

Robotics Reference Guide

Version 2.0

This guide undergoes continuous revision, including the addition of more reference guides. Be sure to visit the engineering section of the Virtual Academy to ensure that you have the most recent version. To determine whether you have the most up-to-date version, reference the date in the filename.



ROBOTC®

The logo for Carnegie Mellon Robotics Academy features a blue robot head icon next to the text "Carnegie Mellon Robotics Academy".

Carnegie Mellon
Robotics Academy

The logo for VEX Robotics Design System features the word "VEX" in large red letters with a grey "V" and "X" on either side, and "ROBOTICS DESIGN SYSTEM" in smaller red text below it.

VEX
ROBOTICS DESIGN SYSTEM

Robotics Reference Guide



Reference Links

Each Reference below is a link to that file.

Building with VEX

- [Introduction to the Structure Subsystem - VEX Inventor's Guide](#)
- [Robust Fabrication - VEX Inventor's Guide](#)
- [Introduction to the Motion Subsystem - VEX Inventor's Guide](#)
- [Cortex Pin Guide - ROBOTC Reference](#)

Getting Started

- [VEX Cortex Configuration Over USB - ROBOTC Reference](#)
- [Using the PLTW Template - ROBOTC Reference](#)
- [Sample Programs - ROBOTC Reference](#)
- [Running a Program - ROBOTC Reference](#)
- [VEXnet Joystick Configuration - ROBOTC Reference](#)

Programming

- [Sense, Plan, Act - ROBOTC Reference](#)
- [Behaviors - ROBOTC Reference](#)
- [Pseudocode & Flow Charts - ROBOTC Reference](#)
- [Program Design - ROBOTC Reference](#)
- [ROBOTC Natural Language Cortex Quick Reference - ROBOTC Reference](#)
- [The ROBOTC Debugger - ROBOTC Reference](#)
- [White Space - ROBOTC Reference](#)
- [Comments - ROBOTC Reference](#)
- [Boolean Logic - ROBOTC Reference](#)
- [Variables - ROBOTC Reference](#)
- [Reserved Words - ROBOTC Reference](#)
- [While Loops - ROBOTC Reference](#)
- [If Statements - ROBOTC Reference](#)
- [Variables - ROBOTC Reference](#)
- [Thresholds - ROBOTC Reference](#)
- [Timers - ROBOTC Reference](#)
- [Functions - ROBOTC Reference](#)
- [Switch Cases - ROBOTC Reference](#)
- [Random Numbers - ROBOTC Reference](#)

Troubleshooting

- [Error Messages in ROBOTC Code- ROBOTC Reference](#)
- [Troubleshooting ROBOTC with Cortex - ROBOTC Reference](#)

Robotics Reference Guide



Reference Links

Each Reference below is a link to that file.

Motor Outputs

- [2-Wire Motor 269 - VEX Inventor's Guide](#)
- [2-Wire Motor 393 - VEX Inventor's Guide](#)
- [Servo Modules - ROBOTC Reference](#)
- [Servo Motor - VEX Inventor's Guide](#)
- [Flashlight - VEX Inventor's Guide](#)
- [Color Camera - VEX Inventor's Guide](#)

Digital Inputs / Outputs

- [Ultrasonic Sensor - VEX Inventor's Guide](#)
- [Bumper Switch - VEX Inventor's Guide](#)
- [Limit Switch - VEX Inventor's Guide](#)
- [Optical Shaft Encoder - ROBOTC Reference](#)
- [Optical Shaft Encoder - VEX Inventor's Guide](#)

Analog Inputs

- [Potentiometers - ROBOTC Reference](#)
- [Potentiometers - VEX Inventor's Guide](#)
- [Line Following - VEX Inventor's Guide](#)
- [Light Sensor - VEX Inventor's Guide](#)

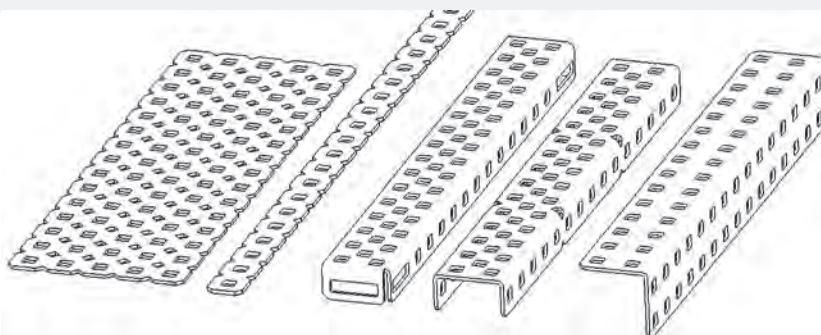
[ROBOTC Reference Glossary](#)

[VEX Inventors Guide Glossary](#)

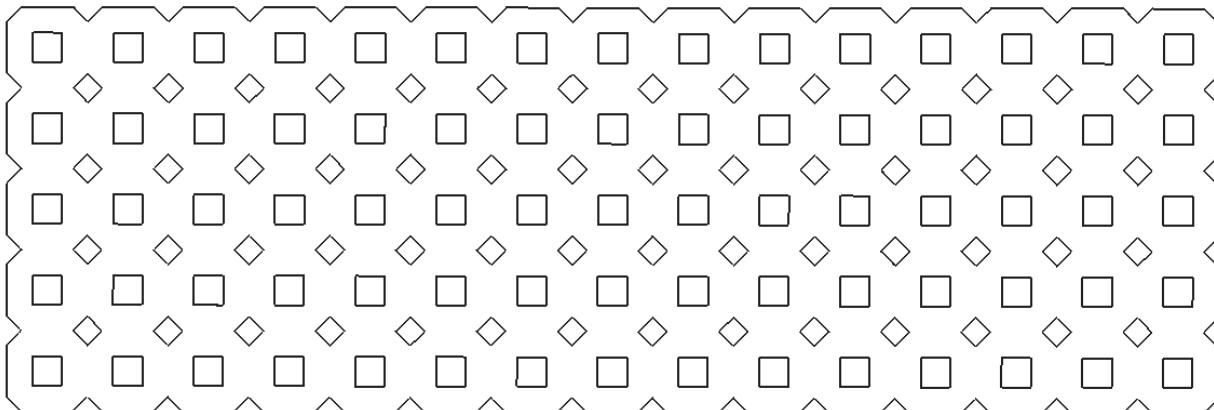
Introduction to the Structure Subsystem

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.

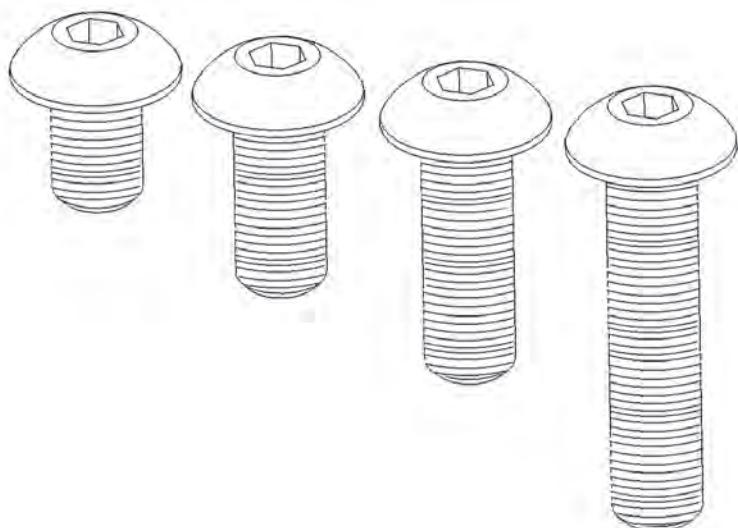
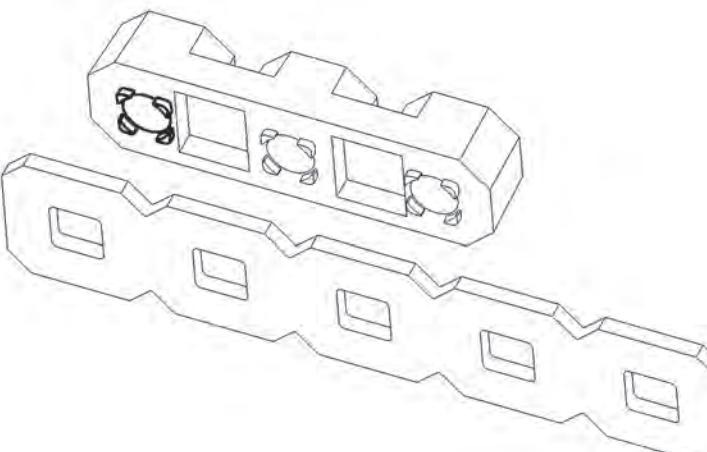


The VEX structural pieces all contain square holes (0.182" sq) on a standardized 1/2" grid. This standardized hole-spacing allows for VEX parts to be connected in almost any configuration. The smaller diamond holes are there to help users cut pieces using tin-snips or fine-toothed hacksaws without leaving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

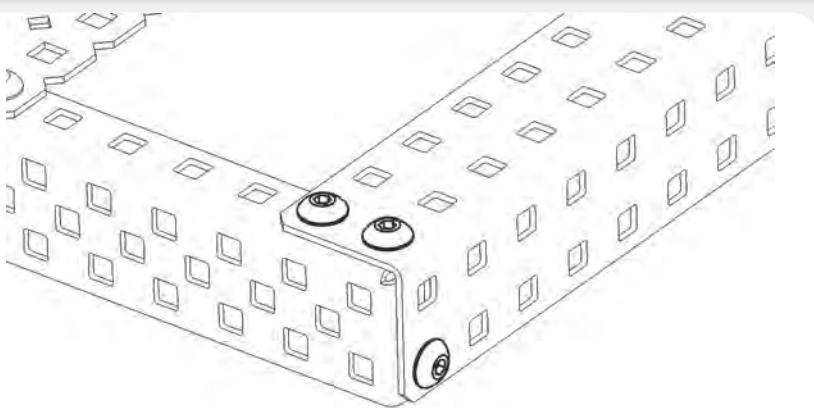
Note: There are two types of screws that are part of the VEX Robotics Design System.

- 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

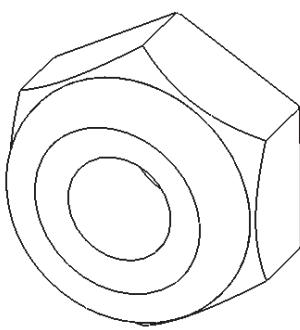
HINT:

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

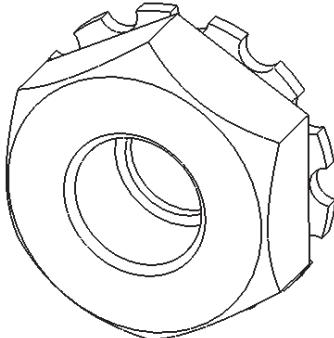


When using screws to attach things together, there are three types of nuts which can be used.

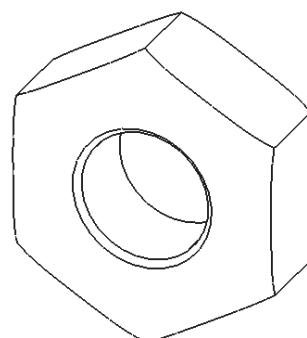
- Nylock nuts have a plastic insert in them which will prevent them from unscrewing. These are harder to install, as you need to use an open-ended wrench to tighten them up. These nuts will not come off due to vibration or movement.
- KEPS nuts have a ring of “teeth” on one side of them. These teeth 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



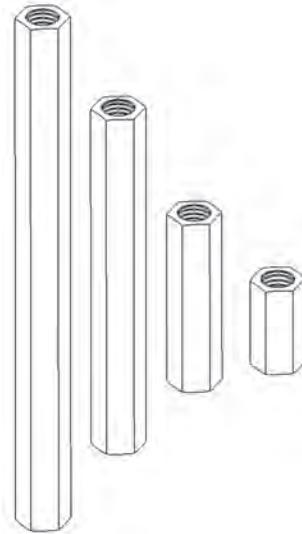
Regular (Hex) Nut

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!

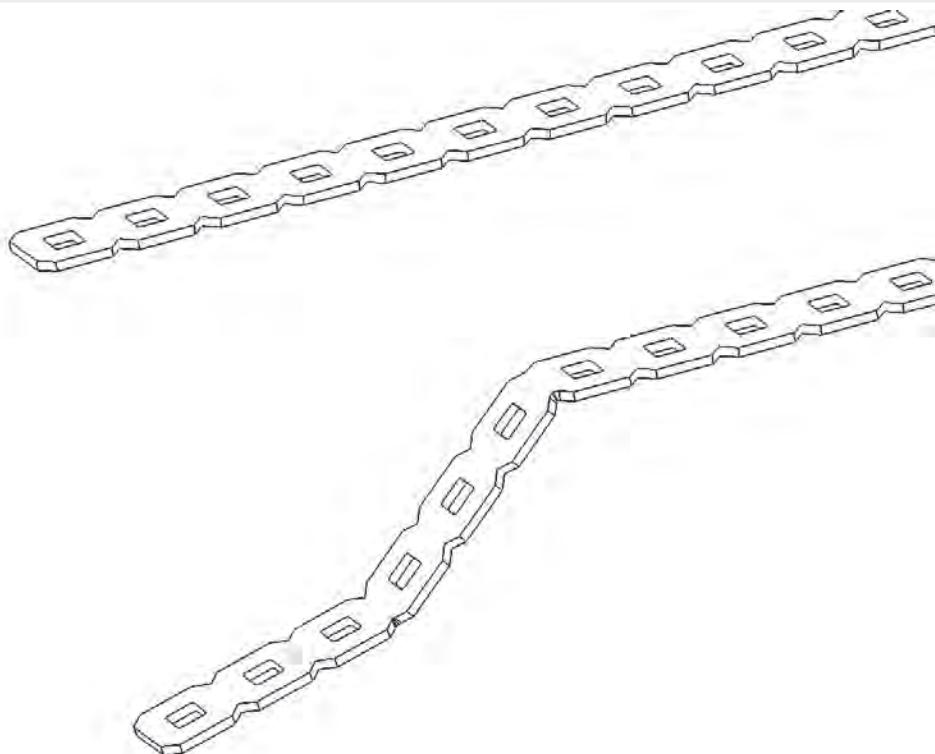
Introduction to the Structure Subsystem, continued

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.



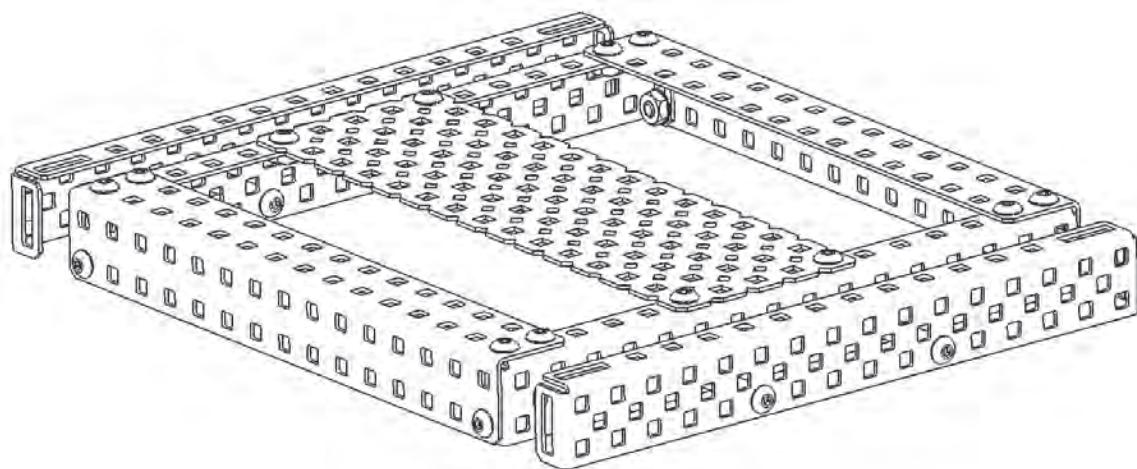
One of the key features of many VEX structural parts is their “bend-able” and “cut-able” nature. Users can easily modify many of these structural parts into new configurations better suited for their current needs. Flat plates can be bent into brackets. Many metal components can be cut to custom lengths.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

Robust Fabrication

Fasteners

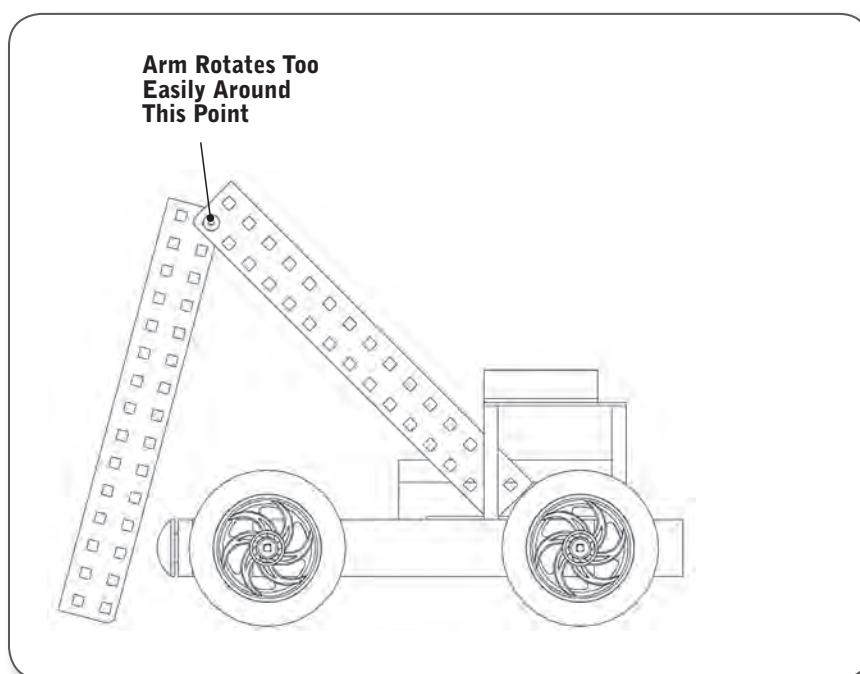
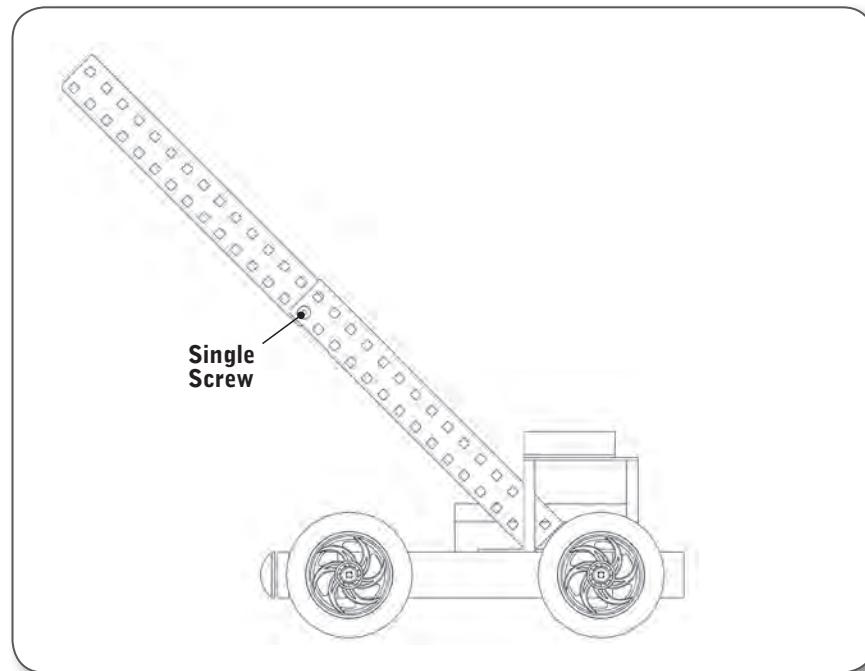
The most common problem with robots that fall apart or lose pieces easily is that groups of parts are not joined securely enough and separate from each other and move around.

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

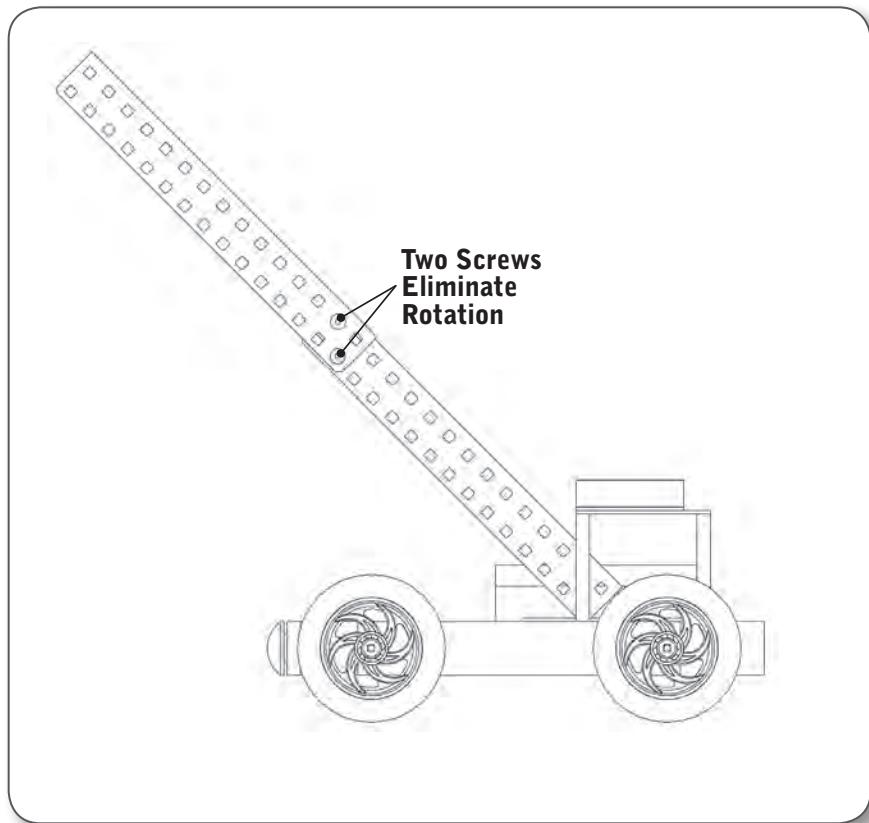
EXAMPLE 1,

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.



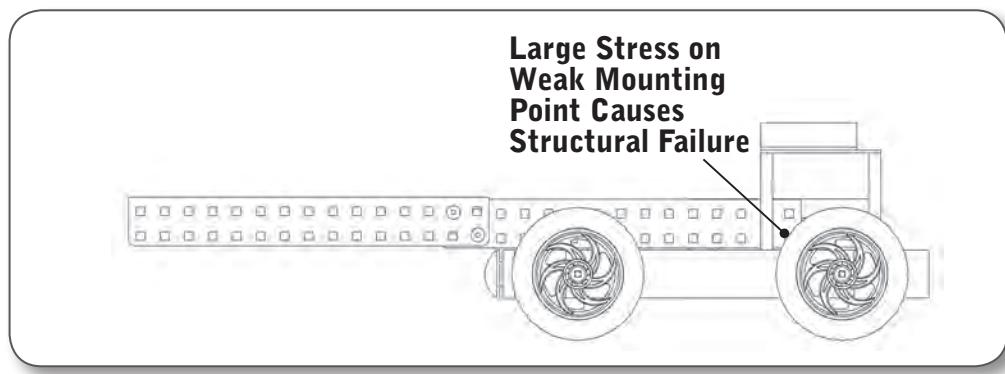
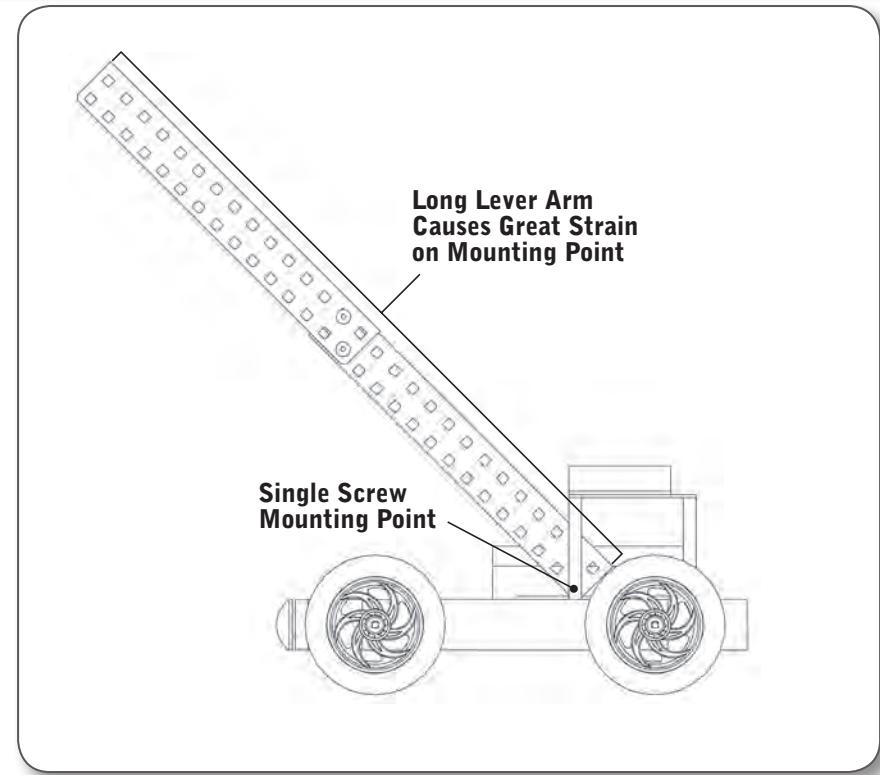
Concepts to Understand, continued

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.



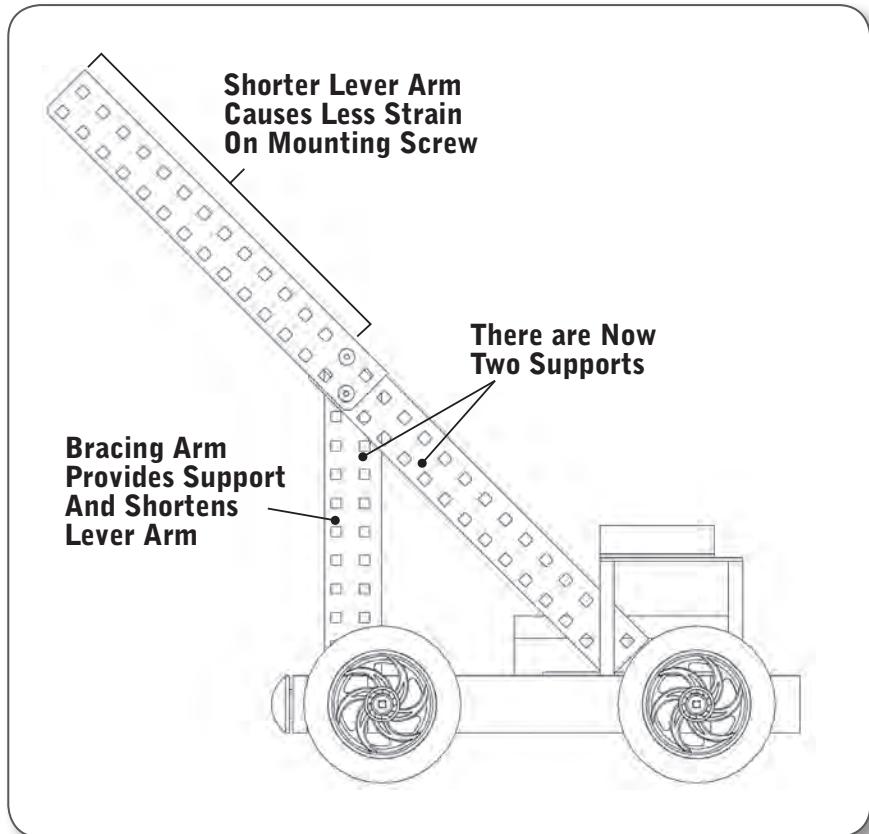
Concepts to Understand, continued

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.



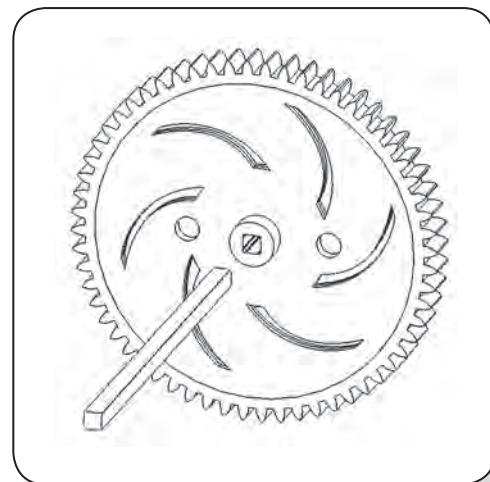
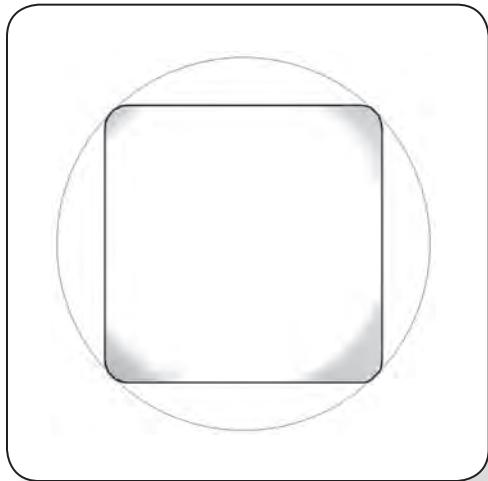
The arm is now more stable and better able to withstand stresses placed on it from both its own weight, and any external forces acting on it. The bracing arm has both decreased the mechanical advantage from the long lever arm, and spread the load over two supports instead of just one.

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.

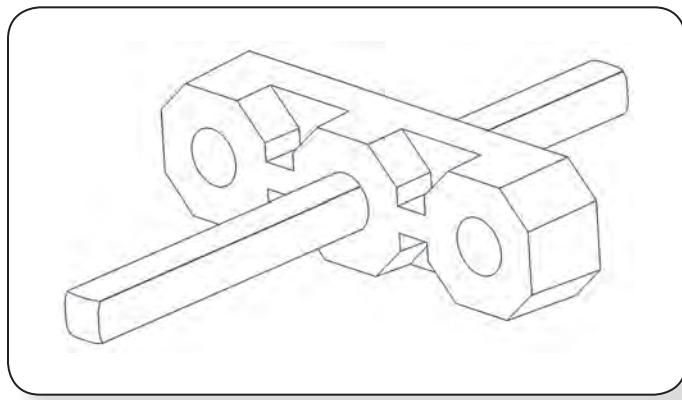
In the VEX Robotics Design System the motion components are all easily integrated together. This makes it simple to create very complex systems using the basic motion building blocks.

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.

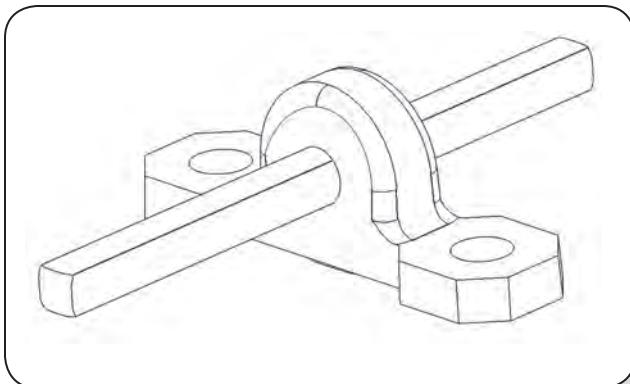


The square shaft has rounded corners which allow it to spin easily in a round hole. This allows the use of simple bearings made from Delrin (a slippery plastic). The Delrin bearing will provide a low-friction piece for the shafts to turn in.

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.

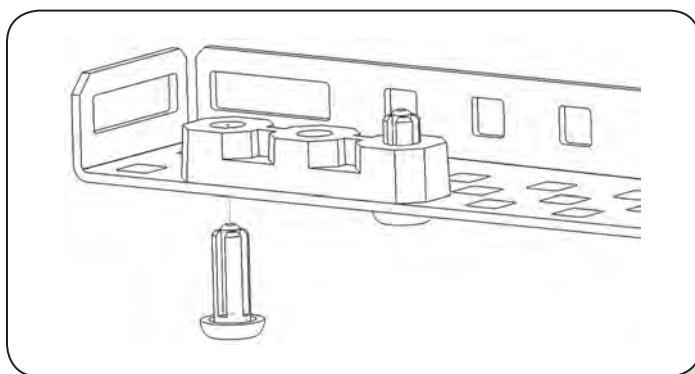


Introduction to the Motion Subsystem, continued



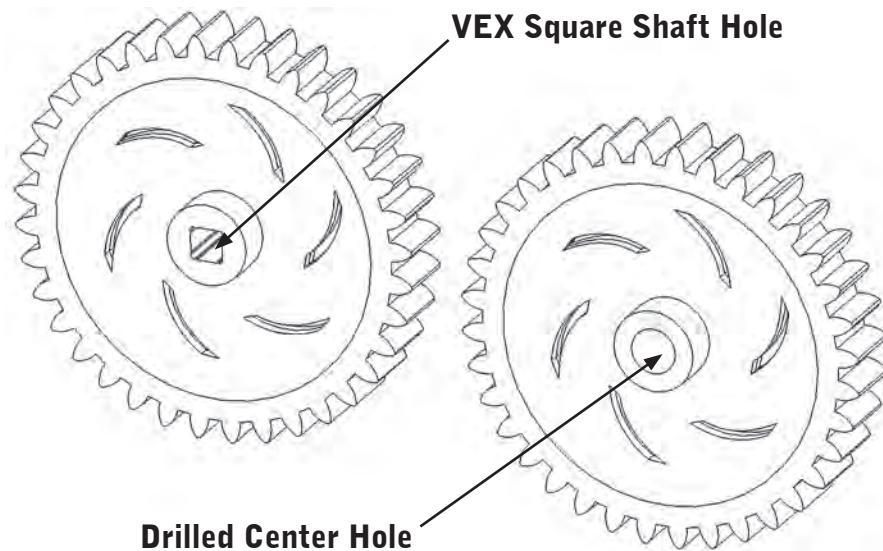
Another type of bearing used in the VEX Motion Subsystem is a Bearing Block; these are similar to the “pillow-blocks” used in industry. The Bearing Block mounts on a piece of structure and supports a shaft which is offset either above, below, or to the side of the structure.

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.

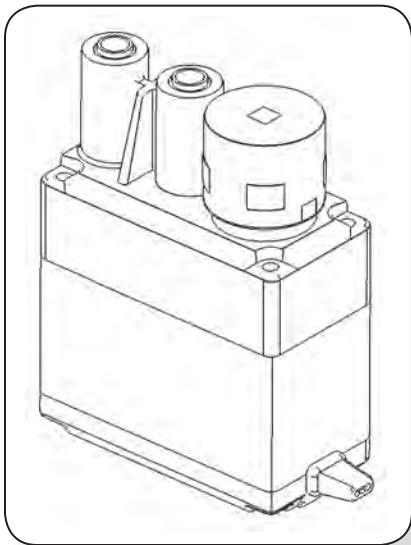


HINT:

It is also possible to 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



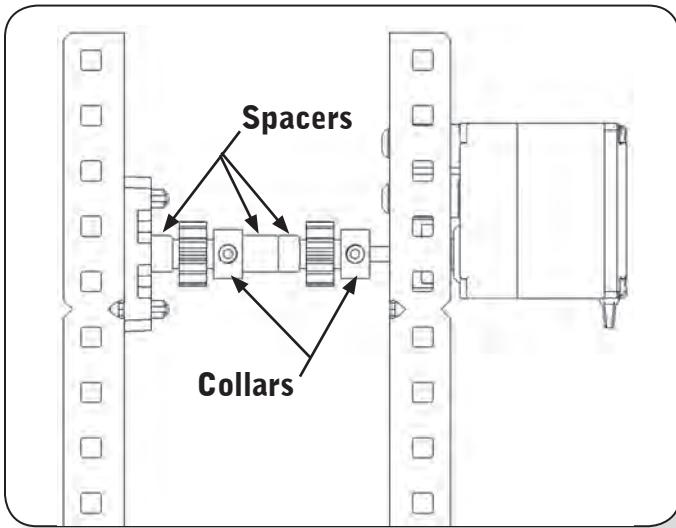
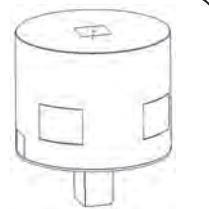
The key component of any motion system is an actuator (an actuator is something which causes a mechanical system to move). In the VEX Robotics Design System, there are several different actuator options. The most common types of actuators used are the VEX Continuous Rotation Motors (the 3-Wire Motor and the "high strength" 2-Wire Motor 393) 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.

The most common types of actuators used are the VEX Continuous Rotation Motors (the 3-Wire Motor, the 2-Wire Motor 269, and the "high strength" 2-Wire Motor 393) and the VEX Servos.

WARNING:

Some 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. 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. 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.

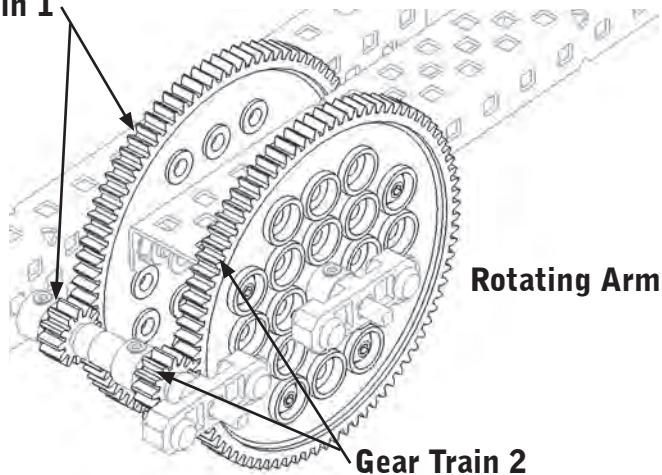
If a setscrew is lost any other VEX 8-32 screw can be substituted although the additional height of the screw head must be considered!

Introduction to the Motion Subsystem, continued

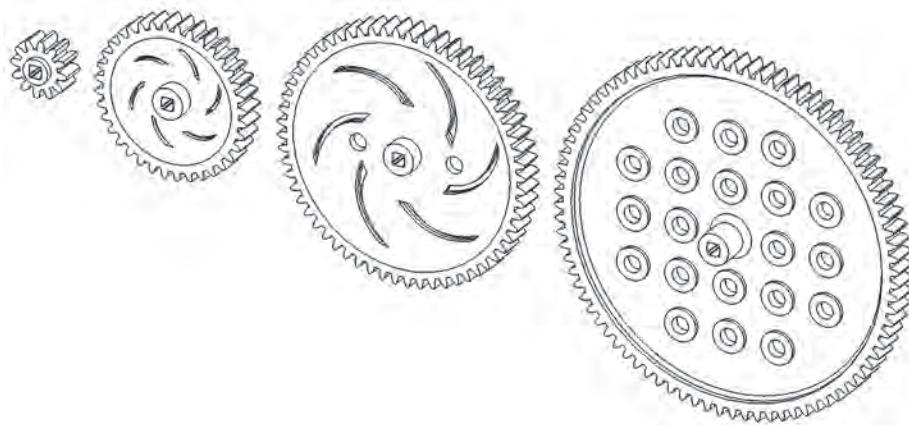
In some applications excessive loads can damage the components of the VEX Motion Subsystem. In these cases there are often ways to reinforce the system to reduce the load each individual component will experience, or so that the load is no longer concentrated at a single location on any given component.

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.

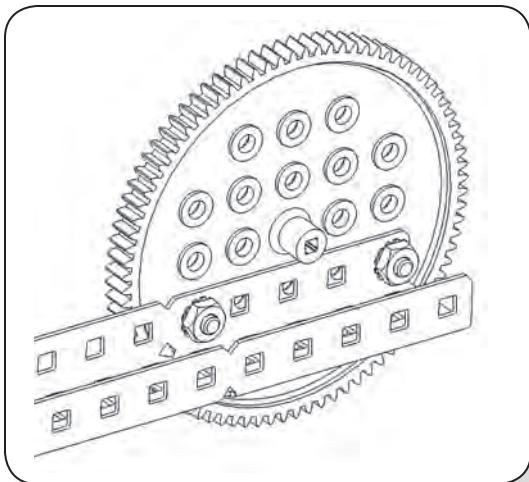


There are several ways to transfer motion in the VEX Robotics Design System. A number of Motion Subsystem accessory kits are available with a variety of advanced options. The primary way to transfer motion is through the use of spur gears. Spur gears transfer motion between parallel shafts, and can also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



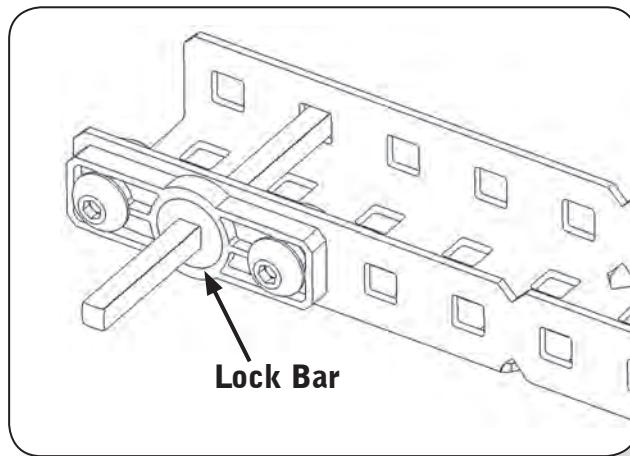
These gears can also be combined with sprocket & chain reductions, and also with advanced gear types to create even more complex mechanisms.

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.

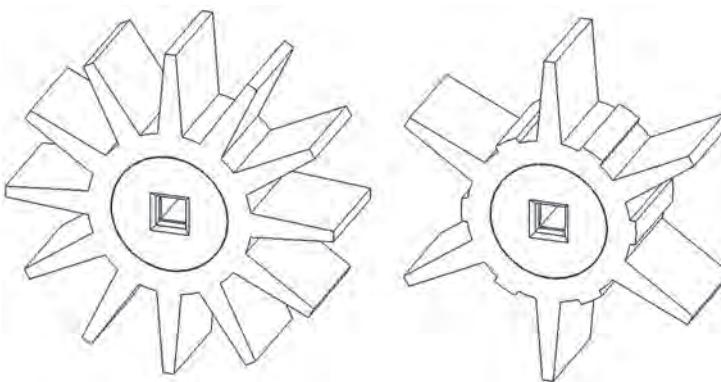
Another option to drive structural pieces using the Motion Subsystem is through a Lock Bar. These pieces are designed such that they can bolt onto any VEX structural component using the standard VEX 1/2" pitch. In the center of each piece there is a square hole which matches the VEX square shaft. As 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.

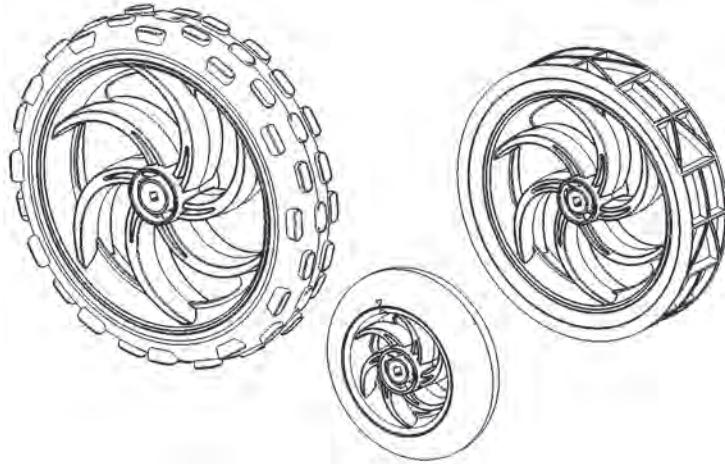
HINT:

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.

