# Phoenix 6

## CTR Electronics

**Nov 02, 2026**

# Getting Started


**1 Requirements** **3**
- 1.1 Supported Devices . . . 3
- 1.2 CAN Bus Requirements . . . 3
- 1.3 System Requirements . . . 4

**2 Installing Phoenix 6** **5**
- 2.1 Installing Phoenix 6 (FRC) . . . 5
- 2.2 Installing Phoenix 6 (non-FRC) . . . 7

**3 New for 2026** **13**
- 3.1 API . . . 13
- 3.2 Talon FX® Improvements . . . 18
- 3.3 Phoenix Tuner X . . . 19

**4 Phoenix 6 Features** **23**
- 4.1 Phoenix 6 . . . 23
- 4.2 Phoenix Pro . . . 24
- 4.3 Feature Breakdown . . . 25

**5 API Migration** **29**
- 5.1 API Structure . . . 29
- 5.2 Configuration . . . 32
- 5.3 Status Signals . . . 35
- 5.4 Control Requests . . . 38
- 5.5 Closed-Loop Control . . . 43
- 5.6 Feature Replacements . . . 51

**6 CAN Bus Utilization** **55**

**7 Phoenix Tuner X** **57**
- 7.1 Connecting Tuner . . . 57
- 7.2 Device List . . . 60
- 7.3 Tuner History . . . 66
- 7.4 Device Details . . . 67
- 7.5 Tuner Configs . . . 71
- 7.6 Self Test Snapshot . . . 73

7.7 Controlling Devices . . . 74
7.8 Plotting . . . 76
7.9 Multi-device Plot & Control . . . 83
7.10 Profile Page . . . 85
7.11 Pigeon® 2.0 Mount Calibration . . . 86
7.12 Tools . . . 86
7.13 Swerve Project Generator . . . 91
7.14 Tuner Elevator Generator . . . 101

**8 What is Licensing** **111**
8.1 Types of Licenses . . . 111

**9 Device Licensing** **113**
9.1 Purchasing a License . . . 113
9.2 Activating a License . . . 113
9.3 Verifying Activation State . . . 116
9.4 Troubleshooting . . . 119

**10 Season Pass Licensing** **121**
10.1 When is a Season Pass Applicable? . . . 121
10.2 Attaching a Team Number to Season Pass . . . 121
10.3 How many devices can I activate? . . . 122

**11 CANcoder®** **123**
11.1 Status Light Reference . . . 123
11.2 Magnet Placement . . . 124
11.3 Verifying Sensor Direction . . . 124
11.4 Zeroing the CANcoder® . . . 124

**12 CANdi®** **127**
12.1 Status Light Reference . . . 127

**13 CANdle®** **129**
13.1 Status Light Reference . . . 129

**14 CANrange®** **131**
14.1 Tuning CANrange® . . . 131
14.2 Status Light Reference . . . 135

**15 Pigeon® 2.0** **137**
15.1 Pigeon® 2.0 Troubleshooting . . . 137
15.2 Status Light Reference . . . 139
15.3 Mount Calibration . . . 139

**16 Talon FX®** **141**
16.1 Utilizing your Talon FX® . . . 142
16.2 Motor Orientation . . . 148
16.3 Actuator Limits . . . 150
16.4 Status Light Reference . . . 151

**17 Talon FXS®** **153**
17.1 Supported Motors . . . 153
17.2 Utilizing your Talon FXS® . . . 153
17.3 Status Light Reference . . . 153
17.4 Troubleshooting Fault LED . . . 154

**18 General API Usage** **155**
18.1 API Overview . . . 156
18.2 Configuration . . . 157
18.3 Control Requests . . . 163
18.4 Status Signals . . . 166
18.5 Signal Logging . . . 176
18.6 Hoot Replay . . . 182
18.7 Device Faults . . . 193
18.8 Enabling Actuators . . . 195
18.9 Actuator Limits . . . 196
18.10 Orchestra . . . 199

**19 Device API** **203**
19.1 Talon FX® / Talon FXS® . . . 203

**20 Mechanisms** **249**
20.1 Swerve Overview . . . 249
20.2 Differential Overview . . . 294
20.3 Generating Mechanisms . . . 311

**21 Simulation** **313**
21.1 Introduction to Simulation . . . 313

**22 WPILib Integration** **319**
22.1 MotorController Integration . . . 319
22.2 Annotation Logging with Epilogue . . . 321
22.3 SysId Integration . . . 322
22.4 Unit Testing . . . 328

**23 Examples** **333**
23.1 Open-Loop Quickstart . . . 333

**24 CANivore® Intro** **339**
24.1 Status Light Reference . . . 339

**25 CANivore® Setup** **341**
25.1 Installing for roboRIO . . . 341
25.2 Installing for Linux (non-FRC) . . . 341
25.3 Viewing Attached CANivore® Devices . . . 341
25.4 Field Upgrading CANivore® . . . 342
25.5 Renaming CANivore® . . . 343

**26 CANivore® API** **345**
26.1 CANivore® Status Prints . . . 347

**27 Hardware-Attached Simulation** **349**

**28 Advanced Configuration** **353**
28.1 CAN Bus Termination . . . 353
28.2 caniv - CANivore® CLI . . . 354

**29 Development Blog** **355**
29.1 Latency and Frequency . . . 355
29.2 Tuner and an evolution in configuration . . . 356
29.3 Factors that Impact Odometry . . . 361

**30 Troubleshooting** **367**
30.1 CAN Bus Troubleshooting . . . . . 367
30.2 Running the Diagnostic Server . . . . . 369

**31 Support** **371**

**32 Code Generators** **373**

**33 Useful Categories** **375**

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

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

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

https://store.ctr-electronics.com/blogs/news New for 2026 What's new in Phoenix 6 for 2026.

*New for 2026* Migration Guide A Phoenix 5 migration guide is available here.

*API Migration* Changelog A full changelog with bug fixes is available here.

https://api.ctr-electronics.com/changelog Known Issues A list of current known issues is available here.

https://api.ctr-electronics.com/known_issues

# 1 Requirements

This document explains the requirements to use Phoenix 6.

## 1.1 Supported Devices

Phoenix 6 supports the following devices:

- CANcoder®
- CANdi®
- CANdle®
- CANrange®
- Pigeon® 2.0
- Talon FX® (Falcon 500, Kraken X60, Kraken X44)
- Talon FXS®

## 1.2 CAN Bus Requirements

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

### FRC

- roboRIO
- CANivore®

### non-FRC

On Linux systems, any SocketCAN capable adapter will work, but the CANivore® is highly recommended. CANivore® offers *additional functionality* over other SocketCAN adapters. On Windows systems, you will need a CANivore® to communicate with hardware.

## 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 Languages | Supports *CANivore®* | Supports *High-Fidelity Simulation* |
|---|---|---|---|
| NI roboRIO | Java, C++, Python | Yes | n/a |
| Windows 10/11 x86-64 | Java, C++, Python | Yes | Yes |
| Linux x86-64 (desktop)[1] and ARM64[2] | Java, C++, Python | Yes | Yes |
| macOS | Java, C++, Python | No | Yes |

### non-FRC

| Targets | Supported Languages | Supports *CANivore®* | Supports *High-Fidelity Simulation* |
|---|---|---|---|
| Windows 10/11 x86-64 | C#, Python | Yes | Yes |
| Linux x86-64 (desktop)[1] and ARM64[2] | C++, C#, Python | Yes | Yes (C# and Python only) |
| macOS (Simulation Only) | C#, Python | No | Yes |

[1] Supported Linux x86-64 (desktop) targets:

- Ubuntu 22.04 or newer
- Debian Bookworm or newer

[2] Supported Linux ARM64 targets:

- Raspberry Pi
- NVIDIA Jetson (Jetpack 6 or newer)
- Ubuntu 22.04 or newer
- Debian Bookworm or newer

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

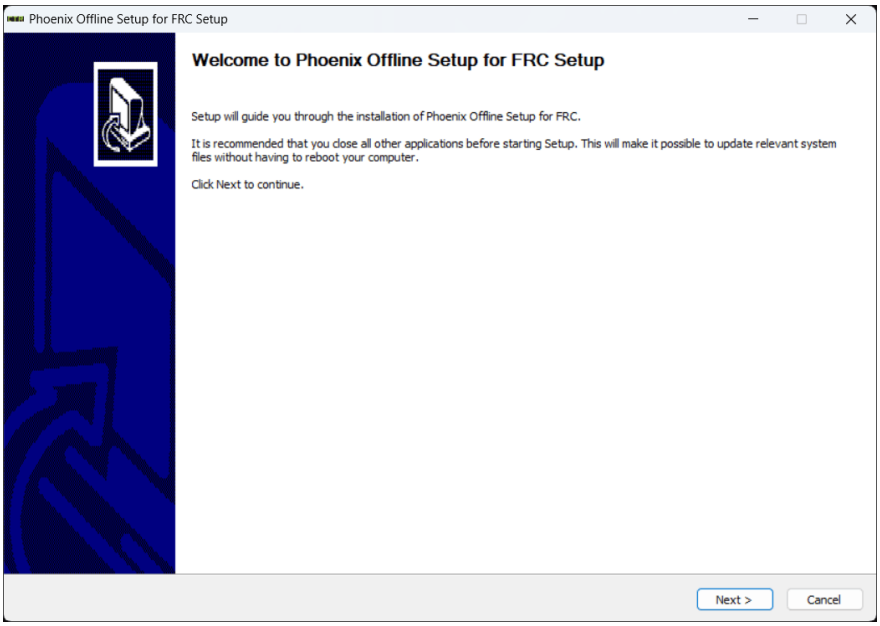
*Installing Phoenix 6 (non-FRC)*

## 2.1 Installing Phoenix 6 (FRC)

### Java/C++

### Offline

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

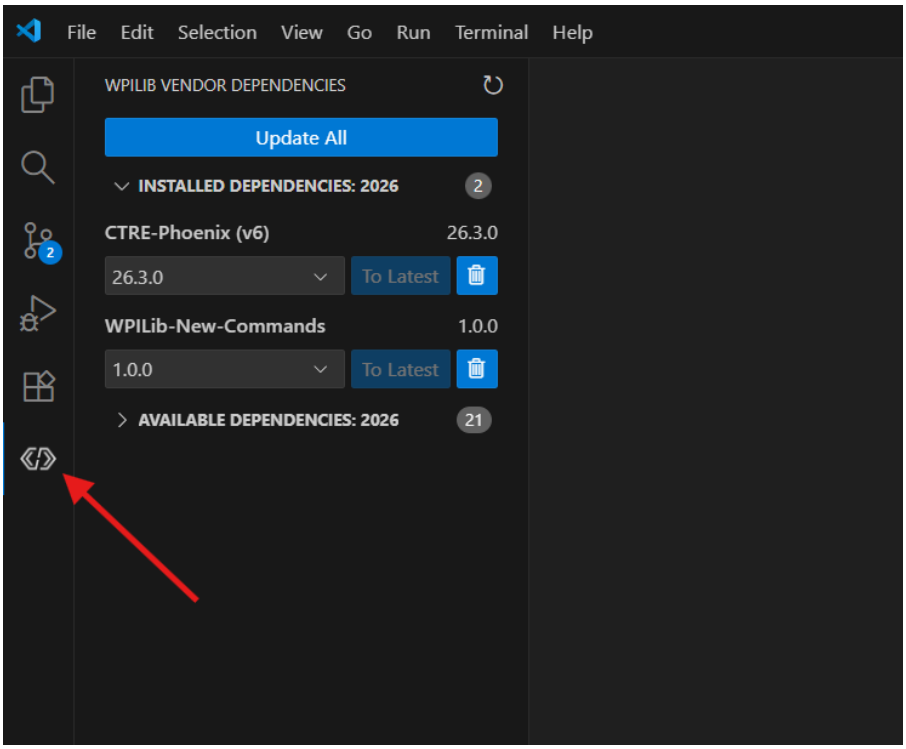


3. Apply the vendordep via WPILib VS Code Adding Offline Libraries

### Online

Users can install Phoenix without an installer using WPILib's Vendor Dependencies 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 left sidebar.




Then under the *Available Dependencies* dropdown, find *CTRE-Phoenix (v6)* and click *Install* to add the vendordep to the robot project. The vendordep can also be updated from this page by pressing the *To Latest* button.

Additionally, v5 can safely installed alongside it by installing the *CTRE-Phoenix (v5)* vendordep.

**Important**

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

Alternatively, the Hoot Replay version of the vendordep can be installed using *CTRE-Phoenix Replay (v6)*, as well as the v5 Replay-compatible vendordep using *CTRE-Phoenix (v5) Replay Compatibility*.

The vendordeps are also available at the following links:

- Phoenix 6: https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/Phoenix6-frc2026-latest.json
- Phoenix 5: https://maven.ctr-electronics.com/release/com/ctre/phoenix/Phoenix5-frc2026-latest.json

- Phoenix 6 Replay: https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/Phoenix6-replay-frc2026-latest.json
- Phoniex 5 Replay Compatibility: https://maven.ctr-electronics.com/release/com/ctre/phoenix/Phoenix5-replay-frc2026-latest.json

#### Python

First, make sure to install RobotPy. From there, installation of Phoenix 6 is available through PyPI.

```
python3 -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" pallette 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.

## 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).

Installation is composed of a few primary pieces. Linux users may skip CANivore® installation if they are using a generic USB-to-CAN adapter.

1. *APT Repository Installation* (Linux only)
2. (Optional) *CANivore® Installation* (Linux only)
3. *API Installation*

### 2.2.1 Linux Setup

Linux systems require some extra steps to install Phoenix 6 and CANivore®.

#### APT Repository Installation

The Phoenix 6 C++ API, CANivore® kernel module, and other utilities are distributed through our APT repository. The repository must be added 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.

**Warning**

Users who previously modified the `.list` distribution to use `raspberrypi` or `jetson` should switch to the `stable` distribution moving forward.

#### CANivore® Installation

On non-FRC Linux systems, the `canivore-usb` kernel module must be installed to add SocketCAN support for the CANivore®. The kernel module is distributed through our APT repository.

**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.

#### Raspberry Pi

1. Update APT and install the Raspberry Pi kernel headers.

```
sudo apt update
sudo apt install linux-headers-rpi-v8 # previously raspberrypi-kernel-headers
```

2. Install `canivore-usb`.

```
sudo apt install canivore-usb -y
```

#### Other Distributions

1. Update APT and install `canivore-usb`.

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

### 2.2.2 API Installation

#### Python

Installation is available through PyPI.

```
python3 -m pip install phoenix6
```

> **Tip**
>
> To get a robot application up and running quickly, check out our non-FRC Python example.

#### C#

Installation is available through NuGet.

An example on adding NuGet packages to a Visual Studio project is available in the Microsoft Quickstart. Alternatively, the package can be added using the dotnet CLI.

```
dotnet add package Phoenix6.Hardware
```

#### C++ (Linux)

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.

#### Simulation vs Hardware

#### Python

Starting in Phoenix 6 26.3.0, the default behavior is to run using hardware devices if WPILib/robotpy is not present and the host platform supports it (see *requirements* for a full list of supported platforms).

On platforms that support both simulation and hardware, the robot program can switch between communicating with physical devices and with simulated devices using the `CTR_TARGET` environment variable. Possible values of `CTR_TARGET` are:

- `Hardware` (see *Hardware-Attached Simulation*)

- `Simulation` (see *Introduction to Simulation*)
- `Replay` (see *Hoot Replay*)

Examples of setting `CTR_TARGET` are shown below.

#### Windows

```
$env:CTR_TARGET="Hardware" # Set the environment variable, which will persist␣
↪for the duration of this powershell instance.
```

#### Linux

```
export CTR_TARGET=Hardware # Export the environment variable so it's␣
↪persistent in the shell
```

Or

```
CTR_TARGET=Hardware python3 application.py # Set the environment variable␣
↪only for the python call
```

#### C#

There are several variations of the Phoenix 6 NuGet package to switch between real and simulated devices on supported platforms (see *requirements*). The available NuGet packages are:

- `Phoenix6.Hardware` (see *Hardware-Attached Simulation*)
- `Phoenix6.Simulation` (see *Introduction to Simulation*)
- `Phoenix6.Replay` (see *Hoot Replay*)

Note that only one of these packages may be active in the program at a time.

### 2.2.3 Firmware Installation

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

### 2.2.4 Troubleshooting

#### `Bad return status on module...`

This error can show up on Raspberry Pi or Jetson systems when using the incorrect tools distribution for APT. Refer to *APT Repository Installation* to correct your `.list` file.

#### Diagnostics is running but no CAN

**Symptom:** Tuner can ping the system and see that diagnostics is running but there are no devices.

**Possible Solutions:**

- Verify that the CAN Bus is valid. Motors should be blinking orange. Consult docs/hardware-reference/index:hardware reference for a full list of LED codes.

- Verify that the `canivore-usb` kernel module is correctly installed by running `modinfo canivore-usb`.
  - If `canivore-usb` shows installed with APT but does not appear with `modinfo`, then an update may have broken the module. Uninstall and reinstall `canivore-usb` with:

```
sudo apt remove canivore-usb canivore-usb-kernel
sudo apt install canivore-usb
```

### 2.2.5 Examples

The FRC Phoenix 6 Examples can be used as a reference outside of FRC. Most of the API is identical, aside from a few FRC-specific integrations and the WPILib framework used by the robot program.

#### Non-FRC Projects

We recommend that users use the standard language projects to use our library outside of FRC.

- A typical Python project will have a `robot.py` that utilizes the installed `phoenix6` library, as demonstrated by the Python example.
- A typical C# project is a Visual Studio or `dotnet` project that includes our libraries via NuGet.
- A typical C++ Linux project uses CMake, as demonstrated by the C++ Linux example.

> **Note**
>
> When utilizing actuators outside of FRC, the user must continuously feed an enable signal to the device. For more information, see *Enabling Actuators*.

### 2.2.6 Alternative Support

Some older devices such as the Talon SRX are not supported by Phoenix 6 and require usage of 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 **Phoenix 5** devices.

# 3 New for 2026

Engineering never stops, and neither do we. At CTR Electronics, we are constantly analyzing and reflecting on our software, documentation and hardware integration. This past year, we've made immense investments in our differential stack, data analysis with Tuner, data logging and replay pipeline and much more. We are proud to present our new for 2026 changelog!

Firmware for the 2026 release of Phoenix 6 can be found by selecting "2026" in the firmware selection menu.

The API vendordep for 2026 is available under `https://maven.ctr-electronics.com/release/com/ctre/phoenix6/latest/Phoenix6-frc2026-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 ecosystem. For a detailed list of changes and bug fixes, visit the API changelog.

## 3.1 API

### 3.1.1 Breaking Changes

- The `new <Device>(int id)` and `new <Device>(int id, String canbus)` constructors are now deprecated and will be removed in 2027. Use the `new <Device>(int id, CANBus canbus)` constructors instead. This change is intended to prepare users for 2027, where an explicit CAN bus declaration is necessary.
- The minimum supported C++ version is now C++ 20, and the minimum Linux requirement is Ubuntu 22.04 / Debian Bookworm. Linux ARM32 is no longer supported.
- C++: Improved robot project compilation times. This results in the following breaking changes: The `ctre/phoenix6/configs/Configs.hpp` header has been split into separate files. In generated swerve projects, `TunerConstants.h` must now explicitly include the motor controllers and encoder used.
- `DifferentialMechanism` and `SimpleDifferentialMechanism` have been reworked to more closely align with the swerve API. The behavior of the difference axis has also

been adjusted.

### 3.1.2 Signal Logger Improvements & Behavior Changes




Signal logger has had quite a few changes over the off-season. Signal Logger now has support for Protobuf and WPILib Structs, arbitrarily sized user signals, and annotation-based logging as an *Epilogue Backend*.

The behavior of the signal logger auto-start functionality has also been changed in response to user feedback to make the experience consistent between at-home testing and events. Signal Logging auto start is **only enabled** on the roboRIO 1 if a flash drive is present, and otherwise enabled by default on the roboRIO 2. Signal logging is started by any of the following (whichever occurs first):

- It has been at least 1 second since program startup (allowing for calls to `setPath`), and the robot is enabled.
- It has been at least 5 seconds since program startup (allowing for calls to `setPath`), and the Driver Station is connected to the robot.

Additionally, `OptimizeBusUtilization()` now defaults to setting optimized signals to 4 Hz instead of 0 Hz, preserving data for hoot logs. Users can still get the old behavior by explicitly passing in 0 Hz.

#### Miscellaneous Changes

- Logs from the same robot program instance / FRC match will now be grouped into a subdirectory named after the match and the date/time of the start of the match.
- The list of *Free Signals* (including custom user signals) is no longer limited to WPILOG export and can now be exported to MCAP.

### 3.1.3 Hoot Replay Improvements

Hoot replay has been enhanced to simplify use cases beyond status signal playback. A `HootAutoReplay` API has been added to easily register custom "input" signals for logging on hardware and playback in replay. This also enables easy replay of the robot FPGA timestamp and joystick information.

```
private final Camera camera = new Camera(...);
private PoseEstimate cameraPoseEst = new PoseEstimate();

private final HootAutoReplay autoReplay = new HootAutoReplay()
   .withTimestampReplay()
   .withJoystickReplay()
   .withStruct(
      "CameraPoseEst/pose", Pose2d.struct,
      /* getter lambda returns the value to log */
      () -> cameraPoseEst.pose,
      /* setter lambda applies the value from the log */
      val -> cameraPoseEst.pose = val.value
   )
   .withDouble(
      "CameraPoseEst/timestamp",
      () -> cameraPoseEst.timestamp,
      val -> cameraPoseEst.timestamp = val.value
   );

private void fetchInputs() {
   cameraPoseEst = camera.getPoseEstimate();
}

@Override
public void robotPeriodic() {
   if (!Utils.isReplay()) {
      fetchInputs();
   }
   autoReplay.update();
}
```

We highly recommend reading over the *improved hoot replay* documentation for understanding how to integrate hoot replay into your robot program.

Additionally, the Signal Logger is now supported in Hoot Replay with the following behavior:

- Signal Logger is always enabled during replay. As a result, `SignalLogger::Start()` and `SignalLogger::Stop()` are ignored.
- The replayed log will always be written to a `replay_<date>_<time>/` subfolder next to the original log.
- All custom signals written during replay will be automatically placed under `hoot_replay/` in the log.
- The replayed log also contains all status signals and custom signals from the original log, excluding those starting with `hoot_replay/`.

This can be useful to log the new outputs after making changes to program logic.

### 3.1.4 Investments in Differential

Our differential API has seen a variety of changes and feature additions that should enhance and expand the capabilities of teams building differential systems (such as Differential Wrists or Elevators). `SimpleDifferentialMechanism` and `DifferentialMechanism` have been reworked to more closely align with swerve. Both APIs now take a `DifferentialMotorConstants` object on construction, internally construct and configure the motor controllers (with initial configs objects), and provide useful APIs such as `getAveragePosition()` and `setPosition(avg, diff)`. Support has also been added for `MotionMagicExpo` and `MotionMagicVelocity` on the average axis.

Additionally, configs have been added to control the behavior of continuous wrapping and gear ratios on the difference axis. This also comes with a **breaking change** in behavior on the differential axis: the **full differential output** is now added/subtracted from each motor, and the difference axis uses **half the difference** in position/velocity between the two motors. This effectively means that PID gains do not change, but setpoints on the difference axis must be halved.

User documentation for the differential API can be found *here*. We highly recommend users read over the full changelog for a full list changes to the differential API.

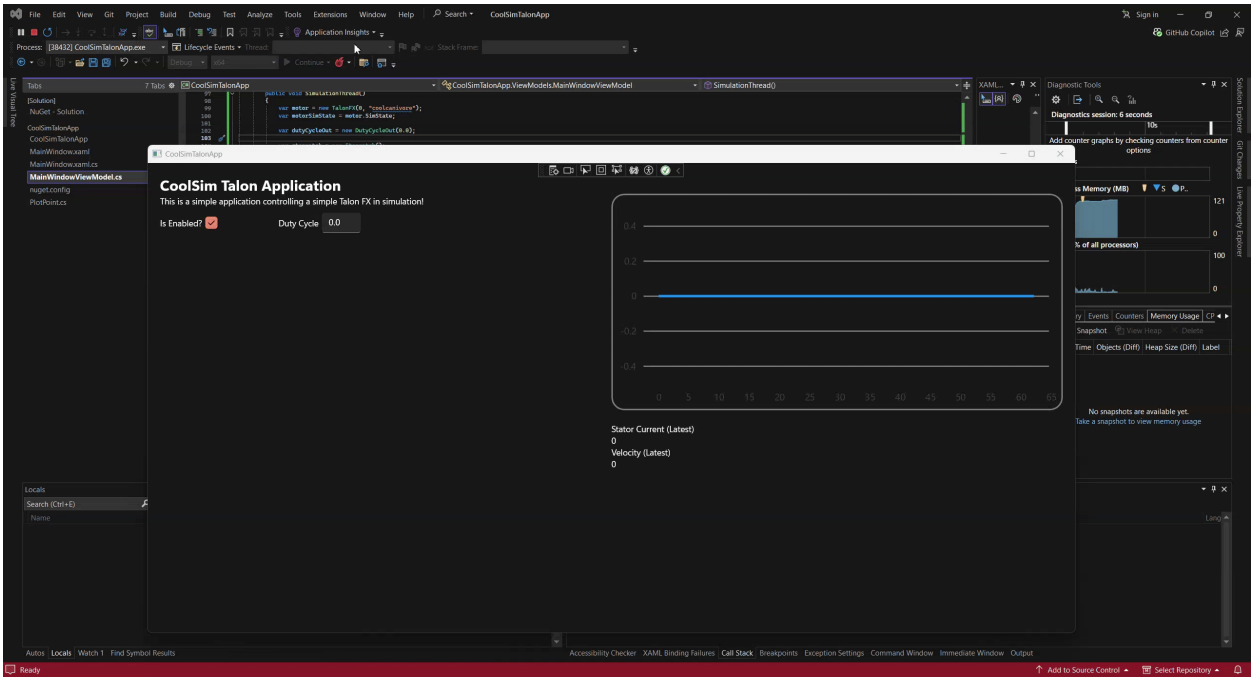
### 3.1.5 Improvements to Swerve

Swerve has seen a number of enhancements. We've added a `LinearPath` API that generates a linear path between two poses with constant velocity and acceleration limits. A `WheelForceCalculator` has also been added that calculates the wheel force feedforwards to apply for the given target robot accelerations or change in `ChassisSpeeds`, based on the robot mass and MOI.

The generated swerve project has been updated to include a default "drive straight" autonomous command that slowly drives forward for 5 seconds.

Additionally, the `SeedFieldCentric(Rotation2d)` overload has been added to reset the heading of the robot to the given operator-perspective heading. The generated swerve project has also been updated to include a simple drive-straight auton and log to `SignalLogger` using the new WPILib Struct and Struct array support.

### 3.1.6 Additional Language Support for C#




C# has received some much needed updates that bring it to feature parity with the other supported languages. C# fits the middle-ground for users looking for type safety without using an unmanaged language like C++.

Check out the *installation instructions* on installing the nuget.

### 3.1.7 Additional Utility Functions

We've added a couple new utility functions, specifically for interacting with status signals.

- Added `StatusSignal::IsNear(target, tolerance)` utility function that checks whether the signal is near a target value given tolerance.
- Added `StatusSignalCollection`, a lightweight `List<BaseStatusSignal>` wrapper that provides `waitForAll`/`refreshAll`/etc. This can be used to easily register status signals from multiple classes for a single `refreshAll` call.
- Java: Added `List<BaseStatusSignal>` overloads to `waitForAll`/`refreshAll`/etc.
- C++: Replaced the `std::vector` and `std::array` overloads with a `std::span` overload for APIs such as `WaitForAll`/`RefreshAll`/`OptimizeBusUtilization`/etc.

### 3.1.8 Enhancements to Error Reporting

Some changes have been made to the API to improve error reporting.

- StatusSignal `WaitForAll()` and `RefreshAll()` now report errors for all the status signals involved that errored.
- StatusSignal `WaitForAll()` and `RefreshAll()` now propagate an `InvalidNetwork` error to all the provided status signals.

## 3.2 Talon FX® Improvements

Improvements to follower have been made, specifically in regards to when the leader active request is Voltage-based.

- Follower and DifferentialFollower now follow `MotorVoltage` instead of `DutyCycle` when the leader is running a voltage control output type.
- Follower now follows the leader's coast/brake.

Additionally, *Dynamic Motion Magic® Expo* control requests have been added. Furthermore, the `GravityArmPositionOffset` config has also been added to offset the position used for arm kG calculations (within (-0.25, 0.25) rot).

#### Miscellaneous Changes

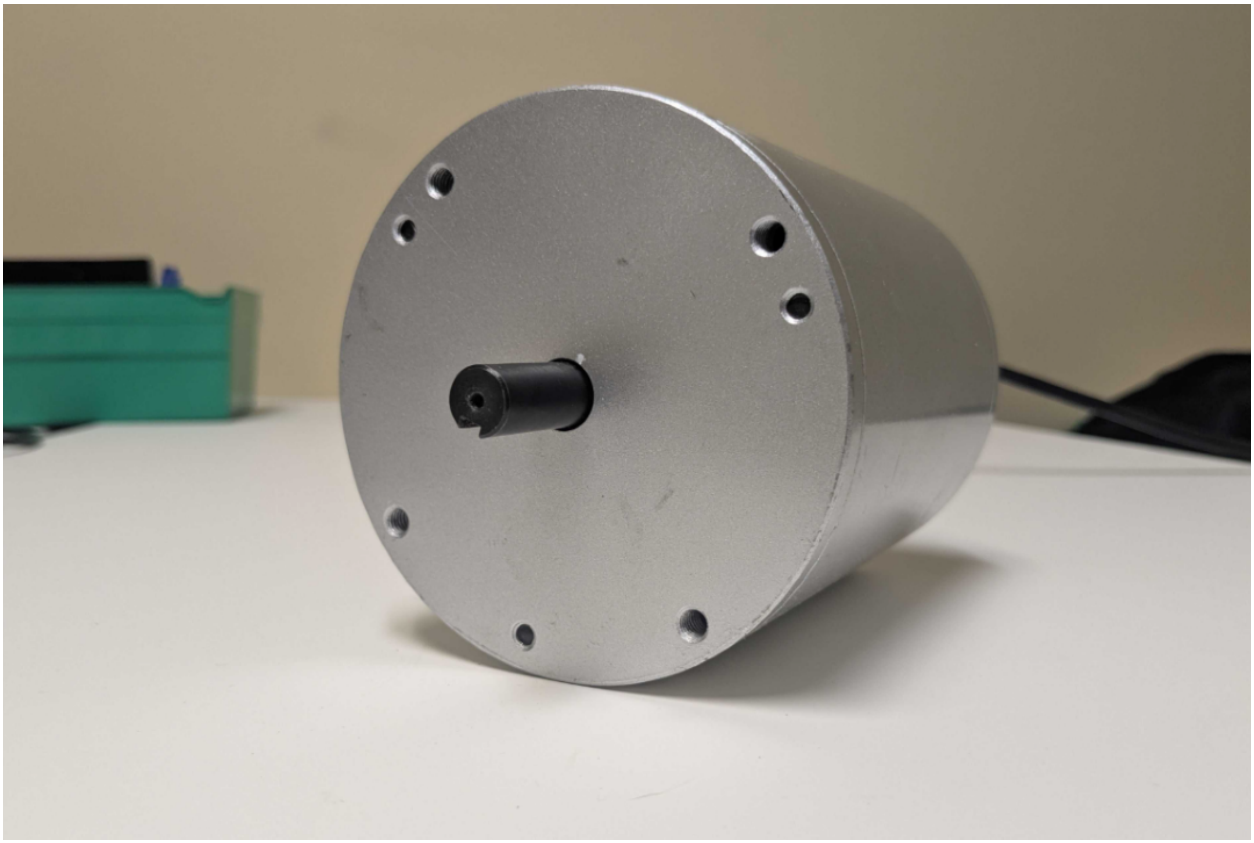
- Added support for simple gain scheduling in position PID closed-loop control based on closed-loop error. For more information, see the API documentation of `ClosedLoopGeneralConfigs::GainSchedErrorThreshold`, `ClosedLoopGeneralConfigs::GainSchedKpBehavior`, and `SlotConfigs::GainSchedBehavior`.
- Added `MotionMagicAtTarget` status signal, which returns whether the motion profile has completed (equivalent to checking that `MotionMagicIsRunning`, the `ClosedLoopReference` is the final target, and the `ClosedLoopReferenceSlope` is 0).

### 3.2.1 Kraken X44

Improvements to FOC have been made to improve peak performance.

Additionally, Kraken X44 simulation support has been added via `TalonFXSimState::SetMotorType`.

### 3.2.2 Talon FXS®

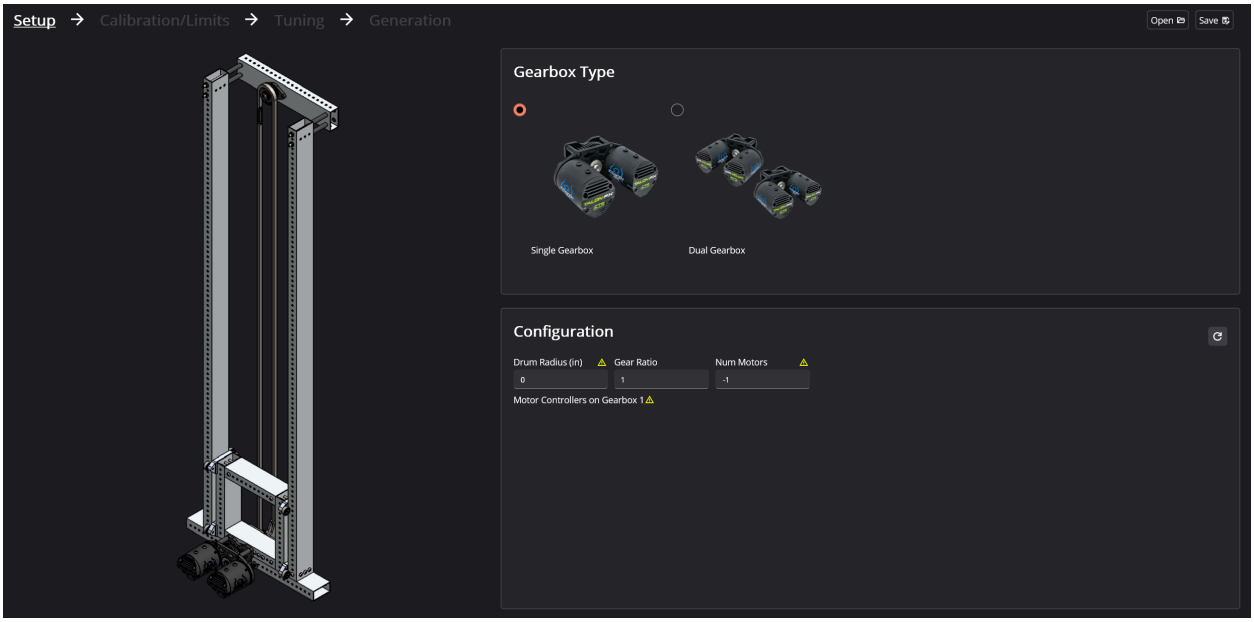


Our **non-FRC** customers can now utilize generic sensored (with hall effects) BLDC motors up to 24V on Talon FXS®. This enables use cases for industrial motors in automation. Take a look at the `Custom Brushless Motor` configs for more information on how to configure this.

## 3.3 Phoenix Tuner X

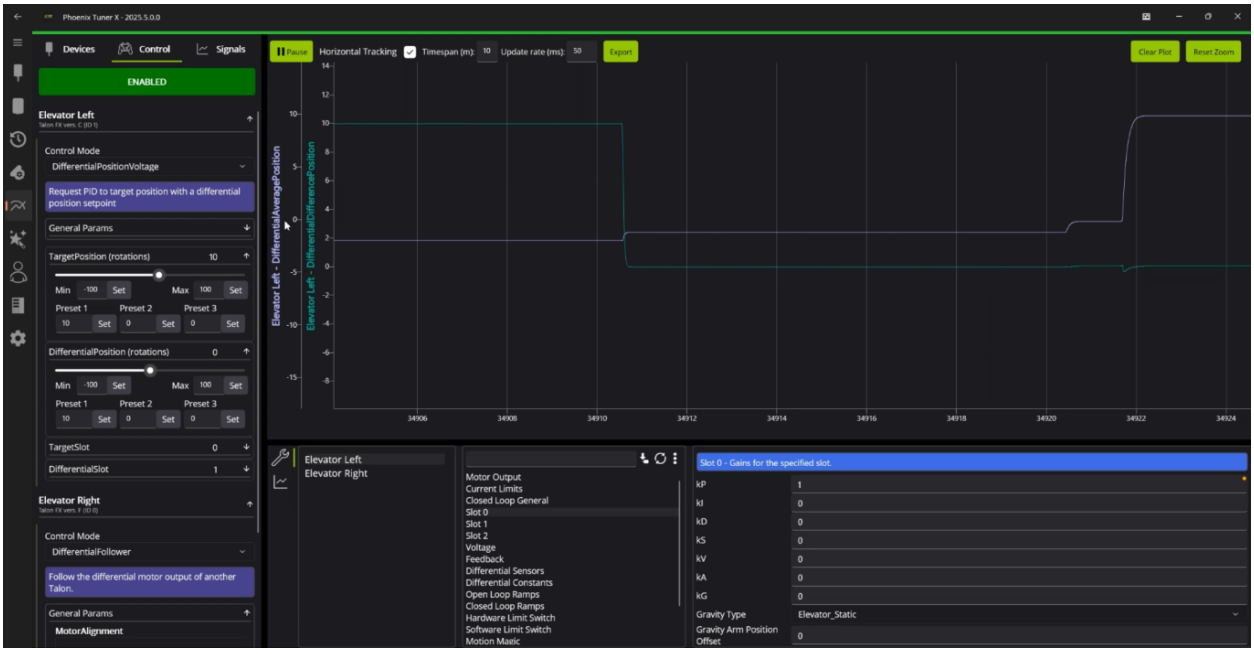
Tuner has seen a number of new feature additions, bug fixes, and general improvements.

### 3.3.1 Elevator Mechanism Support



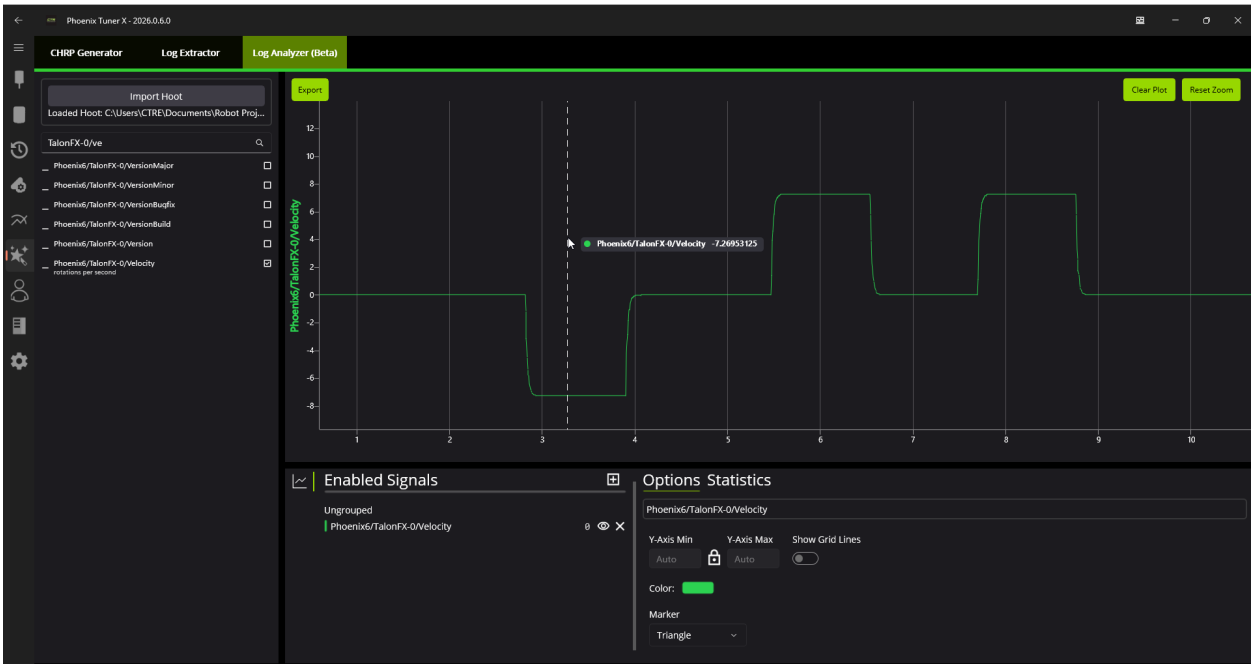

Under the Mechanisms page in Tuner X is the Elevator Generator. This utility guides the user through determining the necessary constants and configurations for a working Elevator subsystem. Take a read through the *relevant documentation pages* for information on how this works!

### 3.3.2 Multi-device Control & Plot




One of the most asked features of Phoenix Tuner has been the ability to control and plot multiple devices simultaneously. We are proud to announce that this feature will now be available for the 2026 season. This page can be found in the left-hand sidebar.

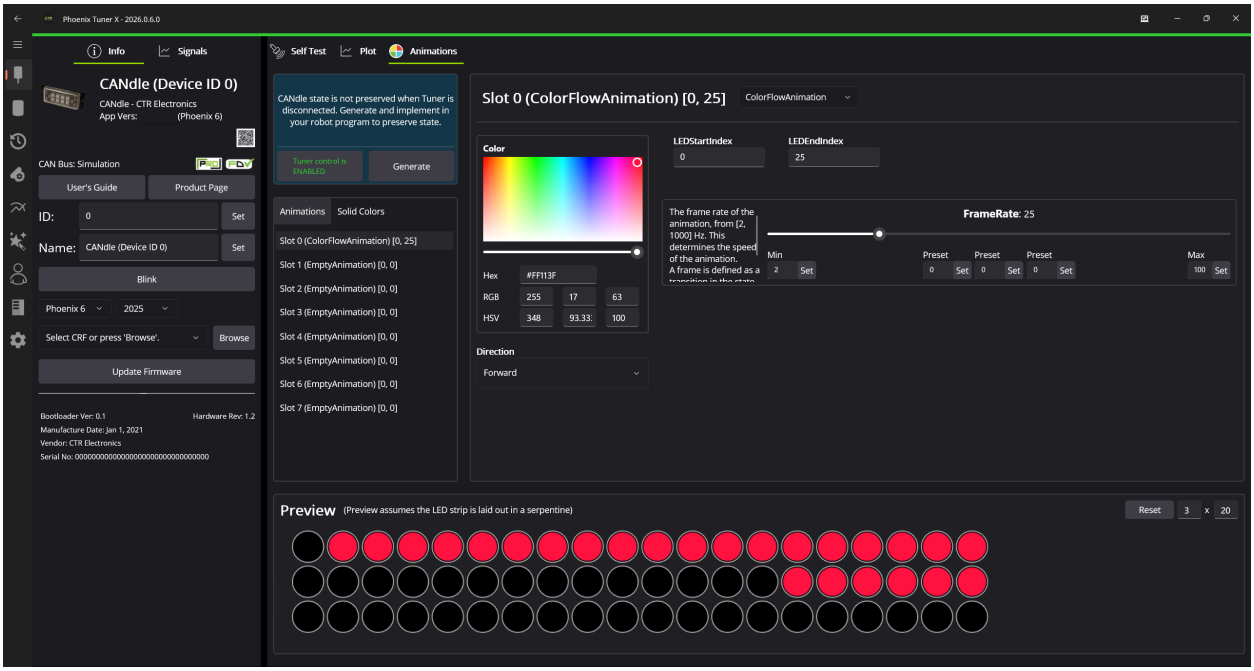
### 3.3.3 Integrated Log Analysis (Beta)




Hoot logs can now be directly plotted in Tuner. This feature is marked as beta as we continue to refine and improve the process. This page is available by going to *Tools* (magic wand in the sidebar) and then clicking on *Log Analyzer (Beta)*.

Feedback is welcome and can be submitted by sending us an email!

### 3.3.4 CANdle® Animations Preview




The traditional control interface has been replaced with a CANdle® animations tab, directly

in Tuner. This allows you to have a real-time preview of the state of your LED strip and also includes generation functionality. The built-in generator will take your current CANdle® animations and solid colors, and create a subsystem that you can implement in your robot program.

### 3.3.5 Miscellaneous Improvements

Beyond the big ticket feature adds is loads of smaller enhancements throughout the application.

- Reworked the flyout to be minimized by default and show a flat list of icons.
- Reworked configs to be a nested menu. This improves the performance and makes it more reusable across the application.
- Added support for extracting and deleting folders on the Hoot Extractor page.
- Added batch licensing to device history.

# 4 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 C++ units library and standardizes on SI units.
- Provides overloads using the Java units library.
- 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.
- Supported in Java, C++, and Python.
- Minimized GC impact in Java and Python using native C++ implementation.
- 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* and *Plotting*

- Detailed and resolute self tests to improve debugging.
- Plot signals at the configured signal update frequency.
- Combine multiple signal axes together and customize display of signal plots.

### 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 *Timesync®*

- Using Timesync®, signals from all devices on a CANivore® 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.
- Application of device control requests can be synchronized to signal publishing

### 4.2.3 FusedCANcoder

- *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 on a roboRIO 1 with a USB flash drive or a roboRIO 2.
- 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*.

### 4.2.5 *Replay Hoot Logs*

- Rerun your robot program in simulation using status signals and custom signals from a `hoot` log generated by the robot.
- No architecture changes necessary for automatic replay of device status signals.
- Robot automatically enables in the correct mode and runs through all maneuvers in the hoot log.
- Test code changes such as odometry improvements or failure condition detection and handling.
- Support for step timing and changing the speed of playback.

## 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 |

continues on next page

Table 1 – continued from previous page

| Feature | Phoenix 6 (rio) | Phoenix 6 + Pro (rio) | Phoenix 6 (CANivore®) | Phoenix 6 + Pro (CANivore®) |
|---|---|---|---|---|
| *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 Timestamps | x | x | x | x |
| *Limited Signal Logger Export* | x | x | x | x |
| *Explicit Control Requests* | x | x | x | x |
| *Motion Magic®* | 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 |
| Simple Differential Control | x | x | x | x |
| *Improved Self-Test Snapshot* | x | x | x | x |
| *Improved Tuner X Plotting* | x | x | x | x |
| CANivore® Timestamps | | | x | x |
| CAN FD | | | x | x |
| *Field Oriented Control (FOC)* | | x | | x |
| *FusedCANcoder + Talon FX®/Talon FXS®* | | x | | x |
| *SyncCANcoder + Talon FX®/Talon FXS®* | | x | | x |
| *Full Signal Logger Export* | | x | | x |
| *Hoot Log Replay* | | x | | x |
| *Timesync® Signal Publishing and Control* | | | | x |
| Device Timestamps | | | | x |

continues on next page

Table 1 – continued from previous page

| Feature | Phoenix 6 (rio) | Phoenix 6 + Pro (rio) | Phoenix 6 (CANivore®) | Phoenix 6 + Pro (CANivore®) |
|---|---|---|---|---|
| *Dynamic Motion Magic®* | | | | x |
| Full Differential Control | | | | x |
| *Swerve API* | + | ++ | ++ | +++ |

**Note**

+ The Swerve API is freely available; however, performance improves when used on a CANivore® bus and further improves when used with Pro devices. For more information, see *Factors that Impact Odometry*.

# 5 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 Talon FX®
- ***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*.

v5 **Java**

```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++**

```cpp
// 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);
```

continues on next page

Table 1 – continued from previous page

v6

**Java**

```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;

// All hardware classes already have WPILib integration
final TalonFX m_talonFX = new TalonFX(0);
final CANcoder m_cancoder = new CANcoder(0);

final TalonFXSimState m_talonFXSim = m_talonFX.getSimState();

final DutyCycleOut m_talonFXOut = new DutyCycleOut(0);

final TalonFXConfiguration m_talonFXConfig = new TalonFXConfiguration();
final CANcoderConfiguration m_cancoderConfig = new␣
↪CANcoderConfiguration();

InvertedValue m_talonFXInverted =␣
↪InvertedValue.CounterClockwise_Positive;

m_talonFX.setControl(m_talonFXOut);
```

**C++**

```cpp
// Phoenix 6 is in the ctre/phoenix6 headers
#include "ctre/phoenix6/CANcoder.hpp"
#include "ctre/phoenix6/TalonFX.hpp"

// Phoenix 6 uses the ctre::phoenix6 namespace
using namespace ctre::phoenix6;

// now types are organized cleanly by namespace

// All hardware classes already have WPILib integration
hardware::TalonFX m_talonFX{0};
hardware::CANcoder m_cancoder{0};

sim::TalonFXSimState& m_talonFXSim{m_talonFX.GetSimState()};

controls::DutyCycleOut m_talonFXOut{0};

configs::TalonFXConfiguration m_talonFXConfig{};
configs::CANcoderConfiguration m_cancoderConfig{};

signals::InvertedValue␣
↪m_talonFXInverted{signals::InvertedValue::CounterClockwise_Positive};

m_talonFX.SetControl(m_talonFXOut);
```



continues on next page

Table 1 – continued from previous page

## 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

v5

**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);
```

continues on next page

Table 2 – continued from previous page

v6

**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

v5

**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();
```

continues on next page

Table 3 – continued from previous page

v6

**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

v5

**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);
```

continues on next page

Table 4 – continued from previous page

v6

**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

v5

**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
```

continues on next page

Table 5 – continued from previous page

v6

**Java**

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

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

// retrieve position value that we just refreshed
// units are rotations, uses the units library
var rotorPos = rotorPosSignal.getValue();
// the units library can be bypassed using getValueAsDouble()
double rotorPosRotations = rotorPosSignal.getValueAsDouble();

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

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

**C++**

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

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

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

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

// synchronously wait 20 ms for new data
rotorPosSignal.WaitForUpdate(20_ms);
```

### Changing Update Frequency (Status Frame Period)

v5

**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);
```

v6

**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 `setInverted`) | `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® 2.0 Signals

> **Note**
>
> Many Pigeon® 2.0 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

v5

**Java**

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

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

**C++**

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

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

v6

**Java**

```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));
// the control request `with` methods also accept unit types
m_motor.setControl(m_request.withOutput(Volts.of(12.0)));
```

**C++**

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

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

#### Follower Motors

v5

**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
```

continues on next page

Table 8 – continued from previous page

v6

**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(),␣
↪MotorAlignmentValue.Aligned));
// set m_strictFollower to strict-follow m_leader
// strict followers ignore the leader's invert and use their own
m_strictFollower.setControl(new StrictFollower(m_leader.getDeviceID()));

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

**C++**

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

// robot init, set m_follower to follow m_leader
// m_follower should NOT oppose leader
m_follower.SetControl(controls::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(controls::StrictFollower{m_leader.GetDeviceID()});

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

### Changing Update Frequency (Control Frame Period)

v5

**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);
```

v6

**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 10: Open-loop Control Requests

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

Table 11: 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.

> **Note**
>
> This calculator assumes the `RotorToSensorRatio` and `SensorToMechanismRatio` configs are both set to 1. If this is not the case in your robot program, **divide** the resulting values by both ratios.

### 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.

> **Note**
>
> This calculator assumes the `RotorToSensorRatio` and `SensorToMechanismRatio` configs are both set to 1. If this is not the case in your robot program, **multiply** the resulting gains by both ratios.

#### 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.

> **Note**
>
> This calculator assumes the `RotorToSensorRatio` and `SensorToMechanismRatio` configs are both set to 1. If this is not the case in your robot program, **multiply** the resulting gains by both ratios.

#### 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.

> **Note**
>
> This calculator assumes the `RotorToSensorRatio` and `SensorToMechanismRatio` configs are both set to 1. If this is not the case in your robot program, **multiply** the resulting gains by both ratios.

#### 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.

**Note**

This calculator assumes the `RotorToSensorRatio` and `SensorToMechanismRatio` configs are both set to 1. If this is not the case in your robot program, **multiply** the resulting gains by both ratios.

### 5.5.3 Using Closed-Loop Control

v5 **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);
```

continues on next page

Table 12 – continued from previous page

v6

**Java**

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

// robot init, set slot 0 gains
var configs = new TalonFXConfiguration();
configs.Slot0.kV = 0.12;
configs.Slot0.kP = 0.11;
configs.Slot0.kI = 0.48;
configs.Slot0.kD = 0.01;
m_talonFX.getConfigurator().apply(configs, 0.050);

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

**C++**

```
// class member variable
controls::VelocityVoltage m_velocity{0_tps};

// robot init, set slot 0 gains
configs::TalonFXConfiguration configs{};
configs.Slot0.kV = 0.12;
configs.Slot0.kP = 0.11;
configs.Slot0.kI = 0.48;
configs.Slot0.kD = 0.01;
m_talonFX.GetConfigurator().Apply(configs, 50_ms);

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

### Motion Magic®

continues on next page

Table 13 – continued from previous page

v5

**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);
```



continues on next page

Table 13 – continued from previous page

v6

**Note**

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

**Java**

```
// class member variable
final MotionMagicVoltage m_motmag = new MotionMagicVoltage(0);

// robot init
var talonFXConfigs = new TalonFXConfiguration();

// set slot 0 gains
var slot0Configs = talonFXConfigs.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
// PID runs on position
slot0Configs.kP = 4.8;
slot0Configs.kI = 0;
slot0Configs.kD = 0.1;

// set Motion Magic settings
var motionMagicConfigs = talonFXConfigs.MotionMagicConfigs;
motionMagicConfigs.MotionMagicCruiseVelocity = 80; // 80 rps cruise␣
↪velocity
motionMagicConfigs.MotionMagicAcceleration = 160; // 160 rps/s␣
↪acceleration (0.5 seconds)
motionMagicConfigs.MotionMagicJerk = 1600; // 1600 rps/s^2 jerk (0.1␣
↪seconds)

m_talonFX.getConfigurator().apply(talonFXConfigs, 0.050);

// periodic, run Motion Magic with slot 0 configs,
// target position of 200 rotations
m_motmag.Slot = 0;
m_motor.setControl(m_motmag.withPosition(200));
```

**C++**

```
// class member variable
controls::MotionMagicVoltage m_motmag{0_tr};

// robot init
configs::TalonFXConfiguration talonFXConfigs{};

// set slot 0 gains
auto& slot0Configs = talonFXConfigs.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
// PID runs on position
slot0Configs.kP = 4.8;
slot0Configs.kI = 0;
slot0Configs.kD = 0.1;

// set Motion Magic settings
auto& motionMagicConfigs = talonFXConfigs.MotionMagicConfigs;
```

Table 13 – continued from previous page

### 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 configs = new TalonFXConfiguration();
configs.Slot0.kS = 0.24; // add 0.24 V to overcome friction
configs.Slot0.kV = 0.12; // apply 12 V for a target velocity of 100 rps
configs.Slot0.kP = 4.8;
configs.Slot0.kI = 0;
configs.Slot0.kD = 0.1;
m_talonFX.getConfigurator().apply(configs, 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::TalonFXConfiguration configs{};
configs.Slot0.kS = 0.24; // add 0.24 V to overcome friction
configs.Slot0.kV = 0.12; // apply 12 V for a target velocity of 100 rps
configs.Slot0.kP = 4.8;
configs.Slot0.kI = 0;
```

(continues on next page)

(continued from previous page)

```
configs.Slot0.kD = 0.1;
m_talonFX.GetConfigurator().Apply(configs, 50_ms);

// periodic, update the profile setpoint for 20 ms loop time
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®.
- When using an external sensor connected to a Talon FXS®, the sensor phase can be changed using a config (Java, C++).

### 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 Talon FX® 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.

On the Talon FX®, users can optionally configure the `VelocityFilterTimeConstant` config (Java, C++) to apply stronger filtering to the velocity signal. However, note that stronger filtering may negatively impact velocity PID control.

### 5.6.8 Allowable Closed-Loop Error

To check if a motor controller is within a given range of the target setpoint, call `isNear` (Java, C++) on the `Position` or `ClosedLoopError` status signal object.

Allowable closed-loop error, which zeroed motor output when close to the setpoint, has been replaced by the more flexible `GainSchedErrorThreshold` and `GainSchedKpBehavior` configs (Java, C++) and the per-slot `GainSchedBehavior` config (Java, C++). Users can set the `GainSchedBehavior` to `ZeroOutput` to get similar behavior to allowable closed-loop error, improved to allow kG and custom FeedForward output while within the threshold.

### 5.6.9 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.

Users can optionally use the new `GainSched*` configs mentioned above to mimic the behavior of Integral Zone by setting up a secondary PID slot with a stronger kI gain and setting the `GainSchedBehavior` of the primary PID slot to `UseSlot*`.

### 5.6.10 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.

### 5.6.11 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.

# 6 CAN Bus Utilization

CTR Electronics goes through great efforts to make our products efficient on CAN bus bandwidth. This article highlights the average default 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)[1] |
|---|---|---|---|---|
| CANcoder® | 1.8% | 1.7% | 0.9% | 0.9% |
| CANdi® | N/A | 2.5% | N/A | 1.1% |
| CANdle®[2] | 1.5% | 0.4% | 0.7% | 0.2% |
| CANrange® | N/A | 2.0% | N/A | 1.0% |
| Pigeon® 2.0 | 5.5% | 3.1% | 2.5% | 1.3% |
| Talon FX® | 4.7% | 4.1% | 2.0% | 1.8% |
| Talon FXS® | N/A | 4.2% | N/A | 1.8% |

[1] Phoenix 6 devices on CAN FD also increase the default update frequency of many status signals to 100 Hz.
[2] CANdle® device utilization is without LED usage. Animating or setting LEDs will increase bus utilization.

# 7 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, the Google Play Store, and the Apple App Store. On Windows machines, Phoenix Tuner X can also be installed using the Phoenix Offline Installer.

Phoenix Tuner X can also be installed on Windows using winget:

```
winget install "Phoenix Tuner X"
```

Phoenix Tuner X supports Windows 11 (21H2+), Android (9.0+), macOS on Apple Silicon (15.2+), and iOS (18.2+).

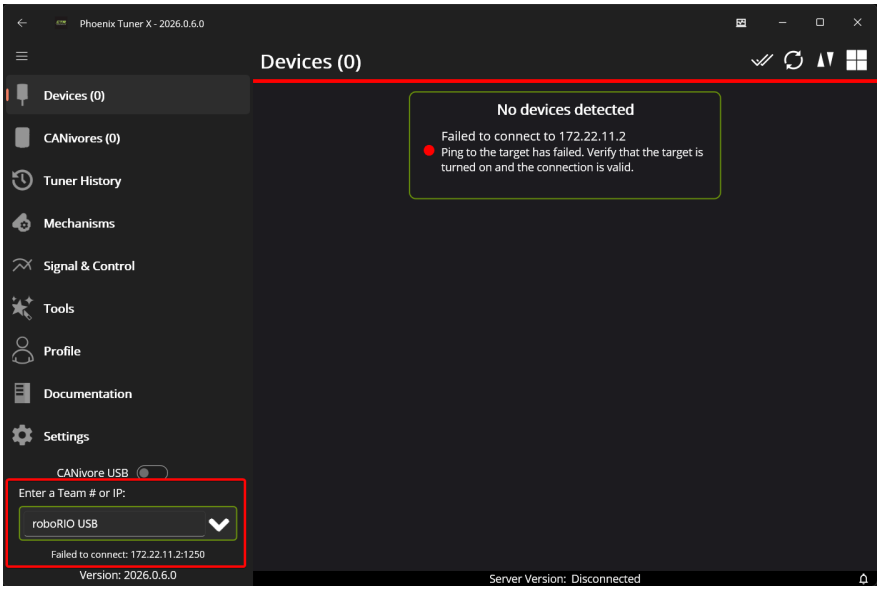
> **Tip**
>
> Many UI elements contain hover tooltips. That means the user can hover over them with their mouse for a text explanation of what they do.

## 7.1 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.

### 7.1.1 Connecting to the Server

A dropdown/textbox is available in the upper-left flyout menu.



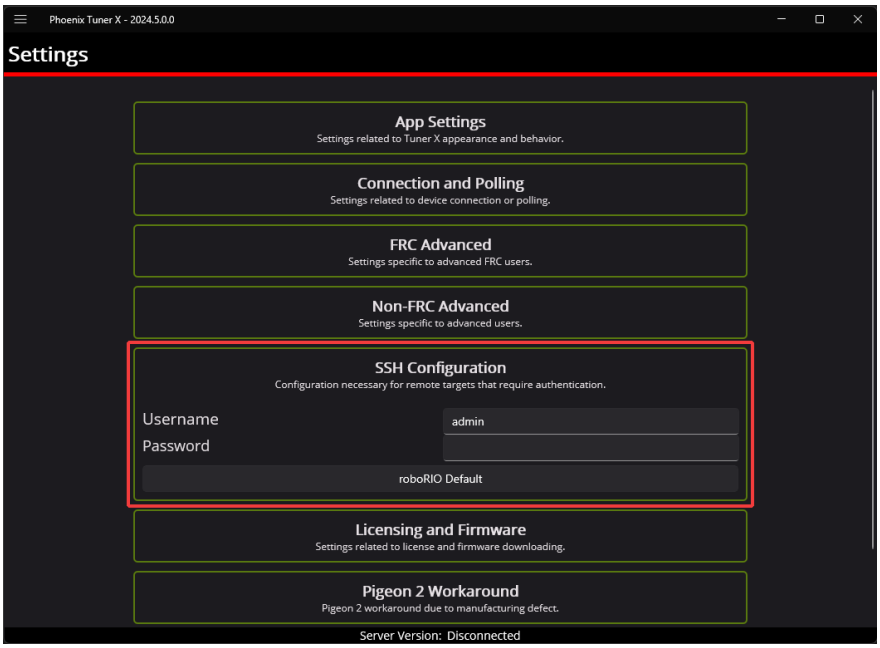

By clicking the arrow, you can change between presets such as:

- *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.

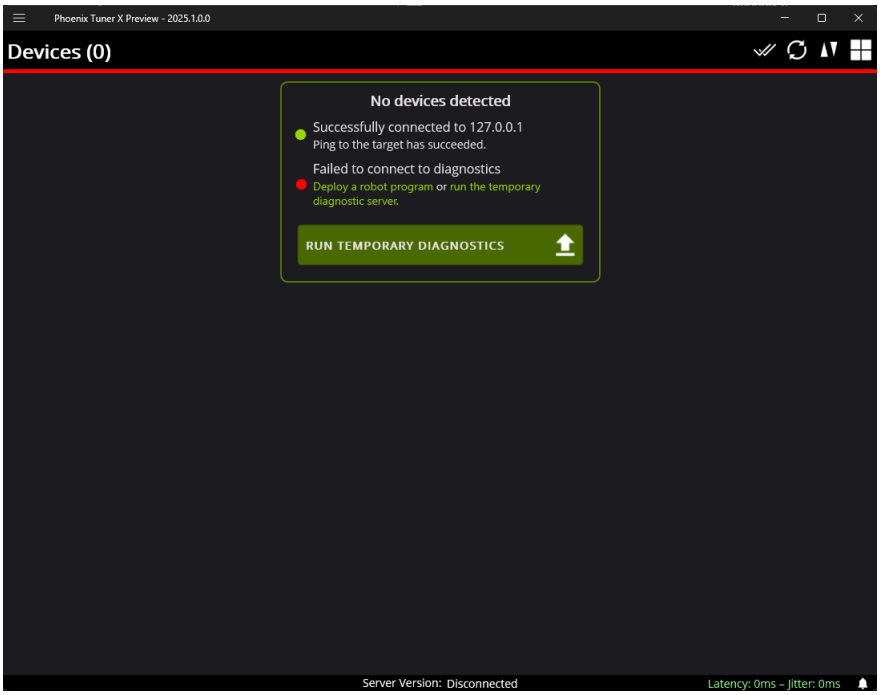
#### Configuring SSH Credentials (non-FRC)

When using a non-FRC robot controller (non-roboRIO) with SSH (instead of the default POST), credentials must be configured in *Settings* for general use.




### 7.1.2 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*.

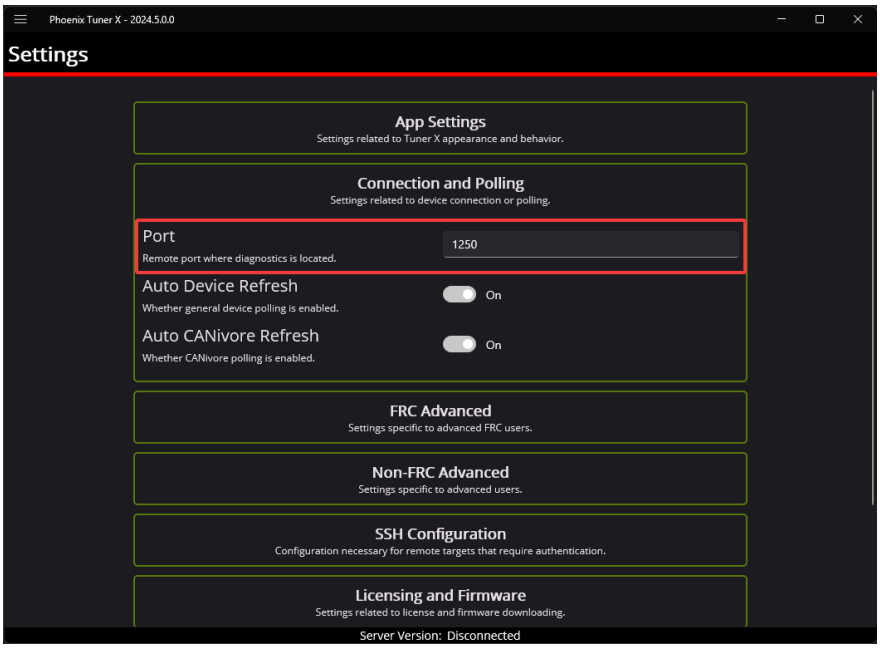


### 7.1.3 Changing Diagnostics Server Port (non-FRC)

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

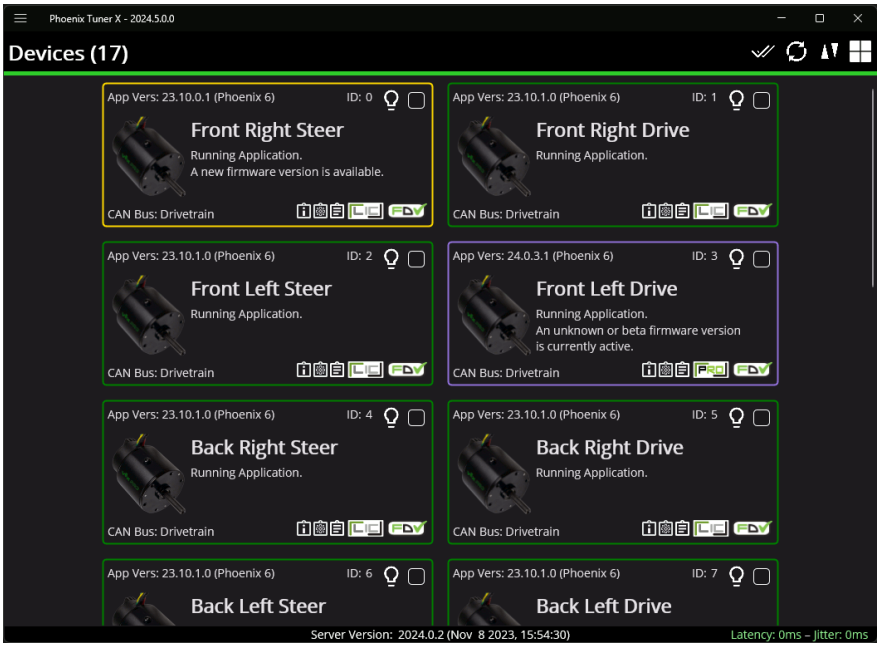
**Important**

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

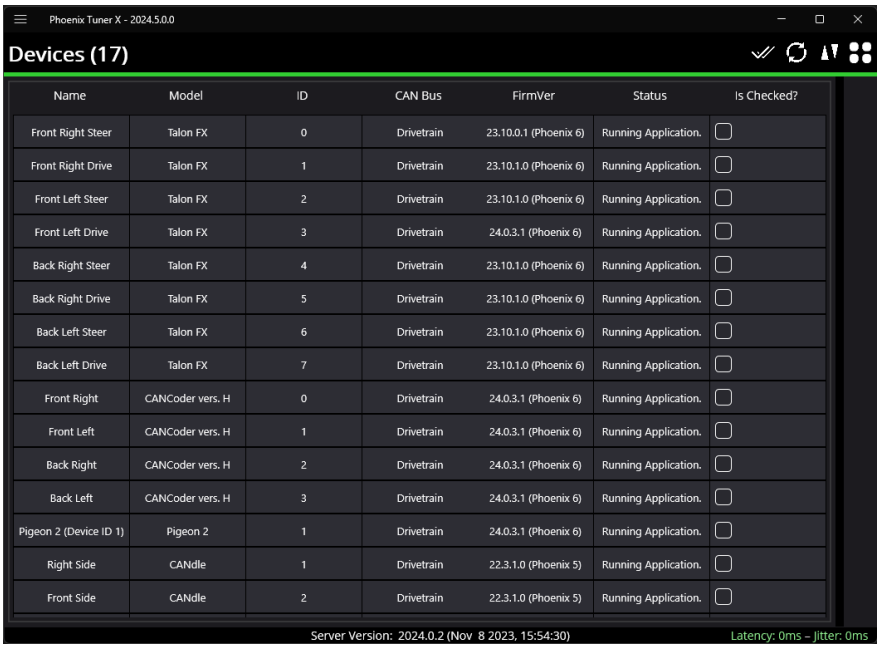



## 7.2 Device List

## Card Layout




## Grid Layout




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).

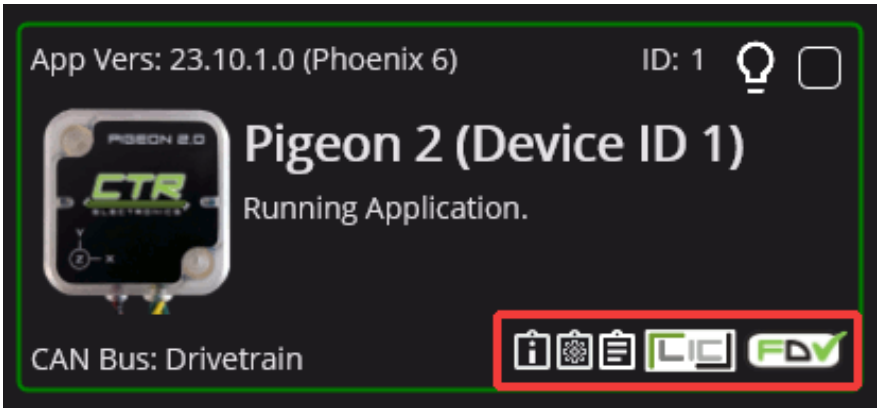
### 7.2.1 Card Colors

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. |

### 7.2.2 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.

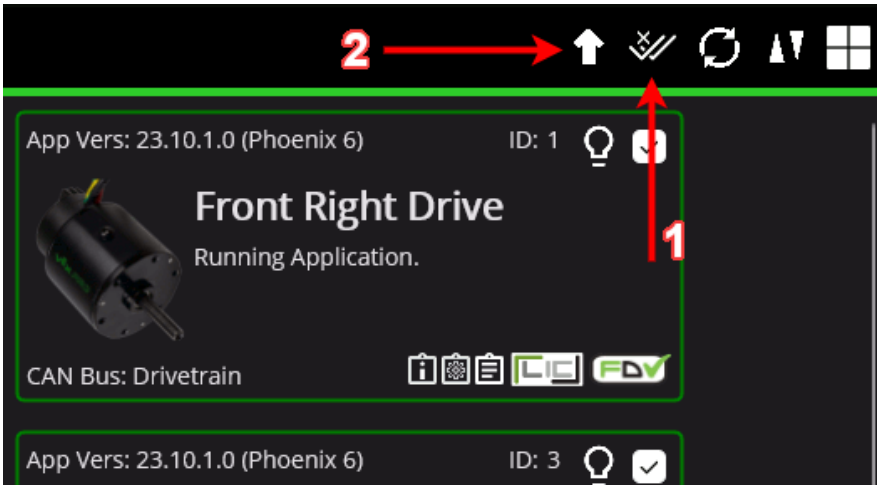
The other major icon in the bottom right of the device card is the licensing indicator. This showcases the *licensing states* and when clicked, will open the licensing dialog.

### 7.2.3 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



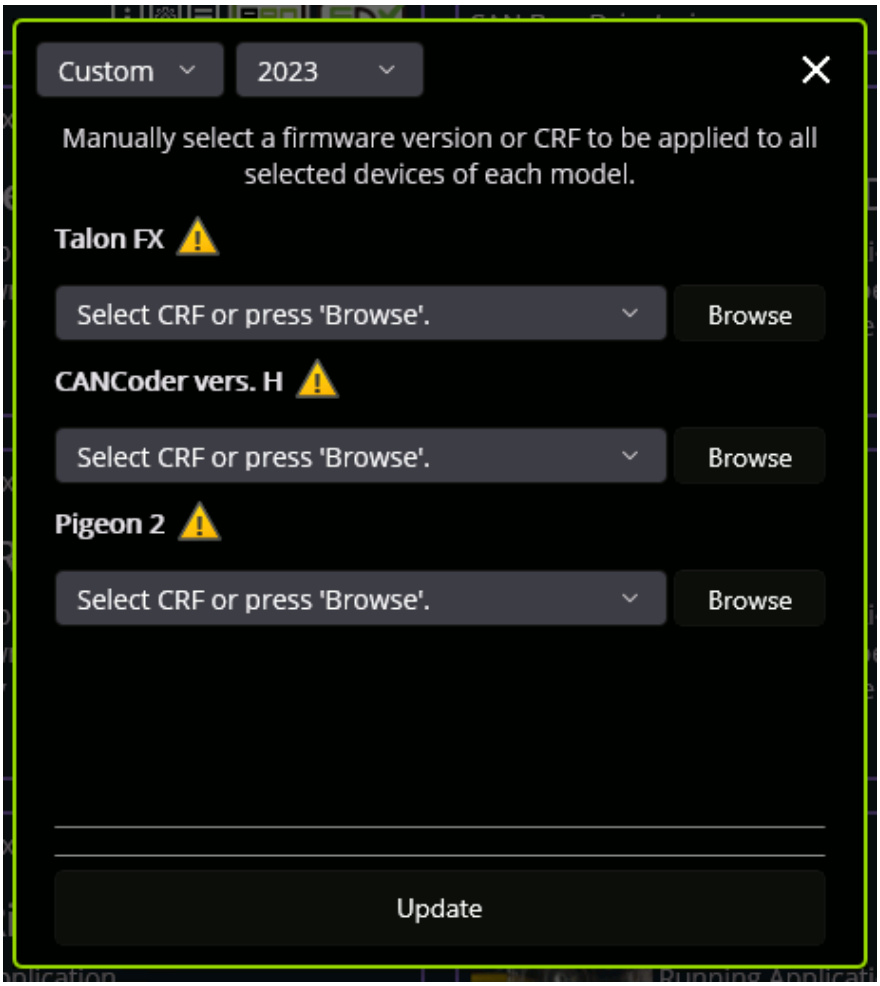

Step 1 in the above image selects all devices of the same models selected (or all devices if no device is currently check-boxed).

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.

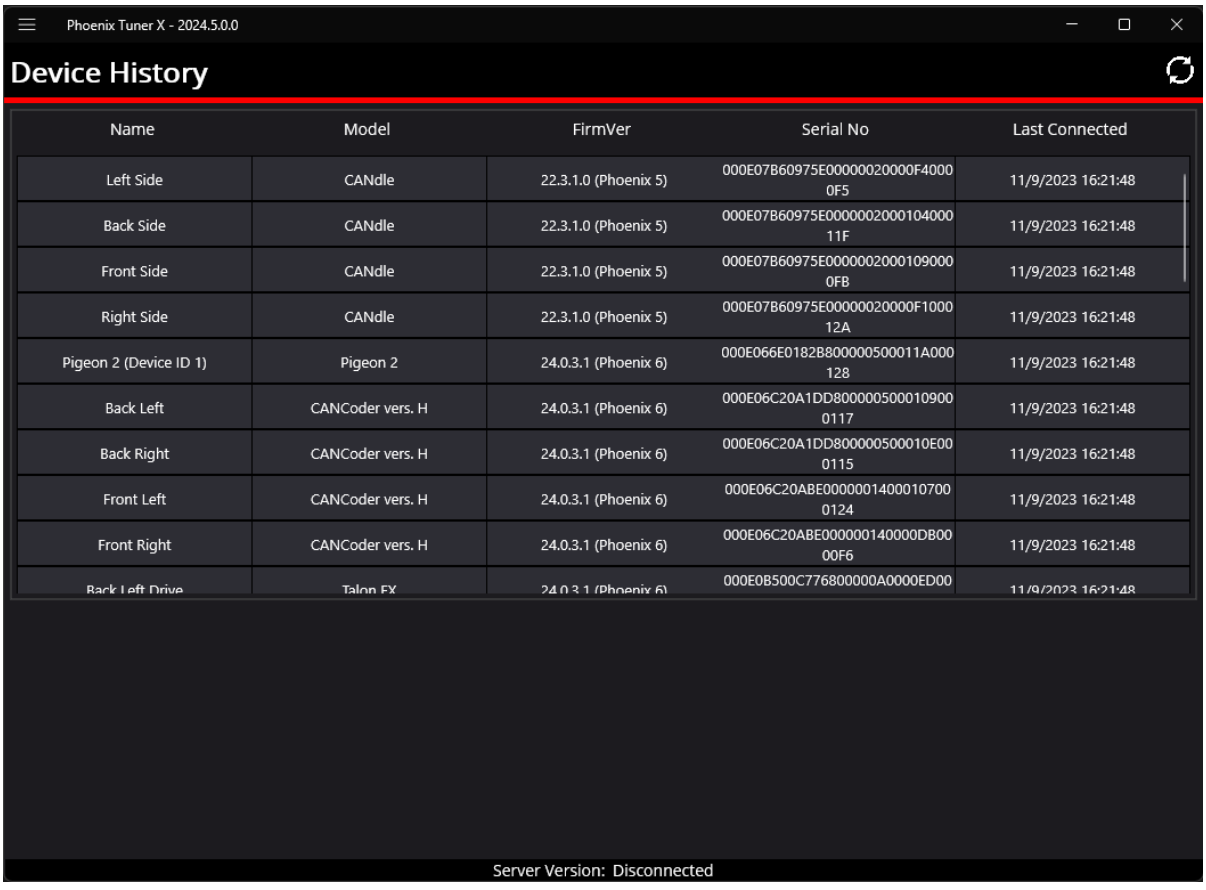
### 7.2.4 Batch Licensing

See *Batch Activating Licenses*

## 7.3 Tuner History

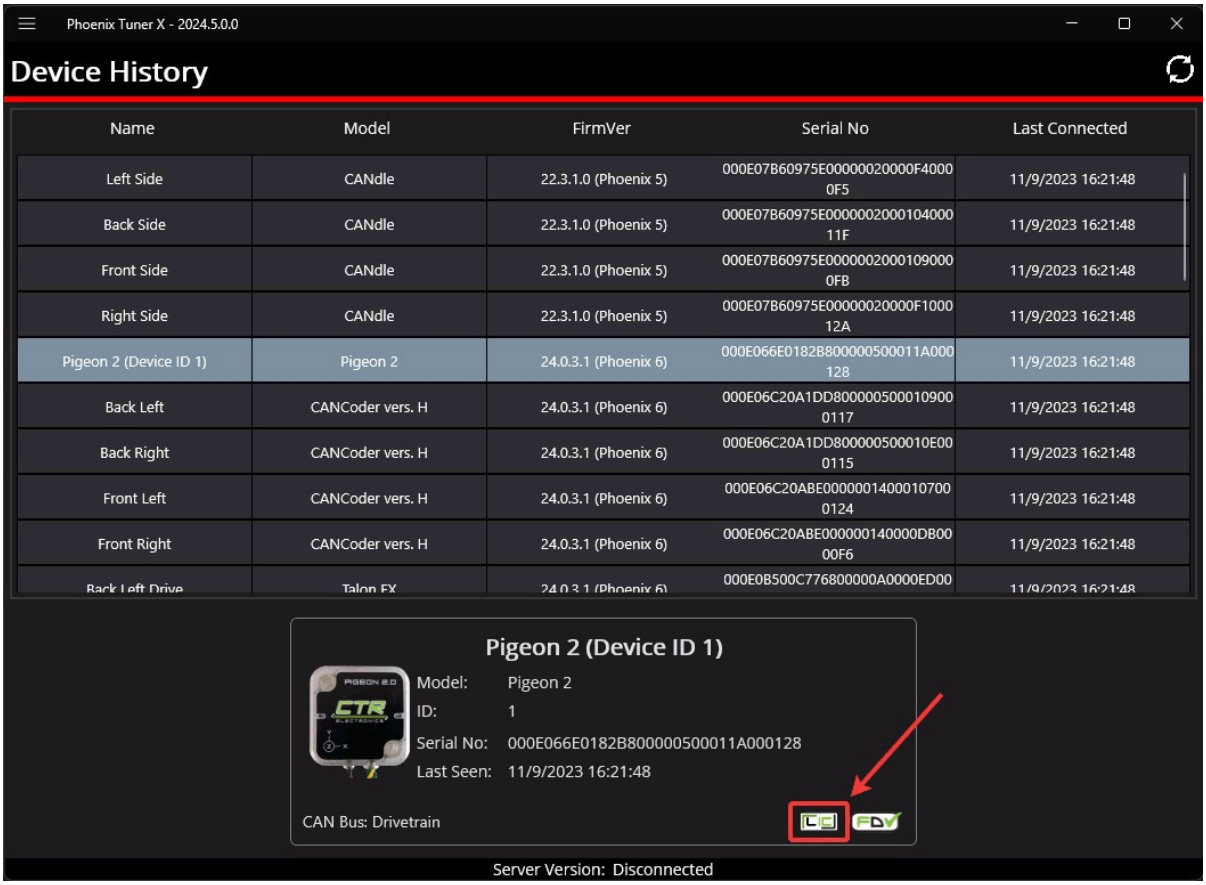
Tuner history provides insight on past connected devices and robot networks. **Device History** allows teams to view previously connected devices and license them without being directly connected to them. Users may wish to license previously connected devices due to a lack of internet connection while being connected to them. **Network History** indicates a list of past connected robot networks.

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.



### 7.3.1 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.

## 7.4 Device Details

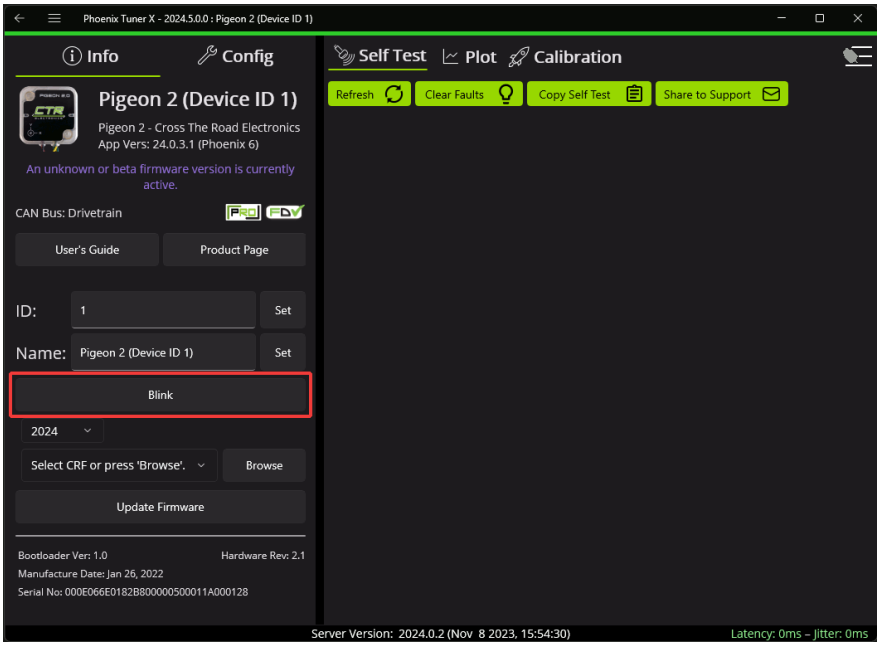
The **Device Details** page can be accessed by clicking on the device card (or clicking on *View more details…* when in grid view). This view allows you to access detailed device actions such as:

- 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® 2.0 Mount Calibration

### 7.4.1 Blinking

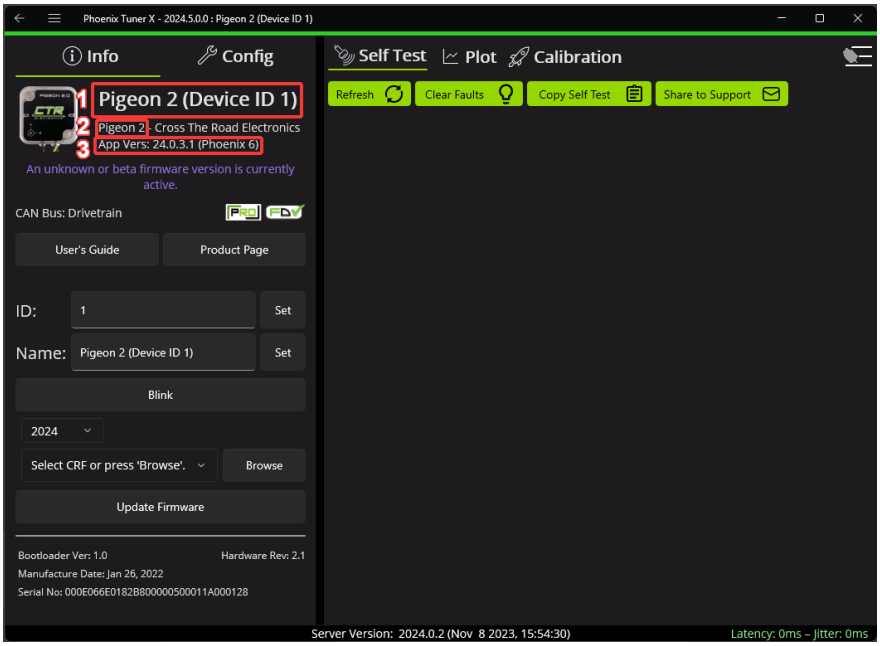
All CTR Electronics devices can be blinked (rapidly flash the LEDs). This can be useful for handling whenever you have duplicate devices using the same ID on the CAN bus.




### 7.4.2 Verifying Device Details

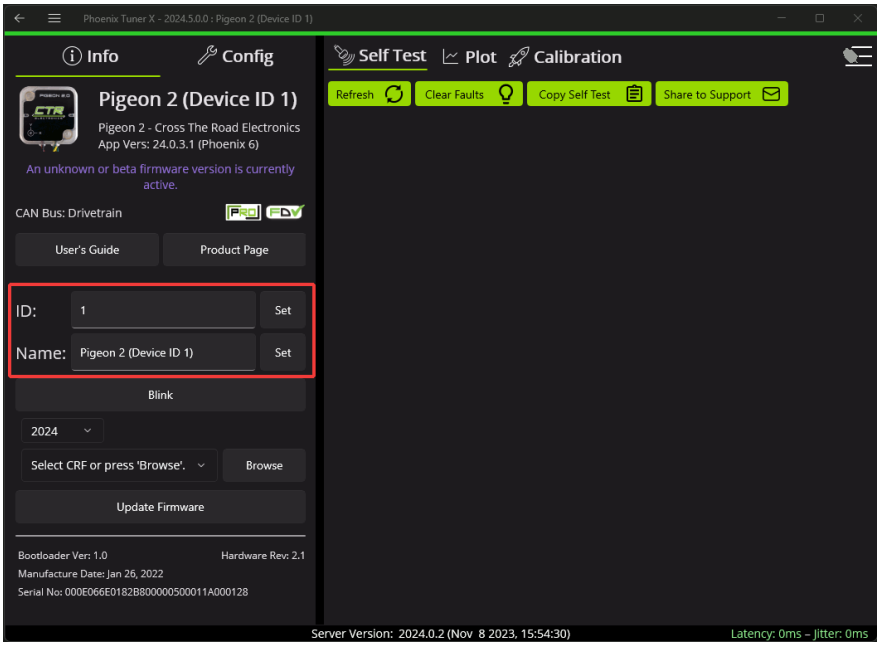
This screen highlights information such as (1) Device Name, (2) Device Model, (3) Firmware Version.

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




## 7.4.3 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.

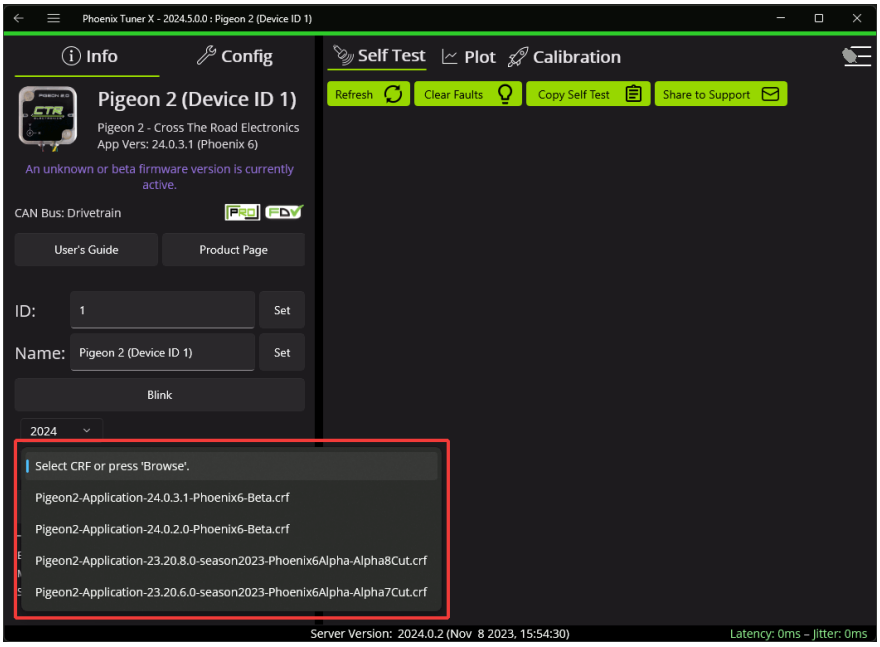


### 7.4.4 Field-Upgrade Firmware Version

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*.

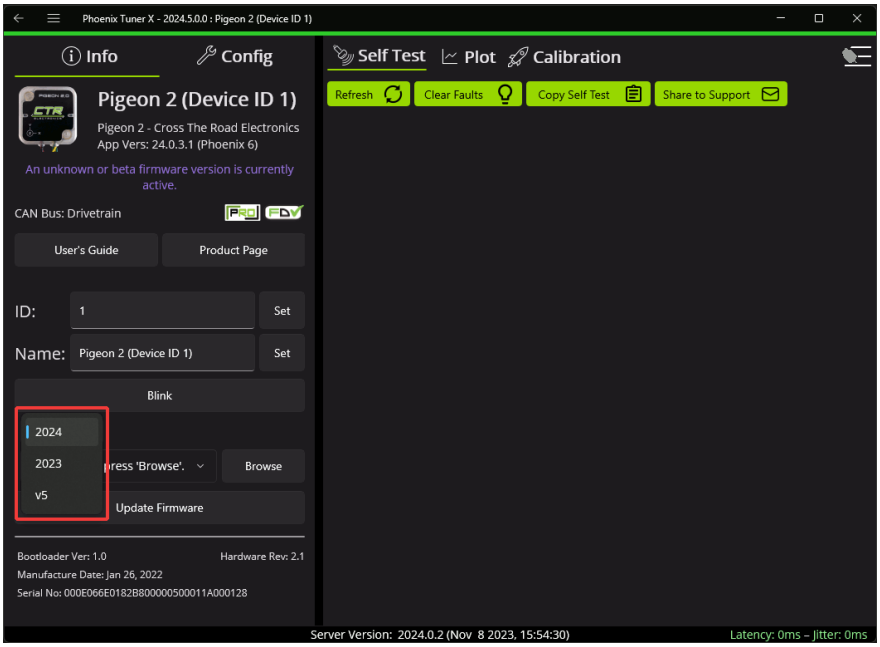
**Important**

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.

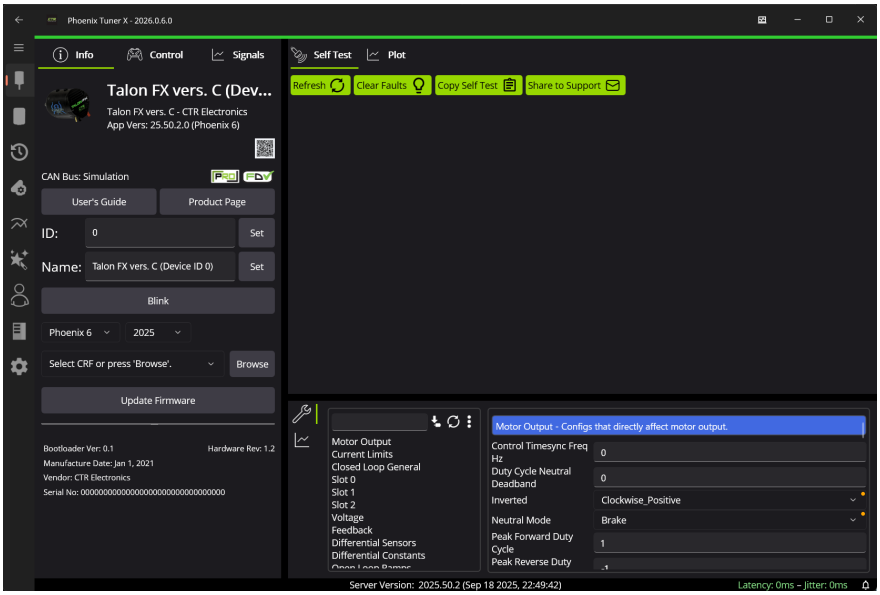
> **Note**
>
> The toggle between firmware years only affects the firmware versions downloaded by Tuner X. Files selected using the “Browse” button are not affected.



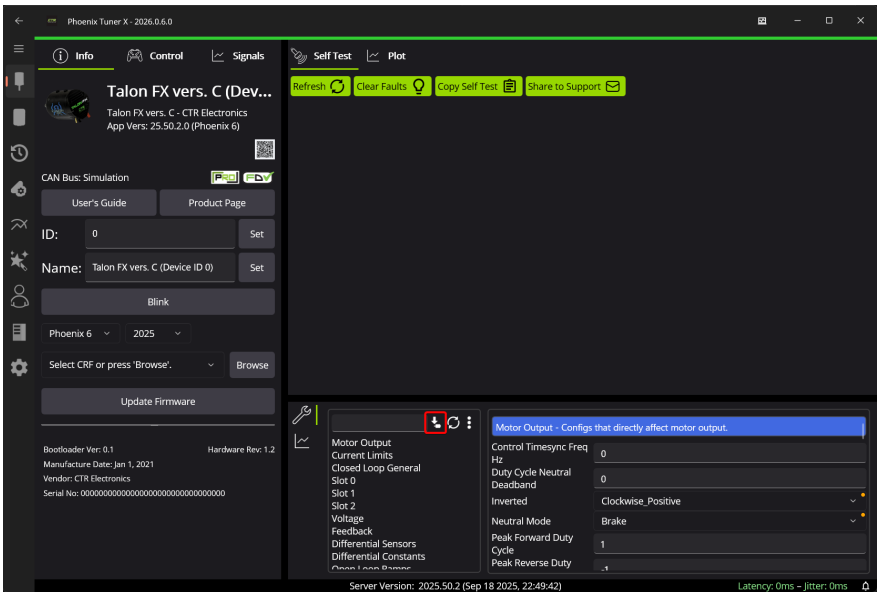

## 7.5 Tuner Configs

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

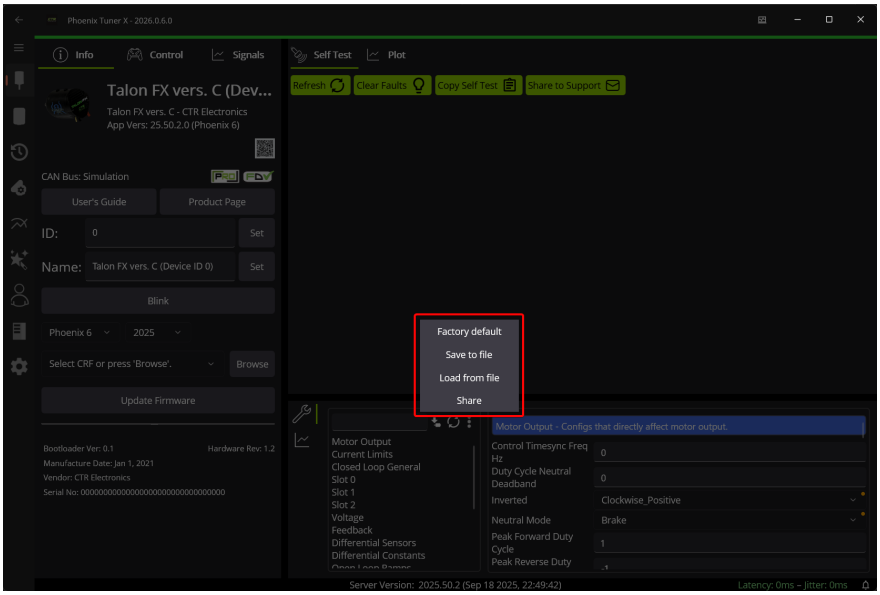
Configs can be viewed, modified, backed-up, restored, and factory-defaulted via the *Configs* tab in Phoenix Tuner X.




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

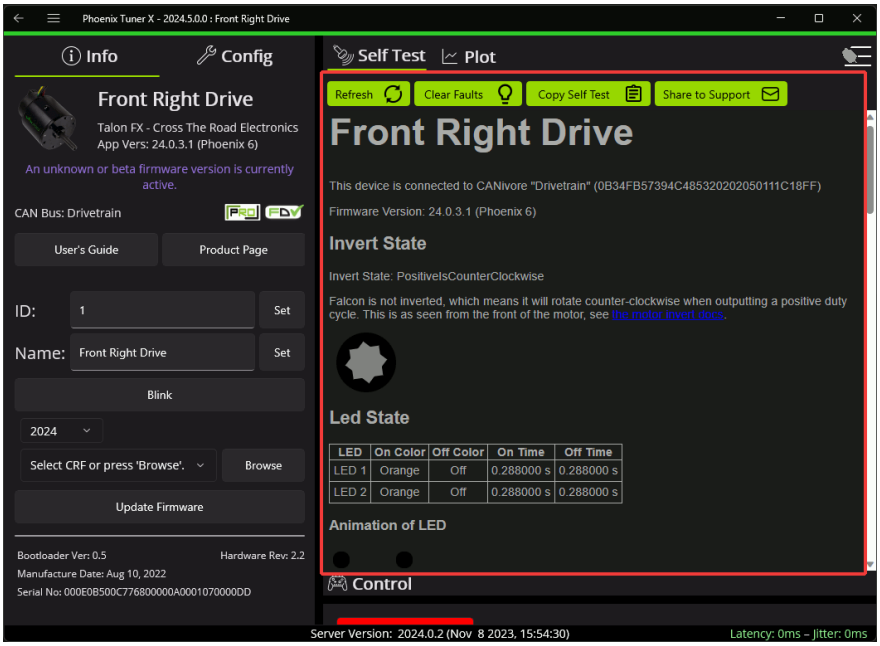



Additional options are available by clicking on the 3-dots icon on the top button bar.




## 7.6 Self Test Snapshot

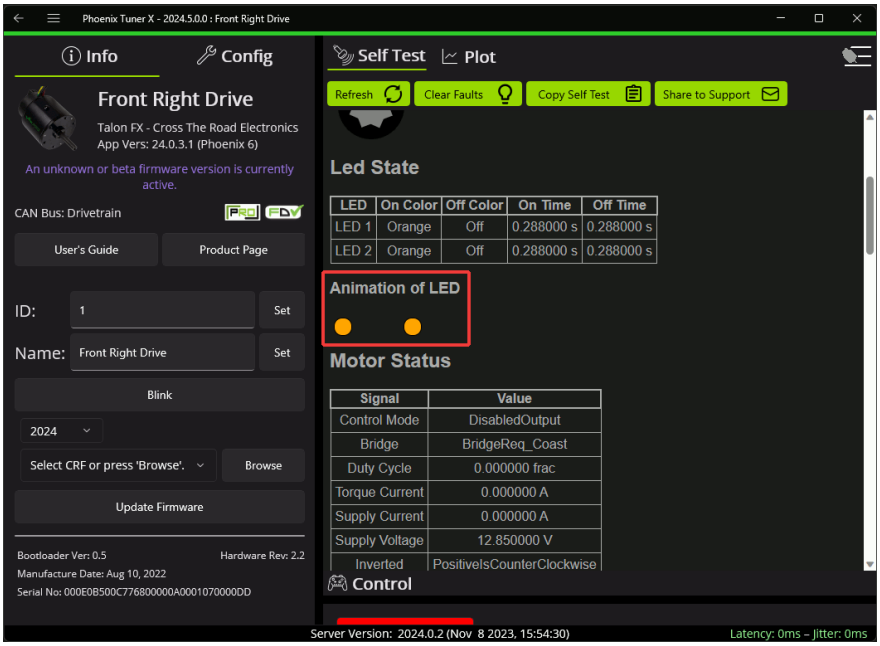
Self Test Snapshot is a diagnostic feature of all supported devices that will show the immediate state of the device. This is extremely useful for troubleshooting and ensuring the device is working properly. Phoenix 6 with Phoenix Tuner X improves upon Self Test by showing the information in clean tables,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




Self Test also includes 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.

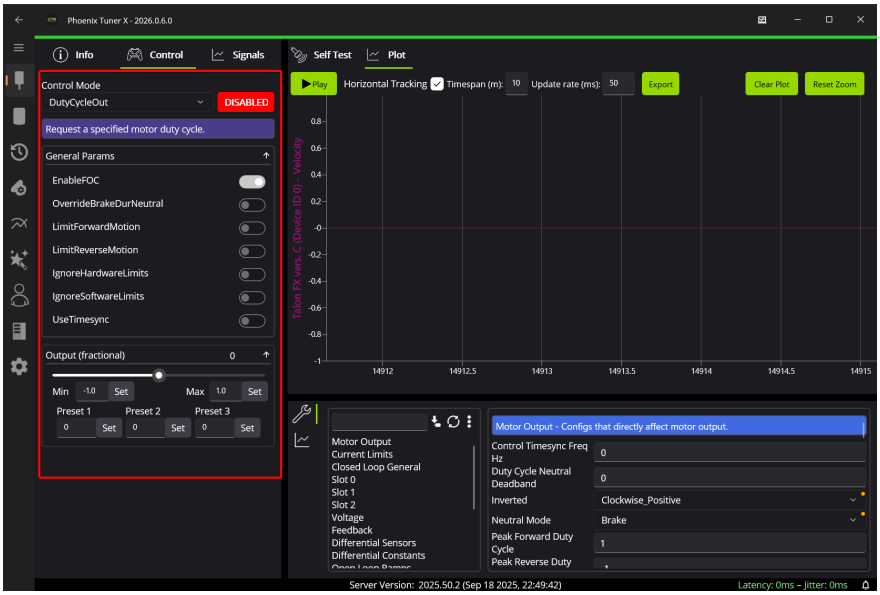
### 7.6.1 Viewing Status LEDs

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.




## 7.7 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.

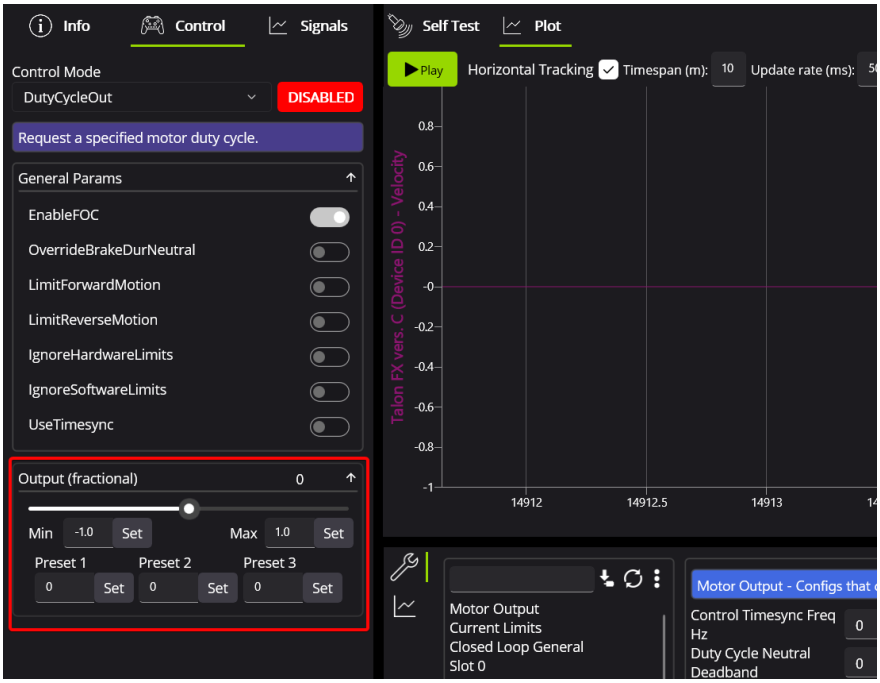



Devices can be controlled by clicking on the red "DISABLED" button, switching it to "ENABLED".

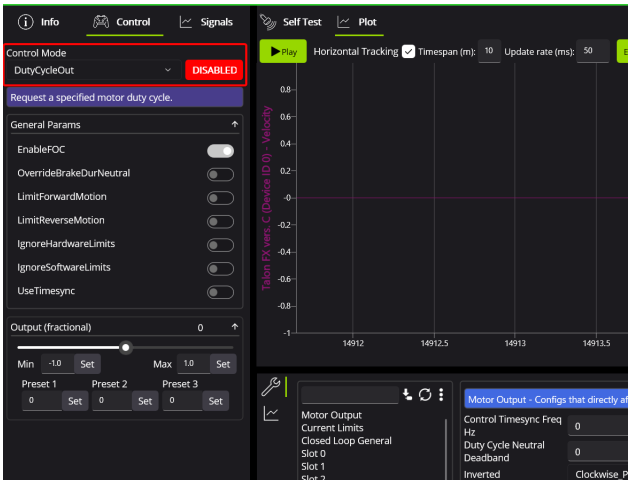
**Important**

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.

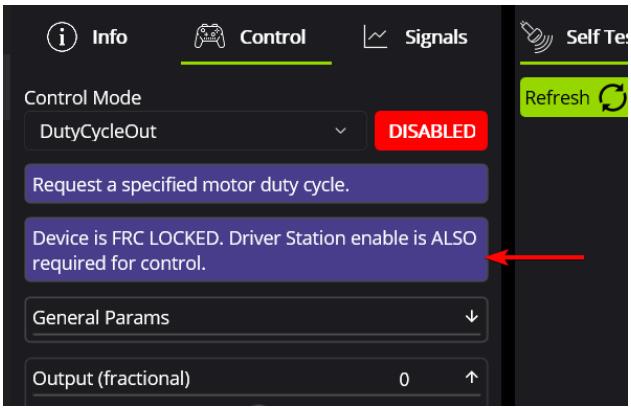



Control modes can be changed using the dropdown below the disable/enable button.



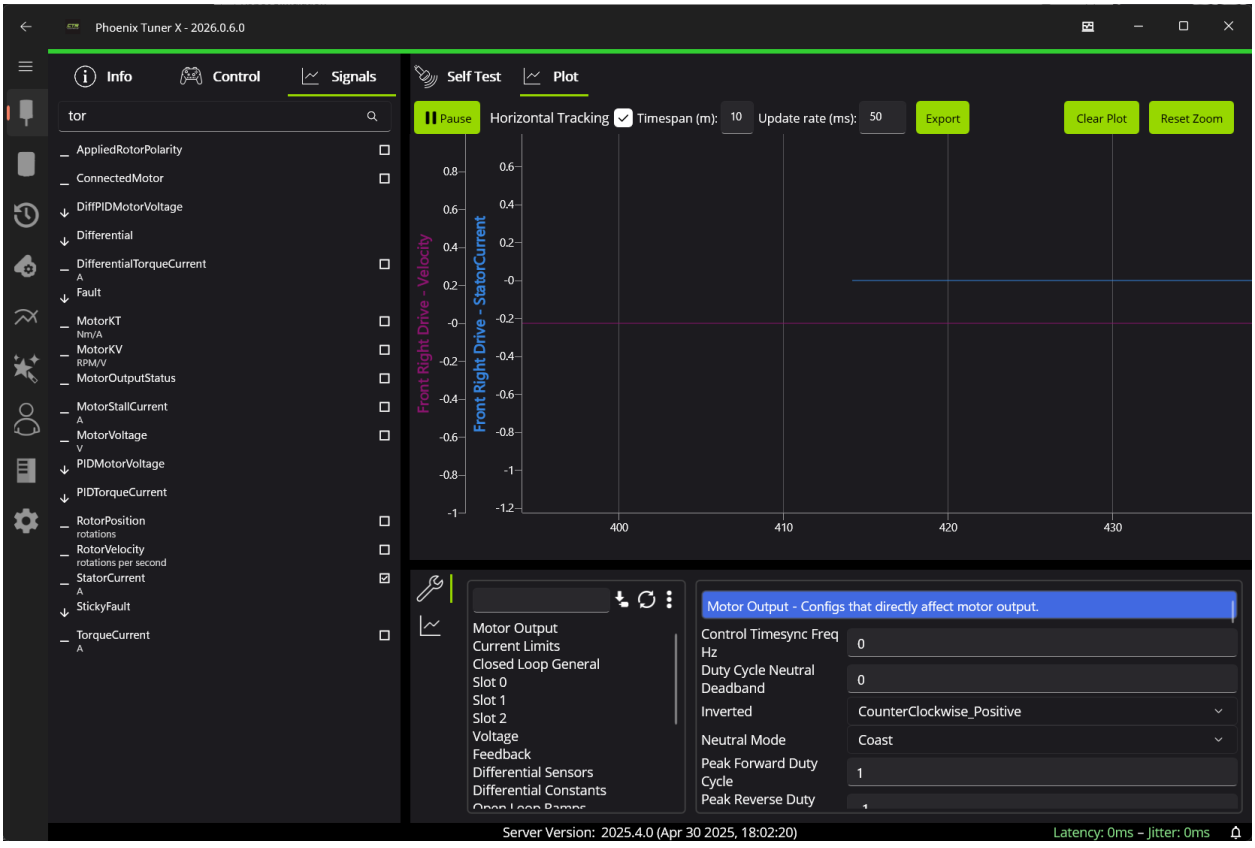
## 7.7.1 FRC Locked

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*.

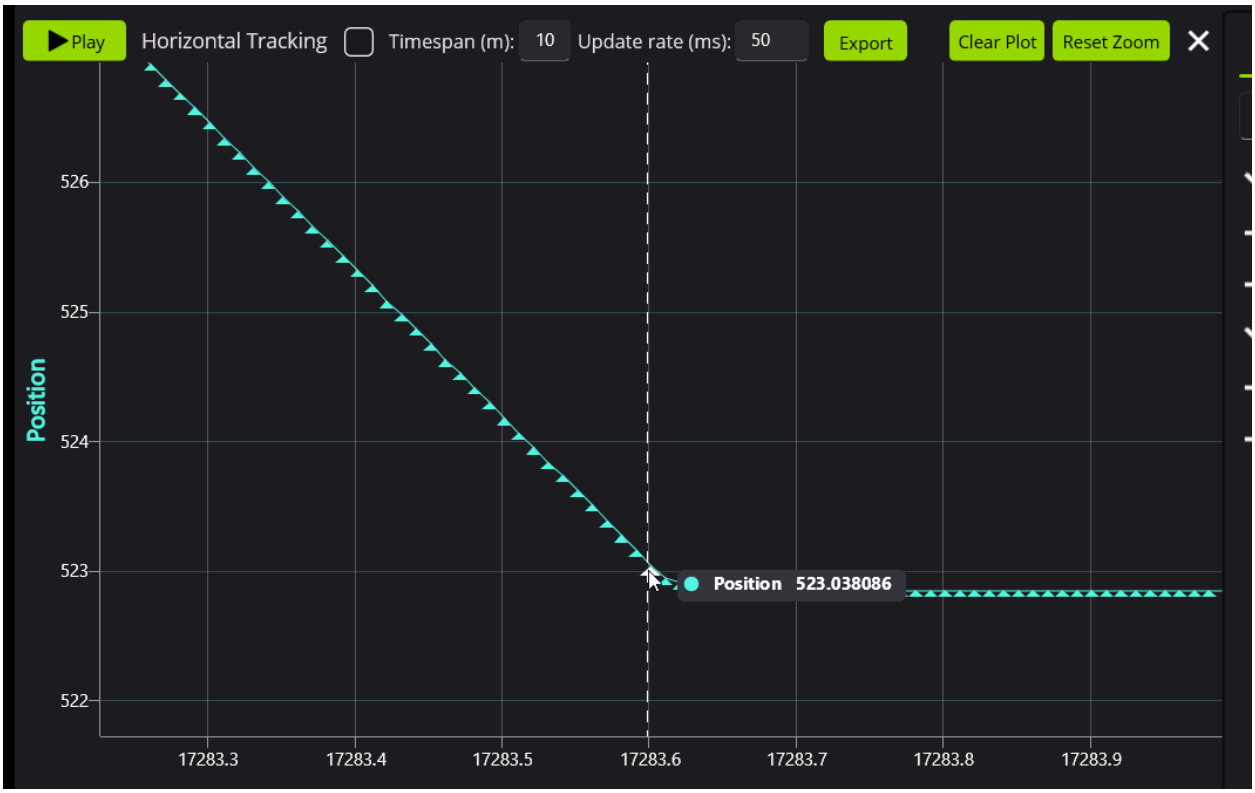



## 7.8 Plotting

Phoenix Tuner X supports an accurate and highly performant real-time plotter. Users can plot and manipulate multiple signals simultaneously. Plotter has undergone strenuous stress testing to ensure hours of plotting operation. This can be used in conjunction with *configs* and *control* for tasks like tuning PID loops.

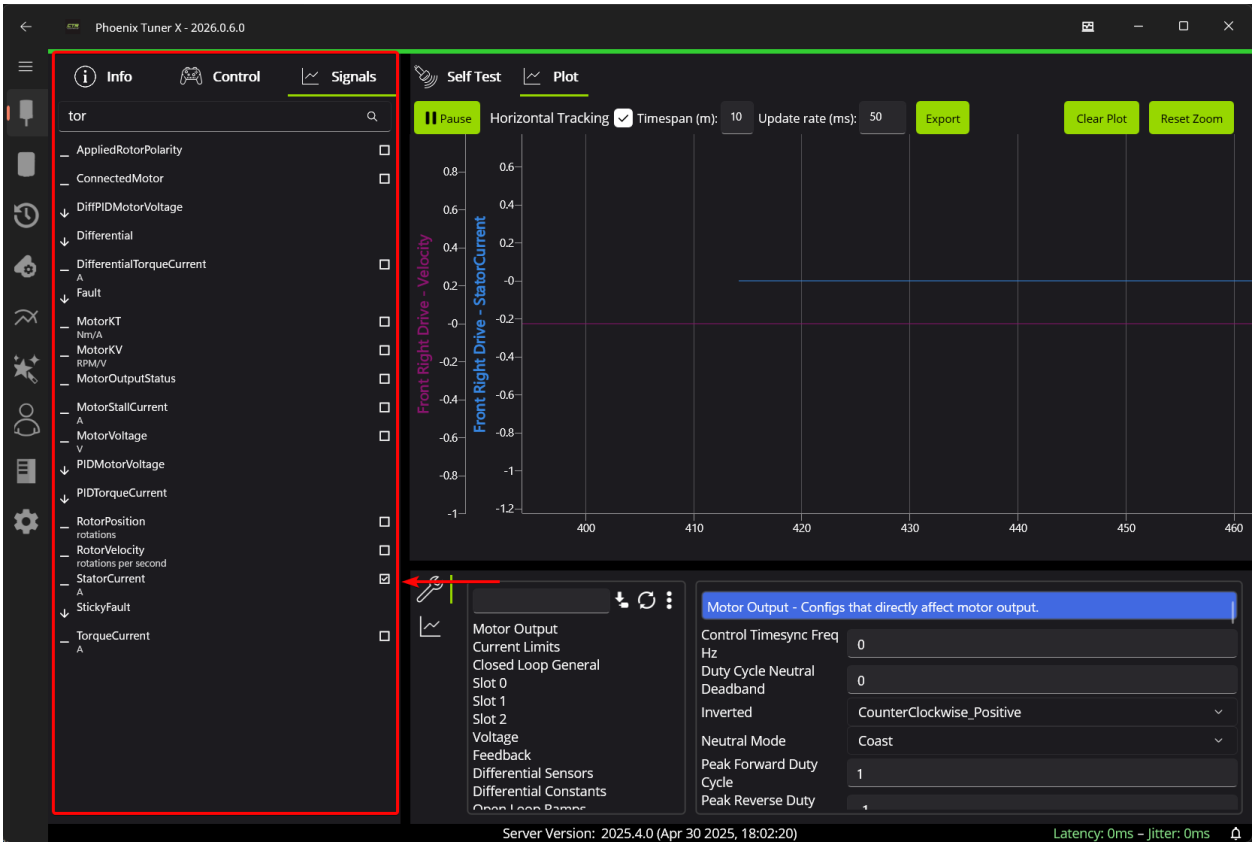



Tuner supports plotting signals as they arrive, ensuring that every visible point is a signal update that has been sent by a device. Users can zoom in and hover over points for a tooltip highlighting the exact value of the datapoint.

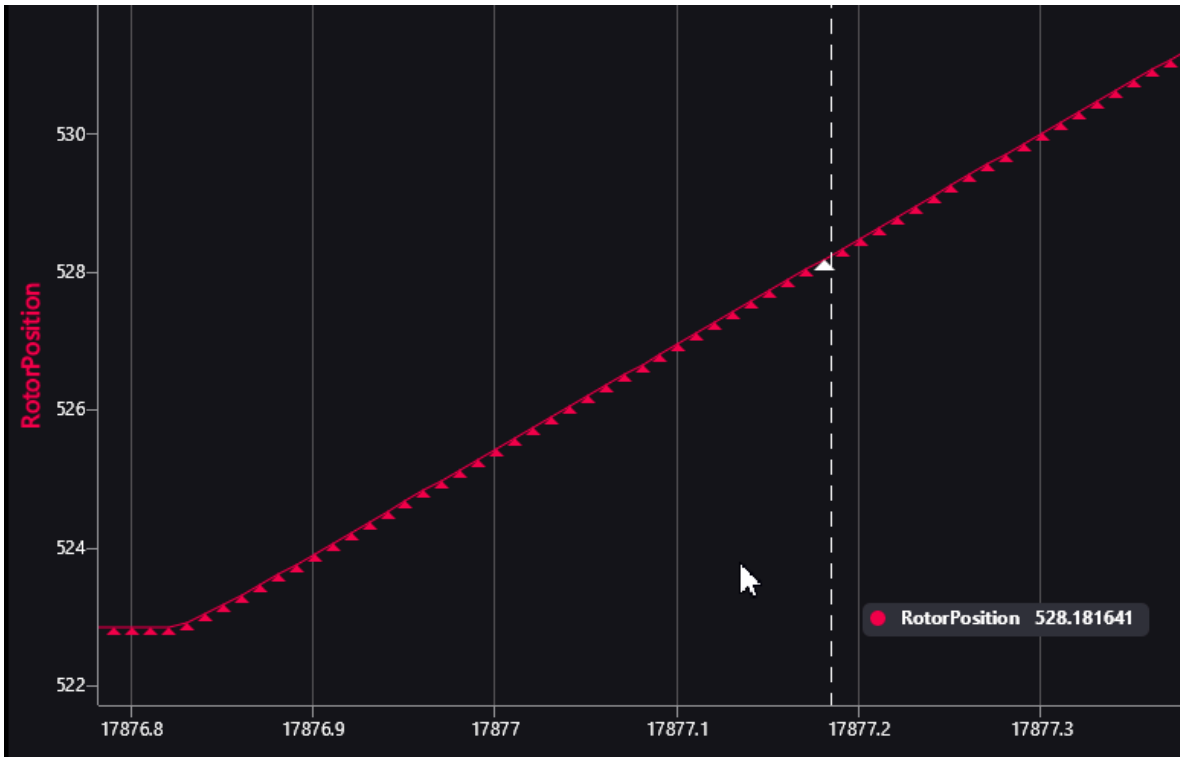



### 7.8.1 Adding a Signal

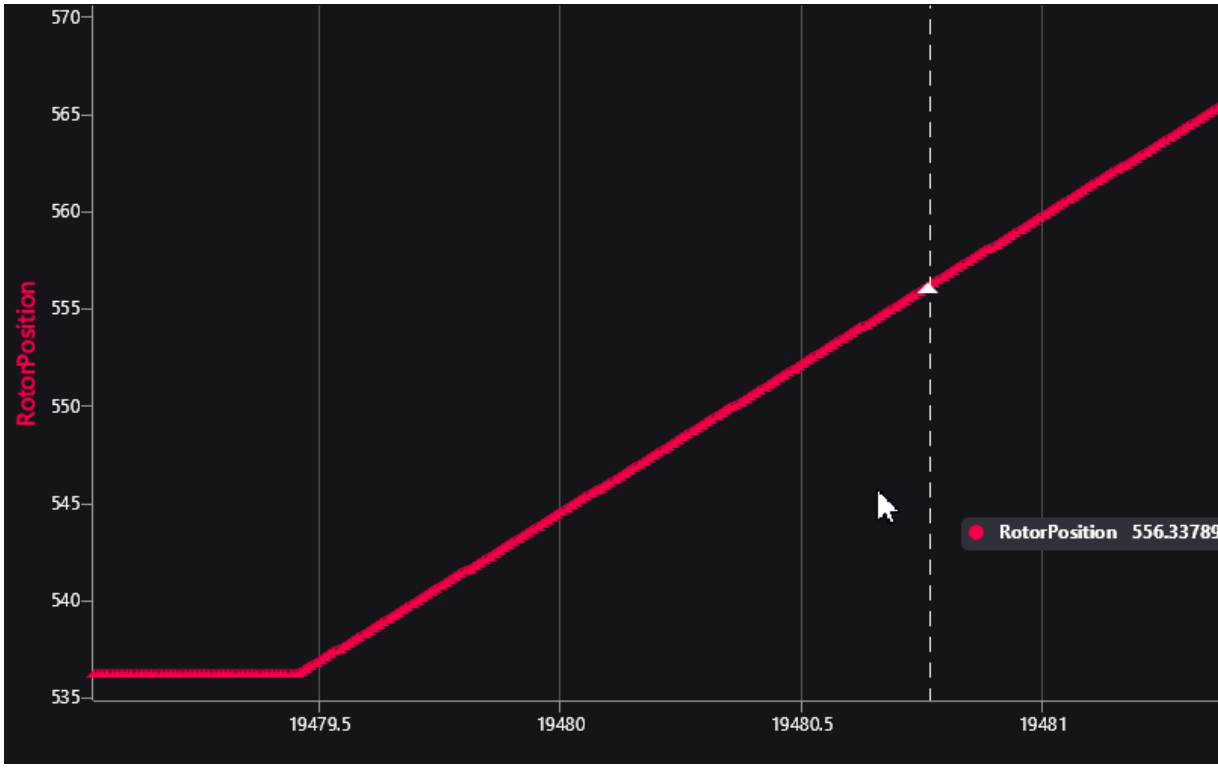
Signals can be added from the right-side menu.



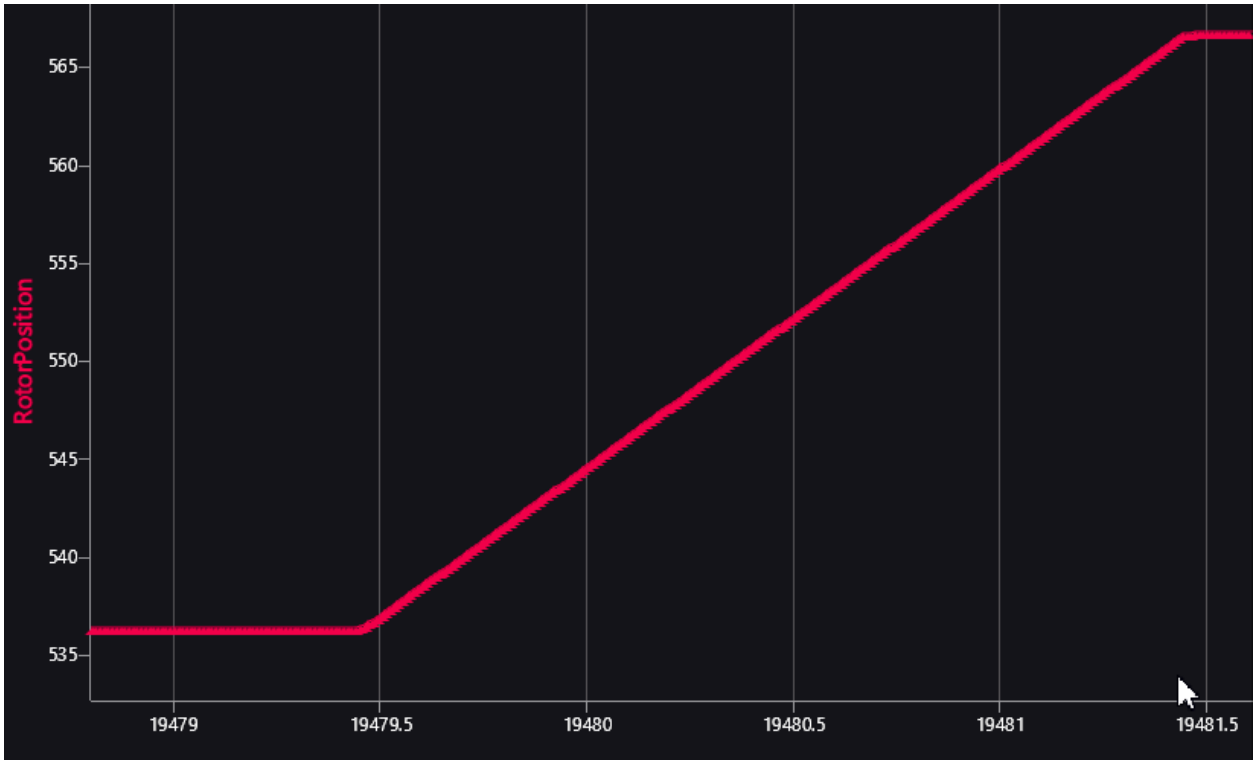

## 7.8.2 Manipulating the Plot



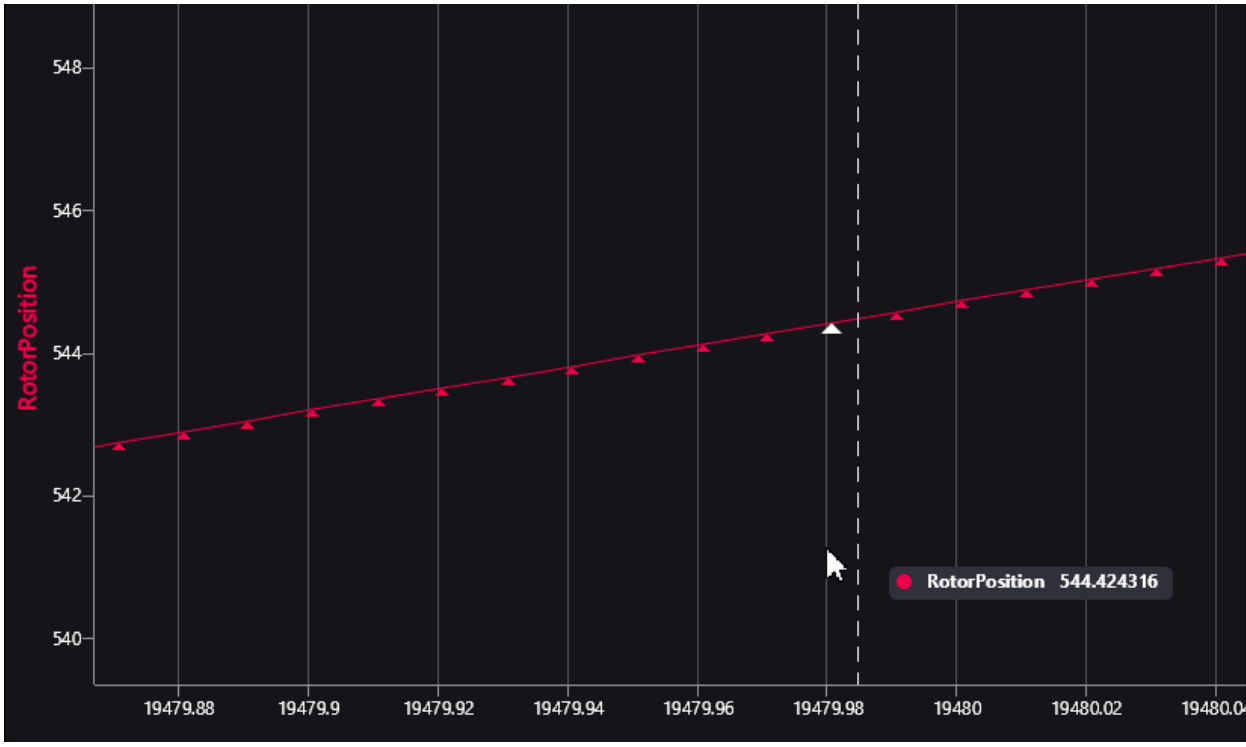

Horizontal Stretch




Vertical Stretch




Panning

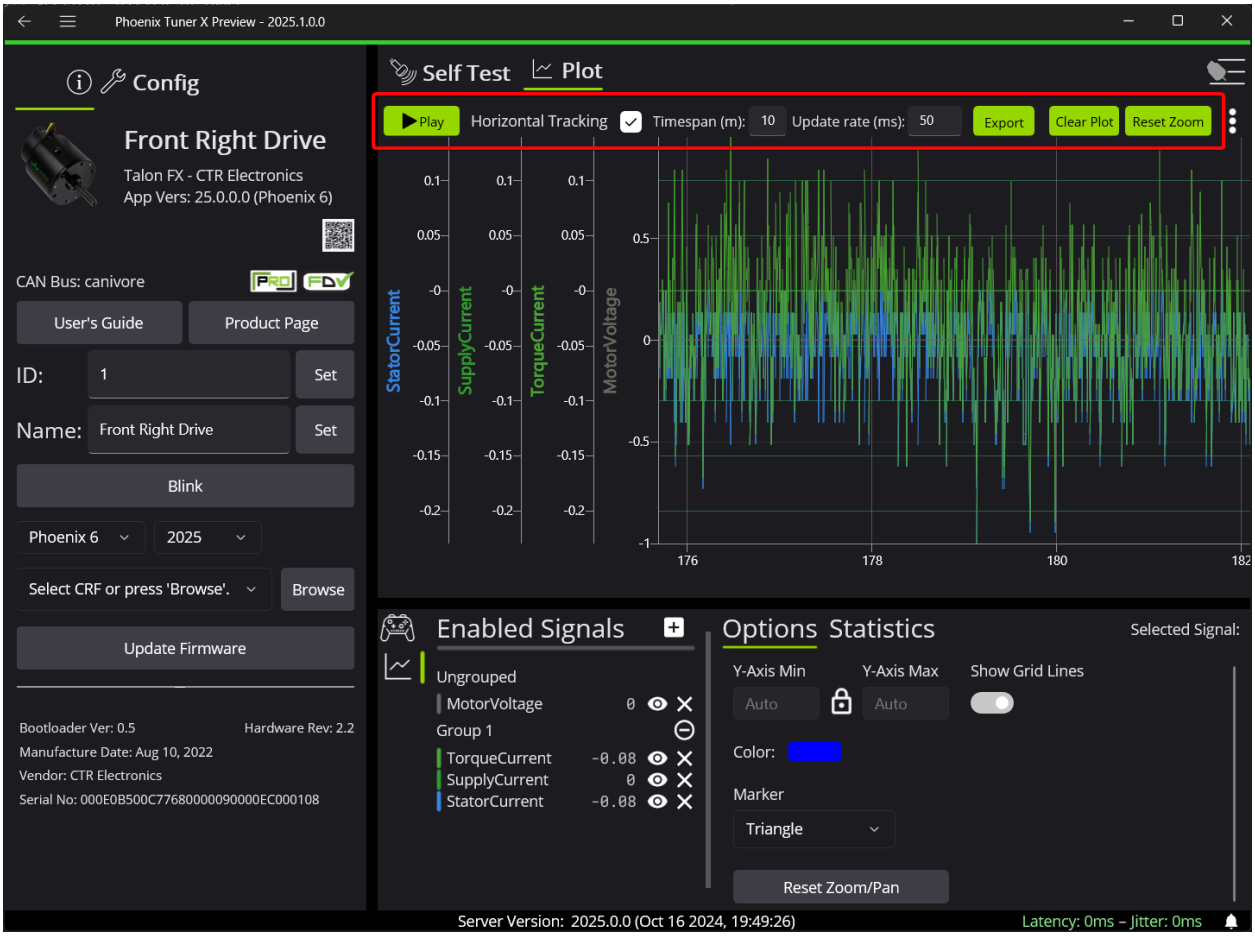



Box Selection

The plot can be manipulated in a variety of ways:

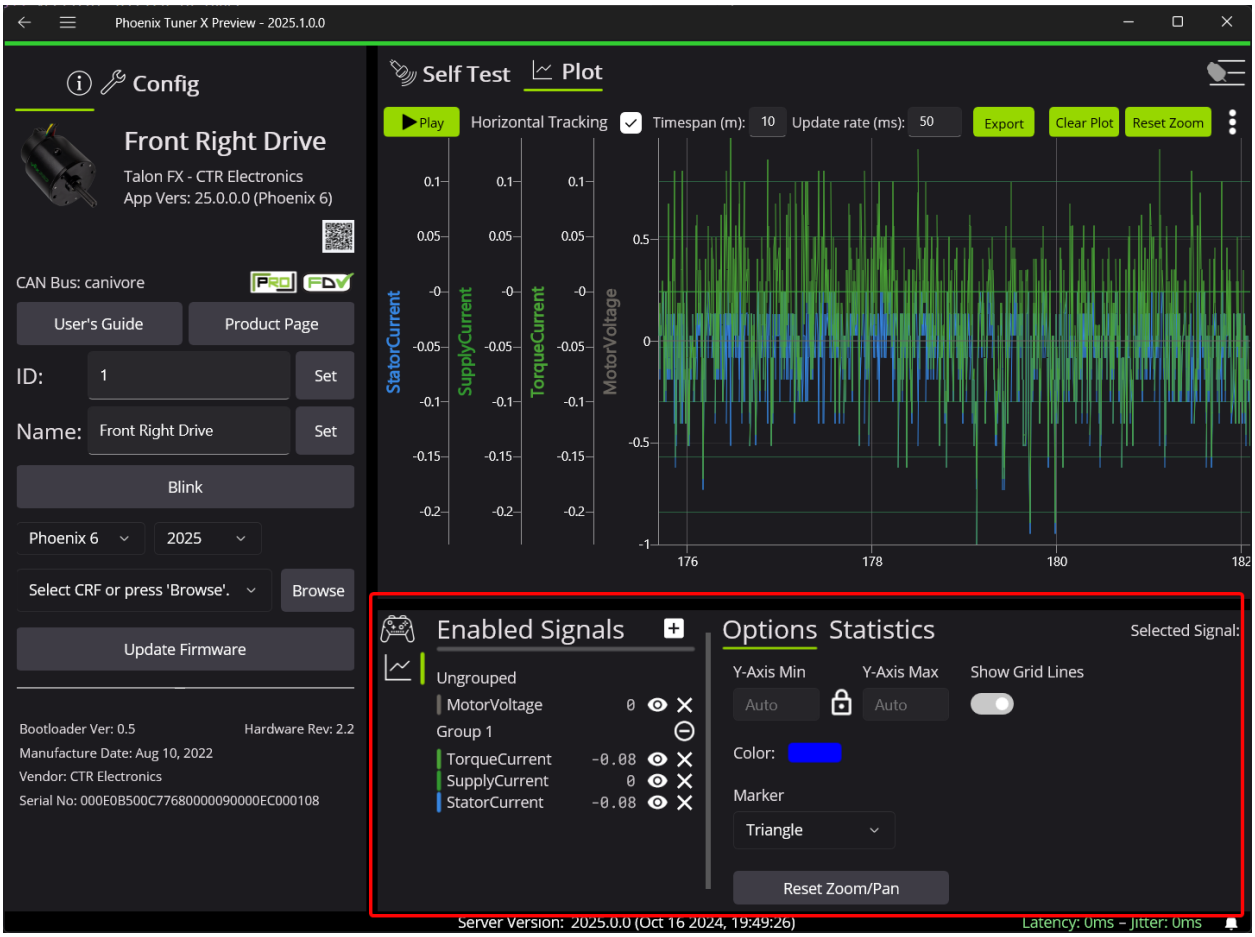
- `Click` + `Drag` to pan around the plot
- `Scroll` over the X-axis or the plot to horizontally stretch the timescale
- `Shift` + `Scroll` over a signal's Y-axis to vertically stretch that signal
- `Shift` + `Scroll` over the plot to vertically stretch all visible signals' Y-axis
- `Ctrl` + `Drag` to pan across all signals' Y-axis

### 7.8.3 Additional Customizations




The topbar of plotter contains a variety of options used for controlling data collection.

- **Play**: Play/Pauses the plotter. Points are **not** collected when the plotter is paused.
- **Horizontal Tracking**: Whether the horizontal axis should be locked to the most recent point. This automatically turns off when pan operations occur. Users can click this checkbox to keep their current Y-axis min/max but resume seeking at the beginning.
- **Timespan**: How long points should be kept before being discarded. Increasing this value will result in increased memory usage.
- **Update Rate**: How often to fetch points from the diagnostic server. Update rates larger than 100ms may result in lost points and lower update rates will result in increased CPU utilization.
- **Export**: Export currently visible signals to CSV.
- **Clear Point**: Remove all points and reset the plot back to its default state.
- **Reset Zoom**: Keep current points but reset horizontal and vertical zoom to its defaults



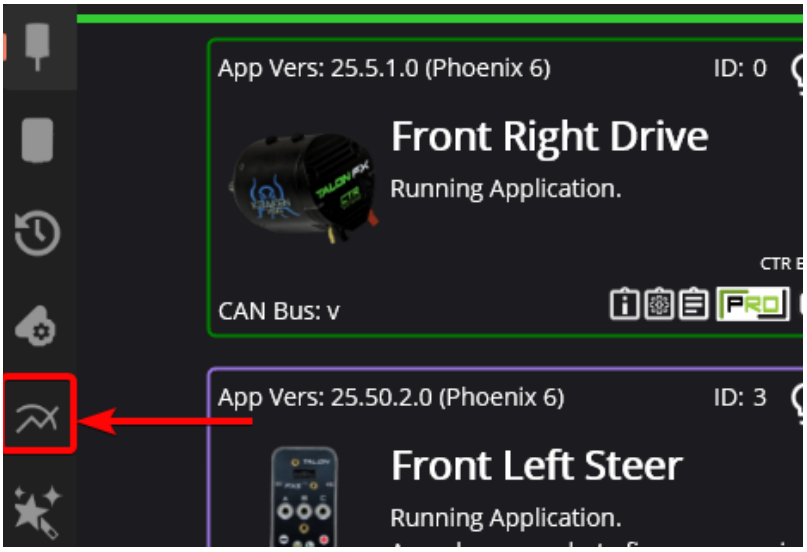

A plot customization tab is available at the bottom of the device view. This tab allows users to group signals together (create a group with the `Plus` icon, and then drag signals over the group name), customize the color of the signal, explicitly set min/max, etc.

Additionally, statistics for a signal can be viewed under *Statistics*.

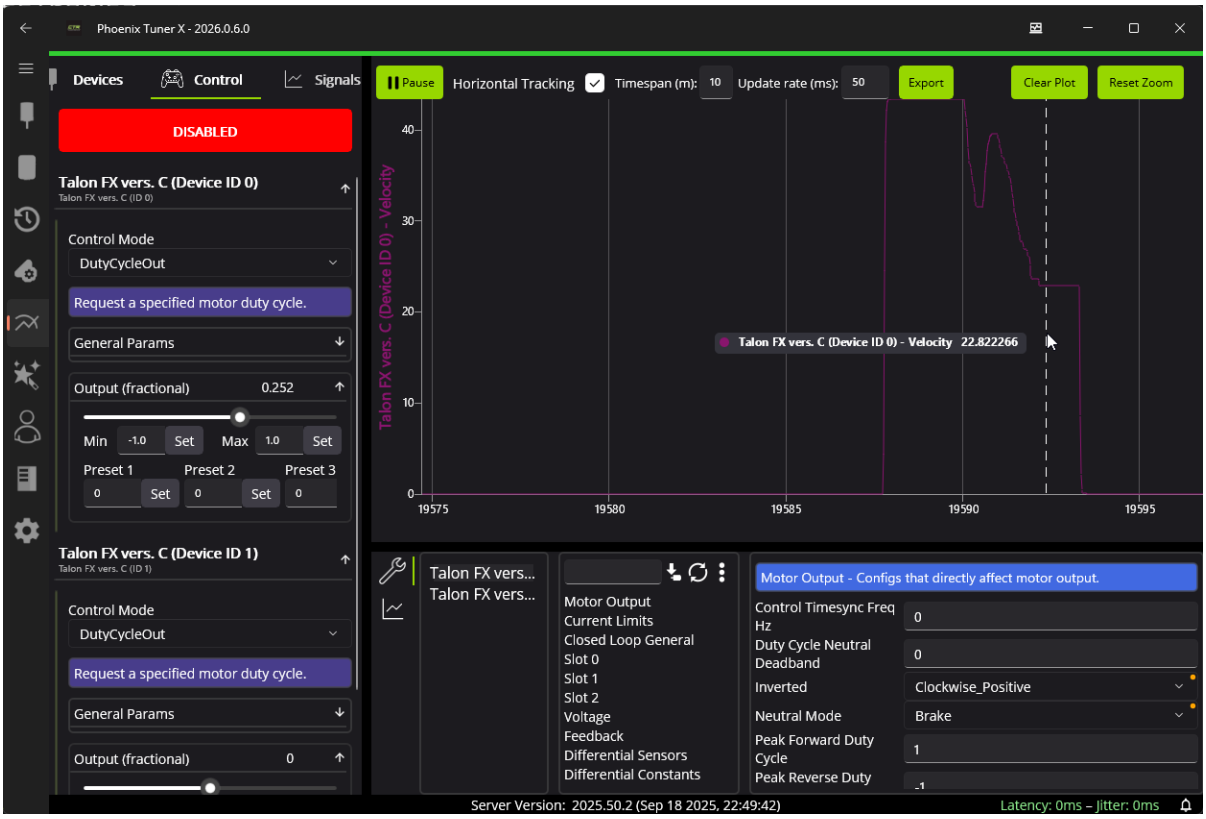



## 7.9 Multi-device Plot & Control

Multiple devices can be controlled and plotted simultaneously using the *Signal & Control* tab in Phoenix Tuner X. The interface mirrors the existing device details page, with dropdowns or comboboxes for choosing devices in a specific context.

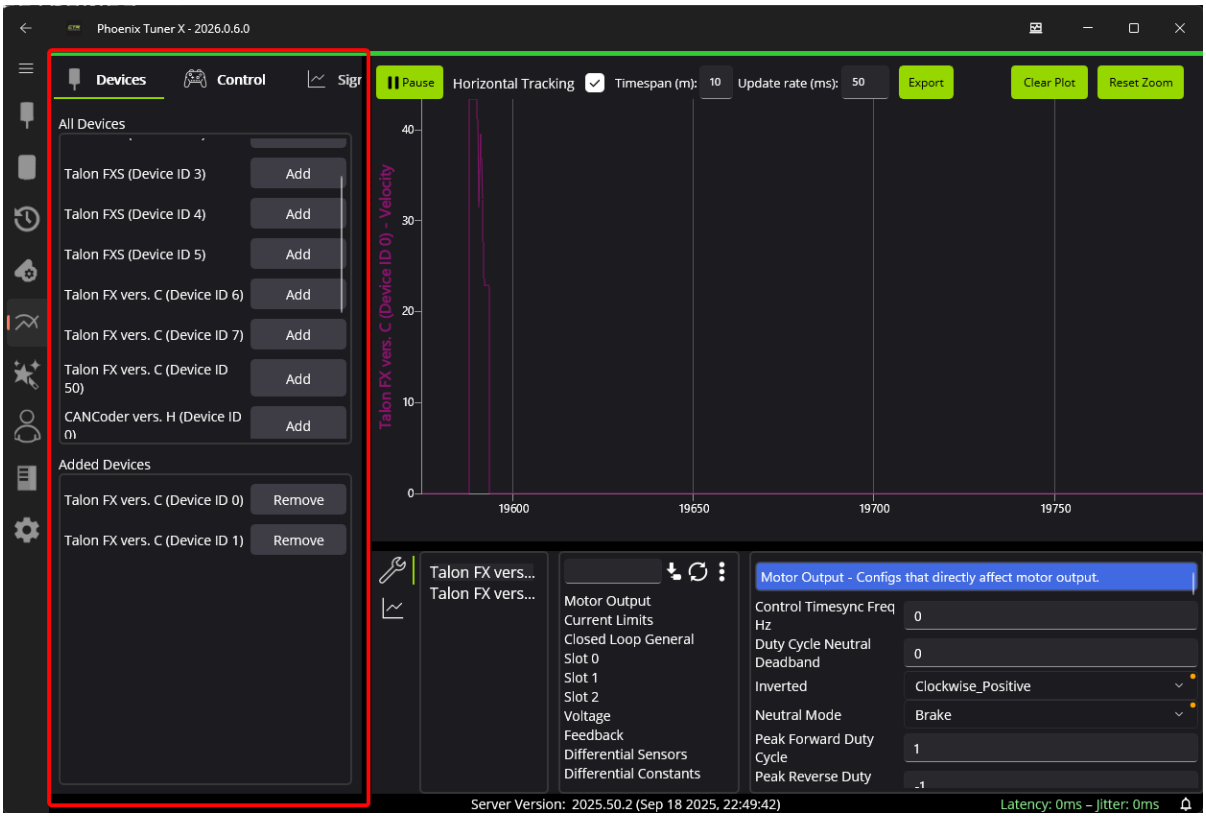


The *Signal & Control* interface can be incredibly useful when tuning subsystems with multiple motors.



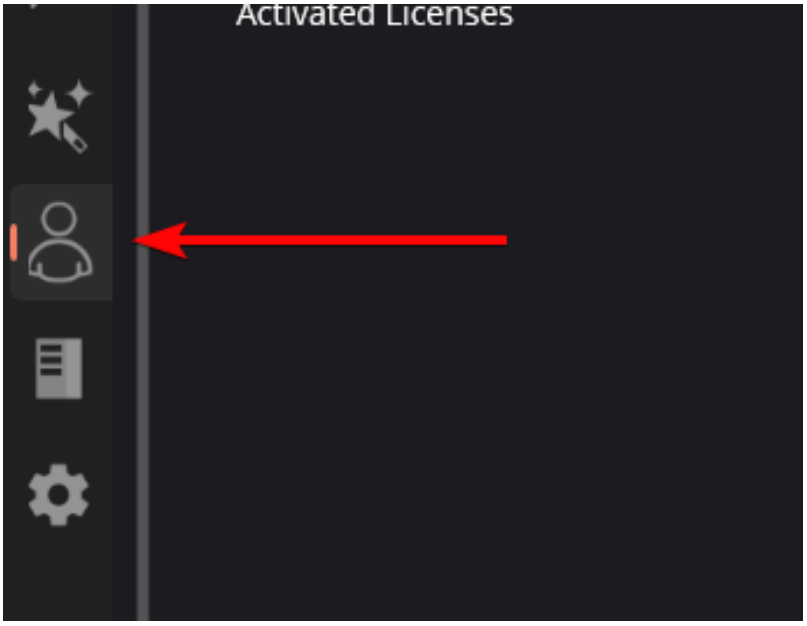

### 7.9.1 Adding Devices

Devices can be added or removed to the instance using the *Devices* tab on the left-hand side.




## 7.10 Profile Page

The profile page gets the user access to account management and is where *team numbers are assigned to a season pass*. This can be accessed by clicking the profile icon in the left-side menu.



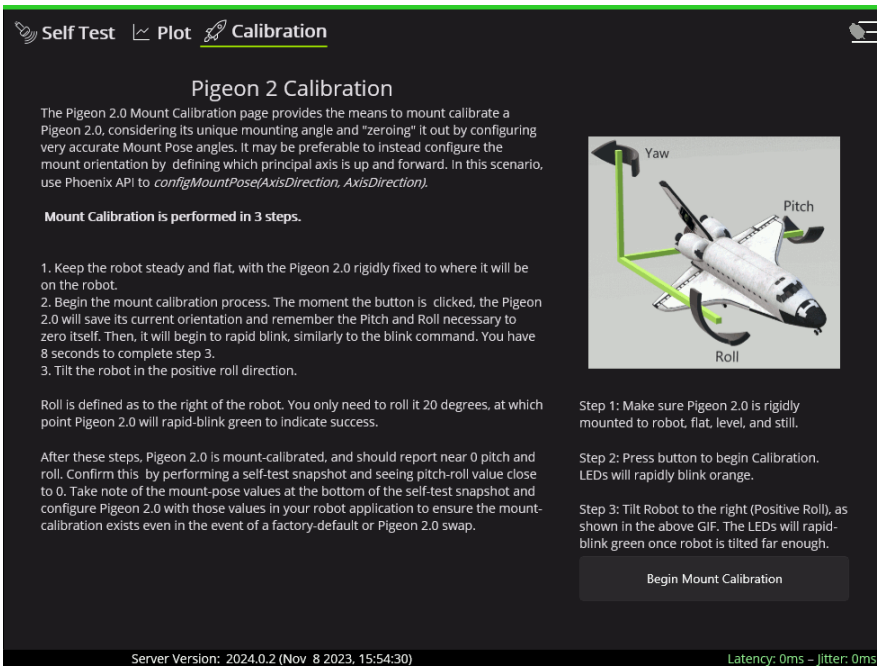

Users can use this page to see how many seats they've redeemed for their season pass, how

many licenses are available, how many licenses have been redeemed, and much more. This is also where the user can manually log out of their account, by clicking *LOGOUT* in the top-right corner.

## 7.11 Pigeon® 2.0 Mount Calibration

Mount Calibration can be performed once the Pigeon® 2.0 has been mounted to the robot. This 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 Mount 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*.



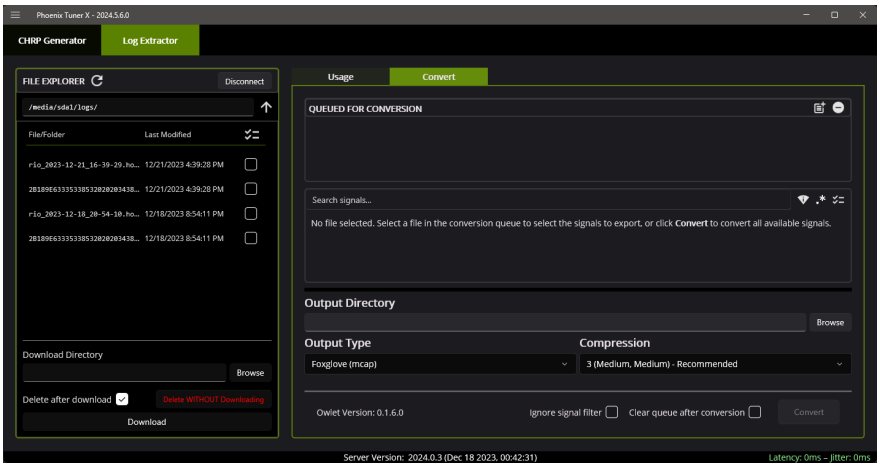

## 7.12 Tools

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

### 7.12.1 Extracting Signal Logs

> **Tip**
>
> Information on how to use the signal logger API can be found in the *corresponding API article*.

Tuner X 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.



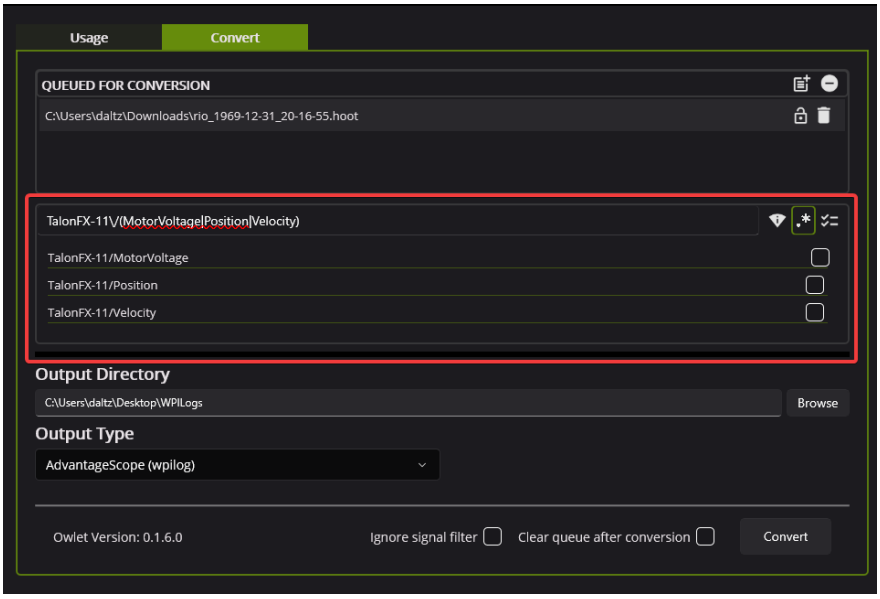

If a log file is reported as unlicensed, users can perform a *Deep Scan* of the log if they believe it should contain pro-licensed devices. By default, Tuner will only scan the first few megabytes of the log for pro-licensed devices to save time during scan and export.

### 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.

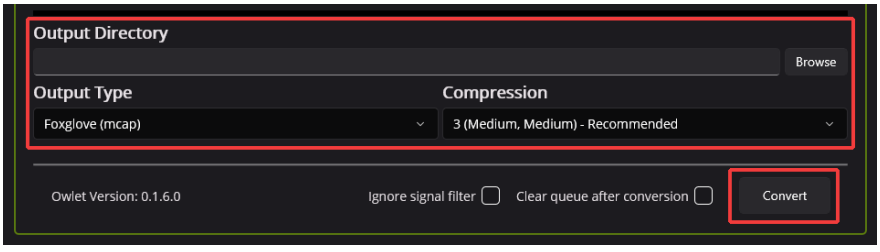
In the below picture, regex is used to select only the `MotorVoltage`, `Position`, and `Velocity` signals for `TalonFX-11`.




> **Important**
>
> If the `hoot` log does not contain any Pro-licensed devices, a *limited set of signals* may be exported for free.

## 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`

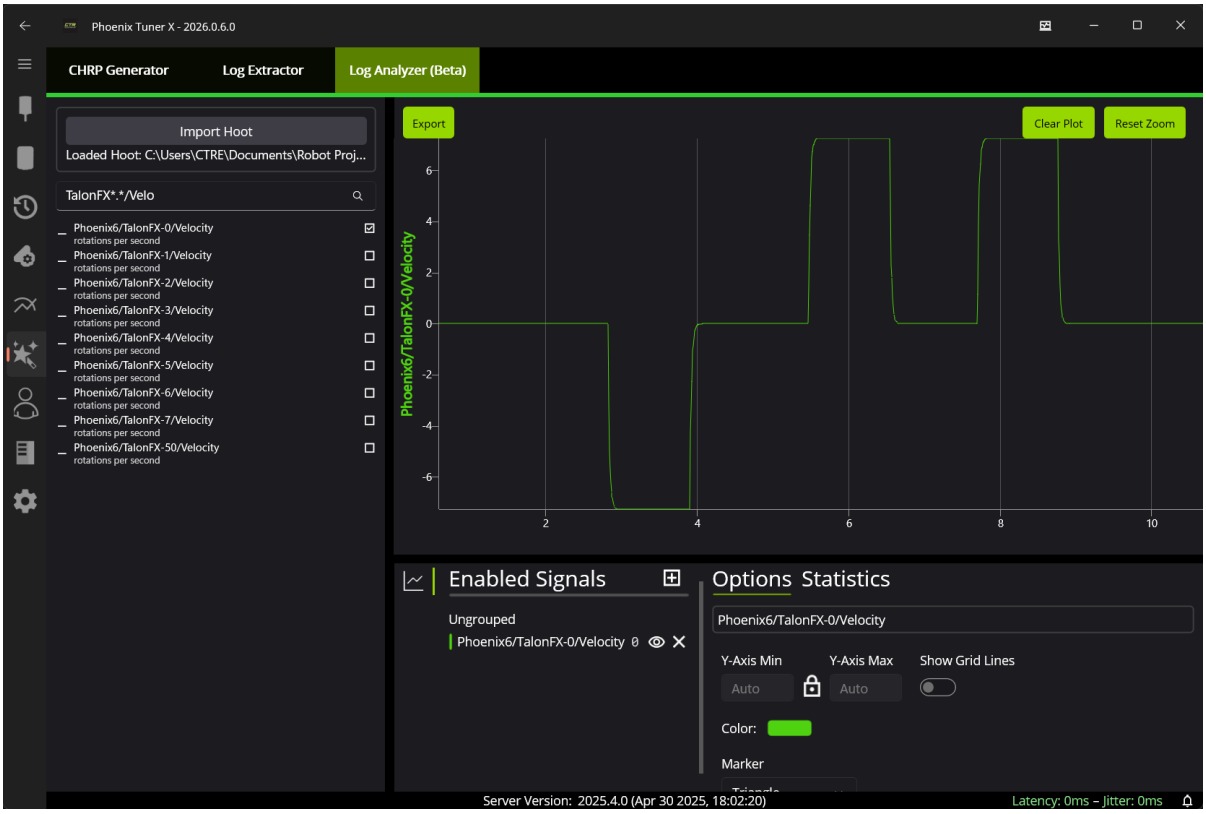
**Solution:** This may happen if your hoot file was generated using an old version of Phoenix. Update your Phoenix installation (by updating your vendordep in *Installing Phoenix 6*) and recreate your log file. If the log file recorded is critical, reach out to support@ctr-electronics.com.

#### 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.

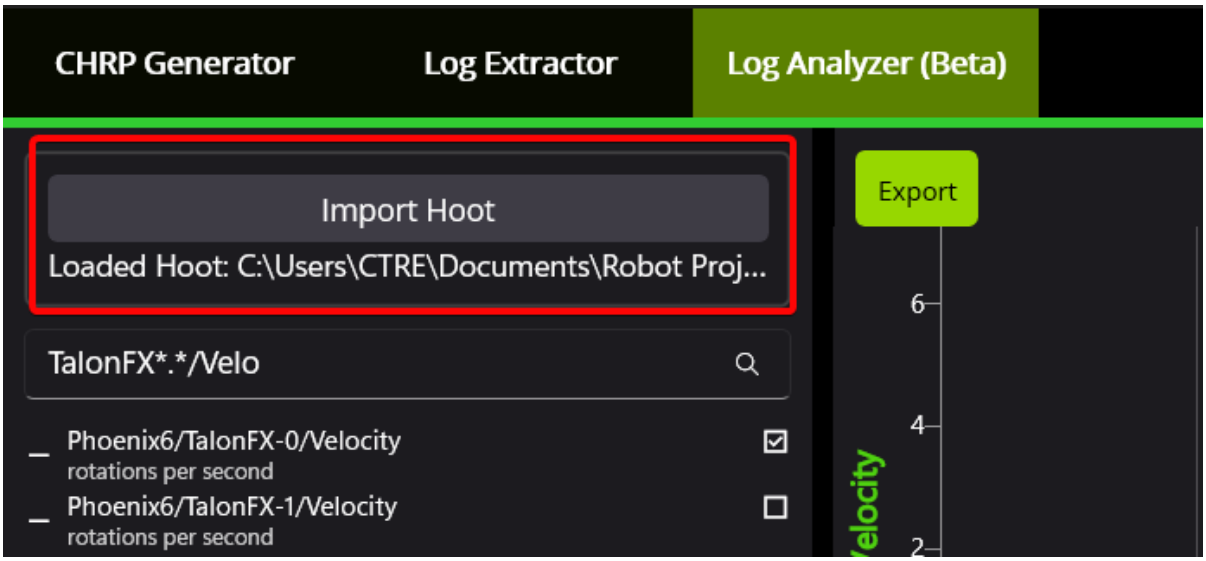
### 7.12.2 Log Analysis (Beta)

Hoot logs can be directly viewed and analyzed in Tuner using the built-in Log Analysis tool. This tool can be found by navigating to *Tools -> Log Analysis (Beta)* tab. Users can take advantage of our high-performance *plotter* to analyze their logs for trends.



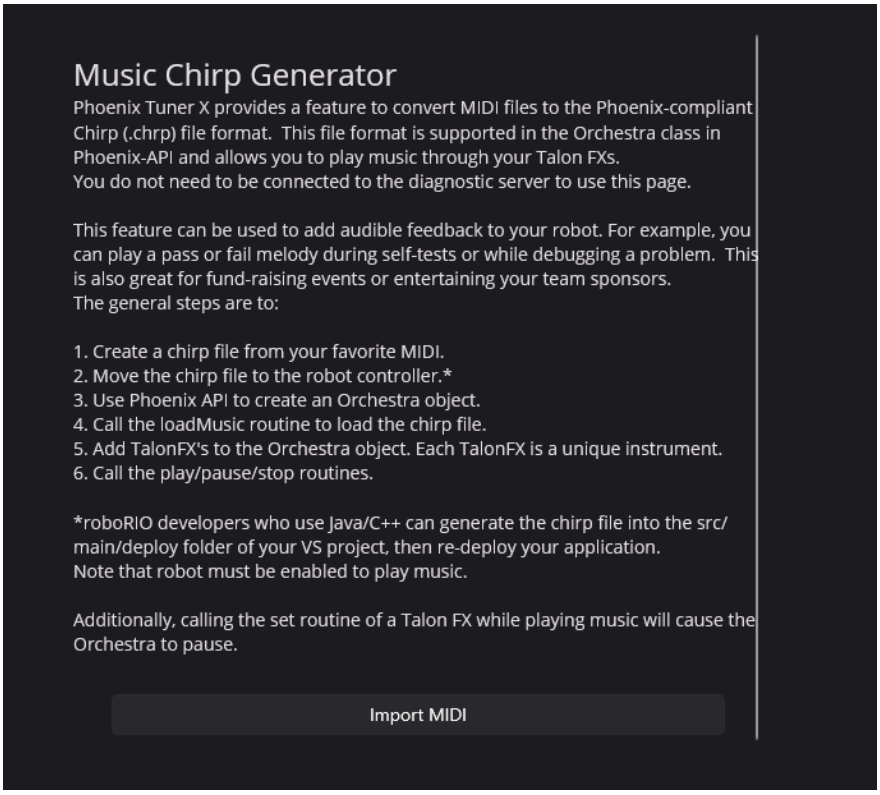

#### Loading Logs

A hoot log can be loaded by clicking on the *Import Hoot* button. Depending on the size of the log, it may take a long time for the log to be loaded.




### 7.12.3 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*.



#### `passerine` CLI

In addition to the Tuner X CHRP Converter, the `passerine` CLI tool can be used from a terminal to convert MIDI files to CHRP. `passerine` can be downloaded from the CLI Tools download page.

To view a list of available commands, run `passerine` either with no parameters or with `--help`.

