
Phoenix 6

CTR Electronics

Apr 28, 2024

Getting Started

1	Requirements	3
2	Installing Phoenix 6	5
3	New for 2024	11
4	Phoenix 6 Features	21
5	API Migration	27
6	CAN Bus Utilization	53
7	Phoenix Tuner X	55
8	Device Licensing	87
9	Season Pass Licensing	93
10	TalonFX	95
11	Pigeon 2.0	105
12	CANcoder	109
13	General API Usage	113
14	Device API	143
15	Mechanisms	179
16	Simulation	189
17	WPILib Integration	195
18	Examples	207
19	CANivore Intro	213

20	CANivore Setup	215
21	CANivore API	219
22	Hardware-Attached Simulation	223
23	Advanced Configuration	225
24	Development Blog	229
25	Troubleshooting	241
26	Support	245

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.

Important: Users looking for beta documentation for the Phoenix 6 API can find it on the [latest version of this site](#).

CTR Electronics Blog For news and updates about your CTR Electronics device, please check out our [blog](#).

New for 2024 What's new in Phoenix 6 for 2024.

Migration Guide A Phoenix 5 migration guide is available [here](#).

Changelog A full changelog with bug fixes is available [here](#).

Known Issues A list of current known issues 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.

Requirements

This document explains the requirements to use Phoenix 6.

1.1 Supported Devices

Phoenix 6 supports the following devices:

- TalonFX (Falcon 500, Kraken X60)
- CANcoder
- Pigeon 2.0

1.2 CAN Bus Requirements

Phoenix 6 devices are supported on the below CAN bus adapters.

FRC

- roboRIO
- CANivore

non-FRC

Any SocketCAN capable adapter will work, but the [CANivore](#) is highly recommended. CANivore offers *additional functionality* over other SocketCAN adapters.

1.3 System Requirements

Phoenix 6 supports a plethora of languages and operating systems. The table found below details which languages are supported on what platforms.

FRC

Targets	Supported guages	Lan-	Supports <i>CANivore</i>	Supports <i>High-Fidelity Simulation</i>
NI roboRIO	Java, C++, Python		Yes	n/a
Windows 10/11 x86-64	Java, C++, Python		Yes	Yes
Linux x86-64 (desktop) ¹	Java, C++, Python		Yes	Yes
macOS	Java, C++, Python		No	Yes

non-FRC

Targets	Supported guages	Lan-	Supports <i>CANivore</i>	Supports <i>High-Fidelity Simulation</i>
Windows 10/11 x86-64	C#, C++, Python		Yes	Yes
Linux x86-64 (desktop) ¹	C++, Python		Yes	Yes
Linux ARM32 and ARM64 ²	C++, Python		Yes	No
macOS (Simulation Only)	C++, Python		No	Yes

¹ Supported Linux x86-64 (desktop) targets:

- Ubuntu 22.04 or newer
- Debian Bullseye or newer

² Supported Linux ARM32 and ARM64 targets:

- Raspberry Pi
- NVIDIA Jetson
- Ubuntu 20.04 or newer
- Debian Bullseye or newer

Installing Phoenix 6

Installing Phoenix 6 (FRC) [Click here](#) to learn about installing the Phoenix 6 library for FRC. This explains adding the Phoenix vendordep into your robot project.

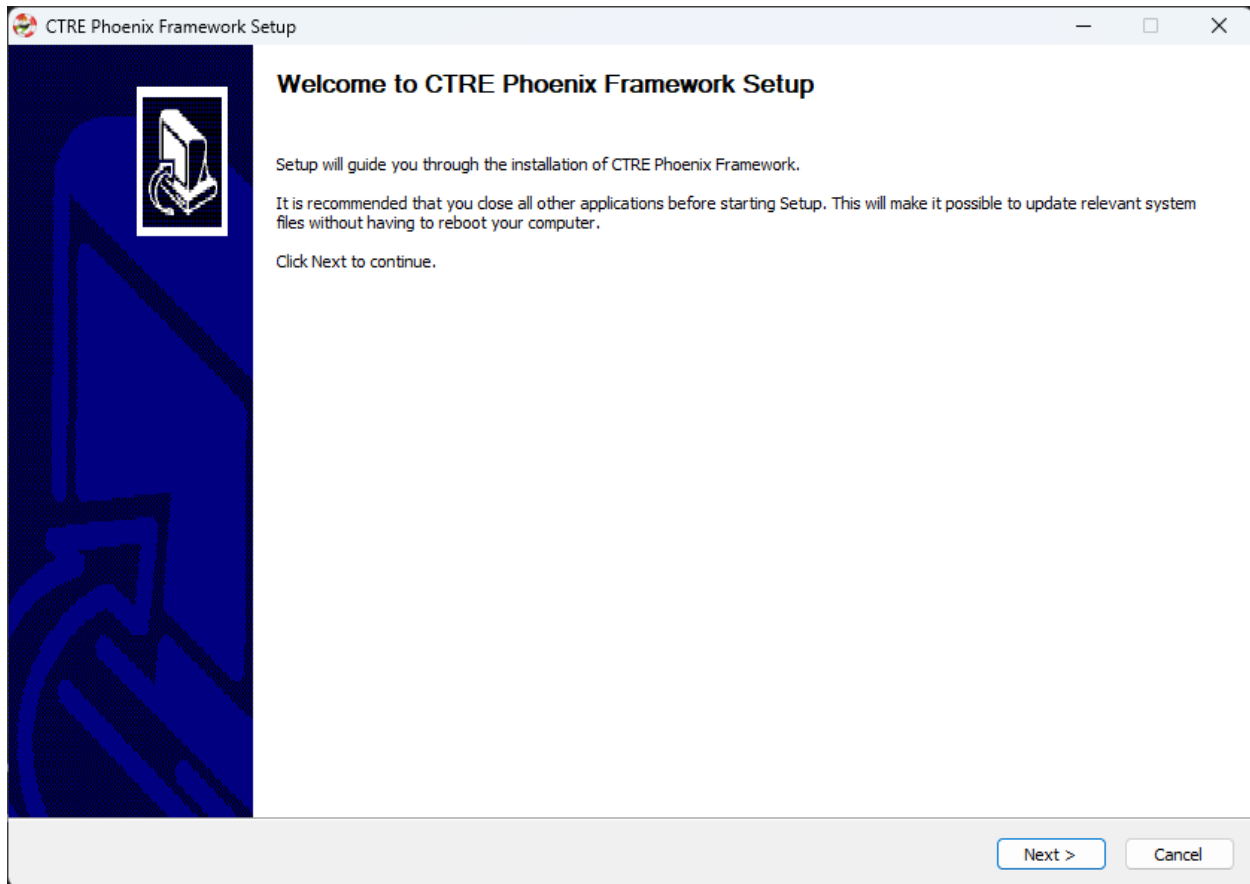
Installing Phoenix 6 (non-FRC) [Click here](#) to learn about installing the Phoenix 6 library outside of FRC. This explains including our apt-get repository and applicable binaries.

2.1 Installing Phoenix 6 (FRC)

Java/C++

Offline

1. Download the [Phoenix Framework Installer](#)
2. Navigate through the installer, ensuring applicable options are selected

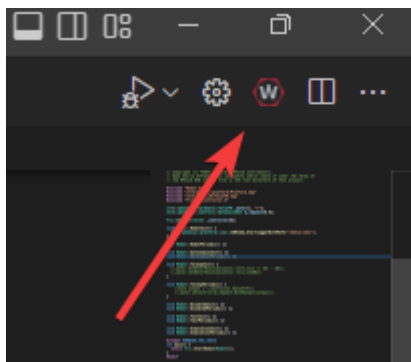


3. Apply the vendordep via WPILib VS Code [Adding Offline Libraries](#)

Online

Users 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.

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

- <https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/Phoenix6-frc2024-latest.json>

Additionally, v5 can safely installed alongside it by installing the v5 vendordep.

- <https://maven.ctr-electronics.com/release/com/ctre/phoenix/Phoenix5-frc2024-latest.json>

Important: Users utilizing only v5 devices still need the v6 vendordep added to their robot project.

Python

First, make sure to [install RobotPy](#). From there, installation of Phoenix 6 is available through [PyPI](#).

```
py -3 -m pip install phoenix6
```

LabVIEW

Download the Phoenix Offline Installer from the [Latest GitHub Release](#), and install it on the computer (with the LabVIEW component checked). This will put the Phoenix LabVIEW VIs into the “WPI Robotics Library -> Third Party -> CTRE” palette for LabVIEW development.

To deploy robot projects with Phoenix, you need to first download the Phoenix Libraries to the roboRIO. This can be done one of two ways:

1. Phoenix Tuner X under “Settings -> FRC Advanced -> Install LabVIEW”
2. LabVIEW under “Tools -> FIRST Robotics Tools -> Download CTRE Phoenix Libs”.

After the libraries are downloaded, hard deploy (run as startup) a LabVIEW program and restart the roboRIO.

Note: Currently only Phoenix 5 is supported in LabVIEW. A Phoenix 6 LabVIEW alpha is in development, teams that are interested in this may contact us directly at support@ctr-electronics.com.

2.2 Installing Phoenix 6 (non-FRC)

Important: Users in FRC can skip this article.

Phoenix 6 is fully supported outside of [FRC](#) on [supported](#) systems with a USB to CAN adapter.

Note: [CANivore](#) is the recommended USB to CAN adapter with support for [CAN FD](#), name specification, and Windows (as host).

2.2.1 Installation

API Installation

C++ (Linux)

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

```
sudo apt update
sudo apt install phoenix6
```

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

Python

Installation is available through [PyPI](#).

```
py -3 -m pip install phoenix6
```

C# (Windows)

Installation is available through [Nuget](#). An example on adding Nuget packages to a Visual Studio project is available in the [Microsoft Quickstart](#).

Firmware Installation

Device firmware upgrading is handled by the *Phoenix Tuner X* configuration client.

2.2.2 Examples

The same Phoenix 6 examples can be used as a reference outside of FRC. The function calls are identical, although the framework that is used to call them may be different and specialized for FRC.

2.2.3 Non FRC Projects

We recommend that users use the standard language projects to use our library outside of FRC. For example, a typical Python project will have a `main.py` that utilizes the `phoenix6` library that's installed with `pip`.

A typical C# project is a [Visual Studio](#) project that includes our libraries via `nuget`.

For C++, an example utilizing CMake is provided [here](#).

Note: When utilizing actuators outside of FRC, the user must continuously feed an enable signal to the device. For more information, see [Enabling Actuators](#).

2.2.4 Alternative Support

In the event that the Phoenix 6 API does not fit the user's needs (e.g. when using devices such as a [TalonSRX](#)), users can utilize the [Phoenix 5 API](#).

We also provide a hardware robot controller called the [HERO development board](#). The HERO can be programmed in C# to utilize [compatible v5](#) devices.

New for 2024

The CTR Electronics development team has been hard at work expanding the Phoenix 6 API based on user feedback. We are proud to announce several exciting new features with this release!

Firmware for the 2024 release of Phoenix 6 can be found by selecting “2024” in the firmware selection menu.

The API vendordep for 2024 is available under <https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/Phoenix6-frc2024-latest.json>.

Users will need to update both firmware and API to make use of these features.

Note: This changelog is intended to highlight the major additions to the Phoenix 6 API. For a detailed list of changes and bug fixes, visit the [Phoenix changelog](#).

3.1 Phoenix Pro Licensing

Introduced earlier this year is the new season pass licensing model. Season pass improves licensing flexibility when utilizing multiple robots and the roboRIO CAN bus. Additional information on this can be found in the [blog post](#).

As a reminder, many of the features available in Phoenix 6 do **not** require Pro to use. A full breakdown of what is and is not supported is available under [the feature table](#).

A variety of new (Pro and non-Pro) features have been added and are described in the API section below.

3.2 API

3.2.1 New Vendordep Procedures

Installing Phoenix 6 and Phoenix 5 has changed to make installation easier for teams using both Phoenix 5 and Phoenix 6 devices. See *Installing Phoenix 6 (FRC)* for more details.

3.2.2 New Language Support

Phoenix 6 now supports the following languages.

- Java
- C++
- Python
- C# (non-FRC only)

Feedback for the new language targets is welcome at feedback@ctr-electronics.com.

3.2.3 Swerve API

Now included in the Phoenix 6 API is a high performance Swerve API (Java). Teams using Phoenix 6 compatible devices for their swerve drivetrain can utilize the API to maximize performance and **eliminate** the boilerplate that comes with copying swerve template code. Empower your odometry and maximize your robot responsiveness with our synchronous, latency-compensated API with simulation support!

To complement the Swerve API, we've added a *project generation utility* to Tuner X to handle boilerplate related to inverts, offsets and gearing.

Swerve drive code is as easy as the following.

Note: The following example utilizes a generated TunerConstants class from Tuner X, but users can create the CommandSwerveDrivetrain or SwerveDrivetrain directly if they prefer. Additionally, this API is currently only available for Java.

```
/* Setting up bindings for necessary control of the swerve drive platform */
CommandXboxController joystick = new CommandXboxController(0); // My joystick
CommandSwerveDrivetrain drivetrain = TunerConstants.DriveTrain; // My drivetrain
SwerveRequest.FieldCentric drive = new SwerveRequest.FieldCentric().
    withIsOpenLoop(true); // I want field-centric

// driving in open loop
SwerveRequest.SwerveDriveBrake brake = new SwerveRequest.SwerveDriveBrake();
SwerveRequest.PointWheelsAt point = new SwerveRequest.PointWheelsAt();
Telemetry logger = new Telemetry(MaxSpeed);

private void configureBindings() {
    drivetrain.setDefaultCommand( // Drivetrain will execute this command periodically
        drivetrain.applyRequest(() ->
            drive.withVelocityX(-joystick.getLeftY() * MaxSpeed) // Drive forward with
```

(continues on next page)

(continued from previous page)

```

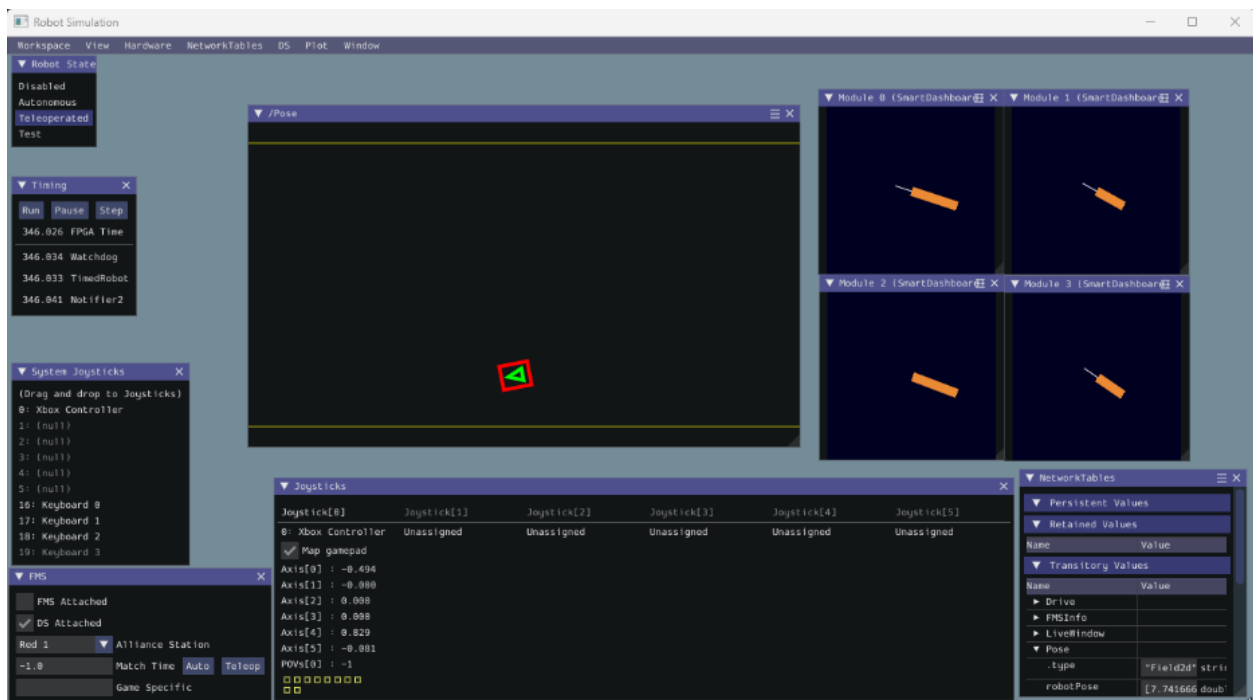
↪negative Y (forward)
    .withVelocityY(-joystick.getLeftX() * MaxSpeed) // Drive left with
↪negative X (left)
    .withRotationalRate(-joystick.getRightX() * MaxAngularRate) // Drive
↪counterclockwise with negative X (left)
    )
);

joystick.a().whileTrue(drivetrain.applyRequest(() -> brake));
joystick.b().whileTrue(drivetrain
    .applyRequest(() -> point.withModuleDirection(new Rotation2d(-joystick.
↪getLeftY(), -joystick.getLeftX()))));

if (Utils.isSimulation()) {
    drivetrain.seedFieldRelative(new Pose2d(new Translation2d(), Rotation2d.
↪fromDegrees(90)));
}

drivetrain.registerTelemetry(logger::telemeterize);
}

```



Important: Swerve API requires all necessary swerve devices to be v6 devices. e.g. 4 drive TalonFX, 4 steer TalonFX, 1 Pigeon 2.0, 4 CANcoders.

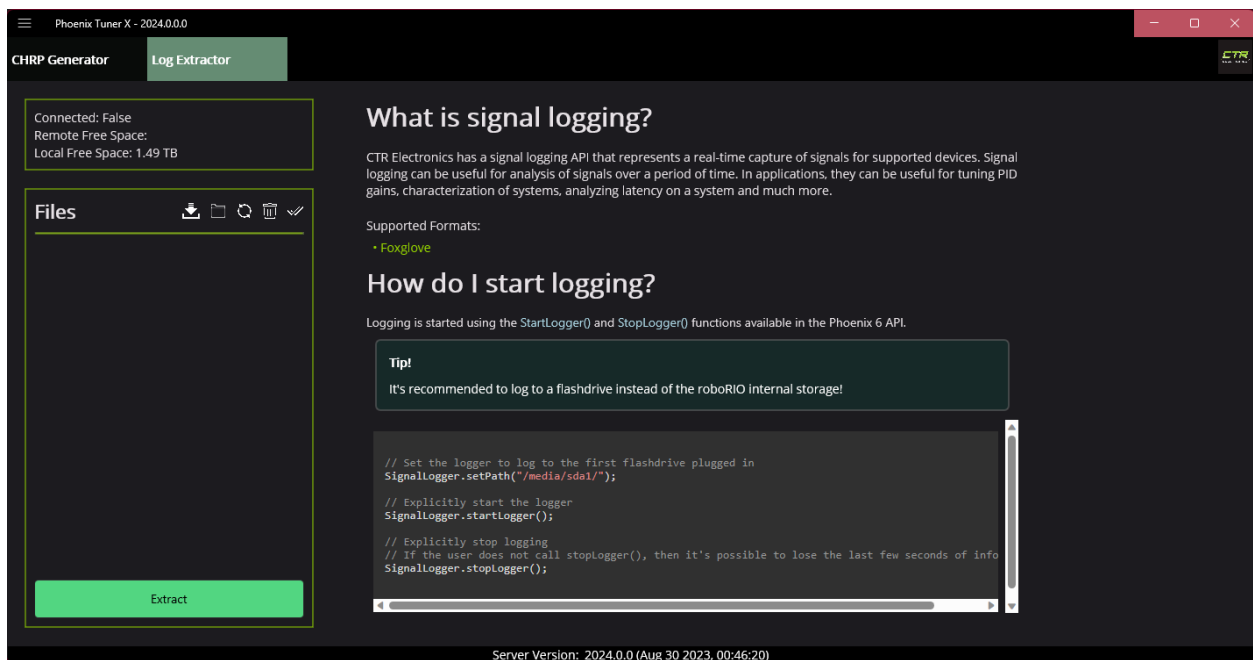
3.2.4 Signal Logging

We've added a comprehensive signal logger ([Java](#), [C++](#), [Python](#), [C#](#)) that provides a real-time capture of signals for supported devices. Signal logging can be useful for analysis of signals over a period of time. In applications, they can be useful for tuning PID gains, characterization of systems, analyzing latency on a system and much more. Did something unexpected happen in a match? Go back and check your logs to inspect positions, velocities, voltages, currents, temperatures, etc. Logging is automatic, and does not require choosing which signals you need captured ahead of time.

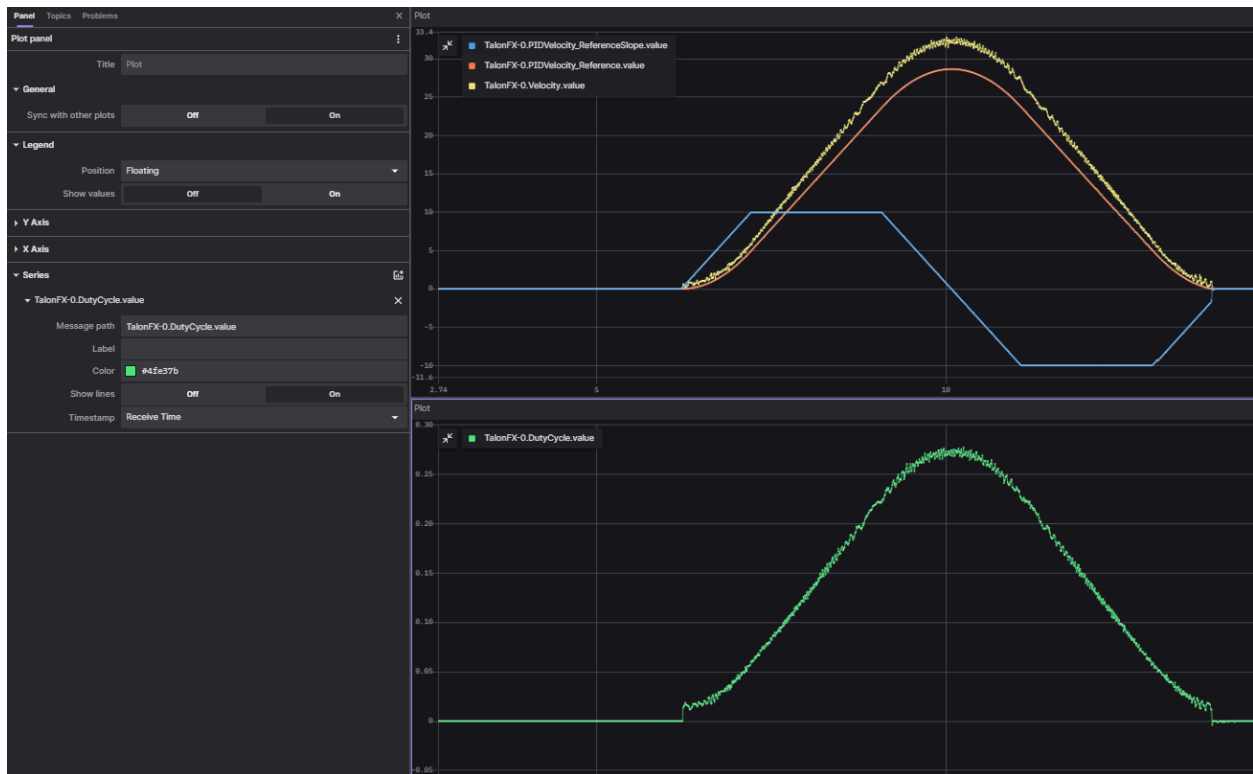
Important: MCAP Export requires Pro Licensing

Note: Documentation on configuring and extracting logs will be available soon.

Log Extractor Logs can be extracted and converted to compatible formats directly in Tuner X.



Foxglove Log Analysis Logs can then be analyzed in [Foxglove](#) to identify hardware failures, tuning gains, etc.



3.2.5 Signal API Improvements

Users can now disable signals by setting their update rate to 0 Hz. To reduce boilerplate when disabling signals, we have added an `optimizeBusUtilization()` function on device objects. This will automatically disable all signals that have not explicitly been given an update frequency with `setUpdateFrequency()`. There is also a `ParentDevice.optimizeBusUtilizationForAll()` static function that takes a list of devices to optimize. Additionally, update frequencies are automatically reapplied when devices reset.

Setting a given signal's frequency behavior has been improved by keeping track of the signal with the highest frequency in a frame. The highest frequency of all signals in the frame is used for the frame instead of the most recent signal.

Additionally, the following new functions have been added.

- `BaseStatusSignal.refreshAll()`
 - Refreshes all passed in signals
- `BaseStatusSignal.setUpdateFrequencyForAll()`
 - Applies the given update frequency to all signals that are passed in
- `getAppliedUpdateFrequency()`
 - Retrieves the actual update frequency of a given signal

3.2.6 New Motion Magic® Controls

We have added a Motion Magic® Velocity control mode, which produces a motion profile in real-time for a velocity controller. This allows for smooth transitions between velocity setpoints. Additionally, we have added a Dynamic Motion Magic® control mode for our Pro CANivore users, which supports modifying the cruise velocity, acceleration, and jerk settings during motion.

Furthermore, we have added Motion Magic® Expo control requests. Whereas traditional Motion Magic® uses a trapezoidal profile or an S-Curve, Motion Magic® Expo uses an exponential profile. The profile follows the kV and kA characteristics of the system, and optionally a cruise velocity. This allows the profile to best match the system dynamics, reducing both overshoot and time to target.

For a full list of new Motion Magic® controls, see the controls API documentation ([Java](#), [C++](#), [Python](#), [C#](#)).

3.2.7 Differential Mechanisms

Important: DifferentialMechanism requires both Pro and CANFD. SimpleDifferentialMechanism is a lower performance alternative that requires neither.

DifferentialMechanism ([Java](#), [C++](#)) provides an easy way to control two-axis differential mechanisms, such as a two motor elevator (motors on the left and right sides of the elevator that are not mechanically linked).

3.2.8 New Configs

We've added several new configs. A full list of available configs is available in the configs ([Java](#), [C++](#), [Python](#), [C#](#)) namespace.

3.2.9 Improved Support for roboRIO Motion Profiles

In addition to the kS and kV feedforward terms supported in 2023, Phoenix 6 now supports kG and kA. kG can be constant for use with an elevator, or it can calculate feedforward based on the cosine of position for mechanisms such as a rotating arm.

Additionally, there is now improved integration with roboRIO motion profiling using velocity setpoints in position control modes, along with acceleration setpoints in velocity control modes.

```
// Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
final TrapezoidProfile m_profile = new TrapezoidProfile(
    new TrapezoidProfile.Constraints(80, 160)
);
// Final target of 200 rot, 0 rps
TrapezoidProfile.State m_goal = new TrapezoidProfile.State(200, 0);
TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();

// robot loop
```

(continues on next page)

(continued from previous page)

```
m_setpoint = m_profile.calculate(0.020, m_setpoint, m_goal);
m_positionControl.Position = m_setpoint.position;
m_positionControl.Velocity = m_setpoint.velocity;
m_talonFX.setControl(m_positionControl);
```

3.2.10 New SyncCANcoder Remote Sensor

Added support for SyncCANcoder feedback. This allows users to synchronize the TalonFX's internal rotor sensor against the remote CANcoder, but continue to use the rotor sensor for all closed loop control. TalonFX will continue to monitor the remote CANcoder and report if its internal position differs significantly from the reported position, or if the remote CANcoder disappears from the bus. Users may want this instead of FusedCANcoder if there is risk that the sensor can fail in a way that the sensor is still reporting "good" data, but the data does not match the mechanism, such as if the entire sensor mount assembly breaks off. Users using this over FusedCANcoder will not have the backlash compensation, as the CANcoder position is not continually fused in.

3.2.11 Miscellaneous Improvements

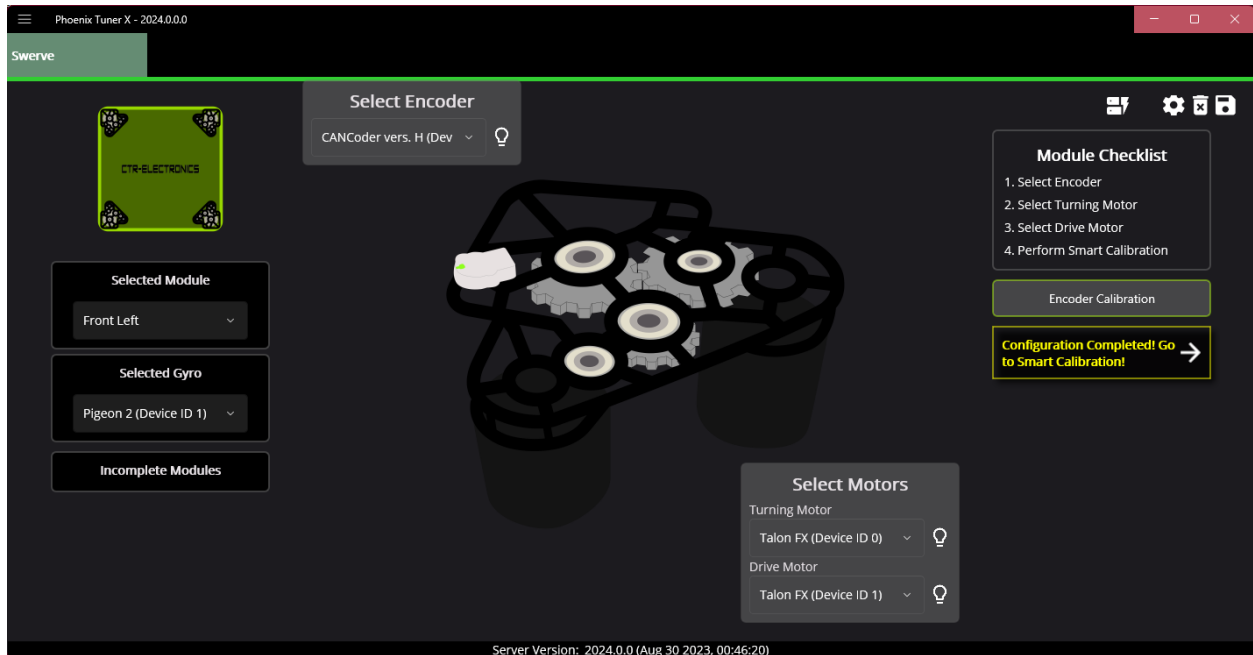
- Orchestra (Java, C++, Python, C#) has been ported from Phoenix 5.
 - Now supports multiple devices playing a single track.
 - Now works when the robot is disabled.
 - A new MusicTone control mode has been added and can be used for playing a specific frequency.
- Remote limits have been ported from Phoenix 5.
- Improved support for *unit tests*.

3.3 Tuner X

3.3.1 Swerve Project Generator

Swerve has many common pitfalls (inverts, encoder offsets, gearing, etc.). Utilizing our new Tuner X *Swerve Project Generator* can help eliminate these problems. This utility will guide the user through specifying their drivebase characteristics, device selection, CANcoder offset configuration and drive/steer validation. This utility will then generate a project that provides minimum viable swerve control!

Important: This utility does not characterize the swerve. To maximize robot responsiveness, we recommend characterizing and modifying the gains specified in the generated TunerConstants class.

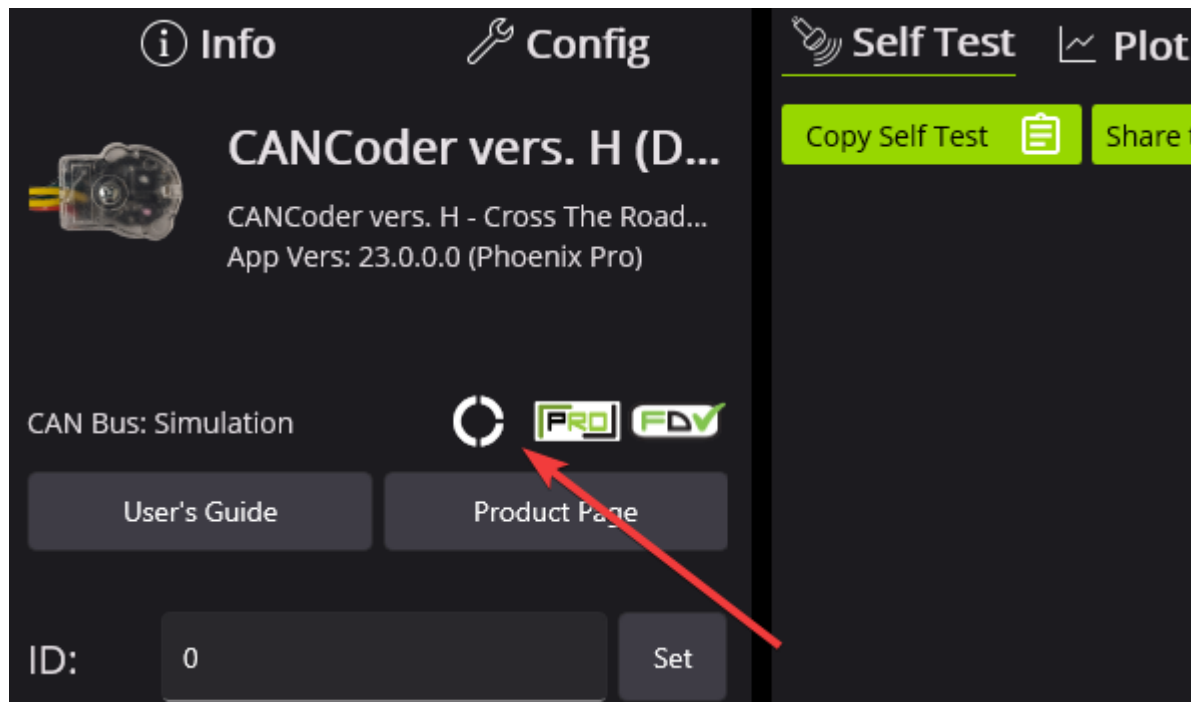


Note: The Swerve Project Generator is only supported in FRC Java.

3.3.2 CANcoder Zero Button

Important: This feature requires 2024 diagnostics or newer.

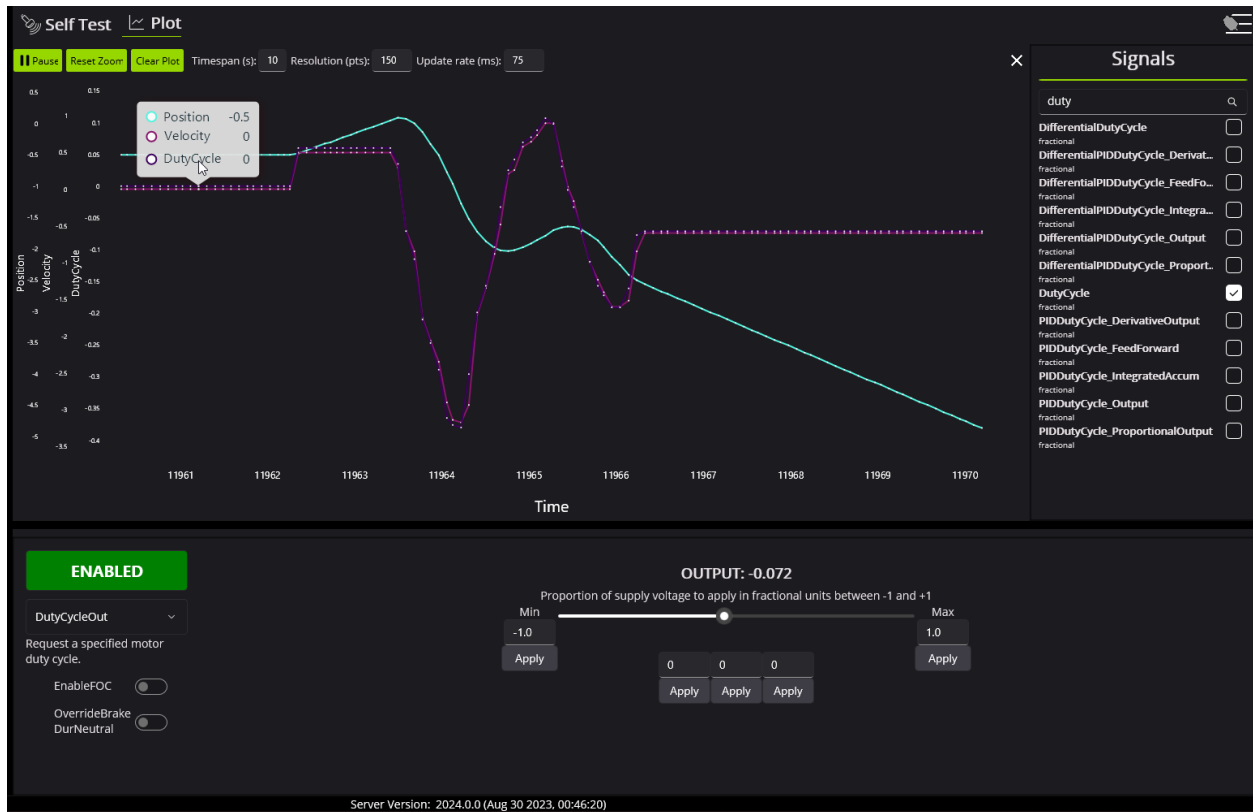
CANcoders can be zeroed by pressing on the button shown below. This applies an offset to the encoder config and reports the applied offset to the user.



3.3.3 Improved Plotting

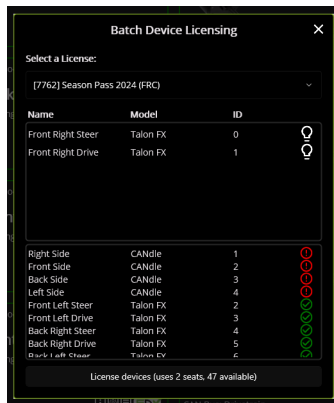
Important: This feature requires 2024 diagnostics or newer.

All signals exposed in API can now be plotted directly in Tuner X.



3.3.4 Batch Licensing

Tuner X now supports licensing multiple devices at once using *batch licensing*, streamlining the process of applying a Season Pass to your devices.



Phoenix 6 Features

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.
- New *Motion Magic*® *Expo* control to use an exponential profile following system dynamics, reducing both overshoot and time to target.
- Kalman-based algorithms to reduce latency while maintaining smooth data.

4.1.4 Swerve API

- High-performance Swerve API using synchronous, latency-compensated odometry.
- Eliminate the boilerplate from copying swerve template code.
- Improved *odometry performance* with CANivore and Phoenix Pro.
- Tuner X *Swerve Project Generator* gets swerve drive up and running quickly.
- Built-in high-fidelity *simulation* support.

