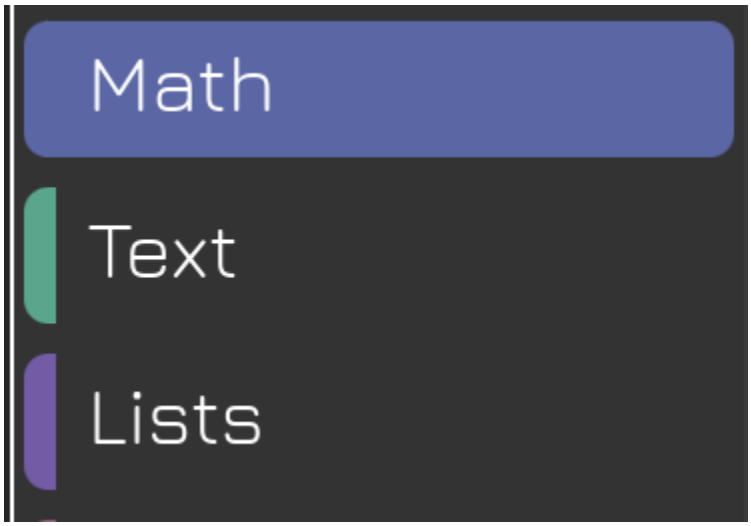
- When encountered within a loop, the continue statement immediately terminates the current iteration.
- The program flow jumps back to the beginning of the loop for the next iteration.

Key points:

- Continue is often used in conjunction with conditional statements to skip certain iterations based on specific criteria.
- It's different from break, which terminates the entire loop.
- By using continue, you can efficiently process data or perform actions on specific elements within a loop while skipping others.
- Can’t find it? You need to choose the “break out of loop” block, then choose “continue with next iteration” from the drop down menu.

Math

The Math section in your block programming environment equips you with a versatile toolkit for performing various mathematical operations. These blocks enable your robot to make precise calculations, process sensor data, and execute complex maneuvers with greater accuracy.



number

A fixed numerical value that can be used in calculations, comparisons, or as input to other blocks.

Key points:

- Fixed value: The number cannot be changed during program execution.
- Data type: Can represent integers, decimals, or other numeric formats as needed.

- Usage: Used in various calculations, comparisons, and as input to other blocks.

+

Returns the sum of two numbers.

Key points:

- The order of operands is not important.
- Can be used with both integers and floating-point numbers.
- Often combined with other arithmetic operations to form more complex expressions.

-

Returns the difference between two numbers.

Key points:

- The order of operands is important. Subtracts the second number from the first number.
- Can be used with both integers and floating-point numbers.
- Often combined with other arithmetic operations to form more complex expressions.

*

Returns the product of two numbers.

Key points:

- The order of operands is not important.
- Can be used with both integers and floating-point numbers.
- Often combined with other arithmetic operations to form more complex expressions.

/

Returns the quotient of the first number (dividend) divided by the second number (divisor).

Key points:

- Division by zero: Attempting to divide a number by zero results in infinity or negative infinity.
- Integer division: Dividing two integers results in an integer quotient if dividend is divisible by the divisor. If not divisible, return a float number.
- Floating-point division: Dividing floating-point numbers produces a floating-point result.

^

Returns the first number raised to the power of the second number.

Key points:

- The first number is the base, and the second number is the exponent.
- Exponentiation can be used for various calculations, including growth models, scientific formulas, and financial calculations.

negative

Used as a unary operator to negate a numerical value.

Key points:

- Can be applied to both integers and floating-point numbers.
- Often used in mathematical calculations and expressions.

square root

Calculates the square root of a non-negative number.

Key points:

- Returns the value that, when multiplied by itself, equals the input number.
- The square root of an integer may be a float number.

- If a negative number is provided, the result becomes NaN.

absolute

Calculates the absolute value of a number.

Function:

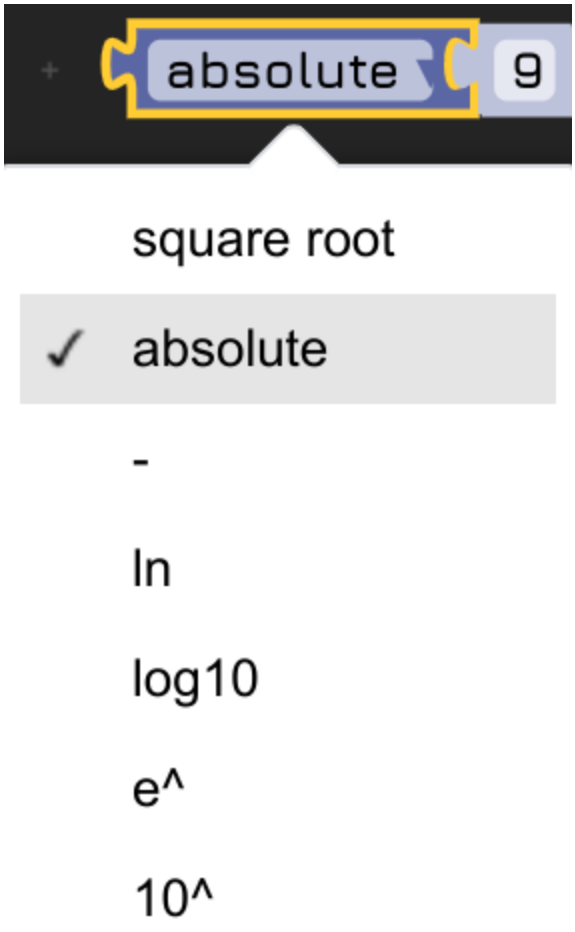
Takes a real number (positive, negative, or zero) as input. Returns the distance of the number from zero, which is always a non-negative value.

Key points:

- The absolute value of a positive number is the number itself.
- The absolute value of a negative number is its positive counterpart.
- The absolute value of zero is zero.

You can choose the other operations from the drop list of absolute blocks:

- ln. Calculates the natural logarithm of a positive number.
- log10: Calculates the logarithm base 10 of a positive number.
- e^: Calculates the exponential of a number, where the base is the mathematical constant 'e' (approximately 2.71828).
- 10^: Calculates the exponential of a number, where the base is 10.



sin

Calculates the sine of an angle in degree (not radian).

Key points:

- Returns a value between -1 and 1, representing the sine of the angle.
- The sine function is periodic with a period of 2π radians (or 360 degrees).
- The sine of 0 is 0 (sin(0) = 0).
- The sine of π/2 radians (or 90 degrees) is 1 (sin(π/2) = 1).

- The sine of π radians (or 180 degrees) is 0 (sin(π) = 0).
- The sine of 3π/2 radians (or 270 degrees) is -1 (sin(3π/2) = -1).

You can choose the other operations from the drop list of sin blocks:

- cos: Calculates the cosine of an angle in degree (not radian).
- tan: Calculates the tangent of an angle in degree (not radian). Returns a value, which can be any real number, representing the tangent of the angle. The tangent function is undefined for angles that are odd multiples of π/2 radians (or 90 degrees).
- asin: Calculates the arcsine (or inverse sine) of a number, which is the angle whose sine is the input number. Takes a number between -1 and 1 as input and returns an angle in degree. The arccosine of a number outside the range of -1 to 1 is NaN.
- acos: Calculates the arccosine (or inverse cosine) of a number, which is the angle whose cosine is the input number. Takes a number between -1 and 1 as input and returns an angle in degree. The arccosine of a number outside the range of -1 to 1 is NaN.
- atan: Calculates the arctangent (or inverse tangent) of a number, which is the angle whose tangent is the input number. It takes a real number (positive, negative, or zero) as input.

atan2

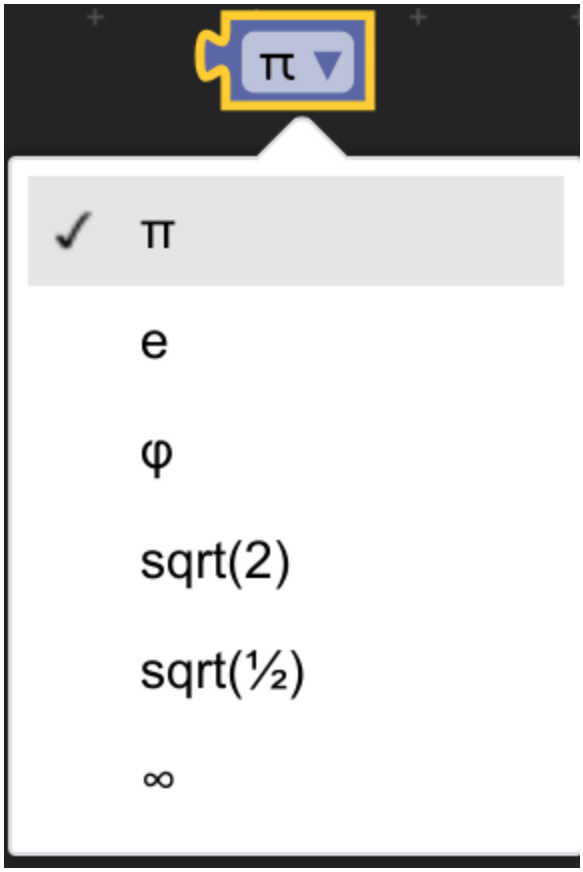
Returns a numerical value between -180 and +180 degrees, representing the counterclockwise angle between the positive X axis, and the point (x, y).

π

A mathematical constant representing the ratio of the circumference of a circle to its diameter, which is approximately 3.14159.

You can choose the other constants from the dropdown list:

- e: A mathematical constant that is the base of the natural logarithm. The number is approximately 2.71828.
- φ: Golden Ratio or Phi. The value is approximately 1.61803.
- sqrt(2): square root of 2. Roughly 1.4142.
- sqrt(1/2): square root of 1/2. Roughly 0.7071.
- ∞: A concept representing a value that is greater than any finite number. There are two types of infinity: positive infinity (+∞) and negative infinity (-∞). Positive infinity represents a value greater than any positive number, while negative infinity represents a value smaller than any negative number.



is even

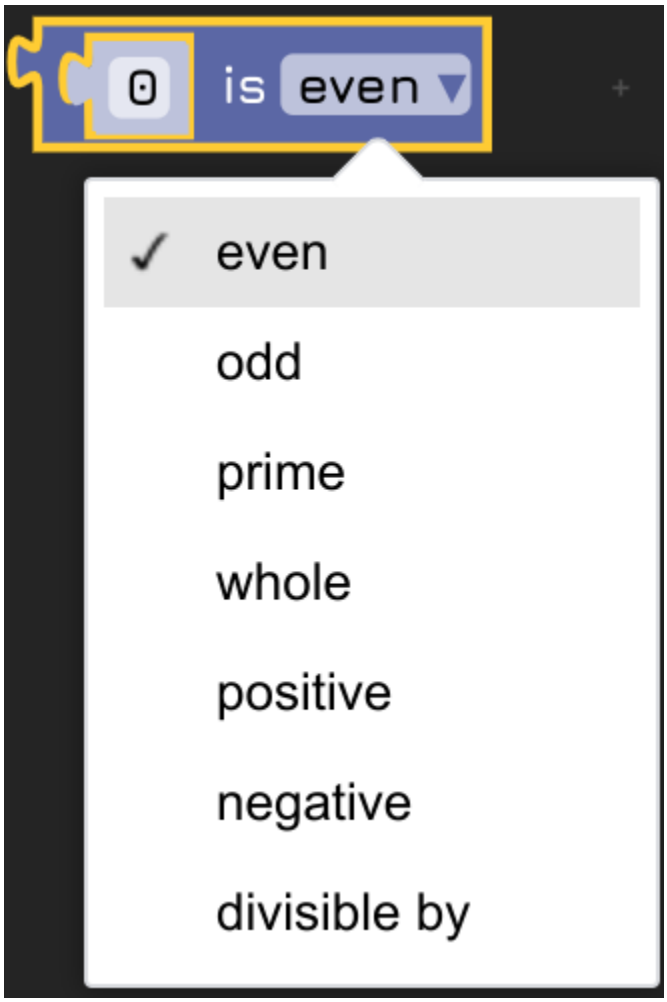
Determines whether an integer is even or not.

Key points:

- Returns True if the integer is even, False if the integer is odd.
- An even integer is any integer that is exactly divisible by 2 (i.e., leaves no remainder when divided by 2).
- An odd integer is any integer that is not exactly divisible by 2.

You can choose the tests of numbers from the dropdown list:

- odd: Determines whether an integer is odd or not.
- prime: Determines whether an integer is a prime number or not. A prime number is divisible only by 1 and itself.
- whole: Returns true if the number is an integer (whole number), false if it's a fraction or decimal.
- positive: Returns true if the number is greater than zero. Returns false otherwise.
- negative: Returns true if the number is less than zero, false otherwise.
- divisible by: Returns true if the number is divisible by another given number.



round

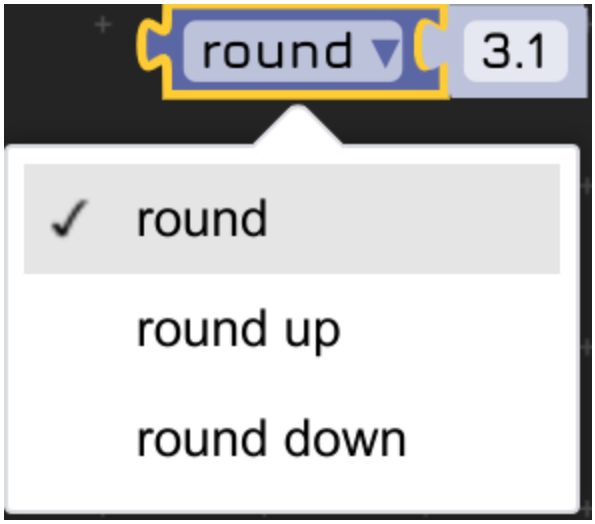
Rounds a floating-point number to the nearest integer or to a specified number of decimal places.

Key points:

- If the fractional part is 0.5 or greater, the number is rounded up. If the fractional part is less than 0.5, the number is rounded down.
- Rounding helps to simplify calculations and make numbers easier to work with.
- Rounding can introduce errors, so it is important to be mindful of the context in which it is used.

From the drop list, you can choose “round up”, or “round down”.

- round up: Rounds a floating-point number up to the nearest integer or to a specified number of decimal places.
- round down: Rounds a floating-point number down to the nearest integer or to a specified number of decimal places.



sum of list

Returns the sum of all numbers in a list.

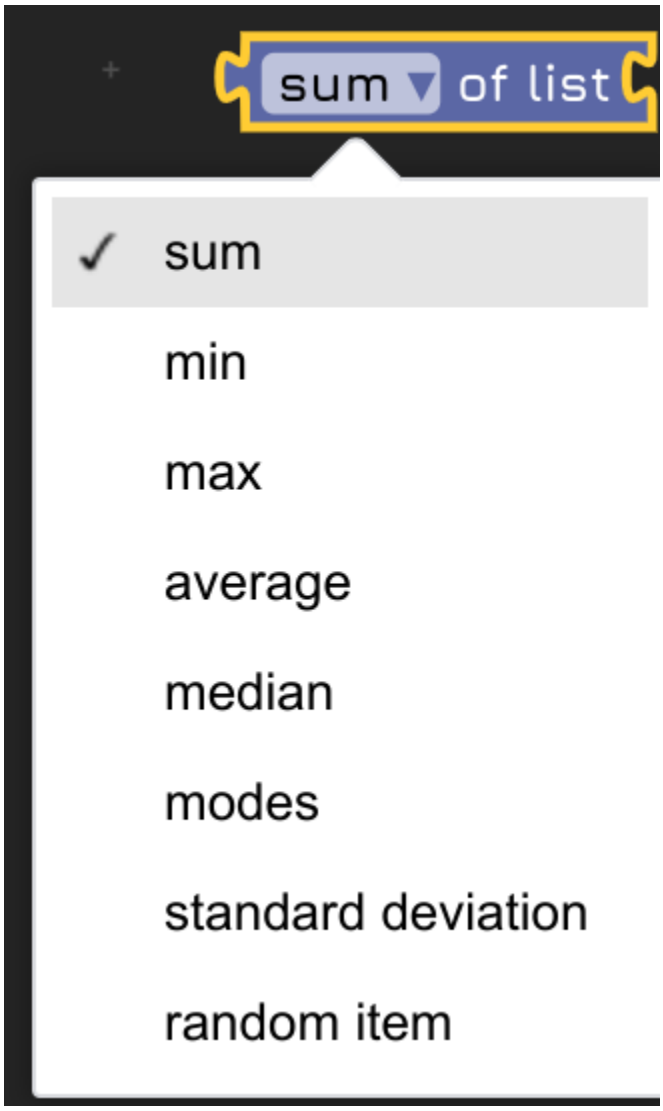
Key points:

- Return a single float number.
- The list can be of any length.

From the drop down list, you can find the other operations against a list:

- min of list: Finds the minimum (smallest) value in a list of numbers.
- max of list: Finds the maximum (largest) value in a list of numbers.
- average of list: Calculates the average (mean) of all elements in a list of numbers.
- median of list: Calculates the median value of a list of numbers.
- modes of list: Calculates the modes (most frequent items) of a list of numbers.

- standard deviation of list: Calculates the standard deviation of a list of numbers.
- random item of list: Selects a random item from a list of numbers.



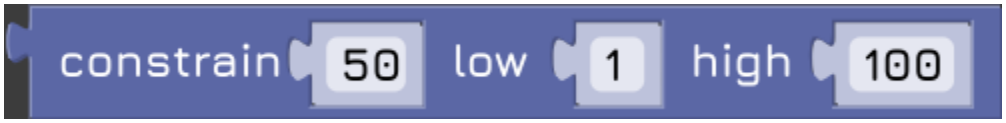
remainder of

Calculates the remainder when one integer is divided by another.

Key points:

- Both inputs must be integers.
- Returns an integer representing the remainder of the division.

constrain low high



Limits a given value to a specified range defined by a lower bound (low) and an upper bound (high). The limits are inclusive.

Key points:

- If the given value is within the range [low, high], returns value.
- If the given value is less than low, returns low.
- If the given value is greater than high, returns high.

random integer from to

Generates a random integer within a specified range, inclusive of both the lower and upper bounds.

Key points:

- Uses a random number generator to produce a value within the range.
- Ensures that the lower bound is less than or equal to the upper bound.

- The generated integer has a uniform distribution, meaning each value in the range has an equal probability of being chosen.

random fraction

Generates a random fraction between 0 and 1.

Function:

Takes no input.

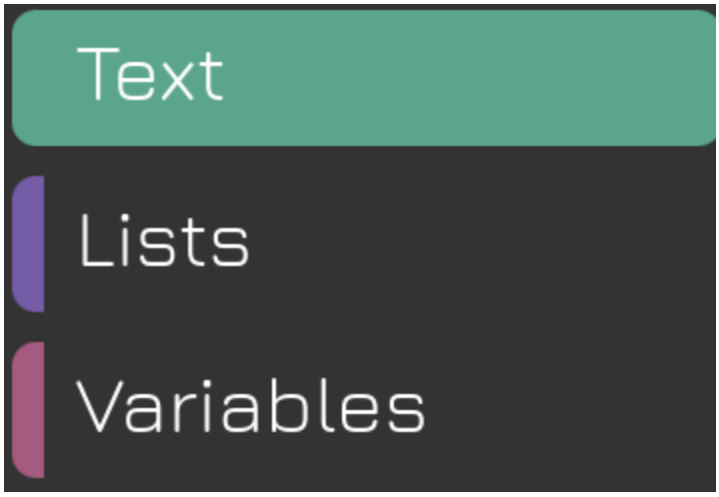
Returns a floating-point number between 0 (inclusive) and 1 (exclusive).

Key points:

- Uses a random number generator to produce a value within the range [0, 1).
- The generated fraction has a uniform distribution, meaning each value in the range has an equal probability of being chosen.

Text

The Text section offers a collection of blocks that enable you to work with text and characters, facilitating communication, data display, and user interaction within your robot programs.



“”

A letter, word, or line of text. It allows you to input an initial text value.

create text with

Returns a new string by combining two texts.

Key points:

- It takes two strings as input.
- In the new string, the first (top) string is followed by the second (bottom) string.

to append text “”

Appends text to a variable.

Key points:

- The to variable must be a string.
- the text is appended after the original string item.

length of “”

Returns the number of characters in the input text string.

“” is empty

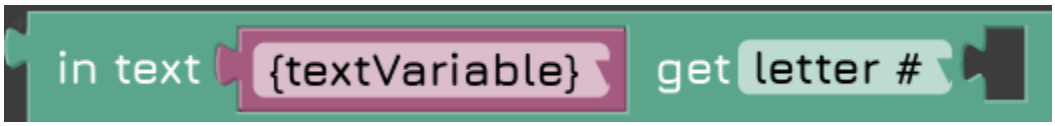
Returns true if a given text input is empty.

In text find first occurrence of text “”

Finds the first occurrence of a specific text within a variable. The index starts from 1. Returns 0 if text is not found. For example, the first occurrence of “abc” in “abc abc” is 1.

From the dropdown list, you can choose to find the last occurrence of a text. If a text can not be found, returns 0. For example, the last occurrence of “abc” in “abc abc” is 4.

In text find get letter



Returns the letter in the specified location. #1 is the first item.

In text find get substring from letter to letter

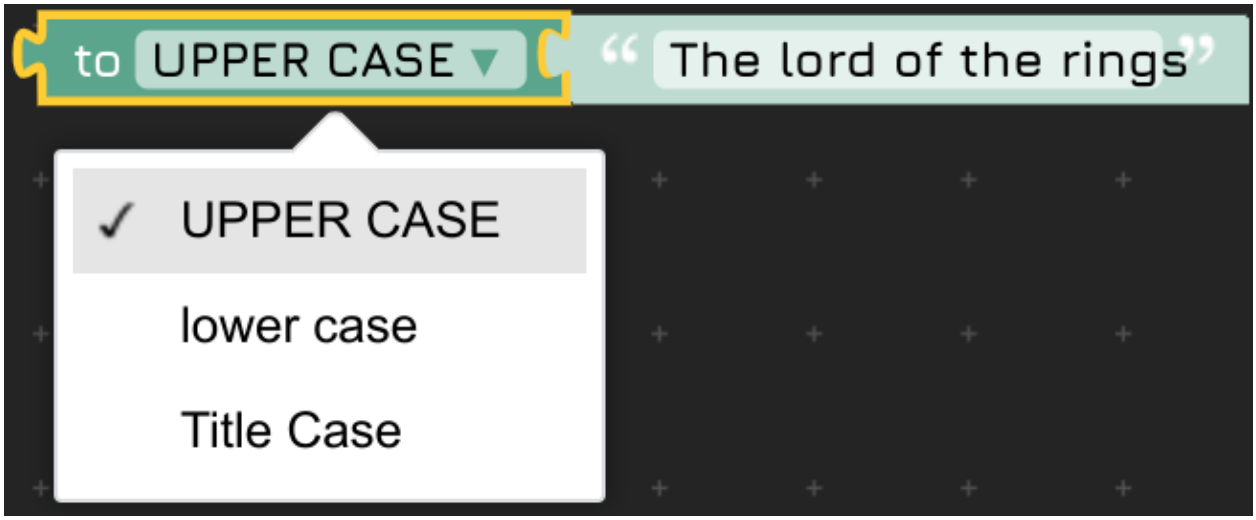
Returns the substring of the text variable specified between “from” and “to” values.

to Upper Case

Returns uppercase of a string. In the example above, it returns “ABC”.

From the drop down list you can choose the other cases:

- to lower Case. Returns lowercase of a string. For example, “the lord of the rings”.
- to Title Case. Returns title case of a string. For example, “The Lord Of The Rings”.

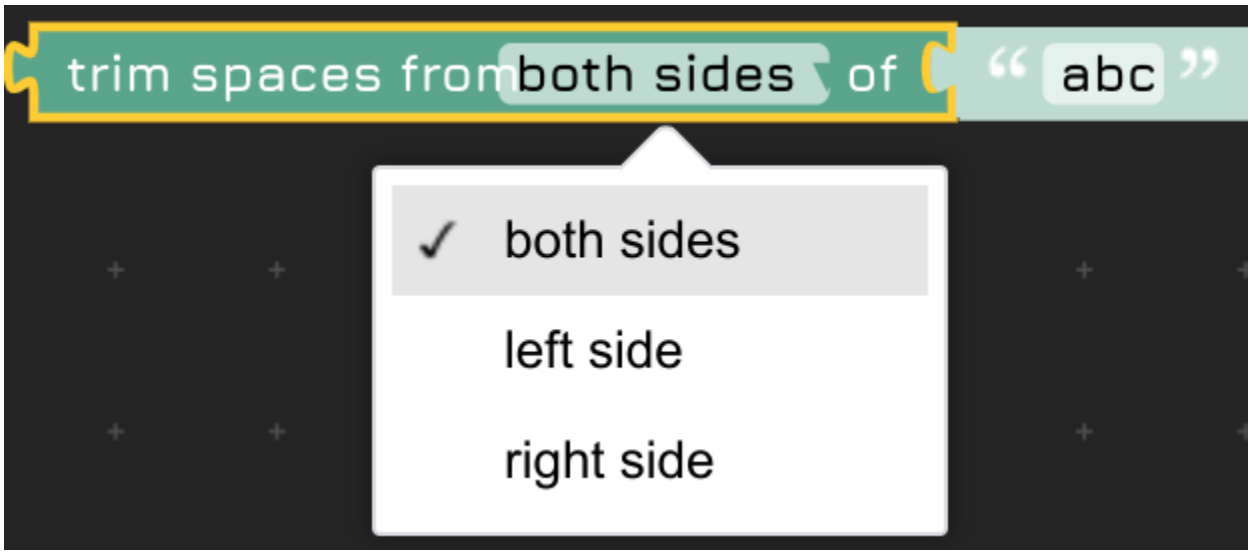


trim spaces from both sides

Trim spaces from both sides of a string.

