Tinymovr

Yannis Chatzikonstantinou

Mar 12, 2023

CONTENTS:

CHAPTER

ONE

FROM ZERO TO SPIN

1.1 Preparation

If you are using a Tinymovr Servo Kit/Tinymovr Dev Kit, please ensure you have completed *Connecting Data* and *Connecting Power*.

If you are using a Tinymovr board in your own setup, please go through *Hardware Overview* and *Hardware Setup*.

If using Tinymovr on Windows with a CANtact-compatible CAN Bus adapter, such as CANine, you will need to install an .inf file to enable proper device naming. You can download the inf file here. Extract the archive, right click and select Install.

If using CAN bus, you may need to enable at least one termination resistor, either on the CANine adapter, or on one Tinymovr. To enable the termination resistor on CANine, you can take a look at the CANine connectors diagram.

Before proceeding to the next steps, ensure your Tinymovr is powered up.

1.2 Installing and Launching Studio

Tinymovr can be installed using pip. Python 3.6 or greater is required.

```
pip3 install tinymovr
tinymovr
```

You should now be looking at the Tinymovr Studio IPython interface.

1.3 Installing on Raspberry Pi

Installation on Raspberry Pi requires a few additional steps.

```
sudo apt update
sudo apt install python3-pip python3-numpy libopenjp2-7 libtiff5
pip3 install tinymovr
```

You may also need to append a directory to your PATH variable:

```
echo 'export PATH="/home/pi/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Now you should be able to run Tinymovr Studio:

```
tinymovr
```

1.4 Checking Functionality and Calibrating

```
tm1.device_info
```

IPython should display an array of device-related information.

If using a Tinymovr Servo Kit/Tinymovr Dev Kit, the motor and encoder should have been already calibrated. If you have your own setup, or if you experience problems with prior calibration, you'll need to go through the following brief calibration procedure.

Note: After issuing the command below, the motor will spin. Ensure the rotor is free of obstructions or loads, and the motor is firmly fixed.

```
tm1.calibrate()
```

Follow the on-screen prompts. The motor will produce an audible beep and rotate in one direction. Your Tinymovr is now ready for operation. You can view information about the attached motor as follows:

```
tm1.motor_config
```

This will reveal identified motor parameters, namely: phase resistance, phase inductance, number of pole pairs and encoder ticks.

1.5 Testing Position Control Mode

Note: After issuing the command below, the motor will hold position and may spin.

```
tm1.position_control()
```

The motor should now be actively holding it's position. Try moving it by hand and you should feel resistance.

Now try to command a new position:

```
tm1.set_pos_setpoint(0)
tm1.set_pos_setpoint(8000)
```

The motor should jump to the commanded positions.

CHAPTER

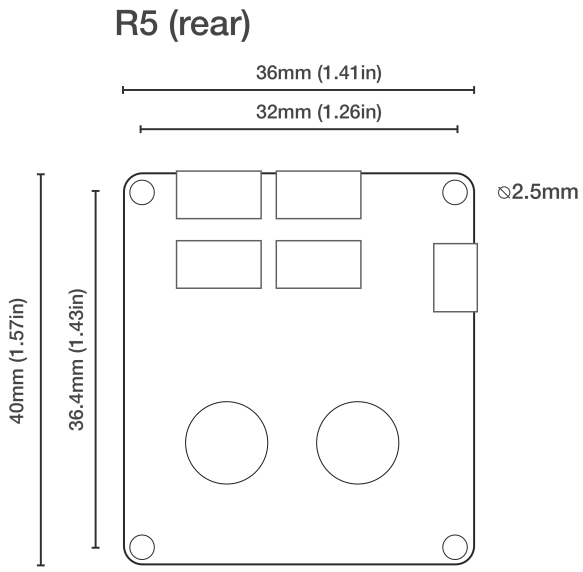
TWO

HARDWARE OVERVIEW

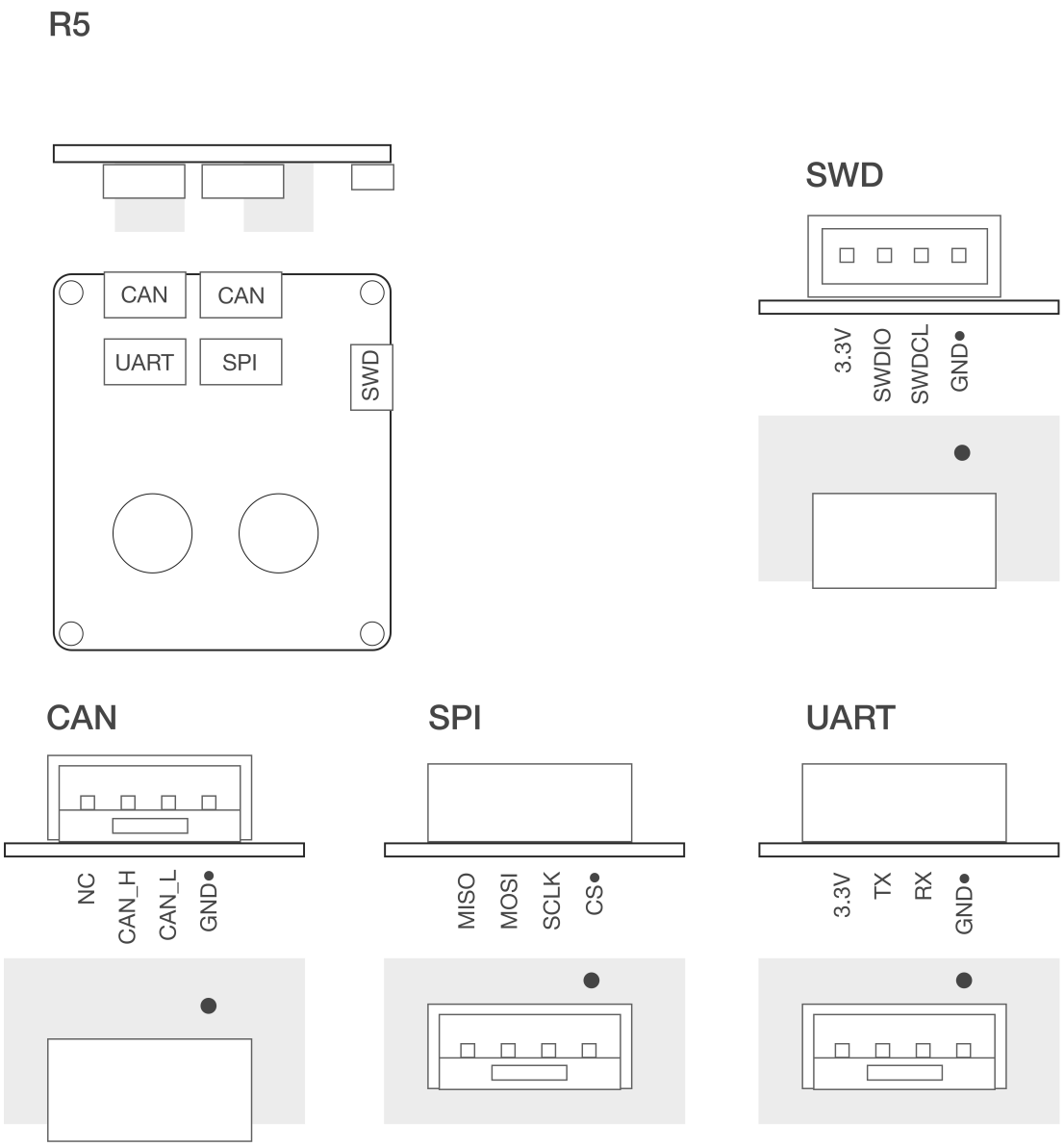
2.1 R5.x

Tinymovr R5.x is the latest Tinymovr revision. It features increased connectivity in a reduced footprint.

2.1.1 Board Dimensions



2.1.2 Connectivity

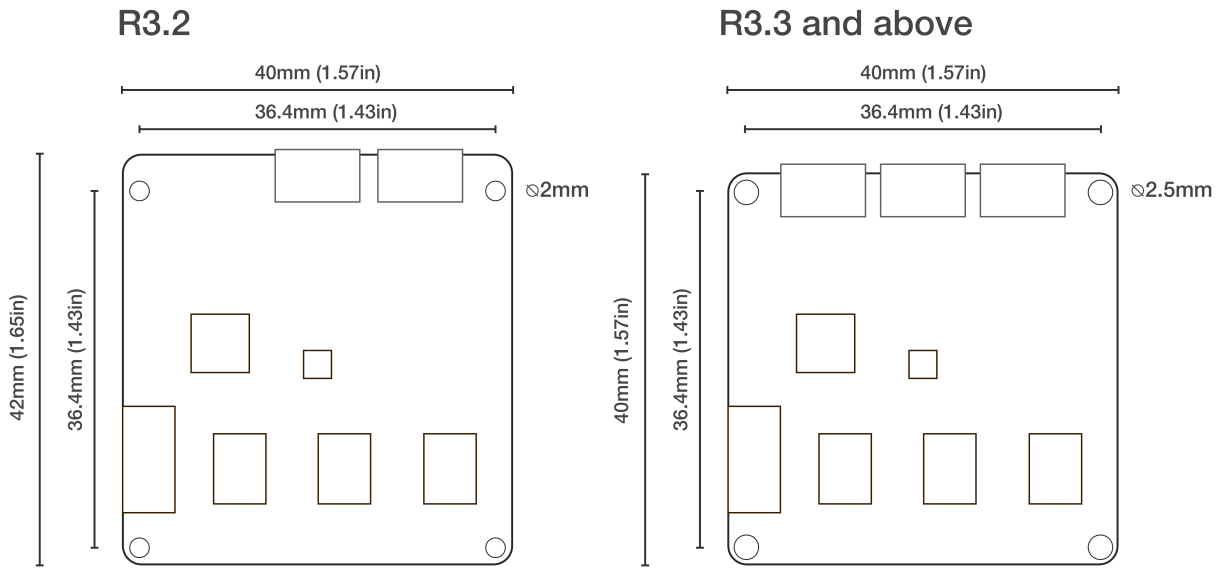


Warning: The UART pins in Tinymovr R5.1 have the silkscreen reversed. If you are planning to use UART with R5.1, consult *Tinymovr R5 UART Silkscreen Reversed*. This only affects R5.1 boards.

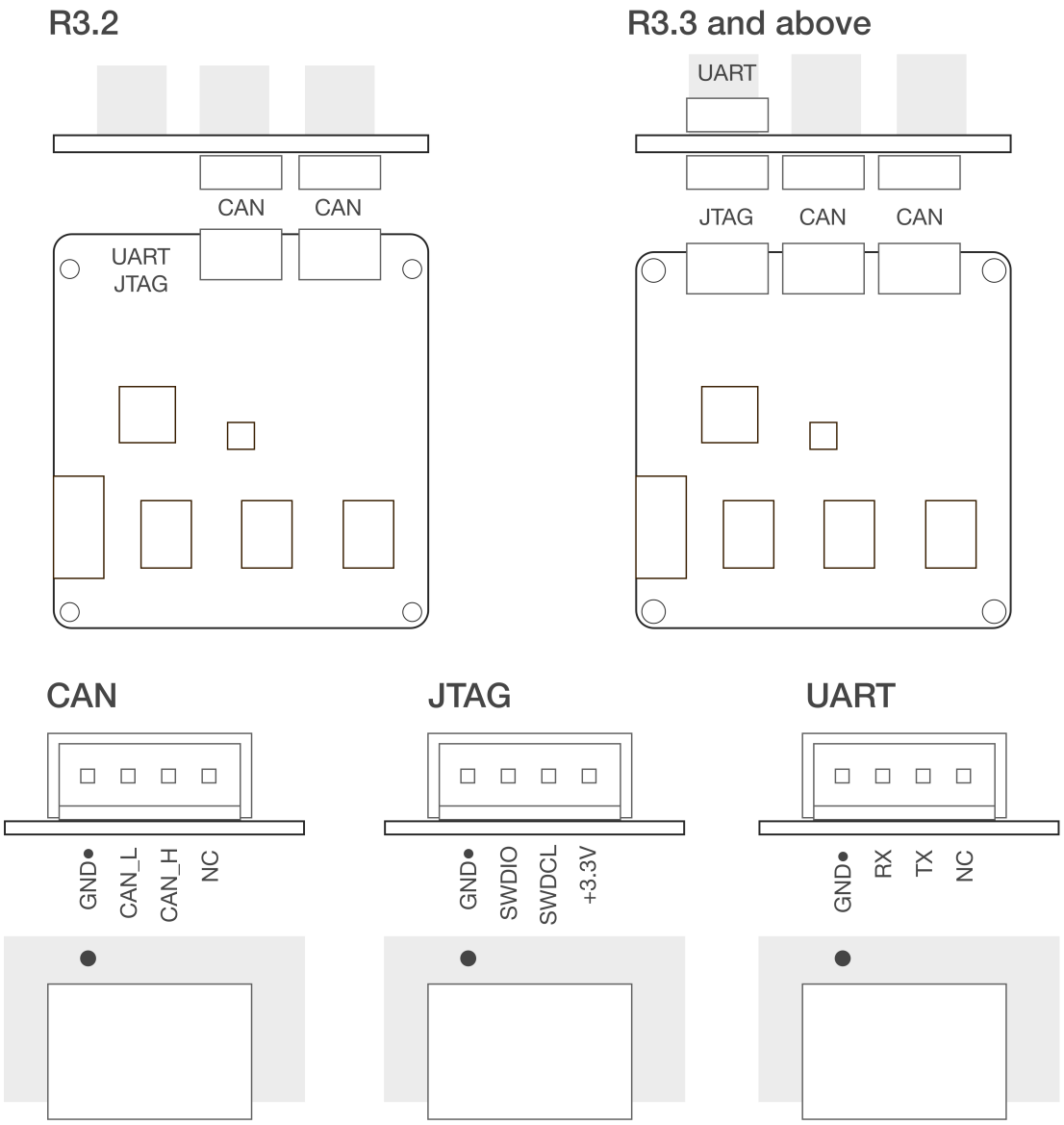
2.2 R3.x

R3.x is the previous Tinymovr revision, with CAN and UART connectivity.

2.2.1 Board Dimensions



2.2.2 Connectivity



CHAPTER

THREE

HARDWARE SETUP

3.1 Requirements

1. A 3-phase brushless motor (see below for supported types)
2. A means to talk CAN Bus, such as CANine or a Canable-compatible adapter.
3. A mechanical rig that ensures firm connection between the Tinymovr PCB and the brushless motor. Designs that can be 3D printed are available.

Note that the Tinymovr Servo Kit includes all of the above in a ready to use kit.

3.2 Supported Motor Types

Most three-phase pancake-style outrunners can be used with Tinymovr. While there is a lot of variation between motors of even the same size and external appearance, as a general rule-of-thumb motors ranging from 40mm outer diameter to 110mm should work fine with Tinymovr.

(image)

3.3 Mechanical Setup

3.3.1 Mounting motor and Tinymovr

The most important aspect of a correct setup is to ensure the controller is properly positioned in relation to the motor. The center of the PCB, where the encoder is located, should lie as close to the motor rotation axis as possible. In addition, the distance from the encoder magnet to the encoder IC should be less than 2mm (less than 1mm if you are mounting the PCB packwards, i.e. the encoder IC is facing away from the magnet).

A 3D printable encoder magnet jig is available, suitable for 6mm disc magnets and 14, 19, 25 and 30mm motor hole diameters.

For a 3D printable motor mount design, check out the Tinymovr alpha dev kit mount (suitable for 40xx motors).

Note: For safety reasons, you should always ensure the motor & controller assembly are secured to a stable surface before operation. The motor rotor may experience high acceleration that may cause damage or injury if not secured properly.

Magnet on the rear side of the PCB

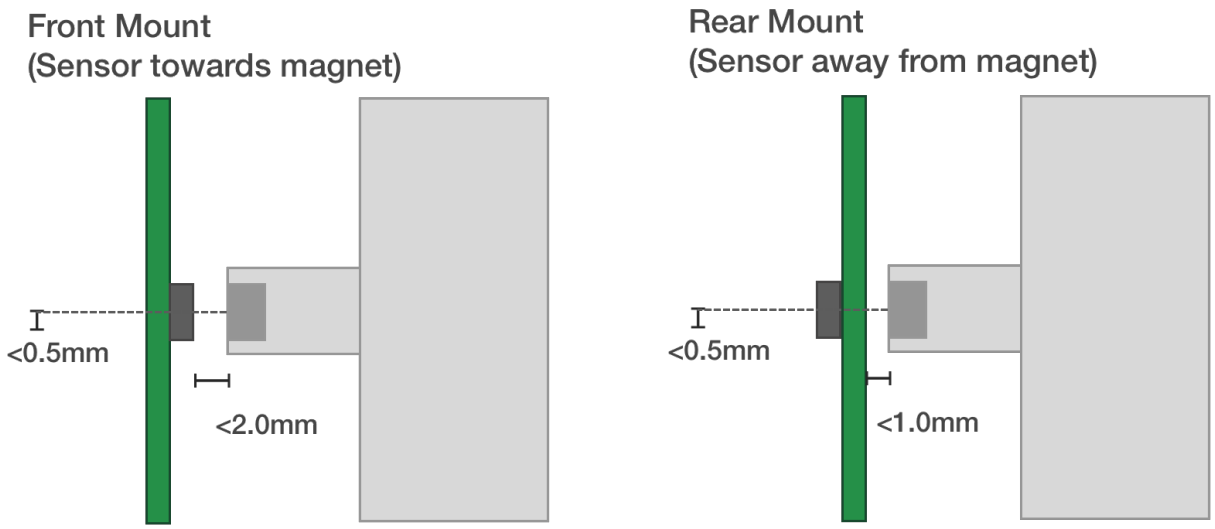


Fig. 1: Tinymovr and motor mechanical mounting

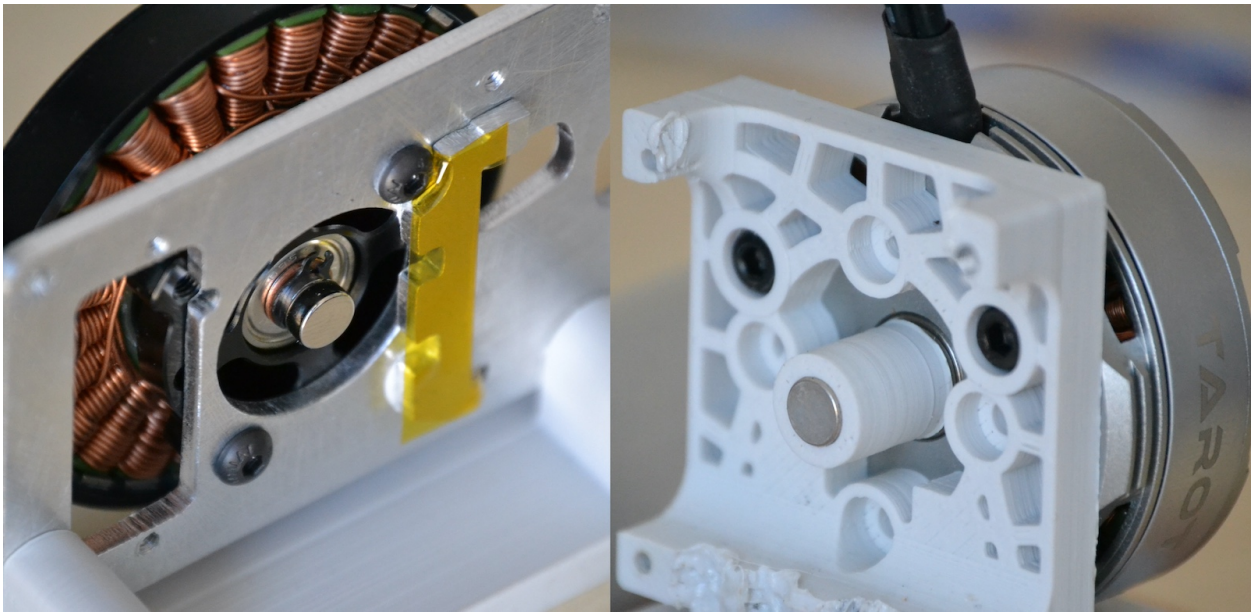


Fig. 2: Left: Magnet mount directly on shaft. Right: Magnet mount using 3d-printed holder.

TL;DR: It is possible to have the magnet on the rear dise of the PCB, i.e. opposite of the magnet sensor IC, but the gap needs to be reduced to account for the PCB thickness.

This has been verified by MPS in this forum post, quoted below:

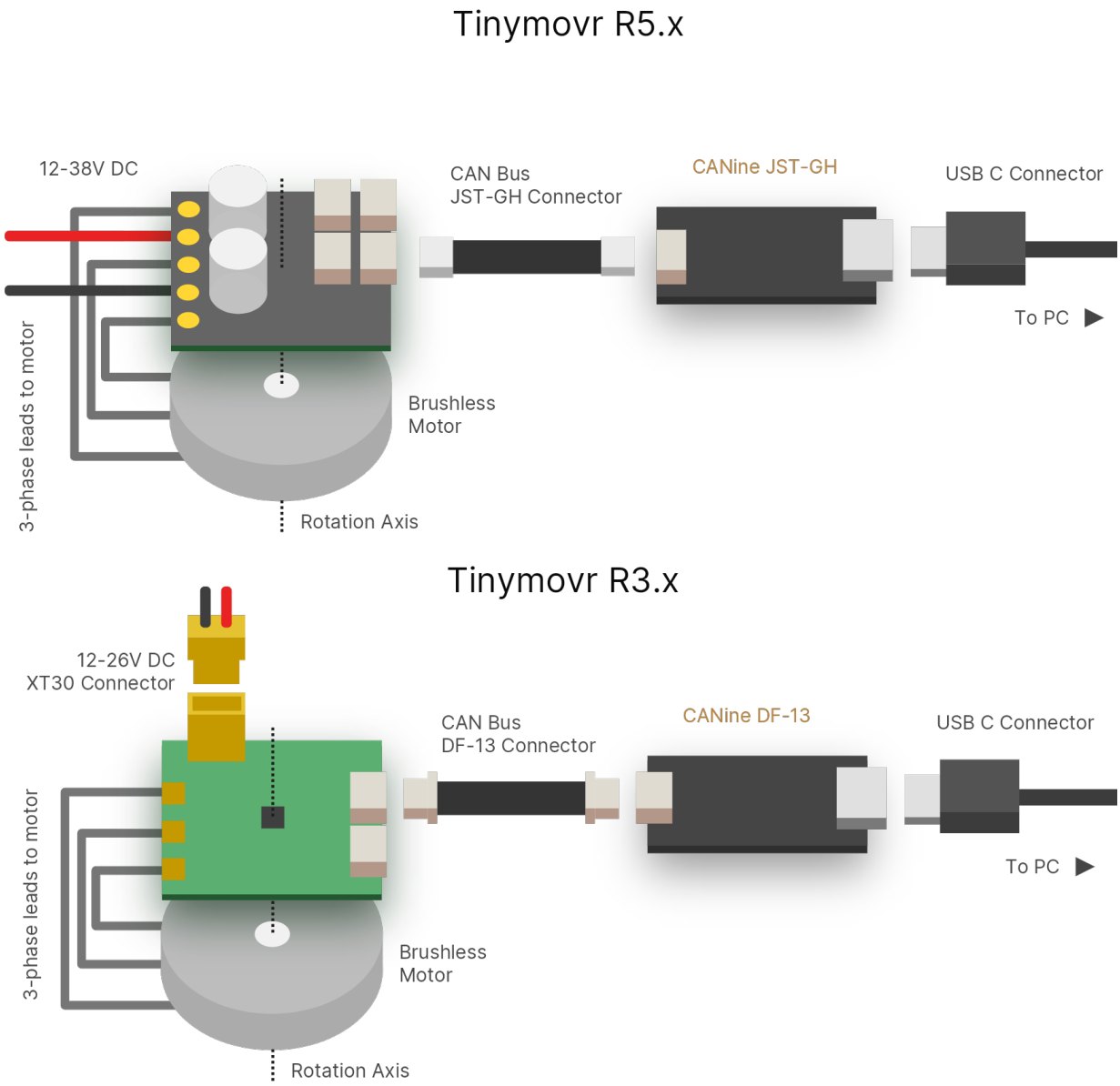
> [...] this type of arrangement is possible, what really matters in the end is that there is enough magnetic field reaching the sensor. Of course the minimum distance is imposed by the thickness of the PCB, so it puts some constraints on the design, that you have to take into account when chosing the magnet (you can use our online simulation tool for that). But as long as the PCB is not acting as a magnetic shield (due to copper plane), then it is fine.

Mounting Tips

- Ensure the encoder magnet is firmly attached to the motor shaft, otherwise it may slip out of sync. Use strong adhesive to secure.
- Calibration needs to be performed without any loads on the motor. If the motor is coupled to a load, the encoder offset angle may not be determined correctly, leading to a sub-optimal setup.
- Adjust your termination resistor DIP switch (if needed) before putting together your actuator, to avoid needing to disassemble it for adjustment later on. See also *Connecting Data*.

3.4 Electrical Setup

Electrical setup comprises three main parts: Motor connection, data connection and power connection. Below is a diagram with the electrical connection scheme.



3.5 Connecting Motor

Connect the three motor phases to the three terminals on Tinymovr. The order of connection is not important, motor direction will be determined during motor/encoder calibration.

The connection can be achieved in two ways. Either by soldering the motor leads directly on the terminals, or by securing the leads with a 2mm lug.

Note: If using a lug connection, ensure that the screw and nut are not rotating against the PCB surface, as this may remove parts of the soldermask or even conductive copper layer.

3.6 Connecting Data

Connect the CAN bus header to one of the two CAN sockets on the board. It is not important which one you choose. If this is a terminal node in the CAN network, you may need to use a termination resistor, as follows:

- Tinymovr R3.x: flip ONLY the DIP switch labelled "CAN 120R" to on to enable the 120 termination resistor.
- Tinymovr R5.x: you will need to provide an external 120 termination resistor.

In small setups with few nodes and short wires, it is better to enable just a single termination resistor, either on one Tinymovr board or on the CAN adapter. In setups with many nodes and long cables, you may need to enable termination resistors in both terminal nodes.

Warning: The UART pins in Tinymovr R5.1 have the silkscreen reversed. If you are planning to use UART with R5.1, consult *Tinymovr R5 UART Silkscreen Reversed*.

3.7 Connecting Power

- Tinymovr R3.x can be powered from a 12-26V (3S-6S) power source.
- Tinymovr R5.x can be powered from a 12-38V (3S-9S) power source.

With the power source off/disconnected, connect the power leads observing correct polarity. Turn on/connect the power source. Upon successful power-up, the onboard LED should light up.

Note: Each Tinymovr board has a capacitance of around 500F/160. Such capacitance can introduce significant inrush current upon power-on, especially if several boards are connected to the same power supply. To prevent damage to components from overcurrent, the use of an inrush current limiter or a current-limited power supply is advised.

CHAPTER

FOUR

STUDIO INSTALLATION

4.1 Preparation

If using Windows with a CANtact-compatible CAN Bus adapter, such as CANine, you will need to install an .inf file to enable proper device naming. You can download the inf file here. Extract the archive, right click and select Install.

4.2 Using pip

This is the most straightforward method to install Tinymovr studio and have access to hardware.

