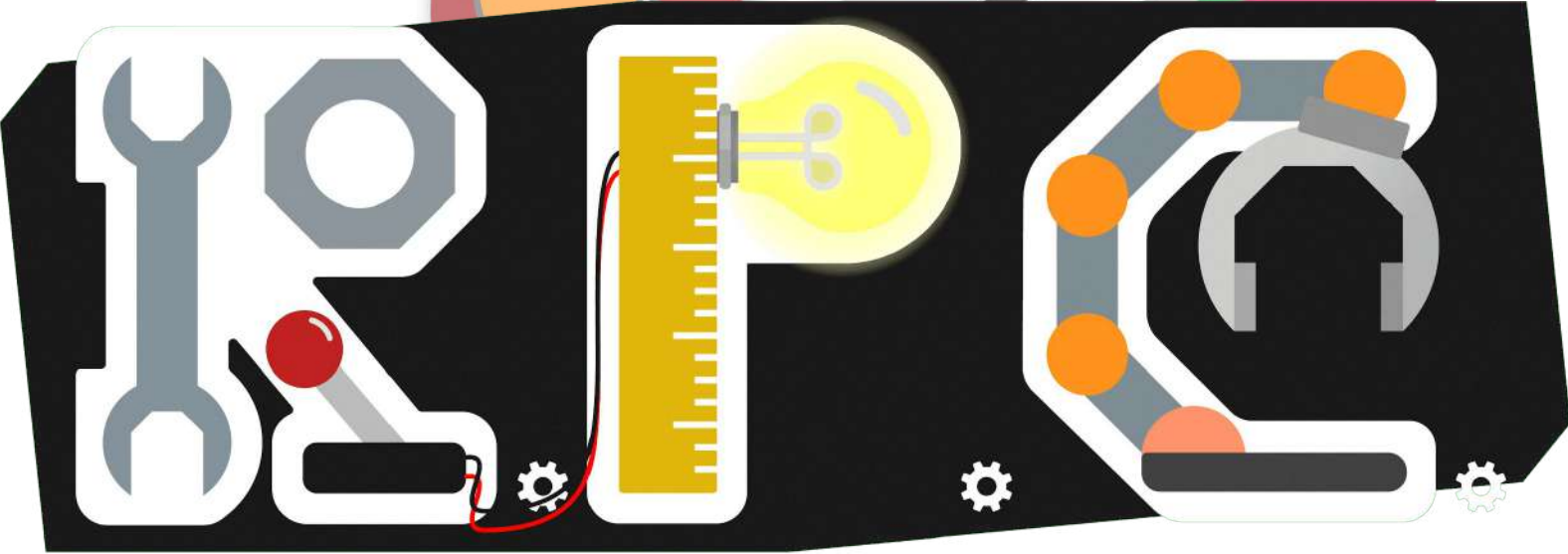


WELCOME TO...

THE



vex[®]
ROBOTICS
COMPETITION
(TOWER TAKEOVER)



A **robot** is a programmable mechanical device that can perform tasks and interact with its environment, without the aid of human interaction

CONTENTS TREE

I. How to Plan

- A. [Draft a Design Brief](#)
- B. [Brainstorm](#)
- C. [The Technical Design Review System](#)
 - 1. [Author Design Docs](#)
 - 2. [Discuss](#)

II. How to Build

- A. Structure
 - 1. [Metal](#)
 - 2. [Screws](#)
 - 3. [Nuts](#)
 - 4. [Standoffs](#)
 - 5. [Additional Topics in Structure](#)
- B. Motion
 - 1. [The Square Shaft](#)
 - 2. [Actuators](#)
 - 3. [Spacers & Collars](#)
 - 4. [Gears and Wheels](#)
 - 5. [Additional Topics in Motion](#)
- C. Sensorial
 - 1. [Analog vs. Digital](#)
 - 2. [Primitive vs. "Smart" Hardware](#)
 - 3. [The Microcontroller](#)
 - 4. [The Cortex Microcontroller](#)
 - 5. [Wiring Up the Cortex Microcontroller](#)
 - 6. [The V5 Robot Brain](#)
 - 7. [Wiring Up the V5 Robot Brain](#)
 - 8. [Bumper Switch](#)
 - 9. [Limit Switch](#)
 - 10. [Ultrasonic Sensor](#)
 - 11. [Light Sensor](#)
 - 12. [Potentiometer](#)
 - 13. [Optical Shaft Encoder](#)
 - 14. [V5 "Smart" Motor](#)
 - 15. [V5 Vision Sensor](#)
 - 16. [Sense, Plan, Act \(SPA\)](#)
 - 17. [Programming the Robot](#)

III. How to Program

- A. [RobotC and the Cortex Microcontroller](#)
- B. [VEX Code Studio and the V5 Brain](#)

1. How to Plan

The Design Process

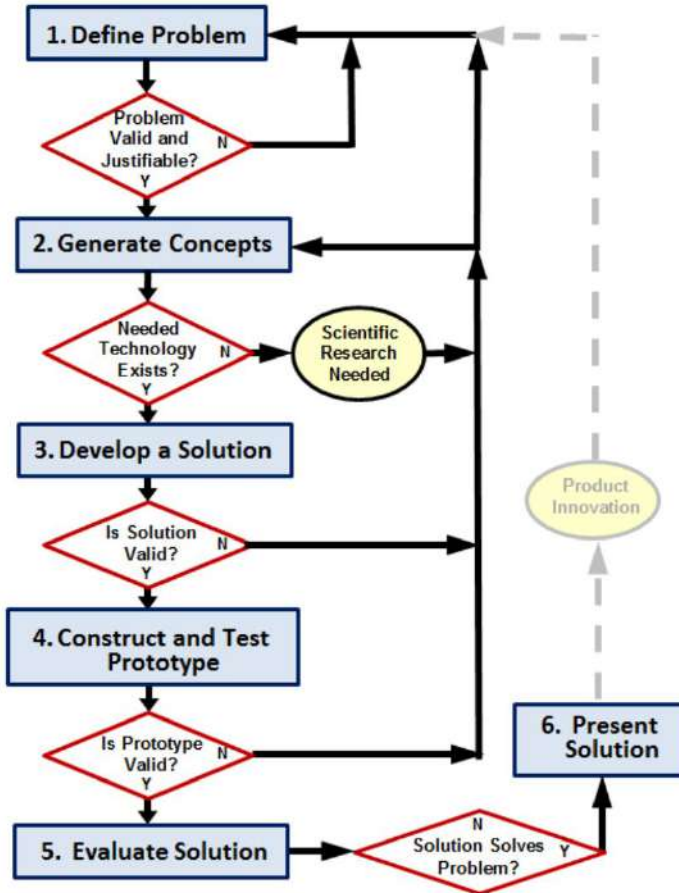
- Create a Design Brief
(shown on the next slide)

- Research
- Brainstorm
- Select an approach

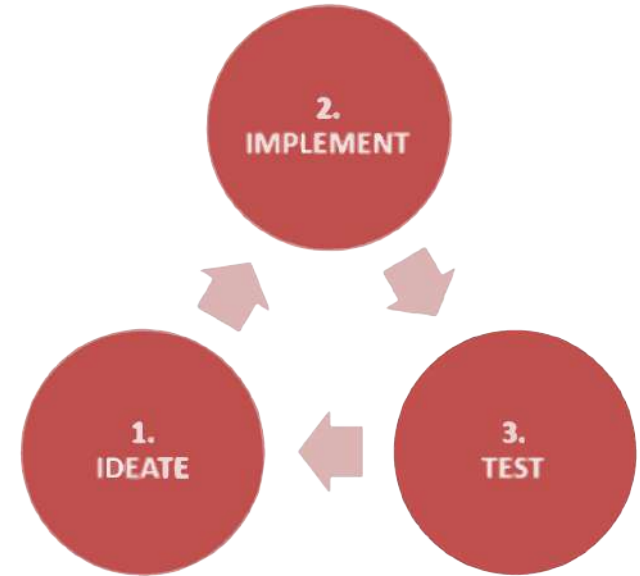
- Create detailed design solution (*design doc*)
- Create a technical drawing

- Construct the robot
- Program robot behavior
- Iterative development

- See if the solution actually works



[1]



[2]

DRAFT A DESIGN BRIEF.

The problem statement clearly and concisely identifies the problem.

A problem statement must never imply or state a solution. The solution is not the problem.

The design statement challenges the engineer to take action to address the need and to solve the problem.

A good design statement should not unintentionally bias the engineer's creative thought process by using terminology that suggests an already existing solution.

Problem Statement

Robotic Engineers need a quick and easy way to confirm the working functionality of all programmable robotic components before they are deployed on a project.

Design Statement

Design a tool that can accommodate all common programmable robotic components including motors for proper testing of functionality.

Criteria

1. Robust
2. Reusable
3. Expandable

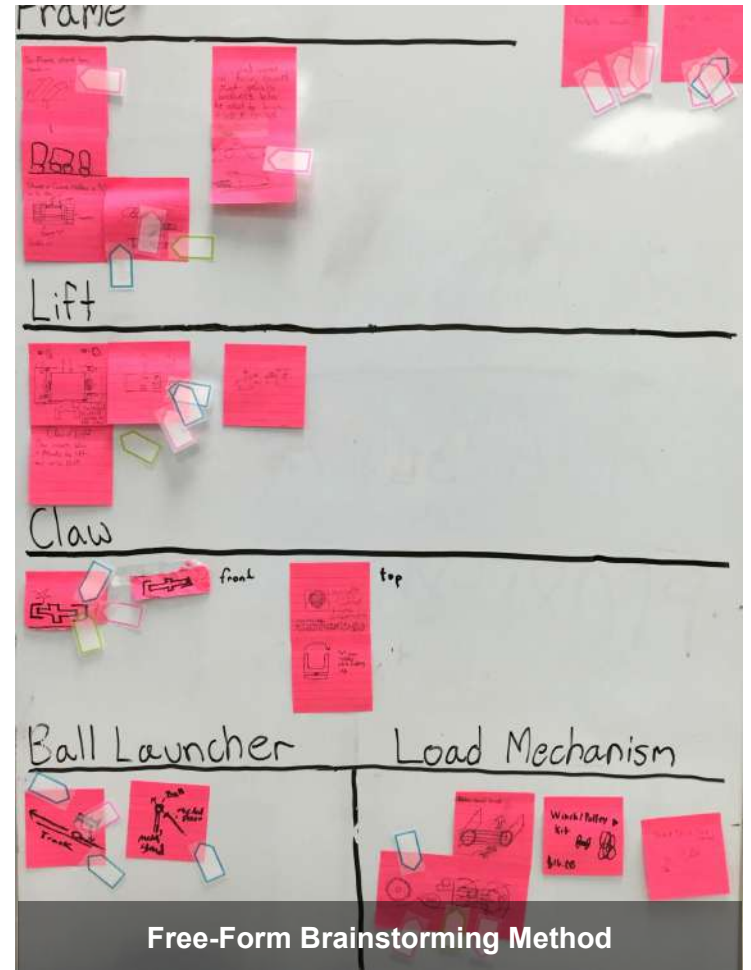
BRAINSTORM.

Rules

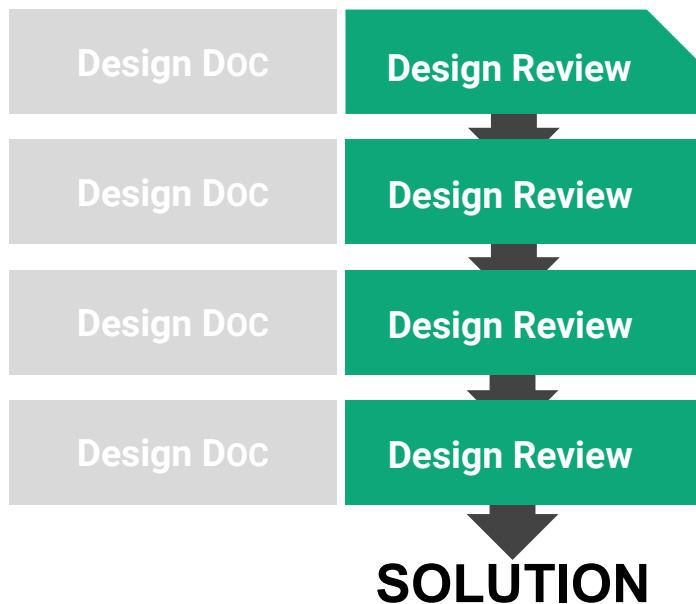
- No criticism allowed
- Work for quantity
- Welcome piling-on
- Allow free-for-all

Dysfunctions

- Utilizing a poor design brief
- Assuming there is only one right answer
- Getting hooked on the first solution
- Considering ideas from only one or two team members
- Feeling too anxious to finish
- Becoming frustrated by lack of success
- Getting hooked on a solution that almost works



THE TECHNICAL DESIGN REVIEW SYSTEM ^[3]



A new engineer will change the code and break something because they don't know the context that came before; engineers will overlap, working on similar problems without realizing it; and significant time (particularly for the senior engineers untangling the mess) will be wasted.

The Technical Design Review System has been the most successful method of correspondence at Google. It uses design documents and discussions for keeping engineering teams healthy, communicating clearly, and effective even as they absorb more people and projects.

AUTHOR DESIGN DOCS.

Design documents describe, scope, and approve projects or features.

If catalogued historically, they can be the best way for new engineers to get up to speed on why a product or feature was built, why it operates a certain way, what experiments were tried, and why certain decisions were made.

As a team gets bigger and starts working on multiple facets of a product, the right hand stops knowing everything the left hand is doing. Code and systems can conflict if people aren't able to understand everything that's going on. Design docs serve as a single place that can be discussed, consulted and understood team-wide.

"CLAW TO FRAME" GEAR BOX

TECHNICAL SKETCH

GEAR BOX
For enabling two rotational degrees of freedom in claw

DESIGN GOALS

1. This component must properly enable TWO degrees of rotational freedom (see System Diagram)
 - Rotational DOF on claw allows for the flipping of caps
 - Rotational DOF on the gearbox itself allows for an upright position (this is a tradeoff; see "Tradeoffs")
2. The box must be as compact as possible to account for the 18x18x18 volume limit.
3. The box shouldn't allow the claw to sag; hence the use of pinion gears to stabilize the high strength gear.
4. The box should be positioned low enough to allow the claw to reach caps.

DESIGN DETAILS

Design Details
Requires

- Two 5x5x1 C-channels
- Transfer case with two bevel gears attached
- Hinge
- Four 12T pinion gears
- One 36T High strength gear
- Right angle rawled gussets
- Reasonably longlegged shafts

TRADEOFFS

Tradeoffs

1. The claw when in a down right position exceeds 18x18x18. Consequently, the upright position enabled by the hinge will be assumed before the start of the match. A rubber band adhering to the claw and IFA will most likely be needed to maintain the upright position.

SYSTEM DIAGRAM

DESIGN SUMMARY

Design Summary
The gearbox will be mounted to the lift and will allow a claw to retract when attached to the center high strength gear. The gearbox uses two bevel gears inside (what is called a "transfer") powered by two C-channels mounted together via struts and secured by a gusset to transfer the output of a motor at 90° to the center high strength gear. This is advantageous because there isn't adequate space to position a motor inside the lift stage.

