

The Beginners Guide to **ROBOTC** By GEORGE GILLARD

Table of Contents

Introduction to ROBOTC

Part 1 – The Basics

- Section A – Setting up
- Section B – Writing a Drive Code
- Section C – Downloading a programme
- Section D – Making a basic Autonomous

Part 2 – Including basic sensors

- Section A – Introduction to Sensors
- Section B – Bumper and Limit Switches
- Section C – Potentiometers

Part 3 – Extras

- Section A – using the competition template
- Glossary

Introduction to ROBOTC

ROBOTC is an application used for programming robots. There are many different versions of ROBOTC, but I highly recommend using the latest version, and use the same version across your whole team. That way, different people in your team can programme the same robot without having to download the firmwares every time.

In this guide to ROBOTC, I'm using version 3.05 of ROBOTC for Cortex and PIC, a VEX Cortex Microcontroller and a VEXnet Joystick. ROBOTC for Cortex and PIC is able to programme for both Cortex and PIC, however, there is also ROBOTC for IFI, which is simply just ROBOTC for PIC. Everything taught here applies to the VEX Cortex microcontroller and VEX PIC, though there are some differences, which are explained.

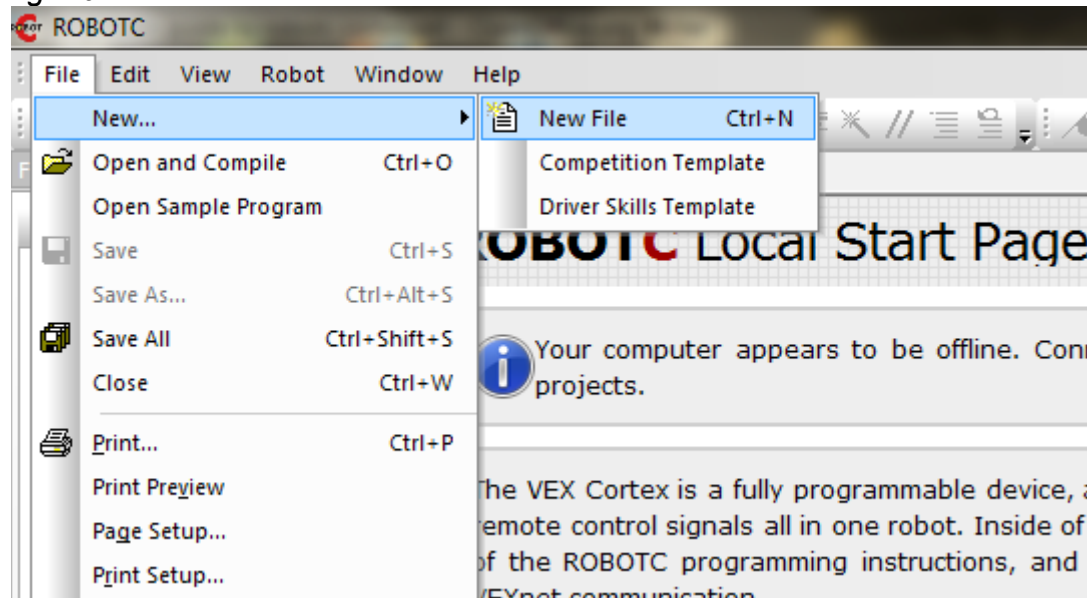
There are also many different methods of programming, this guide has been written the way that I programme.

