**Problem\_00:**

Any robot design can be used for this guide. Every problem in this tutorial will use the same drive function over and over again. The drive function you will be using is designed for an arcade style controller with only two motors. This function can be easily changed to adapt to any drive style or wheel setup.

The drive function takes two arguments which is throttle and steering. Throttle is used to control the speed of forward and backward movement while steering is to control the sped of turning left and right. Positive being forward and negative being backwards. The steering argument controls the speed and direction on which the robot will turn. Positive is to the right and negative is to the left. The reason the function is designed this way is because it can be used to control the robot with a control and for autonomous functions. The controllers’ joystick axis channel three and channel two are positive when pushed up and negative when pushed down. The controllers’ joystick axis channel four and channel one is positive to the right and negative to the left. The functions signature is the following.

*void drive(short throttle, short steering);*

You can include the functions file into your project or you can copy and paste the function.

All projects must include the same motor names and sensor names as described in the instructions or you must change the names in the functions. The name of a sensor or motor will always be the same in this guide.

All code will be shown in pseudo code so you can adapt the ideas to any robotics language. All source code is available at [https://github.com/MicahaelWallace30/MtsacGuideCode.git](https://github.com/MicahaelWallace30/MtsacGuideCode.git%20) for ROBOTC 3.62, but it is recommended not to use the source code so you can learn what you are doing and how to fix common problems such as syntax errors, logic errors, failed electronics, and new versions or different compilers. After you can look at the code to see how you can improve your code.

The Drive\_Function.h file will grow and include more useable function over the course of this guide.

If you don’t know how to setup motors and sensors open the document called “RobotC Setup”. If you don’t know what a variable is, how to create a loop, or what a function is open up the basic programming guide called “Programming Basics”.

**Problem\_01:**

Your robot should be setup and able to use the drive function to move the robot forward, backward, left, and right uncontrollably. If you cannot do this you cannot continue.

The easiest way to control the movement of the robot is from the microcontrollers internal. Vex cortex which is the required micro controller of the time this guide was written has four internal timers which are T1, T2, T3, and T4. All four timers can be accessed in milliseconds. If timers are used you should clearly associate timers with robot functions. An example is to have T1 for drive, T2 for lift, and T3 for feed to prevent timer conflicts. Any time you use a timer it should be cleared to prevent error build up or have the timer overlapping back to zero seconds. You can read the timer in milliseconds, ten milliseconds, and hundred milliseconds. To stop the robot just set the throttle and steering to zero.

ClearTimer(T1); //clears timer 1

time1[T1]; //return the value of timer 1 in milliseconds (5000ms) = 5000

time10[T1]; //return the value of timer 1 in 10 milliseconds (5000ms) = 500

time100[T1]; //return the value of timer 1 in 100 milliseconds (5000ms) = 50

wait1Msec(some\_integer);//does nothing for some integer milliseconds

wait10Msec(some\_integer);//does nothing for some integer 10 milliseconds

wait100Msec(some\_integer);//does nothing for some integer 100milliseconds

Here is a simple pseudo code to show you how to use a timer to control robot movements:

clear timer 1

while timer 1 is less than 5 seconds

drive forward

end while

stop drive

clear timer 1

while timer 1 is less than 2seconds

turn left

end while

stop drive

clear timer 1

while timer 1 is less than 5 seconds

drive forward

end while

stop drive

drive forward

wait 1000 ms

stop drive

**Common Issues, solutions, and tips:**

I recommend playing with the timer length and drive speed to see how it works. Motor speeds for Vex are from -127 to +127, yet it does not become faster around +- 85. Adjust the code so the robot gets exactly back to where is started in the right direction.

Any time you set a value of a motor it will stay that value unless you change it. If you set a motor value to 100 and call the wait function the motor will still run at 100 for the duration of the wait function.

You should not use timers for robot movement because the chance of a robot traveling the same distance for the same amount of time is unlikely. Timers are very useful to check for robot errors and have alternative plans for the robot which will discussed later.

**Problem\_02:**

One solution to control the robots movement is to use encoders which are digital sensors that keep track of the rotation of the shaft. There are Quadrature Encoders, Single Wire Encoders, and Integrated Motor Encoder (I2C). Quadrature Encoders are able to keep track of bidirectional rotation while the Single Wire Encoders does not keep track of direction only movement. Quadrature Encoders are the preferred type of encoders for Vex robotics. Integrated Motor Encoder currently have a static issue where static electricity is sent back into the cortex from the integrated motor encoders and reset or change the count value. I recommend not using Integrated Motor Encoders for drive motors which seems to have the most issue from that static build up on the competition mats, yet you can use the Integrated Motor Encoders for lifts and feeds with little to no issues.