BACKGROUND

Background
The O-Day Robotics team is tasked with placing "caps" on "posts" so that the claw of the team's adhesive "traces up". To achieve this a gearbox must rotate the claw to the elevator (lift) and enable TWO rotational degrees of freedom on it. This component will have contact with the elevator and the claw (Brandon Harrison)

Example Design Doc

This format acts as an important forcing function for an engineer to have conversations with other members of the team who may be impacted or have input. As a byproduct, communication flows better, bad ideas get weeded out sooner, and any negative surprises get nipped in the bud.

Background

Background on the problem you're solving. Why does it need to be solved? What other systems, features or products touch it? Who should be involved throughout?

Design goals

Requirements and goals of the project. This should also include numbers like traffic assumptions, usage, uptime requirements, etc.

System diagram

Diagram of all the binaries, databases and third-party services that this design touches. Having a visual helps many folks get the high-level picture and understand what's being impacted a lot more easily.

Design summary

Summary of the solution in a paragraph or two. This should not go on too long; it's meant to paint a quickly and easily accessible picture of what's being built.

Design details

Where the actual specifics of the design are listed out. This can include a variety of things from detailing subcomponents, code locations, testing strategies, internationalization, scaling tactics, etc.

Tradeoffs made

This is a great place for disclaimers on why certain choices are being made, what any negative implications might look like, limitations being taken into account, any technical debt that might be earned along the way, and changes that may need to be made in the future as a result.

DISCUSS.

Once a design document has been completed, it's time for its presentation.

First, choose a moderator and note taker for the discussion. They should be neutral parties with no emotional stake in what's be presented (but an above average aptitude for guiding and following conversations).

Then, send out the design doc along with a blank Google doc for questions and comments. The questions will be the agenda of the review and allow the optimization of people's time by diving straight into the issues.

“Everyone is welcome to sit and listen, but there's no talking in the meeting unless you've read the design doc. The moderator will jump in and cut off questions if the answer is in the doc. We do this so we don't waste people's time. We have X people in the room, so let's make the most of it.”

As a consequence of this rule, many people will skim through the doc in the minutes right before the meeting — better late than never.

“We're here to give the next 15 minutes to this product/feature with all of our attention. You'll appreciate people doing the same for you when you present a design doc in the future.”

“The moderator will keep things moving and might ask for specific discussions to be followed up offline. Again, this is to make the most of the group's time. Hold non-on-topic questions until the end, but feel free to jump in with any questions that pertain to the current discussion.”

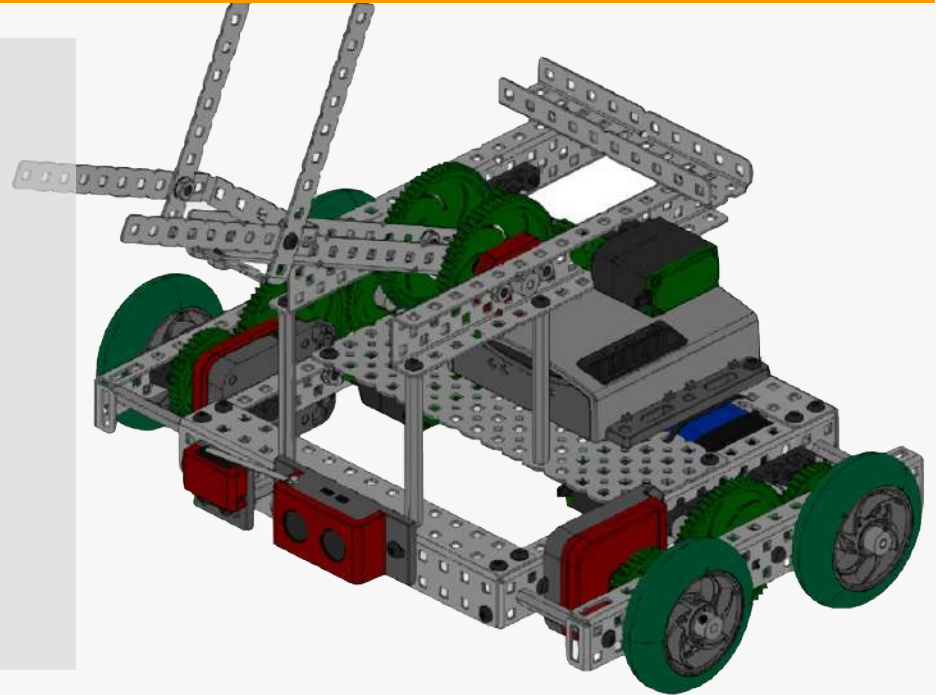
2. How to Build

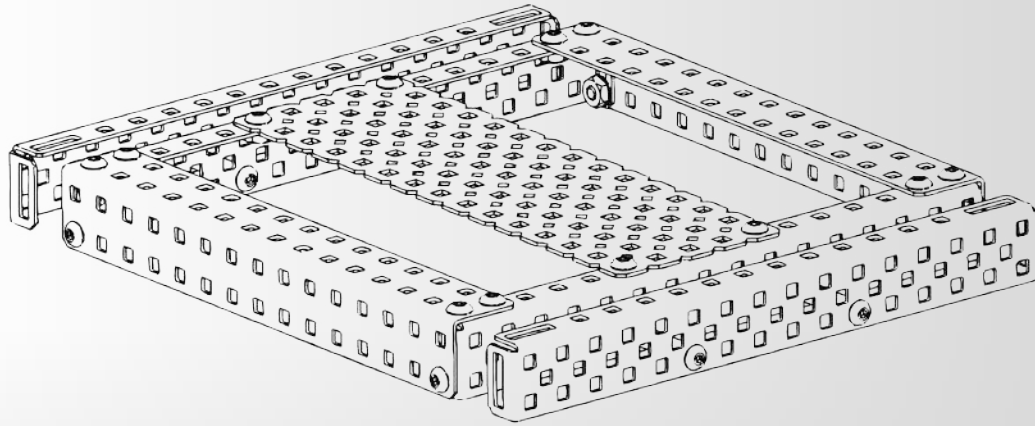
THE ROBOT DESIGN SYSTEM

Structure
Subsystem

Motion
Subsystem

Sensorial
Subsystem

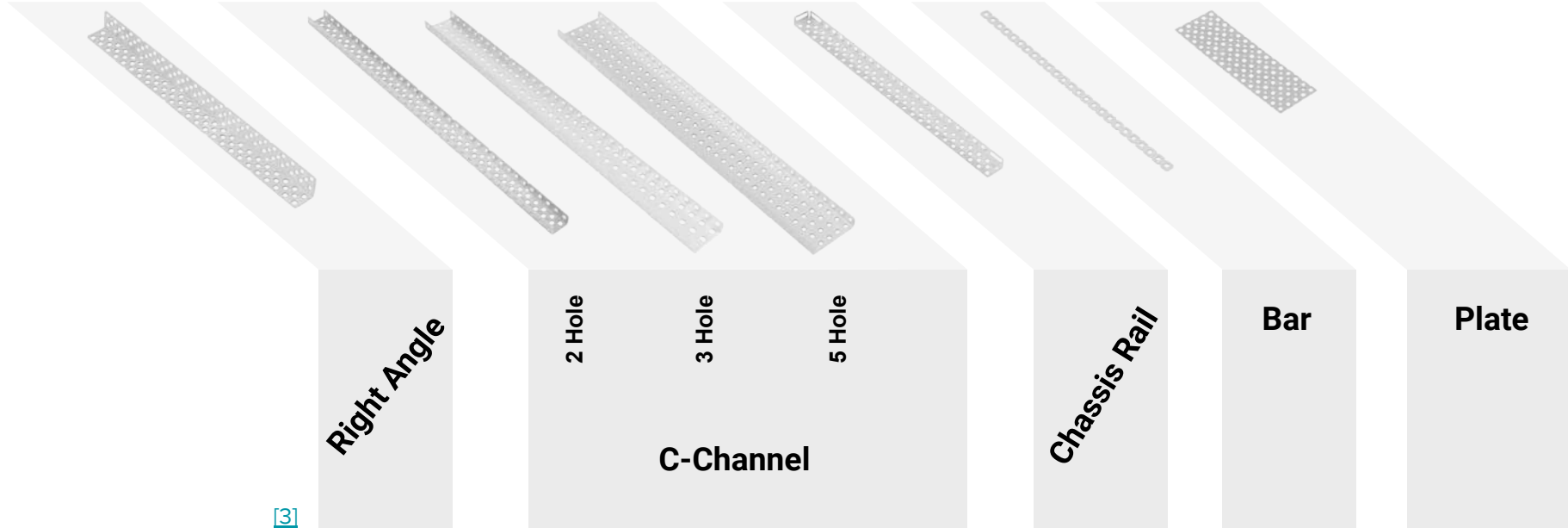


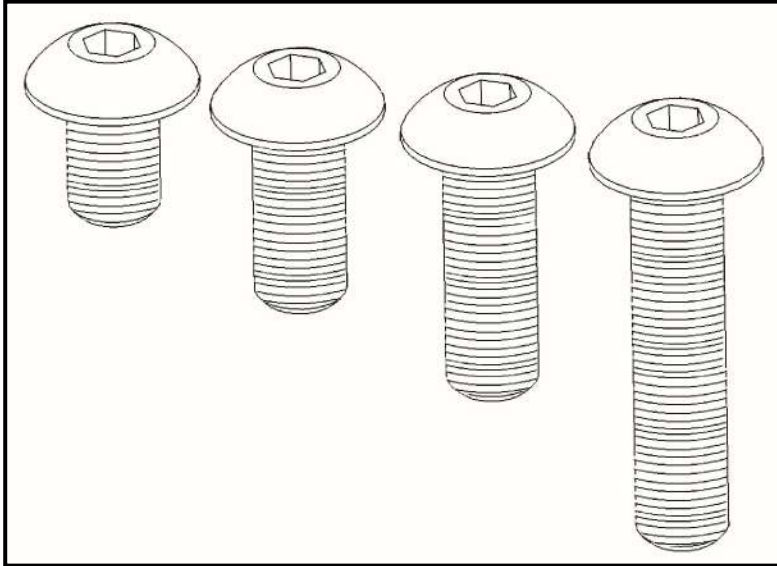


STRUCTURE.

The Structural Subsystem of the robot is responsible for physical support. It holds everything in place, and is, in effect, the durable “skeleton” of the robot to which all the other subsystems are attached.

The parts in the Structure Subsystem form the base of every robot. These parts are the “skeleton” of the robot to which all other parts are attached. This subsystem consists of all the main structural components in the Design System including all the metal components and hardware pieces. These pieces connect together to form the “skeleton” or frame of the robot.





8-32 Screws

Size 8-32

The primary screws used to build robot structure.

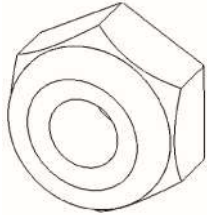
Size 6-32

Smaller screws which are used for special cases like mounting legacy motors and motion subsystem components.

Metal components can be directly attached together using 8-32 screws and nuts. Screws come in a variety of lengths and can be used to attach multiple thicknesses of metal together, or to mount other components onto structural pieces.

When using screws to attach things together, there are three types of nuts which can be used.

Nylock Nut

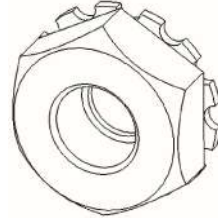


Have a plastic insert in them which will prevent them from unscrewing.

These nuts will not come off due to vibration or movement.

Always use on moving components.

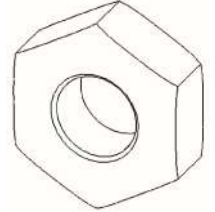
KEPS Nut



Have a ring of “teeth” on one side of them. These teeth will grip the piece they are being installed on.

These nuts are installed with the teeth facing the structure.

Regular Nut



Have no locking feature.

May loosen up over time, especially when under vibration or movement.

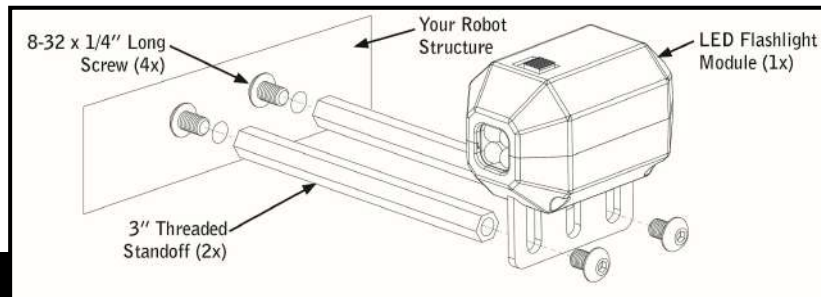
Very thin and can be used in some locations where it is not practical to use a Nylock or KEPS nut.

Standoffs

Components can also be offset from each other using 8-32 threaded standoffs. Standoffs come in a variety of lengths and work great for mounting components as well as for creating structural beams.

8-38 Standoffs

Typical use case for standoffs



When designing a robot's structure, it is important to think about making it strong and robust while still trying to keep it as lightweight as possible. Sometimes overbuilding can be just as detrimental as underbuilding.

The frame is the skeleton of the robot and should be designed to be integrated cleanly with the robot's other components. The overall robot design should dictate the chassis, frame, and structural design; not vice-versa.

Design is an iterative process; experiment to find out what works best for a given robot.

ADDITIONAL TOPICS IN STRUCTURE

Structure

Center of Gravity

Support Polygon

Achieving Stability

Exposure & Vulnerability

Center of Gravity

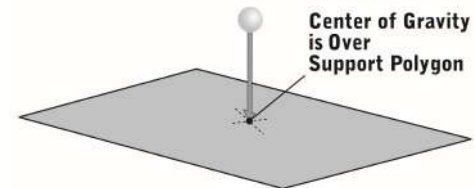
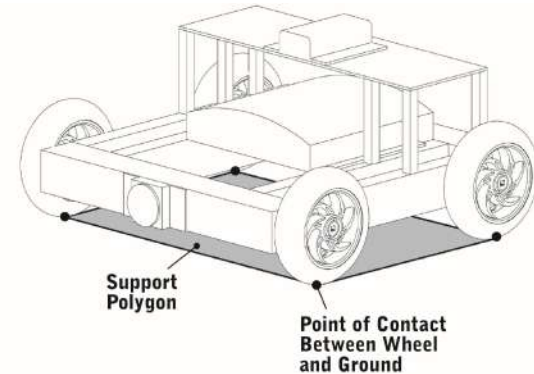
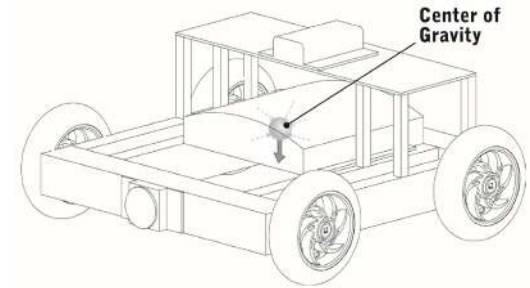
The “average position” of all the weight on the robot. Because it is an average of both weight and position, heavier objects count more than lighter ones in determining where the center of gravity is, and pieces that are farther out count more than pieces that are near the middle.

