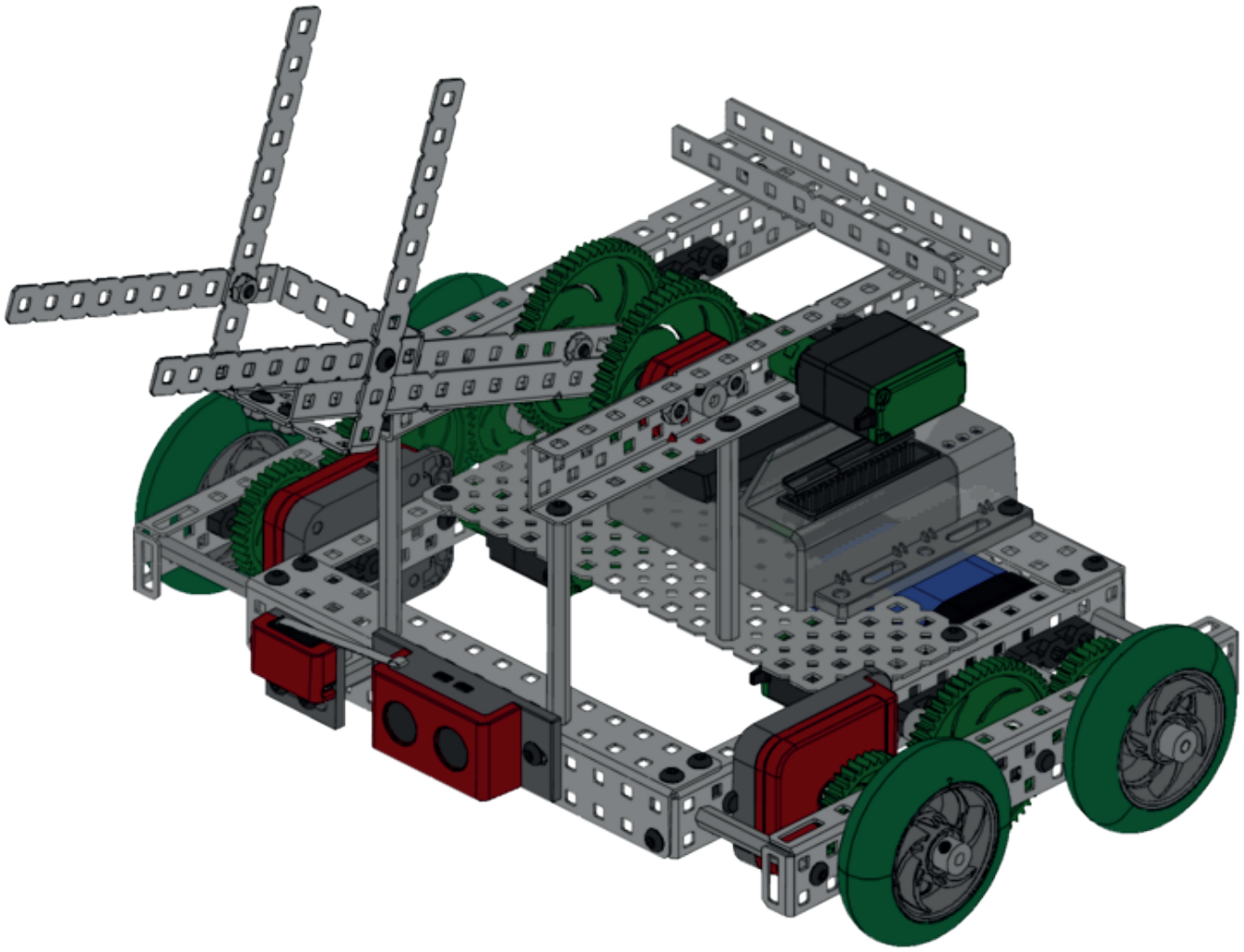
CENTRO ESCOLAR
LOS ALTOS

VEX Robotics



Manual de Programación

Robotics Student Training

CENTRO ESCOLAR
LOS ALTOS

CONTENIDO TEMÁTICO

Unidad 1

1. Definición de programación y robótica.
2. Programación básica de movimientos
3. Proceso de la Ingeniería
4. Introducción al lenguaje de programación
5. Selección de equipos
6. Construcción básica de un robot

Anotaciones

Unidad 2

1. Introducción al lenguaje de RobotC
    a. Comportamientos
    b. Pseudocódigo y flujo de datos
    c. Espacios y tabulaciones
    d. Comentarios
    e. Palabras reservadas
    f. Joystick
    g. Timers
    h. Switch Case
    i. While
    j. If … else
    k. Condicionantes booleanos
    l. Variables
2. Paper programming
3. Optimizando código
4. Ejecutar un programa

Anotaciones

Unidad 3

1. Wait States power level
2. Simulated Acceleration
3. Power Levels investigation
4. Turning
5. Sentry one
6. LCD display

Unidad 4

1. Introducción a los sensores
2. Descripción técnica
3. ShaftEncoder
4. Driving Straight 1 y 2
5. Encoder Investigation
6. Turning Investigation
7. Seeing difference
8. Incorporing functions
9. Programación general con sensores

Anotaciones

Unidad 5

1. Vexnet
2. Control Remoto
3. RoboSlalom
4. Repaso de programación

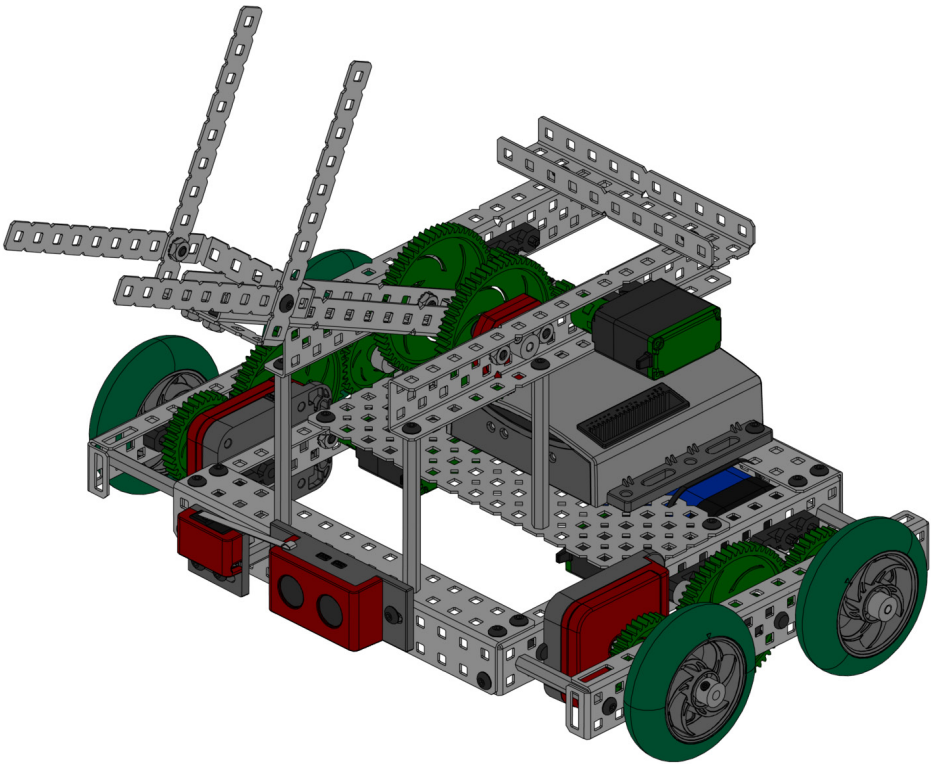
Anotaciones

Unidad 6

1. Diseño libre de construcción

Anotaciones

UNIDAD 1



Definición de programación y robótica

La **programación** es el proceso de diseñar, codificar, depurar y mantener el código fuente de programas computacionales. El código fuente es escrito en un lenguaje de programación. El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas, además del dominio del lenguaje a utilizar, algoritmos especializados y lógica formal.

La programación se rige por reglas y un conjunto más o menos reducido de órdenes, expresiones, instrucciones y comandos que tienden a asemejarse a una lengua natural acotada (en inglés); y que además tienen la particularidad de una reducida ambigüedad. Cuanto menos ambiguo es un lenguaje de programación, se dice, es más potente. Bajo esta premisa, y en el extremo, el lenguaje más potente existente es el binario.

La **robótica** es una ciencia o rama de la tecnología, que estudia el diseño y construcción de máquinas capaces de desempeñar tareas realizadas por el ser humano o que requieren del uso de inteligencia.
La robótica combina diversas disciplinas como la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control y la física. Otras áreas importantes en robótica son el álgebra, los autómatas programables, la animatrónica y las máquinas de estados.

El término robot se popularizó con el éxito de la obra R.U.R. (Robots Universales Rossum), escrita por Karel Čapek en 1920. En la traducción al inglés de dicha obra, la palabra checa robota, que significa trabajos forzados, fue traducida al inglés como robot.

¿Por qué crees que tiene relación la programación y la robótica?

Why we need to study?

The world needs the students of today to become the scientists, engineers, and problem solving leaders of tomorrow. The constant breakthroughs in chemistry, medicine, materials and physics reveal a new set of challenges and create an even greater opportunity for problem solving through technology. These problems are

not academic; the solutions could help save the world and those technology problem solvers will be the ones to make it possible.

Recognizing this dilemma, scores of organizations are creating programs designed to attract and engage young students in the study of science and technology. Many have found that robotics is a very powerful platform to attract and hold the attention of today's multi-tasking, connected youths. Robotics has strong appeal to this intensely competitive generation and represents the perfect storm of applied physics, mathematics, computer programming, digital prototyping and design, integrated problem solving, teamwork and thought leadership. Students with a previously undiscovered aptitude for STEM (Science, Technology, Engineering, and Math) curriculum are flourishing in growing numbers due to the efforts of schools, volunteer organizations, corporations, and governments internationally.

Working in teams will develop many new skills in response to the challenges and obstacles that stand before them. Some problems will be solved by individuals, while others will be handled through interaction with their student teammates and adult mentors. Teams will work together to build a VEX robot to compete in one of many tournaments, where they celebrate their accomplishments with other teams, family and friends. After the season, students come away not only with the accomplishment of building their own competition robot, but with an appreciation of science and technology and how they might use it to positively impact the world around them. In addition, they cultivate life skills such as planning, brainstorming, collaboration, teamwork, and leadership as well as research and technical skills. (Fuente: Vex Robotics Competiton SkyRise)

Escribe 5 OSOS

1. _____
2. _____
3. _____
4. _____
5. _____

Programación básica de movimientos

1. Observa el video **what is Computer Science** y contesta las siguientes preguntas:
2. Define Computer Programming utilizando los ejemplos que se dan en el video e investigando (escribe con tus palabras).

3. Escribe el nombre y empresa en la que trabajan las personas que aparecen en el video:

Nombre	**Empresa en la que trabaja**

4. Crea tu usuario utilizando la cuenta de correo institucional (utiliza de contraseña "altos" + los 4 dígitos de SV).
5. Contesta la siguiente actividad con la actividad de programación del primer bloque:

Escribe el código que aparece en el reto 1 e interpreta su funcionamiento:

Código	**Funcionamiento**

Escribe el código que aparece en el reto 6 e interpreta su funcionamiento:

Código	**Funcionamiento**

Programación básica de movimientos

Escribe el código que aparece en el reto 10 e interpreta su funcionamiento:

Código	**Funcionamiento**

Escribe el código que aparece en el reto 13 e interpreta su funcionamiento:

Código	**Funcionamiento**

Escribe el código que aparece en el reto 15 e interpreta su funcionamiento:

Código	**Funcionamiento**

Copia la pantalla de los bloques construidos para resolver el reto 17 e interpreta su funcionamiento:

Código	**Funcionamiento**

Copia la pantalla de los bloques construidos para resolver el reto 18 e interpreta su funcionamiento:

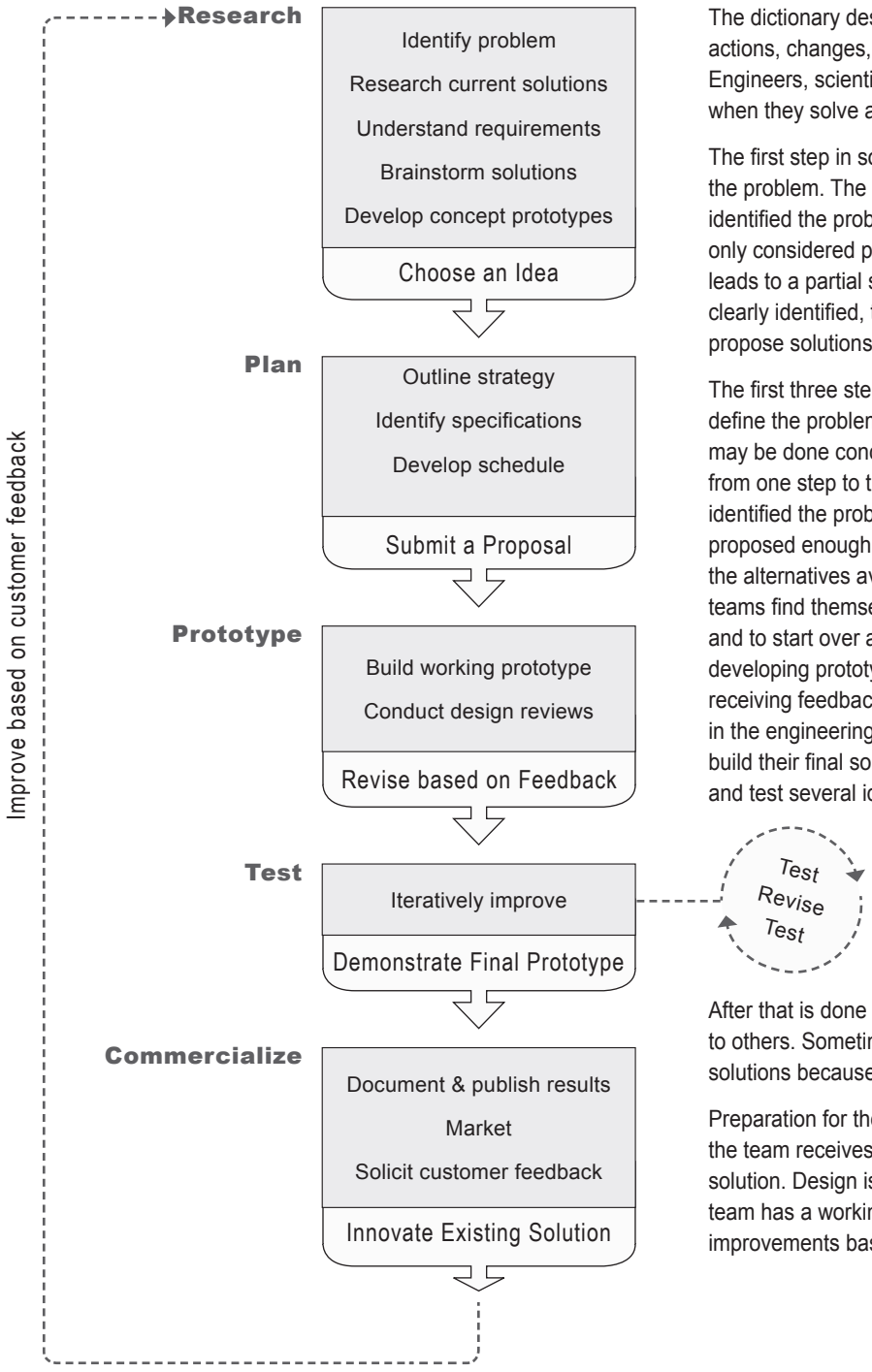
Código	**Funcionamiento**

6. Contesta las siguientes preguntas:

¿Cómo ejecuta la computadora los códigos? (observa como se ejecutan en code.org, se resaltan conforme se van ejecutando)

Los humanos cómo podemos darnos cuenta de lo que se encuentra en nuestro entorno y los robots ¿cómo podrían conocer su entorno?

Engineering Process



The dictionary describes a "process" as a series of actions, changes, or functions bringing about a result. Engineers, scientist, and researchers use a process when they solve a problem.

The first step in solving a problem is to clearly define the problem. The team may think that they have identified the problem, when in fact they may have only considered part of the problem, which typically leads to a partial solution. Once the problem has been clearly identified, the team can begin to brainstorm and propose solutions.

The first three steps of the problem-solving model: define the problem, brainstorm, and propose solutions, may be done concurrently. If the team moves too quickly from one step to the next, they may not have thoroughly identified the problem, brainstormed sufficiently, or proposed enough potential solutions to consider all of the alternatives available. When that happens, most teams find themselves returning to the beginning and to start over again. The next couple of steps: developing prototypes, testing, design reviews, and receiving feedback from others, are all important steps in the engineering process. Before a team begins to build their final solution, it is important that they model and test several ideas.

After that is done they will want to present their ideas to others. Sometimes, problem solvers miss potential solutions because they haven't considered all options.

Preparation for the design review and the feedback the team receives will move them closer to a working solution. Design is an iterative process. Even when the team has a working solution, they will want to consider improvements based on their testing.

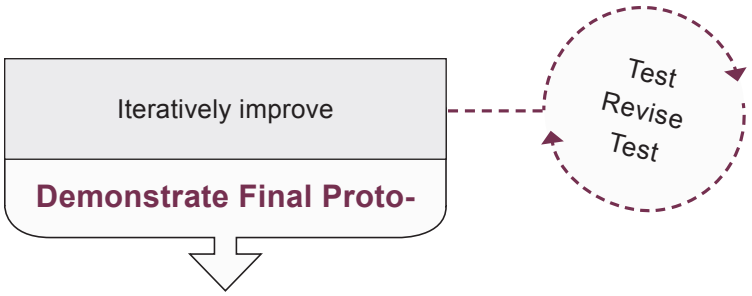
Definitions of the word 'Engineering'

- Derived from the Latin *ingenium*, engineering means something like brilliant idea, flash of genius. The word was created in the 16th century and originally described a profession that we would probably call an artistic inventor.
- Engineering is the principled application of science, methods, tools, and experience to the production of designed objects.
- Includes chemical engineering and materials techniques, civil engineering, electrical and electronics engineering, surveying, industrial engineering, metallurgical engineering, mining engineering, mechanical engineering, agricultural and forestry engineering techniques, fishery engineering techniques.
- Science, work, or profession of planning, building, or managing engines, machines, roads, bridges, canals, railroads, etc. (Old French *engin* = skill, which stems from Latin *ingenium* = ability to invent, brilliance, genius.)
- Analysis and/or design work requiring licensing through formal education, experience and the use of mathematics and the applied sciences.
- The art and science by which the mechanical properties of matter are made useful to man in structures and machines.
- The art of doing something well with one dollar what any idiot can do with two (or was it the other way around?).
- The use of scientific knowledge and trial-and-error to design systems.
- Engineering comprises any kind of activity which aims at either solving a problem or completing a task related to the definition, design and specification of a product

Iterative Development

Iterative development is the process of testing and continually making improvements to a product before it is finalized. In this process, multiple iterations of the product are developed, each iteration being closer to the final product than the last.

Iterative development happens in the Test portion of the Engineering Process. By now, the team has constructed a prototype, and is testing it to see how well it works. Undoubtedly, though, things will go wrong and the prototype will not perform as planned, so changes will need to be made. The group will make the necessary changes, and test again to see if the problem was solved. This cycle of continually building, testing and revising is the groundwork of the iterative development process.



Example: Line-tracking robot

1. Research how to make a line tracking robot
2. Plan out how to build robot and program
3. Build robot and program
4. Set up line and test robot
5. Testing shows that robot does not recognize line
6. Solution: reset threshold of Switch block and download program
7. Test again. Testing shows that robot moves too fast
8. Solution: lower motor power and download program
9. Test again: robot travels in circles at end of line. Robot is tracking too far!
10. Solution: reset the Loop, so that it goes for an amount of time
11. Test again. Robot tracks line well
12. Demonstrate and document results

As you can see through this example, the robot did not track the line properly the first time. Nor did it work well the second time. The robot designers found, through testing, many problems that they did not foresee. This is why iterative testing and revision are so important during the design process.

Engineering Design Notebook

Keeping Track of Your Project

An Engineering Design Notebooks is a place to keep track of all of the important information for the project. It can be a folder, binder or notebook, as long as it is wholly devoted to this project alone. In it, you will keep all of the notes, handouts, sketches and assignments that are related to the project.

You are responsible for your own Engineering Design Notebook, and for keeping it categorized, updated, and safe. Your instructor will collect the entire notebook from you at any time, so make sure it is up-to-date and organized.

All material should be kept in chronological order.

Your personal Engineering Design Notebook will include:

- Class handouts
- Daily logs and notes
- All sketches, plans, and drawings
- Notes from design reviews
- Calculations relevant to your project
- Documentation of the evolutionary changes of your project
- All completed and returned assignments
- Final (turned-in) version of any individual assignments that are due

Your group Design Notebook will include:

- Research information, such as computer print-outs and newspaper articles
- Meeting minutes or logs, including explanatory sketches and concept drawings
- Scheduling tools like PERT or Gantt Charts
- Notes for presentations, reports, proposals, etc.
- Final (turned-in) version of any group assignments that are due
- Graded and returned group assignments

Note: Your teacher may instruct you to keep one copy of the group Engineering Design Notebook per group, or to simply make a copy of all group deliverables and have each group member keep one. If your group only needs to keep one notebook for the whole group, choose a group member to be responsible for it so it does not get lost!

Assessment

- The notebook itself will be graded based on completeness and organization
- Students are responsible for lost, damaged, or poorly kept Journals. There will be no credit given for lost

Engineering Design Notebook

- When requested at any time, students must hand in their notebooks, to be returned after the contents have been graded
- Notes and logs are the only evidence of work done on a daily basis. Make sure they are complete and will explain your individual contributions to the project

Actividad del Proceso de Ingeniería

1. Escribe la definición más completa de **Ingeniería**, utilizando las definiciones dadas anteriormente.

2. Realiza un esquema lo más simplificado y completo del ciclo del **proceso de Ingeniería.**

3. Escribe 5 razones por las cuales es importante tener una bitácora.

1. _____
2. _____
3. _____

4. _____

5. _____

Considera en la elaboración de una bitácora la siguiente información:

- Datos generales: número del equipo, integrantes, fecha de la bitácora, tema, asignatura (Robótica)
- Descripción de la actividad realizada (en pasos generales qué hicieron y cómo trabajaron)
- Observaciones (hallazgos, descubrimientos, nueva experiencia).
- Aprendizajes (lo aprendido en qué podría servirte para tu vida).
- Fotografías y dibujos (mejorar la calidad) incluyendo descripción de las imágenes y dibujos.
- Orden, limpieza.
- Verificar la redacción (se entienden las ideas que trasmiten y hay secuencia en la información presentada).
- Verifica la ortografía.

Al realizar una bitácora puedes considerar:

- Incluir dibujos o bocetos escaneados.
- Iniciar la primer letra de con mayúsculas.
- Iniciar con la descripción de la actividad ¿cuál es la meta del día? Qué hicieron durante la clase? Cómo lo lograron?...
- Las observaciones pueden narrarse así: “nos dimos cuenta de …”, “encontramos…”, “descubrimos…”, etc)
- En aprendizajes puedes incluir la información del tema visto.
- Creatividad: utiliza tu creatividad para hacer la bitácora, puede cambiar de formato no necesariamente debe ser un texto plano acompañado de imágenes.

Introducción al Lenguaje de programación

1. Define qué es un robot y para qué sirve.

2. ¿Qué son los algoritmos y para qué se utilizan?

3. Escribe la diferencia entre lenguaje de programación y programa

El lenguaje máquina es conocido como el más simple, es aquel que procesa una computadora y solo interpreta los valores 1 y 0 que en combinaciones pueden representar números, letras, operaciones, etc.