From the drop down list you can choose the other ways of trimming spaces:

- trim spaces from left sides. Trim spaces from the left side of a string.
- trim spaces from right sides. Trim spaces from the right side of a string.



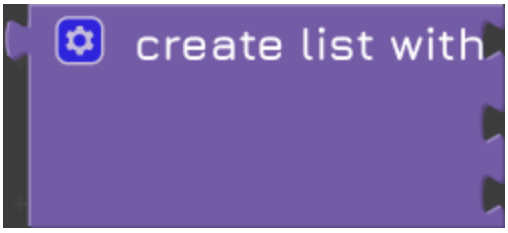
Lists

Lists are a fundamental data structure used to store multiple pieces of information in a single variable. Think of a list as a container that holds a collection of items arranged in a specific order. Each item within the list is assigned an index, which allows you to access and manipulate individual elements. The blocks in this section allows you to create lists, manipulate them, and retrieve items from them.

create empty list

Creates an empty list.

create list with



Creates a list with any number of items.

create list with item repeated times

Creates a list with the same item repeated a given number of times.

length of

Returns the length of a given list.

is empty

Returns whether the given list is empty or not.

add item to list

Adds an item to the end of the list.

In list find first occurrence of item

Returns the position of the first occurrence of the item in the list. The index starts from 1. Returns 0 if not found.

Key points:

- The function iterates through the list in order to find the first occurrence.
- It can handle lists containing elements of any data type.

In list find last occurrence of item

Returns the position of the last occurrence of the item in the list. The index starts from 1. Returns 0 if not found.

Key points:

- The function iterates through the list in reverse order to find the last occurrence.
- It can handle lists containing elements of any data type.
- Can't find it? You need to click the "first" to get the dropdown menu and then choose last.

In list get/get and remove/remove #/#from end/first/last/random

This legend shows the block of getting a particular element at the specified location of a list. #1 is the first item.

Key points:

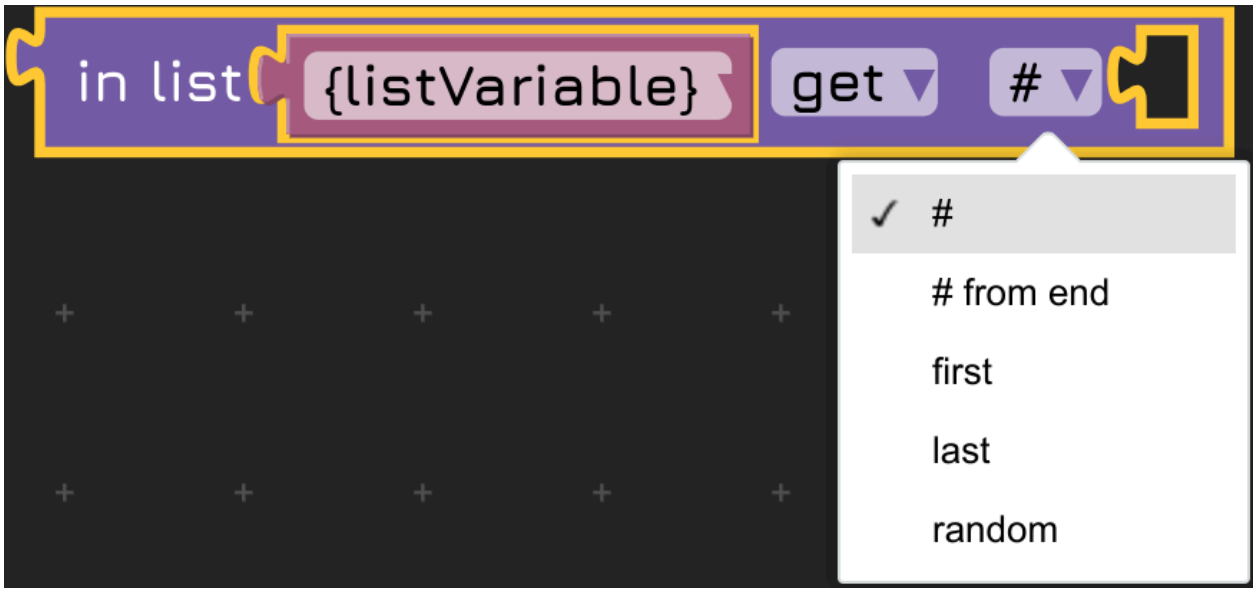
- The index parameter is 1-based, meaning the first element is at index 1, the second at index 2, and so on.
- Return value undefined when the index is smaller than 1.
- Return value undefined when the index is larger than the length of the list.

This block can be extended in multiple ways. In addition to "get", you can choose "get and remove", or "remove".

- get and remove: Returns a particular element at the specified location of a list and removes the element in the list.
- remove: Removes a particular element at the specified location of a list.

The position of the element can be specified by multiple ways:

- # from end: The index from the end. #1 is the last element.
- first: the index of the first element, essentially #1.
- last: the index of the last element, essentially #length.
- random: a random position in the list.



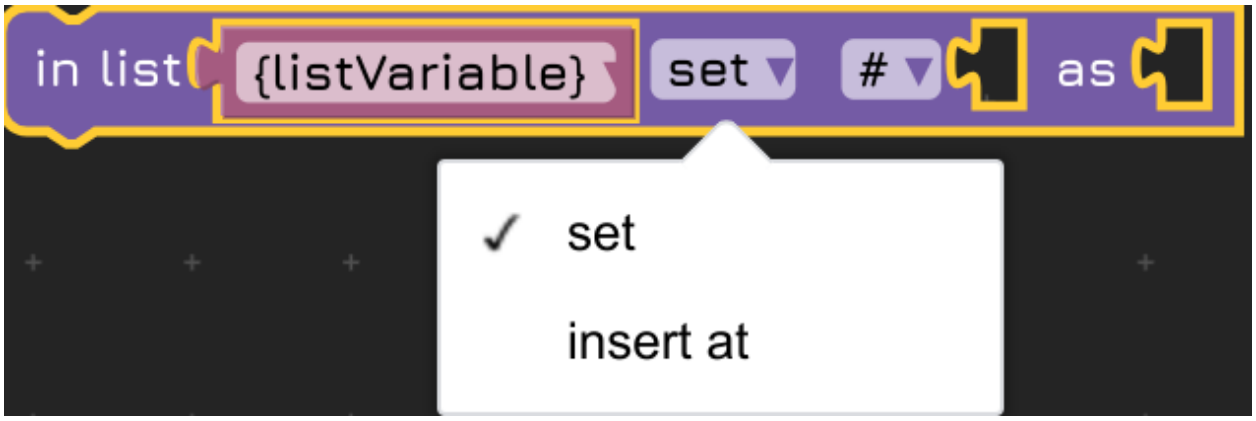
In list set/insert at and remove/remove #/#from end/first/last/random as

This function sets a particular element at the specified location of a list to the given value. #1 is the first item.

Key points:

- These functions provide flexibility for modifying lists.
- The index parameter is 1-based, meaning the first element is at index 1, the second at index 2, and so on.
- There is no change of the list when the index is smaller than 1.
- There is no change of the list when the index is larger than the length of the list.

This block can be extended in multiple ways. In addition to “set”, you can choose “insert at”.

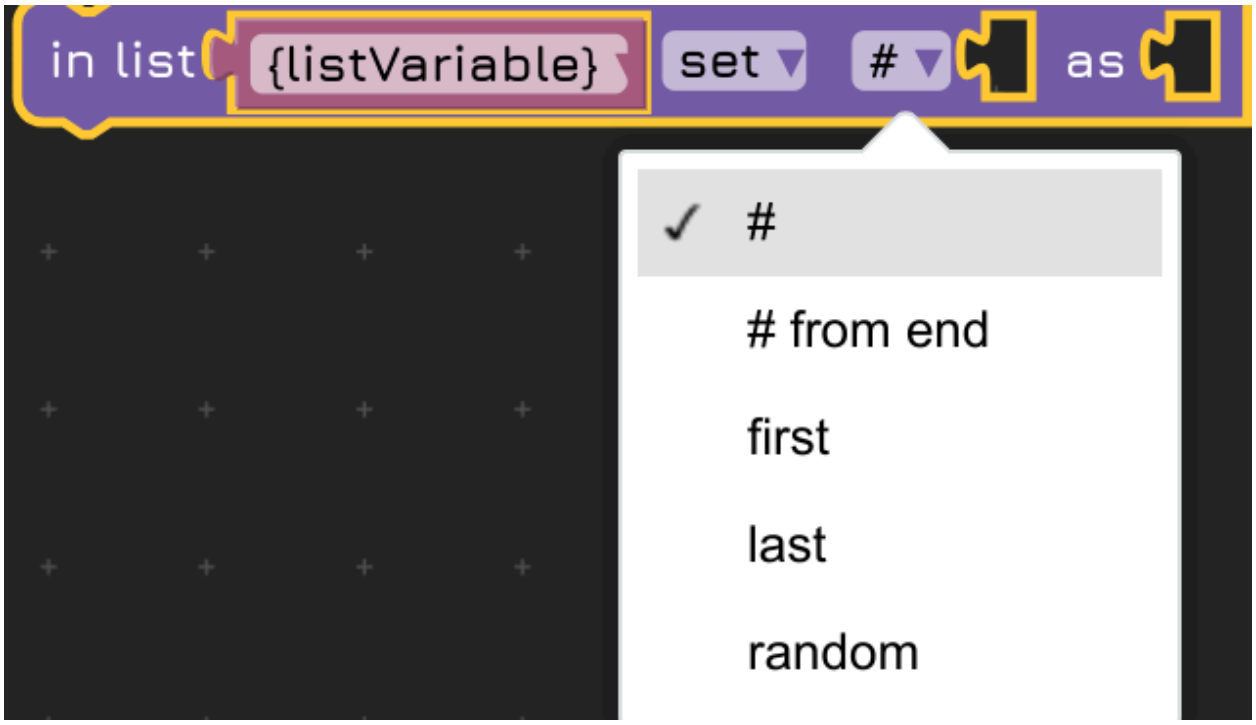


- insert at: Insert the given item at the specified position in a list. #1 is the first item.

The position of the element can be specified by multiple ways:

- # from end: The index from the end. #1 is the last element.
- first: the index of the first element, essentially #1.

- last: the index of the last element, essentially #length.
- random: a random position in the list.



In list get sub-list from #/from end/first to #/from end/last

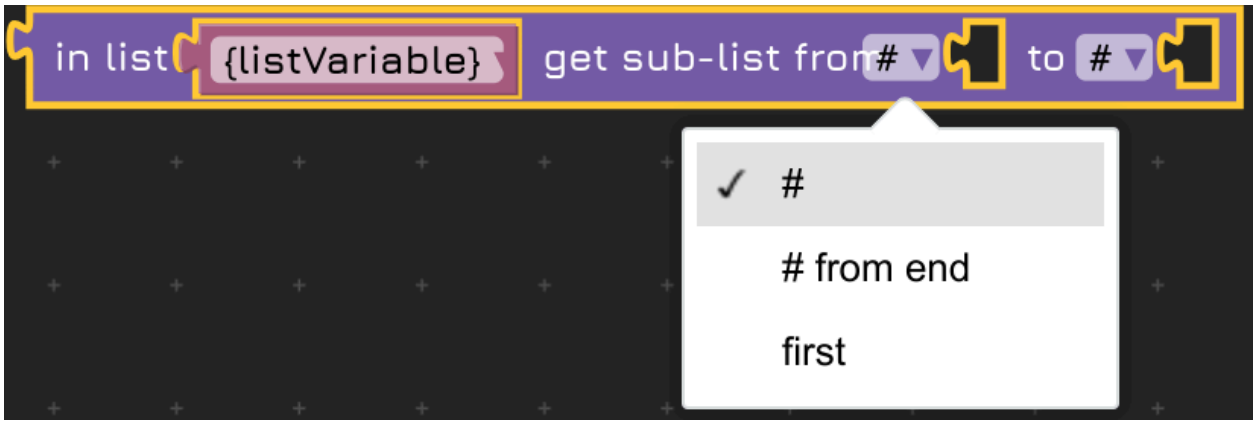
Returns a new sub-list, which is a copy of the specified portion of the list specified by start and end locations. The start and end location are included.

Key points:

- The function does not modify the original list; it creates a new sub-list.
- The index must be valid (between 1 and length of the list, inclusive). Otherwise the return value is undefined.

The from and to location can be specified in other ways:

- # from end: The index from the end. #1 is the last element.
- first: the index of the first element, essentially #1. Can only be used as from location.
- last: the index of the last element, essentially #length. Can only be used as to location.



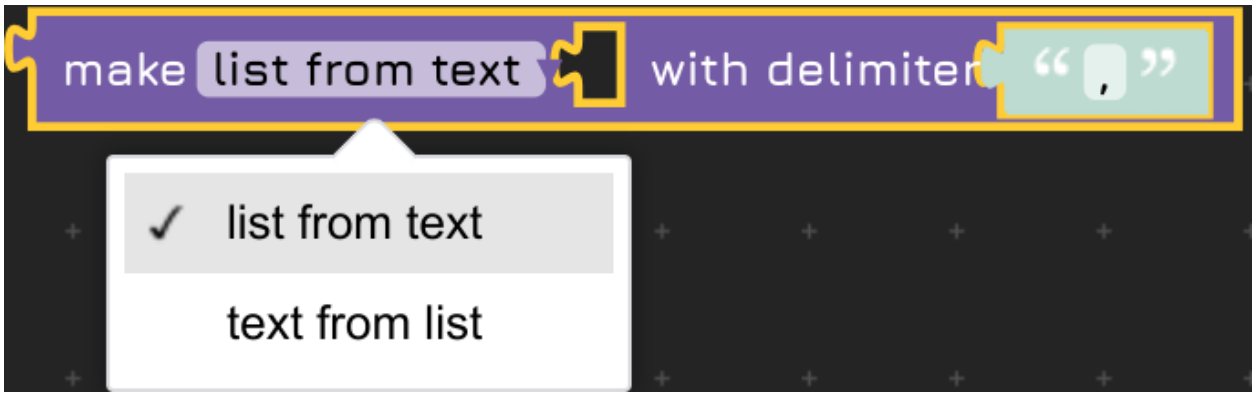
make list from text/text from list with delimiter “”

Split the given text into a list of texts, breaking at each delimiter. Return the list of texts.

Key points:

- The delimiter can be any character or text, such as a comma (,), space ( ), semicolon (;), or pipe (|), “abc”.
- If the delimiter is an empty text, the given text will be broken character by character.

You can choose “text from list” in the dropdown list. The “text from list” block combines the list of texts with the given delimiter. The returned value is a text.



sort numeric/alphabetic/alphabetic, ignore case ascending/descending

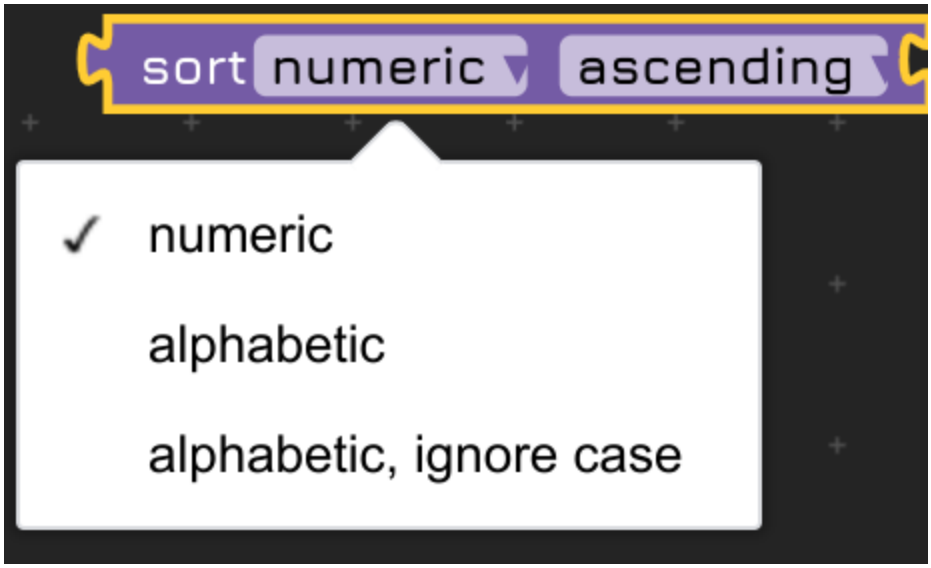
This block sorts a list of numbers in ascending order.

Key points:

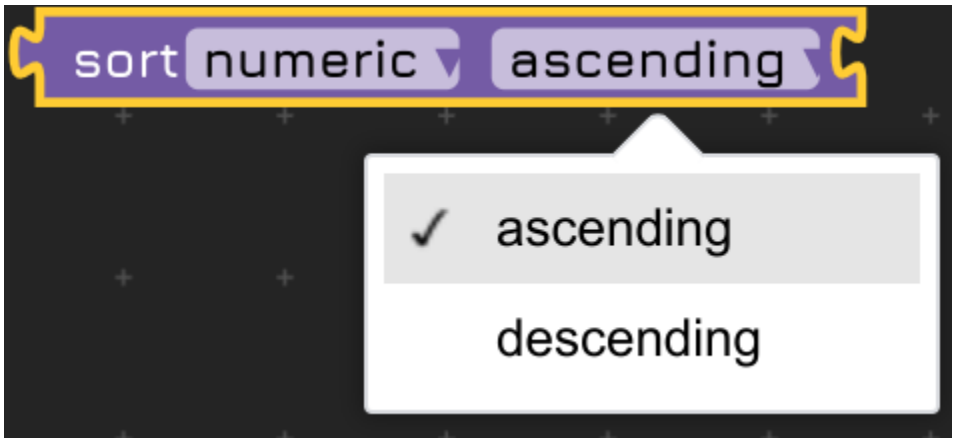
- The functions use different sorting algorithms depending on the data type (numeric, alphabetic, or alphabetic and ignore case) and the desired order.
- For numeric sorting, the natural ordering of numbers is used.
- For alphabetic sorting, the lexicographical order (dictionary order) is used, with optional case-insensitive comparison.

You can choose to sort more than numeric type, depending on the item type of the list.

- alphabetic: sort in lexicographical order. Only applied to text.
- alphabetic, ignore case: sort in lexicographical order, case insensitive. Only applied to text.



In addition to ascending order, you can also choose descending order.



Variables

A variable is a named container that stores a value. It's like a labeled box where you can put different things inside. You can create variables, change the values they hold (assign new values to them), and use them in your code to perform various operations or make decisions.

create variable

Create a new variable.

Key points:

- The variable name has to be unique.
- After the variable is created, you can see three additional blocks in the variables section: set value, change value, and get value.

In the following examples, we assume we have created a variable called “newVar”.

set to

Set a variable to a specified value.

Key points:

- The variable can be of any type: text, integer, float, vector…

change by

Add the numeric value (e.g. 1) to the variable.

Key points:

- This function assumes the variable contains a numeric value.
- If the variable does not have a numeric value (e.g. text, vector), the system auto converts the variable to number 0 when the “change by” block is applied.

get variable Value

Returns the value of a variable by choosing it. For example, you can get the value of the newVar variable here.

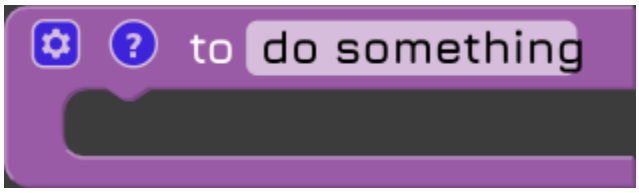
Key points:

- The variable is global. So you can get its value anywhere in the program.
- The variable needs to be set or changed before you can get value. Otherwise, the return value is undefined.

Function

A function is a reusable block of code that performs a specific task. It's like a mini-program within your main program. You give it a name, define what it does, and then you can "call" or use it multiple times in your code without having to write the same instructions repeatedly.

to do something



Creates a function with no output.

Key points:

- You can change “do something” to a unique tag for your future reference.
- This function has no output.

to do something return

Creates a function with an output.

Key points:

- You can change “do something” to a unique tag for your future reference.
- This function returns a value as specified.

if return

If a value is true, then return the second value.

Key points:

- This block can only be used within a function.
- When the condition is true and the value is returned, no other blocks will be executed anymore.

runOpMode

Run the user defined function “runOpMode”. You may have defined a function with a different name. The user defined function name will auto populate in the functions list.

Miscellaneous

The Miscellaneous section in FTC block programming serves as a toolbox for various functions that don't neatly fit into other categories. The blocks range from comment, format numbers, to how to handle null value.

Comment

Comments serve as explanatory notes within your code. They have no impact on the Robot, no on the telemetry.

formatNumber

Rounds a number to a certain precision and returns the rounded number as text. In the example, the block returns text “3.14”.

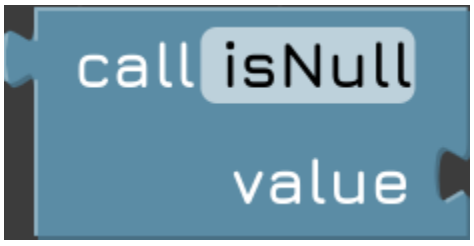
Key points:

- Returns a text, instead of a numeric value.
- Padded with 0 if necessary. For example, when 3.14 is rounded to precision of 3, “3.140” is returned.

null

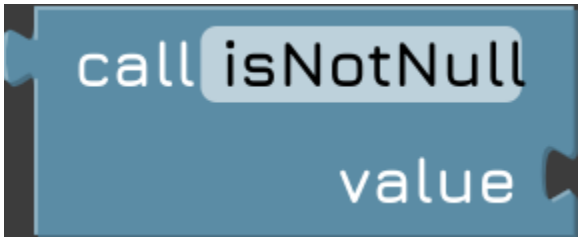
null. It represents the absence of a value or object.

isNull



Returns true if the given value is null. Returns false if the given value is not null.

isNotNull



Returns true if the given value is not null. Returns false if the given value is null.

Basic Lessons

Drivetrain

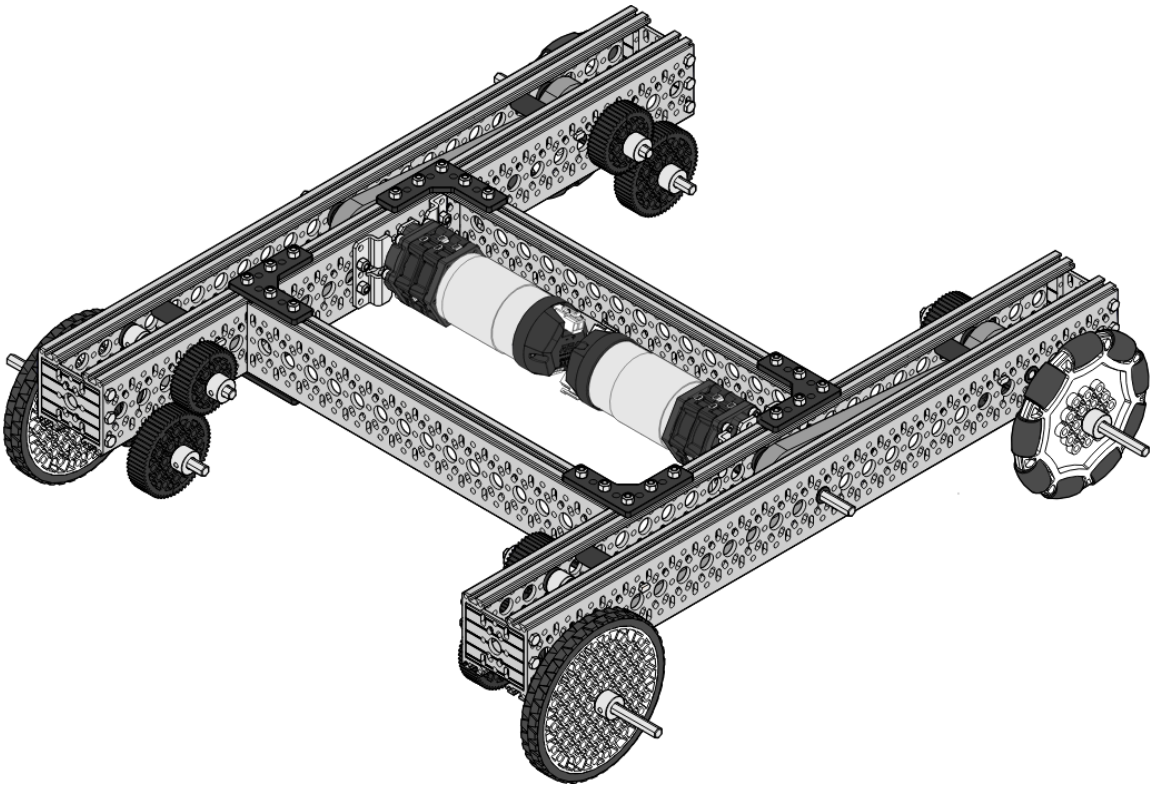
Drivetrains are the first steps in every engineer's journey. They refer to any system that allows a robot to move around the field. Learning to control these systems both autonomously and manually is a crucial first step. When building a robot, the two most common types of drivetrains used are 2 motor and 4 motor drives.

