# ROBOTC for MINDSTORMS

## 1. Introduction

**Created by: Timothy Friez, Dick Swan, Bence Feher & Jesse Flot**

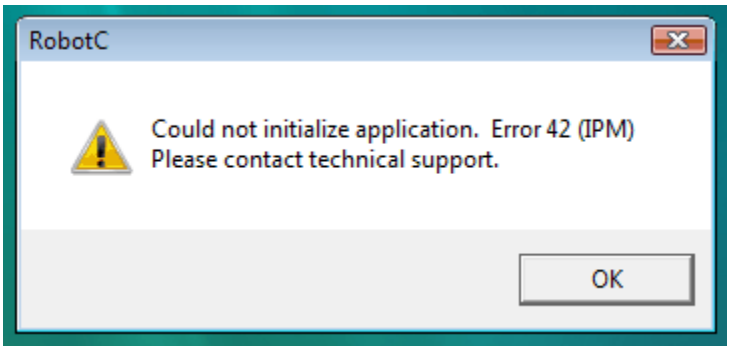


# 2. Installation Help

## 2.1 Error 42 IPM Issues

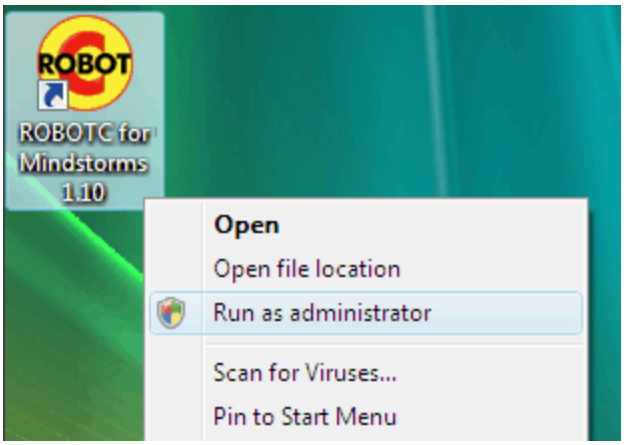
### ROBOTC Troubleshooting:

"**Error 42 (IPM)**" when launching ROBOTC after installing ROBOTC

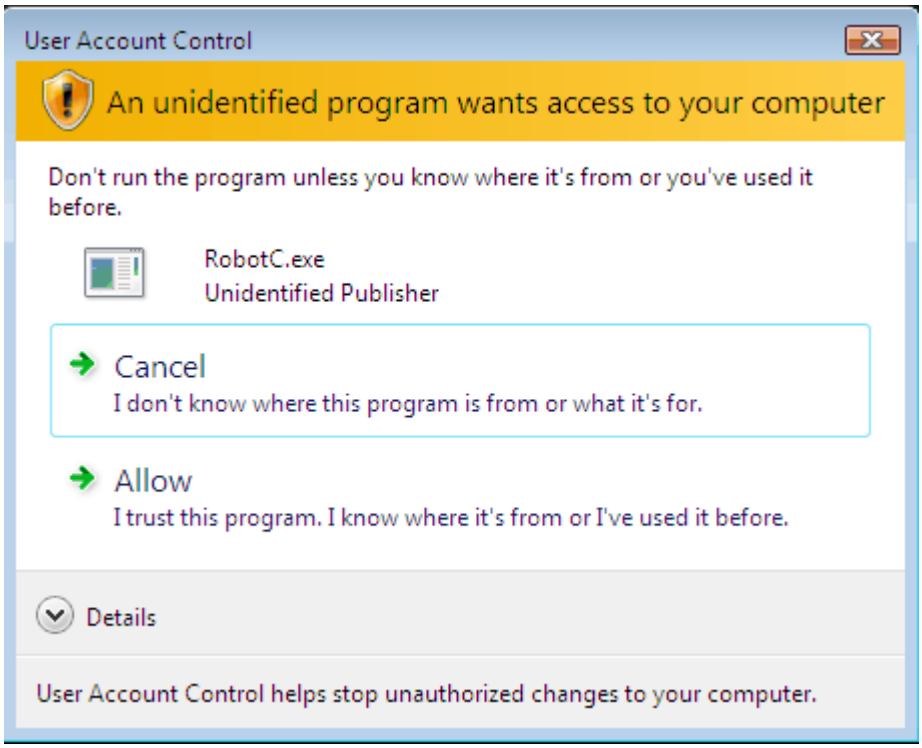



**Solution:**
This error is caused when Windows XP or Windows VISTA does not have the appropriate rights to initialize the activation software. To fix this issue, you will need to run the application "As an Administrator." Once you have followed these steps once, you can use ROBOTC as a normal user (without Administrative Privileges) without issue.

***Windows VISTA***



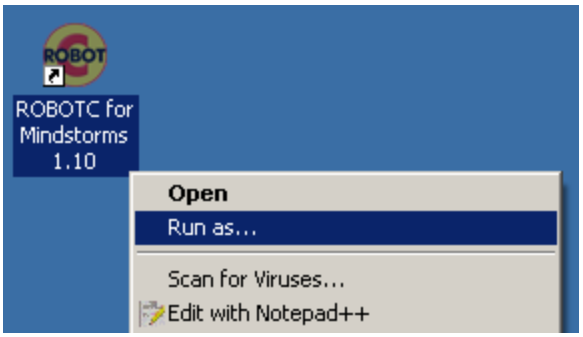
Step 1: Right Click on the ROBOTC icon. Select "Run as Administrator"



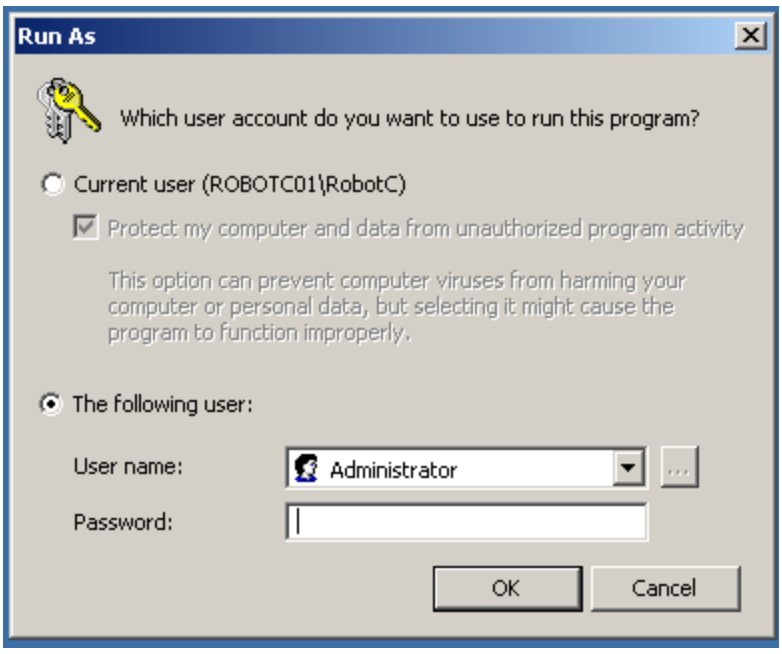
Step 2: Windows VISTA will prompt you to give Administrative Access to this application. Select "Allow" to continue.

ROBOTC should now open to the Licensing Screen. You will not have to repeat these steps again to use ROBOTC in the future.

***Windows XP***



Step 1: Right Click on the ROBOTC icon. Select "Run as..."



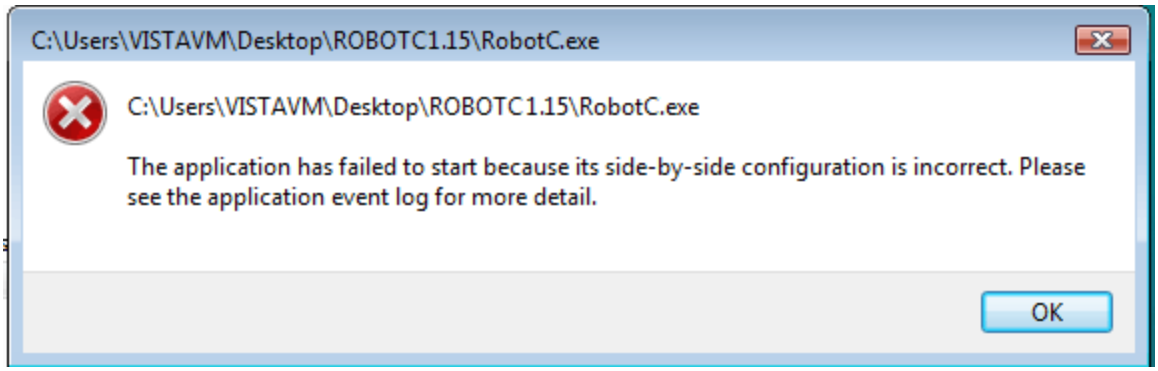
Step 2. At the "Run As" dialog, select the "Following User" dialog and enter the user name and password of a user with administrative rights.

ROBOTC should now open to the Licensing Screen. You will not have to repeat these steps again to use ROBOTC in the future.

# 2.2 Side-By-Side Error

### ROBOTC Troubleshooting:

**"Application Failed to Start: Side-by-Side" Error** - ROBOTC will not open




**Solution:**
If you have installed ROBOTC and tried to move the ROBOTC folder to another computer, you will recieve this error. This error is caused because the runtime engines ROBOTC is based off of need to be installed along with the application. If the runtime engines are not installed, you will recieve a "Side-by-Side configuration is incorrect" error.

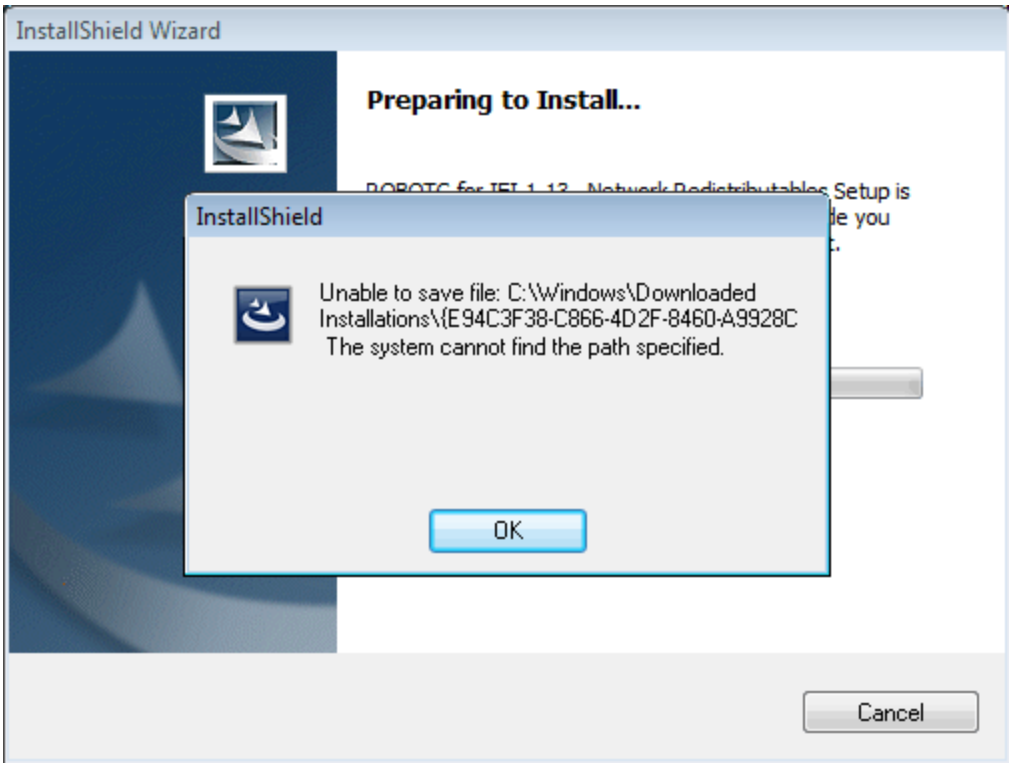
This issue will also be caused if you are using ROBOTC from a network location without running the "Network Dependencies" on the computer attempting to use ROBOTC.

To fix this issue, you can do two things:
1. Re-install the latest version of ROBOTC
2. Install the Network Dependencies pack which is found from the download section of the ROBOTC.net website.

# 2.3 Installer "Unable To Save File" Issue

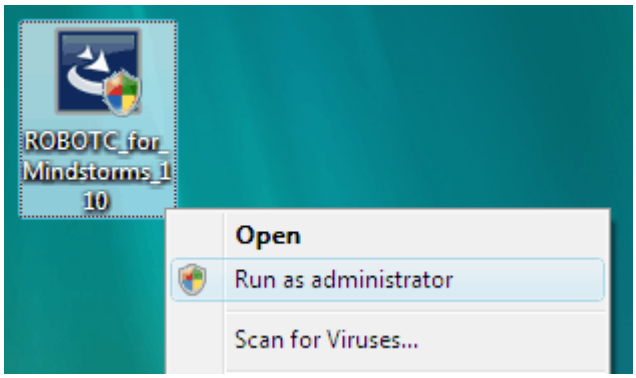
### ROBOTC Troubleshooting:

**"Unable to Save File" Error** - ROBOTC Will Not Install

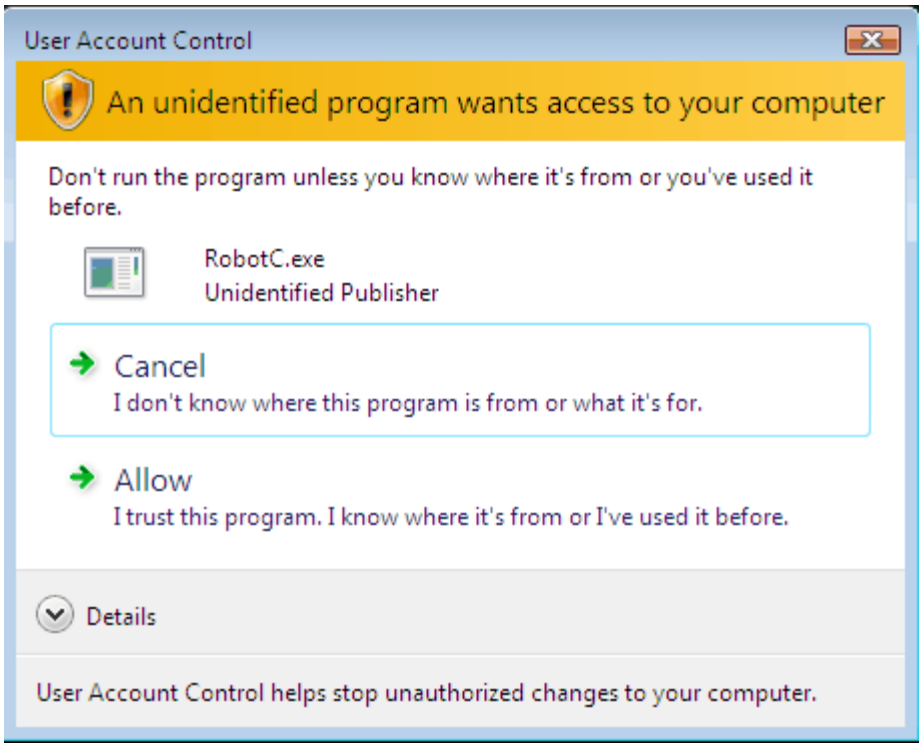



**Solution:**
If the ROBOTC installer is "Unable to save file", this error means that ROBOTC does not have permission to write to a temporary installation directory inside of the Windows folder. The installer needs to opened with Administrator privileges.

***Windows VISTA***



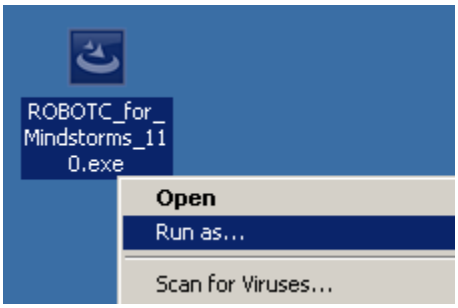

Step 1: Right Click on the Installer icon. Select "Run as adminisrator."



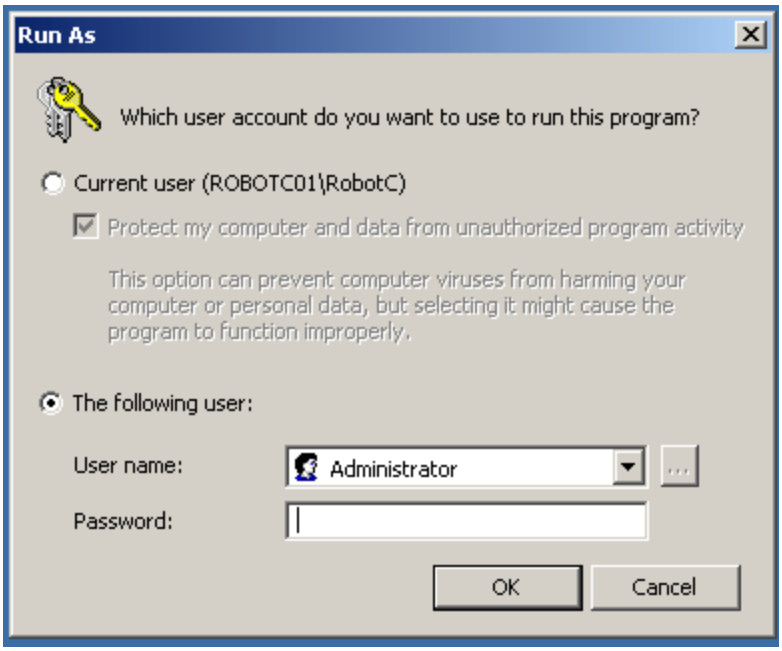

Step 2: Windows VISTA will prompt you to give Administrative Access to this application. Select "Allow" to continue.

Step 3: Follow the directions on the installer. ROBOTC should install successfully now.

***Windows XP***




Step 1: Right Click on the Installer icon. Select "Run as..."



Step 2. At the "Run As" dialog, select the "Following User" dialog and enter the user name and password of a user with administrative rights.

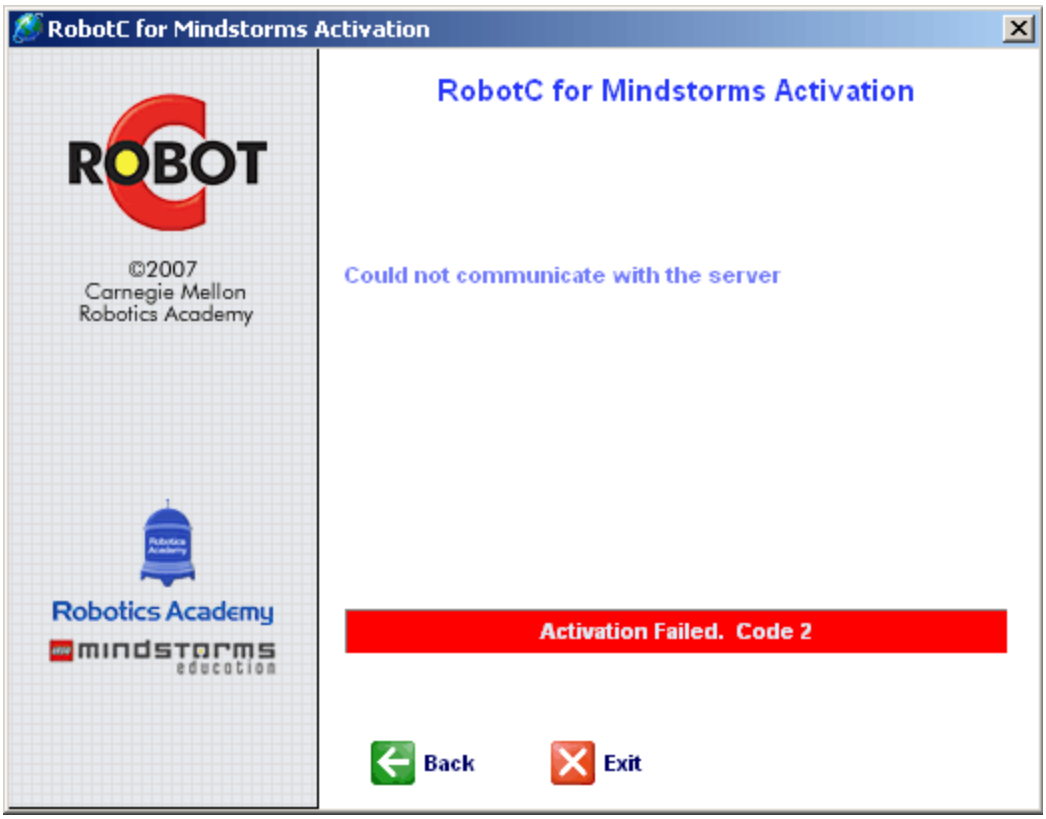
Step 3: Follow the directions on the installer. ROBOTC should install successfully now.

# 2.4 Other Activation Issues

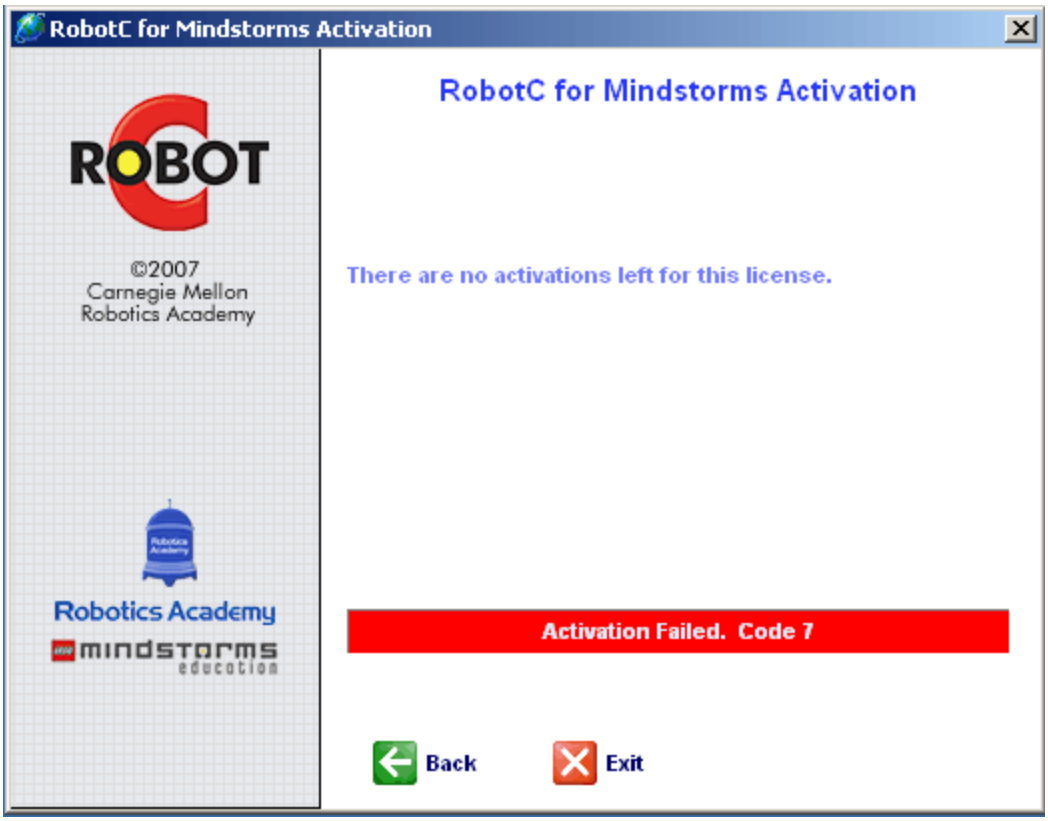
## ROBOTC Troubleshooting:

### Common Activation Errors

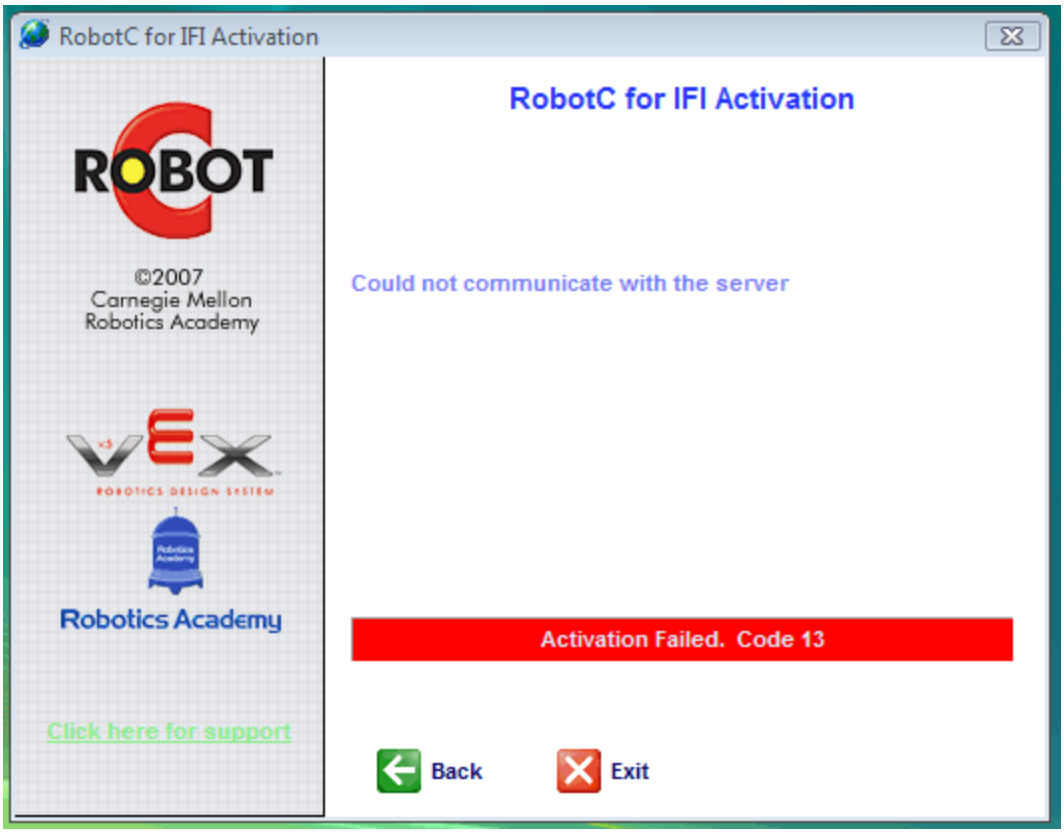
**Error 2** - Could Not Communicate With License Server
**Error 7** - No Activations Left for this License
**Error 13** - Firewall or Trouble Communicating with License Server
**Error 69** - Invalid Registration Code
**Error 100** - Invalid License ID or Password
All Other Errors



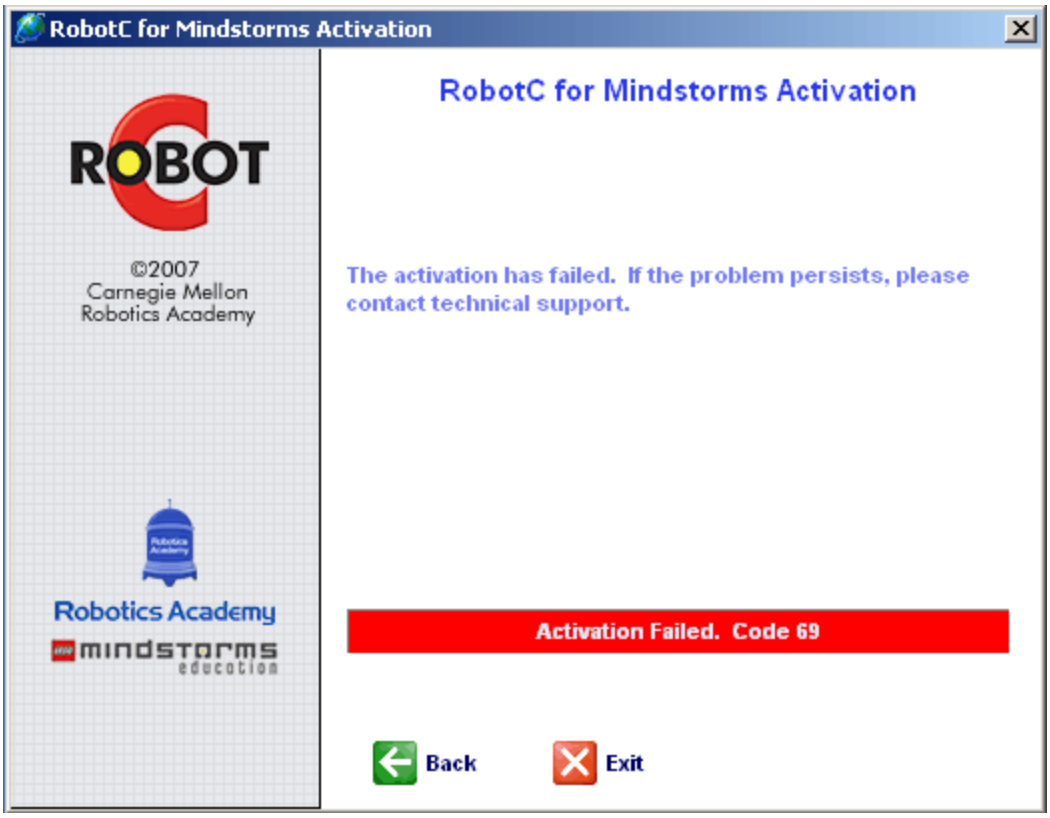

**Error #2**
**Cause:** Could Not Communicate With License Server
**Solution:** Connect to the internet to the computer you are trying to activate, or follow these instructions on using the "Activate by Web."



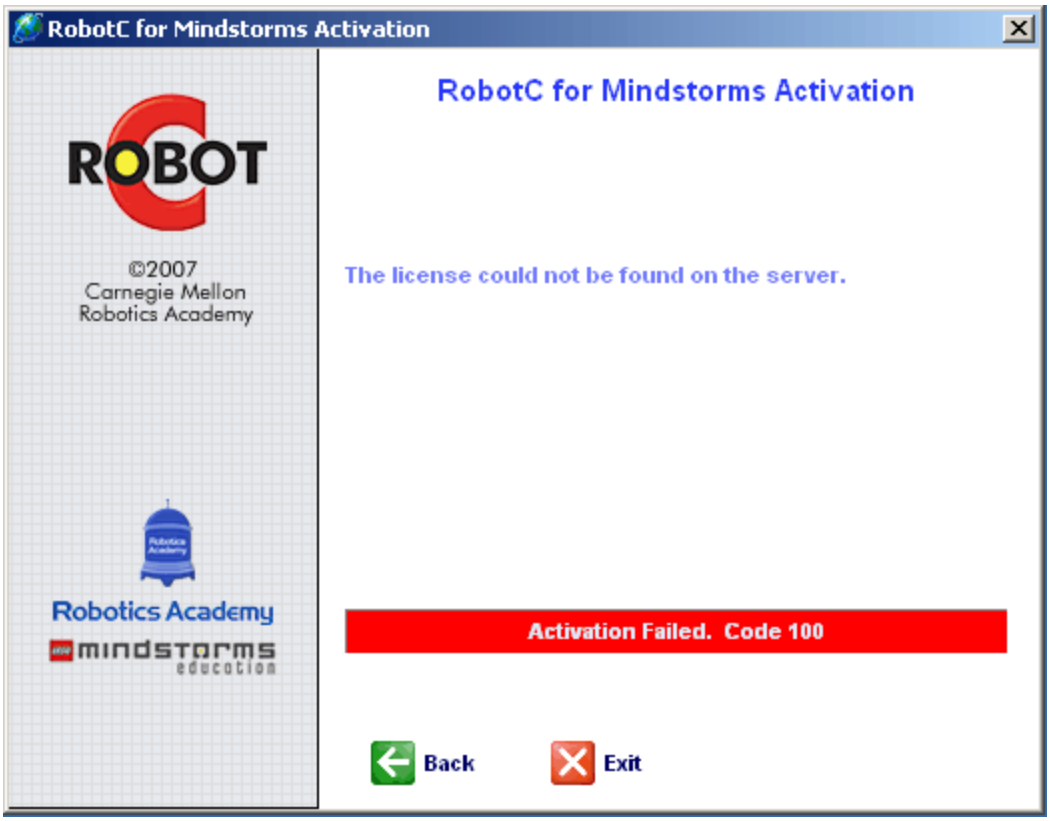

**Error #7**
**Cause:** The License ID you are trying to activate with has been activated on too many computers.
**Solution:** Contact ROBOTC Support at support@robotc.net




**Error #13**
**Cause:** The license password is invalid or could not communicate with server.
**Solution:** Check the LicenseID and Password. Connect to the internet to the computer you are trying to activate, or follow these instructions on using the "Activate by Web."




**Error #69**
**Cause:** Invalid "Web Activation" Registration Code.
**Solution:** Make sure you have typed the generated code correctly, or follow the instructions on using the "Activate by Web" again.




**Error #100**
**Cause:** Invalid "Web Activation" Registration Code.
**Solution:** Make sure you have typed the generated code correctly, or follow the instructions on using the "Activate by Web" again.

**All Other Errors:** Contact support@robotc.net

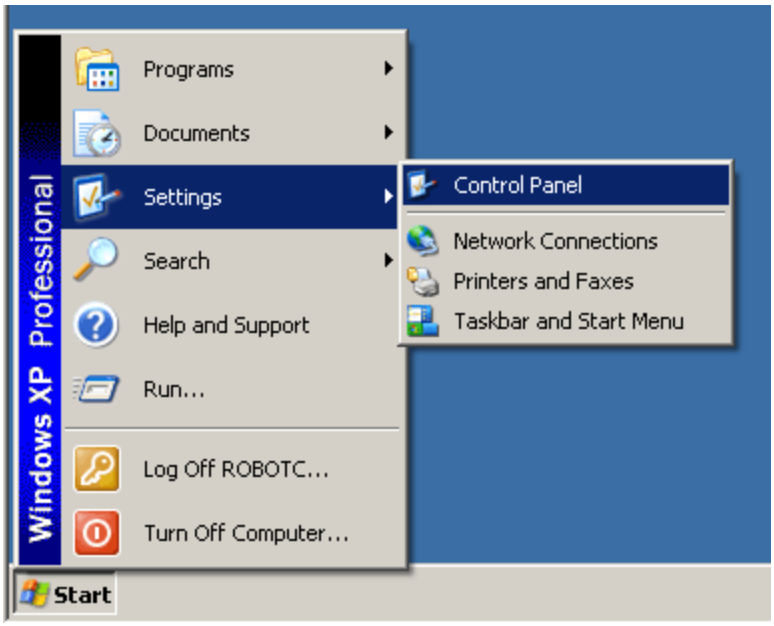
# 3. Getting Started

## 3.1 Uninstalling and Installing ROBOTC

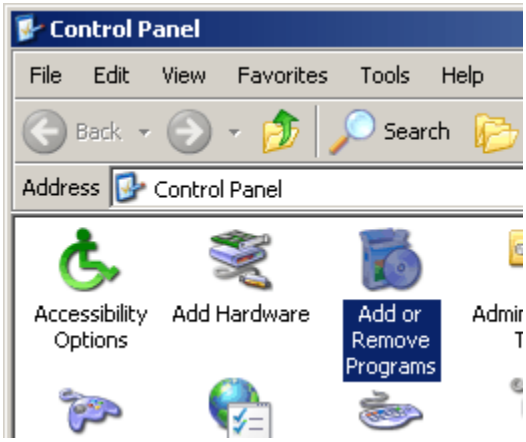
You must uninstall your previous version of ROBOTC before upgrading to the newest version of ROBOTC.

To uninstall ROBOTC, follow these steps:

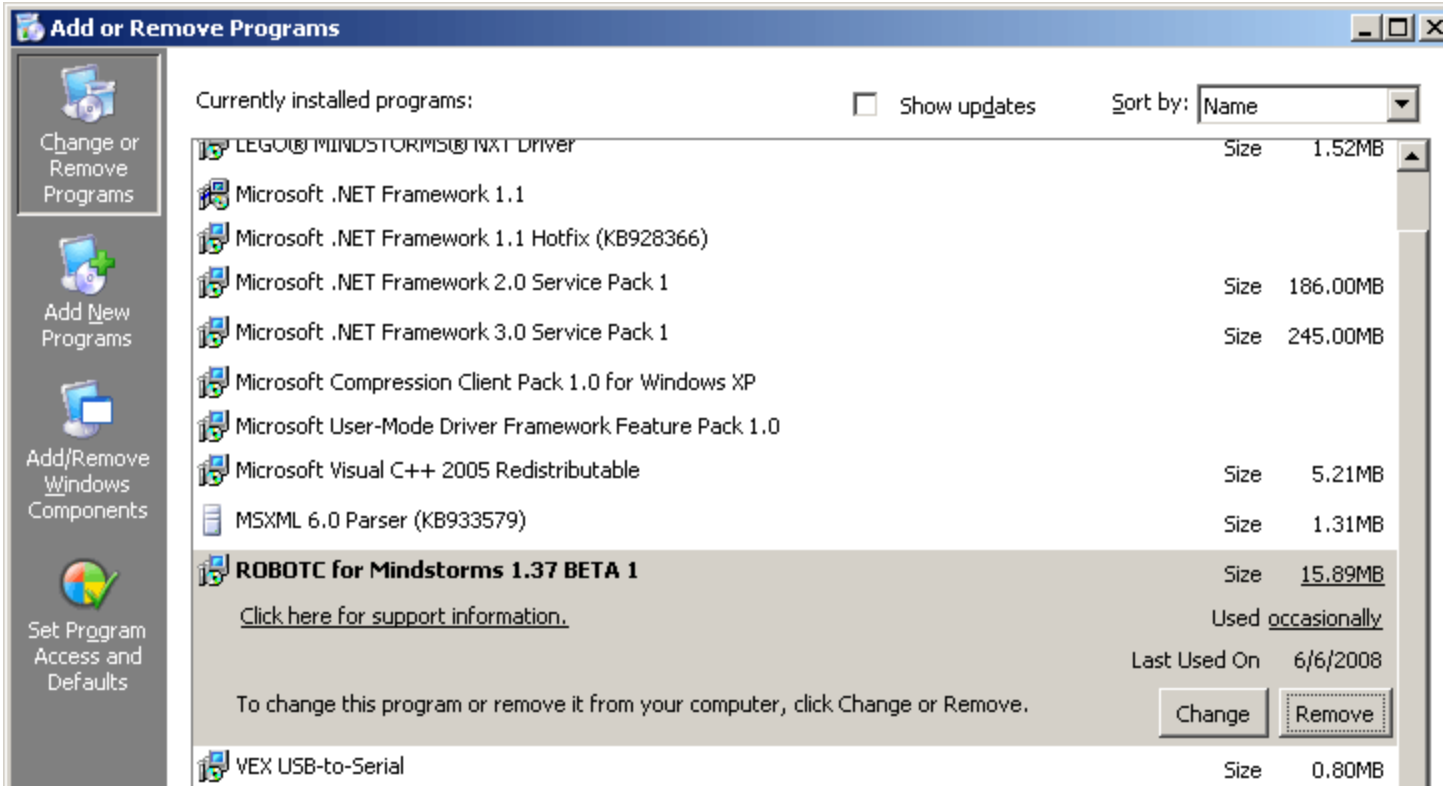
1. Open the Control Panel from your "Start Menu"



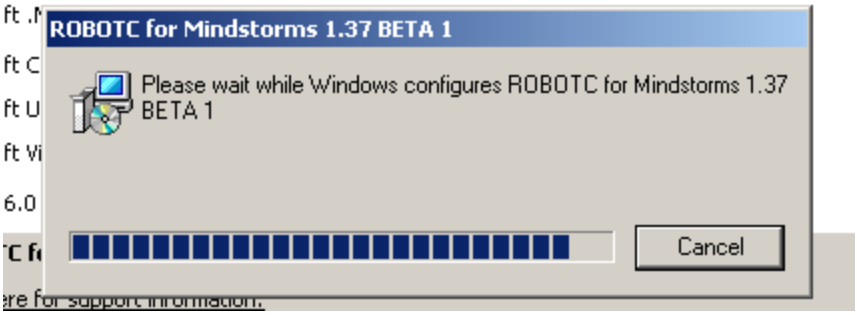

2. Inside of the Control Panel, select "Add or Remove Programs"




3. Navigate through the "Add or Remove Programs" list until you find your version of ROBOTC that you wish to uninstall. Press the "Remove button"



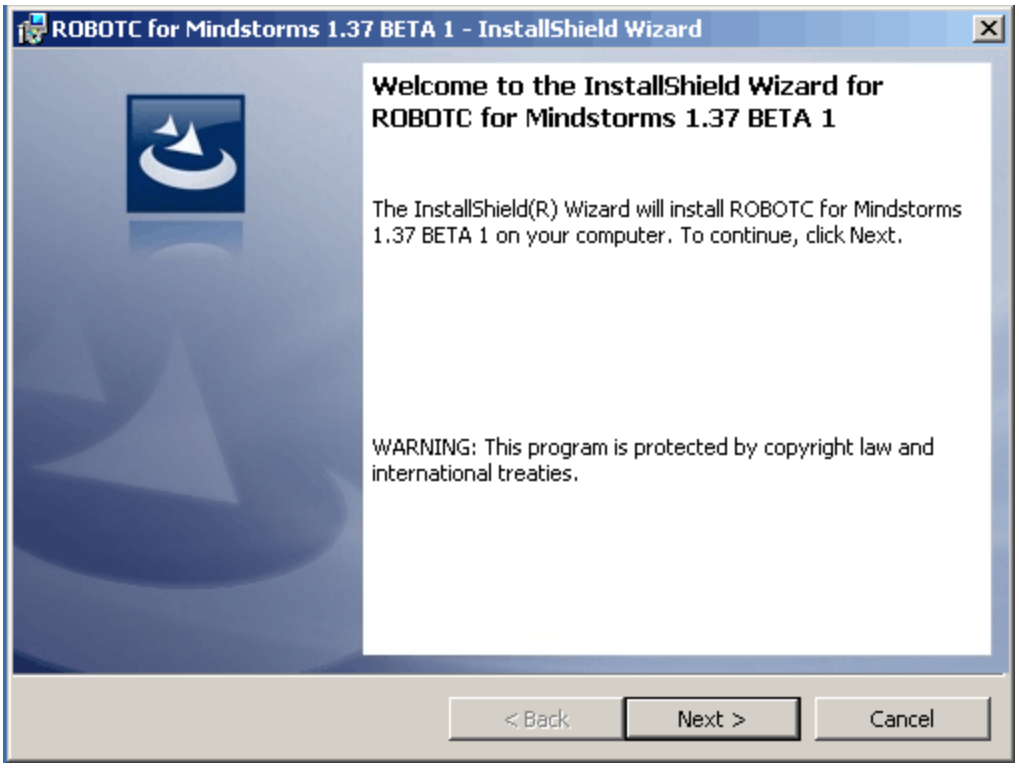

4. Windows will start the uninstall process. When the window below disappears and you are returned to the "Add or Remove Programs" list, ROBOTC has been successfully uninstalled.



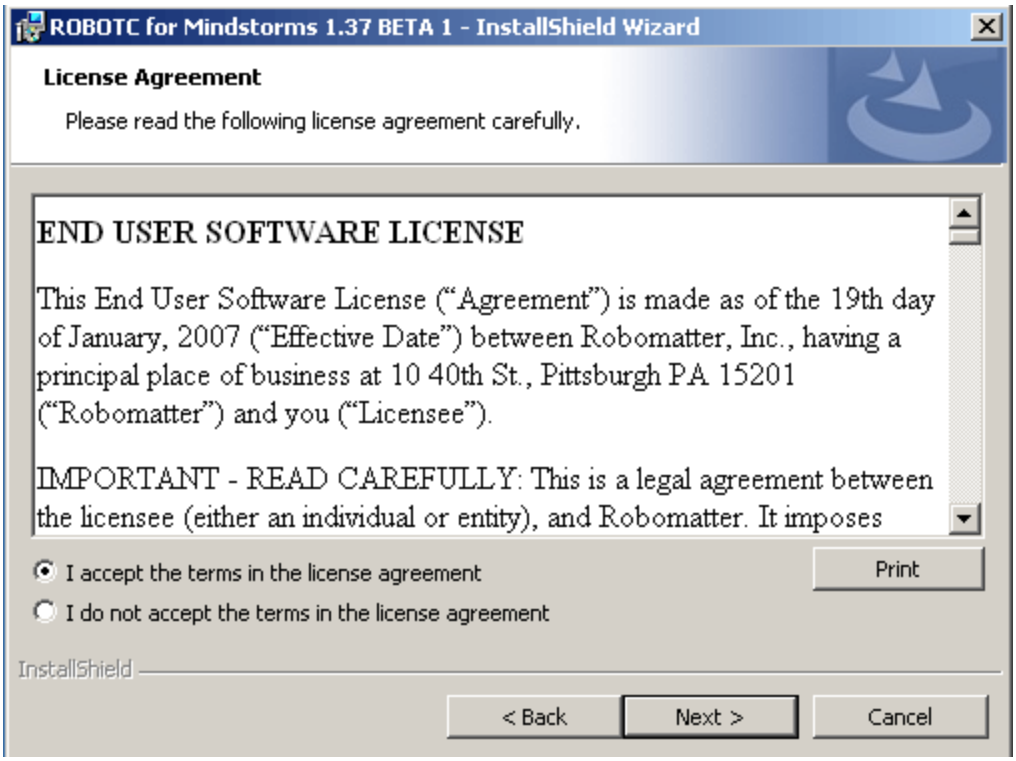

---

To Install the latest version of ROBOTC, follow these steps:

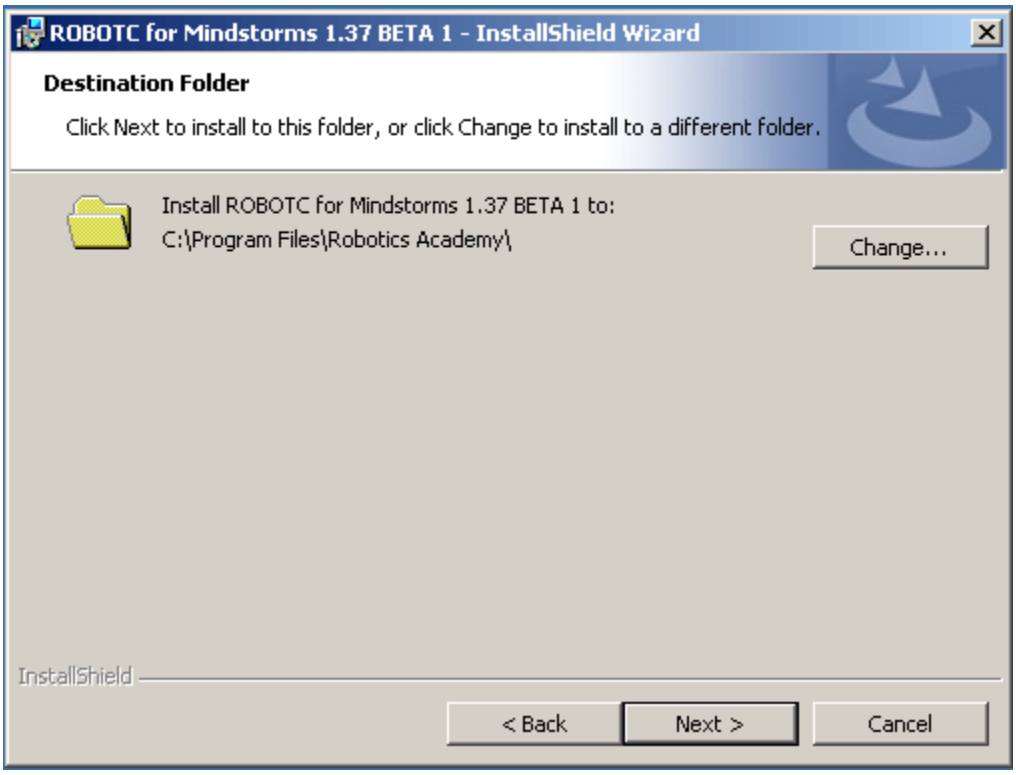
1. Double click on the ROBOTC installer. You will be presented with the Installshield installer for ROBOTC. Click "Next" to proceed.



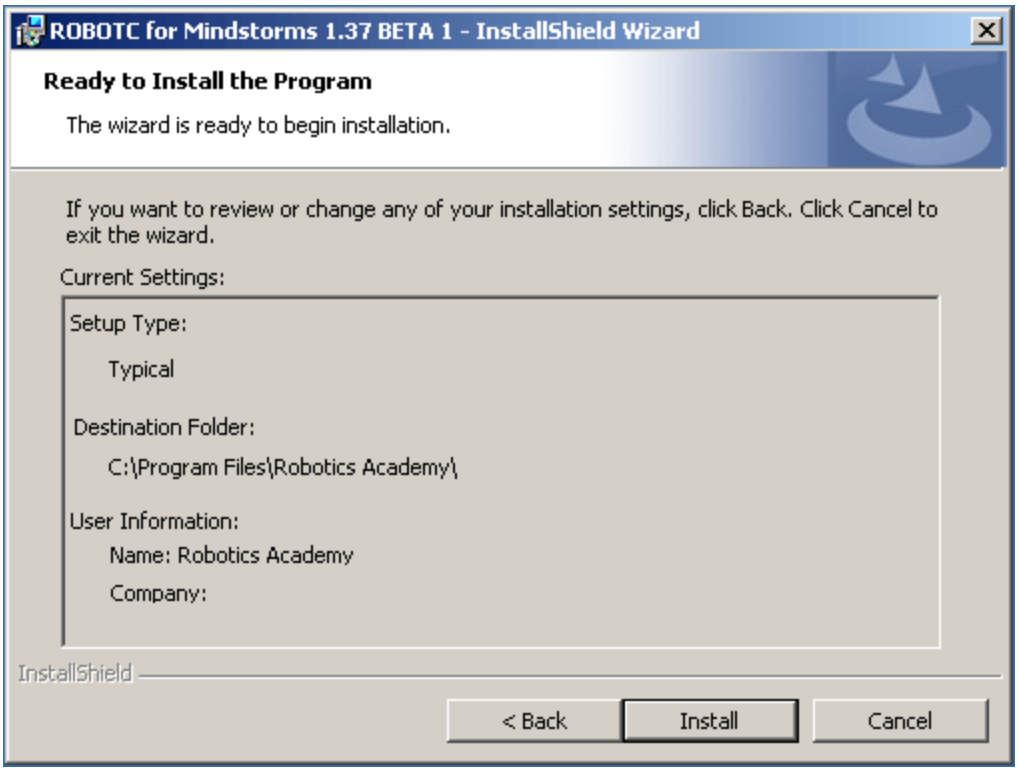

2. Before you can proceed, you must read over the accept the End User License Agreement. Click "I accept" and then the "Next" button to proceed.



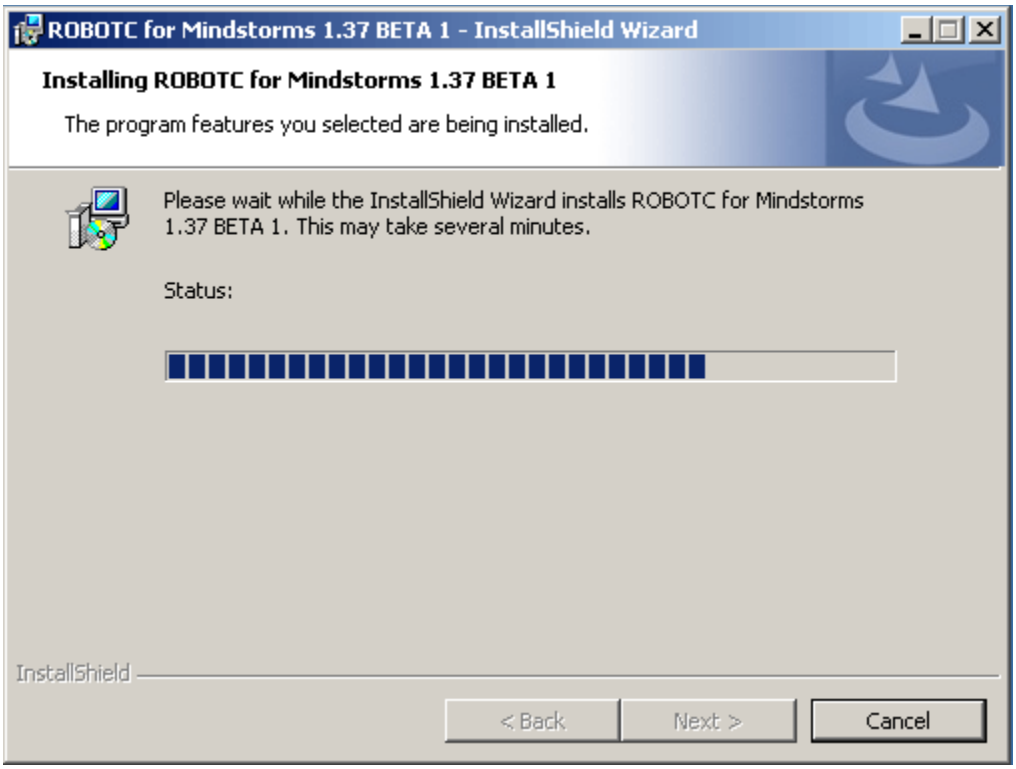
3. Choose where you would like to install ROBOTC to. By default, ROBOTC is installed to your "Program Files" folder. Click "Next" to proceed.



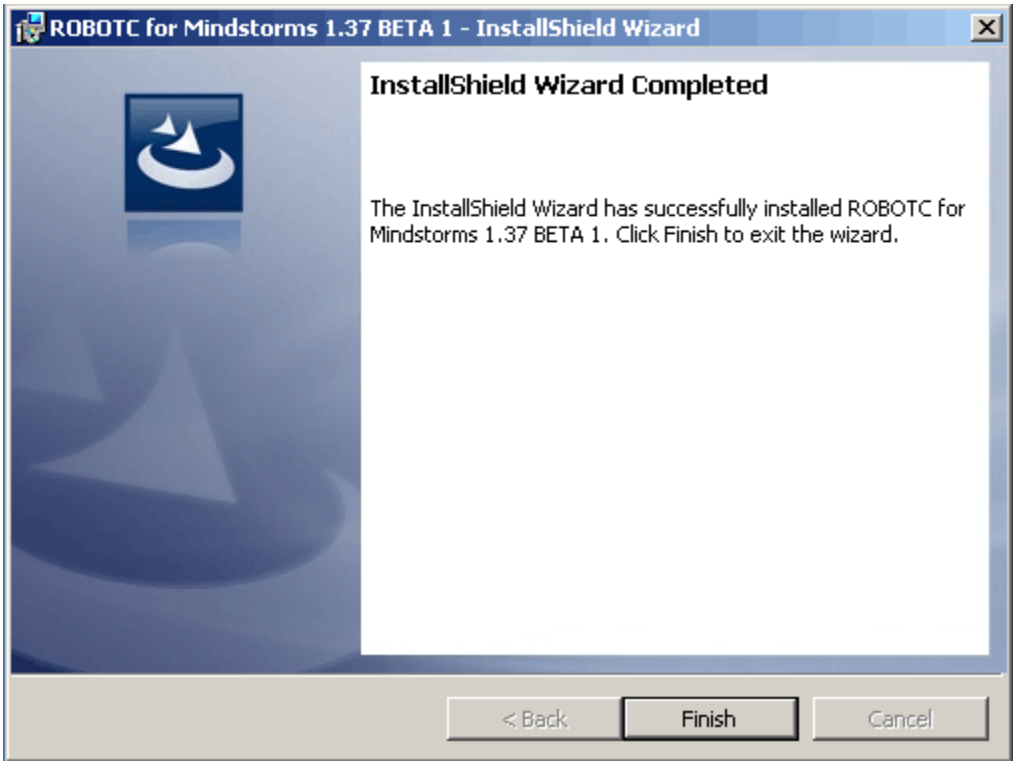

4. The installer will ask you to confirm your settings. If everything looks correct, click "Install" to being the installation process.




5. The installer will begin copying files from the installer to your computer. Please wait until this is complete.



6. When the installation is complete, the install will inform you of this. Click "Finish" to complete your install.




7. Double click the "ROBOTC for Mindstorms" icon to launch ROBOTC.


# 3.2 System Requirements

ROBOTC for MINDSTORMS requires basic system specifications to run properly. The development environment is not processor intensive, but the interactive debugger may slow down computer performance on older machines.

**System Requirements:**
Intel® Pentium® processor or compatible, 800 MHz minimum
Windows XP Professional or Home Edition with Service Pack 2 or greater
256MB of RAM minimum
Up to 30MB of available hard disk space
1 available USB port
Compatible Bluetooth adapter (optional)

ROBOTC for MINDSTORMS will not run natively in any other operating system other than Microsoft Windows. ROBOTC for MINDSTORMS will run with a Virtualization client on a Apple Macintosh. ROBOTC has been tested with VMWare Fusion, Parallels Desktop, and Apple's Boot Camp. You will still

need to download/install the RCX/NXT device drivers found on your purchased CD or on the ROBOTC website.

# 3.3 Activate Online

**To continue using ROBOTC past the 30 day trial period,** you must purchase a ROBOTC license and then activate the license on each computer.

To activate ROBOTC, you will need the **License ID and Password** that was sent to you during the purchase process.

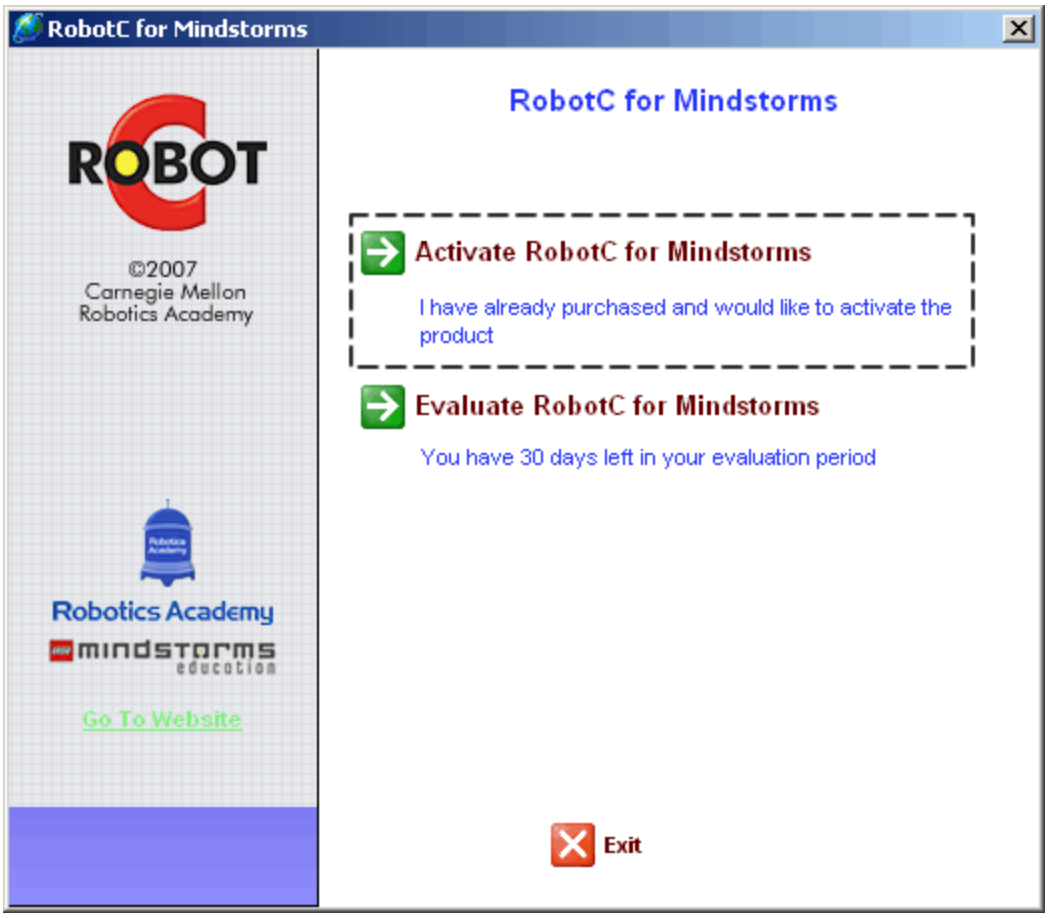
- **If you purchased a ROBOTC license online,** the License ID and Password were sent to the email address specified.
- **If you ordered the CD-ROM version of ROBOTC,** the License ID and Password are printed on the CD label.

**If you are Activating the ROBOTC license on computers with internet access, follow the Activation instructions below.** If you are activating the ROBOTC license on computers without internet access, skip to Activate by Web instructions.

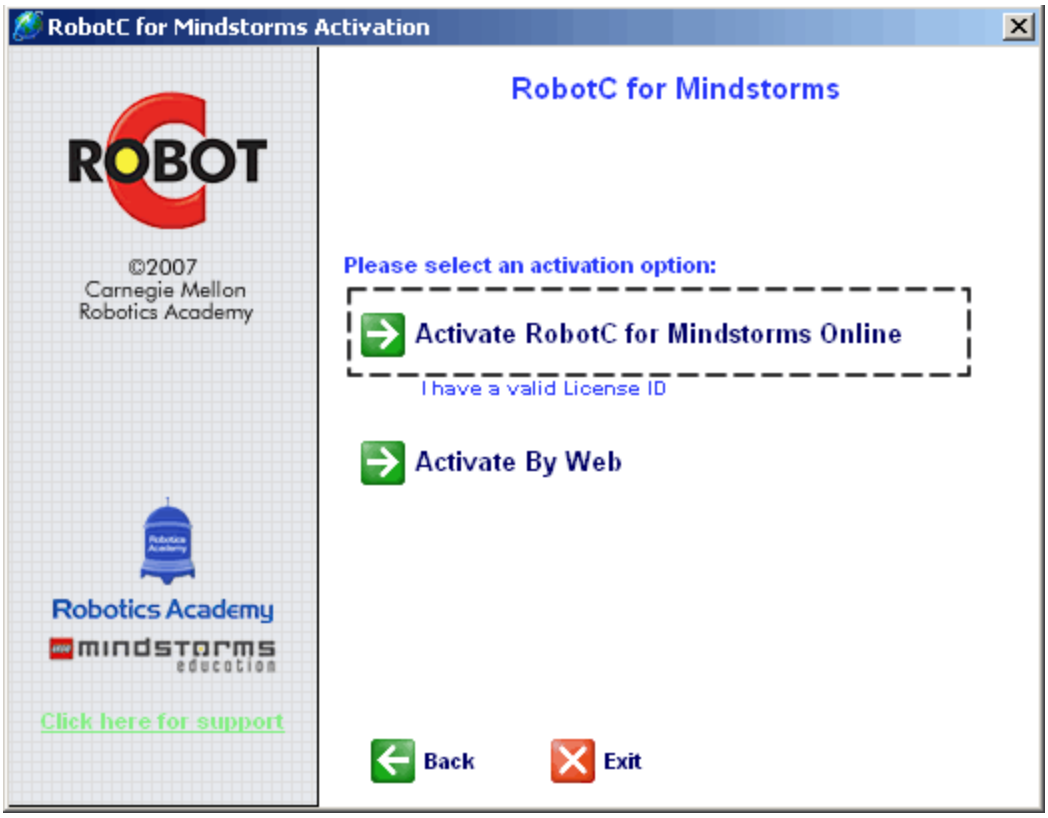
*NOTE: ROBOTC must be activated on each individual computer.*

Follow these steps to activate on a computer with internet access:

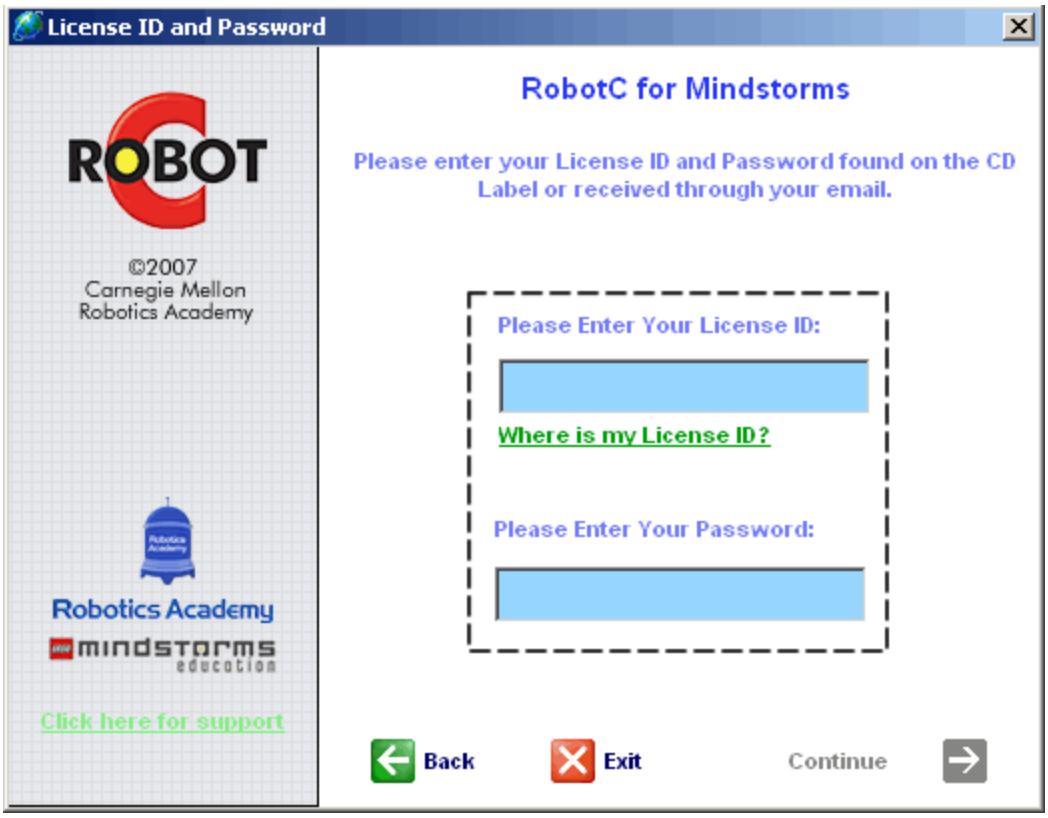
1. Open ROBOTC and select "Activate ROBOTC" from the start-up prompt.



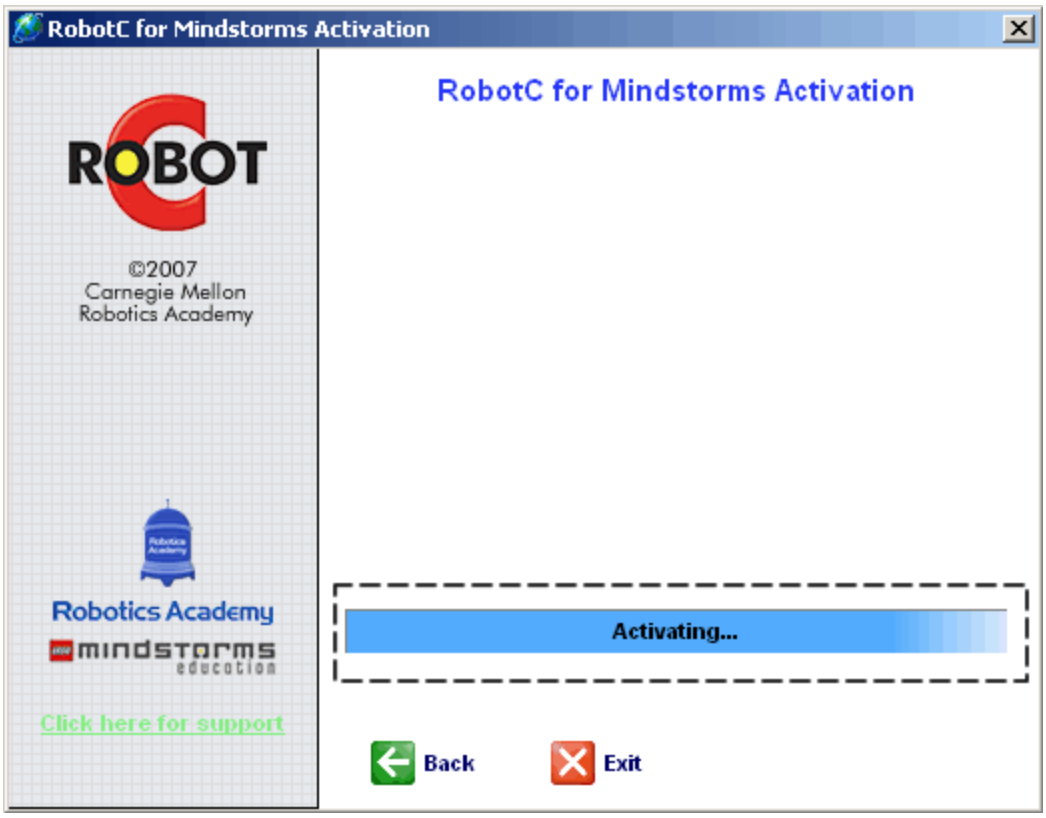

2. Select "Activate ROBOTC for MINDSTORMS Online"



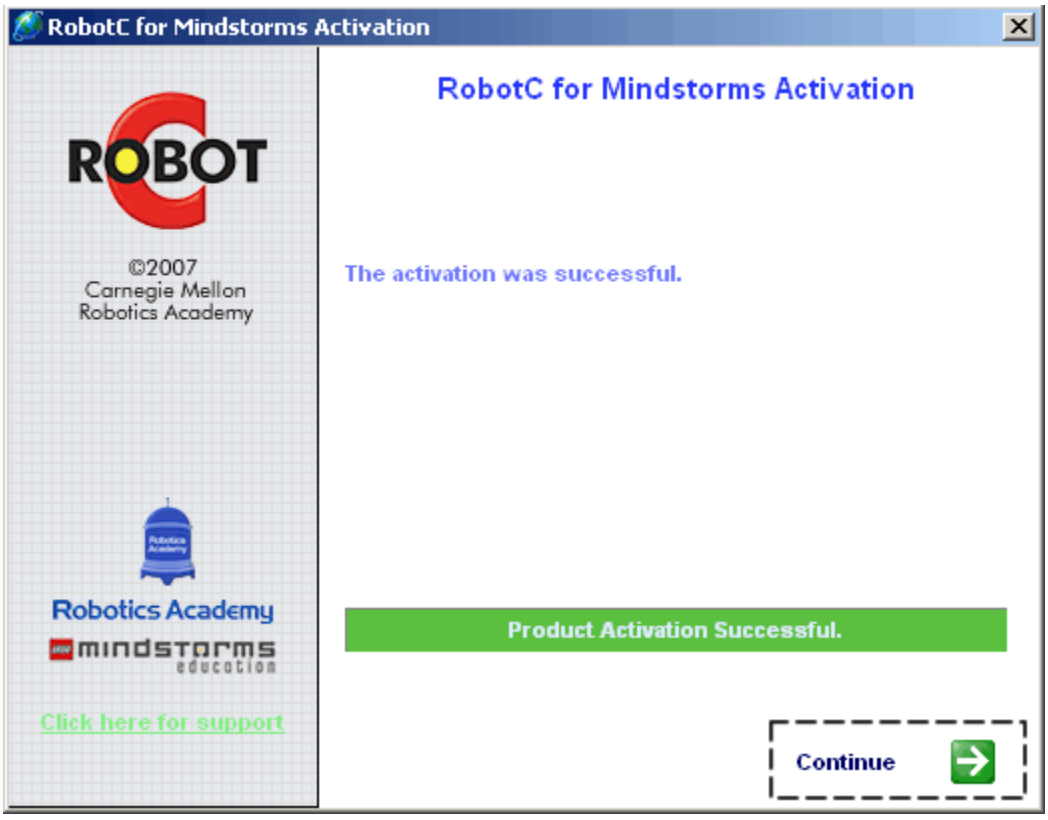

3. Enter in your License ID and Password that you received via emaill. Click the "Continue" button to proceed.




4. ROBOTC will communicate with the activation server via the internet to verify your License ID and Password.




5. ROBOTC is activated and ready for use. Click Continue to use ROBOTC for MINDSTORMS.

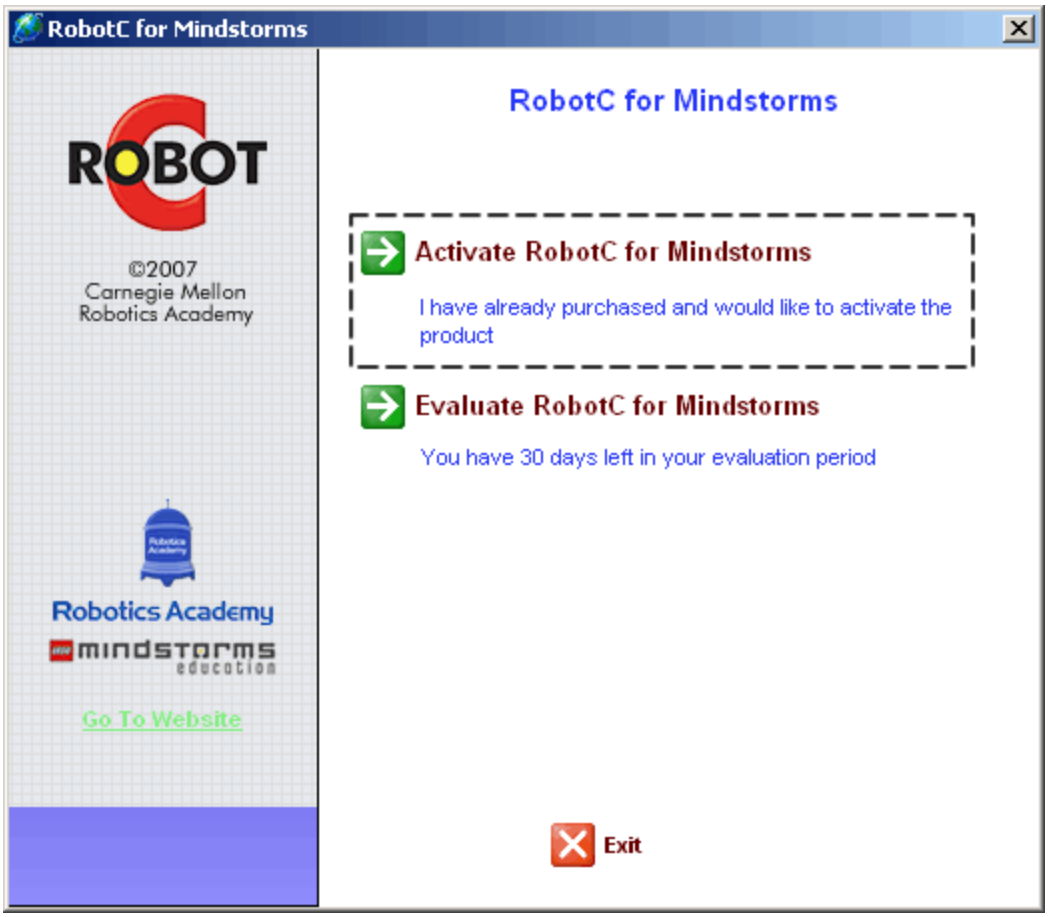


# 3.4 Activate by Web

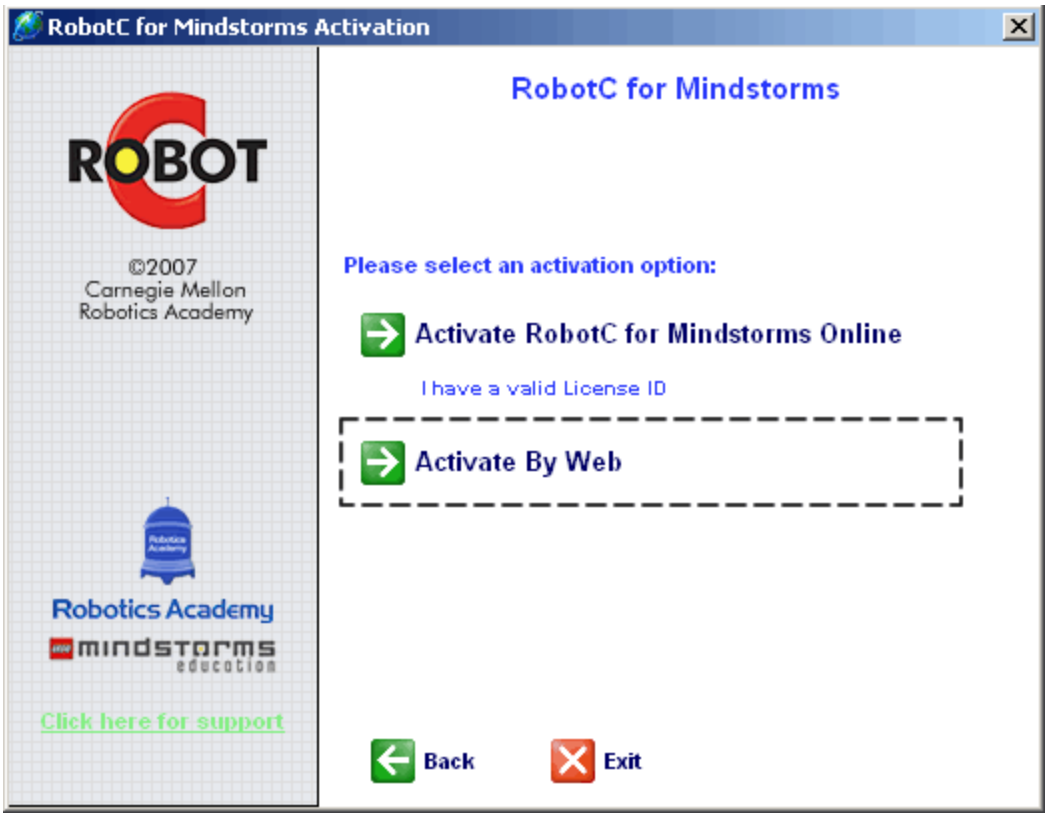
Follow these steps to activate on a computer without internet access:

Note: You will need to use a separate computer with Internet access in step 4 at this activation process. You must repeat this process on each computer without internet access.

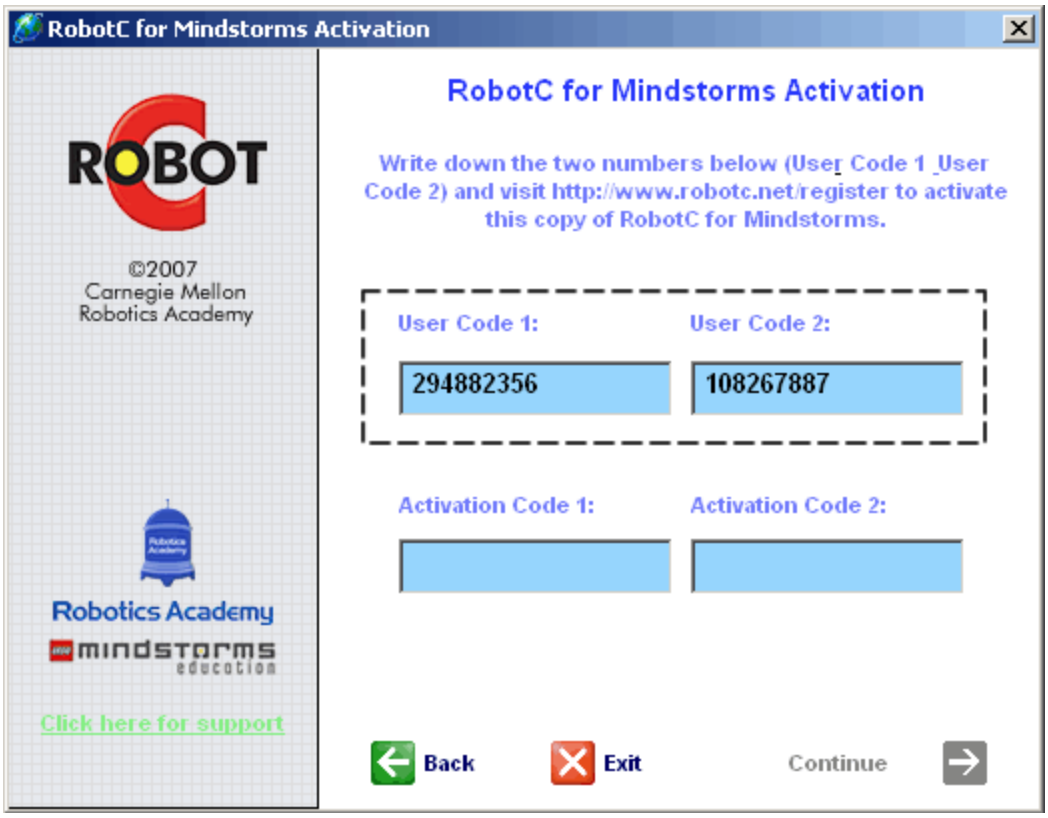
1. Open ROBOTC and select "Activate ROBOTC" from the menu.



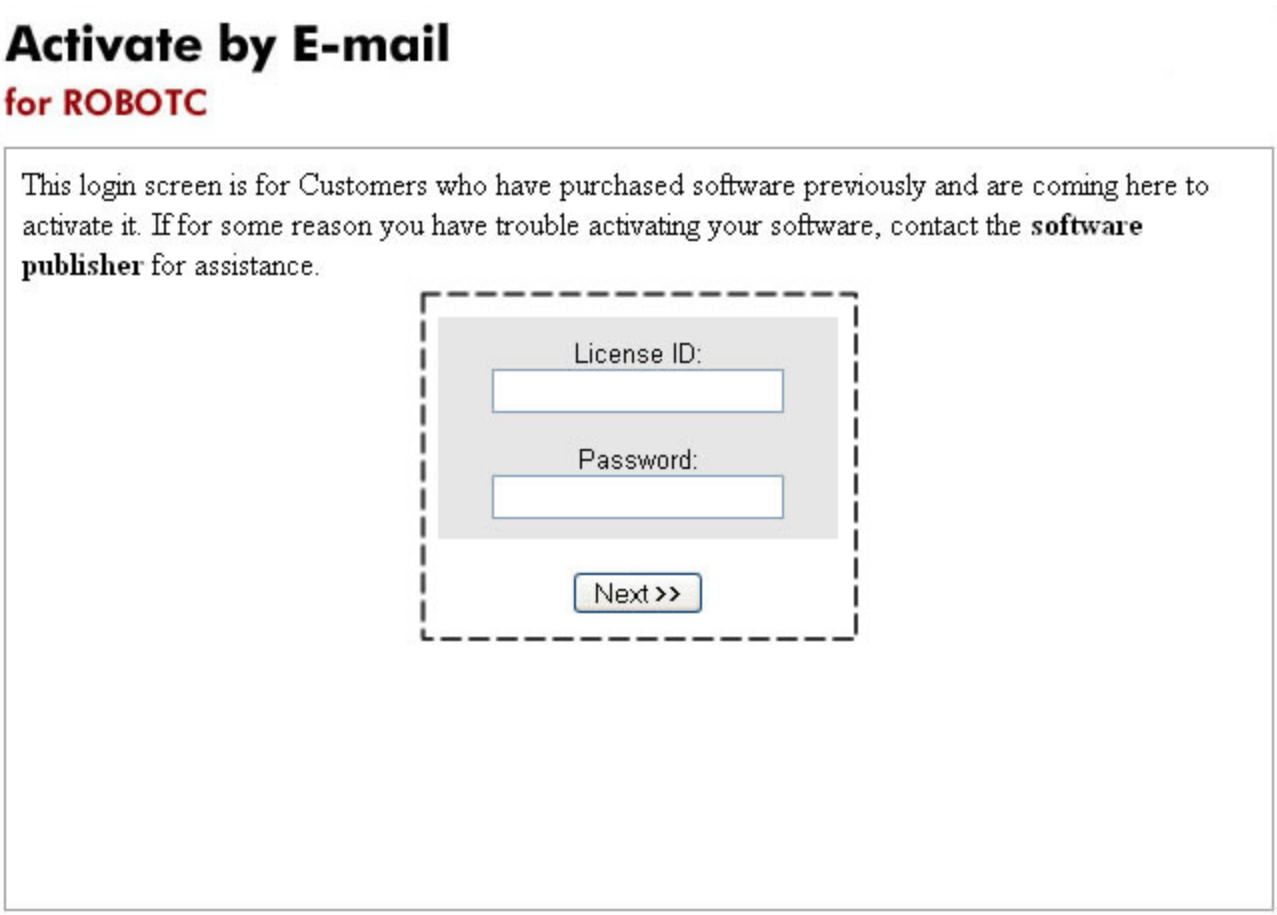

2. Select "Activate by Web"



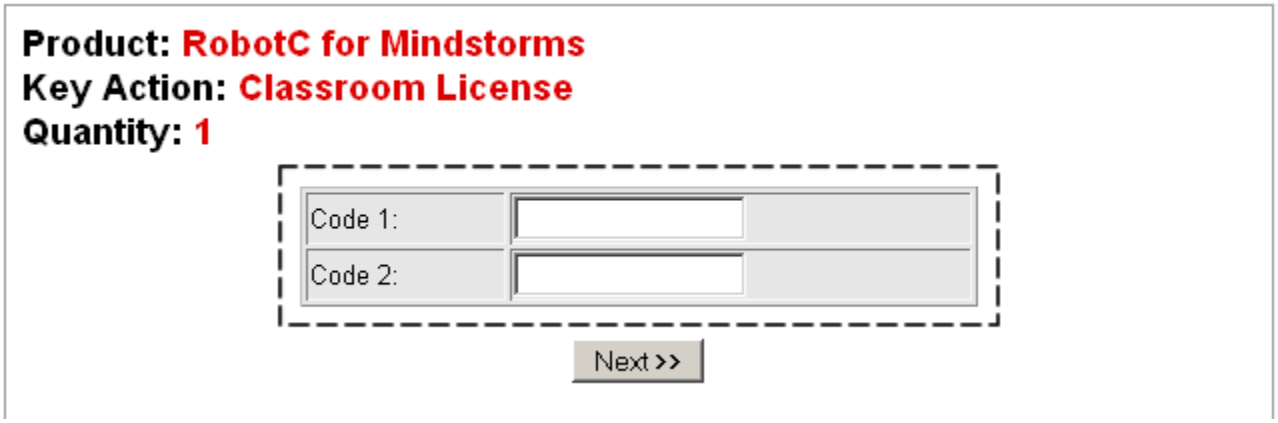

3. Two user codes will be displayed. Copy both of these down. Take these codes with you to a separate computer with Internet access.




