Spectral Micro

Description	Documentation for the Spectral Micro BLDC motor controller by Source Robotics.
Repository	https://github.com/Source-Robotics/Spectral-BLDC-docs
Copyright	© 2026 Source Robotics d.o.o. · Text and images CC BY-SA 4.0 · Code examples MIT

Table of Contents

- Spectral Micro Manual
  - Spectral Micro Manual
  - Before you start
    - Quick start
      - Quick start
      - 1. Fit the motor
      - 2. Connect
      - 3. Calibrate
      - 4. First moves
    - About this manual
  - Chapters
    - 1. Overview
      - Overview
      - The driver at a glance
      - Spectral Micro and STEPFOC
      - What you need
    - 2. Safety
    - 3. Technical data
      - Technical data
      - Ratings and limits
      - Connectors and pinout
      - Dimensions and CAD
      - Tested motors
    - 4. Mounting and wiring
      - Mounting and wiring
      - The encoder magnet
      - Mounting on the motor
      - Wiring the motor
      - Power and the CAN daisy chain
      - Thermistor
    - 5. First setup

First setup
Motor GUI
The serial terminal
Calibration
Encoder table and anti-cogging
Current limit and loop tuning
Saving the configuration
6. Operation
Operation
Control modes
Over UART
Over CAN
The watchdog
With Python
Working in Motor GUI
Errors and recovery
Status and diagnostics
Sound
Gripper mode
7. Firmware
Firmware
Releases and versions
The bootloader
Updating through the bootloader
Upgrading from older firmware
Installing the bootloader on an older board
Building from source
8. Troubleshooting
Troubleshooting
Known issues
Appendix
Firmware release notes
Versions and compatibility
Hardware revisions

- Theory corner
- Resources
  - Glossary
  - Downloads and files

API reference

- API reference
- CAN frame tool and downloads
- CAN messages
  - CAN protocol
  - Motion
  - Gains and limits
  - Status requests
  - System
  - Parameters and buffers
  - Sound
  - Gripper
- UART commands
  - UART commands
  - Motion
  - Gains and limits
  - Calibration and motor parameters
  - Status and utility
  - Gripper
- Defaults
- Errors and LED states
- Python library

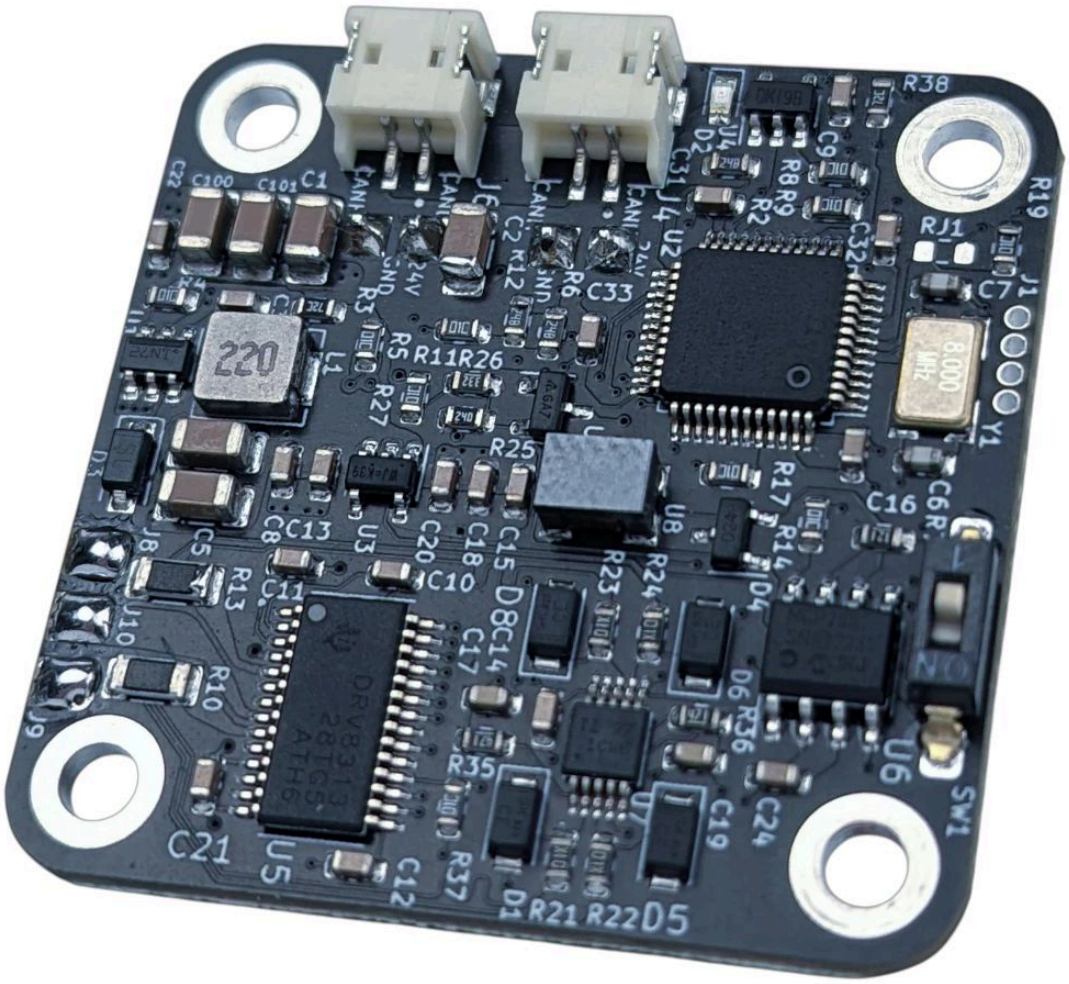
Guides

- Guides
- Python
- Mobile robot
- Quadruped leg
- Gravity compensation
- Bilateral teleoperation
- SimpleFOC

Support

- Getting help
- Report a problem
- Contributing

Spectral Micro Manual



Document number: SR-5002

Firmware version: 110

Revision: 2026.C

Published: 2026-11-13

Applies to: Spectral Micro driver PCB v2.0, firmware 110

Original instructions

Quick start | About this manual

Source Robotics d.o.o. publishes this document for the Spectral Micro BLDC driver. It is revised as the product and its firmware change; the document number, firmware version, revision and date above identify the version you are reading, and the document history on the About page says what changed. Take care that the firmware version named under *Applies to* matches what `#Info` reports on your driver: the version selector in the header switches between firmware releases. The text and the images are licensed CC BY-SA 4.0, the code examples MIT.

Quick start

Four short parts from the box to a calibrated motor that follows commands, through Motor GUI or the terminal, with a link into the manual at every step.

The manual

Safety, Technical data, Mounting and wiring, First setup, Operation, Firmware, Troubleshooting.

Motor GUI

The usual way to set up and test: the tool and working in it — connect, calibrate, tune with live plots, record telemetry, update the firmware.

API reference

Every CAN message and UART command, the defaults, the errors and the Python library, read from the firmware.

Learn

Guides with code, from a mobile robot to a quadruped leg, the bootloader, and the theory corner on FOC, the cascade and impedance control.

Before you start

Quick start

Quick start

From the box to a calibrated motor that follows commands, in four parts. The quickest way through is Motor GUI in the browser; every step also shows the terminal commands.

Read first

Reversed supply polarity, a daisy-chain cable the wrong way round, or 5 V on the UART port destroys the driver. The motor turns during calibration. The Safety chapter is two minutes long.

Before you start

You start from the box: a driver, a motor and the parts below. The wiring needs a multimeter and a soldering iron.

Nothing else is assumed: no experience with motor drivers, a terminal or a program.

The four parts

Part	What you do	What you end with
1. Fit the motor	Fit the magnet, mount the board behind the motor, solder the three phases	A driver on its motor, wired
2. Connect	Connect the supply and the PC, open Motor GUI or a terminal	A driver that talks to your PC
3. Calibrate	Restore the defaults, calibrate, set the current limit, save	A calibrated driver that keeps its settings
4. First moves	Turn the motor at a speed, move it to a position, stop it	A motor that follows commands

Each step links the page that explains it. Follow the link when a step is not clear or something fails.

What you need

	Where to get it
Spectral Micro driver	single driver or the starter kit, which has everything below except the motor, the bracket and the supply
A brushless motor with a phase resistance of 2 to 24 Ω and 2.8 A or less: a gimbal motor, from the 2804 class to the 8108 class	the tested motors are safe choices
The encoder magnet, diametrically magnetised; the stock one is 6 × 2.5 mm, another thickness is fine if it gives the 1 mm gap	magnet
A bracket that holds the board behind the motor, and M3 screws	the example bracket to adapt and print; a NEMA 17-format motor needs only a spacer
USB-to-serial adapter, 3.3 V, and its cable	serial adapter, UART cable
Power cable	power cable
A 12 to 29 V supply, 24 V recommended	a bench supply with a current limit is the safest start
For CAN later: **CANvas USB-to-CAN adapter** and a **CAN cable**	CAN adapter, CAN cable
Optional: **100 kΩ NTC thermistor** for motor temperature protection	thermistor
Software: Motor GUI in Chrome or Edge, or a serial terminal	motorgui.com; PuTTY, the PlatformIO monitor or the Arduino serial monitor

More: What you need.

1. Fit the motor

The driver goes behind the motor, over a magnet on the end of the motor shaft.

1. Fit the magnet

1. Press the magnet onto the end of the motor shaft, centred.
2. Glue it, so that it cannot turn on the shaft.

The magnet must end up about **1 mm** above the encoder chip in the middle of the board.

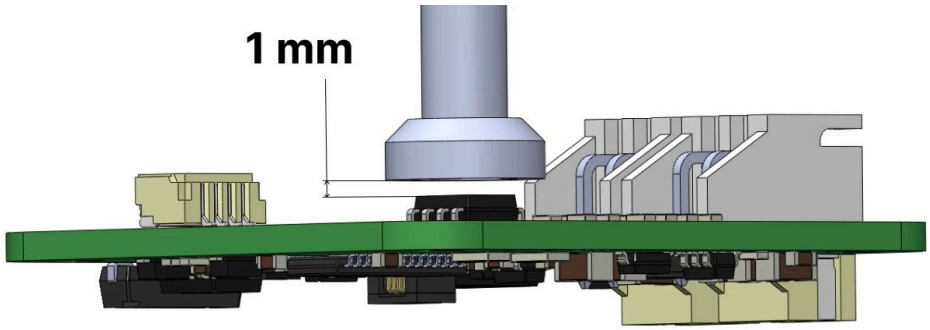


Figure 1: About 1 mm between the magnet and the encoder chip in the middle of the board.

A magnet that slips is the usual cause of a calibration that "succeeds" and a motor that then runs badly.

Check

The magnet sits centred on the shaft, and once the glue has set it does not move when you push it with a finger. If it does: The encoder magnet.

More: The encoder magnet.

2. Mount the board

1. Bolt the bracket to the bottom of the motor.

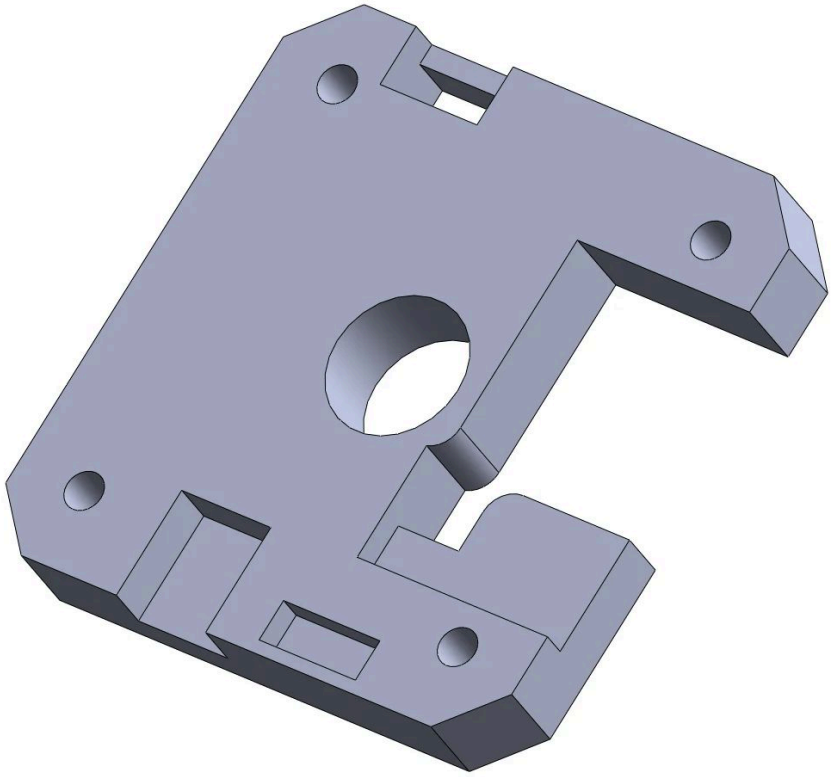


Figure 2: The example bracket. The motor bolts to one side, the board to the other; the hole in the middle is for the shaft and the magnet.

2. Bolt the board to the bracket, with the encoder chip (the bottom of the board) towards the magnet.
3. Check the gap between the magnet and the encoder chip: 1 mm, not touching.

A NEMA 17-format motor takes the board on a spacer with its own rear screws.

> ✓ **Check**
>
> The board sits parallel to the motor, with about 1 mm between the magnet and the encoder chip, and the rotor turns by hand without the magnet touching the chip. If not: Mounting on the motor.

More: Mounting on the motor.

3. Wire the motor

Solder the motor's three wires to **U, V and W**. The order does not matter: calibration detects it.

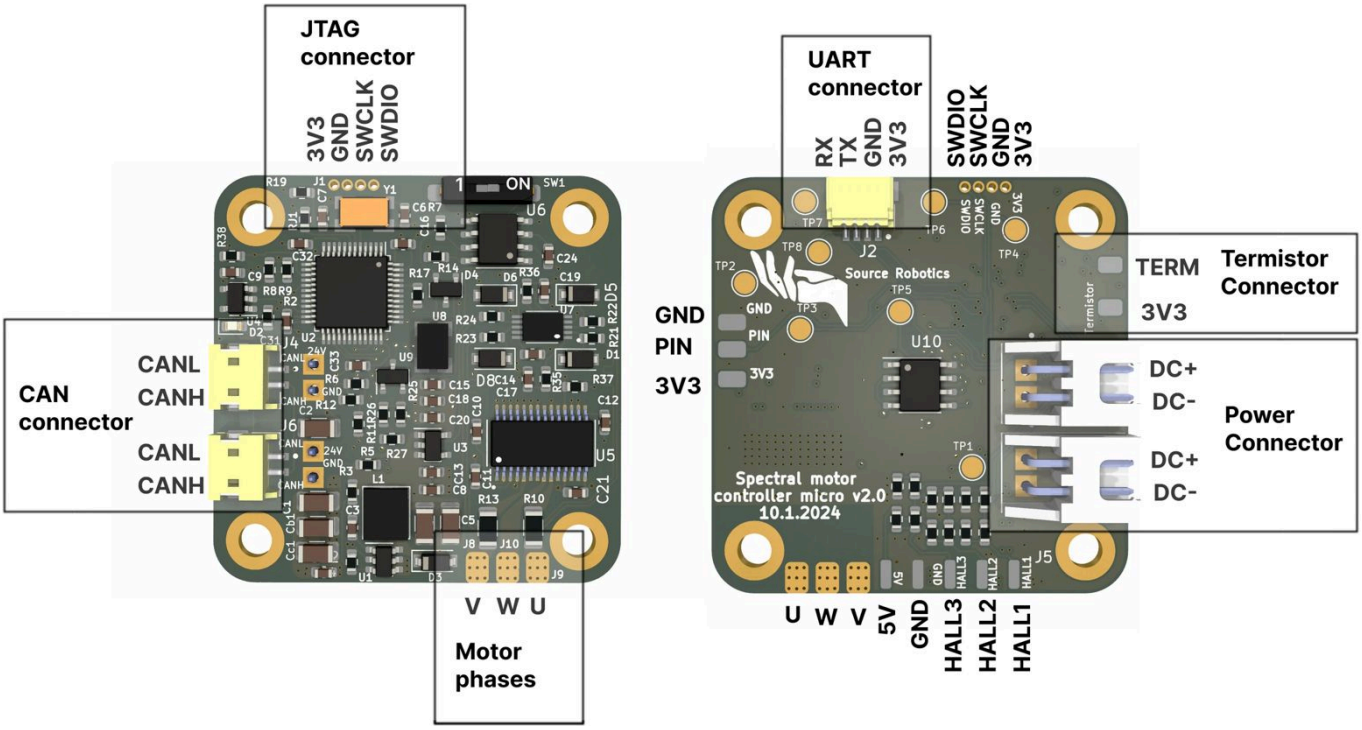


Figure 3: The connectors. The motor phases go to U, V, W; the supply to DC+ and DC−.

Check

With the driver unpowered, a multimeter shows about the same resistance between U and V, V and W, and W and U, and no strand bridges two groups. If not: Wiring the motor.

More: Wiring the motor.

2. Connect

The supply powers the motor, the serial adapter connects the PC.

1. Connect power and the PC

1. Connect the supply to a power connector: **DC+ to positive, DC− to negative**. Check twice.
2. Connect the serial adapter to the UART connector with the UART cable: TX to RX, RX to TX. 3.3 V only.

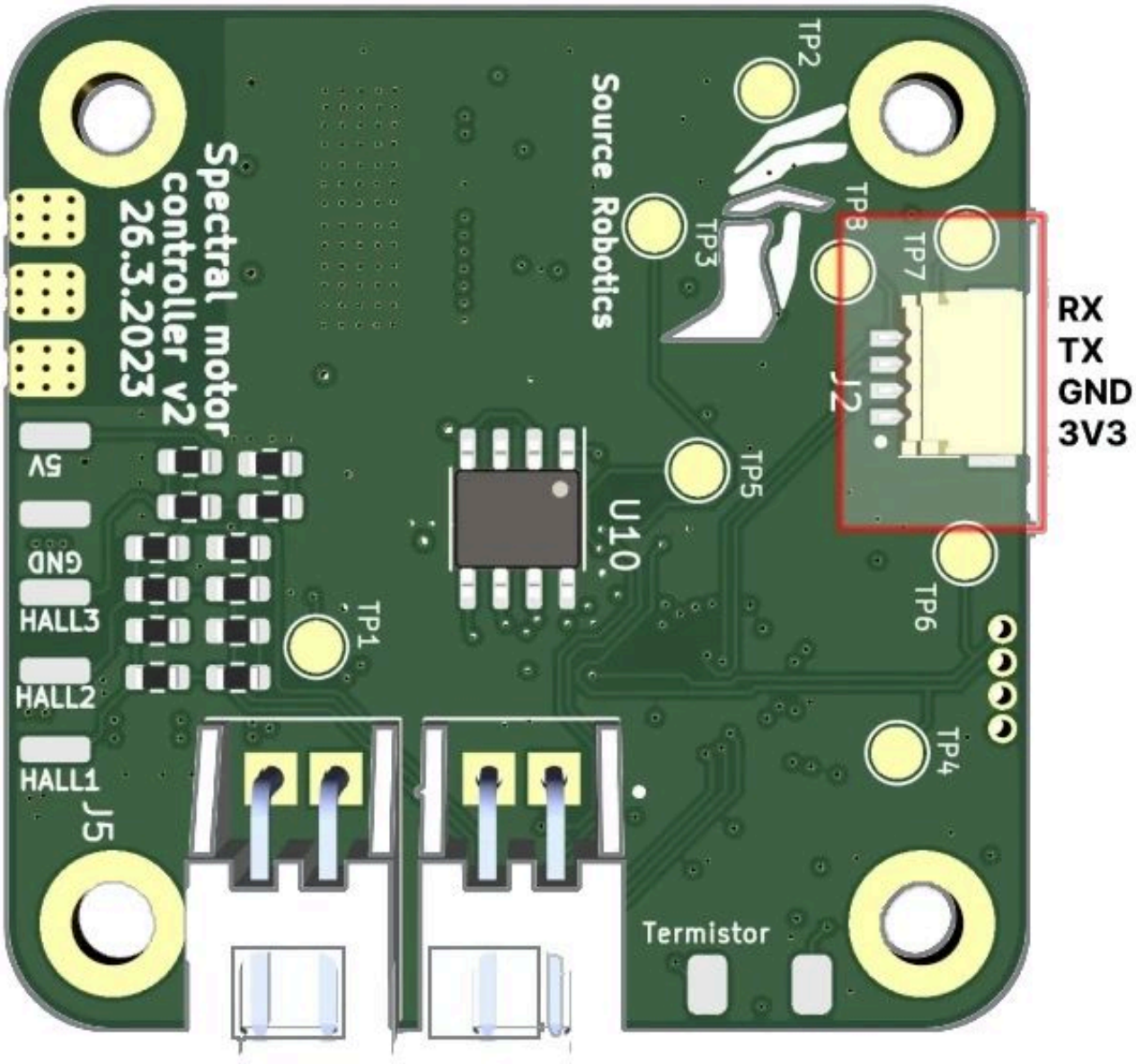


Figure 1: The UART connector. TX of the driver goes to RX of the adapter and RX to TX.

3. Power on.

The LED blinks for about two seconds while the bootloader waits for a host. Then the firmware starts, and the LED stays on steadily: the driver is not calibrated yet, which it reports as an error state.



Figure 2: On, steady.

The serial adapter alone

More: Power and the CAN daisy chain · Connectors and pinout.

2. Open Motor GUI or a terminal

Motor GUI

1. Open motorgui.com in Chrome or Edge.
2. Choose the serial connection.
3. Press **Connect**, and choose the adapter's port.

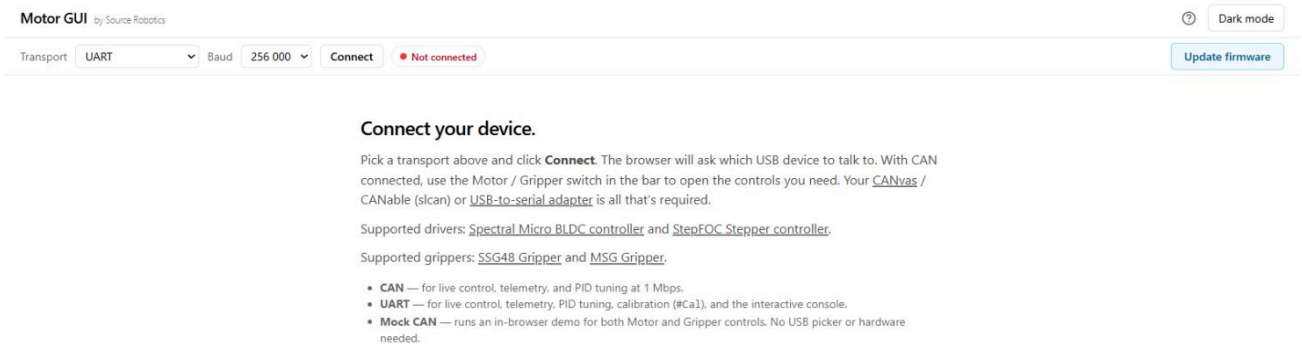


Figure 3: Motor GUI connection bar with UART transport, baud rate and Connect.

The driver's identity and state appear: hardware and firmware version, serial number, *Not calibrated*, *No errors*.

If nothing connects: the port, the cable's TX/RX, the browser (Web Serial needs a Chromium browser).

More: Motor GUI.

Terminal

1. Open a terminal on the adapter's port: 256000 baud, line ending CR+LF.
2. Type `#Info`.

The driver prints its hardware and firmware version, its CAN ID, whether it is calibrated, and the motor's parameters.

More: The serial terminal.

Check

The driver's identity and live state appear before you change settings. A missing connection is covered in Motor GUI or Troubleshooting.

3. Calibrate

The driver learns your motor. There is nothing to enter: calibration measures the motor's resistance and inductance and finds its pole pairs and phase order by itself.

1. Restore the defaults

First the factory defaults, so that calibration does not start from another motor's values or from junk in the EEPROM.

Motor GUI

Press *Default* in the settings.

Terminal

```
#Default
```

What else #Default resets

2. Calibrate

We recommend a Motor GUI preset

In Motor GUI's **Presets** tab, apply *Generic motor*, *BLDC motor on Spectral Micro (soft)* before you calibrate. It writes a low current limit and soft loops and leaves the resistance, the inductance and the pole pairs to the calibration, which measures them, so the motor calibrates safely. Raise the limits and the gains from there (Current limit and loop tuning). More: Motor GUI.

1. Fix the motor to the bench, with nothing on the shaft.
2. Set the supply to 24 V.
3. Start the calibration:

   Motor GUI

   Start the calibration. The GUI switches to *Calibration mode* and shows the driver's log step by step.

   Terminal

   ```
   #Cal
   ```

4. Keep the shaft free until the end, about a minute more after the log below.

The LED flashes, the motor clicks, then turns in both directions, and the driver reports each step: the magnet, the supply, the resistance and inductance, the pole pairs, the phase order, the motor constants, and finally

```
Calibration success!
Next, by itself: the encoder table, then anti-cogging (about a minute; keep the shaft free).
Everything is saved when they finish (#Encal state, #Cogcal state)
Going to idle!
```

The driver then goes on to measure the encoder table and the anti-cogging, and saves everything when they succeed.

A failed step names itself:

- `Magnet test: Failed` is the magnet;
- `Resistance test: Failed` means a phase is not connected or the motor is outside the 2 to 24 Ω range;
- `Open loop: Failed` means the rotor did not follow the field: check the three wires and that the shaft is free;
- `Commutation/alignment check failed!` means the motor did not respond: check the three wires, the supply, and that the magnet does not slip.

> ✓ **Check**
>
> The calibration log ends with `Calibration success!`. If it does not, use the failed step in Calibration troubleshooting; do not treat a stopped motor as success.

More: Calibration, with the whole log and every message.

3. Set the current limit and save

The current limit is the current the motor may draw, in mA; it is also the grip of the motor against a load.

From firmware 109 calibration puts back the limit that was in force (107 and older left 1600 mA), so check it now, before you save. 1000 mA suits the gimbal motors in the tested list; a small 2804 is better at 500 mA, a motor rated for more can go up to 2800 mA.

Motor GUI

Set the current limit in the limits, then **Save** (the GUI warns about unsaved changes until you do).

Terminal

```
#Ilim 1000
#Save
```

Without saving, the calibration is gone at the next power cycle. The LED now shows three short flashes and a pause: normal operation.



Figure 1: Three short flashes and a pause.

Check

After **Save**, Motor GUI no longer marks unsaved changes, and after a power cycle `#Info` prints `Calibrated`. Saving explains which settings are kept.

More: Current limit and loop tuning · Saving the configuration.

4. First moves

The motor is calibrated and saved. Now it moves on command.

1. Move it

Motor GUI

Use the jog controls:

- *Jog velocity* turns the motor at a speed;
- *Jog position* moves it to a position;
- *Jog current* applies a torque;
- *Send setpoint* sends a value in the current control mode.

The plots show position, speed and current while it moves.