2 motor drives come in a variety of forms. They use 2 motors, typically one to power each side of the robot, for locomotion.
They have several advantages:

- Simplicity: Easier to build and program compared to more complex drivetrains.
- Cost-effective: Requires fewer components, reducing overall cost.
- Reliability: Fewer components generally mean fewer potential points of failure.
- Power: Can deliver significant power and torque for specific tasks.
- Suitable for beginners**:** Ideal for teams new to FTC robotics.

But they also have several disadvantages:

- Limited maneuverability: Difficulty in performing precise turns and movements compared to more complex drivetrains.
- Lower power output: Generally less powerful compared to 4-motor drivetrains, especially for heavier robots or demanding tasks.
- Increased wear and tear: Due to higher load on each motor, components might degrade faster.



The 4 motor drive is the predominant drivetrain used in FTC and the drivetrain type for both of the test robots. They use 4 motors, typically one for each wheel, for locomotion.
Their advantages are:

- Increased power: More motors mean more torque and speed.
- Improved traction: Better weight distribution and grip on various surfaces.

- Enhanced maneuverability: Greater control and ability to execute complex movements.
- Versatility: Can accommodate different drivetrain configurations (tank, mecanum, etc.).
- Better performance in challenging conditions: Handles rough terrain and obstacles more effectively.

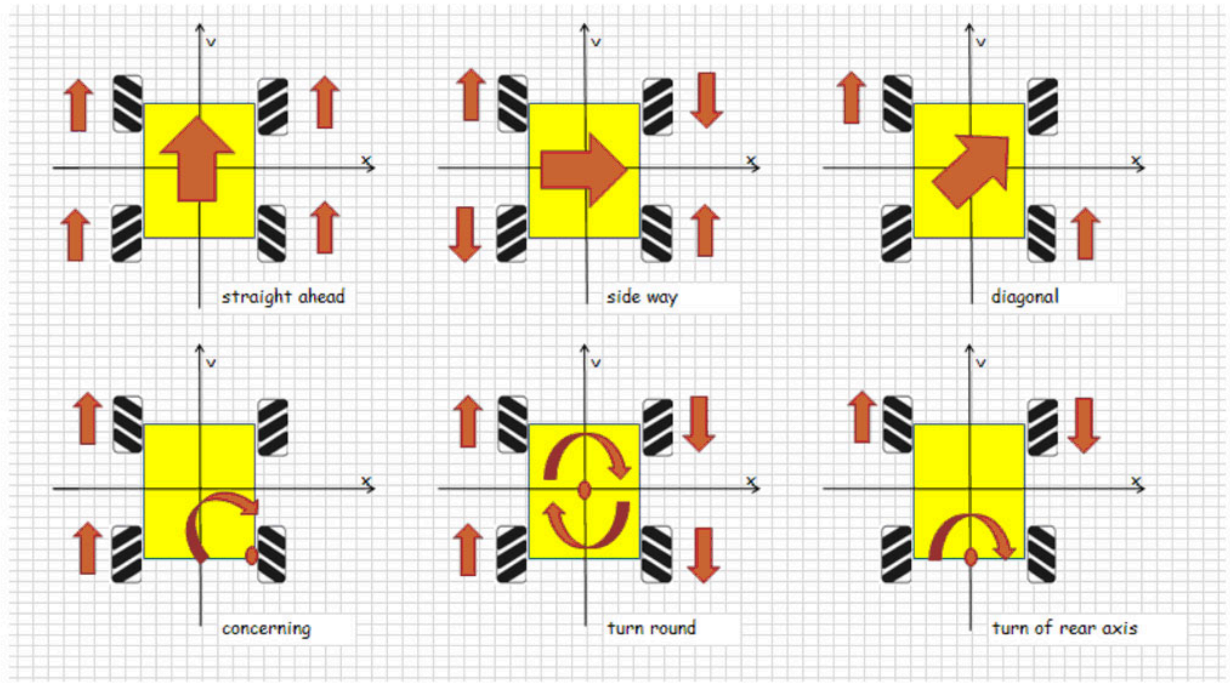
Their disadvantages are:

- Increased complexity: More components and wiring, leading to potential issues.
- Higher cost: Requires additional motors and components.
- Increased weight: Can impact robot performance.
- Advanced programming: Requires more sophisticated control algorithms.
- More components to inspect and maintain: With additional motors, gears, and wiring, there are more parts that can potentially fail or malfunction.

In short, 4 motor drives offer superior performance compared to 2 motor drives but require more complicated control methods and maintenance.

4 wheel drives also offer the unique ability to use Mecanum Wheels. By installing the wheels in an X shape, the robot can strafe, or move left and right without turning. This is due to force cancellations by rotating wheels in certain directions, causing a net movement to the overall drivetrain.

Here is a diagram showing how different motor movements can cause different movements.

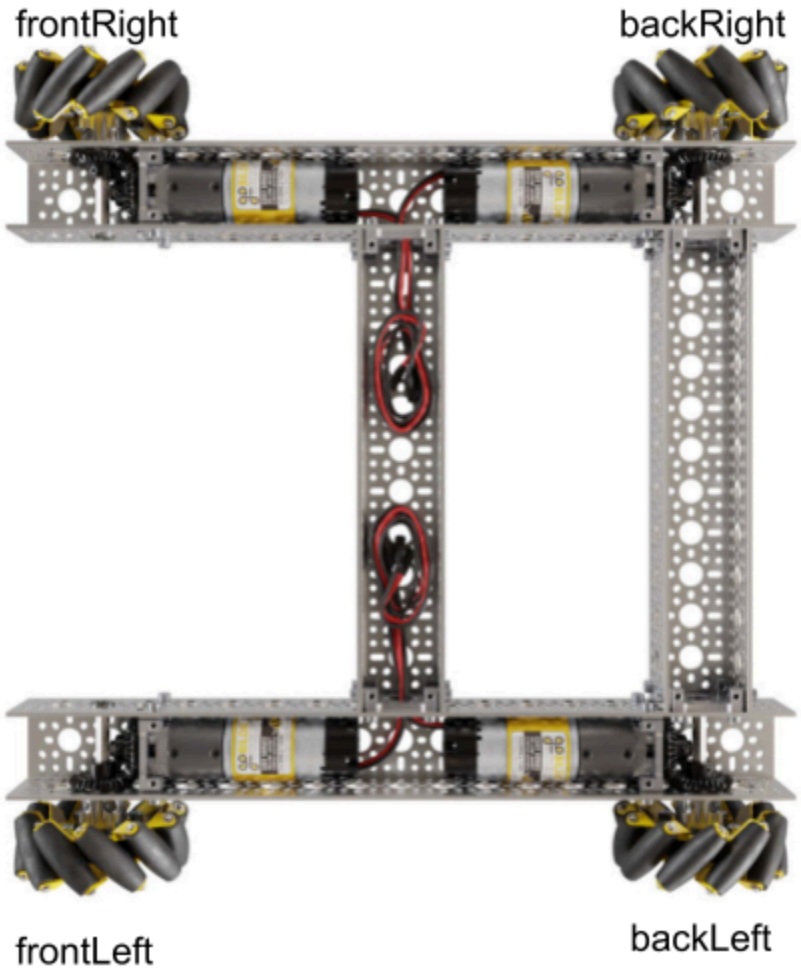


All VRS simulated robots can use this feature.

Programming

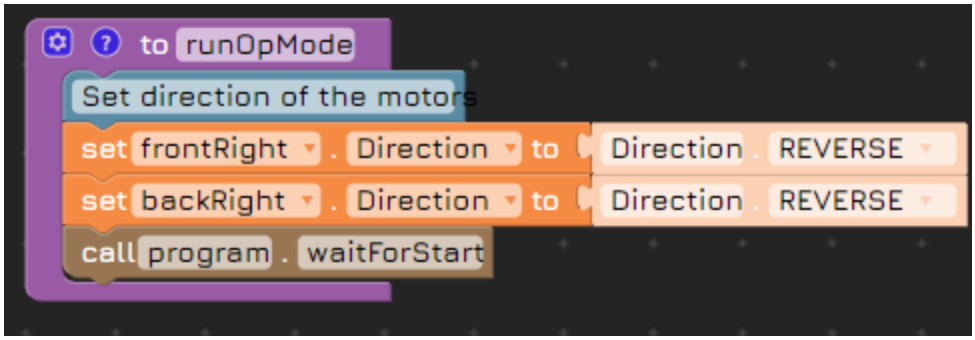
I will be teaching how to program a 4 motor drivetrain as it is the drivetrain all VRS simulated robots use. If you want to learn to use a 2 motor drivetrain, just follow the guide but limit yourself to only using 2 wheels.

Here is a basic diagram of how to motors are named:



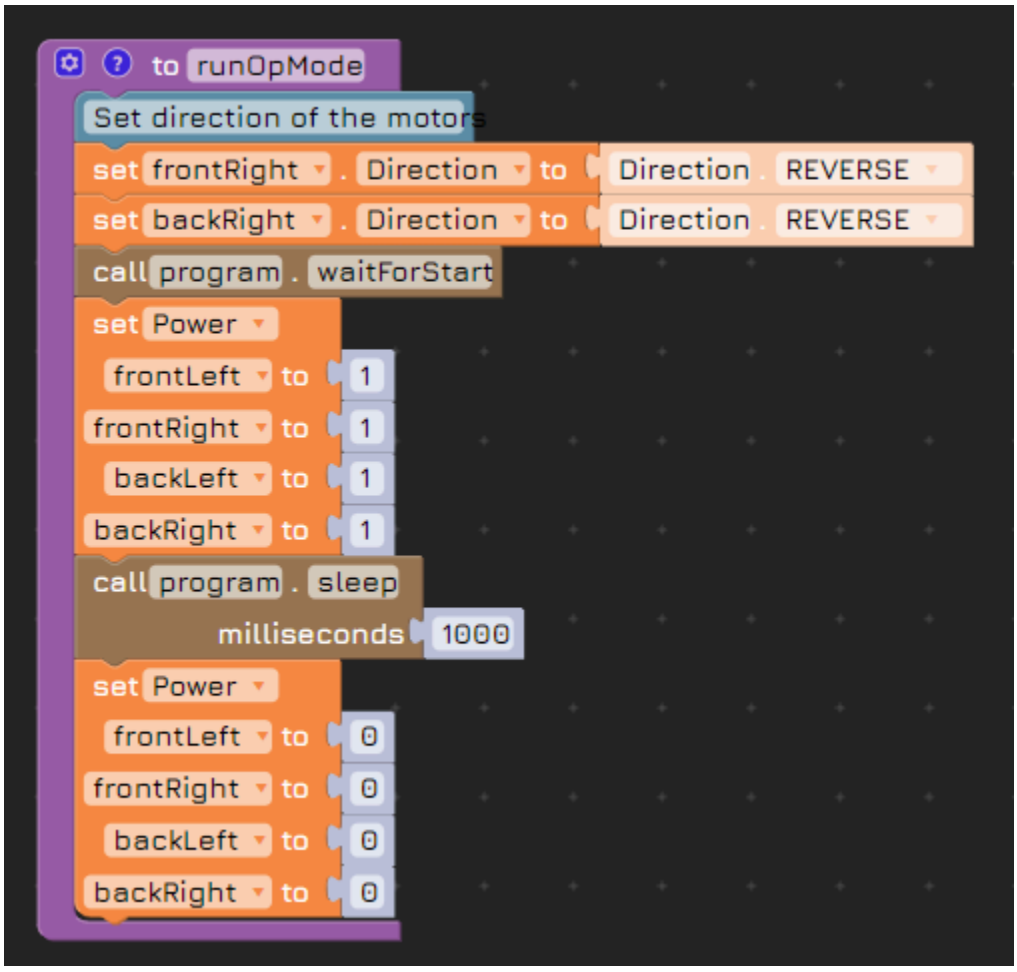
Before starting to move the robot, the direction of the motors should be first set. Motors can be set to either “FORWARD” or “REVERSE”. All motors are set by default to “FORWARD”, which means they turn counter-clockwise when given a positive power signal. When a motor is set to “REVERSE”, the motor now turns clockwise when given a positive power signal.

In our robots, the right motors (frontRight, backRight) should be set to reverse so that the robot moves forward when given a positive power signal. Although it might be arduous to have to set the direction of each motor, it makes programming the rest of the robot significantly simpler!



Autonomous:

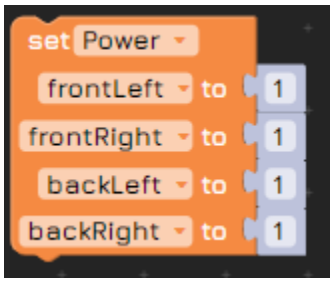
The most common way to control the robot autonomously is to simply set each motor's power, sleep the program for some time, and then cut the power to each motor.



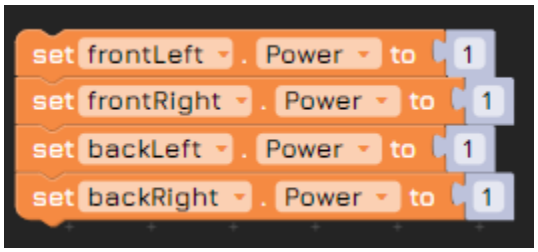
But what does this mean?

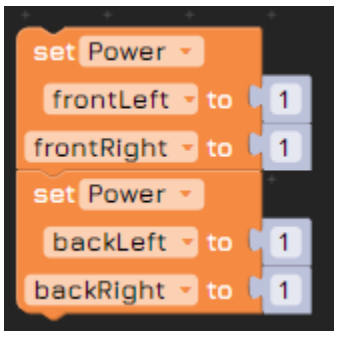
Let me break it down.

1. Setting the power



For the lesson I am using a quad block for ease of understanding. You could write the same code using 4 set power blocks or 2 dual set power blocks:

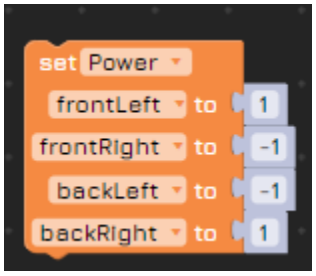




What this code does is that it tells the motors what direction to move and how fast to move it. If the power is positive, the motor moves counter-clockwise. If the power is negative, the motor moves

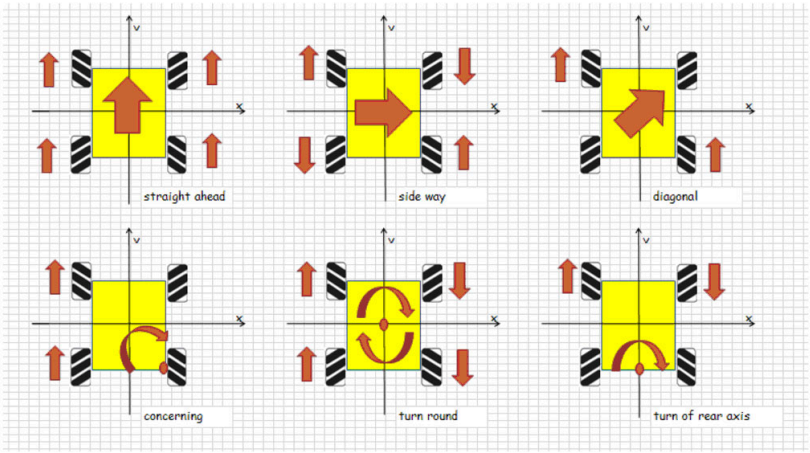
counter-clockwise. The motor power ranges from -1 to 1. The greater the magnitude of the motor, the faster it goes! The power doesn't have to be a whole number either! It can be a decimal like 0.1 or -0.7.

Figuring out the correct power values to make the robot move how you want to might be a bit confusing but thankfully, due to setting the motor direction from earlier, a simple rule is positive makes the robot go forward, negative makes it go in backwards. By changing how the power values like this:



,

if you will back to the Mecanum Wheel chart:

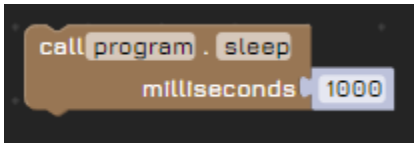


, the robot will strafe right.

If you want to move the robot in other ways, continue to reference the chart. Remember, for a precise strafe, the magnitude of the motors' power should all be the same. An up arrow represents a positive power and a down arrow represents a negative power.

Feel free to play around with the power though to make the robot move in interesting ways.

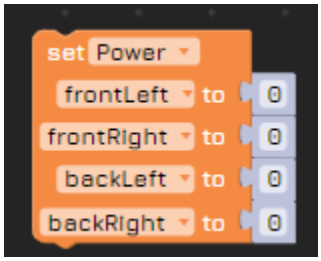
2. Sleeping the program



By calling this block, the program doesn't execute for the specified amount of milliseconds. This may seem dumb but it has an important purpose: it gives the motor direction in step 1 time to move the robot. Just setting the power of a motor doesn't make the robot move! Instead it defines the motor's behavior. To let the motor move the robot, you need to give it time, literally! Sleeping the program will allow the robot time to move around.

3. Stopping the robot.

To stop the robot, set the power of all motors to 0.



By completely cutting off the power the motor receives, it will completely stop.

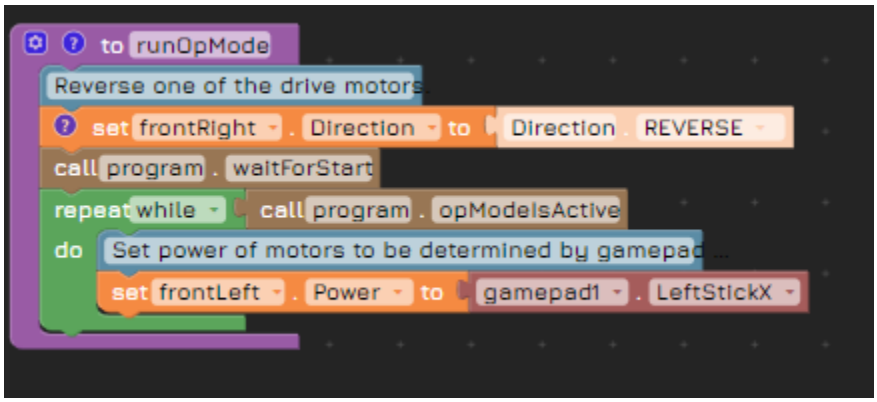
Although the robot will automatically stop moving when it has fully executed its code, setting the power to 0 is a helpful way to ensure the robot moves precisely how it was intended.

Typically steps 1 and 2 are repeated several times to create more complex movements before stopping the robot. Step 3 may be incorporated, however, to allow other parts of the robot, like a claw or slide, to move before resuming locomotion.

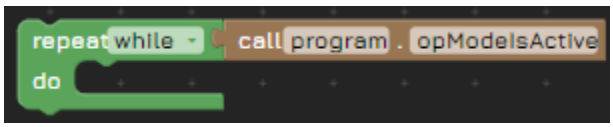
Manual:

You get to set the controls to whatever you want! Just configure the controls in whatever way feels most intuitive!

Most of your code will be generally structured like this:



1. After the program has started, you repeat the setting the power for the motors as long as opMode, the designated period to control the robot, is active.



2. Map how you want the power of each motor to be set to.

   The basic sections are:

set [MotorName].Power to which determines which motor is being affected.

Then you either set the motor to a gamepad control, like LeftStickX or RightStickY directly:

or

add a negative marker to reverse the gamepad's effect on the motor.

Simply repeat the set command until you can control all the motors you want!

Don't forget that the motor direction can be reversed. This has the same effect as simply putting the negative marker on the control. A negative marker on a reversed motor will cancel out, causing the motor to behave as if it was just directly controlled by the gamepad.

Feel free to experiment with what you have learned and, most importantly, don't forget to have fun!

IMU

IMU, or inertial measurement unit, is an incredible device that measures the robot's acceleration, angular velocity, and orientation! It works by tracking changes in motion over time. By integrating acceleration data, you can calculate

velocity and position. Similarly, integrating angular velocity data gives you orientation.

Located in the FTC Control Hub,

IMUs are typically used for autonomous navigation. In combination with other sensors, the robot can autonomously and efficiently traverse across the gamefield, using the IMU to help guide itself and identify where the robot is located. It's also used to automatically stabilize the robot, making piloting the robot easier.

This lesson will teach you the fundamentals on how to receive data from an IMU. If you want to explore how to use this information, please check out the advanced guide where I will show you a simplified version of Roadrunner, a common program used in FTC that uses the IMU to quickly traverse the field autonomously.

IMU programming comes in two parts: initialization and recording.

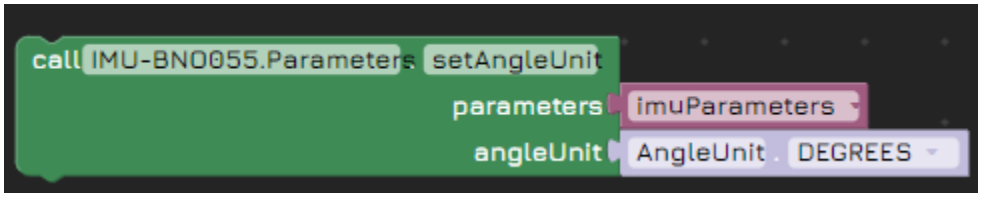
Initialization

The first step is defining the units the IMU will use. To stick with default parameters, this step can be skipped but it is recommended to define the units the IMU will use so that it can fit its intended need and make calculations easier.

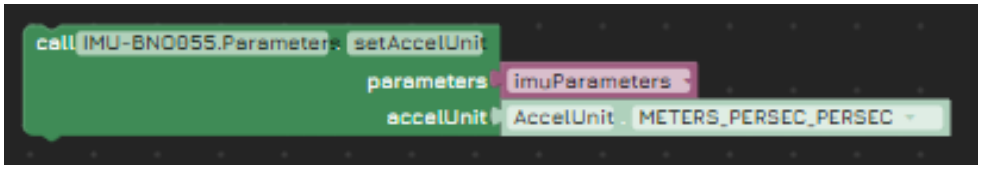
1. set the variable imuParameters to new IMU-BNO055.Parameters

   This will tell the program to set the variable to a set of parameters object for the IMU. This is necessary as the imu cannot be set to anything other than IMU-BNO055.Parameters objects.
2. call IMU-BNO055.parameters to set what measurement you want.
   There are several options to set the IMU parameters to. For example some common ones are:
   setAngleUnit will set the angular units the IMU uses.

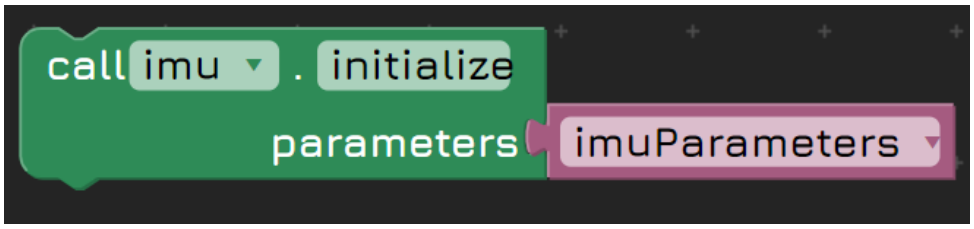
   setAccelUnit which will set the acceleration units the IMU uses.

   The parameters is the IMU-BNO055.Parameters object you want to change. The unit defines the new unit the object will use.

More options are listed in the IMU-BNO055.Parameters section under Sensors.

3. Initialize the imu with the new parameters.

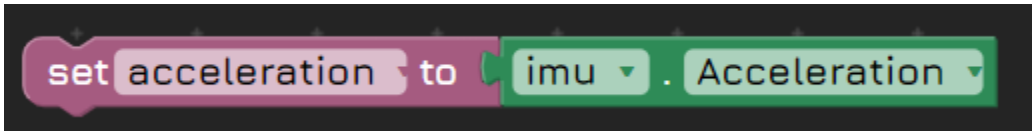


Finally, this will set the IMU's parameters to the variable containing a IMU-BNO055.Parameters object, changing the units the IMU uses.

Recording

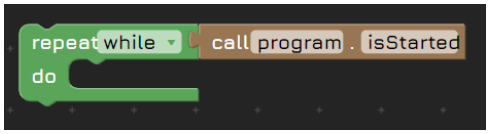
You can access the IMU data recording in the IMU using the get___ block like getAngularVelocity or getAngularOrientation or the imu.___ blocks with the wanted parameter listed afterwards. The options are listed in the IMU-BNO055 section under Sensors.

