

# Using Sensors with V5

- Sense, Plan, Act (SPA) summarizes the three critical capabilities that every robot must have in order to operate effectively:
  1. **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?
  2. **PLAN**: The robot needs to take the sensed data and respond appropriately to it, based on a pre-existing strategy.
    - Does your program determine the appropriate response, based on the sensed data?
  3. **ACT**: Finally, the robot must 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?

# Sensors

- Sensors are part of what makes a robot, a robot.
- Rather than just blindly running sequences of instructions, robots use sensors to gather information about the world around them, and respond based on programming.
- To take advantage of reacting to sensor readings while the program is running, we need to take a look at VEXcode Structures.



# Variables - Continued

# Programming Discussion – Data and Variables – Review

- Your robot is a type of computer.
- Computers are designed to store, process, and retrieve information, or data.
- Your robot is able to store, process, and retrieve data using **Variables**.
  - This is not the same as an Algebraic variable!
  - A Variable is space set aside in your robot's memory to store and retrieve data...
  - ..but we can still use them in calculations!



## Programming Discussion – Data and Variables

- In VEX C++, all variables must have a **type** and a **name**.
  - Variable **type** defines the type of data that the variable will hold.
  - Variable **name** defines how we can refer to the variable throughout the program.

## Programming Discussion – Variable Types

- There are multiple types of variables based on the type (and size) of data that you want to store, retrieve, and process in VEX C++.
- Some common types (there are more):

| Type      | Keyword             | Sample Data                              | Notes  |
|-----------|---------------------|--|--|
| Integer   | <code>int</code>    | <code>-337, 0, 488, 10283</code>         | - Whole numbers  |
| Float     | <code>float</code>  | <code>-22.7, 0.0, 0.432, 3.14</code>     | - Decimal numbers  |
| Bool      | <code>bool</code>   | <code>true, false, 1, 0</code>           | - True/False values  |
| Character | <code>char</code>   | <code>'a', 'x', '7', '&amp;'</code>      | - Single Characters using single quotes  |
| String    | <code>string</code> | <code>"Hello World!", "asdf_1234"</code> | - Multiple Characters using double quotes<br>- Must include <code>using namespace std;</code><br>or <code>prepend std::</code> |

## Programming Discussion – Variable Names

- Variable names follow the same rules as Motor and Sensor names.
- Variable names must be:
  - One word
  - Made up of letters, numbers, and underscores
  - Not already a VEX C++ reserved word
- Variable names should be:
  - Descriptive of the data that is being stored



- To create a variable, first specify its **type**, then provide a **name**.
- To store a value in a variable, follow the format:  
“**variable name = value;**”

```
4  int main() {  
5      int targetEncoderCount;  
6      targetEncoderCount = 720;  
7  
8  }
```

- To **initialize** a variable to a value, you can also use the following shortcut:

```
4  int main() {  
5      int targetEncoderCount = 720;  
6  
7  }
```

## Programming – Calculations

```
4  int main() {
5      //Create Variables to store values
6      int targetEncoderCount = 0;
7      float wheelCircumferenceCM = 31.9;
8      float targetDistanceCM = 0;
9      //Specify the Target Distance
10     targetDistanceCM = 120;
11     //Calculate the necessary number of Encoder Degrees to traverse the Target Distance
12     targetEncoderCount = (targetDistanceCM / wheelCircumferenceCM)*360;
13
14 }
```

- Variables allow you to store and retrieve data to perform calculations!
- Notice that the result of our calculation is stored in a variable – the equal sign on line 12 would be impossible as an Algebraic equation!

# Loops

- By the end of this section you will be able to use a while loop to repeat code.



Repeating a section of code while a condition is true.

```
while (I have not reached my desired destination)
{
    Take another step in the right direction
}
```

When to use it:

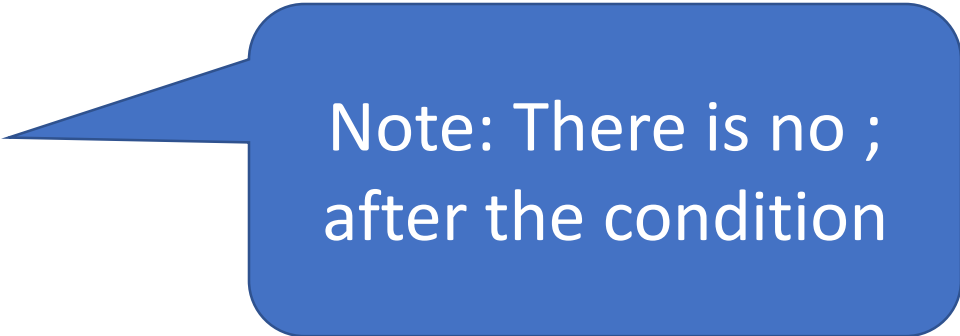
When you want to repeat something an unknown number of times (take another step) AND may never repeat (You were already at your desired location. No need to move)

# while loop

- How it works
  - A while(){} loop repeats any code between its curly braces while the condition between its parenthesis is true

- Syntax

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



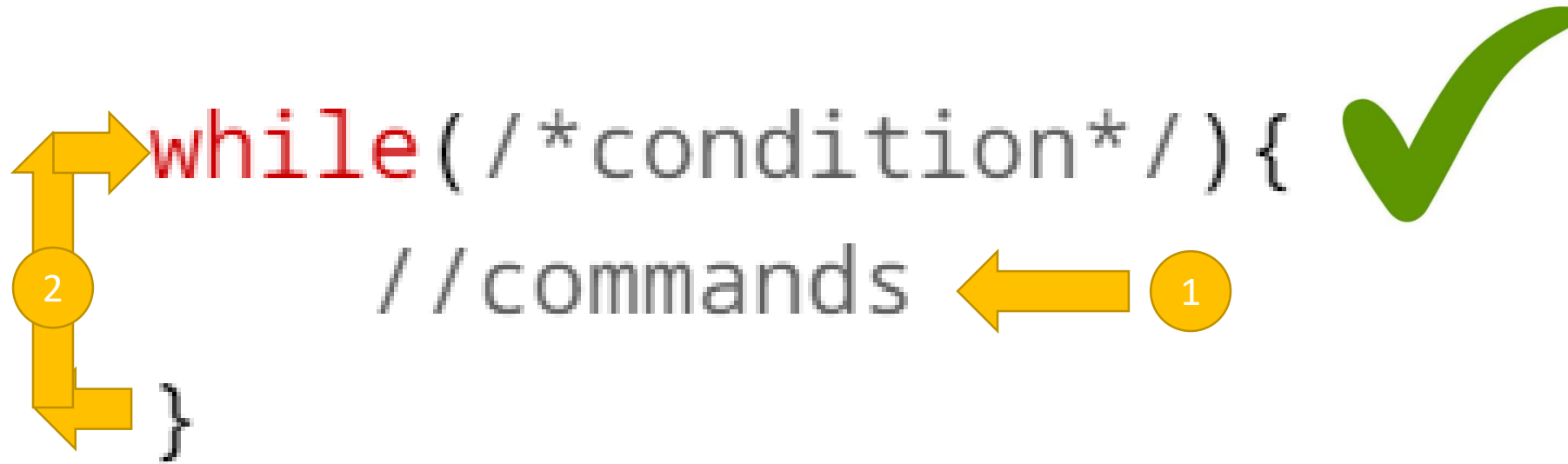
Note: There is no ;  
after the condition

## Programming – `while () {}` loops – Program Flow

```
while ( /*condition*/  ) { ←  
    // commands  
}
```

- When the robot reaches a `while () {}` loop the first thing it does is check the **condition** between the parenthesis.
- A **condition** is a special type of question the robot can answer based on **Boolean Logic**.
  - Has to be answerable with a “Yes/No” or “True/False”

## Programming – `while () {}` loops – Program Flow



- If the condition was **true** then the robot runs all of the code between the curly braces, **once**.
- After running all code between the curly braces, Program Flow returns to the beginning of the `while () {}` loop and the condition is checked again.
- Wash, rinse, repeat!