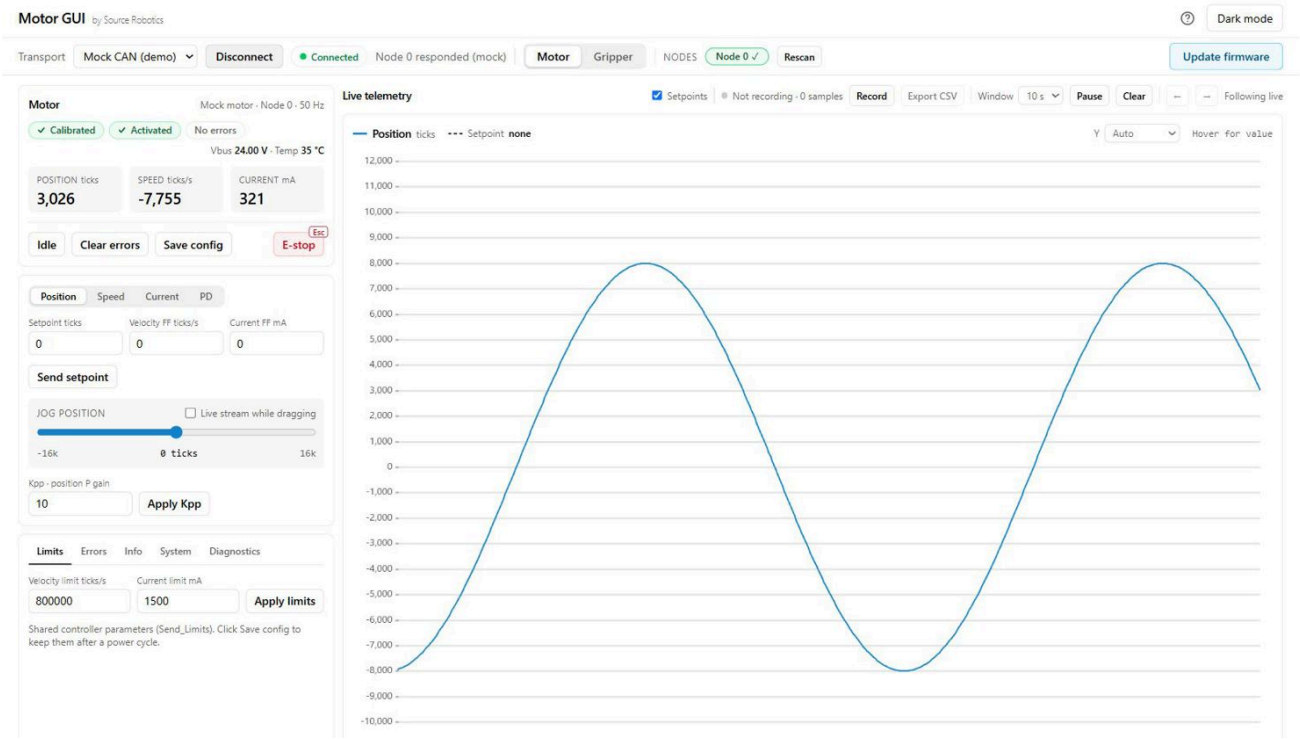


Figure 1: Motor GUI simulated motor: device state, controls and live telemetry.

Terminal

```
#V 16384        one revolution per second
#V 0
#P 0            back to where it started
#P 16384        one revolution forward
#Idle
```

Position is in encoder ticks, 16384 per revolution; speed in ticks per second; current in mA. If the motor hums or whines, lower the current limit or tune the loops.

Check

The position and speed feedback follow the request, and **Idle** stops driving the motor. A motor that does not follow needs Troubleshooting before a load is attached.

More: Control modes · Current limit and loop tuning.

2. Where to go next

You want to	Read
get to know Motor GUI: the screen, the plots, recordings, several motors	Motor GUI, Working in Motor GUI
control the driver from a program over CAN	Over CAN, then With Python
put several drivers on one bus	give each a node ID and terminate the ends: Power and the CAN daisy chain
protect the motor from overheating	Thermistor
update the firmware	Firmware
look up a message or a command	API reference
build something: a wheeled robot, a leg, a compliant arm	Guides
understand why it does not work	Errors and recovery, Troubleshooting

About this manual

Scope and audience

The Spectral Micro Manual covers the Spectral Micro BLDC driver from unpacking to a tuned motor on a CAN bus: what the driver is, how to mount it on a motor and wire it, the first setup with Motor GUI or the serial terminal, calibration and tuning, how the driver is controlled over UART and CAN or from Python, and how its firmware is updated. The API reference lists every CAN message and UART command of the firmware release named on the landing page. The Guides are worked examples.

It is written for someone who builds a robot or a machine with the Spectral Micro: a maker, a student or an engineer who can solder three motor wires and open a serial terminal. It does not assume any knowledge of field-oriented control; the Theory corner in the appendix explains the concepts the control modes rely on.

Read the Safety chapter before you power anything. The Quick start is the whole first setup, in four short parts; the chapters are in the order you need them and go deeper.

Finding your way

The header has four tabs. **Spectral Micro Manual** is the document you read from the start: the quick start, then safety, technical data, mounting, the first setup, operation and firmware updates, in numbered chapters. Motor GUI, the tool in the browser, is the usual way to do what the chapters describe; each procedure shows it first and the terminal command beside it. **API reference** lists every CAN message, every UART command and the Python library's calls for the firmware release this site describes, with the defaults and the error flags, and is where you look things up. **Guides** are worked examples with code. **Support** says where to get help and how to report a problem.

The whole site describes one firmware release. The dropdown next to the site name switches between releases, and a page from an older release says so in a bar at the top. The repository card on the right of the header opens the firmware repository; every page ends with a link to edit it and one to report a problem with it.

Warnings and notes

Three kinds of highlighted text appear in this documentation. Read every Danger and Notice before you act on the step it belongs to.

Note

Points out something that is easy to miss or that changes what follows: a default, a limit, a side effect.

Notice

The robot, the tool, or data on the control box can be damaged or lost if this is ignored. Nobody is at risk.

Danger

Someone can be injured if this is ignored. Stop and read the whole box before acting on the step it belongs to.

Related documents

Document	Number	Covers
Spectral Micro Manual	SR-5002	the Spectral Micro BLDC driver: quick start, safety, technical data, mounting, setup with Motor GUI or a terminal, operation, firmware; the API reference; guides

A document marked with a firmware version describes that firmware release, here 110; the version selector in the header switches between releases. The compatibility table says which firmware, Python library and hardware revisions work together.

The STEPFOC stepper driver has similar hardware and firmware and is run the same way; its documentation is at docs.source-robotics.com/STEPFOC-docs. The SSG-48 gripper, which is driven by a Spectral Micro, is documented at docs.source-robotics.com/SSG48-gripper-docs.

Document history

The Spectral Micro Manual describes one firmware release; one revision is kept per release, and the version selector in the header switches between them. The earlier version of this site, for firmware 101, is kept as version `101`.

The text and the images of this documentation are licensed under CC BY-SA 4.0, the code examples under the MIT licence. The Source Robotics name, the product names and the logos are not covered by either.

Revision	Date	Changes
2026.C	2026-11-13	Firmware 110: the gripper (each end read under a light push, 0 and 255 clear of the stops, the soft start, a clean arrival and hold, an object after 50 ms, its own current limit), `#Estop` over UART, `#Close` during a calibration, brake or coast kept by Save, the watchdog while idle, parameters 90 to 93; the upgrade from 109

Revision	Date	Changes
2026.B	2026-11-10	Firmware 109: the parameters and buffers (CAN 36 to 43), the identity and telemetry frames, the watchdog actions, the encoder table and the anti-cogging, the current loop from R, L and a bandwidth, the speed estimate, the commutation delay compensation, the EEPROM blocks, status and diagnostics, sound through the motor (CAN 44 and 45, `#Tone`), upgrading from older firmware; new pages Encoder table and anti-cogging, The watchdog, Status and diagnostics, Sound, Upgrading from older firmware, Parameters and buffers, Sound (CAN)
2026.A	2026-11-03	First version in this form: the manual chapters, the generated CAN and UART reference, firmware 107

Chapters

1. Overview

Overview

What the Spectral Micro is and what it is made of, how it relates to the STEPFOC stepper driver that is built on it, and what you need around it to get a motor turning.

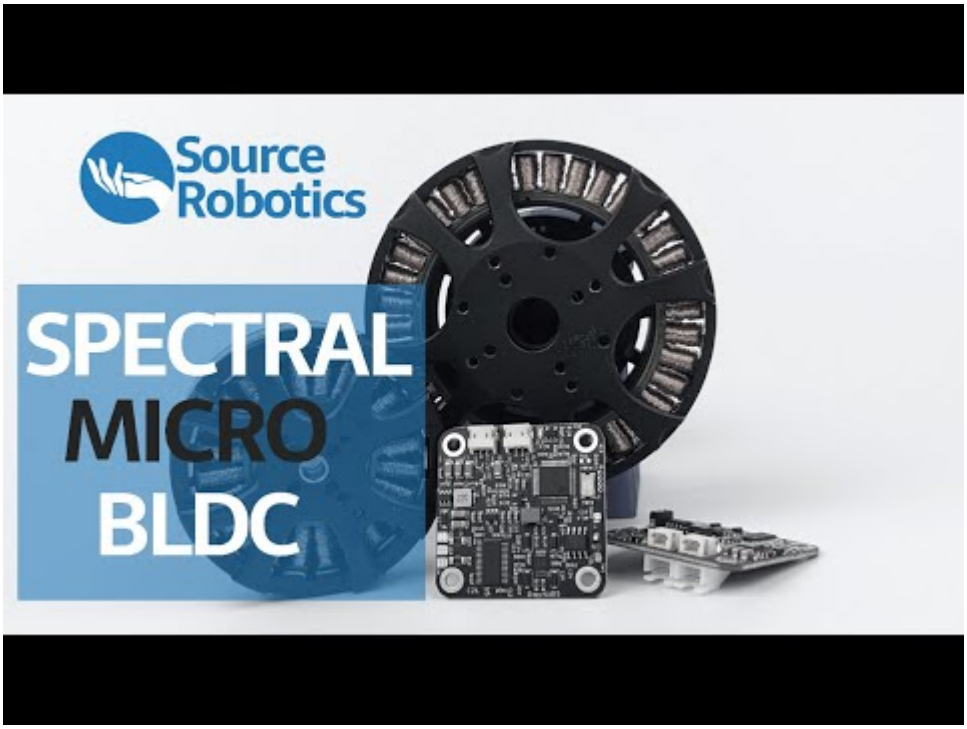


Figure 1.1: The introduction video: the driver, the motors it runs and what it does with them.

In this chapter

- The driver at a glance
- Spectral Micro and STEPFOC
- What you need

The driver at a glance

The Spectral Micro is a field-oriented controller for small three-phase brushless motors: gimbal motors and other low-current motors with a phase resistance of a few ohms. It mounts behind the motor, reads the rotor position from a magnet on the motor shaft, and drives the three phases with sinusoidal currents placed from that position, the way a servo drive does.

The motor can be commanded in position, speed, current (torque) or impedance, holds a position at standstill, reports where it is, and runs quietly and efficiently. It is made for robot joints, legs, arms and grippers.

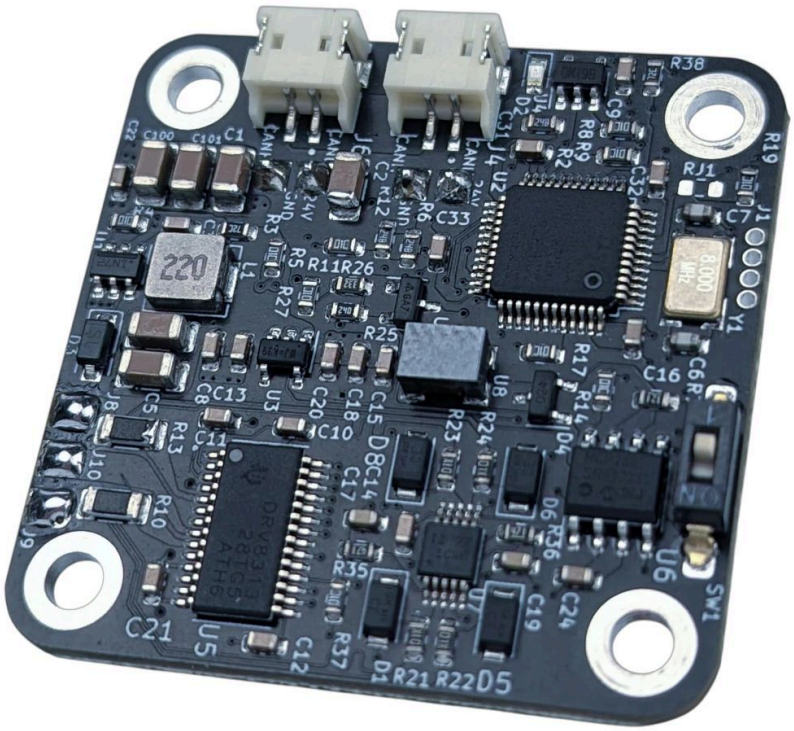


Figure 1.2: The Spectral Micro driver, 39 × 39 mm, with the encoder chip in the middle of the board.

What is on the board

Part	What it does
STM32F103CB microcontroller	runs the control loops at 5 kHz and the CAN and UART interfaces
14-bit magnetic encoder (MT6816)	reads the position of the magnet on the motor shaft: 16384 ticks per revolution
Three-phase driver stage with inline current sensing	drives the phases with up to 2.8 A and measures the phase currents
16 Kbit EEPROM	keeps the configuration and the calibration over a power cycle
Two CAN connectors and two power connectors	daisy-chain several drivers on one bus and one supply
CAN termination switch	the 120 Ω resistor the first and the last node need
UART connector (3.3 V)	the serial terminal: every setting, calibration, tuning
Thermistor pads	an NTC in the motor windings for temperature protection
PIN header	a spare pin for your own use: an analog input or a digital input/output
HALL header	pads for hall sensors; not implemented in the firmware
JTAG pads (1.27 mm)	the programming adapter, for installing the bootloader or for development
Status LED	normal operation, calibration or error, see below

The connectors and their pinout are in Connectors and pinout.

What the firmware does

The firmware closes three nested loops on the motor: a current (torque) loop, a velocity loop around it and a position loop around that, plus an impedance (PD) loop that makes the motor behave like a spring and damper.

A command picks the loop: a current setpoint, a speed, a position or a PD setpoint. An open-loop mode and a voltage-torque mode exist beside them; Control modes describes each.

Calibration measures the motor: its phase resistance and inductance, its pole pairs and the order of its phases, the angle between the encoder and the rotor field, and its torque and velocity constants. Nothing about the motor has to be typed in; see Calibration.

The driver protects itself and the motor: it stops on a supply voltage outside the limits, on an encoder fault, on an over-speed, on a motor over-temperature (with the thermistor), on a hardware fault of the driver stage, on an E-stop command and on a watchdog that runs out. The status LED and `#Error` tell which; see Errors and recovery.

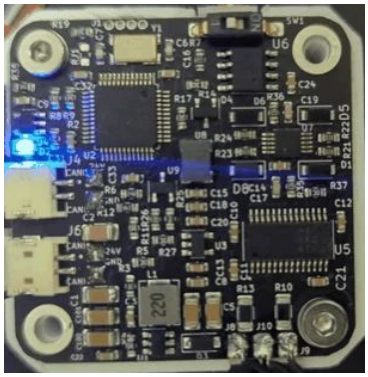
How it is controlled

- **UART**, a 3.3 V serial port at 256000 baud: human-readable commands such as `#P 16384`, and the only way to run the calibration. See Over UART.
- **CAN**, 1 Mbit/s, up to 14 drivers on one bus: compact frames for real-time control from a PC with the CANvas adapter, a microcontroller or a robot control box. See Over CAN.
- **Python**, the `Spectral-BLDC` library on top of CAN. See With Python.
- **Motor GUI**, in the browser, over serial or CAN: the usual tool for setup, testing and tuning. See Motor GUI.

The status LED

Pattern	Meaning
three short flashes and a pause	normal operation: no error, calibrated
fast flashing, about 10 times a second	the calibration routine is running
on, not flashing	error mode: at least one error is active, or the motor is not calibrated

The LED can be turned off with `#LED 0`.

Normal operation	Calibration	Error mode
		

Spectral Micro and STEPFOC

The Spectral Micro and the STEPFOC stepper driver are sister drivers: similar hardware and similar firmware, each made for its kind of motor. The Spectral Micro drives three-phase brushless motors; STEPFOC drives two-phase stepper motors.

They are not the same board and do not run the same firmware, but they are controlled and run practically the same way: the same CAN and UART commands, the same Python library, Motor GUI and bootloader.

What this means in practice:

- **The commands are practically the same.** The CAN messages and UART commands of the API reference exist on both drivers, with the same IDs, names and layouts; STEPFOC adds a few that only make sense on a stepper (homing against a sensor, open-loop stepping). A program written for one runs on the other. The idle behaviour is the exception over UART: the Spectral Micro sets it with `#Brake 0` or `#Brake 1` and coasts by default, STEPFOC has the actions `#Brake` and `#Coast`. Send_Settings over CAN is the same on both.
- **The firmware releases go in step, and from 109 share their numbers.** From release 109 the two drivers are released together under the same version number; the Spectral Micro went from 107 straight to 109, so there is no Spectral Micro 108. Before, the numbers did not always match (Spectral Micro 107 went with STEPFOC 108); the compatibility table lists the pairs. Some defaults differ, each measured on its own driver: the direct PLL speed estimate at 100 Hz and 234 µs delay compensation here, 60 Hz and 166 µs on STEPFOC.
- **Calibration differs.** The Spectral Micro measures the motor's resistance and inductance and finds its pole pairs; STEPFOC cannot, so there the three values are entered from the motor's datasheet first. See Calibration.
- **The motor terminals differ.** U, V and W for the three phases, instead of A+, A−, B+, B− for the two coils of a stepper.
- **The grippers differ.** The SSG-48 gripper is driven by a Spectral Micro, the MSG gripper by STEPFOC; both use the driver's gripper mode.

The guides under the Guides tab were written with the Spectral Micro; the code in them runs unchanged on STEPFOC.

What you need

This page lists what a Spectral Micro setup is made of, what comes with the driver, and what you buy or make yourself.

The driver

The Spectral Micro is sold as a single driver or as a starter kit in the Source Robotics shop:

- Spectral Micro BLDC controller
- Spectral Micro starter kit, which adds the adapters, the magnet, the cables and a thermistor listed below

Every driver is end-of-line tested and ships with the current firmware release and the bootloader already installed, so a driver works out of the box with a serial terminal, the Python library or Motor GUI. The firmware can be updated later without a programming adapter; see Firmware.

A brushless motor

The Spectral Micro drives three-phase brushless motors with a phase resistance of about 2 to 24 Ω and a current of 2.8 A or less: gimbal motors from the small 2804 class up to the 8108 class, and other brushless motors of similar resistance, such as the NEMA 17-format 42BLS40. The tested motors are known to work. The motor needs a place for the encoder magnet on the axis of its shaft, on the side where the board is mounted.

Low-resistance motors

The calibration routine's default voltages are made for motors of 2 Ω and more. A motor with a much lower resistance, such as a drone motor, draws a destructive current at those voltages. Do not calibrate one without lowering the calibration voltages (`#Resv` , `#Openv` , `#Vp`) first. The current limit does not protect it: calibration sets its own, up to 1600 mA.

The encoder magnet

A diametrically magnetised 6 × 2.5 mm neodymium magnet on the end of the motor shaft, about 1 mm above the encoder chip in the middle of the board: 6 × 2.5 mm diametrical magnet. An axially magnetised magnet, or one that wobbles, makes calibration fail or report bad positions. The 6 × 2.5 mm size is the stock one, not a requirement: another thickness is a way to set the 1 mm gap when a shaft or a bracket leaves the wrong distance. The encoder magnet says how to fit it.

A mounting bracket

The board does not bolt to a gimbal motor directly: a bracket holds it behind the motor, with the encoder chip on the shaft's axis and 1 mm from the magnet. The firmware repository has an example bracket to adapt to your motor; see Mounting on the motor. A motor with the NEMA 17 hole pattern, such as the 42BLS40, takes the board on a spacer with its own screws.

Adapters and cables

Part	Needed for	Shop
USB-to-serial adapter (3.3 V)	the serial terminal: calibration, tuning, any `#` command	Serial adapter
CANvas USB-to-CAN adapter	CAN control from a PC, the Python library, Motor GUI over CAN	CAN adapter
CAN cable and power cable	daisy-chaining several drivers	CAN cable, power cable
100 kΩ NTC thermistor	motor temperature protection (optional but recommended)	Thermistor
JTAG programming adapter	only for installing the bootloader on an older board, or for firmware development	Programming adapter

Cable orientation

The daisy-chain cables for CAN and power are keyed by their orientation, not by their connector. A cable plugged in the wrong way round destroys the driver. Use the cables from the shop or follow Power and the CAN daisy chain exactly when you make your own.

A power supply

12 to 29 V DC, 24 V recommended; see Ratings and limits. Calibrate at 24 V: the resistance, inductance and motor constants come out wrong at a lower voltage. A bench supply with a current limit is the safest way to start.

Software

- Motor GUI, in the browser, over the serial adapter or the CAN adapter: the usual tool for setup, calibration, tuning, plots and firmware updates; see Motor GUI.

- Or a serial terminal: PuTTY, the PlatformIO monitor in VS Code, or the Arduino serial monitor; see The serial terminal.
- The Python library `Spectral-BLDC` for CAN control from your own programs; see With Python.

Safety

The Spectral Micro is a bare circuit board that drives a motor from a supply of up to 29 V. Read this chapter before you connect anything; the few rules here are the ones that destroy a driver, a motor or a finger when they are ignored.

Electrical

Supply polarity

Reversing the polarity on DC+ and DC− destroys the driver at once. Check the polarity of every power cable before you plug it in, every time.

Cable orientation on the daisy chain

The CAN and power cables that link one driver to the next are keyed by their orientation, not by their connector: a cable plugged in the wrong way round puts the supply on the wrong pins and destroys the driver. Use the cables from the Source Robotics shop, or make yours exactly as shown in Power and the CAN daisy chain.

UART is 3.3 V

The UART port runs at 3.3 V. A 5 V serial adapter destroys the board. Use a 3.3 V adapter, such as the one from the shop.

Supply voltage

12 to 29 V DC. The driver raises a Vbus error below 9 V and above `#Vmax`, 29 V after `#Default`. 29 V is also the hardware limit: do not connect a supply that can rise above it, and allow for the voltage a decelerating motor pushes back into the supply. A bench supply with a current limit is the safest way to start.

Do not disconnect a running motor

Never unplug a motor, or a power cable in the daisy chain, while the driver is powered. The inductive kick of the windings and the arcing at the connector damage the driver stage.

Static electricity

Static electricity damages the microcontroller, the encoder chip and the driver stage. Unplug the supply and touch earthed metal before you handle the board, and hold it by its edges, away from the chips and the pads.

Mechanical

Moving parts

A calibrating or running motor turns without warning, in both directions, and a motor in position mode pushes back hard against a hand. Keep fingers, hair and cables away from the shaft and whatever is mounted on it, fix the motor to the bench before the first test, and test without a load on the shaft.

A powered motor can move with nobody at it

The driver moves its motor whenever a command arrives, over UART or from any host on the CAN bus, and keeps following the last one. A speed command goes on turning the motor after the program that sent it has crashed or its cable has come out, unless the watchdog is set (Stopping the motor). Before you reach for the motor or the mechanism it drives, put the driver in idle (`#Idle`, Send_Idle) or switch the supply off.

The encoder magnet

The magnet sits about 1 mm above the encoder chip. A magnet that touches the chip, or sits far too close, can damage the microcontroller; a magnet too far away gives a wrong position. Mount it as described in The encoder magnet.

Hot motor and driver

A motor running at its rated current gets hot enough to burn, and so do the driver's chips and the board around them. Switch off and let both cool for a few minutes before you touch them. Set the current limit for your motor (Current limit and loop tuning), use the thermistor, and give the driver airflow when it runs near 2.8 A. `#Ilim` is clamped to 2800 mA, and from firmware 109 so are the current limit of Send_Limits over CAN and the current of a gripper move. The motor's rating is usually lower: a host must keep them within it.

Stopping the motor

- `Send_ESTOP` over CAN, or `#Idle` over UART, stops the motor at once; the E-stop error stays until it is cleared.
- Cutting the supply stops the motor as well; the motor then holds nothing, so a vertical axis drops.
- The watchdog (Send_Watchdog) stops the motor by itself when the host stops talking. From firmware 109 its action can instead hold the position or let a joint down at a limited speed, for a set time at most, then brake or coast (The watchdog): while that runs the motor is still driven.

Nothing on the driver is a certified safety function. A machine that can hurt someone needs an emergency stop that cuts the motor supply, independent of the driver.

Firmware updates and programming

When you connect a programming adapter to the JTAG connector, disconnect the motor supply and the UART adapter first. Updating over the bootloader needs no adapter; see Firmware.

Liability

The Spectral Micro is sold as a component for people who build their own machines. Its hardware designs, firmware and this documentation are under development and may contain errors. Whoever builds a machine with it is responsible for its safe design, assembly and operation.

3. Technical data

Technical data

The numbers: ratings and limits, the connectors with their pinout, the dimensions, and the motors that have been tested with the driver.

In this chapter

- Ratings and limits
- Connectors and pinout
- Dimensions and CAD
- Tested motors

Ratings and limits

Electrical

	Value
Supply voltage	12 to 29 V DC, 24 V typical; 10 V absolute minimum
Vbus error limits (firmware)	below 9 V or above `#Vmax`, 29 V after `#Default`
Phase current, continuous	1.5 A typical
Phase current, maximum	2.8 A (`#Ilim` and, from firmware 109, Send_Limits over CAN are clamped to 2800 mA); needs good cooling and short bursts
Power, maximum	80 W
Power, idle	0.3 W
PWM frequency	25 kHz
Control loop frequency	5 kHz
Electrical frequency, maximum	500 Hz
Current sensing	inline, on two phases
Encoder	14-bit magnetic, 16384 ticks per revolution, diametrical magnet 1 mm above the chip
UART	3.3 V, 256000 baud by default (1 000 000 tested)
CAN	CAN 2.0, 1 Mbit/s, 11-bit identifiers, node IDs 0 to 13 (14 and 15 are reserved by the bootloader), 120 Ω termination by switch
Thermistor input	100 kΩ NTC, 54.9 kΩ pull-down
PIN	analog input or general-purpose input/output, 3.3 V

	Value
Driver stage protection	hardware overcurrent (trips at 3 to 5 A), undervoltage lockout (6.3 to 8 V) and thermal shutdown (150 to 180 °C); shown as a driver error

⚠ **Reversing the polarity on DC+ and DC− destroys the board.**

Thermal and environmental

	Value
Operating temperature of the board	−20 to 130 °C
Motor temperature limit (firmware, with the thermistor)	75 °C by default (`#Tlim`), −20 °C minimum

Mechanical

	Value
Dimensions	39 × 39 mm
Mass	8 g
Mounting	four holes on the NEMA 17 pattern, 31 mm apart; the encoder chip in the middle of the board

Firmware limits

	Value
Position	signed 24-bit over CAN (±8.4 million ticks, ±512 revolutions), 32-bit internally
Speed	signed 24-bit over CAN, encoder ticks per second
Current setpoint	signed 16-bit over CAN, mA

	Value
Motors the calibration defaults are made for	a phase resistance of 2 to 24 Ω
Configuration	kept in the EEPROM with `#Save` / `Send_Save_config`

The defaults of every setting are listed in the Defaults reference.