Part 1 – The Basics

Section A – Setting up

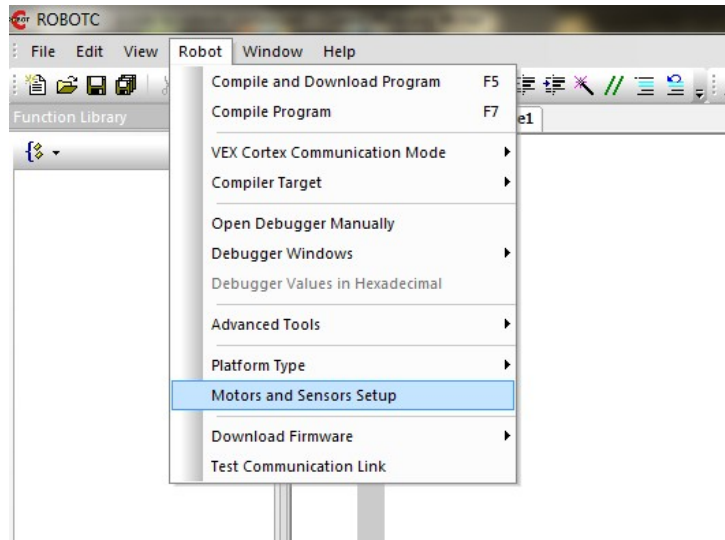
Once you have opened ROBOTC, open a new file (File → New → File, or Ctrl + N, *Fig 1.01*). First, lets start by making just a basic “drive code”, then we'll move onto the competition code.

Fig 1.01



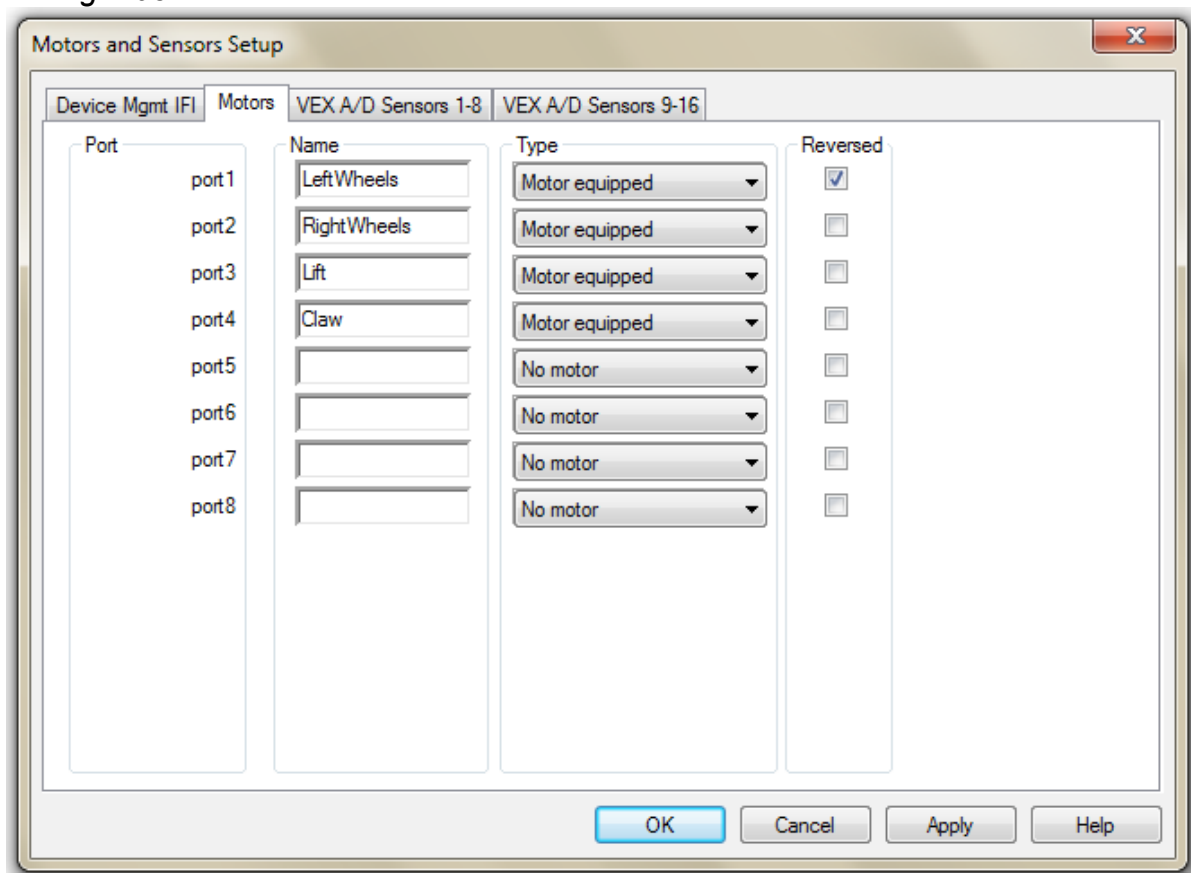
In your new file, use the toolbar to go to, Robot → Motors and Sensors setup. (*Fig 1.02*)

Fig 1.02



It will open a window where you enter what you want to name the motor in the motor port gap and tick the box at the end of the line if it needs to be reversed. Using my “test bot” I have set up the motors. (Fig 1.03) Lets say that the left wheel's motor needs to be reversed.