## Programming – `while () {}` loops – Program Flow

```
while ( /*condition*/ ) {  
    // commands  
}
```



- If the condition is ever not **true**, or **false**, when it is being checked then the robot **skips** all of the code between the curly braces and moves on in the program.
- **Note:** The Program Flow is only ever at one “spot” at a time!
  - The robot is **not** simultaneously checking the condition and running the code!



# But what if you want to loop forever?

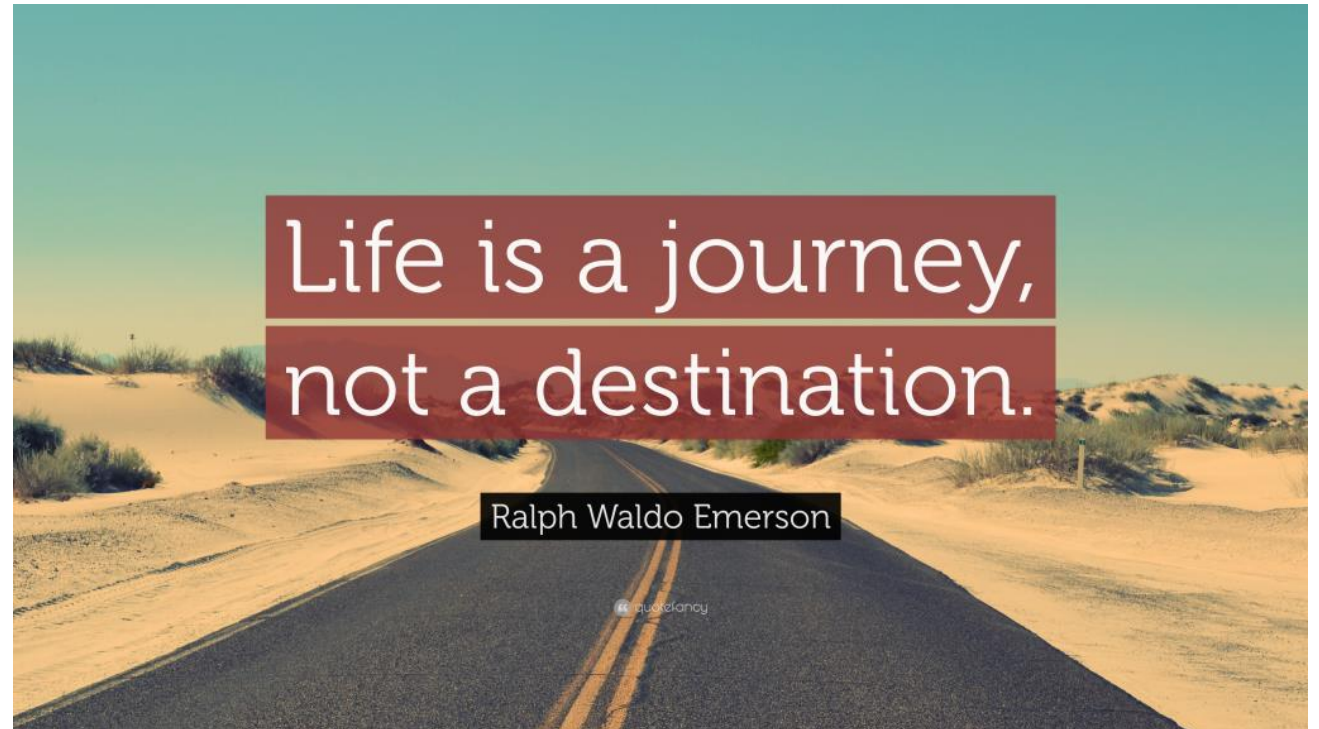
- Infinite loop. When the condition is ALWAYS true.

Examples:

```
while (true)  
{
```

```
while (1==1)  
{
```

```
while (1)  
{
```



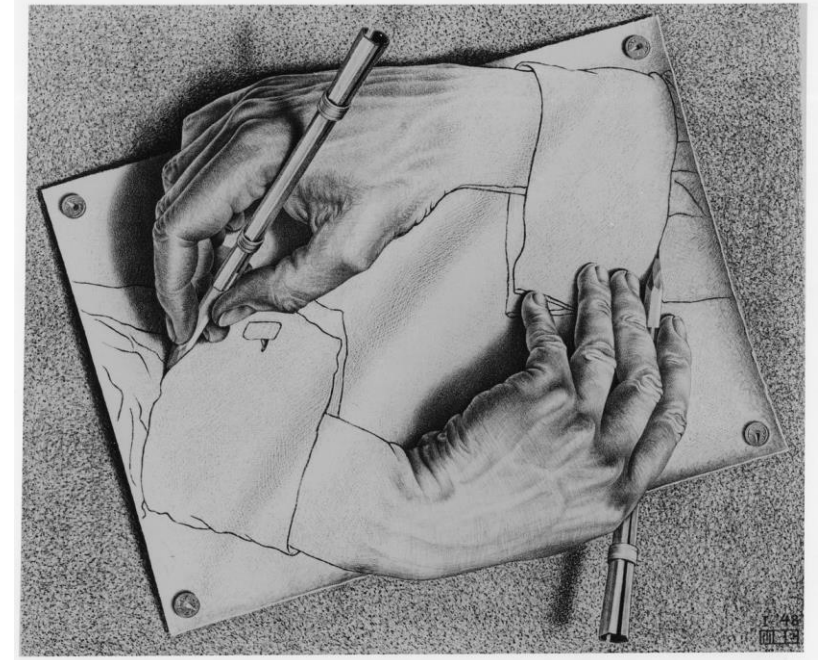
## Conditions: Programming Discussion – Relational Operators

- Relational Operators (sometimes called Inequality or Comparison Operators) allow you to form a **Boolean Condition**.
  - These operators compare the value on the left with the value on the right, and the result is either **true** or **false**.
- Note that = and == are not the same thing in programming!
  - = **sets** the value on the left equal to the value on the right
  - == **checks** if the value on the left is equal to the value on the right.

| Operator | Meaning                                       |
|----------|---|
| ==       | Is Equal To                                   |
| !=       | Is Not Equal To                               |
| >        | Is Greater Than                               |
| <        | Is Less Than                                  |
| >=       | Is Greater Than or Equal To                   |
| <=       | Is Less Than or Equal To                      |
| !        | Not. Turns a true to false or a false to true |

# while loop example

```
int main()
{
    int count = 0;
    while (count < 10)
    {
        Brain.Screen.print("Count = %d", count);
        Brain.Screen.newLine();
        task::sleep(500);
        count = count + 1;
    }
}
```



```
int main()
{
    int x = 10;
    int y = 120;
    int radius = 10;
    Brain.Screen.setPenColor(black);
    while (x<480)
    {
        Brain.Screen.setFill-color(green);
        Brain.Screen.drawCircle(x, y, radius);
        task::sleep(50);
        Brain.Screen.setFill-color(black);
        Brain.Screen.drawCircle(x, y, radius);
        x = x + 2;
    }
    Brain.Screen.setFill-color(red);
    radius = 1;
    while (radius < 500)
    {
        Brain.Screen.drawCircle(x, y, red);
        Brain.Screen.setFill-color(radius);
        task::sleep(10);
        radius = radius + 1;
    }
}
```

# while loop example program

# There are three programming Structures

- Linear (Block)
  - Brain...
  - $x=x+1$ ;
- Looping (Repetition)
  - while ()
- Decision (Selection)
  - None yet,
  - Time for if and else

# if and else

- When you want to respond to something one time (not looped)
- If the sensor is pressed then stop lifting the arm
- If something is too close back up
- If I cross a white line then turn,

# How it works.

- If the condition is true then run the commands that are in the next set of {}.  
If the condition is false then skip to the code after the {}.
- Syntax

```
if (condition)
{

}
```

# if Example

```
int main() {  
    int count = 0;  
    while (count < 10)  
    {  
        if (count < 4)  
        {  
            Brain.Screen.print("Less than 4!@!@#@!#@@!#");  
        }  
        Brain.Screen.print("Count = %d", count);  
        Brain.Screen.newLine();  
        task::sleep(500);  
        count = count + 1;  
    }  
}
```

Programming Style Note:  
Indent the code that is being  
controlled by a structure.  
i.e. The code between the {}.