4.1.5 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.

4.1.6 New Tuner X Self Tests

- Detailed and resolute self tests to improve debugging.

4.1.7 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.

4.1.8 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.
- Device timestamps captured when the signal is sampled provides best possible latency compensation.

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.

4.2.4 Real-Time High-Fidelity Signal Logger

- Log all status signals from every device with timestamps from CAN
- Data captured as it arrives at the full update rate of the status signals
- Improved sensitivity and accuracy of system identification with *WPILib SysID*
- Automatically starts logging during an FRC match
- Support for *custom user signals* alongside auto-captured data
- Efficient hoot logging format minimizes disk space and CPU usage
- *Export to multiple formats* including WPILOG and MCAP
- Free users can export a *limited set of signals* to WPILOG

4.3 Feature Breakdown

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

Feature	Phoenix 6 (rio)	Phoenix 6 + Pro (rio)	Phoenix 6 (CANivore)	Phoenix 6 + Pro (CANivore)
Canonical Units	x	x	x	x
Improved Bus Utilization	x	x	x	x
CANcoder Always Absolute	x	x	x	x
Kalman-based Velocity	x	x	x	x
Synchronous Wait for Data	x	x	x	x
System Time-stamps	x	x	x	x
Limited Signal Logger WPI-LOG Export	x	x	x	x
Explicit Control Requests	x	x	x	x
Motion Magic® Velocity	x	x	x	x
Motion Magic® Expo	x	x	x	x
Continuous Wrap Mode	x	x	x	x
Improved Self-Test Snapshot	x	x	x	x
Tuner X Improved Plotting	x	x	x	x
CANivore Timestamps			x	x
CAN FD			x	x
Field Oriented Control (FOC)		x		x
Fused CAN-coder + TalonFX		x		x
Sync CANcoder + Talon FX		x		x
Full Signal Logger Export		x		x
Signal Logger MCAP Export		x		x
Time-Synced Signal Publishing				x
Device Time-stamps				x
Dynamic Motion Magic®				x

Note: + The Java Swerve API is freely available, however performance improves when used on a CANivore bus and further improves when used with Pro devices.

API Migration

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

- **API Structure**
 - General structure of the Phoenix 6 namespaces and packages
- **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 API Structure

Phoenix 6 uses a separate, simpler set of namespaces and packages from Phoenix 5.

Note: For more information about the structure of Phoenix 6, see [API Overview](#).

v

Java

```
// Phoenix 5 is in the com.ctre.phoenix.* packages
import com.ctre.phoenix.motorcontrol.can.WPI_TalonFX;
import com.ctre.phoenix.motorcontrol.TalonFXConfiguration;
import com.ctre.phoenix.motorcontrol.TalonFXControlMode;
import com.ctre.phoenix.motorcontrol.TalonFXInvertType;
import com.ctre.phoenix.motorcontrol.TalonFXSimCollection;
import com.ctre.phoenix.sensors.CANCoderConfiguration;
import com.ctre.phoenix.sensors.WPI_CANCoder;

// WPI_* for WPILib integration
final WPI_TalonFX m_talonFX = new WPI_TalonFX(0);
final WPI_CANCoder m_cancoder = new WPI_CANCoder(0);

final TalonFXSimCollection m_talonFXSim = m_talonFX.getSimCollection();

final TalonFXConfiguration m_talonConfig = new TalonFXConfiguration();
final CANCoderConfiguration m_cancoderConfig = new CANCoderConfiguration();

TalonFXInvertType m_talonFXInverted = TalonFXInvertType.CounterClockwise;

m_talonFX.set(TalonFXControlMode.PercentOutput, 0);
```

C++

```
// Phoenix 5 is in the ctre/phoenix headers
#include "ctre/phoenix/motorcontrol/can/WPI_TalonFX.h"
#include "ctre/phoenix/sensors/WPI_CANCoder.h"

// Phoenix 5 uses the ctre::phoenix namespace
using namespace ctre::phoenix;

// WPI_* for WPILib integration
motorcontrol::can::WPI_TalonFX m_talonFX{0};
sensors::WPI_CANCoder m_cancoder{0};

motorcontrol::TalonFXSimCollection& m_talonFXSim{m_talonFX.GetSimCollection()};

motorcontrol::TalonFXConfiguration m_talonConfig{};
sensors::CANCoderConfiguration m_cancoderConfig{};

motorcontrol::TalonFXInvertType m_talonFXInverted
    ← {motorcontrol::TalonFXInvertType::CounterClockwise};

m_talonFX.Set(motorcontrol::TalonFXControlMode::PercentOutput, 0);
```

v

Java

```
// Phoenix 6 is in the com.ctre.phoenix6.* packages
import com.ctre.phoenix6.configs.CANCoderConfiguration;
import com.ctre.phoenix6.configs.TalonFXConfiguration;
import com.ctre.phoenix6.controls.DutyCycleOut;
import com.ctre.phoenix6.hardware.CANCoder;
import com.ctre.phoenix6.hardware.TalonFX;
import com.ctre.phoenix6.signals.InvertedValue;
import com.ctre.phoenix6.sim.TalonFXSimState;
```


5.2 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](#).

5.2.1 Applying Configs

v

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++

```
// 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

```
var talonFXConfigs = new TalonFXConfiguration();

// set slot 0 gains and leave every other config factory-default
var slot0Configs = talonFXConfigs.Slot0;
slot0Configs.kV = 0.12;
slot0Configs.kP = 0.11;
slot0Configs.kI = 0.5;
slot0Configs.kD = 0.001;

// apply all configs, 50 ms total timeout
m_talonFX.getConfigurator().apply(talonFXConfigs, 0.050);
```

C++

```
configs::TalonFXConfiguration talonFXConfigs{};

// set slot 0 gains and leave every other config factory-default
configs::Slot0Configs& slot0Configs = talonFXConfigs.Slot0;
slot0Configs.kV = 0.12;
slot0Configs.kP = 0.11;
slot0Configs.kI = 0.5;
slot0Configs.kD = 0.001;

// apply all configs, 50 ms total timeout
m_talonFX.GetConfigurator().Apply(talonFXConfigs, 50_ms);
```

Factory Defaulting Configs

v

Java

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

C++

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

v

Java

```
// Any unmodified configs in a configuration object are *automatically* factory-
↪defaulted.
// As a result, factory-defaulting before applying configs is *unnecessary* when
↪using a
// full device configuration object, such as TalonFXConfiguration.

// Users 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;
// As a result, factory-defaulting before applying configs is *unnecessary* when
↪using a
// full device configuration object, such as TalonFXConfiguration.

// Users can perform a full factory default by passing a new device configuration
↪object.
m_motor.GetConfigurator().Apply(configs::TalonFXConfiguration{});
```

5.2.2 Retrieving Configs

v

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++

```
// 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

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

C++

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

5.3 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](#).

5.3.1 Using Status Signals

v

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++

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

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

v

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++

```
// 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();
```

```
// 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

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

C++

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

v

Java

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

C++

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

Note: When different update frequencies are specified for signals that share a status frame, the highest update frequency of all the relevant signals will be applied to the entire frame. Users can get a signal's applied update frequency using the `getAppliedUpdateFrequency()` method.

5.3.2 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*

5.4 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](#).

5.4.1 Using Control Requests

v

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++

```
// 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);
```

v

Java

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

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

C++

```
// 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

```
// 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++

```
// 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

```
// class member variables
final DutyCycleOut m_request = new DutyCycleOut(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()));
```

40

```
// 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
```

Changing Update Frequency (Control Frame Period)

v

Java

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

C++

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

v

Java

```
// class member variables
final DutyCycleOut m_request = new DutyCycleOut(0);

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

C++

```
// class member variables
controls::DutyCycleOut 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.

5.4.2 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	<p>This has been deprecated in Phoenix 6.</p> <ul style="list-style-type: none"> • 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)

5.5 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 Overview](#).

5.5.1 Closed-Loop Setpoints

Phoenix 6 uses canonical units for closed-loop setpoints.

5.5.2 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.

5.5.3 Using Closed-Loop Control

v

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++

```
// 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);
```

v

Java

```
// class member variable
final 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.getControl(m_velocity.withVelocity(50));
```

C++

Motion Magic®

v

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++

```
// 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

Closed-loop control requests have been expanded to support motion profiles generated by the robot controller.

Java

```
// class member variable
final PositionVoltage m_position = new PositionVoltage(0);
// Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
final TrapezoidProfile m_profile = new TrapezoidProfile(
    new TrapezoidProfile.Constraints(80, 160)
);
// Final target of 200 rot, 0 rps
TrapezoidProfile.State m_goal = new TrapezoidProfile.State(200, 0);
TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();

// robot init, set slot 0 gains
var slot0Configs = new Slot0Configs();
slot0Configs.kS = 0.24; // add 0.24 V to overcome friction
slot0Configs.kV = 0.12; // apply 12 V for a target velocity of 100 rps
slot0Configs.kP = 4.8;
slot0Configs.kI = 0;
slot0Configs.kD = 0.1;
m_talonFX.getConfigurator().apply(slot0Configs, 0.050);

// periodic, update the profile setpoint for 20 ms loop time
m_setpoint = m_profile.calculate(0.020, m_setpoint, m_goal);
// apply the setpoint to the control request
m_position.Position = m_setpoint.position;
m_position.Velocity = m_setpoint.velocity;
m_motor.setControl(m_position);
```

C++

```
// class member variable
controls::PositionVoltage m_position{0_tr};
// Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
frc::TrapezoidProfile<units::turns> m_profile{{80_tps, 160_tr_per_s_sq}};
// Final target of 200 rot, 0 rps
frc::TrapezoidProfile<units::turns>::State m_goal{200_tr, 0_tps};
frc::TrapezoidProfile<units::turns>::State m_setpoint{};

// robot init, set slot 0 gains
configs::Slot0Configs slot0Configs{};
slot0Configs.kS = 0.24; // add 0.24 V to overcome friction
slot0Configs.kV = 0.12; // apply 12 V for a target velocity of 100 rps
slot0Configs.kP = 4.8;
slot0Configs.kI = 0;
slot0Configs.kD = 0.1;
m_talonFX.GetConfigurator().Apply(slot0Configs, 50_ms);

// periodic, update the profile setpoint for 20 ms loop time
```

(continues on next page)

(continued from previous page)