4. Escribe los pasos y/o proceso para convertir un número decimal en binario y viceversa.

5. Traduce los siguientes números binarios a decimales y viceversa.
Utiliza la siguiente tabla (recuerda que los números se acomodan de izquierda a derecha):

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^o
1024	512	256	128	64	32	16	8	4	2	1

a) 111011 _____
b) 10001001 _____
c) 11010110 _____
d) 1011010 _____
e) 1101 _____
f) 1100011010 _____
g) 1011101 _____
h) 63 _____
i) 543 _____
j) 1602 _____
k) 897 _____
l) 642 _____
m) 998 _____

Brainstorming Primer

It is extremely difficult to get a diverse group of people to agree on many things. It becomes impossible if all members of the team want to have things their way. Below are general rules that apply to all team oriented problem solving situations. If the team would like to maintain good group dynamics, it is important that they spend a several minutes talking about these important rules.

Listening Tips For Your Team

- Listen; easy to say, hard to do. Listening is hard work– *work at it!*
- Never assume anything.
- Don't jump to conclusions. Listen to all opinions before you form your own.
- Focus on the problem; it is easy to get sidetracked.
- Be positive.
- If you aren't getting it, listen harder to your teammates.
- Ask questions.

Brainstorming Ideas

Students need to understand that there are no bad ideas. All ideas are to be treated with respect. Often, one idea will lead to another idea. What may seem "off the wall" to the group may stimulate an idea from someone else. Many people are shy and won't share their opinions, particularly if they feel they will be ridiculed.

Be Flexible

There is more than one way to solve any problem. Team members must be prepared to compromise, combine good ideas, listen to others opinions, and treat all suggestions fairly and with respect.

Consider Resources

All problem solvers have a limited set of resources. When problem solving, your two biggest resources are people and information. If you know someone with expertise in the area on which you are working, consult them. Put in time upfront, researching how others have solved similar problems; there is no reason to "reinvent the wheel." Time is the one resource that is in your control at the beginning of the problem, but becomes totally out of your control as the project moves along. There is a saying that "time waits for no one." That becomes extremely important when you have a fixed deadline.

Test Multiple Solutions

Students often select the first idea that comes to mind without thinking about other options. It is important for the team to take the time to look at several options if they want to come up with the best solution.

Be Positive

Do you see yourself as "part of the solution" or "part of the problem?" We all would like to see ourselves as part of the solution. Unfortunately it doesn't always work out the way that we would like and this can lead to unhappy teammates. It is imperative that you understand how important it is to be positive when you are working in teams.

Things to Think About

Here are some useful starter questions to get your first brainstorming session going. Make sure to write down all ideas (even some that may seem unfeasible now may come in handy later) and listen to everyone who has something to say. There are no bad ideas in a brainstorming session. Make no judgments about the ideas offered.

- ***What are we trying to accomplish?***
- ***Has anyone done this sort of thing before? How?***
- ***How do you think we should…?***
- ***Is there another way to do this?***
- ***What should be the next…?***
- ***What skills and abilities do we have that would be useful?***
- ***How can we use them?***
- ***What can we improve?***
- ***How can we change the criteria to fit our assets?***
- ***How do you plan on…?***
- ***How can we change to attack the problem?***
- ***Can we use something else?***
- ***What if we did the opposite?***
- ***Is it time to leave it alone?***
- ***What are the technical hurdles we must overcome to complete the task?***
- ***How can we break this down into smaller tasks?***
- ***How is our approach different from other approaches? Do we stand out?***

Brainstorming Tips

Here are some useful tips that you may want to consider while you try to solve problems.

Accept all options
In the beginning of the brainstorming process, have the group write down as many ideas as possible without actually evaluating any of them. Often, ideas are dismissed early, before they have a chance to be discussed. Discussions may lead to other good ideas. Remember that one person's idea may stimulate another person to come up with another idea that may lead to a solution. Sometimes combining several ideas will lead to a solution.

Avoid distraction
Stay focused on the problem. Take yourself away from the television and phone.
Make sure to keep group members on task, as it is very easy to be distracted by friends.

Work in a new setting
For some people, working in a new setting is stimulating and may jump-start new ideas.

Ask experts
Your two biggest resources as a problem solver are people and information.
If you are trying to solve problems that are new to you, find people who are experts in the field & ask for their opinion.

Be positive
Ideas are more likely to evolve if everyone has a positive attitude about the outcome.
The words "I can't" never solved a problem.

Be confident
The team needs to feel confident that the problem can be solved.

Break problems into small parts
Sometimes solving small parts of the problem will lead to a solution to the larger problem.

Take a break
A short break from the problem can inspire new ideas. Starting fresh and mentally alert can lead to new ways to solve the problem. Also, changing environments and trying to view the problem in the context of the new environment can give inspiration.

Persist
Genius has been described as "1% inspiration and 99% perspiration."
Hard work and persistence are keys to finding solutions to difficult problems.

Formación de equipo

1. Escribe el nombre de las integrantes del equipo

3. Definan con sus palabras lo que significa "**trabajo cooperativo**"

Escribe 5 ideas importantes a partir de la lectura de **Brainstorming Primer**

1. _____
2. _____
3. _____
4. _____
5. _____

Escribe en resumen las recomendaciones para trabajar de forma efectiva en equipo: (puedes utilizar la lectura de **Brainstorming Tips**)

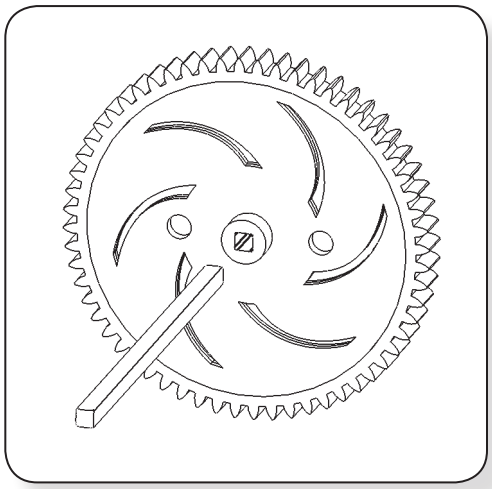
1. _____
2. _____
3. _____
4. _____
5. _____

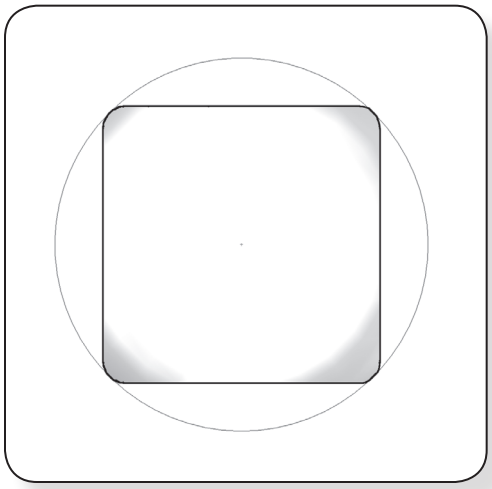
Introduction to the Motion Subsystem

The Motion Subsystem comprises all the components in the VEX Robotics Design System which make a robot move. These components are critical to every robot. The Motion Subsystem is tightly integrated with the components of the Structure Subsystem in almost all robot designs.

In the VEX Robotics Design System the motion components are all easily integrated together. This makes it simple to create very complex systems using the basic motion building blocks.

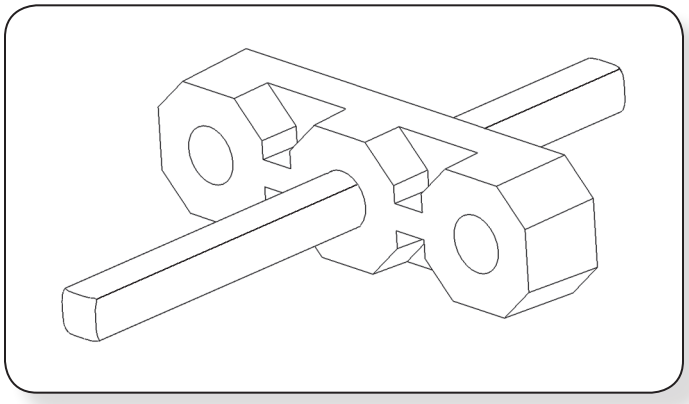
The most fundamental concept of the Motion Subsystem is the use of a square shaft. Most of the VEX motion components use a square hole in their hub which fits tightly on the square VEX shafts. This square hole – square shaft system transmits torque without using cumbersome collars or clamps to grip a round shaft.



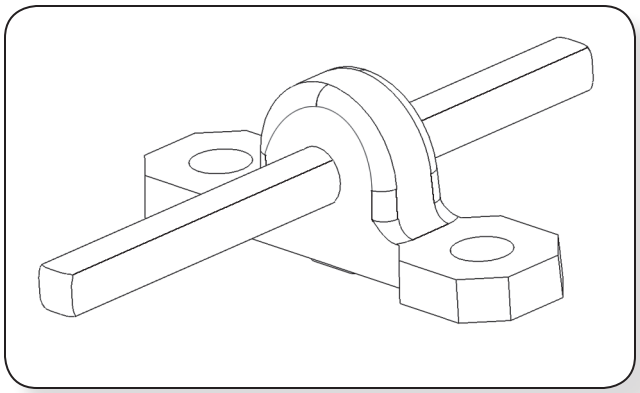


The square shaft has rounded corners which allow it to spin easily in a round hole. This allows the use of simple bearings made from Delrin (a slippery plastic). The Delrin bearing will provide a low-friction piece for the shafts to turn in.

These VEX Delrin bearings come in two types, the most common of which is a Bearing Flat. The Bearing Flat mounts directly on a piece of VEX structure and supports a shaft which runs perpendicular and directly through the structure.

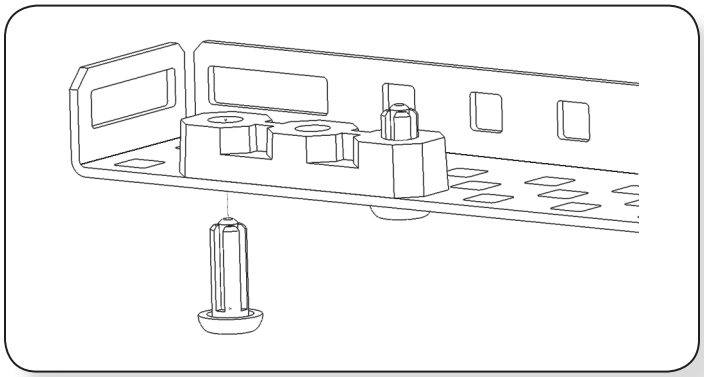


Introduction to the Motion Subsystem, continued

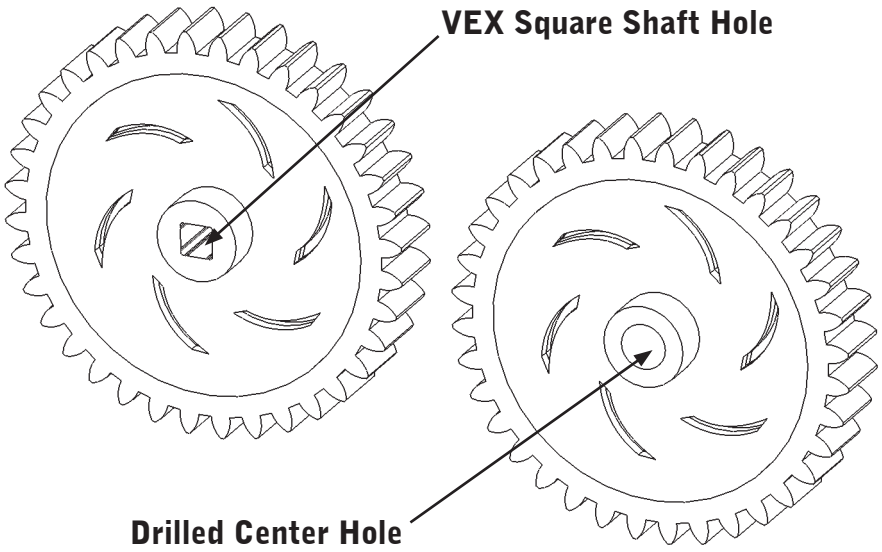


Another type of bearing used in the VEX Motion Subsystem is a Bearing Block; these are similar to the "pillow-blocks" used in industry. The Bearing Block mounts on a piece of structure and supports a shaft which is offset either above, below, or to the side of the structure.

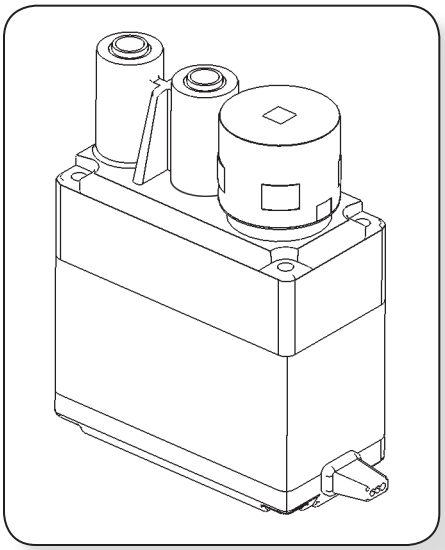
Some bearings can be mounted to VEX structural components with Bearing Pop Rivets. These rivets are pressed into place for quick mounting. These Rivets are removable; pull out the center piece by pulling up on the head of the Rivet to get it to release.



HINT:
It is also possible to to convert the square hole(s) in some Motion Subsystem Components to a round hole by using a drill (approximately 0.175" diameter) to create a round hole that replaces the part's original square hole. A VEX square shaft can then spin freely in the newly created round hole. This is useful for some specialty applications.



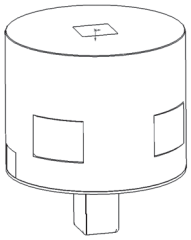
Introduction to the Motion Subsystem, continued

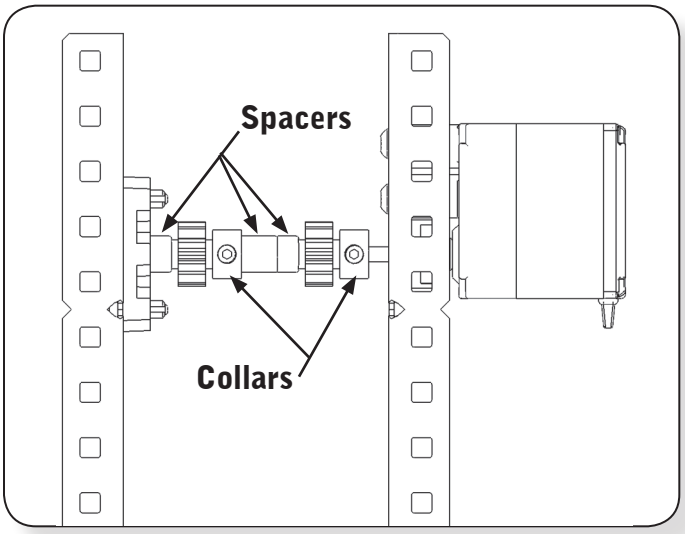


The key component of any motion system is an actuator (an actuator is something which causes a mechanical system to move). In the VEX Robotics Design System there are several different actuator options. The most common types of actuators used are the VEX Continuous Rotation Motors and the VEX Servos. (For more information on Motors & Servos refer to the "Concepts to Understand" section of this chapter.)

Each VEX Robotics Motor & Servo comes with a square socket in its face, designed to connect it to the VEX 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.

WARNING:
VEX Motors include a clutch assembly which is designed to prevent damage to the internals of the VEX Motor in the event of a shock-load. Motors can be used without clutches, but it is not recommended. For more information on VEX Clutches refer to the "Concepts to Understand" section of this chapter.





The Motion Subsystem also contains parts designed to keep pieces positioned on a VEX shaft. These pieces include washers, spacers, and shaft collars. VEX Shaft Collars slide onto a shaft, and can be fastened in place using a setscrew. Before tightening the setscrew, it is important to slide the Shaft 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.

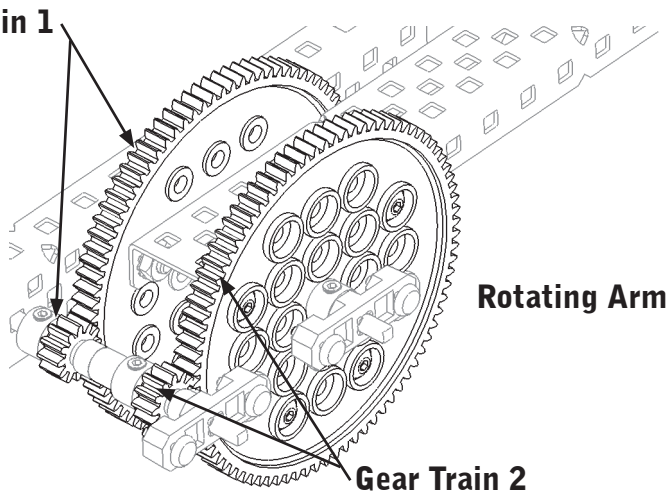
HINT: The setscrews used in VEX Shaft Collars are 8-32 size threaded screws; this is the same thread size used in the rest of the kit. There are many applications where it might be beneficial to remove the setscrew from the Shaft Collar and use a normal VEX screw.

If a setscrew is lost any other VEX 8-32 screw can be substituted although the additional height of the screw head must be considered!

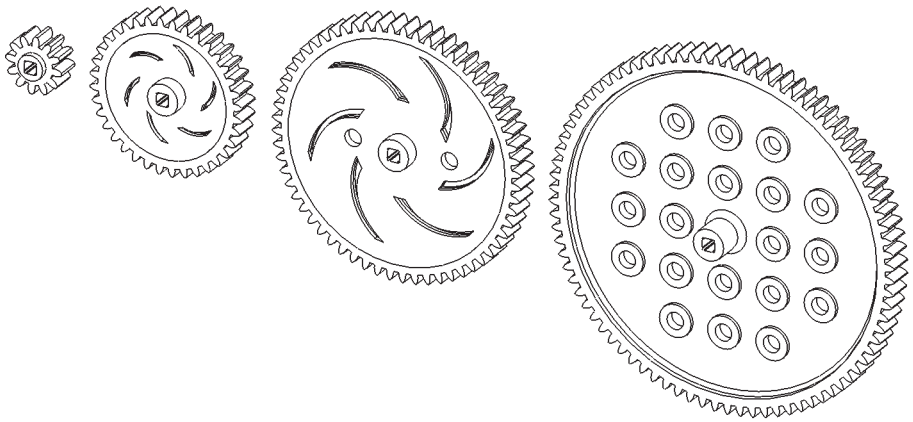
Introduction to the Motion Subsystem, continued

In some applications excessive loads can damage the components of the VEX Motion Subsystem. In these cases there are often ways to reinforce the system to reduce the load each individual component will experience, or so that the load is no longer concentrated at a single location on any given component.

EXAMPLE:
One example of a component failure is fracturing gear teeth. Another example is rounding out the square hole the shaft goes through. If either of these situations exists an easy way to fix it is to use multiple gears in parallel. Try using two gear trains next to each other to decrease the load on each individual gear.

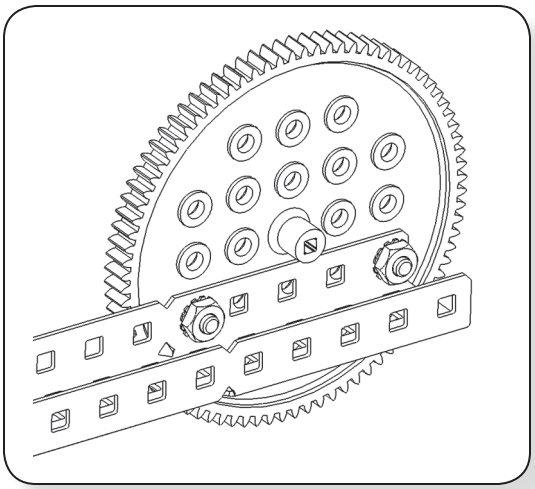


There are several ways to transfer motion in the VEX Robotics Design System. A number of Motion Subsystem accessory kits are available with a variety of advanced options. The primary way to transfer motion is through the use of spur gears. Spur gears transfer motion between parallel shafts, and can also be used to increase or decrease torque through the use of gear ratios.



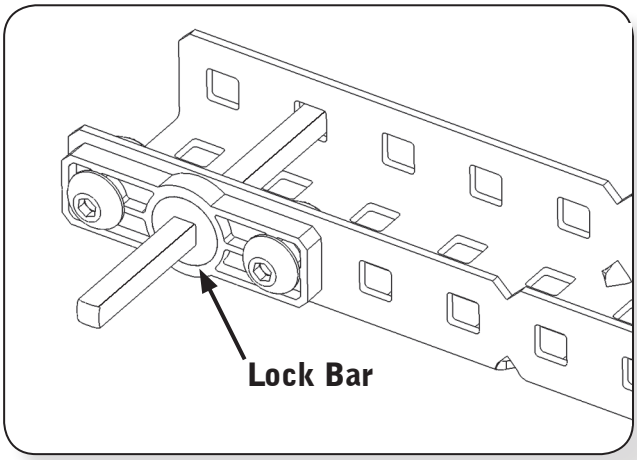
These gears can also be combined with sprocket & chain reductions, and also with advanced gear types to create even more complex mechanisms.

Introduction to the Motion Subsystem, continued



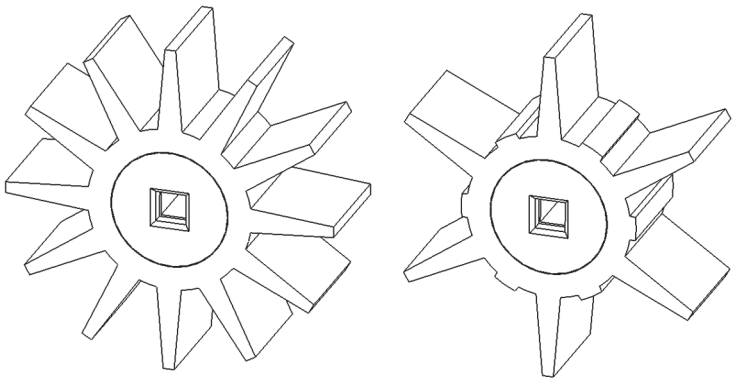
It is easy to drive components of the VEX Structure Subsystem using motion components in several different ways. Most of the VEX Gears have mounting holes in them on the standard VEX 1/2" hole spacing; it is simple to attach metal pieces to these mounting holes. One benefit of using this method is that in some configurations, the final gear train will transfer torque directly into the structural piece via a gear; this decreases the torque running through the shaft itself.

Another option to drive structural pieces using the Motion Subsystem is through a Lock Bar. These pieces are designed such that they can bolt onto any VEX structural component using the standard VEX 1/2" pitch. In the center of each piece there is a square hole which matches the VEX square shaft. As such, any VEX component can be "locked" to a shaft using the Lock Bar so that it will spin with the shaft. Note that the insert in each Lock Bar is removable and can be reinserted at any 15° increment.



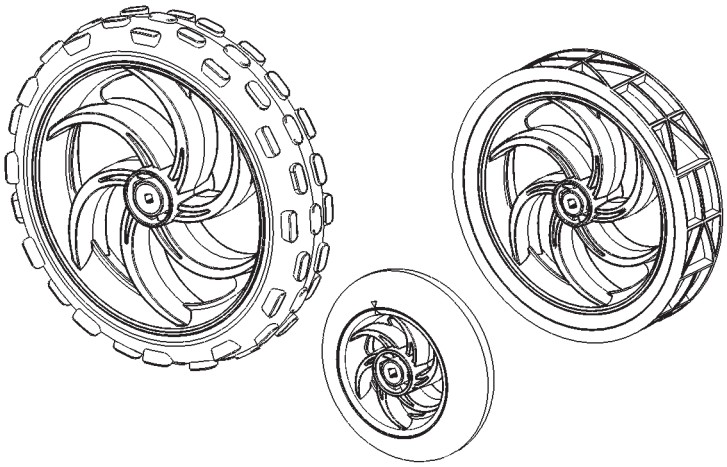
Intake Rollers can be used in a variety of applications. These components were originally designed to be rollers in an intake or accumulator mechanism. The "fins" or "fingers" of the roller will flex when they contact an object; this will provide a gripping force which should pull on the object.

HINT:
Try cutting off some of the fins of an Intake Roller for better performance on some objects.



Introduction to the Motion Subsystem, continued

The VEX Motion Subsystem contains a variety of components designed to help make robots mobile. This includes a variety of wheel sizes, tank treads, and other options. Robots using these in different configurations will have greatly varying performance characteristics.



