Phoenix 6

CTR Electronics

Sep 18, 2023

Getting Started

Welcome to the Phoenix 6 documentation. Individuals looking for *Phoenix 5* documentation may locate it here.

The Phoenix 6 software framework allows you to control and configure your CTR Electronics Phoenix 6 Devices. Phoenix 6 represents a complete rewrite of the software framework over the existing Phoenix 5 framework. With Phoenix 6, users have access to many new features that expand the control the user has over their devices.

CTR Electronics Blog For news and updates about your CTR Electronics device, please check out our blog.

Changelog A changelog containing API, Tuner and Firmware changes is available here.

Migration Guide A Phoenix 5 migration guide is available *here*.

Installation Installation instructions for Phoenix API & Tuner.

Phoenix Tuner Documentation that introduces the companion application to manage your CTR Electronics devices.

Hardware Reference Documentation for device specific configuration, troubleshooting and setup instructions.

API Reference Documentation and details on using the CTR Electronics device API. This includes usage of signals, configs, control requests, etc.

Examples Software API examples for controlling your devices.

Troubleshooting Common troubleshooting for hardware or software problems.

Installing Phoenix 6

Installation of Phoenix 6 is comprised of a few steps

- *Installing API*
- *Installing Tuner*
- *Running the Diagnostic Server*
- *Updating Device Firmware*
- *Device Licensing*

1.1 API Installation

Phoenix 6 currently supports Java and C++ for development.

### 1.1.1 System Requirements

The following targets are supported:

- NI roboRIO
- Windows 10/11 x86-64
- Linux x86-64 (desktop)
    - Ubuntu 22.04 or newer
    - Debian Bullseye or newer
- Linux ARM32 and ARM64 (Raspberry Pi, NVIDIA Jetson)
    - Ubuntu 20.04 or newer
    - Debian Bullseye or newer
- macOS (regular simulation **only**)

1.1.2 Offline

Important: Users on non-Windows devices should skip to the *Online* installation instructions.

---

1. Download the Phoenix Framework Installer
2. Navigate through the installer, ensuring applicable options are selected
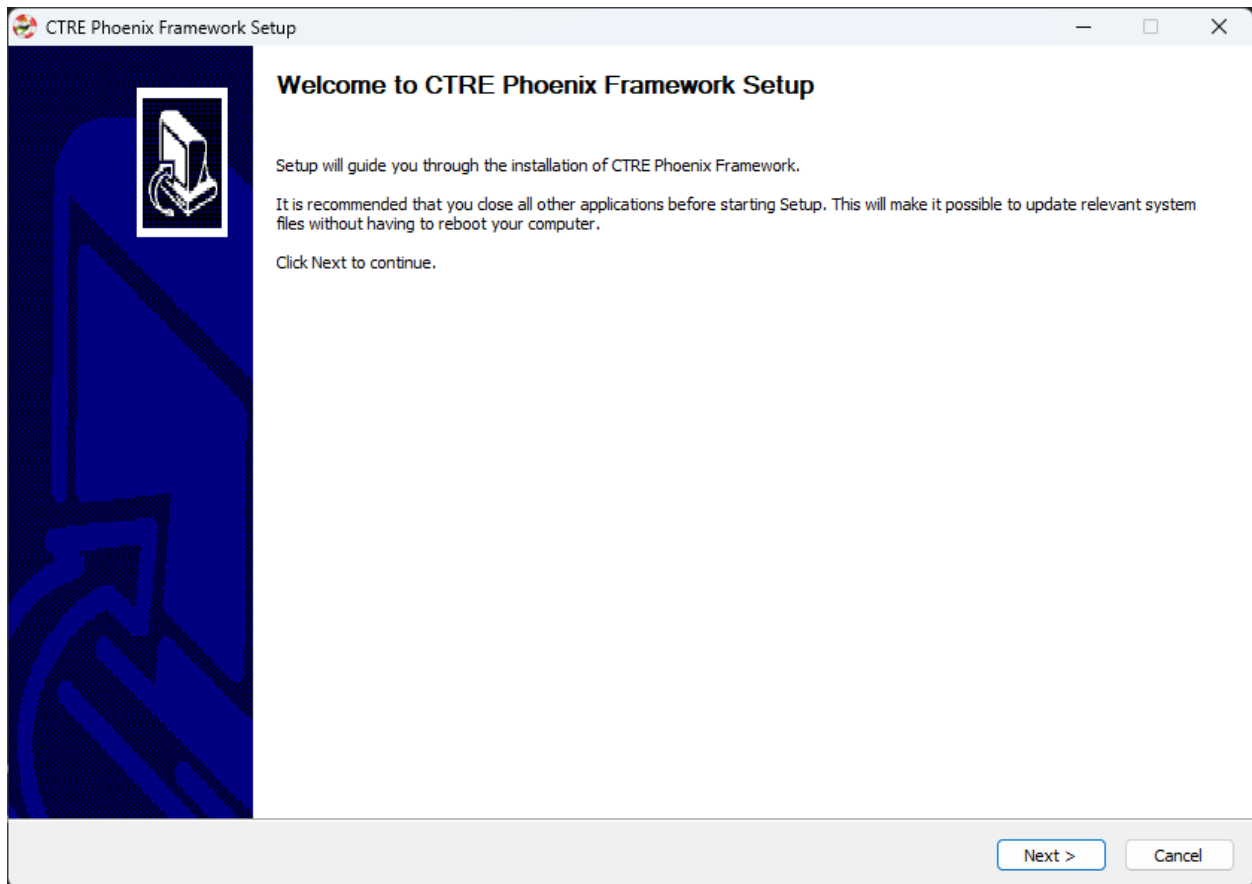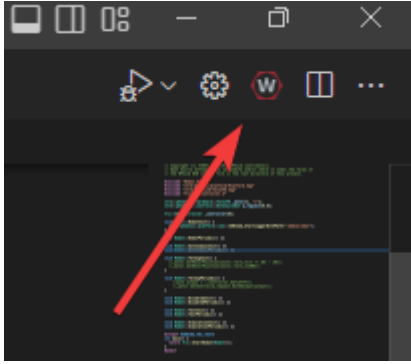

3. Apply the vendordep via WPILib VS Code Adding Offline Libraries

1.1.3 Online

Users in FRC can install Phoenix without an installer using WPILib's Install New Libraries functionality in VS Code. This requires the user to have an installation of WPILib on their machine.

To begin, open WPILib VS Code and click on the WPILib icon in the top right.

Then type *Manage Vendor Libraries* and click on the menu option that appears. Click *Install new libraries (online)* and a textbox should appear. Follow the remaining instructions below on pasting the correct link into the textbox.

FRC (v6 Only)

Important: This vendordep is for robot projects that are **only** using devices with Phoenix 6 firmware.

Paste the following URL in WPILib VS Code *Install new libraries (online)*:

- https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/
  Phoenix6-frc2023-latest.json

FRC (v6 & Phoenix 5)

Important: This vendordep is for robot projects that are using **both** Phoenix 6 devices & Phoenix 5 devices.

Paste the following URL in WPILib VS Code *Install new libraries (online)*:

- https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/
  Phoenix6And5-frc2023-latest.json

Important: Devices on Phoenix 6 firmware **must** use the Phoenix 6 API. Device on Phoenix 5 firmware **must** use the Phoenix 5 API.

---

non-FRC (Linux)

Phoenix 6 is distributed through our APT repository. Begin with adding the repository to your APT sources.

```
sudo curl -s --compressed -o /usr/share/keyrings/ctr-pubkey.gpg "https://deb.ctr-
↪electronics.com/ctr-pubkey.gpg"
sudo curl -s --compressed -o /etc/apt/sources.list.d/ctr<year>.list "https://deb.ctr-
↪electronics.com/ctr<year>.list"
```

Note: `<year>` should be replaced with the year of Phoenix 6 software for which you have purchased licenses.

After adding the sources, Phoenix 6 can be installed and updated using the following:

```
sudo apt update
sudo apt install phoenix-pro
```

Tip: To get a robot application up and running quickly, check out our non-FRC Linux example.

1.2 Tuner X Installation

Phoenix Tuner X is a modern version of the legacy Phoenix Tuner v1 application that is used to configure CTRE Phoenix CAN devices.

Phoenix Tuner X is supported on Android, Windows 10 (build 1903+), and Windows 11. Installation is available from the respective OS stores.

- Windows: https://apps.microsoft.com/store/detail/phoenix-tuner/9NVV4PWDW27Z
- Android: https://play.google.com/store/apps/details?id=com.ctre.phoenix_tuner

Running the Diagnostic Server

Phoenix Tuner utilizes an on-device HTTP server called Phoenix Diagnostic Server to communicate with the device. The user can run the diagnostic server through one of two ways.

2.1 1: Deploying a Robot Program

Phoenix Diagnostics will automatically run assuming you have instantiated a CTR Electronics device in your robot program. This can be as simple as having a motor declared somewhere in your program.

Note: The ID of the device does not need to be valid to run diagnostics.

Java

```
private TalonFX m_motor = new TalonFX(0);
```

C++

```
hardware::TalonFX m_talonFX{0};
```

When the program runs, it will print text to the console similar to the below

Note: WPILib users will see this text in the Driver Station or RioLog

```
[phoenix] Starting Standalone Diagnostics Server (23.1.0-Jun  2 2023,23:17:09)
[phoenix-diagnostics] Server 2023.1.0 (Jun  2 2023, 23:17:56) running on port: 1250
```

2.2 2: Running Temporary Diagnostic Server

Alternatively, users can *run a temporary diagnostic server* in Tuner X. The temporary diagnostic server will only run until the next reboot of the target system.

3

Configuring your Device

All CTR Electronics devices have an ID that distinguishes multiple devices of the same type on the same CAN bus. This should be configured to the user's preference. Firmware upgrading is also generally **required** for each new release of Phoenix 6 API. Please visit the relevant Tuner pages on how to complete these steps.

Updating Firmware   *Click here to visit the section on updating your device firmware*

Configuring IDs   *Click here to visit the section on configuring your device ID*

4

Phoenix 6 currently offers the following features and will further expand.

4.1 Phoenix 6

The following features are available for free in the Phoenix 6 API.

4.1.1 Comprehensive API

- Device signal getters return a *StatusSignal* object, expanding the functionality of status signals.
- Control devices with an extensive list of flexible, strongly-typed *control request objects*.

4.1.2 Canonical Units

- Uses the popular Units library for C++ and standardizes on SI units.
- Signals are documented with the unit type and the minimum and maximum values.

4.1.3 Improved Device Control

- New and improved *control output types* and *closed-loop* configuration.
- Improved *Motion Magic®* with jerk control and support for modifying the profile on the fly.
- Kalman-based algorithms to reduce latency while maintaining smooth data.

4.1.4 Enhanced Support for CAN FD

- Improved CAN FD framing further reduces any *CAN bus utilization* issues.
- Larger CAN frames allow for the addition of more advanced features.

4.1.5 New Tuner X Self Tests

- Detailed and resolute self tests to improve debugging.

4.1.6 Free High-Fidelity Simulation

- Simulation closely follows the behavior of real hardware.
- Write unit-tests for your robot code, and make sure the robot works before deploying.

4.1.7 Continuous Wrap Mode

- Takes the shortest path for continuous mechanisms.
- Ideal for mechanisms such as Swerve Drive Steer.

4.2 Phoenix Pro

Certain Phoenix 6 features require the device or CANivore to be *Pro licensed*. The list of features that require licensing is available below.

4.2.1 Field Oriented Control (FOC)

- ~15% increase in peak power.
- Increased torque output; faster acceleration and higher speeds under load.
- Greater efficiency; the motor draws less current for the same output power, increasing battery life.
- Support for direct *torque control*.

4.2.2 Time Base Synchronization

- Using *CANivore Timesync*, signals from all devices are sampled and published to the CAN bus at the same time.
- API can synchronously wait for data from multiple devices on a CANivore to arrive.

4.2.3 Fused CANcoder

- *Fuse a CANcoder* with the motor's internal rotor, getting absolute data all the time while using the fast internal sensor for closed looping.

4.3 Feature Breakdown

A full comparison of features between the free Phoenix 6 API and Phoenix Pro is shown below.

Feature	Phoenix 6	Phoenix Pro
Canonical Units	x	x
Improved Bus Utilization	x	x
CANcoder Always Absolutely	x	x
Kalman-based Velocity	x	x
Synchronous Wait for Data	x	x
System + CANivore Timestamps	x	x
Explicit Control Requests	x	x
Continuous Wrap Mode	x	x
Improved Self-Test Snapshot	x	x
Tuner X Improved Plotting	x	x
Time-Synced Signal Publishing		x
Field Oriented Control (FOC)		x
CAN Device Timestamps		x
Fused CANcoder + TalonFX		x

API Migration

This section serves as a "cheat sheet" of commonly-used functions in Phoenix 5 and their equivalents in Phoenix 6.

- *Configuration*
    - Configuring device configs in robot code
- *Status Signals*
    - Using status signals to retrieve sensor data from devices
- *Control Requests*
    - Using control requests to control the functionality of actuators, such as the TalonFX
- *Closed-Loop Control*
    - Configuring and using closed-loop control requests
- *Feature Replacements*
    - Other features replaced or improved upon in Phoenix 6

5.1 Configuration

Phoenix 6 simplifies the configuration process through the use of device-specific `Configuration` classes, as well as configuration groups.

Note: For more information about configuration in Phoenix 6, see *Configuration*.

5.1.1 Applying Configs

v

Java

```java
// set slot 0 gains
// 50 ms timeout on each config call
m_motor.config_kF(0, 0.05, 50);
m_motor.config_kP(0, 0.046, 50);
m_motor.config_kI(0, 0.0002, 50);
m_motor.config_kD(0, 0.42, 50);
```

C++

```cpp
// set slot 0 gains
// 50 ms timeout on each config call
m_motor.Config_kF(0, 0.05, 50);
m_motor.Config_kP(0, 0.046, 50);
m_motor.Config_kI(0, 0.0002, 50);
m_motor.Config_kD(0, 0.42, 50);
```

v

Java

```java
// set slot 0 gains
var slot0Configs = new Slot0Configs();
slot0Configs.kV = 0.12;
slot0Configs.kP = 0.11;
slot0Configs.kI = 0.5;
slot0Configs.kD = 0.001;

// apply gains, 50 ms total timeout
m_talonFX.getConfigurator().apply(slot0Configs, 0.050);
```

C++

```cpp
// set slot 0 gains
configs::Slot0Configs slot0Configs{};
slot0Configs.kV = 0.12;
slot0Configs.kP = 0.11;
slot0Configs.kI = 0.5;
slot0Configs.kD = 0.001;

// apply gains, 50 ms total timeout
m_talonFX.GetConfigurator().Apply(slot0Configs, 50_ms);
```

Factory Defaulting Configs

V

Java

```
// user must remember to factory default if they configure devices in code
m_motor.configFactoryDefault();
```

C++

```
// user must remember to factory default if they configure devices in code
m_motor.ConfigFactoryDefault();
```

V

Java

```
// any unmodified configs in a configuration object are *automatically* factory-
↪defaulted;
// user can perform a full factory default by passing a new device configuration␣
↪object
m_motor.getConfigurator().apply(new TalonFXConfiguration());
```

C++

```
// any unmodified configs in a configuration object are *automatically* factory-
↪defaulted;
// user can perform a full factory default by passing a new device configuration␣
↪object
m_motor.GetConfigurator().Apply(TalonFXConfiguration{});
```

5.1.2 Retrieving Configs

v

Java

```java
// a limited number of configs have configGet* methods;
// for example, you can get the supply current limits
var supplyCurLim = new SupplyCurrentLimitConfiguration();
m_motor.configGetSupplyCurrentLimit(supplyCurLim);
```

C++

```cpp
// a limited number of configs have ConfigGet* methods;
// for example, you can get the supply current limits
SupplyCurrentLimitConfiguration supplyCurLim{};
m_motor.ConfigGetSupplyCurrentLimit(supplyCurLim);
```

v

Java

```java
var fx_cfg = new TalonFXConfiguration();
// fetch *all* configs currently applied to the device
m_motor.getConfigurator().refresh(fx_cfg);
```

C++

```cpp
configs::TalonFXConfiguration fx_cfg{};
// fetch *all* configs currently applied to the device
m_motor.GetConfigurator().Refresh(fx_cfg);
```

5.2 Status Signals

Phoenix 6 expands the functionality of status signals with the introduction of the `StatusSignal` (Java, C++).

Note: For more information about status signals in Phoenix 6, see *Status Signals*.

5.2.1 Using Status Signals

Java

```java
// get latest TalonFX selected sensor position
// units are encoder ticks
int sensorPos = m_talonFX.getSelectedSensorPosition();

// latency is unknown
// cannot synchronously wait for new data
```

C++

```cpp
// get latest TalonFX selected sensor position
// units are encoder ticks
int sensorPos = m_talonFX.GetSelectedSensorPosition();

// latency is unknown
// cannot synchronously wait for new data
```

Java

```java
// acquire a refreshed TalonFX rotor position signal
var rotorPosSignal = m_talonFX.getRotorPosition();

// because we are calling getRotorPosition() every loop,
// we do not need to call refresh()
//rotorPosSignal.refresh();

// retrieve position value that we just refreshed
// units are rotations
var rotorPos = rotorPosSignal.getValue();

// get latency of the signal
var rotorPosLatency = rotorPosSignal.getTimestamp().getLatency();

// synchronously wait 20 ms for new data
rotorPosSignal.waitForUpdate(0.020);
```

C++

```cpp
// acquire a refreshed TalonFX rotor position signal
auto& rotorPosSignal = m_talonFX.GetRotorPosition();

// because we are calling GetRotorPosition() every loop,
// we do not need to call Refresh()
//rotorPosSignal.Refresh();

// retrieve position value that we just refreshed
// units are rotations, uses the units library
auto rotorPos = rotorPosSignal.GetValue();
```

```cpp
// get latency of the signal
auto rotorPosLatency = rotorPosSignal.GetTimestamp().GetLatency();

// synchronously wait 20 ms for new data
```

Changing Update Frequency (Status Frame Period)

v

Java

```java
// slow down the Status 2 frame (selected sensor data) to 5 Hz (200ms)
m_talonFX.setStatusFramePeriod(StatusFrameEnhanced.Status_2_Feedback0, 200);
```

C++

```cpp
// slow down the Status 2 frame (selected sensor data) to 5 Hz (200ms)
m_talonFX.SetStatusFramePeriod(StatusFrameEnhanced::Status_2_Feedback0, 200);
```

v

Java

```java
// slow down the position signal to 5 Hz
m_talonFX.getPosition().setUpdateFrequency(5);
```

C++

```cpp
// slow down the position signal to 5 Hz
m_talonFX.GetPosition().SetUpdateFrequency(5_Hz);
```

Important: Currently in Phoenix 6, when different status signal frequencies are specified for signals that share a status frame, the last specified frequency is applied to the status frame. As a result, users should apply the slowest status frame frequencies first and the fastest frequencies last.

5.2.2 Common Signals

Several status signals have changed name or form in Phoenix 6.

General Signals

Phoenix 5	Phoenix 6
BusVoltage	SupplyVoltage
Faults / StickyFaults (fills an object)	Fault_* / StickyFault_* (individual faults)
FirmwareVersion	Version

Talon FX Signals

Phoenix 5	Phoenix 6
MotorOutputPercent	DutyCycle
StatorCurrent	
	StatorCurrent (motoring +, braking -), TorqueCurrent (forward +, reverse -)
Inverted (true/false; matches set-Inverted)	AppliedRotorPolarity (CCW+/CW+; typically matches Inverted config, affected by follower features)
SelectedSensorPosition / SelectedSensorVelocity	Position / Velocity
IntegratedSensor* (in SensorCollection)	Rotor*
ActiveTrajectory* (only Motion Magic® and the Motion Profile Executor)	ClosedLoopReference* (all closed-loop control requests)
IsFwdLimitSwitchClosed / IsRevLimitSwitchClosed (true/false)	GetForwardLimit / GetReverseLimit (Open/Closed)

CANcoder Signals

Phoenix 5	Phoenix 6
MagnetFieldStrength	MagnetHealth

Pigeon 2 Signals

Note: Many Pigeon 2 signal getters in Phoenix 5 fill an array, such as YawPitchRoll. In Phoenix 6, these signals have been broken up into their individual components, such as Yaw, Pitch, and Roll.

Phoenix 5	Phoenix 6
RawGyro	AngularVelocity*
6dQuaternion	Quat*
BiasedAccelerometer	Acceleration*
BiasedMagnetometer	MagneticField*
RawMagnetometer	RawMagneticField*

5.3 Control Requests

Phoenix 6 provides an extensive list of flexible control modes through the use of strongly-typed control requests.

Note: For more information about control requests in Phoenix 6, see *Control Requests*.

5.3.1 Using Control Requests

Java

```java
// robot init, set voltage compensation to 12 V
m_motor.configVoltageComSaturation(12);
m_motor.enableVoltageCompensation(true);

// main robot code, command 12 V output
m_motor.set(ControlMode.PercentOutput, 1.0);
```

C++

```cpp
// robot init, set voltage compensation to 12 V
m_motor.ConfigVoltageComSaturation(12);
m_motor.EnableVoltageCompensation(true);

// main robot code, command 12 V output
m_motor.Set(ControlMode::PercentOutput, 1.0);
```

Java

```java
// class member variable
VoltageOut m_request = new VoltageOut(0);

// main robot code, command 12 V output
m_motor.setControl(m_request.withOutput(12.0));
```

C++

```cpp
// class member variable
controls::VoltageOut m_request{0_V};

// main robot code, command 12 V output
m_motor.SetControl(m_request.WithOutput(12_V));
```

Follower Motors

v

Java

```java
// robot init, set m_follower to follow m_leader
m_follower.follow(m_leader);
// m_follower should NOT oppose m_leader
m_follower.setInverted(TalonFXInvertType.FollowMaster);
// set m_strictFollower to follow m_leader
m_strictFollower.follow(m_leader);
// set m_strictFollower to ignore m_leader invert and use its own
m_strictFollower.setInverted(TalonFXInvertType.CounterClockwise);

// main robot code, command 100% output for m_leader
m_leader.set(ControlMode.PercentOutput, 1.0);
// - m_follower and m_strictFollower will also run at 100% output
// - m_follower will follow m_leader's invert, while m_strictFollower
//   ignores it and uses its own
// NOTE: if set(), neutralOutput(), or disable() is ever called on
//       the followers, they will stop following
```

C++

```cpp
// robot init, set m_follower to follow m_leader
m_follower.Follow(m_leader);
// m_follower should NOT oppose m_leader
m_follower.SetInverted(TalonFXInvertType::FollowMaster);
// set m_strictFollower to follow m_leader
m_strictFollower.Follow(m_leader);
// set m_strictFollower to ignore m_leader invert and use its own
m_strictFollower.SetInverted(TalonFXInvertType::CounterClockwise);

// main robot code, command 100% output for m_leader
m_leader.Set(ControlMode::PercentOutput, 1.0);
// - m_follower and m_strictFollower will also run at 100% output
// - m_follower will follow m_leader's invert, while m_strictFollower
//   ignores it and uses its own
// NOTE: if Set(), NeutralOutput(), or Disable() is ever called on
//       the followers, they will stop following
```

v

Java

```java
// class member variables
DutyCycle m_request = new DutyCycle(0);

// robot init, set m_follower to follow m_leader
// m_follower should NOT oppose leader
m_follower.setControl(new Follower(m_leader.getDeviceID(), false));
// set m_strictFollower to strict-follow m_leader
// strict followers ignore the leader's invert and use their own
m_strictFollower.setControl(new StrictFollower(m_leader.getDeviceID()));

// main robot code, command 100% output for m_leader
m_motor.setControl(m_request.withOutput(1.0));
// - m_follower and m_strictFollower will also run at 100% output
// - m_follower will follow m_leader's invert, while m_strictFollower
//   ignores it and uses its own
```

Chapter 5. API Migration

Changing Update Frequency (Control Frame Period)

v

Java

```java
// slow down the Control 3 frame (general control) to 50 Hz (20ms)
m_talonFX.setControlFramePeriod(ControlFrame.Control_3_General, 20);
```

C++

```cpp
// slow down the Control 3 frame (general control) to 50 Hz (20ms)
m_talonFX.SetControlFramePeriod(ControlFrame::Control_3_General, 20);
```

v

Java

```java
// class member variables
DutyCycle m_request = new DutyCycle(0);

// slow down the control request to 50 Hz
m_request.UpdateFreqHz = 50;
```

C++

```cpp
// class member variables
controls::DutyCycle m_request{0};

// slow down the control request to 50 Hz
m_request.UpdateFreqHz = 50_Hz;
```

Tip: UpdateFreqHz can be set to 0 Hz to synchronously one-shot the control request. In this case, users must ensure the control request is sent periodically in their robot code. Therefore, we recommend users call setControl no slower than 20 Hz (50 ms) when the control is one-shot.

5.3.2 Control Types

In Phoenix 6, voltage compensation has been replaced with the ability to directly specify the *control output type*.

All control output types are supported in open-loop and closed-loop control requests.

Table 1: Open-loop Control Requests

Phoenix 5	Phoenix 6
PercentOutput	DutyCycleOut
PercentOutput + Voltage Compensation	VoltageOut
Phoenix 5 does not support torque control	TorqueCurrentFOC (requires Pro)
Current closed-loop	This has been deprecated in Phoenix 6. • Users looking to control torque should use TorqueCurrentFOC (requires Pro) • Users looking to limit current should use supply and stator current limits

Table 2: Closed-loop Control Requests

Phoenix 5	Phoenix 6
Position	PositionDutyCycle
Velocity	VelocityDutyCycle
MotionMagic	MotionMagicDutyCycle
Closed-loop + Voltage Compensation	{ClosedLoop}Voltage
Closed-loop + Torque Control (not supported in Phoenix 5)	{ClosedLoop}TorqueCurrentFOC (requires Pro)

5.4 Closed-Loop Control

Phoenix 6 enhances the experience of using onboard closed-loop control through the use of standardized units and a variety of control output types.

Note: For more information about closed-loop control in Phoenix 6, see *Closed-Loop Control*.

5.4.1 Closed-Loop Setpoints

Phoenix 6 uses canonical units for closed-loop setpoints.

5.4.2 Closed-Loop Gains

Position without Voltage Comp

Phoenix 5 `ControlMode.Position` with voltage compensation **disabled** maps to the Phoenix 6 `PositionDutyCycle` control request.

Position with Voltage Comp

Phoenix 5 `ControlMode.Position` with voltage compensation **enabled** has been replaced with the Phoenix 6 `PositionVoltage` control request, which directly controls voltage.

Velocity without Voltage Comp

Phoenix 5 `ControlMode.Velocity` with voltage compensation **disabled** maps to the Phoenix 6 `VelocityDutyCycle` control request.

Additionally, kF from Phoenix 5 has been replaced with kV in Phoenix 6.

Velocity with Voltage Comp

Phoenix 5 `ControlMode.Velocity` with voltage compensation **enabled** has been replaced with the Phoenix 6 `VelocityVoltage` control request, which directly controls voltage.

Additionally, kF from Phoenix 5 has been replaced with kV in Phoenix 6.