```
$ ./passerine
passerine 1.0.0
Converts MIDI files (.mid/.midi) to CTR Electronics CHRP files (.chrp) for use in Phoenix Orchestra.
Copyright (C) CTR Electronics. All rights reserved.

Usage: passerine.exe --input-midi <INPUT_MIDI> --output-chrp <OUTPUT_CHRP>

Options:
  -i, --input-midi <INPUT_MIDI>    Input MIDI (.mid/.midi) file for conversion
  -o, --output-chrp <OUTPUT_CHRP>  Output CHRP (.chrp) file
  -h, --help                       Print help
  -V, --version                    Print version

For support and suggestions, contact support@ctr-electronics.com
```

## 7.13 Swerve Project Generator

**Important**

Full swerve project generation is only available for FRC users. However, non-FRC users can still generate the constants file.

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*.

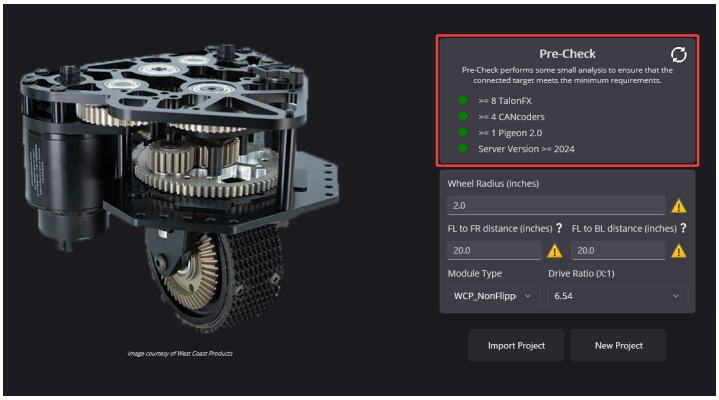
### 7.13.1 Swerve Requirements

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. Talon FX®, Talon FXS®, CANcoder®, CANdi®, Pigeon® 2.0).
  - 8 Talon FX® or Talon FXS® (4 steer, 4 drive)
  - 4 CANcoder® (4 steer) **OR** 4 Talon FXS® + PWM encoder **OR** 4 CANdi®
  - 1 Pigeon® 2.0
- Requires Phoenix 6 software released for the current release year of Tuner X.
  - Firmware version of devices should match the API year (26.X firmware corresponds to 2026 API/Tuner)
  - Ensure Tuner X shows server version (located at the bottom of Tuner X) that matches the Tuner X major version (2026 Tuner X requires 2026 server version)
- Temporary diagnostic server (or an existing robot project for that year of software) 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 inforced 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* and FusedCANdi improves the accuracy of module positions.

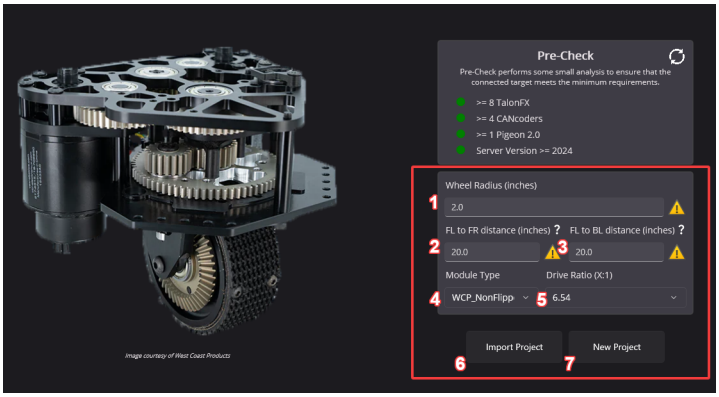
### Requirement Checklist

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 Talon FX®/Talon FXS®, 4 CANcoder®/CANdi®/Talon FXS® + PWM encoder, 1 Pigeon® 2.0) | |
| **REQUIRED:** Do all devices appear in Tuner X? | |
| **REQUIRED:** Are all devices on the same CAN bus? | |
| **REQUIRED:** Is all firmware up-to-date? (26.X) | |
| **REQUIRED:** Is the current year (2026) of diagnostics running? | |
| **Recommended:** Have devices been renamed? (e.g. "Talon FX (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.

### 7.13.2 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-left 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.

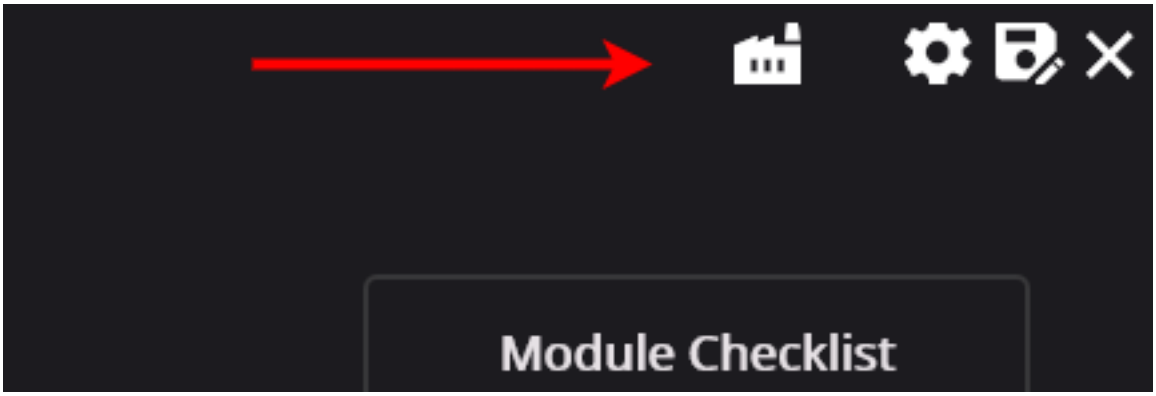
6. Import Project
    - Import an existing Tuner X swerve project save file.
7. New Project
    - Create a new project based on the settings configured.

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

### 7.13.3 Configuring Modules

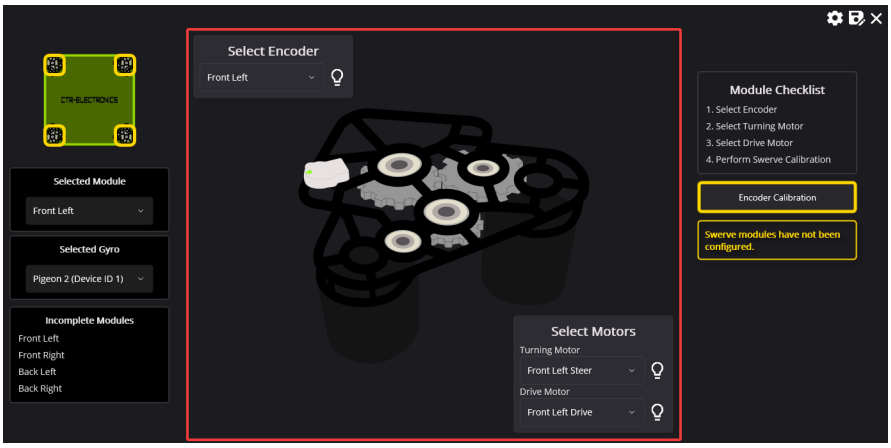
Module configuration is comprised of 4 steps.

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.

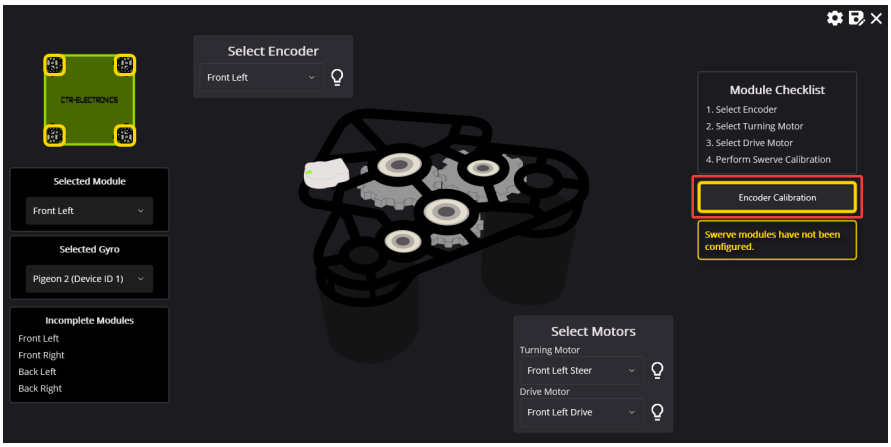
> **Tip**
>
> If a warning icon shows where the *Blink* icon is, this typically means this device has already been assigned to a module.




## 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.



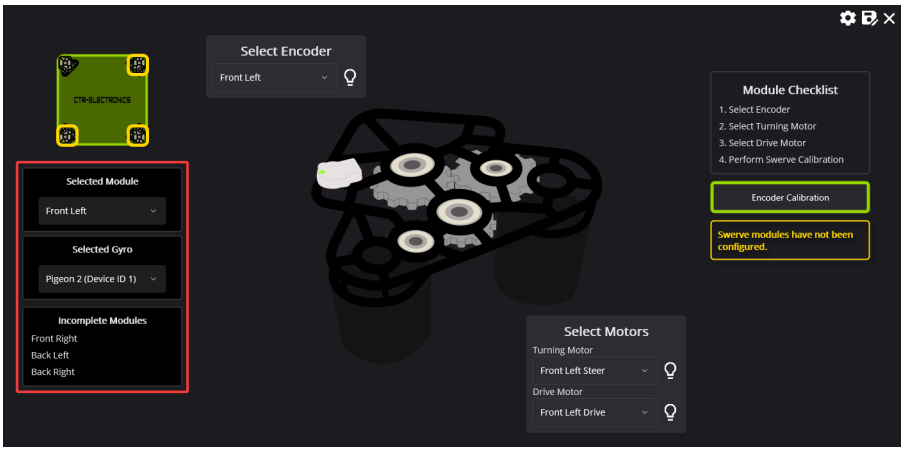

Once calibration has been completed, the module will be removed from the **Incomplete Modules** list on the left.

### 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.

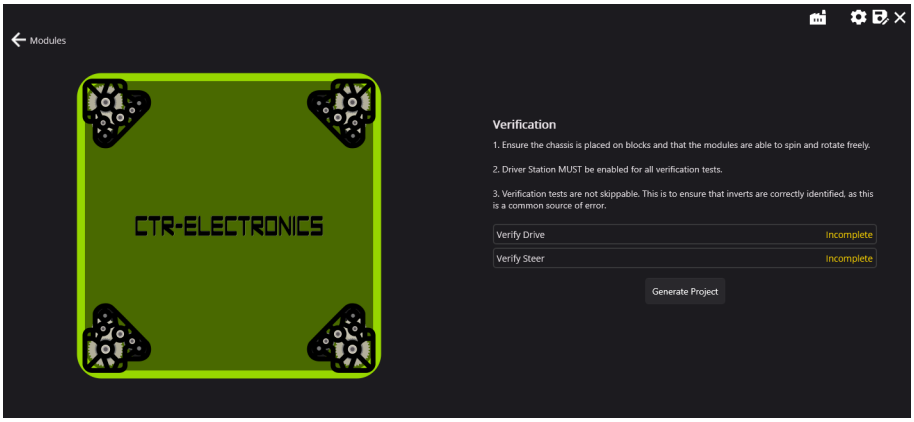
> **Warning**
>
> To ensure the drive and steer tests are accurate, the devices are factory defaulted. It's recommended that the user makes a *backup* of their configs if they are not applied in their robot program.

## 7.13.4 Validating the Drivetrain

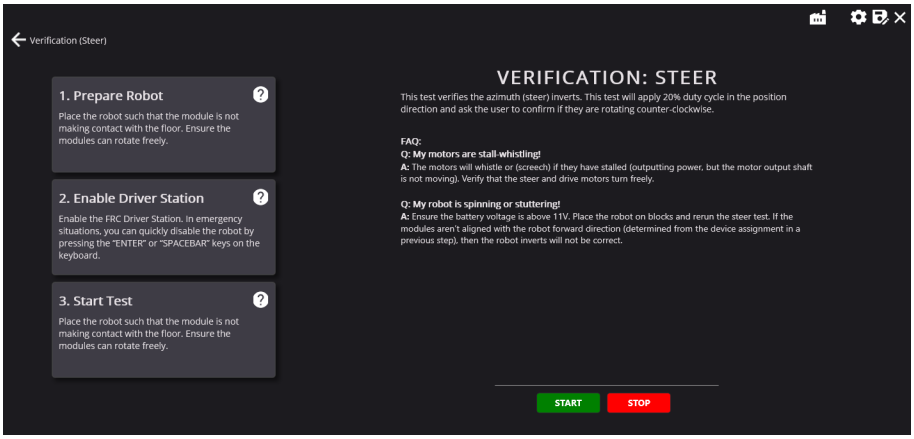
Tuner performs several validation procedures to confirm inverts, offsets and mechanical functionality of the drivetrain.

This test consists of two steps:

- Verify Steer
- Verify Drive



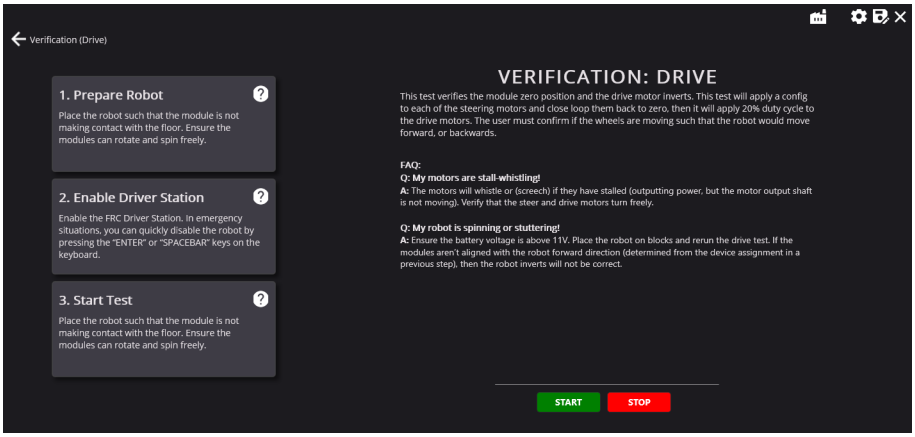

## Verify Steer




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 encoder 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!*.

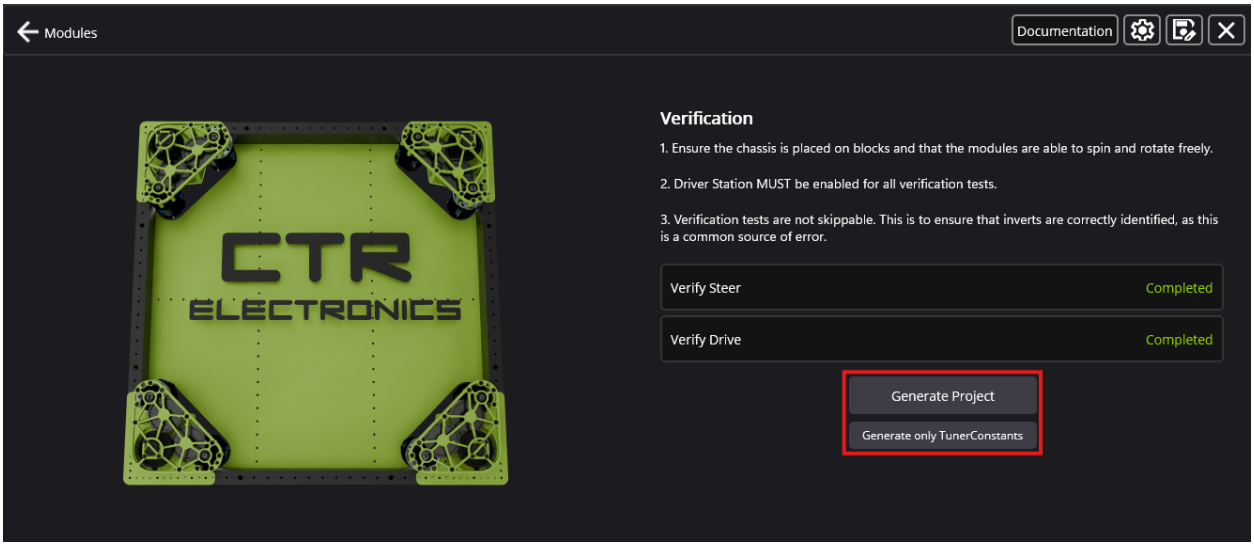
#### Q: My motors are stall-whistling

**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.

### 7.13.5 Generating the Project




The robot program can be generated by clicking the *Generate Project* button. This will open a prompt asking for the team number for the generated project, followed by the programming language.

Alternatively, the constants file alone can be regenerated for an existing project using the *Generate only TunerConstants* button.

> **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 or constants file needs to be regenerated for any reason.

### 7.13.6 Swerve Next Steps

Now that you have a working swerve drive, you may want to explore some of the following topics.

#### Setting Current Limits

Current limits are an important part of battery management. The following represents a short example of setting the stator/slip current limit and the supply current limit of the drive motors in a Tuner generated swerve project. To determine the correct current limits, and understand what they do and how they affect robot control, please refer to *Improving Performance with Current Limits*.

#### Java

```
52      // The stator current at which the wheels start to slip;
53      // This needs to be tuned to your individual robot
54      private static final Current kSlipCurrent = Amps.of(120);
55
```

(continues on next page)

(continued from previous page)

```
    // Initial configs for the drive and steer motors and the azimuth␣
↪encoder; these cannot be null.
    // Some configs will be overwritten; check the `with*InitialConfigs()`␣
↪API documentation.
    private static final TalonFXConfiguration driveInitialConfigs = new␣
↪TalonFXConfiguration()
        .withCurrentLimits(
            new CurrentLimitsConfigs()
                // Default supply current limit is 70 A, but it can be␣
↪lowered to avoid brownouts.
                // Supply current limits can be larger than the breaker␣
↪current rating.
                .withSupplyCurrentLimit(Amps.of(70))
                .withSupplyCurrentLimitEnable(true)
        );
```

### C++

```
    // The stator current at which the wheels start to slip;
    // This needs to be tuned to your individual robot
    static constexpr units::ampere_t kSlipCurrent = 120_A;

    // Initial configs for the drive and steer motors and the azimuth␣
↪encoder; these cannot be null.
    // Some configs will be overwritten; check the `With*InitialConfigs()`␣
↪API documentation.
    static constexpr configs::TalonFXConfiguration driveInitialConfigs =␣
↪configs::TalonFXConfiguration{}
        .WithCurrentLimits(
            configs::CurrentLimitsConfigs{}
                // Default supply current limit is 70 A, but it can be␣
↪lowered to avoid brownouts.
                // Supply current limits can be larger than the breaker␣
↪current rating.
                .WithSupplyCurrentLimit(70_A)
                .WithSupplyCurrentLimitEnable(true)
        );
```

### Python

```
    # The stator current at which the wheels start to slip;
    # This needs to be tuned to your individual robot
    _slip_current: units.ampere = 120.0

    # Initial configs for the drive and steer motors and the azimuth encoder;␣
↪these cannot be null.
    # Some configs will be overwritten; check the `with_*_initial_configs()`␣
↪API documentation.
    _drive_initial_configs = configs.TalonFXConfiguration().
↪with_current_limits(
```

(continues on next page)

(continued from previous page)