Support Polygon

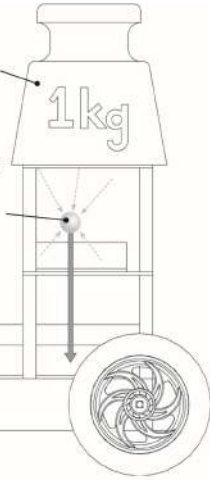
The imaginary polygon formed by connecting the points where your robot touches the ground (usually the wheels). It varies by design, but there is always one support polygon in any stable configuration.

Stability

The robot will be most stable when the center of gravity is centered over the support polygon.



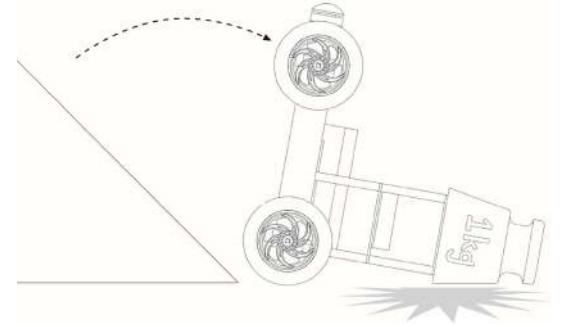
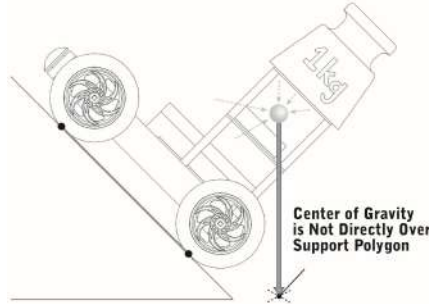
Weight Represents
the Bulk of a Gripper
or Similar Apparatus



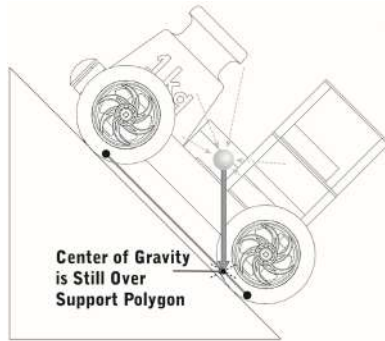
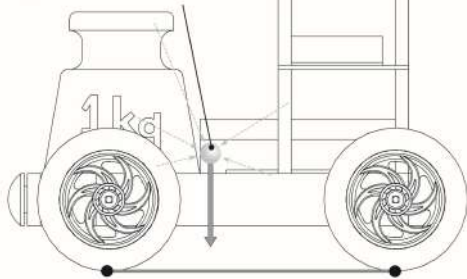
The Center of Gravity is higher
because of the
new weight added
to the top of the
robot

Inappropriate center of gravity

The robot's center of gravity is no longer over the support polygon.
The robot falls over as soon as it starts the ramp.



The Center of Gravity is
now lower because the
weight is mounted lower



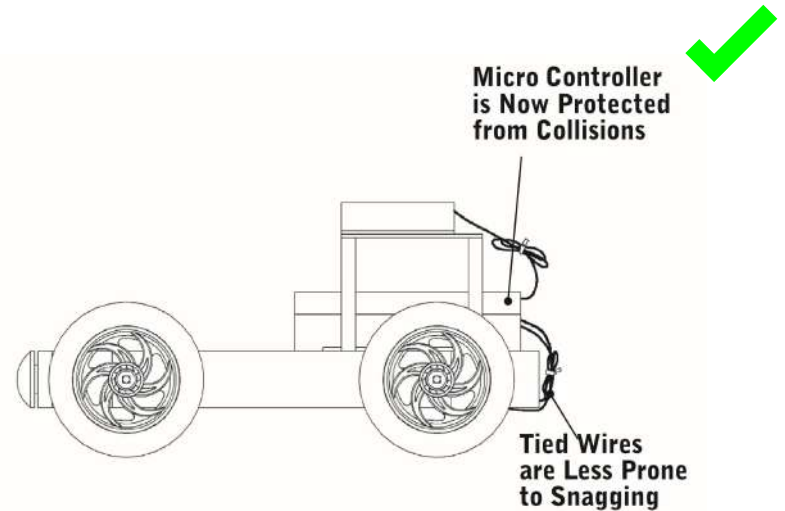
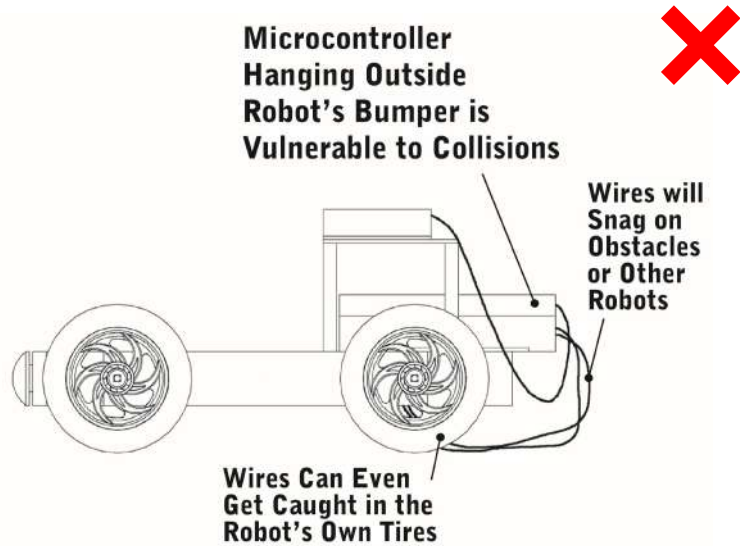
Appropriate center of gravity

The robot's center of gravity
is closer to the ground.

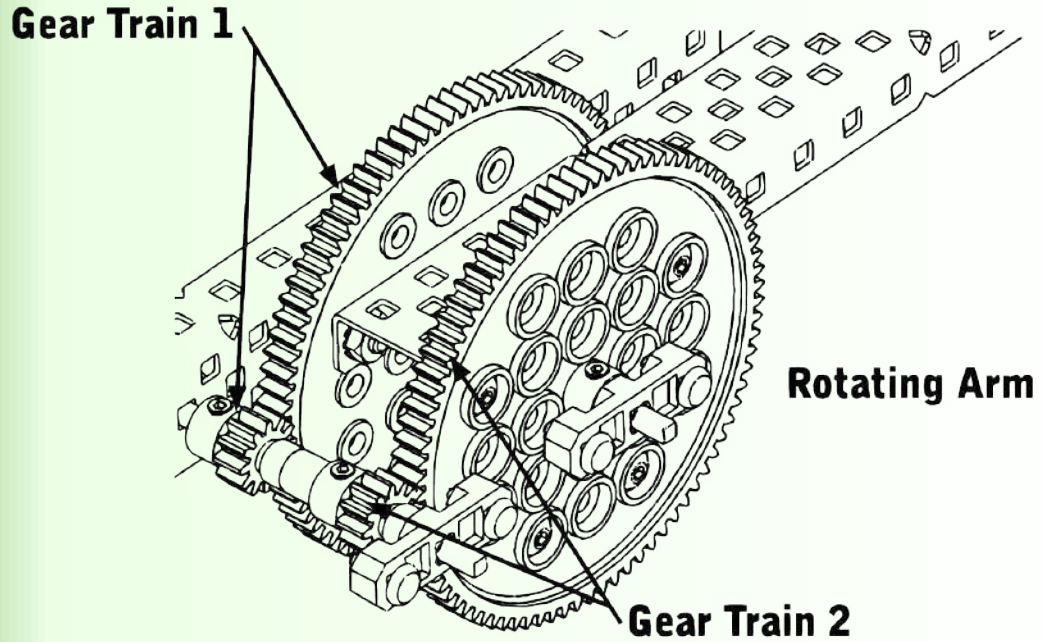


Exposure & Vulnerability

Robot components that can be damaged are well shielded and inside robot structure. Route wires inside the robot and away from all moving components.

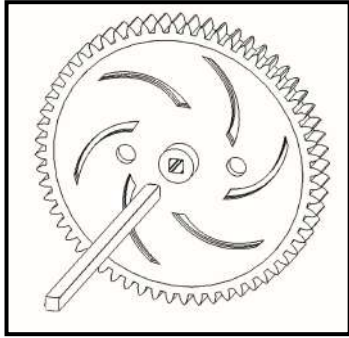


The Motion Subsystem of the robot is responsible for exactly that, motion. It includes both the motors that generate motion, and the wheels and gears that transfer and transform that motion into the desired forms. With the Structural Subsystem as the robot's skeleton, the Motion Subsystem is its muscle.



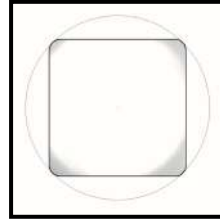
MOTION.

The Square Shaft

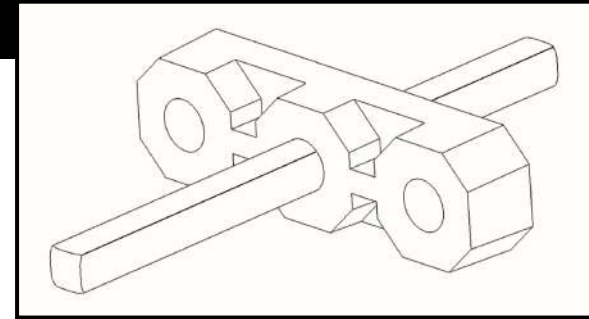


Gear and Shaft

Most of the motion components use a square hole in their hub which fits tightly on square shafts. This square hole – square shaft system transmits torque.



Delrin Bearing

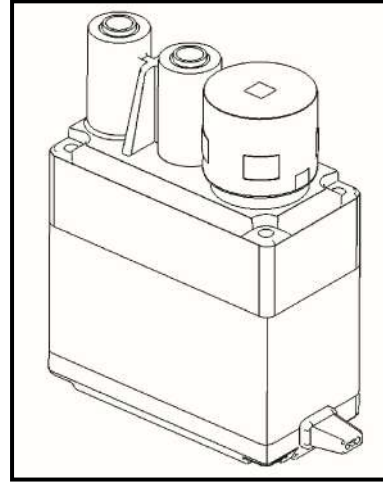


The square shaft has rounded corners which allow it to spin easily in a round hole. This allows the use of simple bearings made from Delrin (a slippery plastic). The Delrin bearing will provide a low-friction piece for the shafts to turn in.

Actuators

The key component of any motion system is an actuator. An *actuator* is something which causes a mechanical system to move. There are two types of actuators: *motors* and *servos*. Both of them have different use cases.

Each motor and servo comes with a square socket in its face, designed to connect to the square shafts. By simply inserting a shaft into this socket it is easy to transfer torque directly from a motor into the rest of the Motion Subsystem

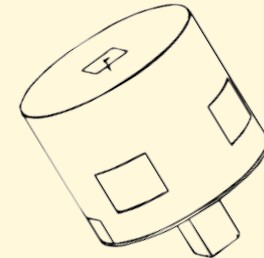


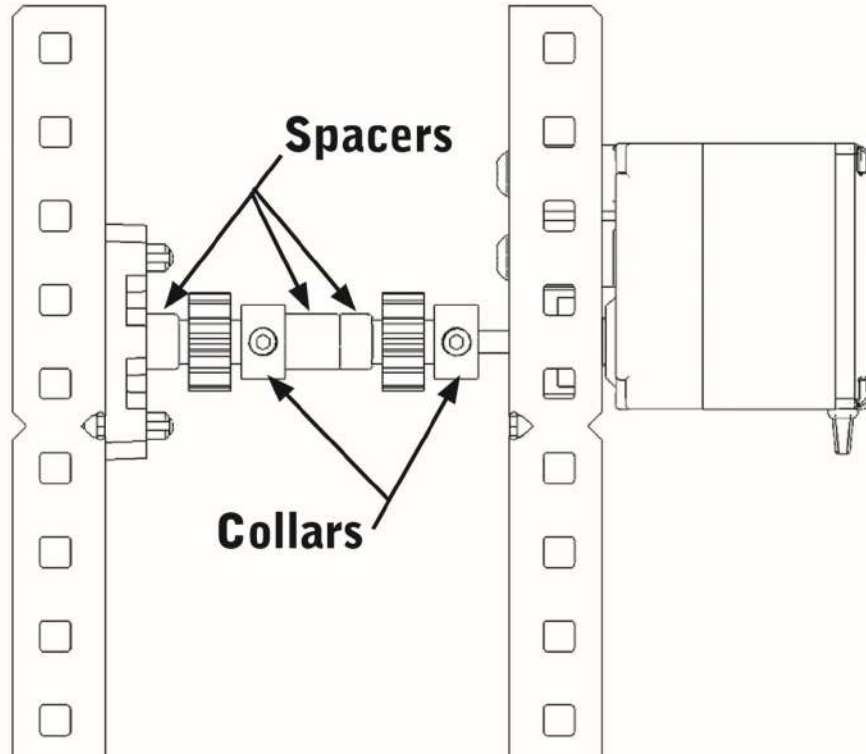
An Actuator (Both appear the same)

Motor	Servo
A standard motor spins its shaft around and around for as long as it's receiving power.	A servo rotates its shaft to a set angular position, between 0 and 120 degrees and holds it there for as long as it's receiving power.

PROTECTING ACTUATORS FROM ABNORMAL SHOCK-LOADS

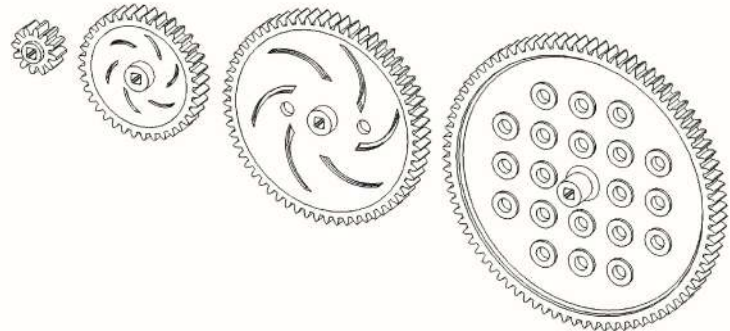
Gears can break in some applications when an actuator is under significant load, over a short duration of time (a shock-load). Equipping the actuator with a **clutch** will prevent this from happening when an abnormal shock-load is applied. The clutch will absorb some of the energy in these situations by “popping” and giving way. This will protect the actuator.





The Motion Subsystem also contains parts designed to keep pieces positioned on a shaft. These pieces include spacers and collars. Collars slide onto a shaft, and can be fastened in place using a setscrew. Before tightening the setscrew, it is important to slide the Collars along the square shafts until they are next to a fixed part of the robot so that the collar prevents the shaft from sliding back and forth

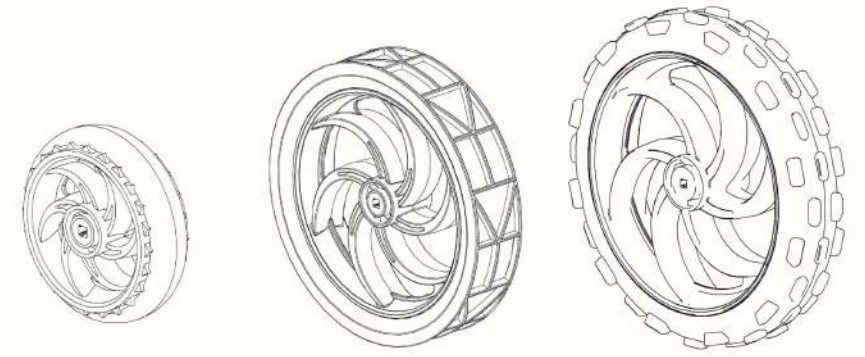
Gears & Wheels









The primary way to transfer motion is through the use of spur gears. Spur gears transfer motion between parallel shafts, and can also be used to increase or decrease torque through the use of gear ratios.