Tank Tread components and wheels can also be used to construct intake mechanisms and conveyor belts. These are frequently used on competition robots.

When designing the Motion Subsystem of a robot it is important to think about several factors:

- First, it needs to be able to perform all the moving functions of the robot.
- Second, it needs to be robust enough to survive normal robot operation; it also needs to be robust enough to survive some abnormal shock loads.
- Third, it needs to be well integrated into the overall robot system.

The Motion Subsystem combines with the Structure Subsystem to form the primary physical parts of the robot. The motion components will be used throughout a robot's construction, and will likely be part of every major robot function. As such, this Subsystem needs to be well thought out in advance.

Concepts to Understand, continued

Speed vs. Torque

A motor can generate a set amount of power; that is, it can provide a specific amount of energy every second—this energy is most commonly used to make a wheel spin. Since there is only so much energy to go around, however, there is an inherent trade-off between ***Torque***—the force with which the motor can turn the wheel—and ***Speed***—the rate at which the motor can turn the 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.

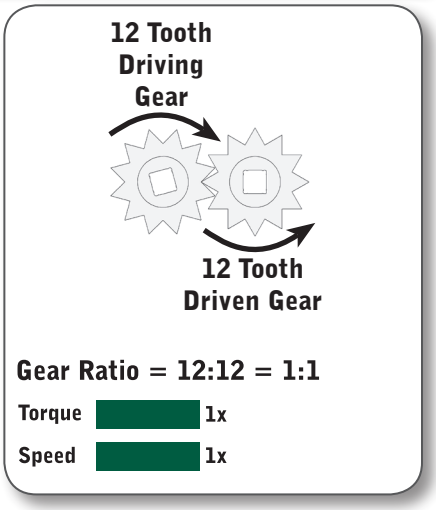
Gears

Gear Ratio

You can think of gear ratio as a "multiplier" on torque and a "divider" on speed. If you have a gear ratio of 2:1, you have twice as much torque as you would if you had a gear ratio of 1:1, but only half as much speed.

Calculating the gear ratio between a pair of gears is simple. First, identify which gear is the "driving" gear, and which is the "driven" gear. The "driving" gear is the one that is providing force to turn the other one. Often, this gear is attached directly to the motor axle. The other gear, the one that the driving gear is turning, is called the "driven" gear.

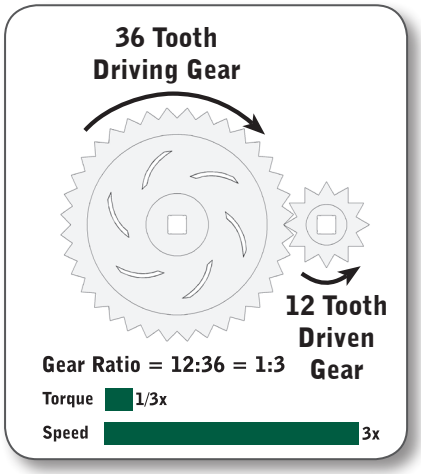
To find gear ratio, you just need to count the number of teeth on the "driven" gear, and divide it by the number of teeth on the "driving" gear.

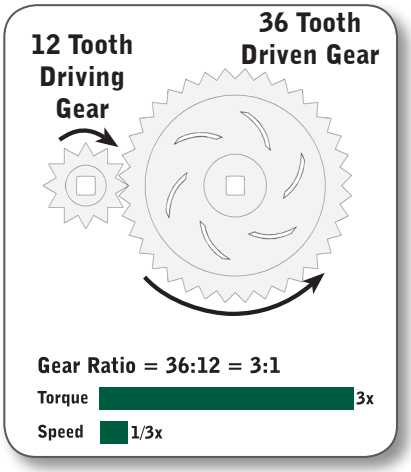


Mechanical Advantage –

The ratio of the force a machine can exert to the amount of force that is put in. Mechanical advantage can also be thought of as the "force multiplier" factor that a mechanical system provides.

If a vehicle has a gear train with a mechanical advantage of 2, for instance, it has twice as much force available to it, enabling it to go up hills that are twice as steep, or tow a load that is twice as heavy.





This additional force is never "free." It always comes at the expense of something else, such as speed. Also note that mechanical advantages are frequently fractional, indicating that force is being sacrificed for speed or some other similar performance factor in a system.

Concepts to Understand, continued

Gears, continued

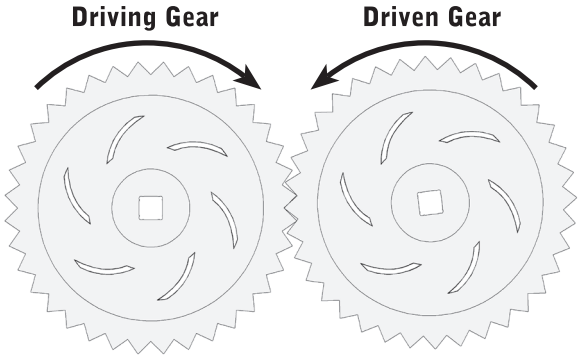
Idler Gears

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 (because they are a driven gear relative to the first gear, and a driving gear relative to the last gear—you would first multiply by the number of teeth on the idler gear and then divide by the same number, which always cancels out).

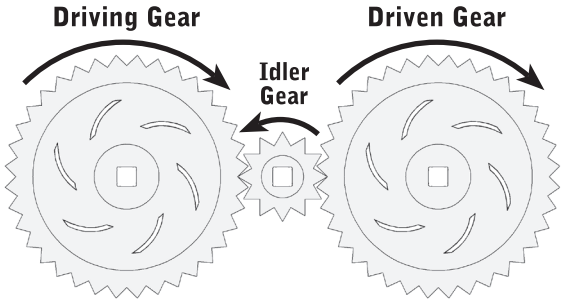
However, idler gears do reverse the direction of spin. Normally, the driving gear and the driven gear would turn in opposite directions. Adding an idler gear would make them turn in the same direction. Adding a second idler gear makes them turn in opposite directions again.

Idler gears are typically used either to reverse the direction of spin between two gears, or to transmit force from one gear to another gear far away (by using multiple idler gears to physically bridge the gap).

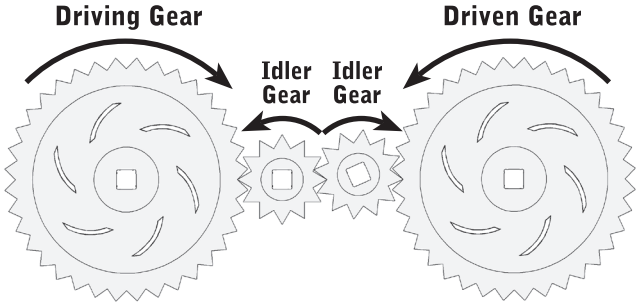
No Idler – Opposite Direction



One Idler – Same Direction



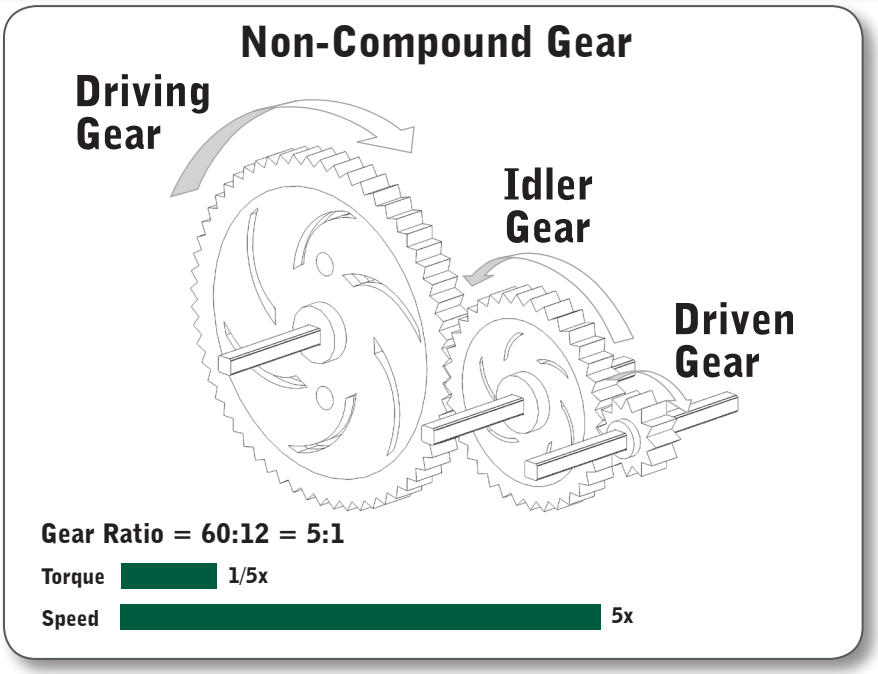
Two Idlers – Opposite Direction



Concepts to Understand, continued

Gears, continued

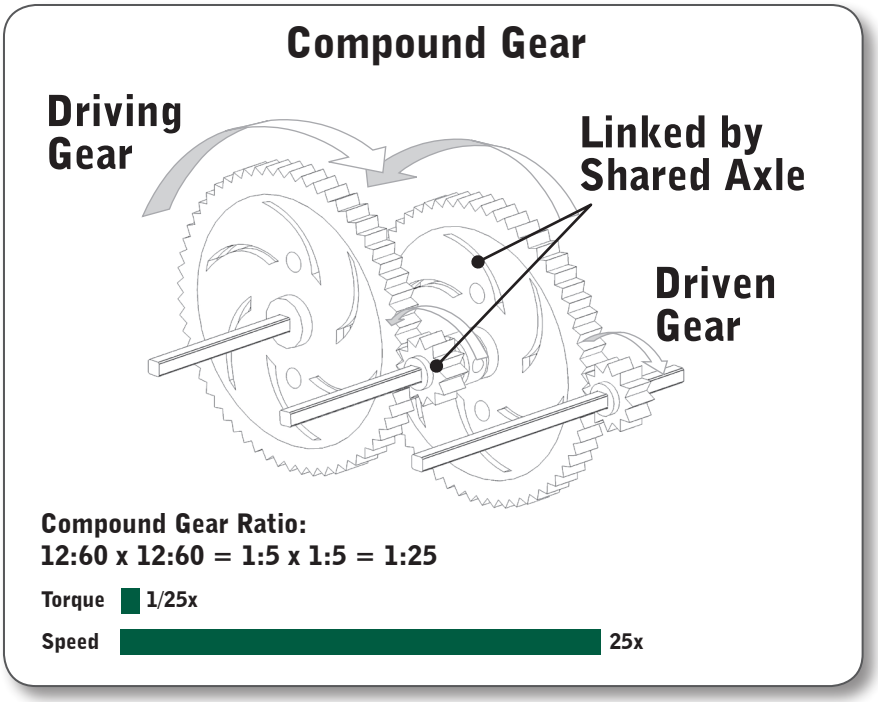
Compound Gear Ratio
Compound gears are formed when you have more than one gear on the same axle. Compound gears are not to be confused with idler gears, as compound gears can affect the overall gear ratio of a system!



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.



Compound gears allow configurations with gear ratios that would not normally be achievable with the components available. In the example above, a compound gear ratio of 1:25 was achieved using only 12 and 60-tooth gears. This would give your robot the ability to turn an axle 25 times faster than normal (though it would only turn with 1/25th of the force)!

Concepts to Understand, continued

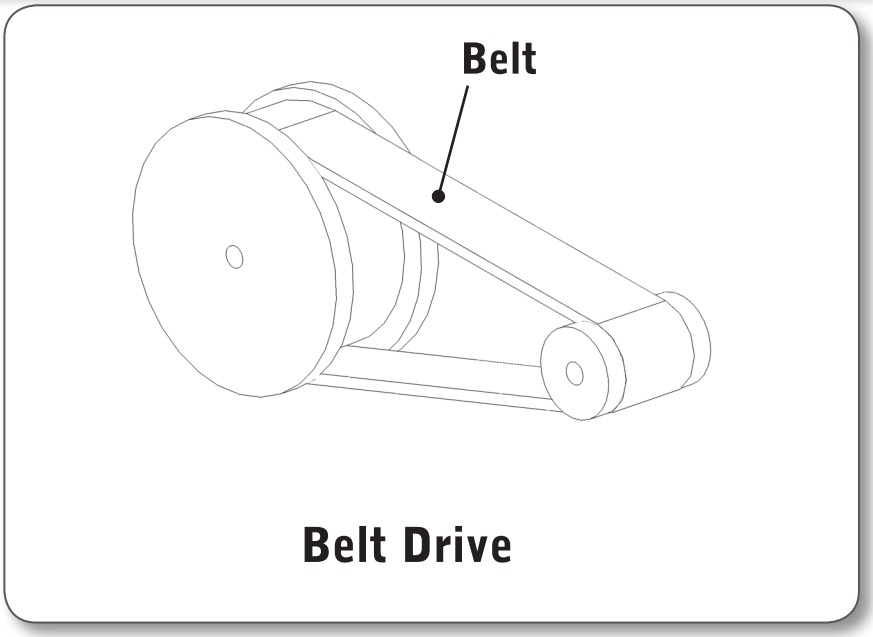
Gears, continued

Gear ratio with non-gear systems

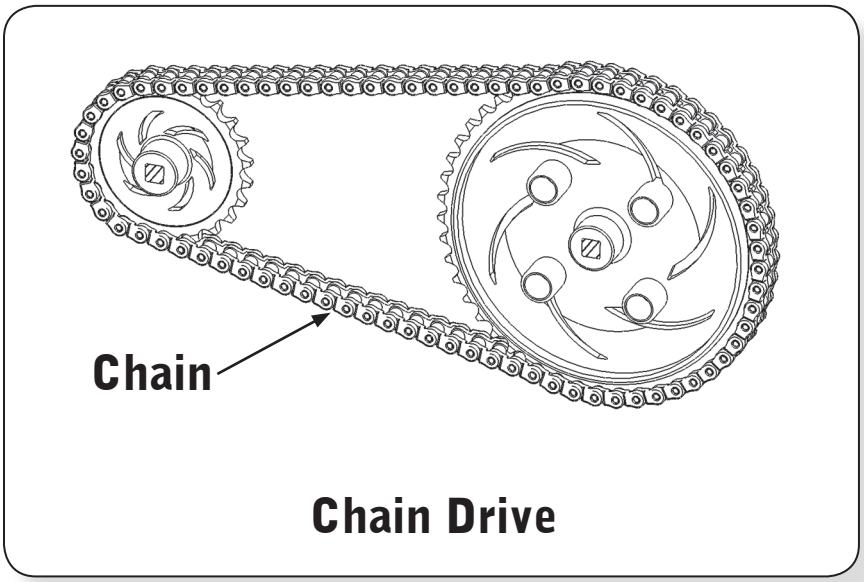
The real nature of gear ratios is a little more complex than just counting teeth on gears. Gear ratio is actually defined as the number of rotations that the driving axle needs to make in order to turn the driven axle around once. When dealing with toothed gears or sprockets, you can find the number of turns needed by counting teeth, as you have seen previously (see "Gear ratio").

With other types of systems, you can still find the "gear ratio" by measuring the number of rotations on the driven and driving axles. Some of these other drive types include belt-and-pulley drives and chain-and-sprocket drives.

Belt or chain drives are often preferred over gears when torque is needed to be transferred over long distances. Unlike spur gear reductions, Sprocket and Chain reductions do NOT reverse rotation.



Belt Drive



Chain Drive

Concepts to Understand, continued

Wheels

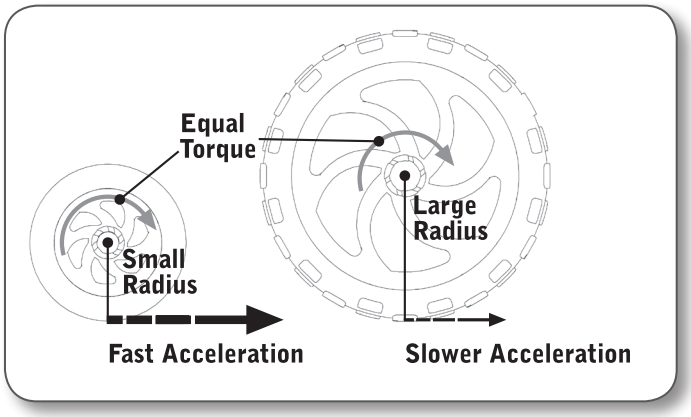
Wheel Sizes

Often, the role of the Motion Subsystem on a robot will be to move the robot along the ground. The last step in the drive train, after the motors and gears, is the wheels.

Like motors and gears, different properties of the wheel will affect your robot's performance. The size of the wheels will be an important factor here, and will affect two distinct and different characteristics of the robot: its acceleration, and its top speed.

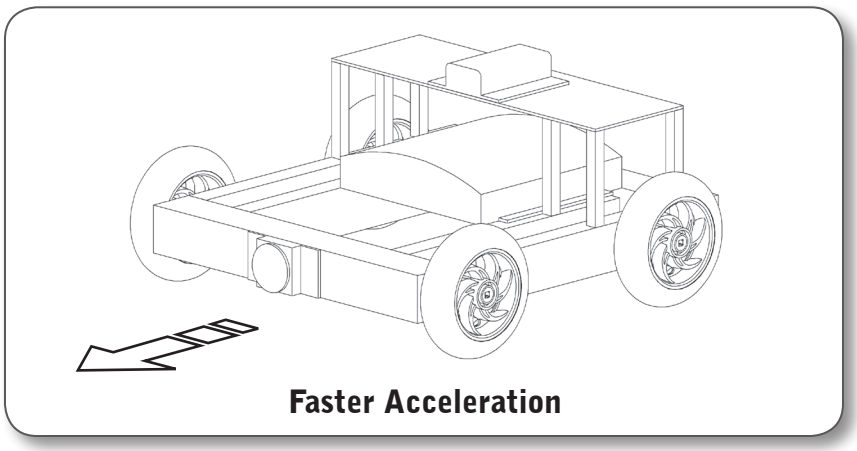
Wheel Sizes and Acceleration

The relationship between wheel size and acceleration is simple: bigger tires give you slower acceleration, while smaller tires give you faster acceleration.



This relationship is the product of the physics of converting the spinning motion of a motor into the forward or reverse motion of the vehicle.

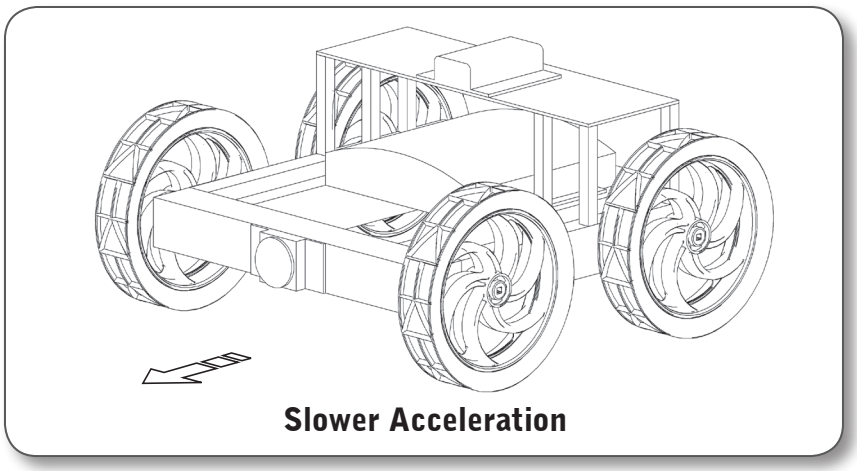
Motors generate a "spinning" force (torque), which wheels convert into a "pushing" force at the point where they contact the ground. The larger this "pushing" force is, the faster the robot will accelerate.



The relationship between torque and force is:

$$\text{Force} = \frac{\text{Torque}}{\text{Wheel Radius}}$$

A larger radius will produce a smaller force for the same amount of torque, hence the larger wheel (which has the longer distance) has a smaller force, and hence the slower acceleration.



Concepts to Understand, continued

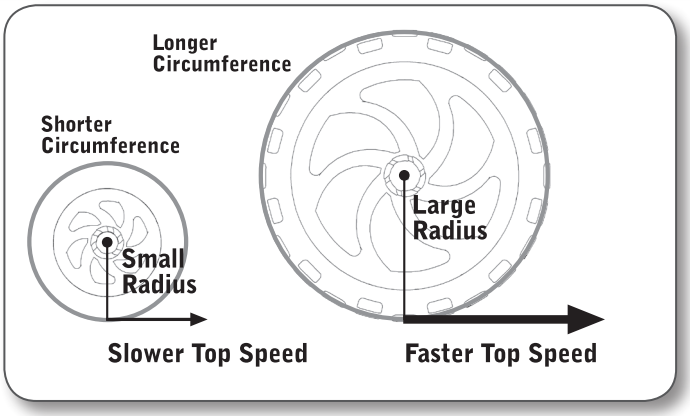
Wheels, continued

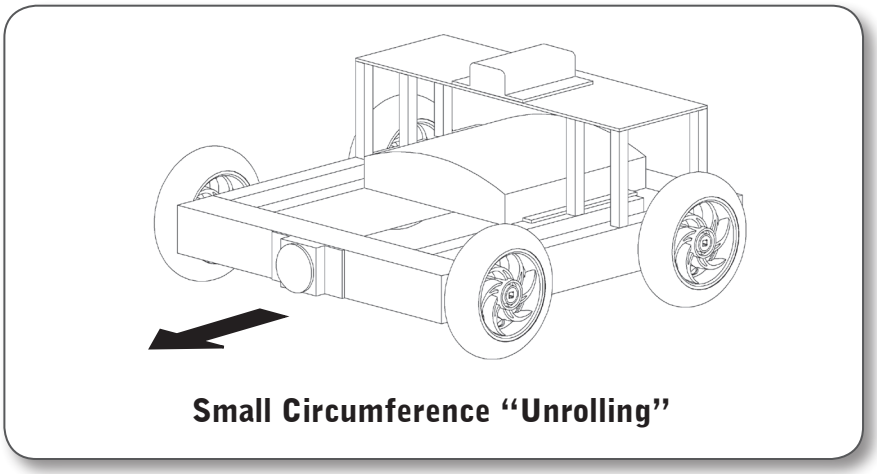
Wheel Sizes and Top Speed
Robots may take some time to reach their top speed, especially if they have high gear ratios (high gear ratio = low torque), but eventually, they tend to reach it, or at least come close.

When a wheel rolls along the ground, it is effectively "unrolling" its circumference onto the surface it is traveling on, every time it goes around. Larger wheels have longer circumferences, and therefore "unroll" farther per rotation.

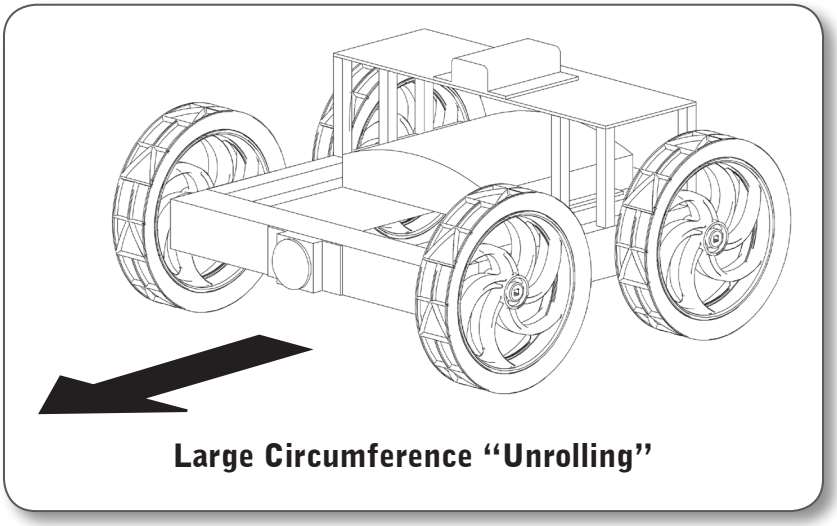
Putting these two observations together, you can see that a robot with larger wheels will have a higher top speed. The robot with larger wheels goes farther with each turn of the wheels. At top speed, robots with the same motor and gears will have their wheels turning the same number of times per second. Same number of turns times more distance per turn equals more distance, so the robot with larger wheels goes faster.

$$\text{Speed} = \text{Circumference} \times \frac{\text{turns}}{\text{second}}$$





Small Circumference "Unrolling"



Large Circumference "Unrolling"

Notice that this sets up a tough design decision, since you need to decide on a balance between acceleration and top speed when choosing a wheel size. You can't have it both ways, so you'll need to plan ahead, decide which is more important to your robot, and choose wisely.