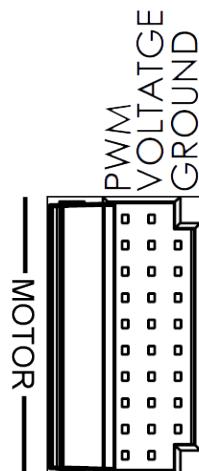
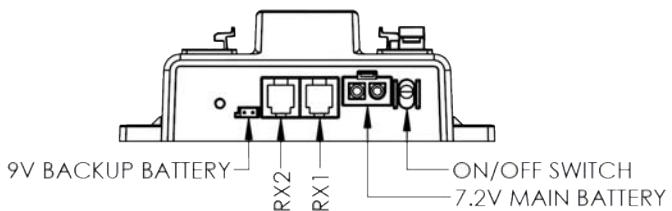
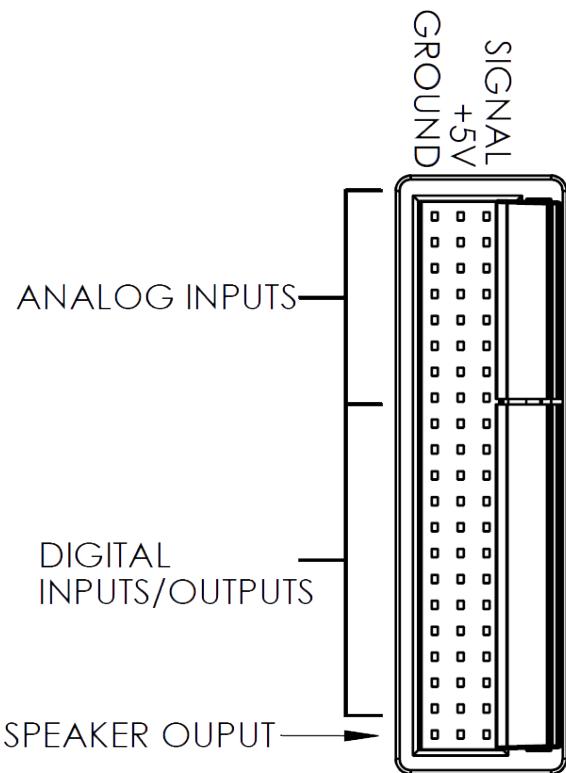
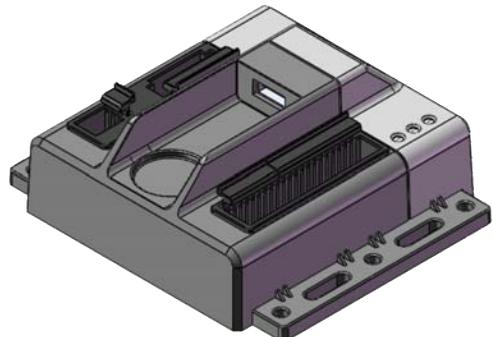
When designing the Motion Subsystem of a robot it is important to think about several factors:

- 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.

Cortex Pin Guide

The VEX Cortex Microcontroller coordinates the flow of all information and power on the robot. It has built in bidirectional communication for wireless driving, debugging and downloading using the state of the art VEXnet 802.11 wireless link. The Microcontroller is the brain of every VEX robot.



Analog Outputs & Digital Inputs / Outputs:

- Analog Outputs are used by any sensors that provide a range of values. Examples include: potentiometers, light sensors, line followers, and accelerometers.
- Digital ports are available for digital input signals. Examples include: bumper switches, limit switches, ultrasonic range finders, and optical shaft encoders.
- The digital ports can also be used as digital

Motor Outputs:

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

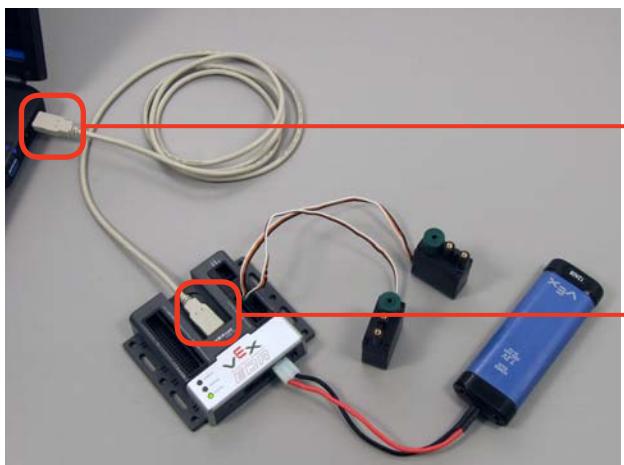
VEX Cortex Configuration over USB

The VEX Cortex is a fully programmable device, and is what enables you to incorporate motors, sensors, an LCD screen, and remote control signals all in one robot. Inside of the Cortex, there are two separate processors; a user processor handles all of the ROBOTC programming instructions, and a master processor controls lower-level operations, like motor control and VEXnet communication. This document is a guide for downloading the Master CPU firmware and ROBOTC firmware to the VEX Cortex using the USB A-to-A cable.

You will need:

- 1 VEX Cortex Microcontroller with one 7.2V Robot Battery
- A computer with ROBOTC for Cortex and PIC installed
- 1 USB A-to-A Cable

1. Leaving the POWER switch in the OFF position, connect your Cortex to the computer using the USB A-to-A cable. Once the cable is attached, move the POWER switch to the ON position.



- 1a. Connect the Cortex to your PC
Use the USB A-to-A cable to connect your Cortex to your PC.

Note: The order detailed in this step is crucial. When the Cortex is powered on, it immediately tries to determine how it is connected (over VEXnet, USB, or no connection). Some power is provided to the Cortex over USB, which will allow it to determine that it is connected to your computer.

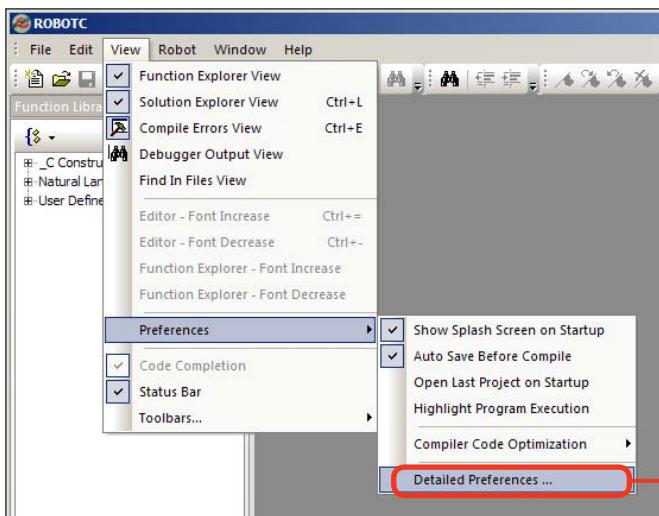


- 1b. Turn the Cortex ON
Make sure a 7.2V Robot battery is connected and move the POWER switch on the Cortex to the ON position.

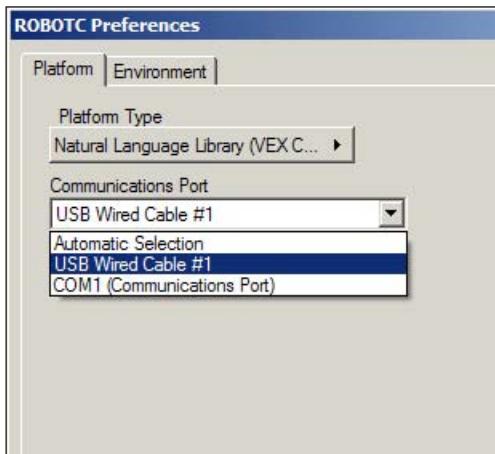
Note: If your Cortex is connected to a mobile robot, it's recommended that you prop the robot up to prevent its wheels from making contact with a surface. The motors may turn on and off during the firmware download process.

VEX Cortex Configuration over USB (cont.)

- Specify that you are using the Cortex and how it is connected to your computer in ROBOTC.



- 2a. Detailed Preferences...
Go to View > Preferences and select Detailed Preferences...

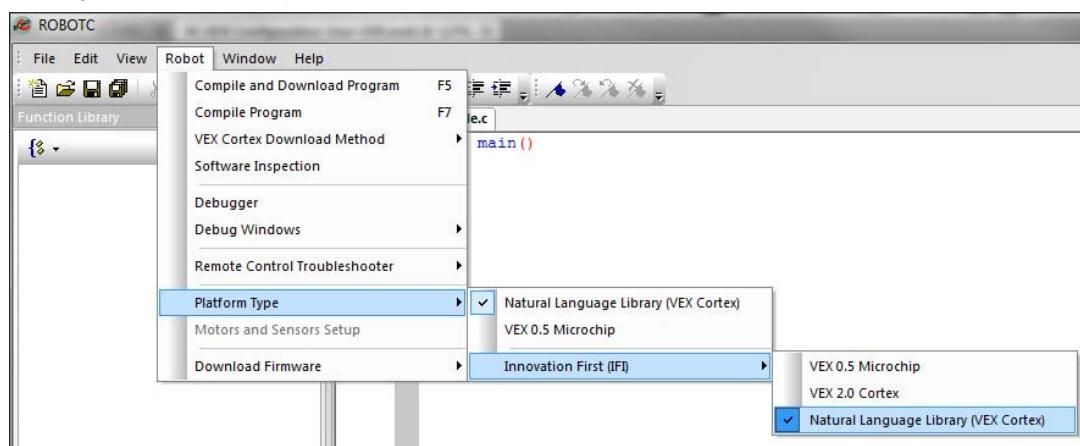


- 2b. Platform Settings
Make sure that the Platform tab is selected on the ROBOTC Preferences window.

Next, specify the Natural Language (VEX Cortex) as your Platform Type.

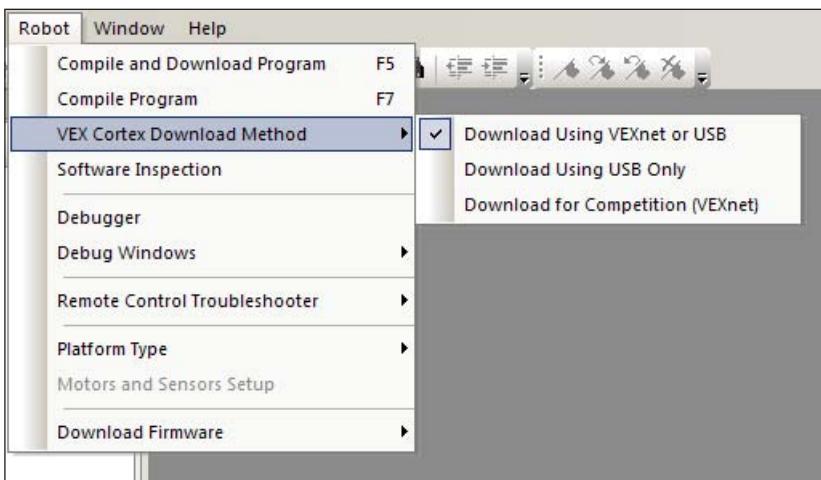
Finally, to program directly over the USB A-to-A cable, select the option that specifies the USB Wired Cable.
Press OK to finalize your settings.
Note: The Automatic Selection option should be used if you will be switching between VEXnet using the USB-to-Serial Programming Cable, and the USB A-to-A Cable.

Note: The Platform Type can also be modified by going to the Robot menu in ROBOTC, selecting Platform Type, and choosing one of the available options.



VEX Cortex Configuration over USB (cont.)

3. The VEX Cortex Download Method controls how ROBOTC downloads firmware and programs to your Cortex, as well as what types of connections your Cortex checks for when it is powered on. Confirm that your VEX Cortex Download Method is set to Download Using VEXnet or USB or Download Using USB Only.



Option 1: Download Using VEXnet or USB

With this option selected, ROBOTC will download ROBOTC firmware and programs to your Cortex using a VEXnet or USB connection. In this mode, when the Cortex is powered ON it will look for a VEXnet or USB connection for up to 10 seconds before running your program. (The Automatic Selection option in the ROBOTC Preferences should be selected if you plan on switching between VEXnet and USB as your download method.)

Option 2: Download Using USB Only

With this option selected, ROBOTC will download firmware and programs to your Cortex using only the USB connection. In this mode, when the Cortex is powered ON it will immediately run your program. This option is NOT recommended if you are using the VEXnet Joysticks to download to the Cortex, or remotely control it.

Option 1: Download for Competition (VEXnet)

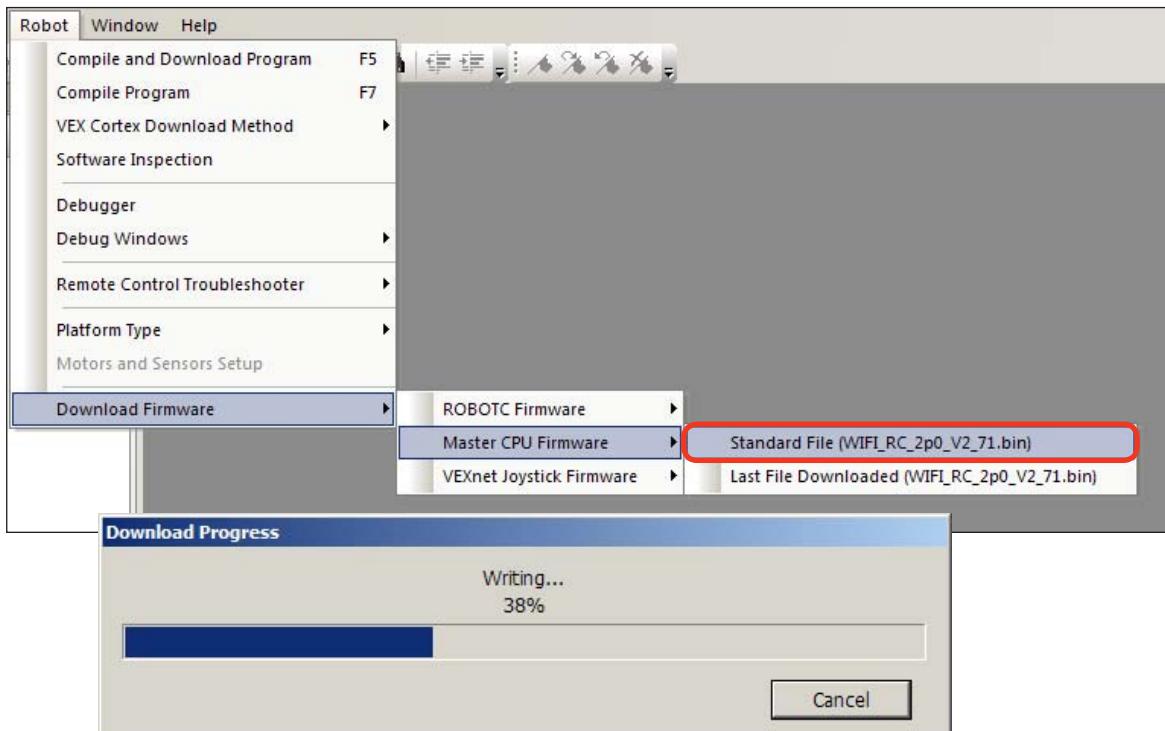
This option disables the ROBOTC debugger, and is not recommended for classroom use.

Important Note: Restarting the Cortex

The VEX Cortex Download Method setting is stored in ROBOTC and on the Cortex. If you change the setting, the Cortex must be power cycled (turned fully off, and then back on) before the change will take effect.

VEX Cortex Configuration over USB (cont.)

4. Go to Robot > Download Firmware > Master CPU Firmware and select Standard File to download the latest Master CPU Firmware to your robot.



4. Download Progress

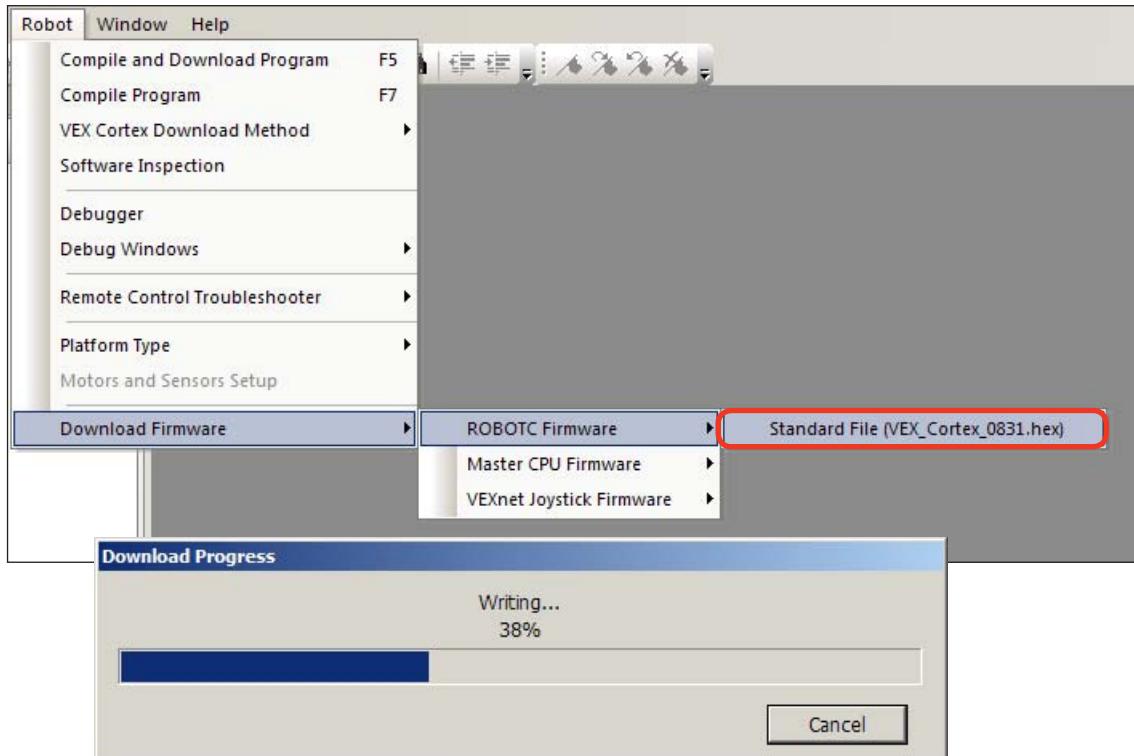
A Download Progress window will appear and begin the download process. When the window closes, the firmware download is complete. A ROBOTC Message will appear, and remind you to also download the ROBOTC Firmware.



Note: You only need to download the Master CPU Firmware once, when you first start using a VEX Cortex with ROBOTC, or when you upgrade to a newer version of ROBOTC. Switching programs or download methods does not require a re-download.

VEX Cortex Configuration over USB (cont.)

- The ROBOTC Firmware enables you to download ROBOTC programs to your robot and utilize the various debug windows. Go to Robot > Download Firmware > ROBOTC Firmware and select Standard File to download the ROBOTC Firmware to your robot.



- Download Progress
A Download Progress window will appear and begin the download process. When the window closes, the firmware download is complete.

Note: You only need to download the ROBOTC Firmware once, when you first start using a VEX Cortex with ROBOTC, or when you upgrade to a newer version of ROBOTC.

End of Lesson

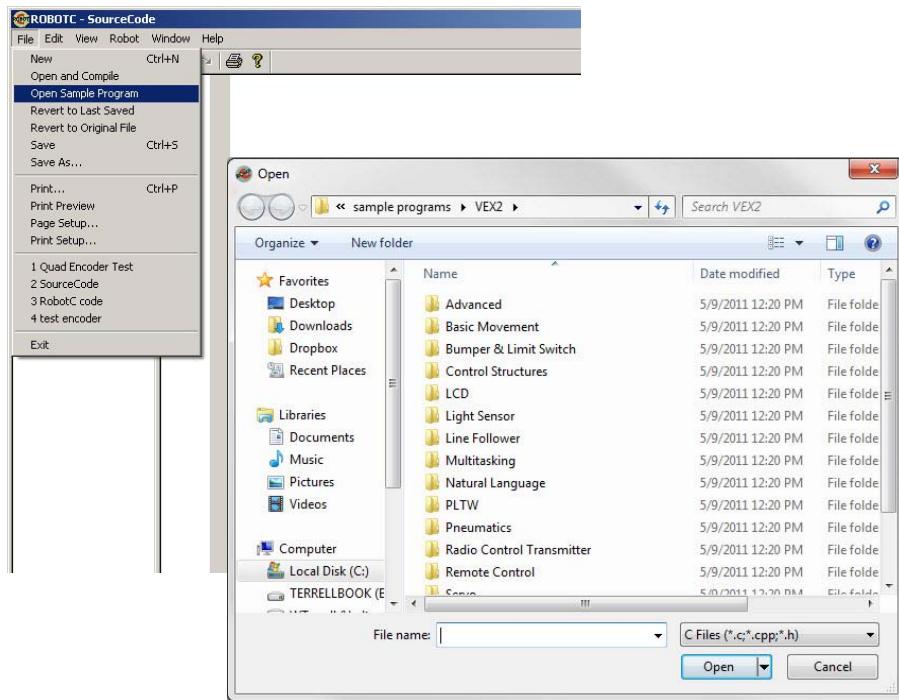
Once the Download Progress window closes, the ROBOTC Firmware download is complete. Your VEX Cortex is now ready to be programmed in ROBOTC.

If you are also using the VEXnet Joysticks, you can follow the provided instructions in the VEXnet Joysticks Setup document. Otherwise, move on to the Downloading Sample Programs over USB guide to learn how to download sample code, and verify that your setup is fully functional.

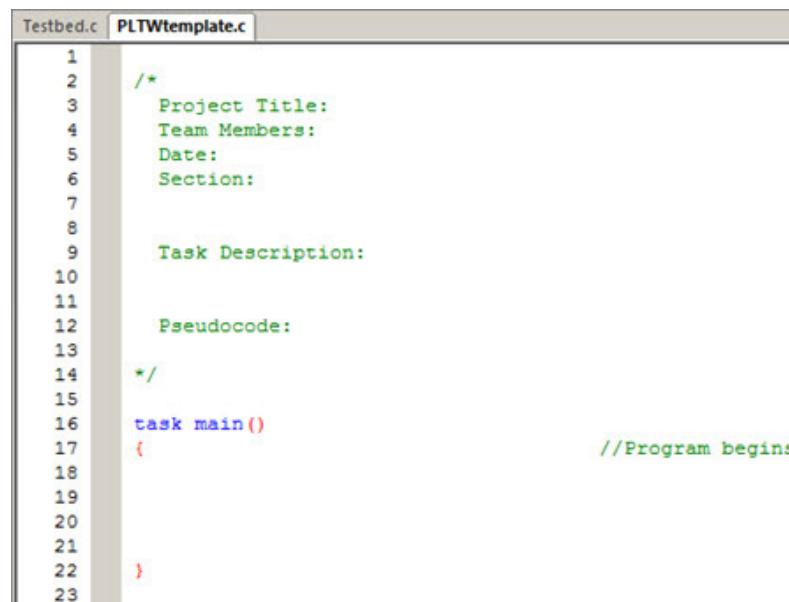
Using the PLTW Template

The PLTW template is the starting point for all your programs. The template is located in the Sample Programs in the PLTW folder.

Before typing in the template you MUST go to File, Save As, then navigate to the folder to save your robotics projects in, appropriately name your program, and click Save.



All Robotics Projects should be completed using the PLTW template. Open the PLTW folder in Sample Programs to get the PLTWtemplate file.



```

1
2  /*
3   Project Title:
4   Team Members:
5   Date:
6   Section:
7
8
9   Task Description:
10
11
12   Pseudocode:
13
14 */
15
16 task main()
17 {
18
19
20
21
22 }
23

```

Before typing in the template you MUST go to File, Save As, then navigate to the folder to save your robotics projects in, appropriately name your program, and click Save.

Using the PLTW Template

The commented section above task main provides an area to complete your identification information date, a narrative of what the program will and a place for pseudocode.

```

1  /*
2   * Project Title: Fill in personal information
3   * Team Members:
4   * Date:
5   * Section:
6
7
8
9   Task Description: Describe project requirements in your
10  own words
11
12  Pseudocode: Program planning
13 */
14
15
16 task main()
17 {
18   //Program begins
19
20   Section between curly braces is
21   designated for the actual program.
22 }
23

```

ROBOTC uses different colors to help identify code and text. This makes it easy to navigate through the program in addition to providing clues about mistakes when items are a different color than expected.

```

6  /*
7  Project Title: Robot Drag Race
8  Team Members: PLTW
9  Date: 2/1/11
10 Section: 4th Period
11
12 Task Description: Program a robot dragster to travel as fast
13 as possible after started with a switch.
14
15 Pseudocode:
16 {
17 When pushbutton pressed
18   {
19     Turn both motors on full power
20   }
21
22 /*
23
24 task main()
25 {
26   untilButtonPress(StartButton);
27   {
28     startMotor(Motor1, 127);
29     startMotor(Motor10, 127);
30   }
31 }
32
33

```

Notice the color changes in your document:

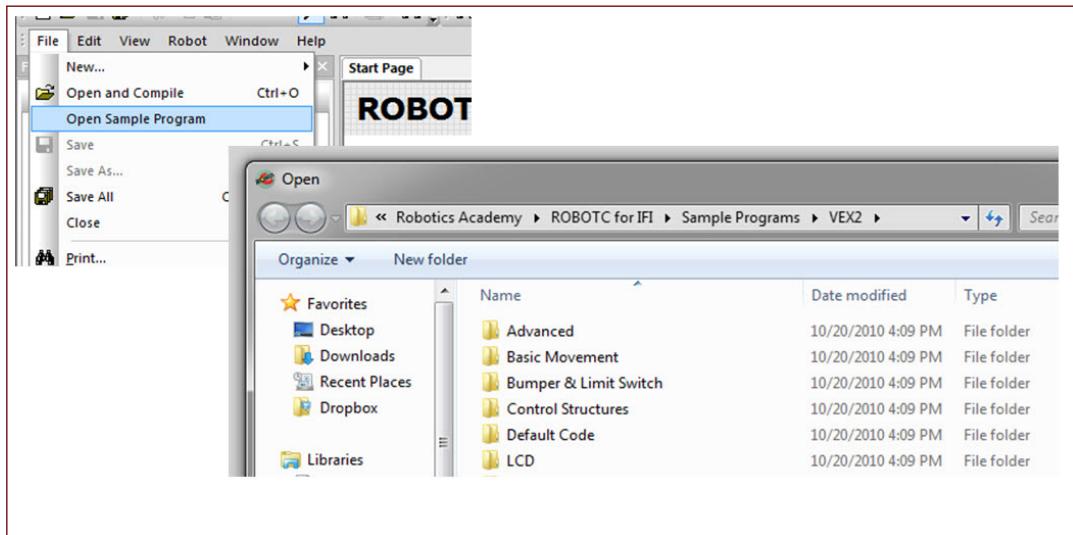
- Green = Comments (single and multiple line)
- Blue = RobotC reserved words
- Bright Red = syntax (i.e. {}, (), ;) and operators (i.e. =, ++, -)
- Black = Normal Text
- Dark Red = numbers

//When button is pressed
 //Motors 1 and 10 will start

Sample Programs

One of the easiest ways to begin programming is to start with existing code, try it out, and then modify it. ROBOTC includes over 70 sample programs to help you get started with learning how to program. To open a sample program, go to the File menu and select Open Sample Program. All of the ROBOTC sample programs have “comments” that tell how the robot should be configured, and

To access Sample Programs go File > Open Sample Programs.



Running a Program

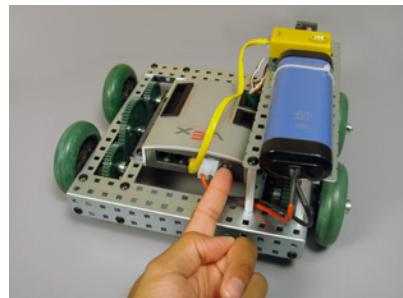
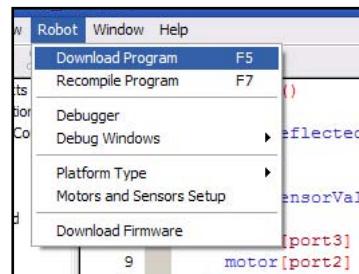
Once a program has been successfully written, it needs to be given to the robot to run.

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. Make sure your VEX is turned on. Move the switch from the OFF position to the ON position.
2. Make sure your robot is plugged in to the computer via the Programming Module. The program will be loaded onto the robot through this connection.
3. Click “Robot” on the top menu bar of the ROBOTC window, and select “Download Program” or “Compile and Download Program” (they’re effectively the same; which one you see depends on whether you have made changes since the last time you compiled).
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.

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 still 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 via its cable, you can obtain live variable and sensor feedback by using such debug windows as “Global Variables” and “Devices.”
- Run Independently
If you want to run the program while the robot is not connected just remove the cable once the program has been downloaded. On the back of the VEX, toggle the power switch in the “Off” position, and then back to “On”. The program will run immediately.



VEXnet Joystick Configuration in ROBOTC

The VEXnet Joystick enables more than just the remote control of your robot. It also provides the wireless communication link between your computer and the VEX Cortex, enabling you to wirelessly download firmware, programs and run the ROBOTC debugger. In this document, you will learn how to configure VEXnet Joystick using ROBOTC.

This document is broken into 3 sections:

1. Downloading Firmware to the VEXnet Joystick
2. Creating a Wireless Link Between the VEXnet Joystick and VEX Cortex
3. Calibrating the VEXnet Joystick Values

You will need:

- 1 VEXnet Joystick with 6 AAA Batteries
- 1 Small Phillips Screwdriver
- A computer with ROBOTC for Cortex and PIC installed
- 1 USB A-to-A Cable
- 1 USB-to-Serial Programming Cable

Section 1: Downloading Firmware to the VEXnet Joystick

1. Begin by installing 6 AAA batteries in the VEXnet Joystick. You will need a small Phillips screwdriver to remove the battery cover.



- 1a. Install 6 AAA Batteries
Remove the battery cover using a small Phillips screwdriver and install 6 AAA batteries, being careful to align them as indicated.



- 1b. Verify Correct Installation
Turn the VEXnet Joystick ON to verify that you correctly installed the batteries. If any of the LED's on the front turn on, you installed the batteries correctly. Turn the controller OFF and secure the battery cover using the Philips screwdriver.

VEXnet Joystick Configuration in ROBOTC (cont.)

2. Connect the VEXnet Joystick to your computer using the USB A-to-A cable and turn it ON.



- 2a. Connect the VEXnet Joystick
Use the USB A-to-A cable to connect your VEXnet Joystick to your computer.

Note: The VEXnet light should turn green.

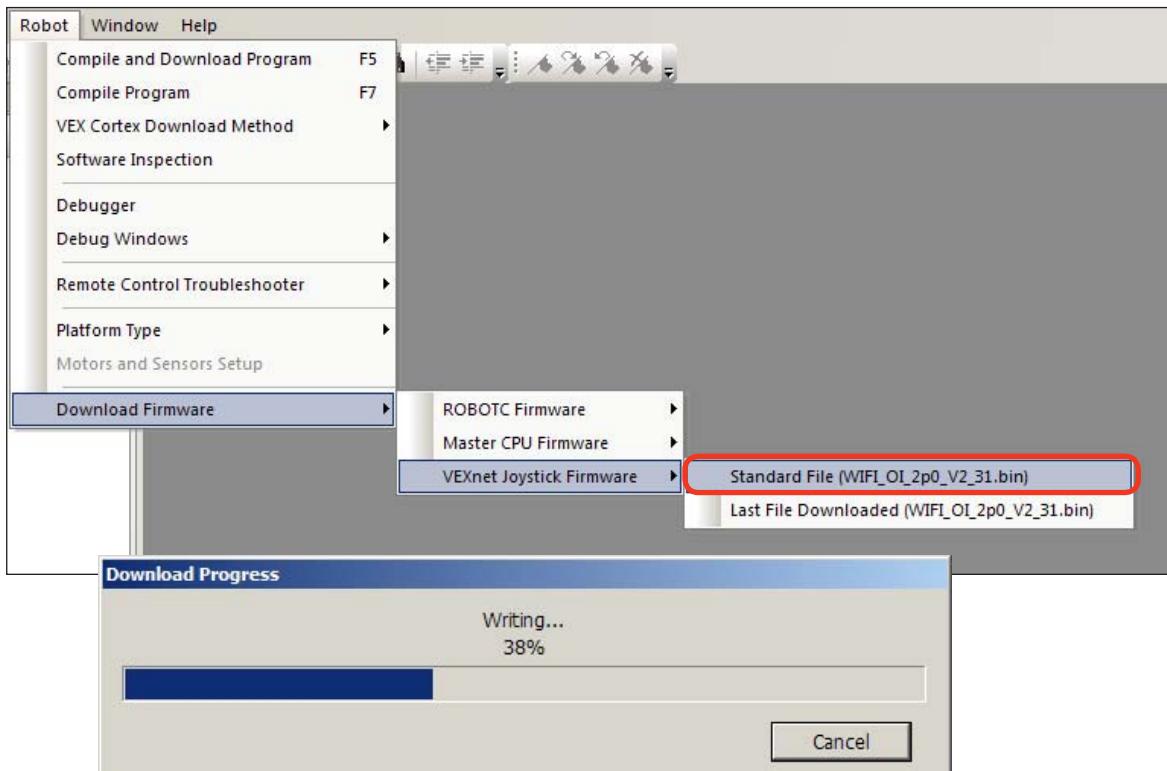


- 2b. Turn the VEXnet Joystick ON
Switch the VEXnet Joystick to the ON position.

Note: The Joystick light should turn green.

VEXnet Joystick Configuration in ROBOTC (cont.)

3. Go to Robot > Download Firmware > VEXnet Joystick Firmware and select Standard File to download the latest VEXnet Joystick Firmware to the controller.



3. Download Progress

A Download Progress window will appear and begin the download process. When the window closes, the firmware download is complete.

Note: You only need to download the VEXnet Joystick Firmware once, when you first start using a VEX Cortex with ROBOTC, or when you upgrade to a newer version of ROBOTC.

End of Section: Downloading Firmware to the VEXnet Joystick

Once the Download Progress window closes, the VEXnet Joystick Firmware download is complete.

Move on to the next section to learn how to create a wireless link between the VEXnet Joystick and VEX Cortex.

VEXnet Joystick Configuration in ROBOTC (cont.)

Section 2: Creating a wireless link between the VEXnet Joystick and VEX Cortex

In this section, you will learn how to pair a VEX Cortex Microcontroller to a VEXnet Joystick, allowing them to communicate over VEXnet. This section assumes that you have already updated the master firmware on the VEX Cortex and VEXnet Remote Control.

VEXnet is an 802.11 WiFi communication system between the VEX Cortex and VEXnet Remote Control.

VEXnet features include:

- Easy to connect (No IP addresses, MAC addresses, passwords, or special security modes)
- Multiple layers of security built-in and always on
- No wireless access point needed; each VEXnet pair makes its own private network
- Hundreds of robots can operate at once; every VEXnet robot has a hidden unique ID
- Optional tether for wired communication
- Optional 9V battery backup to maintain wireless link during a main 7.2V power loss
- LED scheme displays the status of the Robot, VEXnet link, and Game (Competition Mode)

1. Begin by verifying that both the Cortex and VEXnet Joystick are connected to charged batteries.



- 1a. Connect a Battery to the Cortex
Connect a 7.2V robot battery to the Cortex, but do not power it ON.



- 1b. Install Batteries in the VEXnet Remote Control
Remove the battery cover plate on the remote control. Install 6 AAA batteries, and replace the battery cover plate. Do not power the remote control ON.

VEXnet Joystick Configuration in ROBOTC (cont.)

2. Tether the USB port on the VEXnet Joystick to the USB port on the Cortex using a USB A-to-A cable.



- 2a. VEXnet Joystick USB Port
Plug one end of the USB A-to-A cable into the USB port on the VEXnet Joystick.



- 2b. VEX Cortex USB Port
Plug the other end of the USB A-to-A cable into the USB port on the VEX Cortex.

3. Power the Cortex ON. After a few seconds, ROBOT and VEXnet LEDs will blink green, indicating that the Cortex and VEXnet Joystick have successfully paired.



- 3a. Turn the Cortex ON

- 3b. Status LEDs
The ROBOT and VEXnet LEDs will blink green once the Cortex and VEXnet Joystick have successfully paired. The GAME LED will also blink green if a program is stored on your Cortex.

VEXnet Joystick Configuration in ROBOTC (cont.)

4. Turn the Cortex OFF.



5. Remove the USB A-to-A cable from the VEXnet Joystick and Cortex.



6. Insert VEXnet USB Keys into both the VEXnet Joystick and Cortex.



6. VEXnet USB Keys
Insert VEXnet USB Keys into the VEXnet Joystick and Cortex.

Note:

It does not matter which VEXnet USB Key you insert into the Cortex versus the VEXnet Joystick. Pairing the Cortex and VEXnet Joystick establishes the link; the VEXnet USB Keys simply act as antennas for the

VEXnet Joystick Configuration in ROBOTC (cont.)

7. Power the Cortex and Joystick ON. After roughly 15 seconds, the ROBOT and VEXnet LED's will blink green, indicating that the VEXnet communication link has been established.



7a. Turn the Cortex ON



7b. Turn the VEXnet Joystick ON



7c. Status LEDs

After roughly 15 seconds, the ROBOT and VEXnet status LEDs will start quickly blinking green. With the VEXnet link established, you should power OFF your Cortex and VEXnet Joystick to preserve battery.

End of Section: Creating a Wireless Link between the VEXnet Joystick and VEX Cortex
Your VEXnet Joystick and VEX Cortex can now communicate over the VEXnet USB Keys. Move on to the next section to calibrate the values your VEXnet Joystick sends out.

VEXnet Joystick Configuration in ROBOTC (cont.)

Section 3: Calibrating the VEXnet Joystick Values

This section contains the procedure for calibrating the VEXnet Remote Control joysticks. Some steps are time-sensitive, so it's recommended that you read through the instructions once before following along.

The VEXnet Remote Control includes two joysticks (each having an X and Y-axis), 8 buttons on the front, and 4 additional trigger buttons on the top. Inside, there is also 3-Axis accelerometer, capable of providing X-Y-Z acceleration values. Values from the joysticks, buttons, and accelerometer are sent as a constant stream of information over VEXnet to the robot, enabling a user to control the robot in real-time.

To ensure that the VEXnet Joystick sends out accurate joystick values, the joysticks must be calibrated before their first use, and after any firmware updates are applied.

You will need:

- A VEXnet Joystick with batteries
- A VEX Cortex with robot battery
- A small Allen wrench (1/16" or smaller) or paper clip

-
1. Power on the VEXnet Joystick and VEX Cortex. Allow them to sync over VEXnet.



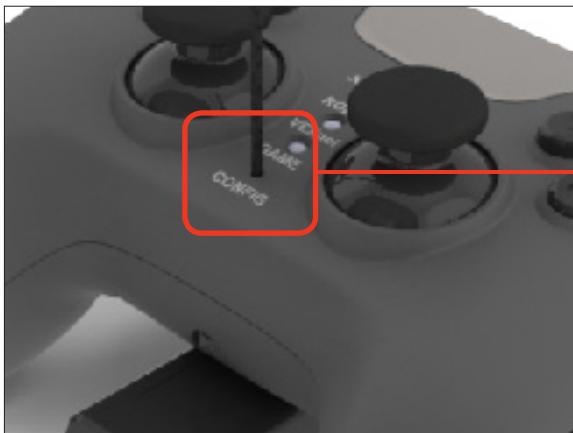
VEXnet Joystick Configuration in ROBOTC (cont.)

2. Press and hold the 6U trigger button.



2. Press and hold the 6U trigger button

3. While keeping the 6U trigger button pressed in, use your Allen wrench or paper clip to press in the internal CONFIG button until the JOYSTICK LED blinks red and green.



- 3a. Press and the CONFIG button
While still pressing in the 6U trigger button,
use an Allen wrench or paper clip to press
in
the CONFIG button.



- 3b. JOYSTICK LED
Once the JOYSTICK LED begins to blink
red and green, release both the 6U and
CONFIG buttons.

VEXnet Joystick Configuration in ROBOTC (cont.)



Important - Time Sensitive Instructions

There is a 10 second time limit to complete steps 4 and 5. If they are not completed in time, the calibration process will timeout and the VEXnet LED will blink red briefly.

4. Move both joysticks through their full ranges of motion. When the remote control detects that the joysticks have been fully rotated, the JOYSTICK LED stops blinking red and green, and switches to a solid green.



4a. Move the Joysticks

Move the joysticks through their full ranges of motion - Up, Down, Left, Right, and in a circle.



4b. JOYSTICK LED

Once the remote control detects that the joysticks have been fully rotated, the JOYSTICK LED switches to solid green, indicating that you can stop moving the joysticks.

VEXnet Joystick Configuration in ROBOTC (cont.)

5. Press the 8U button to save the new calibration.



5. Save

Press the 8U button to save the joystick calibration on your remote control. The JOYSTICK LED will blink green for a few seconds.



Additional Information

- If the calibration is not saved, the process will timeout after 10 seconds and the VEXnet LED will blink red.
- To cancel a calibration, press the 7U button. The calibration process will be discontinued and the VEXnet LED will blink red.
- Once the calibration is discontinued or saved, all of the remote control LEDs will resume their normal function.
- The joysticks must be calibrated any time the firmware on the remote control is downloaded.

End of Section: Calibrating the VEXnet Joystick Values

The joysticks on your VEXnet Joystick are now properly calibrated and ready to be used to remote control your robot. If you had any issues during the process, troubleshooting tips can be found on the following page.

VEXnet Joystick Configuration in ROBOTC (cont.)



Troubleshooting

Issue: Slow blinking green ROBOT light on the Cortex

Solution: Download the Cortex Master Firmware using ROBOTC.

Issue: Slow blinking ROBOT green light on the VEXnet Joystick

Solution: Push and hold CONFIG button for about 5 seconds, until the status LEDs starts blinking green. Release it, wait for another 5 seconds, and then turn the VEXnet Joystick OFF and then back ON. If that fails, download the VEXnet Joystick Firmware using ROBOTC.

Issue: Yellow or red ROBOT light on the Cortex

Solution: Make sure you are using fully charged Robot battery.

Issue: Yellow or red ROBOT light on the VEXnet Joystick, even though they are both green on the Cortex.

Solution: Power cycle both the VEXnet Joystick and CORTEX.

Sense Plan Act (SPA)

Sense, Plan, Act was an early robot control procedure commonly abbreviated SPA. Today we use its fundamental concepts to remind us of the three critical capabilities that every robot must have in order to operate effectively:

- SENSE:** The robot needs the ability to sense important things about its environment, like the presence of obstacles or navigation aids. What information does your robot need about its surroundings, and how will it gather that information?
- PLAN:** The robot needs to take the sensed data and figure out how to respond appropriately to it, based on a pre-existing strategy. Do you have a strategy? Does your program determine the appropriate response, based on that strategy and the sensed data?
- ACT:** Finally, the robot must actually act to carry out the actions that the plan calls for. Have you built your robot so that it can do what it needs to, physically? Does it actually do it when told?

Where are S, P, and A in this program?

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

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

SENSE: The robot uses a Bumper Switch to sense whether it has collided with an object.

- PLAN:** The overall strategy for this robot is to run forward unless something is in its way, which it will detect using the Bumper Switch. If the Bumper Switch is unpressed, the motors will be run forward; if the Bumper Switch is pressed, the robot will turn away from the obstacle. This is all captured in the program, which runs on the robot, reading the sensor's data and issuing the appropriate motor commands.
- ACT:** The robot acts by moving its motors in response to the given motor commands, which are given in combinations that produce forward movement and turns as appropriate.

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 |
|---|--|---|
| <code>SensorValue(sonarSensor) > 45</code> | 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 |
|-------------------|------------------|---------------|
| <code>1==1</code> | Is 1 equal to 1? | True, always |
| <code>0==1</code> | 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.

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

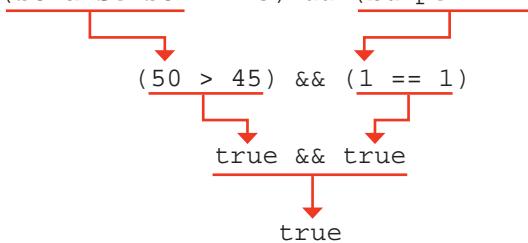
Logical Operators

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...



| 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 | <code>int</code> |
| Floating Point Decimal | Numeric values with decimal points (even if the decimal part is zero). | -.123, 0.56, 3.0, 1000.07 | <code>float</code> |
| Boolean | True or False. Useful for expressing the outcomes of comparisons. | true, false | <code>bool</code> |

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

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

```
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 |

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

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

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

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

Reserved Words

Data Types

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
```

While Loops with Natural Language

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

There are three main parts to every while loop.

Part 1. The keyword “while”.

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

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

Part 2. The condition.

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

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

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

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

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

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

```
task main()
{
    while(1 == 1)
    {
        startMotor(port2, 63);
        wait(5.0);

        startMotor(port2, -63);
        wait(5.0);
    }
}
```

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

While the condition is true, the port2 motor will turn forward for 5 seconds, then in reverse for 5 seconds.

Result: The port2 motor will turn back and forth, forever.

While Loops with Natural Language

Below is an example of a program using a counter-controlled While Loop.

```
task main()
{
    int count = 0;                                Creates an integer variable named "count" and gives it an initial value of 0.

    while(count < 4)                                Checks if count is "less than" 4.

    {
        startMotor(port2, 63);
        wait(5.0);

        startMotor(port2, -63);
        wait(5.0);

        count = count + 1;                            Adds 1 to count every time the loop runs.
    }
}
```

Result: The loop repeats 4 times, causing the port2 motor to turn back and forth, four times.

Below is an example of a program using a sensor-controlled While Loop.

```
#pragma config(Sensor, dgt11, Estop, sensorTouch)
#pragma config(Sensor, dgt12, controlBtn, sensorTouch)
#pragma config(Sensor, dgt13, LED, sensorDigitalOut)

task main()
{
    while(SensorValue[Estop] == 0)                Checks if the "Estop" touch sensor is equal to 0 (unpressed).

    {
        if(SensorValue[controlBtn] == 1)            If the "controlBtn" is pressed, turn the LED on; if it's not, turn the LED off.

        {
            turnLEDOn(LED);
        }
        else
        {
            turnLEDOFF(LED);
        }
    }
}
```

Result: The loop repeats continuously, allowing the LED to be turned on while the "controlBtn" is pressed, and off while "controlBtn" is released. The loop will stop as soon as the "Estop" touch sensor is pressed.

if Statements with Natural Language

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

Pseudocode of an if Statement:

```
if(condition)
{
    // true-commands
```

(condition)
Either true or false
(true) commands
Commands placed here will run if the (condition) is true.