4. At a separate computer with internet access, visit http://www.robotc.net/register. Enter your License ID and Password that you received via email and click "Next."




5. Enter User Code 1 and User Code 2 that you copied down in step 3. Click "Next" to proceed.



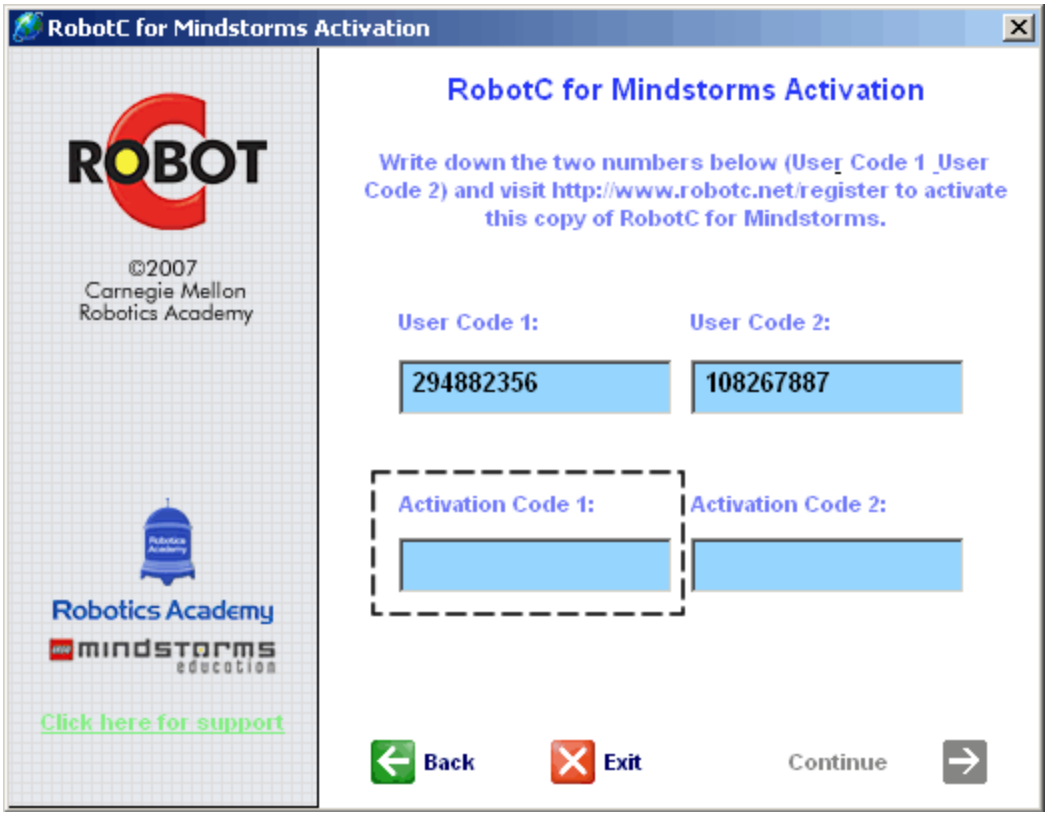

6. A Registration code will be displayed. Write this down and return to the original computer on which ROBOTC is installed.

**Product: RobotC for Mindstorms Classroom License**
**Quantity: 1**
**License ID: 775098**

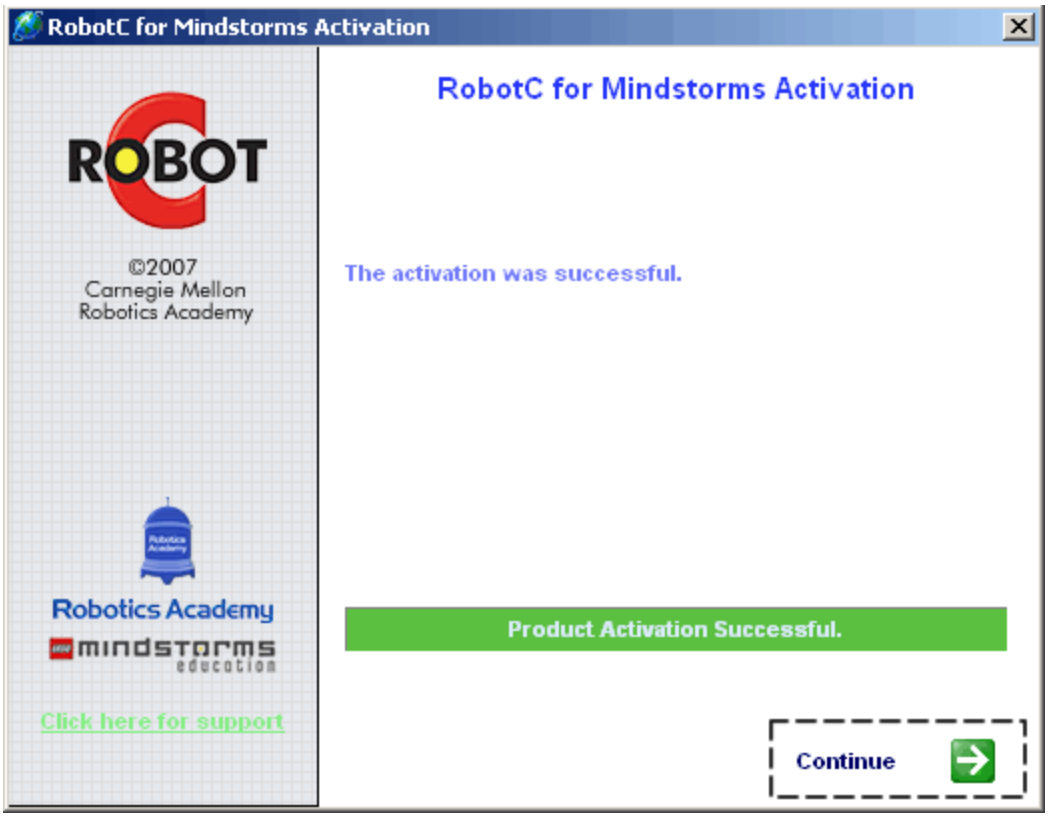
**Registration code1 = 221150841**

Go back to License Page

7. Enter the Registration code from step 6 into the "Activation Code 1" field.
Note: Leave Activation Code 2 blank.




8. ROBOTC will be activated and ready for use. Click Continue to use ROBOTC for MINDSTORMS.

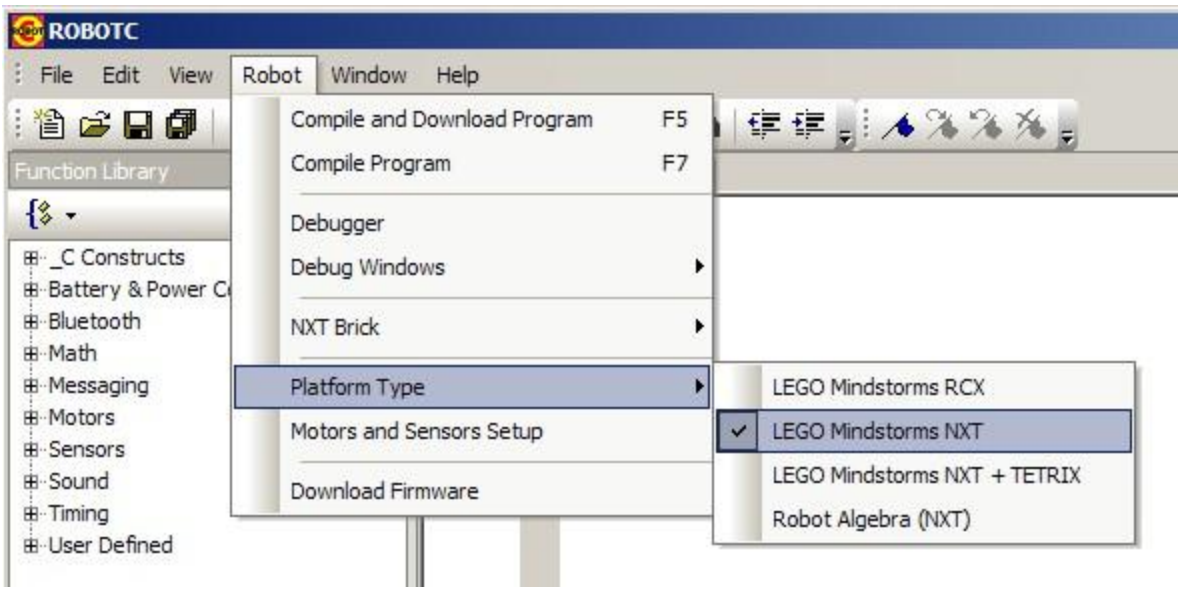



# 3.5 Select Platform Type

ROBOTC for MINDSTORMS has support for four different platforms:
- LEGO Mindstorms RCX
- LEGO Mindstorms NXT
- LEGO Mindstorms NXT + TETRIX
- Robot Algebra (NXT)

Before you use ROBOTC, make sure to select which platform you are currently using.

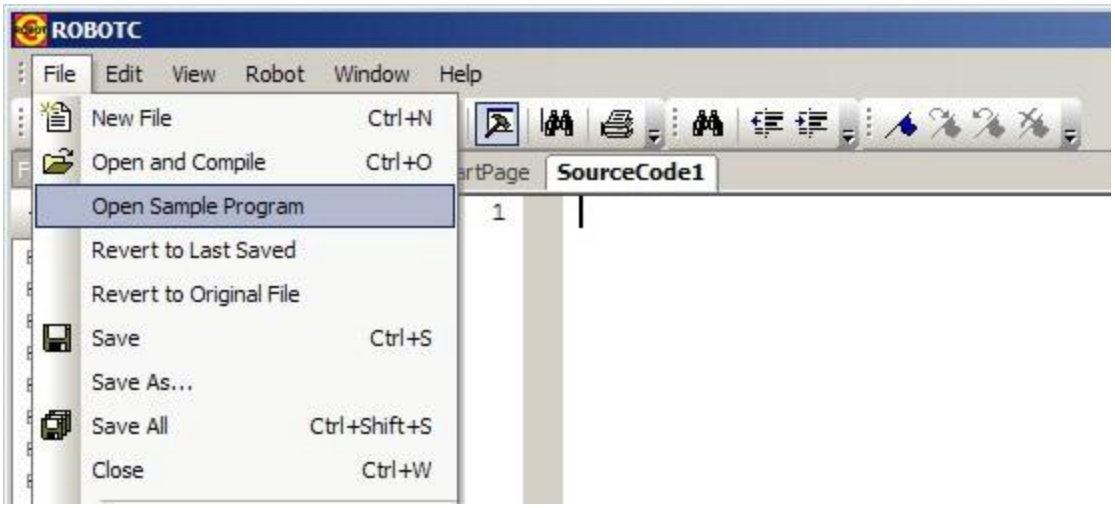
**Changing Platform Type:**
Open the "Robot" menu in ROBOTC and hover your cursor over the "Platform Type" menu option. This will open up the platform select menu. Select the platform you wish you use with ROBOTC. You can change the platform type at any time.



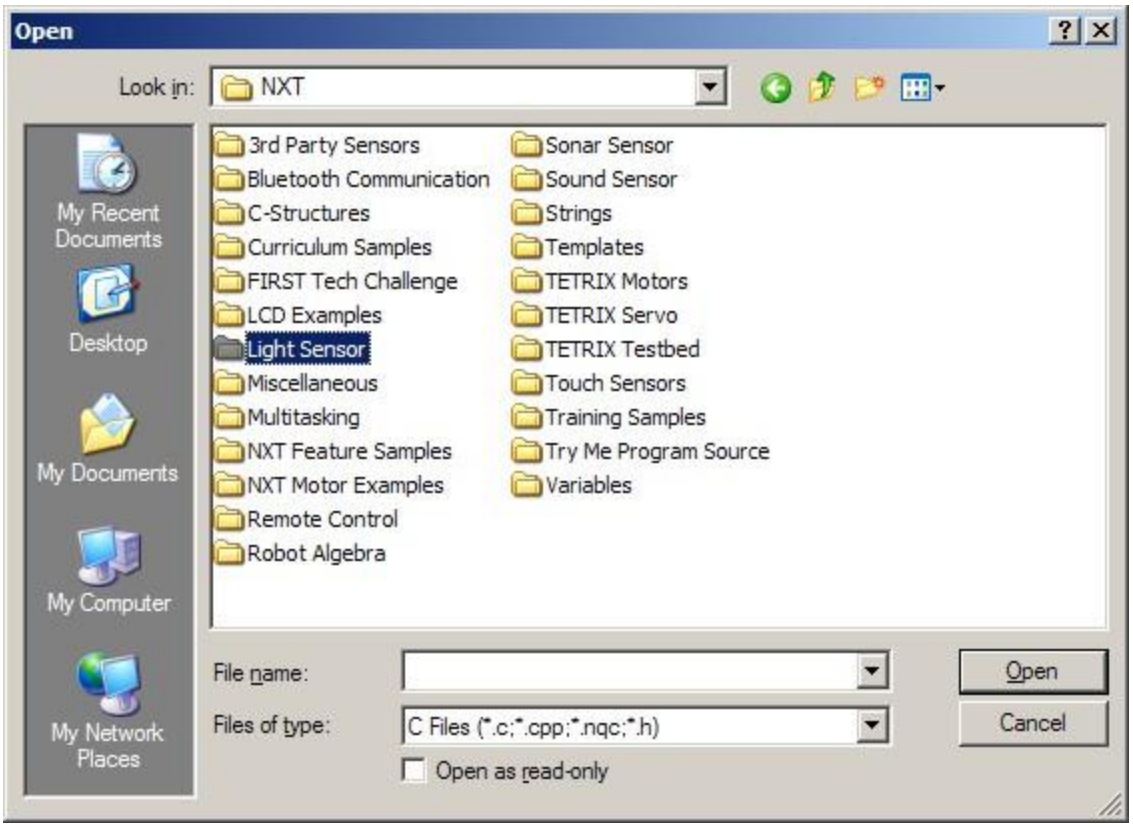

# 3.6 Sample Programs

One of the best ways to start working with ROBOTC is to look at already developed programs. ROBOTC for Mindstorms comes with over 150 sample programs to help new programmers learn how to program their robots.

To access the sample programs, go to the "File" menu and select "Open Sample Program."




The sample programs folder is organized by topic. ROBOTC will automatically open the sample programs folder for the robot platform you have selected.



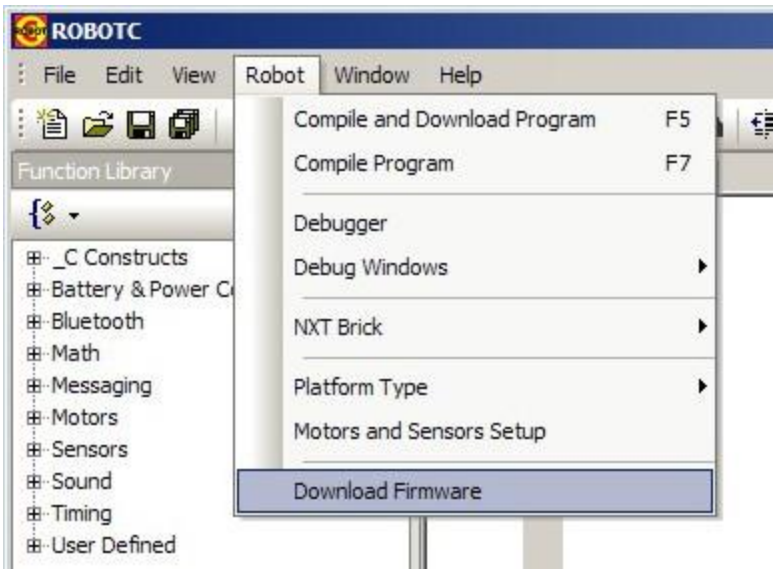

# 3.7 Download Firmware - NXT and TETRIX

ROBOTC requires a different firmware from the other programming languages available for the NXT. This firmware is what unlocks most of the advanced functionality found in ROBOTC.

You can update the firmware on the NXT directly from the ROBOTC application. To update the firmware, follow these instructions:

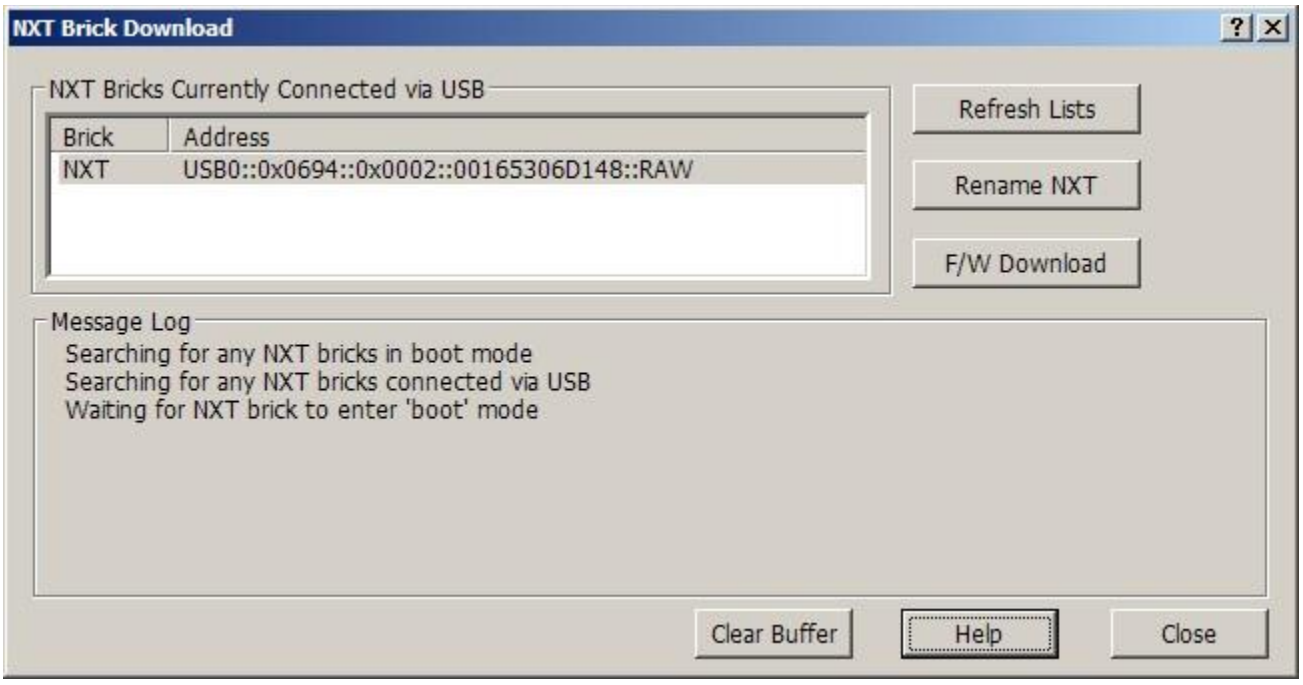
**Prerequisite: Make sure that your NXT is connected to your computer via USB and is powered on with a fully charged battery.**

1. Open the "Robot" menu and select "Download Firmware."




2. Since NXT is set as the Platform Type, the "NXT Brick Download" screen will appear. From this screen you can rename your NXT and update the firmware. Make sure that your NXT appears under the "NXT Bricks Currently Connected via USB" list before proceeding. If the NXT does not appear, make sure it is connected via USB, powered on, and that the NXT USB driver is installed. Then click the "Refresh Lists" button to see if your brick can be found.

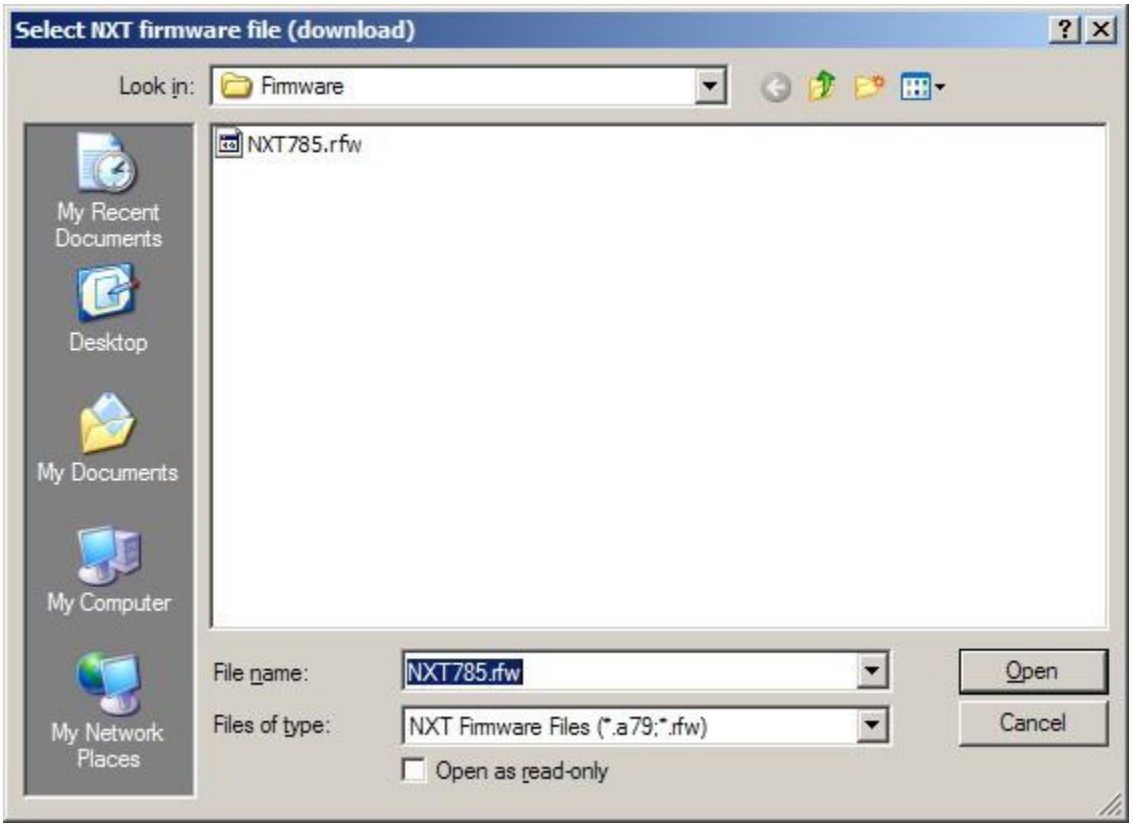
3. To start downloading firmware, click the "F/W Download Button."



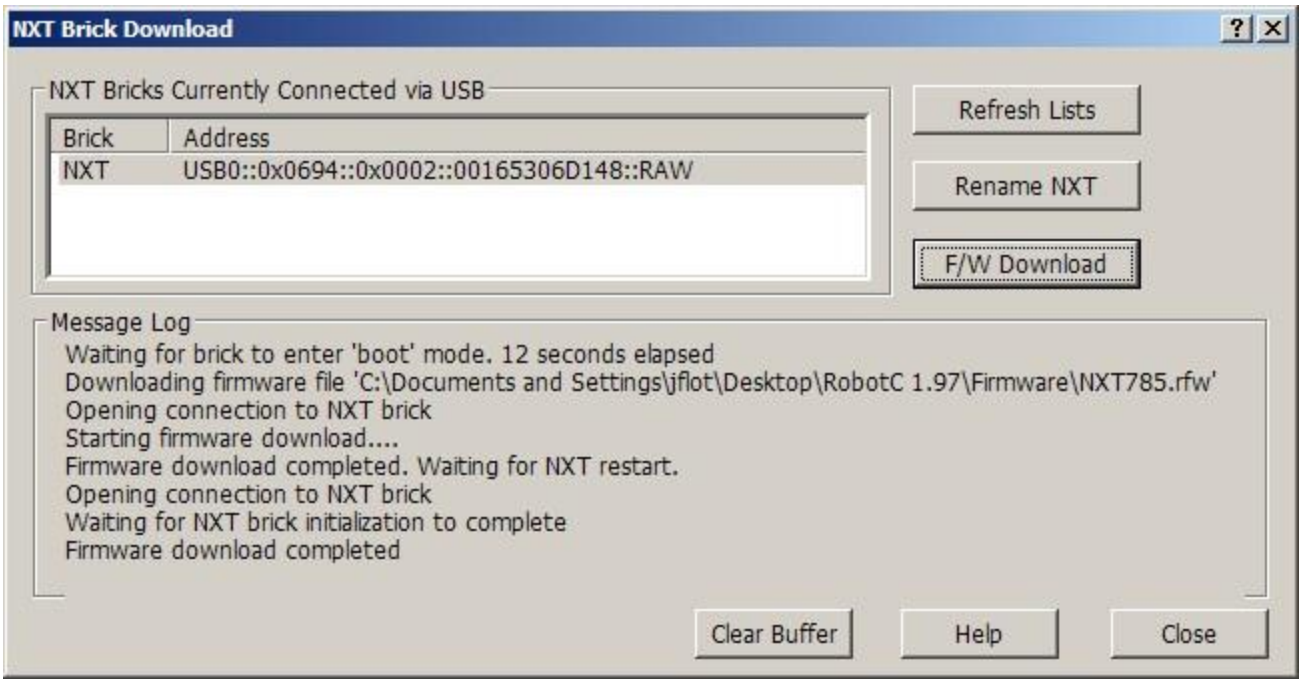

4. After clicking the "F/W Download" button, you will see the "Select NXT firmware file" file selection screen. ROBOTC will automatically open to the Firmware folder and show you all of the available firmwares available for the NXT. NXT firmware files have the extension ".rfw". Select the highest

numbered firmware file that follows the "NXTXXXX.rfw" format. Once that firmware file is selected, click the "Open" button to start downloading the firmware to the NXT.

Note: Always select the highest version number available with the "NXT" prefix.




5. As the firmware is downloading, your NXT will shut itself off and turn itself back on. Once the firmware has been successfully downloaded, you should see "Firmware download completed" on the "NXT Brick Download" screen, under the "Message Log". Your NXT is now ready to be used with ROBOTC.



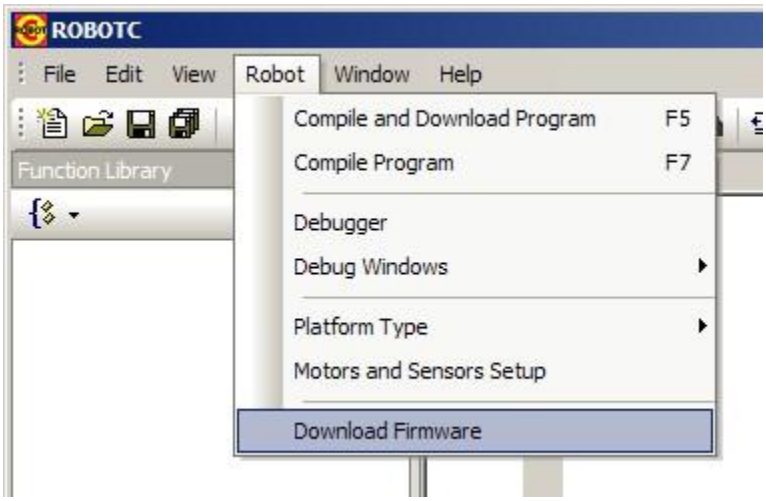

# 3.8 Download Firmware - RCX

ROBOTC requires a different firmware from the other programming languages available for the RCX. This firmware is what unlocks most of the advanced functionality found in ROBOTC.

You can update the firmware on the RCX directly from the ROBOTC application. To update the firmware, follow these instructions:

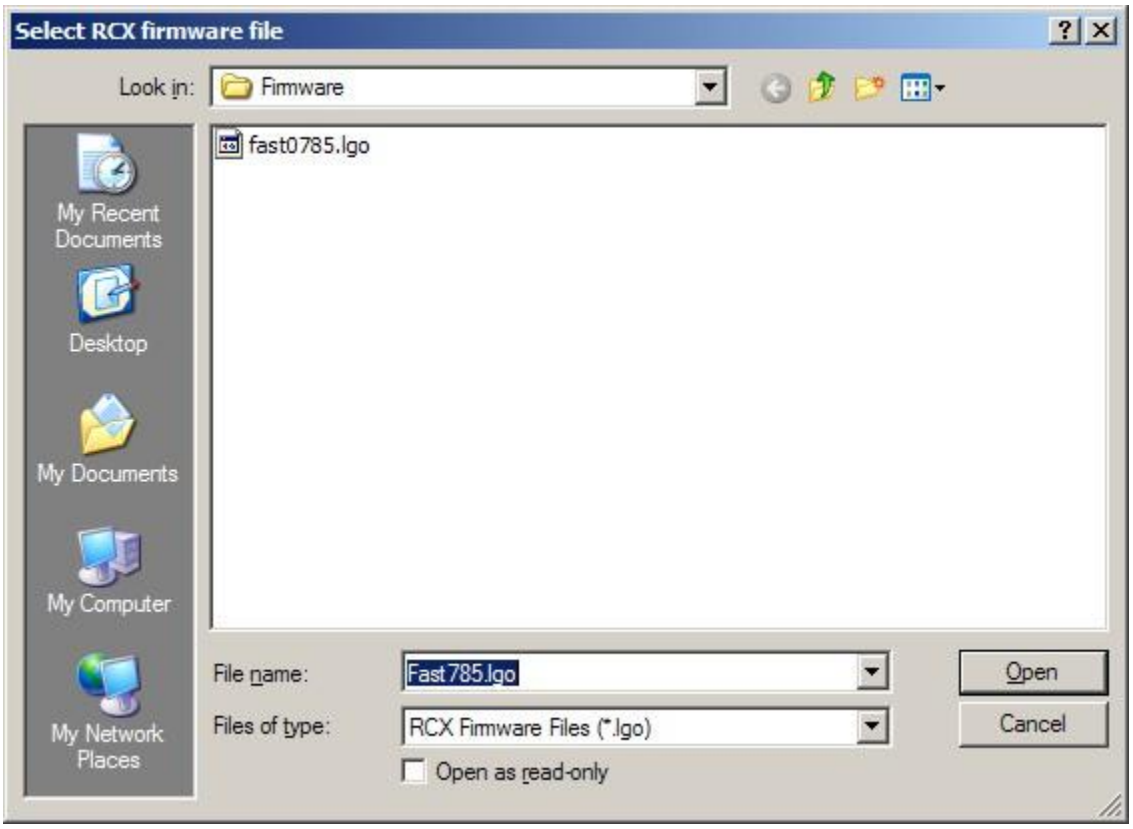
**Prerequisite: Make sure that your RCX powered on with fresh batteries, and that the IR Tower is facing the front of the RCX.**

1. Open the "Robot" menu and select "Download Firmware."

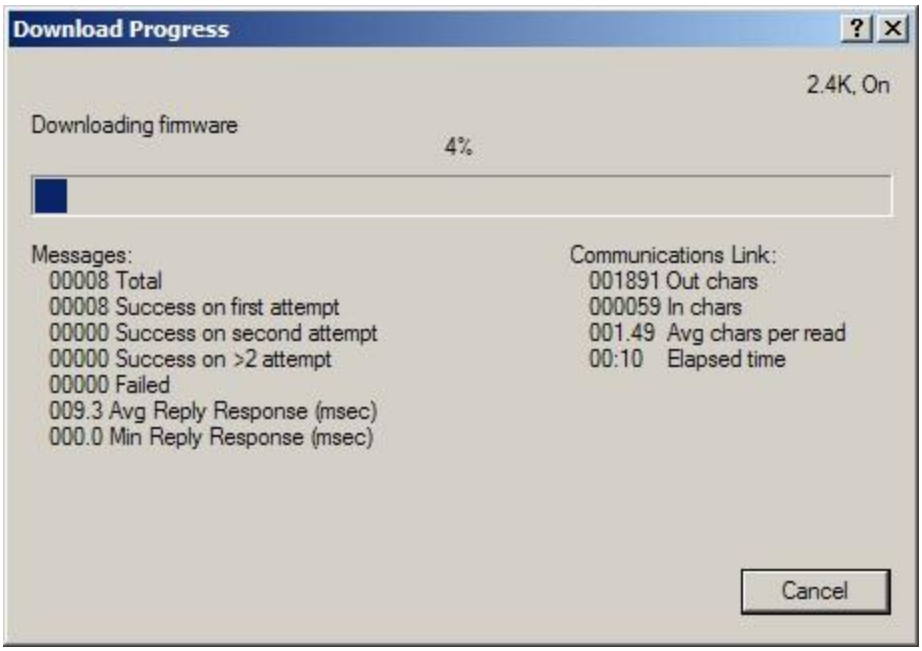


2. After clicking the "F/W Download" button, you will see the "Select RCX firmware file" selection screen. ROBOTC will automatically open to the Firmware folder and show you all of the available firmwares for the RCX. RCX firmware files have the extension ".lgo". Select the highest numbered firmware file that follows the "fastXXXX.lgo" format. Once that firmware file is selected, click the "Open" button to start downloading the firmware to the RCX.

Note: Always select the highest version number available with the "fast" prefix.

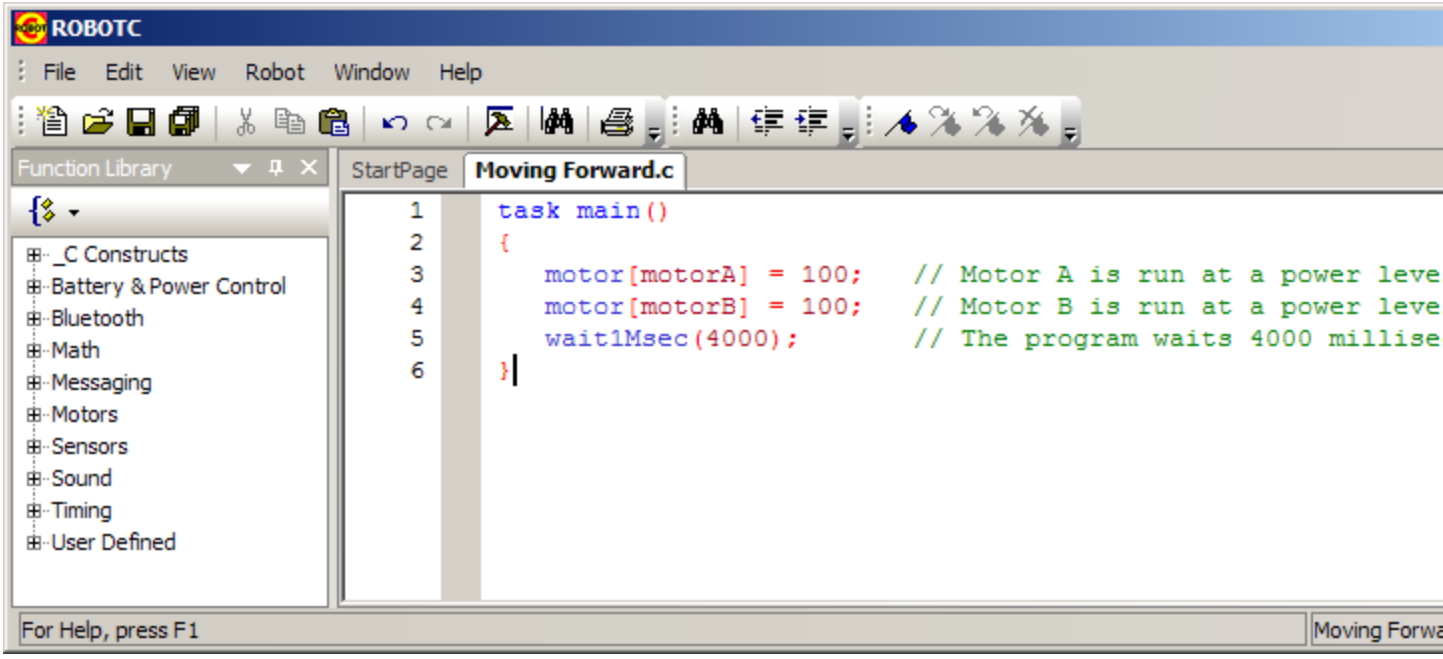



3. A download progress window appears showing the RCX firmware being downloaded. The firmware may take up to 5 minutes to download completely. To assist the firmware download and prevent any failures, make sure to cover the RCX and IR Tower to sheild them from ambient light or IR radiation from other towers. When the firmware transfer is complete, the RCX will make a sound and the "Download Progress" window will close. Your RCX is now ready to be used with ROBOTC.

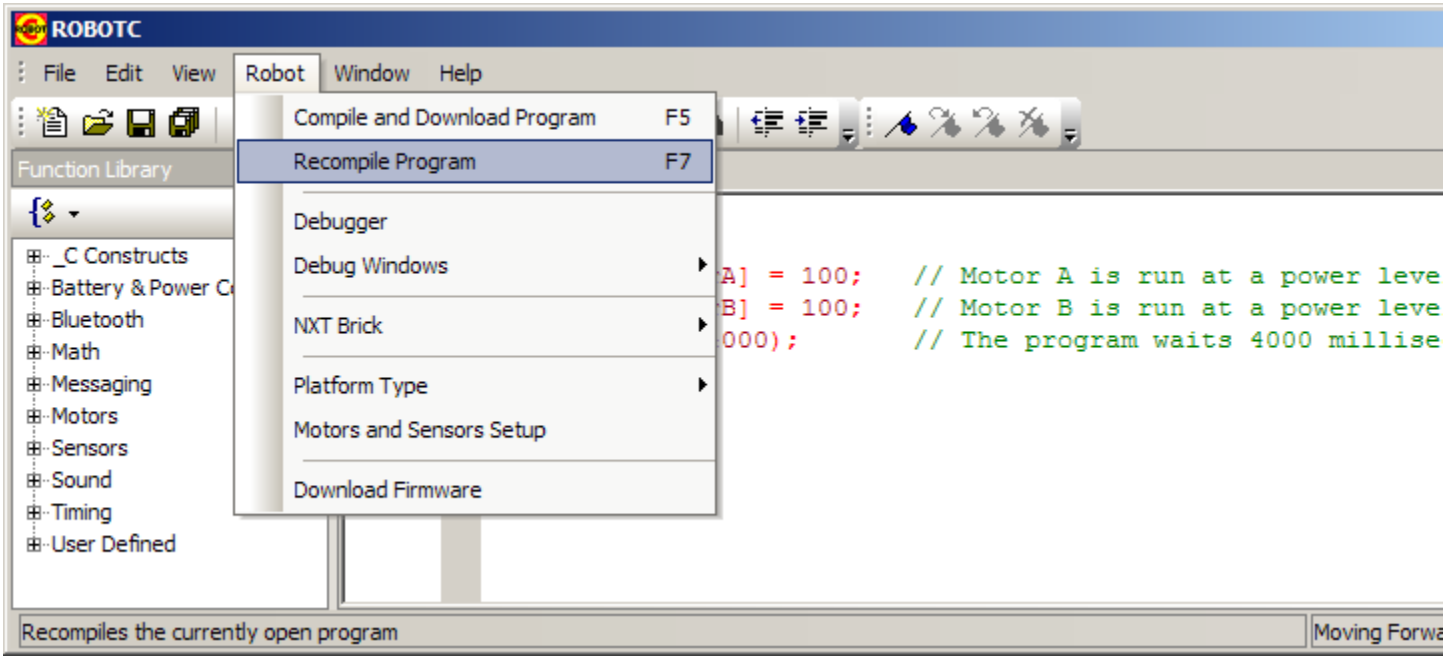



# 3.9 Compiling and Downloading

ROBOTC is an "Integrated Development Environment." ROBOTC has a custom text editor to assist the programmer by color coding different potions of code to differentiate between integers, reserved words, functions and parameters.



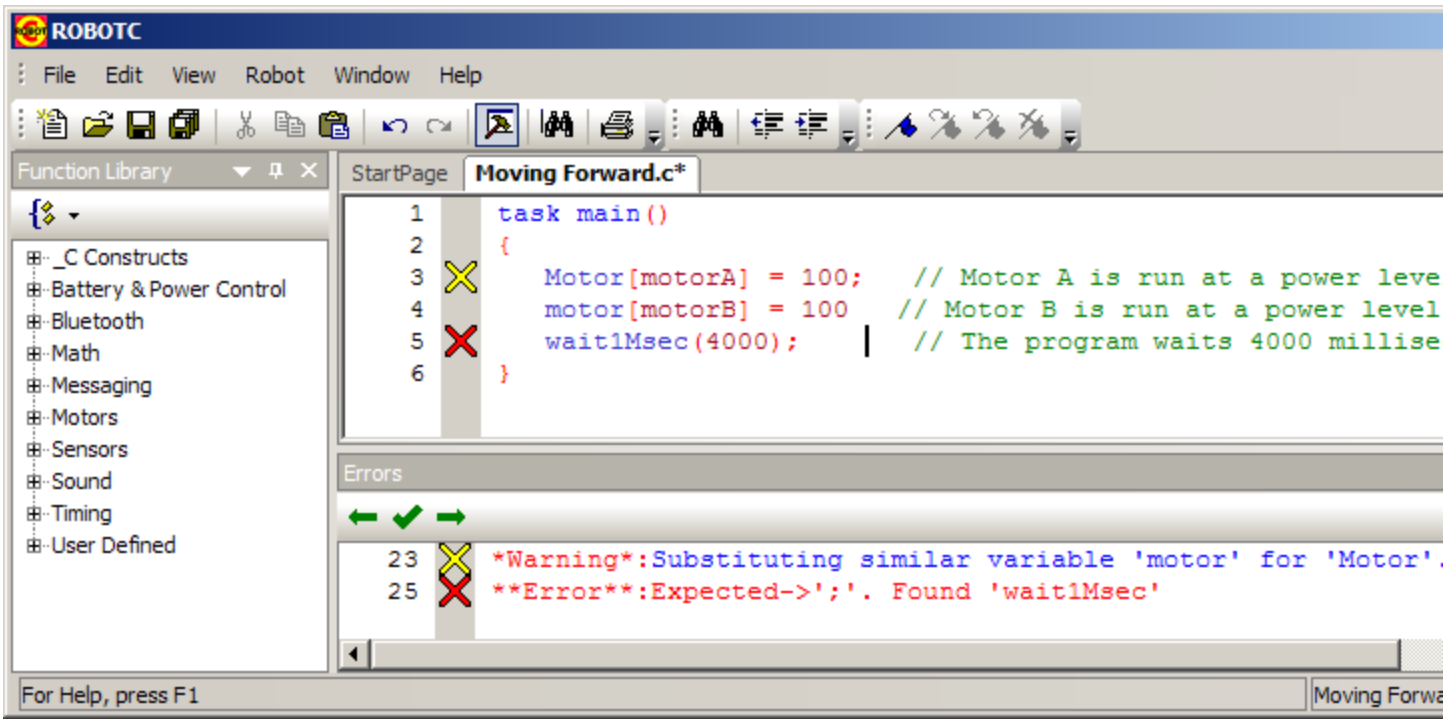

ROBOTC also contains a compiler for turning user generated code into a byte-code language that the different robot platforms can understand. To run the compiler, go to the "Robot" menu and select "Recompile Program" or hit F7 on your keyboard.



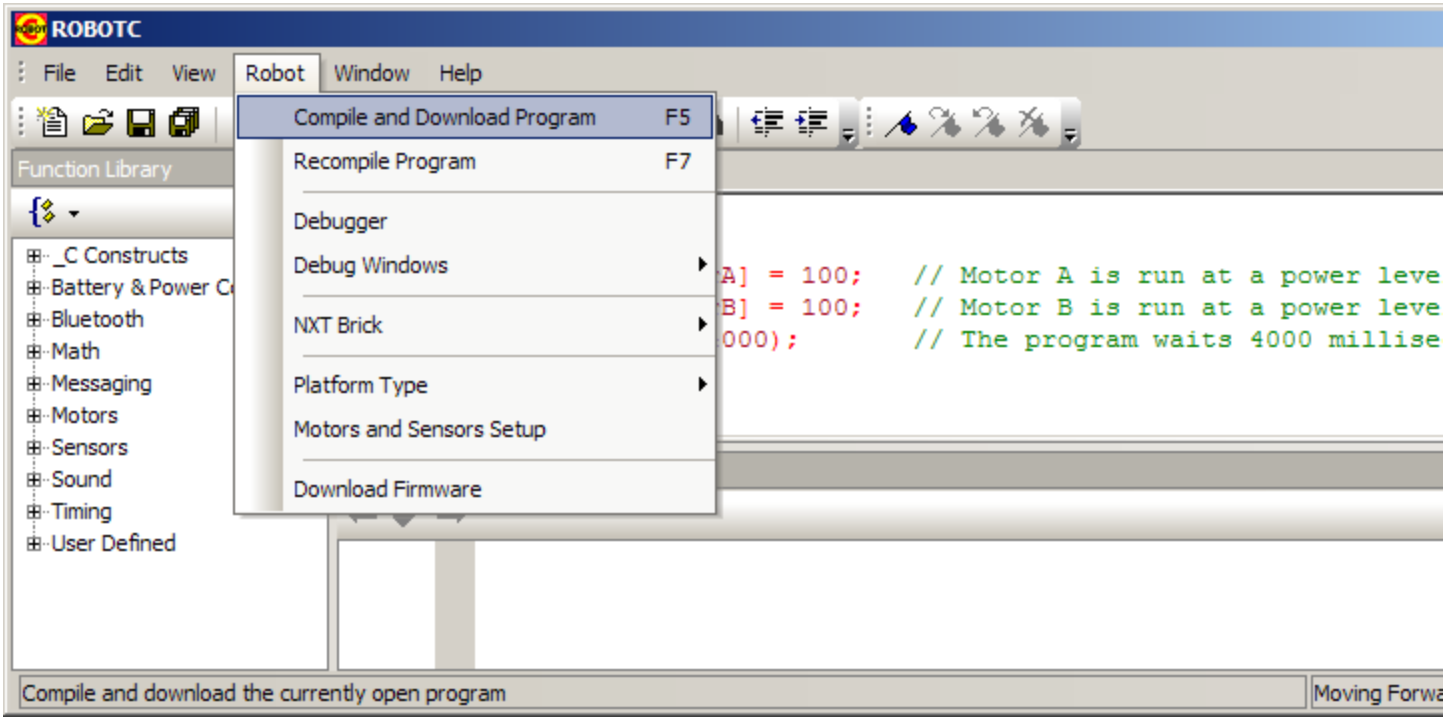

When the compiler runs, ROBOTC will check your program for errors and warnings that would cause it to run improperly on the robot. Rather than send broken code to your robot, ROBOTC informs you of these errors.

In the example below:
- T he reserved word 'motor' was incorrectly typed as 'Motor', generating a warning. A warning tells the user that they've created an error, but ROBOTC is able to correct it when the program is sent to the robot. A program with warnings will compile successfully, but users should use good programming practice to avoid warnings. Warnings are denoted by a yellow "X".

- The code on line 5 is missing a semicolon, which is generating an error. An error will prevent the program from being sent to the controller. In this example, ROBOTC is informing the user they left a semicolon out of their program which is preventing the program from being compiled. Errors are denoted by a red "X".

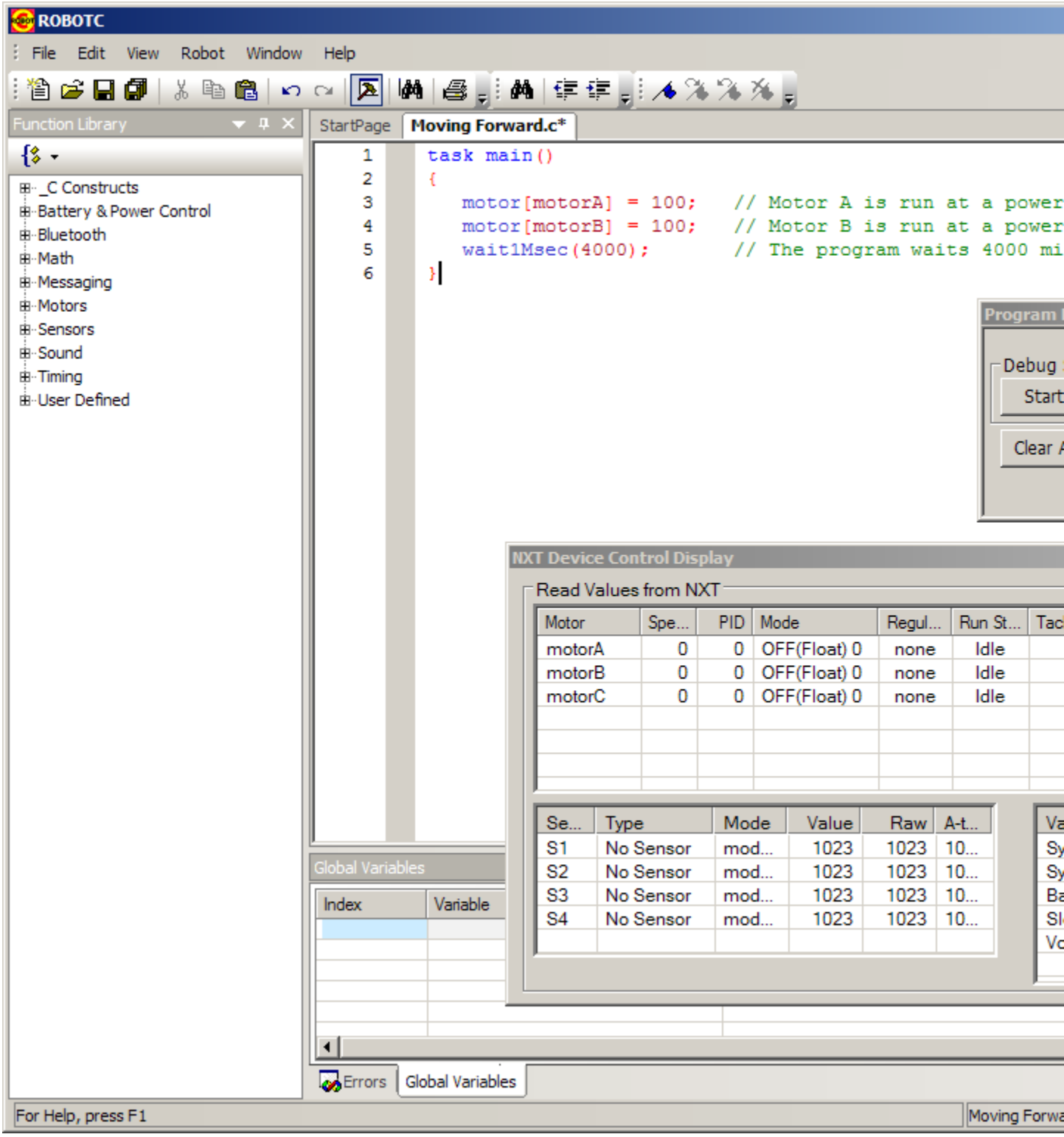



Once your program has successfully compiled, you can send the program to the controller by selecting the "Compile and Download Program" option under the "Robot" menu or hitting F5 on your keyboard.



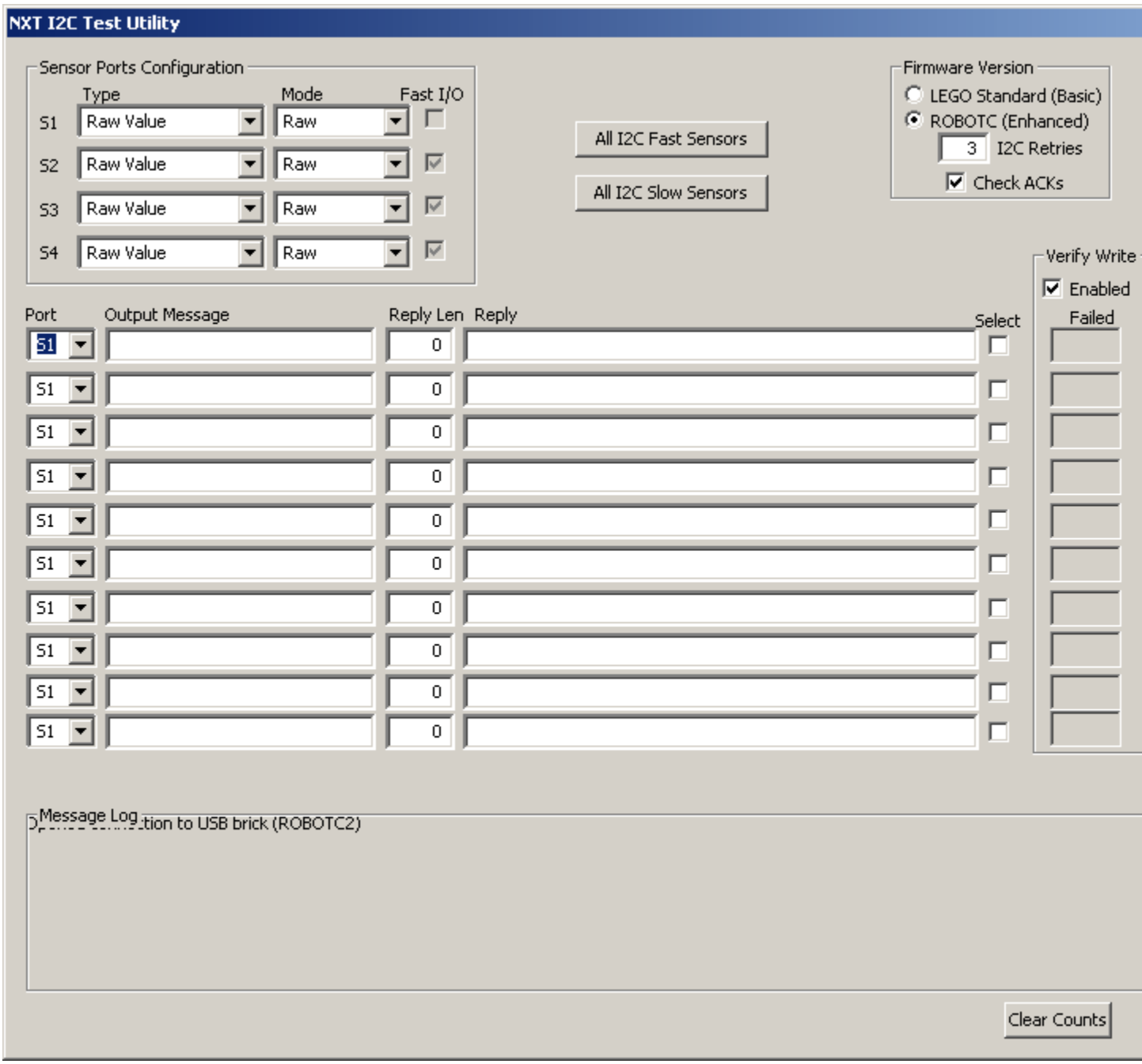

Once the download starts, a "Download Progress" window will appear. This window shows the transfer status of the compiled byte-code to the robot.

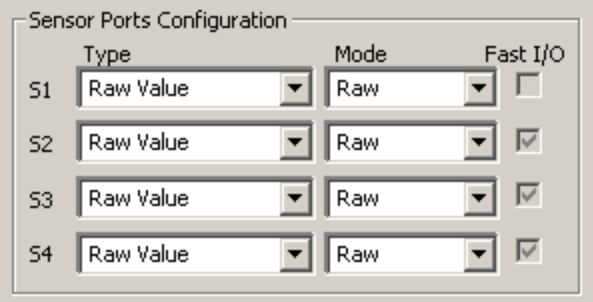
Once the download finishes, the built-in debugger launch. Some windows will launch, such as the "NXT Device Control Display" and the "Program Debug" window. Others may launch docked into the interface, such as the "Global Variables" screen. You can start your programs execution by clicking the "Start" button on the "Program Debug" window. Learn more about these windows in the 'Debugger' section of the help file.



# 3.10 Test I2C

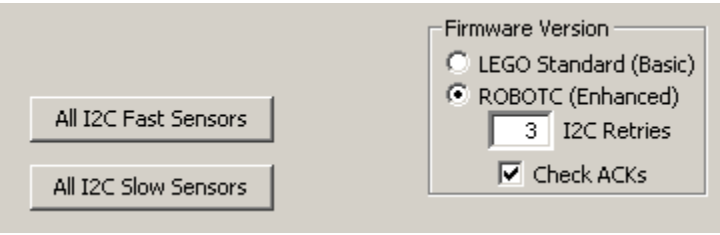
ROBOTC has a very powerful utility for testing digital sensors on the NXT. Digital sensors are those that support the industry standard I2C protocol for communications between the NXT and the sensor. The utility allows you to easily test an I2C sensor on the NXT. A screen shot of the utility is shown below.






### Sensor Port Configuration

Use these drop down menu items to set the sensor type and sensor mode for each of the four sensor ports. When this utility is first opened, it will interrogate the NXT to get the current settings of these values. The “Fast I/O” check-mark is read-only. It is calculated from the type of sensor. If you want to test a LEGO ultrasonic tester with this utility, do not set the type to “SONAR”. This will activate the built-in firmware device driver which will periodically poll the value of the ultra-sonic sensor. The device driver does not expect any other application to send I2C messages to the sensor! Instead set the type to “I2C Custom Slow”.

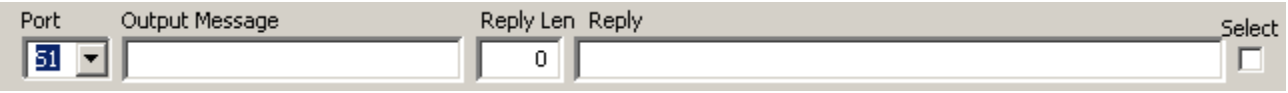



### All I2C Fast Sensor/All I2C Slow Sensors

Use either of these two buttons to quickly configure all four sensor ports as a custom I2C sensor. “Fast” sensors will use the ROBOTC firmware capability to transmit I2C messages at five times the speed found in the standard LEGO supplied firmware. Fast mode generally works with all third party sensors, slow mode is required for the LEGO ultrasonic sensor. The difference is that most 3rd party sensors include a microprocessor that has I2C messaging support in hardware and can keep up with the faster ROBOTC messaging rate. Slow sensors have a “bit-banged’ implementation and cannot keep up.

### Firmware Version

Configure these parameters for either standard LEGO firmware or ROBOTC firmware. Standard firmware does not support fast mode signaling and will always try to send a I2C message three times before reporting failures. Trying three times can easily mask intermittent transient errors. The number of total tries can be user configured in the ROBOTC firmware. The default value is a total of three tries. Setting the number of retries to zero is useful to ensure transient errors are not masked.




### Port

Selects which of the four ports on the NXT the message will be sent to.

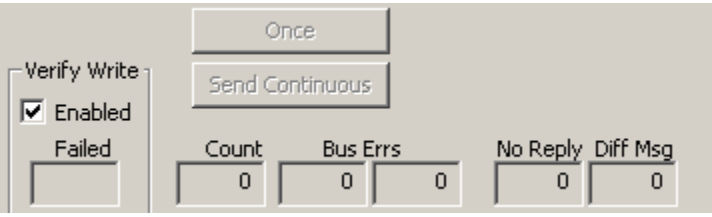
### Output Message

The hexadecimal bytes of the message to be sent to the sensor

**Reply Len (Length)**
The length of the reply expected from the sensor in bytes

**Reply**
The reply returned from the sensor in hedadecimal (and converted to ASCII)

**Select**
A checkbox to determine if the selected message should be included or excluded from the test cycle.




**Once/Continuous Buttons**
These buttons select whether a single test cycle or continuous testing should be performed.

**Verify Write**
If checked, the NXT will send an ACK message to check if the I2C message was successfully sent to the NXT. The "Failed" text box will return a count of the number of failed messages.

**Count**
A text box containing the number of messages sent in total.

**Bus Errors**
Number of bus errors encountered during error transmission. Bus error is detected by the I2C firmware and usually indicates an attempt to send an I2C message to a sensor that does not support I2C messaging.

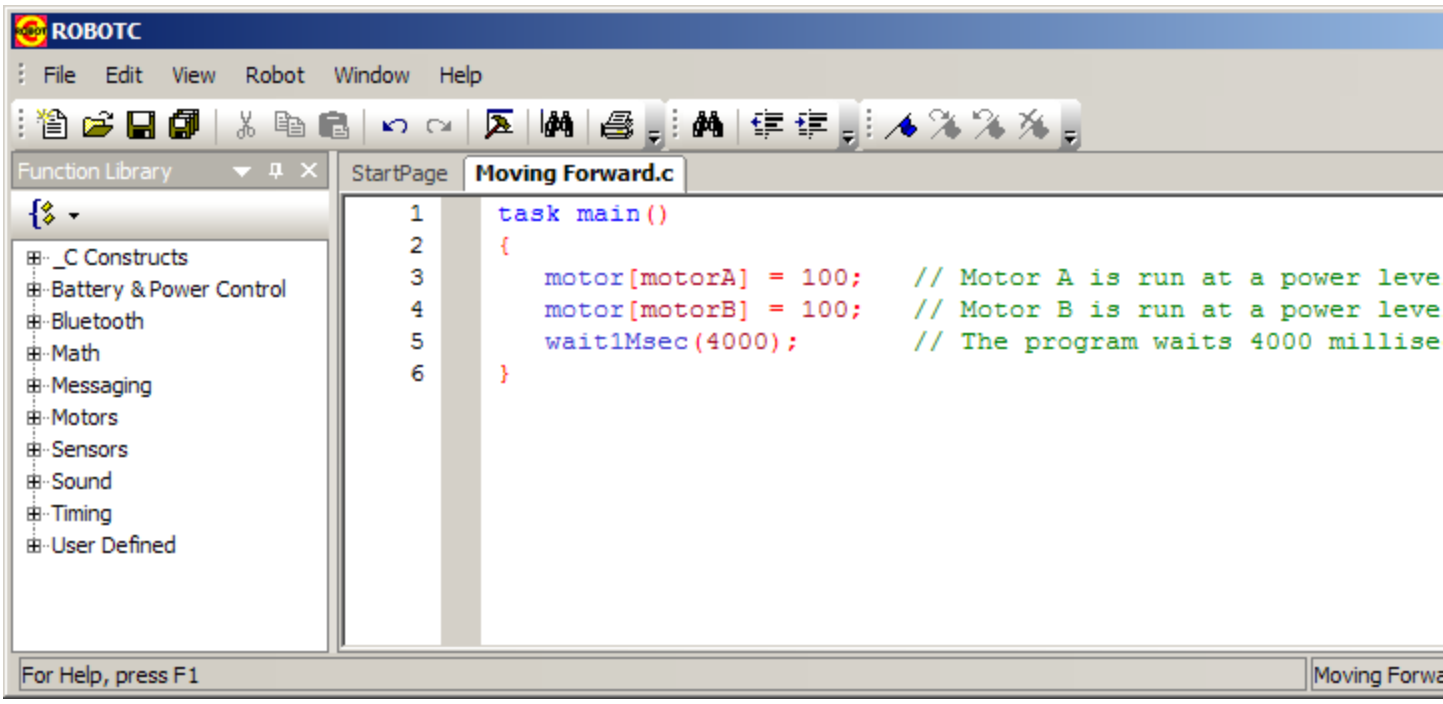
**No Reply**
Number of messages sent to the I2C message that did not recieve a reply from the sensor.

**Diff Msg (Different Message)**
Number of different messages recieved from the I2C sensor... this counter will not increment when the same message is recieved from the I2C sensor in succession.

# 4. ROBOTC Interface

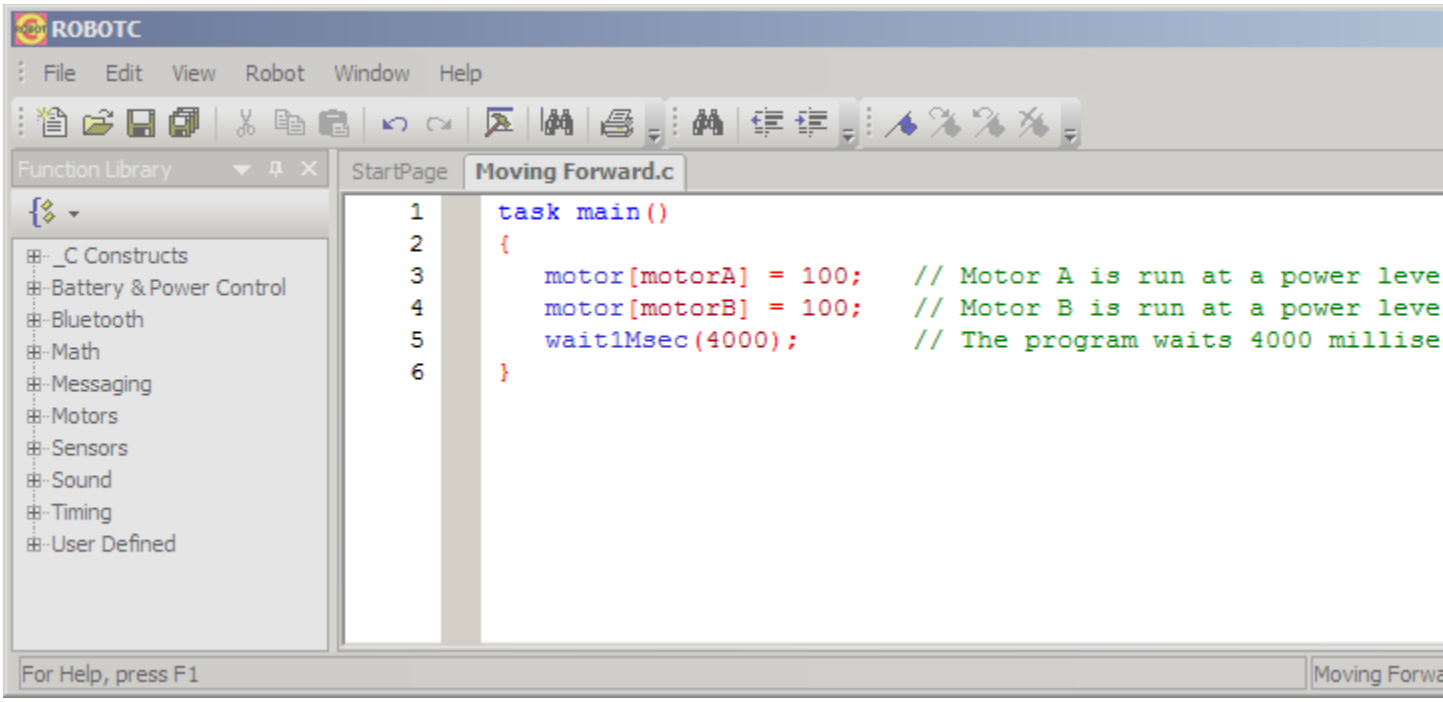
## 4.1 Overview

ROBOTC for Mindstorms is an Integrated Development Environment. It has been developed to provide as much support as possible for the platforms it is compatible with. The ROBOTC interface will modify itself to accommodate the functionality found in your controller. ROBOTC extends the 'C' programming language with a number of built-in variables and functions to provide control over a robot's hardware devices, i.e. the motors and sensors.

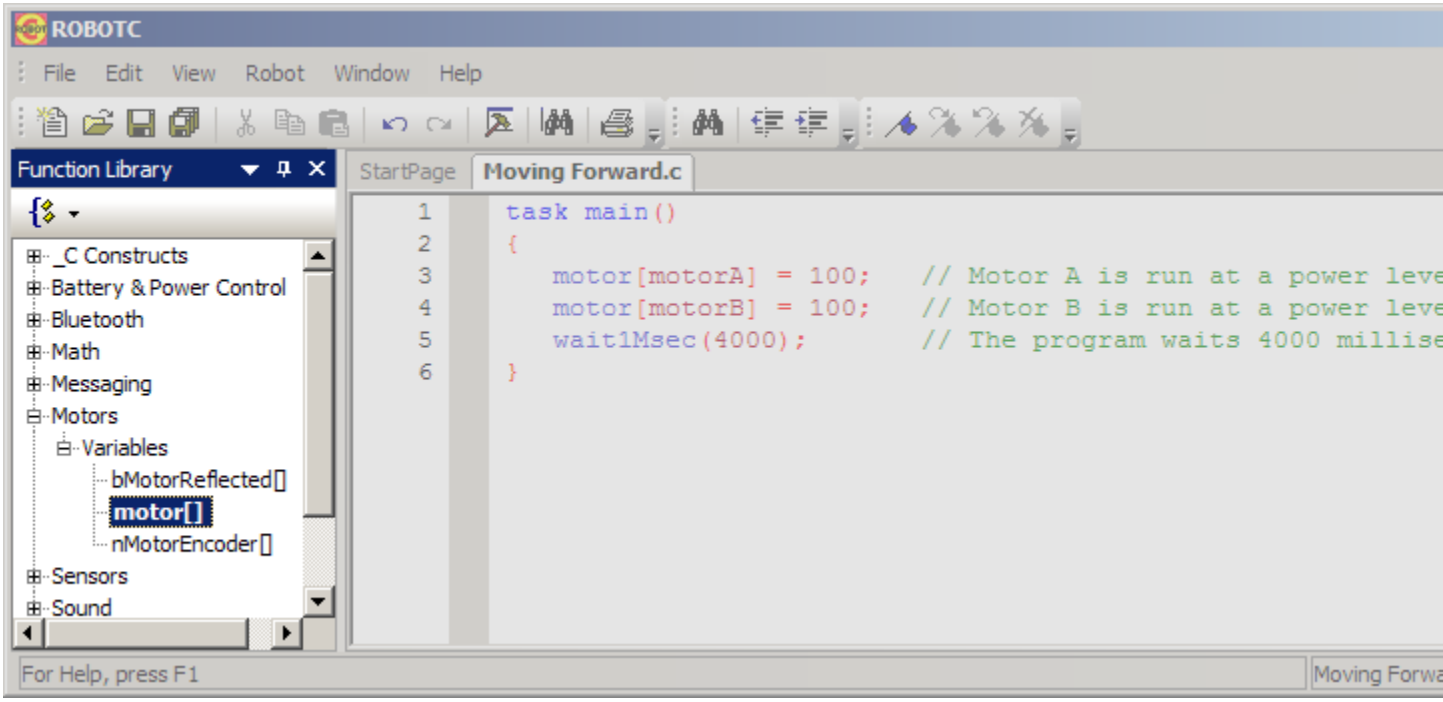



There are three main sections to the ROBOTC IDE:
1. The Editor
2. The Code Templates / Function Library
3. The Main Menu / Toolbar

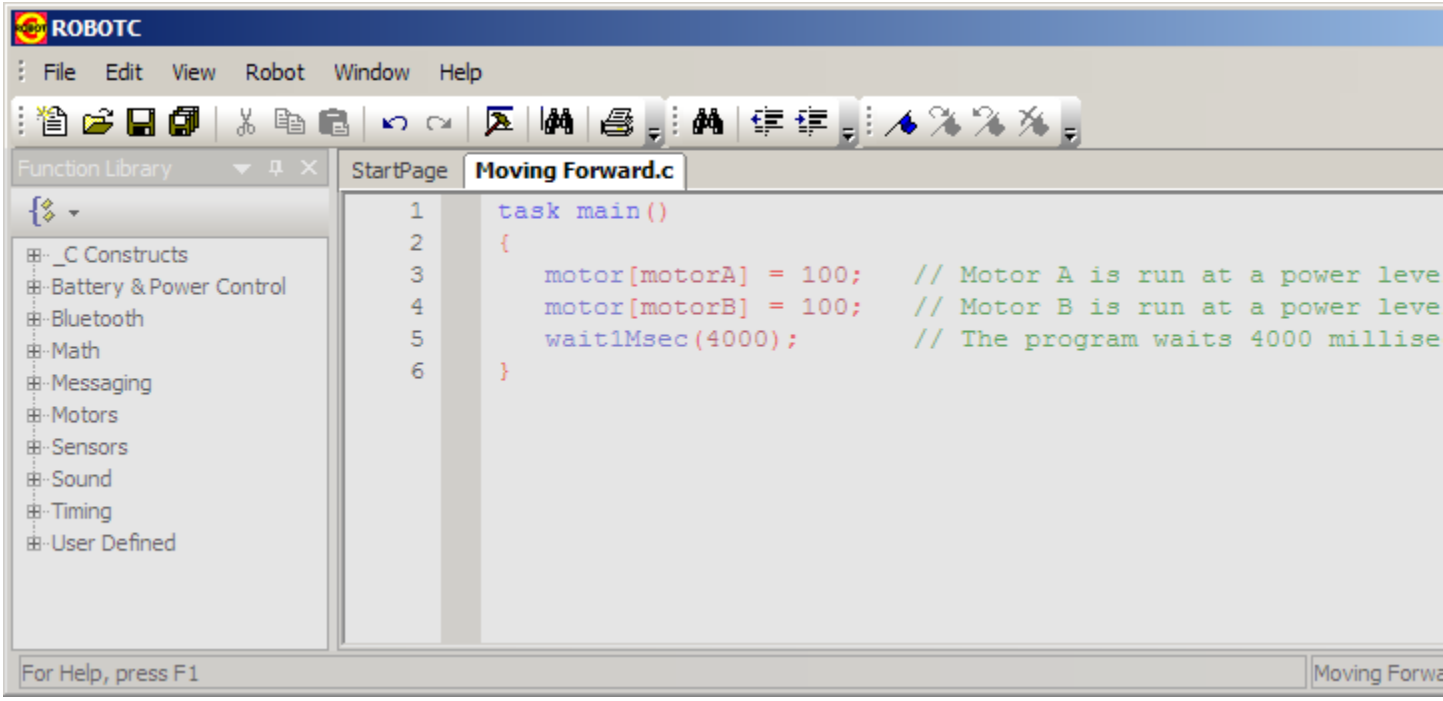
**Editor:**
This is the part of the interface where the user can write code.



**Code Templates / Function Library:**
This portion of the interface allows user to see all the functions available in ROBOTC at their user level. Portions of code from the "Function Library" can be dragged into the Editor.



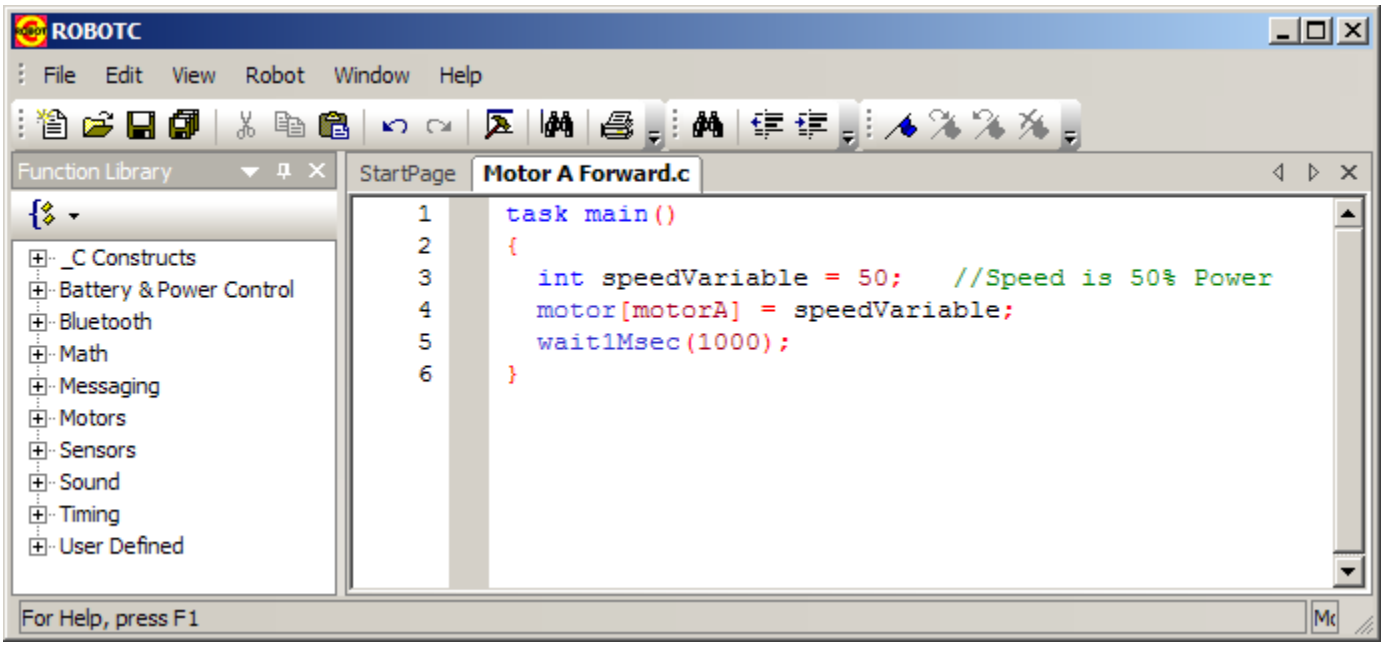

**Main Menu / Toolbar:**
This area allows you to perform basic tasks (saving, copy & pasting, undo, etc.), switch between multiple open programs using the tabbed interface, and access all of ROBOTC's additional functionality through the various menus.



# 4.2 Code Editor

## 4.2.1 Writing Code

Writing Code is the primary part of programming in ROBOTC, being a C-Based development environment. The ROBOTC Code Editor provides some visual assistance while programming with the use of line numbers and color coding.




Normal Text is displayed as **black text**. Normal text is used for variable and function names that are user-defined and not recognized by the compiler as reserved words.

Comments are displayed as **green text** . Comments are text in a program prefaced with a "**//**" or surrounded by a "**/***" and "***/** ". This "commented text" is not considered code and is ignored when the compiler generates byte-code to send to the robot controller.

Reserved Words/Pre-Defined Functions (int, motor) are displayed as **blue text** .

Constants and Parameters (50, motor1) are displayed as **dark red text** .

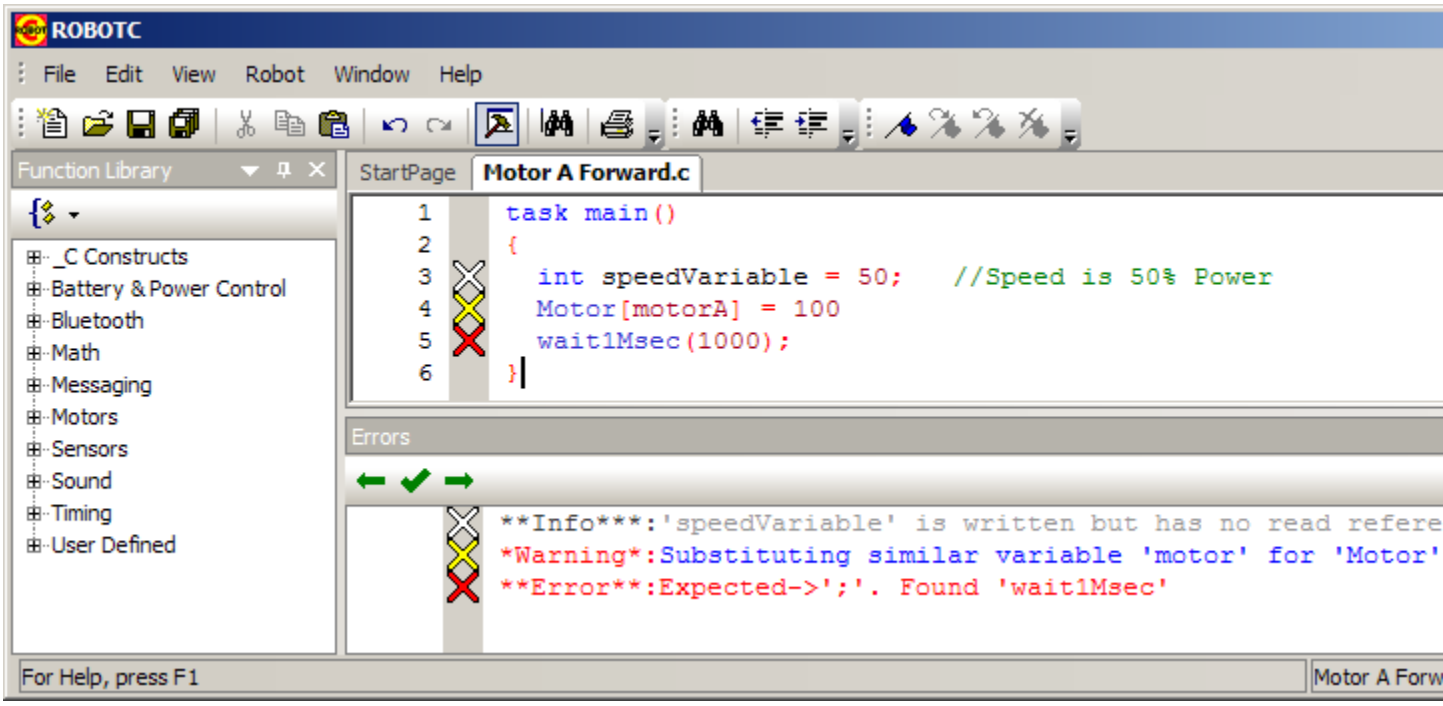
Operators (+, -, *, {, <, [, etc.) are displayed as **light red text .**

## 4.2.2 Error Display

At compilation time, the ROBOTC compiler analyzes your code to identify syntax errors, capitalization and spelling mistakes, and code inefficiency, such as unused variables and redundant code. The ROBOTC compiler also has a powerful code optimizer that can shrink your program size by up to 50% before sending it to the robot to preserve memory space on your controller.

The Error display screen reports the number of errors in your code, as well as their types. Double-clicking a compiler message in the Error display screen will highlight the relevant line of code in your program. The "Previous", "Select", and "Next" buttons can also be used to cycle through and select the errors in your program.

Depending on the type of error, ROBOTC will only be able to highlight the approximate location of the error. For instance, in the example below the missing semicolon is on line 4, but ROBOTC will highlight line 5.




ROBOTC generates three types of compiler messages: Errors, Warnings and Information Statements.

**Errors:**
ROBOTC has found an issue while compiling your program that prevented it from compiling. These are usually misspelled words, missing semicolons, and improper syntax. Errors are denoted with a **Red X**.

**Warnings:**
ROBOTC has found a minor issue with your program, but the compiler was able to fix or ignore it to get your code to compile. These are usually incorrect capitalizations of words or infinite loops without any code inside. Warnings are denoted with a **Yellow X**.

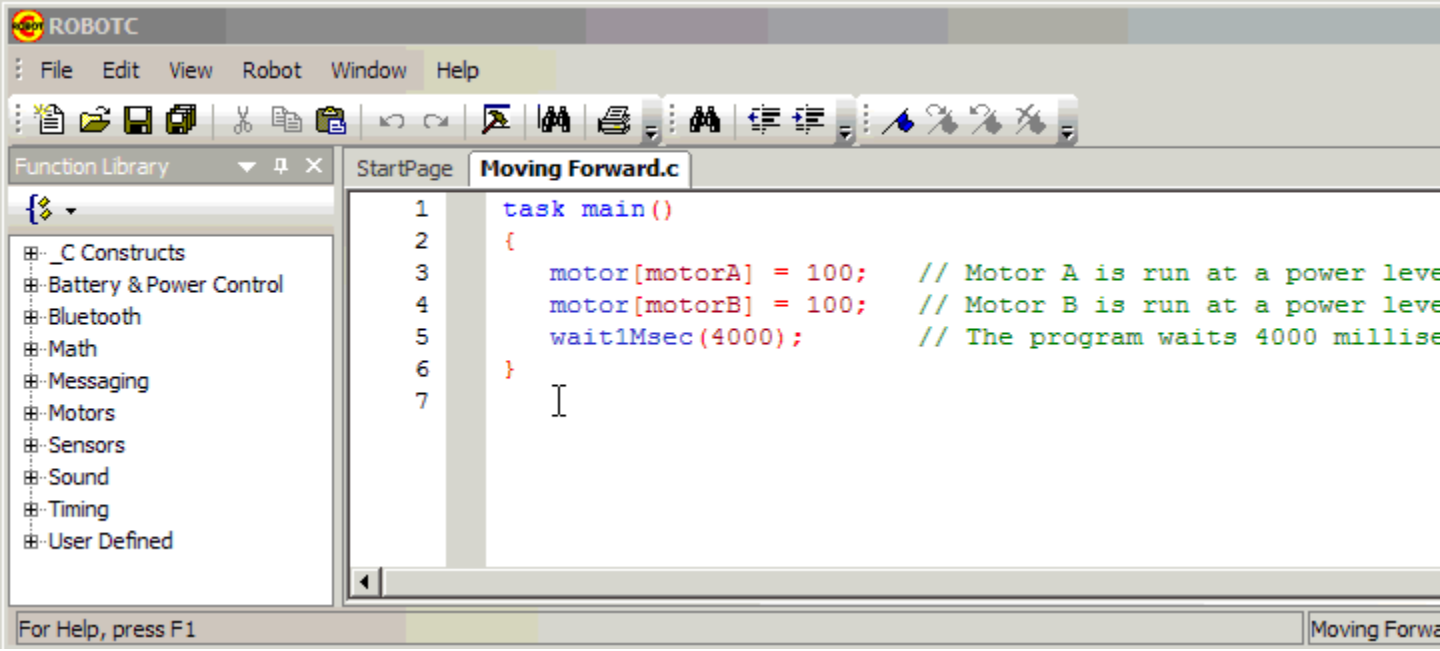
**Information:**
ROBOTC will generate information messages when it thinks you have declared functions or variables that are not used in your program. These messages have no affect on your program and only serve the purpose of informing you about inefficient programming. Information messages are denoted with a **White X** .

## 4.2.3 Breakpoints

Breakpoints are a useful debugging tool in ROBOTC. Breakpoints can be placed on any line of code and tell the controller to stop executing the program when that line is reached. When a breakpoint is reached, the ROBOTC debugger informs the programmer, who can then check the status of the robot, code editor, and debugger windows at that point. ROBOTC supports multiple break points.

