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 program
- 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

Last Edited Feb 14, 2011

Copyright © George Gillard 2011

Introduction to RobotC

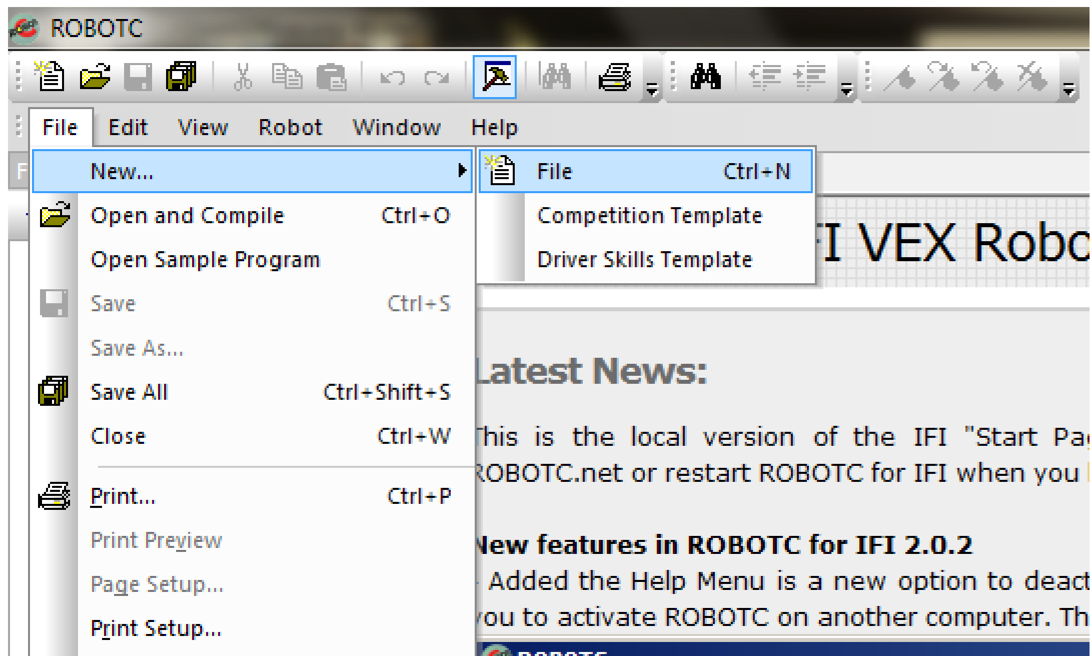
RobotC is an application used for programming VEX robots. There are many different versions of RobotC, but I highly recommend using the latest version, and use the same version across your whole team. In this guide to RobotC, I'm using version 2.02, a VEX Microcontroller V0.5 and a VEX 75MHz Transmitter. Everything taught here applies to the VEXnet upgrade, but not VEXnet Cortex.

Part 1 – The Basic

Section A – Setting up

Once you have opened RobotC for IFI (VEX), 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.