Example program containing two if Statements:

```
task main()
{
    while(true)
    {
        if(SensorValue(bumper) == 0)
        {
            startMotor(port3, 63);
        }

        if(SensorValue(bumper) == 1)
        {
            stopMotor(port3);
        }
    }
}
```

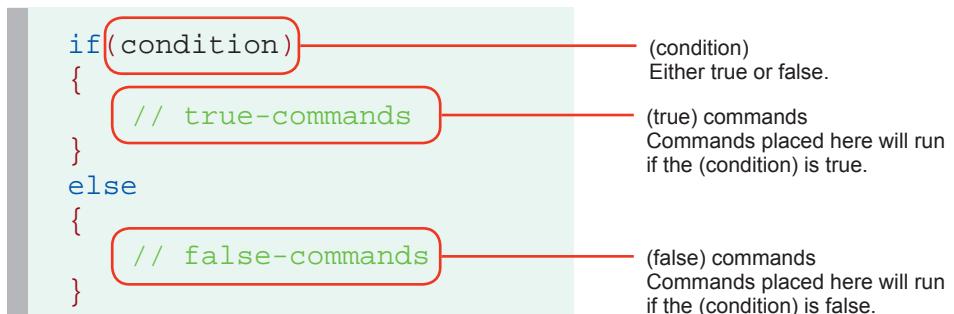
(condition)
true if the sensor is unpressed; false otherwise
(true) commands
Commands here run if the (condition) is true.
(condition)
true if the sensor is pressed; false otherwise
(true) commands
Commands here run if the (condition) is true.

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

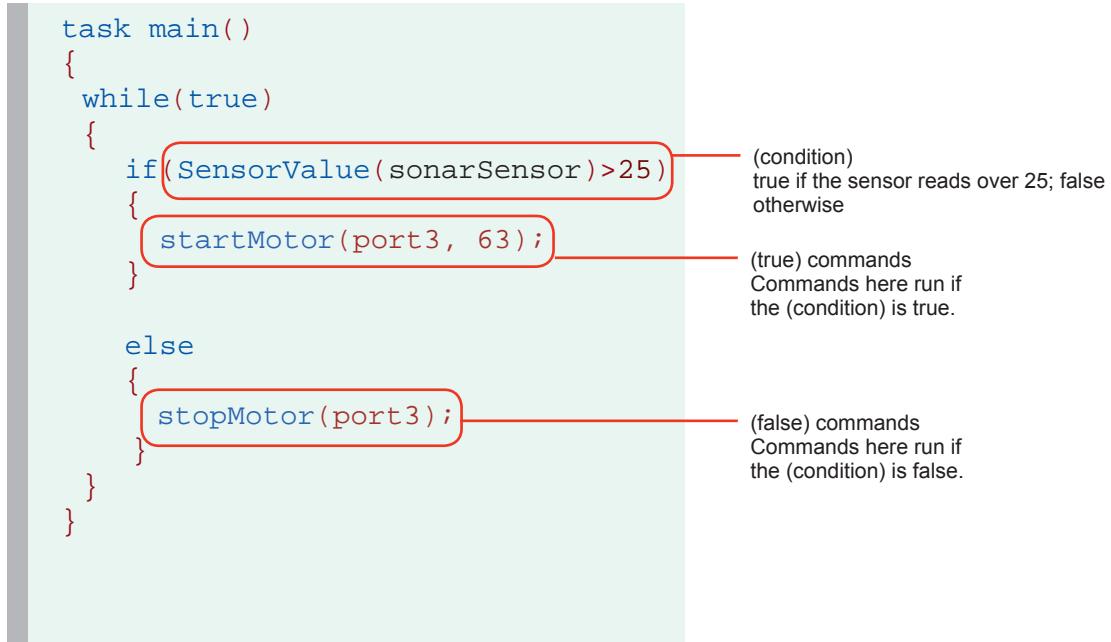
if-else Statements with Natural Language

The if-else Statement is an expansion of the basic if Statement. The “if” section still checks the condition and runs the appropriate commands when it evaluates to true, but using the “else” allows for specific code to be run only when the condition is false.

Pseudocode of an if-else Statement:



Example program containing an if-else Statement:



This if-else Statement tells the robot to run port3 at half power if the nearest object the Ultrasonic Rangefinder detects is more than 25 centimeters away. If the Ultrasonic Rangefinder detects an object closer than 25 centimeters, then the “else” portion of the code will be run and the motor on port3 will stop moving. The outer `while(true)` loop makes the if-else statement repeat forever.

Embedded if/if-else Statements with Natural Language

Sometimes, especially with more complex tasks, your robot will have to make multiple consecutive decisions before performing a behavior. This can be accomplished by embedding, or placing, if Statements within other if Statements.

Pseudocode of an embedded if Statement:

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

(condition)
Either true or false

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

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

Variables with Natural Language

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;
    startMotor(port3, speed);
    startMotor(port2, speed);
    wait1(2.0);
}
```

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 startMotor 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 startMotor 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;
    startMotor(port3, speed);
    startMotor(port2, speed);
    wait1(2.0);
}
```

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 with Natural Language

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

Rules for Variable Names

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

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

Variables with Natural Language

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

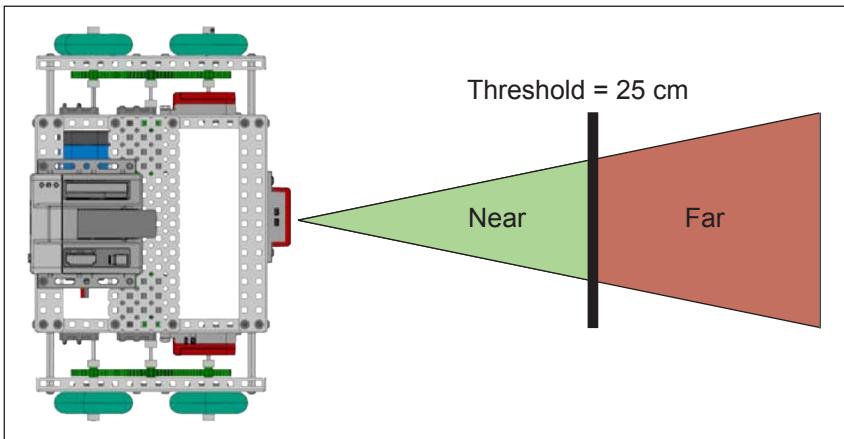
- “Use” a variable simply by putting its name where you want its value to be used
- The current value of the variable will be used every time the variable appears

Examples:

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

Thresholds with Natural Language

Thresholds are values that set a cutoff in a range of values, so that even if there are many possibilities, the value eventually falls above the threshold, or below the threshold. Using thresholds allows you to perform certain behaviors depending on where a certain value (usually a sensor value) falls in relation to the threshold.



If you look at this image, it shows an VEX using an Ultrasonic Rangefinder. The threshold in this case is 25 centimeters. We can create behaviors that tell the robot to go forward until the Ultrasonic Rangefinder detects something closer than 25 centimeters.

```
startMotor(leftMotor, 63);  
untilSonarLessThan(25);
```

The threshold is just used to determine at which point the robot should be performing a different behavior.

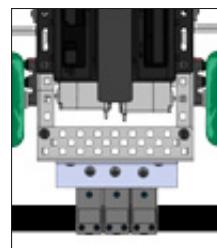
Calculated Thresholds

Some sensors, like the Ultrasonic Rangefinder and Potentiometer, provide the same set of values no matter what environment the robot is in; for the most part, their thresholds can simply be chosen based on their application. Other sensors, like the Light Sensor and Line Tracking Sensor, will provide very different values based on the environment they're in, due to factors such as the amount of ambient light or the type of surface the robot is tracking. Sensors that provide a different range of values based on their environment need to have their thresholds calculated.

For example, to find a dark line on a light surface, you must first calculate a threshold to distinguish light from dark. One recommended method is:

1. Measure the Line Follower Sensor value of the light surface. (For more information on finding sensor values, reference the ROBOTC Debugger document.)
2. Measure the Line Follower Sensor value of the dark surface
3. Add the two light sensor readings together
4. Divide by two to find the average, and use it as your threshold

In equation form:
$$\frac{\text{light value} + \text{dark value}}{2} = \text{threshold}$$



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.

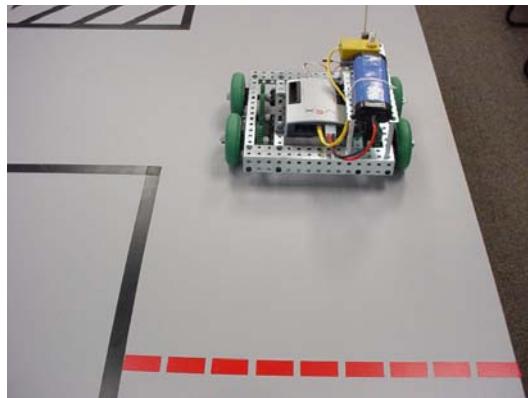
Behaviors

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()
{
    bMotorReflected[port2] = 1;

    motor[port3] = 63;
    motor[port2] = 63;
    wait1Msec(2000);

    motor[port3] = -63;
    motor[port2] = 63;
    wait1Msec(400);

    motor[port3] = 63;
    motor[port2] = 63;
    wait1Msec(2000);
}
```

The code is annotated with dashed boxes and descriptions:

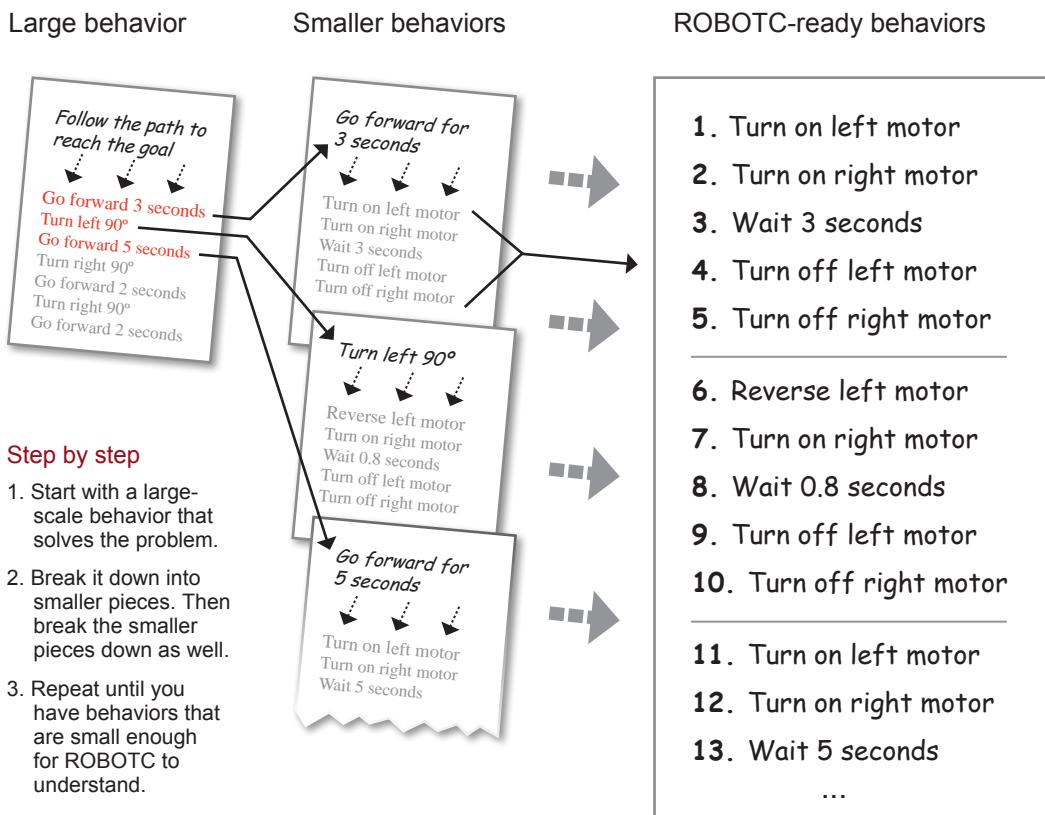
- Basic behavior**: This code turns the left motor on at half power.
- Simple behavior**: This code makes the robot go forward for 2 seconds at half power.
- Complex behavior**: This code makes the robot move around a

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.



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

Functions with Natural Language

A function is a group of statements that are run as a single unit when the function is called from another location, such as task main(). Commonly, each function will represent a specific behavior in the program.

Functions offer a number of distinct advantages over basic step-by-step coding.

- They save time and space by allowing common behaviors to be written as functions, and then run together as a single statement (rather than re-typing all the individual commands).
- Separating behaviors into different functions allows your code to follow your planning more easily (one function per behavior or even sub-behavior).
- Through the use of parameters, multiple related (but not identical) tasks can be handled with a single, intuitive function.

Using Functions

Functions must be created and then run separately. A function is created by “declaring” it, and run by “calling” it.

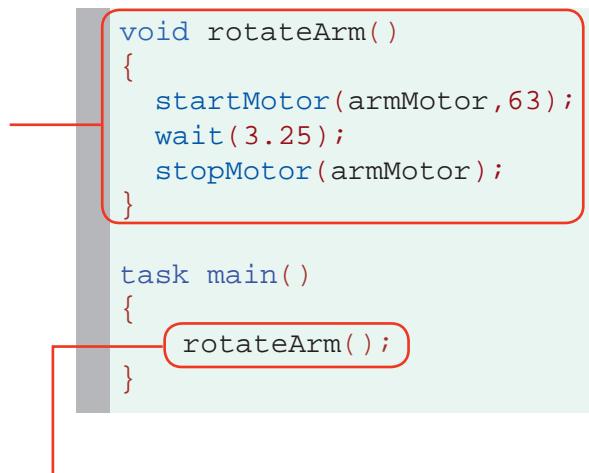
1. Declare Your Function

Declare the function by using the word “void”, followed by the name you wish to give to the function. It’s helpful to give the function a name that reflects the behavior it will perform.

Within the function’s {curly braces}, write the commands exactly as you would normally. When the function is called, it will run the lines between its braces in order, just like task main does with the code between its own braces.

2. Call Your Function

Once you declare your function, it acts like a new command in the language of ROBOTC. To run the function, simply “call” it by name – remember that its name includes the parentheses – followed by a semicolon.



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

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

The diagram illustrates the relationship between a function declaration and its invocation. On the left, a vertical grey bar represents memory. Inside, the `rotateArm()` function is declared with its body containing `startMotor`, `wait`, and `stopMotor` commands. On the right, the `main()` task is shown, which calls the `rotateArm()` function. Red arrows point from the declaration to the function call, indicating the flow of control.

Advanced Functions

Parameters

Parameters are a way of passing information into a function, allowing the function to run its commands differently, depending on the values it is given. It may help to think of the parameters as placeholders – all parameters must be filled in with real values when the function is called, so in the places where a parameter appears, it will simply be replaced by its given value.

1. Declare parameter

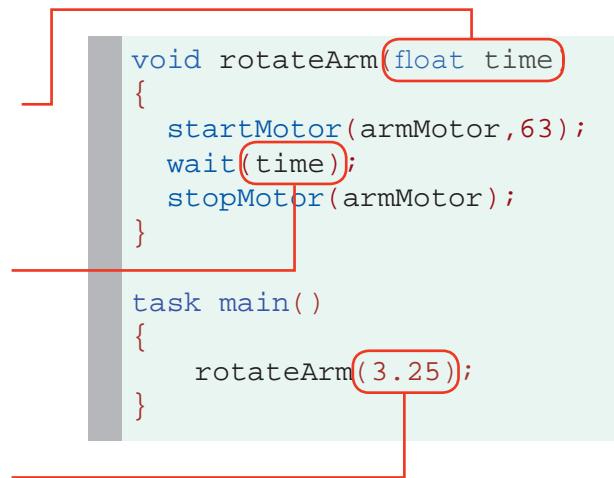
A parameter is declared in the same way that a variable is (type then name) inside the parentheses following the function name.

2. Use parameter

The parameter value behaves like a “placeholder”. Whatever value is provided for the parameter when the function is called will appear here.

3. Call function with parameter

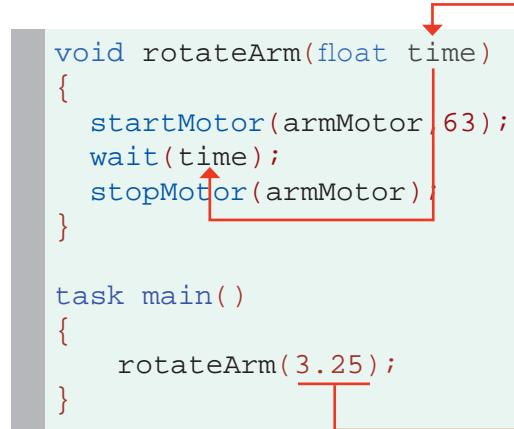
When the function is called, a value must be provided within the parentheses to take the place of the parameter inside the function.



Substitution

The arrows in the illustration to the right show the general “path” of the value from the place where it is provided in the function call, to where its value is substituted into the function.

The function will run as if the code read as it does in the bottom box.



```

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

Advanced Functions

Return Values

Not all functions are declared “void”. Sometimes, you may wish to capture a mathematical computation in a function, for instance, or perform some other task that requires you to get information back out of the function at the end. The function will “return” a value, causing it to behave as if the function call itself were a value in the line that called it.

1. Declare return type

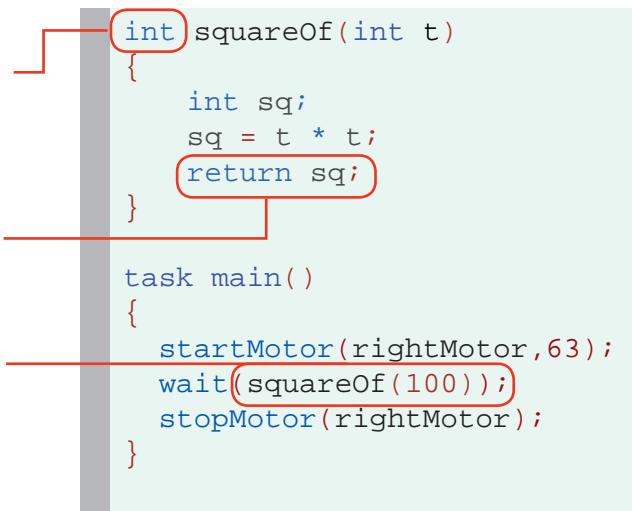
Because the function will give a value back at the end, it must be declared with a type other than void, indicating what type of value it will give.

2. Return value

The function must “return” a value. In this case, it is returning the result of a computation, the square of the parameter.

3. Function call becomes a value

The function call itself becomes a value to the part of the program that calls it. Here, it is acting as an integer value for the wait command.

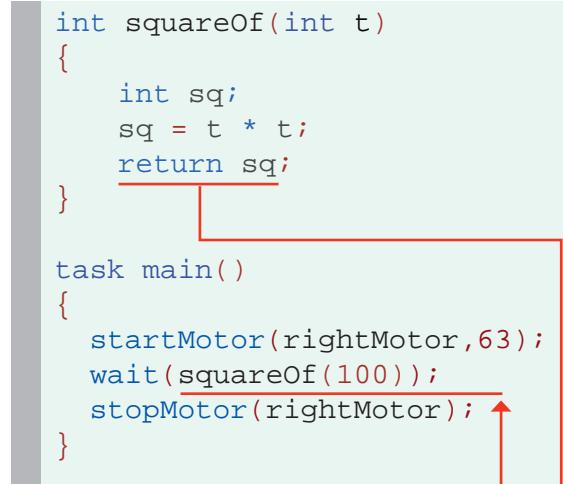


Substitution

The arrows in the illustration to the right show the general “path” of the value as it is returned and goes back into the main function.

The parameter 100 is used (as `t` in the function) to calculate the value, but is not shown in the arrows.

The function will run as if the code read as it does in the bottom box.

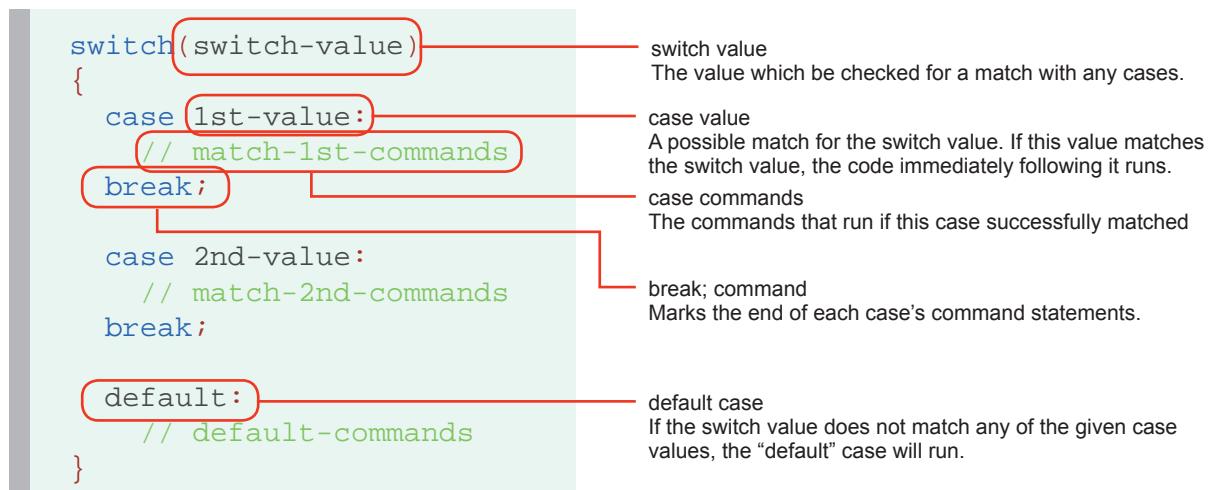


```
wait(10000);
```

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 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

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.

Random Numbers

Sometimes a behavior will call for a robot to use a random number in one of its measurements. This may seem strange, but randomness can actually be helpful to a robot in avoiding patterns of movement that would otherwise get it “stuck”.

Using Random Numbers

Random numbers is pretty straightforward. Wherever you want the random number to appear, simply add the code `random(maxNumber)`. Each time the line is run, a random (whole) number between 0 and the number you entered will fill in the spot where the `random()` command is.

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

Wait for a random time
The number of milliseconds
that the `wait1Msec` command
will wait for will be a random
number between 0 and 5000.

This program runs the robot
forward for a random amount
of time up to 5 seconds.

Using Other Numbers

If you need something other than whole numbers between zero and something, you may need to be a little creative...

4000 + random(1000)

Minimum value (as shown: 4000-5000)
Adding the random value “on top of” a
base number lets you get random numbers
between a minimum (the base number) and
a maximum (base+maximum random) value.

random(100)/100

Percent (as shown: 0-100% in 1% increments)
Dividing your random value by its own maximum
value normalizes the value so that it always falls
between 0 and 1.

Seeds

Computers can’t be truly random. Instead, they try to use a hard-to-predict series of numbers based off a “seed” value. Under certain circumstances, you may want to set the seed

srand(123);
 wait1Msec(random(5000));

Set random seed
The `srand` command sets the random
number seed for this robot. Run with the
same seed, “random” numbers will always be
generated in the same sequence.

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 controls a motor and an LED as long as a touch sensor is not pressed. A motor turns on and an LED turns off if no object is detected within 20cm of a sonar sensor; the motor turns off and an LED turns on if an object is detected within 20 cm.

```
task main()
{
    while ( touch sensor is not pressed )
    {
        if(sonar detects object > 20cm away)
        {
            Right Motor runs forward
            Red LED turns off
        }
        else
        {
            Right Motor stops
            Red LED turns on
        }
    }
}
```

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: “wait 1 second”; “increment LineCount by 1”; or “motors full ahead”.



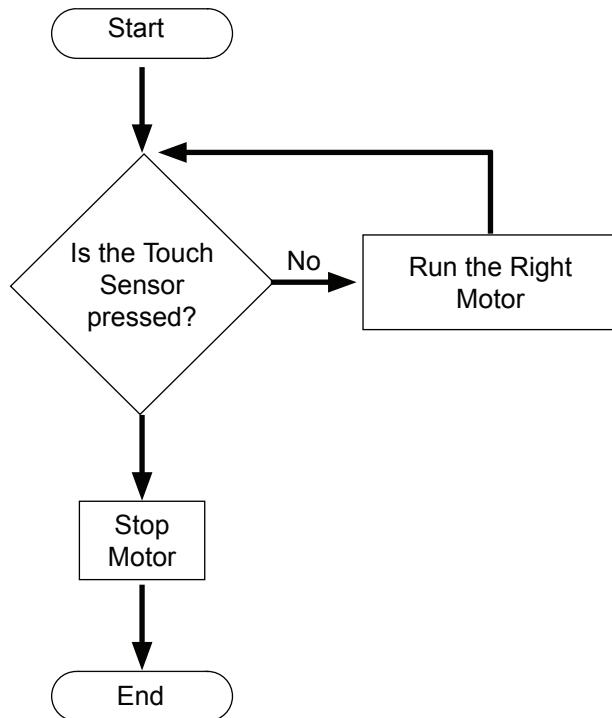
Decision blocks are represented as diamonds. These typically contain Yes/No questions. Decision blocks have two or more arrows coming out of them, representing the different paths that can be followed, depending on the outcome of the decision. The arrows should always be labeled accordingly.



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



Program Design

Planning your program may occur after you have sketched or built your physical device. It may also occur before or at the same time depending on the your challenge. Regardless it is good practice to do some planning for your program before writing code. This document outlines a strategy for a simple example to show the process. Please review the reference **Behaviors**, to familiarize yourself with basic, simple, and complex behaviors.

The steps below should be documented in your engineering notebook . Some of the information will then be transferred to the PLTW ROBOTC program template.

1. Describe the task or overall goal that your program will accomplish. This may be described as one or more **complex behaviors**.

Example: A fan will run until someone needs it to stop. There will be a warning light as a safety device before the fan turns on and another light to indicate that the fan has stopped.

*Note: This text will be used for the **Task Description** in the PLTW ROBOTC program template*

Creating Pseudocode

As you begin to break down your behaviors into individual actions do not worry about syntax or which commands will be used with ROBOTC. Simply describe them in short phrases such *turn a motor on for three seconds* or *follow a line until running into a wall*.

2. For each complex behavior break it down into **Simple Behaviors** line by line in the order that each should happen. Try to describe actions and what prompts each action to continue, start, stop, etc.

Example:

A warning light comes on before the fan starts for three seconds

The fan turns on and runs until a button is pressed

A different light turns for three seconds before the program stops

3. For each simple behavior, break it down further to **Basic Behaviors**. Try to think in terms of what each input and output component will be on your device.

Example:

Program begins

Light 1 (LED 1) turns on

for three seconds

Fan (Motor 1) turns on

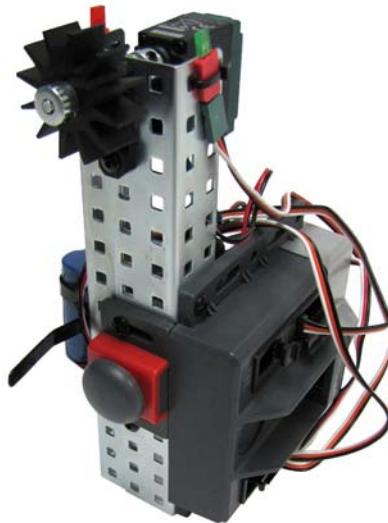
Until a button (bumper switch)is pressed

Light 2 (LED 2) turns on

for 3 seconds

Program ends

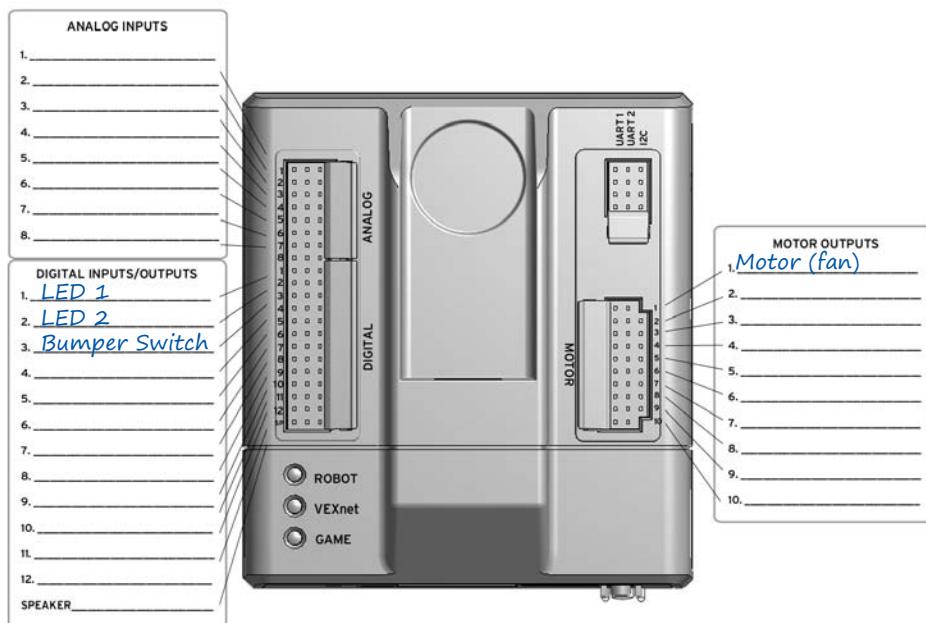
*Note: This text will be used for the **Pseudocode** in the PLTW ROBOTC program template*



Program Design Identify Inputs and Outputs

4. Now that you know what inputs and outputs you will need, identify which ports each will be plugged into on the Cortex. Pay attention which sensors are analog and which are digital. Below is a sketch of a possible configuration for the example on the previous page.

Example:



Note: The last page of this document contains a clean image of the Cortex that you can label, then cutout and attach in your engineering notebook for your own projects.

Program Design **PLTW ROBOTC Program Template**

Note: Be sure the Cortex you are using has been updated with the Master CPU and ROBOTC Firmware. Refer to the reference **Firmware Over USB** to acquire detailed instruction for this procedure.

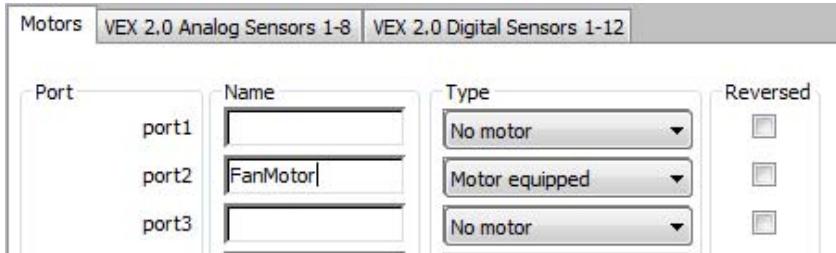
5. Open ROBOTC and open the Sample Program PLTWTTemplate.
6. Use your initial description (Complex Behaviors) of your overall goal for the program for the Task Description.
7. Copy your final pseudocode (Basic Behaviors) for the Pseudocode section of the PLTW ROBOTC program template.
8. It is recommended that you include your pseudocode mostly intact as comments beside programming commands.

Example:

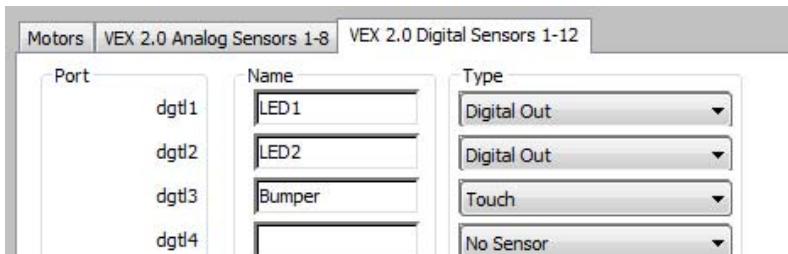
```
5  /*
6   * Project Title:
7   * Team Members:
8   * Date:
9   * Section:
10
11
12   Task Description:
13
14   A fan will run until someone needs it to stop. There will be a warning light
15   as a safety device before the fan turns on and another light to indicate that the
16   fan has stopped.
17
18   Pseudocode:
19
20   Program begins
21   Light 1 (LED 1) turns on
22   for three seconds
23   Fan (Motor 1) turns on
24   Until a button (bumper switch) is pressed
25   Light 2 (LED 2) turns on
26   for 3 seconds
27   Program ends
28
29 */
30
31 task main()
32 {
33   //Program begins
34   //Light 1 (LED 1) turns on
35   //for three seconds
36   //Fan (Motor 1) turns on
37   //Until a button (bumper switch) is pressed
38   //Light 2 (LED 2) turns on
39   //for 3 seconds
40   //Program ends
41 }
```

Program Design **PLTW ROBOTC Program Template Cont.**

9. Identify all inputs and outputs in the Motors and Sensors Setup window.

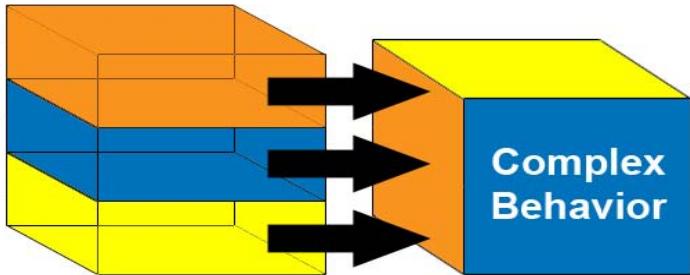


10. Use the Debugger to confirm that all inputs and outputs are working as expected. Refer to the reference **Debugger** to learn more about these functions.



Program Design PLTW ROBOTC Program Template Cont.

Basic Behavior
Basic Behavior
Basic Behavior



Remember many basic behaviors generally come together to create a complex behavior . You can solve simple and basic behaviors one at a time, and troubleshoot them as they come together to form a complex behavior.

Test and debug the combined program. Make sure your behavior functions as intended within the program. Many times, you will need to make adjustments to compensate for orientation, momentum, or other unforeseen factors as they begin to work together

11. Code and test small behaviors or sets of behaviors individually and edit / add comments as you build your code.

```

33  task main()
34  {
35      //Program begins
36      turnLEDOn(LED1);           //LED1 turns on
37      wait(3);                 //for three seconds
38      turnLEDOFF(LED1);        //LED1 turns off
39
40      //FanMotor turns on
41      //Until Bumper is pressed
42
43
44
45
46

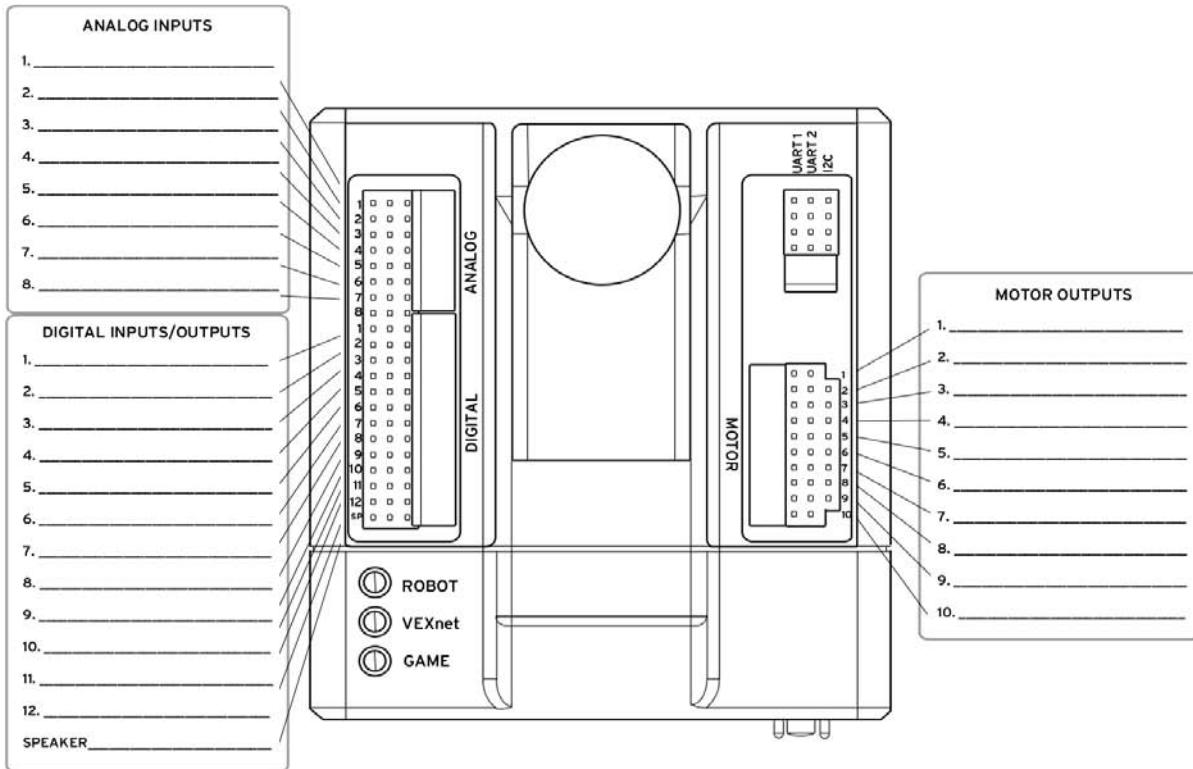
```

12. Continue programming and testing one behavior at a time. To test individual behaviors as you go temporarily turn sections of code into comments using /* followed by */.

```

33  task main()
34  {
35      //Program begins
36      /*
37      turnLEDOn(LED1);           //LED1 turns on
38      wait(3);                 //for three seconds
39      turnLEDOFF(LED1);        //LED1 turns off
40
41
42      startMotor(FanMotor, 127); //FanMotor turns on
43      untilBump(Bumper);       //Until Bumper is pressed
44      stopMotor(FanMotor);     //FanMotor turns off
45
46      //Light 2 (LED 2) turns on

```



ROBOTC Natural Language - Cortex Quick Reference:

| | | |
|---|--|---|
| Set Servo Set a servo to a desired position. Default servo and position: port6, 0. | <code>setServo();</code> | <code>setServo(port7, 95);</code> |
| Start Motor Set a specific motor to a speed. Default motor and speed: port6, 95. | <code>startMotor();</code> <code>wait();</code> <code>stopMotor();</code> | <code>startMotor(port8, -32);</code> <code>wait(0.5);</code> <code>stopMotor(port8);</code> |
| Stop Motor Stop a specific motor. Default motor: port6. | <code>startMotor();</code> <code>wait();</code> <code>stopMotor();</code> | <code>startMotor(port8, -32);</code> <code>wait(0.5);</code> <code>stopMotor(port8);</code> |
| Wait Wait an amount of time measured in seconds. Default time: 1.0. | <code>startMotor();</code> <code>wait();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>wait(2.7);</code> <code>stop();</code> |
| Wait in Milliseconds Wait an amount of time measured in milliseconds. Default time: 1000. | <code>startMotor();</code> <code>waitInMilliseconds();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>waitInMilliseconds(2700);</code> <code>stop();</code> |
| Until Touch The robot waits for the Touch Sensor to be pressed. Default sensor port: dgtl6. | <code>startMotor();</code> <code>untilTouch();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilTouch(dgtl10);</code> <code>stop();</code> |
| Until Release The robot waits for the Touch Sensor to be released. Default sensor port: dgtl6. | <code>startMotor();</code> <code>untilRelease();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilRelease(dgtl10);</code> <code>stop();</code> |
| Until Bump The robot waits for the Touch Sensor to be pressed in and then released out. Default sensor port: dgtl6. | <code>startMotor();</code> <code>untilBump();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilBump(dgtl10);</code> <code>stop();</code> |
| Until Button Press The robot waits for a button on the VEX LCD to be pressed. | <code>startMotor();</code> <code>untilButtonPress();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilButtonPress(rightBtnVEX);</code> <code>stop();</code> |
| Until Sonar - Less Than The robot waits for the Sonar Sensor to read a value in cm less than the threshold. Default threshold and sensor port: 30, dgtl8+9. | <code>startMotor();</code> <code>untilSonarLessThan();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilSonarLessThan(45, dgtl12);</code> <code>stop();</code> |
| Until Sonar - Greater Than The robot waits for the Sonar Sensor to read a value in cm greater than the threshold. Default threshold and sensor port: 30, dgtl8+9. | <code>startMotor();</code> <code>untilSonarGreater Than();</code> <code>stop();</code> | <code>startMotor(port8, 63);</code> <code>untilSonarGreater Than(45, dgtl12);</code> <code>stop();</code> |

ROBOTC Natural Language - Cortex Quick Reference:

| | | |
|--|--|---|
| Until Potentiometer - Greater Than The robot waits for the Potentiometer Sensor to read a value greater than a set position. Default threshold and sensor port: 2048, in6. | <code>startMotor(port8, 63); untilPotentiometerGreaterThan(); stop();</code> | <code>startMotor(port8, 63); untilSonarGreaterThan(4000, in4); stop();</code> |
| Until Potentiometer - Less Than The robot waits for the Potentiometer Sensor to read a value less than a set position. Default threshold and sensor port: 2048, in6. | <code>startMotor(port8, 63); untilPotentiometerLessThan(); stop();</code> | <code>startMotor(port8, 63); untilSonarLessThan(40, in4); stop();</code> |
| Until Dark The robot waits for the Light Sensor to read a value less than the threshold. Default threshold and sensor port: 505, in2. | <code>startMotor(); untilDark(); stop();</code> | <code>startMotor(port8, 63); untilDark(1005, in4); stop();</code> |
| Until Light The robot waits for the Light Sensor to read a value greater than the threshold. Default threshold and sensor port: 505, in2. | <code>startMotor(); untilLight(); stop();</code> | <code>startMotor(port8, 63); untilLight(1005, in4); stop();</code> |
| Until Rotations The robot waits for an encoder to reach a specified number of rotations. Default rotations, encoder: 1.0, dgtl1+2 | <code>startMotor(); untilRotations(); stop();</code> | <code>startMotor(port8, 63); untilRotations(2.75, dgtl3); stop();</code> |
| Until Encoder Counts The robot waits for an encoder to reach a specified number of encoder counts. Default counts, encoder: 360, dgtl1+2. | <code>startMotor(); untilEncoderCounts(); stop();</code> | <code>startMotor(port8, 63); untilEncoderCounts(990, dgtl3); stop();</code> |
| LED ON Turn an LED in a specified digital port ON. Default sensor port: dgtl2. | <code>turnLEDon(); wait(); turnLEDOff();</code> | <code>turnLEDon(dgtl7); wait(0.5); turnLEDOff(dgtl7);</code> |
| LED OFF Turn an LED in a specified digital port OFF. Default sensor port: dgtl2. | <code>turnLEDon(); wait(); turnLEDOff();</code> | <code>turnLEDon(dgtl7); wait(0.5); turnLEDOff(dgtl7);</code> |
| Flashlight ON Turn a VEX Flashlight in a specified motor port ON at a specified brightness. Default motor port and brightness: port4, 63. | <code>turnFlashlightOn(); wait(); turnFlashlightOff();</code> | <code>turnFlashlightOn(port10, 127); wait(0.5); turnFlashlightOff(port10);</code> |
| Flashlight OFF Turn a VEX Flashlight in a specified motor port OFF. Default motor port: port4. | <code>turnFlashlightOn(); wait(); turnFlashlightOff();</code> | <code>turnFlashlightOn(port10, 127); wait(0.5); turnFlashlightOff(port10);</code> |
| Robot Type Choose which robot you are using (Recbot or Swervebot). Default bot: none. | <code>robotType();</code> | <code>robotType(swervebot);</code> |

ROBOTC Natural Language - Cortex Quick Reference:

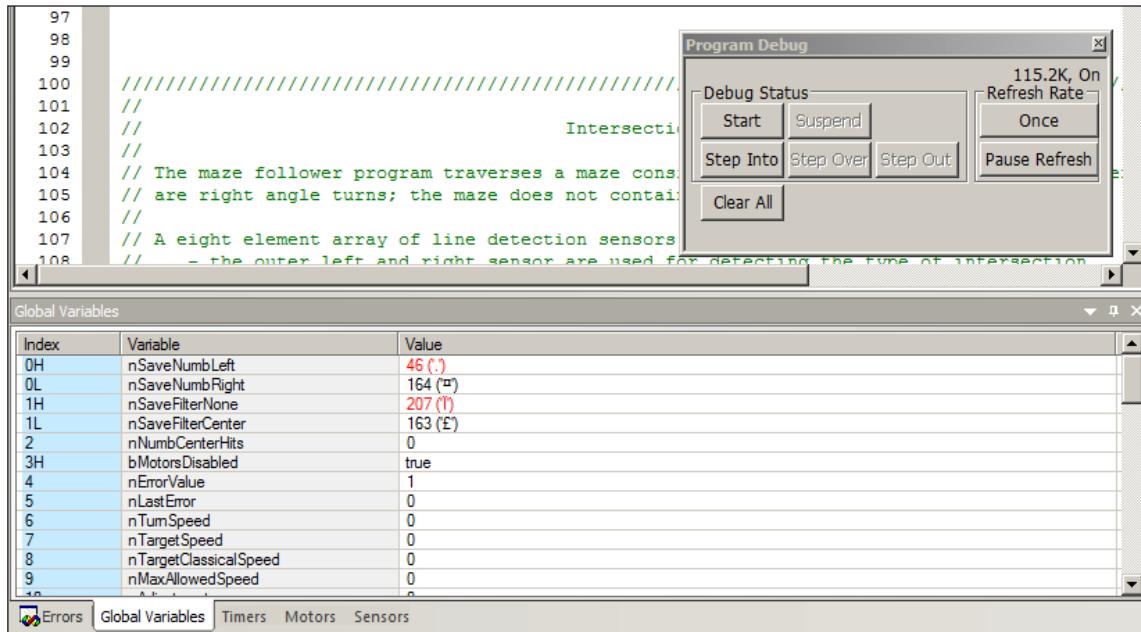
| | | |
|---|---|---|
| Forward The robot drives straight forward. Default speed: 95. | <code>forward();</code> <code>wait();</code> <code>stop();</code> | <code>forward(63);</code> <code>wait(2.0);</code> <code>stop();</code> |
| Backward The robot drives straight backward. Default speed: -95. | <code>backward();</code> <code>wait();</code> <code>stop();</code> | <code>backward(63);</code> <code>wait(2.0);</code> <code>stop();</code> |
| Point Turn The robot makes a sharp turn in place. Default direction and speed: right, 95. | <code>pointTurn();</code> <code>wait();</code> <code>stop();</code> | <code>pointTurn(left, 63);</code> <code>wait(0.4);</code> <code>stop();</code> |
| Swing Turn The robot makes a wide turn, activating only one drive motor. Default direction and speed: right, 95. | <code>swingTurn();</code> <code>wait();</code> <code>stop();</code> | <code>swingTurn(left, 63);</code> <code>wait(0.75);</code> <code>stop();</code> |
| Stop The robot halts both driving motors, coming to a stop. | <code>forward();</code> <code>wait();</code> <code>stop();</code> | <code>forward(63);</code> <code>wait(2.0);</code> <code>stop();</code> |
| Line Track - for Time The robot tracks a dark line on a light surface for a specified time in seconds. Default time, threshold, sensors: 5.0, 505, in1, in2, in3 (Left, Center, | <code>lineTrackForTime();</code> <code>stop();</code> | <code>lineTrackForTime(7.5, 99, in6, in7, in8);</code> <code>stop();</code> |
| Line Track - for Rotations The robot tracks a dark line on a light surface for a specified distance in rotations. Default time, threshold, sensors: 3.0, 505, in1, in2, in3 (Left, Center, | <code>lineTrackForRotations();</code> <code>stop();</code> | <code>lineTrackForRotations(4.75, 99, in6, in7, in8);</code> <code>stop();</code> |
| Move Straight - for Time The robot will use encoders to maintain a straight path for a specified time in seconds. Default time, rightEncoder, leftEncoder: 5.0, dgtl1+2, dgtl3+4. | <code>moveStraightForTime();</code> <code>stop();</code> | <code>moveStraightForTime(7.5, dgtl5, dgtl13);</code> <code>stop();</code> |
| Move Straight - for Rotations The robot will use encoders to maintain a straight path for a specified distance in encoder rotations. Default rotations, rightEncoder, leftEncoder: 1.0, dgtl1+2, dgtl3+4. | <code>moveStraightForRotations();</code> <code>stop();</code> | <code>moveStraightForRotations(4.75, dgtl5, dgtl13);</code> <code>stop();</code> |
| Tank Control The robot is remote controlled with the right motor mapped to the right joystick and the left motor mapped to the left joystick. Default right and left and joystick: Ch2, Ch3. | <code>while(true)</code> { <code>tankControl();</code> } | <code>while(true)</code> { <code>tankControl(Ch1, Ch4);</code> } |
| Arcade Control The robot is remote controlled with both motors mapped to a single joystick. Default vertical and horizontal joysticks: Ch2, Ch1. | <code>while(true)</code> { <code>arcadeControl();</code> } | <code>while(true)</code> { <code>arcadeControl(Ch1, Ch4);</code> } |

ROBOTC Debugger Overview

A “debugger” is a programming tool that enables you to quickly write and correct code, and allows you to interact with all of the inputs (sensors, timers, ect.) and outputs (motors, LED's, ect.) connected to your VEX microcontroller.