```
m_setpoint = m_profile.Calculate(20_ms, m_setpoint, m_goal);  
// apply the setpoint to the control request  
m_position.Position = m_setpoint.position;  
m_position.Velocity = m_setpoint.velocity;  
m_motor.SetControl(m_position);
```

5.6 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.

5.6.1 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.

5.6.2 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++).

5.6.3 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++).

5.6.4 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.

5.6.5 Sensor Initialization Strategy

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

5.6.6 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++](#)).

5.6.7 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 (~1 ms). As a result, the velocity measurement period/window configs are no longer necessary in Phoenix 6 and have been removed.

5.6.8 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.

5.6.9 CANcoder Sensor Coefficient and Units

In Phoenix 6, CANcoder does not support setting a custom sensor coefficient, unit string, and sensor time base. Instead, the CANcoder uses canonical units of rotations and rotations per second using the [C++ units library](#).

5.6.10 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
CANdle Support	Normal priority

5.6.11 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 have been improved to cover many of the use cases of the Motion Profile Executor.
- Allowable Closed-Loop Error

CAN Bus Utilization

CTR Electronics goes through great efforts to make our products efficient on CAN bus bandwidth. This article highlights the average default bus 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 [status signal](#) and [control request](#) documentation.

Note: Using Phoenix API will automatically start up a diagnostic 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) ^{Page 53, 1}
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%

¹ Phoenix 6 devices on CAN FD also increase the default update frequency of many status signals to 100 Hz.

Phoenix Tuner X

7.1 What is Phoenix Tuner X?

Phoenix Tuner X is the companion application allowing you to update, configure, analyze, and control your devices. Phoenix Tuner X can be installed from the [Microsoft Store](#) and the [Google Play Store](#).

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

Important: While CTR Electronics supports both Phoenix Tuner v1 and Phoenix Tuner X, certain features such as device licensing, Phoenix 6 Self Test, and improved field upgrading are only available in Phoenix Tuner X.

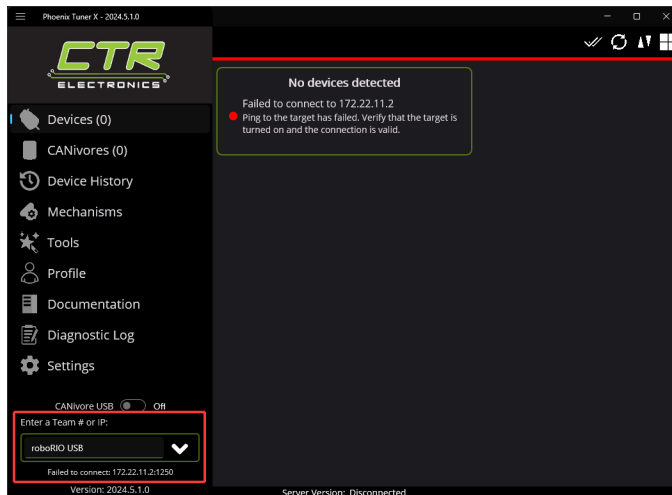
Tip: Many UI elements contain hover tooltips. That means the user can hover over them with their mouse for a text explanation of what they do.

7.1.1 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.



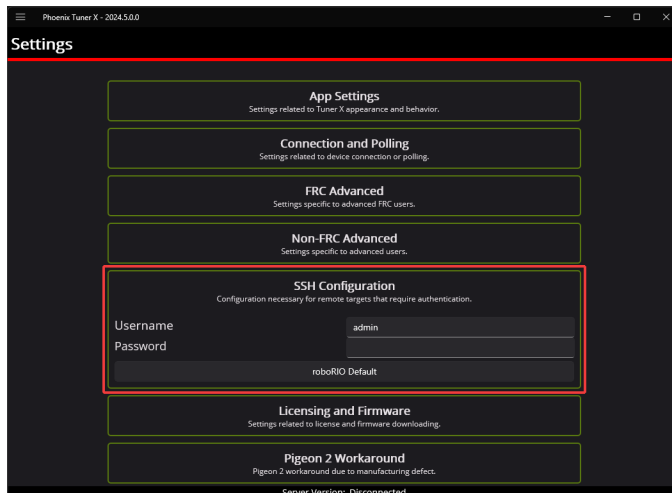
By clicking the arrow, you can change between presets such as:

- *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.

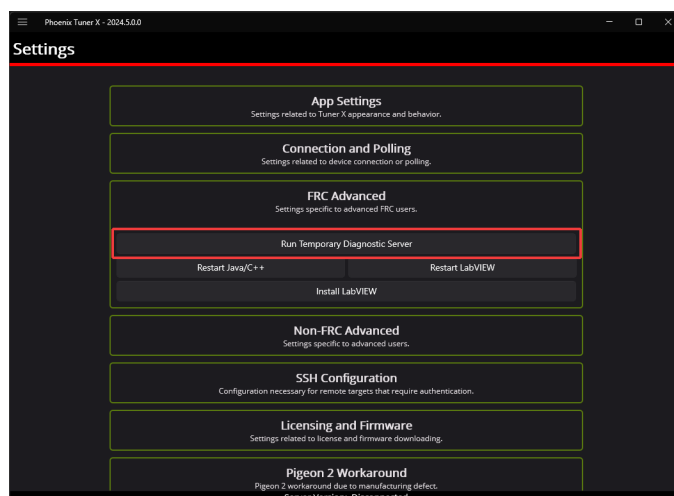
Configuring SSH Credentials (non-FRC)

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



Temporary Diagnostics (FRC)

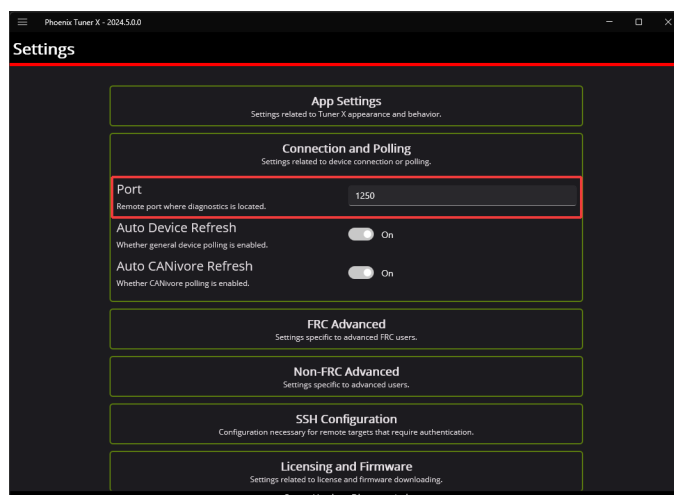
Devices can be configured without a diagnostic server present. This can be useful if the roboRIO has been freshly imaged. Ensure that you are pointed at the roboRIO IP address (usually 10.TE.AM.2 where TE.AM is the team number) and then click the *Run Temporary Diagnostic Server* in *Settings*.



Changing Diagnostics Server Port (non-FRC)

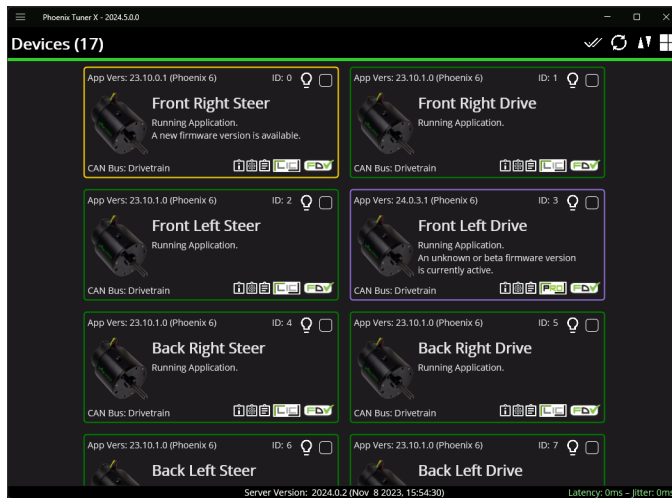
The target server port can be changed in the Tuner X *Settings* page, which is accessed from the flyout menu.

Important: The default port for diagnostic server is 1250. FRC users should not change this.



7.1.2 Device List

Card Layout



Grid Layout

Name	Model	ID	CAN Bus	FirmVer	Status	Is Checked?
Front Right Steer	Talon FX	0	Drivetrain	23.10.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Front Right Drive	Talon FX	1	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Front Left Steer	Talon FX	2	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Front Left Drive	Talon FX	3	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Right Steer	Talon FX	4	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Right Drive	Talon FX	5	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Left Steer	Talon FX	6	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Left Drive	Talon FX	7	Drivetrain	23.10.1.0 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Front Right	CANCoder vers. H	0	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Front Left	CANCoder vers. H	1	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Right	CANCoder vers. H	2	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Back Left	CANCoder vers. H	3	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Pigeon 2 (Device ID 1)	Pigeon 2	1	Drivetrain	24.0.3.1 (Phoenix 6)	Running Application.	<input type="checkbox"/>
Right Side	CANdle	1	Drivetrain	22.3.1.0 (Phoenix 5)	Running Application.	<input type="checkbox"/>
Front Side	CANdle	2	Drivetrain	22.3.1.0 (Phoenix 5)	Running Application.	<input type="checkbox"/>

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).

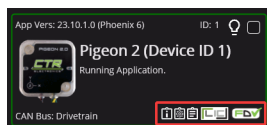
Card Colors

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 an unexpected/beta 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.

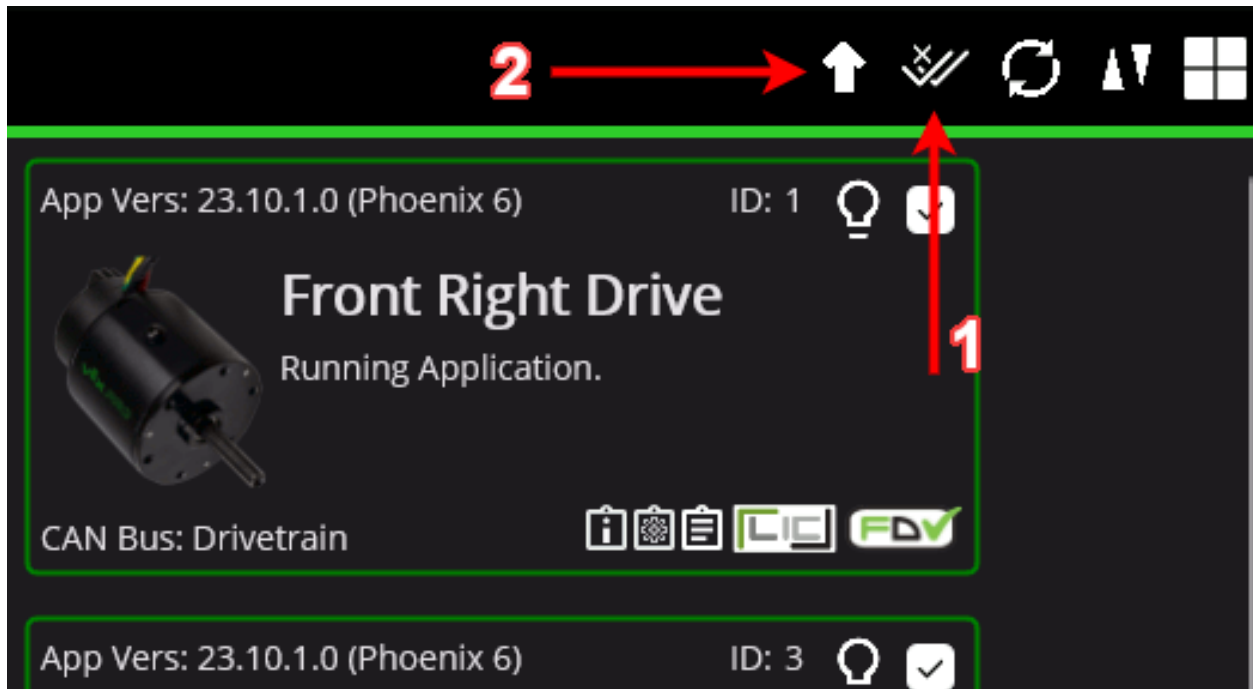
Note: The **CAN FD** icon does **not** indicate that the device is currently on a CAN FD bus, merely that it supports CAN FD.

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.

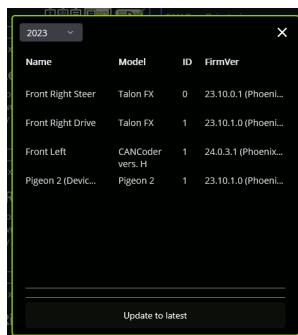
Tip: Selecting a device using their checkbox and clicking the checkmark in the top right will select all devices of the same models



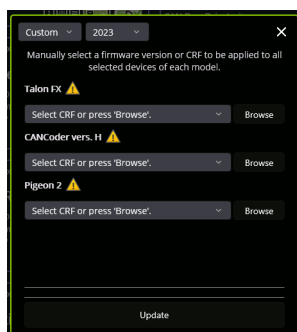
Step 1 in the above image selects all devices of the same models selected (or all devices if no device is currently check-boxed).

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

Once the dialog is opened, information detailing the device name, model, ID, and firmware version is presented. There is a year selector in the top-left corner to select the firmware version year.



Once the correct firmware year is selected user can begin the upgrade progress by selecting *Update to latest*. If the user does not want to use the latest firmware version, the *Custom* year selection allows for the selection of a specific firmware version for each device 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.

7.1.3 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.

Phoenix Tuner X - 2024.5.0.0

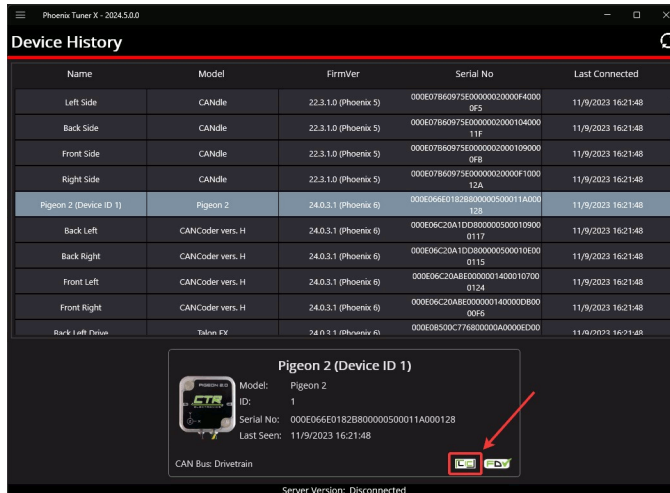
Device History

Name	Model	FirmVer	Serial No	Last Connected
Left Side	CANdle	22.3.1.0 (Phoenix 5)	000E07B60975E0000002000F4000 9F5	11/9/2023 16:21:48
Back Side	CANdle	22.3.1.0 (Phoenix 5)	000E07B60975E0000002000104000 11F	11/9/2023 16:21:48
Front Side	CANdle	22.3.1.0 (Phoenix 5)	000E07B60975E0000002000109000 9F8	11/9/2023 16:21:48
Right Side	CANdle	22.3.1.0 (Phoenix 5)	000E07B60975E0000002000F1000 12A	11/9/2023 16:21:48
Pigeon 2 (Device ID 1)	Pigeon 2	24.0.3.1 (Phoenix 6)	000E066E0182B800000500011A000 128	11/9/2023 16:21:48
Back Left	CANCoder vers. H	24.0.3.1 (Phoenix 6)	000E06C20A1DC0800000500010900 0117	11/9/2023 16:21:48
Back Right	CANCoder vers. H	24.0.3.1 (Phoenix 6)	000E06C20A1DC0800000500010E00 0115	11/9/2023 16:21:48
Front Left	CANCoder vers. H	24.0.3.1 (Phoenix 6)	000E06C20A8E000001400010700 0124	11/9/2023 16:21:48
Front Right	CANCoder vers. H	24.0.3.1 (Phoenix 6)	000E06C20A8E00000140000C800 00F6	11/9/2023 16:21:48
Back Left Talon	Talon FX	24.0.3.1 (Phoenix 6)	000E08500C776800000A0000ED00	11/9/2023 16:21:48

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.

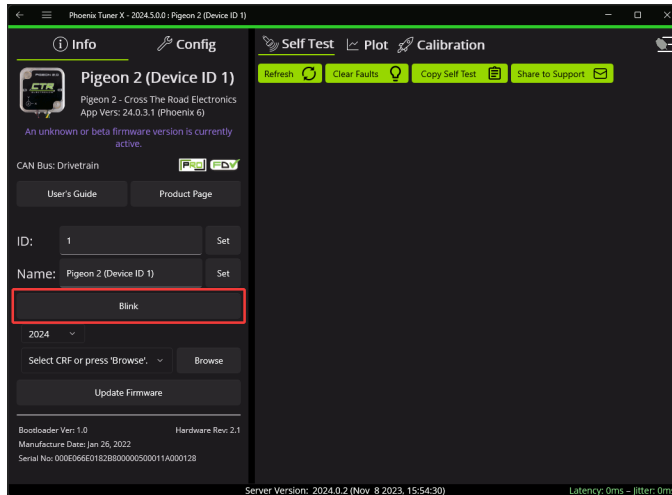
7.1.4 Device Details

The **Device Details** page can be accessed by clicking on the device card (or clicking on *View more details...* 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

Blinking

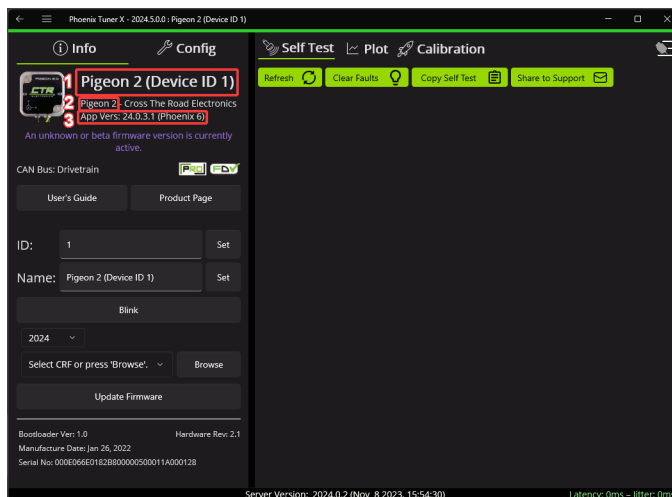
All CTR Electronics devices can be blinked (rapidly flash the LEDs). This can be useful for handling whenever you have duplicate devices using the same ID on the CAN bus.



Verifying Device Details

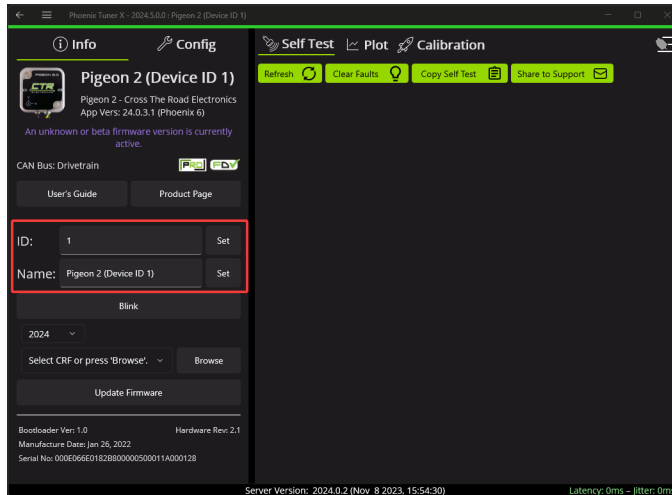
This screen highlights information such as (1) Device Name, (2) Device Model, (3) Firmware Version.

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



Configuring Name & IDs

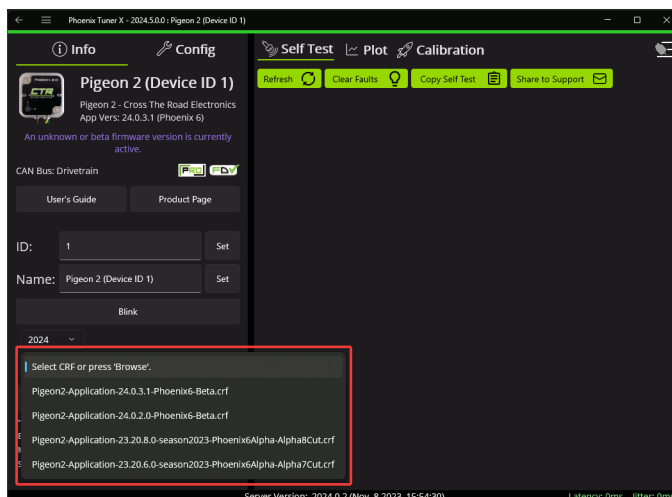
All devices can have their Name and ID configured via their respective textbox. IDs are limited to the range of 0 to 62 (inclusive). After inputting the ID or name, press the *Set* button to save the changes to the device.



Field-Upgrade Firmware Version

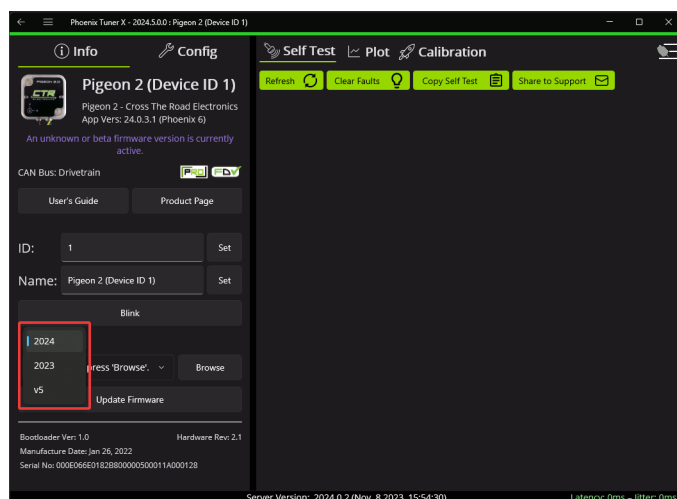
Tuner X has improved firmware upgrading functionality by **automatically downloading and caching** firmware. Upon initial Tuner X launch, the latest firmware for all devices will automatically be downloaded in the background (takes <10s on most internet connections). The individual device page allows 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 firmware release years by selecting from the dropdown above the firmware selection.

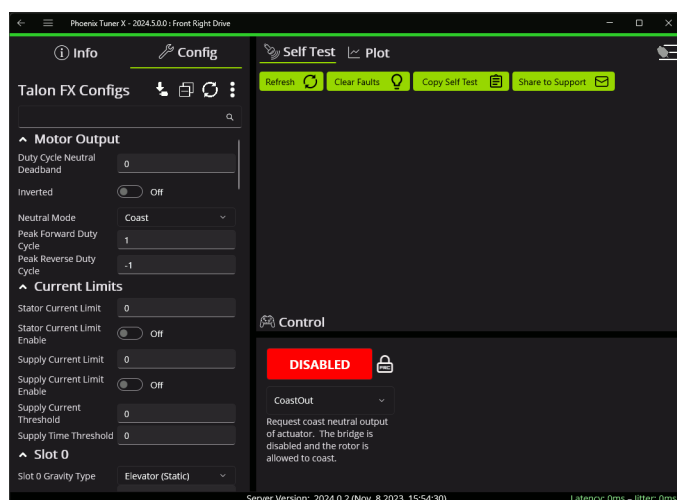
Note: The toggle between firmware years only affects the firmware versions downloaded by Tuner X. Files selected using the “Browse” button are not affected.



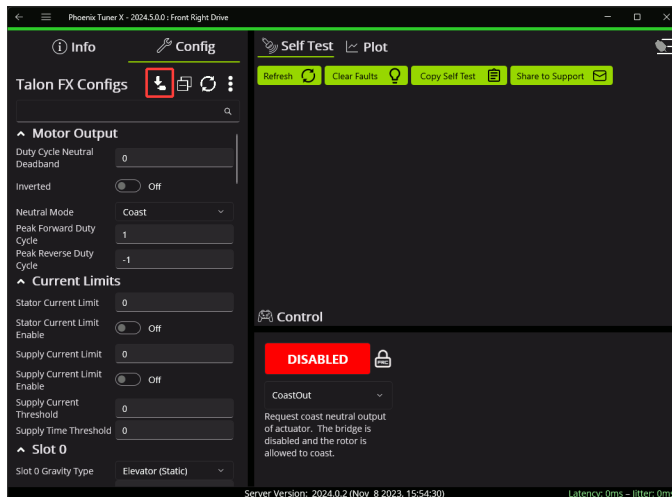
7.1.5 Tuner Configs

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

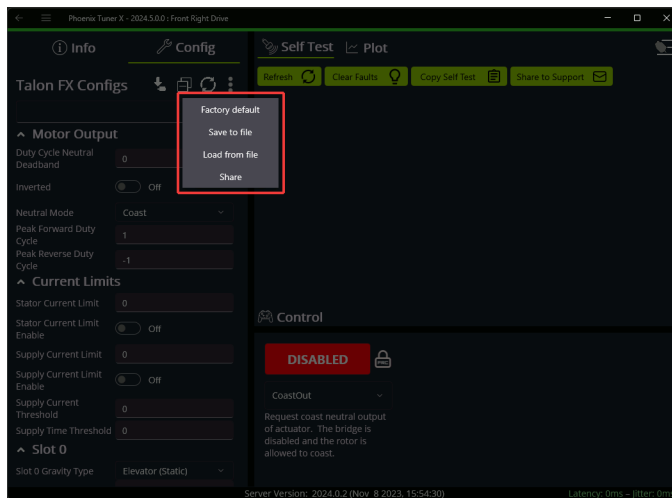
Configs can be viewed, modified, backed-up, restored, and factory-defaulted via the *Configs* tab in Phoenix Tuner X.



To apply a modified config, press the apply button (download icon) on the top button bar.

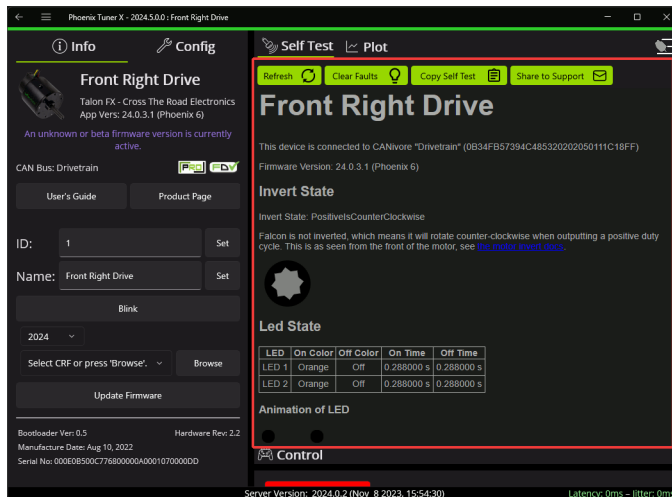


Additional options are available by clicking on the 3-dots icon on the top button bar.



7.1.6 Self Test Snapshot

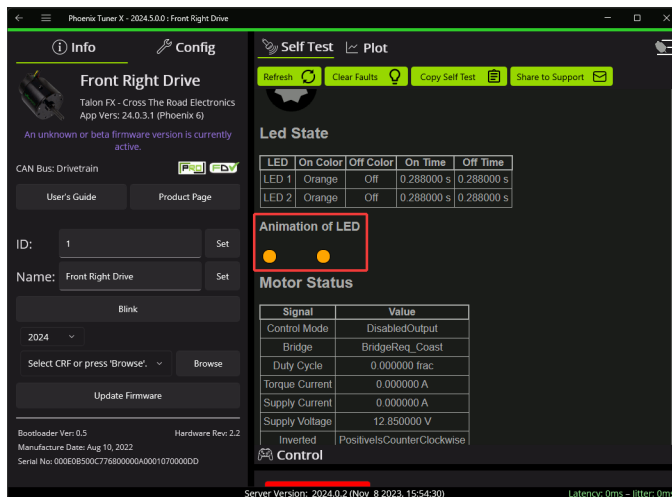
SelfTest Snapshot is a diagnostic feature of all supported devices that will show the immediate state of the device. This is extremely useful for troubleshooting and ensuring the device is working properly. Phoenix 6 with Phoenix Tuner X improves upon Self Test by showing the information in clean tables,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



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

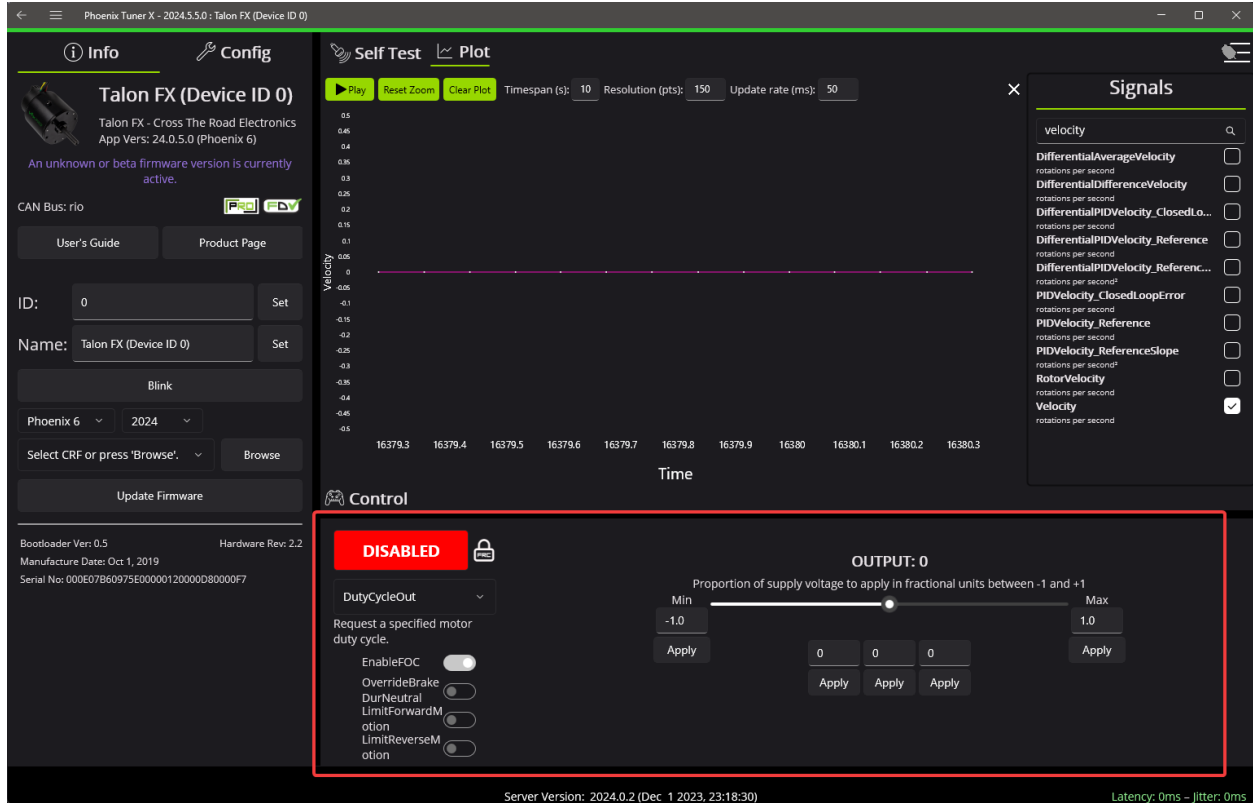
Viewing Status LEDs

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.



7.1.7 Controlling Devices

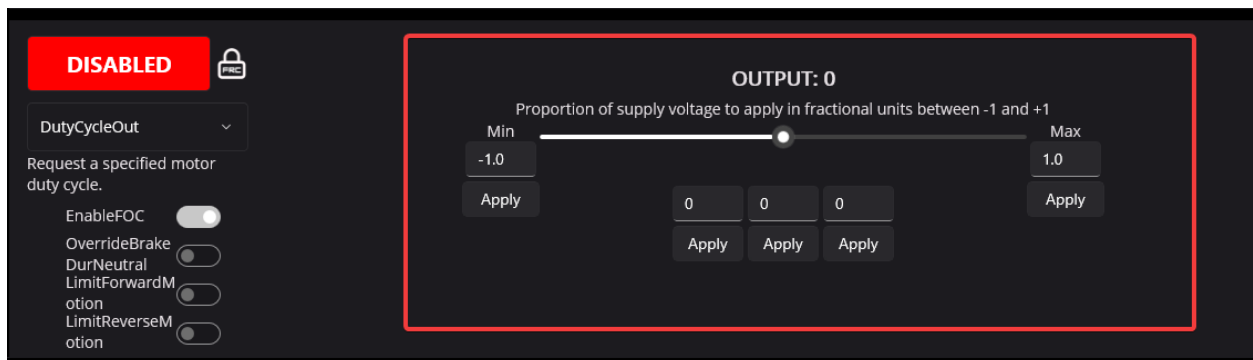
Tuner X can be used to directly control devices outside a robot program. When combined with *Plotting*, it can be an excellent tool for calculating closed loop gains or isolating mechanical issues.



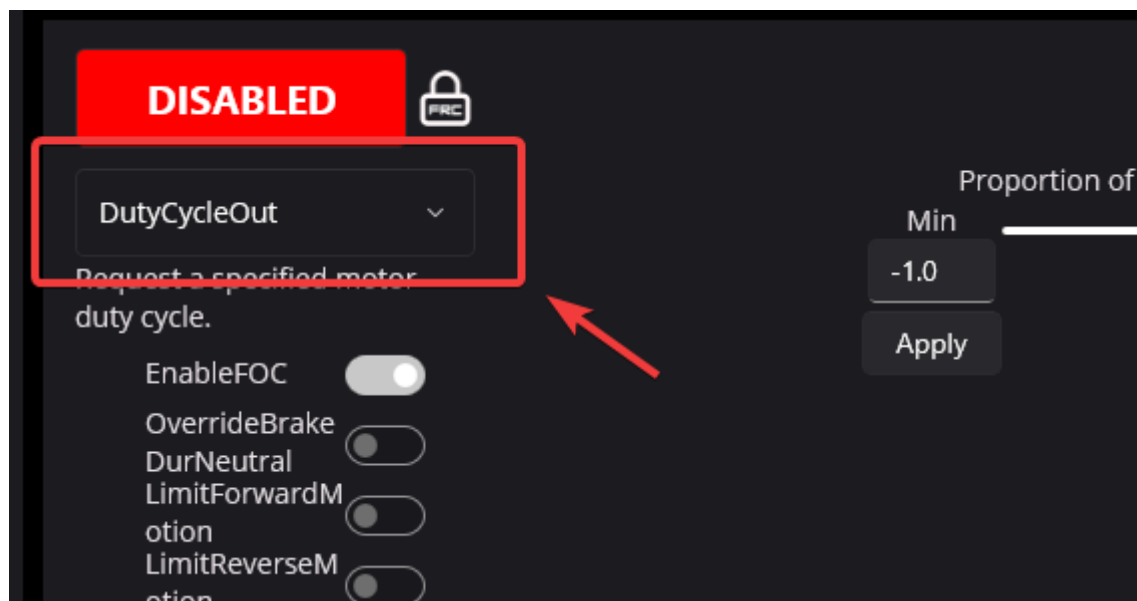
Devices can be controlled by clicking on the red “DISABLED” button, switching it to “ENABLED”.

Important: FRC users must enable the robot in Driver Station while using Tuner X control.

During this time, the output can be adjusted using the sliders or the text entries below it.

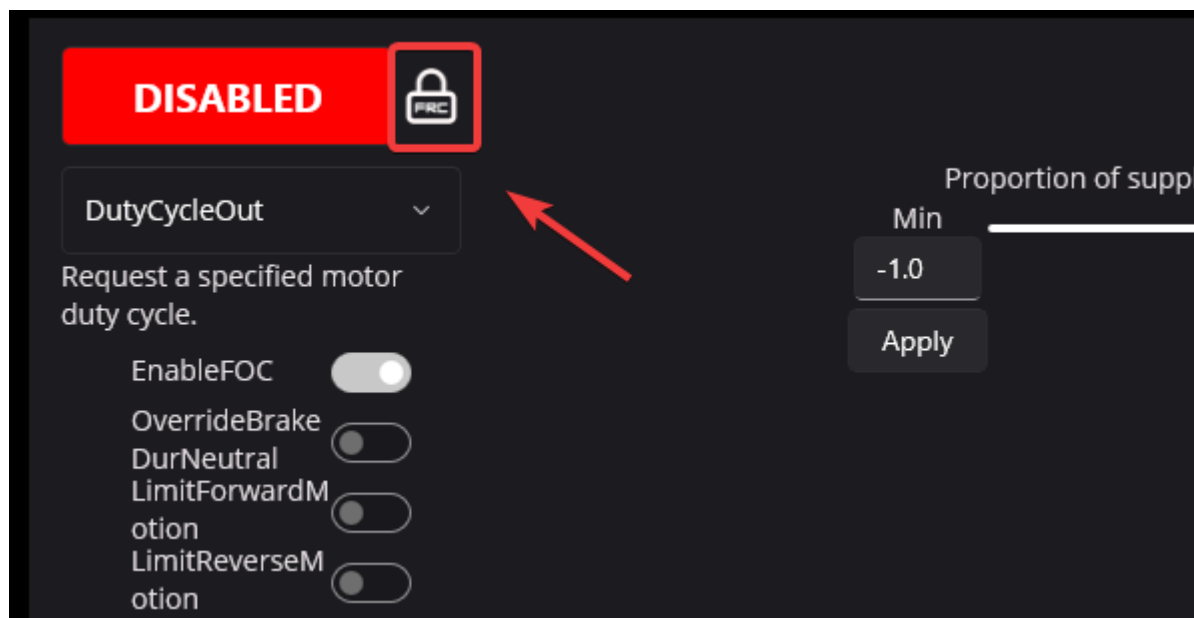


Control modes can be changed using the dropdown below the disable/enable button.



FRC Locked

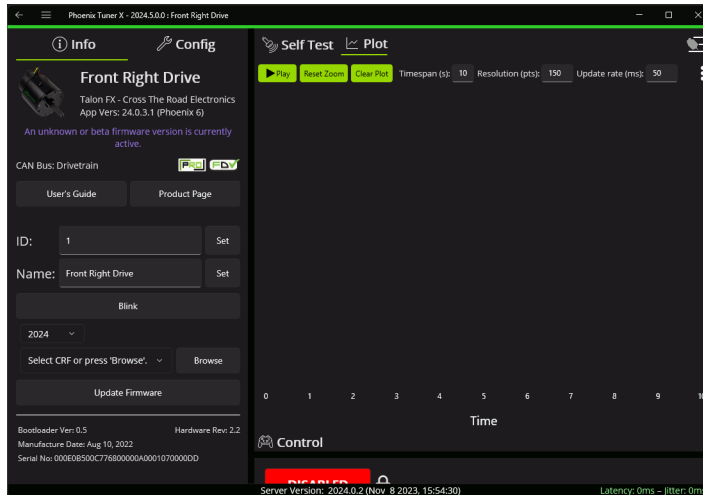
The “lock” icon next to the “DISABLED” button indicates that this device is FRC locked. This means the FRC Driver Station **must** also be enabled for the device to actuate. For more information, see [FRC Lock](#).



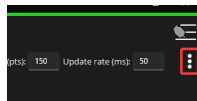
7.1.8 Plotting

Supported devices can have certain signals/sensor data plotted in real-time without any additional configuration.

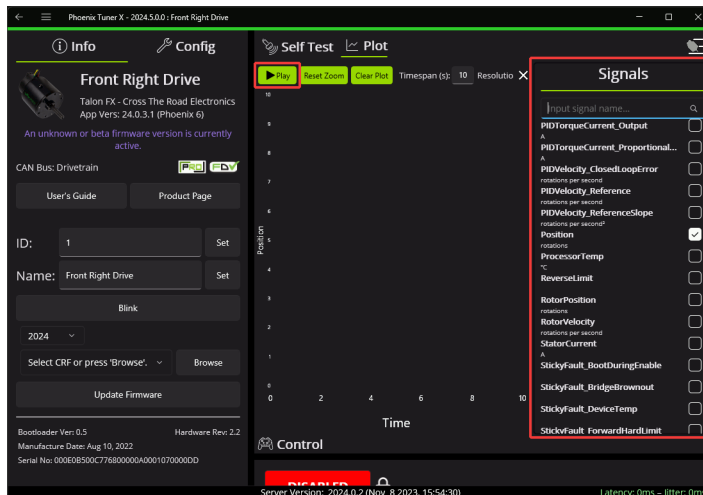
Tip: Plotting is supported in both Phoenix 5 and Phoenix 6.



If the list of signals is not visible, the 3-dots button at the top right of the page can be used to open them.

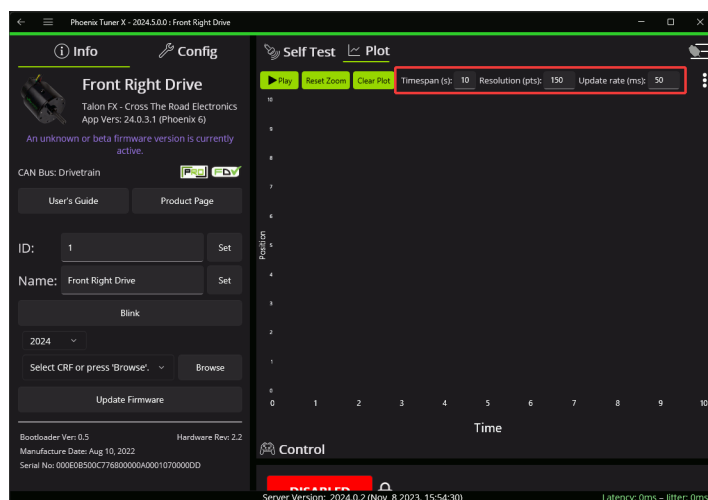


From the list on the right of the page, select the signals that you wish to plot. Then click the *Play* button at the top left.



Adjusting Plotting Settings

Plotting time period (the time frame that points are recorded) can be adjusted using the *Timespan* textbox. Additionally, the resolution (number of points shown) can be adjusted using the *Resolution* textbox. The rate at which the plot fetches updates from the robot program can be adjusted using the *Update rate* textbox.



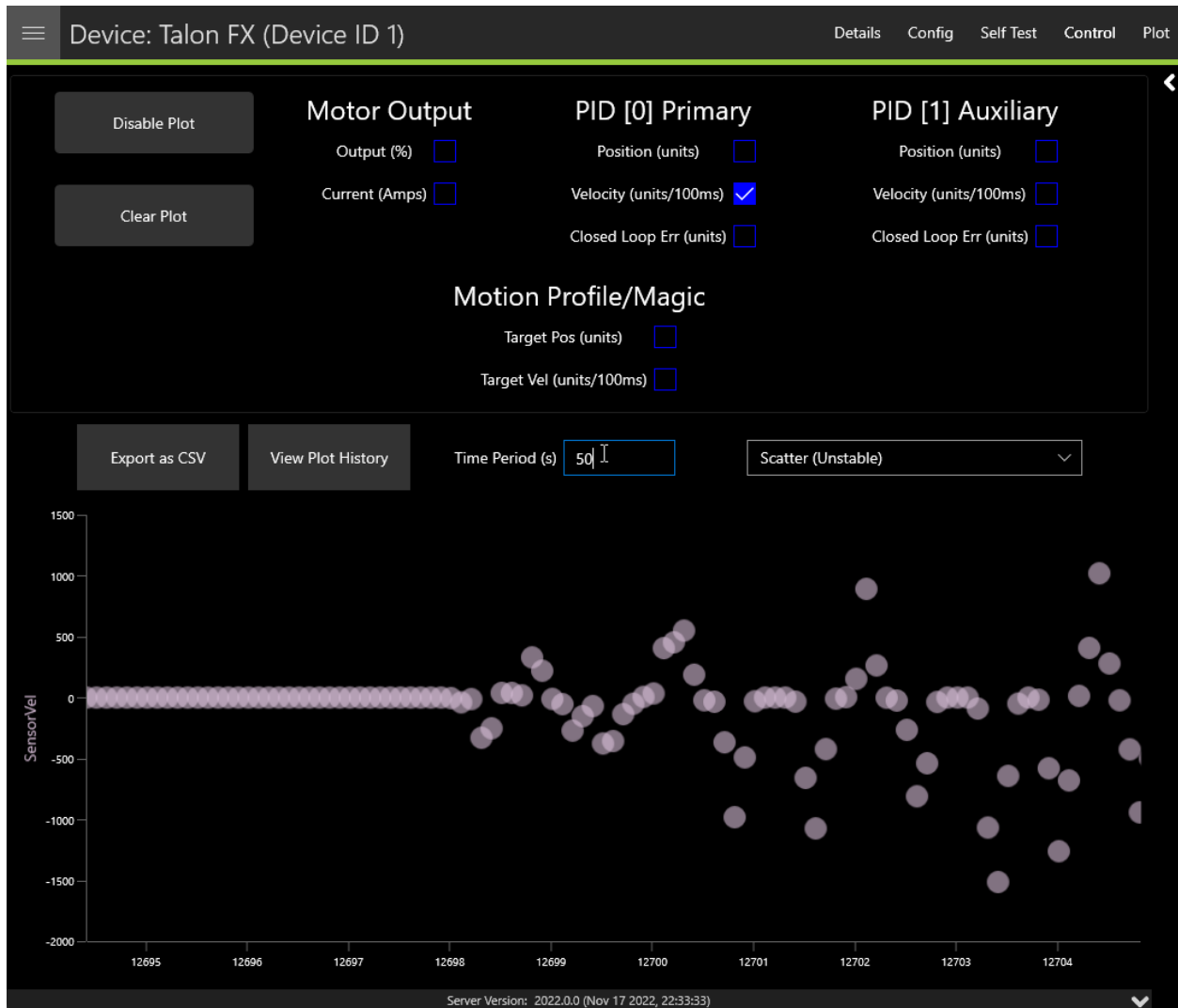
Exporting Data

Plots can be exported into the 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.

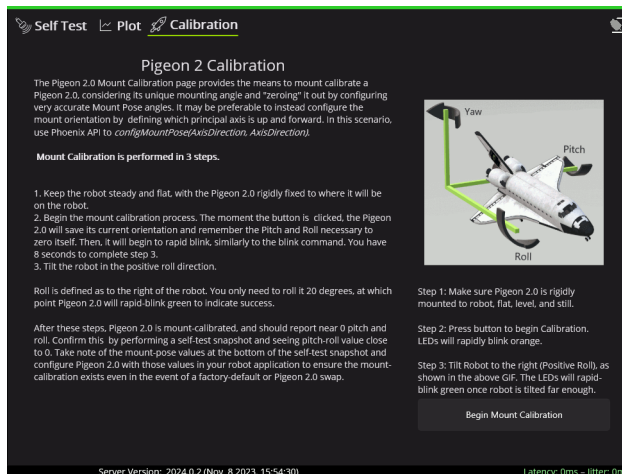
Plotting supports zoom and panning via the mouse and scroll wheel (or via gestures on Android). The point appearance can also be adjusted between “Spline” and “Scatter”.



7.1.9 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*.

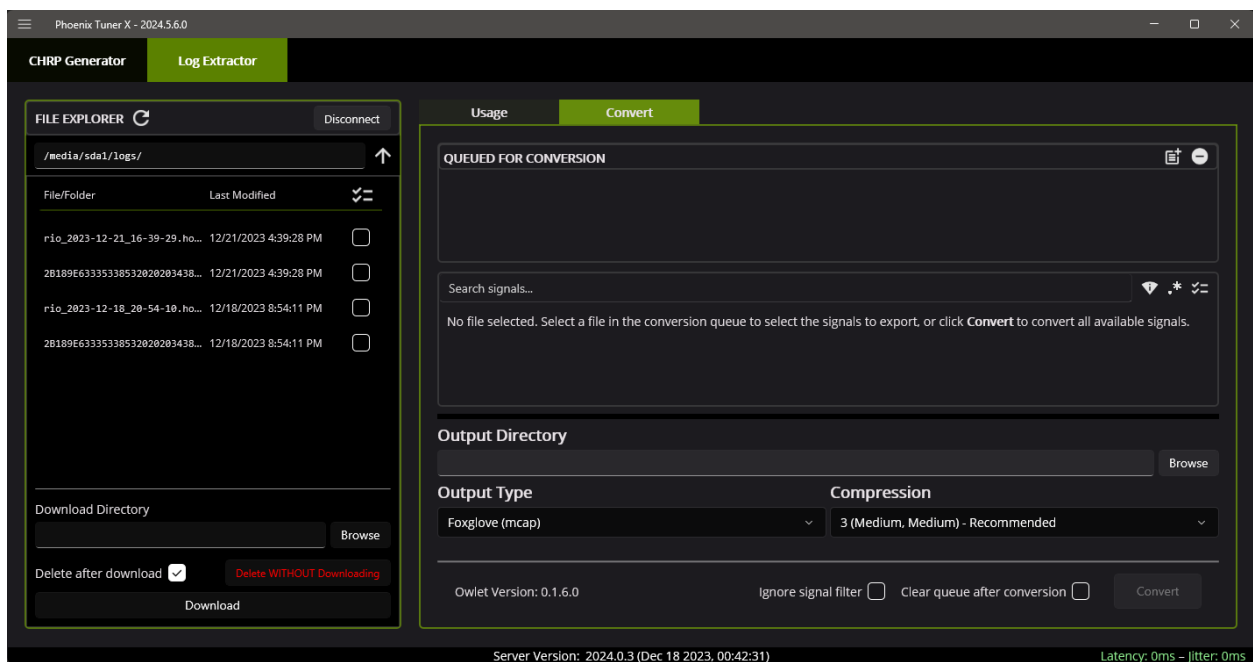


7.1.10 Tools

Tuner offers additional, miscellaneous functionality in the form of tool pages.

Extracting Signal Logs

Tip: Information on how to use the signal logger API can be found in the [corresponding API article](#). Tuner offers in-app functionality to retrieve, manage, and convert hoot logs to compatible formats.



CTRE hoot logs can be retrieved utilizing the file explorer on the left of the application. The file explorer offers the functionality to download and delete logs on a remote target.

On the right side, the *Convert* tab can be used to import and convert hoot logs to available formats.

Once a hoot log has been downloaded, it is automatically placed in the conversion queue to the right. Logs can also be manually imported or removed using the two buttons at the top-right of the conversion queue.



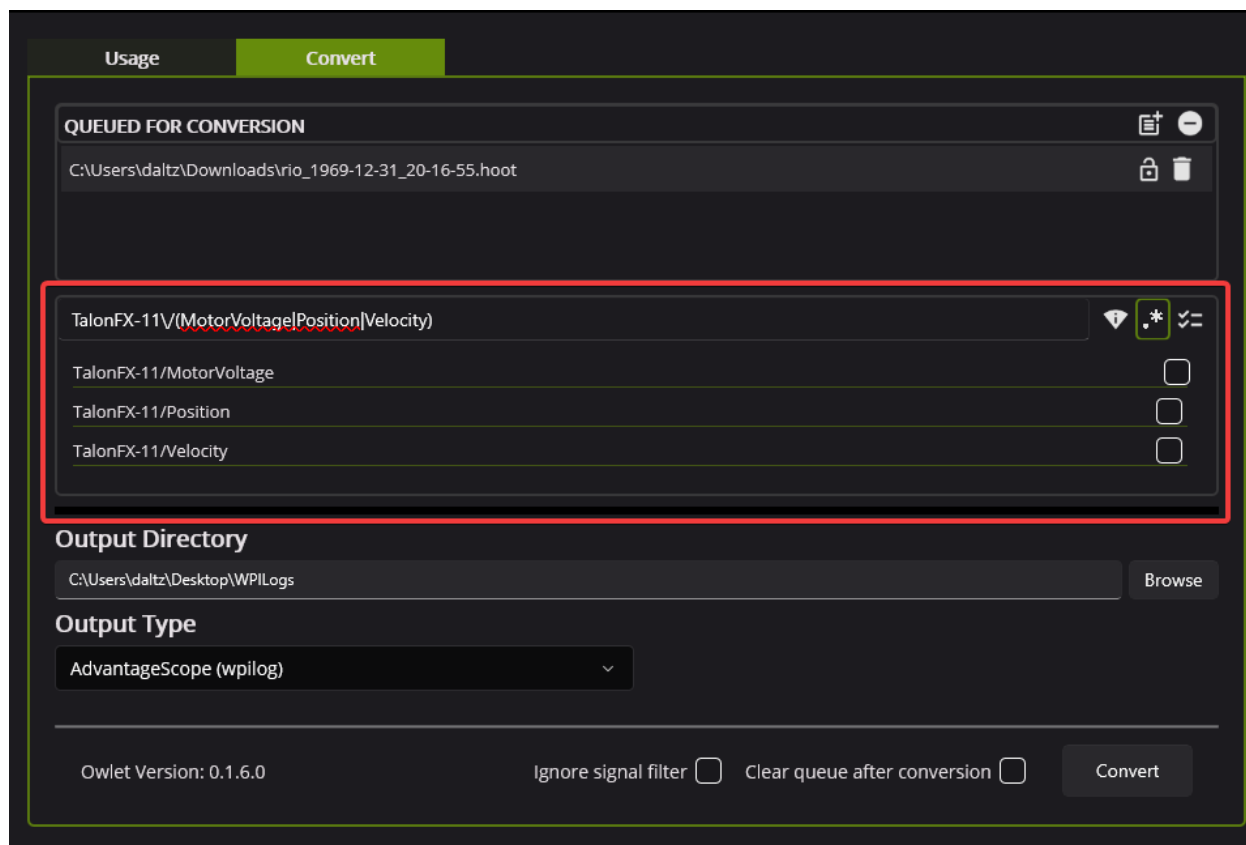
If a log file is reported as unlicensed, users can perform a *Deep Scan* of the log if they believe it should contain pro-licensed devices. By default, Tuner will only scan the first few megabytes of the log for pro-licensed devices to save time during scan and export.

Filtering for Signals

Since hoot logs can contain a massive amount of data, users may want to trim the exported log file. Tuner supports simple search and regex filtering of signals in a hoot. Filters are optional and configured on a per-log basis.

Note: If no signals are selected, all signals will be exported during conversion.

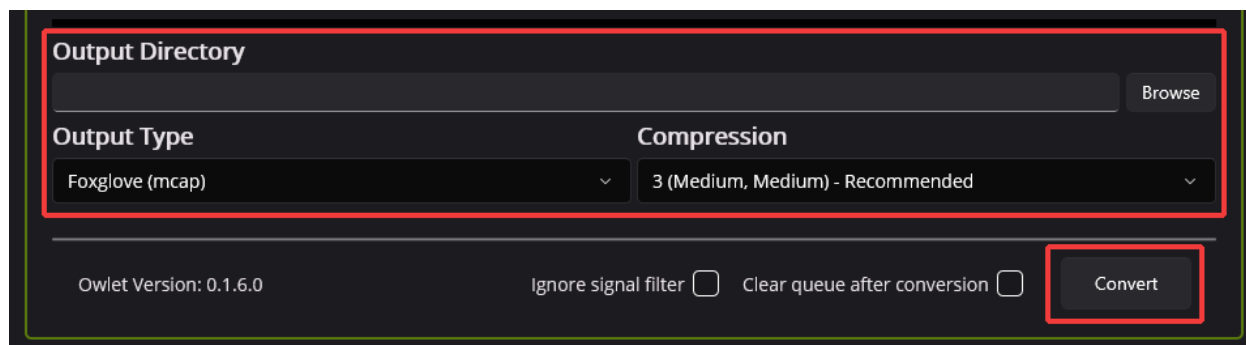
In the below picture, regex is used to select only the MotorVoltage, Position, and Velocity signals for TalonFX-11.



Important: If the hoot log does not contain any Pro-licensed devices, a *limited set of signals* may be exported for free using the *AdvantageScope (wpilog)* option.

Converting

After adding hoot logs to the queue, select the output directory and one of the output types. Then, click the *Convert* button to begin the conversion process. This may take some time depending on the output format, the size of the hoot file, and the number of selected signals.



Common Issues

Problem: When converting, I get hoot log API version too old, cannot export its signals

Solution: This may happen if your hoot file was generated using an old version of Phoenix. Update your Phoenix installation (by updating your vendordep in *Installing Phoenix 6*) and recreate your log file. If the log file recorded is critical, reach out to support@ctr-electronics.com.

Problem: When converting, I get Could not read to end of input file

Solution: This occurs when the converter encounters bad data. This typically occurs when the robot is turned off in the middle of writing to the log. Users can often ignore this error message, although the last few seconds of data may be lost. To avoid this issue, run `SignalLogger.stop()` before shutting down the robot program.

CHRP Converter

Orchestra uses CHRP files to play music using compatible Talon FX motors. Tuner offers the ability to convert **MIDI** soundtracks to compatible CHRPs. Simply follow the on-screen instructions and press *Import MIDI*.

Music Chirp Generator

Phoenix Tuner X provides a feature to convert MIDI files to the Phoenix-compliant Chirp (.chrp) file format. This file format is supported in the Orchestra class in Phoenix-API and allows you to play music through your Talon FXs. You do not need to be connected to the diagnostic server to use this page.

This feature can be used to add audible feedback to your robot. For example, you can play a pass or fail melody during self-tests or while debugging a problem. This is also great for fund-raising events or entertaining your team sponsors. The general steps are to:

1. Create a chirp file from your favorite MIDI.
2. Move the chirp file to the robot controller.*
3. Use Phoenix API to create an Orchestra object.
4. Call the loadMusic routine to load the chirp file.
5. Add TalonFX's to the Orchestra object. Each TalonFX is a unique instrument.
6. Call the play/pause/stop routines.

*roboRIO developers who use Java/C++ can generate the chirp file into the src/main/deploy folder of your VS project, then re-deploy your application. Note that robot must be enabled to play music.

Additionally, calling the set routine of a Talon FX while playing music will cause the Orchestra to pause.

Import MIDI

7.1.11 Swerve Project Generator

Important: Swerve functionality is only available for FRC users.

Under the *Mechanisms* page in Tuner X is the Swerve Project Generator. This utility guides the user through configuring their modules, verifying their drivetrain, encoder inverts, drivetrain inverts and more.

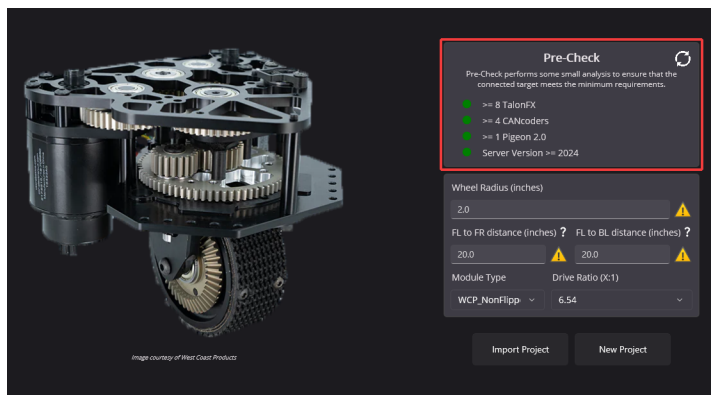
Note: The generated swerve project utilizes the [Swerve API](#).

Swerve Requirements

The swerve project creator and *swerve API* have several limitations. These limitations are in place to maximize performance and improve maintainability.

- Only Phoenix 6 supported hardware (e.g. TalonFX, CANcoder, Pigeon 2.0).
 - 8 TalonFX (4 steer, 4 drive)
 - 4 CANcoder (4 steer)
 - 1 Pigeon 2.0
- Requires Phoenix 6 software released for year 2024 or newer.
 - Firmware version should begin with 24.X
 - Ensure Tuner shows server version as “2024” or newer
- Temporary diagnostic server (or an existing 2024 robot project) must be running and Tuner should be connected to the robot.
 - This allows the generator to perform it’s setup and auto-calibration routines

Some of these requirements are enforced via a mechanism called “precheck”. You can see and refresh the precheck status by clicking the *Refresh* button in the top right.



Note: While the Swerve API and project generator can be utilized without Pro or FD, both of these enhance robot control. When utilizing Pro and CAN FD, sensor data is acquired synchronously. *FusedCANcoder* improves the accuracy of module positions.

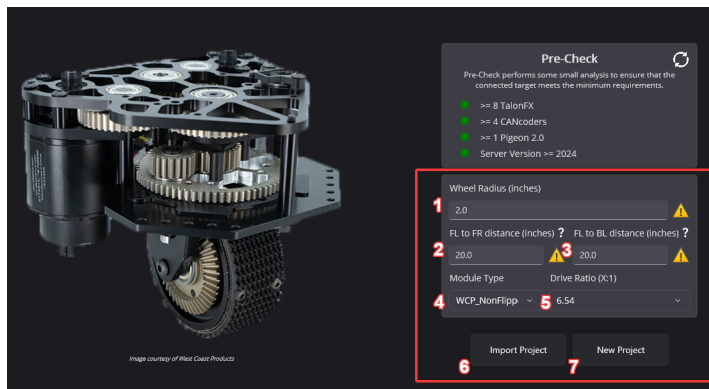
Requirement Checklist

Users can utilize the below checklist to ensure their robot is ready for project generation.

Requirement	Done?
REQUIRED: Is there the minimum number of devices? (8 TalonFX, 4 CANcoder, 1 Pigeon 2.0)	
REQUIRED: Do all devices appear in Tuner X?	
REQUIRED: Is all firmware up-to-date? (24.X)	
REQUIRED: Is 2024 (or newer) diagnostics running?	
Recommended: Have devices been renamed? (e.g. "TalonFX (Device ID 1)" -> "FL Steer Motor")	
Recommended: Are the devices on a CANivore?	
Recommended: Are the devices Pro-licensed?	

Once the user has reviewed the requirements, continue to *Creating your Project* to get started.

Creating your Project



1. Wheel Radius (inches)

- The radius can be found by measuring the width of the module wheel, then dividing that by 2.

2. FL to FR distance (inches)

- This is the distance between the center of the front-left module, and the center of the front-right module.

3. FL to BL distance (inches)

- This is the distance between the center of the front-left module, and the center of the back-right module.

4. Module Type

- The type of swerve module, such as WCP Swerve X standard, flipped gear, or flipped belt. Users not using any of the supported modules should select *Custom* instead.

5. Drive Ratio

- This is the gearing ratio between the output shaft of the motor and the module wheel. Swerve X users can find that information [here](#).
- Steer Ratio (Custom)

- This is the gearing ratio between the output shaft of the steering motor and the azimuth gear. For the *Custom* module type, users must calculate this based on their gearing themselves, or consult their manufacturer.

6. Import Project

- Import an existing Tuner X swerve project save file.

7. New Project

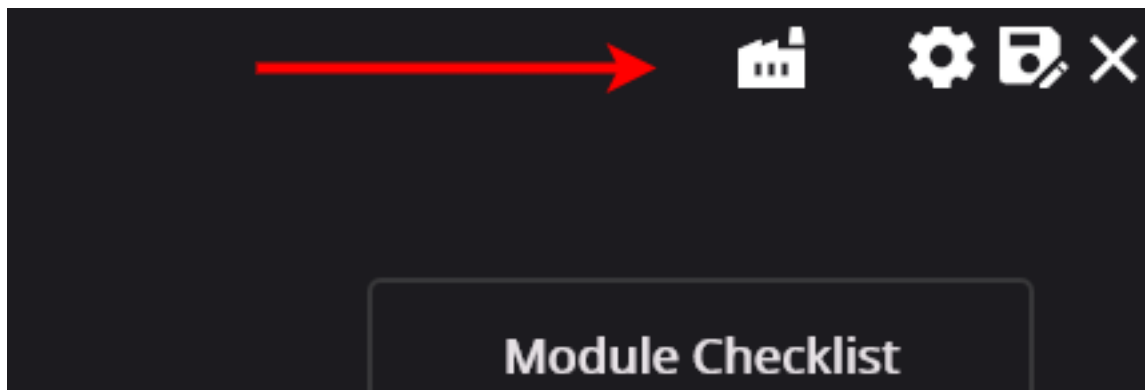
- Create a new project based on the settings configured.

Users should configure the settings applicable for their robot and click *New Project* once they are done.