```
pip3 install tinymovr
tinymovr
```

You should now be looking at the Tinymovr Studio IPython interface.

4.2.1 Anaconda Installation

Tinymovr can be installed inside a Virtualenv or Anaconda environment.

With Anaconda, create a new environment:

```
conda create --name tinymovr python=3.7 -y
conda activate tinymovr
```

Alternatively you can use any Python version >= 3.6.

Then simply install and run Tinymovr:

```
pip install tinymovr
tinymovr
```

4.3 Using git clone

Note: The master branch of the Github repository represents the state of art of development, and it may contain bugs. For a stable version, especially if you are starting with the project, please consider installing Tinymovr Studio using pip as shown above.

First clone the Tinymovr repo to a local directory:

```
git clone https://github.com/yconst/Tinymovr
```

Then cd to the cloned repo directory and install in developer mode:

```
cd Tinymovr/studio/Python
pip3 install -e .
```

CHAPTER

FIVE

STUDIO USAGE

5.1 Overview

Tinymovr Studio is an IPython application that enables communication with multiple Tinymovr instances, allowing configuration and control.

5.2 Launching the command line app

```
tinymovr
```

5.3 Discovery

Tinymovr Studio uses a polling mechanism to discover Tinymovr nodes in the CAN network. Upon launching the app, by default the first ten nodes are scanned. The ones that are present are available through the variable handle 'tmx', where x the device index. To specify the scan range, take a look at *Command-line options*.

5.4 Alternative Adapters/Firmwares

By default Tinymovr Studio searches for slcan-compatible CAN bus adapters, which appear as USB Virtual COM Port devices. To specify an alternative device, use the *–bustype* and *–chan* command line arguments.

For instance, to work with adapters using the CANine firmware, launch Tinymovr Studio as follows:

```
tinymovr --bustype=canine -chan=0xC1B0
```

Note that you need to have completed setting up CANine before issuing the above command to work with CANine.

5.5 Compatibility

Tinymovr Studio includes by default a version check to determine compatibility of firmware with studio version. This is performed each time a node is discovered, and prior to initializing the tinymovr object. Compatibility is determined through comparison of versions with minimun requirements embedded in both firmware and studio. If you wish to disable version check (e.g. for backwards compatibility), you can use the `--no-version-check` command line argument. Please note that unexpected things can happen if you combine incompatible firmware and studio versions.

5.6 Issuing Commands

You can read variables and issue commands using the respective Tinymovr handle, e.g.:

```
tmx.device_info
```

or

```
tmx.set_vel_setpoint(0)
```

Where x is the device ID. Full tab completion is available.

5.7 Multiple Instances

In order for multiple Tinymovr instances to coexist in the same network, they need to have unique IDs. The default ID is 1. To assign different IDs to each board, follow the method below:

1. Connect a single Tinymovr to the bus and launch Studio. The board will be assigned the default ID, 1, and will be accessible as tm1.
2. Issue the id change

```
tm1.set_can_config(x)
```

where x is the desired ID. Valid IDs are from 1-64, but the Studio app currently discovers IDs up to 8.

3. Relaunch Studio
4. The board will be discovered with the new ID. Save configuration.

```
tmx.save_config()
```

5. Power down or reset the board. Tinymovr is now ready to use with the new ID.

5.8 Command-line options

Tinymovr Studio supports the following command line options.

5.8.1 --ids=<ids>

The –ids option specifies a set of CAN node IDs to scan.

Example:

```
tinymovr --ids=1,3,5,7-9
```

All syntax options supported by Pynumparser are available.

5.8.2 --bustype=<bustype>

The –bustype option specifies a CAN bus type to use.

Example:

```
tinymovr --bustype=robotell
```

All interfaces offered by python-can are supported.

5.8.3 --chan=<chan>

The –chan options specifies a channel to use, optionally together with the –bustype option.

Example:

```
tinymovr --bustype=robotell --chan=COM3
```

By default, Tinymovr Studio will use slcan as the interface, and will search for CANAble/CANtact-type devices with slcan firmware. Such is the CANine adapter supplied with Tinymovr Servo Kits.

5.8.4 --no-version-check

Disables the firmware-studio version compatibility check that is performed by default when discovering a Tinymovr node.

5.9 Units

Tinymovr Studio introduced physical units and quantities since v0.3.0. Units are introduced through the Pint package. Using units you will see all values that you query associated with a unit, which forms a physical quantity.

With units, you can do the following:

```
In [1]: tm1.encoder_estimates
Out[1]: {'position': 0.0 <Unit('tick')>, 'velocity': 0.0 <Unit('tick / second')>}
```

You can also set quantities in any (defined) unit you wish. For instance:

```
In [1]: tm1.set_pos_setpoint(2.0 * ureg('rad'))
```

The above will set the rotor position to 2 radians from the initial position. Similarly for velocity:

```
In [1]: tm1.set_vel_setpoint(2.0 * ureg('rad/s'))
```

Will set velocity to 3 radians/second. If not unit is used in setting a value, the default units will be assumed, in the above cases ticks and ticks/second.

The ureg object is the unit registry, and it is that which holds all unit definitions. You can use it to do all sorts of cool stuff such as doing conversions, defining your own shortcuts or even new units.

For instance, to define a few frequently used shortcuts in a program:

```
from tinymovr.units import get_registry
ureg = get_registry()
mA = ureg.milliampere
rad = ureg.radian
s = ureg.second
```

Then you can use the defined shortcuts to intuitively set values, such as a position setpoint with velocity and current feed-forwards:

```
tm.set_pos_setpoint(2*PI * rad, 0 * rad/s, 1500 * mA)
```

Take a look at the *API Reference* for default units used in each command.

For more information on units and their usage, take a look at Pint's documentation

5.10 Plotting

Tinymovr Studio features a capable and fast plotter to visualize your setup in real time. The plotter is accessible from within the IPython terminal that hosts Tinymovr Studio.

5.10.1 Example: Plotting Encoder Estimates

Let us imagine that we want to plot the position and velocity estimates of our encoder. The following will do the trick:

```
plot(lambda: [tm1.encoder_estimates])
```

A plot window will show up on screen. Notice that both values (position and velocity) are plotted. This is because we passed the endpoint itself as an argument. The plotter is smart enough to know to expand the returned dictionary, and assign values to the correct keys.

Also note that there are two y-axes in the plot, one on the left and one on the right. Each of these corresponds to one value being plotted, and will adjust to that value's range. You can have as many values as you wish, the plotter will add so-called 'parasite' axes on the right side of the plot for each value. However with more than three or four it doesn't really look pretty…

5.10.2 Plotting values from multiple endpoints

It is possible to plot multiple endpoints from the same or multiple devices with the same syntax:

```
plot(lambda: [tm1.encoder_estimates, tm1.setpoints])
```

In the above example, estimates are plotted together with setpoints for position and velocity.

5.11 Socketcan & Linux

You can use a socketcan-enabled CAN adapter with Tinymovr Studio. The CANine adapter supplied with Tinymovr Servo Kits supports Socketcan natively with the alternative Candlelight Firmware. To connect to a Socketcan device, run Studio as follows:

```
tinymovr --bustype=socketcan --chan=CAN0
```

5.12 Tinymovr in-silico

Tinymovr studio implements a simplistic simulation of the actual controller, in order to facilitate validation of basic commands etc. To use the simulation mode, run Studio as follows:

```
tinymovr --bustype=insilico --chan=test
```

Basic commands such as `state`, `encoder_estimates`, `set_pos_setpoint` work, more to be implemented soon.

CHAPTER

SIX

TRAJECTORY PLANNER

Since firmware version 0.8.8 (studio 0.3.8), Tinymovr can use a trapezoidal trajectory planner to generate acceleration- and velocity-limited trajectories, as well as time-limited trajectories for multi-axis synchronization. The planner executes at the motor control rate, which at the moment equals the PWM rate of 20kHz.

6.1 Trapezoidal Trajectories

Trapezoidal trajectories are named after the shape of the velocity profile, which is a trapezoid. As such, it comprises a constant acceleration phase, where the velocity increases linearly, a constant velocity (cruise) phase, and a constant deceleration phase, where the velocity linearly decreases.

6.2 Acceleration and Velocity-Limited Trajectory

The acceleration- and velocity-limited planner generates trajectories that respect constraints on acceleration cruise velocity and deceleration set by the user. The corresponding time windows for each phase are computed by the planner.

Using the Tinymovr API, an acceleration- and velocity-limited trajectory has the following signatures:

```
tm1.set_max_plan_acc_dec(max_acc, max_dec)
tm1.plan_v_limit(pos_setpoint, max_vel)
```

The first function call sets the max acceleration and deceleration parameters – it does not execute a trajectory. Once you set the desired acceleration and deceleration parameters, they do not need to be re-set – they are stored in RAM. However, upon device restart they need to be re-set, as they are not stored in non-volatile memory. The second function executes the trajectory.

As an example, a trajectory is defined and executed as follows:

```
tm1.set_max_plan_acc_dec(100000, 100000)
tm1.plan_v_limit(100000, 50000)
```

6.3 Time-Limited Trapezoidal Velocity Trajectory

The time-limited trajectory planner generates trajectories whose acceleration, cruise and deceleration phases are defined by the user. The corresponding acceleration, velocity and deceleration values are computed by the planner. In this sense, it can be thought as the 'inverse' process of the acceleration and velocity limited planner.

Using the Tinymovr API, a time-limited trajectory has the following signature:

```
tm1.plan_t_limit(pos_setpoint, total_time, acc_percent, dec_percent)
```

Please note that due to CAN frame limitations, the maximum trajectory time can not be longer than 65 seconds. In addition, the "acc_percent" and "dec_percent" parameters are expressed in the 0-255 range, with 0-0% and 255-100%.

As an example, a time-limited trajectory is executed as follows:

```
tm1.plan_t_limit(100000, 3000, 50, 50)
```

6.4 Multi-axis Synchronization

Time-limited trajectories are useful for synchronizing the acceleration, cruise and deceleration phases for multiple axes. For instance, to synchronize three axes with different setpoints, you would issue the following three commands to three different controllers in sequence:

```
tm1.plan_t_limit(100000, 5000, 50, 50)
tm2.plan_t_limit(5000, 5000, 50, 50)
tm3.plan_t_limit(200000, 5000, 50, 50)
```

This will generate three trajectories that will start and end at the same time. Only the first parameter (the setpoint) changes, while the other ones are the same for all trajectories. This guarantees that the trajectory phases are synchronized.

CHAPTER

SEVEN

ENCODERS

7.1 Overview

This document provides information and guides for using the various encoder types supported by Tinymovr.

7.2 Observer bandwidth

Tinymovr uses an observer in order to filter readings from the encoders. The bandwidth value corresponds to the desired observer bandwidth. It is a configurable value and depends on the dynamics that you wish to achieve with your motor. Keep in mind that high bandwidth values used with motors with fewer pole pairs will make the motors oscillate around the setpoint and have a rough tracking performance (perceivable "knocks" when the rotor moves). On the other hand, too low of a bandwidth value may cause the motor to lose tracking in highly dynamic motions. If you are certain such motions will not be possible (e.g. in heavy moving platforms) you may reduce the bandwidth to ensure smoother motion.

7.3 Onboard Magnetic

All Tinymovr controllers feature an onboard magnetic absolute angle sensor that allows high precision angle measurement for efficient commutation and high bandwidth motor control. This is enabled by default and does not require any specific setup, apart from initial encoder/motor calibration.

The onboard sensor is enabled by default, so no special configuration is necessary. Should you need to switch to the onboard sensor, use the following commands:

```
tm1.set_encoder_config(0, 1500) # encoder type, bandwidth
tm1.save_config()
tm1.reset() # encoder change is applied after reset
```

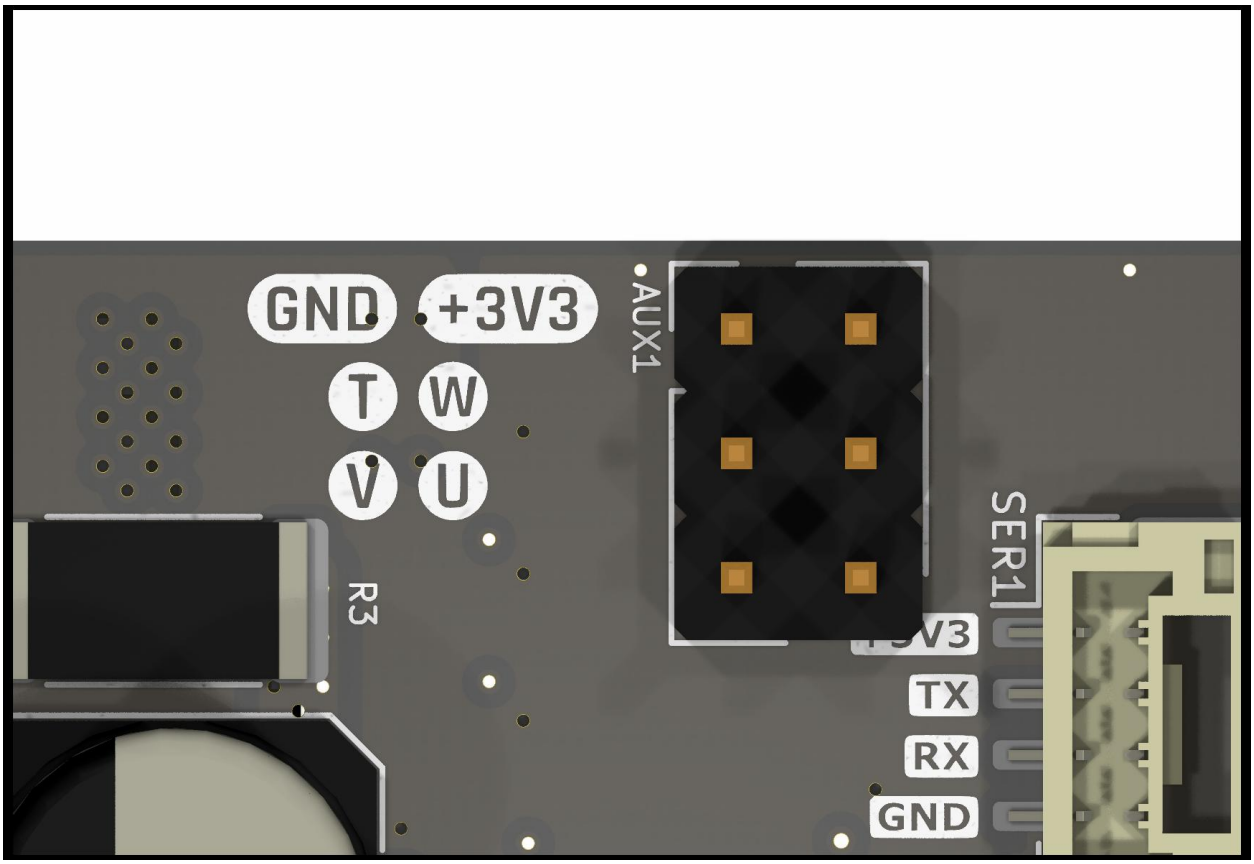
The value of 1500 in bandwidth is the default value configured for the onboard sensor, which works well for most configurations.

7.4 Hall Effect Sensor

Tinymovr R5 supports external Hall effect sensors for commutation and positioning. Hall effect sensors generate a specific sequence in the 3 phase encoder signal as the rotor moves. By reading this sequence, the rotor position is determined in one of six 60 degree sectors along the electrical cycle.

7.4.1 Hardware Setup

To use Hall effect sensors, you need to connect the sensor's power supply, phases and ground to the correct pins on the Tinymovr board. The pinout for the Hall effect sensor connector is shown below.



Note: The diagram shows the connector side of the board, i.e. the side where the CAN, UART and SPI connectors, and also the DC-link capacitors are located.

Note the U, V and W pins. These need to be connected to the respective pins of the sensor. The pin labeled T is currently not in use. In addition, the 3.3V power supply and the GND need to be connected to the sensor as well.

Note: Tinymovr supplies 3.3V on the AUX power supply pin. If your sensor uses 5V, or if it needs more than 50mA, you'll need to provide power externally, e.g. through a dedicated buck converter.

7.4.2 Configuration

As a first step you need to configure the sensor type and observer bandwidth.

```
tm1.set_encoder_config(1, 100) # encoder type, bandwidth
```

This sets the encoder type to Hall effect sensor, and the encoder bandwidth to 100.

Next, you need to set the motor configuration:

```
tm1.set_motor_config(0, 15, 4) # motor type, pole pairs, calibration current
tm1.save_config()
tm1.reset() # encoder change is applied after reset
```

This sets the motor type and pole pairs, and restarts Tinymovr. The board needs to be reset following saving of the config, to enable the encoder change. For safety reasons, any change to the encoder type is only enabled at next boot.

Next comes tuning of gains. Gains are determined on the resolution of a full mechanical turn fo the motor. When using the onboard magnetic sensor, the resolution is fixed to 8192 ticks. However, when using the Hall effect sensor, the mechanical resolution is variable, and amounts to *6 * pole_pair_count*. As such, if you have a 15 pp motor, your mechanical resolution would be 90.

Because of this vast change in resolution (almost 2 orders of magnitude), the gains need to be updated:

```
tm1.set_gains(5, 0.07) # position gain, velocity gain
```

The values above are just an example using a 15 pp hoverboard motor. For your own motor, you need to determine these experimentally. In position control mode, start by raising the default velocity gain until your motor experiences oscillations. The back up by a factor of two, and repeat the same for velocity control. This simple tuning heuristic does not result in an optimal configuration but the gains are workable.

Last step is motor/encoder calibration:

```
tm1.calibrate()
```

After calibration finishes, you should be able to control the motor:

```
tm1.velocity_control()
tm1.set_vel_setpoint(100) # around 60 rpm for a 15 pp motor
```

The motor should now move at a constant velocity.

CHAPTER

EIGHT

API

8.1 Overview

This document outlines the main API used to interface with Tinymovr. This API comprises a series of read/write endpoints. The endpoints are defined taking into account the capabilities and constraints of the CAN bus, the main communication bus used by Tinymovr.

The Tinymovr API allows full hardware control from within Python scripts, using a high-level interface to hardware. At the same time, it is possible to interface directly with the CAN bus endpoints, for instance in an embedded application. In both cases, the *API Reference* provides all the necessary information.

Tinymovr API is part of Tinymovr Studio. For help installing Studio, please take a look at *Studio Installation*.

8.2 Use with Python

Here below is an example using the API from Python scripts and controlling hardware:

```
import can
from tinymovr import Tinymovr
from tinymovr.iface.can_bus import CAN

bus = can.Bus(bustype="slcan", channel="your_com_port", bitrate=1000000)
iface = CAN(bus)
tm = Tinymovr(node_id=1, iface=iface)

tm.calibrate()
```

The above code block will instantiate a Tinymovr with CAN bus id of 1 and calibrate it. Following the above, you can issue commands such as:

```
tm.position_control()
tm.set_pos_setpoint(0)

tm.velocity_control()
tm.set_vel_setpoint(80000)
```

8.3 Shortcuts

A few commands exist in Tinymovr studio which are shortcuts to Tinymovr endpoints with specific argument values. These are mostly shortcuts to changes of states and modes that are easier to read and remember. They are briefly presented below.

8.3.1 idle()

Switch to idle mode and disable the inverter.

Shortcut for `tmx.set_state(0, 0)`

8.3.2 calibrate()

Start calibration sequence. This command presents a safety notice that needs to be accepted by the user before proceeding.

Shortcut for `tmx.set_state(1, 0)`

8.3.3 position_control()

Switch to closed loop control (enable the inverter) and to position control mode.

Shortcut for `tmx.set_state(2, 2)`

8.3.4 velocity_control()

Switch to closed loop control (enable the inverter) and to velocity control mode.

Shortcut for `tmx.set_state(2, 1)`

8.3.5 current_control()

Switch to closed loop control (enable the inverter) and to current control mode.

Shortcut for `tmx.set_state(2, 0)`

8.4 API Reference

Note: Where "float32" is mentioned, an IEEE 754, 32-bit floating point representation is assumed.

8.4.1 state

endpoint: `0x03`
type: Read-only

Retrieves an object containing the controller state, control mode and error flags. The object is pretty-printed if inside the Tinymovr Studio iPython environment.

This command has been revised as of firmware 0.8.2 and studio 0.3.3 to report multiple error flags if available. The above and newer versions can display up to five error flags simultaneously, and with the order that they were registered by the firmware error handler.

Tinymovr Studio 0.3.3 and newer is backwards compatible with the legacy error reporting system, as such newer Studio versions can be used with older firmware. However, newer firmware (0.8.2 and later) is not compatible with older Studio versions. Make sure that you run the latest version of Studio before upgrading your firmware.

Return Values

Member	Description	Data Type	Data Offset
`errors`	Legacy Error Flag	uint8	0
`state`	Control State	uint8	1
`mode`	Control Mode	uint8	2
`errors`	1st Error Flag	uint8	3
`...`	2nd Error Flag	uint8	4
`...`	3rd Error Flag	uint8	5
`...`	4th Error Flag	uint8	6
`...`	5th Error Flag	uint8	7

Example

Legacy system

```
>>>tmx.state
{"error": 0, "state": 0, "mode": 0}
```

New system

```
>>>tmx.state
State: Idle     Mode: Position
Errors:
    Invalid State (1): Attempt to transition to invalid state

>>>tmx.state.mode
0
```

8.4.2 set_state()

endpoint: `0x07`
type: Write-only

Sets the controller state and control mode.

Note: Results of calibration are not automatically saved to Non-Volatile Memory (NVM). You need to issue a `save_config` command after calibration is finished to save calibration data to NVM.

Arguments

Member	Description	Data Type	Data Offset
`state`	Control State	uint8	1
`mode`	Control Mode	uint8	2

Example

```
>>>tmx.set_state(state=0, mode=0)
```

8.4.3 can_config

endpoint: `0x05`
type: Read-only

Retrieves the CAN configuration.

Return Values

Member	Description	Data Type	Data Offset
`id`	CAN Bus ID	uint8	0
`baud_rate`	Baud Rate	uint16	1

Example

```
>>>tmx.can_config
{"id": 1, "baud_rate": 250}
```

8.4.4 set_can_config()

endpoint: `0x06`
type: Write-only

Sets the CAN configuration.

Arguments

Member	Description	Data Type	Data Offset
`id`	CAN Bus ID	uint8	0
`baud_rate`	Baud Rate	uint16	1

Example

```
>>>tmx.set_can_config(id=1, baud_rate=250)
```

8.4.5 encoder_estimates

endpoint: `0x09`
type: Read-only

Retrieves the position and velocity encoder estimates.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Estimate	float32	0	ticks
`velocity`	Velocity Estimate	float32	4	ticks/second

Example

```
>>>tmx.encoder_estimates
{"position": 1000.0, "velocity": 0.0}
```

8.4.6 setpoints

endpoint: `0x0A`
type: Read-only

Retrieves the position and velocity setpoints of the controller.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Setpoint	float32	0	tick
`velocity`	Velocity Setpoint	float32	4	tick/second

Example

```
>>>tmx.setpoints
{"position": 1000.0, "velocity": 0.0}
```

8.4.7 encoder_config

endpoint: `0x0B`
type: Read-only

Retrieves the encoder configuration.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`type`	Encoder Type	uint8	0	
`bandwidth`	Encoder Bandwidth	float32	1	radians/second

Example

```
>>>tmx.encoder_config
{"type": 0, "bandwidth": 1500.0}
```

8.4.8 set_pos_setpoint()

endpoint: `0x0C`
type: Write-only

Sets the position setpoint, and optionally velocity and current feed-forward values. Due to the fact that data types of feed-forward values are limited by type, multiples of the root units are used.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Setpoint	float32	0	tick
`velocity`	Velocity Setpoint	int16	4	decatick/second
`current`	Current Setpoint	int16	6	centiampere

Example

```
>>>tmx.set_pos_setpoint(1000.0)
```

```
>>>tmx.set_pos_setpoint(position=1000.0, velocity=10000.0, current=0.0)
```

8.4.9 set_vel_setpoint()

endpoint: `0x0D`
type: Write-only

Sets the velocity setpoint, and optionally current feed-forward value.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`velocity`	Velocity Setpoint	float32	0	ticks/second
`current`	Current Setpoint	float32	4	ampere

Example

