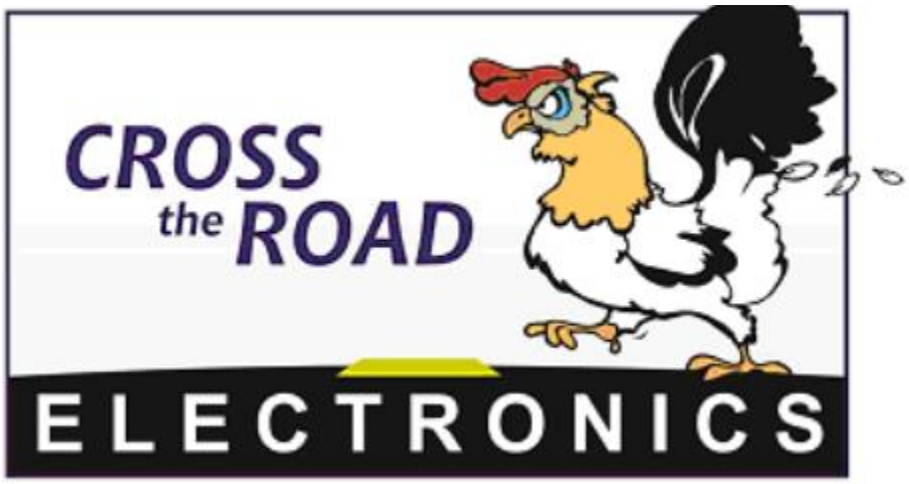
TALON SRX Software Reference Manual

Rev 1.3



Cross The Road Electronics

www.crosstheroadelectronics.com

Table of Contents

1. CAN bus Device Basics 8
2. roboRIO Web-based Configuration: Firmware and diagnostics 9
2.1. Device ID ranges 9
2.2. Common ID Talons 10
2.3. Firmware Field-upgrade a Talon SRX 12
2.3.1. When I update firmware, I get “You do not have permissions…” 14
2.3.2. What if Firmware Field-upgrade is interrupted? 16
2.3.3. Other Field-upgrade Failure Modes 17
2.3.4. Where to get CRF files? 18
2.4. Self-Test 19
2.4.1. Clearing Sticky Faults 21
2.5. Custom Names 22
2.5.1. Re-default custom name 23
3. Creating a Talon Object (and basic drive) 24
3.1. Programming API and Device ID 24
3.2. New Classes/Virtual Instruments 24
3.3. LabVIEW 25
3.4. C++ 26
3.5. Java 26
3.6 Changing Mode 27
3.6.1. LabVIEW 27
3.6.2. C++ 27
3.6.3. Java 27
3.6.4. Check Control Mode with Self-Test 28
4. Limit Switch and Neutral Brake Mode 29
4.1. Default Settings 29
4.2. roboRIO Web-based Configuration: Limit Switch and Brake 30
4.3. Overriding Brake and Limit Switch with API 31
4.3.1. LabVIEW 32
4.3.2. C++ 32
4.3.3. Java 32
4.4. Changing limit switch mode between “Normally Open” or “Normally Closed” 33

4.4.1. LabVIEW....33
4.4.2. C++....33
4.4.3. Java....33
5. Getting Status and Signals....34
5.1. LabVIEW....35
5.2. C++....36
5.3. Java....37
6. Setting the Ramp Rate....38
6.1. LabVIEW....38
6.2. C++....38
6.3. Java....38
6.4. What is the slowest ramp possible?....38
7. Selecting a Feedback Device....39
7.1. LabVIEW....39
7.2. C++....39
7.3. Java....40
7.4. Reversing sensor direction, best practices.....41
8. Soft Limits....42
8.1. LabVIEW....42
8.2. C++....43
8.3. Java....43
9. Follower Mode....44
9.1. LabVIEW....44
9.2. C++....44
9.3. Java....44
10. Closed-Loop Modes....45
11. Motor Control Profile Parameters....45
11.1. Persistent storage and Reset/Startup behavior....46
11.2. Inspecting Signals....48
12. Position/Velocity Closed-Loop Example....49
12.1. Setting Motor Control Profile Parameters....49
12.1.1. LabVIEW....49
12.1.2. C++....49

12.1.3. Java 49
12.2. Clearing Integral Accumulator (I Accum) 50
12.2.1. LabVIEW 50
12.2.2. C++/Java 50
12.2.3. Is Integral Accum cleared any other time? 50
13. Setting Sensor Position 51
13.1. LabVIEW 51
13.2. C++ 51
13.3. Java 51
14. Fault Flags 52
14.1. LabVIEW 52
14.2. C++ 52
14.3. Java 53
15. CAN bus Utilization/Error metrics 54
15.1. How many Talons can we use? 55
16. Troubleshooting Tips and Common Questions 56
16.1. When I press the B/C CAL button, the brake LED does not change, neutral behavior does not change. 56
16.2. Changing certain settings in Disabled Loop doesn't take effect until the robot is enabled. 56
16.3. The robot is TeleOperated/Autonomous enabled, but the Talon SRX continues to blink orange (disabled). 56
16.4. When I attach/power a particular Talon SRX to CAN bus, The LEDs on every Talon SRX occasionally blink red. Motor drive seems normal. 56
16.5. If I have a slave Talon SRX following a master Talon SRX, and the master Talon SRX is disconnected/unpowered, what will the slave Talon SRX do? 57
16.6. Is there any harm in creating a software Talon SRX for a device ID that's not on the CAN bus? Will removing a Talon SRX from the CAN bus adversely affect other CAN devices? 57
16.7. Driver Station log says Error on line XXX of CANTalon.cpp 57
16.8. Driver Station log says -44087 occurred at NetComm... 58
16.9. Why are there multiple ways to get the same sensor data? GetEncoder() versus GetSensor()? 58
16.10. So there are two types of ramp rate? 58
16.11. Why are there two feedback "analog" device types: Analog Encoder and Analog Potentiometer? 59

16.12. After changing the mode in C++/Java, motor drive no longer works. Self-Test says “No Drive” mode? 59
16.13. All CAN devices have red LEDs. Recommended Preliminary checks for CAN bus. 60
16.14. Driver Station reports “MotorSafetyHelper.cpp: A timeout…”, motor drive no longer works. roboRIO Web-based Configuration says “No Drive” mode? Driver Station reports error -44075? 60
16.15. Motor drive stutters, misbehaves? Intermittent enable/disable? 61
16.16. What to expect when devices are disconnected in roboRIO’s Web-based Configuration. Failed Self-Test? 62
16.17. When I programmatically change the “Normally Open” vs “Normally Closed” state of a limit switch, the Talon SRX blinks orange momentarily. 63
16.18. How do I get the raw ADC value (or voltage) on the Analog Input pin? 63
16.19. Recommendation for using relative sensors. 63
16.20. Does anything get reset or lost after firmware updates? 63
16.21. Analog Position seems to be stuck around ~100 units? 63
16.22. Limit switch behavior doesn’t match expected settings. 64
16.23. How fast can I control just ONE Talon SRX? 65
16.24. Expected symptoms when there is excessive signal reflection. 65
17. Units and Signal Definitions 66
17.1. (Quadrature) Encoder Position 66
17.2. Analog Potentiometer 66
17.3. Analog Encoder, “Analog-In Position” 66
17.4. EncRise (a.k.a. Rising Counter) 66
17.5. Duty-Cycle (Throttle) 66
17.6. (Voltage) Ramp Rate 67
17.7. (Closed-Loop) Ramp Rate 67
17.8. Integral Zone (I Zone) 67
17.9. Integral Accumulator (I Accum) 67
17.10. Reverse Feedback Sensor 67
17.11. Reverse Closed-Loop Output 67
17.12. Closed-Loop Error 68
17.13. Closed-Loop gains 68
18. How is the closed-loop implemented? 69
19. Motor Safety Helper 71

19.1. Best practices 71
19.2. C++ example....72
19.3. Java example....73
19.4. LabVIEW Example....73
20. Going deeper - How does the framing work? 74
20.1. General Status....74
20.2. Feedback Status....74
20.3. Quadrature Encoder Status....74
20.4. Analog Input / Temperature / Battery Voltage Status....75
20.5. Modifying Status Frame Rates 75
20.5.1. C++....75
20.5.2. Java 76
20.5.3. LabVIEW Example 76
20.6. Control Frame 77
20.7. Modifying the Control Frame Rate....77
21. Functional Limitations 78
21.1. Firmware 1.1-1.4: Voltage Compensation Mode is not supported. 78
21.2. Firmware 1.1-1.4: Current Closed-Loop Mode is not supported. 78
21.3. Firmware 1.1-1.4: EncFalling Feedback device not supported. 78
21.4. Firmware 1.1-1.4: `ConfigMaxOutputVoltage()` not supported.....78
21.5. Firmware 1.1-1.4: `ConfigFaultTime()` not needed 78
21.6. Firmware 1.1: Changes in Limit Switch “Normally Open” vs “Normally Closed” may require power cycle during a specific circumstance.....78
21.7. LabVIEW: EncRising Feedback mode not selectable.....78
21.8. LabVIEW/C++/Java API: `ConfigEncoderCodesPerRev()` is not supported. 79
21.9. LabVIEW/C++/Java API: `ConfigPotentiometerTurns()` is not supported. 79
21.10. Java: Once a Limit Switch is overridden, they can’t be un-overridden. 79
21.11. LabVIEW: Modifying status frame rate is not available. 79
21.12. LabVIEW: Modifying control frame rate is not available.....79
21.13. Firmware 1.1: After selecting “Analog Encoder”, “Sensor Position” does not reliably decode when sensor wraps around (3.3V => 0V).....79
21.14. LabVIEW: Certain SRX VI's running in parallel can affect the GET PID VI signals. 80
21.15. C++: There is no method to reverse the output of a slave Talon SRX. 81

21.16. Firmware <0.36: Limit Switch Faults and Soft Limit Faults may cause Talon SRX to disable for approximately two seconds during the “first time”. 82

21.17. Firmware 1.4: When setting the “Sensor Position” of an analog encoder, multiple set commands are required. 83

22. CRF Firmware Revision Information....84

23. Document Revision Information 85

1. CAN bus Device Basics

Talon SRX, when used with CAN bus, has similar functional requirements with other FRC supported CAN devices. Specifically every Talon SRX requires a unique device ID for typical FRC use (settings, control and status). The device ID is usually expressed as a number between ‘0’ and ‘62’, allowing use for up to 63 Talon SRXs at once. This range does not intercept with device IDs of other CAN device types. For example, there is no harm in having a Pneumatics Control Module (PCM) and a Talon SRX both with device ID ‘0’. However having two Talon SRXs with device ID ‘0’ will be problematic.

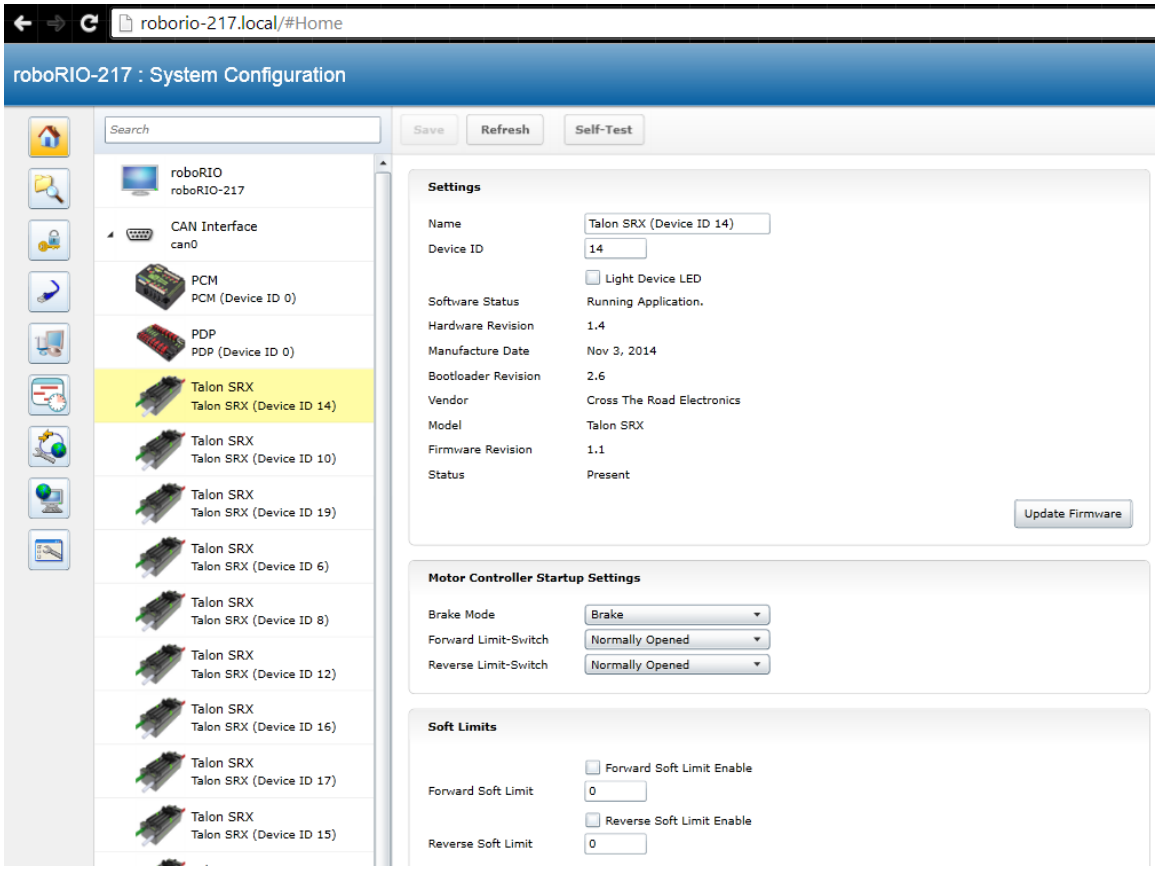
Talon SRXs are field upgradable, and the firmware shipped with your Talon SRX will predate the “latest and greatest” tested firmware intended for FRC use. Firmware update can be done easily using the FRC 2015 roboRIO Web-based Configuration.

Talon SRX provides two pairs of twisted CANH (yellow) and CANL (green) allowing for daisy chaining. Unlike previous seasons, the CAN termination resistors are built into the FRC robot controller (roboRIO) and in the Power Distribution Panel (PDP) assuming the PDP’s termination jumper is in the ON position.

More information on wiring and hardware requirements can be found in the **Talon SRX User’s Guide**.

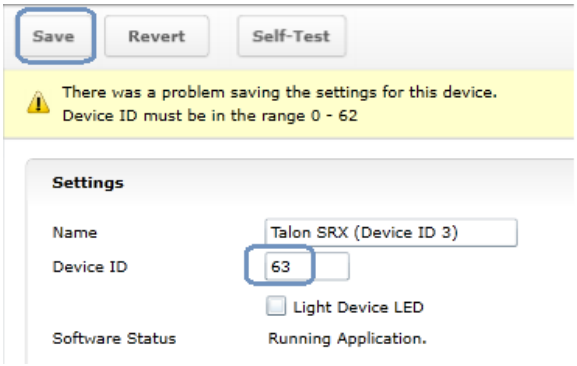
2. roboRIO Web-based Configuration: Firmware and diagnostics

A new diagnostic feature in the 2015 FRC Season is the roboRIO's Web-based Configuration and Monitoring page. This provides diagnostic information on all discovered CAN devices, including Talon SRXs. Talon SRXs can also be field-upgraded using this interface. This feature is accessible by entering the mDNS name of your robot in a web browser, typically "roborio-XXXX.local" where XXXX is the team number (no leading zeros for three digit team numbers).



2.1. Device ID ranges

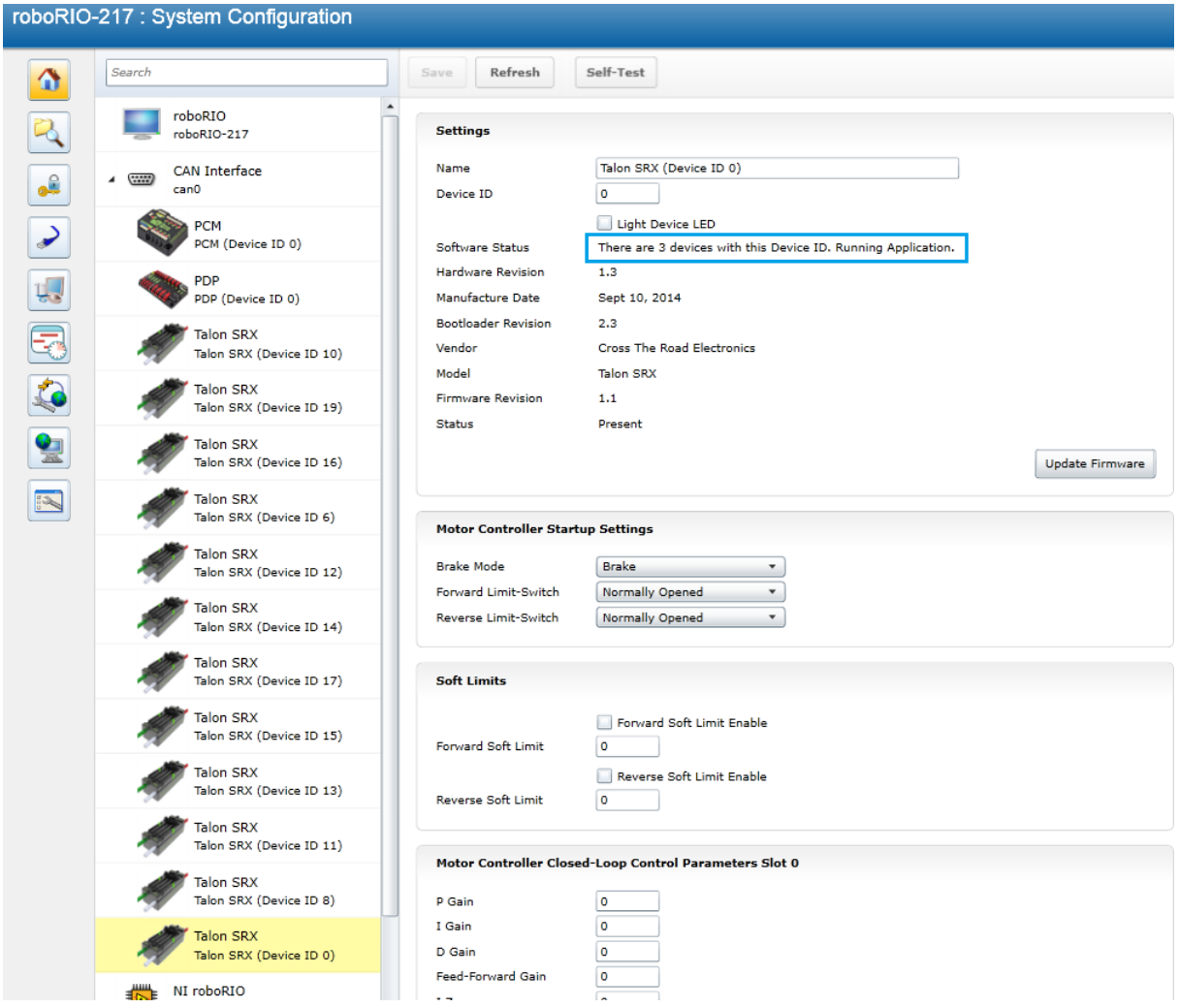
A Talon SRX can have a device ID from 0 to 62. 63 is reserved for broadcast.
If you select an invalid ID you will get an immediate prompt.



2.2. Common ID Talons

During initial setup (and when making changes to your robot), there may be occasions where the CAN bus contains multiple running Talon SRXs with the same device ID. "Common ID" Talon SRXs are to be avoided since they prevent reliable communication and prevents your robot application from being able to distinguish one Talon SRX from another. However the roboRIO's Web-based Configuration and Talon SRX firmware is designed to be tolerant of this problem condition to a degree.

In the event there are "common ID" Talons, they will reveal themselves as a single tree element (see image below). In this example, there is only one "Talon SRX (Device ID 0)" graphical element on the left, however the software status shows that there are three detected Talon SRXs with that device ID. If the number of "common ID" Talon SRXs is small (typically five or less) you will still be able to firmware update, modify settings, and change the device ID. This makes solving device ID contentions possible without having to isolate/disconnect "common ID" Talon SRXs.



When “common ID” Talon SRXs are present, correct this condition by changing the device ID to a “free” number, (one not already in use) before doing anything else. Then manually refresh the browser. This allows the web page to re-populate the left tree view with a new device ID.

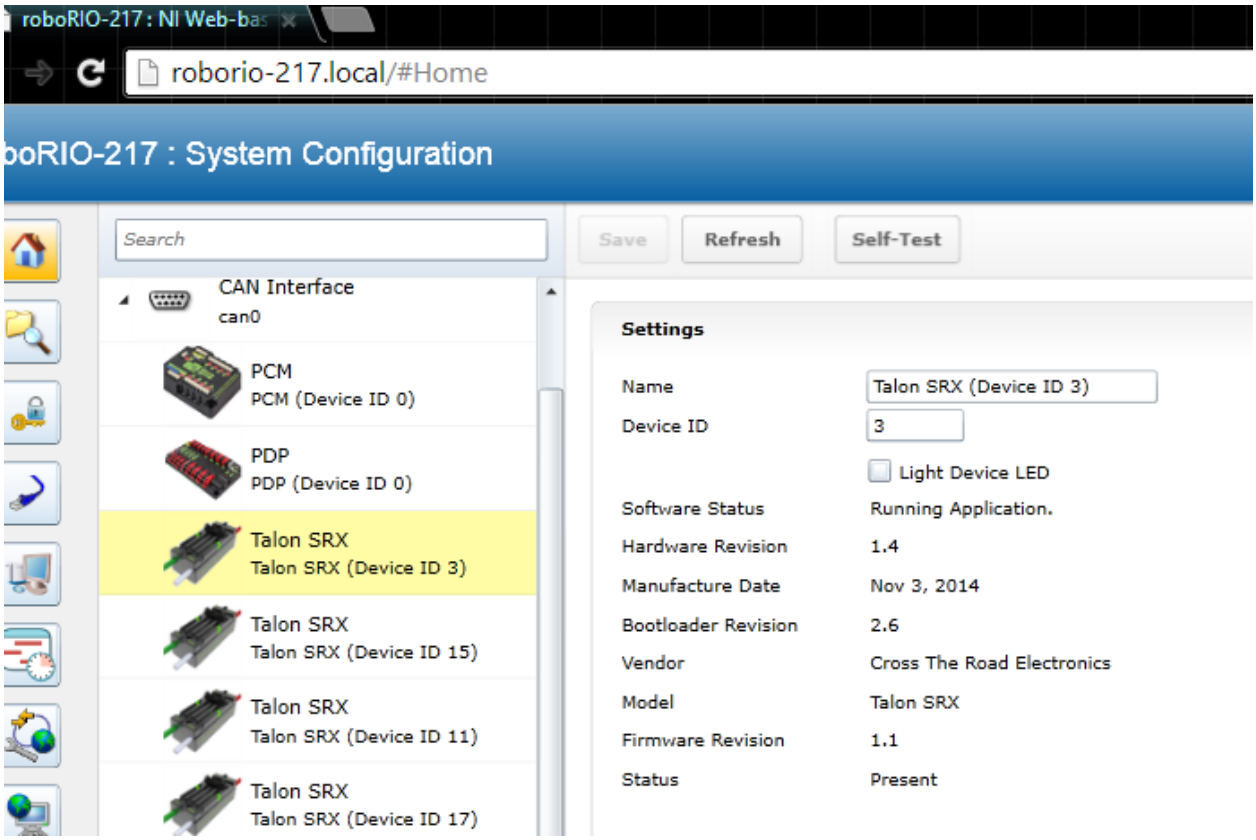
Since the web page allows control of one Talon SRX at a time, you may need to determine *which* “common ID” Talon SRX you are modifying. Checking the “Light Device LED” and pressing “Save” can be used to identify *which* physical Talon SRX is selected, and therefore which one will be modified. This will cause the selected Talon SRX to blink its LEDs uniquely (fast orange blink) for easy identification. In the unlikely event the device is in boot-loader (orange/green LED), it will still respond to this by increasing the blink rate of the orange/green pattern. The “Light Device LED” will uncheck itself after pressing “Save”.