```
        configs.CurrentLimitsConfigs()
        # Default supply current limit is 70 A, but it can be lowered to␣
↪avoid brownouts.
        # Supply current limits can be larger than the breaker current rating.
        .with_supply_current_limit(70.0)
        .with_supply_current_limit_enable(True)
    )
```

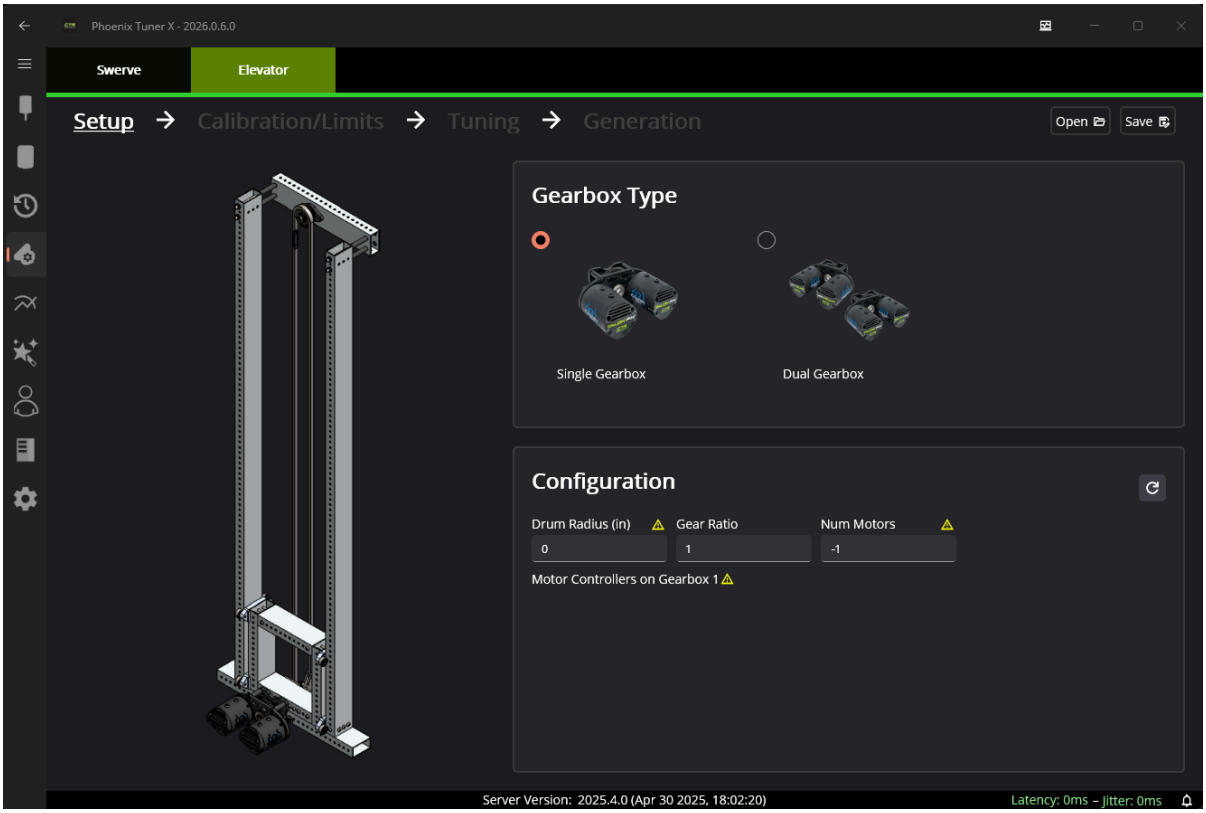
## 7.14 Tuner Elevator Generator

**Important**

The generated Elevator subsystem assumes WPILib command based, but can trivially be adopted for non-FRC by removing the WPILib references.

Under the *Mechanisms* page in Tuner X is the Elevator Generator. This utility guides the user through determining the necessary constants and configurations for a working Elevator subsystem.

### 7.14.1 Setup

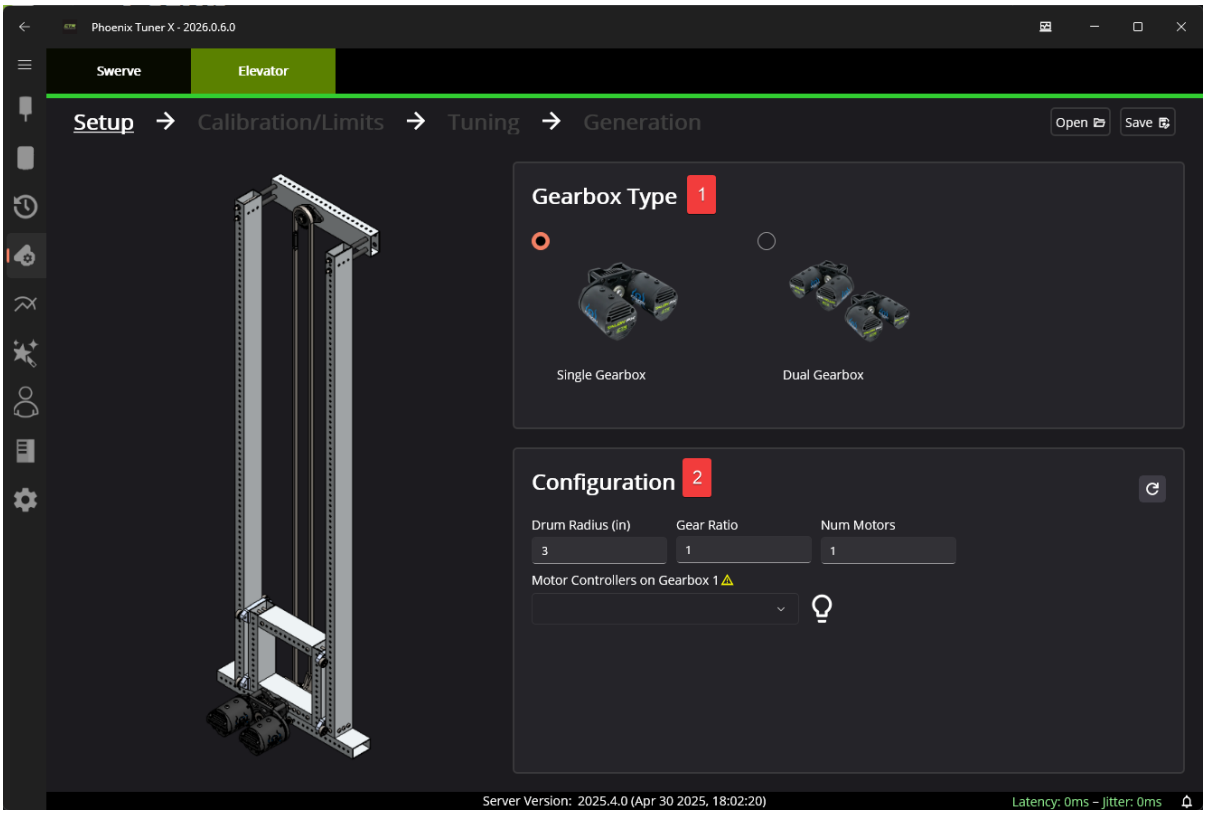


### Prerequisites

The elevator generator and generated project make a few assumptions. To determine if the elevator generator is the best fit for your mechanism, consult the following checklist.

- The elevator is a single or two gearbox mechanism.
- All gearboxes on the elevator consist of the same gearing.
- All gearboxes on the elevator have the same number of motors.
- If using two gearboxes, ensure that the orientation of the gearbox's are identical.
- All motors on the elevator are identical.

### Initial Configuration




Setup is done with two steps.

### 1. Choosing your Gearbox

The supported gearbox types are `Single` and `Dual`.

### 2. Configuration

A typical elevator is composed of a number of motors, driving a gearbox which spins a spool that drives the elevator up or down. The generated Elevator subsystem will automatically handle conversion between raw mechanism rotations to linear units (inches, meters, feet, etc).

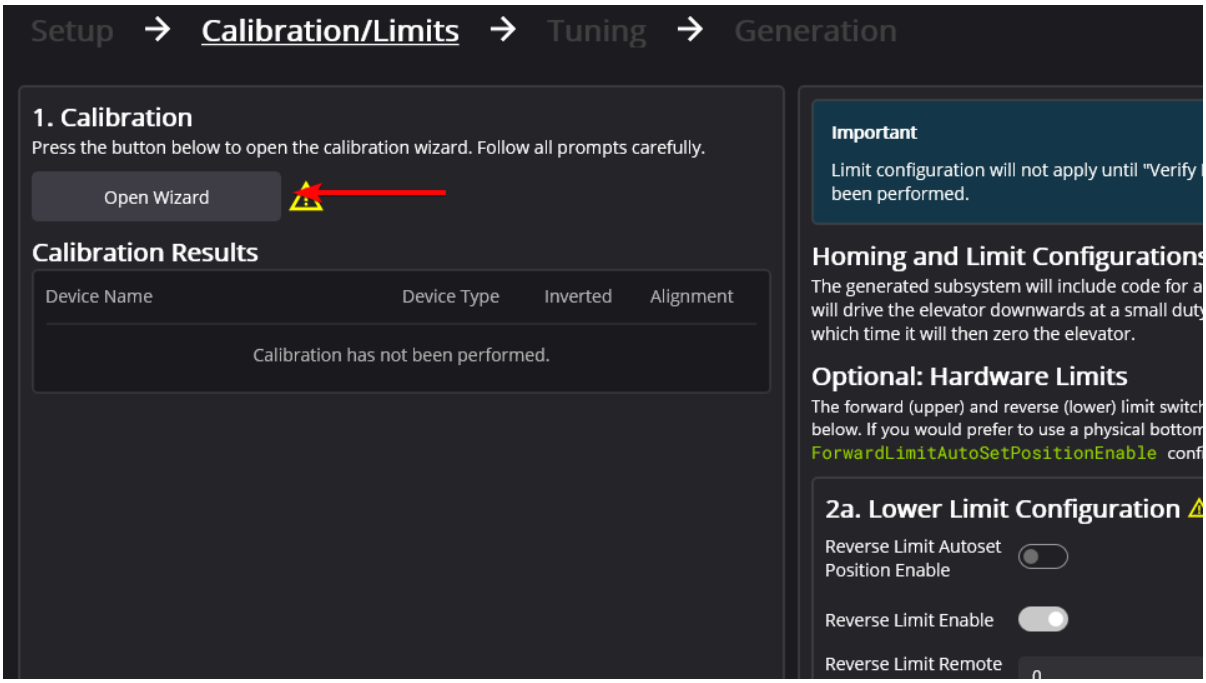
To handle this scenario, certain constants cannot be automatically determined. Users will need to input the *Drum Radius (in)*, *Gear Ratio*, and *Num Motors*.

Once this has been done, a list of motor controllers will be populated per gearbox.

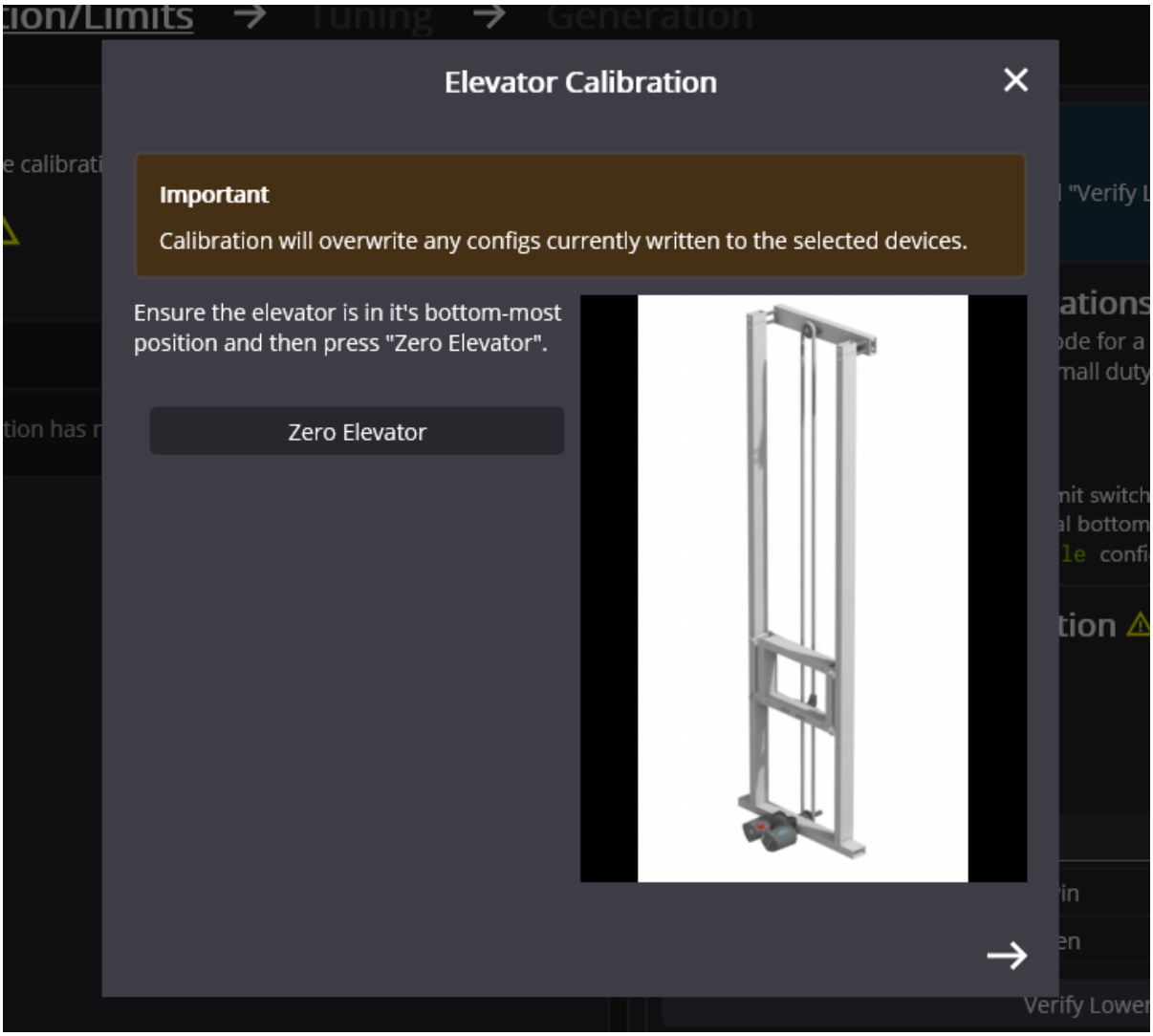
### 7.14.2 Calibration and Limits

Tuner will have the user perform a calibration routine, during which the user will bring the elevator to it's lowest position, and then manually raise the elevator to it's top-most position. This routine will automatically determine the bounds of the elevator and motor inverts.

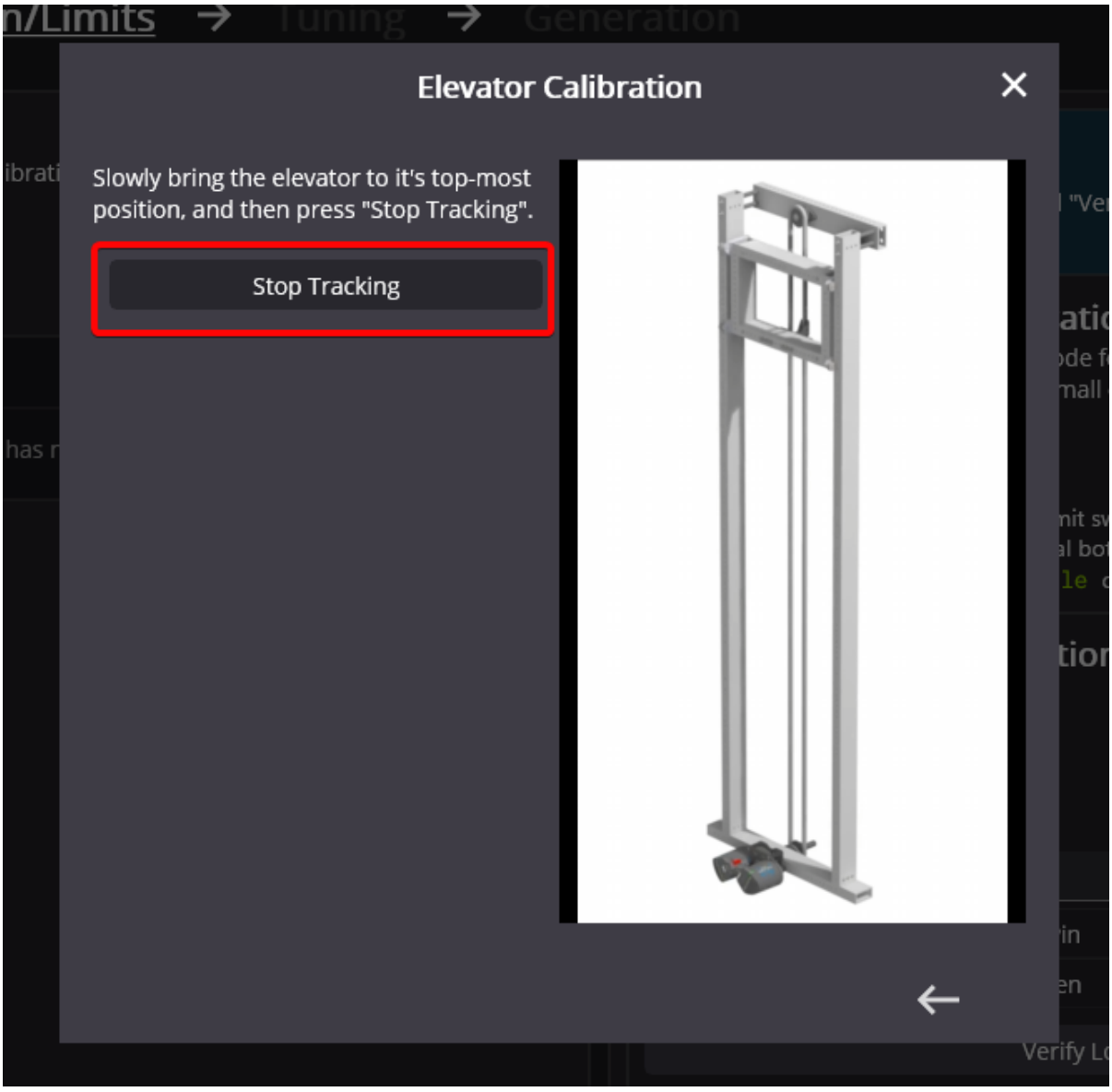
To begin, select *Open Wizard*.



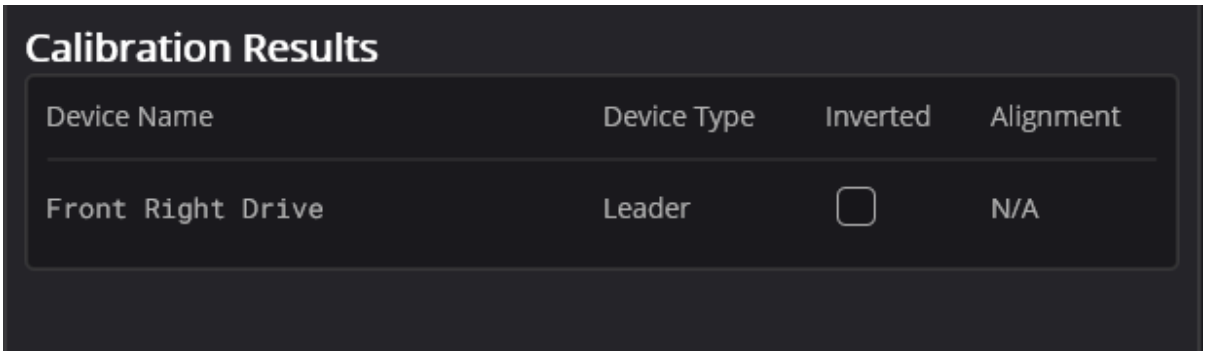

This will open the Elevator Calibation popup. Manually bring the elevator to it's bottom-most position. This will act as it's zero. Once this is done, press *Zero Elevator*.




Press the arrow on the bottom-right of the popup to navigate to the next step in the wizard. Bring the elevator to it's top-most position. Once this is done, press *Stop Tracking*.

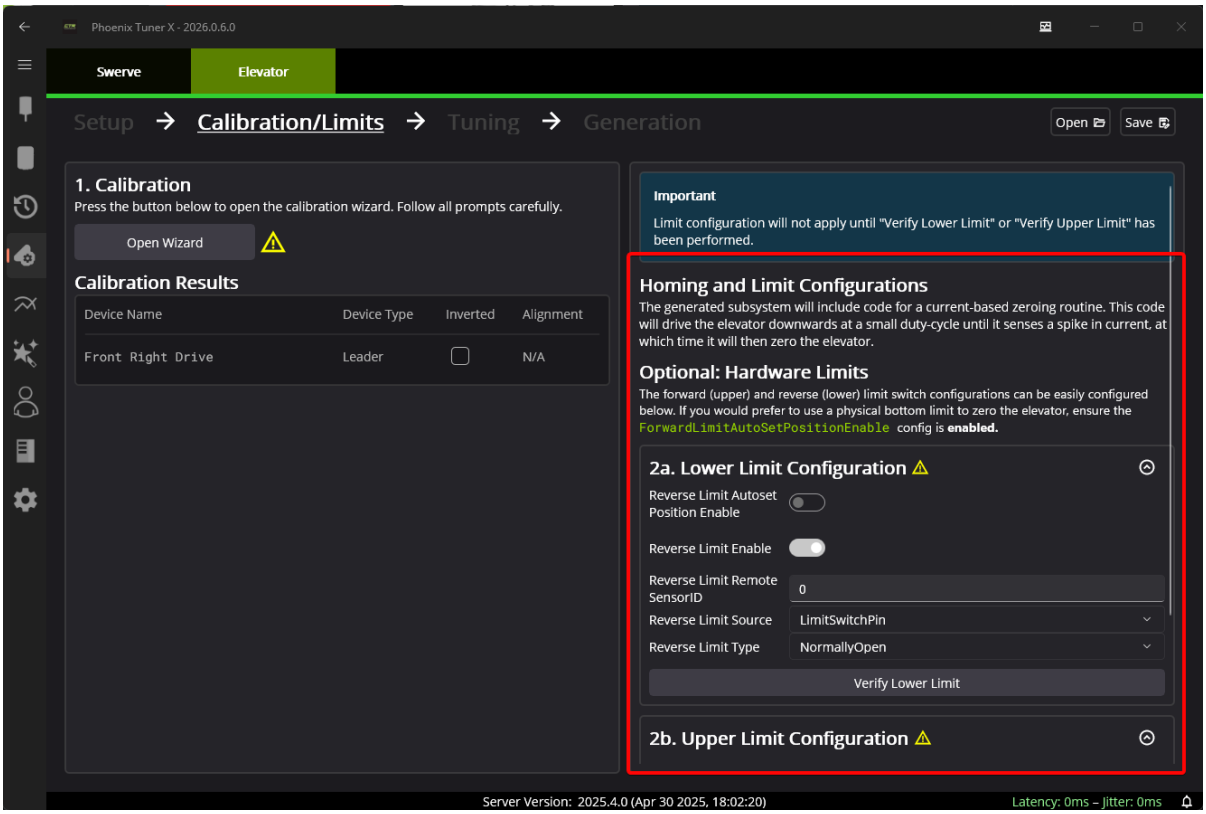



Go ahead and exit the popup. The *Calibration Results* will be populated with it's results.



### Homing Limits

The generated Elevator subsystem includes a function for performing a **current-based** homing routine to zero the elevator. However, the calibration and limits interface provides the ability to configure hardware or remote limits.

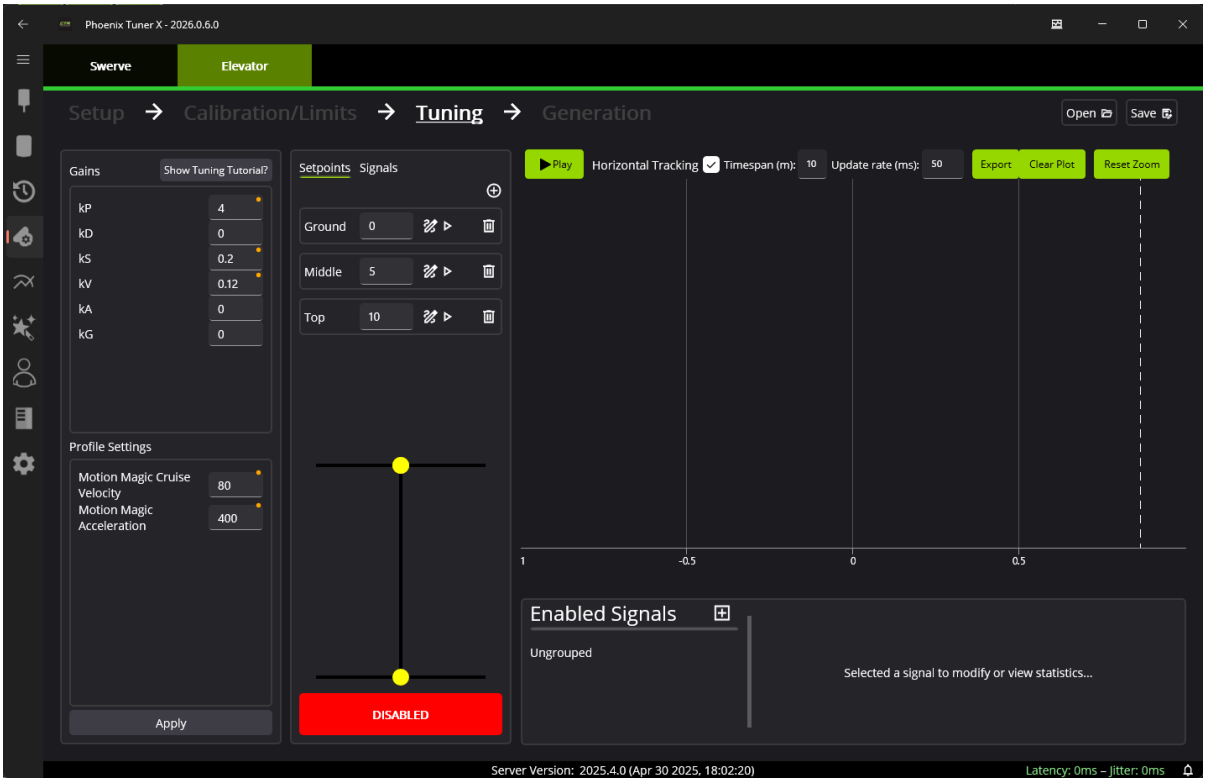



## 7.14.3 Tuning your Elevator

In the third step of the elevator wizard, the user is guided through **creating setpoints** and **calculating closed-loop** gains.

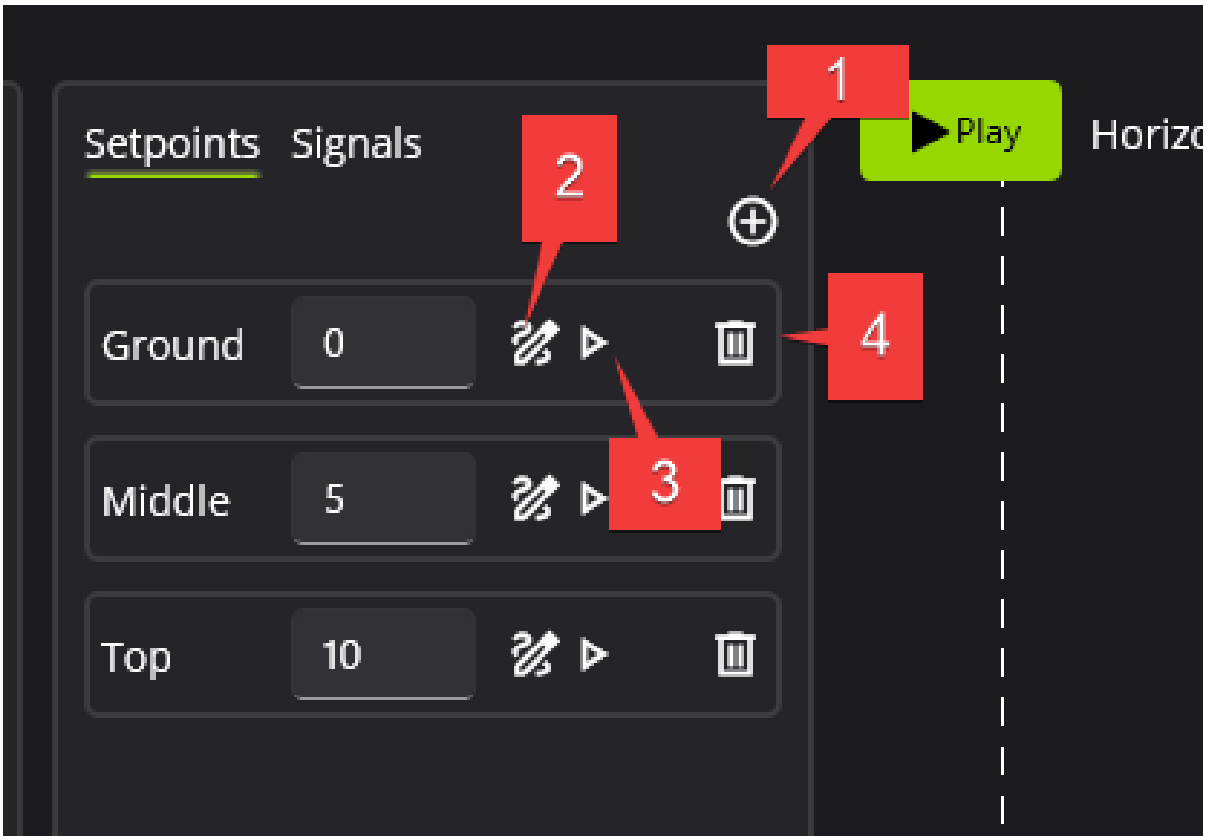
**Tip**

Check out *Basic PID and Profiling* for information on how to tune a closed-loop position system.




### Elevator Setpoints

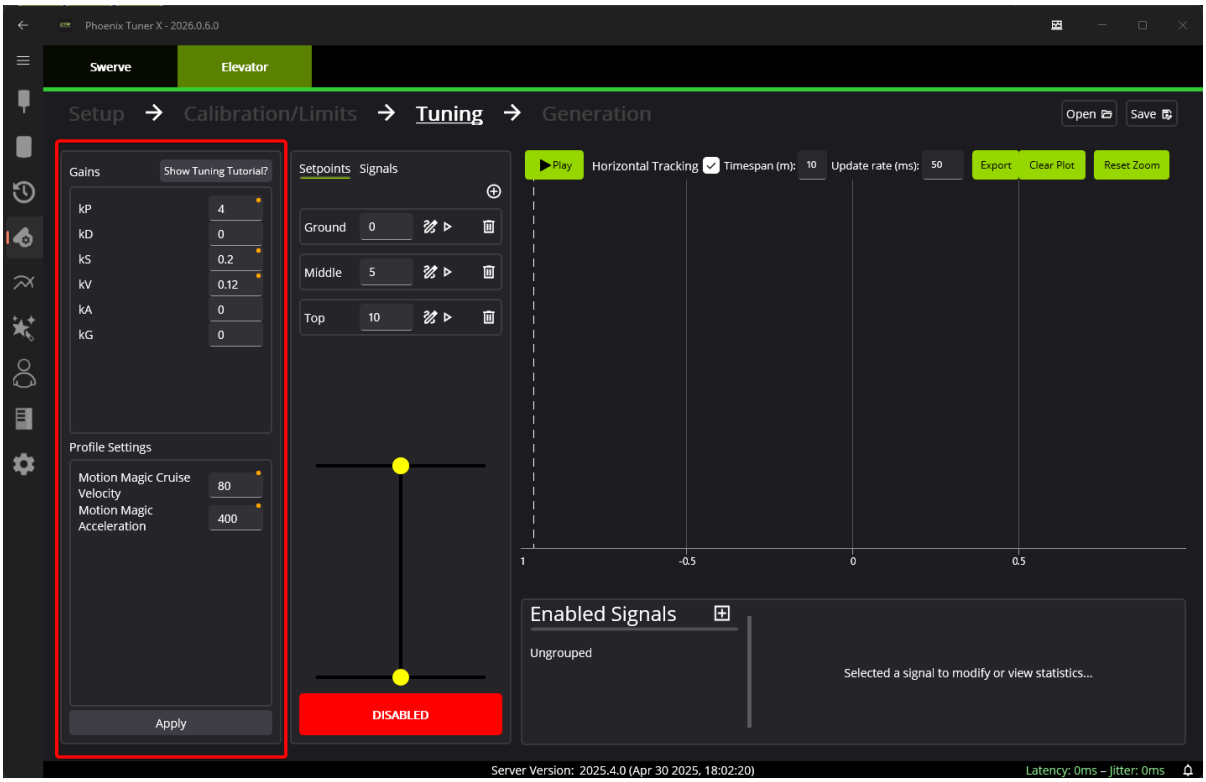
Setpoints are configured in the *Setpoints* tab in the second column.




1. Add a new setpoint
2. Rename a setpoint
   - Ensure setpoint names are unique, or there will be compile errors when you generate your Elevator.
3. Run setpoint
   - The robot must be **ENABLED**, or nothing will happen.
4. Delete setpoint

### Closed-loop Gains

Gains can be configured in the first column. While default gains have been calculated, it is **highly recommended** to just use this as a starting point. The elevator should be tuned in it's final configuration, with any load that it may need to bear (holding a game piece).

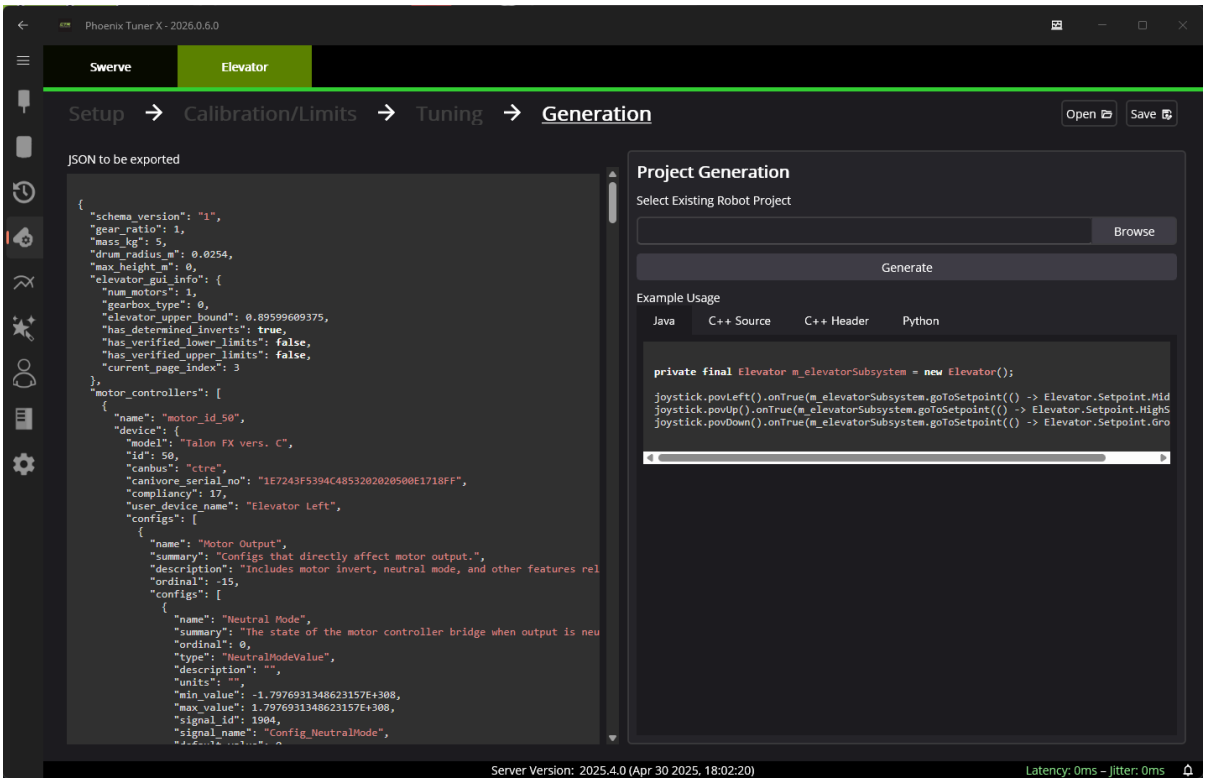



The control request used to command the elevator is `MotionMagicVoltage` (Java, C++, Python), which allows the user to directly control velocity and acceleration for smooth travel.

### 7.14.4 Generation

The subsystem is generated directly into an **existing robot project**. Select *Browser* and navigate to the **root** of a robot project. Then, press *Generate*.

Open the robot project in WPILib VS Code.


Phoenix Tuner X - 2026.0.6.0
Swerve
Elevator
Setup → Calibration/Limits → Tuning → Generation
Open
Save
JSON to be exported
{
"schema_version": "1",
"gear_ratio": 1,
"mass_kg": 5,
"drum_radius_m": 0.0254,
"max_height_m": 0,
"elevator_gui_info": {
"num_motors": 1,
"gearbox_type": 0,
"elevator_upper_bound": 0.89599609375,
"has_determined_inverts": true,
"has_verified_lower_limits": false,
"has_verified_upper_limits": false,
"current_page_index": 3
},
"motor_controllers": [
{
"name": "motor_id_50",
"device": {
"model": "Talon FX vers. C",
"id": 50,
"canbus": "ctre",
"canivore_serial_no": "1E7243F5394C4853202020500E1718FF",
"compliancy": 17,
"user_device_name": "Elevator Left",
"configs": [
{
"name": "Motor Output",
"summary": "Configs that directly affect motor output.",
"description": "Includes motor invert, neutral mode, and other features rel
"ordinal": -15,
"configs": [
{
"name": "Neutral Mode",
"summary": "The state of the motor controller bridge when output is neu
"ordinal": 0,
"type": "NeutralModeValue",
"description": "",
"units": "",
"min_value": -1.7976931348623157E+308,
"max_value": 1.7976931348623157E+308,
"signal_id": 1904,
"signal_name": "Config_NeutralMode",
Project Generation
Select Existing Robot Project
Browse
Generate
Example Usage
Java
C++ Source
C++ Header
Python
private final Elevator m_elevatorSubsystem = new Elevator();
joystick.povLeft().onTrue(m_elevatorSubsystem.goToSetpoint(() -> Elevator.Setpoint.Mid
joystick.povUp().onTrue(m_elevatorSubsystem.goToSetpoint(() -> Elevator.Setpoint.HighS
joystick.povDown().onTrue(m_elevatorSubsystem.goToSetpoint(() -> Elevator.Setpoint.Gro
Server Version: 2025.4.0 (Apr 30 2025, 18:02:20)
Latency: 0ms – Jitter: 0ms

# 8 What is Licensing

All supported Phoenix 6 devices can freely use the Phoenix 6 API. Licensing a device can enable additional features that can enhance performance or user experience. A breakdown of the features offered by licensing your device is available at *Feature Breakdown*.

Licenses will enable Phoenix Pro features on hardware when the device firmware and API versions match the license year. For example, a 2023 license enables Pro when used with 23.X firmware and 23.X Phoenix 6 API. A 2024 license would enable Pro features when used with 24.X firmware and 24.X API.

**Note**

Using Phoenix 6 simulation with Pro features does not require licensing.

## 8.1 Types of Licenses

Season Pass (FRC Teams Only) A single purchase that allows FRC Teams to license the team's devices for Phoenix Pro.

Season Pass generated licenses are equivalent to Single Device licenses but are tied to the team number assigned to the Season Pass after purchase.

*Season Pass Licensing* Single Device A single device license can be activated and installed individually per device. Each purchase licenses exactly one compatible CAN bus device to use Phoenix Pro features.

This may be preferred for systems with a small number of devices that need to utilize Pro features, or when used in a benchtop application.

*Device Licensing* CANivore® Bus This license is activated and installed onto the CANivore®, and enables Phoenix Pro features for every CAN-connected device. This means that every compatible device that is attached to a licensed CANivore® via CAN bus will be able to use Phoenix Pro features.

This license type is advantageous if there are a large number of devices and eliminates the need to license additional devices in the event of a replacement.

*Device Licensing*

# 9 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.

All *Phoenix 6 supported devices* support device licensing.

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.

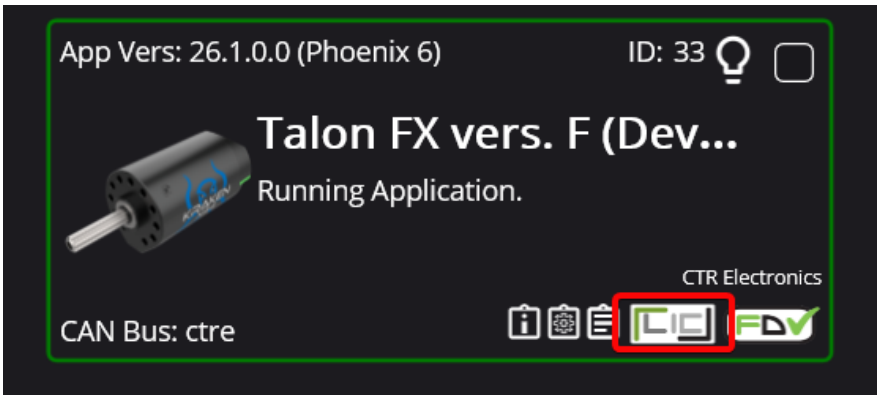
## 9.1 Purchasing a License

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.

## 9.2 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.

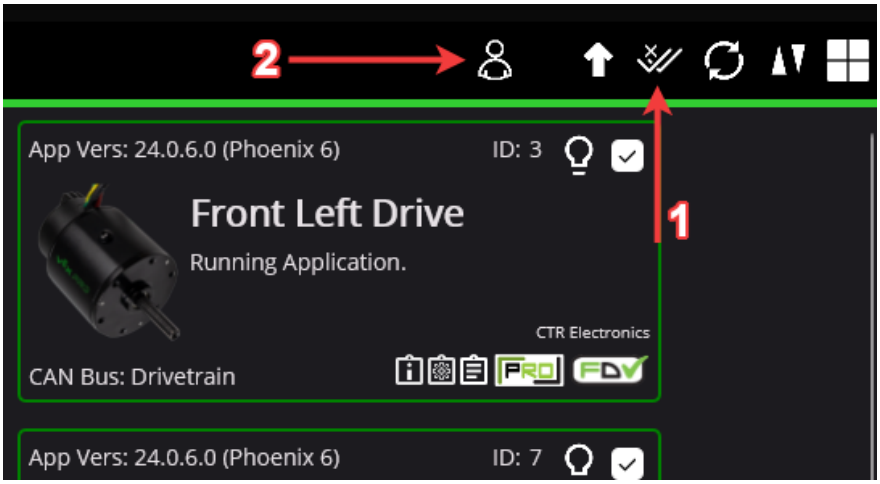
> **Warning**
>
> Users should be aware that license activation is permanent and irreversible

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.

### 9.2.1 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.

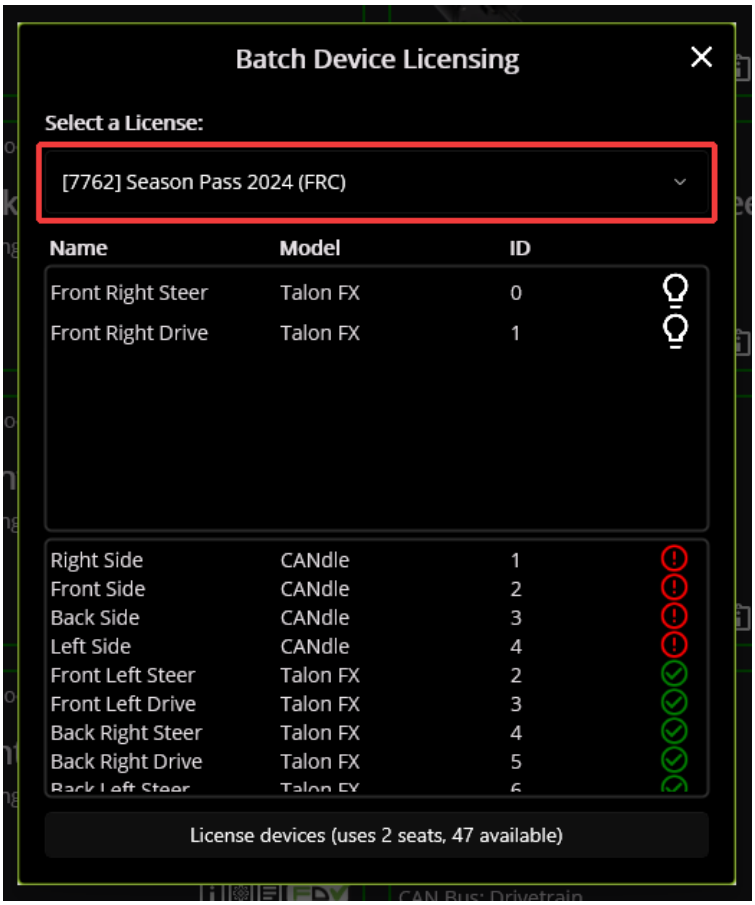
> **Tip**
>
> Selecting a device using their checkbox and clicking the checkmark in the top right will select all devices of the same models




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.

### 9.2.2 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.

## 9.3 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.

## 9.4 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?

# 10 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*.

## 10.1 When is a Season Pass Applicable?

Season Pass licenses enable Pro features when the licensed device is:

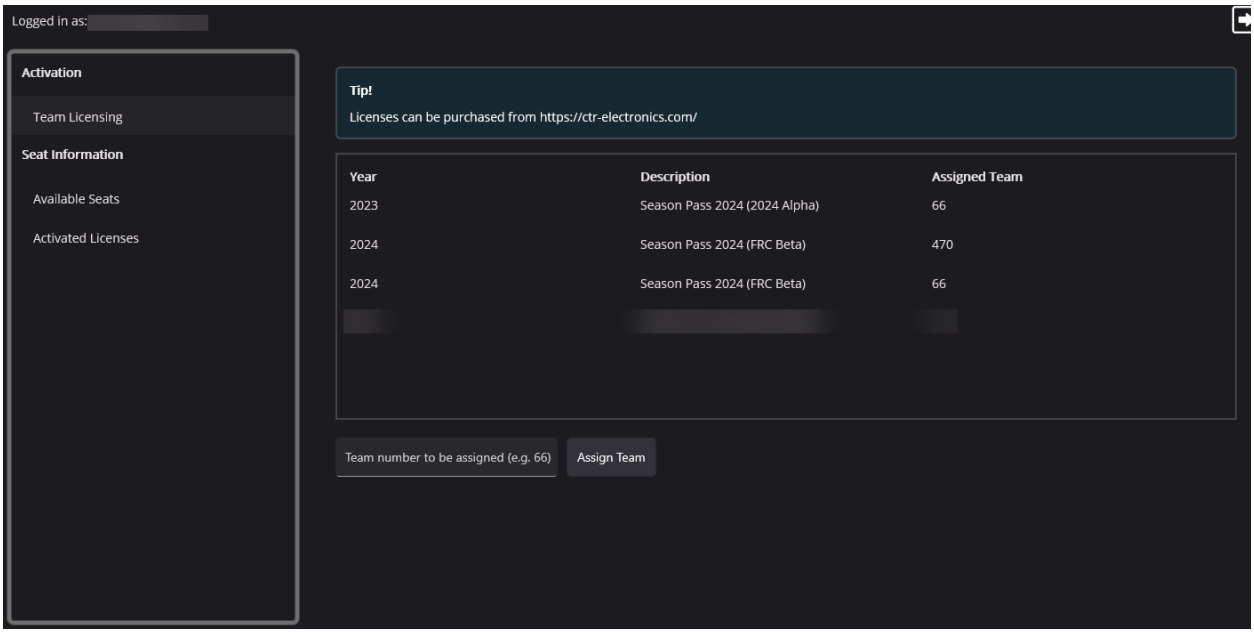
- Used on a roboRIO configured for the team number assigned to the Season Pass
  - Season Pass licensing will work on the roboRIO regardless of whether the device is on the native CAN bus or a CANivore® bus.
- Used for Hardware Attached Simulation (a CANivore® connected to PC) with a WPILib robot project
- Used on a roboRIO configured for one of the reserved “Off-Season Demo” team numbers (currently team numbers 9970 - 9999)

## 10.2 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.




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.

## 10.3 How many devices can I activate?

A season pass contains 100 individual device licenses. In the event that a team needs more licenses, contact us at support@ctr-electronics.com.

# 11

# CANcoder®

The CTR Electronics 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.

https://store.ctr-electronics.com/cancoder/ Hardware User Manual Wiring and mount instructions in PDF format.

https://store.ctr-electronics.com/content/user-manual/CANCoder%20User's%20Guide.pdf

## 11.1 Status Light Reference

**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

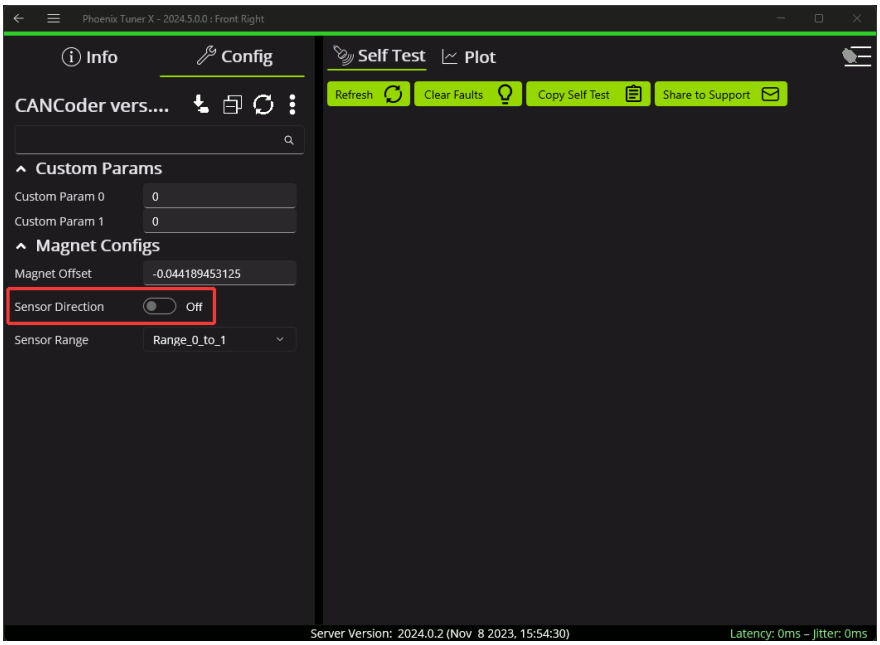
> **Note**
>
> Users wishing to test magnet placement must wait 8 seconds after boot for the LEDs to blink the magnet placement status.

## 11.2 Magnet Placement

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

## 11.3 Verifying Sensor Direction

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



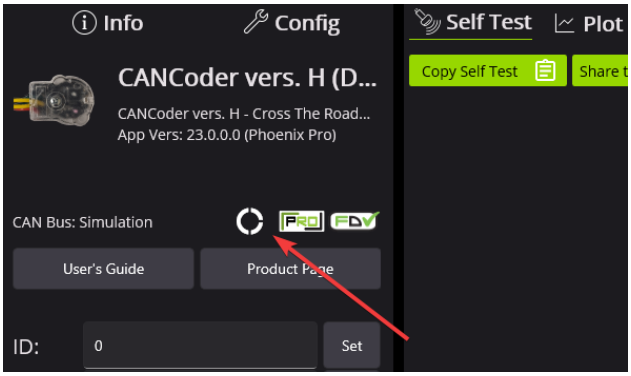

## 11.4 Zeroing the CANcoder®

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

Additionally, the CANcoder® 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.



# 12 CANdi®

The CTR Electronics CANdi® branded device seamlessly integrates digital signals into existing CAN bus networks, simplifying wiring and allowing multiple devices to share and utilize valuable input data. CANdi® enables CAN interopability with sensors such as: PWM encoders, Quadrature encoders, beam break sensors, and limit switches.

Store Page CAD and purchase instructions.

https://store.ctr-electronics.com/products/candi

## 12.1 Status Light Reference

**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

# 13 CANdle®

The CTR Electronics CANdle® makes it easy to control individually addressable LEDs over CAN. Combined with being a 5V high-efficiency DC voltage regulator, the CANdle® is a versatile addition to any robot.

Store Page  CAD and purchase instructions.

https://store.ctr-electronics.com/products/candle

## 13.1 Status Light Reference

**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

[1] CANdle® can be directly controlled using a WS2812B-compliant pulse train from other devices, such as an Arduino. See the User's Guide for more information.

# 14 CANrange®

The CTR Electronics CANrange® is a CAN-enabled Time-of-Flight distance measurement sensor. This product uses laser measurements to calculate precise distance to a surface parallel to the sensor. Users can also configure the CANrange® to act as a limit switch or beam break sensor in the *Device Configs*.

Store Page CAD and purchase instructions.

https://store.ctr-electronics.com/products/canrange Hardware User Manual https://ctre.download/files/user-manual/CANrange%20User's%20Guide.pdf

## 14.1 Tuning CANrange®

There are a few aspects of tuning a CANrange® to correctly detect the presence of game pieces, robot components, or field elements.

### 14.1.1 Signal Strength Tuning

The first step of detection tuning is to verify that the distance measurement is a valid measurement. This is accomplished by looking at the signal strength, with higher values indicating more confident distance measurements. By default, the minimum signal strength is 2500 to classify a distance measurement as "valid" for the purpose of the IsDetected signal.

There are two primary scenarios that affect the signal strength for the CANrange®:

1. **CANrange® looks into open air when nothing is in front of it** - This results in low signal strength under "not detected" and high signal strength with some distance when "detected"
2. **CANrange® looks at something further away when nothing is in front of it** - This typically results in high signal strength regardless of wither something is in front of it, but the distance shortens when something is in front of it.

The first scenario requires Signal Strength tuning to identify the difference between open air and objects in front of the sensor. The process for tuning is to put an object in front of the CANrange®, as it would happen on the final mechanism, and measure the signal strength. Then, remove that object and measure the signal strength.

Do this in as many configurations as is practical for the system and identify a signal strength threshold that will allow all detections when the object is present while disallowing as many

of the detections when the object is not present as possible. Typically, in an open-air system, this alone is sufficient to classify an object as detected or not.

### 14.1.2 Distance Tuning

Distance tuning is essentially the same process as signal strength tuning: place an object in front of the CANrange®, take a measurement, remove the object, take a measurement, and repeat. Just like in signal strength tuning, distance threshold should be set such that it splits the two scenarios as best as possible.

In open air, the distance measurement may be unreliable when the object is not detected, in which case the signal strength should filter it out and reject it. For these cases, simply ensure the distance threshold is above the distance when an object is present.

### 14.1.3 Examples

#### Open Air

The following is an example of a CANrange® being used to detect an object with it otherwise pointing to open air.




In the image, the distance measurement is unreliable at signal strengths below 2000, and very reliable when the strength is above 2000.

Given that the signal strength alone is sufficient to detect game piece detection, a large distance threshold can be used without much concern. In this case, the distance threshold is 0.4 meters, even though the measured distance is only 0.11 meters.

This results in the following IsDetected result, which aligns nicely with the presence of the object:




## Always-Valid Measurements

The following is an example of a CANrange® being used to detect an object when it is also pointed at a backing material when the object is not present.




In this scenario, note that the signal strength is always high, typically larger than 2000. However, the distance measurement is always valid, so object detection must instead rely on the distance thresholding.

In this case, a signal strength of 2000 is appropriate, and a distance threshold of 0.15 meters would cleanly identify when an object is detected. This results in the following plot:




This, too, aligns nicely with when an object is placed in front of the CANrange®.

## Hysteresis

In some cases, the object being detected results in a noisy distance measurement, and the CANrange® must avoid accidentally declaring an object as no longer detected when it is still present and vice versa. An increased hysteresis may be appropriate in this circumstance, as it will require the distance measurement to move beyond the threshold to account for this noise.

Below is an example of such a case.



In this scenario, the threshold should be approximately 0.115; however, the distance measurement sometimes dips below that threshold when the object is not present and above that threshold when it is. In this case, the hysteresis can be set to approximately 0.01 to account for the dips above and below, resulting in the following:




In this scenario, the hysteresis correctly prevents the IsDetected signal from blipping while the distance measured crosses the threshold.

## 14.2 Status Light Reference

**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

The rate at which the LED is blinking can be used as a rough indicator of measured distance. For example, the below LED shows that the detected distance is close to the CANrange®.

# 15 Pigeon® 2.0

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

With no on-boot calibration or temperature calibration required and dramatic improvement to drift, the Pigeon® 2.0 is the easiest IMU to use yet.

## 15.1 Pigeon® 2.0 Troubleshooting

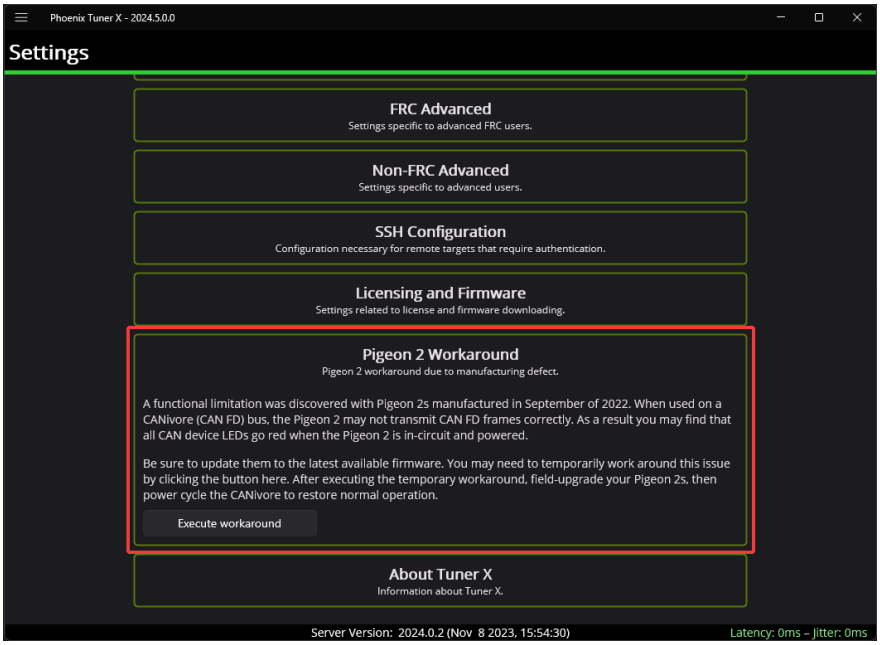
A functional limitation was discovered in Pigeon® 2.0 devices manufactured in September of 2022. When used on a CANivore® (CAN FD) Bus, the Pigeon® 2.0 may not transmit CAN FD frames correctly. As a result, you may find that all CAN device LEDs go red when the Pigeon® 2.0 is in-circuit and powered.

A firmware fix has been published, to update the firmware of an affected Pigeon® 2.0, one of the below options can be used.

### 15.1.1 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 CANivore® devices will enter a special mode that allows them to see the offending Pigeon® 2.0 devices. 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®.

### 15.1.2 Option 2: Connect to the roboRIO Bus

Connect the Pigeon® 2.0 to the roboRIO CAN Bus and *field-upgrade* the firmware version.

> **Note**
>
> We recommend power cycling the Pigeon® 2.0 after moving CAN bus leads from CANivore® to roboRIO CAN bus to ensure a clean transition.

Store Page CAD and purchase instructions.

https://store.ctr-electronics.com/pigeon-2/ Hardware User Manual Wiring and mount instructions in PDF format.

https://store.ctr-electronics.com/content/user-manual/Pigeon2%20User's%20Guide.pdf

## 15.2 Status Light Reference

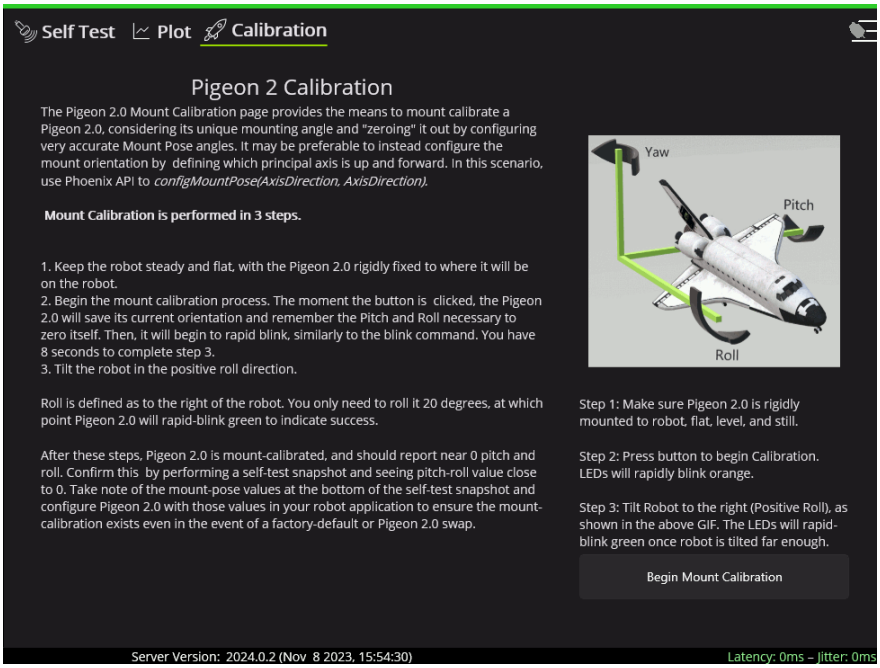



**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

## 15.3 Mount Calibration

It is 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.



# 16 Talon FX®

### 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 and purchase instructions.

https://store.ctr-electronics.com/products/kraken-x60/ Hardware User Manual Wiring and mount instructions available on the WestCoast Products documentation site.

https://docs.wcproducts.com/kraken-x60/

### Kraken X44

The Kraken X44 powered by Talon FX® is a compact 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 X44 BLDC motor while providing best-in-class motion control.

Store Page CAD and purchase instructions.

https://store.ctr-electronics.com/products/kraken-x44/ Hardware User Manual Wiring and mount instructions available on the WestCoast Products documentation site.

https://docs.wcproducts.com/kraken-x44/

### 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.

https://store.ctr-electronics.com/falcon-500-powered-by-talon-fx/ Hardware User Manual Wiring and mount instructions in PDF format.

https://ctre.download/files/user-manual/Falcon%20500%20v3%20User's%20Guide.pdf

## 16.1 Utilizing your Talon FX®

### 16.1.1 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 configured with default current limits, listed in the relevant config API documentation. However, 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.

> **Note**
>
> Current limits are applied in `TorqueCurrentFOC` in addition to the `PeakForwardTorque-Current` and `PeakReverseTorqueCurrent` configs, where the most restrictive limit is used. Torque current is equivalent to stator current in magnitude.

#### Stator Limits

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 from zero velocity.

#### Relationship between Supply and Stator Current

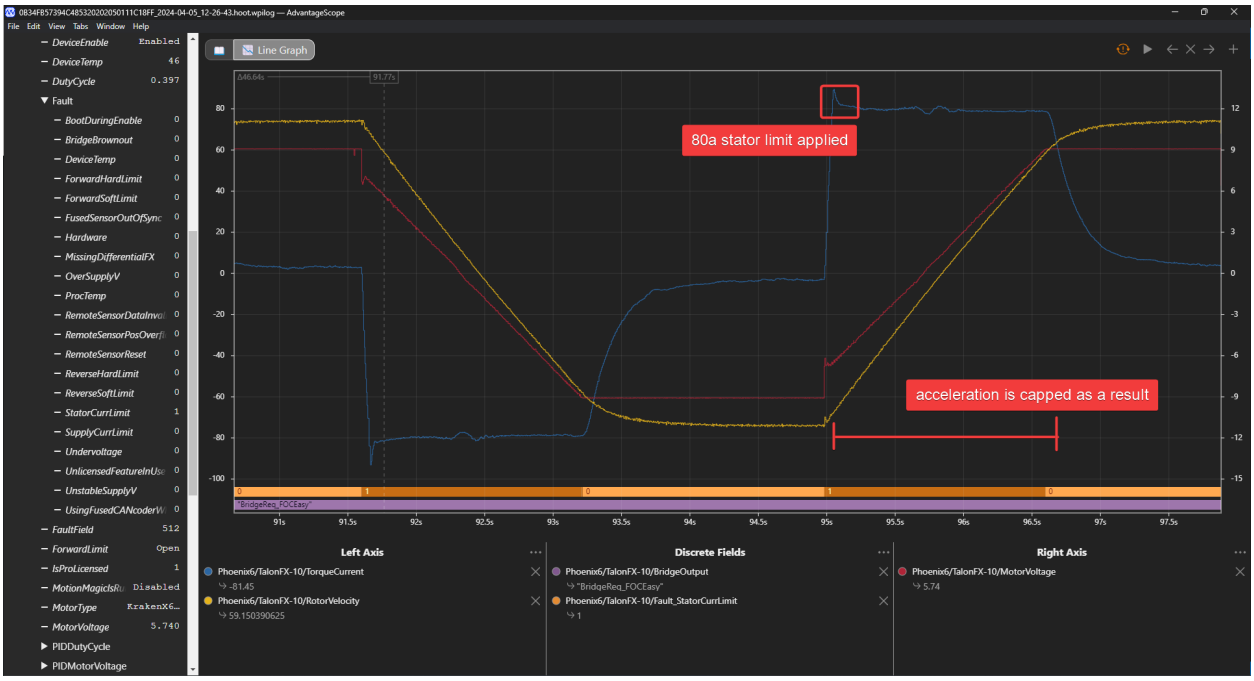



Fig. 1: *80 A stator current limit taking effect*

**Note**

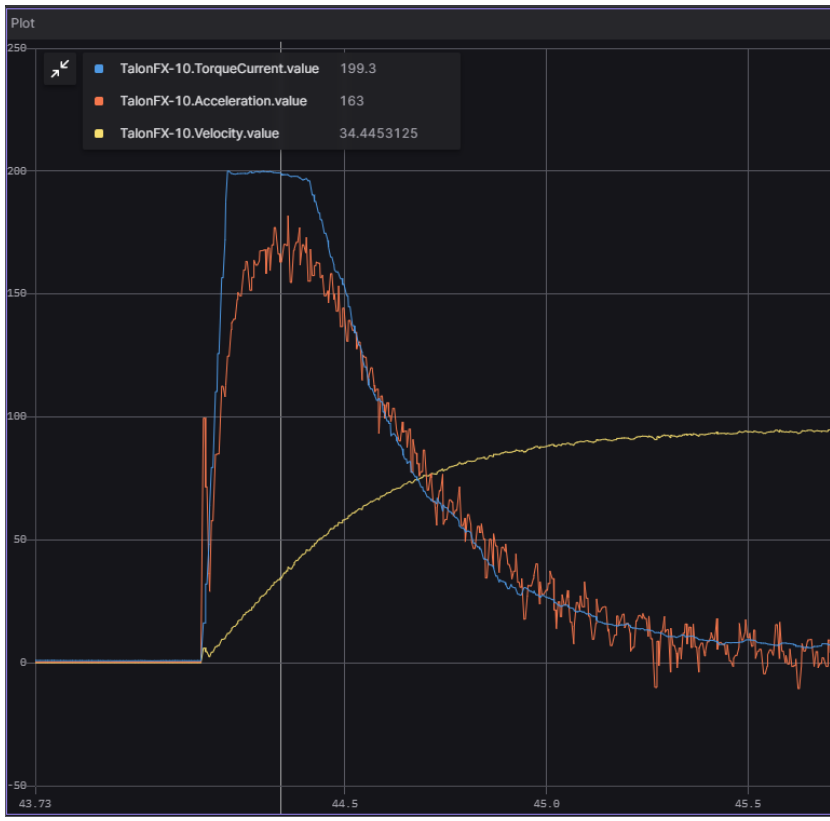
This explanation ignores energy loss from heat or other inefficiencies.

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 $I_{supply} = I_{stator} * dutycycle$.

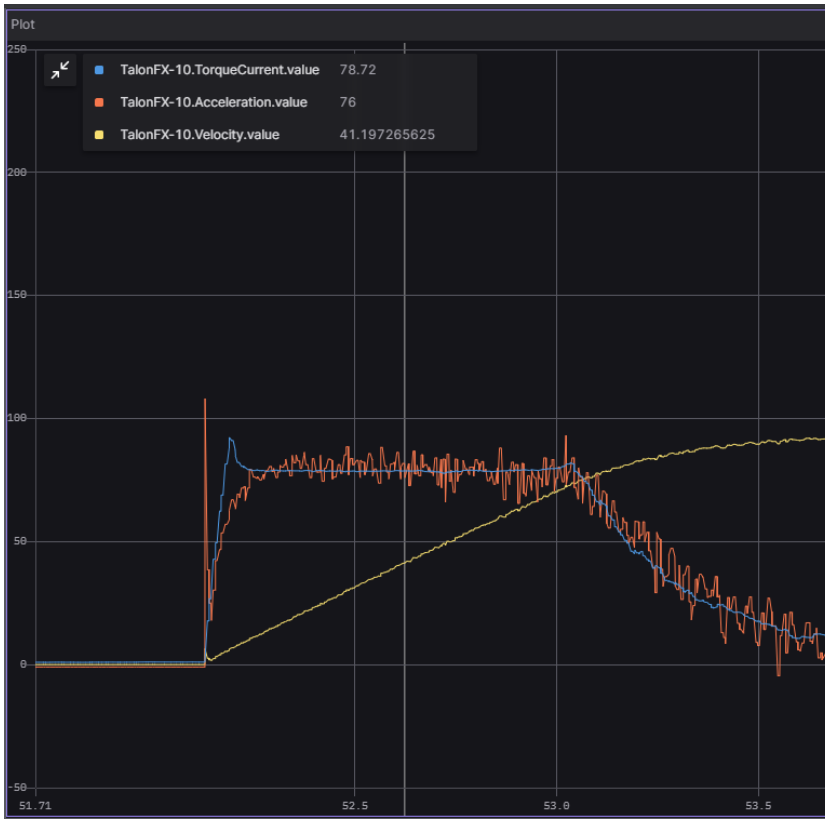
As an example, if a motor is applied 100% output (~12 V) and has 80 A of measured stator current, as is the case at a high rotor velocity, then the supply current will also be 80 A. However, if the motor is applied 50% output (~6 V) and has 80 A of measured stator current, as is the case at a low rotor velocity, 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

Supply current is the current drawn from the battery. As a result, limiting supply current can avoid brownouts, prevent breakers from tripping, and improve the longevity of the battery.

> **Tip**
>
> Since stator current limits also limit supply current, it is sometimes not necessary to enable both limits.

When enabled, the supply current limiter always ensures that the supply current does not exceed the configured `SupplyCurrentLimit`, preventing brownouts. Optionally, if the supply current limiter has been actively limiting for the `SupplyCurrentLowerTime`, the limit will be reduced to the `SupplyCurrentLowerLimit` until supply current is below that limit, preventing breakers from tripping.

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

### Determining Current Limits

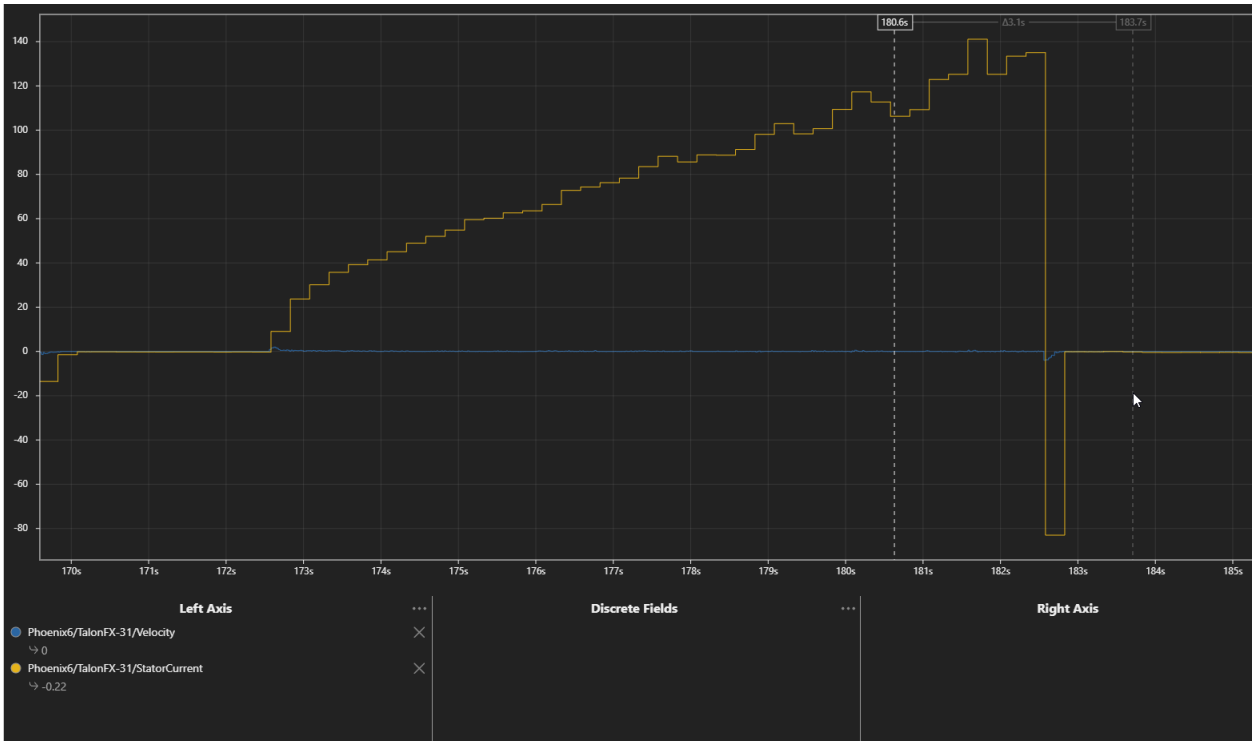
While supply current limits can be theoretically estimated by calculating max supply draw for every mechanism, stator limits are not easy to estimate. Determine your stator current limits first, then supply if necessary. In some cases, stator current limits are sufficient to prevent battery brownouts,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

$$V_{loaded} = V_{unloaded} - (I_{totalsupply} * R_{battery})$$

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 - 60 A stator, no supply limit (<60 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 <570 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 configs = new TalonFXConfiguration();

// enable stator current limit
configs.CurrentLimits.StatorCurrentLimit = 120;
configs.CurrentLimits.StatorCurrentLimitEnable = true;
// set up other configs here

talonFXConfigurator.apply(configs);
```

#### C++

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

// enable stator current limit
configs.CurrentLimits.StatorCurrentLimit = 120;
configs.CurrentLimits.StatorCurrentLimitEnable = true;
// set up other configs here

talonFXConfigurator.Apply(configs);
```

#### Python

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

# enable stator current limit
configs.current_limits.stator_current_limit = 120
configs.current_limits.stator_current_limit_enable = true
# set up other configs here

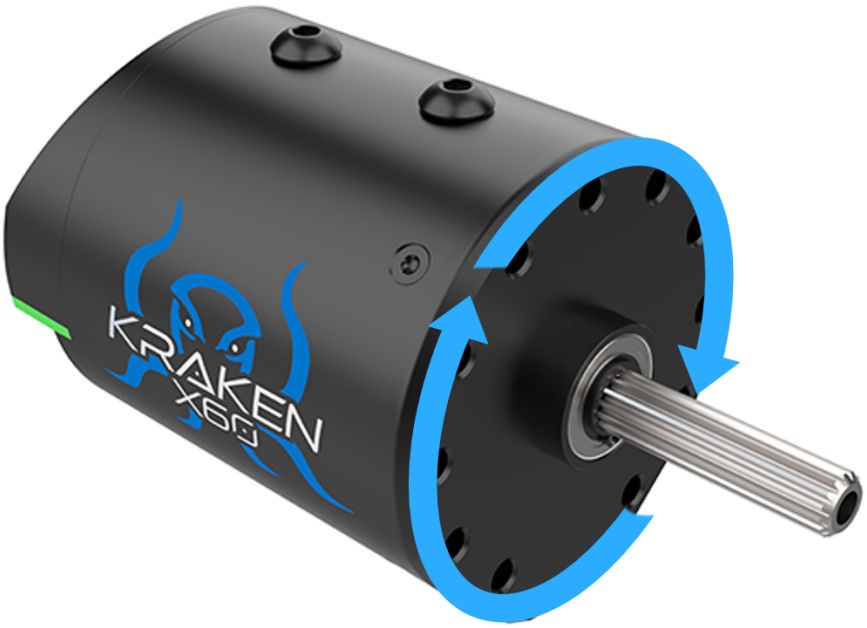
talonfx_configurator.apply(configs)
```

## 16.2 Motor Orientation

Talon FX® supports Clockwise and Counter Clockwise inverts. Inverts are determined from the perspective of looking at the face of the motor.




Counter Clockwise




Clockwise

## 16.3 Actuator Limits

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

**Note**

The Talon FX® + Kraken X60 does not support hardware limit switches. Instead, *control request limit* overrides can be used, or a CANcoder® can be used as a *remote limit switch*.

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

### 16.3.1 Limit Switches

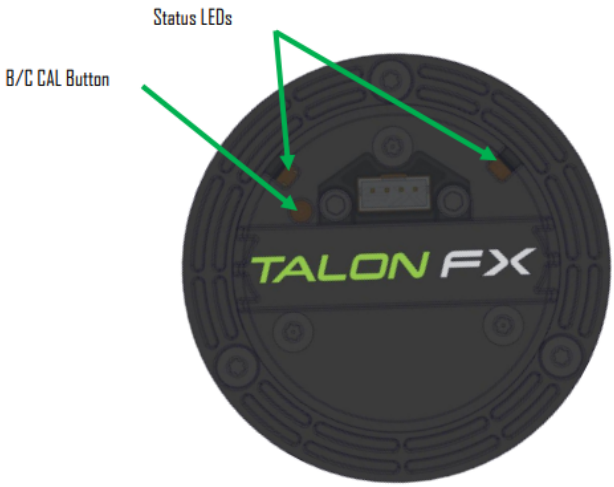
CTR Electronics supported actuators have limit features that will automatically neutral the actuator output (set voltage to 0) if a limit switch is activated. By default, limits are set to "normally open". This means that the switch needs to be explicitly closed (or grounded) for the actuator output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

## 16.4 Status Light Reference




**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

# 17 Talon FXS®

Talon FXS® is a versatile motor controller compatible with the CTRE software ecosystem. This standalone device seamlessly supports both brushless and brushed motors, offering unparalleled flexibility and performance.

The Talon FXS® supports the CTR Electronics Minion® and other third-party motors. Talon FXS® also employs methods to reduce hall sensor velocity measurement noise and phase delay - a common problem with similar standalone motor controllers.

Phoenix Pro users also benefit with “Advanced Hall Support” which can increase motor peak efficiency as high as an additional 2% percentage points and further reduce velocity measurement noise, making it ideal for velocity closed loop modes.

Store Page  CAD and purchase instructions.

https://store.ctr-electronics.com/products/talon-fxs

## 17.1 Supported Motors

- CTR Electronics Minion®
- Third-party NEO
- Third-party NEO 550
- Third-party NEO Vortex with Solo Adapter
- Most 3rd party brushed motors

## 17.2 Utilizing your Talon FXS®

## 17.3 Status Light Reference

**Important**

If the status lights do not exactly match any of the blink codes below, the device may be alternating between multiple blink codes (most commonly between good and bad CAN).

## 17.4 Troubleshooting Fault LED

A Talon FXS® thermal fault, as indicated with an "Offset Orange/Off" blink code, can be triggered for a variety of reasons. The following list can be used to help identify the reason and a potential solution.

- Brushless Motor
  - JST is disconnected or damaged. Plug-in the motor JST cable into the JST port and ensure the cable is not damaged.
  - Motor arrangement is incorrect. Please select the correct motor in *configs*.
  - Talon FXS® or motor has reached thermal cut-off. Allow time for the device to cool and consider configuring a *Stator Current limit*.
- Brushed Motor
  - Motor arrangement is incorrect. Please select one of the brushed options in *configs*.
  - Talon FXS® has reached thermal cut-off. Allow time for the device to cool and consider configuring a *Stator Current limit*.

# 18 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++, Python).

**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.

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

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

*Status Signals* Talon FX® Quickstart Quickstart on controlling a Talon FX® with open loop control requests and a Joystick.

*Open-Loop Control*

- ***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 Talon FX®.
- ***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.
- ***Hoot Replay***
  - Highlights playing back captured signals from a hoot log to test robot program changes.
- ***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.

## 18.1 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`
- `mechanisms` - advanced multi-device *mechanisms*, such as `DifferentialMechanism`
- `signals` - enumeration types for device *signals*
- `sim` - classes related to device *simulation*
- `swerve` - classes related to the Phoenix 6 *Swerve API*

### 18.1.1 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};
```

(continues on next page)

(continued from previous page)

```
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;`.

### 18.1.2 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 *`.

### 18.1.3 Thread Safety

The vast majority of Phoenix 6 is thread-safe with a few exceptions. Objects that are **not** thread-safe include:

- **`StatusSignal` objects**
  - Calling the same device `StatusSignal` getter (e.g. `TalonFX.getVelocity()`) from multiple threads is unsafe. This is because device signal getters refresh the `StatusSignal` implicitly.
  - Users should clone or copy a `StatusSignal` object to get a unique instance for a given thread.
- **`Config` objects**
  - Includes `TalonFX.setInverted()` and `TalonFX.setNeutralMode()`
  - However, device `Configurator` objects and other setters (e.g. `TalonFX.setPosition()`) **are** thread-safe.
- **`Control` objects**
  - However, sending a control request to a device **is** thread-safe.

## 18.2 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.

### 18.2.1 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).

> **Note**
>
> Phoenix 6 utilizes the C++ units library and, optionally, the Java units library when applicable. Using the Java units library may increase GC overhead.

#### Java

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

#### C++

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

#### Python

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

> **Tip**
>
> The device `Configuration` objects document all available configuration parameters in the documentation of each public member variable, making it easy to find specific configs (example).

#### Modifying Configurations

Configuration objects are mutable, so they can be saved in a member variable and reused. Additionally, configuration objects support modification using method chaining. This can be useful for constructing them as a class member variable or at compile time. In Java, this can also be used to provide unit types.

Configuration and config group objects can also be cloned, making it easy to share common configs across device configurations.

### Java

```
final TalonFXConfiguration commonConfigs = new TalonFXConfiguration()
   .withMotorOutput(
      new MotorOutputConfigs()
         .withNeutralMode(NeutralModeValue.Brake)
   )
   .withCurrentLimits(
      new CurrentLimitsConfigs()
         .withStatorCurrentLimit(Amps.of(120))
         .withStatorCurrentLimitEnable(true)
   );

/* create a copy with a different invert */
final TalonFXConfiguration leaderConfigs = commonConfigs.clone()
   .withMotorOutput(
      commonConfigs.MotorOutput.clone()
         .withInverted(InvertedValue.Clockwise_Positive)
   );
```

### C++

```
static constexpr configs::TalonFXConfiguration commonConfigs =
   configs::TalonFXConfiguration{}
      .WithMotorOutput(
         configs::MotorOutputConfigs{}
            .WithNeutralMode(signals::NeutralModeValue::Brake)
      )
      .WithCurrentLimits(
         configs::CurrentLimitsConfigs{}
            .WithStatorCurrentLimit(120_A)
            .WithStatorCurrentLimitEnable(true)
      );

/* create a copy with a different invert */
configs::TalonFXConfiguration leaderConfigs =
   configs::TalonFXConfiguration{commonConfigs}
      .WithMotorOutput(
         configs::MotorOutputConfigs{commonConfigs.MotorOutput}
            .WithInverted(signals::InvertedValue::Clockwise_Positive)
      );
```

### Python

```
self._common_configs = (
   configs.TalonFXConfiguration()
   .with_motor_output(
      configs.MotorOutputConfigs()
      .with_neutral_mode(signals.NeutralModeValue.BRAKE)
   )
   .with_current_limits(
      configs.CurrentLimitsConfigs()
```

(continues on next page)

(continued from previous page)

```
      .with_stator_current_limit(120.0)
      .with_stator_current_limit_enable(True)
   )
)

# create a copy with a different invert
self._leader_configs = (
   copy.deepcopy(self._common_configs)
   .with_motor_output(
      copy.deepcopy(self._common_configs.motor_output)
      .with_inverted(signals.InvertedValue.CLOCKWISE_POSITIVE)
   )
)
```

#### 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.

### 18.2.2 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. During initialization, it is recommended that a full device `Configuration` object is applied instead of separate config groups to ensure that configs left unmodified by code are factory defaulted.