For this exercise we will use a combination of two Quadrature Encoders one for each side of the robot. Quadrature Encoders send back values of integers with a total of 360 for one full rotation of the shaft that is inserted in the encoder.

RobotC quadrature and single wire encoder code:

//sets the value of the encoder sensor to some number

SensorValue(*Encoder\_Name*) = some\_integer;

//gets the value of the encoder and stores it into a variable

variable = SensorValue(*Encoder\_Name*);

With the encoders it is always a good idea to set the value to zero before you use them to prevent build up error. If the Quadrature Encoder is reading negative when going forward you can just switch the two wires or always negate the value of the sensor. If you do not want to worry about the sign of the encoder value you can always read the absolute value of the sensor.

We will now perform the same route as in exercise 01, but with only encoders. Here is the pseudo code for the problem. All sensor values for encoder are assumed to be absolute values in the pseudo code. Try and get the robot to end up in the same exact spot again or change the routine to any route you please.

Here is a simple pseudo code to show you how to use a encoders to control robot movements:

clear both encoders

while left encoder Value and right encoder value are less than a number

drive forward

end while

stop drive

clear both encoders

while left encoder and right encoder is less than a number

turn left

end while

stop drive

clear both encoders

while(left encoder and right encoder are less than a number

drive forward

end while

stop drive

**Common Issues, solutions, and tips:**

One of the problems which can happen is the robot will get stuck during a competition and if the robot is waiting to reach a specific encoder value the robot will cause havoc. A solution to this problem is to add a timer to check if the robot should have made the distance and if it has not than stop or do something else. Just create a function which returns true if the correct distance was made or false if time has run out then you can just check the value of the function to decide what to do next.

If you do not add waits in-between change of direction the robot will jerk around and change the distance it travels depending on the lack of roll from the change of direction.

You might notice depending on the speed and weight of the robot the robot will not stop exactly at the number you told it to which is due to lack of breaks and the robot rolling. This problem can be fixed by either implementing a breaking system or a proportional encoder function which will be discussed later.

To see the actual robot roll and not end up at the intended encoder value you can find the value of the encoder takes to drive one meter and double the value of the encoder and see if the robot will got two meters.

If you want to make a function which is specific to your robot to determine the distance it will travel you can measure the circumference or radius of the wheel than you can create the function to calculate the distance per rotation and find the distance of one value of the encoder sensor. For example if the wheel circumference is 50 cm then 360 values of the encoder will equal 50 cm hence one value is 0.138 cm.

The actual code has many function examples of how to make your job much easier and reduces the amount of code you will have to write.

**Problem\_03:**

The next sensor available for you to use is an analog sensor called a Potentiometer. Vex potentiometer can only do one rotation and should not be used for drive motors. Potentiometers are ideal for lift motors or shafts. You need to put the potentiometer on a shaft that does not complete more than one full rotation.

You need to physically set the potentiometer to read around zero for the bottom of the lift. You can do this by running an infinite loop on the IDE and use the sensor debugger to see the change in the potentiometer. You will need to mark down the value of the desired heights for at least the bottom and top of the lift. Potentiometersvalues cannot be changed, so you need to make sure you place the potentiometers in the correct position so it will completely rotate and increase in value when desired.

//reading the value of the potentiometer

variable = SensorValue(pot\_name);

Pseudo code to control a simple one arm one degree of freedom lift

MAX\_LIFT\_HEIGHT = some value

MIN\_LIFT\_HEIGHT = some value

while potentiometer value is less than MAX\_LIFT\_HEIGHT

lift motor power set to full

end while

lift motor power set to value it will hold lift up and not fall

wait some time

while potentiometer value is greater than MIN\_LIFT\_HEIGHT

lift motor power to negative one fourth full power

end while

lift motor power set to zero

**Common Issues, solutions, and tips:**

A Potentiometer is nothing more than a variable resistor which is used to measure the voltage. The fact the potentiometer is analog and works by measuring change in resistance therefore no potentiometer is the same. If you put two potentiometers on one shaft they will not be the same values for the whole rotation which I suggest testing out.

You can use two potentiometers for each side of a lift, but if the lift does not differ from side to side you only need one potentiometer. The good thing about potentiometer is if the lift does not slip at all the values of the potentiometer for the desired height will always be the same.