Tip: Throughout the application is various tooltips, that when you hover on them, provide instructions. If you are unsure on what something means, try hovering on it!

Wizard Options

Once a project is open, a couple of options are exposed at the top-right.



In order from left to right:

- Factory default all devices
- Open the swerve settings menu
- Export project
- Exit project

Configuring Modules

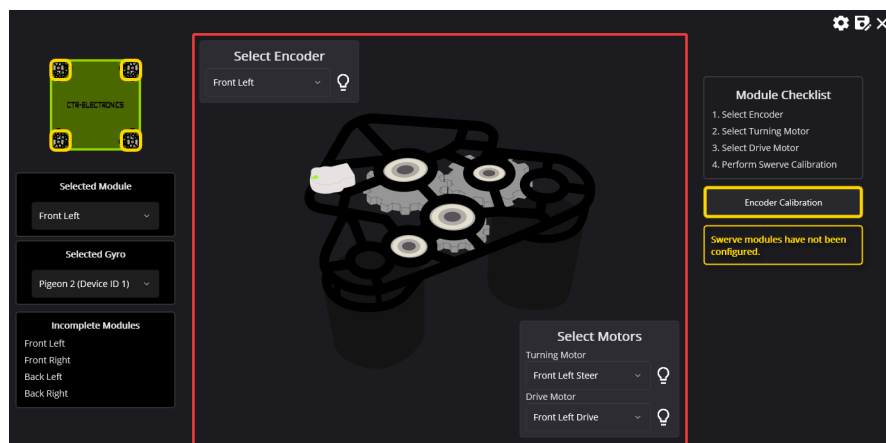
Module configuration is comprised of 4 steps.

1. Selecting encoder
2. Selecting steer/turning motor
3. Selecting drive motor
4. Performing encoder calibration

Selecting Module Devices

The first step is to select the encoder and motor controllers for the selected swerve module. In the screenshot below, **Talon FX (Device ID 0)** and **Talon FX (Device ID 1)** are selected as the steer and drive motor controllers.

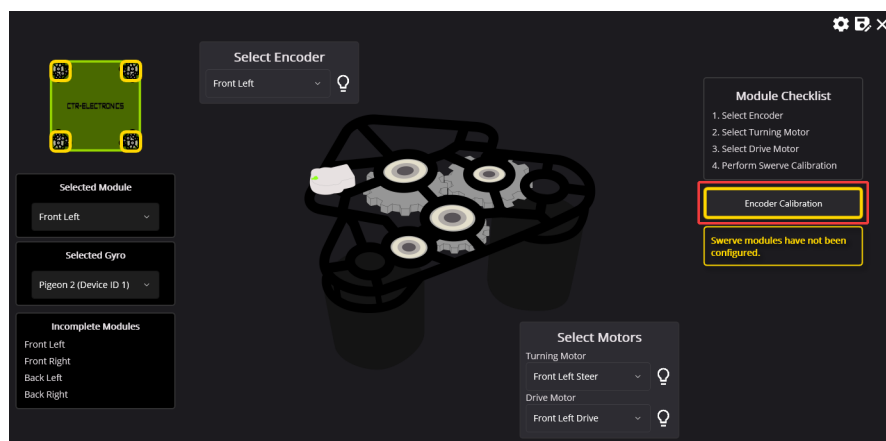
Tip: If a warning icon shows where the *Blink* icon is, this typically means this device has already been assigned to a module.



Encoder Calibration

Once a module's devices have been selected, an *Encoder Calibration* button will appear on the right-side of the screen. This will open a popup that guides the user through aligning their module and calibrating their CANcoder offsets.

Important: It's extremely important for the modules to be aligned such that the bevel gear faces the **vertical center** of the robot. Failure to perform this step may lead the drive verification tests to fail.



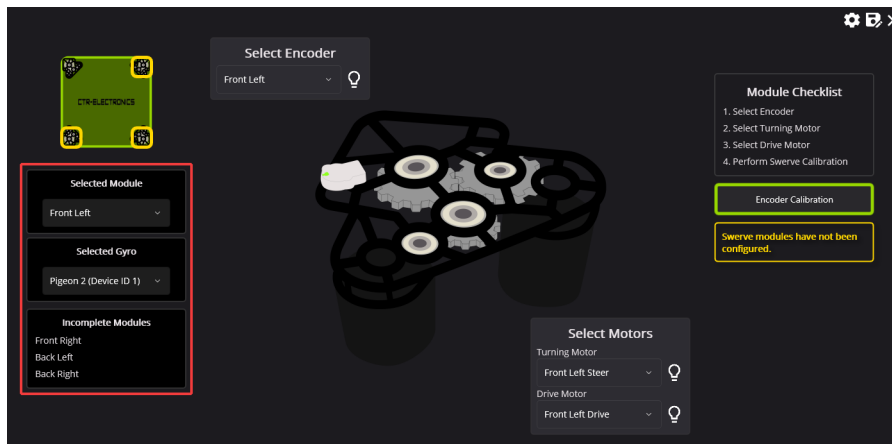
Once calibration has been completed, the module will be removed from the **Incomplete Modules** list on the left.

Switching between Modules

Modules can be swapped by selecting the module in the dropdown located on the left-hand side of the wizard.

Alternatively, a module can be selected by clicking on one of the yellow squares on the swerve bot in the top-left corner.

Warning: If an encoder has been reassigned to a new module, users should perform encoder calibration. Failure to perform encoder calibration will lead to unexpected module behavior.



Once all modules have been configured, click on the *Configuration Completed!* button that appears.

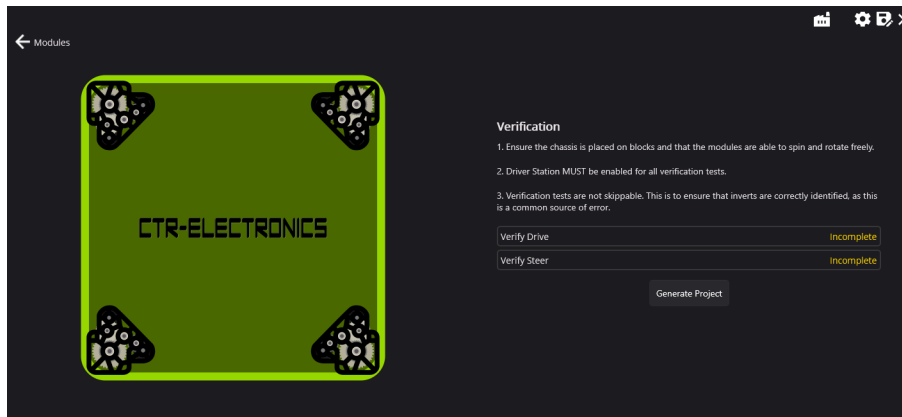
Warning: To ensure the drive and steer tests are accurate, the devices are factory defaulted. It's recommended that the user makes a *backup* of their configs if they are not applied in their robot program.

Validating the Drivetrain

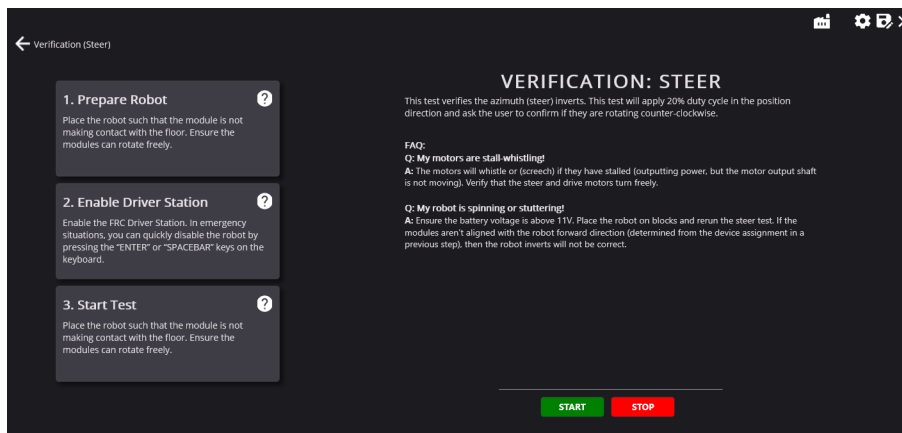
Tuner performs several validation procedures to confirm inverts, offsets and mechanical functionality of the drivetrain.

This test consists of two steps:

- Verify Steer
- Verify Drive



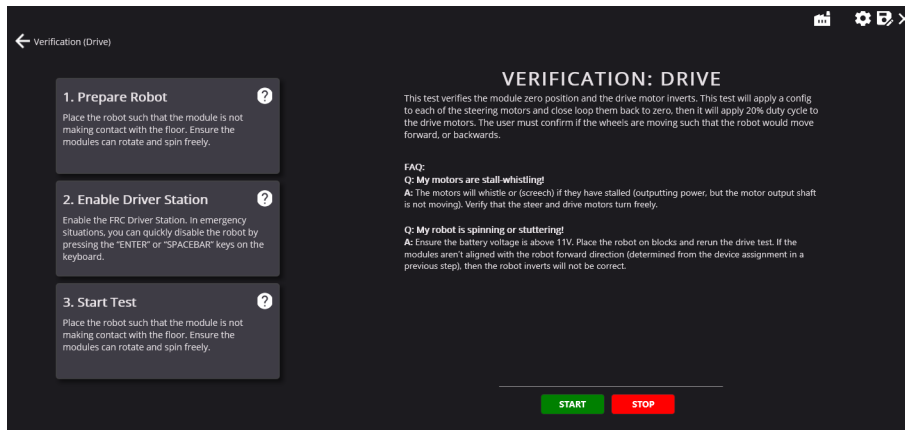
Verify Steer



The **Verify Steer** test is used to confirm that the module azimuth (or steer) can rotate freely and what direction they rotate in.

This test simply rotates all of the modules and confirms with the user if they rotated clockwise or counter-clockwise. The modules should rotate **counter-clockwise** (looking down at the module).

Verify Drive



The **Verify Drive** test is used to determine if the offsets were correctly applied and what the inverts of the drive motors should be. This process is as listed:

1. Rotate the modules until they are at position 0 (the module 0 should be calibrated from the *calibration* step).
2. Apply ~10% dutycycle.
3. Confirm with the user if the robot would've moved forward or backwards.

Without placing the robot on the ground, forward can be determined with the following steps.

1. Look at the robot from the right side of the robot.
2. Observe the wheels rotating. If the wheels rotate **clockwise**, this is forward (for the right side of the robot!).

Repeat for the left side of the robot, but instead of clockwise, forward is **counter-clockwise**.

Troubleshooting

A couple of common troubleshooting steps are listed below.

Q: The left or right-side of the drivetrain rotates in the wrong direction!

Note: If **both** sides are rotating in the incorrect direction, simply select *No* in the prompt that appears at the end of the test.

Answer: Go back to the configuration page and reperform CANcoder calibration. Ensure that the bevel gear of the robot is facing toward the vertical center of the robot and that the module is centered (either with a locking pin or ruler).

Q: The robot wheels did not move!

Answer: Ensure that the robot is enabled in the [FRC Driver Station](#).

Q: The robot says the average velocity was too small!

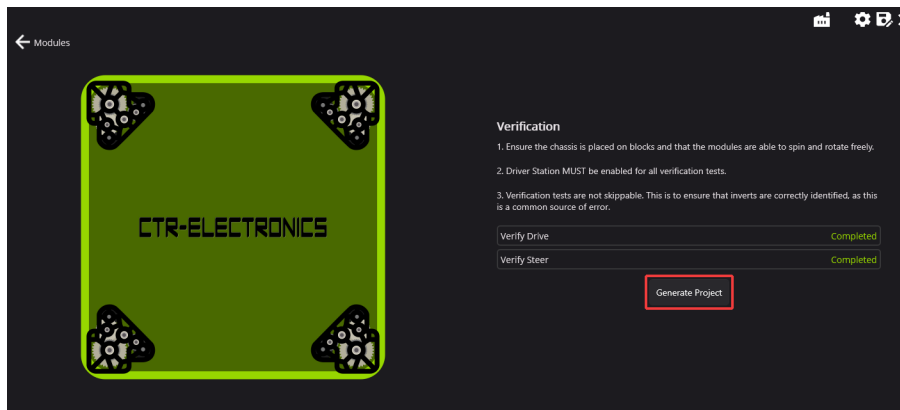
Answer: Confirm what direction the wheel is rotating. Same solution as *Q: The left or right-side of the drivetrain rotates in the wrong direction!*.

Q: My motors are stall-whistling

Answer: The motors will whistle (or screech) if they have stalled (outputting power, but the motor output shaft can't move). Verify by hand that the steer and drive motors can turn freely.

Q: My robot is spinning or stuttering

Answer: Ensure the battery voltage is above 11V. Place the robot on blocks and rerun the steer test. If the modules aren't aligned the robot forward direction (determined from the device assignment in a previous step), then the robot inverts will not be correct.

Generating the Project

The robot program can be generated by clicking the *Generate Project* button. This will open a prompt asking what team number to generate a project for.

Tip: It's highly recommended to export the project settings using the Save As icon in the top-right. This will save time if the robot program needs to be regenerated for any reason.

Device Licensing

Note: Users utilizing season pass must attach a team number before continuing. See [Attaching a Team Number to Season Pass](#) for more information.

The following devices are eligible for single-device licensing:

- TalonFX (Falcon 500, Kraken X60)
- 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.

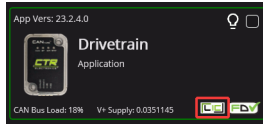
8.1 Purchasing a License

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.

8.2 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.

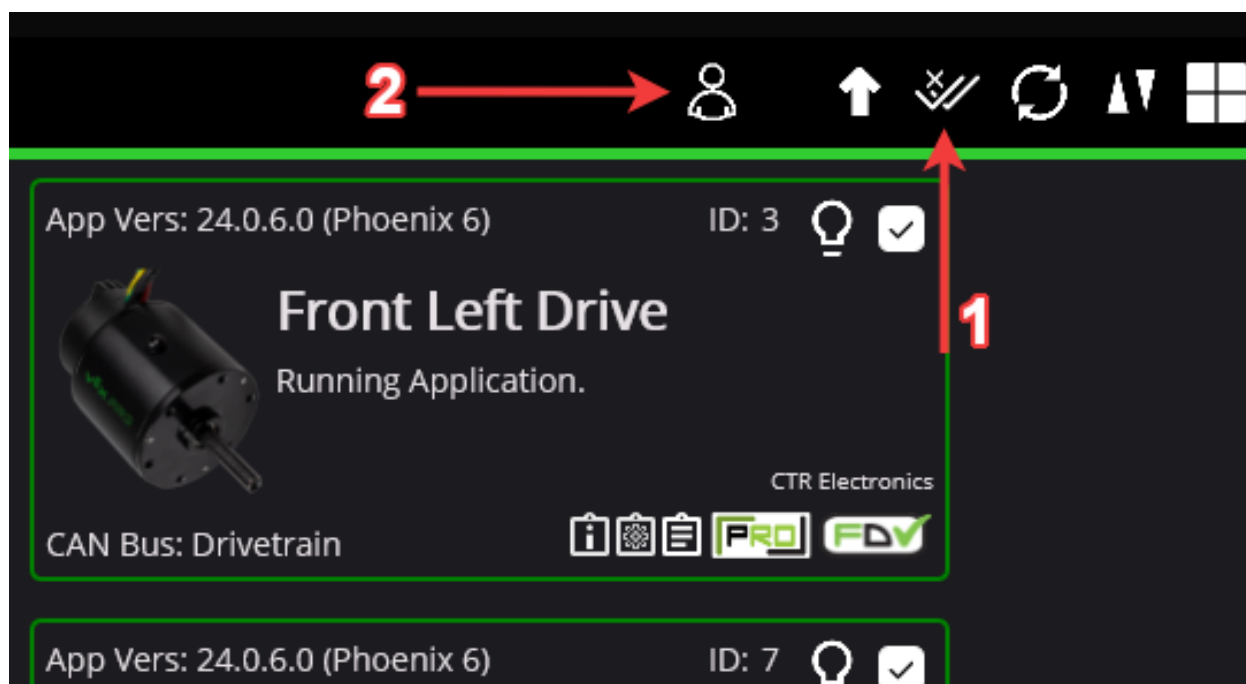
Warning: Users should be aware that license activation is permanent and irreversible

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.

8.2.1 Batch Activating Licenses

Tuner X also supports batch activating licenses 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.

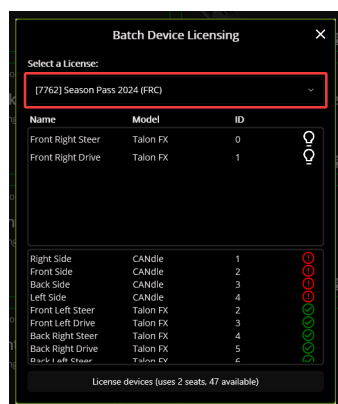
Tip: Selecting a device using their checkbox and clicking the checkmark in the top right will select all devices of the same models



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

Step 2 in the above image opens the batch licensing dialog.

Once the dialog is opened, select a license from the dropdown at the top of the popup.



The first list contains devices that will be batch licensed, while the second list contains devices that are ineligible due to one of the following:

- Device is not running Phoenix 6 firmware that supports licensing
- Device does not support the selected license
- Device is already licensed with the selected license

The *License devices* button at the bottom of the popup shows the number of device licenses that will be applied and the number of seats currently available. After confirming that everything looks correct, press the *License devices* button to apply the licenses.

8.2.2 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.

8.3 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 the Phoenix 6 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 the Phoenix 6 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.

8.4 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-6/Libs version.
 - Confirm the **vendordep** in your robot project is the latest version.
 - Alternatively, you can deploy the **temporary diagnostic server**.
- Is the latest **Phoenix 6** firmware flashed onto the device?

- **FRC Only:** If using Season Pass, is the roboRIO configured with the correct team number?

Season Pass Licensing

Tip: Season Pass can be purchased at <https://store.ctr-electronics.com/>.

Season Pass is a single, cost-effective purchase that allows **FRC teams** to license *compatible devices* with Phoenix Pro for the entire season. The steps for licensing devices with season pass are as follows.

1. Purchase a season pass at <https://store.ctr-electronics.com/>.
2. Wait for the email that says your license is ready (typically 1-2 minutes).
3. Follow the below steps for attaching a team number to your license.
4. Once a team number has been attached, licensing process is the same as *Device Licensing*.

9.1 Attaching a Team Number to Season Pass

Warning: Attaching a team number to a season pass is permanent.

1. On the “Profile” page of Tuner X, click on the license to attach a team for.

Logged in as: [redacted]

Activation

Team Licensing

Seat Information

Available Seats

Activated Licenses

Tip!

Licenses can be purchased from <https://ctr-electronics.com/>

Year	Description	Assigned Team
2023	Season Pass 2024 (2024 Alpha)	66
2024	Season Pass 2024 (FRC Beta)	470
2024	Season Pass 2024 (FRC Beta)	66
[redacted]	[redacted]	[redacted]

2. Enter the team number in the box below the list of licenses.
3. Click *Assign Team*. A prompt will appear asking the user to confirm the entered team number.

Note: Note that the robot must be configured for the assigned team number. An invalid team number on the robot will result in the device not appearing as Pro licensed.

9.2 How many devices can I activate?

A season pass contains 60 individual device licenses. In the event that a team needs more licenses, contact us at support@ctr-electronics.com.

Kraken X60

The Kraken X60 powered by Talon FX is a brushless motor developed by [WestCoast Products](#) that uses the latest BLDC motor control technology from CTR Electronics. The integrated Talon FX unlocks the full performance of the Kraken X60 BLDC motor while providing best-in-class motion control.

[Store Page](#) [CAD](#), [Firmware](#) and [purchase instructions](#).

[Hardware User Manual](#) [Wiring and mount instructions](#) available on the [WestCoast Products](#) documentation site.

Falcon 500

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.

10.1 Utilizing your TalonFX

10.1.1 Improving Performance with Current Limits

Current limiting is the process of restricting motor output when a given current has surpassed a limit. There are two types of current limits available: stator and supply. Each of these limits accomplishes different goals. This article goes over **why** current limiting is important, **when** to configure these limits, and **how** to configure them.

Note: By default, devices are not configured with any current limits. This is because the optimal limits depend on how the motor is integrated into the system. There are additional safety measures in place to prevent damage to the motor or motor controller under excessive load.

Stator and Supply Current Limits

It's important to understand the need for current limits and how they work. When a motor is under a load, it takes an increasing amount of current to continue rotating the shaft of the motor (and by extension the mechanism). The cumulative sum of all the currents in a multi-motor system, such as a robot, may excessively drain batteries, trigger brownout protection, or in the worst case trip breakers.

There are two forms of current limiting: stator and supply. The relevant [configs](#) are located in the CurrentLimits config group ([Java](#), [C++](#), [Python](#)). All current limits must be **separately enabled** using the appropriate enable config.

Warning: Current limits are not applied in torque based control modes, such as Torque-CurrentFOC. Limits for torque based control modes should be applied with PeakForwardTorqueCurrent and PeakReverseTorqueCurrent instead. Torque current is equivalent to stator current in magnitude.

Stator Limits

Important: Stator current limits are only applicable in non-torque control modes. Users utilizing torque based control modes should use the PeakForwardTorqueCurrent and PeakReverseTorqueCurrent configs instead, where the following documentation is still applicable.

Stator current is the output current of the motor and is directly proportional to torque.

Stator current limits are used to restrict the torque output of the motor. This can be used to prevent wheel slip or avoid damaging a mechanism when running into a hard stop. Stator current limits also reduce heat buildup in the motor.

Since stator current limits also limit supply current, they are also effective at preventing brownouts when accelerating.

Relationship between Supply and Stator Current

Note: This explanation ignores energy loss from heat or other inefficiencies.

By conservation of energy, power going into the motor must equal power out. Since power is equivalent to voltage times current, $V_{supply} * I_{supply} = V_{stator} * I_{stator}$, where V_{stator} is the output voltage of the motor. The duty cycle output of a motor is equivalent to V_{stator}/V_{supply} , so the relationship between supply and stator current can be described as $I_{supply} = I_{stator} * dutycycle$.

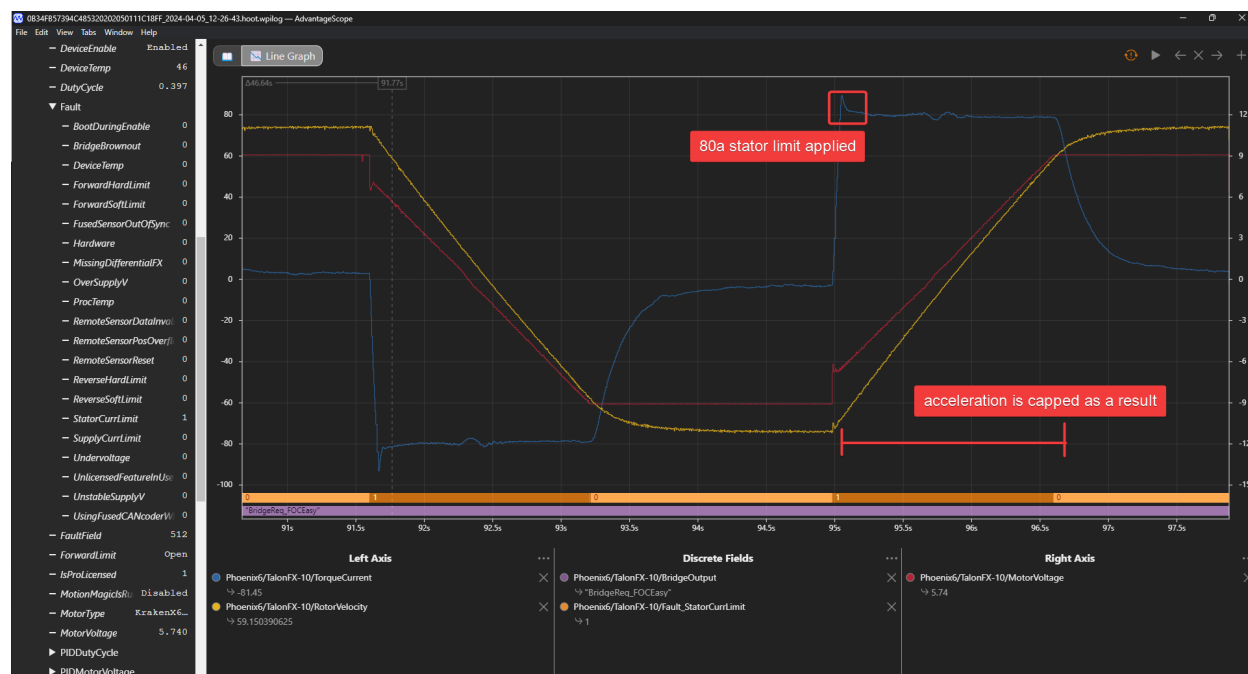
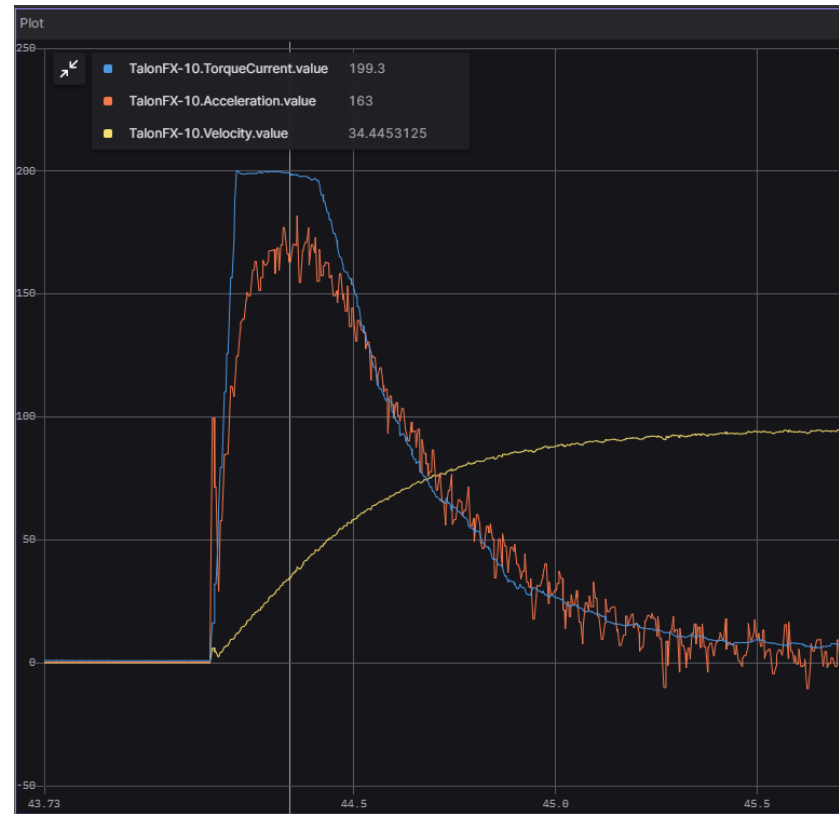


Fig. 1: 80 A stator current limit taking effect

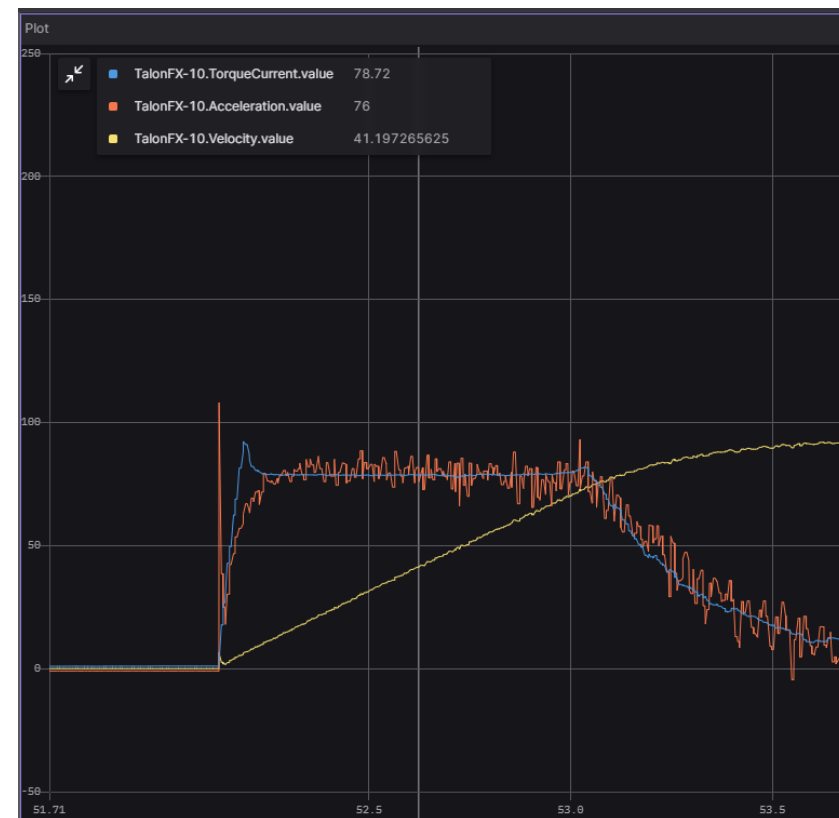
As an example, if a motor is applied 100% output (~12 V) and has 80 A of measured stator current, then the supply current will also be 80 A. However, if the motor is applied 50% output (~6 V) and has 80 A of measured stator current, then supply current will only be 40 A.

This means that stator current limits also effectively limit supply current. Supply current will not exceed a stator current limit and is often significantly lower than stator current.

Another impact that stator current limits have is that they restrict acceleration. Stator current limits cap the torque output of the motor, which directly increases the amount of time it takes to accelerate to max velocity. As a result, it is important not to set stator current limits too low, otherwise performance may be negatively impacted.



Without stator limit (~ 170 rotations/second²)



With 80 A stator limit (~ 75 rotations/second²)

Supply Limits

Important: Supply limits are not functional in torque based control modes. Use the `PeakForwardTorqueCurrent` and `PeakReverseTorqueCurrent` configs instead.

Supply current is the current drawn from the battery. As a result, limiting supply current can be useful to prevent breakers from tripping, as well as to improve the longevity of the battery.

Tip: Since stator current limits also limit supply current, it is often not necessary to enable both limits.

When the `SupplyCurrentThreshold` has elapsed for `SupplyTimeThreshold` amount of time, the supply current limiter will activate and reduce motor output until supply current is within range of the limit.

In the rare case where the robot experiences brownouts despite configuring stator current limits, a supply current limit can also further help avoid brownouts. However, such brownouts are most commonly caused by a bad battery or poor power wiring, so those should be examined first.

Determining Current Limits

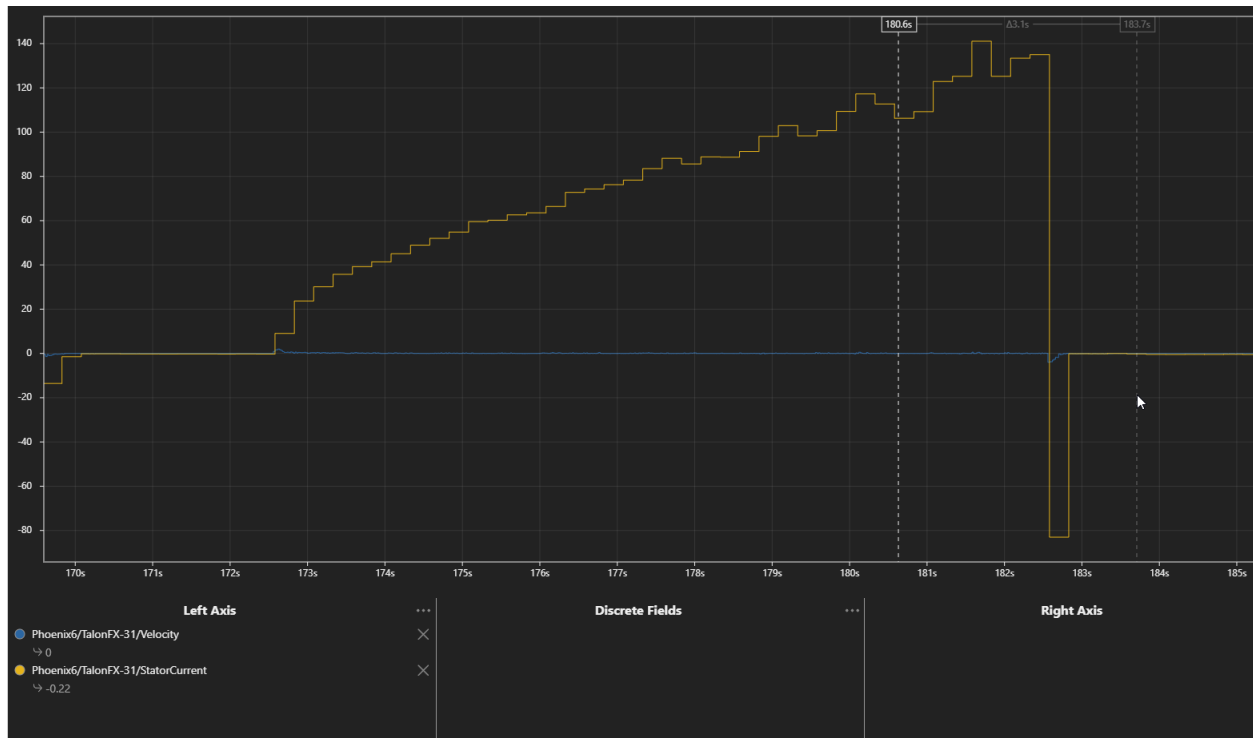
While supply current limits can be theoretically estimated by calculating max supply draw for every mechanism, stator limits are not easy to estimate. Determine your stator current limits first, then supply only if necessary. In many cases, stator current limits are sufficient to prevent battery brownouts, as stator current limits also limit supply current.

Preventing Wheel Slip

Stator current limits are excellent at preventing wheel slip, which is necessary to maximize power output of the system. To determine wheel slip, perform the following instructions:

1. Place the robot on carpet against a wall.
2. Begin plotting velocity and stator current in *Tuner X*.
3. Slowly increase voltage output until the velocity becomes non-zero and the stator current drops.

Set your stator current limit to a value below the observed stator current in Tuner. In the example below, the wheels began slipping at around 130 A.



Preventing Brownouts

Brownouts occur when the robot voltage dips below a threshold. For the [FRC roboRIO](#), this threshold is 6.3 V on the roboRIO 1 and 6.75 V (by default) on the roboRIO 2. When the roboRIO dips below the threshold, it will disable all actuators to prevent a total robot reboot.

Brownouts most commonly occur when the motor accelerates or is otherwise under high load (such as in a pushing match). Since stator current limits are highly effective at limiting supply current, especially at the start of acceleration, they are also highly effective at preventing brownouts in both of these scenarios.

For a few high-inertia mechanisms, such as some flywheels, supply current limits can also be used to further prevent brownouts during long periods of acceleration or high load. However, if a robot is still experiencing brownouts after configuring reasonable stator current limits, the robot should be checked for some common electrical issues before considering more restrictive current limits:

- Check the health of the battery, which can be done using a [Battery Beak](#) or by performing a full discharge test with a [battery analyzer](#).
- Make sure your battery leads are properly tightened to the battery, and the battery connector is properly crimped.
- If using the CTRE PDP, make sure the leads going into the PDP are properly crimped and secured.
- Check all device power connections at the PDP/PDH. The PDP/PDH should be clearly connected to copper and not wire insulation.

As motor supply currents increase, the battery voltage will decrease in a similar fashion. A simplified equation for modeling voltage sag is shown below along with a calculator. However,

it is easier and more accurate to empirically determine supply current limits. The process of empirically finding these limits is the same as in [Improving Battery Longevity](#).

$$V_{loaded} = V_{unloaded} - (I_{total\,supply} * R_{battery})$$

Be aware that battery health (in the form of battery resistance above) significantly impacts how large current draw affects the output voltage of the battery. Health of the battery can be roughly determined via a [Battery Beak](#) or by performing a battery discharge test with a [battery analyzer](#).

Improving Battery Longevity

While supply limits can be estimated using battery datasheets and average mechanism current draw, the easiest and most reliable method is to test, analyze, and reduce. By applying a conservative supply limit to mechanisms that can operate with minimal current draw, you can improve the performance of other mechanisms (e.g. reducing your intake current limits to increase the amount of current available for a swerve drivetrain).

1. Plot a mechanism's supply currents throughout the match.
2. Reduce the supply current limits until your performance begins to suffer (unable to intake items, etc.). Increase this value slightly for some comfort room.
3. Repeat this for all mechanisms on a fresh battery each time. Time how long it takes before brownouts are regular and battery capacity has dropped.

For example, a user may have the following mechanisms and current limits:

- x4 Kraken(s) on swerve drive - 120 A stator, 70 A supply
- x4 Kraken(s) on swerve azimuth - 40 A stator, no supply limit (<40 A)
- x1 Kraken(s) on elevator - 80 A stator, 30 A supply
- x1 Kraken(s) on intake - 20 A stator, no supply limit (<20 A)

This would yield a peak supply current of <490 A in the worst case scenario. However, this current draw is **extremely unlikely**. Stator current limits ensure all motors and mechanisms will not be under peak load at the same time. Peak current draw occurs towards the end of acceleration when stator current limits are enabled. Additionally, peak supply current is often extremely brief (>60 A on all 4 swerve drive motors lasts <0.5 seconds). A more common scenario is 4 swerve drive motors accelerating at the same time for a peak supply current of 280 A.

When determining supply current limits for a mechanism, ensure that its peak supply current and the duration of that peak does not cause any breakers to trip. FRC breakers typically trip from temperature and can sustain well beyond their rated amperage for a given amount of time. Consult the manufacturer datasheet for the breakers you use to see their trip times.

Reduce your current limits until your battery life is in an acceptable range.

How to Apply Current Limits

Current limits must be **enabled** and **configured**. This can be done in *Tuner X configs* or using the Phoenix 6 *configuration API*.

Java