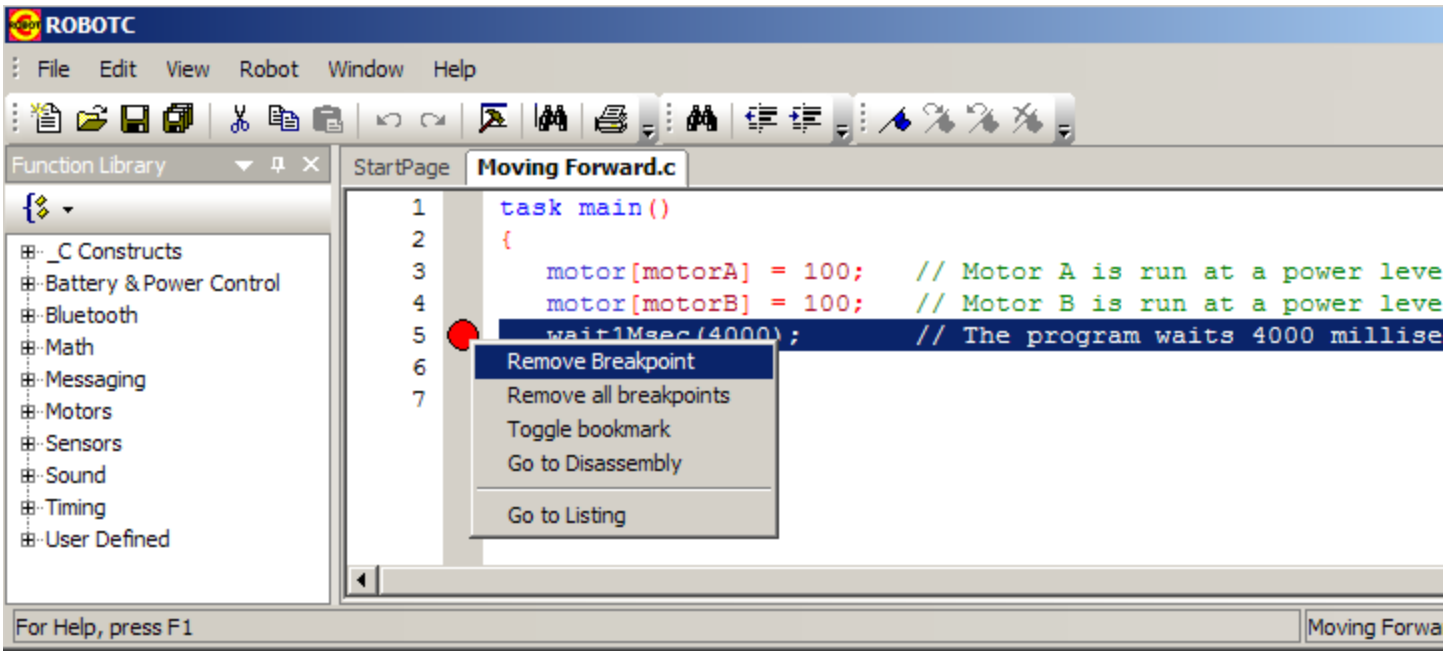
Breakpoints are denoted by a red circle in the gray area between the code and the line numbers.

**To add a breakpoint:**
Right-click on the line number where you wish to add a break point and a context menu will appear. Select "Insert Breakpoint" to place a breakpoint at that line.

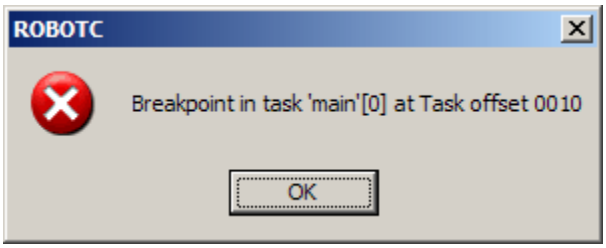



*Animation: Adding a Breakpoint*

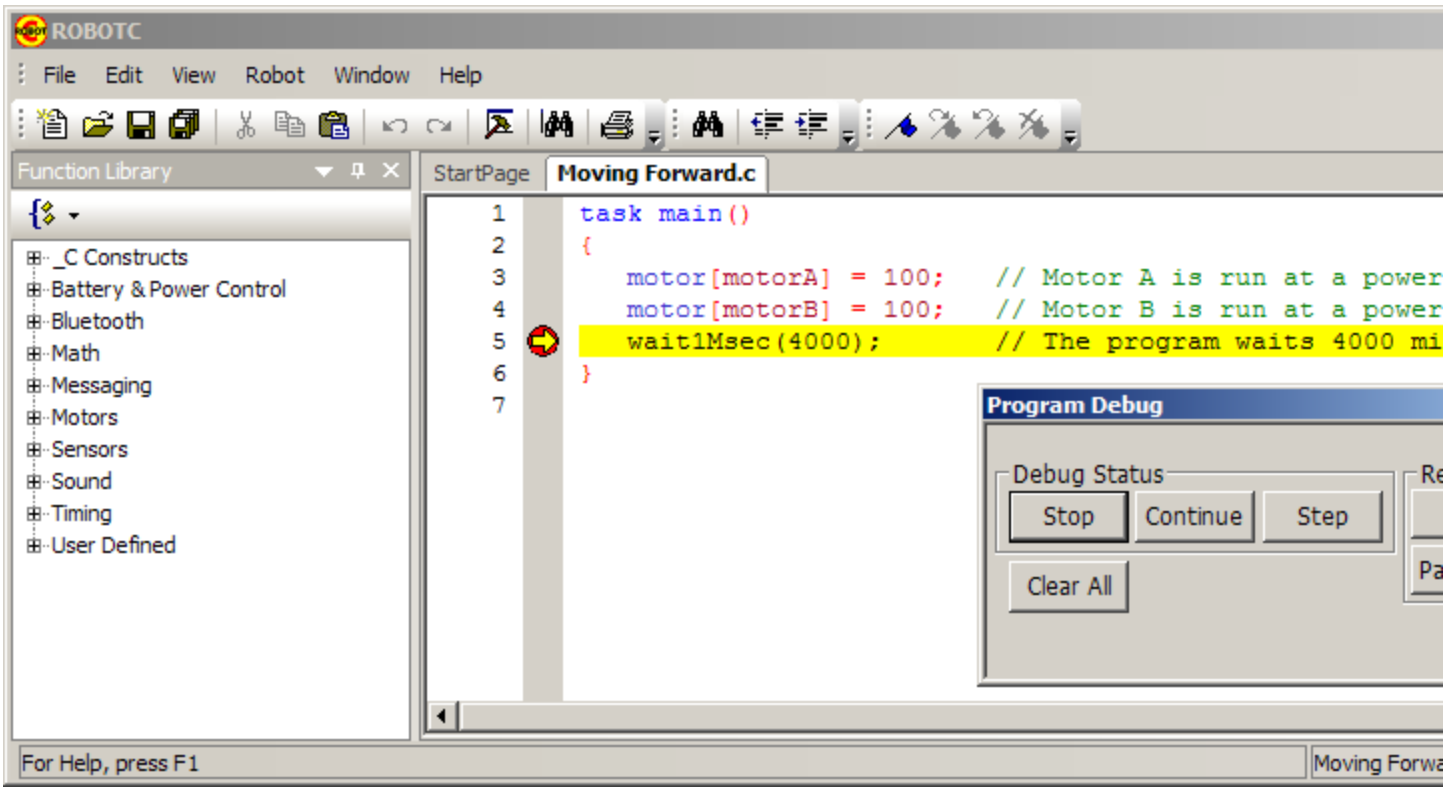
**To remove a breakpoint:**
Right click on the red breakpoint circle and select "Remove Breakpoint".




**Running your program with breakpoints:**
When a breakpoint reached in a program while the debugger is open, you will see the following message from ROBOTC:



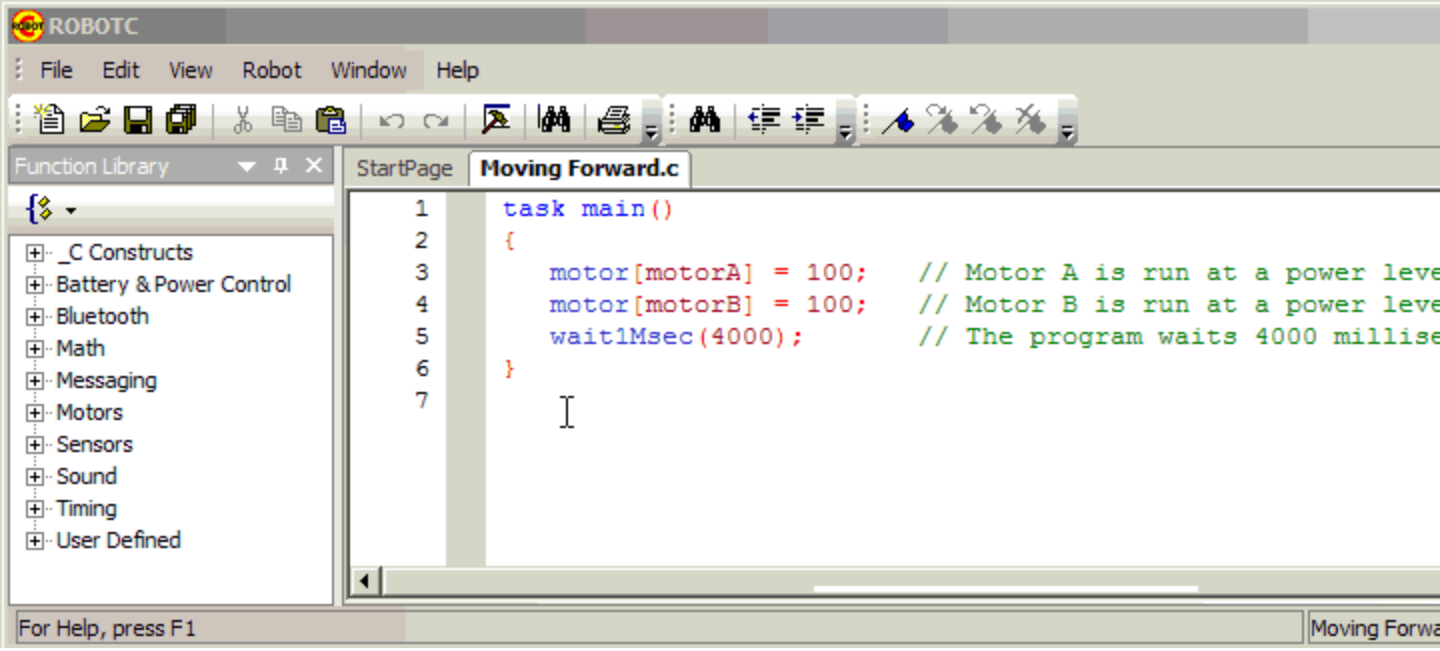

This message is just an indication that the breakpoint has been reached. Click OK and then look at your program in the code editor. ROBOTC highlights the line of the breakpoint that was reached, allowing you to see exactly which breakpoint the program execution stopped at (assuming you have multiple breakpoints).




## 4.2.4 Bookmarks

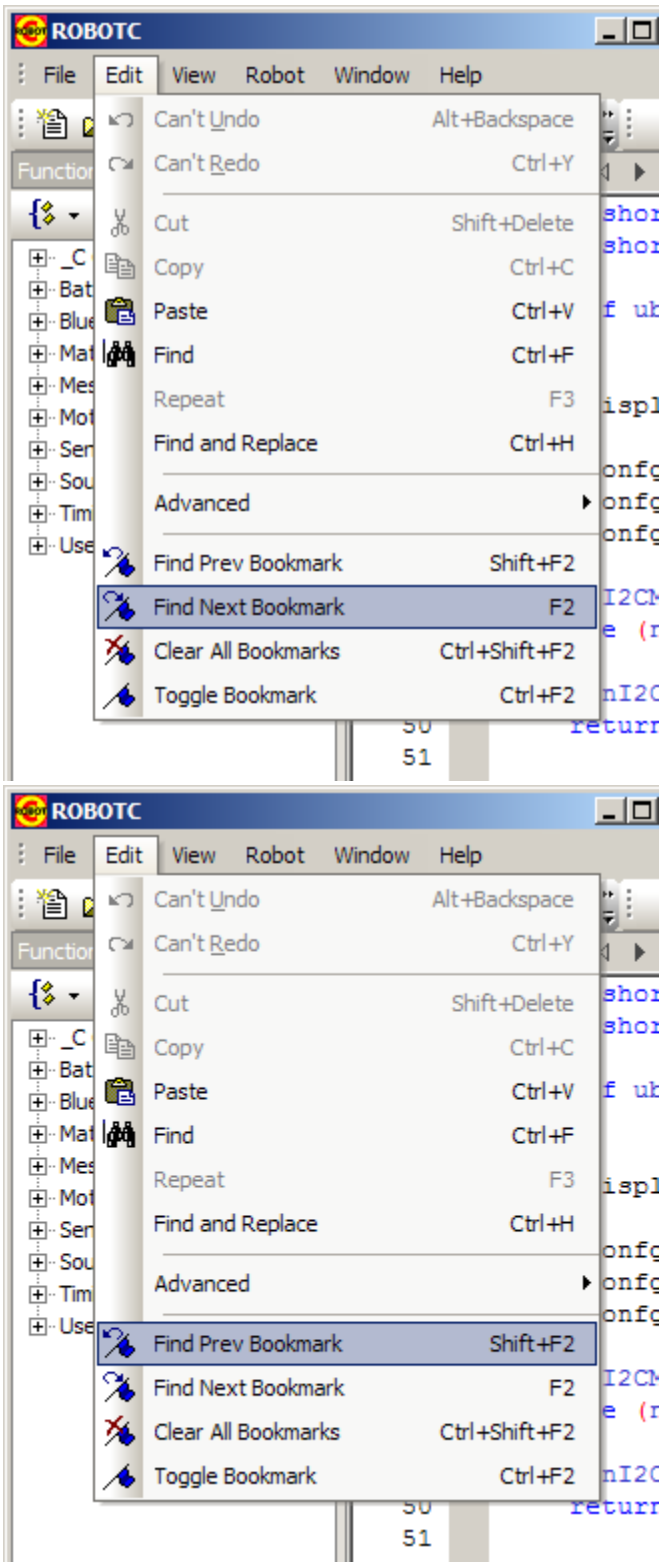
Bookmarks are useful for navigating larger ROBOTC programs. You can use bookmarks to jump from one line of code to the next.

**To Add/Remove a Bookmark:** (Ctrl-F2)
Right click on the line number that you wish to add or remove a bookmark and a context menu will appear. Select "Toggle Bookmark" to place or remove a bookmark at that line. Pressing Ctrl-F2 on your keyboard will also add or remove a bookmark on the line with the cursor.




*Animation: Adding a  Bookmark*

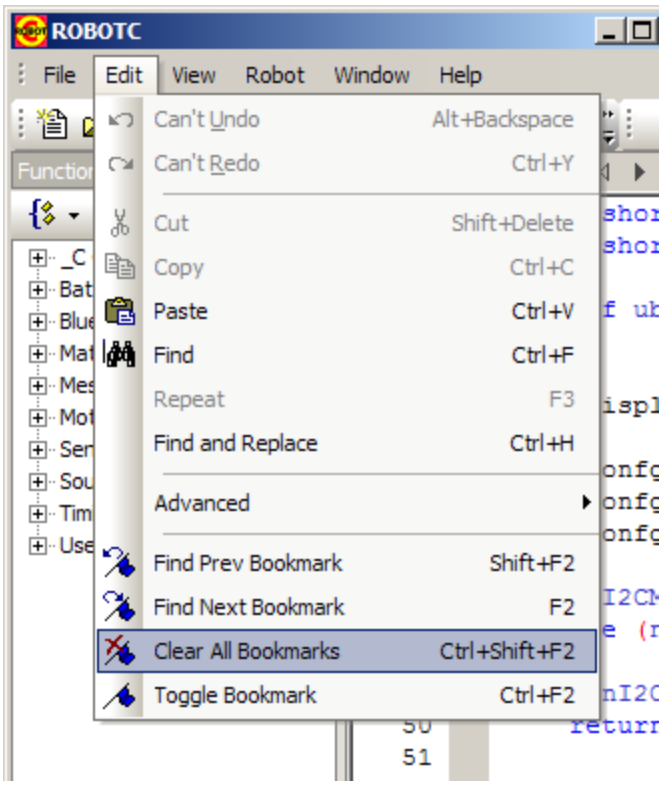
**To Use Bookmarks:** (F2) & (Shift + F2)
You can jump to the next line of code that has a bookmark by pressing the (F2) key on your keyboard, or by going to the "Edit" menu and selecting "Find Next Bookmark." You can also go to a previous bookmark by pressing (Shift + F2) , or going to the "Edit" menu and selecting "Find Prev Bookmark."



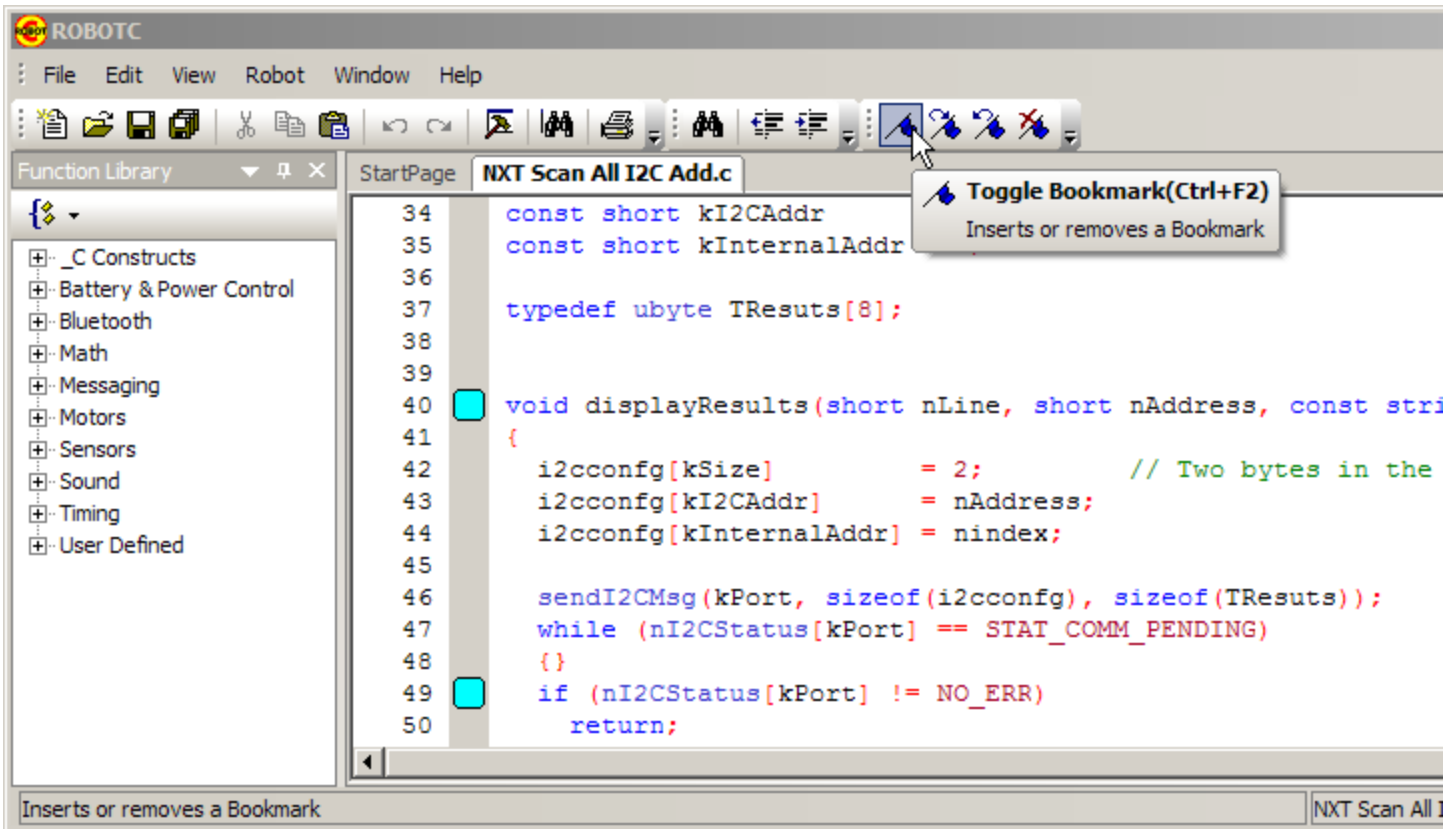




**Clear all bookmarks: (Ctrl + Shift + F2)**
You can clear all bookmarks by going to the "Edit" menu and clicking "Clear All Bookmarks". You can also use the keyboard shortcut (Ctrl + Shift + F2) to clear all bookmarks as well.

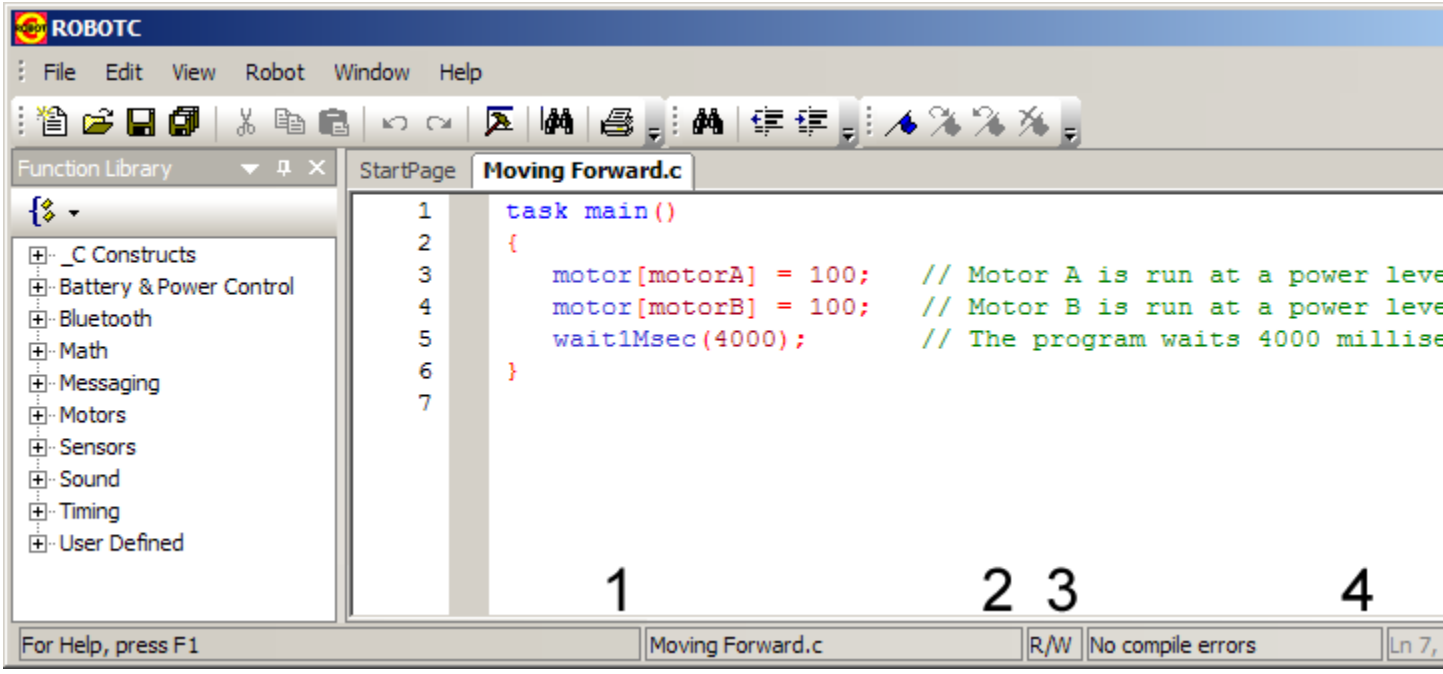



**Bookmarks Toolbar:**
If you have the Bookmarks toolbar enabled, you can use it to toggle, switch between, and clear bookmarks as well. To enable the Bookmarks toolbar, right-click on your toolbar and select "Bookmark".




# 4.3 Status Bar

The Status Bar gives you information about the ROBOTC editor and the current program.



**1.** Context Help - Hover over different sections of the ROBOTC interface for brief descriptions.

**2.** Active Project - Displays the most recently compiled program.

**3.** Read-only Status - Displays the file permissions for the most recently compiled program.
R/W - Read and Write Access
R/O - Read Only Access

**4.** Compiler Errors - Displays the number of Warnings and Errors in a program, if any.

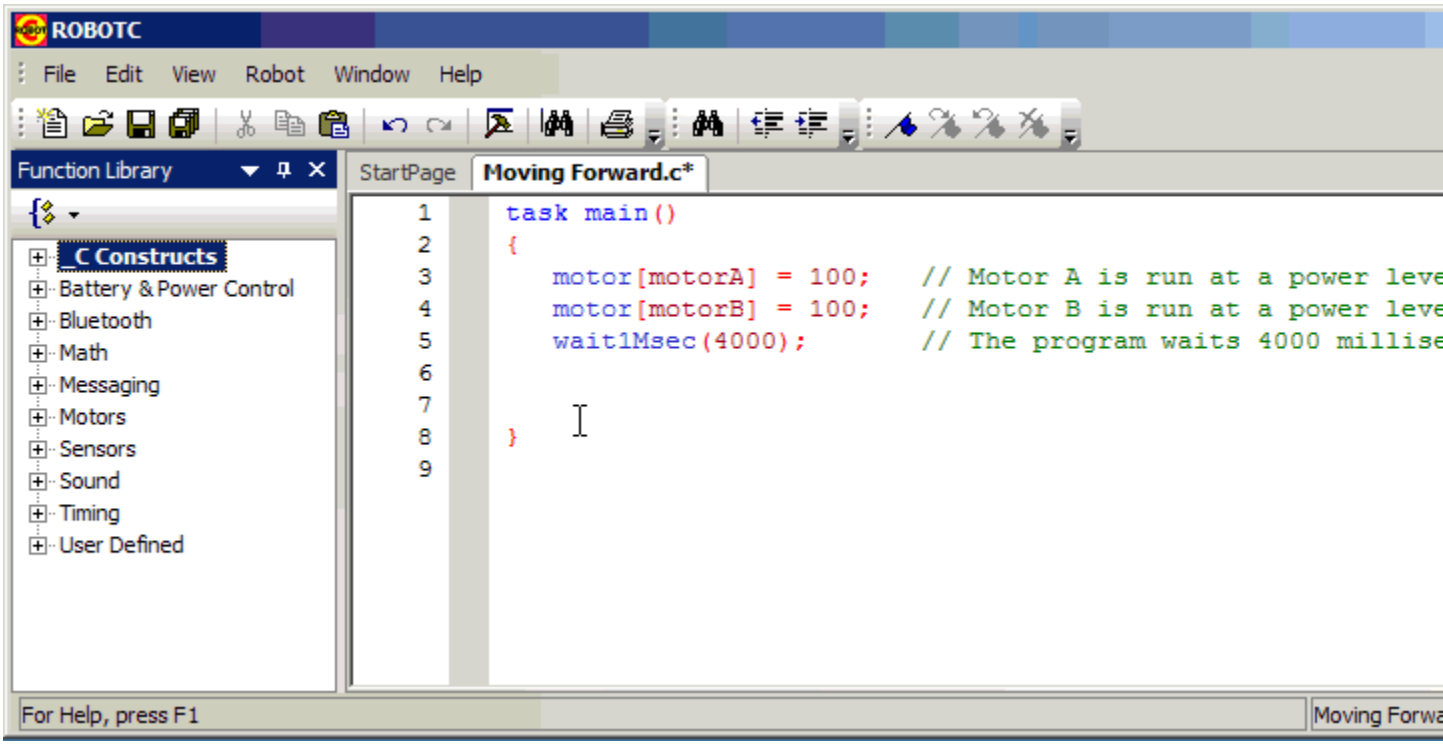
**5**. Line/Column Display - Shows the line number and character of where the text cursor is currently located.

# 4.4 Function Library

The Function Library panel provides three convenient features:

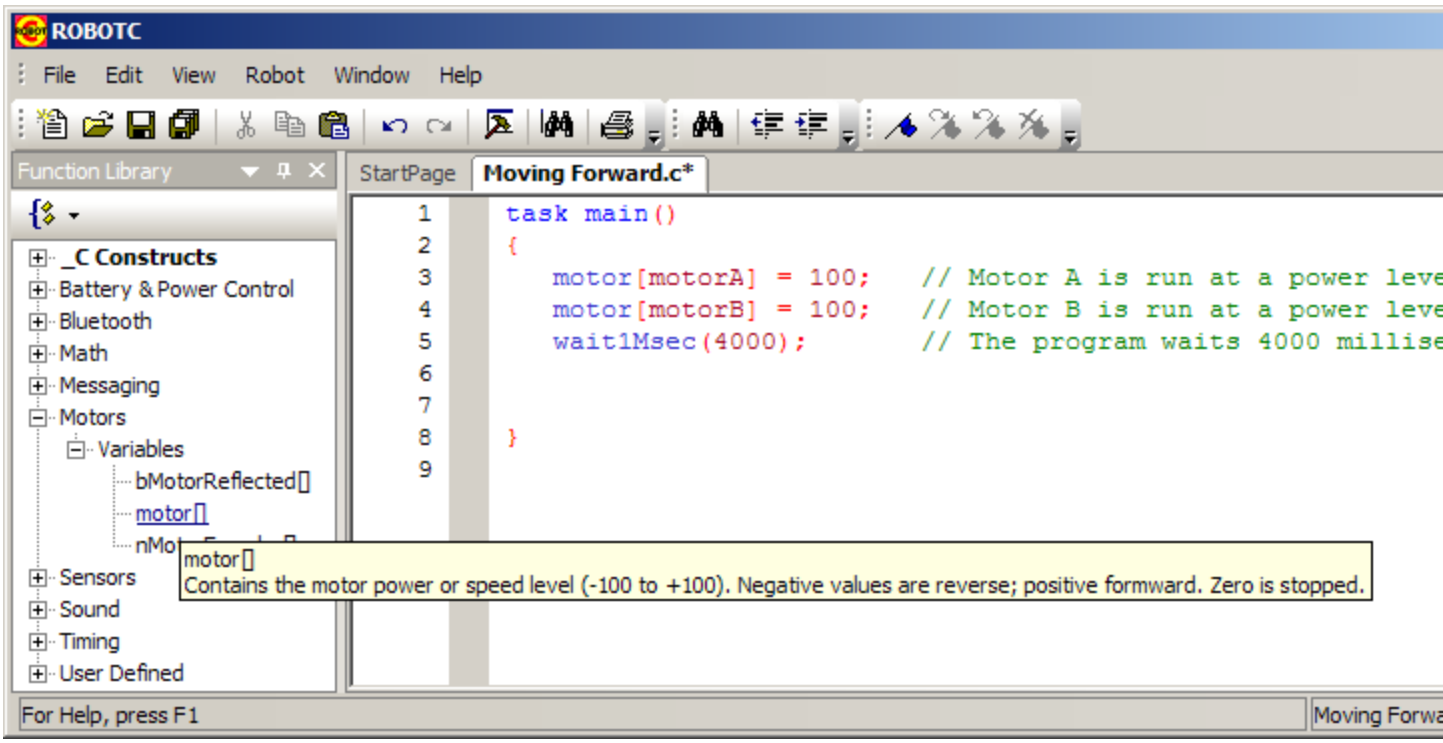
**1.** The Function Library panel provides a listing of all the platform-specific functions available in ROBOTC at your user level, along with any compiled user-defined functions.

**2.** You can drag items from the panel into your program.




*Animation: Dragging a function*

**3.** If you hover over an item, a 'tool-tip' popup will display a brief description of the item.



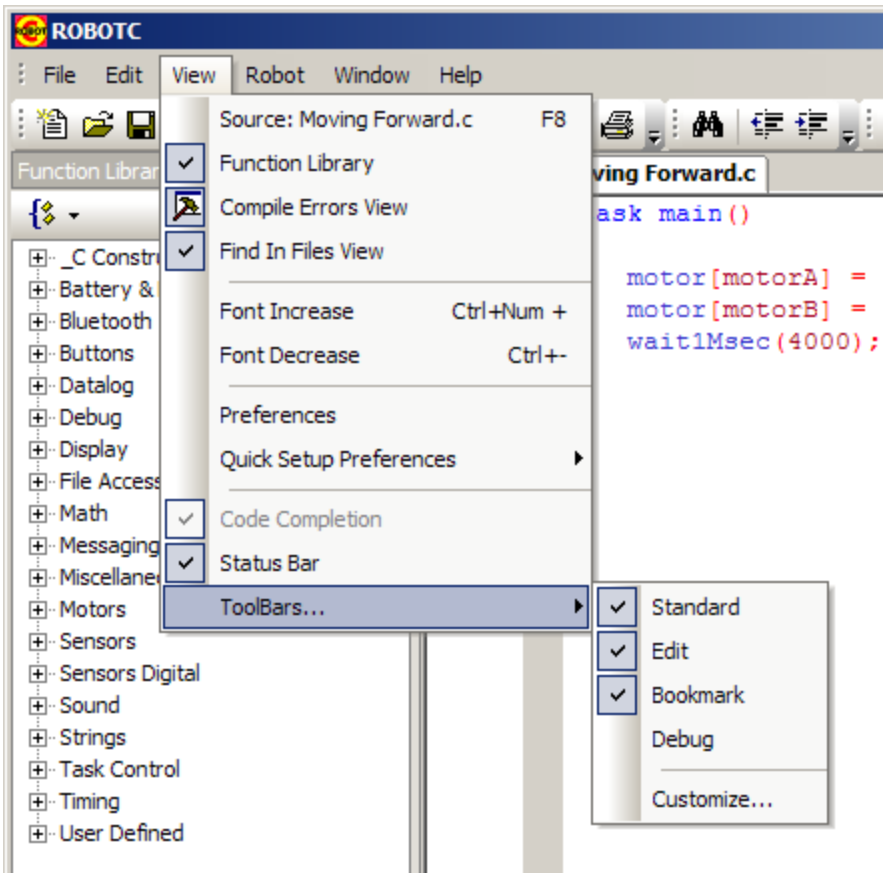

# 4.5 Toolbar

The Toolbar provides a convenient way to access frequently used commands in ROBOTC.

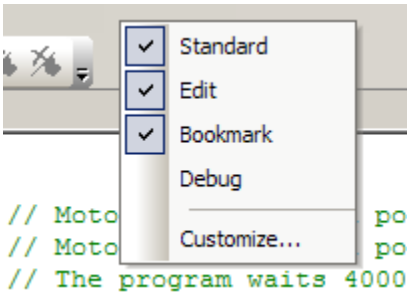
| | | | |
|---|---|---|---|
|  | **New (Ctrl - N)** - Creates a new blank program. |  | **Open (Ctrl - O)** - Opens an existing program. |
|  | **Save (Ctrl - S)** - Saves your current program. Your program is automatically saved when your code is compiled. |  | **Save All (Ctrl - Shift - S)** - Saves all open programs. |
|  | **Cut (Ctrl - X)** - Removes highlighted code and places it on the Windows Clipboard. |  | **Copy (Ctrl - C)** - Creates a copy of the currently highlighted code on the Windows Clipboard. |
|  | **Paste (Ctrl - V) -** Places a copy the contents of the Windows Clipboard onto the code editor window at the current cursor position. |  | **Print (Ctrl - P) -** Prints out the code currently in the code editor window. |
|  | **Undo (Ctrl - Z) -** Reverts the last action taken in the code editor window. Undoes are discarded when the code is saved. |  | **Redo (Ctrl - Y) -** The opposite of Undo. Only available if an Undo command was recently used. |

| | | | |
|---|---|---|---|
|  | **Errors** - Shows or hides the compiler error output. | | |
|  | **Find (Ctrl - F)** - Finds the specified text. |  | **Find All (Ctrl - Shift - F)** - Finds all occurances of the specified text in all source files. |
|  | **Decrease Indent** - Indents the selection text left one tab stop |  | **Increase Indent** - Indents the selection text right one tab stop |
|  | **Format Whole File (Ctrl - Shift - Alt - F)** - Reformats the entire file, aligning all indents and braces. |  | **Toggle Comment (Ctrl - Q)** - This will either comment or un-comment the selected lines of code. |
|  | **Comment Line(s) (Ctrl - Shift - Q)** - This will add "//" to the selected lines, commenting them out. |  | **Un-Comment Line(s) (Ctrl - Alt - Q)** - Removes "//" from the selected lines, un-commenting them. |
|  | **Toggle Bookmark (Ctrl - F2)** - Inserts or removes a bookmark. |  | **Clear Bookmarks (Ctrl - Shift - F2)** - Clears all bookmarks in the current file. |
|  | **Next Bookmark (F2)** - Goes to the next bookmark. |  | **Previous Bookmark (Shift - F2)** - Goes to the previous bookmark. |
|  | **Start** - Start the execution of a program. |  | **Stop** - Stop the execution of a program. |
|  | **Resume** - Resume the execution of a program. |  | **Suspend** - Suspend the execution of a program. |
|  | **Step In** - Step the debugger into the current statement. |  | **Step Over** - Step the debugger over the current statement. |
|  | **Step Out** - Step the debugger out of the current function. |  | **Step** - Step the debugger one statement. |

You can configure which sections of the Toolbar are visible by going to the "View" menu and selecting "Toolbars...":

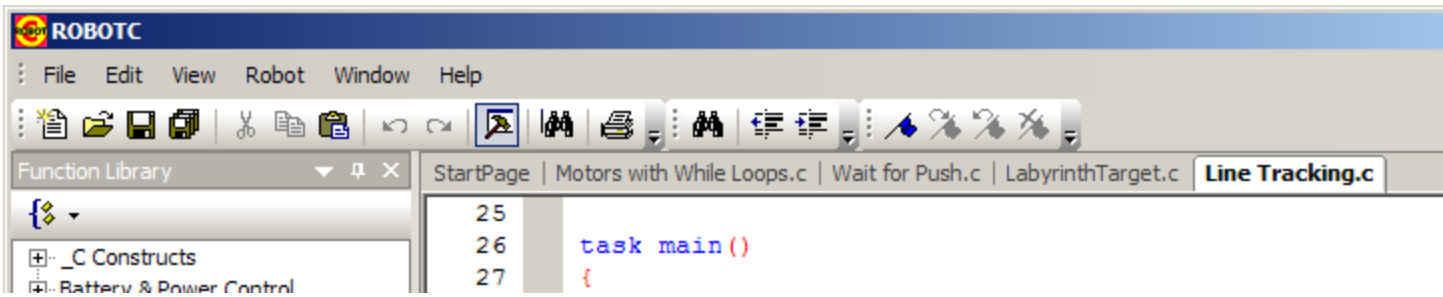



Or by right-clicking on the Toolbar:

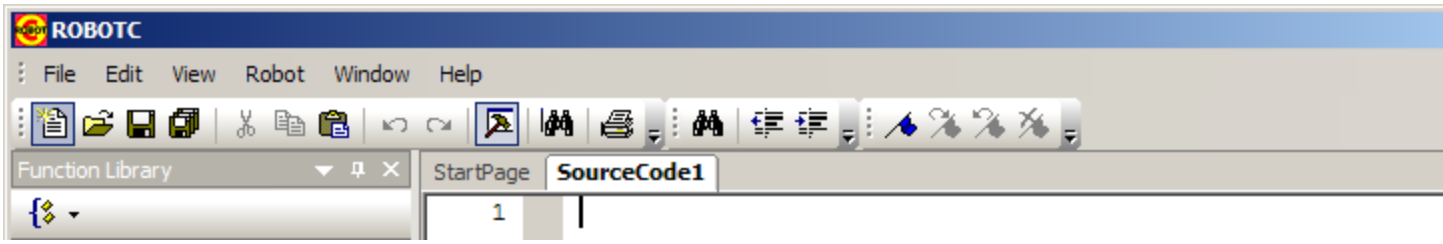



# 4.6 Tabbed Interface

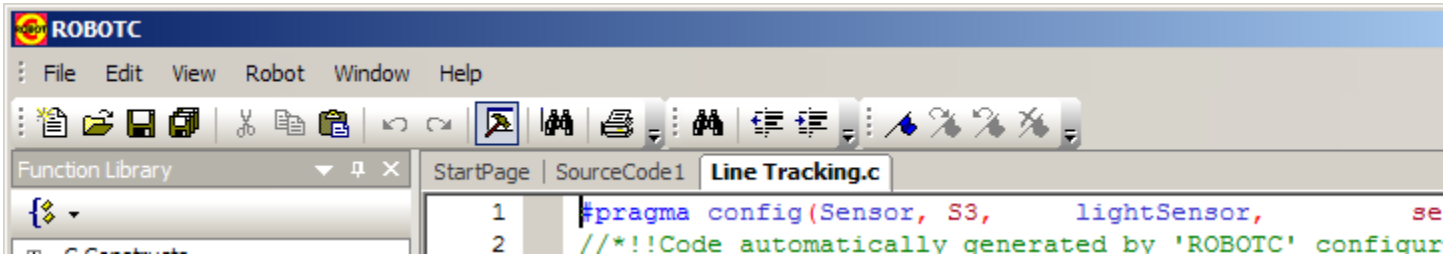
ROBOTC allows you to have multiple programs open at once, using a **Tabbed Interface**.



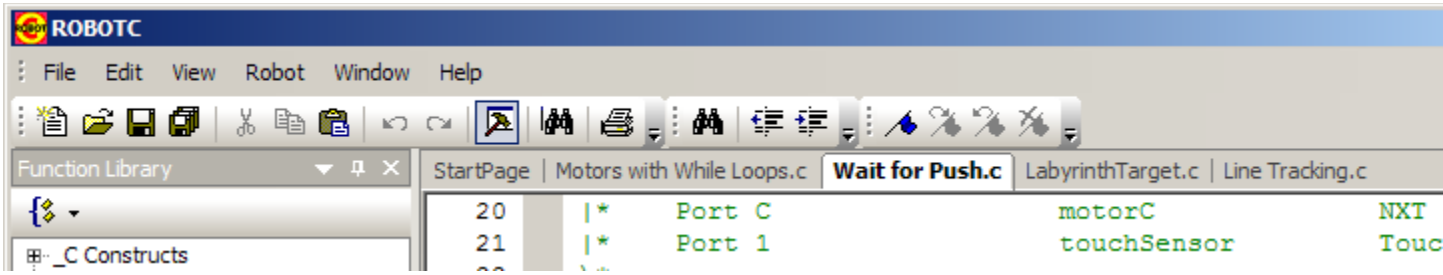

Starting a new program will generate a new tab with a blank code editor.



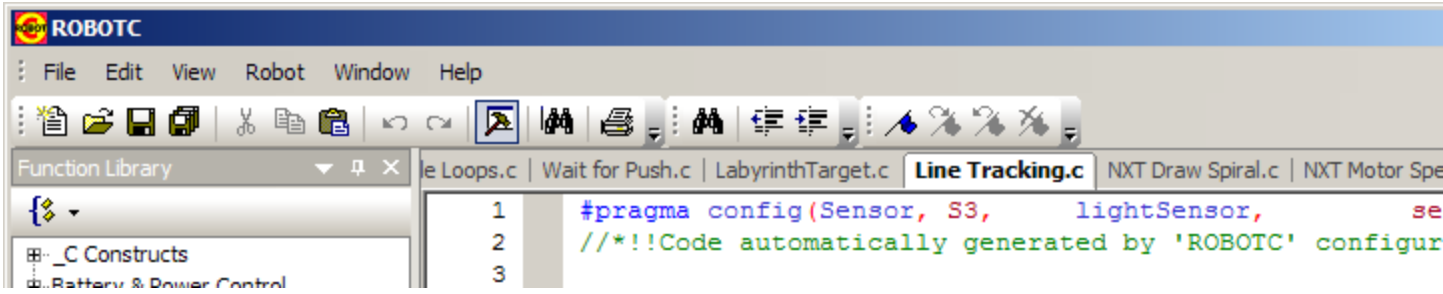

Opening an existing program will cause it to open in a new tab. The text in the tab displays the file name for the program.



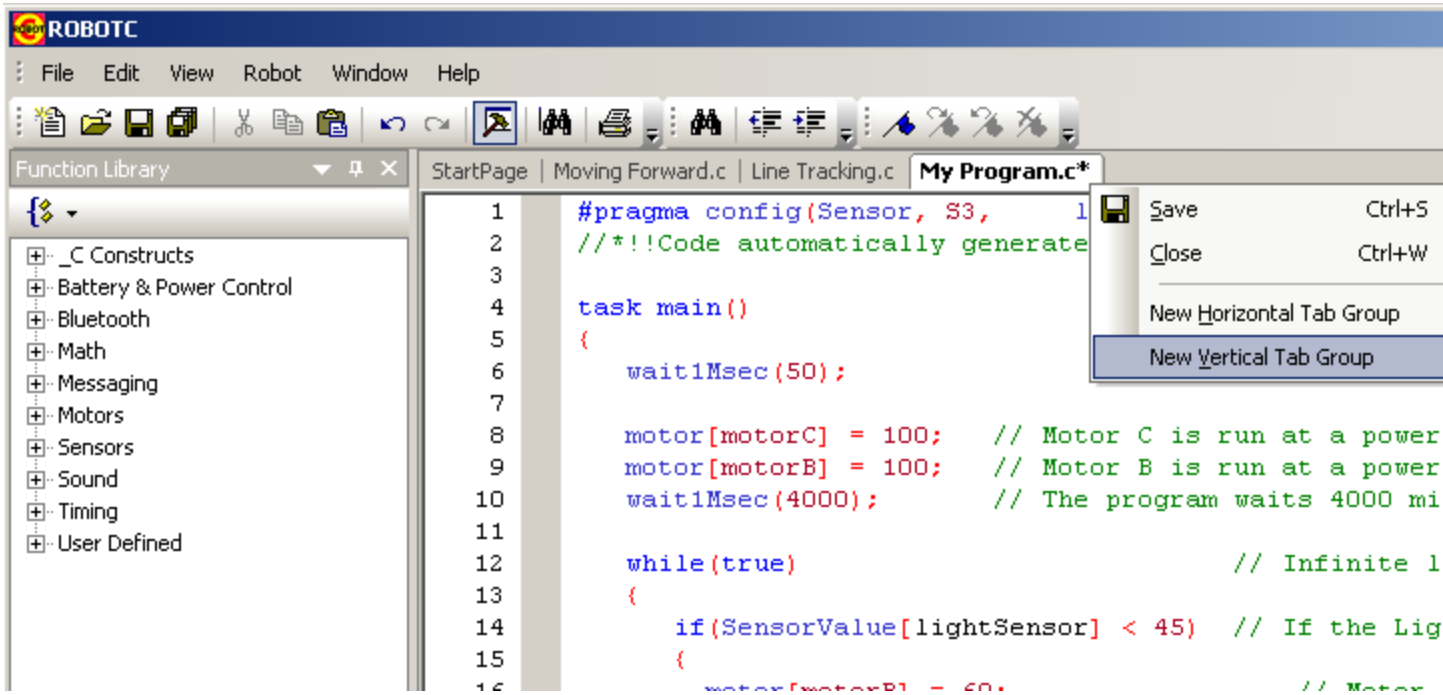

The program currently viewable in the code editor has a highlighted tab (Wait for Push.c in the example below). To select a different program, click on the name of the file in the tab interface. Tabs can be re-ordered by clicking on them and dragging them to the left or right.



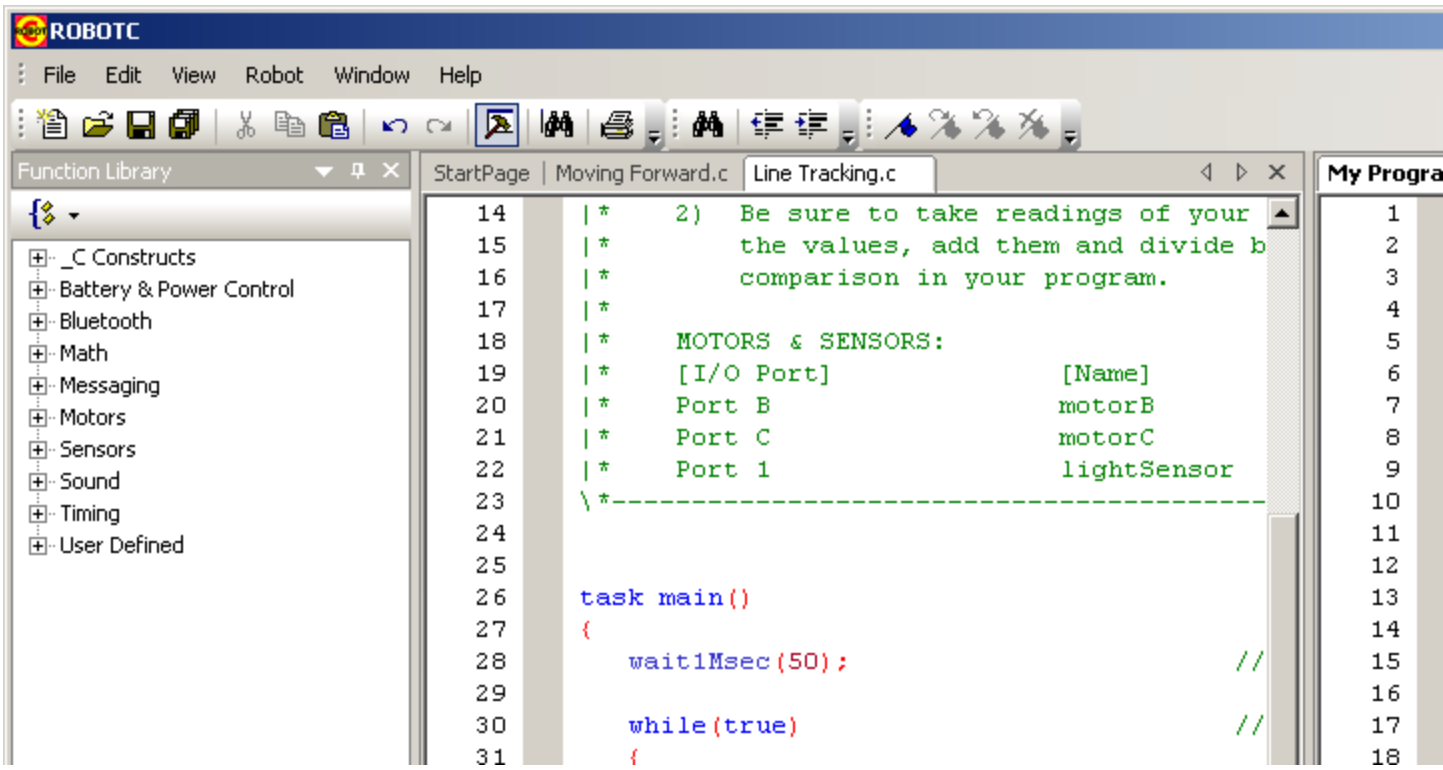

If the number of tabs you have open fills up the horizontal space, the "Scroll Left" and "Scroll Right" buttons will darken, allowing you to scroll through your open programs. Clicking the "X" next to the scroll buttons will close the program at the forefront (Line Tracking.c in the example below).




With the ROBOTC tabbed interface, you're able to easily reference other programs and copy-and-paste code between multiple files. The "Tab Group" feature makes it even easier. Right-click on a tab, and select "New Vertical Tab Group" or "New Horizontal Tab Group" for a side by side code display. See below for result.




The example below is a Vertical Tab Group. The program at the forefront of the code editor has its file name in bold (My Program.c in the example below). Compiling, Downloading, Saving, and other ROBOTC options will be applied to whatever program is at the forefront of the code editor. To restore the original tabbed view, right-click on the tab and select "Move to Previous Tab Group".



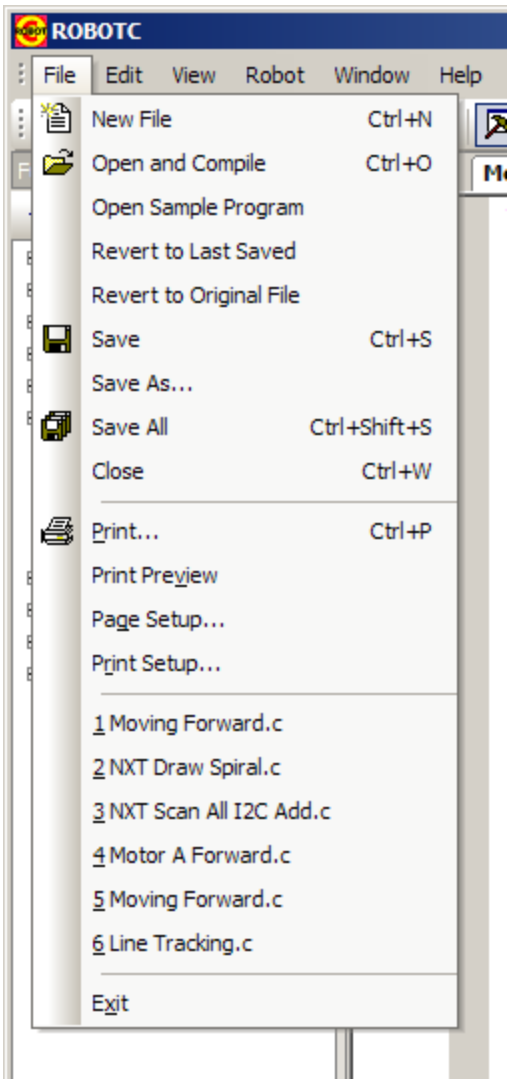

Additional Notes:

- Pressing the "Save All" button on the Toolbar will save all of the programs opened in tabs.
- Downloading a program to your robot controller will download the program at the forefront. Switching between tabs will not close the debugger, but the debugger data will still only pertain to the downloaded program.
- The Status Bar at the bottom of the ROBOTC interface shows the last compiled program, not necessarily the program in the code editor.

# 4.7 Menus

## 4.7.1 File

The **File** menu is used for starting new programs, opening existing programs, saving programs, printing, and accessing recently opened programs.




**New File**
Creates a blank document in a new tab.

**Open and Compile**
Launches a select file dialog box. User selected file is opened and compiled in a new tab.

**Open Sample Program**
Automatically opens the "Sample Programs" folder for the robot platform you have selected. User selected file is opened and compiled in a new tab.

**Revert to Last Saved**
Will undo any changes to the program since last time the program has been saved. You may need to recompile your program after selecting this action.

**Revert to Original File**
Will undo any changes from the time the program was opened. You may need to recompile your program after selecting this action.

**Save**
Saves the current program to disk.

**Save As...**
Saves the current program to disk with a different name.

**Save All**
Saves all open programs.

**Close**
Closes the document at the front.

**Print...**
Prints the current program.

**Print Preview**
Shows how the current program would look when printed.

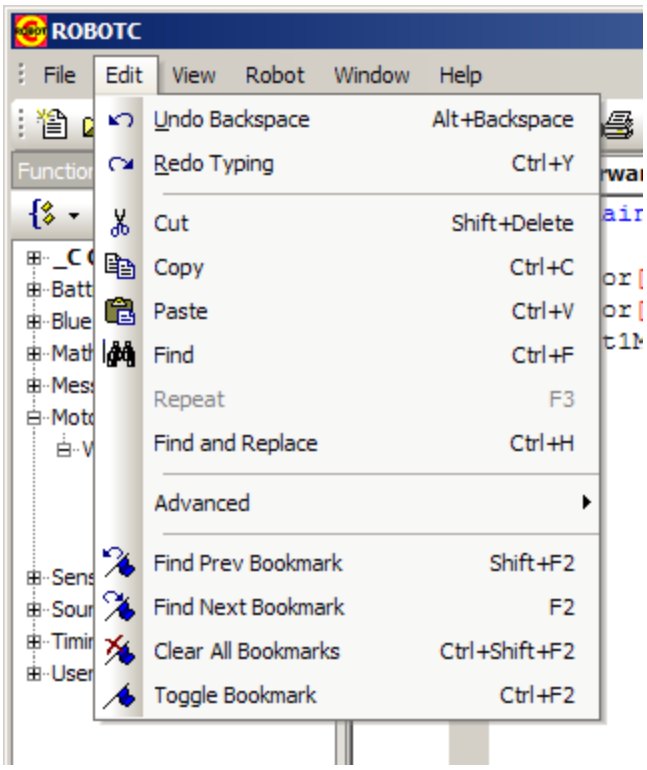
**Page Setup...**
Accesses the Windows dialogs for the printer's settings for page orientation.

**Print Setup...**
Accesses the Windows dialogs for Printer configuration.

**Last Accessed Files:**
ROBOTC keeps a history of the current file along with the previous 5 programs opened. You can open one of these files in a new tab by clicking on the name.

## 4.7.2 Edit

The **Edit** menu contains helpful tools for writing programs. Functionality like Undo, Redo, Copy, Paste, Find, and Bookmarks can be found in the Edit menu.




**Undo "Command"**
Reverts the last action taken in the code editor window. Undoes are discarded when a program is saved.

**Redo "Command"**
The opposite of Undo. Only available if an Undo command was previously used.

**Cut**
Removes highlighted code and places it on the Windows Clipboard.

**Copy**
Creates a copy of the currently highlighted code on the Windows Clipboard.

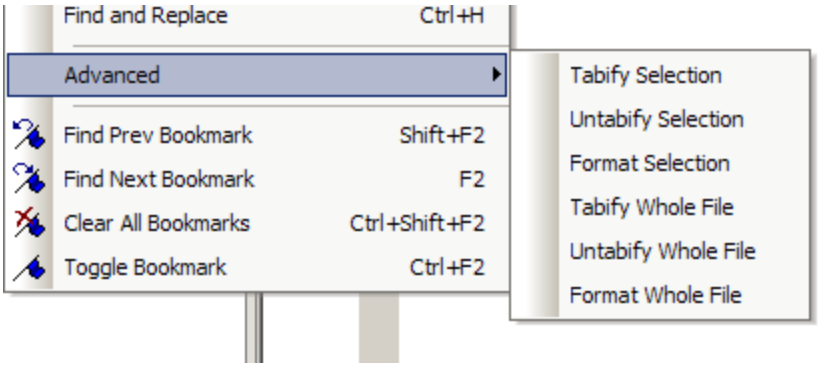
**Paste**
Places a copy the contents of the Windows Clipboard onto the code editor at the current cursor position.

**Find**
Opens a dialog to search for a string of text/code in a program.

**Repeat**
Used in conjunction with the Find command; Searches for the next occurrence of the search string in your program.

**Find and Replace**
Similar to find, but used to find strings of code and overwrite with other strings of code. Useful for making multiple changes at once.

**Advanced:**




**Tabify Selection**
Converts selected text's equivalent spaces to tabs.

**Untabify Selection**
Converts selected text's equivalent tabs to spaces.

**Format Selection**
Converts the selected text and corrects the indentations of the text.

**Tabify Whole File**
Converts program's equivalent spaces to tabs.

**Untabify Whole File**
Converts program's equivalent tabs to spaces.

**Format Whole File**
Converts the program and corrects the indentations of the text.

**Find Prev Bookmark**
Moves the text cursor jump to the previous bookmark. See more information here.

**Find Next Bookmark**
Moves the text cursor jump to the next bookmark. See more information here.

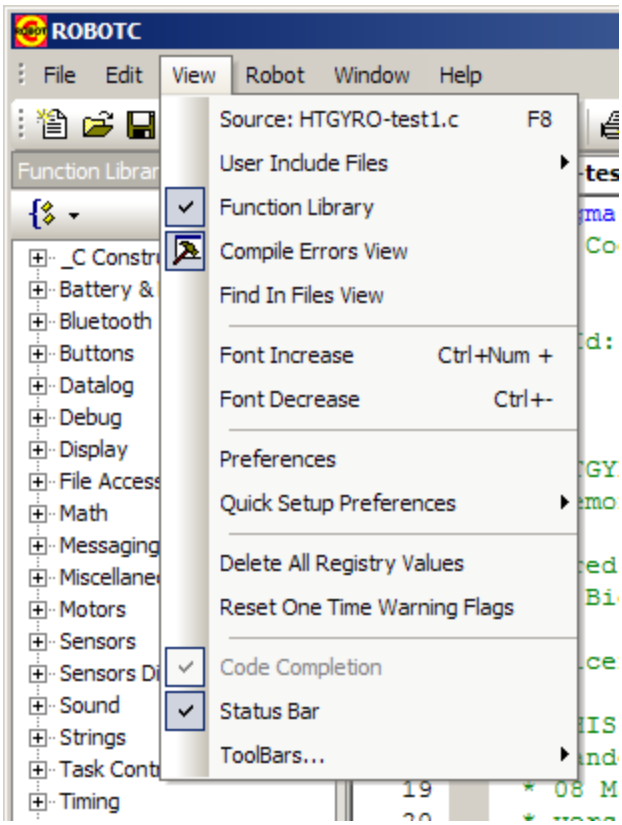
**Clear All Bookmarks**
Removes all bookmarks on the current program. See more information here.

**Toggle Bookmarks**
Turns a bookmark on or off on the line the text cursor is currently located. See more information here.

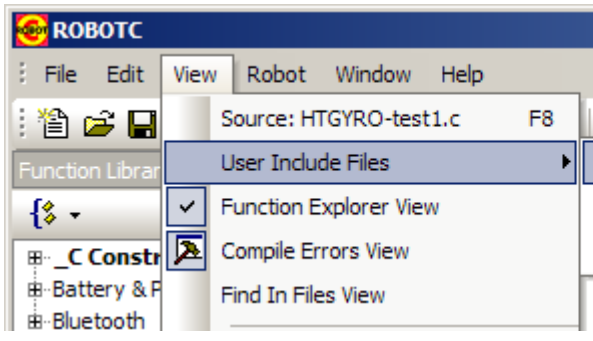
## 4.7.3 View

The **View** menu allows you to customize how the ROBOTC interface is displayed. There are two main levels of the View menu, depending if ROBOTC is set to "Basic" or "Expert" mode.

**View Menu:**




**Source: "File Name" (F8)**
Displays the current file that is opened. If there are "#include" files in your program, they will be listed under the "User Include Files" option, where they can be opened in a new tab.

| Source Code | | View Menu |
|---|---|---|
| 23 #include "drivers/HTGYRO-driver.h"<br>24<br>25 task main () { | ------> |  |



**Function Library**
Toggles the visibility of the Function Library panel in the ROBOTC interface. See more information here.

**Compile Errors View**
Toggles the visibility of the Compile Errors panel in the ROBOTC interface. See more information here.

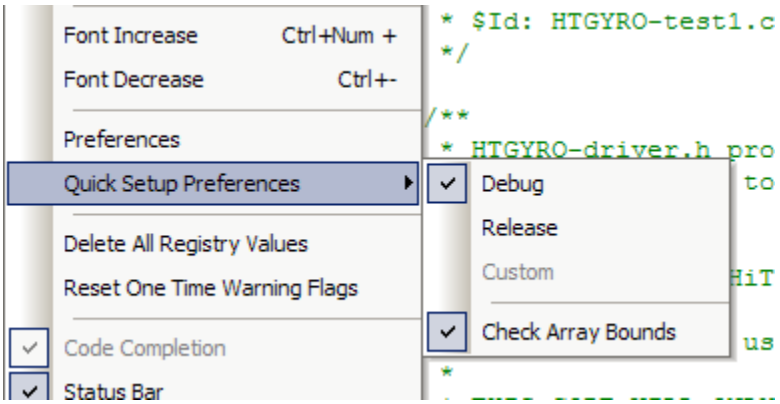
**Find in Files View**
Searches for specified text in multiple files.

### Font Increase
Increases the font size of the user program in the code editor.

### Font Decrease
Decreases the font size of the user program in the code editor.

### Preferences
When in Basic mode, opens the Basic version of the Preferences window. See more information here .
When in Expert mode, opens the Expert version of the Preferences window. See more information here.

### Quick Setup Preferences
Allows the user to quickly specify how the compiler optimizations occur.

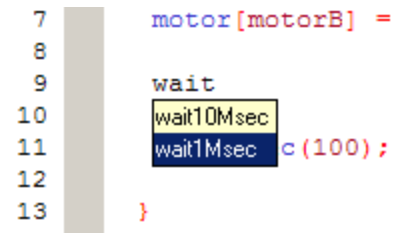



- **Debug**
  - Disables most compiler code optimization
  - Automatically enables compiler to generate code to detect and create exceptions for accessing arrays with an out-of-bound index during program execution.
  - For advanced users:
    - Enables the “ASSERT” macro which will only generate code during a “Debug” compile. The macro validates that the parameter you’ve passed it is a “true” (i.e. non-zero) value.
    - Defines the “_DEBUG” macro preprocessor variable that you can use in your own conditional compilation – i.e. “#if” preprocessor statements – code.
- **Release**
  - Enables all compiler code optimization techniques. This includes a “code re-ordering” optimization where the compiler may re-order the instructions it generates to reduce the amount of code and/or make it execute faster.
  - Re-ordered code may be difficult to debug because the debugger is often not able to map instructions back to the source code line that generated the low-level instruction!
  - The ROBOTC compiler typically reduces the size of generated code by 10 to 30% as a result of the compiler optimization techniques. Program execution time tends to be 5 to 15% faster.
- **Custom**
  - User specified compiler settings. These settings can be defined in the Expert Preferences window.

### Delete All Registry Values
(Expert Mode) Restores ROBOTC's internal settings to the first time you installed ROBOTC. All settings and histories are cleared. Requires ROBOTC to be restarted to take effect.

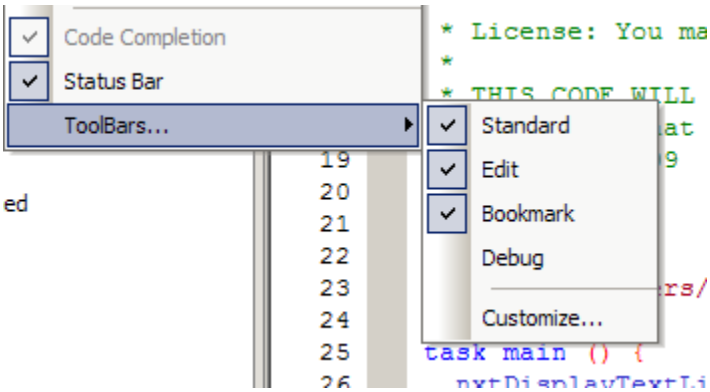
**Reset One Time Warning Flags**
(Expert Mode) Resets optional ROBOTC Warnings and Reminders so that they appear.

**Code Completion**
When checked, ROBOTC will offer suggestions as you program to shorten the amount of characters you will have to type.




**Status Bar**
Toggles if the status bar on the bottom of the ROBOTC interface is visible. See more information here.

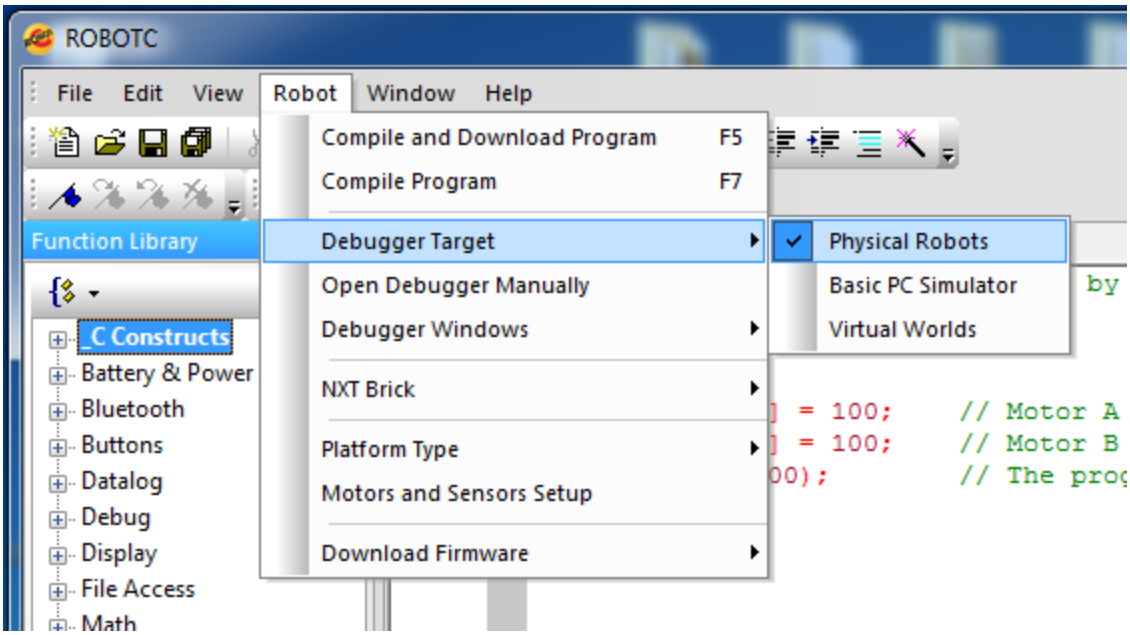
**Toolbars...**
Allows you to toggle what sections of the toolbar are visible. See more information here.




# 4.7.4 Robot

The **Robot** menu provides access to all of the tools available for interaction with your robot controller. Downloading programs, firmware, and launching the various debug windows are done through this menu.

You can also change which platform ROBOTC is set for and configure sensors and motors for that specific platform.

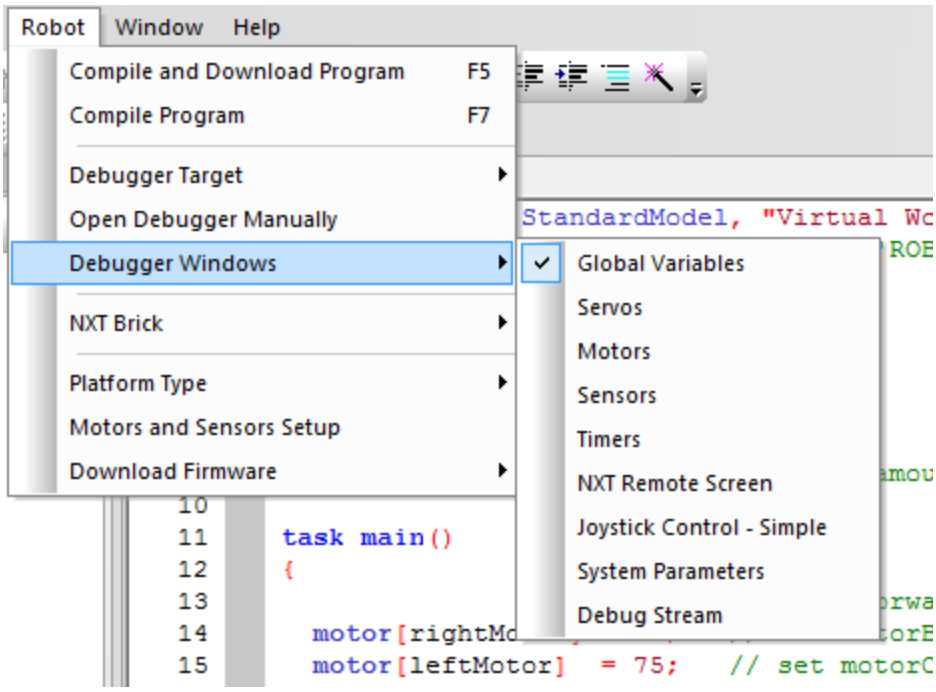



**Compile and Download Program**
Downloads the current program to the robot controller. ROBOTC will automatically recompile the program before downloading if the program has changed since it was last compiled.

**Recompile Program**
Compiles the current program without downloading it to the robot controller. This is useful for testing if your code will compile without actually having the controller attached to the computer.

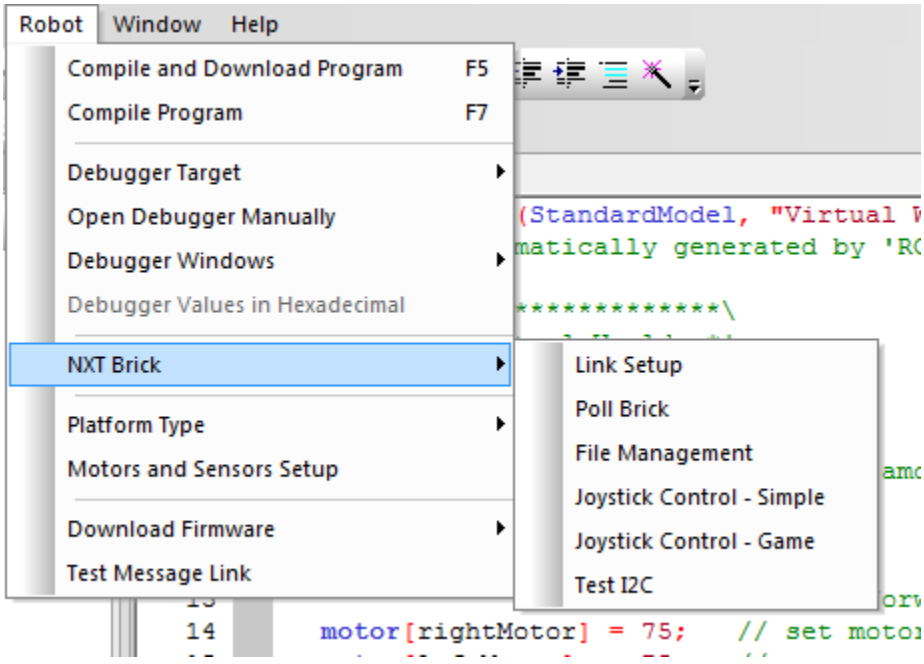
**Debugger Target**
Download your program to either a Physical Robot, PC Simulator, or a Virtual World.

**Debug Windows**
Allows you to toggle which Debug Windows are visible, only available if a connection has been established between your computer and the robot controller.



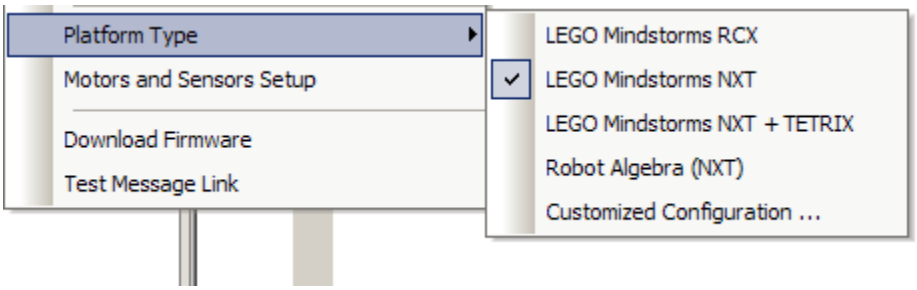

### Hexadecimal
(Expert Mode) Debug Windows will display values in hexadecimal format.

### NXT Brick
Provides access to additional NXT utility windows. See the NXT Brick Menu section for more help.




### Platform Type
Allows you to specify which robot controller you're developing programs for. ROBOTC will customize the

its interface, the available functions, and options available based on the Platform Type. Programming in the wrong Platform Type may cause compilation errors.



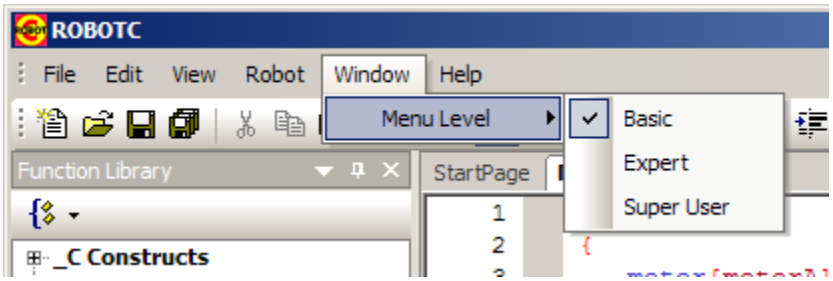

**Motors and Sensors Setup**
Opens the Motors and Sensor Setup window. See more information here.

**Download Firmware**
Downloads the ROBOTC VM firmware to the current platform. An "Open File" dialog will appear asking for the firmware file you wish to load. NXT firmware files are called "NXT0xxx.rfw" and RCX firmware files are called "fast0xxx.lgo", where xxx is the current version number of the firmware. See more information here (NXT) and here (RCX).

**Test Message Link**
(Expert Mode) See more information here.

## 4.7.5 Window

The **Window** menu is used to switch between the menu levels in ROBOTC.




**Basic** mode hides more of the advanced configuration settings, making the interface easier to navigate for new users.

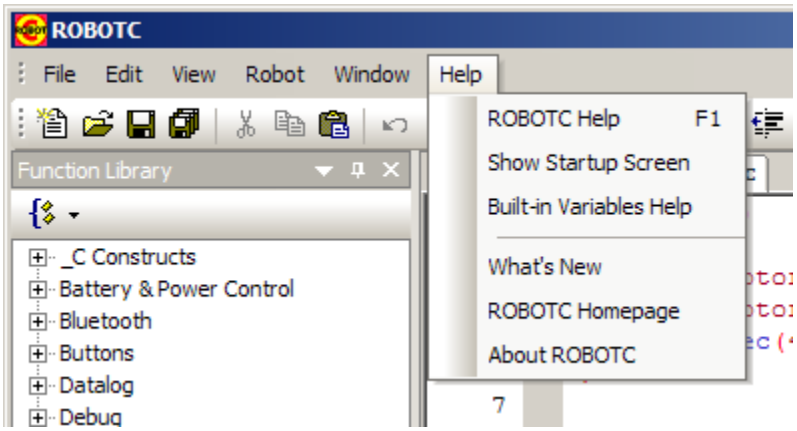
**Expert** mode shows most of ROBOTC's advanced settings, providing more tools for experienced users.

**Super User** mode shows all of ROBOTC's advanced settings, allowing a power user to take full advantage of ROBOTC's capabilities.

To switch menu levels, click on the "Window" menu, select "Menu Level" and then click on the desired mode.

# 4.7.6 Help

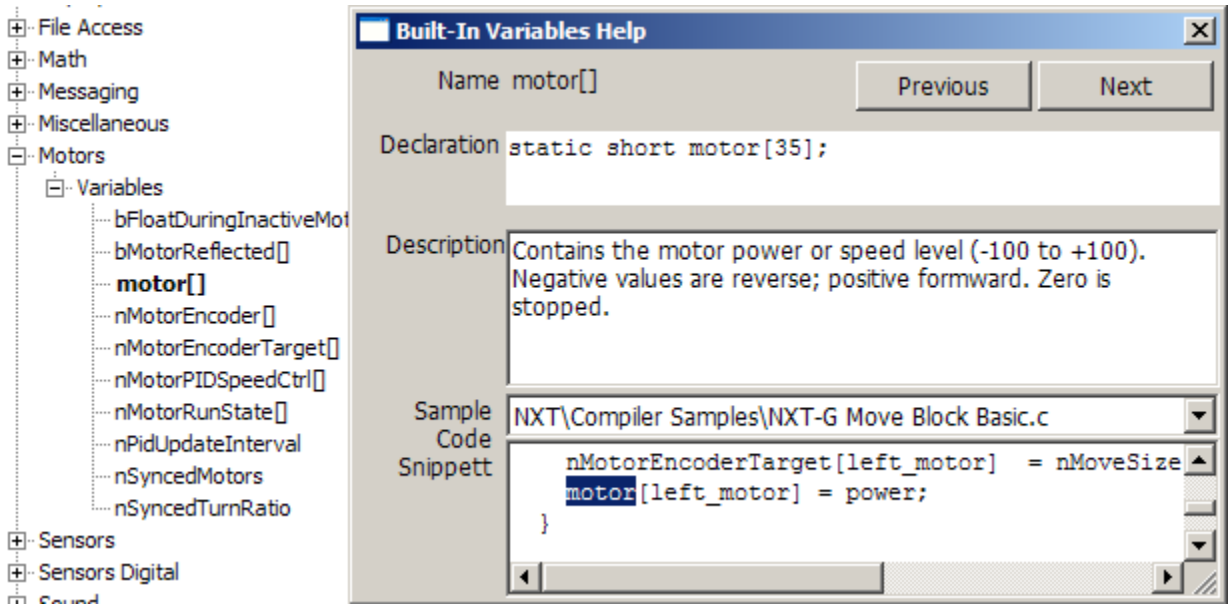
The **Help** menu allows you to access ROBOTC programming resources.




**ROBOTC Help (F1)**
Launches the built-in ROBOTC Help Documentation.

**Show Startup Screen**
Opens the ROBOTC Startup Screen in a new tab.

**Built-In Variables Help**
Launches the Built-In Variables Help, which gives a description of the keywords in ROBOTC, as well as examples from the Sample Programs.




**What's New**
Launches the What's New change log in a new tab.

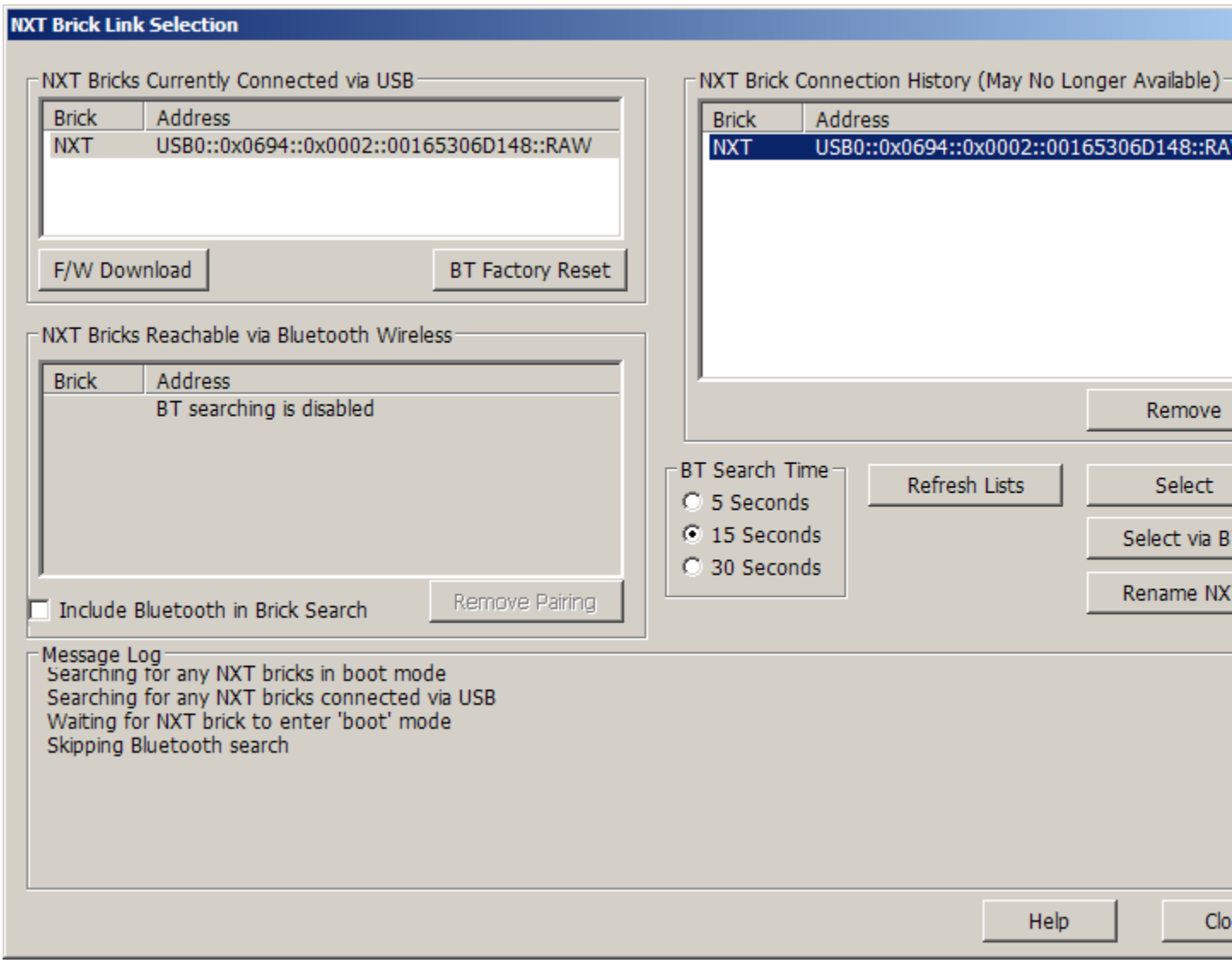
**ROBOTC Homepage**
Launches ROBOTC.net in your default browser.