Connectors and pinout

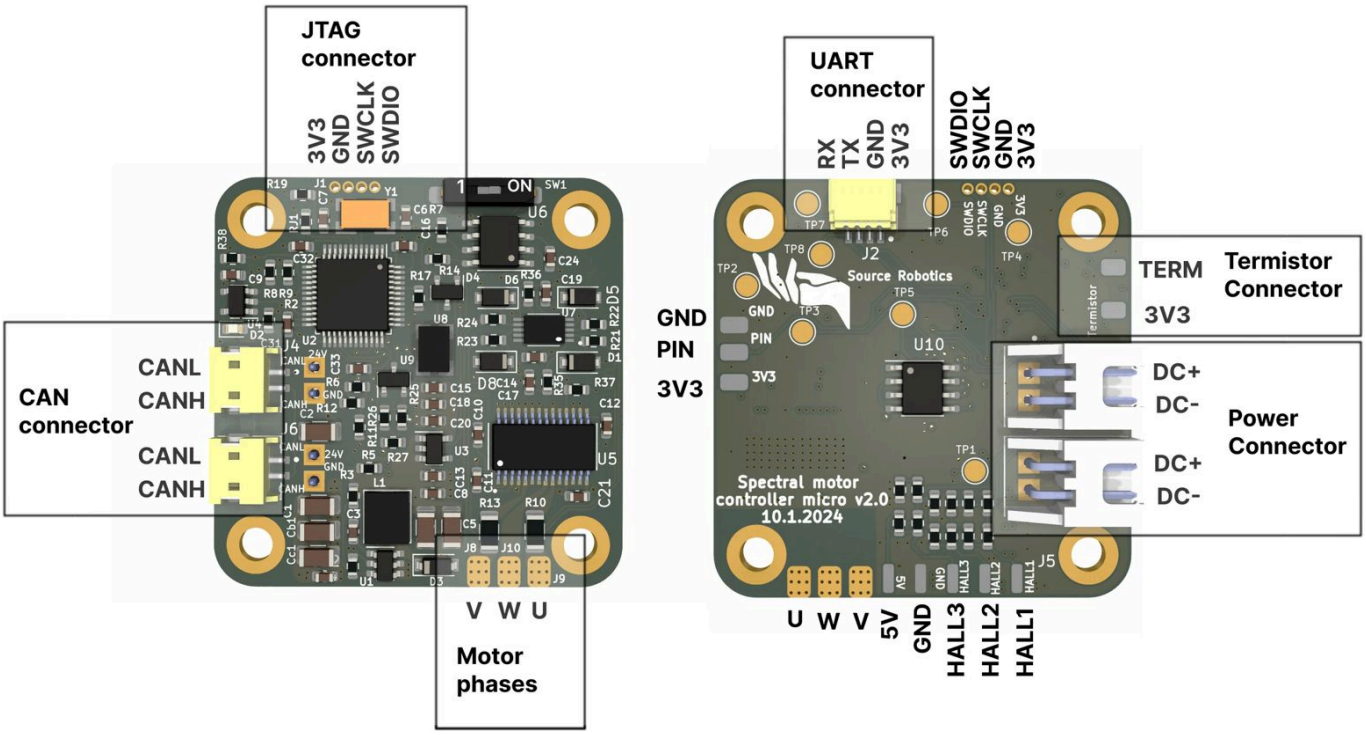


Figure 3.1: The connectors. The motor phases go to U, V, W; the supply to DC+ and DC−.

Connector	Pins	Type	Cable
Motor	U, V, W	solder pads, three per phase	the motor's own leads, see Wiring the motor
Power, two of them	DC+, DC−	JST XH, 2 pins	Power cable; the second connector feeds the next driver
CAN, two of them	CANL, CANH	JST ZH, 2 pins	CAN cable; the second connector feeds the next driver
UART	3V3, GND, TX, RX	JST SH, 4 pins	UART cable
Thermistor	TERM, 3V3	solder pads	the 100 kΩ NTC from the shop or any 100 kΩ NTC, see Thermistor

Connector	Pins	Type	Cable
JTAG	3V3, GND, SWCLK, SWDIO	1.27 mm pads	Programming adapter, pressed onto the pads
PIN	GND, PIN, 3V3	solder pads	your own signal: an analog input or a digital input/output, 3.3 V
HALL	5V, GND, HALL1, HALL2, HALL3	solder pads	hall sensors: not implemented in the firmware, see the note below

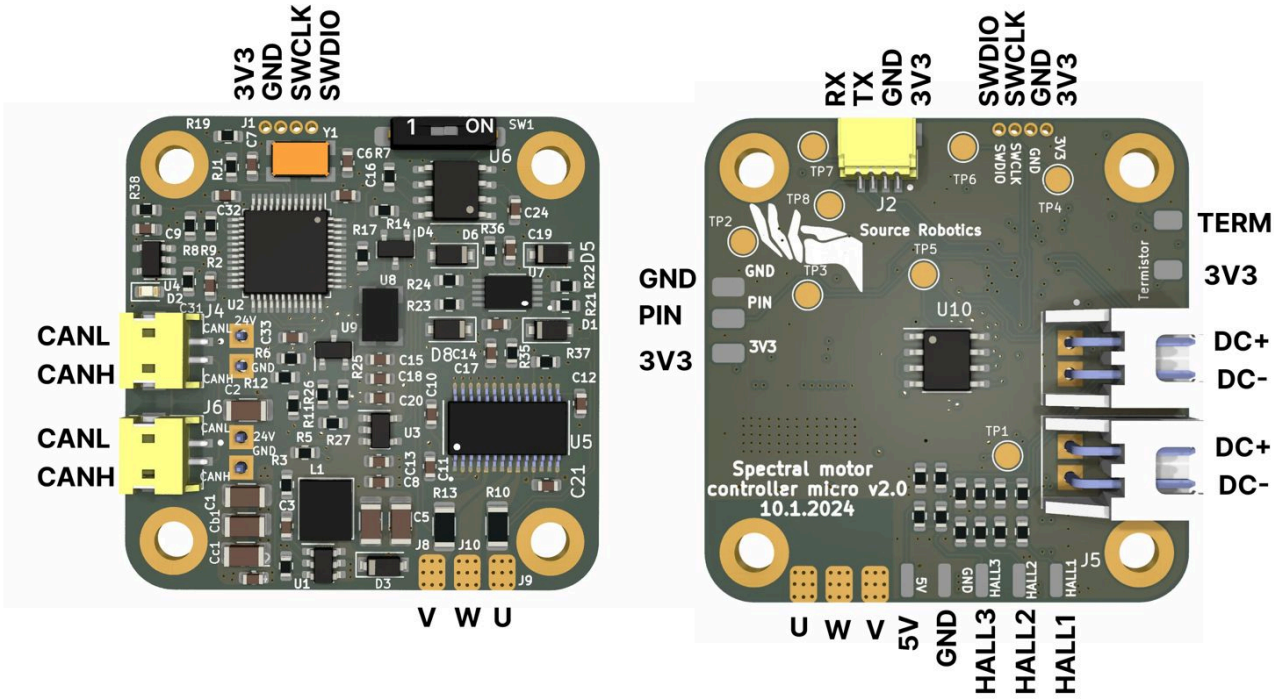


Figure 3.2: The pinout. Left the top of the board with the CAN connectors, the motor pads and the termination switch; right the bottom with the UART connector, the power connectors, the thermistor pads and the encoder chip.

Hall sensors are not implemented

The HALL header and the twelve resistor pads above it are on the board for hall-sensor commutation, but the Spectral firmware does not implement it: the magnetic encoder is the only position source. Leave the header unpopulated.

Polarity and orientation

DC+ to the positive supply, DC− to the negative; a cable the wrong way round destroys the driver. The daisy-chain cables have an orientation too: see Power and the CAN daisy chain.

Note

- UART is TX to RX and RX to TX between the driver and the adapter.
- The UART port and the PIN header are 3.3 V. 5 V on either destroys the board.
- The thermistor pin has a 54.9 kΩ pull-down; the firmware's temperature table is for a 100 kΩ NTC.
- The CAN termination switch is next to the CAN connectors; **ON** connects the 120 Ω resistor.
- The encoder chip sits on the bottom of the board, in the middle: that side faces the magnet.

Dimensions and CAD

The board is 39 × 39 mm with four mounting holes on the NEMA 17 pattern (31 mm apart). The encoder chip is exactly in the middle of the board, so a bracket that centres the board on the motor shaft puts the chip on the shaft's axis.

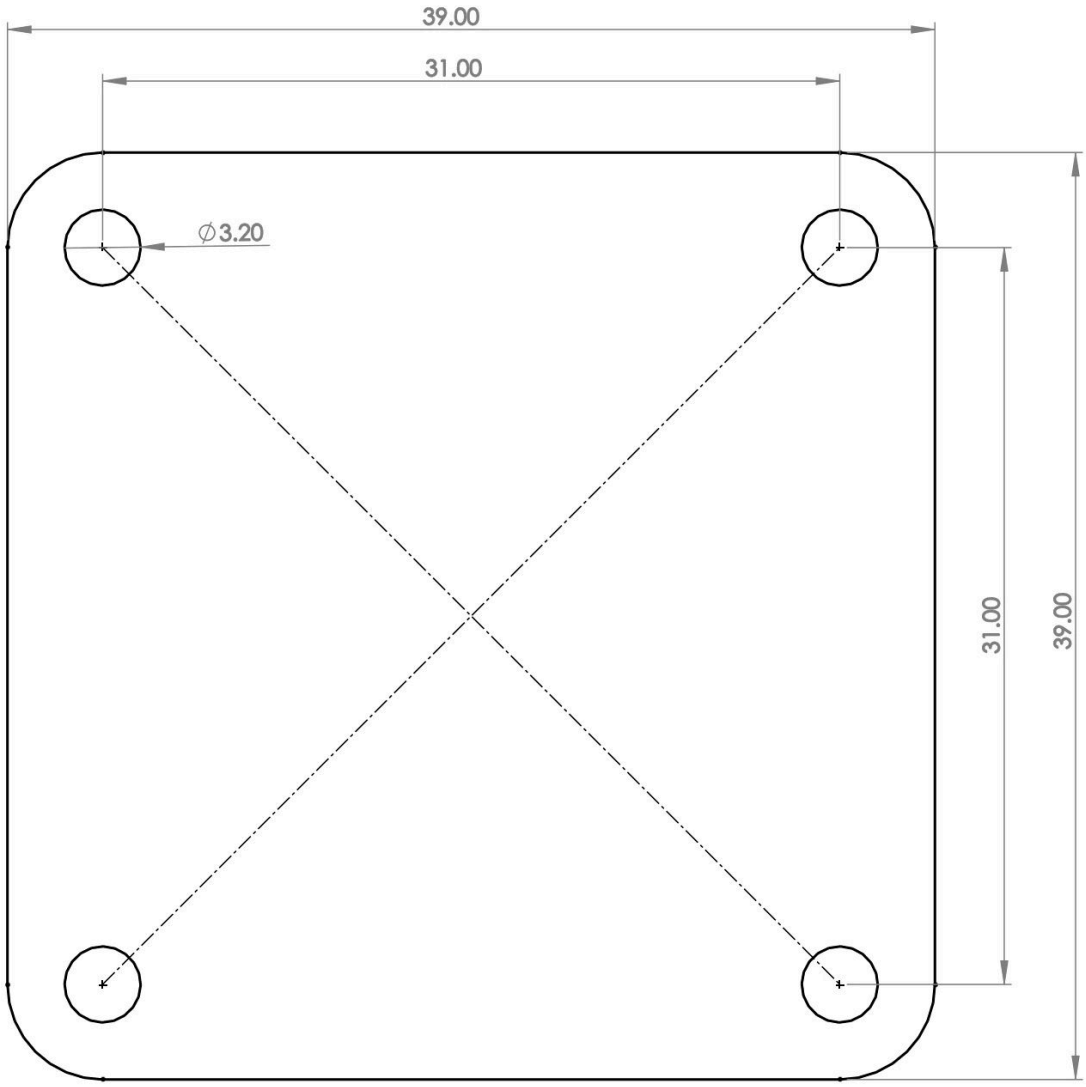


Figure 3.3: Board dimensions and hole pattern.

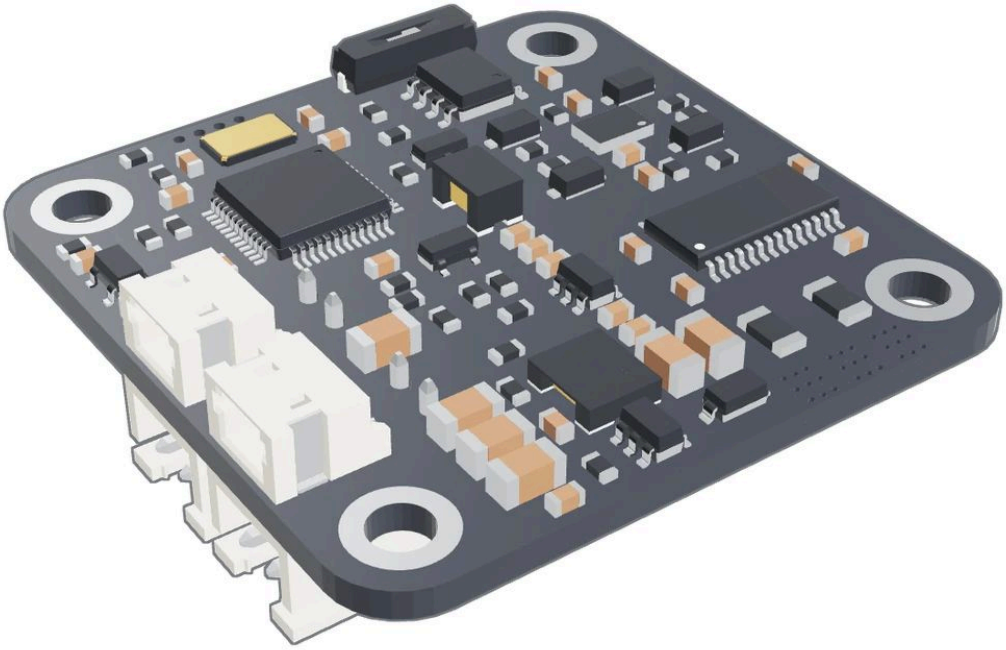


Figure 3.4: The board's model, from its STEP file: the PCB and everything mounted on it.

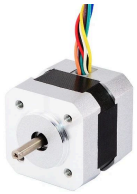
The STEP files of the board and of an example mounting bracket are in the firmware repository, in STEP files: `Spectral_micro_STEP.STEP` is the board, `Spectral_micro_bracket.STEP` the bracket to adapt to your motor (Mounting on the motor).

Downloads and files lists everything that can be downloaded.

Tested motors

The Spectral Micro drives three-phase brushless motors with a phase resistance of about 2 to 24 Ω: the calibration routine's default voltages are made for that range. The motors below have been run on it. Calibration measures each motor's resistance, inductance and pole pairs, so nothing has to be looked up before the first run.

	Motor	Kind	Where
	BGM8108	gimbal motor, 8108 class	AliExpress
	GM8112	gimbal motor, 8112 class	AliExpress
	GM7008H	gimbal motor, 7008 class	AliExpress
	GM6008H	gimbal motor, 6008 class	AliExpress
	GBM5208	gimbal motor, 5208 class	AliExpress
	GM5208	gimbal motor, 5208 class	AliExpress

	Motor	Kind	Where
	GBM5010	gimbal motor, 5010 class	AliExpress
	GB4114	gimbal motor, 4114 class	AliExpress
	2804	gimbal motor, 2804 class	AliExpress
	2804 (RCTimer)	gimbal motor, 2804 class	AliExpress
	2804 100KV	gimbal motor, 2804 class, 100 KV	AliExpress
	42BLS40-24-01	NEMA 17-format brushless motor, 24 V, 26 W, 1.8 A, 219 g	Stepperonline

Before the first test, set the current limit for the motor with `#Ilim` (firmware 107 and older leave 1600 mA after a calibration; from 109 calibration puts your limit back). The factory default of 1000 mA suits the gimbal motors above; a motor rated for more can go up to the driver's 2800 mA. See Current limit and loop tuning.

Choosing a motor

A lower phase resistance and inductance give a higher speed at a given supply voltage; the velocity constant `KV` that calibration finds tells how fast the motor can turn: roughly 0.8 × Vbus × KV revolutions per minute. A high pole-pair count (the large gimbal motors have 11 to 14) gives smooth torque at low speed and limits the top speed through the driver's maximum electrical frequency. Motors with visibly uneven windings, a resistance that differs between the phases or an uneven back-EMF calibrate badly and run rough.

Drone motors

Motors with a phase resistance well under 1 Ω draw a destructive current at the calibration routine's default voltages. The Spectral Micro is not made for them.

4. Mounting and wiring

Mounting and wiring

How the driver goes onto the motor and what gets connected to it: the encoder magnet on the shaft, the board on a bracket behind the motor, the three phases to the motor terminals, the supply and the CAN bus, and the thermistor. Do the pages in order; the magnet first, because it is the one thing you cannot check from the outside later.

In this chapter

- The encoder magnet
- Mounting on the motor
- Wiring the motor
- Power and the CAN daisy chain
- Thermistor

When all of it is done, continue with First setup.

The encoder magnet

The encoder reads a diametrically magnetised magnet on the end of the motor shaft. The magnet must sit centred on the shaft's axis, parallel to the board, about **1 mm** above the encoder chip, and it must not move. Every calibration problem that is not a wiring problem comes from this magnet.

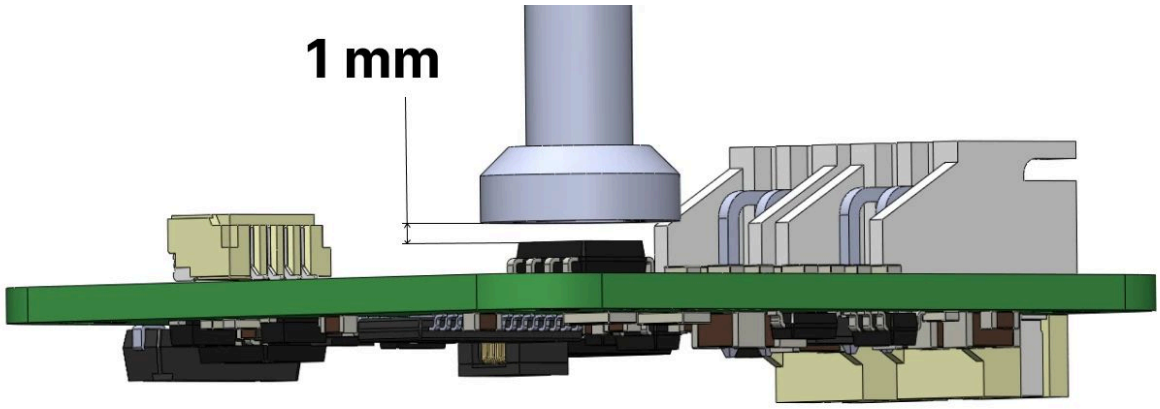


Figure 4.1: About 1 mm between the magnet and the encoder chip in the middle of the board.

The magnet

A neodymium disc, **diametrically** magnetised (north and south across the diameter, not through the thickness). The shop's 6 × 2.5 mm diametrical magnet is the stock size. An axially magnetised magnet looks the same and does not work: the magnet test may even pass, and the position is wrong.

The size is not fixed

6 × 2.5 mm is the stock size, not a requirement. Any diametrically magnetised disc of about 6 mm diameter works, and the thickness is a way to set the gap: a thicker magnet brings its face closer to the encoder, a thinner one farther, so with a shaft or a bracket that leaves the wrong distance you can pick the thickness that gives 1 mm instead of changing the bracket. What matters is the 1 mm from the magnet's face to the chip, the magnet centred on the shaft, and the diametrical magnetisation.

Fitting it

1. The magnet goes on the end of the shaft that faces the board. On most gimbal motors the shaft ends at the bottom of the motor, flush with the housing or a little proud of it, and the board is mounted on that side (Mounting on the motor). On a motor whose shaft ends inside the housing, a short holder pressed into the shaft's end carries the magnet out to where the board can see it.
2. Press the magnet on, centred on the shaft's axis. A magnet that sits off-centre wobbles its field as the shaft turns and gives a position error that repeats every revolution.
3. Fix it with a drop of glue. A magnet that can turn on the shaft is the classic cause of a calibration that reports success and a motor that then runs badly: the magnet slipped between calibration and use.
4. Check the height after the board is mounted: 1 mm, the magnet not touching the chip. A magnet that touches the chip can destroy the microcontroller; a magnet 2 mm away or more gives a weak signal.

Check

The magnet sits centred on the shaft, and once the glue has set it does not move when you push it with a finger.

Checking it

With the driver powered and the serial terminal open, `#Magnet` answers `Magnet Good` when the chip sees enough field, `Magnet Bad` when it does not. The calibration routine runs the same test first.

What the magnet test checks

Only that a magnet is close enough to the encoder. It passes with a magnet that is not glued, sits off-centre or can slip on the shaft, and fails only when there is no magnet or it is too far away. The 1 mm gap, the centring and the glue are yours to check.

Mounting on the motor

The board sits behind the motor, with the encoder chip, in the middle of the board, on the axis of the shaft and 1 mm from the magnet. The board's four holes are on the NEMA 17 pattern, 31 mm apart.

Gimbal motors

A gimbal motor has no standard mounting for a board; a bracket holds the board at the right distance. The firmware repository has an example to adapt to your motor, `Spectral_micro_bracket.STEP` in STEP files: change its hole pattern to the motor's bottom holes and its height so that the magnet ends up 1 mm from the chip.

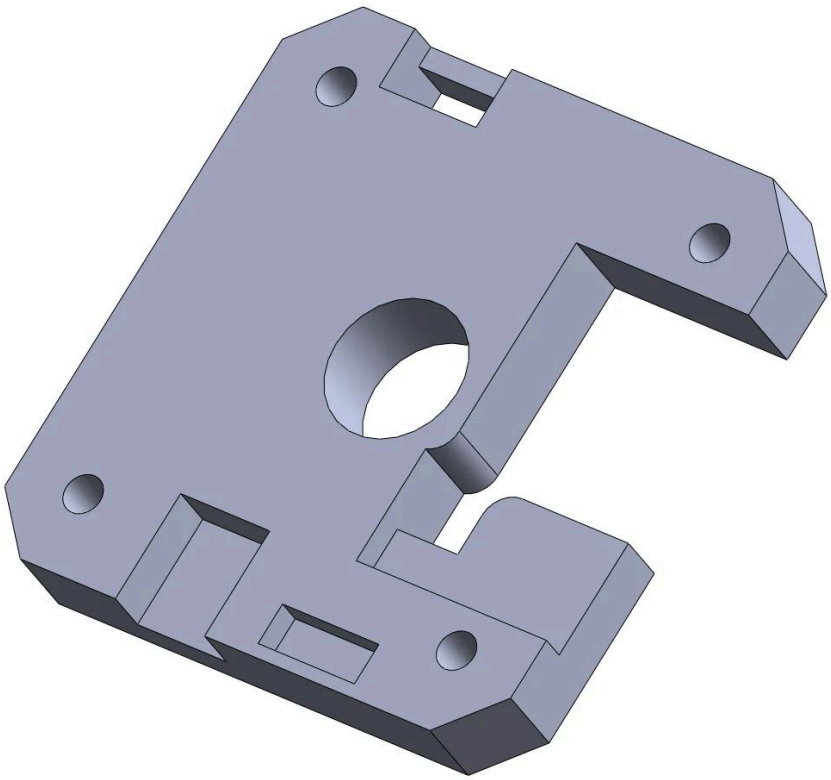


Figure 4.2: The example bracket. The motor bolts to one side, the board to the other; the hole in the middle is for the shaft and the magnet.

1. Bolt the bracket to the bottom of the motor. The magnet on the shaft's end sits in the bracket's centre hole.

2. Bolt the board to the bracket with M3 screws, with the encoder chip (the bottom of the board, see Connectors and pinout) towards the magnet. Tighten the screws evenly; the board must be parallel to the magnet's face.
3. Check the gap between the magnet and the encoder chip: about 1 mm, the magnet not touching it. If it is wrong, change the bracket's height or the magnet's thickness (The encoder magnet).

> **Check**
>
> The board sits parallel to the motor, with about 1 mm between the magnet and the encoder chip, and the rotor turns by hand without the magnet touching the chip. If the gap is wrong: The encoder magnet.

Motors with the NEMA 17 pattern

A brushless motor in the NEMA 17 format, such as the 42BLS40 in the tested motors, takes the board directly, with the motor's own rear screws: remove the four screws, put on a spacer that sets the magnet height, the board on the spacer with the chip towards the magnet, and longer M3 screws in. Check the gap.

> **The board is bare**
>
> Nothing on the board may touch the motor's housing or a metal bracket. Keep the bracket clear of the components on both sides of the board (the connectors on the bottom need room), and keep metal swarf and screws away from the board while you work. Touch earthed metal before you pick the board up and hold it by its edges: static electricity damages its chips.

Wiring the motor

A three-phase brushless motor has three wires. They go to the terminals **U, V and W** on the board (see Connectors and pinout), in any order: calibration finds the order of the phases. Firmware 105 and newer compensates it in software; firmware 101 asks you to swap two wires when it is reversed.

Soldering

1. Cut the leads to a comfortable length: long enough to reach the pads with the board mounted, short enough not to get into the rotor. With firmware older than 105, leave them long enough to swap two of them later.
2. Tin the leads and solder each to one group of pads, U, V or W. The three pads of a group are one terminal.
3. Make sure no strand bridges two groups and that nothing touches the rotor or the housing.

A motor that comes with a connector keeps it: solder a short lead with the mating connector to each pad group.

> **Check**
>
> With the driver unpowered, a multimeter shows about the same resistance between U and V, V and W, and W and U, and no strand bridges two groups.

> **Warning**
>
> Never connect or disconnect the motor while the driver is powered.

Phase order and direction

Calibration spins the motor in open loop and watches the encoder, then checks that a current command moves the rotor the expected way.

With firmware 105 and newer, the order of the phases is detected and compensated, so the motor calibrates identically with any wiring, and positive commands turn the motor the same way whatever the wiring: counter-clockwise seen from the encoder side of the board, see Control modes. To turn the other way, use negative setpoints.

Firmware older than 105 does not compensate the order: calibration stops with `Phase order is wrong! Switch motor phases and try again!`. Swap any two of U, V and W and calibrate again.

If the motor does not respond to the commutation check, calibration says so (`Commutation/alignment check failed!`). That means a phase is not connected, `#Iqoffset` is set too low or the magnet slipped, not that the wires are in the wrong order; see Calibration.