Tip : Since the default device ID of an “out of the box” Talon SRX is device ID ‘0’, when you setup your robot for the first time, start assigning device IDs at ’1’. That way you can, at any time, add another default Talon to your bus and easily identify it.

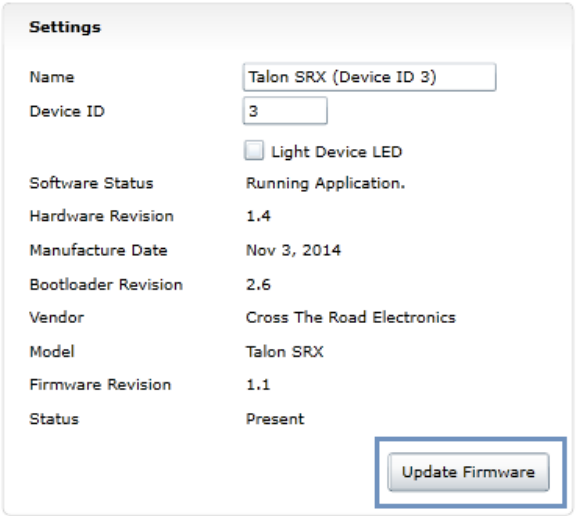
2.3. Firmware Field-upgrade a Talon SRX

Talon SRX uses a file format call CRF. To firmware flash a Talon SRX, navigate to the following page and select it in the left tree view.

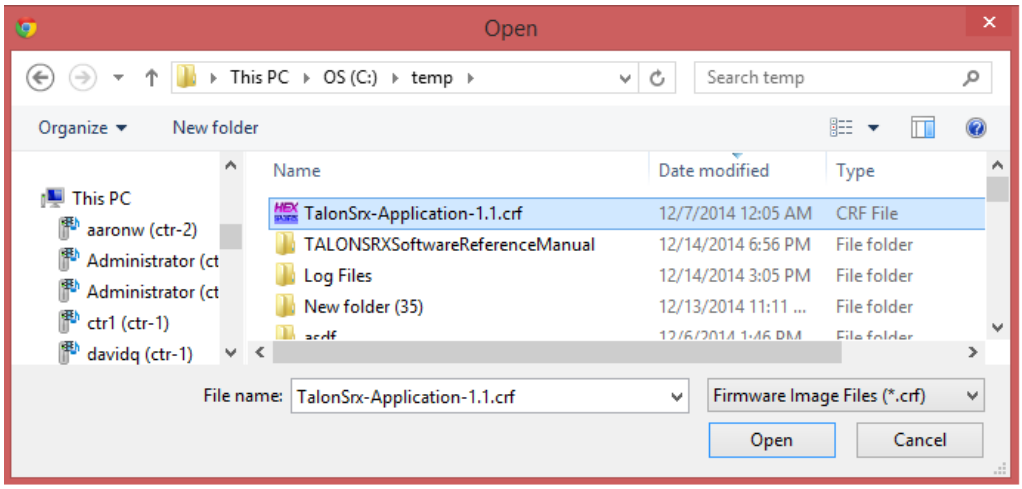
To get the latest firmware files see Section 2.3.4. Where to get CRF files?



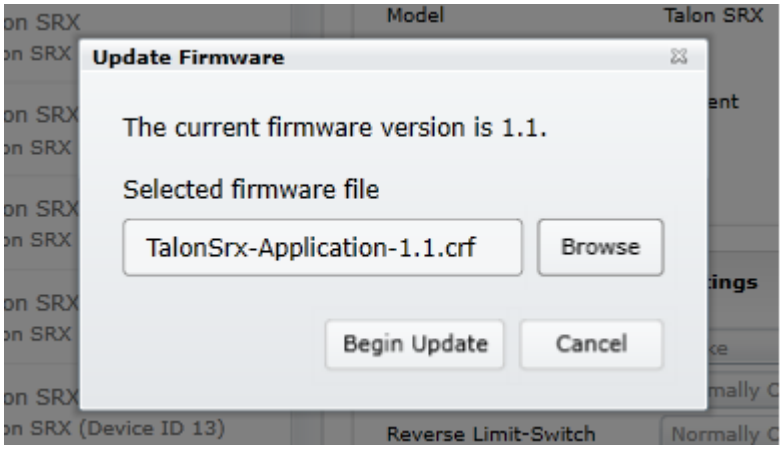
Press “Update Firmware”.



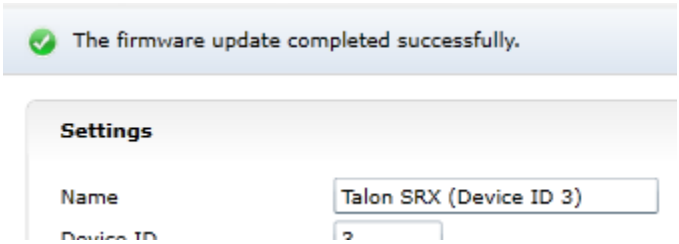
Select the firmware file (*.crf) to flash.



You will be prompted again, press “Begin Update”.



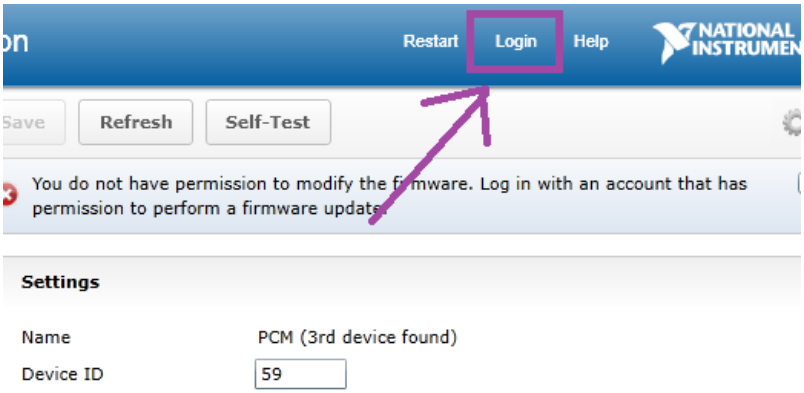
A progress bar will appear and finish with the following prompt. Total time to field-upgrade a Talon SRX is approximately ten seconds. The progress bar will fill quickly, then pause briefly at the near end, this is expected.



2.3.1. When I update firmware, I get “You do not have permissions…”

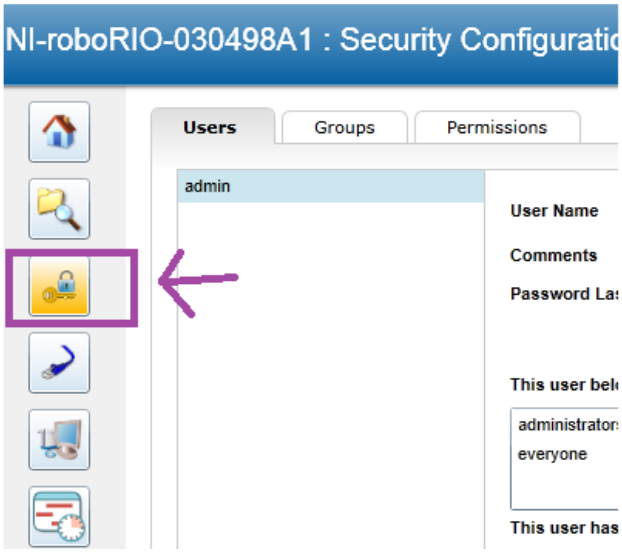
If you get the following error…

...then you have to log into the web interface using the username “admin”.

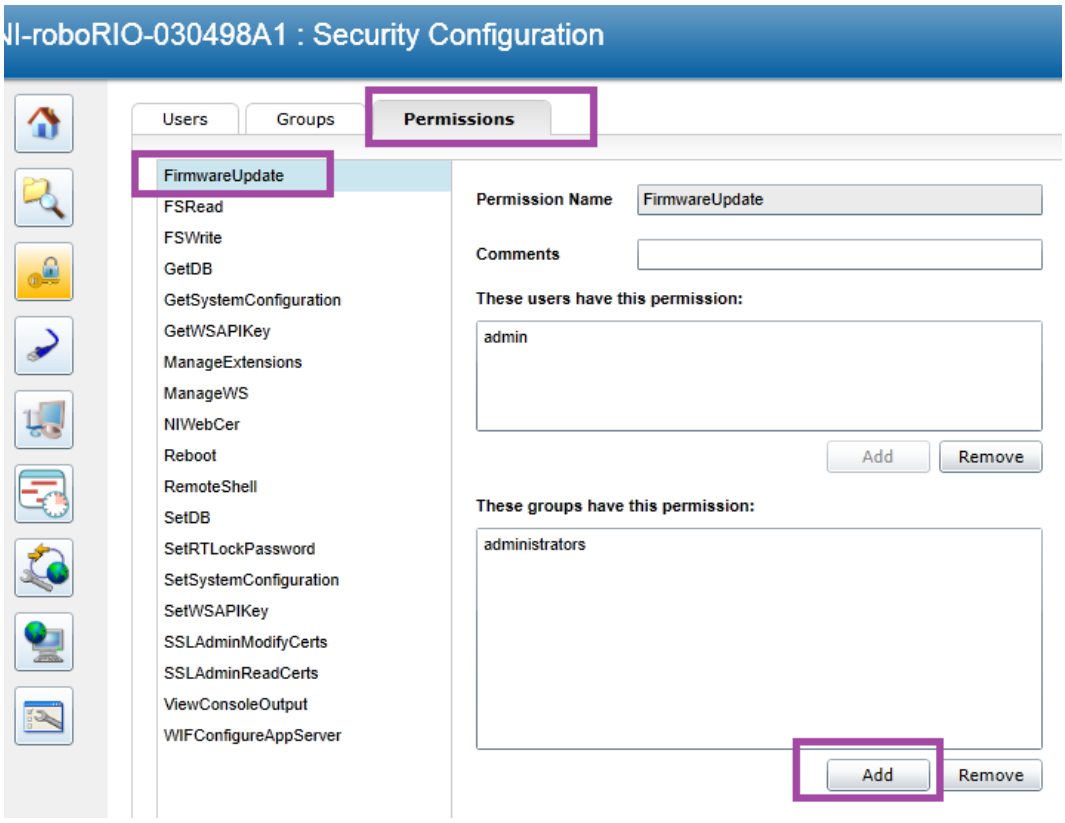


The user name is “admin” and the password is blank “”. Don’t enter any keys for password.

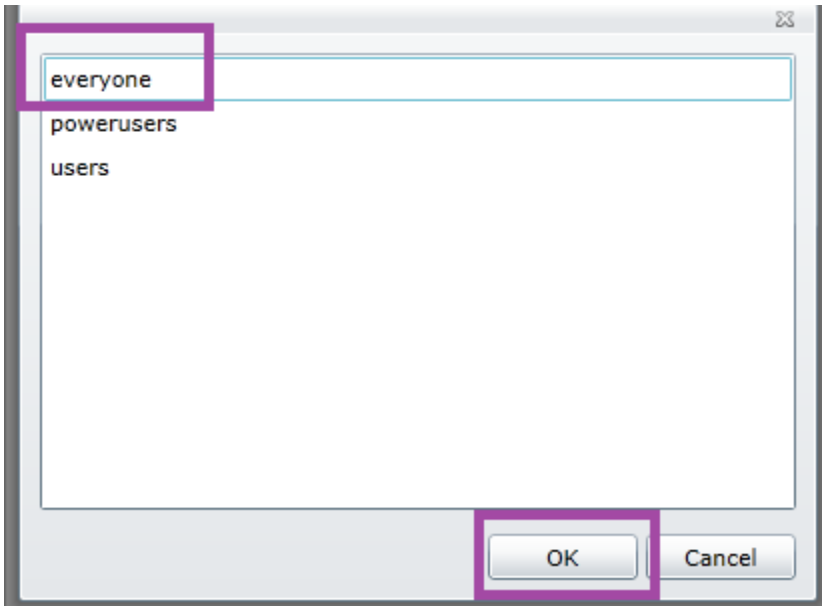
Additionally you can modify permissions to allow field upgrade without being asked for login every single time. If security isn’t a concern then modify the permissions so that “anyone” can access “FirmwareUpdate” features.
Click on the key/lock icon in the left icon list.



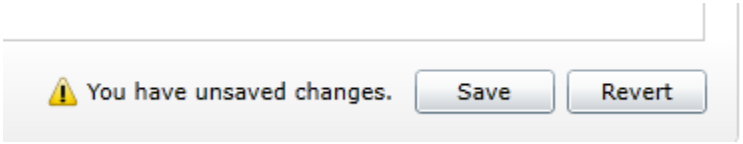
Then click on the “Permissions” tab. Select “FirmwareUpdate”, then press “Add” button.



Select everyone, then OK.



Click “Save” to save changes.

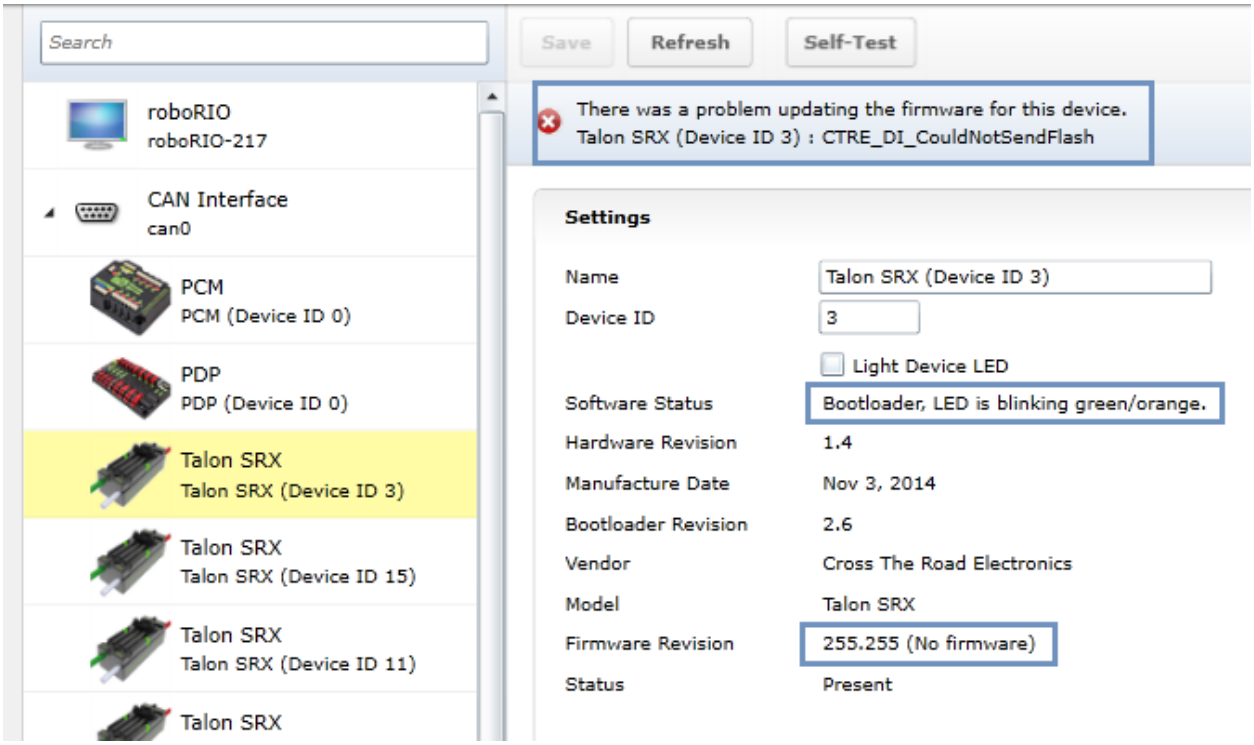


2.3.2. What if Firmware Field-upgrade is interrupted?

Since ten seconds is plenty of time for power or CAN bus to be disconnected, it is always possible for a field-update to be interrupted. An error code will be reported if the firmware field-update is interrupted or fails. Additionally the Software Status will report “Bootloader” and Firmware Revision will be 255.255 (blank).

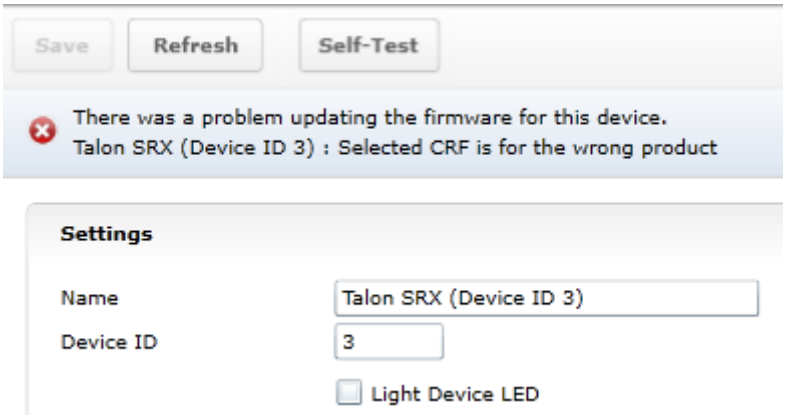
If a Talon SRX has no firmware, its boot-loader will take over and blink green/yellow on the device’s corresponding LED. It will also keep its device ID, so the roboRIO can still be used to change the device ID or (re)flash a new application firmware (crf). This means you can reattempt field-upgrade using the same web interface. There is no need for any sort of recovery steps, nor is it necessary to isolate no-firmware Talon SRXs.

Example capture of disconnecting the CAN bus in the middle of a firmware-upgrade…

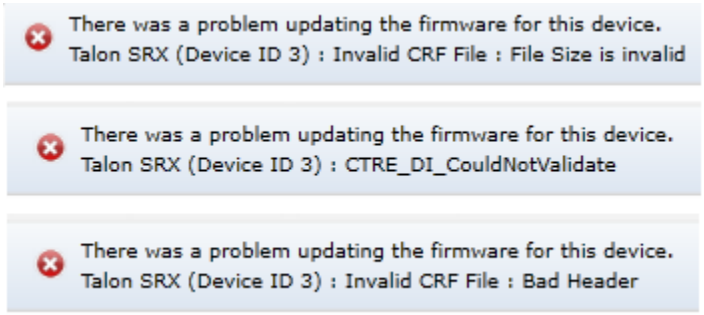


2.3.3. Other Field-upgrade Failure Modes

Here's an example error when trying to flash the wrong CRF into the wrong product. The device will harmlessly stay in boot-loader, ready to be (re)flashed again.

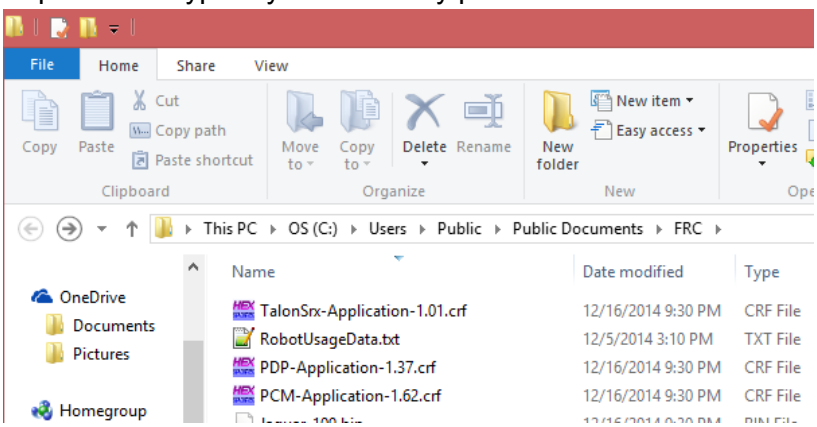


Here's what to expect if your CRF file is corrupted (different errors depending on where the file is corrupted). The device will harmlessly stay in boot-loader, ready to be (re)flashed again. Re-downloading the CRF firmware file is recommended if this is occurring persistently.



2.3.4. Where to get CRF files?

The FRC Software installer will create a directory with various firmware files/tools for many control system components. Typically the directory path is “C:\Users\Public\Documents\FRC”.



When the path is entered into a browser, the browser may fix-up the path into “C:\Users\Public\Public Documents\FRC”.

In this directory are the initial release firmware CRF files for all CTRE CAN bus devices, including the Talon SRX.

Additionally newer updates may be provided online at www.crosstheroadelectronics.com and www.vex.com.

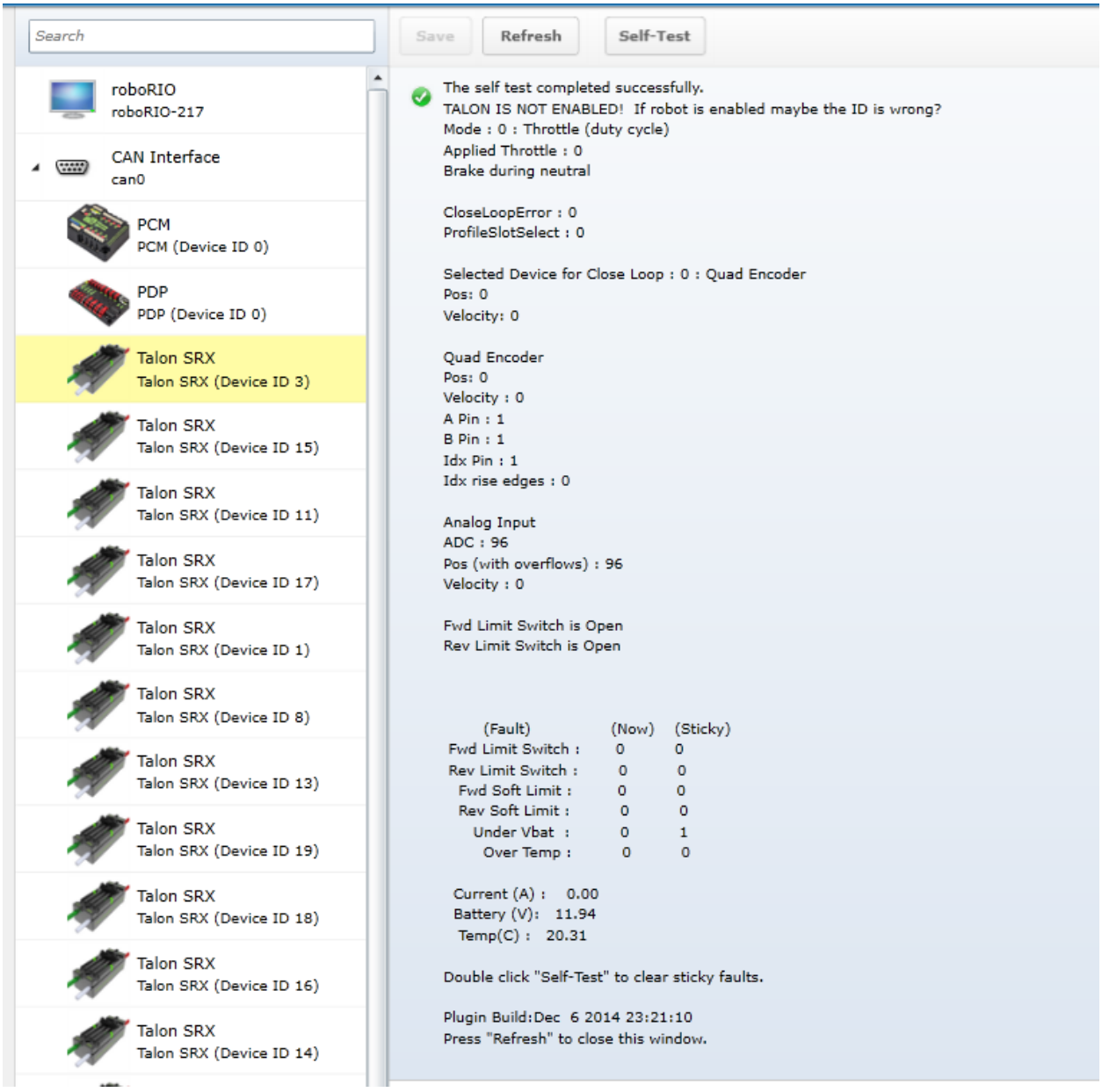
The latest firmware to be used at time of writing is version 1.1 (or newer).