Fig 1.03



Once you have completed setting up your motors, click “OK”. Your code should now look something like this (Fig 1.04):

Fig 1.04

```
LocalStart.htm | SourceCode1*
1  #pragma config(Motor, port1, LeftWheels, tmotorNormal, openLoop, reversed)
2  #pragma config(Motor, port2, RightWheels, tmotorNormal, openLoop)
3  #pragma config(Motor, port3, Lift, tmotorNormal, openLoop)
4  #pragma config(Motor, port4, Claw, tmotorNormal, openLoop)
5  /**!!Code automatically generated by 'ROBOTC' configuration wizard      !**//
6
7
```

Now is a good time to save your code. To save, click File → Save or Ctrl + S. If this is the first time you have saved the programme, it should automatically open a “Save As” window, but if not, click File → Save As. It is a good idea to regularly save your code, as it will not save by itself. Also, if ROBOTC crashes, it doesn't save your programme.

Once saved, simply type in your code:

```
task main()
{
}
}
```

“task main” is the main task. In a competition template, there is a “pre_auton” function, where you reset sensor values, an “autonomous” task, where you write your autonomous code and a “usercontrol” task which is where you write the code that allows you to drive. In a task or a function, you always write your code between the two curly brackets. These show the beginning and end of a task/function.

You have now set up the basic code! In a drive code, this is where you will type the code which enables the robot to be driven by a human operator. Task main can be used for testing autonomous as well.

Section B – Writing the Drive Code

A Drive Code is a code which allows the robot to be controlled by a joystick or transmitter. The way we write this is (the motor speed) = (the transmitter channel).

When we write about a motor, we write:

```
motor[motor name here]
```

On a VEXnet Joystick, there are 8 channels. They are:

- Channel 1, the sideways channel on the right thumb-stick
- Channel 2, the vertical channel on the right thumb-stick
- Channel 3, the vertical channel on the left thumb-stick
- Channel 4, the sideways channel on the left thumb-stick
- Channel 5, the back buttons on the left (looking from the front)
- Channel 6, the back buttons on the right (looking from the front)
- Channel 7, the 4 buttons on the left of the joystick, on the top
- Channel 8, the 4 buttons on the right of the joystick, on the top

On a 75mHz transmitter, there are 6 channels. They are:

- Channel 1, the sideways channel on the right thumb-stick
- Channel 2, the vertical channel on the right thumb-stick
- Channel 3, the vertical channel on the left thumb-stick
- Channel 4, the sideways channel on the left thumb-stick
- Channel 5, the back buttons on the left (looking from the front)
- Channel 6, the back buttons on the right (looking from the front)

When we write about a transmitter channel **on one of the thumb-sticks**, we write

`vexRT(Chchannel number here)`

So that means that if we want to control the left wheels on Ch3 (vertical channel on the left thumb-stick), the code would be:

```
motor[LeftWheels] = vexRT(Ch3);
```

This means that the motor value will equal the value of Channel 3.

Note the semicolon at the end of the line. We use them at the end of each command (when we tell the programme something, e.g. `motor[motor name] = speed;`).

On the VEXnet Joystick AND the VEX 75mHz transmitter, the thumb-stick values range from -127 to 127, the same as the motor values. For the buttons, it is a little bit different...

The buttons on the 75mHz transmitter are either -127 (bottom button pressed) or 0 (either no button pressed, or both buttons pressed) or 127 (top button pressed). Just like for a thumb-stick, you would write:

```
motor[motor name here] = vexRT(Chchannel number here);
```

So if you wanted the lift to be controlled of the back left buttons on a 75mHz transmitter, you would write:

```
motor[Lift] = vexRT(Ch5);
```

For the VEXnet Joystick, the buttons are different. Instead, to control them, you write:

`vexRT(Btnchannel number and then direction (U, D, L or R))`

Where U is Up, D is Down, L is Left and R is Right. So, if you wanted to write about the top button (Up) on the left hand side back buttons (Channel 5), you would write:

`vexRT(Btn5U)`

In addition, the values of these buttons are also different. Instead, each button has a value of either 1 (pressed) or 0 (released). Therefore, if you wanted to control the lift from the Channel 5 buttons, you would need to write something a little more complex like one of the two examples below.