Concepts to Understand, continued

Wheels, continued

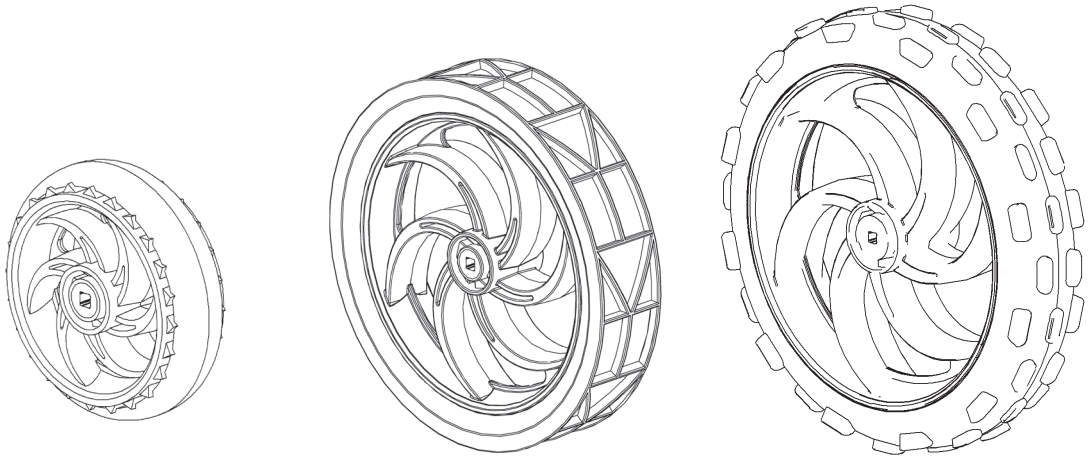
Friction

Friction occurs everywhere two surfaces are in contact with each other. It is most important when considering the wheels for your robot, however, because you will need to decide how much friction you want in order to maximize your robot's performance.

Wheel friction has both positive and negative consequences for your robot. On the one hand, friction between the wheel and the ground is absolutely essential in getting the robot to accelerate. Without friction, your robot would spin its wheels without going anywhere, like a car stuck on a patch of ice. Friction between the wheels and the ground gives the robot something to "push off" of when accelerating, decelerating, or turning.

On the other hand, wheel friction is also responsible for slowing your robot down once it is moving. A robot running over a sticky surface will go slower than one running over a smooth one, because the friction dissipates some of the robot's energy.

For a Given Surface:



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

Often there will be tradeoffs during wheel selection.

As shown above, wheel characteristics will vary greatly depending on the surface it is driving on. Some wheels which will perform well on carpet would not be as good on loose gravel.

The width, texture, and material of a tire all contribute to its friction characteristics. Again, there is no "best" solution. Rather it is a matter of picking the tire best suited to the robot's task, and the surface it will drive on.

Concepts to Understand, continued

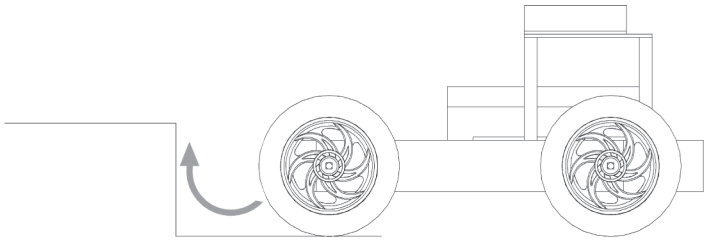
Wheels, continued

Terrain
Sometimes robots will come across physical objects that must be traversed. Both the size of a tire and the amount of friction it generates will be very important in ensuring that you can successfully navigate over them. These obstacles may be numerous and complex, so you will need to plan for them, and test your solutions to make sure that they work reliably.

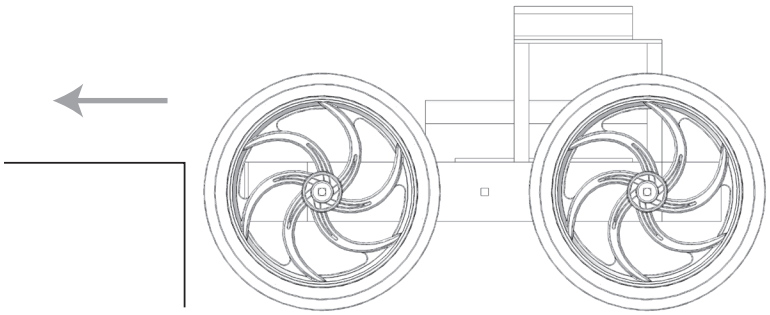
Drive trains can have a variety of functions. Be sure to design accordingly for what your robot will encounter. Remember the tradeoffs shown in this chapter and choose designs based on what your robot needs most.

NOTE: On some surfaces it is good to spread the robot load over multiple tires or a larger surface area to prevent it from sinking. Think of how snowshoes work.

Example 1:
Robot attempting to climb a step



The robot with smaller wheels has a much steeper angle to climb – in fact, it's a sheer vertical face.

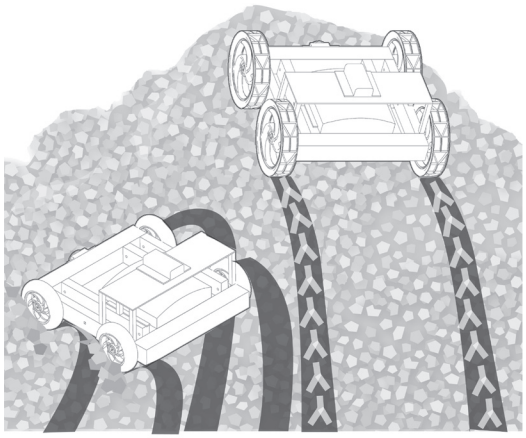


The robot with larger wheels has a much less difficult angle to climb to get up the step. This robot is much more likely to succeed.

Example 2:
Robots attempting to climb a gravel hill

The robot with slippery tires cannot get enough traction to climb the hill and slides off.

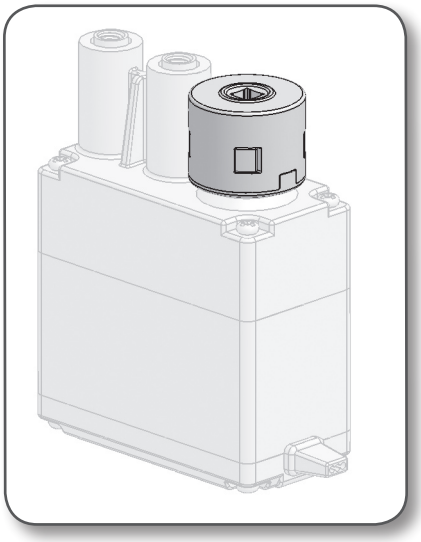
The robot with wheels that dig into the gravel can make it up the hill.



Concepts to Understand, continued

Clutches

Clutches
Every motor in the VEX Robotics Design System comes with a pre-attached clutch module. These clutch modules are designed to protect the gears internal to the motor from "shock-loads".



- The motor, even in a stall situation, CANNOT exert enough force to break the internal gears.

- The gears will break in some applications when the motor is under significant load, over a short duration of time (a shock-load).
  - The clutch is designed to absorb some of this energy in these situations by "popping" and giving way. This will protect the motor.
  - When a clutch "pops" it is briefly releasing the connection between the shaft and the motor.

- **When a clutch pops, it is doing its job.**

- **When a clutch pops, it is a sign that there is something wrong with the robot design.**

- When a robot is designed such that the load on the motor is minimized (using gearing to reduce the load) there shouldn't be any popping clutches.

- Once a clutch pops for the first time, it is easier for it to pop every time after that; for some robotics applications it may be necessary to regularly replace clutches in key areas.

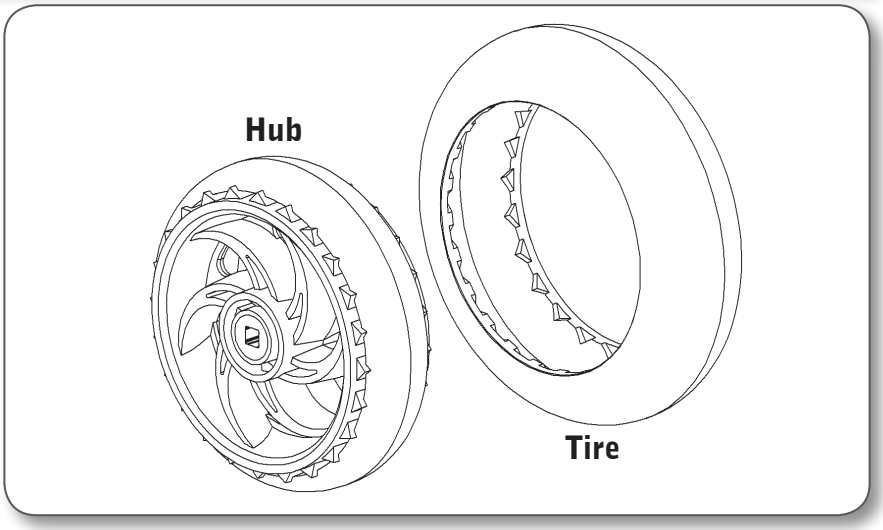
- There are some applications which do not need a clutch, however using a clutch is ALWAYS recommended. Any motor without a clutch is at risk of internal damage.

Spare clutches are available from **www.VEXrobotics.com**

Concepts to Understand, continued

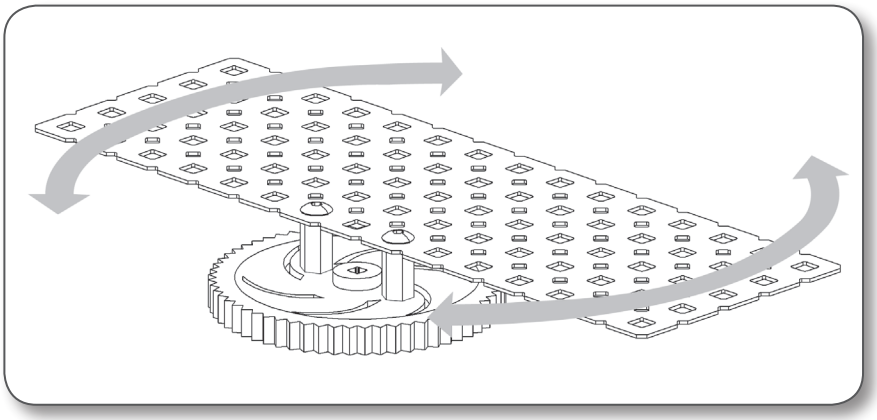
Motion Part Features

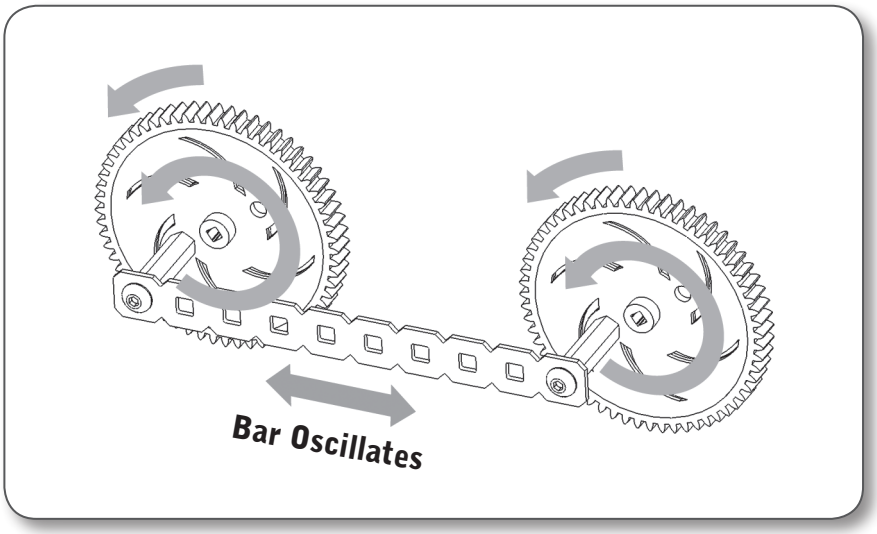
Hub and Tire
Most wheels in the VEX Robotics Design System are actually two wheels in one. By pulling off the rubbery green tire surface, the grey hubs can be used directly in different applications on your robot.



Non-Axial Mounting Points
In addition to the central hole for the gear shaft, some gears in the VEX Robotics Design System have a number of additional off-center mounting holes.

These mounting points have a number of applications. For instance, a larger structure could be built on top of the gear, which would rotate as the gear turned. Alternately, the "orbiting" motion of a non-axial mount can be used to create linear motion from rotational motion.

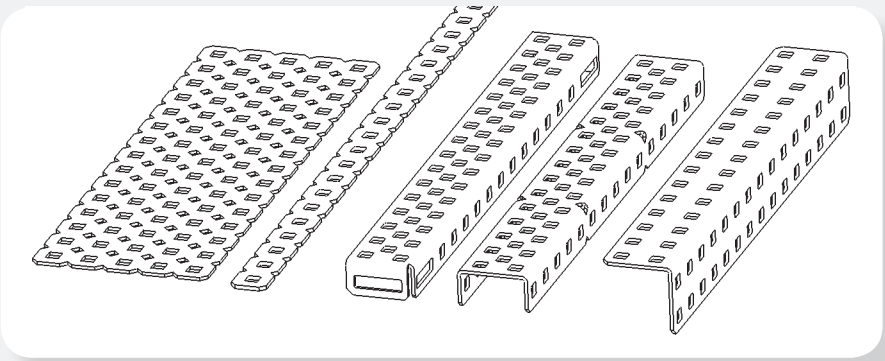




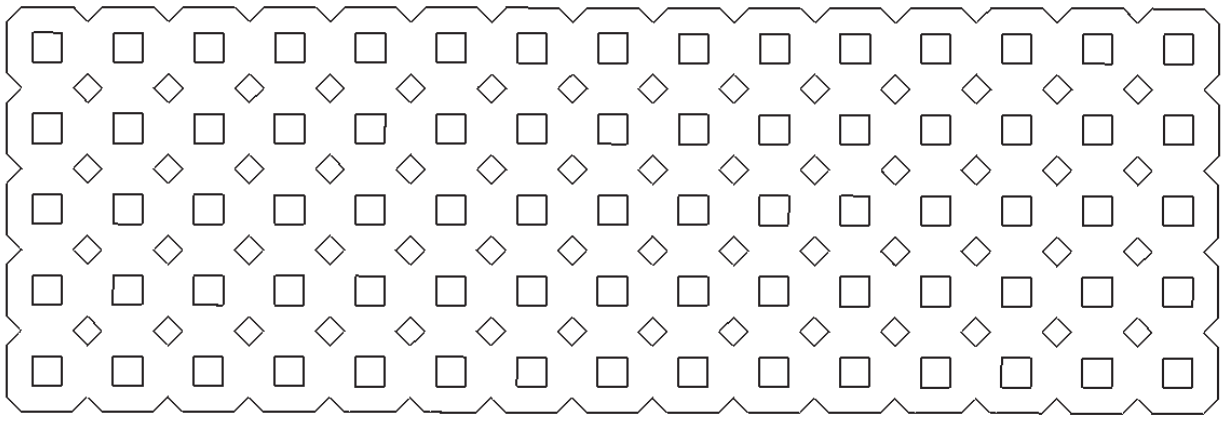
Introduction to the Structure Subsystem

The parts in the VEX 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 VEX Design System including all the metal components and hardware pieces. These pieces connect together to form the "skeleton" or frame of the robot.

In the VEX Robotics Design System the majority of the components in the Structure Subsystem are made from bent sheet-metal. These pieces (either aluminum or steel) come in a variety of shapes and sizes and are suited to different functions on a robot. Different types of parts are designed for different applications.

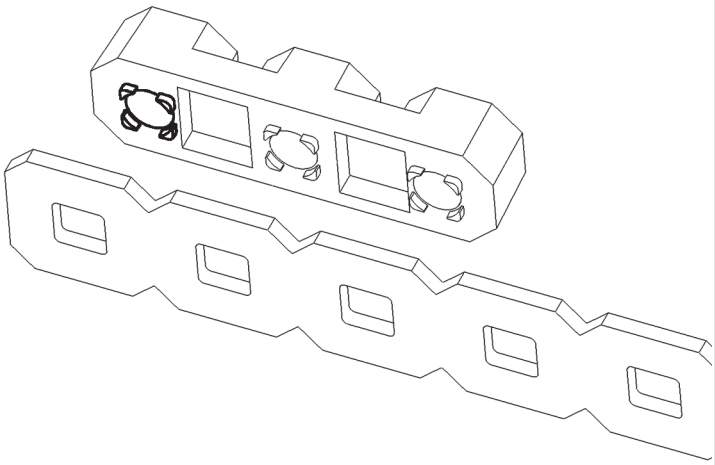


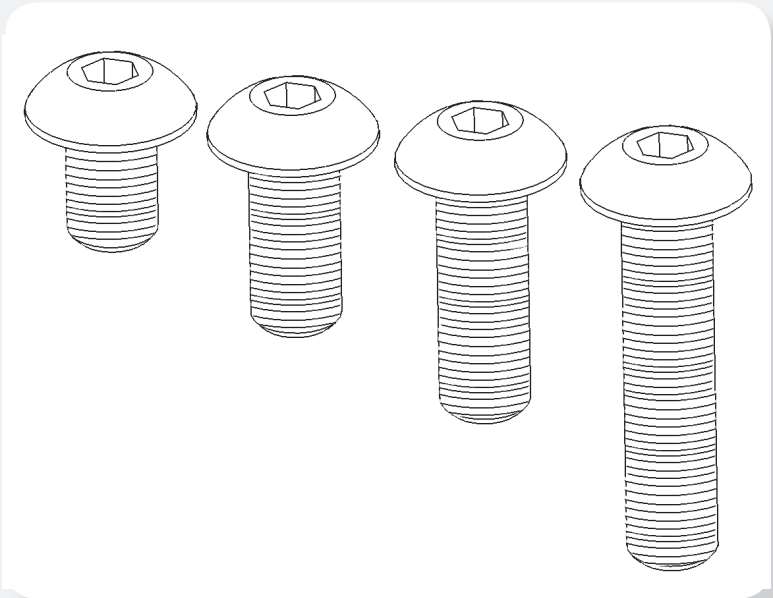
The VEX structural pieces all contain square holes (0.182" sq) on a standardized 1/2" grid. This standardized hole-spacing allows for VEX parts to be connected in almost any configuration. The smaller diamond holes are there to help users cut pieces using tin-snips or fine-toothed hacksaws without leaving sharp corners.



Introduction to the Structure Subsystem, continued

VEX square holes are also used as "alignment features" on some components. These pieces will "snap" in place into these square holes. For example, when mounting a VEX Bearing Flat there are small tabs which will stick through the square hole and hold it perfectly in alignment. This allows for good placement of components with key alignment requirements. (It would be bad if a bearing slipped out of place!) Note that hardware is still required to hold the Bearing Flat onto a structural piece.





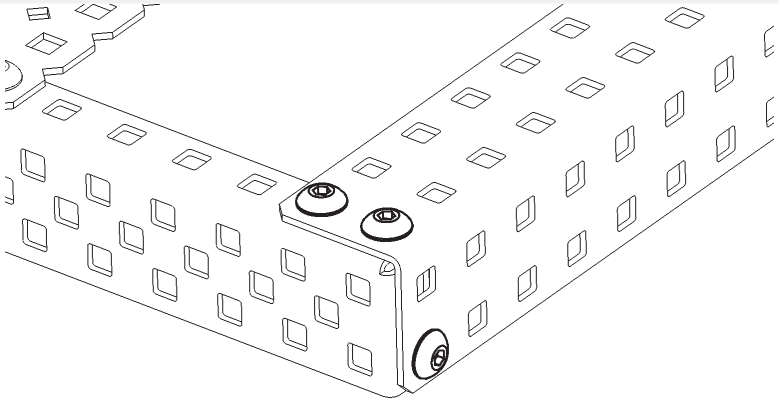
Hardware is an important part of the Structure Subsystem. Metal components can be directly attached together using the 8-32 screws and nuts which are standard in the VEX kit. The 8-32 screws fit through the standard VEX square holes. These screws come in a variety of lengths and can be used to attach multiple thicknesses of metal together, or to mount other components onto the VEX structural pieces.

Allen wrenches and other tools are used to tighten or loosen the hardware.

Note: There are two types of screws that are part of the VEX Robotics Design System.
- Size 8-32 screws are the primary screws used to build robot structure.
- Size 6-32 screws are smaller screws which are used for specialty applications like mounting the VEX Motors and Servos.

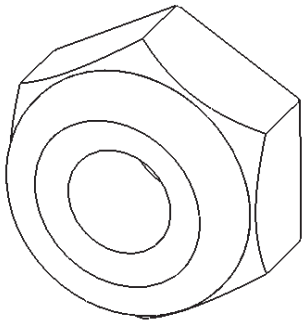
Introduction to the Structure Subsystem, continued

HINT:
Attach components together with multiple screws from different directions to keep structural members aligned correctly and for maximum strength!

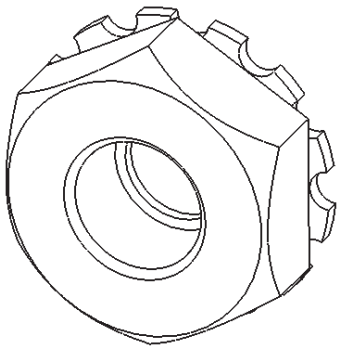


When using screws to attach things together, there are three types of nuts which can be used.

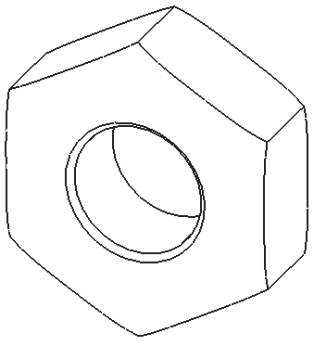
- Nylock nuts have a plastic insert in them which will prevent them from unscrewing. These are harder to install, as you need to use an open-ended wrench to tighten them up. These nuts will not come off due to vibration or movement.
- KEPS nuts have a ring of "teeth" on one side of them. These teeth will grip the piece they are being installed on. This means you do not NEED to use an open-ended wrench to tighten them (but it is still recommended). These nuts are installed with the teeth facing the structure. These nuts can loosen up over time if not properly tightened; however they will work great in most applications.
- Regular nuts have no locking feature. These basic hex nuts require a wrench to install and may loosen up over time, especially when under vibration or movement. They are very thin and can be used in some locations where it is not practical to use a Nylock or KEPS nut.



Nylock Nut



KEPS Nut



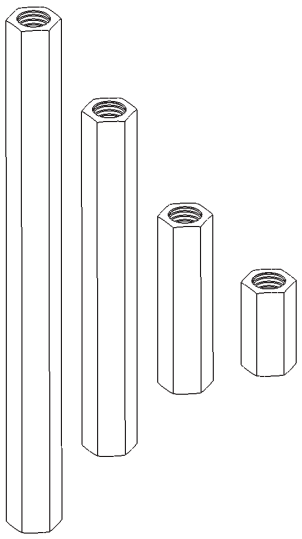
Regular (Hex) Nut

WARNING: It is important to be careful when tightening screws. The allen wrenches may round or "strip out" the socket on the head of the screw if they are not fully inserted into the socket.

Use care when tightening screws to prevent stripping out the head of the screw!