2.4. Self-Test

Pressing Self-Test will display data captured from CAN bus at the time of press. This can include fault states, sensor inputs, output states, measured battery voltage, etc…

At the bottom of the Self-Test results, the build time of the library that implements web-based CAN features is also present.

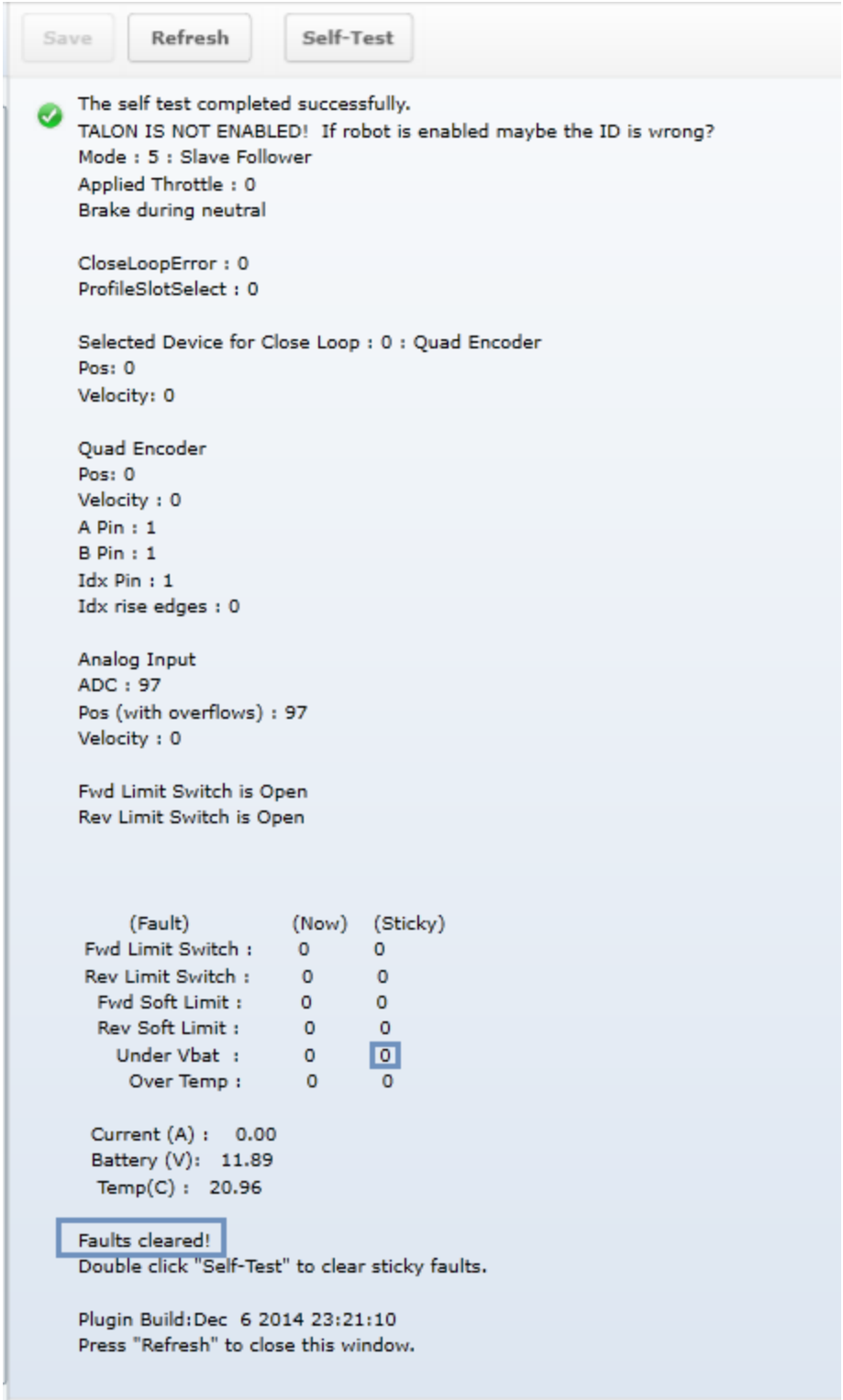
Here's an example of pressing "Self-Test" with Talon SRX. Be sure to check if Talon SRX is ENABLED or DISABLED. If Talon SRX is DISABLED then either the robot is disabled or the robot application has not yet created a Talon SRX object (see Section 3. Creating a Talon SRX Object (and basic drive)).



After enabling the robot and repressing “Self-Test” we see the Talon SRX is enabled.
Additionally we see there is a sticky fault asserted for low battery voltage.
Sticky faults persist across power cycles for identifying intermittent problems after they occur.

2.4.1. Clearing Sticky Faults

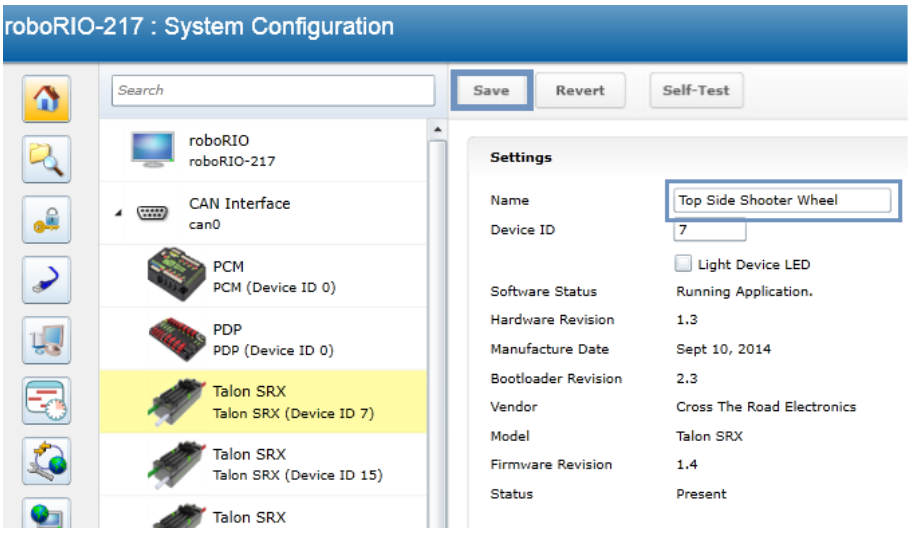
After double clicking Self-Test in a rapid fashion we see our fault gets cleared.



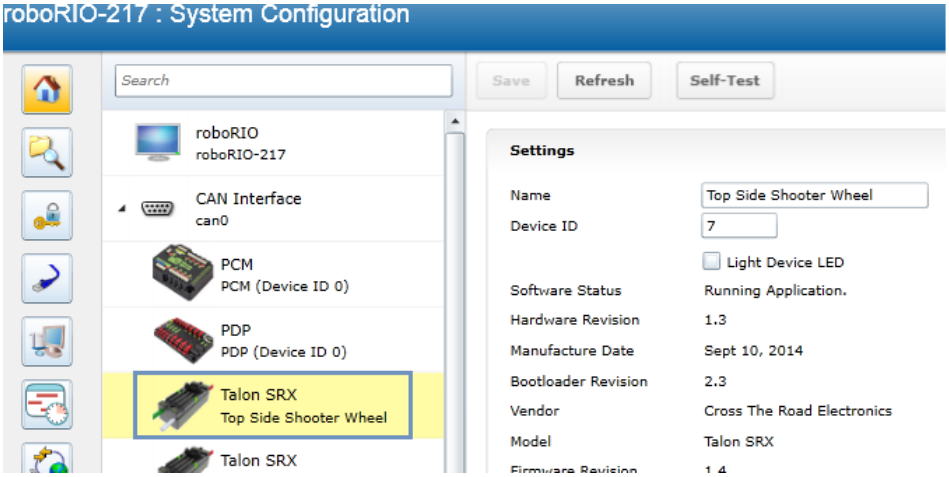
2.5. Custom Names

Another feature made available by the Web-based Configuration is the ability to rename Talon SRXs with custom string descriptions. A Talon SRX's custom name is saved persistently inside the Talon. To modify the default name highlight the contents of the "Name" text entry.

…then replace with a custom text description and press "Save".

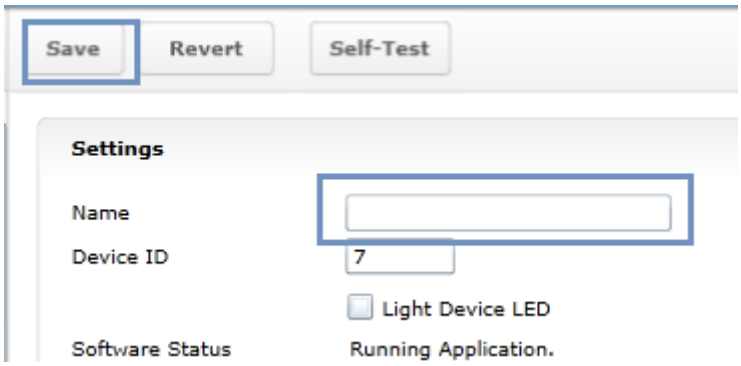


The new description will appear in the left tree view.

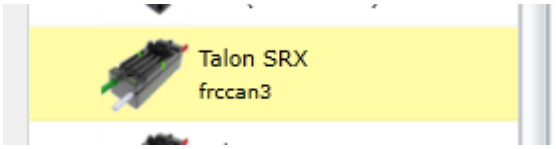


2.5.1. Re-default custom name

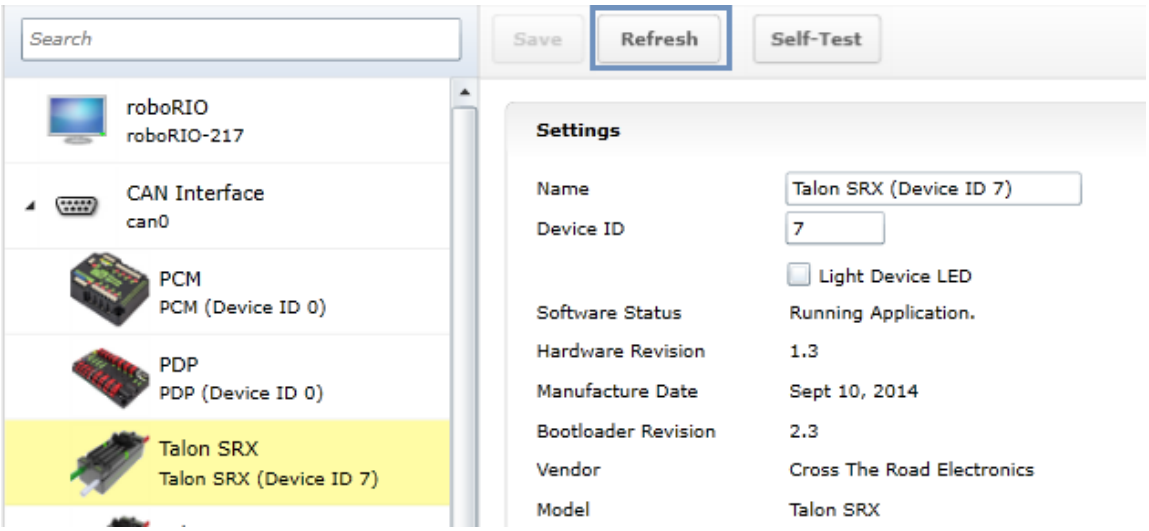
To re-default the custom name, clear the “Name” text entry and press “Save”.



Left tree view will update with a temporary name until the “Refresh” button is pressed.



After pressing “Refresh” the default name will appear.



3. Creating a Talon Object (and basic drive)

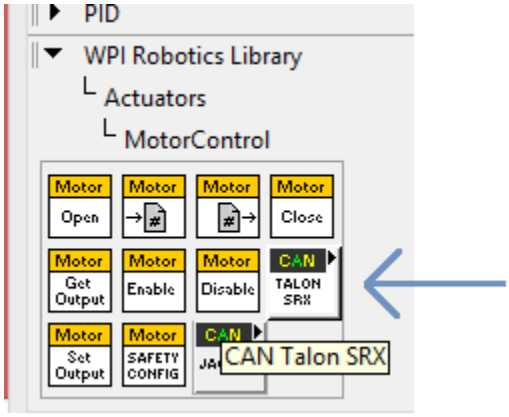
3.1. Programming API and Device ID

Regardless of what language you use on the FRC control system (LabVIEW/C++/Java), the method for specifying which Talon SRX you are programmatically controlling is the device ID. Although the roboRIO Web-based Configuration is tolerant of "common ID" Talon SRXs to a point, the robot programming API will not enable/control "common ID" Talons reliably. For the robot to function properly, there CANNOT BE "COMMON ID" Talon SRXs. See Section 2.2. Common ID Talons for more information.

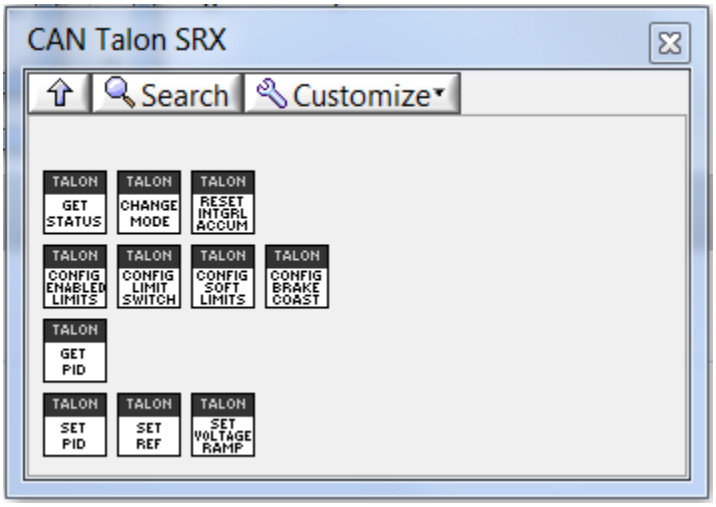
3.2. New Classes/Virtual Instruments

C++/Java now contains a new class **CANTalon** (.h/.cpp/.java).

LabVIEW contains three new motor types for the FRC 2015 season: Talon SRX, Victor SP, CAN Talon SRX. When using Talon SRX on CAN bus, select the **CAN Talon SRX**. The other two modes are for PWM use. Additionally new VIs are available in the CAN Talon SRX palette.

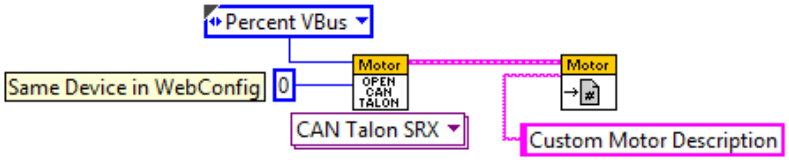


After selecting CAN Talon SRX, the new VIs are visible...



3.3. LabVIEW

Creating a “bare-bones” Talon SRX object is similar to previously supported motor controllers. Start by creating a `WPI_MotorControlOpen.vi` object (left) and a `WPI_MotorControlRefNum.vi` object (right).

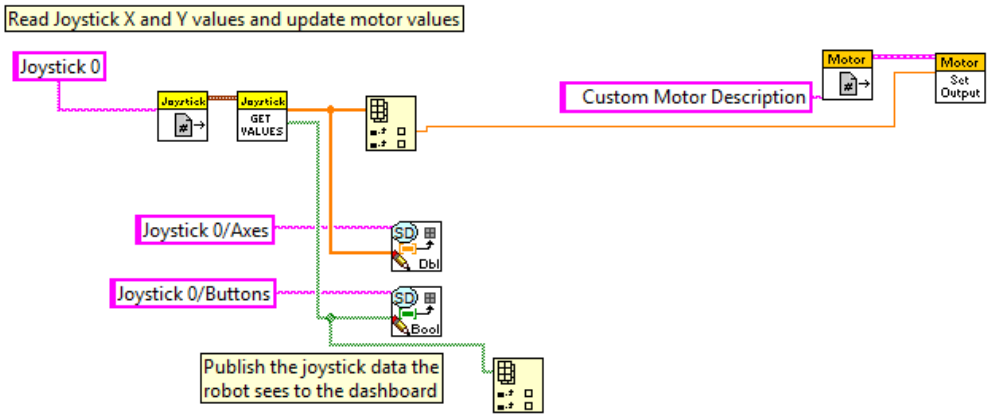


These are accessible in the actuator palette (same VIs as previous seasons). Just like with other motor controllers, the `WPI_MotorControlOpen.vi` has a dropdown to select the motor controller type.

To begin with basic control, select “CAN Talon SRX”. The text underneath “Motor” will then change to ““OPEN CAN TALON”. Then create two constants for the “Device Number” and “Control Mode” inputs. The control mode will default to “Percent VBus” and “0” for the Device ID. Enter the appropriate Device ID that was selected in the roboRIO Web-based Configuration.

Also similarly to other motor controllers, you may register a custom string reference using `WPI_MotorControlRefNum.vi` to reference the motor controller by description in other block diagrams. In this example we use “Custom Motor Description”.

Setting the output value of the Talon SRX is done similarly to other motor controllers.
In this example we directly control the motor output with a Joystick axis.
When using a closed-loop mode, the `Set Output VI` is also the method for specifying the set-point.



3.4. C++

When using a script language, the API class to support Talon SRX is called **CANTalon** (.cpp/.h/.java). When the object is constructed, the device ID is the first parameter. There may be an optional second parameter to change the frequency at which the Talon SRX is updated over CAN (default 10ms).

```
#include "WPILib.h"

class Robot: public IterativeRobot
{
    Joystick joy;
    CANTalon customMotorDescrip;

public:
    Robot() : joy(0),   /* gamepad at the first slot */
            customMotorDescrip(0)  /* device ID '0', match the RIO Web Config Page */
    {
    }

    void TeleopPeriodic()
    {
        double leftAxis = joy.GetY(Joystick::kLeftHand);

        customMotorDescrip.Set(leftAxis);
    }

};

START_ROBOT_CLASS(Robot);

```

3.5. Java

When a **CANTalon** object is constructed in Java, the device ID is the first parameter. There may be an optional second parameter to change the frequency at which the Talon SRX is updated over CAN (default 10ms).

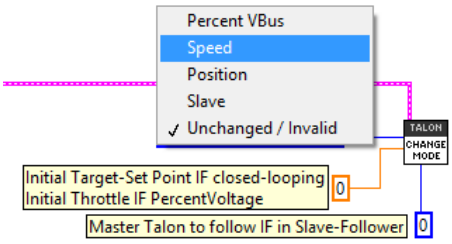
```
package org.usfirst.frc.team217.robot;
import edu.wpi.first.wpilibj.CANTalon;
import edu.wpi.first.wpilibj.IterativeRobot;
import edu.wpi.first.wpilibj.Joystick;
import org.usfirst.frc.team217.robot.subsystems.ExampleSubsystem;

/**
 * The VM is configured to automatically run this class, and to call the
 * functions corresponding to each mode, as described in the IterativeRobot
 * documentation. If you change the name of this class or the package after
 * creating this project, you must also update the manifest file in the resource
 * directory.
 */
public class Robot extends IterativeRobot {

    public static final ExampleSubsystem exampleSubsystem = new ExampleSubsystem();

    Joystick joy = new Joystick(0);                /* gamepad at the first slot */
    CANTalon customMotorDescrip = new CANTalon(0);  /* device ID '0', match the RIO Web Config Page */


    /**
     * This function is called periodically during operator control
     */
    public void teleopPeriodic() {

        double axis = joy.getY();

        customMotorDescrip.set(axis);
    }
}
```

3.6 Changing Mode

After a Talon software object is created, the Talon SRX mode can be changed from the default Percent Vbus (open loop throttle) to the other supported modes programmatically. Additionally the LabVIEW `OPEN CAN TALON VI` also allows caller to select the initial control mode.

3.6.1. LabVIEW

The `CHANGE MODE VI` can be used to change the Talon SRX mode, and set the first target set point, throttle, or Talon Master ID to follow.



3.6.2. C++

The function `SetControlMode()` can be used to change the Talon SRX mode. Caller should ensure `Set()` is called immediately after to properly set the initial target set point, throttle, or Master ID to follow.

```
/* Possible modes to choose from.  Call Set() immediately after changing mode to set the target-set-point/throttle/ or Master Talon ID */
customMotorDescrip.SetControlMode(CANSpeedController::kPercentVbus);    /* direct throttle control, Set() controls drive */
customMotorDescrip.SetControlMode(CANSpeedController::kFollower);       /* follow another Talon, Set() determines Talon to follow. */
customMotorDescrip.SetControlMode(CANSpeedController::kSpeed);          /* Speed Closed-Loop, Set() controls set point */
customMotorDescrip.SetControlMode(CANSpeedController::kPosition);       /* Position Closed-Loop, Set() controls set point */
```

3.6.3. Java

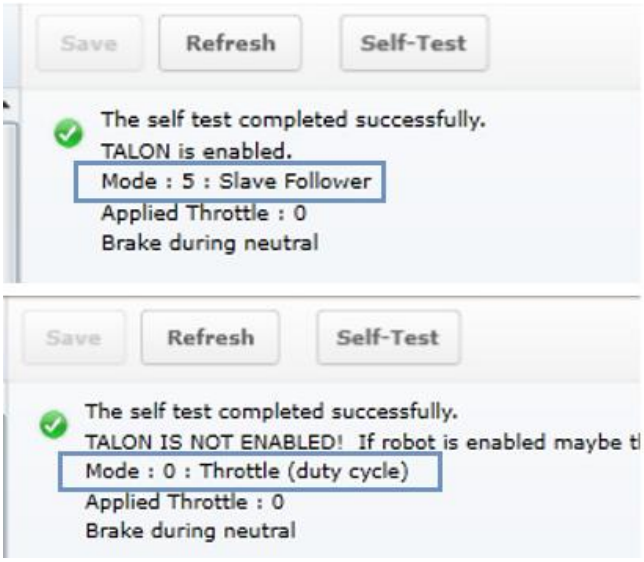
The function `changeControlMode()` can be used to change the Talon SRX mode. Caller should ensure `set()` is called immediately after to properly set the initial target set point, throttle, or Master ID to follow.

```
/* Possible modes to choose from.  Call set() immediately after changing mode to set the target-set-point/throttle/ or Master Talon ID */
customMotorDescrip.changeControlMode(ControlMode.PercentVbus);  /* direct throttle control, set() controls drive */
customMotorDescrip.changeControlMode(ControlMode.Follower);     /* follow another Talon, set() determines Talon to follow. */
customMotorDescrip.changeControlMode(ControlMode.Speed);           /* Speed Closed-Loop, set() controls set point */
customMotorDescrip.changeControlMode(ControlMode.Position);     /* Position Closed-Loop, set() controls set point */
```

3.6.4. Check Control Mode with Self-Test

The Self-Test can be used to confirm the desired mode of the Talon SRX (Throttle, Slave, Position Closed-Loop, and Velocity Closed-Loop). However note that the Talon SRX mode will not update until robot is enabled.