> **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 TalonFXConfiguration();

// set invert to CW+
motorConfigs.MotorOutput.Inverted = InvertedValue.Clockwise_Positive;
// apply all configs to the motor
talonFXConfigurator.apply(motorConfigs);

// later in the match, change the neutral mode to Brake
motorConfigs.MotorOutput.NeutralMode = NeutralModeValue.Brake;
// apply only the MotorOutput configs, leaving all other configs untouched
talonFXConfigurator.apply(motorConfigs.MotorOutput);
```

### C++

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

// set invert to CW+
motorConfigs.MotorOutput.Inverted =␣
↪signals::InvertedValue::Clockwise_Positive;
// apply all configs to the motor
talonFXConfigurator.Apply(motorConfigs);

// later in the match, change the neutral mode to Brake
motorConfigs.MotorOutput.NeutralMode = signals::NeutralModeValue::Brake;
// apply only the MotorOutput configs, leaving all other configs untouched
talonFXConfigurator.Apply(motorConfigs.MotorOutput);
```

### Python

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

# set invert to CW+
motor_configs.motor_output.inverted = signals.InvertValue.CLOCKWISE_POSITIVE
# apply all configs to the motor
talonfx_configurator.apply(motor_configs)

# later in the match, change the neutral mode to Brake
motor_configs.motor_output.neutral_mode = signals.NeutralModeValue.BRAKE
# apply only the motor_output configs, leaving all other configs untouched
talonfx_configurator.apply(motor_configs.motor_output)
```

**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()`. Applying and refreshing a single config group can be

faster in this scenario.

#### 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())
```

## 18.3 Control Requests

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

**Note**

Phoenix 6 utilizes the C++ units library and, optionally, the Java units library when applicable. Using the Java units library may increase GC overhead.

### 18.3.1 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))
```

### 18.3.2 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. In Java, this can also be used to provide a unit type.

#### 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(Amps.of(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))
```

### 18.3.3 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++, Python) 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.

## 18.4 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.

### 18.4.1 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

```
// fetch with refresh
var supplyVoltageSignal = m_device.getSupplyVoltage();
// fetch WITHOUT refresh
var supplyVoltageSignal = m_device.getSupplyVoltage(false);
```

#### C++

```
// fetch with refresh
auto& supplyVoltageSignal = m_device.GetSupplyVoltage();
// fetch WITHOUT refresh
auto& supplyVoltageSignal = m_device.GetSupplyVoltage(false);
```

#### Python

```
# fetch with refresh
supply_voltage_signal = self.device.get_supply_voltage()
# fetch WITHOUT refresh
supply_voltage_signal = self.device.get_supply_voltage(False)
```

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

#### Java

```
// get the value as a unit type
var supplyVoltage = supplyVoltageSignal.getValue();
// convert the unit type to our desired units
double supplyVoltageVolts = supplyVoltage.in(Volts);

// alternatively, get the value as a double in the documented canonical units
double supplyVoltage = supplyVoltageSignal.getValueAsDouble();
```

#### C++

```
// get the value as a unit type
auto supplyVoltage = supplyVoltageSignal.GetValue();
// pull out the underlying value from the unit type
double supplyVoltageVolts = supplyVoltage.value();

// alternatively, get the value as a double in the documented canonical units
double supplyVoltage = supplyVoltageSignal.GetValueAsDouble();
```

#### Python

```
supply_voltage = supply_voltage_signal.value
```

**Note**

Phoenix 6 utilizes the Java units library and C++ units library when applicable. Using the Java units library may increase GC overhead.

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.

### 18.4.2 Refreshing the Signal Value

The device `StatusSignal` getters implicitly refresh the cached signal values by default. However, if the user application caches the `StatusSignal` object or passes in `false` to the device signal getters, the `refresh()` method must be called to fetch fresh data. Multiple signals can be refreshed in one call using `BaseStatusSignal.refreshAll()` (Java, C++, Python), which can improve performance compared to individual refreshes.

> **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);
// refresh an array or List of differential signals
BaseStatusSignal[] signals = {diffPositionSignal, diffVelocitySignal};
BaseStatusSignal.refreshAll(signals);
```

#### C++

```
// refresh the supply voltage signal
supplyVoltageSignal.Refresh();
// refresh the position and velocity signals
BaseStatusSignal::RefreshAll(positionSignal, velocitySignal);
// refresh a std::span of differential signals
std::vector<BaseStatusSignal*> signals{&diffPositionSignal, &
↪diffVelocitySignal};
BaseStatusSignal::RefreshAll(signals);
```

#### Python

```
# refresh the supply voltage signal
supply_voltage_signal.refresh()
# refresh the position and velocity signals
BaseStatusSignal.refresh_all(position_signal, velocity_signal)
# refresh a list of differential signals
signals = [diff_position_signal, diff_velocity_signal]
BaseStatusSignal.refresh_all(signals)
```

### 18.4.3 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)
```

### 18.4.4 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);
// speed up array or List of differential signals to 100 Hz
BaseStatusSignal[] signals = {diffPositionSignal, diffVelocitySignal};
BaseStatusSignal.setUpdateFrequencyForAll(100, signals);
```

### 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);
// speed up std::span of differential signals to 100 Hz
std::vector<BaseStatusSignal*> signals{&diffPositionSignal, &
↪diffVelocitySignal};
BaseStatusSignal::SetUpdateFrequencyForAll(100_Hz, signals);
```

### 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)
# speed up list of differential signals to 100 Hz
signals = [diff_position_signal, diff_velocity_signal]
BaseStatusSignal.set_update_frequency_for_all(100, signals)
```

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 or slow down 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 or slowed down. This results in an opt-in model for status signals, maximizing the reduction in bus utilization.

**Tip**

The update frequency of optimized signals can be specified (Java, C++, Python), where 0 Hz completely disables the signals. The default update frequency is 4 Hz, ensuring that all signals are available when using *Signal Logging*.

**Warning**

When using followers, the leader motor must keep the `DutyCycle`, `MotorVoltage`, and `TorqueCurrent` status signals enabled. Additionally, remote sensors must keep related status signals enabled (such as position and velocity).

#### 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)
```

### Resetting All to Default

The update frequencies of all status signals for a device can be reset to the defaults by calling `resetSignalFrequencies()` on the device (Java, C++, Python). Additionally, multiple devices can be reset at once using `ParentDevice.resetSignalFrequenciesForAll()` (Java, C++, Python).

Since devices typically maintain their configured status signal update frequencies until they are power cycled, this can be useful to restore everything to the defaults before reconfiguring update frequencies.

#### Java

```
m_pigeon.resetSignalFrequencies();
ParentDevice.resetSignalFrequenciesForAll(m_leftMotor, m_rightMotor,␣
↪m_cancoder);
```

#### C++

```
m_pigeon.ResetSignalFrequencies();
hardware::ParentDevice::ResetSignalFrequenciesForAll(m_leftMotor,␣
↪m_rightMotor, m_cancoder);
```

#### Python

```
self.pigeon.reset_signal_frequencies()
hardware.ParentDevice.reset_signal_frequencies_for_all(self.left_motor, self.
↪right_motor, self.cancoder)
```

### 18.4.5 Timestamps

StatusSignals can have multiple timestamps associated from them, as there are often multiple sources of time. Users can call `getTimestamp()` (Java, C++, Python) to return the "best" timestamp determined by Phoenix.

`getAllTimestamps()` can also be used to return all timestamps associated with a given `StatusSignal`, which returns a collection of `Timestamp` (Java, C++, Python) objects. The `Timestamp` objects can be used to perform latency compensation math.

### 18.4.6 Timesync®

**Important**

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

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

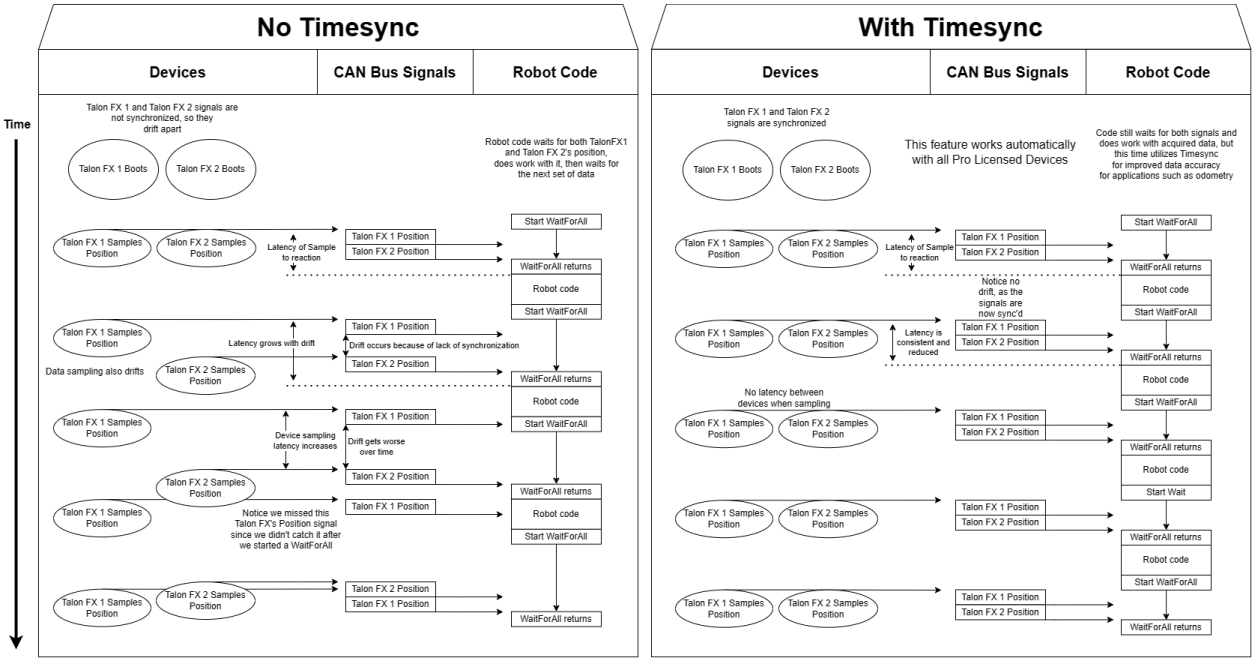
Users can synchronously wait for these signals to update using `BaseStatusSignal.waitForAll()` (Java, C++, Python).

> **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® enabled 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 Timesync®.

### Java

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

BaseStatusSignal.waitForAll(0.020, talonFXPositionSignal,␣
↪cancoderPositionSignal, pigeon2YawSignal);
// can also send down an array or List of signals
```

(continues on next page)

(continued from previous page)

```
BaseStatusSignal[] signals = {talonFXPositionSignal, cancoderPositionSignal,␣
↪pigeon2YawSignal};
BaseStatusSignal.waitForAll(0.020, signals);
```

#### C++

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

BaseStatusSignal::WaitForAll(20_ms, talonFXPositionSignal,␣
↪cancoderPositionSignal, pigeon2YawSignal);
// can also send down a std::span of signal pointers
std::vector<BaseStatusSignal*> signals{&talonFXPositionSignal, &
↪cancoderPositionSignal, &pigeon2YawSignal};
BaseStatusSignal::WaitForAll(20_ms, signals);
```

#### Python

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

BaseStatusSignal.wait_for_all(0.020, talonfx_position_signal,␣
↪cancoder_position_signal, pigeon2_yaw_signal)
# can also send down a list of signals
signals = [talonfx_position_signal, cancoder_position_signal,␣
↪pigeon2_yaw_signal]
BaseStatusSignal.wait_for_all(0.020, signals)
```

### 18.4.7 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()
)
```

### 18.4.8 StatusSignalCollection

The StatusSignalCollection (Java, C++, Python) class provides a lightweight wrapper around a list of status signals on a common network. This simplifies the process of refreshing or waiting on multiple status signals.

#### Java

```
final StatusSignalCollection signals = new StatusSignalCollection();

// register all the signals we want to refresh (on the same network)
signals.addSignals(
   m_talonFX.getPosition(false),
   m_cancoder.getPosition(false),
   m_pigeon2.getYaw(false)
);
// set all the signals to a 200 Hz update frequency
signals.setUpdateFrequencyForAll(Hertz.of(200));

// now wait on all the signals in the collection
signals.waitForAll(0.010);
```

#### C++

```
StatusSignalCollection signals{};

// register all the signals we want to refresh (on the same network)
signals.AddSignals(
   m_talonFX.GetPosition(false),
   m_cancoder.GetPosition(false),
   m_pigeon2.GetYaw(false)
);
// set all the signals to a 200 Hz update frequency
signals.SetUpdateFrequencyForAll(200_Hz);

// now wait on all the signals in the collection
signals.WaitForAll(10_ms);
```

#### Python

```
self.signals = StatusSignalCollection()

# register all the signals we want to refresh (on the same network)
self.signals.add_signals(
   self.talonfx.get_position(False),
   self.cancoder.get_position(False),
   self.pigeon2.get_yaw(False)
)
# set all the signals to a 200 Hz update frequency
self.signals.set_update_frequency_for_all(200.0)

# now wait on all the signals in the collection
self.signals.wait_for_all(0.010)
```

### 18.4.9 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.

## 18.5 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.
- **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** on a roboRIO 1 with a USB flash drive or a roboRIO 2, where logging is started by any of the following (whichever occurs first):

- It has been at least 1 second since program startup (allowing for calls to `setPath`), and the robot is enabled.
- It has been at least 5 seconds since program startup (allowing for calls to `setPath`), and the Driver Station is connected to the robot.

Users can disable this behavior by calling `SignalLogger.enableAutoLogging(false)` (Java, C++, Python) at robot program startup.

> **Tip**
>
> Device status signals can also be viewed live in the *Tuner X Plotting page*.

### 18.5.1 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. All logs will be placed in a subfolder named after the date and time of the start of the program.

### 18.5.2 Start/Stop Logging

The signal logger can be started and stopped using the `start()` and `stop()` functions (Java, C++, Python).

#### Java

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

#### C++

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

#### Python

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

### 18.5.3 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.

> **Tip**
>
> In a WPILib robot project, custom data types can be logged using Struct and Protobuf. Additionally, Java robot projects can take advantage of *Epilogue integration*.

#### Java

```
// Log the odometry pose
SignalLogger.writeStruct("odometry", Pose2d.struct, pose);
// Log the odometry period with units of "seconds"
SignalLogger.writeDouble("odom period", state.OdometryPeriod, "seconds");
// Log the camera pose with calculated latency
SignalLogger.writeStruct(
   "camera pose", Pose2d.struct, camPose,
   Timer.getTimestamp() - camRes.getTimestampSeconds()
);
```

#### C++

```
// Log the odometry pose
SignalLogger::WriteStruct<frc::Pose2d>("odometry", pose);
// Log the odometry period with units of "seconds"
SignalLogger::WriteDouble("odom period", state.OdometryPeriod, "seconds");
// Log the camera pose with calculated latency
```

(continues on next page)

(continued from previous page)

```
SignalLogger::WriteStruct<frc::Pose2d>(
   "camera pose", camPose,
   frc::Timer::GetTimestamp() - camRes.GetTimestamp()
);
```

#### Python

```
# Log the odometry pose
SignalLogger.write_struct("odometry", Pose2d, pose)
# 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_struct(
   "camera pose", Pose2d, cam_pose,
   Timer.getTimestamp() - cam_res.getTimestamp()
)
```

### 18.5.4 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
- SupplyVoltage
- Fault_UnlicensedFeatureInUse
- Fault_BootDuringEnable
- Fault_Hardware
- Fault_Undervoltage

#### Talon FX®

- SupplyCurrent
- StatorCurrent
- MotorVoltage
- Position
- Velocity
- DeviceEnable

- RobotEnable
- ConnectedMotor
- Fault_DeviceTemp
- Fault_ProcTemp
- Fault_RemoteSensorDataInvalid
- Fault_StaticBrakeDisabled
- Fault_BridgeBrownout
- Fault_RotorFault1
- Fault_RotorFault2

### Talon FXS®

- SupplyCurrent
- StatorCurrent
- MotorVoltage
- Position
- Velocity
- DeviceEnable
- RobotEnable
- ConnectedMotor
- Fault_DeviceTemp
- Fault_ProcTemp
- Fault_RemoteSensorDataInvalid
- Fault_StaticBrakeDisabled
- Fault_BridgeBrownout
- Fault_HallSensorMissing
- Fault_DriveDisabledHallSensor
- Fault_MotorTempSensorMissing
- Fault_MotorTempSensorTooHot
- Fault_MotorArrangementNotSelected

### CANcoder®

- Position
- Velocity

### Pigeon® 2.0

- Yaw
- AngularVelocityZWorld
- NoMotionEnabled
- NoMotionCount
- UpTime

### CANrange®

- DistanceMeters
- ProximityDetected
- SignalStrength

### CANdi®

- Pin1State
- Pin2State
- S1Closed
- S2Closed
- QuadPosition
- QuadVelocity
- Pwm1_Position
- Pwm1_Velocity
- Pwm2_Position
- Pwm2_Velocity
- Overcurrent
- Fault_5V

### CANdle®

- OutputCurrent
- DeviceTemp
- MaxSimultaneousAnimationCount
- Fault_Overvoltage
- Fault_5VTooHigh
- Fault_5VTooLow
- Fault_Thermal
- Fault_SoftwareFuse
- Fault_ShortCircuit

### 18.5.5 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.

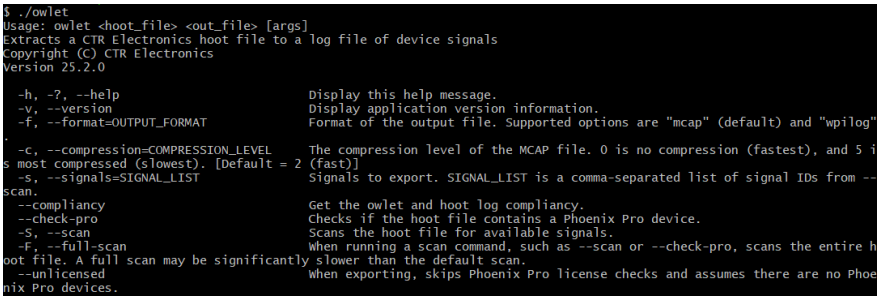
```
[phoenix] Signal Logger: Available disk space (3 MB) below 5 MB, stopping log
```

### 18.5.6 Converting Signal Logs

Signal logs can be converted to other common file formats such as WPILOG or MCAP using the *Tuner X Log Extractor*.

Additionally, the `owlet` CLI tool can be used from a terminal, including on platforms not supported by Tuner X. `owlet` can be downloaded from the CLI Tools download page.

To view a list of available commands, run `owlet` either with no parameters or with `--help`.




As an example, to convert a `hoot` file to WPILOG, run:

```
./owlet -f wpilog "input.hoot" "output.wpilog"
```

## 18.6 Hoot Replay

> **Note**
>
> Information on logging to `hoot` files can be found in *Signal Logging*.

Hoot Replay allows users to playback hoot logs in their robot program. This allows for testing changes to robot code in simulation using measurements recorded from the real robot.

> **Important**
>
> Hoot Replay requires the hoot log to have a Pro-licensed device. Currently, only one hoot log may be replayed at a time.

Compared to other replay frameworks, Hoot Replay offers **automatic** status signal playback without any code changes. However, replaying custom logged signals still requires modifications to the robot's subsystems.

Hoot Replay is controlled using the `HootReplay` class (Java, C++, Python) and supports playing back device status signals and custom user signals. Configs and control requests are ignored during replay.

Hoot Replay uses a different vendordep (see *Installing Phoenix 6*) that replaces *Hardware-Attached Simulation* with Hoot Replay. Note that only one Phoenix 6 vendordep may be used in the `vendordeps` folder at a time.

### 18.6.1 Starting Hoot Replay

At the start of the robot program, the desired hoot log can be loaded using `HootReplay.loadFile(fileName)` (Java, C++, Python). Hoot Replay will start automatically after the log has been loaded.

#### Java

```
HootReplay.loadFile("./logs/example.hoot");
```

#### C++

```
HootReplay::LoadFile{"./logs/example.hoot"};
```

#### Python

```
HootReplay.load_file("./logs/example.hoot")
```

It is important that the hoot log is loaded before any devices are constructed or configured. To make this process easier, the `CANBus(canbus, fileName)` constructor (Java, C++, Python) can instead be used to load the hoot log.

#### Java

```
// construct the CANBus with the hoot log
final CANBus canbus = new CANBus("canivore", "./logs/example.hoot");
// now the log is guaranteed to be loaded before devices are constructed
final TalonFX talonFX = new TalonFX(0, canbus);
```

#### C++

```
// construct the CANBus with the hoot log
CANBus canbus{"canivore", "./logs/example.hoot"};
// now the log is guaranteed to be loaded before devices are constructed
hardware::TalonFX talonFX{0, canbus};
```

#### Python

```
# construct the CANBus with the hoot log
self._canbus = CANBus("canivore", "./logs/example.hoot")
# now the log is guaranteed to be loaded before devices are constructed
self._talon_fx = hardware.TalonFX(0, canbus)
```

### 18.6.2 Controlling Replay

During Hoot Replay, the simulated robot will automatically enable and run through all the maneuvers recorded in the hoot log. There are also several APIs that can be used to manage playback.

- `HootReplay.pause()` temporarily pauses playback
- `HootReplay.play()` resumes playback from the current point in the file
- `HootReplay.stop()` pauses playback and returns to the start of the file
- `HootReplay.restart()` restarts playback from the start of the file
- `HootReplay.setSpeed(speed)` changes the speed of playback by a scalar

The status of Hoot Replay can be checked using `HootReplay.isFileLoaded()`, `HootReplay.isPlaying()`, and `HootReplay.isFinished()`. Threads can also wait for Hoot Replay to start using `HootReplay.waitForPlaying(timeout)`.

> **Tip**
>
> `HootReplay.isPlaying()` and `HootReplay.waitForPlaying(timeout)` immediately return true when running on the robot or in regular simulation.

Additionally, after pausing or stopping Hoot Replay, playback can advance by a fixed timestep using `HootReplay.stepTiming(timeDelta)`, updating all signals accordingly.

#### Java

```
// stop and return to start of log
HootReplay.stop();
var startPos = talonFX.getPosition().getValue();
// advance by 1 second and compare positions
HootReplay.stepTiming(1.0);
var endPos = talonFX.getPosition().getValue();
```

#### C++

```
// stop and return to start of log
HootReplay::Stop();
auto const startPos = talonFX.GetPosition().GetValue();
// advance by 1 second and compare positions
HootReplay::StepTiming(1_s);
auto const endPos = talonFX.GetPosition().GetValue();
```

#### Python

```
# stop and return to start of log
HootReplay.stop()
start_pos = self._talon_fx.get_position().value
# advance by 1 second and compare positions
HootReplay.step_timing(1.0)
end_pos = self._talon_fx.get_position().value
```

### 18.6.3 Replaying Custom Signals

Users can also fetch custom signals written to the loaded hoot log by utilizing the `get*()` functions. An example application of this is replaying vision data to test changes in the drivetrain pose estimator.

All custom signal getters return a `StatusSignal.SignalMeasurement` (Java, C++, Python) object containing information about the signal, including its timestamp and any logged units. The success of fetching the custom signal can be validated by checking the `status` field.

#### Java

```
// Fetch the logged raw vision measurements
var visionData = HootReplay.getStruct("camera pose", Pose2d.struct);
if (visionData.status.isOK() && visionData.value.isPresent()) {
   // now run regular vision processing on the vision data
   var cameraPose = visionData.value.get();
   if (isCameraPoseValid(cameraPose)) {
      drivetrain.addVisionMeasurement(cameraPose, visionData.timestamp);
   }
}
```

#### C++

```
// Fetch the logged raw vision measurements
auto const visionData = HootReplay::GetStruct<frc::Pose2d>("camera pose");
if (visionData.status.IsOK() && visionData.value.has_value()) {
   // now run regular vision processing on the vision data
   auto const cameraPose = visionData.value.value();
   if (IsCameraPoseValid(cameraPose)) {
      drivetrain.AddVisionMeasurement(cameraPose, visionData.timestamp);
   }
}
```

#### Python

```
# Fetch the logged raw vision measurements
vision_data = HootReplay.get_struct("camera pose", Pose2d)
if visionData.status.is_ok() and vision_data.value is not None:
   # now run regular vision processing on the vision data
   camera_pose = vision_data.value
   if is_camera_pose_valid(camera_pose):
      drivetrain.add_vision_measurement(camera_pose, vision_data.timestamp)
```

Additionally, the Signal Logger is supported in Hoot Replay with the following behavior:

- Signal Logger is always enabled during replay. As a result, `SignalLogger::Start()` and `SignalLogger::Stop()` are ignored.
- The replayed log will always be written to a `replay_<date>_<time>/` subfolder next to the original log.
- All custom signals written during replay will be automatically placed under `hoot_replay/` in the log.

- The replayed log also contains all status signals and custom signals from the original log, excluding those starting with `hoot_replay/`.

This can be useful to log the new outputs after making changes to program logic.

### Adjusting Robot Code Architecture

Replaying **custom** signals typically requires changes to the architecture of the robot's subsystems or other hardware classes. As an example, consider changes to a `Vision` class.

When working with vision, a robot program will typically fetch raw/unfiltered camera/pose data (the "inputs") from the vision library (such as a list of targets). Then, it will perform some additional processing and filtering on that data to get the final vision pose estimate (the "output").

The goal is to **replay the "inputs"** so the replayed program can test changes to the subsequent data processing and filtering. This means the robot program should only replay data that is **unaffected by code logic**, letting the program determine all the new outputs.

The simplest way to make the code easily replayed is to consolidate all input fetches into a single `fetchInputs()` function and save the results to class member variables. From there, a single periodic function would be responsible for calling that function when `Utils.isReplay()` (Java, C++, Python) returns `false`, as well as performing all data processing to get the outputs.

For example, if the only data pulled from the vision library is the raw vision pose estimate and its timestamp:

#### Java

```java
public class Vision {
   private final Camera camera = new Camera(...);
   /* vision inputs */
   private PoseEstimate cameraPoseEst = new PoseEstimate();

   // ...

   private void fetchInputs() {
      cameraPoseEst = camera.getPoseEstimate();
   }

   public void periodic() {
      if (!Utils.isReplay()) {
         fetchInputs();
      }

      // process inputs here...
   }
}
```

#### C++

```cpp
class Vision {
private:
```

(continues on next page)

(continued from previous page)

```
   Camera camera{...};
   /* vision inputs */
   PoseEstimate cameraPoseEst{};

   // ...

   void FetchInputs()
   {
      cameraPoseEst = camera.GetPoseEstimate();
   }

public:
   void Periodic()
   {
      if (!utils::IsReplay()) {
         FetchInputs();
      }

      // process inputs here...
   }
};
```

#### Python

```
class Vision:
   def __init__(self):
      self._camera = Camera(...)
      # vision inputs
      self._camera_pose_est = PoseEstimate()

      # ...

   def _fetch_inputs(self):
      self._camera_pose_est = camera.getPoseEstimate()

   def periodic(self):
      if not utils.is_replay():
         self._fetch_inputs()

      # process inputs here...
```

At this point, there are two methods to set up Hoot Replay of the inputs.

#### Automatic (`HootAutoReplay`)

The inputs can be registered to a `HootAutoReplay` (Java, C++, Python) instance using getter and setter lambdas. The periodic function then calls `update()` on the `HootAutoReplay` instance to write the inputs to the hoot log on the real robot (or *simulation*) and fetch the inputs from the log during Hoot Replay.

> **Tip**
>
> In a WPILib robot program, `HootAutoReplay` also supports replaying `Timer.getTimestamp()` and joystick inputs. These can be registered using `withTimestampReplay()` and `withJoystickReplay()`, respectively. See the API documentation for more information.

For example, the previous `Vision` example would now have:

### Java

```
public class Vision {
   private final Camera camera = new Camera(...);
   /* vision inputs */
   private PoseEstimate cameraPoseEst = new PoseEstimate();

   /* NEW: register inputs for replay */
   private final HootAutoReplay autoReplay = new HootAutoReplay()
      .withStruct(
         "CameraPoseEst/pose", Pose2d.struct,
         /* getter lambda returns the value to log */
         () -> cameraPoseEst.pose,
         /* setter lambda applies the value from the log */
         val -> cameraPoseEst.pose = val.value
      )
      .withDouble(
         "CameraPoseEst/timestamp",
         () -> cameraPoseEst.timestamp,
         val -> cameraPoseEst.timestamp = val.value
      );

   // ...

   private void fetchInputs() {
      cameraPoseEst = camera.getPoseEstimate();
   }

   public void periodic() {
      if (!Utils.isReplay()) {
         fetchInputs();
      }
      /* NEW: log the inputs on hardware/sim, and playback in replay */
      autoReplay.update();

      // process inputs here...
   }
}
```

### C++

```cpp
class Vision {
private:
   Camera camera{...};
   /* vision inputs */
   PoseEstimate cameraPoseEst{};

   /* NEW: register inputs for replay */
   HootAutoReplay autoReplay = HootAutoReplay{}
      .WithStruct<frc::Pose2d>(
         "CameraPoseEst/pose",
         /* getter lambda returns the value to log */
         [this] -> frc::Pose2d const& { return cameraPoseEst.pose; },
         /* setter lambda applies the value from the log */
         [this](HootReplay::SignalData<frc::Pose2d> val) {
            cameraPoseEst.pose = val.value;
         }
      )
      .WithValue<units::second_t>(
         "CameraPoseEst/timestamp",
         [this] -> units::second_t const& { return cameraPoseEst.timestamp; },
         [this](HootReplay::SignalData<units::second_t> val) {
            cameraPoseEst.timestamp = val.value;
         }
      );

   // ...

   void FetchInputs()
   {
      cameraPoseEst = camera.GetPoseEstimate();
   }

public:
   void Periodic()
   {
      if (!utils::IsReplay()) {
         FetchInputs();
      }
      /* NEW: log the inputs on hardware/sim, and playback in replay */
      autoReplay.Update();

      // process inputs here...
   }
};
```

### Python

```python
class Vision:
   def __init__(self):
      self._camera = Camera(...)
```

(continues on next page)

(continued from previous page)

```
      # vision inputs
      self._camera_pose_est = PoseEstimate()

      # NEW: register inputs for replay
      def _set_camera_pose(val: HootReplay.SignalData[Pose2d]):
         self._camera_pose_est.pose = val.value
      def _set_camera_timestamp(val: HootReplay.SignalData[float]):
         self._camera_pose_est.timestamp = val.value
      self._auto_replay = (
         HootAutoReplay()
         .with_struct(
            "CameraPoseEst/pose",
            # getter lambda returns the value to log
            lambda: self._camera_pose_est.pose,
            # setter lambda applies the value from the log
            _set_camera_pose,
         )
         .with_double(
            "CameraPoseEst/timestamp",
            lambda: self._camera_pose_est.timestamp,
            _set_camera_timestamp
         )
      )

      # ...

   def _fetch_inputs(self):
      self._camera_pose_est = camera.getPoseEstimate()

   def periodic(self):
      if not utils.is_replay():
         self._fetch_inputs()
      # NEW: log the inputs on hardware/sim, and playback in replay
      self._auto_replay.update()

      # process inputs here...
```

### Manual (`HootReplay.get*()`)

The `fetchInputs()` function should also log the inputs using `SignalLogger`. Then, it is possible to implement a `fetchInputsReplay()` function, which should be called when `Utils.isReplay()` returns `true`. This function would do the reverse of the regular `fetchInputs()`, pulling out the data from `HootReplay` and saving it to the class member variables.

For example, the previous `Vision` example would now have:

### Java

```
public class Vision {
   private final Camera camera = new Camera(...);
   /* vision inputs */
   private PoseEstimate cameraPoseEst = new PoseEstimate();
```

(continues on next page)

(continued from previous page)

```
   // ...

   private void fetchInputs() {
      cameraPoseEst = camera.getPoseEstimate();

      /* NEW: manually log the inputs */
      SignalLogger.writeStruct("CameraPoseEst/pose", Pose2d.struct,␣
↪cameraPoseEst.pose);
      SignalLogger.writeDouble("CameraPoseEst/timestamp",␣
↪cameraPoseEst.timestamp);
   }

   private void fetchInputsReplay() {
      /* NEW: pull out inputs from the log */
      final var cameraPoseEst_pose = HootReplay.getStruct("CameraPoseEst/pose
↪", Pose2d.struct);
      final var cameraPoseEst_timestamp = HootReplay.getDouble(
↪"CameraPoseEst/timestamp");

      if (
         cameraPoseEst_pose.status.isOK() &&
         cameraPoseEst_timestamp.status.isOK()
      ) {
         cameraPoseEst = new PoseEstimate();
         cameraPoseEst.pose = cameraPoseEst_pose.value;
         cameraPoseEst.timestamp = cameraPoseEst_timestamp.value;
      }
   }

   public void periodic() {
      /* NEW: fetch the correct inputs based on if we're in replay */
      if (Utils.isReplay()) {
         fetchInputsReplay();
      } else {
         fetchInputs();
      }

      // process inputs here...
   }
}
```

### C++

```
class Vision {
private:
   Camera camera{...};
   /* vision inputs */
   PoseEstimate cameraPoseEst{};

   // ...
```

(continues on next page)

(continued from previous page)

```
   void FetchInputs()
   {
      cameraPoseEst = camera.GetPoseEstimate();

      /* NEW: manually log the inputs */
      SignalLogger::WriteStruct<frc::Pose2d>("CameraPoseEst/pose",␣
↪cameraPoseEst.pose);
      SignalLogger::WriteValue("CameraPoseEst/timestamp",␣
↪cameraPoseEst.timestamp);
   }

   void FetchInputsReplay()
   {
      /* NEW: pull out inputs from the log */
      auto const cameraPoseEst_pose =
         HootReplay::GetStruct<frc::Pose2d>("CameraPoseEst/pose");
      auto const cameraPoseEst_timestamp =
         HootReplay::GetValue<units::second_t>("CameraPoseEst/timestamp");

      if (
         cameraPoseEst_pose.status.IsOK() &&
         cameraPoseEst_timestamp.status.IsOK()
      ) {
         cameraPoseEst = PoseEstimate{};
         cameraPoseEst.pose = cameraPoseEst_pose.value.value();
         cameraPoseEst.timestamp = cameraPoseEst_timestamp.value;
      }
   }

public:
   void Periodic()
   {
      /* NEW: fetch the correct inputs based on if we're in replay */
      if (utils::IsReplay()) {
         FetchInputsReplay();
      } else {
         FetchInputs();
      }

      // process inputs here...
   }
};
```

### Python

```
class Vision:
   def __init__(self):
      self._camera = Camera(...)
      # vision inputs
      self._camera_pose_est = PoseEstimate()
```

(continues on next page)

(continued from previous page)

```
    # ...

  def _fetch_inputs(self):
    self._camera_pose_est = camera.getPoseEstimate()

    # NEW: manually log the inputs
    SignalLogger.write_struct("CameraPoseEst/pose", Pose2d, self._
↪camera_pose_est.pose)
    SignalLogger.write_double("CameraPoseEst/timestamp", self._
↪camera_pose_est.timestamp)

  def _fetch_inputs_replay(self):
    # NEW: pull out inputs from the log
    camera_pose_est_pose = HootReplay.get_struct("CameraPoseEst/pose")
    camera_pose_est_timestamp = HootReplay.get_double(
↪"CameraPoseEst/timestamp")

    if (
      camera_pose_est_pose.status.is_ok()
      and camera_pose_est_timestamp.status.is_ok()
    ):
      self._camera_pose_est = PoseEstimate()
      self._camera_pose_est.pose = camera_pose_est_pose.value
      self._camera_pose_est.timestamp = camera_pose_est_timestamp.value

  def periodic(self):
    # NEW: fetch the correct inputs based on if we're in replay
    if utils.is_replay():
      self._fetch_inputs_replay()
    else:
      self._fetch_inputs()

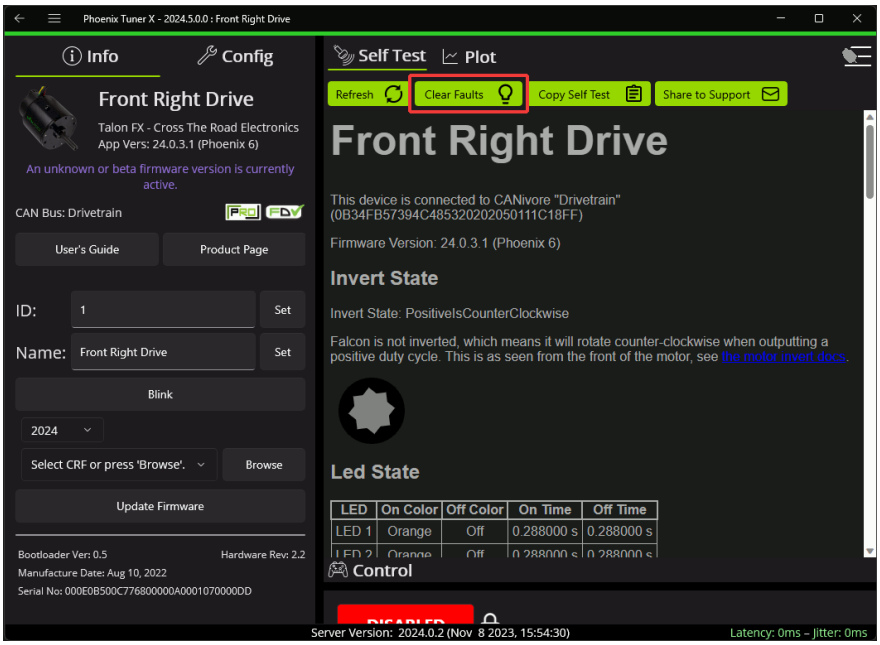
    # process inputs here...
```

## 18.7 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.




### 18.7.1 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.

### 18.7.2 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()
```

## 18.8 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.

### 18.8.1 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.

### 18.8.2 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.

### 18.8.3 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*.

## 18.9 Actuator Limits

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

> **Note**
>
> The Talon FX® + Kraken X60 does not support hardware limit switches. Instead, *control request limit* overrides can be used, or a CANcoder®/CANdi®/CANrange® can be used as a *remote limit switch*.

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

### 18.9.1 Retrieving Limit Switch State

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

#### Java

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

var forwardSoftLimit = m_motor.getFault_ForwardSoftLimit();
if (forwardSoftLimit.getValue()) {
   // do action when forward soft limit is reached
}
```

#### C++

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

auto& forwardSoftLimit = m_motor.GetFault_ForwardSoftLimit();
if (forwardSoftLimit.GetValue()) {
   // do action when forward soft limit is reached
}
```

#### 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

forward_soft_limit = self.motor.get_fault_forward_soft_limit()
if forward_soft_limit.value:
   # do action when forward soft limit is reached
```

### 18.9.2 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())
```

(continues on next page)

(continued from previous page)

```
      .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())
)
```

### 18.9.3 Remote Limit Switches

Supported devices (Talon FX®, CANifier®, CANcoder®, CANdi®, CANrange®) can be utilized as a remote limit switch, disabling actuator outputs when triggers.

- When utilizing a CANcoder® as a remote limit, the limit will trigger when the magnet strength changes from BAD (red) to ADEQUATE (orange) or GOOD (green).
- When utilizing a CANrange® as a remote limit, the limit will trigger when the proximity detect is tripped following the `ProximityParamsConfigs` (Java, C++, Python).
- When utilizing a CANdi® as a remote limit, the limit will trigger when the `S1Closed` or `S2Closed` signal is true.

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

#### Java

```
var configs = new TalonFXConfiguration();
configs.HardwareLimitSwitch.ForwardLimitSource =␣
↪ForwardLimitSourceValue.RemoteCANcoder;
```

(continues on next page)

(continued from previous page)

```
configs.HardwareLimitSwitch.ForwardLimitRemoteSensorID =␣
↪m_cancoder.getDeviceID();
// set up other configs here

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

#### C++

```
configs::TalonFXConfiguration configs{};
configs.HardwareLimitSwitch.ForwardLimitSource =␣
↪signals::ForwardLimitSourceValue::RemoteCANcoder;
configs.HardwareLimitSwitch.ForwardLimitRemoteSensorID =␣
↪m_cancoder.GetDeviceID();
// set up other configs here

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

#### Python

```
configs = configs.TalonFXConfiguration()
configs.hardware_limit_switch.forward_limit_source = signals.
↪ForwardLimitSourceValue.REMOTE_CANCODER
configs.hardware_limit_switch.forward_limit_remote_sensor_id = self.cancoder.
↪device_id
# set up other configs here

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

## 18.10 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);
```

(continues on next page)

(continued from previous page)

```
// 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()
```

### 18.10.1 Playback While Disabled (FRC)

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

# 19 Device API

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

## 19.1 Talon FX® / Talon FXS®

### 19.1.1 Introduction to Control Types

Phoenix 6 motor controllers support a variety of *open-loop* and *closed-loop* control requests. Additionally, the TalonFX supports *Field Oriented Control (FOC)*.

#### Control Output Types

The Talon FXS® currently supports two base control output types: DutyCycle and Voltage. The Talon FX® additionally supports 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*. Additionally, the Talon FXS® does not support TorqueCurrentFOC at this time.

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*. Additionally, the Talon FXS® does not support FOC at this time.

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*

### 19.1.2 Open-Loop Control

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

There are open-loop control requests for all Talon FX® and Talon FXS® 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

```java
// initialize devices on the rio can bus
final CANBus kCANBus = CANBus.roboRIO();
final TalonFX m_leftLeader = new TalonFX(0, kCANBus);
final TalonFX m_rightLeader = new TalonFX(1, kCANBus);

// 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++

```cpp
// initialize devices on the rio can bus
static constexpr CANBus kCANBus = CANBus::RoboRIO();
hardware::TalonFX m_leftLeader{0, kCANBus};
hardware::TalonFX m_rightLeader{1, kCANBus};

// 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.canbus = CANBus.roborio()
self.left_leader = hardware.TalonFX(0, self.canbus)
self.right_leader = hardware.TalonFX(1, self.canbus)

# 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))
```

### 19.1.3 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 all gains to zero.
2. Determine $K_g$ if using an *elevator* or *arm*.
3. Select the appropriate *Static Feedforward Sign* for your closed-loop type.
4. Increase $K_s$ until just before the motor moves.
5. If using velocity setpoints, increase $K_v$ until the output velocity closely matches the velocity setpoints.
6. Increase $K_p$ until the output starts to oscillate around the setpoint.
7. Increase $K_d$ as much as possible without introducing jittering to the response.

All closed-loop control requests follow the naming pattern `{ClosedLoopMode}{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 Talon FX® and Talon FXS® support 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® and Talon FXS® often require 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® or Talon FXS® rotor.

Some arm mechanisms have a center of gravity that is offset from the zero position. The `GravityArmPositionOffset` config can be adjusted to account for this offset within ±0.25 rotations.

### Static Feedforward Sign

The static feedforward $K_s$ is the output needed to overcome the system's static friction, in units of the *control output type*. Because friction always opposes the direction of motion, the sign of $K_s$ also depends on the direction of motion. Phoenix 6 provides two possible methods of determining this signage using the `StaticFeedforwardSign` config in the gain slots.

### Velocity Sign

By default, the signage of $K_s$ is determined by the signage of the velocity setpoint. In other words, if the velocity setpoint is positive, then the output of $K_s$ is positive; if the velocity setpoint is negative, then $K_s$ is negative. This option is always used when running velocity closed loops, and it is recommended for Motion Magic® controls and motion-profiled position closed loops.

### Closed-Loop Sign

When using a position closed-loop controller, signage of $K_s$ can instead be determined by the sign of the closed-loop error. For example, if the position error (target - measured) is positive, then the output of $K_s$ is positive; if the error is negative, then $K_s$ is negative. This option is typically used when a velocity setpoint is otherwise not available, such as when running unprofiled position closed loops.

**Important**

When using the sign of closed-loop error for $K_s$, it is important that the selected $K_s$ value is not too large. Otherwise, the motor output may dither or oscillate when near the closed-loop target.

### 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).

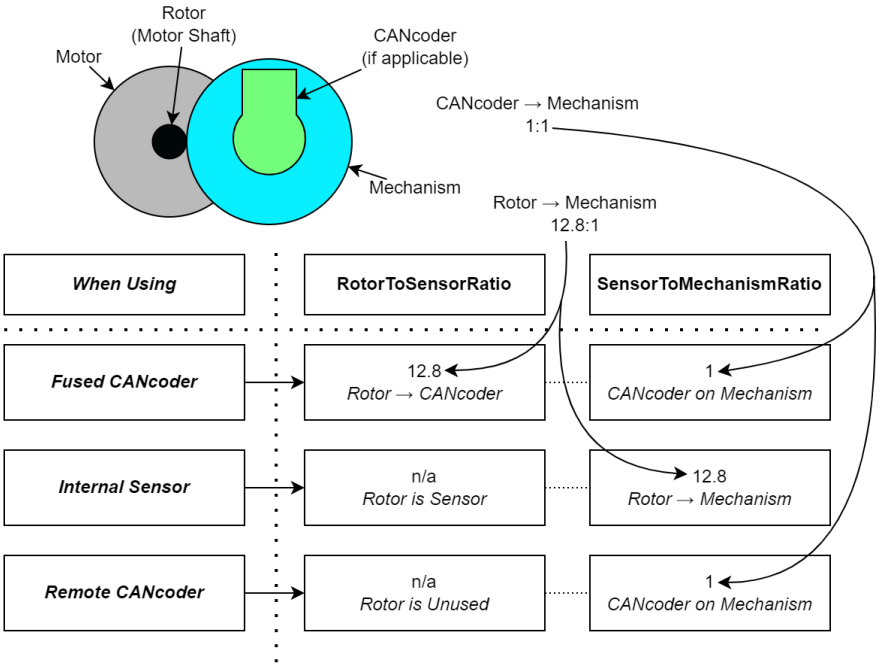
`ContinuousWrap` (Java, C++, Python) is a mode of closed loop operation that enables the Talon to take the "shortest path" to a target position for a continuous mechanism. It does this by assuming that the mechanism is continuous within 1 rotation.

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.



### 19.1.4 Basic PID and Profiling

The Talon FX® and Talon FXS® support basic PID control and motion profiling for position and velocity.

> **Note**
>
> For more information on feedback and feedforward gains, see *Closed-Loop Overview*.

#### 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:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - Velocity Sign: unused; Closed-Loop Sign: output to overcome static friction (output)
- $K_v$ - 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

```
var configs = new TalonFXConfiguration();
// in init function, set slot 0 gains
configs.Slot0.kP = 2.4; // An error of 1 rotation results in 2.4 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0.1; // A velocity of 1 rps results in 0.1 V output

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

#### C++

```
configs::TalonFXConfiguration configs{};
// in init function, set slot 0 gains
configs.Slot0.kP = 2.4; // An error of 1 rotation results in 2.4 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0.1; // A velocity of 1 rps results in 0.1 V output

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

#### Python

```
configs = configs.TalonFXConfiguration()
# in init function, set slot 0 gains
configs.slot0.k_p = 2.4 # An error of 1 rotation results in 2.4 V output
configs.slot0.k_i = 0 # no output for integrated error
configs.slot0.k_d = 0.1 # A velocity of 1 rps results in 0.1 V output

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

Once the gains are configured, the Position closed loop control request can be sent to the motor controller. 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:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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

```
var configs = new TalonFXConfiguration();
// in init function, set slot 0 gains
configs.Slot0.kS = 0.1; // Add 0.1 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kP = 0.11; // An error of 1 rps results in 0.11 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0; // no output for error derivative

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

### C++

```
configs::TalonFXConfiguration configs{};
// in init function, set slot 0 gains
configs.Slot0.kS = 0.1; // Add 0.1 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kP = 0.11; // An error of 1 rps results in 0.11 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0; // no output for error derivative

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

### Python

```
configs = configs.TalonFXConfiguration()
# in init function, set slot 0 gains
configs.slot0.k_s = 0.1 # Add 0.1 V output to overcome static friction
configs.slot0.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
configs.slot0.k_p = 0.11 # An error of 1 rps results in 0.11 V output
configs.slot0.k_i = 0 # no output for integrated error
configs.slot0.k_d = 0 # no output for error derivative

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

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

#### Java

```java
// create a velocity closed-loop request, voltage output, slot 0 configs
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++

```cpp
// 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

```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® and Talon FXS® support several onboard motion profiles using *Motion Magic®*.

### Position

In a Position motion profile, the gains should be configured as follows:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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

```
var configs = new TalonFXConfiguration();
// in init function, set slot 0 gains
configs.Slot0.kS = 0.25; // Add 0.25 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kP = 4.8; // A position error of 2.5 rotations results in 12 V␣
↪output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0.1; // A velocity error of 1 rps results in 0.1 V output

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

### C++

```
configs::TalonFXConfiguration configs{};
// in init function, set slot 0 gains
configs.Slot0.kS = 0.25; // Add 0.25 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kP = 4.8; // A position error of 2.5 rotations results in 12 V␣
↪output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0.1; // A velocity error of 1 rps results in 0.1 V output

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

### Python

```
configs = configs.TalonFXConfiguration()
# in init function, set slot 0 gains
configs.slot0.k_s = 0.25 # Add 0.25 V output to overcome static friction
configs.slot0.k_v = 0.12 # A velocity target of 1 rps results in 0.12 V output
configs.slot0.k_p = 4.8 # A position error of 2.5 rotations results in 12 V␣
↪output
configs.slot0.k_i = 0 # no output for integrated error
configs.slot0.k_d = 0.1 # A velocity error of 1 rps results in 0.1 V output

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

Once the gains are configured, the Position closed-loop control request can be sent to the motor controller. 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
```

(continues on next page)

(continued from previous page)

```
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:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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

```
var configs = new TalonFXConfiguration();
// in init function, set slot 0 gains
configs.Slot0.kS = 0.25; // Add 0.25 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
configs.Slot0.kP = 0.11; // An error of 1 rps results in 0.11 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0; // no output for error derivative

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

### C++

```
configs::TalonFXConfiguration configs{};
// in init function, set slot 0 gains
configs.Slot0.kS = 0.25; // Add 0.25 V output to overcome static friction
configs.Slot0.kV = 0.12; // A velocity target of 1 rps results in 0.12 V␣
↪output
configs.Slot0.kA = 0.01; // An acceleration of 1 rps/s requires 0.01 V output
configs.Slot0.kP = 0.11; // An error of 1 rps results in 0.11 V output
configs.Slot0.kI = 0; // no output for integrated error
configs.Slot0.kD = 0; // no output for error derivative
```

(continues on next page)

(continued from previous page)

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

### 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 motor controller. 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,␣
```

(continues on next page)

(continued from previous page)

```
↪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()

# 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)
```

### 19.1.5 Motion Magic® Controls

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

> **Note**
>
> For more information on feedback and feedforward gains, see *Closed-Loop Overview*.

### 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:

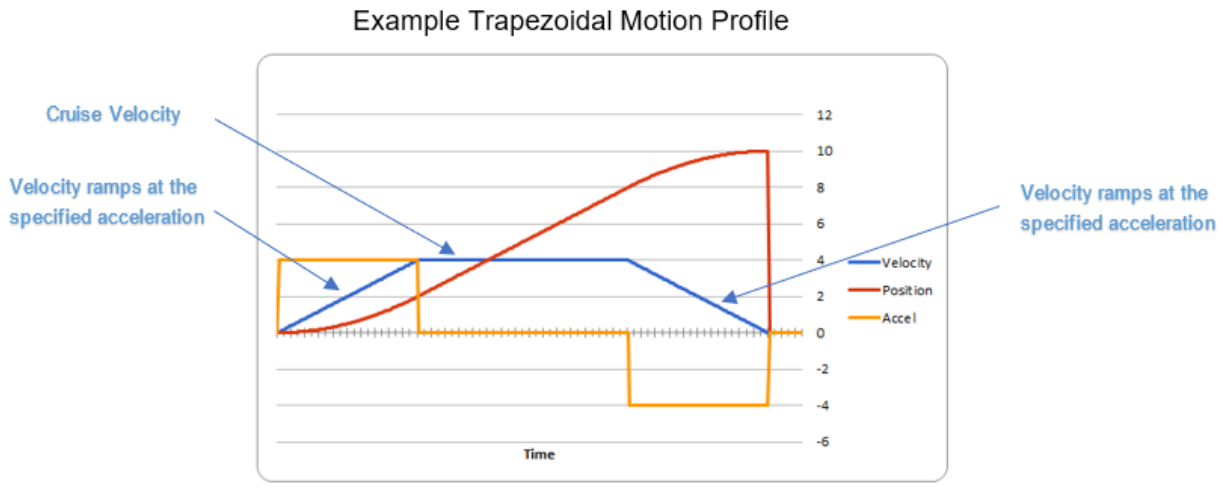
- Control of the mechanism throughout the entire motion (as opposed to racing to the end target position)
- Control of the mechanism's inertia to ensure smooth transitions between setpoints
- Improved repeatability despite changes in battery load
- Improved repeatability despite changes in motor load

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".



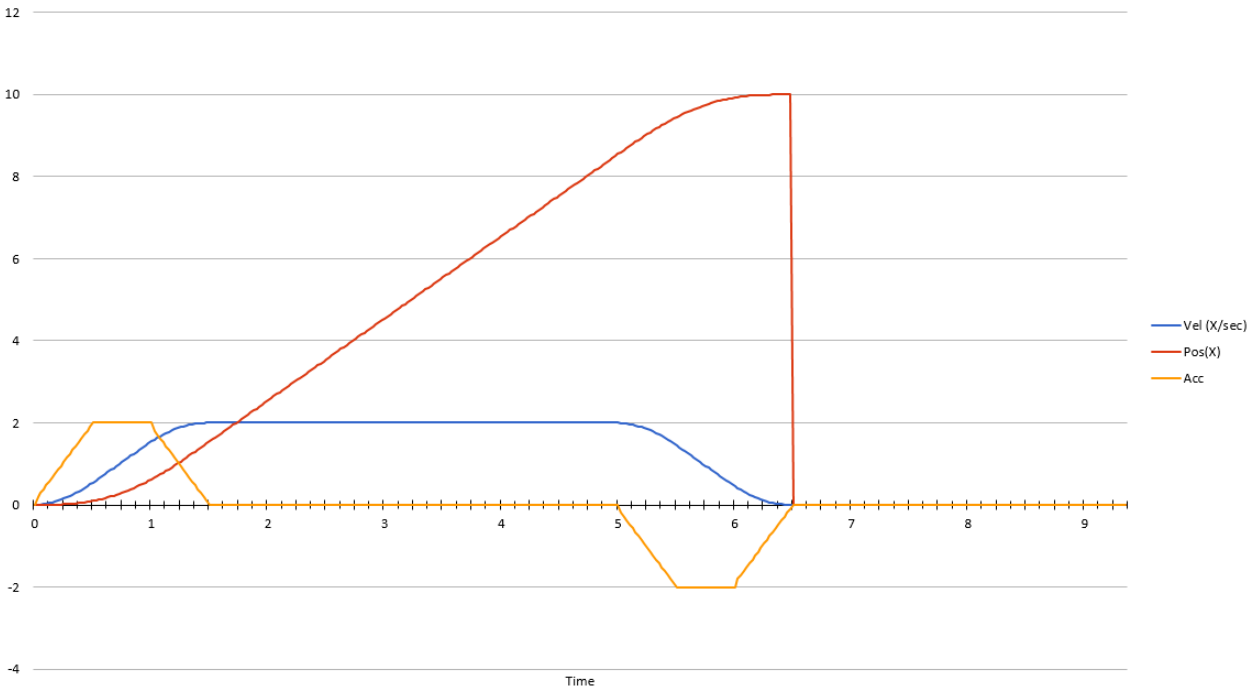

If the Motion Magic® jerk is set to a nonzero value, the generated velocity profile is no longer trapezoidal, but instead is a continuous S-Curve (corner points are smoothed).

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

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

**Note**

The jerk control feature, by its nature, will increase the amount of time a movement requires. This can be compensated for by increasing the configured acceleration value.




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

- Cruise Velocity - peak/cruising velocity of the motion
- Acceleration - controls acceleration and deceleration rates during the beginning and end of motion
- Jerk (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:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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)
```

(continues on next page)

(continued from previous page)

```
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 motor controller. 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 motor controller will disable control output and trip the UnlicensedFeatureInUse fault.

When using a Pro-licensed Talon FX® or Talon FXS® 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 motor controller. 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).withJerk(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
```

(continues on next page)

(continued from previous page)

```
   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 =
   controls::DynamicMotionMagicVoltage{0_tr, 80_tps, 400_tr_per_s_sq}
      .WithJerk(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
```

(continues on next page)

(continued from previous page)

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

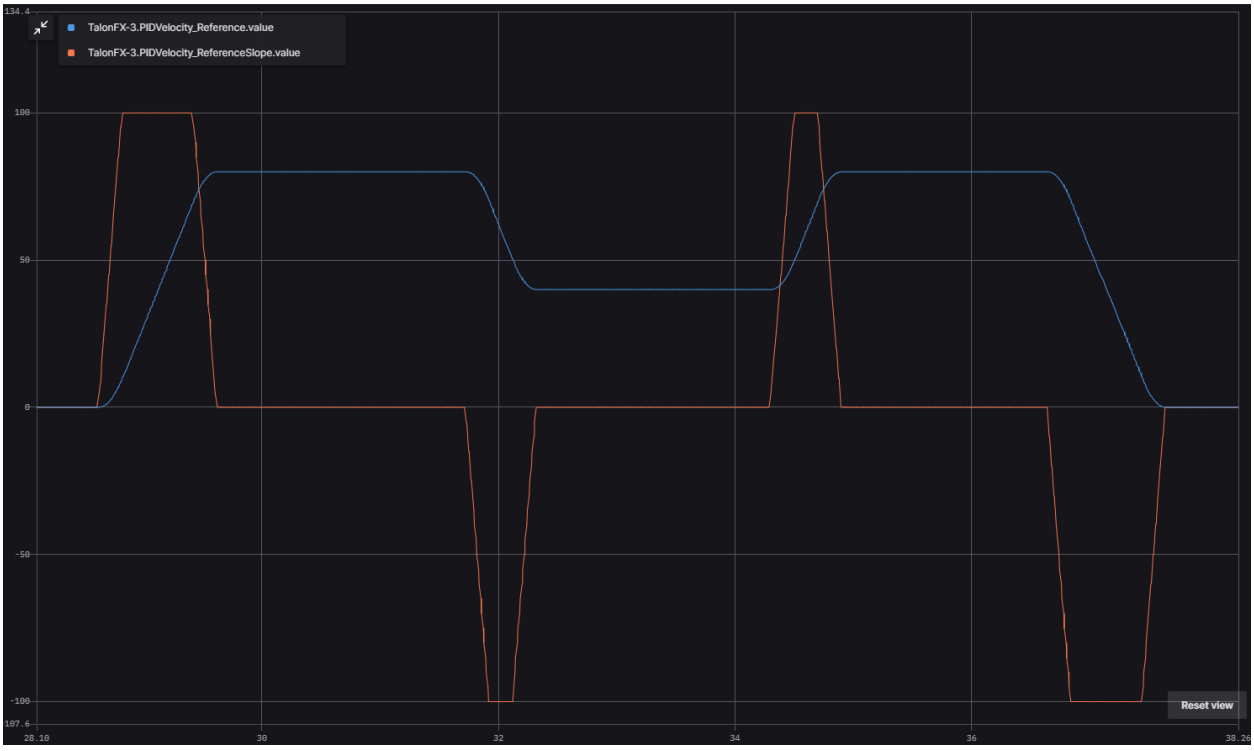
### Motion Magic® Velocity

The Talon FX® and Talon FXS® also support 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 `MotionMagicConfigs` (Java, C++, Python) object.

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

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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
```

(continues on next page)

(continued from previous page)

```
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 motor controller.

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

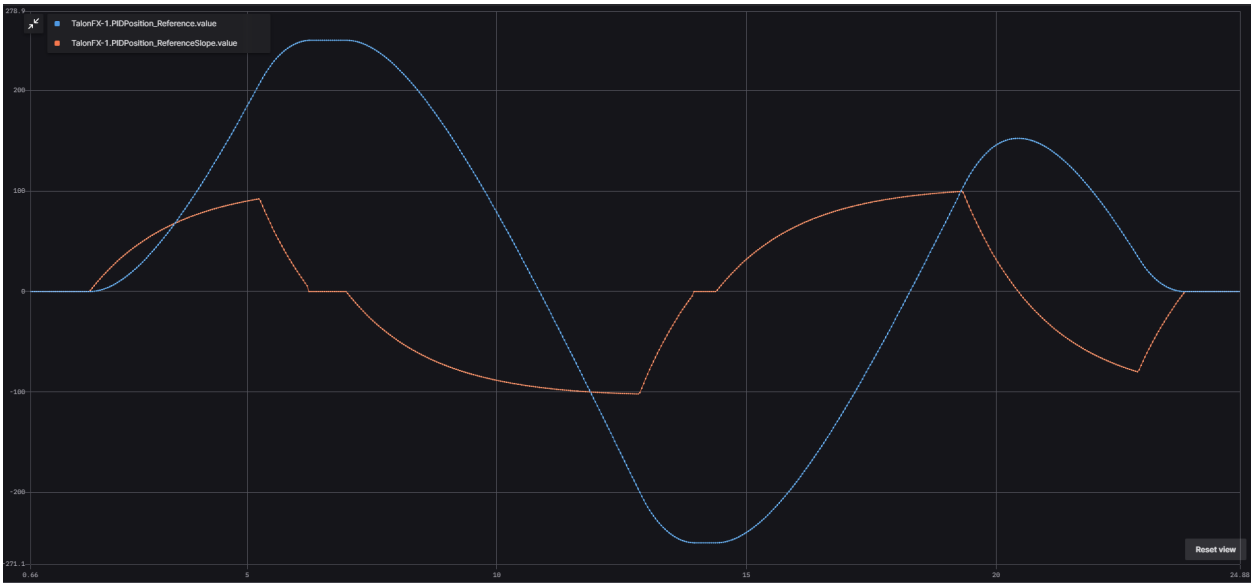
```
# create a Motion Magic Velocity request, voltage output
self.request = controls.MotionMagicVelocityVoltage(0)

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:

- $K_g$ - output to overcome gravity (output)
- $K_s$ - 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 motor controller. 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))
```

### Dynamic Motion Magic® Expo

> **Important**
>
> This feature requires the device to be *Pro licensed* and on a *CANivore®*. When unlicensed, the motor controller will disable control output and trip the UnlicensedFeatureInUse fault.

When using a Pro-licensed Talon FX® or Talon FXS® connected to a CANivore®, Dynamic Motion Magic® Expo can be used, allowing for the cruise velocity, Expo kV, and Expo kA to be modified directly in the control request during motion. This can be used to set up different values for forward vs reverse 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® Expo request. However, the cruise velocity, Expo kV, and Expo kA parameters are set up in the control request, **not** the Motion Magic® config group.

Once the gains are configured, the Dynamic Motion Magic® Expo request can be sent to the motor controller. 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 Expo request, voltage output
// default Expo kV of 0.12 V/rps and kA of 0.1 V/(rot/s^2), unlimited cruise␣
↪velocity
final DynamicMotionMagicExpoVoltage m_request =
   new DynamicMotionMagicExpoVoltage(0, 0.12, 0.1);

if (m_joy.getAButton()) {
   // while the joystick A button is held, use a slower profile
   // cap the cruise velocity and weaken acceleration (larger kA)
   m_request.Velocity = 40; // rps
   m_request.kA = 0.2; // V/(rot/s^2)
} else {
   // otherwise use a faster profile
   m_request.Velocity = 0; // rps, 0 is unlimited
   m_request.kA = 0.1; // V/(rot/s^2)
}

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

### C++

```
// create a Dynamic Motion Magic Expo request, voltage output
// default Expo kV of 0.12 V/rps and kA of 0.1 V/(rot/s^2), unlimited cruise␣
↪velocity
controls::DynamicMotionMagicExpoVoltage m_request{
   0_tr, 0.12_V/1_tr_per_s, 0.1_V/1_tr_per_s_sq
};

if (m_joy.GetAButton()) {
   // while the joystick A button is held, use a slower profile
   // cap the cruise velocity and weaken acceleration (larger kA)
   m_request.Velocity = 40_tps;
   m_request.kA = 0.2_V/1_tr_per_s_sq;
} else {
   // otherwise use a faster profile
   m_request.Velocity = 0_tps; // 0 is unlimited
   m_request.kA = 0.1_V/1_tr_per_s_sq;
}