The last step in the drive train (series of gears transferring torque for the purpose of mobility), after the motors and gears, is the wheels.

Bigger tires give you slower acceleration, while smaller tires give you faster acceleration.



2.75" Wheel	4" Wheel	5" Wheel
Acceleration  Faster	Acceleration  Medium	Acceleration  Slower
Traction  Slippery	Traction  Medium	Traction  Heavy Grip

Speed vs. Torque

Gear Ratios

Compound Gear Ratios

Gear Ratios With Non-Gear Systems

Idler Gears

Linear Motion

Lifts

Speed vs. Torque

Because a motor can only generate a set amount of power, there is an inherent trade-off between **Torque**—the force with which a motor can turn a wheel—and **Speed**—the rate at which a motor can turn a wheel.

The exact configuration of torque and speed is usually set using gears. By putting different combinations of gears between the motor and the wheel, the speed-torque balance will shift.

Gear Ratios

Gear ratio can be thought of as a “multiplier” on torque and a “divider” on speed. A gear ratio of 2:1 would yield twice as much torque as a gear ratio of 1:1, but only half as much speed.

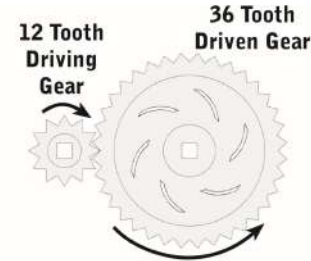
Gear ratio is a ratio of the number of teeth on a “driven” gear to the number of teeth on its “driving” gear.



$$\text{Gear Ratio} = 12:12 = 1:1$$

Torque  1x

Speed  1x



$$\text{Gear Ratio} = 36:12 = 3:1$$

Torque  3x

Speed  1/3x



$$\text{Gear Ratio} = 12:36 = 1:3$$

Torque  1/3x

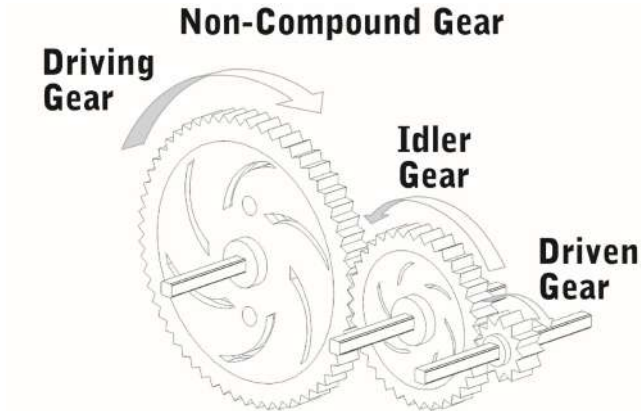
Speed  3x

Compound Gear Ratios

Compound gears are formed when two or more gears are on the same axle. In a compound gear system, there are multiple gear pairs. Each pair has its own gear ratio, but the pairs are connected to each other by a shared axle.

The resulting compound gear system still has a driving gear and a driven gear, and still has a gear ratio (now called a “compound gear ratio”).

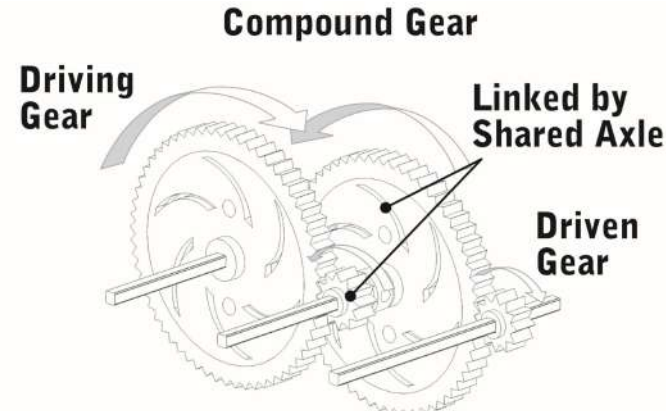
The compound gear ratio between the driven and driving gears is then calculated by multiplying the gear ratios of each of the individual gear pairs.



$$\text{Gear Ratio} = 60:12 = 5:1$$

Torque  1/5x

Speed  5x



Compound Gear Ratio:

$$12:60 \times 12:60 = 1:5 \times 1:5 = 1:25$$

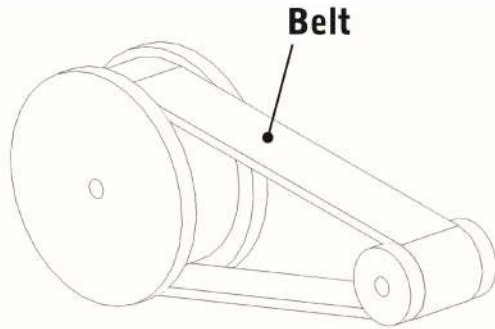
Torque  1/25x

Speed  25x

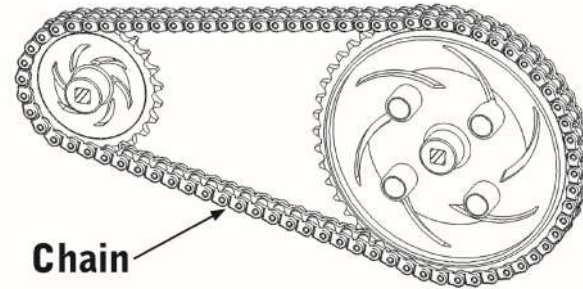
Gear Ratios With Non-Gear Systems

Belt or chain drives are often preferred over gears when torque is needed to be transferred over long distances.

When the number of teeth cannot be determined, gear ratio can be measured by the number of rotations on the driven and driving axles.



Belt Drive

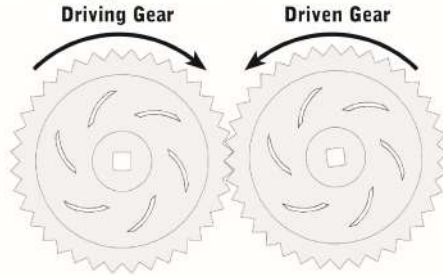


Chain Drive

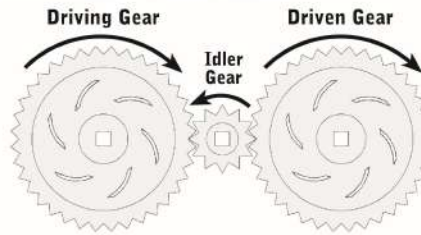
Idler Gears

Gears can be inserted between the driving and driven gears. These are called idler gears, and they have no effect on the robot's gear ratio because their gear ratio contributions always cancel themselves out.

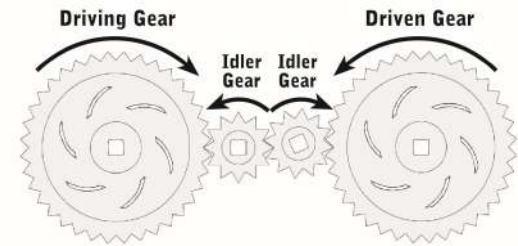
No Idler – Opposite Direction



One Idler – Same Direction



Two Idlers – Opposite Direction

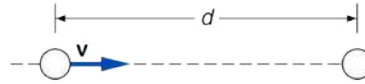


However, idler gears do reverse the direction of spin. Normally, the driving gear and the driven gear would turn in opposite directions. Adding an idler gear would make them turn in the same direction. Adding a second idler gear makes them turn in opposite directions again.

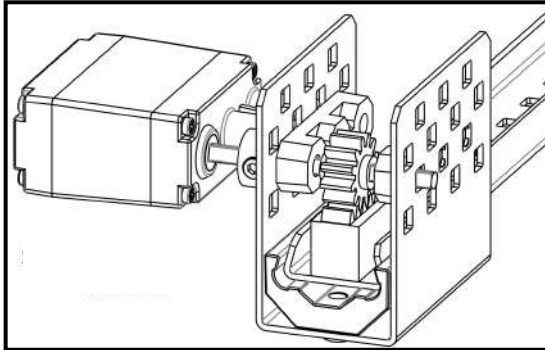
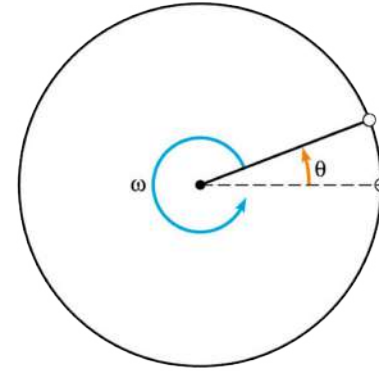
Linear Motion

Linear motion involves an object moving from one point to another in a straight line.
Rotational motion involves an object rotating about an axis.

Linear motion



Rotational motion



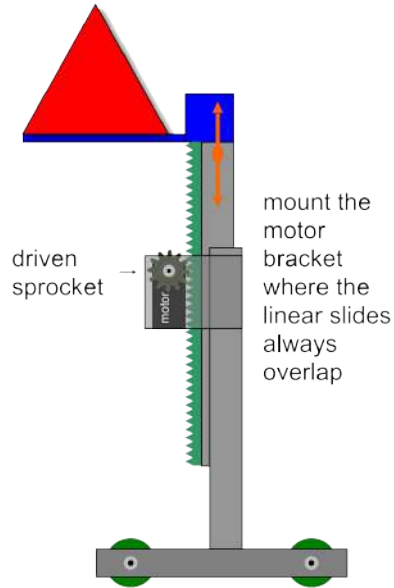
Outer Linear Slide

Using a rack and pinion is one of the best ways to articulate a linear movement. This is known as a “rack and pinion linear slide.”

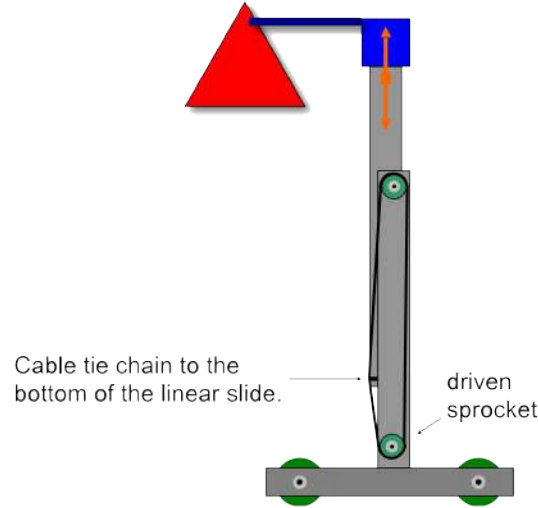
Lifts

A lift is a device that extends upwards.

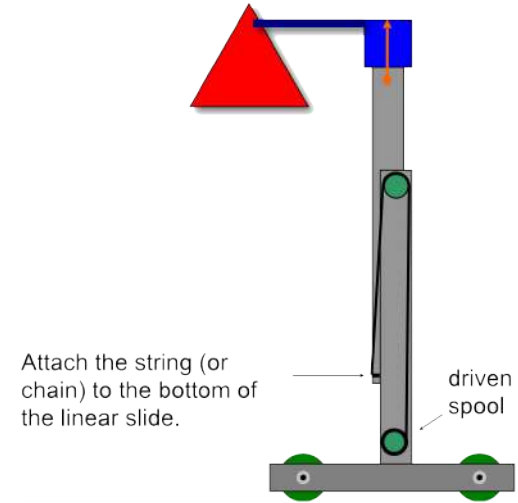
The Extension Lift is one type of lift and can be achieved different ways:



Rack & Pinion

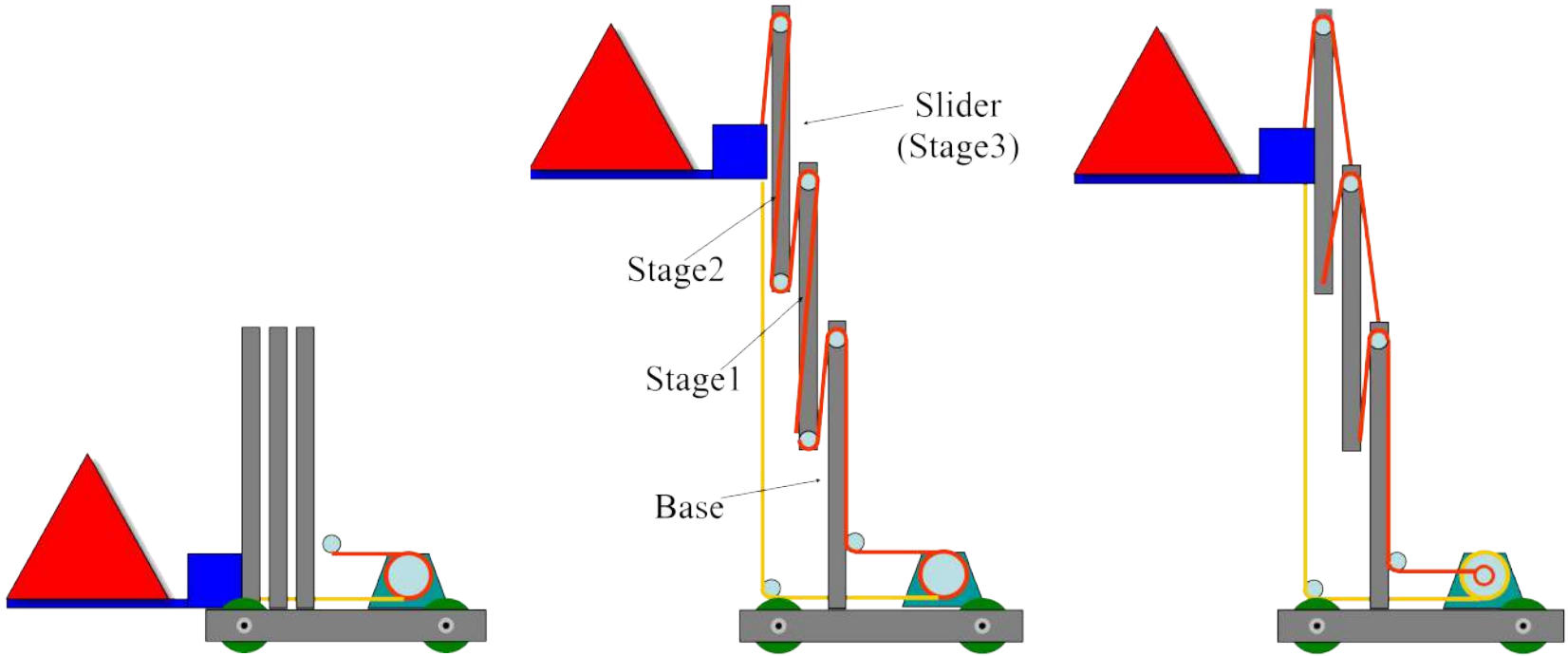


Chain



Winch

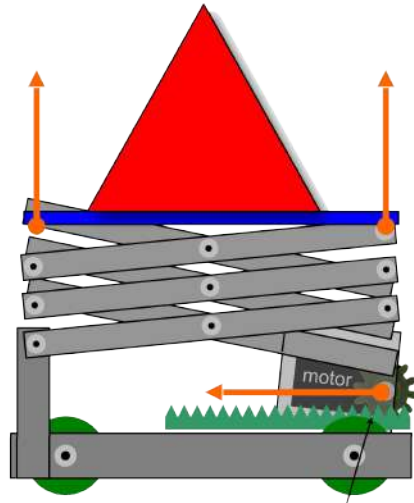
Extension lifts can also be multi-stage to achieve greater height.