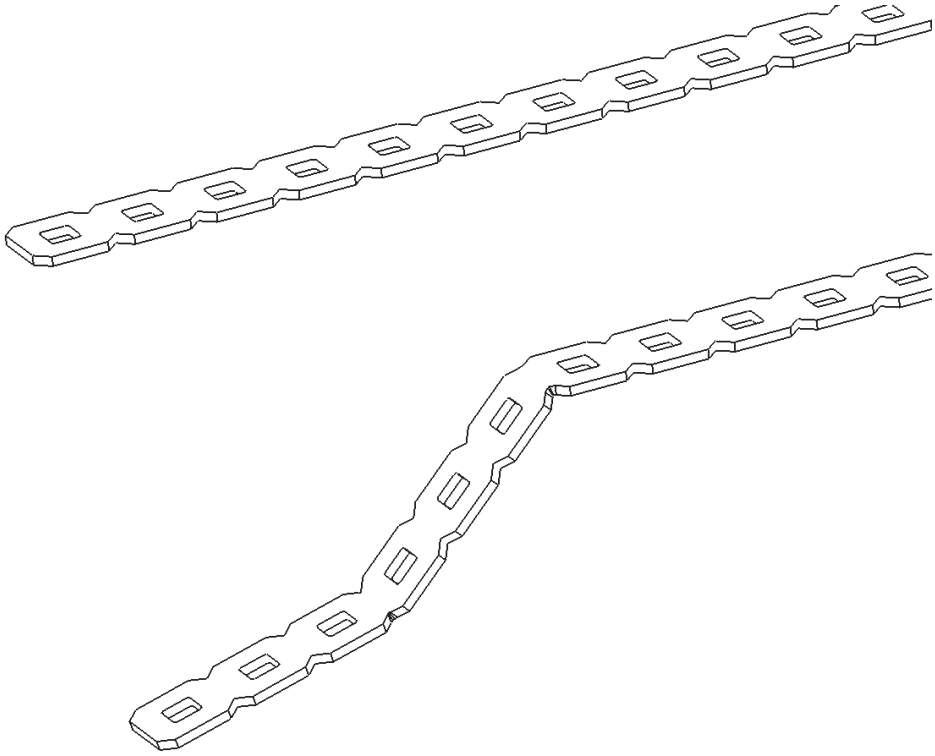
Introduction to the Structure Subsystem, continued

Components can also be offset from each other using 8-32 threaded standoffs; these standoffs come in a variety of lengths and add great versatility to the VEX kit. These standoffs work great for mounting components in the VEX system as well as for creating structural beams of great strength.



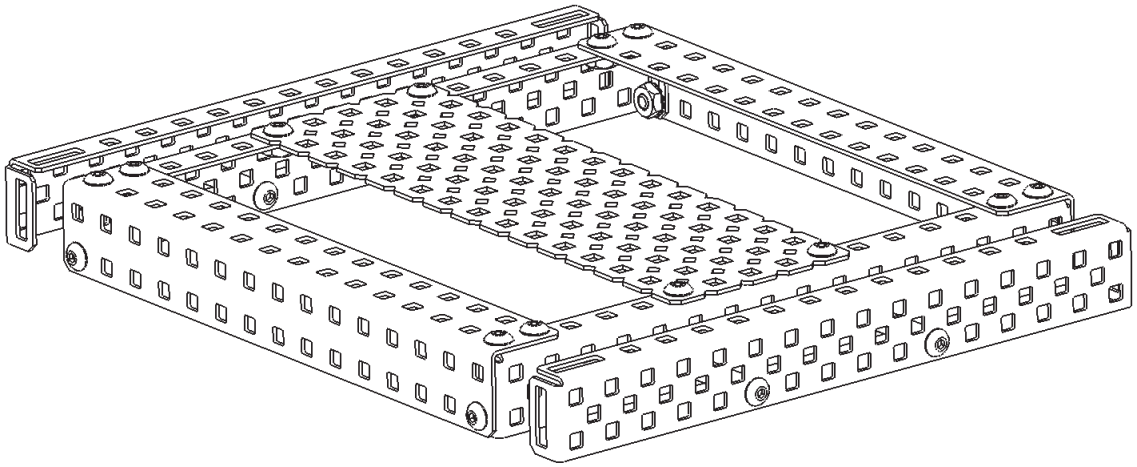
One of the key features of many VEX structural parts is their "bend-able" and "cut-able" nature. Users can easily modify many of these structural parts into new configurations better suited for their current needs. Flat plates can be bent into brackets. Many metal components can be cut to custom lengths. These parts were DESIGNED to be modified.

Note: It is almost impossible to fully flatten a piece once it has been bent.



Introduction to the Structure Subsystem, continued

The VEX structural components come in a variety of shapes and sizes. Each of these structural shapes may be strong in some ways but weak in others. It is very easy to bend a piece of VEX Bar in one orientation, but it is almost impossible to bend it when it is in another orientation. Applying this type of knowledge is the basis of structural engineering. One can experiment with each piece and see how it can be used to create an extremely strong robot frame!



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.

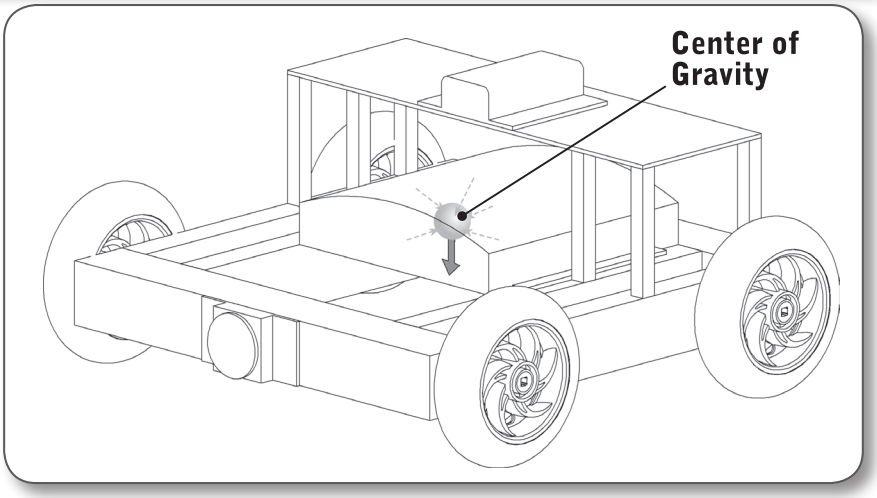
The frame is the skeleton of the robot and should be designed to be integrated cleanly with the robot's other components. The overall robot design should dictate the chassis, frame, and structural design; not vice-versa.

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

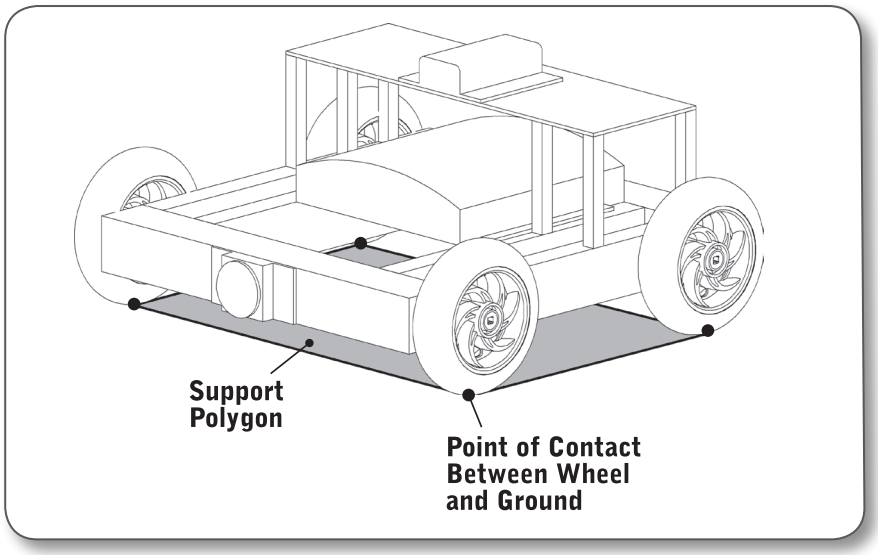
Concepts to Understand

Stability: Center of Gravity Considerations

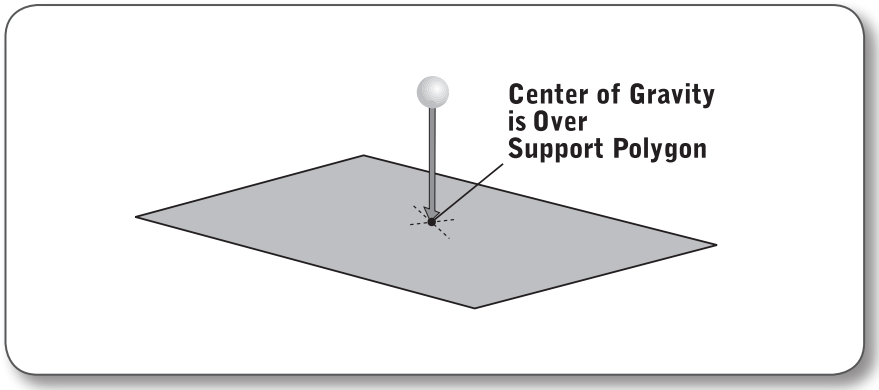
Center of Gravity
You can think of the robot's center of gravity as the "average position" of all the weight on the robot. Because it is an average of both weight and position, heavier objects count more than lighter ones in determining where the center of gravity is, and pieces that are farther out count more than pieces that are near the middle.



Support Polygon
The support polygon is 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.



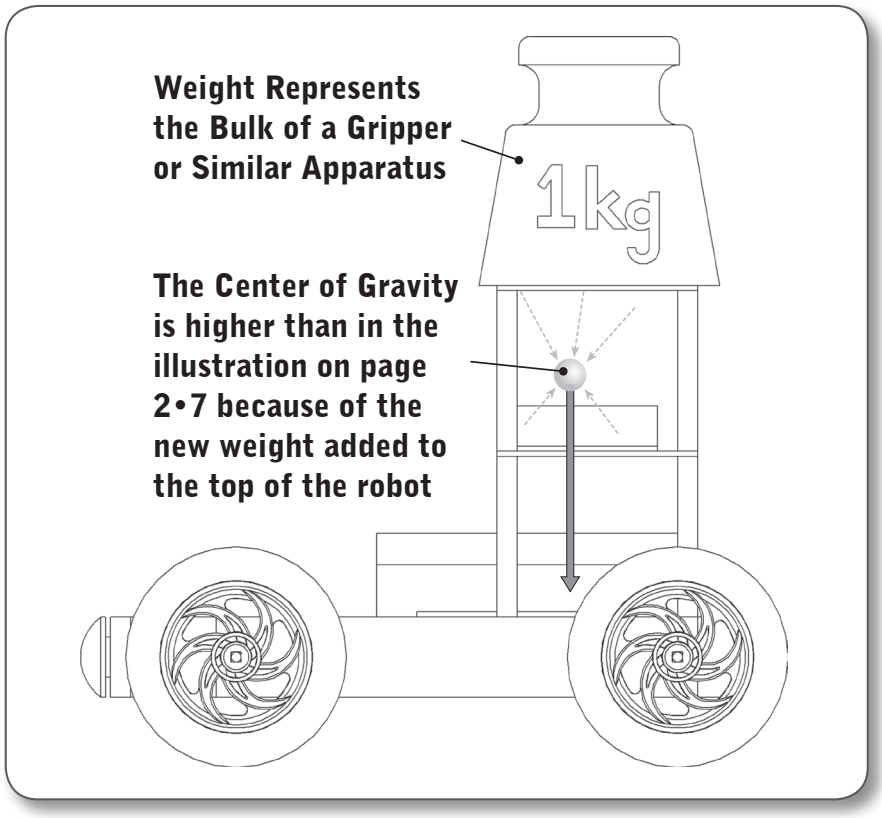
Stability
The rule for making a robot stable is very simple: the robot will be most stable when the center of gravity is centered over the support polygon. Your robot will encounter much more complex situations than just standing still; you need to take these into account when making your design.



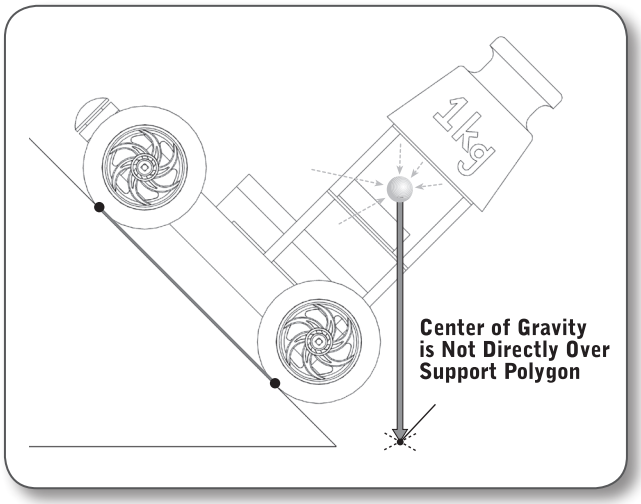
Concepts to Understand, continued

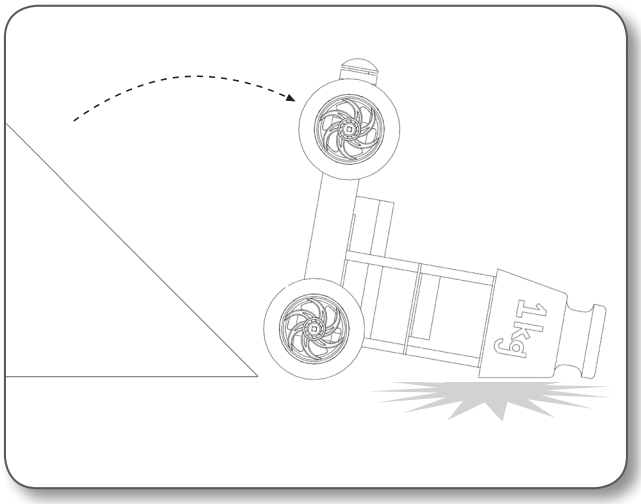
Stability: Center of Gravity Considerations, continued

EXAMPLE 1: Towerbot
This robot was built very tall so that it would be able to reach a hanging goal for a challenge. However, along the way, it had to first climb a ramp.



Notice that the robot's center of gravity is no longer over the support polygon. This robot would fall over as soon as it started up the ramp.



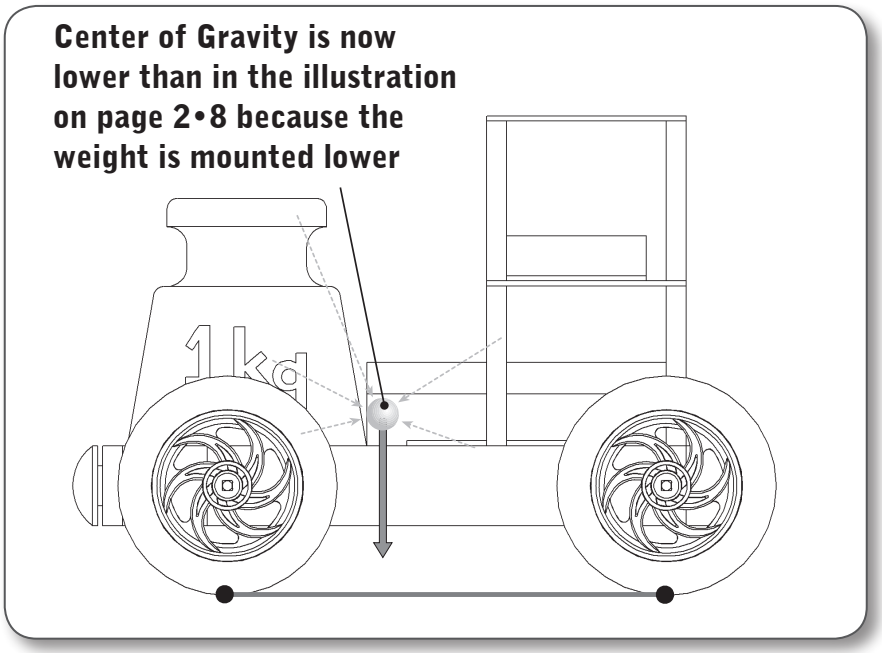


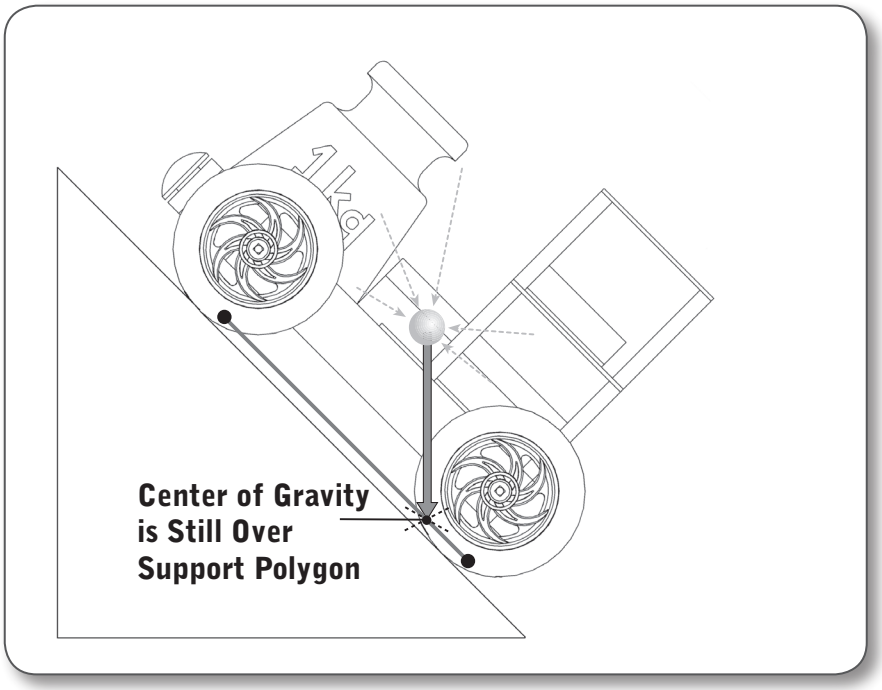
Concepts to Understand, continued

Stability: Center of Gravity Considerations, continued

EXAMPLE 1: Towerbot, continued

To fix this problem, you must lower the robot's center of gravity so that it does not move as far when the robot is on an incline. In general, it is advantageous to have your robot's center of gravity as close to the ground as possible!



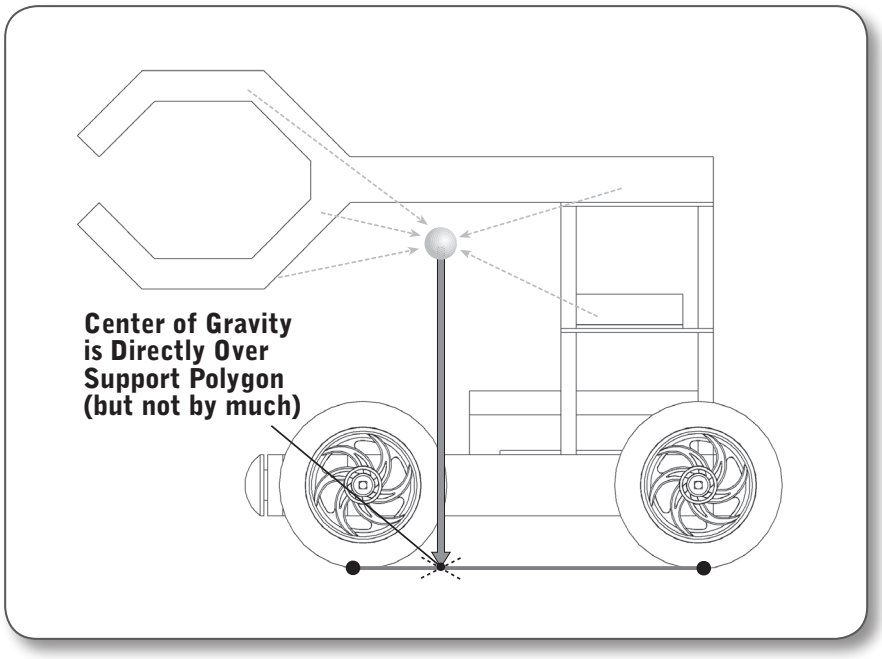


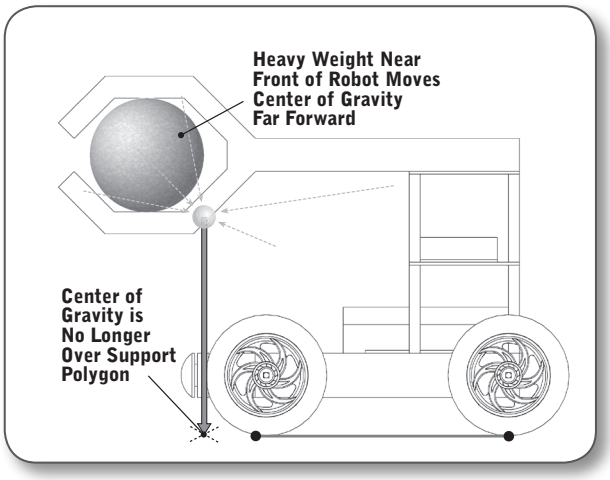
Concepts to Understand, continued

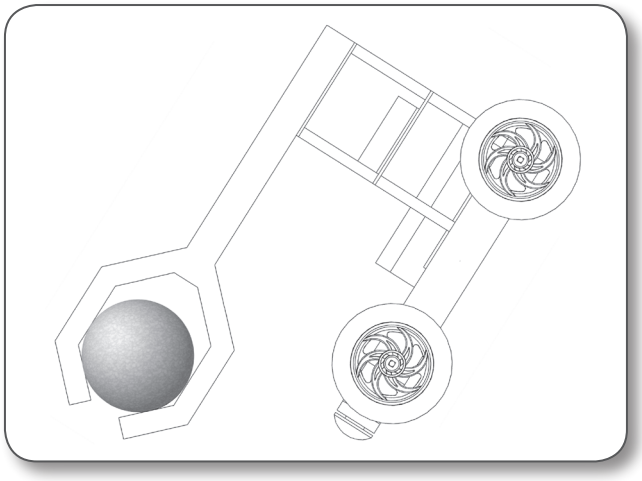
Stability: Center of Gravity Considerations, continued

EXAMPLE 2: Grabberbot
This robot is designed to pick up a heavy object using the gripper claw on the front, and transport the object to another location.

When the robot picks up the object, it effectively adds the object's weight to the robot's structure. The combined robot-ball structure now has the new center of gravity (shown below), which is outside the support polygon. The robot tips over as a consequence.