```
var talonFXConfigurator = m_talonFX.getConfigurator();
var limitConfigs = new CurrentLimitConfigs();

// enable stator current limit
limitConfigs.StatorCurrentLimit = 120;
limitConfigs.StatorCurrentLimitEnable = true;

talonFXConfigurator.apply(limitConfigs);
```

C++

```
auto& talonFXConfigurator = m_talonFX.GetConfigurator();
configs::CurrentLimitConfigs limitConfigs{};

// enable stator current limit
limitConfigs.StatorCurrentLimit = 120;
limitConfigs.StatorCurrentLimitEnable = true;

talonFXConfigurator.Apply(limitConfigs);
```

Python

```
talonfx_configurator = self.talonfx.configurator
limit_configs = configs.CurrentLimitConfigs()

# enable stator current limit
limit_configs.stator_current_limit = 120
limit_configs.stator_current_limit_enable = true

talonfx_configurator.apply(limit_configs)
```

10.2 Actuator Limits

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

Note: The TalonFX + Kraken X60 does not support hardware limit switches. Instead, *control request limit* overrides can be used, or a CANcoder can be used a *remote limit switch*.

Documentation on retrieving and configuring limits can be found [here](#).

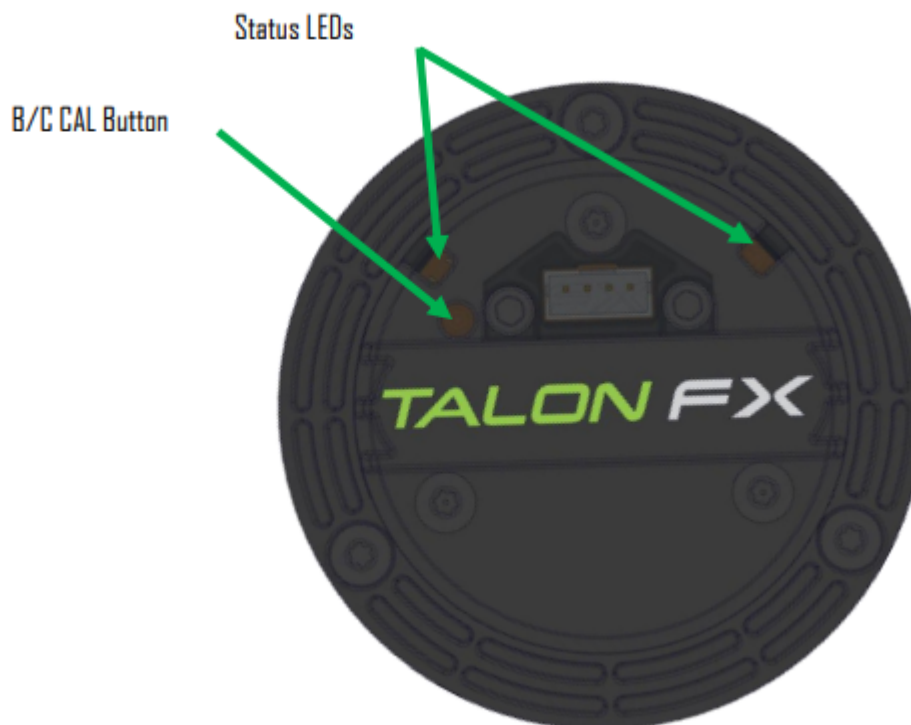
10.2.1 Limit Switches

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 in the Falcon 500, consult the [Falcon 500 User's Guide](#).

10.3 Status Light Reference



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.

11.1 Pigeon 2 Troubleshooting

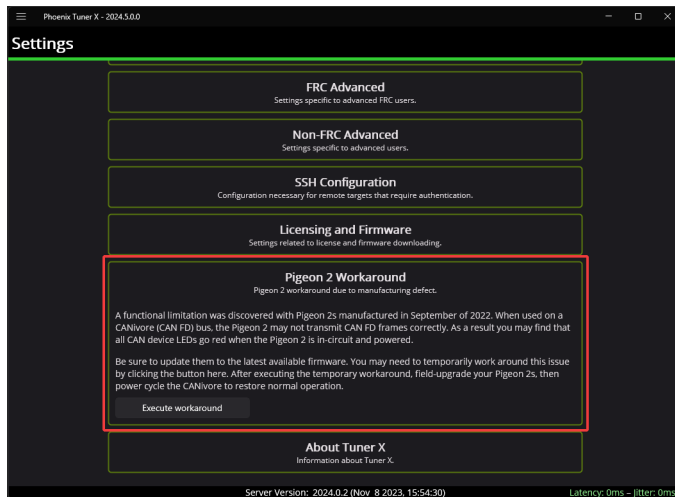
A functional limitation was discovered in Pigeon 2s manufactured in September of 2022. When used on a CANivore (CAN FD) Bus, the Pigeon 2 may not transmit CAN FD frames correctly. As a result, you may find that all CAN device LEDs go red when the Pigeon 2 is in-circuit and powered.

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

11.1.1 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.

11.1.2 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.

11.2 Status Light Reference



11.3 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.

Self Test
Plot
Calibration

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 `configMountPoseAxisDirection` and `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

Server Version: 2024.0.2 (Nov 8 2023, 15:54:30)
Latency: 0ms - Jitter: 0ms

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.

12.1 Status Light Reference

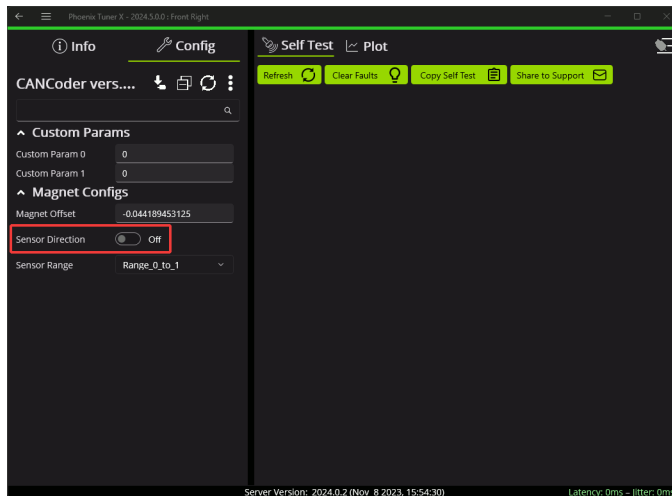
Note: Users wishing to test magnet placement must wait 8 seconds after boot for the LEDs to blink the magnet placement status.

12.2 Magnet Placement

Using the [CANcoder User's Guide](#), verify that magnet placement is correct for the CANcoder.

12.3 Verifying Sensor Direction

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

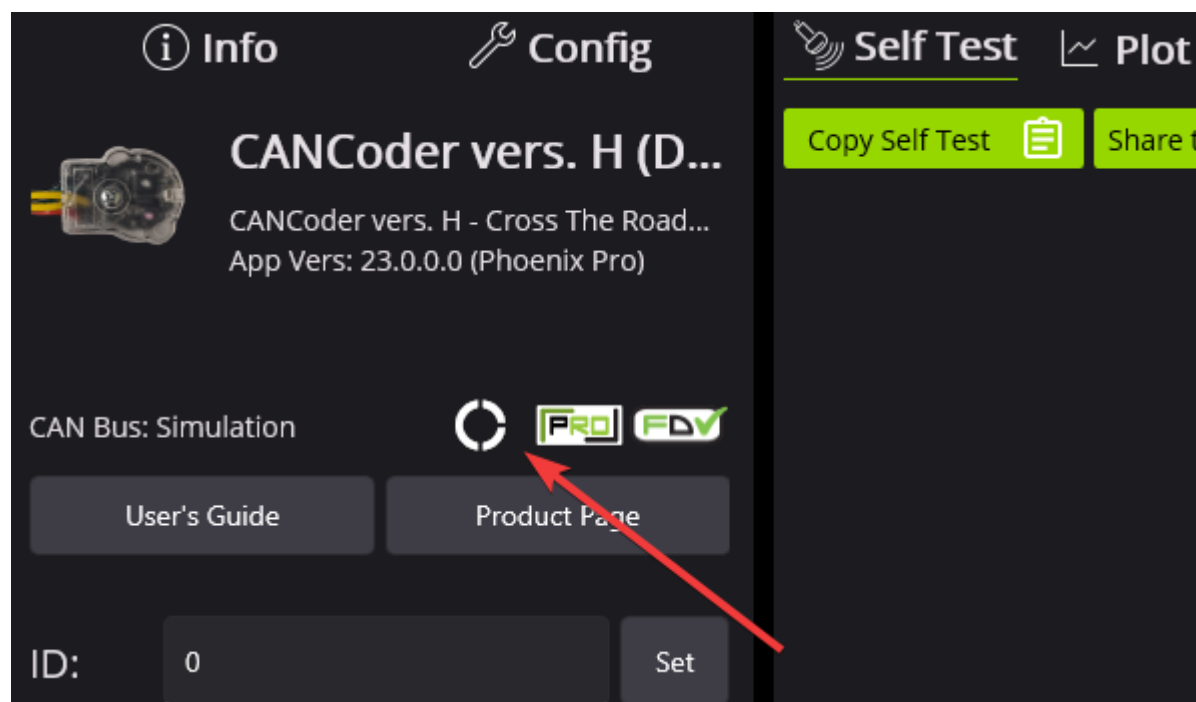


12.4 Zeroing the CANcoder

The zero offset for the CANcoder absolute position can be configured using the Magnet Offset config.

Additionally, CANcoders can be zeroed in Tuner X by pressing on the button shown below. This sets the Magnet Offset config and reports the applied offset to the user.

Important: The Tuner X CANcoder zeroing button requires 2024 diagnostics or newer.



General API Usage

This section serves to provide general 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.

TalonFX Quickstart Quickstart on controlling a TalonFX with open loop control requests and a Joystick.

- **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 output of actuators such as the TalonFX.
- **Status Signals**
 - Details using status signals to retrieve sensor data from devices.
- **Signal Logging**

- Information on the signal logging API used for capturing signal traffic on the bus.
- **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.
- **Orchestra**
 - Information on playing music and sounds using the Orchestra API.

13.1 API Overview

The Phoenix 6 API resides in the `com.ctre.phoenix6` package in Java, the `ctre::phoenix6` namespace in C++, and the `phoenix6` module in Python. 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*

13.1.1 C++ IntelliSense

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

```
// 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()};

controls::DutyCycleOut m_talonFXOut{0};

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

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

13.1.2 Python Imports

Python also takes advantage of the module structure to improve IntelliSense:

```
# first import the relevant modules and types
from phoenix6 import controls, configs, hardware, signals

# now types are organized cleanly by module
self.talonfx = hardware.TalonFX(0)
self.talonfx_out = controls.DutyCycleOut(0)

talonfx_configs = configs.TalonFXConfiguration()
talonfx_inverted = signals.InvertedValue.COUNTER_CLOCKWISE_POSITIVE
```

All Python code examples in this documentation will assume the presence of `from phoenix6 import *`.

13.2 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.

13.2.1 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 `TalonFXConfiguration`, 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++](#), [Python](#)).

Java

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

C++

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

Python

```
talonfx_configs = configs.TalonFXConfiguration()
```

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++, Python) field.

13.2.2 Configurator API

Device objects have a `getConfigurator()` method that returns a device-specific Configurator 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();
```

Python

```
talonfx_configurator = self.talonfx.configurator
```


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++, Python) 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++, Python) 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);
```

Python

```
talonfx_configurator = self.talonfx.configurator
talonfx_configs = configs.TalonFXConfiguration()

# optional timeout (in seconds) as a second optional parameter
talonfx_configurator.refresh(talonfx_configs)
```

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++, Python) for the response when no timeout parameter is specified.

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++

```
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);
```

Python

```
talonfx_configurator = self.talonfx.configurator
motor_configs = configs.MotorOutputConfigs()

# set invert to CW+ and apply config change
motor_configs.inverted = signals.InvertValue.CLOCKWISE_POSITIVE
talonfx_configurator.apply(motor_configs)
```

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. As a result, it is **unnecessary** to factory default a device before applying a modified device Configuration object.

A device's configs can be explicitly restored to the factory defaults by passing a newly-created Configuration object to the device Configurator.

Java

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

C++

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

Python

```
self.talonfx.configurator.apply(configs.TalonFXConfiguration())
```

13.3 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.

13.3.1 Applying a Control Request

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

Java

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

C++

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

Python

```
# Command m_motor to 100% of duty cycle
self.motor.set_control(controls.DutyCycleOut(1.0))
```

13.3.2 Modifying a Control Request

Control requests are mutable, so they can be saved in a member variable and reused. For example, `DutyCycleOut` (Java, C++, Python) 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

```
final DutyCycleOut m_motorRequest = new DutyCycleOut(0.0);

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

C++

```
controls::DutyCycleOut m_motorRequest{0.0};

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

Python

```
self.motor_request = controls.DutyCycleOut(0.0)

self.motor_request.output = 1.0
self.motor.set_control(self.motor_request)
```

Method Chaining API

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

Java

```
// initialize torque current FOC request with 0 amps
final TorqueCurrentFOC m_motorRequest = new TorqueCurrentFOC(0);

// mutate request with output of 10 amps and max duty cycle 0.5
m_motor.SetControl(m_motorRequest.withOutput(10).withMaxAbsDutyCycle(0.5));
```

C++

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

// mutate request with output of 10 amps and max duty cycle 0.5
m_motor.SetControl(m_motorRequest.WithOutput(10_A).WithMaxAbsDutyCycle(0.5));
```

Python

```
# initialize torque current FOC request with 0 amps
self.motor_request = controls.TorqueCurrentFOC(0)

# mutate request with output of 10 amps and max duty cycle 0.5
self.motor.set_control(self.motor_request.with_output(10).with_max_abs_duty_cycle(0.5))
```

13.3.3 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

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

C++

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

Python

```
# create a duty cycle request
self.motor_request = controls.DutyCycleOut(0)
# reduce the update frequency to 50 Hz
self.motor_request.update_freq_hz = 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.

13.4 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.

Tip: A *Signal Logging* API is available for logging received signals. This can be useful for any form of post analysis, including diagnosing issues after a match or using [WPILib SysID](#).

13.4.1 StatusSignal

The `StatusSignal` ([Java](#), [C++](#), [Python](#)) 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

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

C++

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

Python

```
supply_voltage_signal = self.device.get_supply_voltage()
```

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

Java

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

C++

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

Python

```
supply_voltage = supply_voltage_signal.value
```

Note: Phoenix 6 utilizes the [C++ units library](#) when applicable.

The `StatusCode` ([Java](#), [C++](#), [Python](#)) of the signal can be retrieved by calling `getStatus()`. 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.

13.4.2 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. Multiple signals can be refreshed in one call using `BaseStatusSignal.refreshAll()` (Java, C++, Python).

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

Java

```
// refresh the supply voltage signal
supplyVoltageSignal.refresh();
// refresh the position and velocity signals
BaseStatusSignal.refreshAll(positionSignal, velocitySignal);
```

C++

```
// refresh the supply voltage signal
supplyVoltageSignal.Refresh();
// refresh the position and velocity signals
BaseStatusSignal::RefreshAll(positionSignal, velocitySignal);
```

Python

```
# refresh the supply voltage signal
supply_voltage_signal.refresh()
# refresh the position and velocity signals
BaseStatusSignal.refresh_all(position_signal, velocity_signal)
```

13.4.3 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 `waitForUpdate()` and `getValue()` on one line.

Java

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

C++

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

Python

```
# wait up to 1 robot loop iteration (20ms) for fresh data
supply_voltage_signal.wait_for_update(0.020)
```

13.4.4 Changing Update Frequency

All signals can have their update frequency configured via the `setUpdateFrequency()` method. Additionally, the update frequency of multiple signals can be specified at once using `BaseStatusSignal.setUpdateFrequencyForAll()` (Java, C++, Python).

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

```
// disable supply voltage reporting (0 Hz)
supplyVoltageSignal.setUpdateFrequency(0);
// speed up position and velocity reporting to 200 Hz
BaseStatusSignal.setUpdateFrequencyForAll(200, positionSignal, velocitySignal);
```

C++

```
// disable supply voltage reporting (0 Hz)
supplyVoltageSignal.SetUpdateFrequency(0_Hz);
// speed up position and velocity reporting to 200 Hz
BaseStatusSignal::SetUpdateFrequencyForAll(200_Hz, positionSignal, velocitySignal);
```

Python

```
# disable supply voltage reporting (0 Hz)
supply_voltage_signal.set_update_frequency(0)
# speed up position and velocity reporting to 200 Hz
BaseStatusSignal.set_update_frequency_for_all(200, position_signal, velocity_signal)
```

When different update frequencies are specified for signals that share a status frame, the highest update frequency of all the relevant signals will be applied to the entire frame. Users can get a signal's applied update frequency using the `getAppliedUpdateFrequency()` method.

Signal update frequencies are automatically reapplied by the robot program on device reset.

Optimizing Bus Utilization

For users that wish to disable every unused status signal for their devices to reduce bus utilization, device objects have an `optimizeBusUtilization()` method ([Java](#), [C++](#), [Python](#)). Additionally, multiple devices can be optimized at once using `ParentDevice.optimizeBusUtilizationForAll()` ([Java](#), [C++](#), [Python](#)).

When optimizing the bus utilization for devices, all status signals that have not been given an update frequency using `setUpdateFrequency()` will be disabled. This results in an opt-in model for status signals, maximizing the reduction in bus utilization.

Java

```
m_pigeon.optimizeBusUtilization();
ParentDevice.optimizeBusUtilizationForAll(m_leftMotor, m_rightMotor, m_cancoder);
```

C++

```
m_pigeon.OptimizeBusUtilization();
hardware::ParentDevice::OptimizeBusUtilizationForAll(m_leftMotor, m_rightMotor, m_
↪cancoder);
```

Python

```
self.pigeon.optimize_bus_utilization()
hardware.ParentDevice.optimize_bus_utilization_for_all(self.left_motor, self.right_
↪motor, self.cancoder)
```

13.4.5 Timestamps

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

13.4.6 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.

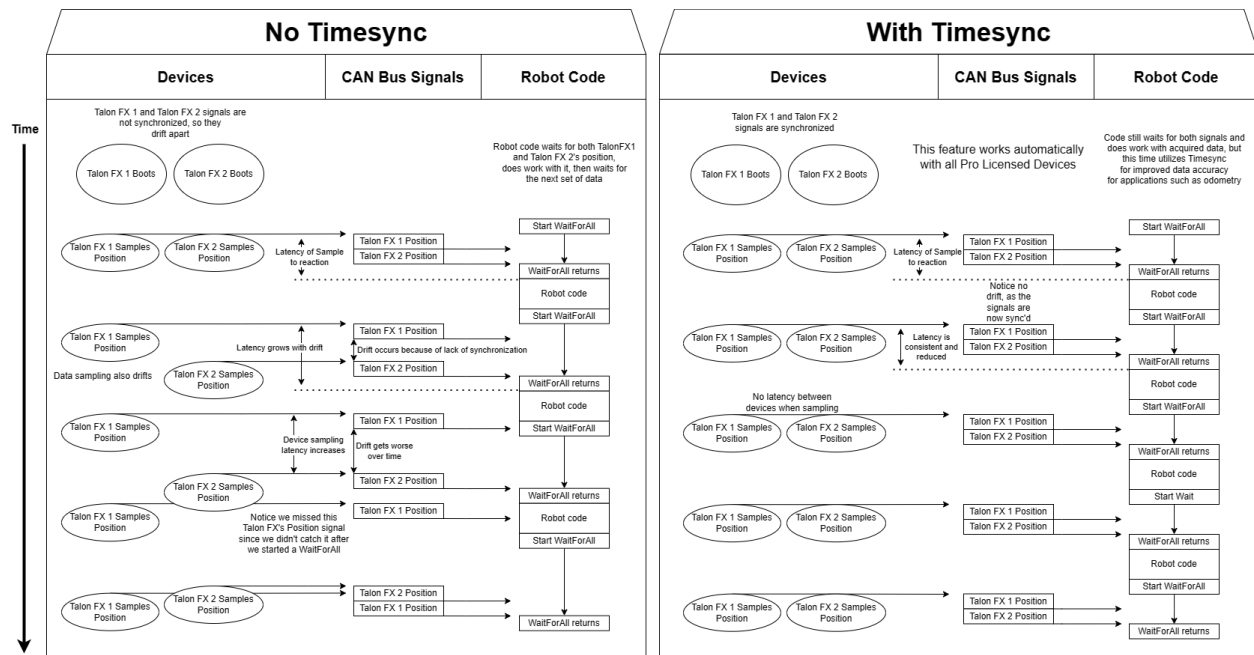
Users can synchronously wait for these signals to update using `BaseStatusSignal.waitForAll()` (Java, C++, Python).

Tip: `waitForAll()` with a timeout of zero matches the behavior of `refreshAll()`, performing 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.



Check the API documentation for information on whether a status signal supports CANivore Timesync.

Java

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

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

C++

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

BaseStatusSignal::WaitForAll(20_ms, talonFXPositionSignal, cancoderPositionSignal,
    ↪ pigeon2YawSignal);
```

Python

```

talonfx_position_signal = self.talonfx.get_position()
cancoder_position_signal = self.cancoder.get_position()
pigeon2_yaw_signal = self.pigeon2.get_yaw()

BaseStatusSignal.wait_for_all(0.020, talonfx_position_signal, cancoder_position_
↪signal, pigeon2_yaw_signal)

```

13.4.7 Latency Compensation

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

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

```

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

```

C++

```

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

```

Python

```

compensated_turns = BaseStatusSignal.get_latency_compensated_value(self.motor.get_
↪position(), self.motor.get_velocity())

```

13.4.8 SignalMeasurement

All `StatusSignal` objects have a `getDataCopy()` method that returns a new `SignalMeasurement` (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.

13.5 Signal Logging

Note: Information on how to retrieve and convert hoot files to compatible formats can be found in [Extracting Signal Logs](#).

Phoenix 6 comes with a real-time, high-fidelity signal logger. This can be useful for any form of post analysis, including diagnosing issues after a match or using [WPILib SysID](#).

The Phoenix 6 signal logger provides the following advantages over alternatives:

- **All status signals** are captured **automatically** with their **timestamps from CAN**.
- Status signals are captured **as they arrive** at their configured update frequency.
- Logging is **not affected** by the timing of the main robot loop or Java GC, significantly improving the sensitivity and accuracy of system identification.
- Signal logging is **started automatically** during an **FRC match** by default.
- **Custom user signals** can be logged alongside the automatically captured status signals on the **same timebase**.
- The **highly efficient** hoot file format minimizes the **size** of the log files and the **CPU usage** of the logger.

The signal logging API is available through static functions in the `SignalLogger` ([Java](#), [C++](#), [Python](#)) class. Signal logging is **enabled by default** whenever it detects an FRC match is currently being played. Users can disable this behavior with `SignalLogger.enableAutoLogging(false)` ([Java](#), [C++](#), [Python](#)).

13.5.1 Setting Log Path

The logging directory can optionally be changed using `SignalLogger.setPath()` ([Java](#), [C++](#), [Python](#)). If the specified directory does not exist, `SignalLogger.setPath()` will return an error code. Setting the path while logging will restart the log.

The below example sets the logging path to a `ctre-logs` folder on the first USB drive found.

Java

```
SignalLogger.setPath("/media/sda1/ctre-logs/");
```

C++

```
SignalLogger::SetPath("/media/sda1/ctre-logs/");
```

Python

```
SignalLogger.set_path("/media/sda1/ctre-logs/")
```

Note: Each CAN bus gets its own dedicated log file.

13.5.2 Start/Stop Logging

The signal logger can be started and stopped using the Start/Stop functions.

Java

```
SignalLogger.start();  
SignalLogger.stop();
```

C++

```
SignalLogger::Start();  
SignalLogger::Stop();
```

Python

```
SignalLogger.start()  
SignalLogger.stop()
```

13.5.3 Writing Custom Signals

Users can write custom signals to the currently opened logs by utilizing the `write*()` functions. An example application of this is logging your swerve odometry data.

The integer and floating-point `write*()` functions can optionally be supplied a units string to log alongside the data. Additionally, all `write*()` functions support an optional latency parameter that is subtracted from the current time to get the latency-adjusted timestamp of the signal. This can be useful for logging high-latency data, such as vision measurements.

Java

```
// Log the odometry pose as a double array
SignalLogger.writeDoubleArray("odometry", new double[] {pose.getX(), pose.getY(),
↳ pose.getRotation().getDegrees()});
// Log the odometry period with units of "seconds"
SignalLogger.writeDouble("odom period", state.OdometryPeriod, "seconds");
// Log the camera pose with calculated latency
SignalLogger.writeDoubleArray("camera pose", new double[] {camPose.getX(), camPose.
↳ getY(), camPose.getRotation().getDegrees()},
    "", Timer.getFPGATimestamp() - camRes.getTimestampSeconds());
```

C++

```
// Log the odometry pose as a double array
SignalLogger::WriteDoubleArray("odometry", std::array<double, 3>{pose.X().value(),
↳ pose.Y().value(), pose.Rotation().Degrees().value()});
// Log the odometry period with units of "seconds"
SignalLogger::WriteDouble("odom period", state.OdometryPeriod, "seconds");
// Log the camera pose with calculated latency
SignalLogger::WriteDoubleArray("camera pose", std::array<double, 3>{camPose.X().
↳ value(), camPose.Y().value(), camPose.Rotation().Degrees().value()},
    "", frc::Timer::GetFPGATimestamp() - camRes.GetTimestamp());
```

Python

```
# Log the odometry pose as a double array
SignalLogger.write_double_array("odometry", [pose.X(), pose.Y(), pose.rotation().
↳ degrees()])
# Log the odometry period with units of "seconds"
SignalLogger.write_double("odom period", state.odometry_period, "seconds")
# Log the camera pose with calculated latency
SignalLogger.write_double_array("camera pose", [cam_pose.X(), cam_pose.Y(), cam_pose.
↳ rotation().degrees()],
    "", wpilib.Timer.getFPGATimestamp() - cam_res.getTimestamp())
```

13.5.4 Free Signals

Any log that contains a *pro-licensed* device will export all signals. Otherwise, the following status signals and all custom signals can be exported for free.

[Click here to view free signals](#)

Common Signals:

- VersionMajor
- VersionMinor
- VersionBugfix
- VersionBuild
- IsProLicensed
- Fault_UnlicensedFeatureInUse
- Fault_BootDuringEnable

TalonFX:

- SupplyVoltage
- SupplyCurrent
- MotorVoltage
- Position
- Velocity
- DeviceEnable
- Fault_DeviceTemp
- Fault_ProcTemp

CANcoder:

- SupplyVoltage
- Position
- Velocity

Pigeon 2.0:

- SupplyVoltage
- Yaw
- AngularVelocityZWorld

13.5.5 Low Storage Space Behavior

If the target drive (i.e. flash drive or roboRIO internal storage) reaches 50 MB free space, old logs will be deleted, and a warning will be printed.

If the target drive reaches 5 MB of free space, logging will be stopped, and an error will be printed. Logging cannot be resumed until more disk space is made available.

An example error that may occur if the free space limit is reached is shown below.

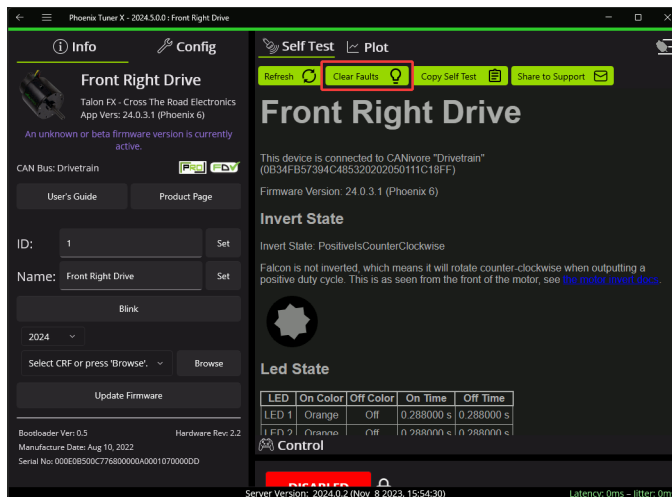
[phoenix] Signal Logger: Available disk space (3 MB) below 5 MB, stopping log

13.6 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 *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.



13.6.1 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

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

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

C++

```
auto faulted = m_cancoder.GetFault_BadMagnet().GetValue();
if (faulted) {
    // do action when bad magnet fault is set
}
```

Python

```
faulted = self.cancoder.get_fault_bad_magnet().value
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.

13.6.2 Using API to Clear Sticky Faults

Sticky faults can be cleared in API using the `clearStickyFaults()` method on the device objects. Additionally, individual sticky faults may be cleared using the `clearStickyFault_*`() APIs.

Note: Clearing sticky faults is a blocking operation and should not be run in a periodic loop.

Java

```
// clear the undervoltage sticky fault
m_cancoder.clearStickyFault_Undervoltage();
```

C++

```
// clear the undervoltage sticky fault
m_cancoder.ClearStickyFault_Undervoltage();
```

Python

```
# clear the undervoltage sticky fault
self.cancoder.clear_sticky_fault_undervoltage()
```

13.7 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.

13.7.1 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.

13.7.2 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);
```

Python

```
# feed the enable signal, timeout after 100ms
phoenix6.unmanaged.feed_enable(0.100)
```

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.

13.7.3 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*.

13.8 Actuator Limits

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

Note: The TalonFX + Kraken X60 does not support hardware limit switches. Instead, *control request limit* overrides can be used, or a CANcoder can be used a *remote limit switch*.

Documentation on wiring limit switches can be found [here](#).

13.8.1 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++

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

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

Python

```
forward_limit = self.motor.get_forward_limit()

if forward_limit.value is signals.ForwardLimitValue.CLOSED_TO_GROUND:
    # do action when forward limit is closed
```

13.8.2 Control Request Limits

Many *control requests* support overriding the limit switch values using `LimitForwardMotion` and `LimitReverseMotion` parameters ([Java](#), [C++](#), [Python](#)). These allow users to use other limit switch sensors connected to the robot controller.

Java

```
final DigitalInput m_forwardLimit = new DigitalInput(0);
final DigitalInput m_reverseLimit = new DigitalInput(1);

final DutyCycleOut m_dutyCycle = new DutyCycleOut(0.0);

m_motor.setControl(m_dutyCycle.withOutput(0.5)
    .withLimitForwardMotion(m_forwardLimit.get())
    .withLimitReverseMotion(m_reverseLimit.get()));
```

C++

```
frc::DigitalInput m_forwardLimit{0};
frc::DigitalInput m_reverseLimit{1};

controls::DutyCycleOut m_dutyCycle{0.0};

m_motor.SetControl(m_dutyCycle.WithOutput(0.5)
    .WithLimitForwardMotion(m_forwardLimit.Get())
    .WithLimitReverseMotion(m_reverseLimit.Get()));
```

Python

```
self.forward_limit = wpilib.DigitalInput(0)
self.reverse_limit = wpilib.DigitalInput(1)

self.duty_cycle = controls.DutyCycleOut(0.0)

self.motor.set_control(self.duty_cycle.with_output(0.5)
    .with_limit_forward_motion(self.forward_limit.get())
    .with_limit_reverse_motion(self.reverse_limit.get()))
```

13.8.3 Remote Limit Switches

Supported devices (TalonFX, CANcoder, CANifier) can be utilized as a remote limit switch, disabling actuator outputs when triggers. When utilizing a CANcoder as a remote sensor, the limit will trigger when the magnet strength changes from BAD (red) to ADEQUATE (orange) or GOOD (green).

The remote limit switch can be selected using the `LimitSource` and `LimitRemoteSensorID` configs.

Java

```
var limitConfigs = new HardwareLimitSwitchConfigs();
limitConfigs.ForwardLimitSource = ForwardLimitSourceValue.RemoteCANcoder;
limitConfigs.ForwardLimitRemoteSensorID = m_cancoder.getDeviceID();

m_motor.getConfigurator().apply(limitConfigs);
```

C++

```
configs::HardwareLimitSwitchConfigs limitConfigs{};
limitConfigs.ForwardLimitSource = signals::ForwardLimitSourceValue::RemoteCANcoder;
limitConfigs.ForwardLimitRemoteSensorID = m_cancoder.GetDeviceID();

m_motor.GetConfigurator().Apply(limitConfigs);
```

Python

```
limit_configs = configs.HardwareLimitSwitchConfigs()
limit_configs.forward_limit_source = signals.ForwardLimitSourceValue.REMOTE_CANCODER
limit_configs.forward_limit_remote_sensor_id = self.cancoder.device_id

self.motor.configurator.apply(limit_configs)
```

13.9 Orchestra

Compatible motors powered by Talon FX have the ability to produce audible output through the `MusicTone` (Java, C++, Python) control request. The Orchestra API extends this ability and orchestrates multiple motors to play music.

To get started, construct an `Orchestra` (Java, C++, Python) object with an instrument and `CHRP`. Ensure that `addInstrument()` and `loadMusic()` are not called periodically, as they are blocking functions.

Note: For information on converting MIDI to CHRP, see [CHRP Converter](#).

Java

```
Orchestra m_orchestra = new Orchestra();

// Add a single device to the orchestra
m_orchestra.addInstrument(m_motor);

// Attempt to load the chrp
var status = m_orchestra.loadMusic("track.chrp");

if (!status.isOK()) {
    // log error
}
```

C++

```
Orchestra m_orchestra;

// Add a single device to the orchestra
m_orchestra.addInstrument(m_motor);

// Attempt to load the chrp
auto status = m_orchestra.loadMusic("track.chrp");

if (!status.IsOK()) {
    // log error
}
```

Python

```
self.orchestra = Orchestra()

self.orchestra.add_instrument(self.motor);

status = self.orchestra.load_music("track.chrp")

if not status.is_ok():
    # log error
```

Once the track has been loaded, play/pause/stop can be used to manage the track. `play()` only needs to be called once.

Java

```
m_orchestra.play();
```

C++

```
m_orchestra.Play();
```

Python

```
self.orchestra.play()
```

13.9.1 Playback While Disabled (FRC)

Playback can be safely enabled during robot disable by enabling the Allow Music Dur Disable (Java, C++, Python) config.

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

14.1 TalonFX

14.1.1 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 currently supports three base control output types: DutyCycle, Voltage, and TorqueCurrentFOC.

Note: There are various configuration options available that influence the onboard control (for example, see *Improving Performance with Current Limits*).

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 torque current of the motor. Unlike the other control output types, where output roughly controls the velocity of the motor, a TorqueCurrentFOC request **directly 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++](#), [Python](#)). There are also control types that require FOC, such as TorqueCurrentFOC.

Behavior While Unlicensed

When controlling an unlicensed device, the device will automatically fall back to non-FOC commutation for control requests that support the EnableFOC field.

For control requests that require FOC, such as TorqueCurrentFOC, the unlicensed device will:

- Disable control output
- *Set the UnlicensedFeatureInUse fault*
- *Blink unlicensed*

14.1.2 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}.

In the below example, note that devices are initialized with two arguments. These arguments correspond to *the device ID* and CAN bus name. For CANivore, this is the name of

the CANivore as configured in *Phoenix Tuner X*. For more information, see the *CANivore API* documentation.

Java

```
// initialize devices on the rio can bus
final TalonFX m_leftLeader = new TalonFX(0, "rio");
final TalonFX m_rightLeader = new TalonFX(1, "rio");

// users should reuse control requests when possible
final DutyCycleOut m_leftRequest = new DutyCycleOut(0.0);
final DutyCycleOut m_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(m_leftRequest.withOutput(leftOut));
m_rightLeader.setControl(m_rightRequest.withOutput(rightOut));
```

C++

```
// initialize devices on the rio can bus
hardware::TalonFX m_leftLeader{0, "rio"};
hardware::TalonFX m_rightLeader{1, "rio"};

// users should reuse control requests when possible
controls::DutyCycleOut m_leftRequest{0.0};
controls::DutyCycleOut m_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(m_leftRequest.WithOutput(leftOut));
m_rightLeader.SetControl(m_rightRequest.WithOutput(rightOut));
```

Python

```
# initialize devices on the rio can bus
self.left_leader = hardware.TalonFX(0, "rio")
self.right_leader = hardware.TalonFX(1, "rio")

# users should reuse control requests when possible
self.left_request = controls.DutyCycleOut(0.0)
self.right_request = controls.DutyCycleOut(0.0)

# retrieve joystick inputs
forward = -self.driver_joy.getLeftY()
turn = self.driver_joy.getRightX()

# calculate motor outputs, utilizes a "arcade" style of driving
# where left Y controls forward and right X controls rotation/turn
left_out = forward + turn
right_out = forward - turn

# set request to motor controllers
self.left_leader.set_control(self.left_request.with_output(left_out))
self.right_leader.set_control(self.right_request.with_output(right_out))
```

14.1.3 Closed-Loop Overview

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 benefits of onboard closed-loop control are that there is no sensor latency, and the closed-loop controller has a 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.

Choosing Output Type

The choice of *control output type* can affect the reproducibility and stability of the closed-loop control.

DutyCycle has the benefit of being the simplest control output type, as it is unaffected by voltage and current measurements. However, because DutyCycle represents a proportion of the supply voltage, changes in battery voltage can affect the reproducibility of the control request.

Voltage control output takes into account the supply voltage to ensure its voltage output remains consistent. As a result, Voltage control often results in more stable and reproducible behavior compared to DutyCycle control, so Voltage control is often preferred.

A disadvantage with both DutyCycle and Voltage control output types is that they **control acceleration indirectly** and require a velocity feedforward K_v to hold a constant velocity. On the other hand, torque-based control output types, such as TorqueCurrentFOC, **directly control acceleration**, which has several advantages:

- Since the torque request is directly proportional to acceleration, K_v is generally unnecessary. A torque output of 0 corresponds to a constant velocity, assuming no external forces.
- K_a can be tuned independently of all the other closed-loop gains by comparing the measured acceleration with the requested acceleration.
- Because the output is in units of torque, the units of the gains more closely match those of forces in the real world.

As a result, torque-based control output types offer more stable and reproducible behavior that can be easier to tune compared to the other control output types.

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++, Python) objects.

Gravity Feedforward

The gravity feedforward K_g is the output necessary to overcome gravity, in units of the *control output type*. Phoenix 6 supports the two most common use cases for K_g —elevators and arms—using the GravityType config in the gain slots.

Elevator/Static

For systems with a constant gravity component, such as an elevator, K_g adds a constant value to the closed-loop output. To find K_g , determine the output necessary to hold the elevator at a constant height in open-loop control.

Arm/Cosine

For systems with an angular gravity component, such as an arm, the output of K_g is dependent on the cosine of the angle between the arm and horizontal. The value of K_g can be found by determining the output necessary to hold the arm horizontally forward.

Since the arm K_g uses the angle of the arm relative to horizontal, the Talon FX often requires an absolute sensor whose position is 1:1 with the arm, and the sensor offset and ratios must be configured.

When using an absolute sensor, such as a CANcoder, the sensor offset must be configured such that a position of 0 represents the arm being held horizontally forward. From there, the RotorToSensor ratio must be configured to the ratio between the absolute sensor and the Talon FX rotor.

Converting from Meters

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

$$rotations = \frac{meters}{2\pi \cdot wheelRadius} \cdot gearRatio$$

where `meters` is the target in meters, `wheelRadius` is the radius 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 or acceleration from m/s² to rps/s.

Continuous Mechanism Wrap

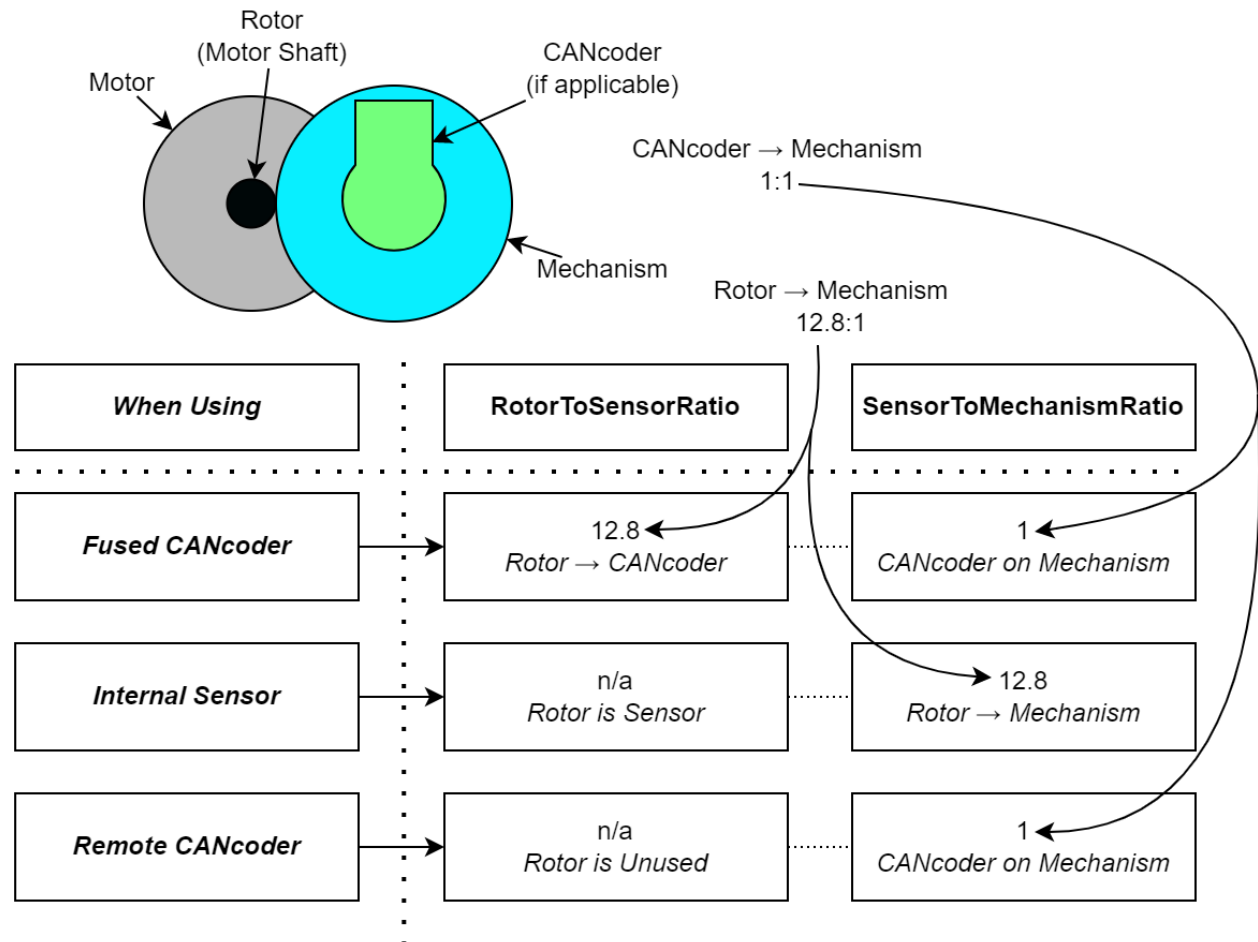
A continuous mechanism is a mechanism with unlimited travel in any direction, and whose rotational position can be represented with multiple unique position values. Some examples of continuous mechanisms are swerve drive steer mechanisms or turrets (without cable management).

`ContinuousWrap` ([Java](#), [C++](#), [Python](#)) 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++, Python) 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.



14.1.4 Basic PID and Profiling

The Talon FX supports basic PID control and motion profiling for position and velocity.

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 are 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_a - unused, as there is no target acceleration

- 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);
```

Python

```
# in init function, set slot 0 gains
slot0_configs = configs.Slot0Configs()
slot0_configs.k_p = 24 # An error of 0.5 rotations results in 12 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0.1 # A velocity of 1 rps results in 0.1 V output

self.talonfx.configurator.apply(slot0_configs)
```

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
final PositionVoltage m_request = new PositionVoltage(0).withSlot(0);

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

C++

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

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

Python

```
# create a position closed-loop request, voltage output, slot 0 configs
self.request = controls.PositionVoltage(0).with_slot(0)

# set position to 10 rotations
self.talonfx.set_control(self.request.with_position(10))
```

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 are 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_a - unused, as there is no target acceleration
- 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; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

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; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

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

Python

```
slot0_configs = configs.Slot0Configs()
slot0_configs.k_s = 0.05 # Add 0.05V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_p = 0.11 # An error of 1 rps results in 0.11 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0 # no output for error derivative

self.talonfx.configurator.apply(slot0_configs)
```

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

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

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

C++

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

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

Python

```
# create a velocity closed-loop request, voltage output, slot 0 configs
self.request = controls.VelocityVoltage(0).with_slot(0)

# set velocity to 8 rps, add 0.5 V to overcome gravity
self.talonfx.set_control(self.request.with_velocity(8).with_feed_forward(0.5))
```

Motion Profiling

The Position and Velocity closed-loop requests can be used to run a [motion profile](#) generated by the robot controller.