**About ROBOTC**
Launches the About ROBOTC window.

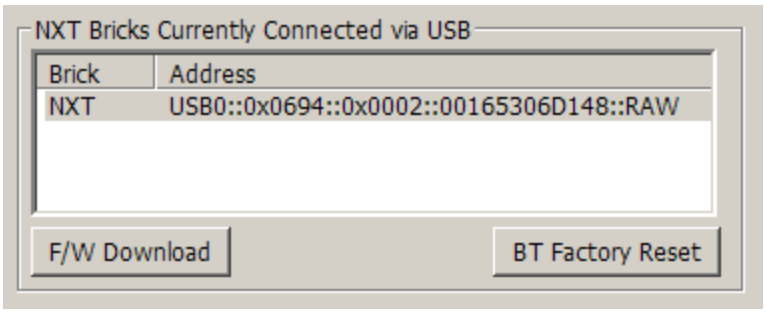
# 4.8 NXT Brick Menu

## 4.8.1 Link Setup

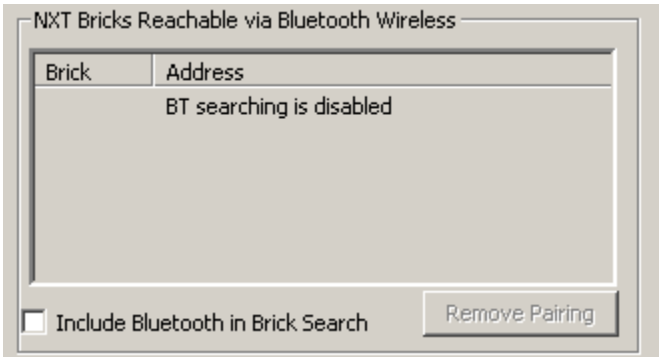
Overview Image



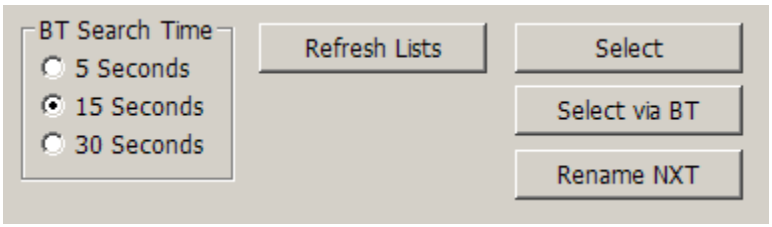

USB Connected Bricks



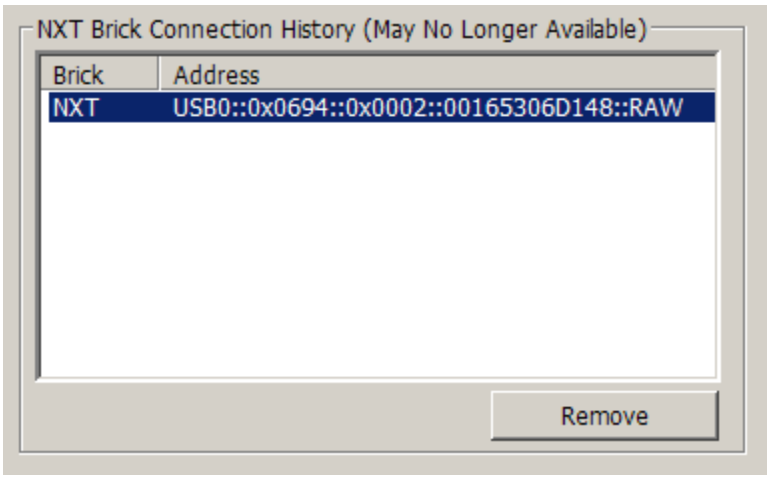

NXTs found by Bluetooth Search




Bluetooth Search Time, Refresh Lists, Selecting Bricks and Renaming NXTs



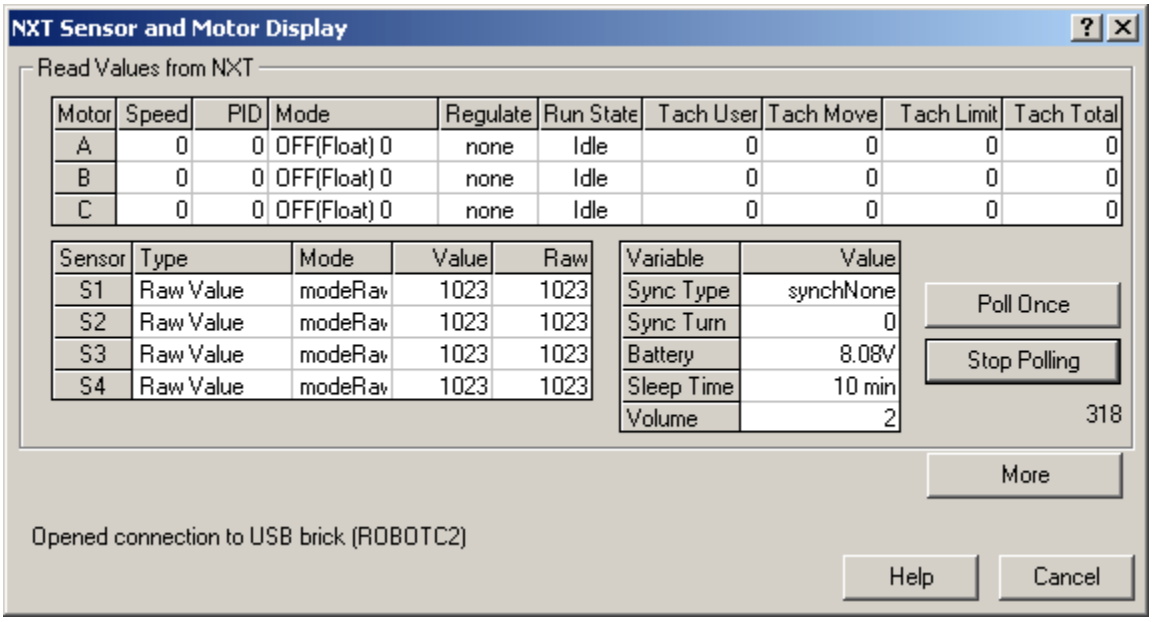

Connection History



# 4.8.2 Poll Brick

Note: This screen is very similar to the "NXT Devices" found in the ROBOTC debugger.

This is the Poll Brick window for the NXT robot controller platform. Different versions of the screen are available for other platforms supported by ROBOTC. The screen provides access to current values for the motors and sensors on the NXT. The “More” button expands the window to provide controls that enable you to “write” to the sensors and motors to setup their initial configuration.




**Poll Once**
Update all of the values on the Poll Brick screen only once.

**Poll Contiuously/Stop Polling**
Updaes all of the values on the Poll Brick screen contiuously until the "Stop Polling" button is pressed.

**Motor Section:**
Motor - Which motor the row of information refers to.
Speed - The current speed set by the NXT to the motor.
PID - If speed control is enabled, this is the actual speed of the motor (different from the "set" speed)
Mode - The configuration the motor is currently set to. (On, Off, Float, Brake)
Regulate - The current regulation mode the motor is in (none, speed regulated, encoder regulated, etc.)
Run State - The current state of the motor, either "Running" or "Idle." Idle draws signifigantly less battery power.
Tach (Encoder) User - Encoder count value that is total under user control (this variable is able to be reset.)
Tach (Encoder) Move - Amount you told the encoder to move.
Tach (Encoder) Limit - Value to stop when you ask the program to stop at a specific position.
Tach (Encoder) Total - Total Encoder count since program started.

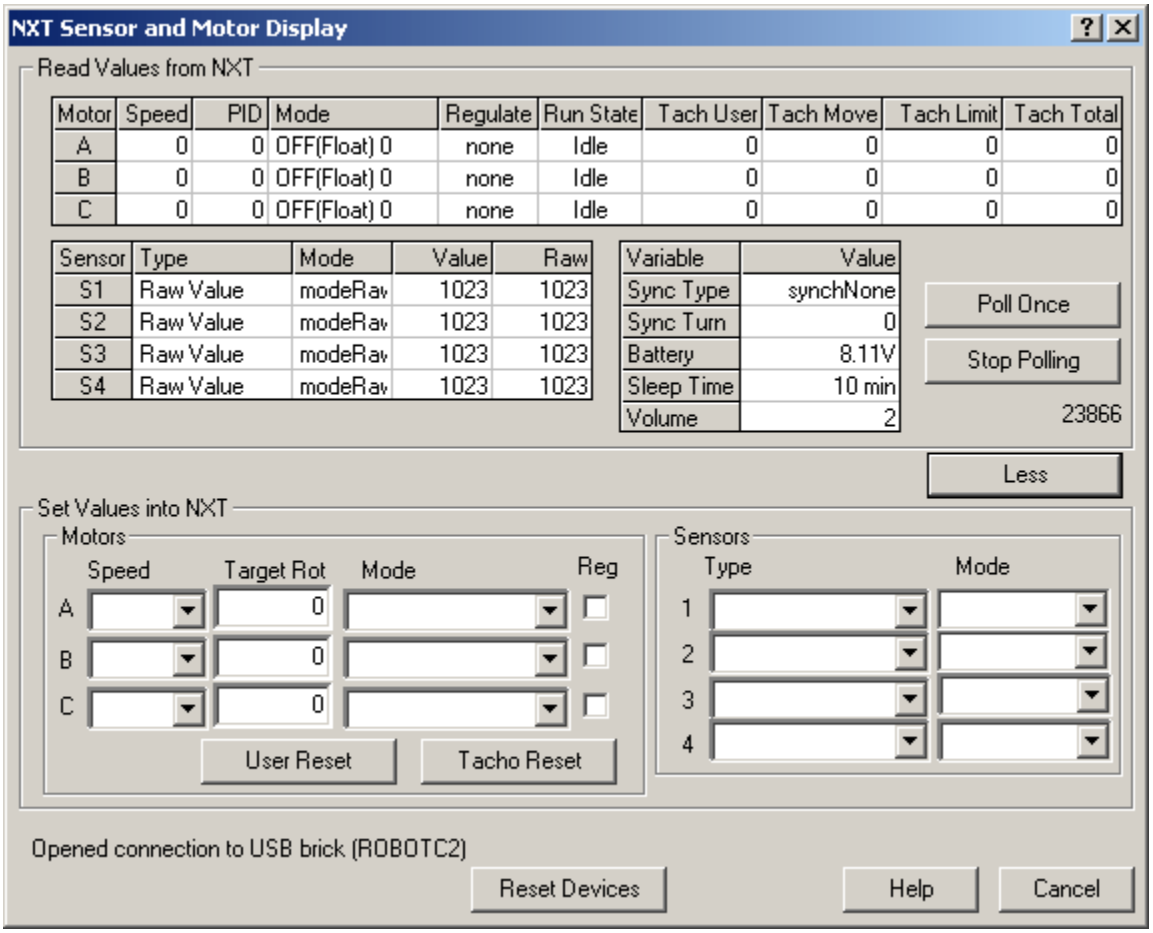
**Sensor Section:**
Sensor - Which sensor the row of information refers to.
Type - The current sensor type the sensor port is defined as.
Mode - The type of normalized data the sensor should return. (Raw, Percentage, Counts, etc.)

Value - The value of the normalized sensor.
Raw - The raw value of what the sensor is returning.

**Variable Section:**
Sync Type - The Sync Setting the NXT is currently set to (SyncAB, SyncAC, SyncBA, etc.)
Sync Turn - The Sync Ratio the NXT is currently set to.
Battery - The current voltage level of the battery.
Sleep Time - The current setting of the sleep timer on the NXT .
Volume - The level the internal speaker volume is set to.

---

**More/Advanced Display**

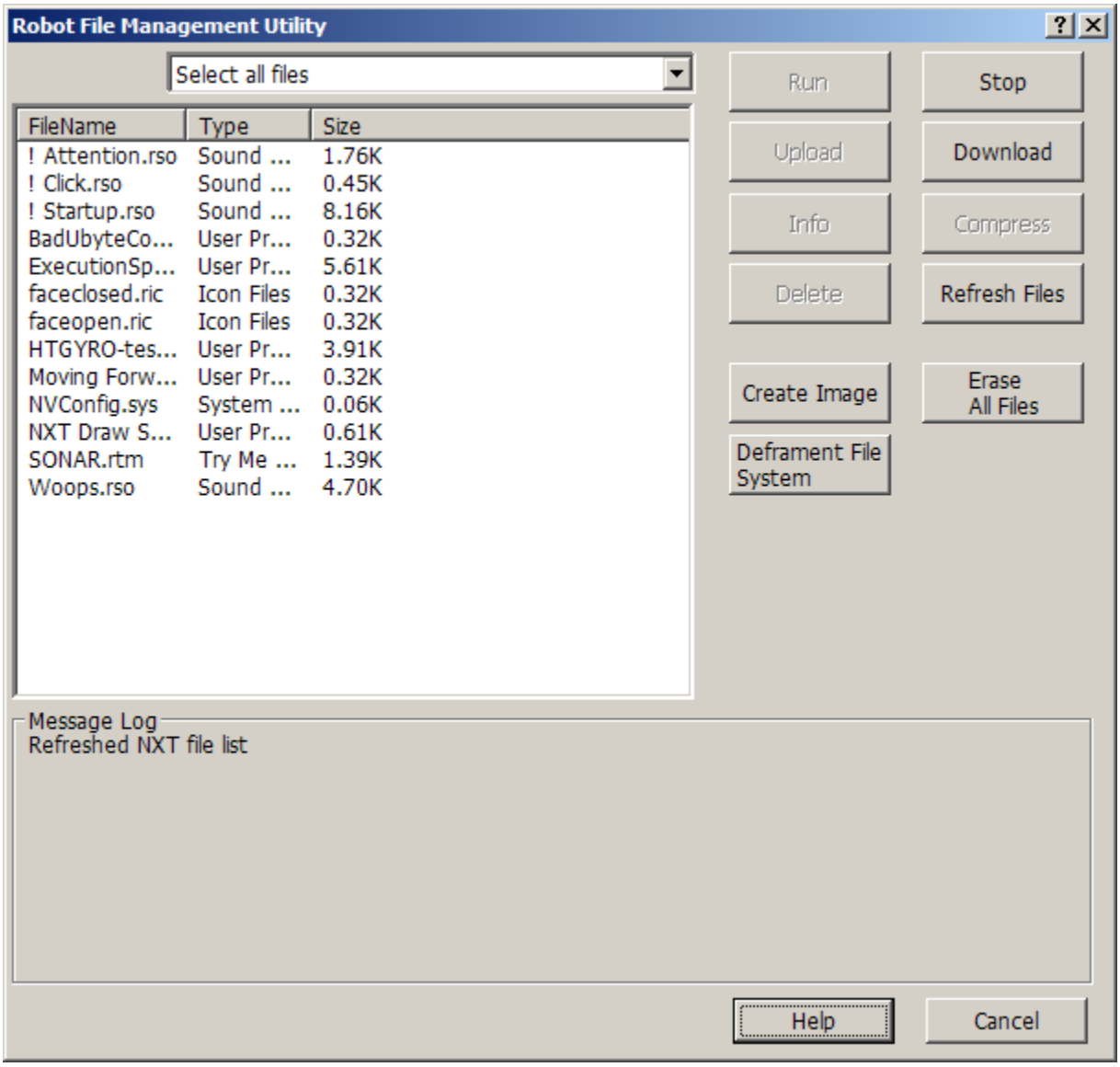



**Motor Section:**
Speed - Set the speed of the motor manually. (-100 to +100)
Target Rot - Set a rotation target for the built-in encoders to reach and then stop.
Mode - Change the mode of the motors (Brake, Float, On, Off)
Reg - Enable Speed Regulation on that motor.

**Sensor Section:**
Type - Manually change the type of the sensor on the labeled sensor port.
Mode - Manually change the mode of the sensor on the labeled sensor port.

## 4.8.3 File Management

This window is for managing files – upload to PC, download from PC, delete, etc – on the NXT.




The left half of this window contains a list of the files found on the NXT. Clicking on a column title will cause the list to be sorted in ascending order based on the contents of that column.

You can select one or more items from the list. Based on the types of items selected, the buttons on the right of the screen will be selectively enabled/disabled.

**Run / Play**
Used to run a program or play a sound file. Applies to the selected file.

**Stop**
Used to stop execution of a program or playback of a sound file.

**Upload**
Copies the selected files from the NXT to PC

**Download**
Copies files from the PC to the NXT. You will be prompted to select one (or more) files from the PC to transfer.

**Info**
Provides detailed info on the selected files – i.e. exact size and type. Information is displayed in the “Message Log” window

**Compress**
Used to compress the selected sound files on the NXT. Compressed files take about half the size of regular sound files.

**Refresh Files**
Refreshes the list of files on the NXT.

**Create Image**
Creates a firmware image file from the contents of the NXT flash memory. These image files can then be downloaded to another NXT and will have exactly the same files.

**Erase All Files**
Erases all files from the NXT

**Defragment File System**
As files are added and deleted from the NXT there may be gaps in the flash memory storage that are too small to use for a new file. The “Defragment” command will eliminate the gaps between files.

## 4.8.4 Joystick Control - Basic and Game

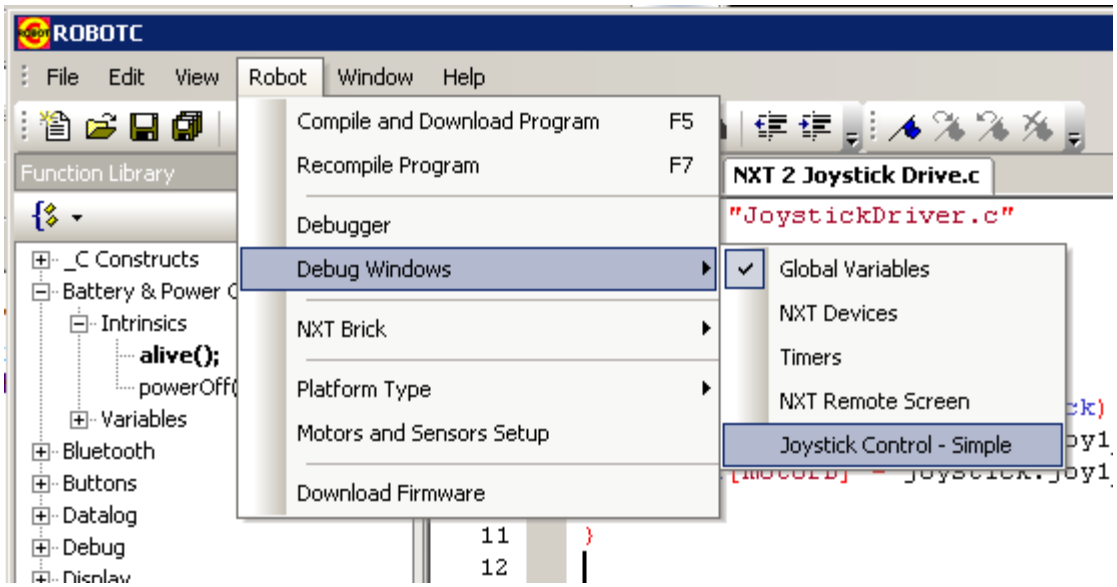
ROBOTC has built two different Joystick Controller stations built into the interactive debugger. "Joystick Control - Simple" is a debugger window to control the NXT via a Logitech USB remote control. "Joystick Control - Game" is a full featured Controller Station which is used mainly for FIRST Tech Challange or other competitions that are NXT or TETRIX based.

---

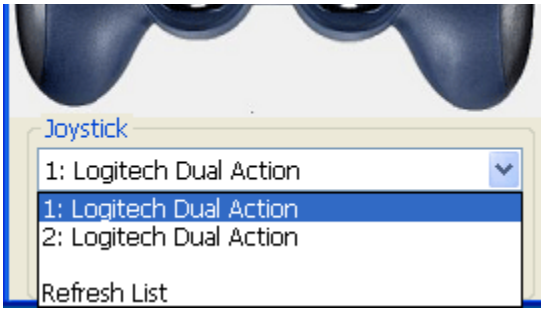
**"Joystick Control - Simple"**

After your program has been downloaded and the debugger is opened, you can open the Controller Station by:

- Downloading your program and starting the debugger
- Go to the "Robot" menu
- Choose the "Debug Windows" sub-menu
- Click on the "Joystick Control - Simple" menu option to open the Controller Station




Once the Controller Station is opened, ROBOTC will look for any joysticks attached to your computer via USB. You can choose which joystick you want to robot to be controlled with by changed the joystick under the available drop-down menu. If you have no joysticks available, this list will be empty and ROBOTC will alert you that you have "No Controllers Configured"




You can see what data is being generated by the Joystick Station by looking at the X1, Y1, POV, X2, Y2 and Buttons display directly below the dropdown menu. This will give you realtime feedback of what values are being sent to your NXT from the Joystick Station. This data will also be illustrated with green dots to reflect the values and button presses.

ROBOTC will send joystick data to your NXT over Bluetooth or USB, but only when the Joystick Control window is opened. You will need to have the debugger open to use the Joystick Station.

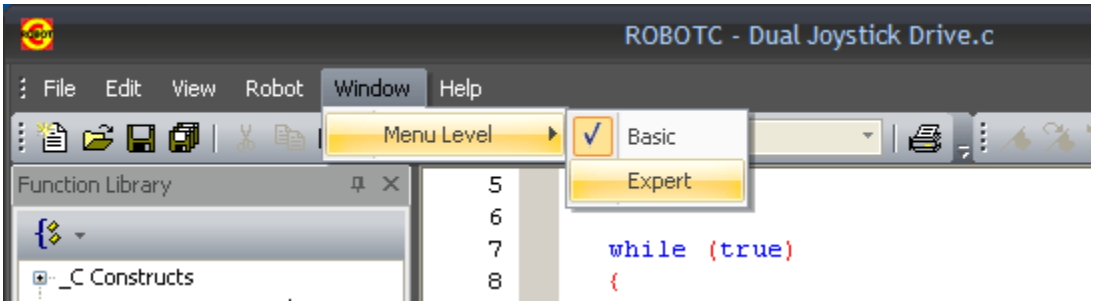

---

**"Joystick Control - Game"**

There is also a second Joystick Control window, "Joystick Control - Game". This window is specifically designed to emulate the FIRST Tech Challenge game mode. To test your FTC competition programs, you can use the controller station to mimic what the Field Management System will do. This includes switching between Autonomous and User Control, Changing if your robot is on the blue or red alliance and also disabling (or pausing) your robot. These commands can be found on the left side of the Joystick Station.

You can open the Controller Station by:

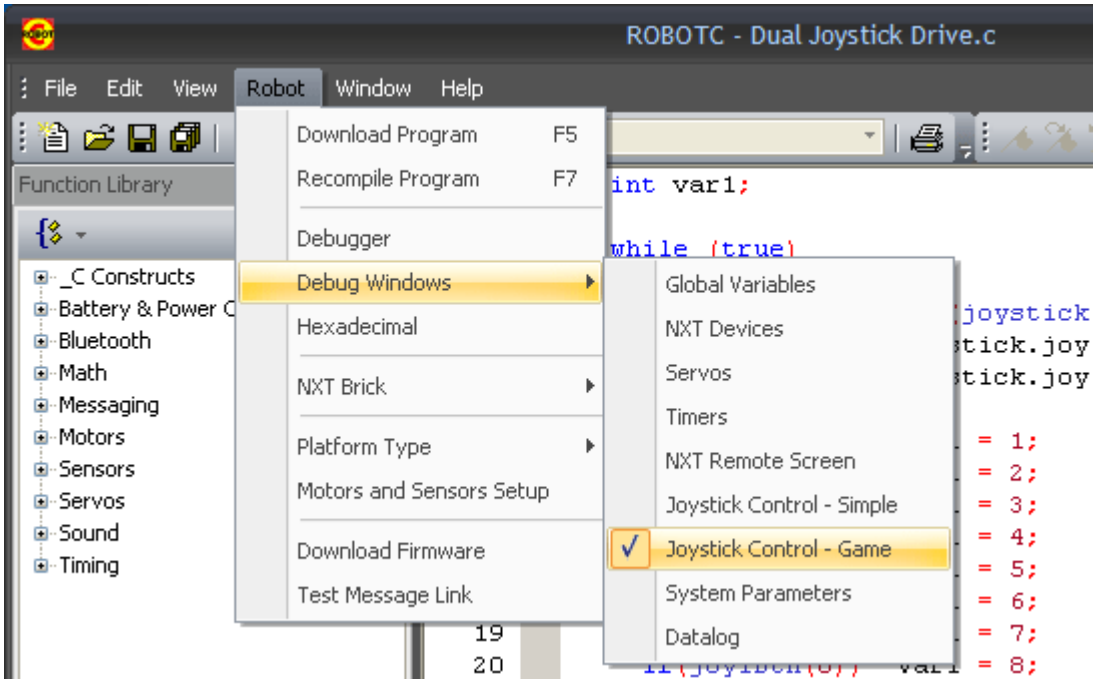
- Go to the "Window" menu
- Choose the "Menu Level" sub-menu
- Click on the "Expert" menu option




After setting the Menu Level to "Expert", you can then access the "Joystick Control - Game" station by:

- Downloading your program and starting the debugger
- Going to the "Robot" menu
- Choose the "Debug Windows" sub-menu

- Click on the "Joystick Control - Simple" menu option to open the Controller Station




As you can see, the window has a few more options than the "Joystick Control - Simple" screen did.

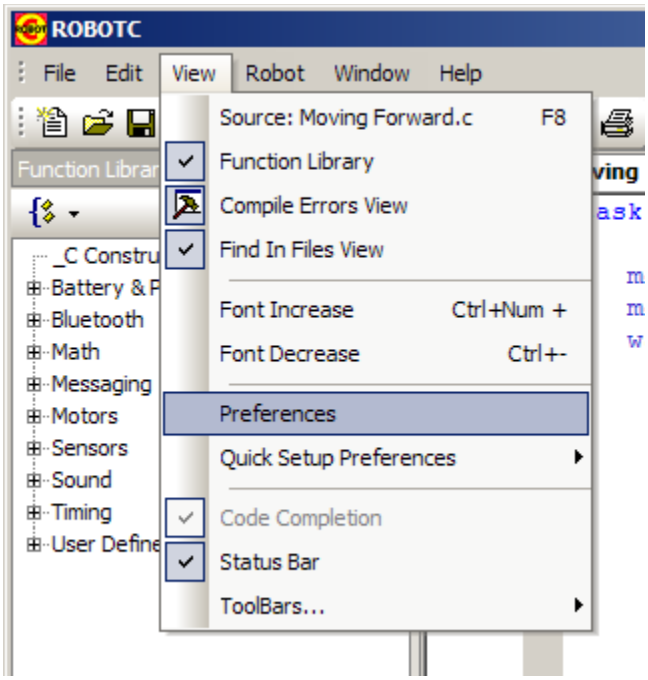

If you would like to use two controllers, click the "Dual Joysticks" button to expand the Joystick Control window to facilitate two controllers. You can assign the same controller to both Primary and Secondary Joysticks, but this is not recommended.


# 4.9 Preferences - Basic

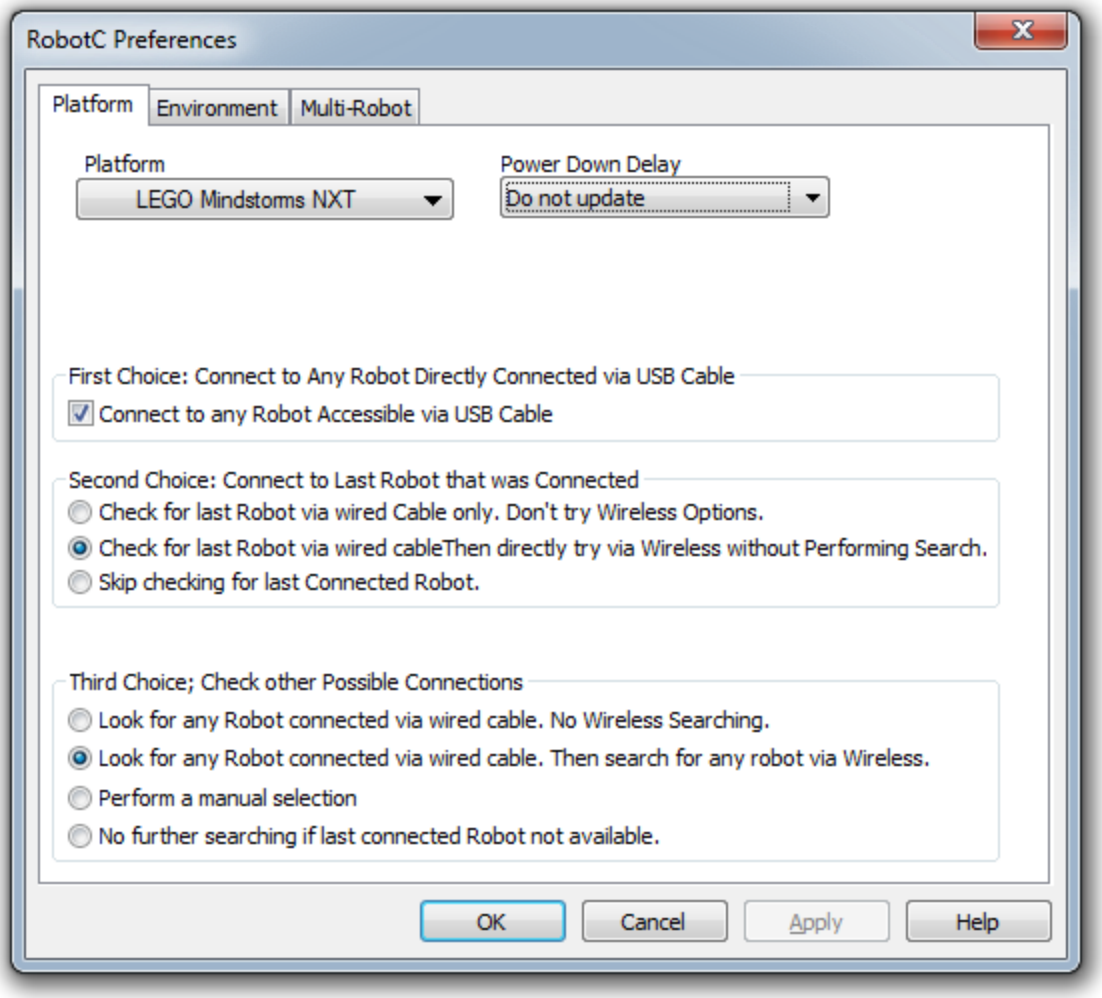
The preferences window allows you to modify platform-related and connection settings. In order to get to the Preferences window, simply click on the View menu, and select "Preferences".

You have two different sets of preferences that you can access: Basic and Expert. This corresponds with which "Window" mode you are currently in.

**View Menu - Basic Mode:**




**Preferences - Basic Mode:**




**Platform**
You can choose the platform you are using in this drop-down menu.

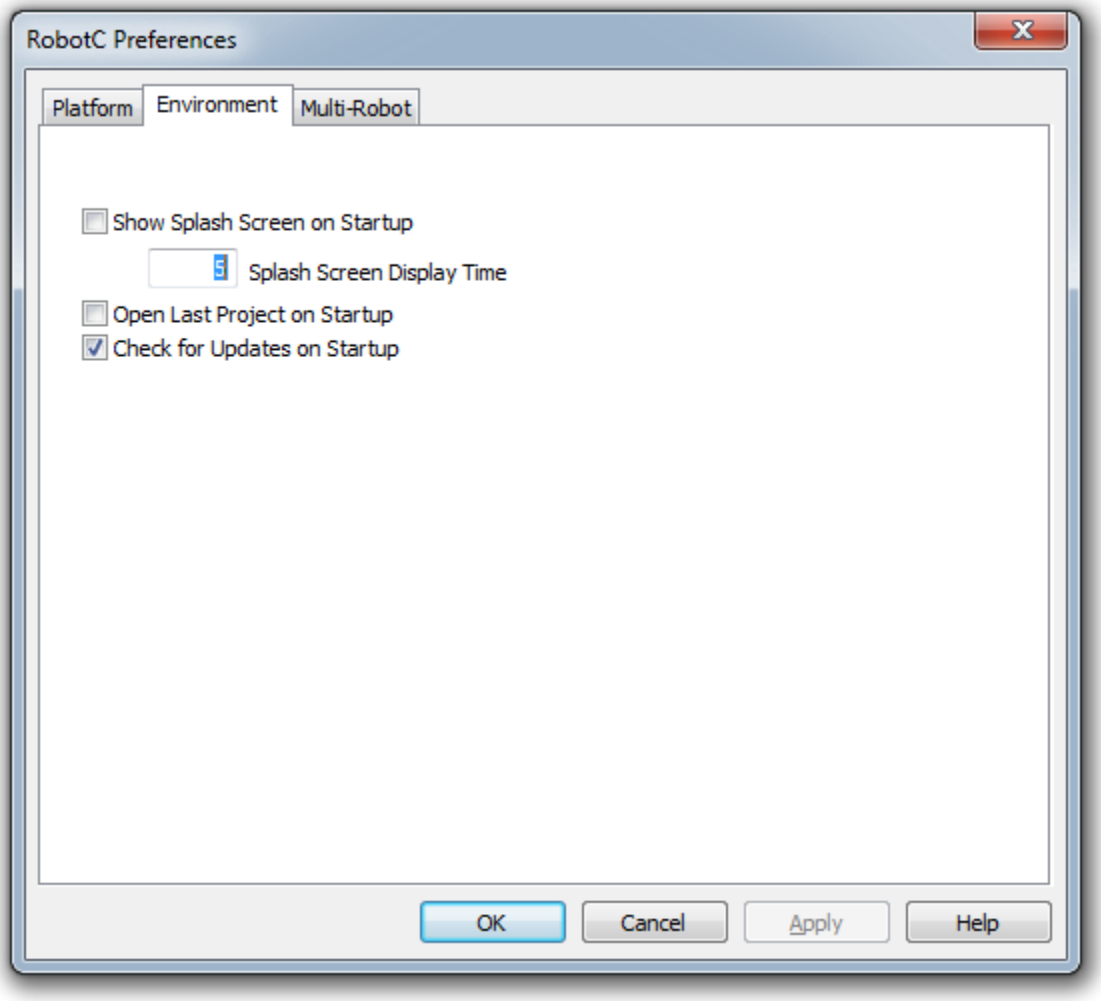
**Power Down Delay**
This setting adjusts the time it takes for the brick to automatically power down. The ROBOTC preference is set to "Do Not Update" by default, which means ROBOTC will not override the setting you've manually set via the NXT's on-brick interface.

**First Choice: Connect to Any NXT Connected via USB**
Checking this box allows ROBOTC to communicate with any NXT brick that's connected to the computer with the USB cable.

**Second Choice: Connect to Last NXT that was Connected**
If an NXT isn't found (A brick isn't connected via USB to the computer) then ROBOTC will attempt to connect to the previous NXT by USB, and then the previous Bluetooth contact by default.
You can set ROBOTC to only connect via USB, or not check for the previous NXT.

**Third Choice: Check other Possible Connections**
If the previous NXT isn't found, then ROBOTC will again attempt to just find any NXT connect to the computer through USB or Bluetooth.

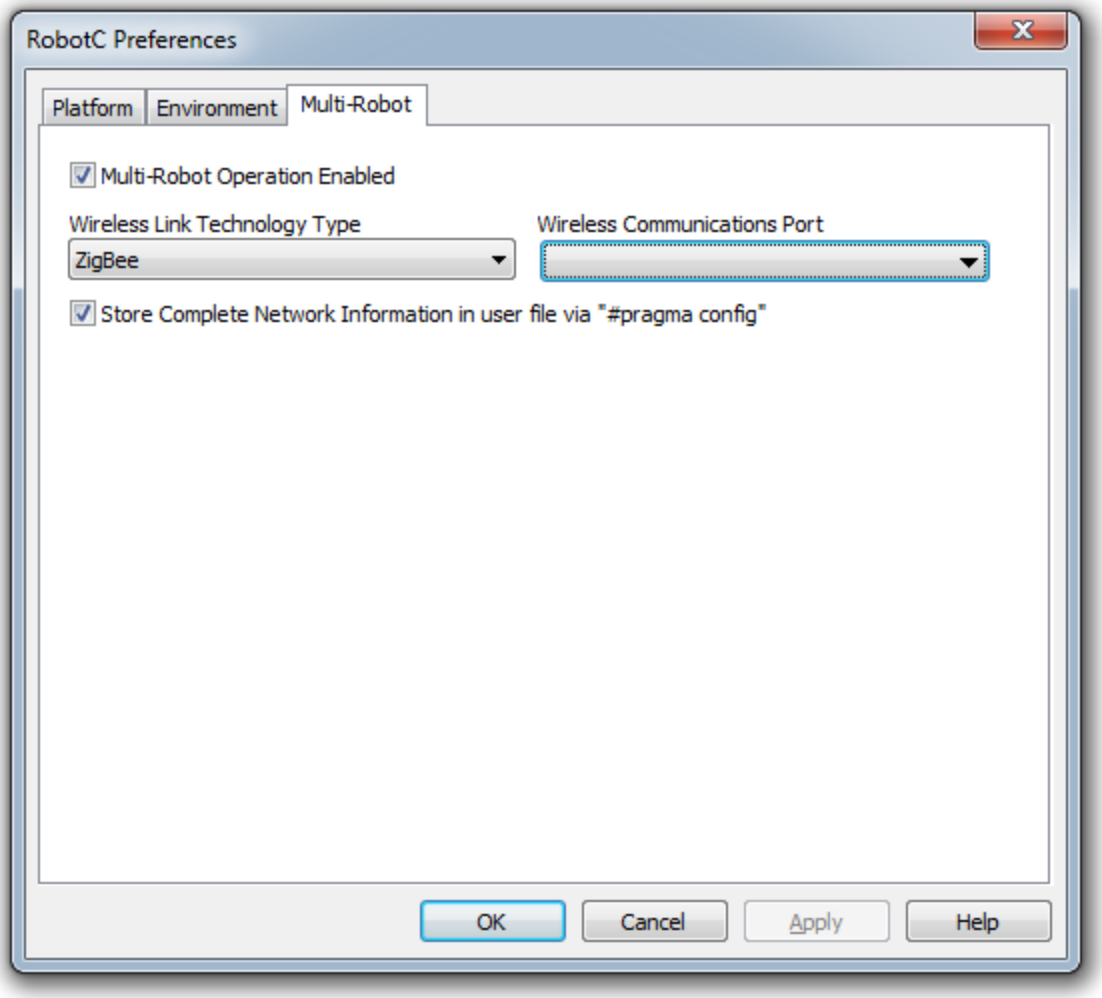
You can also set it to check again for any NXTs connected through Bluetooth, perform a manual selection, or not check again for another connection.




**Show Splash Screen on Startup**
This setting toggles whether or not you wish to be greeted by the ROBOTC info splash page on startup. You can adjust how long the splash screen remains open by entering the amount of seconds in "Splash Screen Display Time".

**Open Last Project on Startup**
Toggle wheter or not ROBOTC should automatically open the last project when it starts up.

**Check For Updates on Startup**
If this option is checked, ROBOTC will attempt to check for updates via the internet every 7 days.




**Multi-Robot Operation Enabled**
Toggle whether or not ROBOTC should enable multi-robot operation.

**Wireless Link Technology Type**
Choose how the robots will communicate.  Currently ROBOTC supports WiFi, Bluetooth, and ZigBee radios.

**Wireless Communcations Port**
Select which port to use for communcating with the robot(s).

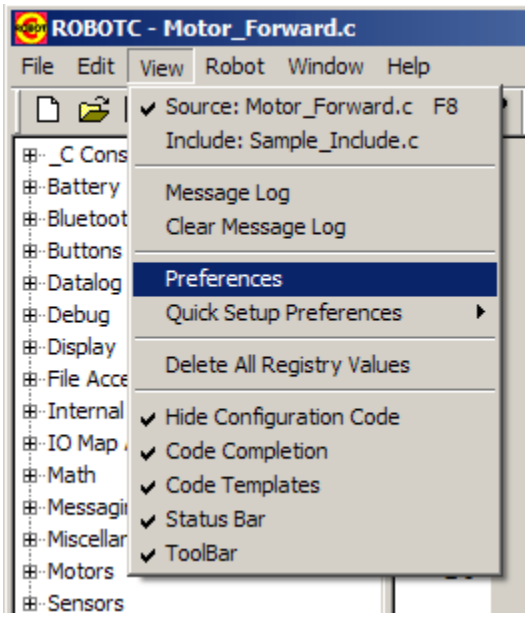
**Store Complete Network Information in user file via "#pragma config"**
Select this if you'd like to save your setup for future use.

# 4.10 Preferences - Expert

The preferences window allows you to modify platform-related and connection settings. In order to get to the Preferences window, simply click on the View menu, and select "Preferences".

You have two different sets of preferences that you can access: Basic and Expert. This corresponds with which "Window" mode you are currently in.

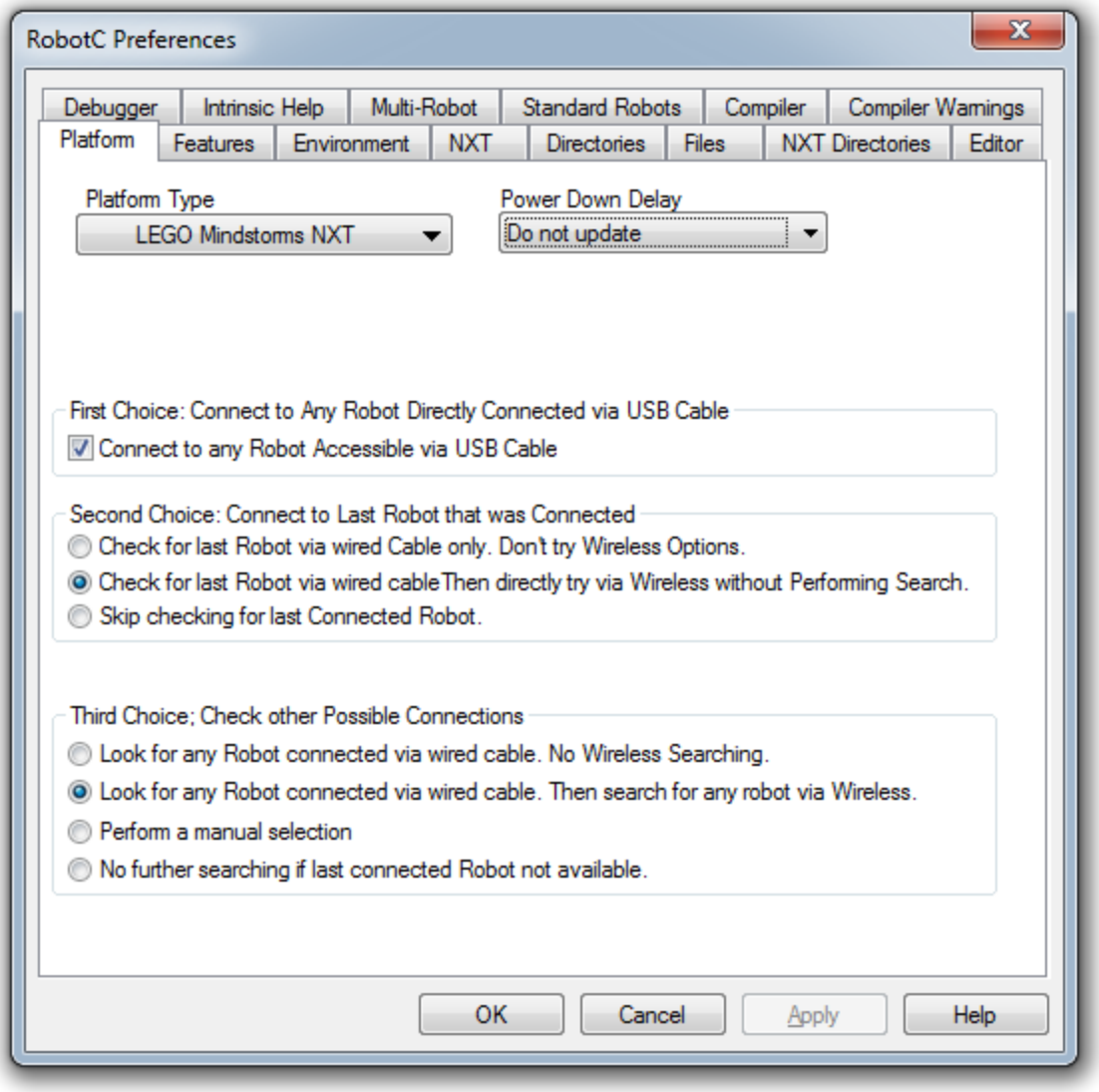
### View Menu - Expert Mode:




---

### Preferences - Expert Mode

### Platform Tab




**Platform Type**
You can choose the platform you are using in this drop-down menu.

**Power Down Delay**
This setting adjusts the time it takes for the brick to automatically power down. The NXT is set to not shut down automatically by default.

**First Choice: Connect to Any NXT Connected via USB**
Checking this box allows ROBOTC to communicate with any NXT brick that's connected to the computer with the USB cable.

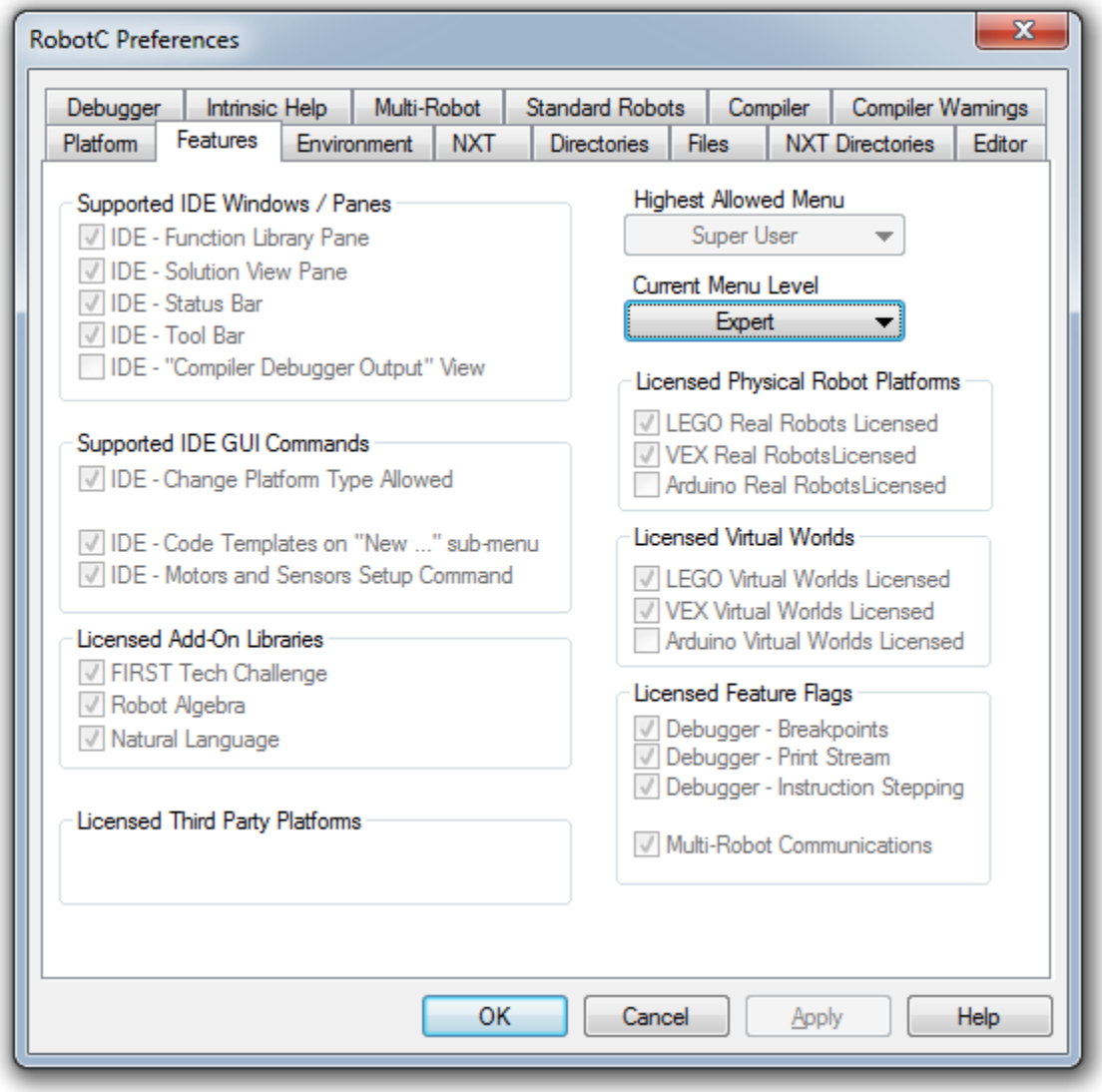
**Second Choice: Connect to Last NXT that was Connected**
If an NXT isn't found (A brick isn't connected via USB to the computer) then ROBOTC will attempt to connect to the previous NXT by USB, and then the previous Bluetooth contact by default.
You can set ROBOTC to only connect via USB, or not check for the previous NXT.

### Third Choice: Check other Possible Connections

If the previous NXT isn't found, then ROBOTC will again attempt to just find any NXT connect to the computer through USB or Bluetooth.
You can also set it to check again for any NXTs connected through Bluetooth, perform a manual selection, or not check again for another connection.

---

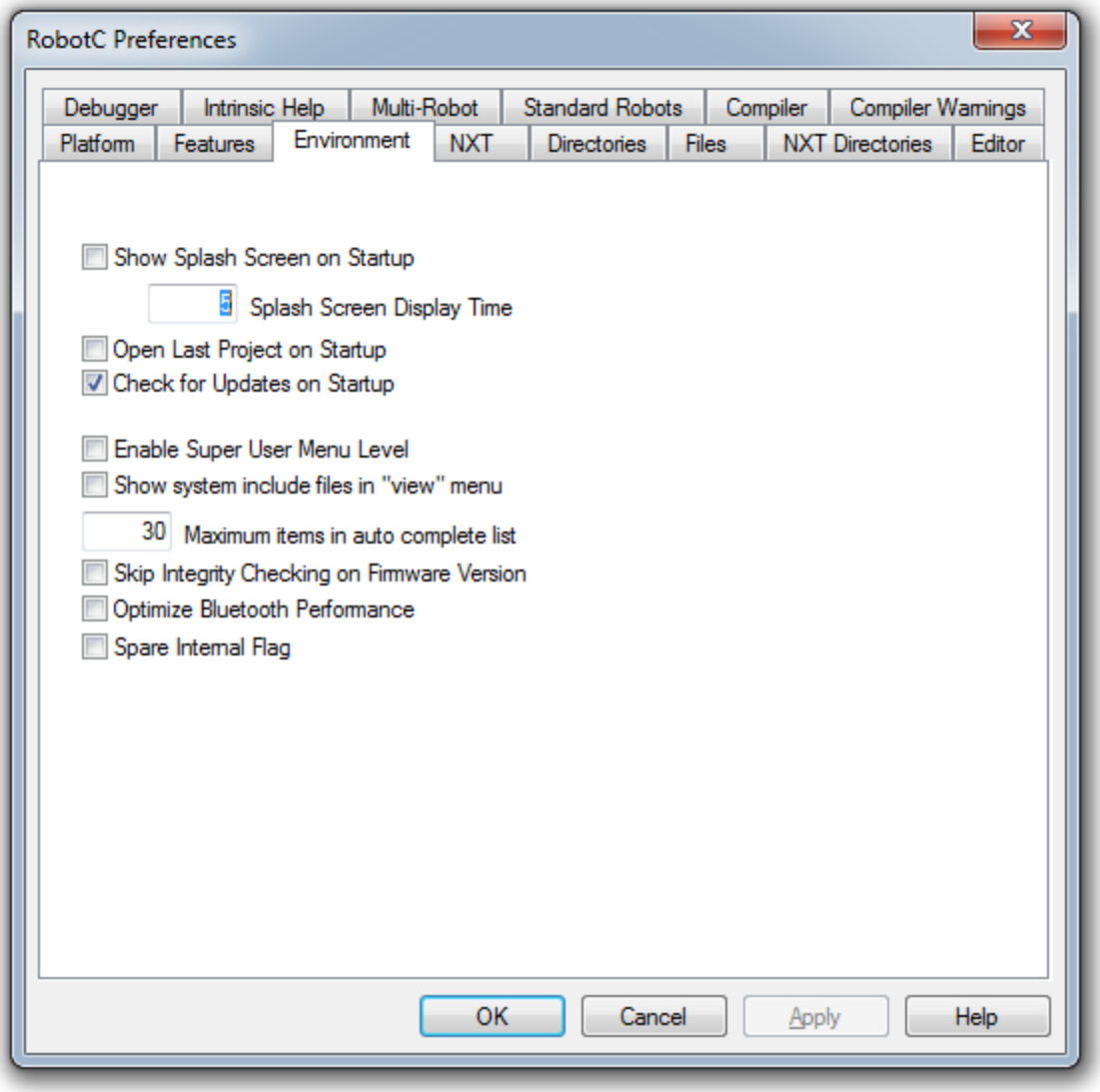
### Features Tab




The Features Tab allows the user to control which features to use within ROBOTC.

---

### EnvironmentTab




**Show Splash Screen on Startup**
This setting toggles whether or not you wish to be greeted by the ROBOTC info splash page on startup. You can adjust how long the splash screen remains open by entering the amount of seconds in "Splash Screen Display Time".

**Open Last Project on Startup**
Toggle wheter or not ROBOTC should automatically open the last project when it starts up.

**Check For Updates on Startup**
If this option is checked, ROBOTC will attempt to check for updates via the internet every 7 days.

**Enable Super User Menu Level**
Toggle wheter or not users are able to enter Super User Menu Level.

**Show system include files in "view" menu**
Toggle wheter or not ROBOTC will show system files in the "view" menu.

**Maximum items in auto complete list**
Choose how many items will appear in the auto complete list while typing

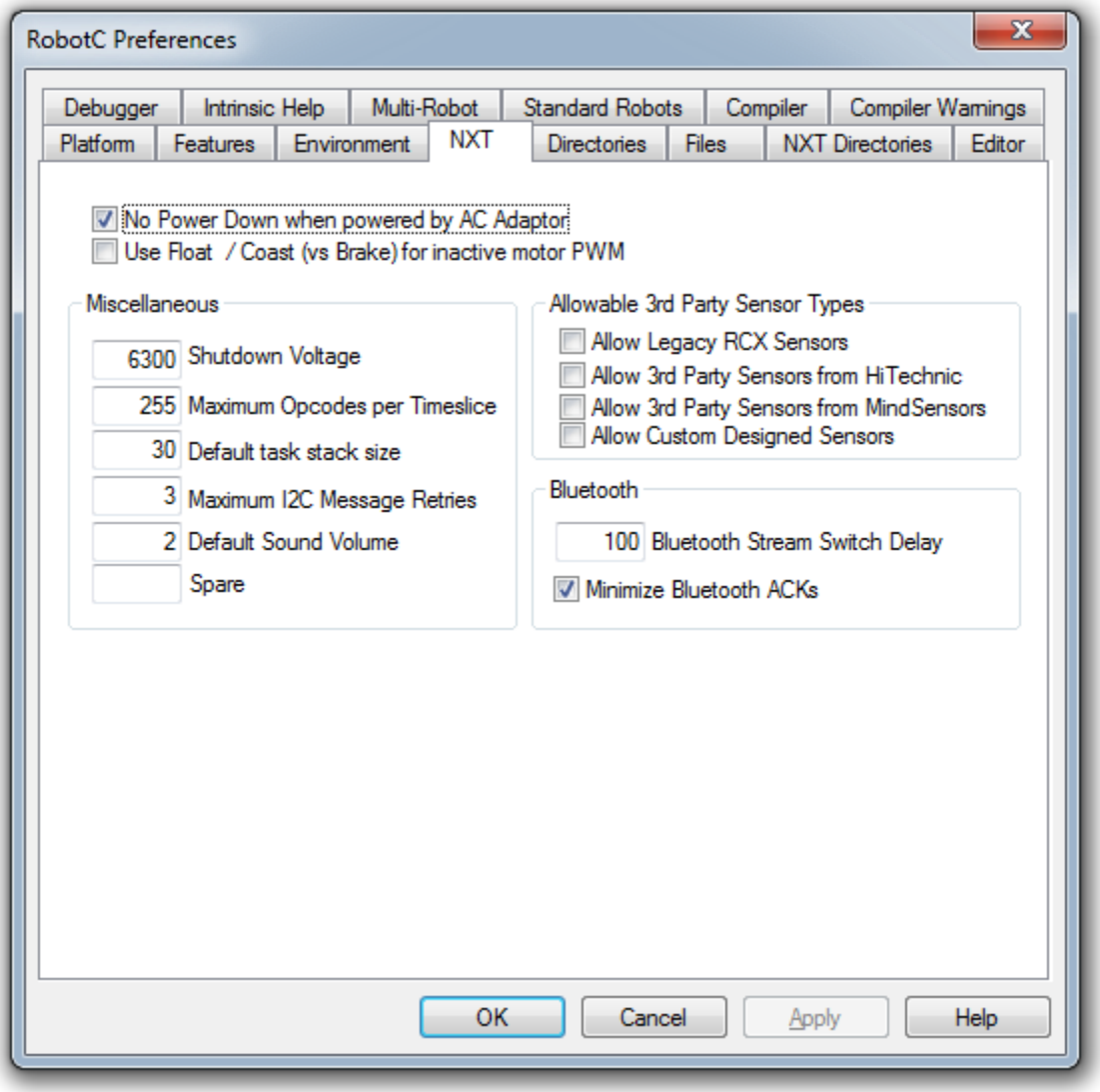
**Skip Integrity Checking on Firmware Version**
Toggle wheter or not ROBOTC will run an integrity check of the firmware before downloading a program to the robot.

**Optimize Bluetooth Performance**
Toggle wheter or not ROBOTC optimizes bluetooth performance.

**Spare Internal Flag**
Toggle wheter or not ROBOTC will include an extra internal flag for debugging.

---

**NXT Tab:**



**No Power Down when powered by AC Adaptor**
Check this box if you don't want the NXT to power down when it is plugged in with the AC adaptor.

**Use Float (vs Brake) for inactive motor PWM**
Check this box if you want the motor to "coast" or "float" to a stop at the end of a behavior. For instance, if the last behavior in your program is a "turn left for 2 seconds" behavior, then the robot will turn left for 2 seconds, and then coast to a stop if the checkbox is CHECKED. (otherwise it will immediately brake to a stop.)

**Minimize Bluetooth ACKs**
Advanced feature. We advise that you keep this checkbox checked in order to minimize the overhead involved with Bluetooth communication.

**Miscellaneous**
**Maxiumum Opcodes per Timeslice**
Advanced Feature. Do not modify.

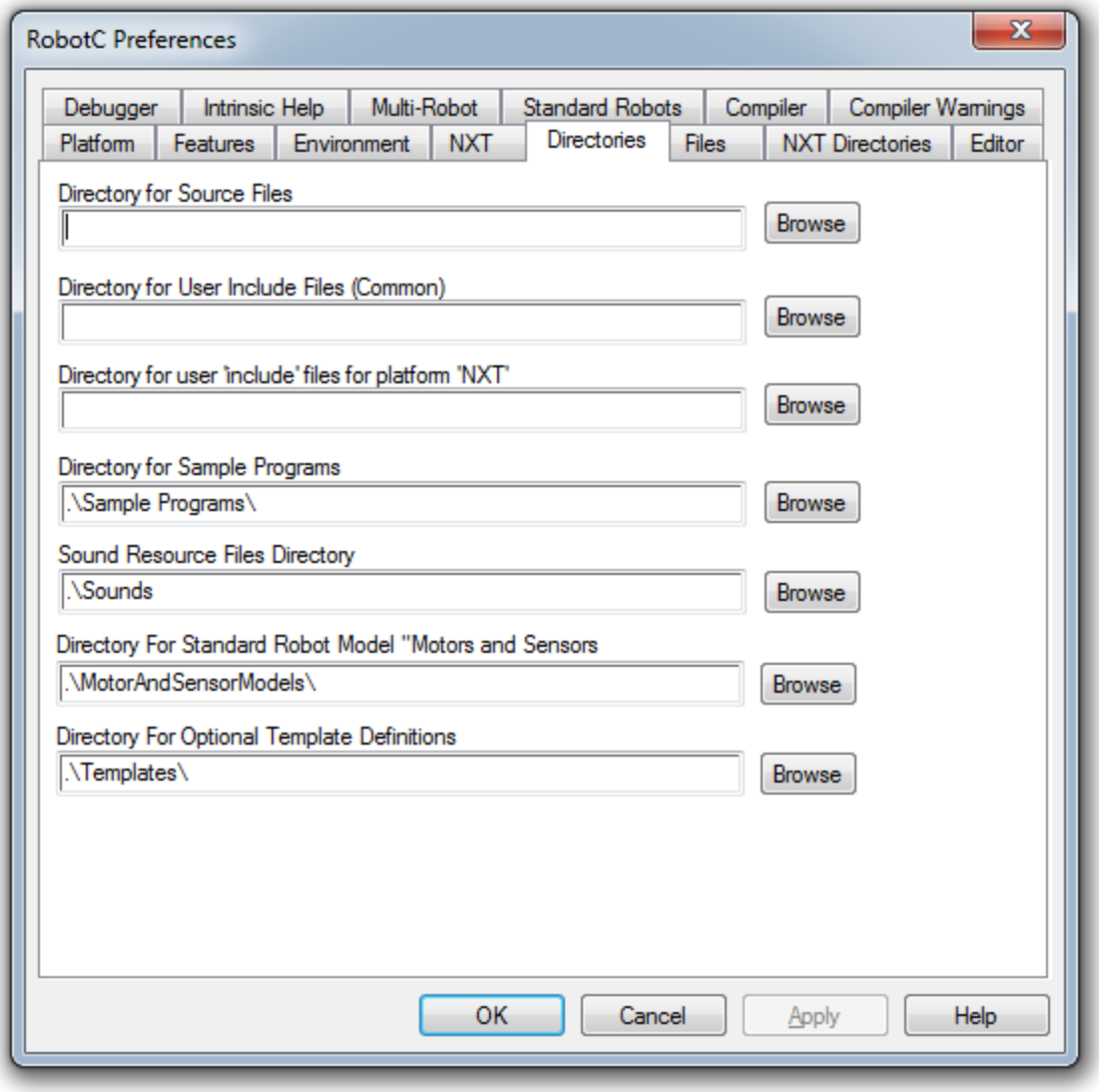
**Default Task Stack Size**
Advanced Feature. Do not modify.

**Maximum I2C Message Retries**
The number of times an NXT I2C message will be retried before an error message will be returned.

**Default Sound Volume**
The default volume the NXT is set to. Scale from 0-4, 0 being mute and 4 being the loudest.

---

### Directories Tab:




**System Include Header File Name**
This is the directory where the ROBOTC header file is located. You can change this location by clicking on the "Browse" button and selecting a directory.

**Directory for Source Files**
You can set a default directory for your source files by clicking on the Browse button and selecting a directory.

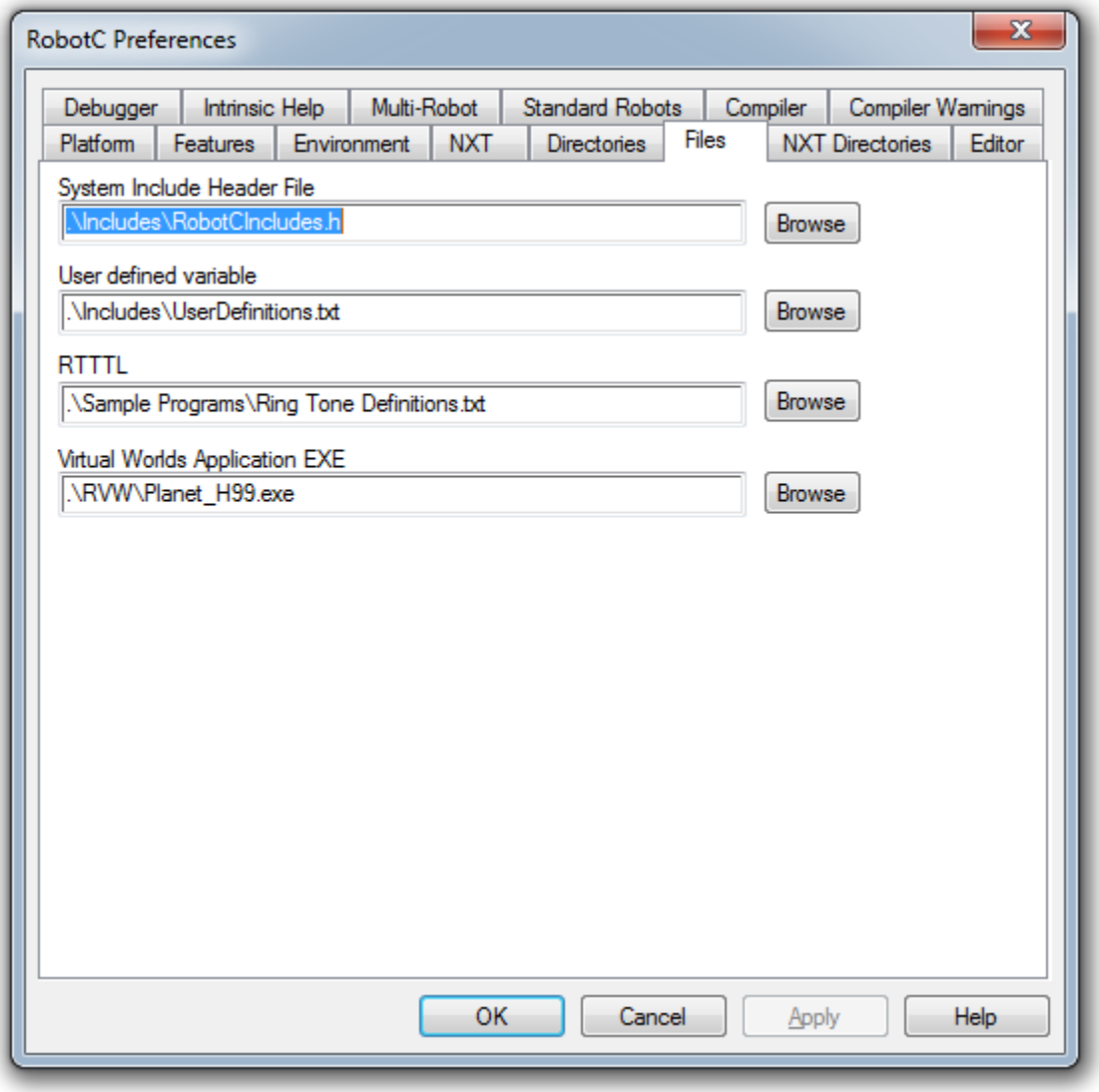
**Directory for Sample Programs**
This is the location of Sample Programs that are packaged with the ROBOTC installer.

**Debugging trace log file**
The Trace.txt file is associated with ROBOTC to log all information dealing with the ROBOTC's debugging process.

### RTTTL File
The location of the Ring Tone definitions is set here.

---

### Files:




### System Include Header File
Specify a default header file to include in all programs.

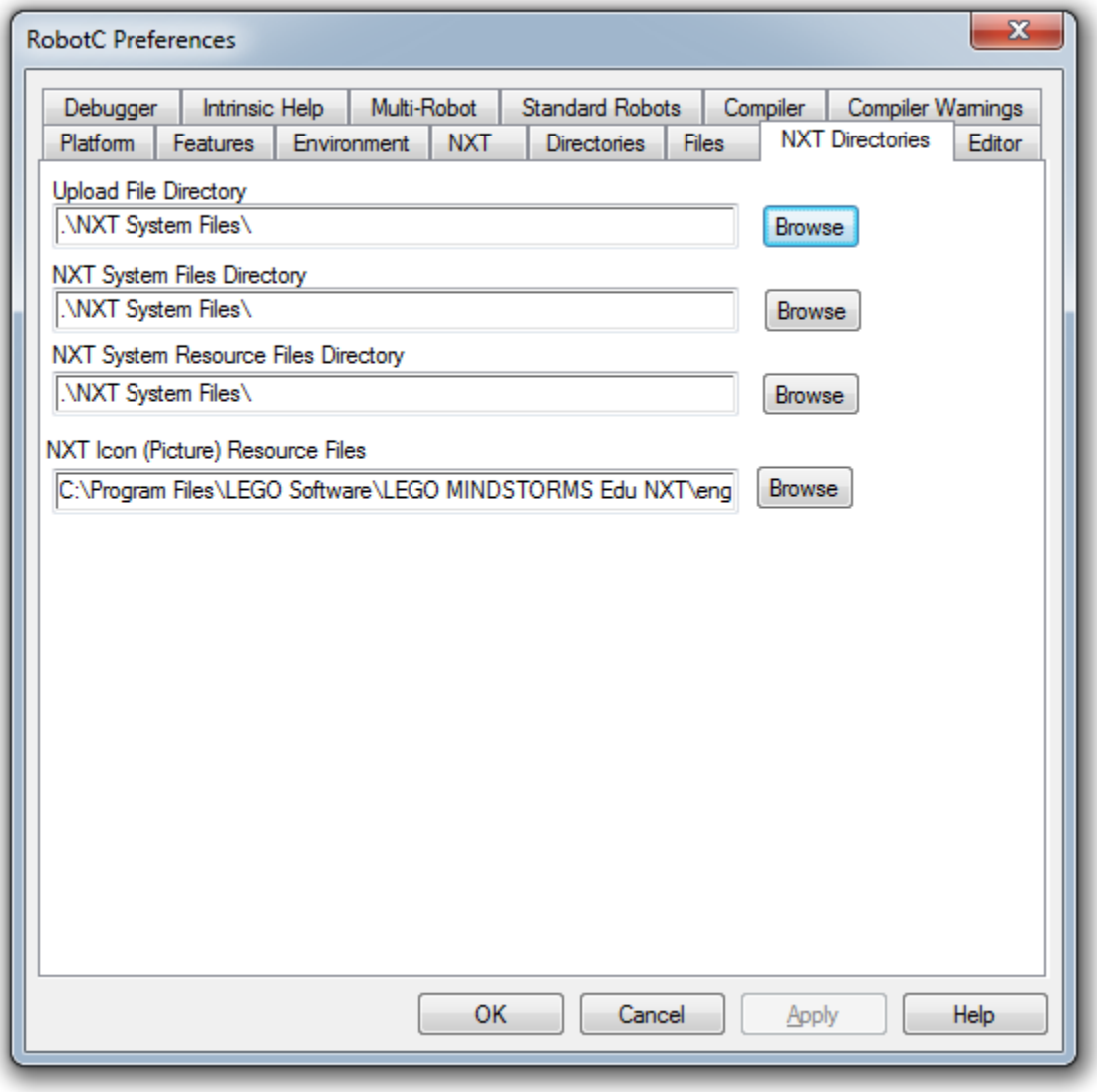
**User defined variable**
Specify a file to include that contains any user defined variables.

**RTTTL**
Specify a default ring tone definitions file.

**Virtual Worlds Application EXE**
Specify a default Virtual Worlds file.

---

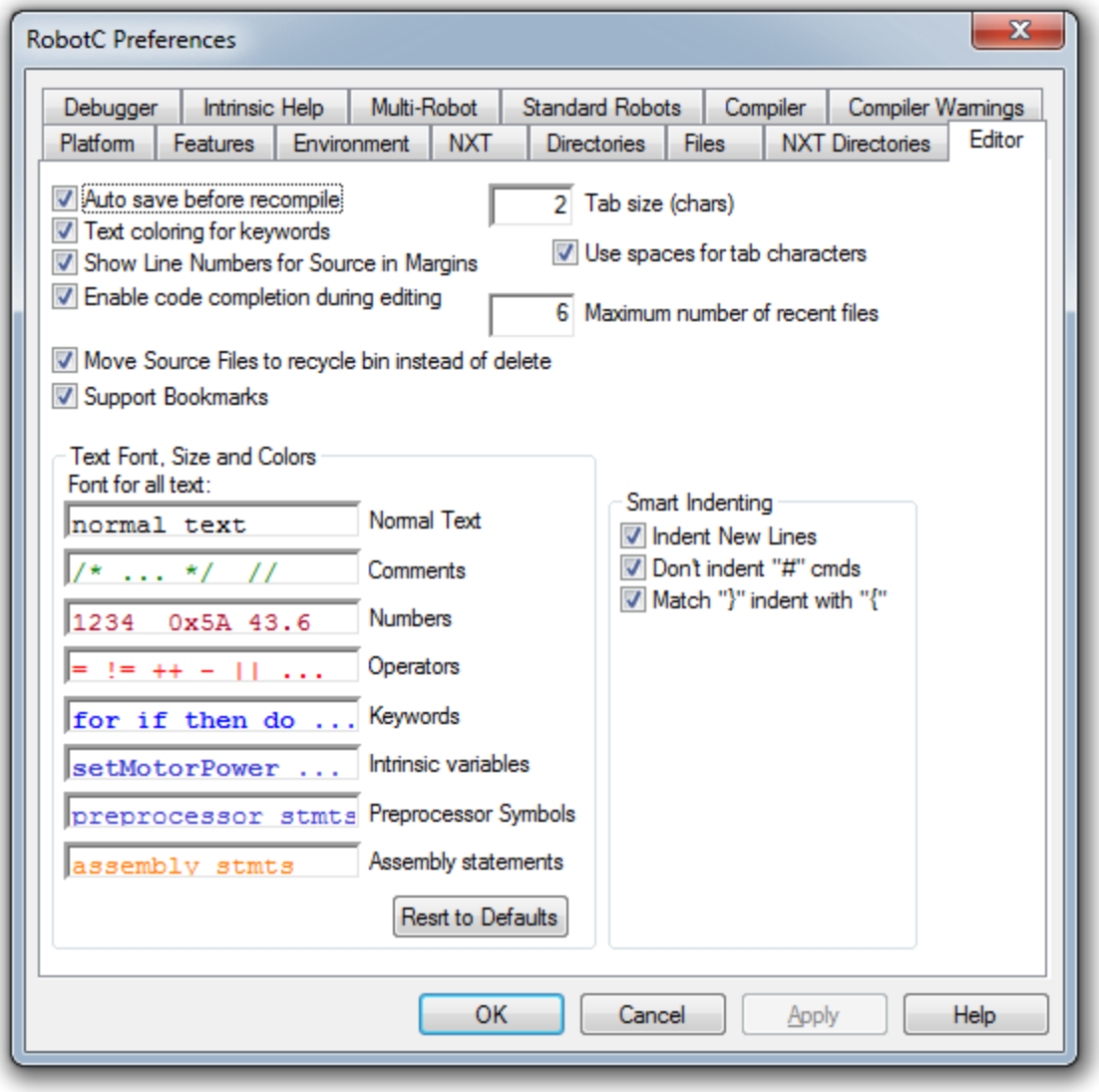
### NXT Directories:




### Upload File Directory, NXT System Files Directory, NXT System Resource Files Directory
Default diretory to use to upload files from the NXT. Typically contains the compiled “Try Me” files and other files that are included in a NXT firmware file ( i.e. sound files, image files).

---

### Editor:




**Auto save before recompile**
ROBOTC will automatically save your program prior to compiling. You can turn this feature off by unchecking the box.

**Text coloring for keywords**
Checking this box allows ROBOTC to color code keywords within the program editing window. For instance, the motor command (motor[*motorname*]) will display as a different color to make it easier for the programmer to distinguish between normal (black, by default) text and keywords.

**Show Line Numbers for Source in Margins**
This setting toggles the display of the line numbers in the program editing window.

**Enable code completion during editing**
Code completion is used when you begin to type a keyword in ROBOTC. It helps you type code more efficiently and remember longer keywords .

**Enable code templates**
The code templates window is the window to the left of the program editing window. It contains the structure, or "template", of code used in ROBOTC. You can set ROBOTC to not show the code templates window by unchecking this box.

**Move Source Files to recycle bin instead of delete**
In ROBOTC, when you start a new program the default name for the program is SourceCode. If you decide not to save this program, ROBOTC will, by default, delete the file. If this setting is checked, ROBOTC will place this file in the Recycle Bin in case you accidentally forgot to save it.

**Text Font, Size and Colors**
You can set the font, size, and color for different types of text in ROBOTC.

**Tab Size**
The tab size is the size of the spacing input when you press the TAB button on your keyboard. The larger the number, the larger the spacing.

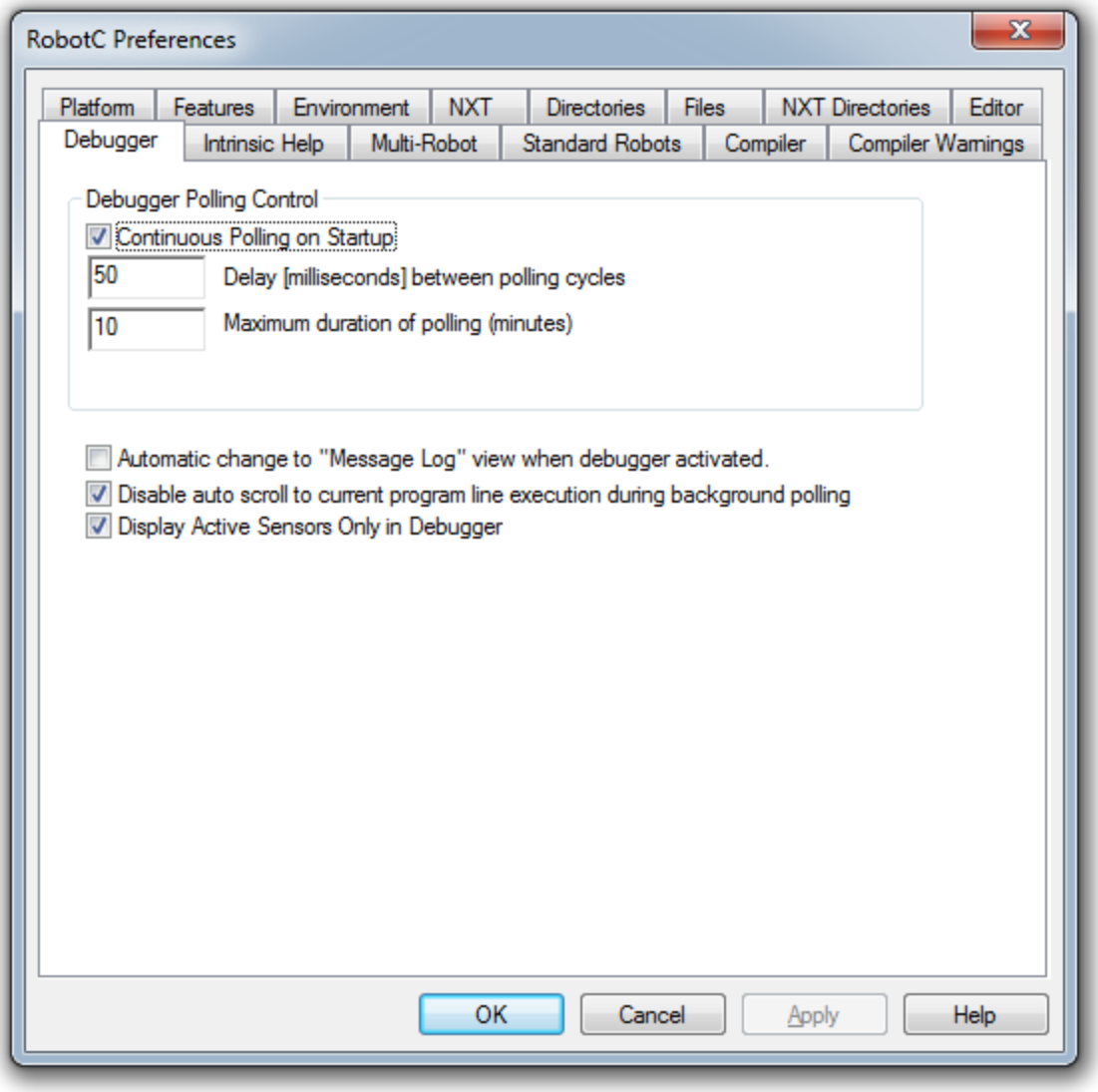
**Use space for tab characters**
You can set ROBOTC to use space characters instead of tab characters when you press the TAB button on your keyboard.

**Maximum number of recent files**
You can set the number of files shown in the File menu here.

**Smart Indenting**
ROBOTC takes care of common typing steps such as indenting on a new line, and matching a "}" with the beginning "{". You can uncheck these options if you don't want ROBOTC to perform these tasks.

---

**Debugger:**



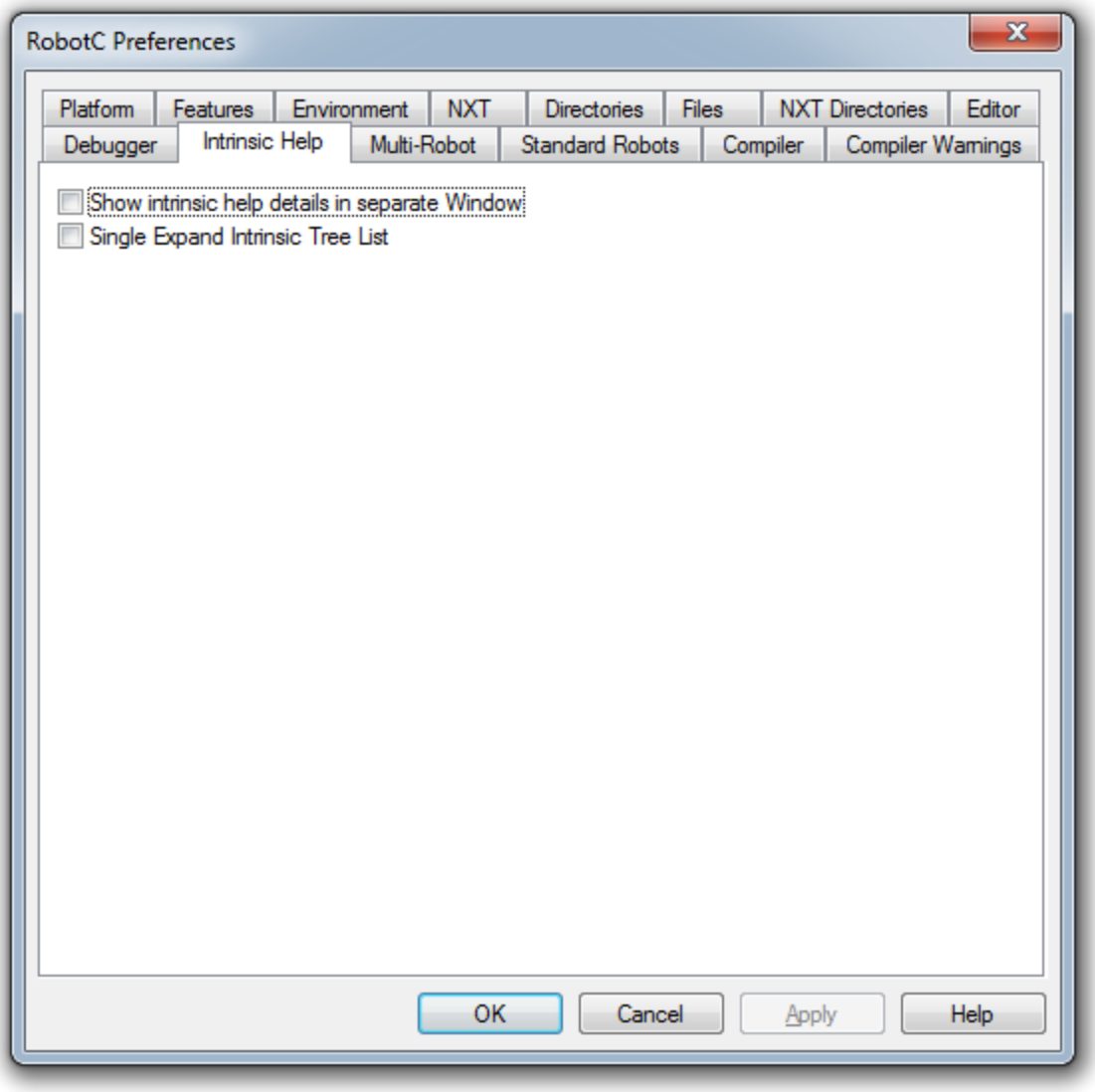

**Continuous Polling on Startup**
Sets polling to Continuous upon startup of the Program Debugger. The Program Debugger window appears automatically when you download a program.

**Delay [milliseconds] between polling cycles**
Sets the delay between polling cycles. The smaller the number, the faster ROBOTC will poll, or read, the values from the NXT. Keep in mind that the faster ROBOTC polls the values, the more resources your computer has to use. Using a small delay may result in a slow down of your computer (depending on the processing power of your computer).

**Maximum duration of polling (minutes)**
Sets the duration of polling. **Need more info**

**Maximum variable words 'polled' per upload**
You can set the maximum number of variables polled per upload. The result of this is shown in the Global Variables window.

## Intrinsic Help:

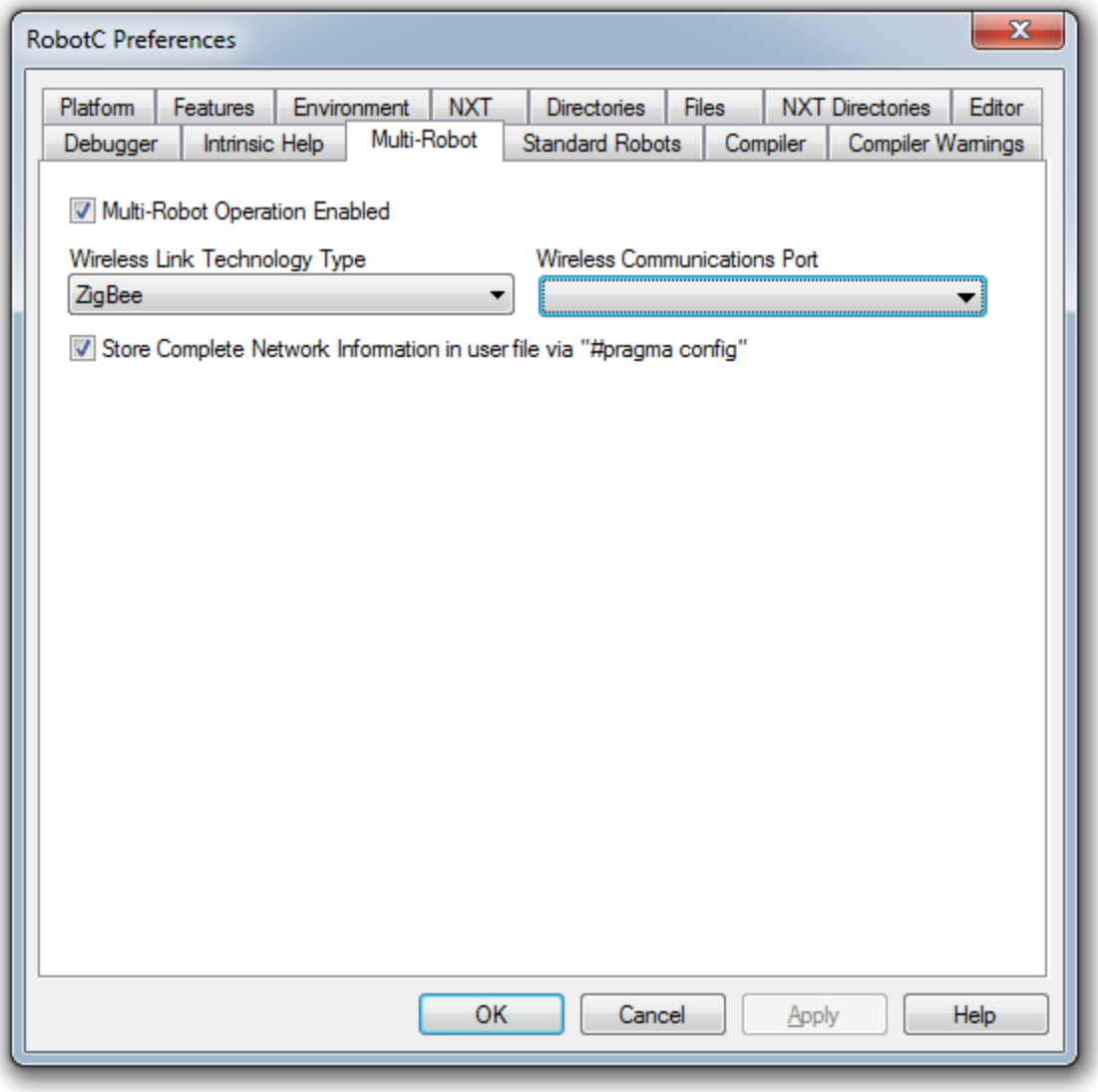



### Show intrinsic help details in separate window
When checked, ROBOTC will open a seperate window that contains the tool-tip text and a listing of all the sample programs that the currently selected code template is used in.