ROBOTC has a debugging capability that enables unparalleled, interactive access to the robot as your program is running. Using the debugger will significantly reduce the time it takes to write programs and find errors in your code. With ROBOTC’s real-time debugger you can:

- Start and stop your program from the computer
- “Single step” through your program, running one line of code at a time and examine the results (the values of variables, sensors, ect.) and the flow of execution
- Read and write the values of all the variables defined in your program
- Read and write the values of all the motors and sensors configured on your microcontroller



Note: Traditional Debugging Techniques

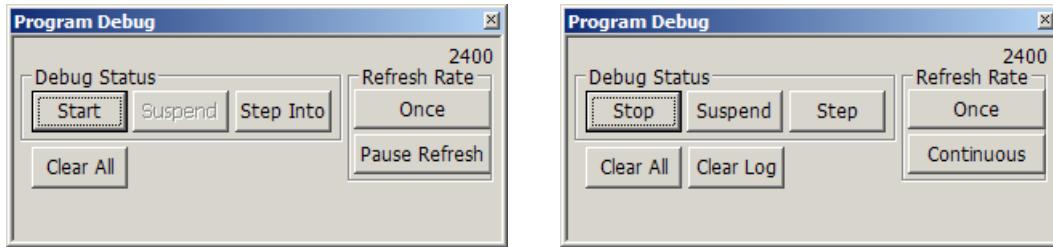
Debugging a program (finding the errors and correcting them) can be a slow process without a real-time debugger. Without a debugger you may have to resort to other techniques:

- Adding code to turn on different LED's as the microcontroller executes different sections of code. You then try to determine from the LED's what is being executed within your program.
- Adding “print” statements to your code at various points in the program, if your microcontroller has a display device. By examining the display, you can (hopefully) determine what is happening in your program.

Both of the above techniques are available in ROBOTC, but a real-time debugger eliminates the need to resort to them. There's no need to add code to debug your program!

ROBOTC Debugger Debug Window

The Program Debug window appears every time you download a program to your VEX microcontroller, and is in control of the connection between your computer and robot controller. Closing it will terminate the connection between your computer and the robot controller, along with any other open debug windows.



Start / Stop

Pressing the Start button will start the program execution on your robot controller, and the text on the button will change to “Stop”. Pressing the Stop button will stop the program execution.

Suspend

Pressing the Suspend button will suspend (pause) the program execution on your robot controller.

Step Into

Pressing the Step Into button will execute the next command in your program.

Clear All

The Clear All button will reset all of the values being displayed by the other debug windows.

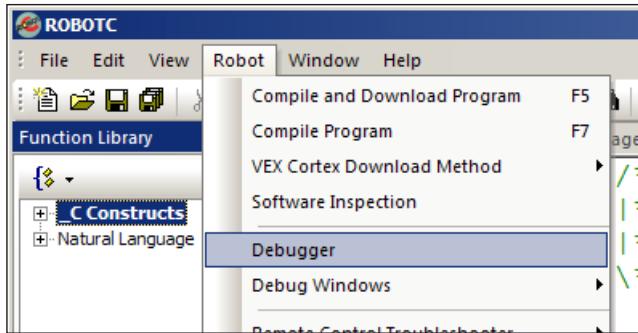
Once

Pressing the Once button will update the values in the other debugger windows once.

Pause Refresh / Continuous

Pressing the Pause Refresh button will cause the values in the debugger windows to stop updating. Pressing it will also cause the text to change to “Continuous”. Pressing the Continuous button will cause the values in the debugger windows to update continuously. Pressing it will also cause the text to change to “Pause Refresh”.

Note: For continuous value updates on the other debug windows, make sure the button says Pause Refresh, and not Continuous.



The recommended method of opening the Program Debug window, and establishing a connection with the robot is by downloading a program to the robot. However, the debugger can also be launched by selecting “Debugger” from the Robot menu in ROBOTC.

The ROBOTC Debugger Global Variables

The Global Variables window displays the current values of every variable declared in your program. Using the ROBOTC debugger, not only can you view the variable's names and values, you can also change their values in real-time. To change the value of one of the variables, select the Value box of the variable you'd like to change, type in the new value, and press Enter on your keyboard.

| Global Variables | | |
|------------------|-----------|-------|
| Index | Variable | Value |
| 149 | time | 10000 |
| 150 | threshold | 505 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Global Variables Timers Motors Sensors

Index

The index of the variable, in memory.

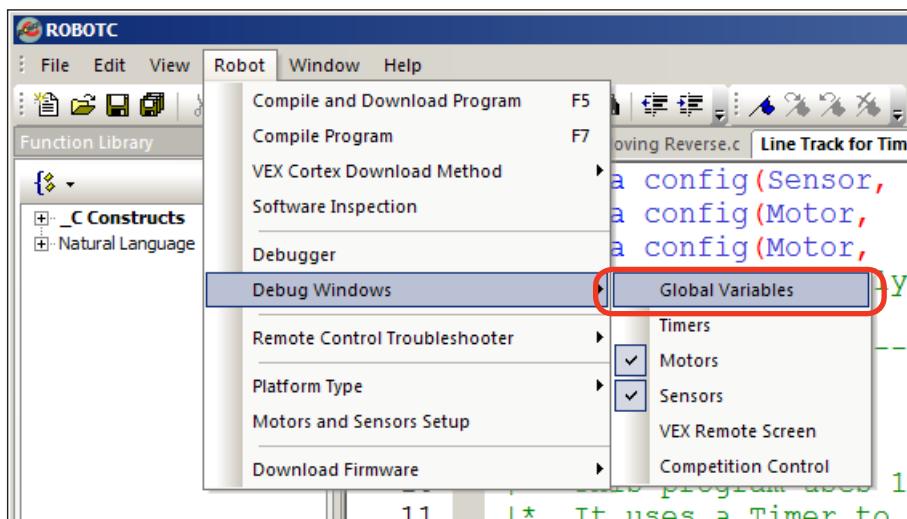
Variable

The name of the variable, defined in the program.

Value

The value of the variable during program execution. Values will update automatically if the Program Debug window is set to update continuously.

The Global Variables window can be opened by going to the Robot menu, Debug Windows, and selecting Global Variables.



The ROBOTC Debugger Timers

The Timers debug window provides access to current values of the timers built-in to your microcontroller. On the VEX Cortex, there are 4 user-accessible timers (T1, T2, T3, and T4), and two system timers (nSysTime and nPgmTime). The 4 user-accessible timers can be modified in real-time using the Timers debug window, but the two system timers cannot.

| Timers | | |
|--------|----------|--------------|
| Index | Timer | time |
| | nSysTime | 7.28.690 sec |
| | nPgmTime | 5.000 sec |
| T1 | Timer1 | 6.26.811 sec |
| T2 | Timer2 | 6.26.811 sec |
| T3 | Timer3 | 6.26.811 sec |
| T4 | Timer4 | 6.26.811 sec |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Index

The index of the timer (T1-T4).

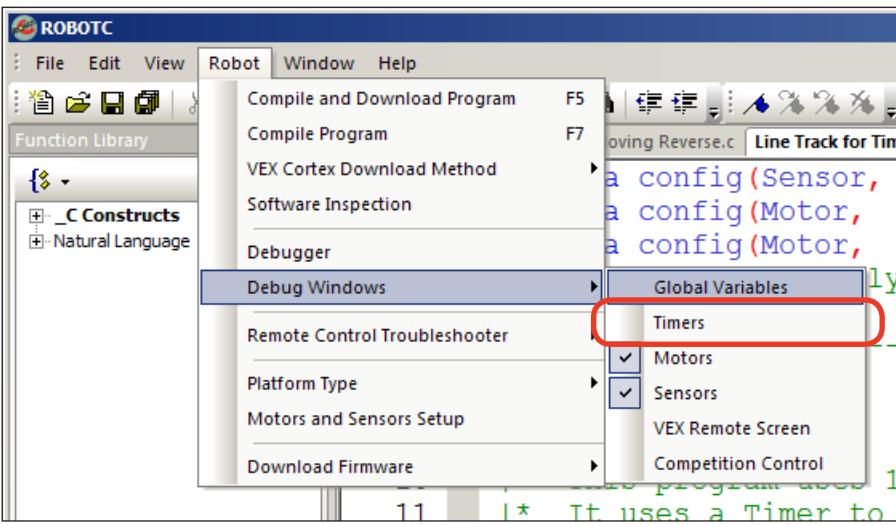
Timer

Name of the timer. “nSysTime” is the amount of time the controller has been powered on. “nPgmTime” is the amount of time the current program has run. Timer1 through Timer4 can be reset and monitored in your programs.

Time

Displays the elapsed time.

The Timers window can be opened by going to the Robot menu, Debug Windows, and selecting Timers.



The ROBOTC Debugger Motors

The Motors debug window provides access to the current values of the motors, servos and flashlights on your microcontroller. Motor, servo and flashlight power levels can be viewed and changed using this window.

| Motors | | |
|--------|--------|-------|
| Index | Motor | Power |
| port1 | port1 | 0 |
| port2 | port2 | 0 |
| port3 | port3 | 0 |
| port4 | port4 | 0 |
| port5 | port5 | 0 |
| port6 | port6 | 0 |
| port7 | port7 | 0 |
| port8 | port8 | 0 |
| port9 | port9 | 0 |
| port10 | port10 | 0 |

Global Variables Timers **Motors** Sensors

Index

The index of where the current device is located (port1-port10).

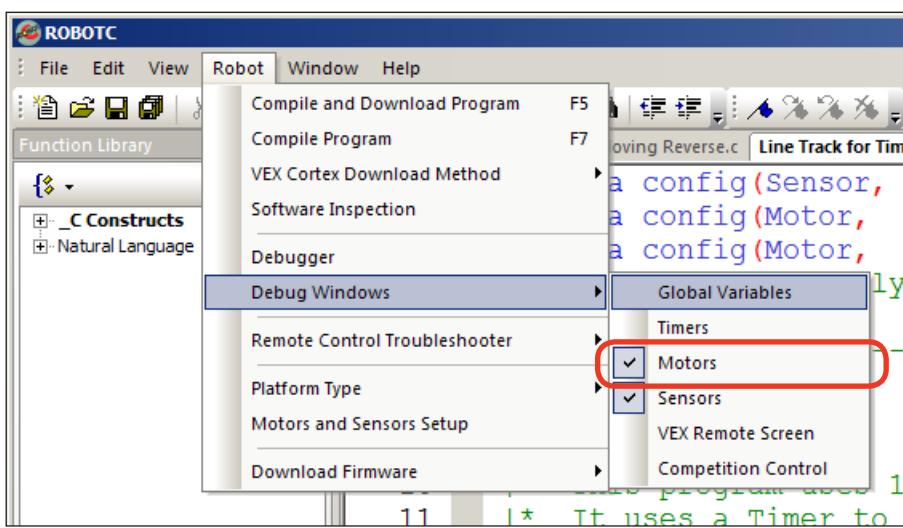
Motor

Current name of the motor. These names can be customized through the Motors and Sensor Setup window.

Value

Displays the current power level of the motor.

The Motors window can be opened by going to the Robot menu, Debug Windows, and selecting Motors.



The ROBOTC Debugger Sensors

The Sensors debug window provides access to the current values of all sensors, digital inputs and digital outputs configured on your microcontroller. Sensor values can be viewed and changed using this window, but you must first use the Motors and Sensors Setup menu to tell ROBOTC what types of sensors are connected to which ports. Different sensors are interpreted differently by ROBOTC and the microcontroller, and appropriate values will not be displayed if they are not properly configured.

| Sensors | | | |
|---------|--------|-------------------|-------|
| Index | Sensor | Type | Value |
| in1 | in1 | Line Follower | 2928 |
| in2 | in2 | Line Follower | 2963 |
| in3 | in3 | Line Follower | 2903 |
| in4 | in4 | No Sensor | 1668 |
| in5 | in5 | No Sensor | 125 |
| in6 | in6 | Potentiometer | 2160 |
| in7 | in7 | No Sensor | 2303 |
| in8 | in8 | No Sensor | 2067 |
| dgtl1 | dgtl1 | Quadrature Enc... | 0 |
| dgtl2 | dgtl2 | Quad Encoder 2... | 1 |
| dgtl3 | dgtl3 | Quadrature Enc... | 0 |
| dgtl4 | dgtl4 | Quad Encoder 2... | 1 |
| dgtl5 | dgtl5 | No Sensor | 1 |
| dgtl6 | dgtl6 | Touch | 0 |
| dgtl7 | dgtl7 | Touch | 0 |
| dgtl8 | dgtl8 | SONAR (cm) | 49 |
| dgtl9 | dgtl9 | SONAR 2nd Port | 0 |
| dgtl10 | dgtl10 | No Sensor | 1 |
| dgtl11 | dgtl11 | No Sensor | 1 |
| dgtl12 | dgtl12 | No Sensor | 1 |

Global Variables Timers Motors Sensors

Index

The index of where the current device is located (in1 - in8 for ANALOG Ports 1 - 8, and dgtl1 - dgtl12 for DIGITAL Ports 1 - 12).

Device

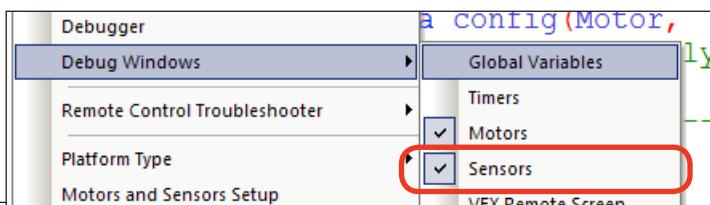
Current name of the sensor. These names can be customized through the Motors and Sensor Setup window.

Type

Displays the type of the current sensor. The type must be set using the Motors and Sensor Setup window.

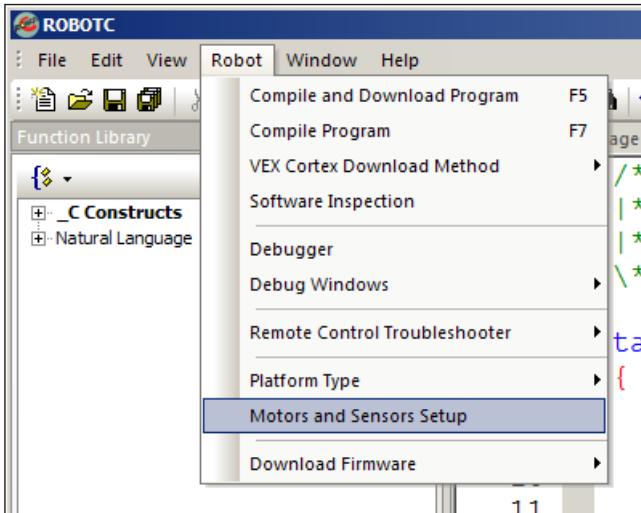
Value

Displays the current value of the sensor.



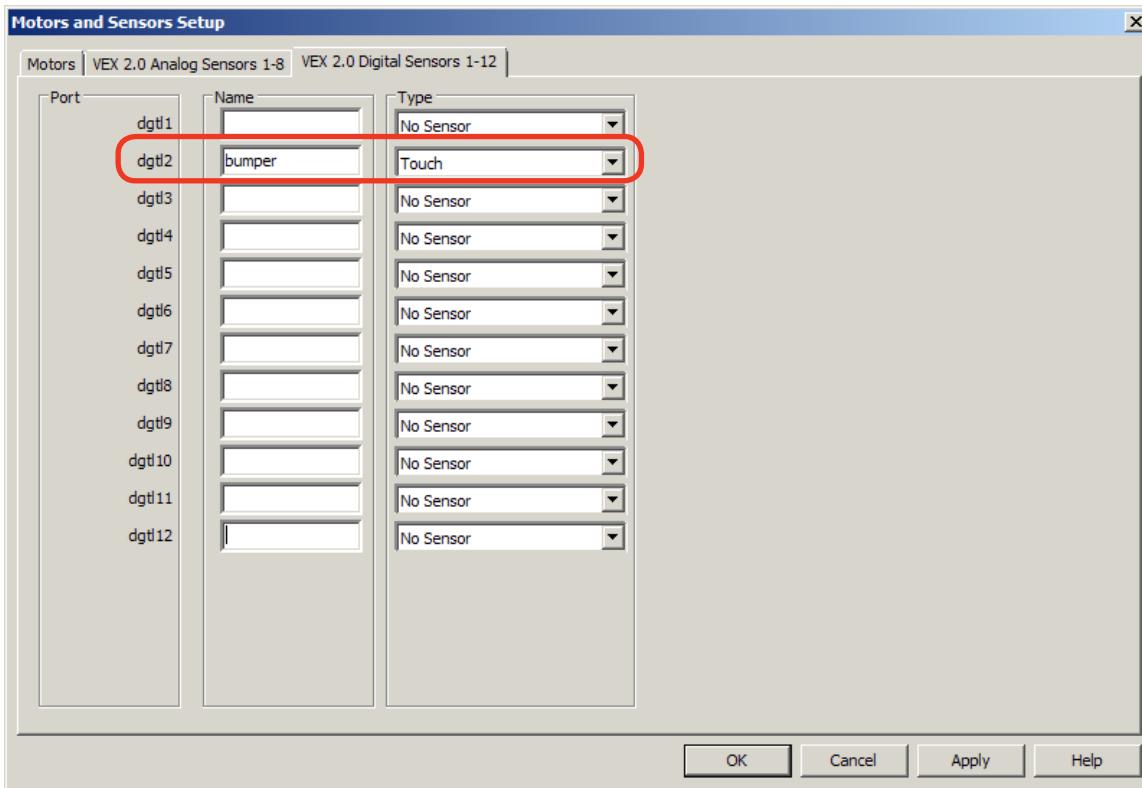
The Sensors window can be opened by going to the Robot menu, Debug Windows, and selecting Sensors.

The ROBOTC Debugger Sensors



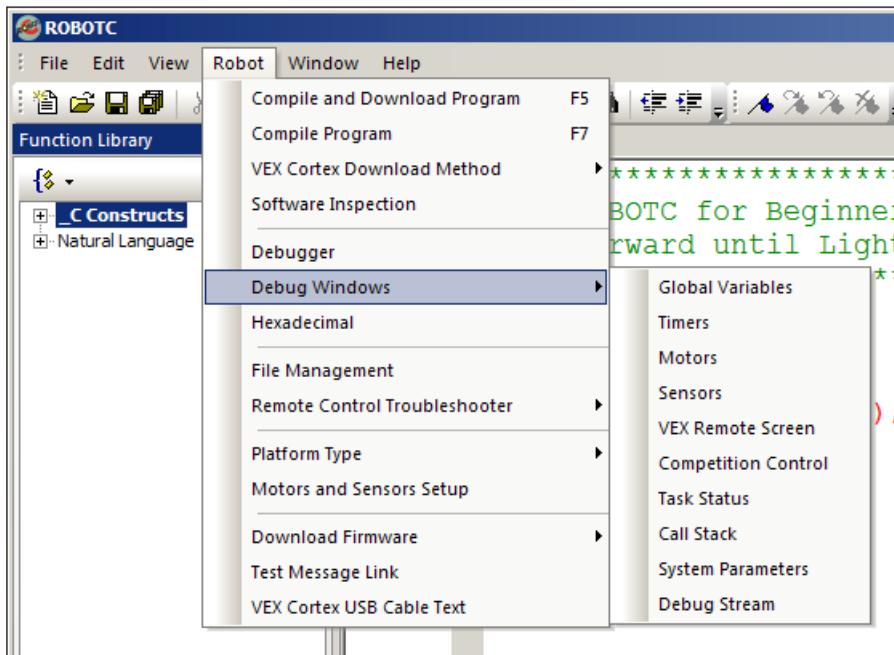
To configure the sensors connected to your microcontroller, open the Motors and Sensors Setup from the Robot menu in ROBOTC.

Digital sensors (Bumper, Limit, Encoder, Ultrasonic) can be configured on the “VEX 2.0 Digital Sensors 1-12” tab, and analog sensors (Light, Line Follower, Potentiometer, Accelerometer) can be configured on the “VEX 2.0 Analog Sensors 1-8” tab. To configure a sensor, first locate the row that aligns with where the sensor is plugged in on the Cortex. For example, “dgtl2” is short for DIGITAL Port 2 on the Cortex. Next, give the sensor a custom name and define its sensor type in the dropdown menu. After applying your changes, redownload your program to the microcontroller to have the changes take effect in the Sensor debug window.

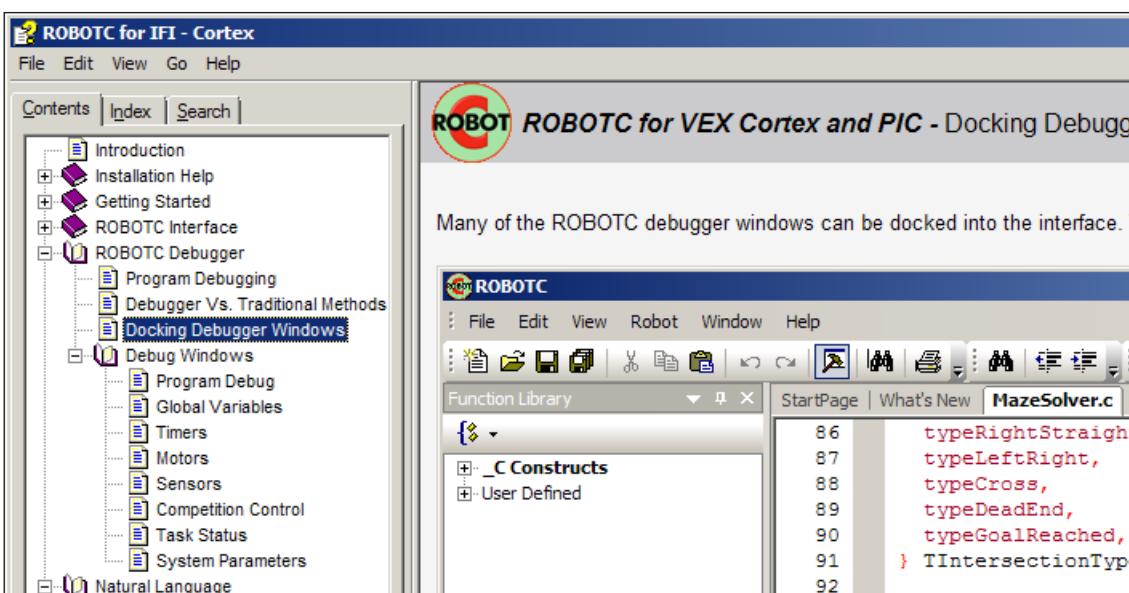


The ROBOTC Debugger Miscellaneous

There are several additional debug windows available in the “Expert” and “Super User” modes of ROBOTC. To unlock these windows, change your Menu Level, by going to Window > Menu Level, and selecting one of the other modes. The additional debug windows are very powerful, and can be very helpful in advanced applications.



For additional information on these debug windows, along with the ones covered in this document, view the ROBOTC Debugger section of the built-in ROBOTC Help documentation.



White Space with Natural Language

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 moves an arm up until a touch sensor is pressed, stops, waits for 2 seconds, and then moves down until the touch sensor is released could look like either of these:

Program Without White Space

```
task main()
{
    startMotor(armMotor, 63);
    untilTouch(bumper);
    stopMotor(armMotor);
    wait(2.0);
    startMotor(armMotor, 63);
    untilRelease(bumper);
    stopMotor(armMotor);
}
```

Program With White Space

```
task main()
{
    startMotor(armMotor, 63);
    untilTouch(bumper);
    stopMotor(armMotor);

    wait(2.0);

    startMotor(armMotor, 63);
    untilRelease(bumper);
    stopMotor(armMotor);
}
```

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 the arm up and moving the arm down. 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()
{
    while(true)
    {
        if(SensorValue(touch)==0)
        {
            startMotor(armMotor, 63);
        }
        else
        {
            startMotor(armMotor, -63);
        }
    }
}
```

Program With White Space

```
task main()
{
    while(true)
    {
        if(SensorValue(touch)==0)
        {
            startMotor(armMotor, 63);
        }
        else
        {
            startMotor(armMotor, -63);
        }
    }
}
```

Comments with Natural Language

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.

```
#pragma config(Sensor, dgtl1, bumper, sensorTouch)
#pragma config(Motor, port2, armMotor, tmotorNormal, openLoop)
//**!Code automatically generated by 'ROBOTC'!!*/

/*
    This is part of a multi-line comment.
    This program uses commenting to describe each process.
*/
task main()
{
    startMotor(armMotor, 63); //Turn armMotor on at 1/2 power
    untilTouch(bumper); //Wait for bumper switch to be touched
    stopMotor(armMotor); //Stop the armMotor
}
```

“Commenting out” Code

Commenting is also sometimes used to temporarily “disable” code in a program without actually deleting it. In the program below, the programmer has code to move an arm up and then move the arm down. However, in order to test only the second half of the program, the programmer made the first behavior into a comment, so the robot will ignore it. When the programmer is done testing the second behavior, he/she can remove the // comment marks to re-enable the first behavior in the program.

```
task main()
{
    //startMotor(armMotor, 63); //Turn armMotor on at 1/2 power
    //untilTouch(bumper); //Wait for bumper switch to be touched
    //stopMotor(armMotor); //Stop the armMotor

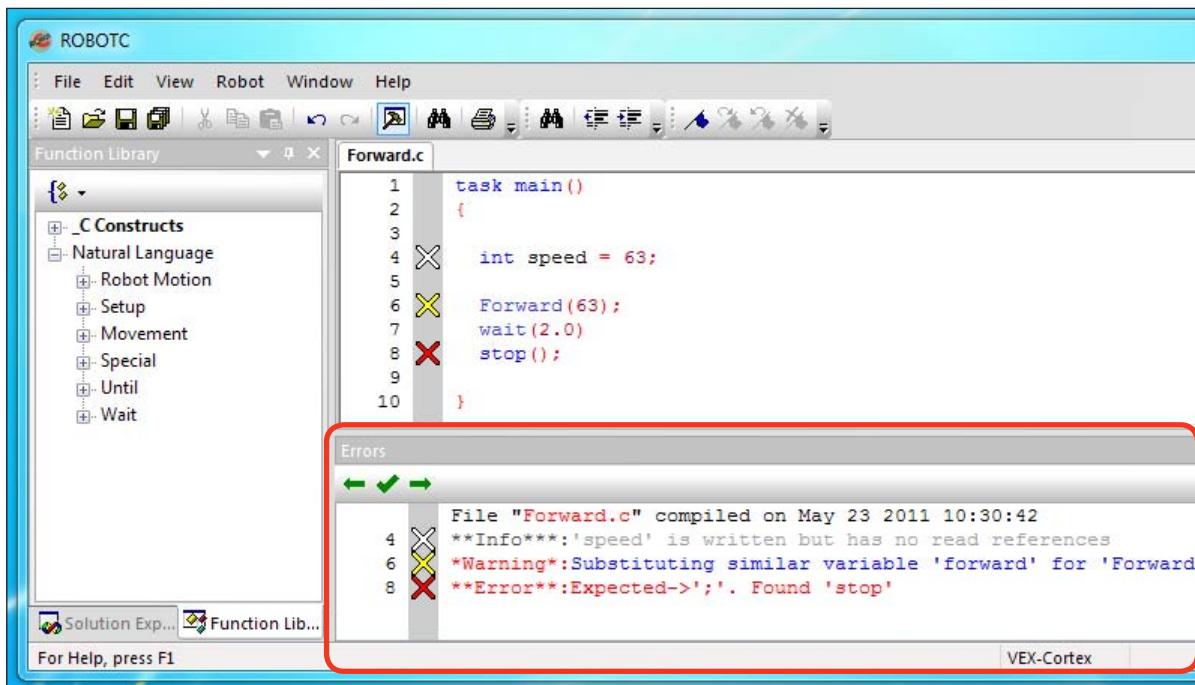
    wait(2.0); //Wait 2.0 seconds

    startMotor(armMotor, -63); //Turn armMotor on at -1/2 power
    untilRelease(bumper); //Wait for bumper switch to be released
    stopMotor(armMotor); //Stop the armMotor
}
```

Error Messages in ROBOTC Code

ROBOTC has a built-in compiler that analyzes your programs to identify syntax errors, capitalization and spelling mistakes, and code inefficiency (such as unused variables). The compiler runs every time you download code to the robot and when you choose to compile your program from the Robot menu in ROBOTC.

Notifications regarding any errors, warnings and important information the compiler finds are displayed in the Errors display screen of the ROBOTC interface.



The Errors display screen reports the number of errors in your code, as well as their types. Double-clicking a compiler message in the Error display screen will highlight the relevant line of code in your program. Depending on the type of error, ROBOTC will only be able to highlight the approximate location. For instance, in the example above the missing semicolon is on line 7 but ROBOTC will highlight line 8.

ROBOTC generates three types of compiler messages: Errors, Warnings and Information:

Errors:

There was an issue your program that prevented it from compiling. These are usually misspelled words, missing semicolons, and improper syntax. Errors are denoted with a **Red X**.

Warnings:

There was a minor issue with your program, but the compiler was able to fix or ignore it. These are usually incorrect capitalizations or empty, infinite loops. Warnings are denoted with a **Yellow X**.

Information:

ROBOTC will generate information messages when it thinks you have declared functions or variables that are not used in your program. These messages inform you about inefficient programming. Information messages are denoted with a **White X**.

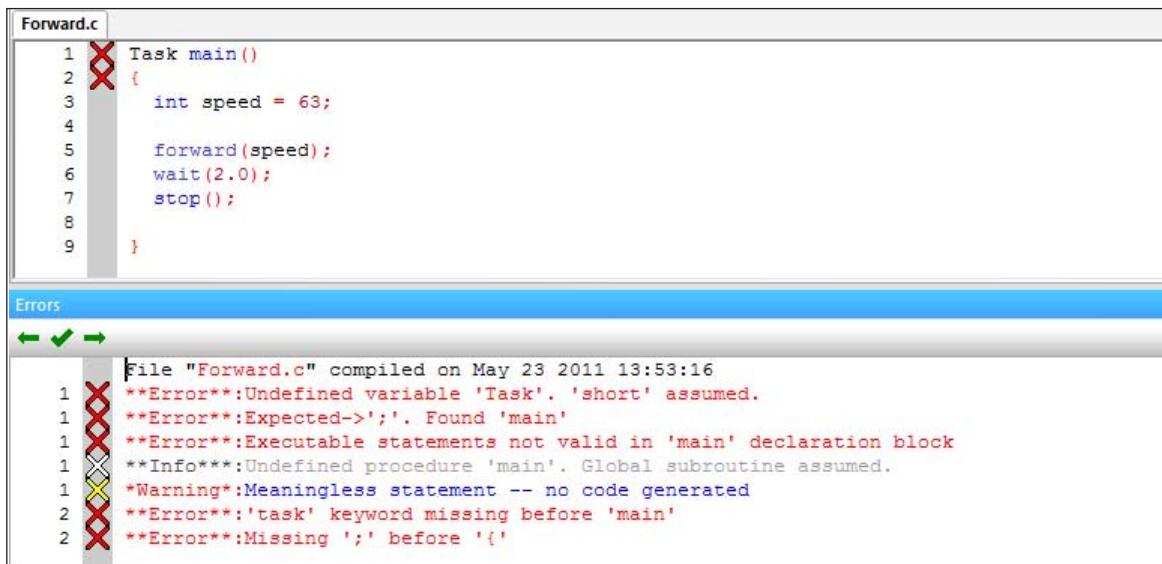
Error Messages in ROBOTC Code

Common **Error** Messages

Error messages will prevent your program from compiling and downloading to your robot. You must correct any and all error messages in your program before you will be able to download it to your robot. Also, error messages can have a “ripple” effect; errors at the beginning of your program can cause subsequent errors in the code. Because of this, it’s recommended that you correct errors at the beginning of your program first, recompile your code, and then correct any remaining errors.

Most error messages are caused by misspelled reserved words and improper syntax. Many of these mistakes can be avoided by dragging commands from the Function Library into your ROBOTC programs.

In the example below, the T in task main is capitalized, causing multiple subsequent errors and warnings to appear in the Errors display screen.



The screenshot shows the ROBOTC IDE interface. The top window is titled "Forward.c" and contains the following C code:

```
1 Task main()
2 {
3     int speed = 63;
4
5     forward(speed);
6     wait(2.0);
7     stop();
8
9 }
```

The bottom window is titled "Errors" and displays the following error messages:

```
File "Forward.c" compiled on May 23 2011 13:53:16
1 **Error**:Undefined variable 'Task'. 'short' assumed.
1 **Error**:Expected->'('; Found 'main'
1 **Error**:Executable statements not valid in 'main' declaration block
1 **Info**:Undefined procedure 'main'. Global subroutine assumed.
1 *Warning*:Meaningless statement -- no code generated
2 **Error**:'task' keyword missing before 'main'
2 **Error**:Missing ';' before '{'
```

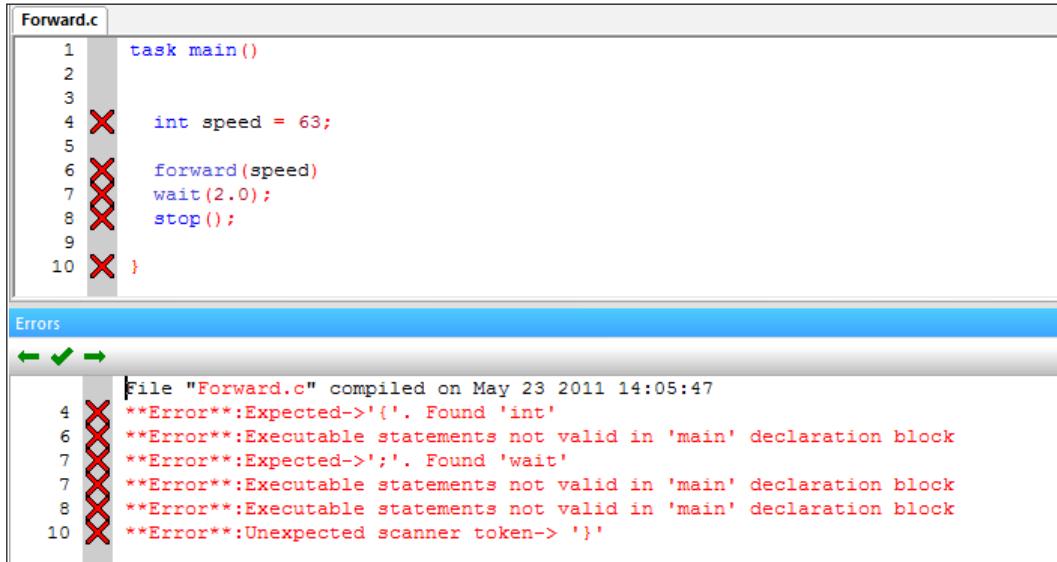
In situations like these, it’s recommended that you try to correct the first error in your program before moving on. The first error message reads “**Error**:Undefined variable ‘Task’. ‘short’ assumed.” When the words “Undefined variable” appear in the Errors display screen, it indicates that ROBOTC does not recognize the specified word; the fact that Task is colored black instead of blue like other ROBOTC reserved words also indicates that ROBOTC does not recognize it.

To correct this error, you should replace the uppercase T with a lowercase t in task, and recompile your code. The compiler will reevaluate your code and generate a new set of notifications.

Error Messages in ROBOTC Code

Common **Error** Messages

The example below contains two syntax errors: a missing curly brace on line 2 and a missing semicolon on line 6. Once again, you should try to correct the first error in the program before moving on.



The screenshot shows the ROBOTC IDE interface. The code editor window displays the following C code:

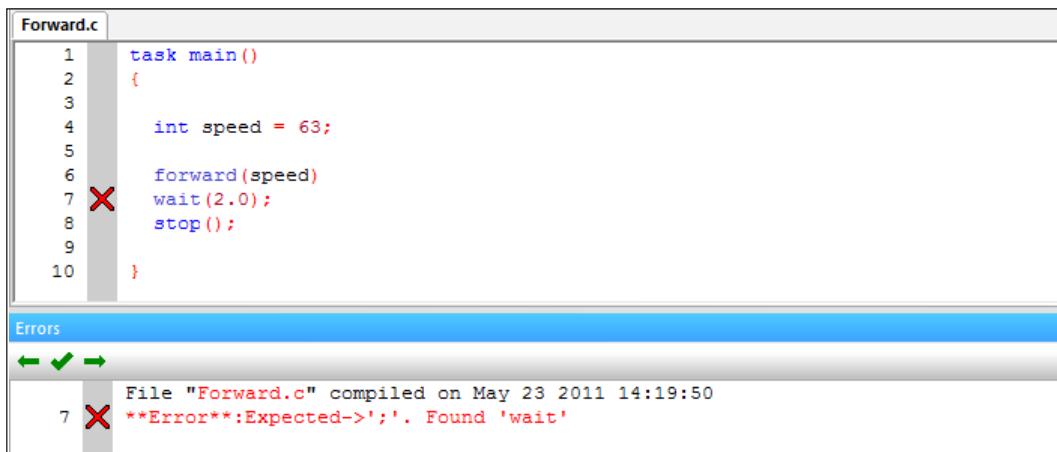
```

Forward.c
1 task main()
2
3
4 int speed = 63;
5
6 forward(speed)
7 wait(2.0);
8 stop();
9
10 }
  
```

The code editor highlights several errors with red X marks: line 2 (missing opening brace), line 6 (missing semicolon), line 7 (missing semicolon), line 8 (missing semicolon), and line 10 (missing closing brace). The Errors window below lists the following error messages:

- File "Forward.c" compiled on May 23 2011 14:05:47
- 4 **Error**:Expected->'{'. Found 'int'
- 6 **Error**:Executable statements not valid in 'main' declaration block
- 7 **Error**:Expected->;'. Found 'wait'
- 7 **Error**:Executable statements not valid in 'main' declaration block
- 8 **Error**:Executable statements not valid in 'main' declaration block
- 10 **Error**:Unexpected scanner token-> '}'

The first error message comes up on line 4, saying “**Error**:Expected->'{. Found ‘int’”. When the word “Expected->” appears in the Errors display screen, it usually indicates that a piece of syntax is missing. In this case, it expected to find the missing curly brace immediately after task main(), but found the reserved word int instead. To correct this error, you should add the opening curly brace on line 2 and then recompile your code.



The screenshot shows the ROBOTC IDE interface. The code editor window displays the following C code:

```

Forward.c
1 task main()
2 {
3
4     int speed = 63;
5
6     forward(speed)
7 wait(2.0);
8 stop();
9
10 }
  
```

The code editor highlights the missing semicolon on line 7. The Errors window lists the following error message:

- File "Forward.c" compiled on May 23 2011 14:19:50
- 7 **Error**:Expected->;'. Found 'wait'

After recompiling your code, any remaining errors will be displayed. Missing semicolons also display an “Expected->” style error message, but notice that the error for the missing semicolon on line 6 appears on line 7. This is because the compiler ignores whitespace (blank lines, spaces and tabs), but expected a semicolon before it encountered the wait command. To correct this error, you should add a semicolon after the forward command.

Error Messages in ROBOTC Code

Common Error Messages

The example below, the ROBOTC compiler does not recognize the forward, wait or stop commands. Error messages that begin with “**Error**:Undefined procedure” indicate that ROBOTC does not recognize the command; this is also indicated by the commands failing to turn blue like other ROBOTC reserved words.