# if example with AND &&

```
int main() {  
    int count = 0;  
    while (count < 10)  
    {  
        if ((count > 3) && (count < 6))  
        {  
            Brain.Screen.print("In the middle");  
        }  
        Brain.Screen.print("Count = %d", count);  
        Brain.Screen.newLine();  
        task::sleep(500);  
        count = count + 1;  
    }  
}
```

The condition is true ONLY if  
count > 3 AND count < 6.

# if example with OR ||

```
int main() {  
    int count = 0;  
    while (count < 10)  
    {  
        if ((count < 2) || (count > 6))  
        {  
            Brain.Screen.print("On the outside");  
        }  
        Brain.Screen.print("Count = %d", count);  
        Brain.Screen.newLine();  
        task::sleep(500);  
        count = count + 1;  
    }  
}
```

The condition is true if count < 3  
OR count > 6.

# If else :How it works

- If the condition is true then run the commands that are in the next set of {}.
- If the condition is false then run the commands in the {} after the else
- Syntax

```
if (condition)
{
    //Commands done when the condition is true
} else
{
    //Commands done when the condition is false
}
```

```
int main()
```

```
{
```

```
    int count = 0;
```

```
    while (count < 10)
```

```
    {
```

```
        if (count < 5)
```

```
        {
```

```
            Brain.Screen.print("Low ");
```

```
        }
```

```
        else
```

```
        {
```

```
            Brain.Screen.print("High ");
```

```
        }
```

```
        Brain.Screen.print("Count = %d", count);
```

```
        Brain.Screen.newLine();
```

```
        task::sleep(500);
```

```
        count = count + 1;
```

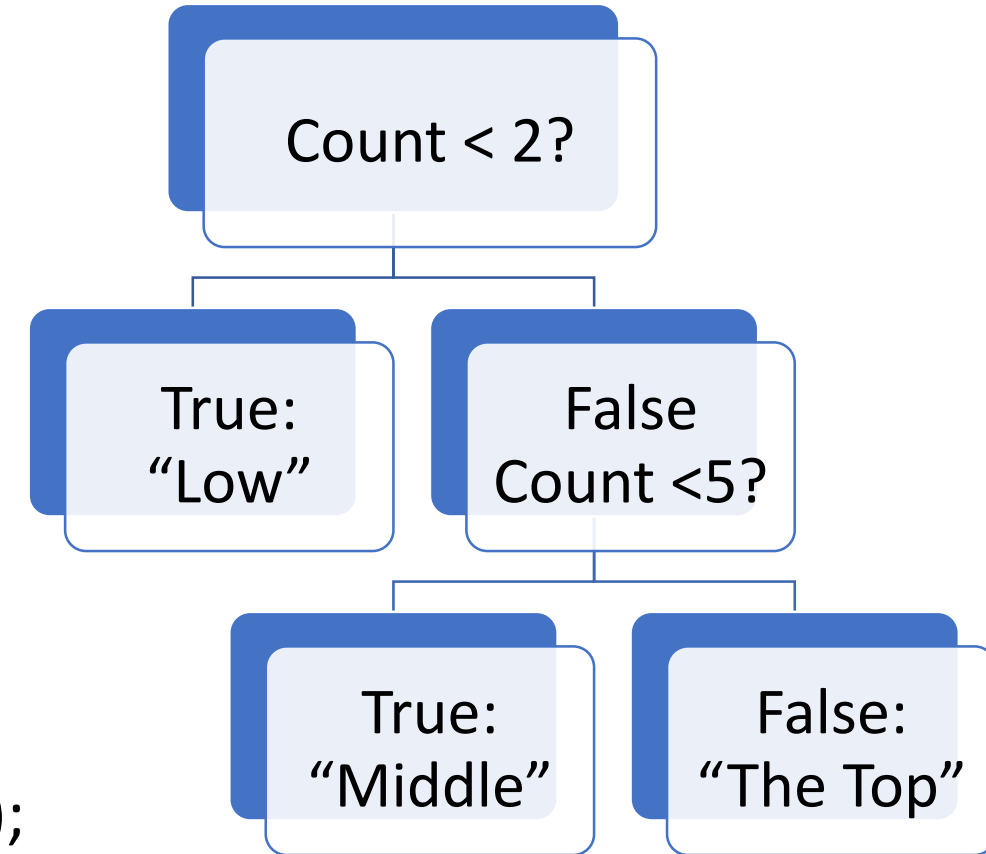
```
    }
```

```
}
```

# If else example

# Chaining if..else

```
int main() {
    int count = 0;
    while (count < 10)
    {
        if (count < 2)
        {
            Brain.Screen.print("Low ");
        } else if (count < 5)
        {
            Brain.Screen.print("Middle ");
        } else
        {
            Brain.Screen.print("The top ");
        }
        Brain.Screen.print("Count = %d", count);
        Brain.Screen.newLine();
        task::sleep(500);
        count = count + 1;
    }
}
```



Code Break: Enter and test this program.



# Back to Sensors

- Sensors are part of what makes a robot, a robot.
- Rather than just blindly running sequences of instructions, robots use sensors to gather information about the world around them, and respond based on programming.



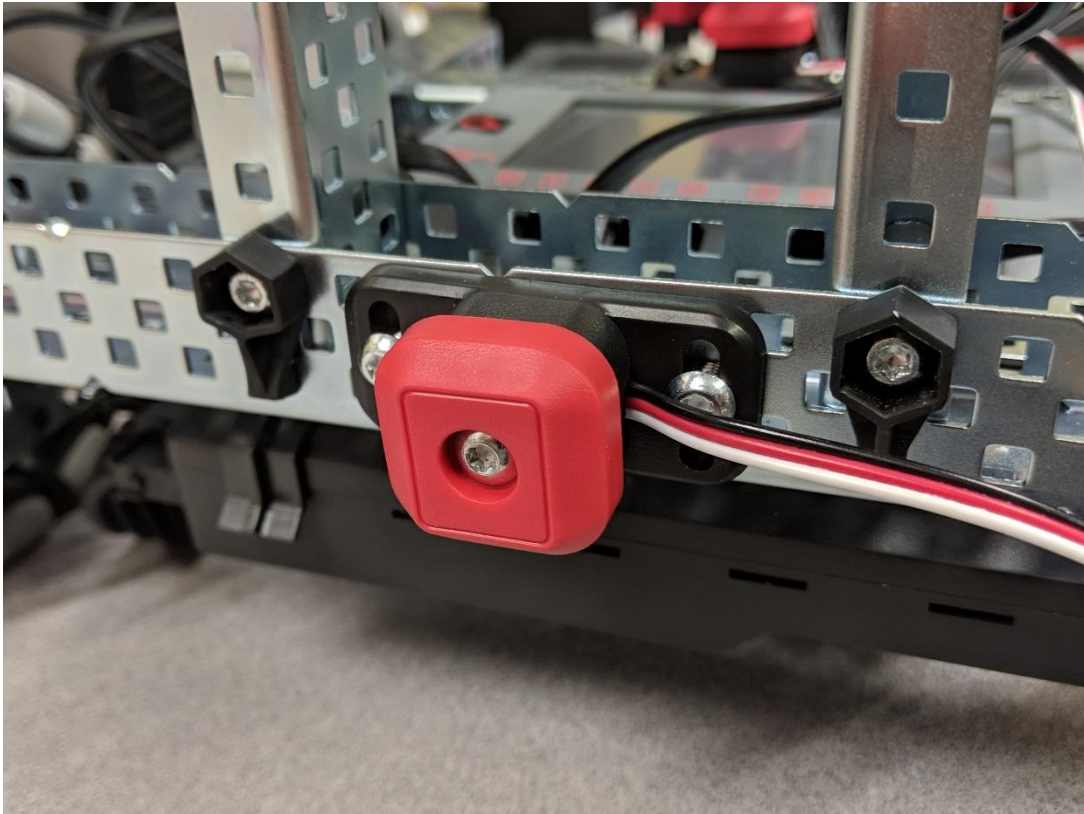
## Sensor Overview – Bumper Switch v2

- The Bumper Switch v2 reports to the Robot Brain is whether it is pressed in, versus released.
- The Bumper Switch v2 allows the robot to sense physical collisions:
  - Robot body bumps into an obstacle
  - Robot arm reaches a mechanical limit
- This sensor connects to 1 of the 8 Tri-Ports (A – H) on the V5 Robot Brain.
- The sensor “cap” may be disconnected and other components may be used to create a more ideal “hit area”.