Example 1:

```
if(vexRT(Btn5U) == 1)
{
    motor[Lift] = 127;
}

if(vexRT(Btn5D) == 1)
{
    motor[Lift] = -127;
}

if(vexRT(Btn5U) == 0 && vexRT(Btn5D) == 0)
{
    motor[Lift] = 0;
}
```

This will make it so that if the top button is pressed it will lift the arm, if the bottom button is pressed it will lower the arm, otherwise, if no button is pressed it will stop the arm. This is using an if loop (explained more [here](#)).

Example 2:

```
motor[Lift] = (vexRT(Btn5U) * 127) + (vexRT(Btn5D) * -127);
```

The way the above code works, is that it will multiply (symbol *) the top button by 127 (to change it from a value of 1 to 127) and also multiply the bottom button by -127 (to change it from a value of 1 to -127) and then finally add those together. So, an example of how this works, when the top button is pressed and the bottom button not pressed:

```
motor[Lift] = (vexRT(Btn5U) * 127) + (vexRT(Btn5D) * -127);
              = (1 * 127) + (0 * -127);
              = 127 + 0;
              = 127;
```

An example of how it works when the bottom button is pressed and the top button not pressed:

```
motor[Lift] = (vexRT(Btn5U) * 127) + (vexRT(Btn5D) * -127);  
             = (0 * 127) + (1 * -127);  
             = 0 + -127;  
             = -127;
```

And if neither (or both) of the buttons were pressed, the motor value would be 0.

If you are using the PIC microcontroller, you will need to tell the programme that you are not running autonomous – it automatically assumes you are using autonomous. If you are using Cortex, you won't need to worry about switching in and out of autonomous mode. Please note this is ONLY for PIC. If you are using PIC, put this line of code to the beginning of task main(), just after the curly bracket:

```
blfiAutonomousMode = false;
```

Also, when we write a drive code, we need to put all the code in what we call a “loop”. When the robot reads the loop, it will start at the top, work its way down (like reading a book) and then if it is a while loop, it will loop back to the top again and start all over. It will do this while the condition is true. Then it will stop and continue the code after the loop. If we want the loop to be never ending, we write a condition that will always be true, such as something as simple as this:

```
while(true)
```

another condition is

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

some examples of a condition that have got ends are

```
while(bumper switch is not pressed)  
while(arm is too low)  
etc.
```

The reason we need to write the drive code in a loop is that otherwise the robot will read through the code, give the motors the value of the controller and then finish. The robot reads through code VERY fast, so all the robot will do is assign the motors the correct speeds, then do nothing more. The motors will continue at the speeds they were assigned until a new speed is given. So if there is no while loop, it will just assign the motors the appropriate speed and then you won't be able to control it anymore, and your robot will still drive at the speeds given.

If we were to write the whole drive code, using tank drive (left joystick is the left wheels, right joystick is the right wheels) with the lift and claw on the back buttons, it could be something like this (using the VEXnet Joystick):

```

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

    motor[Lift] = (vexRT(Btn5U) * 127) + (vexRT(Btn5D) * -127);
    motor[Claw] = (vexRT(Btn6U) * 127) + (vexRT(Btn6D) * -127);

  }
}

```

When we want to use 2 controllers, instead of

`vexRT(Chchannel number)`

we use

`vexRT(Chchannel number hereXmtr2)`

The 'Xmtr2' is what tells the programme that you are programming 'transmitter 2'. So say if we wanted the wheels to be on the first controller, but the lift and claw to be on a second controller (using tank drive and the lift on the vertical left thumb-stick, the claw on the horizontal right thumb-stick), we would write:

```

motor[DriveLeft] = vexRT(Ch3);
motor[DriveRight] = vexRT(Ch2);
motor[lift] = vexRT(Ch3Xmtr2);
motor[claw] = vexRT(Ch1Xmtr2);

```

The same applies for the VEXnet Joystick buttons. Simply add "Xmtr2" after you state which button.