Example Self-Test



4. Limit Switch and Neutral Brake Mode

4.1. Default Settings

An “out of the box” Talon will default with the limit switch setting of “Normally Open” for both forward and reverse. This means that motor drive is allowed when a limit switch input is not closed (i.e. not connected to ground). When a limit switch input is closed (is connected to ground) the Talon SRX will disable motor drive and individually blink both LEDs red in the direction of the fault (red blink pattern will move towards the M+/white wire for positive limit fault, and towards M-/green wire for negative limit fault).

An “out of the box” Talon SRX will typically have a default brake setting of “Brake during neutral”. The B/C CALL button will be illuminated red (brake enabled).

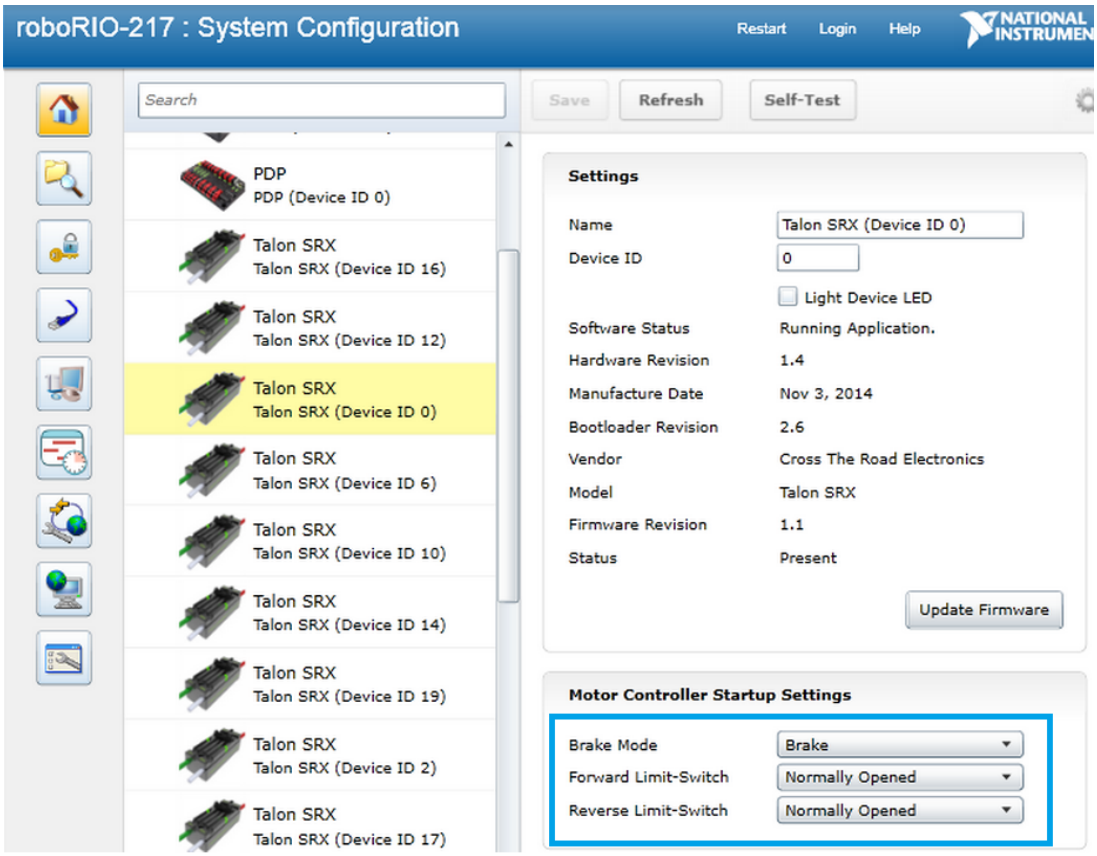
Since an “out of the box” Talon will likely not be connected to limit switches (at least not initially) and because limit switch inputs are internally pulled high (i.e. the switch is open), the limit switch feature is default to “normally open”. This ensures an “out of the box” Talon will drive even if no limit switches are connected.

For more information on Limit Switch wiring/setup, see the **Talon SRX User’s Guide**.

Forward Limit Switch Mode	Limit Switch NO pin	Limit Switch NC pin	Limit Switch COM pin	Motor Drive Switch open Fwd. throttle	Motor Drive Switch closed Fwd. throttle	*Voltage (Switch Open)	*Voltage (Switch Closed)
Normally Open	pin4	N.A.	pin10	Y	N	~2.5V	0 V
Normally Closed	N.A.	pin4	pin10	N	Y	0 V	~2.5V
Disabled	N.A.	N.A.	N.A.	Y	Y	N.A.	N.A.
Reverse Limit Switch Mode	Limit Switch NO pin	Limit Switch NC pin	Limit Switch COM pin	Motor Drive Switch open Rev. throttle	Motor Drive Switch closed Rev. throttle	*Voltage (Switch Open)	*Voltage (Switch Closed)
Normally Open	pin8	N.A.	pin10	Y	N	~2.5V	0 V
Normally Closed	N.A.	pin8	pin10	N	Y	0 V	~2.5V
Disabled	N.A.	N.A.	N.A.	Y	Y	N.A.	N.A.
	*Measured voltage at the Talon SRX Limit Switch Input pin.						
	Limit Switch Input Forward Input - pin4 on Talon SRX						
	Limit Switch Input Reverse Input - pin8 on Talon SRX						
	Limit Switch Ground - pin10 on Talon SRX						

4.2. roboRIO Web-based Configuration: Limit Switch and Brake

Limit switch features can be disabled or changed to “Normally Closed” in the roboRIO Web-based Configuration. Similarly the brake mode can be change through the same interface.



Changing the settings will take effect once the “Save” button is pressed. The settings are saved in persistent memory.

If the Brake or Limit Switch mode is changed in the roboRIO Web-based Configuration, the Talon SRX will momentarily disable then resume motor drive. All other settings can be changed without impacting the motor drive or enabled-state of the Talon SRX.

Additionally the brake mode can be modified by pressing the B/C CAL Button on the Talon SRX itself, just like with previous generation Talons.

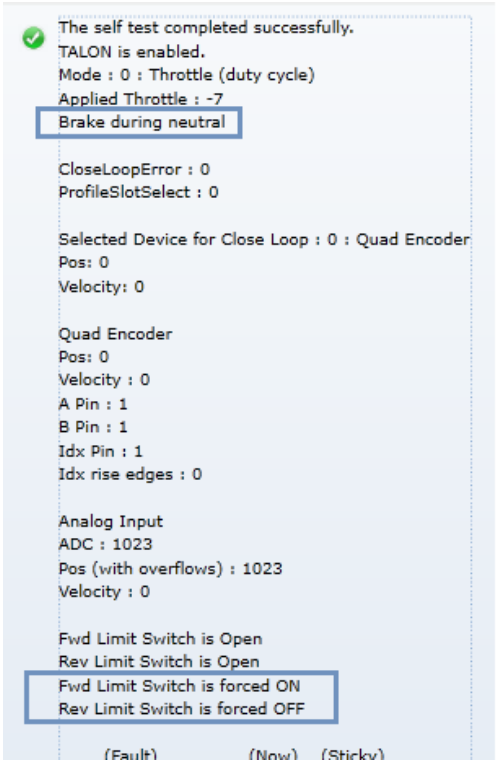
4.3. Overriding Brake and Limit Switch with API

The Brake and Limit Switch can, to a degree, be changed programmatically (during a match). A great example of this would be for dynamic braking.

The programming API allows for overriding the active neutral brake mode. When this is done the Brake/Coast LED will reflect the overridden value (illuminated red for brake, off for coast) regardless of the startup brake mode specified in the roboRIO Web-based Configuration (i.e. what's saved in persistent memory).

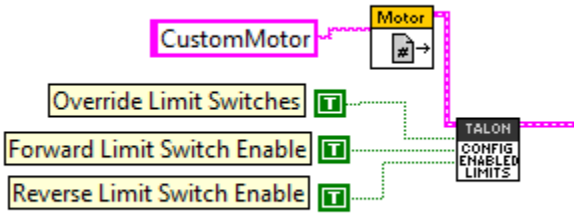
Similarly the enabled states of the limit switches (on/off) for the forward and reverse direction can be individually enabled/disabled by overriding them with programming API.

The brake and limit switch overrides can be confirmed in the Self-Test results. If limit switches are overridden by the robot application, the forced states are displayed as "forced ON" or "forced OFF". Also the currently active brake mode is in the Self-Test results.

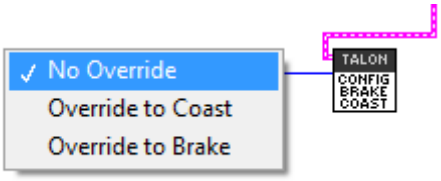


4.3.1. LabVIEW

The `CONFIG ENABLED LIMITS VI` can be used to override the limit switch enable states. When overriding the limit switch enable states, set the override signal to true, then pass true/false to the forward and reverse limit switch enable.



The neutral brake mode can also be overridden to Brake or Coast using the `CONFIG BRAKE COAST VI`. If “No Override” is selected then the Startup Brake Mode is used.



4.3.2. C++

Limit Switches can be forced on or off using `ConfigLimitMode()`, along with soft limits (see Section 8. Soft Limits).

```
customMotorDescrip.ConfigLimitMode(CANSpeedController::kLimitMode_SwitchInputsOnly);        /* limit switches only */
customMotorDescrip.ConfigLimitMode(CANSpeedController::kLimitMode_SoftPositionLimits);      /* limits switches and soft-limits */
customMotorDescrip.ConfigLimitMode(CANSpeedController::kLimitMode_DisableSwitchInputs); /* disabled limit switches and disable soft-limits */
```

`ConfigNeutralMode()` can be used to override the brake/coast mode. Also selecting the enumerated value of `kNeutralMode_Jumper` will signal the Talon SRX to use its default setting (controlled by roboRIO Web-based Configuration and B/C CAL button).

```
customMotorDescrip.ConfigNeutralMode(CANSpeedController::NeutralMode::kNeutralMode_Brake); /* override to brake during neutral */
customMotorDescrip.ConfigNeutralMode(CANSpeedController::NeutralMode::kNeutralMode_Coast); /* override to coast during neutral */
customMotorDescrip.ConfigNeutralMode(CANSpeedController::NeutralMode::kNeutralMode_Jumper);/* default to flash setting - No Override */
```

4.3.3. Java

`enableLimitSwitch()` can be used to override the enabled state for forward and reverse limit switch enable. `enableBrakeMode()` can be used to override the brake/coast setting.

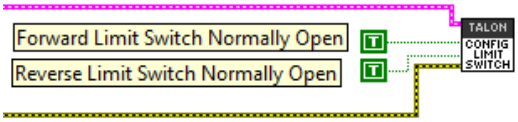
```
customMotorDescrip.enableLimitSwitch(true,true);    /* forward enable, reverse enable */
customMotorDescrip.enableBrakeMode(true);           /*when in neutral : true for brake, false for coast */
```

4.4. Changing limit switch mode between “Normally Open” or “Normally Closed”

The limit switch setting that determines “Normally Open” vs “Normally Closed” can also be set programmatically using the `CONFIG LIMIT SWITCH VI`. However care should be taken when this is done. When a Talon SRX’s limit switch mode is changed from its current setting to a different value, it briefly disables motor drive during the transition. This should not be a problem since the limit switches in the robot are typically not changed during a match.

However it may be convenient to ensure NO/NC settings at startup (particularly when using the non-default setting of Normally Closed) so as to avoid needing to use the roboRIO Web-based Configuration to select NC whenever a Talon SRX needs to be added/replaced.

4.4.1. LabVIEW



4.4.2. C++

`ConfigFwdLimitSwitchNormallyOpen()` and `ConfigRevLimitSwitchNormallyOpen()` can be used to change the NO/NC state of a limit switch input.

```
if(btn4){
    customMotorDescrip.ConfigFwdLimitSwitchNormallyOpen(true); /* forward limit set to Normally Open */
}else if(btn2){
    customMotorDescrip.ConfigFwdLimitSwitchNormallyOpen(false);/* forward limit set to Normally Closed */
}

if(btn1){
    customMotorDescrip.ConfigRevLimitSwitchNormallyOpen(true); /* reverse limit set to Normally Open */
}else if(btn3){
    customMotorDescrip.ConfigRevLimitSwitchNormallyOpen(false);/* reverse limit set to Normally Closed */
}
```

4.4.3. Java

`ConfigFwdLimitSwitchNormallyOpen()` and `ConfigRevLimitSwitchNormallyOpen()` can be used to change the NO/NC state of a limit switch input.

```
if(btn4){
    customMotorDescrip.ConfigFwdLimitSwitchNormallyOpen(true); /* forward limit set to Normally Open */
}else if(btn2){
    customMotorDescrip.ConfigFwdLimitSwitchNormallyOpen(false);/* forward limit set to Normally Closed */
}

if(btn1){
    customMotorDescrip.ConfigRevLimitSwitchNormallyOpen(true); /* reverse limit set to Normally Open */
}else if(btn3){
    customMotorDescrip.ConfigRevLimitSwitchNormallyOpen(false);/* reverse limit set to Normally Closed */
}
```

5. Getting Status and Signals

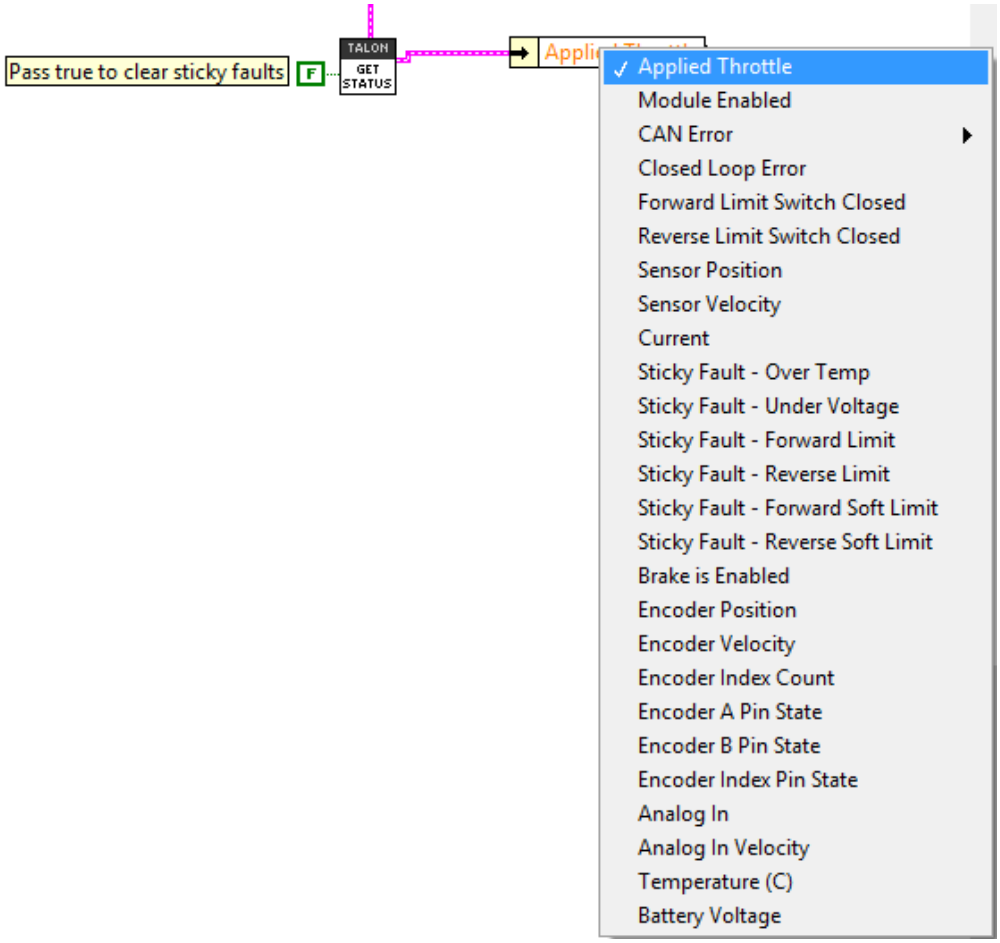
The Talon SRX transmits most of its status signals periodically, i.e. in an unsolicited fashion. This improves bus efficiency by removing the need for “request” frames, and guarantees the signals necessary for the wide range of use cases Talon supports, are available.

These signals are available in API regardless of what control mode the Talon SRX is in. Additionally the signals can be polled in the roboRIO Web-based Configuration (see Section 2.4. Self-Test).

Included in the list of signals are…
- Quadrature Encoder Position, Velocity, Index Rise Count, Pin States (A, B, Index)
- Analog-In Position, Analog-In Velocity, 10bit ADC Value,
- Battery Voltage, Current, Temperature
- Fault states, sticky fault states,
- Limit switch pin states
- Applied Throttle (duty cycle) regardless of control mode.
- Applied Control mode: Voltage % (duty-cycle), Position/Velocity closed-loop, or slave follower.
- Brake State (coast vs brake)
- Closed-Loop Error, the difference between closed-loop set point and actual position/velocity.
- Sensor Position and Velocity, the signed output of the selected Feedback device (robot must select a Feedback device, or rely on default setting of Quadrature Encoder).

5.1. LabVIEW

The `GET STATUS VI` can be used to retrieve the latest value for the signals Talon SRX periodically transmits. Additionally, sticky faults can be cleared if “true” is passed into the “Clear Sticky Fault” signal. To get a particular signal, unbundle-by-name the output of the `GET STATUS VI`. Then select the signal to get by right clicking the center of the unbundle object.



5.2. C++

All get functions are available in C++. Here are a few examples….

```
double currentAmps = customMotorDescrip.GetOutputCurrent();
double outputV = customMotorDescrip.GetOutputVoltage();
double busV = customMotorDescrip.GetBusVoltage();

int quadEncoderPos = customMotorDescrip.GetEncPosition();
int quadEncoderVelocity = customMotorDescrip.GetEncVel();

int analogPos = customMotorDescrip.GetAnalogIn();
int analogVelocity = customMotorDescrip.GetAnalogInVel();

int selectedSensorPos = customMotorDescrip.GetPosition();
int selectedSensorSpeed = customMotorDescrip.GetSpeed();

int closeLoopErr = customMotorDescrip.GetClosedLoopError();
if(bEverySecond){

    printf("currentAmps:%f\n",currentAmps);
    printf("outputV:%f\n",outputV);
    printf("busV:%f\n\n",busV);

    printf("quadEncoderPos:%i\n",quadEncoderPos);
    printf("quadEncoderVelocity:%i\n\n",quadEncoderVelocity);

    printf("analogPos:%i\n",analogPos);
    printf("analogVelocity:%i\n\n",analogVelocity);

    printf("selectedSensorPos:%i\n",selectedSensorPos);
    printf("selectedSensorSpeed:%i\n\n",selectedSensorSpeed);

    printf("closeLoopErr:%i\n",closeLoopErr);
}
```

5.3. Java

All get functions are available in java. Here are a few examples….

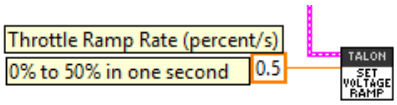
```
double currentAmps = _talons[masterId].getOutputCurrent();
double outputV = _talons[masterId].getOutputVoltage();
double busV = _talons[masterId].getBusVoltage();
double quadEncoderPos = _talons[masterId].getEncPosition();
double quadEncoderVelocity = _talons[masterId].getEncVelocity();
int analogPos = _talons[masterId].getAnalogInPosition();
int analogVelocity = _talons[masterId].getAnalogInVelocity();
double selectedSensorPos = _talons[masterId].getPosition();
double selectedSensorSpeed = _talons[masterId].getSpeed();
int closeLoopErr = _talons[masterId].getClosedLoopError();

if(bEverySecond){
    System.out.println("currentAmps" + currentAmps);
    System.out.println("outputV:" + outputV);
    System.out.println("output%:" + 100*(outputV / busV) );
    System.out.println("busV:" + busV);
    System.out.println("");
    System.out.println("quadEncoderPos:" + quadEncoderPos);
    System.out.println("quadEncoderVelocity:" + quadEncoderVelocity);
    System.out.println("");
    System.out.println("analogPos:" + analogPos);
    System.out.println("analogVelocity:" + analogVelocity);
    System.out.println("");
    System.out.println("selectedSensorPos:" + selectedSensorPos);
    System.out.println("selectedSensorSpeed:" + selectedSensorSpeed);
    System.out.println("");
    System.out.println("closeLoopErr:" + closeLoopErr);
}
```

6. Setting the Ramp Rate

The Talon SRX can be set to honor a ramp rate to prevent instantaneous changes in throttle. This ramp rate is in effect regardless of which mode is selected (throttle, slave, or closed-loop).

6.1. LabVIEW

Use the SET VOLTAGE RAMP to specify the ramp rate in percent per second.



6.2. C++

Ramp can be set in Volts per second using `SetVoltageRampRate()`.

```
customMotorDescrip.SetVoltageRampRate(6.0); /* 0V to 6V in one second */
```

6.3. Java

Ramp can be set in Volts per second using `setVoltageRampRate()`.

```
customMotorDescrip.setVoltageRampRate(6); /* 0V to 6V in one second */
```

6.4. What is the slowest ramp possible?

The Talon SRX internally expresses the (Voltage) Ramp Rate in throttle units per 10ms (see Section 17.6). As a result, at the minimum (slowest) ramp rate, the time from zero-to-full-throttle is 10.23 seconds. This is derived from 1 throttle unit per 10ms. In terms of voltage per second, this is equivalent to 1.173 V per second or 9.77% per second. When choosing an initial ramp rate avoid specifying a rate that is slower than this limitation. Choosing a slower rate than what's possible will cause the programming API to truncate the calculated result to zero throttle units per 10ms, leading to the effect of no ramp at all.

7. Selecting a Feedback Device

Although the analog and quadrature signals are available all the time, the Talon SRX requires the robot application to “pick” a particular “Feedback Device” for soft limit and closed-loop features.

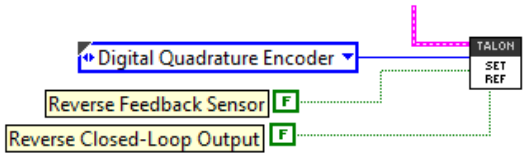
The selected “Feedback Device” defaults to Quadrature Encoder.

Once a “Feedback Device” is selected, the “Sensor Position” and “Sensor Velocity” signals will update with the output of the selected feedback device. It will also be multiplied by (-1) if “Reverse Feedback Sensor" is asserted programmatically.

Alternatively the output of the closed loop logic can also be inverted if necessary.

7.1. LabVIEW

Use SET REF to select which Feedback Sensor to use for soft limits and closed-loop features. The supported selections are: Quadrature Encoder, Analog Encoder (or any continuous 3.3V analog sensor) and Analog Potentiometer.



7.2. C++

`SetFeedbackDevice()` can be used to select Quadrature Encoder, Analog Encoder (or any continuous 3.3V analog sensor) or Analog Potentiometer. Depending on software release EncRising may also be supported (increment position per rising edge on Quadrature-A).