## Connecting the Bumper Switches

- Uses for the Bumper Switch
  1. 1 Bumper Switch on the rear chassis of the V5 Clawbot
  2. 1 Bumper Switch on the top-front of the chassis, so that the arm makes contact





Connect Bumper Switch and check how the values change only using the V5 brain. My examples will be based on the sensor plugged into port A.

## Checking Sensor Values from the V5 Brain

- The **V5 Robot Brain** allows you to check the values of the Bumper Switches under the **Devices** menu.
  - Choose Devices, then select the triangular icon
- The Devices menu does not automatically “know” what type of sensor is connected.
  - Tap the box where the Bumper Switch is connected until it displays **Digital Input**.
  - The Bumper Switch is a type of touch sensor, which is a **Digital Sensor**.
- When the Bumper Switch is pressed, it should have a value of **High (1)**, and when released a value of **Low (0)**.



## General Sensor Configuration in VEXcode: Before task main()

```
// define your global instances of motors and other devices here  
limit Limit1 = limit( Brain.ThreeWirePort.A );
```

Device Type

Device Name

Device Port

### •Formatting Notes

- The **device type**. Defines the type of sensor. More options on the next slide.
- The **device name** can be any name that isn't already used by the programming language.
  - Start with a letter
  - No spaces/punctuation
  - Not a reserved word or a name that is previously used.
  - Describes the sensor**
- The **device port** needs to be in parenthesis right after the sensor's name. Assignment for three wire port devices will always start with **Brain.ThreeWirePort** and can be assigned any letter A-H matching where the sensor is plugged.

```
// define your global instances of motors and other devices here  
limit Limit1 = limit( Brain.ThreeWirePort.A );
```

Device Type

Device Name

Device Port

## • Some Device Types Supported

- bumper (Works just like limit)
- limit (Works just like bumper)
- sonar
- pot
- light
- line
- vision



So many ways to do the same thing...

Defining a limit switch/touch sensor/bumper switch

```
limit Limit1 ( Brain.ThreeWirePort.A );
```

```
vex::limit Limit1 ( Brain.ThreeWirePort.A );
```

```
vex::limit Limit1 = limit( Brain.ThreeWirePort.A );
```

```
vex::limit Limit1 = vex::limit( Brain.ThreeWirePort.A );
```

```
bumper Limit1 ( Brain.ThreeWirePort.A );
```

```
vex::bumper Limit1 ( Brain.ThreeWirePort.A );
```

```
vex::bumper Limit1=limit( Brain.ThreeWirePort.A );
```

```
vex::bumper Limit1 = vex::bumper( Brain.ThreeWirePort.A );
```

For this workshop I'll focus on the shorter to code options.

# Declaring and using the bumper/limit switch/touch sensor

- **Declaring the limit switch. Before task main()**
  - `limit Limit1 ( Brain.ThreeWirePort.A );`
  - **Limit1** is the what the sensor is named in this example. The programmer can pick a different name. It must start with a letter, no spaces, no punctuation, not a reserved word, not used somewhere else and is descriptive.
- **Getting the state: Inside task main()**
  - `Limit1.pressing()`
  - With return true if it is being pressed
  - Will return false if it is not being pressed
- **Example on next slide**



# Example Limit/Bumper Switch Program

Code Break: Enter  
and test.

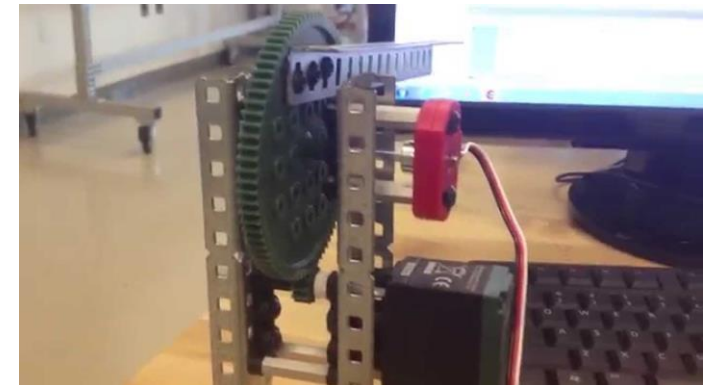
```
// define your global instances of motors and other devices here
limit Limit1 = limit( Brain.ThreeWirePort.A ); // Plug limit switch or bumper into 3-wire port 'A'

int main()
{
    Brain.Screen.print("User Program has Started.");
    int pressedCount = 0;
    Brain.Screen.printAt(1, 40, "The pressed count is %d", pressedCount);
    while(true) //Will loop everything inside the next {} as long as true is true
    {
        if( Limit1.pressing() ) //If the limit switch is pressed do the commands in the next {},
        //or else skip the commands in the next {}
        {
            while( Limit1.pressing() ) //While the switch is still pressed do everything inside the next {}
            {
                task::sleep(20); //Wait for 20 milliseconds
            }
            pressedCount = pressedCount + 1; //Add 1 to the variable pressedCount
            Brain.Screen.printAt(1, 40, "The pressed count is %d", pressedCount); //Shows the count
        }
        task::sleep(20);
    }
}
```

High Flyers. Use 4 touch sensors to make your Brain into an 'Etch-A-Sketch'. Can even add another touch sensor to 'Clear the Screen'.

# Potentiometer

- Measure the angle position
- Measures 0 to 250 +/- 20 degrees
- Returns values in degrees, percentage or millivolts
- Uses
  - Knowing the angle of an arm
  - Selecting Autonomous code
  - ...



# Potentiometer Declaration and Use



- Declaration: Before *task main()*

**pot Potentiometer1 ( Brain.ThreeWirePort.E );**

- **Potentiometer1** is the what the sensor is named in this example. The programmer can pick a different name. It must start with a letter, no spaces, no punctuation, not a reserved word, not used somewhere else and is descriptive.
- **E** is the port where the Potentiometer will be attached.
- Reading the values of the potentiometer. Inside *task main()*. Returns an integer value representing the angle.
  - **Potentiometer1.value(rotationUnits::deg)**
  - **Potentiometer1.value(rotationUnits::pct)**
  - **Potentiometer1.value(rotationUnits::mV)**



# Potentiometer Sample Program

```
// define your global instances of motors and other devices here
```

```
pot Potentiometer1 ( Brain.ThreeWirePort.E );  
controller Controller1 = vex::controller();
```

```
int main()  
{
```

```
    // Clear the controller's screen.  
    Controller1.Screen.clearScreen();
```

```
    while(true)  
    {
```

```
        // Manually turn the shaft connected to the potentiometer. This will cause the values to change.  
        // Print the value of the potentiometer in degrees to the Screen.
```

```
        Brain.Screen.printAt(1, 40, "Potentiometer = %5f", Potentiometer1.value(rotationUnits::deg));
```

```
        // use line 3, the bottom line, on the controller
```

```
        Controller1.Screen.setCursor(3,1);
```

```
        // Print the value of the Potentiometer sensor on the controller's screen.
```

```
        Controller1.Screen.print("Pot1 = %5f", Potentiometer1.value(rotationUnits::deg));
```

```
        task::sleep(100);
```

```
    }
```

```
}
```

Code Break:

Enter code, test and experiment.

Be sure to have a Controller connected to the Brain. Include comments as needed for your reference.

More on using the Controller's display on the next slide.