Oftentimes, you will set the parameters you want to a variable like

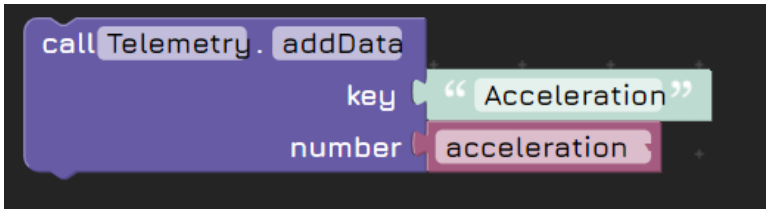


which will make it easier for the coder (that's you!) to understand what parameter refers to what.

If you want to see what the data is like in real time then, in the program's main loop like with

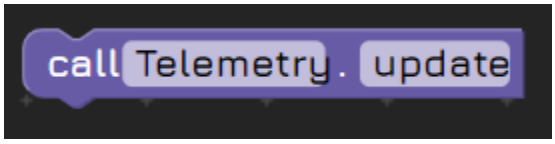


which will run as long as the game has begun, call the Telemetry to add the data you want:



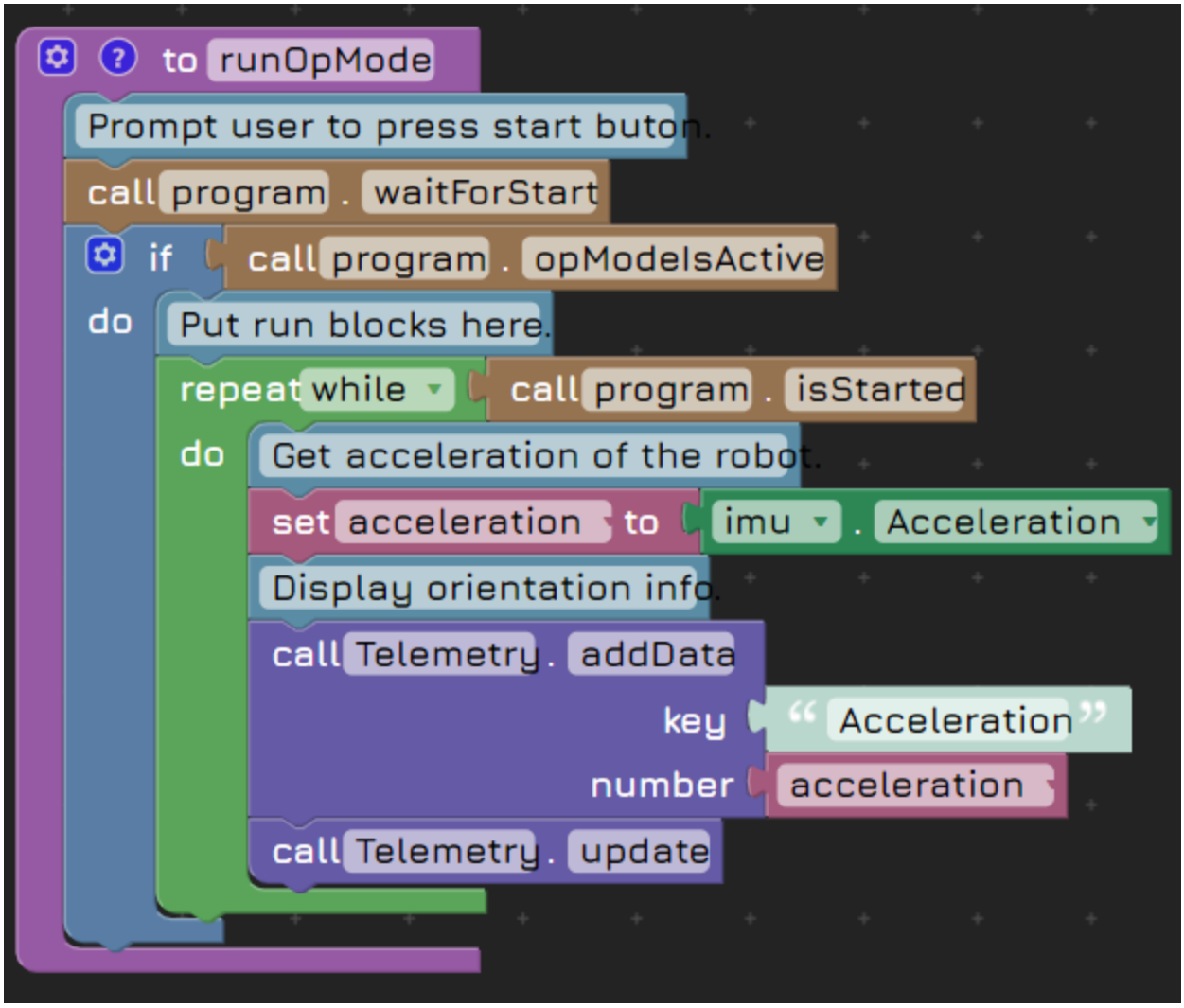
. The key is a helpful description for the displayed number, making it easier to understand which number refers to what. The number is set to any number or variable that represents a number.

Don’t forget to call Telemetry.Update at the end of the loop to update the Telemetry!



You should see the resulting telemetry displayed on the upper right hand corner:

Putting it all together, the code for this example looks like:



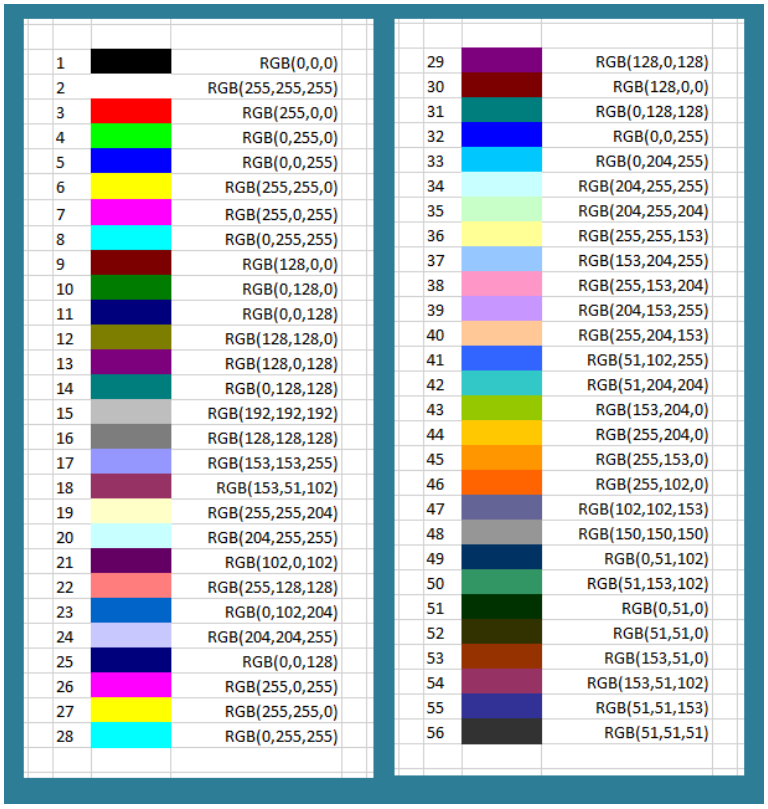
Now you can access the IMU! If you want to see how the IMU can work in action then check out the advanced guide for more!

Color Sensor

The REV Color/Range sensor is a very versatile device used in robotics, particularly in FTC. It combines both color sensing and proximity detection capabilities.

This section will cover the color detection aspect of the sensor.

The color sensor works by measuring the intensity of red, green, and blue light intensity (RGB). Using a combination of these colors, the entire visible color spectrum can be made.



1		RGB(0,0,0)	29		RGB(128,0,128)
2		RGB(255,255,255)	30		RGB(128,0,0)
3		RGB(255,0,0)	31		RGB(0,128,128)
4		RGB(0,255,0)	32		RGB(0,0,255)
5		RGB(0,0,255)	33		RGB(0,204,255)
6		RGB(255,255,0)	34		RGB(204,255,255)
7		RGB(255,0,255)	35		RGB(204,255,204)
8		RGB(0,255,255)	36		RGB(255,255,153)
9		RGB(128,0,0)	37		RGB(153,204,255)
10		RGB(0,128,0)	38		RGB(255,153,204)
11		RGB(0,0,128)	39		RGB(204,153,255)
12		RGB(128,128,0)	40		RGB(255,204,153)
13		RGB(128,0,128)	41		RGB(51,102,255)
14		RGB(0,128,128)	42		RGB(51,204,204)
15		RGB(192,192,192)	43		RGB(153,204,0)
16		RGB(128,128,128)	44		RGB(255,204,0)
17		RGB(153,153,255)	45		RGB(255,153,0)
18		RGB(153,51,102)	46		RGB(255,102,0)
19		RGB(255,255,204)	47		RGB(102,102,153)
20		RGB(204,255,255)	48		RGB(150,150,150)
21		RGB(102,0,102)	49		RGB(0,51,102)
22		RGB(255,128,128)	50		RGB(51,153,102)
23		RGB(0,102,204)	51		RGB(0,51,0)
24		RGB(204,204,255)	52		RGB(51,51,0)
25		RGB(0,0,128)	53		RGB(153,51,0)
26		RGB(255,0,255)	54		RGB(153,51,102)
27		RGB(255,255,0)	55		RGB(51,51,153)
28		RGB(0,255,255)	56		RGB(51,51,51)

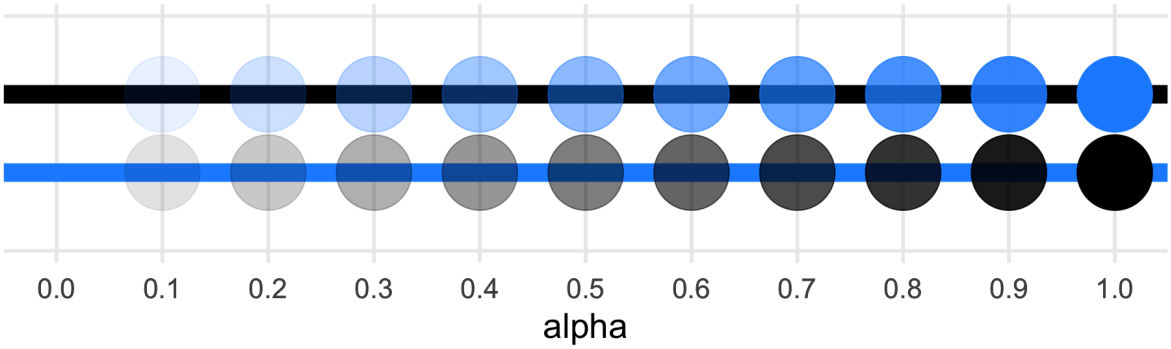
The sensitivity of this measurement is called the gain of a color. It can be set using

Basically, it makes lighter colors lighter and darker colors darker. A higher gain is more useful when there is little light to differentiate between different colors. A low gain is used when there is high amounts of light to help normalize the values.

Different FTC fields will have different amounts of light which can cause the color detector to become uncalibrated. Using the LightDetected block

you can see how intense the light is. You can use the value returned to adjust the gain to allow for optimal color sensing.

Another way the colors are recorded are in Argb format. This is similar to RGB but there is a 4th value alpha (A), which represents the opacity or transparency of a color. A higher alpha value means the colors are more opaque.



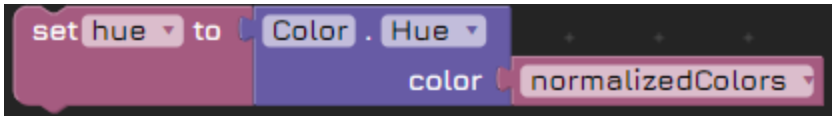
You can also use

to get normalized RGBA values instead of raw inputs. This will expand or contrast the color values into a set range, allowing for more predictable color input values.

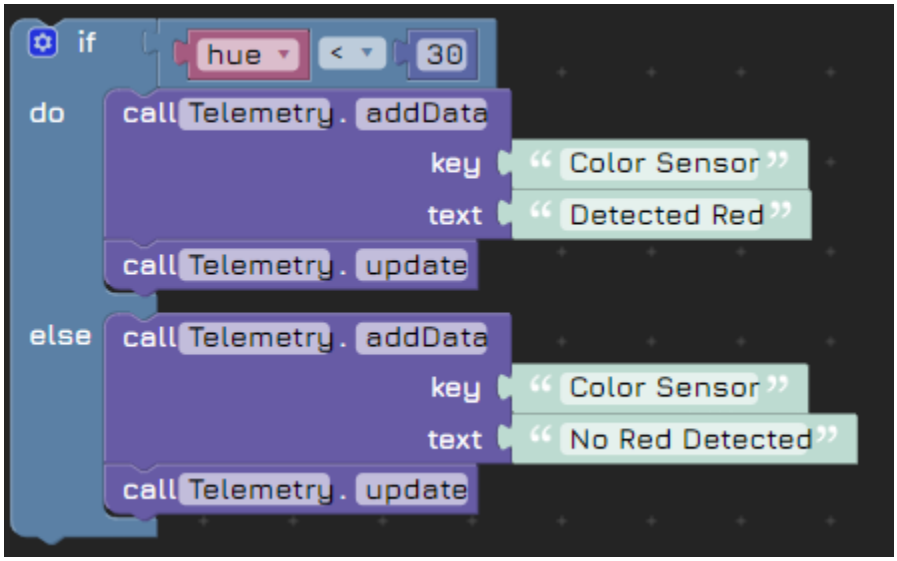
So now you have these RBG or RBGA values. What do you do with them? Well, in their raw state the values are useless. To get meaningful information out of them you have to interpret parts of the data. You do this by setting a variable to your desired data

and then using a color block to get specific values like saturation, hue, and value. You can then use these values in logic statement to determine if the desired color has been detected.

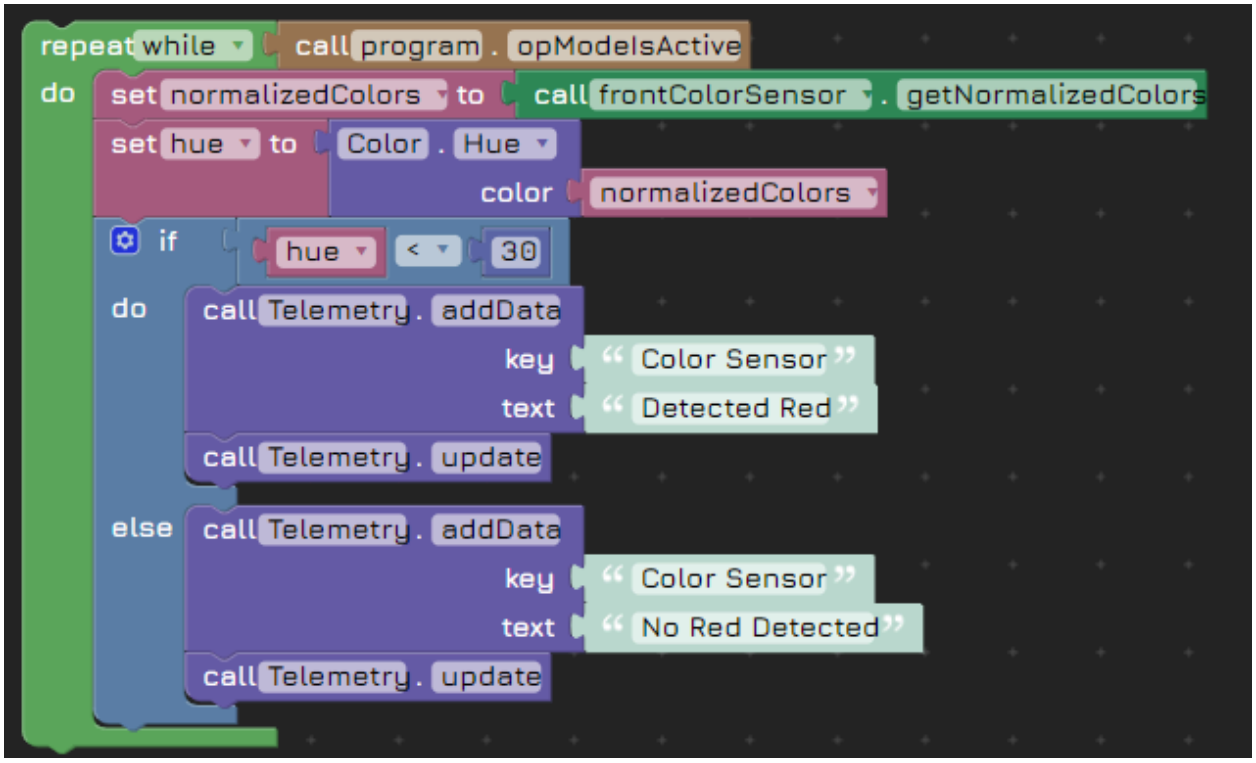
One way is to use hue. Here we set the variable “hue” to the hue to the normalized colors.



You can then use an if statement to check if the color red has been found and display it on the telemetry.



Finally put the code within a loop for continuous monitoring and you get the following red color detection code.



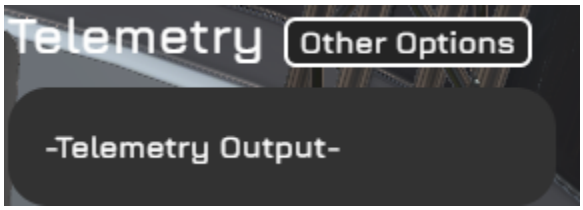
To find out more about which color codes correspond to which color, please use google where tools are available to help you find out.

RGB Color chart:
https://www.rapidtables.com/web/color/RGB_Color.html
RGBA Color Chart:
https://rgbacolorpicker.com/
Color Format Converter:
https://www.myfixguide.com/color-converter/

Color sensors are invaluable for object detection in FTC, helping to differentiate otherwise equally shaped or hard to find objects. It is also used in autonomous navigation where color lines can help let your robot know its position or indicate when it should stop. The options are limitless!

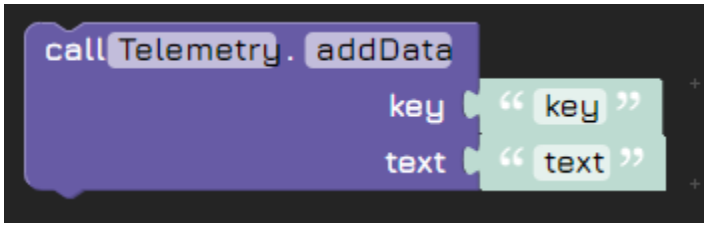
Telemetry

Have you wanted to see how your robot perceives the world? Well then, telemetry is for you! Telemetry refers to the process of collecting and transmitting data from the robot to a driver station or computer. This data can include sensor readings (like color, distance, or touch), motor power, and other relevant information. Telemetry shows up in the top right side of the robot testing screen.



Telemetry readings are an important part in debugging and troubleshooting problems as you can easily tell if there is a mismatch between the robot and your expectations.

Using telemetry is simple. Whenever you want to read something in your code use the addData block:

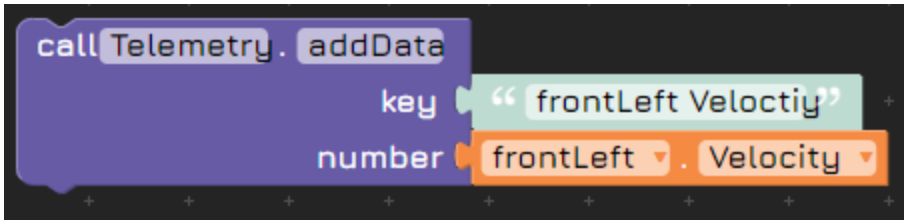


.

This block has several variants but in essence the key is the label for the lower value. It's important to denote the key as it is your primary method of figuring out what the telemetry value is.

The second, lower value can be anything: numbers, text, etc, it just depends on the block.

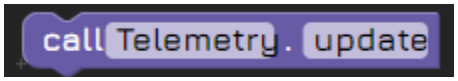
For example:



Will tell you the velocity of the frontLeft motor.

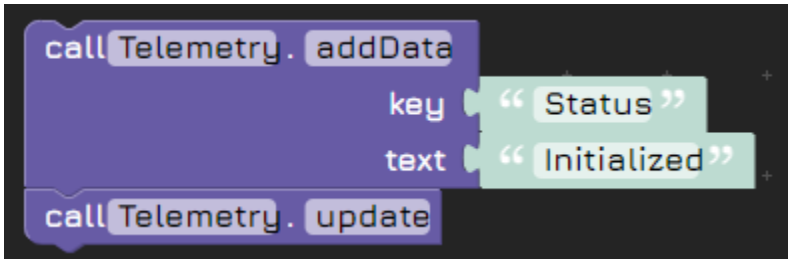
Repeat the addData block for each data value you want to see.

Then call the update block to have the key and value displayed on the telemetry output screen.



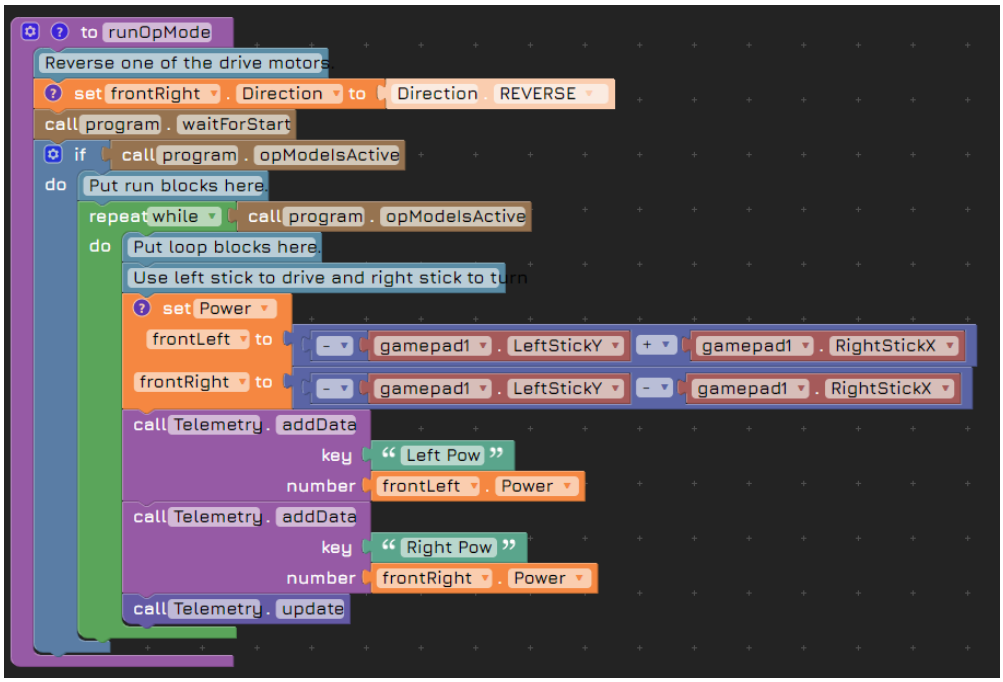
So what is happening? Well the addData block adds data to the telemetry. The repeated addData blocks will continue to add more data to the telemetry. But why does the data now show up immediately in the output then? Well, the output shown is a snapshot of a previous version of the telemetry. When you update the telemetry again, the VRS will convert the current telemetry into readable text and display that. Afterwards, it will clear out the telemetry so that new data can be added.

This single display form of telemetry is useful for letting you know what part of the program you are on. It is common to add a

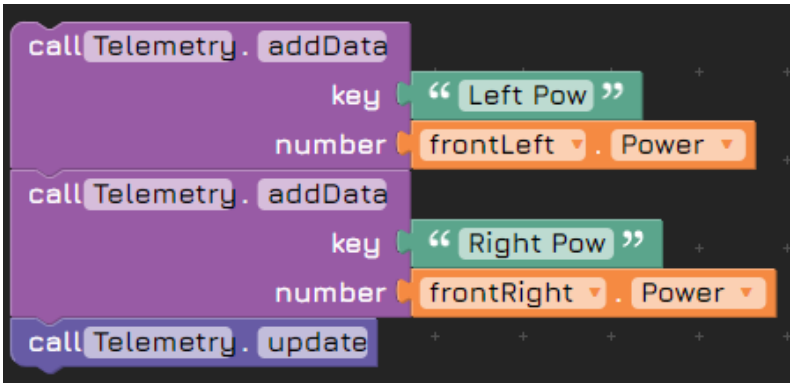


while waiting for the program to start in order to let the driver know that the code has been successfully initialized. Such text is invaluable in helping the driver to quickly check if the code is working or not.

Another common way to use telemetry is in a loop. In the example:



The telemetry



will provide a constant update on the power of each motor. This updating information is great for real time feedback of the robot, helping to quickly identify errors or flaws in code.

Although telemetry may not directly affect the robot, it is nevertheless an important tool at a programmer’s disposal. Good telemetry can greatly speed up coding and provide feedback for drivers, resulting in more points scored. This often is the difference between a good robot and a great robot.

Range Sensor

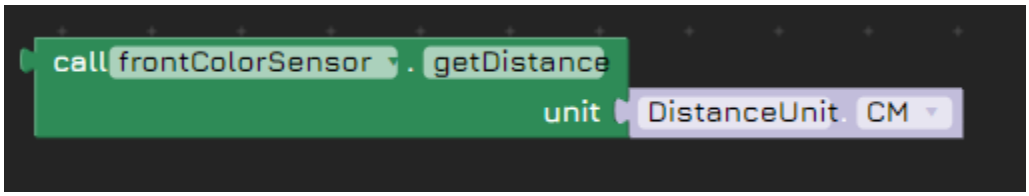
The REV Color/Range sensor is a very versatile device used in robotics, particularly in FTC. It combines both color sensing and proximity detection capabilities.