Tip: The Talon FX supports several onboard motion profiles using *Motion Magic®*.

Position

In a Position motion profile, 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_a - unused, as there is no target acceleration
- 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.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

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

C++

```
// in init function, set slot 0 gains
configs::Slot0Configs 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

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

Python

```
# in init function, set slot 0 gains
slot0_configs = configs.Slot0Configs()
slot0_configs.k_s = 0.25 # Add 0.25 V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_p = 4.8 # A position error of 2.5 rotations results in 12 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0.1 # A velocity error of 1 rps results in 0.1 V output

self.talonfx.configurator.apply(slot0_configs)
```

Once the gains are configured, the Position closed-loop control request can be sent to the TalonFX. The Velocity parameter is used to specify the current setpoint velocity of the motion profile.

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

```
// Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
final TrapezoidProfile m_profile = new TrapezoidProfile(
    new TrapezoidProfile.Constraints(80, 160)
);
// Final target of 200 rot, 0 rps
TrapezoidProfile.State m_goal = new TrapezoidProfile.State(200, 0);
TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();

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

// calculate the next profile setpoint
m_setpoint = m_profile.calculate(0.020, m_setpoint, m_goal);

// send the request to the device
m_request.Position = m_setpoint.position;
m_request.Velocity = m_setpoint.velocity;
m_talonFX.setControl(m_request);
```

C++

```
// Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
frc::TrapezoidProfile<units::turn_t> m_profile{{80_tps, 160_tr_per_s_sq}};
// Final target of 200 rot, 0 rps
frc::TrapezoidProfile<units::turn_t>::State m_goal{200_tr, 0_tps};
frc::TrapezoidProfile<units::turn_t>::State m_setpoint{};

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

// calculate the next profile setpoint
m_setpoint = m_profile.Calculate(20_ms, m_setpoint, m_goal);

// send the request to the device
m_request.Position = m_setpoint.position;
m_request.Velocity = m_setpoint.velocity;
m_talonFX.SetControl(m_request);
```

Python

```
# Trapezoid profile with max velocity 80 rps, max accel 160 rps/s
self.profile = TrapezoidProfile(
    TrapezoidProfile.Constraints(80, 160)
)
# Final target of 200 rot, 0 rps
self.goal = TrapezoidProfile.State(200, 0)
self.setpoint = TrapezoidProfile.State()

# create a position closed-loop request, voltage output, slot 0 configs
self.request = controls.PositionVoltage(0).with_slot(0)

# calculate the next profile setpoint
self.setpoint = self.profile.calculate(0.020, self.setpoint, self.goal)

# send the request to the device
self.request.position = self.setpoint.position
self.request.velocity = self.setpoint.velocity
self.talonfx.set_control(self.request)
```

Velocity

In a Velocity motion profile, 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_a - output per unit of requested acceleration (output/(rps/s))
- 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 = 0.25; // 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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

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

C++

```
// in init function, set slot 0 gains
configs::Slot0Configs 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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

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

Python

```
# in init function, set slot 0 gains
slot0_configs = configs.Slot0Configs()
slot0_configs.k_s = 0.25 # Add 0.25 V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_a = 0.01 # An acceleration of 1 rps/s requires 0.01 V output
slot0_configs.k_p = 0.11 # An error of 1 rps results in 0.11 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0 # no output for error derivative

self.talonfx.configurator.apply(slot0_configs)
```

Once the gains are configured, the Velocity closed-loop control request can be sent to the TalonFX. The Acceleration parameter is used to specify the current setpoint acceleration of the motion profile.

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

```
// Trapezoid profile with max acceleration 400 rot/s^2, max jerk 4000 rot/s^3
final TrapezoidProfile m_profile = new TrapezoidProfile(
    new TrapezoidProfile.Constraints(400, 4000)
);
// Final target of 80 rps, 0 rps/s
TrapezoidProfile.State m_goal = new TrapezoidProfile.State(80, 0);
TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();

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

// calculate the next profile setpoint
m_setpoint = m_profile.calculate(0.020, m_setpoint, m_goal);

// send the request to the device
// note: "position" is velocity, and "velocity" is acceleration
m_request.Velocity = m_setpoint.position;
m_request.Acceleration = m_setpoint.velocity;
m_talonFX.setControl(m_request);
```

C++

```
// Trapezoid profile with max acceleration 400 rot/s^2, max jerk 4000 rot/s^3
frc::TrapezoidProfile<units::turns_per_second_t> m_profile{{400_tr_per_s_sq, 4000_tr_
    ↪per_s_cu}};
// Final target of 80 rps, 0 rot/s^2
frc::TrapezoidProfile<units::turns_per_second_t>::State m_goal{80_tps, 0_tr_per_s_sq};
frc::TrapezoidProfile<units::turns_per_second_t>::State m_setpoint{};

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

// calculate the next profile setpoint
m_setpoint = m_profile.Calculate(20_ms, m_setpoint, m_goal);

// send the request to the device
// note: "position" is velocity, and "velocity" is acceleration
m_positionControl.Velocity = m_setpoint.position;
m_positionControl.Acceleration = m_setpoint.velocity;
m_talonFX.SetControl(m_request);
```

Python

```
# Trapezoid profile with max acceleration 400 rot/s^2, max jerk 4000 rot/s^3
self.profile = TrapezoidProfile(
    TrapezoidProfile.Constraints(400, 4000)
)
# Final target of 80 rps, 0 rot/s^2
self.goal = TrapezoidProfile.State(80, 0)
self.setpoint = TrapezoidProfile.State()
```

(continues on next page)

(continued from previous page)

```
# create a velocity closed-loop request, voltage output, slot 0 configs
self.request = controls.VelocityVoltage(0).with_slot(0)

# calculate the next profile setpoint
self.setpoint = self.profile.calculate(0.020, self.setpoint, self.goal)

# send the request to the device
# note: "position" is velocity, and "velocity" is acceleration
self.request.velocity = self.setpoint.position
self.request.acceleration = self.setpoint.velocity
self.talonfx.set_control(self.request)
```

14.1.5 Motion Magic® Controls

In addition to basic PID control, the Talon FX also supports onboard motion profiling using Motion Magic® controls.

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 setpoints
- Improved repeatability despite changes in battery load
- Improved repeatability despite changes in motor load

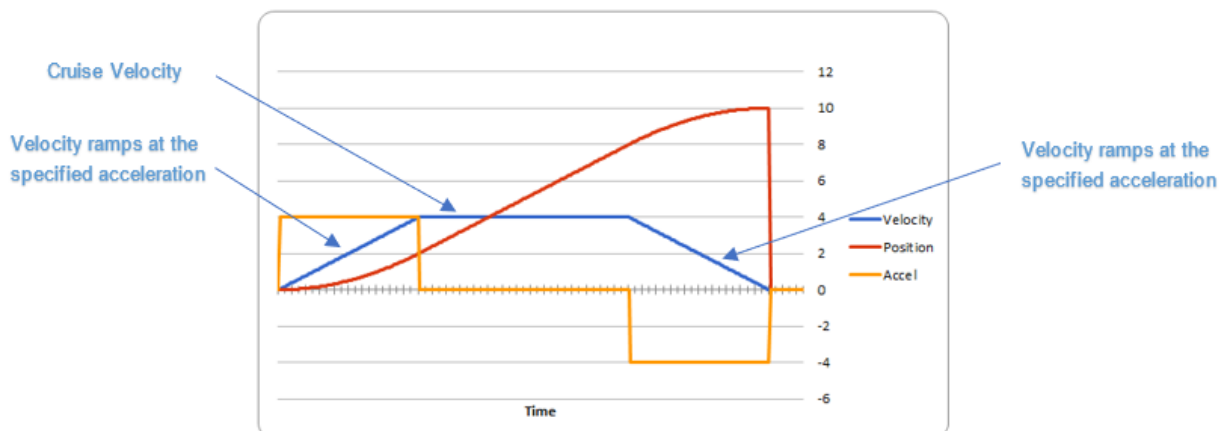
After gain/settings are determined, the robot controller only needs to periodically set the target position.

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

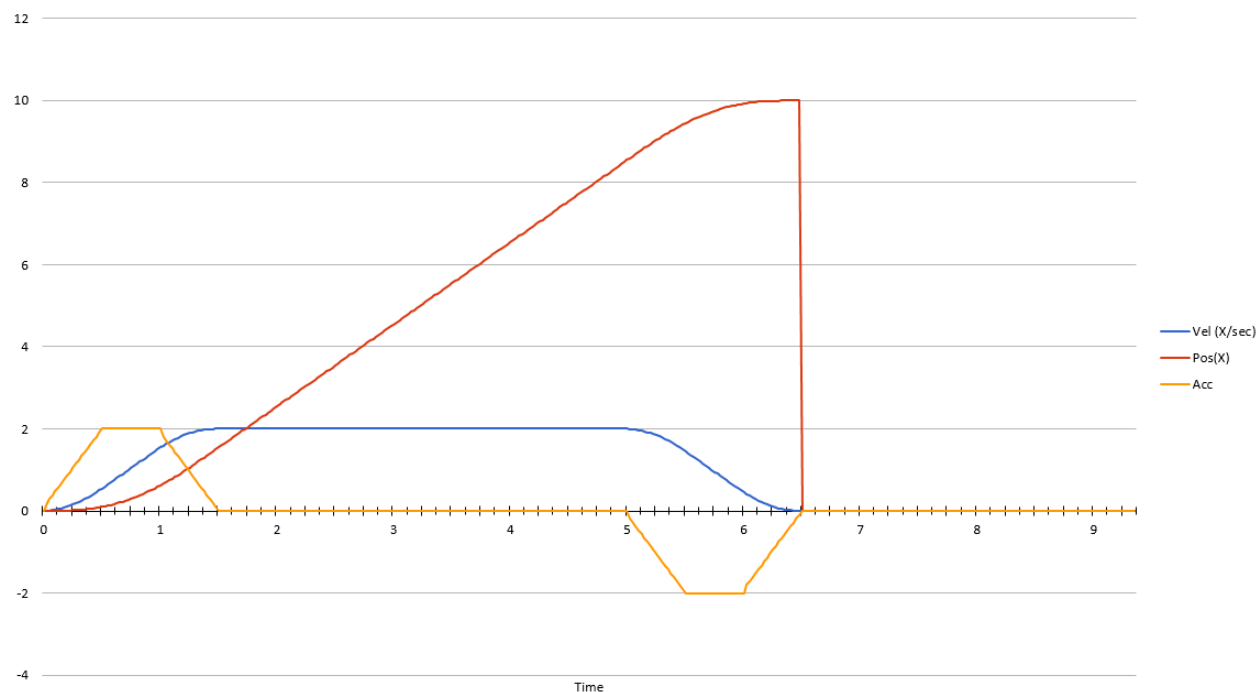


If the Motion Magic® jerk is set to a nonzero value, the generated velocity profile is no longer trapezoidal, but instead is a continuous S-Curve (corner points are smoothed).

An S-Curve profile has the following advantages 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 (optional) - 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 are determined by the control output type.

The Motion Magic® jerk, acceleration, and cruise velocity can be *configured in code* using a MotionMagicConfigs (Java, C++, Python) 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_a - output per unit of target acceleration (output/(rps/s))
- 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.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 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.MotionMagic;
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.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 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.MotionMagic;
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);
```

Python

```
# in init function
talonfx_configs = configs.TalonFXConfiguration()

# set slot 0 gains
slot0_configs = talonfx_configs.slot0
slot0_configs.k_s = 0.25 # Add 0.25 V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_a = 0.01 # An acceleration of 1 rps/s requires 0.01 V output
slot0_configs.k_p = 4.8 # A position error of 2.5 rotations results in 12 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0.1 # A velocity error of 1 rps results in 0.1 V output

# set Motion Magic settings
motion_magic_configs = talonfx_configs.motion_magic
motion_magic_configs.motion_magic_cruise_velocity = 80 # Target cruise velocity of 80
↳ rps
motion_magic_configs.motion_magic_acceleration = 160 # Target acceleration of 160 rps/
↳ s (0.5 seconds)
motion_magic_configs.motion_magic_jerk = 1600 # Target jerk of 1600 rps/s/s (0.1
↳ seconds)

self.talonfx.configurator.apply(talonfx_configs)
```

Tip: Motion Magic® supports modifying cruise velocity, acceleration, and jerk on the fly (requires firmware version 24.0.6.0 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
final MotionMagicVoltage m_request = new MotionMagicVoltage(0);

// set target position to 100 rotations
m_talonFX.setControl(m_request.withPosition(100));
```

C++

```
// create a Motion Magic request, voltage output
controls::MotionMagicVoltage m_request{0_tr};

// set target position to 100 rotations
m_talonFX.SetControl(m_request.WithPosition(100_tr));
```

Python

```
# create a Motion Magic request, voltage output
self.request = controls.MotionMagicVoltage(0)

# set target position to 100 rotations
self.talonfx.set_control(self.request.with_position(100))
```

Dynamic Motion Magic®

Important: This feature requires the device to be *Pro licensed* and on a *CANivore*. When unlicensed, the TalonFX will disable control output and trip the `UnlicensedFeatureInUse` fault.

When using a Pro-licensed Talon FX connected to a CANivore, Dynamic Motion Magic® can be used, allowing for the cruise velocity, acceleration, and jerk to be modified directly in the control request during motion. This can be used to set up different values for acceleration vs deceleration or to speed up and slow down the profile on the fly.

The gain slots are configured in the same way as a regular Motion Magic® request. However, the cruise velocity, acceleration, and jerk parameters are set up in the control request, **not** the Motion Magic® config group.

Once the gains are configured, the Dynamic 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 Dynamic Motion Magic request, voltage output
// default velocity of 80 rps, acceleration of 400 rot/s^2, and jerk of 4000 rot/s^3
final DynamicMotionMagicVoltage m_request =
    new DynamicMotionMagicVoltage(0, 80, 400, 4000);

if (m_joy.getAButton()) {
    // while the joystick A button is held, use a slower profile
    m_request.Velocity = 40; // rps
    m_request.Acceleration = 80; // rot/s^2
    m_request.Jerk = 400; // rot/s^3
} else {
    // otherwise use a faster profile
    m_request.Velocity = 80; // rps
    m_request.Acceleration = 400; // rot/s^2
    m_request.Jerk = 4000; // rot/s^3
}

// set target position to 100 rotations
m_talonFX.setControl(m_request.withPosition(100));
```

C++

```
// create a Dynamic Motion Magic request, voltage output
// default velocity of 80 rps, acceleration of 400 rot/s^2, and jerk of 4000 rot/s^3
controls::DynamicMotionMagicVoltage m_request{
    0_tr, 80_tps, 400_tr_per_s_sq, 4000_tr_per_s_cu};

if (m_joy.GetAButton()) {
    // while the joystick A button is held, use a slower profile
    m_request.Velocity = 40_tps;
    m_request.Acceleration = 80_tr_per_s_sq;
    m_request.Jerk = 400_tr_per_s_cu;
} else {
    // otherwise use a faster profile
    m_request.Velocity = 80_tps;
    m_request.Acceleration = 400_tr_per_s_sq;
    m_request.Jerk = 4000_tr_per_s_cu;
}

// set target position to 100 rotations
m_talonFX.SetControl(m_request.WithPosition(100_tr));
```

Python

```
# create a Dynamic Motion Magic request, voltage output
# default velocity of 80 rps, acceleration of 400 rot/s^2, and jerk of 4000 rot/s^3
self.request = controls.DynamicMotionMagicVoltage(0, 80, 400, 4000)

if self.joy.getAButton():
    # while the joystick A button is held, use a slower profile
    self.request.velocity = 40 # rps
    self.request.acceleration = 80 # rot/s^2
    self.request.jerk = 400 # rot/s^3
else:
    # otherwise use a faster profile
    self.request.velocity = 80 # rps
    self.request.acceleration = 400 # rot/s^2
    self.request.jerk = 4000 # rot/s^3

# set target position to 100 rotations
self.talonfx.set_control(self.request.with_position(100))
```

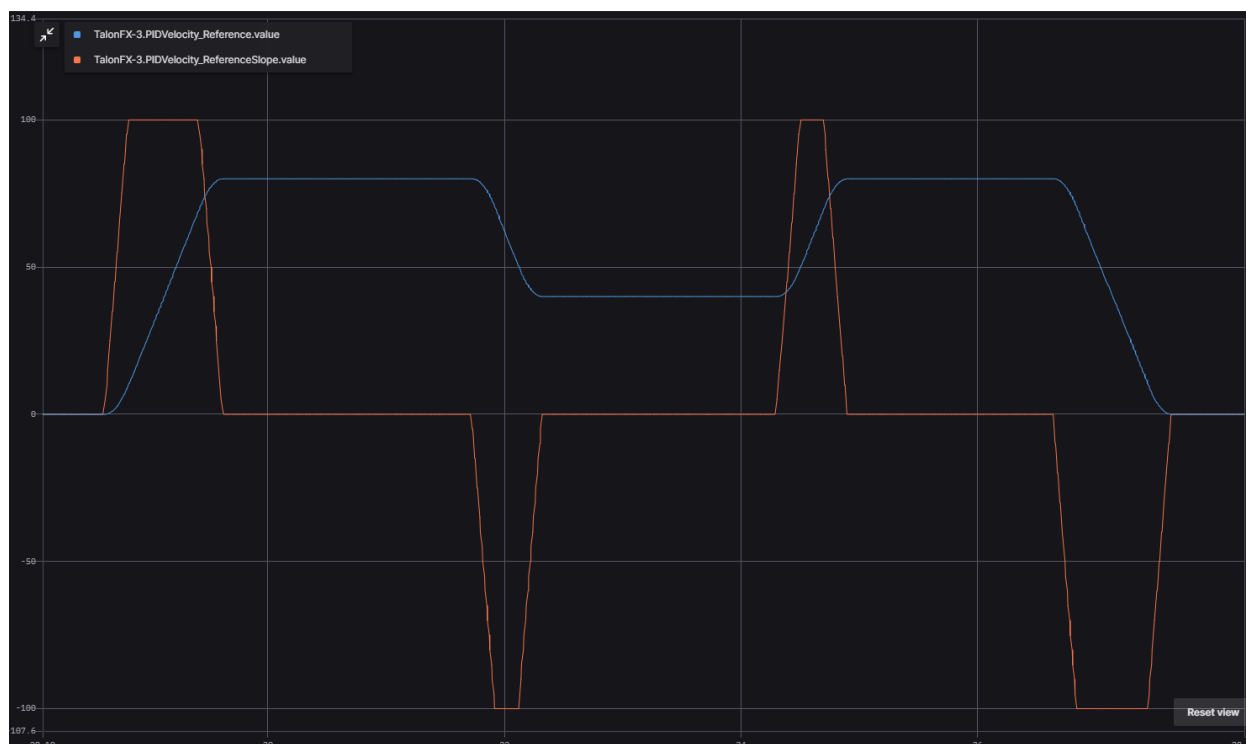
Motion Magic® Velocity

The Talon FX also supports onboard velocity motion profiling using Motion Magic® Velocity. When using Motion Magic® Velocity, the motor will ramp to a target velocity using a trapezoidal acceleration profile that honors the specified acceleration and optional jerk.

The benefits of this control mode over “simple” PID velocity closed-looping are:

- Control of the mechanism throughout the entire motion (as opposed to racing to the end target velocity)
- Control of the mechanism’s inertia to ensure smooth transitions between setpoints
- 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 velocity.



The following parameters must be set when controlling using Motion Magic® Velocity

- Acceleration - controls acceleration and deceleration rates during the beginning and end of motion
- Jerk (optional) - controls jerk, which is the derivative of acceleration

Using Motion Magic® Velocity in API

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

The Motion Magic® Velocity jerk and acceleration can be *configured in code* using a Motion-MagicConfigs (Java, C++, Python) object.

In Motion Magic® Velocity, 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_a - output per unit of target acceleration (output/(rps/s))
- 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
var talonFXConfigs = new TalonFXConfiguration();

// set slot 0 gains
var slot0Configs = talonFXConfigs.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

// set Motion Magic Velocity settings
var motionMagicConfigs = talonFXConfigs.MotionMagic;
motionMagicConfigs.MotionMagicAcceleration = 400; // Target acceleration of 400 rps/s
↳ (0.25 seconds to max)
motionMagicConfigs.MotionMagicJerk = 4000; // Target jerk of 4000 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.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
slot0Configs.kP = 0.11; // An error of 1 rps results in 0.11 V output
slot0Configs.kI = 0; // no output for integrated error
slot0Configs.kD = 0; // no output for error derivative

// set Motion Magic Velocity settings
auto& motionMagicConfigs = talonFXConfigs.MotionMagic;
motionMagicConfigs.MotionMagicAcceleration = 400; // Target acceleration of 400 rps/s
↳ (0.25 seconds to max)
motionMagicConfigs.MotionMagicJerk = 4000; // Target jerk of 4000 rps/s/s (0.1
↳ seconds)

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

Python

```
# in init function
talonfx_configs = configs.TalonFXConfiguration()

# set slot 0 gains
slot0_configs = talonfx_configs.slot0
slot0_configs.k_s = 0.25 # Add 0.25 V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_a = 0.01 # An acceleration of 1 rps/s requires 0.01 V output
slot0_configs.k_p = 0.11 # An error of 1 rps results in 0.11 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0 # no output for error derivative

# set Motion Magic Velocity settings
motion_magic_configs = talonfx_configs.motion_magic
motion_magic_configs.motion_magic_acceleration = 400 # Target acceleration of 400 rps/
↳s (0.25 seconds to max)
motion_magic_configs.motion_magic_jerk = 4000 # Target jerk of 4000 rps/s/s (0.1
↳seconds)

self.talonfx.configurator.apply(talonfx_configs)
```

Tip: Motion Magic® Velocity supports modifying acceleration and jerk on the fly (requires firmware version 24.0.6.0 or newer).

Once the gains are configured, the Motion Magic® Velocity request can be sent to the TalonFX.

The Motion Magic® Velocity request has an Acceleration parameter that can be used to override the profile acceleration during motion. If the Acceleration parameter is left 0, the acceleration config will be used instead.

The control request object also 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 Velocity request, voltage output
final MotionMagicVelocityVoltage m_request = new MotionMagicVelocityVoltage(0);

if (m_joy.getAButton()) {
    // while the joystick A button is held, use a slower acceleration
    m_request.Acceleration = 100; // rot/s^2
} else {
    // otherwise, fall back to the config
    m_request.Acceleration = 0;
}

// set target velocity to 80 rps
m_talonFX.setControl(m_request.withVelocity(80));
```

C++

```
// create a Motion Magic Velocity request, voltage output
controls::MotionMagicVelocityVoltage m_request{0_tps};

if (m_joy.GetAButton()) {
    // while the joystick A button is held, use a slower acceleration
    m_request.Acceleration = 100_tr_per_s_sq;
} else {
    // otherwise, fall back to the config
    m_request.Acceleration = 0_tr_per_s_sq;
}

// set target velocity to 80 rps
m_talonFX.SetControl(m_request.WithVelocity(80_tps));
```

Python

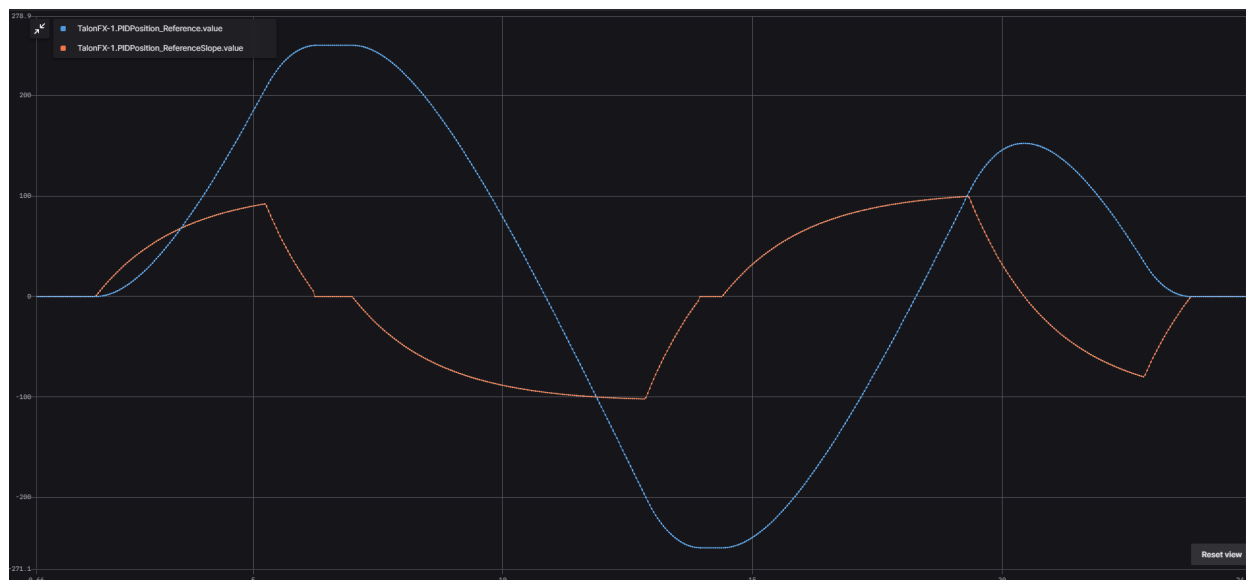
```
# create a Motion Magic Velocity request, voltage output
self.request = controls.MotionMagicVelocityVoltage(0)

if self.joy.getAButton():
    # while the joystick A button is held, use a slower acceleration
    self.request.acceleration = 100 # rot/s^2
else:
    # otherwise, fall back to the config
    self.request.acceleration = 0

# set target velocity to 80 rps
self.talonfx.set_control(self.request.with_velocity(80))
```

Motion Magic® Expo

Whereas traditional Motion Magic® generates a trapezoidal or S-Curve profile, Motion Magic® Expo generates an exponential profile. This allows the profile to best match the system dynamics, reducing both overshoot and time to target compared to a trapezoidal profile.



Motion Magic® Expo uses the kV and kA characteristics of the system, as well as an optional cruise velocity. The Motion Magic® Expo kV and kA configs are separate from the slot gain configs, as they may use different units and have different behaviors.

The Motion Magic® Expo kV represents the voltage required to maintain a given velocity and is in units of Volts/rps. Dividing the supply voltage by kV results in the maximum velocity of the profile. As a result, when supply voltage is fixed, a **higher profile kV** results in a **lower profile velocity**. Unlike with gain slots, it is safer to start from a higher kV than what is ideal.

The Motion Magic® Expo kA represents the voltage required to apply a given acceleration and is in units of Volts/(rps/s). Dividing the supply voltage by kA results in the maximum acceleration of the profile from 0. As a result, when supply voltage is fixed, a **higher profile kA** results in a **lower profile acceleration**. Unlike with gain slots, it is safer to start from a higher kA than what is ideal.

If the Motion Magic® cruise velocity is set to a non-zero value, the profile will only accelerate up to the cruise velocity. Otherwise, the profile will accelerate towards the maximum possible velocity based on the profile kV.

The following parameters must be set when controlling using Motion Magic® Expo:

- Expo kV - voltage required to maintain a given velocity, in V/rps
- Expo kA - voltage required to apply a given acceleration, in V/(rps/s)
- Cruise Velocity (optional) - peak velocity of the profile; set to 0 to target the system's max velocity

Using Motion Magic® Expo in API

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

The Motion Magic® Expo kV, kA, and cruise velocity can be *configured in code* using a MotionMagicConfigs (Java, C++, Python) object.

Important: Unlike the gain slots, the MotionMagicExpo_kV and MotionMagicExpo_kA configs are always in output units of Volts.

In Motion Magic® Expo, 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_a - output per unit of target acceleration (output/(rps/s))
- 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.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 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 Expo settings
var motionMagicConfigs = talonFXConfigs.MotionMagic;
motionMagicConfigs.MotionMagicCruiseVelocity = 0; // Unlimited cruise velocity
motionMagicConfigs.MotionMagicExpo_kV = 0.12; // kV is around 0.12 V/rps
motionMagicConfigs.MotionMagicExpo_kA = 0.1; // Use a slower kA of 0.1 V/(rps/s)

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

C++

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

// set slot 0 gains
auto& slot0Configs = talonFXConfigs.Slot0;
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.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 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 Expo settings
auto& motionMagicConfigs = talonFXConfigs.MotionMagic;
motionMagicConfigs.MotionMagicCruiseVelocity = 0; // Unlimited cruise velocity
motionMagicConfigs.MotionMagicExpo_kV = 0.12; // kV is around 0.12 V/rps
motionMagicConfigs.MotionMagicExpo_kA = 0.1; // Use a slower kA of 0.1 V/(rps/s)

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

Python

```
# in init function
talonfx_configs = configs.TalonFXConfiguration()

# set slot 0 gains
slot0_configs = talonfx_configs.slot0
slot0_configs.k_s = 0.25 # Add 0.25 V output to overcome static friction
slot0_configs.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
slot0_configs.k_a = 0.01 # An acceleration of 1 rps/s requires 0.01 V output
slot0_configs.k_p = 4.8 # A position error of 2.5 rotations results in 12 V output
slot0_configs.k_i = 0 # no output for integrated error
slot0_configs.k_d = 0.1 # A velocity error of 1 rps results in 0.1 V output

# set Motion Magic Expo settings
motion_magic_configs = talonfx_configs.motion_magic
motion_magic_configs.motion_magic_cruise_velocity = 0 # Unlimited cruise velocity
motion_magic_configs.motion_magic_expo_k_v = 0.12 # kV is around 0.12 V/rps
motion_magic_configs.motion_magic_expo_k_a = 0.1 # Use a slower kA of 0.1 V/(rps/s)

self.talonfx.configurator.apply(talonfx_configs)
```

Tip: Motion Magic® Expo supports modifying cruise velocity, kV, and kA on the fly.