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

#### Python

```
# create a Dynamic Motion Magic Expo request, voltage output
# default Expo kV of 0.12 V/rps and kA of 0.1 V/(rot/s^2), unlimited cruise␣
↪velocity
self.request = controls.DynamicMotionMagicExpoVoltage(0, 0.12, 0.1)

if self.joy.getAButton():
   # while the joystick A button is held, use a slower profile
   # cap the cruise velocity and weaken acceleration (larger kA)
   self.request.velocity = 40 # rps
   self.request.k_a = 0.2 # V/(rot/s^2)
else:
   # otherwise use a faster profile
   self.request.velocity = 0 # rps, 0 is unlimited
   self.request.k_a = 0.1 # V/(rot/s^2)

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

### 19.1.6 Manually Tuning PID with TorqueCurrentFOC

There are plenty of tools available that allow users to characterize their mechanism and find the ideal gains, such as the WPILib System Identification tool. However, many of these tools require an understanding of how to manually tune a mechanism, and manual tuning can sometimes be faster and easier. This guide covers how to tune Phoenix 6 PID control manually with a focus on TorqueCurrentFOC.

#### General Information

Every closed loop controller has the following aspects consistent:

1. Gains are canonical
   - $k_G = \frac{\text{output}}{\cos(\text{angle})}$ **for arm;** $k_G = \text{output}$ **for elevator**
     - The amount of output to apply to counteract the force of gravity.
   - $k_S = \text{output}$
     - A static or constant amount of output to apply, typically used to overcome friction.
   - $k_V = \frac{\text{output}}{\text{vel}}$
     - The amount of output to apply per target velocity. In Voltage control modes, this is primarily a feed forward to counteract the back-emf of the motor. In Torque control modes, this is a feed forward to counteract the force of drag (or viscous friction) on the system.
   - $k_A = \frac{\text{output}}{\text{accel}}$
     - The amount of output to apply per target acceleration. This is used to account for the inertia of a system.
   - $k_P = \frac{\text{output}}{\text{error}}$
     - The amount of output to apply per unit of error in the system.

- $k_I = \frac{\text{output}}{\text{error} \cdot \text{s}}$
  - The amount of output to apply per unit of error for every second of that error.
- $k_D = \frac{\text{output}}{\text{error/s}} = \frac{\text{output} \cdot \text{s}}{\text{error}}$
  - The amount of output to apply per unit of change in error over time.

2. Motor output is dependent on control type
   - See *Control Output Types* and *Choosing Output Type*
3. Everything operates in mechanism units
   - This generally multiplies the gains by the gear reduction
   - A 100:1 reduction means a $k_P = 1$ is 1 Volt output at 1 rotation error at the mechanism, or 1 Volt output at 100 rotations error at the motor.

As a result, the first major aspect of manual PID tuning is finding a good start. Since every gain is canonical, users can back-calculate what value the gain should start at based on the error the mechanism sees and the desired motor voltage. For example, consider an arm mechanism that is currently 0.1 rotations away from its target. If 1 volt of output is enough to meaningfully move it, at 0.1 rotation error the arm can apply 1 Volt to slowly bring it to the setpoint. That results in a kP of $k_P = \frac{\text{Volts}}{\text{error}} = \frac{1 \text{ V}}{0.1 \text{ rot}} = 10 \text{ V/rot}$. This is a good starting point for the arm that results in a safe response for further tuning.

A similar process can be used for every other gain constant to find good starting points.

### Specific Response Tuning

The specific mechanism being tuned is going to affect the starting points for the gains and the process of finding the ideal gains.

### Flywheel Tuning with TorqueCurrentFOC

Below is a list of steps and a simulator that provides the opportunity to try tuning a flywheel system using PID with TorqueCurrentFOC. The Red line is the setpoint of the flywheel controller, purple is the current velocity, and the green line is the torque current of the motor in amps.

This particular flywheel has a maximum velocity of 100 rps and a 1:1 gear ratio.

Tuning a flywheel is largely done with the following steps:

1. Zero all PID gains.
2. Set a low setpoint (typically 10% of the maximum velocity).
3. Set kP to a very low number (typically 1 * `RotorToSensorRatio` * `SensorToMechanismRatio` is a good starting point).
4. Adjust kS until flywheel achieves setpoint.
5. Set a high setpoint (typically 80% of the maximum velocity).
6. Adjust kV until flywheel achieves setpoint. Note that for some velocity systems, this may occur at kV = 0 (as is the case with swerve drive motors).
7. Go back to the low setpoint.
8. Repeat steps 4-7 until the gains do not change.

9. Go back to the high setpoint. Increase kP until the flywheel oscillates, then back off to just before that oscillation.
10. Verify gains hold for expected velocities.

### "Why" for each step

1. Starting with PID gains at 0 isolates as many of the forces in play as possible to iteratively get closer to the "ideal" gains.
2. There needs to be a setpoint for the parameters to take affect, and a lower setpoint sets up the following steps.
3. A low kP will magnify the requirement for a good kS and kV. If this gain is too high, it will mask a "bad" kS and kV during the kS/kV tuning.
4. The kS gain is meant to reduce the effect of friction without it moving the mechanism on its own. However, tuning kS by simply finding the point at which the mechanism moves only accounts for static friction, which is not ideal in a flywheel that typically experiences kinetic friction. With the low setpoint, the kS term will dwarf the kV term, so kS can be tuned to correctly account for kinetic friction.
5. Setting a high setpoint will prioritize the kV term during its tuning phase.
6. With the high setpoint, the kV output will dwarf the kS term in its effectiveness, so the kV term should be prioritized to achieve the setpoint.
7. Going back to a low setpoint verifies the kS term is still correct.
8. Typically, after reducing kS the system will not achieve its high setpoint, so the kV term needs to increase. Similarly, increasing kV may cause the system to overshoot the low setpoint, requiring the kS to lower. This procedure continues until the kS/kV gains stabilize and stop changing, indicating the feed forwards are correct.
9. With proper kS/kV terms, the kP can be increased to quickly achieve the setpoint. The system wants as high a kP gain as possible to decrease the time taken to get to the setpoint. The limit of how high the kP term can be is determined by the system latency, at which point the oscillation is impossible to avoid.
10. Always verify the gains work for the setpoints you expect the system to be commanded, as it is possible the generic gains may not work under the operating range of the system. If that is the case, adjust the setpoints to be within the expected operating range and re-tune with them.

The simulator below allows you to follow these steps to find the right gains.

> **Note**
>
> The flywheel will react to the ball getting launched at 5 seconds.

### Tuning Process Example

Following the guide, I start with all gains set to 0 and set a setpoint of 10 rps (100 rps maximum). The gear ratios are 1, so I set the kP to 1 amp output per rps error and notice that the wheel starts moving up to the setpoint, but it can't quite reach it, so I begin playing with the kS parameter.

Setting kS to 1 significantly undershoots, so I double it to 2, which still undershoots. Doubling it to 4 gets it pretty close, and 5 overshoots, so I leave the kS at 4 and move on to the next step.

Now I set a setpoint of 80 rps and notice that the wheel stalls out at 65-70 rps. This means the drag is significant and preventing us from reaching the setpoint, necessitating a kV. I set kV to 1, and notice that it significantly overshoots. I halve it to 0.5, 0.25, then 0.125 before I notice it just barely makes it to the target. Increasing it to 0.13 is almost perfect, so I leave it at 0.13.

Then, I set the setpoint to 10 and notice that I'm overshooting. This means I need to decrease the kS gain. I try 3 and notice that it still overshoots. So I bring it down to 2.5, then up to 2.8 and see it's pretty much perfect.

Going back to 80 rps, I'm now undershooting, so I increase kV to 0.14, then 0.15 before I'm happy with it.

Back down to 10 rps, I'm overshooting again, so I bring kS down to 2.6, at which point I'm happy with it reaching the target.

Again up at 80 rps, I'm now generally happy with how close it's reaching the target, so I move on to increasing kP.

I first double kP to 2, then 5, 10, and 20, noticing that the time to target is decreasing with a larger kP. A kP of 20 results in a bit of overshoot that I don't like, so I decrease it to 18, 17, then 16 before it matches what I want, so I leave it at 16.

This gives final gains of kS = 2.6 A, kV = 0.15 A/rps, and kP = 16 A/rps.

And that's the flywheel tuned! This took 3 iterations of going between low setpoint and high setpoint, but sometimes you may need more depending on how difficult your system's dynamics are and if you need tighter tolerances. In this case I'm eyeballing the response and saying it's good enough, but in practice you should use the closed loop error Status Signal to verify the error is within the tolerance of your mechanism.

### Turret Tuning with TorqueCurrentFOC

Tuning a Turret is identical to any other position controller that has no gravity component.

One key thing to note with any position-based torque controller is the reliance on the kD term. When tuning a position controller with voltage, it's often enough to rely on the natural dampening of the system to dampen the response with a weak enough kP, negating some of the need for kD. However when using torque as the control type, most of that natural dampening is gone, so kD is always necessary for the system to stop itself in any reasonable amount of time.

Similar to the velocity controller, below is a list of steps and simulator for turret tuning. Red is the setpoint in rotations, purple is the current position, and green is the torque current in amps. This particular turret has a 20:1 gear reduction, with the `RotorToSensorRatio` and `SensorToMechanismRatio` configs set up accordingly.

The following steps cover the general idea:

1. Zero all PID gains.
2. Set a setpoint relatively nearby (such as 0.1 mechanism rotations).
3. Increase kS until the turret starts moving, then back off to just before that movement.
4. Increase kP until you notice significant overshoot.
5. Increase kD until the overshoot stops happening.

6. Repeat steps 4 and 5 until increasing kD results in more oscillation, or until the system oscillates on its way to the setpoint. If oscillation on the way to setpoint is seen, decrease kD until it stops. Then reduce kP until any remaining overshoot stops. The goal is to maximize kP and kD to minimize the travel time.
7. Verify gains work for other setpoints as well. Tune kP/kD as appropriate for most general cases.

> **Tip**
>
> Values of kP=2000, kD=150 demonstrate the "oscillates on its way to the setpoint" case for setpoints within 1 rotation.

### "Why" for each step

1. Starting with PID gains at 0 isolates as many of the forces in play as possible to iteratively get closer to the "ideal" gains.
2. A nearby setpoint ensures the system response should be relatively small to start with when tuning.
3. The kS gain is meant to reduce the effect of friction, so the largest possible value that still prevents the system from moving will reduce the effect of friction in general.
4. The kP gain will control how quickly the system gets to the setpoint. However, in Torque-CurrentFOC modes there is no natural dampening force, so overshoot and oscillation is expected at the beginning. At that point, kD should be tuned.
5. The kD gain will effectively slow down the system as it reaches the setpoint. Increasing it will increase the force slowing it down, so it should be increased until the system no longer overshoots.
6. In general, the system wants as high a kP gain as possible to decrease the time taken to get to the setpoint. This also requires a high kD gain to properly dampen the system. The limit of how high the kP/kD term can be is determined by the system latency, at which point the oscillation is impossible to avoid. The goal of repeating steps 4 and 5 is to find that limit.
7. Always verify the gains work for the expected setpoints of the system; it's possible the general solution may not work under the expected operating range of the system. If that's the case, re-tune for the expected operating range using the generic gains as a basis.

### Tuning Process Example

Following the guide, I start with all gains at 0 and set a setpoint of 0.1 rotations.

I start with a kS of 1 amp and notice it still does not move, so I double it to 2 amps and notice it now moves. Then, I reduce it to 1.5, then 1.25 until it stops. Increasing to 1.3 gets the turret moving again, so I leave it at 1.25 amps.

I then set a kP of 1, then 10 and see significant overshoot, so I add a kD of 1, then 10. This is very overdamped system, but that's fine, as I'll start increasing kP again.

I increase kP to 20 and see no overshoot. Increase again to 50, then 100, and I see a little overshoot. Increase again to 200 and I see significant overshoot, indicating I should increase kD again. I double it to 20 and the overshoot becomes minimal, but then I increase it again to 40 before it becomes significantly overdamped again.

Increasing kP again to 500 still looks fine, to 1000 is still fine, 2000 finally has significant overshoot. I increase kD to 60 and that overshoot is gone.

So I increase kP again to 3000, where I notice it oscillates a bit. I try to stop this oscillation by increasing kD to 80, then to 100 where I notice it's always oscillating. This means I've reached the limit of the system, and need to back off on gains a bit.

I reduce kD back to 80 where I notice a bit of oscillation on its way to the setpoint, then 70, when that oscillation goes away, and start dialing back kP. I start with a kP of 2800, where it's overdamped and oscillating on its way to the setpoint, so I reduce kD to 65.

From here I continue to reduce kP to 2600, then 2500, then 2400, and the system response looks relatively good at this point. Now it's time to play with different setpoint. When I set a setpoint of 0.45, I notice significant overshoot that I should correct in PID.

At this point, I know that my kD can't go much higher, otherwise I have oscillation on my way to the setpoint at smaller setpoints. So I try to stop the oscillation only with kP. Reducing it to 2200, 2000, then finally 1800 before the overshoot stops, but now it looks overdamped.

So I reduce kD to 60, and I have a bit of overshoot, so I reduce kP to 1700, then 1650 which looks good. Back to a setpoint of 0.1 and I still have some overdamped behavior, but it's minimal at this point and what I'd consider acceptable.

This gives final gains of kS = 1.25 A, kP = 1650 A/rot, and kD = 60 A/rps.

If my system normally expects setpoints within a rotation of my current position, then I'd prioritize the within-1-rotation situation for my PID controller. However, if my system normally expects setpoints closer to 20 rotations away from current position then I'd prioritize that situation. If I really needed both close and far away behavior, then I'd look at gain-scheduling based on the value of the error, using both Slots 0 and 1, with 0 for the within-1-rotation situation, and 1 for the outside-1-rotation situation.

### Arm Tuning with TorqueCurrentFOC

Tuning an Arm is very similar to tuning a turret, just with the addition of needing to account for gravity. As such, the process is nearly identical, except for a small section dedicated to dialing in the kG term.

As with the previous example, red is the setpoint in rotations, purple is the current position, and green is the torque current in amps. This particular arm has a 35:1 gear reduction, with the `RotorToSensorRatio` and `SensorToMechanismRatio` configs set up accordingly.

The steps:

1. Zero all PID gains.
2. Increase kG and find the smallest possible kG that stops the arm from moving.
3. Increase kG and find the largest possible kG that stops the arm from moving.
4. Set kG to the middle of the two values.
5. Set kS to half the difference between the upper and lower values.
6. Set a setpoint relatively nearby (typically 0.1 mechanism rotations).
7. Increase kP until you notice significant overshoot.
8. Increase kD until the overshoot stops happening.
9. Repeat steps 7 and 8 until increasing kD results in more oscillation, or until the system oscillates on its way to the setpoint. If oscillation on the way to setpoint is seen, decrease

kD until it stops. Then reduce kP until any remaining overshoot stops. The goal is to maximize kP and kD to minimize the travel time.

10. Verify gains work for other setpoints as well. Tune kP/kD as appropriate for most general cases.

> **Tip**
>
> Values of kP=5000, kD=800 demonstrate the "oscillates on its way to the setpoint" case for setpoints within 1 rotation.

### "Why" for each step

1. Starting with PID gains at 0 isolates as many of the forces in play as possible to iteratively get closer to the "ideal" gains.
2. The kG gain is meant to counteract the force of gravity; however, the force of friction is also at play in an arm. The lowest possible kG that prevents the system from moving is the lower bound of the gravity and friction component (kG - kS).
3. The highest possible kG that prevents the system from moving is the upper bound of the gravity and friction component (kG + kS).
4. Setting kG to the middle point of the upper and lower bounds is a good approximation for the true effect of gravity. ((kG + kS) + (kG - kS)) / 2 = kG.
5. Setting kS to half the difference of the upper and lower bounds is a good approximation for the true effect of friction. ((kG + kS) - (kG - kS)) / 2 = kS.
6. A nearby setpoint ensures the system response should be relatively small to start with when tuning.
7. The kP gain will control how quickly the system gets to the setpoint. However, in TorqueCurrentFOC modes there is no natural dampening force, so overshoot and oscillation is expected at the beginning. At that point, kD should be tuned.
8. The kD gain will effectively slow down the system as it reaches the setpoint. Increasing it will increase the force slowing it down, so it should be increased until the system no longer overshoots.
9. In general, the system wants as high a kP gain as possible to decrease the time taken to get to the setpoint. This also requires a high kD gain to properly dampen the system. The limit of how high the kP/kD term can be is determined by the system latency, at which point the oscillation is impossible to avoid. The goal of repeating steps 7 and 8 is to find that limit.
10. Always verify the gains work for the expected setpoints of the system; it's possible the general solution may not work under the expected operating range of the system. If that's the case, re-tune for the expected operating range using the generic gains as a basis.

### Tuning Process Example

Following the guide, I start with all gains at 0 to dial in kG and kS.

I start with a kG of 1, and notice that the arm's still falling, so I increase it to 2, 4, 8, 16, and 20 before it stops moving. From there I reduce it to 18, then 16 and notice it fall again. I

bring it up to 17 and see it still falls appreciably, but 17.5 does not, so I use 17.5 for the lower bound.

Going back up, I start at 20 again, then to 22, 24, and 26 before it moves its way up. 25 also produces appreciable movement, but 24.5 does not. This means my kG is (24.5 + 17.5) / 2 = 21 amps, and my kS is (24.5 - 17.5) / 2 = 3.5 amps.

From here, I set a setpoint of 0.1, and now it's time for kP/kD tuning. I bring kP up to 1, 2, 4, 8, and 16 before I get significant overshoot, where I dial kD in to 1, 2, 4, and 8 before that overshoot is gone. kP keeps increasing to 32 and 64, then kD goes up to 16 and 32, then kP up to 128, 250, and 500, then kD to 64 and 128 before it's back to kP. I go up to 1000 and 2000 where I notice a bit of oscillation, and I may be near the limit at this point. kD increases to 250, then kP to 4000 and 8000, then kD to 500 and I get oscillation on the way to the target.

At this point, I have reached the limit of kD, so I bring it down to 400 then 300 before I'm happy with it.

I check with a setpoint of 0.6 and notice there is severe overshoot, but I cannot increase kD any further, so I reduce kP to 4000, then 3700 at which point I am happy with the behavior. I check with other setpoints of -0.1 and 0.4 and confirm the movement looks good, so the PID tuning is done.

This gives final gains of kG = 21 A, kS = 3.5 A, kP = 3700 A/rot, and kD = 300 A/rps.

If my arm normally expects small movements or has a small range of motion, then I'd prioritize the smaller setpoints for my PID controller. However, if my arm frequently drives long distances (such as going to the other side of the robot), then I'd prioritize that situation. If I really needed both close and far away behavior, then I'd look at gain-scheduling based on the value of the error, using both Slots 0 and 1, with 0 for the small distance situation, and 1 for the large distance situation.

### Profiled Tuning

Profiled tuning can be treated much the same way as tuning a normal PID, but the introduction of a profile means much of the response can be calculated in advance with feed-forwards. This results in much of the work being done due to feed forward, with the feedback gains being used to account for any error in the system.

In Phoenix 6, you can either generate your own profile and feed in the position and velocity setpoints, or use Motion Magic® and let the Talon generate the profile for you. In either case the Talon will have Velocity and/or Acceleration setpoints that it use for the kV and kA feedforward terms, resulting in more accurate profile following.

Red is the position setpoint in rotations, blue is the velocity setpoint in rps, yellow is the acceleration setpoint in rot/s$^2$, purple is the current position, brown is the current velocity, orange is the current acceleration, and green is the torque current in amps. This particular mechanism has a 1:1 gear ratio with high inertia.

The example below uses a pre-generated profile for the system to follow, and the general steps to tune it are below:

1. Zero all PID gains.
2. Set a setpoint relatively nearby (typically 0.1 mechanism rotations). This isn't relevant for the simulation, as the setpoint is determined by the pre-generated profile.
3. Increase kS until the system starts moving, then back off to just before that movement.
4. Increase kA until the slope of the measured velocity matches the slope of the profiled velocity at the beginning. In the simulation, you can also look at the measured and

profiled acceleration.

5. Increase kV until the measured velocity matches the profiled velocity towards the end of the "cruise" time (constant velocity setpoint). For many mechanisms, the kV can be left 0.
6. Increase kP until you notice significant overshoot or oscillation (even during motion at cruise velocity).
7. Increase kD until the overshoot/oscillation stops happening.
8. Repeat steps 6 and 7 until increasing kD results in more oscillation, or until the system oscillates on its way to the setpoint. If oscillation on the way to setpoint is seen, decrease kD until it stops. Then reduce kP until any remaining overshoot stops. The goal is to maximize kP and kD to minimize the travel time.

**Tip**

Values of kP=0, kD=6000 demonstrate the "oscillates on its way to the setpoint" case.

### "Why" for each step

1. Starting with PID gains at 0 isolates as many of the forces in play as possible to iteratively get closer to the "ideal" gains.
2. A nearby setpoint ensures the system response should be relatively small to start with when tuning.
3. The kS gain is meant to reduce the effect of friction, so the largest possible value that still prevents the system from moving will reduce the effect of friction in general.
4. The kA gain effectively accounts for the inertia of the system. Since torque current is proportional to the torque applied at the rotor, kA is the coefficient used to scale the amperes applied to an acceleration the system will see. It is the backbone of the profile and doing most of the heavy lifting.
5. The kV gain controls the compensation due to drag in the system. If the mechanism sees a lot of drag, the measured velocity will gradually fall behind the setpoint, so tuning kV to account for the drag compensates in that manner. However, many position systems do not have significant drag and can use a kV of 0.
6. With the feed forwards taken care of, the feedback tuning comes into play, with kP being used to control how strongly the system minimizes error. However, in TorqueCurrentFOC modes there is no natural dampening force, so overshoot/oscillation is expected at the beginning. At that point, kD should be tuned.
7. The kD gain will effectively act as a kP on velocity. Increasing it will increase the force bringing the system to the target velocity, so it should be increased until the system no longer oscillates.
8. Steps 6 and 7 are repeated until the gains reach the maximum they can be, indicating they're optimal for the system. These gains probably will not work in normal position closed loops, as they rely on a significant amount of feedforward to naturally dampen the system as physics would expect.

#### Tuning Process Example

Following the guide, I have set all gains to 0.

I start with kS and set it to 1, and notice it doesn't move, but it starts moving after increase it to 2. Back off to 1.5, and 1.25 until it stops. I nudge it up to 1.3 and see it's still moving, so I move it back down to 1.25 and leave it there.

Moving on to kA, I start with a kA of 1, and the system doesn't reach the necessary acceleration or velocity at all, so I increase it to 10, 20, 40, then 80 before I see it finally overshoot. I then start reducing kA to 60, then back up to 65 where I'm pretty much at the right acceleration.

I then look at tuning kV to account for the friction due to drag. Looking at the graph, the velocity remains relatively consistent throughout the long motion, so I could use a kV of 0, but for the sake of the example I will increase kV. Starting with 1 creates some overshoot. Down to 0.5, then 0.2 and it's close, but still gaining a bit at the end. Finally 0.15 produces a good response to account for drag.

Now it's time to tune P. I start with a kP of 1, and notice there's barely a response. Increasing kP to 10, then 100 has a noticeable change and a slow oscillation, but it's still far too weak. Going up to 2000 creates a noticeable oscillation, so kD should be increased to dampen it.

Starting with a kD of 1, the oscillation is still present, so I increase it to 10, then 100 where there's an impact but it's not enough. Doubling at this point to 200, then 400 looks much better, but it also looks like it can be further improved, so it doubles again to 800 where it looks sufficient to move back to kP.

Since kP is already at 2000, I'll double to 4000, then 8000, then 16000 before the oscillation at ~2 seconds appears significant. The end point looks good, though, so there's no more need to increase kP, as long as I can remove the oscillation with kD.

So kD increases to 1600, then 3200, at which point there is some oscillation on the way to the target. Backing off to 2400, then 2000 looks to have hardly any overshoot or oscillation at all, and the end position is right on top of the target, so it looks sufficient for this mechanism.

This gives final gains of kS = 1.25 A, kV = 0.15 A/rps, kA = 65 A/(rot/s$^2$), kP = 16000 A/rot, and kD = 2000 A/rps.

### 19.1.7 Remote Sensors

The Talon FX® and Talon FXS® support various remote sensors. Remote sensors allow on-board 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 =␣
↪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 Talon FX® and Talon FXS® 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

```
60      /* Configure CANcoder to zero the magnet appropriately */
61      CANcoderConfiguration cc_cfg = new CANcoderConfiguration();
62      cc_cfg.MagnetSensor.AbsoluteSensorRange =␣
```

(continues on next page)

(continued from previous page)

```
↪AbsoluteSensorRangeValue.Signed_PlusMinusHalf;
63     cc_cfg.MagnetSensor.SensorDirection =␣
↪SensorDirectionValue.CounterClockwise_Positive;
64     cc_cfg.MagnetSensor.withMagnetOffset(Rotations.of(0.4));
65     m_cc.getConfigurator().apply(cc_cfg);
66
67     TalonFXConfiguration fx_cfg = new TalonFXConfiguration();
68     fx_cfg.Feedback.FeedbackRemoteSensorID = m_cc.getDeviceID();
69     fx_cfg.Feedback.FeedbackSensorSource =␣
↪FeedbackSensorSourceValue.FusedCANcoder;
70     fx_cfg.Feedback.SensorToMechanismRatio = 1.0;
71     fx_cfg.Feedback.RotorToSensorRatio = 12.8;
72
73     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_tr;
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
fx_cfg.feedback.sensor_to_mechanism_ratio = 1.0
```

(continues on next page)

(continued from previous page)

```
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 Talon FX® or Talon FXS® internal rotor sensor against the remote CANcoder®, but continue to use the rotor sensor for all closed loop control. Talon FX® and Talon FXS® 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_CANCODER
fx_cfg.feedback.sensor_to_mechanism_ratio = 1.0
fx_cfg.feedback.rotor_to_sensor_ratio = 12.8

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

# 20 Mechanisms

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

Swerve Documentation on the Phoenix 6 Swerve API

*Swerve Overview*

Differential Documentation on the Phoenix 6 Differential Mechanism APIs

*Differential Overview*

## 20.1 Swerve Overview

**Important**

Some swerve features, such as simulation support, are only available for FRC users.

Phoenix 6 incorporates a high performance swerve API supported in Java, C++, and Python. This API simplifies the boilerplate necessary for swerve and maximizes performance.

**Tip**

Tuner X supports a *swerve project creator* that greatly simplifies the setup process and eliminates common error cases.

- Small API surface, easily debuggable
  - Build robot characteristics using `SwerveModuleConstants` (Java, C++, Python) and `SwerveDrivetrainConstants` (Java, C++, Python).
  - Integrates cleanly into the WPILib command-based framework using `CommandSwerveDrivetrain` (from our examples or Tuner X).
  - Provide a lambda to telemetrize directly in the odometry loop using `registerTelemetry(...)` (Java, C++, Python).
  - Extensible and powerful control of the drivetrain via `SwerveRequest` (Java, C++, Python).

    * Built-in requests tuned for both autonomous and teleoperated robot-centric, field-centric and field-centric facing angle control.
    * Supports common scenarios such as X brake (point all modules toward the center of the robot).

- Simulation
  - Test your autonomous paths and pose estimation without a physical robot.
  - Simply call `updateSimState(...)` (Java, C++, Python) in `simulationPeriodic()` or on a separate thread.
- Performance
  - Odometry is updated synchronously with the motor controllers.
  - Odometry is received as fast as possible using a separate thread.
  - Control is run inline with odometry updates.
  - Combine with Phoenix Pro and a CANivore® with *Timesync*® for improved performance.

**Tip**

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

### 20.1.1 Hardware Requirements

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

- 4 Talon FX® or Talon FXS® drive motor controllers
- 4 Talon FX® or Talon FXS® steer motor controllers
- 1 Pigeon® 2.0
- 4 encoders (must all be one of the following)
  - 4 CANcoder® devices
  - 4 PWM absolute encoders connected to at least 2 CANdi® devices
  - 4 PWM absolute encoders connected to their corresponding steer Talon FXS®

**Note**

All drive motor controllers must be of the same type, and all steer motor controllers must be of the same type. However, the drive and steer motor controllers can be different types from each other. For example, you can utilize 4 Talon FXS® connected to a Minion® for steer and 4 Kraken X60 for drive.

### 20.1.2 Overview of the API

Simple usage is comprised of 5 core APIs:

- `SwerveDrivetrainConstants` (Java, C++, Python)
  - This class handles characteristics of the robot that are not module specific. e.g. CAN bus, Pigeon® 2.0 ID, whether FD is enabled or not.
- `SwerveModuleConstantsFactory` (Java, C++, Python)
  - Factory class with common constants used to instantiate `SwerveModuleConstants` for each module on the robot.
- `SwerveModuleConstants` (Java, C++, Python)
  - Represents the characteristics for a given module.
- `SwerveDrivetrain` (Java, C++, Python)
  - Created using `SwerveDrivetrainConstants` and a `SwerveModuleConstants` for each module, this is used to control the swerve drivetrain.
- `SwerveRequest` (Java, C++, Python)
  - Controls the drivetrain, such as driving field-centric.

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.

> **Note**
>
> Phoenix 6 supports the Java units library when applicable.

#### Defining Drivetrain Characteristics

Drivetrain, in this instance, refers to the `SwerveDrivetrainConstants` class (Java, C++, Python). This class defines characteristics that are not tied to the swerve modules, such as the CAN bus or Pigeon® 2.0 device ID.

> **Note**
>
> All devices in the swerve drivetrain must be on the same CAN bus.

Users can optionally provide a configuration object to apply custom configs to the Pigeon® 2.0, such as mount orientation. Leaving the configuration object `null` will skip applying configs to the Pigeon® 2.0.

### Defining Module Characteristics

The typical FRC drivetrain includes 4 identical modules. To simplify module creation, the `SwerveModuleConstantsFactory` (Java, C++, Python) class is used to set up constants common across all modules, such as the drive/steer gear ratios and the wheel radius.

Some extra steps may be required to determine some constants, described below.

**`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 counter-clockwise (observed from the top of the robot). Observe the number of rotations reported by the drive motor. The coupling ratio will be $driveRotations/3$, or $driveRotations/azimuthRotations$.

> **Warning**
>
> The drive motor invert must be configured to be `CounterClockwise_Positive`, or the module constants `DriveMotorInverted` set to false, before finding the coupling ratio. In a swerve project generated by Tuner X, this is typically already the case for the left two modules (see `kInvertLeftSide` and `kInvertRightSide` in `TunerConstants`).

**`SlipCurrent`**
: This is the amount of stator current the drive motors can apply without slippage. Follow the instructions in *Preventing Wheel Slip* to find the slip current of the drivetrain.

**`DriveMotorInitialConfigs/SteerMotorInitialConfigs/EncoderInitialConfigs`**
: An initial configuration object that can be used to apply custom configs to the backing devices for each swerve module. This is useful for situations such as applying supply current limits.

### Building the Swerve Module Constants

`SwerveModuleConstants` (Java, C++, Python) can be 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.

> **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, C++, Python), where forward is positive X and left is positive Y.

### Building the `SwerveDrivetrain`

`SwerveDrivetrain` (Java, C++, Python) 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`.

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

> **Note**
>
> `CommandSwerveDrivetrain` is a version created by the Tuner X *Swerve Project Generator* that implements `Subsystem` (Java, C++, Python) for easy command-based integration.

### Java

```
// Generated by the 2026 Tuner X Swerve Project Generator
// https://v6.docs.ctr-electronics.com/en/stable/docs/tuner/tuner-swerve/index.html
public class TunerConstants {
    // 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(0.5)
        .withKS(0.1).withKV(1.91).withKA(0)
        .withStaticFeedforwardSign(StaticFeedforwardSignValue.UseClosedLoopSign);
    // 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(0.1).withKI(0).withKD(0)
        .withKS(0).withKV(0.124);

    // 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 kSteerClosedLoopOutput = 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 kDriveClosedLoopOutput = ClosedLoopOutputType.Voltage;

    // The type of motor used for the drive motor
    private static final DriveMotorArrangement kDriveMotorType = DriveMotorArrangement.TalonFX_Integrated;
    // The type of motor used for the steer motor
    private static final SteerMotorArrangement kSteerMotorType = SteerMotorArrangement.TalonFX_Integrated;

    // The remote sensor feedback type to use for the steer motors;
```

(continues on next page)

(continued from previous page)

```
    // When not Pro-licensed, Fused*/Sync* automatically fall back to Remote*
    private static final SteerFeedbackType kSteerFeedbackType = SteerFeedbackType.FusedCANcoder;

    // The stator current at which the wheels start to slip;
    // This needs to be tuned to your individual robot
    private static final Current kSlipCurrent = Amps.of(120);

    // Initial configs for the drive and steer motors and the azimuth encoder; these cannot be null.
    // Some configs will be overwritten; check the `with*InitialConfigs()` API documentation.
    private static final TalonFXConfiguration driveInitialConfigs = new TalonFXConfiguration()
        .withCurrentLimits(
            new CurrentLimitsConfigs()
                // Default supply current limit is 70 A, but it can be lowered to avoid brownouts.
                // Supply current limits can be larger than the breaker current rating.
                .withSupplyCurrentLimit(Amps.of(70))
                .withSupplyCurrentLimitEnable(true)
        );
    private static final TalonFXConfiguration steerInitialConfigs = new TalonFXConfiguration()
        .withCurrentLimits(
            new CurrentLimitsConfigs()
                // Swerve azimuth does not require much torque output, so we can set a relatively low
                // stator current limit to help avoid brownouts without impacting performance.
                .withStatorCurrentLimit(Amps.of(60))
                .withStatorCurrentLimitEnable(true)
        );
    private static final CANcoderConfiguration encoderInitialConfigs = new CANcoderConfiguration();
    // Configs for the Pigeon 2; leave this null to skip applying Pigeon 2 configs
    private static final Pigeon2Configuration pigeonConfigs = null;

    // CAN bus that the devices are located on;
    // All swerve devices must share the same CAN bus
    public static final CANBus kCANBus = new CANBus("canivore", "./logs/example.hoot");

    // Measured robot speed (m/s) at 12 V applied output;
    // This is NOT the desired max robot speed - see MaxSpeed in RobotContainer instead;
    // This needs to be tuned to your individual robot
    public static final LinearVelocity kSpeedAt12Volts = MetersPerSecond.of(4.54);

```

(continues on next page)

(continued from previous page)

```
69      // Every 1 rotation of the azimuth results in kCoupleRatio drive motor␣
    ↪turns;
70      // This may need to be tuned to your individual robot
71      private static final double kCoupleRatio = 3.8181818181818183;
72
73      private static final double kDriveGearRatio = 7.363636363636365;
74      private static final double kSteerGearRatio = 15.42857142857143;
75      private static final Distance kWheelRadius = Inches.of(2.167);
76
77      private static final boolean kInvertLeftSide = false;
78      private static final boolean kInvertRightSide = true;
79
80      private static final int kPigeonId = 1;
81
82      // These are only used for simulation
83      private static final MomentOfInertia kSteerInertia =␣
    ↪KilogramSquareMeters.of(0.01);
84      private static final MomentOfInertia kDriveInertia =␣
    ↪KilogramSquareMeters.of(0.035);
85      // Simulated voltage necessary to overcome friction
86      private static final Voltage kSteerFrictionVoltage = Volts.of(0.2);
87      private static final Voltage kDriveFrictionVoltage = Volts.of(0.2);
88
89      public static final SwerveDrivetrainConstants DrivetrainConstants = new␣
    ↪SwerveDrivetrainConstants()
90              .withCANBusName(kCANBus.getName())
91              .withPigeon2Id(kPigeonId)
92              .withPigeon2Configs(pigeonConfigs);
93
94      private static final SwerveModuleConstantsFactory<TalonFXConfiguration,␣
    ↪TalonFXConfiguration, CANcoderConfiguration> ConstantCreator =
95              new SwerveModuleConstantsFactory<TalonFXConfiguration,␣
    ↪TalonFXConfiguration, CANcoderConfiguration>()
96              .withDriveMotorGearRatio(kDriveGearRatio)
97              .withSteerMotorGearRatio(kSteerGearRatio)
98              .withCouplingGearRatio(kCoupleRatio)
99              .withWheelRadius(kWheelRadius)
100             .withSteerMotorGains(steerGains)
101             .withDriveMotorGains(driveGains)
102             .withSteerMotorClosedLoopOutput(kSteerClosedLoopOutput)
103             .withDriveMotorClosedLoopOutput(kDriveClosedLoopOutput)
104             .withSlipCurrent(kSlipCurrent)
105             .withSpeedAt12Volts(kSpeedAt12Volts)
106             .withDriveMotorType(kDriveMotorType)
107             .withSteerMotorType(kSteerMotorType)
108             .withFeedbackSource(kSteerFeedbackType)
109             .withDriveMotorInitialConfigs(driveInitialConfigs)
110             .withSteerMotorInitialConfigs(steerInitialConfigs)
111             .withEncoderInitialConfigs(encoderInitialConfigs)
112             .withSteerInertia(kSteerInertia)
113             .withDriveInertia(kDriveInertia)
114             .withSteerFrictionVoltage(kSteerFrictionVoltage)
```

(continues on next page)

(continued from previous page)

```
            .withDriveFrictionVoltage(kDriveFrictionVoltage);


    // Front Left
    private static final int kFrontLeftDriveMotorId = 3;
    private static final int kFrontLeftSteerMotorId = 2;
    private static final int kFrontLeftEncoderId = 1;
    private static final Angle kFrontLeftEncoderOffset = Rotations.of(0.15234375);
    private static final boolean kFrontLeftSteerMotorInverted = true;
    private static final boolean kFrontLeftEncoderInverted = false;

    private static final Distance kFrontLeftXPos = Inches.of(10);
    private static final Distance kFrontLeftYPos = Inches.of(10);

    // Front Right
    private static final int kFrontRightDriveMotorId = 1;
    private static final int kFrontRightSteerMotorId = 0;
    private static final int kFrontRightEncoderId = 0;
    private static final Angle kFrontRightEncoderOffset = Rotations.of(-0.4873046875);
    private static final boolean kFrontRightSteerMotorInverted = true;
    private static final boolean kFrontRightEncoderInverted = false;

    private static final Distance kFrontRightXPos = Inches.of(10);
    private static final Distance kFrontRightYPos = Inches.of(-10);

    // Back Left
    private static final int kBackLeftDriveMotorId = 7;
    private static final int kBackLeftSteerMotorId = 6;
    private static final int kBackLeftEncoderId = 3;
    private static final Angle kBackLeftEncoderOffset = Rotations.of(-0.219482421875);
    private static final boolean kBackLeftSteerMotorInverted = true;
    private static final boolean kBackLeftEncoderInverted = false;

    private static final Distance kBackLeftXPos = Inches.of(-10);
    private static final Distance kBackLeftYPos = Inches.of(10);

    // Back Right
    private static final int kBackRightDriveMotorId = 5;
    private static final int kBackRightSteerMotorId = 4;
    private static final int kBackRightEncoderId = 2;
    private static final Angle kBackRightEncoderOffset = Rotations.of(0.17236328125);
    private static final boolean kBackRightSteerMotorInverted = true;
    private static final boolean kBackRightEncoderInverted = false;

    private static final Distance kBackRightXPos = Inches.of(-10);
    private static final Distance kBackRightYPos = Inches.of(-10);


```

(continues on next page)

(continued from previous page)

```
163     public static final SwerveModuleConstants<TalonFXConfiguration, TalonFXConfiguration, CANcoderConfiguration> FrontLeft =
164         ConstantCreator.createModuleConstants(
165             kFrontLeftSteerMotorId, kFrontLeftDriveMotorId, kFrontLeftEncoderId, kFrontLeftEncoderOffset,
166             kFrontLeftXPos, kFrontLeftYPos, kInvertLeftSide, kFrontLeftSteerMotorInverted, kFrontLeftEncoderInverted
167         );
168     public static final SwerveModuleConstants<TalonFXConfiguration, TalonFXConfiguration, CANcoderConfiguration> FrontRight =
169         ConstantCreator.createModuleConstants(
170             kFrontRightSteerMotorId, kFrontRightDriveMotorId, kFrontRightEncoderId, kFrontRightEncoderOffset,
171             kFrontRightXPos, kFrontRightYPos, kInvertRightSide, kFrontRightSteerMotorInverted, kFrontRightEncoderInverted
172         );
173     public static final SwerveModuleConstants<TalonFXConfiguration, TalonFXConfiguration, CANcoderConfiguration> BackLeft =
174         ConstantCreator.createModuleConstants(
175             kBackLeftSteerMotorId, kBackLeftDriveMotorId, kBackLeftEncoderId, kBackLeftEncoderOffset,
176             kBackLeftXPos, kBackLeftYPos, kInvertLeftSide, kBackLeftSteerMotorInverted, kBackLeftEncoderInverted
177         );
178     public static final SwerveModuleConstants<TalonFXConfiguration, TalonFXConfiguration, CANcoderConfiguration> BackRight =
179         ConstantCreator.createModuleConstants(
180             kBackRightSteerMotorId, kBackRightDriveMotorId, kBackRightEncoderId, kBackRightEncoderOffset,
181             kBackRightXPos, kBackRightYPos, kInvertRightSide, kBackRightSteerMotorInverted, kBackRightEncoderInverted
182         );
183
184     /**
185      * Creates a CommandSwerveDrivetrain instance.
186      * This should only be called once in your robot program,.
187      */
188     public static CommandSwerveDrivetrain createDrivetrain() {
189         return new CommandSwerveDrivetrain(
190             DrivetrainConstants, FrontLeft, FrontRight, BackLeft, BackRight
191         );
192     }
193
194
195     /**
196      * Swerve Drive class utilizing CTR Electronics' Phoenix 6 API with the selected device types.
197      */
198     public static class TunerSwerveDrivetrain extends SwerveDrivetrain<TalonFX, TalonFX, CANcoder> {
199         /**
200          * Constructs a CTRE SwerveDrivetrain using the specified constants.
```

(continues on next page)

(continued from previous page)

```
         * <p>
         * This constructs the underlying hardware devices, so users should not construct
         * the devices themselves. If they need the devices, they can access them through
         * getters in the classes.
         *
         * @param drivetrainConstants   Drivetrain-wide constants for the swerve drive
         * @param modules               Constants for each specific module
         */
        public TunerSwerveDrivetrain(
            SwerveDrivetrainConstants drivetrainConstants,
            SwerveModuleConstants<?, ?, ?>... modules
        ) {
            super(
                TalonFX::new, TalonFX::new, CANcoder::new,
                drivetrainConstants, modules
            );
        }

        /**
         * Constructs a CTRE SwerveDrivetrain using the specified constants.
         * <p>
         * This constructs the underlying hardware devices, so users should not construct
         * the devices themselves. If they need the devices, they can access them through
         * getters in the classes.
         *
         * @param drivetrainConstants     Drivetrain-wide constants for the swerve drive
         * @param odometryUpdateFrequency The frequency to run the odometry loop. If
         *                                unspecified or set to 0 Hz, this is 250 Hz on
         *                                CAN FD, and 100 Hz on CAN 2.0.
         * @param modules                 Constants for each specific module
         */
        public TunerSwerveDrivetrain(
            SwerveDrivetrainConstants drivetrainConstants,
            double odometryUpdateFrequency,
            SwerveModuleConstants<?, ?, ?>... modules
        ) {
            super(
                TalonFX::new, TalonFX::new, CANcoder::new,
                drivetrainConstants, odometryUpdateFrequency, modules
            );
        }

        /**
         * Constructs a CTRE SwerveDrivetrain using the specified constants.
```

(continues on next page)

(continued from previous page)

```
         * <p>
         * This constructs the underlying hardware devices, so users should not construct
         * the devices themselves. If they need the devices, they can access them through
         * getters in the classes.
         *
         * @param drivetrainConstants       Drivetrain-wide constants for the swerve drive
         * @param odometryUpdateFrequency   The frequency to run the odometry loop. If
         *                                  unspecified or set to 0 Hz, this is 250 Hz on
         *                                  CAN FD, and 100 Hz on CAN 2.0.
         * @param odometryStandardDeviation The standard deviation for odometry calculation
         *                                  in the form [x, y, theta]ᵀ, with units in meters
         *                                  and radians
         * @param visionStandardDeviation   The standard deviation for vision calculation
         *                                  in the form [x, y, theta]ᵀ, with units in meters
         *                                  and radians
         * @param modules                   Constants for each specific module
         */
        public TunerSwerveDrivetrain(
            SwerveDrivetrainConstants drivetrainConstants,
            double odometryUpdateFrequency,
            Matrix<N3, N1> odometryStandardDeviation,
            Matrix<N3, N1> visionStandardDeviation,
            SwerveModuleConstants<?, ?, ?>... modules
        ) {
            super(
                TalonFX::new, TalonFX::new, CANcoder::new,
                drivetrainConstants, odometryUpdateFrequency,
                odometryStandardDeviation, visionStandardDeviation, modules
            );
        }
    }
}
```

### C++ (Header)

```
#include "ctre/phoenix6/swerve/SwerveDrivetrain.hpp"
#include "ctre/phoenix6/CANcoder.hpp"
#include "ctre/phoenix6/TalonFX.hpp"

using namespace ctre::phoenix6;

namespace subsystems {
```

(continues on next page)

(continued from previous page)

```
    /* Forward declaration */
    class CommandSwerveDrivetrain;
}

// Generated by the 2026 Tuner X Swerve Project Generator
// https://v6.docs.ctr-electronics.com/en/stable/docs/tuner/tuner-swerve/index.html
class TunerConstants {
    // 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
    static constexpr configs::Slot0Configs steerGains = configs::Slot0Configs{}
        .WithKP(100).WithKI(0).WithKD(0.5)
        .WithKS(0.1).WithKV(1.91).WithKA(0)
        .WithStaticFeedforwardSign(signals::StaticFeedforwardSignValue::UseClosedLoopSign);
    // When using closed-loop control, the drive motor uses the control
    // output type specified by SwerveModuleConstants::DriveMotorClosedLoopOutput
    static constexpr configs::Slot0Configs driveGains = configs::Slot0Configs{}
        .WithKP(0.1).WithKI(0).WithKD(0)
        .WithKS(0).WithKV(0.124);

    // The closed-loop output type to use for the steer motors;
    // This affects the PID/FF gains for the steer motors
    static constexpr swerve::ClosedLoopOutputType kSteerClosedLoopOutput = swerve::ClosedLoopOutputType::Voltage;
    // The closed-loop output type to use for the drive motors;
    // This affects the PID/FF gains for the drive motors
    static constexpr swerve::ClosedLoopOutputType kDriveClosedLoopOutput = swerve::ClosedLoopOutputType::Voltage;

    // The type of motor used for the drive motor
    static constexpr swerve::DriveMotorArrangement kDriveMotorType = swerve::DriveMotorArrangement::TalonFX_Integrated;
    // The type of motor used for the steer motor
    static constexpr swerve::SteerMotorArrangement kSteerMotorType = swerve::SteerMotorArrangement::TalonFX_Integrated;

    // The remote sensor feedback type to use for the steer motors;
    // When not Pro-licensed, Fused*/Sync* automatically fall back to Remote*
    static constexpr swerve::SteerFeedbackType kSteerFeedbackType = swerve::SteerFeedbackType::FusedCANcoder;

    // The stator current at which the wheels start to slip;
    // This needs to be tuned to your individual robot
    static constexpr units::ampere_t kSlipCurrent = 120_A;
```

(continues on next page)

(continued from previous page)

```

    // Initial configs for the drive and steer motors and the azimuth␣
↪encoder; these cannot be null.
    // Some configs will be overwritten; check the `With*InitialConfigs()`␣
↪API documentation.
    static constexpr configs::TalonFXConfiguration driveInitialConfigs =␣
↪configs::TalonFXConfiguration{}
        .WithCurrentLimits(
            configs::CurrentLimitsConfigs{}
                // Default supply current limit is 70 A, but it can be␣
↪lowered to avoid brownouts.
                // Supply current limits can be larger than the breaker␣
↪current rating.
                .WithSupplyCurrentLimit(70_A)
                .WithSupplyCurrentLimitEnable(true)
        );
    static constexpr configs::TalonFXConfiguration steerInitialConfigs =␣
↪configs::TalonFXConfiguration{}
        .WithCurrentLimits(
            configs::CurrentLimitsConfigs{}
                // Swerve azimuth does not require much torque output, so we␣
↪can set a relatively low
                // stator current limit to help avoid brownouts without␣
↪impacting performance.
                .WithStatorCurrentLimit(60_A)
                .WithStatorCurrentLimitEnable(true)
        );
    static constexpr configs::CANcoderConfiguration encoderInitialConfigs{};
    // Configs for the Pigeon 2; leave this nullopt to skip applying Pigeon 2␣
↪configs
    static constexpr std::optional<configs::Pigeon2Configuration>␣
↪pigeonConfigs = std::nullopt;

    static constexpr std::string_view kCANBusName = "canivore";

public:
    // CAN bus that the devices are located on;
    // All swerve devices must share the same CAN bus
    static inline const CANBus kCANBus{kCANBusName, "./logs/example.hoot"};

    // Measured robot speed (m/s) at 12 V applied output;
    // This is NOT the desired max robot speed - see MaxSpeed in␣
↪RobotContainer instead;
    // This needs to be tuned to your individual robot
    static constexpr units::meters_per_second_t kSpeedAt12Volts = 4.54_mps;

private:
    // Every 1 rotation of the azimuth results in kCoupleRatio drive motor␣
↪turns;
    // This may need to be tuned to your individual robot
    static constexpr units::scalar_t kCoupleRatio = 3.8181818181818183;

```

(continues on next page)

(continued from previous page)

```
    static constexpr units::scalar_t kDriveGearRatio = 7.363636363636365;
    static constexpr units::scalar_t kSteerGearRatio = 15.42857142857143;
    static constexpr units::inch_t kWheelRadius = 2.167_in;

    static constexpr bool kInvertLeftSide = false;
    static constexpr bool kInvertRightSide = true;

    static constexpr int kPigeonId = 1;

    // These are only used for simulation
    static constexpr units::kilogram_square_meter_t kSteerInertia = 0.01_
↪kg_sq_m;
    static constexpr units::kilogram_square_meter_t kDriveInertia = 0.035_
↪kg_sq_m;
    // Simulated voltage necessary to overcome friction
    static constexpr units::volt_t kSteerFrictionVoltage = 0.2_V;
    static constexpr units::volt_t kDriveFrictionVoltage = 0.2_V;

public:
    static constexpr swerve::SwerveDrivetrainConstants DrivetrainConstants =␣
↪swerve::SwerveDrivetrainConstants{}
            .WithCANBusName(kCANBusName)
            .WithPigeon2Id(kPigeonId)
            .WithPigeon2Configs(pigeonConfigs);

private:
    static constexpr swerve::SwerveModuleConstantsFactory ConstantCreator =
        swerve::SwerveModuleConstantsFactory<configs::TalonFXConfiguration,␣
↪configs::TalonFXConfiguration, configs::CANcoderConfiguration>{}
            .WithDriveMotorGearRatio(kDriveGearRatio)
            .WithSteerMotorGearRatio(kSteerGearRatio)
            .WithCouplingGearRatio(kCoupleRatio)
            .WithWheelRadius(kWheelRadius)
            .WithSteerMotorGains(steerGains)
            .WithDriveMotorGains(driveGains)
            .WithSteerMotorClosedLoopOutput(kSteerClosedLoopOutput)
            .WithDriveMotorClosedLoopOutput(kDriveClosedLoopOutput)
            .WithSlipCurrent(kSlipCurrent)
            .WithSpeedAt12Volts(kSpeedAt12Volts)
            .WithDriveMotorType(kDriveMotorType)
            .WithSteerMotorType(kSteerMotorType)
            .WithFeedbackSource(kSteerFeedbackType)
            .WithDriveMotorInitialConfigs(driveInitialConfigs)
            .WithSteerMotorInitialConfigs(steerInitialConfigs)
            .WithEncoderInitialConfigs(encoderInitialConfigs)
            .WithSteerInertia(kSteerInertia)
            .WithDriveInertia(kDriveInertia)
            .WithSteerFrictionVoltage(kSteerFrictionVoltage)
            .WithDriveFrictionVoltage(kDriveFrictionVoltage);


    // Front Left
```

(continues on next page)

(continued from previous page)

```
136     static constexpr int kFrontLeftDriveMotorId = 3;
137     static constexpr int kFrontLeftSteerMotorId = 2;
138     static constexpr int kFrontLeftEncoderId = 1;
139     static constexpr units::turn_t kFrontLeftEncoderOffset = 0.15234375_tr;
140     static constexpr bool kFrontLeftSteerMotorInverted = true;
141     static constexpr bool kFrontLeftEncoderInverted = false;
142 
143     static constexpr units::inch_t kFrontLeftXPos = 10_in;
144     static constexpr units::inch_t kFrontLeftYPos = 10_in;
145 
146     // Front Right
147     static constexpr int kFrontRightDriveMotorId = 1;
148     static constexpr int kFrontRightSteerMotorId = 0;
149     static constexpr int kFrontRightEncoderId = 0;
150     static constexpr units::turn_t kFrontRightEncoderOffset = -0.4873046875_
    ↪tr;
151     static constexpr bool kFrontRightSteerMotorInverted = true;
152     static constexpr bool kFrontRightEncoderInverted = false;
153 
154     static constexpr units::inch_t kFrontRightXPos = 10_in;
155     static constexpr units::inch_t kFrontRightYPos = -10_in;
156 
157     // Back Left
158     static constexpr int kBackLeftDriveMotorId = 7;
159     static constexpr int kBackLeftSteerMotorId = 6;
160     static constexpr int kBackLeftEncoderId = 3;
161     static constexpr units::turn_t kBackLeftEncoderOffset = -0.219482421875_
    ↪tr;
162     static constexpr bool kBackLeftSteerMotorInverted = true;
163     static constexpr bool kBackLeftEncoderInverted = false;
164 
165     static constexpr units::inch_t kBackLeftXPos = -10_in;
166     static constexpr units::inch_t kBackLeftYPos = 10_in;
167 
168     // Back Right
169     static constexpr int kBackRightDriveMotorId = 5;
170     static constexpr int kBackRightSteerMotorId = 4;
171     static constexpr int kBackRightEncoderId = 2;
172     static constexpr units::turn_t kBackRightEncoderOffset = 0.17236328125_tr;
173     static constexpr bool kBackRightSteerMotorInverted = true;
174     static constexpr bool kBackRightEncoderInverted = false;
175 
176     static constexpr units::inch_t kBackRightXPos = -10_in;
177     static constexpr units::inch_t kBackRightYPos = -10_in;
178 
179 
180 public:
181     static constexpr swerve::SwerveModuleConstants FrontLeft =␣
    ↪ConstantCreator.CreateModuleConstants(
182             kFrontLeftSteerMotorId, kFrontLeftDriveMotorId,␣
    ↪kFrontLeftEncoderId, kFrontLeftEncoderOffset,
183             kFrontLeftXPos, kFrontLeftYPos, kInvertLeftSide,␣
```

(continues on next page)

(continued from previous page)

```
↪kFrontLeftSteerMotorInverted, kFrontLeftEncoderInverted);
184     static constexpr swerve::SwerveModuleConstants FrontRight =␣
↪ConstantCreator.CreateModuleConstants(
185             kFrontRightSteerMotorId, kFrontRightDriveMotorId,␣
↪kFrontRightEncoderId, kFrontRightEncoderOffset,
186             kFrontRightXPos, kFrontRightYPos, kInvertRightSide,␣
↪kFrontRightSteerMotorInverted, kFrontRightEncoderInverted);
187     static constexpr swerve::SwerveModuleConstants BackLeft =␣
↪ConstantCreator.CreateModuleConstants(
188             kBackLeftSteerMotorId, kBackLeftDriveMotorId, kBackLeftEncoderId,␣
↪kBackLeftEncoderOffset,
189             kBackLeftXPos, kBackLeftYPos, kInvertLeftSide,␣
↪kBackLeftSteerMotorInverted, kBackLeftEncoderInverted);
190     static constexpr swerve::SwerveModuleConstants BackRight =␣
↪ConstantCreator.CreateModuleConstants(
191             kBackRightSteerMotorId, kBackRightDriveMotorId,␣
↪kBackRightEncoderId, kBackRightEncoderOffset,
192             kBackRightXPos, kBackRightYPos, kInvertRightSide,␣
↪kBackRightSteerMotorInverted, kBackRightEncoderInverted);
193
194     /**
195      * Creates a CommandSwerveDrivetrain instance.
196      * This should only be called once in your robot program.
197      */
198     static subsystems::CommandSwerveDrivetrain CreateDrivetrain();
199 };
200
201
202 /**
203  * \brief Swerve Drive class utilizing CTR Electronics' Phoenix 6 API with␣
↪the selected device types.
204  */
205 class TunerSwerveDrivetrain : public swerve::SwerveDrivetrain
↪<hardware::TalonFX, hardware::TalonFX, hardware::CANcoder> {
206 public:
207     using SwerveModuleConstants = swerve::SwerveModuleConstants
↪<configs::TalonFXConfiguration, configs::TalonFXConfiguration,␣
↪configs::CANcoderConfiguration>;
208
209     /**
210      * \brief Constructs a CTRE SwerveDrivetrain using the specified␣
↪constants.
211      *
212      * This constructs the underlying hardware devices, so users should not␣
↪construct
213      * the devices themselves. If they need the devices, they can access them
214      * through getters in the classes.
215      *
216      * \param drivetrainConstants Drivetrain-wide constants for the swerve␣
↪drive
217      * \param modules             Constants for each specific module
218      */
```

(continues on next page)

(continued from previous page)

```
219     template <std::same_as<SwerveModuleConstants>... ModuleConstants>
220     TunerSwerveDrivetrain(swerve::SwerveDrivetrainConstants const &
    ↪driveTrainConstants, ModuleConstants const &... modules) :
221         SwerveDrivetrain{driveTrainConstants, modules...}
222     {}
223
224     /**
225      * \brief Constructs a CTRE SwerveDrivetrain using the specified␣
    ↪constants.
226      *
227      * This constructs the underlying hardware devices, so users should not␣
    ↪construct
228      * the devices themselves. If they need the devices, they can access them
229      * through getters in the classes.
230      *
231      * \param drivetrainConstants        Drivetrain-wide constants for the␣
    ↪swerve drive
232      * \param odometryUpdateFrequency    The frequency to run the odometry␣
    ↪loop. If
233      *                                   unspecified or set to 0 Hz, this is␣
    ↪250 Hz on
234      *                                   CAN FD, and 100 Hz on CAN 2.0.
235      * \param modules                    Constants for each specific module
236      */
237     template <std::same_as<SwerveModuleConstants>... ModuleConstants>
238     TunerSwerveDrivetrain(
239         swerve::SwerveDrivetrainConstants const &driveTrainConstants,
240         units::hertz_t odometryUpdateFrequency,
241         ModuleConstants const &... modules
242     ) :
243         SwerveDrivetrain{driveTrainConstants, odometryUpdateFrequency,␣
    ↪modules...}
244     {}
245
246     /**
247      * \brief Constructs a CTRE SwerveDrivetrain using the specified␣
    ↪constants.
248      *
249      * This constructs the underlying hardware devices, so users should not␣
    ↪construct
250      * the devices themselves. If they need the devices, they can access them
251      * through getters in the classes.
252      *
253      * \param drivetrainConstants        Drivetrain-wide constants for the␣
    ↪swerve drive
254      * \param odometryUpdateFrequency    The frequency to run the odometry␣
    ↪loop. If
255      *                                   unspecified or set to 0 Hz, this is␣
    ↪250 Hz on
256      *                                   CAN FD, and 100 Hz on CAN 2.0.
257      * \param odometryStandardDeviation  The standard deviation for odometry␣
    ↪calculation
```

(continues on next page)

(continued from previous page)

```
     *                                    in the form [x, y, theta]ᵀ, with␣
↪units in meters
     *                                    and radians
     * \param visionStandardDeviation    The standard deviation for vision␣
↪calculation
     *                                    in the form [x, y, theta]ᵀ, with␣
↪units in meters
     *                                    and radians
     * \param modules                     Constants for each specific module
     */
    template <std::same_as<SwerveModuleConstants>... ModuleConstants>
    TunerSwerveDrivetrain(
        swerve::SwerveDrivetrainConstants const &driveTrainConstants,
        units::hertz_t odometryUpdateFrequency,
        std::array<double, 3> const &odometryStandardDeviation,
        std::array<double, 3> const &visionStandardDeviation,
        ModuleConstants const &... modules
    ) :
        SwerveDrivetrain{
            driveTrainConstants, odometryUpdateFrequency,
            odometryStandardDeviation, visionStandardDeviation, modules...
        }
    {}
};
```

### C++ (Source)

```
#include "generated/TunerConstants.h"
#include "subsystems/CommandSwerveDrivetrain.h"

subsystems::CommandSwerveDrivetrain TunerConstants::CreateDrivetrain()
{
    return {DrivetrainConstants, FrontLeft, FrontRight, BackLeft, BackRight};
}
```

### Python

```
    from subsystems.command_swerve_drivetrain import CommandSwerveDrivetrain


class TunerConstants:
    """
    Generated by the 2026 Tuner X Swerve Project Generator
␣
↪https://v6.docs.ctr-electronics.com/en/stable/docs/tuner/tuner-swerve/index.html
    """

    # Both sets of gains need to be tuned to your individual robot

    # The steer motor uses any SwerveModule.SteerRequestType control request␣
```

(continues on next page)

(continued from previous page)

```
    ↪with the
13      # output type specified by␣
    ↪SwerveModuleConstants.SteerMotorClosedLoopOutput
14      _steer_gains = (
15          configs.Slot0Configs()
16          .with_k_p(100)
17          .with_k_i(0)
18          .with_k_d(0.5)
19          .with_k_s(0.1)
20          .with_k_v(1.91)
21          .with_k_a(0)
22          .with_static_feedforward_sign(
23              signals.StaticFeedforwardSignValue.USE_CLOSED_LOOP_SIGN
24          )
25      )
26      # When using closed-loop control, the drive motor uses the control
27      # output type specified by␣
    ↪SwerveModuleConstants.DriveMotorClosedLoopOutput
28      _drive_gains = (
29          configs.Slot0Configs()
30          .with_k_p(0.1)
31          .with_k_i(0)
32          .with_k_d(0)
33          .with_k_s(0)
34          .with_k_v(0.124)
35      )
36
37      # The closed-loop output type to use for the steer motors;
38      # This affects the PID/FF gains for the steer motors
39      _steer_closed_loop_output = swerve.ClosedLoopOutputType.VOLTAGE
40      # The closed-loop output type to use for the drive motors;
41      # This affects the PID/FF gains for the drive motors
42      _drive_closed_loop_output = swerve.ClosedLoopOutputType.VOLTAGE
43
44      # The type of motor used for the drive motor
45      _drive_motor_type = swerve.DriveMotorArrangement.TALON_FX_INTEGRATED
46      # The type of motor used for the steer motor
47      _steer_motor_type = swerve.SteerMotorArrangement.TALON_FX_INTEGRATED
48
49      # The remote sensor feedback type to use for the steer motors;
50      # When not Pro-licensed, Fused*/Sync* automatically fall back to Remote*
51      _steer_feedback_type = swerve.SteerFeedbackType.FUSED_CANCODER
52
53      # The stator current at which the wheels start to slip;
54      # This needs to be tuned to your individual robot
55      _slip_current: units.ampere = 120.0
56
57      # Initial configs for the drive and steer motors and the azimuth encoder;␣
    ↪these cannot be null.
58      # Some configs will be overwritten; check the `with_*_initial_configs()`␣
    ↪API documentation.
59      _drive_initial_configs = configs.TalonFXConfiguration().
```

(continues on next page)

(continued from previous page)

```
↪with_current_limits(
60          configs.CurrentLimitsConfigs()
61          # Default supply current limit is 70 A, but it can be lowered to␣
↪avoid brownouts.
62          # Supply current limits can be larger than the breaker current rating.
63          .with_supply_current_limit(70.0)
64          .with_supply_current_limit_enable(True)
65      )
66      _steer_initial_configs = configs.TalonFXConfiguration().
↪with_current_limits(
67          configs.CurrentLimitsConfigs()
68          # Swerve azimuth does not require much torque output, so we can set a␣
↪relatively low
69          # stator current limit to help avoid brownouts without impacting␣
↪performance.
70          .with_stator_current_limit(60.0)
71          .with_stator_current_limit_enable(True)
72      )
73      _encoder_initial_configs = configs.CANcoderConfiguration()
74      # Configs for the Pigeon 2; leave this None to skip applying Pigeon 2␣
↪configs
75      _pigeon_configs: configs.Pigeon2Configuration | None = None
76
77      # CAN bus that the devices are located on;
78      # All swerve devices must share the same CAN bus
79      canbus = CANBus("canivore", "./logs/example.hoot")
80
81      # Measured robot speed (m/s) at 12 V applied output;
82      # This is NOT the desired max robot speed - see _max_speed in␣
↪RobotContainer instead;
83      # This needs to be tuned to your individual robot
84      speed_at_12_volts: units.meters_per_second = 4.54
85
86      # Every 1 rotation of the azimuth results in _couple_ratio drive motor␣
↪turns;
87      # This may need to be tuned to your individual robot
88      _couple_ratio = 3.8181818181818183
89
90      _drive_gear_ratio = 7.363636363636365
91      _steer_gear_ratio = 15.42857142857143
92      _wheel_radius: units.meter = inchesToMeters(2.167)
93
94      _invert_left_side = False
95      _invert_right_side = True
96
97      _pigeon_id = 1
98
99      # These are only used for simulation
100     _steer_inertia: units.kilogram_square_meter = 0.01
101     _drive_inertia: units.kilogram_square_meter = 0.035
102     # Simulated voltage necessary to overcome friction
103     _steer_friction_voltage: units.volt = 0.2
```

(continues on next page)

(continued from previous page)

```
    _drive_friction_voltage: units.volt = 0.2

    drivetrain_constants = (
        swerve.SwerveDrivetrainConstants()
        .with_can_bus_name(canbus.name)
        .with_pigeon2_id(_pigeon_id)
        .with_pigeon2_configs(_pigeon_configs)
    )

    _constants_creator = (
        swerve.SwerveModuleConstantsFactory[
            configs.TalonFXConfiguration,
            configs.TalonFXConfiguration,
            configs.CANcoderConfiguration,
        ]()
        .with_drive_motor_gear_ratio(_drive_gear_ratio)
        .with_steer_motor_gear_ratio(_steer_gear_ratio)
        .with_coupling_gear_ratio(_couple_ratio)
        .with_wheel_radius(_wheel_radius)
        .with_steer_motor_gains(_steer_gains)
        .with_drive_motor_gains(_drive_gains)
        .with_steer_motor_closed_loop_output(_steer_closed_loop_output)
        .with_drive_motor_closed_loop_output(_drive_closed_loop_output)
        .with_slip_current(_slip_current)
        .with_speed_at12_volts(speed_at_12_volts)
        .with_drive_motor_type(_drive_motor_type)
        .with_steer_motor_type(_steer_motor_type)
        .with_feedback_source(_steer_feedback_type)
        .with_drive_motor_initial_configs(_drive_initial_configs)
        .with_steer_motor_initial_configs(_steer_initial_configs)
        .with_encoder_initial_configs(_encoder_initial_configs)
        .with_steer_inertia(_steer_inertia)
        .with_drive_inertia(_drive_inertia)
        .with_steer_friction_voltage(_steer_friction_voltage)
        .with_drive_friction_voltage(_drive_friction_voltage)
    )


    # Front Left
    _front_left_drive_motor_id = 3
    _front_left_steer_motor_id = 2
    _front_left_encoder_id = 1
    _front_left_encoder_offset: units.rotation = 0.15234375
    _front_left_steer_motor_inverted = True
    _front_left_encoder_inverted = False

    _front_left_x_pos: units.meter = inchesToMeters(10)
    _front_left_y_pos: units.meter = inchesToMeters(10)

    # Front Right
    _front_right_drive_motor_id = 1
    _front_right_steer_motor_id = 0
```

(continues on next page)

(continued from previous page)

```
    _front_right_encoder_id = 0
    _front_right_encoder_offset: units.rotation = -0.4873046875
    _front_right_steer_motor_inverted = True
    _front_right_encoder_inverted = False

    _front_right_x_pos: units.meter = inchesToMeters(10)
    _front_right_y_pos: units.meter = inchesToMeters(-10)

    # Back Left
    _back_left_drive_motor_id = 7
    _back_left_steer_motor_id = 6
    _back_left_encoder_id = 3
    _back_left_encoder_offset: units.rotation = -0.219482421875
    _back_left_steer_motor_inverted = True
    _back_left_encoder_inverted = False

    _back_left_x_pos: units.meter = inchesToMeters(-10)
    _back_left_y_pos: units.meter = inchesToMeters(10)

    # Back Right
    _back_right_drive_motor_id = 5
    _back_right_steer_motor_id = 4
    _back_right_encoder_id = 2
    _back_right_encoder_offset: units.rotation = 0.17236328125
    _back_right_steer_motor_inverted = True
    _back_right_encoder_inverted = False

    _back_right_x_pos: units.meter = inchesToMeters(-10)
    _back_right_y_pos: units.meter = inchesToMeters(-10)


    front_left = _constants_creator.create_module_constants(
        _front_left_steer_motor_id,
        _front_left_drive_motor_id,
        _front_left_encoder_id,
        _front_left_encoder_offset,
        _front_left_x_pos,
        _front_left_y_pos,
        _invert_left_side,
        _front_left_steer_motor_inverted,
        _front_left_encoder_inverted,
    )
    front_right = _constants_creator.create_module_constants(
        _front_right_steer_motor_id,
        _front_right_drive_motor_id,
        _front_right_encoder_id,
        _front_right_encoder_offset,
        _front_right_x_pos,
        _front_right_y_pos,
        _invert_right_side,
        _front_right_steer_motor_inverted,
        _front_right_encoder_inverted,
```

(continues on next page)

(continued from previous page)

```
    )
    back_left = _constants_creator.create_module_constants(
        _back_left_steer_motor_id,
        _back_left_drive_motor_id,
        _back_left_encoder_id,
        _back_left_encoder_offset,
        _back_left_x_pos,
        _back_left_y_pos,
        _invert_left_side,
        _back_left_steer_motor_inverted,
        _back_left_encoder_inverted,
    )
    back_right = _constants_creator.create_module_constants(
        _back_right_steer_motor_id,
        _back_right_drive_motor_id,
        _back_right_encoder_id,
        _back_right_encoder_offset,
        _back_right_x_pos,
        _back_right_y_pos,
        _invert_right_side,
        _back_right_steer_motor_inverted,
        _back_right_encoder_inverted,
    )

    @classmethod
    def create_drivetrain(cls) -> "CommandSwerveDrivetrain":
        """
        Creates a CommandSwerveDrivetrain instance.
        This should only be called once in your robot program.
        """
        from subsystems.command_swerve_drivetrain import␣
↪CommandSwerveDrivetrain

        return CommandSwerveDrivetrain(
            cls.drivetrain_constants,
            [
                cls.front_left,
                cls.front_right,
                cls.back_left,
                cls.back_right,
            ],
        )


class TunerSwerveDrivetrain(
    swerve.SwerveDrivetrain[hardware.TalonFX, hardware.TalonFX, hardware.
↪CANcoder]
):
    """Swerve Drive class utilizing CTR Electronics' Phoenix 6 API with the␣
↪selected device types."""

    @overload
```

(continues on next page)

(continued from previous page)

```
    def __init__(
        self,
        drivetrain_constants: swerve.SwerveDrivetrainConstants,
        modules: list[swerve.SwerveModuleConstants],
        /,
    ) -> None:
        """
        Constructs a CTRE SwerveDrivetrain using the specified constants.

        This constructs the underlying hardware devices, so users should not construct
        the devices themselves. If they need the devices, they can access them through
        getters in the classes.

        :param drivetrain_constants: Drivetrain-wide constants for the swerve drive
        :type drivetrain_constants:  swerve.SwerveDrivetrainConstants
        :param modules:              Constants for each specific module
        :type modules:               list[swerve.SwerveModuleConstants]
        """
        ...

    @overload
    def __init__(
        self,
        drivetrain_constants: swerve.SwerveDrivetrainConstants,
        odometry_update_frequency: units.hertz,
        modules: list[swerve.SwerveModuleConstants],
        /,
    ) -> None:
        """
        Constructs a CTRE SwerveDrivetrain using the specified constants.

        This constructs the underlying hardware devices, so users should not construct
        the devices themselves. If they need the devices, they can access them through
        getters in the classes.

        :param drivetrain_constants:        Drivetrain-wide constants for the swerve drive
        :type drivetrain_constants:         swerve.SwerveDrivetrainConstants
        :param odometry_update_frequency:   The frequency to run the odometry loop. If
                                            unspecified or set to 0 Hz, this is 250 Hz on
                                            CAN FD, and 100 Hz on CAN 2.0.
        :type odometry_update_frequency:    units.hertz
        :param modules:                     Constants for each specific module
        :type modules:                      list[swerve.SwerveModuleConstants]
        """
```

(continues on next page)

(continued from previous page)

```
        ...

    @overload
    def __init__(
        self,
        drivetrain_constants: swerve.SwerveDrivetrainConstants,
        odometry_update_frequency: units.hertz,
        odometry_standard_deviation: tuple[float, float, float],
        vision_standard_deviation: tuple[float, float, float],
        modules: list[swerve.SwerveModuleConstants],
        /,
    ) -> None:
        """
        Constructs a CTRE SwerveDrivetrain using the specified constants.

        This constructs the underlying hardware devices, so users should not␣
↪construct
        the devices themselves. If they need the devices, they can access␣
↪them through
        getters in the classes.

        :param drivetrain_constants:        Drivetrain-wide constants for the␣
↪swerve drive
        :type drivetrain_constants:         swerve.SwerveDrivetrainConstants
        :param odometry_update_frequency:   The frequency to run the odometry␣
↪loop. If
                                            unspecified or set to 0 Hz, this␣
↪is 250 Hz on
                                            CAN FD, and 100 Hz on CAN 2.0.
        :type odometry_update_frequency:    units.hertz
        :param odometry_standard_deviation: The standard deviation for␣
↪odometry calculation
                                            in the form [x, y, theta]ᵀ, with␣
↪units in meters
                                            and radians
        :type odometry_standard_deviation:  tuple[float, float, float]
        :param vision_standard_deviation:   The standard deviation for vision␣
↪calculation
                                            in the form [x, y, theta]ᵀ, with␣
↪units in meters
                                            and radians
        :type vision_standard_deviation:    tuple[float, float, float]
        :param modules:                     Constants for each specific module
        :type modules:                      list[swerve.SwerveModuleConstants]
        """
        ...

    @overload
    def __init__(
        self,
        drivetrain_constants: swerve.SwerveDrivetrainConstants,
        arg0: None,
```

(continues on next page)

(continued from previous page)

```
        arg1: None,
        arg2: None,
        arg3: None,
        /,
    ) -> None: ...

    def __init__(
        self,
        drivetrain_constants: swerve.SwerveDrivetrainConstants,
        arg0=None,
        arg1=None,
        arg2=None,
        arg3=None,
    ):
        swerve.SwerveDrivetrain.__init__(
            self,
            hardware.TalonFX,
            hardware.TalonFX,
            hardware.CANcoder,
            drivetrain_constants,
            arg0,
            arg1,
            arg2,
            arg3,
        )
```

### Swerve Requests

Controlling the drivetrain is done by calling `setControl(SwerveRequest)` (Java, C++, Python) periodically, which takes a given `SwerveRequest` (Java, C++, Python). There are multiple pre-defined `SwerveRequest` implementations that cover the majority of use cases. In some advanced scenarios, users can also define their own.