Example:

```
vexRT(Btn5UXmtr2);
```

So now you have learnt how to write a drive code! All there is left to do is download it!

Section C – Downloading a programme

If the robot that you are using has been programmed using a different version of ROBOTC, or has never been programmed before, you will need to download the firmwares. If you are using Cortex, by clicking “Robot → Download Firmware → Automatically Update VEX Cortex” ROBOTC will update all the firmwares required. If you are using a VEXnet Joystick, you will need to also download the firmwares onto that, too. For the Joystick firmware, plug in the USB – USB cable into the underside of the Joystick, where the VEXnet adapter key would go. Just like the Cortex microcontroller, you can easily download the firmware by clicking “Robot → Download Firmware → Automatically Update VEXnet Joystick”

All the firmwares are found through “Robot → Download Firmware”.

In the case where you need to manually load the firmwares, you can select them through “Robot → Download Firmware → Manually Update Firmware”.

For the Cortex Microcontroller:

Select the “Master CPU Firmware” option and then the “CORTEX_V3_20.BIN” file. Then you will need to download the ROBOTC firmware. Select the “ROBOTC Firmware” option and then the “VEX_Cortex_0907.hex”

For the VEXnet Joystick:

Select the “VEXnet Joystick Firmware” option and then the “JOY_V3_20.BIN” file.

For PIC:

PIC does not have an Automatically Update option, so you will need to update them manually. To do this, first select the “Master CPU Firmware” option and then the “VEX_MASTER_V10.bin” file. Then, update the ROBOTC firmware by selecting the “ROBOTC Firmware → VEX_PIC_0907.hex” file.

When you download a programme, the first thing you should do is save it if you haven't already. Next, compile the programme (Robot → Compile Program, or F7). When you download the programme onto the robot, it will automatically compile it, but it is still better to compile it first, to check there are no errors. It is a good idea to regularly compile your code as you build it up to check for errors. Once compiled, download the code onto the microcontroller by Robot → Compile and Download programme or simply F5. If it fails to download, check that the robot is switched on and plugged into the computer with the VEX programming cable (USB – USB for Cortex, USB – Serial for PIC). Also, make sure the programming cable is plugged into the right place on the robot (the Serial port on the far left for PIC, or the USB port on the top of the Cortex, where the VEXnet adapter key goes).

If you are using VEXnet, you can download a programme wirelessly. This can be done by plugging the serial cable into the serial (or “Program”) port on the VEXnet Joystick or VEXnet Upgrade transmitter module (for PIC) instead of into the robot's serial port.

Section D – Making a basic Autonomous

The easiest autonomous to make is a timed autonomous. This is basically like “Go forward for 5 seconds at full speed then turn left for 1 second and go forwards at half speed”.

Motor speeds can vary from +127 to -127. +127 is full speed forwards, 0 is stopped and -127 is full speed backwards.

There is more than one way to write code about time as explained in my next guide, but for now this is how we will write it:

```
wait1Msec(length of time here);
```

or

```
wait10Msec(length of time here);
```

The difference is counting in either milliseconds or 10 milliseconds. The main difference between counting in milliseconds or 10 milliseconds is that the command “wait1Msec” can count up to a maximum of 32.768 seconds, whereas the command “wait10Msec” can count a maximum of 327.68 seconds.

It is important to say what is going to happen after that length of time, for instance:

```
motor[LeftWheels] = 127;//full speed forwards  
motor[RightWheels] = 127;//full speed forwards  
wait1Msec(5000);//five seconds later...
```

Like this, you have not explained what the robot will do after five seconds, so it will continue to go forwards. If you wanted it to stop after five seconds, the correct way to write that would be:

```
motor[LeftWheels] = 127;//full speed forwards  
motor[RightWheels] = 127;//full speed forwards  
wait1Msec(5000);//five seconds later...  
motor[LeftWheels] = 0;//stop  
motor[RightWheels] = 0;//stop
```

Like this, you are telling it to stop moving after five seconds.