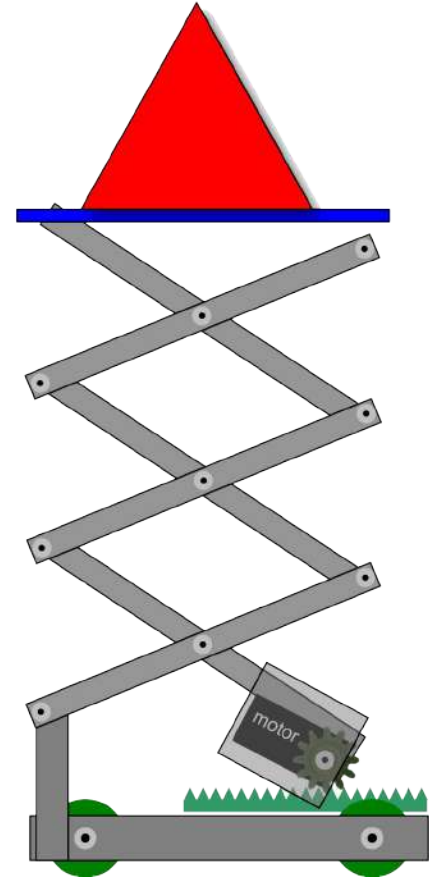
Continuous Rigging

Cascade Rigging

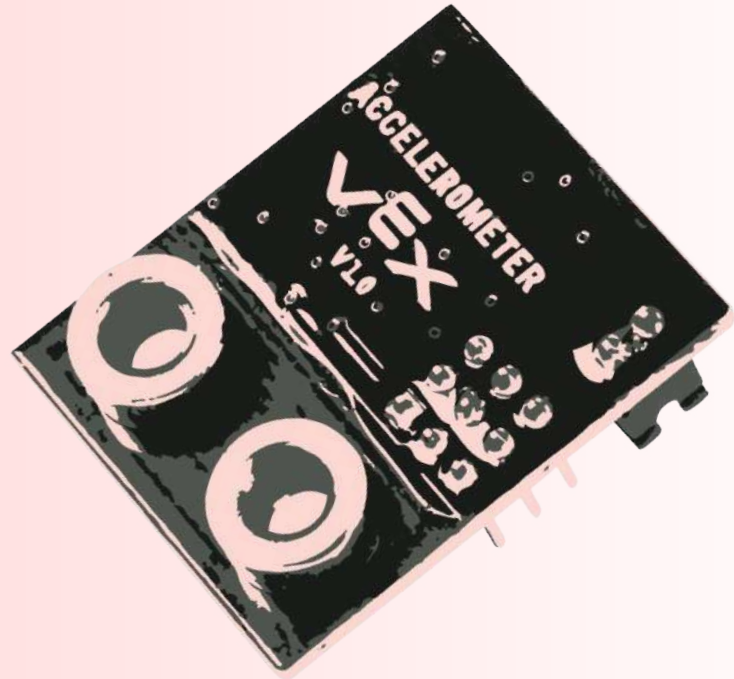
The Scissor Lift is another type of lift. When the bottom of the scissors is pulled together it extends upwards. In this example a rack and pinion pulls the bottom of the scissors together.



driven gear



The Sensor Subsystem gives the robot the ability to detect various things in its environment. The sensors are the “eyes and ears” of the robot, and can even enable the robot to function independently of human control.



SENSORIAL.

Analog vs. Digital

Sensorial

Analog

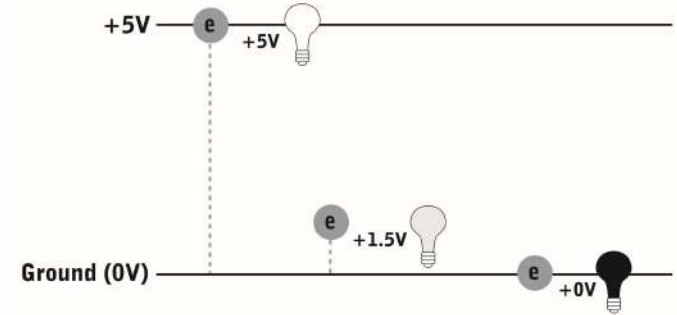
Analog sensors communicate with the Microcontroller by sending it an electrical voltage along a wire. By measuring where the sent voltage falls between zero and maximum voltage, the Microcontroller can interpret the voltage as a numeric value for processing. Analog sensors can therefore detect and communicate any value in a range of numbers.

States

Range of numerical values

Weakness

Difficulty sending and maintaining an exact, specific voltage on a wire in a live circuit. Less reliable than Digital.



Digital

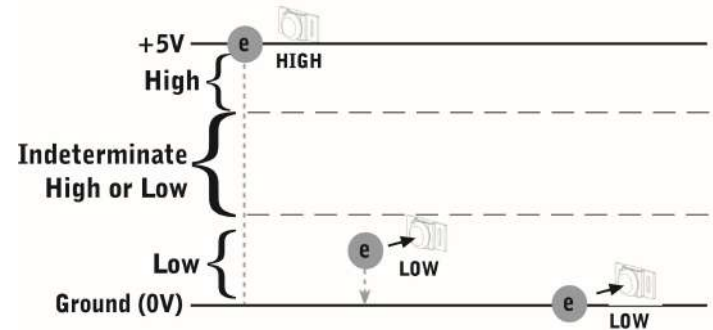
A digital sensor sends a voltage, just like an analog sensor, but instead of sending a voltage between zero and maximum, it will send only zero OR maximum. If the Microcontroller detects a voltage that is above a guaranteed Low or below a guaranteed High the results cannot be determined, it can be reported as a High or Low.

States

HIGH or LOW

Weakness

Can only indicate two values rather than a whole range.

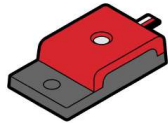


Primitive vs. “Smart” Hardware

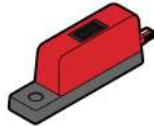
Starting in 2018, the VEX robot system has been shifting away from a primitive, low-level hardware design in turn for hardware that is more sophisticated. Consequently, there is a line between hardware.

Primitives

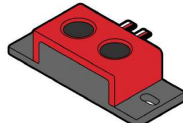
The smallest most fundamental unit of hardware of a specific function in a robot.



Light Sensor



Infrared Sensor



Ultrasonic Sensor



Potentiometer



Shaft Encoder



393 Motor

“Smart” Hardware

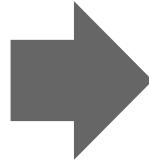
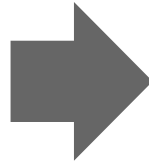
Term used by the VEX robot system for hardware that uses the RJ-11 interface. This type hardware is more complex as it uses an collection of primitives to serve a more broad function.



Vision Sensor



“Smart” Motor



The Microcontroller

Sensorial

V5 Robot Brain

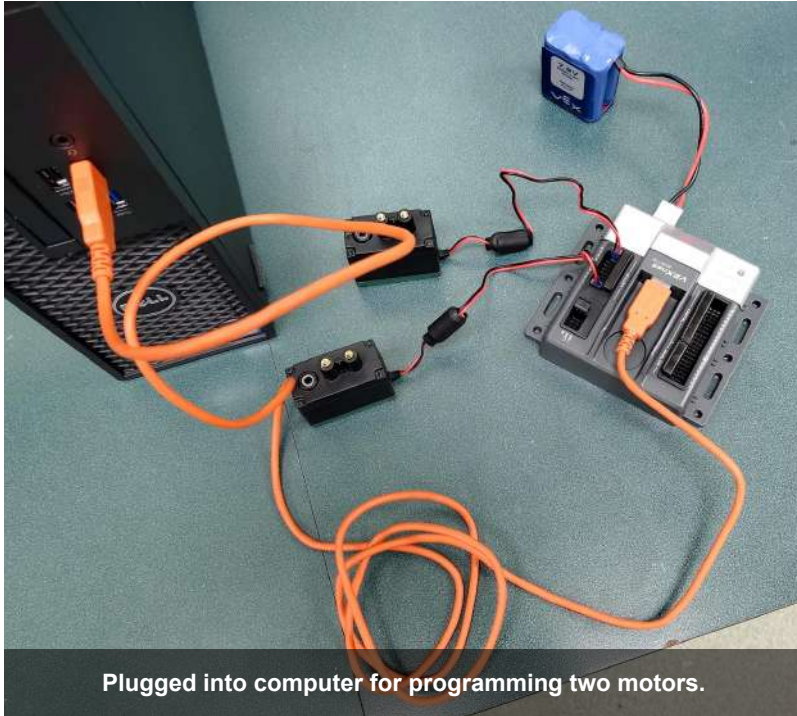


Cortex Microcontroller



Motor Ports	Use any of the 21 Smart Ports	10
Tether Ports	Use any of the 21 Smart Ports	remove radios, use USB cable
Digital Ports	Use any of the 8 built-in 3-Wire Ports	12
Analog Ports	Use any of the 8 built-in 3-Wire Ports	8
VEXos Processor	One Cortex A9 at 667 MHz Two Cortex M0 at 32 MHz each One FPGA1	ARM7
User Processor	One Cortex A9 1333 Million Instructions per second (MIPS)	Cortex M3 90 MIPS
Ram	128 MBytes	0.0625 MBytes
Flash	32 MBytes	0.375 MBytes
User Program Slots	8	1
USB	2.0 High Speed (480 Mbit/s)	Full Speed (12 Mbit/s)
Color Touch Screen	4.25", 480 x 272 pixels, 65k colors	
Expansion	microSD up to 16 GB, FAT32 format	
Wireless	VEXnet 3 and Bluetooth 4.2	VEXnet 2
System Voltage	12.8 V	7.2 V

The *Microcontroller* is the “brain” of the robot. It’s a fully programmable device, and is what enables motors, sensors, an LCD screen, and remote control signals to be connected. One of two can be used in a single robot.



Plugged into computer for programming two motors.

Inside of the Cortex, there are two separate processors; a user processor handles program execution, and a master processor controls lower-level operations, like motor control and radio communication. Downloading the written programs to the Microcontroller uses a USB A-to-A cable as shown on the left.

Wiring Up the Cortex Microcontroller

Sensorial

Analog Inputs

Used by any sensors that provide a range of values. Examples include:

- Potentiometers
- Light sensors
- Line followers
- Accelerometers

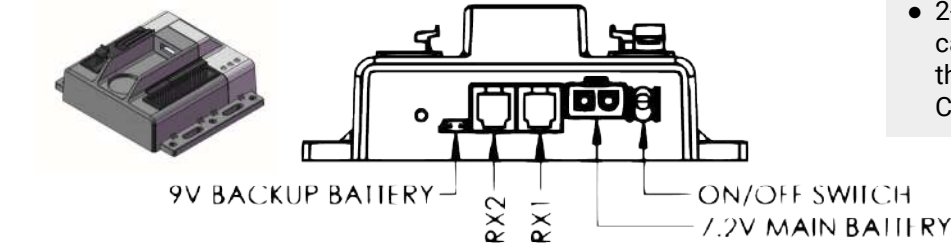
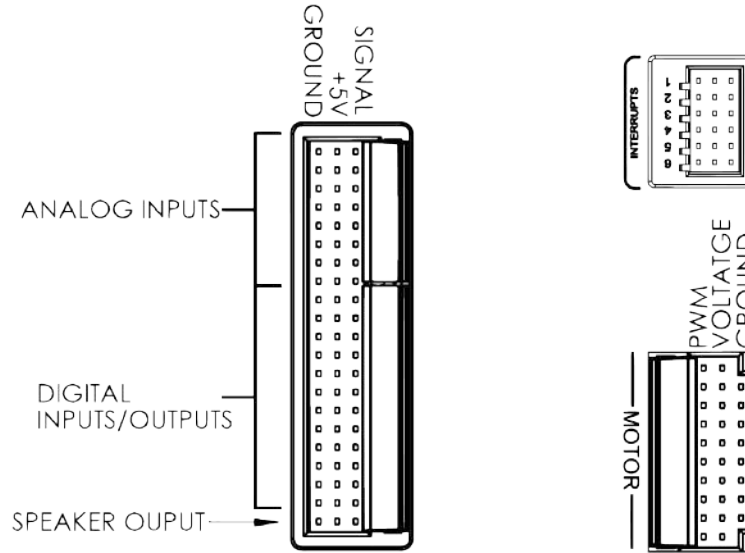
Digital Inputs/Outputs

Digital ports are available for digital input signals. Examples include:

- Bumper switches
- Limit switches
- Ultrasonic range finders
- Optical shaft encoders.

Speaker Output

For connecting a single speaker. Enables the robot to play tones, sounds and wave (.wav) sound files.



Interrupts

Digital inputs designed for high priority signals that need immediate attention from the Microcontroller. These are used with some of the advanced sensors of the Robot Design System, such as the following:

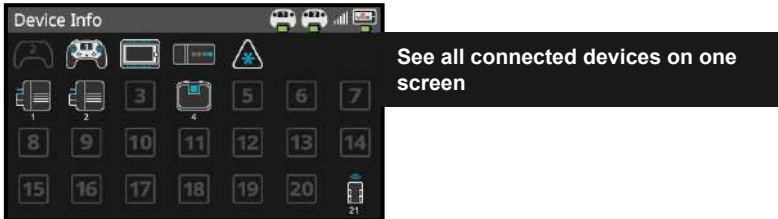
- Ultrasonic Range Finder
- Quadrature Shaft Encoder

Motor Outputs

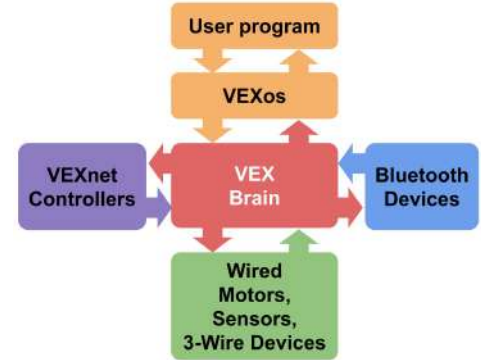
- 2-wire motors and flashlights can be directly connected and controlled in ports 1 and 10.
- 3-wire motors and servos can be directly connected and controlled in ports 2 through 9.
- 2-wire motors and flashlights can be connected to ports 2 through 9 using a Motor Controller 29.

The V5 Robot Brain

Sensorial



V5 uses a technology called "Centralized Intelligence", which provides the user processor with all sensor information. All "Smart" Sensors have their own processor, which allows them to simultaneously collect and process data as fast as possible. New information is instantaneously sent to the user processor's high speed local

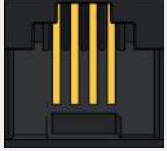


RAM without interrupting the processor. Each time a line of code calls for sensor data as a user's program runs, such as motor position, the most recent calculation is instantly accessed from memory.