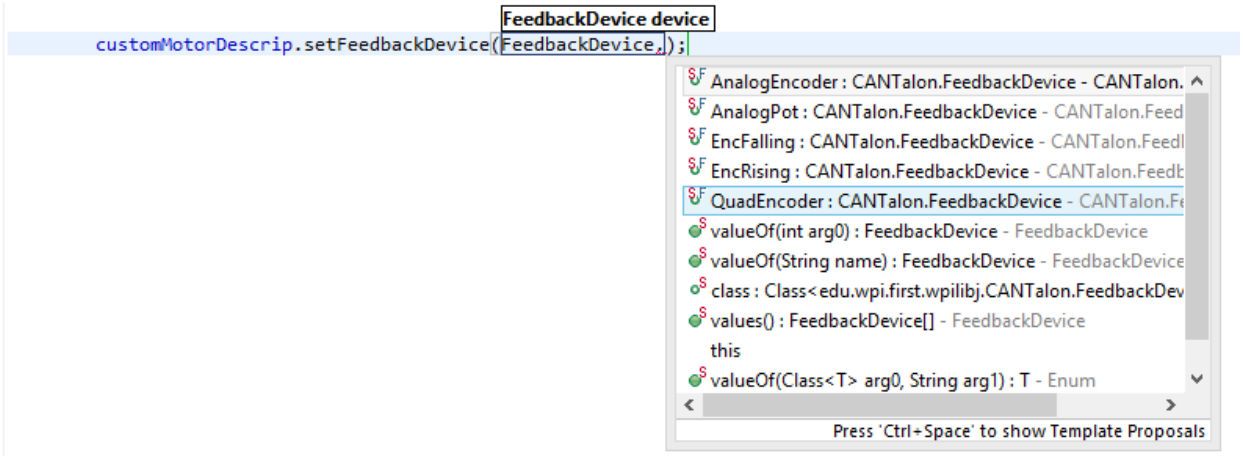
```
customMotorDescrip.SetFeedbackDevice(CANTalon::QuadEncoder);     /* Quadrature encoder */
customMotorDescrip.SetFeedbackDevice(CANTalon::AnalogPot);       /* Absolute analog signal, 0-3.3V */
customMotorDescrip.SetFeedbackDevice(CANTalon::AnalogEncoder);   /* Relative analog signal, 0-3.3V */
```

`SetSensorDirection()` can be used to keep the sensor and motor in phase for proper limit switch and closed loop features. This functions sets the "Reverse Feedback Sensor" signal.

```
/* pass true to reverse feedback sensor, false to leave it pure. */
customMotorDescrip.SetSensorDirection(true);
```

7.3. Java

`setFeedbackDevice()` can be used to select Quadrature Encoder, Analog Encoder (or any continuous 3.3V analog sensor) or Analog Potentiometer. Depending on software release EncRising may also be supported (increment position per rising edge on Quadrature-A).

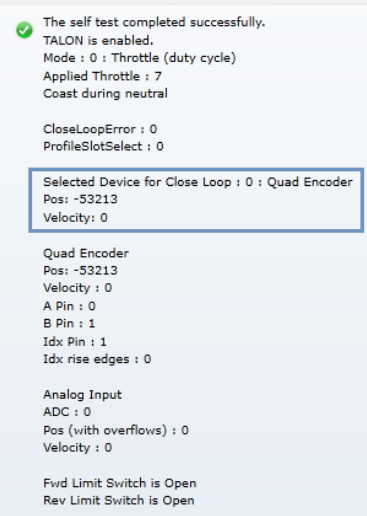


`reverseSensor()` and `reverseOutput()` are also available in java. `reverseSensor()` sets the "Reverse Feedback Sensor" signal. `reverseOutput()` sets the "Reverse Closed-Loop Output" signal.

```
/* pass true to reverse feedback sensor, false to leave it pure. */
customMotorDescrip.reverseSensor(true);
/* pass true to reverse the output of the closed loop math as an alternative method to flip motor direction.
 * Typically reverseSensor is sufficient to keep sensor and motor in
 * phase for proper limit switch and closed loop features. */
customMotorDescrip.reverseOutput(false);
```

7.4. Reversing sensor direction, best practices.

In order for limit switches and closed-loop features to function correctly the sensor and motor has to be “in-phase”. This means that the sensor position must move in a positive direction as the motor controller drives positive throttle. To test this, first drive the motor manually (using gamepad axis for example). Watch the sensor position either in the roboRIO Web-based Configuration Self-Test, or by calling `GetSensorPosition()` and printing it to console. If the “Sensor Position” moves in a negative direction while Talon SRX throttle is positive (blinking green), then use the “Reverse Feedback Sensor” signal to multiply the sensor position by (-1). Then retest to confirm “Sensor Position” moves in a positive direction with positive motor drive.



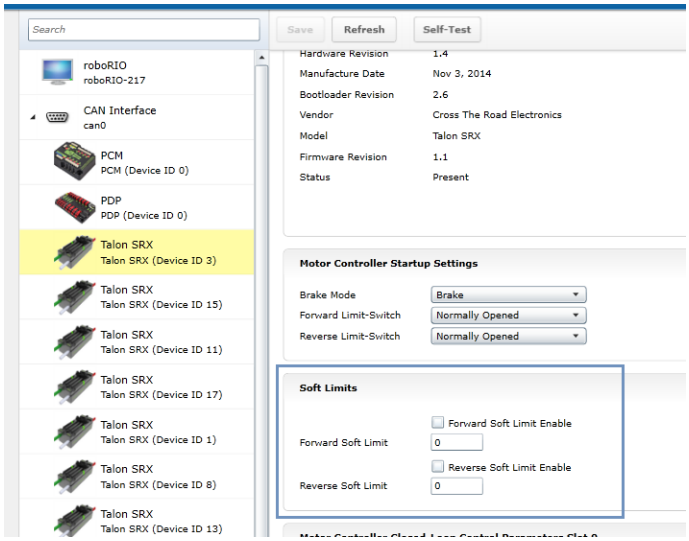
When using the Self-Test be sure to track the Selected Device Position which is **above** the Quadrature Encoder signals. These signals will reflect if "Reverse Feedback Sensor" is asserted.

In the special case of using the EncRising feedback device, "Reverse Feedback Sensor" will need to be false. This Feedback Device is guaranteed to be positive since it increments per rising edge, and never decrements. "Reverse Closed-Loop Output" can then be used to output a negative motor duty-cycle. "Reverse Closed-Loop Output" can also be used to reverse a slave Talon SRX to be the signed opposite of the master Talon SRX.

8. Soft Limits

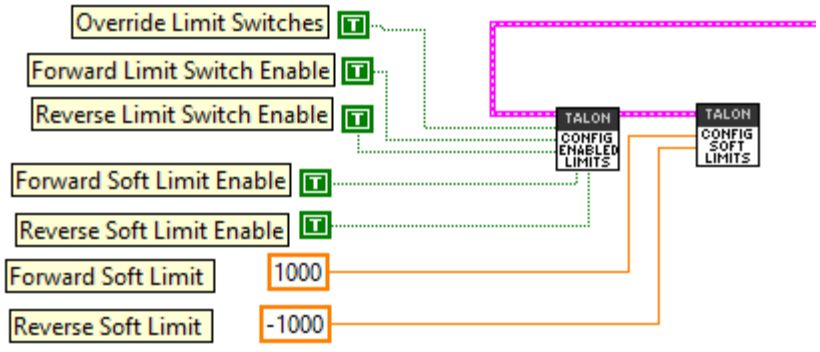
Soft limits can be used to disable motor drive when the “Sensor Position” is outside of a specified range. Forward throttle will be disabled if the “Sensor Position” is greater than the Forward Soft Limit. Reverse throttle will be disabled if the “Sensor Position” is less than the Reverse Soft Limit. The respective Soft Limit Enable must be enabled for this feature to take effect.

The settings can be set and confirmed in the roboRIO Web-based Configuration.



8.1. LabVIEW

The soft limits can also be set up programmatically. In LabVIEW, Soft Limit enables and thresholds can be set using both the `CONFIG ENABLED LIMITS VI` and `CONFIG SOFT LIMITS VI`.



8.2. C++

`ConfigLimitMode()` can be used to enable soft limits (and optionally limit switches if they are wired).

```
customMotorDescrip.ConfigLimitMode(CANSpeedController::kLimitMode_SoftPositionLimits);      /* limits switches and soft-limits */
customMotorDescrip.ConfigForwardLimit(20000);   /* set forward soft limit to 20,000 */
customMotorDescrip.ConfigReverseLimit(-20000);  /* set reverse soft limit to -20,000 */
```

8.3. Java

The limit threshold and enabled states can be individually specified using: `setForwardSoftLimit()`, `enableForwardSoftLimit()`, `setReverseSoftLimit()`, and `enablReverseSoftLimit()`.

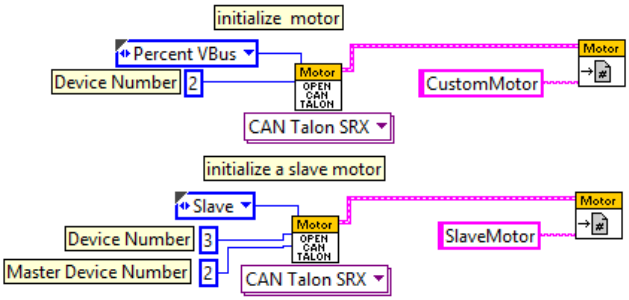
```
customMotorDescrip.setForwardSoftLimit(10000);      /* set forward soft limit to 10,000 */
customMotorDescrip.enableForwardSoftLimit(true);    /* enable forward soft limit */
customMotorDescrip.setReverseSoftLimit(-10000);     /* set reverse soft limit to -10,000 */
customMotorDescrip.enableReverseSoftLimit(true);    /* enable reverse soft limit */
```

9. Follower Mode

Any given Talon SRX on CAN bus can be instructed to “follow” the drive output of another Talon SRX. This is done by putting a Talon SRX into “slave” mode and specifying the device ID of the “Master Talon” to follow. The “Slave Talon” will then mirror the output of the “Master Talon”. The “Master Talon” can be in any mode: closed-loop, voltage percent (duty-cycle), or even following yet another Talon SRX.

9.1. LabVIEW

When opening a Talon SRX, the Master Device Number determines which Talon to follow when in Slave Mode. The `CHANGE MODE VI` can also be used to enter Slave mode and specify the Master Device ID to follow.



9.2. C++

CANTalon objects can be constructed with the Follower mode, or can be changed afterwards. Pass the device ID of the Master Talon into `Set()`. The device ID should be between 0 and 62 (inclusive).

```
slaveMotor.SetControlMode(CANSpeedController::kFollower); /* set this motor to follow another TALON */
slaveMotor.Set(2); /* follow master TALON with device ID 2 */
```

9.3. Java

CANTalon objects can be constructed with the Follower mode, or can be changed afterwards. Pass the device ID, of the Master Talon into set(). The device ID should be between 0 and 62 (inclusive). Alternatively you can call the `getDeviceID()` routine of the Talon object created with the Master Talon SRX’s device ID.

```
slaveMotor.changeControlMode(CANTalon.ControlMode.Follower);
slaveMotor.set(customMotorDescrip.getDeviceID());
```

9.4. Reversing Slave Motor Drive

"Reverse Closed-Loop Output" can be used to invert the output of a slave Talon. This may be useful if a slave and master Talon are wired out of phase with each other.

10. Closed-Loop Modes

The 2015 release firmware for Talon SRX supports position closed-loop and velocity closed-loop. The actual implementation can be seen in Section 18. How is the closed-loop implemented?. Future firmware updates will include Current Closed-Loop, Voltage compensation, and other modes.

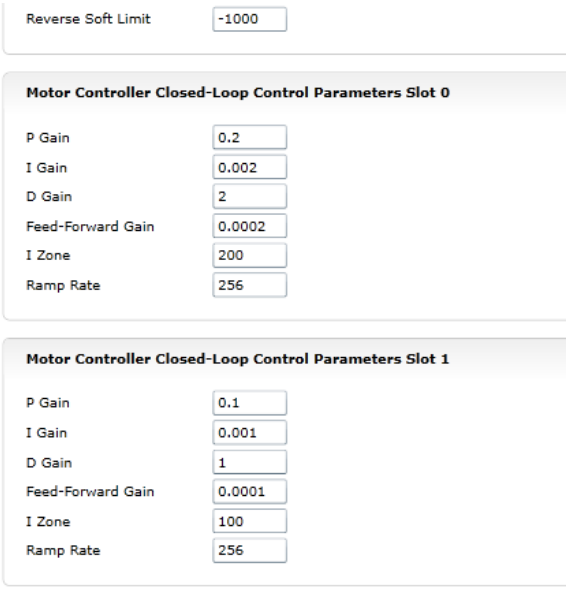
11. Motor Control Profile Parameters

The Talon persistently saves two unique Motor Control Profiles.
Each Motor Control Profile contains…

- P Gain: K_p constant to use when control mode is a closed-loop mode.
- I Gain: K_I constant to use when control mode is a closed-loop mode.
- D Gain: K_D constant to use when control mode is a closed-loop mode.
- F Gain: K_F constant to use when control mode is a closed-loop mode.
- I Zone: Integral Zone. When nonzero, Integral Accumulator is automatically cleared when the absolute value of Closed-Loop Error exceeds it.
- (Closed-Loop) Ramp Rate: Ramp rate to apply when control mode is a closed-loop mode.

One unique feature of the Talon SRX is that gain values specified in a Motor Control Profile are not dedicated to just one type of closed-loop. When selecting a closed-loop mode (for example position or velocity) the robot application can select either of the two Motor Control Profiles to select *which* set of values to use. This can be useful for gain scheduling (changing gain values on-the-fly) or for persistently saving two sets of gains for two entirely different closed loop modes.

The settings can be set and read in the web control page.



11.1. Persistent storage and Reset/Startup behavior

The Talon SRX was designed to reduce the “setup” necessary for a Talon SRX to be functional, particularly with closed-loop features. This is accomplished with efficient CAN framing and persistent storage.

All settings in the Motor Control Profile (MCP) are saved persistently in flash. Additionally there are two complete Motor Control Profiles. Teams that use a constant set of values can simply set them using the roboRIO Web-based Configuration, and they will “stick” until they are changed again.

Additionally Motor Control Profile (MCP) Parameters can be changed though programming API. When they are changed, the values are ultimately copied to persistent memory using a wear leveled strategy that ensures Flash longevity, but also meets the requirements for teams.
-Changing MCP values programmatically always take effect immediately (necessary for gain tuning).
-If the MCP Parameters have remained unchanged for fifteen seconds, and an MCP Parameter value is then changed using programming API, they are copied to persistent memory immediately.
-If the persistent memory has been updated within the last fifteen seconds due to a previous value change, and an MCP Parameter value is changed again, it will be applied to persistent memory once fifteen seconds has passed since the last persistent memory update. However the closed-loop will react immediately to the latest values sent over CAN bus.
-If power loss occurs during the period of time when MCP Parameters are being saved to persistent storage, the previous values for all MCP Parameters prior to last value-change is loaded. This is possible because the Talon SRX keeps a small history of all value changes.

These features fit well with the two common strategies that FRC teams utilize when programmatically changing closed-loop parameters...

(1) Teams use programming API at startup to apply previous tested constants.
(2) Teams use programming API to periodically set/change the constants because they are “gain scheduled” or action specific.

For use case (1), the constants are eventually saved in Talon SRX persistent memory (worst case fifteen seconds after robot startup). Once this is done the Talon SRX will have the values in persistent storage, so even after Talons are power cycled, they will load the constants that were previous set. This frees the robot controller from needing to re-set the values during a power cycle, reset, brownout, etc…. On subsequent robot startups, when the robot controller sends the same values again, and Talon SRX will still react by updating its variables, and comparing against what’s saved in persistent storage to see if it needs to be updated again. In the event the robot code changes to use new constants, the Talon will again update the persistent storage shortly after getting the new values.

For use case (2) teams, there are two “best” solutions depending on what’s being accomplished. If a team needs to switch between two sets of gains, they can leverage both MCP slots by setting one set of constants in slot 0, and another unique set of constants in slot 1. Then during the match, teams can switch between the two with a single API. This means that as far as the Talon is concerned, the values in each slot never changes so the contents of the Talon’s persistent storage never changes. Instead the robot controller just changes *which* slot to use. So this use case regresses to use case (1), and a freshly booted Talon already has all the MCP parameters it needs to function.

For use case(2) teams that requires more than two gain sets likely are changing gain values so frequently (as a function of autonomous, or state machine driven logic) that they would prefer not to rely on the previous set of gains sent to the Talon (despite it being available at startup). In which case they likely will periodically set the MCP parameters continuously (every number of loops or fixed period of time). Talon SRX always honors whatever parameters are requested over CAN bus, overriding what was loaded at startup or mirrored in persistent storage. And since the persistent storage is wear-leveled and mirrored at fifteen second intervals, this has no harmful impact on Flash longevity. So this use case is also supported well.

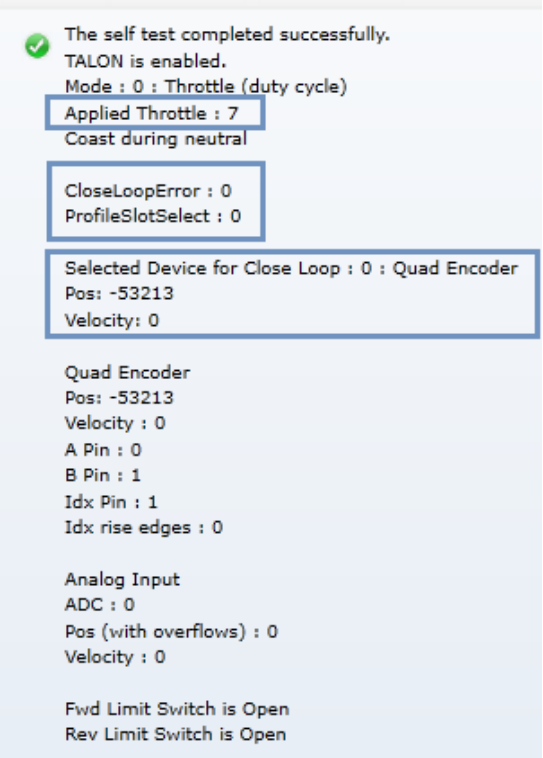
Beyond the Motor Control Profile Parameters, closed-loop modes requires selecting
-which control mode (position or velocity)
-which feedback sensor to use
-if the feedback sensor should be reversed
-if the closed-loop output should be reversed
-what is the latest target or set point
-the global ramp rate (if specified)
-which Motor Control Profile Slot to use.

The programming API provides set functions for all of these, but what’s noteworthy is that all of these signals are saved in the robot controller, and periodically sent inside *one complete* CAN frame. This means that if a Talon SRX loses power and is booted back up again (due to cable disconnect, battery brownout, etc…) the Talon receives all of the necessary signals after *getting one single control frame*. This is far more robust than requiring the robot application to re-set and re-acknowledge each parameter individually in the event of a reset.

11.2. Inspecting Signals

When testing/calibrating closed-loops it is helpful to plot/check…
-Closed-Loop Error
-Output (Applied) Throttle
-Profile Slot Select (which profile slot the closed-loop math is using).
-Position and Velocity depending control mode.

The Self-Test can provide these values for quick sanity checking. These values are also available with programming API for custom plotting, smart dashboard, LabVIEW front panels, etc...

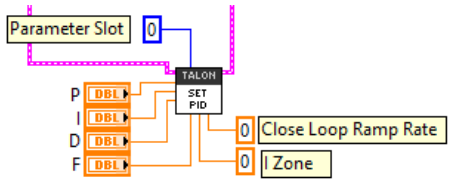


12. Position/Velocity Closed-Loop Example

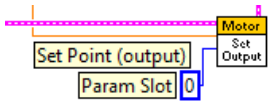
12.1. Setting Motor Control Profile Parameters

12.1.1. LabVIEW

Setting the Motor Controller Profile parameters can be done with the SET PID VI.
This allows filling all parameters for a given Parameter Slot.



Specifying the set point is also done with the Set Output VI. Additionally you can select the Parameter Slot to use for the selected closed-loop.



12.1.2. C++

Closed-loop parameters for a given profile slot can be modified with several different functions.

```
double p = 0.3;         /*Kp */
double i = 0.003;       /*Ki */
double d = 3;           /*Kd */
double f = 0.0003;      /*Kf */
int izone = 300;        /* encoder ticks / analog units */
double ramprate = 48;   /* volts per second, => 0% to 100% in 250ms */
int profile = 1;    /* can be 0 or 1 */
customMotorDescrip.SelectProfileSlot(profile);
customMotorDescrip.SetPID(p, i, d, f);
customMotorDescrip.SetIzone(izone);
customMotorDescrip.SetCloseLoopRampRate(ramprate);
```

Setting the target position or velocity is also done with `Set()`.

```
customMotorDescrip.Set(targetPosOrVel); /* use Set() to servo to target position or velocity */
```

12.1.3. Java

Closed-loop parameters for a given profile slot can be modified using `setPID()`. This also sets the “Profile Slot Select” to the slot being modified. There are also individual Set functions for each signal.

```
double p = 0.1;         /*Kp */
double i = 0.001;       /*Ki */
double d = 1;           /*Kd */
double f = 0.0001;      /*Kf */
int izone = 100;        /* encoder ticks / analog units */
double ramprate = 36;   /* volts per second */
int profile = 0;    /* can be 0 or 1 */
customMotorDescrip.setPID(p, i, d, f, izone, ramprate, profile);
```

Setting the target position or velocity is also done with `set()`.

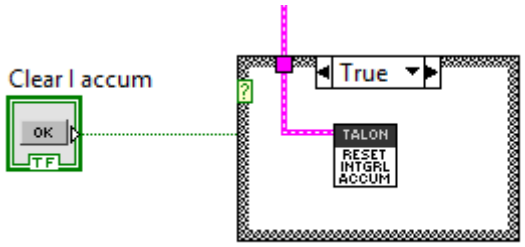
```
customMotorDescrip.set(targetPosOrVel); /* use set() to servo to target position or velocity */
```

12.2. Clearing Integral Accumulator (I Accum)

Clearing the integral accumulator ("I Accum") may be necessary to prevent integral windup. When using "I Zone" this is done automatically when the Closed-Loop Error is outside the "I Zone". However there may be other situations when manually clearing the integral accumulator is necessary. For example, if the mechanism that's being closed-looped is "close enough" and its desirable to reduce occasional spurts of movement caused by a slowly incrementing integral term, then the robot logic can periodically clear the "I Accum" to prevent this.

12.2.1. LabVIEW

In this example a case structure is leveraged to conditionally clear the Integral Accumulator when the case structure conditional evaluates true (this example uses a system button on the front panel).



12.2.2. C++/Java

The `ClearIaccum()` function is available in C++/Java.

```
customMotorDescrip.ClearIaccum();
```

12.2.3. Is Integral Accum cleared any other time?

In addition to the "I Zone" feature and manual clear, there are certain cases where the integral accumulator is automatically cleared for more predicable motor response…
-Whenever the control mode of a Talon is changed.
-When a Talon is in the disabled state.
-When the motor control profile slot has changed.

13. Setting Sensor Position

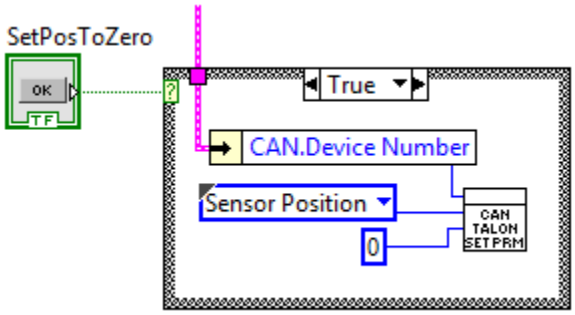
Depending on the sensor selected, the user can modify the “Sensor Position”. This is particularly useful when using a Quadrature Encoder (or any relative sensor) which needs to be “zeroed” or “home-ed” when the robot is in a known position.