So say if we wanted the robot to go forwards for 5 seconds, then turn left for one second, and then go forwards for another 2 and a half seconds, the code would look like this:

```
motor[LeftWheels] = 127;//full speed forwards
motor[RightWheels] = 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);//wait 2 and a half seconds
motor[LeftWheels] = 0;//stop
motor[RightWheels] = 0;//stop
```

You may have realised that throughout this tutorial I have added “//” and then a comment. The “//” defines that the following text on that line is a comment. Comments in your code can be useful to help make sense of things. The different ways to comment are:

```
//single line comment
```

or

```
/*
```

area comment

```
*/
```

The difference is, a single line comment can be used to explain what is happening at the end of a line of code, whereas a multiple line comment can be used to comment out large areas of text, or temporarily disable an area of code.

Comments show up green in ROBOTC, for easy spotting. Here's an example:

```
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
```

Part 2 – Including basic Sensors

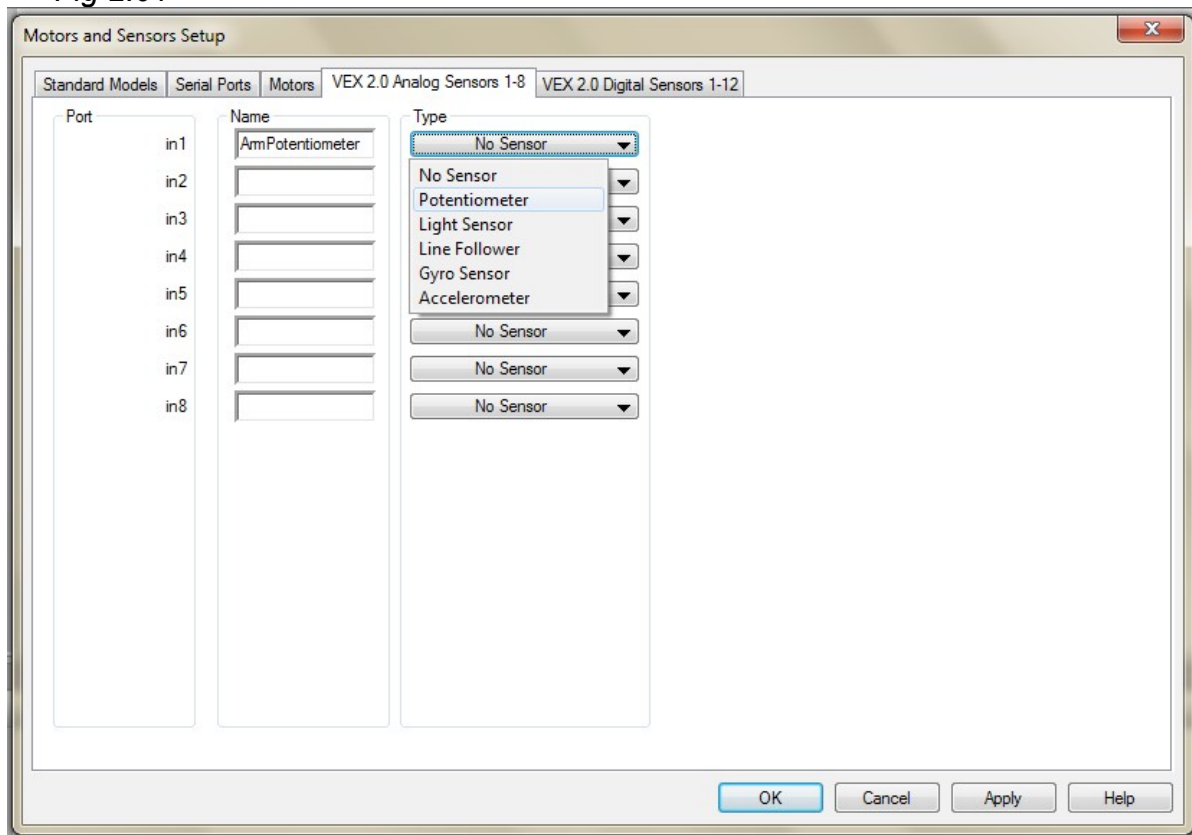
Section A – Introduction to Sensors

Sensors are used for more accurate and precise programming. For instance, instead of saying, go forward for ___ seconds, you can say go forward ___ amount of rotations. For this, we would use an encoder. Instead of saying lift the arm for ___ seconds, we could replace that with lift the arm to a certain height. A good example of why sensors can make your programming more accurate, is that the arm may have already been slightly raised, therefore timing how long to raise the arm would not be accurate and the arm would overshoot.