```

Forward.c
1 task main()
2 {
3
4     int speed = 63;
5
6     forward(speed);
7     wait(2.0);
8     stop();
9
10 }

```

Errors

File "Forward.c" compiled on May 23 2011 14:29:52

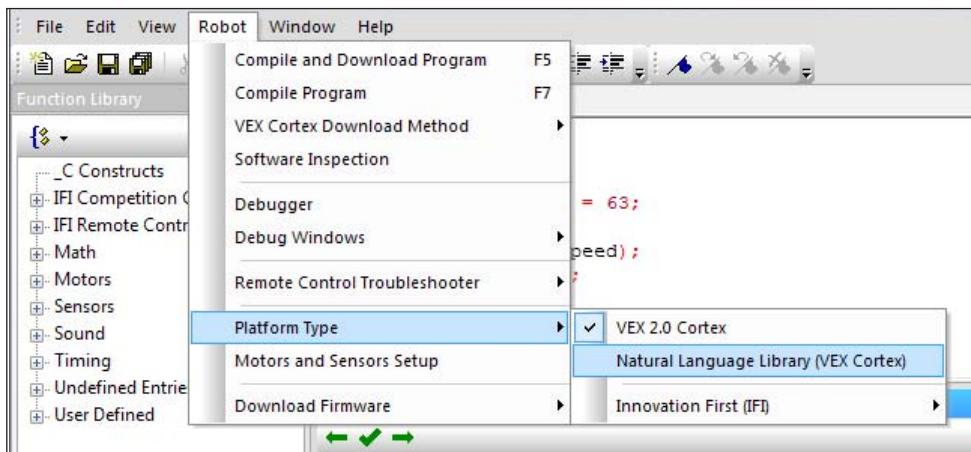
6 **Error**:Undefined procedure 'forward'.
 7 **Error**:Undefined procedure 'wait'.
 8 **Info**:**Undefined procedure 'stop'. Global subroutine assumed.

There are two main causes for this error:

1. The command is misspelled
2. The correct ROBOTC Platform Type is not selected.

To correct this error:

1. Verify that your spelling and capitalization is correct.
2. Verify that you have the appropriate Platform Type selected. If you are using Natural Language commands, you must have the Natural Language Platform Type selected.



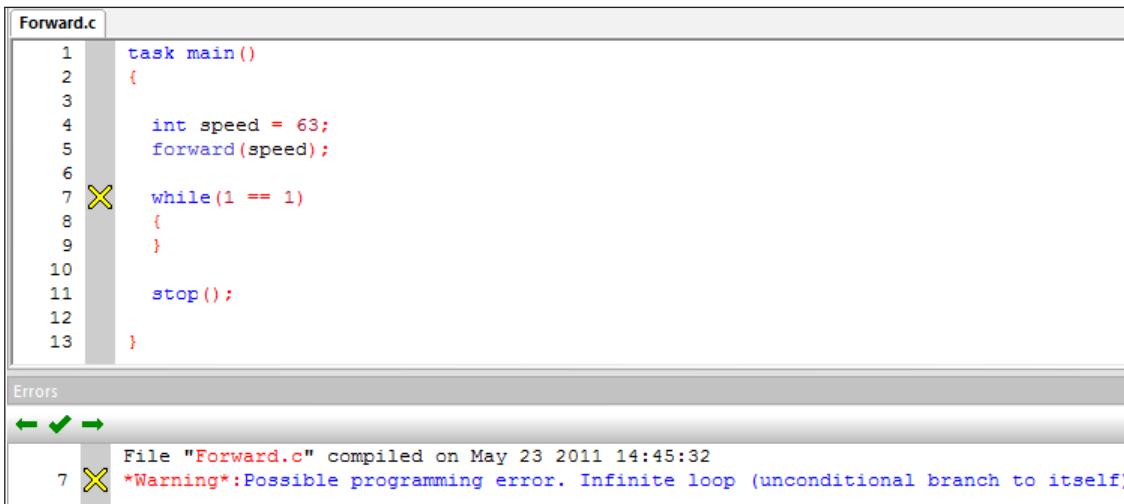
Error Messages in ROBOTC Code

Common **Warning** Messages

Warning messages are used to notify you about possible programming and logic errors in your program. With warning messages, the compiler is able to fix or ignore the issues so they will not prevent your program from compiling or downloading to your robot.

A common occurrence of warning messages are empty, infinite loops in your code. In the example below, the an infinite loop is created, with no code embedded within the loop to stop it from repeating forever. This is considered a warning, rather than an error, because it is valid code and can be intentionally used by a programmer.

The warning message will inform you that there was a possible programming error caused by an infinite loop: “*Warning*:Possible programming error. Infinite loop (unconditional branch to itself) detected.”



The screenshot shows a code editor window titled "Forward.c". The code is as follows:

```
1 task main()
2 {
3
4     int speed = 63;
5     forward(speed);
6
7     while(1 == 1)
8     {
9     }
10
11     stop();
12
13 }
```

Line 7 is highlighted with a yellow X icon. Below the code editor is a "Errors" panel. It shows a green checkmark icon followed by a yellow X icon. The text in the panel reads:

File "Forward.c" compiled on May 23 2011 14:45:32
7 ***Warning*:Possible programming error. Infinite loop (unconditional branch to itself)**

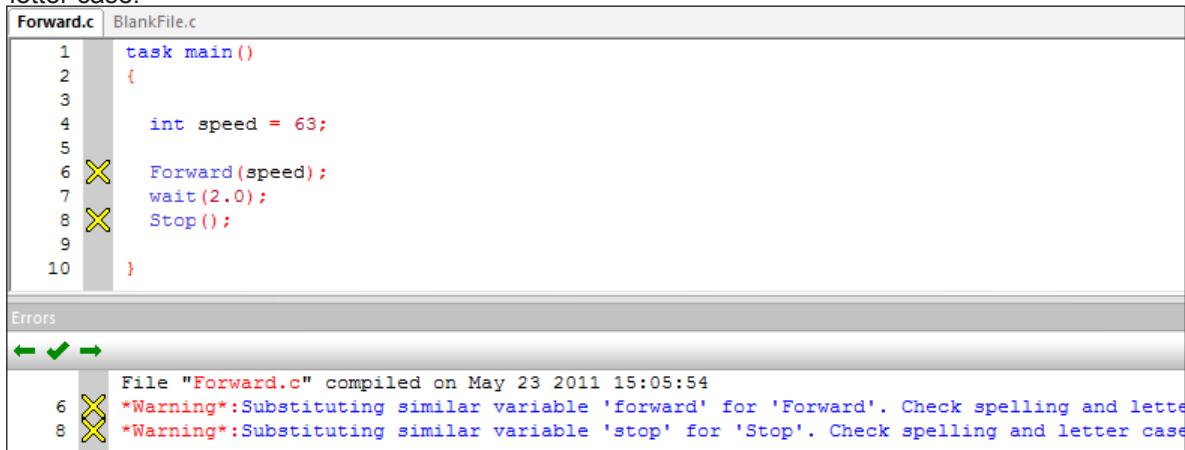
If creating an infinite loop was your intention, you do not need to correct this message. If it was not intentional, you can:

1. Include code within the curly braces of the while loop, for it to repeat.
2. Change the condition of the while loop, so that it does not repeat forever.

Error Messages in ROBOTC Code

Common **Warning** Messages

In the example below, the forward and stop commands are improperly capitalized. The ROBOTC compiler is able to substitute in the correct forms of the commands, but uses warning messages to notify you of the substitution. Note that it does not correct the capitalization in your code, only what it sends to the robot. Again, the actual warning message inform you about the substitution: “*Warning*:Substituting similar variable ‘forward’ for ‘Forward’. Check spelling and letter case.”



The screenshot shows the ROBOTC IDE interface. The code editor window is titled "Forward.c" and contains the following C code:

```
task main()
{
    int speed = 63;
    Forward(speed);
    wait(2.0);
    Stop();
}
```

The errors panel at the bottom shows two warning messages:

- File "Forward.c" compiled on May 23 2011 15:05:54
- *Warning*:Substituting similar variable 'forward' for 'Forward'. Check spelling and letter case
- *Warning*:Substituting similar variable 'stop' for 'Stop'. Check spelling and letter case

You do not need to correct this message. If it was not intentional, you can:

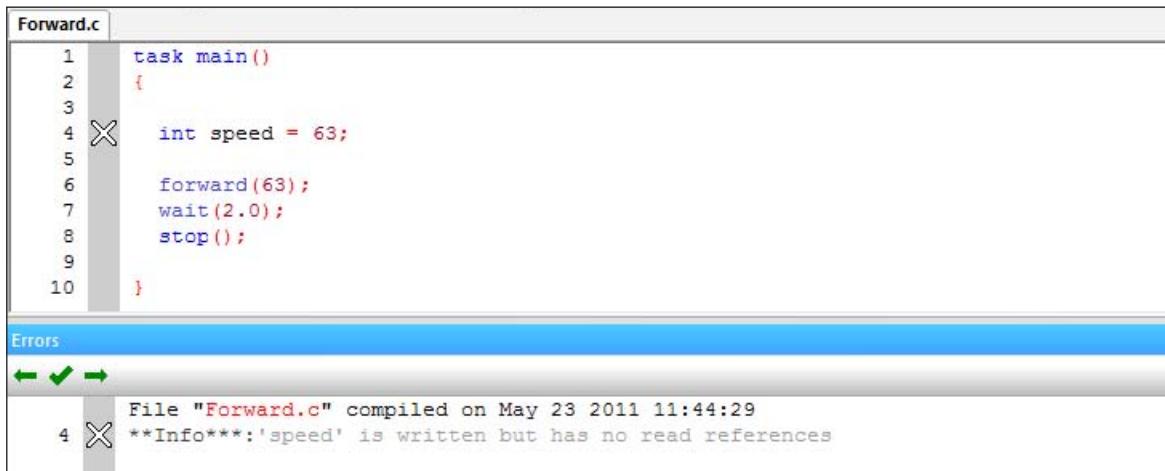
1. Include code within the curly braces of the while loop, for it to repeat.
2. Change the condition of the while loop, so that it does not repeat forever.

Error Messages in ROBOTC Code

Common Information Messages

Information messages will not prevent your program from compiling or downloading to your robot. They only notify you regarding possible inefficiencies in your code.

The most common occurrence of information messages are unused variables in your code. In the example below, the integer variable speed is created and initialized, but never actually used in the program. This is indicated in the Errors display screen with the message: “speed” is written but has no read references”.



```
Forward.c
1 task main()
2 {
3
4 X int speed = 63;
5
6 forward(63);
7 wait(2.0);
8 stop();
9
10 }
```

Errors

File "Forward.c" compiled on May 23 2011 11:44:29
4 X **Info***:'speed' is written but has no read references

You do not need to correct this message, but you can in two ways:

1. Eliminate the code on line number 4, deleting the variable.
2. Call the speed variable in the forward command on line 6, in place of the integer 63.

Handling other Error Messages

Learning how to program robots isn’t easy; it’s no different than learning a foreign language. This document covers how to handle some of the more common mistakes and error messages that you may encounter, but you may run into others. That said, here are some general rules for dealing with all error messages:

- Determine if the message is an error, a warning, or just information. The message may not even require additional work on your part!
- Read the error message for clues! Error messages aren’t always the most intuitive, but they always contain some information about what the compiler found.
- When your program contains multiple errors, fix them one at a time, recompiling your code after each fix. The “ripple” effect can make it seem like there are errors even if the rest of your program is perfect.
- Pseudocode, pseudocode, pseudocode! Make sure you have a plan in place before you try to write a complex program. That way you can work out the logic first, without having to worry about it and the syntax, spelling and capitalization at the same time.

Troubleshooting ROBOTC with Cortex

This guide is designed to be used by a student or teacher as a reference for help troubleshooting ROBOTC software issues.

Troubleshooting Topics

- Computer will not Recognize the VEX Cortex
- Not able to Download my ROBOTC program over USB
- Not able to Download ROBOTC Firmware over USB
- Not able to Download Master CPU Firmware over USB
- Program will not Compile
- Program compiles, but does not behave as desired
- Not able to open the ROBOTC Debugger
- Motors and/or Sensors Debug windows not functioning correctly
- Program does not immediately run when Cortex is turned on

Problem: Computer will not Recognize the VEX Cortex

1. Was the correct startup sequence followed when connecting the Cortex to the computer?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On

2. Is the connected battery sufficiently charged?

- Swap in a fully charged battery

3. Does the Cortex need to be power cycled?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On

4. Try another USB port on the computer.

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On

5. Try putting the Cortex into “Bootload” mode

- Start with the Cortex turned OFF (but with a battery connected)
- Push and hold Config button in on the Cortex.
- Attach the USB cable between the PC and Cortex.
- Wait for the Robot, VEXnet and Game lights to blink green.
- Release Config button.
- Turn the Cortex ON.
- Then in ROBOTC, click “Robot”, “Download Firmware”, “Master CPU Firmware”, “Standard File”.
- After the Master CPU Firmware finishes downloading, click “Robot”, “Download Firmware”, “ROBOTC

Firmware”, “Standard File”.

Additional Note: This step may be necessary if the Master CPU Firmware was corrupted and/or only partially downloaded to the Cortex. Also possible, if the wrong firmware (VEXnet Joystick) was downloaded to the Cortex, this step may be necessary.

Troubleshooting ROBOTC with Cortex

6. Make sure that your computer allows for “new hardware” to be connected. Extremely locked down computers may prohibit new hardware such as the VEX Cortex from being connected.

- Contact your Tech Support for additional privileges

7. Some SmartBoard software causes a conflict with the Cortex. The SMART Virtual TabletPC device can be disabled to resolve the conflict.



Troubleshooting ROBOTC with Cortex

Problem: Not able to Download my ROBOTC program over USB

1. Was the correct startup sequence followed when connecting the Cortex to the computer?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the program

2. Does the program compile?

- Fix any errors (red x's)
- Retry downloading the program

3. Is the correct Platform Type Selected?

- Verify that the correct platform type is selected under Robot > Platform Type
- Retry downloading the program

4. Is the connected battery sufficiently charged?

- Swap in a fully charged battery
- Retry downloading the program

5. Does the Cortex need to be power cycled?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the program

6. Have the Master CPU and ROBOTC Firmware been downloaded successfully to the Cortex?

- Do they need to be re-downloaded?
 - Download the Master CPU Firmware.....
 - Download the ROBOTC Firmware
 - Power Cycle the Cortex
 - Retry downloading the program

7. Try another USB port on the computer.

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the program

8. Restart ROBOTC.

- Close ROBOTC
- Open ROBOTC
- Retry downloading the program

9. Restart the computer.

- Restart your computer
- Open ROBOTC
- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the program

Troubleshooting ROBOTC with Cortex

10. Additional Steps

- Try using the same program on another computer, with the same Cortex.
- Try using the same program on the same computer, with a different Cortex.
- Try using a different USB A-to-A cable.

Problem: Not able to Download ROBOTC Firmware over USB

1. Was the correct startup sequence followed when connecting the Cortex to the computer?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the ROBOTC Firmware

2. Has the Master CPU Firmware been successfully downloaded?

- Download the Master CPU Firmware
- Retry downloading the ROBOTC Firmware

3. Is the correct Platform Type Selected?

- Verify that the correct platform type is selected under Robot > Platform Type
- Retry downloading the ROBOTC Firmware

4. Is the connected battery sufficiently charged?

- Swap in a fully charged battery
- Retry downloading the ROBOTC Firmware

5. Does the Cortex need to be power cycled?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the ROBOTC Firmware

6. Try another USB port on the computer.

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the ROBOTC Firmware

7. Restart ROBOTC.

- Close ROBOTC
- Open ROBOTC
- Retry downloading the ROBOTC Firmware

8. Restart the computer.

- Restart your computer
- Open ROBOTC
- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the ROBOTC Firmware

Troubleshooting ROBOTC with Cortex

9. Additional Steps

- Try downloading the ROBOTC Firmware using another computer, with the same Cortex.
- Try downloading the ROBOTC Firmware using the same computer, with a different Cortex.
- Try using a different USB A-to-A cable.

10. Slow down the firmware download by inserting delays.

- Go to Window > Menu level > and select Super User
- Go to View > Preferences > Detailed Preferences...
- Go to the VEX Cortex Tab
- - The box next to “Delay Between HID Write” allows you to specify a number of milliseconds to insert as delays.
- Add a 5 millisecond delay
- Retry downloading the ROBOTC Firmware
- Continue to add short delays up until 100 milliseconds.
- Retry downloading the Master CPU firmware until success.

Problem: Not able to Download Master CPU Firmware over USB

1. Was the correct startup sequence followed when connecting the Cortex to the computer?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the Master CPU Firmware

2. Is the correct Platform Type Selected?

- Verify that the correct platform type is selected under Robot > Platform Type
- Retry downloading the Master CPU Firmware

3. Is the connected battery sufficiently charged?

- Swap in a fully charged battery
- Retry downloading the Master CPU Firmware

4. Does the Cortex need to be power cycled?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the Master CPU Firmware

5. Try another USB port on the computer.

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the Master CPU Firmware

6. Restart ROBOTC.

- Close ROBOTC
- Open ROBOTC
- Retry downloading the Master CPU Firmware

7. Restart the computer.

Troubleshooting ROBOTC with Cortex

- Restart your computer
- Open ROBOTC
- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Retry downloading the Master CPU Firmware

8. Additional Steps

- Try downloading the Master CPU Firmware using another computer, with the same Cortex.
- Try downloading the Master CPU Firmware using the same computer, with a different Cortex.
- Try using a different USB A-to-A cable.

9. Slow down the firmware download by inserting delays.

- Go to Window > Menu level > and select Super User
- Go to View > Preferences > Detailed Preferences...
- Go to the VEX Cortex Tab
- - The box next to “Delay Between HID Write” allows you to specify a number of milliseconds to insert as delays.
- Add a 5 millisecond delay
- Retry downloading the ROBOTC Firmware
- Continue to add short delays up until 100 milliseconds.
- Retry downloading the Master CPU firmware until success.

10. Try the download using the VEXnet Firmware Upgrade Utility, supplied by VEX Robotics.

- Download the VEXnet Firmware Upgrade Utility, available here:
- http://www.vexforum.com/wiki/index.php/Software_Downloads
- Unzip the utility and instructions
- Follow the instructions included with the utility to update the firmware

Problem: Program will not Compile

1. If you're using Natural Language functions, make sure you're in the Natural Language platform type.

- Go to Robot > Platform Type
- Select the Natural Language option
- Go to Robot > Compile Program to recompile your code

2. Is the correct Platform Type Selected?

- Verify that the correct platform type is selected under Robot > Platform Type
- Retry downloading the program

3. Check your code for mistakes.

- Are you missing any curly braces?
- Are you missing any semicolons?
- Are any of your commands or variables improperly capitalized?
- Do any of your commands or variables contain typos?
- Go to Robot > Compile Program to recompile your code

Troubleshooting ROBOTC with Cortex

4. Check the ROBOTC Errors window for hints.
 - The ROBOTC Errors window (usually located at the bottom of the screen) will display a list of known errors, what line they're on, and some information about the error.
 - Double-click errors in the ROBOTC Errors window to highlight the affected line in your program.
 - Correct errors (keep in mind yellow x's are only warnings, and white x's are only information, not errors).
 - Go to Robot > Compile Program to recompile your code.
5. Compare your code to programs you know work and compile.
 - Compare your code to similar ROBOTC Sample Programs (File > Open Sample Program)

Problem: Program compiles, but does not behave as desired

1. Compare your code to programs you know work and compile.
 - Compare your code to similar ROBOTC Sample Programs (File > Open Sample Program)
2. Think "like the robot". The robot does exactly what you tell it to do. Nothing more and nothing less.
 - Is there an important step you're forgetting to tell the robot?
 - Go back to your pseudocode plan. Does the sequence of steps make sense for the robot?
3. Use the ROBOTC Program Debug window to "Step" through your code line-by-line.
 - Download the program to the robot.
 - When the Program Debug window appears, repeatedly press the "Step" button to run the program line-by-line.
 - Observe the robot's behavior. How does it compare to the desired behavior?
 - Try to identify where the robot's behavior differs from the desired behavior.
4. Insert visual "flags" in your program.
 - Insert optional wait statements or turn on different LED's at different parts of the program.
 - When the robot reaches one of the wait statements or LED's, and behaved correctly, you know that the program was correct up until that point.
 - Continue to insert and observe these optional flags until you identify the problem.

Problem: Not able to open the ROBOTC Debugger

1. Was the correct startup sequence followed when connecting the Cortex to the computer?
 - Start with the Cortex Turned OFF
 - Connect the Cortex to the computer over USB
 - Turn the Cortex On
 - Retry downloading the program to open the debugger
2. Has the program been downloaded to the Cortex?
 - Download the program to open the debugger
3. Have the Master CPU and ROBOTC Firmware been downloaded successfully to the Cortex?
 - Do they need to be re-downloaded?
 - Download the Master CPU Firmware
 - Download the ROBOTC Firmware
 - Power Cycle the Cortex
 - Retry downloading the program to open the debugger

Troubleshooting ROBOTC with Cortex

4. Check the VEX Cortex Download Method.

- Verify that the Robot > VEX Cortex Download Method is set to “Download using VEXnet or USB” or “Download using USB Only”
- If it was set for Competition:
 - Download the program
 - Turn Cortex OFF
 - Disconnect the Cortex from the computer
 - Reconnect the Cortex to the computer over USB
 - Turn the Cortex On
 - Retry downloading the program to open the debugger

Problem: Motors and/or Sensors Debug windows not functioning correctly

1. Is the Cortex turned on and connected to a charged battery?

- Swap in a fully charged battery
- Turn the Cortex on
- Observe the debug window data

2. Has a program that correctly configures the sensors been downloaded to the robot?

- Download a program that correctly configures the sensors on the robot.
- Run the program
- Stop the program
- Observe the sensor data

3. Is the Program Debug window set to provide “Continuous” updates?

- Download the program to the robot
- Under the “Refresh Rate” section, verify that a button is not labeled “Continuous”
- If a button is labeled “Continuous”, press it to receive continuous updates
- Observe the sensor data

4. Have the Master CPU and ROBOTC Firmware been downloaded successfully to the Cortex?

- Do they need to be re-downloaded?
- Download the Master CPU Firmware
- Download the ROBOTC Firmware
- Power Cycle the Cortex
- Re-download the program
- Observe the sensor data

5. Does the Cortex need to be power cycled?

- Start with the Cortex Turned OFF
- Connect the Cortex to the computer over USB
- Turn the Cortex On
- Re-download the program
- Observe the sensor data

6. Restart ROBOTC.

- Close ROBOTC
- Open ROBOTC
- Re-download the program
- Observe the sensor data

Troubleshooting ROBOTC with Cortex

Problem: Program does not immediately run when Cortex is turned on

1. Check the VEX Cortex Download Method.

- Go to Robot > VEX Cortex Download Method
- Select “Download using USB Only” to have the program start automatically
- Download the program
- Turn Cortex OFF
- Disconnect the Cortex from the computer
- Turn the Cortex On
- Program should start automatically

2-Wire Motor 269

The 2-Wire Motor 269 replaces the 3-Wire Motor as the standard

- VEX motor. All of the internal gears are made from a steel alloy, which means the clutches and replacement gears are no longer required. The 2-wire motor can be directly connected to the Cortex and ARM9 microcontrollers' internal motor controllers. An external motor control module is required to connect the 2-wire motor to the PIC Microcontroller V0.5. External motor control modules can also be used with the Cortex and ARM 9 microcontrollers.

INSERT THIS PAGE
at the **back of the**
Motion Chapter in your
VEX Inventor's Guide.

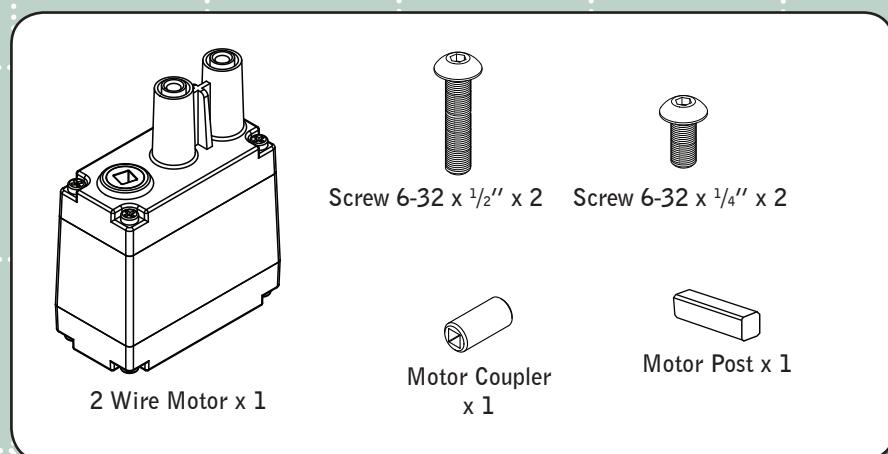
Motor Coupler

The 2-Wire Motor 269 kit includes the new shaft coupler, which can be used in place of the clutch to connect the motor to VEX shafts. The coupler can also be used to connect VEX shafts together.

Motor Specifications

All motor specifications are at 7.2 volts. Actual motor specifications are within 20% of the values below.

| Description | Specification |
|---------------|----------------------|
| Stall Torque | 8.6 in-lb [0.97 N-m] |
| Free Speed | 100 RPM |
| Stall Current | 2.6 Amps |
| Free Current | 0.18 Amps |



Limited 90-day Warranty

This product is warranted by Innovation First against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation First dealers. For complete warranty details and exclusions, check with your dealer.

Innovation First, Inc.
1519 IH 30 W
Greenville, TX 75402

03/10

For More Information, and additional Parts & Pieces refer to:
www.VEXrobotics.com

2 Wire Motor 393

The 2 Wire Motor 393 provides up to 60% more torque than the standard motor, which will allow more powerful mechanisms and drive bases. All of the internal gears are made from a steel alloy, which means that clutches and replacement gears are no longer required.

The 2 wire motor can be directly connected to the Cortex and ARM 9 microcontrollers' internal motor controllers. An external motor control module is required to connect the 2 wire motor to the PIC Microcontroller V0.5. External motor control modules can also be used with the Cortex and ARM 9 microcontrollers.

INSERT THIS PAGE
at the **back of the**
Motion Chapter in your
VEX Inventor's Guide.

High Speed Option

Want to go faster than the standard motor but still have the same output torque as the standard motor? No problem! The 2 Wire Motor 393 kit can be configured into a "high speed" version. Simply follow the "Gear Change Procedure" step-by-step instructions to increase the output speed by 60%.

Motor Coupler

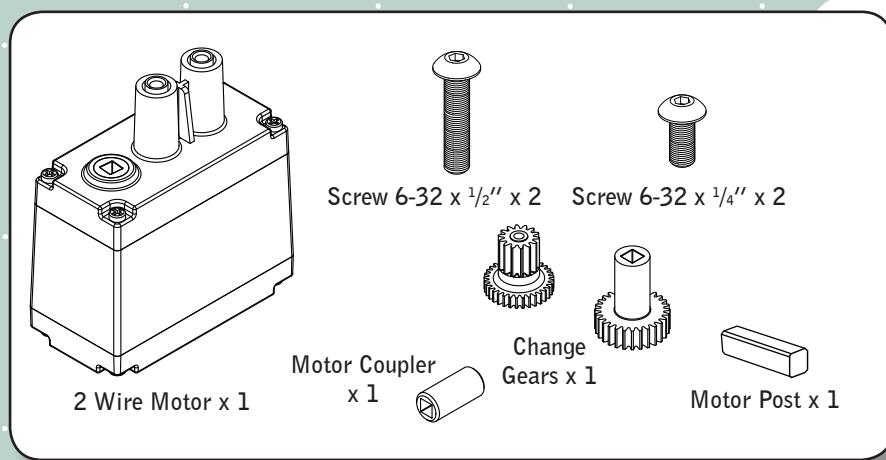
The 2 Wire Motor 393 Kit includes the new shaft coupler which can be used in place of the clutch to connect the motor to VEX shafts. The coupler can also be used to connect VEX shafts together.

Motor Specifications

All motor specifications are at 7.2 volts. Actual motor specifications are within 20% of the values below.

| Description | As Shipped | High Speed Option |
|---------------|-----------------------|----------------------|
| Stall Torque | 13.5 in-lb [1.68 N-m] | 8.4 in-lb [1.05 N-m] |
| Free Speed | 100 RPM | 160 RPM |
| Stall Current | | 3.6 Amps |
| Free Current | | 0.15 Amps |

Limited 90-day Warranty
This product is warranted
by Innovation First against
manufacturing defects in material
and workmanship under normal
use for ninety (90) days from the
date of purchase from authorized
Innovation First dealers. For
complete warranty details and
exclusions, check with your dealer.



Innovation First, Inc.
1519 IH 30 W
Greenville, TX 75402

12/10

For More Information, and additional Parts & Pieces refer to:
www.VEXrobotics.com

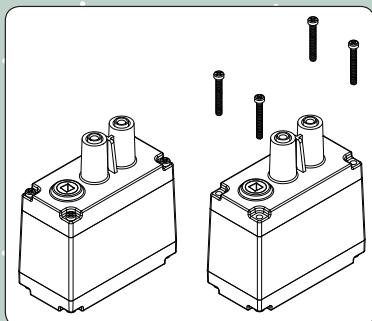
Motion Accessories

2 Wire Motor Kit, continued

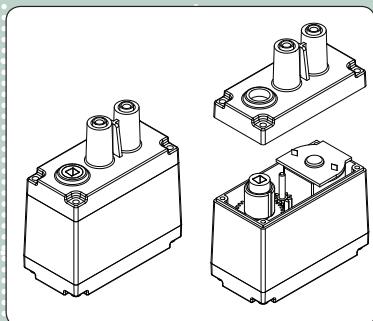
Gear Change Procedure

To configure the high speed option, follow these instructions:

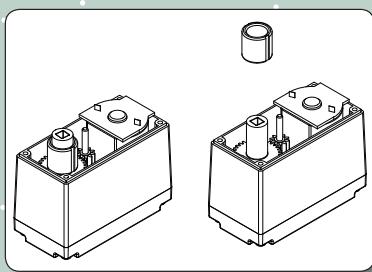
1. Remove the four screws in the corners of the front of the motor case.



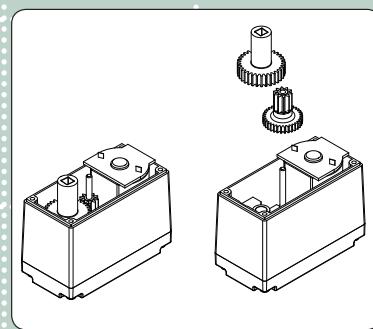
2. Lift off the top cover. Do not disturb the gears inside.



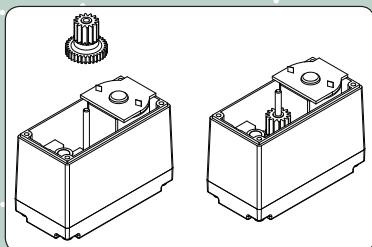
3. Lift off the output bushing and place to the side. This will be used later.



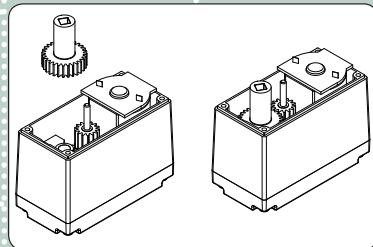
4. Remove the middle gear and the output shaft gear.



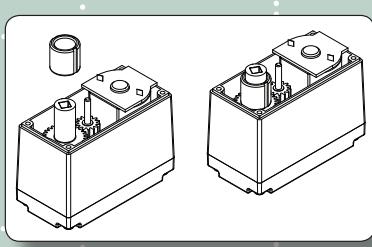
5. Install the high speed middle gear.



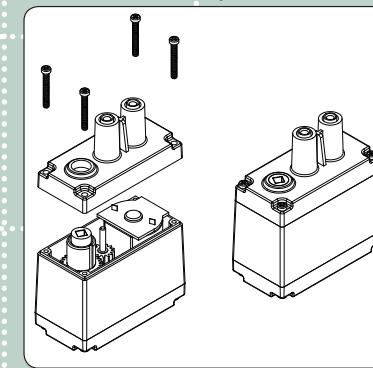
6. Install the high speed output shaft gear.



7. Install the output bushing removed in step 3. Make sure the bushing orientation is as shown.



8. Replace the cover and four screws removed in steps 1 and 2.



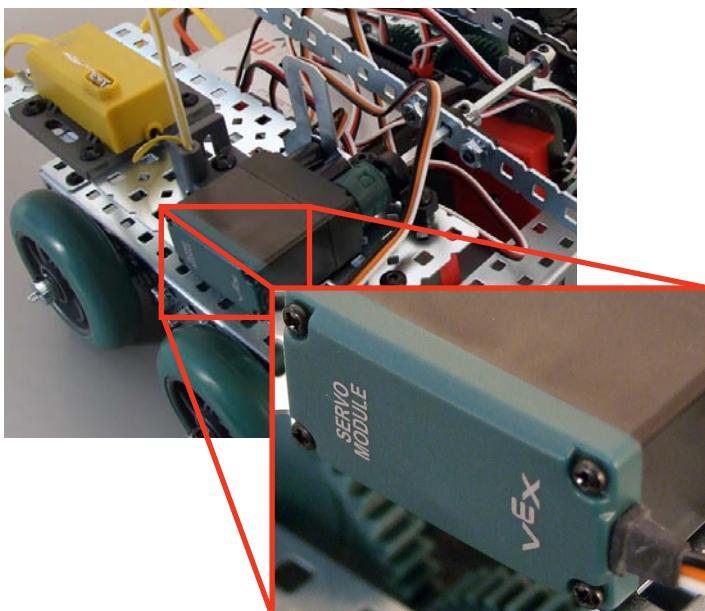
Servo Motors Overview

A Servo Module (or Servo Motor) rotates its shaft to a set angular position, between 0 and 120 degrees. Once its position has been set in a ROBOTC program, the Servo Module will continually draw power to maintain that position until another is specified. Servo Modules can be plugged into any of the MOTORS ports in ROBOTC.



The only differences in appearance between the VEX Servo and Motor Modules are their labels on the back, but the two should not be confused. When a Motor Module is set equal to a value, it uses that value as a power setting and starts spinning its shaft in continuous rotations. When a Servo Module is set equal to a value, however, it uses that value to rotate its shaft to a position and hold it there.

Servo Modules are typically and appropriately found in robotic grippers and arms because of their small range of motion, ability to be set to a specific position, and ability to hold that position. They can be set to values ranging from -127 to 127, with -127 being fully rotated one way and 127 fully rotated the other.



Mounted Servo Module
Here, the Servo Module is mounted on Squarebot 3.0. It allows the arm to be rotated and held at specific positions.

Servo Motors Sample Code

Rotating the Servo Modules Shaft to Different Positions

This code rotates a Servo Module on MOTOR Port 6 to a different position every second, starting at -127 (fully backward) and ending at 127 (fully forward).

```
motor[port6] = -127; //Set position fully backward
wait1Msec(1000); //Wait for 1 second

motor[port6] = -95; //Set position 3/4 backward
wait1Msec(1000); //Wait for 1 second

motor[port6] = -63; //Set position 1/2 backward
wait1Msec(1000); //Wait for 1 second

motor[port6] = -31; //Set position 1/4 backward
wait1Msec(1000); //Wait for 1 second

motor[port6] = 0; //Set position to middle
wait1Msec(1000); //Wait for 1 second

motor[port6] = 31; //Set position 1/4 forward
wait1Msec(1000); //Wait for 1 second

motor[port6] = 63; //Set position 1/2 forward
wait1Msec(1000); //Wait for 1 second

motor[port6] = 95; //Set position 3/4 forward
wait1Msec(1000); //Wait for 1 second