VEXos CPU	User CPU	Shared Ram	Shared Flash	microSD
Cortex A9	Cortex A9	128MB	32MB	Up to
667 MHz	667 MHz	533 MHz	800 MHz	16 GB

Wiring Up the V5 Robot Brain

RJ-11



The V5 Robot Brain has 21 Smart Ports available which enables the use of “Smart” hardware. Each of these are equipped with a digital circuit breaker, called an eFuse, that allow for short circuit protection without limiting motor performance.

3-Pin Ports

3-Wire ports are multi-purpose. Any 3-Wire port can be a digital input, digital output, analog input, or PWM motor control. This enables the use of primitives:

- Bumper switches
- Limit switches
- Potentiometers
- Shaft Encoders
- Ultrasonic Sensors
- Light Sensors
- Infrared Sensors
- Accelerometers
- Gyroscopes
- 393 Motors

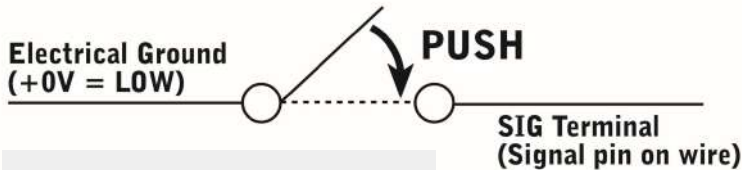
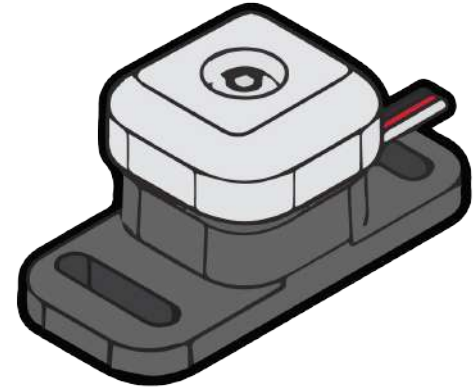


Bumper Switch

Sensorial

The bumper sensor is a physical switch. It tells the robot whether the bumper on the front of the sensor is being pushed in or not.

When the switch is not being pushed in, the sensor maintains a digital HIGH signal on its sensor port. This High signal is coming from the Microcontroller. When an external force (like a collision or being pressed up against a wall) pushes the switch in, it changes its signal to a digital LOW until the switch is released.

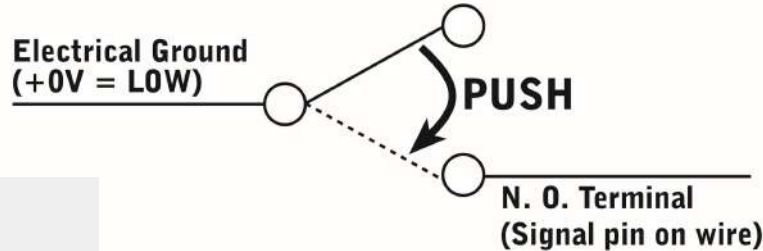
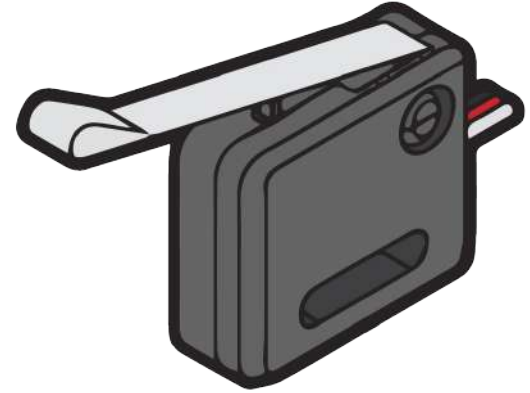


Pressed = 1 (or true)

Unpressed = 0 (or false)

The limit switch sensor is a physical switch. It can tell the robot whether the sensor's metal arm is being pushed down or not.

When the limit switch is not being pushed in, the sensor maintains a digital HIGH signal on its sensor port. This High signal is coming from the Microcontroller. When an external force (like a collision or being pressed up against a wall) pushes the switch in, it changes its signal to a digital LOW until the limit switch is released.

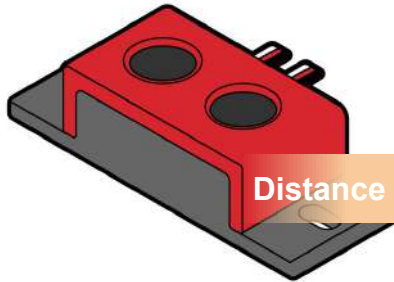


Pressed = 1 (or true)

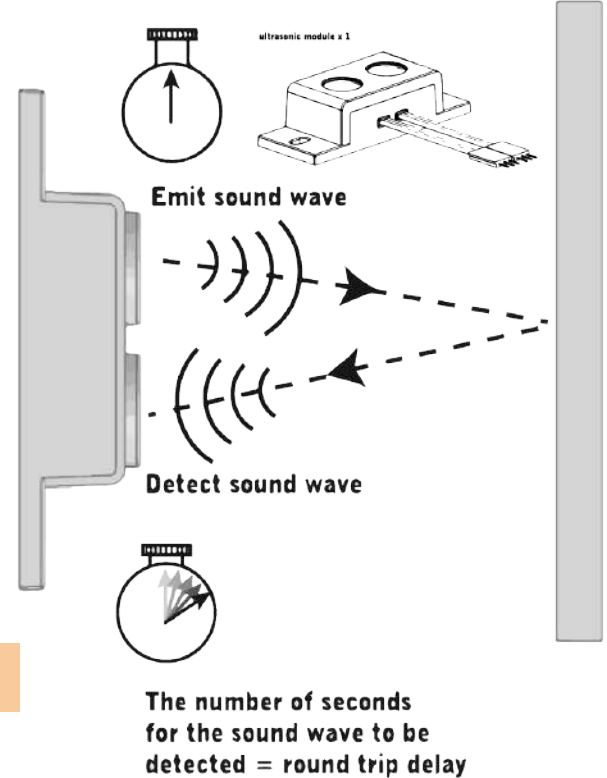
Unpressed = 0 (or false)

“Ultrasonic” refers to very high-frequency sound – sound that is higher than the range of human hearing. Sonar, or “Sound Oriented Navigation And Ranging,” is an application of ultrasonic sound that uses propagation of these high frequency sound waves to navigate and detect obstacles.

The ultrasonic sensor determines the distance to a reflective surface by emitting high-frequency sound waves and measuring the time it takes for the echo to be picked up by the detector.



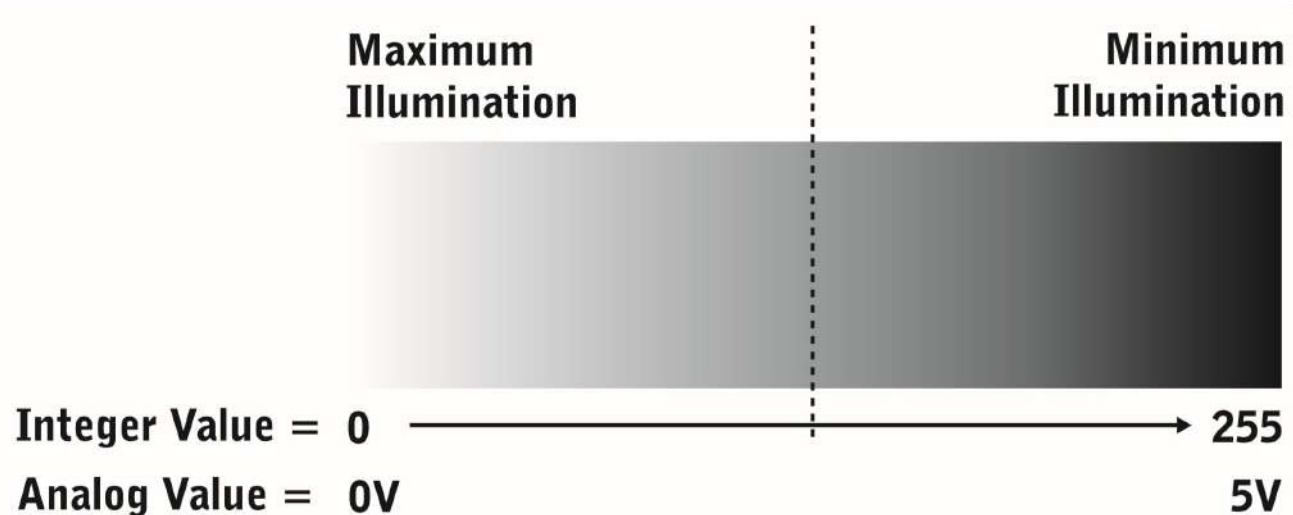
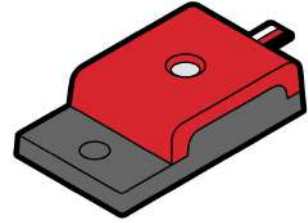
$$\text{Distance to object} = \frac{1}{2} (\text{speed of sound}) \times (\text{round trip delay})$$



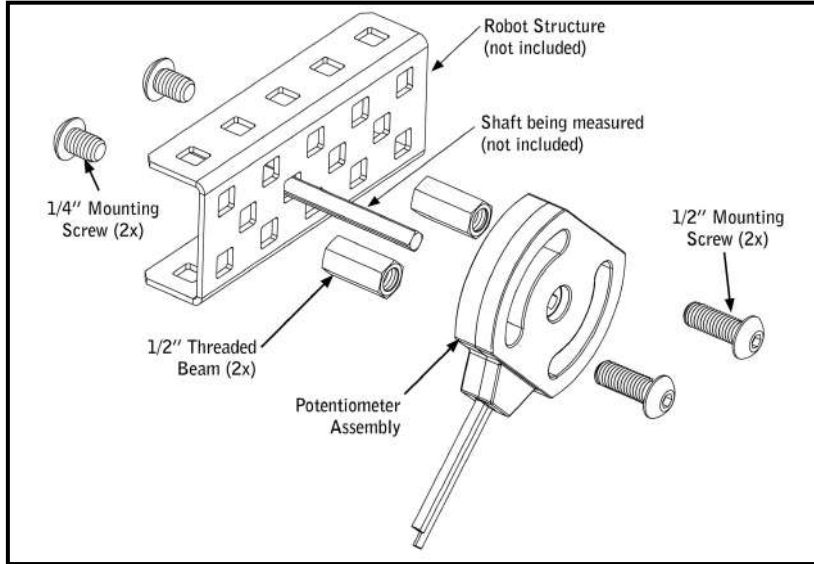
Light Sensor

Sensorial

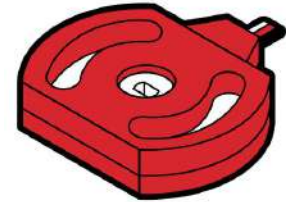
The light sensor uses a Cadmium Sulfoselenide photoconductive photocell, or CdS cell for short. The light sensor does what you think; it detects changes in light level. A low value (around 0) corresponds to very bright light, and a high value (around 255) corresponds to darkness.



Potentiometer



The Potentiometer is used to measure the angular position of the axle or shaft passed through its center. The center of the sensor can rotate roughly 265 degrees and outputs values ranging from 0-1023 to the Microcontroller. This measurement can help to understand the position of robot arms, or other mechanisms.



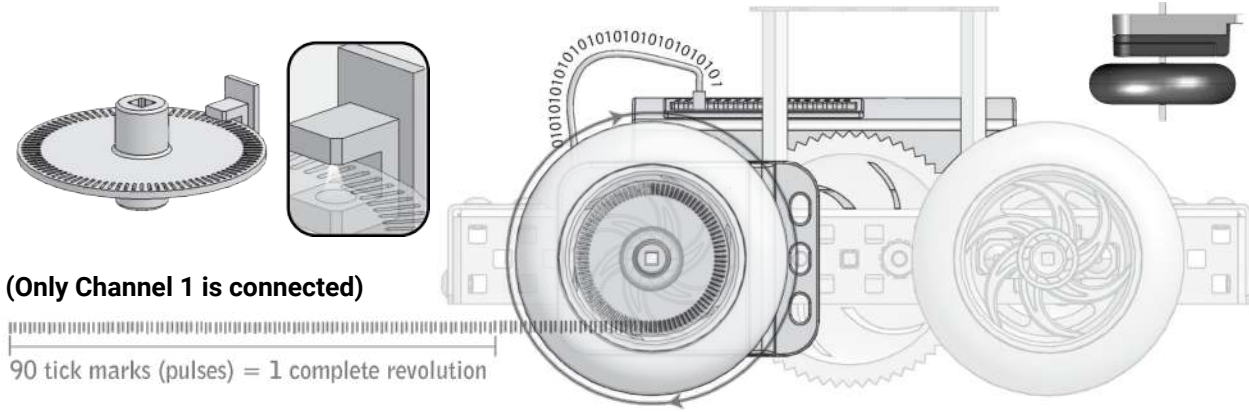
Instructions for mounting the Potentiometer

CAUTION

When mounting the Potentiometer on your robot, ensure that the range of motion of the rotating shaft does not exceed that of the sensor. Failure to do so may result in damage to your robot and the Potentiometer. The arcs provide flexibility for the orientation of the Potentiometer, allowing the full range of motion to be utilized more easily.

Optical Shaft Encoder

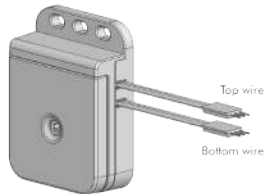
Basic Optical Shaft Encoders are commonly used for position and motion sensing. Basically, a disc with a pattern of cutouts around the circumference is positioned between an LED and a light detector; as the disc rotates, the light from the LED is blocked in a regular pattern. This pattern is processed to determine how far the disc has rotated. If the disc is then attached to a wheel on a robot, it is possible to determine the distance that wheel traveled, based on the circumference of the wheel and the number of revolutions it made.



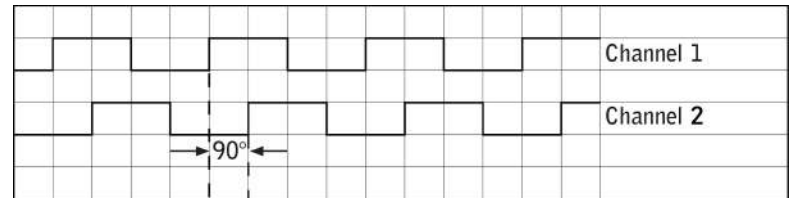
The Encoder contains two optical sensors making it quadrature. This allows the sensor to detect if the internal disk is spinning clockwise or counterclockwise and increases the resolution to 360 counts per revolution (2 count intervals). Two output channels (wires) are needed to transmit its sensor data.

(Only Channel 1 is connected)

90 tick marks (pulses) = 1 complete revolution

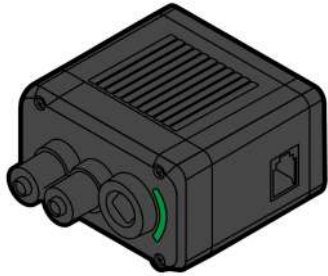


The term quadrature refers to the situation where there are two output channels; that is, two square waves 90 degrees out of phase with each other, being outputted by the unit.