5.4.3 Using Closed-Loop Control

Java

```java
// robot init, set slot 0 gains
m_motor.config_kF(0, 0.05, 50);
m_motor.config_kP(0, 0.046, 50);
m_motor.config_kI(0, 0.0002, 50);
m_motor.config_kD(0, 4.2, 50);

// enable voltage compensation
m_motor.configVoltageComSaturation(12);
m_motor.enableVoltageCompensation(true);

// periodic, run velocity control with slot 0 configs,
// target velocity of 50 rps (10240 ticks/100ms)
m_motor.selectProfileSlot(0, 0);
m_motor.set(ControlMode.Velocity, 10240);
```

C++

```cpp
// robot init, set slot 0 gains
m_motor.Config_kF(0, 0.05, 50);
m_motor.Config_kP(0, 0.046, 50);
m_motor.Config_kI(0, 0.0002, 50);
m_motor.Config_kD(0, 4.2, 50);

// enable voltage compensation
m_motor.ConfigVoltageComSaturation(12);
m_motor.EnableVoltageCompensation(true);

// periodic, run velocity control with slot 0 configs,
// target velocity of 50 rps (10240 ticks/100ms)
m_motor.SelectProfileSlot(0, 0);
m_motor.Set(ControlMode::Velocity, 10240);
```

Java

```java
// class member variable
VelocityVoltage m_velocity = new VelocityVoltage(0);

// robot init, set slot 0 gains
var slot0Configs = new Slot0Configs();
slot0Configs.kV = 0.12;
slot0Configs.kP = 0.11;
slot0Configs.kI = 0.48;
slot0Configs.kD = 0.01;
m_talonFX.getConfigurator().apply(slot0Configs, 0.050);

// periodic, run velocity control with slot 0 configs,
// target velocity of 50 rps
m_velocity.Slot = 0;
m_motor.setControl(m_velocity.withVelocity(50));
```

C++

Motion Magic®

Java

```java
// robot init, set slot 0 gains
m_motor.config_kF(0, 0.05, 50);
// PID runs on position
m_motor.config_kP(0, 0.2, 50);
m_motor.config_kI(0, 0, 50);
m_motor.config_kD(0, 4.2, 50);

// set Motion Magic settings
m_motor.configMotionCruiseVelocity(16384); // 80 rps = 16384 ticks/100ms cruise
→velocity
m_motor.configMotionAcceleration(32768); // 160 rps/s = 32768 ticks/100ms/s
→acceleration
m_motor.configMotionSCurveStrength(3); // s-curve smoothing strength of 3

// enable voltage compensation
m_motor.configVoltageComSaturation(12);
m_motor.enableVoltageCompensation(true);

// periodic, run Motion Magic with slot 0 configs
m_motor.selectProfileSlot(0, 0);
// target position of 200 rotations (409600 ticks)
// add 0.02 (2%) arbitrary feedforward to overcome friction
m_motor.set(ControlMode.MotionMagic, 409600, DemandType.ArbitraryFeedforward, 0.
→02);
```

C++

```cpp
// robot init, set slot 0 gains
m_motor.Config_kF(0, 0.05, 50);
// PID runs on position
m_motor.Config_kP(0, 0.2, 50);
m_motor.Config_kI(0, 0, 50);
m_motor.Config_kD(0, 4.2, 50);

// set Motion Magic settings
m_motor.ConfigMotionCruiseVelocity(16384); // 80 rps = 16384 ticks/100ms cruise
→velocity
m_motor.ConfigMotionAcceleration(32768); // 160 rps/s = 32768 ticks/100ms/s
→acceleration
m_motor.ConfigMotionSCurveStrength(3); // s-curve smoothing strength of 3

// enable voltage compensation
m_motor.ConfigVoltageComSaturation(12);
m_motor.EnableVoltageCompensation(true);

// periodic, run Motion Magic with slot 0 configs
m_motor.SelectProfileSlot(0, 0);
// target position of 200 rotations (409600 ticks)
// add 0.02 (2%) arbitrary feedforward to overcome friction
m_motor.Set(ControlMode::MotionMagic, 409600, DemandType::ArbitraryFeedforward, 0.
→02);
```

v **Note:** The Motion Magic® S-Curve Strength has been replaced with jerk control in Phoenix 6.

Motion Profiling

The Motion Profile Executor is not supported in the current release of Phoenix 6. Users can use *Motion Magic®* or run a motion profile on the robot controller.

5.5 Feature Replacements

In addition to the changes shown in the other sections, several other Phoenix 5 features have been replaced or improved upon in Phoenix 6.

5.5.1 Motor Invert

In Phoenix 6, motor invert is now a persistent config (Java, C++) instead of a control signal.

> **Warning:** Since invert is a persistent config, getting and setting motor inverts are now blocking API calls. We recommend that users only set the invert once at program startup.

5.5.2 Neutral Mode

In Phoenix 6, Neutral mode is now available in API as a config (Java, C++). Many control requests also have the ability to override the neutral mode to either force braking (Java, C++) or force coasting (Java, C++).

5.5.3 Nominal Output

The Talon FX forward and reverse Nominal Output configs have been removed in Phoenix 6.

The typical use case of the nominal output configs is to overcome friction in closed-loop control modes, which can now be achieved using the kS feedforward parameter (Java, C++).

5.5.4 Sensor Phase

The Talon FX `setSensorPhase()` method has been removed in Phoenix 6.

- The Talon FX integrated sensor is always in phase, so the method does nothing in Phoenix 5.
- When using a remote sensor, you can invert the remote sensor to bring it in phase with the Talon FX.

5.5.5 Sensor Initialization Strategy

The Talon FX and CANcoder sensors are always initialized to their absolute position in Phoenix 6.

5.5.6 Clear Position on Limit

In Phoenix 5, users could configure the TalonFX to clear its sensor position (i.e. set to 0) when a limit switch is triggered. In Phoenix 6, this feature has been improved to allow users to specify the applied sensor position when a limit switch is triggered. This can be configured using the *LimitAutosetPositionValue configs (Java, C++).

5.5.7 Velocity Measurement Period/Window

In Phoenix 6, the velocity rolling average window in Talon FX and CANcoder has been replaced with a Kalman filter, resulting in a less noisy velocity signal with a minimal impact on latency. As a result, the velocity measurement period/window configs are no longer necessary in Phoenix 6 and have been removed.

5.5.8 Integral Zone and Max Integral Accumulator

Phoenix 6 automatically prevents integral windup in closed-loop controls. As a result, the Integral Zone and Max Integral Accumulator configs are no longer necessary and have been removed.

5.5.9 Features to Be Implemented

The following Phoenix 5 features are not implemented in the current release of Phoenix 6 but are planned to be implemented in the future.

Feature	Status
Remote sensors and limits	High priority
Auxiliary PID	High priority
Orchestra	Normal priority
Talon FX PWM Input	Normal priority
CANdle Support	Normal priority

5.5.10 Features Omitted

The following Phoenix 5 features have been omitted from Phoenix 6. While there are no plans for these features to be added, if there is customer demand for these features, they may be considered for addition in the future.

Feedback is welcome at feedback@ctr-electronics.com.

- Motion Profile Executor
  - Control requests are planned to be improved to cover many of the use cases of the Motion Profile Executor.
- Allowable Closed-Loop Error

6

CAN Bus Utilization

CTR Electronics goes through great efforts to make our products efficient on CAN Bus bandwidth. This article highlights the average default bandwidth utilization of supported Phoenix 6 devices. Users should keep total CAN Bus utilization below 90% to prevent any unexpected behavior. Information on changing the default CAN update frequency is available in the *API documentation*.

Note: Using Phoenix API will automatically start up a diagnostics server which adds a constant 0-5% total CAN Bus utilization.

Device	Phoenix 5 (CAN 2.0)	Phoenix 6 (CAN 2.0)	Phoenix 5 (CAN FD)	Phoenix 6 (CAN FD)
Talon FX	4.7%	4.1%	2.0%	1.8%
CANcoder	1.8%	1.7%	0.9%	0.9%
Pigeon 2	5.5%	3.1%	2.5%	1.3%

Device Licensing

The following devices are eligible for single-device licensing:

- TalonFX (Falcon 500)
- Pigeon 2
- CANcoder

Additionally, CANivore is supported for licensing. When a CANivore is licensed, all devices on that bus are **Pro** enabled without additional activation.

Important: All license activation and verification features are only available in **Phoenix Tuner X**. Phoenix Tuner v1 does not support licensing actions.

7.1 Purchasing a License

Licenses can be purchased in the licensing section on the CTR Electronics store. Click here to purchase a license.

Once a license has been purchased, you will receive an email confirmation confirming your purchase. Once this email is received, the license should be visible in the list of licenses in Tuner X.

7.2 Activating a License

Licenses are activated by first clicking on the **LIC** icon in the bottom right corner of the device card.

This will open up a screen which displays a list of currently attached licenses for that device. Click on the *Activate a new license* button on the bottom of the popup.

A list of purchased (but unattached) license seats are shown here. Click on the license you would like to redeem and press the *Activate Selected License* button to confirm redemption of that seat.

Warning: Users should be aware that license activation is permanent and irreversible

Once the activation is complete, the license will be downloaded to the device. In the event that Tuner X disconnects from the internet or from the robot before this completes, the license is still activated and available for download the next time Tuner X is connected to the internet/robot.

7.2.1 Activating a License without a Robot

Devices that have been seen by Tuner X at least once will be available in *Device History*. This can be useful for licensing a device when disconnected from the robot.

7.3 Verifying Activation State

An icon displaying the license state of your device is located in the bottom right of the device card.
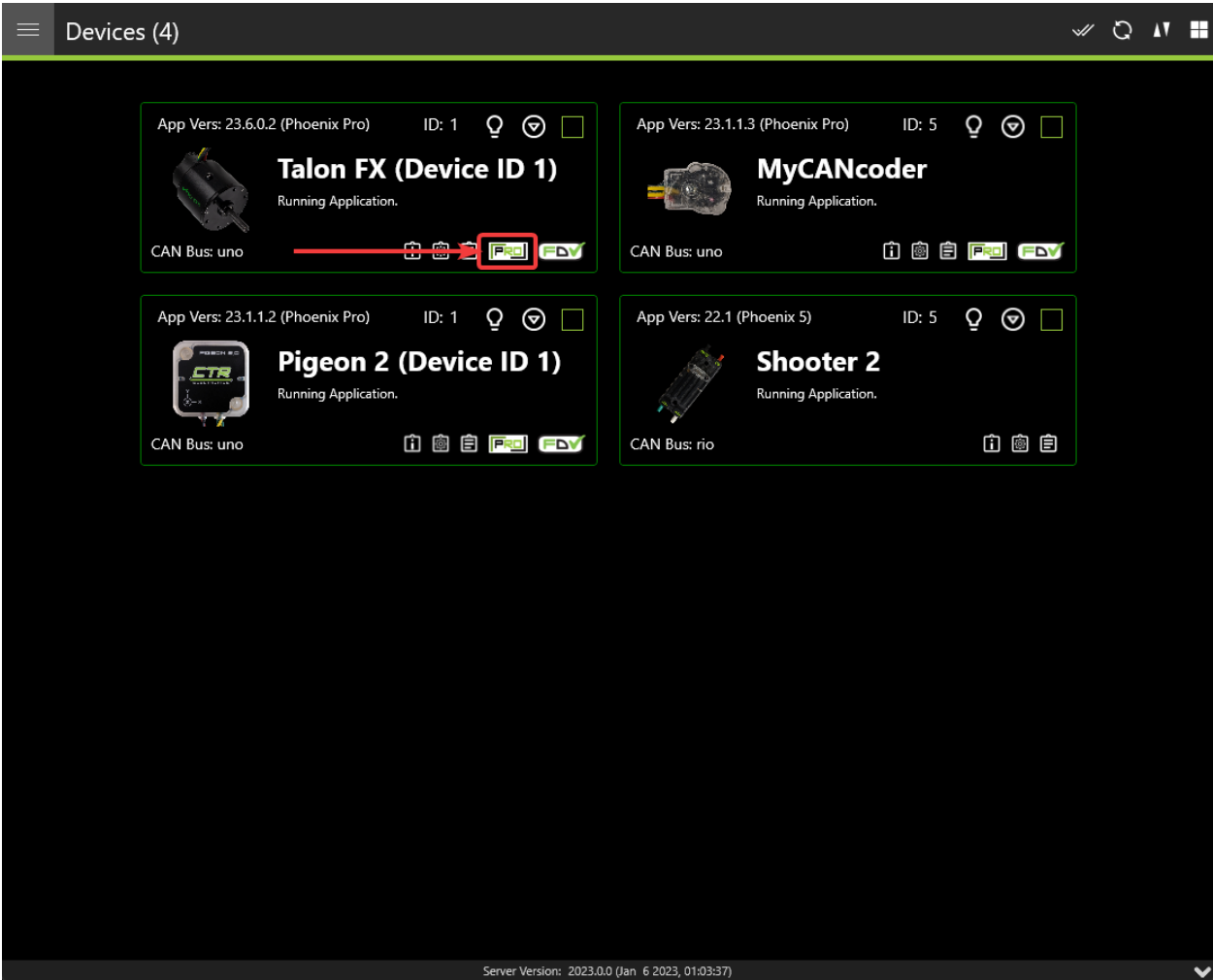
The below table can be used to determine your device license state for troubleshooting.

State	Image	Description
Licensed		Device is licensed for the current version of Phoenix Pro API.
CANivore contains Licenses		CANivore contains at least one bus license, which it will use to remote-license all compliant CAN devices.
Pro Licensing Error		Device is licensed and there was an error communicating license state.
Licensing Error		Device is not licensed and there was an error communicating license state.
Not Licensed		Device is not licensed for this version of Phoenix Pro API.
Licensing Not Supported	Icon not present	Device does not support licensing or is using an incompatible firmware for device licensing.

Additionally, users can perform a *Self Test* to verify that the device has a valid license.

7.4 Troubleshooting

- Did you activate a license for this device?
    - Clicking on the icon will show licenses that are attached to this device
- Is the latest diagnostic server running?
    - Check the version at the bottom of Tuner X's devices page.
        * Latest version details can be found in the changelog under the latest Phoenix-Pro version.
    - Confirm the **vendordep** in your robot project is the latest version.
    - Alternatively, you can deploy the **temporary diagnostic server**.
- Is the latest **Pro** firmware flashed onto the device?

Phoenix Tuner X

8.1 What is Phoenix Tuner X?

Note: The legacy Phoenix Tuner v1 is still available for use and is installed as part of the Phoenix installer.

Phoenix Tuner is the companion application allowing you to configure, analyze, update and control your device. Users may choose to use either Phoenix Tuner v1 (included as part of the Phoenix Installer) or Phoenix Tuner X (Android, Windows).

Phoenix Tuner X supports Android 8.0+ and Windows 10 (1903+) and Windows 11.

Important: While CTR Electronics supports both Phoenix Tuner v1 and Phoenix Tuner X, certain features such as device licensing and improved batch upgrading are only available in Phoenix Tuner X.

Tip: Many UI elements contain hover tooltips. That means the user can hover over them with their mouse for a text explanation of what they do.

8.1.1 Connecting Tuner

Installed onto the robot controller (either manually or via a robot program) is the Phoenix Diagnostics Server. This program enables communication between Tuner X and the robot controller for managing and setting up devices.

Connecting to the Server

A dropdown/textbox is available in the upper-left flyout menu.
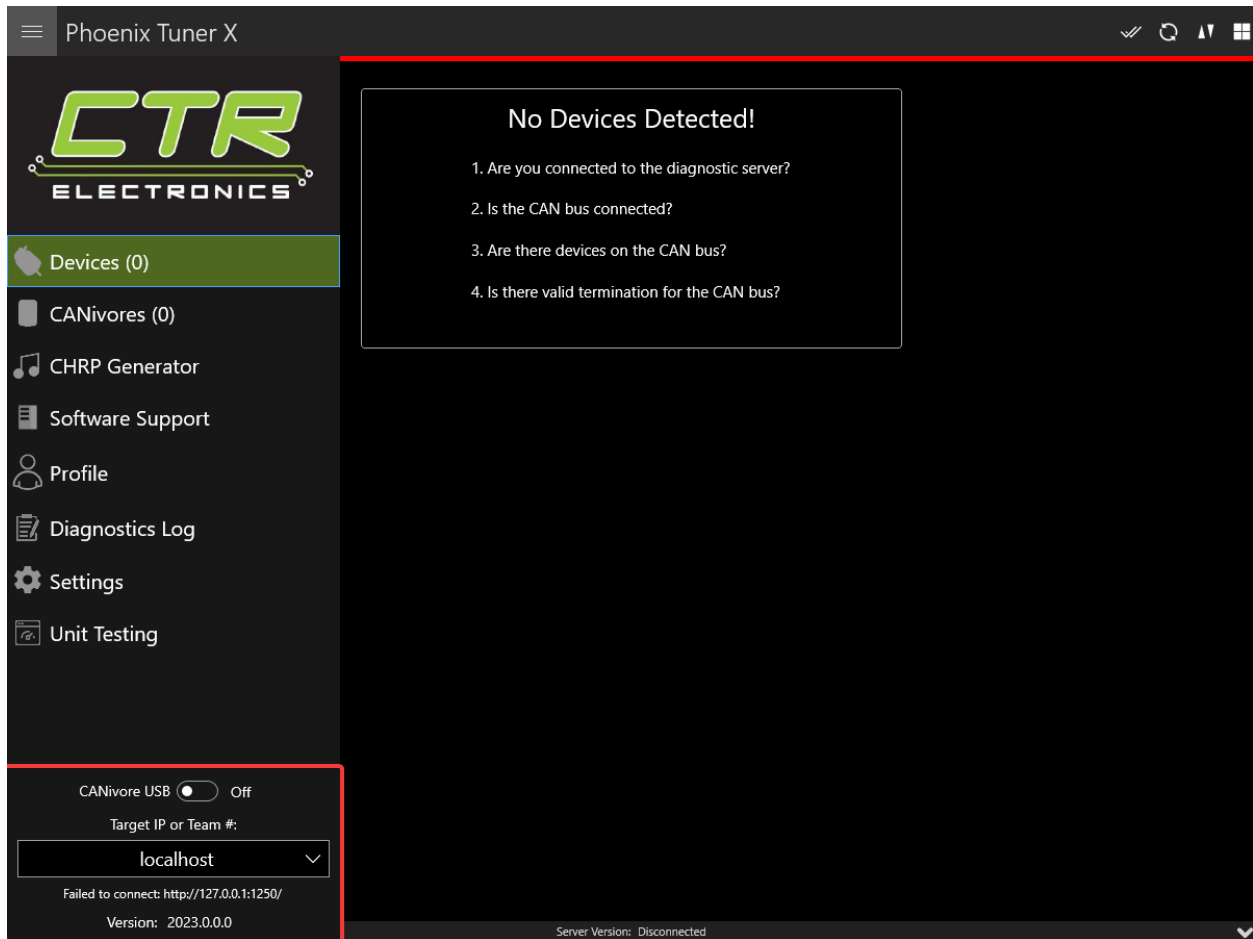
By clicking the arrow, you can change between presets such as:

- *Driver Station* – Retrieves the robot IP from the FRC Driver Station if launched
- *roboRIO USB* – Defaults to `172.22.11.2` which is the roboRIO IP when connected via USB
- *localhost* – Use for simulation or hardware-attached CANivore.

Alternatively, the user can manually enter the robot IP into the textbox.

Configuring SSH Credentials (non-FRC)

When using a non-FRC robot controller (non-roboRIO), users must have their SSH credentials configured in *Settings* for general use.

Temporary Diagnostics (FRC)

Devices can be configured without a diagnostic server present. This can be useful if the roboRIO has been freshly imaged. Ensure that you are pointed at the roboRIO IP address (usually `10.TE.AM.2` where `TE.AM` is the team number) and then click the *Run Temporary Diagnostic Server* in *Settings*.

Changing Diagnostics Server Port (non-FRC)

The target server port can be changed in the Tuner X *Settings* page, which is accessed from the flyout menu.

Important: The default port for diagnostic server is 1250. FRC users must not change this under any circumstance.

Localhost Troubleshooting

When Tuner X is first started after installation, it may request admin privileges to access the localhost network. If the user disallows admin access, diagnostic servers hosted on the local machine (simulation and hardware-attached CANivore) may not be visible in Tuner X. Users can manually grant this permission afterward by clicking the *Grant Localhost Permissions* in *Settings*.

8.1.2 Device List

Card Layout

Grid Layout

Name	Model	ID	CAN Bus	Firmware Version	Software Status	Manufacture Date	Canivore Dev Name	Hardware Version	Serial No	Vendor	Update?
Talon FX (Device ID 1)	Talon FX	1	2B189E6333533 8532020203438 3803FF	22.99.22.10 (Phoenix Pro)	Running Application.	Oct 1, 2019	uno	2.2	000E07B60975E 00000120000D8 0000F7	Cross The Road Electronics	☐
CANCoder (Device ID 5)	CANCoder	5	2B189E6333533 8532020203438 3803FF	22.99.22.4 (Phoenix Pro)	Running Application.	Nov 1, 2019	uno	1.0	000E07B60975E 00000020000F7 000110	Cross The Road Electronics	☐
Pigeon 2 (Device ID 0)	Pigeon 2	0	2B189E6333533 8532020203438 3803FF	22.99.22.2 (Phoenix Pro)	Running Application.	Feb 4, 2022	uno	2.1	000E066E0182B 8000005000114 000121	Cross The Road Electronics	☐
Shooter 2	Talon SRX	5	rio	20.1 (Phoenix 5)	Running Application.	Aug 14, 2015		1.7	0000000000000 0000000000000 000000	Cross The Road Electronics	☐

The **Devices** page is the first page that is shown to the user upon launching the application. The **Devices** page by default shows a grid of cards, but can be changed to a flat grid view (similar to Phoenix Tuner v1) by clicking on the 4 grid square icon located in the top right corner (not available in Android Tuner X).

Card Colors

The color of the device cards is helpful as a visual indicator of device state. The meaning of the card color is also shown as text underneath the device title.

Color	Description
Green	Device has latest firmware.
Purple	Device has unexpected firmware version.
Yellow	A new firmware version is available. Check the changelog to determine if the new version matters to your application
Red	Device has a duplicate ID.
Blue	Failed to retrieve list of available firmware.

Clipboard Options & Licensing

Phoenix Tuner X provides icons at the bottom right of each card that will allow the user to copy to the clipboard the device details, configs and Self Test. This can be useful for support requests and additional debugging.
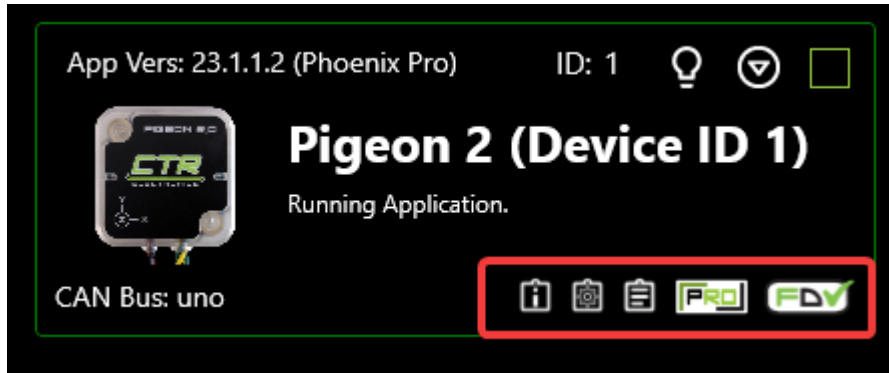
Devices that support CAN FD are shown via a *CAN FD* icon in the bottom right of the card.

Note: The *CAN FD* icon does **not** indicate that the device is currently on a CAN FD bus, merely that it supports CAN FD.

The other major icon in the bottom right of the device card is the licensing indicator. This showcases the *licensing states* and when clicked, will open the licensing dialog.

Batch Field Upgrade

Phoenix Tuner X allows the user to batch field upgrade from the **Devices** page. The user can either select devices by their checkbox (in the top right corner of their respective card) or by selecting the checkmark icon in the top right.
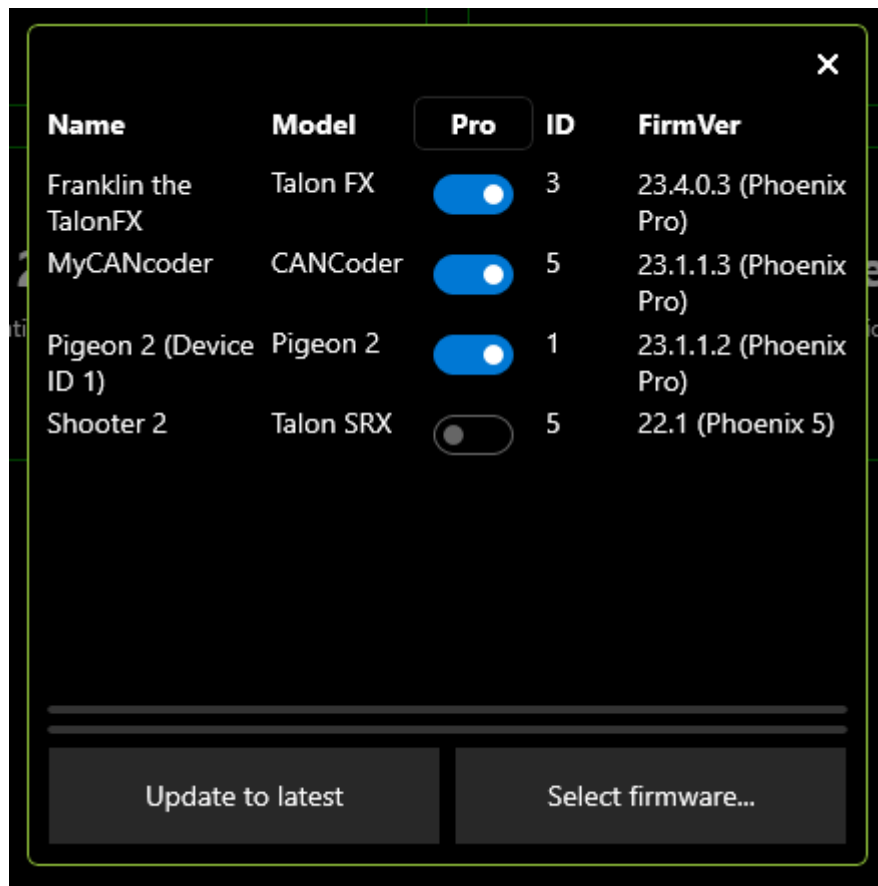

Tip: Selecting a device using their checkbox and clicking the checkmark in the top right will select all devices of the same models
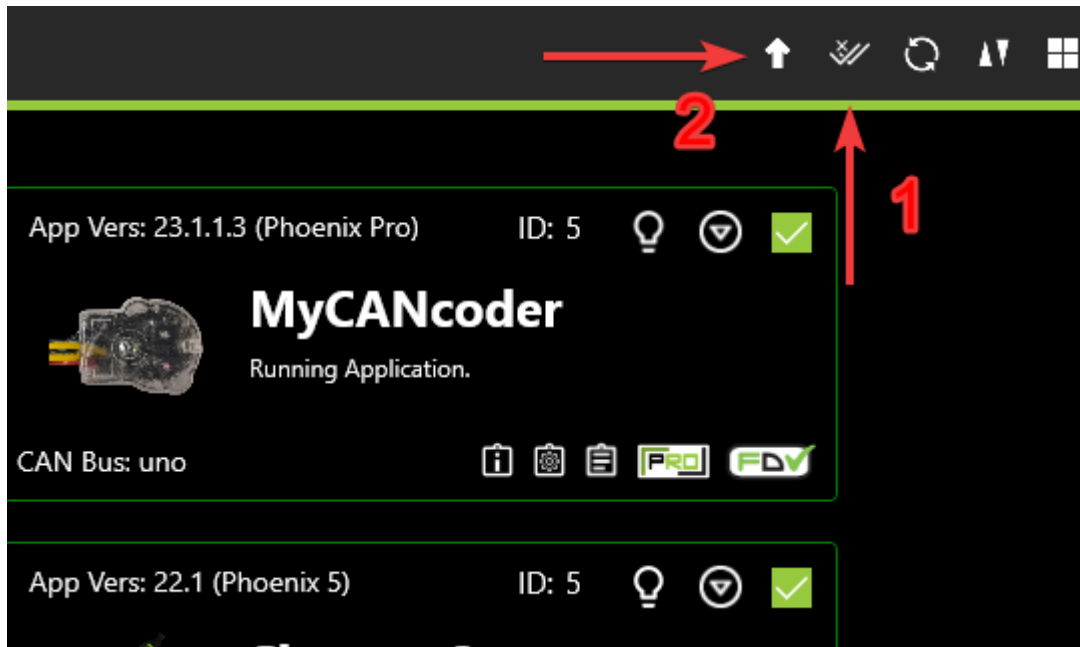

Step 1 in the above image selects all devices of the same model (or all devices if no device is currently check-boxed).

Step 2 in the above image opens the field-upgrade dialog.

Once the upgrade dialog is opened, information detailing the device name, model, ID, and firmware version is presented. There is a *Pro* column that has a toggle. This toggle represents whether to upgrade to **Pro** or **v5** firmware. If this toggle is disabled (as evident from being greyed out), then there is no available **Pro** firmware for that device (TalonSRX, legacy devices).

Name	Model	Pro	ID	FirmVer
Franklin the TalonFX	Talon FX	(on)	3	23.4.0.3 (Phoenix Pro)
MyCANcoder	CANCoder	(on)	5	23.1.1.3 (Phoenix Pro)
Pigeon 2 (Device ID 1)	Pigeon 2	(on)	1	23.1.1.2 (Phoenix Pro)
Shooter 2	Talon SRX	(off)	5	22.1 (Phoenix 5)

Update to latest   Select firmware...

