**- Green Army Robotics - http://greenarmyrobotics.org -**

**ROBOTC’s JoystickDriver.c**

Posted By greenarmy On October 18, 2011 @ 7:46 pm In ROBOTC | [Comments Disabled](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/print/#comments_controls)

**What is it?**

JoystickDriver.c is the defacto way to interact with the official FTC joysticks as well as the Field Control System (FCS).  Your code needs it to see not only what buttons are being pressed on the joystick, but whether the field is in Autonomous/Driver mode and when the match starts.

**Including the driver**

If you’re using the FTC competition templates which come with ROBOTC, you should already have the driver included for you:

#include "JoystickDriver.c"  //Include file to "handle" the Bluetooth messages.

You can ignore the comment about the include handling Bluetooth messages as it works over USB or WiFi (Samantha) & the template comment is just out of date.

If you’re not using the template (because you’re testing or playing around), just add the same *#include “JoystickDriver.c“* include line somewhere below your pragma statements & above any variable or function declarations.

*\*\* Make sure you include JoystickDriver.c in your autonomous program files as well or they will not be able to communicate with the field!*

**Calling getJoystickSettings()**

If you want to read the current state of the joystick, you have to first call this function.  In general, this is not needed for your autonomous programs, but is absolutely essential for tele-op programs.

The syntax is:

getJoystickSettings(joystick);

What this function does is copy the most recent state of the joystick buttons, analog sticks, & field flags from a message buffer to a local joystick structure which is what your code looks at when you check to see if a particular button is pressed or what the y-value is of an analog stick.  Remember that the robot is constantly receiving messages from the field (something like every 50ms), but your code is only looking at a snapshot of one of those messages at a time.  This function updates that snapshot with the contents of the most recent message buffered.

The ideal place for this critical line of code is at the top of your infinite while loop inside of your task main.  This ensures that you have the most recent state at the beginning of each loop through your drive code.  Below is an example of where you should place the call:

task main()

{

  initializeRobot();

waitForStart();   // wait for start of tele-op phase

while (true)

  {

 getJoystickSettings(joystick);  //Get current joystick state

   //The rest of your main loop code here

}

}

Note that the waitForStart() function internally calls getJoystickSettings() so that it knows as soon as the match begins.

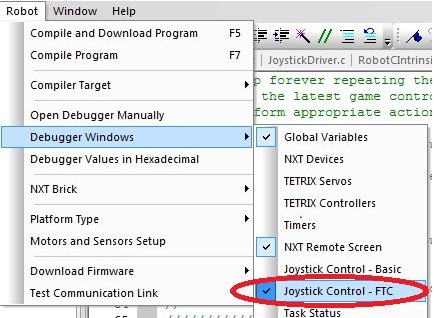
There is also some confusion about whether this was needed in years past.  There may have been something calling it for you (possibly in the displayDiagnostics task) , but as a best practice you should always make the call in your code yourself.

**Using the joystick**

To test with the joysticks inside of ROBOTC, you must have the joystick debug window open. Note that there are two different joystick debug windows to chose from:

* **Joystick Control – Basic** – Choose this one if you’re using the PC-based Emulator compiler target, otherwise you should stick with the FTC version.
* **Joystick Control – FTC** – This should be your default when using ROBOTC for FTC.  It has autonomous/tele-op as well as Ready/Running buttons that are needed when your tele-op code is using the official template.

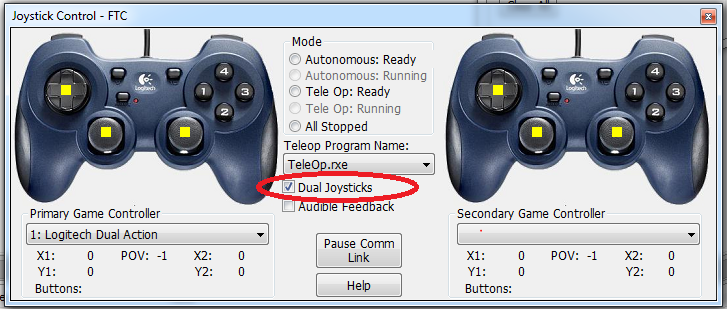
You can open the FTC joystick debug window by selecting **Robot**, **Debugger Windows**, **Joystick Control – FTC**:



FTC allows you the choice of using one or two joysticks.  As such, it is important to always be conscious of which joystick you are looking at.  Even if you are only using a single joystick to drive your robot, you’ll need to specifically look at one of the two.