Once the gains are configured, the Motion Magic® Expo 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 Expo request, voltage output
final MotionMagicExpoVoltage m_request = new MotionMagicExpoVoltage(0)

// set target position to 100 rotations
m_talonFX.setControl(m_request.withPosition(100));
```

C++

```
// create a Motion Magic Expo request, voltage output
controls::MotionMagicExpoVoltage m_request{0_tr}

// set target position to 100 rotations
m_talonFX.SetControl(m_request.WithPosition(100_tr));
```

Python

```
# create a Motion Magic Expo request, voltage output
self.request = controls.MotionMagicExpoVoltage(0)

# set target position to 100 rotations
self.talonfx.set_control(self.request.with_position(100))
```

14.1.6 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++](#), [Python](#)).

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 = u
↳ signals::FeedbackSensorSourceValue::RemoteCANcoder;

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

Python

```
fx_cfg = configs.TalonFXConfiguration()
fx_cfg.feedback.feedback_remote_sensor_id = self.cancoder.device_id
fx_cfg.feedback.feedback_sensor_source = signals.FeedbackSensorSourceValue.REMOTE_
↳ CANCODER

self.talonfx.configurator.apply(fx_cfg)
```

FusedCANcoder

Important: This feature requires the device to be *Pro licensed*. When unlicensed, the TalonFX will fall back to RemoteCANcoder and trip the UsingFusedCANcoderWhileUnlicensed fault.

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();
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);

```

Python

```

cc_cfg = configs.CANcoderConfiguration()
cc_cfg.magnet_sensor.absolute_sensor_range = signals.AbsoluteSensorRangeValue.SIGNED_
↪ PLUS_MINUS_HALF
cc_cfg.magnet_sensor.sensor_direction = signals.SensorDirectionValue.COUNTER_
↪ CLOCKWISE_POSITIVE
cc_cfg.magnet_sensor.magnet_offset = 0.4
self.cc.configurator.apply(cc_cfg)

fx_cfg = configs.TalonFXConfiguration()
fx_cfg.feedback.feedback_remote_sensor_id = self.cc.device_id
fx_cfg.feedback.feedback_sensor_source = signals.FeedbackSensorSourceValue.FUSED_
↪ CANCODER

```

(continues on next page)

(continued from previous page)

```
fx_cfg.feedback.sensor_to_mechanism_ratio = 1.0
fx_cfg.feedback.rotor_to_sensor_ratio = 12.8

self.fx.configurator.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;
```

Python

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

print("FX Position: " + fx_pos.value)
print("CANcoder Position: " + cc_pos.value)
```

SyncCANcoder

Important: This feature requires the device to be *Pro licensed*. When unlicensed, the TalonFX will fall back to RemoteCANcoder and trip the UsingFusedCANcoderWhileUnlicensed fault.

SyncCANcoder allows users to synchronize the TalonFX's internal rotor sensor against the remote CANcoder, but continue to use the rotor sensor for all closed loop control. TalonFX will continue to monitor the remote CANcoder and report if its internal position differs significantly from the reported position or if the remote CANcoder disappears from the bus.

Users may want SyncCANcoder instead of FusedCANcoder if there is risk that the sensor can fail in a way that the sensor is still reporting "good" data, but the data does not match the mechanism, such as if the entire sensor mount assembly breaks off. Using SyncCANcoder over FusedCANcoder will not benefit from backlash compensation, as the CANcoder position is not continually fused in.

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

Java

```
/* Configure CANcoder to zero the magnet appropriately */
CANcoderConfiguration cc_cfg = new CANcoderConfiguration();
cc_cfg.MagnetSensor.AbsoluteSensorRange = AbsoluteSensorRangeValue.Signed_
↳ PlusMinusHalf;
cc_cfg.MagnetSensor.SensorDirection = SensorDirectionValue.CounterClockwise_Positive;
cc_cfg.MagnetSensor.MagnetOffset = 0.4;
m_cc.getConfigurator().apply(cc_cfg);

TalonFXConfiguration fx_cfg = new TalonFXConfiguration();
fx_cfg.Feedback.FeedbackRemoteSensorID = m_cc.getDeviceID();
fx_cfg.Feedback.FeedbackSensorSource = FeedbackSensorSourceValue.SyncCANcoder;
fx_cfg.Feedback.SensorToMechanismRatio = 1.0;
fx_cfg.Feedback.RotorToSensorRatio = 12.8;

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

C++

```
/* Configure CANcoder to zero the magnet appropriately */
configs::CANcoderConfiguration cc_cfg{};
cc_cfg.MagnetSensor.AbsoluteSensorRange = signals::AbsoluteSensorRangeValue::Signed_
↳ PlusMinusHalf;
cc_cfg.MagnetSensor.SensorDirection = signals::SensorDirectionValue::CounterClockwise_
↳ Positive;
cc_cfg.MagnetSensor.MagnetOffset = 0.4;
m_cc.GetConfigurator().Apply(cc_cfg);

configs::TalonFXConfiguration fx_cfg{};
fx_cfg.Feedback.FeedbackRemoteSensorID = m_cc.GetDeviceID();
fx_cfg.Feedback.FeedbackSensorSource =
↳ signals::FeedbackSensorSourceValue::SyncCANcoder;
fx_cfg.Feedback.SensorToMechanismRatio = 1.0;
fx_cfg.Feedback.RotorToSensorRatio = 12.8;

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

Python

```
cc_cfg = configs.CANcoderConfiguration()
cc_cfg.magnet_sensor.absolute_sensor_range = signals.AbsoluteSensorRangeValue.SIGNED_
↳ PLUS_MINUS_HALF
cc_cfg.magnet_sensor.sensor_direction = signals.SensorDirectionValue.COUNTER_
↳ CLOCKWISE_POSITIVE
cc_cfg.magnet_sensor.magnet_offset = 0.4
self.cc.configurator.apply(cc_cfg)
```

(continues on next page)

(continued from previous page)

```
fx_cfg = configs.TalonFXConfiguration()
fx_cfg.feedback.feedback_remote_sensor_id = self.cc.device_id
fx_cfg.feedback.feedback_sensor_source = signals.FeedbackSensorSourceValue.SYNC_
    ↳ CANCELER
fx_cfg.feedback.sensor_to_mechanism_ratio = 1.0
fx_cfg.feedback.rotor_to_sensor_ratio = 12.8

self.fx.configurator.apply(fx_cfg)
```


This section serves to provide API usage of mechanisms supported by Phoenix 6.

Swerve Documentation on the Phoenix 6 Swerve API

15.1 Swerve Overview

Important: Swerve functionality is only available for FRC users and is limited to Java at this time.

New in 2024 is a high performance swerve API. This API simplifies the boilerplate necessary for swerve and maximizes performance.

This section focuses on utilizing the Swerve API and configuring it correctly. Tuner X supports a [swerve project creator](#) that greatly simplifies this process and removes common error cases.

- Small API surface, easily debuggable
 - Build robot characteristics using SwerveModuleConstants (Java) and SwerveDrivetrainConstants (Java).
 - Integrates cleanly into WPILib [command-based](#) framework using CommandSwerveDrivetrain (from our examples or Tuner X).
 - Telemetrize directly in the odometry loop using the registerTelemetry() (Java) lambda.
 - Supports handling the swerve state via SwerveRequest (Java).
 - * Supports robot-centric, field-centric and field-centric facing angle.
 - * Supports common scenarios such as X mode (point all modules toward the center of the robot).
- Simulation
 - Swerve simulation focuses on usability, and as such isn't perfectly accurate to a real robot.

- Simply call `updateSimState (Java)` in `simulationPeriodic() (Java)`.
- Performance
 - Odometry is updated synchronously with the motor controllers.
 - Odometry is received as fast as possible using a separate thread.
 - Combine with [Phoenix Pro](#) and a [CANivore](#) with *timesync* for improved performance.

Note: Simulation boilerplate is automatically handled when generating a robot project using Tuner X.

15.1.1 Hardware Requirements

Utilizing the swerve API requires that the robot drivetrain is composed of supported Phoenix 6 devices. At a minimum, these requirements are:

- 8 TalonFX motor controllers (4 steer, 4 drive)
- 4 CANcoders
- 1 Pigeon 2.0

15.1.2 Overview on the API

Simple usage of the API is comprised of 4 core classes:

- `SwerveDrivetrainConstants (Java)`
 - This class handles characteristics of the robot that are not module specific. e.g. CAN bus, Pigeon 2 ID, whether FD is enabled or not.
- `SwerveModuleConstantsFactory (Java)`
 - Factory class that is used to instantiate `SwerveModuleConstants` for each module on the robot.
- `SwerveModuleConstants (Java)`
 - Represents the characteristics for a given module.
- `SwerveDrivetrain (Java)`
 - Created using `SwerveDrivetrainConstants` and a `SwerveModuleConstants` for each module, this is used to control the swerve drivetrain.

Usage of these classes is available in the following articles in this section.

Swerve Builder API

To simplify the API surface, both **builder** and **factory** paradigms are used. Users create a `SwerveDrivetrain` by first defining the global drivetrain characteristics and then each module characteristics.

Defining Drivetrain Characteristics

Drivetrain, in this instance, refers to the `SwerveDrivetrainConstants` class (Java). This class defines characteristics that are not module specific. Mandatory parameters include `.withPigeon2Id()`.

SwerveDrivetrainConstants Example

Java

```
private static final SwerveDrivetrainConstants DrivetrainConstants = new SwerveDrivetrainConstants()
    .withPigeon2Id(kPigeonId)
    .withCANbusName(kCANbusName);
```

Defining Module Characteristics

The typical FRC drivetrain includes 4 identical modules. To simplify module creation, there exists a `SwerveModuleConstantsFactory` (Java) class to simplify module creation.

Mandatory parameters for this factory are:

- `withDriveMotorGearRatio()` - Gearing between the drive motor output shaft and the wheel.
- `withSteerMotorGearRatio()` - Gearing between the steer motor output shaft and the azimuth gear.
- `withWheelRadius()` - Radius of the wheel in inches.
- `withSteerMotorGains()` - Instance of `Slot0Configs`, closed-loop gains for the steering motor.
- `withDriveMotorGains()` - Instance of `Slot0Configs`, closed-loop gains for the drive motor.
- `withSteerMotorClosedLoopOutput()` - The `ClosedLoopOutputType` to use for steer motor closed-loop control requests.
- `withDriveMotorClosedLoopOutput()` - The `ClosedLoopOutputType` to use for drive motor closed-loop control requests.
- `withSpeedAt12VoltsMps()` - Required for open-loop control, theoretical free speed (m/s) at 12v applied output.
- `withFeedbackSource()` - Instance of `SteerFeedbackType`. Typically `FusedCANcoder` (requires Pro) or `RemoteCANcoder`.

For functional simulation, the following additional parameters **must** be defined.

- `withSteerInertia()`
- `withDriveInertia()`

For a full reference of the available functions, see the API documentation of `SwerveModuleConstantsFactory` ([Java](#)).

SwerveModuleConstantsFactory Example

Java

```
private static final SwerveModuleConstantsFactory ConstantCreator = new SwerveModuleConstantsFactory()  
    .withDriveMotorGearRatio(kDriveGearRatio)  
    .withSteerMotorGearRatio(kSteerGearRatio)  
    .withWheelRadius(kWheelRadiusInches)  
    .withSlipCurrent(kSlipCurrentA)  
    .withSteerMotorGains(steerGains)  
    .withDriveMotorGains(driveGains)  
    .withSteerMotorClosedLoopOutput(steerClosedLoopOutput)  
    .withDriveMotorClosedLoopOutput(driveClosedLoopOutput)  
    .withSpeedAt12VoltsMps(kSpeedAt12VoltsMps)  
    .withSteerInertia(kSteerInertia)  
    .withDriveInertia(kDriveInertia)  
    .withFeedbackSource(SteerFeedbackType.FusedCANcoder)  
    .withCouplingGearRatio(kCoupleRatio)  
    .withSteerMotorInverted(kSteerMotorReversed);
```

Additional Constants

In the previous section, several optional constants are defined. These constants are not mandatory for usable swerve, but they can greatly increase swerve controllability and accuracy.

CouplingGearRatio

The ratio at which the output wheel rotates when the azimuth spins. In a traditional swerve module, this is the inverse of the 1st stage of the drive motor.

To manually determine the coupling ratio, lock the drive wheel in-place, then rotate the azimuth three times. Observe the number of rotations reported by the drive motor. The coupling ratio will be $driveRotations/3$, or $driveRotations/azimuthRotations$.

SlipCurrent

This is the amount of stator current the drive motors can apply without slippage. This can be found by placing the robot against a solid wall and slowly increase the output voltage. As the output voltage increases, *plot* the drive wheel velocity and stator current. Observe when the drive wheel velocity starts to rise (wheel is slipping) and at what stator current this begins.

Building the Swerve Module Constants

SwerveModuleConstants (Java) can be derived, or created, from the previous SwerveModuleConstantsFactory. A typical swerve drivetrain consists of four identical modules: Front Left, Front Right, Back Left, Back Right. While these modules can be instantiated directly (only really useful if the modules have different physical characteristics), the modules can also be created by calling createModuleConstants() with the aforementioned factory.

Calling createModuleConstants() takes the following arguments:

- Steer Motor ID
- Drive Motor ID
- Steer Encoder ID
- Steer Encoder Offset
- X position in meters
- Y position in meters
- Whether the drive motor is reversed

Note: The X and Y position of the modules is measured from the center point of the robot along the X and Y axes, respectively. These values use the same coordinate system as Translation2d (Java), where forward is positive X and left is positive Y.

SwerveModuleConstants Example

Java

```
private static final SwerveModuleConstants FrontLeft = ConstantCreator.  
    ↪ createModuleConstants(  
        kFrontLeftSteerMotorId, kFrontLeftDriveMotorId, kFrontLeftEncoderId, ↪  
        ↪ kFrontLeftEncoderOffset, Units.inchesToMeters(kFrontLeftXPosInches), Units.  
        ↪ inchesToMeters(kFrontLeftYPosInches), kInvertLeftSide);  
private static final SwerveModuleConstants FrontRight = ConstantCreator.  
    ↪ createModuleConstants(  
        kFrontRightSteerMotorId, kFrontRightDriveMotorId, kFrontRightEncoderId, ↪  
        ↪ kFrontRightEncoderOffset, Units.inchesToMeters(kFrontRightXPosInches), Units.  
        ↪ inchesToMeters(kFrontRightYPosInches), kInvertRightSide);  
private static final SwerveModuleConstants BackLeft = ConstantCreator.  
    ↪ createModuleConstants(  
        kBackLeftSteerMotorId, kBackLeftDriveMotorId, kBackLeftEncoderId, ↪  
        ↪ kBackLeftEncoderOffset, Units.inchesToMeters(kBackLeftXPosInches), Units.  
        ↪ inchesToMeters(kBackLeftYPosInches), kInvertLeftSide);  
private static final SwerveModuleConstants BackRight = ConstantCreator.  
    ↪ createModuleConstants(  
        kBackRightSteerMotorId, kBackRightDriveMotorId, kBackRightEncoderId, ↪  
        ↪ kBackRightEncoderOffset, Units.inchesToMeters(kBackRightXPosInches), Units.  
        ↪ inchesToMeters(kBackRightYPosInches), kInvertRightSide);
```

Building the SwerveDrivetrain

Note: CommandSwerveDrivetrain is a version created by the Tuner X *Swerve Project Generator* that implements Subsystem (Java) for easy command-based integration.

SwerveDrivetrain (Java) is the class that handles odometry, configuration and control of the drivetrain. The constructor for this class takes the previous SwerveDrivetrainConstants and a list of SwerveModuleConstants.

Java

```
public static final CommandSwerveDrivetrain DriveTrain = new
↳ CommandSwerveDrivetrain(DrivetrainConstants, FrontLeft,
    FrontRight, BackLeft, BackRight);
```

Utilization of SwerveDrivetrain consists of SwerveRequests that define the state of the drivetrain. For full details of using SwerveRequests to control your swerve, see *Swerve Requests*.

Full Example

Java

```
// Both sets of gains need to be tuned to your individual robot.

// The steer motor uses any SwerveModule.SteerRequestType control request with the
// output type specified by SwerveModuleConstants.SteerMotorClosedLoopOutput
private static final Slot0Configs steerGains = new Slot0Configs()
    .withKP(100).withKI(0).withKD(0.2)
    .withKS(0).withKV(1.5).withKA(0);
// When using closed-loop control, the drive motor uses the control
// output type specified by SwerveModuleConstants.DriveMotorClosedLoopOutput
private static final Slot0Configs driveGains = new Slot0Configs()
    .withKP(3).withKI(0).withKD(0)
    .withKS(0).withKV(0).withKA(0);

// The closed-loop output type to use for the steer motors;
// This affects the PID/FF gains for the steer motors
private static final ClosedLoopOutputType steerClosedLoopOutput =
↳ ClosedLoopOutputType.Voltage;
// The closed-loop output type to use for the drive motors;
// This affects the PID/FF gains for the drive motors
private static final ClosedLoopOutputType driveClosedLoopOutput =
↳ ClosedLoopOutputType.TorqueCurrentFOC;

// The stator current at which the wheels start to slip;
// This needs to be tuned to your individual robot
private static final double kSlipCurrentA = 300.0;

// Theoretical free speed (m/s) at 12v applied output;
// This needs to be tuned to your individual robot
private static final double kSpeedAt12VoltsMps = 6.0;
```

(continues on next page)

(continued from previous page)

```

// Every 1 rotation of the azimuth results in kCoupleRatio drive motor turns;
// This may need to be tuned to your individual robot
private static final double kCoupleRatio = 3.5;

private static final double kDriveGearRatio = 6.75;
private static final double kSteerGearRatio = 15.43;
private static final double kWheelRadiusInches = 2;

private static final boolean kSteerMotorReversed = false;
private static final boolean kInvertLeftSide = false;
private static final boolean kInvertRightSide = true;

private static final String kCANbusName = "drivetrain";
private static final int kPigeonId = 1;

// These are only used for simulation
private static double kSteerInertia = 0.00001;
private static double kDriveInertia = 0.001;

private static final SwerveDrivetrainConstants DrivetrainConstants = new
↳ SwerveDrivetrainConstants()
    .withPigeon2Id(kPigeonId)
    .withCANbusName(kCANbusName);

private static final SwerveModuleConstantsFactory ConstantCreator = new
↳ SwerveModuleConstantsFactory()
    .withDriveMotorGearRatio(kDriveGearRatio)
    .withSteerMotorGearRatio(kSteerGearRatio)
    .withWheelRadius(kWheelRadiusInches)
    .withSlipCurrent(kSlipCurrentA)
    .withSteerMotorGains(steerGains)
    .withDriveMotorGains(driveGains)
    .withSteerMotorClosedLoopOutput(steerClosedLoopOutput)
    .withDriveMotorClosedLoopOutput(driveClosedLoopOutput)
    .withSpeedAt12VoltsMps(kSpeedAt12VoltsMps)
    .withSteerInertia(kSteerInertia)
    .withDriveInertia(kDriveInertia)
    .withFeedbackSource(SteerFeedbackType.FusedCANcoder)
    .withCouplingGearRatio(kCoupleRatio)
    .withSteerMotorInverted(kSteerMotorReversed);

// Front Left
private static final int kFrontLeftDriveMotorId = 1;
private static final int kFrontLeftSteerMotorId = 0;
private static final int kFrontLeftEncoderId = 0;
private static final double kFrontLeftEncoderOffset = -0.75;

private static final double kFrontLeftXPosInches = 10.5;
private static final double kFrontLeftYPosInches = 10.5;

// Front Right
private static final int kFrontRightDriveMotorId = 3;
private static final int kFrontRightSteerMotorId = 2;
private static final int kFrontRightEncoderId = 1;

```

(continues on next page)

(continued from previous page)

```

private static final double kFrontRightEncoderOffset = -0.75;

private static final double kFrontRightXPosInches = 10.5;
private static final double kFrontRightYPosInches = -10.5;

// Back Left
private static final int kBackLeftDriveMotorId = 5;
private static final int kBackLeftSteerMotorId = 4;
private static final int kBackLeftEncoderId = 2;
private static final double kBackLeftEncoderOffset = -0.75;

private static final double kBackLeftXPosInches = -10.5;
private static final double kBackLeftYPosInches = 10.5;

// Back Right
private static final int kBackRightDriveMotorId = 7;
private static final int kBackRightSteerMotorId = 6;
private static final int kBackRightEncoderId = 3;
private static final double kBackRightEncoderOffset = -0.75;

private static final double kBackRightXPosInches = -10.5;
private static final double kBackRightYPosInches = -10.5;

private static final SwerveModuleConstants FrontLeft = ConstantCreator.
↳ createModuleConstants(
    kFrontLeftSteerMotorId, kFrontLeftDriveMotorId, kFrontLeftEncoderId,
↳ kFrontLeftEncoderOffset, Units.inchesToMeters(kFrontLeftXPosInches), Units.
↳ inchesToMeters(kFrontLeftYPosInches), kInvertLeftSide);
private static final SwerveModuleConstants FrontRight = ConstantCreator.
↳ createModuleConstants(
    kFrontRightSteerMotorId, kFrontRightDriveMotorId, kFrontRightEncoderId,
↳ kFrontRightEncoderOffset, Units.inchesToMeters(kFrontRightXPosInches), Units.
↳ inchesToMeters(kFrontRightYPosInches), kInvertRightSide);
private static final SwerveModuleConstants BackLeft = ConstantCreator.
↳ createModuleConstants(
    kBackLeftSteerMotorId, kBackLeftDriveMotorId, kBackLeftEncoderId,
↳ kBackLeftEncoderOffset, Units.inchesToMeters(kBackLeftXPosInches), Units.
↳ inchesToMeters(kBackLeftYPosInches), kInvertLeftSide);
private static final SwerveModuleConstants BackRight = ConstantCreator.
↳ createModuleConstants(
    kBackRightSteerMotorId, kBackRightDriveMotorId, kBackRightEncoderId,
↳ kBackRightEncoderOffset, Units.inchesToMeters(kBackRightXPosInches), Units.
↳ inchesToMeters(kBackRightYPosInches), kInvertRightSide);

public static final CommandSwerveDrivetrain DriveTrain = new
↳ CommandSwerveDrivetrain(DrivetrainConstants, FrontLeft,
    FrontRight, BackLeft, BackRight);

```

Swerve Requests

Controlling the drivetrain is done using `setControl(SwerveRequest request)` (Java) which takes a given `SwerveRequest` (Java). There are multiple pre-defined `SwerveRequest` implementations, or users can define their own (only recommended for advanced scenarios).

Applying a Request

Requests are instantiated once and then mutated using various `withX` functions. In the below example, a `FieldCentric` (Java) request is created and given values from a joystick.

Java

```
private final SwerveDrivetrain m_drivetrain = TunerConstants.DriveTrain;
private final SwerveRequest.FieldCentric m_driveRequest = new SwerveRequest.
    ↪FieldCentric()
    ↪a 10% deadband
        .withDeadband(MaxSpeed * 0.1).withRotationalDeadband(MaxAngularRate * 0.1) // Add
        .withDriveRequestType(DriveRequestType.OpenLoopVoltage)
        .withSteerRequestType(SteerRequestType.MotionMagicExpo);

private final XboxController m_joystick = new XboxController(0);

@Override
public void teleopPeriodic() {
    m_drivetrain.setControl(
        m_driveRequest.withVelocityX(-joystick.getLeftY())
            .withVelocityY(-joystick.getLeftX())
            .withRotationalRate(-joystick.getRightX())
    );
}
```

Tip: Users can optionally make their own `SwerveRequests` by implementing the `SwerveRequest` interface.

Swerve Simulation

The API supports a functionality focused simulation. This means that the simulation API assumes that the swerve drive is perfect (no scrub and no wheel slip). Additionally, it assumes the inertia of the steer module and the drive modules. Robot-wide, rotational and translational inertia is not accounted for.

To update the simulated swerve robot state, ensure `m_drivetrain.updateSimState()` is called in `simulationPeriodic()`, where `m_drivetrain` is a `SwerveDrivetrain`. The typical update rate of a robot project is 20 ms (0.020 seconds), and `RobotController.getBatteryVoltage()` (Java) can be used to get the simulated battery voltage.

Java

```
@Override
public void simulationPeriodic() {
    /* Assume 20ms update rate, get battery voltage from WPILib */
    updateSimState(0.020, RobotController.getBatteryVoltage());
}
```

Important: When utilizing CommandSwerveDrivetrain (from our examples or Tuner X), this is handled in the subsystem simulationPeriodic instead.

Simulation FAQ

Q: My robot does not move in simulation

A: Verify that all gains are non-zero and that the steer/drive inertia is non-zero.

Q: My robot stutters when driving

A: Simulation uses predetermined constants that represent the drivetrain, as such, gains may be inaccurate compared to the real robot.

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.

16.1 Introduction to Simulation

Many CTR Electronics devices support high-fidelity simulation, allowing the simulated robot to match the behavior of the real robot hardware as closely as possible. This makes simulation a powerful tool to quickly diagnose and fix bugs in robot code without relying on access to hardware.

16.1.1 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**.

16.1.2 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

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

C++

```
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

```
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++

```
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 `RobotController.getBatteryVoltage()` (Java, C++) to take advantage of WPILib's `BatterySim` (Java, C++) API.

Java

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

C++

```
// 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

```
private final DCMotorSim m_motorSimModel =
    new DCMotorSim(DCMotor.getKrakenX60Foc(1), 1.0, 0.001);

public void simulationPeriodic() {
    var talonFXSim = m_talonFX.getSimState();

    // get the motor voltage of the TalonFX
    var motorVoltage = talonFXSim.getMotorVoltage();

    // use the motor voltage to calculate new position and velocity
    // using WPILib's DCMotorSim class for physics simulation
    m_motorSimModel.setInputVoltage(motorVoltage);
    m_motorSimModel.update(0.020); // assume 20 ms loop time

    // apply the new rotor position and velocity to the TalonFX;
    // note that this is rotor position/velocity (before gear ratios)
    talonFXSim.setRawRotorPosition(m_motorSimModel.getAngularPositionRotations());
    talonFXSim.setRotorVelocity(
        Units.radiansToRotations(m_motorSimModel.getAngularVelocityRadPerSec())
    );
}
```

C++

```
frc::sim::DCMotorSim m_motorSimModel{
    frc::DCMotor::KrakenX60FOC(1), 1.0, 0.001_kg_sq_m
};

void SimulationPeriodic()
{
    auto& talonFXSim = m_talonFX.GetSimState();

    // get the motor voltage of the TalonFX
    auto motorVoltage = talonFXSim.GetMotorVoltage();

    // use the motor voltage to calculate new position and velocity
    // using WPILib's DCMotorSim class for physics simulation
    m_motorSimModel.SetInputVoltage(motorVoltage);
    m_motorSimModel.Update(20_ms); // assume 20 ms loop time

    // apply the new rotor position and velocity to the TalonFX;
    // note that this is rotor position/velocity (before gear ratios)
    talonFXSim.SetRawRotorPosition(m_motorSimModel.GetAngularPosition());
    talonFXSim.SetRotorVelocity(m_motorSimModel.GetAngularVelocity());
}
```

16.1.3 High Fidelity CAN Bus Simulation

As a part of high-fidelity simulation, 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.

In unit tests, it may be useful to increase the update rate of status signals to avoid erroneous failures and minimize delays. The update rate can be modified for simulation by wrapping the *signal update frequency* in a `Utils.isSimulation()` (Java, C++) condition.

Java

```
if (Utils.isSimulation()) {  
    // set update rate to 1ms for unit tests  
    m_velocitySignal.setUpdateFrequency(1000);  
}
```

C++

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


WPILib Integration

Phoenix 6 API used as part of [WPILib robot projects](#) provides implementations of common WPILib interfaces that FRC teams use.

Note: While the Python API does support simulation, it currently does not hook into WPILib's simulation GUI at this time.

17.1 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

```
// 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)

```
void Robot::TeleopPeriodic() {  
    auto forward = -m_driverJoy.GetLeftY();  
    auto rot = -m_driverJoy.GetRightX();  
  
    m_diffDrive.ArcadeDrive(forward, rot);  
}
```

C++ (Header)

```
// 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};
```

17.1.1 Motor Safety

CTR Electronics supported actuators implement WPILib [Motor Safety](#). In addition to the normal *enable signal* of CTR Electronics actuators, Motor Safety will automatically disable the device according to the WPILib Motor Safety implementation.

17.1.2 Simulation

It's recommended that users set supply voltage to `RobotController.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.



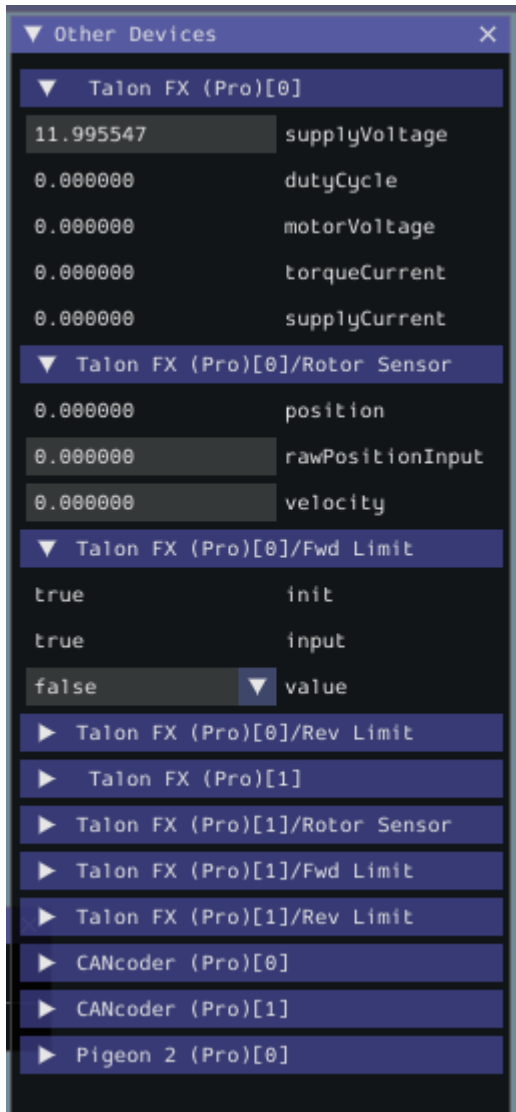
17.2 Gyro Integration

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.

17.2.1 Simulation

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



17.3 SysId Integration

System Identification, or commonly referred to as SysId, is the process of identifying the characteristics of a given [system](#).

This identification usually consists of:

1. Mechanism testing
 - In FRC, two tests are performed: Quasistatic and Dynamic.
2. Data collection
 - Position, Velocity, and MotorOutput samples are collected while the tests are running.

3. Data analysis

- Collected data is analyzed to calculate constants such as *PID* gains, slip current (maximum stator current), maximum robot velocity, etc.

Note: This documentation assumes that the user is utilizing a [command-based robot program](#).

17.3.1 Advantages of SignalLogger over DataLog

When collecting data for analysis, it's important to take into account several factors, such as:

- Impact of CAN latency
- Signals sent faster than the 20ms main robot loop
- Language data collection issues (such as Java garbage collection causing pauses in the log)

When users utilize the *Phoenix 6 signal logging API*, these issues are eliminated.

This section guides the user through characterizing a motor, converting hoot logs to a WPILib WPILOG for data analysis, and integrating gains for control. This section can also be used as a characterizing other mechanisms such as a *swerve* azimuth and drive motors.

Characterization begins with a functioning robot program. Users should have basic code for the mechanism already put together, and all *configs* in the FeedbackConfigs group should be applied. Any changes to the gear ratios and sensor source in FeedbackConfigs may require the user to recharacterize their mechanism.

Get started:

Plumbing & Running SysId

For the purpose of this documentation, we focus on the integration of Phoenix 6 and WPILib's SysId API to characterize common mechanisms. Detailed documentation on the SysId routines can be found [here](#).

To get started, users must construct a SysIdRoutine that defines a Config and Mechanism.

The Config constructor allows the users to define the voltage ramp rate, dynamic step voltage, characterization timeout, and a lambda that accepts the SysIdRoutineLog.State for logging. The lambda needs to be overridden to log the State string using the *Phoenix 6 Signal Logger*.

The Mechanism constructor takes a lambda accepts a Measure<Voltage>. This lambda is used to apply the voltage request to the motors during characterization, which can be done using a VoltageOut request. The second argument to the constructor is a logging callback; this is left null when using the Signal Logger, as all signals are logged automatically. The last parameter is a reference to this Subsystem.

Putting this all together results in the example shown below.

Java

```
private final TalonFX m_motor = new TalonFX(0);
private final VoltageOut m_voltReq = new VoltageOut(0.0);

private final SysIdRoutine m_sysIdRoutine =
    new SysIdRoutine(
        new SysIdRoutine.Config(
            null,           // Use default ramp rate (1 V/s)
            Volts.of(4),    // Reduce dynamic step voltage to 4 to prevent brownout
            null,           // Use default timeout (10 s)
            // Log state with Phoenix SignalLogger class
            (state) -> SignalLogger.writeString("state", state.toString())
        ),
        new SysIdRoutine.Mechanism(
            (volts) -> m_motor.setControl(m_voltReq.withOutput(volts.in(Volts))),
            null,
            this
        )
    );
```

C++

```
hardware::TalonFX m_motor{0};
controls::VoltageOut m_voltReq{0_V};

frc2::sysid::SysIdRoutine m_sysIdRoutine{
    frc2::sysid::Config{
        std::nullopt, // Use default ramp rate (1 V/s)
        4_V,          // Reduce dynamic step voltage to 4 to prevent brownout
        std::nullopt, // Use default timeout (10 s)
        // Log state with Phoenix SignalLogger class
        [](frc2::sysid::State state)
        {
            SignalLogger::WriteString("state",
↪ frc2::sysid::SysIdRoutineLog::StateEnumToString(state));
        }
    },
    frc2::sysid::Mechanism{
        [this](units::volt_t volts) { m_motor.SetControl(m_voltReq.WithOutput(volts)); }
↪,
        [](auto) {},
        this
    }
};
```

Python

```

self.motor = hardware.TalonFX(0)
self.voltage_req = controls.VoltageOut(0)

self.sys_id_routine = SysIdRoutine(
    SysIdRoutine.Config(
        # Use default ramp rate (1 V/s) and timeout (10 s)
        # Reduce dynamic voltage to 4 to prevent brownout
        stepVoltage = 4.0,
        # Log state with Phoenix SignalLogger class
        recordState = lambda state: SignalLogger.write_string("state", SysIdRoutineLog.
→ stateEnumToString(state))
    ),
    SysIdRoutine.Mechanism(
        lambda volts: self.motor.set_control(self.voltage_req.with_output(volts)),
        lambda log: None,
        self
    )
)

```

Now that the routine has been plumbed, the characterization commands need to be exposed from the subsystem.

Java

```

public Command sysIdQuasistatic(SysIdRoutine.Direction direction) {
    return m_sysIdRoutine.quasistatic(direction);
}

public Command sysIdDynamic(SysIdRoutine.Direction direction) {
    return m_sysIdRoutine.dynamic(direction);
}

```

C++

```

frc2::CommandPtr SysIdQuasistatic(frc2::sysid::Direction direction)
{
    return m_sysIdRoutine.Quasistatic(direction);
}

frc2::CommandPtr SysIdDynamic(frc2::sysid::Direction direction)
{
    return m_sysIdRoutine.Dynamic(direction);
}

```

Python

```
def sys_id_quasistatic(self, direction: SysIdRoutine.Direction) -> Command:
    return self.sys_id_routine.quasistatic(direction)

def sys_id_dynamic(self, direction: SysIdRoutine.Direction) -> Command:
    return self.sys_id_routine.dynamic(direction)
```

From there, the program can bind buttons to these commands in RobotContainer.

Java

```
m_joystick.leftBumper().onTrue(Commands.runOnce(SignalLogger::start));
m_joystick.rightBumper().onTrue(Commands.runOnce(SignalLogger::stop));

/*
 * Joystick Y = quasistatic forward
 * Joystick A = quasistatic reverse
 * Joystick B = dynamic forward
 * Joystick X = dyanmic reverse
 */
m_joystick.y().whileTrue(m_mechanism.sysIdQuasistatic(SysIdRoutine.Direction.
    ↪ kForward));
m_joystick.a().whileTrue(m_mechanism.sysIdQuasistatic(SysIdRoutine.Direction.
    ↪ kReverse));
m_joystick.b().whileTrue(m_mechanism.sysIdDynamic(SysIdRoutine.Direction.kForward));
m_joystick.x().whileTrue(m_mechanism.sysIdDynamic(SysIdRoutine.Direction.kReverse));
```

C++

```
m_joystick.LeftBumper().OnTrue(frc2::cmd::RunOnce(SignalLogger::Start));
m_joystick.RightBumper().OnTrue(frc2::cmd::RunOnce(SignalLogger::Stop));

/*
 * Joystick Y = quasistatic forward
 * Joystick A = quasistatic reverse
 * Joystick B = dynamic forward
 * Joystick X = dynamic reverse
 */
m_joystick.Y().WhileTrue(m_mechanism.
    ↪ SysIdQuasistatic(frc2::sysid::Direction::kForward));
m_joystick.A().WhileTrue(m_mechanism.
    ↪ SysIdQuasistatic(frc2::sysid::Direction::kReverse));
m_joystick.B().WhileTrue(m_mechanism.SysIdDynamic(frc2::sysid::Direction::kForward));
m_joystick.X().WhileTrue(m_mechanism.SysIdDynamic(frc2::sysid::Direction::kReverse));
```

Python

```
self.joystick.leftBumper().onTrue(cmd.runOnce(SignalLogger.start))
self.joystick.rightBumper().onTrue(cmd.runOnce(SignalLogger.stop))

# Joystick Y = quasistatic forward
# Joystick A = quasistatic reverse
# Joystick B = dynamic forward
# Joystick X = dynamic reverse
self.joystick.y().whileTrue(self.mechanism.sys_id_quasistatic(SysIdRoutine.Direction.
    ↪ kForward))
self.joystick.a().whileTrue(self.mechanism.sys_id_quasistatic(SysIdRoutine.Direction.
    ↪ kReverse))
self.joystick.b().whileTrue(self.mechanism.sys_id_dynamic(SysIdRoutine.Direction.
    ↪ kForward))
self.joystick.x().whileTrue(self.mechanism.sys_id_dynamic(SysIdRoutine.Direction.
    ↪ kReverse))
```

All four tests must be run and captured in a single log file. As a result, it is important that the user starts the Signal Logger before running the tests and stops the Signal Logger after all tests have been completed. This will ensure the log is not cluttered with data from other actions such as driving the robot to an open area.

Note: Consult the [WPILib documentation](#) for additional details on mechanism characterization.

Before Characterization

There are a couple of important things to consider before running the characterization tests.

Characterization Can Be Dangerous:

Danger: Always use caution when mechanisms are moving and ensure that the robot can be disabled swiftly at any time!

- Since characterization applies a scaling (quasistatic) or constant (dynamic) voltage to the motor, it can very easily hit a wall (drivetrain) or break the mechanism (elevator) if unprepared. Ensure that the ramp rate is set appropriately and adequate space is given (15m recommended for drivetrain) for the tests.

Ensure Adequate Space

- If the mechanism is continuous (swerve azimuth or a flywheel), then this is not an issue. However, mechanisms such as a drivetrain or elevator have a limited degree of movement. Ensure the configuration parameters match what is possible, and be prepared to disable the robot early.

Only Run Each Test Once

- Limitations of the SysId desktop utility prevent multiple of the same tests to be properly analyzed. Ensure each test is run exactly once.

Running Characterization

The quasistatic test will slowly ramp up voltage until the button has been released or a timeout has been hit. It is always safe to end the tests early, but at least ~3-5 seconds of data is necessary. Ensure ramp rate is configured such that this can be accomplished.

The dynamic test will immediately run the mechanism at the target voltage. This voltage may need to be adjusted if there is not sufficient room for the test.

With the routines configured and buttons set up, the characterization tests can be performed. To keep things simple and debuggable, perform tests in the following order.

1. Quasistatic forward
2. Quasistatic reverse
3. Dynamic forward
4. Dynamic reverse

Ensure each test is ran once, and only once. If a test is accidentally started multiple times, stop and restart the Signal Logger and try again.

Once you have a log with all the tests, you can use Tuner X to *extract the hoot log to WPILOG*. The exported WPILOG can then be *loaded into SysId* for analysis using the Talon FX Position, Velocity, and MotorVoltage signals.

17.4 Unit Testing

High-fidelity simulation with CTR Electronics devices can be used for *unit testing* robot applications.

When writing unit tests, the regular device APIs should be used to control devices and read status signals. Just like in simulation, the device `SimState` API can be used to update the simulated state of the device.

Additionally, users must ensure the robot is enabled prior to controlling actuators. This can be accomplished in WPILib by calling `DriverStationSim.setEnabled(true)` (Java, C++), followed by `DriverStation.notifyNewData()` to apply the change (Java, C++).

Important: There may be a short delay between enabling the robot and the simulated actuators being enabled. Unit tests should delay for ~100ms after constructing all devices and enabling the robot to account for this delay.

In unit tests, users should utilize the `StatusSignal.waitForUpdate()` and `BaseStatusSignal.waitForAll()` APIs to wait for fresh data after sending a control request or modifying the simulated device state.

Important: There may be a short delay between sending a control request and the simulated device applying the control. Unit tests should delay for ~20ms after sending a control request to account for this delay.

Below is an example unit test that verifies the robot is enabled and verifies that the device responds to a control request.

Java

```

public class TalonFXTest implements AutoCloseable {
    static final double DELTA = 1e-3; // acceptable deviation range

    TalonFX m_fx;
    TalonFXSimState m_fxSim;

    @Override
    public void close() {
        /* destroy our TalonFX object */
        m_fx.close();
    }

    @BeforeEach
    public void constructDevices() {
        assert HAL.initialize(500, 0);

        /* create the TalonFX */
        m_fx = new TalonFX(0);
        m_fxSim = m_fx.getSimState();

        /* enable the robot */
        DriverStationSim.setEnabled(true);
        DriverStationSim.notifyNewData();

        /* delay ~100ms so the devices can start up and enable */
        Timer.delay(0.100);
    }

    @AfterEach
    void shutdown() {
        close();
    }

    @Test
    public void robotIsEnabled() {
        /* verify that the robot is enabled */
        assertTrue(DriverStation.isEnabled());
    }

    @Test
    public void motorDrives() {
        /* set the voltage supplied by the battery */
        m_fxSim.setSupplyVoltage(RobotController.getBatteryVoltage());

        var dutyCycle = m_fx.getDutyCycle();

        /* wait for a fresh duty cycle signal */
        dutyCycle.waitForUpdate(0.100);
        /* verify that the motor output is zero */
        assertEquals(dutyCycle.getValue(), 0.0, DELTA);

        /* request 100% output */
        m_fx.setControl(new DutyCycleOut(1.0));
        /* wait for the control to apply */
        Timer.delay(0.020);
    }
}

```

(continues on next page)

(continued from previous page)

```

    /* wait for a new duty cycle signal */
    dutyCycle.waitForUpdate(0.100);
    /* verify that the motor output is 1.0 */
    assertEquals(dutyCycle.getValue(), 1.0, DELTA);
}
}

```

C++

```

class TalonFXTest : public testing::Test {
protected:
    /* create the TalonFX */
    hardware::TalonFX m_fx{0};
    sim::TalonFXSimState& m_fxSim{m_fx.GetSimState()};

    void SetUp() override
    {
        /* enable the robot */
        frc::sim::DriverStationSim::SetEnabled(true);
        frc::sim::DriverStationSim::NotifyNewData();

        /* delay ~100ms so the devices can start up and enable */
        std::this_thread::sleep_for(std::chrono::milliseconds{100});
    }
};

TEST_F(TalonFXTest, RobotIsEnabled)
{
    /* verify that the robot is enabled */
    EXPECT_TRUE(frc::DriverStation::IsEnabled());
}

TEST_F(TalonFXTest, MotorDrives)
{
    /* set the voltage supplied by the battery */
    m_fxSim.SetSupplyVoltage(frc::RobotController::GetBatteryVoltage());

    auto& dutyCycle = m_fx.GetDutyCycle();

    /* wait for a fresh duty cycle signal */
    dutyCycle.WaitForUpdate(100_ms);
    /* verify that the motor output is zero */
    EXPECT_DOUBLE_EQ(dutyCycle.GetValue(), 0.0);

    /* request 100% output */
    m_fx.SetControl(controls::DutyCycleOut{1.0});
    /* wait for the control to apply */
    std::this_thread::sleep_for(std::chrono::milliseconds{20});

    /* wait for a new duty cycle signal */
    dutyCycle.WaitForUpdate(100_ms);
    /* verify that the motor output is 1.0 */
    EXPECT_DOUBLE_EQ(dutyCycle.GetValue(), 1.0);
}

```

Comprehensive API usage examples and tutorials.