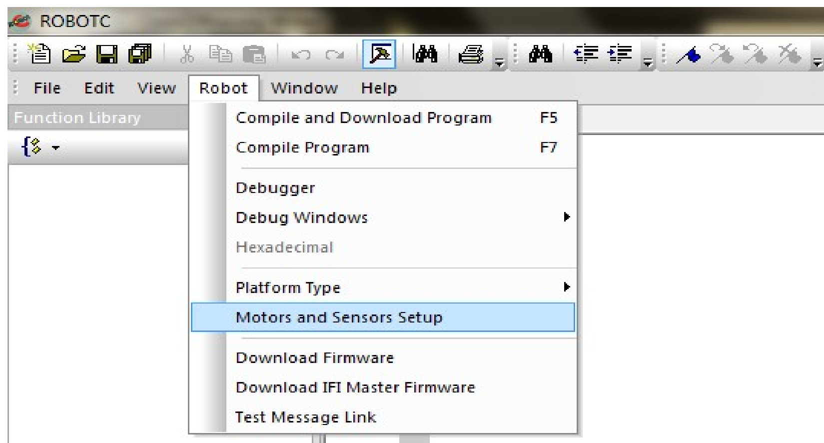
Fig 1.01



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

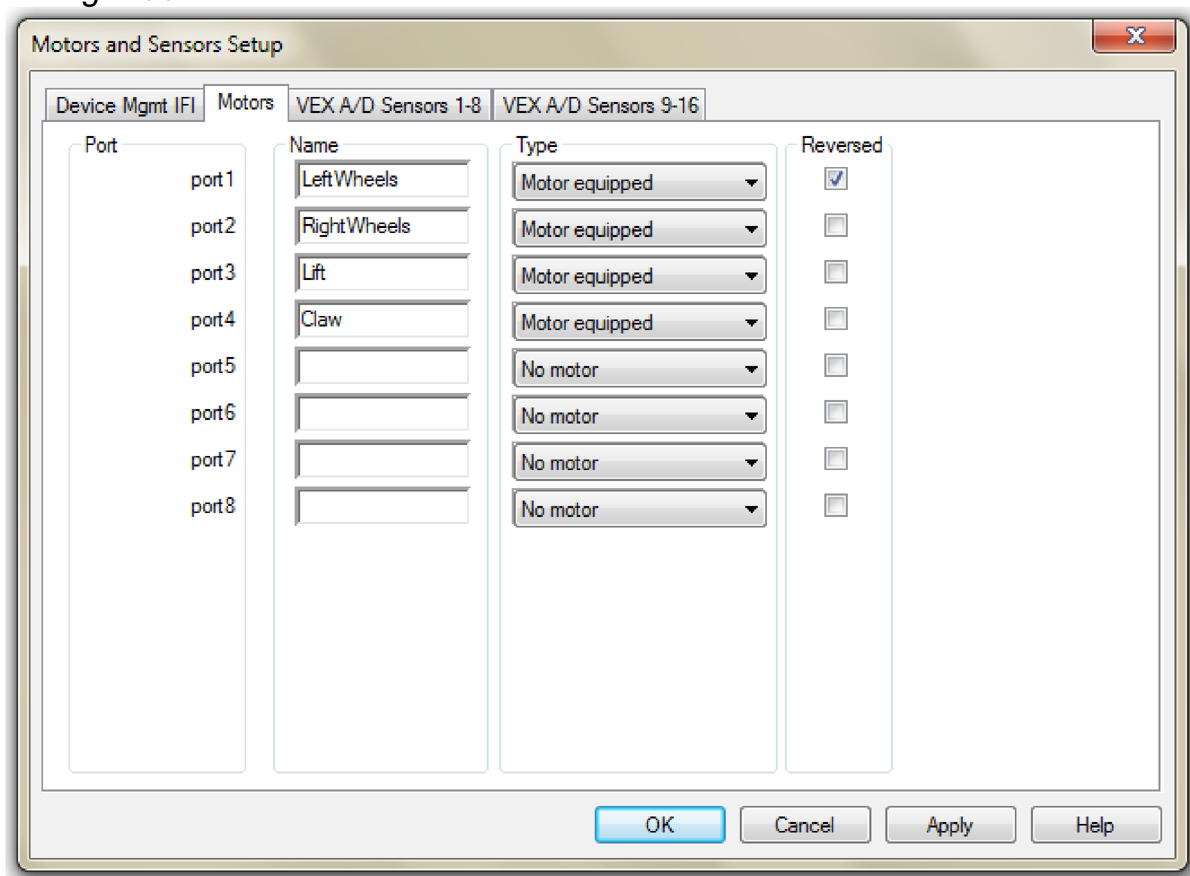
 Copyright © George Gillard 2011

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 wheels motor needs to be reversed.