Power and the CAN daisy chain

Each driver has two power connectors (JST XH) and two CAN connectors (JST ZH). The second connector of each pair feeds the next driver, so a chain of drivers hangs on one supply cable and one CAN cable.

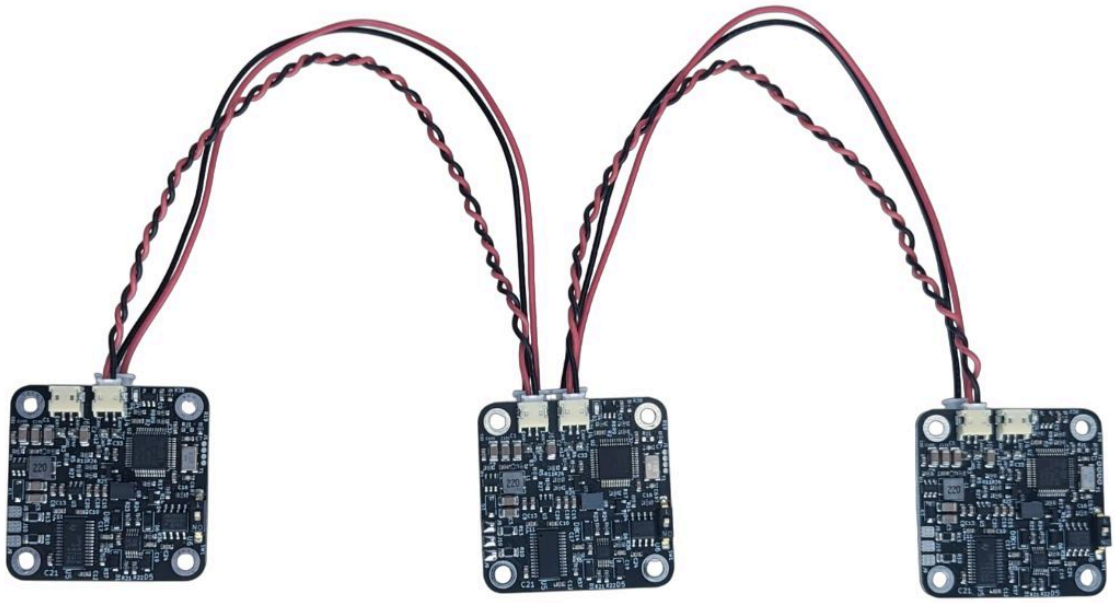


Figure 4.3: Power and CAN pass from driver to driver. The first and the last node on the bus have their termination switch on.

Power

- 12 to 29 V DC, 24 V recommended. **DC+ to the positive supply, DC− to the negative**; reversed polarity destroys the driver.
- The power connectors are rated for 4 A. A chain of drivers shares that: with motors at 1.5 A each, plan for at most two or three drivers per chain that draw their full current at the same time (three or four with 10 Ω gimbal motors), and feed a longer chain from both ends or from several supplies.
- A motor that decelerates pushes energy back into the supply. A lab supply or a battery absorbs it; a supply that cannot (some switch-mode supplies) sees its voltage rise, and the driver raises a Vbus error above `#Vmax` (29 V after `#Default`).

The cable orientation

Danger

The daisy-chain cables are keyed by their orientation, not by the connector: the same 2-pin cable fits both ways, and the wrong way puts DC+ on DC−. Every cable must run as in the picture below, the same way at both ends. A wrong cable destroys the driver at the moment it is powered.



Figure 4.4: The orientation of the daisy-chain cables, the same for the 2-pin power and CAN cables.

The cables from the shop are made this way: CAN cable, power cable, and all of them under connectors and cables.

CAN

- CAN 2.0, 1 Mbit/s, 11-bit identifiers. Up to 14 drivers on one bus, each with its own node ID, 0 to 13 (0 by default; set it with `#CANID` or Send_CAN_ID and save). The firmware accepts 14 and 15 too, but the bootloader reserves them: a driver on one of those IDs cannot be updated over CAN.
- Twisted-pair wire; keep the bus short at 1 Mbit/s (a few metres at most).
- **Termination:** the first and the last node on the bus need a 120 Ω resistor between CAN H and CAN L. The switch next to the CAN connectors turns the driver's resistor on (**ON**). Drivers in the middle stay off. A bus with the right termination measures about 60 Ω between CAN H and CAN L with everything unpowered.
- A PC joins the bus with the CANvas adapter, which has its own termination.

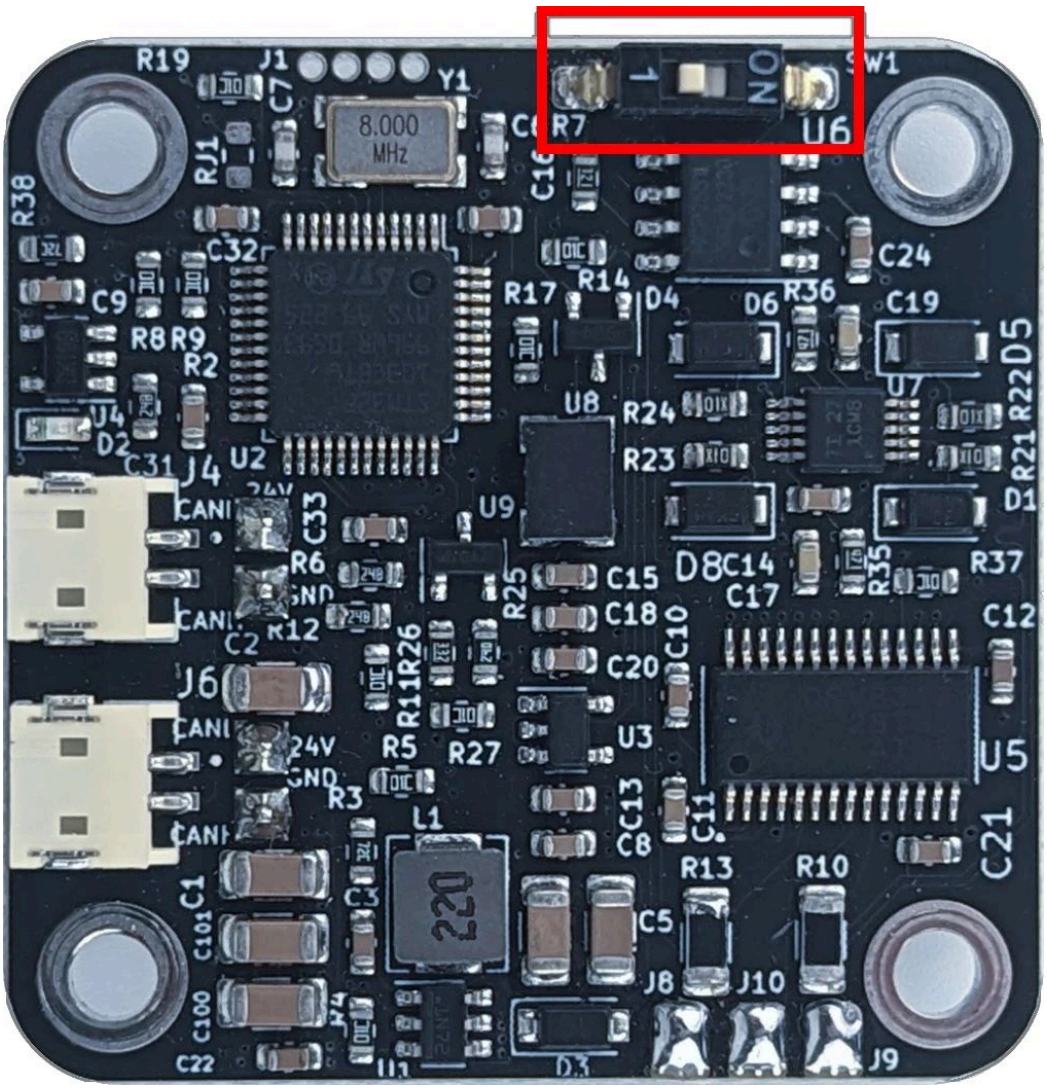


Figure 4.5: The termination switch, ON for the first and the last node.

Thermistor

A 100 kΩ NTC thermistor in the motor lets the driver stop before the motor overheats. It is optional, and off by default; use it on any motor that runs near its current limit for long, and on a motor that holds a load at standstill, which heats one winding without any movement to show it.

Without it, nothing protects the motor from overheating

The driver limits the current, not the temperature. A motor that runs continuously, holds a load for long, or is set up badly (a current limit above its rating, loops that oscillate, a wrong resistance or inductance) can overheat and burn its windings without any warning. With the thermistor fitted and enabled, the driver stops first. The starter kit includes one.

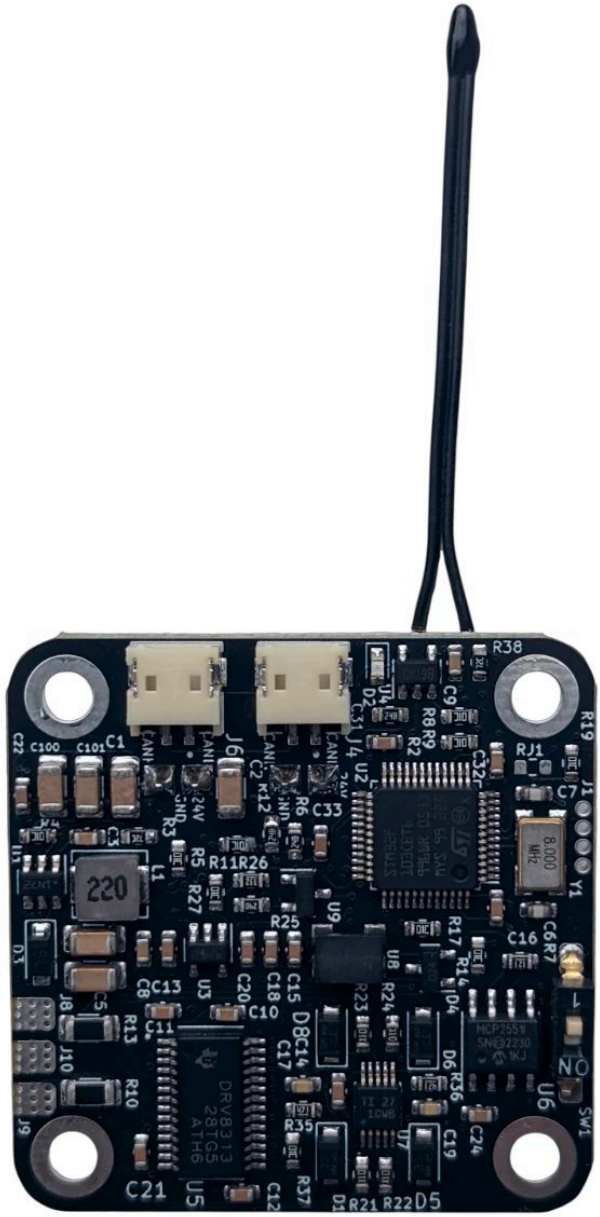


Figure 4.6: The thermistor on the TERM and 3V3 pads.

Fitting it

Solder the thermistor's leads to the two thermistor pads, TERM and 3V3 (Connectors and pinout). Put its bead inside the motor's windings, where the heat is: on a gimbal motor the windings are open at the bottom and can be reached through one of the mounting holes or past the edge of the stator.

Thermal cement holds the bead in place and conducts the heat; keep the leads clear of the rotor.



Figure 4.7: Thermal cement: it sets hard and conducts heat. Thermal paste stays soft and lets the bead move, ordinary glue insulates it. Where to buy it.

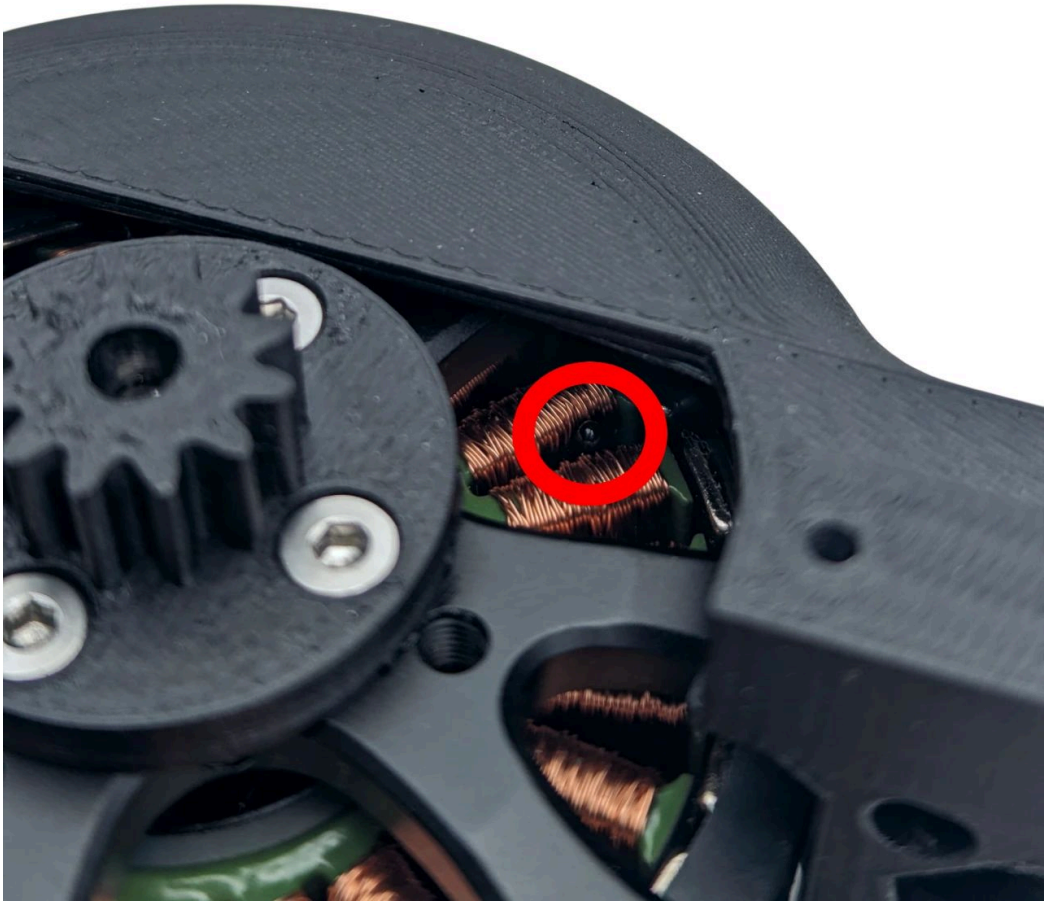


Figure 4.8: The thermistor placed in the windings.

Enabling it

```
#Term 1
#Tlim 75
#Save
```

`#Temp` then prints the temperature; with the thermistor disabled it prints `Temp Off`. Above the limit the driver raises a temperature error and stops; it clears with `#Clear` once the motor has cooled.

Below −20 °C it raises the same error, so `#Term 1` without a thermistor, or with a broken lead, stops the driver too. 75 °C is the default limit and a safe one for the enamel of most small motors; do not go higher unless the motor is rated for it.

Over CAN, Send_Respond_Temperature returns the temperature, and the temperature error is one of the bits of Send_Respond_State_of_Errors.

5. First setup

First setup

From a mounted and wired driver to a calibrated, tuned motor. Motor GUI in the browser is the usual tool for it; every step also gives the terminal command, for a terminal or for the console in Motor GUI.

1. Motor GUI: the browser, the serial adapter, opening the port, what is on the screen.
2. The serial terminal: the alternative — PuTTY, the PlatformIO monitor or the Arduino monitor, and the `#` commands.
3. Calibration: the factory defaults, then the run that measures the motor and finds its pole pairs, phase order and constants.
4. Encoder table and anti-cogging: two corrections the driver can measure on itself after a calibration, how to run them and how to check them.
5. Current limit and loop tuning: the current limit for the motor, the loop gains if the defaults do not suit it, the speed estimate and the delay compensation.
6. Saving the configuration: nothing survives a power cycle until it is saved; where it is kept.

A driver that has been through these pages is ready for Operation. The Quick start is the same path in four short parts.

Motor GUI

Motor GUI is the usual way to set up, test and tune a driver: a tool in the browser that connects to one driver over the serial adapter or to a bus of drivers over the CANvas adapter, shows the state, moves the motor, runs the calibration, tunes the loops with live plots, records telemetry and updates the firmware. Nothing is installed. Everything it does, the driver also does from a terminal or from Python — the GUI reads and writes the same settings — so every procedure in this manual shows the GUI first and the terminal command beside it.

The browser

Motor GUI reaches the hardware through the browser's serial port access (Web Serial), which Chromium-based browsers have: Chrome, Edge, Brave, Opera. Firefox and Safari cannot open serial ports.

Two ways to connect

	Adapter	Use it for
One driver over serial	the 3.3 V USB-to-serial adapter on the driver's UART connector, 256000 baud	the first setup and calibration (calibration runs over UART only), tuning on the bench
A bus over CAN	the CANvas adapter; it appears as a serial port and speaks slcan	every driver on the bus by its node ID: testing a machine, updating firmware on all joints

Opening the port

1. Plug in the adapter and the motor supply. The adapter alone powers the driver's logic, enough to open the port and read the settings, but the driver reports a Vbus error and will not calibrate or move until the supply is on.
2. Open motorgui.com, choose the connection kind, and **Connect**. The browser asks which port; pick the adapter's.
3. The driver's identity appears: hardware version, firmware version, serial number, batch date, whether it is calibrated and activated, and the active errors. In a terminal, `#Info`, `#SERNUM` and `#Error` print the same.

If the port is not offered, the browser is not Chromium-based, or another program holds the port (a terminal left open, PlatformIO's monitor), or the operating system does not give the browser the port:

Windows

The adapter is `COM` and a number. Device Manager lists it under **Ports (COM & LPT)**: the entry that disappears when you unplug the adapter is yours. An adapter that shows a warning sign there instead needs the driver of its USB chip (on most 3.3 V adapters a CP210x or a CH340) from the chip maker.

Linux

The serial adapter is `/dev/ttyUSB0` on most adapters, the CANvas adapter `/dev/ttyACM0`; `ls /dev/ttyUSB* /dev/ttyACM*` lists them. To open them without root, join the `dialout` group once, then log out and in again:

```
sudo usermod -aG dialout $USER
```

macOS

The serial adapter is `/dev/cu.usbserial-` and a few characters on most adapters, the CANvas adapter `/dev/cu.usbmodem` and a number; `ls /dev/cu.*` lists them. Use the `cu.` name, not the `tty.` one, in a terminal or a script.

The browser's port picker lists the same adapters. If the port opens and nothing appears: the driver is unpowered, or TX and RX are crossed on an adapter that is not the Source Robotics one.

What is on the screen

- **State**: the identity above, the control mode the driver is in, the errors, and **Clear errors**.
- **Jog**: *Jog position*, *Jog velocity* and *Jog current* move the motor in the matching control mode; *Send setpoint* sends a value in the current mode. *Live stream while dragging* follows a slider as you move it.
- **Plots** of position, speed and current, panning and zooming, with the value under the pointer.
- **Settings**: the motor parameters (pole pairs, resistance, inductance), the limits (current, velocity, voltage), the loop gains with *Apply gains* and *Apply preset*, the CAN node ID, the watchdog, the thermistor. The GUI marks *Unsaved changes* until you **Save** to the driver's EEPROM; *Default* restores the factory values.
- **Calibration**: starts `#Cal` and shows the driver's log step by step; afterwards the resistance, inductance and pole pairs it measured, the phase order and direction, Kt, KV and the flux linkage.
- **Telemetry**: the stream from the driver, and recordings of it for a test or a bug report.
- **Firmware update**: the releases from GitHub, flashed through the bootloader; *Scan bootloaders* on a bus.
- **Gripper**: the gripper controls, for a driver in gripper mode.
- **Console**: type any `#` command by hand; the answer appears below it.
- **Wiring diagrams** and **Keyboard shortcuts** under help.

Screen guide

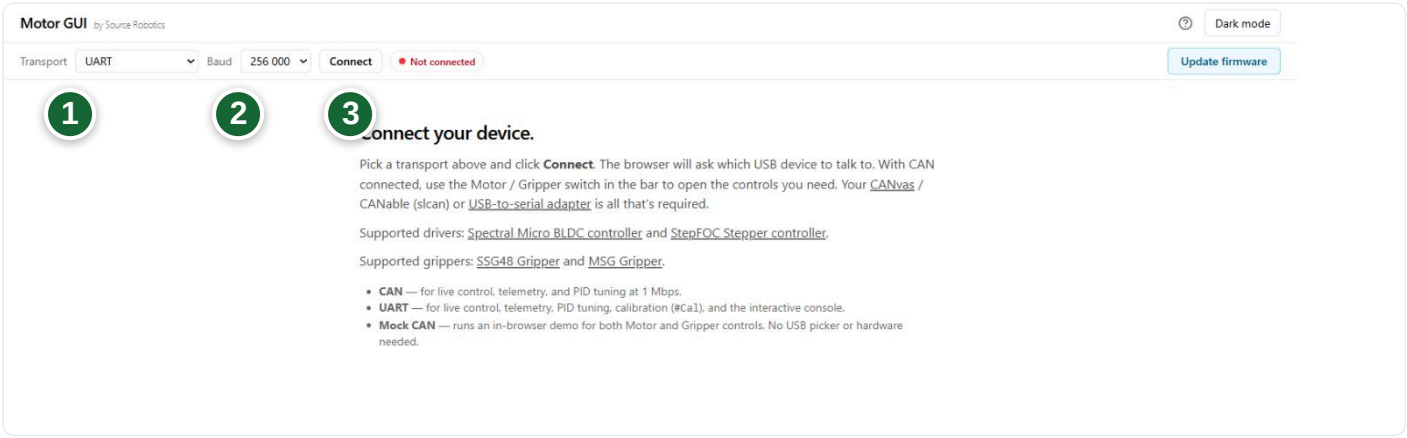


Figure 5.1: **1. Transport:** choose UART for the serial adapter. **2. Baud:** 256000 for a stock driver. **3. Connect:** choose the adapter in the browser's port picker.

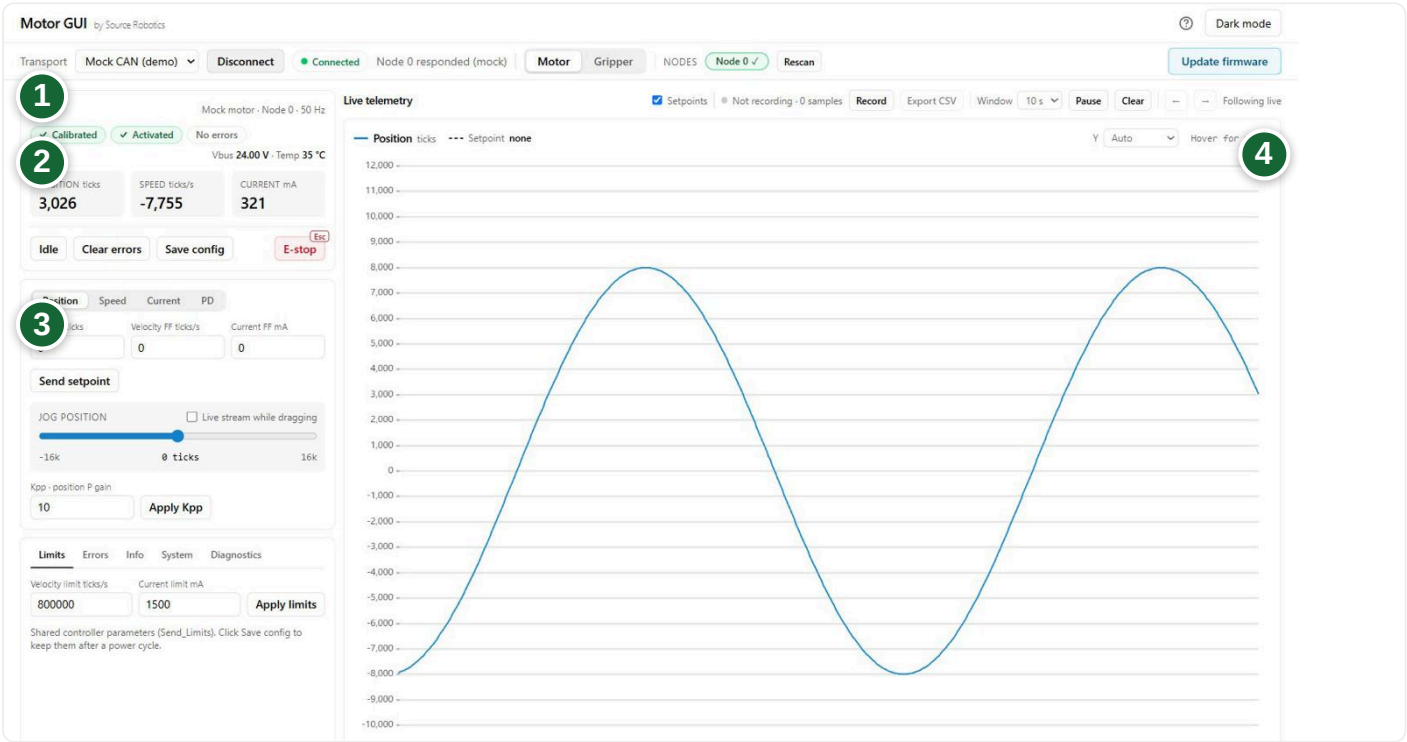


Figure 5.2: **1. Connection and node:** confirm the intended driver. **2. State:** calibration, activation and faults. **3. Command mode:** setpoints and jog controls. **4. Live telemetry:** measured position, speed and current. This screenshot uses **Mock CAN (demo)**, not physical hardware.

Check

After connecting real hardware, the state and identity belong to your driver. If they do not appear, check the transport, the selected port, the supply and the TX/RX wiring before trying to move.

Try it without hardware

Open Motor GUI, choose **Mock CAN (demo)**, and press **Connect**. The motor and gripper panels run in the browser without a port picker. Motor calibration (`#Cal`) still requires a real UART connection; the CAN demo does not perform it.

What next

Calibration is the first thing to do with a connected driver; Working in Motor GUI covers everyday use: moving the motor, the plots, recordings, several motors on a bus.

The serial terminal

Motor GUI is the usual way to work with the driver; a plain serial terminal is the alternative, and it is what the console inside Motor GUI is.

Everything in this manual that is done in the GUI is also one `#` command, and the terminal shows the driver's answers exactly as it prints them, which is useful when something does not work.

The driver talks on its UART connector at 256000 baud, 8N1, 3.3 V. Any terminal that sends a line ending in CR+LF works.