If the potentiometer is reading negative for up and positive down you can simply negate the value if you don’t want to change the potentiometer, but I do not suggest this to avoid confusion.

Another use for a potentiometer is autonomous routine selection. All you need to do is find a range of values for a position and have the program check to range to decide a routine.

**Problem\_04:**

Next sensor is just a button which can be used to check if a mechanical part has hit the button. You can check if robot has run into something or if a lift or feed has reached its closing point. There is not much use for a button and they use a digital port so I do not recommend using many if any. The digital button will return 0 if not pressed and 1 if pressed.

//check if button is pressed

if(SensorValue(ButtonFront) = 1)

Pseudo code to have robot drive around like a mad man running into walls

while forever loop

while front button is not pressed

drive forward

end while

random direction left or right store in variable

turn random direction in reverse for duration of encoder count to make

end loop

**Common Issues, solutions, and tips:**

Void

**Problem\_05:**

The next sensor available is an analog sensor which detects the amount of light and can be used as a line sensor. Line sensors can be used to detect when you cross a line or are used to follow lines. Most people in Vex you a combination of three line sensor for line following. There is an optimal distance which the line sensor should be positioned from the ground, but as long as you can see a difference between the mat and line there shouldn’t be an issue. The optimal distance greatly increases the range of values you will pick up with the line sensor.

To detect if the line sensor sees a line or floor you should write down the value of the sensor on the line and the value on the floor and determine the middle value of the two. If the value for the line is greater than the floow to detect if the robot is on the line you just check if the sensor value is greater than the middle value. If the line is lighter than the floor you would check if the sensor value is less than the middle point.

//To store a line sensor value in RobotC in a variable:

Variable = SensorValue(lineSensorName);

Pseudo code to drive to a line and stop on black tape with white floor:

Declare constant integer value THRESHOLD of (line value + floor value) /2

while line sensor value is less than THRESHOLD

drive forward

end while

stop drive

Pseudo code for simple line follower to follow a black line on white floor for distance:

Declare constant integer value THRESHOLD of (line value + floor value) /2

while encoder distance is less than desired distance

variable left = value of left line sensor

variable middle = value of left line sensor

variable right = value of left line sensor

if middle is greater than THRESHOLD

drive forward

else if left greater than THRESHOLD

drive right

else if right greater than THRESHOLD

drive left

else

robot has lost line what should it do?

end while

stop drive

**Common Issues, solutions, and tips:**

Line following is one of the more challenging things a Vex robot can do. Just to follow a line isn’t hard, but to find a line and follow the line properly and decide what to do if lost or finds a cross section of lines is harder.

One of the solutions to handle the case when a robot loses track of the line is turn sharper in the last direction. Another solution if the line is lost is to strafe in the direction of the last line sensor to see the line.

If the robot is going so fast is crosses the line for just the line detection the best solution is to back up slower to find the line once again. There is no adequate proportional line detection formula.

There is many forms of PID used to follow a line. One of the simplest methods to follow a line with PID is to use one line sensor and determine if the line sensor is most on the line or most on the floor and turn in proportion to the difference, but this will be discussed later.

A complicated line following program is no always the best solution for vex. If it gets the job done who cares what I looks like just make sure its consistent.

**Problem\_06:**

The next sensor you can use is a digital sonar sensor which can be used to measure the distance to an object from the sonar sensor. The sonar sensor sends back raw data which most programing languages convert to cm, mm, or inches. The smaller the increment of measurement the more error you will notice with sonar sensors. You can play with the different measurement increments and see what fits your robot event best. Sonars are very easy to use once they are set up, yet they are not reliable.

//get value of sonar into variable

Variable = SensorValue(SonarName);

Pseudo code for sonar distance detection.

while sonar sensor distance is greater than 0 and greater than some distance

drive forward

end while

stop drive

**Common Issues, solutions, and tips:**

Just like the encoders the robot will probably roll past the desired distance wanted within the wall. Sonar sensors have a big problem of returning bad readings. If the sonar hits an angle it will return bad values. The cheaper the sonar the more bad values you will see. There are a couple ways people try and fix this with find the middle reading of x amount of readings, but really if you get bad readings you will get them a lot and there is no sure way of removing them. Sonars are also very limited on the distance they will be accurate. Just beware of the risk of using sonars and try not to use them too much.

I suggest hooking up a sonar sensor to a cortex and running a loop to see the output values of the sonar. Attempt to move it around and notice how many bad values it returns.

**Problem\_07:**

A non-Vex sensor which is an analog sensor known as an Infrared distance Sensor can be used in replace of a sonar sensor to measure the distance of an object. An Infrared sensor commonly called an IR sensor is not the same as an IR distance sensor. An IR sensor only detects object within a range you want to use an IR distance sensor. There are many brand and types you can purchase depending on the distance you desire.

The easiest way to us an IR sensor with Vex Cortex is by plugging in the IR distance sensor you want to use to an analog port and read the raw values for increments of distances. For example if you have an IR distance sensor which reads ten centimeters to eighty centimeters you would start at ten and increment by two or five centimeters all the way to eighty centimeters logging the raw value with the actual distance. Once you have a table of raw values with matching distance values you can input these numbers into a graphing program like Logger Pro which is used by physic student in Mount San Antonio College or any free graphing program with will perform curve fits to your data. After you create the graph and apply a power fit you will know have the desired function to determine the distance of the IR distance sensor.

//get value of IR distance sensor into variable

Variable = SensorValue(SonarName);

You can repeat the same exercise with IR distance sensor as you did with the sonar and find the dead area in front of the IR sensor. How can you solve this problem? There is no right solution. If it works it works.

**Common Issues, solutions, and tips:**

IR distance sensors are more reliable than a sonar sensor and only use one analog port compared to two digital ports. Choose the best option for your robot. The Vex Cortex does not have a FPU (floating-point unit) which brings up the issue about the IR distances sensors function which is the lack of FPU will decrease the speed of processing the data . A solution to this problem is to create a linearized function that does not use floating point numbers which can be accomplished by following the linked website for linearizing sharp ranger data or any other way of linearizing a function. A problem with linearizing the IR distance sensor is you will lose accuracy of the sensor. Again it’s up to you to choose which is best for you speed or accuracy. <https://acroname.com/articles/linearizing-sharp-ranger-data>

**Problem\_08:**

This section will cover proportional control very quickly because it is so easy and useful. Proportional control is part of PID control which you can research on your own or use the links provided in the links.txt. Proportional control can be used for any type of sensor which is measure some king of distance or speed. Proportional control looks at the desired value and the current value to find how much speed or power should be applied to your problem. We will look at how to solve the problem of rolling robots with proportional control.

//pseudo

//find the minimal amount of power to move a robot will assume 50

//this will change depending on the voltage of the battery

minPower = 50;

//max amount of power you will ever want

maxPower = 100;

//drive to distance with proportional control

While(SensorValue(encoder) < desiredDistance)

{

Value = SensorValue(encoder);//reading of sensor

Error = desiredDistance – value;//difference of desired and current

Power = minPower + error;//calculate power

If(power > maxPower) power = maxPower;

Drive(power,0);

}

Stop drive

**Common Issues, solutions, and tips:**

It’s always better to have a larger increment of the sensor for proportional control. For example if you use a sonar sensor which outputs inches the error will average around five which is not useful for proportional control. If the error is too large you can always divide the error until you find a desirable change.

**Problem\_09:**

Tasks are a pseudo multithreading ability that RobotC uses. In a competition file RobotC will use a couple of tasks, so it is not recommended to use too many task. How many is too many depend on what you are having each task do. If you are using RobotC you will notice main is not a function, but is a task. The only time you would want to use a task is if you want to always have the robot check or do something and really don’t want to integrate the method check into every piece of your code. For example if you wanted to check if the robot has been active for X amount of time you would need to check ever the amount of time passed in every part of your code frequently. Instead of integrating the function to check time you can use a task to check all the time. For the problem we will create a routine for the robot to run and stop the robot at X amount of time no matter what.

//declare a task

task nameOfTask()

{

//code

}

//start task

StartTask(nameOfTask);

//stop task

StopTask(nameOfTask);

**Common Issues, solutions, and tips:**

There is no real multithreading in the Vex cortex, so you want to make sure any task you create does not hog the CPU. To make sure a task does not hog the CPU you should add waits end the end of each task loop to give time for other tasks to operate. The wait time can be as little as 2ms or 5ms or larger if desired. You can also look up task priority to delicate how important each task is.

**Problem\_10:**

The last Vex sensor that should be mentioned is the Vex Gyro which is an analog single axis gyro used to measure mostly turns. We do not currently have any Vex gyro sensor, but there is a nice library made by QCC2 which you should use or look at for examples of how to efficiently use the Vex gyro sensor. The current link can be found in the Links.txt file or <https://github.com/JMMcKinneyWPI/GyroPIDLibrary>.