V5 “Smart” Motor

Sensorial

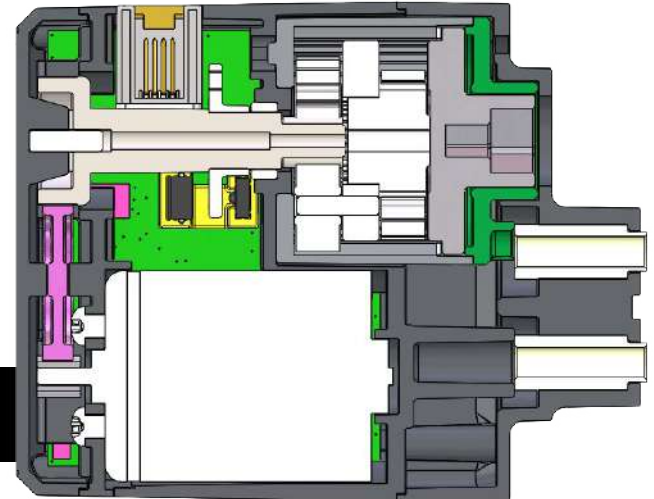


Inside the motor are gears, an encoder, modular gear cartridge, circuit board, and thermal management components. Users can control the motor’s direction, speed, acceleration, position, and torque limit. The motor’s internal circuit board has a full H-Bridge and its own Cortex M0 microcontroller to measure position, speed, direction, voltage, current, and temperature.



Feedback data in motor dashboard

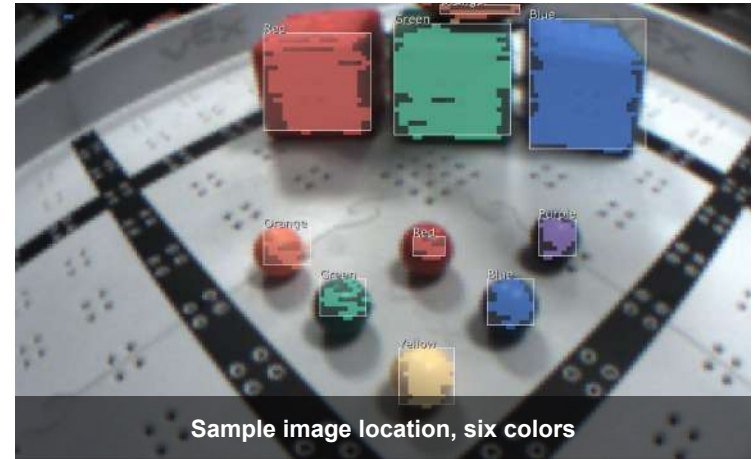
Cross section of a V5 “Smart” Motor

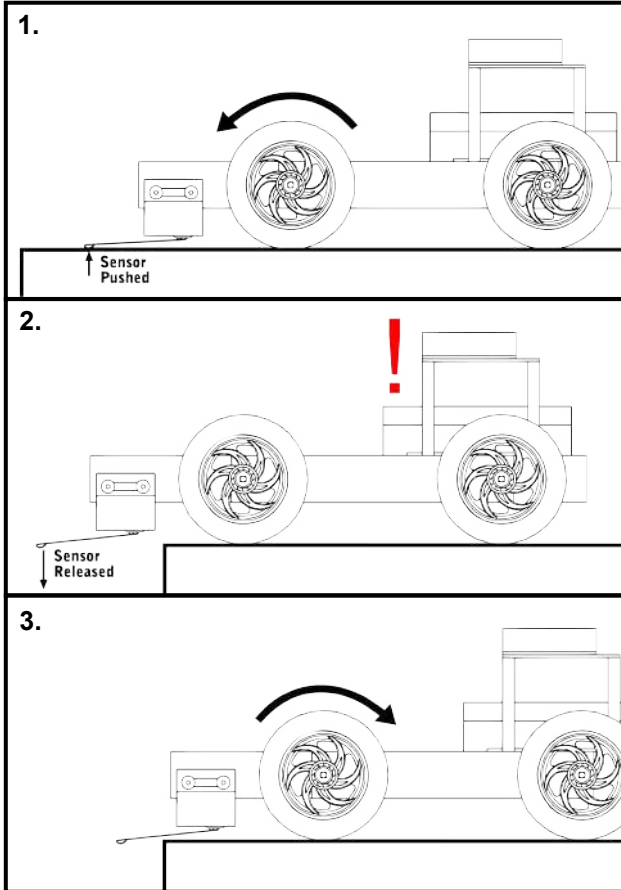




At its most basic mode, the sensor tells you where a colored object is located. The location's X value gives you the right and left position. When the camera is tilted down, the Y value gives you the distance to the object, with a little basic trigonometry on your part. The Vision Sensor combines a dual ARM Cortex M4+M0 processor, color camera, WiFi, and USB into a single smart sensor.

The sensor can be trained to locate objects by color. Every 200 milliseconds, the camera provides a list of the object found matching up to eight unique colors. The object's height, width, and location is provided. Multi-colored objects can also be programmed, allowing color codes to provide new information to the robot.





1. SENSE

The robot needs the appropriate hardware to sense important things about its environment, like the presence of obstacles or navigation aids.

2. PLAN

The robot needs to take the sensed data and figure out how to respond appropriately to it, based on a pre-existing strategy. This pre-existing strategy is called "behavior." Behavior is added by programming the Microcontroller.

3. ACT

Finally, the robot must actually act to carry out the actions that the plan calls for.

Robotic Engineers use the Sense-Plan-Act concept to build robust robots that can operate in numerous environments, independent of human control.

```
task main()
{
  bMotorReflected[port2]=1;
  while(true)
  {
    if (SensorValue (bumper)==0)
    {
      motor[port3]=127;
      motor[port2]=127;
    }

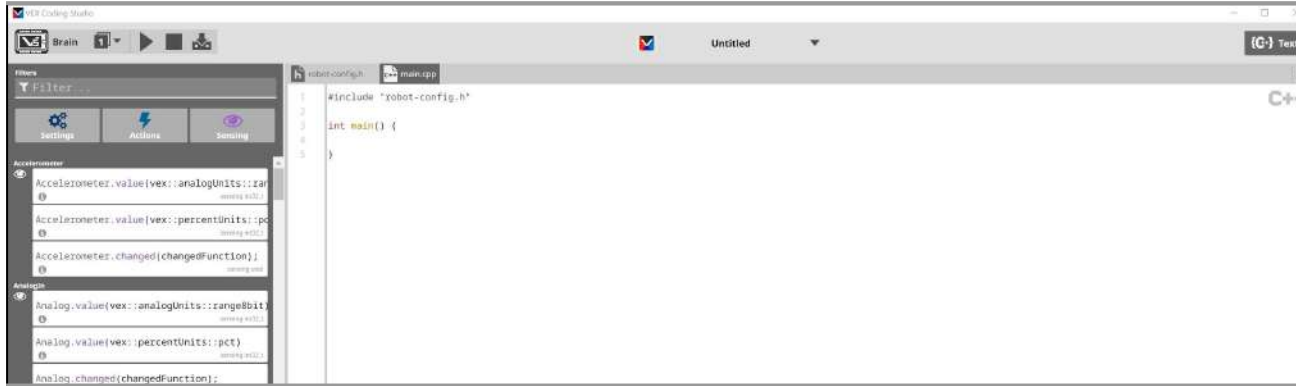
    else
    {
      motor[port3]=127;
      motor[port2]=-127;
      wait1Msec(1500);
    }
  }
}
```

Program running on the robot

Programming the Robot

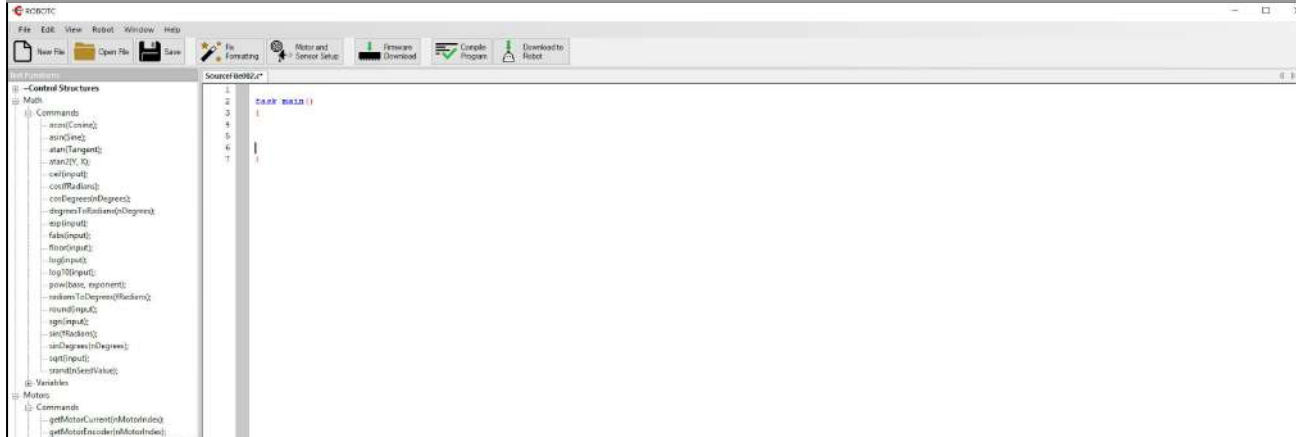
Sensorial

Two options exist for giving a robot behavior depending on which microcontroller used.



C++/VEX Coding Studio

VEX Coding Studio is an unlimited programming environment with all the capabilities of the VEX V5 Brain. Users have a full Industry Standard C++ environment available.



C/RobotC



RobotC is a C Programming Language for robotics. RobotC is also the name of the code editor that's used to write procedural code that is executed by the VEX Cortex microcontroller. The RobotC Natural Language API contains all the commands necessary to add behavior.

3. How to Program



RobotC and the Cortex Microcontroller

Variables

Variables are places to store values (such as sensor readings) for later use, or for use in calculations. There are three main steps involved in using a variable:

```
task main()
{
  int speed;
  speed = 75;
  motor[port3] = speed;
  motor[port2] = speed;
  wait1Msec(2000);
}
```

Declaration

The variable is created by specifying its *type*, followed by its *name*. Here, it is a variable named `speed` that will store an integer.

Assignment

The variable is assigned a *value* using a '='. The variable `speed` now contains the integer value 75.

Use

The variable can now "stand in" for any value of the appropriate type, and will act as if its stored value were in its place.

Rules for Variable Types

- You must choose a data type that is appropriate for the value you want to store

Data Type	Description	Example Values	Code
Integer	Positive and negative whole numbers, as well as zero.	-35, -1, 0, 33, 100, 345	<code>int</code>
Floating Point Decimal	Numeric values with decimal points (even if the decimal part is zero).	-.123, 0.56, 3.0, 1000.07	<code>float</code>
Boolean	True or False. Useful for expressing the outcomes of comparisons.	true, false	<code>bool</code>
Character	Individual characters, such as letters and numbers, placed in single quotes.	'n', '5', 'Z'	<code>char</code>
String	Multiple characters in a row, can optionally form sentences and words, placed in double quotes.	"Hello World!", "asdf", "Zebra Number 56"	<code>string</code>

Rules for Variable Names

- A variable name can not have spaces in it
- A variable name can not have symbols in it
- A variable name can not start with a number
- A variable name can not be the same as an existing reserved word

Proper Variable Names	Improper Variable Names
<code>linecounter</code>	<code>line counter</code>
<code>threshold</code>	<code>threshold!</code>
<code>distance3</code>	<code>3distance</code>
<code>timecounter</code>	<code>time1[T1]</code>

Boolean Logic

Comparison Operators

+

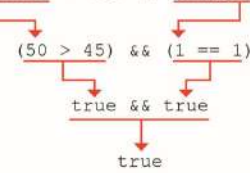
Logical Operators

=

More narrow, complicated conditions

ROBOTC Symbol	Meaning	Sample comparison	Result
==	"is equal to"	50 == 50	true
		50 == 100	false
		100 == 50	false
!=	"is not equal to"	50 != 50	false
		50 != 100	true
		100 != 50	true
<	"is less than"	50 < 50	false
		50 < 100	true
		100 < 50	false
<=	"is less than or equal to"	50 <= 50	true
		50 <= 100	true
		50 <= 0	false
>	"is greater than"	50 > 50	false
		50 > 100	false
		100 > 50	true
>=	Greater than or equal to	50 >= 50	true
		50 >= 100	false
		100 >= 50	true

The Boolean statement `(sonarSensor > 45) && (bumper == 1)` would be evaluated...



ROBOTC Symbol	Meaning	Sample comparison	Result
&&	"AND"	true && true	true
		true && false	false
		false && true	false
		false && false	false
	"OR"	true true	true
		true false	true
		false true	true
		false false	false

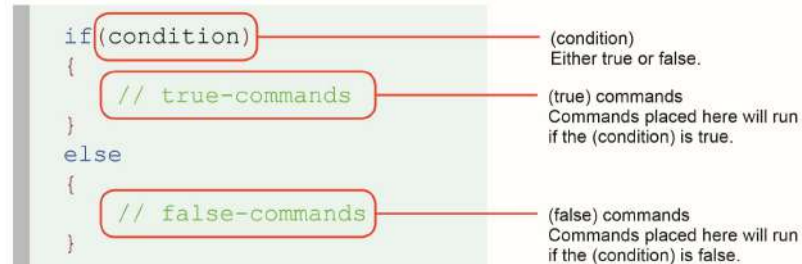
If Statements

An if-Statement allows your robot to make a decision. When your robot reaches an if Statement in the program, it evaluates the condition contained between the parenthesis. If the condition is true, any commands between the braces are run. If the condition is false, those same commands are ignored

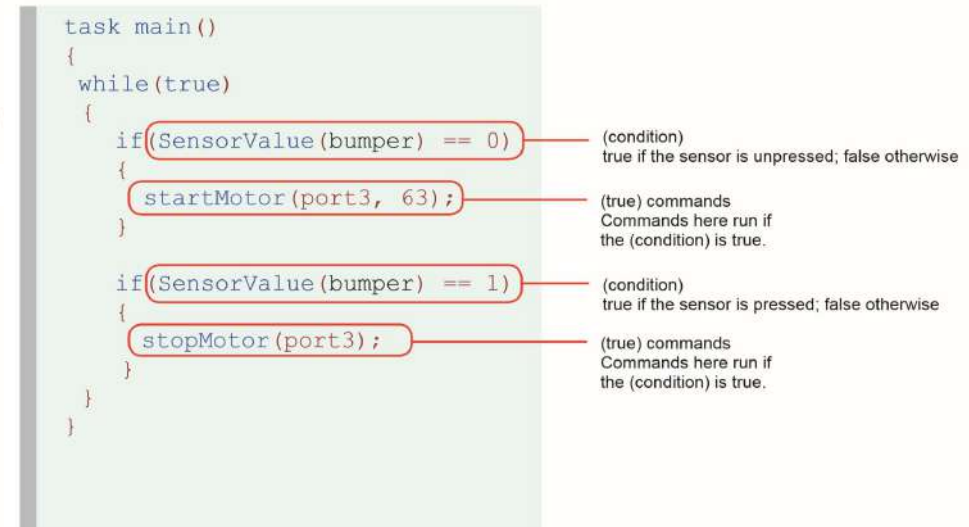
Pseudocode of an if Statment:



Pseudocode of an if-else Statment:



Example program containing two if Statements:



This program uses a Bumper Switch and two if Statements to control when the port3 motor moves. The first if Statement sets the motor to half power forward if the Bumper Switch has not been pressed, while the second turns the motor off if it has been pressed. Continually repeating these two behaviors within the while loop causes the motor to spin forward while the Bumper Switch is released, and to remain stopped for as long as it is pressed.