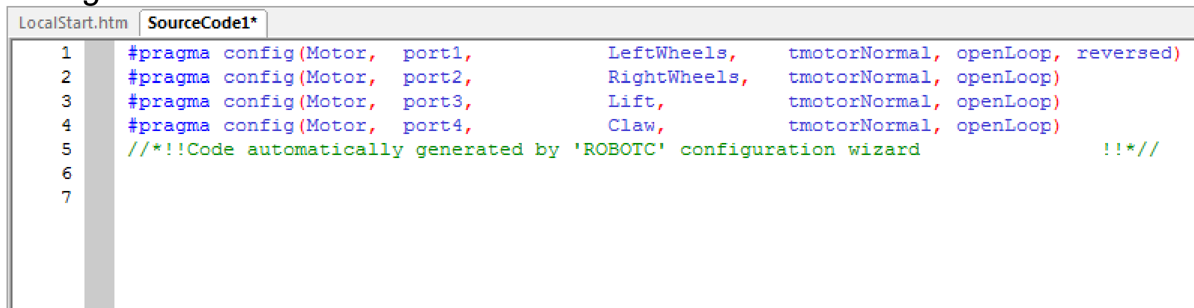
Fig 1.03



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

 Copyright © George Gillard 2011

Fig 1.04



Now is a good time to save your code. To save, click Robot → Compile Program or F7. 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 program. To save your program, click File → Save or Ctrl + S.

Once saved, simply type in your code:

```
task main()
{

}
```

“task main” is the main task. In a competition template, there is a “pre_auton” task, where you reset your encoders, an “autonomous” task, where you write your autonomous and a “usercontrol” task which is where you write the code that allows you to drive. In a task, you always write your code between the curly brackets.

You have now fully set up you new basic code! Task main is the main task, and this is where you will type your drive code. Task main can also be used for testing autonomous.

Section B – Writing a Drive Code

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

 Copyright © George Gillard 2011

When we write about a motor, we write:

motor[*motor name here*]

There are a few exceptions, like when using functions (explained in my next level guide, *The Intermediate Guide to RobotC*).

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

Channel 1 is the sideways one on the right joystick
Channel 2 is the vertical one on the right joystick
Channel 3 is the vertical one on the left joystick
Channel 4 is the sideways one on the left joystick
Channel 5 is the back button on the left (looking from the front)
Channel 6 is the back button on the right (looking from the front)

When we write about a transmitter channel, we write

vexRT(Ch*channel number here*)

So that means that if we want the left wheels on Ch3 (left joystick), the code would be:

motor[LeftWheels] = vexRT(Ch3);

This means that the value that is the output from the transmitter is the motor speed.

Note the semicolon at the end of the line. We use them at the end of each statement (when we tell the program something, e.g. motor[*motor name*] = 127;).

When we add the drive code to the program, we need to tell it that there is no autonomous. If we don't say so, the drive code will not work. Otherwise, if we wanted an autonomous and no drive code, we could just write the autonomous code straight in. To tell the program there is no autonomous, we write:

bIfiAutonomousMode = false;

Otherwise, the program will assume that there is an autonomous task.

 Copyright © George Gillard 2011

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 would be :

```
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  task main()
8  {
9    bIfiAutonomousMode = false;
10
11   while (true)// can also be "while (1 == 1)"
12   {
13     motor[LeftWheels] = vexRT(Ch3);
14     motor[RightWheels] = vexRT(Ch2);
15     motor[Lift] = vexRT(Ch5);
16     motor[Claw] = vexRT(Ch6);
17
18   }
19
20 }
```

When we want to use 2 controllers, instead of

vexRT(Ch*channel number here*)

we use

vexRT(Ch*channel number here*Xmtr2)

The 'Xmtr2' is what tells the program 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 joystick, the claw on the horizontal right joystick), we would write:

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

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

 Copyright © George Gillard 2011

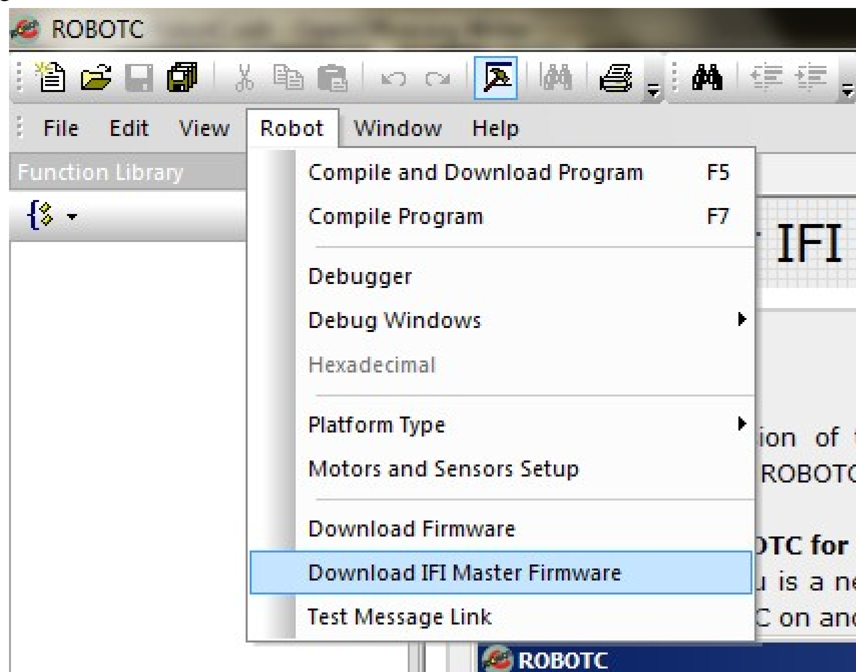
Section C – Downloading a program

When you download a program, the first thing you should do is compile the program. When you download the program 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. This can be done either Robot → Compile Program or simply F7. Once compiled, download the code onto the microcontroller (the robots 'brain') by Robot → Compile and Download Program or simply F5. If it fails to download, check that the robot is switched on and plugged into the computer.

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 Master Firmware. This can be found at Robot → Download IFI Master Firmware. (*Fig 1.05*)