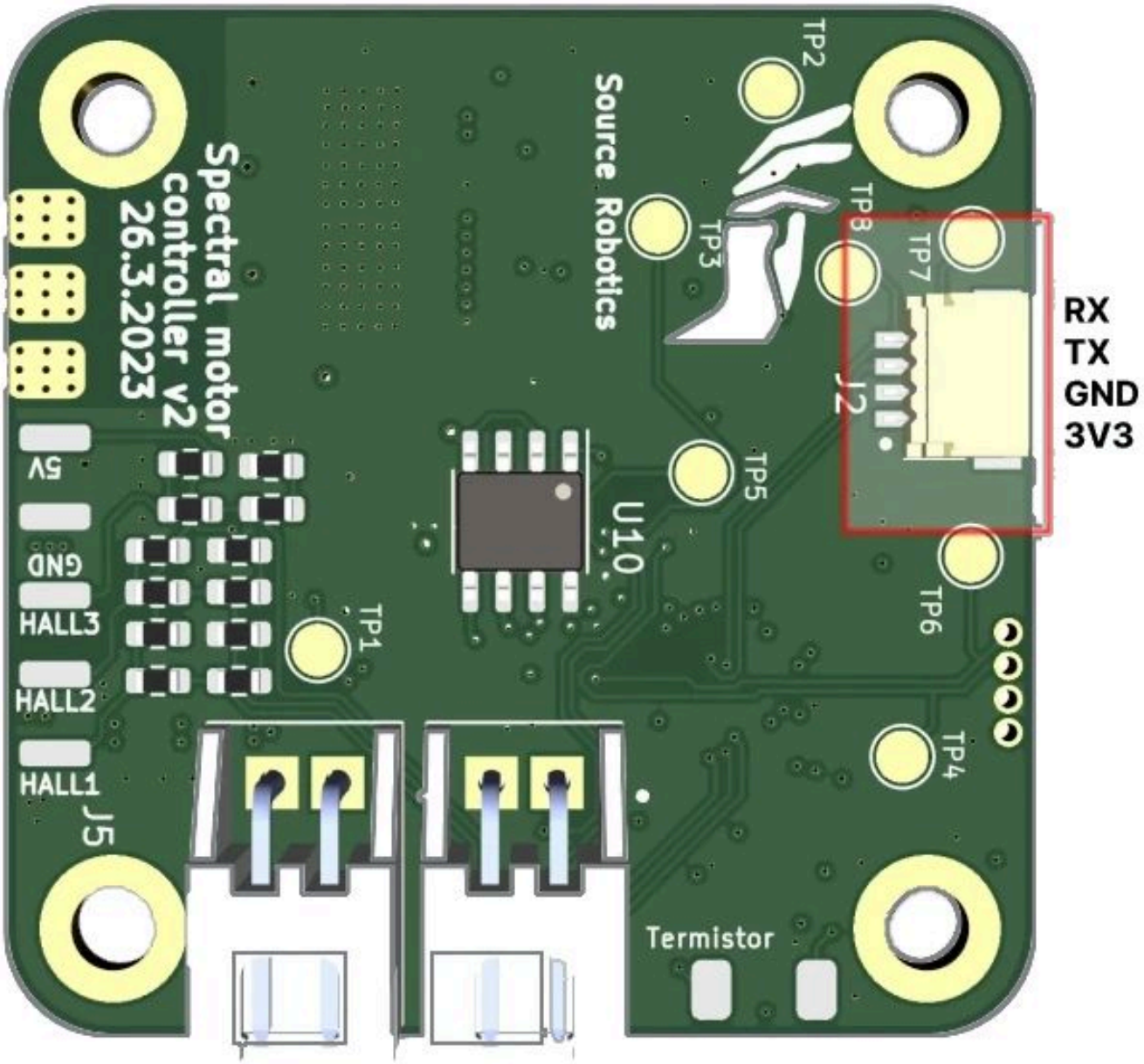


Figure 5.3: The UART connector. TX of the driver goes to RX of the adapter and RX to TX.

Connecting

1. Plug the USB-to-serial adapter into the UART connector with the UART cable. With another adapter: 3.3 V only, and cross TX and RX: the driver's TX to the adapter's RX, the driver's RX to the adapter's TX.
2. Connect the motor supply. The adapter's 3.3 V powers the driver's logic, so the driver answers over UART with only the adapter plugged in, but it reports a Vbus error and refuses calibration and every motion command until the 12 to 29 V supply is on.
3. Find the adapter's port:

Windows

The adapter is `COM` and a number. Device Manager lists it under **Ports (COM & LPT)**: the entry that disappears when you unplug the adapter is yours. An adapter that shows a warning sign there instead needs the driver of its USB chip (on most 3.3 V adapters a CP210x or a CH340) from the chip maker.

Linux

The serial adapter is `/dev/ttyUSB0` on most adapters, the CANvas adapter `/dev/ttyACM0`; `ls /dev/ttyUSB* /dev/ttyACM*` lists them. To open them without root, join the `dialout` group once, then log out and in again:

```
sudo usermod -aG dialout $USER
```

macOS

The serial adapter is `/dev/cu.usbserial-` and a few characters on most adapters, the CANvas adapter `/dev/cu.usbmodem` and a number; `ls /dev/cu.*` lists them. Use the `cu.` name, not the `tty.` one, in a terminal or a script.

Danger

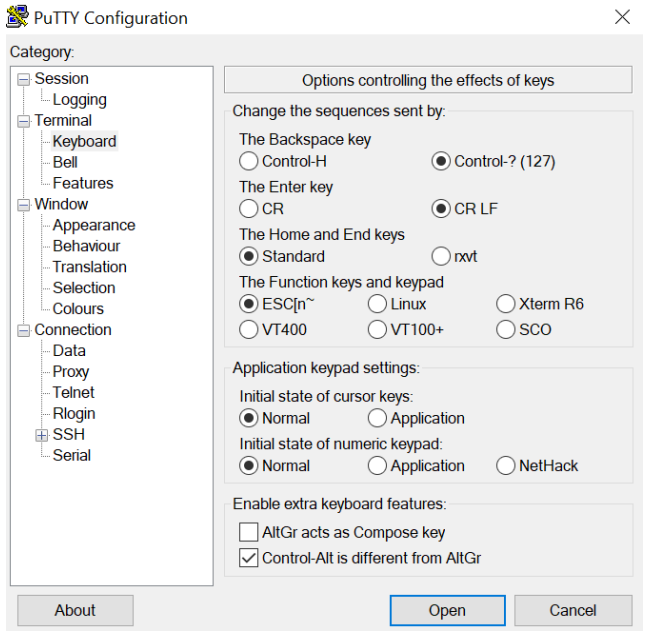
5 V on the UART pins destroys the board.

A terminal

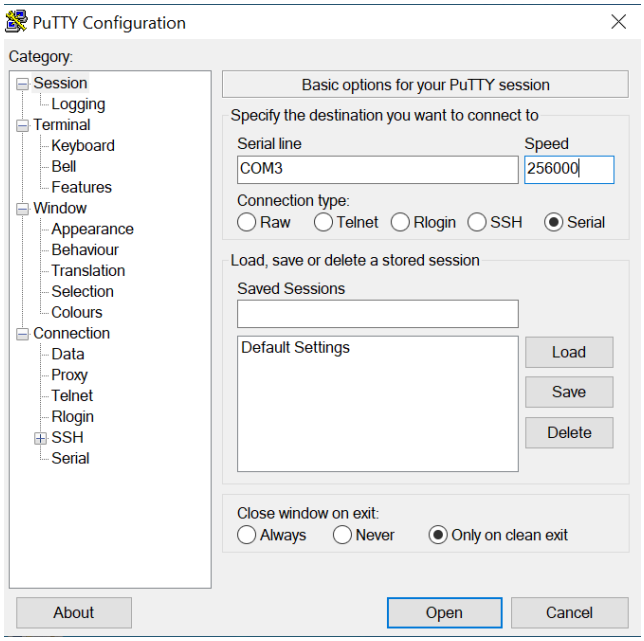
PuTTY

PuTTY needs to send CR+LF when you press Enter, which the stock build cannot. Use the modified PuTTY: under *Terminal → Keyboard* select the CR LF option, then open the serial port at 256000.

Keyboard setting



Session



VS Code (PlatformIO)

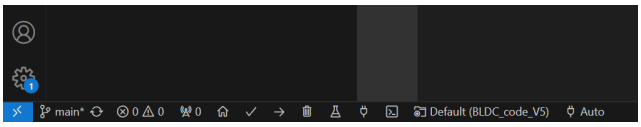
With the firmware project open in VS Code with the PlatformIO extension, the serial monitor (the plug icon in the status bar) uses the speed and the `send_on_enter` filter from `platformio.ini`, so a line goes out with CR+LF when you press Enter.

platformio.ini

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:genericSTM32F103CB]
platform = ststm32
board = genericSTM32F103CB
framework = arduino
upload_protocol = stlink
;upload_port = COM3
build_flags =
    ;-DHAL_ADC_MODULE_ONLY
debug_tool = stlink
monitor_port= COM3
monitor_speed = 256000
monitor_filters = send_on_enter
```

The monitor



Arduino IDE

Select the adapter's port under *Tools*, any board (the monitor does not care), open the serial monitor, set 256000 baud and *Both NL & CR* as the line ending.

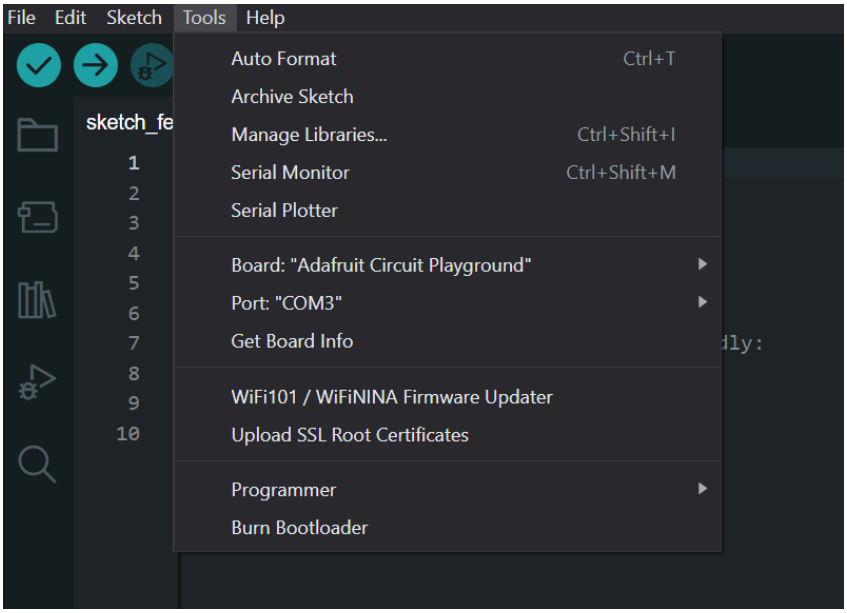


Figure 5.4: The Arduino serial monitor at 256000 baud, with Both NL & CR as the line ending.

The first commands

```
#Info
```

prints the driver's identity: the hardware and firmware version, the CAN ID, the calibration state and the motor parameters. `#Help` (firmware 106 and newer) prints a short command reference, `#Error` the active errors, `#SERNUM` the serial number and `#Param` the gains.

Check

`#Info` answers with the driver's identity, the hardware and firmware version first. If nothing comes back: the baud rate, the CR+LF line ending, TX and RX, see The driver does not answer.

Every command starts with `#`, and a Set/Get command takes one number after a space: `#P 16384` moves to position 16384; `#P` alone prints the position. Names are case-sensitive. The whole set is in the UART commands reference, and Over UART explains the protocol.

Calibration

Calibration teaches the driver its motor. It checks the magnet and the supply, then measures the motor's phase resistance and inductance and computes the current loop gains from them.

It spins the motor in open loop to find the pole pairs and the direction, checks the order of the phases and aligns the encoder to the rotor field, and measures the motor constants Kt, KV and the flux linkage. Nothing about the motor has to be entered first.

It runs once per motor; the result is kept by saving. From firmware 109 every `#Cal` goes on by itself to two more runs, the encoder table and the anti-cogging, and saves. `#Calonly` is the calibration alone.

The motor turns

Calibration spins the motor in both directions, fast at times. Fix the motor to the bench and take everything off the shaft.

Supply voltage

Calibrate at 24 V. At a lower voltage the resistance, the inductance, KV and Kt come out wrong.

The settings

The routine uses six settings, which it prints when it starts. The defaults suit motors with a phase resistance of 2 to 24 Ω; change them only for a motor outside that range.

Setting	Default	Command	What it does
Current loop bandwidth	160 Hz	`#Cbw`	the bandwidth the current loop gains are computed for (Kpiq ≈ ω·L, Kiiq = ω·R·T, with ω = 2π × the bandwidth and T the 200 µs control period); too high makes the motor vibrate and whine. 500 Hz up to firmware 107
Resistance voltage	1000 mV	`#Resv`	the voltage the resistance measurement starts at; it rises until the power reaches the limit below
Max power dissipation	12 W	`#Calpwr`	the power the resistance and inductance measurements may put into the motor

Setting	Default	Command	What it does
Open loop voltage	5000 mV	`#Openv`	the voltage of the open-loop spin that finds the pole pairs
Open loop speed	20 rad/s	`#Opens`	the electrical speed of that spin
Phase search voltage	5000 mV	`#Vp`	the voltage of the spin that checks the phase order

A seventh setting is not printed: `#Iqoffset`, 300 mA by default, the current of the angle-offset search and of the commutation check. None of the seven is saved; after a power cycle they are back at these defaults. The current gains the bandwidth gives are saved with the calibration.

Low-resistance motors

These voltages are made for motors of 2 Ω and more. On a motor with a much lower resistance, such as a drone motor, they drive a destructive current. The Spectral Micro is not made for such motors; if you must try one, lower the three voltages first. Lowering the current limit does not help: calibration sets its own.

Calibration sets its own current limit

The routine does not use the current limit you set. It raises the limit to 1600 mA, and drives the motor with up to 1250 mA while it measures Kt and KV and with `#Iqoffset` while it aligns the encoder. From firmware 109 it puts the limit that was in force back when it ends; firmware 107 and older left 1600 mA in force (known issue KI-2).

1. Restore the defaults

The EEPROM of a new driver, or of one that ran another motor, can hold values that do not fit: another motor's parameters, or junk that shows as `-1` in `#Info`. Calibration starts from what is there, so start from the factory values.

Motor GUI

Default in the settings.

Terminal

```
#Default
```

This also sets the hardware version and batch date to factory values, and clears an encoder table and an anti-cogging set from another motor; note what `#Info` printed if you need them. From firmware 109 the CAN node ID is kept. Saving the configuration says what else it does.

2. Run it

We recommend a Motor GUI preset

In Motor GUI's **Presets** tab, apply *Generic motor*, *BLDC motor on Spectral Micro (soft)* before you calibrate. It writes a low current limit and soft loops and leaves the resistance, the inductance and the pole pairs to the calibration, which measures them, so the motor calibrates safely. Raise the limits and the gains from there (Current limit and loop tuning). More: Motor GUI.

Motor GUI

Start the calibration. The driver enters *Calibration mode* and the GUI shows its log step by step; when it finishes, the resistance, inductance and pole pairs it measured, the phase order and direction, Kt, KV and the flux linkage appear in the settings.

Terminal

```
#Cal
```

The LED flashes fast, about 10 times a second, while the routine runs, and the driver prints each step as it goes. The numbers below are those of one tested motor; yours will differ:

```
----------------
Calibration mode
-----Settings-----
Current loop bandwidth: 160.00Hz
Resistance voltage: 1000mV
Max Power dissipation: 12.00W
Open loop voltage: 5000mV
Open loop Speed: 20.00Rad/s
Phase search voltage: 5000mV
----------------
Magnet test: Passed
Vbus test: Passed
Resistance = 7.026575Ohm
Inductance = 0.001492H
Kp_iq = ...
Ki_iq = ...
Pole pair = 11
Dir = 1
Calibrating angle offset... stage 1/13, sweep point 0
Calibrating angle offset... stage 2/13, sweep point 0
...
Phase order = 2
Phase order is good!
KV is: ...
Kt is: ...
KV runs: ... ... ... ... (same direction within ... %)
Flux linkage is: ...
----------------
Calibration success!
Next, by itself: the encoder table, then anti-cogging (about a minute; keep the shaft free).
Everything is saved when they finish (#Encal state, #Cogcal state)
Going to idle!
----------------
```

The `Calibrating angle offset...` lines come every half second while the encoder is aligned, which can take about 40 seconds; the routine has not stopped.

`#Calonly` runs the same calibration and nothing after it; it ends with `Save config with #Save` instead.

From firmware 110 `#Estop` or `#Close` stops a running calibration; `#Idle` does not. It ends as a failed calibration (`Calibration failed!`): the motor is released, and the driver is not calibrated until it calibrates again. After `#Estop`, send `#Clear` first.

On firmware 109 a reset (`#Reset`) was the only clean way over UART: `#Close` switched the mode under the running calibration and left it half done.

What the steps are:

Step	What happens	Fails with
Magnet test	the encoder sees enough field: a magnet is there and close enough. The test says nothing about centring, the gap or the glue	`Magnet test: Failed`: magnet missing, too far, or axially magnetised

Step	What happens	Fails with
Vbus test	the supply is within limits	`Vbus test: Failed`
Resistance	a rising voltage is put on the windings and the current measured; the motor clicks. The resistance follows from the two	`Resistance test: Failed`: no current flows, a phase is not connected, or the motor is far outside the 2 to 24 Ω range
Inductance	a voltage step and the rate at which the current rises give the inductance. Kp_iq and Ki_iq are computed from R, L and the bandwidth	`Inductance test: Failed`
Open loop	the stator field is turned at the open-loop speed and the encoder watched: the pole pairs are the ratio of electrical to mechanical turns, and 1 to 49 are accepted. The direction the rotor turned is remembered as the commutation direction (`Commutation dir` in `#Info`); `Dir` is always 1 after a calibration	`Open loop: Failed`, then `The rotor did not follow the open-loop field (pole pairs must come out as 1 to 49).` and `Check the motor's three wires and that the shaft is free.`
Phase order	a current command must move the rotor the expected way, and the encoder's zero is aligned to the rotor field (the angle offset, `#Offset`). A motor that does not respond gets a second try at another angle	`Commutation/alignment check failed!`, `Motor did not respond as expected to a current command...` and a line `Spin-confirm delta: … ticks (need > 8192), Vbus: … mV` that says how far the motor turned (half a revolution is needed) and at what voltage
Kt, KV, flux linkage	four spins from standstill, about a second each and alternately forward and back; KV is the mean speed per volt available, and the constants follow from it. `KV runs:` in the log shows all four and how far they agree	nothing. `KV runs disagree` is a warning, not a failure: two runs in the same direction differ by more than 5 %. The shaft was not free, or something held it; calibrate again

The measured values are estimates

The resistance, the inductance and the motor constants differ a little from one run to the next, and KV, and with it Kt and the flux linkage, can be off by a factor of two or three on cheap motors with uneven windings. For the loops this does not matter; for a torque estimate in newton-metres, measure Kt yourself.

What follows the calibration

Half a second after `Calibration success!` of a `#Cal` the driver starts two runs on its own: the encoder table, then the anti-cogging identification. Keep the shaft free for about a minute more.

When they succeed, everything is saved. A command from a host (a motion command, `#Idle`) stops them; then nothing is saved, and the calibration is kept only if you `#Save` it.

A host that takes the drive before they start has 3 s: after that the runs are dropped and do not start later. `#Cal` always runs them; `#Calonly` never does. Encoder table and anti-cogging explains both.

> **Set the current limit before you calibrate**
>
> The encoder table turns the field with a current of 80 % of the current limit, for about 20 s. Calibration uses its own 1600 mA for its steps and puts the limit that was in force back before the encoder table runs. So set `#Ilim` for the motor before `#Cal` (known issue KI-19).

3. Set the current limit and save

Calibration puts back the current limit that was in force when it started (firmware 107 and older left 1600 mA). If that is not your motor's yet, set it now (Current limit and loop tuning), then save. After `#Cal` the driver saves by itself when the runs succeed; save anyway once they are done, or after `#Calonly`.

Motor GUI

Set the current limit in the limits, then **Save**. The GUI marks *Unsaved changes* until you do.

Terminal