The user can begin the upgrade progress by selecting *Update to latest* or *Select firmware...*. The first option will upgrade all listed devices to their latest available firmware (v6 or v5 depending on the toggle state). The second option will open a popup allowing you to select a specific version or firmware file per model.
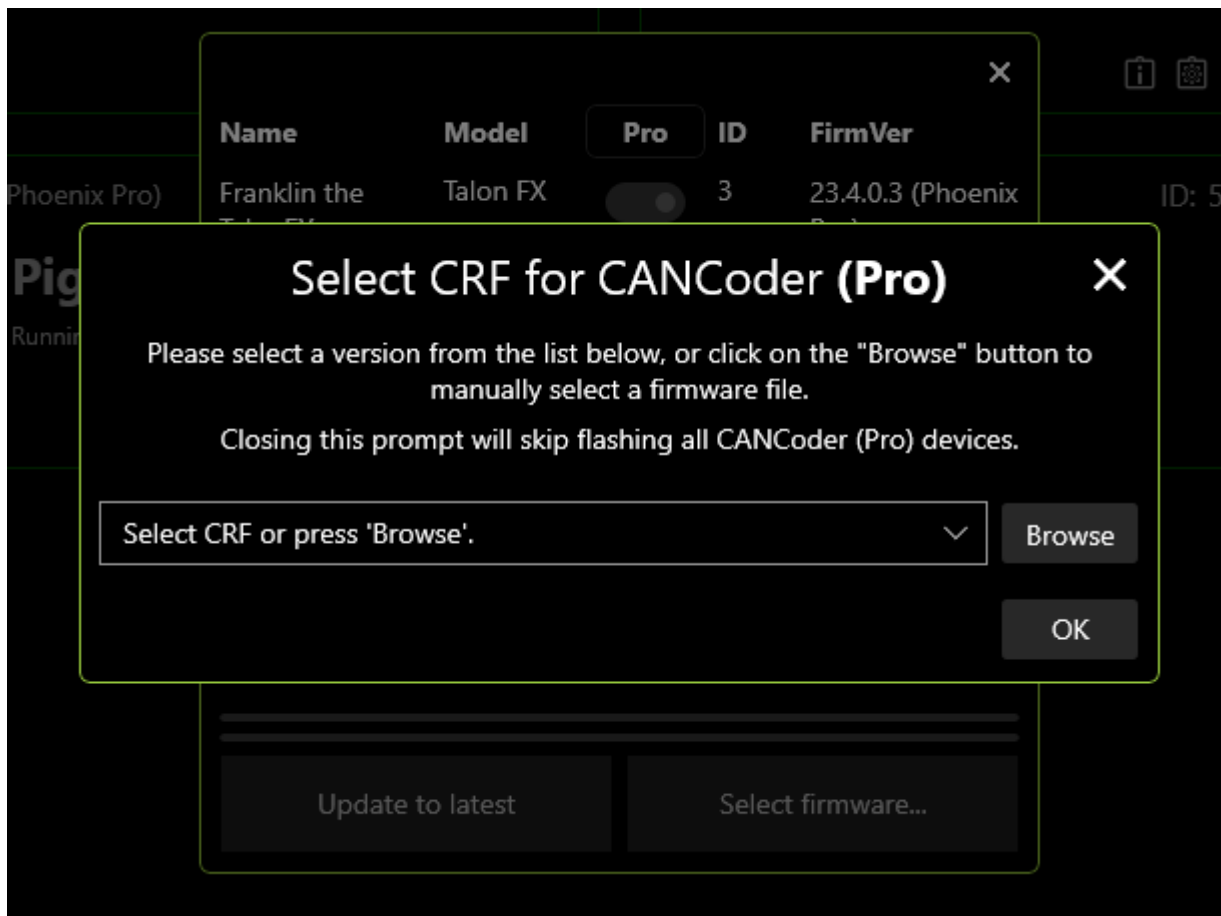
Tip: Generally, users should update their devices to the latest available firmware version. If manually selecting a CRF is important, the firmware files are available for download on our GitHub Repo.

Important: While the user can cancel firmware upgrading using the "X" button in the top-right, this will **not** cancel the current device in progress. It will finish upgrading the current device and will not upgrade subsequent devices. Typical Tuner X behavior will resume once the current device finishes flashing.

8.1.3 Device History

Users can access a list of past devices connected to Tuner X and license them via the *Device History* page. This is accessible from the left-hand sidebar. This list is not automatically refreshed, but users can refresh it by pressing the refresh icon in the top-right of the page.

Name	Model	ID	CAN Bus	Firmware Version	Manufacture Date	Hardware Version	Serial No	Last Seen At
Shooter 2	Talon SRX	5	rio	22.1 (Phoenix 5)	Aug 14, 2015	1.7	000000000000000000 0000000000000000	1/19/2023 7:33:30 PM
Pigeon 2 (Device ID 1)	Pigeon 2	1	uno	23.1.1.2 (Phoenix Pro)	Feb 4, 2022	2.1	000E066E0182B80000 05000114000121	1/19/2023 7:33:30 PM
MyCANcoder	CANCoder	5	uno	23.1.1.3 (Phoenix Pro)	Nov 1, 2019	1.0	000E07B60975E00000 020000F7000110	1/19/2023 7:33:30 PM
Talon FX (Device ID 1)	Talon FX	1	uno	23.5.0.0 (Phoenix Pro)	Oct 1, 2019	2.2	000E07B60975E00000 120000D80000F7	1/19/2023 7:33:30 PM
uno	CANivore			23.2.0.0	July 1, 2022	1.2	2B189E633353385320 202034383803FF	1/19/2023 7:33:30 PM

Server Version: Disconnected

Licensing from Device History

Users can activate a license for a disconnected device by clicking on the device in the Grid. Then, select the "PRO" icon at the bottom right of the device card.

From there, the user can activate a license for the device like normal. Once the device license has been activated, the user still needs to connect Tuner X to the robot to transfer the activated license to the device.

The "PRO" icon may be replaced with a greyed "LIC" icon in the following situations:

- The device is on Phoenix 5 firmware and actively connected to Tuner X
- The device is not a Phoenix 6 compatible device

Users who license an eligible Phoenix 6 device running Phoenix 5 firmware must *update the device firmware* to v6 compatible firmware to utilize licensed features.

8.1.4 Device Details

The **Device Details** page can be accessed by clicking on the device card (or double clicking on the device row when in grid view). This view allows you to access detailed device actions such as:
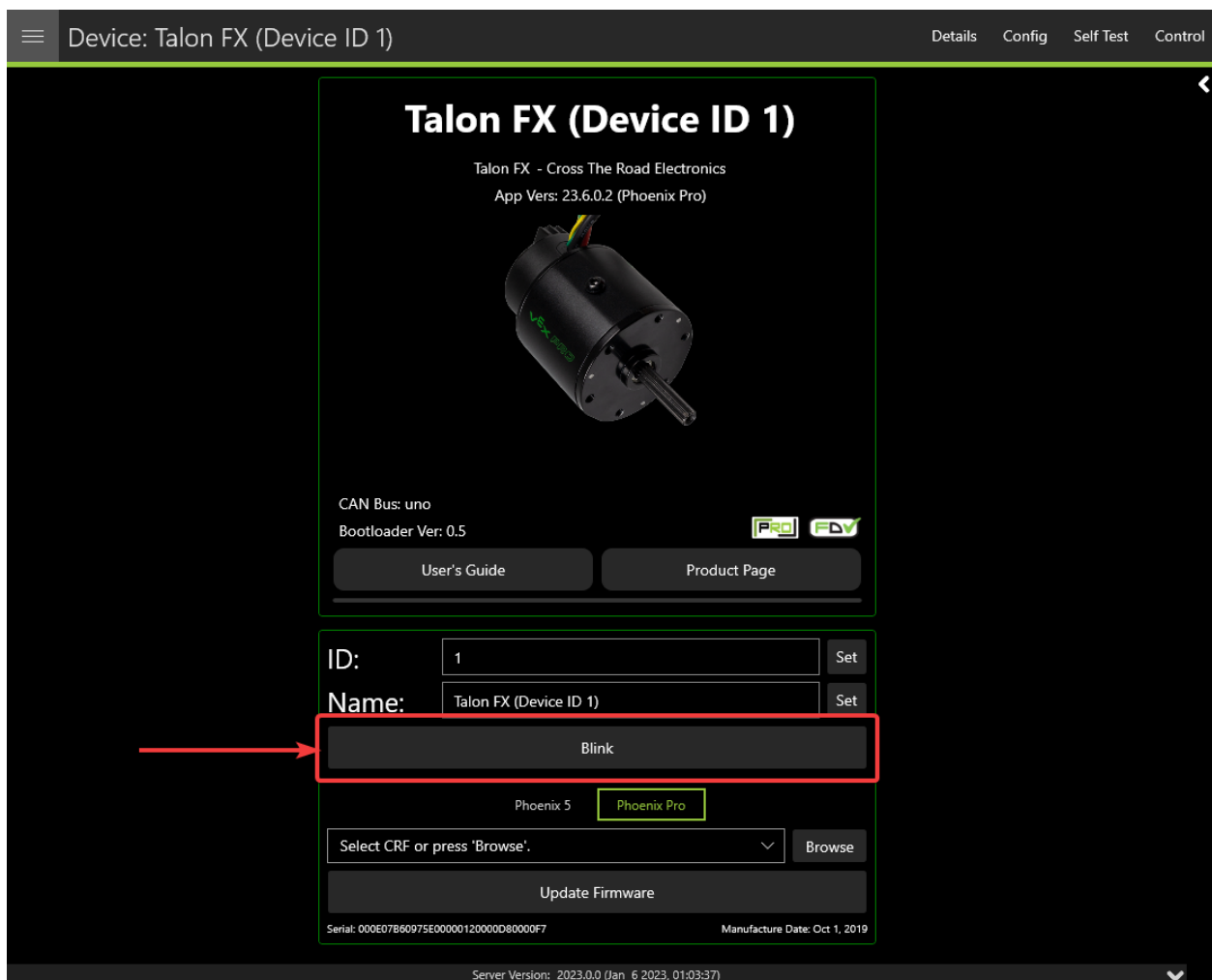
- Device Details (Name, ID, Firmware Version, Model, Serial No, etc.)
- Blinking LEDs
- Field Upgrading
- Licensing Details (by clicking on the LIC/PRO icon)
- Configs

- Control
- Self Tests
- Plotting
- Pigeon 2 Mount Calibration

Note: Plotting currently only works with Phoenix 5 devices.


Blinking

All CTR Electronics devices can be blinked (rapidly flash the LEDs). This can be useful for handling whenever you have duplicate devices using the same ID on the CAN bus.

Verifying Device Details

This screen highlights information such as (1) Device Name, (2) Device Model, (3) Firmware Version.

Tip: Clicking in the blank space outside the detail frames will bring the user back to the devices page.
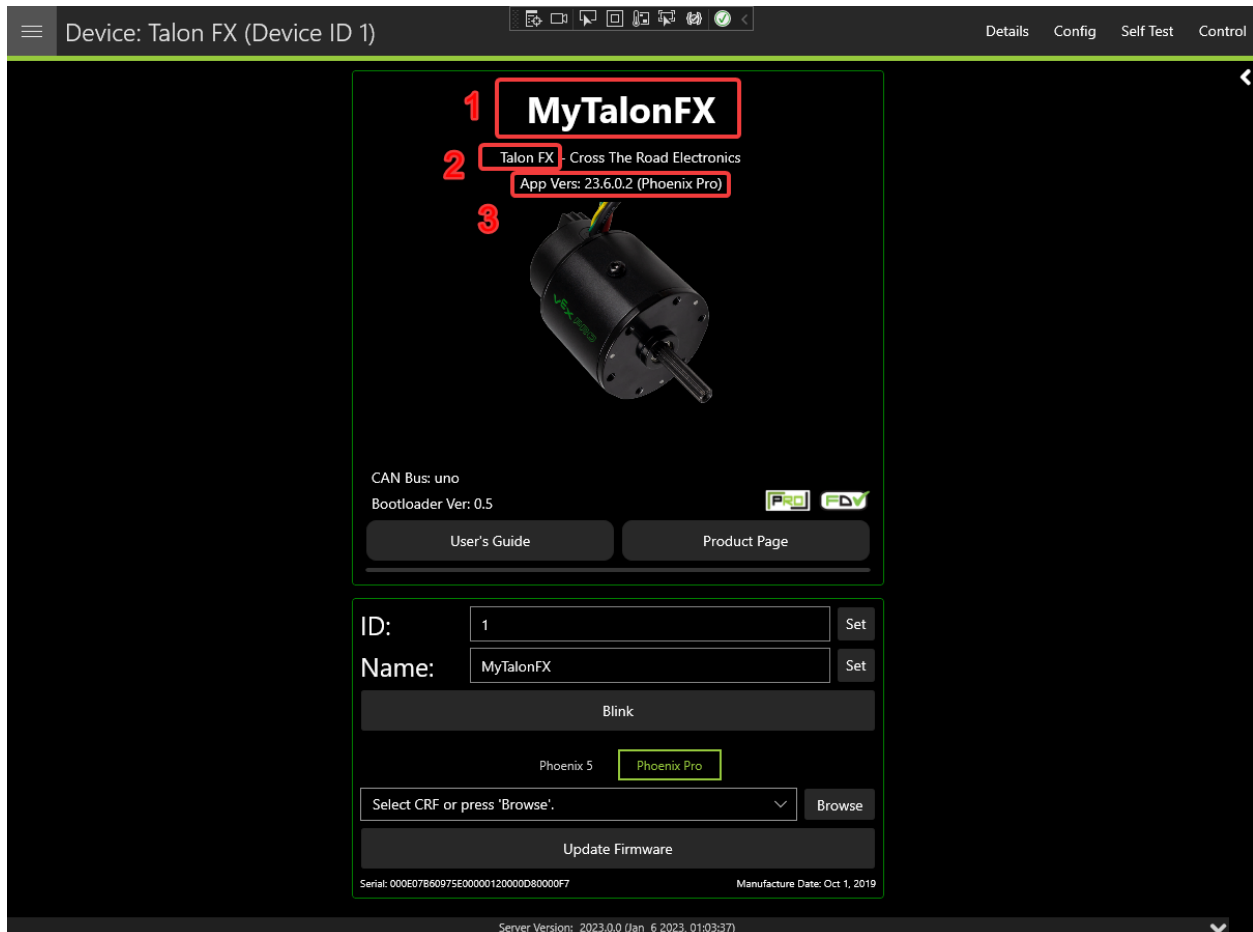
Configuring Name & IDs

All devices can have their Name (1) and ID (2) configured via their respective textbox. IDs are limited to the range of 0 to 62 (inclusive). After inputting the ID or name, press the *Set* button to save the changes to the device.
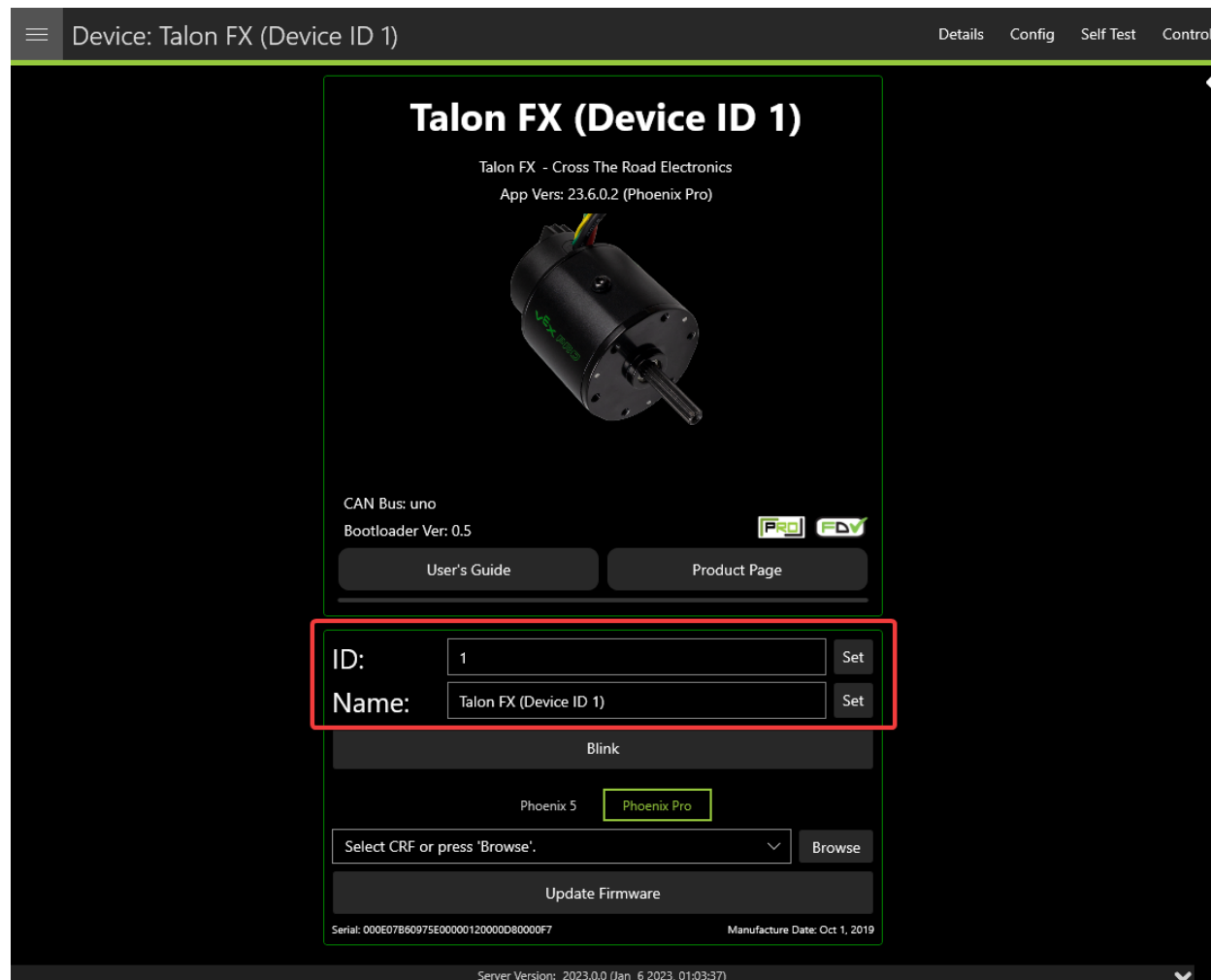
## Field-Upgrade Firmware Version

Tuner X has improved firmware upgrading functionality by **automatically downloading and caching** firmware. Upon initial Tuner X launch, the latest firmware for all devices will automatically be downloaded in the background (takes <10s on most internet connections). The individual device page allows you to select specific firmware versions for your device via the firmware dropdown. Batch firmware can also be completed via the *batch field upgrade pop-up*.

Important: Users should ensure they select Phoenix 6 firmware when using Phoenix 6 API, and Phoenix 5 firmware when using Phoenix 5 API. A single robot project may use both APIs simultaneously.
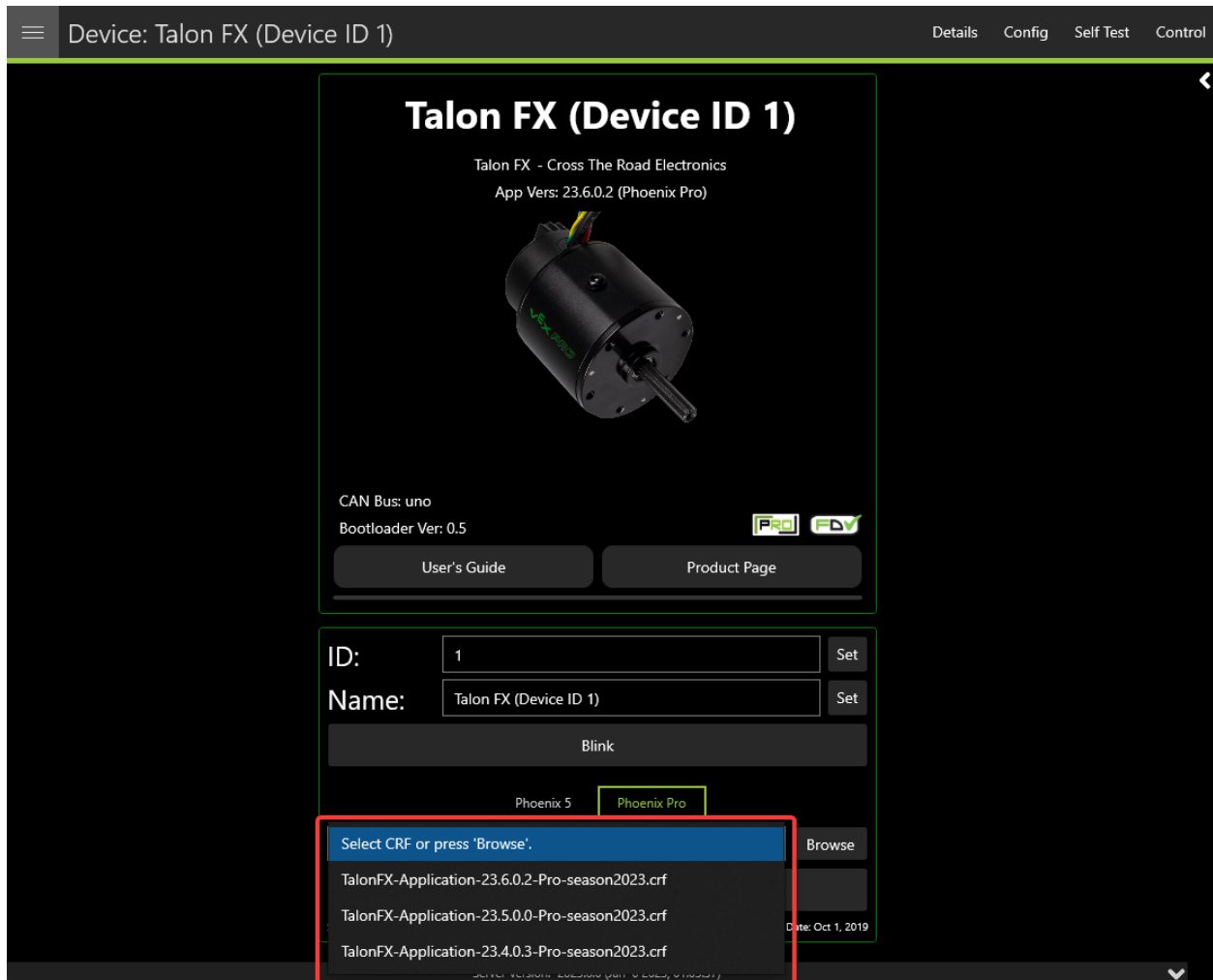
Users can switch between "Phoenix 6" and "Phoenix 5" by clicking on the toggle above the firmware dropdown.

Note: The toggle between Phoenix 6 and Phoenix 5 firmware only affects online field-upgrades.
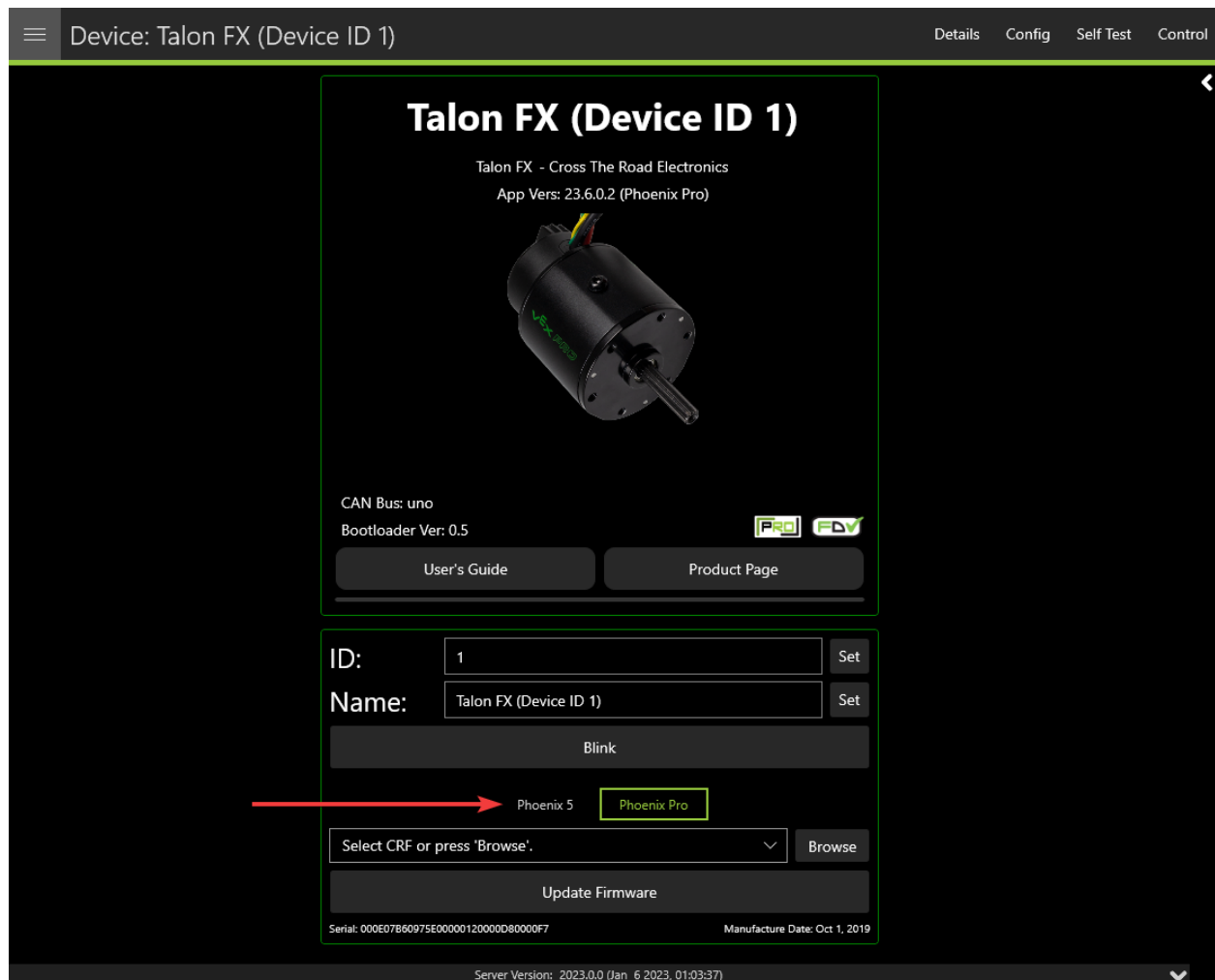
8.1.5 Tuner Configs

Tip: Devices can also be *configured in code*.

Configs can be viewed, modified, backed-up, restored, and factory-reset via the *Configs* tab in Phoenix Tuner X.

To apply a modified config, press the *Apply Configs* button on the bottom button bar.

8.1.6 Self Test Snapshot

Self Test Snapshot is a diagnostic feature of all supported devices that will show the immediate state of the device. This is extremely useful for troubleshooting and ensuring the device is working properly. Phoenix 6 with Phoenix Tuner X improves upon Self Test by showing the information in clean tables, animations and detailed units.

Self Test also includes 3 buttons: *Refresh*, *Blink/Clear Faults* and *Share to Support*. *Refresh* will refresh the Self Test information, *Blink/Clear faults`* will blink the device and clear any faults on the device. *Share to Support* will open the default email client with an email to CTR Electronics support.

Viewing Status LEDs

Phoenix 6 devices report status LEDs as an animated GIF in Phoenix Tuner X. This can be useful for diagnosing a device when it's buried in a robot.

8.1.7 Plotting

Supported devices can have certain signals/sensor data plotted in real-time without any additional configuration. To get started, click on the *Plot* button in the top right navigation bar.