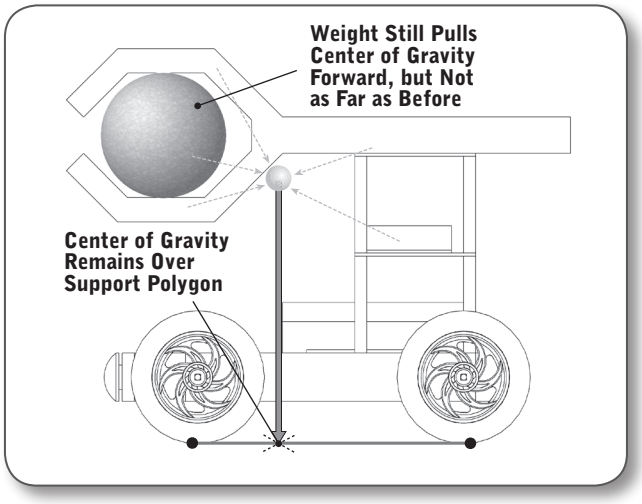
Concepts to Understand, continued

Stability: Center of Gravity Considerations, continued

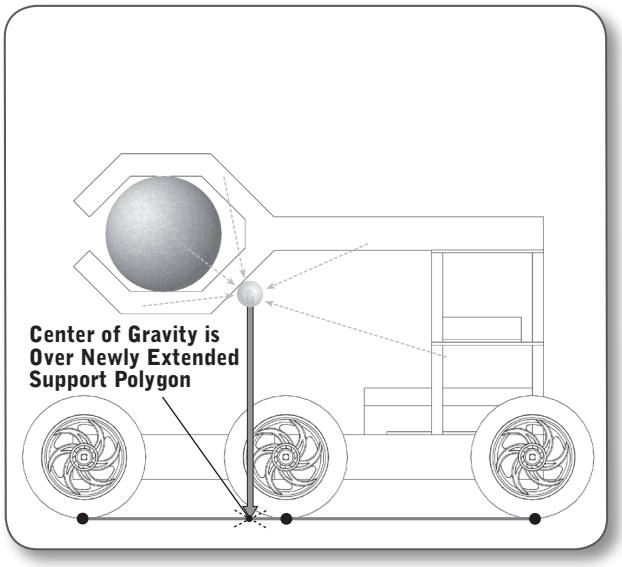
EXAMPLE 2: Grabberbot, continued

There are many solutions to this problem. Depending on the specifics of the challenge, some or all of these modifications could work:

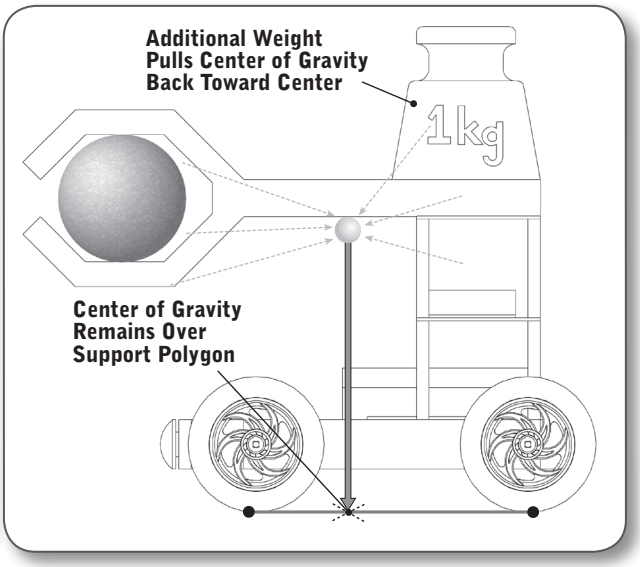
Moving the center of gravity back by moving the gripper farther back on the robot



Extending the support polygon by adding more wheels farther out



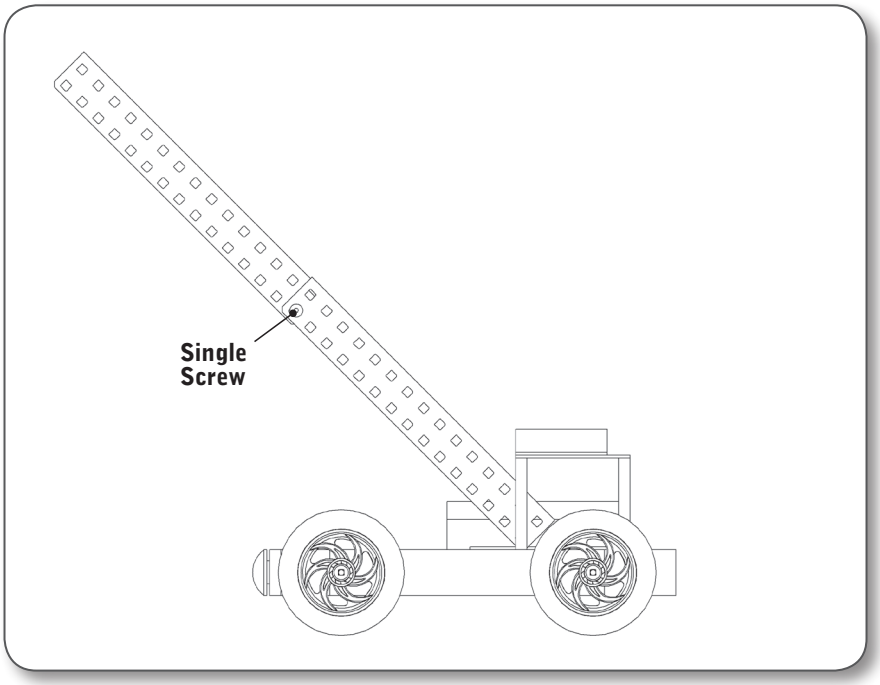
Moving the center of gravity back by adding counterweights on the back of the robot



Concepts to Understand, continued

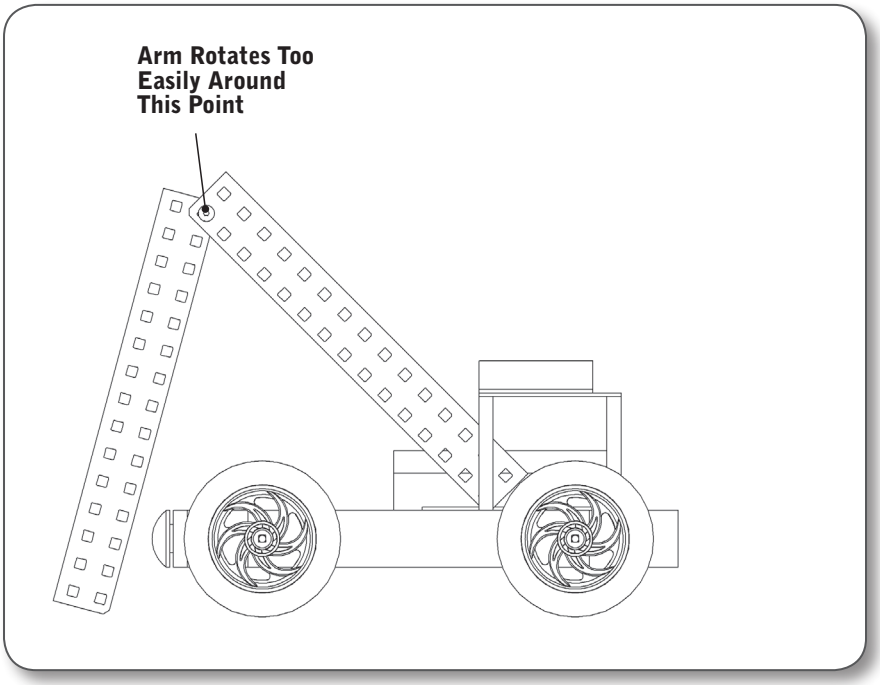
Robust Fabrication

Fasteners
The most common problem with robots that fall apart or lose pieces easily is that groups of parts are not joined securely enough and separate from each other and move around.



EXAMPLE 1:
Arm Extension
A robot needs to be able to reach a goal that is high off the ground. The goal is so high that a single long piece will not reach it. Two pieces must be joined together to reach the desired height.

This attachment uses a single screw to join the two bars. As you can see, it has a problem when weight is applied to it: the extension bar rotates around the screw. Also, if this screw were to come loose or fall out for any reason, the entire arm would come crashing down.

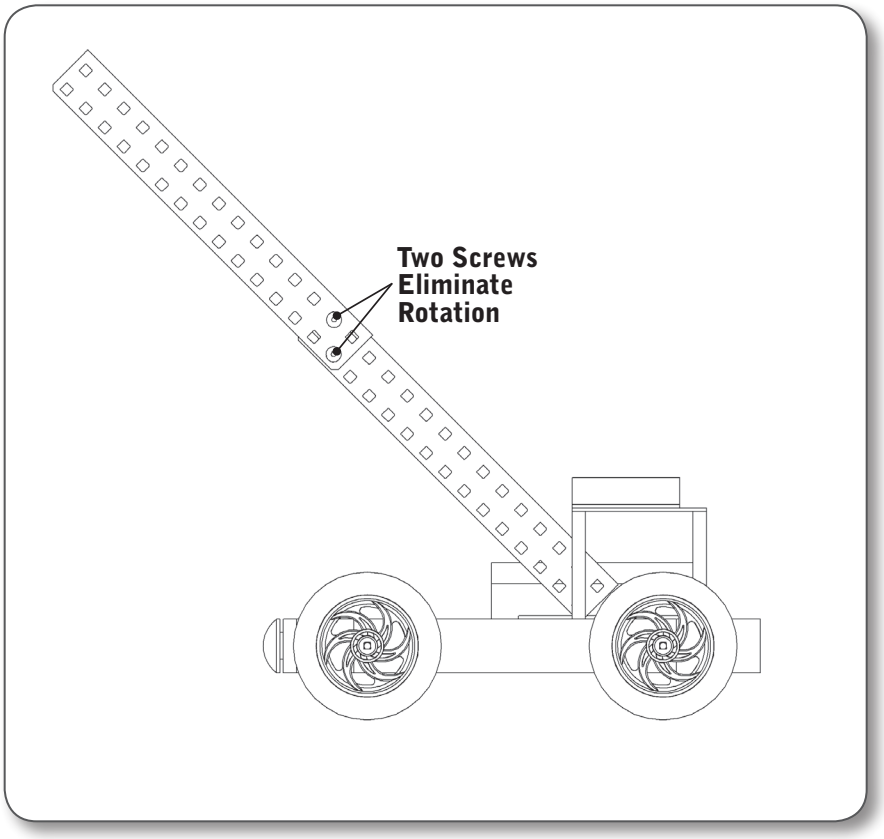


Concepts to Understand, continued

Robust Fabrication, continued

EXAMPLE 1, continued: Arm Extension, continued

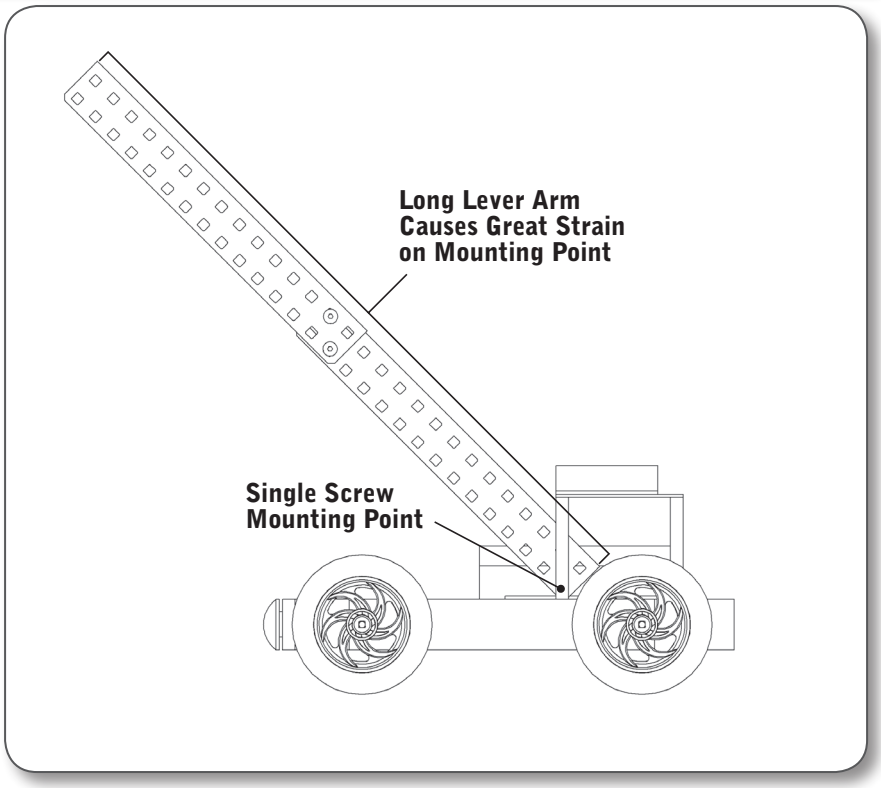
By using two screws, this design removes the possibility of rotation around either one of them. Additionally, the design is more resilient.

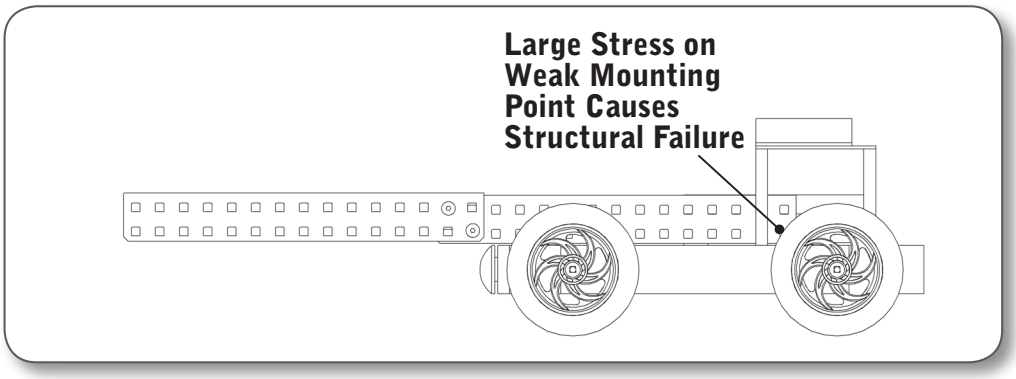


Concepts to Understand, continued

Robust Fabrication, continued

EXAMPLE 2:
Bracing
The extended bars are now attached firmly to each other, and the long arm is mounted on your robot. However, the long arm is going to generate huge stresses at its mounting point because it is so long, especially when the arm is used to lift a load.





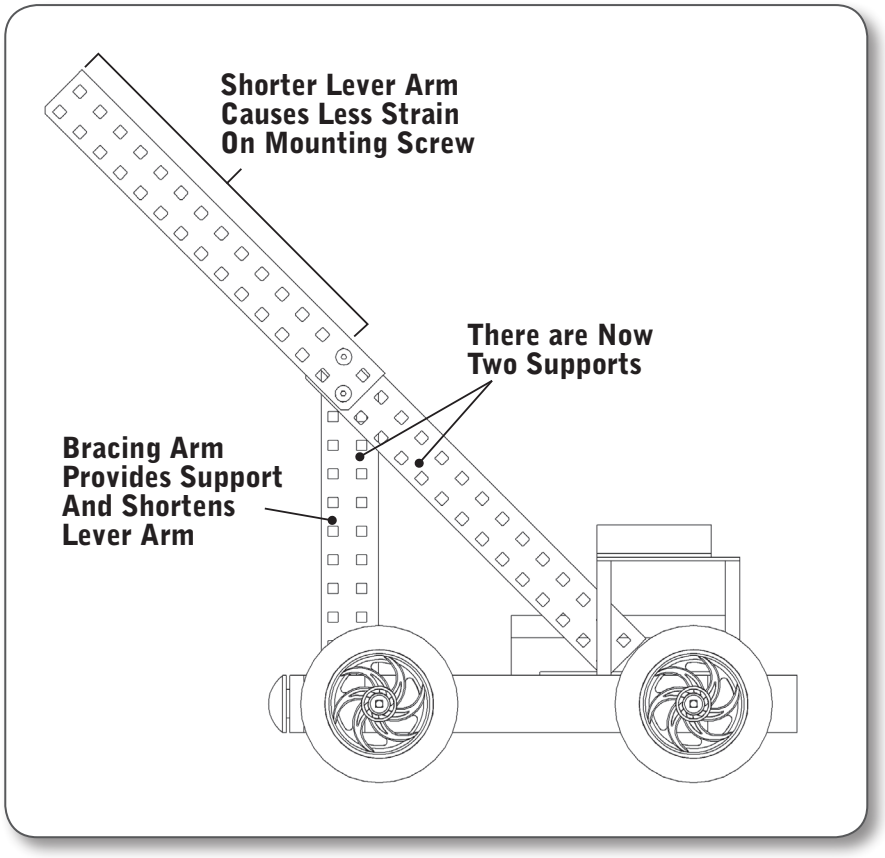
Concepts to Understand, continued

Robust Fabrication, continued

EXAMPLE 2:
Bracing the Bars, continued

In order to keep the arm from falling down, you will need to brace it. You could use a second screw to hold it, like you did with the arm itself, but because the arm is such a long lever arm, that screw would actually be in danger of deforming or breaking. A better solution would be to give the structure support at a point closer to the end, thus reducing the mechanical advantage that the arm has relative to the supports.

The arm is now more stable and better able to withstand stresses placed on it from both its own weight, and any external forces acting on it. The bracing arm has both decreased the mechanical advantage from the long lever arm, and spread the load over two supports instead of just one.



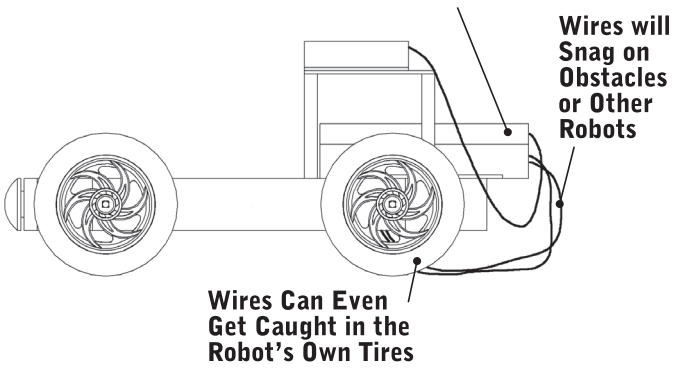
Concepts to Understand, continued

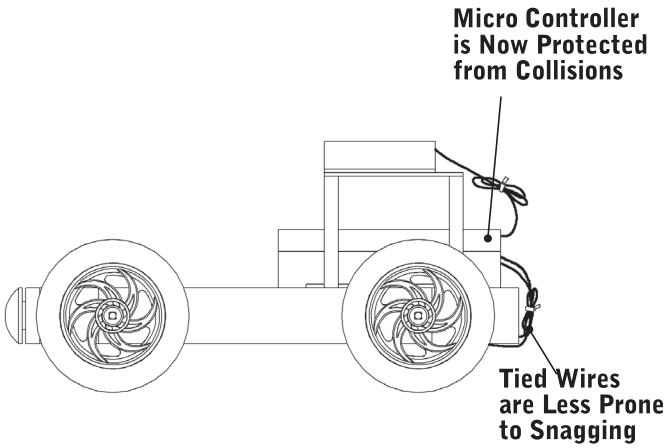
Exposure and Vulnerability

There are certain parts of a robot that are more fragile than others. Always plan the structural design to protect these parts from unwanted physical contact if possible.

The design in the example at the right is asking for trouble. The VEX Microcontroller is a sensitive piece of electronic equipment, and it can be a poor design decision to put it somewhere it could be damaged by a simple physical impact. In particular, this design leaves the back of the VEX Microcontroller exposed in such a way that a passing robot or a careless driver could smash the entire rear connector panel, potentially damaging the radio control and power connections. Also, the wires are a mess. Wires should be secured and protected, because if one of those wires were to snag on another robot (or even on the robot's own wheels!), the connector would be forcibly removed from its port. Not only would this disable the robot on the field, but it could cause permanent damage to the cable or the ports on the VEX Microcontroller.

Adjusting the position of the controller so that it is not likely to get hit by anything, and cleaning up the wires (the kit comes with wire ties/tie wraps/ zip ties) will reduce the chance of damage to the sensitive electronic components on the robot. As a bonus, it looks a lot cleaner as well.





TIP:
For the best protection ensure all robot components that can be damaged are well shielded and inside robot structure. Route wires inside the robot and away from all moving components.

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

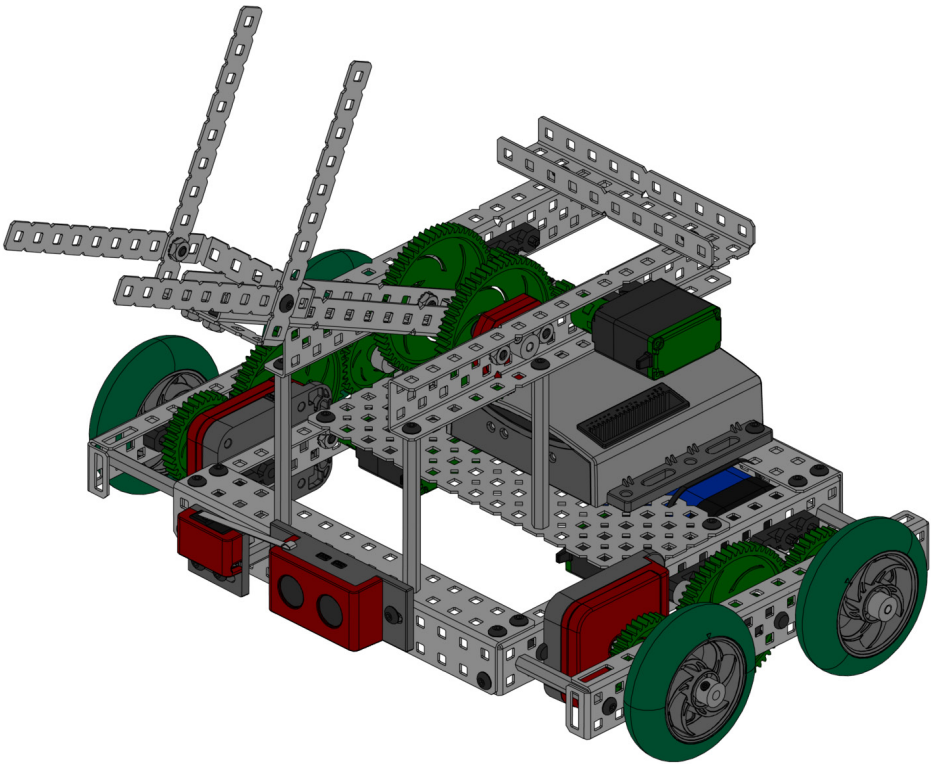
ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

UNIDAD 2

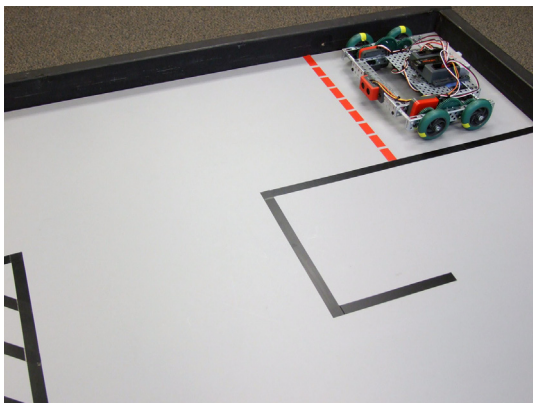


Behaviors

A **behavior** is anything your robot does: turning on a single motor is a behavior, moving forward is a behavior, tracking a line is a behavior, navigating a maze is a behavior. There are three main types of behaviors that we are concerned with: **basic** behaviors, **simple** behaviors, and **complex** behaviors.

Basic Behaviors

Example: Turn on Motor Port 3 at half power



At the most basic level, everything in a program must be broken down into tiny behaviors that your robot can understand and perform directly. In ROBOTC, these are behaviors the size of **single statements**, like **turning on a single motor**, or **resetting a timer**.

Simple Behaviors

Example: Move forward for 2 seconds

Simple behaviors are small, bite-size behaviors that allow your robot to perform a **simple, yet significant task**, like **moving forward for a certain amount of time**. These are perhaps the most useful behaviors to think about, because they are big enough that you can describe **useful actions** with them, but small enough that you can program them easily from basic ROBOTC commands.

Complex Behaviors

Example: Follow a defined path through an entire maze

These are behaviors at the **highest levels**, such as **navigating an entire maze**. Though they may seem complicated, one nice property of complex behaviors is that they are always composed of smaller behaviors. If you observe a complex behavior, you can always break it down into smaller and smaller behaviors until you eventually reach something you recognize.

```
task main()
{
  motor[leftMotor] = 63;
  motor[rightMotor] = 63;
  wait1Msec(2000);

  motor[leftMotor] = -63;
  motor[rightMotor] = 63;
  wait1Msec(400);

  motor[leftMotor] = 63;
  motor[rightMotor] = 63;
  wait1Msec(2000);
}
```

Basic behavior
This code turns the left motor on at half power.

Simple behavior
This code makes the robot go forward for 2 seconds at half power.