### Single Expand Intrinsic Tree List
When checked, ROBOTC will automatically close the last tree in the code templates when a new tree is opened.

---

## Multi-Robot:




**Multi-Robot Operation Enabled**
Toggle whether or not ROBOTC should enable multi-robot operation.

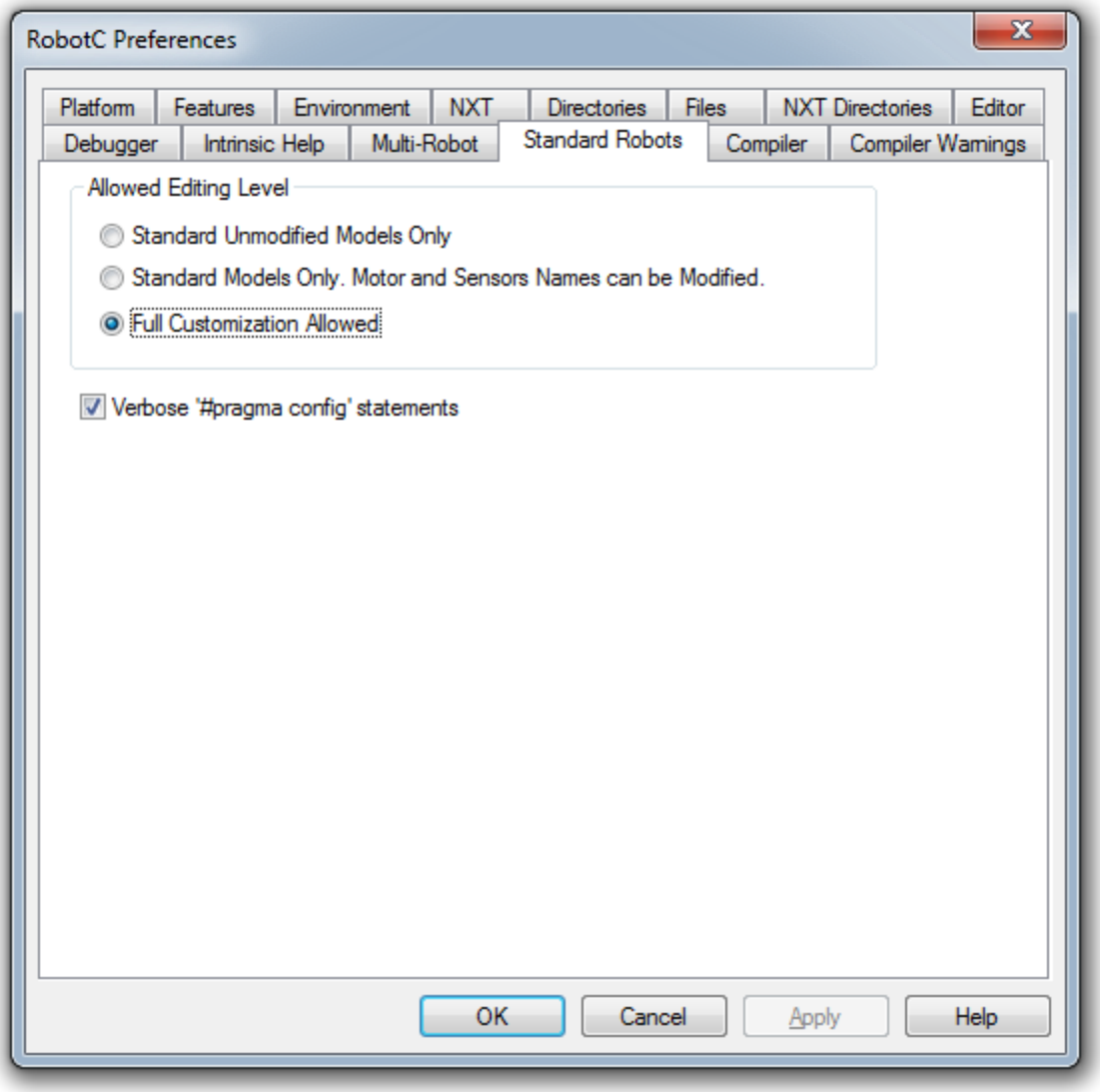
**Wireless Link Technology Type**
Choose how the robots will communicate.  Currently ROBOTC supports WiFi, Bluetooth, and ZigBee radios.

**Wireless Communcations Port**
Select which port to use for communcating with the robot(s).

**Store Complete Network Information in user file via "#pragma config"**
Select this if you'd like to save your setup for future use.

---

**Standar Robots:**

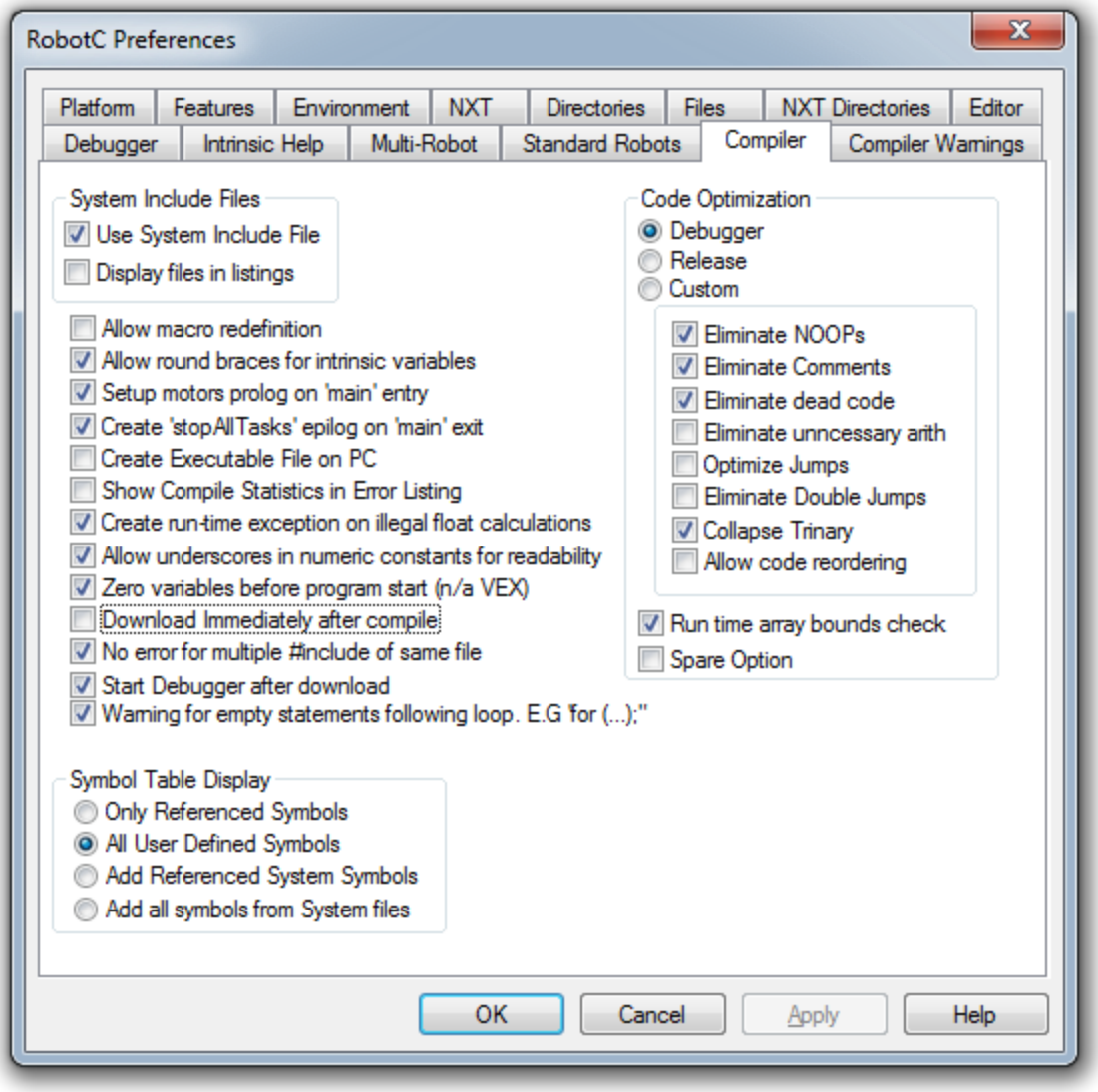



**Allow Editing Level**
Select how much freedom a user has to customize the preconfigured robot models.

**Verbose '#pragma config' statements**
Allow ROBOTC to replace the '#pragma config' statements if they match those of a preconfigured robot model.

---

**Compiler:**



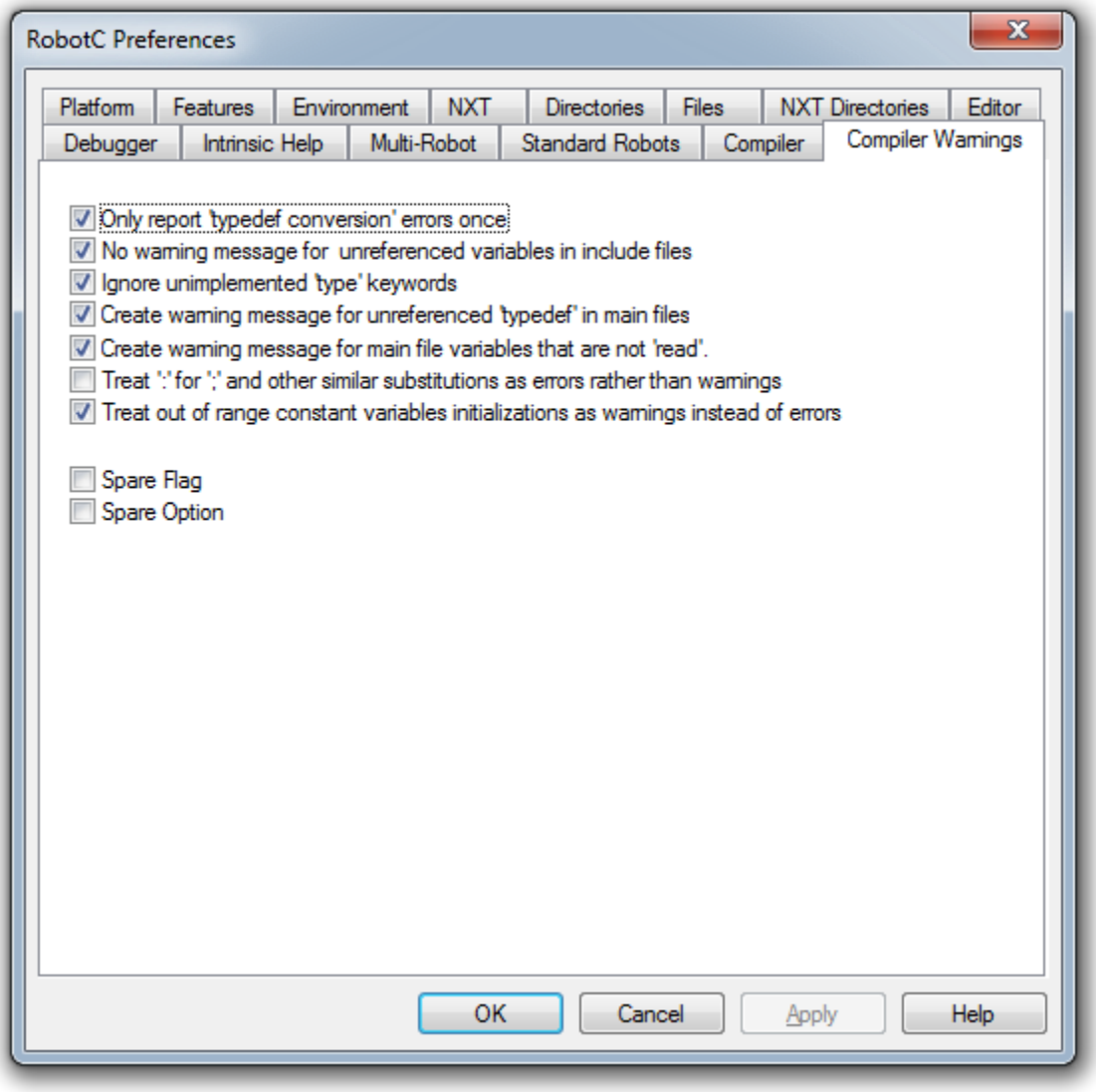

Advanced users may be interested in exploring the various compiler optimization techniques. When the user level is set to “Expert” (see the command “Menu Level” in the “Window” menu), there is a tab in the ROBOTC “Preferences” setup that enables individual control of the individual compiler optimization methods. When the level is set to “Basic” then this tab is hidden from the user.

The picture on the above illustrates this tab. See the “Code Optimization” section on the left of this window.

This screenshot also illustrates several useful ROBOTC options like:

- Automatically opening the “last project” used when ROBOTC is first started.
- Automatically generating “prolog” and “epilog” code that runs at the beginning and end of your program execution to setup the motors and sensors.
- Automatically starting the debugger after every program download.
- The default values for these toggles are the most popular options. Most users will not find a need to modify these values

### Compiler Warnings:

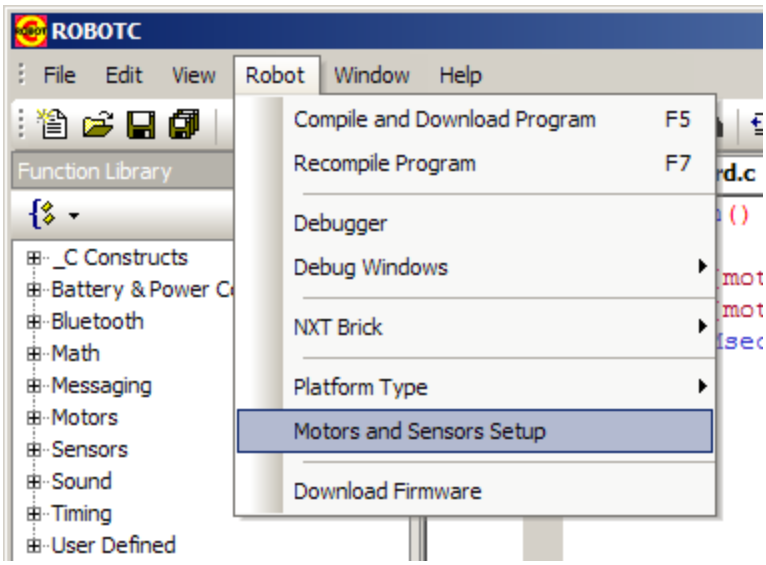



Choose how the compiler warning system behaves.

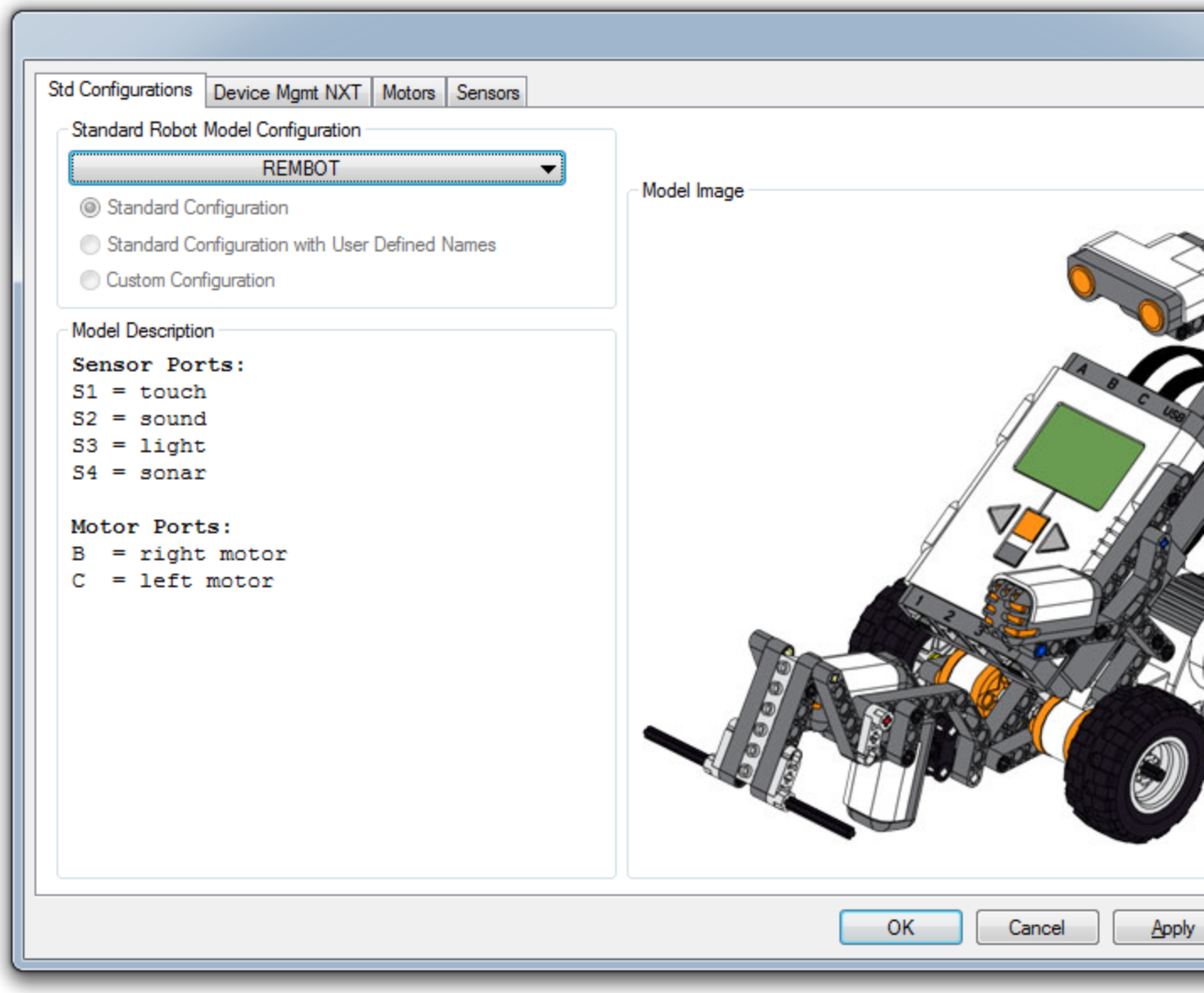
# 4.11 Motors and Sensors Setup

The **Motor and Sensor Setup** window is one of the central points of ROBOTC. This screen is where you can configure all of the inputs and outputs attached to your robot controller. All configurations created in the Motors and Sensors setup screen will only remain persistant for that program. Configurations are saved with your program, so you will not have to reconfigure the Motors and Sensor Setup every time you open that program.

To open the Motors and Sensor Setup screen, click the Robot menu and then select "Motors and Sensor Setup."



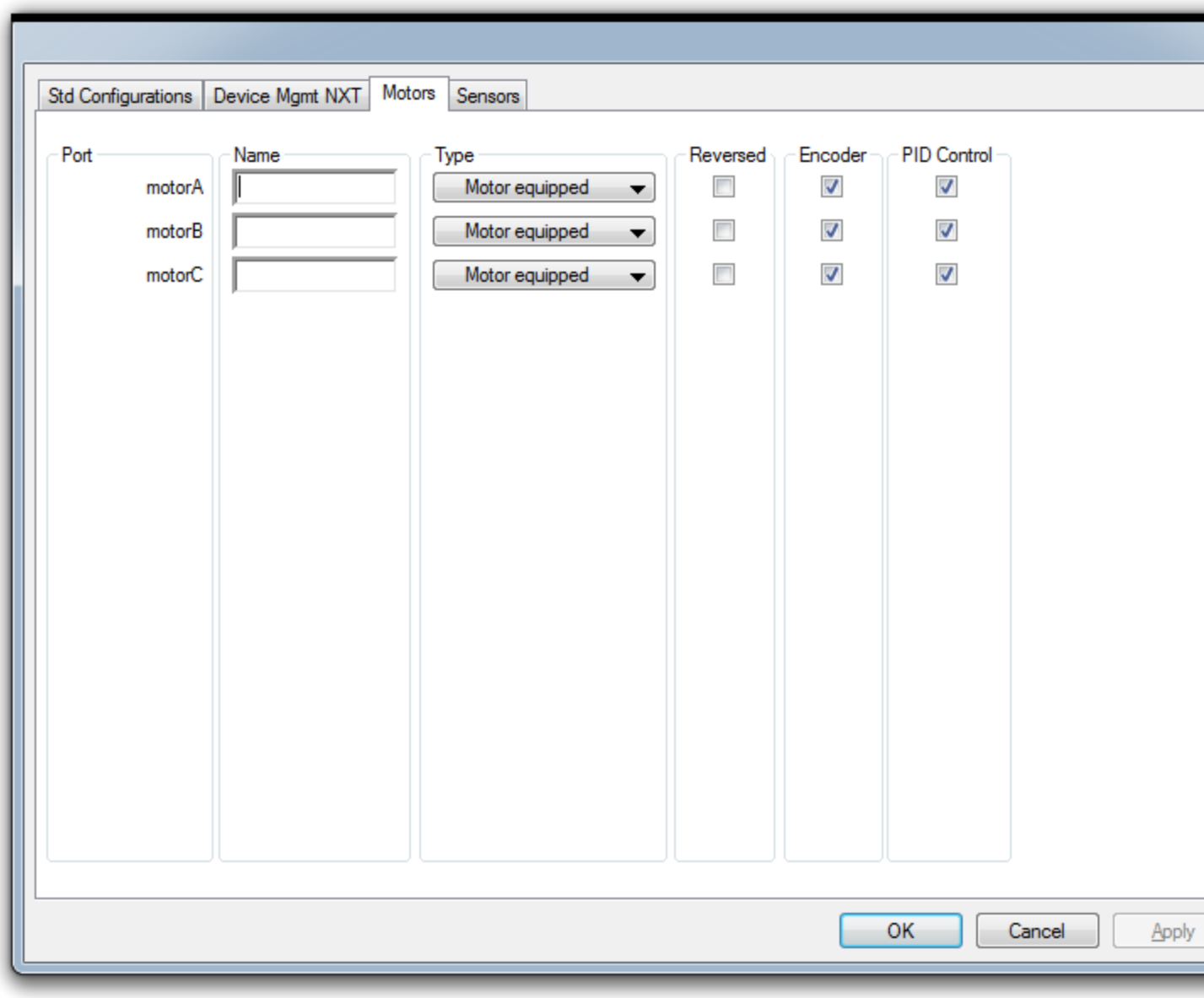

**Std Configurations Tab:**




In this menu, a user can select a preconfigured model for use. This will take care of setting up the motors and sensors automatically to match the preconfigured robot model. You may also select how much freedom a user has in customizing the preconfigured robot models.

---

**Motors Tab:**
The "Motors" tab is used to configure the outputs on your controller, as well as assign aliases to the motors to make your program more readable. There are three parts to the motors page:




**Index**
The index will tell you which ports settings you are currently modifying. motorA = Motor plugged into Port A.

**Name**
The text box for the "name" field allows you to create an alias for that specific motor port. In this example, both motor[motorA] and motor[left_motor] would access the same motor.

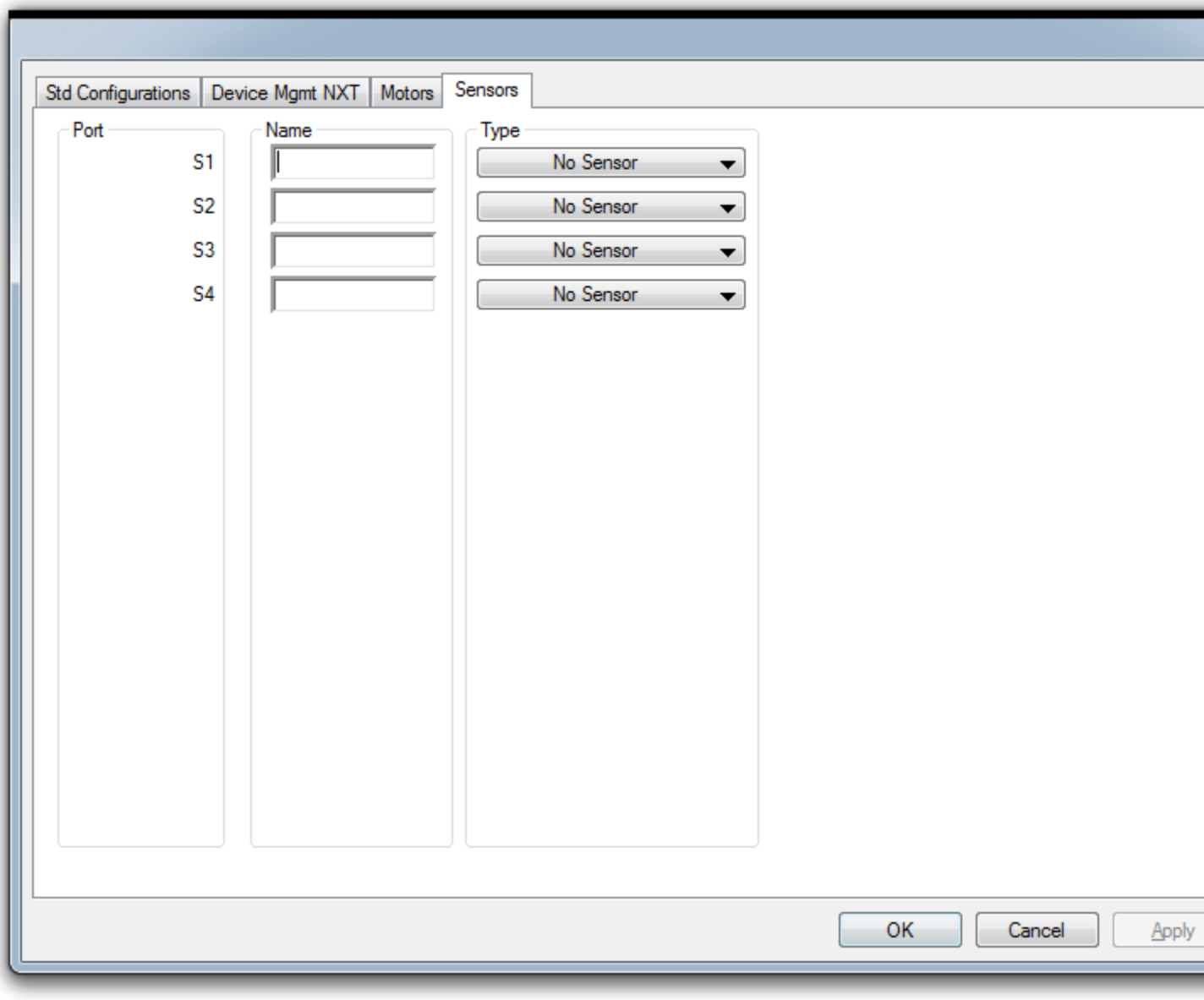
**Type**
The type drop down box is used set the type of motor you want to use. There are four options:

- No Motor (default) - This will assign no special behavior to the motor port selected and will not generated any code.
- RCX Motor - Sets the motor port as an RCX motor with an alias defined by the "name" text box.

- NXT Motor with Speed Ctrl - Sets the motor port as an NXT Motor with Speed Control. Speed control includes the use of PID and syncing motors.

---

**Sensors Tab:**
The "Sensors" tab is used to configure specific sensor types. ROBOTC will take care of most of the background tasks for using sensors (such as calibration/scaling for light sensors and creating a counter for rotation sensors), so it is highly advisable to use the sensors portion of the Motors and Sensor Setup screen.




**Index**
The index will tell you which port's settings you are currently modifying. S1 = Sensor Port 1.

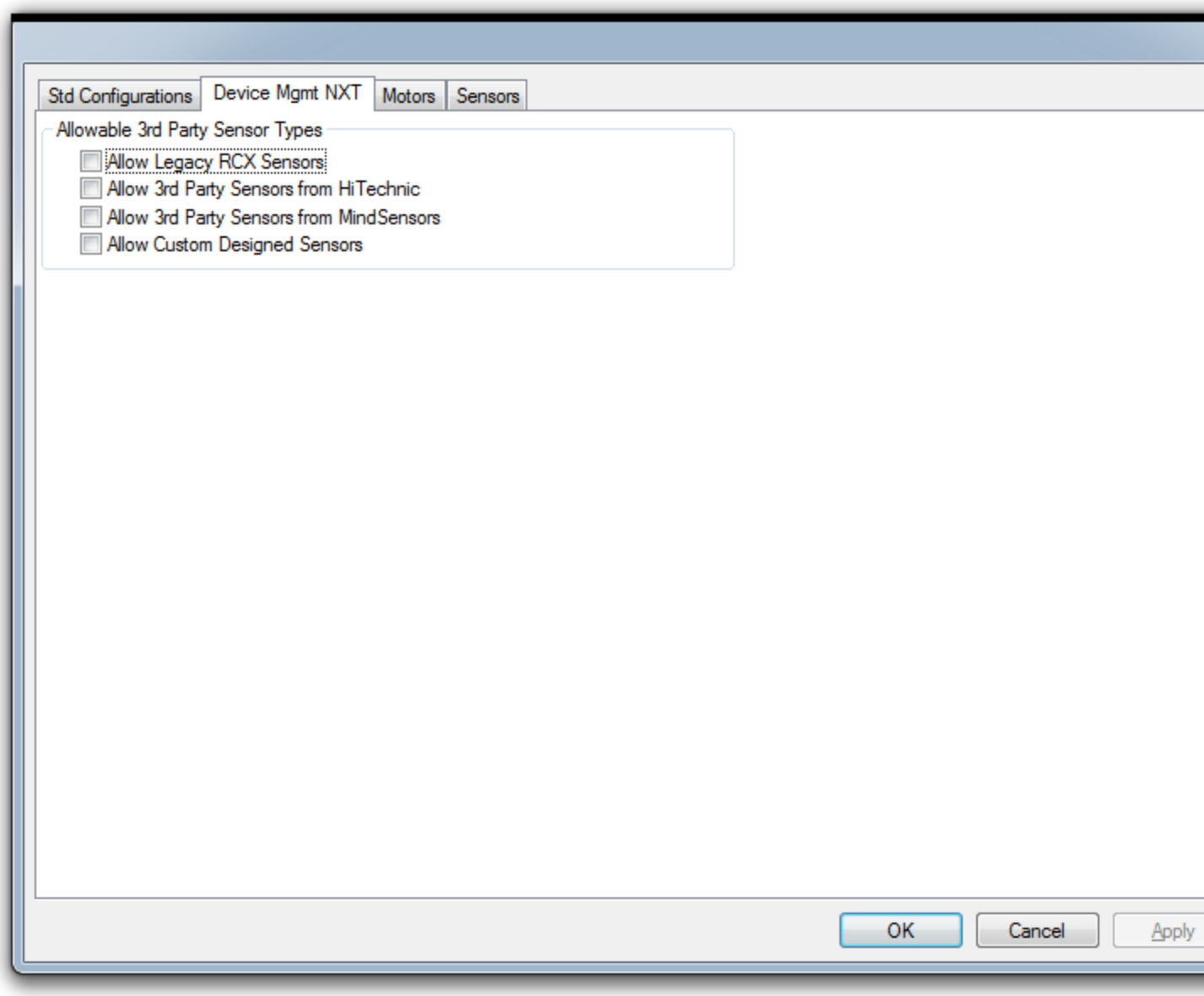
**Name**
The text box for the "name" field allows you to create an alias for that specific sensor port. In this example, both SensorValue[in1] and SensorValue[touch1] would access the same sensor.

**Type**
The type drop down box is used set the type of Sensor you want to use. There are multiple options:

- No Sensor (default) - This will assign no special behavior to the sensor port selected and will not generated any code.
- Raw Value - Returns an analog value between 0 and 1023 - Used for testing new sensors or reading raw data from sensors.
- Touch - Returns a digital value. A "1" means a closed circuit and a "0" means an open circuit.
- Light Active - Returns an analog percentage value between 0 and 100. "Active" means that the LED is turned on to emit light onto a surface to be reflected back to the sensor.
- Light Inactive - Returns an analog percentage value between 0 and 100. "Inactive" means that the LED is turned off, and the light sensor relies on ambient light.
- Sound DB - Returns the measurement of sound intensity over the standard threshold of hearing.
- Sound DBA - Returns the measurement of sound intensity with an "A" contour filter. The filter adjusts the measurement to account for the way inw hich the ear responds to different frequencies of sound.
- SONAR - Returns an analog value in inches (i.e. a value of 20 means 20 inches away). A value of "-1" means the sensor does not receive a "reflection".

---

**Device Mgmt NXT / Advanced Sensors:**
The Sensors Tab is used to configure specific sensor types. When ROBOTC is set to the "Expert" setting, more sensor types are available. The visiblty of these advanced sensors is controlled by the "Device Mgmt NXT" tab.




**Allow Legacy RCX Sensors**
Enables the legacy RCX sensors, such as the RCX Light Sensor, RCX Touch Sensor, RCX Temperature Sensor and RCX Rotation Sensor.

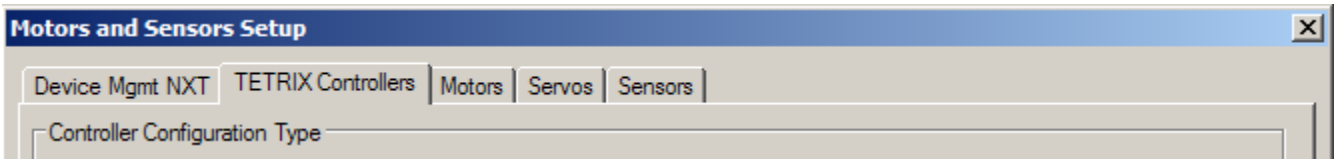
**Allow 3rd Party Sensors from HiTechnic**
Enabled the built in support for the HiTechnic sensors such as the Accelormeter, Gyro, Compass and Color Sensors.

**Allow 3rd Party Sensors from MindSensors**
Enabled the built in support for the HiTechnic sensors such as the Accelormeter and Compass Sensors.

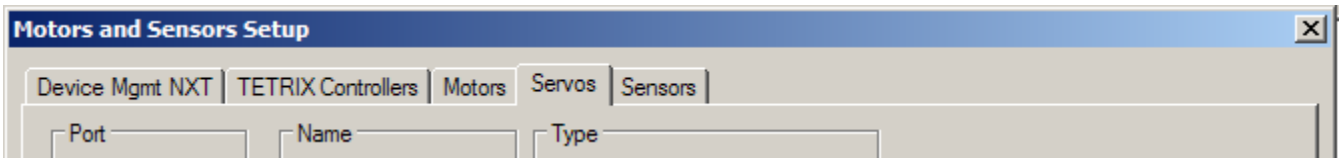
**Allow Custom Designed Sensors**
Enables the built in custom I2C sensor types for low and high voltage support and slow and fast I2C message support.

**TETRIX Specific Tabs:**
Selecting "NXT + TETRIX" as the Platform Type adds two additional tabs to the Motors and Sensors Setup window.

**TETRIX Controllers:**
Configure the TETRIX DC Motor and Servo Controllers connected to the NXT using a simple, interactive interface. Configuring DC Motor Controllers will add additional entries on the Motors tab. See more information here.



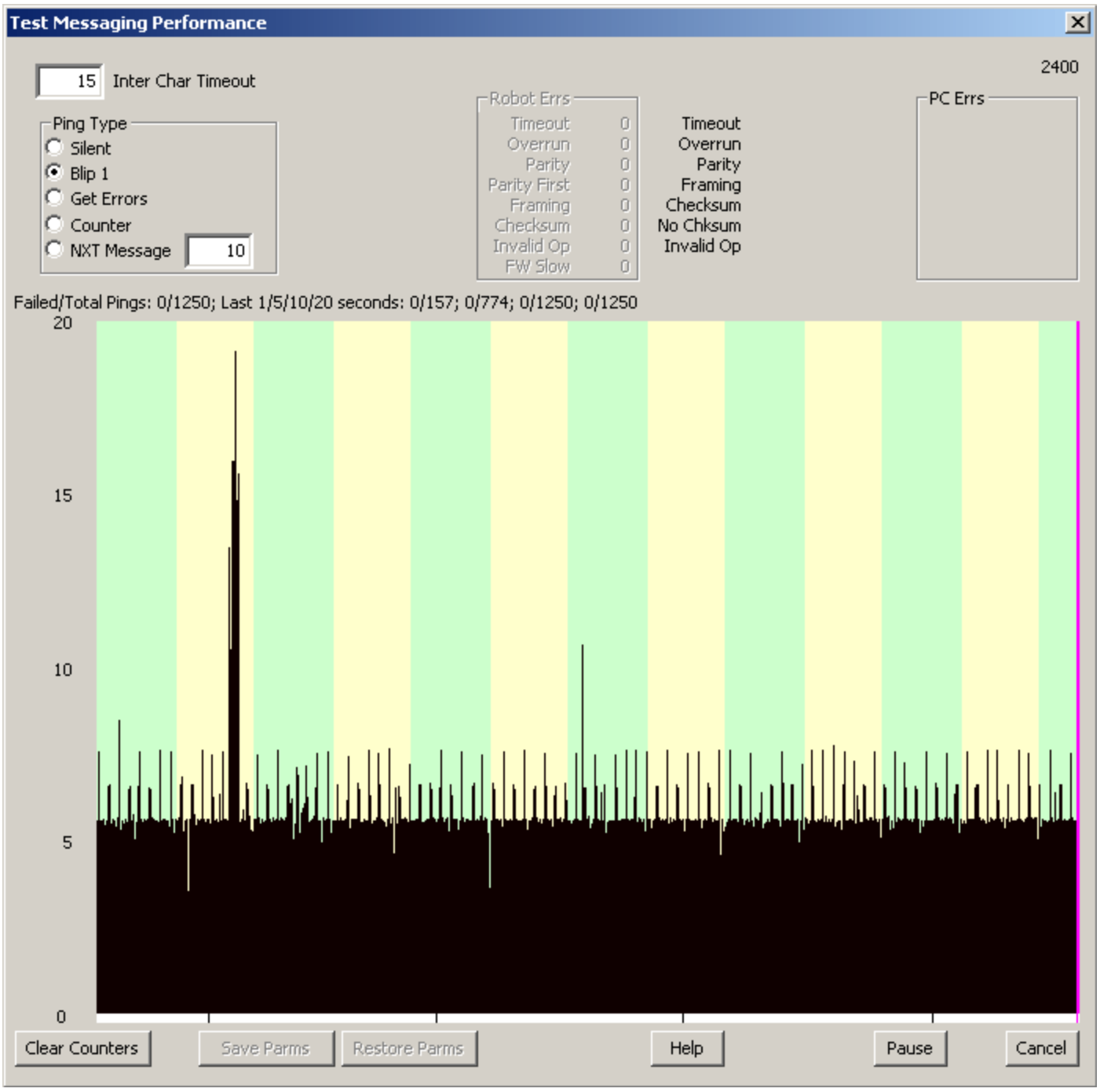

**Servos Tab:**
Configuring Servo Controllers will add entries on the TETRIX-only Servos tab. See more information here




# 4.12 Test Message Link

The Test Message Link screen is used to test how quickly a packet of data can be sent to the NXT and have the NXT respond that it recieved the packet. ROBOTC will produce graphical data about the last 160 data points and statistics about all pings sent from the start of the test.

ROBOTC by default is set to make the NXT play a Blip sound to denote a packet being recieved. You can change the type of ping message to silent, play a blip, get errors only from the nxt and product a counter of the number of packets. You can also use the "NXT Message" to send a 1-byte message to the NXT (0-255). The NXT Message ping type is the only setting that is compatible with both the ROBOTC firmware and the NXT-G firmware.




# 5. Bluetooth and the NXT

## 5.1 Bluetooth Overview

Bluetooth (BT) is an industry standard short-distance (up to 10 meters) wireless communications protocol operating at 2.4 GHz. It can optionally use a higher power transmission to achieve distances up to 100 meters. The NXT utilizes the 10 meter option.

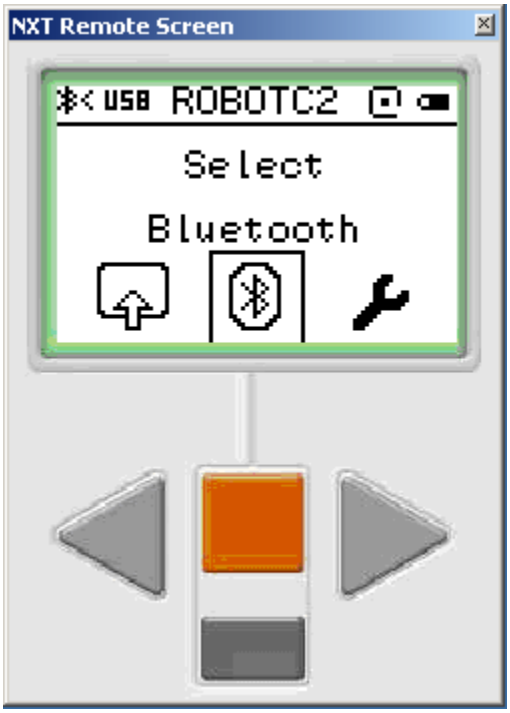
BT includes not only the low-level radio transmission (at 2.5 GHz) but also several higher layer message protocols (or profiles) designed for different applications. There are over 25 different BT profiles currently defined. Some of the more popular profiles include:

- The Serial Port Profile (SPP) is used to provide wireless emulation of a conventional RS-232 serial communications cable. This is the only protocol supported on the NXT.
- The Human Interface Device (HID) protocol is used for communication on wireless keyboards and mice. It is also used on some game controllers like the Nintendo Wii or SONY Playstation 3.
- The Headset Profile (HSP) is used to connect wireless headsets to devices like cellphones.

In order to connect two devices via BT, the devices must not only support BT but also support the type of profile that will be used for the connection. The NXT only supports the SPP so that it cannot, for example, directly connect to a Nintendo or Sony game controller.

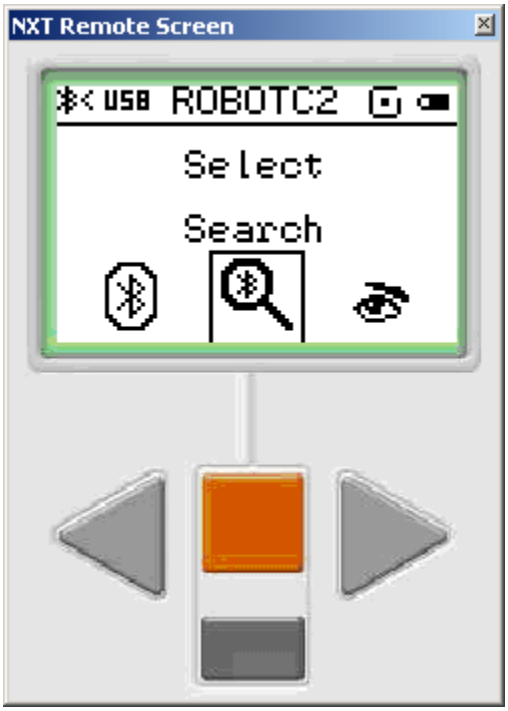
# 5.2 Connecting Two NXT Bricks

1. Use the NXT user interface to select Bluetooth commands.

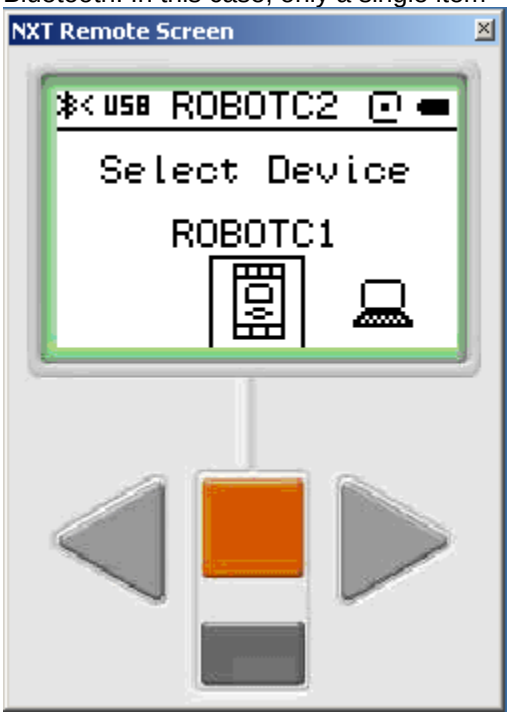



2. The top left of the NXT LCD status display shows the Bluetooth status. The leftmost icon  is the BT symbol and indicates that BT is enabled on the NXT. The icon indicates that the NXT BT visibility is enabled -- if visibility is disabled, then the NXT will not respond to search commands from other BT devices. "ROBOTC2" is the friendly name for the NXT.

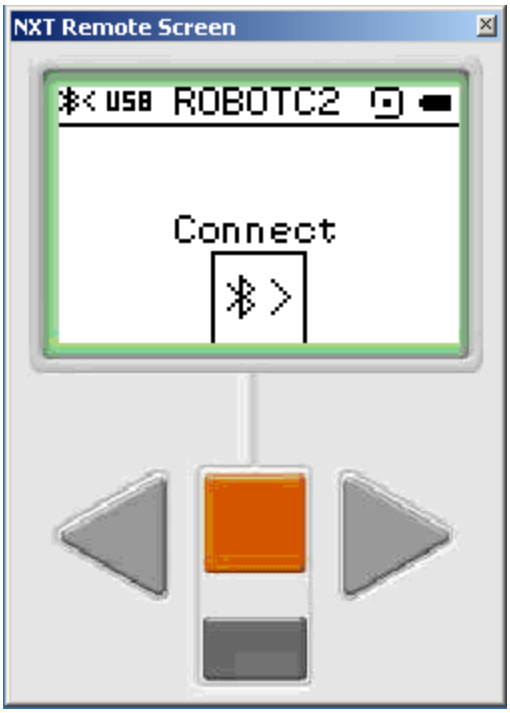
3. Then select "Search" command and run the command.



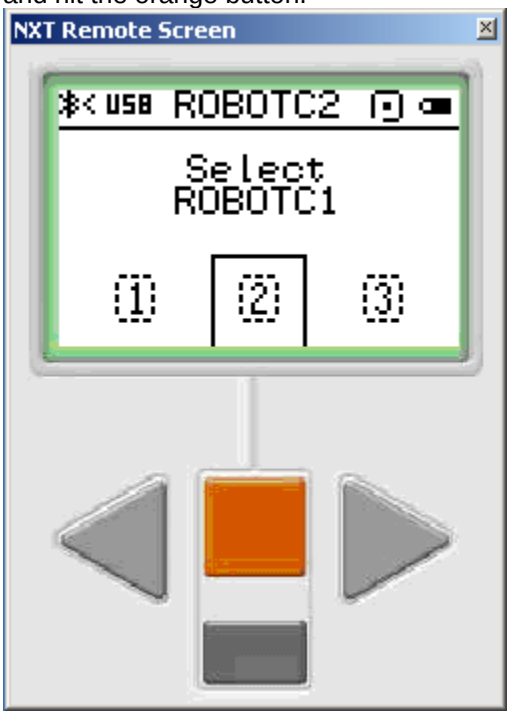

4. Once search has been performed, you'll be presented with a menu of the NXTs that were found via Bluetooth. In this case, only a single item -- the device "ROBOTC1" was found.



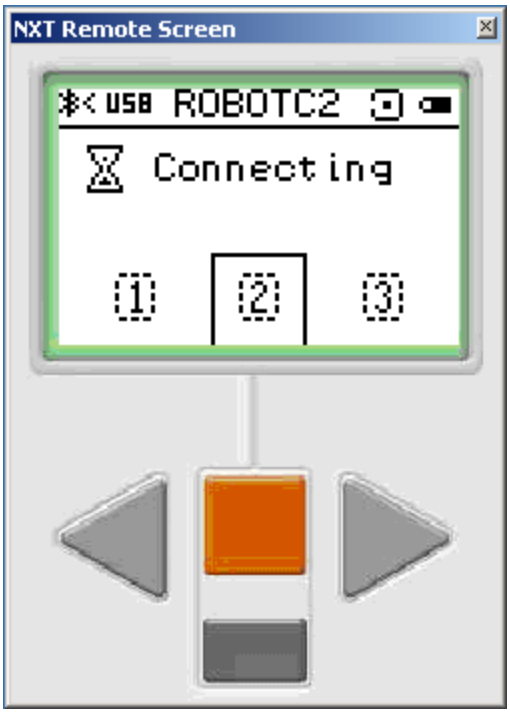

5. Select the target device for the connection from the search results -- i.e. "ROBOTC1" -- and then select "Connect"



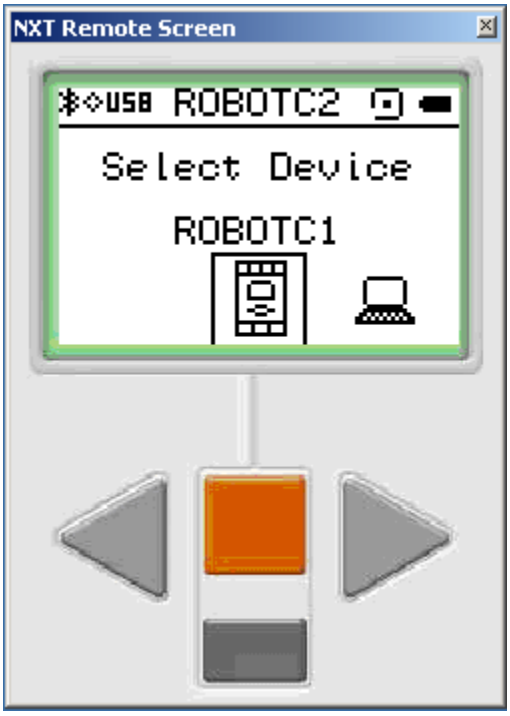

6. The NXT supports connections to up to three "slave" devices. You need to select the appropriate "slot" (1, 2, or 3) for the connection. Select an empty slot -- all three slots are empty in the following picture -- and hit the orange button.




7. The NXT screen will show "Connecting" as the connection is attempted. Depending on the settings on your NXT, you may be prompted for a password.



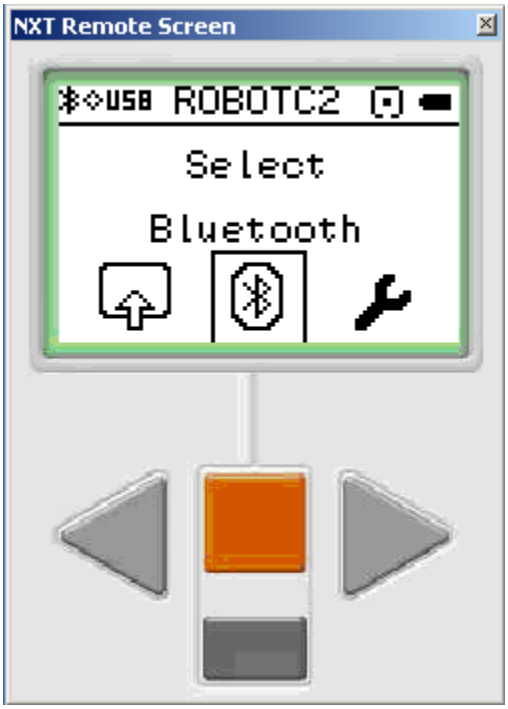

8. Once the connection is made, the status ICON will change to "<>". The ">" indicates that the NXT is connected to another BT device.



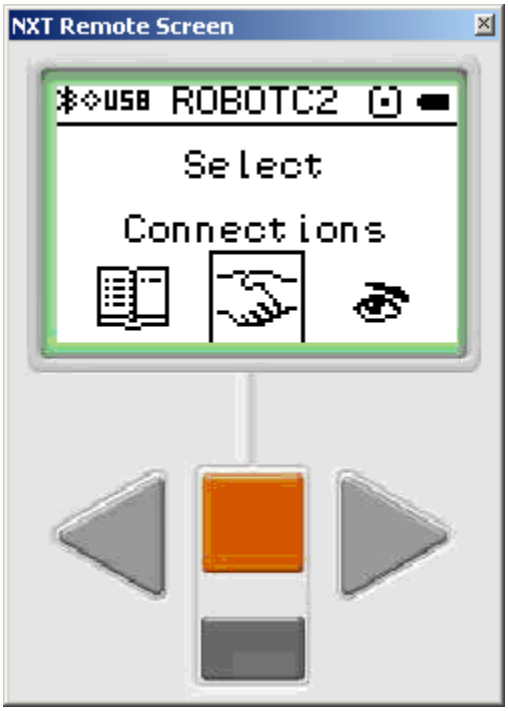

# 5.3 Disconnecting Two NXT Bricks

Disconnecting a Bluetooth connection is even easier than making a connection.

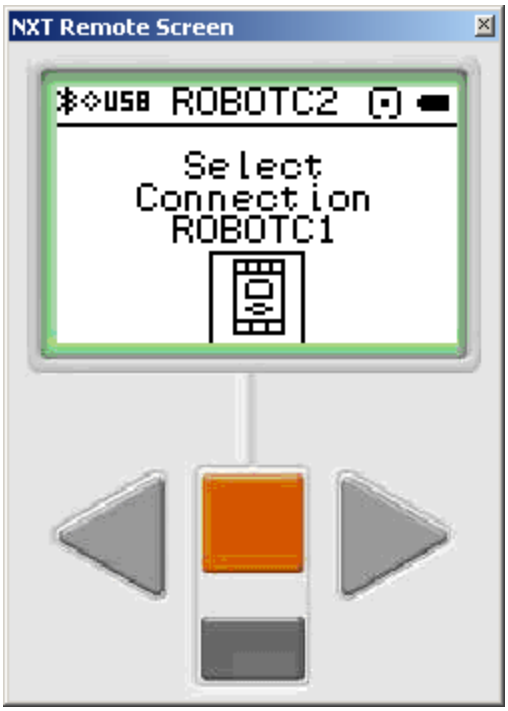
1. Use the NXT user interface to select Bluetooth commands.



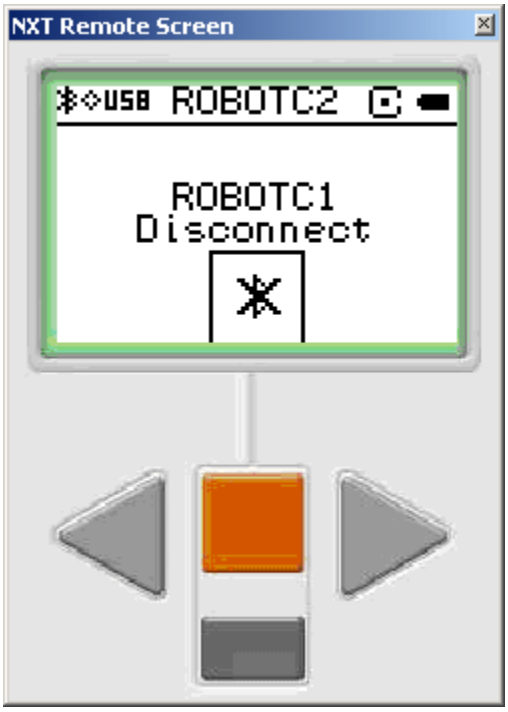

2. Select the "Connections" Menu item.



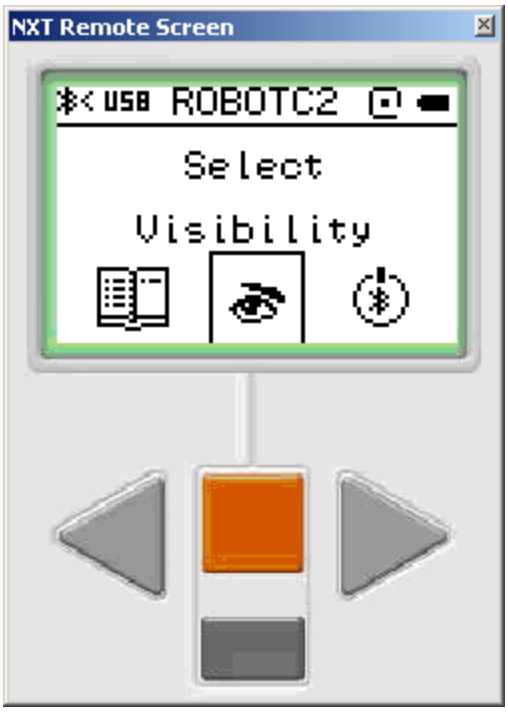

3. You now get a list of the devices connected via Bluetooth. In this case, there is a single connection to the "ROBOTC1" Bluetooth device.




4. Select this item. Then select the "Disconnect" command.




5. The connection will be removed and the NXT GUI will return to the Bluetooth commands menu. Note that the Bluetooth status icon in the top-left of the screen has changed from "<>" (visible + connected) to "<" (visible + not connected).

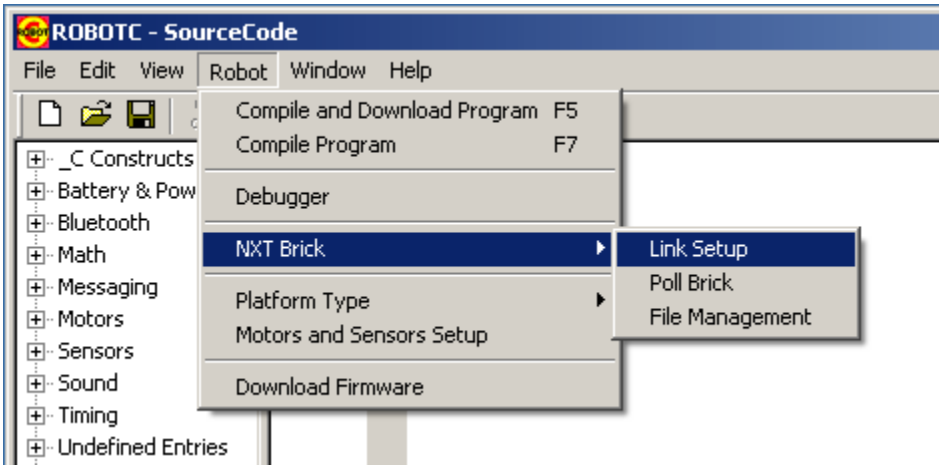



# 5.4 PC/NXT Pairing with ROBOTC

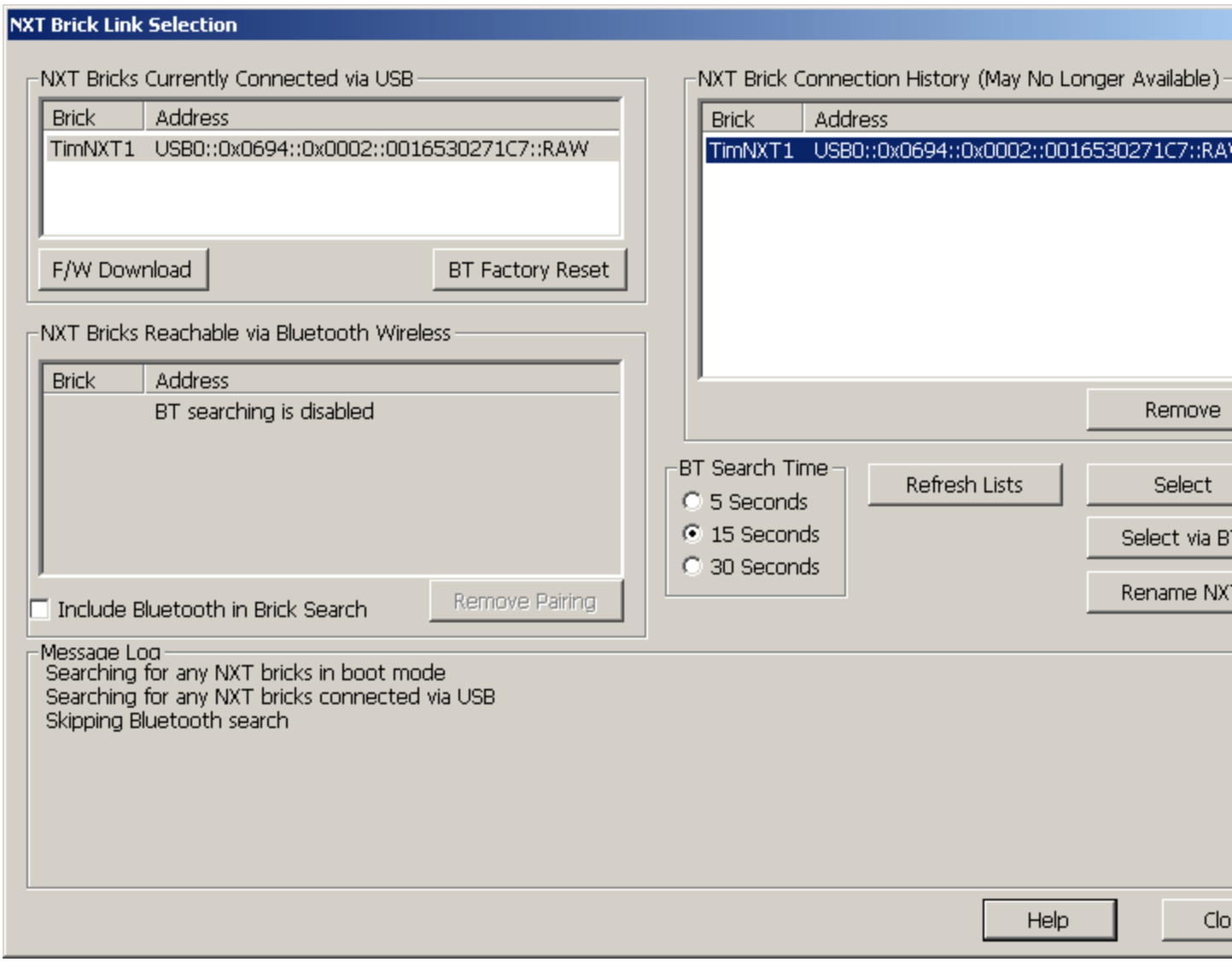
To pair your NXT with your Bluetooth adapter to use ROBOTC, follow these steps.

Note: You cannot download firmware to an NXT over Bluetooth. Please download ROBOTC's firmware before trying to connect via Bluetooth.

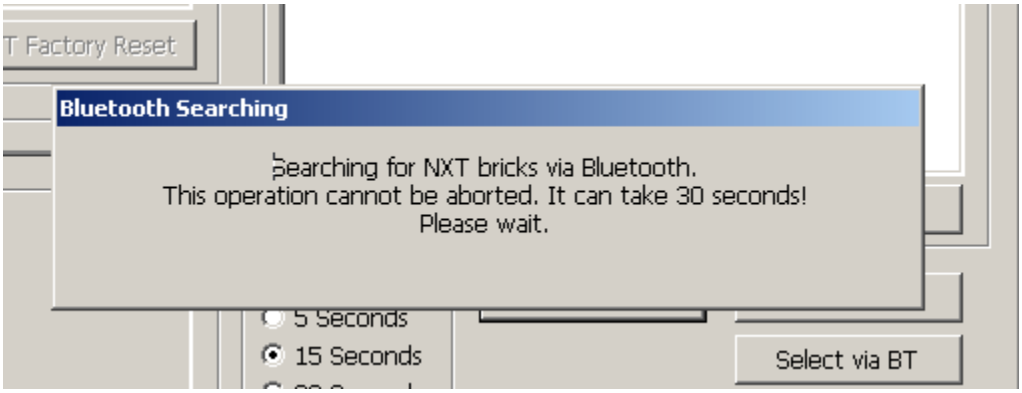
1. Go to the "Robot" menu, and under the "NXT Brick" submenu, select "Link Setup."



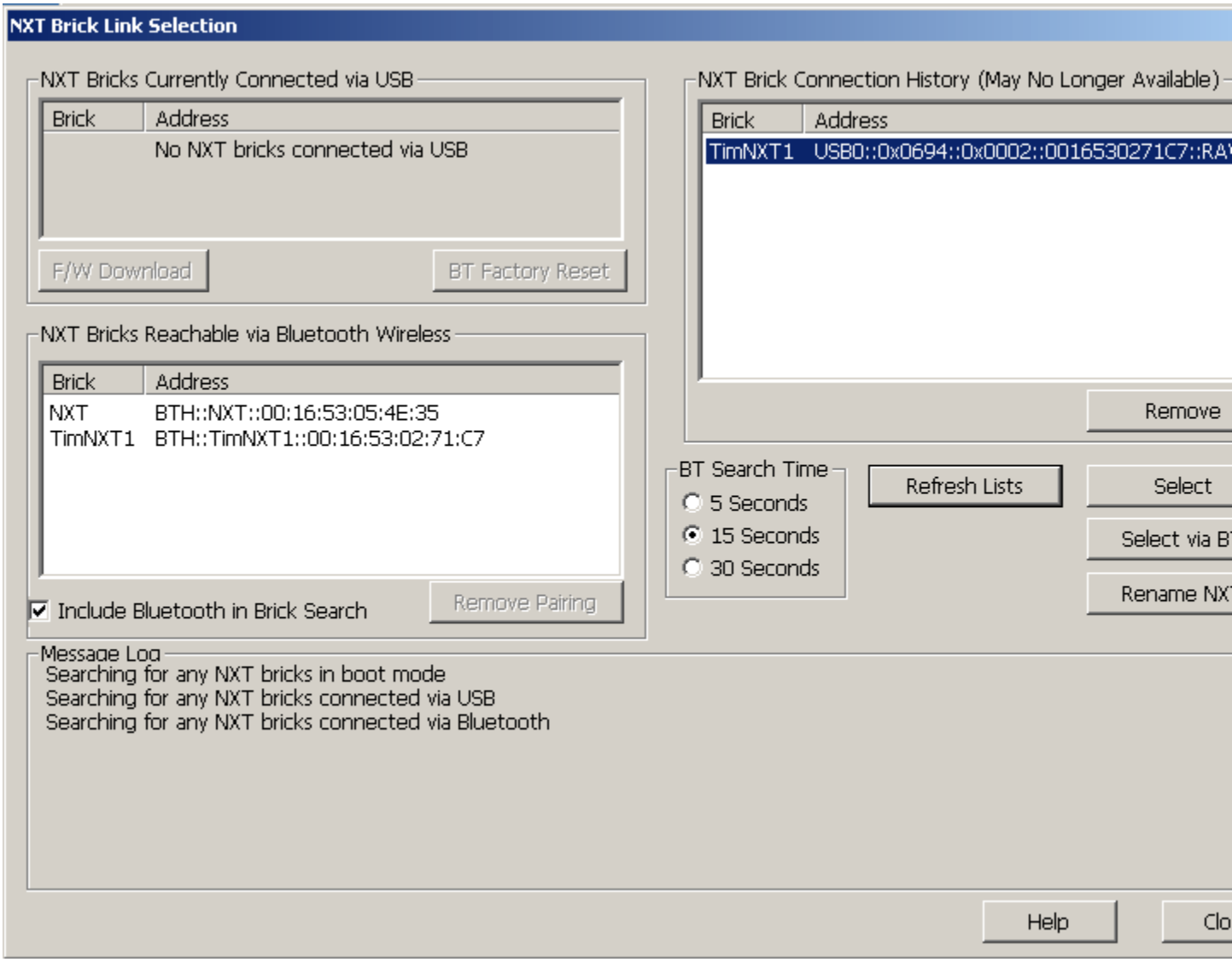

2. In the NXT Brick Link Selection screen, we want to enable Bluetooth seaching. Click the checkbox labeled "Include Bluetooth in Brick Search" and then click the "Refresh Lists" button.




3. ROBOTC will use the bluetooth adapter to search for NXT bricks. This may take 30 seconds.

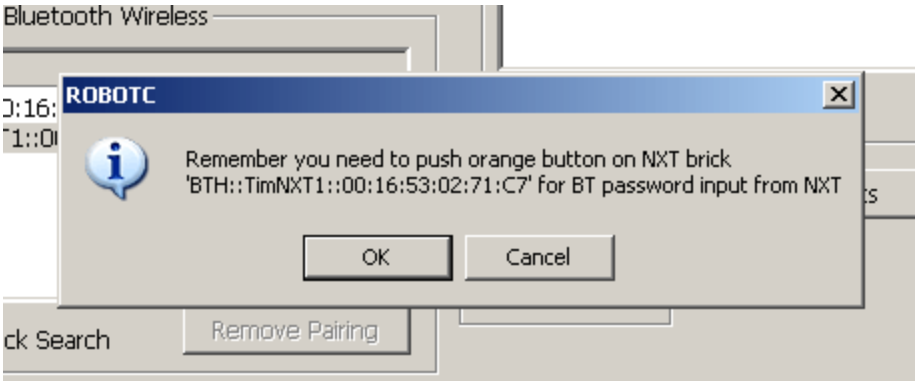


4. When ROBOTC is competed searching for Bricks, you can select which brick you would like to connect to. Click the name of the brick under the "NXT Bricks Reachable via Bluetooth Wireless" menu and then click the "Select" button on the right.

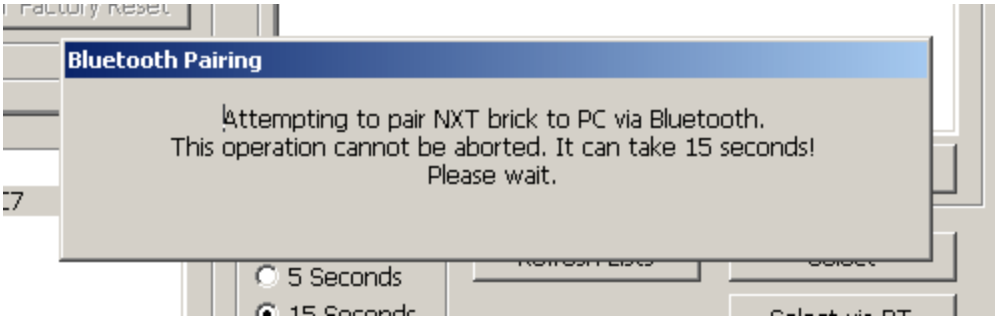



5. ROBOTC will remind you to press the orange button on the NXT when the NXT make a sound. Click "OK" to continue.

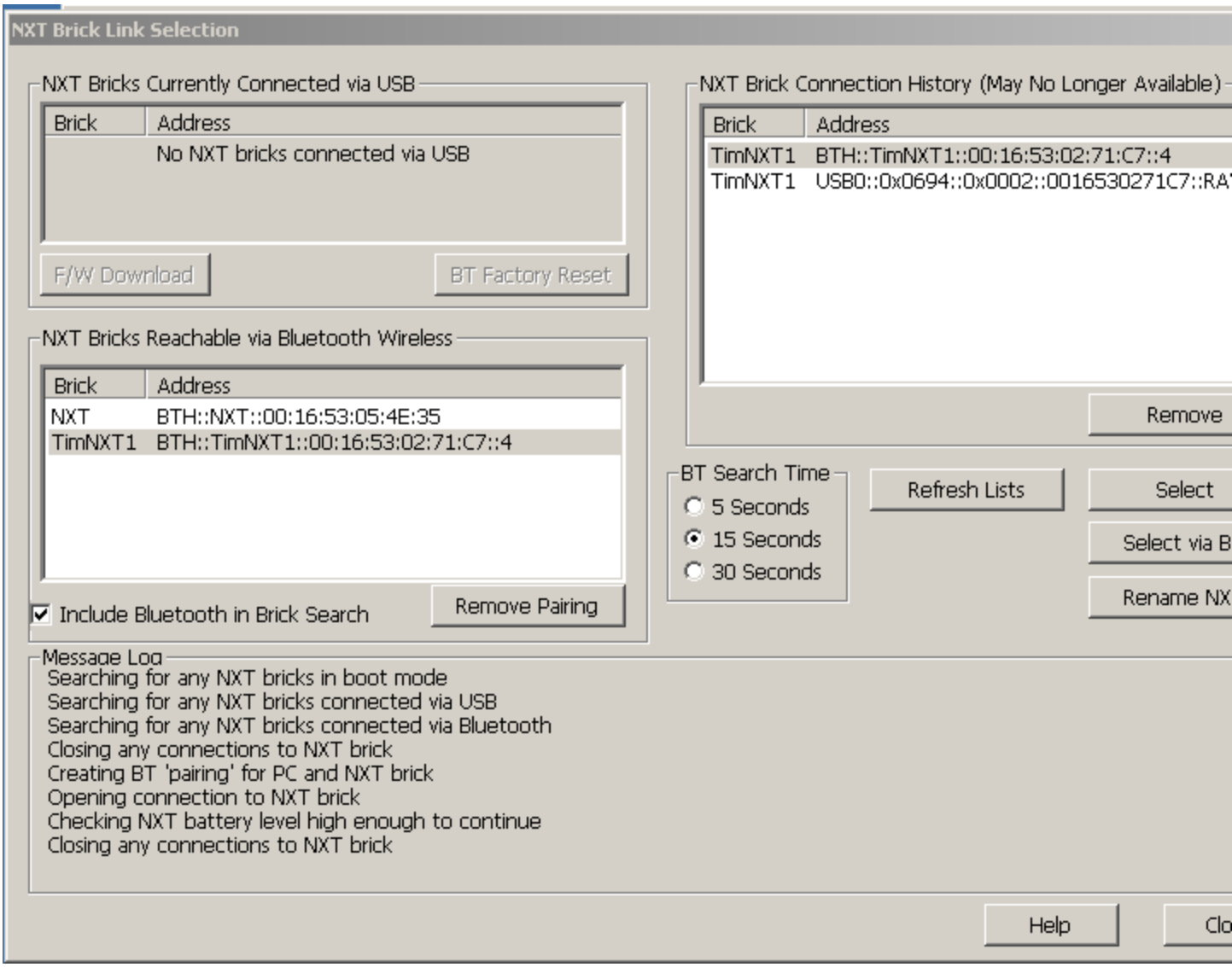
This is the NXT requiring input for the Bluetooth passcode. The default passcode will automatically be entered, so just press the orange button.




6. ROBOTC will start to pair with the NXT. This may take 15-30 seconds and you will have to press the orange button once during this time.




7. Once the brick has connected via Bluetooth, you will see the brick's name appear under the "NXT Brick Connection History". Your NXT is now connected via Bluetooth! You can continue to use ROBOTC as if your brick was connected via USB... everything will work, including the debugger and program downloading. Click "Close" to exit this screen.




# 5.5 Connecting via Bluetooth - Pairing

Each BT device is identified by a unique 12 hexadecimal digit address. It's a little awkward to refer to devices with this address so the BT protocols include a "friendly name" that is a more conventional 15-character string. The default friendly name for a NXT brick is "NXT" but you can modify this using the NXT's on-brick user interface, or from the ROBOTC IDE, or directly from within a ROBOTC program.

Before you can connect two devices via BT they must be "paired". The pairing process exchanges messages between the two devices where they share their BT address, their friendly names and their supported profiles. The devices confirm that they support the same profiles and that the (optional) password matches. Once two devices have been paired, you can make subsequent connections using just the friendly names. The devices remember the passwords so that they don’t have to be re-entered each time

There's a lot of low-level messages used in establishing a paired connection. Fortunately, this is hidden from the user on the NXT with a simple user interface via the user interface on the NXT.

From the BT menu on the NXT, a search is performed to build a internal list in the NXT of all the devices that are currently available via BT. The NXT simply broadcasts a "who's available" message over BT and collects a list of all the devices that replied.

To learn how to connect two NXTs together, please see the "Connecting NXTs" page here.

Once connected, the "slave" device will be added to the NXT's internal table of paired devices, i.e. the "My Contacts" list on the NXT. Next time you want to make a connection you can select the device from the "My Contacts" and avoid the 30-second search.

NOTE: Sometimes you'll find that you cannot get two previously paired devices to connect. One cause of this is that one device is no longer powered on. Another cause is that somehow the devices have got out of sync on the pairing status where one device has "lost" the paired status. If this happens, try removing both devices from the "My Contacts" list on each NXT and restart the connection using the "Search" function.

# 5.6 Sending Messages via Bluetooth

Once you have two NXTs connected via BT, it is very easy to send messages between them. There are only three functions that are needed:

1. **`cCmdMessageWriteToBluetooth`** writes a message.
2. **`cCmdMessageGetSize`** get the size of the first message in a mailbox containing a queue of received messages.
3. **`cCmdMessageRead`** removes the first message from a mailbox queue and copies it to a user buffer.

After calling each of the above functions you should check the returned value to determine the success/failure of the function.
Use the function "**`cCmdMessageWriteToBluetooth(nQueueID, nXmitBuffer, nSizeOfMessage)`**" to send a BT message to the far end NXT. Check the error code to make sure message was transmitted successfully. This sends the message (up to 58 bytes in length) in the variable **`nXmitBuffer`** to queue or mailbox number **`nQueueID`** on the far end NXT. **`nSizeOfMessage`** is the length of the message.

When messages are received over BT by a NXT they are automatically added to the end of one of the 10 message or mailbox queues. Use the function "**`cCmdMessageGetSize(nQueueID)`** " to determine whether any messages have been received. A positive return value indicates that a message was received and is the number of bytes in the message.

Use the function "**`cCmdMessageRead(nQueueID, nRcvBuffer, nSizeOfMessage)`**" to retrieve the first message from the specified mailbox and copy it to a user's buffer at **`nRcvBuffer`**. Only the first **`nSizeOfMessage`** bytes of the message are copied. **`nQueueID`** is the mailbox number to obtain the message from.

The sample program "NXT BT Messaging No Error Checking.c" is a simple program to show how to use all three of the functions.

# 5.7 Simple Bluetooth Messaging

The NXT has a very powerful communication capability between two NXTs using the wireless Bluetooth functionality built into every NXT. The "Messaging" functions described here provide an easy approach for using this communication. It limits the communication to messages containing three 16-bit parameters.

It is very easy to set up Bluetooth communications between two NXTs.

- Use the NXT's on-brick user interface to make a connection between two NXTs.
    - It is also possible to set up the connections within a user's program but that is for advanced users.
- Use the sendMessage(nMessageID)and sendMessageWithParms(nMessageID, nParm1, nParm2) functions to send outgoing messages.
- Use the variables message and messageParms to retrieve the contents of a message. When you've finished processing a message, use the ClearMessage() function to "zero out" the message so that your program can process subsequent messages. The next time your program references the message variable, it will be for the next message that has arrived.

---

## Sending Messages:

There are two functions for easily sending messages. One sends a single 16-bit value and the other sends three 16-bit values. Either of the two NXTs can send messages at either time. It is the responsibility of the user program to not send messages too frequently as they may cause congestion on either the Bluetooth link or overflow of the NXT's transmit and receive queues.

**`sendMessage(nMessageID)`**

Sends a single 16-bit word message. 'nMessageID' should range in value form -32767 to +32767. Message value 0 is invalid and should not be used. It is a special value to indicate "no message" received when using the "message" variable.

**`sendMessageWithParm(nMessageID, nParm1, nParm2)`**

This function is identical to the sendMessage function except that the message contains three 16-bit values. This is useful in easily sending separate items of information in a single message. Do not use a value of zero for 'nMessageID'.

---

## Receiving Messages:

The NXT firmware automatically receives messages and adds them to a queue of incoming messages. The application program takes the messages from this queue and processes them one at a time. The variables message and messageParm contain the contents of the current message being processed. The function ClearMessage discards the current message and sets up to process the next message.

**`Message`**

This variable contains the 16-bit value of message received over the Bluetooth channel. It has a range of -32767 to 32767. A value of zero is special and indicates that there is "no message". Whenever the value is zero and the message variable is accessed, the firmware will check to see if it has any received messages in its queue; if so, it will take the first message and transfer its contents to the message and messageParms variables. These two variables will continue to contain the message contents until the user's program indicates it has finished processing the message by calling the ClearMessage() function.

**messageParm[]**
Array containing optional message parameters (up to 3 16-bit words) for messages received over the RCX infrared channel. messageParm[0] is the same as message messageParm[1] and messageParm[2] are additional 16-bit values.

**bQueuedMsgAvailable()**
Boolean function that indicates whether a unprocessed message is available in the NXT's received message queue. This is useful when multiple messages have been queued and your program wants to skip to the last message received. Your program can simply read and discard messages as long as a message is available in the queue.

**ClearMessage()**
Clears the current message. The next time the message variable is accessed, the firmware will attempt to obtain the first message from the queue of messages received by the NXT.
Do not send messages faster than about one message per 30 milliseconds or it is possible for some messages to be lost.

---

## Skipping Queued Messages:

A typical application might have one NXT send a message to the NXT on a periodic basis. For example, it might send a message containing the current values of sensors S1 and S2 every 100 milliseconds. Due to processing delays in the receiving NXT, several messages may have been queued and your program may want to rapidly skip to the last message received. The following are two code snippets that show how this might be accomplished.

**The code in the NXT sending the sensor values:**

```
while (true)
{
  sendMessageWithParm(SensorValue[S1], SensorValue[S2], 0);
  wait1Msec(100);  // Don't send messages too frequently.
}
```

**The code in the receiving NXT:**

```
while (true)
{
  //
  // Skip to the last message received
  //
  while (bQueuedMsgAvailable())
  {
    word temp;
    ClearMessage();    // We're ready to process the next message
    temp = message;    // Obtain the next message
  }
  if (message == 0)
  {
   // No message is available to process
    wait1Msec(5);
    continue;
```

```
  }
  // A message is ready to be processed
  remoteSensor1 = message;          // the value of 'S1' from the remote NX
  remoteSensor2 = messageParm[1];   /* the value of 'S2' from the remote NX
.                    */
                                    /* user code to process the message. It
delays.    */
}
```

---

### Bluetooth Messaging is Incompatible with Debugging Over BT Link

NOTE: The ROBOTC IDE debugger can operate over either USB or Bluetooth connection to the NXT. When an application program uses the Bluetooth connection to send messages to another NXT then you cannot use the Bluetooth debugger connection. They are incompatible.

### Differences Between NXT Messaging and the RCX Infrared Messaging

This functionality is similar to that found on the LEGO Mindstorms RCX with a few notable exceptions:

1. The RCX uses an infrared communications link. The NXT uses a wireless Bluetooth link.
2. The RCX used broadcast messages that could be received by any RCX. The NXT uses "directed" messages that go to a single NXT.
3. Two RCXes sending messages simultaneously would corrupt both messages. The NXT's Bluetooth allows simultaneous message transmission.
4. There was no queuing of received messages on the RCX. When a new message arrives at the RCX it overwrites previously unhandled messages or it is discarded if the current message is not finished processing.

## 5.8 ROBOTC Bluetooth Messaging

The ROBOTC BT messaging has been optimized for a single slave connection on the master. This allows for significantly less latency on the BT message link. Slaves can immediately transmit messages without having to wait for a polling request from the master. ROBOTC also allows for multiple slave support, but a description of this is beyond the scope of this tutorial. This optimization gives a performance improvement of three to 20 times over other architecture that support – at a significantly reduced performance level – multiple simultaneous slaves.

ROBOTC measured performance to send a BT message, receive it at far end, process it, generate a reply and receive it at original device is about 36 such transactions per second. Other implementations (e.g. NXT-G) typically support about 13 transactions per second.

ROBOTC allows full duplex operation over a single BT stream. Measured performance indicates that each end of the stream can autonomously send 250 messages per second. The half-duplex implementation is limited to 13.

## 5.9 Master Vs. Slave Device

One end of a BT connection is the master device and the other end is the slave device. The master device generates the clocking signal used for the BT connection and the slave device derives its clock from the received radio signal. This results in the following restrictions:

- A device can be either a slave or a master, but not both.

- A single master can (optionally) make connections to multiple slaves.
- A slave can only connect to a single master. It cannot have connections to other masters as it would then need to support multiple clock sources!
- The BT protocol is asymmetric. At its lowest level, A master can transmit over radio at any time.
- A slave only transmits in response to a request from a master.

During the initial connection setup, the two devices negotiate the maximum bandwidth that they will use and how often they are listening for traffic. This allows the devices to use a low-power mode (i.e. the radios are only enabled part of the time) at the expense of lower bandwidth.

# 5.10 Bluecore Bluetooth Information

BT implementation on the NXT uses a "Bluecore" chip from CSR. Bluecore is a self-contained implementation of BT that manages the BT hardware and protocols. It has a few limitations that are common to many other BT hardware implementations:

- The "search" function requires 100% of the BT resources. When a search is in progress no other BT activity can take place on the NXT.
- The module can be in either "command" or "data" mode. They are mutually exclusive states.
    - In command mode, housekeeping functions related to the BT protocol are being performed -- this includes things like searching, pairing, making connections and disconnection.
    - Data mode is used to transfer data between two devices over the BTX wireless link.
- Bluecore supports a single "master" device connecting to up to three 'slave' devices at one time. However, it can only communicate data with one device at a time; it could not simultaneously receive data from all three slaves!

On the radio side, Bluecore can have three connections (or "streams") to different slave devices. However, on the other side -- i.e. the connection from Bluecore to the NXT CPU, only a single connection is possible. It is this implementation that leads to the above restrictions. So, for example:

- If you wanted to add a second slave connection to a NXT, the Bluecore module would be switched into "command" mode. This would interrupt "data" traffic from the original connection. The appropriate commands would be performed to set up the connection and then Bluecore would be switched back to "data" mode.
- In data mode, Bluecore is only connection to one of the two streams/connections at a time. If data from the disconnected stream is received, it is discarded.
- Bluecore can be switched from "data mode on stream 1" to "data mode on stream 2". This is done by switching to "command" mode, sending the "switch stream" command and then switching back to "data" mode; this takes over 100 milliseconds.

# 5.11 NXT-G Bluetooth Messaging Compatibility

The NXT-G firmware has built a message passing system on top of the NXT Bluecore implementation. It works as follows:

- Messages can contain up to 58 bytes of data. The restriction is that firmware has a fixed 64-byte format for message buffers and there are six bytes of overhead in the structure.
- Single master can connect to up to three slaves.
- Messages can be sent to one of ten queues. The queue number is one of the overhead bytes.
- When a message is sent from the master, it is put on a queue of outgoing BT messages.

    - Messages are transmitted from the queue in a FIFO basis whenever the previous queue item is completed.
    - The firmware looks at the stream ID -- ie. which of the three possible connections -- before transmitting a message.
    - If the stream ID does not match the currently "active" stream, then the active stream is switched to this stream by the 100+ millisecond process described above.
- Slave devices cannot use the above process! The slave does not know whether it's stream is the master's "active" stream. The implementation for messaging from the slave is as follows:
    - An outgoing message from the slave is always added to one of the ten outgoing message queues. One queue for each of the ten possible "mailboxes".
    - The message sits in the outgoing queue until the slave receives a "poll for message from mailbox N" message from the master.
    - In response to the "poll for message" the slave responds with either "the selected mailbox is empty" or it sends the first message found in the queue.