Firmware 1.1: When using an “Analog Encoder”, setting the “Sensor Position” updates the top 14 bits of the 24bit Sensor Position value (which is the overflow/underflow portion). The bottom 10 bits will still reflect the analog voltage scaled over 3.3V (read only). With version 1.4 and on the full Sensor Position can be set.

Setting this signal when “Analog Potentiometer” is selected has no effect.

13.1. LabVIEW

In order to modify the “Sensor Position”, user will likely have to leverage the `WPI_CANTalonSRX_SetParameter.vi`. This can be drag dropped after locating the LabVIEW installer directory and searching for the VI file location. Select “Sensor Position” for the “Parameter” signal and the desired constant for the “Value” signal. In this example “0” is selected to re-zero the sensor when a front panel button is pressed.



13.2. C++

`SetPosition()` can be used to change the current sensor position, if a relative sensor is used.

```
customMotorDescrip.SetPosition(0); /* set the sensor position to zero */
```

13.3. Java

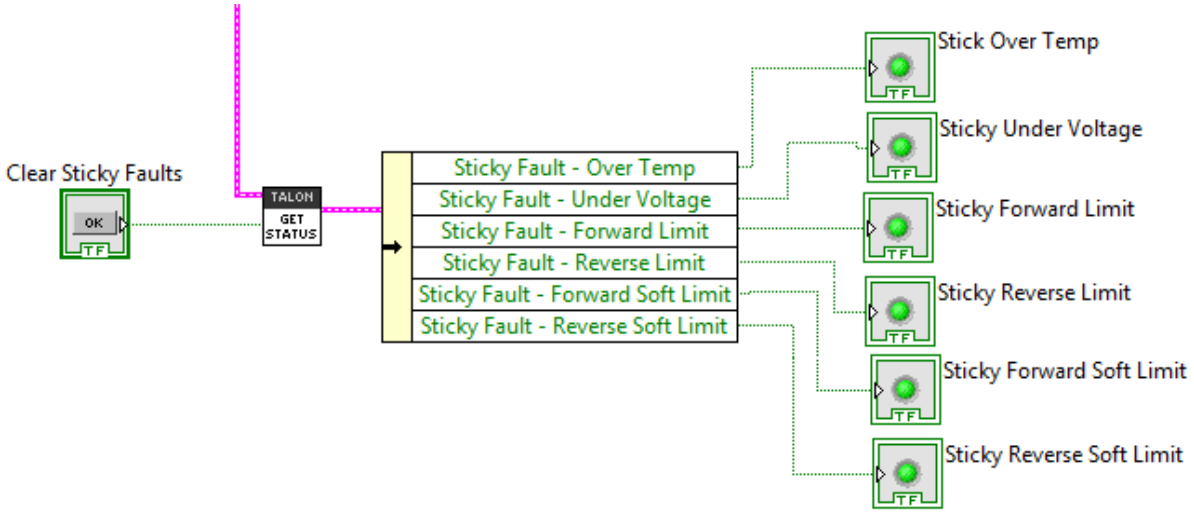
`setPosition()` can be used to change the current sensor position, if a relative sensor is used.

```
customMotorDescrip.setPosition(0); /* set the sensor position to zero */
```

14. Fault Flags

The `GET STATUS VI` can be used to retrieve sticky flags, and clear them.

14.1. LabVIEW



14.2. C++

Use `GetFaults()` and `GetStickyFaults()` to get integral bit fields that can be masked against the constants available in the `CANSpeedController` header.

`ClearStickyFaults()` can be used to clear all sticky fault flags.

```
if(btn5){
    customMotorDescrip.ClearStickyFaults();            /* clear sticky faults */
}
if(bEverySecond){
    int faults = customMotorDescrip.GetFaults();           /* get bitfield of all faults */
    int stickyFaults = customMotorDescrip.GetStickyFaults();/* get bitfield of all sticky faults */

    if(faults & CANSpeedController::kTemperatureFault)
        printf("kTemperatureFault\r\n");
    if(faults & CANSpeedController::kBusVoltageFault)
        printf("kBusVoltageFault\r\n");
    if(faults & CANSpeedController::kFwdLimitSwitch)
        printf("kFwdLimitSwitch\r\n");
    if(faults & CANSpeedController::kRevLimitSwitch)
        printf("kRevLimitSwitch\r\n");
    if(faults & CANSpeedController::kFwdSoftLimit)
        printf("kFwdSoftLimit\r\n");
    if(faults & CANSpeedController::kRevSoftLimit)
        printf("kRevSoftLimit\r\n");

    if(stickyFaults & CANSpeedController::kTemperatureFault)
        printf("Sticky - kTemperatureFault\r\n");
    if(stickyFaults & CANSpeedController::kBusVoltageFault)
        printf("Sticky - kBusVoltageFault\r\n");
    if(stickyFaults & CANSpeedController::kFwdLimitSwitch)
        printf("Sticky - kFwdLimitSwitch\r\n");
    if(stickyFaults & CANSpeedController::kRevLimitSwitch)
        printf("Sticky - kRevLimitSwitch\r\n");
    if(stickyFaults & CANSpeedController::kFwdSoftLimit)
        printf("Sticky - kFwdSoftLimit\r\n");
    if(stickyFaults & CANSpeedController::kRevSoftLimit)
        printf("Sticky - kRevSoftLimit\r\n");
}
```

14.3. Java

Use the various `getFault<name>` and `getStickyFault<name>` functions to individually detect the various fault conditions.

```
getFaultOverTemp()
getFaultUnderVoltage()
getFaultForLim()
getFaultRevLim()
getFaultForSoftLim()
getFaultRevSoftLim()
getStickyFaultOverTemp()
getStickyFaultUnderVoltage()
getStickyFaultForLim()
getStickyFaultRevLim()
getStickyFaultForSoftLim()
getStickyFaultRevSoftLim()
```

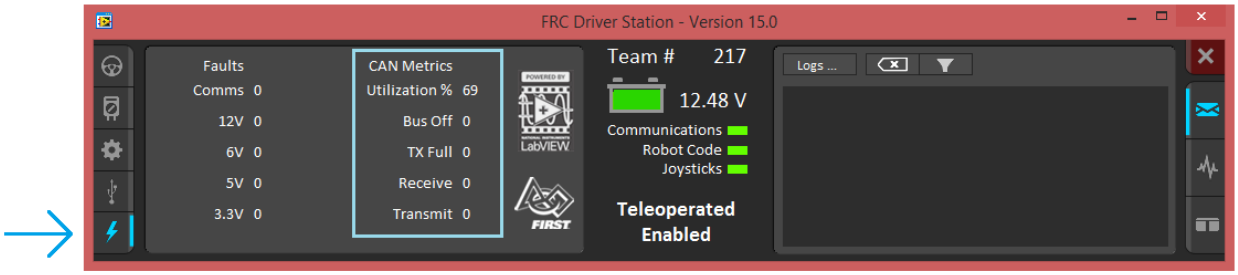
`clearStickyFaults()` can be used to clear all sticky fault flags.

```
if(btn5){
    customMotorDescrip.clearStickyFaults();            /* clear sticky faults */
}
if(bEverySecond){

    if(customMotorDescrip.getFaultOverTemp() != 0)
        System.out.println("FaultOverTemp");
    if(customMotorDescrip.getFaultUnderVoltage() != 0)
        System.out.println("FaultUnderVoltage");
    if(customMotorDescrip.getFaultForLim() != 0)
        System.out.println("FaultForLim");
    if(customMotorDescrip.getFaultRevLim() != 0)
        System.out.println("FaultRevLim");
    if(customMotorDescrip.getFaultForSoftLim() != 0)
        System.out.println("FaultForSoftLim");
    if(customMotorDescrip.getFaultRevSoftLim() != 0)
        System.out.println("FaultRevSoftLim");

    if(customMotorDescrip.getStickyFaultOverTemp() != 0)
        System.out.println("StickyFaultOverTemp");
    if(customMotorDescrip.getStickyFaultUnderVoltage() != 0)
        System.out.println("StickyFaultUnderVoltage");
    if(customMotorDescrip.getStickyFaultForLim() != 0)
        System.out.println("StickyFaultForLim");
    if(customMotorDescrip.getStickyFaultRevLim() != 0)
        System.out.println("StickyFaultRevLim");
    if(customMotorDescrip.getStickyFaultForSoftLim() != 0)
        System.out.println("StickyFaultForSoftLim");
    if(customMotorDescrip.getStickyFaultRevSoftLim() != 0)
        System.out.println("StickyFaultRevSoftLim");
}
```

15. CAN bus Utilization/Error metrics

The driver station provides various CAN bus metrics under the “lightning bolt” tab.
Utilization is the percent of bus time that is in use relative to the total bandwidth available of the 1Mbps Dual Wire CAN bus. So at 100% there is no idle bus time (no time between frames on the CAN bus).

Demonstrated here is 69% bus use when controlling 16 Talon SRXs, along with 1 Pneumatics Control Module (PCM) and the Power Distribution Panel (PDP).



The “Bus Off” counter increments every time the CAN Controller in the roboRIO enters “bus-off”, a state where the controller “backs off” transmitting until the CAN bus is deemed “healthy” again. A good method for watching it increment is to short/release the CAN bus High and Low lines together to watch it enter and leave “Bus Off” (counter increments per short).

The “TX Full” counter tracks how often the buffer holding outgoing CAN frames (RIO to CAN device) drops a transmit request. This is another common symptom when the roboRIO no longer is connected to the CAN bus.

The “Receive” and “Transmit” signal is shorthand for “Receive Error Counter” and “Transmit Error Counter”. These signals are straight from the CAN bus spec, and track the error instances occurred “on the wire” during reception and transmission respectively. These counts should always be zero. Attempt to short the CAN bus and you can confirm that the error counts rise sharply, then decrement back down to zero when the bus is restored (remove short, reconnect daisy chain).

When starting out with the FRC control system and Talon SRXs, it is recommend to watch how these CAN metrics change when CAN bus is disconnected from the roboRIO and other CAN devices to learn what to expect when there is a harness or a termination resistor issue. Determining hardware related vs software related issues is key to being successful when using a large number of CAN devices.

15.1. How many Talons can we use?

Generally speaking a maximum of 16 Motor controllers can be powered at once using a single PDP (sixteen breaker slots). However FRC game rules should always be checked as it determines what it considered legal. This is typically the bottleneck for how many Talon SRXs can be used despite having CAN device ID space for 63 device IDs. Release software is always tested to support 16 Talon SRXs, 1 PCM, and 1 PDP with guaranteed control of each Talon at a rate of 10ms. However this is not the limit. There is still additional bandwidth for more nodes. Additionally, if faster response time is desired, control frame periods can be decreased from the default 10ms, but keep a watchful eye of the CAN bus utilization to ensure reliable communication.

16. Troubleshooting Tips and Common Questions

16.1. When I press the B/C CAL button, the brake LED does not change, neutral behavior does not change.

This is the expected behavior if the robot application is overriding the brake mode. The B/C CAL button press does toggle the brake mode in persistent memory, however the LED and selected neutral behavior will honor the override sent over CAN bus. Check if the override API is being used in the robot application logic.

16.2. Changing certain settings in Disabled Loop doesn't take effect until the robot is enabled.

This is the expected behavior, the control frame that updates the Talon SRX with the latest brake/limit switch override, control mode, and set-point does not get transmitted in robot-disable mode. However the values are saved so that when the robot is enabled, the first control frame sent will have up to date values.

For example, the B/C CAL LED will not reflect changes in the overridden brake mode if `SetBrake()` is called in the disabled loop until **after** the robot has been enabled. Once the robot is enabled, the control frame sent to the Talon will enable the motor controller and contain the latest settings that were cached during disabled loop (including the brake override).

Similarly Self-Test results also won't reflect parameters that are changed programmatically until the robot is enabled, specifically the brake/limit switch overrides.

The startup values for limit switch and brake mode can be specified in the roboRIO's Web-based Configuration if they must be configured prior to robot enable.

16.3. The robot is TeleOperated/Autonomous enabled, but the Talon SRX continues to blink orange (disabled).

Most likely the device ID of that Talon is not being used. In other words there is no Open Motor (LabVIEW) or constructed CANTalon (C++/Java) with that device ID. This can be confirmed by doing a Self-Test in the roboRIO Web-based Configuration, and confirm the "TALON IS NOT ENABLED!" message at the top.

16.4. When I attach/power a particular Talon SRX to CAN bus, The LEDs on every Talon SRX occasionally blink red. Motor drive seems normal.

If there is a single CAN error frame, you can expect all Talon SRXs on the bus to synchronously blink red. This is a great feature for detecting intermittent issues that normally would go unnoticed. If attaching a particular Talon brings this behavior out, most likely its device ID is common with another Talon already on the bus. This means two or more "common ID" Talon SRXs are periodically attempting to transmit using the same CAN arbitration ID, and are stepping on each other's frame. This causes an intermittent error frame which then reveals

itself when all Talon SRXs blink red. Check the roboRIO Web-based Configuration for the "There are X devices with this Device ID" explained in Section 2.2. Common ID Talons.

16.5. If I have a slave Talon SRX following a master Talon SRX, and the master Talon SRX is disconnected/unpowered, what will the slave Talon SRX do?

The slave Talon SRX monitors for throttle updates from the master. If the slave Talon doesn't see an update after 100ms, it will disable its drive. The LEDs will reflect robot-enable but with zero throttle (solid orange LEDs).

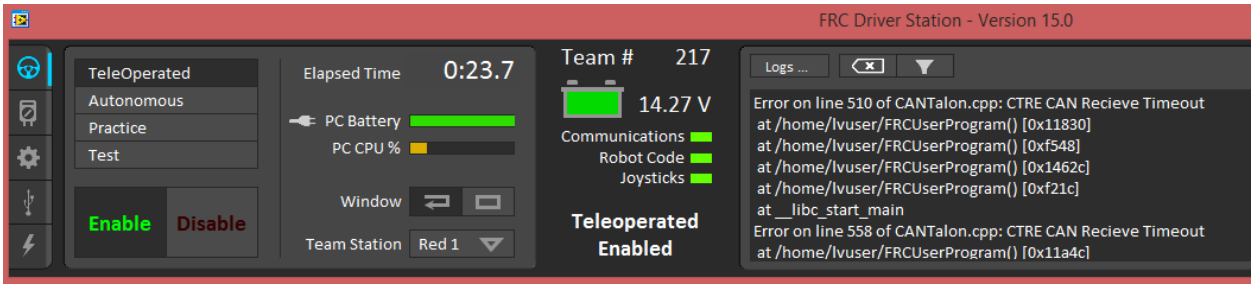
16.6. Is there any harm in creating a software Talon SRX for a device ID that's not on the CAN bus? Will removing a Talon SRX from the CAN bus adversely affect other CAN devices?

No! Attempting to communicate with a Talon SRX that is not present will not harm the communication with other CAN nodes. The communication strategy is very different than previously support CAN devices, and this use case was in mind when it was designed.

Creating more Talon software objects (LabVIEW Motor Open, or C++/Java class instances) will increase the bus utilization since it means sending more frames, however this should not adversely affect robot behavior so long as the bus utilization is reasonable.

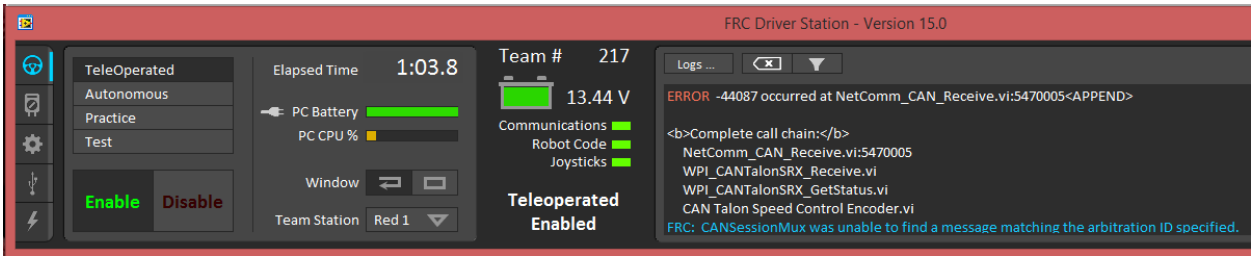
However the resulted error messages in the DS may be a distraction so when permanently removing a Talon SRX from the CAN bus, it is helpful to synchronously remove it from the robot software.

16.7. Driver Station log says Error on line XXX of CANTalon.cpp



This is to be expected when constructing a CANTalon with a device ID that is not present on CAN bus in C++/Java. These are caused by asserts in the programming API meant to signal the developer that some expected CAN frame was not detected on the bus. Although it should not impact other CAN nodes, it can be a distraction since it may mask other unrelated errors reported in the driver station that are no longer visible.

16.8. Driver Station log says -44087 occurred at NetComm...



This is to be expected when referencing a “CAN Talon SRX” with a device ID that is not present on CAN bus in LabVIEW. Although it should not impact other CAN nodes, it can be a distraction since it may mask other unrelated errors reported in the driver station that are no longer visible.

16.9. Why are there multiple ways to get the same sensor data? GetEncoder() versus GetSensor()?

The API that fetches latest values for Encoder (Quadrature) and Analog-In (potentiometer or a continuous analog sensor) reflect the pure decoded values sent over CAN bus (every 100ms). They are available all the time, regardless of which control mode is applied or whether the sensor type has been selected for soft limits and closed-loop mode. These signals are ideal for instrumenting/logging/plotting sensor values to confirm the sensors are wired and functional. Additionally they can be read at the same time (you can wire a potentiometer AND a quadrature encoder and get both position and velocities programmatically). Furthermore the robot application could actually use this method to process sensor information directly. If the 100ms update rate is not sufficient, it can be overridden to a faster rate.

For the purpose of using soft limits and/or closed-loop modes, the robot application must select which sensor to use for position/velocity. Selecting a sensor will cause the Talon SRX to mux the appropriate sensor to the closed-loop logic, the soft limit logic, to the “Sensor Position” and “Sensor Velocity” signals (update 20ms). These signals also can be inverted using the "Reverse Feedback Sensor" signal in order to keep the sensor in phase with the motor.

Since “Sensor Position” and “Sensory Velocity” are updated faster (20ms) they can also be used for processing sensor information instead of overriding the frame rates.

16.10. So there are two types of ramp rate?

There are two ways to “ramp” or acceleration cap the motor drive.
The “Voltage Ramp” API in all three languages are functional, and takes effect regardless of which mode the Talon SRX is in (duty cycle, slave, Position, Speed).

The selected Motor Control Profile also has a “Closed-Loop Ramp Rate” which can be used to apply a unique ramp rate only when the motor controller is closed-looping and the Profile Slot has been selected.

By having two options, a robot can have mode specific ramping with minimal effort in the robot controller. For example, a team may require a “weak” ramp to slightly dampen motor drive to

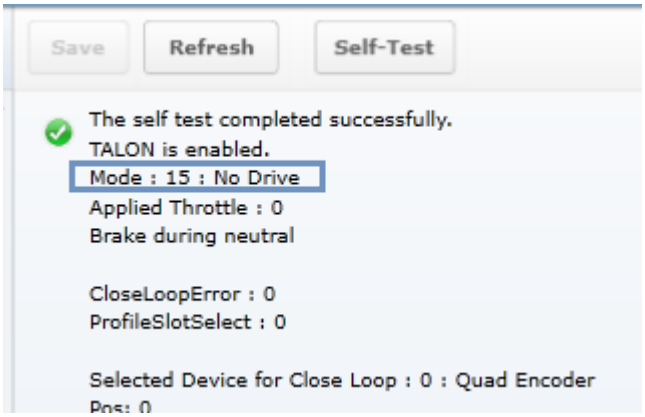
prevent driver error (flipping the robot), or to reduce impulse stress (snapping chains). But when closed-looping, an additional ramp might be necessary to smooth the closed-loop maneuver.

16.11. Why are there two feedback “analog” device types: Analog Encoder and Analog Potentiometer?

When Analog Potentiometer is selected, the 10 bit Analog to Digital Converter (ADC) converts the 0 to 3.3V signal present on the analog input pin to a value between 0 and 1023.

When Analog Encoder is selected, the same conversion takes place, but rollovers to and from the min/max voltage are tracked so that analog sensors can be used as relative sensors. If an overflow is detected (1023 => 0), the position signal transitions (1023 => 1024). Likewise an underflow (0=>1023) is interpreted as (0 => -1). This is useful when using an analog encoder and allowing it to exceed the max turn count. This way an analog encoder can be used as a continuous relative sensor.

16.12. After changing the mode in C++/Java, motor drive no longer works. Self-Test says “No Drive” mode?



After calling a Talon SRX object’s `changeMode()` function, the Talon SRX mode is set to disabled until the `Set()/set()` routine is called. This is to ensure the robot application has a chance to pass a new target set point before the new control mode is applied. Any call to `changeMode()` should be immediately followed with a `Set()` so that motor drive is not set to neutral.

This ensures that when the robot application changes a Talon’s mode, it also specifies the throttle/set-point/or slave ID for the new mode to ensure all the necessary information is set for the mode switch.

16.13. All CAN devices have red LEDs. Recommended Preliminary checks for CAN bus.

Some basic checks for the CAN harness are…

Turn off robot, measure resistance between CANH and CANL.

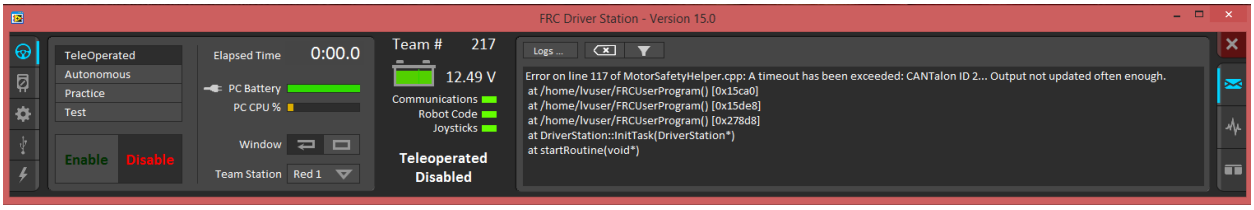
- ~60 ohm is typical (120ohm at each end of the cable).
- ~120 ohm suggests that one end is missing termination resistor. Terminate the end using PDP jumper or explicit 120 ohm resistor.
- ~0 ohm suggests a short between CANH and CANL.
- INF or large resistances, missing termination resistor at each side.

More information can be found in **Talon SRX User's Guide**.

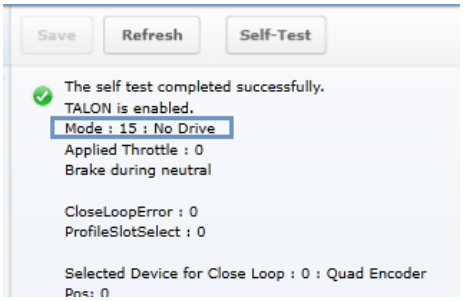
Check the roboRIO's Web-based Configuration to see if any devices appear, and ensure there are no Talon SRX's sharing the same device ID.

16.14. Driver Station reports "MotorSafetyHelper.cpp: A timeout…", motor drive no longer works. roboRIO Web-based Configuration says "No Drive" mode? Driver Station reports error -44075?

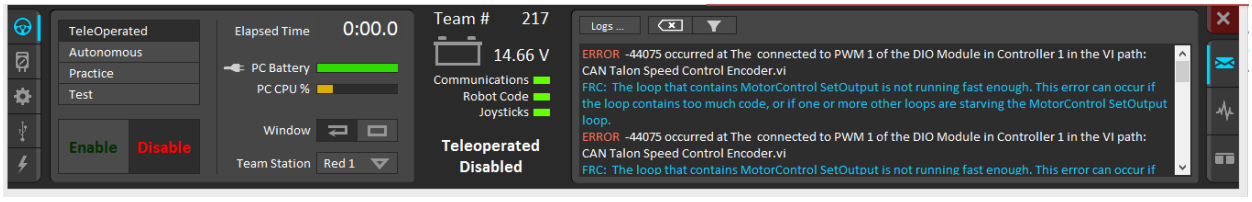
This can happen after enabling Motor Safety Enable and not calling `Set()/set()` often enough to meet the expiration timeout.