Tip: Plotting is supported in both Phoenix 5 and Phoenix 6.
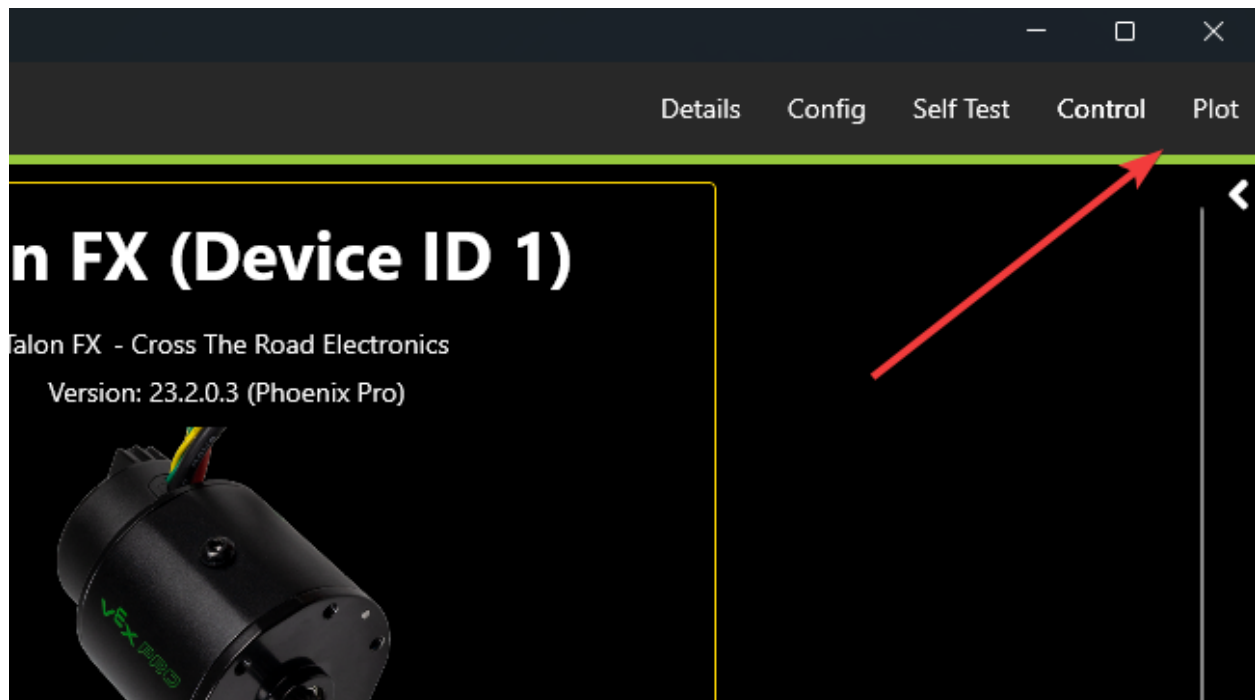
At the top of this page is a list of supported values that can then be plotted. Click on the signal that you wish to plot. Then click *Enable Plot* on the left.

Adjusting Plotting Time Period

Plotting time period (the time frame that points are recorded) can be adjusted using the *Time Period* textbox.

Exporting Data

Plots can be exported into *csv* format for viewing in an external analysis tool. Click on the *Export as CSV* button.

Plot Appearance & Behavior

Important: Scatter points may dramatically affect Tuner X performance.

Plotting supports zoom and panning via the mouse and scroll wheel (or via gestures on Android). The point appearance can also be adjusted between "Spline" and "Scatter".
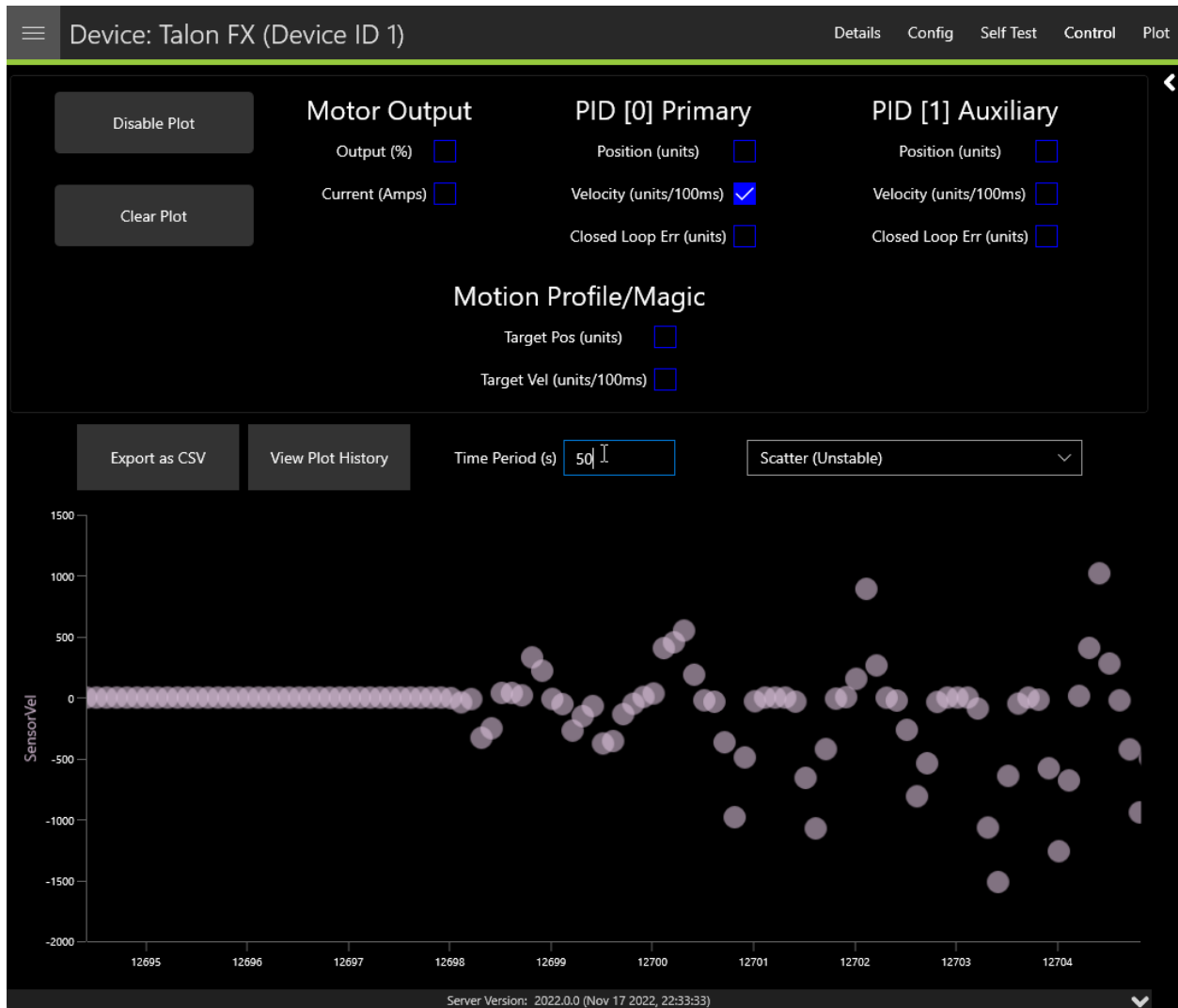
8.1.8 Pigeon 2.0 Calibration

It is recommended that calibration is performed once the Pigeon 2.0 has been mounted to the robot. Calibration will calculate the optimal offsets to apply to ensure that Pose, Pitch and Yaw is 0 when the robot is considered "flat". Users can access the calibration menu by clicking on the Pigeon 2.0 in *Devices* and clicking *Calibration* in the top right.

Read through the on-screen instructions and click *Begin Mount Calibration*.

TalonFX

The Falcon 500 powered by Talon FX is a brushless motor with an integrated motor controller and high-resolution encoder, custom designed specifically for the FIRST Robotics Competition, through a collaboration between Cross the Road Electronics and VEX Robotics.

Store Page   CAD, Firmware and purchase instructions.

Hardware User Manual   Wiring and mount instructions in PDF format.

9.1 Actuator Limits

CTR Electronics actuators, such as the TalonFX, support various kinds of hardware and software limits.

Documentation on retrieving and configuring limits can be found *here*.

9.1.1 Limit Switches

CTR Electronics supported actuators have limit features that will automatically neutral the actuator output (set voltage to 0) if a limit switch is activated. By default, limits are set to "normally open". This means that the switch needs to be explicitly closed (or grounded) for the actuator output to be set to neutral.

When the limit switch is closed (connected to ground), the actuator will disable and the pattern will move toward the forward/reverse limit pin (red blink pattern will move toward the forward limit pin when the forward limit is closed, and vice-versa).

Tip: For more information on limit switch wiring, consult the TalonFX User's Guide.

9.2 Status Light Reference
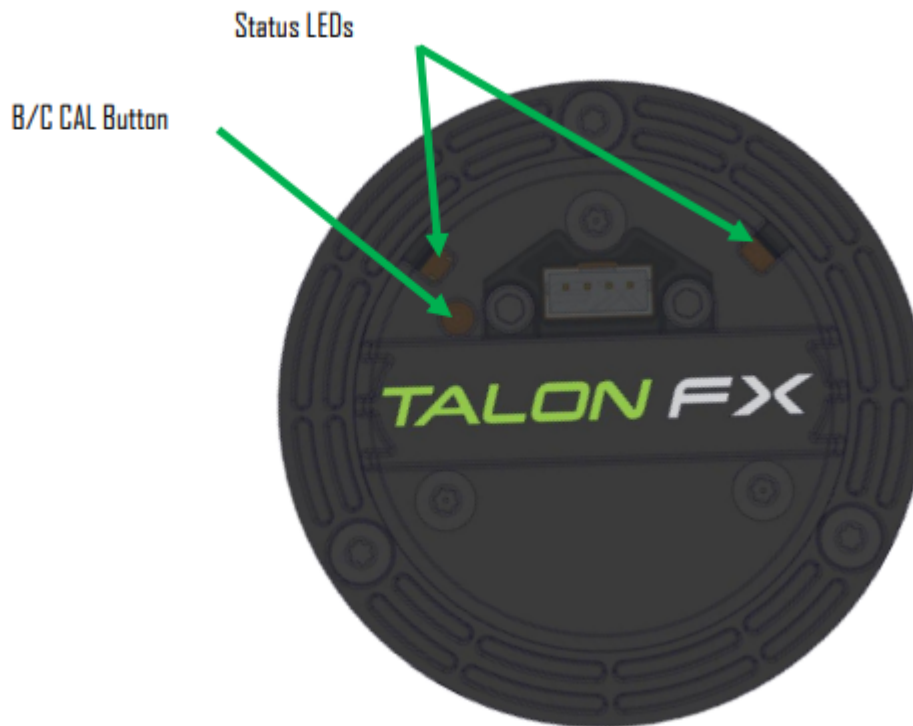
LED State	Description
Alternating Off/Orange	Talon FX is disabled. Robot controller is missing on the bus or the diagnostic server is not installed.
Simultaneous Off/Orange	Talon FX is disabled. Phoenix is running in Robot Controller.
Alternating Red/Green	Talon FX is not licensed. Please license device in Phoenix Tuner.
Off/Slow Red	CAN/PWM is not detected.
Red/Orange	Damaged Hardware
Off/Red	Limit Switch or Soft Limit triggered.
Green/Orange	Device is in bootloader.

Pigeon 2.0

Pigeon 2.0 is the next evolution in the family of Pigeon IMUs.

With no on-boot calibration or temperature calibration required and dramatic improvement to drift, the Pigeon is the easiest IMU to use yet.

10.1 Pigeon 2 Troubleshooting

A functional limitation was discovered in Pigeon 2s manufactured in September of 2022. When used on a CANivore (CAN FD) Bus, the Pigeon 2 may not transmit CAN FD frames correctly. As a result, you may find that all CAN device LEDs go red when the Pigeon 2 is in-circuit and powered.

A firmware fix has been published, to update the firmware of an affected Pigeon 2, one of the below options can be used.

10.1.1 Option 1: Workaround with Tuner X

Note: If you do not see the below option, then Tuner X is likely older than version **2023.1.5.0**.

A new section in Tuner X *Settings* labeled **Pigeon 2 Workaround** has been added. When the *Execute Pigeon 2 workaround* button is pressed, all CANivores will enter a special mode that allows them to see the offending Pigeon 2s. This mode is reverted when the CANivore is power cycled.

Once the workaround has been applied, the device will show up in the *Devices* menu and the LED should be alternating green/orange. *Field-upgrade* the firmware version and power cycle the CANivore.

10.1.2 Option 2: Connect to the roboRIO Bus

Connect the Pigeon 2 to the roboRIO CAN Bus and *field-upgrade* the firmware version.

Note: We recommend power cycling Pigeon after moving CAN bus leads from CANivore to roboRIO CAN bus to ensure a clean transition.

Store Page   CAD, Firmware and purchase instructions.

Hardware User Manual   Wiring and mount instructions in PDF format.

10.2 Status Light Reference

LED Color	Blink Pattern	Description
Off		Pigeon 2.0 is not powered.
Yellow/(	Only a single LED will blink with this pattern.	Device is in boot-loader, most likely because firmware upgrading has failed. Inspect CAN bus wiring and retry firmware upgrading. If device has valid firmware, turn device off, wait 10 seconds, and turn device back on.
Red/(	Alternating Red/Green	Device is not licensed. License device in Phoenix Tuner.
Red/'	LEDs are never off - one of the two colors are **always illuminated**	Hardware is damaged
Red Blink		Check CAN bus health and connection to the Pigeon 2.0
Yellow	Alternate Blinking	CAN bus detected but robot controller is not detected (or Pigeon 2.0 is not referenced in code)
Yellow	Simultaneous Blinking	CAN bus detected, robot is disabled.
Gree Blink		CAN bus detected. Robot is enabled

10.3 Mount Calibration

It's recommended to perform a mount calibration when placement of the Pigeon 2.0 has been finalized. This can be done via the *Calibration page* in Tuner X.

Pigeon 2 Calibration
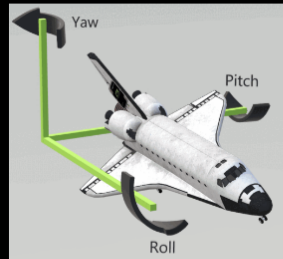
The Pigeon 2.0 Mount Calibration page provides the means to mount calibrate a Pigeon 2.0, considering its unique mounting angle and "zeroing" it out by configuring very accurate Mount Pose angles. It may be preferable to instead configure the mount orientation by defining which principal axis is up and forward. In this scenario, use Phoenix API to *configMountPose(AxisDirection, AxisDirection)*.

Mount Calibration is performed in 3 steps.

1. Keep the robot steady and flat, with the Pigeon 2.0 rigidly fixed to where it will be on the robot.

2. Begin the mount calibration process. The moment the button is clicked, the Pigeon 2.0 will save its current orientation and remember the Pitch and Roll necessary to zero itself. Then, it will begin to rapid blink, similarly to the blink command. You have 8 seconds to complete step 3.

3. Tilt the robot in the positive roll direction.

Roll is defined as to the right of the robot. You only need to roll it 20 degrees, at which point Pigeon 2.0 will rapid-blink green to indicate success.

After these steps, Pigeon 2.0 is mount-calibrated, and should report near 0 pitch and roll. Confirm this by performing a self-test snapshot and seeing pitch-roll value close to 0. Take note of the mount-pose values at the bottom of the self-test snapshot and configure Pigeon 2.0 with those values in your robot application to ensure the mount-calibration exists even in the event of a factory-default or Pigeon 2.0 swap.

Step 1: Make sure Pigeon 2.0 is rigidly mounted to robot, flat, level, and still.

Step 2: Press button to begin Calibration. LEDs will rapidly blink orange.

Step 3: Tilt Robot to the right (Positive Roll), as shown in the above GIF. The LEDs will rapid-blink green once robot is tilted far enough.

Begin Mount Calibration

CANcoder

Important: As of late August 2022, there are multiple hardware versions of CANcoder available. This is due to the ongoing worldwide chip shortage causing CTR Electronics to replace the original processor with a substitute. This new version of CANcoder requires a different firmware, but is otherwise functionally identical to the original. Details on checking the version can be found in the *device details section*.

The CANcoder is the next evolution in the line of CTRE magnetic encoder products. As its name implies, this product is a rotary magnetic encoder that communicates over the CAN bus. Supporting CAN FD and CAN 2.0, this product provides the same position and velocity with the same resolutions you've come to expect from the SRX Magnetic Encoder.

Store Page   CAD, Firmware and purchase instructions.

Hardware User Manual   Wiring and mount instructions in PDF format.

11.1 Status Light Reference

Note: Users wishing to test magnet placement must wait 8 seconds after boot for the LEDs to blink the magnet placement status.

LED Color	Led Brightness	CAN bus Detection	Magnet Field Strength	Description
Off	•	•	•	CANcoder is not powered/plugged in. Check power cabling to the CANcoder.
Yellow/Green	Bright	•	•	Device is in boot-loader, most likely because firmware upgrading has failed. Inspect CAN bus wiring and retry firmware upgrading. If device has valid firmware, turn device off, wait 10 seconds, and turn device back on.
Slow Red Blink	Bright	CAN bus has been lost.	•	Check CAN bus health and connection to the CANcoder.
Red/Green	Bright	•	•	Device is not licensed. Please license device in Phoenix Tuner.
Rapid Red Blink	Dim	CAN bus never detected since boot	<25mT or >135mT	Magnet is out of range
Rapid Yellow Blink	Dim	CAN bus never detected since boot	25-45mT or 75-136mT	Magnet in range with slightly reduced accuracy
Rapid Green Blink	Dim	CAN bus never detected since boot	Magnet in range	Magnet in range
Rapid Red Blink	Bright	CAN bus present	<25mT or >135mT	Magnet is out of range
Rapid Yellow Blink	Bright	CAN bus present	25-45mT or 75-136mT	Magnet in range with slightly reduced accuracy
Rapid Green Blink	Bright	CAN bus present	Magnet in range	Magnet in range

11.2 Magnet Placement

Using the CANcoder User's Guide, verify that magnet placement is correct for the CANcoder.

11.3 Verifying Sensor Direction

CANcoder sensor direction can be configured via the *Config* page in Phoenix Tuner X.
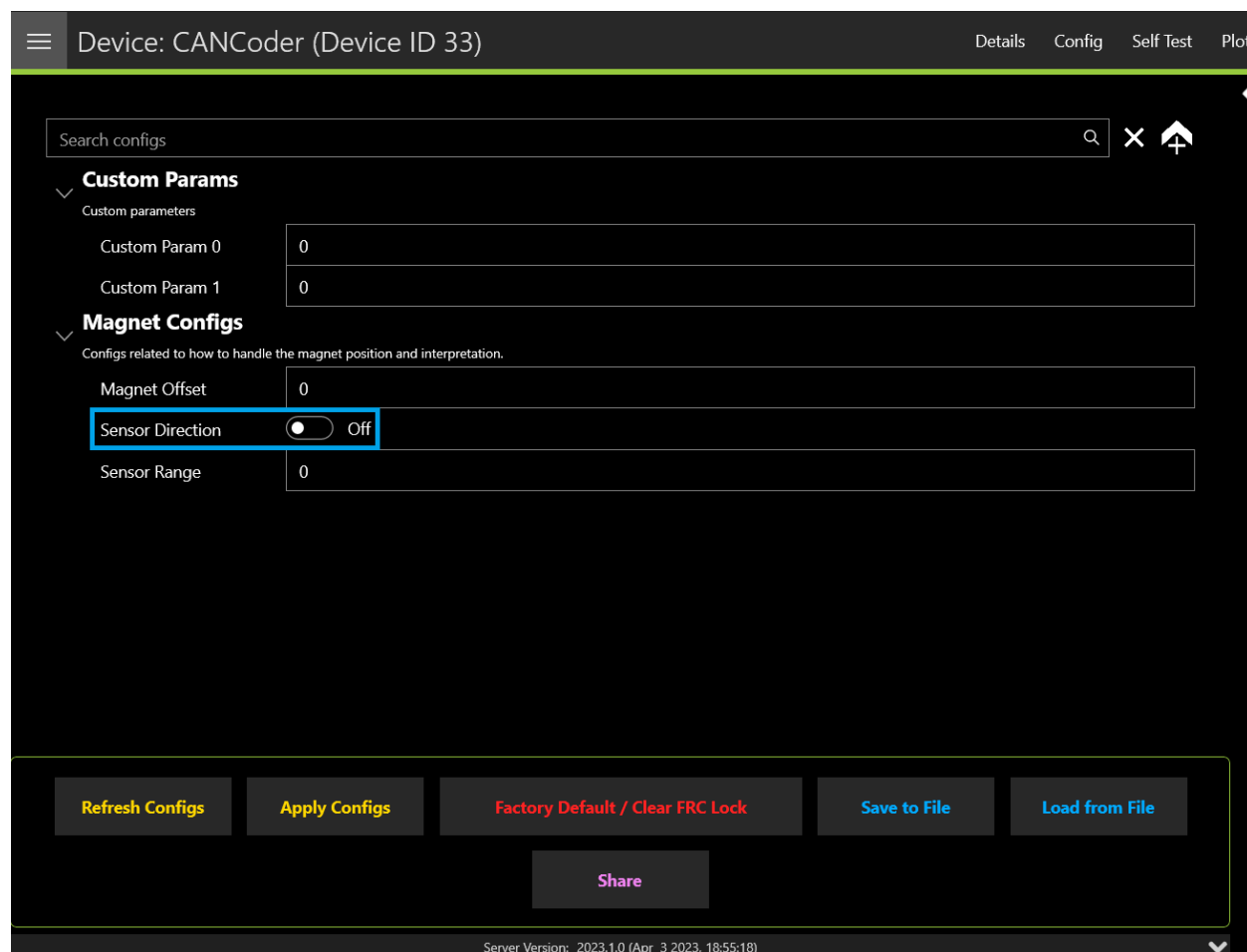
API Usage

This section serves to provide basic API usage for the Phoenix 6 API. For full details, please visit the API docs (Java, C++).

Important: While Phoenix 6 and Phoenix 5 devices may exist on the same CAN bus and same robot project, each robot project **must** use the API tied to the device firmware version. This means Phoenix 5 devices **must** use the Phoenix 5 API, and Phoenix 6 devices **must** use the Phoenix 6 API.

There are three major components to the Phoenix 6 API:

Configs   Configs represent a **persistent configuration** for a device. For example, closed-loop gains.

Control Requests   Control Requests represent the **output** of a device, typically a motor controller.

Signals   Signals represent **data retrieved** from a device. This can be velocity, position, yaw, pitch, roll, temperature, etc.

- *API Overview*

    – Details a high level overview of what makes up the Phoenix 6 API.

- *Configuration*

    – Describes configuring device configs via code.

- *Control Requests*

    – Highlights using control requests to control the open and closed loop functionality of actuators such as the TalonFX.

- *Status Signals*

    – Details using status signals to retrieve sensor data from devices.

- *Device Faults*

    – Documents how faults are used to indicate device hardware status.

- *Enabling Actuators*

> **–** Information on the FRC Lock safety feature and enabling actuators.

- *Actuator Limits*

> **–** Documents how to retrieve and configure software and hardware actuator limits.

12.1 API Overview

The Phoenix 6 API resides in the `com.ctre.phoenix6` package in Java and the `ctre::phoenix6` namespace in C++. The API is then further organized into smaller packages and namespaces that group together similar types of classes and functions:

- `configs` - classes related to device *configuration*
- `controls` - classes related to device *control*
- `hardware` - the device hardware classes, such as `TalonFX`
- `signals` - enumeration types for device *signals*
- `sim` - classes related to device *simulation*

12.1.1 C++ IntelliSense

In C++, this namespace structure has the advantage of cleaning up IntelliSense when searching for classes:

```cpp
// first use the ctre::phoenix6 namespace
using namespace ctre::phoenix6;

// now types are organized cleanly by namespace
hardware::TalonFX m_talonFX{0};
sim::TalonFXSimState& m_talonFXSim{m_talonFX.GetSimState()};

configs::TalonFXConfiguration m_talonFXConfig{};
signals::InvertedValue m_talonFXInverted{signals::InvertedValue::CounterClockwise_
↪Positive};

controls::DutyCycleOut m_talonFXOut{0};
```

All C++ code examples in this documentation will assume the presence of `using namespace ctre::phoenix6;`.

12.2 Configuration

Devices support persistent settings through the use of "configs".

Tip: Configs can also be configured using Phoenix Tuner X. See *Tuner Configs* for more information.

12.2.1 Configuration Objects

There are device-specific `Configuration` classes that group configuration data of devices in a meaningful way. These classes are Passive Data Structures. One example is `TalonFXCon-figuration`, which has subgroups of configs such as `MotorOutputConfigs`. The configs can be modified through public member variables of the `Configuration` object. The complete list of configuration objects can be found in the API documentation (Java, C++).

Java

```
var talonFXConfigs = new TalonFXConfiguration();
```

C++

```
configs::TalonFXConfiguration talonFXConfigs{};
```

Future Proofing Configs

There is a corner case with configs where the device may have firmware with newer configs that didn't exist when the version of the API was built. To account for this problem, device `Configuration` objects have a `FutureProofConfigs` (Java, C++) field.

12.2.2 Configurator API

Device objects have a `getConfigurator()` method that returns a device-specific `Configura-tor` object. The `Configurator` is used to retrieve, apply, and factory default the configs of a device.

Note: The `getConfigurator()` routine can be called frequently without any performance implications.

The device-specific configurators have type-specific overloads that allow for the widest variety of device-compatible configs. As a result, the caller can pass the entire device `Configuration` object or just the relevant subgroup of configs to the `Configurator` API.

Java

```
var talonFXConfigurator = m_talonFX.getConfigurator();
```

C++

```
auto& talonFXConfigurator = m_talonFX.GetConfigurator();
```

Reading Configs

To read configs stored in a device, use the `refresh()` method to update a `Configuration` object. The example below demonstrates retrieving a full `TalonFXConfiguration` (Java, C++) object from a `TalonFX` device.

> **Warning:** `refresh()` is a blocking API call that waits on the device to respond. Calling `refresh()` periodically may slow down the execution time of the periodic function, as it will always wait up to `defaultTimeoutSeconds` (Java, C++) for the response when no timeout parameter is specified.

Java

```
var talonFXConfigurator = m_talonFX.getConfigurator();
var talonFXConfigs = new TalonFXConfiguration();

// optional timeout (in seconds) as a second optional parameter
talonFXConfigurator.refresh(talonFXConfigs);
```

C++

```
auto& talonFXConfigurator = m_talonFX.GetConfigurator();
configs::TalonFXConfiguration talonFXConfigs{};

// optional timeout (in seconds) as a second optional parameter
talonFXConfigurator.Refresh(talonFXConfigs);
```

Applying Configs

Configs can be applied to a device by calling `apply()` on the `Configurator` with a `Configuration` object.

> **Warning:** `apply()` is a blocking API call that waits on the device to respond. Calling `apply()` periodically may slow down the execution time of the periodic function, as it will always wait up to `defaultTimeoutSeconds` (Java, C++) for the response when no timeout parameter is specified.

Java

```java
var talonFXConfigurator = m_talonFX.getConfigurator();
var motorConfigs = new MotorOutputConfigs();

// set invert to CW+ and apply config change
motorConfigs.Inverted = InvertedValue.Clockwise_Positive;
talonFXConfigurator.apply(motorConfigs);
```

C++

```cpp
auto& talonFXConfigurator = m_talonFX.GetConfigurator();
configs::MotorOutputConfigs motorConfigs{};

// set invert to CW+ and apply config change
motorConfigs.Inverted = signals::InvertedValue::Clockwise_Positive;
talonFXConfigurator.Apply(motorConfigs);
```

Tip: To modify a single configuration value without affecting the other configs, users can call refresh() after constructing the config object, or users can cache the config object and reuse it for future calls to apply().

Factory Default

A newly-created Configuration object contains the default configuration values of a device. Passing this newly-created Configuration object to the device Configurator will factory default the device's configs.

Java

```java
m_talonFX.getConfigurator().apply(new TalonFXConfiguration());
```

C++

```cpp
m_talonFX.GetConfigurator().Apply(configs::TalonFXConfiguration{});
```

12.3 Control Requests

Control Requests represent the **output** of a device. A list of control requests can be found in the API docs (Java, C++).

Note: Phoenix 6 utilizes the C++ units library when applicable.

12.3.1 Applying a Control Request

Control requests can be applied by calling `setControl()` on the motor object. `setControl()` returns a `StatusCode` (Java, C++) enum that represents success state. A successful request will return `StatusCode.OK`.