If there is only a single slave, then the above process does not incur the 100+ msec delays of switching 'active' streams on the master. It still incurs a delay while the slave waits for a polling request from the master.

# 6. TETRIX / FTC

## 6.1 HiTechnic Controllers

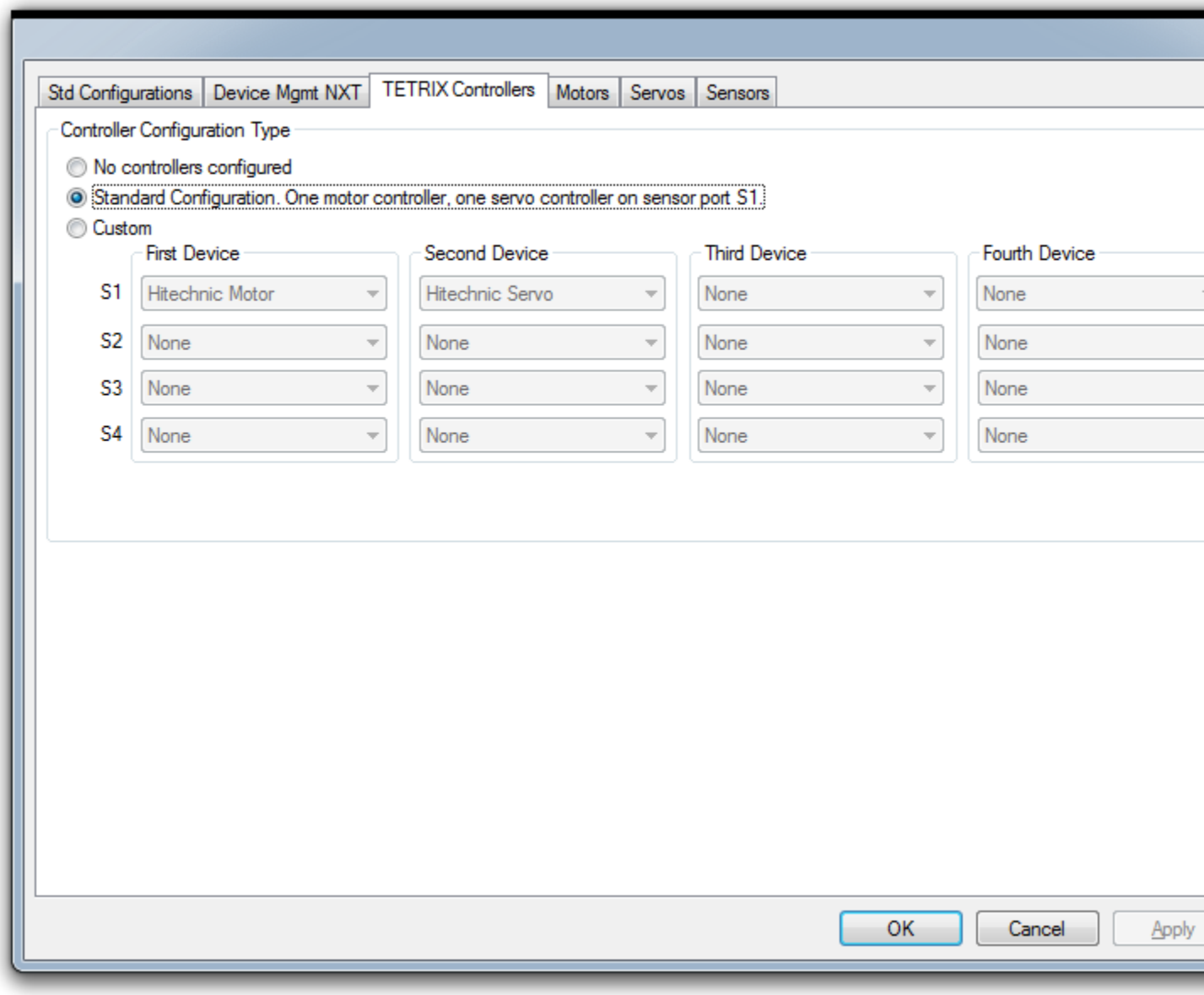
### 6.1.1 Using Motor/Servo Controllers

Because the HiTechnic controllers are able to be daisy chained in a number of configurations, the NXT must be told which sensor port and what order each controller is connected to before you can use the DC Motor or Servo Controllers.

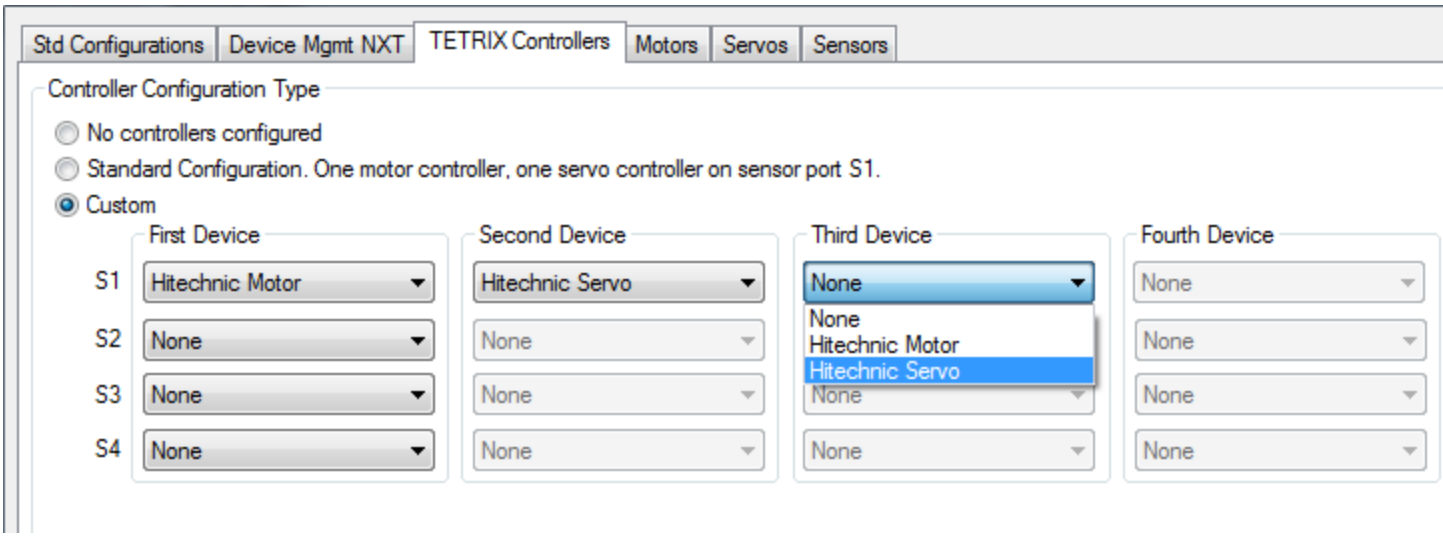
ROBOTC supports every possible configuration, but some shortcuts have been added to make configuring your controllers easier.

There are three settings:

- No Controllers Configured - This is the default configuration. No Motor Controllers or Servo Controllers will be configured.
- Standard Configuration - This configuration is the default FTC configuration, where teams only have one motor and one servo controller . This configuration automatically configures the devices connected to Sensor Port 1, with the Motor Controller connected directly to the NXT and the Servo Controller attached to the daisy-chain port of the Motor Controller. (NXT -> Motor Controller -> Servo Controller). *This is the configuration shown below.*




- Custom Configuration - This allows you to connect as many Motor and Servo controllers in any order. Simply match your hardware configuration to the configuration on the screen and ROBOTC will take care of all of the configuration code for your program.

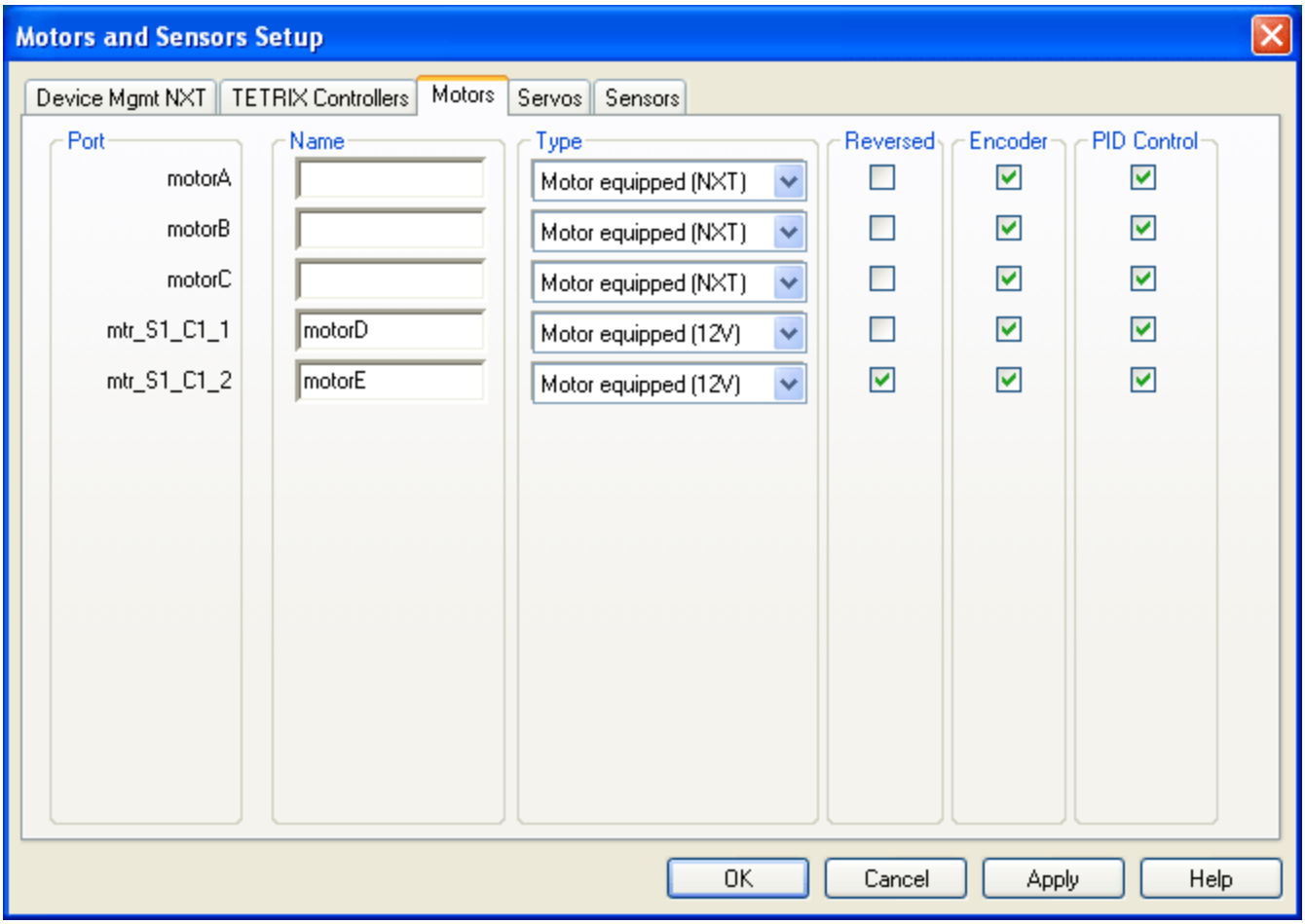



## 6.1.2 Motor Controller

The HiTechnic Motor Controller allows you to drive two 12V motors and read values from two shaft encoders.

---

**Configuration:**
Once your HiTechnic DC Motor Controller is configured in the "TETRIX Controllers" tab of the Motors and Sensor Setup screen, you can use the Motors tab of the Motors and Sensors Setup to configure your motors.




**Port**
The name of the motor. NXT motors are given names "motorA" through "motorC". The TETRIX DC motors are named based on how they're connected to the NXT. The port name can be used to reference a motor in your program (i.e. motor[motorC] = speed;)

The TETRIX motor names can be translated as follows:
**`mtr_S1_C1_1`** - Motor (mtr) on Sensor Port 1 (S1) connected on the first controller (C1) in the daisy chain attached to Motor Port 1 (1)
**`mtr_S1_C1_2`** - Motor (mtr) on Sensor Port 1 (S1) connected on the first controller (C1) in the daisy chain attached to Motor Port 2 (2)

**Name**
Motors can be given more descriptive names, such as "LeftMotor" or "FrontMotor". This name is an alias for the motor port name, so you can use it anywhere you want to specify that motor. The first two configured TETRIX motors are given the names "motorD" and "motorE" by default, but these names can be changed based on your preference.

**Type**
This allows you to set the type of motors, wether they are 12V DC motors or NXT motors.

**PID Control**
This checkbox enables the PID Speed Control functionality for a motor. The TETRIX DC motors don't have built-in encoders, so they must first be installed to use this feature.

**Reverse**
This checkbox will reverse the direction of a motor in an entire program. This is useful when motors are installed opposite one another. Instead of giving one motor a positive power level and the other motor a negative power level to make the robot move forward, both can be given a positive power level, simplifying your code.

---

**ROBOTC Functions:**

**motor[]**
Contains the motor power or speed level (-100 to +100). Negative values are reverse; positive formward. Zero is stopped.
*Example:* (program example can be found at: " \Sample Programs\NXT\Motor\Moving Forward TETRIX.c ")

```
motor[motorD] = 75;                    // Set motorD (FTC motor #1) to power level of 7
```

**nMotorEncoder[]**
Current value of the motor encoder. This value will be returne rotation counts as an integer. Set the value of this function to zero to reset the encoders. (i.e. nMotorEncoder[motor] = 0;)
*Example:* (program example can be found at: " \Sample Programs\NXT\Motor\Motors with While Loops TETRIX.c ")

```
while(nMotorEncoder[motorD] < 10000)       // While motorD encoder is less than 10000 rotat
```

**bMotorReflected[]**
This function when set to true checked will reflect all motor commands to this motor. This is useful if the motors are installed opposite to each other and allows the same "Motor" command to be given to each motor to have the robot drive in a forward direction. (i.e. 100 becomes -100.) This command only needs to be issued once per program and this function is not required if you have enabled "Reverse" control via the Motors and Sensor Setup.
*Example:*

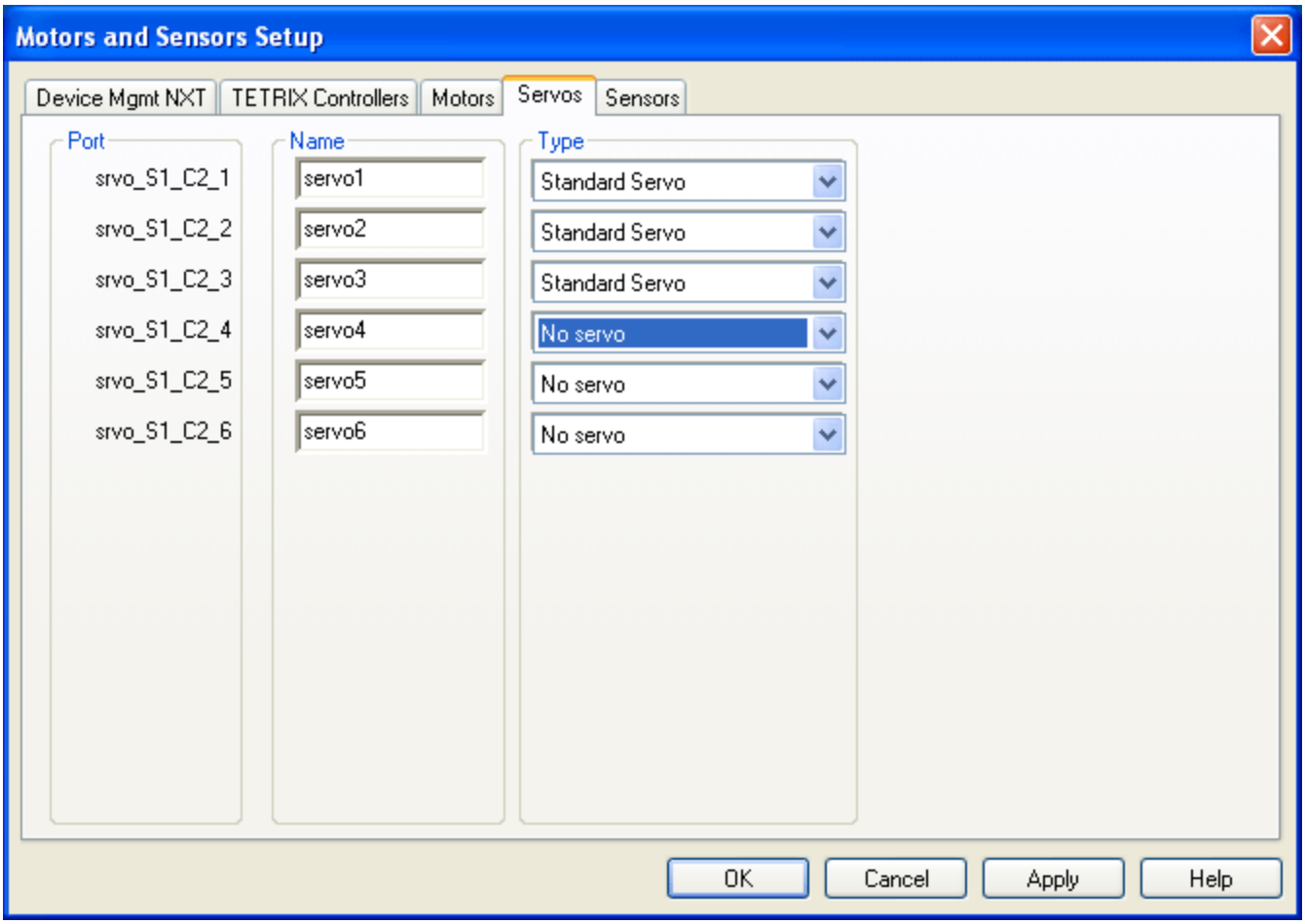
```
bMotorReflected[motorD] = true;            // Enable "Reflected" motorD commands (100 becom
```

## 6.1.3 Servo Controller

The HiTechnic Servo Controller allows you to drive six different servo motors and read their positions via a single NXT sensor port.

---

**Configuration:**
Once your HiTechnic Servo Controller is configured in the "FTC Servo/Motor Ctrl" tab of the Motors and Sensors Setup screen, you can use the Servos tab of the Motors and Sensors Setup to rename your servos.




**Port**
Name of the servos - servos are given names "servo1" through "servo6". This is the name you would use to refer to a specific servo port (i.e. servoTarget[servo1] = position;)

**Name**
You can rename the servo motor to something more descriptive, such as "FrontServo" or "RearServo". This name assigns an alias to your servo, so you can use your own user-defined name, or the default name of "servo#".

**Type**
This allows you to set the types of servos. You can choose between a Standard Servo that has a limited movement range, or a Rotational Servo which can rotate freely.

---

**ROBOTC Functions:**

**`ServoValue[servo#]`**
This read-only function is used to read the current position of the servos on a sensor port Servo controller. Values can range from 0 to 255. The value returned in this variable is the last position that the firmware has told the servo to move to. This may not be the actual position because the servo may not have finished the movement or the mechanical design may block the servo from fully reaching this

position. To set the position of a servo, use the "servoTarget" or "servo" functions.
*Example:* (program example can be found at: "\Sample Programs\NXT\Servos\Servo Sweep TETRIX.c")

```
int a = ServoValue [servo1];    // Assigns the value of servo1 to integer variable 'a'.
```

**servo[servo#] = position** or **servoTarget[servo#] = position;**
This function is used to set the position of the servos on a sensor port Servo controller. Values can range from 0 to 255. The firmware will automatically move the servo to this poisition over the next few update intervals. (Be sure to give the servo some amount of time to reach the new position before going on in your code.)
*Example:* (program example can be found at: " \Sample Programs\NXT\Servos\Servo Sweep TETRIX.c ")

```
servo[servo1] = 160;            // Changes the position of servo1 to 160.
servoTarget[servo1] = 160;      // Changes the position of servo1 to 160.
```

**servoChangeRate[servo#] = changeRate;**
Specifies the rate at which an individual servo value is changed. A value of zero inidcates servo will move at maximum speed. The change rate is a useful variable for "smoothing" the movement of the servos and preventing jerky motion from software calculated rapid and wide changes in the servo value. The default value is a change rate of 10 positions on every servo update which occurs. (updates occur every 20 milliseconds)
*Example:* (program example can be found at: " \Sample Programs\NXT\Servos\Servo Sweep TETRIX.c ")

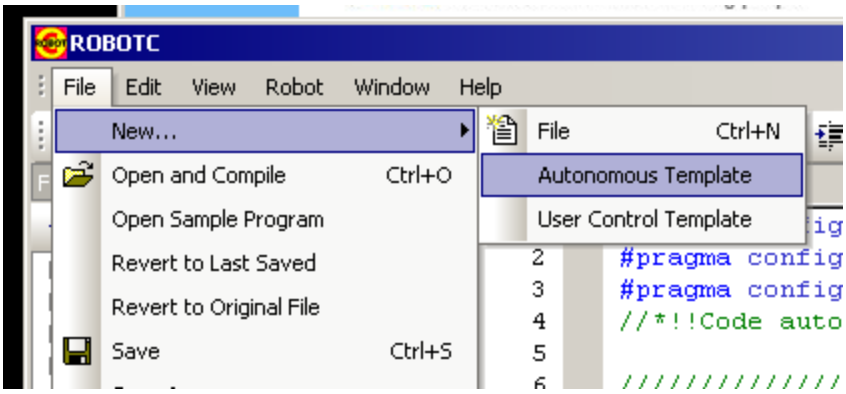
```
servoChangeRate[servo1] = 2;    // Slows the Change Rate of servo1 to 2 positions per upda
```

# 6.2 Competition Control

## 6.2.1 Using Templates

The Field Management System (FMS) during the FTC competition will use a two-program architecture to make programming easier. This means that you will have two sperate programs for the competition, an autonomous program for the first 30 seconds of the competition and a user control program for the remainder of the competition time.

To program in the two-program architecture, you should use the Autonomous and User Control templates available in ROBOTC. To open these templates, use the File Menu - New command.




During the competition, you will manually start your Autonomous program, and then the FMS will automatically switch to your TeleOp program at the start of the User Control portion of the competition. Which program is ran is depended upon the "FTCTeleop.txt" file that lives on your NXT. The FTCTeleop.txt file tells the FMS which program you would like to run when the TeleOp portion of the competition begins.

To generate or change which program you would like to run during TeleOp, use the "Teleop Program" selector built into the Joystick Control window.


# 6.3 Joystick Control

ROBOTC supports using Logitech USB joystick controllers to drive your NXT over the USB or Bluetooth communication link. This allows the user to send commands to their robot in real time, rather than having only pre-programmed behaviors.The joystick functionality works by taking data from the joystick controller and sends it to the NXT over the debugger link as a single message.

ROBOTC has a driver included to take this Bluetooth packet and break it into data, such as variables, that your robot can use. This driver file is provided as an include file. To use this include file, add this line of code to your program:

```
#include "JoystickDriver.c"
```

This include file will create 12 new variables for your program to use. These variables can be used to assign motor speeds, active blocks of code, or control different behaviors. For more examples of using these variables, see the sample code provided under the "Remote Control" sample code folder.

Before you can use these variables, you have to "update" the variables by getting the newest packet of data from the joysticks. Because the joystick station may only send updates every 50-100ms, you should update your joystick values as often as possible to get the most up to date joystick data. To "update" your joystick variables, use this function:

```
getJoystickSettings(joystick);
```

**Sample Program:**

```
#include "JoystickDriver.c"     // Tells ROBOTC to include the driver file for the joystic

task main()
{
 while(true)
  {
    getJoystickSettings(joystick);  // Update Buttons and Joysticks
    motor[motorC] = joystick.joy1_y1;
    motor[motorB] = joystick.joy1_y2;
  }
}
```

## 6.3.1 Using Joysticks

**joystick.joy1_x1**
Value of the X Axis on the Left Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy1_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy1_y1**
Value of the Y Axis on the Left Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy1_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy1_x2**
Value of the X Axis on the Right Joystick on Controller #1. Ranges in values between -128 to +127.

*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                             // Infinite loop:
{
  motor[motorB] = joystick.joy1_x2;       // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy1_y2;       // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy1_y2**
Value of the Y Axis on the Right Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                             // Infinite loop:
{
  motor[motorB] = joystick.joy1_x2;       // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy1_y2;       // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy1_Buttons**
Returns a "Bit Map" for the 12 buttons on Controller #1. For more information on how to use buttons to control actions, See the "Using Buttons" help section.
*Example:*

```
while(true)                         // Infinite loop:
{
  if(joystick.joy1_Buttons == 32)     // If Button 6 is pressed on joy1:
  {
    motor[motorA] = 50;                 // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy1_TopHat**
Returns the value of the direction pad (or "Top Hat") on Controller #1. A value of -1 is returned when nothing is pressed, and a value of 0 to 7 for selected "octant" when pressed.
*Example:*

```
while(true)                         // Infinite loop:
{
  if(joystick.joy1_TopHat == 0)       // If the topmost button on joy1's D-Pad ('TopHat') i
  {
    motor[motorA] = 50;                 // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy2_x1**
Value of the X Axis on the Left Joystick on Controller #2. Ranges in values between -128 to +127.

*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy2_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy2_y1**
Value of the Y Axis on the Left Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\Single Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy2_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy2_x2**
Value of the X Axis on the Right Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x2;      // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy2_y2;      // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy2_y2**
Value of the Y Axis on the Right Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x2;      // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy2_y2;      // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy2_Buttons**
Returns a "Bit Map" for the 12 buttons on Controller #2. For more information on how to use buttons to control actions, See the "Joystick Button Commands" help section.
*Example:*

```
while(true)                        // Infinite loop:
```

```
{
  if(joystick.joy2_Buttons == 32)      // If Button 6 is pressed on joy2:
  {
    motor[motorA] = 50;                   // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy2_TopHat**
Returns the value of the direction pad (or "Top Hat") on Controller #2. A value of -1 is returned when nothing is pressed, and a value of 0 to 7 for selected "octant" when pressed.
*Example:*

```
while(true)                          // Infinite loop:
{
  if(joystick.joy2_TopHat == 0)       // If the topmost button on joy2's D-Pad ('TopHat') i
  {
    motor[motorA] = 50;                   // MotorA is run at a power level of 50.
  }
}
```

## 6.3.2 Using Buttons

Once you have updated your joystick values, you can now use the variables **joystick.joy1_Buttons** or **joystick.joy2_Buttons** to access a bit-masked value of all 12 buttons sent. This value, however, is not useable in your program right away.

ROBOTC has added two new functions, **joy1Btn(button)** and **joy2Btn(button)** to allow you to know which buttons have been pressed when multiple buttons are pressed at a time. Using **joy1Btn(button)** and **joy2Btn(button)**, each button returns the a value of 1 if pressed and a value of 0 if not pressed. This will allow you to write your program to work with multiple buttons simultaenously.

```
#include "JoystickDriver.c"      // Tells ROBOTC to include the driver file for the joystic

task main()
{
 while(true)
  {
    getJoystickSettings(joystick);  // Update Buttons and Joysticks

    if(joy1Btn(1) == 1)                  // If Joy1-Button1 is pressed:
    {
      motor[motorA] = 100;              // Turn Motor A On at full power
    }
    else                                // If Joy1-Button1 is NOT pressed:
    {
     motor[motorA] = 0;                  // Turn Motor A Off
    }
  }
}
```


# 6.4 TETRIX Debugger Windows

## 6.4.1 Joystick Station

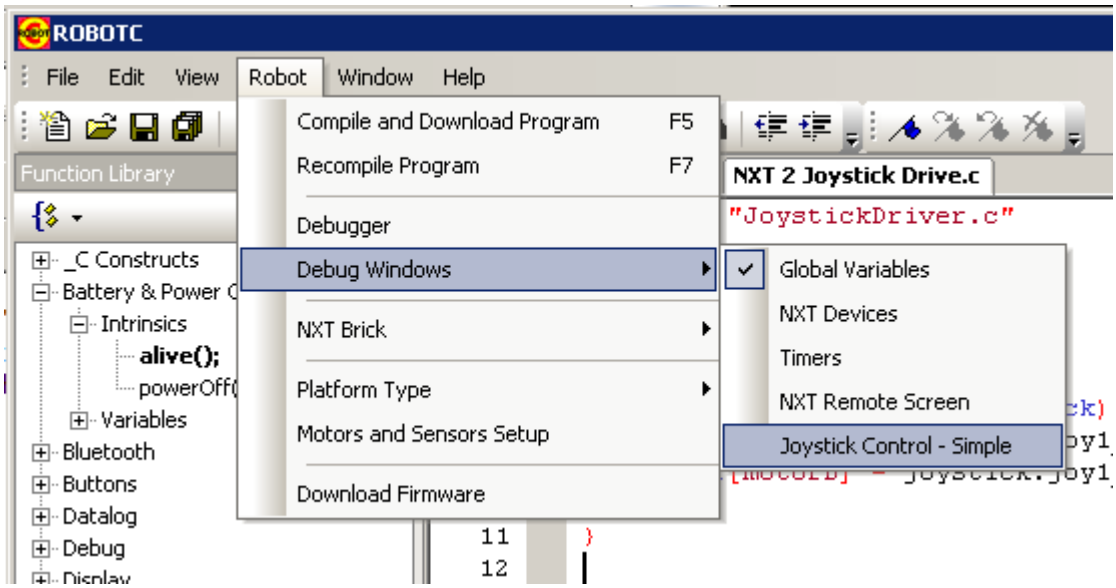
ROBOTC has built two different Joystick Controller stations built into the interactive debugger. "Joystick Control - Simple" is a debugger window to control the NXT via a Logitech USB remote control. "Joystick Control - Game" is a full featured Controller Station which is used mainly for FIRST Tech Challange or other competitions that are NXT or TETRIX based.

---

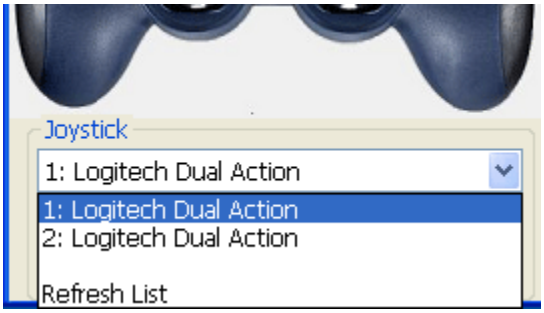
**"Joystick Control - Simple"**

After your program has been downloaded and the debugger is opened, you can open the Controller Station by:

- Downloading your program and starting the debugger
- Go to the "Robot" menu
- Choose the "Debug Windows" sub-menu
- Click on the "Joystick Control - Simple" menu option to open the Controller Station




Once the Controller Station is opened, ROBOTC will look for any joysticks attached to your computer via USB. You can choose which joystick you want to robot to be controlled with by changed the joystick under the available drop-down menu. If you have no joysticks available, this list will be empty and ROBOTC will alert you that you have "No Controllers Configured"




You can see what data is being generated by the Joystick Station by looking at the X1, Y1, POV, X2, Y2 and Buttons display directly below the dropdown menu. This will give you realtime feedback of what values are being sent to your NXT from the Joystick Station. This data will also be illustrated with green dots to reflect the values and button presses.

ROBOTC will send joystick data to your NXT over Bluetooth or USB, but only when the Joystick Control window is opened. You will need to have the debugger open to use the Joystick Station.

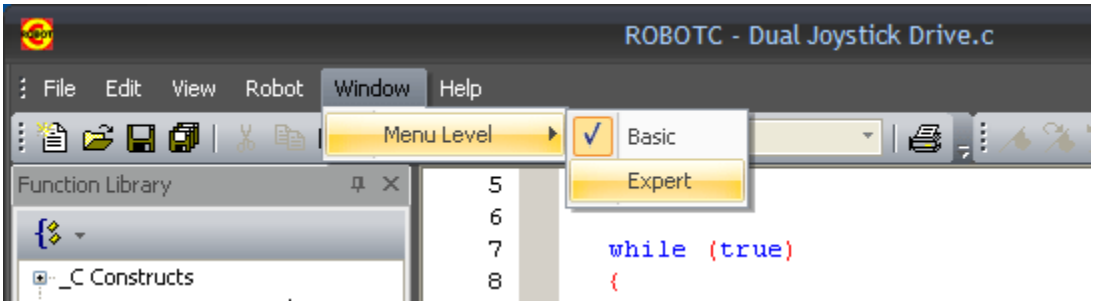

---

**"Joystick Control - Game"**

There is also a second Joystick Control window, "Joystick Control - Game". This window is specifically designed to emulate the FIRST Tech Challenge game mode. To test your FTC competition programs, you can use the controller station to mimic what the Field Management System will do. This includes switching between Autonomous and User Control, Changing if your robot is on the blue or red alliance and also disabling (or pausing) your robot. These commands can be found on the left side of the Joystick Station.

You can open the Controller Station by:

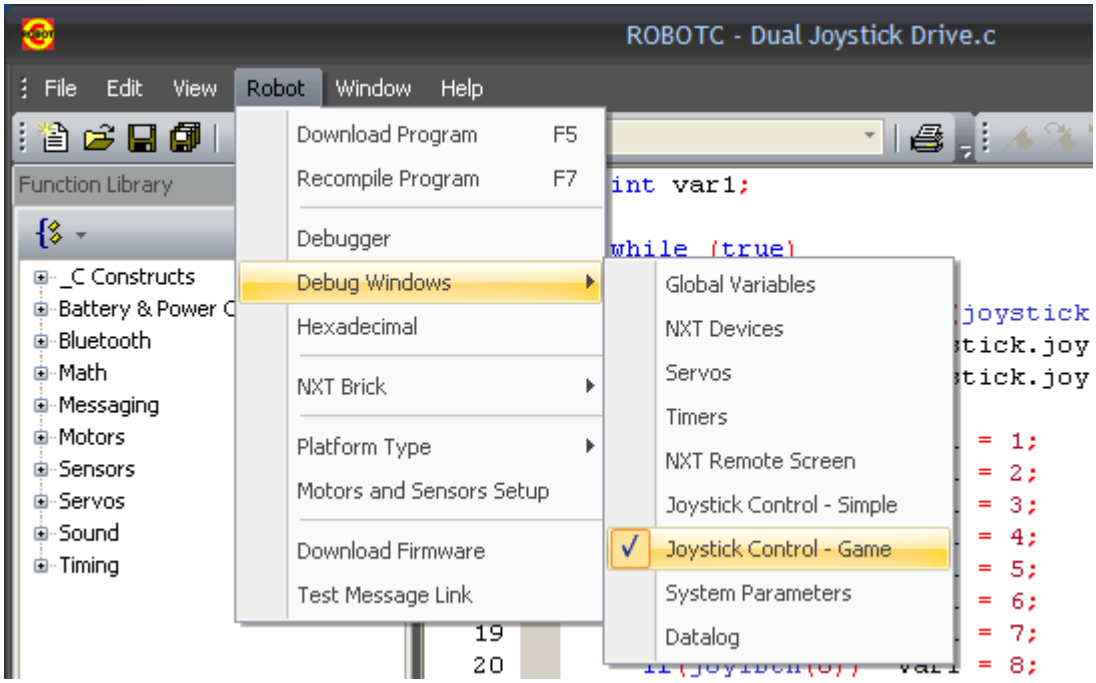
- Go to the "Window" menu
- Choose the "Menu Level" sub-menu
- Click on the "Expert" menu option




After setting the Menu Level to "Expert", you can then access the "Joystick Control - Game" station by:

- Downloading your program and starting the debugger
- Going to the "Robot" menu
- Choose the "Debug Windows" sub-menu

- Click on the "Joystick Control - Simple" menu option to open the Controller Station



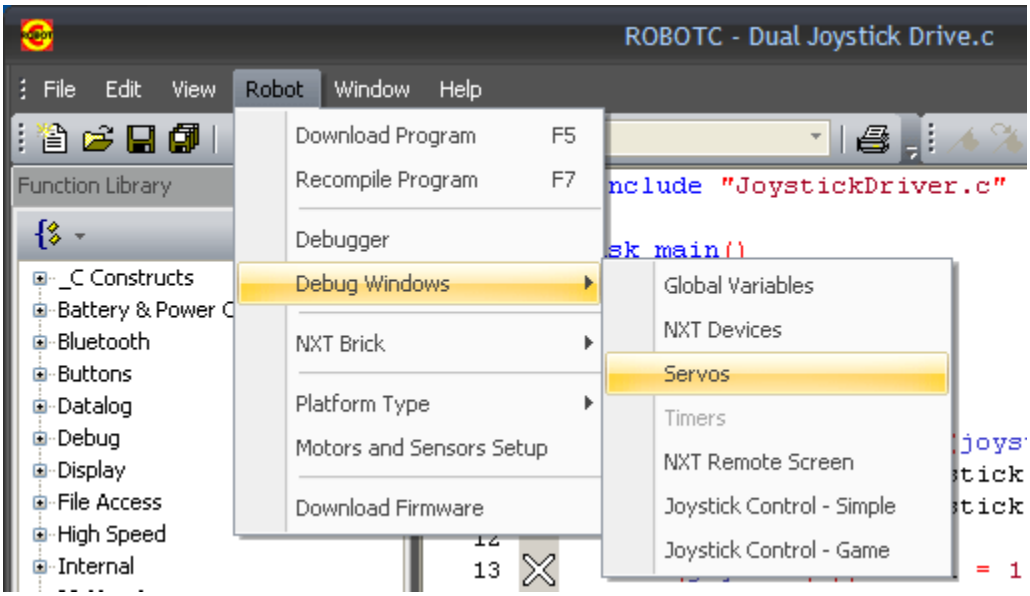

As you can see, the window has a few more options than the "Joystick Control - Simple" screen did.


If you would like to use two controllers, click the "Dual Joysticks" button to expand the Joystick Control window to facilitate two controllers. You can assign the same controller to both Primary and Secondary Joysticks, but this is not recommended.

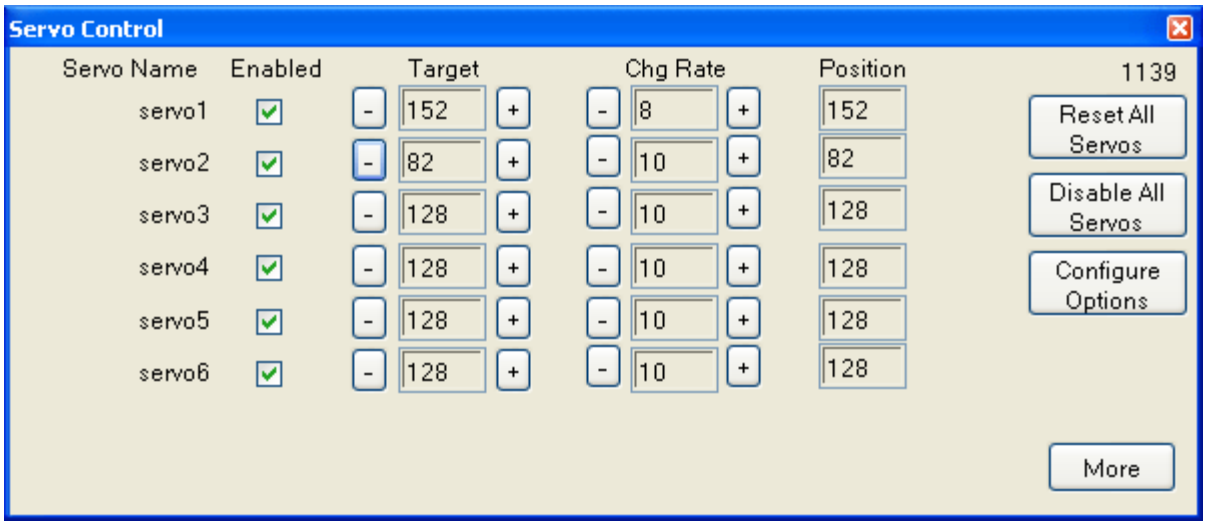

## 6.4.2 Servo Control

The "Servo Control" debug window allows you to see the positions of all of your servos attached to your Servo Controller. This window will also allow you to manually move the servo position by expanding the window by using the "More" button.

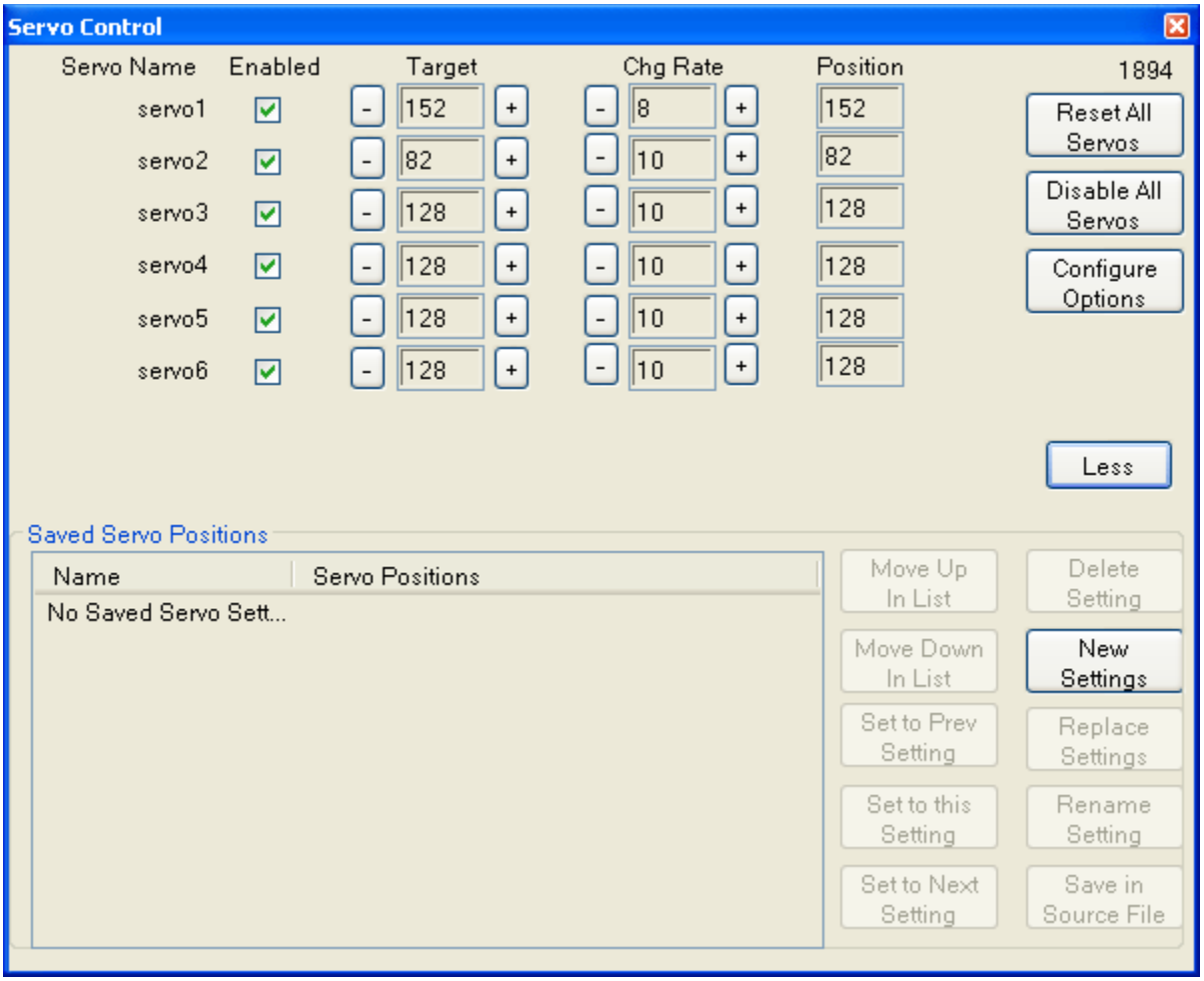
You can open this debugger window by going to the "Robot" menu when the debugger is already opened:




Once the Servo Control window is opened you will be able to see a run-time update of all the servo's positions. You can also change the Targets and Change Rates of each servo manually.



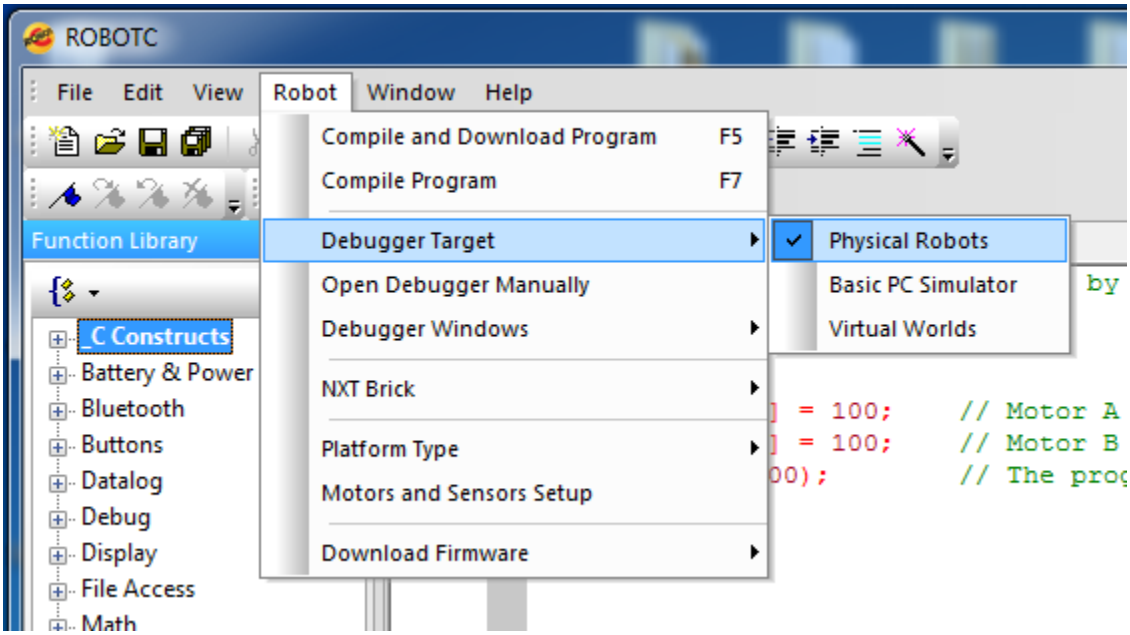

By clicking the "More" button, you can save, edit, and load servo positions.



# 7. ROBOTC Debugger

## 7.1 Debug Target

The ROBOTC Debugger has 3 possible targets: Physical Robots, Basic PC Simulator, and Virtual Worlds.




**Physical Robots**
Download your programs to a real robot.

**Basic PC Simulator**
Download your programs to a simulator that runs within ROBOTC. Certain functionality will be missing such as sensor feedback. This is useful for debugging when you are not near a real robot.

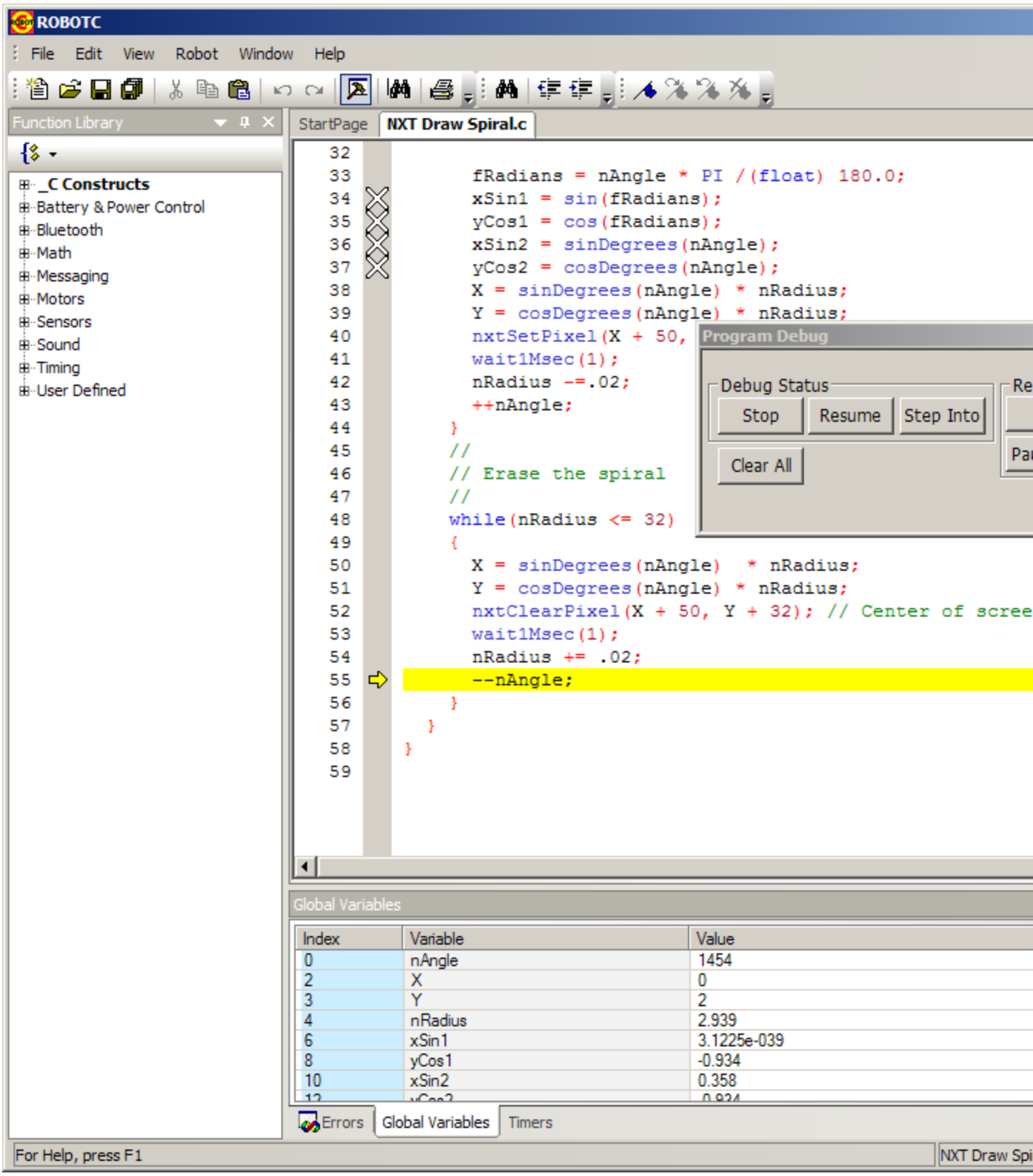
**Virtual Worlds**
Download your programs to a Virtual World and run them on your virtual robot.

## 7.2 Debugging

ROBOTC has a debugging capability that enables unparalleled interactive real-time access to the robot as your program is running. This has the potential to significantly reduce the time it takes to find and correct errors in your programs. With ROBOTC's debugger you can:

- Start and stop your program execution from the PC
- “Single step” through your program executing one line of code at a time and examining the results (i.e. the values of variables) and the flow of execution.

- Define "breakpoints" in your code that will cause program execution to be suspended when they are reached during program execution
- Read and write the values of all the variables defined in your program
- Read the write the values of motors and sensors.


ROBOTC
File Edit View Robot Window Help
Function Library
_C Constructs
Battery & Power Control
Bluetooth
Math
Messaging
Motors
Sensors
Sound
Timing
User Defined
StartPage
NXT Draw Spiral.c
32
33 fRadians = nAngle * PI /(float) 180.0;
34 xSin1 = sin(fRadians);
35 yCos1 = cos(fRadians);
36 xSin2 = sinDegrees(nAngle);
37 yCos2 = cosDegrees(nAngle);
38 X = sinDegrees(nAngle) * nRadius;
39 Y = cosDegrees(nAngle) * nRadius;
40 nxtSetPixel(X + 50,
41 wait1Msec(1);
42 nRadius -=.02;
43 ++nAngle;
44 }
45 //
46 // Erase the spiral
47 //
48 while(nRadius <= 32)
49 {
50 X = sinDegrees(nAngle) * nRadius;
51 Y = cosDegrees(nAngle) * nRadius;
52 nxtClearPixel(X + 50, Y + 32); // Center of scree
53 wait1Msec(1);
54 nRadius += .02;
55 --nAngle;
56 }
57 }
58 }
59
Program Debug
Debug Status
Stop
Resume
Step Into
Clear All
Global Variables
Index Variable Value
0 nAngle 1454
2 X 0
3 Y 2
4 nRadius 2.939
6 xSin1 3.1225e-039
8 yCos1 -0.934
10 xSin2 0.358
Errors
Global Variables
Timers
For Help, press F1

There are no restrictions on the number of debugger windows that can be simultaneously opened. However, each window does require the PC to communicate with the robot controller to refresh its data. The more windows that are open, the more data transmission required for the refreshing activity.

ROBOTC has been optimized to minimize refresh times. Typically only one or two messages are required to refresh each window; this is valuable on the robot controllers that have a “slow” communications channel between the robot and the PC.

For example, Bluetooth messaging on the NXT platform is slow, taking about 25 milliseconds per message. Out of the box, NXT Bluetooth messages are restricted to 58 bytes of data, so normally 13 messages are required to refresh the 800 byte NXT LCD image. The ROBOTC enhanced firmware is able to refresh the NXT LCD using a single message.

You can optimize how fast the debugger windows update by closing unused windows and also shrinking the debugger windows to show less information. The debugger will not update values that are not visible on the screen, minimizing how much "data" must be sent from the robot to the PC.

# 7.3 Debugger Vs. Traditional Methods

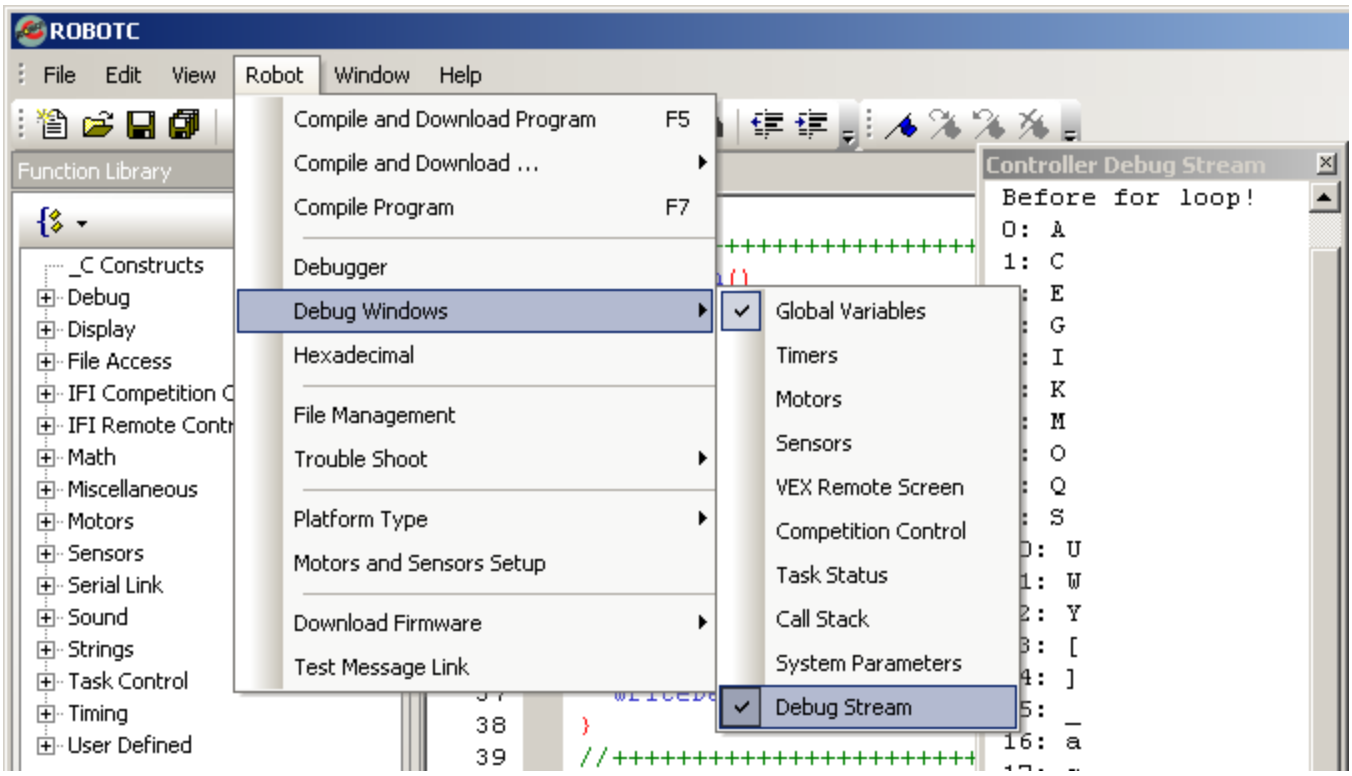
**“Traditional” Debugging Techniques**

Debugging a program – finding the errors and correcting them – can be a slow process in solutions without a run-time debugger. Without a debugger you may have to resort to different techniques like:

- There’s no way to determine if your program is executing the intended logic. So you add code to play different tones/sounds to your program as it executes different “blocks” of code. You determine from the sound what is being executed within your program.
- If your robot platform supports a display device (which could be a serial link to your PC or an LCD display on the robot) then you would have to add “print” statements to your program code at various points in your program. By examining the display, you can (hopefully) determine what’s happened in your program's execution by the display.

Both of the above techniques are available in ROBOTC. However, a real-time debugger eliminates the need to resort to them. There’s no need to add code for debugging to your program. A built-in debugger provides better functionality without ever having to modify your source code!

There is also a built-in Debug Stream that you can use to keep track of your program from behind the scenes. For example, you could print a message to the Debug Stream when you enter and exit loops, functions, etc. Then you can view the cached Debug Stream to help in the debugging process.




```
writeDebugStream();
```
Writes a String to the Debug Stream.

```
writeDebugStream("int x is: %d", x);        // Writes the current value of int 'x' to the d
```

```
writeDebugStreamLine();
```
Writes a String to the Debug Stream starting on a new line.

```
writeDebugStreamLine("int x is: %d", x);    // Writes the current value of int 'x' to the d
                                            // on a new line.
```

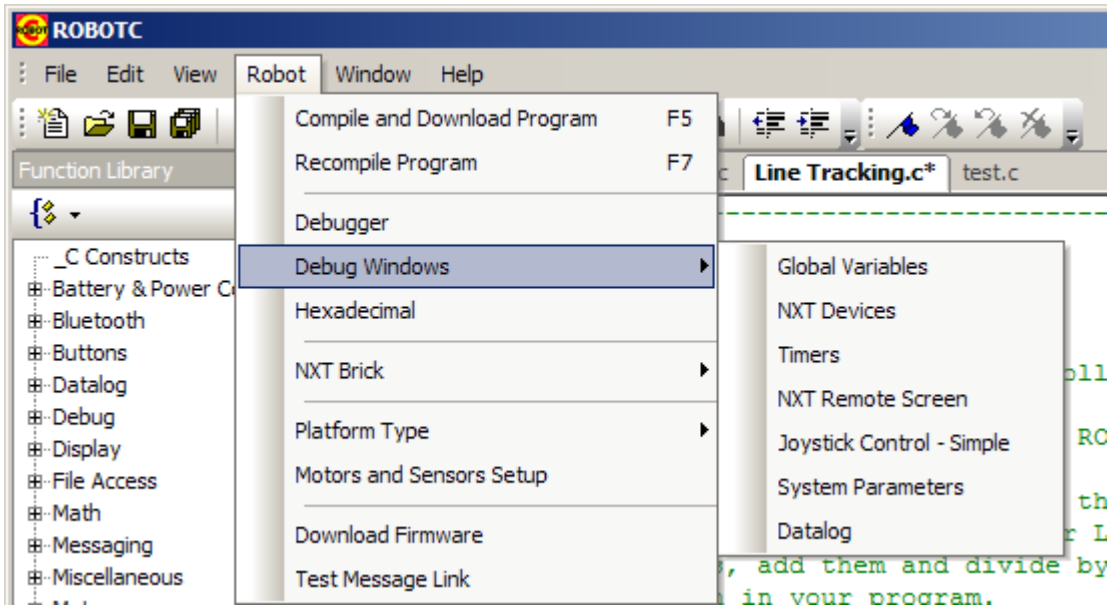
# 7.4 Debugger Windows

There are several different debugger windows that can be used when the ROBOTC debugger is open. The default windows are the Devices and Global Variables, which gives access to variables, motors, sensor, and timer statuses. Some of the windows are infrequently used but can be useful when needed. Debugging windows can be opened from the sub-menu shown in the following picture.

There are no restrictions on the number of debugger windows that can be simultaneously opened. However, each window does require the PC to “message” with the robot controller to refresh its data. The more windows that are open the more data transmission required for the refreshing activity.

ROBOTC has been optimized to minimize refresh times. Typically only one or two messages are required to refresh each window; this is valuable on the robot controllers that have a “slow” communications channel between the robot and the PC.

For example, Bluetooth messaging on the NXT platform is slow taking about 25 milliseconds per message. Out of the box, NXT Bluetooth messages are restricted to 58 bytes of data and 13 messages are required to refresh the 800 byte NXT LCD image. The ROBOTC enhanced firmware performs only requires a single message.



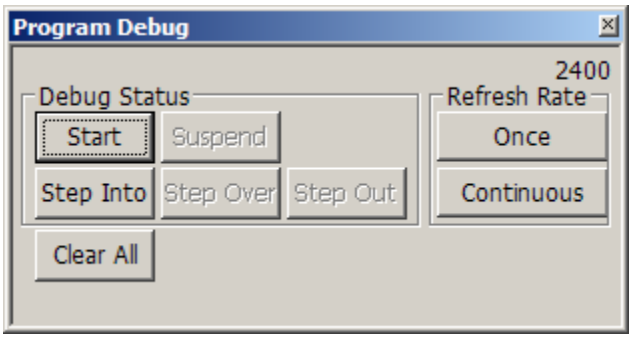

The number of available windows is also impacted by the “Menu Level” setting found on the “Window” sub-menu. "Basic" mode will have less menu options than "Expert" mode, which is shown above.

**Basic Mode**
- Program Debug
- Global Variables
- NXT Devices
- Timers
- NXT Remote Screen
- Joystick Control - Simple

**Expert Mode**
- Program Debug
- Global Variables
- NXT Devices
- Timers
- NXT Remote Screen
- Joystick Control - Simple
- System Parameters
- Datalog

# 7.4.1 Program Debug

The Program Debug window appears every time you download a program to your robot controller. Closing it will close the connection between your computer and the robot controller, along with any other open debug windows.




**Start**
The Start button will start the program execution on your robot controller.

**Suspend**
The Suspend button will suspend (pause) the program execution on your robot controller.

**Step Into**
The Step Into button will execute the next command in your program.

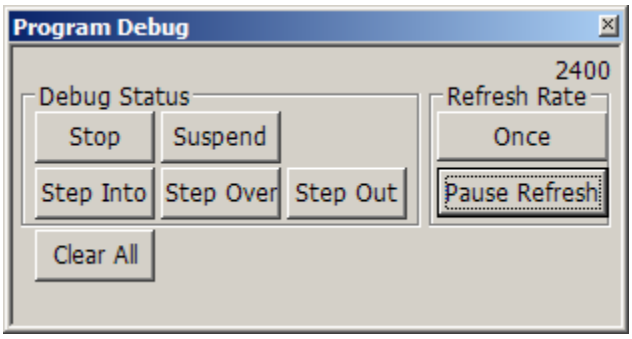
**Step Over**
The Step Over button will execute an entire function in your program.

**Step Out**
The Step Out button will step the program execution out of a function in your code.

**Clear All**
The Clear All button will reset all of the values being displayed by the other debug windows.

**Once**
Pressing the Once button will update the values in the debugger windows once.

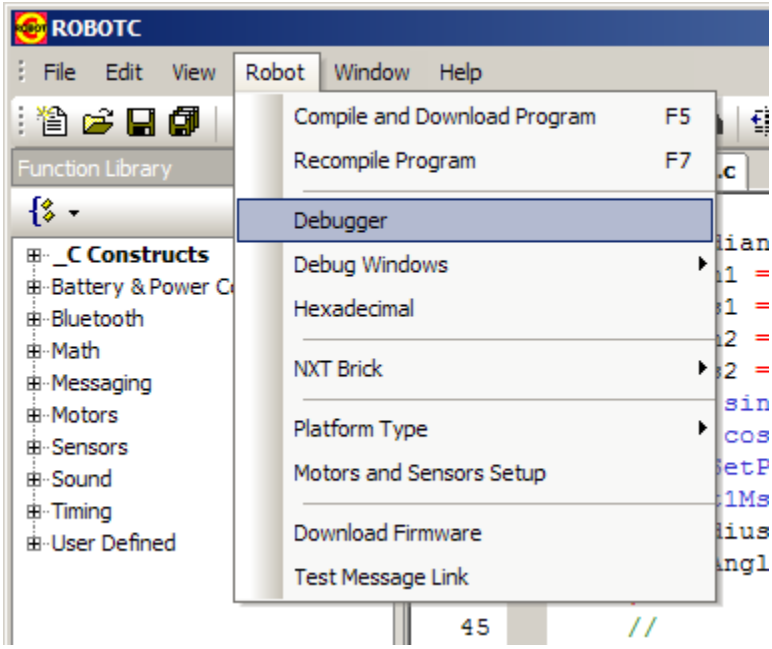
**Continuous**
Pressing the Continuous button will cause the values in the debugger windows to update continuously. Pressing it will also cause the text to change to "Pause Refresh".



**Stop**
The Stop button will stop the program execution on your robot controller. It only appears after a program has been started.

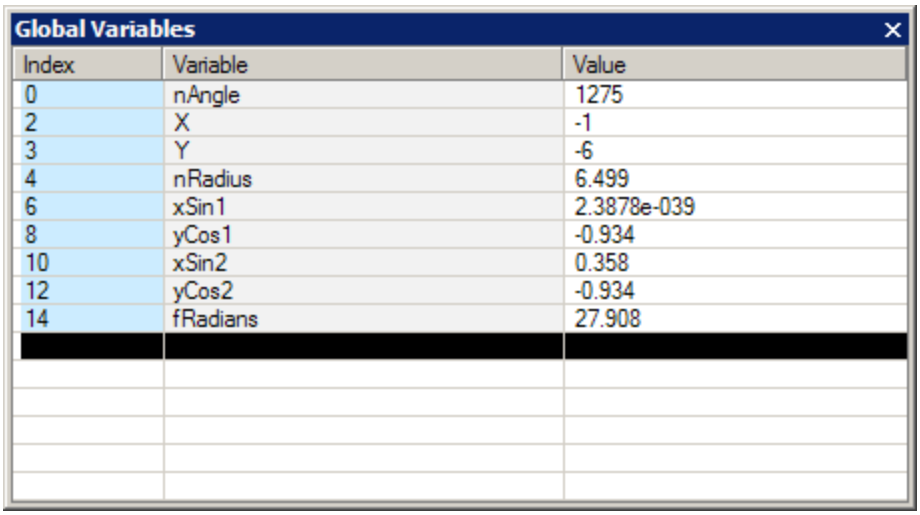
**Pause Refresh**
Pressing the Pause Refresh button will cause the values in the debugger windows to stop updating. Pressing it will also cause the text to change to "Continuous".

The Program Debug window can also be opened by going to the Robot menu and selecting "Debugger".




# 7.4.2 Global Variables

The Global Variables window displays the Index, Variable name, and Value for each user defined variable in a program. The Global Variables window can only be open after a connection has been established between your computer and the robot controller.

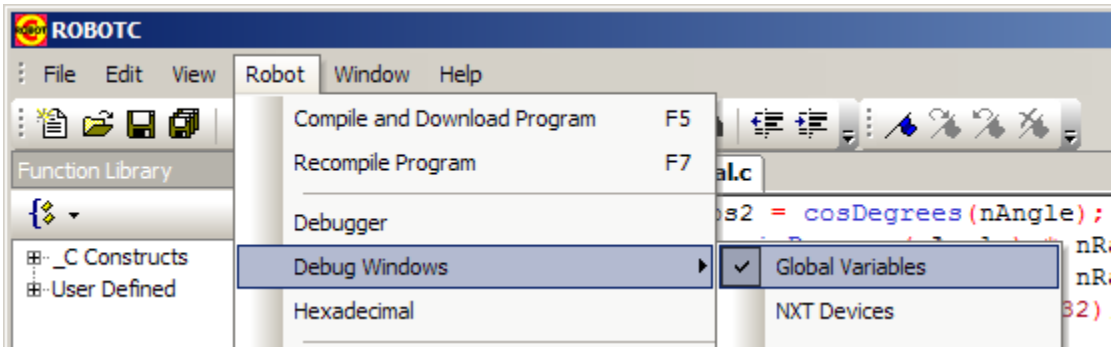



**Index**
The index of the variable, in memory.

**Variable**
The name of the variable, defined in the program.

**Value**
The value of the variable during program execution. Values will update automatically if the Program Debug window is set to update continuously.

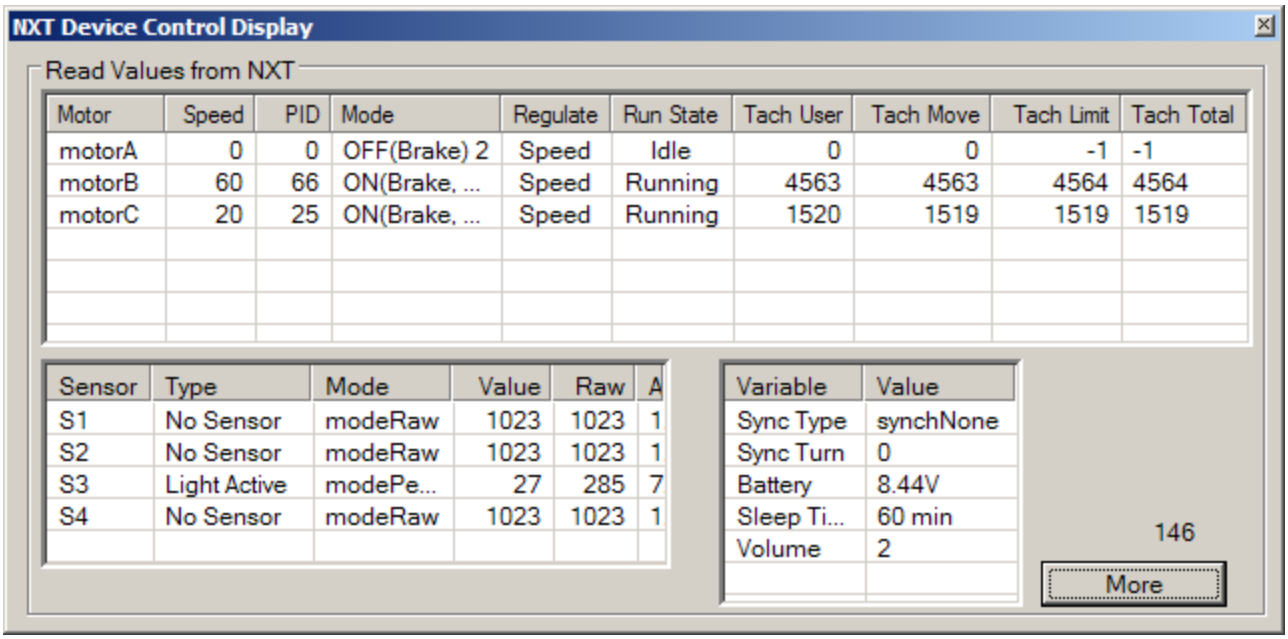
The Global Variables window can be opened by going to the Robot menu, Debug Windows, and selecting Global Variables.




# 7.4.3 NXT Devices

The NXT Device Control Display is an interactive window that allows you to read values from and write values to the NXT Devices - motors, sensors, additional motor controllers, ect. The NXT Device Control Display can only be open after a connection has been established between your computer and the robot controller.

By default, the NXT Device Control Display appears in a simplified display.




**Motor**
The name/port of the referenced motor.

**Speed**
The power level of the motor, set by the programmer.

**PID**
The actual power level being applied to the motor to compensate for external forces, like friction. Only applicable if PID is enabled.

**Mode**
The mode of the motor - On or Off, Brake or Float, PID Regulated or Not Regulated

**Regulate**
Displays whether the motors Speed is PID regulated or not.

**Run State**
Displays whether a motor is Idle or Running.

**Tach (Encoder) User**
Encoder count value that is total under user control (this variable is able to be reset.)

**Tach (Encoder) Move**
The distance in encoder counts that you specified the motorr to move.

**Tach (Encoder) Limit**
Distance until the motor stops when the program is set to stop at a specific position.

**Tach (Encoder) Total**
Total Encoder count since program started.

**Sensor**
The name/port of the referenced sensor.

**Type**
The type of sensor configured - Light, Touch, Temperature, ect.

**Mode**
The mode of the sensor, determines how values are interpreted by the controller - Raw, Percentage, ect.

**Value**
The interpreted value of the sensor.

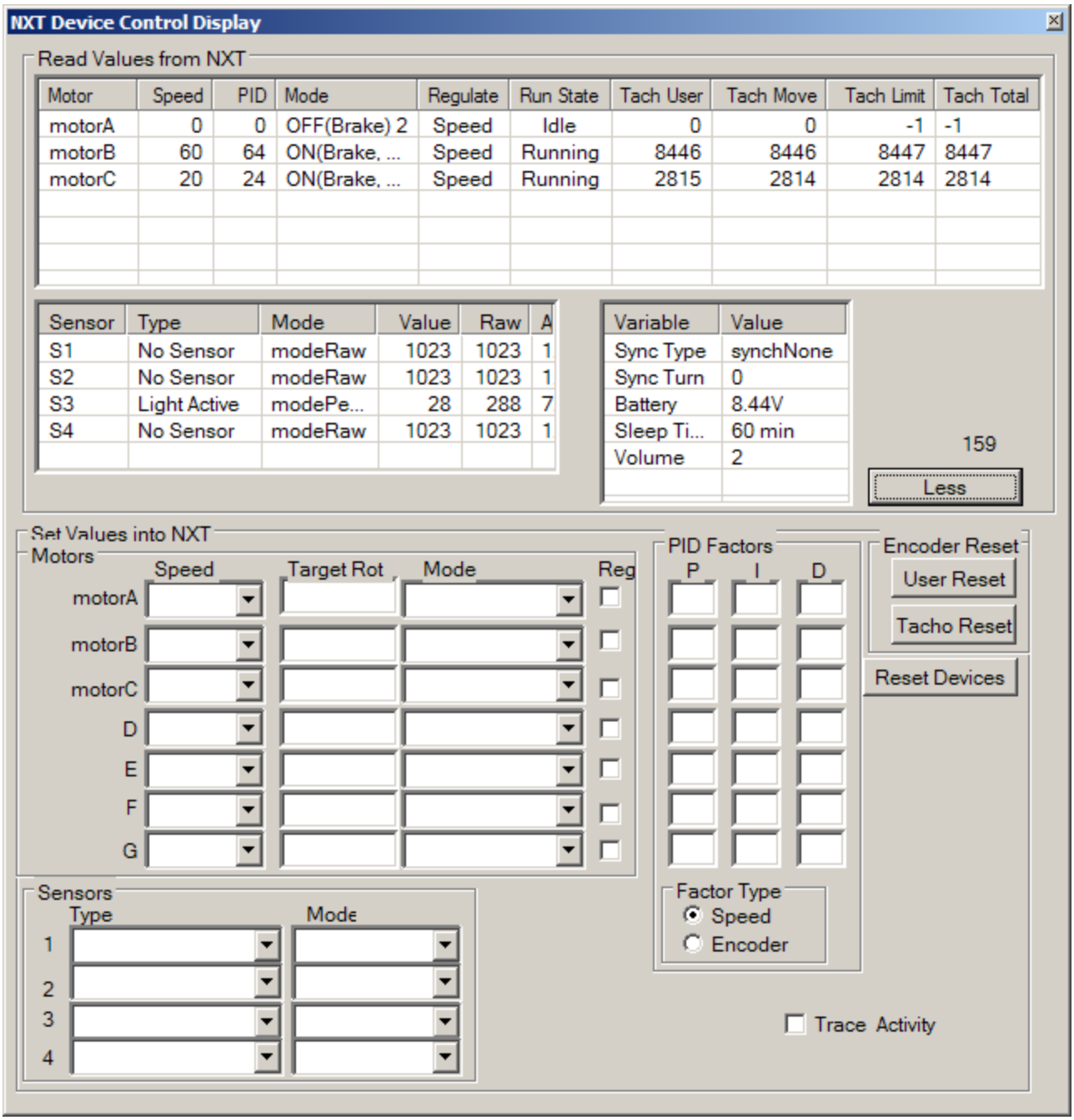
**Raw**
The raw value of the sensor.

**A-to-D**
The Analog-to-Digital value of the sensor.

**Variable**
Additional values internal to the NXT:

- **Sync Type -** Displays how motors are synchronized, if at all.
- **Sync Turn -** Displays the turn ratio between syncronized motors.
- **Battery -** Displays the battery level of your NXT.
- **Sleep Time -** Displays the amount of time before your NXT will automatically shut off.
- **Volume -** Displays the Volume of your NXT (0-4).

**More**
Expands the NXT Device Control Display to its full, interactive mode.




**Motors**
Allows you to set the speed of the motors, along with additional configuration options.

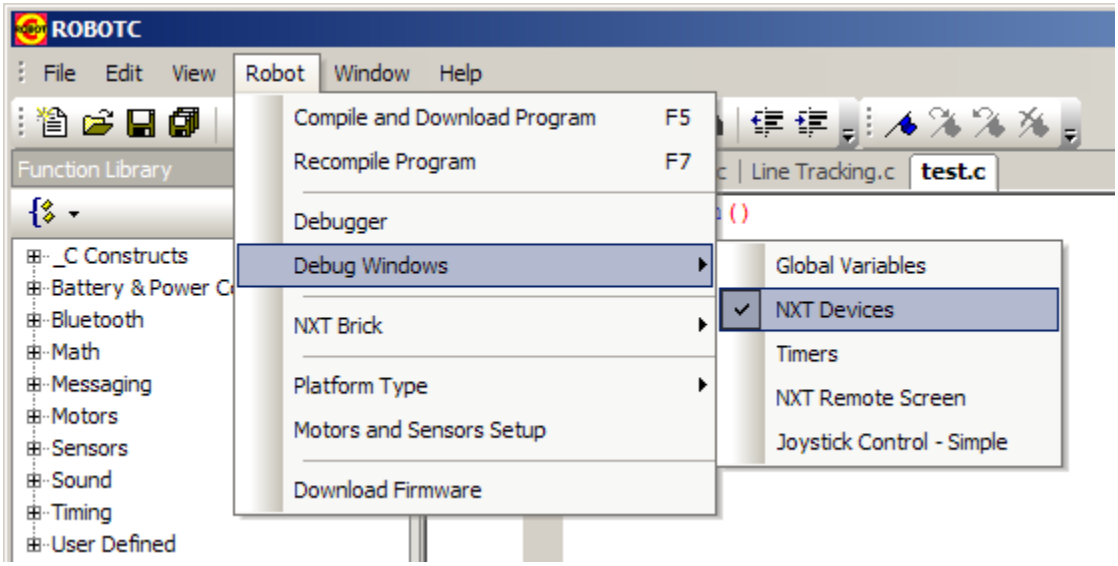
**PID Factors**
Allows you to customize how the PID algorithm calculates the applied power level.

**Encoder Reset**
Allows you to reset the values of the encoders.

**Reset Devices**
Resets all settings in the "Set Values into NXT" area.

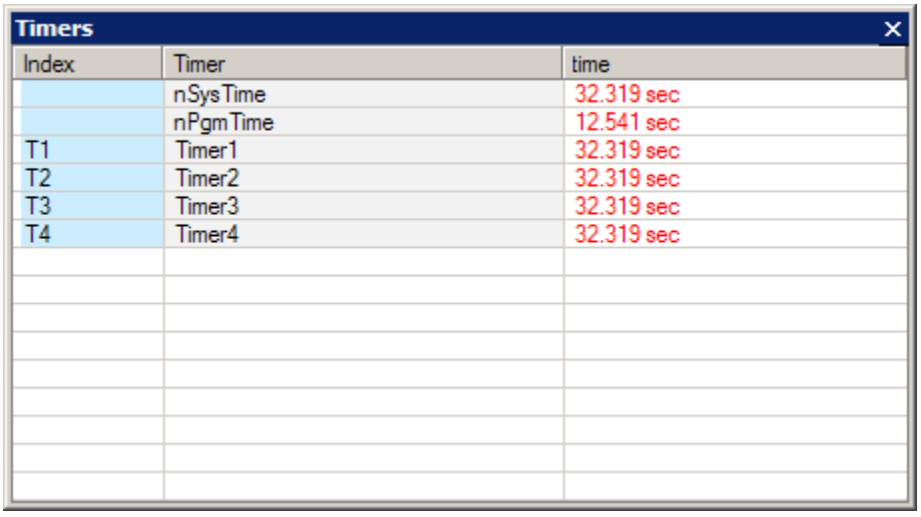
**Sensors**
Allows you to set the sensor type and mode.

The NXT Device Control Display can be opened by going to the Robot menu, Debug Windows, and selecting NXT Devices.




# 7.4.4 Timers

The Timers window displays the Index, Timer name, and elapsed time for each NXT timer.

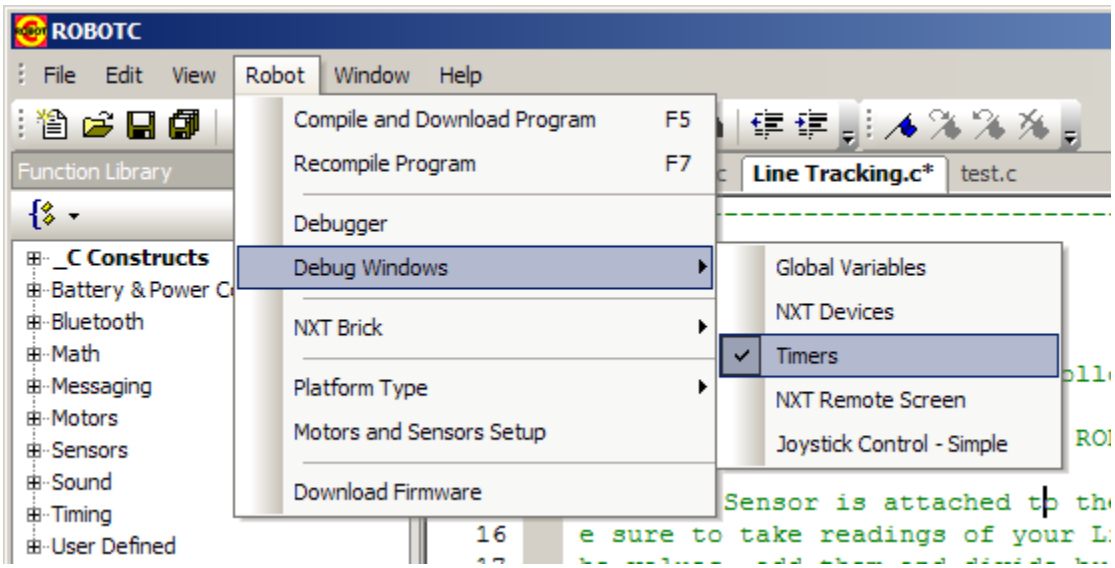



**nSysTime**
The system timer. Displays how long the NXT has been turned on.

**nPgmTime**
The program timer. Displays how long the current program has run.

**Timer1 - Timer4**
User accessible timers that can be reset to 0 in programs, and then used to monitor elapsed time and control program flow. See more information here .

The Global Variables window can be opened by going to the Robot menu, Debug Windows, and selecting Global Variables.




# 7.4.5 NXT Joystick

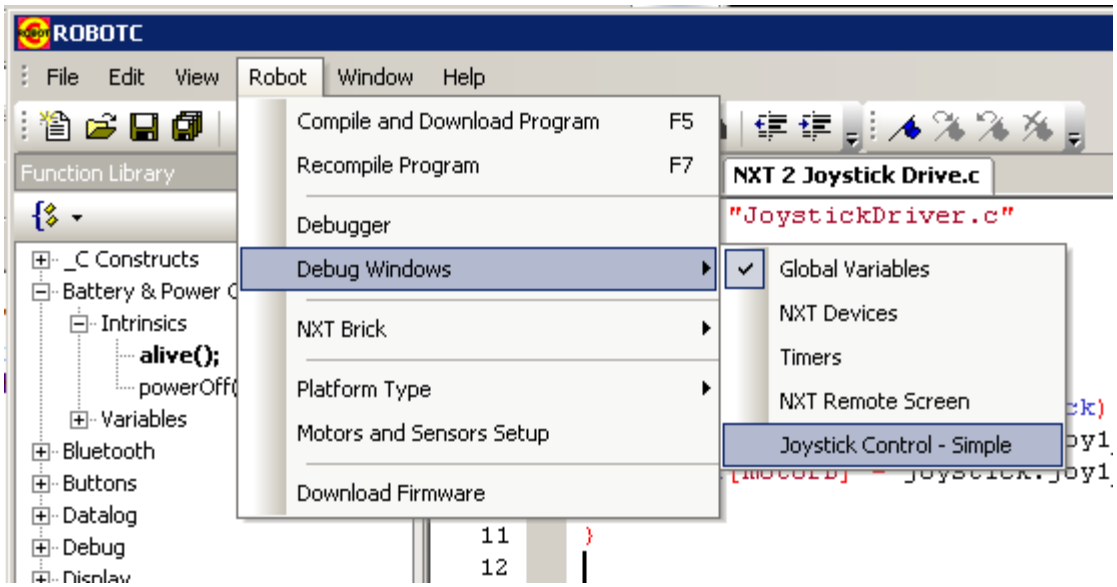
ROBOTC has built two different Joystick Controller stations built into the interactive debugger. "Joystick Control - Simple" is a debugger window to control the NXT via a Logitech USB remote control. "Joystick Control - Game" is a full featured Controller Station which is used mainly for FIRST Tech Challange or other competitions that are NXT or TETRIX based.

---

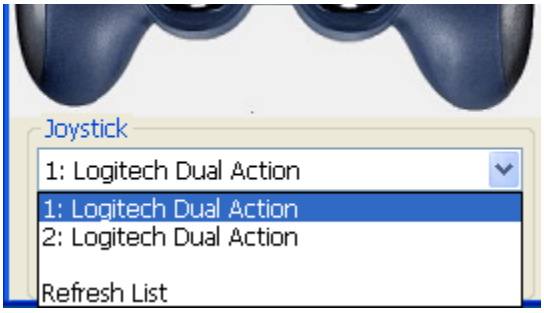
**"Joystick Control - Simple"**

After your program has been downloaded and the debugger is opened, you can open the Controller Station by:

- Downloading your program and starting the debugger
- Go to the "Robot" menu
- Choose the "Debug Windows" sub-menu
- Click on the "Joystick Control - Simple" menu option to open the Controller Station




Once the Controller Station is opened, ROBOTC will look for any joysticks attached to your computer via USB. You can choose which joystick you want to robot to be controlled with by changed the joystick under the available drop-down menu. If you have no joysticks available, this list will be empty and ROBOTC will alert you that you have "No Controllers Configured"




You can see what data is being generated by the Joystick Station by looking at the X1, Y1, POV, X2, Y2 and Buttons display directly below the dropdown menu. This will give you realtime feedback of what values are being sent to your NXT from the Joystick Station. This data will also be illustrated with green dots to reflect the values and button presses.

ROBOTC will send joystick data to your NXT over Bluetooth or USB, but only when the Joystick Control window is opened. You will need to have the debugger open to use the Joystick Station.

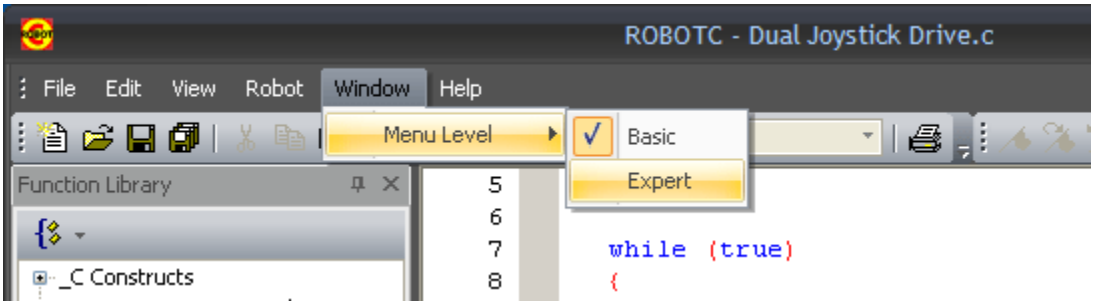

---

**"Joystick Control - Game"**

There is also a second Joystick Control window, "Joystick Control - Game". This window is specifically designed to emulate the FIRST Tech Challenge game mode. To test your FTC competition programs, you can use the controller station to mimic what the Field Management System will do. This includes switching between Autonomous and User Control, Changing if your robot is on the blue or red alliance and also disabling (or pausing) your robot. These commands can be found on the left side of the Joystick Station.

You can open the Controller Station by:

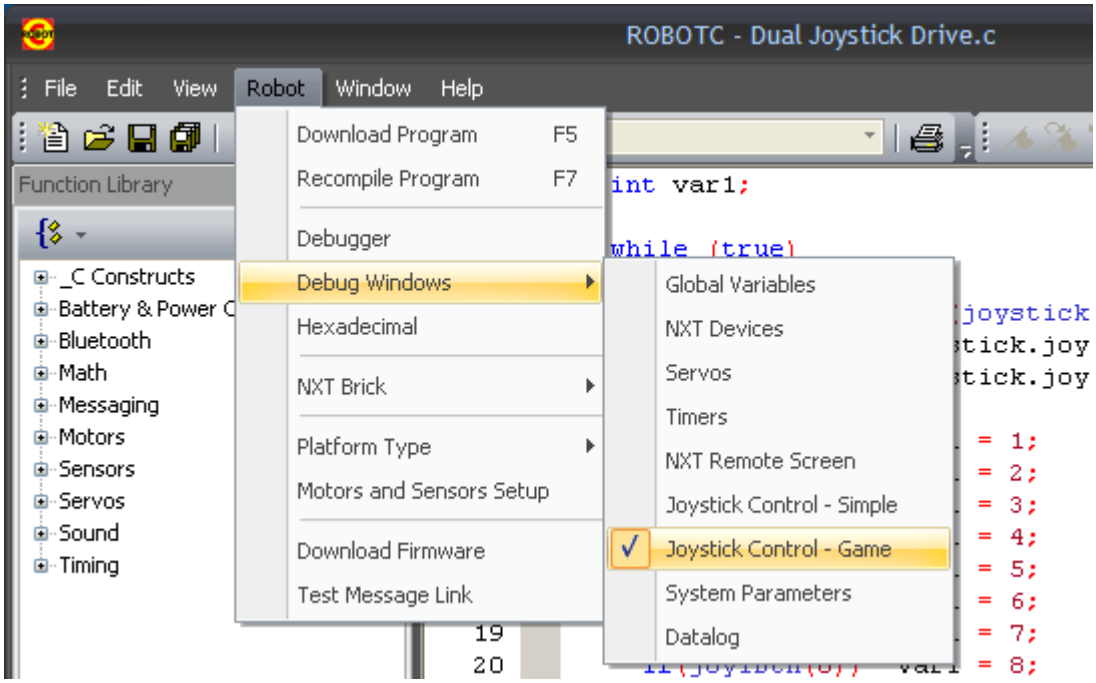
- Go to the "Window" menu
- Choose the "Menu Level" sub-menu
- Click on the "Expert" menu option




After setting the Menu Level to "Expert", you can then access the "Joystick Control - Game" station by:

- Downloading your program and starting the debugger
- Going to the "Robot" menu
- Choose the "Debug Windows" sub-menu

- Click on the "Joystick Control - Simple" menu option to open the Controller Station



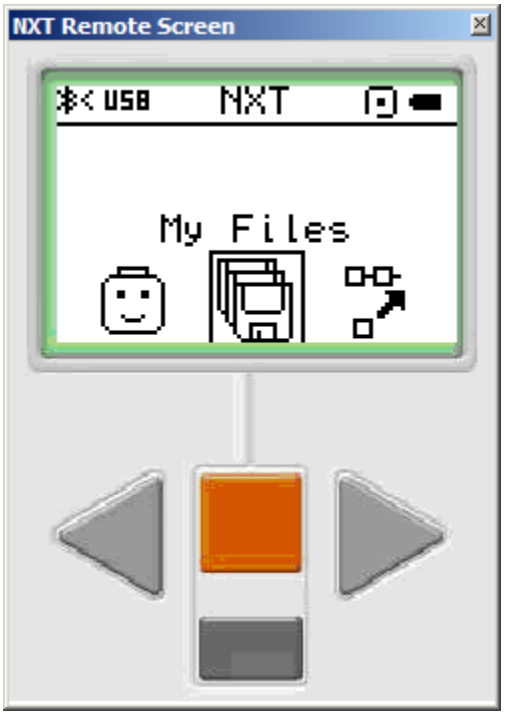

As you can see, the window has a few more options than the "Joystick Control - Simple" screen did.


If you would like to use two controllers, click the "Dual Joysticks" button to expand the Joystick Control window to facilitate two controllers. You can assign the same controller to both Primary and Secondary Joysticks, but this is not recommended.

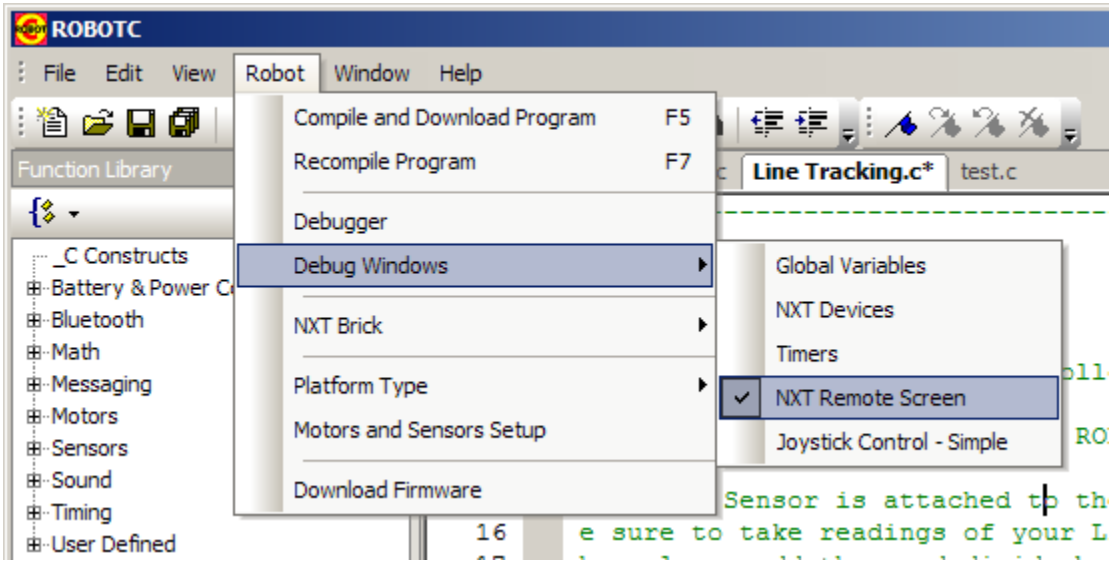

## 7.4.6 NXT Remote Screen

The NXT Remote Screen allows you to control and view your NXT from your PC. The NXT Remote Screen can only be open after a connection has been established between your computer and the robot controller.

Clicking on one of the four on-screen NXT buttons simulates an actual button press. For the LCD screen to update, the Program Debug window must be set to update Continuously.



The NXT Remote Screen can be opened by going to the Robot menu, Debug Windows, and selecting NXT Remote Screen.




## 7.4.7 System Parameters

ROBOTC has built-in variables that customize the performance and behavior of your robot. The "System Parameters" window provides PC access to these variables. The example below is for the NXT robot controller. All of these variables are directly accessible within your program code. Although they are not usually needed, an advanced user may find a few of the fields particularly interesting.

- 'avgBackgroundTime' is the overhead time spent in the device driver code. The remaining CPU time is available for user program execution. This example shows background overhead at 13%.
- 'bNoPowerDownOnACAdaptor' is a neat variable to prevent the NXT from automatically powering down if a rechargeable battery is being used and it is connected to AC power.

**System Parameters**

| Index | Variable | Value |
|---|---|---|
| 0 | nClockMinutes | 18 |
| 1 | version | 7.85 |
| 2 | nSysTime | 18:22.201 sec |
| 3 | nPgmTime | 0.000 sec |
| 4 | nxtHS_Mode | hsInactive |
| 5 | nxtHS_Status | HS_Unassigned |
| 6 | nAvgBatteryLevel | 8.51 V |
| 7 | nImmediateBatteryLevel | 8.52 V |
| 8 | LowVoltageBatteryCountLimits | 6 |
| 9 | bNoPowerDownOnACAdaptor | true |
| 10 | bNxtRechargable | true |
| 11 | nPowerDownDelayMinutes | 60 |
| 12 | nPowerDownDelayMinutesDefault | 60 |
| 13 | nOpcodesPerTimeslice | 255 |
| 14 | nDebugTaskMode | 3 |
| 15 | bFloatConversionErrors | true |
| 16 | bClearVariablesOnPgmStart | true |
| 17 | bFloatDuringInactiveMotorPWM | false |
| 18 | nVirtualMotorChanges | 0 |
| 19 | nVirtualSensorTypeChanges | 0 |
| 20 | nVirtualSensorModeChanges | 0 |
| 21 | nDatalogSize | 4665 |
| 22 | avgBackgroundTime | 13% |
| 23 | avgInterpreterTime | 0% |
| 24 | nNxtButtonPressed | 0xFFFFFFFF |
| 25 | nNxtButtonTask | 255 |
| 26 | nNxtExitClicks | 1 |
| 27 | nSyncedMotors | synchNone |
| 28 | nSyncedTurnRatio | 0 |
| 29 | nPidUpdateInterval | 25 |
| 30 | nPidUpdateInterval12V | 50 |
| 31 | nMaxRegulatedSpeedNxt | 1000 |
| 32 | nMaxRegulatedSpeed12V | 4000 |
| 33 | nLCDContrast | 90 |
| 34 | nLCDContrastDefault | 90 |
| 35 | nI2CRetries | 2 |
| 36 | nAvailFlash | true |
| 37 | bNxBTEnabled | true |
| 38 | bBTVisble | true |
| 39 | bBTBusy | false |
| 40 | bBTHasProgressSounds | true |
| 41 | bBTDebugTrace | true |
| 42 | bBTSkipPswdPrompt | false |
| 43 | nLastBTCommand | BTON |
| 44 | nBluetoothState | BT_STATE_PORT_OPEN \| B... |
| 45 | nBTCurrentStreamIndex | -1 |
| 46 | NXTGMessagingCompatability | 0 |
| 47 | bBTOptimizePerformance | false |
| 48 | nBTStreamSwitchDelay | 100 |
| 49 | bHideDataFiles | true |
| 50 | nMaxDataFileSize | 80 |
| 51 | bUltrasonicFilterCnt | false |
| 52 | nAvailableUserFlash | 93.9K |
| 53 | nVolume | 4 |
| 54 | bPlaySounds | true |

System Parameters | Timers

# 8. NXT Functions

## 8.1 Battery and Power Control

The NXT firmware keeps track of the minutes of inactivity on a NXT. When this exceeds a configurable limit, then the NXT will automatically power off. Inactivity is defined as time without a user keypress.

The inactivity timeout value can be set via the NXT GUI. It can also be set via editing ROBOTC property pages. Whenever ROBOTC makes a connection to a NXT it will set the timeout value to the value from the property page configuration.

Note that the NXT can behave unpredictably at low voltages and when trying to write to flash memory. So whenever ROBOTC attempts a connection between PC and NXT, it checks the battery voltage. If the voltage is too low for reliable operation, the ROBOTC connection will be aborted.

---

**alive();**
Periodically calling this function will reset the timer used to put the controller to sleep after long periods of keyboard inactivity.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\Play All Sounds.c")*