# Some Controller Communication Features

- Initializing the Controller. Before *task main()*
  - **controller Controller1 = vex::controller();**
  - **Controller1** is the what the sensor is named in this example. The programmer can pick a different name. It must start with a letter, no spaces, no punctuation, not a reserved word, not used somewhere else and **is descriptive**.
- Clear the screen: Inside *task main()*
  - **Controller1.Screen.clearScreen();**
- Set the Cursor position. (1,1) is row 1 column 1, the top left of the screen. You can use up to three rows.
  - **Controller1.Screen.setCursor(1,1);**
- Printing on the Screen at the current cursor location
  - **Controller1.Screen.print("Hello World");**
  - **Controller1.Screen.print("Pot1 = %.5f", Potentiometer1.value(rotationUnits::deg));**
    - **"%.5 f"** This is setting the position for a float (real) value showing 5 values to the right of the decimal. The value returned from the Potentiometer1.value() function will go into this position.
- Haptic (Force) Feedback!!!
  - **Controller1.rumble("--- ...");**
    - Dash is long
    - Dot is short
    - Space is pause
    - Up to 8 characters



# Potentiometer Activities: Complete one of the following.

- Shake it Up.
  - 0 to 90 degrees. No shake
  - 91 to 180 degrees. Shake with short shakes
  - 181 to 270 degrees. Shake with long shakes
- Draw a circle on the brain where the size changes based on the Potentiometer reading.
  - The lower the reading, the smaller the circle.

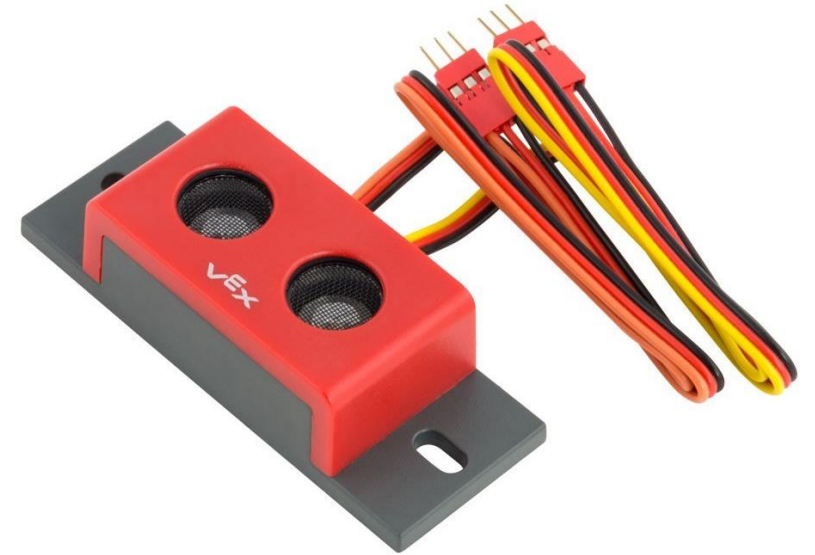


# Ultrasonic Rangefinder



# Ultrasonic Rangefinder

- The Ultrasonic Rangefinder uses sound waves to detect how far away it is from the nearest object.
- One side emits a sound wave - the other side waits for the echo to bounce off of an object and return to the sensor.
  - Distance is calculated based on elapsed time!
- Features:
  - Can provide **distance** values in millimeters, centimeters, or inches
  - Connects to the V5 Robot Brain through two Tri-Ports
  - Has two sets of wires, labeled INPUT and OUTPUT



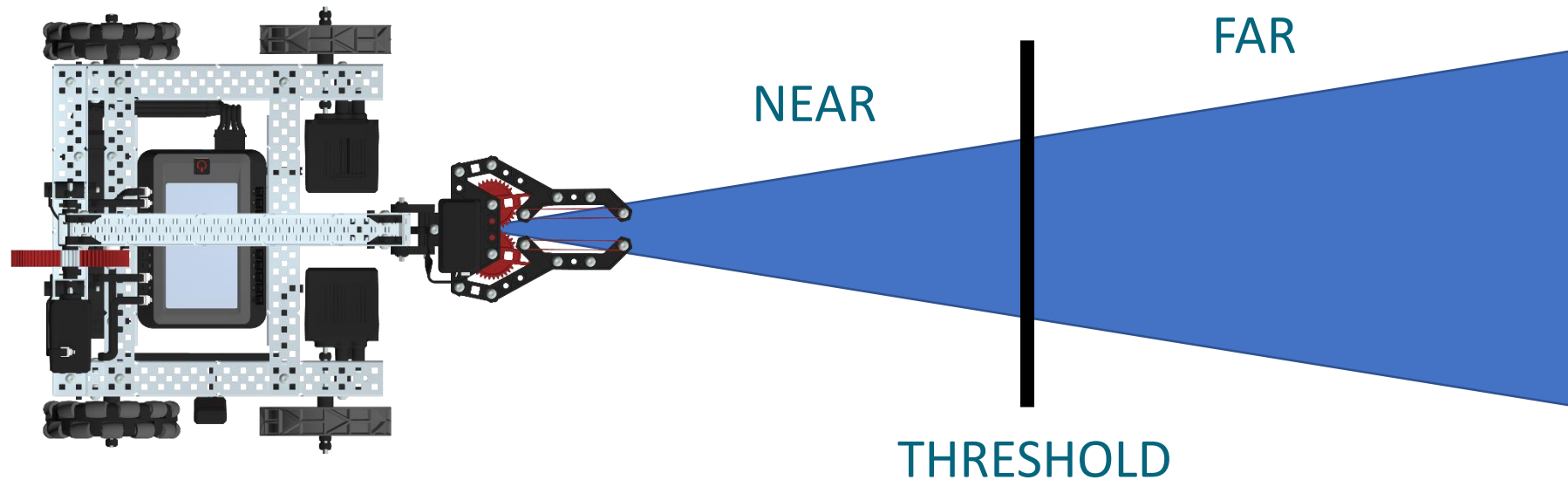
## Ultrasonic Range Finder: Connecting to Brain

- It requires 2 Tri-Ports!
  - The ports must be one of the ordered pairs (A-B, **C-D**, E-F, or G-H)
  - OUTPUT wire must be plugged in on the earlier port (first alphabetically)
  - INPUT wire must be plugged in on the later port (second alphabetically)
- It is good practice to for custom sensor names to reflect **where** and **what** they are!



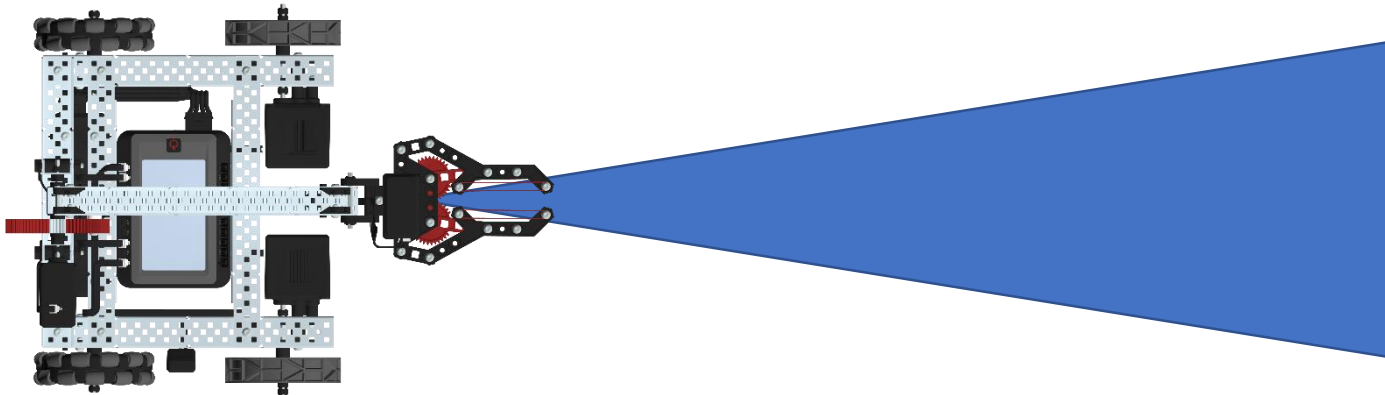
# Thresholds

- The Ultrasonic Rangefinder provides a **range** of values, but the robot can only make Yes-No, True-False Boolean Comparisons.
- A **Threshold** value is chosen to separate the range of many values into two categories: **Near** or **Far**