```
#Ilim 1000
#Save
```

Without it the calibration is gone at the next power cycle. After `#Calonly` the driver is idle. After `#Cal` it ends the two runs holding the position the shaft reached: `#Idle` releases it. The LED shows normal operation; `#Info` shows *Calibrated*, the pole pairs, the resistance and inductance, Kt and KV, the phase order and the commutation direction.

> **Check**
>
> After a power cycle `#Info` still prints `Calibrated`, and Motor GUI shows *Calibrated*. `Not calibrated` means the save did not happen: calibrate and save again.

When to calibrate again

- After changing the motor, the magnet or the wiring.

- After restoring the defaults.
- After an update from firmware 107 or older to 109, where the motor can turn freely: the older calibration stored the resistance and the inductance too low. An assembled SSG-48 keeps its calibration and gets its preset (Upgrading from older firmware).
- After a firmware update that clears the calibration (a release note says so when it does).

A calibration that reports success but a motor that runs badly afterwards usually means the magnet moved on the shaft between calibration and use: fix the magnet and calibrate again.

Troubleshooting

Message	Cause
`Magnet test: Failed`	no magnet, magnet too far (more than about 1.5 mm), or an axially magnetised magnet
`Vbus test: Failed`	supply off, too low, or the cable the wrong way round (if the driver survived it)
`Resistance test: Failed`, `Inductance test: Failed`	a phase not connected or shorted; a motor far outside the 2 to 24 Ω range
the motor wobbles left and right and the routine fails	one phase is not connected
`Open loop: Failed`	the rotor did not follow the field, or the pole pairs came out outside 1 to 49: a phase open, the shaft not free, the open-loop voltage (`#Openv`) too low for a high-resistance motor, the supply's current limit set too low
`Commutation/alignment check failed!`	the motor did not respond to the current command: a phase open, `#Iqoffset` set too low, a magnet that slips. Run `#Cal` again. Firmware 101 printed `Phase order is wrong! Switch motor phases and try again!` here; since 105 the wiring is compensated and swapping wires does not help
`Cal: not started, the E-stop is latched (#Clear first)`, or `Stopped by an E-stop: #Clear, then calibrate again.` after `Calibration failed!`	an E-stop is latched or ended the calibration: `#Clear` , then `#Cal` ; the alignment starts again from its first stage

Message	Cause
the routine stops in a step and prints nothing more	stop it with `#Close` (firmware 110) or reset the driver (`#Reset`), and try again
success, but the LED stays on and no error is reported	`#Info` shows `-1` somewhere: the EEPROM holds a value from an older release (The check at power-up); restore the defaults and calibrate again
the resistance, KV or Kt look wrong	the supply was not 24 V, or the motor is one of the uneven ones; see the note above

Calibration runs over the serial connection only: a terminal, or Motor GUI over the serial adapter, not over CAN.

Encoder table and anti-cogging

Firmware 109 measures two corrections on the drive itself, with nothing but the motor on the bench. The **encoder table** corrects the small error of the encoder's angle reading. The **anti-cogging** is a current that cancels the motor's own torque ripple. The same routines run on the sister driver STEPFOC.

After `Calibration success!` of a `#Cal` both run by themselves, and everything is saved when they succeed (What follows the calibration). `#Calonly` is the calibration alone. This page explains what they are, how to run them again, and how to check them.

On the Spectral Micro bench (gimbal motor, 2026-11-10) a full `#Cal` measured the encoder table and identified the anti-cogging, and saved both. Check the result (Checking them) before you keep it.

> **The motor turns**
>
> Each run turns the motor one revolution forward and one back, slowly. Keep the shaft free: nothing on it and nothing it could drive. A mechanism that cannot turn a whole revolution, a gripper's jaws for example, has to come off first.

The encoder table

The magnetic encoder reads the shaft angle with a small error that changes slowly around the turn. The magnet is never exactly on the axis, and the chip itself is not perfectly linear. On the bench gimbal motor it was 0.50° peak to peak.

The STEPFOC bench stepper read 0.61 to 0.65° from run to run. Two of its tables agreed within 0.021°.

The commutation multiplies it by the pole pairs. On an 11-pole-pair motor 0.50° spans about 5.5 electrical degrees. The electrical offset sits near the middle of that span, so the field is a few electrical degrees off at places. That costs torque and makes the current uneven around the turn.

To measure it, the driver turns the field open loop, one shaft revolution forward and one back at 2048 ticks/s (about 20 s). A current of 80 % of the current limit holds the rotor to the field.

At each 1/128 of a turn it records how far the encoder's angle is from the field's. Forward and backward are averaged, so friction, which pulls the rotor behind the field in each direction, cancels. The motor's own pull at higher orders is left out by keeping only the harmonics 1 to 16 per turn.

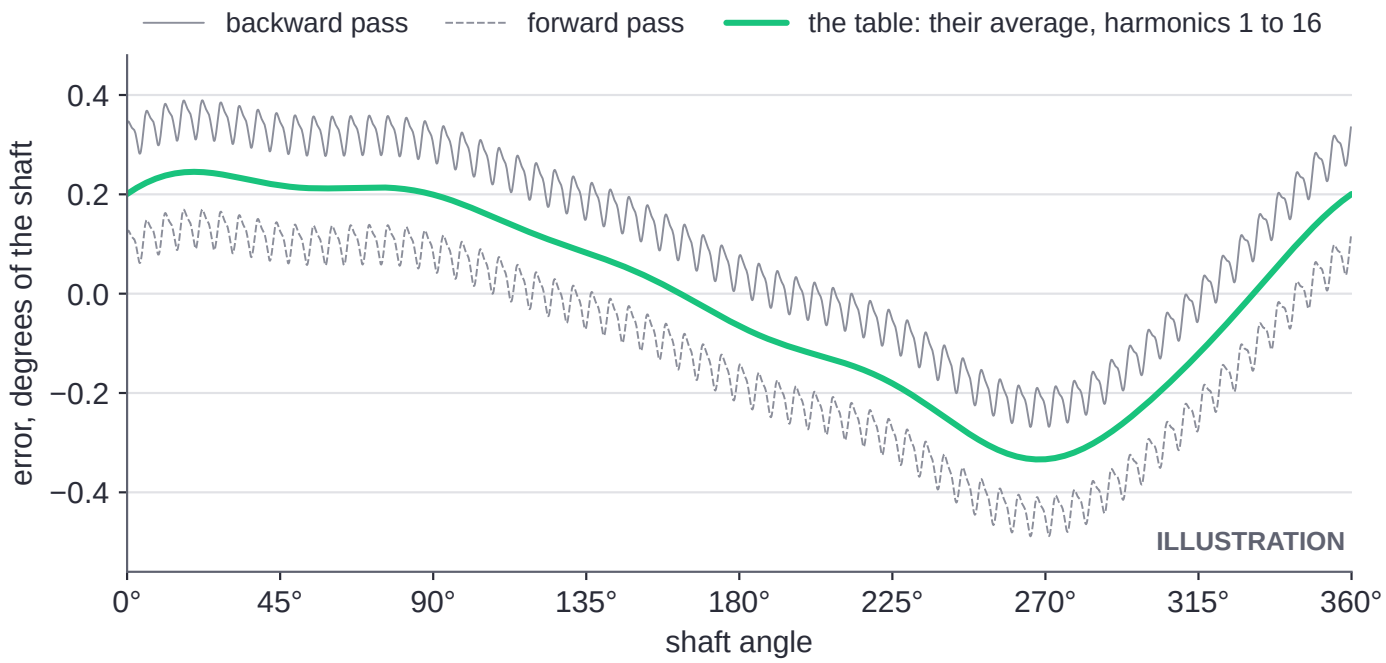


Figure 5.5: Illustration, not a measurement: friction pulls the forward and the backward pass to either side of the encoder's error, and the motor's own pull adds a fast ripple; their average, kept to the harmonics 1 to 16 per turn, is the table.

The result is a table of 128 corrections. From then on the encoder reading is corrected with it. The electrical offset is moved by the table's mean error, so the commutation stays right.

The plot shows a measured table from the sister driver STEPFOC, which runs the same routine, on its bench NEMA 17. Its right-hand scale multiplies by the stepper's 50 pole pairs. On an 11-pole-pair motor the same error is about 4.5 times smaller in electrical degrees.



Figure 5.6: Measured with the STEPFOC on its bench NEMA 17: the encoder's error at each 1/128 of a turn, 0.61° peak to peak, about 31° electrical on the right-hand scale for 50 pole pairs (about 7° on an 11-pole-pair motor); the dashed line is its once-per-turn part.

A check of the routine on a simulated 11-pole-pair motor at the Spectral Micro's loop rate took the commutation error from 30 to 0.4 electrical degrees.

A table over 4° peak to peak, or one that would move the offset by more than 60 electrical degrees, is rejected as implausible: the magnet is loose or far off its place.

Anti-cogging

A motor's magnets pull towards the stator's teeth. Errors in the current measurement add a ripple once and twice per electrical turn.

At low speed both show as a ripple in the speed and the position. Anti-cogging adds a current to the current setpoint that cancels it: a function of the electrical angle, added in velocity and position modes.

To find it, the driver runs in velocity mode at 1500 ticks/s (parameter 44), one motor revolution forward and one back. It records the current the velocity loop asks for in 256 bins over the electrical angle. Then it averages the two directions, each without its mean, so friction drops out.

From that it keeps two forms of the same correction:

- **harmonics** (parameter 37 = 1): the 8 largest of the harmonics 1 to 32 of the electrical angle, each a·cos(h·θ) + b·sin(h·θ) in mA;

- **table** (parameter 37 = 2): 256 values in mA over one electrical turn, interpolated between them.

The plot shows both forms on the STEPFOC's bench NEMA 17. Its cogging current was about the size of the Spectral Micro's bench gimbal motor's (7.1 and 6.8 mA rms); the shape over the turn depends on the motor.

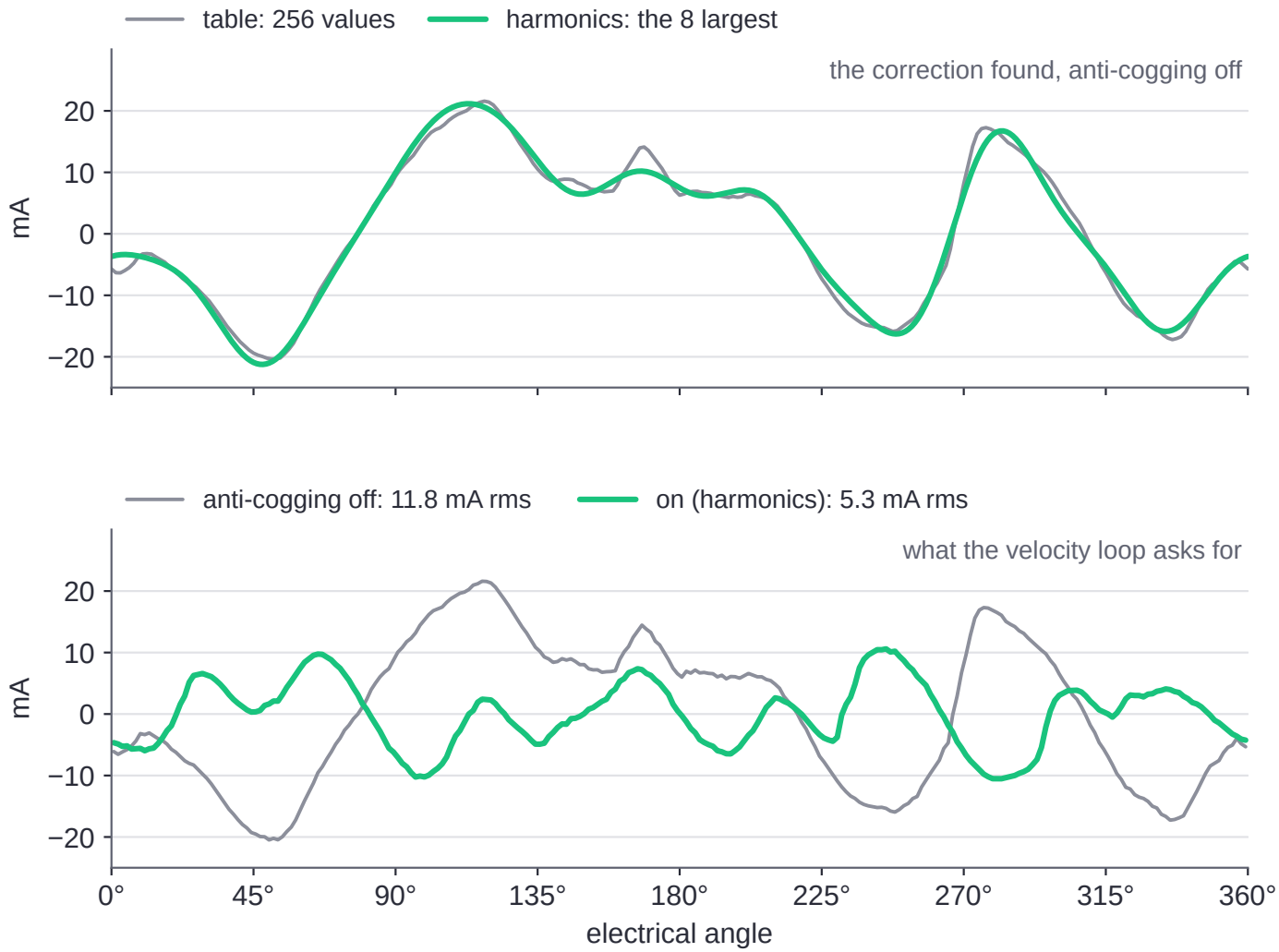


Figure 5.7: Measured with the STEPFOC on its bench NEMA 17: the anti-cogging over one electrical turn, identified at 1500 ticks/s before that drive had the encoder table and the zero trim: above, its two forms; below, the current the velocity loop asks for with it off and on.

The identification switches the harmonics on. On the bench gimbal motor it found 6.8 mA rms. Parameter 38 scales the correction from 0 to 200 % for experiments; 100 % is as identified.

The angle here comes from the encoder alone, without the calibration's offset. The part of the correction that comes from the current measurement is tied to the motor's electrical angle, though, which the offset defines. That is one reason `#Cal` identifies the anti-cogging again.

The identification records the current the loops ask for, so it depends a little on the gains. After a large change of the current or velocity gains, identify it again.

The current-sense zero trim

The STEPFOC firmware can also correct the current-sense zeros from the anti-cogging's once-per-turn part (the **zero trim**, 43 = 4 or 5). The Spectral Micro firmware 109 skips that step until the map of its current channels is measured.

So here `#Cogcal trim` (43 = 4) and 43 = 5 identify like 1 and 2. The state reads 3, never 11 (done without the trim). The combined runs and the runs after a `#Cal` ask for the trim too, with the same result.

Parameters 62 and 63 hold the 1st harmonic before and after a trim, in 0.1 mA. After a run that asked for the trim, both read the 1st harmonic of the set found.

The trims (parameters 59 and 60, up to ±60 counts) change only when written. They are saved with the encoder table and added to the zero the driver measures at every start-up.

Parameter 56 has no effect on the Spectral Micro. It keeps its number so that the two drivers' parameters match. `#Cal` always runs both corrections, `#Calonly` neither.

Running them

The driver must be calibrated and without an error. Each run has a UART command and a CAN parameter:

What	UART	CAN: Send_Parameter
the encoder table, used at once	`#Encal`	52 = 1
the encoder table, used and saved	`#Encal save`	52 = 2
the encoder table, then the anti-cogging, then save all	`#Encal all`	52 = 3, or 43 = 3
the anti-cogging, switched on	`#Cogcal`	43 = 1
the anti-cogging, switched on and saved	`#Cogcal save`	43 = 2
stop a run	`#Encal stop`, `#Cogcal stop`	52 = 0, 43 = 0

`#Encal stop` does not stop the anti-cogging part of a combined run (`#Encal all`). `#Cogcal stop` stops a run in either part, and so do 52 = 0 and 43 = 0 over CAN.

1. Make sure the shaft is free: nothing on it and nothing it could drive.
2. Send the run's command from the table, on the UART or over CAN.
3. Send nothing else while the motor turns, one revolution forward and one back for each run.
4. Read the result with `#Encal state` or `#Cogcal state`, or parameter 52 or 43 over CAN.

> ✓ **Check**
>
> The state reads 3, done. The motor holds the position the shaft reached. Any other state is in the table "When a run fails" under Checking them.

Both runs keep the CAN watchdog fed while they go, and end by holding the position the shaft reached. They stop at once when a host sends a command (a motion command, Idle) or an error comes up. Nothing is saved then.

While a run turns the motor a capture is not armed. During the anti-cogging run parameters 34 and 35 refuse a write: the runs use the capture's memory and the histogram.

A save from a run is the whole configuration, as `#Save` writes it. After a change of the motor, the magnet or its position, run the encoder table and the anti-cogging again, in that order, or calibrate again.

Two guards keep a correction from running on the wrong data:

- parameter 37 refuses 1 or 2 while no set is stored for that form;
- an encoder table measured on its own (52 = 1 or 2) switches an anti-cogging set from before off, because that set was found on the uncorrected encoder. The anti-cogging state then reads 10 until it is identified again.

Checking them

`#Info` ends with what is in force:

```
In force:
  encoder table: on, error 0.61 deg p-p
  anti-cogging: on (harmonics), saved
  current-sense zero trim: 0 / 0 counts
  delay compensation: 234 us
  speed estimate: direct PLL, 100 Hz
  input filter: off; integral deadband: off
```

The example shows the defaults, with an example encoder error. While a run goes, a line `running: encoder table` or `running: anti-cogging identification` follows.

`#Encal state` and `#Cogcal state` print one run's state without starting it. After a finished identification (state 3) it also lists the harmonics. Parameter 43 gives the same state over CAN.

Over CAN the same is in the parameters (the parameter table):

Parameter	What it tells	On the bench gimbal motor
52	the encoder table's state: 3 done, 4 aborted, 5 error or not calibrated, 6 the rotor did not follow, 7 rejected	3
54	the table's error peak to peak, 0.01° of the shaft	50 (0.50°)
55	the amplitude of its once-per-turn part, 0.01° of the shaft	
57	how far the last table moved the electrical offset, 0.1° electrical	
43	the anti-cogging state: 3 done, 6 the shaft did not turn	3
45	the cogging current found, 0.1 mA rms	68 (6.8 mA)
79	the bits of what is in force, see What is in force	

Send_Buffer_Read reads the corrections themselves: kind 4 the encoder table (128 values in 1/16 count), kind 2 the harmonics, kind 3 the anti-cogging table (Respond_Buffer_Data).

A capture of the speed while the motor turns slowly, with the anti-cogging off and on (Status and diagnostics), shows whether it helps on your motor.

On the STEPFOC's bench NEMA 17 the speed ripple that repeats with the angle was 1.2 to 2 times smaller with the anti-cogging on, at 1500 to 6000 ticks/s. The Spectral Micro has not been measured this way yet.

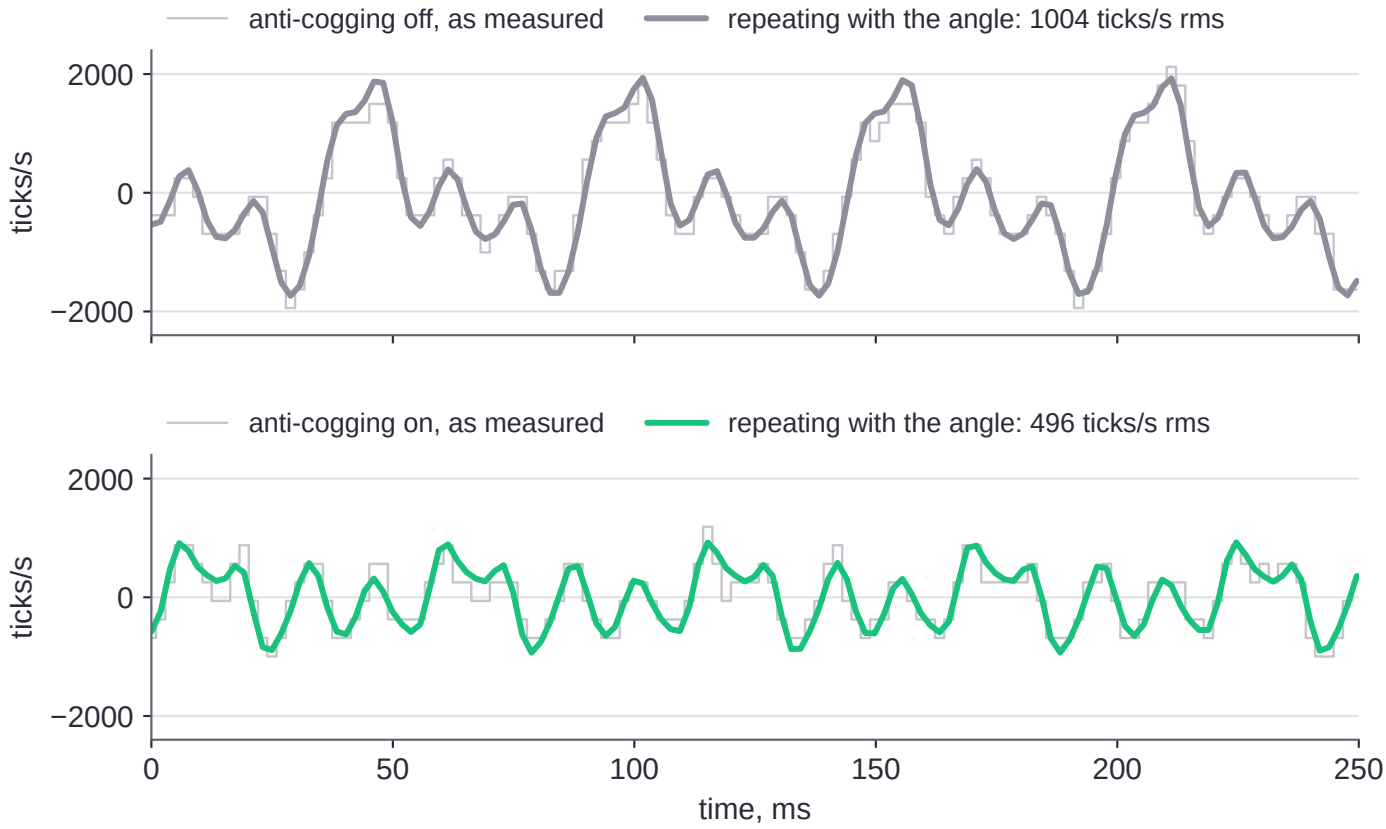


Figure 5.8: Measured with the STEPFOC: the speed of its bench NEMA 17 turning at 6000 ticks/s, minus its mean, before that drive had the encoder table and the zero trim: with the anti-cogging on, the part that repeats with the angle is about half as large.

When a run fails:

State	Cause	What to do
52 = 6, 43 = 6	the rotor did not follow: the shaft is blocked, or the current is too low to turn it	free the shaft; for the encoder table raise the current limit (it uses 80 % of it)
52 = 5, 43 = 5	the driver was in error or is not calibrated	`#Error`, clear it, calibrate
52 = 4, 43 = 4	a command came first	run it again and send nothing meanwhile
52 = 7	the table is implausible	check the magnet: glued, centred, at the right distance (The encoder magnet), then calibrate

State	Cause	What to do
43 = 8	the encoder table of a combined run failed	parameter 52 says why

When a run fails after a `#Cal`, the calibration itself is still good: save it with `#Save`.

Switching them off

	UART	CAN: Send_Parameter
the encoder table off / on	`#Encal off`, `#Encal on`	53 = 0, 53 = 1
the anti-cogging off, harmonics, table	`#Par 37,0`, `#Par 37,1`, `#Par 37,2`	37 = 0, 1, 2
how much of the anti-cogging	`#Par 38,` and the percentage	38 = 0 to 200 (%)

`#Save` keeps the switches with the corrections. `#Default` clears both corrections and the trims.

STEPFOC

The STEPFOC firmware 109 has the same encoder table and anti-cogging, and the current-sense zero trim as well. Its manual says more.

Current limit and loop tuning

Calibration leaves the driver with current loop gains computed for your motor and with conservative velocity and position gains. Two things remain: the current limit, which you must set for your motor, and the loop gains, which you tune if the defaults do not suit it.

Motor GUI shows the motor's response on its plots while you change a gain, which is what makes tuning quick.

The current limit

Set the current limit to the rated phase current of your motor in mA. Calibration uses its own 1600 mA for its steps and, from firmware 109, puts your limit back when it ends (107 and older left 1600 mA: set it again after calibrating there).

The factory default (`#Default`) of 1000 mA suits the gimbal motors in Tested motors; a motor rated for more can go up to the driver's 2800 mA.

`#Ilim` clamps the limit to 2800 mA; from firmware 109 Send_Limits over CAN does too. The firmware does not trip on phase current by itself; the MOSFET driver's hardware protection only catches a hard fault. A limit above the motor's rating overheats the motor.

Motor GUI

In the limits set the current limit, then **Save**.

Terminal

```
#Ilim 1500
#Save
```

Crunching and whining sounds at speed usually mean the current limit, or the current loop gains, are too high.

The loops

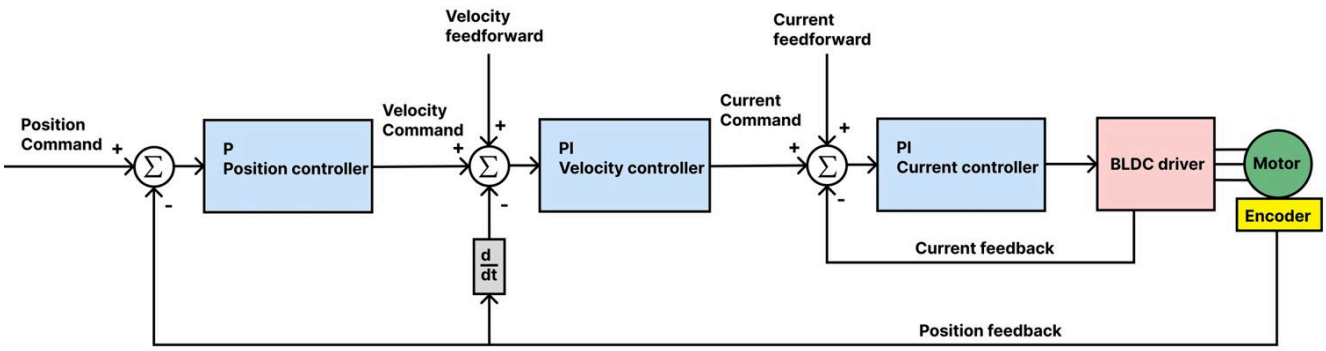


Figure 5.9: The position loop commands the velocity loop, which commands the current loop.

Tune from the inside out: the current loop first, then velocity, then position. Before you start:

- The motor must be calibrated and without active errors; the driver refuses the control modes otherwise.
- Fix the motor to the bench, nothing on the shaft. A loop that oscillates can shake a loose motor off the table.
- If the defaults oscillate, lower all gains until the motor is quiet, then raise them one at a time in small steps.

Motor GUI

The gains are in the settings: change one, *Apply gains*, command a move with the jog controls and read the response on the plots. *Apply preset* loads a starting set for a kind of motor: a product preset writes its own current and velocity gains; the *Generic motor* presets set soft limits and velocity gains and leave the current gains to `#Cbw` and the calibration. Apply a preset before you calibrate, so that the calibration computes the current gains. **Save** when you are done.

Terminal

Each gain is a command: `#Kpiq 0.5`, `#Kpv 0.02`, …, then `#Save`. Command a move with `#Iq`, `#V` or `#P` and judge the response by ear and by `#Cyc 20`, which streams position, speed and current.

The current loop

The current loop is a PI controller that turns a current request into a voltage.

Its gains `Kpiq` and `Kiiq` follow from the motor's resistance R, its inductance L (both measured by calibration) and one number, the bandwidth: Kpiq ≈ ω·L and Kiiq = ω·R·T, with ω = 2π × the bandwidth and T the 200 µs control period (Kiiq is applied once per control loop).

The PI then cancels the coil's own lag, and the current follows a request like a first-order filter at that bandwidth. Calibration computes them this way at 160 Hz, the default from firmware 109; so does setting the bandwidth:

Terminal

```
#Cbw 200          new gains at once from R and L; it prints them
#Save             keeps the gains (the bandwidth itself is 160 again after a power cycle)
```

CAN

Send_Parameter(49, 200), then Send_Save_config. Reading parameter 49 gives the bandwidth the gains in force amount to with the R and L in force.

The bandwidth can be 10 to 500 Hz (a tenth of the 5000 Hz control loop). A higher one follows a request and rejects a disturbance faster, and is louder. Gains set by hand with `#Kpiq` and `#Kiiq` work too; parameter 49 then tells which bandwidth they amount to.

Drives calibrated with older firmware

Firmware 107 and older computed the gains for 500 Hz on the 5000 Hz loop, which is fast for that loop rate. Their saved gains stay as they are after an update to 109: parameter 49 reads about 500 Hz until the drive is calibrated again. Calibrate again after the update where the motor can turn freely. A calibration made with 107 or older stored the resistance about 10 % and the inductance about 45 % too low, so `#Cbw` on the old values gives gains for a lower bandwidth than it names (on an SSG-48 apply its preset instead, which writes the gains). 160 Hz is the default; on the gimbal-motor bench 300 Hz settled a speed step 3.6 times faster at the same holding noise, and 500 Hz added half again the current noise. The SSG-48 preset sets 300 Hz. See Upgrading from older firmware.

Test with a current setpoint of a few hundred mA (*Jog current*, or `#Iq 300`): the motor should push with a steady, quiet torque. Keep the current well under the limit.

The velocity loop

1. Raise `Kpv` until the motor oscillates at a steady speed command, then back off 20 %.
2. Raise `Kiv` slowly until it oscillates, then back off a little. `Kiv` is an integrator: too much makes the motor unstable, none leaves a steady-state error.
3. Test with a speed setpoint (*Jog velocity*, or `#V 20000`; 16384 ticks per second is one revolution per second) and with different speeds.

The position loop

1. Raise `Kpp` until the motor overshoots or oscillates when it reaches a target, then back off.
2. Test with a position step (*Jog position*, or `#P 16384` and `#P 0`): one revolution and back. The response should be quick, without overshoot; the plot shows both.

Testing

Run the motor over the whole range of speeds and loads it will see, with a load if you can, and adjust. Then save.

The speed estimate

The velocity loop needs the speed, which the driver estimates from the encoder count. From firmware 109 parameter 50 picks how:

Parameter 50	Estimate	How it behaves
0	moving average of the last 20 count differences	about 2 ms late, in steps of 250 ticks/s: the estimate of every release before
1	PLL, the speed is its integrator	smooth, a little late
2 (default)	direct PLL, integrator plus the proportional term	almost no lag, a little more noise

Both PLLs are critically damped at the natural frequency of parameter 51, 100 Hz by default (10 to 250 Hz). The lag of the moving average is what limits the velocity gain.

On the gimbal-motor bench the direct PLL at 100 Hz allowed twice the `Kpv` of the moving average before the motor vibrated (0.11 against 0.056), lowered the speed ripple 1.3 to 3.4 times between 1500 and 48000 ticks/s, and cut the overshoot of a speed step from about 67 % to 12 % at the same gains.

At 60 Hz it overshot more, at 200 Hz it held with more current noise. Calibration and the gripper's stop detection always use the moving average. Neither parameter is kept by Save. The speed in Respond_data_pack_1 and Send_Respond_Encoder_data, the watchdog descend and the capture is the picked estimate.

Commutation delay compensation

The voltage a control loop sets reaches the motor later than the angle it was computed from was read: the rest of the loop, the wait for the PWM update and half a control period.

By then the rotor has turned on, and the voltage lands a little behind the angle it was meant for. Parameter 64 turns the output voltage forward by the angle the rotor covers in that delay, in the closed-loop modes above 100 ticks/s; 1 to 500 µs are accepted.

The default is 234 µs, the Spectral Micro's own delay measured on the bench: 20 points between 24000 and 120000 ticks/s gave one delay within 2 to 4 % of theory at every speed.

Without it the field lags about 6° electrical at 120000 ticks/s. The delay belongs to the driver's timing, not to the motor, so it suits any motor; 0 switches it off. It is not kept by Save.

The finer points of the velocity loop

Two settings for special cases, both off by default and not kept by Save; both were tested on the STEPFOC bench:

- **Position input filter**, parameter 74 (Hz, 0 to 200). A host that sends positions at a fixed rate sends a staircase; without the speed the steps are all the loops see. The filter turns the staircase into a smooth path and makes its own speed from it, in position and PD modes. Use it only for a host that sends positions **without** the speed feedforward, at about the host's rate ÷ 12 (8 Hz for 100 Hz); with a host that sends the speed it makes the tracking worse.
- **Velocity-integral deadband**, parameter 75 (ticks, 0 to 2000). In position mode the velocity integral sees the position error less this band, so a holding motor stops pushing against a small error it cannot correct. It trades accuracy for holding current.

Impedance (PD) mode

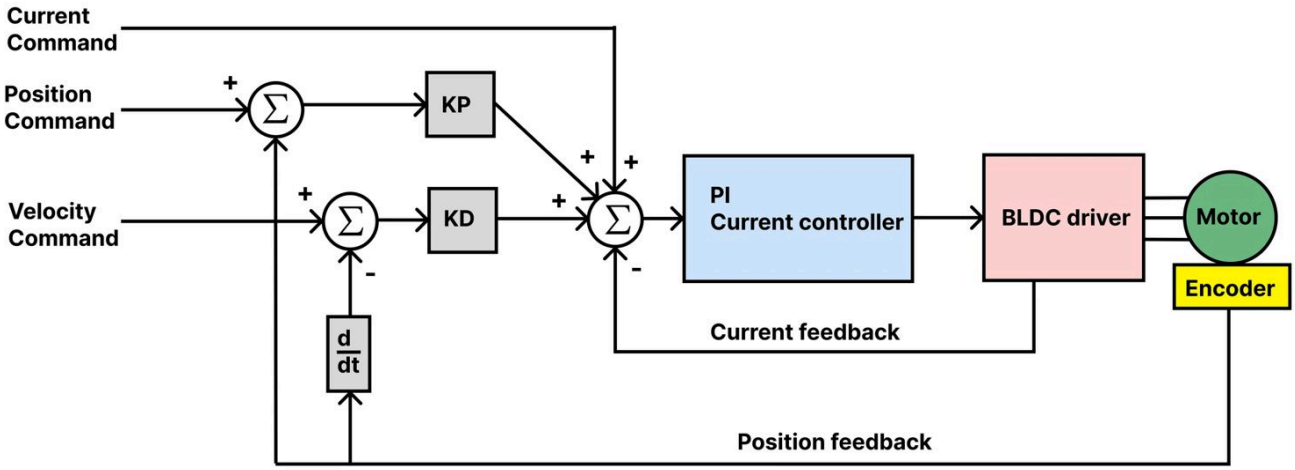


Figure 5.10: The impedance loop: a spring (KP) and a damper (KD) around the position setpoint, plus a current feedforward.

In PD mode the motor behaves like a spring and a damper: the torque is `KP × (setpoint − position) + KD × (speed setpoint − speed) + feedforward current`.

Set `KP` and `KD` (the gains, or `#KP`, `#KD`, or Send_PD_Gains), then command with `#PD`, `#PDV` and `#PDI` or Send_data_pack_PD. Start with a small `KP` and `KD` of 0, raise `KP` until the motor holds position firmly, then add `KD` until it stops ringing. The theory corner explains the loop.

Limits

- The velocity limit (`#Vlimit`) clamps the speed the loops may command, in encoder ticks per second. A rough maximum for your motor is 0.8 × Vbus × KV × 16384 / 60 ticks per second, with KV from calibration.
- The voltage limit (`#Vlim`) clamps the voltage of the current loops in mV; 0 uses the supply voltage.
- `#Rstint 1` resets the integrators on every new setpoint, which avoids windup when a program sends one setpoint after another with large steps.

Saving the configuration

Nothing you set survives a power cycle until you save it.

Motor GUI

Save. The GUI marks *Unsaved changes* until you do.

Terminal