This section will cover the proximity detection aspect of the sensor. The distance sensor works by bouncing infrared light off an object and using the time between sending and receiving the light to calculate the distance between the sensor and the object.

The distance sensor is useful for both object detection and obstacle avoidance. The distance sensor can indicate if an obstacle is close or not to the robot, allowing for the robot to react accordingly. The range sensor can also be

attached to the intake of the robot, allowing the robot to sense if an object has been collected or not.

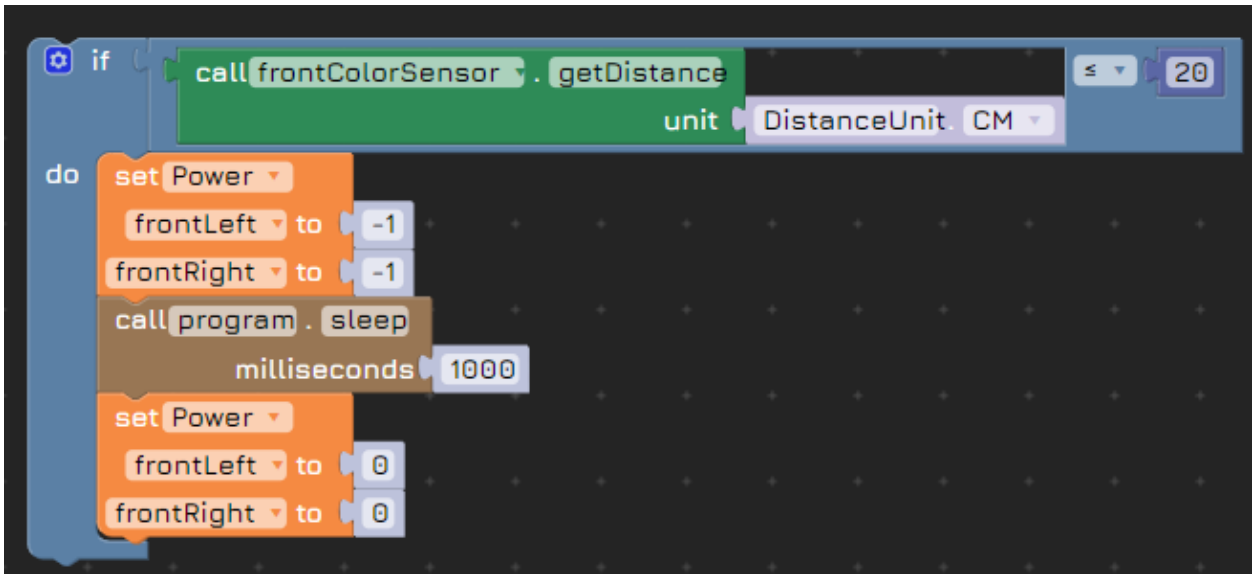
Using the sensor is easy. Simply use the block getDistance



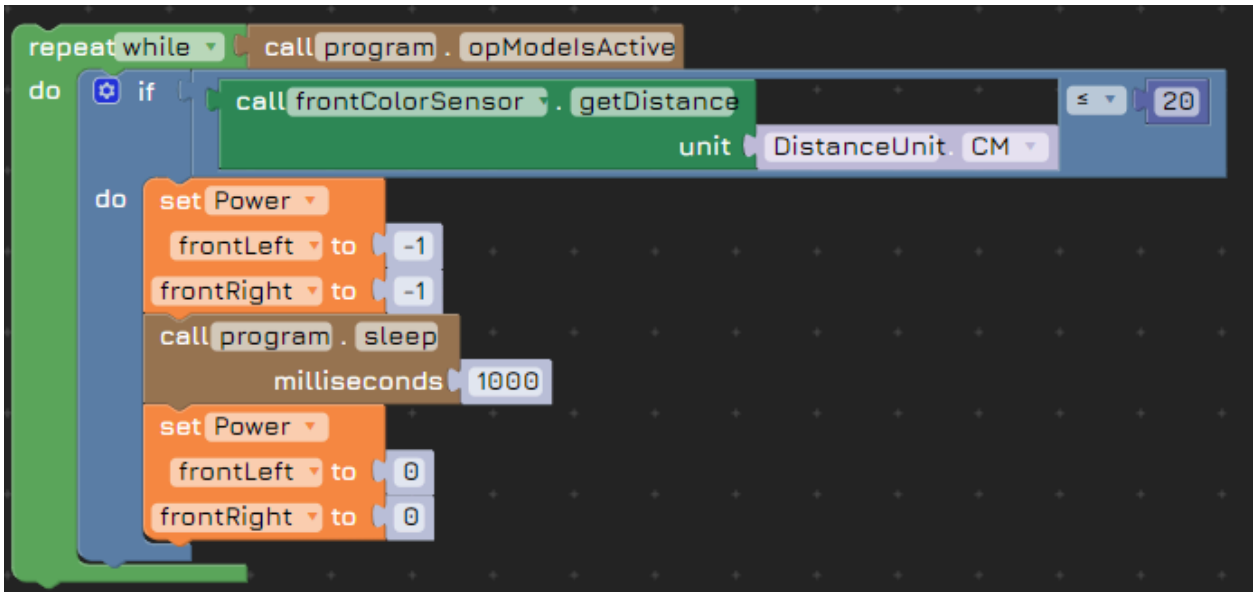
to determine the distance of the closest object in front of the distance sensor. You can select the units the distance is recorded in.

This is often used in logic statements to check if the robot has encountered an obstacle.

For example, if the range sensor detects if there is an object within 20cm from the front of the robot, the robot will move backwards for 1 second.



Putting this within a for loop for continuous detection



You get a simple anti collision algorithm!

The range sensor can prove to have numerous uses if you are creative! Several examples are present in this book. Feel free to experiment.

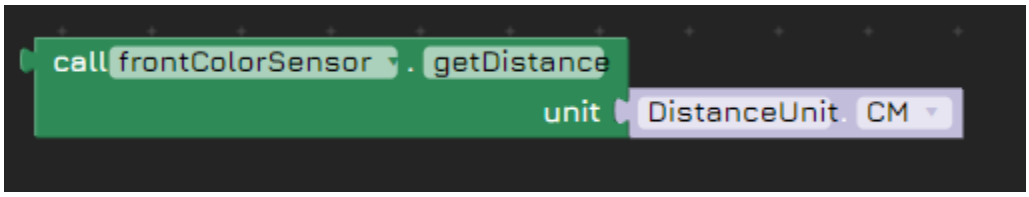
Touch Sensor

The FTC touch sensor is simple but effective. It detects whether the sensor has been pressed or not. When it button has been depressed a circuit is completed, registering it as a touch.

The range sensor is used for obstacle avoidance and as a limit switch. When attached to the outside of the robot, when the touch sensor detects a touch, you know the robot has collided with something and should reverse. Limit switches are devices used to prevent robot parts from moving beyond a certain point. For example, if your robot has a robotic arm, you don’t want the robot arm dragging

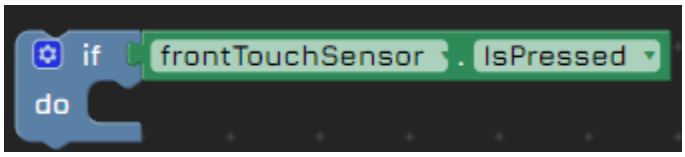
on the ground. To ensure that doesn't happen you install a touch sensor at certain joints. If the arm has moved below a certain point, it will touch the touch sensor and signal that the arm has been depressed too far. An automated response can then be sent to raise the arm back up to a safe level.

Programming using the sensor is easy. Simply use the block getDistance

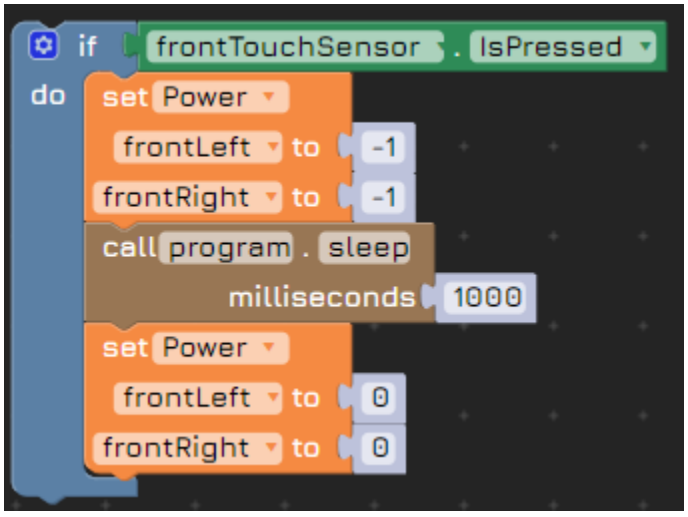


to determine the distance of the closest object in front of the distance sensor. You can select the units the distance is recorded in.

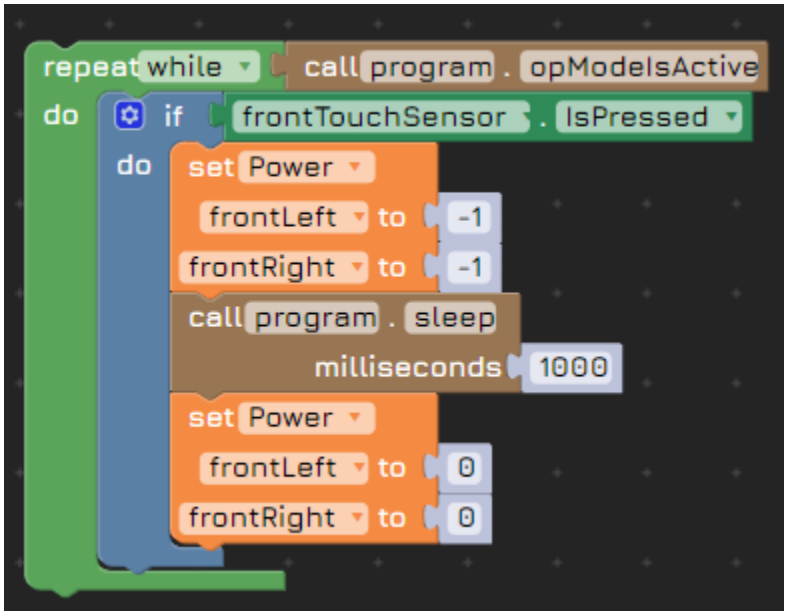
Here is how the touch sensor can be used for touch avoidance.
Create a simple logic statement:



Then when collision is detected, reverse the robot by reverse motor power for 1 second.



Put this within a for loop for continuous detection.



And there you have an anti collision algorithm!

Touch sensors can prove to be useful in certain situations. Be creative!

Servo

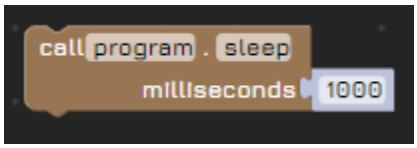
A servo is a type of motor that can be precisely controlled to rotate to a specific position. Unlike traditional DC motors, servos have built-in feedback mechanisms that allow them to maintain their desired position. Although they have a limited rotational range, they are more precise than CR Servos and have higher holding torques.

Programming these servos are similar to setting the target position on a DC motor. There are only 2 steps, setting the position and sleeping the program.

1. Setting the position

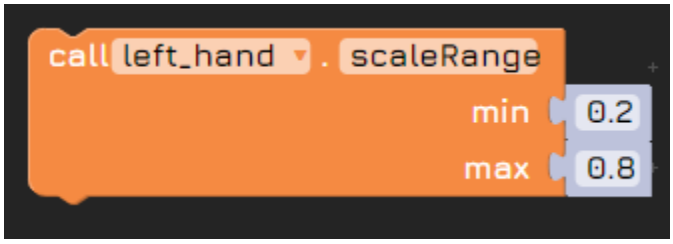
What this code does is that it tells the servo what position to move to. Based on the encoder's readings the servo will automatically rotate until it has reached the desired position.

2. Sleeping the program



By calling this block, the program doesn't execute for the specified amount of milliseconds. This may seem dumb but it has an important purpose: it gives the servo in step 1 time to move the desired position. Just setting the target position of a servo doesn't make the robot move! Instead it defines the servo's behavior. To let the servo move the robot, you need to give it time, literally! Sleeping the program will allow the robot time to move around. When the servo has arrived at the target position, it will automatically stop moving so no further action is necessary.

Often servos will need to limit the range at which they operate. Some robotics arms, for example, cannot depress below a certain point since the servo then could no longer lift the arm back up without overloading it. Other tasks may only operate within a precise range. To prevent such incidents from occurring you can use the scaleRange block to limit the servo's range.



This will prevent the robot from moving past a minimum and maximum range so you can rest easy when programming the robot.

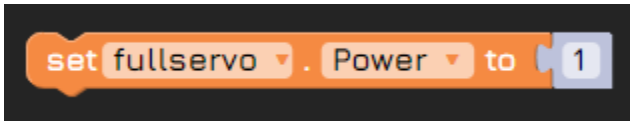
Servos play an important part in controlling more precise devices unfit for motors. Their uniquely high precision, holding torque, and lightweightness make them a fantastic way to power grippers, rotate a turret, or power any number of devices.

CRServo

CR Servo, or Continuous Rotation Servo, is a servo that can rotate indefinitely in either direction. CR Servos typically have higher torque and can achieve a wider range of speeds compared to standard servos. They are a versatile component in robotics, providing continuous rotation and power for various applications.

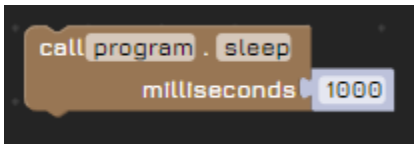
Programming these servos are similar to setting the power on a DC motor. There are 3 steps, setting the power, sleeping the program, and then turning off the power.

1. Setting the power



What this code does is that it tells the CR servo what direction to move and how fast to move it. If the power is positive, the CR servo moves counter-clockwise. If the power is negative, the CR servo moves counter-clockwise. The CR servo power ranges from -1 to 1. The greater the magnitude of the CR servo, the faster it goes! The power doesn't have to be a whole number either! It can be a decimal like 0.1 or -0.7.

2. Sleeping the program



By calling this block, the program doesn't execute for the specified amount of milliseconds. This may seem dumb but it has an important purpose: it gives the CR servo direction in step 1 time to move the robot. Just setting the power of a CR servo doesn't make the robot move! Instead it defines the CR servo's behavior. To let the CR servo move the robot, you need to give it time, literally! Sleeping the program will allow the robot time to move around.

3. Stopping the robot.

To stop the robot, set the power of the CR servo to 0.

By completely cutting off the power the CR servo receives, it will completely stop.

Although the robot will automatically stop moving when it has fully executed its code, setting the power to 0 is a helpful way to ensure the robot moves precisely how it was intended.

Typically steps 1 and 2 are repeated several times to create more complex movements before stopping the robot. Step 3 may be incorporated, however, to allow other parts of the robot, like a claw or slide, to move before resuming locomotion.

You can also reverse the direction of the servo. Which will invert the power to the servo.

This is useful when multiple servos power the same mechanism like a robot arm but are attached facing each other. By reversing the direction to one of the servos, controlling the servo becomes much easier as the same power input will make both servos turn the same direction, synchronizing their movement. This can prevent mechanical failure and overloading the servo.

CR Servos have many uses in FTC. They are handy as a miniature motor, helping to power light and more compact devices.

DCMotor

DC motors are a fundamental component of FIRST Tech Challenge robots. They are used to provide power and movement to various parts of the robot, such as wheels, arms, and other mechanisms. These motors are direct current and brushed. DC Motors come in various sizes and power ratings built for different applications so be sure to select the perfect one for you.

Motors have two primary running methods, setting their power or target position. The most common method is setting the power the motor receives, controlling the motor to rotational speed and direction. The second is setting the target

position of the motor. This involves using an encoder to record the rotational positional of a motor and then rotating it unless it reaches the target position.

Here are their pros and cons:
setPower

Pros:

- Direct control: Provides immediate control over the motor's speed and direction.
- Real-time adjustments: Allows for dynamic adjustments based on sensor feedback or other conditions.

Cons:

- Requires continuous input: The motor needs constant power input to maintain its desired speed or direction.
- Less precise: Can be less precise for tasks requiring accurate positioning.
- Potential for overshoot: May overshoot the target position if not carefully controlled.

setTargetPosition:

Pros:

- Precise positioning: Ensures the motor moves to a specific target position.
- Closed-loop control: Uses feedback (e.g., from an encoder) to maintain the target position.
- Less prone to overshoot: Reduces the risk of overshooting the target.

Cons:

- Slower response time: May take longer to reach the target position compared to setPower.
- Requires additional hardware: Often requires encoders or other sensors for accurate position feedback.
- More complex programming: Can be more complex to implement due to the closed-loop control.

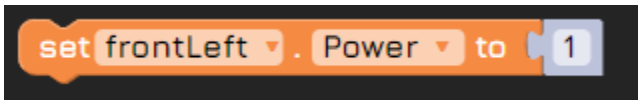
Generally setPower is more used for active control like moving a drivetrain. setTargetPosition is more suitable for tasks that require precise positioning like aligning a robot arm.

Lets see how each method is typically used:

setPower:

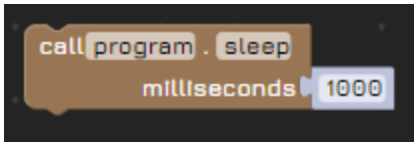
There are 3 steps, setting the power, sleeping the program, and then turning off the power.

1. Setting the power



What this code does is that it tells the motor what direction to move and how fast to move it. If the power is positive, the motor moves counter-clockwise. If the power is negative, the motor moves counter-clockwise. The motor power ranges from -1 to 1. The greater the magnitude of the motor, the faster it goes! The power doesn't have to be a whole number either! It can be a decimal like 0.1 or -0.7.

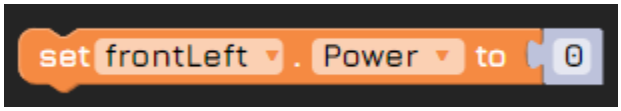
2. Sleeping the program



By calling this block, the program doesn't execute for the specified amount of milliseconds. This may seem dumb but it has an important purpose: it gives the motor direction in step 1 time to move the robot. Just setting the power of a motor doesn't make the robot move! Instead it defines the motor's behavior. To let the motor move the robot, you need to give it time, literally! Sleeping the program will allow the robot time to move around.

3. Stopping the robot.

   To stop the robot, set the power of the motor to 0.



   By completely cutting off the power the motor receives, it will completely stop.

   Although the robot will automatically stop moving when it has fully executed its code, setting the power to 0 is a helpful way to ensure the robot moves precisely how it was intended.

   Typically steps 1 and 2 are repeated several times to create more complex movements before stopping the robot. Step 3 may be incorporated, however, to allow other parts of the robot, like a claw or slide, to move before resuming locomotion.

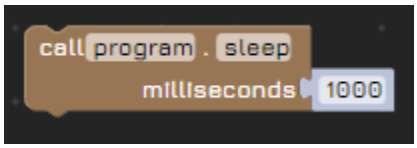
setTargetPosition:

Unlike setPower, there are only 2 steps, setting the target position and sleeping the program.

1. Setting the target position

What this code does is that it tells the motor what position to move to. Based on the encoder's readings the motor will automatically rotate until it has reached the desired position.

2. Sleeping the program



By calling this block, the program doesn't execute for the specified amount of milliseconds. This may seem dumb but it has an important purpose: it gives the motor in step 1 time to move the desired position. Just setting the target position of a motor doesn't make the robot move! Instead it defines the motor's behavior. To let the motor move the robot, you need to give it time, literally! Sleeping the program will allow the robot time to move around. When the motor has arrived at the target position, it will automatically stop moving so no further action is necessary.

Now that you have learned both ways of controlling a motor, try to experiment to find the right method for you. Remember to customize the approach based on the motor and its purpose to achieve optimal results.

Logic and Loops

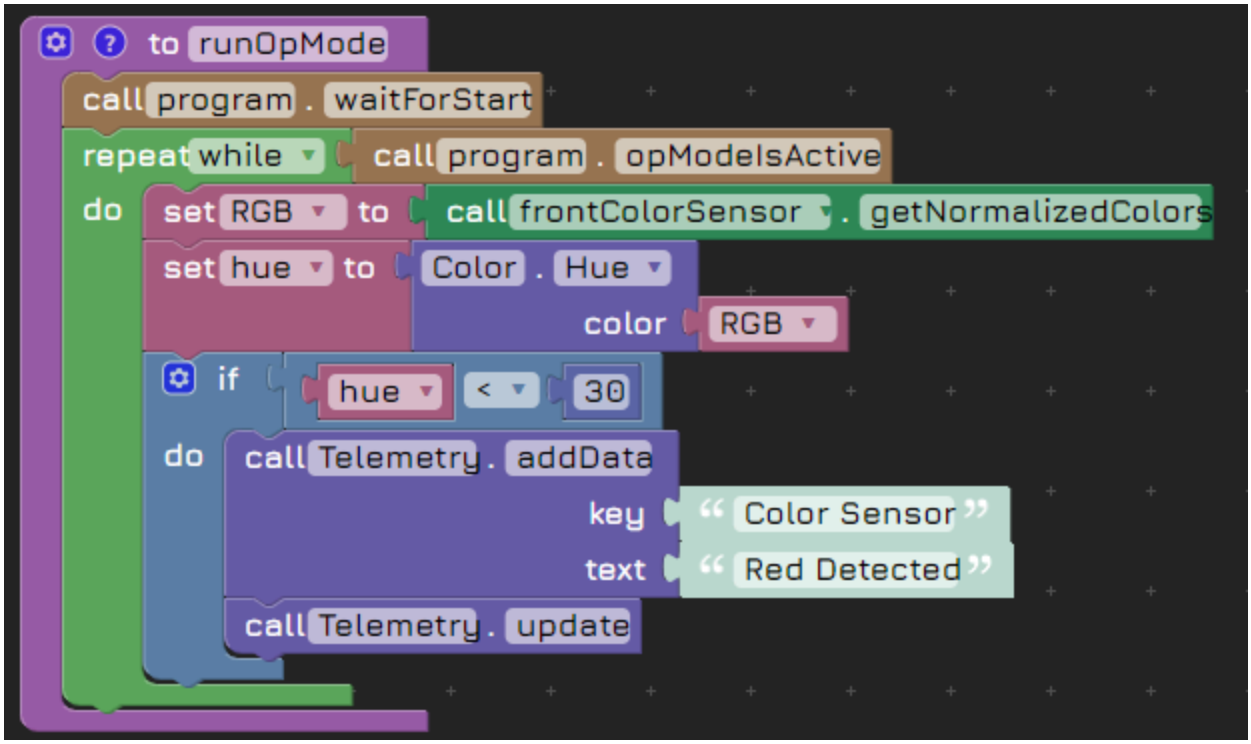
Logic and loops are vital for responsive code. Logic is most commonly seen with the if statement, they allow the program to make decisions based on conditions. They generally follow the pattern of if something is true then do this otherwise do this. These if statements can be chained together to then form complex logic structures.

Loops are most prevalent with the repeat loop. They simply repeat some chuck of code several times until a condition is no longer true. This is useful for not only saving yourself from copy and pasting the same steps over and over but also making code more readable. This helps allow code to become smaller and coding faster.

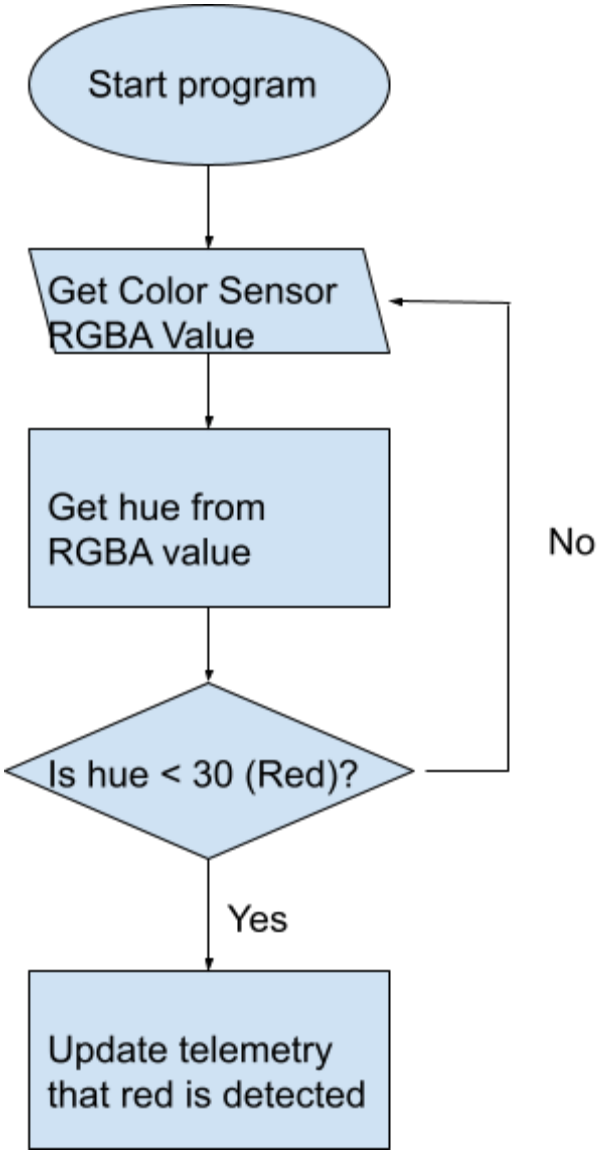
By combining loops and logic, complex code can be created.