Java

```java
// Command m_motor to 100% of duty cycle
m_motor.setControl(new DutyCycleOut(1.0));
```

C++

```cpp
// Command m_motor to 100% of duty cycle
m_motor.SetControl(controls::DutyCycleOut{1.0});
```

12.3.2 Modifying a Control Request

Control requests are mutable, so they can be saved in a member variable and reused. For example, `DutyCycleOut` (Java, C++) has an `Output` member variable that can be manipulated, thus changing the output DutyCycle (proportion of supply voltage).

Note: Java users should reuse control requests to prevent excessive invocation of the Garbage Collector.

Java

```java
var motorRequest = new DutyCycleOut(0.0);

motorRequest.Output = 1.0;
m_motor.setControl(motorRequest);
```

C++

```cpp
controls::DutyCycleOut motorRequest{0.0};

motorRequest.Output = 1.0;
m_motor.SetControl(motorRequest);
```

Method Chaining API

Control requests also supports modification using method chaining. This can be useful for mutating multiple values of a control request.

Java

```java
// initialize torque current FOC request with 0 amps
var motorRequest = new TorqueCurrentFOC(0);

// mutate request with output of 10 amps and max duty cycle 0.5
m_motor.SetControl(motorRequest.withOutputAmps(10).withMaxDutyCycle(0.5));
```

C++

```cpp
// initialize torque current FOC request with 0 amps
controls::TorqueCurrentFOC motorRequest{0_A};

// mutate request with output of 10 amps and max duty cycle 0.5
m_motor.SetControl(motorRequest.WithOutputAmps(10_A).WithMaxDutyCycle(0.5));
```

12.3.3 Changing Update Frequency

Control requests are automatically transmitted at a fixed update frequency. This update frequency can be modified by changing the UpdateFreqHz (Java, C++) field of the control request before sending it to the device.

Java

```java
// create a duty cycle request
var motorRequest = new DutyCycleOut(0);
// reduce the update frequency to 50 Hz
motorRequest.UpdateFreqHz = 50;
```

C++

```cpp
// create a duty cycle request
controls::DutyCycleOut motorRequest{0};
// reduce the update frequency to 50 Hz
motorRequest.UpdateFreqHz = 50;
```

Tip: UpdateFreqHz can be set to 0 Hz to synchronously one-shot the control request. In this case, users must ensure the control request is sent periodically in their robot code. Therefore, we recommend users call setControl no slower than 20 Hz (50 ms) when the control is one-shot.

12.4 Status Signals

Signals represent live data reported by a device; these can be yaw, position, etc. To make use of the live data, users need to know the value, timestamp, latency, units, and error condition of the data. Additionally, users may need to synchronize with fresh data to minimize latency.

12.4.1 StatusSignal

The StatusSignal (Java, C++) is a signal object that provides APIs to address all of the requirements listed above.

The device object provides getters for all available signals. Each getter returns a StatusSignal that is typed appropriately for the signal.

Note: The device getters return a cached StatusSignal. As a result, frequently calling the getter does not influence RAM performance.

Java

```java
var supplyVoltageSignal = m_device.getSupplyVoltage();
```

C++

```cpp
auto& supplyVoltageSignal = m_device.GetSupplyVoltage();
```

The value of the signal can be retrieved from the StatusSignal by calling getValue().

Java

```
var supplyVoltage = supplyVoltageSignal.getValue();
```

C++

```
auto supplyVoltage = supplyVoltageSignal.GetValue();
```

Note: Phoenix 6 utilizes the C++ units library when applicable.

The `StatusCode` (Java, C++) of the signal can be retrieved by calling `getError()`. This can be used to determine if the device is not present on the CAN bus.

Note: If a status signal is not available on the CAN bus, an error will be reported to the Driver Station.

Refreshing the Signal Value

The device `StatusSignal` getters implicitly refresh the cached signal values. However, if the user application caches the `StatusSignal` object, the `refresh()` method must be called to fetch fresh data.

Tip: The `refresh()` method can be method-chained. As a result, you can call `refresh()` and `getValue()` on one line.

Java

```
supplyVoltageSignal.refresh();
```

C++

```
supplyVoltageSignal.Refresh();
```

Waiting for Signal Updates

Instead of using the latest value, the user can instead opt to synchronously wait for a signal update. `StatusSignal` provides a `waitForUpdate(timeoutSec)` method that will block the current robot loop until the signal is retrieved or the timeout has been exceeded. This replaces the need to call `refresh()` on cached `StatusSignal` objects.

Tip: If you want to zero your sensors, you can use this API to ensure the set operation has completed before continuing program flow.

Tip: The `waitForUpdate()` method can be method-chained. As a result, you can call `wait-ForUpdate()` and `getValue()` on one line.

Java

```java
// wait up to 1 robot loop iteration (20ms) for fresh data
supplyVoltageSignal.waitForUpdate(0.020);
```

C++

```cpp
// wait up to 1 robot loop iteration (20ms) for fresh data
supplyVoltageSignal.WaitForUpdate(20_ms);
```

Changing Update Frequency

All signals can have their update frequency configured via the `setUpdateFrequency()` method.

Warning: Increasing signal frequency will also increase CAN bus utilization, which can cause indeterminate behavior at high utilization rates (>90%). This is less of a concern when using CANivore, which uses the higher-bandwidth CAN FD bus.

Java

```java
// slow down supply voltage reporting to 10 Hz
supplyVoltageSignal.setUpdateFrequency(10);
```

C++

```
// slow down supply voltage reporting to 10 Hz
supplyVoltageSignal.SetUpdateFrequency(10_Hz);
```

Important: Currently in Phoenix 6, when different status signal frequencies are specified for signals that share a status frame, the last specified frequency is applied to the status frame. As a result, users should apply the slowest status frame frequencies first and the fastest frequencies last.

Timestamps

The timestamps of a `StatusSignal` can be retrieved by calling `getAllTimestamps()`, which returns a collection of `Timestamp` (Java, C++) objects. The `Timestamp` objects can be used to perform latency compensation math.

12.4.2 CANivore Timesync

Important: CANivore Timesync requires the devices or the CANivore to be *Pro licensed*.

When using CANivore, the attached CAN devices will automatically synchronize their time bases. This allows devices to sample and publish their signals in a synchronized manner.

Users can synchronously wait for these signals to update using `BaseStatusSignal.waitForAll()` (Java, C++).

Tip: `waitForAll()` can be used with a timeout of zero to perform a non-blocking refresh on all signals passed in.

Because the devices are synchronized, time-critical signals are sampled and published on the same schedule. This combined with the `waitForAll()` routine means applications can considerably reduce the latency of the timesync signals. This is particularly useful for multi-device mechanisms, such as swerve odometry.

Note: When using a non-zero timeout, the signals passed into `waitForAll()` should have the same update frequency for synchronous data acquisition. This can be done by calling `setUpdateFrequency()` or by referring to the API documentation.

The diagram below demonstrates the benefits of using timesync to synchronously acquire signals from multiple devices.
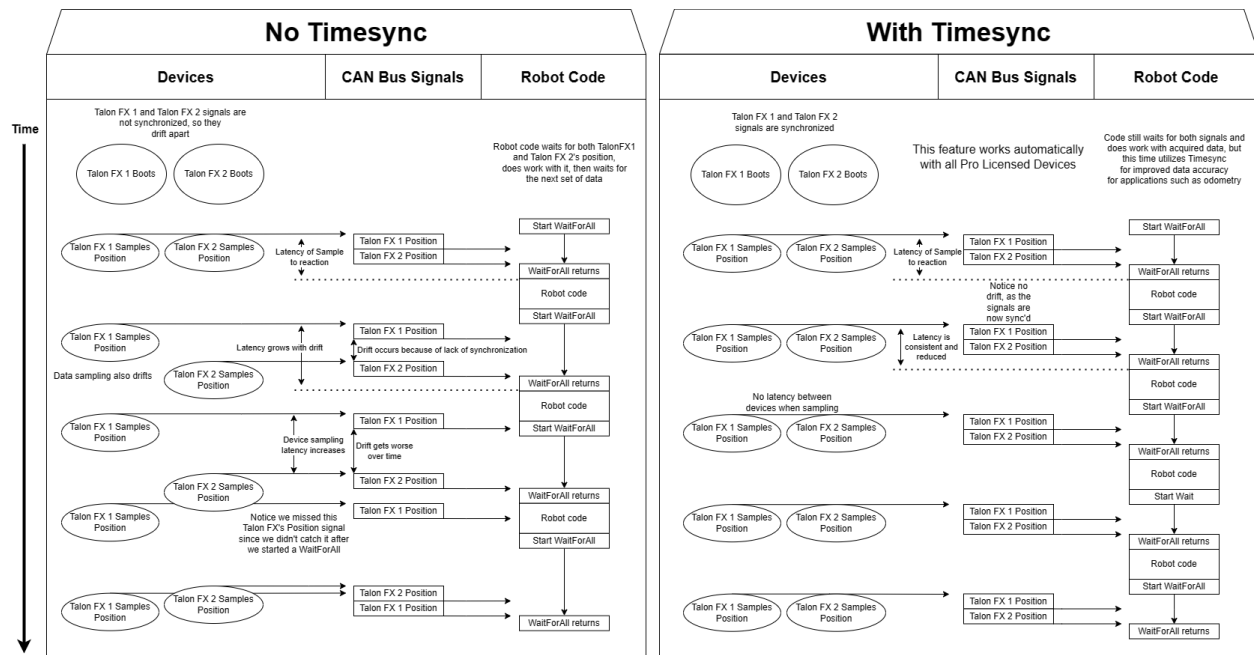
The following signals are time-synchronized:

- TalonFX

    - All Signals

- CANcoder

    - All Signals

- Pigeon 2.0

    - Yaw, Pitch, & Roll

    - Quaternion

    - Gravity Vector

    - Accum Gyro

    - Angular Rate

    - Accelerometer

    - Temperature

Java

```java
var talonFXPositionSignal = m_talonFX.getPosition();
var cancoderPositionSignal = m_cancoder.getPosition();
var pigeon2YawSignal = m_pigeon2.getYaw();

BaseStatusSignal.waitForAll(0.020, talonFXPositionSignal, cancoderPositionSignal,
↪pigeon2YawSignal);
```

C++

```cpp
auto& talonFXPositionSignal = m_talonFX.GetPosition();
auto& cancoderPositionSignal = m_cancoder.GetPosition();
auto& pigeon2YawSignal = m_pigeon2.GetYaw();

BaseStatusSignal::WaitForAll(20_ms, {&talonFXPositionSignal, &cancoderPositionSignal,␣
↪&pigeon2YawSignal});
```

12.4.3 Latency Compensation

Users can perform latency compensation using BaseStatusSignal.
getLatencyCompensatedValue() (Java, C++).

Important: getLatencyCompensatedValue() does not automatically refresh the signals. As
a result, the user must ensure the signal and signalSlope parameters are refreshed before
retrieving a compensated value.

Java

```java
double compensatedTurns = BaseStatusSignal.getLatencyCompensatedValue(m_motor.
↪getPosition(), m_motor.getVelocity());
```

C++

```cpp
auto compensatedTurns = BaseStatusSignal::GetLatencyCompensatedValue(m_motor.
↪GetPosition(), m_motor.GetVelocity());
```

12.4.4 SignalMeasurement

All StatusSignal objects have a getDataCopy() method that returns a new SignalMeasure-
ment (Java, C++) object. SignalMeasurement is a Passive Data Structure that provides all
the information about a signal at the time of the getDataCopy() call, which can be useful for
data logging.

> **Warning:** getDataCopy() returns a **new** SignalMeasurement object every call. **Java**
> users should **avoid** using this API in RAM-constrained applications.

12.5 Device Faults

"Faults" are status indicators on CTR Electronics CAN devices that indicate a certain behavior or event has occurred. Faults do not directly affect the behavior of a device; instead, they indicate the device's current status and highlight potential issues.

Faults are stored in two fashions. There are "live" faults, which are reported in real-time, and "sticky" faults, which assert persistently and stay asserted until they are manually cleared (like trouble codes in a vehicle).

Sticky Faults can be cleared by clicking the *Blink/Clear Faults* button in Phoenix Tuner X, or by calling `clearStickyFaults()` on the device in the robot program. A regular fault can only be cleared when the offending problem has been resolved.

12.5.1 Using API to Retrieve Faults

Faults can also be retrieved in API using the getFault_*() (regular) or getStickyFault_*() (sticky) methods on the device object. This can be useful for diagnostics or error handling.

Java

```java
var faulted = m_cancoder.getFault_BadMagnet().getValue();

if (faulted) {
    // do action when bad magnet fault is set
}
```

C++

```cpp
auto faulted = m_cancoder.GetFault_BadMagnet().GetValue();

if (faulted) {
    // do action when bad magnet fault is set
}
```

A list of possible faults can be found in the API documentation for each device.

12.6 Enabling Actuators

CTR Electronics supported actuators have a safety feature where they will automatically disable output if they have not recently received an enable signal.

12.6.1 FRC Applications

In FRC applications, the enable signal is automatically sent to devices based on the Driver Station enable signal. This includes controlling devices in Phoenix Tuner X.

> **Warning:** The device *FRC Lock* must be cleared to control devices in hardware-attached simulation.

12.6.2 Non-FRC Applications

In non-FRC applications, Unmanaged.feedEnable() **must be called periodically** to enable actuators.

> **Warning:** The device *FRC Lock* must be cleared to control devices.

Java

```
// feed the enable signal, timeout after 100ms
Unmanaged.feedEnable(100);
```

C++

```
// feed the enable signal, timeout after 100ms
unmanaged::FeedEnable(100);
```

This must also be called to control devices in Phoenix Tuner X.

Tip: The Tuner X CANivore USB server automatically calls `Unmanaged.feedEnable()` when control is enabled.

12.6.3 FRC Lock

When a device is connected to a roboRIO for use in FRC, the device becomes FRC-locked and will require the Driver Station enable signal for actuation. The device FRC lock can be reset by factory-defaulting the device in *Phoenix Tuner X*.

12.7 Actuator Limits

CTR Electronics actuators, such as the TalonFX, support various kinds of hardware and software limits.

Documentation on wiring limit switches can be found *here*.

12.7.1 Retrieving Limit Switch State

The state of the forward or reverse limit switch can be retrieved from the API via `getForwardLimit()` and `getReverseLimit()`.

Java

```
var forwardLimit = m_motor.getForwardLimit();

if (forwardLimit.getValue() == ForwardLimitValue.ClosedToGround) {
    // do action when forward limit is closed
}
```

C++

```cpp
auto& forwardLimit = m_motor.GetForwardLimit();

if (forwardLimit.GetValue() == signals::ForwardLimitValue::ClosedToGround) {
    // do action when forward limit is closed
}
```

13

<div style="background:#d9d9d9">

Device Specific

</div>

This section is intended to highlight any device specific API functionality. This include features such as the TalonFX + CANcoder fusion, details on using TalonFX *Control Requests* and more.

13.1 TalonFX

13.1.1 Introduction to TalonFX Control

The TalonFX has a variety of open-loop and closed-loop control requests and supports Field Oriented Control.

Control Output Types

The TalonFX supports three base control output types: DutyCycle, Voltage, and TorqueCurrentFOC.

DutyCycle

A DutyCycle control request outputs a proportion of the supply voltage, which typically ranges from -1.0 to 1.0, inclusive. This control output type is typically used in systems where it is important to be capable of running at the maximum speed possible, such as in a typical robot drivetrain.

Voltage

A Voltage control request directly controls the output voltage of the motor. The output voltage is capped by the supply voltage to the device. Since the output of a Voltage control request is typically unaffected by the supply voltage, this control output type results in more stable and reproducible behavior than a DutyCycle control request.

TorqueCurrentFOC

Important: This feature requires the device to be *Pro licensed*.

A TorqueCurrentFOC control request uses Field Oriented Control to directly control the output current of the motor. Unlike the other control output types, where output roughly controls the velocity of the motor, a TorqueCurrentFOC request controls the **acceleration** of the motor.

Field Oriented Control

Important: This feature requires the device to be *Pro licensed*.

Field Oriented Control (FOC) is a commutation mode that increases peak power by ~15%. All control modes that optionally support FOC have an `EnableFOC` field (Java, C++). There are also control types that require FOC, such as TorqueCurrentFOC.

13.1.2 Open-Loop Control

Open-Loop control typically refers to directly controlling device output.

There are open-loop control requests for all TalonFX control output types. With the exception of FOC-only control requests, all open-loop control requests follow the naming pattern `{ControlOutputType}Out`. For example, the open-loop Voltage control request is called `VoltageOut`. FOC-only control requests follow the naming pattern `{ControlOutputType}`.

Java

```java
// users should reuse control requests when possible
var leftRequest = new DutyCycleOut(0.0);
var rightRequest = new DutyCycleOut(0.0);

// retrieve joystick inputs
var forward = -m_driverJoy.getLeftY();
var turn = m_driverJoy.getRightX();

// calculate motor outputs, utilizes a "arcade" style of driving;
// where left Y controls forward and right X controls rotation/turn
var leftOut = forward + turn;
```

```
var rightOut = forward - turn;

// set request to motor controller
m_leftLeader.setControl(leftRequest.withOutput(leftOut));
m_rightLeader.setControl(rightRequest.withOutput(rightOut));
```

C++

```
// users should reuse control requests when possible
controls::DutyCycleOut leftRequest{0.0};
controls::DutyCycleOut rightRequest{0.0};

// retrieve joystick inputs
auto forward = -m_driverJoy.GetLeftY();
auto turn = m_driverJoy.GetRightX();

// calculate motor outputs, utilizes a "arcade" style of driving;
// where left Y controls forward and right X controls rotation/turn
auto leftOut = forward + turn;
auto rightOut = forward - turn;

// set request to motor controller
m_leftLeader.SetControl(leftRequest.WithOutput(leftOut));
m_rightLeader.SetControl(rightRequest.WithOutput(rightOut));
```

13.1.3 Closed-Loop Control

Closed-loop control typically refers to control of a motor that relies on sensor data to adjust based on error. Systems/mechanisms that rely on maintaining a certain position or velocity achieve this state using closed-loop control. This is achieved by feedback (PID) and feedforward control. Closed-loop control can be performed on the robot controller or on the individual motor controllers. The benefit of onboard closed-loop control is that there is no sensor latency and 1 kHz update frequency. This can result in a more responsive output compared to running the closed-loop on the robot controller.

Since closed-loop control changes based on the dynamics of the system (velocity, mass, CoG, etc.), closed-loop relies on PID and feedforward parameters. These parameters are configured either via *Tuner Configs* or in *code*. The parameters can be determined using System Identification (such as with WPILib SysID) or through manual tuning.

Manual tuning typically follows this process:

1. Set K_p, K_i and K_d to zero.

2. Increase K_p until the output starts to oscillate around the setpoint.

3. Increase K_d as much as possible without introducing jittering to the response.

All closed-loop control requests follow the naming pattern {ClosedLoop-Mode}{ControlOutputType}. For example, the VelocityVoltage control request performs a velocity closed-loop using voltage output.

Gain Slots

It may be useful to switch between presets of gains in a motor controller, so the TalonFX supports multiple gain slots. All closed-loop control requests have a member variable `Slot` that can be assigned an integer ID to select the set of gains used by the closed-loop. The gain slots can be *configured in code* using `Slot*Configs` (Java, C++) objects.

Velocity Control

A Velocity closed loop can be used to maintain a target velocity (in rotations per second). This can be useful for controlling flywheels, where a velocity needs to be maintained for accurate shooting.

Velocity closed loop is currently supported for all base *control output types*. The units of the output is determined by the control output type.

In a Velocity closed loop, the gains should be configured as follows:

- K_s - output to overcome static friction (output)
- K_v - output per unit of requested velocity (output/rps)
- K_p - output per unit of error in velocity (output/rps)
- K_i - output per unit of integrated error in velocity (output/rotation)
- K_d - output per unit of error derivative in velocity (output/(rps/s))

Java

```
// in init function, set slot 0 gains
var slot0Configs = new Slot0Configs();
slot0Configs.kS = 0.05; // Add 0.05 V output to overcome static friction
slot0Configs.kV = 0.12; // A velocity target of 1 rps results in 0.12 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0.5; // An error of 1 rps increases output by 0.5 V each second
slot0Configs.kD = 0.01; // An acceleration of 1 rps/s results in 0.01 V output

m_talonFX.getConfigurator().apply(slot0Configs);
```

C++

```
// in init function, set slot 0 gains
configs::Slot0Configs slot0Configs{};
slot0Configs.kS = 0.05; // Add 0.05 V output to overcome static friction
slot0Configs.kV = 0.12; // A velocity target of 1 rps results in 0.12 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0.5; // An error of 1 rps increases output by 0.5 V each second
slot0Configs.kD = 0.01; // An acceleration of 1 rps/s results in 0.01 V output

m_talonFX.GetConfigurator().Apply(slot0Configs);
```

Once the gains are configured, the Velocity closed loop control request can be sent to the TalonFX. The control request object has an optional feedforward term that can be used to add an arbitrary value to the output, which can be useful to account for the effects of gravity.

Java

```java
// create a velocity closed-loop request, voltage output, slot 0 configs
var request = new VelocityVoltage(0).withSlot(0);

// set velocity to 8 rps, add 0.5 V to overcome gravity
m_talonFX.setControl(request.withVelocity(8).withFeedForward(0.5));
```

C++

```cpp
// create a velocity closed-loop request, voltage output, slot 0 configs
auto request = controls::VelocityVoltage{0_tps}.WithSlot(0);

// set velocity to 8 rps, add 0.5 V to overcome gravity
m_talonFX.SetControl(request.WithVelocity(8_tps).WithFeedForward(0.5_V));
```

Converting from Meters

In some applications, it may be useful to translate between meters and rotations. This can be done using the following equation:

$$rotations = \frac{meters}{\pi \cdot wheelDiameter} \cdot gearRatio$$

where `meters` is the target in meters, `wheelDiameter` is the diameter of the wheel in meters, and `gearRatio` is the gear ratio between the output shaft and the wheel.

This equation also works with converting velocity from m/s to rps.

Position Control

A Position closed loop can be used to target a specified motor position (in rotations).

Position closed loop is currently supported for all base *control output types*. The units of the output is determined by the control output type.

In a Position closed loop, the gains should be configured as follows:

- K_s - unused, as there is no target velocity
- K_v - unused, as there is no target velocity
- K_p - output per unit of error in position (output/rotation)
- K_i - output per unit of integrated error in position (output/(rotation*s))
- K_d - output per unit of error derivative in position (output/rps)

Java

```
// in init function, set slot 0 gains
var slot0Configs = new Slot0Configs();
slot0Configs.kP = 24; // An error of 0.5 rotations results in 12 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0.1; // A velocity of 1 rps results in 0.1 V output

m_talonFX.getConfigurator().apply(slot0Configs);
```

C++

```
// in init function, set slot 0 gains
configs::Slot0Configs slot0Configs{};
slot0Configs.kP = 24; // An error of 0.5 rotations results in 12 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0.1; // A velocity of 1 rps results in 0.1 V output

m_talonFX.GetConfigurator().Apply(slot0Configs);
```

Once the gains are configured, the Position closed loop control request can be sent to the TalonFX. The control request object has an optional feedforward term that can be used to add an arbitrary value to the output, which can be useful to account for the effects of gravity or friction.

Java

```
// create a position closed-loop request, voltage output, slot 0 configs
var request = new PositionVoltage(0).withSlot(0);

// set position to 10 rotations
m_talonFX.setControl(request.withPosition(10));
```

C++

```
// create a position closed-loop request, voltage output, slot 0 configs
auto request = controls::PositionVoltage{0_tr}.WithSlot(0);

// set position to 10 rotations
m_talonFX.SetControl(request.WithPosition(10_tr));
```

Motion Magic®

Motion Magic® is a control mode that provides the benefit of Motion Profiling without needing to generate motion profile trajectory points. When using Motion Magic®, the motor will move to a target position using a motion profile, while honoring the user specified acceleration, maximum velocity (cruise velocity), and optional jerk.

The benefits of this control mode over "simple" PID position closed-looping are:

- Control of the mechanism throughout the entire motion (as opposed to racing to the end target position)
- Control of the mechanism's inertia to ensure smooth transitions between set points
- Improved repeatability despite changes in battery load
- Improved repeatability despite changes in motor load

After gain/settings are determined, the robot controller only needs to periodically set the target position.

There is no general requirement to "wait for the profile to finish". However, the robot application can poll the sensor position and determine when the motion is finished if need be.

Motion Magic® functions by generating a trapezoidal/S-Curve velocity profile that does not exceed the specified cruise velocity, acceleration, or jerk. This is done automatically by the motor controller.

Note: If the remaining sensor distance to travel is small, the velocity may not reach cruise velocity as this would overshoot the target position. This is often referred to as a "triangle profile".

Example Trapezoidal Motion Profile
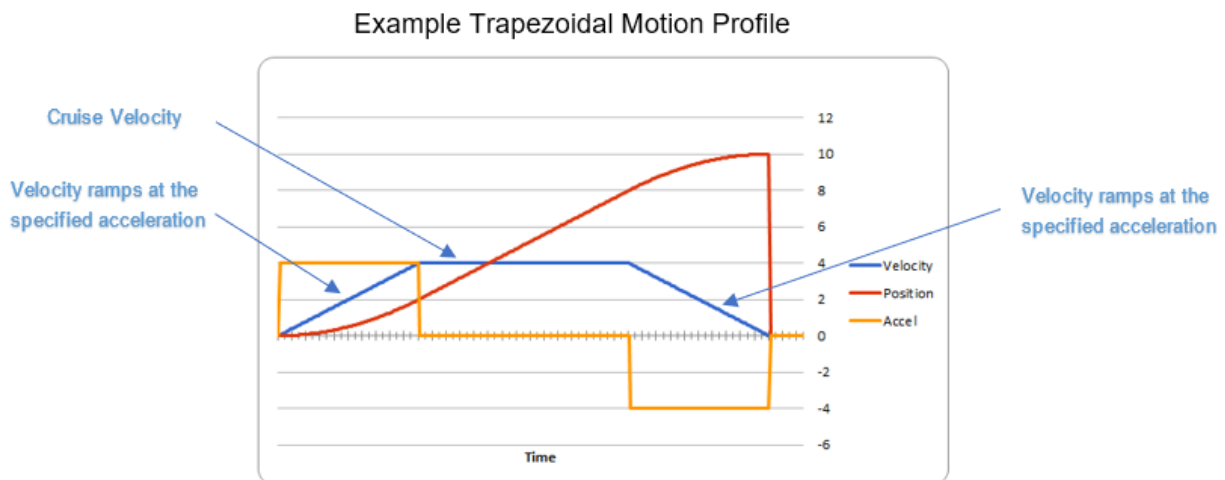
If the Motion Magic® jerk is set to a nonzero value, the generated velocity profile is no longer trapezoidal, but instead is a continuous S-Curve (corner points are smoothed).

An S-Curve profile has the following advantaged over a trapezoidal profile:

- Reducing oscillation of the mechanism.
- Maneuver is more deliberate and reproducible.