While Loops

A while loop is a structure which allows a section of code to be repeated as long as a certain condition remains true.

There are three main parts to every while loop.

Part 1. The keyword “while”.

```
while(condition)
{
    // repeated-commands
}
```

while
Every while loop begins with the keyword “while”.

Part 2. The condition.

```
while(condition)
{
    // repeated-commands
}
```

(condition)
The condition controls how long or how many times a while loop repeats. While the condition is true, the while loop repeats; when the condition is false, the while loop ends and the robot moves on in the program. The condition is checked every time the loop repeats, before the commands between the curly braces are run.

Part 3. The commands to be repeated, or “looped”.

```
while(condition)
{
    // repeated-commands
}
```

Repeated commands
Commands placed between the curly braces will repeat while the (condition) is true when the program checks at the beginning of each pass through the loop.

Functions

Function

Declare Function

```
void rotateArm()  
{  
  startMotor(armMotor, 63);  
  wait(3.25);  
  stopMotor(armMotor);  
}
```

Call Function

```
task main()  
{  
  rotateArm();  
}
```

A function is a group of statements that are run as a single unit when the function is called from another location.

Parameters are a way of passing information into a function. That value will typically influence how the function runs. It may help to think of the parameters as placeholders – all parameters must be filled in with real values when the function.

Not all functions are declared “void”. Sometimes a task might need information back out of the function at the end. The function will “return” a value, causing it to behave as if the function call itself were a value in the line that called it.

Parameterized Function

Declare Parameter

A parameter is declared similar to a variable (type & name)

Use Parameter

The parameter value behaves like a “placeholder”

Call function with parameter

```
void rotateArm(float time)  
{  
  startMotor(armMotor, 63);  
  wait(time);  
  stopMotor(armMotor);  
}
```

```
task main()  
{  
  rotateArm(3.25);  
}
```

```
void rotateArm(float time)  
{  
  startMotor(armMotor, 63);  
  wait(time);  
  stopMotor(armMotor);  
}
```

```
task main()  
{  
  rotateArm(3.25);  
}
```

```
startMotor(armMotor, 63);  
wait(3.25);  
stopMotor(armMotor);
```

(Parameterized) Return Function

Declare Return Type

Indicate what type of value it will return

Return Value

Note the value that will be returned.

Call function with parameter

```
int squareOf(int t)  
{  
  int sq;  
  sq = t * t;  
  return sq;  
}
```

```
task main()  
{  
  startMotor(rightMotor, 63);  
  wait(squareOf(100));  
  stopMotor(rightMotor);  
}
```

```
int squareOf(int t)  
{  
  int sq;  
  sq = t * t;  
  return sq;  
}
```

```
task main()  
{  
  startMotor(rightMotor, 63);  
  wait(squareOf(100));  
  stopMotor(rightMotor);  
}
```

```
wait(10000);
```

SUBSTITUTIONS

Switch Case

```
task main()
{
  bMotorReflected[port2]=1;
  int turnVar=0;

  while(true)
  {
    if(SensorValue(touch1)==1)
      turnVar=1;

    if(SensorValue(touch2)==1)
      turnVar=2;

    switch (turnVar)
    {
      case 1:
        motor[port3]=-127;
        motor[port2]=127;
        turnVar=0;
        break;

      case 2:
        motor[port3]=127;
        motor[port2]=-127;
        turnVar=0;
        break;

      default:
        motor[port3]=127;
        motor[port2]=127;
    }
  }
}
```

The switch-case command is a decision-making statement which chooses commands to run from a list of separate “cases”. A single “switch” value is selected and evaluated, and different sets of code are run based on which “case” the value matches.

Switch statement

The “switch” line designates the value that will be evaluated to see if it matches any of the case.

Case statement

The first line of a case includes the word “case” and a value. If the value of the “switch” variable (turnVar) matches this case value (1), the code following the “case” line will run.

Commands

These commands belong to the case “1”, and will run if the value of the “switch” variable (turnVar) is equal to 1.

Break statement

Each “case” ends with the command break;

Default case statement

If the “switch” value above did not match any of the cases presented by the time it reaches this point, the “default” case will run.

Timers

Timers are very useful for performing a more complex behavior for a certain period of time.

```
task main()
{
  bMotorReflected[port2]=1;
  ClearTimer(T1);
  while(time1[T1] < 3000)
  {
    if(SensorValue(lineFollower) < 45)
    {
      motor[port3]=63;
      motor[port2]=0;
    }
    else
    {
      motor[port3] = 0;
      motor[port2] = 63;
    }
  }
}
```

Clear the Timer

Clearing the timer resets and starts the timer. You can choose to reset any of the timers, from T1 to T4.

Timer in the (condition)

This loop will run “while the timer’s value is less than 3 seconds”, i.e. less than 3 seconds have passed since the reset. The line tracking behavior inside the {body} will continue for 3 seconds.

First, you must reset and start a timer by using the ClearTimer() command. Here’s how the command is set up:

```
ClearTimer(Timer_number);
```

The VEX has 4 built in timers: T1, T2, T3, and T4.

So if you wanted to reset and start Timer T1, you would type:

```
ClearTimer(T1);
```

Then, you can retrieve the value of the timer by using `time1[T1]`, `time10[T1]`, or `time100[T1]` depending on whether you want the output to be in 1, 10, or 100 millisecond values.

In the example above, you should see in the condition that we used `time1[T1]`. The robot will track a line until the value of the timer is less than 3 seconds. The program ends after 3 seconds.

Reserved Words

Reserved words (also known as “keywords”) are provided directly by the RobotC Programming Language. Because they are a feature of the language itself, they will always be accessible, even without the Natural Language API.

MOTORS

```
motor[output] = power;
```

This turns the referenced VEX motor output either on or off and simultaneously sets its power level. The VEX has 8 motor outputs: `port1`, `port2`... up to `port8`. The VEX supports power levels from -127 (full reverse) to 127 (full forward). A power level of 0 will cause the motors to

```
motor[port3]= 127;    //port3 - Full speed forward
motor[port2]= -127;   //port2 - Full speed reverse
```

```
bMotorReflected[output] = 1; (or 0;)
```

When set equal to one, this code reverses the rotation of the referenced motor. Once set, the referenced motor will be reversed for the entire program (or until `bMotorReflected[]` is set equal to zero).

This is useful when working with motors that are mounted in opposite directions, allowing the programmer to use the same power level for each motor.

There are two settings: 0 is normal, and 1 is reverse. You can use “true” for 1 and “false” for 0.

Before:

```
motor[port3]= 127;    //port3 - Full speed forward
motor[port2]= 127;    //port2 - Full speed reverse
```

After:

```
bMotorReflected[port2]= 1; //Flip port2's direction
motor[port3]= 127;        //port3 - Full speed forward
motor[port2]= 127;        //motorA - Full speed forward
```

Reserved Words

TIMING

```
wait1Msec(wait_time);
```

This code will cause the robot to wait a specified number of milliseconds before executing the next instruction in a program. "wait_time" is an integer value (where 1 = 1/1000th of a second). Maximum wait_time is 32768, or 32.768 seconds.

```
motor[port3]= 127;    //port3 - full speed forward
wait1Msec(2000);     //Wait 2 seconds
motor[port3]= 0;     //port3 - off
```

```
wait10Msec(wait_time);
```

This code will cause the robot to wait a specified number of hundredths of seconds before executing the next instruction in a program. "wait_time" is an integer value (where 1 = 1/100th of a second). Maximum wait_time is 32768, or 327.68 seconds.

```
motor[port3]= 127;    //port3 - full speed forward
wait10Msec(200);     //Wait 2 seconds
motor[port3]= 0;     //port3 - off
```

```
time1[timer]
```

This code returns the current value of the referenced timer as an integer. The resolution for "time1" is in milliseconds (1 = 1/1000th of a second).

The maximum amount of time that can be referenced is 32.768 seconds (~1/2 minute)

The VEX has 4 internal timers: T1, T2, T3, and T4

```
int x;                //Integer variable x
x=time1[T1];          //Assigns x=value of Timer 1 (1/1000 sec.)
```

Reserved Words

TIMING

`time10[timer]`

This code returns the current value of the referenced timer as an integer. The resolution for "time10" is in hundredths of a second (1 = 1/100th of a second).

The maximum amount of time that can be referenced is 327.68 seconds (~5.5 minutes)

The VEX has 4 internal timers: `T1`, `T2`, `T3`, and `T4`

```
int x;           //Integer variable x
x=time10[T1];    //Assigns x=value of Timer 1 (1/100 sec.)
```

`time100[timer]`

This code returns the current value of the referenced timer as an integer. The resolution for "time100" is in tenths of a second (1 = 1/10th of a second).

The maximum amount of time that can be referenced is 3276.8 seconds (~54 minutes)

The VEX has 4 internal timers: `T1`, `T2`, `T3`, and `T4`

```
int x;           //Integer variable x
x=time100[T1];   //assigns x=value of Timer 1 (1/10 sec.)
```

`ClearTimer(timer);`

This resets the referenced timer back to zero seconds.

The VEX has 4 internal timers: `T1`, `T2`, `T3`, and `T4`

```
ClearTimer(T1); //Clear Timer #1
```

Reserved Words

SOUND

```
PlayTone(frequency, duration);
```

This plays a sound from the VEX internal speaker at a specific frequency (1 = 1 hertz) for a specific length (1 = 1/100th of a second).

```
PlayTone(220, 500); //Plays a 220hz tone for 1/2 second
```

RADIO CONTROL

```
bVexAutonomousMode
```

Set the value to either 0 for radio enabled or 1 for radio disabled (autonomous mode). You can also use "true" for 1 and "false" for 0.

```
bVexAutonomousMode = 0; //enable radio control  
bVexAutonomousMode = 1; //disable radio control
```

```
vexRT[joystick_channel]
```

This command retrieves the value of the specified channel being transmitted.

```
bVexAutonomousMode = false; //enable radio control  
while(true)  
{  
    motor[port3] = vexRT[Ch3]; //right joystick, y-axis  
                                //controls the motor on port 3  
    motor[port2] = vexRT[Ch2]; //left joystick, y-axis  
                                //controls the motor on port 2  
}
```

If the RF receiver is plugged into Rx 1, the following values apply:

Control Port	Joystick Channel	Possible Values
Right Joystick, X-axis	Ch1	-127 to 127
Right Joystick, Y-axis	Ch2	-127 to 127
Left Joystick, Y-axis	Ch3	-127 to 127
Left Joystick, X-axis	Ch4	-127 to 127
Left Rear Buttons	Ch5	-127, 0, or 127
Right Rear Buttons	Ch6	-127, 0, or 127

If the RF receiver is plugged into Rx 2, the following values apply:

Control Port	Joystick Channel	Possible Values
Right Joystick, X-axis	Ch1Xmtr2	-127 to 127
Right Joystick, Y-axis	Ch2Xmtr2	-127 to 127
Left Joystick, Y-axis	Ch3Xmtr2	-127 to 127
Left Joystick, X-axis	Ch4Xmtr2	-127 to 127
Left Rear Buttons	Ch5Xmtr2	-127, 0, or 127
Right Rear Buttons	Ch6Xmtr2	-127, 0, or 127

Reserved Words

MISCELLANEOUS

`SensorValue(sensor_input)`

SensorValue is used to reference the integer value of the specified sensor port. Values will correspond to the type of sensor set for that port.

The VEX has 16 analog/digital inputs: `in1`, `in2...` to `in16`

```
if(SensorValue(in1) == 1) //If in1 (bumper) is pressed
{
    motor[port3] = 127;    //Motor Port 3 full speed forward
}
```

`srand(seed);`

Defines the integer value of the "seed" used in the `random()` command to generate a random number. This command is optional when using the `random()` command, and will cause the same sequence of numbers to be generated each time that the program is run.

```
srand(16);    //Assign 16 as the value of the seed
```

`random(value);`

Generates random number between 0 and the number specified in its parenthesis.

```
random(100);    //Generates a number between 0 and 100
```

Type of Sensor	Digital/Analog?	Range of Values
Touch	Digital	0 or 1
Reflection (Ambient)	Analog	0 to 1023
Rotation (Older Encoder)	Digital	0 to 32676
Potentiometer	Analog	0 to 1023
Line Follower (Infrared)	Analog	0 to 1023
Sonar	Digital	-2, -1, and 1 to 253
Quadrature Encoder	Digital	-32678 to 32768
Digital In	Digital	0 or 1

Reserved Words

CONTROL STRUCTURE

```
task main() {}
```

Creates a task called "main" needed in every program. Task main is responsible for holding the code to be executed within a program.

```
while(condition) {}
```

Used to repeat a {section of code} while a certain (condition) remains true. An infinite while loop can be created by ensuring that the condition is always true, e.g. "1==1" or "true".

```
while(timer1[T1]<5000)//While the timer is less than 5 sec...
{
    motor[port3]= 127;//...motor port3 runs at 100%
}
```

```
if(condition) {}/else {}
```

With this command, the program will check the (condition) within the if statement's parentheses and then execute one of two sets of code. If the (condition) is true, the code inside the if statement's curly braces will be run. If the (condition) is false, the code inside the else statement's curly braces will be run instead. The else condition is not required when using an if statement.

```
if(sensorValue(bumper) ==1)//the bumper is used as...
{
    //...the condition
    motor[port3]= 0; //if it's pressed port3 stops
}
else
{
    motor[port3]= 127; //if it's not pressed port3 runs
}
```

Reserved Words

DATA TYPES

int

This data type is used to store integer values ranging from -32768 to 32768.

```
int x; //Declares the integer variable x
x = 765; //Stores 765 inside of x
```

The code above can also be written:

```
int x = 765; //Declares the integer variable x and...
            //...initializes it to a value of 765
```

long

This data type is used to store integer values ranging from -2147483648 to 2147483648.

```
long x; //Declares the long integer variable x
x = 76543210; //Stores 76543210 inside of x
```

float

This data type is used to store decimal or floating point numbers.

```
float x; //Declares the float variable x
x = 77.932; //Stores 77.932 inside of x
```

bool

This data type is used to store boolean values of either 1 (also true) or 0 (also false).

```
bool x; //Declares the bool variable x
x = 0; //Sets x to 0
```

Reserved Words

DATA TYPES

char

This data type is used to store a single character, specified between a set of single quotes.

```
char x;           //Declares the char variable x
x = 'J';         //Stores the character J inside of x
```

string

This data type is used to store a string of characters, such as a word or sentence, specified between a set of double quotes.

```
string x;        //Declares the long integer variable x
x = "ROBOTC rocks!"; //Stores ROBOTC rocks! inside of x
```


Using the Joystick Controller in ROBOTC



```
motor[motor name] = vexRT[channel number];
```





VEX Code Studio and the V5 Robot Brain