Another symptom of this is seeing "No Drive" has the control mode in the Self-Test.



When the safety timeout expires in LabVIEW, the error message will be different…



16.15. Motor drive stutters, misbehaves? Intermittent enable/disable?

Check the CAN Utilization to ensure it's not near 100%. An abnormally high percent may be a symptom of "common ID" Talons. This also can occur when selecting custom frame rates that are too fast.

Check the roboRIO's Web-based Configuration to confirm all expected Talons are populated and are enabled according to the Self-Test.

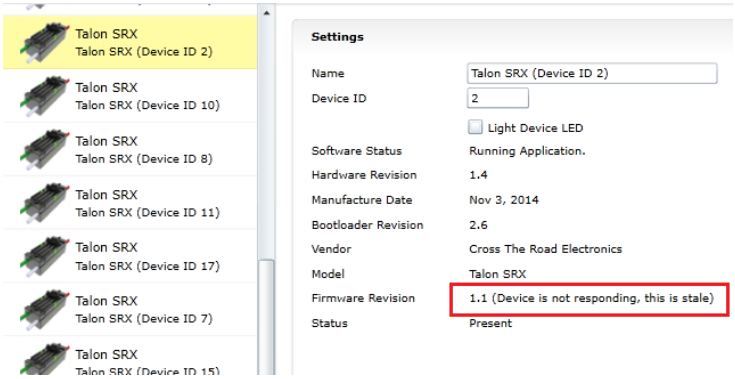
Check the "Under Vbat" sticky flag in the Self-Test. This will rule out power/voltage related issues.

If the issues occur only during rapid changes in direction and magnitude, the power cables/crimps may not be efficient enough to deliver power during the stall-period when a loaded motor changes direction. This can be confirmed if increasing the voltage ramp rate removes/fixes this symptom.

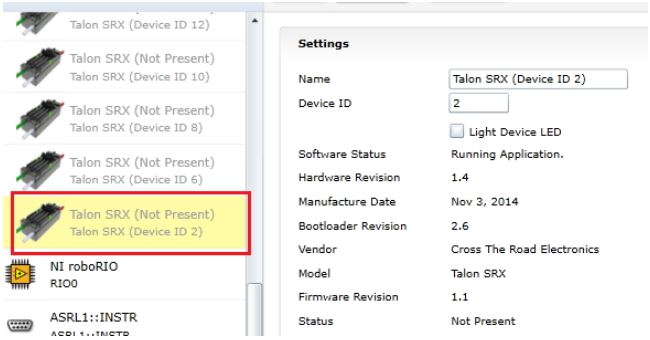
16.16. What to expect when devices are disconnected in roboRIO's Web-based Configuration. Failed Self-Test?

Depending what version of software is released, a discovered Talon will display loss of connection one of two ways.

The Firmware Version may report (Device is not responding).

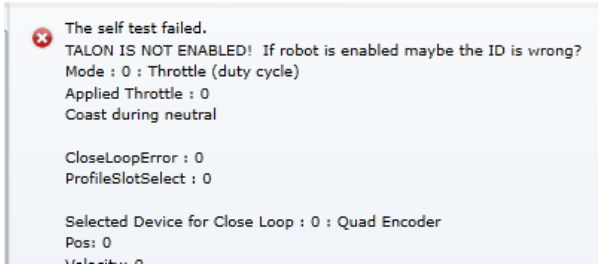


Alternatively the tree element will gray out to indicate loss of communication….



The roboRIO internals rechecks the CAN bus once every five seconds so when connecting/disconnecting Talons to/from the bus, be sure to wait at least five seconds and refresh the webpage to detect changes in connection state.

Doing a Self-Test when the Talon SRX is not present on the CAN bus will report a red 'X' in the top left portion of the Self-Test report. Depending on what robot controller image is release you may see the stale values of all signals when the red "X" is present.



16.17. When I programmatically change the "Normally Open" vs "Normally Closed" state of a limit switch, the Talon SRX blinks orange momentarily.

Changing the "Normally Open" versus "Normally Closed" setting of a motor controller will cause it to disable motor drive momentarily. If the goal is to enable/disable the limit switch feature, this can be done without affecting motor drive using the Limit Switch overrides.

Typically the only time a Talon SRX NO/NC setting won't match what is specified programmatically will be when a new Talon SRX is installed for the first time. Once the programming API has changed the setting once, the Talon SRX's persistent limit switch mode will match what the programming API requests, and therefore will not impact robot performance.

16.18. How do I get the raw ADC value (or voltage) on the Analog Input pin?

The bottom ten bits of Analog-In Position is equal to the converted ADC value. The range of [0, 1023] scales to [0V, 3.3V]. Additionally, if "Analog Potentiometer" is selected as the Feedback Device, the signal "Sensor Position" will exactly equal the bottom ten bits of Analog-In Position.

16.19. Recommendation for using relative sensors.

When using relative sensors for closed-loop control, it's always good practice to design in a way to re-zero your robot. Regardless of how/where relative sensors are connected (robot controller IO, Talon SRX, etc…), there is always the potential for sensors to "walk" or "drift" due to…

-Mechanical slip issues
-Skipped gear teeth in chain
-Intermittent electrical connections (harness gets damaged in middle of a match)
-Power cycle robot when armatures are not in their "home" position.
-Remote resets of robot controller when armatures are not in their "home" position.

A common solution to this is to design a way in the gamepad logic to force your robot into a "manual mode" where the driver/arm operator can manually servo motors to a home position and press a button (or button combination) to re-zero (or set to the "home" position values) all involved sensors.

Teams that do this already can continue to use this method with Talon SRX since there is are set functions to modify "Sensor Position".

16.20. Does anything get reset or lost after firmware updates?

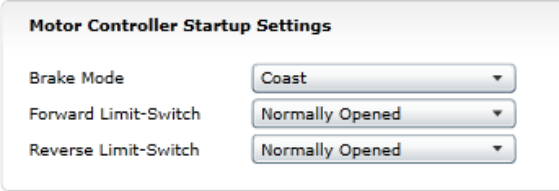
The device ID, limit switch startup settings, brake startup settings, Motor Control Profile Parameters, and sticky flags are all unaffected by the act of field-upgrading. If a particular firmware release has a "back-breaking" change, it will be explicitly documented.

16.21. Analog Position seems to be stuck around ~100 units?

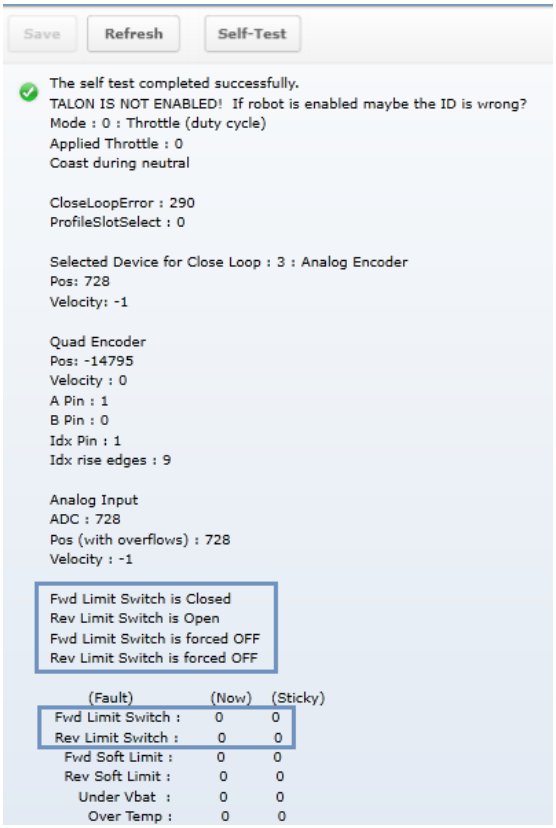
When the analog input is left unconnected, it will hover around 100 units. If an analog sensor has been wired, most likely it's connected to the wrong pin. Recheck wiring against the **Talon SRX User's Guide**.

16.22. Limit switch behavior doesn't match expected settings.

First check the Startup Settings in the roboRIO Web-based Configuration to determine that the "Normally Open"/ "Normally Closed" settings are correct. They can be changed programmatically and in the web page so it's worth confirming. Here we see both directions use NO limit switches...



Then press the "Self-Test" button to check…
-The open/closed state of the limit switch input pin on the Talon SRX.
-If enable/disable state of the limit-switch logic is overridden programmatically.
-Check the fault and sticky faults to see if limit fault conditions are detected.



In this example the Fwd. Limit Switch fault is not set despite the Fwd. Limit Switch being closed. This is because the Limit Switch logic forced OFF, because the feature is disable programmatically. As a result closing the forward limit switch will not disable motor drive.

16.23. How fast can I control just ONE Talon SRX?

The fastest control frame rate that can be specified is 1ms. That means that the average period at which the throttle/set point can be updates is 1ms. This will increase bus utilization by approximately 15%, which is acceptable if the number of Talon SRXs is few. Always check the CAN bus performance metrics in the Driver Station when doing this.

16.24. Expected symptoms when there is excessive signal reflection.

If the CAN bus harness has excessive signal reflection due to improper wiring or missing termination resistors, the following symptoms may be seen…

-Driver Station will show Rx and Tx CAN errors intermittently (see Section 15), particularly with higher bus utilization.

-CAN bus utilization will be higher than normal. This is because CAN devices transmit error frames in response to detecting improper frames. This is helpful if you are in the habit of checking your bus utilization every once in a while and knowing what is typical for your robot. See Section 15 for more details.

-The LED of every CAN device on the bus will blink red intermittently during normal use (the same symptom as Section 16.4). Both common-ID Talons and excessive signal reflection can cause error frames to appear, which trigger every CTRE CAN device to intermittently blink red during normal use.

One reliable way to observe this LED behavior is to deliberately leave a couple common-ID Talon SRXs on your CAN Bus. Then, power up your robot and leave it disabled. All Talon SRXs will rail-road orange (healthy CAN bus and disabled). Now watch any particular Talon SRX for a minute or so. It will blink red intermittently as the two (or more) common-ID Talon SRXs inevitably disrupt each other's frame transmission.

-Measured DC resistance between CANH and CANL (when robot is unpowered) should be approximately 60 Ω. If this is not the case then recheck the CAN wiring and termination resistors (see Talon SRX User's Guide).

17. Units and Signal Definitions

This section describes the units used for various signals.

17.1. (Quadrature) Encoder Position

When measuring the position of a Quadrature Encoder, the position is measured in 4X encoder edges. For example, if a US Digital Encoder with a 360 cycles per revolution (CPR) will count 1440 units per rotation when read using “Encoder Position” or “Sensor Position”.

The velocity units of a Quadrature Encoder is the change in Encoder Position per T_{velMea} ($T_{velMeas}$=0.1sec). For example, if a US Digital Encoder (CPR=360) spins at 20 rotations per second, this will result in a velocity of 2880 (28800 position units per second).

17.2. Analog Potentiometer

When measuring the position of a 3.3V Analog Potentiometer, the position is measured as a 10 bit ADC value. A value of 1023 corresponds to 3.3V. A value of 0 corresponds to 0.0V.

The velocity units of a 3.3V Analog Potentiometer is the change in Analog Position per T_{velMea} ($T_{velMeas}$=0.1sec). For example if an Analog Potentiometer transitions from 0V to 3.3V (1023 units) in one second, the Analog Velocity will be 102.

17.3. Analog Encoder, “Analog-In Position”

Like 3.3V Analog Potentiometers, the 10 bit ADC is used to scale [0 V, 3.3 V] => [0, 1023]. However when the Analog Encoder “wraps around” from 1023 to 0, the Analog Position will continue to 1024. In other words, the sensor is treated as “continuous”.

The velocity units of a 3.3V Analog Encoder is the change in Analog Position per 100ms ($T_{velMeas}$=0.1sec). For example if an Analog Encoder transitions from 0V to 3.3V (1023 units) in one second, the Analog Velocity will be 102.

17.4. EncRise (a.k.a. Rising Counter)

Every rising edge seen on the Quadrature A pin is counted as a unit.
The velocity units is the change in RisingEdge Count per T_{velMea} ($T_{velMeas}$=0.1sec).
This mode is useful for single direction sensors (tachometer / gear-tooth sensor).

17.5. Duty-Cycle (Throttle)

The Talon SRX uses 10bit resolution for the output duty cycle. This means a -1023 represents full reverse, +1023 represents full forward, and 0 represents neutral.

The programming API made available in LabVIEW and C++/Java performs the scaling into percent, so the duty cycle resolution is not necessary for programming purposes. However when evaluating PIDF gain values, it is helpful to understand how the calculated output of the closed-loop is interpreted.

17.6. (Voltage) Ramp Rate

The Talon SRX natively represents Ramp Rate as the change in throttle per $T_{RampRate}$ ($T_{RampRate}$=10ms). Throttle is represented as a 10bit signed value (1023 is full forward, -1023 is full reverse). For example, if the robot application requires motor drive ramping from 0% to 100% to take one second of ramping, the result Ramp Rate would be ([1023 – 0] / 1000ms X $T_{RampRate}$) or 10 units.

The programming API made available in LabVIEW and C++/Java performs the scaling into appropriate units (Voltage or percent).

17.7. (Closed-Loop) Ramp Rate

The Closed Loop Ramp Rate that can be specified in the selected Motor Control Profile is measured in change in output throttle (from PIDF loop) per $T_{ClosedLoopRampRate}$ ($T_{ClosedLoopRampRate}$ =1ms). For example, if the selected Motor Control Profile requires motor drive ramping from 0% to 100% to take one second of ramping, the result Ramp Rate would be ([1023 – 0] / 1000ms X $T_{RampRate}$) or 1 unit.

The programming API made available in LabVIEW and C++/Java performs the scaling into appropriate units (Voltage or percent).

However when inspecting/setting the Closed-Loop Ramp Rate in the roboRIO webpage, the units will be shown in output throttle per $T_{ClosedLoopRampRate}$.

17.8. Integral Zone (I Zone)

The motor control profile contains Integral Zone (I Zone), which (when nonzero), is the maximum error where Integral Accumulation will occur during a closed-loop Mode. If the Closed-loop error is outside of the I Zone, “I Accum” is automatically cleared. This can prevent total instability due to integral windup, particularly when tweaking gains.

The units are in the same units as the selected feedback device (Quadrature Encoder, Analog Potentiometer, Analog Encoder, and EncRise).

17.9. Integral Accumulator (I Accum)

The accumulated sum of Closed-Loop Error. It is accumulated in line with Closed-Loop math every 1ms.

17.10. Reverse Feedback Sensor

Boolean signal for inverting the selected Feedback Sensor’s position and velocity. This is the preferred method for keeping the motor and sensor in phase for limit switch, soft limit, and closed-loop mode.

17.11. Reverse Closed-Loop Output

Boolean signal for inverting the output of the closed-loop PIDF controller. This signal is also used during slave-follower mode to drive slave Talon in the opposite direction of the master.

17.12. Closed-Loop Error

Calculated as the difference between target set point and the actual Sensor Position/Velocity.

The units are matched to Analog-In or Encoder depending on which “Feedback Device” and control mode (position vs. speed) is selected.

17.13. Closed-Loop gains

P gain is specified in throttle per error unit. For example, a value of 102 is ~9.97% (which is 102/1023) throttle per 1 unit of Closed-Loop Error.

I gain is specified in throttle per integrated error. For example, a value of 10 equates to ~0.97% for each accumulated error (Integral Accumulator). Integral accumulation is done every 1ms.

D gain is specified in throttle per derivative error. For example a value of 102 equates to ~9.97% (which is 102/1023) per change of Sensor Position/Velocity unit per 1ms.

F gain is multiplied directly by the set point passed into the programming API made available in LabVIEW and C++/Java. This allows the robot to feed-forward using the target set-point.

18. How is the closed-loop implemented?

The closed-loop logic is the same regardless of which feedback sensor or closed-loop mode is selected. The verbatim implementation in the Talon firmware is displayed below.
This includes…

- The logic for PIDF style closed-loop.
- Inverting the output of the closed-loop if enabled in API.
- Capping the output to positive values only IF using a single direction feedback sensor.
- Closed- Loop Ramp Rate, ramping the output if enabled.

Note: The `PID_Mux_Unsigned` and `PID_Mux_Sign` routines are merely multiply functions.

```
/**
 * 1ms process for PIDF closed-loop.
 * @param pid ptr to pid object
 * @param pos signed integral position (or velocity when in velocity mode).
 *                  The target pos/velocity is ramped into the target member from caller's 'in'.
 *                  If the CloseLoopRamp in the selected Motor Controller Profile is zero then
 *                  there is no ramping applied. (throttle units per ms)
 *                  PIDF is traditional, unsigned coefficients for P,i,D, signed for F.
 *                  Target pos/velocity is feed forward.
 *
 *                  Izone gives the abilty to autoclear the integral sum if error is wound up.
 * @param revMotDuringCloseLoopEn nonzero to reverse PID output direction.
 * @param oneDirOnly when using positive only sensor, keep the closed-loop from outputing negative throttle.
 */
void PID_Calc1Ms(pid_t * pid, int32_t pos,uint8_t revMotDuringCloseLoopEn, uint8_t oneDirOnly)
{
          /* grab selected slot */
          MotorControlProfile_t * slot = MotControlProf_GetSlot();
          /* calc error : err = target - pos*/
          int32_t err = pid->target - pos;
          pid->err = err;
          /*abs error */
          int32_t absErr = err;
          if(err < 0)
                    absErr = -absErr;
          /* integrate error */
          if(0 == pid->notFirst){
                    /* first pass since reset/init */
                    pid->iAccum = 0;
                    /* also tare the before ramp throt */
                    pid->out = BDC_GetThrot(); /* the save the current ramp */
          }else if((!slot->IZone) || (absErr < slot->IZone) ){
                    /* izone is not used OR absErr is within iZone */
                    pid->iAccum += err;
          }else{
                    pid->iAccum = 0;
          }
          /* dErr/dt */
          if(pid->notFirst){
                    /* calc dErr */
                    pid->dErr = (err - pid->prevErr);
          }else{
                    /* clear dErr */
                    pid->dErr = 0;
          }
          /* P gain X the distance away from where we want */
          pid->outBeforRmp = PID_Mux_Unsigned(err, slot->P);
          if(pid->iAccum && slot->I){
                    /* our accumulated error times I gain.  If you want the robot to creep up then pass a nonzero Igain */
                    pid->outBeforRmp += PID_Mux_Unsigned(pid->iAccum, slot->I);
          }
          /* derivative gain, if you want to react to sharp changes in error (smooth things out). */
          pid->outBeforRmp += PID_Mux_Unsigned(pid->dErr,   slot->D);
          /* feedforward on the set point */
          pid->outBeforRmp += PID_Mux_Signed(pid->target,   slot->F);
          /* arm for next pass */
          {
                    pid->prevErr = err; /* save the prev error for D */
                    pid->notFirst = 1;  /* already serviced first pass */
          }
          /* if we are using one-direction sensor, only allow throttle in one dir.
```

```
                If it's the wrong direction, use revMotDuringCloseLoopEn to flip it */
        if(oneDirOnly){
                if(pid->outBeforRmp < 0)
                        pid->outBeforRmp = 0;
        }
        /* honor the direction flip from control */
        if(revMotDuringCloseLoopEn)
                pid->outBeforRmp = -pid->outBeforRmp;
        /* honor closeloopramprate, ramp out towards outBeforRmp  */
        if(0 != slot->CloseLoopRampRate){
                if(pid->outBeforRmp >= pid->out){
                        /* we want to increase our throt */
                        int32_t deltaUp = pid->outBeforRmp - pid->out;
                        if(deltaUp > slot->CloseLoopRampRate)
                                deltaUp = slot->CloseLoopRampRate;
                        pid->out += deltaUp;
                }else{
                        /* we want to decrease our throt */
                        int32_t deltaDn = pid->out - pid->outBeforRmp;
                        if(deltaDn > slot->CloseLoopRampRate)
                                deltaDn = slot->CloseLoopRampRate;
                        pid->out -= deltaDn;
                }
        }else{
                pid->out = pid->outBeforRmp;
        }
}
```

19. Motor Safety Helper

The Motor Safety feature works in a similar manner as the other motor controllers. The goal is to set an expiration time to a given motor controller, such that, if the `Set()/set()` routine is not called within the expiration time, the motor controller will disable. Additionally the DS will report the error and the roboRIO Web-based Configuration Self-Test will report kDisabled as the mode. As a result, the set routine must be called periodically for sustained motor drive when motor safety is enabled.

One example where this feature is useful is when laying breakpoints with the debugger while the robot is enabled and moving. Ideally when a breakpoint lands, its safest to disable motor drive while the developer performs source-level debugging.

19.1. Best practices

Be sure to test that the time between enabling Motor Safety features, and the first `Set()/set()` call is small enough to not risk accidently timing out. Calling `Set()/set()` immediately after enabling the feature can be used to ensure transitioning into the enabled modes doesn't intermittently cause a timeout.

Even if tripping the motor-safety expiration time is not an expected condition, it's best to re-enable the motors somewhere in the source so that the timeouts can be reset easily, for example in AutonInit()/TeleopInit(). That way normal robot functionality can be safely resumed after a motor controller expires (usually during source-level debugging).

Additionally if source-level debugging is not required (for example during a competition or if logging-style debugging is preferred) the motor-safety enable can be turned off.

19.2. C++ example

`SetSafeyEnabled()` can be used to turn on this feature. `SetExpiration()` can be used to set the expiration time. The default expiration time is typically 100ms.

```
class Robot: public IterativeRobot
{
private:
    CANTalon *_talons[20]; //!< Create a bunch of Talons
    Joystick _joy;
    static const int masterId = 2; //!< Which Talon device ID to make the master.

public:
    Robot() : _joy(0)
    {
        /* create a bunch of talons, say 20 of them.  Doesn't matter if 20 are actually wired or not. */
        for(int i=0;i<20;++i){
            _talons[i] = new CANTalon(i);
            /* make every Talon follow master ID */
            _talons[i]->SetControlMode(CANSpeedController::kFollower);
            _talons[i]->Set(masterId);
        }
    }
    void TeleopInit()
    {
        /* just in case we already safety-timed out previously, when we re-enter teleop we
            need to re-Enable the motor controller, otherwise it will stay timed out. */
        _talons[masterId]->EnableControl();

        /* turn on safety enable features */
        _talons[masterId]->SetSafetyEnabled(true);
        _talons[masterId]->SetExpiration(0.100);
        _talons[masterId]->Set(0);
    }
    void TeleopPeriodic()
    {
        /* grab some gamepad values */
        double dThrot = -1*_joy.GetY();
        double bBtn1 = _joy.GetRawButton(1);

        /* make the master Percent Vbus.  you can do this once or every loop, it doesn't hurt anything */
        _talons[masterId]->SetControlMode(CANSpeedController::kPercentVbus);

        if(bBtn1){
            /* button is pressed, don't update motor to negative test safety features */
        }else{
            /* button not pressed, keeping updating ~20ms per set */
            _talons[masterId]->Set(dThrot);
        }
    }
};
```

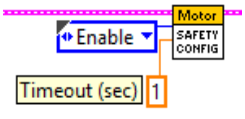
19.3. Java example

`setSafeyEnabled()` can be used to turn on this feature. `setExpiration()` can be used to set the expiration time. The default expiration time is typically 100ms.

```
10 public class Robot extends IterativeRobot {
11
12     CANTalon [] _talons = new CANTalon[20]; //!< Create a bunch of Talons
13     Joystick _joy = new Joystick(0);
14     int masterId = 2; //!< Which Talon device ID to make the master.
15
16     public Robot(){
17         /* create a bunch of talons, say 20 of them.  Doesn't matter if 20 are actually wired or not. */
18         for(int i=0;i<20;++i){
19             _talons[i] = new CANTalon(i);
20             /* make every Talon follow master ID */
21             _talons[i].changeControlMode(ControlMode.Follower);
22             _talons[i].set(masterId);
23         }
24     }
25
26     public void teleopInit(){
27         /* just in case we already safety-timed out previously, when we re-enter teleop we
28         need to re-Enable the motor controller, otherwise it will stay timed out. */
29         _talons[masterId].enableControl();
30
31         /* turn on safety enable features */
32         _talons[masterId].setSafetyEnabled(true);
33         _talons[masterId].setExpiration(0.100);
34         _talons[masterId].set(0);
35     }
36
37     /**
38      * This function is called periodically during operator control
39      */
40     public void teleopPeriodic() {
41         /* grab some gamepad values */
42         double dThrot = -1*_joy.getY();
43         boolean bBtn1 = _joy.getRawButton(1);
44
45         /* make the master Percent Vbus.  you can do this once or every loop, it doesn't hurt anything */
46         _talons[masterId].changeControlMode(ControlMode.PercentVbus);
47
48         if(bBtn1){
49             /* button is pressed, don't update motor to negative test safety features */
50         }else{
51             /* button not pressed, keeping updating ~20ms per set */
52             _talons[masterId].set(dThrot);
53         }
54     }
55
56
57
58 }
```

19.4. LabVIEW Example

The `Motor SAFETY CONFIG VI` can be used to turn on this feature. Select “Enable” for the mode and specify the timeout in seconds.