Note: The jerk control feature, by its nature, will increase the amount of time a movement requires. This can be compensated for by increasing the configured acceleration value.
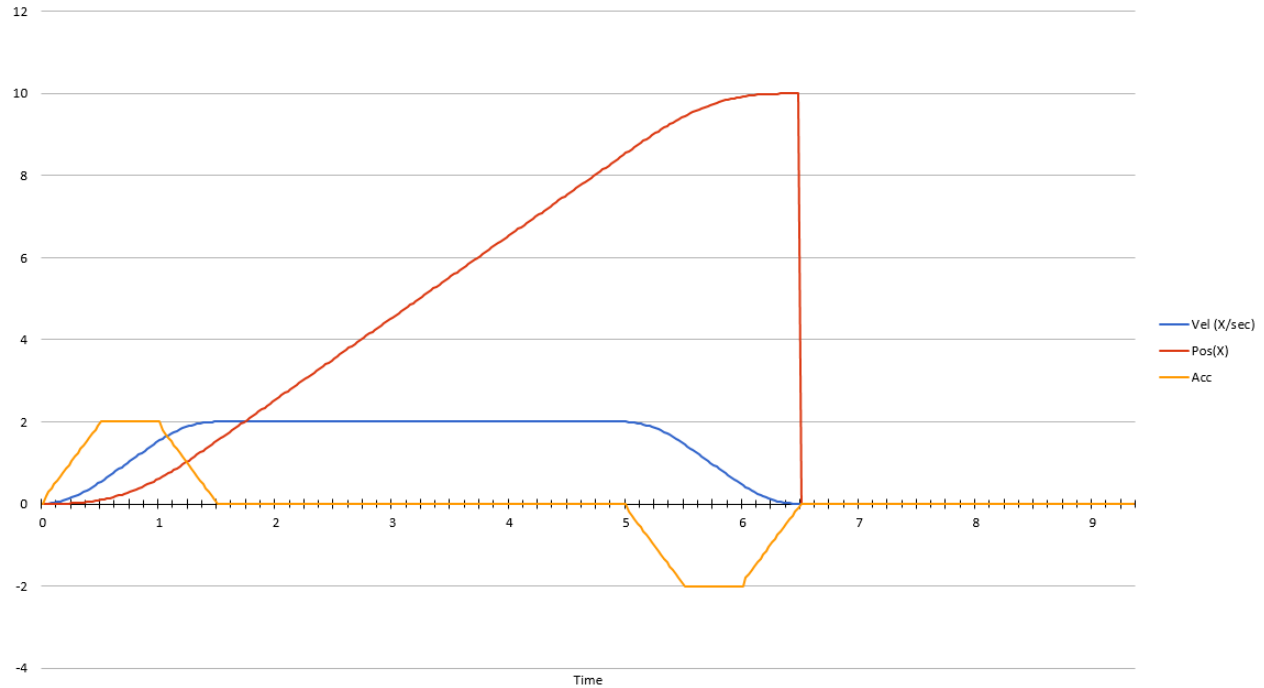
The following parameters must be set when controlling using Motion Magic®

- Cruise Velocity - peak/cruising velocity of the motion
- Acceleration - controls acceleration and deceleration rates during the beginning and end of motion
- Jerk - controls jerk, which is the derivative of acceleration

Using Motion Magic® in API

Motion Magic® is currently supported for all base *control output types*. The units of the output is determined by the control output type.

The Motion Magic® jerk, acceleration, and cruise velocity can be *configured in code* using a `MotionMagicConfigs` (Java, C++) object.

In Motion Magic®, the gains should be configured as follows:

- K_s - output to overcome static friction (output)
- K_v - output per unit of target velocity (output/rps)
- K_p - output per unit of error in position (output/rotation)
- K_i - output per unit of integrated error in position (output/(rotation*s))
- K_d - output per unit of error in velocity (output/rps)

Java

```
// in init function
var talonFXConfigs = new TalonFXConfiguration();

// set slot 0 gains
var slot0Configs = talonFXConfigs.Slot0Configs;
slot0Configs.kS = 0.25; // Add 0.25 V output to overcome static friction
slot0Configs.kV = 0.12; // A velocity target of 1 rps results in 0.12 V output
slot0Configs.kP = 4.8; // A position error of 2.5 rotations results in 12 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0.1; // A velocity error of 1 rps results in 0.1 V output

// set Motion Magic settings
var motionMagicConfigs = talonFXConfigs.MotionMagicConfigs;
motionMagicConfigs.MotionMagicCruiseVelocity = 80; // Target cruise velocity of 80 rps
motionMagicConfigs.MotionMagicAcceleration = 160; // Target acceleration of 160 rps/s
→(0.5 seconds)
motionMagicConfigs.MotionMagicJerk = 1600; // Target jerk of 1600 rps/s/s (0.1
→seconds)

m_talonFX.getConfigurator().apply(talonFXConfigs);
```

C++

```
// in init function
configs::TalonFXConfiguration talonFXConfigs{};

// set slot 0 gains
auto& slot0Configs = talonFXConfigs.Slot0Configs;
slot0Configs.kS = 0.25; // Add 0.25 V output to overcome static friction
slot0Configs.kV = 0.12; // A velocity target of 1 rps results in 0.12 V output
slot0Configs.kP = 4.8; // A position error of 2.5 rotations results in 12 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0.1; // A velocity error of 1 rps results in 0.1 V output

// set Motion Magic settings
auto& motionMagicConfigs = talonFXConfigs.MotionMagicConfigs;
motionMagicConfigs.MotionMagicCruiseVelocity = 80; // Target cruise velocity of 80 rps
motionMagicConfigs.MotionMagicAcceleration = 160; // Target acceleration of 160 rps/s
→(0.5 seconds)
motionMagicConfigs.MotionMagicJerk = 1600; // Target jerk of 1600 rps/s/s (0.1
→seconds)

m_talonFX.GetConfigurator().Apply(talonFXConfigs);
```

Tip: Motion Magic® supports modifying jerk and acceleration on the fly (requires firmware version 23.6.10.1 or newer).

Once the gains are configured, the Motion Magic® request can be sent to the TalonFX. The control request object has an optional feedforward term that can be used to add an arbitrary value to the output, which can be useful to account for the effects of gravity.

Java

```
// create a Motion Magic request, voltage output, slot 0 configs
var request = new MotionMagicVoltage(0).withSlot(0);

// set position to 10 rotations
m_talonFX.setControl(request.withPosition(10));
```

C++

```
// create a Motion Magic request, voltage output, slot 0 configs
auto request = controls::MotionMagicVoltage{0_tr}.WithSlot(0);

// set position to 10 rotations
m_talonFX.SetControl(request.WithPosition(10_tr));
```

Continuous Mechanism Wrap

A continuous mechanism is a mechanism with unlimited travel in any direction, and whose rotational position can be represented with multiple unique position values. Some examples of continuous mechanisms are swerve drive steer mechanisms or turrets (without cable management).

ContinuousWrap (Java, C++) is a mode of closed loop operation that enables the Talon to take the "shortest path" to a target position for a continuous mechanism. It does this by assuming that the mechanism is continuous within 1 rotation.

For example, if a Talon is currently at 2.1 rotations, it knows this is equivalent to every position that is exactly 1.0 rotations away from each other (3.1, 1.1, 0.1, -0.9, etc.). If that Talon is then commanded to a position of 0.8 rotations, instead of driving backwards 1.3 rotations or forwards 0.7 rotations, it will drive backwards 0.3 rotations to a target of 1.8 rotations.

Note: The ContinuousWrap config only affects the closed loop operation. Other signals such as Position are unaffected by this config.

In order to use this feature, the FeedbackConfigs (Java, C++) ratio configs must be configured so that the mechanism is properly described. An example is provided below, where there is a continuous mechanism with a 12.8:1 speed reduction between the rotor and mechanism.

When Using	RotorToSensorRatio	SensorToMechanismRatio
Fused CANcoder	12.8 *Rotor → CANcoder*	1 *CANcoder on Mechanism*
Internal Sensor	n/a *Rotor is Sensor*	12.8 *Rotor → Mechanism*
Remote CANcoder	n/a *Rotor is Unused*	1 *CANcoder on Mechanism*
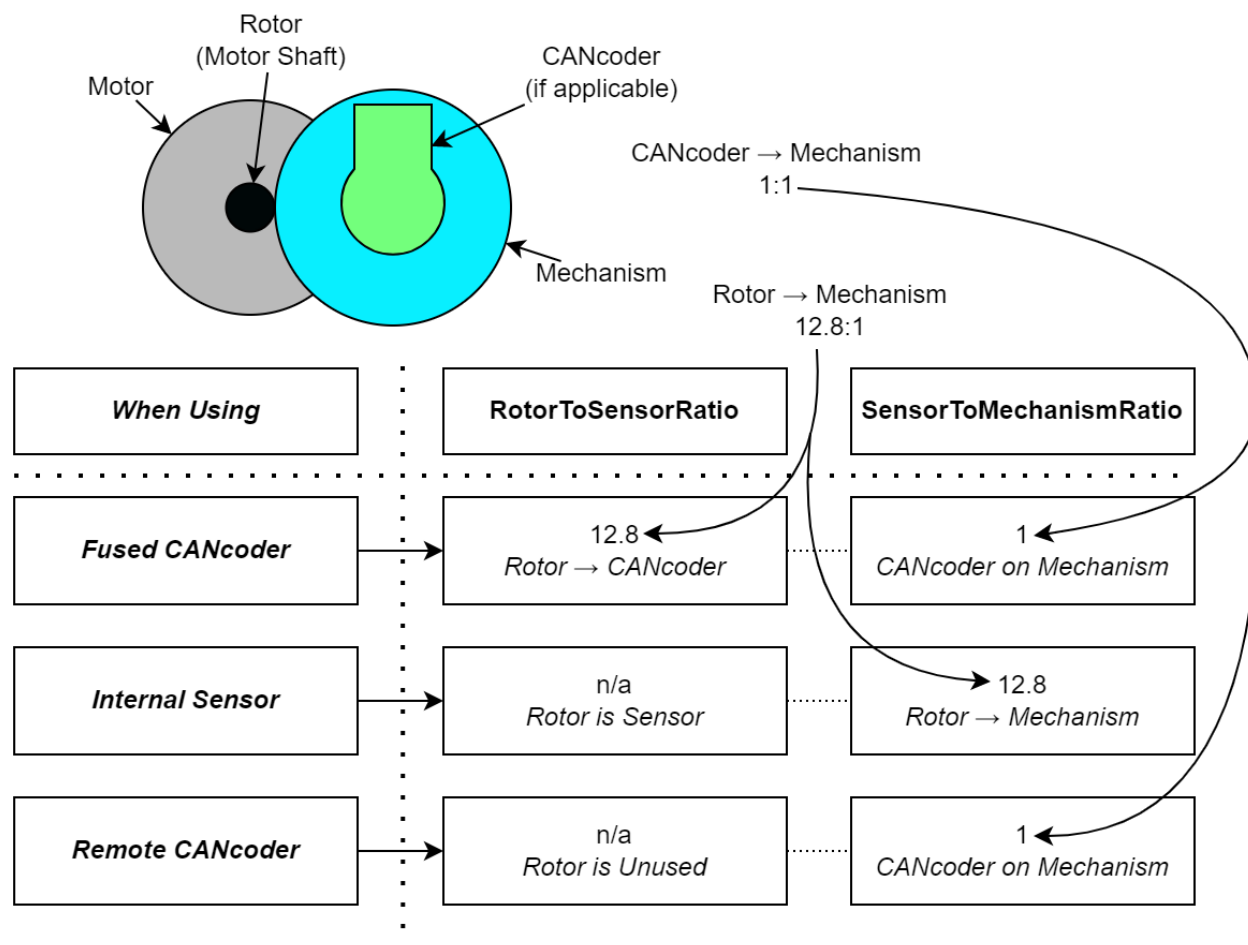
13.1.4 TalonFX Remote Sensors

The TalonFX supports various remote sensors. Remote sensors allow onboard closed-loop functionality at rates faster than a traditional robot processor (~1Khz) by reading the remote sensor directly from the CAN bus. This allows supported motor controllers to execute closed-loop modes with sensor values sourced by supported sensors.

A list of supported remote sensors can be found in the API docs (Java, C++).

Remote sensors can be configured using *Tuner X* or via code. This document highlights how to configure a remote sensor in a robot program.

RemoteCANcoder

A supported motor controller will update its position and velocity whenever the CANcoder publishes its information on the CAN bus.

Java

```
var fx_cfg = new TalonFXConfiguration();
fx_cfg.Feedback.FeedbackRemoteSensorID = m_cancoder.getDeviceID();
fx_cfg.Feedback.FeedbackSensorSource = FeedbackSensorSourceValue.RemoteCANcoder;

m_talonFX.getConfigurator().apply(fx_cfg);
```

C++

```
configs::TalonFXConfiguration fx_cfg{};
fx_cfg.Feedback.FeedbackRemoteSensorID = m_cancoder.GetDeviceID();
fx_cfg.Feedback.FeedbackSensorSource =
→signals::FeedbackSensorSourceValue::RemoteCANcoder;

m_talonFX.GetConfigurator().Apply(fx_cfg);
```

FusedCANcoder

Important: This feature requires the device to be *Pro licensed*.

New in Phoenix 6 is a feedback sensor type called FusedCANcoder. FusedCANcoder will fuse another CANcoder's information with the motor's internal rotor, which provides the best possible position and velocity for accuracy and bandwidth. This is useful in applications such as swerve azimuth.

FusedCANcoder requires the configuration of several Feedback config group items, shown below.

Full example: Java, C++

Java

```
51    /* Configure CANcoder to zero the magnet appropriately */
52    CANcoderConfiguration cc_cfg = new CANcoderConfiguration();
53    cc_cfg.MagnetSensor.AbsoluteSensorRange = AbsoluteSensorRangeValue.Signed_
   →PlusMinusHalf;
54    cc_cfg.MagnetSensor.SensorDirection = SensorDirectionValue.CounterClockwise_
   →Positive;
55    cc_cfg.MagnetSensor.MagnetOffset = 0.4;
56    m_cc.getConfigurator().apply(cc_cfg);
57
58    TalonFXConfiguration fx_cfg = new TalonFXConfiguration();
```

(continues on next page)

```
59        fx_cfg.Feedback.FeedbackRemoteSensorID = m_cc.getDeviceID();
60        fx_cfg.Feedback.FeedbackSensorSource = FeedbackSensorSourceValue.FusedCANcoder;
61        fx_cfg.Feedback.SensorToMechanismRatio = 1.0;
62        fx_cfg.Feedback.RotorToSensorRatio = 12.8;
63
64        m_fx.getConfigurator().apply(fx_cfg);
```

C++

```
11     /* Configure CANcoder to zero the magnet appropriately */
12     configs::CANcoderConfiguration cc_cfg{};
13     cc_cfg.MagnetSensor.AbsoluteSensorRange = signals::AbsoluteSensorRangeValue::Signed_
     ↪PlusMinusHalf;
14     cc_cfg.MagnetSensor.SensorDirection =␣
     ↪signals::SensorDirectionValue::CounterClockwise_Positive;
15     cc_cfg.MagnetSensor.MagnetOffset = 0.4;
16     m_cc.GetConfigurator().Apply(cc_cfg);
17
18     configs::TalonFXConfiguration fx_cfg{};
19     fx_cfg.Feedback.FeedbackRemoteSensorID = m_cc.GetDeviceID();
20     fx_cfg.Feedback.FeedbackSensorSource =␣
     ↪signals::FeedbackSensorSourceValue::FusedCANcoder;
21     fx_cfg.Feedback.SensorToMechanismRatio = 1.0;
22     fx_cfg.Feedback.RotorToSensorRatio = 12.8;
23
24     m_fx.GetConfigurator().Apply(fx_cfg);
```

Usage is the same as any *status signal*:

Java

```
fx_pos.refresh();
cc_pos.refresh();

System.out.println("FX Position: " + fx_pos.toString());
System.out.println("CANcoder Position: " + cc_pos.toString());
```

C++

```
fx_pos.Refresh();
cc_pos.Refresh();

std::cout << "FX Position: " << fx_pos << std::endl;
std::cout << "CANcoder Position: " << cc_pos << std::endl;
```

Simulation

Phoenix 6 supports comprehensive simulation support. All hardware features are available in simulation, including configs, control requests, simulated CAN bus timing, and Phoenix Tuner X support.

14.1 Introduction to Simulation

14.1.1 Supported Devices

Currently, all Phoenix 6 devices are supported in simulation.

> **Warning:** **Multiple** CAN buses using the *CANivore API* is not supported at this time. All CAN devices will **appear on the same CAN bus**. If you wish to run your robot code in simulation, ensure devices have **unique IDs across CAN buses**.

14.1.2 Simulation API

Each supported device has a device-specific `SimState` object that can be used to manage I/O with the simulated device. The object can be retrieved by calling `getSimState()` on an instance of a device.

Java

```java
var talonFXSim = m_talonFX.getSimState();
```

C++

```cpp
auto& talonFXSim = m_talonFX.GetSimState();
```

Note: Phoenix 6 utilizes the C++ units library when applicable.

Orientation

The `SimState` API ignores typical device invert settings, as the user may change invert for any reason (such as flipping which direction is forward for a drivebase). As a result, for some devices, the `SimState` object supports specifying the orientation of the device relative to the robot chassis (Java, C++).

This orientation represents the **mechanical** linkage between the device and the robot chassis. It **should not be changed with runtime invert**, as runtime invert specifies the *logical* orientation of the device. Rather, the orientation should **only be modified when the mechanical linkage itself changes**, such as when switching between two gearboxes inverted from each other.

Java

```java
var leftTalonFXSim = m_leftTalonFX.getSimState();
var rightTalonFXSim = m_rightTalonFX.getSimState();

// left drivetrain motors are typically CCW+
leftTalonFXSim.Orientation = ChassisReference.CounterClockwise_Positive;

// right drivetrain motors are typically CW+
rightTalonFXSim.Orientation = ChassisReference.Clockwise_Positive;
```

C++

```cpp
auto& leftTalonFXSim = m_leftTalonFX.GetSimState();
auto& rightTalonFXSim = m_rightTalonFX.GetSimState();

// left drivetrain motors are typically CCW+
leftTalonFXSim.Orientation = sim::ChassisReference::CounterClockwise_Positive;

// right drivetrain motors are typically CW+
rightTalonFXSim.Orientation = sim::ChassisReference::Clockwise_Positive;
```

Inputs and Outputs

All `SimState` objects contain multiple inputs to manipulate the state of the device based on simulation physics calculations. For example, all device `SimState` objects have a supply voltage input:

Important: Non-FRC platforms are required to set supply voltage, as it affects simulation calculations. It's recommended that FRC users set supply voltage to `getBatteryVoltage()` (Java, C++) to take advantage of WPILib's `BatterySim` (Java, C++) API.

Java

```java
// set the supply voltage of the TalonFX to 12 V
m_talonFXSim.setSupplyVoltage(12);
```

C++

```cpp
// set the supply voltage of the TalonFX to 12 V
m_talonFXSim.SetSupplyVoltage(12_V);
```

Some device `SimState` objects also contain outputs that can be used in simulation physics calculations. For example, the `TalonFXSimState` (Java, C++) object has a motor voltage output that can be used to calculate position and velocity:

Java

```java
// get the motor voltage of the TalonFX
var motorVoltage = m_talonFXSim.getMotorVoltage();

// use the motor voltage to calculate new position and velocity using an external␣
↪MotorSimModel class
m_motorSimModel.setMotorVoltage(motorVoltage);
m_motorSimModel.update(0.020); // assume 20 ms loop time

// apply the new rotor position and velocity to the TalonFX
m_talonFXSim.setRawRotorPosition(m_motorSimModel.getPosition());
m_talonFXSim.setRotorVelocity(m_motorSimModel.getVelocity());
```

C++

```cpp
// get the motor voltage of the TalonFX
auto motorVoltage = m_talonFXSim.GetMotorVoltage();

// use the motor voltage to calculate new position and velocity using an external␣
↪MotorSimModel class
m_motorSimModel.SetMotorVoltage(motorVoltage);
m_motorSimModel.Update(20_ms); // assume 20 ms loop time
```

```
// apply the new rotor position and velocity to the TalonFX
m_talonFXSim.SetRawRotorPosition(m_motorSimModel.GetPosition());
m_talonFXSim.SetRotorVelocity(m_motorSimModel.GetVelocity());
```

14.1.3 High Fidelity CAN Bus Simulation

Many popular CTR Electronics CAN devices support high-fidelity simulation, where the influence of the CAN bus is simulated at a level similar to what happens on a real robot. This means that the timing behavior of control and status signals in simulation will align to the same framing intervals seen on a real CAN bus. In simulation, this may appear as a delay between setting a signal and getting its real value, or between setting its real value and getting it in API.

The update rate can be modified for simulation by wrapping the *signal's frequency* in a RobotBase.isSimulation() (Java, C++) condition.

Java

```
if (RobotBase.isSimulation()) {
    m_velocitySignal.setUpdateFrequency(1000); // set update rate to 1ms
}
```

C++

```
if (RobotBase::IsSimulation()) {
    m_velocitySignal.SetUpdateFrequency(1000_Hz); // set update rate to 1ms
}
```

WPILib Integration

Phoenix 6 API used as part of WPILib robot projects provides implementations of common WPILib interfaces that FRC teams use.

15.1 MotorController Integration

Phoenix 6 motor controller classes such as `TalonFX` (Java, C++) implement the `MotorController` (Java, C++) interface. This allows Phoenix 6 motor controllers to be used in WPILib drivetrain classes such as `DifferentialDrive`.

Java

```java
// instantiate motor controllers
TalonFX m_motorLeft = new TalonFX(0);
TalonFX m_motorRight = new TalonFX(1);

// create differentialdrive object for robot control
DifferentialDrive m_diffDrive = new DifferentialDrive(m_motorLeft, m_motorRight);

// instantiate joystick
XboxController m_driverJoy = new XboxController(0);

public void teleopPeriodic() {
   var forward = -m_driverJoy.getLeftY();
   var rot = -m_driverJoy.getRightX();

   m_diffDrive.arcadeDrive(forward, rot);
}
```

C++ (Source)

```cpp
void Robot::TeleopPeriodic() {
    auto forward = -m_driverJoy.GetLeftY();
    auto rot = -m_driverJoy.GetRightX();

    m_diffDrive.ArcadeDrive(forward, rot);
}
```

C++ (Header)

```cpp
// instantiate motor controllers
hardware::TalonFX m_motorLeft{0};
hardware::TalonFX m_motorRight{1};

// create differentialdrive object for robot control
frc::DifferentialDrive m_diffDrive{m_motorLeft, m_motorRight};

// instantiate joystick
frc::XboxController m_driverJoy{0};
```

15.1.1 Motor Safety

CTR Electronics supported actuators implement WPILib Motor Safety. In additional to the normal *enable signal* of CTR Electronics actuators, Motor Safety will automatically disable the device according to the WPILib Motor Safety implementation.

15.1.2 Simulation

It's recommended that users set supply voltage to getBatteryVoltage() (Java, C++) to take advantage of WPILib's BatterySim (Java, C++) API. Additionally, the simulated device state is shown in the simulation *Other Devices* menu.

15.2 Gyro Integration

CTR Electronics IMUs, such as the Pigeon 2.0, implement the WPILib `Gyro` (Java, C++) interface.

Note: `calibrate()` does nothing on the Pigeon 2.0, as it does not require manual calibration.

15.2.1 Simulation

The simulated device state is shown in the simulation *Other Devices* menu.

```
▼ Other Devices                    ✕

  ▼     Talon FX (Pro)[0]

  11.995547              supplyVoltage

  0.000000               dutyCycle

  0.000000               motorVoltage

  0.000000               torqueCurrent

  0.000000               supplyCurrent

  ▼ Talon FX (Pro)[0]/Rotor Sensor

  0.000000               position

  0.000000               rawPositionInput

  0.000000               velocity

  ▼  Talon FX (Pro)[0]/Fwd Limit

  true                   init

  true                   input

  false          ▼       value

  ▶  Talon FX (Pro)[0]/Rev Limit

  ▶     Talon FX (Pro)[1]

  ▶  Talon FX (Pro)[1]/Rotor Sensor

  ▶  Talon FX (Pro)[1]/Fwd Limit

  ▶  Talon FX (Pro)[1]/Rev Limit

  ▶  CANcoder (Pro)[0]

  ▶  CANcoder (Pro)[1]

  ▶  Pigeon 2 (Pro)[0]
```

Examples

Comprehensive API usage examples and tutorials.

16.1 Open-Loop Quickstart

The below example showcases controlling a four-motor drivetrain.

16.1.1 Declaring Motor Controllers

The `TalonFX` motor controller constructor (Java, C++) requires a **device ID** (int) and an optional **CAN bus** (string).

Note: The name of the native roboRIO CAN bus is `rio`. This is also the default CAN bus on the roboRIO when none is specified.

Java

```java
public class Robot extends TimedRobot {
    private static final String kCANBus = "canivore";

    private final TalonFX m_leftLeader = new TalonFX(0, kCANBus);
    private final TalonFX m_rightLeader = new TalonFX(1, kCANBus);
    private final TalonFX m_leftFollower = new TalonFX(2, kCANBus);
    private final TalonFX m_rightFollower = new TalonFX(3, kCANBus);
}
```

```cpp
class Robot : public frc::TimedRobot {
private:
    static constexpr char const *kCANBus{"canivore"};

    ctre::phoenix6::hardware::TalonFX m_leftLeader{0, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_rightLeader{1, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_leftFollower{2, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_rightFollower{3, kCANBus};
}
```

16.1.2 Configure Followers & Inverts

In a traditional robot drivetrain, there are two motors attached to each horizontal side of the drivetrain. This setup *typically* (unless mechanically inverted) causes the right side to rotate in an opposite direction when given the same voltage.