Generally you would use a potentiometer on an arm, as it is a rotation sensor, like an encoder. The difference between a potentiometer and an encoder is that a potentiometer can't spin numerous times, and unlike an encoder, cannot be reset.

Sensors can also be used to tell when a robot has hit something, distance between it and an object and loads more. Sensors should be setup in the Motors and Sensors Setup window (Robot → Motors and Sensors Setup). When doing this, you will need to define what type of sensor it is, by using the multi – choice drop down bar. I have done an example with a potentiometer. (Fig 2.01)

Fig 2.01



Sensors give a number called a value. Different Sensors give a different range of values.

Section B – Bumper and Limit Switches

The first sensor we will use is a Bumper Switch. They are little buttons generally used for hitting into walls, which is the example we will use them for. Limit switches are programmed exactly the same way, the only major difference is that a limit switch has a thin metal tab that often snaps off. These switches, known as touch sensors, have 2 different values. They are 0, or 1. You will need to set these as touch sensors in the motors and sensors setup.

0 is when they are not pressed and 1 is when they are. For learning how to use touch sensors, we will use a basic robot with 2 motors – Left wheels and right wheels. In this example, (Fig 2.02) the robot will drive forwards and stop if the bumper switch is pressed.

Fig 2.02

```
1 #pragma config(Sensor, in1, bumper, sensorTouch)
2 #pragma config(Motor, port1, leftwheels, tmotorNormal, openLoop)
3 #pragma config(Motor, port2, rightwheels, tmotorNormal, openLoop, reversed)
4 /*!!Code automatically generated by 'ROBOTC' configuration wizard !!*/
5
6 task main()
7 {
8     while(true)//The code needs to be constantly refreshed to work
9     {
10         if(SensorValue(bumper) == 0)//if the bumper is not pressed
11         {
12             motor[leftwheels] = 127;
13             motor[rightwheels] = 127;
14         }
15         else //if the bumper is pressed
16         {
17             motor[leftwheels] = 0;
18             motor[rightwheels] = 0;
19         }
20     }
21 }
```

The “while(true)” loop is used to refresh the if/else control structure, as it won't refresh itself. The while loop needs to be there for the rest of the code to work correctly. Within the while loop, there are two more loops. The first is an “if” loop. If the bumper isn't pressed, the motors will be set to full speed. Otherwise they will be stopped, which the second loop performs. The second loop is the same as “if the bumper is pressed” but since the first loop is if the bumper isn't pressed, the second one can be shortened down to “else”. This is because there are only two options – the bumper is pressed, or its not. If the first statement is true, it will run the first loop. If the statement is false, it will run the “else” loop.

Section C – Potentiometers

As explained in the introduction to sensors, potentiometers are generally used on an arm, where it pivots. Potentiometers are a simple rotation sensor with a range of 0 → 1024. In this section, I will teach you how to use them to move an arm to a certain position. Please note, that the value depends on which way the potentiometer is facing, and where it has been exactly positioned.

A suitable statement to use would be a “while” loop. Let's say we want the arm to be at a sensor value of 650 on the potentiometer, with the higher the value, the higher the arm. This may vary depending on which direction the potentiometer is facing. This is a very basic arm raising code. It should work providing that the arm is already set to a value less than 650. (Fig 2.03)

Fig 2.03

```
1 #pragma config(Sensor, in1, Potentiometer, sensorPotentiometer)
2 #pragma config(Motor, port1, Arm, tmotorNormal, openLoop)
3 /*!!Code automatically generated by 'ROBOTC' configuration wizard !!*/
4
5 task main()
6
7 {
8 while (SensorValue(Potentiometer) < 650);
9 {
10 motor[Arm] = 127;
11 }
12
13 }
```

So this is saying “While the sensor value of the potentiometer is less than 650, raise the arm”. And that's the very basic way of using a potentiometer.