motor[port6] = 127; //Set position fully forward
wait1Msec(1000); //Wait for 1 second
```

servomotor kit

Servomotor

As explained in the Motion Subsystem section of the Inventor's Guide, servomotors are a type of motor that can be directed to turn to face a specific direction, rather than just spin forward or backward.

INSERT THIS PAGE
at the **back of the**
Motion Chapter in your
Vex Inventor's Guide.

Limited 90-day Warranty

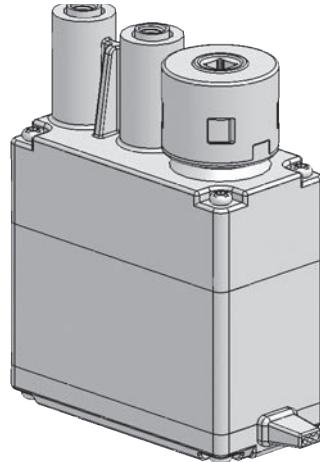
This product is warranted by Innovation One against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation One dealers. For complete warranty details and exclusions, check with your dealer.

Innovation One, Inc.
350 North Henderson Street
Fort Worth, TX 76102

11/04

Printed in China
0105

servomotor x 1



screw 6-32 x 1/2" x 2



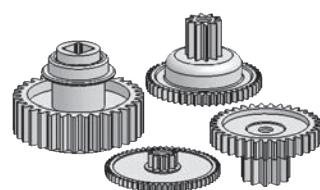
screw 6-32 x 1/4" x 2



clutch post x 1



replacement gears x 2

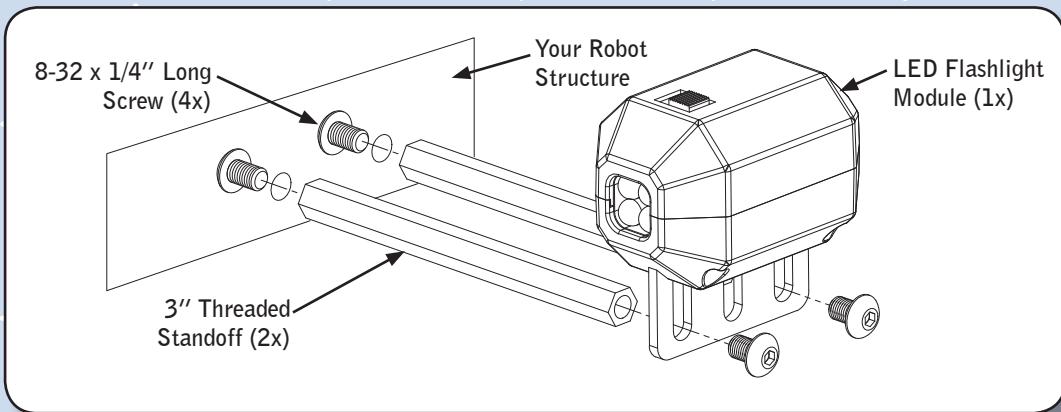


Flashlight

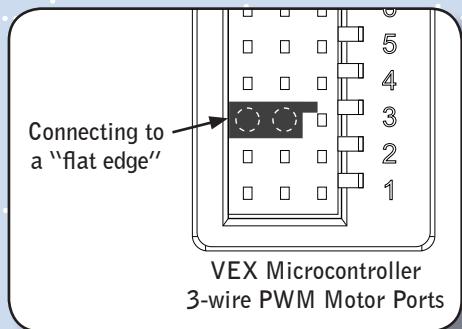
The Flashlight will help your robots see in the dark! The included Flashlight will turn night into day with its four powerful LEDs, allowing for robot operation in low-light conditions.

Instructions for use:

1. Simply mount the Flashlight either using the included standoffs and screws, or other VEX hardware.
 - a. The mounting tab at the bottom of the Flashlight will allow it to mount easily onto the side of any VEX component.



2. Plug the 2-prong cable into a power-source. This accessory will draw power from a 3-wire PWM motor port on a VEX Microcontroller. When connecting to a VEX Microcontroller - plug the 2-prong connector into a 3-pin socket of a Motor Port such that the connector is against the flat edge (not the keyed edge) as shown. The key on the 2-prong Flashlight connector should NOT be inserted into the key of the socket.



3. Switch the Flashlight switch to "on".
 - a. Ensure the Microcontroller is turned on and has power.

Limited 90-day Warranty
This product is warranted by VEX Robotics, Inc. against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized VEX Robotics dealers. For complete warranty details and exclusions, check with your dealer.

VEX Robotics, Inc.
1519 IH 30 W
Greenville, TX 75402

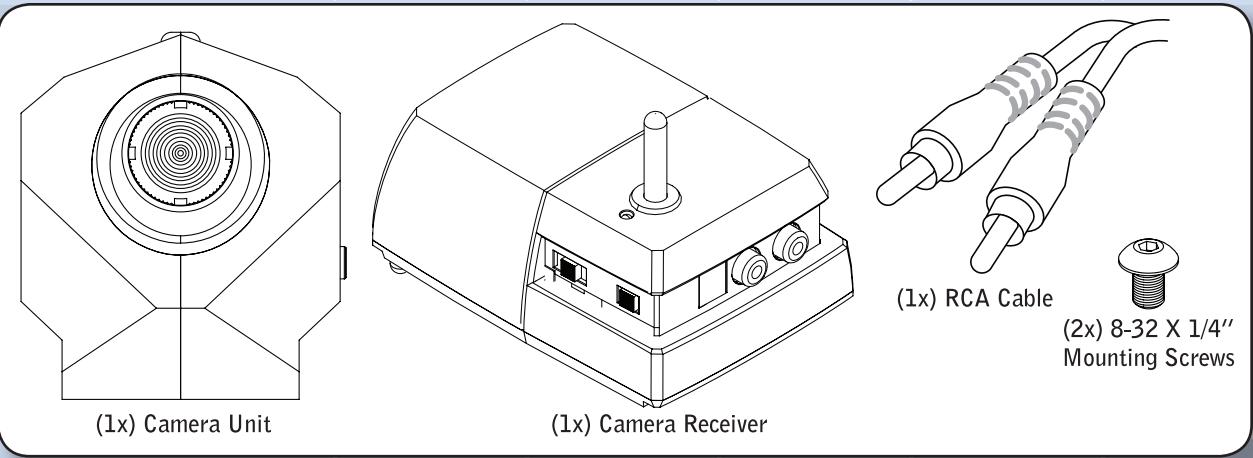
4/10

For More Information, and additional Parts & Pieces refer to:
www.VEXRobotics.com

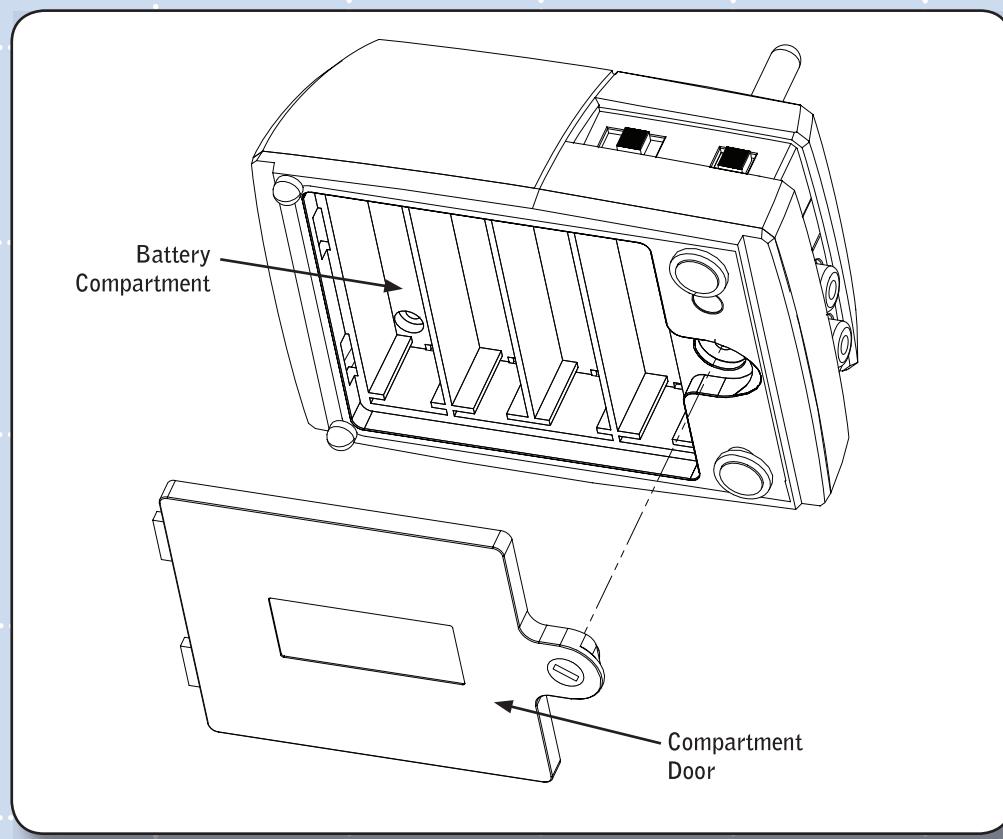
Auxiliary Accessories

Color Camera Kit

The Color Camera Kit can give you a new perspective. Use this wireless color camera to see through your robot's eyes. Connect the Camera Receiver to any television to see and hear from your robot's point-of-view via the 2.4 GHz camera link.



The first step is to insert the batteries into the Camera Receiver. Using a Phillips head screw driver, remove the screw and open the battery door on the bottom of the Camera Receiver. Install (4x) AA-batteries, not included. Insert the batteries in the correct orientation by following the polarity diagram inside the battery compartment. Reinstall battery door and screw.



Limited 90-day Warranty
This product is warranted by VEX Robotics, Inc. against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized VEX Robotics dealers. For complete warranty details and exclusions, check with your dealer.

VEX Robotics, Inc.
1519 IH 30 W
Greenville, TX 75402

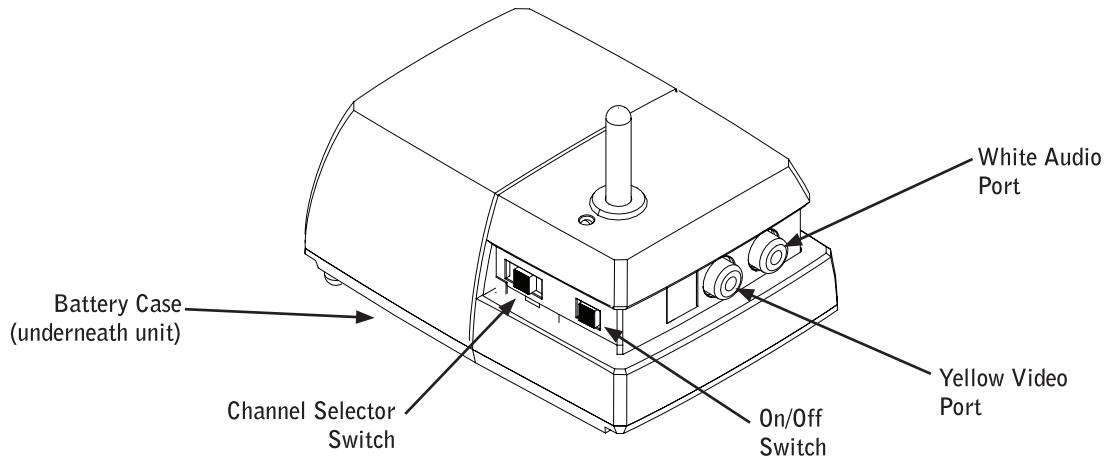
0710

For More Information, and additional Parts & Pieces refer to:
www.VEXRobotics.com

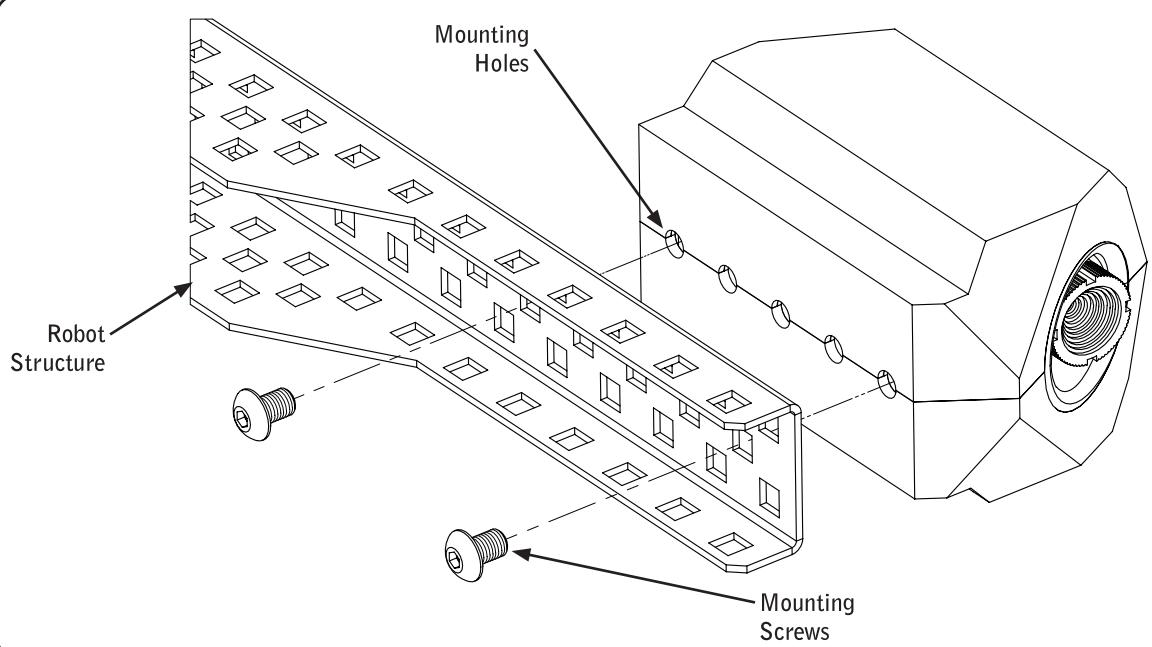
Auxiliary Accessories

Color Camera Kit, continued

Connect the Camera Receiver to a Television (not included) using the RCA Cable provided in the kit. Make sure the yellow connector is connected to the "Video" port on the Receiver, and the television, and the white connector is connected to the "Audio" port on the Receiver and the television.



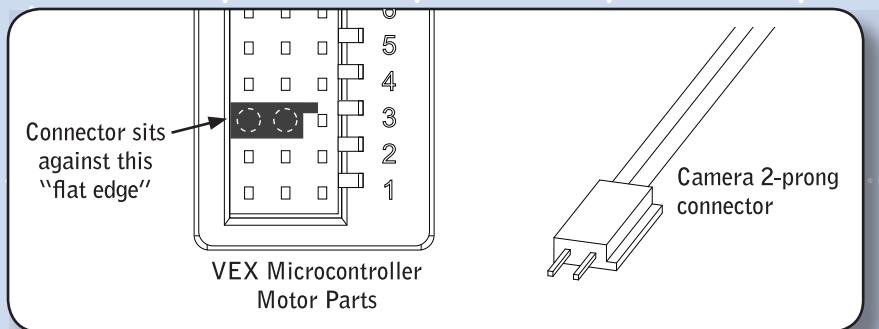
Once the Receiver has batteries installed and is connected to the TV, it is time to mount the Camera to your robot. The Camera has a row of (5x) threaded mounting holes on its bottom. These holes can be used to attach to Camera unit to any VEX structural component. The image below shows an example of camera mounting using the 2 Mounting Screws provided in this kit.



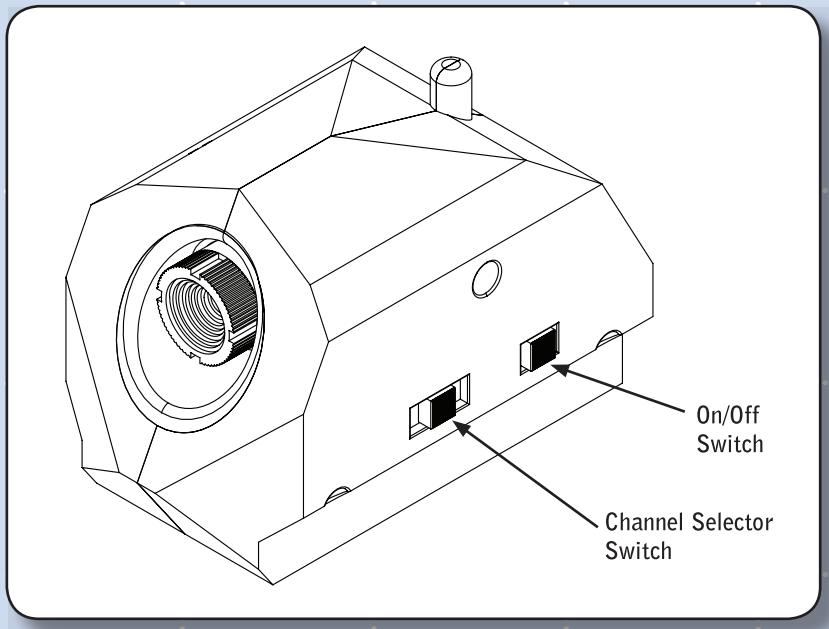
Color Camera Kit, continued

The Camera draws power from a motor port on a VEX Microcontroller plug the 2-prong connector into a 3-pin socket so that the connector is against the flat edge (not the keyed edge) as shown. The key on the connector should NOT be inserted into the key of the socket.

Note: The robot must be turned “on” for the Camera to receive power. You cannot have the robot turned “off” and still operate the camera.



Once the Camera is mounted and connected to power, it is time to turn the system on. Ensure that both the Camera unit and the Receiver are set to the same channel. (Each of these units has a “Channel Selector Switch”, set these to the same number). Turn both units “on”, and set the TV to the correct video input channel (this will vary, depending on the TV).



Color Camera Kit, continued

Camera Troubleshooting Guide:

Problem:

The LED light on the Camera Receiver is not “on”.

Solutions:

1. Check your batteries to see if they have been installed correctly.
2. Check that batteries are charged.
3. Check that the power switch is in the “on” position.

Problem:

The LED light on the Camera Unit is not “on”.

Solutions:

1. Check that the power switch on the Camera is in the “on” position.
2. Check that the Camera 2-prong connector is correctly connected.
3. Check that the robot is turned “on” and has fresh batteries.

Problem:

The LED light is activated on both the Camera Receiver and the Camera Unit, but I get no video.

Solutions:

1. Check that the RCA Video Cable is correctly attached to your television and to the Camera Receiver.
2. Check that your television is on the correct input channel (refer to your TV manual).
3. Check that the “Channel Selector Switch” on the Camera Unit and Receiver are set to the same number.

Problem:

The video quality is poor.

Solutions:

1. Check that the Camera is not too far away from the Receiver. The Camera will only work within a certain distance from the Receiver.
2. There may be interference from outside sources. Try changing the channel on the Camera and the Receiver to a different number, and see if the video quality improves. Note, the Camera and Receiver must both be set to the same channel.
3. The RCA Video Cable may be slightly disconnected, try disconnecting and reinstalling the cable for a better connection.
4. The RCA Cable may be damaged or broken. Try an alternate RCA cable.
5. The battery level on your robot or Camera Receiver is low, replace the batteries.

ultrasonic sensor kit

Ultrasonic Sensor Kit

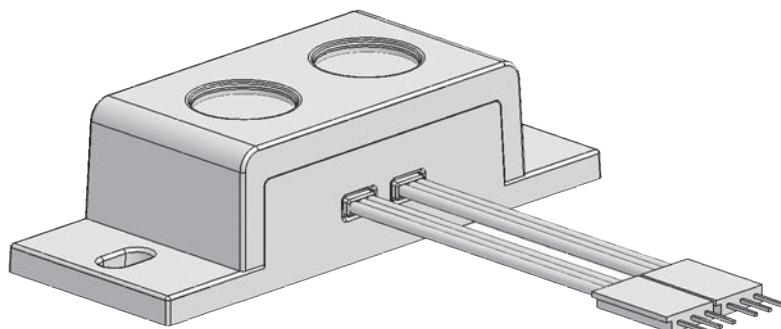
"Ultrasonic" refers to very high-frequency sound – sound that is higher than the range of human hearing. Sonar, or "Sound Navigation And Ranging," is an application of ultrasonic sound that uses propagation of these high-frequency sound waves to navigate and detect obstacles.

Sonar has a wide variety of applications and a wide variety of users, from submarines avoiding underwater obstacles to hungry bats looking for their dinner!

INSERT THESE PAGES
at the **back of the**
Sensor Chapter in your
Vex Inventor's Guide.

**YOU MUST HAVE A
PROGRAMMING KIT
TO USE THIS SENSOR!**

ultrasonic module x 1



**screw x 2
(8-32, $\frac{3}{8}$ "")**



keps nut x 2



Limited 90-day Warranty

This product is warranted by Innovation One against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation One dealers. For complete warranty details and exclusions, check with your dealer.

Innovation One, Inc.
350 North Henderson Street
Fort Worth, TX 76102

11/04

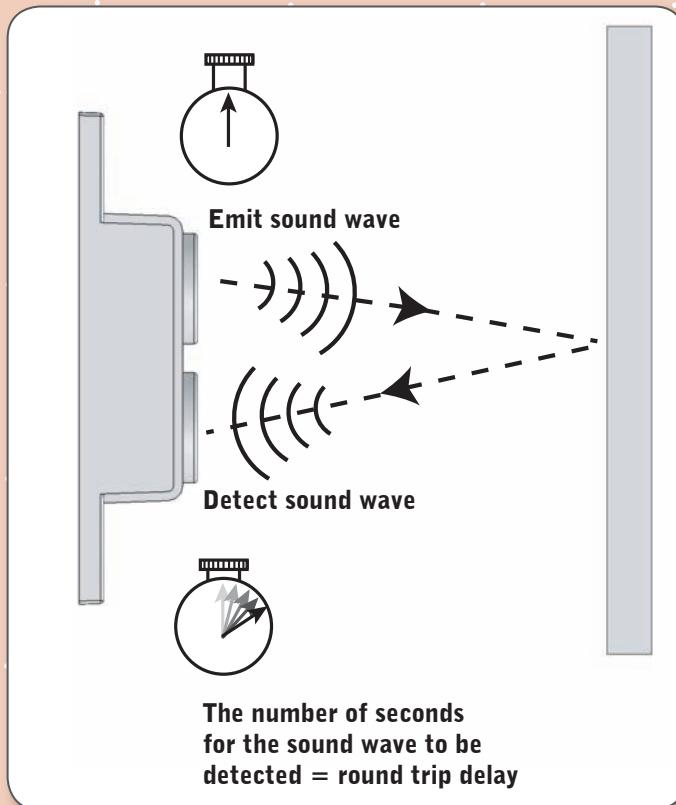
Printed in China
0405



ultrasonic sensor kit, continued

1 Technical overview

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



The ultrasonic sensor can determine the distance to an object between 3cm and 3m away; closer than 3cm will result in the sound waves echoing back to the sensor before the detector is ready to receive.

The ultrasonic sensor actually consists of two parts: an emitter, which produces a 40kHz sound wave; and a detector, which detects 40kHz sound waves and sends an electrical signal back to the microcontroller. In order to determine the distance to an object, it is necessary to implement a timing loop in your microcontroller code to measure the length of time required for the sound wave generated by the emitter to traverse the distance to the object.

The distance to the object can then be calculated with the following formulas:

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

[Note: speed of sound varies with altitude and temperature. At sea level and room temperature, it's approximately 344.2 m/s or 1135 ft/s. It will increase with temperature and decrease with altitude.]

Therefore

$$\text{Distance in feet} = 567.5 \text{ ft/s} \times (\text{round trip delay})$$

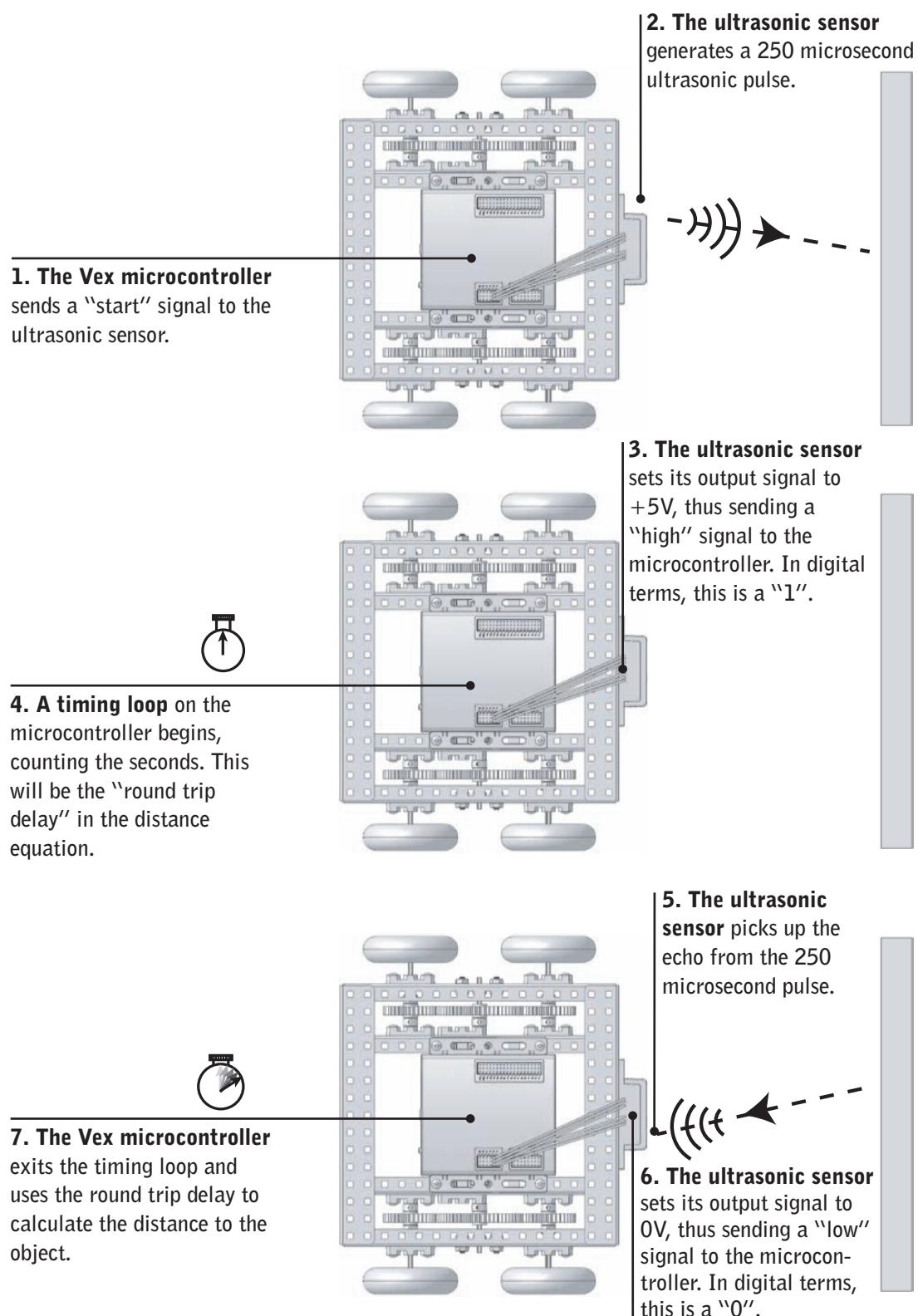
or

$$\text{Distance in meters} = 172.1 \text{ m/s} \times (\text{round trip delay})$$

ultrasonic sensor kit, continued

1 Technical overview

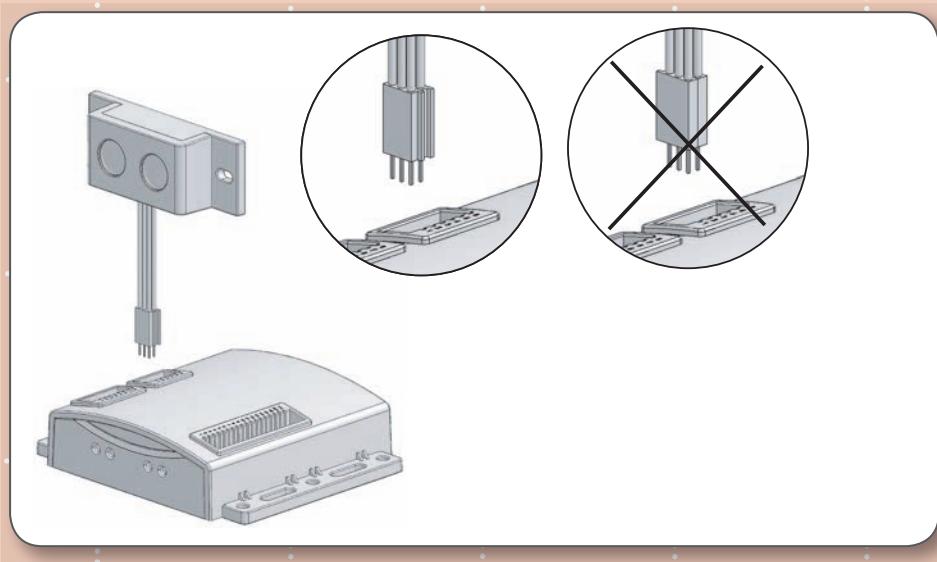
The steps your robot's program will have to follow in order to calculate the distance to an object are:



ultrasonic sensor kit, continued

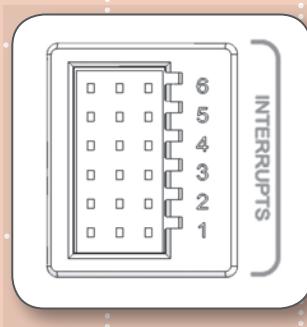
2 Connecting the ultrasonic sensor to the microcontroller

The ultrasonic module has two three-pin connectors that will each plug into an interrupt port on the Vex Microcontroller. These can be adjacent ports, but do not have to be. The connector labelled “INPUT” is the trigger output of the Vex microcontroller; the ultrasonic module receives a start signal from the Vex microcontroller on this line. The connector labelled “OUTPUT” is the echo response from the ultrasonic detector; this is the line through which the Vex microcontroller receives output from the detector, indicating that it has picked up an echo.



3 Reprogramming the microcontroller to enable the ultrasonic sensor to generate a “ping”

Start by plugging the “INPUT” and “OUTPUT” connectors into any two ports in the Interrupts bank on the Vex Microcontroller.



In order for your robot to be able to read the sensor, you will have to reprogram the microcontroller. Sample code to help you get started is available on the VexLABS.com website. Refer to the Programming chapter in your Vex Inventor’s Guide for information on how to add or change code.

Concepts to Understand, continued

Bumper Switch Sensor

Bumper Switch Sensor

Signal: Digital

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

Technical Info:

Type: SPST switch

("Single Pole, Single Throw") configured for Normally Open behavior.

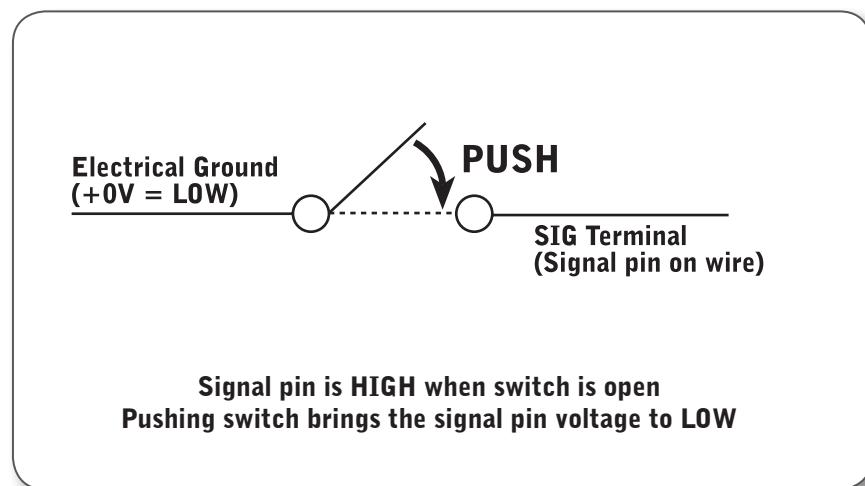
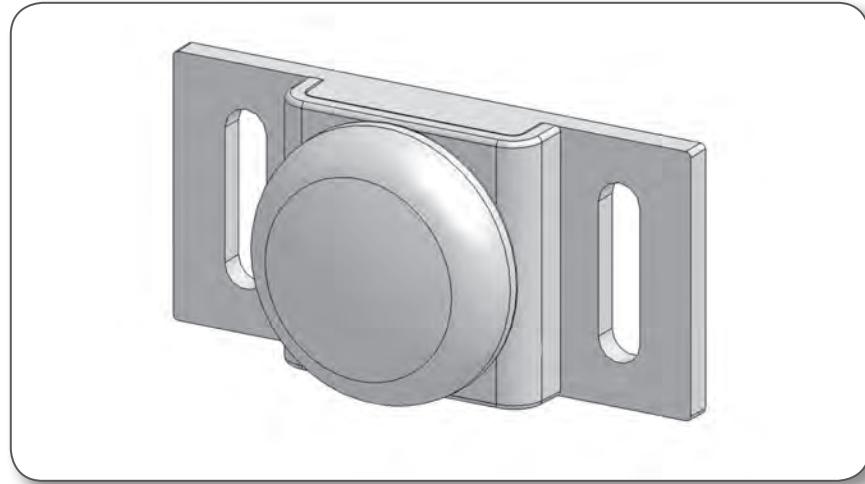
Signal Behavior: When the switch is not being pushed in, the sensor maintains a digital HIGH signal on its sensor port. This High signal is coming from the Microcontroller. When an external force (like a collision or being pressed up against a wall) pushes the switch in, it changes its signal to a digital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

Note: You can connect multiple switches to the same port using a y-cable.

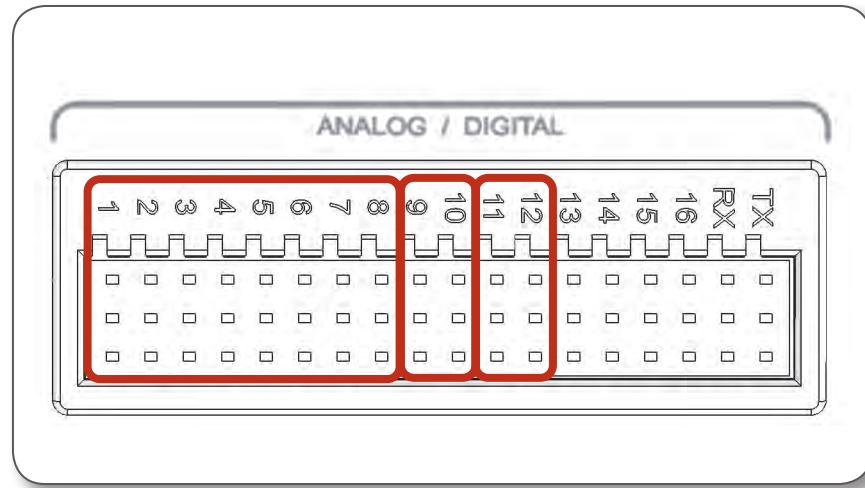
PIC Microcontroller Default Code Behavior Info:

Usable Ports: Analog/Digital 1-8 (Limit Switch Behavior), 9-10 (Tag Behavior), 11-12 (Autonomous Behavior)

For more info, see Programmed Behaviors later in this section.



Signal pin is HIGH when switch is open
Pushing switch brings the signal pin voltage to LOW



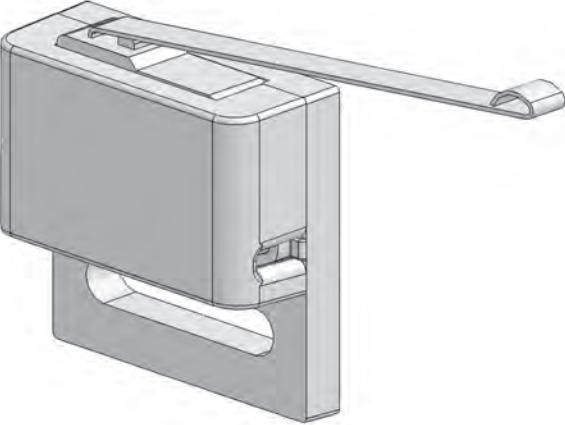
Concepts to Understand, continued

Limit Switch Sensor

Limit Switch Sensor

Signal: Digital

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



Technical Info:

Type: SPDT microswitch, configured for SPST

Normally Open behavior.

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

Note: You can connect multiple switches to the same port using a y-cable.

PIC Microcontroller Default

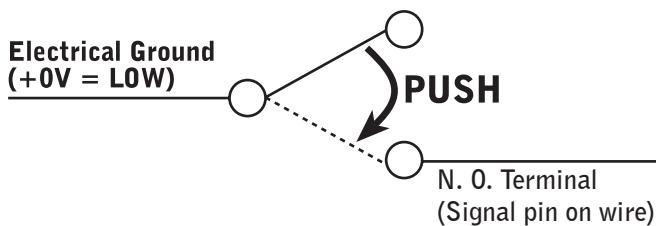
Code Behavior Info:

Usable Ports: Analog/Digital 1-8 (Limit Switch Behavior), 9-10 (not recommended),

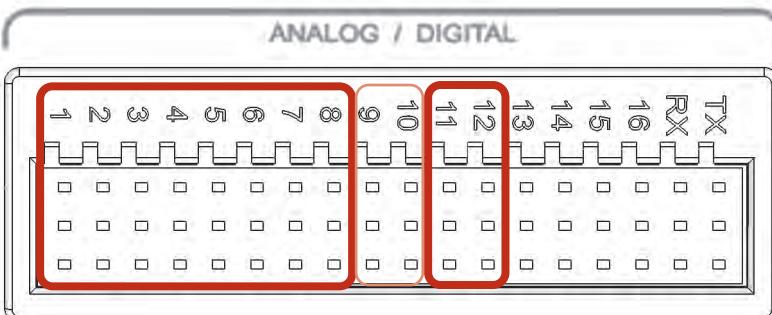
11-12 (Autonomous Behavior)

For more info, see

Programmed Behaviors later in this section.



Pushing switch brings the signal pin voltage to LOW



Shaft Encoders Overview

The Quadrature Shaft Encoder detects the rotation of an axle that passes through it. It has a resolution of 360 counts per revolution (2 count intervals), and can distinguish between clockwise and counterclockwise rotation.



The Quadrature Shaft Encoder is an upgrade from the original Shaft Encoder. The original version contains only one internal sensor, which detects the slits in an internal disc as it spins, giving it a resolution of 90 counts per revolution. Only one output channel (wire) is needed to transmit the sensor data to the Vex Microcontroller.



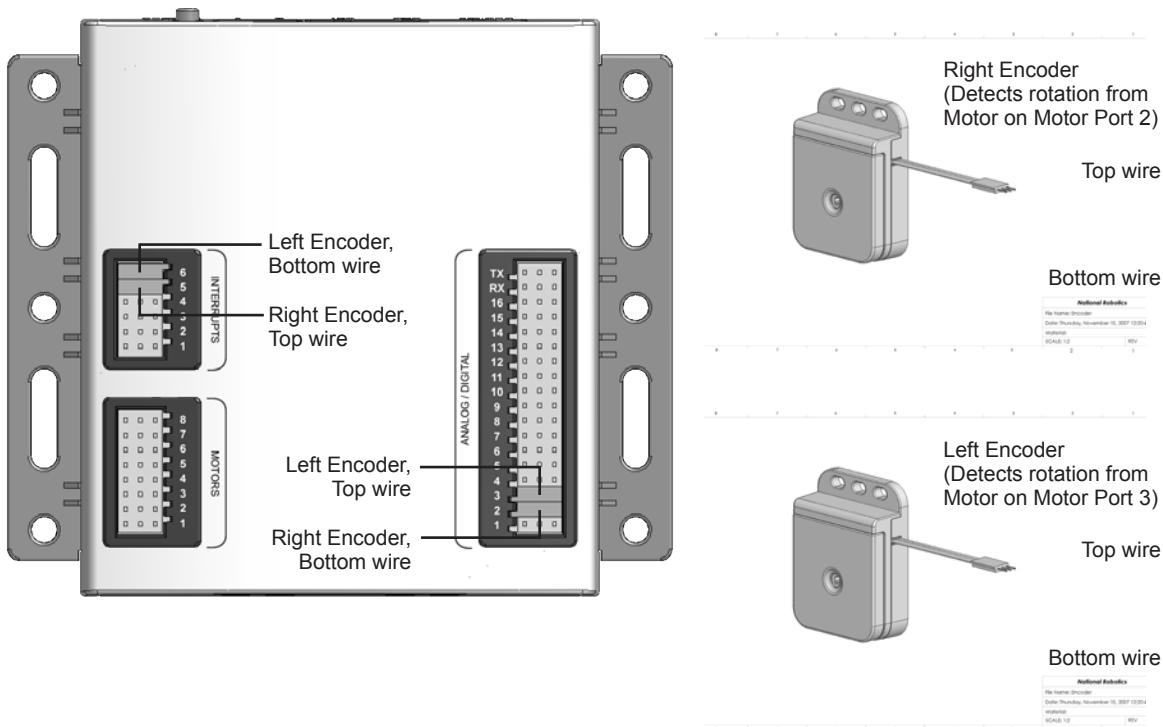
Original Shaft Encoder
Only has one output wire.

Quadrature Shaft Encoder
Has two output wires.

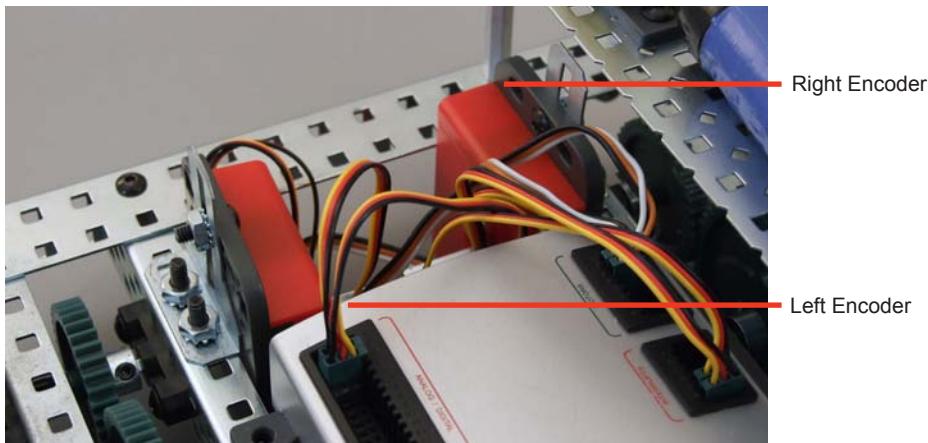
The upgraded Quadrature Shaft Encoder includes a second optical sensor which allows the sensor to detect if the internal disk is spinning clockwise or counterclockwise and increases the resolution to 360 counts per revolution (2 count intervals). Two output channels (wires) are needed to transmit its sensor data to the Vex.

Shaft Encoders Wiring Configuration

The Quadrature Shaft Encoder is fully compatible with the existing Squarebot 2.0 and 3.0 models. Use the following wiring configuration to ensure that the Encoders count "up" when the robot drives forward, and "down" when the robot moves in reverse. Reversing the placement of the wires will cause the Encoders to count in the wrong direction (-2, -4, -6 instead of 2, 4, 6); failing to place the wires in the correct ports will result the Quadrature Encoder behaving as an original Encoder, or not at all.

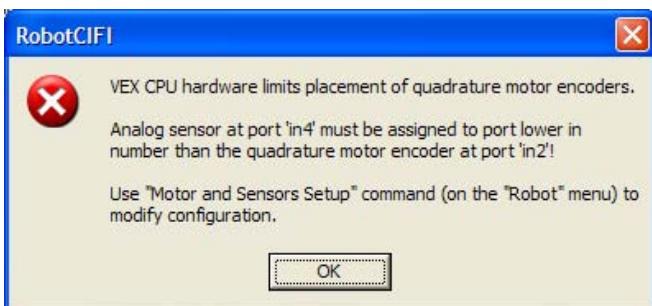
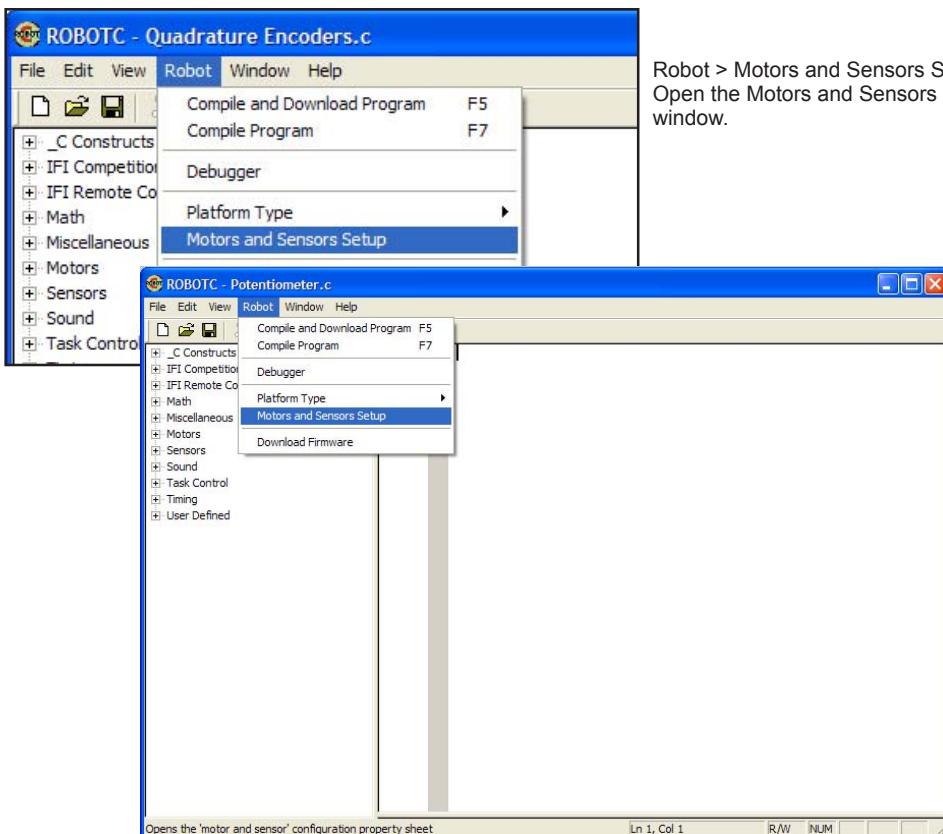


Two Quadrature Shaft Encoders mounted on Squarebot 2.0



Shaft Encoders ROBOTC Setup

The Quadrature Shaft Encoder is also fully supported by ROBOTC for IFI (v. 1.40 and up). Use the following instructions and the wiring configuration on the previous page to correctly configure them within ROBOTC.



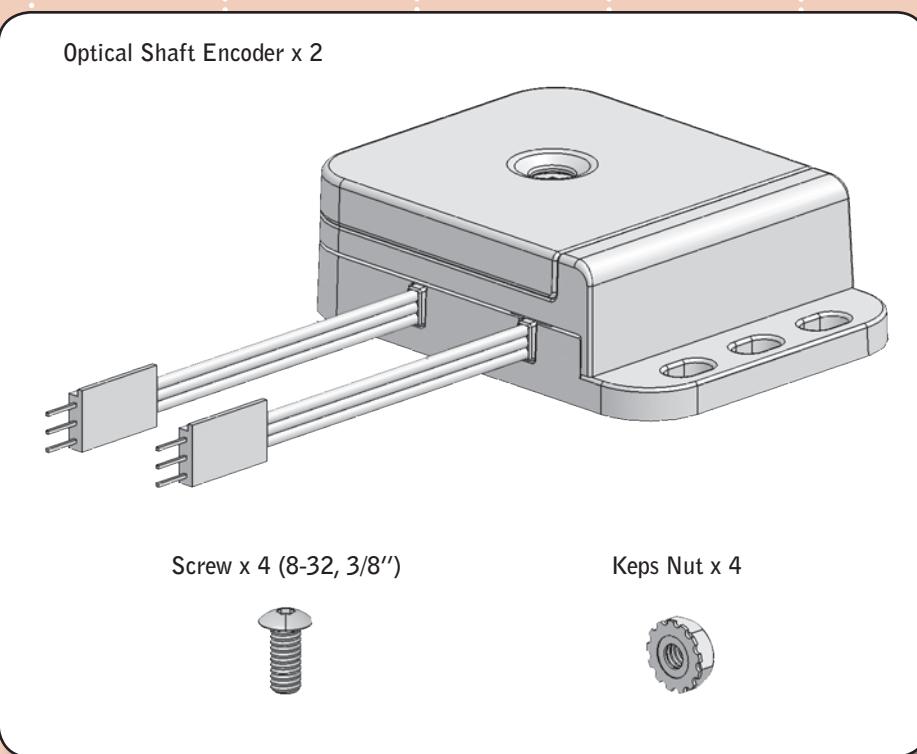
Note: The Quadrature Encoders can be plugged into any of the Analog / Digital Ports (in1 through in16) and Interrupt Ports int3 through int6. However, if your robot is being configured with analog sensors (Potentiometer, Reflection, Light) as well, the Encoders must be plugged into higher port numbers for them to

Optical Shaft Encoder Kit

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

**YOU MUST HAVE A
PROGRAMMING KIT
TO USE THIS SENSOR!**

With the Quadrature Encoder, there are 2 output channels. Only one output can be used as a basic Optical Shaft Encoder. The term quadrature refers to the situation where there are two output channels; that is, two square waves 90 degrees out of phase with each other, being outputted by the unit. The two output channels of the Quadrature Encoder can be used to indicate both position and direction of rotation.



Limited 90-day Warranty

This product is warranted by Innovation First against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation First dealers. For complete warranty details and exclusions, check with your dealer.

Innovation First, Inc.
1519 IH 30 W
Greenville, TX 75402

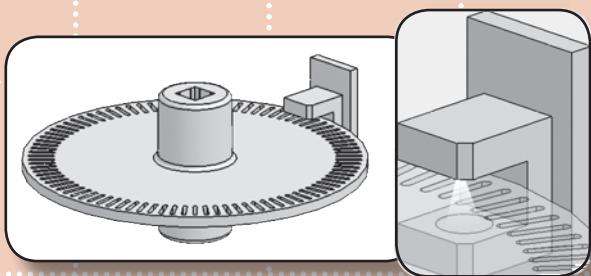
08/07

For More Information, and additional Parts & Pieces refer to:
www.VexRobotics.com

Optical Shaft Encoder Kit, continued

1 Technical overview

The Optical Shaft Encoder uses an infrared light sensor to detect illumination from an infrared LED passing through slots cut in the circumference of a rotating wheel.



From basic geometry, we know that the circumference of a circle is equal to (π) times the diameter of the circle.

$$\text{circumference} = \text{diameter of wheel} \times \pi \quad (\pi = \text{approx. } 3.14)$$

The distance travelled by a wheel, then, is simply the circumference of the wheel times the number of revolutions the wheel has made.

$$\text{distance} = (\text{circumference}) \times (\text{number of revolutions})$$

For a standard wheel in the Vex Inventor's kit, the diameter is 2.75". So the distance the wheel travelled would be:

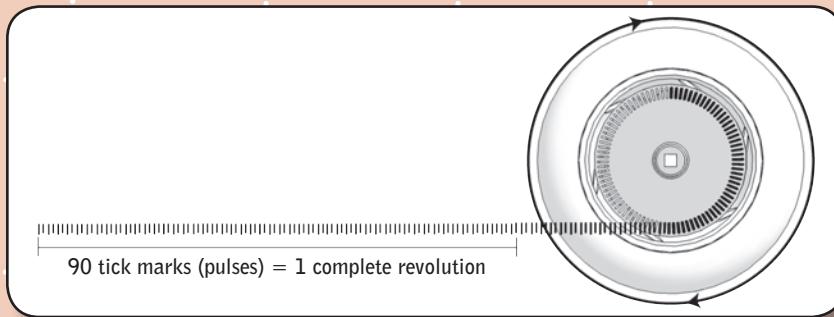
$$\text{distance} = 8.64" \times (\text{number of revolutions})$$

For More Information, and additional Parts & Pieces refer to:
www.VexRobotics.com

Optical Shaft Encoder Kit, continued

1 Technical overview continued

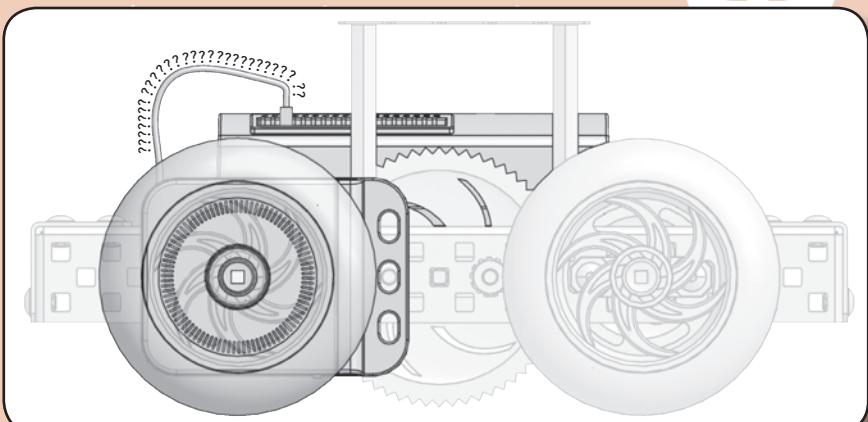
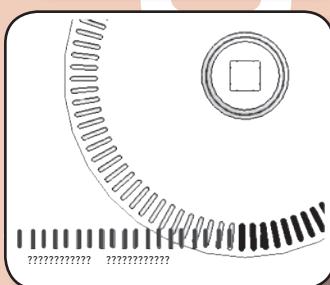
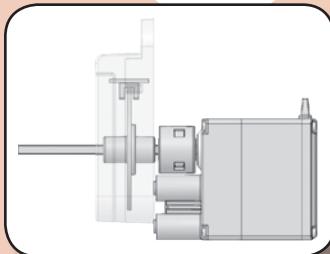
By knowing how many slots are cut into the encoder wheel, we can determine how many revolutions the robot wheel has made based on the number of times the light sensor has picked up illumination from the LED. The encoder wheel included in this kit has 90 slots.



By mounting a shaft encoder on the axle of one of your robot's wheels, you'll be able to determine how many times that wheel has rotated. That, in turn, can be used to calculate the distance the robot has travelled, based on the diameter of the wheel.

The Optical Shaft Encoder can detect up to 1700 pulses per second, which corresponds to 18.9 revolutions per second and 1133 rpm (revolutions per minute). Faster revolutions will not be interpreted correctly, resulting in erroneous positional data being passed to the microcontroller.

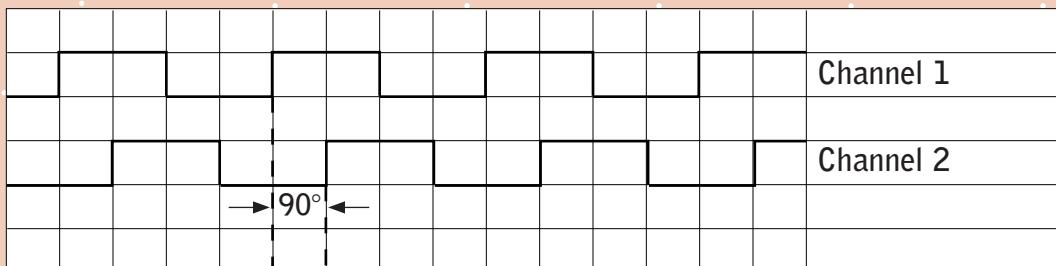
This is a digital sensor, which means that the signal it will pass to the Vex microcontroller will either be high (1) or low (0). The sensor output is low (0) when the light from the IR LED passes through a cutout segment of the encoder wheel and falls on the detector, and high (1) when the light is blocked by an opaque segment of the encoder wheel. This means that the Vex microcontroller will be receiving a string of 1's and 0's as your robot moves. The string of 1's and 0's will then be interpreted by your program and used to determine the robot's actions.



For More Information, and additional Parts & Pieces refer to:
www.VexRobotics.com

Optical Shaft Encoder Kit, continued

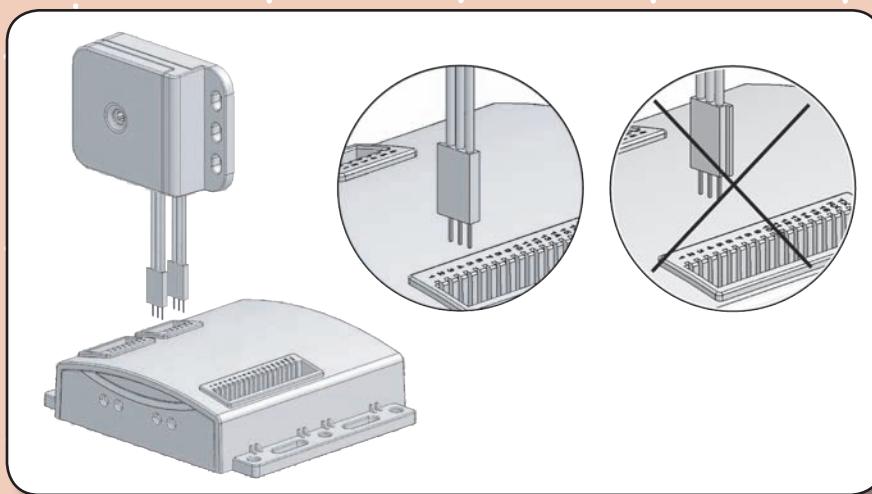
With the Quadrature Encoder, you will use both outputs (Channel 1 and Channel 2) to determine of the direction of rotation. The channels are separated in phase by 90 degrees as shown below.



For example, if channel 1 leads channel 2, the wheel is rotating clockwise. Likewise, if channel 2 leads channel 1, the wheel is rotating counter-clockwise. By monitoring the relative phase and number of pulses of channel 1 and 2, you can determine how fast, how far, and what direction your robot is traveling.

2 Reprogramming your microcontroller to read the sensor

You'll need to plug your shaft encoder into any port in the Interrupt bank on the Vex Microcontroller. Depending on your specific application, you may be able to use any port in the Analog/Digital bank. Note that the connector is keyed to fit into the microcontroller port in a specific orientation; plugging it in backwards could damage or even destroy your sensor.



In order for your robot to be able to read the sensor, you will have to reprogram the microcontroller. Sample code to help you get started is available on the Vex website.

For More Information, and additional Parts & Pieces refer to:

www.VexRobotics.com

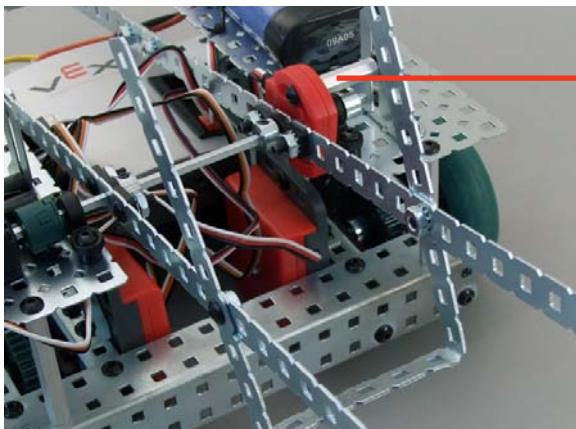
Potentiometers Overview

The Potentiometer is used to measure the angular position of the axle or shaft passed through its center. The center of the sensor can rotate roughly 265 degrees and outputs values ranging from 0-1023 to the Vex Microcontroller.



The Potentiometer can be attached to the robot using the mounting arcs surrounding the center of the sensor. The arcs provide flexibility for the orientation of the Potentiometer, allowing the full range of motion to be utilized more easily.

When mounted on the rotating shaft of a moving portion of the robot, such as an arm or gripper, the Potentiometer provides precise feedback regarding its angular position. This sensor data can then be used for accurate control of the robot.

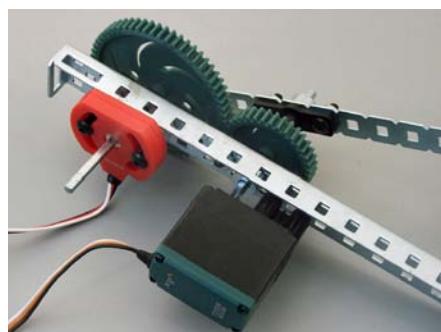


Mounted Potentiometer
Here, the Potentiometer is mounted on Squarebot 3.0. It provides feedback regarding the position of the movable arm.

CAUTION! When mounting the Potentiometer on your robot, ensure that the range of motion of the rotating shaft does not exceed that of the sensor. Failure to do so may result in damage to your robot and the Potentiometer.

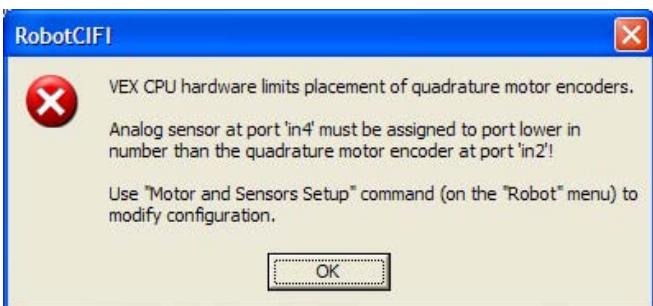
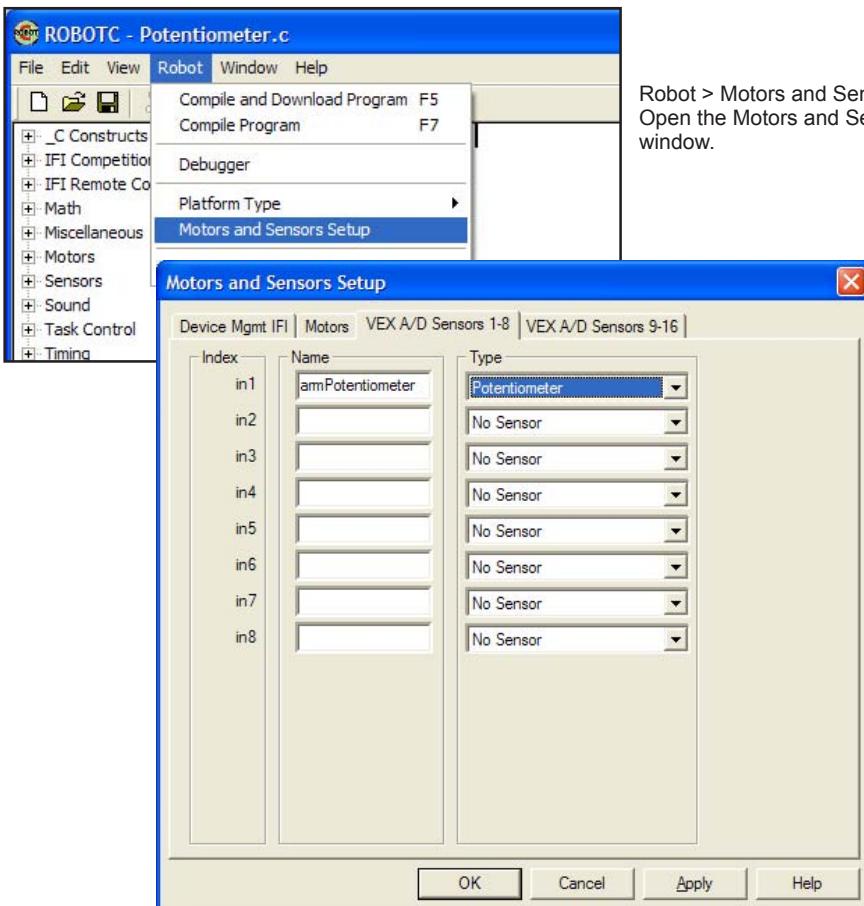
Gear it Up
If the range of motion is too large for the Potentiometer, try developing a gear train that would allow you to measure the rotation of the shaft.

Note: Your sensor feedback will lose some resolution.



Potentiometers ROBOTC Setup

The Potentiometer is fully supported by ROBOTC for IFI (v. 1.4 and up). Use the following instructions and to correctly configure one within ROBOTC.



Note: The Potentiometer can be plugged into any of the Analog / Digital ports (in1 through in16). Any digital sensors (Limit Switches, Bumper Switches, Encoders, Ultrasonic Rangefinders) must be used in higher Port numbers for them to be configured correctly.

Potentiometers Sample Code

Limiting Arm Movement with the Potentiometer

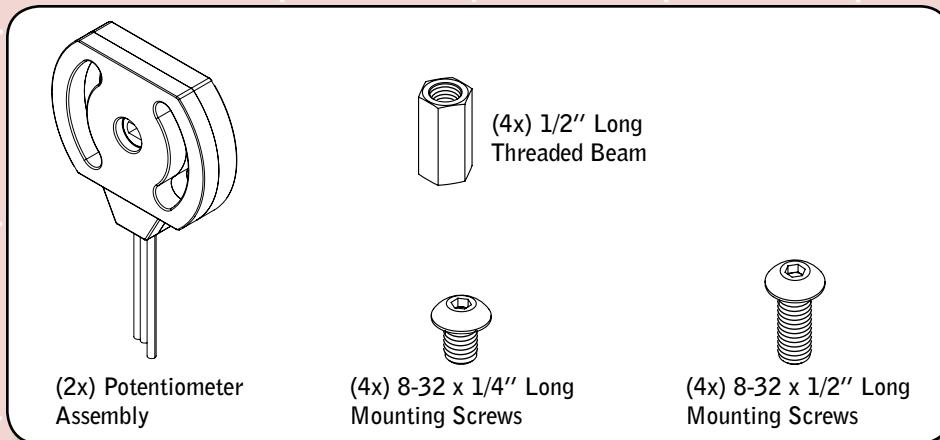
This code allows the rotating arm of a robot to be remote controlled using the Ch5 rear buttons on the Radio Control Transmitter. The Potentiometer is used to prevent the motor from spinning once the arm has reached its minimum and maximum points.