```
#Save
```

Saving writes the configuration to the EEPROM: the motor parameters and the calibration, every gain and limit, the CAN node ID, the heartbeat and watchdog settings, the thermistor and LED settings, the temperature and voltage limits and the gripper flag.

From firmware 109 also the tool id and the watchdog fallback, the anti-cogging set with its mode and scale, and the encoder table with its on/off switch, parameter 56 and the current-sense zero trims. From firmware 110 also the idle behaviour (`#Brake`, Send_Settings or parameter 18).

Over CAN, Send_Save_config does the same. Saving also copies the node ID into the bootloader's board ID (The bootloader). Save after calibration, after tuning, and after changing the node ID.

> **Set the current limit before you save a calibration**
>
> Firmware 107 and older left the current limit at 1600 mA after a calibration, and a save stored it. From firmware 109 calibration puts the limit that was in force back; check that it suits your motor (`#Ilim`) before you save.

What is not saved

These settings are lost at a power cycle, whatever you save. The driver starts with the values in the Defaults reference:

- the calibration settings `#Cbw` (160 Hz; the gains it computed are kept), `#Resv`, `#Calpwr`, `#Openv`, `#Opens`, `#Vp` and `#Iqoffset`: set them again before each calibration that needs other values;
- the parameters of firmware 109 that tune the loops: the speed estimate (the direct PLL at 100 Hz), the commutation delay compensation (234 µs), the sin/cos table (off), the position input filter and the velocity-integral deadband (off), the anti-cogging identification speed, the d-axis current, and the capture and histogram settings. The parameter table says for every parameter whether it is kept;
- the cyclic output (`#Cyc`, `#Cyca`);
- the serial number. Saving writes it, but the firmware does not read it back at power-up: after a power cycle the driver reports 1, and the next save writes that 1 over the stored number.

The defaults

Motor GUI

Default in the settings.

Terminal

```
#Default
```

Restoring the defaults writes the factory values to the EEPROM and loads them. It erases the calibration (the driver reports *Not calibrated*), clears the gripper flag, the heartbeat rate and the watchdog, sets the hardware version to 1 and the batch date to 24092024, and stores 1111 as the serial number.

From firmware 109 it also clears the tool id, the watchdog fallback, the anti-cogging set, the encoder table and its zero trims, and sets parameter 56 back to 0. From firmware 110 it also sets the idle behaviour back to coasting.

The CAN node ID is kept from firmware 109 (it is 0 only when no valid ID is stored); firmware 107 and older set it to 0.

Note what the driver reports before you use it. The values are listed in the Defaults reference.

Use it before the first calibration of a motor, after a firmware update when the driver behaves oddly, and when the state shows nonsense values (a `-1` where a number should be).

The check at power-up

The configuration lives in an EEPROM chip on the board (a 24C16, 2 KB), apart from the firmware: flashing a new firmware never touches it, and only the firmware's check at power-up decides to reset a part of it. Since firmware 106 the saved configuration is in blocks, each with its own checksum; firmware 109 adds three:

Block	Bytes	Checksum at	Holds
calibration	60-188, reserved 204-228	200	the motor's resistance, inductance, pole pairs, phase order, direction, angle offset, Kt, KV and the current loop gains
settings 1	44-180	196	the other gains, the limits, the node ID, the watchdog, the heartbeat and the flags
settings 2 (109)	260-312	256	tool id, watchdog fallback, the product stamp, the calibration ID, from 110 the idle behaviour; reserved slots

Block	Bytes	Checksum at	Holds
anti-cogging (109)	324-879	320	the harmonics, the table, mode and scale
encoder table (109)	884-1147	880	the table, its on/off switch, parameter 56, the current-sense zero trims

At power-up the driver checks each. A block that fails, after a power loss during a save for instance, is replaced by its factory values without a message, and the others are kept.

A failed settings block gives the default gains and limits (from firmware 109 the node ID stays, if it was valid); a failed calibration block leaves the driver *Not calibrated*.

The new blocks need the chip's upper banks; on a smaller chip they keep their defaults in RAM and are not saved, which `#Info` says (`Settings block 2: not available`, `block 3 not available`).

Going back to an older firmware is safe: the first two blocks are unchanged, and older firmware ignores everything from byte 256 on. Upgrading from older firmware says what changes when you update.

A board that comes from a release before 106 has no checksums yet: the driver keeps its values once and seals them; from firmware 109 every board that was ever saved is trusted this way.

A value that was already wrong (a `-1` in `#Info`, for instance) then stays, and `#Default` with a new calibration is the fix. A blank EEPROM gets the defaults.

What the driver reports

- `#Info` prints the identity and the motor: hardware version, batch date, firmware version, CAN ID, whether the motor is calibrated, temperature, supply voltage, the phase order and direction, the pole pairs, resistance, inductance, Kt and KV, and from firmware 109 the build, the board ID and the calibration ID, the tool id, the watchdog, the EEPROM blocks and, at its end, what is in force;
- `#SERNUM` prints the serial number;
- `#Param` the gains and limits;
- `#Error` the active errors.

They are the first thing to look at when something is off, and the thing to copy into a bug report. Status and diagnostics explains the new lines.

6. Operation

Operation

How a calibrated driver is used: the control modes and what each one does, the two ways to talk to the driver and the two tools on top of them, what happens when something goes wrong, and gripper mode.

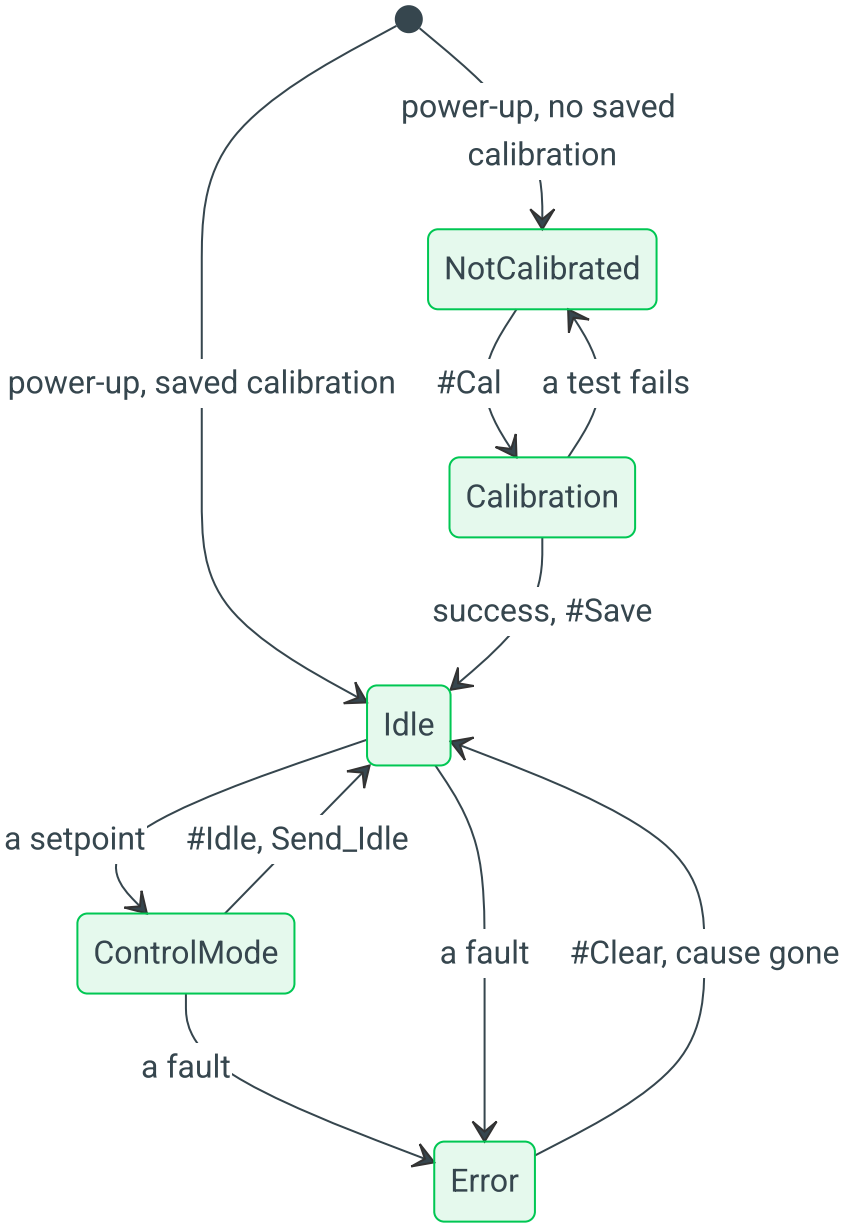
In this chapter

- Control modes
- Over UART
- Over CAN
- The watchdog: what the driver does when the host stops talking
- With Python
- Working in Motor GUI
- Errors and recovery
- Sound: tones, tunes and recorded sound through the motor
- Status and diagnostics: what is in force, the identity, telemetry, captures
- Gripper mode

The exact layout of every message and command is in the API reference.

Control modes

The driver is always in one control mode. A command that carries a setpoint switches to the mode that uses it; `#Idle` or Send_Idle switches back to idle. The driver refuses the closed-loop modes while it is not calibrated or while an error is active.



A setpoint is `#P`, `#V`, `#Iq`, `#PD` or a CAN motion frame, and a new setpoint switches the mode without passing through idle; a fault is any of the errors on Errors and recovery.

ControlMode	What the motor does	Entered by	Setpoint unit
Idle	nothing: coasts (phases open), or brakes (phases shorted) after `#Brake 1` or Send_Settings; with an error it always coasts, see Braking in idle	`#Idle`, Send_Idle, `#Clear`, errors	—
Position	holds and moves to a position with the cascade of position, velocity and current loops	`#P`, Send_data_pack_1 (8 bytes)	encoder ticks
Velocity	turns at a speed with the velocity and current loops	`#V`, Send_data_pack_1 (5 bytes)	encoder ticks/s
Current (torque)	applies a current, so a torque, with the current loop alone	`#Iq`, Send_data_pack_1 (2 bytes)	mA
PD (impedance)	behaves like a spring and a damper around a position, with `KP` and `KD`	`#PD`, Send_data_pack_PD	encoder ticks
Voltage torque	applies a voltage on the q axis directly, without the current loop	`#Uq`	mV
Open loop	turns the stator field at a set electrical speed without the encoder in the loop; the rotor follows it like a stepper. It applies the voltage `#Openv` (5000 mV by default), not a current: `#Ilim` does not apply, and a low-resistance motor draws far more than its limit	`#Openloop`	electrical speed
Gripper	runs the gripper state machine: move to a jaw position with a speed and a current limit	`#Grippos`, Send_gripper_data_pack (with `#Gripper 1`)	0 to 255
Calibration	the calibration routine, or the gripper calibration	`#Cal`, `#Gripcal`	—

Braking in idle

`#Brake 1` (or Send_Settings over CAN) makes the idle motor brake: the driver keeps the three phases shorted, so the motor resists being turned. `#Brake 0`, the default, lets it coast.

From firmware 110 `#Save` keeps the setting; on 109 every power-up started with coasting. Braking works only in idle without an error; an error or an E-stop disables the driver, and the motor coasts.

> **The braking current is not limited**
>
> A motor that is turned while it brakes drives its own back-EMF current through the shorted phases, and the current loop does not limit that current. Use braking only on a motor whose resistance keeps the current under the driver's 2.8 A at the speeds it turns at.

Units

- **Position** is in encoder ticks: 16384 per revolution of the motor shaft, counting up without limit over several revolutions. Positive is counter-clockwise seen from the encoder side of the board, see below.
- **Speed** is in encoder ticks per second: 16384 is one revolution per second, 60 rpm.
- **Current** is in mA. Torque is current times the motor's Kt, which calibration finds.
- **Feedforward**: the 8-byte motion frame carries a speed and a current feedforward beside the position, the 5-byte one a current feedforward beside the speed. They are added after the loops, so a host that knows the trajectory can command it directly and leave the loops to correct the rest. Send 0 when you do not use them.

Try it

The units of this page, with your numbers: 16384 encoder ticks per turn, the practical speed limit from the electrical frequency the control loop can follow (Ratings and limits), and the torque as Kt times the current.

Positive rotation

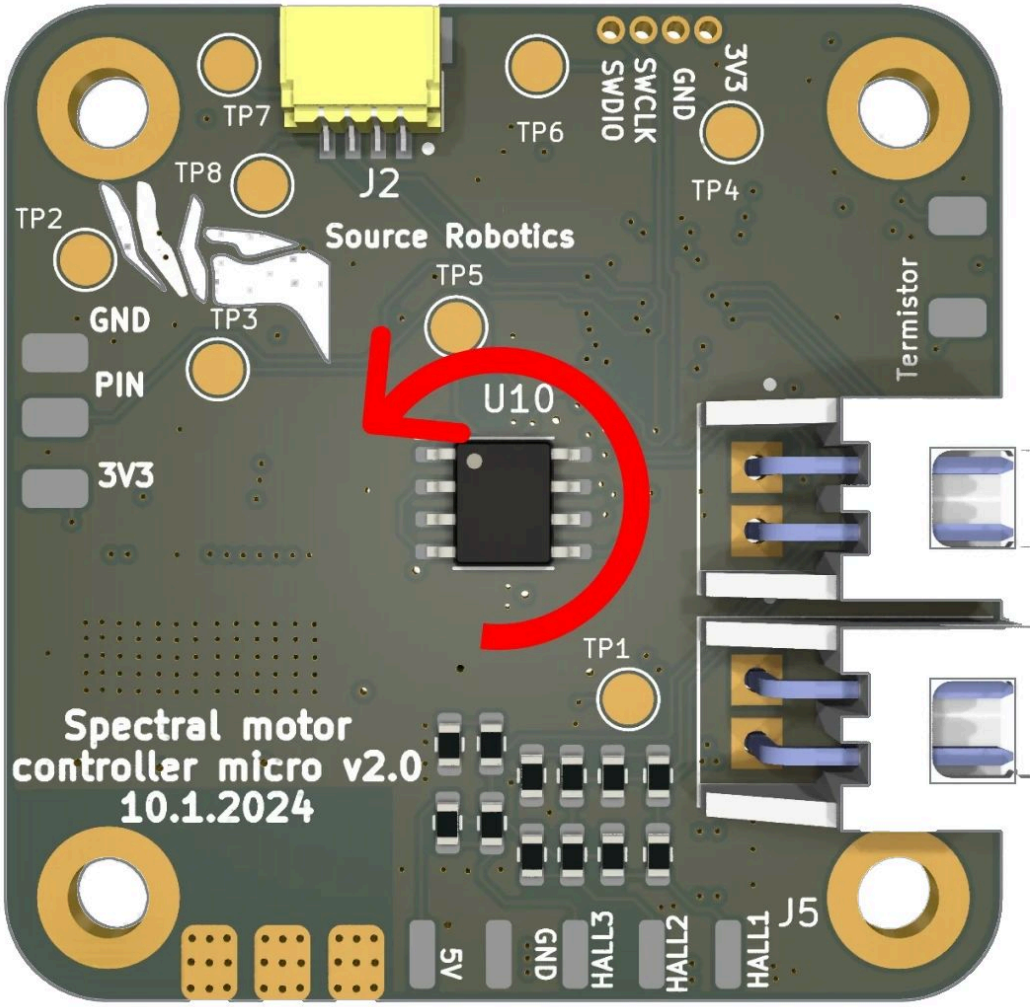


Figure 6.1: Looking at the encoder side of the board, the side that faces the magnet, positive rotation is counter-clockwise. A positive current turns the motor that way, a positive speed too, and the position grows.

The direction is tied to the encoder, not to the wiring: since firmware 105 calibration compensates the order of the phases, so the motor turns the same way whichever wire went to which terminal.

Which mode am I in?

`#Mode` prints the mode: `Idle mode`, `Position control`, `Speed control`, `Current control`, `PD control`, `Open loop`, `Gripper mode` or `Gripper calib mode`. In voltage-torque mode it prints nothing at all; `#Uq` then prints the voltage setpoint.

Over UART

The serial port is the interactive way to use the driver: you type a command, the driver answers in text. It is the only way to calibrate, and the quickest way to try a setting. For real-time control of several motors use CAN.

The protocol

A command is `#`, a name, optionally a space and one number, and a line ending:

```
#P 16384
#P
#Cal
```

- The name is case-sensitive and matches the firmware's spelling exactly (`#Kpiq`, not `#kpiq`); at most 19 characters, the argument at most 19 characters and without spaces (`#WdFb 2000,0,1000,2`). Firmware 107 and older allowed 20 characters for both together.
- The line ends with LF (`\n`); a CR before it is accepted. Terminals send CR+LF when you press Enter, with the setting described in The serial terminal.
- A **Set/Get** command with a number sets the value and prints it; without a number it only prints it. `#P 16384` moves to position 16384 and prints the position; `#P` prints the position. One exception: `#Openloop` without a number starts the motor at the stored open-loop speed.
- An **action** command takes no number: `#Cal`, `#Save`, `#Idle`.
- A name the driver does not know is answered with `Unknown command`; firmware 107 and older answered so also the few names its parser accepted but had no handler for.
- 256000 baud, 8N1, 3.3 V.

The driver only answers

The driver never sends anything on its own, except with the cyclic commands: `#Cyc 100` makes it print position, speed and Iq every 100 ms as `$ <position> <speed> <Iq>` (for BetterSerialPlotter), `#Cyca 100` as `Position:x,Speed:y,Iq:z` (for the Arduino plotter); `#Cyc` or `#Cyca` without a number stops it.

Typical sessions

A first test after calibration:

```
#Ilim 1500        current limit for the motor
#V 16384          one revolution per second
#V 0
#P 0              back to the start
#Idle
```

Tuning:

```
#Kpv 0.02
#Kiv 0.001
#V 30000
#Save
```

Status:

```
#Info
#Error
#Temp
#Vbus
```

Every command is in the UART reference, grouped by what it does.

Over CAN

CAN is the interface for running a machine: compact frames, a fixed cost per message, up to 14 drivers on one bus, and a host that can be a PC, a microcontroller or a robot control box.

The driver answers a motion command with its position, speed and current in the same frame exchange, so a control loop on the host needs one frame out and one frame back per motor and cycle.

The frame

CAN 2.0 data frames with 11-bit identifiers at 1 Mbit/s. The identifier carries the node, the command and the error state:

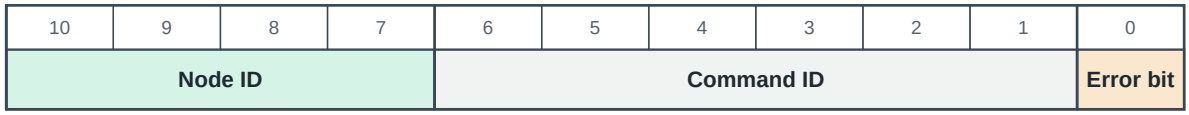


Figure 6.2: Bits 10 to 7 of the identifier are the node ID, bits 6 to 1 the command ID, bit 0 the error bit.

- **Node ID**, bits 10 to 7: which driver, 0 to 15. A driver acts only on frames with its own node ID and sends with it. Set it with `#CANID` or Send_CAN_ID and save; every driver ships as node 0. Use 0 to 13: the bootloader reserves 14 and 15, and a driver on one of those cannot be updated over CAN.
- **Command ID**, bits 6 to 1: which message, 0 to 63. The CAN reference lists all of them.
- **Error bit**, bit 0: in a frame from the driver, 1 while any error is active. In a frame from the host it is ignored.

So a host sends Send_data_pack_1 (command 2) to node 3 with the identifier `(3 << 7) | (2 << 1)` = 0x184, and the driver answers with command 3 under `(3 << 7) | (3 << 1) | error`. A lower node ID wins the bus arbitration.

Multi-byte values are big-endian: the most significant byte is in data[0] and goes first. Floats are IEEE 754 single precision. Flags inside a byte are numbered from the most significant bit.

Three kinds of message

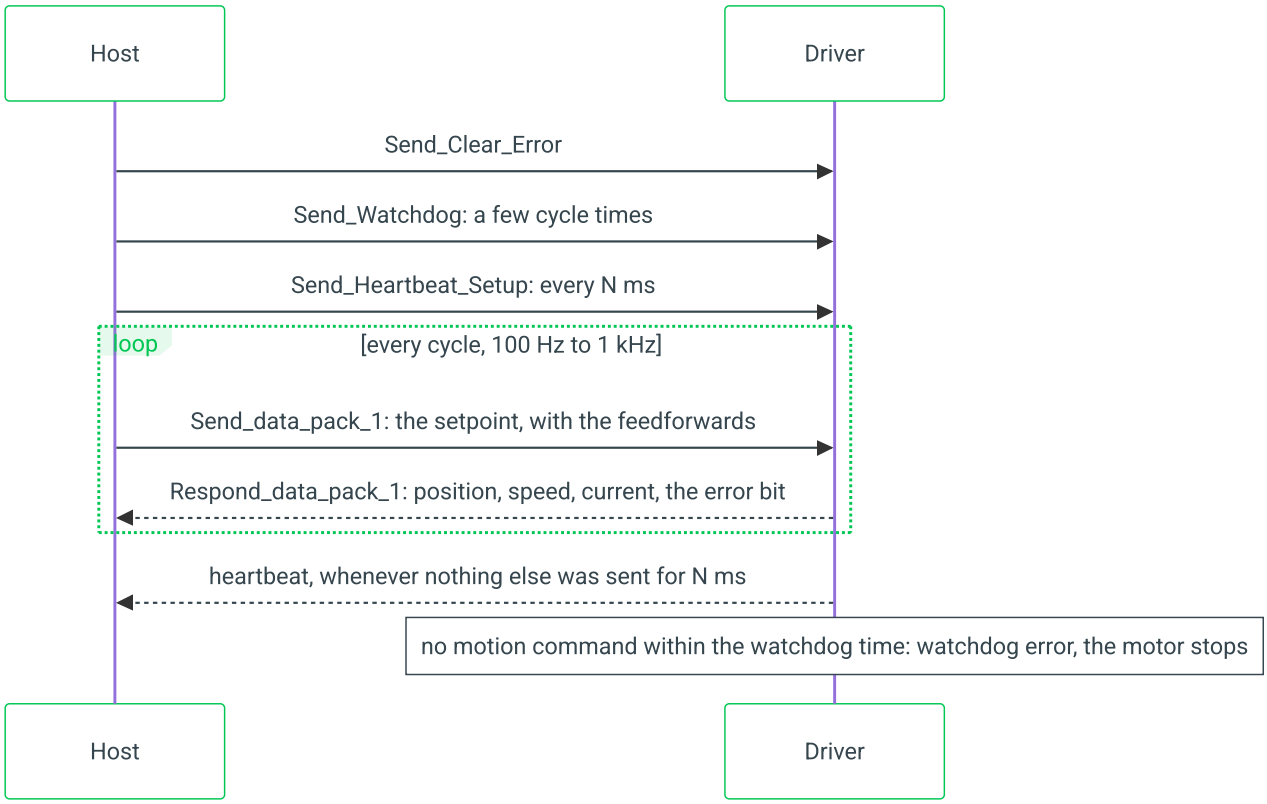
- **Send_…**: the host sends data. Some of these are answered (a motion command with Respond_data_pack_1, the gripper command with Respond_Gripper_data_pack); most are not.
- **Send_Respond_…**: the host asks for data with a **remote frame** (the RTR bit set, no data) carrying the command ID; the driver answers with a data frame with the same command ID.
- **Respond_…**: frames only the driver sends: the answers above, and the heartbeat.

A control loop

Send Send_data_pack_1 at a fixed rate, 100 Hz to 1 kHz, with the setpoint for this cycle; read Respond_data_pack_1 for the position, speed and current of the last cycle.

Every motion command (and every request the driver answers) resets the watchdog, so with Send_Watchdog set to a few cycle times the motor stops when the host stops; from firmware 109 it can also hold its position, brake, coast or let a joint down slowly (The watchdog).

Other frames, such as gains, limits, Send_Parameter or Send_Idle, do not reset it, and neither does anything sent over UART. The heartbeat is the other way round: Send_Heartbeat_Setup makes the driver send a frame on its own every N ms when it has nothing else to send, so the host can see it is alive.



Parameters, identity and telemetry

From firmware 109 a host can also:

- set and read any setting by its number with Send_Parameter and Send_Get_Parameter (IDs 36 and 37, the answer on 38), including the gains and limits in force, which it could not read back before (Parameters and buffers);
- ask a drive what it is with Send_Respond_Identity (ID 39): product, tool, firmware release, pole pairs, whether it is calibrated;
- poll temperature, supply voltage, errors and current in one frame with Send_Respond_Telemetry (ID 40);
- record signals inside the control loop and read them with Send_Buffer_Read (ID 41) (Status and diagnostics).

From a PC

The CANvas adapter shows up as a serial port and speaks slcan; `python-can` opens it with `bustype='slcan'`. The Python library wraps the whole protocol, see With Python. Any CAN adapter that `python-can` supports works the same way.

Several drivers

Give each driver its own node ID before you put them on one bus (a bus with two node 0s cannot tell them apart): connect one driver at a time over UART, `#CANID 3`, `#Save`. Terminate the first and the last node, and wire the daisy chain as in Power and the CAN daisy chain.

The watchdog

The watchdog stops the motor when the host stops talking: a host program that crashes, a cable that comes out, a USB adapter that hangs. It is a timer in the driver.

Every valid motion frame and every request the driver answers starts it again; when it runs out, the driver raises the watchdog error and does what its **action** says. From firmware 109 the action can hold the position, brake, coast, or let a joint down slowly; before, the driver always disabled the motor, which then coasted.

Setting it

Python