In the following sections, you’ll notice that every function is specific to one of the two joysticks.

To actually use the second joystick in the ROBOTC debugger, you’ll need to make sure you are using the FTC joystick debug window (not the standard one) and that the Dual Joysticks option  is checked.  
Once the joystick window is open, you can select the Dual Joysticks option and then use the drop down to pick which physical joystick attached via USB to use as the second one:

[](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/attachment/dual_joysticks)

Also note that you cannot use two joysticks if the compiler target is set to PC-Based Emulator or Virtual Worlds; it only works for NXT-Tetrix with the compiler target set to Physical Robot.

**Using the Analog Sticks**

Each joystick has two analog sticks, each with a X (left/right) and Y (UP/DOWN) axis.  The following functions allow you to read the value of each:

|  |  |  |  |
| --- | --- | --- | --- |
| **Analog Stick** | **Axis** | **Joystick 1** | **Joystick 2** |
| Left | X | joy1\_x1 | joy2\_x1 |
| Y | joy1\_y1 | joy2\_y1 |
| Right | X | joy1\_x2 | joy2\_x2 |
| Y | joy1\_y2 | joy2\_y2 |

[](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/attachment/analog_sticks)

All eight functions above will return values between -128 and 127.

Below is an example of code which reads the y-values of each analog stick on the 1st joystick, passes them to a function which scales them to -100 to +100 for the motors, and then passes that result to the driver motors:

//tank-drive code

motor[leftDriveMotor] = scaleForMotor(joystick.joy1\_y1);

motor[rightDriveMotor] = scaleForMotor(joystick.joy1\_y2);

**Using the Buttons**

To access the state of a given button, you can call the Joy1Btn() function for the first joystick or Joy2Btn() function for the second and pass the number of the button you want the state of.  These functions will return a boolean with true indicating that the button is currently pressed or false if it is not.  There are 12 buttons on the Logitech joysticks used in FTC.

|  |  |
| --- | --- |
| **Button** | **Location** |
| 1 | Left-most on the front face (Labeled X on newer joysticks) |
| 2 | Bottom-most on the front face (Labeled A on newer joysticks) |
| 3 | Top-most on the front face (Labeled Y on newer joysticks) |
| 4 | Right-most on the front face (Labeled B on newer joysticks) |
| 5 | Top-left shoulder button  (Labeled LB on newer joysticks) |
| 6 | Top-right shoulder button  (Labeled RB on newer joysticks) |
| 7 | Bottom-left shoulder/trigger button  (Labeled LT on newer joysticks) |
| 8 | Bottom-right shoulder/trigger button (Labeled RT on newer joysticks) |
| 9 | Back button |
| 10 | Start button |
| 11 | Left analog thumbstick (when pressed in) |
| 12 | Right analog thumbstick (when pressed in) |

Below is an example where we turn on a motor if button 2 on joystick 1 is pressed and turn it off if the button is not pressed:

if(joy1Btn(2)) //check if btn 2 is pressed

{

motor[exampleMotor] = 100; //turn motor on

}

else //btn 2 not pressed

{

motor[exampleMotor] = 0; //turn motor off

}

**Using the Hat Switch**

The hat switch, also called a d-pad or POV (point-of-view) switch has 8 directions it can be pressed in.  In ROBOTC the hat switch has nine unique values, 0-7 for the different directions the hat can be pressed in and -1 if it is not pressed at all.  The directional values start with 0 at the top and go clockwise through 7:

[](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/attachment/pov_values)

You get the values from the joy1\_TopHat and joy2\_TopHat variables on the joystick struct in JoystickDriver.c.