```
bIfiAutonomousMode = false; //Enable Radio Control mode
while(true)
{
    if(vexRT[Ch5] == 127) //If the top Ch5 button is pressed...
    {
        if(SensorValue[armPotentiometer] < 900) //If the Potentiometer
        {
            motor[port6] = 31; //has not reached its maximum point...
            //turn the motor on forward.
        }
        else
        {
            motor[port6] = 0; //If the Potentiometer has reached
            //its maximum point...
            //turn the motor off.
        }
    }
    if(vexRT[Ch5] == -127) //If the bottom Ch5 button is pressed...
    {
        if(SensorValue[armPotentiometer] > 550) //If the Potentiometer
        {
            motor[port6] = -31; //has not reached its minimum point...
            //turn the motor on in reverse.
        }
        else
        {
            motor[port6] = 0; //If the Potentiometer has reached
            //its minimum point..
            //turn the motor off.
        }
    }
}
```

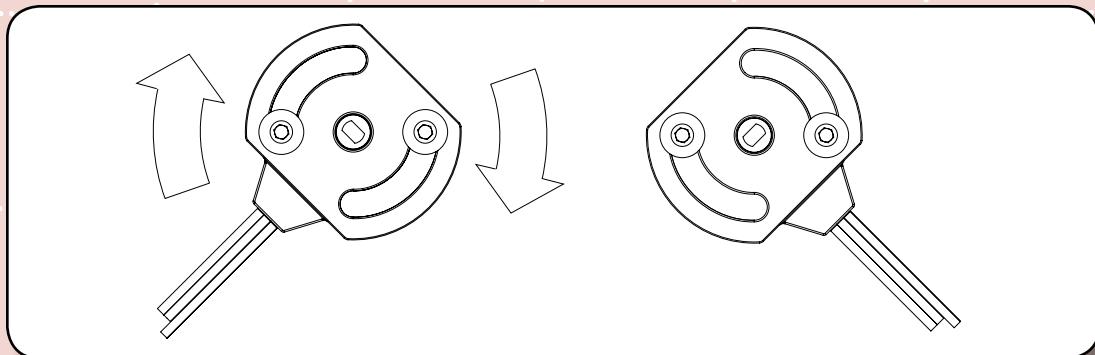
Potentiometer Kit

The Vex Potentiometer will keep things ‘on the level’. Use this sensor to get an analog measurement of angular position. This measurement can help to understand the position of robot arms, or other mechanisms. To effectively utilize this sensor, users are required to use the Vex Programming Kit.

**YOU MUST HAVE A
PROGRAMMING KIT
TO USE THIS SENSOR!**



The Potentiometer is designed with a “D-hole” in the center. This hole should slide easily over the Vex square shafts. The Potentiometer also includes (2) “arcs” which are 1/2” from the center hole; these arcs are used for mounting the Potentiometer to the robot structure.



The mounting arcs allow for 90-degrees of adjustment to the Potentiometer position. Since the Potentiometer has limited travel, it is important to ensure that the shaft that is being measured by the Potentiometer does not travel more than 260-degrees (the Potentiometer can only move mechanically about 265-degrees \pm 5 and can only measure electrically 250-degrees \pm 20). The adjustment arcs allow the Potentiometer’s “range of motion” to be repositioned to match the shaft’s range of motion. To measure the motion of something which moves more than 230-degrees, try gearing down the shaft’s motion to a secondary shaft (this secondary shaft will move less distance) and then measure this secondary shaft.

Limited 90-day Warranty

This product is warranted by Innovation First against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation First dealers. For complete warranty details and exclusions, check with your dealer.

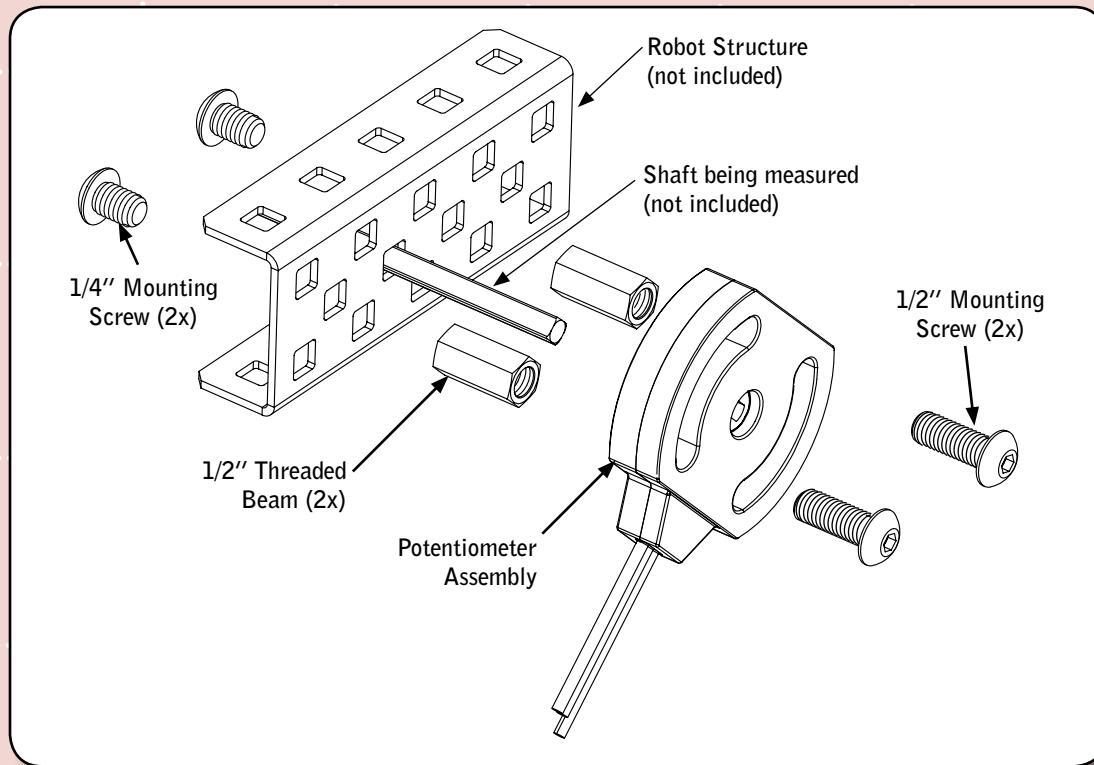
Innovation First, Inc.
1519 IH 30 W
Greenville, TX 75402

10/07

For More Information, and additional Parts & Pieces refer to:
www.VexRobotics.com

Potentiometer Kit, continued

Slide the Potentiometer down the shaft being measured, and ensure that it sticks out of the Potentiometer a little bit on the far side. Mount the Potentiometer using the provided hardware. Ensure the Potentiometer is centered on the shaft and that there is no mechanical bind BEFORE tightening the mounting screws.



The Potentiometer (or Pot for short) describes an electrical device in which the user can adjust the resistance. As the resistance of the sensor changes, a varying voltage is created and thus the sensor acts as a variable voltage divider. This varying analog voltage can be measured by the Vex Controller and is proportional to the position of the shaft connected to the center of the Pot. This is how you obtain an analog measurement of an angular position.

Before you can use the Potentiometer, you must reprogram your Vex Controller to read the varying voltage of the sensor on the corresponding port you are planning on connecting to. How to write/change your code to read the varying voltage is not covered in these instructions. We suggest searching our Forum for help at www.vexforum.com. To connect the Potentiometer Sensor to the Vex Controller, you plug the Sensor Connector into any port in the Analog/Digital Bank on the Vex Controller, typically you start with the 1st position. Note that the Connector is keyed to fit into the Vex Controller Port in a specific orientation; plugging it in backwards could damage your Sensor.

For More Information, and additional Parts & Pieces refer to:
www.VexRobotics.com

line follower kit

Line Follower Kit

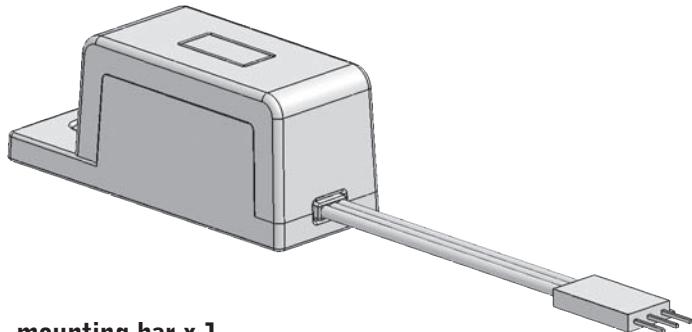
A line follower consists of an infrared light sensor and an infrared LED. It works by illuminating a surface with infrared light; the sensor then picks up the reflected infrared radiation and, based on its intensity, determines the reflectivity of the surface in question. Light-colored surfaces will reflect more light than dark surfaces, resulting in their appearing brighter to the sensor. This allows the sensor to detect a dark line on a pale surface, or a pale line on a dark surface.

INSERT THESE PAGES
at the **back of the**
Sensor Chapter in your
Vex Inventor's Guide.

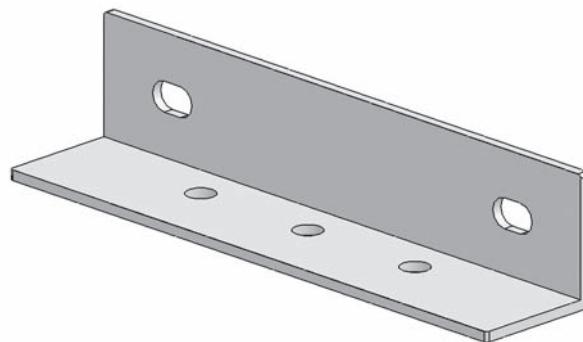
**YOU MUST HAVE A
PROGRAMMING KIT
TO USE THIS SENSOR!**

You can use a line follower to help your robot navigate along a marked path, or in any other application involving discerning the boundary between two high-contrast surfaces. A typical application uses three line follower sensors, such that the middle sensor is over the line your robot is following.

line follower x 3



mounting bar x 1



screw x 5 (8-32, 3/8")



keps nut x 5



Limited 90-day Warranty

This product is warranted by Innovation One against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation One dealers. For complete warranty details and exclusions, check with your dealer.

Innovation One, Inc.
350 North Henderson Street
Fort Worth, TX 76102

11/04

Printed in China
0405

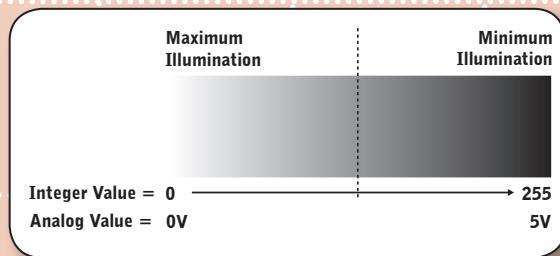


line follower kit, continued

1 Technical overview

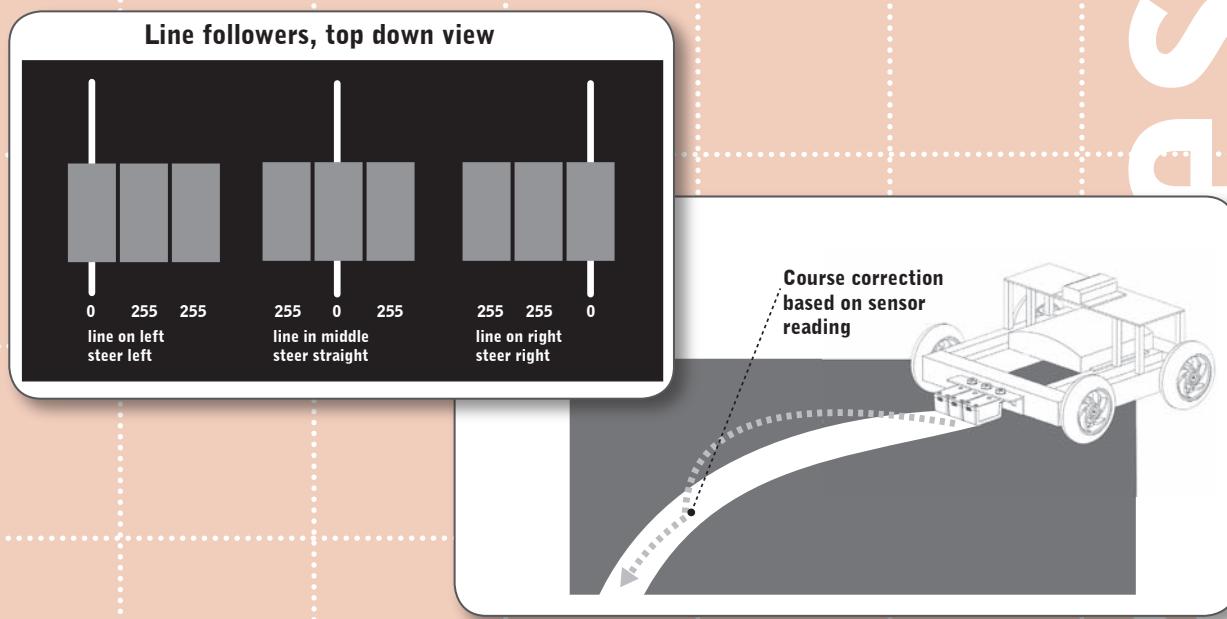
This is an analog sensor, meaning that its output covers a range of values (in this case, from zero to five volts) rather than being only high (five volts) or low (zero volts), as is the case for a digital sensor. This range of output from zero to five volts is sent to the microcontroller, which reads it as a range of integer values from 0 to 255. [For more detail, refer to the Sensors chapter in your Vex Inventor's Guide.]

For this particular sensor, sensor output will be low (around 0) when the infrared light bounces back to the detector – in other words, when the surface is pale or highly reflective – and high (around 255) when the light is absorbed and does not bounce back.



We can then set a threshold value in our code to act as a trigger for behaviors.

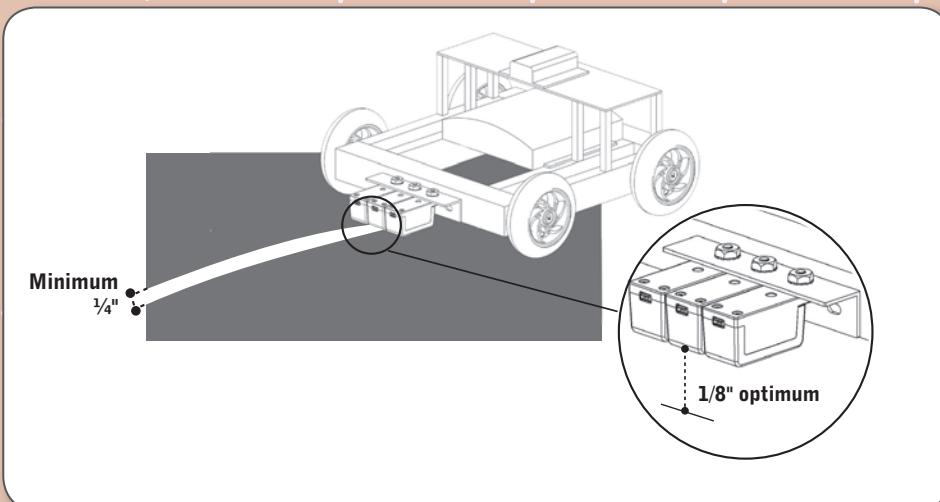
From this basic premise, we can build more complicated behaviors. For example, if you have three line sensors on the front of your robot [hint: use the mounting bar included in your kit!], then you can program your robot to follow a white line on a black surface. LineFollower_Middle should always see white, and the other two — LineFollower_Left and LineFollower_Right — should always see black. If LineFollower_Left starts seeing white, then your robot needs to steer back to the left. If LineFollower_Right starts seeing white, then your robot needs to steer back to the right.



line follower kit, continued

1 Technical overview, continued

The optimal range for the line follower is approximately 0.02 to 0.25 inch.
The minimum line width it can detect is 0.25".



Sensor output will be low (0V) when the infrared light bounces back to the detector – in other words, when the surface is pale or highly reflective – and high (+5V) when the light is absorbed and does not bounce back.

HELPFUL HINT:

Because the line follower uses an infrared LED to illuminate its target and an infrared sensor to detect the reflected light, it will actually work in low-light conditions or even in the dark! However, **this also means that it can easily become saturated** — in other words, everything will look white to it, like an over-exposed photograph — in environments where there is a lot of infrared radiation. You'll find environments like this in competition settings where tungsten lights are used for illumination. **To avoid saturating the infrared sensor**, consider mounting it underneath the robot or adding a cardboard shield to block ambient radiation.

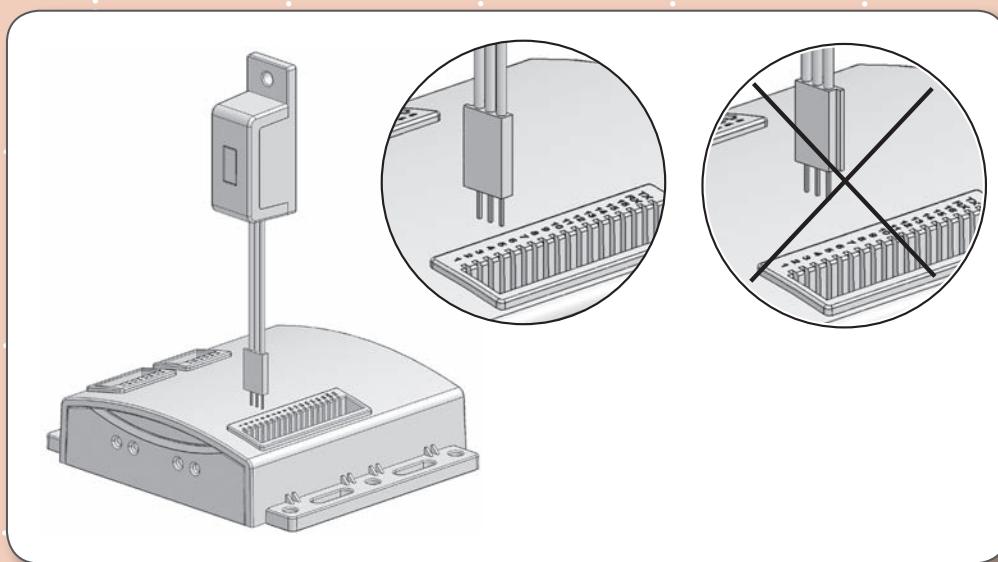
line follower kit, continued

2 Reading data from the line follower:

Reprogramming your microcontroller to read the sensor

Start by plugging your line follower into any port in the Analog/Digital bank on the Vex Microcontroller. Note that the connector is mechanically keyed to fit into the microcontroller ports in a specific orientation.

Plugging it in backwards could result in damage to your sensor!



In order for your robot to be able to read the sensor, you will have to reprogram the microcontroller. Sample code to help you get started is available on the Vex website. Refer to the Programming chapter in your Vex Inventor's Guide for information on how to add or change code.

accessories

light sensor kit

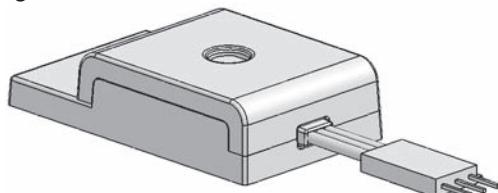
Light Sensor Kit

With a light sensor, you can add a whole new range of capabilities to your robot. Design a simple tracker that follows the beam of a flashlight, or use a light sensor to help your robot to avoid getting stuck under furniture by making it steer away from shadows. Conserve battery power by programming your robot to shut down in the absence of light. You can even give your robot color vision by putting colored filters on different light sensors!

INSERT THESE PAGES
at the **back of the**
Sensor Chapter in your
Vex Inventor's Guide.

**YOU MUST HAVE A
PROGRAMMING KIT
TO USE THIS SENSOR!**

light sensor x 1



**screw x 2
(8-32, 3/8")**



keps nut x 2



Limited 90-day Warranty

This product is warranted by Innovation One against manufacturing defects in material and workmanship under normal use for ninety (90) days from the date of purchase from authorized Innovation One dealers. For complete warranty details and exclusions, check with your dealer.

Innovation One, Inc.
350 North Henderson Street
Fort Worth, TX 76102

11/04

Printed in China
0405



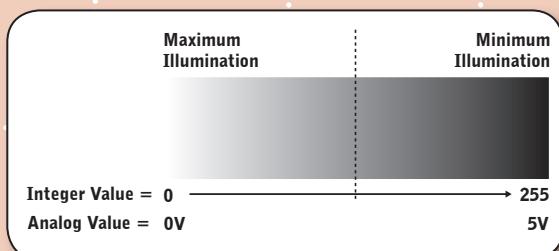
light sensor kit, continued

1 Technical overview

The light sensor uses a Cadmium Sulfoselenide photoconductive photocell, or CdS cell for short. A CdS cell is a photoresistor, meaning that its resistance value changes based on the amount of incident light.

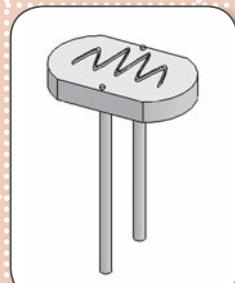
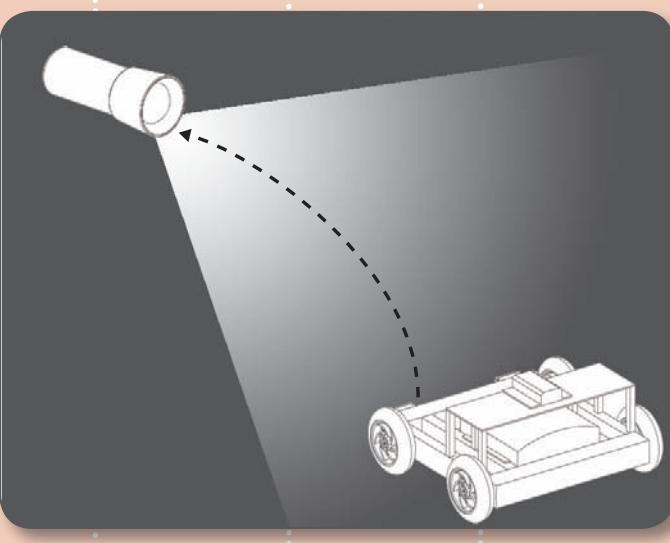
This is an analog sensor, so its output covers a range of values (in this case, from zero to five volts) rather than being only high (five volts) or low (zero volts), as is the case for a digital sensor. This range of outputs from zero to five volts is sent to the microcontroller, which reads it as a range of integer values from 0 to 255. [For more detail, refer to the Sensors chapter in your Vex Inventor's Guide.]

For this particular sensor, a low value (around 0) corresponds to very bright light, and a high value (around 255) corresponds to darkness.



We can then set a threshold value in our code to act as a trigger for behaviors.

From this basic premise, we can build more complicated behaviors. For example, if you have two light sensors on the front of your robot (one on the left, and one on the right), then you can program your robot to follow a bright light by telling it to steer toward bright light (in the direction of the sensor that is receiving low values) and away from darkness (away from the direction of the sensor that is receiving high values).



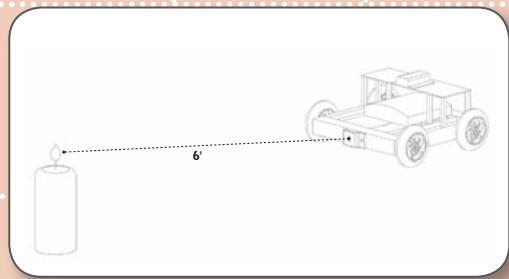
accessories

light sensor kit, continued

1 Technical overview, continued

The light sensor has a usable range of 0 to 6 feet, so it can distinguish a light source from ambient light up to six feet away; a light source more than 6 feet away will blend into the ambient light and be lost. The range is dependent on the intensity of the light source as well as the intensity of the ambient light in the environment. The range will be greater for a very bright point source in a very dark room, but dramatically reduced for a flashlight outdoors on a sunny day.

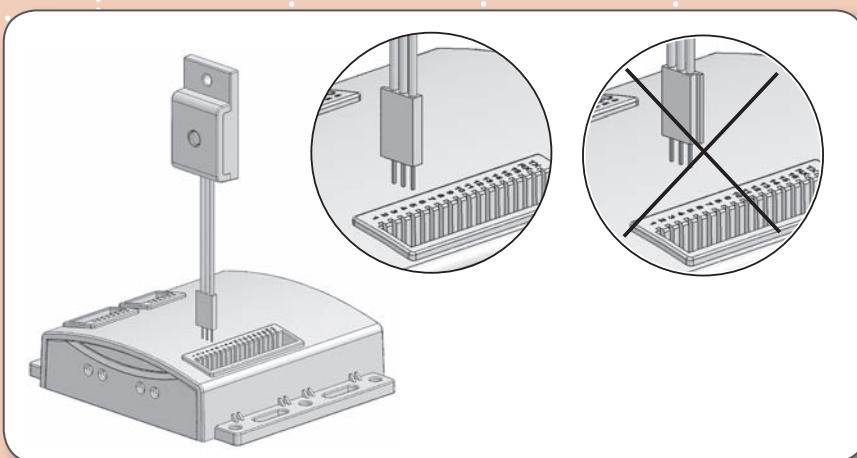
This light sensor is sensitive to visible light only; it will not provide useful data for infrared or ultraviolet sources.



2 Detecting light level:

Reprogramming your microcontroller to read the sensor

Start by plugging your light sensor into any port in the Analog/Digital bank on the Vex Microcontroller. Note that the connector is keyed to fit into the microcontroller port in a specific orientation; plugging it in backwards could damage or even destroy your sensor.



In order for your robot to be able to read the sensor, you will have to reprogram the microcontroller. Sample code to help you get started is available on the Vex website. Refer to the Programming chapter in your Vex Inventor's Guide for information on how to add or change code.

Glossary

Actual measurements: Data that is found by making measurements, as opposed to predictions.

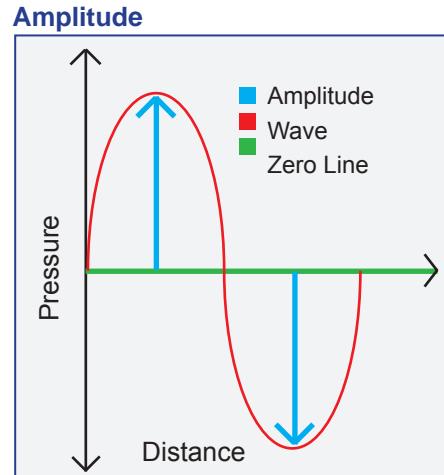
All-Purpose: Usable for a number of different tasks.

Algorithm: A systematic method for solving a certain kind of problem that is guaranteed to always give a correct answer. Sometimes used more generally to mean any well-defined, systematic method of doing something.

Amplitude: The difference between the “highest” or “lowest” point of a wave, and the “rest” or “zero” level. For a sound wave, the difference in air pressure between the most-compressed “peak” areas of the wave and undisturbed air (which is represented as the middle “zero” line on a graph). Note that amplitude is NOT the difference between the highest point and the lowest point on the wave – amplitude is measured from the top to the middle, or the bottom to the middle, but not top to bottom.

Autonomus: Something that can work by itself. Often used as a synonym for “robotic.” For example, an autonomous harvester is one that can harvest without a human operator.

Autonomous Navigation: See Navigation, Autonomus.



Behaviors: Anything a robot “does,” including both observable actions (e.g. move forward for 10 cm) and internal actions (e.g. add 1 to a variable in the program). Complex behaviors are often made of numerous simpler behaviors put together; moving through a maze is a behavior composed of smaller moving and turning behaviors.

Best-fit Line: A straight line that best represents all the data on a graph. It is also known as a trendline. A line of best fit is usually written in a form called slope-intercept, containing two variables, x and y.

Brainstorming: The process of coming up with ideas. Good brainstorming in teams requires team members to encourage different, unexpected and unusual approaches to a problem, and to build on each others’ ideas.

Budget: See Resources. In projects, the amount of money which is available to spend on completing the project, and, by extension, the amount of other resources available. Budgeting money, time and human resources adequately is essential to the successful completion of a project.

Bumper Sensor: A Touch Sensor used as a bumper, activated when a robot bumps into another object. A bumper sensor can be used for obstacle detection.

Calibrate: To set the correct position, value or capacity of something. Calibrating the Sound or Light Sensor will set minimum and maximum values for it.

Caliper: An instrument using converging or diverging arms to determine the external or internal width of an object.

Center of Mass: The “average” location of all the mass in an object. In many cases, you can make predictions about the entire object’s behavior based on the location of its center of mass alone.

Circumference: The distance around the edge of a circle. Equal to diameter times π .

Glossary continued

Code: General term for any command or group of commands in a program.

Comment: A written note in the program that explains something about that portion of the program. Comments do not actually change the way the robot behaves, but are very important to the programmer's ability to remember what the code does.

Commercialize: To bring into the commercial market. In a robotics project, commercialization (if it occurs) begins with the completed robotic prototype and ends with a product which is mass-produced and sold on the commercial market.

Communication: The process of sending and/or receiving information by two or more parties.

Communication, One-way: One-way communication occurs when one of the two or more communicating parties functions only as the sender of information, and the other(s) only as the receiver(s) of information.

Communication, Remote: Remote communication is communication occurring over some distance, and typically by a specialized technology, like Bluetooth.

Communication, Two-way: Two-way communication occurs when all of the communicating parties function as both sender(s) and receiver(s) of information.

Compiler: The compiler is a part of the VEX Programming Software that takes the code in a program and converts them into machine language that the VEX brick can understand and run. The compiled code is not exactly the same as the code written on your computer; this is why you cannot load the program back onto the computer once it is compiled and downloaded to the VEX.

Condition (experimental): A portion of an experiment corresponding to one specific setting of the independent variable. If your experiment involves large wheels and small wheels, for instance, the part of the experiment where you use the large wheels is the "large wheel condition." Condition can also refer to the setting of the variable itself ("large wheels"). See also Conditional Statement (programming).

Conditional Statement (programming): A programming block that chooses to run different pieces of code, depending on some user-defined factor (for example, it may choose to run straight ahead if the robot does not detect an obstacle, but turn to the left if there is a nearby object).

Cross-Multiplication: A mathematical procedure used to solve an equation of the form $X/A=Y/B$ (one fraction equals another fraction, with no other terms outside the fractions). The result of cross-multiplication in the example to the right is $BX=AY$.

Crystal: Crystals are the channels used to send signals from the transmitter to the receiver. The transmitter's frequency module and the matching receiver crystal determine the control frequency for a robot. Each robot operating at the same time should be on a different control frequency.

Data: Factual information, like the weight of a robot or the value of a sensor. Note that the word data is plural. A single piece of factual information is a datum.

Data Analysis: The process of manipulating data to increase understanding of a certain topic or issue.

Data Flow: The process of moving data around inside of a program.

Cross-Multiplication

$$\frac{X}{A} = \frac{Y}{B}$$

$\cancel{X} \quad \cancel{Y}$

$$BX = AY$$

Glossary **continued**

dB and dBA: dB and dBA are modes of the Sound Sensor that refer to different frequency sensitivity settings. When used this way, the terms “dB” and “dBA” are not units! dB mode produces readings that are simply based on how much sound is picked up by the Sound Sensor. dBA mode also gives readings based on the amount of sound detected, but adjusts those readings so that the displayed values more closely match the pattern of human hearing, which is less sensitive to very high and very low frequencies. Just as a human would perceive these sounds to be quieter, dBA mode will give lower readings for those frequencies even if the actual amount of sound is the same.

Decibel: A relative unit of measure commonly used with reference to the amplitude (loudness) of sound. In sound, 0 dB is the quietest sound a person can hear (measured in micropascals, a very small unit of pressure, since sound is based on pressure waves). Every 10 dB increase then means that the sound gets ten times louder. 10 dB is ten times louder than the quietest sound you can hear. 20 dB is ten times louder than that. 30 dB is ten times louder than 20 dB, and is about the level of noise in a quiet room. Note that the Sound Sensor does not measure in decibels, even when its mode is set to dB. Instead, it measures a % value of the loudest sound it can detect.

Demonstration(demo): An event in which a project prototype demonstrates some or all of its capabilities for an audience, which usually includes the project sponsors.

Dependent Variable: In an experimental setup, the variable whose value is measured to see whether it was affected by a change in the independent variable. Also called the responding variable

Design: Both the process of originating and developing a plan for a new object, like a project prototype, and the plan itself.

Design Candidate: An idea selected for evaluation with a group of other ideas, the most applicable of which will become a prototype.

Design Review: A process in which a design or designs are evaluated, usually by the team or experts.

Design Review, External: A process by which outside parties, particularly those who are funding the project, review the concept that has been chosen as the prototype and offer feedback and suggestions.

Design Review, Internal: A process designed to facilitate a fair and efficient comparison of all available design ideas, so that the best one can be chosen for continued development.

Design Specification: A document created to help fully understand the problem before beginning a solution.

Diameter: The distance “across” the center of a circle from edge to edge. Equal to two times the radius of the circle.

Document: A real or virtual written item which provides information. To document means to provide supporting information, usually so as to make something easier to reference or modify. For example, a well-documented program will have all successful versions saved, named and commented descriptively, so that future modifications to the program can be done as efficiently and accurately as possible.

Downloading: Transferring data (usually a compiled program) from the computer to the VEX. See also Uploading.

Driven Axle: When considering a pair of axles connected by gears, pulleys, or other means, the driven axle is the one whose movement is an effect of the other's (rather than the cause). See also Driven Gear, Driving Axle.

Driven Gear: When considering a pair of connected gears, the driven gear is the one whose movement is an effect of the other's. If axles are being considered, the driven gear is the gear on the driven axle. See also Driven Axle, Driving Gear.

Driving Axle: When considering a pair of axles connected by gears, pulleys, or other means, the driving axle is the one whose movement is the cause of the other's. See also Driving Gear, Driven Axle.

Glossary continued

Driving Gear: When considering a pair of connected gears, the driving gear is the one whose movement is the cause of the other's. If axles are being considered, the driving gear is the gear on the driving axle. See also Driving Axle, Driven Gear.

Engineering: The study and application of science, mathematics, and technology to find solutions to real-world problems.

Engineering Journal: A notebook which serves as a personal organizer for a project. It should maintain and order all items related to the project, including research, brainstorming ideas, schedules, daily activities, design reviews, presentations, etc.

Environmental: Of or pertaining to the environment, sometimes the natural world around us and sometimes the area in which the robot will be operating.

External Design Review: See Design Review, External

Feedback: See Input. Input, such as what a sensor gives to the VEX. For example, a robot uses Light Sensor feedback to follow a line. By extension, feedback can also mean human response, both positive and negative. Efficiently gathering and making use of available human feedback, both internal and external, will tend to help the success of any project.

Follow-up Proposal: A second proposal made to continue work begun in the first. A follow-up proposal typically takes into account both the successes and the failures of the original project, and is based on new perspectives gained from it.

Funding: Money made available for a specific purpose, like a project.

Frequency: The number of waves that pass by a point in space in a certain amount of time. For counting purposes, one "wave" is usually considered an entire cycle from peak to peak (i.e. two "tops" of waves pass by) or trough to trough (two "bottoms" pass by). Frequency is usually expressed in hertz (Hz), which is one wave per second.

Gantt Chart: A bar chart that illustrates a project schedule broken into subschedules for each of the tasks needed to complete the project.

Gear Ratio: The number of times the driving axle in a system must spin to make the driven axle turn once. With gears, the gear ratio can be found by counting the number of teeth on the driven gear, and dividing by the number of teeth on the driving gear.

Gear Train: A series of gears that transmit power between axles.

Gimbal: A mechanical device that allows the rotation of an object in multiple dimensions.

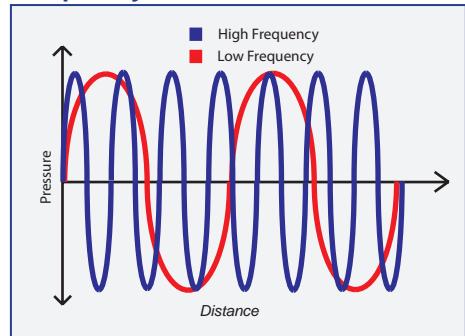
GPS: An acronym which stands for Global Positioning System. A GPS receiver can accurately determine its location (latitude, longitude and altitude) by processing signals sent by more than two dozen GPS satellites.

Graph: A line or curve representing the variation of one quantity with another.

Graphing: Representing data on a graph.

Hertz: Unit of measurement for the frequency of repeating events, defined as one repetition per second. With sound, for instance, frequency is the number of pressure waves that travel past a certain point in a certain amount of time... each time the "peak" of a wave travels by that point, you can count one cycle of the wave. Thus, if ten peaks travel by in one second, the wave has a frequency of 10 hertz. Many waves travel so quickly that thousands of peaks will go by in a second, thus the kilohertz (kHz, 1000 Hz) is a frequently used unit also.

Frequency



Glossary **continued**

Horticulture: Culture or growing of garden plants.

Hypothesis: An educated explanation describing the possible relationship between two or more factors. A good hypothesis is very specific, providing detailed, useful information. A good hypothesis is also testable, meaning that experimentation can help to show whether the hypothesis is correct or incorrect (though it cannot ever conclusively prove correctness).

Independent Variable: In an experimental setup, the variable that is set and changed in different experimental conditions by the experimenter in order to see whether these changes cause a change in the dependent (responding) variable. Also called the manipulated variable

Innovate: Introduce something new, usually to make an improvement.

Input: See Sensor. Something which is sent to the controller which is used in its program. An input is typically a sensor value sent by a sensor. An input may also refer to the sensor itself.

Integration: The process of combining or accumulating, usually in a well-ordered and useful way. To integrate sensor data, for example, would be to combine data from two or more sensors in a useful way. To integrate two parts of a robot would be to combine them into one machine.

Internal Design Review: See Design Review, Internal.

Inventory: An inventory is the total stock available at a given place and time. To inventory means to list systematically the items that are available in a particular place or situation, or for a particular purpose.

Irrigation: Bringing water to crops by human effort (in place of, or in addition to, water coming from rainfall or by natural waterways). An irrigation ditch, for example, is something people dig to channel water onto cropland.

Iterative Development: See Test/Revise/Repeat. The process of repeatedly testing and 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.

LCD (Liquid Crystal Display): A transparent screen containing a light polarizing liquid that is controlled by electric fields to create visible readouts on some controllers.

Light Sensor: A sensor that detects the presence of certain wavelengths of light and reports the intensity of light back to the controller. Light Sensors have two modes: Reflected Light and Ambient Light. In Reflected Light mode, the Light Sensor will shine a red light and look for the amount of that light that bounces back to it off objects in the environment. In Ambient Light mode, the sensor will not shine the light, instead looking for light that reaches it from other sources.

Library: A collection of programs, or parts of programs, stored to help optimize the programming process. Without a library of programs, everything a programmer does must be done from scratch. A well-ordered library, on the other hand, enables a programmer, for example, to track a line by calling a line tracking program from a library, instead of having to place and configure each icon needed to track a line.

LIDAR: An acronym standing for Light Detection and Ranging; or Laser Imaging Detection and Ranging. An optical remote sensing technology which measures properties of light to find range and/or other information of a distant target. The most common method is to send laser pulses into the environment, and determine the distance to various objects by measuring the amount of time they take to reflect back. Other methods, like measuring the frequency of the reflected light, are also used. LIDAR is an important and widely used remote sensing and obstacle detection technology for mobile robots.

Limit Switch: A Touch Sensor used to limit the motion of a moving device like a mechanical arm. Limit switches may be used to provide a precise beginning and end point to mechanical motion.

Glossary **continued**

Linear Regression: See Best-fit Line. The linear function referred to in the phrase linear regression may also be a best-fit line, or trendline. Regression, in general, is the problem of estimating a conditional expected value.

Logic: A data type that the controller can understand. Logic data has only two possible values, which can be represented in multiple ways. The value True is often also represented by a 1 or a checkmark, and False is also represented by a 0 or an X.

Manipulated Variable: See Independent Variable.

Mapping: Making a map. In robotics, mapping usually refers to the ability of an autonomous robot to enter an area, and navigate, sense and record information in such a way as to allow an accurate map of the area to be constructed.

Marketing: Refers to the process of advertising and selling a product or service. Marketing also refers to bringing the product or service to the market, where it can be bought or sold.

Marketing Presentation: A marketing presentation is one concerned with marketing a product or service, usually to potential buyers.

Miner, Continuous: In the mining industry, a large tracked machine that uses moving claws to tear coal loose from its natural formation and pull it into a large scoop. The continuous miner then transfers the coal to other coal moving machines.

Mining: The extraction of valuable minerals from the earth. The area where minerals are extracted is called a mine. There are, for example, coal mines, iron mines, gold mines and uranium mines.

Mining, Longwall: Longwall mining is a form of underground coal mining where a long wall (about 250-400 m long typically) of coal is mined in a single slice (typically 1-2 m thick). The longwall “panel” (the block of coal that is being mined) is typically 3-4 km long and 250-400 m wide. The longwall equipment consists of a number of hydraulic jacks, called chocks, roof supports or shields, which are placed in a long line up to 400 m in length in order to support the roof. An individual chock can extend to a maximum cutting height of up to 5 m. The coal is cut by a rotating drum with bits called a shearer that moves along the length of the face in front of the chocks, disintegrating the coal. Continuous miners are used in longwall mining primarily to open the spaces needed for the longwall machine and chocks.

Monitor: See Receiver, Sender. To watch, or to receive information.

Motion Detector: A sensor that detects motion. It may be quite simple or very complex, and may use a wide variety of sensors, separately or together.

Multitask: To do more than one thing at once. In the VEX programming software, multitasking is accomplished by creating a program with more than one task.

Navigation: Directing a vehicle from one place to another.

Navigation, Autonomous: The ability of a robot to get to a pre-determined place without human intervention, sometimes despite the presence of unknown obstacles, or from an unknown starting point.

Number: Another word for data type that the VEX can understand.

Obstacle Detection: The ability of a robot to detect obstacles in its environment. LEGO robots usually detect obstacles using a Touch Sensor, an Ultrasonic Sensor, or both.

One-way Communication: See Communication, One-way.

Glossary continued

Output: Something which the controller sends. An output is typically power sent to a motor. An output may also refer to the motor itself, or to sensor values that are displayed or collected in a file.

Peak: The “top” of a wave on a graph. The point of greatest disturbance from the “rest” state in one direction. See also “trough”.

Percent Error: The percentage that the measured value differs from the calculated value, which can be determined by the formula $(\text{calculated value} - \text{measured value})/\text{calculated value} \times 100\%$

Perpendicular: Intersecting at a 90 degree angle.

PERT Chart: An acronym standing for Program Evaluation and Review Technique, it is a method for analyzing the tasks involved in completing a given project in order to identify the minimum time needed to complete the total project.

Point Turn: A turn where one wheel rotates forward and the other rotates backward, causing the robot to sit and spin in place. Also called a “skid turn” in general robotics.

Ports: The designated areas for connecting sensors and/or motors to the controller. A wire of appropriate length should be used to connect the controller with each sensor or motor device.

Problem Context: The overall dimensions of the problem, which include exactly what the problem consists of, what person or group wants the problem solved, what places, situations or people the solution has to function in or with, what commercial markets, if any, the solution will be marketed in, and what previous efforts have been made to solve the problem.

Project: A project is a temporary endeavor undertaken to create a unique product or service. A project is typically worked on by a project team, for a particular customer, client or sponsor who funds it, and has certain goals, a schedule and a budget.

Project Management: The process of managing a project. Good project management includes breaking down the project work into tasks, assigning responsibilities for them to team members, ensuring that adequate communication takes place both within the team and between the team and any other groups or people involved, scheduling the tasks to ensure the project meets its deadline(s), and budgeting resources in such a way that all necessary work can occur before any of the resources run out.

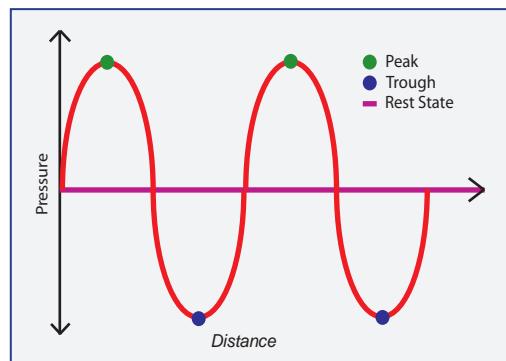
Project Manager: The member of the project team who manages the project, especially the schedule and budget.

Proposal: See Request for Proposals. A properly accomplished proposal educates the prospective client about the full nature of his or her need, and argues as well as possible that the proposal writer’s group can meet it. Sponsors sometimes issue a Request for Proposals, commonly referred to as an RFP, in order **Protractor** to invite proposals on a particular subject.

Prototype: Literally, first of its kind. Creating a working prototype, that is, creating a first-of-its-kind robot that accomplishes the task(s) it is meant to, is the typical goal of a robotic project.

Protractor: A device used for measuring angles.

Peak



Glossary **continued**

Pseudocode: is a compact and informal description of a computer programming code. It is a hybrid language which combines the features of the programming language with the native language of the person writing the program.

RADAR: See LIDAR. An acronym standing for Radio Detection And Ranging, it is a system that uses radio waves to detect, determine the distance and/or speed of objects. It functions by bouncing radio waves off of objects and timing how long it takes the reflected wave to come back to a receiver.

Radius: The length of the line segment that joins the center of a circle with any point on its circumference. Equal to half the diameter of the circle.

Receiver: See Monitor, Sender. A receiver gets messages sent by the transmitter.

Request for Proposals (RFP): See Proposal. Clients sometimes issue a Request for Proposal, commonly referred to as an RFP, on a particular subject. These RFPs are issued to find a provider for a new product or service, one that typically addresses a particular problem or set of problems.

Research: The process of gathering information pertaining to a subject. In a project, thorough research investigates the nature, limits and complicating factors involved in the problem the project seeks to solve, and all previous attempts to address it. Lack of adequate research is often a cause for a project failing to meet its goals.

Resource: Any limited good that can be drawn upon to complete a project. Projects typically are limited in money, time and human resources, and need a plan to allocate them carefully in order to meet project goals.

Responsibility Matrix: After the project work has been divided into tasks, a responsibility matrix may be created which assigns responsibility for these tasks to project members. A well-constructed responsibility matrix will ensure both that no tasks necessary for successful completion of the project are left undone, and also that there is no duplication of effort (more than one member, or team of members, working on the same task in an uncoordinated way.)

Responding Variable: See Dependent Variable.

Revise: See Iterative Development, Test/Revise/Repeat. To remake or improve based on testing and or feedback. Test/revise/repeat is the typical project pattern. Testing reveals both what works and what doesn't. Revision maintains what works while trying to fix what doesn't work. Revision is done after feedback from people as well as testing.

Right Angle: A 90 degree angle.

Robot: A machine that is able to interact with and respond to its environment in an autonomous fashion. A robot is characterized by three central capabilities: the ability to Sense, the ability to Plan, and the ability to Act. See Sense, Plan, Act.

Rotation Sensor: A device that measures the amount of rotation of a certain piece or object.

Rubric: A method and tool for evaluation, consisting of a chart of criteria for evaluation of work. It allows for standardized evaluation according to specified criteria, making evaluation more transparent.

Scatterplot: Used to illustrate the relationship between two aspects of the same set of data. One aspect is represented by the X coordinate (horizontal location) and the other is represented by the Y coordinate (vertical location) of the data point

Glossary **continued**

Scanning: See Mapping. The act of examining sequentially, part by part. Scanning may be part of an automated process which searches for either a single object or a type of object. It may also be part of the process whereby mapping is accomplished. In either case, the robot scans part of the area to be examined, gathering sensor input allowing a part of a map to be constructed, or for the presence or absence of the desired object(s) to be determined. Then it proceeds to the next part of the area, continuing in a systematic fashion until all parts of the area have been scanned and a map of the entire area can be constructed, or the presence or absence of the desired object(s) in the area can be determined.

Schedule: In project management, a schedule consists of a list of a project's terminal elements, or deadlines, with intended start and finish dates. Gantt and PERT Charts are important project management scheduling tools.

Scientific Inquiry: The process by which scientists seek to ask and answer questions about the world. Evidence, models, and logical explanation are all key parts of the process. Inquiry is NOT a rigid series of steps, but rather a fluid cycle of proposing, examining, and revising explanations to find the best answer to a question.

Sender, Remote: Used to sense something at a distance. Remote communication may be used to transfer the remote sensor's data.

Sense-Plan-Act: The three characteristic capabilities that define a robot. The robot must be able to "Sense" things about its environment, it must be able to "Plan" an appropriate response to those factors, and it must be able to "Act" accordingly.

Sensor: A device that detects some important physical quality or quantity about the surrounding environment, and conveys the information to the robot in electronic form.