### Applying a Request

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

### Java

```
private double MaxSpeed = TunerConstants.kSpeedAt12Volts.in(MetersPerSecond);
private double MaxAngularRate =␣
↪RotationsPerSecond.of(0.75).in(RadiansPerSecond);

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

(continues on next page)

(continued from previous page)

```
private final XboxController m_joystick = new XboxController(0);
public final TunerSwerveDrivetrain drivetrain = TunerConstants.createDrivetrain();

@Override
public void teleopPeriodic() {
   // Note that X is defined as forward according to WPILib convention,
   // and Y is defined as to the left according to WPILib convention.
   drivetrain.setControl(
      m_driveRequest.withVelocityX(-joystick.getLeftY() * MaxSpeed)
         .withVelocityY(-joystick.getLeftX() * MaxSpeed)
         .withRotationalRate(-joystick.getRightX() * MaxAngularRate)
   );
}
```

### C++

```
private:
   units::meters_per_second_t MaxSpeed = TunerConstants::kSpeedAt12Volts;
   units::radians_per_second_t MaxAngularRate = 0.75_tps;

   swerve::requests::FieldCentric m_driveRequest = swerve::requests::FieldCentric{}
      .WithDeadband(MaxSpeed * 0.1).WithRotationalDeadband(MaxAngularRate * 0.1) // Add a 10% deadband
      .WithDriveRequestType(swerve::DriveRequestType::OpenLoopVoltage)
      .WithSteerRequestType(swerve::SteerRequestType::Position);

   frc::XboxController m_joystick{0};

public:
   TunerSwerveDrivetrain drivetrain{TunerConstants::CreateDrivetrain()};

   void TeleopPeriodic() override
   {
      // Note that X is defined as forward according to WPILib convention,
      // and Y is defined as to the left according to WPILib convention.
      drivetrain.SetControl(
         m_driveRequest.WithVelocityX(-joystick.GetLeftY() * MaxSpeed)
            .WithVelocityY(-joystick.GetLeftX() * MaxSpeed)
            .WithRotationalRate(-joystick.GetRightX() * MaxAngularRate)
      );
   }
```

### Python

```
self._max_speed = (
   TunerConstants.speed_at_12_volts
)
```

(continues on next page)

(continued from previous page)

```
self._max_angular_rate = rotationsToRadians(
    0.75
)

self._drive_request = (
    swerve.requests.FieldCentric()
    .with_deadband(self._max_speed * 0.1)
    .with_rotational_deadband(
        self._max_angular_rate * 0.1
    )  # Add a 10% deadband
    .with_drive_request_type(
        swerve.SwerveModule.DriveRequestType.OPEN_LOOP_VOLTAGE
    )
    .with_steer_request_type(
        swerve.SwerveModule.SteerRequestType.POSITION
    )
)

self._joystick = XboxController(0)
self.drivetrain = TunerConstants.create_drivetrain()

def teleopPeriodic():
    # Note that X is defined as forward according to WPILib convention,
    # and Y is defined as to the left according to WPILib convention.
    self.drivetrain.set_control(
        self._drive_request
        .with_velocity_x(
            -self._joystick.getLeftY() * self._max_speed
        )
        .with_velocity_y(
            -self._joystick.getLeftX() * self._max_speed
        )
        .with_rotational_rate(
            -self._joystick.getRightX() * self._max_angular_rate
        )
    )
```

### Command-Based

When using the command-based `CommandSwerveDrivetrain` generated by Tuner X, the `applyRequest(Supplier<SwerveRequest>)` method can instead be used to get a command that periodically applies the `SwerveRequest` returned by the lambda.

### Java

```
private double MaxSpeed = TunerConstants.kSpeedAt12Volts.in(MetersPerSecond);
private double MaxAngularRate =␣
↪RotationsPerSecond.of(0.75).in(RadiansPerSecond);

private final SwerveRequest.FieldCentric m_driveRequest = new␣
↪SwerveRequest.FieldCentric()
```

(continues on next page)

(continued from previous page)

```
   .withDeadband(MaxSpeed * 0.1).withRotationalDeadband(MaxAngularRate * 0.1)␣
↪// Add a 10% deadband
   .withDriveRequestType(DriveRequestType.OpenLoopVoltage)
   .withSteerRequestType(SteerRequestType.Position);

private final CommandXboxController m_joystick = new CommandXboxController(0);
public final CommandSwerveDrivetrain drivetrain =␣
↪TunerConstants.createDrivetrain();

public void configureBindings() {
   // Note that X is defined as forward according to WPILib convention,
   // and Y is defined as to the left according to WPILib convention.
   drivetrain.setDefaultCommand(
      // Drivetrain will execute this command periodically
      drivetrain.applyRequest(() ->
         m_driveRequest.withVelocityX(-joystick.getLeftY() * MaxSpeed)
            .withVelocityY(-joystick.getLeftX() * MaxSpeed)
            .withRotationalRate(-joystick.getRightX() * MaxAngularRate)
      )
   );

   // Idle while the robot is disabled. This ensures the configured
   // neutral mode is applied to the drive motors while disabled.
   final var idle = new SwerveRequest.Idle();
   RobotModeTriggers.disabled().whileTrue(
      drivetrain.applyRequest(() -> idle).ignoringDisable(true)
   );
}
```

### C++

```
private:
   units::meters_per_second_t MaxSpeed = TunerConstants::kSpeedAt12Volts;
   units::radians_per_second_t MaxAngularRate = 0.75_tps;

   swerve::requests::FieldCentric m_driveRequest =␣
↪swerve::requests::FieldCentric{}
      .WithDeadband(MaxSpeed * 0.1).WithRotationalDeadband(MaxAngularRate *␣
↪0.1) // Add a 10% deadband
      .WithDriveRequestType(swerve::DriveRequestType::OpenLoopVoltage)
      .WithSteerRequestType(swerve::SteerRequestType::Position);

   frc::XboxController m_joystick{0};

public:
   subsystems::CommandSwerveDrivetrain␣
↪drivetrain{TunerConstants::CreateDrivetrain()};

   void ConfigureBindings()
   {
      // Note that X is defined as forward according to WPILib convention,
```

(continues on next page)

(continued from previous page)

```
      // and Y is defined as to the left according to WPILib convention.
      drivetrain.SetDefaultCommand(
         // Drivetrain will execute this command periodically
         drivetrain.ApplyRequest([this]() -> auto&& {
            return m_driveRequest.WithVelocityX(-joystick.GetLeftY() *␣
↪MaxSpeed)
                .WithVelocityY(-joystick.GetLeftX() * MaxSpeed)
                .WithRotationalRate(-joystick.GetRightX() * MaxAngularRate);
         })
      );

      // Idle while the robot is disabled. This ensures the configured
      // neutral mode is applied to the drive motors while disabled.
      frc2::RobotModeTriggers::Disabled().WhileTrue(
         drivetrain.ApplyRequest([] {
            return swerve::requests::Idle{};
         }).IgnoringDisable(true)
      );
   }
```

### Python

```
self._max_speed = (
   TunerConstants.speed_at_12_volts
)
self._max_angular_rate = rotationsToRadians(
   0.75
)

self._drive_request = (
   swerve.requests.FieldCentric()
   .with_deadband(self._max_speed * 0.1)
   .with_rotational_deadband(
      self._max_angular_rate * 0.1
   )  # Add a 10% deadband
   .with_drive_request_type(
      swerve.SwerveModule.DriveRequestType.OPEN_LOOP_VOLTAGE
   )
   .with_steer_request_type(
      swerve.SwerveModule.SteerRequestType.POSITION
   )
)

self._joystick = CommandXboxController(0)
self.drivetrain = TunerConstants.create_drivetrain()

def configureButtonBindings() -> None:
   # Note that X is defined as forward according to WPILib convention,
   # and Y is defined as to the left according to WPILib convention.
   self.drivetrain.setDefaultCommand(
      # Drivetrain will execute this command periodically
```

(continues on next page)

(continued from previous page)

```
        self.drivetrain.apply_request(
            lambda: (
                self._drive_request.with_velocity_x(
                    -self._joystick.getLeftY() * self._max_speed
                )  # Drive forward with negative Y (forward)
                .with_velocity_y(
                    -self._joystick.getLeftX() * self._max_speed
                )  # Drive left with negative X (left)
                .with_rotational_rate(
                    -self._joystick.getRightX() * self._max_angular_rate
                )  # Drive counterclockwise with negative X (left)
            )
        )
    )

    # Idle while the robot is disabled. This ensures the configured
    # neutral mode is applied to the drive motors while disabled.
    idle = swerve.requests.Idle()
    Trigger(DriverStation.isDisabled).whileTrue(
        self.drivetrain.apply_request(lambda: idle).ignoringDisable(True)
    )
```

### Custom Swerve Requests

In many cases, advanced control logic can live in the command applying the swerve request. For example, path following is typically implemented using a WPILib `Command` factory in the subsystem. Most path planning libraries generate the path setpoints in the main robot loop, and PID on the `Pose2d` must be run inline with setpoint generation.

However, there are some advanced cases where it is beneficial to run some of the control logic at the higher update frequency of the odometry thread. To accomplish that, users can define custom swerve requests by implementing the `SwerveRequest` interface.

In a custom swerve request, the control logic lives in the `apply(...)` method, which is called by the odometry thread.

**Important**

Custom swerve requests can have a performance cost compared to the native implementations. Additionally, the `apply(...)` method must be fast to avoid blocking odometry updates.

### Swerve Requests with Composition

To maximize performance and minimize duplicate code, most custom swerve requests should be built on top of existing ones. For example, the built-in `FieldCentricFacingAngle` (Java, C++, Python) request uses a regular `FieldCentric` request under the hood, as demonstrated below.

### Java

```
private final FieldCentric m_fieldCentric = new FieldCentric();

@Override
public StatusCode apply(SwerveControlParameters parameters, SwerveModule<?, ?,
↪ ?>... modulesToApply) {
   Rotation2d angleToFace = TargetDirection;
   if (ForwardPerspective == ForwardPerspectiveValue.OperatorPerspective) {
      /* If we're operator perspective, rotate the direction we want to face␣
↪by the angle */
      angleToFace = angleToFace.rotateBy(parameters.operatorForwardDirection);
   }

   double toApplyOmega = TargetRateFeedforward +
      HeadingController.calculate(
         parameters.currentPose.getRotation().getRadians(),
         angleToFace.getRadians(),
         parameters.timestamp
      );

   /* The rest of the logic is the same as FieldCentric, so
    * set up and call FieldCentric's apply() method */
   return m_fieldCentric
      .withVelocityX(VelocityX)
      .withVelocityY(VelocityY)
      .withRotationalRate(toApplyOmega)
      .withDeadband(Deadband)
      .withRotationalDeadband(RotationalDeadband)
      .withCenterOfRotation(CenterOfRotation)
      .withDriveRequestType(DriveRequestType)
      .withSteerRequestType(SteerRequestType)
      .withDesaturateWheelSpeeds(DesaturateWheelSpeeds)
      .withForwardPerspective(ForwardPerspective)
      .apply(parameters, modulesToApply);
}
```

### C++

```
ctre::phoenix::StatusCode Apply(
   swerve::requests::SwerveRequest::ControlParameters const &parameters,
   std::span<std::unique_ptr<swerve::impl::SwerveModuleImpl> const>␣
↪modulesToApply
) override {
   swerve::Rotation2d angleToFace = TargetDirection;
   if (ForwardPerspective ==␣
↪swerve::requests::ForwardPerspectiveValue::OperatorPerspective) {
      /* If we're operator perspective, rotate the direction we want to face␣
↪by the angle */
      angleToFace = angleToFace.RotateBy(parameters.operatorForwardDirection);
   }
```

(continues on next page)

(continued from previous page)

```
   units::radians_per_second_t toApplyOmega = TargetRateFeedforward +
      units::radians_per_second_t{HeadingController.Calculate(
         parameters.currentPose.Rotation().Radians().value(),
         angleToFace.Radians().value(),
         parameters.timestamp
      )};

   /* The rest of the logic is the same as FieldCentric, so
    * set up and call FieldCentric's Apply() method */
   return swerve::requests::FieldCentric{}
      .WithVelocityX(VelocityX)
      .WithVelocityY(VelocityY)
      .WithRotationalRate(toApplyOmega)
      .WithDeadband(Deadband)
      .WithRotationalDeadband(RotationalDeadband)
      .WithCenterOfRotation(CenterOfRotation)
      .WithDriveRequestType(DriveRequestType)
      .WithSteerRequestType(SteerRequestType)
      .WithDesaturateWheelSpeeds(DesaturateWheelSpeeds)
      .WithForwardPerspective(ForwardPerspective)
      .Apply(parameters, modulesToApply);
}
```

### Python

```
def __init__(self):
   # ...
   self.__field_centric = FieldCentric()

def apply(
   self,
   parameters: swerve.SwerveControlParameters,
   modules_to_apply: list[swerve.SwerveModule]
) -> StatusCode:
   angle_to_face = self.target_direction
   if self.forward_perspective is swerve.requests.ForwardPerspectiveValue.
↪OPERATOR_PERSPECTIVE:
      # If we're operator perspective, rotate the direction we want to face␣
↪by the angle
      angle_to_face = angle_to_face.rotateBy(parameters.
↪operator_forward_direction)

   to_apply_omega = self.target_rate_feedforward + self.heading_controller.
↪calculate(
      parameters.current_pose.rotation().radians(),
      angle_to_face.radians(),
      parameters.timestamp
   )

   # The rest of the logic is the same as FieldCentric, so
   # set up and call FieldCentric's apply() method
```

(continues on next page)

(continued from previous page)

```
    return (
        self.__field_centric
        .with_velocity_x(self.velocity_x)
        .with_velocity_y(self.velocity_y)
        .with_rotational_rate(to_apply_omega)
        .with_deadband(self.deadband)
        .with_rotational_deadband(self.rotational_deadband)
        .with_center_of_rotation(self.center_of_rotation)
        .with_drive_request_type(self.drive_request_type)
        .with_steer_request_type(self.steer_request_type)
        .with_desaturate_wheel_speeds(self.desaturate_wheel_speeds)
        .with_forward_perspective(self.forward_perspective)
        .apply(parameters, modules_to_apply)
    )
```

### Swerve Requests with Module Targets

In a few cases, none of the existing swerve request implementations may be suitable for the desired request. For example, there is no built-in swerve request that directly accepts an array of `SwerveModuleState` instances. In that situation, the custom swerve request can call `apply(SwerveModule.ModuleRequest)` (Java, C++, Python) on each `SwerveModule` instance provided to the `apply(...)` method.

Note, however, that this can **negatively impact performance** of the robot, both in terms of loop times and control accuracy, compared to reusing the built-in requests. As a result, we recommend converting to supported types, such as `ChassisSpeeds`, and reusing existing swerve requests, such as `ApplyFieldSpeeds` (Java, C++, Python), whenever possible.

> **Warning**
>
> We recommend against using a custom swerve request for the WPILib `SwerveControllerCommand`, as it does not follow modern WPILib best practices. Instead, the command can be reimplemented as a command factory using `ApplyFieldSpeeds` to maximize performance.

### Java

```
public class ApplyModuleStates implements SwerveRequest {
    public SwerveModuleState[] ModuleStates = new SwerveModuleState[0];

    @Override
    public StatusCode apply(SwerveControlParameters parameters, SwerveModule<?,
↪ ?, ?>... modulesToApply) {
        var moduleRequest = new SwerveModule.ModuleRequest()
            .withUpdatePeriod(parameters.updatePeriod);
        for (int i = 0; i < modulesToApply.length && i < ModuleStates.length;␣
↪++i) {
            /* apply the SwerveModuleState to the module */
            modulesToApply[i].apply(moduleRequest.withState(ModuleStates[i]));
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

### C++

```
struct ApplyModuleStates : public swerve::requests::SwerveRequest {
   std::vector<SwerveModuleState> ModuleStates;

   ctre::phoenix::StatusCode Apply(
      swerve::requests::SwerveRequest::ControlParameters const &parameters,
      std::span<std::unique_ptr<swerve::impl::SwerveModuleImpl> const>␣
↪modulesToApply
   ) override {
      auto moduleRequest = impl::SwerveModuleImpl::ModuleRequest{}
         .WithUpdatePeriod(parameters.updatePeriod);
      for (size_t i = 0; i < modulesToApply.size() && i <␣
↪ModuleStates.size(); ++i) {
         /* apply the SwerveModuleState to the module */
         modulesToApply[i]->Apply(moduleRequest.WithState(ModuleStates[i]));
      }
   }
};
```

### Python

```
class ApplyModuleStates(swerve.requests.SwerveRequest):
   def __init__(self):
      self.module_states: list[SwerveModuleState] = []

   def apply(
      self,
      parameters: swerve.SwerveControlParameters,
      modules_to_apply: list[swerve.SwerveModule]
   ) -> StatusCode:
      module_request = (
         SwerveModule.ModuleRequest()
         .with_update_period(parameters.update_period)
      )
      for (module, state) in zip(modules_to_apply, self.module_states):
         # apply the SwerveModuleState to the module
         module.apply(module_request.with_state(state))
      }
   }
```

### Swerve Requests with Direct Control

Swerve modules by default have some built-in control optimizations and support a limited set of control types. However, for something like the built-in SysId swerve requests, such high-level control may not be desirable.

As a result, `SwerveModule` also has `apply(ControlRequest drive, ControlRequest steer)` (Java, C++, Python) to directly apply control requests to the drive and steer motors. For

example, the built-in `SysIdSwerveSteerGains` (Java, C++, Python) request directly applies a `CoastOut` to the drive motor and a `VoltageOut` to the steer motor.

**Important**

We recommend **against** using this strategy in competition code, as it does not benefit from the built-in control optimizations.

### Java

```
private final CoastOut m_driveRequest = new CoastOut();
private final VoltageOut m_steerRequest = new VoltageOut(0);

@Override
public StatusCode apply(SwerveControlParameters parameters, SwerveModule<?, ?,
↪ ?>... modulesToApply) {
   for (int i = 0; i < modulesToApply.length; ++i) {
      /* directly apply the control requests to the drive and steer motors */
      modulesToApply[i].apply(m_driveRequest,␣
↪m_steerRequest.withOutput(VoltsToApply));
   }
   return StatusCode.OK;
}
```

### C++

```
ctre::phoenix::StatusCode Apply(
   SwerveRequest::ControlParameters const &parameters,
   std::span<std::unique_ptr<impl::SwerveModuleImpl> const> modulesToApply
) override {
   for (size_t i = 0; i < modulesToApply.size(); ++i) {
      /* directly apply the control requests to the drive and steer motors */
      modulesToApply[i]->Apply(controls::CoastOut{},␣
↪controls::VoltageOut{VoltsToApply});
   }
   return ctre::phoenix::StatusCode::OK;
}
```

### Python

```
def __init__(self):
   # ...
   self.__drive_request = CoastOut()
   self.__steer_request = VoltageOut(0)

def apply(
   self,
   parameters: swerve.SwerveControlParameters,
   modules_to_apply: list[swerve.SwerveModule]
) -> StatusCode:
```

(continues on next page)

(continued from previous page)

```
    for module in modules_to_apply:
        # directly apply the control requests to the drive and steer motors
        module.apply(
            self.__drive_request,
            self.__steer_request.with_output(self.volts_to_apply)
        )
    return StatusCode.OK
```

### Swerve Simulation

**Important**

Swerve simulation is only supported for FRC users.

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 a constant drive motor inertia regardless of the type of motion.

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

The behavior of the simulated drivetrain can be improved to more closely match hardware by running the simulation logic at a faster update rate, such as by using a WPILib `Notifier` as demonstrated below.

**Important**

When using `CommandSwerveDrivetrain` from our examples or Tuner X, this is already handled by the subsystem.

#### Java

```
private static final double kSimLoopPeriod = 0.004; // 4 ms
private Notifier m_simNotifier = null;
private double m_lastSimTime;

@Override
public void simulationInit() {
    m_lastSimTime = Utils.getCurrentTimeSeconds();

    /* Run simulation at a faster rate so PID gains behave more reasonably */
    m_simNotifier = new Notifier(() -> {
        final double currentTime = Utils.getCurrentTimeSeconds();
        double deltaTime = currentTime - m_lastSimTime;
        m_lastSimTime = currentTime;

        /* Use the measured time delta, get battery voltage from WPILib */
```

(continues on next page)

(continued from previous page)

```
      drivetrain.updateSimState(deltaTime,␣
↪RobotController.getBatteryVoltage());
   });
   m_simNotifier.startPeriodic(kSimLoopPeriod);
}
```

### C++

```
private:
   static constexpr units::second_t kSimLoopPeriod = 4_ms;
   std::unique_ptr<frc::Notifier> m_simNotifier;
   units::second_t m_lastSimTime;

public:
   void SimulationInit() override
   {
      m_lastSimTime = utils::GetCurrentTime();

      /* Run simulation at a faster rate so PID gains behave more reasonably␣
↪*/
      m_simNotifier = std::make_unique<frc::Notifier>([this] {
         units::second_t const currentTime = utils::GetCurrentTime();
         auto const deltaTime = currentTime - m_lastSimTime;
         m_lastSimTime = currentTime;

         /* Use the measured time delta, get battery voltage from WPILib */
         drivetrain.UpdateSimState(deltaTime,␣
↪frc::RobotController::GetBatteryVoltage());
      });
      m_simNotifier->StartPeriodic(kSimLoopPeriod);
   }
```

### Python

```
_SIM_LOOP_PERIOD: units.second = 0.004  # 4 ms

def __init__(self):
   self._sim_notifier: Notifier | None = None
   self._last_sim_time: units.second = 0.0
   # ...

def simulationInit(self):
   def _sim_periodic():
      current_time = utils.get_current_time_seconds()
      delta_time = current_time - self._last_sim_time
      self._last_sim_time = current_time

      # Use the measured time delta, get battery voltage from WPILib
      self.drivetrain.update_sim_state(delta_time, RobotController.
↪getBatteryVoltage())
```

(continues on next page)

(continued from previous page)

```
    # Run simulation at a faster rate so PID gains behave more reasonably
    self._last_sim_time = utils.get_current_time_seconds()
    self._sim_notifier = Notifier(_sim_periodic)
    self._sim_notifier.startPeriodic(self._SIM_LOOP_PERIOD)
```

### 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 drifts a bit when driving while rotating

A: Azimuth inertia and control latency is simulated. As a result, simulated swerve modules match the behavior of hardware in lagging behind the module targets, which can be improved by tuning the steer PID gains.

### Using the Swerve Drivetrain

In addition to *control* and *simulation*, the `SwerveDrivetrain` (Java, C++, Python) has many other APIs to manage the built-in pose estimator, telemetry, and more.

### Changing Neutral Mode

The neutral mode of the drive motors can be reconfigured at runtime using `configNeutralMode` (Java, C++, Python).

> **Tip**
>
> The neutral mode can be applied on construction by modifying the `DriveMotorInitialConfigs` and `SteerMotorInitialConfigs` provided to the `SwerveModuleConstants`.

#### Java

```
// The drivetrain was constructed in brake mode, switch to coast
drivetrain.configNeutralMode(NeutralModeValue.Coast);
```

#### C++

```
// The drivetrain was constructed in brake mode, switch to coast
drivetrain.ConfigNeutralMode(signals::NeutralModeValue::Coast);
```

#### Python

```
# The drivetrain was constructed in brake mode, switch to coast
self.drivetrain.config_neutral_mode(signals.NeutralModeValue.COAST)
```

### Using Field-Centric Control

In field-centric control, the robot is driven using velocities relative to the field. This makes it so the forward direction is constant (typically away from the driver) regardless of the robot orientation. There are two common field coordinate systems: Blue Alliance Perspective (which are absolute coordinates) and Operator Perspective (which are alliance-relative coordinates).

For field-centric control to behave correctly, the drivetrain needs to know which directions the driver and the robot are facing.

### Setting the Operator Perspective

The `OperatorPerspective` is typically used during teleop control, ensuring that forward (+X) is always away from the driver. The forward direction for `OperatorPerspective` can be set using `setOperatorPerspectiveForward` (Java, C++, Python). This tells the drivetrain which direction the **driver** is facing.

Looking at the Blue Alliance Perspective coordinates, facing away from the **blue alliance** is a heading of **0 degrees**. On the other hand, the red alliance is flipped, so facing away from the **red alliance** is a heading of **180 degrees**.

**Important**

When using `CommandSwerveDrivetrain` from our examples or Tuner X, this is already handled by the subsystem.

#### Java

```
/* Blue alliance sees forward as 0 degrees (toward red alliance wall) */
private static final Rotation2d kBlueAlliancePerspectiveRotation =␣
↪Rotation2d.kZero;
/* Red alliance sees forward as 180 degrees (toward blue alliance wall) */
private static final Rotation2d kRedAlliancePerspectiveRotation =␣
↪Rotation2d.k180deg;
/* Keep track if we've ever applied the operator perspective before or not */
private boolean m_hasAppliedOperatorPerspective = false;

@Override
public void periodic() {
   // Periodically try to apply the operator perspective
   // if we haven't yet or if we're currently disabled.
   if (!m_hasAppliedOperatorPerspective || DriverStation.isDisabled()) {
      DriverStation.getAlliance().ifPresent(allianceColor -> {
         setOperatorPerspectiveForward(
            allianceColor == Alliance.Red
               ? kRedAlliancePerspectiveRotation
               : kBlueAlliancePerspectiveRotation
         );
         m_hasAppliedOperatorPerspective = true;
      });
   }
}
```

### C++

```
/* Blue alliance sees forward as 0 degrees (toward red alliance wall) */
static constexpr frc::Rotation2d kBlueAlliancePerspectiveRotation{0_deg};
/* Red alliance sees forward as 180 degrees (toward blue alliance wall) */
static constexpr frc::Rotation2d kRedAlliancePerspectiveRotation{180_deg};
/* Keep track if we've ever applied the operator perspective before or not */
bool m_hasAppliedOperatorPerspective = false;

void Periodic() override
{
   // Periodically try to apply the operator perspective
   // if we haven't yet or if we're currently disabled.
   if (!m_hasAppliedOperatorPerspective || frc::DriverStation::IsDisabled()) {
      auto const allianceColor = frc::DriverStation::GetAlliance();
      if (allianceColor) {
         SetOperatorPerspectiveForward(
            *allianceColor == frc::DriverStation::Alliance::kRed
               ? kRedAlliancePerspectiveRotation
               : kBlueAlliancePerspectiveRotation
         );
         m_hasAppliedOperatorPerspective = true;
      }
   }
}
```

### Python

```
_BLUE_ALLIANCE_PERSPECTIVE_ROTATION = Rotation2d.fromDegrees(0)
"""Blue alliance sees forward as 0 degrees (toward red alliance wall)"""
_RED_ALLIANCE_PERSPECTIVE_ROTATION = Rotation2d.fromDegrees(180)
"""Red alliance sees forward as 180 degrees (toward blue alliance wall)"""

def __init__(self, ...):
   # ...

   self._has_applied_operator_perspective = False
   """Keep track if we've ever applied the operator perspective before or␣
↪not"""

def periodic(self):
   # Periodically try to apply the operator perspective
   # if we haven't yet or if we're currently disabled.
   if not self._has_applied_operator_perspective or DriverStation.
↪isDisabled():
      alliance_color = DriverStation.getAlliance()
      if alliance_color is not None:
         self.set_operator_perspective_forward(
            self._RED_ALLIANCE_PERSPECTIVE_ROTATION
            if alliance_color == DriverStation.Alliance.kRed
            else self._BLUE_ALLIANCE_PERSPECTIVE_ROTATION
         )
         self._has_applied_operator_perspective = True
```

### Setting the Robot Heading

After setting the operator perspective, the drivetrain also needs to know which direction the **robot** is facing.

> **Note**
>
> Many path planning libraries automatically reset the full pose of the robot, including heading, at the start of the path.

### Current Direction is Forward

If the robot is currently facing the driver's forward direction, call `seedFieldCentric()` (Java, C++, Python) to reset the heading.

> **Tip**
>
> The Tuner X generated swerve project and our examples bind `seedFieldCentric()` to the left bumper.

#### Java

```
// Reset the field-centric heading on left bumper press.
joystick.leftBumper().onTrue(drivetrain.runOnce(drivetrain::seedFieldCentric));
```

#### C++

```
// reset the field-centric heading on left bumper press
joystick.LeftBumper().OnTrue(drivetrain.RunOnce([this] {␣
↪drivetrain.SeedFieldCentric(); }));
```

#### Python

```
# reset the field-centric heading on left bumper press
self._joystick.leftBumper().onTrue(
    self.drivetrain.runOnce(self.drivetrain.seed_field_centric)
)
```

### Angle Relative to Forward

If the robot is facing some other angle relative to the driver's forward direction, call `seedFieldCentric(Rotation2d)` (Java, C++, Python) with the relative angle. For example, if the robot is facing left, then pass in an angle of +90 degrees (counter-clockwise).

> **Tip**
>
> The Tuner X generated swerve project calls `seedFieldCentric(Rotation2d)` at the start of the default autonomous command.

#### Java

```
// Reset the field-centric heading so the robot is facing left (90 deg CCW)
drivetrain.seedFieldCentric(Rotation2d.kCCW_90deg);
```

#### C++

```
// Reset the field-centric heading so the robot is facing left (+90 deg)
drivetrain.SeedFieldCentric(frc::Rotation2d{90_deg});
```

#### Python

```
# Reset the field-centric heading so the robot is facing left (+90 deg)
self.drivetrain.seed_field_centric(Rotation2d.fromDegrees(90))
```

### Blue Alliance Heading or Pose

When using a path planning library such as PathPlanner or Choreo, the paths often operate using the `BlueAlliancePerspective` and reset the robot's pose at the start of the path. Vision libraries similarly often operate using a `BlueAlliancePerspective` heading or pose.

The robot's heading can be reset to a `BlueAlliancePerspective` heading using `resetRotation(Rotation2d)` (Java, C++, Python), and the pose can be reset using `resetPose(Pose2d)` (Java, C++, Python).

> **Tip**
>
> PathPlanner and Choreo can call `resetPose` automatically at the start of the autonomous path.

#### Java

```
// Reset the robot's heading to 0 deg (away from the Blue Alliance wall)
drivetrain.resetRotation(Rotation2d.kZero);
```

```
// Reset the robot's pose to the initial pose of the autonomous path
drivetrain.resetPose(initialPose);
```

#### C++

```
// Reset the robot's heading to 0 deg (away from the Blue Alliance wall)
drivetrain.ResetRotation(frc::Rotation2d{});
```

```
// Reset the robot's pose to the initial pose of the autonomous path
drivetrain.ResetPose(initialPose);
```

#### Python

```
# Reset the robot's heading to 0 deg (away from the Blue Alliance wall)
drivetrain.reset_rotation(Rotation2d())
```

```
# Reset the robot's pose to the initial pose of the autonomous path
self.drivetrain.reset_pose(initial_pose)
```

### Odometry and State

The `SwerveDrivetrain` has a built-in pose estimator running on a separate odometry thread (250 Hz on CANivore®, 100 Hz on roboRIO). This significantly improves the accuracy and consistency of odometry and robot pose estimation.

Information about the robot's state can be retrieved using `getState()` (Java, C++, Python), and a thread-safe copy can be retrieved using `getStateCopy()` (Java, Python). This returns a `SwerveDriveState` (Java, C++, Python) instance that includes information such as the pose estimate, module states, and chassis speeds.

#### Java

```
var state = drivetrain.getState();
// pull out the pose estimate and chassis speeds
Pose2d pose = state.Pose;
ChassisSpeeds speeds = state.Speeds;
```

#### C++

```
auto state = drivetrain.GetState();
// pull out the pose estimate and chassis speeds
frc::Pose2d pose = state.Pose;
frc::ChassisSpeeds speeds = state.Speeds;
```

#### Python

```
state = self.drivetrain.get_state()
# pull out the pose estimate and chassis speeds
pose = state.pose
speeds = state.speeds
```

### Drivetrain Telemetry

The state of the drivetrain can also be telemeterized inline with odometry updates, ensuring that all information is captured in logs. A telemetry function that accepts the latest `SwerveDriveState` as a parameter can be registered using `registerTelemetry` (Java, C++, Python).

**Tip**

The Tuner X generated swerve project and our examples have a `Telemetry` class that is already registered with the drivetrain.

### Java

```
public Robot() {
   drivetrain.registerTelemetry(this::telemeterize);
}

/** Accept the swerve drive state and telemeterize it to SignalLogger. */
public void telemeterize(SwerveDriveState state) {
   SignalLogger.writeStruct("DriveState/Pose", Pose2d.struct, state.Pose);
   SignalLogger.writeStruct("DriveState/Speeds", ChassisSpeeds.struct,␣
↪state.Speeds);
   SignalLogger.writeStructArray("DriveState/ModuleStates",␣
↪SwerveModuleState.struct, state.ModuleStates);
   SignalLogger.writeStructArray("DriveState/ModuleTargets",␣
↪SwerveModuleState.struct, state.ModuleTargets);
   SignalLogger.writeStructArray("DriveState/ModulePositions",␣
↪SwerveModulePosition.struct, state.ModulePositions);
   SignalLogger.writeDouble("DriveState/OdometryPeriod", state.OdometryPeriod,
↪ "seconds");
   SignalLogger.writeInteger("DriveState/FailedDaqs", state.FailedDaqs);
}
```

### C++

```
Robot()
{
   drivetrain.RegisterTelemetry(
      [this](auto const &state) { Telemeterize(state); }
   );
}

/** Accept the swerve drive state and telemeterize it to SignalLogger. */
void Telemeterize(subsystems::TunerSwerveDrivetrain::SwerveDriveState const &
↪state)
{
   SignalLogger::WriteStruct("DriveState/Pose", state.Pose);
   SignalLogger::WriteStruct("DriveState/Speeds", state.Speeds);
   SignalLogger::WriteStructArray<frc::SwerveModuleState>(
↪"DriveState/ModuleStates", state.ModuleStates);
   SignalLogger::WriteStructArray<frc::SwerveModuleState>(
↪"DriveState/ModuleTargets", state.ModuleTargets);
   SignalLogger::WriteStructArray<frc::SwerveModulePosition>(
↪"DriveState/ModulePositions", state.ModulePositions);
   SignalLogger::WriteValue("DriveState/OdometryPeriod",␣
↪state.OdometryPeriod);
   SignalLogger::WriteInteger("DriveState/FailedDaqs", state.FailedDaqs);
}
```

**Python**

```
def __init__(self):
    # ...
    self.drivetrain.register_telemetry(self.telemeterize)

def telemeterize(self, state: swerve.SwerveDrivetrain.SwerveDriveState):
    """
    Accept the swerve drive state and telemeterize it to SignalLogger.
    """
    SignalLogger.write_struct(
        "DriveState/Pose", Pose2d, state.pose
    )
    SignalLogger.write_struct(
        "DriveState/Speeds", ChassisSpeeds, state.speeds
    )
    SignalLogger.write_struct_array(
        "DriveState/ModuleStates", SwerveModuleState, state.module_states
    )
    SignalLogger.write_struct_array(
        "DriveState/ModuleTargets", SwerveModuleState, state.module_targets
    )
    SignalLogger.write_struct_array(
        "DriveState/ModulePositions", SwerveModulePosition, state.
↪module_positions
    )
    SignalLogger.write_double(
        "DriveState/OdometryPeriod", state.odometry_period, "seconds"
    )
    SignalLogger.write_integer("DriveState/FailedDaqs", state.failed_daqs)
```

## 20.2 Differential Overview

Phoenix 6 has two robust differential mechanism APIs taking advantage of the motor controllers' onboard differential controls. These APIs greatly simplify the setup and usage of common differential mechanisms, ranging from a differential wrist to a two-gearbox elevator without mechanical linkage.

### 20.2.1 What is a Differential Mechanism?

A differential mechanism has two axes of motion, where the position along each axis is determined by two motors in separate gearboxes:

- Driving both motors in a common direction causes the mechanism to move forward/reverse, up/down, etc.
  - This is the **Average axis**: position is determined by the average of the two motors' positions.
- Driving the motors in opposing directions causes the mechanism to twist or rotate left/right.
  - This is the **Difference axis**: rotation is determined by half the difference of the two motors' positions.

As an example, a differential drivetrain has a few motors on each side of the robot. Driving both sides of the robot in the "forward" direction causes the robot to move forward. However, driving the left side "forward" and the right side "reverse" causes the robot to turn right.

Another example is a two-gearbox elevator without mechanical linkage between the two sides. If the two sides of the elevator are not driven together, the elevator carriage twists, potentially breaking if the twist is too extreme. As a result, the elevator can be treated as a differential mechanism that always targets a difference of 0.

In a more advanced setup, a remote sensor can be used on the Difference axis as an absolute sensor source. For example, a differential drivetrain can use the yaw of a Pigeon® 2.0 to target an absolute heading.

#### Differential Leader and Follower

In a differential mechanism, one of the motor controllers is selected as the "differential leader", while the other is selected as the "differential follower".

- The **leader** is responsible for running all closed-loop calculations and applies `Average + Difference` to its output.
- The **follower** reports its position and velocity to the leader and applies `Average - Difference` to its output.

The selection of the leader and follower motor controllers is **only important** when using a **remote sensor** on the difference axis. For example, consider a differential drivetrain using the yaw of a Pigeon® 2.0.

The Pigeon® 2.0 is counter-clockwise positive, so the robot should rotate counter-clockwise (left) from a positive output on the Difference axis. This occurs when the right side drives forward (positive) and the left side drives reverse (negative). As a result, the right motor controller should be selected as the leader.

### 20.2.2 Hardware Requirements

All differential mechanism APIs require at least 2 Talon FX® or Talon FXS® motor controllers, one on each side of the mechanism. Optionally, a remote CANcoder®, CANdi®, or Pigeon® 2.0 can be used on the Difference axis as an absolute sensor source.

> **Note**
>
> Both motor controllers must be of the same type.

### 20.2.3 Overview of the API

There are two differential mechanism APIs:

- `DifferentialMechanism` (Java, C++, Python)
  - Requires Phoenix Pro and CANivore®.
  - Full functionality, including full support for feedforwards and custom motion profiles.
  - Difference axis supports open-loop control and position/velocity closed-loop control.
  - Supports all control output type.

- `SimpleDifferentialMechanism` (Java, C++, Python)
  - Free and supports CAN 2.0.
  - Limited functionality.
  - Difference axis only supports position closed-loop control.
  - Only supports Duty Cycle and Voltage control output types.

Both types of mechanism are constructed using a `DifferentialMotorConstants` (Java, C++, Python) object.

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

### Differential Mechanism Setup

The differential mechanism APIs are constructed by setting up a `DifferentialMotorConstants` (Java, C++, Python) object. This includes necessary information such as the leader motor controllers' IDs, gear ratio on the differential, and alignment of the two sides of the mechanism.

**Important**

The differential mechanism only manages one motor controller from each gearbox. Any other motors in the gearboxes should be separately constructed and configured to follow their corresponding leader.

### Defining the Motor Constants

An example configuration for a differential wrist is shown below with the following setup:

- A 3:1 gear ratio on the average axis
- An additional 2:1 gear ratio on the differential axis
- The two motor directions are aligned (ignoring inverts)
- Closed-loop control and relevant status signals run at 200 Hz

**Warning**

Many of these constants, including the PID gains, are specific to this example and **will not work** on your mechanism.

#### Java

```
// Average axis gains typically go in Slot 0
private static final Slot0Configs averageGains = new Slot0Configs()
   .withKP(20).withKI(0).withKD(0.1)
   .withKG(0.2).withKS(0.1).withKV(0.36).withKA(0)
   .withGravityType(GravityTypeValue.Arm_Cosine);
// Difference axis gains typically go in Slot 1
private static final Slot1Configs differenceGains = new Slot1Configs()
   .withKP(30).withKI(0).withKD(0.1)
```

(continues on next page)

(continued from previous page)

```
    .withKS(0.1).withKV(0.72);

private static final double kAverageGearRatio = 3.0;
private static final double kDifferenceGearRatio = 2.0;

// Initial configs for the differential leader and follower motor controllers.
// Some configs will be overwritten; check the `with*InitialConfigs()` API␣
↪documentation.
private static final TalonFXConfiguration leaderInitialConfigs = new␣
↪TalonFXConfiguration()
    .withMotorOutput(
        new MotorOutputConfigs()
            .withNeutralMode(NeutralModeValue.Brake)
    )
    .withCurrentLimits(
        new CurrentLimitsConfigs()
            .withStatorCurrentLimit(Amps.of(80))
            .withStatorCurrentLimitEnable(true)
    )
    .withFeedback(
        new FeedbackConfigs()
            .withSensorToMechanismRatio(kAverageGearRatio)
    )
    .withClosedLoopGeneral(
        new ClosedLoopGeneralConfigs()
           // differential wrist is continuous on the difference axis
            .withDifferentialContinuousWrap(true)
    )
    .withSlot0(averageGains)
    .withSlot1(differenceGains)
    .withMotionMagic(
        new MotionMagicConfigs()
            .withMotionMagicCruiseVelocity(80)
            .withMotionMagicAcceleration(320)
    );
private static final TalonFXConfiguration followerInitialConfigs = new␣
↪TalonFXConfiguration()
    .withFeedback(
        new FeedbackConfigs()
            .withSensorToMechanismRatio(kAverageGearRatio)
    );

// CAN bus that the devices are located on;
// All mechanism devices must share the same CAN bus
private static final CANBus kCANBus = new CANBus("canivore", "./
↪logs/example.hoot");

private static final DifferentialMotorConstants<TalonFXConfiguration>␣
↪differentialConstants =
    new DifferentialMotorConstants<TalonFXConfiguration>()
        .withCANBusName(kCANBus.getName())
        .withLeaderId(0)
```

(continues on next page)

(continued from previous page)

```
        .withFollowerId(1)
        .withAlignment(MotorAlignmentValue.Aligned)
        .withSensorToDifferentialRatio(kDifferenceGearRatio)
        .withClosedLoopRate(200.0)
        .withLeaderInitialConfigs(leaderInitialConfigs)
        .withFollowerInitialConfigs(followerInitialConfigs)
        .withFollowerUsesCommonLeaderConfigs(true);
```

### C++

```
// Average axis gains typically go in Slot 0
static constexpr configs::Slot0Configs averageGains = configs::Slot0Configs{}
   .WithKP(10).WithKI(0).WithKD(0.1)
   .WithKG(0.2).WithKS(0.1).WithKV(0.12).WithKA(0)
   .WithGravityType(signals::GravityTypeValue::Arm_Cosine);
// Difference axis gains typically go in Slot 1
static constexpr configs::Slot1Configs differenceGains =␣
↪configs::Slot1Configs{}
   .WithKP(10).WithKI(0).WithKD(0.1)
   .WithKS(0.1).WithKV(0.12);

static constexpr units::scalar_t kAverageGearRatio = 3.0;
static constexpr units::scalar_t kDifferenceGearRatio = 2.0;

// Initial configs for the differential leader and follower motor controllers.
// Some configs will be overwritten; check the `With*InitialConfigs()` API␣
↪documentation.
static constexpr configs::TalonFXConfiguration leaderInitialConfigs =␣
↪configs::TalonFXConfiguration{}
   .WithMotorOutput(
      configs::MotorOutputConfigs{}
         .WithNeutralMode(signals::NeutralModeValue::Brake)
   )
   .WithCurrentLimits(
      configs::CurrentLimitsConfigs{}
         .WithStatorCurrentLimit(80_A)
         .WithStatorCurrentLimitEnable(true)
   )
   .WithFeedback(
      configs::FeedbackConfigs{}
         .WithSensorToMechanismRatio(kAverageGearRatio)
   )
   .withClosedLoopGeneral(
      configs::ClosedLoopGeneralConfigs{}
         // differential wrist is continuous on the difference axis
         .WithDifferentialContinuousWrap(true)
   )
   .WithSlot0(averageGains)
   .WithSlot1(differenceGains)
   .WithMotionMagic(
      configs::MotionMagicConfigs{}
```

(continues on next page)

(continued from previous page)

```
        .WithMotionMagicCruiseVelocity(80_tps)
        .WithMotionMagicAcceleration(320_tr_per_s_sq)
   );
static constexpr configs::TalonFXConfiguration followerInitialConfigs =␣
↪configs::TalonFXConfiguration{}
   .WithFeedback(
      configs::FeedbackConfigs{}
         .WithSensorToMechanismRatio(kAverageGearRatio)
   );

// CAN bus that the devices are located on;
// All mechanism devices must share the same CAN bus
static constexpr std::string_view kCANBusName = "canivore";
static inline CANBus kCANBus{kCANBusName, "./logs/example.hoot"};

static constexpr mechanisms::DifferentialMotorConstants differentialConstants␣
↪=
   mechanisms::DifferentialMotorConstants<configs::TalonFXConfiguration>{}
      .WithCANBusName(kCANBusName)
      .WithLeaderId(0)
      .WithFollowerId(1)
      .WithAlignment(signals::MotorAlignmentValue::Opposed)
      .WithSensorToDifferentialRatio(kDifferenceGearRatio)
      .WithClosedLoopRate(200_Hz)
      .WithLeaderInitialConfigs(leaderInitialConfigs)
      .WithFollowerInitialConfigs(followerInitialConfigs)
      .WithFollowerUsesCommonLeaderConfigs(true);
```

### Python

```
# Average axis gains typically go in Slot 0
_average_gains = (
   configs.Slot0Configs()
   .with_k_p(10)
   .with_k_i(0)
   .with_k_d(0.1)
   .with_k_g(0.2)
   .with_k_s(0.1)
   .with_k_v(0.12)
   .with_k_a(0)
   .with_gravity_type(signals.GravityTypeValue.ARM_COSINE)
)
# Difference axis gains typically go in Slot 1
_difference_gains = (
   configs.Slot1Configs()
   .with_k_p(10)
   .with_k_i(0)
   .with_k_d(0.1)
   .with_k_s(0.1)
   .with_k_v(0.12)
)
```

(continues on next page)

(continued from previous page)

```
_average_gear_ratio = 3.0
_difference_gear_ratio = 2.0

# Initial configs for the differential leader and follower motor controllers.
# Some configs will be overwritten; check the `with*InitialConfigs()` API␣
↪documentation.
_leader_initial_configs = (
   configs.TalonFXConfiguration()
   .with_motor_output(
      configs.MotorOutputConfigs()
      .with_neutral_mode(signals.NeutralModeValue.BRAKE)
   )
   .with_current_limits(
      configs.CurrentLimitsConfigs()
      .with_stator_current_limit(80.0)
      .with_stator_current_limit_enable(True)
   )
   .with_feedback(
      configs.FeedbackConfigs()
      .with_sensor_to_mechanism_ratio(_average_gear_ratio)
   )
   .with_closed_loop_general(
      configs.ClosedLoopGeneralConfigs()
      # differential wrist is continuous on the difference axis
      .with_differential_continuous_wrap(True)
   )
   .with_slot0(_average_gains)
   .with_slot1(_difference_gains)
   .with_motion_magic(
      configs.MotionMagicConfigs()
      .with_motion_magic_cruise_velocity(80)
      .with_motion_magic_acceleration(320)
   )
)
_follower_initial_configs = (
   configs.TalonFXConfiguration()
   .with_feedback(
      configs.FeedbackConfigs()
      .with_sensor_to_mechanism_ratio(_average_gear_ratio)
   )
)

# CAN bus that the devices are located on;
# All mechanism devices must share the same CAN bus
self._canbus = CANBus("canivore", "./logs/example.hoot")

self._differential_constants: mechanisms.DifferentialMotorConstants[configs.
↪TalonFXConfiguration] = (
   mechanisms.DifferentialMotorConstants()
   .with_can_bus_name(_canbus.name)
   .with_leader_id(0)
```

(continues on next page)

(continued from previous page)

```
    .with_follower_id(1)
    .with_alignment(signals.MotorAlignmentValue.OPPOSED)
    .with_sensor_to_differential_ratio(_difference_gear_ratio)
    .with_closed_loop_rate(200.0)
    .with_leader_initial_configs(_leader_initial_configs)
    .with_follower_initial_configs(_follower_initial_configs)
    .with_follower_uses_common_leader_configs(True)
)
```

### Building the Mechanism

The differential motor constants can then be used to construct the `DifferentialMechanism` (Java, C++, Python) or `SimpleDifferentialMechanism` (Java, C++, Python). The mechanisms have constructor overloads to provide a remote sensor for the Difference axis, such as the yaw of a Pigeon® 2.0.

#### `DifferentialMechanism`

#### Java

```
// Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
private final DifferentialMechanism<TalonFX> diffMech =
    new DifferentialMechanism<TalonFX>(TalonFX::new, differentialConstants);

// Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
private final Pigeon2 pigeon2 = new Pigeon2(0, kCANBus);
private final DifferentialMechanism<TalonFX> diffMech =
    new DifferentialMechanism<TalonFX>(
        TalonFX::new, differentialConstants,
        pigeon2, DifferentialPigeon2Source.Yaw
    );
```

#### C++

```
// Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
mechanisms::DifferentialMechanism<hardware::TalonFX>␣
↪diffMech{differentialConstants};

// Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
hardware::Pigeon2 pigeon2{0, kCANBus};
mechanisms::DifferentialMechanism<hardware::TalonFX> diffMech{
    differentialConstants, pigeon2,
    mechanisms::DifferentialPigeon2Source::Yaw
};
```

#### Python

```
# Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
self._diff_mech: mechanisms.DifferentialMechanism[hardware.TalonFX] = (
   mechanisms.DifferentialMechanism(hardware.TalonFX, self._
↪differential_constants)
)

# Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
self._pigeon2 = hardware.Pigeon2(0, self._canbus)
self._diff_mech: mechanisms.DifferentialMechanism[hardware.TalonFX] = (
   mechanisms.DifferentialMechanism(
      hardware.TalonFX,
      self._differential_constants
      self._pigeon2,
      mechanisms.DifferentialPigeon2Source.YAW
   )
)
```

#### SimpleDifferentialMechanism

#### Java

```
// Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
private final SimpleDifferentialMechanism<TalonFX> diffMech =
   new SimpleDifferentialMechanism<TalonFX>(TalonFX::new,␣
↪differentialConstants);

// Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
private final Pigeon2 pigeon2 = new Pigeon2(0, kCANBus);
private final SimpleDifferentialMechanism<TalonFX> diffMech =
   new SimpleDifferentialMechanism<TalonFX>(
      TalonFX::new, differentialConstants,
      pigeon2, DifferentialPigeon2Source.Yaw
   );
```

#### C++

```
// Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
mechanisms::SimpleDifferentialMechanism<hardware::TalonFX>␣
↪diffMech{differentialConstants};

// Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
hardware::Pigeon2 pigeon2{0, kCANBus};
mechanisms::SimpleDifferentialMechanism<hardware::TalonFX> diffMech{
   differentialConstants, pigeon2,
   mechanisms::DifferentialPigeon2Source::Yaw
};
```

### Python

```
# Construct the mechanism, Difference axis uses half the difference between␣
↪the motors
self._diff_mech: mechanisms.SimpleDifferentialMechanism[hardware.TalonFX] = (
   mechanisms.SimpleDifferentialMechanism(hardware.TalonFX, self._
↪differential_constants)
)

# Construct the mechanism, Difference axis uses the yaw of a Pigeon 2
self._pigeon2 = hardware.Pigeon2(0, self._canbus)
self._diff_mech: mechanisms.SimpleDifferentialMechanism[hardware.TalonFX] = (
   mechanisms.SimpleDifferentialMechanism(
      hardware.TalonFX,
      self._differential_constants
      self._pigeon2,
      mechanisms.DifferentialPigeon2Source.YAW
   )
)
```

### Using the Differential Mechanism API

`DifferentialMechanism` (Java, C++, Python) and `SimpleDifferentialMechanism` (Java, C++, Python) **automatically** set up and applies **all configs** to the two provided motor controllers on construction. Additionally, the mechanisms provide functions to control the mechanism, pull out relevant signals, and automatically detect and disable on fault conditions.

**Tip**

The configs applied to the motor controllers, including invert and neutral mode, can be adjusted using the `LeaderInitialConfigs` and `FollowerInitialConfigs`. For more information, see *Differential Mechanism Setup*.

### Running Control Requests

The differential mechanism can be controlled by calling `setControl` with one of the supported control requests.

#### `DifferentialMechanism`

The full `DifferentialMechanism` (requires Phoenix Pro and CANivore®) accepts two separate control requests. The first is run on the Average axis, while the second is run on the Difference axis. Both control requests must use the same *control output type*. All control output types are supported, as are all parameters within the two provided control requests. However, note that Motion Magic® is not supported on the Difference axis.

**Note**

Common parameters such as `UseTimesync` only need to be set on the Average axis control request.

### Java

```
// Run Motion Magic on the Average axis, Position on the Difference axis
final MotionMagicVoltage avgRequest = new MotionMagicVoltage(0).withSlot(0);
final PositionVoltage diffRequest = new PositionVoltage(0).withSlot(1);

// Apply a TrapezoidProfile to the Difference axis for smoother motion
final TrapezoidProfile diffProfile = new TrapezoidProfile(
   new TrapezoidProfile.Constraints(80, 320)
);
TrapezoidProfile.State diffSetpoint = new TrapezoidProfile.State();
TrapezoidProfile.State diffGoal = new TrapezoidProfile.State();

// Update the Difference setpoint and control the mechanism
diffGoal = new TrapezoidProfile.State(5.0, 0.0);
diffSetpoint = diffProfile.calculate(0.020, diffSetpoint, diffGoal);

diffMech.setControl(
   avgRequest.withPosition(10.0),
   diffRequest.withPosition(diffSetpoint.position)
      .withVelocity(diffSetpoint.velocity)
);
```

### C++

```
// Run Motion Magic on the Average axis, Position on the Difference axis
controls::MotionMagicVoltage avgRequest = controls::MotionMagicVoltage{0_
↪tr}.WithSlot(0);
controls::PositionVoltage diffRequest = controls::PositionVoltage{0_
↪tr}.WithSlot(1);

// Apply a TrapezoidProfile to the Difference axis for smoother motion
frc::TrapezoidProfile<units::turn_t> diffProfile{{80_tps, 320_tr_per_s_sq}};
frc::TrapezoidProfile<units::turn_t>::State diffSetpoint{};
frc::TrapezoidProfile<units::turn_t>::State diffGoal{};

/* Update the Difference setpoint and control the mechanism */
diffGoal = frc::TrapezoidProfile<units::turn_t>::State{5_tr, 0_tps};
diffSetpoint = diffProfile.Calculate(20_ms, diffSetpoint, diffGoal);

diffMech.SetControl(
   avgRequest.WithPosition(10_tr),
   diffRequest.WithPosition(diffSetpoint.position)
      .WithVelocity(diffSetpoint.velocity)
);
```

### Python

```
# Run Motion Magic on the Average axis, Position on the Difference axis
self._avg_request = controls.MotionMagicVoltage(0).with_slot(0)
self._diff_request = controls.PositionVoltage(0).with_slot(1)
```

(continues on next page)

(continued from previous page)

```
# Apply a TrapezoidProfile to the Difference axis for smoother motion
self._diff_profile = TrapezoidProfile(
   TrapezoidProfile.Constraints(80, 320)
)
self._diff_setpoint = TrapezoidProfile.State()
self._diff_goal = TrapezoidProfile.State()

# Update the Difference setpoint and control the mechanism
self._diff_goal = TrapezoidProfile.State(5.0, 0.0)
self._diff_setpoint = self._diff_profile.calculate(0.020, self._
↪diff_setpoint, self._diff_goal)

self._diff_mech.set_control(
   self._avg_request.with_position(10.0),
   self._diff_request
   .with_position(self._diff_setpoint.position)
   .with_velocity(self._diff_setpoint.velocity)
)
```

#### SimpleDifferentialMechanism

The limited `SimpleDifferentialMechanism` accepts a single `Differential*` control request. These control requests contain a limited number of parameters compared to the non-differential requests. For example, `DifferentialPositionVoltage` does not support `FeedForward` or `Velocity`. Additionally, all of these control requests run position PID on the Difference axis, and only Duty Cycle and Voltage *control output types* are supported.

The `Average*` parameters specify the targets for the Average axis, while the `Differential*` parameters specify the targets for the Difference axis.

#### Java

```
// Run Motion Magic on the Average axis, unprofiled Position on the␣
↪Difference axis
// Defaults to Slot 0 on Average, Slot 1 on Differential
final DifferentialMotionMagicVoltage request = new␣
↪DifferentialMotionMagicVoltage(0, 0);

// Apply the control to the mechanism
simpleDiffMech.setControl(
   request.withAveragePosition(10.0)
      .withDifferentialPosition(5.0)
);
```

#### C++

```
// Run Motion Magic on the Average axis, unprofiled Position on the␣
↪Difference axis
// Defaults to Slot 0 on Average, Slot 1 on Differential
controls::DifferentialMotionMagicVoltage request{0_tr, 0_tr};
```

(continues on next page)

(continued from previous page)

```
// Apply the control to the mechanism
simpleDiffMech.SetControl(
   request.WithAveragePosition(10_tr)
      .WithDifferentialPosition(5_tr)
);
```

#### Python

```
# Run Motion Magic on the Average axis, unprofiled Position on the Difference␣
↪axis
# Defaults to Slot 0 on Average, Slot 1 on Differential
self._request = controls.DifferentialMotionMagicVoltage(0, 0)

# Apply the control to the mechanism
self._simple_diff_mech.set_control(
   self._request
   .with_average_position(10.0)
   .with_differential_position(5.0)
)
```

### Disabling Output

The differential mechanism also provides a few easy ways to disable output:

- `setNeutralOut()` applies the configured neutral mode (brake or coast) to both motors.
- `setCoastOut()` forces both motors to coast by applying a `CoastOut` request.
- `setStaticBrake()` forces both motors to brake by applying a `StaticBrake` request.

#### Java

```
// Disable mechanism output using the configured neutral mode
diffMech.setNeutralOut();
```

#### C++

```
// Disable mechanism output using the configured neutral mode
diffMech.SetNeutralOut();
```

#### Python

```
# Disable mechanism output using the configured neutral mode
self._diff_mech.set_neutral_out()
```

### Set Neutral Mode and Position

The neutral mode of the mechanism can be changed after construction by calling `configNeutralMode` (Java, C++, Python).

> **Tip**
>
> The neutral mode can be applied on construction by modifying the `LeaderInitialConfigs` provided to the `DifferentialMotorConstants`.

Additionally, the position of the mechanism can be reset using `setPosition` (Java, C++, Python), which accepts the new Average axis and Difference axis positions (default 0). This can be useful for recalibrating the mechanism position. Note that any remote sensor on the Difference axis will not be affected by `setPosition` and can be adjusted separately.

#### Java

```
// The mechanism was constructed in brake mode, switch to coast
diffMech.configNeutralMode(NeutralModeValue.Coast);

// Reset the mechanism positions to 0
diffMech.setPosition(Rotations.of(0), Rotations.of(0));
```

#### C++

```
// The mechanism was constructed in brake mode, switch to coast
diffMech.ConfigNeutralMode(signals::NeutralModeValue::Coast);

// Reset the mechanism positions to 0
diffMech.SetPosition(0_tr, 0_tr);
```

#### Python

```
# The mechanism was constructed in brake mode, switch to coast
self._diff_mech.config_neutral_mode(signals.NeutralModeValue.COAST)

# Reset the mechanism positions to 0
self._diff_mech.set_position(0.0, 0.0)
```

### Fetching Status Signals

Status signals relevant to the mechanism, such as the `AveragePosition` and `DifferentialPosition`, can be directly fetched from the mechanism. For other status signals, such as supply current, the differential leader and follower motor controllers can be pulled out from the mechanism using `getLeader()` and `getFollower()`.

#### Java

```
// Pull out some StatusSignals from the mechanism
var avgPosition = diffMech.getAveragePosition();
var diffPosition = diffMech.getDifferentialPosition();
var avgClosedLoopRef = diffMech.getAverageClosedLoopReference();
var diffClosedLoopRef = diffMech.getDifferentialClosedLoopReference();
```

(continues on next page)

(continued from previous page)

```
// Also pull out supply and torque current from both motors
var leaderSupplyCurrent = diffMech.getLeader().getSupplyCurrent();
var leaderTorqueCurrent = diffMech.getLeader().getTorqueCurrent();
var followerSupplyCurrent = diffMech.getFollower().getSupplyCurrent();
var followerTorqueCurrent = diffMech.getFollower().getTorqueCurrent();
```

#### C++

```
// Pull out some StatusSignals from the mechanism
auto &avgPosition = diffMech.GetAveragePosition();
auto &diffPosition = diffMech.GetDifferentialPosition();
auto &avgClosedLoopRef = diffMech.GetAverageClosedLoopReference();
auto &diffClosedLoopRef = diffMech.GetDifferentialClosedLoopReference();

// Also pull out supply and torque current from both motors
auto &leaderSupplyCurrent = diffMech.GetLeader().GetSupplyCurrent();
auto &leaderTorqueCurrent = diffMech.GetLeader().GetTorqueCurrent();
auto &followerSupplyCurrent = diffMech.GetFollower().GetSupplyCurrent();
auto &followerTorqueCurrent = diffMech.GetFollower().GetTorqueCurrent();
```

#### Python

```
# Pull out some StatusSignals from the mechanism
avg_position = self._diff_mech.get_average_position()
diff_position = self._diff_mech.get_differential_position()
avg_closed_loop_ref = self._diff_mech.get_average_closed_loop_reference()
diff_closed_loop_ref = self._diff_mech.
 ↪get_differential_closed_loop_reference()

# Also pull out supply and torque current from both motors
leader_supply_current = self._diff_mech.leader.get_supply_current()
leader_torque_current = self._diff_mech.leader.get_torque_current()
follower_supply_current = self._diff_mech.follower.get_supply_current()
follower_torque_current = self._diff_mech.follower.get_torque_current()
```

### Automatic Fault Protection

The differential mechanisms provide an optional `periodic()` (Java, C++, Python) function that, when called periodically, automatically detects dangerous fault conditions and disables the mechanism to prevent damage. The state of the mechanism can be fetched using `getMechanismState()` (Java, C++, Python).