Example of running a motor forward when hat is pressed up, reverse when hat is pressed down, or stopping otherwise:

if(joystick.joy1\_TopHat == 0)

{

motor[exampleMotor] = 50; //forward

nxtDisplayTextLine(0, "forward");

}

else if(joystick.joy1\_TopHat == 4)

{

motor[exampleMotor] = -50; //reverse

nxtDisplayTextLine(0, "reverse");

}

else

{

motor[exampleMotor] = 0; //off

nxtDisplayTextLine(0, "stop");

}

**\*** Notice that you need to access the hat switch value through the joystick variable.  You cannot simply look at “joy1\_TopHat”; you have to look at “**joystick.**joy1\_TopHat”.

**New connection drop feature**

With the switch to Samantha last season, the issue of  comunication with the robot getting lost during tele-op became more serious.  At many events, if the field communication to the robots was lost (due to a field fault, a robot losing power to their samantha, a robot’s usb cable between the samantha and their NXT coming loose, etc.), the robots would continue reading the last good joystick value as the most recent joystick value.  This means that if you were holding down a button which made your robot driver forward @ 50% power when the communication was lost, the robot would never know if you released the button and would continue driving forward until it the field communication was re-established or the robot physically disabled.  Obviously this resulted in a lot of burnt out motors, broken parts, and some safety issues. http://greenarmyrobotics.org/wp-content/uploads/2011/10/flail.gif

Enter the new connection drop feature!

The idea is fairly simple.  Inside of JoystickDriver.c, the task which parses new packets from the message buffer and updates the joystick state now looks at whether or not any good messages are received and increments a counter if any are not.  If a good message is received it resets the counter, otherwise if the counter reaches a set limit then the driver will update the joystick state to a default state.  The limit is currently set to 750 and works out to about 3 seconds.  The default state has no buttons pressed and the analog sticks at zero.  In theory this should mimic a driver letting go of the joystick which should stop all motors in a typical FTC tele-op program.

This should work for 90% of the teams out there, but in case you’re one of that 10% who have code which waits for a non-default action to disable a motor (such as a button being pressed to stop spinning a sweeper), then you may need to take some extra action in your code.  The [ROBOTC devs have provided us with three variables](http://www.robotc.net/forums/viewtopic.php?p=14750#p14750) which you can use however works best for you to handle any fringe cases:

**\*bDisconnected** = A flag that is triggered as soon as the nNoMessageCounterLimit is reached (remember messages check on a 4ms interval) to indicate that it has been longer than the specified time since a message has arrived.

**\*bOverrideJoystickDisabling** = A user-settable flag that will still cause the “bDisconnected” flag to be triggered, but will not call the “zero-ing” code to clear out the last joystick packet. This flag can be modified at any time.

**\*nNoMessageCounterLimi**t = A long integer to specify the threshold for “bDisconnected” to be triggered. By default this is set to 750 counts (3 seconds), but you can modify it as you see fit.

**Tips**

**Naming your buttons**

Most teams end up defining a control scheme where most of the buttons on the joystick get dedicated to a particular function.  It is much more meaningful to refer to a button which opens a claw as the “claw open button” than “button #2″ or “the Y button”.

You can easily define constants for each button you use and then refer to these when coding against the buttons rather than against the button numbers.  This is all about code readability and programmer sanity.  As a bonus, if you need to change which button does what, it is much easier to change the mapping in the constant than hunt for button numbers sprinkled throughout your code.

Which of these is easier to read & understand?

[](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/attachment/btn_names)

**Display diagnostics control**

By now you may have noticed that when you’re using JoystickDriver.c, the NXT will by default display some diagnostic info about whether it is waiting for start, in autonomous or tele-op, your battery voltages, and the name of your tele-op file.  If you want to disable it for whatever reason (possibly to display other diagnostics or debug info yourself on the same lines), you can simply set the bDisplayDiagnostics flag to false.  There is even a disableDiagnosticsDisplay() function you can call to do this.

*\*Note that you’ll want to have diagnostic info enabled for competition!*

Article printed from Green Army Robotics: **http://greenarmyrobotics.org**

URL to article: **http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c**

Click [here](http://greenarmyrobotics.org/robotc/robotcs-joystickdriver-c/print/#Print) to print.

Copyright © 2011 Green Army Robotics. All rights reserved.