Normally though, gravity effects the arm and the arm comes back down again. To stop this, you could either use rubber bands or something like that to hold the arm up, or “trim”. Trim adds a slight speed to the motor, just enough to hold the arm up. By adding trim to the code above, it would now look like this:

```
1 #pragma config(Sensor, in1, Potentiometer, sensorPotentiometer)
2 #pragma config(Motor, port1, Arm, tmotorNormal, openLoop)
3 /*!!Code automatically generated by 'ROBOTC' configuration wizard !!*/
4
5 task main()
6
7 {
8 while (SensorValue(Potentiometer) < 650);
9 {
10 motor[Arm] = 127;
11 }
12 motor[Arm] = 15; //15 is just enough in this case to hold the arm up
13
14 }
15
```

You can also add trim to the motor when it is being controlled by a transmitter. All you need to do is add “+trim” to the end. e.g. motor[arm] = vexRT(Ch3Xmtr2) +15;

In my next guide, *The Intermediate Guide to ROBOTC*, I will teach how to make a more reliable and efficient way of using the potentiometers, as well as using them in pre-set heights, for easier driving.

Part 3 – Extras

Section A – using the competition template

First, we'll have a look at the competition template. To open a competition template, you can find it in File → New → Competition Template.

First you will see some “pragma” lines and an “include”. Includes are files that can be embedded in another file. I'll explain more about Includes in my next guide. The “Vex_Competition_Includes.c” include is the background code, which is where task main is found. Task main starts the pre_auton function, and autonomous and user_control tasks that are found in the template.

When we use the competition template, we write the autonomous in the “task autonomous()” area, between the curly brackets. The user control code is typed into the “task user_control()” area, inside the while loop. When you compete at a VEX robotics competition, you must use the competition template. Here's a basic example of using the template:

```
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 while(SensorValue(bumperswitch) == 0)
25 {
26     motor[leftwheels] = 127;
27     motor[rightwheels] = 127;
28 }
29
30 motor[leftwheels] = 0;
31 motor[rightwheels] = 0;
32
33 }
34
35
36
37 task usercontrol()
38 {
39     while (true)
40     {
41         motor[leftwheels] = vexRI(Ch3);
42         motor[rightwheels] = vexRI(Ch2);
43     }
44 }
```

Glossary:

Autonomous – the robot driving by itself using pre-programmed instructions. In the VEX robotics competition, there is an autonomous period at the beginning of a match.

Bumper switch – a small button, generally used to sense when a robot has hit a wall.



Comments – notes that can be put into ROBOTC. Comments show up in a green font.

Competition template – the code template which participants at the VEX robotics competition must use.

Drive code – the code used to enable a human operator to drive a robot.

Encoder – a rotation sensor with no physical rotation limits.

Firmware – the programming software downloaded onto the microcontroller.

Function – a body of code that is declared in one place and called up in multiple others.

If/Else Control Structure – a body of code made of 2 loops. An “if” loop, and an else “loop”.

Include file – a file that can be embedded within multiple other files.

Limit switch – a button with an arm.



Microcontroller – a small computer. The robot's “brain”.

PIC Microcontroller V0.5 – a VEX Microcontroller.



Potentiometer – a rotation sensor that can rotate 250°.



Task - a body of code which can be run at same time as another task.

task main – the main task. This is what the robot runs and everything it does is part of task main. You must include this task in your programme if it isn't already included for the programme to work.

Transmitter – a handheld device generally used to drive a robot.

Trim – a number added or subtracted from another, biasing the output on one side. Generally used to maintain an arm height, without gravity pulling the arm back down.

VEXnet – VEX wireless system. VEXnet is used at most VEX robotics competitions.

VEXnet upgrade – an upgrade for the PIC microcontroller.



While loop - a loop that runs the code in the brackets while the statement is true.

I hope that this guide has helped you understand more about ROBOTC. Look out for my next ROBOTC guide, *The Intermediate guide to ROBOTC*.

Thank you to Michael Lawton and everyone else who has helped to make this guide a success.

Please note: The code provided may not have been tested, so is not guaranteed to work.

Copyright © George Gillard 2012

Images of VEX components courtesy of VEX Robotics Inc.