```
>>>tmx.set_vel_setpoint(10000.0)
```

```
>>>tmx.set_vel_setpoint(velocity=10000.0, current=0.0)
```

8.4.10 set_cur_setpoint()

endpoint: `0x0E`
type: Write-only

Sets the current (Iq) setpoint.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`current`	Current Setpoint	float32	0	amperes

Example

```
>>>tmx.set_cur_setpoint(0.5)
```

8.4.11 limits

endpoint: `0x15`
type: Read-only

Retrieves the velocity and current limits of the controller.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`velocity`	Velocity Limit	float32	0	tick/second
`current`	Current Limit	float32	4	ampere

Example

```
>>>tmx.limits
{"velocity": 300000.0, "current": 10.0}
```

8.4.12 set_limits()

endpoint: `0x0F`
type: Write-only

Sets the velocity and current limits of the controller.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`velocity`	Velocity Limit	float32	0	tick/second
`current`	Current Limit	float32	4	ampere

Example

```
>>>tmx.set_limits(velocity=200000.0, current=15.0)
```

8.4.13 gains

endpoint: `0x18`
type: Read-only

Retrieves the position and velocity gains of the controller.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Gain	float32	0	1/second
`velocity`	Velocity Gain	float32	4	ampere*second/tick

Example

```
>>>tmx.gains
{"position": 35.0, "velocity": 0.000012}
```

8.4.14 set_gains()

endpoint: `0x19`
type: Write-only

Sets the position and velocity gains of the controller.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Gain	float32	0	1/second
`velocity`	Velocity Gain	float32	4	ampere*second/tick

Example

```
>>>tmx.set_gains(position=25.0, velocity=0.00001)
```

8.4.15 offset_dir

endpoint: `0x02`
type: Read-only

Retrieves the user defined rotor position offset and rotor direction values.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`offset`	Offset	float32	0	tick
`direction`	Direction	int8	4	

Example

```
>>>tmx.offset_dir
{"offset": 0.0, "direction": 1}
```

8.4.16 set_offset_dir()

endpoint: `0x08`
type: Write-only

Sets the user defined rotor position offset and rotor direction values.

Note: The `direction` field only accepts -1 or 1 as values. All other values are ignored.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`offset`	Offset	float32	0	tick
`direction`	Direction	int8	4	

Example

```
>>>tmx.set_gains(offset=2500, direction=-1)
```

8.4.17 vel_integrator_params

endpoint: `0x18`
type: Read-only

Retrieves the velocity integrator gain and deadband parameters.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`gain`	Velocity Integrator Gain	float32	0	ampere*second/tick
`deadband`	Velocity Integrator Deadband	float32	4	tick

Example

```
>>>tmx.vel_integrator_params
{"gain": 0.0001, deadband: 200}
```

8.4.18 set_vel_integrator_params()

endpoint: `0x19`
type: Write-only

Sets the velocity integrator gain and deadband parameters.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`gain`	Velocity Integrator Gain	float32	0	ampere*second/tick
`deadband`	Velocity Integrator Deadband	float32	4	tick

Example

```
>>>tmx.set_vel_integrator_params(gain=0.0001, deadband=300)
```

8.4.19 Iq

endpoint: `0x14`
type: Read-only

Retrieves the current (Iq) setpoint and estimate.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`setpoint`	Iq Setpoint	float32	0	ampere
`estimate`	Iq Estimate	float32	4	ampere

Example

```
>>>tmx.Iq
{"setpoint": 1.0, "estimate": 0.9}
```

8.4.20 Iphase

endpoint: `0x10`
type: Read-only

Retrieves the measured phase currents.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
A	A Phase Current	int16	0	ampere
B	B Phase Current	int16	0	ampere
C	C Phase Current	int16	0	ampere

Example

```
>>>tmx.Iphase
{"A": 1.0, "B": -0.6, "C": -0.4}
```

8.4.21 set_encoder_config

endpoint: 0x11
type: Write-only

Sets the encoder configuration.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
type	Encoder Type	uint8	0	
bandwidth	Encoder Bandwidth	float32	1	radians/second

Example

```
>>>tmx.set_encoder_config(0, 1500)
```

8.4.22 plan_t_limit

endpoint: 0x20
type: Write-only

Generate and execute a time-limited trajectory.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`target_position`	Target Position	float	0	tick
`total_time`	Total Trajectory Time	uint16	4	millisecond
`acc_percent`	Acceleration Phase Percent	uint8	6	(none, values 0-255)
`dec_percent`	Deceleration Phase Percent	uint8	7	(none, values 0-255)

Example

```
>>>tmx.plan_t_limit(100000, 3000, 50, 50)
```

8.4.23 plan_v_limit

endpoint: `0x21`
type: Write-only

Generate and execute an acceleration- and velocity-limited trajectory.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`target_position`	Target Position	float	0	tick
`max_velocity`	Max Velocity	float	4	tick/second

Example

```
>>>tmx.plan_v_limit(100000, 50000)
```

8.4.24 set_max_plan_acc_dec

endpoint: `0x22`
type: Write-only

Set maximum acceleration and deceleration values for trajectory generation.

Note: This command only sets values, it does not execute a trajectory. For trajecotry execution with set values, make a call to `plan_v_limit`.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
max_acc	Max Acceleration	float	0	tick/(second^2)
max_dec	Max Deceleration	float	4	tick/(second^2)

Example

```
>>>tmx.set_max_plan_acc_dec(50000, 50000)
```

8.4.25 device_info

endpoint: 0x1A
type: Read-only

Retrieves device-related information.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
device_id	Device ID	uint32	0	
fw_major	FW Major Ver.	uint8	4	
fw_minor	FW Minor Ver.	uint8	5	
fw_patch	FW Patch Ver.	uint8	6	
temp	MCU Temp	uint8	7	°C

Example

```
>>>tmx.device_info
{"device_id": 99999, "fw_major": 0, "fw_minor": 7, "fw_patch": 1, "temp": 45}
```

8.4.26 motor_config

endpoint: 0x1E
type: Read-only

Retrieves motor config (flags, pole pairs, calibration current).

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`flags`	Calibrated, Gimbal	uint8	0	
`pole_pairs`	Motor Pole Pairs	uint8	1	
`I_cal`	Calibration Current	float	2	ampere

Example

```
>>>tmx.motor_config
{"flags": 1, "pole_pairs": 11, "I_cal": 5.0}
```

8.4.27 set_motor_config

endpoint: `0x1F`
type: Write-only

Sets motor config (flags, pole pairs, calibration current).

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`flags`	Gimbal	uint8	0	
`pole_pairs`	Motor Pole Pairs	uint8	1	
`I_cal`	Calibration Current	float	2	ampere

Example

High-current motor: .. code-block:: python

```
>>>tmx.set_motor_config(0, 14, 5)
```

Gimbal motor: .. code-block:: python

```
>>>tmx.set_motor_config(1, 14, 0.5)
```

8.4.28 timings

endpoint: `0x1B`
type: Read-only

Retrieves MCU timings in each control cycle.

Return Values

Member	Description	Data Type	Data Offset
`total`	Total MCU Cycles	uint32	0
`busy`	Busy MCU Cycles	uint32	4

Example

```
>>>tmx.timings
{"total": 7500, "busy": 1000}
```

8.4.29 estop()

endpoint: `0x02`
type: Write-only

Emergency stop: Idles the MCU immediately.

Arguments

No arguments.

Example

```
>>>tmx.estop()
```

8.4.30 reset()

endpoint: `0x16`
type: Write-only

Resets the MCU.

Arguments

No arguments.

Example

```
>>>tmx.reset()
```

8.4.31 save_config()

endpoint: `0x1C`
type: Write-only

Saves board configuration to Non-Volatile Memory.

Note: Saving config only works when Tinymovr is in idle mode, otherwise the command is ignored.

Arguments

No arguments.

Example

```
>>>tmx.save_config()
```

8.4.32 erase_config()

endpoint: `0x1D`
type: Write-only

Erases the configuration stored in NVM and resets the MCU.

Note: Erasing config only works when Tinymovr is in idle mode, otherwise the command is ignored.

Arguments

No arguments.

Example

```
>>>tmx.erase_config()
```

8.4.33 get_set_pos_vel()

endpoint: `0x25`
type: Read-Write

Gets and sets Position and Velocity feedforward in one go.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Setpoint	float32	0	ticks
`velocity`	Velocity Setpoint	float32	4	ticks/second

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Estimate	float32	0	ticks
`velocity`	Velocity Estimate	float32	4	ticks/second

Example

```
>>>tmx.get_set_pos_vel(1000.0, 0)
{"position":0.0, "velocity": 0.0}
```

8.4.34 get_set_pos_vel_Iq()

endpoint: `0x26`
type: Read-Write

Get and set Position, Velocity feedforward and Iq feedforward in one go. Due to the fact that data types of feed-forward values are limited by type, multiples of the root units are used.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Setpoint	float32	0	tick
`velocity`	Velocity Setpoint	int16	4	decatick/second
`current`	Current Setpoint	int16	6	centiampere

Return Values

Member	Description	Data Type	Data Offset	Default Unit
`position`	Position Estimate	float32	0	tick
`velocity`	Velocity Estimate	int16	4	decatick/second
`current`	Current Estimate	int16	6	centiampere

Example

```
>>>tmx.get_set_pos_vel_Iq(1000.0, 0, 0)
{"position":0.0, "velocity": 0.0, "current": 0.0}
```

8.4.35 motor_RL

endpoint: `0x27`
type: Read-only

Retrieves motor resistance and inductance values.

Return Values

Member	Description	Data Type	Data Offset	Default Unit
R	Phase Resistance	float32	0	ohm
L	Phase Inductance	float32	4	henry

Example

```
>>>tmx.motor_RL
{"R": 0.2, "L": 0.00005}
```

8.4.36 set_motor_RL

endpoint: `0x28`
type: Write-only

Sets attached motor resistance and inductance values.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`R`	Phase Resistance	float32	0	ohm
`L`	Phase Inductance	float32	4	henry

Example

```
>>>tmx.set_motor_RL(0.5, 0.0001)
```

8.4.37 set_watchdog

endpoint: `0x2A`
type: Write-only

Enables/disables the CAN timeout watchdog, and sets the timeout length in seconds. This watchdog sets the control state to idle after a period of inactivity on the CAN bus. Maximum of 536s.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`enable`	Enable/disable	uint8	0	state
`timeout`	Watchdog timeout	float32	1	second

Example

```
>>>tmx.set_watchdog(1, 5)
```

```
>>>tmx.set_watchdog(0)
```

8.4.38 set_vel_inc

endpoint: `0x2B`
type: Write-only

Sets the maxiumum increment by which the velocity can change each control loop, making it ramp between velocities to reduce voltage spikes. Default of 100. Setting this to 0 disables velocity ramping.

Arguments

Member	Description	Data Type	Data Offset	Default Unit
`increment`	Velocity increment	float32	0	ticks

Example

```
>>>tmx.set_vel_inc(100)
```

8.5 Error Codes

Tinymovr uses error codes to indicate faults in operation. These are listed below. Note that using Tinymovr studio, the error codes are already presented with an explanation.

8.5.1 0: NO_ERROR

No error present.

8.5.2 1: INVALID_STATE

An invalid state has been requested. This can be triggered when attempting to transition to a state whose controller state constraints are not satisfied. E.g. switching to closed loop control without calibrating.

8.5.3 2: ILLEGAL_VALUE

This is a legacy error code that is not in use.

8.5.4 3: VBUS_UNDERVOLTAGE

The bus voltage has dropped below the undervoltage threshold. In a current-limited power supply, this may also indicate excessive current demand from the power supply.

8.5.5 4: OVERCURRENT

The phase current has exceeded the overcurrent threshold. The overcurrent threshold is 1,5 times the user-defined current limit, and in any case no more than 50A.

8.5.6 5: PWM_LIMIT_EXCEEDED

This is a legacy error code that is not in use.

8.5.7 6: PHASE_RESISTANCE_OUT_OF_RANGE

The phase resistance measured during calibration is out of range. The defined range is 5m to 1.

8.5.8 7: PHASE_INDUCTANCE_OUT_OF_RANGE

The phase inductance measured during calibration is out of range. The defined range is 2H to 5mH.

8.5.9 8: INVALID_POLE_PAIRS

The pole pair detection algorithm did not converge near an integer number during calibration.

8.5.10 9: ENCODER_READING_UNSTABLE

Encoder reading variation is over maximum allowed threshold. This is usually the casse if the magnet is misaligned, too far away from the encoder IC, or missing.

CHAPTER

NINE

COMM INTERFACES

9.1 CAN Bus

CAN Bus is the primary interface for communicating with Tinymovr. The physical and data link interface adheres to the CAN 2.0 standard. Tinymovr exposes to CAN all communication endpoints defined in firmware.

9.1.1 Data rate

By default, the CAN baud rate is set to 1Mbit/s. This is customizable both through CAN as well as through UART. See *can_config*. Possible values are 125kbit/s, 250kbit/s, 500kbit/s and 1Mbit/s.

9.1.2 Addressing

The 11-bit identifier of the CAN frame is used for device and endpoint identification. The 6 least significant bits of the identifier are reserved for endpoints, and the 5 most significant bits for device identification. This suggests that the total number of addressable endpoints in a single device are 64, and the total number of addressable devices are 32.

9.1.3 API

For a detailed description, please see *API Reference*.

9.2 UART

As an alternative to CAN Bus, Tinymovr offers UART-based (serial) communication. The protocol is much simpler than can and mainly designed for troubleshooting or testing in the absence of CAN hardware.

Warning: The UART port on Tinymovr is NOT 5V tolerant. Applying 5V voltage will immediately damage the onboard PAC5527 controller. Please use only 3.3V for UART communication.

9.2.1 Protocol Description

The UART port is TTL at 115200 baud. A regular FTDI-style USB to UART adapter should be sufficient.

UART communication is based on a simple human-readable protocole dubbed the "dot protocol", because the dot is the command starting character. The protocol is response-only, meaning that Tinymovr will only respond to commands initiated by the client, it will never initiate a transmission on it's own.

The command template is as follows:

```
.Cxxxx
```

The command begins with a dot. The next single character identifies the command. The characters following the second one are optional values to pass to write commands. Read commands only include the dot and command character. The command is finalized with a newline character (n, not shown above).

For instance, to get the current position estimate:

```
command: .p
response: 1000
```

To set the velocity estimate in encoder ticks:

```
command: .V10000
(no response)
```

The values passed or returned are always integers scaled by the appropriate factor (see command reference below).

Note that command characters are case-sensitive, i.e. capitals and small represent different commands. As a convention, capital letters are setters and small are getters, where applicable.

9.2.2 Command Reference

.Z

Transition to idle state.

Example

```
.Z
0
```

.Q

Transition to calibration state.

Example

```
.Q
0
```

.A

Transition to close loop control state.

Example

```
.A
0
```

.e

Get the error code.

Example

```
.e
0
```

.p

Get position estimate (ticks).

Example

```
.p
1000
```

.v

Get velocity estimate (ticks/s).

Example

```
.v
-200
```

.i

Get current (Iq) estimate (mA).

Example

```
.i
2000
```

.P

Get/set position setpoint (ticks).

Example

```
.P
1000
```

```
.P1000
```

.V

Get/set velocity setpoint (ticks/s).

Example

```
.V
-10000
```

```
.V-10000
```

.I

Get/set current (Iq) setpoint (mA).

Example

```
.I
1000
```

```
.I1000
```

.W

Get/set current (Iq) limit (mA).

Example

```
.W
10000
```

```
.W15000
```

.Y

Get/set position gain.

Example

```
.Y
25
```

```
.Y25
```

.F

Get/set velocity gain (x0.000001).

Example

```
.F
20
```

```
.F20
```

.G

Get/set velocity integrator gain (x0.001).

Note that high values (e.g. above 10) may cause instability.

Example

```
.G
2
```

```
.G2
```

.h

Get motor phase resistance (mOhm).

Example

```
.h
200
```

.l

Get motor phase inductance (H).

Example

```
.l
2000
```

.R

Reset the MCU.

Example

```
.R
```

.S

Save board configuration.

Example

```
.S
```

.X

Erase board configuration and reset.

Example

```
.X
```

CHAPTER

TEN

UPGRADING FIRMWARE

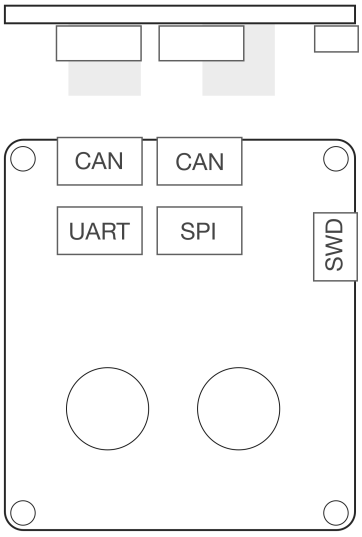
It is possible to upgrade the firmware in two ways: Using the bootloader through UART using the ActiveFlashLight application (Windows only), and through Eclipse using a J-Link adapter.

10.1 Upgrading using bootloader

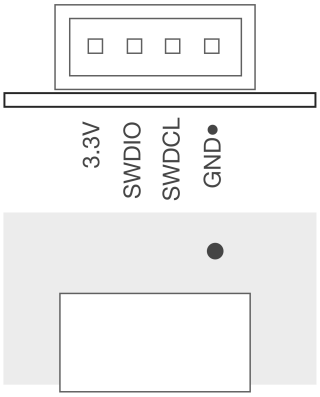
For this method you will need a USB to UART adapter or similar device to connect to Tinymovr's UART interface, such as an FTDI interface or similar. Such devices are very common and inexpensive. Please follow the diagram below to correctly setup the UART interface.

Warning: The UART port on Tinymovr is NOT 5V tolerant. Applying 5V voltage will immediately damage the onboard PAC5527 controller. Please use only 3.3V for UART communication.

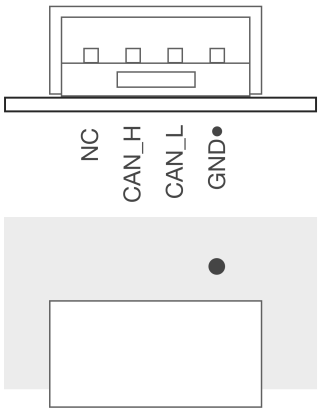
R5



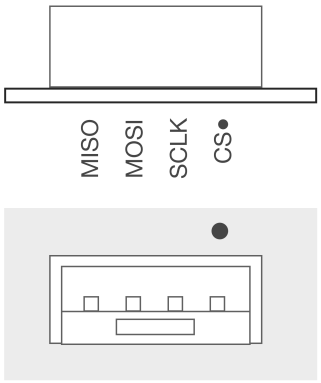
SWD



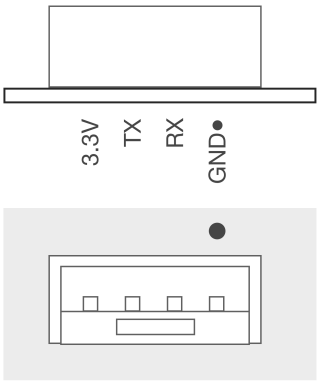
CAN

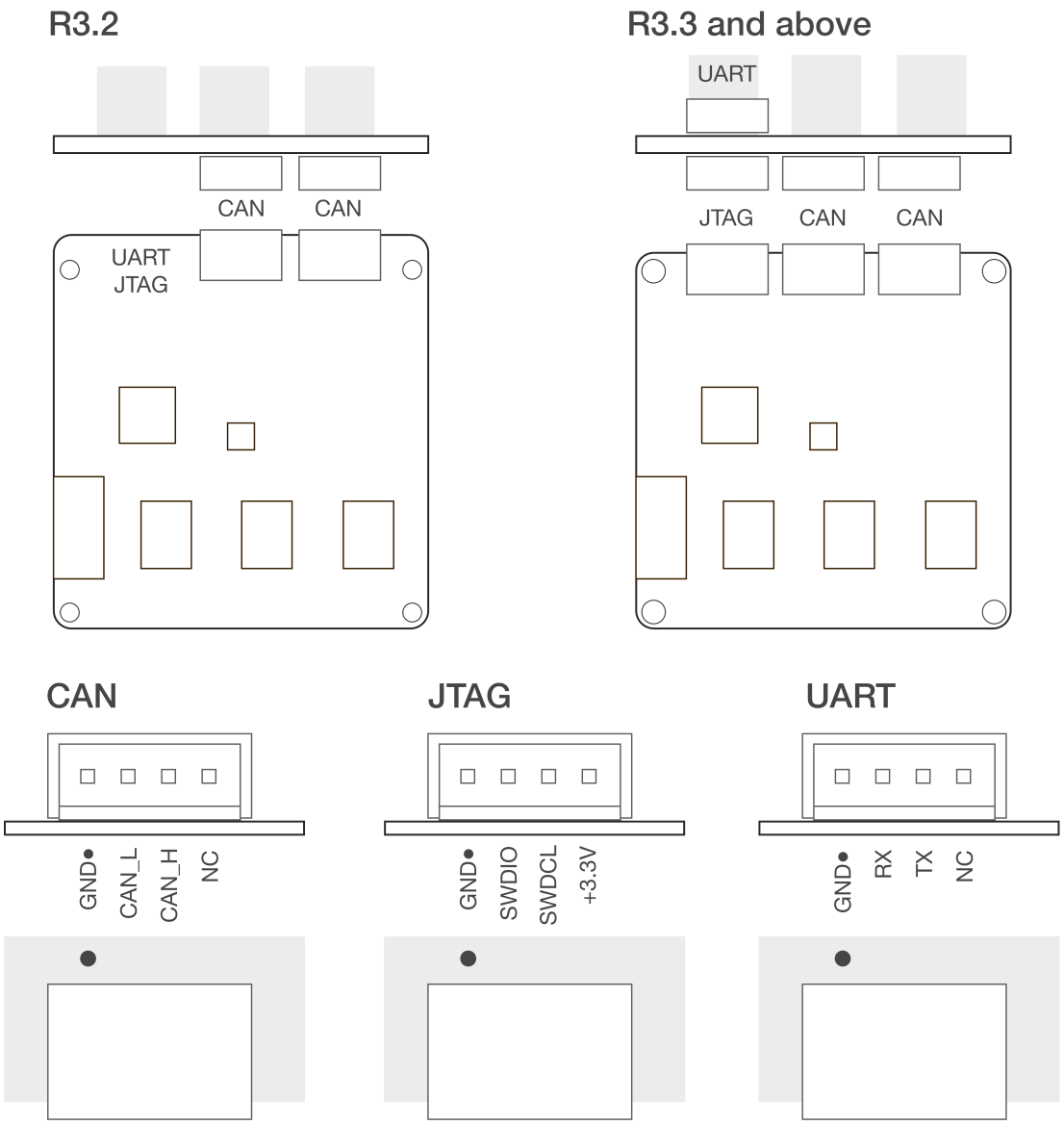


SPI



UART





Qorvo provides an application to interface with the bootloader environment using UART and enable firmware upgrades. It is available through the Qorvo website as an archive named "PAC55xx ActiveFlashLight Bootloader". Please download and extract the archive and follow these steps:

1. Inside the "resources" folder run the ActiveFlashLight application
2. Power on Tinymovr
3. Under 'COMMUNICATION' click on 'Connect'
4. Power off and on Tinymov. The application status should now indicate that it is connected to the device.
5. Under 'PROGRAMMING' click on 'Browse…' and find the latest 'tinymovr<firmware_version>-R<board_revision>.bin' file for the board being used, which can be downloaded from releases. Also take note of any version specific instructions in the release notes.

6. Under 'PROGRAMMING' click on 'Auto Program'. The application will erase, flash and verify the device with the new firmware. Wait for the process to complete
7. Once complete, power off Tinymovr and quit the application

You should now be able to use Tinymovr as usual with the new firmware.

We are actively working to provide a cross-platform utility to allow easy firmware file flashing using the command line.

10.2 Upgrading using J-Link

Please see *Using VSCode*.

CHAPTER

ELEVEN

GIMBAL MOTORS

11.1 Introduction

Since firmware v0.8.4 and studio v0.3.4, Tinymovr can drive gimbal-style brushless motors.

What is a gimbal motor anyway?

A gimbal motor is a 3-phase brushless motor whose stator is wound with many turns, and as such exhibits much larger resistance and inductance compared to the high-current brushless motors used to provide lift to drones, rc planes etc. Gimbal motors are used in… well… camera gimbals mainly, because they offer smooth motion and require small currents to produce torque compared to the 'regular' brushless motors. This in turn can help minimize the size of the motor driver and associated wires etc. Note that we refer to reduction of current through the stator windings and not the power converted to heat as a result of Joule heating of stator windings, which is still the same for the same amount of torque.

To achieve closed-loop current control, motor controllers such as Tinymovr use current measurement resistors in each phase (usually in series with the low-side mosfet) to estimate the current in the motor windings. These resistors have low resistance (in the range of a few milliOhms at most), to limit power dissipation in the resistor and allow operation in wide current ranges (up to several tens of Amps, even hundreds). The drawback is that current measurements exhibit noise of 100s of mA, and as such do not offer accurate measurements of small currents.

The gimbal mode of Tinymovr disables closed-loop current control. The commanded currents are converted to voltages using basic resistance and inductance formulas.

11.2 Enabling Gimbal Mode

Warning:

- Although tested, gimbal mode is still experimental. Please ensure all safety precautions, and use at your own risk!
- DO NOT perform calibration on a gimbal motor without setting gimbal mode first! There is a risk of damaging the motor and board.
- Using arbitrary resistance and inductance settings can damage your motor and board.

To enable gimbal mode, set the motor configuration as follows:

```
>>>tm1.set_motor_config(1, pole_pairs, calibration_current)