18.1 Open-Loop Quickstart

The below example showcases controlling a four-motor drivetrain.

18.1.1 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

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

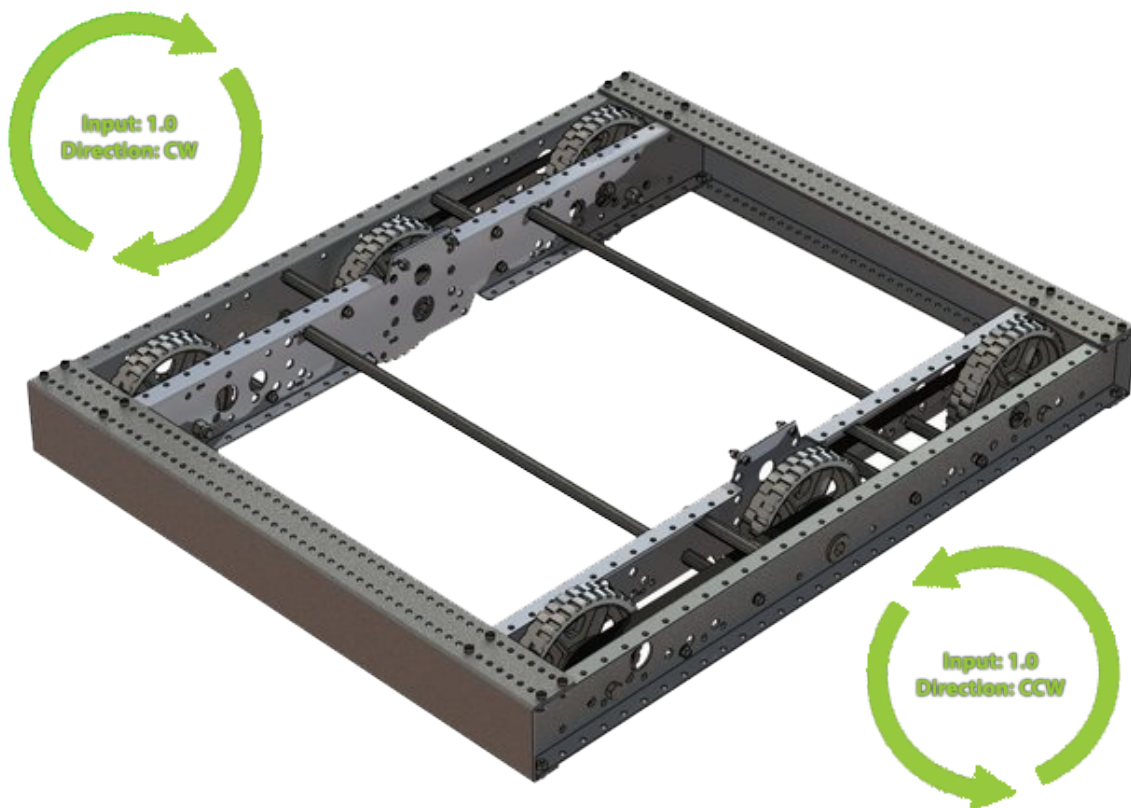
    private final TalonFX m_leftLeader = new TalonFX(0, kCANBus);
    private final TalonFX m_rightLeader = new TalonFX(1, kCANBus);
    private final TalonFX m_leftFollower = new TalonFX(2, kCANBus);
    private final TalonFX m_rightFollower = new TalonFX(3, kCANBus);
}
```

C++ (Header)

```
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};  
}
```

18.1.2 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

```

@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)

```

#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});
}

```

18.1.3 Full Example

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);
}

```

(continues on next page)

(continued from previous page)

```

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)

```

#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;

```

(continues on next page)

(continued from previous page)

```

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)

```

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};

```


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.

20.1 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

Warning: On ARM64 devices, Debian Bookworm is not supported at this time. Raspberry Pi users should select a “legacy” OS version that runs on Debian Bullseye.

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.

20.1.1 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.

20.1.2 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.

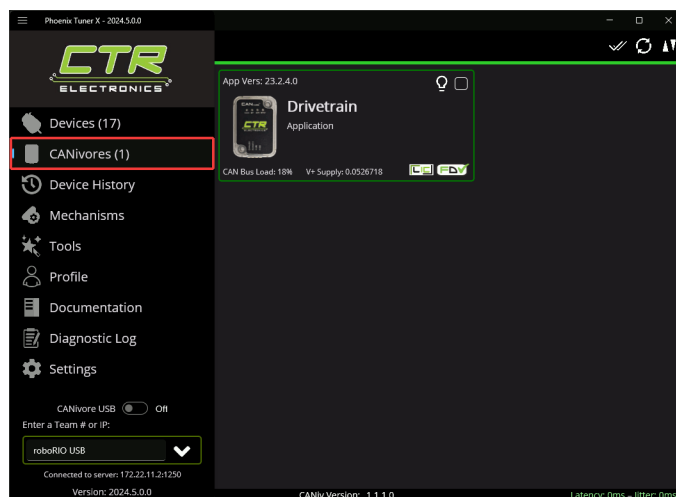
There are two options to work around this issue:

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.

20.2 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.



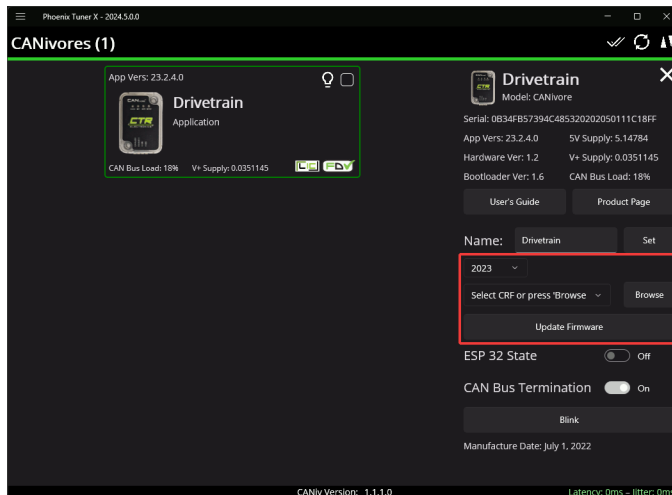
Note: The Phoenix Diagnostic Server must be running on the target system to use the CANivores page.

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.

20.3 Field Upgrading CANivores

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

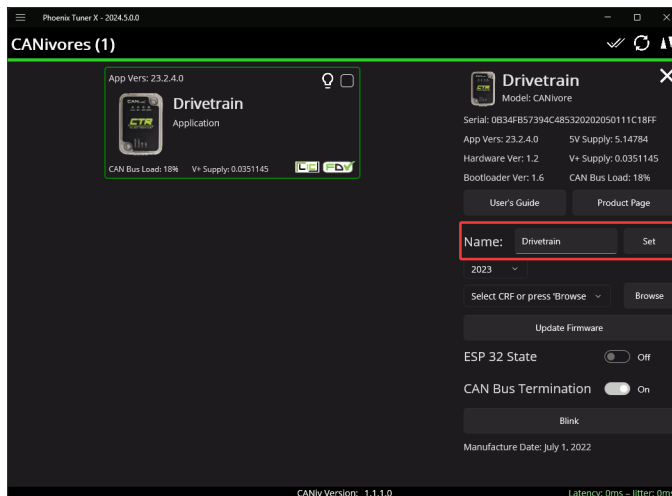
Click or tap on the listed CANivore card to open the device details page. The CANivore can then be field upgraded via the dropdown or by manually selected a file:



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*.

20.4 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"
```

Python

```
fx_default = hardware.TalonFX(0) # On roboRIO, this constructs a TalonFX on the RIO_
↳native CAN bus
fx_rio = hardware.TalonFX(1, "rio") # This also constructs a TalonFX on the RIO_
↳native CAN bus
fx_drivebase = hardware.TalonFX(0, "Drivebase") # This constructs a TalonFX on the_
↳CANivore bus named "Drivebase"
cc_elevator = hardware.CANcoder(0, "Elevator") # This constructs a CANcoder on the_
↳CANivore bus named "Elevator"
```

21.1 CANBus API

The CANBus API (Java, C++, Python) can be used to retrieve information about any given CAN bus, such as the bus utilization.

Java

```
// retrieve bus utilization for the CANivore named drivetrain
CANBusStatus canInfo = CANBus.getStatus("drivetrain");
float busUtil = canInfo.BusUtilization;

if (busUtil > 0.8) {
    System.out.println("CAN bus utilization is greater than 80%!");
}
```

C++

```
// retrieve bus utilization for the CANivore named drivetrain
CANBus::CANBusStatus canInfo = CANBus::GetStatus("drivetrain");
float busUtil = canInfo.BusUtilization;

if (busUtil > 0.8) {
    std::cout << "CAN bus utilization is greater than 80%!" << std::endl;
}
```


Python

```
# retrieve bus utilization for the CANivore named drivetrain
can_info = CANBus.get_status("drivetrain")
bus_util = can_info.bus_utilization

if bus_util > 0.8:
    print("CAN bus utilization is greater than 80%!")
```

21.2 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)

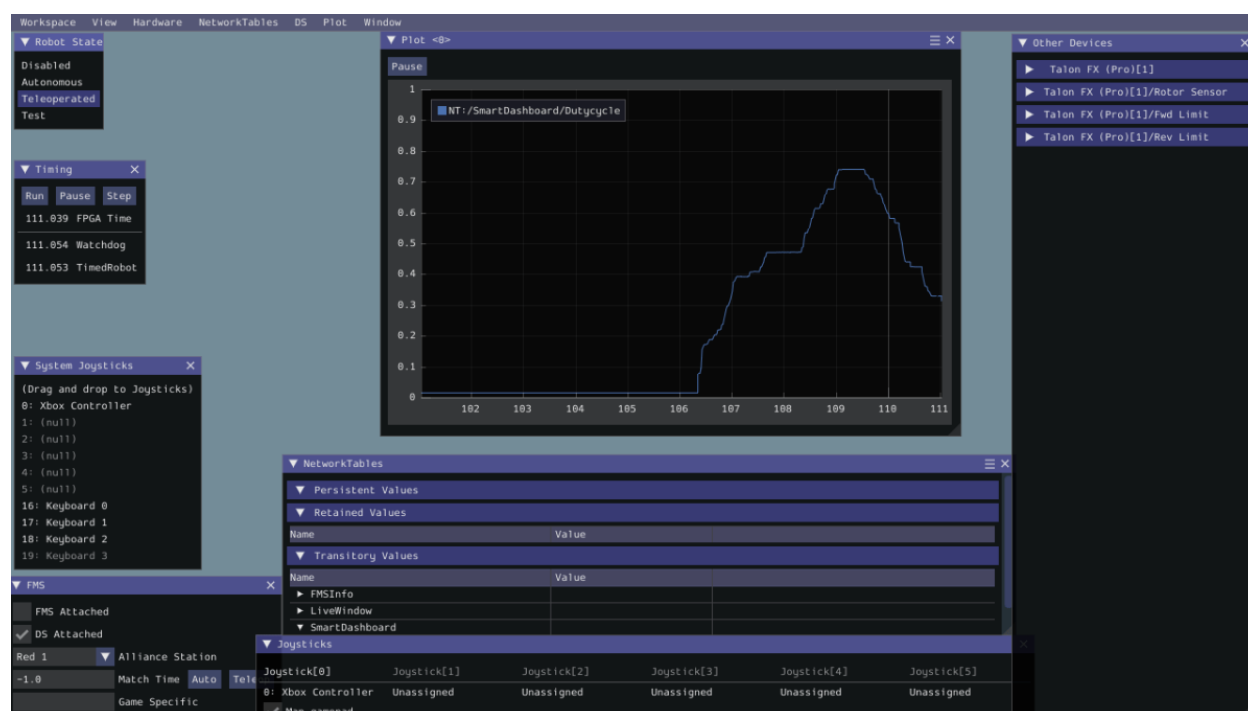
Message	Bring-up Status
CANbus Failed Bring-up	Found and connected to the CANivore, but could not configure the device or start the network
CANbus Successfully Started	Successfully configured the CANivore and started the network

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.

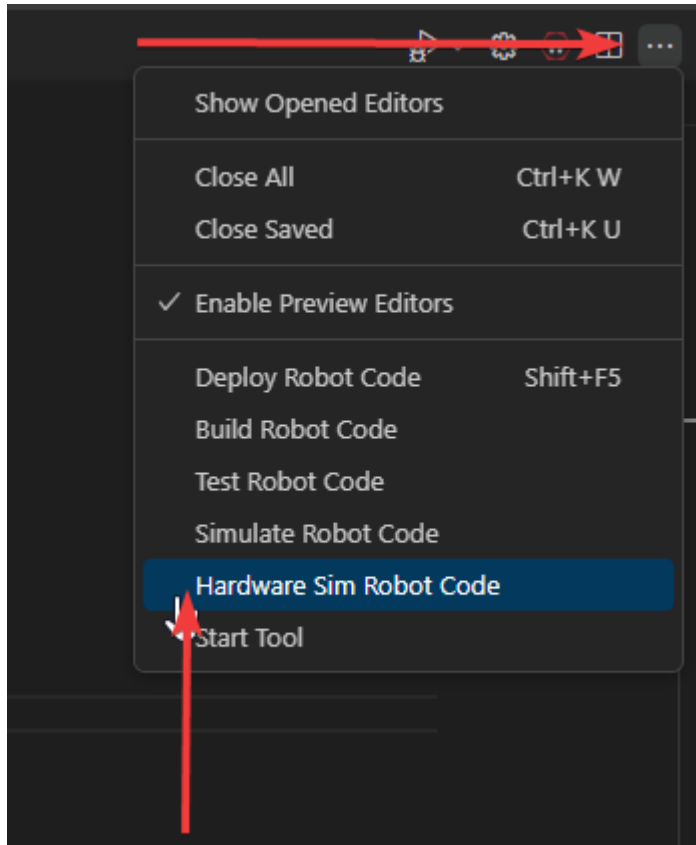
Java

```
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, 2B189E6333533853202034383803FF)  
[phoenix] CANbus Network Up: uno (WinUSB, 2B189E6333533853202034383803FF)  
[phoenix] Library initialization is complete.
```

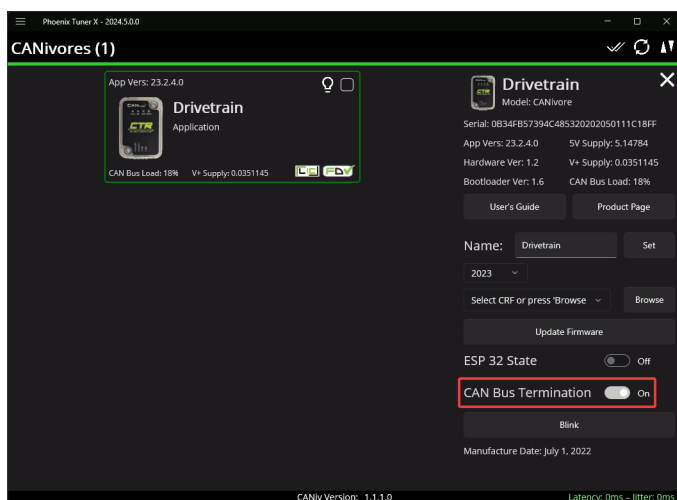
Advanced Configuration

The CANivore provides additional configuration options for advanced users.

23.1 CAN Bus Termination

The CANivore has a $120\ \Omega$ programmable resistor 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.

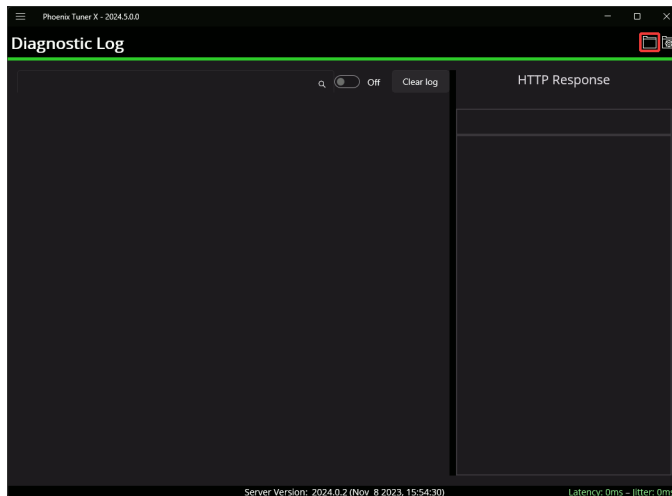


23.2 caniv - CANivore CLI

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:~#
```


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*
- *Factors that Impact Odometry*

24.1 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.

24.1.1 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.

24.1.2 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).

24.1.3 Solutions

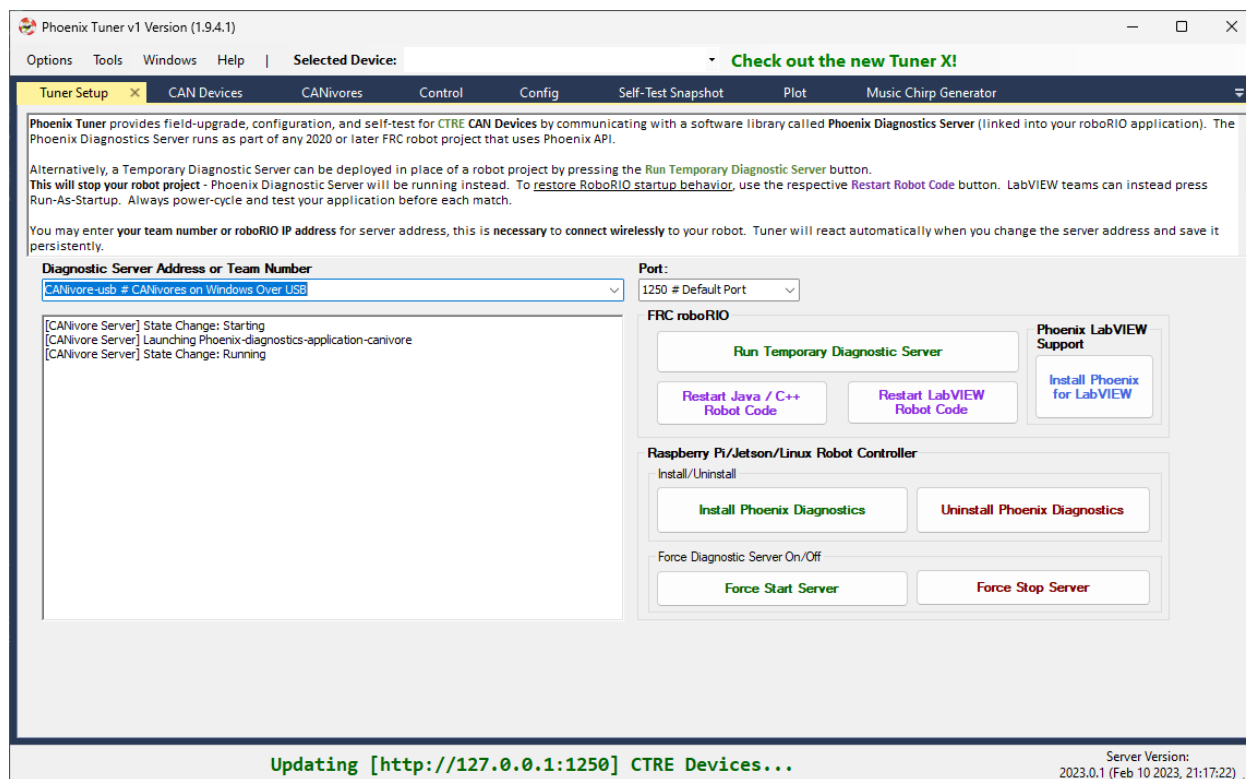
We (CTR Electronics) recognized these problems in general and wanted to solve them, which is why each of these problems are solved in the new Phoenix 6/Pro library and with CANivore.

- *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.

24.2 Tuner and an evolution in configuration

Authored by Dalton



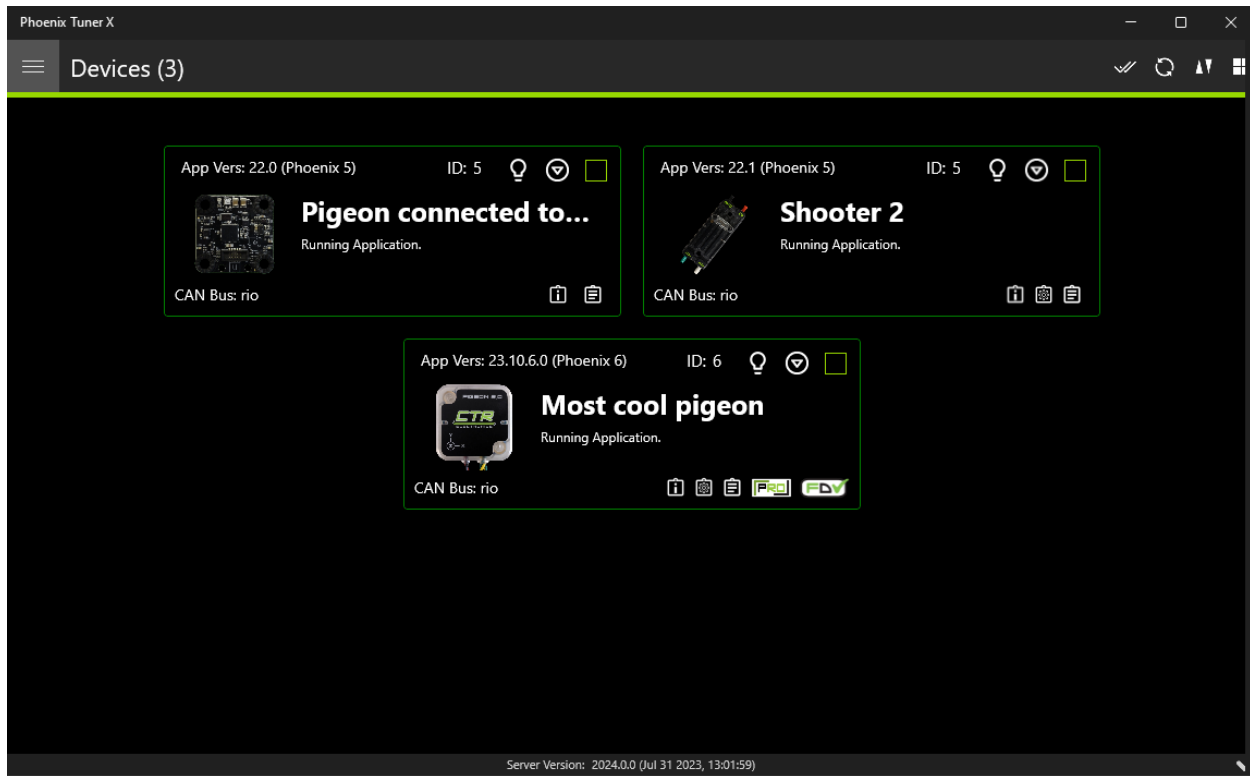
Since the introduction of the CTRE Toolsuite (pre-2018), we at CTR have strived to provide intuitive means of configuring and utilizing our products. In 2018, we launched Phoenix Tuner (now lovingly referred to as Tuner v1). Tuner v1 introduced features like: batch firmware upgrading like devices, diagnostic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

24.2.1 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](#).

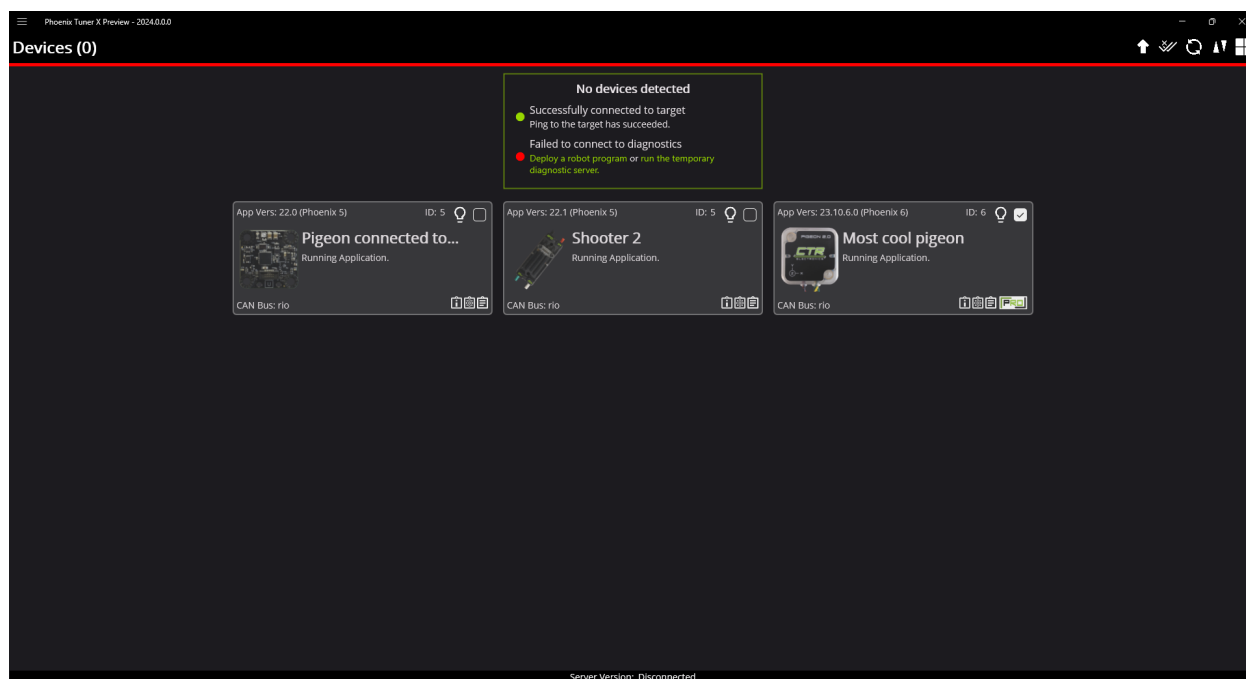
24.2.2 Introducing a new iteration of Tuner X

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.



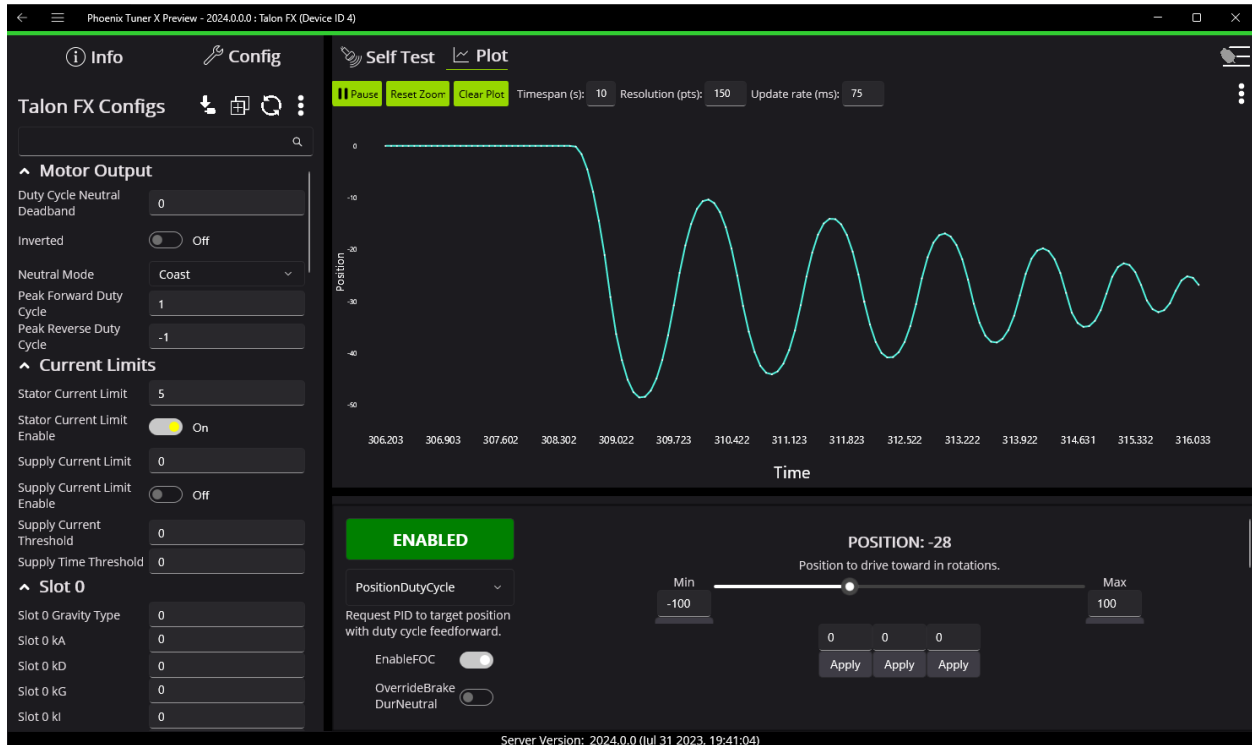
This 3 step check looks for the following:

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

24.2.3 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](#).

24.3 Factors that Impact Odometry

Authored by Cory

Often we've been asked what the impact higher frequencies, time synchronization, and synchronous API have on critical robot features, such as drivetrain odometry. This devblog will go into detail on the theoretical and practical impact they have.

Note: This doesn't cover **all** the factors that impact odometry, but it includes some of the major ones that contribute significantly to odometry error.

24.3.1 Update Frequencies

Update frequency has a direct impact on the accuracy of your localization through odometry, as it is an integration problem. The less frequent the odometry is called, the more time error can accrue before being updated to the present state of the robot. This can be seen graphically, the desmos session below shows the error in position, and how the error decreases as the update frequency increases.

<https://www.desmos.com/calculator/vdgebi9s4t>

Note: This desmos graph shows a Forward Euler discretization of a simple odometry case. The odometry solution provided by WPILib are discretized using the Pose Exponential, which is more accurate than Forward Euler.

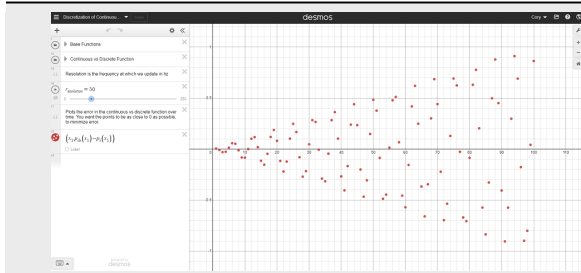


Fig. 1: Discretization error at 50 Hz

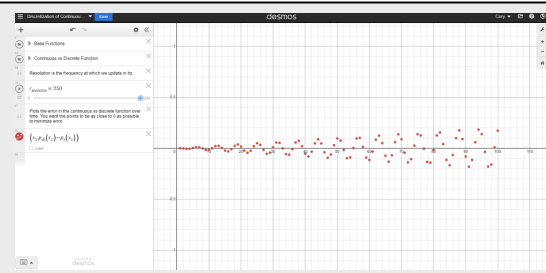


Fig. 2: Discretization error at 250 Hz

24.3.2 Synchronous API and Time Synchronization

Synchronous API and Time Synchronization will further improve the performance of odometry. The two do this by reducing the overall latency of the signals and reducing the random distribution of latency involved in each signal. Further explanation of this is available in the [Time Synchronization](#).

Latency reduces the accuracy of the data being used in the odometry, and with lower accuracy going into the odometry, the result will be less accurate as well (*garbage in, garbage out* concept).

We can add this into our desmos session, including the effect of latency and variable latency to our error calculation. <https://www.desmos.com/calculator/rytssjj158>

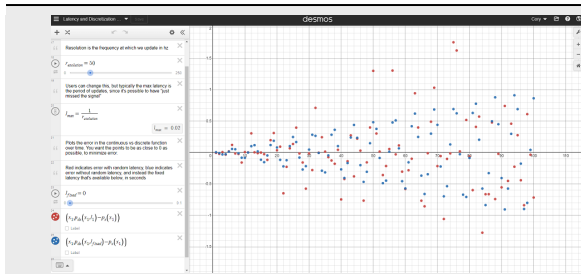


Fig. 3: Discretization error at 50 Hz with latency

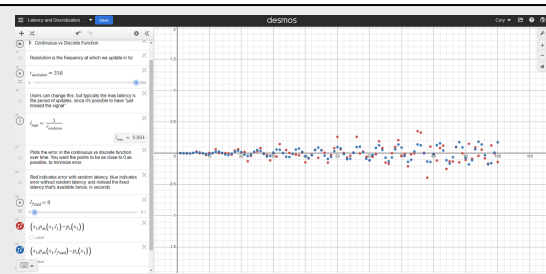


Fig. 4: Discretization error at 250 Hz with latency

24.3.3 Practical Results

With the theory out of the way, we can see what kind of impact this has on the odometry of a real robot performing real maneuvers.

We took a swerve drive robot and, using our Swerve API, drove it around the office. Attached on the robot is a limelight pointing straight up to our ceiling, which has April Tags at regular points. This allows the Limelight to know the absolute position of the robot throughout the motion.

The Limelight is configured for a high resolution capture to reduce the error of its pose estimation, at the cost of less frequent pose calculations. This is acceptable, because as the robot performs its maneuvers, it will come to a rest at key points, and when it's at rest, we can do our comparisons between the Limelight pose and the dead-reckoning from the odometry.

As we performed the maneuvers, we logged the pose of the robot as reported by the odometry and the Limelight for use in playback. The “real” robot pose is the odometry-driven pose, and the ghost is the Limelight reported pose. It can be assumed the limelight pose is the “true” pose while the robot is at rest.

Note: The Limelight pose measurements are not latency-compensated, so they will lag behind the odometry pose. The focus of this experiment is to see the difference in pose between the two methods while the robot is relatively still, so that this lag due to latency is not a factor.

The same maneuver was teleop-driven under the following circumstances, with the results below:

- CANivore CAN bus at 250 Hz (top left, measured at 45% CAN bus utilization)
- CANivore CAN bus at 50 Hz (top right, measured at 16% CAN bus utilization)
- RIO CAN bus at 250 Hz (bottom left, measured at 88% CAN bus utilization)
- RIO CAN bus at 50 Hz (bottom right, measured at 45% CAN bus utilization)



Final States

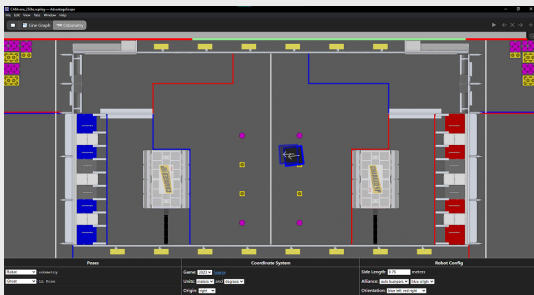


Fig. 5: CANivore 250 Hz end position

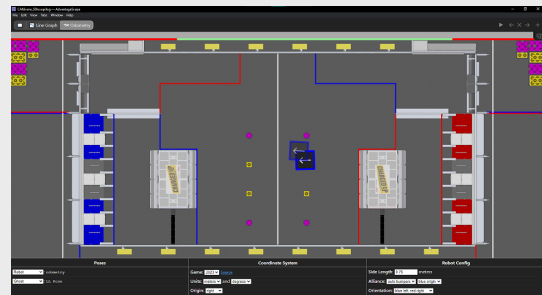


Fig. 6: CANivore 50 Hz end position

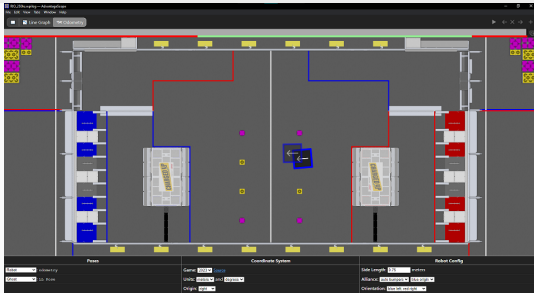


Fig. 7: RIO 250 Hz end position

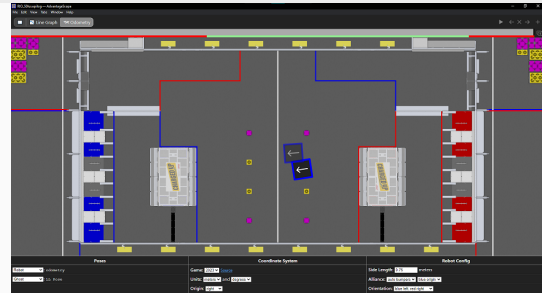


Fig. 8: RIO 50 Hz end position

As can be seen, going from the RIO bus to the CANivore bus, or from 50 Hz to 250 Hz improves the accuracy of the odometry, and by a noticeable amount. Based on this, utilizing faster update frequencies and time synchronization from the CANivore should result in more accurate odometry, even for the “short” movements as shown in the gif.

Data on the pose location is available for download: [OdometryData.xlsx](#).

24.3.4 After-Test Data

Roughly 2 weeks after this initial data was collected and the blog post written, we went back and re-verified the data for the CANivore 250 Hz and RIO 250 Hz cases to further test the impact of time synchronization. These tests were ran in autonomous a total of 20 times (10 for CANivore, 10 for RIO), measuring the error of the odometry against the Limelight data.

The results are below:

RIO	CANivore
0.52	0.14
0.33	0.28
0.47	0.56
0.23	0.22
0.51	0.32
0.23	0.18
0.22	0.30
0.59	0.26
0.15	0.33
0.11	0.25

This resulted in the following average and standard deviation of error:

	RIO	CANivore
Average	0.336	0.284
Standard Deviation	0.173	0.114

25.1 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).

25.1.1 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 *configure IDs* on the device(s).

25.1.2 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 $\sim 60\Omega$). If the measurement is 120Ω , then confirm both RIO and PDP are in circuit, and PDP jumper is in the correct location.

25.1.3 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.

25.2 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.

25.2.1 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
```

25.2.2 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.

CTR Electronics prides itself on excellent customer service. Our contact information can be found on our [website](#).