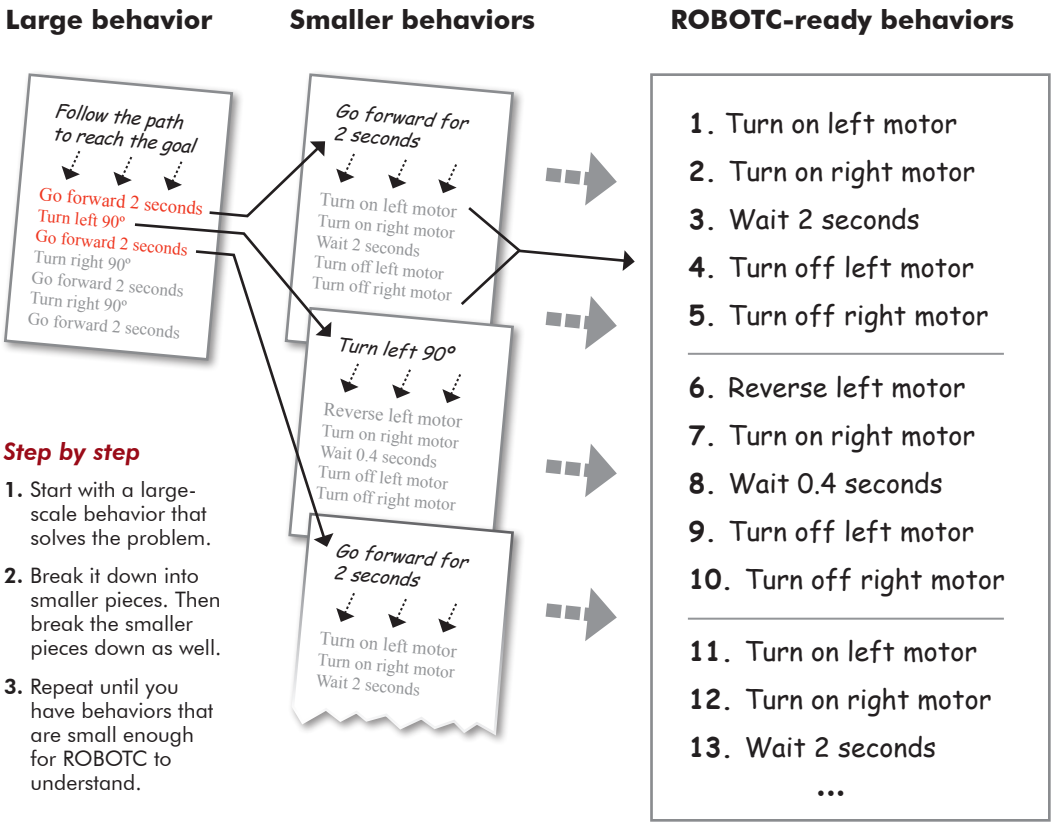
Complex behavior
This code makes the robot move around a corner.

Behaviors

Composition and Analysis

Perhaps the most important idea in behaviors is that they can be **built up or broken down into other behaviors**. Complex behaviors, like going through a maze, can always be broken down into **smaller, simpler behaviors**. These in turn can be broken down further and further until you reach simple or basic behaviors that you recognize and can program.

By looking back at the path of behaviors you broke down, you can also see how the smaller behaviors should be programmed so that they **combine back together**, and produce the larger behavior. In this way, analyzing a complex behavior **maps out the pieces** that need to be programmed, then allows you to **program them**, and **put them together** to build the final product.



Step by step

1. Start with a large-scale behavior that solves the problem.
2. Break it down into smaller pieces. Then break the smaller pieces down as well.
3. Repeat until you have behaviors that are small enough for ROBOTC to understand.

Sometimes it can be hard to tell whether a behavior is "simple" or "complex". Some programs are so complex they need multiple layers of simple behaviors before they reach the basic ones!

"Basic," "Simple," and "Complex" are categories of behaviors which are meant to help you **think about the structure of programs**. They are points of reference in the world of behaviors. Use these distinctions to help you, but don't worry if your "complex" behavior suddenly becomes a "simple" part of your next program... just pick the point of reference that's most useful for what you need.

Pseudocode & Flow Charts

Pseudocode is a **shorthand notation for programming** which uses a combination of **informal programming structures and verbal descriptions of code**. Emphasis is placed on expressing the behavior or outcome of each portion of code rather than on strictly correct syntax (it does still need to be reasonable, though).

In general, pseudocode is used to outline a program before translating it into proper syntax. This helps in the initial planning of a program, by creating the logical framework and sequence of the code. An additional benefit is that because pseudocode does not need to use a specific syntax, it can be translated into different programming languages and is therefore somewhat universal. It captures the **logic and flow of a solution** without the bulk of strict syntax rules.

Below is some pseudocode written for a program which moves as long as a touch sensor is not pressed, but stops and turns to the right if its sonar detects an object less than 20in away.

```
task main()
{
 while ( touch sensor is not pressed )
  {
    Robot runs forward

    if (sonar detects object < 20in away)
    {
      Robot stops

      Robot turns right
    }
  }
}
```

Some intact syntax
The use of a while loop in the pseudocode is fitting because the way we read a while loop is very similar to the manner in which it is used in the program.

Descriptions
There are no actual motor commands in this section of the code, but the pseudocode suggests where the commands belong and what they need to accomplish.

This pseudocode example includes elements of both programming language, and the English language. Curly braces are used as a visual aid for where portions of code need to be placed when they are finally written out in full and proper syntax.

Pseudocode & Flow Charts

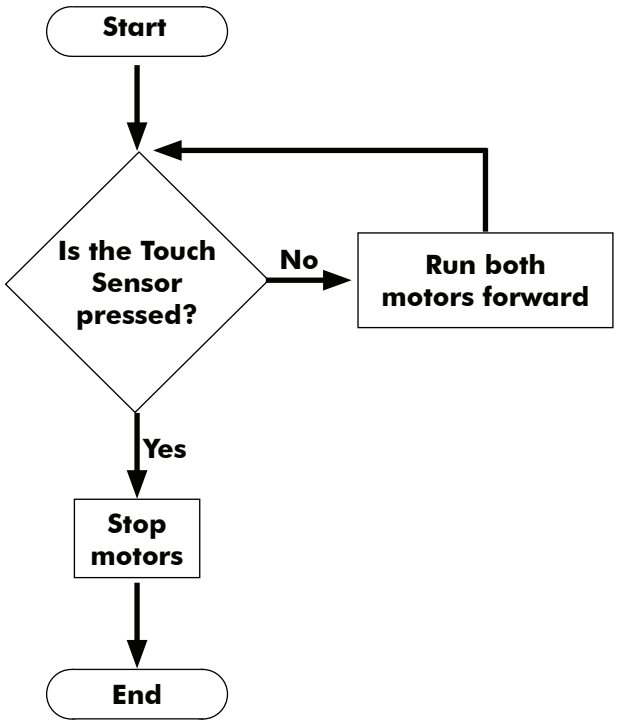
Flow Charts are a **visual representation of program flow**. A flow chart normally uses a combination of **blocks** and **arrows** to represent actions and sequence. Blocks typically represent **actions**. The **order** in which actions occur is shown using arrows that point from statement to statement. Sometimes a block will have multiple arrows coming out of it, representing a step where a **decision** must be made about which path to follow.

Start and End symbols are represented as rounded rectangles, usually containing the word "Start" or "End", but can be more specific such as "Power Robot Off" or "Stop All Motors".

Actions are represented as rectangles and act as basic commands. Examples: "wait1Msec(1000)"; "increment LineCount by 1"; or "motors full ahead".

Decision blocks are represented as diamonds. These typically contain Yes/No questions. Decision blocks have two or more arrows coming out of them, representing the different paths that can be followed, depending on the outcome of the decision. The arrows should always be labeled accordingly.

To the right is the flow chart of a program which instructs a robot to run forward as long as its touch sensor is not pressed. When the touch sensor is pressed the motors stop and the program ends.



To read the flow chart:

- Start at the "**Start**" block, and follow its arrow down to the "Decision" block.
- The **decision block** checks the status of the touch sensor against two possible outcomes: the touch sensor is either pressed or not pressed.
- If the touch sensor is not pressed, the program follows the **"No" arrow** to the action block on the right, which tells the motors to run forward. The arrow leading out of that block points back up and around, and ends back at the Decision block. This forms a **loop**!
- The **loop** may end up repeating many times, as long as the Touch Sensor remains unpressed.
- If the touch sensor is pressed, the program follows the "Yes" arrow and stops the motors, then ends the program.

White Space

White Space is the use of **spaces, tabs, and blank lines** to visually organize code. Programmers use White Space since it can group code into sensible, readable chunks without affecting how the code is read by a machine. For example, a program that would run a robot forward for 2 seconds, and then backward for 4 seconds, could look like either of these:

Program Without White Space

```
task main()
{
  bMotorReflected[port2]=1;
  motor[port3]=127;
  motor[port2]=127;
  wait1Msec(2000);
  motor[port3]=-127;
  motor[port2]=-127;
  wait1Msec(4000);
}
```

Program With White Space

```
task main()
{
  bMotorReflected[port2]=1;
  motor[port3]=127;
  motor[port2]=127;
  wait1Msec(2000);

  motor[port3]=-127;
  motor[port2]=-127;
  wait1Msec(4000);

}
```

Both programs will perform the same, however, the **second uses white space** to organize the code to **separate the program's two main behaviors**: moving forward and moving in reverse. In this case, line breaks (returns) were used to vertically segment the tasks. Horizontal white space characters like spaces and tabs are also important. Below, white space is used in the form of indentations to indicate which lines are within which control structures (task main, while loop, if-else statement).

Program Without White Space

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

Program With White Space

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

Comments

Commenting a program means using descriptive text to explain portions of code. The compiler and robot both ignore comments when running the program, allowing a programmer to leave important notes in non-code format, right alongside the program code itself. This is considered very good programming style, because it cuts down on potential confusion later on when someone else (or even you) may need to read the code.

There are two ways to mark a section of text as a comment rather than normal code:

Type	Start Notation	End Notation
Single line	//	(none)
Multiple line	/*	*/

Below is an example of a program with single and multi-line comments. Commented text turns green.

```
/*
   This program uses commenting
   to describe each process.
*/
task main()
{
  bMotorReflected[port2] = 1; //reflect rotation on port 2
  motor[port3]=127;   //port3 receives full power
  motor[port2]=127;   //port2 receives full power
  wait1Msec(5000);    //both motors run for 5 sec.
}
```

"Commenting out" Code

Commenting is also sometimes used to temporarily "disable" code in a program without deleting it. In the program below, the programmer has code to run straight and then turn right. However, in order to test just the first part of the program, the programmer made the second behavior into a comment, so that the robot would ignore it. When the programmer is done testing the first behavior, he/she will remove the // comment marks to re-enable the second behavior in the program.

```
task main()
{
  bMotorReflected[port2] = 1;
  motor[port3]=127;
  motor[port2]=127;
  wait1Msec(5000);

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

Reserved Words

Motors

Motor control and some fine-tuning commands.

`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 stop.

```
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
```

Timing

The VEX allows you to use Wait commands to insert delays into your program. It also supports Timers, which work like stopwatches; they count time, and can be reset when you want to start or restart tracking time elapsed.

`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
```

Reserved Words

`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.)
```

`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.)
```

Reserved Words

`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
```

`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
}
```

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
Digital Out	Digital	0 or 1

Sounds

The VEX can play sounds and tones using an external piezoelectric speaker attached to a motor port.

`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
```

Reserved Words

Radio Control

ROBOTC allows you to control your robot using input from the Radio Control Transmitter.

`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.

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

```
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
}
```

Reserved Words

Miscellaneous

Miscellaneous useful commands that are not part of the standard C language.

`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
```

Control Structures

Program control structures in ROBOTC enable a program to control its flow outside of the typical top to bottom fashion.

`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(time1[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

Different types of information require different types of variables to hold them.

`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
```

`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
```

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

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

Remote Control Buttons

In this exercise you will:

1. Program the buttons on your remote controller.
2. Identify the names and locations of all buttons on the VEXnet Remote Control.

Remote Control Overview

The VEXnet Remote Control is a very powerful tool that a programmer can use to use to achieve direct control of their robot. Each button can be programmed to control a specific behavior, for example - goStraight, rightTurn, leftTurn, openGripper, closeGripper - allowing limitless options.

Joysticks: Each remote control has two josticks. They are the round knobs that are labeled 1+2 and 3+4 on the picture on the left. To access the y-axis of right joystick the command would be "**`vexRT[Ch2]`**".

The joystick axis names are:

- **`Ch1`**
- **`Ch2`**
- **`Ch3`**
- **`Ch4`**

Note: ROBOTC has the capability of working with two remote controls at a time. Names for the second remote control are appended by Xmtr2. For example, to access the y-axis of right joystick on the second remote control, the command would be "**`vexRT[Ch2Xmtr2]`**".

Buttons: There are 12 programmable buttons on the remote control. The eight buttons on the front are broken into two groups of four, each having up, down, left, and right buttons. Two groups of up and down buttons make up the additional four buttons on the top of the remote control.

Accessing button values in ROBOTC is very similar to accessing joystick values. The vexRT[] command is still used, but now you use the letters "Btn", followed by the group number it belongs to, and finally the letter U, D, L, or R, depending on the buttons direction. For example, if you wanted to access the value of the down button on the front-left of the remote control you would use **`vexRT[Btn7D]`** short for vexRT Button Group 7 Down.

Note: Button names for the second remote control are also appended by Xmtr2. For example, to access the value of the down button on the front-left of the remote control, the command would be "**`vexRT[Btn7DXmtr2]`**".

Engineering Lab

Remote Control Review

This exercise assumes you understand how VEXnet communications works, that you have programmed your robot to move around using the joysticks, and that you have slowed down your robot's speed using different control mappings. If you have not completed those lessons, do so before beginning this exercise.

Things to Remember with Every Remote Control Program

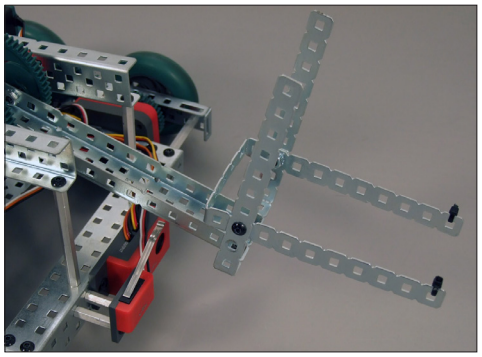
1. Every remote control program will use the vexRT[] command to access the values of the joysticks and buttons.
2. Remote control commands for the robot must be placed in a while loop for the human operator to maintain smooth, continuous control.

Code Review

```
1  #pragma config(Motor, port2, rightMotor, tmotorNormal, openLoop, reversed)
2  #pragma config(Motor, port3, leftMotor,  tmotorNormal, openLoop)
3  //*!!Code automatically generated by 'ROBOTC' configuration wizard !!*//
4
5  task main()
6  {
7    while(1 == 1)
8    {
9     motor[leftMotor] = vexRT[Ch3] / 2;
10    motor[rightMotor] = vexRT[Ch2] / 2;
11   }
12 }
```

Programming Remote Control Buttons

Remote control buttons behave the same way as the VEX touch sensors. When a touch sensor is pressed, it returns a value of "1". When it is not pressed, it returns a value of "0". Similarly, buttons return a value of "1" when pressed and a value of "0" when not pressed.



We will be using the remote control buttons to move the arm connected to the VEX robot. If you have not built the arm attachment, build it now using the instructions found in the Setup Section.

Once you have the robot arm attached, we will program two new behaviors. The new behaviors will be "Lift the Arm" and "Lower the Arm". To activate the arm, we will use the Up and Down buttons in Group 6, on the top of the remote control.

Engineering Lab

Programming Remote Control Buttons

```
1  #pragma config(Motor, port2, rightMotor, tmotorNormal, openLoop, reversed)
2  #pragma config(Motor, port3, leftMotor,  tmotorNormal, openLoop)
3  #pragma config(Motor, port6, armMotor,   tmotorNormal, openLoop)
4  //*!!Code automatically generated by 'ROBOTC' configuration wizard   !!*//
5
6  task main()
7  {
8    while(1 == 1)
9    {
10     motor[leftMotor] = vexRT[Ch3] / 2;
11     motor[rightMotor] = vexRT[Ch2] / 2;
12
13     if(vexRT[Btn6U] == 1)
14     {
15       motor[armMotor] = 40;
16     }
17     else if(vexRT[Btn6D] == 1)
18     {
19       motor[armMotor] = -40;
20     }
21     else
22     {
23       motor[armMotor] = 0;
24     }
25   }
26 }
```

The values of the remote control buttons are accessed using reserved words in the form:

Btn#X

"Btn" is short for button.

"#" is replaced by the button group number (5, 6, 7, or 8).
"X" is replaced by the direction of the button - U, D, R, or L (Up, Down, Right and Left).

Line 13 references remote control button 6U, and line 17 references to remote control button 6D.

What do lines 13 through 24 do? Experiment by commenting that section of code out and seeing what happens.

Copy the program above into ROBOTC and download it to your robot. Make sure that the motor for the arm is connected to Port 6. When you press buttons 6U and 6D, the arm will move up and down.

Note: If you have your robot connected to the computer over VEXnet, you're able to view all of the remote control values using the Remote Control Troubleshooter. To open the Troubleshooter, go to Robot > Remote Control Troubleshooter > VEXnet Cortex Controller.

Timers

Timers are very useful for performing a more **complex behavior for a certain period of time**. Wait states (from **`wait1Msec`**) don't let the robot execute commands during the waiting period, which is fine for simple behaviors like moving forward. If calculations or other actions need to occur **during the timed period**, as with the line tracking behavior below, a Timer must be used.

```
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.

Switch Case

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.

Below is the pseudocode outline of a switch-case Statement.

```
switch(switch-value)
{
  case 1st-value:
    // match-1st-commands
  break;

  case 2nd-value:
    // match-2nd-commands
  break;

  default:
    // default-commands
}
```

switch value
The value which be checked for a match with any cases.

case value
A possible match for the switch value. If this value matches the switch value, the code immediately following it runs.

case commands
The commands that run if this case successfully matched

break; command
Marks the end of each case's command statements.

default case
If the switch value does not match any of the given case values, the "default" case will run.

Switch Case

The touch sensors are used to set the value of turnVar in the program below. The switch-case statement is then used to determine what to do, based on its value. No sensors pressed will leave turnVar with a value of 0, and the robot will run the "default" case and go straight. Pressing touch1 will give turnVar a value of 1, and make case 1 run (left turn). Pressing touch2 makes turnVar 2, which makes case 2 (right turn) run. Both turns reset turnVar to 0 before ending, to allow fresh input on the next pass of the loop.

```
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;
    }
  }
}
```

Switch statement
The "switch" line designates the value that will be evaluated to see if it matches any of the case values.

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.

While Loop

A while loop is a structure within ROBOTC which allows a portion of code to be run over and over, as long as a certain condition remains true.

Below is the pseudocode outline of a while loop.

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

(condition)
Either **true** or **false** (see Reference > Boolean Logic).

Repeated commands
Commands placed here will run over and over as long as the (condition) is **true** when the program checks at the beginning of each pass through the loop.

Below is an example of a program using an infinite While Loop.

```
task main()
{
 bMotorReflected[port2]=1;
 while(1 == 1)
 {
   motor[port3]=127;
   motor[port2]=127;
 }
}
```

The condition is true as long as 1 is equal to 1, which is always.

While the condition is true, both motors will receive full power.

if-else Statement

An if-else Statement is one way you allow a computer to make a decision. With this command, the program will check the (condition) and then execute one of two pieces of code, depending on whether the (condition) is true or false.

Below is the pseudocode outline of an if-else Statment.

```
if(condition)
{
  // true-commands
}
else
{
  // false-commands
}
```

(condition)
Either **true** or **false** (see Reference > Boolean Logic).

(true) commands
Commands placed here will run if the (condition) is **true**.

(false) commands
Commands placed here will run if the (condition) is **false**.

Below is an example program containing an if-else Statement.

```
task main()
{
 bMotorReflected[port2]=1;
 while(true)
 {
   if(SensorValue(sonarSensor)>25)
   {
    motor[port3]=127;
    motor[port2]=127;
   }

   else
   {
    motor[port3]=0;
    motor[port2]=0;
   }
 }
}
```

(condition)
true if the sensor reads over 25
false otherwise

(true) commands
These commands run if the (condition) is **true**.

(false) commands
These commands run if the (condition) is **false**.

This if-else Statement tells the robot to run both motors at full power if the nearest object the Ultrasonic Range Finder detects is more than 25 inches away. If the Ultrasonic Range Finder detects an object closer than 25 inches, then the "else" portion of the code will be run and the robot will stop moving. The outer `while(true)` loop makes the if-else statement run over and over forever.

Boolean Logic

Truth Values

Robots don't like ambiguity when making decisions. They need to know, very clearly, which choice to make under what circumstances. As a consequence, their decisions are always based on the answers to questions which have only two possible answers: yes or no, true or false. Statements that can be only true or false are called **Boolean statements**, and their true-or-false value is called a **truth value**.

Fortunately, many kinds of questions can be phrased so that their answers are Boolean (true/false). Technically, they must be phrased as **statements**, not questions. So, rather than asking whether the sky is blue and getting an answer yes or no, you would state that "the sky is blue" and then find out the truth value of that statement, **true** (it is blue) or **false** (it is not blue).

Note that the truth value of a statement is only applicable **at the time it is checked**. The sky could be blue one minute and grey the next. But regardless of which it is, the statement "the sky is blue" is either true or false at **any specific time**. The truth value of a statement does not depend on **when** it is true or false, only **whether** it is true or false **right now**.

(Conditions)

ROBOTC **control structures** that make decisions about which pieces of code to run, such as **while loops** and **if-else** conditional statements, always depend on a (condition) to make their decisions. **ROBOTC (conditions) are always Boolean statements.** They are always either true or false at any given moment. Try asking yourself the same question the robot does – for example, whether the value of the Ultrasonic Sensor is greater than 45 or not. Pick any number you want for the Ultrasonic Sensor value. The statement "the Ultrasonic Sensor's value is greater than 45" will still either be true, or be false.

Condition	Ask yourself...	Truth value
`SensorValue(sonarSensor) > 45`	Is the value of the Ultrasonic Sensor greater than 45?	**True**, if the current value is more than 45 (for example, if it is 50). **False**, if the current value is not more than 45 (for example, if it is 40).

Some (conditions) have the additional benefit of ALWAYS being true, or ALWAYS being false. These are used to implement some special things like "infinite" loops that will never end (because the condition to make them end can never be reached!).

Condition	Ask yourself...	Truth value
`1==1`	Is 1 equal to 1?	**True**, always
`0==1`	Is 0 equal to 1?	**False**, always

Boolean Logic

Comparison Operators

Comparisons (such as the comparison of the Ultrasonic sensor's value against the number 45) are at the core of the decision-making process. A well-formed comparison typically uses one of a very specific set of operators, the "comparison operations" which generate a true or false result. Here are some of the most common ones recognized by ROBOTC.

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**

Evaluating Values

The "result" of a comparison is either true or false, but the robot takes it one step further. The program will actually substitute the true or false value in, where the comparison used to be. Once a comparison is made, it not only is true or false, it literally **becomes** true or false in the program.

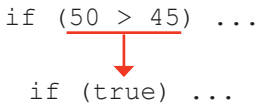
```
if (50 > 45) ...
```

```
if (true) ...
```

Boolean Logic

Use in Control Structures

"Under the hood" of all the major decision-making control structures is a simple check for the Boolean value of the (condition). The line `if (SensorValue(bumper) == 1)...` may read easily as "if the bumper switch is pressed, do...", but the robot is really looking for `if(true)` or `if(false)`. Whether the robot runs the "if true" part of the if-else structure or the "else" part, depends solely on whether the (condition) boils down to `true` or `false`.

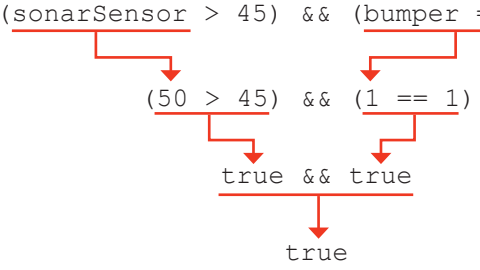


Logical Operators

Some (conditions) need to take **more than one thing** into account. Maybe you only want the robot to run if the traffic light is green AND there's no truck stopped in front of it waiting to turn. Unlike the comparison operators, which produce a truth value by comparing other types of values (is one number equal to another?), the **logical operators** are used to **combine multiple truth values into one single truth value**. The combined result can then be used as the (condition).

Example:
Suppose the value of a Light Sensor named **sonarSensor is 50**, and at the same time, the value of a Bumper Switch named **bumper is 1** (pressed).

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



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**

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:

1. Introduce (create or "**declare**") the variable
2. Give ("**assign**") the variable a value
3. Use the variable to access the stored value

```
task main()
{
  int speed;

  speed = 75;

  motor[port3] = speed;
  motor[port2] = speed;
  wait1Msec(2000);
}
```

Declaration
The variable is created by announcing 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. 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.

Here, both motor commands expect integers for power settings, so the int variable **speed** can stand in. The commands set their respective motor powers to the **value** stored in **speed**, 75.

In the example above, the variable "speed" is used to **store a number**, and then retrieve and **use that value** when it is called for later on. Specifically, it stores a number given by the programmer, and retrieves it twice in the two different places that it is used, once for each of the motor commands. This way both motors are set to the same value, but more interestingly, you would only need to **change one line of code to change both motor powers**.