>>>tm1.set_motor_RL(motor_resistance, motor_inductance)
```

Please note that by default the motor_resistance field is in Ohms, the motor_inductance field is in Henries, and the calibration current in Amps. The pole pairs need not be the correct number, as it will be discovered during calibration. You can leave it at the default 7.

Example

```
>>>tmx.set_motor_config(1, 7, 0.5)

 >>>tmx.set_motor_RL(5, 0.002)
```

This specifies a motor with 5 Ohms resistance, 2 Millihenries inductance and 0.5Amps calibration current. Alternatuvely, using the units interface:

```
>>>tmx.set_motor_config(1, 7, 0.5 * Amps)

>>>tmx.set_motor_config(5 * Ohm, 0.002 * Henry)
```

Control that the settings are correct:

```
>>>tmx.motor_config
{"flags": 2, "pole_pairs": 7, "I_cal": 0.5}

>>>tmx.motor_RL
{"R": 5, "L": 0.002}
```

You can now calibrate the motor. Calibration will bypass resistance and inductance measurement, and will only calculate pole pairs, offset and direction. After calibration you should see the correct number of pole pairs, and a value of *3* in the *flags* field of motor_config:

```
>>>tmx.motor_config
{"flags": 3, "pole_pairs": 11, "I_cal": 0.5}
```

11.3 Controlling the Motor

Gimbal mode has identical functionality as the regular mode. Position, velocity and current control modes are supported. Note that you may have to tune the control gains to achieve optimal performance. In addition, due to the fact that current control is open loop, high angular velocities may not be available.

CHAPTER

TWELVE

CANINE ADAPTER

The documentation for the CANine adapter has [moved to a separate page](https://canine.readthedocs.io).

CHAPTER

THIRTEEN

TROUBLESHOOTING

Please take a look at Github Discussions and Github Issues , and ask any questions there.

You are more than welcome to join the Tinymovr Discord Server and ask any questions there, or have a chat with us!

CHAPTER

FOURTEEN

CONTROL PRINCIPLES

This document provides a high-level introduction to the control principles found in Tinymovr.

14.1 Permanent Magnet Synchronous Motors (PMSMs)

PMSMs is a category which includes the majority of hobby-grade brushless motors. PMSMs produce torque through the interaction of the magnetic field of the rotor permanent magnets with the magnetic field generated by the stator coils. The stator magnetic field "rotates" at a rate that is an integer multiple of the rotor rate of rotation, this is why this type of motor is termed "synchronous".

This is also why, the motor controlled needs to be able to estimate the angular position of the rotor, in order to derive the stator magnetic field that produces torque. Many controllers estimate the rotor using magnetic encoder readings, which is then converted to electrical angle in software. This is termed *sensored estimation*, and this is the method employed in Tinymovr. In contrast, ESCs such as those used in drones use voltage feedback from one of the motor phases to estimate rotor position. This is termed *sensorless estimation*.

14.2 Field Oriented Control (FOC)

FOC is an optimal control scheme used with brushless motors. It is more complex than other algorithms and requires more computing power. A 32 bit micro controller is now enough to run it at a sufficiently high rate, so it is becoming more common. FOC allows for greater performance and controlling directly the current in the phases and consequently the output torque of the motor.

The FOC algorithm comprises three main elements:

1. Precise 3-phase current estimation
2. Transformation of 3-phase measurements to rotating frame.
3. Regulation of the rotational frame variables.

14.2.1 Measuring Phase Currents

In order for FOC to work, the currents in the three phases (U, V, W) of the motor must be precisely estimated. This is done with the help of shunt resistors. Shunt resistors are high-power elements with low, precisely known resistance values, that are placed in series to the low-side mosfet sources, and enable current measurement by measuring the voltage differential across their terminals. Hardware and software filtering techniques improve current measurement accuracy.

As the motor phases are connected, current measurement in each of the three phases is redundant, and measurement in two phases would theoretically suffice. Nonetheless, having all three measurements allows for better accuracy through averaging of measurement errors. This is why Tinymovr uses three phase current measurement.

14.2.2 Reference Frames

Next, the 3-phase measurements are transformed to the rotating frame of the rotor which is termed dq. The relevant transformation is known as the dq0 transform. The resulting quantities are the direct (d) current and quadrature (q) current. Motor torque is attributed to the quadrature component, while the direct component is minimized.

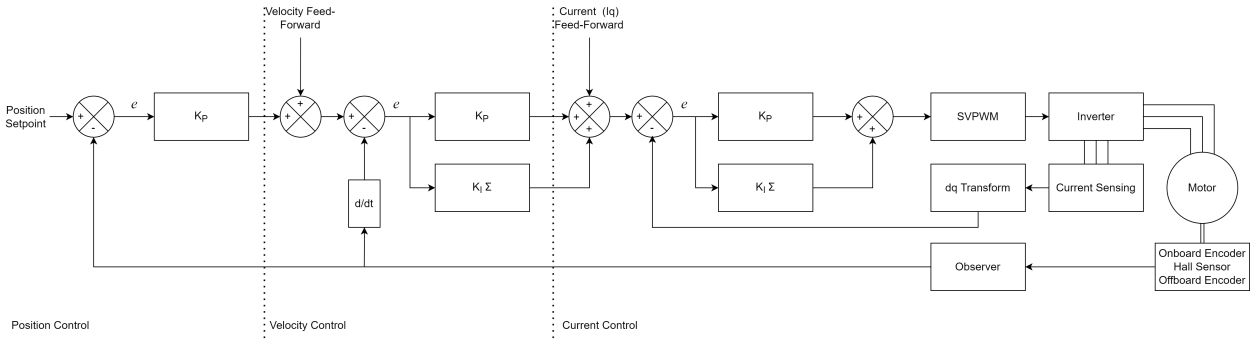
14.2.3 Current Regulation and Motor Parameter Identification

Because the d and q quantities represent DC signals in the rotational frame, it is possible to apply PI regulation to control current. For optimal regulation, the PI current regulation requires correct identification of proportional and integral gains. Tinymovr uses the method proposed in [1] to calculate the gains from the resistance and inductance motor parameters. The parameters are measured automatically by the firmware during the calibration procedure.

Thus the whole process is automated, and you don't need to worry about it.

14.3 Control loop Overview

On top of the FOC loop, Tinymovr implements an embedded control loop. This control loop runs at a 20kHz rate.



14.3.1 Position mode

This is the most versatile mode, it accepts a position setpoint, and additional velocity and current feedforward terms.

You can send these 3 setpoints as a single CAN frame when using the *set_pos_setpoint* endpoint.

You can tune separately each gain of the loop.

P_p: The gain of the position proportional term

P_v: The gain of the velocity proportional term

I_v: The gain of the velocity integral term

The integral term is especially useful for tracking positions at low velocities. You can set it to zero for greater position control bandwith.

Velocity Integrator Deadband

Since firmware version 0.8.12, a configurable integrator deadband has been added. This is useful if you experience "hunting" where the rotor oscillates around the setpoint at standstill. This phenomenon is due to interaction of integrator windup and the non-linearities of cogging torque. The integrator deadband feature is only active in position control mode and disables the integrator term update within a configurable window around the position setpoint (the "deadband").

Take a look at the *vel_integrator_params* endpoint for specifics.

Example applications

Robot joint control, CNC axis.

14.3.2 Velocity mode

In this mode the controller accepts a velocity setpoint, and an optional current feedforward term.

Example applications

Drone and aircraft propeller
: Large airframes where constant angular velocity is desired, without exceptionally high RPM (i.e. 5000rpm or less).

Industrial Automation
: Where constant RPM is required, regardless of load. Pumps, ventilators, cutters, drills, etc.

Wheel propulsion
: For brushless wheel based projects like differential drives or rovers.

14.3.3 Current mode

This is the most direct mode, where you can specify current setpoints, that are direct inputs to the FOC algorithm.

Example applications

Force based control
: Robots controlled in admittance or impedance.

E-scooter
: Where the throttle input is mapped to the current target. It translates to the acceleration of the vehicle, which feels more natural than a velocity input.

14.4 References

[1] High Performance Brushless DC Motor Control

14.5 Further Reading

Vector Control (Wikipedia)

Sensorless Field Oriented Control of Brushless PMSMs

CHAPTER

FIFTEEN

FIRMWARE DEVELOPMENT

15.1 Overview

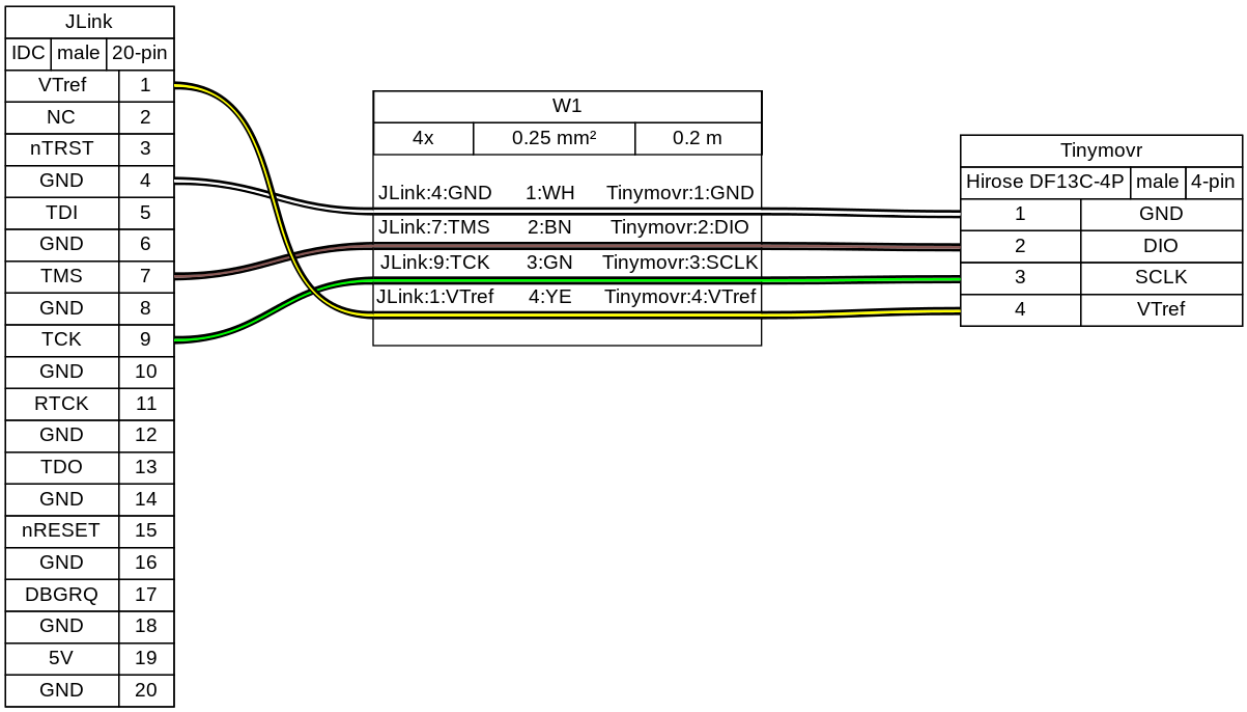
This document provides a guide for setting up a development environment for developing Tinymovr firmware. There are two alternatives, a cross-platform approach using the Arm toolchain, make and VSCode, and a Windows-only approach using Eclipse. The former is suggested.

15.2 Hardware Connections

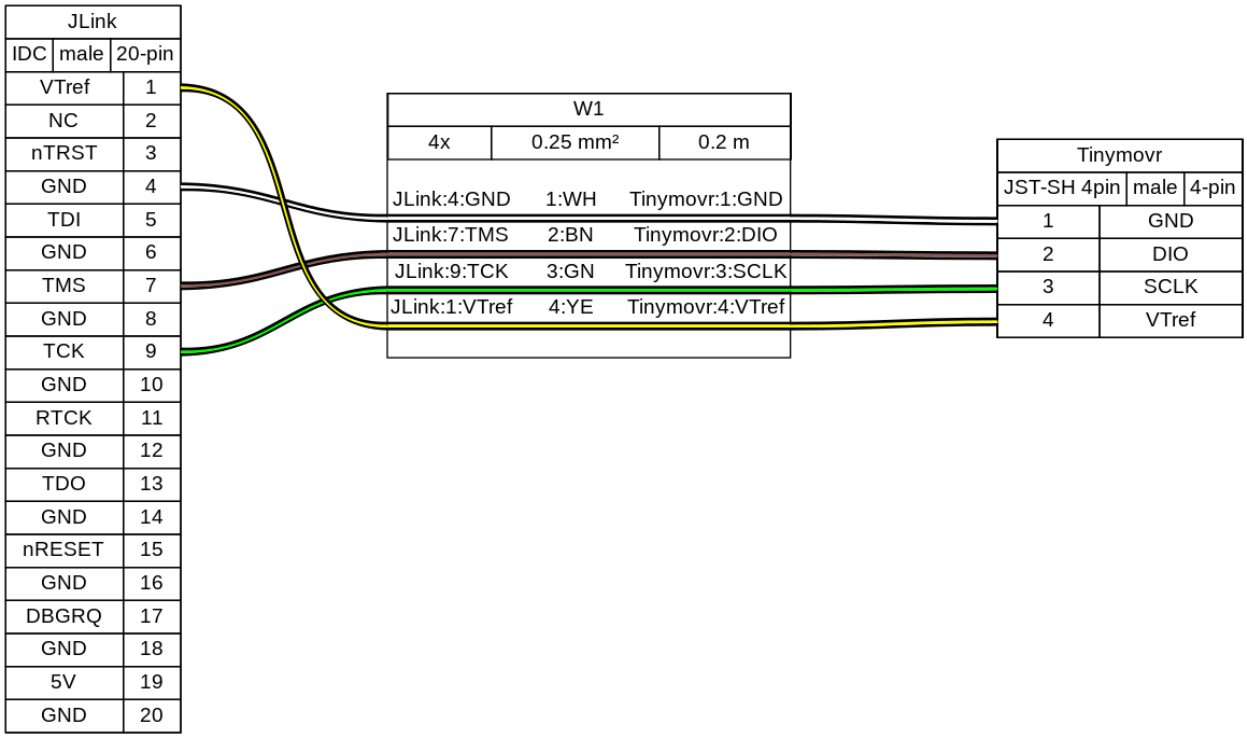
For proper debugging capabilities you will need a Segger J-Link adapter. Unfortunately original J-Link adapters are quite expensive. A more affordable option is the J-Link EDU adapter at around $60 or the J-Link EDU mini adapter at around $20. In addition, there are J-Link clones that can be purchased for very low prices on ebay or Aliexpress. However, reliability of these clones is not guaranteed.

With the board and J-Link adapter powered off, connect the J-Link to Tinymovr as shown below:

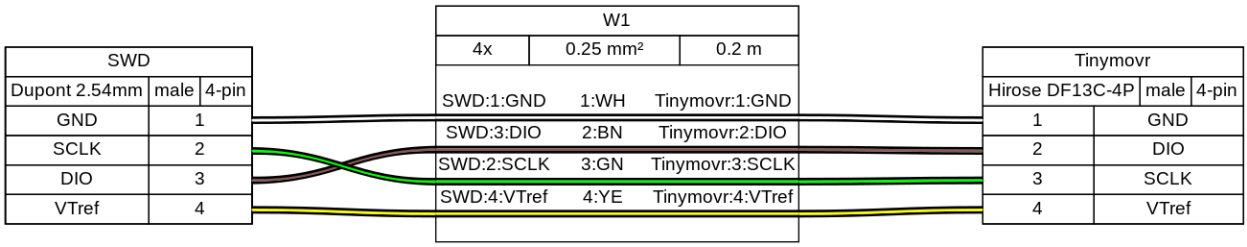
Connection directly to J-Link adapter for R3.x:



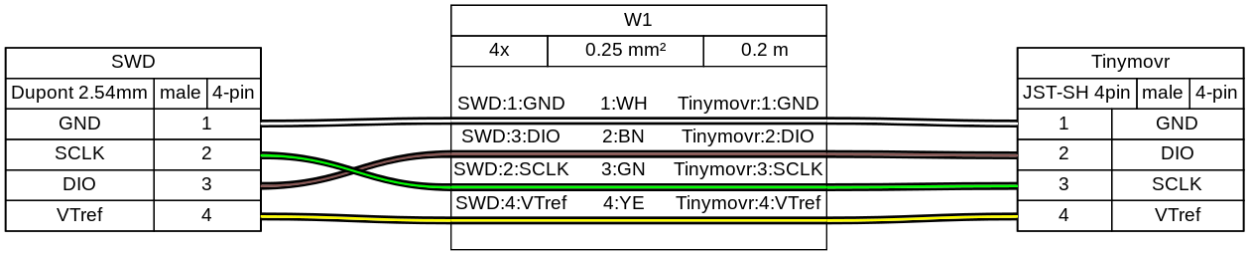
Connection directly to J-Link adapter for R5:



Connection with SWD adapter (e.g. isolator) for R3.x:



Connection with SWD adapter (e.g. isolator) for R5:



(diagrams made with Wireviz)

15.3 Setting up the repo

First, clone the Tinymovr repo:

```
git clone https://github.com/yconst/Tinymovr
```

The Tinymovr repo includes the firmware source code and supporting files, however the PAC55xx SDK is not included due to licensing restrictions imposed by Qorvo. Thus, you will need to download it from the Qorvo website, where you will need to supply your email.

The file comes in a zipped installer exe (!), which all it does is extract the contents to a directory. Navigate to the extracted files directory and copy the 'pac55xx_sdk' directory inside the Tinymovr repo:

```
cp -r <pac55xx_sdk_location> <tinymovr_location>/firmware/
```

Now you have the required PAC SDK almost ready. As a final step, there is a small patch that you will need to apply in the pac55xx_sdk directory. It is suggested to use the Python patch package, which is cross-platform. If you do not have the package, install using pip:

```
pip3 install patch
```

Then:

```
cd <tinymovr_location>/firmware
python3 -m patch sdk_patch.patch
```

Done.

15.4 Using VSCode

VSCode-based Tinymovr development is a cross-platform solution (Linux, Macos and Windows supported) for building, flashing and debugging firmware. As of November 2021, it is the official approach to Tinymovr development.

The Tinymovr repo includes all VSCode settings configured, except for the JLink *serverpath* variable in *launch.json*, which you'll need to update to reflect your system. You will also need to install the GNU Arm Embedded Toolchain, and J-Link drivers.

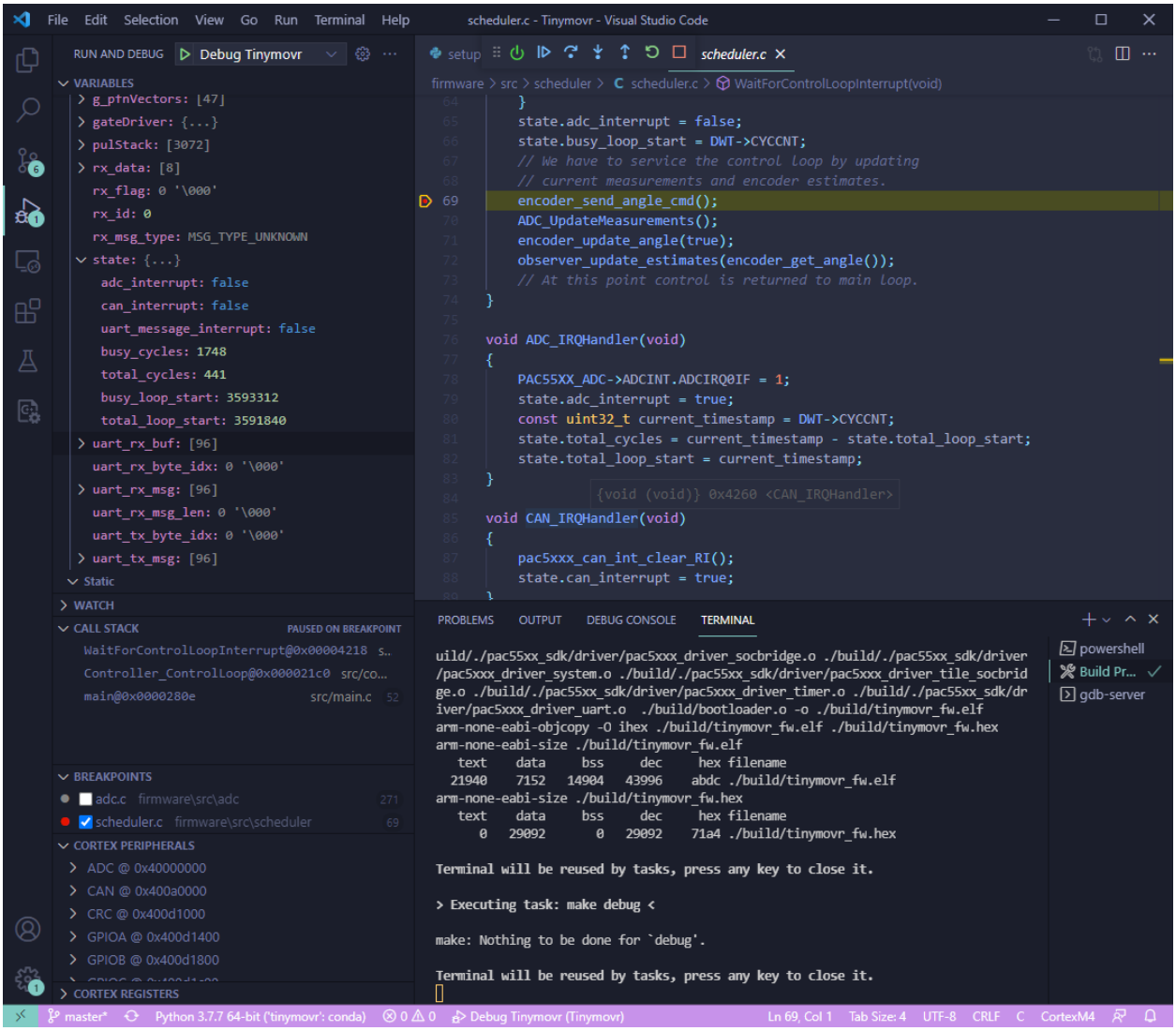
The required J-Link drivers and software, together with instructions, can be found in the Qorvo website, under the download 'Segger J-Link Support'. This download includes a nacessary patch to enable J-Link to work with Qorvo devices. Instructions on how to apply the patch are included.

In addition, if you are in Windows you will need to install GNU make. This is rather easy in Windows 10 or later:

```
choco install make
```

Once you have the J-Link adapter wired up and the software installed, you are ready to flash and debug the firmware. To try out a test build from within VSCode select Terminal -> Run Task... from the menu bar. Then select Clean and Build Project (Debug), and select the board revision against which you are compiling (R32, R33 or R5). You should end up with a `build/` directory inside `firmware/` and there you should see the files `tinymovr_fw.elf` and `tinymovr_fw.hex`.

To flash the firmware, provided your J-Link adapter is connected and drivers properly installed, hit F5. After a while you should see a screen like below:



Congrats! You are now fully set to develop!

15.5 Using Eclipse

Eclipse is the legacy method of developing Tinymovr firmware and is no longer actively supported. Consider switching to VSCode-based development instead.

You will need the Qorvo-supplied Eclipse distribution, which is Windows-compatible. Download from the Qorvo website.

Run the downloaded installer that will install Eclipse and the J-Link utility (ver. 6.31) for you.

Once installation is complete, run Eclipse and import the Tinymovr project. Try building to verify everything is ok. You are now ready to flash and debug using Eclipse.

Please note that for a successful build using Eclipse you will need to *#define* in *config.h* one of *BOARD_REV_R32*, *BOARD_REV_R33* or *BOARD_REV_R5* depending on your board revision.

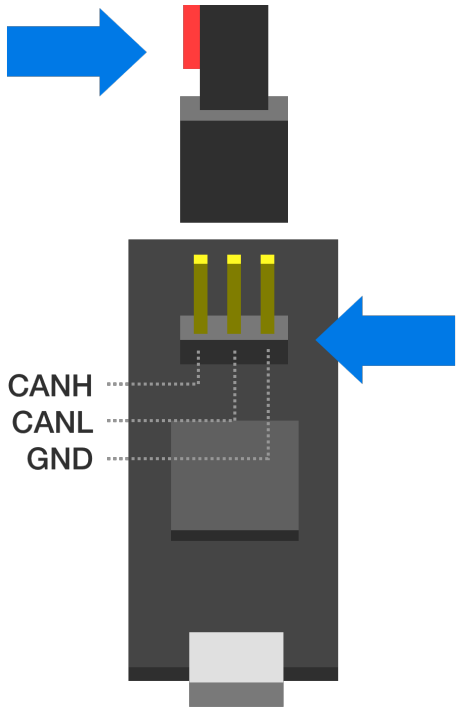
CHAPTER

SIXTEEN

HARDWARE ERRATA

16.1 Tinymovr Alpha CAN Bus Connector Erratum

The CANine v1 Adapter that comes with Tinymovr alpha has the DF-13 pins reversed and as such is not compatible with regular DF-13 cables. Alpha users are advised to use the alternative pin header on the board and the included DF-13 to 2.54mm pin converter cable to communicate with Tinymovr, as shown in the diagram below:



Note above that the red wire should stay disconnected and to the left side of the board as viewed from the USB port side.

16.2 Tinymovr Alpha USB Micro Connector Erratum

The USB Micro connector used in the CANine v1 adapter is unfortunately not very robust. In order to ensure that there is a good contact between the board and the USB cable, please ensure the male connector of the cable is firmly seated in the female connector of the board.

In addition, avoid exerting lateral forces to the connector (upwards or downwards) as they place stress on the soldered retaining flaps.

Later adapter revisions (aka CANine) use a USB Type C connector and do not have this issue.

16.3 Tinymovr R5 UART Silkscreen Reversed

The silkscreen next to the UART port on the rear of Tinymovr R5 has the order of pins reversed. The correct pins are provided in the figure below.