Slope Intercept: See Best-fit Line. A slope-intercept equation contains two variables, x and y, usually in the form $y = mx + b$. The quantity m describes the slope of a line on a graph, and b describes the y-intercept.

Sound Sensor: A sensor that detects sound waves and reports the amount of sound back.

Sound wave: A "moving" pattern of high and low pressure in air (or other medium), perceived as sound.

SPA: See Sense-Plan-Act.

Specifications: A specification is a set of requirements. Normally, a specification is the specific set of (high level) requirements listed by the sponsor, which a project must meet. A project specification may be that a robot can be no bigger than 2 feet long, or that it must be able to travel at over 30 km an hour, for example.

Speed: The rate at which an object is moving. The rate of change an object's position over time. Calculated by dividing the distance an object moves by the amount of time it took to move.

STEM: Acronym for the closely related fields of Science, Technology, Engineering, and Mathematics.

Stimulus: Anything that may have an impact on a system; an input to the system. A program may wait for a given stimulus (like the Touch Sensor being pressed) to execute a specific behavior (like stopping).

Stimulus-Response: Action made in response to a stimulus. The stimulus-response cycle enables robots to interact effectively with their environment.

Strategy/Strategize: A master plan for accomplishing certain goals. To strategize means to come up with a plan to meet a goal.

Support Polygon: The imaginary polygon formed by connecting all the points where an object touches the ground. This polygon marks the boundaries of where the object's Center of Mass can be while remaining stable. If the Center of Mass is not directly over the interior of the Support Polygon, the object will fall over.

Glossary **continued**

Swing Turn: A turn where one wheel rotates and the other stays in place, causing the robot's body to "swing" around the stationary wheel.

Task: A discrete unit of work. A task may refer to a human activity like designing a drivetrain, or a robotic one, like mapping a coal mine. Effectively breaking down tasks is a key to success both for a human project and for a robotic program

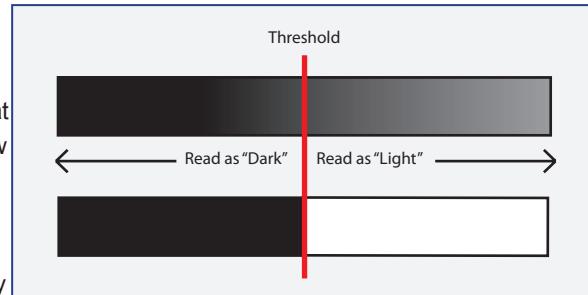
Teamwork: The process of working together in team. Effective and cooperative teamwork is an essential quality of a successful project.

Test/Revise/Repeat: See Iterative Development, Revision. Test/revise/repeat is the typical project pattern. Testing reveals both what works and what doesn't. Revision maintains what works while trying to fix what doesn't. Repeat means that testing and revision are done continuously.

Text: Text includes words, letters, numbers, punctuation, spaces or a combination of all. This data type is often also called a "string."

Theoretical Measurement: A predicted value for a measurement. Usually, these predictions are made by taking a real measurement, then using a hypothesis or theory (hence the name "theoretical") to predict what the value should be under slightly different conditions. After this prediction is made, the real measurement is usually taken, and compared against the theoretical one to see how well the prediction matched the real outcome.

Threshold: A "cutoff" or dividing line between two regions. One common use for thresholds is to divide the hundreds of possible sensor readings from a sensor (a Light Sensor can give a value anywhere from 0-100, for example) into two manageable categories. For the Light Sensor, this would mean setting a threshold value somewhere between 0 and 100, then declaring that all values above the threshold are now "light" while all values below the threshold are now "dark." A light sensor reading can then be easily categorized and handled appropriately. The threshold value can be chosen in any way desired, but it is conventional to choose a value exactly halfway between two known extremes (e.g. halfway between a very dark surface and a very light one).



Time Management: The process of managing time, a limited resource in any project. Project time management tools are schedules, timelines, Gantt Charts and PERT Charts.

Timeline: A visual representation of a process or series of events. It helps chart the progress of the project.

Torque: Roughly speaking, torque is the rotational equivalent of force. Whereas force causes an object to speed up or slow down its (linear) motion, torque causes an object to speed up or slow down its rotation. A motor that generates more torque will let the robot speed up or slow down more rapidly, as well as handle larger tires, heavier loads, and steeper inclines.

Touch Sensor: A sensor that detects physical contact (touch) and reports back to the controller whether its contact area is being pushed in or not.

Trendline: See Best-fit Line.

Trough: The "bottom" of a wave on a graph. The point of greatest disturbance from the "rest" state in one direction (the one that corresponds to "downward" on the graph). See also "peak".

Glossary continued

Two-way Communication: See Communication, Two-way

Ultrasonic Sensor: A sensor that measures distance by emitting ultrasonic sound waves, then measuring how long it takes them to echo back off of objects or surfaces in the environment. The Ultrasonic Sensor then reports the calculated distance back to the controller.

Uploading: Transferring data (usually gathered data) from the robot controller to the computer. See also Downloading.

Variable (mathematics): A stand-in for a not-yet-known value in a mathematical equation. Once a variable's value has been found, the value can be substituted anywhere in place of the variable.

Variable (experimental): A factor that is either manipulated or measured during the course of an experiment. See also Independent Variable, Dependent Variable.

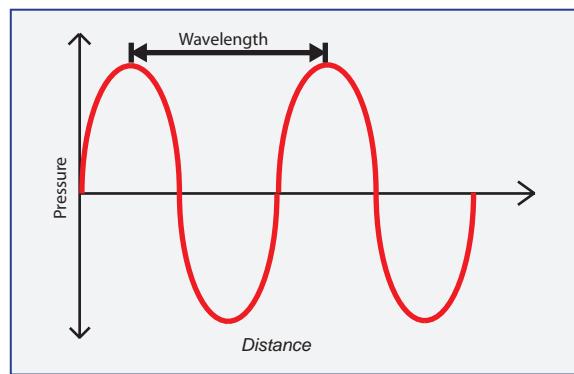
Variable (programming): A “container” for a value. The programmer may choose to store a value (perhaps a sensor reading) in the variable, and use it in a later operation (display it to the screen at the end of the program, for instance). The programmer may also choose to perform mathematical operations on the stored value, such as adding 1 to it.

Wavelength: The distance between successive equivalent points on a wave. For example, the distance between two neighboring peaks on the wave.

While Loop: is a control flow statement that allows code to be executed repeatedly based on a given boolean condition.

Worm Gear: A special type of gear that uses a screw-like shaft and a wheel with slanted teeth to reduce speed or transmit torque between nonparallel axles. Also has the special properties of being a “one-way” gear and having effectively only one tooth for gear ratio calculations.

Wavelength



Glossary

#

12 mode – Control Subsystem

A Transmitter driving mode where axes 1 and 2 are used to control the primary navigation of the robot. Also called Arcade-style controls.

23 mode – Control Subsystem

A Transmitter driving mode where axes 2 and 3 are used to control the primary navigation of the robot. Also called Tank-style controls.

4WD

Short for Four-Wheel Drive. A four-wheel drive robot typically has four wheels, all of which are powered independently. This usage is analogous, but not identical, to the meaning of the term with respect to automobiles.

A

Acceleration – Motion Subsystem

In physics, acceleration is the change in velocity of an object over time. In robotics, acceleration usually refers to the ability of a robot to speed up or slow down quickly on demand.

Actuator – Motion Subsystem

A term commonly used in industry to describe a mechanical device used for moving or triggering a mechanism.

Alkaline (Battery) – Power Subsystem

A class of battery chemistries commonly used in disposable batteries. This type of battery is not suited for use in robotics applications.

Allen Wrench

An L-shaped tool used to work with hex screws.

Analog Sensor – Sensor Subsystem

Analog sensors communicate with the Microcontroller by sending an electrical voltage that varies between 0 and the maximum voltage.

Analog/Digital Port Bank – Logic Subsystem

A group of ports on the Microcontroller used for analog and digital communication with other parts of the robot system.

Arcade-style Controls – Control Subsystem

A driving mode in which the robot is controlled with one joystick on the controller, like an arcade game. Also called 12 mode because axes 1 and 2 are being used to drive the robot.

Attachment – Structure Subsystem

Generally, any piece that is “attached” and not fundamentally part of the basic robot design. Usually refers to such pieces as arms or sensor modules, especially if they are removable.

Autonomous – Logic Subsystem

Technically, a robot must be able to function entirely without human supervision to be considered fully autonomous. Almost all real-world robot systems are designed instead to work with partial autonomy under varying degrees of human supervision.

Autonomous Mode – Logic Subsystem

The VEX robot has a simple pre-programmed autonomous mode that uses two bumper or limit switch sensors to detect obstacles as the robot wanders around a room or course.

Axis (Joystick) – Control Subsystem

One of two axes (X and Y) along which a joystick can move. Each axis on the joystick is associated with an onboard potentiometer that measures the joystick’s position along that axis.

Axis of Rotation – Motion Subsystem

The imaginary line around which a spinning object rotates. This usually coincides with the axle for a wheel or gear.

Axle – Motion Subsystem

A long, rigid piece through the rotational center of an object (like a gear or wheel). Axles serve two main purposes: to hold spinning bodies in place relative to the rest of the structure, and to transfer rotational motion from one spinning piece to another (as in the case of a motor axle turning a gear). Square bars are usually used as axles in the VEX system.

Glossary

B

Back-driving – Motion Subsystem

A condition where torque is transferred backwards through a mechanical system, causing the driving element (typically a motor) to be driven instead. This can often be damaging to the mechanical system and/or the motor. A clutch can be used to disengage the motor if the back-driven force is strong enough to cause damage.

Battery – Power Subsystem

Normally, any portable power source, such as the VEX battery packs. Technically, a battery is a collection of multiple cells, but single cells are often referred to as batteries in common usage.

Battery Holder – Power Subsystem

The Battery Holder creates a 7.2V battery out of (6) 1.2V AA cells (by connecting them in series) or 9.6V out of (8) 1.2V AA cells. The Battery Holder also holds the AA cells in place on board the robot.

Bearing – Structure Subsystem

A piece that is used to hold a moving piece (such as an axle) in place relative to the rest of the system.

Bearing Flat – Motion Subsystem

A commonly used type of bearing in the VEX system. This bearing has three holes in a row. The bearing is secured to the chassis through two of the holes, and an axle is passed through the third, which allows it to spin freely but not move out of place relative to the chassis.

Behavior – Sensor Subsystem, Logic Subsystem

In the context of robotics, a behavior is the pattern of actions a robot will enact when given certain inputs or commands.

Bumper Switch Sensor – Sensor Subsystem

A high-durability sensor designed to detect physical contact. This is a digital sensor.

C

Calibrate (Sensor) – Sensor Subsystem

Calibrating a sensor is the process of matching sensor readings against known values to ensure that the sensor input is being interpreted correctly in the program. Simple sensors, like the Bumper and Limit Switches, typically do not need to be calibrated.

Calibrate (Joysticks) – Control Subsystem

Calibrating the transmitter joysticks (also called “trimming” the sticks) is the process of adjusting trim values on the Transmitter to ensure that the sticks produce no motor movement when they are centered. A more thorough calibration process also includes setting the scaling and end points to ensure a full range of motion.

Carrier Wave – Control Subsystem

The carrier wave for FM communication is a simple sine wave with a set frequency. It is then modified (modulated) by the desired signal wave to produce the final output wave that is sent to the receiver.

Caster Wheel – Motion Subsystem

A free-swiveling wheel mounted on a robot to provide stability while producing a minimum of friction. The front wheels on a shopping cart are caster wheels; they support weight and stabilize the cart, but do not add significant amounts of friction like a skid would, nor do they change the maneuvering characteristics of the cart like an additional locked wheel would.

CCW

Short for Counterclockwise.

Cell – Power Subsystem

A single electrochemical unit producing a known voltage differential, such as a single NiCd AA battery, which has a voltage of 1.2V between the + and - terminals.

Glossary

Center of Gravity – Structure Subsystem

The robot's center of gravity is the average position of all the mass on the robot (technically, this is the center of mass, but under terrestrial gravity conditions, they are the same). It is critical that the center of gravity be kept directly over the support polygon, or the robot will fall over.

Challenge – VEX System

VEX Challenges are designed to give you a specific task to accomplish by building a robot, and to open possibilities for collaboration and competition with other robot designers.

Chassis – Structure Subsystem, Motion Subsystem

A vehicle's basic structural frame, plus its locomotion systems. In the VEX system, this is generally the Structure Subsystem plus the Motion Subsystem, minus any attachments.

Circumference

The distance around the edge of a circle. This quantity is equal to pi times the circle's diameter, or 2 times pi times the radius.

Clockwise

A rotational “direction” that prescribes turning in the same direction as the hands on a clock normally turn.

Clutch – Motion Subsystem

A detachable piece normally mounted to the VEX motors that protects them from shock loads. These should not be removed under most circumstances.

Collar – Structure Subsystem

A type of spacer that can be set to remain stationary at any given point along an axle. These are often used to keep other components on the axle (or sometimes the axle itself) from sliding out of position.

Compound Gear – Motion Subsystem

A system of gears involving several pairs of gears, some of which share axles with each other. When calculating gear ratio, this whole system of gears behaves as if it were a single gear pair with a gear ratio that might not otherwise be achievable.

Compound Gear Ratio – Motion Subsystem

The overall equivalent gear ratio produced by a group of gears in a compound gear configuration. This can often be quite high or quite low, due to the multiplicative nature of gear ratios in a compound gear system.

Configuration (Transmitter) – Control Subsystem

One of the 6 different control setups stored on the RF Transmitter. Each configuration saves the Transmitter menu settings that were set while using that configuration number.

Control Subsystem

The subsystem responsible for collecting human operator input and communicating it to the Microcontroller.

Control Channel – Control Subsystem

One of the 6 pathways for control information traveling from the Controller to the Microcontroller. The X-axis of the right joystick, for instance, sends its data over control channel 1 (that axis of the stick itself is sometimes referred to as Channel 1 as a result). Note that control channels are not the same as radio channels.

Counterclockwise

A rotational “direction” that prescribes turning in the opposite direction from the way the hands on a clock normally turn.

Crystal (Radio Frequency) – Control Subsystem

One of the crystals that determine the frequency on which the Transmitter and Receiver operate. The crystal used in the Transmitter Frequency Module must match the crystal used in the Receiver for controls to be sent and received properly.

CW

Short for Clockwise.

Glossary

D

Deep Cycling – Power Subsystem

Draining a battery down to very low power (below the normal cutoff levels) before recharging it. This will wear a rechargeable battery out very quickly, and should be avoided if possible.

Diameter

The distance from one point on a circle to the point directly across from it. This quantity is equal to two times the radius, or it can be multiplied by pi to find the circumference of the circle.

Digital Sensor – Sensor Subsystem

Digital sensors communicate with the Microcontroller by setting an electrical voltage in the system to one of two values: either a digital LOW equal to 0V, or a digital HIGH equal to the maximum voltage on that port.

Discharge Cycle – Power Subsystem

Technically, any period during which power is drawn from the battery and then recharged. Usually used in one of two contexts: either when referring to the usage pattern of a battery (using a battery for a short time, then recharging it constitutes a pattern of short discharge cycles), or when battery chargers automatically drain the battery before recharging it (the charger performs a “discharge cycle” on the battery).

Drive Train – Motion Subsystem

All the parts involved in the primary locomotion system of a robot, including the motors, gears, axles, and wheels.

Driven Gear – Motion Subsystem

In a gear train, the last gear being turned. Usually, this gear shares an axle with a wheel.

Driving Gear – Motion Subsystem

In a gear train, the gear that provides the energy to turn all the other gears and their connected components. This gear usually shares an axle with a motor.

Driving Mode (Transmitter) – Control Subsystem

The driving mode selected on the Transmitter through the DRIVE menu, either “23 mode” or “12 mode.” This setting (together with Jumper 14 on the Microcontroller) determines which combination of joystick axes on the Transmitter will control the robot’s movement.

E

Electromagnetic Waves – Control Subsystem

Technically, a time-varying electric field that propagates through space at the speed of light, caused by the acceleration of a charged particle. More simply, an electronically controllable wave that travels at the speed of light and can carry information between two points through a variety of encoding techniques.

End Points (Joystick) – Control Subsystem

End points control the percentage of the full power command that will be sent by the Transmitter when the joystick is pushed to the edge of its movement area.

Exponential Scaling – Control Subsystem

A control scaling method that allows for “stiffening” or “softening” of the feel of the joystick controls by causing the output command value to increase faster or slower than it normally would as the joystick is moved away from the center of its movement area.

F

Fastener – Structure Subsystem

A general term for pieces (such as screws) whose primary purpose is to hold two or more other components together.

Floating – Structure Subsystem

As opposed to “locked.” Moving freely, not held in one specific place. A collar floats freely on a square bar when the screw is not tightened (it slides easily up or down the bar).

Flush – Structure Subsystem

As in “flush against another part.” Pushed up against something, leaving no space between them. A collar is flush against a bearing when it is pushed up against the bearing as far as it can go.

Glossary

Four Wheel Drive – Control Subsystem

A four-wheel drive robot typically has four wheels, all of which are powered independently. This usage is analogous, but not identical, to the meaning of the term with respect to automobiles.

Frequency-Modulated Signals – Control Subsystem

Frequency-Modulated (FM) signals are used in the VEX system to encode data in radio transmissions. Radio waves are a form of electromagnetic wave with a very high frequency. The frequencies used by the VEX system all have a carrier frequency near 75MHz, which is part of the VHF (Very High Frequency) band of the electromagnetic spectrum. This carrier wave is then modulated by the signal wave to produce a third wave, which is transmitted through the air and received by the RF Receiver Module on the robot.

Friction – Motion Subsystem

The force between two touching surfaces moving at different speeds that acts to slow their movement relative to each other. In robotics, this usually has one of three contexts: friction between wheels and ground that results in rolling wheels slowing down, friction between wheels and ground that allows wheels to “push off” and start moving to begin with (rather than spinning in place), and friction between any two components rubbing together in the robot that result in loss of energy.

G

Gear – Motion Subsystem

Essentially, gears are spinning discs with teeth that prevent them from slipping past each other. Gears are frequently used to transfer rotational motion from one piece to another, and to provide mechanical advantage while doing so. The number of teeth on a gear (assuming the same spacing between teeth on both gears, so their teeth mesh properly) is directly proportional to the gear disc’s circumference, thus the number of teeth can easily be used to calculate the gear ratios of gear trains.

Gear Ratio – Motion Subsystem

The mechanical advantage, or “force multiplier” generated by a group of 2 or more gears turning together. For simple non-compound gear trains, this can be calculated as the number of teeth on the driven gear divided by the number of teeth on the driving gear.

Gear Train – Motion Subsystem

In general, a group of gears that turn together to transmit motion from one point to another on the robot, often providing mechanical advantage along the way.

Gripper

An attachment designed to pick up or hold an object, often by “gripping” it with claw-like appendages.

Gusset – Structure Subsystem

A piece used to strengthen an angled joint.

H

HIGH (Digital value) – Sensor Subsystem

One of two possible values in a digital system (the other is **LOW**). The voltage used to indicate **HIGH** usually corresponds to the maximum voltage of the system.

Hub – Motion Subsystem

With wheels, the hub is the center portion of the wheel that joins to the axle.

I

Idler Gear – Motion Subsystem

A gear in a gear train that is neither the driven nor the driving gear, and does not share an axle with another gear in the train (i.e. does not form a compound gear). Each idler gear in the train reverses the direction of spin once, but never affects the gear ratio.

Glossary

Interrupt Port Bank – Logic Subsystem

A port bank on the Microcontroller used primarily for advanced programming functions.

J

Jumper – Control Subsystem, Logic Subsystem

A metal wire contained in a plastic housing that can be placed (and removed) by hand to complete (make) an electrical connection. These are most often used to “set” options on the Microcontroller by placing them in ports in the Analog/Digital Port Bank. Placing a jumper in one of these ports closes a circuit, setting the voltage for that port’s input value, just like closing a limit switch sensor would.

K

Keps Nut – Structure Subsystem

A variant of the standard nut that includes a toothed “crown” designed to bite into a mounting surface and prevent the nut from slipping. Nuts are used to allow a screw to function as a fastener when the actual component being fastened does not include its own threading.

Keying (connectors) – Logic Subsystem

An intentionally asymmetrical construction of a connector to prevent backwards insertion. The power port on the VEX Microcontroller is keyed (the two plastic shapes in the middle are not the same), for instance, so that the power plug cannot be inserted upside-down. Keyed connectors are sometimes called polarized connectors.

L

Lever – Structure Subsystem

One of the six “simple machines” that provides a mechanical advantage. There are three main classes of levers with subtle differences, but in general, long pieces that rotate around any point on their length will function as levers and can provide mechanical advantage.

Limit Switch Sensor – Sensor Subsystem

A small, contact-sensitive sensor that is most often used for internal regulation of movement, and should not be exposed to high-impact conditions. This is a digital sensor.

Linear Scaling – Control Subsystem

A control scaling method that allows for control of the overall range of motion and sensitivity of the joysticks on the Transmitter.

Logic Subsystem

The subsystem responsible for onboard robot operation, allocation of power, processing sensor feedback, and interpretation of human operator control.

LOW (Digital value) – Sensor Subsystem

One of two possible values in a digital system (the other is HIGH). The voltage used to indicate LOW usually corresponds to the zero (ground) voltage of the system.

M

Master Channel – Control Subsystem

In a Programmable Mix, the Master Channel is the control channel that, when manipulated by the operator, will also affect the value on the designated slave channel.

Mechanical Advantage – Structure Subsystem, Motion Subsystem

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.

Glossary

Memory Effect – Power Subsystem

Technically, the phenomenon where a rechargeable battery that is repeatedly discharged to the exact same level and then recharged will develop a permanently diminished capacity. True memory effect is observed only under laboratory conditions and on board solar-powered satellites in space. The more common usage of the term is incorrect, and is frequently used mistakenly to refer to voltage drop.

Microcontroller – Logic Subsystem

The “brain” of the robot. The Microcontroller contains the robot’s program and processes all signals received from both human operators and onboard sensor systems. It also manages power allocation on board the robot, and directly controls the motors.

Miscalibrated – Control Subsystem

A condition where two values which should be the same do not, in fact, match each other. This occurs frequently with the joysticks on the Transmitter, which should produce a neutral motor state when centered, but will often cause motors to turn slowly instead when the sticks are released. Sensors that indicate things like distances can also become miscalibrated, and report values that do not reflect the actual physical situation.

Mix (Transmitter) – Control Subsystem

A control setup where inputting commands on one control channel influences the commands being sent on other control channels.

Motion Subsystem

The subsystem responsible for the generation and transmission of physical motion on the robot. This includes motors, gears, wheels, and many others.

Motor (Electric) – Motion Subsystem

An electromechanical device that converts electrical energy into kinetic (physical) energy on demand. The motion generated by a motor is almost always rotational in nature, and may need to be mechanically redirected before it can be used to produce the desired effect.

Motor Port Bank – Logic Subsystem

The port bank on the Microcontroller where the motors or servos are plugged in. The motors/servos receive both commands and power through these ports.

Motor Shaft – Structure Subsystem, Motion Subsystem

A carried-over term from automotive engineering, this usually refers to the axle (square bar) that is directly driven by the motor.

Mounting Point – Structure Subsystem

Any place where a component can be conveniently attached. An open spot on the front bumper, for instance, may serve as a good mounting point for a forward-facing sensor.

N

NiCd (Nickel-Cadmium) – Power Subsystem

The preferred battery chemistry for the VEX Robotics Design System for performance reasons. A NiCd (pronounced Nai-kad) battery is an electrochemical cell which uses Nickel metal as its cathode material, and Cadmium metal as its anode material. Cadmium is highly toxic, and should not be disposed of in the trash (call 1-800-8-BATTERY).

Nut – Structure Subsystem

Nuts are used to allow a screw to function as a fastener when the actual component being fastened does not include its own threading. A screw and a nut “sandwich” the parts that are being fastened, and hold them together. The nut provides threading for the screw to lock into when none is present otherwise.

O

Overcharging – Power Subsystem

Continuing to apply a charging voltage to the battery after it has reached full capacity. This is very likely to damage your battery, and can be dangerous, as the battery will heat up rapidly while being overcharged, and may even explode if it gets too hot. Be sure your charger has the appropriate safeguards so that it will not attempt to overcharge your batteries.

Glossary

P

Parallel (Batteries) – Power Subsystem

A battery arrangement where multiple battery cells are hooked up so that they provide the same voltage as a single cell, but drain power evenly across all the cells, thus behaving similarly to a single cell with a very large capacity.

Pivot – Structure Subsystem

A structural component that provides a mounting point for another component, but rather than locking it in place, the pivot allows the attached component to swivel or turn along a specific arc.

Potentiometer – Sensor Subsystem

An analog sensor which measures angular position.

Power Subsystem

The subsystem responsible for storing and delivering electrical energy to the robot systems.

Programmable Mix – Control Subsystem

A feature of the Transmitter that allows the operator to designate one master channel and one slave channel to be used in a configurable control mix.

R

Radio Channel – Control Subsystem

A shortened name for a radio frequency. Radio frequencies often have long names, so they are given “channel” designations to be used as shorthand. 75.410MHz, for instance, is referred to as Channel 61.

Radio Frequency – Control Subsystem

A designated carrier frequency for radio transmission. Each transmitter-receiver pair should operate on its own radio frequency, and hence transmit data that will not interfere with other signals in the air. The radio frequency for a transmitter-receiver pair is determined by the frequency crystals installed in both devices.

R

Radius

The distance from the center of a circle to the edge. This quantity is equal to half the diameter, or it can be multiplied by two times pi to find the circumference of the circle.

RBRC – Power Subsystem

Rechargeable Battery Recycling Corporation. A non-profit organization that facilitates the collection of rechargeable batteries for recycling, because rechargeable battery chemicals (such as the cadmium in NiCd batteries) tend to be very harmful to the environment when thrown in the trash.

<http://www.rbrc.org>

RF Receiver Module – Control Subsystem

The Control Subsystem component that receives and decodes FM radio signals that are sent by the Transmitter. After decoding the signals, they are passed on to the Microcontroller.

RF – Control Subsystem

Short for Radio Frequency, but often used to refer to any system or component that deals with radio transmission in any way (e.g. RF Receiver).

S

Screw, Hex – Structure Subsystem

A screw with a hexagon-shaped hole in the head, allowing the screw to be tightened or loosened with a hex L wrench.

Sensor Subsystem

The “eyes and ears” of the robot. Electromechanical devices that can detect specific things about the robot and its environment, and communicate that information to the Microcontroller through an electrical signal.

Series (Batteries) – Power Subsystem

A battery arrangement where multiple battery cells are hooked up so that their voltages are added together, thus behaving similarly to a single battery with a much higher voltage.

Glossary

Servomotor – Motion Subsystem

An electromechanical device that converts electrical energy into kinetic (physical) energy on demand. The difference between a standard motor and a servomotor is the way they respond to joystick commands. A motor will spin continuously in one direction or the other, whereas a servomotor will turn to face a specific direction within a limited arc.

Signal Wave – Control Subsystem

In radio transmission, the signal wave represents the data that is being sent, converted into a wave form in order to be included in an FM transmission.

Skid – Motion Subsystem

A non-wheel piece which rests on the ground and provides support for the robot, but is intended to slide when the robot moves. Skids provide support and stability without fundamentally altering the way the robot maneuvers, but they can cause significant friction, and often wear out quickly. Caster wheels are the preferred alternative in most cases.

Slave Channel – Control Subsystem

In a Programmable Mix, the Slave Channel is the control channel that is partly or completely controlled by the Master Channel.

Software 12 Mix – Control Subsystem, Logic Subsystem

A version of the “12 mix” arcade style controls where the control mixing takes place in software on the Microcontroller, rather than in hardware on the Transmitter. The software implementation of the controls also performs a few of the calculation differently, resulting in a somewhat different feel for the driver. This mode is activated by placing a jumper clip on top of Analog/Digital Port 14 on the Microcontroller.

Spacer – Structure Subsystem, Motion Subsystem

There are several plastic spacers which are designed to slide onto square bar axles between other parts (or between parts and rails) to keep them from moving too close together. They can also be used like collars if enough spacers are added to keep the other parts from moving at all.

SPDT switch – Sensor Subsystem

Short for “Single Pole, Double Throw.” A switch that is activated by a single contact (single pole), but changes the state of two outputs at once (double throw). The Limit Switch Sensor is an SPDT switch, but one of the two outputs is hidden, making it function effectively as an SPST switch.

Speed – Motion Subsystem

Technically, speed is the magnitude of velocity (i.e. velocity, but without indicating direction). It is most commonly used to mean the rate of movement of a vehicle. By extension, it can also mean the rate of rotation of a gear or wheel. It is also sometimes used to refer to a vehicle’s potential maximum speed, as opposed to its acceleration capability.

SPST switch – Sensor Subsystem

Short for “Single Pole, Single Throw.” A switch that is activated by a single contact (single pole) and changes the state of a single output (single throw). The Bumper Switch Sensor is an SPST switch.

Stability – Structure Subsystem

The ability of a robot to remain upright and steady while moving over terrain and traversing obstacles.

Stall (Motor) – Motion Subsystem

A condition where a motor encounters so much resistance that it cannot turn. It is damaging for the motor to be in this condition. The motor can get hot and can stop functioning.

Stick Mode – Control Subsystem

An advanced feature of the Transmitter that allows control channels 2 and 3 to trade places on the joysticks (2 becomes the right stick’s vertical axis and 3 becomes the left stick’s vertical axis). The default mode is 2, and should not be changed under most circumstances.

Stress (Structural) – Structure Subsystem

Physical forces acting on an object constitute mechanical stress. Too much stress concentrated on a small area can cause parts to bend or break.

Glossary

Structure Subsystem

The subsystem responsible for holding the rest of the subsystems together and in place, and for protecting them from physical harm.

Subsystem

A subdivision of a system that helps to organize the system into convenient compartmentalized functions. The lines between subsystems are not always perfectly clear (for example, a wheel's axle is both a motion-transferring device and a physical support), but they work to give a general idea of purpose for the components in a system.

Support – Structure Subsystem

The degree of physical stability a piece has, owing to the strength of the foundation provided by the other pieces which are holding it in place. A piece which provides a physical brace or foundation for another piece is also called a support.

Support Polygon – Structure Subsystem

The imaginary polygon formed by connecting all the points at which the robot touches the ground. In cases where the arrangement of ground contact points is complex, the support polygon is the largest convex polygon that can be formed by those points. If the center of mass of the robot is not directly over the support polygon (i.e. projecting a line straight down from the center of gravity would not intersect the support polygon) at all times, the robot will fall over.

T

Tank-style Controls – Control Subsystem

A Transmitter driving mode in which the robot is controlled with only the vertical axes of the joysticks. Each joystick controls the motion of one side of the robot, like an old tank. Also called 23 mode because axes 2 and 3 are being used to drive the robot.

Tether – Control Subsystem

A cable used to connect the Transmitter directly to the Microcontroller. Using a tether allows you to control the robot by sending signals through the cable rather than through the air, eliminating the possibility of radio interference. You can use any telephone handset cable (the one that goes from the base to the handset of a corded phone) as a tether.

Threaded – Structure Subsystem

A threaded piece has threading on or in it, which allows a screw to be fastened into it. Threading is the tiny spiraling texture on the outside of a screw or the inside of a nut (for example) that allows a screw to be locked into place.

Torque – Motion Subsystem

Angular ("spinning") force. Torque can be converted into linear ("pushing") force where a wheel comes in contact with the ground.

Traction – Motion Subsystem

An overall measure of how well a tire is able to grip the ground. Many factors (texture, size, material, etc.) must be taken into account when evaluating a tire's traction on different surfaces.

Transmitter – Control Subsystem

The primary user interface device for the human operator of the robot. The Transmitter gathers input from its two joysticks and four buttons, and transmits them via FM radio wave to the RF Receiver mounted on the robot.

Transmitter Battery Holder – Power Subsystem

The battery container for the Transmitter. The battery holder contains the 8 NiCd AA batteries in place required to operate the Transmitter. If you wish to use the 9.6V battery pack, the Battery Box can be easily removed to make room for the 9.6V pack.

Glossary

Transmitter Frequency Crystal – Control Subsystem

The swappable module in back of the Transmitter that designates the radio frequency that the Transmitter will use to communicate with the RF Receiver Module. The frequency of the Transmitter Frequency Module must match the frequency of the crystal installed in the RF Receiver Module on the robot in order for them to communicate. All VEX Systems come with the same Transmitter Frequency Module. Additional modules on different frequencies are available for purchase separately.

Trickle Charge – Power Subsystem

A very low-power charge that is applied to full batteries in order to keep them full. A trickle charge counteracts a battery's natural loss of charge over time, so that the battery can be left on the charger, and still always maintain a full charge.

Trim – Control Subsystem

The calibration setting for the joysticks on the Transmitter. Also, the name of the menu on the Transmitter that allows for fine tuning of these settings.

V

Voltage (Battery) – Power Subsystem

The electrical voltage difference between the + and – terminals on a battery. Different batteries and battery packs have different starting voltages. Voltage falls (though not all the way to 0) as the battery's power is used up, and can be used as a rough indicator of the amount of capacity remaining on a battery.

Voltage (Electrical) – Power Subsystem

The difference in electrical potential between two points in a circuit or electrical field. An electron or other charged particle has more energy at one of the two points, and will tend to move toward the other point.

Voltage Drop – Power Subsystem

A phenomenon exhibited by rechargeable batteries where a battery that is frequently “shallow discharged” (discharged only a little between recharges) will begin to experience reduced performance. This can be reversed by discharging the batteries to a nearly-empty safe level (when the robot automatically turns off, for example—NOT by shorting them or draining them to 0V by external means!) and then recharging them to full capacity. Repeating the drain-charge cycle a few times should restore the batteries to full performance.

W

Washer – Structure Subsystem

A round metal or plastic disc placed between a screw head or nut and the surface to which it is mounted. The washer gives the screw a secure surface to brace against, and prevents the screw head from bending the metal surrounding the hole and popping all the way through. Steel washers should be used with screws that are not meant to move at all. Delrin (white plastic) washers should be used when the entire screw-nut assembly is meant to turn together (e.g. the screw at the pivot of a movable arm attachment).