```
alive();                    // Reset sleep timer.
```

**powerOff();**
Immediately powers off the controller.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Battery Level.c")*

```
powerOff();                 // Powers off the robot.
```

**bNoPowerDownOnACAdaptor**
A flag to tell the NXT not to power down if it is connected to an AC power source. Set to True to enable this functionality.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Battery Level.c")*

```
bNoPowerDownOnACAdaptor = true;       // The robot will never power down if connected to A
```

**bNxtRechargable**
Boolean variable. Indicates whether the NXT is currently using a rechargeable battery pack.

*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Battery Level.c")*

```
if(bNxtRechargable)               // Checks to see if th NXT is currently using a recharea
{
  // Code goes here
}
```

**LowVoltageBatteryCountLimits**

The NXT will generate a "low voltage" pop up menu on its display when "N" consecutive battery voltage samples are at a low voltage. "N" is the "LowVoltageBatteryCountsLimits" variable. If the value is zero then the pop up window is disabled.

```
LowVoltageBatteryCountLimits  = 10;   // The low voltage screen will not appear until
                                      // 10 consecutive low battery voltages readings.
```

**nAvgBatteryLevel**

The average battery level in millivolts. (A value of 9458 represent 9.458 volts.) It's the average of 20 recent samples and smoothes out sudden voltage transients.

*Example:* *(program example can be found at: "\Sample Programs\NXT\Robot Sample Code\NXT Battery Level.c")*

```
int batteryAvg = nAvgBatteryLevel;   // Assigns variable 'batteryAvg' the average battery
```

**nImmediateBatteryLevel**

The last sampled battery level in millivolts. (A value of 9458 represent 9.458 volts.)

*Example:* *(program example can be found at: "\Sample Programs\NXT\Robot Sample Code\BalancingRbt2.c")*

```
int batteryLevel = nImmediateBatteryLevel;   // Assigns variable 'batteryLevel' the curren
                                             // battery level in millivolts.
```

**externalBattery**

Instantaneous battery voltage (in millivolts) for external 12V battery used to power HiTechnic controllers.

A value of -1 is returned if the external battery cannot be read. Only the HiTechnic Motor Controller has battery monitoring capabilities. The HiTechnic Servo Controller cannot moniotr battery. So at least one Motor Controller is required for the battery moniotiring capability.

```
int ex batteryAvg = externalBattery;  // Assigns variable 'exbatteryAvg' the current batte
millivolts.
```

**externalBatteryAvg**

Average battery voltage (in millivolts) for external 12V battery used to power HiTechnic controllers.

A value of -1 is returned if the external battery cannot be read. Only the HiTechnic Motor Controller has battery monitoring capabilities. The HiTechnic Servo Controller cannot moniotr battery. So at least one Motor Controller is required for the battery moniotiring capability.

```
int ex batteryLevel = externalBatteryAvg;     // Assigns variable 'exbatteryLevel' the aver
                                              // battery level in millivolts.
```

**nPowerDownDelayMinutes**

Specifies the number of minutes of inactivity (i.e. no buttons pressed) used by the sleep timer for automatic power off of controller. The controller automatically powers off to conserve battery power.

*Example:* *(program example can be found at: "\Sample Programs\NXT\Robot Sample Code\NXT Battery Level.c")*

```
nPowerDownDelayMinutes = 15;                  // The NXT will sleep after 15 minutes of inact
```

# 8.2 Control Structures

Control Structures:

```
if(expression)
{
  statement1
}
else
{
  statement1
}
```

In the `if` statement, if the expression in parentheses is nonzero (true), control passes to `statement 1`. If the `else` clause is prese (false), control will pass to `statement 2`. The "else" part is optional, and if absent, a false expression will simply result in skipping ov `else` always matches the nearest previous unmatched `if`; braces may be used to override this when necessary, or for clarity.

```
if(SensorValue(touch1) == 1)          // If the touch sensor 'touch1' reads '1' (pressed):
{
  displayNextLCDString("Touch pressed");   // Display "Touch pressed" to the VEX LCD.
}
else                                  // Else (the touch sensor 'touch1' reads '0' (unpressed))
{
  displayNextLCDString("Touch unpressed"); // Display "Touch unpressed" to the VEX LCD.
}
```

---

```
switch(expression)
{
  case label1:
    statements 1

  case label2:
```

```
    statements 2
    break;

  default:
    statements 3
}
```

The switch statement causes control to be transferred to one of several statements depending on the value of an expression, which mus
Substatement controlled by a switch is typically compound. Any statement within the substatement may be labeled with one or more cas
keyword `case` followed by a constant expression and then a colon (`:`).

No two of the case constants associated with the same `switch` may have the same value. There may be at most one `default` labe
none of the case labels are equal to the expression in the parentheses following `switch`, control passes to the `default` label, or if t
execution resumes just beyond the entire construct. Switches may be nested; a `case` or `default` label is associated with the innerm
Switch statements can "fall through", that is, when one `case` section has completed its execution, statements will continue to be execut
a `break;` statement is encountered. Fall-through is useful in some circumstances, but is usually not desired. In the preceding example
statements `statements 2` are executed and nothing more inside the braces. However if `label1` is reached, both `statements`
executed since there is no `break` to separate the two `case` statements.

```
int nTaskToStart = 2;   // int 'nTaskToStart' is set to '2'.

switch(nTaskToStart)    // Test 'nTaskToStart' in the switch.
{
  case 1:           // If 'nTaskToStart' is '1':
    StartTask(One);       // Start task One.
    break;                    // Break out of this switch statement and continue code after the '}'.

  case 2:           // If 'nTaskToStart' is '2':
    StartTask(Two);       // Start task Two.
    break;                    // Break out of this switch statement and continue code after the '}'.

  default:          // If 'nTaskToStart' is anything other than '1' or '2':
    StartTask(Three);     // Start task Three.
}
```

* In this example only task Two is ever started. Had `nTaskToStart` been '1' then task only One would have bee
`nTaskToStart` had been anything other than '1' or '2' task Three would have been the only one to start. There is
switch statement as each case has a `break` to jump out of the switch once that case is finished.*

---

```
while(expression)
{
  statements
}
```

The while loop will run the statements between the braces over and over as long as the expression results in a non zero value (true). On
the program will skip beyong the while loop and move on. Since the while loop always checks the condition of the expression first, it is p
runs the statements within the braces.

```
while(SensorValue(sonarSensor) > 20)    // While the Sonar Sensor reads data greater than '20':
{
  motor[rightMotor] = 63;               // Run 'rightMotor' at power level 63.
  motor[leftMotor]  = 63;               // Run 'leftMotor' at power level 63.
}

motor[rightMotor] = 0;                  // Stop 'rightMotor'.
motor[leftMotor]  = 0;                  // Stop 'leftMotor'.
```

* In this example, the loop will run the motors forward at a power level of 63 as long as the sonar sensor reads va
it reaches 20 it will skip the loop, running the code after the structure, stopping the motors. This program would
until it was 20 units away from an object.*

---

```
do
{
  statements
}while(expression)
```

The do-while loop will run the statements between the braces over and over as long as the expression results in a non zero value (true).
false, the program will skip beyong the while loop and move on. Since the do-while loop always checks the condition of the expression la
the statments at least once.

```
int bursts = 0;                 // Create variable 'bursts' of type int and set it to '0'.

do
{
  motor[rightMotor] = 63;               // Run 'rightMotor' at power level 63.
  motor[leftMotor]  = 63;               // Run 'leftMotor' at power level 63.

  bursts = bursts + 1;                          // Increment 'bursts' by 1.

  wait1Msec(3000);                              // Wait for 3000 milliseconds before continuing.

}while(bursts < 3)              // While 'bursts' is less than 3:

motor[rightMotor] = 0;                  // Stop 'rightMotor'.
motor[leftMotor]  = 0;                  // Stop 'leftMotor'.
```

* In this example, the loop will run the motors forward at a power level of 63 in three second bursts. It will run f
the code after the structure, stopping the motors. It will always run for at least 1 three second burst before checki
the expression.*

bursts: 0 1 2 3

iterations: - 1 2 X

```
for(expression1; expression2; expression3)
{
  statements
}
```

The for loop runs the code between its braces as long as `expression2` is true (non zero). It is a neat and compact solution to loopin

```
for(int i=0; i<3; i++)    // Initialize int 'i' to 0, and run the loop as long as 'i' is less than 3,
after each iteration of the loop.
{
  motor[rightMotor] = 63;          // Run 'rightMotor' at power level 63.
  motor[leftMotor]  = 63;          // Run 'leftMotor' at power level 63.
  wait1Msec(3000);                                  // Wait for 3000 milliseconds before continuing.
  motor[rightMotor] = 63;          // Run 'rightMotor' at power level 63.
  motor[leftMotor]  = -63;          // Run 'leftMotor' at power level -63.
 wait1Msec(500);                 // Wait for 500 milliseconds before continuing.
}

motor[rightMotor] = 0;           // Stop 'rightMotor'.
motor[leftMotor]  = 0;           // Stop 'leftMotor'.
```

* In this example, the loop will run the motors forward at a power level of 63 for three seconds and then turn left will repeat this process 4 times, completing a square.*

i: 0 1 2 3 4

iterations: 1 2 3 4 X

# 8.3 Bluetooth

For more information on Bluetooth with ROBOTC, please see the "Bluetooth and the NXT" section.

**btConnect(nPort, sFriendlyName)**
Attempts to connect BT device with specified "sFriendlyName" on port 'nPort'. Port should be in the range 1 to 3.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
btConnect(1, "NXT2");              // Connects to NXT named "NXT2" on port 1.
```

**btDisconnect(nPort)**
This function will disconnect a single existing Bluetooth connection on the NXT. 'nPort' ranges from 0 to 3. Port 0 is used if ths is a 'slave' Bluetooth device. Ports 1 to 3 are used when this is a 'master' Bluetooth device and are for the three possible slaves.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
btDisconnect(1);                    // Disconnects from port 1.
```

**btDisconnectAll()**
This function disconnects all existing Bluetooth connections on the NXT.
*Example:*

```
btDisconnectAll();                  // Disconnects from all ports!
```

**btFactoryReset()**
This command will reset the NXT's bluetooth module to the default factory settings. All existing connections are disconnected. The "My Contacts" list is emptied. Any existing paired connections are lost. This command is used for restoring the NXT Bluetooth operation to its original condition. It's really only needed because it's possible the Bluetooth hardware gets confused and in an inconsistent state and this is a method of "last resort" to recover.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
btFactoryReset();                   // Resets the Bluetooth to factory defaults.
```

**btRemoveDevice(sFriendlyName)**
Removes a device with the specified "sFriendlyName" from the "My Contacts" list. Removing a device will erase the "paired connection" for the device.
*Example:*

```
btRemoveDevice("HAL9000");          // HAL 9000 is no longer a friendly.
```

**btRequestLinkQuality(nPort)**
Requests the current BT link quality from a specific port. The function returns immediately before the link quality has been retrieved. The NXT CPU will send a message to the Bluetooth module requesting the data. The data is then returned to the NXT CPU within a few 100 milliseconds.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Handshake.c")*

```
int portQuality = btRequestLinkQuality(1);          // Returns the quality on port 1.
```

**btSearch()**
Begins a search for BT devices and adds new entries to the "My Contacts" lists. The search can take up to 30 seconds to perform. 'btSearch' function returns immediately. Application code should continuously check the status to wait for the search to complete.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
btSearch();                          // Search for new Bluetooth Devices.
```

**btStopSearch()**
This function terminates an existing search on the NXT.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
btStopSearch();                      // Aborts search for Bluetooth Devices.
```

**cCmdBTCheckStatus(nStream)**
Used to check the status of the BT.
*Example:*

```
cCmdBTCheckStatus(nBTCurrentStreamIndex);          // Check Status of Bluetooth.
```

**cCmdBTPurgeRcvBuffer()**
Used to purge data from the BT input buffer. Normally application program would not need to use this command.
*Example:*

```
cCmdBTPurgeRcvBuffer();              // PURGE BUFFER!
```

**cCmdMessageAddToQueue(nQueueID, pData, nLength)**
Adds the message at 'pData' to queue 'nQueueID'. 'nLength' is the length of the message.
*Example:*

```
cCmdMessageAddToQueue(nQueueID, pData, nMaxLength);          // Add message to queue to be
```

**cCmdMessageGetSize(nQueueID)**
Returns the size of the message at the head of queue 'nQueueID'.
A non-zero value indicates that a message is available in the NXT's queue of received messages. A zero value indicates that the queue is empty.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Handshake.c")*

```
int messageSize = cCmdMessageGetSize(nQueueID);              // Assigns the message size t
```

**cCmdMessageRead(pData, nLengthToRead, nQueueID)**
Reads a Bluetooth message from 'nQueueID'.
The message data is stored in the buffer at 'pData'. 'nLengthToRead' is the maximum number of bytes to read -- it is used to prevent overwriting beyond the length of the buffer at 'pData'.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Handshake.c")*

```
TFileIOResult messageIn = cCmdMessageRead(pData, nLengthToRead, nQueueID);      // Read me
```

**cCmdMessageWriteToBluetooth(nStream, pData, nLength, nQueueID)**
Writes a Bluetooth message to 'nQueueID'. The message will be written to the specified 'port' or 'nStream' which should be in the range of 0 to 3. This command is only useful when multiple ports are simultaneously open on the NXT; a configuration that is not recommended because of the much slower communications when multiple ports are in use. The message data is taken from the buffer at 'pData'. 'nLength' is the number of bytes in the message; the maximum length is 58 bytes.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Handshake.c")*

```
TFileIOResult messageOut = cCmdMessageWriteToBluetooth(2,pData, nLengthToRead, nQueueID);
// Write message to a queue on port 2.
```

**cCmdMessageWriteToBluetooth(pData, nLength, nQueueID)**
Writes a Bluetooth message to 'nQueueID'. The message will be written to the currently open stream; the ROBOTC firmware will automatically determine the 'port' or 'stream' to use. This allows the same function call to be used at both slave (port is always 0) or master (port is one of 1 to 3) devices. The message data is taken from the buffer at 'pData'. 'nLength' is the number of bytes in the message; the maximum length is 58 bytes.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Handshake.c")*

```
TFileIOResult messageOut = cCmdMessageWriteToBluetooth(pData, nLengthToRead, nQueueID);
// Write message to a queue on the currently open stream.  (nBTCurrentStreamIndex)
```

**getDefaultPIN(passCode);**
Function retrieves the default BT passcode stored in flash. Immediately after firmware download, the default is set to "1234". ROBOTC has a function that will let you redefine the default port; either for a single power on session or on a permanent basis.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
String passcode = "";          // Creates and empty String variable, 'passcode'.
getDefaultPIN(passCode);       // stores the default pin ("1234") to String variable, 'pass
```

**getSessionPIN(passCode);**
Function retrieves the "session" BT passcode stored in RAM. Upon power up, the default passcode is copied to the "session" passcode. Whenever firmware needs a BT passcode, it uses the current setting of the "session" passcode. F/W needs passcode for initializing the manual entry passcode. Also needs passcode if the "use default (i.e. session) passcode without entering manually" variable is set.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
getSessionPIN(sessionPin);      // 'sessionPin' must be of type, String! (String sessinPin
```

**nxtReadRawBluetooth(pData, nMaxBufferSize)**
Function used to read "raw" data bytes from the bluetooth module. 'pData' is the location of buffer where bytes should be stored and 'nMaxBufferSize' is the size of this buffer. The returned value is the number of bytes that was actually read. The data is retrieved from the currently open 'port' or 'stream'. A typical use of this advanced function is for communicating with devices -- like a BT enabled GPS receiver -- that do

not follow the LEGO defined "Fantom" messaging protocol. Raw data transmission should only be used when there is a single BT connection on the NXT. The function "setBluetoothRawDataMode()" needs to be called to setup raw transmission.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT GPS Task.c")*

```
int bufferSize = 1;                                 // will be the size of buffer.
ubyte BytesRead[bufferSize];                        // create a ubyte array, 'BytesRead' o
nxtReadRawBluetooth(BytesRead[0], bufferSize);      // store 'bufferSize' amount of bytes
```

**nxtWriteRawBluetooth(nStream, pData, nLength)**
Function used to write "raw" data bytes from the bluetooth module. 'pData' is the location of buffer where bytes should be stored and 'nMaxBufferSize' is the size of this buffer. The returned value is the number of bytes that was actually read. The data is sent to the specified 'nSstream'. A typical use of this advanced function is for communicating with devices -- like a BT enabled GPS receiver -- that do not follow the LEGO defined "Fantom" messaging protocol. Raw data transmission should only be used when there is a single BT connection on the NXT. The function "setBluetoothRawDataMode()" needs to be called to setup raw transmission.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT LowLevel.c")*

```
int sendSize = 1;                                   // we will be sending this many bytes
ubyte BytesToSend[sendSize];                        // create a ubyte array, 'BytesToSend'

nxtReadRawBluetooth(nBTCurrentStreamIndex , BytesRead[0], bufferSize);    /* send 'sendSize
                                                                          /* from 'BytesToS
                                                                          /* currently open
```

**resetSessionPIN();**
Function resets the default BT passcode stored in RAM to the default value stored in flash.
*Example:*

```
resetSessionPIN();          // restores the session passcode to the default value stored in
```

**setBluetoothOff()**
This function turns the Bluetooth module OFF. Setting BT off will disconnect any existing connections.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
setBluetoothOff();          // Turn the Bluetooth OFF.
```

**setBluetoothOn()**
This function turns the Bluetooth module ON.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
setBluetoothOn();           // Turn the Bluetooth ON.
```

**setBluetoothRawDataMode()**
This function sets the NXT Bluetooth configuration to "raw data transmission" mode. When in "raw mode" the LEGO defined higher level protocol (i.e. Fantom messages) is disabled and the application program

can directly 'read' and 'write' bytes over an existing Bluetooth connection. "raw mode" can only be exited by the NXT firmware when an application program terminates execution.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT LowLevel.c")*

```
setBluetoothRawDataMode();          // Set Bluetooth to "raw mode".
```

**setBluetoothVisibility(bBluetoothVisible)**
This functions makes the NXT 'visible' or 'invisible' to search requests from other bluetooth devices. Connections can still be made to an 'invisible' device as long as the far end device already knows the BT address of the device.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
setBluetoothVisibility(true);       // NXT is now Visible.
setBluetoothVisibility(fasle);      // NXT is now Invisible.
```

**setDefaultPIN(passCode);**
Function sets the default BT passcode stored in flash. Immediately after firmware download, the default is set to "1234". This function allows you to change the default. The new default value is "permanent"; it is retained until the firmware is reloaded.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
String PIN1 = "4246";          // Create a String variable, 'PIN1' and initialize it to "42
setDefaultPIN(PIN1);           // Sets the default passcode to "4246".
```

**setFriendlyName(sFriendlyName)**
This function is used to set the "friendly name" of the NXT. This is the name displayed centered on the top line of the NXT's LCD display.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
setFriendlyName("Odysseus");       // The NXT is now named, "Odysseus".
```

**setSessionPIN(passCode);**
Function sets the "session" BT passcode stored in RAM. Upon power up, the default passcode is copied to the "session" passcode. Whenever firmware needs a BT passcode, it uses the current setting of the "session" passcode. Firmware needs a passcode for initializing the manual entry passcode. Also needs passcode if the "use default (i.e. session) passcode without entering manually" variable is set.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
String PIN2 = "6670";          // Create a String variable, 'PIN2' and initialize it to "66
setSessionPIN(PIN2);           // Sets the session passcode to "6670".
```

**transferFile(nPort, sFileName);**
Transfers the file 'sFileName' from this NXT to the NXT connected to port 'nPort'.
*Example:*

```
String file = "EnigmaCodes.dat";   // Create a String variable, 'file' and initialize it t
```

```
transferFile(2, file);              // Send file, EnigmaCodes.dat to BT device connected on
```

**bBTBusy**
Read-only boolean variable that indicates whether Bluetooth is currently busy processing a command.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
while(bBTBusy)                                  // While the Bluetooth is busy (bBTBusy =
{
  nxtDisplayCenteredBigTextLine(3, "BT is BUSY");   // Display, "BT is BUSY" in big, cente
  wait1Msec(100);                                   // Wait 100 milliseconds (helps displa
}
```

**bBTDebugTrace**
Boolean variable for advanced users. Enables output of a debug trace of Bluetooth activity to the "Debug Stream" debugger output window.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
bBTDebugTrace = true;       // Generate messages on internal BT operation to "debug stream"
bBTDebugTrace = false;      // Do NOT generate messages on internal BT operation to "debug
```

**bBTHasProgressSounds**
Boolean variable that enables or disables sound feedback on Bluetooth connect/disconnect and failure activities. Depending on whether a connection is "paired" connect and disconnect activities can occur silently. This variable will enable audible output to inform you of these actions.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT Bluetooth Test.c")*

```
bBTHasProgressSounds = true;       // Play speaker tones when BT connected / disconnected.
bBTHasProgressSounds = false;      // Do NOT play speaker tones when BT connected / disconn
```

**bBTRawMode**
Read-only variable that can be used to test whether NXT Bluetooth is currently in "raw data transmission mode". When in "raw mode" the LEGO defined higher level protocol (i.e. Fantom messages) is disabled and the application program can directly 'read' and 'write' bytes over an existing Bluetooth connection.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT LowLevel.c")*

```
setBluetoothRawDataMode();       // Set Bluetooth to "raw mode".
while (!bBTRawMode)              // While the Bluecore is still NOT in raw mode (bBTRawMode
{
  wait1Msec(1);                     // Wait for Bluecore to enter raw data mode.
}
```

**bBTSkipPswdPrompt**
Boolean variable used to indicate whether manual entry of Bluetooth password should be disabled. If set, then the default value stored in RAM (see the set/get session passcode functions will always be used. Password entry using the NXT buttons and LCD can be awkward. This is a convenient way to bypass this entry if you're always going to use the same passcode.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
bBTSkipPswdPrompt = true;       // Manual entry of Bluetooth password will be disabled.
bBTSkipPswdPrompt = false;      // Manual entry of Bluetooth password will NOT be disabled.
```

**bBTVisble**
Read-only boolean variable that indicates whether Bluetooth is 'visible' (true) or 'invisible' (false)
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT GPS Task.c")*

```
if(bBTVisble)                                       // If the Bluetooth is Visible:
{
  eraseDisplay();                                       // Erase the current display.
  nxtDisplayCenteredBigTextLine(3, "Going Loud");       // Display in big, centered type on
}
else                                                // Else (the Bluetooth is Invisible):
{
  eraseDisplay();                                       // Erase the current display.
  nxtDisplayCenteredBigTextLine(3, "Going Silent");     // Display in big, centered type on
}
```

**nBluetoothCmdStatus**
Gets the status/progress of the last BT issued 'command'. Can be used to check whether the command is still in progress. Once completed the status contains the success or fail status of the command.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT Passcodes.c")*

```
TFileIOResult status = nBluetoothCmdStatus;       // 'status' gets set to the current Bluet
```

**nBTCurrentStreamIndex**
Read-only variable containing the current active stream index. Value 0 is used for slave bluetooth devices. Value 1 to 3 is used for the three possible devices that can be connected on a master BT device. Value -1 is used if BT if there are no current BT connections. This variable is useful for checking if there is a current BT connection and whether the device is master or slave.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT GPS Task.c")*

```
if(nBTCurrentStreamIndex >= 0)    // If there is currently an open Bluetooth connection:
{
  return;                           // Return success.
}
```

**nLastBTCommand**
Gets the last command processed by the Bluetooth firmware.
*Example:*

```
TFileIOResult 'lastCmd' = nLastBTCommand;       // 'lastCmd' gets set to the last Bluetooth
```

# 8.4 Buttons

There are four buttons on the NXT front panel. The LEFT, ENTER and RIGHT buttons are on the top row and the EXIT button is on the second row.

Normally, when a program is running, the LEFT and RIGHT buttons are used to toggle between various standard displays. And the EXIT button is used to stop execution of the user program. These characteristics can be overwritten so that button actions can be controlled within a running application program.
There are several sample programs (e.g. Centipede, Tetris) included in the ROBOTC distribution that utilize application program control over buttons.

**nNxtButtonPressed**
Contains the number (0 to 3) of the button that is currently depressed. -1 indicates no button is currently pressed. Note that only one button press can be recognized at a time. This is a limitation in the NXT hardware. It is unable to recognize multiple button presses.
***NOTE* 0 = Gray Rectangle button. 1 = Right Arrow button. 2 = Left Arrow button. 3 = Orange Square button.***
*Example: (program example can be found at: "\Sample Programs\NXT\Encoder\Bread Crumbs.c")*

```
if(nNxtButtonPressed == 1)                          // If the current pressed button is 1
{
  nxtDisplayCenteredBigTextLine(3, "RIGHT ARROW");      // Display on line 3, a big, cente
}
```

**nNxtButtonTask**
The variable serves two purposes. A negative value (default is -1) indicates that the standard firmware processing of NXT buttons should be used. Values in the range of 0 or higher indicates that application program will process buttons. If the value is in the range 0 to 9 (i.e. a valid task number) then this task will be started (or restarted) whenever a button press is detected.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Button.c")*

```
nNxtButtonTask = -2;      /* Grab control of the buttons.  '-3' is invalid for a task, */
                          /* so no task will be executed when a button is pressed.     */
```

**nNxtExitClicks**
Holds the number of 'clicks' of the EXIT button required to abort a program. Default value is one for compatibility with standard firmware. This variable allows end user programs to use the EXIT button. If you set the variable to 'N' (where 'N' is greater than 1) then the first 'N-1' consecutive clicks of the EXIT button can be handled by the application program. On the N-th consecutive click the firmware will automatically EXIT (i.e. end) your program. Having the firmware perform the check ensures that, even if there is an error in your program, you can always EXIT your program. But it also easily allows your program to have access to the EXIT button. NOTE: With the standard firmware, the EXIT button is not accessible to your program!
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Button.c")*

```
nNxtExitClicks = 3;       // Triple clicking EXIT button will terminate program.
```

# 8.5 Datalog

The NXT has a data logging capability similar to that found in the RCX. This capability allows you to save variable and sensor values in a log during program execution. This is useful for collecting data during measurements. It is also useful for program debugging where you may want to analyze the value of a particular variable after a program has been run. There's room for storage of about 5,000 data points.

The data points are stored into RAM. They can be subsequently copied to a NXT flash file. They are not immediately stored into a flash file during program execution because writing to a file in the flash memory can take 3 to 6 milliseconds which could disturb performance of the user application.

There are utilities within the ROBOTC IDE to upload the datalog and display in a PC window. These can be found under the “debug display” and the “NXT -> Files Management” display.

---

**AddToDatalog(data)**
**AddToDatalog(nDataPtIndex, nDataValue);**
Adds an entry to the datalog. The second command adds an entry to the datalog with a user specifc "data point" type.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Datalog Test.c")*

```
int i;                          // Create variable 'i'.
for(i=0; i<10; i++)             // Cycle through "for loop" ten times (0-9), incrementing 'i' w
{
  AddToDatalog(i);                 /* Add variable 'i' to the datalog with each iteration of t
}                                  /*              (the digits 0 through 9 will be added)
```

**SaveNxtDatalog()**
Saves the existing datalog from RAM memory into a NXT file named "DATAnnnn.rdt" where 'nnnn' is numeric and increases by one on every save.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Datalog Test.c")*

```
SaveNxtDatalog();       // Saves the Datalog.
```

**bHideDataFiles**
Boolean flag to indicate whether data files should be hidden or shown in the NXT GUI.
*Example:*

```
bHideDataFiles = true;      // Data files will be HIDDEN and NOT SHOW on NXT GUI.
bHideDataFiles = false;     // Data files will be SHOWN on NXT GUI.
```

**nDatalogSize**
The current size allocated to the datalog.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Datalog Test.c")*

```
const int size = 2000;      // A constant int variable 'size' of size 2000.
nDatalogSize = size;        // Create a new Datalog of size, 'size'.
```

**nMaxDataFiles**
Maximum number of saved datalog files (DATAnnnn.RDT) allowed on a NXT.
*Example:*

```
nMaxDataFiles = 15;                                  // Allow up to 15 Datalog files on
nxtDisplayCenteredTextLine(3, "Max Datalog Files:");    // Display on a centered textline
nxtDisplayCenteredBigTextLine(4, "%d", nMaxDataFiles);  // (will display, "15")
```

**nMaxDataFileSize**
Maximum size (in 100 byte units) of all datalog files saved on the NXT.
*Example:*

```
nMaxDataFileSize = 10;                                    // 10*100 = 1000 = roughly 1 ki
nxtDisplayCenteredTextLine(3, "Max Datalog Files:");      // Display on a centered textli
nxtDisplayCenteredBigTextLine(4, "%d", nMaxDataFileSize); // (will display, "1000")
```

**nUsedDatalogBytes**
The number of bytes currently containing data in the datalog.
*Example:*

```
nMaxDataFileSize = 10;                              // 10*100 = 1000 = roughly 1 kilobyte.
while(nMaxDataFileSize - nUsedDatalogBytes > 0)     // While there are still bytes left (0
{
  AddToDatalog(data);                                  // Write to datalog (fill it up!)
}
```

# 8.6 Debug

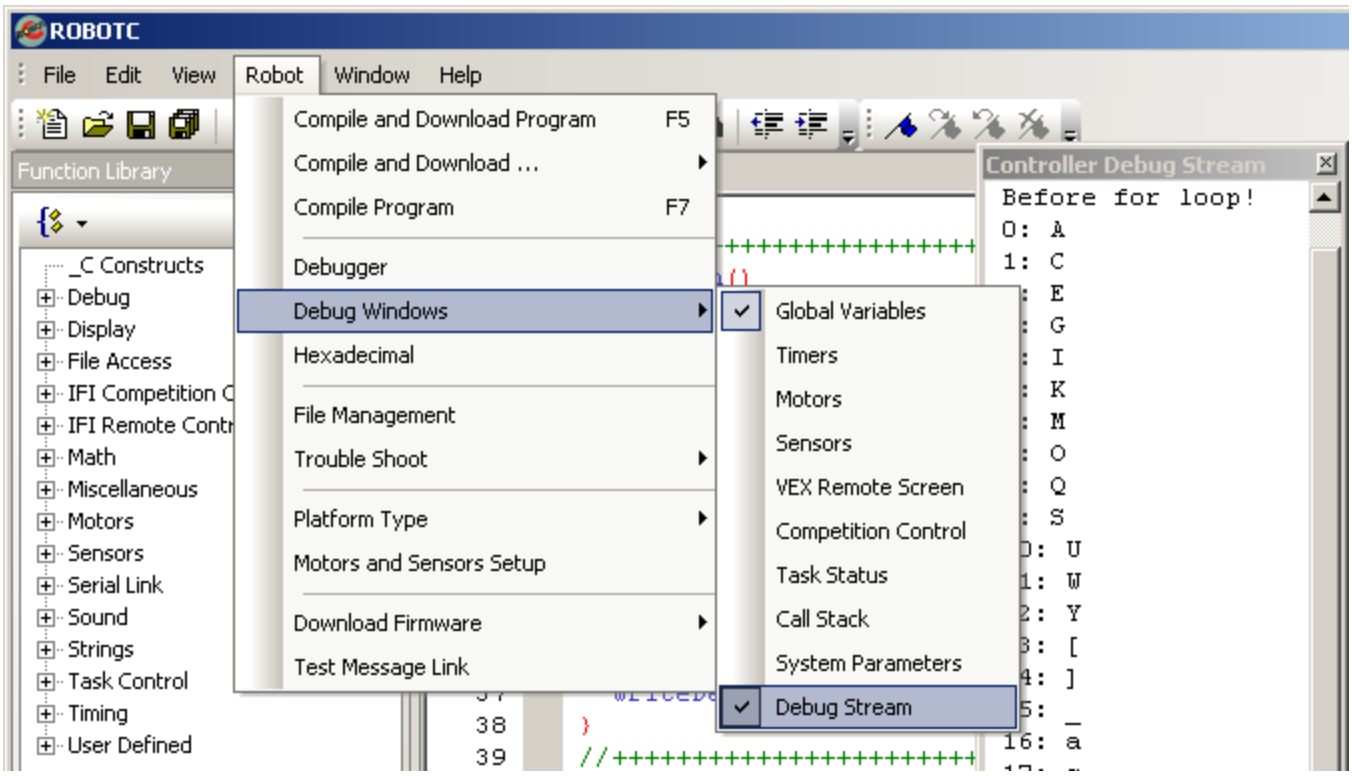
**"Traditional" Debugging Techniques**

Debugging a program – finding the errors and correcting them – can be a slow process in solutions without a run-time debugger. Without a debugger you may have to resort to different techniques like:

- There's no way to determine if your program is executing the intended logic. So you add code to play different tones/sounds to your program as it executes different "blocks" of code. You determine from the sound what is being executed within your program.
- If your robot platform supports a display device (which could be a serial link to your PC or an LCD display on the robot) then you would have to add "print" statements to your program code at various points in your program. By examining the display, you can (hopefully) determine what's happened in your program's execution by the display.

Both of the above techniques are available in ROBOTC. However, a real-time debugger eliminates the need to resort to them. There's no need to add code for debugging to your program. A built-in debugger provides better functionality without ever having to modify your source code!

There is also a built-in Debug Stream that you can use to keep track of your program from behind the scenes. For example, you could print a message to the Debug Stream when you enter and exit loops, functions, etc. Then you can view the cached Debug Stream to help in the debugging process.




```
writeDebugStream();
```
Writes a String to the Debug Stream.

```
writeDebugStream("int x is: %d", x);        // Writes the current value of int 'x' to the d
```

```
writeDebugStreamLine();
```
Writes a String to the Debug Stream starting on a new line.

```
writeDebugStreamLine("int x is: %d", x);    // Writes the current value of int 'x' to the d
                                            // on a new line.
```

# 8.7 Display

ROBOTC has a rich set of functionality for drawing text and shapes on the LCD screen. The NXT is equipped with a 100 wide by 64 pixels high display.

- The bottom left corner is point (0, 0) and the top right corner of the display is point (99, 63).
- There are eight text lines numbers 0 to 7. 0 is the top line and 7 is the bottom line of the display.

| *specifier* | Output | Example Code | Example Output |
|---|---|---|---|
| %d or %i | Signed decimal integer | "%d" | 4246 |
| %e | Scientific notation (mantise/exponent) using e character | "%e" | 3.9265e+2 |
| %E | Scientific notation (mantise/exponent) using E character | "%E" | 3.9265E+2 |

| | | | |
|---|---|---|---|
| %f | Decimal floating point | "%f" | 3.14159 |
| %o | Signed octal | "%o" | 157 |
| %s | String of characters | "%s" | ROBOTC |
| %x | Unsigned hexadecimal integer | "%x" | 7fa |
| %X | Unsigned hexadecimal integer (capital letters) | "%X" | 7FA |
| %c | Character | "%c" | b |

**NOTE on Displaying Digits:**
When displaying floats, for example, you can tell ROBOTC how many decimals places to display. This is standard across all 'C' - like programing languages. For example, if your float is **PI (3.14159265)**, but you only want to dispay **"3.14"**, your string should contain, `" %1.2f "`.

The number *before* the decimal is how many digits *before* the decimal you wish to display, while the number *after* the decimal is how many digits *after* the decimal you wish to display. So `" %1.2f "` tells us to display one digit before the decimal and two digits after the decimal, with **"3.14"** as the final result.

**The tag can also contain *flags*, *width*, *.precision* and *modifiers* sub-specifiers, which are optional and follow these specifications:**

| Specifier | Description | Example Output |
|---|---|---|
| %d | Prints a decimial integer | 50 |
| %6d | Prints a decimal integer, at least 6 characters wide | "_ _ _ _ 50" |
| %f | Prints a floating point | 50.5 |
| %6f | Prints a floating point, at least 6 characters wide | "_ _ _ 50.5" |
| %.2f | Prints a floating point, with at least 2 decimal places | "50.50" |
| %6.2f | Prints a floating point, with at least 6 characters wide and 2 decimal places | "_ _ 50.50" |

*Note: Underscore ( _ ) character denotes a space*

---

`eraseDisplay();`
Erases the complete NXT LCD display
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Draw Spiral.c")*

```
eraseDisplay();      // Erase the entire NXT LCD display.
```

`nxtClearPixel(xPos, yPos);`
Clears a single pixel on the NXT LCD screen.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Draw Spiral.c")*

```
int X = 50;              // Create and initialize variable 'X' = 50.
int Y = 32;              // Create and initialize variable 'Y' = 32.
nxtClearPixel(X, Y);     // Clear a pixel at position (X, Y).
```

**nxtDisplayBigStringAt(xPos, yPos, sFormatString, parm1, parm2);**
Formats a text string and displays it at any (X,Y) coordinate on the LCD display. Drawing uses a large font that is 16-pixels high. 'sFormatString' is the format string to use and 'parm1' and 'parm2' are two optional parameters that can be used within the format string.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Large Font.c")*

```
int printMe = 1;                                                    // Create and Initialize
int printMeToo = 2;                                                 // Create and Initialize
nxtDisplayBigStringAt(0, 31, "%d, %d", printMe, PrintMeToo);       // Displays (in large fo
```

**nxtDisplayBigStringAt(xPos, yPos, sString);**
Formats a text string and displays it at any (X,Y) coordinate on the LCD display. Text is displayed using a double height font.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Large Font.c")*

```
nxtDisplayBigStringAt(0, 31, "1, 2");      // Displays (in large font): "1, 2"
```

**nxtDisplayCenteredTextLine(nLineNumber, sFormatString, parm1, parm2);**
Formats a text string and displays it on one of the 8 possible text lines. The text is horiztonally centered on the LCD display. 'sFormatString' is the format string to use and 'parm1' and 'parm2' are two optional parameters that can be used within the format string.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT GPS Task.c")*

```
int printMe = 1;                                                    // Create and Initialize
int printMeToo = 2;                                                 // Create and Initialize
nxtDisplayCenteredTextLine(3, "%d, %d", printMe, PrintMeToo);      // Displays on line 3 (c
```

**nxtDisplayCenteredTextLine(nLineNumber, sString);**
Displays a text string on one of the 8 possible text lines. The text is horiztonally centered on the LCD display.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT GPS Task.c")*

```
nxtDisplayCenteredTextLine(3, "1, 2");      // Displays on line 3 (centered): "1, 2"
```

**nxtDisplayClearTextLine(nLineNumber)**
Erases a line of text. 'nLineNumber' specifies one of the 8 possible text lines.
*Example: (program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Touch.c")*

```
nxtDisplayClearTextLine(3);      // Clears line 3.
```

**nxtDisplayRICFile(nleft, nBottom, sFileName);**
Display a RIC (i.e. "ICON" or "Picture") file on the NXT display at the specified coordinates.
*Example: (program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Touch.c")*

```
while(true)                                          // Infinite loop:
{
  nxtDisplayRICFile(0, 0, "faceclosed.ric");          // Display the .ric file, "faceclosed.
  wait1Msec(50);                                      // Wait 50 milliseconds (helps refresh
}
```

**nxtDisplayString(nLineNumber, sFormatString, parm1, parm2, parm3);**
Formats a character string according the the format specified in sFormatString using parameters parm1, parm2, and parm3. Display the result on text line nLineNumber. The remainder of the line is not altered; so that if the result string is 8 characters only the first eight characters of the text line are updated. The contents of the remaining characters on the line are not changed. Note that parm1 and parm2 are optional parameters; the value zero will be substituted if they are not present. (They aren't necessary.)
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Trig Demo.c")*

```
while(true)                        // Infinite loop:
{
  string s1 = "Theory";                    // Create the string named s1, "Theory".
  nxtDisplayString(3, "%s", s1);           // Display the string, 's1' on line 3.
  wait1Msec(50);                           // Wait 50 milliseconds (helps refresh rate of LCD
}
```

**nxtDisplayStringAt(xPos, yPos, sFormatString, parm1, parm2, parm3);**
Formats a text string and displays it at (xPos, yPos) coordinate on the LCD display. 'sFormatString' is the format string to use and 'parm1', 'parm2', and 'parm3' are optional parameters that can be used within the format string.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Large Font.c")*

```
while(true)                          // Infinite loop:
{
  string s1 = "Theory";                      // Create the string named s1, "Theory".
  nxtDisplayString(0, 0, "%s", s1);          // Display the string, 's1' at position (x, y).
  wait1Msec(50);                             // Wait 50 milliseconds (helps refresh rate of
}
```

**nxtDisplayStringAt(xPos, yPos, sString);**
Displays a text string at (X,Y) coordinate on the LCD display
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Large Font.c")*

```
while(true)                            // Infinite loop:
{
  nxtDisplayStringAt(0, 0, "Theory");          // Display the string, "Theory" at position (
  wait1Msec(50);                               // Wait 50 milliseconds (helps refresh rate o
}
```

**nxtDisplayTextLine(nLineNumber, sFormatString, parm1, parm2, parm3);**
Formats a text string and displays it on one of the 8 possible text lines. The remainder of the line is padded with blanks. 'sFormatString' is the format string to use and 'parm1', 'parm2', and 'parm3' are optional parameters that can be used within the format string.

*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Trig Demo.c")*

```
while(true)                          // Infinite loop:
{
  string s1 = "Theory";                     // Create the string named s1, "Theory".
  nxtDisplayTextLine(3, "%s", s1);          // Display the string, 's1' on line 3.
  wait1Msec(50);                            // Wait 50 milliseconds (helps refresh rate of L
}
```

**nxtDisplayTextLine(nLineNumber, sString);**
Displays a text string on one of the 8 possible text lines. The remainder of the line is padded with blanks.
*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Trig Demo.c")*

```
while(true)                          // Infinite loop:
{
  nxtDisplayTextLine(3, "Theory");          // Display the string, "Theory" on line 3.
  wait1Msec(50);                            // Wait 50 milliseconds (helps refresh rate of L
}
```

**nxtDrawCircle(Left, Top, Diameter);**
Draws outline of the circle with the specified coordinates.
*Example:*

```
nxtDrawCircle(20, 50, 25);    // Display a circle; left border at 20, top border at 50, di
```

**nxtDrawEllipse(Left, Top, Right, Bottom);**
Draws outline of the ellipse with the specified coordinates.
*Example:*

```
nxtDrawEllipse(20, 50, 60, 25);     // Display an ellipse; left border at 20, top border at
                                    // right border at 60, bottom border at 25.
```

**nxtDrawLine(xPos, yPos, xPosTo, yPosTo);**
Draws a line between two points.
*Example:*

```
nxtDrawLine(20, 50, 60, 25);    // Display a line between the points (20,50) and (60,25).
```

**nxtDrawRect(Left, Top, Right, Bottom);**
Draws outline of the rectangle with the specified coordinates.
*Example:*

```
nxtDrawRect(20, 50, 60, 25);     // Display a rectangle; left border at 20, top border at 5
                                 // right border at 60, bottom border at 25.
```

**nxtEraseEllipse(Left, Top, Right, Bottom);**
Erases (i.e. clears all pixels) the ellipse with the specified coordinates.
*Example:*

```
nxtEraseEllipse(20, 50, 60, 25);    // Erases an ellipse; left border at 20, top border at
                                    // right border at 60, bottom border at 25.
```

**nxtEraseLine(xPos, yPos, xPosTo, yPosTo);**
Erases (i.e. clears) the pixels for the line between the specified pair of points.
*Example:*

```
nxtEraseLine(20, 50, 60, 25);    // Erase a line between the points (20,50) and (60,25).
```

**nxtEraseRect(Left, Top, Right, Bottom);**
Erases (i.e. clears all pixels) the rectangle with the specified coordinates.
*Example:*

```
nxtEraseRect(20, 50, 60, 25);    // Erase a rectangle; left border at 20, top border at 50
                                 // right border at 60, bottom border at 25.
```

**nxtFillEllipse(Left, Top, Right, Bottom);**
Fills (i.e. all pixels black) the ellipse with the specified coordinates.
*Example:*

```
nxtFillEllipse(20, 50, 60, 25);    // Display a solid ellipse; left border at 20, top bord
                                   // right border at 60, bottom border at 25.
```

**nxtFillRect(Left, Top, Right, Bottom);**
Fills (i.e. all pixels black) the rectangle with the specified coordinates.
*Example:*

```
nxtFillRect(20, 50, 60, 25);    // Display a solid rectangle; left border at 20, top borde
                                // right border at 60, bottom border at 25.
```

**nxtInvertLine(xPos, yPos, xPosTo, yPosTo);**
Inverts the pixels for the given line. This function is very useful for functions like drawing a "clock face", a "compass" or a gauge on the LCD screen when you want to "erase" the previous dial pointer and redraw it at a new position. If you use "invert line" for the original drawing and for the "erase" drawing the two calls will cancel each other out! And it work well if the line overdraws some existing pixels -- e.g. some text on the LCD.
*Example:*

```
nxtInvertLine(20, 50, 60, 25);    // Invert the pixels on the line from (20,50) to (60,25)
```

**nxtScrollText(sFormatString, parm1, parm2);**
Shift the LCD image up one line. Thenit formats a text string and displays it on the bottom line of the LCD text screen. 'sFormatString' is the format string to use and 'parm1' and 'parm2' are two optional parameters that can be used within the format string.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Scroll Text.c")*

```
for(i=0; i<1000; ++i)             // For Loop from 0 to 999 incrimenting 'i' by 1 each time
{
  nxtScrollText("Scroll #%d.", i);    // Display "Scroll #i." (%d is replaced with variabl
  wait1Msec(250);                     // Wait 520 milliseconds between each iteration of t
}
```

**nxtSetPixel(xPos, yPos);**
Sets a single pixel on the NXT LCD screen.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Draw Spiral.c")*

```
nxtSetPixel(42, 46);    // "Set" a pixel (make it black) at position (42,46).
```

**bNxtLCDStatusDisplay**
Boolean variable that indicates whether the top status line display on the LCD should be present on user-drawn LCD screens.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT LowLevel.c")*

```
bNxtLCDStatusDisplay = true;    // The NXT top status line WILL display on user-drawn LCD
bNxtLCDStatusDisplay = false;   // The NXT top status line WILL NOT display on user-drawn
```

# 8.8 File Access

**Close(hFileHandle, nIoResult);**
Closes the specified file handle. This should be the last file I/O operation after all reads or write are completed. 'nIoResult' is non-zero when error occurs.

**Delete(sFileName, nIoResult);**
Deletes the specified filename from the NXT. 'nIoResult' is non-zero when error occurs.

**FindFirstFile(hFileHandle, nIoResult, sSearch, sFileName, nFilesize);**
This function is used to start searching (iterating) through the list of files on the NXT. nIoResult is non-zero if no files exist in the NXT file system.. 'sSearch is the pattern match to use for the search string. For example:

- "***.***" will find all files
- "*.rso" will find all files with file extension "rso" which is the extension used for sound files
- 'sFileName is the name of the first file found and nFilesize is the number of data bytes in the file.

- ‘hFileHandle returns a handle used to keep track of the search. You should use “Close” when the search is finished to release the handle so that the NXT can re-use it

**FindNextFile(hFileHandle, nIoResult, sFileName, nFilesize);**
Finds the next file in a previously initiated search.

**OpenRead(hFileHandle, nIoResult, sFileName, nFileSize);**
Opens 'sFileName' for reading. 'hFileHandle' is used for subsequent reads to this file. 'nFileSize' is filled with the file length. 'nIoResult' is non-zero when error occurs.

**OpenWrite(hFileHandle, nIoResult, sFileName, nFileSize);**
Opens sFileName for writing with specified size of nFileSize.. hFileHandle is used for subsequent writes to this file. nIoResult is non-zero when error occurs.sFileName must be a valid NXT file name. The file must not already exist on the NXT for this function to succeed.

**ReadByte(hFileHandle, nIoResult, nParm);**
Reads a byte variable (8-bit) from the specified file.  'nIoResult' is non-zero when error occurs.

**ReadFloat(hFileHandle, nIoResult, fParm);**
Reads a float variable from the specified file. 'nIoResult' is non-zero when error occurs.

**ReadLong(hFileHandle, nIoResult, nParm);**
Reads a long integer variable (32-bit) from the specified file. 'nIoResult' is non-zero when error occurs.

**ReadShort(hFileHandle, nIoResult, nParm);**
Reads a short integer variable (16-bit) from the specified file. 'nIoResult' is non-zero when error occurs.

**Rename(sFileName, nIoResult, sOriginalFileName);**
Renames file 'sOriginalFileName' to 'sFileName'.

**WriteByte(hFileHandle, nIoResult, nParm);**
Writes a single byte to the specified file. 'nIoResult' is non-zero when error occurs.

**WriteFloat(hFileHandle, nIoResult, fParm);**
Writes a float variable to the specified file. 'nIoResult' is non-zero when error occurs.

**WriteLong(hFileHandle, nIoResult, nParm);**
Writes a long integer (32-bit) variable to the specified file. 'nIoResult' is non-zero when error occurs.

**WriteShort(hFileHandle, nIoResult, nParm);**
Writes a short integer variable (16-bit) to the specified file. 'nIoResult' is non-zero when error occurs.

**WriteString(hFileHandle, nIoResult, sParm);**
Writes a string to the specified file includint null terminator. 'nIoResult' is non-zero when error occurs.

**WriteText(hFileHandle, nIoResult, sParm);**
Writes a string to the specified file without null terminator. 'nIoResult' is non-zero when error occurs.

**nAvailFlash**
The amount of flash memory that is currently unused and available for file storage. Units are 1/10 of 1K (i.e. 100 bytes).

# 8.9 Joystick Control

ROBOTC supports using Logitech USB joystick controllers to drive your NXT over the USB or Bluetooth communication link. This allows the user to send commands to their robot in real time, rather than having only pre-programmed behaviors.The joystick functionality works by taking data from the joystick controller and sends it to the NXT over the debugger link as a single message.

ROBOTC has a driver included to take this Bluetooth packet and break it into data, such as variables, that your robot can use. This driver file is provided as an include file. To use this include file, add this line of code to your program:

```
#include "JoystickDriver.c"
```

This include file will create 12 new variables for your program to use. These variables can be used to assign motor speeds, active blocks of code, or control different behaviors. For more examples of using these variables, see the sample code provided under the "Remote Control" sample code folder.

Before you can use these variables, you have to "update" the variables by getting the newest packet of data from the joysticks. Because the joystick station may only send updates every 50-100ms, you should update your joystick values as often as possible to get the most up to date joystick data. To "update" your joystick variables, use this function:

```
getJoystickSettings(joystick);
```

**Sample Program:**

```
#include "JoystickDriver.c"     // Tells ROBOTC to include the driver file for the joystic

task main()
{
 while(true)
  {
    getJoystickSettings(joystick);  // Update Buttons and Joysticks
    motor[motorC] = joystick.joy1_y1;
    motor[motorB] = joystick.joy1_y2;
  }
}
```

## 8.9.1 Using Joysticks

**joystick.joy1_x1**
Value of the X Axis on the Left Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy1_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy1_y1**
Value of the Y Axis on the Left Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy1_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy1_x2**
Value of the X Axis on the Right Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x2;      // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy1_y2;      // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy1_y2**
Value of the Y Axis on the Right Joystick on Controller #1. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy1_x2;      // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy1_y2;      // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy1_Buttons**
Returns a "Bit Map" for the 12 buttons on Controller #1. For more information on how to use buttons to control actions, See the "Using Buttons" help section.
*Example:*

```
while(true)                        // Infinite loop:
{
  if(joystick.joy1_Buttons == 32)     // If Button 6 is pressed on joy1:
  {
    motor[motorA] = 50;                  // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy1_TopHat**
Returns the value of the direction pad (or "Top Hat") on Controller #1. A value of -1 is returned when

nothing is pressed, and a value of 0 to 7 for selected "octant" when pressed.
*Example:*

```
while(true)                          // Infinite loop:
{
  if(joystick.joy1_TopHat == 0)      // If the topmost button on joy1's D-Pad ('TopHat') i
  {
    motor[motorA] = 50;                  // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy2_x1**
Value of the X Axis on the Left Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy2_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy2_y1**
Value of the Y Axis on the Left Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\Single Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x1;      // MotorB's powerlevel is set to the x1 stick's c
  motor[motorC] = joystick.joy2_y1;      // MotorB's powerlevel is set to the y1 stick's c
}
```

**joystick.joy2_x2**
Value of the X Axis on the Right Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                          // Infinite loop:
{
  motor[motorB] = joystick.joy2_x2;      // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy2_y2;      // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy2_y2**
Value of the Y Axis on the Right Joystick on Controller #2. Ranges in values between -128 to +127.
*Example:* (program example can be found at: " \Sample Programs\NXT\Remote Control\NXT 2 Joystick Drive.c ")

```
while(true)                             // Infinite loop:
{
  motor[motorB] = joystick.joy2_x2;       // MotorB's powerlevel is set to the x2 stick's c
  motor[motorC] = joystick.joy2_y2;       // MotorB's powerlevel is set to the y2 stick's c
}
```

**joystick.joy2_Buttons**
Returns a "Bit Map" for the 12 buttons on Controller #2. For more information on how to use buttons to control actions, See the "Joystick Button Commands" help section.
*Example:*

```
while(true)                         // Infinite loop:
{
  if(joystick.joy2_Buttons == 32)     // If Button 6 is pressed on joy2:
  {
    motor[motorA] = 50;                 // MotorA is run at a power level of 50.
  }
}
```

**joystick.joy2_TopHat**
Returns the value of the direction pad (or "Top Hat") on Controller #2. A value of -1 is returned when nothing is pressed, and a value of 0 to 7 for selected "octant" when pressed.
*Example:*

```
while(true)                         // Infinite loop:
{
  if(joystick.joy2_TopHat == 0)       // If the topmost button on joy2's D-Pad ('TopHat') i
  {
    motor[motorA] = 50;                 // MotorA is run at a power level of 50.
  }
}
```

## 8.9.2 Using Buttons

Once you have updated your joystick values, you can now use the variables **joystick.joy1_Buttons** or **joystick.joy2_Buttons** to access a bit-masked value of all 12 buttons sent. This value, however, is not useable in your program right away.

ROBOTC has added two new functions, **joy1Btn(button)** and **joy2Btn(button)** to allow you to know which buttons have been pressed when multiple buttons are pressed at a time. Using **joy1Btn(button)** and **joy2Btn(button)**, each button returns the a value of 1 if pressed and a value of 0 if not pressed. This will allow you to write your program to work with multiple buttons simultaenously.

```
#include "JoystickDriver.c"     // Tells ROBOTC to include the driver file for the joystic

task main()
{
```

```
 while(true)
  {
    getJoystickSettings(joystick);  // Update Buttons and Joysticks

    if(joy1Btn(1) == 1)                  // If Joy1-Button1 is pressed:
    {
      motor[motorA] = 100;              // Turn Motor A On at full power
    }
    else                               // If Joy1-Button1 is NOT pressed:
    {
     motor[motorA] = 0;                // Turn Motor A Off
    }
  }
}
```



## 8.9.3 Joystick Controller Station

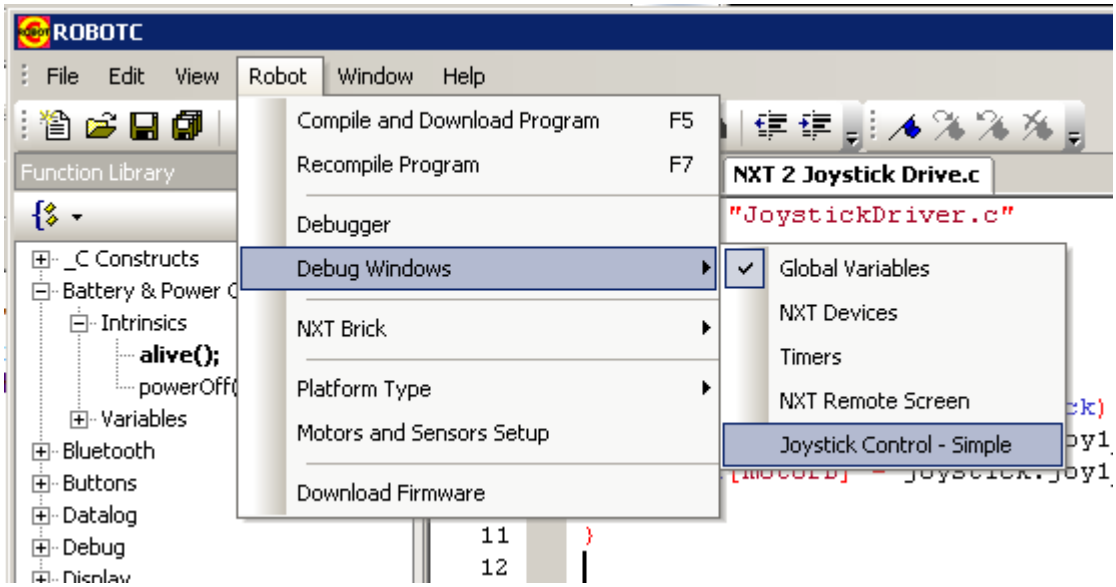
ROBOTC has built two different Joystick Controller stations built into the interactive debugger. "Joystick Control - Simple" is a debugger window to control the NXT via a Logitech USB remote control. "Joystick Control - Game" is a full featured Controller Station which is used mainly for FIRST Tech Challange or other competitions that are NXT or TETRIX based.

---

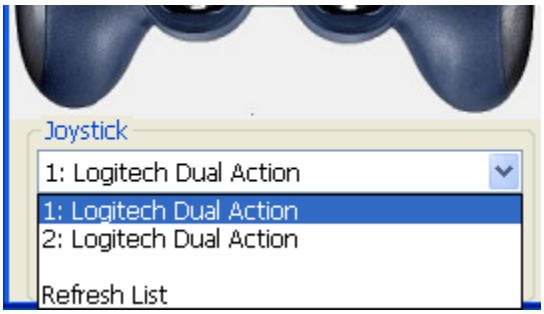
### "Joystick Control - Simple"

After your program has been downloaded and the debugger is opened, you can open the Controller Station by:

- Downloading your program and starting the debugger
- Go to the "Robot" menu
- Choose the "Debug Windows" sub-menu
- Click on the "Joystick Control - Simple" menu option to open the Controller Station




Once the Controller Station is opened, ROBOTC will look for any joysticks attached to your computer via USB. You can choose which joystick you want to robot to be controlled with by changed the joystick under the available drop-down menu. If you have no joysticks available, this list will be empty and ROBOTC will alert you that you have "No Controllers Configured"



You can see what data is being generated by the Joystick Station by looking at the X1, Y1, POV, X2, Y2 and Buttons display directly below the dropdown menu. This will give you realtime feedback of what values are being sent to your NXT from the Joystick Station. This data will also be illustrated with green dots to reflect the values and button presses.

ROBOTC will send joystick data to your NXT over Bluetooth or USB, but only when the Joystick Control window is opened. You will need to have the debugger open to use the Joystick Station.

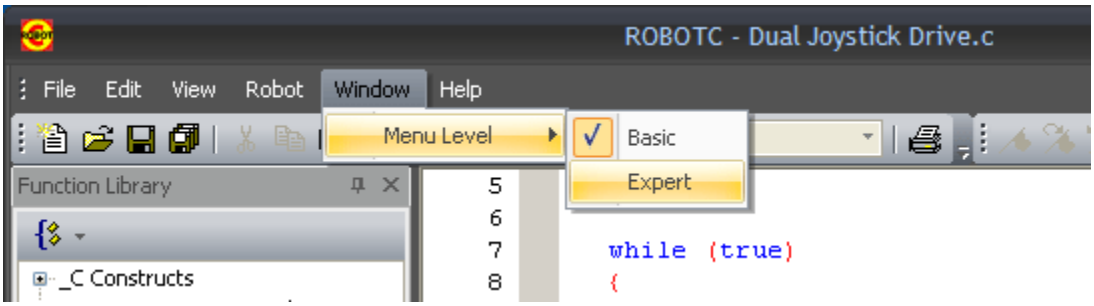

---

**"Joystick Control - Game"**

There is also a second Joystick Control window, "Joystick Control - Game". This window is specifically designed to emulate the FIRST Tech Challenge game mode. To test your FTC competition programs, you can use the controller station to mimic what the Field Management System will do. This includes switching between Autonomous and User Control, Changing if your robot is on the blue or red alliance and also disabling (or pausing) your robot. These commands can be found on the left side of the Joystick Station.

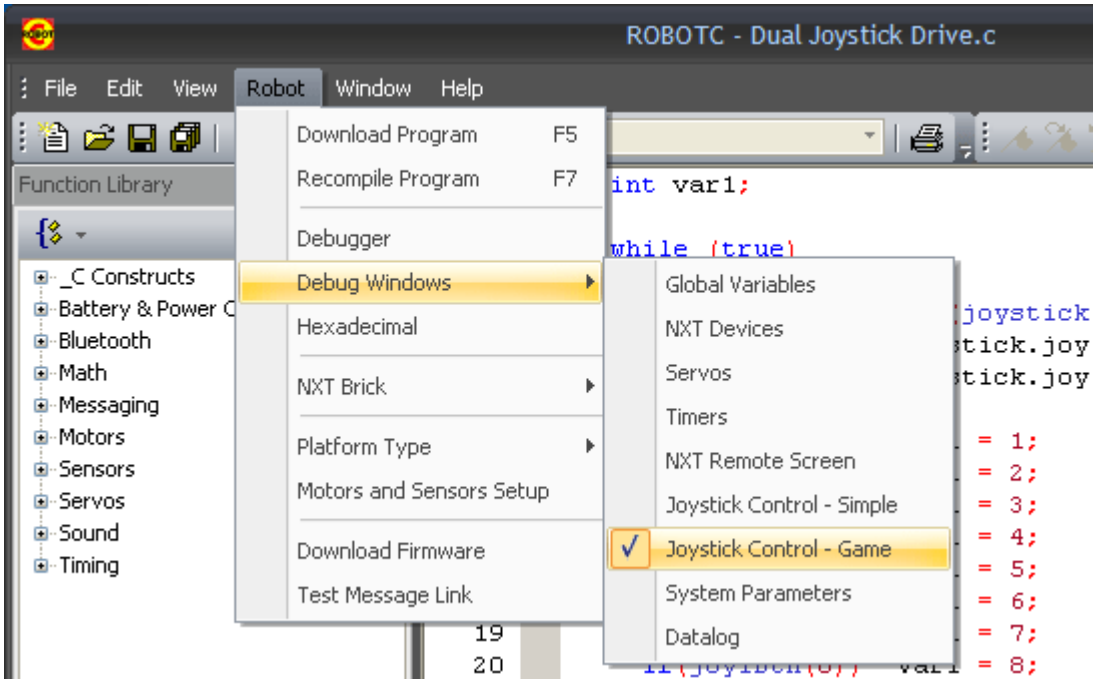
You can open the Controller Station by:

- Go to the "Window" menu
- Choose the "Menu Level" sub-menu
- Click on the "Expert" menu option

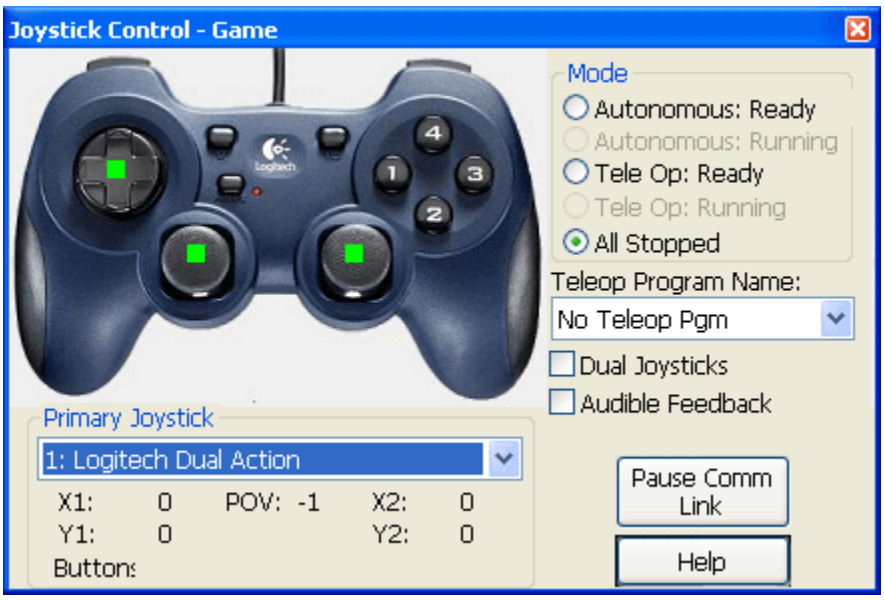


After setting the Menu Level to "Expert", you can then access the "Joystick Control - Game" station by:

- Downloading your program and starting the debugger
- Going to the "Robot" menu
- Choose the "Debug Windows" sub-menu
- Click on the "Joystick Control - Simple" menu option to open the Controller Station




As you can see, the window has a few more options than the "Joystick Control - Simple" screen did.



If you would like to use two controllers, click the "Dual Joysticks" button to expand the Joystick Control window to facilitate two controllers. You can assign the same controller to both Primary and Secondary Joysticks, but this is not recommended.


# 8.10 Math

ROBOTC has a powerful collection of useful math functions for the NXT platform. The RCX platform does not have enough memory to store these more advanced math functions.

---

**sin(fRadians)**
**cos(fRadians)**
Returns the sine or cosine of the input parameter. The input value is a float value specified in radians.
*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Trig Demo.c")*

```
float fSine = sin(PI/4);      // Variable 'fSine' is set to the sin of PI / 4 radians.
float fCosine = cos(PI/4);    // Variable 'fCosine' is set to the cos of PI / 4 radians.
```

**asin(Sine)**
Returns the arc-sine, i.e. the angle (in radians) whose sine value is “Sine”.
*Example:*

```
float fArcSine = asin(PI/4);      // Variable 'fArcSine' is set to the arc-sin of PI / 4.
```

**acos(Cos)**
Returns the arc-cosine, i.e. the angle (in radians) whose cosine value is “Cos”.
*Example:*

```
float fArcCos = acos(PI/4);      // Variable 'fArcCos' is set to the arc-cosine of PI / 4.
```

**atan(Tangent)**

Returns the arc-tangent, i.e. the angle (in radians) whose tangent value is “Tangent”.
*Example:*

```
float fArcTan = atan(PI/4);      // Variable 'fArcTan' is set to the arc-tan of PI / 4.
```

**PI**
Constant float variable containing the value of PI.
*Example:*

```
nxtDisplayTextLine(3, "%1.14f", PI);    // Display PI to 14 digits on line 3 of the NXT LC
```

**pow(number1, number2)**
Returns number1 to the power of number2.
*Example:*

```
pow(2, 8);       // Returns '256'.  (2 ^ 8 = 256)
```

**sinDegrees(degrees)**
**cosDegrees(degrees)**
Returns the sine or cosine of the input parameter. The input value is float value specified in degrees.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Draw Spiral.c")*

```
float xSin2 = sinDegrees(45.0);    // Variable 'xSin2' is set to the sin of 45.0 degrees.
float xCos2 = cosDegrees(45.0);    // Variable 'xCos2' is set to the cos of 45.0 degrees.
```

**radiansToDegrees(radians)**
**degreesToRadians(degrees)**
Converts between degrees and radians. The converted values are normalized – 0 to 359 for degrees and 0 to 2 * PI for radians.
*Example:*

```
float fDegrees = radiansToDegrees(PI/4);    // Variable 'fDegrees' is set to the degree va
float fRadians = degreesToRadians(45.0);    // Variable 'fRadians' is set to the radian va
```

**abs(input)**
**exp(input)**
**sgn(input)**
**sqrt(input)**
Performs the appropriate math function on the input variable.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Test Math.c")*

```
int nAbsNum = -4246;               // 'nAbsNum' is set to -4246, a number we will use for t
int nAbsResult = abs(nAbsNum);     // 'nAbsResult' is set to the absolute value of 'nAbsNum
```

```
float lExpNum = 8.3538;           // 'lExpNum' is set to 8.3538, an interesting number whi
                                  // the "exp()" function.
float lExpResult = exp(lExpNum);  // 'lExpResult' is set to (e ^ 'lExpNum').


int nSgnNum = -46;                // 'nSgnNum' is set to -46, a number we will use for the
int nSgnResult = sgn(nSgnNum);    // 'nSgnResult' is set to < 0 if 'nSgnNum' is negative a
                                  // is positive.


float fSqrtNum = 1802.8516;         // 'fSqrtNum' is set to -46, a number we will use for
float fSqrtResult = sqrt(fSqrtNum); // 'fSqrtResult' is set to < 0 if 'nSgnNum' is negativ
                                    // 'nSgnNum' is positive.
```

**srand(seed)**
Sets the seed for the random number generator.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Random Test.c")*

```
seed(4246);   // The number passed to "seed()" can range from -32767 to 32767 and is used
              // pseudo-random number generator in the firmware.

random(5);    // Generates a random number from 0 to 5. (6 sided dice!)
```

**random(range)**
Returns an integer value in the range 0 to 'range' where 'range' should be a positive integer in the range 0..32767.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Random Test.c")*

```
seed(4246);   // The number passed to "seed()" can range from -32767 to 32767 and is used
              // pseudo-random number generator in the firmware.

random(5);    // Generates a random number from 0 to 5. (6 sided dice!)
```

# 8.11 Messaging

**sendMessage(nMessageID)**
Sends a single 16-bit word message. 'nMessageID' should range in value form -32767 to +32767. Message value 0 is invalid and should not be used. It is a special value to indicate "no message" received when using the "message" variable. The message is sent over BT. Do not send messages faster than about 1 message per 30 milliseconds or it is possible for some messages to be lost.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\NXT BT RCX.c")*

```
long nMessageID = 4246;       // Create 'nMessageID' to hold the 16-bit word message.
sendMessage(nMessageID);      // Sends 'nMessageID' over an existing Bluetooth Connection.
```

**sendMessageWithParm(nMessageID, nParm1, nParm2);**

Identical in function to the "sendMessage" opcode except that the message is three 16-bit values. This is useful in easily sending separate items of information in a single message. Do not use a value of zero for 'nMessageID'.

*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BtBasicMsg.c")*

```
long nMessageID = 4246;                              // Create 'nMessageID' to hold a 16-bi
long nParm1 = messageParm[1];                        // Create 'nParm1' to hold a 16-bit wo
long nParm2 = messageParm[2];                        // Create 'nParm2' to hold a 16-bit wo
sendMessageWithParm(nMessageID, nParm1, nParm2);     // Sends messages over Bluetooth Conne
```

**message**

Contains the value (value -32767 to + 32767) of the last message received over the bluetooth channel. Many messages can be queued at the receiving NXT. See the sample programs for a technique to skip all but the last queued message. A value of '0' indicates that no message has been received of that the last received message has been processed by the user's program. The next time 'message' variable is accessed, it will get the next message, if available, from the queue of received messages stored by the NXT firmware. Call the "ClearMessage()" function to indicate that processing of the current message is complete.

*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BtBasicMsg.c")*

```
long lastMessage = message;    // 'lastMessage' is set equal to the last message received.
```

**messageParm[]**

Contains optional message parms (up to 3 16-bit words) for messages received over the RCX infrared channel. 'messageParm[0]' is the same as the variable 'message'.

*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BtBasicMsg.c")*

```
long lMessage = messageParm[0];    // 'lMessage' is set equal to the last message received
```

## 8.11.1 More Information

The NXT has a very powerful communication capability between two NXTs using the wireless Bluetooth functionality built into every NXT. The "Messaging" functions described here provide an easy approach for using this communication. It limits the communication to messages containing three 16-bit parameters.

It is very easy to set up Bluetooth communications between two NXTs.

- Use the NXT's on-brick user interface to make a connection between two NXTs.
    - It is also possible to set up the connections within a user's program but that is for advanced users.
- Use the sendMessage(nMessageID)and sendMessageWithParms(nMessageID, nParm1, nParm2) functions to send outgoing messages.
- Use the variables message and messageParms to retrieve the contents of a message. When you've finished processing a message, use the ClearMessage() function to "zero out" the message so that your program can process subsequent messages. The next time your program references the message variable, it will be for the next message that has arrived.

---

### Sending Messages:

There are two functions for easily sending messages. One sends a single 16-bit value and the other sends three 16-bit values. Either of the two NXTs can send messages at either time. It is the responsibility of the user program to not send messages too frequently as they may cause congestion on either the Bluetooth link or overflow of the NXT's transmit and receive queues.