All of this is sounding a little confusing. Luckily, programmers have an easy way of visualizing it: flowcharts. Flowcharts are a graphical representation of a process or workflow. It uses various shapes connected by arrows to illustrate the sequence of steps, decisions, and actions involved. Flowcharts are often used in business, engineering, and programming to visualize complex processes and make them easier to understand.

Here is an example:
This is a simple color detection algorithm.



Here is it in flowchart form.



Do you see how much easier it is to understand the code now? It is much more intuitive and the meaning is clearer. Without even learning how flowcharts are read you can already understand it!

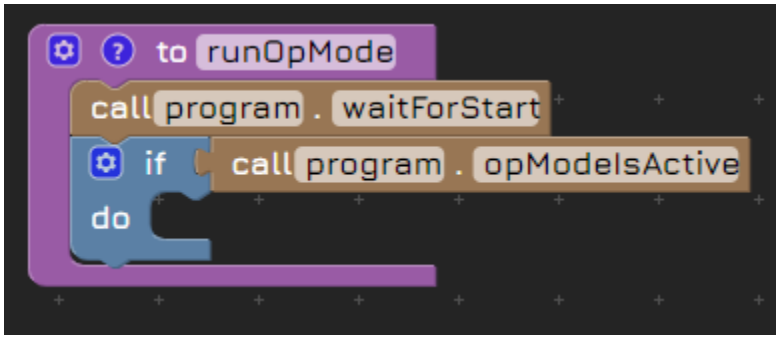
Now let's dive deep and analyze what each part means. Each shape of the flowchart represents different things:

Oval: Represents the start and end of a process.

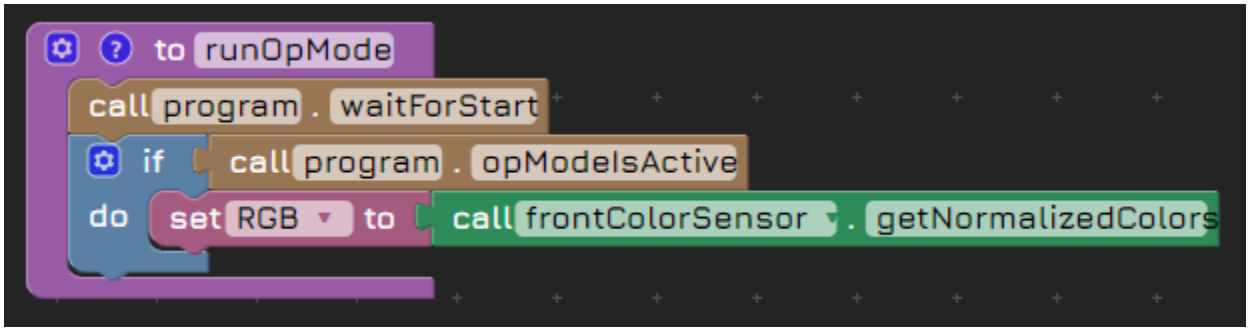
Rectangle: Represents a process or activity.
Diamond: Represents a decision point.
Arrow: Represents the flow of the process.
Parallelogram: Represents input or output.

Reading from the flowchart it is easy to see that this is meant to continuously monitor if the color red has been detected or not. Now implementing this into code becomes easy.

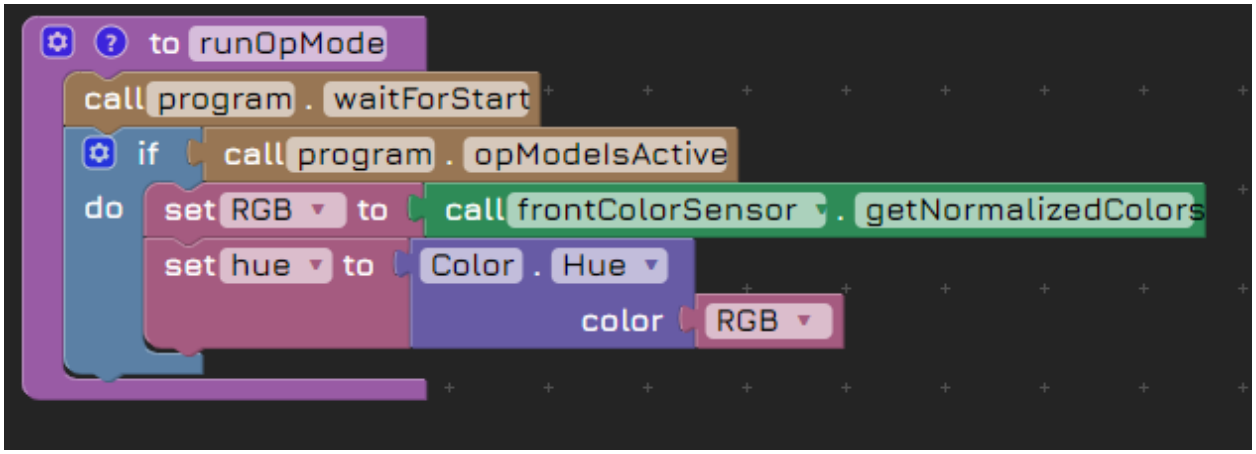
Here is a breakdown of the process adding each block from the flow chart:
Start program



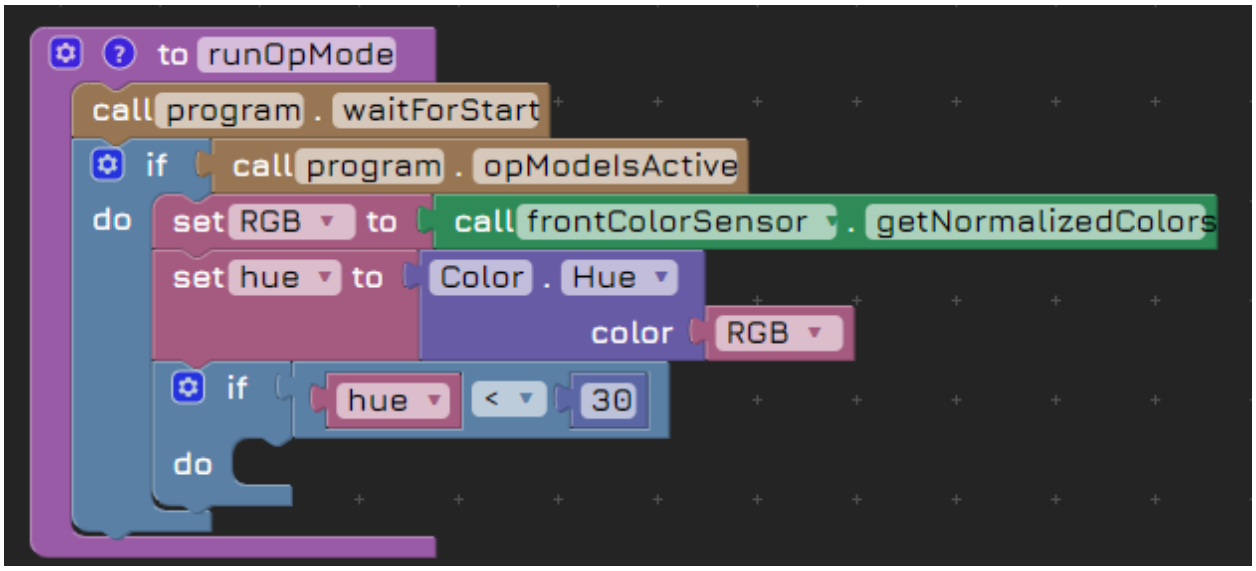
Get Color Sensor RGB Value



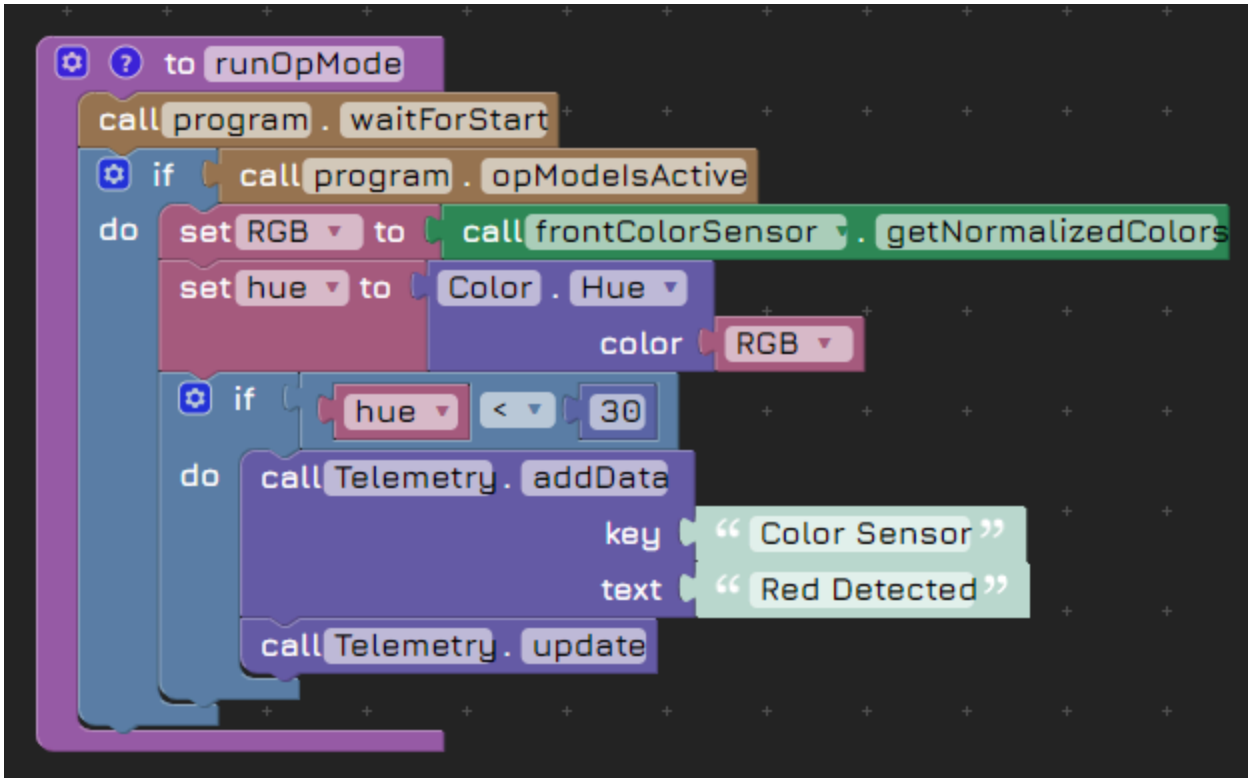
Get hue from RGBA value



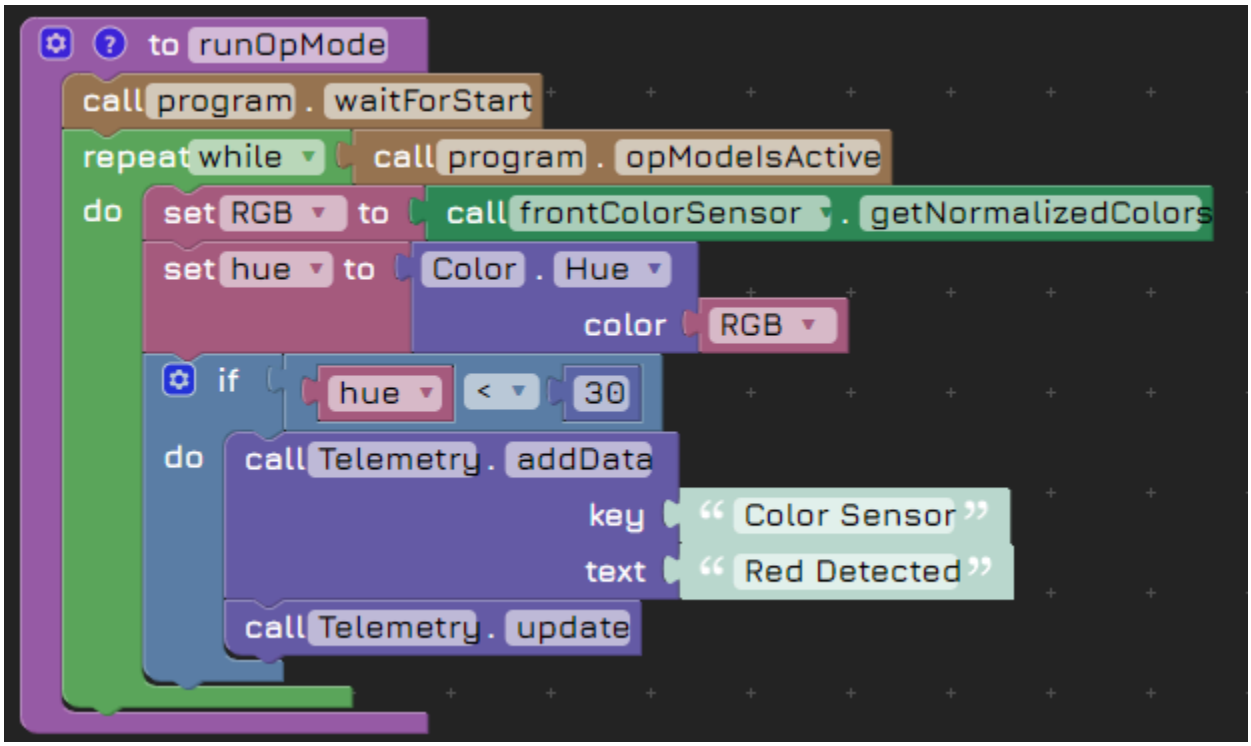
Is hue < 30 (Red)?



If yes:


to runOpMode
call program . waitForStart
if call program . opModeIsActive
do set RGB to call frontColorSensor . getNormalizedColors
set hue to Color . Hue
color RGB
if hue < 30
do call Telemetry . addData
key " Color Sensor "
text " Red Detected "
call Telemetry . update

If no:


to runOpMode
call program . waitForStart
repeat while call program . opModeIsActive
do set RGB to call frontColorSensor . getNormalizedColors
set hue to Color . Hue
color RGB
if hue < 30
do call Telemetry . addData
key " Color Sensor "
text " Red Detected "
call Telemetry . update

As you can tell, flowcharting can be lifesaving when creating complex code with multiple steps. Good coders will often use these flow charts to map out what they will write before coding. If you wish to write both efficiently and effectively you should use flowcharts too!

Functions

Are you tired of rewriting the same code over and over again? Well functions solve that! Functions in programming are reusable blocks of code that perform a specific task. Think of the steps to make a burger: cook patty, sandwich patty between two buns, and add vegetables. Now when you want to implement this, however, you have to arduously map out every substep. Instead of just cooking the patty, you have to grind up the filling, heat up the pan, etc. When you code out all these steps, the original 3 steps become increasingly hard to makeout. When you want to modify your code like by adding another patty, you have to scour your code to duplicate the original sandwich patty instructions. With functions, you don’t have to do that. Instead you can group the substeps into a single line of code. This way code can be easily duplicated and read. Now when you want to add an extra patty all you have to do is repeat the sandwich meat function.

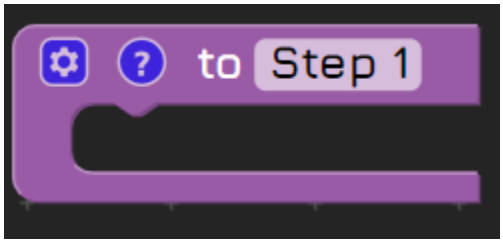
Let's see functions in action.

Take the sample code:

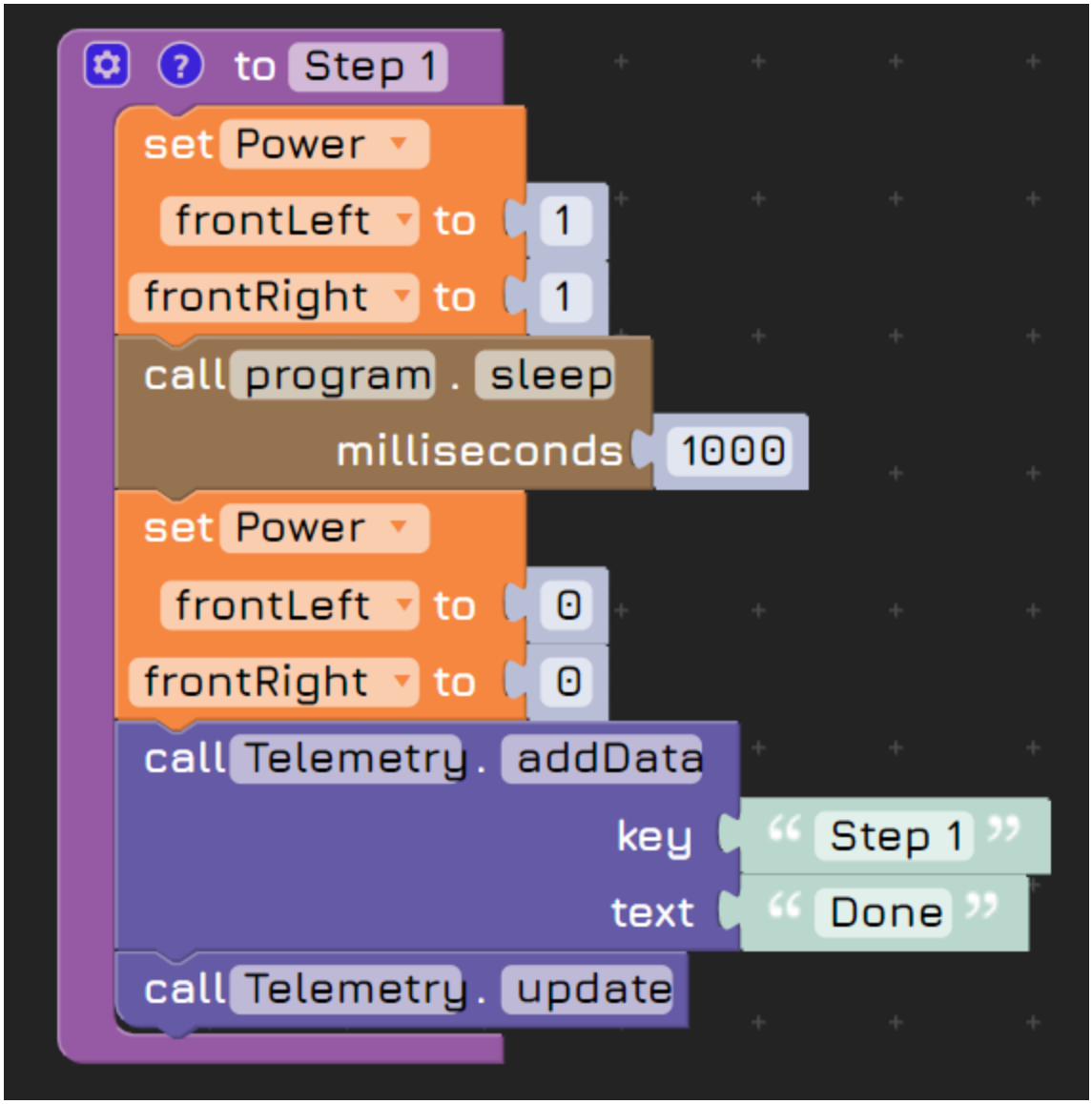

to runOpMode
Put initialization blocks here
set frontRight . Direction to Direction . REVERSE
set backRight . Direction to Direction . REVERSE
call program . waitForStart
if call program . opModeIsActive
do Put run blocks here
set Power
frontLeft to 1
frontRight to 1
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 1 "
text " Done "
call Telemetry . update
set Power
frontLeft to 1
frontRight to 0
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 2 "
text " Done "
call Telemetry . update
set Power
frontLeft to 0
frontRight to 1
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 3 "
text " Done "
call Telemetry . update
set Power
frontLeft to 0
frontRight to 1
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 4 "
text " Done "
call Telemetry . update

At first glance it is difficult to understand what the series of motor controls do. Modifying the code is made even more confusing. When we break down the code, we can notice a pattern: set power on, sleep, set power off, report step is done. Some of the steps are even repeated like steps 3 and 4.

The first step is to put these sets of commands into functions. Using functions are simple, first name a function



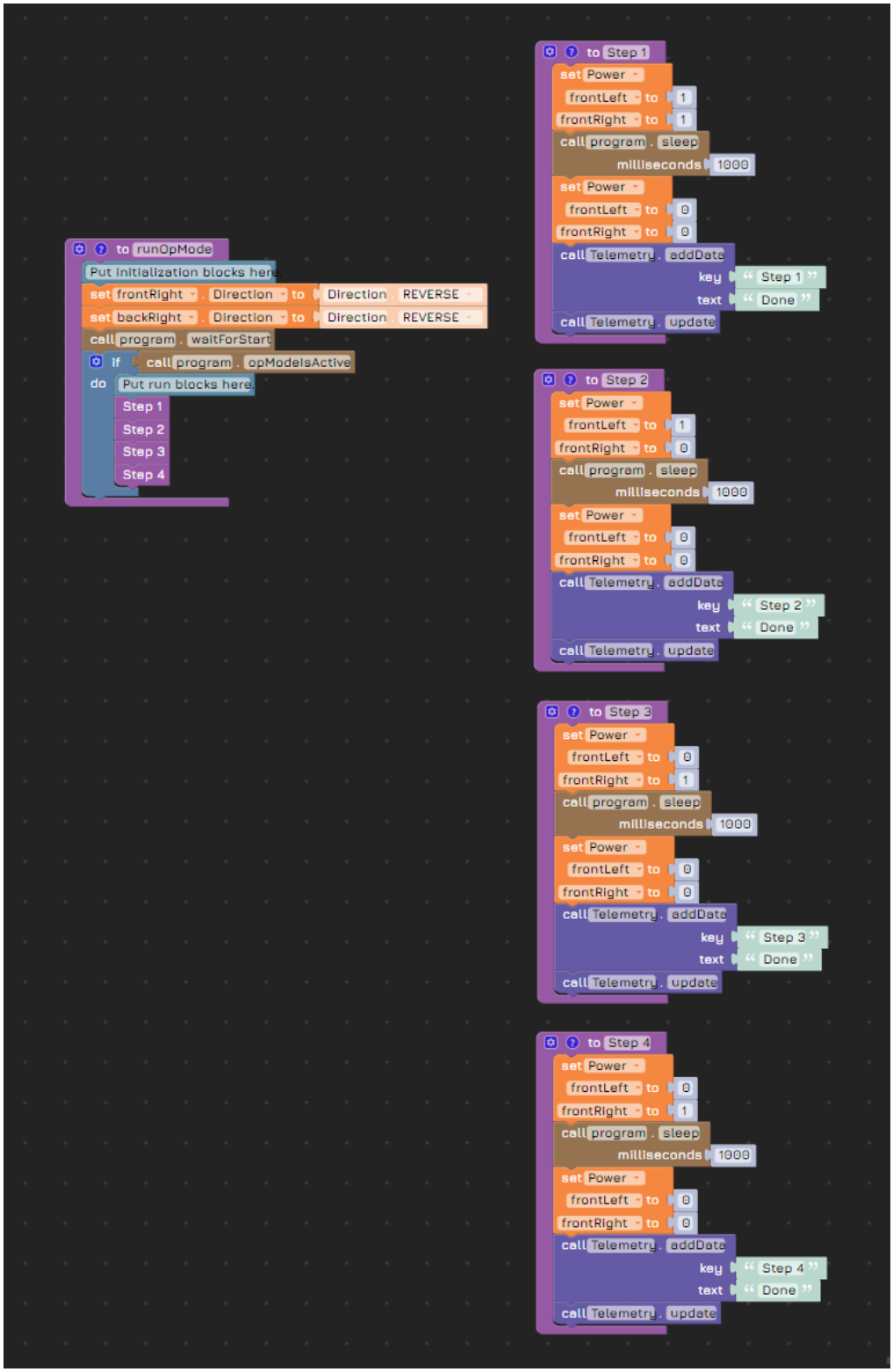
And then insert the code that represents that function.



Now that you have created a function, in order to you use you can simply insert the function's block where the original code was.


to runOpMode
Put initialization blocks here.
set frontRight . Direction to Direction REVERSE
set backRight . Direction to Direction REVERSE
call program . waitForStart
if call program . opModeIsActive
do Put run blocks here.
Step 1
set Power
frontLeft to 1
frontRight to 0
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 2 "
text " Done "
call Telemetry . update
set Power
frontLeft to 0
frontRight to 1
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 3 "
text " Done "
call Telemetry . update
set Power
frontLeft to 0
frontRight to 1
call program . sleep
milliseconds 1000
set Power
frontLeft to 0
frontRight to 0
call Telemetry . addData
key " Step 4 "
text " Done "
call Telemetry . update

Now repeat for steps 2, 3, and 4.