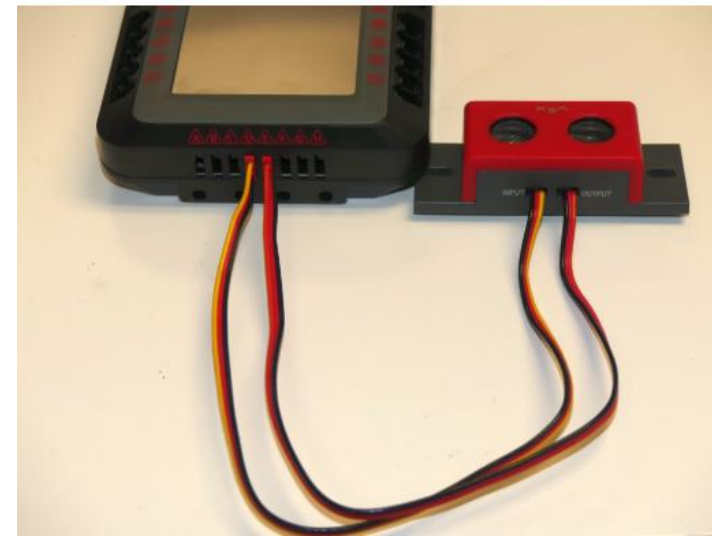
# Additional Considerations

- The Ultrasonic Rangefinder provides distance values-based sound waves – things that affect sound waves will affect this sensor!
  - Sound-absorbent or reflective materials = ☹️
- When it fails to receive an echo, it has a negative distance value.
  - This includes the beginning of your program before the first echo is received.
- Sound waves expand as they travel, so Ultrasonic Rangefinder may be able to detect objects in a cone-shaped Field of View.





# Declaring and using Sonar Sensor



Declaring the Sonar Sensor: Before *task main()*

```
sonar Sonar1( Brain.ThreeWirePort.C );
```

***Sonar1*** is the what the sensor is named in this example. The programmer can pick a different name. It must start with a letter, no spaces, no punctuation, not a reserved word, not used somewhere else and is descriptive.

//Output wire is plugged into Port C in this example.

// Input wire must be plugged into the next port, Port D in this example.

Reading Values from the Sensor, inside *task main()*

- **Sonar1.distance(distanceUnits::in)**
  - Returns the number of inches as a real value (float, double)
- **Sonar1.distance(distanceUnits::mm)**
  - Returns the number of millimeters as a real value (float, double)
- **Sonar1.distance(distanceUnits::cm)**
  - Returns the number of centimeters as a real value (float, double)

# Sample UltraSONIC Range Finder

```
sonar Sonar1 ( Brain.ThreeWirePort.C ); //Output is plugged into Port C
```

```
// Input must be plugged into the next port, Port D in this case
```

```
int main()
{
    // Wait 2 seconds or 2000 milliseconds before starting the program.
    task::sleep( 2000 );
    // Print to the screen that the program has started.
    Brain.Screen.print("User Program has Started.");
    while( true )
    {
        //...Spin the motors forward.
        Brain.Screen.printAt(10,40,"Sonar Reading %f inches ",Sonar1.distance(vex::distanceUnits::in) );
        Brain.Screen.printAt(10,60,"Sonar Reading %f mm ",Sonar1.distance(vex::distanceUnits::mm) );
        Brain.Screen.printAt(10,80,"Sonar Reading %f cm ",Sonar1.distance(vex::distanceUnits::cm) );
        // Sleep the task for a short amount of time to prevent wasted resources.
        task::sleep(20);
    }
}
```

Your turn. Code and test the sensor readings. *What value is returned when it does not see anything?*

```
sonar Sonar1( Brain.ThreeWirePort.C );
```

```
int main()
```

```
{  
  // Print to the screen that the program has started.
```

```
  Brain.Screen.print("User Program has Started.");
```

```
  while( true )
```

```
  {
```

```
    if ((Sonar1.distance(distanceUnits::in) >20) || (Sonar1.distance(distanceUnits::in)<0))
```

```
    {
```

```
      Brain.Screen.printAt(10,40,"Cold! Sonar Reading %.2f inches ",Sonar1.distance(distanceUnits::in) );
```

```
    }
```

```
    else if ((Sonar1.distance(distanceUnits::in)>10))
```

```
    {
```

```
      Brain.Screen.printAt(10,40,"Warmer!! Sonar Reading %.2f inches ",Sonar1.distance(distanceUnits::in) );
```

```
    }
```

```
    else
```

```
    {
```

```
      Brain.Screen.printAt(10,40,"Hot! Sonar Reading %.2f inches ",Sonar1.distance(distanceUnits::in) );
```

```
    }
```

```
    // Sleep the task for a short amount of time to prevent wasted resources.
```

```
    task::sleep(20);
```

```
  }
```

```
}
```

# What does the following code do?

Will display only two digits of a float (real number) to the right of the decimal point

# Optional: UltraSONIC Range Finder Mapping Activity

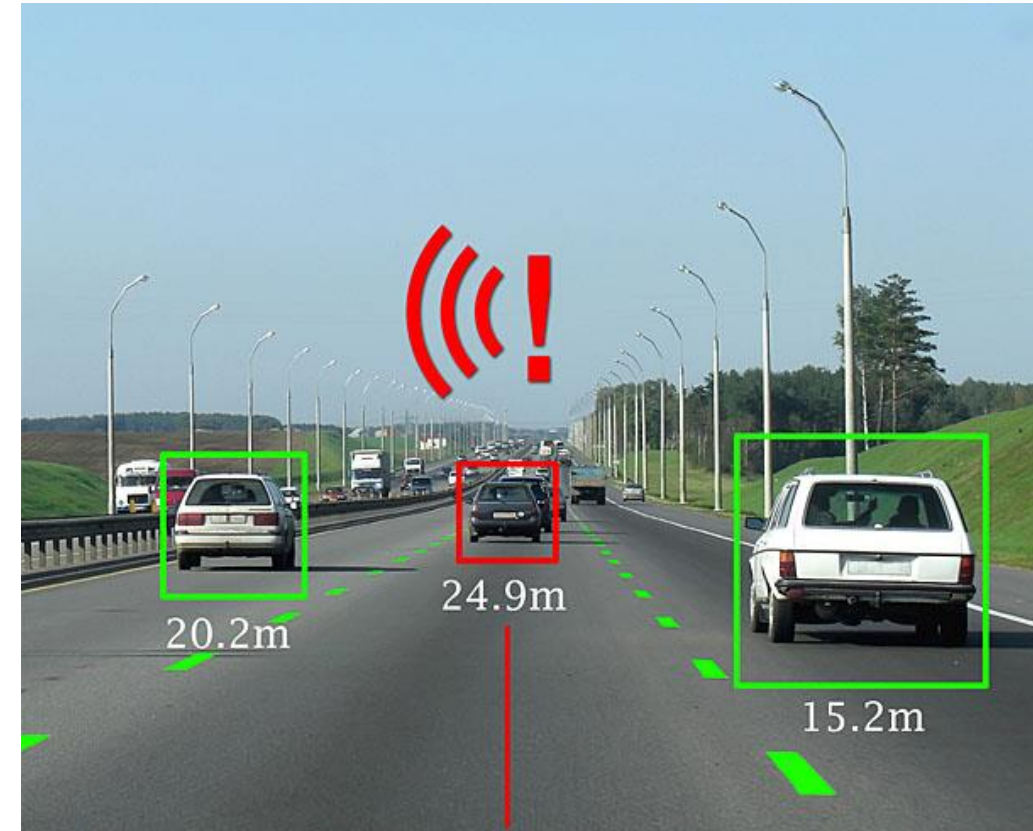
- Using a ruler map the reading to the actual distance from the sensor

| Distance (Inches) | Reading | Error |
|-------------------|---------|-------|
| 0                 |         |       |
| 5                 |         |       |
| 10                |         |       |
| 15                |         |       |
| 20                |         |       |
| 25                |         |       |
| 30                |         |       |
| 35                |         |       |
| 40                |         |       |
| 45                |         |       |
| 50                |         |       |

Graph and develop an equation for Actual Distance and Sonar Sensor distance?