**sendMessage(nMessageID)**

Sends a single 16-bit word message. 'nMessageID' should range in value form -32767 to +32767. Message value 0 is invalid and should not be used. It is a special value to indicate "no message" received when using the "message" variable.

**sendMessageWithParm(nMessageID, nParm1, nParm2)**
This function is identical to the sendMessage function except that the message contains three 16-bit values. This is useful in easily sending separate items of information in a single message. Do not use a value of zero for 'nMessageID'.

---

## Receiving Messages:

The NXT firmware automatically receives messages and adds them to a queue of incoming messages. The application program takes the messages from this queue and processes them one at a time. The variables message and messageParm contain the contents of the current message being processed. The function ClearMessage discards the current message and sets up to process the next message.

**Message**
This variable contains the 16-bit value of message received over the Bluetooth channel. It has a range of -32767 to 32767. A value of zero is special and indicates that there is “no message”. Whenever the value is zero and the message variable is accessed, the firmware will check to see if it has any received messages in its queue; if so, it will take the first message and transfer its contents to the message and messageParms variables. These two variables will continue to contain the message contents until the user’s program indicates it has finished processing the message by calling the ClearMessage() function.

**messageParm[]**
Array containing optional message parameters (up to 3 16-bit words) for messages received over the RCX infrared channel. messageParm[0] is the same as message messageParm[1] and messageParm[2] are additional 16-bit values.

**bQueuedMsgAvailable()**
Boolean function that indicates whether a unprocessed message is available in the NXT's received message queue. This is useful when multiple messages have been queued and your program wants to skip to the last message received. Your program can simply read and discard messages as long as a message is available in the queue.

**ClearMessage()**
Clears the current message. The next time the message variable is accessed, the firmware will attempt to obtain the first message from the queue of messages received by the NXT.
Do not send messages faster than about one message per 30 milliseconds or it is possible for some messages to be lost.

---

## Skipping Queued Messages:

A typical application might have one NXT send a message to the NXT on a periodic basis. For example, it might send a message containing the current values of sensors S1 and S2 every 100 milliseconds. Due to processing delays in the receiving NXT, several messages may have been queued and your program

may want to rapidly skip to the last message received. The following are two code snippets that show how this might be accomplished.

**The code in the NXT sending the sensor values:**

```
while (true)
{
  sendMessageWithParm(SensorValue[S1], SensorValue[S2], 0);
  wait1Msec(100);  // Don't send messages too frequently.
}
```

**The code in the receiving NXT:**

```
while (true)
{
  //
  // Skip to the last message received
  //
  while (bQueuedMsgAvailable())
  {
    word temp;
    ClearMessage();    // We're ready to process the next message
    temp = message;    // Obtain the next message
  }
  if (message == 0)
  {
   // No message is available to process
    wait1Msec(5);
    continue;
  }
  // A message is ready to be processed
  remoteSensor1 = message;          // the value of 'S1' from the remote NX
  remoteSensor2 = messageParm[1];   /* the value of 'S2' from the remote NX
.                    */
                                    /* user code to process the message. It
delays.    */
}
```

## Bluetooth Messaging is Incompatible with Debugging Over BT Link

NOTE: The ROBOTC IDE debugger can operate over either USB or Bluetooth connection to the NXT. When an application program uses the Bluetooth connection to send messages to another NXT then you cannot use the Bluetooth debugger connection. They are incompatible.

## Differences Between NXT Messaging and the RCX Infrared Messaging

This functionality is similar to that found on the LEGO Mindstorms RCX with a few notable exceptions:

1. The RCX uses an infrared communications link. The NXT uses a wireless Bluetooth link.
2. The RCX used broadcast messages that could be received by any RCX. The NXT uses "directed" messages that go to a single NXT.
3. Two RCXes sending messages simultaneously would corrupt both messages. The NXT's Bluetooth allows simultaneous message transmission.

4. There was no queuing of received messages on the RCX. When a new message arrives at the RCX it overwrites previously unhandled messages or it is discarded if the current message is not finished processing.

# 8.12 Miscellaneous

**memcpy(pToBuffer, pFromBuffer, nNumbOfBytes);**
Function copies characters from 'pFromBuffer' to 'pToBuffer'. 'nBytesToCopy' is the number of bytes to copy. Identical to the function found in conventional C 'string.h' library.
*Example: (program example can be found at: "\Sample Programs\NXT\Compiler Samples\StringIndex.c")*

```
string sTemp = "ROBOTC";                    // Create a string, 'sTemp' to be "ROBOTC".
char sArray[20];                            // Create a char array, 'sArray' of size 20.
memcpy(sArray, sTemp, sizeof(sArray));      // Copy characters from 'sArray' to 'sTemp'.
```

**memset(pToBuffer, nValue, nNumbOfBytes);**
Sets a block of memory at 'pBuffer' to the value 'nValue'. 'nBytesToSet' is the number of bytes to set. This is a useful function for initializing the value of an array to all zeros. Identical to the function found in conventional C 'string.h' library.
*Example: (program example can be found at: "\Sample Programs\NXT\Bluetooth\BT LowLevel.c")*

```
int kSendSize = 1;                                  // We will be sending 1 byte.
ubyte BytesToSend[kSendSize];                       // Create a ubyte array of size 1.
short nMsgXmit = 0;                                 // We will be setting them to 0.
memset(BytesToSend, nMsgXmit, sizeof(BytesToSend));    // Set the Byte Array to 0.
```

**version**
Contains the firmware version number.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Draw Spiral.c")*

```
nxtDisplayCenteredBigTextLine(3, "%d", version);    // Display the current firmware virsio
```

# 8.13 Motors

There are three motors on the NXT labeled A, B and C. Each motor is equipped with an integrated encoder that can be used to determine the current position of the motor. The encoders have 360 counts per single revolution of the motor. A count of 180 indicates that the motor has traveled half a rotation.

**bFloatDuringInactiveMotorPWM**
Boolean variable. True 'coasts' ('floats') the motor when power is not applied. False brakes the motor. False is best choice.
*Example: (program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Motor Speed.c")*

```
bFloatDuringInactiveMotorPWM = true;     // The motors WILL coast when power is not applie
bFloatDuringInactiveMotorPWM = false;    // The motors WILL NOT coast when power is not ap
```

**bMotorReflected[]**

Boolean array. Used to indicate that the direction of a motor should be reflected 180 degrees. Useful when mechanical design results in a logical "reversed" condition of a motor. This can also be configured in the Motors and Sensor Setup menu in ROBOTC.

*Example: (program example can be found at: "\Sample Programs\NXT\Robot Sample Code\BalancingRbt2.c")*

```
bMotorReflected[motorA] = true;     // Motor A WILL be reflected 180 degrees.
bMotorReflected[motorA] = false;    // Motor A WILL NOT be reflected 180 degrees.
```

**motor[]**

Contains the motor power or speed level (-100 to +100). Negative values are reverse; positive formward. A power level of 0 (zero) stops the motors.

*Example: (program example can be found at: "\Sample Programs\NXT\Motor\Moving Forward.c")*

```
motor[motorA] = 100;     // Motor A is given a power level of 100 (forward).
motor[motorA] = -100;    // Motor A is given a power level of -100 (reverse).
```

**nMaxRegulatedSpeedNXT**

Specifies the maximum speed, in encoder counts per second, that should be used for speed regulation on the NXT. (Default is 1000)

*Example:*

```
nMaxRegulatedSpeedNXT = 500;     // The PID maximum speed for the NXT motors is scaled dow
```

**nMaxRegulatedSpeed12V**

Specifies the maximum speed, in encoder counts per second, that should be used for speed regulation on the TETRIX motors. (Default is 1000)

*Example:*

```
nMaxRegulatedSpeed12V = 500;     // The PID maximum speed for the Tetrix motors is scaled
```

**nMotorEncoder[]**

Current value of the motor encoder. Range is -32768 to 32767 so it will "wrap" after about ~90 revolutions. The user's program should reset the value of the encoder often to avoid the value "resetting" itself when the maximum distance is met.

*Example: (program example can be found at: "\Sample Programs\NXT\Encoder\Moving by Rotation.c")*

```
nMotorEncoder[motorB] = 0;          // Reset the Motor Encoder of Motor B.
while(nMotorEncoder[motorB] < 360)  // While the Motor Encoder of Motor B has not yet reac
{
  motor[motorB] = 75;               // Motor B is given a power level of 75.
  motor[motorC] = 75;               // Motor C is given a power level of 75.
}
motor[motorB] = 0;    // Motor B is given a power level of 0 (stop).
motor[motorC] = 0;    // Motor C is given a power level of 0 (stop).
```

**nMotorEncoderTarget[]**

The nMotorEncoderTarget is used to set a target distance that a motor should move before system puts motor back in idle or stopped state. A target value of 0 (zero) means run forever. The first example only looks at one motor's encoder, keeping things simple in order to travel a specific distance. The second example monitors both motor's encoders to ensure that they both reach their target before moving on to the next segment of code.

*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Motor Examples.c")*

```
nMotorEncoder[motorB] = 0;                  // Reset the Motor Encoder of Motor B.
nMotorEncoderTarget[motorB] = 360;          // Set the  target for Motor Encoder of Motor B
motor[motorB] = 75;                         // Motor B is run at a power level of 75.
```

```
motor[motorC] = 75;                          // Motor C is run at a power level of 75.

while(nMotorRunState[motorB] != runStateIdle)  // While Motor B is still running (hasn't r
{
  // Do not continue.
}
motor[motorB] = 0;                          // Motor B is stopped at a power level of 0.
motor[motorC] = 0;                          // Motor C is stopped at a power level of 0.
```

```
nMotorEncoder[motorB] = 0;                  // Reset the Motor Encoder of Motor B.
nMotorEncoder[motorC] = 0;                  // Reset the Motor Encoder of Motor C.
nMotorEncoderTarget[motorB] = 360;          // Set the  target for Motor Encoder of Motor B
nMotorEncoderTarget[motorC] = 360;          // Set the  target for Motor Encoder of Motor C
motor[motorB] = 75;                         // Motor B is run at a power level of 75.
motor[motorC] = 75;                         // Motor C is run at a power level of 75.

while(nMotorRunState[motorB] != runStateIdle && nMotorRunState[motorC] != runStateIdle)
// While Motor B AND Motor C are still running (haven't yet reached their target):
{
  // Do not continue.
}
motor[motorB] = 0;                          // Motor B is stopped at a power level of 0.
motor[motorC] = 0;                          // Motor C is stopped at a power level of 0.
```

**nMotorPIDSpeedCtrl[]**

nMotorPIDSpeedCtrl is used to enable or disable the speed regulation that is used to ensure that motors travel at a consistent speed, regardless of the surface or resistance met. Speed Regulation can be set in the Motors and Sensor Setup menu in ROBOTC. To make things easier, by default PID is enabled for LEGO motors.

*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Lego Cuckoo Clock.c")*

```
nMotorPIDSpeedCtrl[motorA] = mtrSpeedReg;          // Enable PID on Motor A.
```

**nMotorRunState[]**

Array containing the internal state of a NXT motor. Useful in checking when a motor movement "command" has finished. There are three different states - runStateRunning, runStateHoldPosition, runStateIdle.

*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Motor Examples.c")*

```
nMotorEncoder[motorB] = 0;                  // Reset the Motor Encoder of Motor B.
nMotorEncoderTarget[motorB] = 360;          // Set the  target for Motor Encoder of Motor B
motor[motorB] = 75;                         // Motor B is run at a power level of 75.
motor[motorC] = 75;                         // Motor C is run at a power level of 75.

while(nMotorRunState[motorB] != runStateIdle)  // While Motor B is still running:
{
  // Do not continue.
}
motor[motorB] = 0;                          // Motor B is stopped at a power level of 0.
motor[motorC] = 0;                          // Motor C is stopped at a power level of 0.
```

**nPidUpdateInterval**

Interval (in milliseconds) to use for motor PID updates. Default is 25. Updating the PID algorithm too often could cause the PID algorithm to be very dynamic and cause a jerky motion. A value too high will not update fast enough and will cause the motors to seem as if they are not regulated.

*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Motor Examples.c")*

```
nPidUpdateInterval = 20;    // A good interval is 20.
```

**nSyncedMotors**
Used to specify synchronization of two NXT motors so they operate in 'lock step' at the same speed turn ratio. The command to sync motors is "synchXY", where X is the "master" motor letter, and Y is the "slave" motor letter.
*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Motor Synchronization.c")*

```
nSyncedMotors = synchBC;    // Synch motors B and C so that B is the master and C is the s
nSyncedTurnRatio = 100;     // The slave motor will recieve only half of what the master m
```

**nSyncedTurnRatio**
Turn ratio to use for a motor that is 'slaved' to another motor. +100 is same as primary motor. Zero is stopped. -100 is same movement as primary but in inverted directon. Values between 100 and 0 give porportional power levels from the primary motor to the slave motor.
*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Motor Synchronization.c")*

```
nSyncedMotors = synchBC;    // Synch motors B and C so that B is the master and C is the s
nSyncedTurnRatio = 100;     // The slave motor will recieve only half of what the master m
```

# 8.13.1 Motors Examples

## Motor Speed / Power Control

Motors are controlled by specifying a power level to apply to the motor. Power levels range from -100 to +100. Negative values indicate reverse direction and positive values indicate forward direction. To move motor 'A' forward at 50% of full power, you would use the following statement:

```
motor[motorA] = 50;
```

**motor[]** is the ROBOTC array variable that controls the motor power and **motorA** is an enum variable for motor A.

The default motor operating mode is an "open loop" configuration where the above statement would apply 50% 'raw' power to the motor. Open loop control does not result in a consistent speed across all motors; e.g. the speed will become lower as the batteries discharge and different motors will produce different speeds because of minor variations in the motors.

A "closed loop" control algorithm uses feedback from the motor encoder to adjust the raw power to provide consistent speed. Closed loop control continuously adjusts or regulates the motor raw power to maintain (for the above example) a motor speed that is 50% of the maximum regulated speed. Regulated motor control is enabled and disabled with the array variable **nMotorPIDSpeedCtrl[]**.

```
nMotorPIDSpeedCtrl[motorA] = mtrNoReg;     // disables motor speed regulation
nMotorPIDSpeedCtrl[motorA] = mtrSpeedReg;  // enables motor speed regulation
```

With brand new alkaline batteries a motor can move at almost 1,000 encoder counts per second. But with a partially discharged motor and a motor that has had some wear and tear the maximum speed may reduce to 750 counts. This has an impact on speed regulation.

With fully charged batteries, the speed regulation software might result in an average to 75% power to achieve a regulated speed of 750 counts per second. But with partially charged batteries, it might be applying power almost all the time!

For speed regulation to function properly there needs to be “room” for the closed loop control firmware to adjust the raw power. If the batteries are partially discharged, you shouldn’t try to achieve a regulated speed above 750 counts per second.

The default maximum speed level is 1,000 counts per second. This value was chosen because it is the same value used by the standard NXT-G firmware. But, as explained above, with partially discharged batteries, perhaps only 750 counts can be achieved. This means that if you want to achieve consistent speeds across battery level you must either not specify speeds above the 75% level or reduce the maximum speed level. The maximum speed can be set with the following variable:

```
nMaxRegulatedSpeed = 750;     // max regulated speed level in degrees per second
```

## Encoder Value

**nMotorEncoder[]** is an array that provides access to the current value of the motor encoder. To wait until the motor moves to a specific encoder position, you might use the following:

```
while (nMotorEncoder[motorA] < 1000) // wait for motor to reach a specific location
{
  motor[motorA] = 50;  /* Run the motors forward at a power level of 50
                          until 1000 encoder counts have been reached.  */
}
```

**nMotorEncoder[]** is a 16-bit variable. The maximum value before it overflows is 32,767 which allows for about 95 motor rotations. In long running applications, you may want to periodically reset the value in your application program to avoid overflow situations.

## Moving to a Specific Encoder Value

You can move a motor a specific amount with the variable **nMotorEncoderTarget[]**. This indicates the number of encoder ticks to move. If you specify a positive value the motor will slow to a stop at this position. A negative value will leave the motor in coast / float mode when the encoder position is reached. **nMotorEncoderTarget[]** does not turn on the motor. It only sets the target “stop” position. You have to start the motor moving with an assignment to **motor[]** variable.

```
nMotorEncoder[motorB] = 0;                  // Reset the Motor Encoder of Motor B.
nMotorEncoderTarget[motorB] = 360;          // Set the  target for Motor Encoder of Motor B
motor[motorB] = 75;                         // Motor B is run at a power level of 75.
motor[motorC] = 75;                         // Motor C is run at a power level of 75.

while(nMotorRunState[motorB] != runStateIdle)  // While Motor B is still running:
{
  // Do not continue.
}
motor[motorB] = 0;                         // Motor B is stopped at a power level of 0.
motor[motorC] = 0;
```

## Synchronizing Two Motors

A very common application is two motors used to power the 'left' and 'right' motors on a robot. The robot will travel a straight line if both motors are rotating an exactly the same speed. The variable **nSyncedMotors** is used to synchronize two motors so that the second motor's speed is slaved to the first motor.

```
nSyncedMotors = synchNone;  // No motor synchronization
nSyncedMotors = synchBC;    // Motor 'C' is slaved to motor 'A'
```

The synchronization "mode" is specified with the variable **nSyncedTurnRatio**. This ranges in value from -100% to +100%. The magnitude (0 to 100) represents the ratio of the slave speed to the primary motor speed. The sign indicates same (positive) or opposite (negative) direction of travel for the slave vs. the primary. A value of less than 100 will cause the robot to turn.

The following example will drive robot in a straight line and then turn the robot:

```
nSyncedMotors = synchBC;              // Motor 'C' is slaved to motor 'B'
// Drive in a straight line
nSyncedTurnRatio = +100;              // Move in a straight line
nMotorEncoder[motorB] = 0;
nMotorEncoderTarget[motorB] = -1000;  // move 1000 encoder counts and stop
motor[motorB] = 100;

while (nMotorEncoder[motorB] < 1000)  // wait until movement is complete
{}

// Rotate in place to turn robot
nSyncedTurnRatio = -100;              // Rotate in place
nMotorEncoderTarget[motorB] = 200;    // move 200 encoder counts and stop
motor[motorB] = 50;
wait1Msec(3000);
```

A value of 100 indicates that the slave will travel in the same direction and at exactly the same speed as the primary motor. Sync ratio -100 indicates that slave will travel at same speed but in opposite direction.

## Brake vs. Coast/Float Mode

Motor speed is controlled via a technique called pulse width modulation (PWM). PWM applies "pulses" of power to the motor thousands of times a second. Each pulse is a square wave containing a period of on or powered time followed by a period of off time. The ratio of on-time to off-time adjusts the power applied to the motor. Full battery voltage is applied during the on-time; this is a better technique for speed adjustment than systems that applies partial battery voltage to regulate the speed.
During the "off" time, power is not applied to the motors. During off-time the motors can be either "open-circuited" ore "short-circuited". Open circuit results in a "floating" voltage where the motor is coasted. Short circuit results in a braking action. ROBOTC has a variable that controls which mode is used.

```
bFloatDuringInactiveMotorPWM = false;  // brake
bFloatDuringInactiveMotorPWM = true;   // coast or float
```

Brake mode works best for the NXT motors. This is the default mode. You can change the default mode within the ROBOTC "preferences" setup when under the "Expert" menu level. The ROBOTC compiler will automatically generate code at the start of your program to setup the brake or float mode.

## Reflected or Mirrored Motors

Forward and reverse direction for a motor can be somewhat arbitrary. You may build a Lego model where the forward / reverse direction of the motor is the opposite of what seems logical. This can be easily handled in ROBOTC with the **bMotorReflected[]** variable which flips or mirrors the motor direction

```
bMotorReflected[motorA] = true;    // flips/reflects the logical motor direction
```

## 8.13.2 Servo Motors

**ServoValue[servo#]**
This read-only function is used to read the current position of the servos on a sensor port Servo controller. Values can range from 0 to 255. The value returned in this variable is the last position that the firmware has told the servo to move to. This may not be the actual position because the servo may not have finished the movement or the mechanical design may block the servo from fully reaching this position. To set the position of a servo, use the "servoTarget" or "servo" functions.
*Example:* (program example can be found at: "\Sample Programs\NXT\Servos\Servo Sweep TETRIX.c")

```
int a = ServoValue [servo1];    // Assigns the value of servo1 to integer variable 'a'.
```

**servo[servo#] = position** or **servoTarget[servo#] = position;**
This function is used to set the position of the servos on a sensor port Servo controller. Values can range from 0 to 255. The firmware will automatically move the servo to this poisition over the next few update intervals. (Be sure to give the servo some amount of time to reach the new position before going on in your code.)
*Example:* (program example can be found at: " \Sample Programs\NXT\Servos\Servo Sweep TETRIX.c ")

```
servo[servo1] = 160;            // Changes the position of servo1 to 160.
servoTarget[servo1] = 160;      // Changes the position of servo1 to 160.
```

**servoChangeRate[servo#] = changeRate;**
Specifies the rate at which an individual servo value is changed. A value of zero inidcates servo will move at maximum speed. The change rate is a useful variable for "smoothing" the movement of the servos and preventing jerky motion from software calculated rapid and wide changes in the servo value. The default value is a change rate of 10 positions on every servo update which occurs. (updates occur every 20 milliseconds)
*Example:* (program example can be found at: " \Sample Programs\NXT\Servos\Servo Sweep TETRIX.c ")

```
servoChangeRate[servo1] = 2;    // Slows the Change Rate of servo1 to 2 positions per upda
```

# 8.14 Preprocessor Defines

Preprocessor Defines:

The ROBOTC Compiler supports several different preprocessor defines that contain information about the compile time environment. Recently added are defines that indicate which features are enabled in the IDE.

The predefined symbols can be used in include files ad user programs for conditionally compiling code based on features. The preprocessor defines include:

| Symbol Name | Usage / Comments |
|---|---|
| ROBOTC | Indicates that the ROBOTC compiler was used. |
| _DEBUG, _RELEASE, or _CUSTOM | Compiler optimization selection. |
| _TARGET | Contains the type of debugger target. One of the following values: “Robot”, “Emulator” or “VirtWorld”. |
| NXT, TETRIX, VEX, VEX20, ARDUINO | The currently selected platform. |
| firmwareVersion | Contains the numerical firmware version; e.g. 852. |
| IFI | When IFI platform is selected |
| TETRIX | When TETRIX features are active |
| FTC | When FTC features are active |
| Algebra | When robot algebra option is enabled |
| MultiRobotSupport | When multi-robot option is enabled. |

---

ROBOTC also supports the standard C preprocessor symbols below:

| Symbol Name | Usage / Comments |
|---|---|
| __FILE__ | Current filename |
| __LINE__ | Current line number |
| __DATE__ | Date “mmm dd yyyy” of the current compilation |
| __TIME__ | Time “hh:mm:ss” of the current compilation |

# 8.15 Sensors

The NXT is equipped with four sensor ports - S1, S2, S3 and S4. There are a variety of functions and variables used for configuring these ports and accessing their values. Configuring sensors can be

complicated, to alleviate this ROBOTC has a built-in tool that can be used to configure the NXT sensors. Inside of this tool, you can do the following:

- The variable name that you want to assign to the sensor. There are a number of “built-in” variable arrays that contain information about the sensors. The array index for these variable can be the sensor port values (i.e. S1, S2, S3 and S4) or even the numerical array index (i.e. 0 to 3). However, it is a good programming practice to assign a meaningful name (e.g. lightSensor, leftBump, rightBump, sonar, etc) to a sensor rather than one of these numerical indices.
- The type of sensor that is involved. The NXT kit comes with several LEGO developed sensors – touch, sound, light, sonar – and there are many more 3rd party sensor types in development. The firmware must know the type of sensor so that it can properly configure the sensor – e.g. some sensors require optional 9V battery power and some do not – and know how to normalize the values received from the sensor.
- The mode of the sensor. The most common mode is ‘percentage’ which will normalize the sensor value to a reading in the range 0 to 100. The next most common mode is ‘raw’ which simply returns the un-normalized value from the sensor. Other modes include “Celsius” and “Fahrenheit” which are used to specify the scale for RCX temperature sensors.

---

**SensorRaw[]**
This array value will return the "raw" (un-normalized) value of a sensor. Usually this is the raw A-D converted value, which is an analog value between 0 to 1023.
*Example: (program example can be found at: "\Sample Programs\NXT\Try Me Program Source\SONAR Enhanced.c")*

```
if(SensorRaw[Light2] > 512)     // If the Raw Value of the Light Sensor is greater than 512
{
  nxtDisplayCenteredTextLine(3, "Light Sensor > 512");     // Display Text to the LCD Scree
}
```

**SensorType[]**
The SensorType array is used to specify what type of sensor is connected to a certain port. Most users should not have to use this functionality and should use the Motors and Sensor Setup instead.
*Example: (program example can be found at: "\Sample Programs\NXT\Try Me Program Source\SONAR Enhanced.c")*

```
SensorType[sonarSensor] = sensorSonar;     // Set 'SonarSensor' to be of type sensorSonar (
                                           // Sensors Setup screen does this for you in the
```

**SensorValue[]**
This array value returns the value of the sensor in a normalized fashion. Rather than returning a raw value of 0 to 1023, ROBOTC will interpret the data from the "SensorType" and return a more accurate representation of the sensor's data. An example of this is the Light Sensor, which will return a percentage value from 0 to 100.
*Example: (program example can be found at: "\Sample Programs\NXT\SONAR\Smooth Sonar Tracking.c")*

```
while(true)     // Infinite loop:
{
  nxtDisplayCenteredTextLine(3, "Sensor Value: %d", SensorValue[sonarSensor]);     // Displ
  wait1Msec(100);     // Wait 100 milliseconds to help display correctly.
}
```

## 8.15.1 Sensors Examples

## Touch Sensor

The NXT Touch Sensor has 2 states, pressed or unpressed. In code this is represented with a 1 for pressed, and a 0 for unpressed. There are no other states for this sensor. Below is a simple program that keeps your program from running until you press the Touch Sensor.

*(This program can be found at, "\Sample Programs\NXT\Touch\Wait for Push.c")*

```
#pragma config(Sensor, S1,     touchSensor,         sensorTouch)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

task main()
{
  while(SensorValue(touchSensor) == 0)   // While the Touch Sensor is inactive (hasn't been pressed):
  {
    // DO NOTHING (wait for press)
  }
  while(SensorValue(touchSensor) == 1)   // While the Touch Sensor is active (pressed):
  {
    // DO NOTHING (wait for release)
  }

  // YOUR CODE GOES HERE
                              // Otherwise (the touch sensor has been activated [pressed] ):
  motor[motorB] = 75;                 /* Run motors B and C forwards */
  motor[motorC] = 75;                 /* with a power level of 75.   */
  wait1Msec(1000);                   // Wait 1000 milliseconds (1 second) before moving to further code.
}
```

---

## Light Sensor

The NXT Light Sensor has a range of states between 0 and 100. The lower the number, the darker the reading is. The higher the number, the lighter the reading is. Below is a simple line-following program that uses only one NXT Light Sensor.

*(This program can be found at, "\Sample Programs\NXT\Light\Line Tracking.c")*

```
#pragma config(Sensor, S3,     lightSensor,         sensorLightActive)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

task main()
{
   wait1Msec(50);                          // The program waits 50 milliseconds to initializ
  while(true)                  // Infinite loop
  {
    if(SensorValue(lightSensor) < 45)  // If the Light Sensor reads a value less than 45:
    {
      motor[motorB] = 60;              // Motor B is run at a 60 power level.
      motor[motorC] = 20;              // Motor C is run at a 20 power level.
    }
    else                       // If the Light Sensor reads a value greater than or equal to 45:
```

```
    {
      motor[motorB] = 20;              // Motor B is run at a 20 power level.
      motor[motorC] = 60;              // Motor C is run at a 60 power level.
    }
  }
}
```

## Sound Sensor

The NXT Sound Sensor has a range of states between 0 and 100. The lower the number, the quieter the reading is. The higher the number, the louder the reading is. Below is a simple program that maps the Sound Sensor reading to the motor speeds.

*(This program can be found at, "\Sample Programs\NXT\Sound\Sound Drive.c")*

```
#pragma config(Sensor, S2,     soundSensor,          sensorSoundDB)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

task main()
{
  wait1Msec(1000);        // A one-second wait is required to cleanly initialize the Sound
  while(true)             // Infinite loop
  {
    motor[motorB] = SensorValue[soundSensor];     /* Motors B and C are run at a power leve
    motor[motorC] = SensorValue[soundSensor];     /* to the value read in by the Sound Sens
  }
}
```

## Ultrasonic (sonar) Sensor

The NXT Ultrasonic Sensor has a range of states between 0 and 255. This number is representative of the current reading in centimeters. However, a reading of 255 means that the current sensor reading is out of range. This is a "range error" and means that the echo is not being read back (looking down a long hall for example). A reading outside of these numbers indicates that there is no sensor attached or some error with the connection.

*(This program can be found at, "\Sample Programs\NXT\Sonar\Detecting Obstacles with Sonar.c")*

```
#pragma config(Sensor, S4,    sonarSensor,        sensorSONAR)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

task main()
{
  int distance_in_cm = 20;   // Create variable 'distance_in_cm' and initialize it to 20(cm).
  while(SensorValue[sonarSensor] > distance_in_cm)   /* While the Sonar Sensor readings are greater */
  }                                  /* than the specified, 'distance_in_cm':       */
    motor[motorB] = 75;                 // Motor B is run at a 75 power level
    motor[motorC] = 75;                 // Motor C is run at a 75 power level
  }
  motor[motorB] = 75;    // Motor B is stopped at a 0 power level
```

```
 motor[motorC] = 75;    // Motor C is stopped at a 0 power level
}
```

# 8.16 Sensors Digital

User programming of NXT digital sensors is for advanced users. The average user will not require this capability.

Sensors on the NXT can be either analog or digital sensors.

- For analog sensors, one of the sensor input wires contains an analog value representing the value of the sensor.
- Digital sensors are more intelligent and typically contain a microprocessor. The sensor communicates with the NXT using the I2C serial communications protocol.

Each of the four sensor ports on the NXT can be configured as either a digital or analog sensor. There are integrated device drivers for the LEGO developed sensors built into the ROBOTC firmware. If an application program wants to provide its own device drivers for either existing or new 3rd party sensors, it is important to set the sensor type to a “custom user provided device driver” so that there are no conflicts between the built-in device drivers and the user written code. There are four sensor types that are valid for application controlled digital sensors on a NXT. These are:

- **sensorI2CCustomStd**
  - Indicates that user program sensor using standard baud rate communications. Standard communications operates at about 1 byte transferred per millisecond.
- **sensorI2CCustomStd9V**
  - Same as above with the addition that battery voltage (i.e. 9V less the drop through a protection diode) is also provided on pin X for those sensors that want the higher voltage level.
- **sensorI2CCustomFast**
  - Indicates that user program sensor using enhanced baud rate communications. The enhanced baud rate is about five times faster than the standard baud rate. So far, all new 3rd party sensors that have been tested are compatible with the enhanced baud rate; in testing only the LEGO Sonar sensor requires use of the standard baud rate.
- **sensorI2CCustomFast9V**
  - Same as above with the addition that battery voltage (i.e. 9V less the drop through a protection diode) is also provided on pin X for those sensors that want the higher voltage level.

There are several sample programs in the ROBOTC distribution that illustrate the use of I2C messaging on the NXT with ROBOTC. There are a few things to remember in I2C messaging:

- The reply bytes from an I2C “read” message are stored in a circular buffer until read by application program.
- Before sending an I2C “read” message, it is good practice to clear out any reply bytes that exist in the buffer.
- I2C messaging can be slow. There are several bytes of protocol overhead on every message.
- In implementing a device driver, it is better to start up a task that continually sends I2C messages to periodically poll the sensor and stores the results. Within the user application it should simply retrieve the last polled value – i.e. no delay – rather than do an inline I2C read to get the sensor value. ROBOTC makes this very easy to achieve by allowing user programs to write as well as read the internal “SensorValue” array. The "SensorValue” array is used to retrieve the value of a sensor from the internal device drivers.

- You program should accommodate and recover from the occasional I2C messaging error. Lots of testing has indicated that most sensors experience errors rates of 10^-3 to 10^-5.

---

**readI2CReply(nPort, replyBytes, nBytesToRead);**
Retrieve the reply bytes from an I2C message
*Example: (program example can be found at: "\Sample Programs\NXT\3rd Party Samples\Mindsensors I2C Scanner.c")*

```
#define i2cScanPort  S1     // Port to Scan!
TI2CStatus nStatus;         // Status variable.
byte replyMsg[10];          // Reply Message byte Array of size 10.

nStatus = nI2CStatus[i2cScanPort];  // Check the status of the port in question (S1 in thi
if(nStatus != NO_ERR)     // If the status is not, 'NO_ERR':
{
  return false;
}

readI2CReply(i2cScanPort, replyMsg[0], 8);  // Retrieve 8 reply bytes from 'i2cScanPort' (
if(replyMsg[0] == 0x00)  // If the bytes are hexidecimal '0' (0x00):
{
  return false;
}
```

**sendI2CMsg(nPort, sendMsg, nReplySize);**
Send an I2C message on the specified sensor port.
*Example: (program example can be found at: "\Sample Programs\NXT\3rd Party Samples\Mindsensors I2C Scanner.c")*

```
sendI2CMsg(i2cScanPort, i2cScanDeviceMsg[0], 8);  // Send an 8 byte message from
                                                  // 'i2cScanDeviceMsg[0]' to 'i2cScanPort
```

**nI2CBytesReady[]**
Number of queued bytes ready for access from previous I2C read requests.
*Example: (program example can be found at: "\Sample Programs\NXT\3rd Party Samples\Mindsensors I2C Scanner.c")*

```
nI2CBytesReady[i2cScanPort] = 0;     // Queue no bytes.
```

**nI2CRetries**
The number of re-transmission attempts that should be made for I2C message.
*Example: (program example can be found at: "\Sample Programs\NXT\3rd Party Samples\Mindsensors I2C Scanner.c")*

```
nI2CRetries = 0;     // Do or do not, there is no try.
```

**nI2CStatus[]**
Currents status of an sensor I2C link.
*Example: (program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Ultrasonic User Device Driver Fast.c")*

```
#define i2cScanPort  S1     // Port to Scan!
```

```
TI2CStatus nStatus;         // Status variable.
byte replyMsg[10];          // Reply Message byte Array of size 10.

nStatus = nI2CStatus[i2cScanPort];  // Check the status of the port in question (S1 in thi
if(nStatus != NO_ERR)     // If the status is not, 'NO_ERR':
{
  return false;
}

readI2CReply(i2cScanPort, replyMsg[0], 8);  // Retrieve 8 reply bytes from 'i2cScanPort' (
if(replyMsg[0] == 0x00)  // If the bytes are hexidecimal '0' (0x00):
{
  return false;
}
```

# 8.17 Sounds

ROBOTC provides a comprehensive suite of functions for controlling the NXT speaker.

ROBOTC firmware queues up to 10 sound items for playback. This allows user programs to initiate playback of a sound item and continue execution without having to wait for a sound item to finish; this mode of operatio nis very desirable for robots where, if you waited for the sound to complete playback, you could be delayed in updating the robot's motors in reaction to a change in sensors.

ROBOTC plays both uncompressed and compressed sound files. The ROBOTC development environment provides a command to compress sound files. It is located in the menu "Windows->NXT Brick->File Management".

---

```
ClearSounds();
```
Clears all existing and buffered sound commands
*Example:* *(program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Touch.c")*

```
ClearSounds();    // Clear the sound buffer.
```

```
MuteSound();
```
Mutes all subsequent sound commands.
*Example:*

```
MuteSound();    // MUTE
```

```
PlayImmediateTone(frequency, durationIn10MsecTicks);
```
Immediately play tone at frequency & duration ahead of queued requests.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Light.c")*

```
while(true)
{
  PlayImmediateTone(SensorValue[lightSensor] * 80, 50);    // Play tone according to light
  wait1Msec(200);                                          // Wait 200 milliseconds before
}
```

**PlaySound(sound);**
Play one of the system predefined sounds (buzz, beep, click, ...)
*Example:* *(program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Light.c")*

```
PlaySound(soundBeepBeep);    // Play the sound, 'soundBeepBeep'.
```

**PlaySoundFile(sFileName);**
Plays a sound file from the NXT file system. File must be present on the NXT. RobotC will automatically download required files with user program.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Touch.c")*

```
PlaySoundFile("Woops.rso");    // Play the sound file, 'Woops.rso'.
```

**PlayTone(frequency, durationIn10MsecTicks);**
Plays a constant tone at the specified frequency and duration.
*Example:* *(program example can be found at: "\Sample Programs\NXT\NXT Feature Samples\NXT Songs.c")*

```
PlayTone(784, 15);    // Play a tone at a frequency of 784 for 150 milliseconds.
```

**UnmuteSound();**
Restores sound playback volume.
*Example:*

```
UnmuteSound();    // UN-MUTE
```

**bPlaySounds**
Boolean flag. Indicates whether new sound requests should be accepted or discarded.
*Example:*

```
bPlaySounds = true;    // ACCEPT new sound requests.
bPlaySounds = false;   // DISCARD new sound requests.
```

**bSoundActive**
Boolean flag. If true indicates sound system is actively playing sound.
*Example:*

```
while(bSoundActive)    // While a sound is actively playing:
{
  // Do not continue until finished playing sound.
}
```

**bSoundQueueAvailable**
Boolean flag. Indicates whether there is space available in the sound queue for another item.
*Example:*

```
if(bSoundQueueAvailable)    // If there is still space in the Sound Queue:
{
  PlayImmediateTone(SensorValue[lightSensor] * 80, 50);    // Play tone according to light
}
else                        // If there is no longer any space in the Sound Queue:
```

```
{
  nxtDisplayCenteredTextLine(3, "Sound Queue Full!");      // Display a centered text on l
}
```

**nVolume**
Sound volume. Range 0 to 4 (loudest).
*Example:* *(program example can be found at: "\Sample Programs\NXT\Try Me Program Source\Light.c")*

```
nVolume = 1;    // Set the volume to 1, a nice and comfortable volume for a quiet location
```

# 8.18 Strings

When displaying floats, for example, you can tell ROBOTC how many decimals places to display. This is standard across all 'C' - like programing languages. For example, if your float is **PI (3.14159265)**, but you only want to dispay **"3.14"**, your string should contain, `" %1.2f "`.
The number ***before*** the decimal is how many digits ***before*** the decimal you wish to display, while the number ***after*** the decimal is how many digits ***after*** the decimal you wish to display. So `" %1.2f "` tells us to display one digit before the decimal and two digits after the decimal, with **"3.14"** as the final result.

---

**strcat(pToBuffer, pFromBuffer)**
Function concatenates 'pFromBuffer' onto end of 'pToBuffer'. The variables are arrays of bytes terminated with zero character. It is user responsibility to ensure that the 'To' array is large enough to hold the result. ROBOTC is not able to do any range checking! Identical to the function found in conventional C 'string.h' library.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Strings\String Demo.c")*

```
string str1 = "ROBOT";                          // String 'str1' is "ROBOT".
string str2 = "C";                              // String 'str2' is "C".

strcat(str1, str2);                             // Concatinate string 'str2' onto string 'str

nxtDisplayCenteredTextLine(3, "%s", str1);      // Display the new 'str' ("ROBOTC").
```

**strcmp(pString1, pString2)**
Function compares 'pString1' with 'pString2'. Returns negative value if less than, 0 if equal and positive value if greater than. The variables are arrays of bytes terminated with a zero char. Identical to the function found in conventional C 'string.h' library.
*Example:* *(program example can be found at: "\Sample Programs\NXT\Strings\String Demo.c")*

```
string str1 = "Fett";                         // String 'str1' is "Fett".
string str2 = "Fett";                         // String 'str2' is "Fett".

if(strcmp(str1, str2) < 0)                                  // If 'str1' < 'str2':
{
  nxtDisplayCenteredTextLine(3, "str1 is < than str2");      // Display that 'str1' is < '
}
else if(strcmp(str1, str2) > 0)                             // If 'str1' > 'str2':
{
  nxtDisplayCenteredTextLine(3, "str1 is > than str2");      // Display that 'str1' is > '
}
```

```
else                                                    // If 'str1' == 'str2':
{
  nxtDisplayCenteredTextLine(3, "str1 = str2");             // Display that 'str1' is = '
}
```

**strcpy(pToBuffer, pFromBuffer)**
Function copies 'pFromBuffer' to 'pToBuffer'. The variables are arrays of bytes terminated with a zero character. It is user responsibility to ensure that the 'To' array is large enough to hold the result. ROBOTC is not able to do any range checking! Identical to the function found in conventional C 'string.h' library.
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\String Demo.c")*

```
string str1 = "";                               // String 'str1' is "" (empty).
string str2 = "DNA";                            // String 'str2' is "DNA".

strcpy(str1, str2);                             // Copy string 'str2' onto string 'str1'.

nxtDisplayCenteredTextLine(3, "%s", str1);    // Display the new 'str' ("DNA").
```

**StringDelete(sDest, nIndex, nSize)**
Deletes a substring from a string
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\StringDelete Demo.c")*

```
string str = "mesGARBAGEsage";                  // String 'str' is "mesGARBAGEsage".

StringDelete(str, 3, 7);                        // Delete from index 3 to the 7th char after
                                                /* Confusing because 3 = index 3, not char 3.
                                                /* starts with 0, index 3 is the 4th characte

nxtDisplayCenteredTextLine(3, "%s", str1);    // Display the new 'str' ("message").
```

**StringFind(sSrce, sSearch)**
Finds the position in a string of the selected substring
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\StringFind Demo.c")*

```
string str = "mesHIDDENsage";                           // String 'str' is "mesGARBAGEsage
int index_of_substr;                                    // Int, 'index_of_substr' to be us

index_of_substr = StringFind(str_msg, "HIDDEN");        /* Search for the substring, "HIDD
                                                        /* withing the string, 'str'.

nxtDisplayCenteredTextLine(3, "%d", index_of_substr);   // Display the new index of the su
```

**StringFormat(sDest, sFmtSpec, nParm1, nParm2)**
Formats a string using the specified format string and up to two parameters for formating
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\StringFormat Demo.c")*

```
string str = "";                                // Create String, 'str' and initialize it as 'e
float num = 3.14159;                            // Create float, 'num' and initialize it as fir

StringFormat(str, "%1.2f", num);                /* Format num into a float to 2 decimal places,
                                                    the string, 'str'.
nxtDisplayCenteredTextLine(3, "%s", str);     // Display the new string, str. ("3.14").
```

**StringFromChars(sToString, FromChars)**
Converts an array of bytes to a string value. You MUST end your char array with a char value of zero!

**strncat(pToBuffer, pFromBuffer, nMaxBufferSize)**
Function concatenates 'pFromBuffer' onto end of 'pToBuffer'. The variables are arrays of bytes terminated with a zero character. nMaxBufferSize is the maximum size of 'pFromBuffer' and is usually created with a 'sizeof(..)' function call. Identical to the function found in conventional C 'string.h' library file.
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\String N Demo.c")*

```
string str1 = "ROBOT";                          // String 'str1' is "ROBOT".
string str2 = "C!";                             // String 'str2' is "C!".

int copy_length = 1;                            // Int 'copy_length' is set to 1.

strncat(str1, str2, copy_length);               /* Concatinate 'copy_length' amount of charac
                                                   from string 'str2' onto string 'str1'.

nxtDisplayCenteredTextLine(3, "%s", str1);      // Display the new 'str' ("ROBOTC"), (the "!"
```

**strncmp(pString1, pString2, nMaxBufferSize)**
Function compares 'pString1' with 'pString2'. Returns negative value if less than, 0 if equal and positive value if greater than. The variables are arrays of bytes terminated with a zero char. 'nMaxBufferSize' is the maximum number of bytes to compare and is usually created with a 'sizeof(..)' function call. Identical to the function found in conventional C 'string.h' library.
*Example: (program example can be found at: "\Sample Programs\NXT\Strings\String N Demo.c")*

```
string str1 = "Fett";                           // String 'str1' is "Fett".
string str2 = "Fetts";                          // String 'str2' is "Fetts".

int cmp_chars = 4;                              /* Int 'cmp_chars' is the amount of chars
                                                   to compare in "strncmp()".  Here we will
                                                   compare the first 4 chars of each string. *

if(strncmp(str1, str2, cmp_chars) < 0)                    // If 'str1' < 'str2' up to 4 cha
{
  nxtDisplayCenteredTextLine(3, "str1 is < than str2");       // Display that 'str1' is < '
}
else if(strncmp(str1, str2, cmp_chars) > 0)               // If 'str1' > 'str2' up to 4 cha
{
  nxtDisplayCenteredTextLine(3, "str1 is > than str2");       // Display that 'str1' is > '
}
else                                                      // If 'str1' == 'str2' up to 4 ch
{
  nxtDisplayCenteredTextLine(3, "str1 = str2");               // Display that 'str1' is = '
}
```

# 8.19 Task Control

Up to ten tasks can execute concurrently within a NXT user program. The 'main' or 'primary' task is automatically started when a user program is run. Execution of other tasks can be started and stopped with the 'StartTask' and 'StopTask' functions. The NXT will share CPU execution time among various tasks by giving each task a "time slice" where it will execute a group of instructions.

Each task can be assigned a priority from 0 to 255. The NXT scheduler gives execution time to the highest priority task that is waiting to run. A round robin scheduling scheme is used when there are multiple tasks ready to run all with the highest priority. Lower priority tasks will not execute until there are no tasks of higher priority that are ready to run.

A task that is waiting for a time period to expire is not ready to run. Once the time period has expired it will again become ready to run. Waiting tasks do not consume any CPU cycles.

---

```
StartTask(TaskID, nPriority);
StartTaskWithPriority(TaskID, nPriority);
```

A functional call to start or restart the execution of a specific task. The "nPriority" command is optional, and can be used to assign a priority to a task at the time the task is told to start executing. If the "nPriority" command is not specified, the task will start with a default priority of "7".

*Example:* *(program example can be found at: "Something.c")*

```
StartTask(MoveArm, 10);  //Starts the task named "MoveArm" with a priorty of "10".
```

```
StopAllTasks();
```

A function call to stop every task running, including the main program task. This function is similar to pressing the gray "exit" button on your NXT.

*Example:* *(program example can be found at: "Something.c")*

```
StopAllTasks();  //Stops all tasks currently running and exits the current program.
```

```
StopTask(TaskID);
```

Stops execution of a single task.

*Example:* *(program example can be found at: "Something.c")*

```
StopTask(MoveArm);  //Stops the task named "MoveArm"
```

```
hogCPU();
```

This special function is used to temporarily suspend the normal task scheduler and give all the CPU time to the current task. As long as this task can run, then it will get all of the CPU time until a “releaseCPU()” function call is made. You can execute the "hogCPU" command inside of task main to hog the CPU away from ROBOTC's background driver task.

*Example:* *(program example can be found at: "Something.c")*

```
hogCPU();  //Suspend every other task and focus 100% on the current task.
```

```
releaseCPU();
```

This special function is used to resume the normal task schedule. It cancels the effect of a previous “hogCPU()” function call.

*Example:* *(program example can be found at: "Something.c")*

```
releaseCPU();  //Resume other tasks and return control to the task scheduler.
```

```
AbortTimeslice();
```

Immediately ends the current task time slice and allows another task to be immediately scheduled and executed. Tasks are given CPU execution time so that highest priority tasks get all the CPU time. If there are multiple tasks with the same highest priority then they are given "slices" of CPU time in a round robin fashion. This function will immediately end the current timeslice.

*Example:* *(program example can be found at: "Something.c")*

```
AbortTimeslice();  //Aborts the current time slice of the task.
```

**nSchedulePriority**
The CPU scheduler priority for the current task. ROBOTC shares CPU execution time among various tasks by giving each task a “time slice” where it will execute a group of instructions. Each task can be assigned a priority from 0 to 255. The scheduler gives execution time to the highest priority task that is waiting to run. A round robin scheduling scheme is used when there are multiple tasks ready to run all with the highest priority. Lower priority tasks will not execute until there are no tasks of higher priority that are ready to run. By default, tasks are assigned a priority value of "7".
*Example:* *(program example can be found at: "Something.c")*

```
nSchedulePriority = 100;  //Assigns the current task with a priority of 100.
```

**kDefaultTaskPriority**
Constant. The default priority assigned to a task. Equivalent to setting a priority of 7.
*Example:* *(program example can be found at: "Something.c")*

```
int myPriority = 0;   //Creates a variable to hold the task priority value.
myPriority = kDefaultTaskPriority;  //Starts the task named "MoveArm"
```

**kHighPriority**
Constant. Highest priority that can be assigned to a task. Equivalent to setting a priority of 255.
*Example:* *(program example can be found at: "Something.c")*

```
int myPriority = 0;  //Creates a variable to hold the task priority value.
nSchedulePriority = kHighPriority;  //Assigns the Priority of the Task to the Highest Prio
myPriority = nSchedulePriority; //Sets the current priority to a variable to view in the d
```

**kLowPriority**
Constant. Lowest priority that can be assigned to a task. Equivalent to setting a priority of 0.
*Example:* *(program example can be found at: "Something.c")*

```
int myPriority = 0;  //Creates a variable to hold the task priority value.
nSchedulePriority = LowPriority; //Assigns the Priority of the Task to the Lowest Priority
myPriority = nSchedulePriority; //Sets the current priority to a variable to view in the d
```

# 8.20 Timing

An internal 32-bit clock is maintained by the NXT firmware. It counts in units of 1-millisecond. Four timers (T1, T2, T3 and T4) are built using this timing capability. These four timers can be individually reset to zero within a program. Theses timers are useful for measuring elapsed time of events.

16-bit integers are the default variable type for ROBOTC on the NXT. All of the timing functions operate with 16-bit signed variables. This does mean that caution should be exercised in programming to avoid overflow of timer values. A 16-bit signed variable has positive values in the range 0 to 32,767 and programs should periodically reset any timers that they use to prevent overflow.

---

**wait1Msec(nMSec);**
**wait10Msec(nTenMSec);**
Program execution will wait for the specified number of clock units. Units can be in either 1-millisecond or 10-millisecond counts.The maximum interval that can be specified is either 32.767 seconds or 327.67

seconds depending on which function is used.An alternative, and far less efficient, method to perform a wait is to continually execute a tight code loop looking to see if a timer has reached the desired interval. It is best to use the wait functions to insert a programmed delay in a program because tasks that are waiting do not consume any CPU cycles. This makes the most number of CPU cycles available for other tasks.
*Example:* *(program example can be found at: "Something.c")*

```
wait1Msec(1000);   //The program will wait for 1 second before moving on. (1ms * 1000)
wait10Msec(1000);  //The program will wait for 10 seconds before moving on. (10ms * 1000)
```

**ClearTimer(theTimer);**
Timers start counting as soon as the NXT is powered on. A user's program should reset a timer before using it, so use this function to reset the value of the specified timer to zero.
*Example:* *(program example can be found at: "Something.c")*

```
ClearTimer(T1);  //Resets the value of Timer "T1" back to zero seconds.
```

**time1[]**
**time10[]**
**time100[]**
These three arrays hold the current value of the respective timers. Each of the timer values can be retrieved in units of 1, 10 and 100 milliseconds depending on which array is used. For example, time1[T1] retrieves the value of timer T1 in units of 1-msec and time10[T1] retrieves the value using a 10-msec tick. And time100[T1] retrieves the value using 100-msec tick. Note that the arrays are “linked”. Setting time1[T1] = 0; will also reset the value of time10[T1] and time100[T1]. The value returned is a signed integer, so each array will meet its upper bounds at a value of 32,768 ticks.
*Example:* *(program example can be found at: "Something.c")*

```
int valTime1, valTime10, valTime100;  //Create three integers to read the value of the tim

valTime1 = time1[T1];  //Gets the value of Timer T1 in 1ms increments and stores it in a v
valTime10 = time10[T1];  //Gets the value of Timer T1 in 1ms increments and stores it in a
valTime100 = time100[T1];  //Gets the value of Timer T1 in 1ms increments and stores it in
```

**nSysTime**
This variable contains the value of the lower 16-bits of the internal 1-msec clock. This variable is reset when NXT is first powered on.
Example: (program example can be found at: "Something.c")

```
int varSysTime;  //Creates a variable
varSysTime = nSysTime;  //Stores the current value of nSysTime to a variable
```

**nPgmTime**
This variable contains the value of the lower 16-bits of the internal 1-msec clock. This variable is reset when user program first starts running. This clock does not increment when the program is in a debugger "suspended" state which is useful during single step debugging as the clock does not increment.
Example: (program example can be found at: "Something.c")

```
int varPgmTime;  //Creates a variable
varPgmTime = nPgmTime;  //Stores the current value of nPgmTime to a variable
```

**nClockMinutes**
This read/write variable provides access to the NXT clock described above. The value ranges from 0 to 1439 before it wraps around back to 0. Note: there are 1440 minutes in 24 hours.
Example: (program example can be found at: "Something.c")

```
int varClock;  //Creates a variable
varClock = nClockMinutes;  //Stores the current value of the NXT Clock to a variable
```

# 9. Robot Algebra

## 9.1 Setup Functions

There are a number of other useful commands that you can use in your robot program. The following is a list of those commands:

***Initialize Robot*** - `initializeRobot(robotModel);`
This command prepares your robot before starting a sequence of moves. It tells the program what robot is doing the moves. You should use the model that corresponds to the robot you are using.

***Start Move Sequence*** - `movesStart(waitForRoutine,waitForParam);`
This command should be called right before you are about to start your first movement. This command will indicate that the sequence of moves is about to start, and will wait for your starting command.

There are 4 options for a starting command:
1. Press the touch sensor - **`WFTouch`** (this is the default)
2. Make a loud noise with the sound sensor - **`WFLoud`**
3. Wait for a set number of seconds - **`WFTime`**
4. Just move right into the routine - **`WFNothing`**

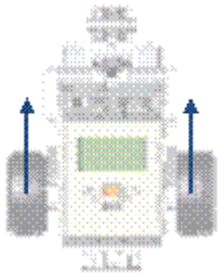
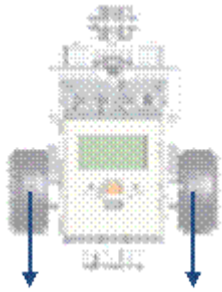
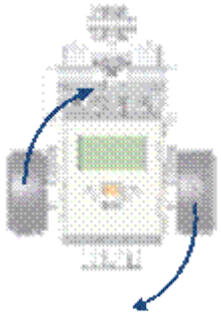
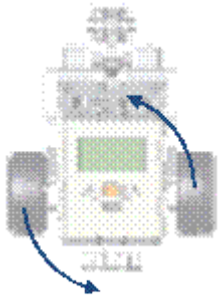
***End Dance Routine*** - `movesEnd();`
This commands indicates the sequence of moves is over and displays that on the robot screen.

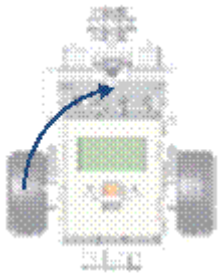
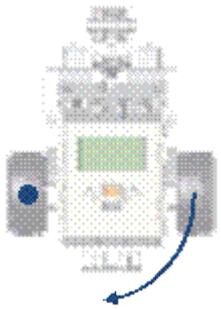
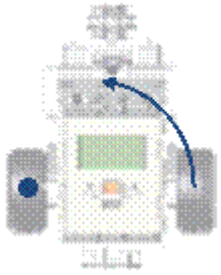
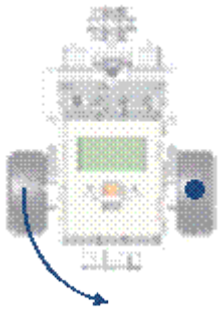
***Set Move Mode*** - `setMoveMode(timingMoveMode,typeMoveMode);`
This command provides different options for analyzing the different moves in a sequence of moves. The timing mode can be set to have all the moves execute one after the other (**`MMContinuousMoves`** – the default) or to execute only one move at a time followed by a pause and wait for you to hit the touch sensor before moving onto the next move (**`MMOneMoveAtATime`** ).

The type mode can be set to execute all move types (**`MMAllMoveTypes`** – the default), to execute only straight moves (**`MMStraightsOnly`**), to execute only turning moves (**`MMTurnsOnly`**), or to execute only a specific type of turn ( **`MMPointTurnsOnly`** or **`MMSwingTurnsOnly`** ).

## 9.2 Robot Movements

When working with your robots in this unit, the robot will only be capable of performing a limited set of movements. The following is a list of the movements that your robot is capable of performing, along with a short description of how the robot does the move and the move command. The thin arrows and marks indicate the movement of the wheels.

| Title | Picture | Description | Usage |
|---|---|---|---|
| Straight - Forward |  | Both wheels rotate forward at the same speed and the robot moves straight forward. | `straightForward(mRot,mRotPerSec);` |
| Straight - Backwards |  | Both wheels rotate forward at the same speed and the robot moves straight backward. | `straightBackward(mRot,mRotPerSec);` |
| Point Turn - Right |  | Both wheels rotate at the same speed, but the left wheel rotates forward and the right wheel rotates backward, so the robot turns to its right around its center. This makes a sharp turn in place. | `turnPointRight(mRot,mRotPerSec);` |
| Point Turn - Left |  | Both wheels rotate at the same speed, but the right wheel rotates forward and the left wheel rotates backward, so the robot turns to its left around its center. This makes a sharp turn in place. | `turnPointLeft(mRot,mRotPerSec);` |

| | | | |
|---|---|---|---|
| Swing Turn - Forward Right |  | The left wheel rotates forward and the right wheel does not move, so the robot pivots around the right wheel as it turns forward. This makes a wider turn. | `turnSwingRightForward(mRot,mRotPerSec);` |
| Swing Turn - Backward Right |  | The right wheel rotates backward and the left wheel does not move, so the robot pivots around the left wheel as it turns backward. This makes a wider turn. | `turnSwingRightBackward(mRot,mRotPerSec);` |
| Swing Turn - Forward Left |  | The right wheel rotates forward and the left wheel does not move, so the robot pivots around the left wheel as it turns forward. This makes a wider turn. | `turnSwingLeftForward(mRot,mRotPerSec);` |
| Swing Turn - Backward Left |  | The left wheel rotates backward and the right wheel does not move, so the robot pivots around the left wheel as it turns backward. This makes a wider turn. | `turnSwingLeftBackward(mRot,mRotPerSec);` |

# 10. Natural Language

## 10.1 NXT

### 10.1.1 Setup Functions

# ROBOTC Natural Language - NXT: (PDF)

## Setup Functions:

| Title | Picture | Description | Default Usage and |
|---|---|---|---|
| **Robot Type** |  | Choose which robot you want to write a program for.<br><br>Acceptable robots: Rembot or none (no setup).<br><br>(If you don't use this function at all, it will default to none!) | `robotType(type)`<br>*default bot:* ***none***<br><br>`robotType();` |

(print me in landscape mode)

### 10.1.2 Wait Functions

# ROBOTC Natural Language - NXT: (PDF)

## Wait Functions:

| Title | Picture | Description | Default Usage and |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Wait** |  | Wait an amount of time measured in seconds. | `wait(time)`<br>*default time: **1.0** seconds*<br>`forward();`<br>`wait();`<br>`stop();` |
| **Wait in Milliseconds** |  | Wait an amount of time measured in milliseconds. | `waitInMilliseconds(time`<br>*default time: **1000** milliseconds*<br>`forward();`<br>`waitInMilliseconds();`<br>`stop();` |

(print me in landscape mode)

### 10.1.3 Movement Functions

# ROBOTC Natural Language - NXT: (PDF)

## Movement Functions:

Title Picture Description Default Usage ar

**Start Motor**

Set a motor to a speed.

Range: -100 to 100

Acceptable Motors: ports A through C,
(and your names for them given in Motors and Sensors Setup.)

```
startMotor(motor, spe
```
*default motor:* ***motorA***
*default speed:* ***75***

```
startMotor();
wait();
stopMotor();
```

---

**Stop Motor**


Stops a motor.

Acceptable Motors: ports A through C,
(and your names for them given in Motors and Sensors Setup.)

```
stopMotor(motor)
```
*default motor:* ***motorA***

```
startMotor();
wait();
stopMotor();
```

---

(print me in landscape mode)

### 10.1.4 Robot Movement Functions

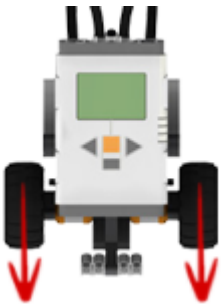
# ROBOTC Natural Language - NXT: (PDF)

## Robot Movement Functions:

| Title | Picture | Description | Default Usage and Sample |
|---|---|---|---|

---

**Forward**



Both wheels rotate forward at the same speed and the robot moves straight forward.

Range: -100 to 100
(Forward will always move your robot forward.)

```
forward(speed)
```
*default speed:* ***75***

```
forward();
wait();
stop();
```

**Backward**

Both wheels rotate backward at the same speed and the robot moves straight backward.

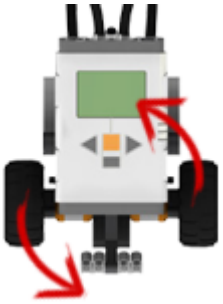
Range: -100 to 100
(Backward will always move your robot backward.)

```
backward(speed)
```
*default speed:* ***-75***

```
backward();
wait();
stop();
```

---

**Point Turn**



Both wheels rotate at the same speed but in opposite directions, causing the robot to turn around it's center. This makes a sharp turn in place.

Range: -100 to 100

```
pointTurn(direction, speed)
```
*default direction:* ***right***
*default speed:* ***75***

```
pointTurn();
wait();
stop();
```

---

**Swing Turn**

One wheel rotates while the other does not move, causing the robot to turn around the stopped wheel. This makes a wide turn.

Range: -100 to 100

```
swingTurn(direction, speed)
```
*default direction:* ***right***
*default speed:* ***75***

```
swingTurn();
wait();
stop();
```

---

**Stop**

Both wheels do not move, causing the robot to stop.

```
stop()
```

```
forward();
wait();
stop();
```

---

**Line Track - For Time**

The robot will track a black line on a white surface for a specified time in seconds.

Threshold Range: (dark) 0 to 100 (light)

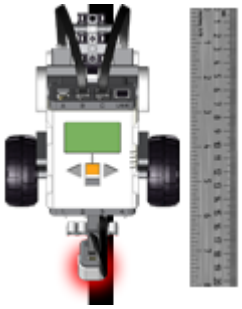
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
lineTrackForTime(time, threshold, sensorPort)
```
*default time:* ***5.0*** *seconds*
*default threshold:* ***45***
*default sensors:* ***S3***

```
lineTrackForTime();
stop();
```

---

**Line Track - For Rotations**



The robot will track a black line on a white surface for a specified distance in rotations.

Threshold Range: (dark) 0 to 100 (light)

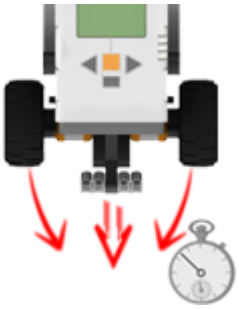
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
lineTrackForRotations(rot, threshold, sensorPort)
```
*default rotations:* ***3.0*** *rotations*
*default threshold:* ***45***
*default sensors:* ***S3***

```
lineTrackForRotations();
stop();
```

---

**Move Straight - For Time**



The robot will use encoders to maintain a straight course for a length of time in seconds.

*NOTE* This function only supports moving forward and only at one speed setting. Future implementations may include moving backwards and variable speeds.

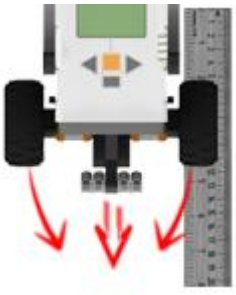
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
moveStraightForTime(time, rightEncoder, leftEncoder)
```
*default time:* ***5.0*** *seconds*
*default motor encoders:* ***motorB, motorC (Right, Left)***

```
moveStraightForTime();
stop();
```

---

**Move Straight - For Rotations**



The robot will use encoders to maintain a straight course for a distance in rotations (360 encoder counts = 1 rotation).

*NOTE* This function only supports moving forward and only at one speed setting. Future implementations may include moving backwards and variable speeds.

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
moveStraightForRotations(rot, rightEncoder, leftEncoder)
```
*default rotations:* ***1.0*** *rotations*
*default motor encoders:* ***motorB, motorC (Right, Left)***

```
moveStraightForRotations();
stop();
```

**Tank Control**

The robot will be remote controlled in such a way that the left motor is mapped to the left joystick and the right motor is mapped to the right joystick.

*NOTE* This function only supports 2 channels and works best with the joystics.

```
tankControl(rightJoystick, leftJoystick, threshold)
```
*default joysticks:* ***joystick.joy1_y2, joystick.joy1_y1 (Right, Left)***
*default threshold:* ***10***

```
while(true)
{
  tankControl();
}
```

---

**Arcade Control**

The robot will be remote controlled in such a way that the movement of the robot is mapped to a single joystick, much like a retro arcade game.

*NOTE* This function only supports 2 channels and works best with the joystics.

```
arcadeControl(verticalJoystick, horizontalJoystick, threshold)
```
*default joysticks:* ***joystick.joy1_y2, joystick.joy1_x2 (Vertical, Horizontal)***
*default threshold:* ***10***

```
while(true)
{
  arcadeControl();
}
```

---

(print me in landscape mode)

### 10.1.5 Until Functions

# ROBOTC Natural Language - NXT: (PDF)

## Until Functions:

| Title | Picture | Description | Default Usage an |
|---|---|---|---|

**Until Touch**

The robot does what it was doing until the touch sensor is pressed in.

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilTouch(sensorPort)
```
*default sensor:* ***S1***

```
forward();
untilTouch();
stop();
```

---

**Until Release**

The robot does what it was doing until the touch sensor is released.

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilRelease(sensorPor
```
*default sensor:* ***S1***

```
forward();
untilRelease();
stop();
```

---

**Until Bump**

The robot does what it was doing until the touch sensor is pressed in and then released out. A delay time in milliseconds can be specified.

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

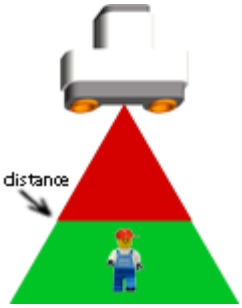
Acceptable Range for Delay Time:
0 to 3,600,000.

```
untilBump(sensorPort, 
```
*default sensor:* ***S1***
*default delay time:* ***10*** *milliseconds*

```
forward();
untilBump();
stop();
```

---

**Until Sonar - Greater Than**




The robot does what it was doing until the sonar sensor reads a value greater than a set distance in centimeters.

Range: 0 to 254 (A value of 255 means it cannot detect anything.)

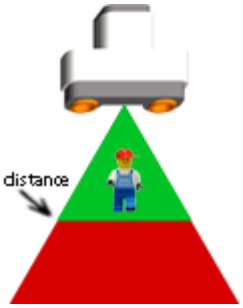
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSonarGreaterThan(
sensorPort)
```
*default distance:* ***30*** *centimeters*
*default sensor:* ***S4***

```
forward();
untilSonarGreaterThan
stop();
```

---

**Until Sonar - Less Than**




The robot does what it was doing until the sonar sensor reads a value less than a set distance in centimeters.

Range: 0 to 254 (A value of 255 means it cannot detect anything.)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSonarLessThan(dis
sensorPort)
```
*default distance: **30** centimeters*
*default sensor: **S2***

```
forward();
untilSonarLessThan();
stop();
```

---

**Until Sound - Greater Than**

The robot does what it was doing until the sound sensor reads a value greater than a set threshold level.

Range: (quiet) 0 to 100 (loud)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSoundGreaterThan(
sensorPort)
```
*default threshold: **50***
*default sensor: **S2***

```
forward();
untilSoundGreaterThan(
stop();
```

---

**Until Sound - Less Than**

The robot does what it was doing until the sound sensor reads a value less than a set threshold level.

Range: (quiet) 0 to 100 (loud)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSoundLessThan(thre
sensorPort)
```
*default threshold: **50***
*default sensor: **S2***

```
forward();
untilSoundLessThan();
stop();
```

---

**Until Button Press**


The robot does what it was doing until a button on the NXT is pressed.

Acceptable Buttons:
centerBtnNXT, rightBtnNXT, leftBtnNXT

```
untilButtonPress(butto
```
*default button: **centerBtnNXT***

```
forward();
untilButtonPress();
stop();
```

---

**Until Light**

The robot does what it was doing until the light sensor reads a value lighter than the threshold.

Range: (light) 0 to 100 (dark)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilLight(threshold, 
```
*default threshold:* ***45***
*default sensor:* ***S3***

```
forward();
untilLight();
stop();
```

---

**Until Dark**

The robot does what it was doing until the light sensor reads a value darker than the threshold.

Range: (light) 0 to 100 (dark)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilDark(threshold, s
```
*default threshold:* ***45***
*default sensor:* ***S3***

```
forward();
untilDark();
stop();
```

---

**Until Rotations**


The robot does what it was doing until the quadrature encoder rotations match the desired value.

Range: 0 to >65000

Acceptable Encoders: motor ports A through C (and your names for them given in Motors and Sensors Setup.)

```
untilRotations(rotatio
motorEncoderPort)
```
*default rotations:* ***1.0*** *rotations*
*default motor encoder:* ***motorB***

```
forward();
untilRotations();
stop();
```

---

**Until Encoder Counts**


The robot does what it was doing until the encoder counts match the desired value.

Range: 0 to >65000

Acceptable Encoders: motor ports A through C (and your names for them given in Motors and Sensors Setup.)

```
untilEncoderCounts(cou
motorEncoderPort)
```
*default counts:* ***360*** *encoder counts*
*default motor encoder:* ***motorB***

```
forward();
untilEncoderCounts();
stop();
```

---

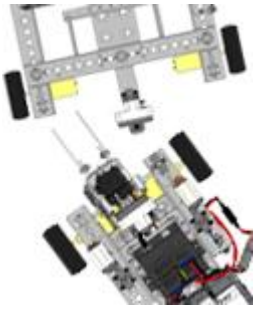
(print me in landscape mode)

## 10.2 TETRIX

### 10.2.1 Setup Functions

# ROBOTC Natural Language - TETRIX: (PDF)

## Setup Functions:

| Title | Picture | Description | Default Usage and |
|---|---|---|---|
| **Robot Type** |  | Choose which robot you want to write a program for.<br><br>Acceptable robots: Mantis, Ranger or none (no setup).<br><br>(If you don't use this function at all, it will default to none!) | `robotType(type)`<br>*default bot:* ***none***<br><br>`robotType();` |

(print me in landscape mode)

### 10.2.2 Wait Functions

# ROBOTC Natural Language - TETRIX: (PDF)

## Wait Functions:

| Title | Picture | Description | Default Usage and |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Wait** |  | Wait an amount of time measured in seconds. | `wait(time)`<br>*default time: **1.0** seconds*<br>`forward();`<br>`wait();`<br>`stop();` |
| **Wait in Milliseconds** |  | Wait an amount of time measured in milliseconds. | `waitInMilliseconds(time`<br>*default time: **1000** milliseconds*<br>`forward();`<br>`waitInMilliseconds();`<br>`stop();` |

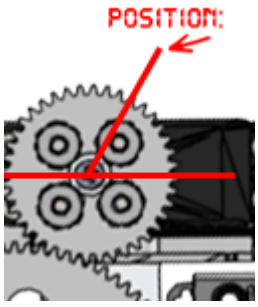
(print me in landscape mode)

### 10.2.3 Movement Functions

# ROBOTC Natural Language - TETRIX: (PDF)

## Movement Functions:

| Title | Picture | Description | Default Sample |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Start Motor** |  | Set a motor to a speed.<br><br>Range: -100 to 100<br><br>Acceptable Motors: ports A through C,<br>(and your names for them given in Motors and Sensors Setup.) | `startMotor`<br>*default motor:* **n**<br>*default speed:* **7**<br><br>`startMotor`<br>`wait();`<br>`stopMotor` |
| **Stop Motor** |  | Stops a motor.<br><br>Acceptable Motors: ports A through C,<br>(and your names for them given in Motors and Sensors Setup.) | `stopMotor(`<br>*default motor:* **n**<br><br>`startMotor`<br>`wait();`<br>`stopMotor` |
| **Set Servo** |  | Set a servo to a desired position.<br><br>Acceptable Servos: servo-ports 1 through 24, (and your names for them given in Motors and Sensors Setup.) | `setServo(s`<br>*default servo:* **s**<br><br>`setServo()` |

(print me in landscape mode)

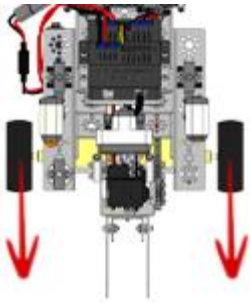
### 10.2.4 Robot Movement Functions

# ROBOTC Natural Language - TETRIX: (PDF)

## Robot Movement Functions:

Title Picture Description Default Usage and Sam

**Forward**



Both wheels rotate forward at the same speed and the robot moves straight forward.

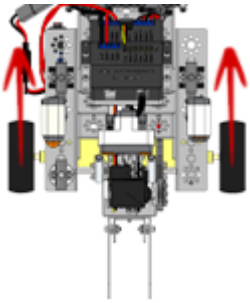
Range: -100 to 100
(Forward will always move your robot forward.)

```
forward(speed)
```
*default speed:* ***75***

```
forward();
wait();
stop();
```

---

**Backward**



Both wheels rotate backward at the same speed and the robot moves straight backward.

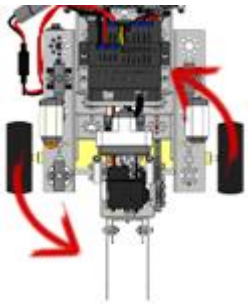
Range: -100 to 100
(Backward will always move your robot backward.)

```
backward(speed)
```
*default speed:* ***-75***

```
backward();
wait();
stop();
```

---

**Point Turn**



Both wheels rotate at the same speed but in opposite directions, causing the robot to turn around it's center. This makes a sharp turn in place.

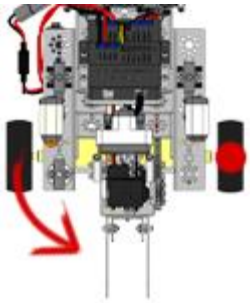
Range: -100 to 100

```
pointTurn(direction, speed)
```
*default direction:* ***right***
*default speed:* ***75***

```
pointTurn();
wait();
stop();
```

---

**Swing Turn**



One wheel rotates while the other does not move, causing the robot to turn around the stopped wheel. This makes a wide turn.

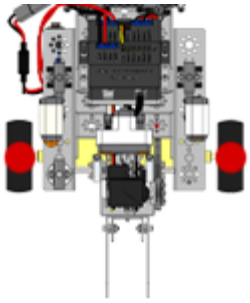
Range: -100 to 100

```
swingTurn(direction, speed)
```
*default direction:* ***right***
*default speed:* ***75***

```
swingTurn();
wait();
stop();
```

---

**Stop**



Both wheels do not move, causing the robot to stop.

```
stop()
```

```
forward();
wait();
stop();
```

---

**Line Track - For Time**

The robot will track a black line on a white surface for a specified time in seconds.

Threshold Range: (dark) 0 to 100 (light)

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
lineTrackForTime(time, thresho
sensorPort)
```
*default time:* ***5.0*** *seconds*
*default threshold:* ***45***
*default sensors:* ***S3***

```
lineTrackForTime();
stop();
```

---

**Line Track - For Rotations**

The robot will track a black line on a white surface for a specified distance in rotations.

Threshold Range: (dark) 0 to 100 (light)

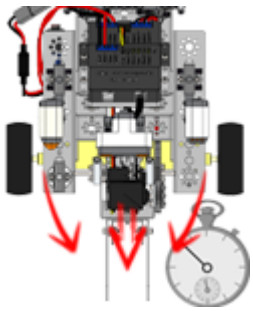
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
lineTrackForRotations(rot,
threshold, sensorPort)
```
*default rotations:* ***3.0*** *rotations*
*default threshold:* ***45***
*default sensors:* ***S3***

```
lineTrackForRotations();
stop();
```

---

**Move Straight - For Time**



The robot will use encoders to maintain a straight course for a length of time in seconds.

*NOTE* This function only supports moving forward and only at one speed setting. Future implementations may include moving backwards and variable speeds.

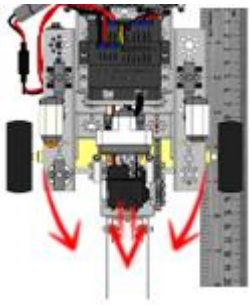
Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
moveStraightForTime(time,
rightEncoder, leftEncoder)
```
*default time:* ***5.0*** *seconds*
*default motor encoders:* ***motorB, motorC (Righ***

```
moveStraightForTime();
stop();
```

---

**Move Straight - For Rotations**



The robot will use encoders to maintain a straight course for a distance in rotations (360 encoder counts = 1 rotation).

*NOTE* This function only supports moving forward and only at one speed setting. Future implementations may include moving backwards and variable speeds.

Acceptable Sensors: sensor ports 1 through 4 (and your names for them given in Motors and Sensors Setup.)

```
moveStraightForRotations(rot, rightEncoder, leftEncoder)
```
*default rotations:* ***1.0*** *rotations*
*default motor encoders:* ***motorB, motorC (Righ***

```
moveStraightForRotations();
stop();
```

---

**Tank Control**

The robot will be remote controlled in such a way that the left motor is mapped to the left joystick and the right motor is mapped to the right joystick.

*NOTE* This function only supports 2 channels and works best with the joystics.

```
tankControl(rightJoystick, leftJoystick, threshold)
```
*default joysticks:* ***joystick.joy1_y2, joystick.joy***
***(Right, Left)***
*default threshold:* ***10***

```
while(true)
{
  tankControl();
}
```

---

**Arcade Control**

The robot will be remote controlled in such a way that the movement of the robot is mapped to a single joystick, much like a retro arcade game.

*NOTE* This function only supports 2 channels and works best with the joystics.

```
arcadeControl(verticalJoystick horizontalJoystick, threshold)
```
*default joysticks:* ***joystick.joy1_y2, joystick.joy***
***(Vertical, Horizontal)***
*default threshold:* ***10***

```
while(true)
{
  arcadeControl();
}
```

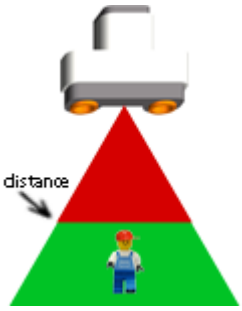
---

(print me in landscape mode)

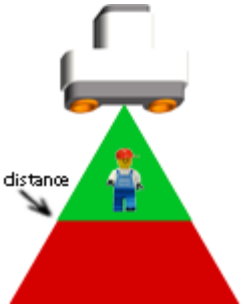
## 10.2.5 Until Functions

# ROBOTC Natural Language - TETRIX: (PDF)

## Until Functions:

| Title | Picture | Description | Default Usage an |
|---|---|---|---|
| **Until Touch** |  | The robot does what it was doing until the touch sensor is pressed in.<br><br>Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.) | `untilTouch(sensorPort)`<br>*default sensor:* ***S2***<br><br>`forward();`<br>`untilTouch();`<br>`stop();` |
| **Until Release** |  | The robot does what it was doing until the touch sensor is released.<br><br>Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.) | `untilRelease(sensorPor`<br>*default sensor:* ***S2***<br><br>`forward();`<br>`untilRelease();`<br>`stop();` |
| **Until Bump** |  | The robot does what it was doing until the touch sensor is pressed in and then released out. A delay time in milliseconds can be specified.<br><br>Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)<br><br>Acceptable Range for Delay Time:<br>0 to 3,600,000. | `untilBump(sensorPort,`<br>*default sensor:* ***S2***<br>*default delay time:* ***10*** *milliseconds*<br><br>`forward();`<br>`untilBump();`<br>`stop();` |
| **Until Sonar - Greater Than** |  | The robot does what it was doing until the sonar sensor reads a value greater than a set distance in centimeters.<br><br>Range: 0 to 254 (A value of 255 means it cannot detect anything.)<br><br>Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.) | `untilSonarGreaterThan(`<br>`sensorPort)`<br>*default distance:* ***30*** *centimeters*<br>*default sensor:* ***S4***<br><br>`forward();`<br>`untilSonarGreaterThan`<br>`stop();` |

**Until Sonar - Less Than**




The robot does what it was doing until the sonar sensor reads a value less than a set distance in centimeters.

Range: 0 to 254 (A value of 255 means it cannot detect anything.)

Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSonarLessThan(dis
sensorPort)
```
*default distance: **30** centimeters*
*default sensor: **S2***

```
forward();
untilSonarLessThan();
stop();
```

---

**Until Sound - Greater Than**

The robot does what it was doing until the sound sensor reads a value greater than a set threshold level.

Range: (quiet) 0 to 100 (loud)

Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSoundGreaterThan(
sensorPort)
```
*default threshold: **50***
*default sensor: **S2***

```
forward();
untilSoundGreaterThan(
stop();
```

---

**Until Sound - Less Than**

The robot does what it was doing until the sound sensor reads a value less than a set threshold level.

Range: (quiet) 0 to 100 (loud)

Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilSoundLessThan(thre
sensorPort)
```
*default threshold: **50***
*default sensor: **S2***

```
forward();
untilSoundLessThan();
stop();
```

---

**Until Button Press**


The robot does what it was doing until a button on the NXT is pressed.

Acceptable Buttons:
centerBtnNXT, rightBtnNXT, leftBtnNXT

```
untilButtonPress(butto
```
*default button: **centerBtnNXT***

```
forward();
untilButtonPress();
stop();
```

---

**Until Light**

The robot does what it was doing until the light sensor reads a value lighter than the threshold.

Range: (light) 0 to 100 (dark)

Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilLight(threshold, 
default threshold: 45
default sensor: S3

forward();
untilLight();
stop();
```

---

**Until Dark**

The robot does what it was doing until the light sensor reads a value darker than the threshold.

Range: (light) 0 to 100 (dark)

Acceptable Sensors: sensor ports 2 through 4 (and your names for them given in Motors and Sensors Setup.)

```
untilDark(threshold, se
default threshold: 45
default sensor: S3

forward();
untilDark();
stop();
```

---

**Until Rotations**


The robot does what it was doing until the quadrature encoder rotations match the desired value.

Range: 0 to >65000

Acceptable Encoders: motor ports A through K (and your names for them given in Motors and Sensors Setup.)

```
untilRotations(rotatio
motorEncoderPort)
default rotations: 1.0 rotations
default motor encoder: motorB

forward();
untilRotations();
stop();
```

---

**Until Encoder Counts**


The robot does what it was doing until the encoder counts match the desired value.

Range: 0 to >65000

Acceptable Encoders: motor ports A through K (and your names for them given in Motors and Sensors Setup.)

```
untilEncoderCounts(cou
motorEncoderPort)
default counts: 360 encoder counts
default motor encoder: motorB

forward();
untilEncoderCounts();
stop();
```

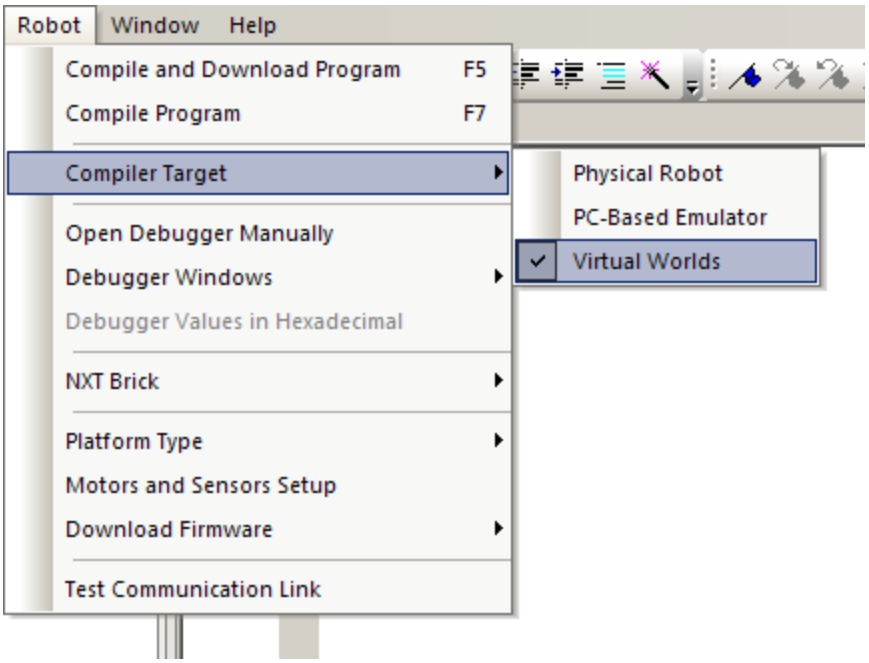
---

(print me in landscape mode)

# 11. Virtual Worlds

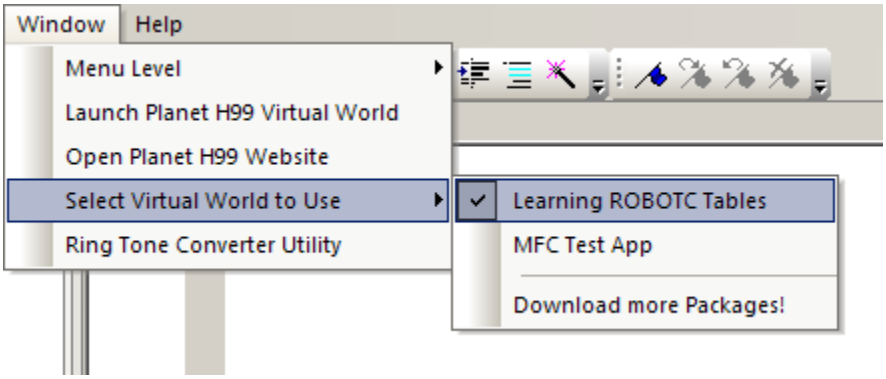
## 11.1 Using Virtual Worlds

**Setting "Debugger Target" to Virtual Worlds:**
Before you can use a Virtual World, please make sure that your "Debugger Target" is set to "Virtua Worlds":




---

**Selecting a Virtual World for use:**
To select a Virtual World Package for use with your program, click on Window >> Select Virtual World for Use, and select your package:




Downloaded packages will show up in this menu.

**Running your program:**

Now, when you compile and download your program it will automatically bring up the selected virtual world and your program will be ready to run!