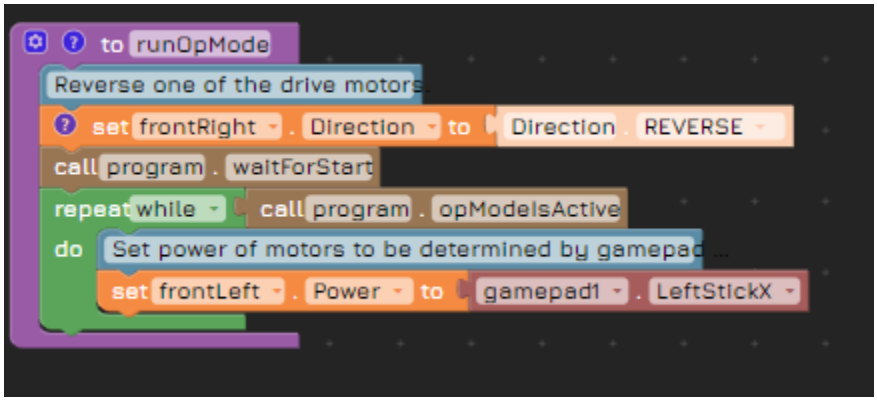
Now the code is much more readable! Remember to use functions when coding. Although they may seem unnecessary, they are vital in organizing more complex code. They will be used extensively in the advanced lessons to help break down more complicated sections.

Tele-Op Control

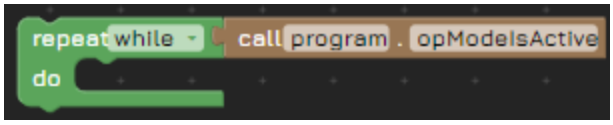
In FTC there is a phase of the competition known as TeleOp mode where teams control their robots in real-time using a driver station to complete tasks or objects. This is the last 2 minutes and 30 seconds of a match. This is famously the part where the drivers strategize, collaborate, and compete head to head.

Programming the robot for manual control is quite easy but making it ergonomic and intuitive is hard.

Most of your code will be generally structured like this:



1. After the program has started, you repeat the setting the power for the motors as long as opMode, the designated period to control the robot, is active.



2. Map how you want the power of each motor to be set to.

   The basic sections are:

   set [MotorName].Power to which determines which motor is being affected.

   Then you either set the motor to a gamepad control, like LeftStickX or RightStickY directly:

   or

   add a negative marker to reverse the gamepad's effect on the motor.

   Simply repeat the set command until you can control all the motors you want!

Don't forget that the motor direction can be reversed. This has the same effect as simply putting the negative marker on the control. A negative marker on a reversed motor will cancel out, causing the motor to behave as if it was just directly controlled by the gamepad.

Feel free to experiment with what you have learned and, most importantly, don't forget to have fun!

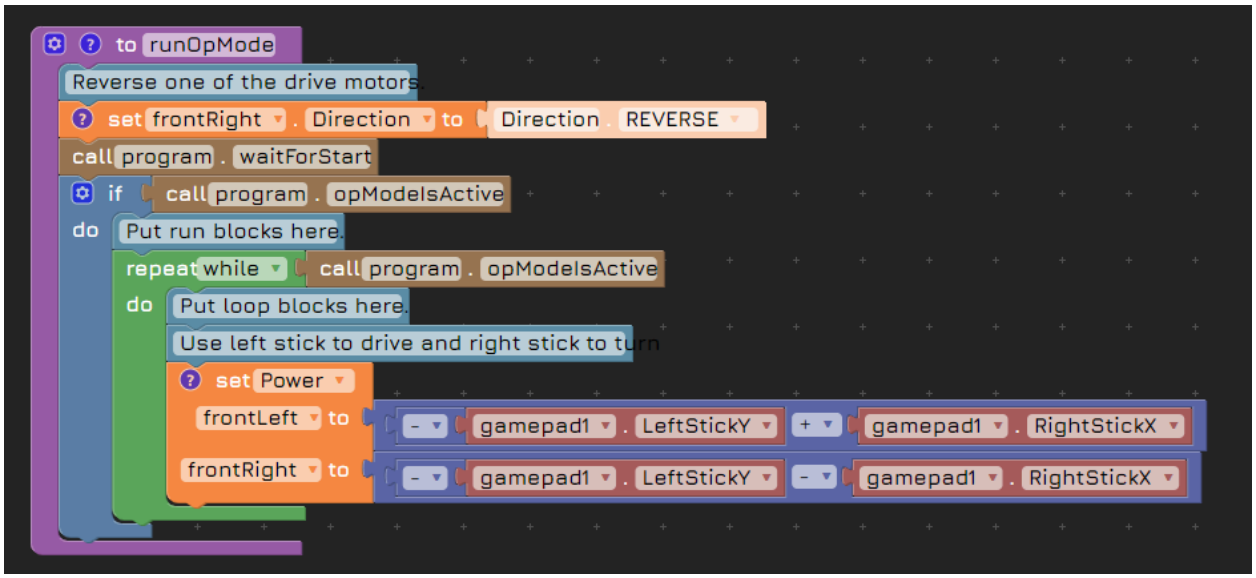
Advanced Lessons

More niche, but still useful lessons mostly expanding on basic topics or covering more specialized information. Not a measure of difficulty.

Advanced Tele-Op Control

Oftentimes simple button mapping isn't enough to pilot a robot. It's inconvenient to have to press an exact series of buttons to score an object or trigger an objective. During the game, in the heat of the moment, drivers can mess up, losing otherwise good points. To prevent this, sets of commands are usually linked to buttons.

Here is how it's typically done. We start with a simple program that has each motor mapped to a joystick.

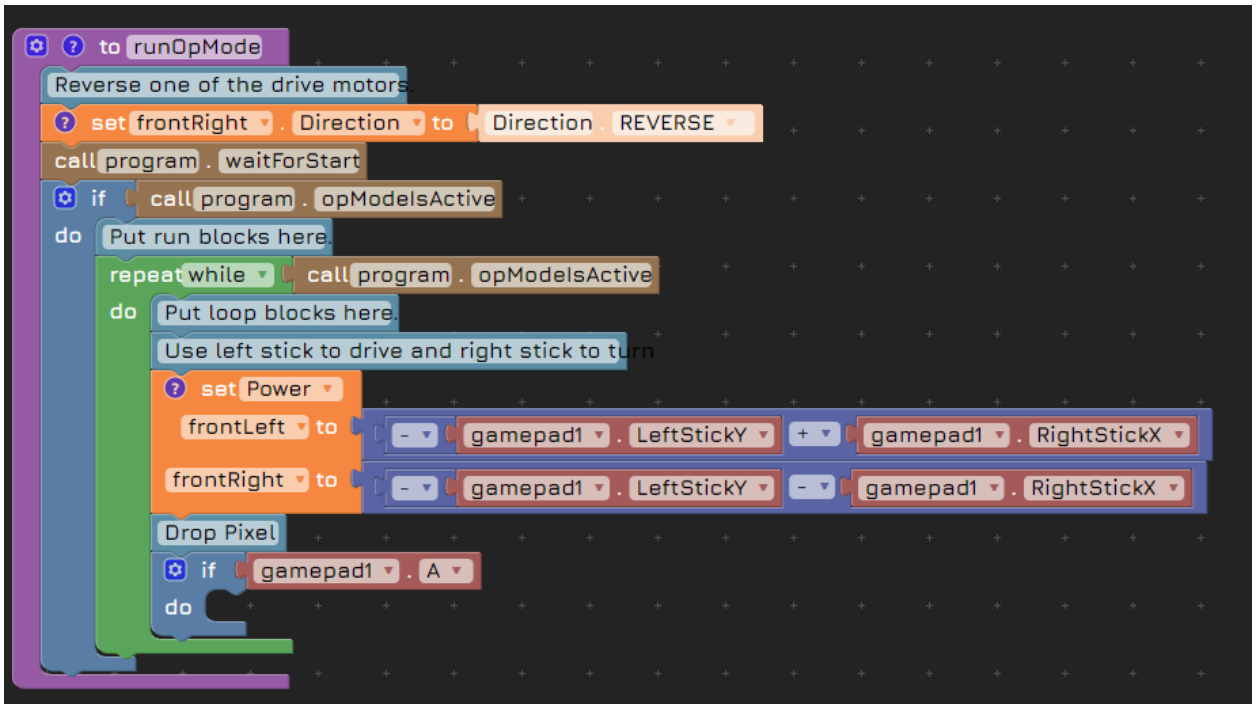


Here is where we can enhance the program.

One common repeated set of commands is opening and releasing a robot claw to pick up and drop game objects. We could code the motor so that a joystick determines how

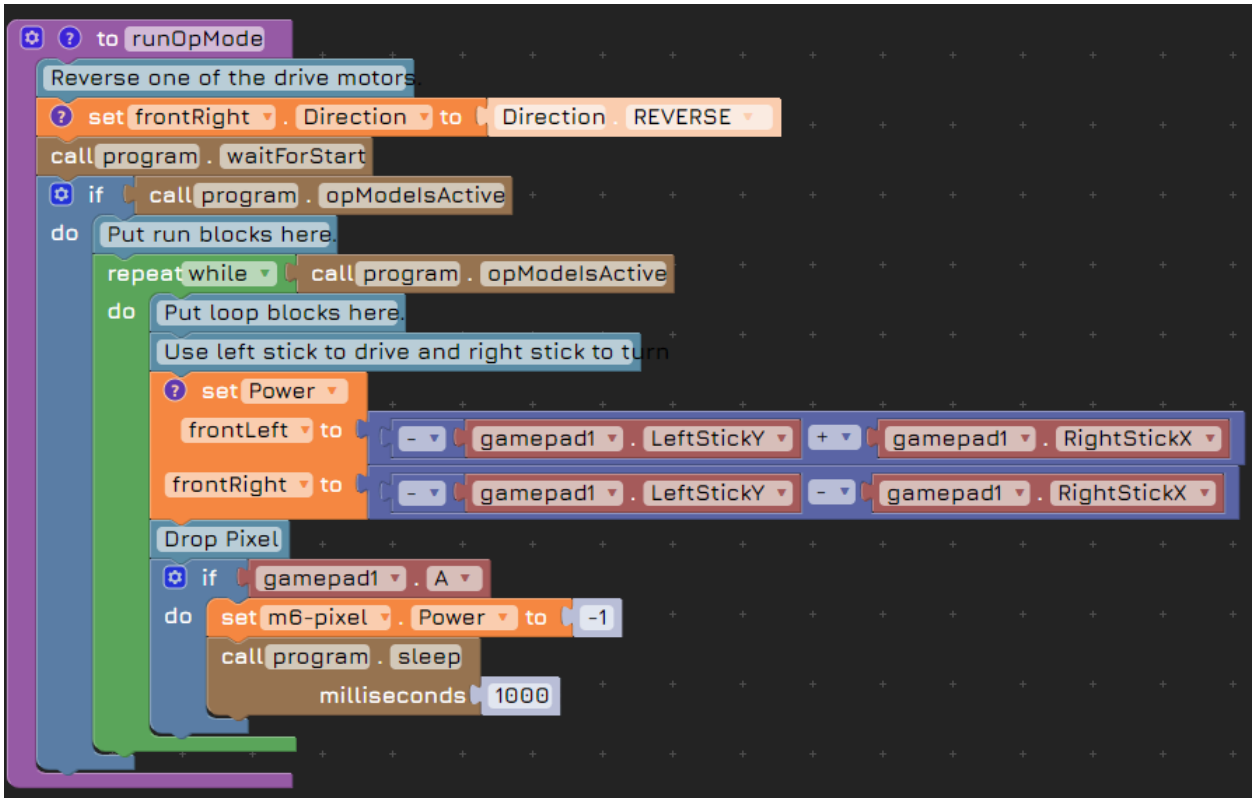
much they open and close but this is inefficient and forgetting to close the claw after each use can cause the claw to get jammed in something. A smarter solution is to just map all the steps onto a single button.

In the main loop of the program, we can add a special trigger if a button (for the demonstration the “A” button is used) is pressed.

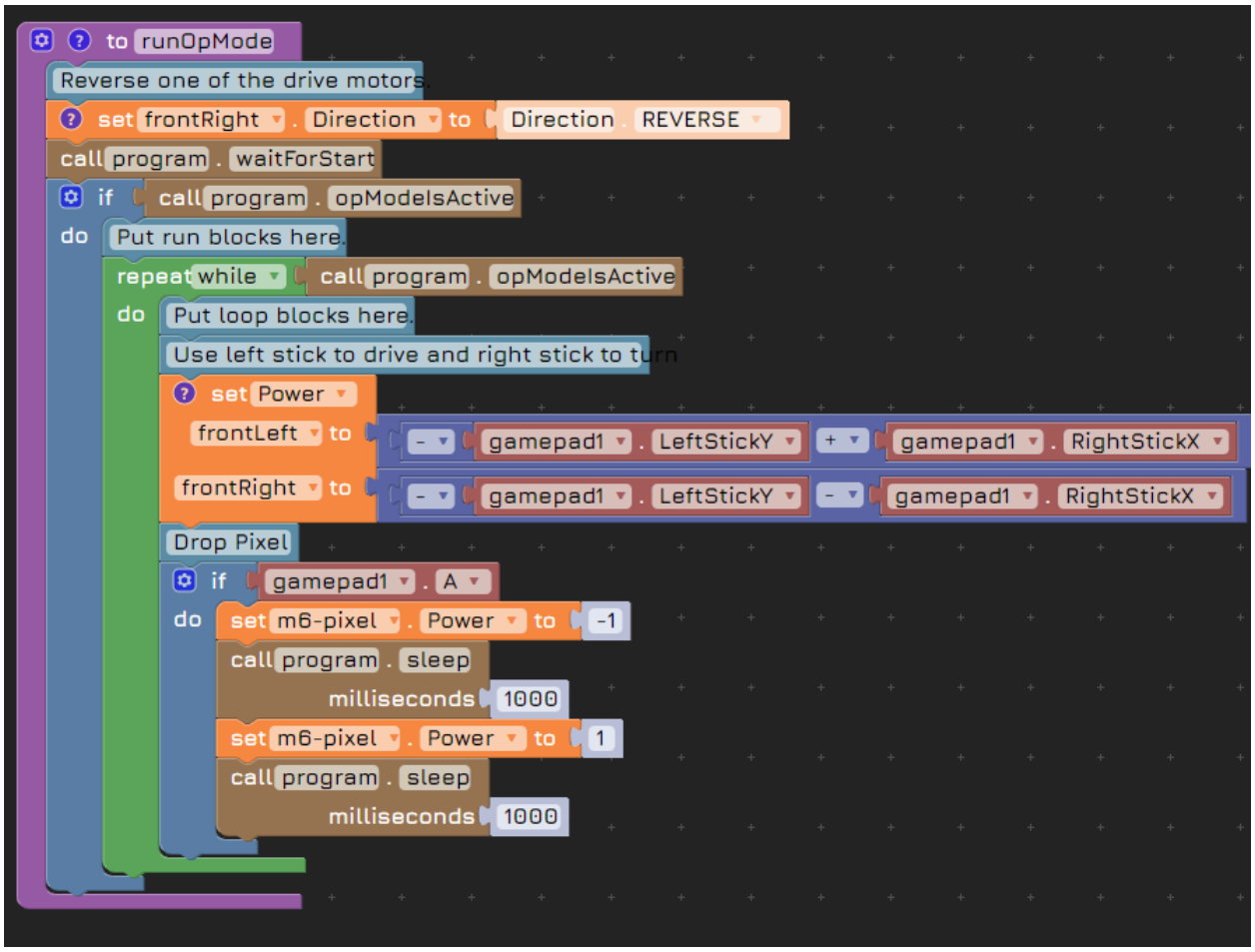


Now if the “A” button is pressed, we can set the robot to open and then close its claw.

First reverse the motor and give it a second so that the claw fully opens.



Then give the program a second to fully close the claw.



And there you have it, an algorithm in the press of a button!

Now, this demonstration may seem very simple but it's the effect it has that makes it so impactful. Now when you're piloting the robot, your controls are much smoother and intuitive. In real competitions, small quality of life features like this can be the tiebreaker between otherwise equally matched robots.

Video Lessons

If you prefer a more visual style of learning, please checkout our video tutorials!

Lesson 1 Introduction

https://www.youtube.com/watch?v=woGm9D8JqAQ&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=3

Lesson 2 User Interface

https://www.youtube.com/watch?v=5p6fw9KoQmw&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=9

Lesson 3 Motors and Loops

https://www.youtube.com/watch?v=lwbgx4CHOwk&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=5

Lesson 4 Gamepad Controls:

https://www.youtube.com/watch?v=42X3sz75xLE&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=4

Lesson 5 Motors and Loops Continued:

https://www.youtube.com/watch?v=vl1KwRwhAA8&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=2

Using VRS code for real robots

https://www.youtube.com/watch?v=o-esWDaF1hI&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=12

Release Video

https://www.youtube.com/watch?v=M0IyIFdGIQQ&list=PL2ovNdvJY8L78xf956fv26Spto0EE0F7M&index=1

Simulation

The simulation section of FTC VRS programming is a virtual environment where you test your robot's code without needing physical hardware. It allows you to visualize your robot's movements and interactions with the game field, test different programming strategies, and identify and fix errors in your code.

Preload Prop

Enabling this will cause the robot to start the match with the preloaded game object.

Use Tile 2

Will cause the robot to start in the alternative, lower starting spot.

Vision

Enables sensors

Select Mode

You can select one of four different modes: Autonomous, Tele-Op, Full Game, and Free Play.

Autonomous: the first 30 seconds of any FTC match. Here robots will move autonomously to complete tasks and score points.

Tele-Op: The last 2 minutes and 30 seconds of the FTC match. During this period, the robot can be manually controlled. During the last 30 seconds, called endgame, additional tasks can be completed, scoring more points.

Full Game: Combines Autonomous with Tele-Op, running Autonomous first. After Autonomous has completed, there will be a 3 second transition period for the driver to assume control before Tele-Op begins.

Free Play: Freely play around and test your robot! Here there is no timer so you can take your time to test your robot.

Select A Bot

Choose between Bot 4 and Bot 2! These two robots offer alternative ways to play the centerstage game.

Here are the alternative controls:

Centerstage Gamepads

CS bot#2

1-Lower the hook
2-raise the hook
3-Move pixel holder out-tilt up
4-Move pixel holder down-tilt back

b-release pixel
a-change pixel color

LB/RB lift pixel -drop pixel

LT-load plane X-launch palne

CSBO#4

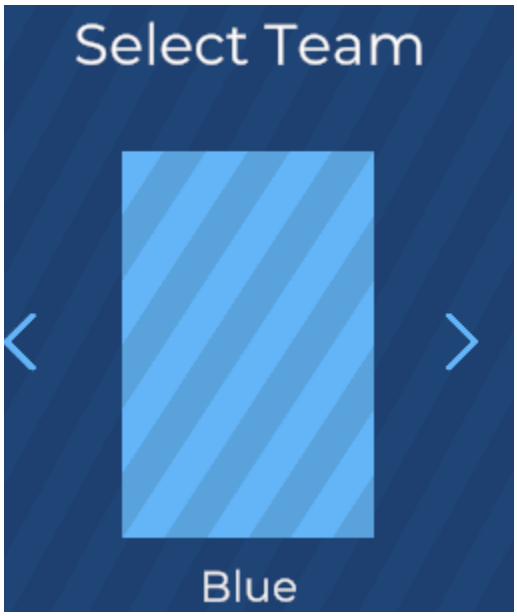
1-Lower the hook
2-raise the hook
3-Move pixel holder out-tilt up
4-Move pixel holder down-tilt back

b-release pixel
a-change pixel color

LB/RB lift pixel -drop pixel

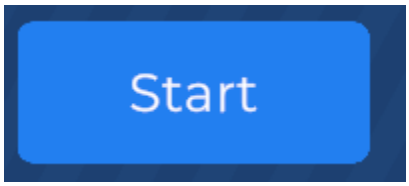
LT-load plane Letter H -launch plane

Select Team



Choose to play for blue or red. This will mainly affect where you spawn and score. Blue spawns on the left side. Red spawns on the right side. Depending on the game, your team color will affect where you can move, pick up blocks, and score.

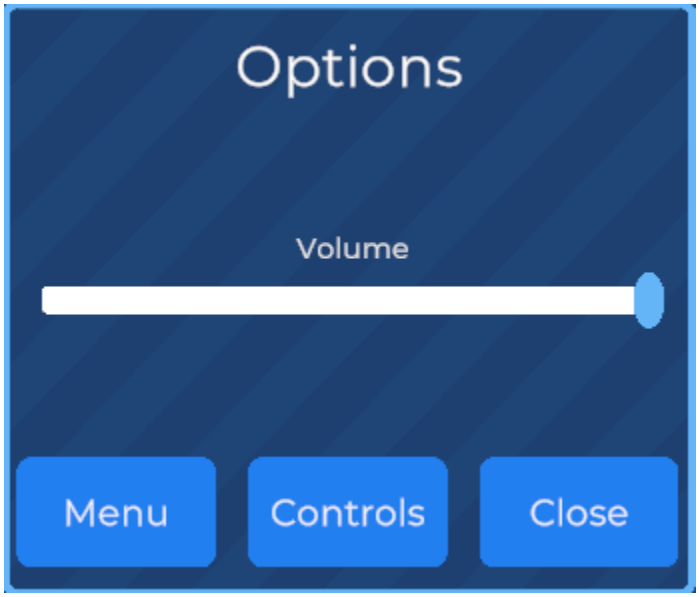
Start



Begins the match, starting the match timer.

Options

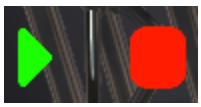
Allows you to adjust the volume, change season games, and rebind the robot's controls.



Other Options

Provides you with a basic diagram of gamepad controls and some lessons.

Pause/Continue



Pauses and resumes the game respectively.

Reset

Resets the game, taking you back to the setup screen.

ScoreLog

A log of all the points scored throughout the game.

Leader

Keeps track of the highest scoring robots.

Exploratory Activities

Now that you have learned the basics, here are some ideas for you to freely explore and develop your skills. These challenges are meant to help you think like a programmer, coming up with inventive solutions to common problems.

Wandering Robot

The wandering robot algorithm is a great way to practice your obstacle avoidance skills. The idea is simple: you program the robot to randomly move without a goal or

destination for as long as possible. The program can also be used for exploration or to map out environments.

The basic algorithm goes as follows:

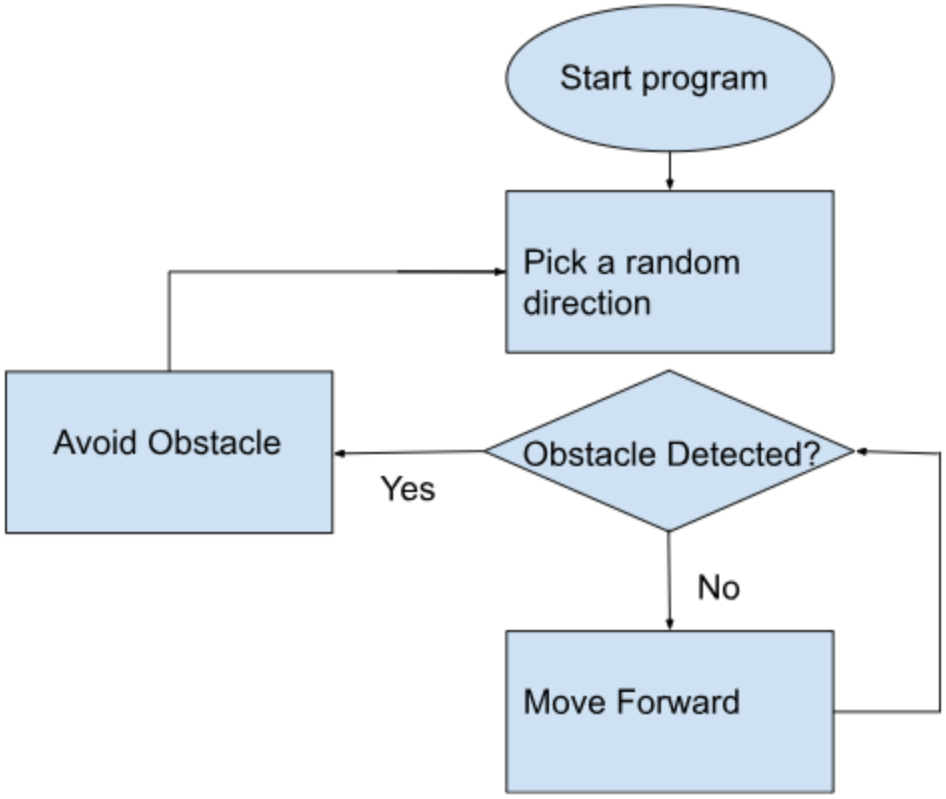
1.
Random Direction: The robot randomly selects a direction to move (e.g., forward, backward, left, right).