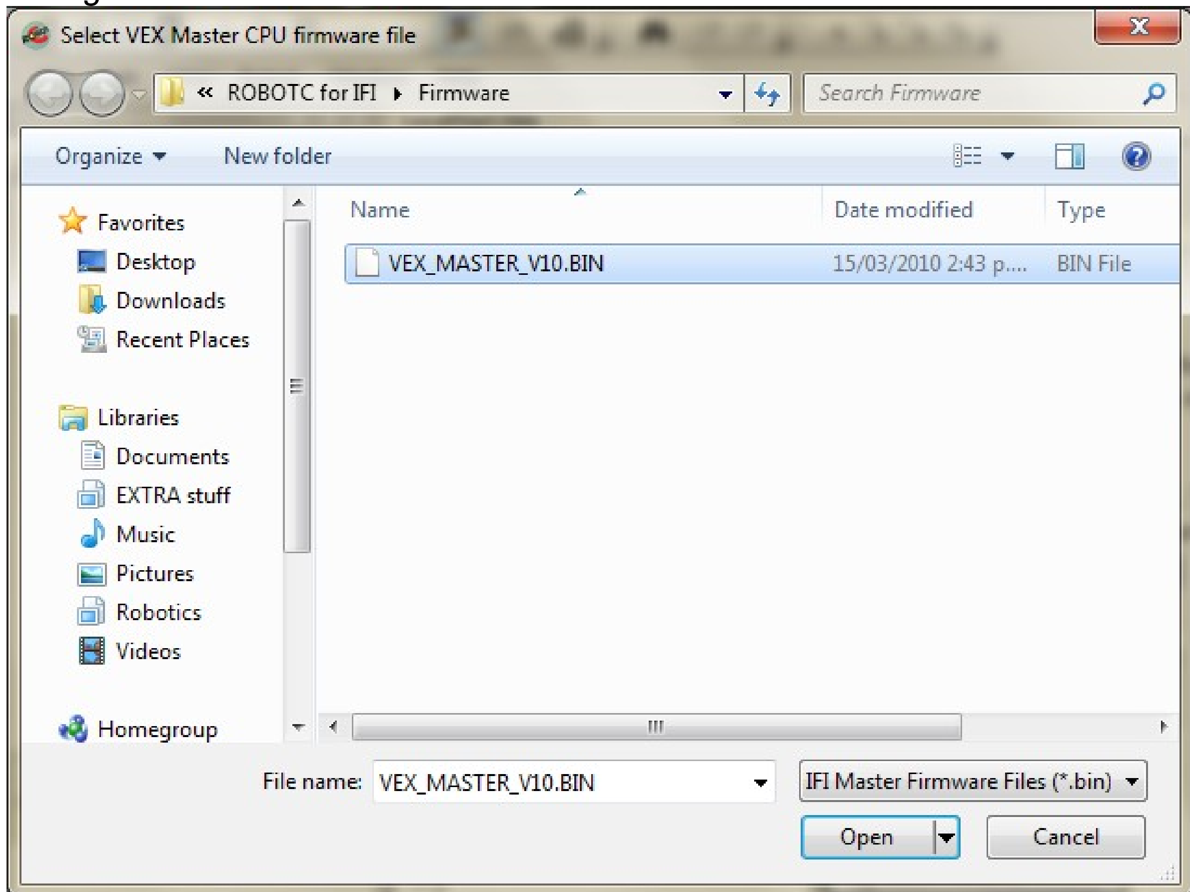
A pop out box will show. Select the “VEX_MASTER_V10.BIN” file and click open. (*Fig 1.06*)

Fig 1.05



 Copyright © George Gillard 2011

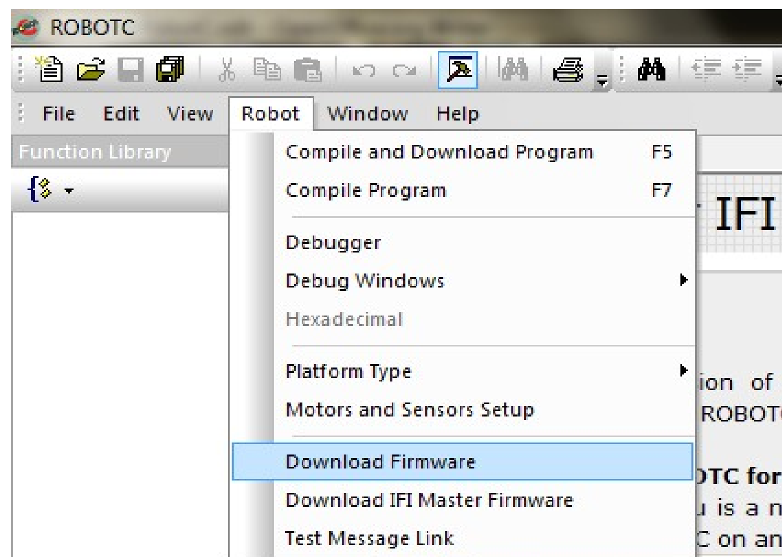
Fig 1.06



Once downloaded, you will need to download the normal firmware. This is found at Robot → Download Firmware. (*Fig 1.07*)