> **Note**
>
> This section only applies when calling `periodic()`.

Some faults temporarily disable the mechanism until they are naturally resolved. For example, the mechanism temporarily disables when one of the motor controllers briefly disconnects from CAN and re-enables once it returns. `isDisabled()` (or `MechanismState.Disabled`) can

be used to check if the mechanism is disabled, and `getDisabledReason()` (Java, C++, Python) reports the reason for the disable.

Other faults are higher in severity and require user action before the mechanism can safely re-enable. For example, the mechanism disables when one of the motor controllers is power cycled, as its position is likely invalid and requires a call to *setPosition*. `requiresUserAction()` (or `MechanismState.RequiresUserAction`) can be used to check for these fault conditions, and `getRequiresUserReason()` (Java, C++, Python) reports the reason that user action is required. Once the mechanism has been confirmed to be in a safe state, the mechanism can be re-enabled using `clearUserRequirement()` (Java, C++, Python).

**Note**

`isDisabled()` also reports true when `requiresUserAction()` reports true.

#### Java

```
// If we encounter a critical fault condition, schedule a
// WPILib Command to recalibrate our zero and re-enable
new Trigger(diffMech::requiresUserAction).onTrue(
   calibrateZero().finallyDo(diffMech::clearUserRequirement)
);
```

#### C++

```
// If we encounter a critical fault condition, schedule a
// WPILib Command to recalibrate our zero and re-enable
frc2::Trigger{[this] { return diffMech.RequiresUserAction(); }}.OnTrue(
   CalibrateZero().FinallyDo([this] { diffMech.ClearUserRequirement(); })
);
```

#### Python

```
# If we encounter a critical fault condition, schedule a
# WPILib Command to recalibrate our zero and re-enable
Trigger(self._diff_mech.requires_user_action).onTrue(
   self.calibrate_zero().finallyDo(self._diff_mech.clear_user_requirement)
)
```

### Tuning a Differential Mechanism

**Important**

This article assumes that you are familiar with tuning PID on a single-axis mechanism. For more information, see *Closed-Loop Overview*.

Because a differential mechanism has *two axes of motion*, it also needs separate sets of PID gains for the Average axis (typically Slot 0) and the Difference axis (typically Slot 1). These gains are applied to the *differential leader*.

**Warning**

Tuner X only supports the `SimpleDifferentialMechanism` control requests. For full functionality while tuning, use the `DifferentialMechanism` API for control and Tuner X for plotting and configs. See *Using the Differential Mechanism API* for more information.

Tuning a differential mechanism typically involves three steps:

### Step 1: Weak kP on the Difference Axis

When tuning closed-loop control on a differential mechanism, it is important that the axis not being tuned is roughly held in place. Failing to do so can result in damage to the mechanism at stronger gains. To accomplish this, a relatively weak kP can be applied to the Difference axis prior to tuning the average axis.

In Tuner X, select the differential leader motor controller. The `DifferentialPosition` signal can be plotted to determine a reasonable setpoint.

From there, apply an open-loop request to the average axis and a position closed-loop request to the Difference axis (for `SimpleDifferentialMechanism`, use the `Differential{OutputType}` requests). Adjust the kP on the Difference axis until there is a reasonable amount of resistance to motion without oscillation.

**Important**

Do not try to optimize the feedforward and PID gains on the Difference axis at this point.

### Step 2: Feedforwards and PID on the Average Axis

With the Difference axis now roughly held in place, the focus shifts to tuning the feedforwards and PID on the Average axis. The tuning process on the average axis is roughly the same as with a single-axis mechanism (such as a single gearbox elevator) with two changes:

- Use the `DifferentialAveragePosition` and `DifferentialAverageVelocity` signals instead of the `Position` and `Velocity` signals.
- Use the differential mechanism or control requests (such as `DifferentialMotionMagicVoltage`) when tuning, keeping the Difference axis held in place.

As is the case with a single-axis mechanism, many gains on the Average axis scale with the `RotorToSensorRatio` and `SensorToMechanismRatio`. For example, a Kraken X60 differential mechanism with 3:1 gearing on the average axis would have a kV of around 0.36 V/rps ($K_{v_avg} = 3.0 * 0.12$ V/rps) on the Average axis.

### Step 3: Feedforwards and PID on the Difference Axis

Finally, the feedforwards and PID gains on the Difference axis can be fully tuned:

- Use the `DifferentialDifferencePosition` and `DifferentialDifferenceVelocity` signals instead of the `Position` and `Velocity` signals.
- Use the `Differential` closed-loop signals (such as `DifferentialClosedLoopReference`) instead of the regular closed-loop signals.

- Use the differential mechanism or control requests (such as `DifferentialMotionMagicVoltage`) when tuning, keeping the Average axis held in place.

> **Note**
>
> For some mechanisms like a two-gearbox elevator, these gains may be left on the weaker side, as the focus is on the Average axis.

At a `SensorToDifferentialRatio` of 1.0, the scale of the gains on the Difference axis will be similar to those on the Average axis. For example, a Kraken X60 differential mechanism with 1:1 gearing on all axes will have a kV of around 0.12 V/rps on both the Average axis and the Differential axis. However, note that **kG is often 0** on the Difference axis.

Otherwise, the gains on the Difference axis scale with the `SensorToDifferentialRatio` **relative to the Average axis**. For example, a 3:1 gearing on the Average axis and an *additional* 2:1 gearing on the Difference axis would result in a kV of around 0.72 V/rps ($K_{v_diff} = 2.0 * K_{v_avg}$) on the Difference axis.

## 20.3 Generating Mechanisms

Mechanisms such as swerve or an elevator can be generated using *Tuner X*, greatly simplifying the setup process and eliminating many sources of error.

Additionally, the `corvus` CLI tool can be used from a terminal to generate a mechanism from a JSON specification, including a full swerve project. `corvus` can be downloaded from the CLI Tools download page.

To view a list of available commands, run `corvus` either with no parameters or with `--help`.

```
$ ./corvus
corvus 26.0.0
Generates mechanism code for multiple languages from common JSON specs.
Copyright (C) CTR Electronics. All rights reserved.

Usage: corvus.exe [OPTIONS] <--infile <INFILE>|--json <JSON>> <MECH> <LANG>
       corvus.exe <COMMAND>

Commands:
  json  Export an example JSON for modification
  help  Print this message or the help of the given subcommand(s)

Arguments:
  <MECH>  Mechanism to generate [possible values: configs, swerve, swerve-proj, elevator, arm, led]
  <LANG>  Language for generation [possible values: java, cpp-header, cpp-source, python]

Options:
  -i, --infile <INFILE>    Input json file for generation
  -j, --json <JSON>        Input json string for generation
  -o, --outfile <OUTFILE>  Output file for generation. If no file is provided, the output will be printed to stdout. Req
uired for swerve-proj
  -h, --help               Print help (see more with '--help')
  -V, --version            Print version

For support and suggestions, contact support@ctr-electronics.com
```

As an example, to generate an example `Elevator` subsystem for a Java robot program, run:

```
./corvus json elevator "Elevator.json"
./corvus elevator java -i "Elevator.json" -o
↪"src/main/java/frc/robot/subsystems/Elevator.java"
```

# 21 Simulation

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

## 21.1 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.

### 21.1.1 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**.

### 21.1.2 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();
```

### Python

```
talon_fx_sim = self.talon_fx.sim_state
```

> **Note**
>
> Phoenix 6 utilizes the Java units library and C++ units library when applicable.

In general, simulation logic should **only** interact with the `SimState` class and the physics simulator (such as WPILib's `DCMotorSim`), and the rest of the robot code should **never** interact with those simulation classes. This clear separation between simulation and hardware logic ensures that simulation most accurately reflects the behavior of the real robot hardware.

### 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++, Python).

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;
```

#### Python

```
left_talon_fx_sim = self.left_talon_fx.sim_state
right_talon_fx_sim = self.right_talon_fx.sim_state

# left drivetrain motors are typically CCW+
left_talon_fx_sim.orientation = sim.ChassisReference.
↪COUNTER_CLOCKWISE_POSITIVE

# right drivetrain motors are typically CW+
right_talon_fx_sim.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(Volts.of(12));
```

#### C++

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

#### Python

```
# set the supply voltage of the TalonFX to 12 V
self.talon_fx_sim.set_supply_voltage(12.0)
```

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

**Important**

For all motor controllers, the `RawRotorPosition` and `RotorVelocity` values must be set for simulated PID and current limits to behave correctly. Additionally, simulated PID more closely matches hardware when updating the `SimState` more frequently, such as using a WPILib `Notifier`.

### Java

```
private static final double kGearRatio = 10.0;
private final DCMotorSim m_motorSimModel = new DCMotorSim(
   LinearSystemId.createDCMotorSystem(
      DCMotor.getKrakenX60Foc(1), 0.001, kGearRatio
   ),
   DCMotor.getKrakenX60Foc(1)
);

public void simulationInit() {
   var talonFXSim = m_talonFX.getSimState();
   talonFXSim.Orientation = ChassisReference.CounterClockwise_Positive;
   talonFXSim.setMotorType(TalonFXSimState.MotorType.KrakenX60);
}

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

   // set the supply voltage of the TalonFX
   talonFXSim.setSupplyVoltage(RobotController.getBatteryVoltage());

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

   // use the motor voltage to calculate new position and velocity
   // using WPILib's DCMotorSim class for physics simulation
   m_motorSimModel.setInputVoltage(motorVoltage.in(Volts));
   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 ratio), but
   // DCMotorSim returns mechanism position/velocity (after gear ratio)

 ↪talonFXSim.setRawRotorPosition(m_motorSimModel.getAngularPosition().times(kGearRatio));

 ↪talonFXSim.setRotorVelocity(m_motorSimModel.getAngularVelocity().times(kGearRatio));
}
```

### C++

```
static constexpr double kGearRatio = 10.0;
frc::sim::DCMotorSim m_motorSimModel{
   frc::LinearSystemId::DCMotorSystem{
      frc::DCMotor::KrakenX60FOC(1),
      0.001_kg_sq_m,
      kGearRatio
   },
   frc::DCMotor::KrakenX60FOC(1)
};

void SimulationInit()
```

(continues on next page)

(continued from previous page)

```
{
    auto& talonFXSim = m_talonFX.GetSimState();
    talonFXSim.Orientation = sim::ChassisReference::CounterClockwise_Positive;
    talonFXSim.SetMotorType(sim::TalonFXSimState::MotorType::KrakenX60);
}

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

    // set the supply voltage of the TalonFX
    talonFXSim.SetSupplyVoltage(frc::RobotController::GetBatteryVoltage());

    // 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 ratio), but
    // DCMotorSim returns mechanism position/velocity (after gear ratio)
    talonFXSim.SetRawRotorPosition(kGearRatio *␣
↪m_motorSimModel.GetAngularPosition());
    talonFXSim.SetRotorVelocity(kGearRatio *␣
↪m_motorSimModel.GetAngularVelocity());
}
```

### Python

```
GEAR_RATIO = 10.0

def __init__(self):
    gearbox = DCMotor.krakenX60FOC(1)
    self.motor_sim_model = DCMotorSim(LinearSystemId.DCMotorSystem(gearbox,␣
↪0.001, GEAR_RATIO), gearbox)

def simulationInit(self):
    talon_fx_sim = self.talon_fx.sim_state
    talon_fx_sim.orientation = sim.ChassisReference.COUNTER_CLOCKWISE_POSITIVE
    talon_fx_sim.set_motor_type(sim.TalonFXSimState.MotorType.KRAKEN_X60)

def simulationPeriodic(self):
    talon_fx_sim = self.talon_fx.sim_state

    # set the supply voltage of the TalonFX
    talon_fx_sim.set_supply_voltage(RobotController.getBatteryVoltage())

    # get the motor voltage of the TalonFX
```

(continues on next page)

(continued from previous page)

```
    motor_voltage = talon_fx_sim.motor_voltage

    # use the motor voltage to calculate new position and velocity
    # using WPILib's DCMotorSim class for physics simulation
    self.motor_sim_model.setInputVoltage(motor_voltage)
    self.motor_sim_model.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 ratio), but
    # DCMotorSim returns mechanism position/velocity (after gear ratio)
    talon_fx_sim.set_raw_rotor_position(
        GEAR_RATIO
        * units.radiansToRotations(self.motor_sim_model.getAngularPosition())
    )
    talon_fx_sim.set_rotor_velocity(
        GEAR_RATIO
        * units.radiansToRotations(self.motor_sim_model.getAngularVelocity())
    )
```

### 21.1.3 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++, Python) condition.

#### Java

```
if (Utils.isSimulation()) {
    // set update rate to 1ms for unit tests
    m_velocitySignal.setUpdateFrequency(Hertz.of(1000));
}
```

#### C++

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

#### Python

```
if utils.is_simulation():
    # set update rate to 1ms for unit tests
    self.velocity_signal.set_update_frequency(1000.0)
```

# 22 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.

## 22.1 MotorController Integration

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

### Java

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

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

// instantiate joystick
final 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{
   [this](double output) { m_motorLeft.Set(output); },
   [this](double output) { m_motorRight.Set(output); }
};

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

#### Python

```
def __init__(self):
   # instantiate motor controllers
   self.motor_left = hardware.TalonFX(0)
   self.motor_right = hardware.TalonFX(1)

   # create DifferentialDrive object for robot control
   self.diff_drive = wpilib.drive.DifferentialDrive(
      self.motor_left.set, self.motor_right.set
   )

   # instantiate joystick
   self.driver_joy = wpilib.XboxController(0)

def teleopPeriodic(self):
   forward = -self.driver_joy.getLeftY()
   rot = -self.driver_joy.getRightX()

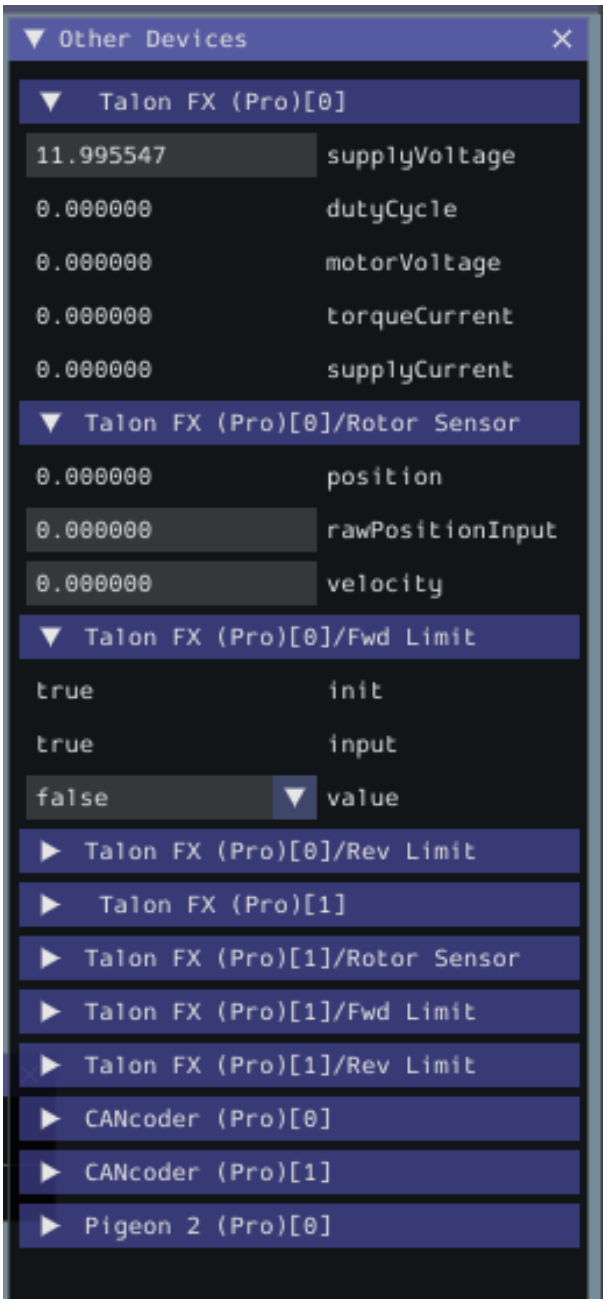
   self.diff_drive.arcadeDrive(forward, rot)
```

### 22.1.1 Motor Safety

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

### 22.1.2 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.




## 22.2 Annotation Logging with Epilogue

In a WPILib Java robot project, the `HootEpilogueBackend` (Java) can be used to integrate the Phoenix 6 *Signal Logger* with Epilogue. This makes it easy to register custom signals for logging using Java annotations.

```java
@Logged
public class Robot extends TimedRobot {
   public Robot() {
      Epilogue.configure(config -> {
         // Log to both the Phoenix 6 SignalLogger
         // and NT4 backends
         config.backend = EpilogueBackend.multi(
            new HootEpilogueBackend(),
            new NTEpilogueBackend(NetworkTableInstance.getDefault())
         );

         if (Utils.isSimulation()) {
            // Re-throw any errors that occur in simulation
            config.errorHandler = ErrorHandler.crashOnError();
         }

         // ...
      });
      Epilogue.bind(this);
   }
}
```

## 22.3 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.

### 22.3.1 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
      [](frc::sysid::State state)
      {
         SignalLogger::WriteString("state",␣
↪frc::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 or the *owlet CLI tool* 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.

**Important**

We recommend users do **not** use third-party tools to export a `hoot` log to WPILOG. Doing so may result in a lossy conversion that impacts the quality of the SysId analysis. This is particularly true in simulation, where a lossy export can result in SysId failing to analyze the data.

## 22.4 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

   static final double kGearRatio = 10.0;

   TalonFX m_fx;
   TalonFXSimState m_fxSim;
   DCMotorSim m_motorSim;

   @Override
   public void close() {
      /* destroy our TalonFX object */
      m_fx.close();
   }

   @BeforeEach
   public void constructDevices() {
```

(continues on next page)

(continued from previous page)

```
        assert HAL.initialize(500, 0);

        /* create the TalonFX */
        m_fx = new TalonFX(0);
        m_fxSim = m_fx.getSimState();
        /* create the simulated DC motor */
        var gearbox = DCMotor.getKrakenX60Foc(1);
        m_motorSim = new DCMotorSim(
            LinearSystemId.createDCMotorSystem(gearbox, 0.001, kGearRatio),
            gearbox
        );

        /* 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.getValueAsDouble(), 0.0, DELTA);

        /* request 100% output */
        m_fx.setControl(new DutyCycleOut(1.0));

        /* wait for the control to apply and the motor to accelerate */
        for (int i = 0; i < 10; ++i) {
            Timer.delay(0.020);
            m_motorSim.setInputVoltage(m_fxSim.getMotorVoltage());
            m_motorSim.update(0.020);
```

(continues on next page)

(continued from previous page)

```
↪m_fxSim.setRawRotorPosition(m_motorSim.getAngularPosition().times(kGearRatio));
↪m_fxSim.setRotorVelocity(m_motorSim.getAngularVelocity().times(kGearRatio));
      }

      /* wait for a new duty cycle signal */
      dutyCycle.waitForUpdate(0.100);
      /* verify that the motor output is 1.0 */
      assertEquals(dutyCycle.getValueAsDouble(), 1.0, DELTA);
   }
}
```

### C++

```
class TalonFXTest : public testing::Test {
protected:
   static constexpr double kGearRatio = 10.0;

   /* create the TalonFX */
   hardware::TalonFX m_fx{0};
   sim::TalonFXSimState& m_fxSim{m_fx.GetSimState()};
   /* create the simulated DC motor */
   frc::sim::DCMotorSim m_motorSim{
      frc::LinearSystemId::DCMotorSystem{
         frc::DCMotor::KrakenX60FOC(1),
         0.001_kg_sq_m,
         kGearRatio
      },
      frc::DCMotor::KrakenX60FOC(1)
   };

   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)
{
```

(continues on next page)

(continued from previous page)

```
   /* 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 and the motor to accelerate */
   for (int i = 0; i < 10; ++i) {
      std::this_thread::sleep_for(std::chrono::milliseconds{20});
      m_motorSim.SetInputVoltage(m_fxSim.GetMotorVoltage());
      m_motorSim.Update(20_ms);

      m_fxSim.SetRawRotorPosition(kGearRatio *␣
↪m_motorSim.GetAngularPosition());
      m_fxSim.SetRotorVelocity(kGearRatio * m_motorSim.GetAngularVelocity());
   }

   /* 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);
}
```

# 23 Examples

Comprehensive API usage examples and tutorials.

## 23.1 Open-Loop Quickstart

The below example showcases controlling a four-motor drivetrain.

### 23.1.1 Declaring Motor Controllers

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

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

#### Java

```java
public class Robot extends TimedRobot {
   private static final CANBus kCANBus = new CANBus("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)

```cpp
class Robot : public frc::TimedRobot {
private:
   static constexpr ctre::phoenix6::CANBus kCANBus{"canivore"};
```

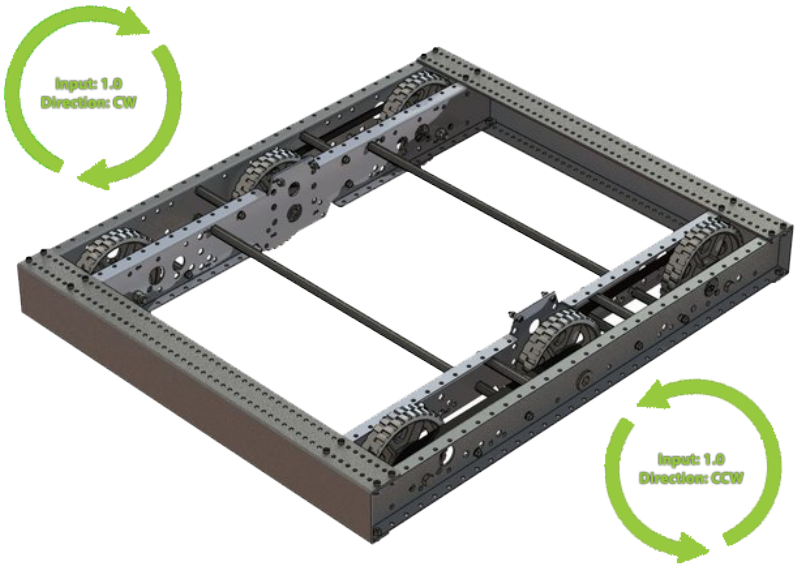
(continues on next page)

(continued from previous page)

```
    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};
}
```

### 23.1.2 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

```
public Robot() {
    // start with factory-default configs
    var configs = new TalonFXConfiguration();

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

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

    // Ensure our followers are following their respective leader
    m_leftFollower.setControl(new Follower(m_leftLeader.getDeviceID(),␣
↪MotorAlignmentValue.Aligned));
    m_rightFollower.setControl(new Follower(m_rightLeader.getDeviceID(),␣
```

(continues on next page)

(continued from previous page)

```
↪MotorAlignmentValue.Aligned));
}
```

#### C++ (Source)

```
#include "Robot.h"

using namespace ctre::phoenix6;

Robot::Robot() {
   // start with factory-default configs
   configs::TalonFXConfiguration configs{};

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

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

   // 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});
}
```

### 23.1.3 Full Example

#### Java

```
public class Robot extends TimedRobot {
   private static final CANBus kCANBus = new CANBus("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);

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

   private final XboxController m_driverJoy = new XboxController(0);

   public Robot() {
      // start with factory-default configs
      var configs = new TalonFXConfiguration();
```

(continues on next page)

(continued from previous page)

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

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

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

   @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;

Robot::Robot() {
   // start with factory-default configs
   configs::TalonFXConfiguration configs{};

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

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

   // Ensure the followers are following their respective leader
```

(continues on next page)

(continued from previous page)

```
   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 ctre::phoenix6::CANBus 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};
```

# 24 CANivore® Intro

The CTR Electronics 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 *Supported Devices*.

**Important**

Details on licensing your CANivore® is available on the *licensing* page.

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

*CANivore® Setup* API Usage Using the CANivore® with devices in API.

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

*Hardware-Attached Simulation* Advanced Configuration Advanced configuration options for the CANivore®.

*Advanced Configuration*

## 24.1 Status Light Reference

# 25 CANivore® Setup

## 25.1 Installing for roboRIO

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

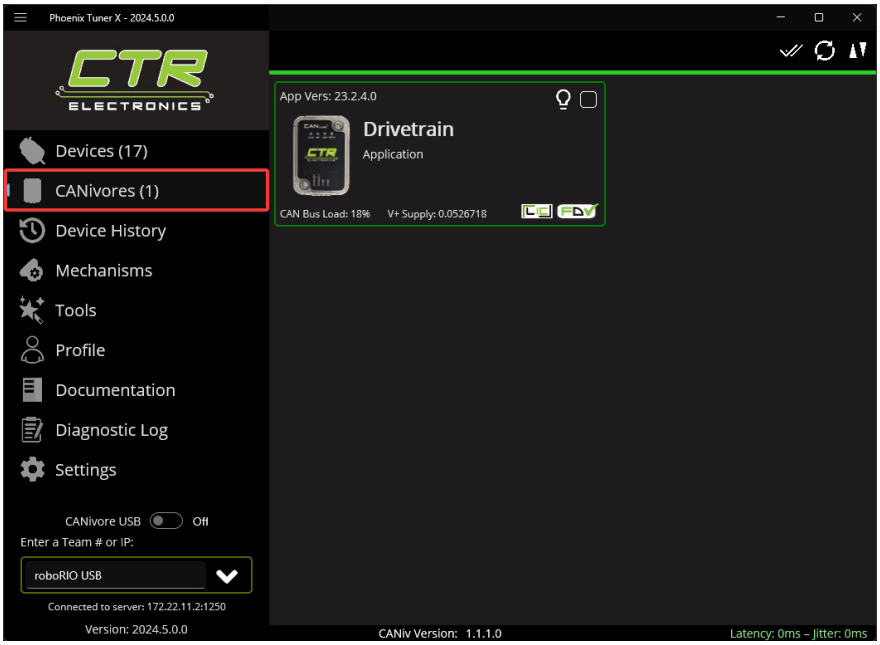
No additional steps are required. The roboRIO comes with the `canivore-usb` kernel module pre-installed.

## 25.2 Installing for Linux (non-FRC)

See *CANivore® Installation* for information on setting up your Linux system for use with a CANivore®.

## 25.3 Viewing Attached CANivore® Devices

Attached CANivore® devices 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.

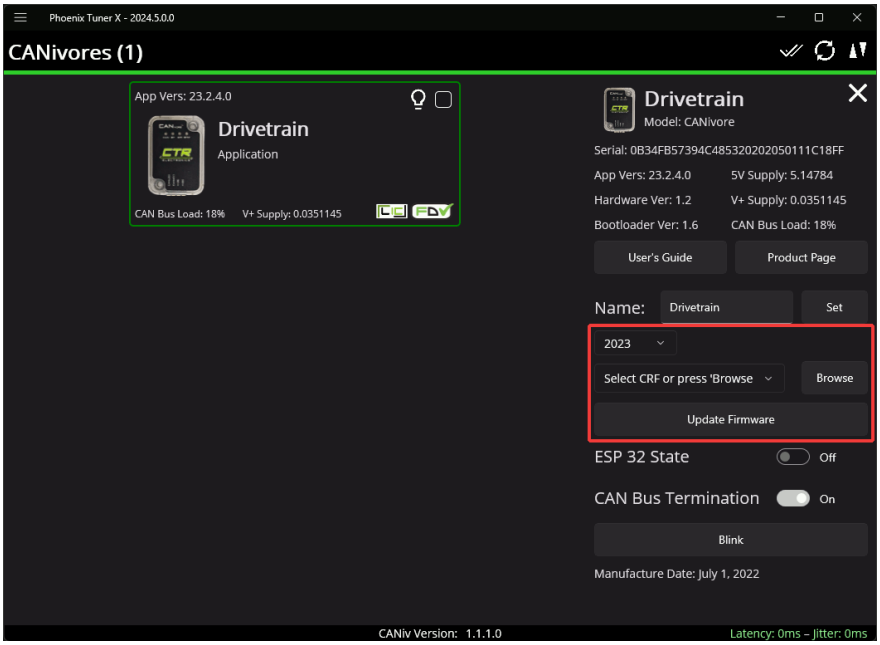
**Tip**

If you are connecting to CANivore® devices 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 CANivore® devices.

## 25.4 Field Upgrading CANivore®

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

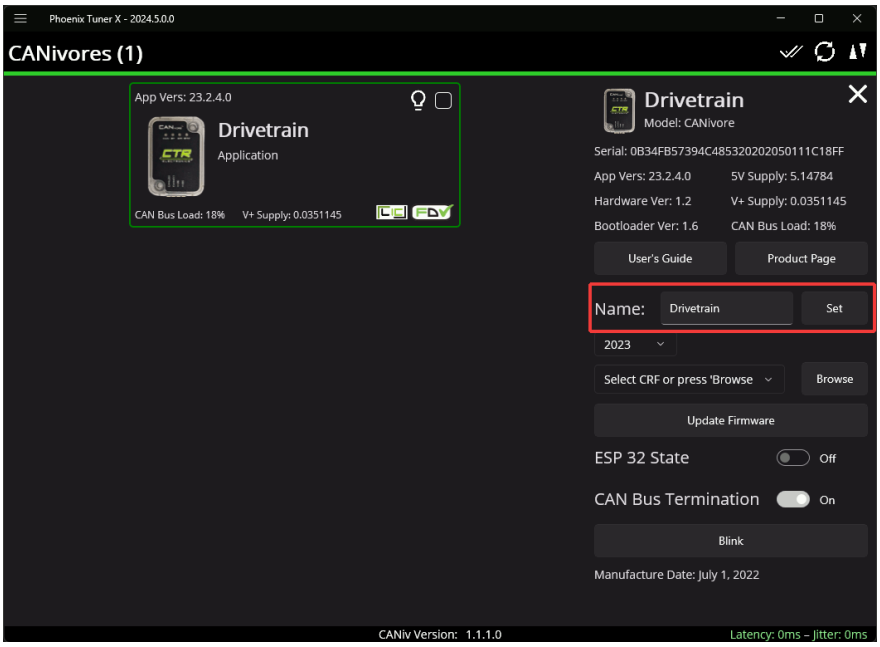
Click or tap on the listed CANivore® card to open the device details page. The CANivore® can then be field upgraded via the dropdown or by manually selected a file:




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

## 25.5 Renaming CANivore®

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



# 26

# CANivore® API

All device constructors have an overload that takes a `CANBus` object (Java, C++, Python). The native roboRIO CAN bus can be constructed using `CANBus.roboRIO()`. Otherwise, the `CANBus` constructor takes a string identifier. This identifier can be * to select the first available CANivore®, or it can be the CANivore® name or serial number. On non-FRC Linux systems, this string can also be a SocketCAN interface.

> **Note**
>
> If there are multiple CANivore® devices 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

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

(continues on next page)

(continued from previous page)

```
hardware::TalonFX fx_rio{1, CANBus::RoboRIO()}; // This also constructs a␣
↪TalonFX on the RIO native CAN bus
hardware::TalonFX fx_drivebase{0, CANBus{"Drivebase"}}; // This constructs a␣
↪TalonFX on the CANivore bus named "Drivebase"
hardware::CANcoder cc_elevator{0, CANBus{"Elevator"}}; // This constructs a␣
↪CANcoder on the CANivore bus named "Elevator"
```

### Python

```
self._fx_default = hardware.TalonFX(0) # On roboRIO, this constructs a␣
↪TalonFX on the RIO native CAN bus
self._fx_rio = hardware.TalonFX(1, CANBus.roborio()) # This also constructs a␣
↪TalonFX on the RIO native CAN bus
self._fx_drivebase = hardware.TalonFX(0, CANBus("Drivebase")) # This␣
↪constructs a TalonFX on the CANivore bus named "Drivebase"
self._cc_elevator = hardware.CANcoder(0, CANBus("Elevator")) # This␣
↪constructs a CANcoder on the CANivore bus named "Elevator"
```

The CANBus API can also be used to retrieve information about any given CAN bus, such as the bus utilization.

### Java

```
// create a CAN bus for the CANivore named drivetrain
final CANBus canbus = new CANBus("drivetrain");
// construct a TalonFX on the CAN bus
final TalonFX fx = new TalonFX(0, canbus);

// retrieve bus utilization for the CAN bus
CANBusStatus canInfo = canbus.getStatus();
float busUtil = canInfo.BusUtilization;

if (busUtil > 0.8) {
   System.out.println("CAN bus utilization is greater than 80%!");
}
```

### C++

```
// create a CAN bus for the CANivore named drivetrain
CANBus canbus{"drivetrain"};
// construct a TalonFX on the CAN bus
hardware::TalonFX fx{0, canbus};

// retrieve bus utilization for the CANivore named drivetrain
CANBus::CANBusStatus canInfo = canbus.GetStatus();
float busUtil = canInfo.BusUtilization;

if (busUtil > 0.8) {
   std::cout << "CAN bus utilization is greater than 80%!" << std::endl;
}
```

**Python**

```
# create a CAN bus for the CANivore named drivetrain
self._canbus = CANBus("drivetrain")
# construct a TalonFX on the CAN bus
self._fx = hardware.TalonFX(0, self._canbus)

# retrieve bus utilization for the CANivore named drivetrain
can_info = self._canbus.get_status()
bus_util = can_info.bus_utilization

if bus_util > 0.8:
   print("CAN bus utilization is greater than 80%!")
```

## 26.1 CANivore® Status Prints

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

Table 1: Connection Messages

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

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

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

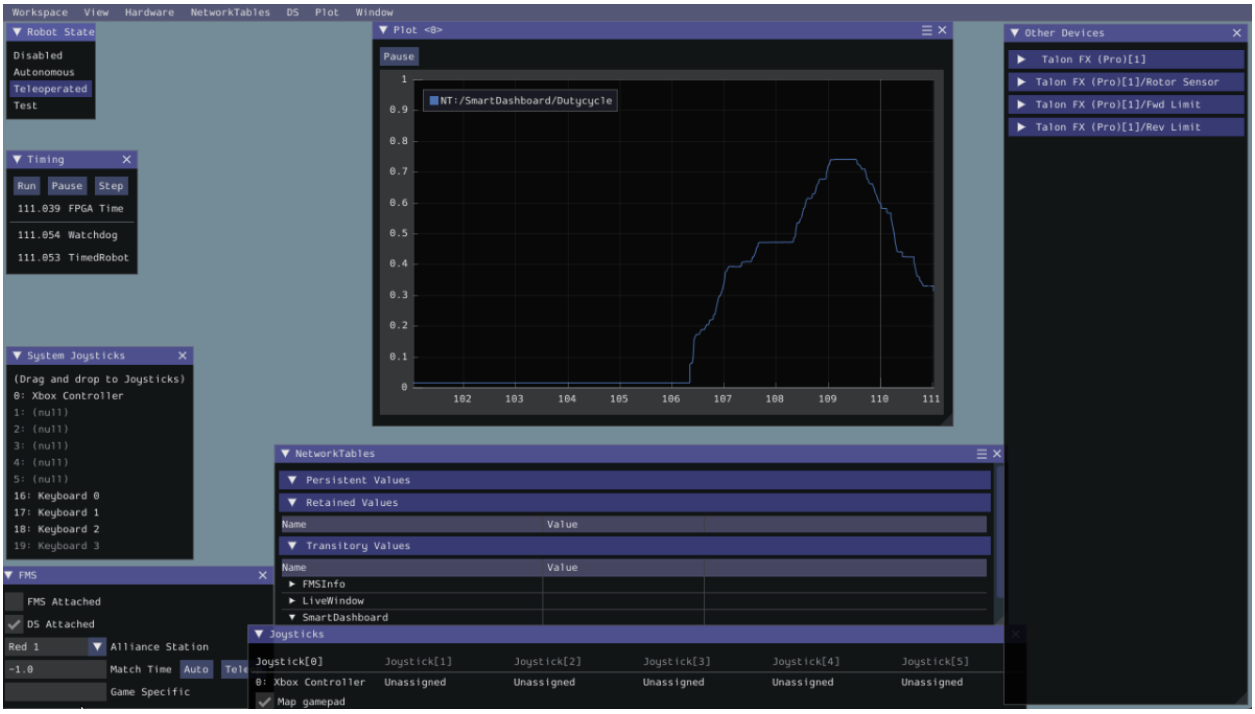
Table 3: Network State Messages

| Message | Network State |
| --- | --- |
| CANbus Network Down | Linux: The SocketCAN network has been deactivated, USB-to-CAN activity has stopped<br>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<br>Windows: Successfully opened the communication channels for USB-to-CAN traffic |

# 27

# 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

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

### C++

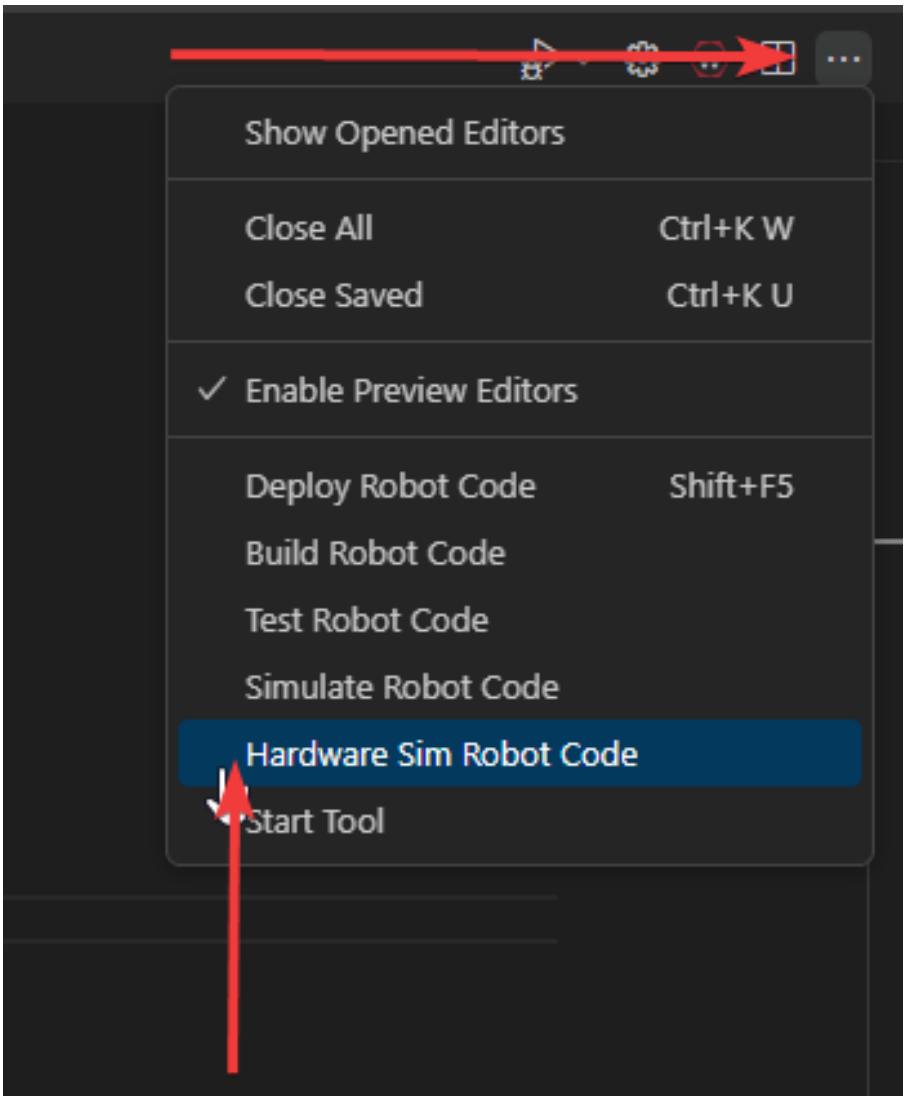
```
static constexpr ctre::phoenix6::CANBus kCANBus{"mycanivore"};
ctre::phoenix6::hardware::TalonFX m_motor{0, kCANBus};
```

### Python

```
self.canbus = CANBus("mycanivore")
self.motor = hardware.TalonFX(0, self.canbus)
```

### Java/C++

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, 2B189E6333533853202020343838 03FF)
[phoenix] CANbus Network Up: uno (WinUSB, 2B189E6333533853202020343838 03FF)
[phoenix] Library initialization is complete.
```

### Python

Simulation can be started using

```
python -m robotpy sim
```

Users may notice the robot program is using simulated devices by default. This is the default behavior if the host platform supports simulation (see *requirements* for a full list of supported platforms).

In order for the robot program to communicate with physical devices (on platforms that support both simulation and hardware), the `CTR_TARGET` environment variable must be set. Examples of this are shown below.

### Windows

```
$env:CTR_TARGET="Hardware" # Set the environment variable, which will persist␣
↪for the duration of this powershell instance.
```

### Linux

```
export CTR_TARGET=Hardware # Export the environment variable so it's␣
↪persistent in the shell
```

Or

```
CTR_TARGET=Hardware python3 application.py # Set the environment variable␣
↪only for the python call
```

# 28 Advanced Configuration

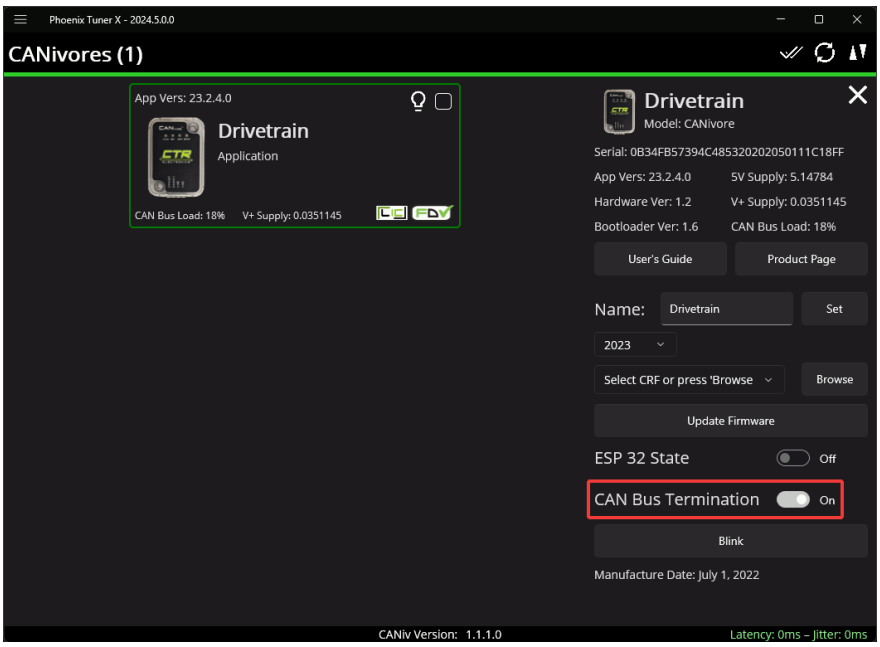
The CANivore® provides additional configuration options for advanced users.

## 28.1 CAN Bus Termination

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

**Warning**

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

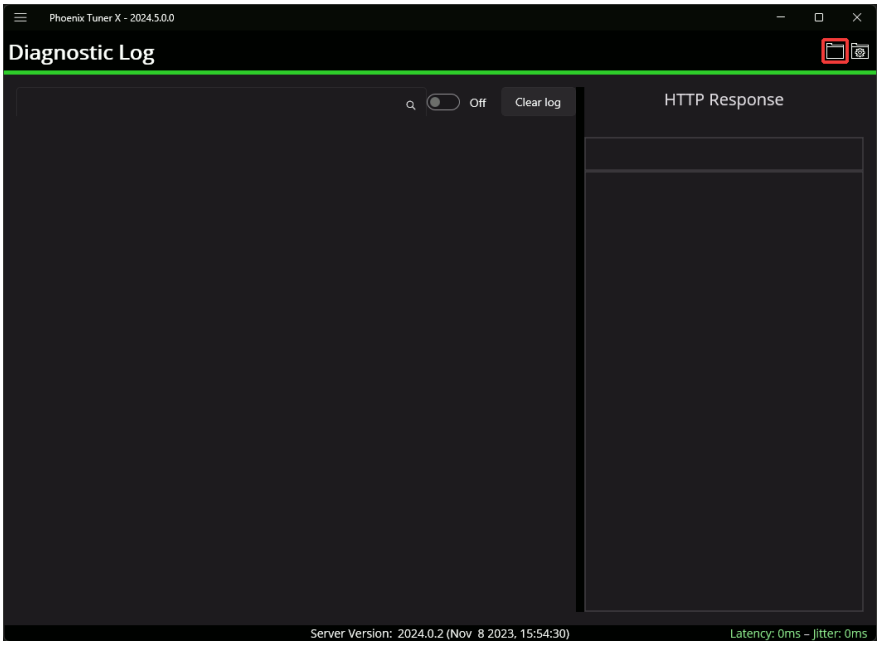


## 28.2 caniv - CANivore® CLI

`caniv` is a Command-line Interface (CLI) to interact with CANivore® devices 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:~#
```

# 29 Development Blog

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

> **Note**
>
> This list may move in the future.


- *Latency and Frequency*
- *Tuner and an evolution in configuration*
- *Factors that Impact Odometry*


## 29.1 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.

### 29.1.1 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.

### 29.1.2 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).

### 29.1.3 Solutions

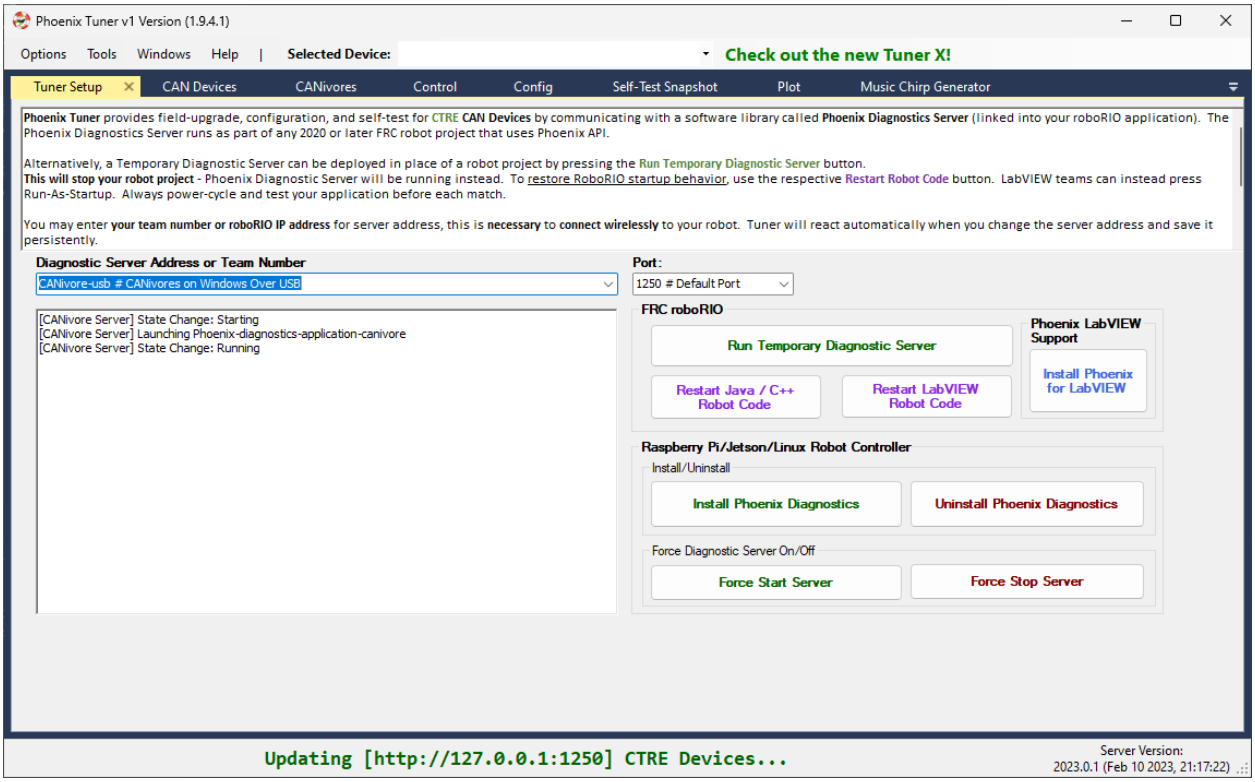
We (CTR Electronics) recognized these problems in general and wanted to solve them, which is why each of these problems are solved in the new Phoenix 6/Pro library and with CANivore®.

- *FusedCANcoder* (**Pro**): By fusing in the CANcoder® position into the Talon FX® or Talon FXS® 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® position can be latency-compensated and fused even when the mechanism is moving.
- *Timesync®* (**Pro & CANivore®**): The CANivore® provides the ability for Pro devices to synchronize to a common clock using Timesync®, 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.
- *Timestamps* (**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.

## 29.2 Tuner and an evolution in configuration

*Authored by Dalton*



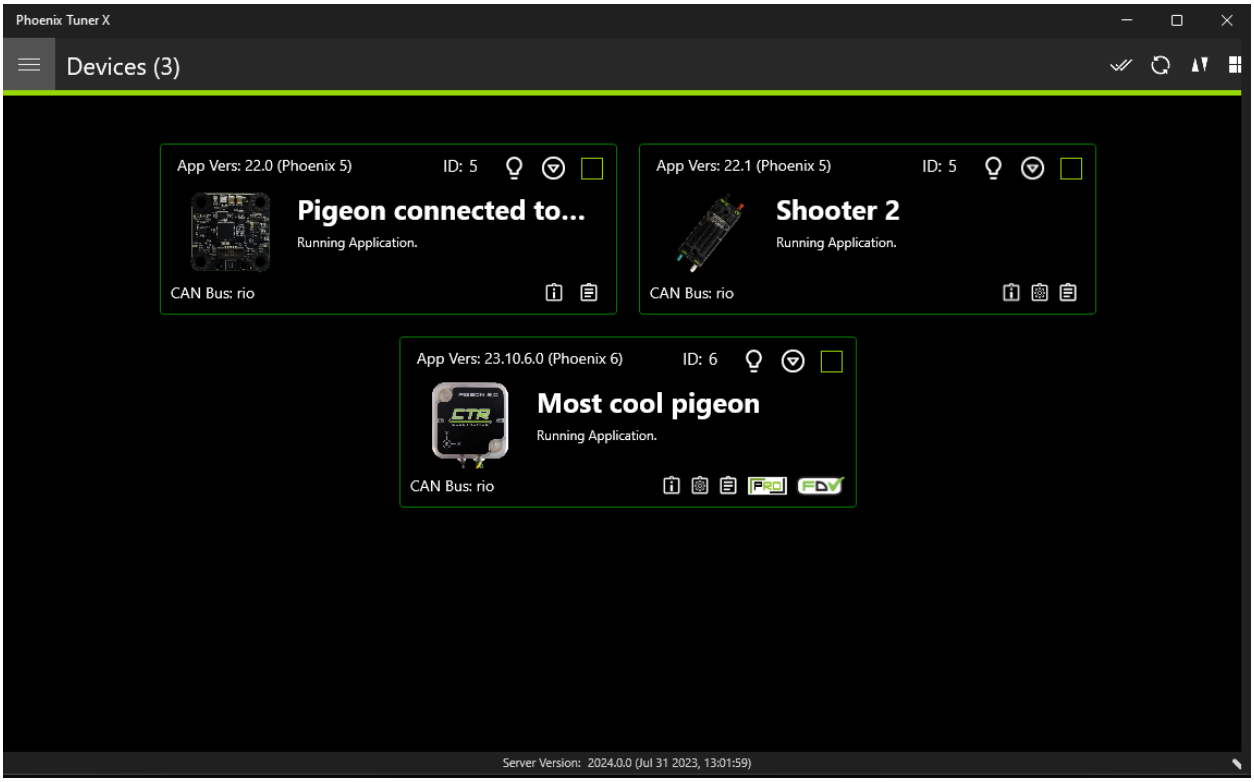

Since the introduction of the CTRE Toolsuite (pre-2018), we at CTR have strived to provide intuitive means of configuring and utilizing our products. In 2018, we launched Phoenix Tuner (now lovingly referred to as Tuner v1). Tuner v1 introduced features like: batch firmware upgrading like devices, diagnostic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

### 29.2.1 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*.

### 29.2.2 Introducing a new iteration of Tuner X

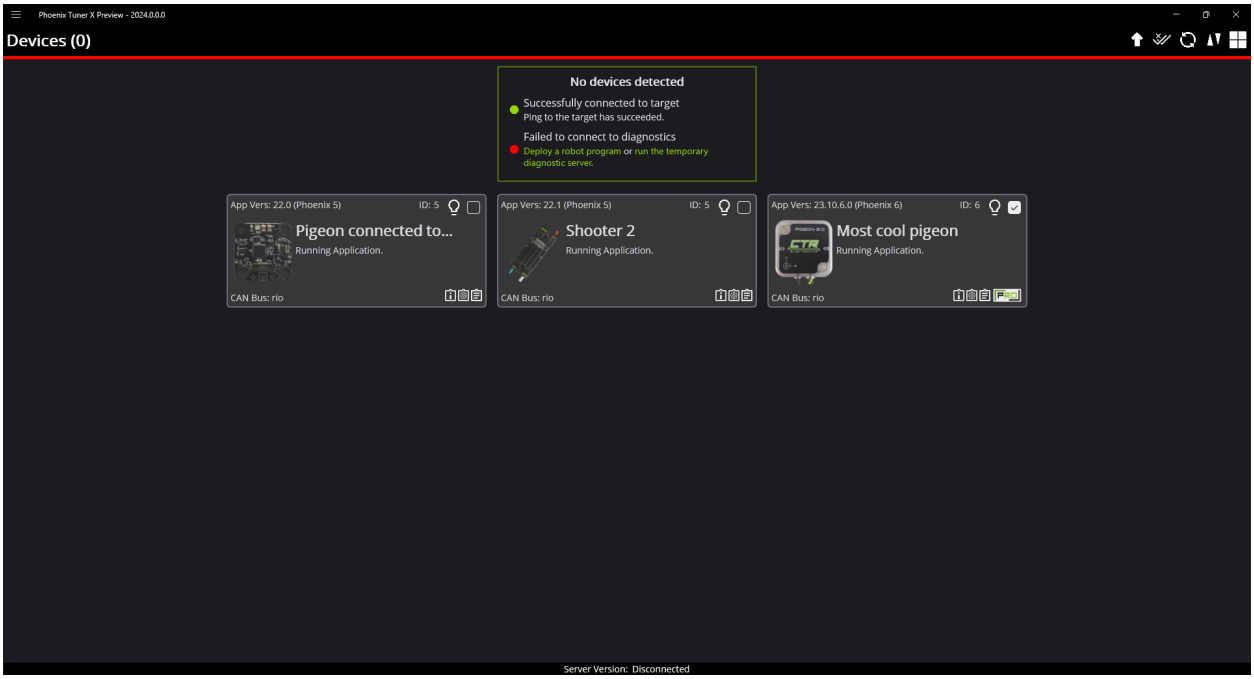
Some of you may have noticed that your version of Tuner X has changed recently. We've been working on several key improvements to the application that should dramatically improve the user experience. While this blog will highlight some of those, it's best to just try out the new Tuner yourself.

**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.



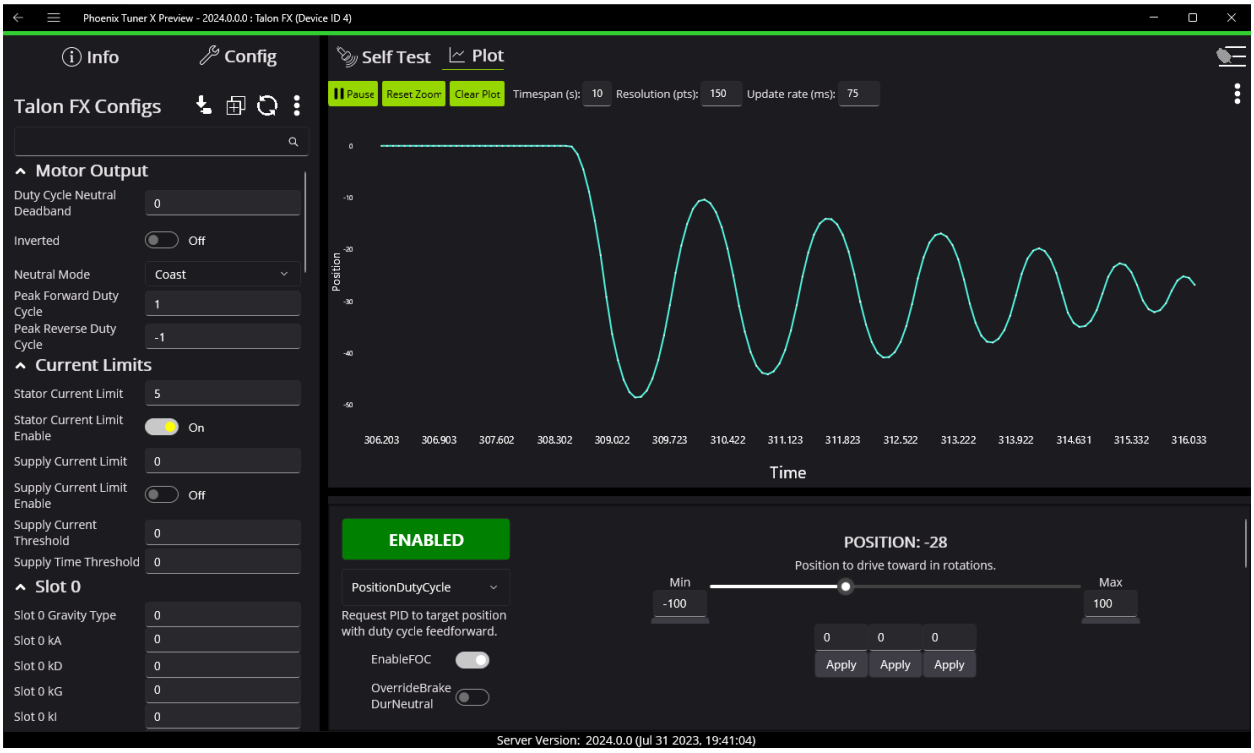

This 3 step check looks for the following:

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

### 29.2.3 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.

## 29.3 Factors that Impact Odometry

*Authored by Cory*

Often we've been asked what the impact higher frequencies, Timesync®, 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.

### 29.3.1 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.

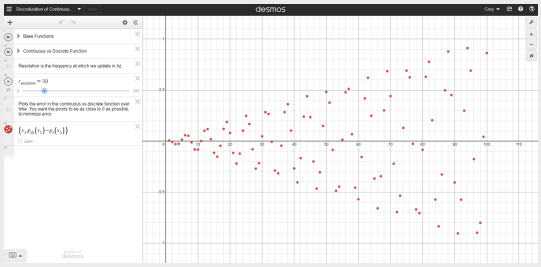


Fig. 1: Discretization error at 50 Hz

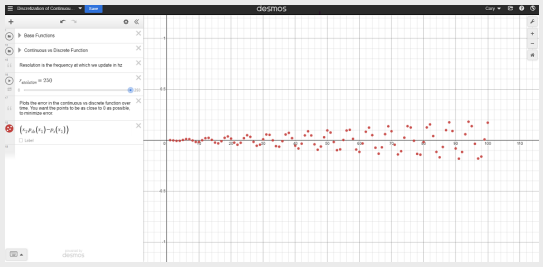


Fig. 2: Discretization error at 250 Hz

### 29.3.2 Synchronous API and Timesync®

Synchronous API and Timesync® 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 *Timesync®* documentation.

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

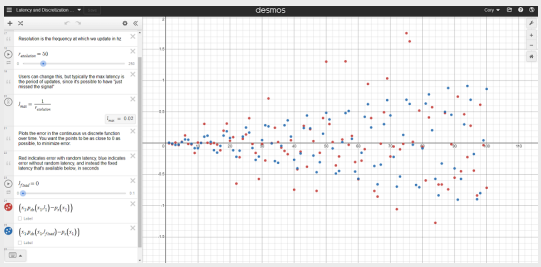


Fig. 3: Discretization error at 50 Hz with latency

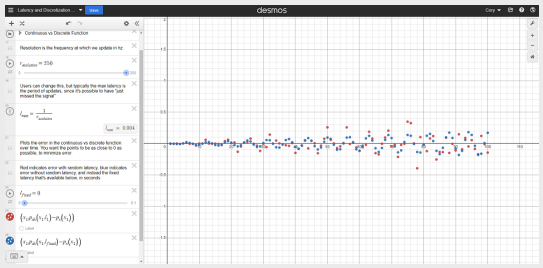


Fig. 4: Discretization error at 250 Hz with latency

### 29.3.3 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.

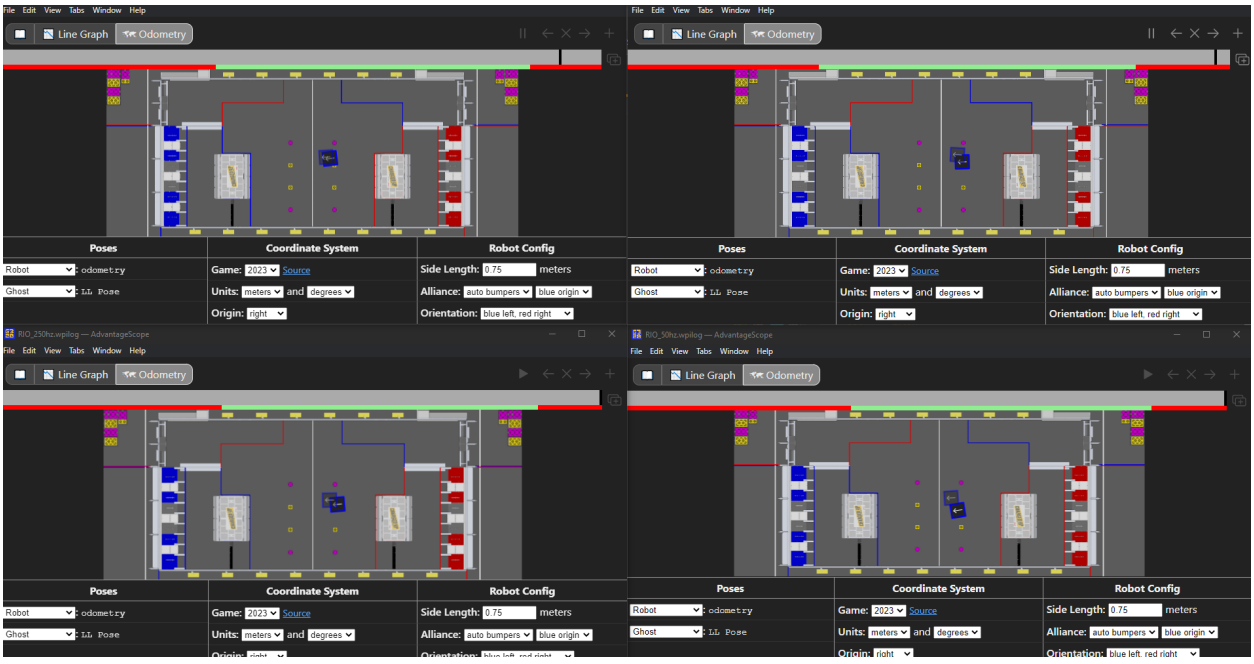
The Limelight is configured for a high resolution capture to reduce the error of its pose estimation, at the cost of less frequent pose calculations. This is acceptable, because as the robot performs its maneuvers, it will come to a rest at key points, and when it's at rest, we can do our comparisons 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

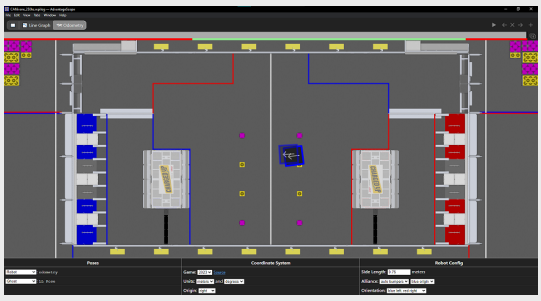


Fig. 5: CANivore® 250 Hz end position

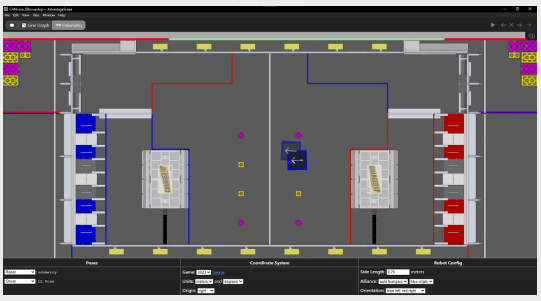


Fig. 6: CANivore® 50 Hz end position

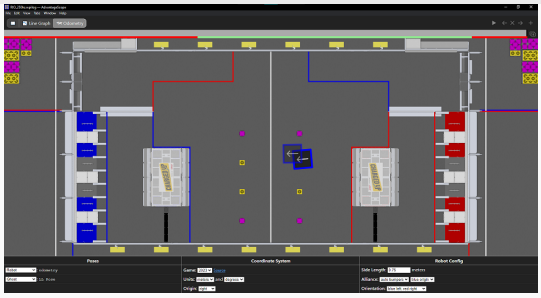


Fig. 7: RIO 250 Hz end position



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 Timesync® 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`.

## 29.3.4 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 Timesync®. 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 |

# 30 Troubleshooting

## 30.1 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).

### 30.1.1 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).

### 30.1.2 Check your wiring

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

- If connectors are used for CAN bus, **tug-test each individual crimped wire** one at a time. Bad crimps/connection points are the most common cause of intermittent connection issues.
- Confirm red and black are not flipped.
- Confirm battery voltage is adequate (through Driver Station or through voltmeter).
- Manually inspect and confirm that green-connects-to-green and yellow-connects-to-yellow at every connection point. **Flipping/mixing green and yellow is a common failure point during hardware bring up**.
- Confirm breakers are installed in the PDP where appropriate.
- Measure resistance between CANH and CANL when system is not powered (should measure ~60Ω). If the measurement is 120Ω, then confirm both RIO and PDP are in circuit, and PDP jumper is in the correct location.

### 30.1.3 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 Talon FX®) 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. Talon FX® and Pigeon® 2.0 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.

## 30.2 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.

### 30.2.1 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
```

### 30.2.2 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.

> **Note**
>
> Temporary diagnostic server can be deployed on *supported non-FRC platforms*. However, devices will only enumerate if the CAN bus interface is `can0`.

# 31 Support

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

# 32

# Code Generators

Swerve Generator  Functional swerve in under 30 minutes.

*Swerve Project Generator*  Elevator Generator  Generate an elevator subsystem into your robot project.

*Tuner Elevator Generator*

# 33

# Useful Categories

Installation Installation instructions for Phoenix API & Tuner.

docs/installation/index Phoenix Tuner Documentation that introduces the companion application to manage your CTR Electronics devices.

*Phoenix Tuner X* Hardware Reference Documentation for device specific configuration, troubleshooting and setup instructions.

docs/hardware-reference/index API Reference Documentation and details on using the CTR Electronics device API. This includes usage of signals, configs, control requests, etc.

docs/api-reference/index Examples Software API examples for controlling your devices.

*Examples* Troubleshooting Common troubleshooting for hardware or software problems.

*Troubleshooting*