2.
Movement: The robot moves in the selected direction for a predetermined distance or time.

3.
Obstacle Avoidance: If the robot encounters an obstacle, it randomly selects a new direction and tries again.

4.
Repeat: The process is repeated indefinitely, creating a seemingly random wandering pattern.

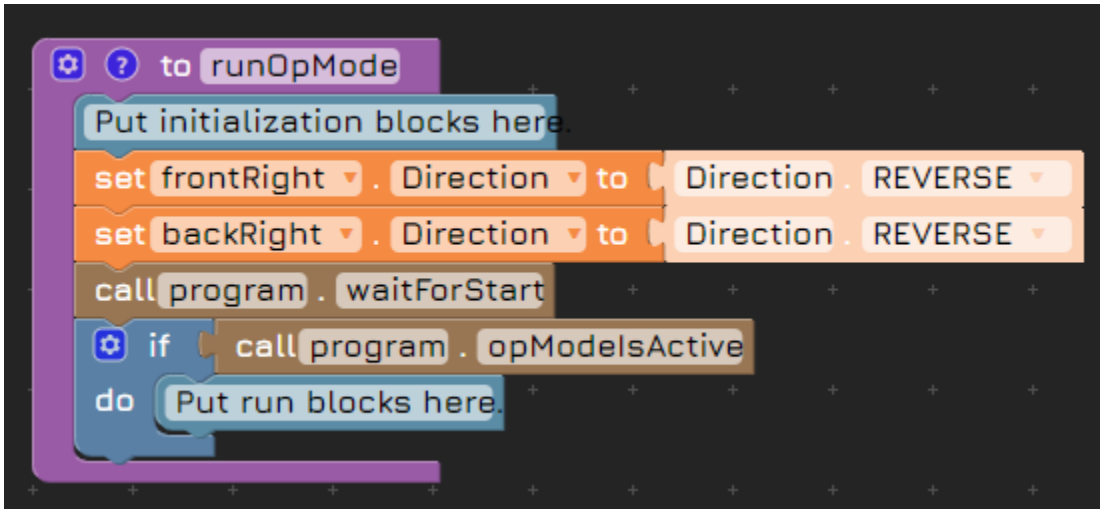
Mapping this out in a flow chart:



So let's implement it!

1.

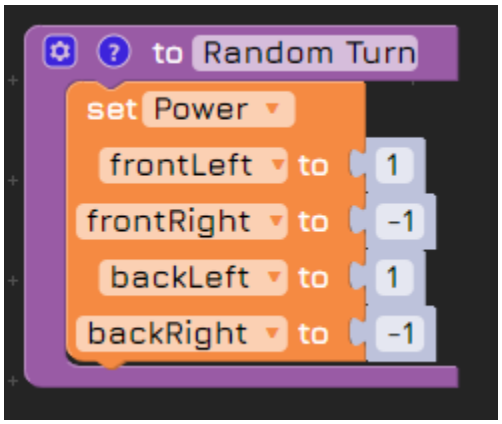
First start the program:



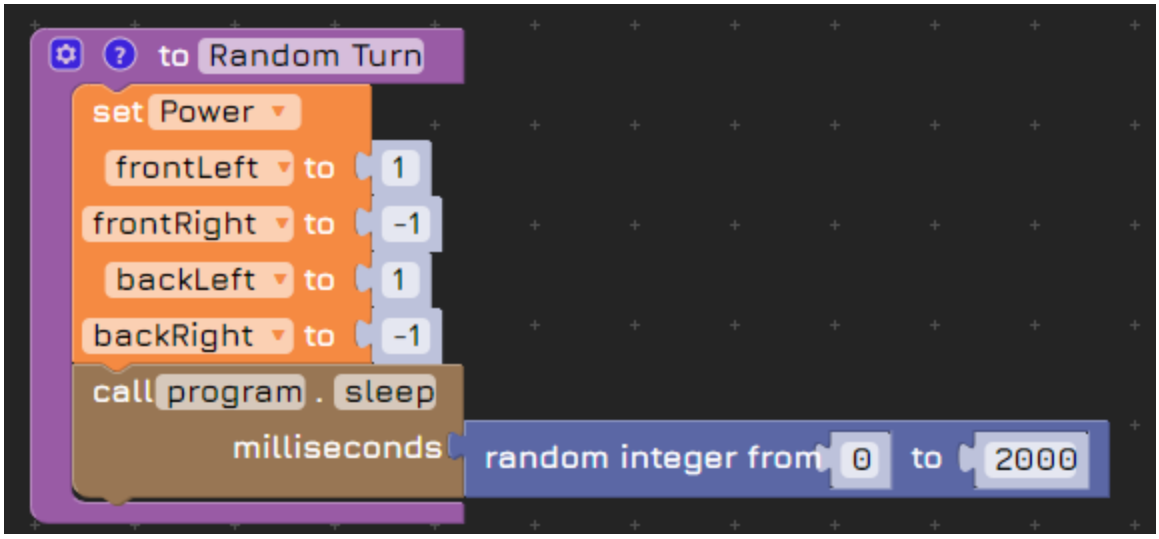
Don’t forget to reverse the direction of the right side motors when initializing. This will make programming a lot easier!

2.
Next implement a random direction function.

To keep things simple, I will have the robot turn for a random amount of time before stopping. This will effectively spin the robot to a random orientation.



By having the right motors reverse when the left motors go forward, the robot can rotate in place. This is important as you don’t want the robot to get stuck while selecting a direction to proceed. The collision detection step isn’t active yet so getting stuck now will stop the robot.

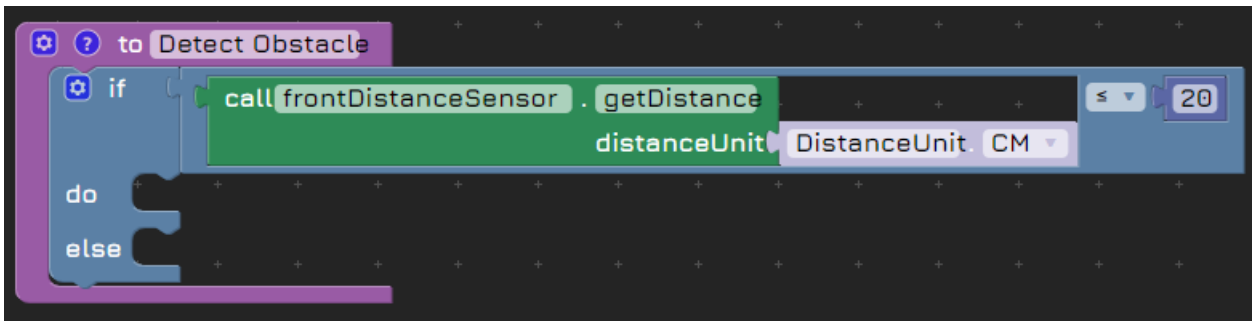


Next, have the program sleep for a random amount of time. I choose between 0 and 2 seconds but this range can be anything.

3.

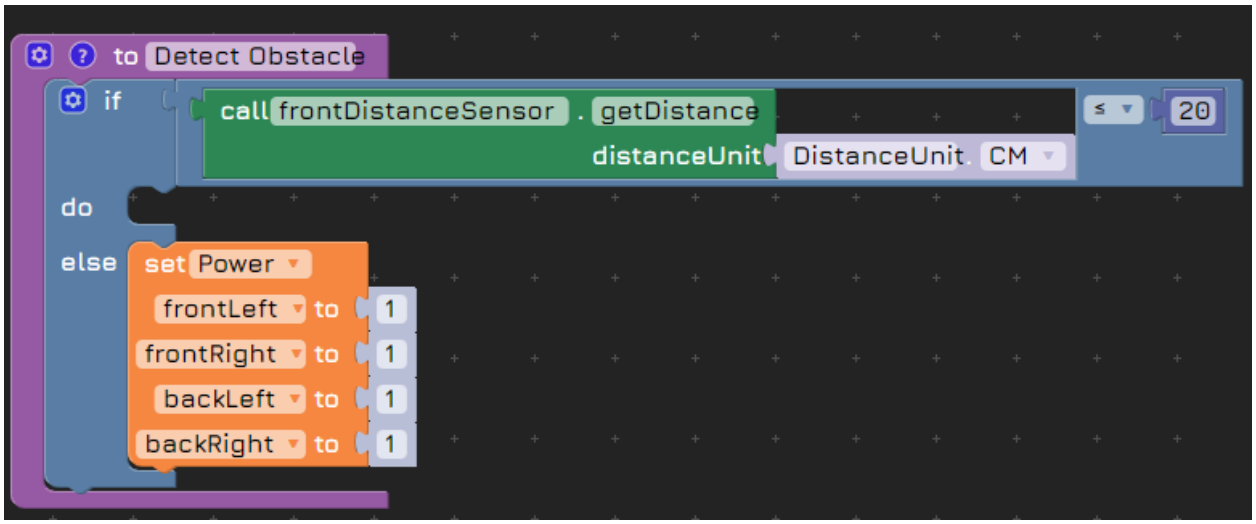
Obstacle Detection

There are multiple different ways of implementing this and you are highly encouraged to explore different methods. One of the easiest is using the distance sensor. Since the robot will be moving forwards, most of the obstacles will be in front of the robot so a distance sensor works well in most cases.

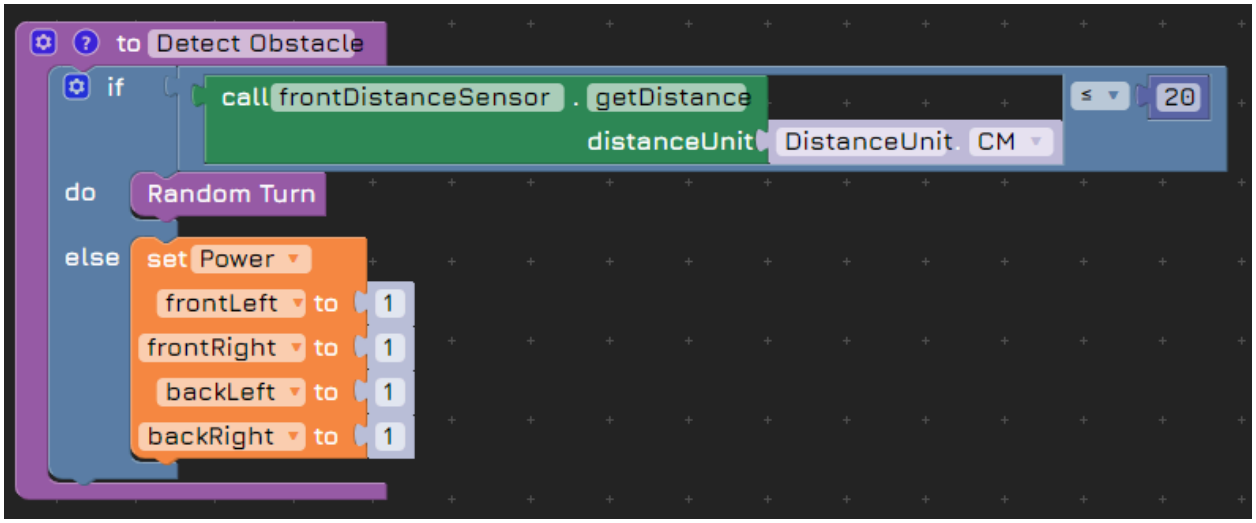


Here I use an if do else logic block. You know collision is imminent if there is an object present within 20 cm of the robot. The 20 cm gap is necessary as you need to give the robot space to turn and avoid the obstacle.

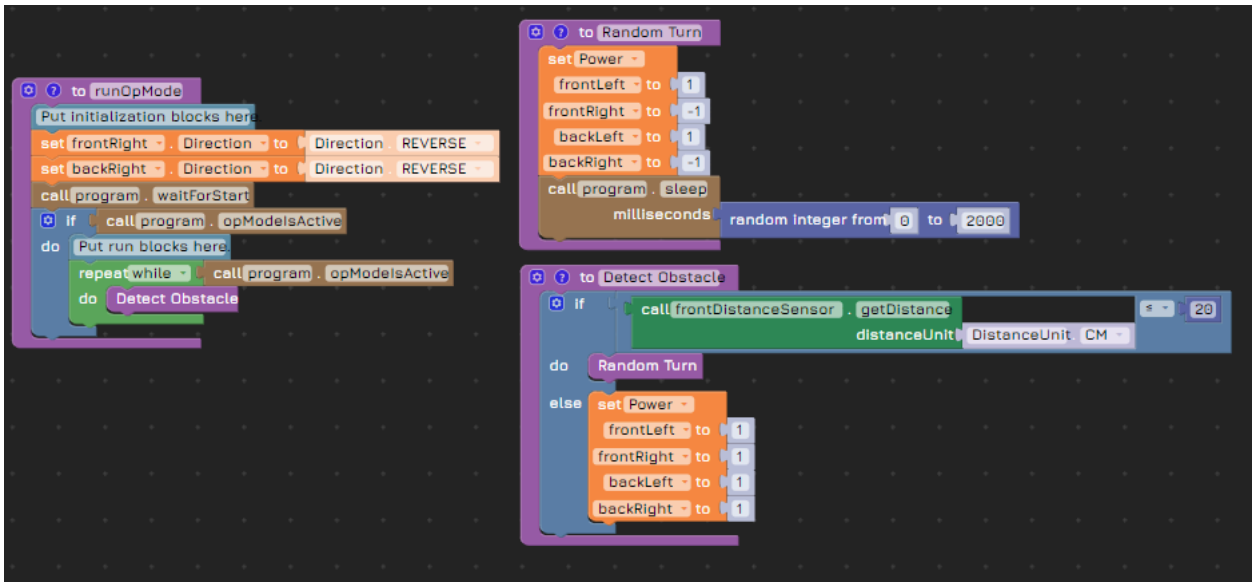
If there is no obstacle, move the robot forward.



If there is an obstacle, we can use the random turn code from earlier as an obstacle avoidance method. Afterall, there the random turn doesn't avoid the obstacle, it will be quickly detected and the robot will turn until no obstacle is found.



Now just insert the Detect Obstacle function into the main program and run it on a loop. Voila! You have a working wandering robot code.



This is just one out of many versions of this code. One variant has the robot repeatedly turn left and right to try to widen the detection range of the distance sensor. Try to challenge yourself to create something better!

Robot Arena

Welcome to the Robot Arena (https://centerstage.vrobotsim.online/arenapage.html)! Here you can practice your robot piloting skills both individually and with other players. Using our selection of premade robots, you can play on FTC games Centerstage, letting you really feel what it's like, piloting a robot from the Alliance Station. Will you get the next high score? Only skill, cooperation, and Gracious Professionalism will tell.

Single Player

"Single Player" mode allows you to practice your robot programming skills against simulated opponents or challenges. This could be a helpful way to test your code and improve your robot's performance before competing against other teams.

Preload Prop

Enabling this will cause the robot to start the match with the preloaded game object.

Use Tile 2

Will cause the robot to start in the alternative, lower starting spot.

Vision

Enables sensors

Select Mode

You can select one of four different modes: Autonomous, Tele-Op, Full Game, and Free Play.

Autonomous: the first 30 seconds of any FTC match. Here robots will move autonomously to complete tasks and score points.

Tele-Op: The last 2 minutes and 30 seconds of the FTC match. During this period, the robot can be manually controlled. During the last 30 seconds, called endgame, additional tasks can be completed, scoring more points.

Full Game: Combines Autonomous with Tele-Op, running Autonomous first. After Autonomous has completed, there will be a 3 second transition period for the driver to assume control before Tele-Op begins.

Free Play: Freely play around and test your robot! Here there is no timer so you can take your time to test your robot.

Select A Bot

Choose between Bot 4 and Bot 2! These two robots offer alternative ways to play the centerstage game.

Here are the alternative controls:

Motor	Keyboard	Motor Controls (Power)
m5-arm up	F	+X
m5-arm down	R	-X
m6-pixel pick up	C	+X
m6-pixel drop	V	-X
m7-launch launch	H	+X
m7-launch spin up	Y	-X

m8-tilt-pixel up m8-tilt-pixel down	T G	-X +X

Centerstage Gamepads

CS bot#2

CSBO#4

1-Lower the hook
2-raise the hook
3-Move pixel holder out-tilt up
4-Move pixel holder down-tilt back

b-release pixel
a-change pixel color

LB/RB lift pixel -drop pixel

LT-load plane X-launch palne

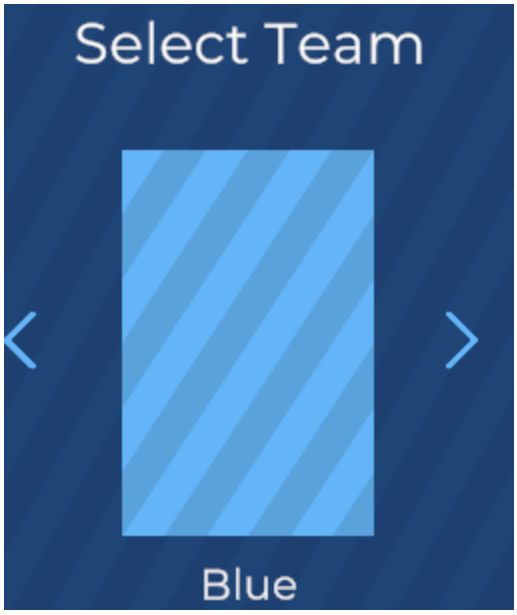
1-Lower the hook
2-raise the hook
3-Move pixel holder out-tilt up
4-Move pixel holder down-tilt back

b-release pixel
a-change pixel color

LB/RB lift pixel -drop pixel

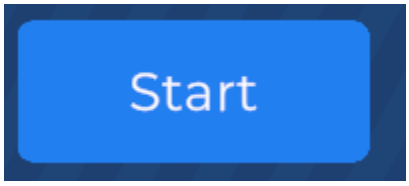
LT-load plane Letter H -launch plane

Select Team



Choose to play for blue or red. This will mainly affect where you spawn and score. Blue spawns on the left side. Red spawns on the right side. Depending on the game, your team color will affect where you can move, pick up blocks, and score.

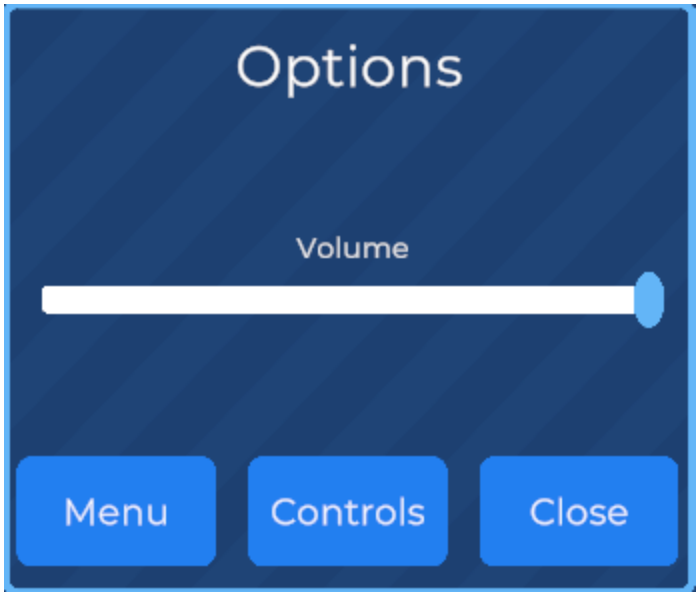
Start



Begins the match, starting the match timer.

Options

Allows you to adjust the volume, change season games, and see the robot's controls.



Other Options

Provides you with a basic diagram of gamepad controls and some lessons.

Pause/Continue

Pauses and resumes the game respectively.

Reset

Resets the game, taking you back to the setup screen.

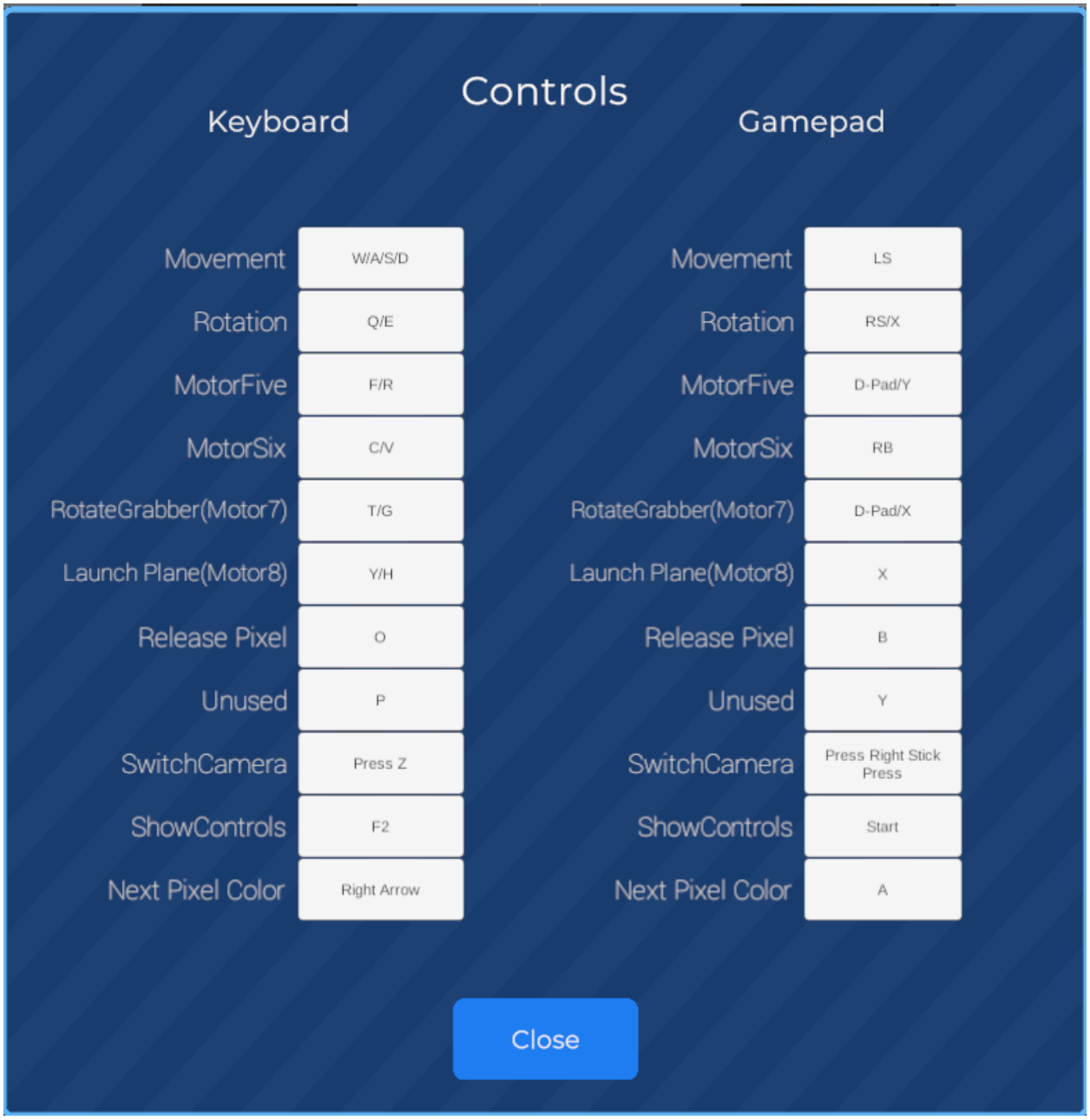
ScoreLog

A log of all the points scored throughout the game.

Leader

Keeps track of the highest scoring robots.

Controls



Activities

Activities to do with the robot arena. Largely just teaching how to host local virtual robot games.

Now that you have learned what everything does, time to use it!

Coding Competition

Hosting a local coding competition can be a rewarding experience for both organizers and participants. It provides a platform for aspiring programmers to showcase their skills, network with peers, and learn from experienced developers. To successfully organize a competition, it's essential to establish clear guidelines, create engaging challenges, and provide adequate support to participants. By fostering a positive and competitive environment, you can inspire the next generation of tech talent and contribute to the growth of the local tech community.

CenterStage Competition

Hosting a local robotics competition is a rewarding experience that can inspire young minds and foster a love for STEM. To successfully organize such an event, careful planning is essential. Begin by securing a suitable venue, such as a school gymnasium or community center. Next, recruit volunteers to assist with various tasks, from registration to judging. Promote the competition through local schools, community organizations, and social media to attract participants. On the day of the event, ensure that all necessary equipment, including fields, robots, and controllers, is readily available. Provide a welcoming atmosphere for teams and spectators, and offer food and refreshments for sale. Finally, celebrate the winners and encourage all participants to continue their robotics journey.

Conclusion

Now that you have reached the end of my book, I hope you have enjoyed it! Keep in mind that the VRS is still very much in development and this book will be updated as new changes roll in. I hope that you have been inspired as much as I have to pursue STEM. If you have enjoyed using the VRS I strongly encourage you to participate in FTC. FTC is a global event and teams are present in most countries around the world. If there are no local teams feel free to organize and start your own local branch. This is very much encouraged so that more aspiring kids can become interested in the sciences. Do check out our video guides on youtube if you want to learn more. As always, remember to have fun!

Author

Hi, I’m Eric Zhang, a senior at Eastlake High School. Ever since I was young I’ve always been interested in engineering. From building castles out of legos to now working with steel beams and bolts, I’ve come a long way. In 2023, I won the FTC World Championship. Now, I want to share the years of knowledge I’ve accumulated to help give you the head start I wish I had when I first started. I hope you really enjoy my book.