Java

```java
@Override
public void robotInit() {
    // start with factory-default configs
    var currentConfigs = new MotorOutputConfigs();

    // The left motor is CCW+
    currentConfigs.Inverted = InvertedValue.CounterClockwise_Positive;
    m_leftLeader.getConfigurator().apply(currentConfigs);

    // The right motor is CW+
    currentConfigs.Inverted = InvertedValue.Clockwise_Positive;
    m_rightLeader.getConfigurator().apply(currentConfigs);

    // Ensure our followers are following their respective leader
    m_leftFollower.setControl(new Follower(m_leftLeader.getDeviceID(), false));
    m_rightFollower.setControl(new Follower(m_rightLeader.getDeviceID(), false));
}
```

C++ (Source)

```cpp
#include "Robot.h"

using namespace ctre::phoenix6;

void Robot::RobotInit() {
    // start with factory-default configs
    configs::MotorOutputConfigs currentConfigs{};

    // The left motor is CCW+
    currentConfigs.Inverted = signals::InvertedValue::CounterClockwise_Positive;
    m_leftLeader.GetConfigurator().Apply(currentConfigs);

    // The right motor is CW+
    currentConfigs.Inverted = signals::InvertedValue::Clockwise_Positive;
    m_rightLeader.GetConfigurator().Apply(currentConfigs);

    // Ensure the followers are following their respective leader
    m_leftFollower.SetControl(controls::Follower{m_leftLeader.GetDeviceID(), false});
    m_rightFollower.SetControl(controls::Follower{m_rightLeader.GetDeviceID(), false});
}
```

16.1.3 Full Example

Java

```java
public class Robot extends TimedRobot {
    private static final String kCANBus = "canivore";

    private final TalonFX m_leftLeader = new TalonFX(0, kCANBus);
    private final TalonFX m_rightLeader = new TalonFX(1, kCANBus);
```
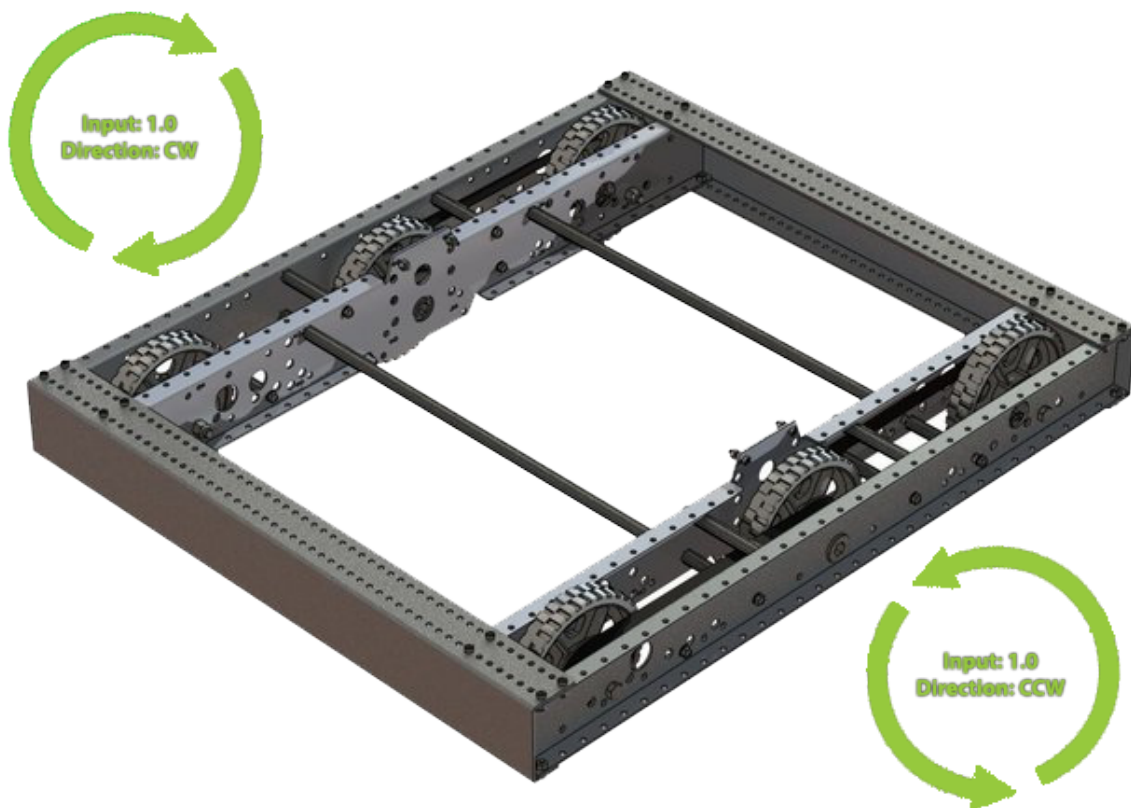
```java
    private final TalonFX m_leftFollower = new TalonFX(2, kCANBus);
    private final TalonFX m_rightFollower = new TalonFX(3, kCANBus);

    private final DutyCycleOut m_leftOut = new DutyCycleOut(0);
    private final DutyCycleOut m_rightOut = new DutyCycleOut(0);

    private final XboxController m_driverJoy = new XboxController(0);

    @Override
    public void robotInit() {
        // start with factory-default configs
        var currentConfigs = new MotorOutputConfigs();

        // The left motor is CCW+
        currentConfigs.Inverted = InvertedValue.CounterClockwise_Positive;
        m_leftLeader.getConfigurator().apply(currentConfigs);

        // The right motor is CW+
        currentConfigs.Inverted = InvertedValue.Clockwise_Positive;
        m_rightLeader.getConfigurator().apply(currentConfigs);

        // Ensure our followers are following their respective leader
        m_leftFollower.setControl(new Follower(m_leftLeader.getDeviceID(), false));
        m_rightFollower.setControl(new Follower(m_rightLeader.getDeviceID(), false));
    }

    @Override
    public void teleopPeriodic() {
        // retrieve joystick inputs
        var fwd = -m_driverJoy.getLeftY();
        var rot = m_driverJoy.getRightX();

        // modify control requests
        m_leftOut.Output = fwd + rot;
        m_rightOut.Output = fwd - rot;

        // send control requests
        m_leftLeader.setControl(m_leftOut);
        m_rightLeader.setControl(m_rightOut);
    }
}
```

C++ (Source)

```cpp
#include "Robot.h"

using namespace ctre::phoenix6;

void Robot::RobotInit() {
    // start with factory-default configs
    configs::MotorOutputConfigs currentConfigs{};

    // The left motor is CCW+
    currentConfigs.Inverted = signals::InvertedValue::CounterClockwise_Positive;
```

```cpp
    m_leftLeader.GetConfigurator().Apply(currentConfigs);

    // The right motor is CW+
    currentConfigs.Inverted = signals::InvertedValue::Clockwise_Positive;
    m_rightLeader.GetConfigurator().Apply(currentConfigs);

    // Ensure the followers are following their respective leader
    m_leftFollower.SetControl(controls::Follower{m_leftLeader.GetDeviceID(), false});
    m_rightFollower.SetControl(controls::Follower{m_rightLeader.GetDeviceID(), false});
}

void Robot::TeleopPeriodic() {
    // retrieve joystick inputs
    auto fwd = -m_driverJoy.GetLeftY();
    auto rot = m_driverJoy.GetRightX();

    // modify control requests
    m_leftOut.Output = fwd + rot;
    m_rightOut.Output = fwd - rot;

    // send control requests
    m_leftLeader.SetControl(m_leftOut);
    m_rightLeader.SetControl(m_rightOut);
}
```

C++ (Header)

```cpp
private:
    static constexpr char const *kCANBus{"canivore"};

    ctre::phoenix6::hardware::TalonFX m_leftLeader{0, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_rightLeader{1, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_leftFollower{2, kCANBus};
    ctre::phoenix6::hardware::TalonFX m_rightFollower{3, kCANBus};

    ctre::phoenix6::controls::DutyCycleOut m_leftOut{0};
    ctre::phoenix6::controls::DutyCycleOut m_rightOut{0};

    frc::XboxController m_driverJoy{0};
```

CANivore Intro

The CANivore is a multipurpose USB-to-CAN FD device. The CANivore:

- Adds a secondary CAN FD bus to the roboRIO
  - CAN FD improves upon CAN with increased device bandwidth and transfer speed.
- Allows the control of CTR Electronics devices on *non-roboRIO platforms*.

Important: Details on licensing your CANivore is available on the *licensing* page.

Initial Setup   Setting up a CANivore for robot projects and desktop development.

API Usage   Using the CANivore with devices in API.

Hardware-Attached Simulation   Using a CANivore with hardware devices in a desktop environment.

Advanced Configuration   Advanced configuration options for the CANivore.

CANivore Setup

18.1 Supported Systems

Currently, the following systems are supported for CANivore development:

- NI roboRIO
- Windows 10/11 x86-64
- Linux x86-64 (desktop)
    - Ubuntu 22.04 or newer
    - Debian Bullseye or newer
- Linux ARM32 and ARM64 (Raspberry Pi, NVIDIA Jetson)
    - Ubuntu 20.04 or newer
    - Debian Bullseye or newer

Note: Custom bit rates and CAN 2.0 are not supported at this time. The parameters passed into SocketCAN are not applied by the firmware.

18.1.1 roboRIO

Note: Phoenix Tuner X requires a 2023 roboRIO image or newer to configure the CANivore.

No additional steps are required. The roboRIO comes with the `canivore-usb` kernel module pre-installed.

18.1.2 Linux (non-FRC)

On non-FRC Linux systems, the `canivore-usb` kernel module must be installed to add Socket-CAN support for the CANivore. The kernel module is distributed through our APT repository. Begin with adding the repository to your APT sources.

```
YEAR=<year>
sudo curl -s --compressed -o /usr/share/keyrings/ctr-pubkey.gpg "https://deb.ctr-
↪electronics.com/ctr-pubkey.gpg"
sudo curl -s --compressed -o /etc/apt/sources.list.d/ctr${YEAR}.list "https://deb.ctr-
↪electronics.com/ctr${YEAR}.list"
```

Note: <year> should be replaced with the year of Phoenix 6 software for which you have purchased licenses.

After adding the sources, the kernel module can be installed and updated using the following:

Important: Users on a Raspberry Pi OS based platform must install the kernel headers before running the below install script. Headers can be installed by running `sudo apt install raspberrypi-kernel-headers`.

```
sudo apt update
sudo apt install canivore-usb
```

Tip: To get a robot application up and running quickly, check out our non-FRC Linux example.

Raspberry Pi 4 Errata

On a Raspberry Pi 4 or newer, the latest 32-bit Raspberry Pi OS image will default to using the 64-bit kernel while still using 32-bit APT packages. As a result, our canivore-usb kernel module will fail to install.

There are two options to work around this issue:

1. (Recommended) Use the 64-bit Raspberry Pi OS. This allows programs to use all available RAM and improves overall system performance and stability.

2. Add `arm_64bit=0` to /boot/config.txt and reboot. This forces the Raspberry Pi to use the 32-bit kernel. Note that programs will be limited to using 3 GB of RAM, and system performance may be impacted.

Warning: Do not add `arm_64bit=0` to /boot/config.txt when using the 64-bit Raspberry Pi OS. Attempting to do so may cause the Pi to be unable to boot.

18.2 Viewing Attached CANivores

Attached CANivores can be viewed in Phoenix Tuner X by selecting the *CANivores* page from the left-hand sidebar. You can specify the target system in the *Target IP or Team #* text box.
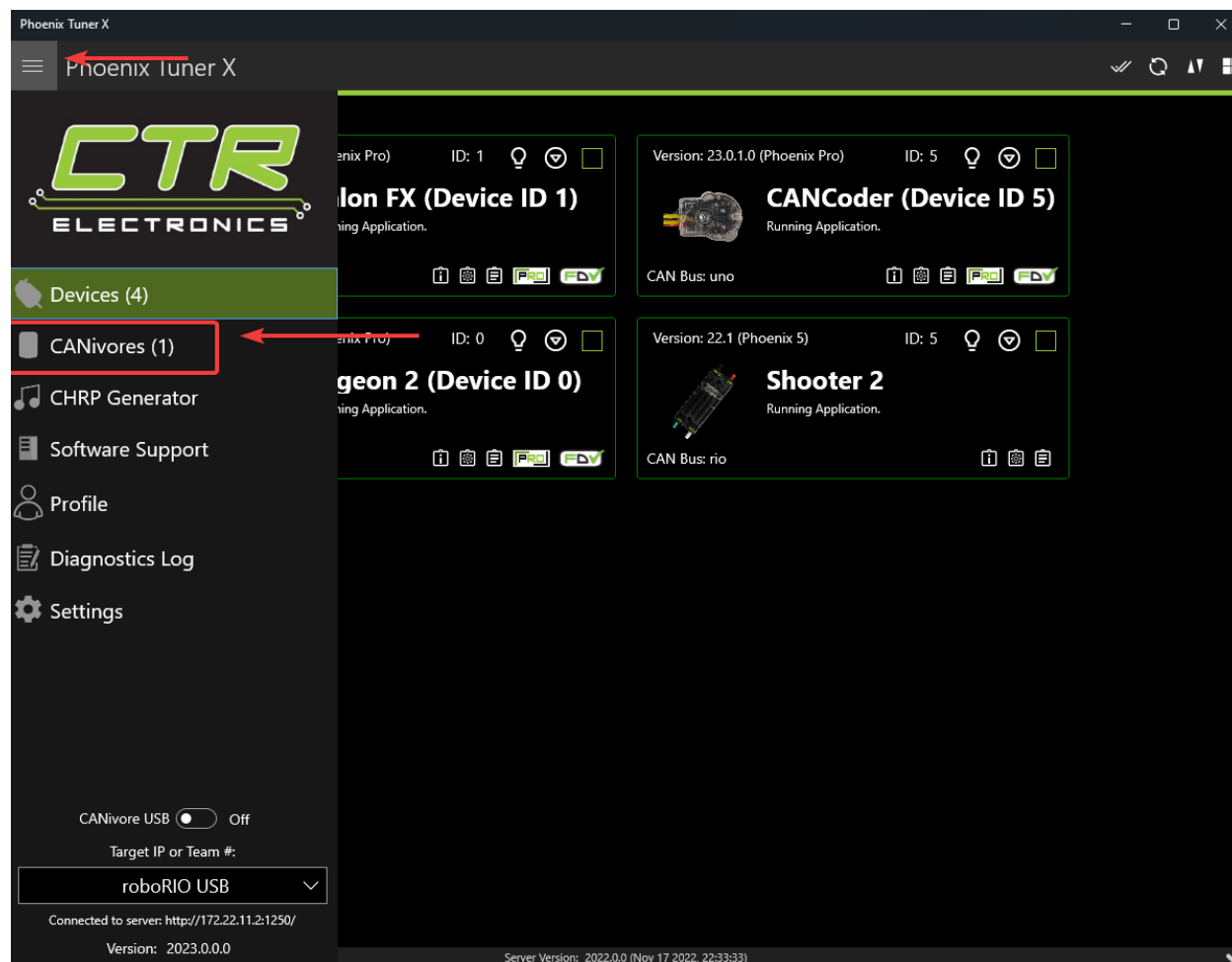

Note: The Phoenix Diagnostic Server must be running on the target system to use the CANivores page.

Tip: If you are connecting to CANivores on your local Windows machine, you can enable the CANivore USB toggle and set the target IP to `localhost`. This runs a diagnostic server within Tuner X so you do not need to run a robot project to communicate with CANivores.
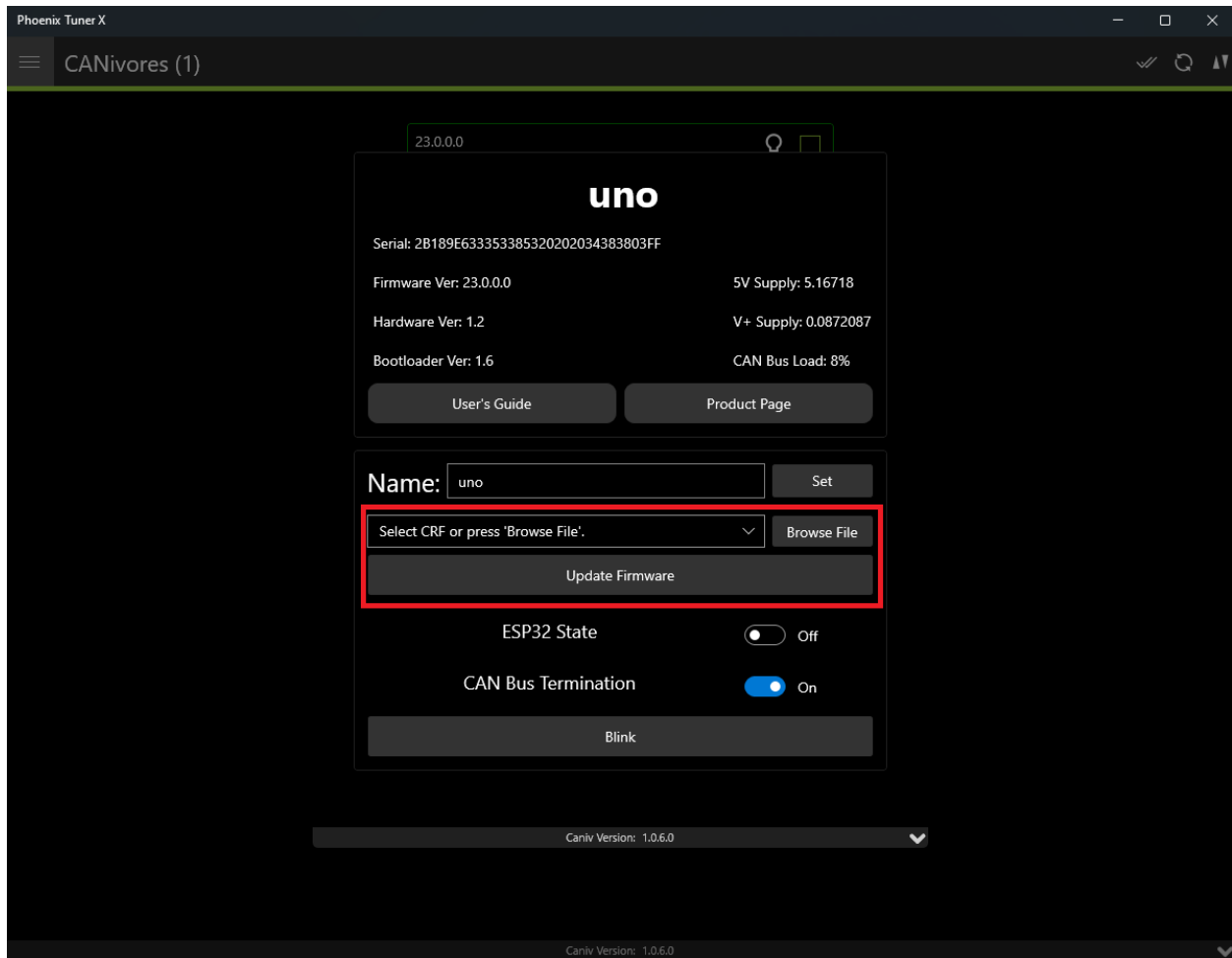
18.3 Field Upgrading CANivores

A CANivore can be field updated using *Phoenix Tuner X*.

Click or tap on the listed CANivore card:
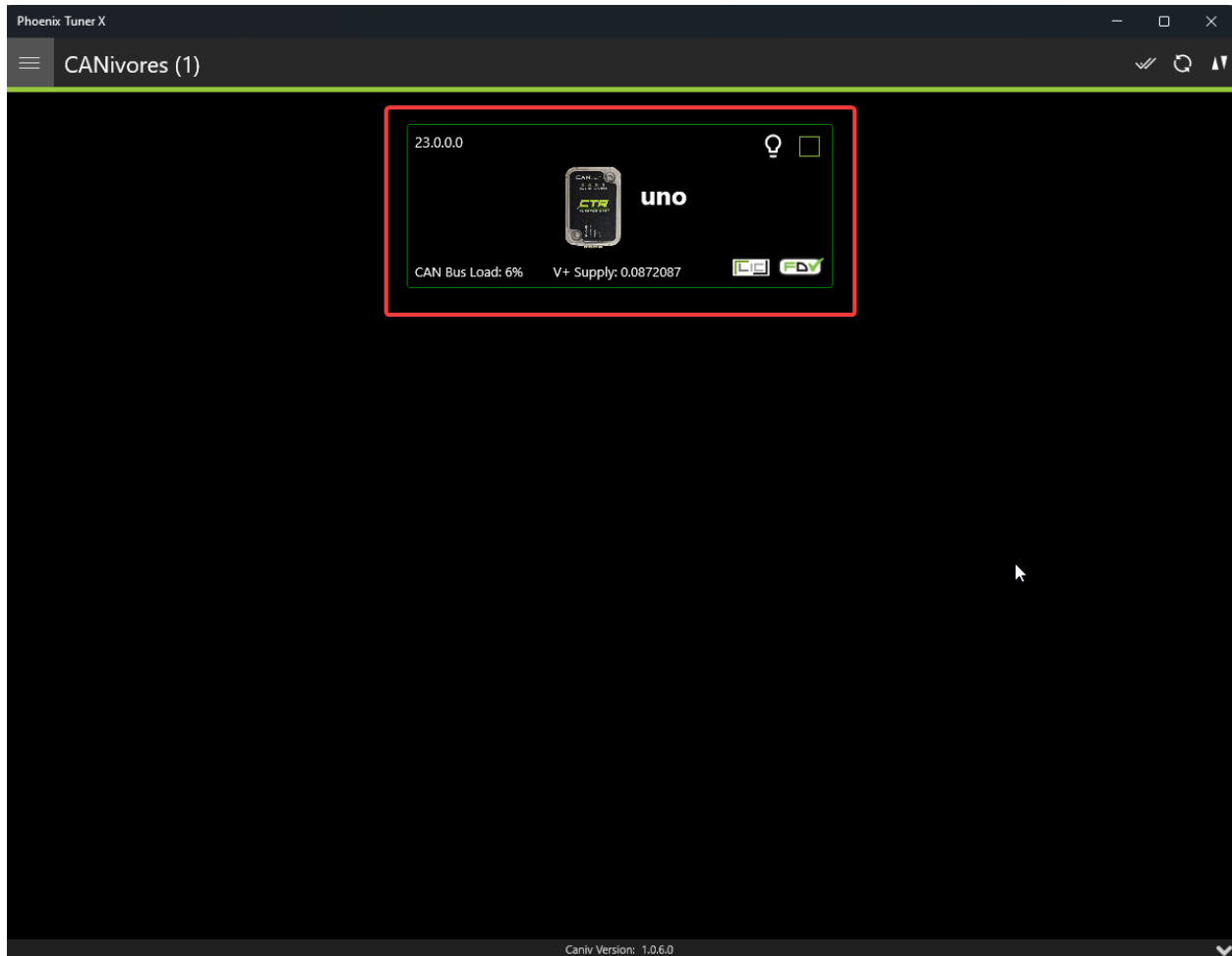
The CANivore can then be field upgraded via the dropdown or by manually selected a file:

Phoenix Tuner X also allows the user to batch field upgrade CANivores from the list of CANivores in the same manner as *batch field upgrading devices*.

18.4 Renaming CANivores

CANivores can be given custom names for use within a robot program. This can be configured through Phoenix Tuner X on the specified device card.

CANivore API

All device constructors have an overload that takes a string CAN bus identifier. This identifier can be `rio` for the native roboRIO CAN bus, * to select the first available CANivore, or a CANivore's name or serial number. On non-FRC Linux systems, this string can also be a SocketCAN interface.

Note: If there are multiple CANivores with the same name, the system will use the first CANivore found.

If no CAN bus string is passed into the constructor, or the CAN bus string is empty, the behavior is platform-dependent:

- roboRIO: use the roboRIO native CAN bus
- Windows: use the first CANivore found
- non-FRC Linux: use SocketCAN interface `can0`

Java

```
TalonFX fx_default = new TalonFX(0); // On roboRIO, this constructs a TalonFX on the␣
↪RIO native CAN bus
TalonFX fx_rio = new TalonFX(1, "rio"); // This also constructs a TalonFX on the RIO␣
↪native CAN bus
TalonFX fx_drivebase = new TalonFX(0, "Drivebase"); // This constructs a TalonFX on␣
↪the CANivore bus named "Drivebase"
CANcoder cc_elevator = new CANcoder(0, "Elevator"); // This constructs a CANcoder on␣
↪the CANivore bus named "Elevator"
```

C++ (Header)

```
hardware::TalonFX fx_default{0}; // On roboRIO, this constructs a TalonFX on the RIO␣
↪native CAN bus
hardware::TalonFX fx_rio{1, "rio"}; // This also constructs a TalonFX on the RIO␣
↪native CAN bus
hardware::TalonFX fx_drivebase{0, "Drivebase"}; // This constructs a TalonFX on the␣
↪CANivore bus named "Drivebase"
hardware::CANcoder cc_elevator{0, "Elevator"}; // This constructs a CANcoder on the␣
↪CANivore bus named "Elevator"
```

19.1 CANivore Status Prints

When working with CANivore CAN buses in a robot program, Phoenix prints some messages to report the state of the CANivore connection. These messages can be useful to debug connection issues (bad USB vs bad CAN) or report bugs to CTR Electronics.

Table 1: Connection Messages

Message	Connection Status
CANbus Failed to Connect	Could not connect to a CANivore with the given name or serial number
CANbus Connected	Successfully found and connected to the CANivore with the given name or serial number
CANbus Disconnected	Detected that a CANivore USB device has been disconnected

Table 2: CANivore Bring-up Messages (Linux only)

| Message	Bring-up Status
CANbus Failed Bring-up	Found and connected to the CANivore, but could not configure the device or start the network
CANbus Successfully Started	Successfully configured the CANivore and started the network

Table 3: Network State Messages

Message	Network State
CANbus Network Down	Linux: The SocketCAN network has been deactivated, USB-to-CAN activity has stopped Windows: Could not open the communication channels for USB-to-CAN traffic
CANbus Network Up	Linux: The SocketCAN network has been activated, USB-to-CAN activity has resumed Windows: Successfully opened the communication channels for USB-to-CAN traffic

Hardware-Attached Simulation

CANivore supports hardware-attached simulation when used in an FRC robot program. This allows a CANivore to be used with real devices on supported host operating systems. The below video showcases controlling a **real** Falcon 500 in a robot program using hardware-attached simulation.
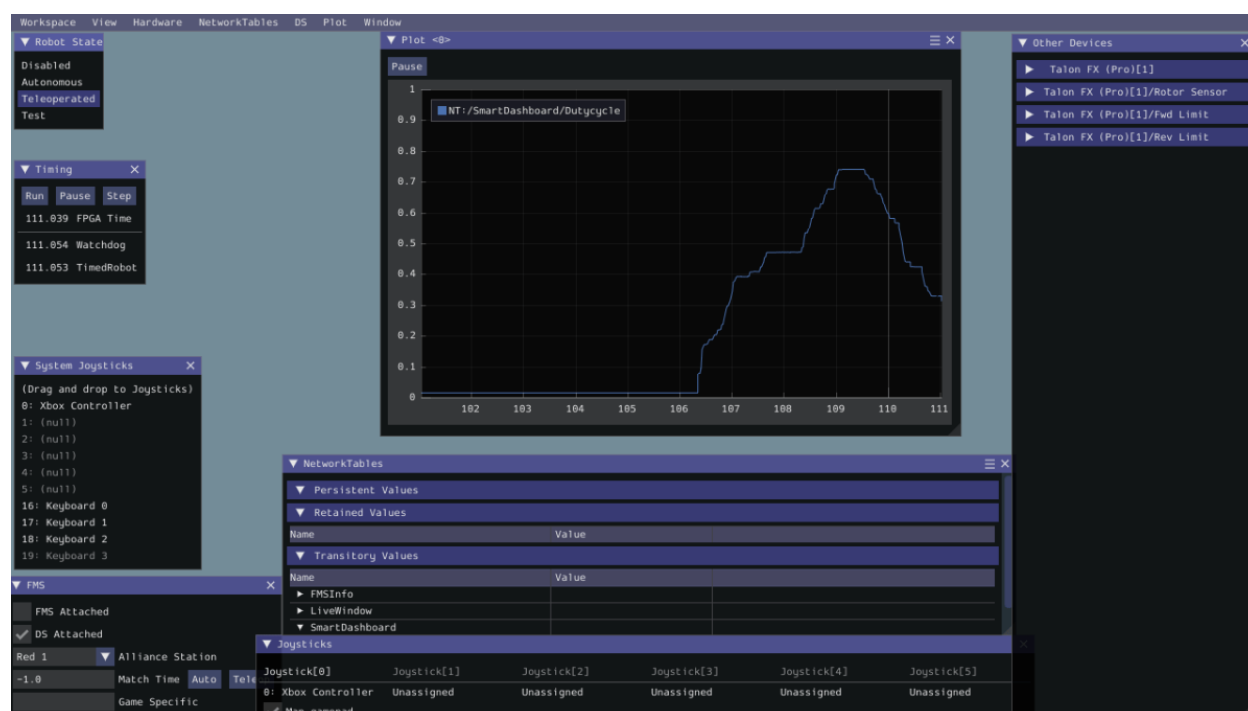
To utilize hardware-attached simulation, ensure the CANivore is connected directly via USB to the machine running the simulation. All devices on the CANivore CAN Bus should be independently powered, as the CANivore does not provide power. In the robot program, the CANivore name or * must be specified in the device constructor.

Important: Any motors/actuators that have been connected to a roboRIO CAN Bus at any time must be factory defaulted due to them being *FRC Locked*. Factory defaulting can be done in *Tuner X* and should be done when the CANivore is **not** connected to a roboRIO.

Java

```
TalonFX m_motor = new TalonFX(0, "mycanivore");
```

C++

```
hardware::TalonFX m_motor{0, "mycanivore"};
```

In VS Code, select the 3 dots in the top-right, then select *Hardware Sim Robot Code*
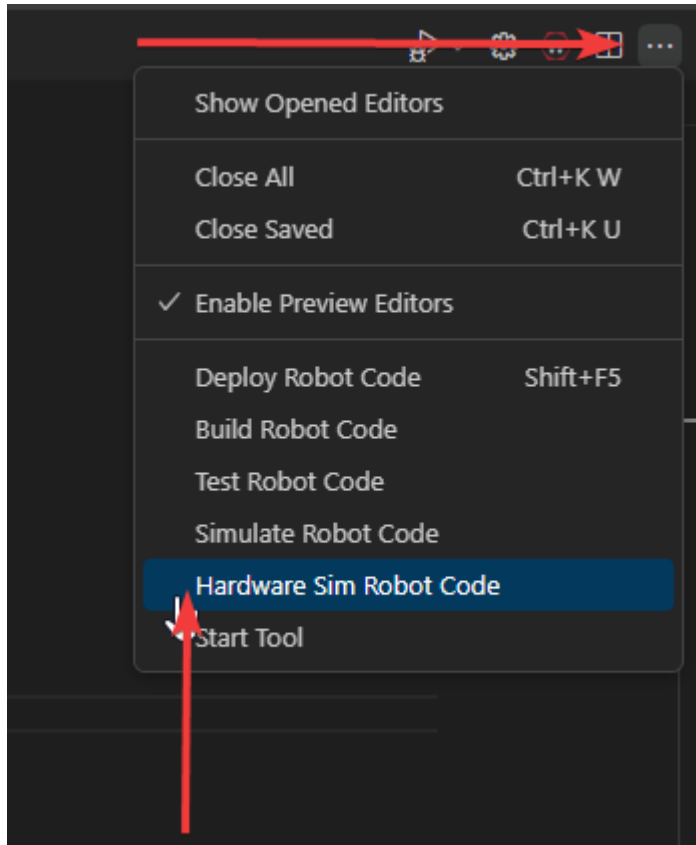
A message in the console should appear that the CAN Bus is connected.

```
********** Robot program startup complete **********
[phoenix] CANbus Connected: uno (WinUSB, 2B189E63335333853202020034383803FF)
[phoenix] CANbus Network Up: uno (WinUSB, 2B189E63335333853202020034383803FF)
[phoenix] Library initialization is complete.
```

_21_

Advanced Configuration

The CANivore provides additional configuration options for advanced users.

21.1 CAN Bus Termination

The CANivore has a 120 Ω programmable resister for terminating the CAN bus. The resistor can be configured using the CAN Bus Termination toggle in the CANivore device card in Phoenix Tuner X.