# Ultrasonic Activity: Complete any one of the following

- Drivers assist
  - When an object is far away – No force feedback
  - When it gets close – Short force feedback
  - When it is very close – Long force feedback
  - Extension: Combine with the Potentiometer where the potentiometer is used to alter the values for close and very close.
- Draw a circle on the brain where the size changes based on how close it is to an object.
  - The closer the object, then larger the circle.

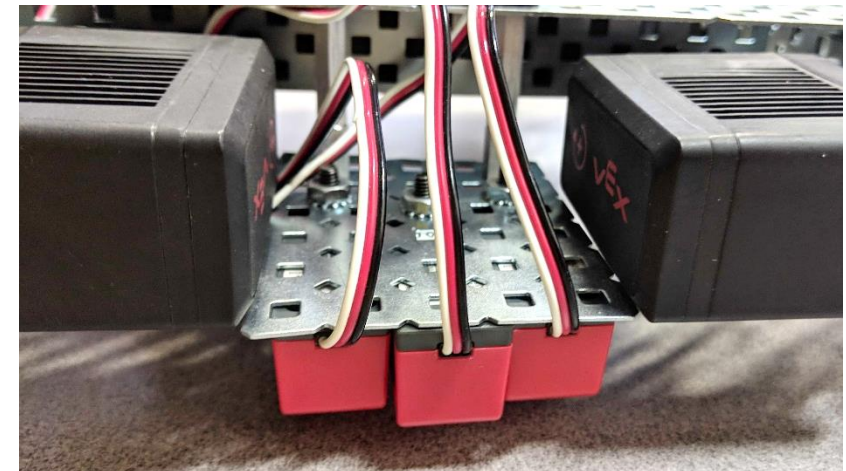


# Line Tracking Sensors



# Line Tracking Sensors

- The Line Tracking Sensor is an **active** light sensor that emits an **Infrared beam** and measures how much is reflected back.
  - The amount of reflected Infrared light allows the robot to know if it is above a dark or light surface.
- The VEX Line Tracking Kit comes with a set of 3 Line Tracking Sensors.
- Features:
  - Can provide **analog values** at different levels of precision (percent, 8bit, 10bit, 12bit, millivolts)
  - Connects to the V5 Robot Brain through Tri-Ports



## Sensor Configuration

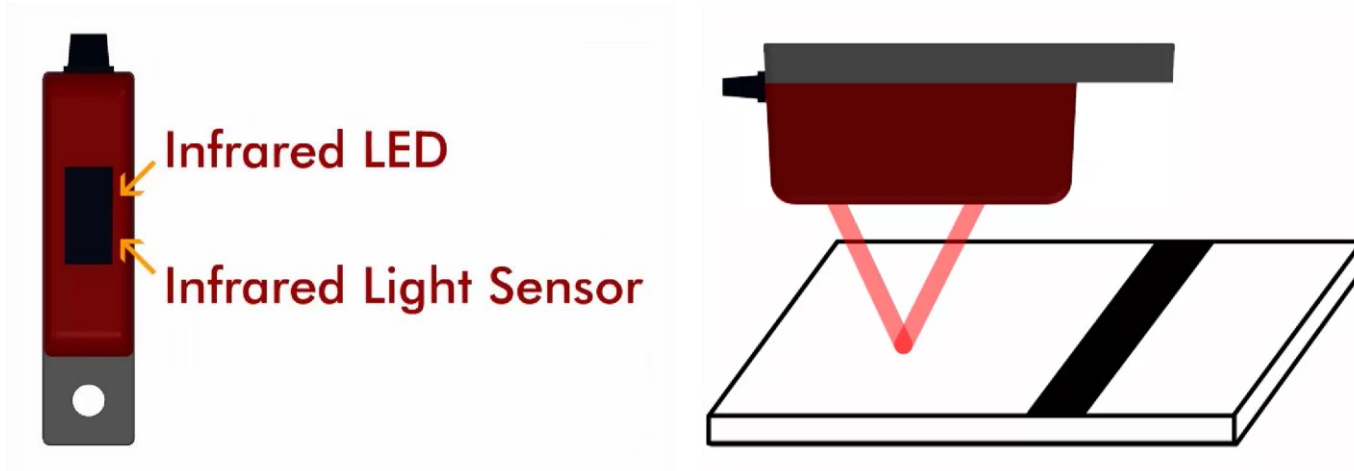
- The Line Tracking Sensors must currently be configured in **code**.
  - **light** defines the type of sensor
  - You must also choose a custom name and specify which Tri-Port it is connected to
- A full kit requires 3 Tri-Ports!
  - Any available ports may be used
- It is good practice to for custom sensor names to reflect **where** and **what** they are!





# Additional Considerations

- Each Line Tracking Sensor includes an Infrared LED and Light Sensor.



- All 3 sensors should be mounted the same distance from the measured surface, and that distance should be between 0.125 and 0.25 inches.
- Ambient light and changes in the measured surface **do** affect sensor readings.
- Line Tracking works better on narrow robots. 😊

```
RightLineTracker.value(analogUnits: |
```

pct  
range8bit  
range10bit  
range12bit  
mV

pct

*An analog unit that is measured in percentage.*

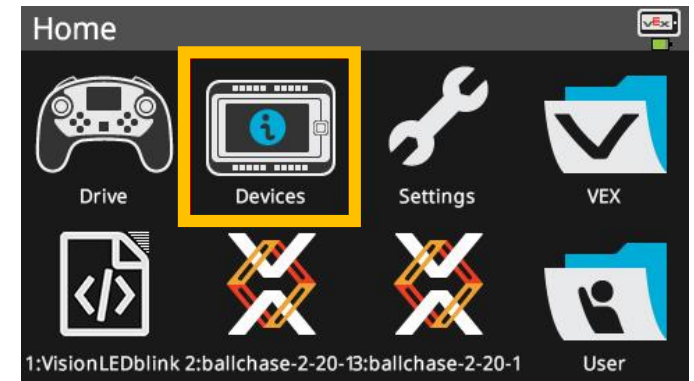
- The **.value ()** command allows you to specify whether the values provided by the Line Tracking Sensor are a percentage 8bit number, 10bit number, 12bit number or millivolt reading.
  - Percentage: **0 – 100**
  - 8bit: **0 - 255**
  - 10bit: **0 - 1023**
  - 12bit: **0 - 4095**

# Measured Thresholds

- The Line Tracking Sensor provides a **range** of values, but the robot can only make Yes-No, True-False Boolean Comparisons.
- A **Threshold** value is chosen to separate the range of many values into two categories: **Light** or **Dark**
  - Unlike the Ultrasonic Rangefinder, the values do not correspond to an intuitive set of units!
  - Instead, we must measure what the Line Tracking Sensors “see” when detecting a Light Surface versus a Dark surface to develop a Threshold.
- All three sensors may not provide identical readings, but the difference between Light values and Dark values should be large enough for one Threshold to work for each.