20. Going deeper - How does the framing work?

The Talon periodically transmits four status frames with sensor data at the given periods. This ensures that certain signals are always available with a deterministic update rate. This also keeps bus utilization stable.

Similarly the control frame sent to the Talon SRX is periodic and contains almost all the information necessary for all control modes.

Although the frame rates are default to ensure stable CAN bandwidth, there *may* be available API to override the frame rates for performance reasons. If this is done, be sure to check the CAN performance metrics to ensure custom settings don't exceed the available CAN bandwidth, see "CAN bus Utilization and Performance metrics".

20.1. General Status

The General Status frame has a default period of 10ms, and provides…
-Closed Loop Error: the closed-loop target minus actual position/velocity.
-Throttle: The current 10bit motor output duty cycle (-1023 full reverse to +1023 full forward).
-Forward Limit Switch Pin State
-Reverse Limit Switch Pin State
-Fault bits
-Applied Control Mode
… These signals are accessible in the various get functions in the programming API.

20.2. Feedback Status

The Feedback Status frame has a default period of 20ms, and provides…
-Sensor Position: Position of the selected sensor
-Sensor Velocity: Velocity of the selected sensor
-Motor Current
-Sticky Faults
-Brake Neutral State
-Motor Control Profile Select
… These signals are accessible in the various get functions in the programming API.

20.3. Quadrature Encoder Status

The Quadrature Encoder Status frame has a default period of 100ms.
-Encoder Position: Position of the quadrature sensor
-Encoder Velocity: Velocity of the selected sensor
-Number of rising edges counted on the Index Pin.
-Quad A pin state.
-Quad B pin state.
-Quad Index pin state.
… These signals are accessible in the various get functions in the programming API.

The quadrature decoder is always engaged, whether the feedback device is selected or not, and whether a quadrature encoder is actually wired or not. This means that the Quadrature Encoder signals are always available in programming API regardless of how the Talon is used. The 100ms update rate is sufficient for logging, instrumentation and debugging. If a faster update rate is required the robot application can select the appropriate sensor and leverage the Sensor Position and Sensor Velocity.

20.4. Analog Input / Temperature / Battery Voltage Status

The Analog/Temp/BattV status frame has a default period of 100ms.
-Analog Position: Position of the selected sensor
-Analog Velocity: Velocity of the selected sensor
-Temperature
-Battery Voltage
… These signals are accessible in the various get functions in the programming API.

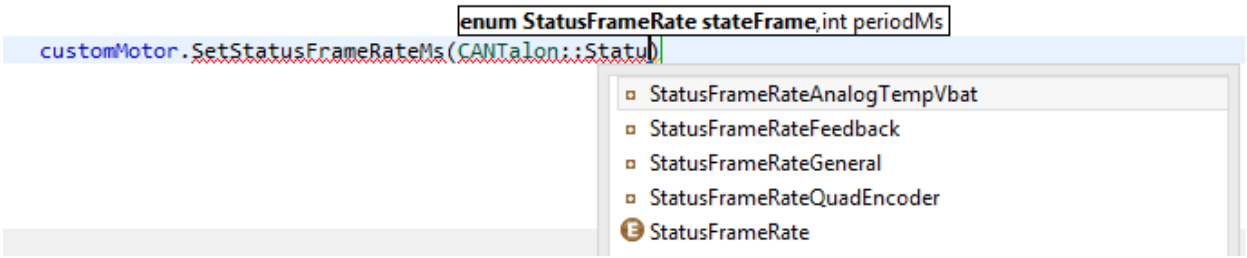
The Analog to Digital Convertor is always engaged, whether the feedback device is selected or not, and whether an analog sensor is actually wired or not. This means that the Analog In signals are always available in programming API regardless of how the Talon is used. The 100ms update rate is sufficient for logging, instrumentation and debugging. If a faster update rate is required the robot application can select the appropriate sensor and leverage the Sensor Position and Sensor Velocity.

20.5. Modifying Status Frame Rates

The frame rates of these signals may be modifiable through programming API.

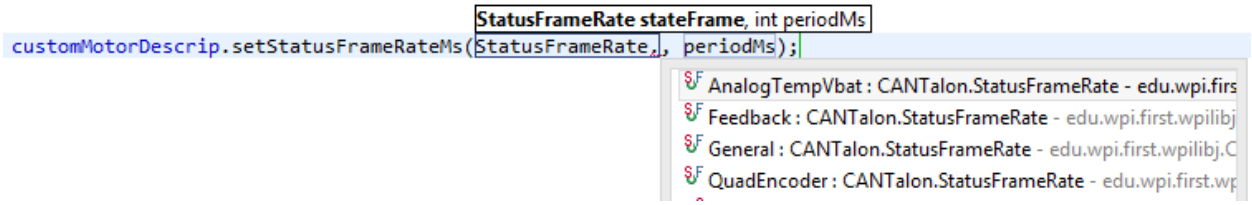
20.5.1. C++

The `SetStatusFrameMs()` function can be used to modify the frame rate period of a particular Status Frame. Use the **StatusFrameRate…** enumerations to specify which frame period to modify.



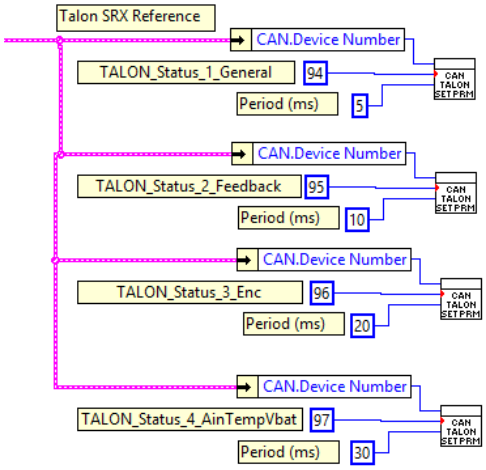
20.5.2. Java

The `setStatusFrameMs()` function can be used to modify the frame rate period of a particular Status Frame. Use the **StatusFrameRate…** enumerations to specify which frame period to modify.

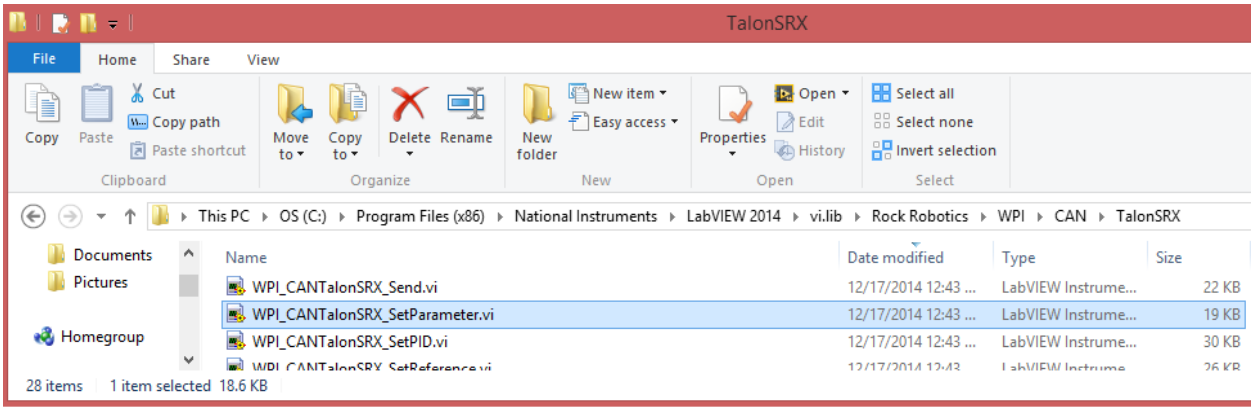


20.5.3. LabVIEW Example

Although there is no explicit VI for modifying the Status Frame Rates, modifying the frame rates can be accomplished with the generic `CAN TALON SETPRM`.



The `WPI_CANTalonSRX_SetParameter.vi` can be drag and dropped from where it resides.



20.6. Control Frame

The Talon is primarily controlled by one periodic control frame. The default period of this frame is 10ms. The control frame provides the Talon…
-which Motor Control Profile Slot to use.
-which control mode (position, velocity, duty cycle, slave mode)
-which feedback sensor to use
-if the feedback sensor should be reversed
-if the closed-loop output should be reversed
-the target/set point or duty cycle or which Talon to follow
-the (voltage) ramp rate
-brake neutral mode override if specified
-limit switch overrides if specified
… These signals are accessible in the various set functions in the programming API.

20.7. Modifying the Control Frame Rate

Depending on the initial release of programming API, the frame rates of these signals may be modified through programming API. The CANTalon constructor contains a second parameter to specify control frame period in milliseconds.

21. Functional Limitations

Functional Limitations describe behavior that deviates than what is documented. Feature additions and improvements are always possible thanks to the field-upgrade features of the Talon SRX.

21.1. Firmware 1.1-1.4: Voltage Compensation Mode is not supported.

Feature was not prioritized for FRC 2015 season.

21.2. Firmware 1.1-1.4: Current Closed-Loop Mode is not supported.

Feature was not prioritized for FRC 2015 season.

21.3. Firmware 1.1-1.4: EncFalling Feedback device not supported.

Feature was not prioritized for FRC 2015 season. Firmware does support EncRising Mode (a.k.a. Rising Edge Counter).

21.4. Firmware 1.1-1.4: `ConfigMaxOutputVoltage()` not supported.

Feature was not prioritized for FRC 2015 season.

21.5. Firmware 1.1-1.4: `ConfigFaultTime()` not needed

Firmware only disables drive for limit faults and soft limits (which are time invariant). Motor drive is not disabled due to current, temp, or battery voltage, therefore there is no fault time.

21.6. Firmware 1.1: Changes in Limit Switch “Normally Open” vs “Normally Closed” may require power cycle during a specific circumstance.

In the specific situation where programming API overrides a limit switch enable (to true or false), then changes the NO/NC mode of the same limit switch using programming API, the setting will not take effect until the motor controller is power cycled, or until the limit switch is no longer overridden.

The “first” time a new Talon SRX’s NO/NC setting is changed programmatically, power cycle the Talon so the setting takes effect. After the initial power cycle, new NO/NC setting will be loaded correctly and match what the robot controller is requesting, therefore Talon will honor the new NO/NC state from then on.

If not setting NO/NC state programmatically, then no symptoms are observed that deviate from reference manual. Changing the NO/NC state in the the roboRIO Web-based Configuration works as expected.

This is fixed in 1.4.

21.7. LabVIEW: EncRising Feedback mode not selectable.

Release software of LabVIEW does not provide option to select EncRising Mode.

21.8. LabVIEW/C++/Java API: `ConfigEncoderCodesPerRev()` is not supported.

Talon does not configure units. Instead the quadrature units are always in 4X mode.

21.9. LabVIEW/C++/Java API: `ConfigPotentiometerTurns()` is not supported.

Talon does not configure units. Instead the 3.3V ADC is 10 bit, therefore 0 => 3.3V scales to 0 => 1023 units.

21.10. Java: Once a Limit Switch is overridden, they can't be un-overridden.

This does not cause any observable symptoms, just an inconsistency between C++ and Java API.

21.11. LabVIEW: Modifying status frame rate is not available.

See Section 20.5.3 for an example workaround as well as Functional Limitation Section 21.14.

21.12. LabVIEW: Modifying control frame rate is not available.

This will not be available in the initial season release.

21.13. Firmware 1.1: After selecting "Analog Encoder", "Sensor Position" does not reliably decode when sensor wraps around (3.3V => 0V).

Sensor Position may not travel above 1023 or below 0, despite selecting "Analog Encoder" and spinning the sensor in one direction in a continuous fashion.

This is fixed in 1.4.

21.14. LabVIEW: Certain SRX VI's running in parallel can affect the GET PID VI signals.

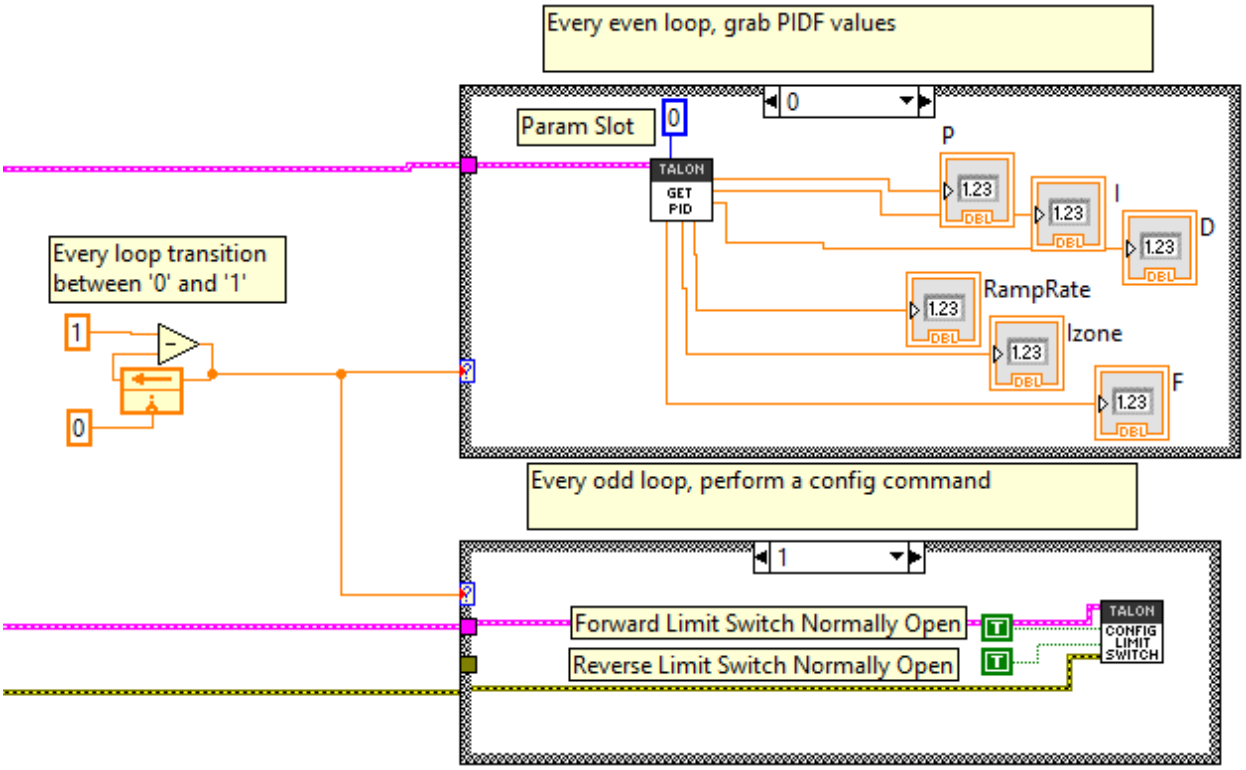
Avoid using the following VIs in “parallel” to reading the signals provided by `GET PID`.

Affected VIs:

```
RESET INTEGRAL ACCUM,
CONFIG LIMIT SWITCH,
CONFIG SOFT LIMIT,
SET PID,
CAN TALON SETPRM,
```

For example, use a state variable to control whether GET PID is used, or the other mentioned VIs are used per periodic loop. Otherwise the GET PID signals may erroneously return zero.

Example Workaround



21.15. C++: There is no method to reverse the output of a slave Talon SRX.

The initial release of WPILIB C++ does not have a method for setting the "Reverse Closed-Loop output" signal. This signal is useful for reversing the output of a slave Talon SRX, ensuring it drives in the opposite direction of its master Talon SRX.

Additionally when using a single-direction sensor in EncRising mode (Rising Counter) this is the preferred method for keeping motor and sensor in phase since "Reverse Feedback Sensor" must be false for single-direction sensors.

The following example can be used to work around this limitation by using the core class `CanTalonSrx` to directly reverse the output signal. Notice the additional include for `"ctre/CanTalonSRX.h"` and the use of `SetRevMotDuringCloseLoopEn()` to accomplish reversing the output.

Although using `CanTalonSRX` is *generally* not recommended (`CANTalon` is the top-level class designed for team-use) this functional limitation is an example where using the lower level class is beneficial.

Example Workaround – Inverting a Slave Talon SRX

Robot.cpp

```
#include "WPILib.h"
#include "ctre/CanTalonSRX.h"  /* use this to grab the underlying core class */
class Robot: public IterativeRobot
{
public:
    Joystick * joy0; /* the first gamepad */
    CANTalon * tal1; /* pointer to a CANTalon */
    CanTalonSRX * tal2; /* pointer to a CanTalonSRX - the low level class */
    Robot()
    {
        tal1 = new CANTalon(1);     /* master Talon device id 1 */
        tal2 = new CanTalonSRX(2);  /* slave Talon device id 2 */
        joy0 = new Joystick(0); /* first gamepad */

        /* just use the CanTalonSRX class since WPILIB is missing the reverseOuput() func */
        tal2->SetModeSelect(CanTalonSRX::kMode_SlaveFollower);  /* Talon2 will follow another Talon */
        tal2->SetDemand(1);                                      /* Talon2 will follow Talon1 */
        tal2->SetRevMotDuringCloseLoopEn(1);                    /* ClosedLoopOut/SlaveOut reverse set to "true" */
    }

    void TeleopPeriodic()
    {
        /* create vars */
        double leftYaxis;
        /* get left axis Y */
        leftYaxis = joy0->GetY(Joystick::kLeftHand);
        /* Left Y => Talon 1 */
        tal1->Set(leftYaxis);

    }
};

START_ROBOT_CLASS(Robot);
```

Note: This example also demonstrates using heap class pointers instead of regular member variables in the interest of covering different methods for allocating objects. Teams may use non-pointer variables if desired.

21.16. Firmware <0.36: Limit Switch Faults and Soft Limit Faults may cause Talon SRX to disable for approximately two seconds during the “first time”.

In the specific situation where a particular limit switch or particular soft limit fault trips for the “first time” when using a Talon SRX with pre-FRC2015-kickoff firmware, the Talon SRX may blink orange for a two second period of time, during which it behaves as though it’s disabled. This can only happen when a Talon SRX is initially out-of-the-box or when a Talon SRX’s sticky faults have been cleared (using the roboRIO web-based configuration or through programming API). The cause is due to the way that firmware earlier than 0.36 saves the sticky faults in persistent memory.

Since teams using CAN are required to update to at least 1.1, this functional limitation will not occur.

Also Talon SRXs used with PWM have no method for clearing sticky faults, so this symptom will only appear once and then never occur again. Additionally PWM-use Talons can easily be firmware updated using the method described in Talon SRX User’s Guide Section 1.3.4.2.

Fixed in 0.36.

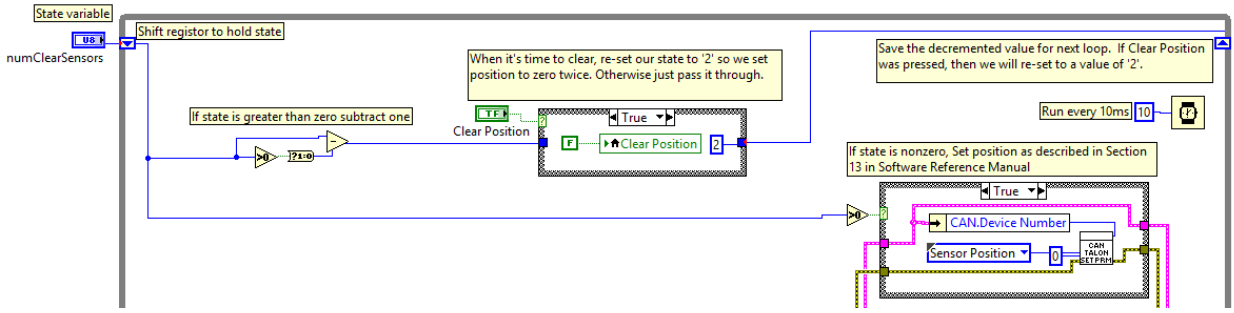
21.17. Firmware 1.4: When setting the “Sensor Position” of an analog encoder, multiple set commands are required.

In the specific situation of setting the “Sensor Position” when the selected feedback device is an analog encoder, the robot application will have to send two set commands to reliably change the sensor position. Additionally there must be at least 9 ms between the two set commands.

The symptom occurs when a sensor position is changed by a large enough value to cause a false detection of an analog encoder wraparound. By re-setting the sensor position to the same new value a second time after the false wrap around is detected, the analog encoder position can be reliably modified.

In C++/Java this can be done by calling the `SetPosition()`/`setPosition()` function twice with the same parameter, and ensuring there is at least 9 ms between the calls.

In LabVIEW this can be accomplished by using state (shift register) to count the number times to consecutively set the Sensor Position. The following example demonstrates clearing the sensor position when the “Clear Position” front panel button is pressed.



22. CRF Firmware Revision Information

CRF Rev	Date	Description
1.4	20-Jan-2015	Functional Limitation 21.6 fixed. Functional Limitation 21.13 fixed. Analog Encoder/Potentiometer Velocity uses a rolling window average to reduce noise. Analog Encoder/Potentiometer Position averaged to reduce noise.
1.1	26-Dec-2014	Initial Release for 2015 FRC Season

23. Document Revision Information

Rev	Date	Description
1.3	1-Feb-2015	-Section 6.4 added. -Section 12.2.3 added. -Section 16.24 added. -Section 21.16 added. -Section 21.17 added.
1.2	23-Jan-2015	-Section 21.15 added. Reversing a slave Talon in C++. -Section 3.6 Added for changing Talon SRX mode.
1.1	20-Jan-2015	-Section 22 moved to Section 23. -New Section 22 added for CRF Firmware. -Updates relating to CRF 1.4. -Section 21.14 Added. Workaround for GET PID VI. -Section 20.5.3 Added. Example for LabVIEW status frame modification. -Section 2.5 Added. Custom Device Names. -Section 13, Clarifying statement added for CRF 1.4.
1.0	26-Dec-2014	-Initial Release for 2015 FRC Season