> **Warning:** A CAN bus requires two termination resistors, one at each extreme end. If only one is present, communication over CAN may fail.

21.2 caniv - CANivore CLI

caniv is a Command-line Interface (CLI) to interact with CANivores outside of Phoenix Tuner X.

Note: Unlike the CANivores page in Phoenix Tuner X, caniv does **not** require a running Phoenix Diagnostic Server.

On Linux systems (including the roboRIO), caniv can be found at /usr/local/bin. On Windows systems, the program is in the Phoenix Tuner X application cache directory, which can be opened by opening the *Diagnostic Log* page and clicking the left folder icon in the top right:
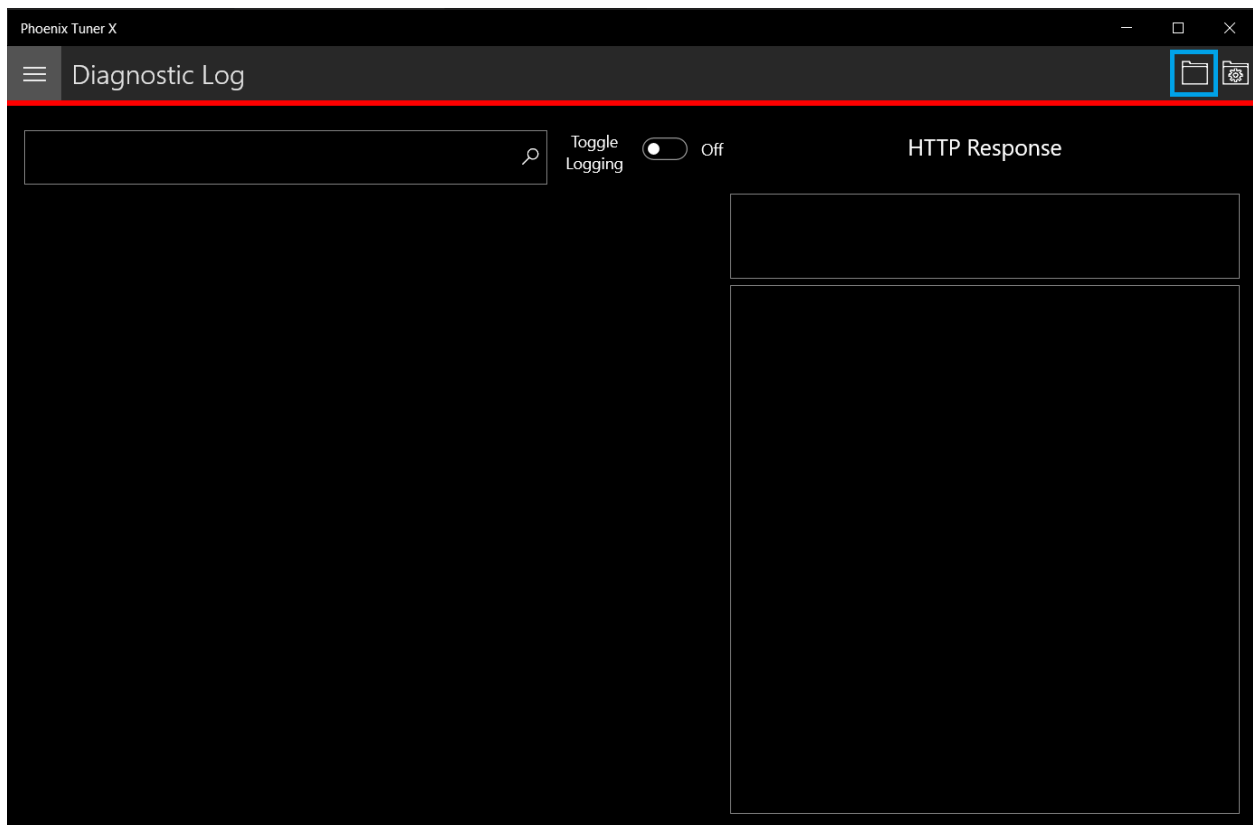
To view a list of available commands, run caniv either with no parameters or with --help.

```
admin@roboRIO-3539-FRC:~# caniv
CTRE CANivore Command-line Interface
Usage:
  caniv [OPTION...]

  -d, --device arg  Selects CANivore by serial number or name. Specify "*" to
                    select any device.
  -a, --any         Equivalent to [-d "*"]
  -v, --verbose     Print verbose messages
  -h, --help        Print this help message

 Action options:
      --version         Get version of this program
  -l, --list            List all discovered CANivore devices
  -s, --setup           Brings up the CANivore network
      --termination arg Enables or Disables the CAN bus termination resistor
      --esp32 arg       Enables or Disables the ESP32 module
  -b, --blink           Blinks selected CANivore
  -r, --reset           Resets selected CANivore
  -n, --name arg        Sets name of selected CANivore. You must reset the
                        CANivore for the new name to take effect.
  -f, --flash arg       Field upgrades selected CANivore with crf file
  -i, --info            Solicit information from selected CANivore

admin@roboRIO-3539-FRC:~# 
```

Development Blog

Welcome to the development blog. Here, we will highlight various features of CTR Electronics devices and how they can be utilized in specific applications.

Note: This list may move in the future.

- *Latency and Frequency*
- *Tuner and an evolution in configuration*

22.1 Latency and Frequency

Authored by Cory

A common discussion with CAN-based sensors and actuators is how latency and update frequency affect robot performance. This devblog is meant to expand what these aspects are and how users can mitigate or eliminate them with Phoenix 6/Pro.

22.1.1 Frequency

Signal frequency largely affects two aspects for robots:

1. Closed loop control
2. Odometry

Closed loop control is pretty clear, the slower the frequency, the greater the time gaps and the more you have to dampen a closed loop controller to keep it smooth. This becomes more important the less inertia your system has, such as with a swerve drive azimuth. At the inertia of a full robot, this becomes less important, as the robot can't change its own heading fast enough for the lower frequency to have much of an impact.

Odometry is a little less clear, but the gist is that odometry itself is generally an integration problem. If your data is coming in at a lower frequency, the integration isn't as accurate, and so you accrue drift and error. This gets worse over time, which is why absolute odometry solutions such as april tag pose are so important at the high level to correct for it.

22.1.2 Latency

Latency itself isn't so much an issue, but nondeterministic latency can be. If you aren't getting data at a steady rate, it can negatively affect the odometry, as the time aspect of the integration problem is no longer constant. You can correct for this with latency compensation if you have the option to, but even that's not a perfect solution as the latency corrected value may not be exactly correct to the real value at that point in time (although it's certainly better than nothing).

22.1.3 Solutions

We (CTR Electronics) recognized these problems in general and wanted to solve them, which is why each of these problems are solved in the new Phoenix 6/Pro library and with CANivore.

- *Fused CANcoder* (**Pro**): By fusing in the CANcoder's position into the Talon FX's internal position, users ensure the position is always absolute while maintaining 1000hz update frequency and 0 latency for closed loop operations. This feature is improved when used with CANivore, as the CANcoder's position can be latency-compensated and fused even when the mechanism is moving.

- *Time Synchronization* (**Pro & CANivore**): The CANivore provides the ability for Pro devices to synchronize to a common clock, which allows all the devices to sample data at the same time and publish data at the same time. This makes latency of data between devices minimal, and when used with our synchronize API keeps total latency low.

- *Synchronous API* (**v6**): The synchronous API allows users to wait for data to arrive. By waiting for all the key data to arrive, the overall latency is reduced and users can update their robot's data as soon as new data is available.

- *Time Stamps* (**v6**): Every data is timestamped so users can perform latency compensation. The timestamp information is improved if used with a CANivore, as the CANivore timestamps the data once it arrives over the wire, providing a more accurate timestamp compared to when the RIO does the timestamping.

- *Improved bus utilization* (**v6**): The improved bus utilization with v6 and further improvement with CANivore allow you to pretty easily increase the frequency of your key signals. We think it's pretty achievable to get 200hz update frequencies (5ms periods) under a full robot.

Every component outlined here is used in our SwerveDriveExample. If you're interested in how to do it for yourself, I'd recommend looking at it and doing something similar.

22.2 Tuner and an evolution in configuration

Authored by Dalton
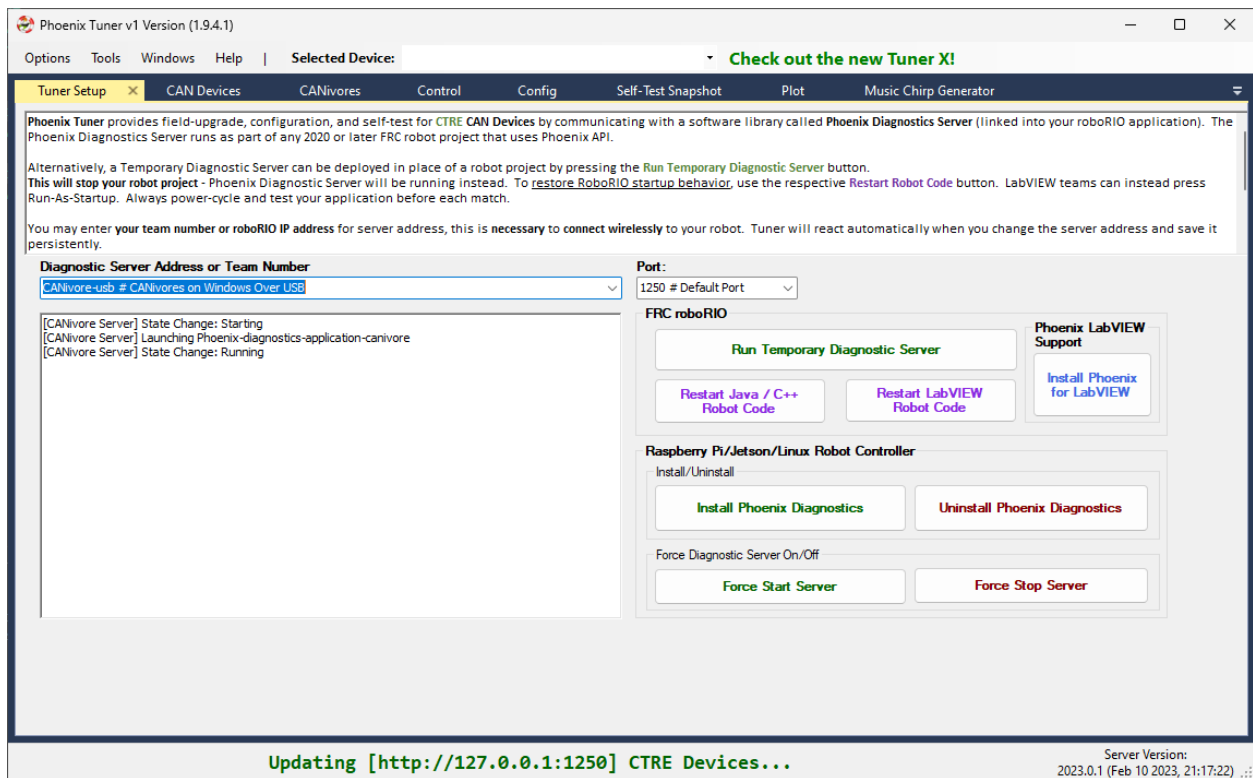
Since the introduction of the CTRE Toolsuite (pre-2018), we at CTR have strived to provide intuitive means of configuring and utilizing our products. In 2018, we launched Phoenix Tuner (now lovingly referred to as Tuner v1). Tuner v1 introduced features like: batch firmware upgrading like devices, diagnostic server deployment, self tests, plotting and control.

With Tuner, and by extension diagnostics, we have several primary objectives:

- Ease of debugging (exposed via self test).
- Seamless setup experience.
- Support and integrate our extension feature-set.

Tuner v1 was and is great, but we wanted to do more. In the 2023 season, we introduced Tuner X.

22.2.1 Introducing Tuner X
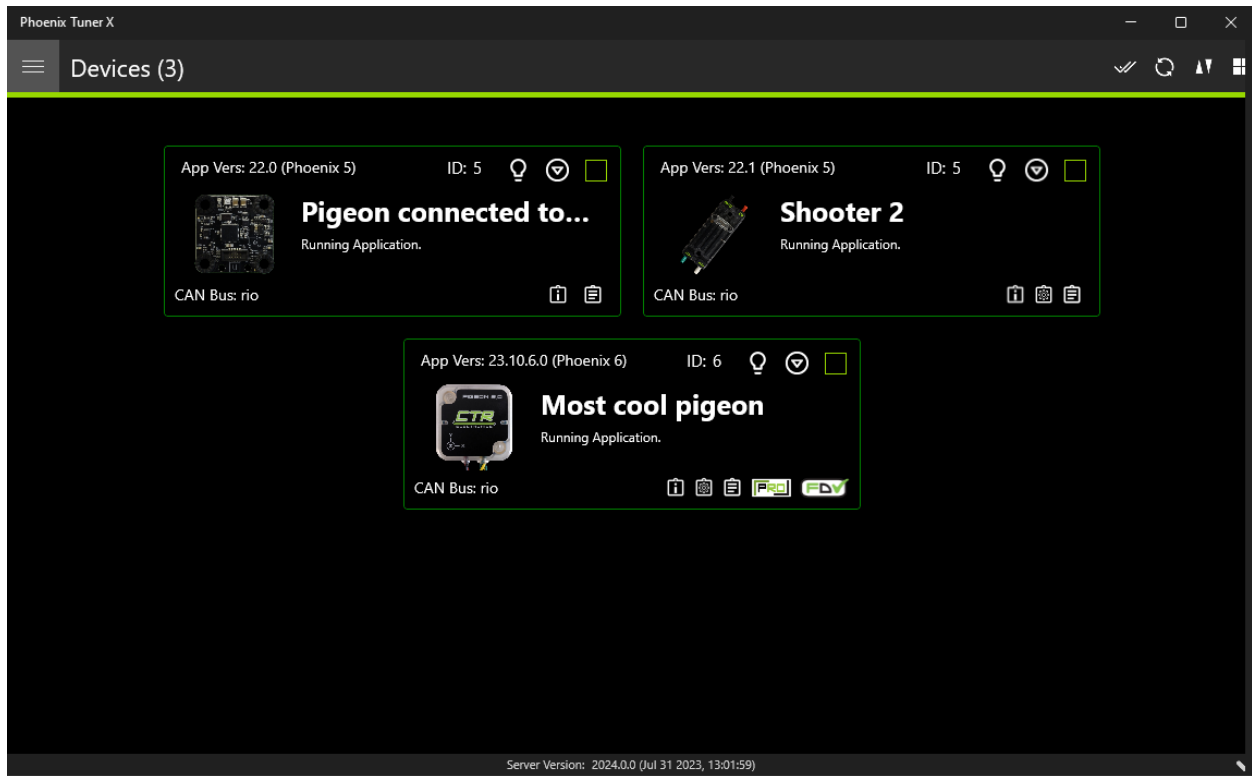
The goal we had with Tuner X development was to refine and enhance the existing Tuner v1 feature set. We introduced Android support, improved batch upgrading, improved highlighting of duplicate devices, automatic firmware downloads (no more downloading CRFs!), improved self test and licensing support.

With Tuner X, users can:

- Configure their device's name & ID
- Blink a device, which is useful for identifying where the device is on the robot.
- Firmware update all devices to the latest version available (no more CRF downloads).
- Control individual motors with their Android phone, or on Windows.
- Plot various signals such as velocity, position and yaw.
- Self test their v6 device, which provides a marked up self test of the device.

Important: Tuner X does not require v6 and can be used with v5 flashed devices.

For a full list of features, check out the v6 documentation.

22.2.2 Introducing a new iteration of Tuner X

Some of you may have noticed that your version of Tuner X has changed recently. We've been working on several key improvements to the application that should dramatically improve the user experience. While this blog will highlight some of those, it's best to just try out the new Tuner yourself.

Note: Feedback is welcome and can be provided by emailing feedback@ctr-electronics.com.

Improved connection diagnostics

Tuner requires a running diagnostic server to work. Typically, this is installed through a robot program utilizing one of our devices. Alternatively, this program is temporarily run using a button in Settings. We've improved the disconnection status card to contain information about the ping of the target and diagnostic state of the device.
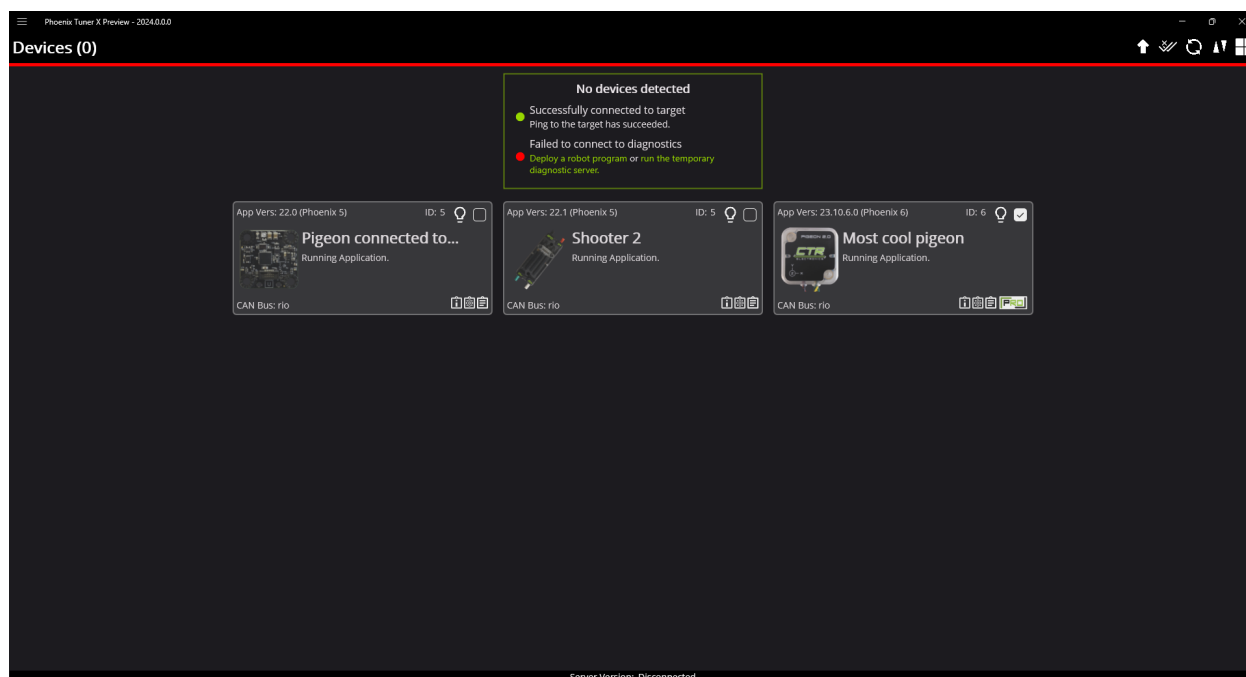
This 3 step check looks for the following:

1. Ping of the target.

2. Is diagnostics (or a robot program with diagnostics) running?

3. Are there any devices reported?

To summarize, if a user is not seeing devices in Tuner but checks 1 and 2 are good, then the next recommendation is to check the LED status of the device. We have an extensive list of status LEDs that indicate if the device is detected on a CAN bus, or other problems. This list can be found on the corresponding device page in the docs. For example, look at the CANcoder LED table.

Redesigned device overview

The device overview page has been redesigned to improve usability of plot, control and configuration. It's never been easier to tune your closed-loop gains directly in Tuner!
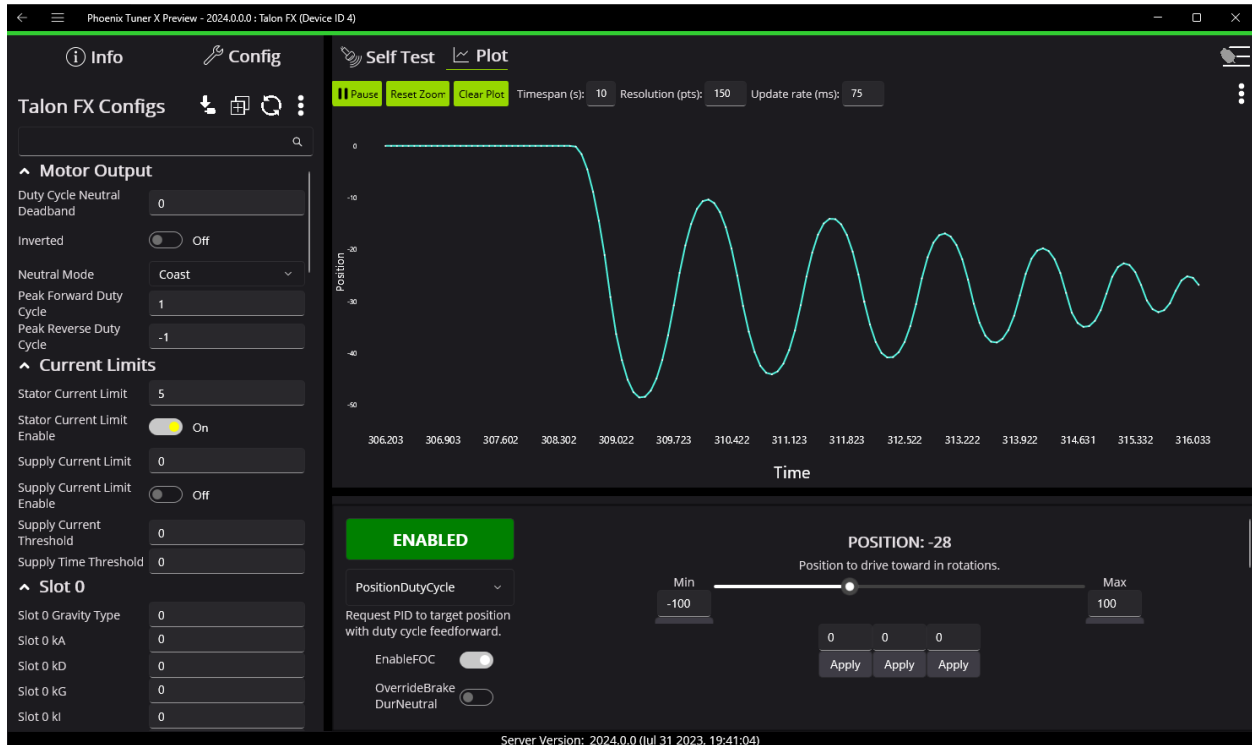
Bug squashing and usability improvements

This list is by no means exhaustive, but provides a good idea of the changes between 2023.X and 2024 versions of Tuner X.

- Firmware selection now has a dropdown for year, allowing you to flash older year firmware
- Dramatically improved startup and navigation performance
- Dramatically improved plotting performance
- Dramatically improved commands timing out on Android Tuner
- Enable/Disable button colors have been adjusted to be more clear
- Fixed "connection blipping" on Android Tuner
- Fixed control sometimes stuttering and causing the device to disable
- Fixed licensing sometimes fail to load on Android Tuner
- Fixed SSH credentials popup not appearing sometimes
- Fixed lag when entering into various entries
- Fixed memory leak when plotting for long periods of time
- Fixed situation where the application would shutdown uncleanly and lose settings

- Fixed various clipping of icons, text and labels

- Fixed issue where CANivore USB toggle would be unable to enable or disable

- Fixed firmware flashing on Raspberry Pi

- Fixed temporary diagnostic deployment on non-RIO platforms

- Slows down CANivore polling, which improves Rio CPU performance when Tuner is open

22.2.3 What's next?

We have a couple of exciting improvements to Tuner on our radar, keep an eye out on our changelog. Tuner X can be downloaded via the Microsoft Store and the Google Play Store.

Troubleshooting

23.1 CAN Bus Troubleshooting

There are typically two failure modes that must be resolved:

- There are same-model devices on the bus with the same device ID (devices have a default device ID of '0').

- CAN bus is not wired correctly or robustly

During hardware validation, you will likely have to isolate each device to assign a unique device ID.

Note: CTRE software has the ability to resolve device ID conflicts without device isolation, and CAN bus is capable of reporting the health of the CAN bus (see Driver Station lightening tab). However, the problem is when **both** root-causes are occurring at the same time, this can confuse students who have no experience with CAN bus systems.

Note: Many teams will pre-assign and update devices (Talon SRXs for example) long before the robot takes form. This is also a great task for new students who need to start learning the control system (with the appropriate mentor oversight to ensure hardware does not get damaged).

23.1.1 Identifying Duplicate IDs

Tip: Label the devices appropriately so there is no guessing which device ID is what. Don't have a label maker? Use tape and/or Sharpie (sharpie marks can be removed with alcohol).

Phoenix Tuner X will report when there are multiple devices of the same model with the same ID. This is shown when the device card is RED and there is a message in the middle of the device card. Users seeing this should iteratively *reassign IDs* on the device(s).

23.1.2 Check your wiring

Specific wiring instructions can be found in the user manual of each product, but there are common steps that must be followed for all devices:

- If connectors are used for CAN bus, **tug-test each individual crimped wire** one at a time. Bad crimps/connection points are the most common cause of intermittent connection issues.

- Confirm red and black are not flipped.

- Confirm battery voltage is adequate (through Driver Station or through voltmeter).

- Manually inspect and confirm that green-connects-to-green and yellow-connects-to-yellow at every connection point. **Flipping/mixing green and yellow is a common failure point during hardware bring up**.

- Confirm breakers are installed in the PDP where appropriate.

- Measure resistance between CANH and CANL when system is not powered (should measure ~60Ω). If the measurement is 120Ω, then confirm both RIO and PDP are in circuit, and PDP jumper is in the correct location.

23.1.3 LEDs are red - now what?

We need to rule out same-ID versus bad-bus-wiring.

There are two approaches:

- Approach 1 will help troubleshoot bad wiring and common IDs.

- Approach 2 will only be effective in troubleshooting common IDs, but this method is noteworthy because it is simple/quick (no wiring changes, just pull breakers).

The specific instructions for changing device ID are in the next section. Review this if needed.

Approach 1 (best)

- **Physically connect CAN bus from roboRIO to one device only. Circumvent your wiring if need be.**

- Power boot robot/bench setup.

- Open Phoenix Tuner X and wait for connection (roboRIO may take ~30 seconds to boot)

- Open the *Devices* page

- Confirm that CAN device appears

- Use Tuner X to change the device ID

- Label the new ID on the physical device

- Repeat this procedure for every device, one at a time

If you find a particular device where communication is not possible, scrutinize device's power and CAN connection to the system. Make the test setup so simple that the only failure mode possible is within the device itself.

Note: Typically, there must be two 120-Ω termination resistors at each end of the bus. CTR Electronics integrates termination resistors into the PDP and the CANivore. The roboRIO also has an integrated termination resistor. During bring-up, if you keep your harness short (such as the CAN pigtail leads from a single TalonFX) then a single resistor is adequate for testing purposes.

Approach 2 (easier)

- **Leave CAN bus wiring as is**
- **Pull breakers and PCM fuse from PDP**
- **Disconnect CAN bus pigtail from PDP**
- **Pick the first device to power up and restore breaker/fuse/pigtail so that only this CAN device is powered**
- Power boot robot/bench setup
- Open Phoenix Tuner X and wait for connection (roboRIO may take ~30 seconds to boot)
- Open the Devices page
- Confirm that CAN device appears
- If device does not appear, scrutinize device's power and CAN connection to the system
- Use Tuner X to change the device ID
- Label the new ID on the physical device
- Repeat this procedure for every device

If you find a particular device or section of devices where communication is not possible, then the CAN bus wiring needs to be re-inspected. Remember to "flick" / "shake" / "jostle" the CAN wiring in various sections to attempt to reproduce red LED blips. This is a sure sign of loose contact points.

If you are able to detect and change device ID on your devices individually, begin piecing your CAN bus together. Start with either roboRIO <—-> device <—> PDP, or CANivore <—-> device <—> 120 Ω resistor, to ensure termination exists at both ends. Then introduce the remaining devices until a failure is observed or until all devices are in-circuit.

If introducing a new device creates a failure symptom, scrutinize that device by replacing it, inspecting common wires, and inspecting power.

At the end of this section, all devices should appear (notwithstanding the above notes) and device LEDs should not be red. TalonFX and Pigeon2 typically blink orange when they are healthy and not controlled, and CANcoder rapid-blinks brightly. PDP may be orange or green depending on its sticky faults.

24

Support

CTR Electronics prides itself on excellent customer service. Our contact information can be found on our website.