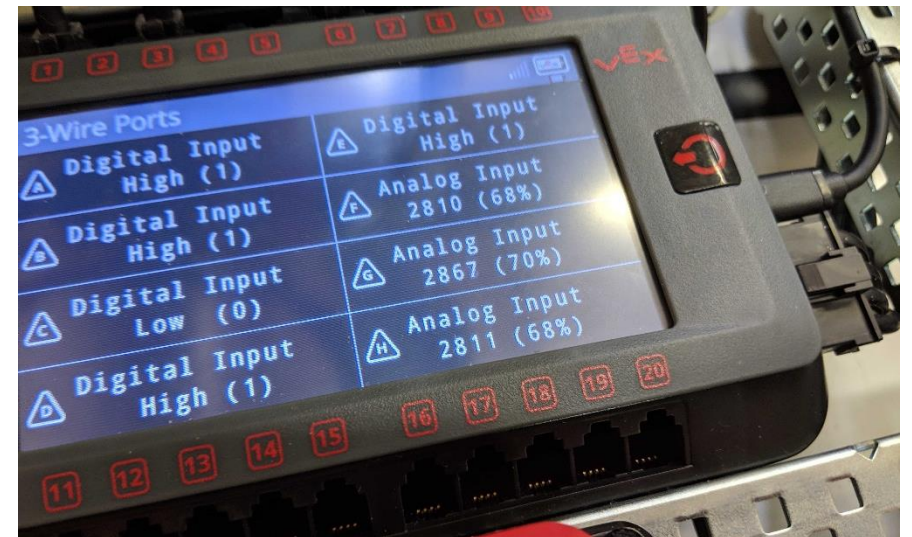
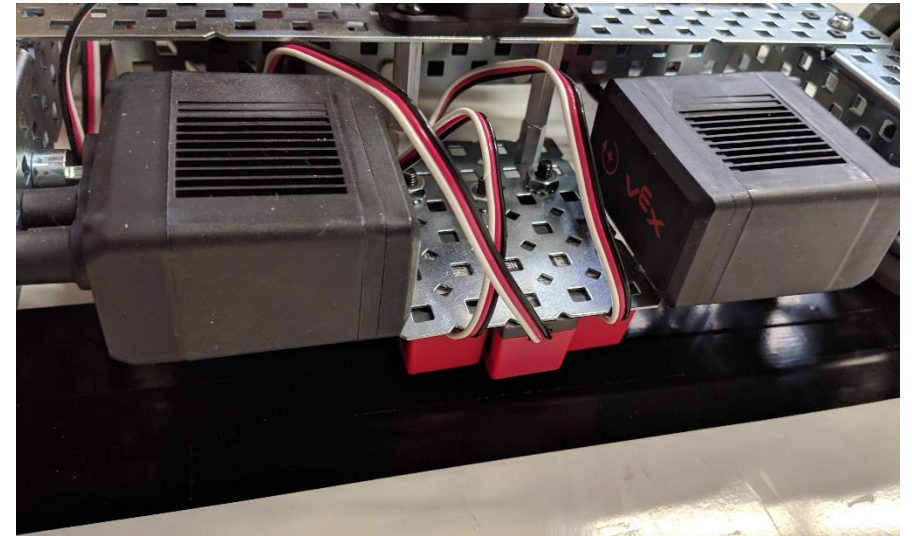
# Measuring Thresholds – Part 1

- The V5 Robot Brain allows you to check the values of the Line Tracking Sensors under the Devices menu.
1. Choose Devices, then tap the triangular icon.
  2. Tap the boxes where the Line Tracking Sensors are connected until they all display **Analog Input**.
  3. The value of each Line Tracking Sensor is displayed as a 12bit number and percentage.



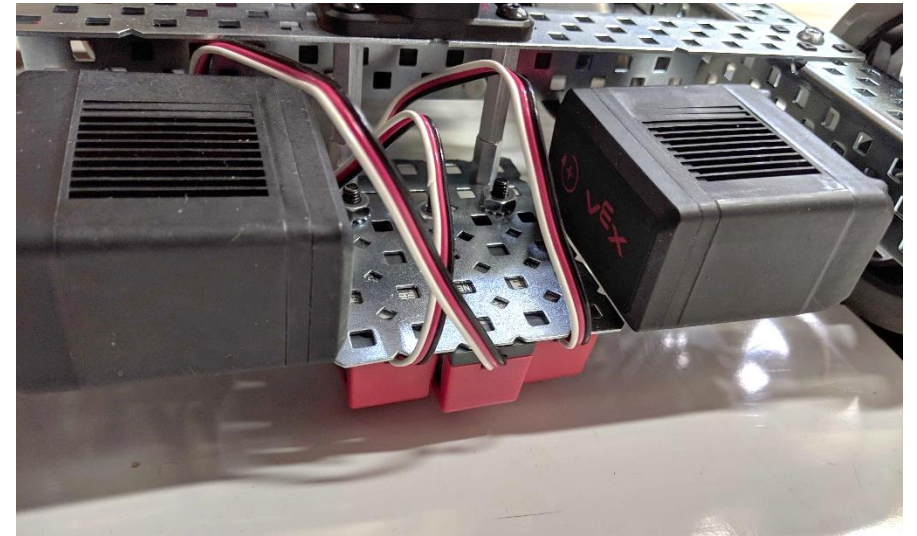
## Measuring Thresholds – Part 2

4. Place the robot so that the Line Tracking Sensors are above the **dark** surface/line.
5. Record the values of the Line Tracking Sensors. You can choose to use the 12bit number or percentage in your program.
  - We will use percentage.
  - It is normal for there to be some variance among the three sensors.



## Measuring Thresholds – Part 3

- Place the robot so that the Line Tracking Sensors are above the **light** surface/line.
  - Record the values of the Line Tracking Sensors. You can choose to use the 12bit number or percentage in your program.
    - We will use percentage.
    - It is normal for there to be some variance among the three sensors.
- Notice that the light values are lower than the dark values!**



## Calculating the Threshold

- A good Threshold value separates the range of many values into two categories: **Light** or **Dark**.
- Sensor readings **above** the Threshold are **Dark**, and **below** are **Light**.
- We can use the following formula to calculate Threshold:

$$\text{THRESHOLD} = \frac{\text{LIGHT VALUE} + \text{DARK VALUE}}{2}$$

$$\text{THRESHOLD} = \frac{31 + 68}{2} = \frac{99}{2} = \mathbf{50}$$

- Values vary by robot, lighting condition, and measured surface!
  - Make sure to calculate the best Threshold for your environment!

# Line Tracking Sensors Setup and Reading Values

- Setup: Before task main()
  - **line rightLineTracker(Brain.ThreeWirePort.F );**
  - **rightLineTracker** is the what the sensor is named in this example. The programmer can pick a different name. It must start with a letter, no spaces, no punctuation, not a reserved word, not used somewhere else and is descriptive.
- **Reading Values** : Inside task main()
  - **rightLineTracker.value(analogUnits::range12bit)**
    - Returns a value 0 to 4096
  - **rightLineTracker.value(analogUnits::pct)**
    - Returns a value 0 to 100



# Line Tracking/Light Sensor Sample Code

```
// define your global instances of motors and other devices here
```

```
light rightLineTracker(Brain.ThreeWirePort.F );
```

```
light centerLineTracker(Brain.ThreeWirePort.G );
```

```
light leftLineTracker(Brain.ThreeWirePort.H );
```

```
int main()
```

```
{
```

```
  while(true)
```

```
  {
```

```
    Brain.Screen.printAt( 10, 50, "Right Tracker 12 bit %d", rightLineTracker.value(analogUnits::range12bit) );
```

```
    Brain.Screen.printAt( 10, 70, "Right Tracker percentage %d", rightLineTracker.value(analogUnits::pct) );
```

```
    Brain.Screen.printAt( 10, 90, "Center Tracker 12 bit %d", centerLineTracker.value(analogUnits::range12bit) );
```

```
    Brain.Screen.printAt( 10, 110, "Center Tracker percentage %d", centerLineTracker.value(analogUnits::pct) );
```

```
    Brain.Screen.printAt( 10, 130, "Left Tracker 12 bit %d", leftLineTracker.value(analogUnits::range12bit) );
```

```
    Brain.Screen.printAt( 10, 150, "Left Tracker percentage %d", leftLineTracker.value(analogUnits::pct) );
```

```
    // Allow other tasks to run  
    task::sleep(100);
```

```
  }
```

```
}
```

Code Break:

Enter the code for up to three of the Line Tracking Sensors and run it to see the values returned.

# Line Tracker Activities/Reinforcement Activity

- 'Out of line driver's' helper: Write a program to inform a driver that they are driving on the line using the line tracker and haptic feedback.
  - Calculate thresholds for the line tracking sensors
  - Have the remote shake when the sensor is on a black line
- Brain Screen
  - Using a graphic on the screen, change the color based on sensor reading
    - White for light
    - Red for dark