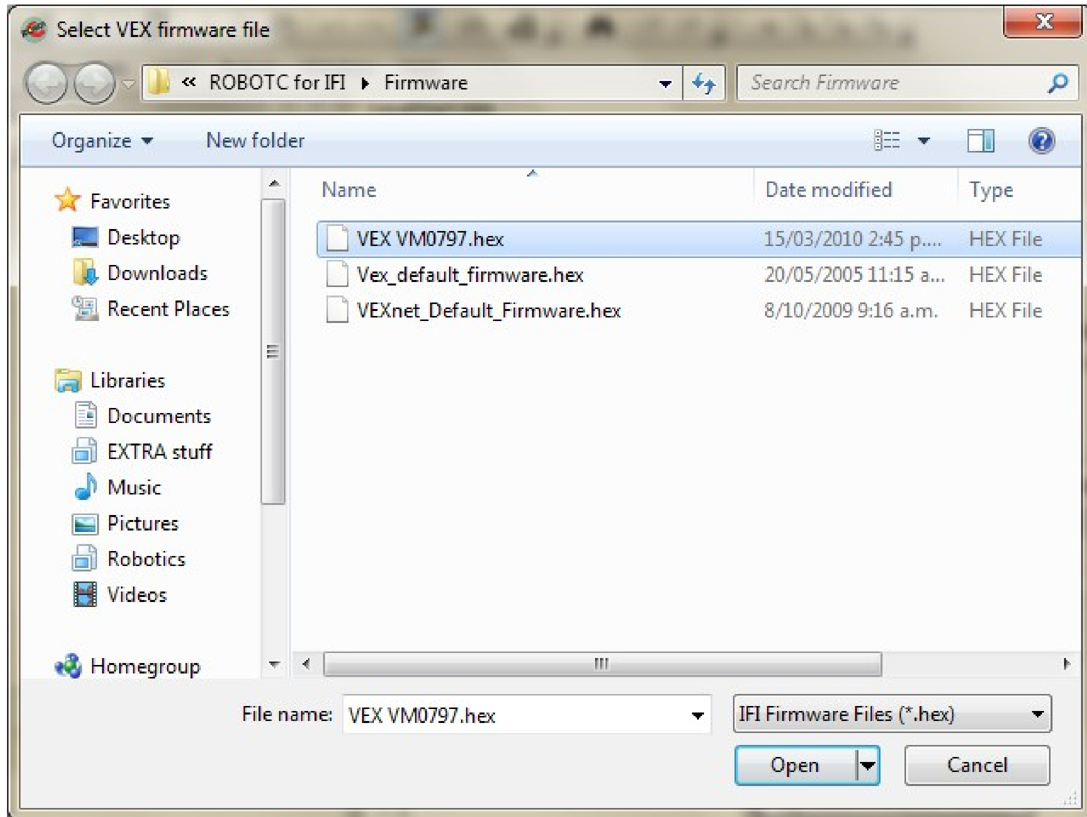
A pop up box will show. Select the "VEX VM0797.hex" file and click open. (*Fig 1.08*)

Fig 1.07



 Copyright © George Gillard 2011

Fig 1.08



If you are using the Vexnet upgrade, you can download a program wirelessly. This can be done by plugging the serial cable into the serial port on the transmitter instead of into the robot's serial port. However, you should not wirelessly download firmware, so you will need to plug into the robot for that.

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 and go forwards at half speed”.

We write

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 10milliseconds is that the

 Copyright © George Gillard 2011

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, 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 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
```

 Copyright © George Gillard 2011

or

/*

unlimited 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 an unlimited area comment can be used to comment out large areas of text, or disable an area of code.

Comments show up green in RobotC,for easy spotting. Heres an example:

```
1  motor[LeftWheels] = 127;//full speed forwards
2  motor[RightWheels] = 127;//full speed forwards
3  wait1Msec(5000);//five seconds later...
4  motor[LeftWheels] = -127;//full speed backwards
5  motor[RightWheels] = 127;//full speed forwards
6  wait1Msec(1000);//the value of how long to spin may vary as to how fast it is
7  motor[LeftWheels] = 127;//full speed forwards)
8  motor[RightWheels] = 127;//full speed forwards)
9  wait1Msec(2500);//wait 2 and a half seconds
10 motor[LeftWheels] = 0;//stop
11 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.