```
task main()
{
  int speed;

  speed = 50;

  motor[port3] = speed;
  motor[port2] = speed;
  wait1Msec(2000);
}
```

One line changed
The value assigned to **speed** is now 50 instead of 75.

Changed without being changed
No change was made to the program here, but because these lines use the value contained in the variable, both lines now tell their motors to run at a power level of 50 instead of 75.

This example shows **just one way** in which variables can be used, as a convenience for the programmer. With a robot, however, the ability to store sensor values (values that are **measured by the robot, rather than set by the programmer**) adds invaluable new capabilities. It gives the robot the ability to take measurements in one place and deliver them in another, or even do its own calculations using stored values. The same basic rules are followed, but the possibilities go far beyond just what you've seen so far!

Variables

Declaration Rules
In order to declare a variable, you must declare its type, followed by its name. Here are some specifics about the rules governing each:

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

The following is a list of data types most commonly used in ROBOTC:

Data Type	Description	Example	Code
Integer	Positive and negative whole numbers, as well as zero.	-35, -1, 0, 33, 100, 345	`int`
Floating Point Decimal	Numeric values with decimal points (even if the decimal part is zero).	-.123, 0.56, 3.0, 1000.07	`float`
Boolean	True or False. Useful for expressing the outcomes of comparisons.	true, false	`bool`

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
linecounter	line counter
threshold	threshold!
distance3	3distance
timecounter	time1[T1]

Variables

Assignment and Usage Rules

Assignment of values to variables is pretty straightforward, as is the use of a variable in a command where you wish its value to be used.

Rules for Assignment

- Values are assigned using the **assignment operator** = (not ==)
- Assigning a value to a variable that already has a value in it will **overwrite** the old value with the new one
- **Math operators** (+, -, *, /) can be used with assignment statements to perform calculations on the values before storing them
- A variable can appear in **both the left and right hand sides** of an assignment statement; this simply means that its **current value** will be used in calculating the new value
- Assignment can be done in the same line that a variable is **declared** (e.g. `int x = 0;` will both create the variable x and put an initial value of 0 in it)

Rules for Variable Usage

- "Use" a variable simply by putting its name where you want its value to be used
- The current value of the variable will be used every time the variable appears

Examples:

Statement	Description
`motorPower = 75;`	Stores the value 75 in the variable "motorPower"
`sonarVariable = SensorValue(sonarSensor);`	Stores the current sensor reading of the sensor "sonarSensor" in the variable "sonarVariable"
`sum = variable1 + variable2;`	Adds the value in "variable1" to the value in "variable2", and stores the result in the variable "sum"
`average = (variable1 + variable2)/2;`	Adds the value in "variable1" and the value in "variable2", then divides the result by 2, and stores the final resulting value in "average"
`count = count + 1;`	Adds 1 to the current value of "count" and places the result back into "count" (effectively, increases the value in "count" by 1)
`int zero = 0;`	Creates the variable x with an initial value of 0 (combination declaration and assignment statement)

Global Variables

When you create a variable, it can only be used inside the function or task where it was declared. This can be a problem when you need to use the same variable in several different places – for example, a function you made, **and** task main.

Scope
Variables exist only within certain boundaries, for example, only within the functions where they are declared. Scope is a definition of these boundaries: **how broadly applicable (or "visible") a value or variable is**.

Scope exists to prevent conflicts between functions with similarly-named variables, and (more importantly) to keep variables from different functions from accidentally interfering with each other, or even with themselves if the same function is run more than once.

In general, the rule is that a variable can only be used **within the task or function where it is declared**, including task main. If you try to use a variable in location outside its scope, ROBOTC will give you an error that no such variable exists, because the program at that point cannot "see" it.

Global Variables
One very rough way to get around the limitations of scope in a program is to declare a variable as **global**. A global variable is declared **outside** any functions or tasks, and therefore typically appears at the very top of a program.

Because they are declared at a level broader than any task or function, **all** functions and tasks can "see" global variables, and they do not lose their value even after a function or task ends. This is both a strength and a liability.

1. Declare global variable
The global variable is declared outside any tasks or functions, allowing them all to see it.

2. Variable is "visible" inside functions
The global variable can be used and changed inside functions.

3. Variable is "visible" inside tasks
The global variable can be used and changed inside tasks such as task main.

A Hint of Trouble
How long will this program wait at the end, 2000 or 6000ms? Because both the function and the task main were able to modify the value of this variable, it is more difficult to tell what its value is by the time it is used. As your program gets more complex, excessive use of global variables may lead to more and more confusion.

```
int timeValue;

void changeValue()
{
  timeValue = 6000;
}

task main()
{
  timeValue = 2000;
  changeValue();
  wait1Msec(timeValue);
}
```

Optimizing Code

Challenge Description

Use programming structures such as functions and loops to minimize the number of lines of code necessary to perform the same functionality as the Sample Code below. Download and run your program and the Sample program to verify that they are the same.

Sample Code

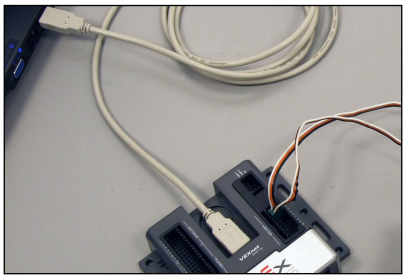
```
1  task main()
2  {
3      bMotorReflected[port2]= 1;
4
5      motor[port3] = 63;
6      motor[port2] = 63;
7      wait1Msec(2000);
8
9      motor[port3] = -63;
10     motor[port2] = 63;
11     wait1Msec(350);
12
13     motor[port3] = 63;
14     motor[port2] = 63;
15     wait1Msec(2000);
16
17     motor[port3] = -63;
18     motor[port2] = 63;
19     wait1Msec(350);
20
21     motor[port3] = 63;
22     motor[port2] = 63;
23     wait1Msec(2000);
24
25     motor[port3] = -63;
26     motor[port2] = 63;
27     wait1Msec(350);
28
29     motor[port3] = 63;
30     motor[port2] = 63;
31     wait1Msec(2000);
32
33     motor[port3] = -63;
34     motor[port2] = 63;
35     wait1Msec(350);
36
37     motor[port3] = 63;
38     motor[port2] = 63;
39     wait1Msec(2000);
40
41     motor[port3] = -63;
42     motor[port2] = 63;
43     wait1Msec(350);
44
45     motor[port3] = 63;
46     motor[port2] = 63;
47     wait1Msec(2000);
48
49     motor[port3] = -63;
50     motor[port2] = 63;
51     wait1Msec(350);
52
53     motor[port3] = 63;
54     motor[port2] = 63;
55     wait1Msec(2000);
56
57     motor[port3] = -63;
58     motor[port2] = 63;
59     wait1Msec(350);
60
61     motor[port3] = 63;
62     motor[port2] = 63;
63     wait1Msec(2000);
64
65     motor[port3] = -63;
66     motor[port2] = 63;
67     wait1Msec(350);
68 }
69
```

Running a Program

Once a program has been successfully written, it needs to be given to the robot to run.
The following steps will guide you through the process of downloading your program to the robot, and then running it remotely or connected to your computer.

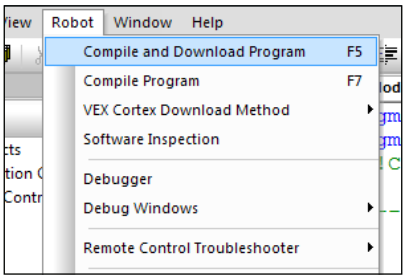
1. Verify that your VEX Cortex is **turned off**. Make sure it's not powered by the battery or PC (through the USB cable).

2. If you are programming over USB, connect the Cortex to the PC using the USB A-to-A cable. This will partially power the Cortex, then **turn the Cortex on** to fully power it using the battery.



If you are programming over VEXnet, turn both the Cortex and VEXnet Remote Control ON and allow them to sync. Make sure the Programming cable is connecting your VEXnet Remote Control to the PC.

3. Click "**Robot**" on the top menu bar of the ROBOTC window, and select "**Compile and Download Program**".



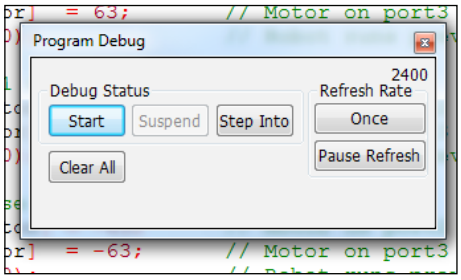
4. You may be prompted to save your program. If so, save it in the same directory as your other programs.

5. If there are **errors** in your code the compiler will identify them for you and you will need to correct them before a successful download can be completed.

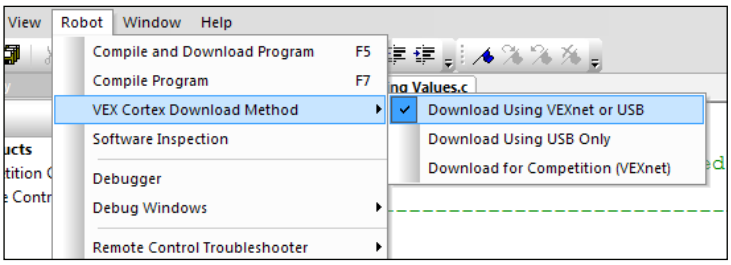
Running a Program

Downloading places your program on the robot to be run on command. You can run the program in two different ways.

- **Run Attached**
  If your robot is connected to your computer you can run the program which was just downloaded by clicking "**Start**" in the "Program Debug" window which automatically appears upon download. This will run your program and because the robot is still connected to the computer, you can obtain live variable and sensor feedback by using such debug windows as "Global Variables" and "Sensors".



- **Run Independently**
  If you want to run the program while the robot is not connected to the computer, just remove the cable once the program has been downloaded. On the back of the Cortex, toggle the power switch in the "Off" position, and then back to "On". The program will run immediately if your "VEX Cortex Download Method" is set to "Download using USB Only". The robot will wait for 10 seconds or until it finds a VEXnet connection if the "VEX Cortex Download Method" is set to "Download using VEXnet or USB".



Design Specifications - Labyrinth

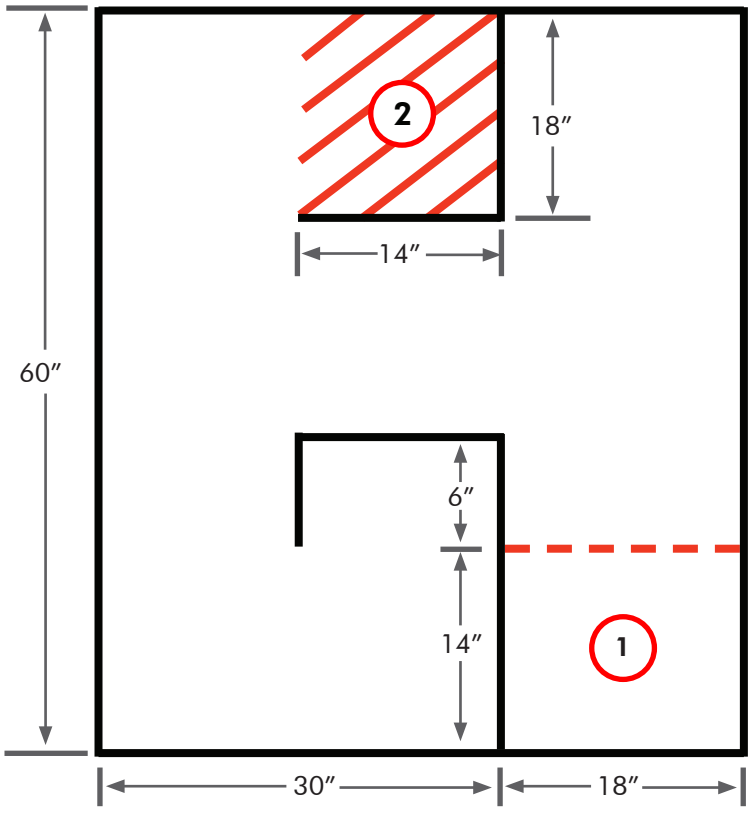
Challenge Description:

This challenge features a sequence of turns that the robot must perform in order to get to the "end" of the Labyrinth. The robot must first begin at the starting point, and get to the goal area by completing turning and forward movement behaviors.

Materials Needed:

- Black electrical tape
- Red electrical tape
- Scissors (or cutting tool)
- Ruler (or straight edge)

Board specifications:



1 Robot must begin here, and then maneuver the robot to get to the goal area.

2 Robot must reach this goal area without crossing any black lines (Goal area lines).

Notas de programación 1

1. Define qué es un comportamiento (Behaviors) y los tipos de comportamientos de programación que existen.

2. ¿Qué es un código?

3. Es lo mismo Task main a task main… ¿por qué?

4. ¿Qué significa cuando aparece una palabra cuando estas escribiendo en RobotC?

5. Escribe un ejemplo de una sentencia simple:

6. Cuando se ejecuta el programa en qué orden Robot C lee las instrucciones.

7. ¿Cómo reconoce RobotC donde comienza o finaliza una sentencia?

8. ¿Con qué propósitos se agregan espacios, tabulaciones y líneas en blanco?

Notas de programación 1

9. Escribe la sintaxis de las palabras reservadas que utilizaremos con mayor y su uso:

Palabras reservadas	Sintaxis	Función ¿para qué sirve?
Motors		
Timing		
Sensors		
Radio Control		
Control Structures		
Data Types		

Investigación de movimiento

1. Escribe los pasos para programar un robot VEX:

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. Si se utiliza joystick: _____

2. Escribe el código necesario para que un robot con 2 motores en el chasis pueda avanzar en un tiempo determinado:

Considerar que en el pragma se han declarado los motores con los siguientes nombres: ***RightMotor*** *(el motor derecho) y* ***LeftMotor*** *(el motor izquierdo).*

```
task main ( ) {




}
```

3. Programa tu robot para que avance a una potencia y en un tiempo determinado, midiendo la distancia para calcular su velocidad.
Completa la siguiente tabla:

Código	Distancia recorrida	Tiempo	Velocidad (m/s)

motor[motor_der]=127;
motor[motor_izq]=127;
wait1Msec(1000);
		1 seg	

motor[motor_der]=63;
motor[motor_izq]=63;
wait1Msec(1000);
		1 seg	

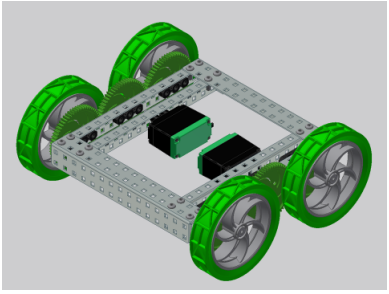
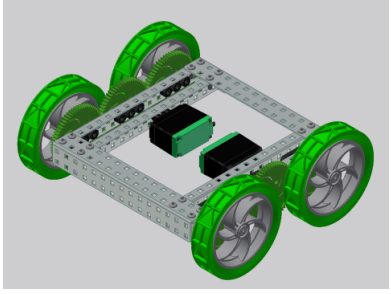
motor[motor_der]=32;
motor[motor_izq]=32;
wait1Msec(1000);
		1 seg	

4. ¿Es lo mismo velocidad y potencia? ¿Por qué?

5. Elimina el código donde se colocan los motores en cero y escribe ¿qué sucede?, ¿hay alguna diferencia?

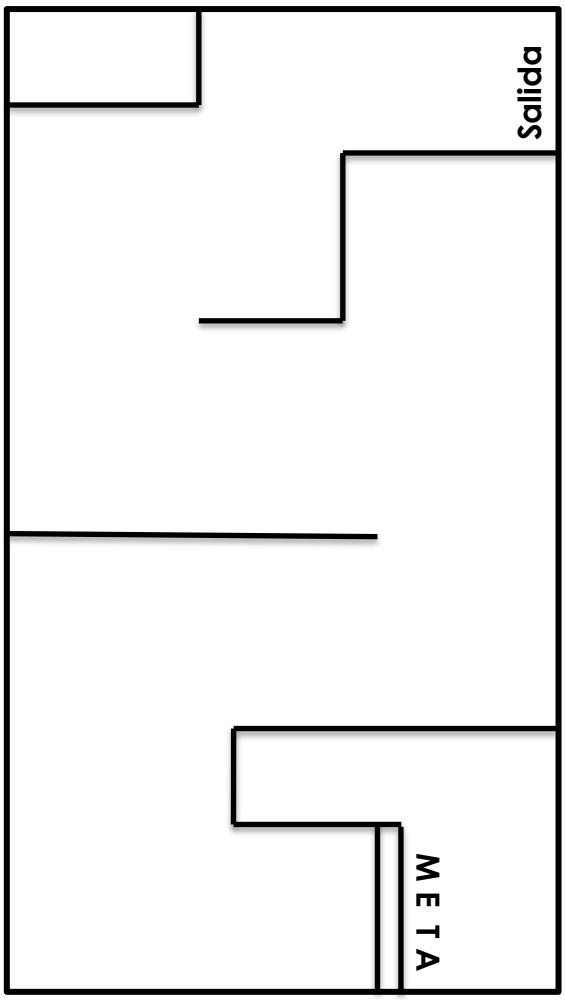
6. Describe qué sucede cuando se programan los siguientes códigos y dibuja sobre la imagen del robot unas flechas para indicar el movimiento de las llantas del robot.

motor[motor_der]= - 63;
motor[motor_izq]= 63;
wait1Msec(1000);
motor[motor_der]=0;
motor[motor_izq]=0;
	motor[motor_der]= 0;
	motor[motor_izq]= 63;
	wait1Msec(1000);
	motor[motor_der]=0;
	motor[motor_izq]=0;

POINT TURN  **¿Qué sucede?**	**SWING TURN**  **¿Qué sucede?**

LABERINTO

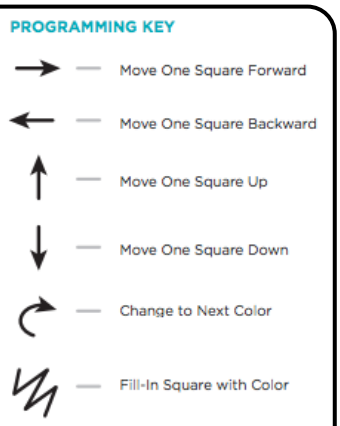
7. Resuelve el laberinto escribiendo el código necesario para llegar a la meta y programa tu robot para que realice el recorrido sin tocar las líneas.

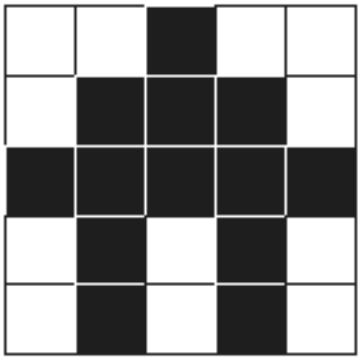


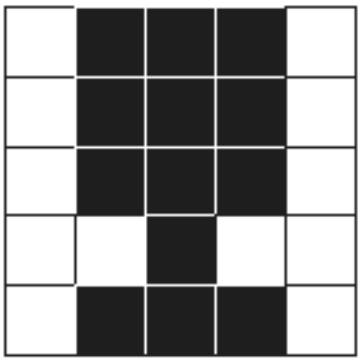
Paper Programming

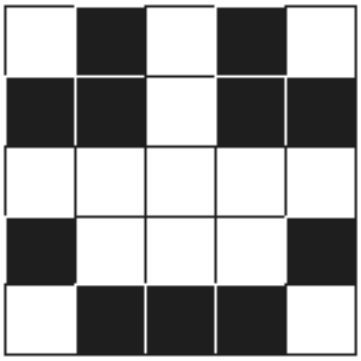
Nombre:
Grupo:

Escribe el código necesario para poder pintar las siguientes figuras, es importante tener orden por lo que utiliza diferentes líneas para cada una de las filas.





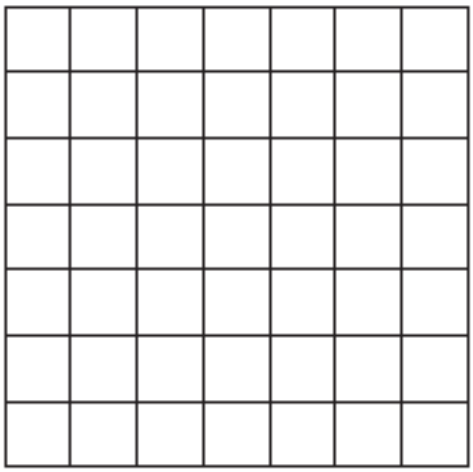




Paper Programming

Nombre:
Grupo:

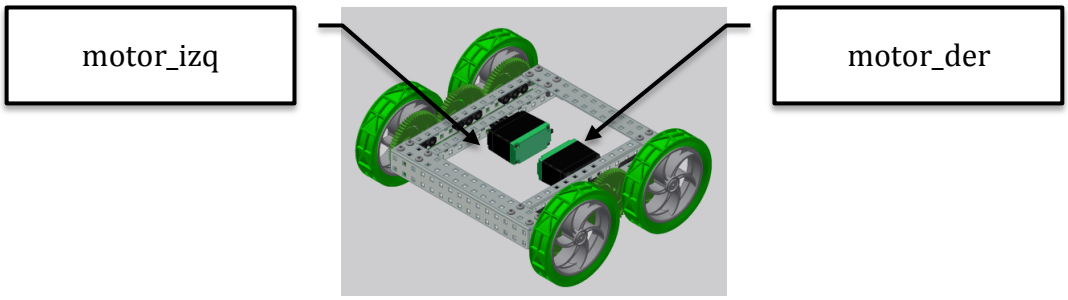
Inventa y dibuja una figura.
Escribe el código correspondiente a tu figura, hazlo con orden para que pueda decifrarse.
Importante: tu dibujo debe ser una figura real.



Código:

Repasando la programación del robot

Utilizando un robot como el siguiente modelo, programa las siguientes acciones:



1) Escribe el código necesario para que el robot avance durante 3 segundos a toda velocidad:

```
task main (){




}
```

2) Escribe el programa para que el robot retroceda durante 5 segundos a velocidad media:

```
task main (){




}
```

3) Escribe el programa para que el robot gire a la izquierda (point turn) sobre su mismo eje por 3 segundos a un cuarto de la velocidad:

```
task main (){




}
```

4) Escribe el programa para que el robot gire a la derecha (swing turn) avanzando durante 3 segundos a un velocidad media:

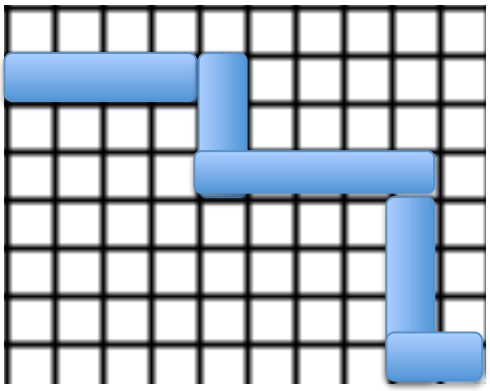
```
task main (){




}
```

RESUELVE EL RETO 1:

Consideraciones:
Si el robot avanza 10 cm en 1 segundo a una velocidad media y gira 90º a un cuarto de velocidad durante 3 segundos.



Para poder cruzar el siguiente laberinto considerar que la medida de cada cuadro es de 10 cm.
Escribe el código y utiliza comentarios para describir el código (si usas swing turn o point turn), hacia donde giras, si avanzas, etc.

UTILIZA **PROCEDIMIENTOS** PARA ACORTAR TU CÓDIGO.

CÓDIGO:

```
task main ( ) {
```

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES