Making Everything Easier!™

Novelty Edition

How to Program in ROBOTC

FOR ~~DUMMIES~®~~

BEGINNERS

Learn to:

- Write Drive Code
- Program an Autonomous
- Use Sensors and Encoders

Team 1437
Patriot Robotics
Viewpoint School

CONTENTS

What is ROBOTC?

ROBOTC is a C-based programming language that is used to create and execute programs for VEX and Lego Mindstorms. Knowing how to program in ROBOTC is essential for participating in VEX Robotics Competitions.

This guide to ROBOTC focuses on version 3.05 of ROBOTC for Cortex and PIC and we use a VEX Cortex and a VEXnet Joystick controller.

Section I: The Basics

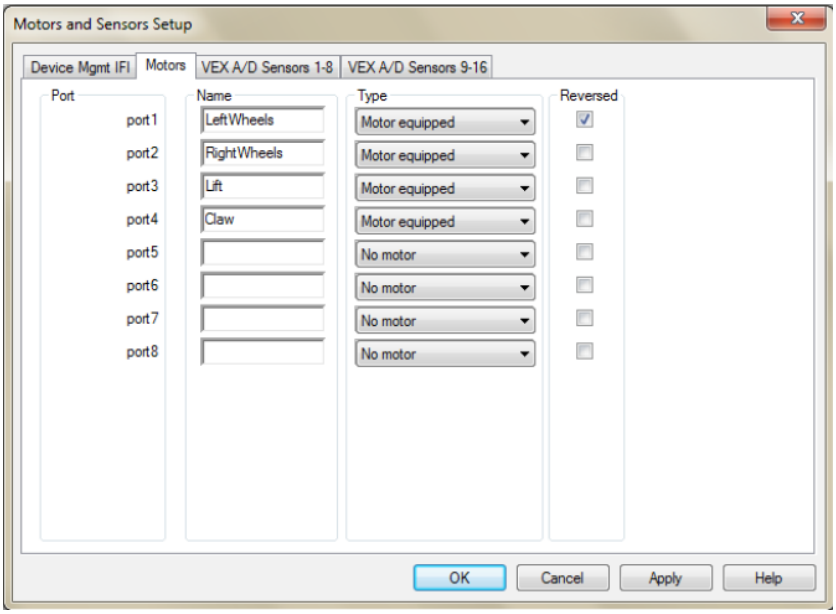
Getting Started

The first thing you will want to do when you open ROBOTC is make a new document. When you click on File and look under New you will see an option for "Competition Template". The Competition Template is a template for how you must structure your code for participation in VEX competitions. As you can see in the image below, ROBOTC will automatically write #pragma and #include lines for you and then create three sections for pre_auton, autonomous, and usercontrol. We will look at each of these sections in detail later. Once you have your blank Competition Template, you are ready to get started.

```
1  #pragma config(Sensor, in1,    bumperswitch,        sensorTouch)
2  #pragma config(Motor,  port1,           leftwheels,    tmotorNormal, openLoop, reversed)
3  #pragma config(Motor,  port2,           rightwheels,   tmotorNormal, openLoop)
4  //*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
5
6  #pragma platform(VEX)
7
8  //Competition Control and Duration Settings
9  #pragma competitionControl(Competition)
10 #pragma autonomousDuration(20)
11 #pragma userControlDuration(180)
12
13 #include "Vex_Competition_Includes.c"   //Main competition background code...do not modify!
14
15
16 void pre_auton()
17 {
18 }
19
20
21 task autonomous()
22 {
23
24
25
26
27
28
29
30
31
32
33 }
34
35
36
37 task usercontrol()
38 {
39
40
41
42
43
44 }
```

Configuring Motors

Motors are a very important part of any robot; they allow us to drive, intake, lift, etc. Before we can even start writing drive code, we need to identify all of our motors and distinguish one from another. In ROBOTC, this is accomplished by using the #pragma lines at the very beginning of the document. A #pragma line typically looks like this: #pragma config(*Sensor*, *Port#*, *Name*, *SensorType*, openLoop). While you could type out a #pragma lines for every motor, it is much easier to go the Robot menu and select "Motors and Sensors Setup". You should see a new window open that looks similar to the one below. Here you can name each of your motors according to their function and you can identify them as motors to ROBOTC (make sure that the port# indicated is the same as the one on your VEX cortex). Selecting the option for reversed will cause that motor to spin in the opposite direction which is necessary when you have to motors facing one another.




```
#pragma config(Motor,  port1,           LeftWheels,    tmotorNormal, openLoop, reversed)
#pragma config(Motor,  port2,           RightWheels,   tmotorNormal, openLoop)
#pragma config(Motor,  port3,           Lift,          tmotorNormal, openLoop)
#pragma config(Motor,  port4,           Claw,          tmotorNormal, openLoop)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
```

These #pragma lines should look similar to yours once your motors have been configured. Note that the sensor type for motors is almost always tmotorNormal. Also, the motor controlling the left wheels of this robot has been reversed since it faces the motor that controls the right wheels.

One important reminder when working on a computer is to save early and save often. You never know what will happen and ROBOTC does not save files after a crash.

Write Drive Code

Now that our motors have been configured, you can write some drive code. Since you want to drive your robot, the drive code needs to be written in the usercontrol section. Drive code and any other code that involves you using the controller to make your robot do something belongs in this section.

All motors work by taking in a number that corresponds to a speed value; that speed determines how quickly the motor rotates. The simplest form of drive code is to have the drive motors receive a speed value from the VEX joystick controller. The VEX controller below shows number labels for each channel (stick) or button. In ROBOTC, we can call on the values of these buttons and channels using vexRT(Channel # or Button label). For example, vexRT(Btn7U) or vexRT(Btn8L) will return boolean values (true when pressed and false when not); vexRT(Ch3) will return a number from -127 to 127 when the left joystick is moved in the vertical direction. Our motors take in a speed value that can be any number from -127 to 127; this means that we could control one wheel or one side of a robot just using Channel 3 of our joystick controller. For drive code, it is best to program using the channels.


```
task usercontrol()
{
  while (true) //Can also be while (1 == 1) etc.
  {
    motor[LeftWheels] = vexRT(Ch3);
    motor[RightWheels] = vexRT(Ch2);

  }
}
```

This is a perfect example of a simple drive code. The first thing we did in our usercontrol section is write a while -loop; a while-loop is a way for our program to keep running constantly so that our robot responds to our joystick commands. Without a while-loop, we would have to wait each time we wanted our robot to move differently; this way, when we move a joystick from all the way up to all the way down, our program will immediately change the speed of all the motors controlled by that joystick channel. Next, we called on the speed value of the motor controlling the left wheels and set it equal to the channel 3 value on the joystick. When the joystick is not being moved, the value of the channel is 0 so the motor will not rotate; once we move the joystick up or down, the channel value will become the motor speed value and the wheels will move accordingly. This will also hold true for the motor controlling the right wheels.

Download a Program to the Cortex

Now that you have a simple drive code, you should test it and make sure that it works. To do this, you will need to download your program onto your VEX Cortex. First, you should press F7 and make sure that your program compiles properly without errors. Assuming that your cortex and controller have been connected properly, you should then connect your joystick controller to the computer using a USB to USB cable. Once you have connected the two and made sure your controller and cortex are turned on and linked, press F5 or go to the Robot menu and select "Compile and Download Program". A progress bar should appear to show that your program is downloading. Once downloaded, you can disconnect your controller from the computer and start testing your code.

If you wish to be able to quickly edit, download, and test your code, you can keep your controller connected to the computer. In this case, you should open the Debugger under the Robot menu and the ..... menu. The .... menu will allow you to chose which code to run (pre_auton, autonomous, or usercontrol). To test your drive code, select usercontrol and then click Start in the Debugger. You will now be able to control your robot using the controller. Since these are useful windows to have open, you may wish to move them to the side when you're working instead of closing them.

Write an Autonomous

With a working drive code, you're now ready to start looking at making an autonomous mode for your robot. In VEX competitions, the autonomous mode is a 15 second period where your robot runs entirely on pre-programmed commands in the autonomous section of your document. Although you have absolutely no control of your robot during these 15 seconds, you can test your autonomous code before a competition and make sure it does what you want it to do.

One way to make an autonomous mode is by using time. To do this, you simply program your robot's actions in the order you want them to be executed. In between these actions, you In ROBOTC there are functions like wait1Msec(# of milliseconds) and wait10Msec(# of milliseconds) which create wait periods or periods when the program essentially does nothing for a set period of time. This becomes useful when you need a certain action to continue for awhile and then stop or change.

```
motor[LeftWheels] = 127; //Full speed forwards
motor[LeftWheels] = 127; //Full speed forwards
wait1Msec(5000); //Five seconds later...
motor[LeftWheels] = -127; //Full speed backwards
motor[RightWheels] = 127; //Full speed forwards
wait1Msec(1000); //One second later...
motor[LeftWheels] = 127; //Full speed forwards
motor[RightWheels] = 127; //Full speed forwards
wait1Msec(2500); //Two and a half seconds later...
motor[LeftWheels] = 0; //Stop
motor[RightWheels] = 0; //Stop
```

In the code sample above, for example, we want to go forward, backward, forward again, and then stop. While it is easy to give the right speed values to the motors, we need to tell the program how long to let each of these things happen. By looking at the

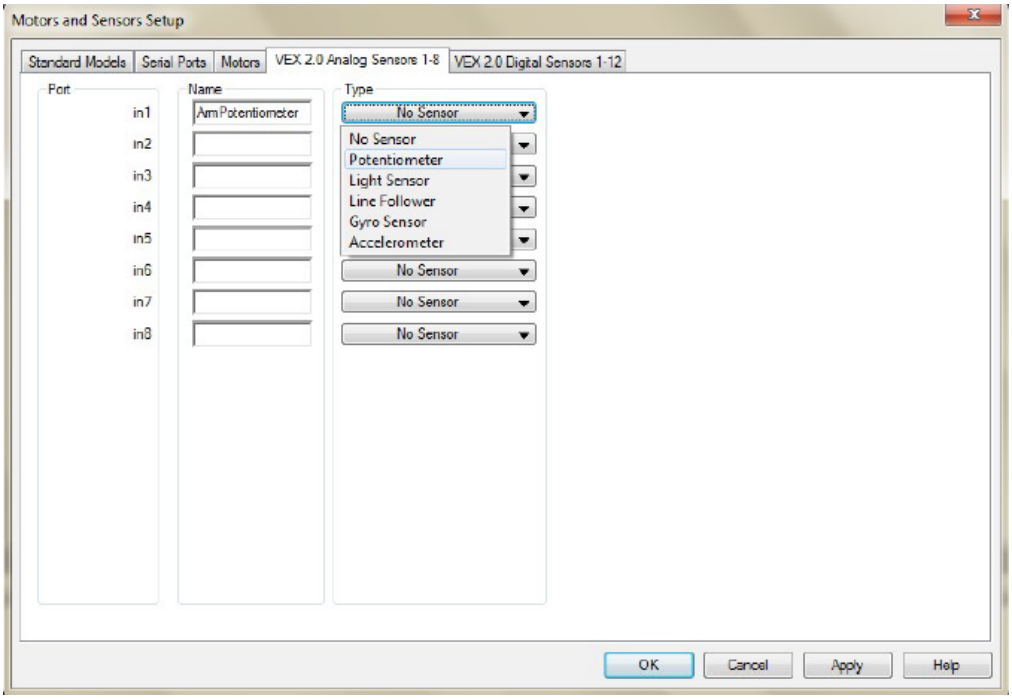
field, we can estimate how long we want each command to continue. The functions wait1Msec() and wait10Msec() both take in a number of milliseconds; if you want an action to continue for 3 seconds, you can use wait1Msec(3000) since 3 seconds is really 3 milliseconds.

As we mentioned earlier, it is important to test your autonomous mode to ensure its accuracy. This becomes especially important when dealing with a time sensitive autonomous; any error in the positioning or placement of the robot could cause a mistake or make the robot do something too early/late.

Section II: Using Sensors

Sensor Setup

Sensors are useful tools that can help make your programming more precise and make your robot more effective. While there are various types of sensors, we will focus on bumper switches and encoders. In order to use any sensors, they must also be identified by the #pragma lines at the beginning of the document just like our motors. To do this, go back to the File menu and select "Motors and Sensors Setup. As you can see below, you will need to give each sensor a name and identify what type of sensor it is. Once your sensors have been configured, you are ready to start programming with them.



Bumper Switches

Bumper switches are basically touch sensors that can be used to make a robot do something when the sensor is bumped. It is like a joystick controller button in the sense that the value it returns is 1 when it is pressed and 0 otherwise. You can use these sensor values with conditionals to make your robot do something when the switch is pressed and a completely different thing when it is not.

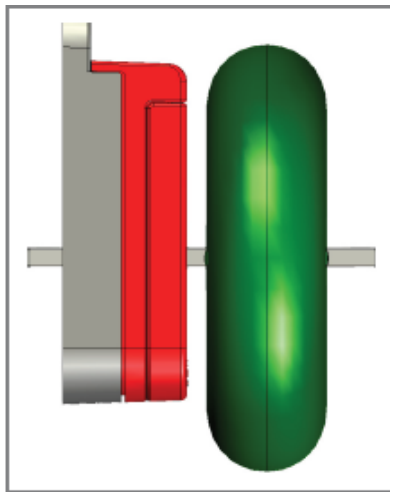
```
#pragma config(Sensor, in1,    bumper,              sensorTouch)
#pragma config(Motor,  port1,           leftwheels,    tmotorNormal, openLoop)
#pragma config(Motor,  port2,           rightwheels,   tmotorNormal, openLoop, reversed)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

task usercontrol()
{
  while(true)//The code needs to be constantly refreshed to work
  {
    if(SensorValue(bumper) == 0)//if the bumper is not pressed
    {
      motor[leftwheels] = 127;
      motor[rightwheels] = 127;
    }
    else                        //if the bumper is pressed
    {
      motor[leftwheels] = 0;
      motor[rightwheels] = 0;
    }
  }
}
```

In this code sample, we are using the bumper switch to tell our robot when to stop moving. To do this, we use an if-else statement inside of the while-loop. The rule to follow is that for every if, there must also be an else. In this case, the "if" part of the statements checks to see if the bumper switch is not pressed; if this is true, then the robot keeps moving forward. The "else" part of the statement says that if the button is being pressed, the speed value to both motors will become zero and the robot will stop moving.

Encoders

Encoders are very useful sensors that can be particularly helpful when programming an autonomous mode. Unlike other sensors, encoders require two connections to the cortex. By inserting the axle into an encoder, the encoder will keep track of how many rotations the axle (and wheel) make. Using the number of rotations to write an autonomous mode is an alternative to using timed sequences of commands.



The code sample below shows how you could use encoders to write a function that allows your robot to reverse in a straight line for 5 rotations. By constantly monitoring the rotations of each wheel, the program can correct the motion of the robot until both wheels are moving at the same speed. The condition of the while-loop will stop the program from executing after 5 revolutions have passed.

```
while(SensorValue[rightEncoder] > -1800) //Reverse for 5 full rotations
{
  if(SensorValue[leftEncoder] < SensorValue[rightEncoder])
  {             //If the left side has reversed more than the right...
     motor[port3] = -50;       //slow down the left...
     motor[port2] = -63;       //and speed up the right
  }
  if(SensorValue[leftEncoder] > SensorValue[rightEncoder])
  {             //If the right side has reversed more than the left...
     motor[port3] = -63;       //speed up the left...
     motor[port2] = -50;       //and slow down the right...
  }
  if(SensorValue[leftEncoder] == SensorValue[rightEncoder])
  {             //If the left and right have reversed the same amount...
     motor[port3] = -63;       //run the motors at the same speed
     motor[port2] = -63;
  }
}
```