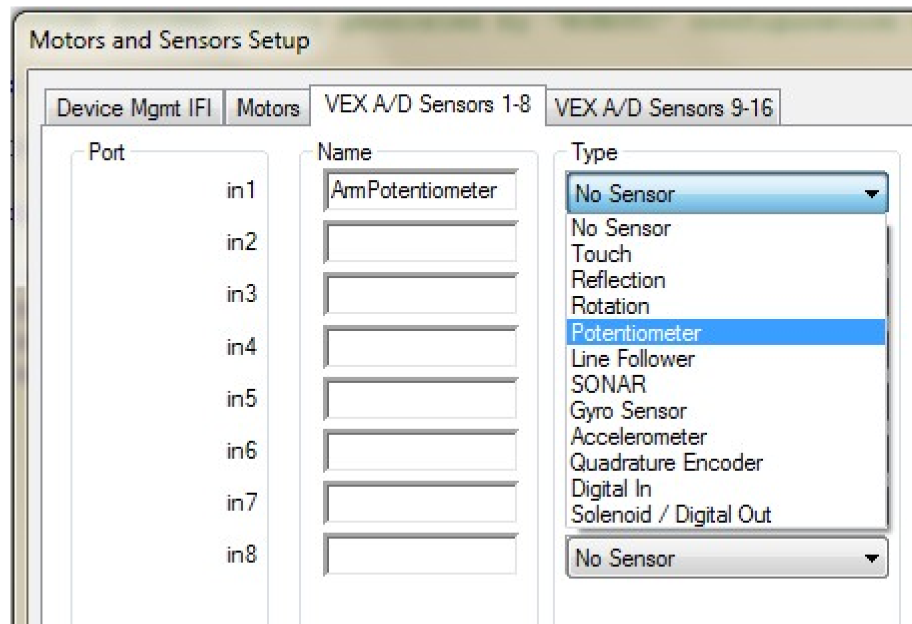
Generally you would use a potentiometer on an arm, as it is a rotation sensor, like an encoder. The difference between a potentiometer and an encoder is that a potentiometer can't spin numerous times, and unlike an encoder, cannot be reset.

Sensors can also be used to tell when a robot has hit something, distance between it and an object and movement. Sensors should be setup in the motors and setup window (Robot → Motors and Sensors Setup) for full performance. When doing this, you will need to define what type of sensor it

 Copyright © George Gillard 2011

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 bumpers 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 – 0, or 1.

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 will drive forwards, and once the bumper switch has been pressed, will stop.

Copyright © George Gillard 2011

Fig 2.02

```
1  #pragma config(Sensor, in1,    BumperSwitch,        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  {
9   motor[LeftWheels] = 127;
10  motor[RightWheels] = 127;
11  if(SensorValue(BumperSwitch) == 1);
12  {
13    motor[LeftWheels] = 0;
14    motor[RightWheels] = 0;
15  }
16 }
```

Note that I have used the statement: *if (SensorValue(BumperSwitch) == 0);*.This basically means what it says; *if the sensor value of the bumper switch is 1.* This "if" loop is often used in conjunction with sensors. For example: "*If the arm is __ high"*, or "*If the wheels have turned __ many times"*. There are different ways of rewriting the code above, this is just a basic way. Here's another way, using a "while" loop:

```
1  #pragma config(Sensor, in1,    BumperSwitch,        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  {
9    while (SensorValue(BumperSwitch) == 0);
10   {
11     motor[LeftWheels] = 127;
12     motor[RightWheels] = 127;
13   }
14   //next bit of code goes here
15 }
```

So this is basically saying *while the sensor value of the bumper switch is 0, go forwards*. This will do exactly the same thing, it has just been written differently. That is all there is to programming bumper switches basically! In my next guide, *The Intermediate Guide to RobotC*, I will teach how to sync multiple bumper switches together, for an even more reliable program.

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.

Copyright © George Gillard 2011

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. 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.

End of The Beginners Guide to RobotC, By George Gillard

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

Copyright © George Gillard 2011