```
motor.Send_Watchdog(_rate=100, _action='Hold')   # 100 ms, then hold the position
```

The library knows all five actions from 1.21: 'Idle', 'Hold', 'Brake', 'Coast' and 'Descend'. 1.20 and older know only 'Idle', 'Hold' and 'Brake'.

CAN

Send_Watchdog (ID 15): the time in ms (4 bytes) and the action (1 byte). Or the parameters 6 (time) and 7 (action) with Send_Parameter.

Motor GUI

Watchdog ms in the settings.

0 ms switches it off, the default. Set it to a few of your host's cycle times: with a 100 Hz loop, 30 to 100 ms. Send_Save_config keeps it, and a kept watchdog runs from power-up: a host that does not start sending within the time finds the driver in watchdog error and clears it with Send_Clear_Error.

What starts the timer again: Send_data_pack_1, Send_data_pack_PD, Send_gripper_data_pack, the status requests (Send_Respond_…), Send_Get_Parameter and Send_Buffer_Read. Setting the watchdog time (Send_Watchdog, parameter 6) starts a fresh window, and so does Send_Save_config, whose save takes time.

What does not: the other setting frames (gains, limits, Idle, Clear, Settings, Send_Parameter) and every UART command (known issue KI-11). While the encoder table or the anti-cogging is measured, the driver feeds it itself.

What a trip does

Action	Name	What the driver does
0	legacy	what firmware 107 and older did: the motor driver is disabled and the motor coasts
1	hold	holds the position the motor had at the trip, for the fallback time; then the after-action
2	brake	shorts the phases at once; the braking current is not limited
3	coast	leaves the phases open at once: the motor turns freely
4	descend	applies the fallback current, but never lets the motor turn faster than the fallback speed, for the fallback time; then the after-action

Any other action byte acts as 0. 0 is also what an unchanged host sends: the Python library's default, and what the PAR6 runtime sends.

Three rules keep a gentler action from softening the response to a real fault:

- **Only a trip of the watchdog alone** gets the action. If another error is active as well (E-stop, encoder, supply voltage, temperature, not calibrated, a gripper E-stop), the driver gives the legacy response.
- **Hold and descend take over only from a running motor**: position, velocity, current, PD or gripper mode. A driver that was idle when the watchdog ran out gives the legacy response: the motor coasts, also when it braked in idle (`#Brake 1`). From firmware 110 that holds also when a motion frame comes before the error is cleared; on 109 such a frame started the hold or the descend late, for the whole fallback time.
- **Hold and descend always end** after the fallback time, at most 60 s, so a lost host never leaves a motor driven for good. The after-action follows: 0 legacy, 2 brake or 3 coast.

The driver reports the watchdog error the whole time, and Send_Clear_Error (or `#Clear`) ends the action at once and goes to idle; the next motion frame takes over.

Hold

Hold keeps the position loop on the position the shaft had at the trip. Coming from position or velocity mode it keeps the velocity integral too, so a joint under load does not sag at the moment of the trip.

A host that recovers within the fallback time (a USB hiccup) finds the joint where it left it. On the bench a gimbal motor held within 6 ticks; with a load pulling on its shaft the holding current rose to 440 mA while the shaft crept 59 ticks.

Descend

Descend applies the fallback current and limits the speed in both directions to the fallback speed. Inside the cap the speed limiter's integral relaxes back to zero (within about 0.25 s), so it holds back only what keeps the speed at the cap and never more:

- a joint under gravity with 0 mA lowers itself at no more than the cap speed and rests on whatever stops it;
- a gripper with its gripping current keeps its grip, and its jaws cannot slam shut;
- a free motor with a current turns at about the cap speed: on the bench a gimbal motor with 200 mA and a 1000 ticks/s cap turned at 1145 to 1163 ticks/s, about 15 % over the cap, which is soft because the limiter's integral relaxes inside it.

It starts with the velocity integral at zero, so gravity can take the joint down.

The fallback

Four values, kept by Send_Save_config in settings block 2:

Parameter	`#WdFb` position	What	Default
2	1st	how long hold or descend runs, ms (0 to 60000)	2000
3	2nd	the current descend applies, mA (signed)	0
4	3rd	the speed cap of descend, ticks/s (0 to 65535)	1000
5	4th	the after-action: 0 legacy, 2 brake, 3 coast	0

Terminal

```
#WdFb 2000,0,1000,2
#Save
```

Four numbers, commas, no spaces. `#WdFb` alone prints them, and `#Info` prints the watchdog with them.

Python

```
motor.Send_Watchdog_Fallback(2000, current_mA=0, speed=1000, after='Brake')
motor.Send_Save_config()
```

Library 1.21 and newer.

Choosing an action

- **A bare motor on the bench, a conveyor, a wheel**: brake (2) stops it fastest; coast (3) or legacy (0) lets it run out.
- **A joint that carries a load**: hold (1) for a second or two, so a short loss of the host does not move it, then descend (4) with 0 mA for a joint that gravity pulls down, or brake for one it does not. Choose the cap speed for a joint you can still catch by hand.
- **A gripper**: descend with the gripping current, its sign the closing direction, keeps an object held.

These suggestions are untested on an assembled robot arm; try the action on your machine before you rely on it. Brake is not current-limited (known issue KI-12): a load that drives a braking motor fast drives a large current through the phases; try it with the motor still first.

Seeing a trip

A capture armed with the error trigger (parameter 28 = 2) records the moments before and after a trip, the watchdog's included: Status and diagnostics.

With Python

The `Spectral-BLDC` library wraps the CAN protocol: one method per message, and the driver's answers unpacked into attributes. It runs on Windows, Linux and macOS with any adapter `python-can` supports, such as the CANvas adapter over slcan.

Installing

In a virtual environment:

Windows

```
py -m venv .venv
.venv\Scripts\activate
pip install Spectral-BLDC
```

The CANvas adapter is `COM` and a number (Device Manager, **Ports (COM & LPT)**): `channel='COM41'`.

Linux

```
python3 -m venv .venv
source .venv/bin/activate
pip install Spectral-BLDC
sudo usermod -aG dialout $USER     # access to the serial port; log out and in again once
```

The CANvas adapter is `/dev/ttyACM0` (`ls /dev/ttyACM*` lists it): `channel='/dev/ttyACM0'`.

macOS

```
python3 -m venv .venv
source .venv/bin/activate
pip install Spectral-BLDC
```

The CANvas adapter is `/dev/cu.usbmodem` and a number (`ls /dev/cu.usbmodem*` lists it): `channel='/dev/cu.usbmodem1101'`, with your number.

Check

`python -c "import Spectral_BLDC"` returns without an error message.

The source and the examples are in Spectral-BLDC-Python. For joints with a gear ratio and an offset, the Source Robotics Toolbox (`pip install SourceRoboticsToolbox`) turns encoder ticks into joint angles.

The first program

```
import Spectral_BLDC as Spectral
import time

# the adapter: a CANvas on COM41 (Windows) or /dev/ttyACM0 (Linux), 1 Mbit/s
comms = Spectral.CanCommunication(bustype='slcan', channel='COM41', bitrate=1000000)
# the driver with node ID 0
motor = Spectral.SpectralCAN(node_id=0, communication=comms)

motor.Send_Clear_Error()

# A configured, calibrated motor with a clear shaft: command one turn for five seconds.
# Each command requests a fresh position/speed/current reply; heartbeat is not required.
try:
    deadline = time.monotonic() + 5.0
    next_cycle = time.monotonic()
    while time.monotonic() < deadline:
        motor.Send_data_pack_1(Position=16384, Speed=0, Current=0)
        reply_deadline = min(deadline, time.monotonic() + 0.02)
        while time.monotonic() < reply_deadline:
            message, ids = comms.receive_can_messages(
                timeout=max(0.0, reply_deadline - time.monotonic()))
            if message is None:
                print("No motion feedback in this cycle; check the bus and node ID")
                break
            if ids.node_id != 0 or ids.command_id != 3:  # Respond_data_pack_1
                continue
            motor.UnpackData(message, ids)
            print(motor.position, motor.speed, motor.current, ids.error_bit)
            if ids.error_bit:
                raise RuntimeError("Driver fault: inspect State_of_Errors before retrying")
            break
        next_cycle += 0.02
        time.sleep(max(0.0, next_cycle - time.monotonic()))
finally:
    motor.Send_Idle()                    # also on Ctrl-C; the shaft no longer holds torque
```

`receive_can_messages` returns the raw `python-can` message and a tuple with the node ID, the command ID and the error bit of the identifier; `UnpackData` decodes the message into the attributes the message carries.

This example resends the same setpoint at 50 Hz for five seconds, prints only matching motion feedback, and puts the motor in idle when it ends. Idle releases torque: support any attached mechanism before running it. A missing reply is reported without claiming that a move finished. Configure and calibrate the driver first (Quick start).

Check

The program prints a line of position, speed, current and error bit for each cycle, the position moves to 16384 and the error bit stays 0. `No motion feedback in this cycle` means no answer came back: the port, the node ID and the bus, see Over CAN.

What is where

Library call	Message	What you get back
`Send_data_pack_1(Position, Speed, Current)`	Send_data_pack_1	`position`, `speed`, `current`
`Send_data_pack_PD(...)`	Send_data_pack_PD	`position`, `speed`, `current`
`Send_Respond_Encoder_data()`	Send_Respond_Encoder_data	`position`, `speed` (32-bit)
`Send_Respond_State_of_Errors()`	Send_Respond_State_of_Errors	the error attributes
`Send_gripper_data_pack(...)`	Send_gripper_data_pack	`gripper_position`, `gripper_current`, the gripper state bits
`Send_Limits(...)`, `Send_Velocity_Gains(...)`, ...	the setting messages	nothing; `Send_Save_config()` keeps them

The complete map of calls to messages is in the Python library reference, and each message's bytes in the CAN reference.

Several motors

One `CanCommunication` per bus, one `SpectralCAN` per driver with its node ID. Send to each, then read the bus: every answer carries the node ID in `ids.node_id`, so route it to the right object.

```
motors = [Spectral.SpectralCAN(node_id=i, communication=comms) for i in range(3)]
for m in motors:
    m.Send_Respond_Encoder_data()
for _ in motors:
    message, ids = comms.receive_can_messages(timeout=0.1)
    if message is not None:
        motors[ids.node_id].UnpackData(message, ids)
```

The Guides build on this: a quadruped leg, gravity compensation, bilateral teleoperation.

Working in Motor GUI

Everyday use of the driver from Motor GUI: moving the motor, watching it, recording what it does, running several motors, and the few things that only work from the console.

Moving the motor

The jog controls send setpoints in the matching control mode: *Jog velocity* turns the motor at a speed in encoder ticks per second (16384 is one revolution per second), *Jog position* moves it to a position in ticks, *Jog current* applies a torque in mA.

Send setpoint sends a value in the mode the driver is in. Idle stops the motor; Control modes says what each mode does. With *Live stream while dragging* the setpoint follows the slider, which is the quickest way to feel a tuning change.

Watching it

The plots show position, speed and current as the driver reports them. Pan and zoom to a move; hover for the value. A step in the position setpoint shows overshoot and settling at once, a speed step shows the velocity loop's response; Current limit and loop tuning uses them step by step.

Recording

A recording keeps the telemetry stream to a file: *New recording*, run the move, *Stop recording*. Record the response before and after a tuning change to compare them, or a failure to attach to a bug report (Report a problem).

Clearing errors

The errors the driver reports are listed with the state; **Clear errors** sends what `#Clear` does. An error whose cause is still present comes straight back; Errors and recovery says what each one needs.

Several motors

Over the CANvas adapter the GUI sees every driver on the bus and addresses each by its node ID.

Give each driver its own ID over the serial adapter first (the CAN node ID in the settings, then Save), terminate the first and the last node, and set a watchdog (*Watchdog ms*) on a bus that drives something that must not run away: Power and the CAN daisy chain, Over CAN.

Gripper

A driver in gripper mode shows the gripper controls: calibrate the jaws, a position from open to closed, a speed, a current limit, and the state the gripper reports. *No gripper connection* means the driver is not in gripper mode or not on the bus. Gripper mode.

Firmware updates

Firmware update lists the releases from GitHub and flashes one through the bootloader; on a bus, *Scan bootloaders* finds the boards. Updating through the bootloader.

The console

Anything the GUI has no control for is one `#` command away in the console, with the driver's answer below it; the whole set is in the UART commands reference. The cyclic output (`#Cyc`) is what the GUI's telemetry stream uses over serial, so do not start it by hand while the plots run.

Reading a fault

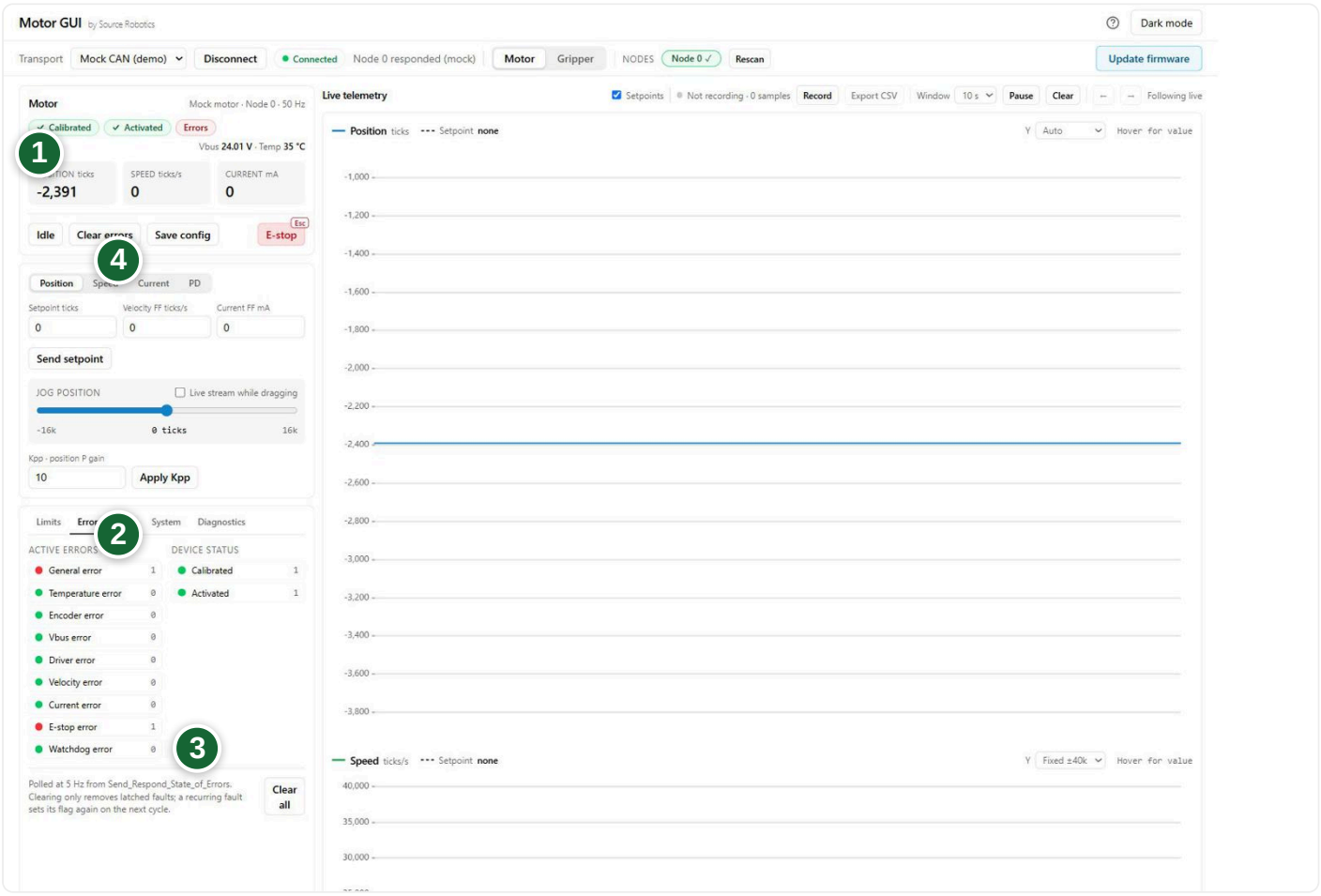


Figure 6.3: **1. Errors state:** a fault is active. **2. Errors tab:** inspect individual flags. **3. E-stop error:** 1 identifies the latched fault in this simulation. **4. Clear errors:** clear it after removing its cause; a persistent cause returns on the next update. Screenshot captured from **Mock CAN (demo)**.

Check

After recovery the fault flag returns to 0 and the state shows *No errors*. If it returns, stop and follow Errors and recovery.

Errors and recovery

When the driver detects a fault it disables the motor, goes to idle, turns the status LED on steadily and sets the error bit in every CAN frame it sends. A disabled motor coasts, also when `#Brake 1` is set. The driver stays there until the error is cleared.

Seeing the error

- **The LED**: on and not flashing means error mode (or not calibrated). Three short flashes and a pause is normal operation; fast flashing, about 10 times a second, is calibration.
- `#Error` prints `In error state!` or `No errors!`, followed by one line per active error.
- **Send_Respond_State_of_Errors** returns every flag as a bit; **the error bit** of any frame says whether there is one.

Normal operation	Calibration	Error mode
		

In error mode the LED does not flash: it stays on, as in the photo on the right.

The errors

Error	`#Error` line	Cause	What to do
Vbus	`Vbus error`	the supply voltage is below 9 V or above `#Vmax` (29 V after `#Default`). From firmware 109 a low voltage is not checked for the first 500 ms after power-up, while the supply ramps up	check the supply and the cable; a decelerating motor can push the voltage up

Error	`#Error` line	Cause	What to do
Encoder	`Encoder error`	the magnet is not detected or the reading is invalid	magnet distance and type, see The encoder magnet
Driver	`Drv error`, and `FAULT pin is LOW: the gate driver reports a fault` while the fault lasts (firmware 107 and older printed `FAULT pin is HIGH`)	the MOSFET driver's own protection tripped: undervoltage, thermal shutdown or overcurrent	lower the current limit, cool the driver, check for a short between the motor leads
Velocity	`Velocity error`	the measured speed exceeded 20,000,000 ticks/s (about 1220 revolutions per second), a fixed limit that `#Vlimit` does not change	a runaway or an encoder fault: check the magnet and the encoder, see The encoder magnet
Temperature	`Temp error`	the thermistor reads more than `#Tlim` (75 °C by default) or less than −20 °C, with `#Term 1`	let the motor cool; lower the current limit
E-stop	`Received ESTOP command`	Send_ESTOP was received, or from firmware 110 `#Estop` over UART	clear it when it is safe
Watchdog	`Watchdog error`	no motion command or answered request over CAN within the watchdog time (Send_Watchdog); what the motor does then is the watchdog's action, see The watchdog	the host stopped sending; fix the host, then clear. See The watchdog
Wrong product	`Wrong firmware for this board (product lock)`	from firmware 109: this board was calibrated by the other product (a STEPFOC firmware on a Spectral Micro, or the other way round), so the firmware drives nothing (The product lock)	flash the right firmware; it does not clear otherwise
Not calibrated	`Not calibrated`	the motor has not been calibrated since `#Default`, or the EEPROM is invalid	run `#Cal` and `#Save`

Error	`#Error` line	Cause	What to do
Current	`Current error`	reserved; this firmware never raises it	—

Clearing

`#Clear` or Send_Clear_Error clears the flags and goes to idle. An error whose cause is still present comes straight back: a Vbus error at 2 V stays until the supply is right; once the supply is 24 V, one `#Clear` returns the driver to normal operation.

The calibration error clears only by calibrating (or, for a motor you know is calibrated, `#Calibrated 1`).

The watchdog

The watchdog is set with Send_Watchdog over CAN, or with parameters 6 (time) and 7 (action): over UART, `#Par 6,<ms>` and `#Par 7,<action>`. It counts the motion commands (Send_data_pack_1, Send_data_pack_PD, Send_gripper_data_pack) and the requests the driver answers (the Send_Respond_ remote frames, Send_Get_Parameter, Send_Buffer_Read).

Other CAN frames and UART commands do not reset it. Send_Save_config keeps it over a power cycle, like the heartbeat interval. What the driver does when it runs out is the watchdog's action (The watchdog).

A driver with a saved watchdog that is used over UART only, or by a host that sends no motion command, raises a watchdog error each time the watchdog time runs out, also right after `#Clear`.

To turn it off, send Send_Watchdog with 0 and then Send_Save_config over CAN. Over UART, `#Par 6,0` and then `#Save` do the same; nothing else is changed.

When the driver does not answer

- No LED at all: no power. Check DC+ and DC− and the cable orientation.
- LED on, no answer on UART: the baud rate (256000), TX and RX crossed, the adapter's port.
- No answer on CAN: the node ID, the termination (about 60 Ω across the bus, unpowered), 1 Mbit/s on the adapter.
- `#Info` shows `-1` or nonsense: the EEPROM holds values the firmware does not expect, usually from an older release (see Saving the configuration); `#Default`, then calibrate again.

Troubleshooting goes through the symptoms one by one.

Status and diagnostics

What the driver tells about itself, from firmware 109: what is in force, who it is, its state in one frame, how long its control loop takes, and recordings of its own signals made inside the control loop. The error flags and what to do about each are on Errors and recovery.

What is in force

`#Info` ends with the section `In force:`, and parameter 79 holds the same as bits (What is in force: parameter 79):

Line	Says
`encoder table:`	`on, error … deg p-p`, `measured, off` or `not measured` (Encoder table and anti-cogging)
`anti-cogging:`	`on (harmonics)`, `on (table)`, `identified, off` or `not identified`, and `, saved` when it is the saved set
`current-sense zero trim:`	the two trims in counts, and whether the zero measured with the bridge switching is in use
`delay compensation:`	the commutation delay in µs, or `off` (Commutation delay compensation)
`speed estimate:`	`moving average`, `PLL, … Hz` or `direct PLL, … Hz` (The speed estimate)
`input filter:`, `integral deadband:`	parameters 74 and 75, or `off`
`running:`	only while one runs: `calibration`, `encoder table`, `anti-cogging identification`, or `the runs after the calibration`

Before that section `#Info` prints, beside what it printed before:

- `Build:` the git commit the firmware was built from, 7 hex digits, with a `+` when it was built with uncommitted changes. `Board ID:` this board, 8 hex digits made from the chip's unique number; it never changes, and it is not the bootloader's board ID (0 to 13). `Calibration ID:` a new number at every successful calibration, kept by Save; `00000000` before the first. Over CAN they are parameters 84, 86 and 87; parameter 85 is the build time in Unix seconds.
- `CAN RX overruns` and `CAN TX dropped`: frames the driver lost because its receive buffer was full, and answers it gave up after waiting 1 ms for a free transmit mailbox. Both should stay 0; they count from power-up. From

firmware 109 the CAN controller passes only the frames addressed to the driver's own node ID, and the driver sends from all three transmit mailboxes, so a busy bus costs it less.

- `Product:` this firmware, Spectral Micro.
- `Current sense zero:` the zeros in use, in ADC counts from 2048, measured at start-up with the bridge off (one count is about 1.6 mA), and what the second measurement with the bridge switching found. On the bench the zeros were −8 and −7 counts, and the current read 3.9 mA with nothing flowing, where it read −20 to +19 mA before. A zero more than 200 counts from 2048 is not trusted: the driver then uses 2048 and says `not measured or out of range`. The current sensors need the motor supply. If the driver starts without it (USB plugged in before the supply), it waits up to half a second for the supply, then uses 2048 and says `no motor supply at start-up: 2048 used; measured once the supply is there`. When the supply comes, the driver measures the zero by itself at rest, coasting for about 100 ms even if it is set to brake, and the line shows the new zeros. The Vbus error the late supply latched stays until `#Clear`.
- `Tool ID`, `Watchdog: … ms, action …, fallback …`, `Settings block 2: ok` and `Anti-cogging: …, scale … %` with whether it is saved.

Who the drive is: the identity frame

Send_Respond_Identity (ID 39, a remote frame) answers with the product (1 STEPFOC, 2 Spectral Micro), the tool id, the firmware release, the hardware version, the pole pairs, whether it is calibrated, and eight flags: set up as a gripper, brakes in idle, coils reversed, settings block 2 available, wrong product, current-sense zero measured, anti-cogging saved, switching zero in use (Send_Respond_Identity).

A host can read it before it commands or flashes a drive, to know it talks to the drive it expects.

The **tool id** says what the drive is built into:

- 0: none (a joint or a bare driver);
- 1: an SSG-48;
- 11 to 33: an MSG (the tens digit the rail, 1 = 100, 2 = 150, 3 = 200 mm; the units digit the stepper, 1 = 21.5, 2 = 40, 3 = 60 mm);
- 200 to 254: your own tools.

Set it with `#Tool` or parameter 1 and save.

Everything at once: telemetry

Send_Respond_Telemetry (ID 40, a remote frame) answers with the temperature, the supply voltage, both error bytes and the current, all from the same moment: one request instead of four, for a host that watches many drives (Send_Respond_Telemetry).

The product lock

STEPFOC and the Spectral Micro use the same bootloader and the same file name for their firmware, and each firmware drives the other board wrongly: the Spectral Micro firmware on a STEPFOC drove a coil at about 1.7 A in every mode on the bench.

From firmware 109 a board remembers which product calibrated it, a stamp in the EEPROM written at the first start-up of a calibrated board and at every save.

A firmware that finds the other product's stamp, or no stamp but a calibration its own product cannot make (the Spectral Micro firmware treats 50 pole pairs or more as a STEPFOC calibration), reads and writes nothing more, drives no pin (its gate driver is enabled only after the check) and starts no control loop.

It stays reachable: it reports an error, `Wrong firmware for this board (product lock)` in `#Error`, the `wrong_product` bit in the error state and the identity, and a warning in `#Info`. Flash the right firmware and the board runs as before; nothing was written.

`#Product` shows the stamp. On a board the rule misjudges, `#Product claim` stamps it as this product and `#Product clear` removes the stamp; power-cycle after either. Both are UART commands only.

How long the control loop takes

The control loop runs every 200 µs (5000 Hz) and has to finish inside that. `#Info` prints the last loop's time. On the bench an early build of firmware 109 took about 80 µs of the 200; anti-cogging and the commutation delay compensation each add some 10 µs (measured on STEPFOC).

The section timing and the register fast paths of the STEPFOC firmware (its parameters 66 to 72) are not in the Spectral Micro firmware.

A capture

The capture records three signals inside the control loop, sample by sample, 1024 samples per channel: what the loop saw and did, which a host polling over the bus cannot see. Parameters 24 to 33 set it up and report on it, Send_Buffer_Read with kind 0 reads it (Reading the capture; the signals are listed under Capture and histogram).

A step response, 1024 samples of speed, current and position error, one every 2 loops (0.41 s):

```
import time
import Spectral_BLDC as Spectral

comms = Spectral.CanCommunication(bustype='slcan', channel='COM41', bitrate=1000000)
motor = Spectral.SpectralCAN(node_id=0, communication=comms)

def read_all(seconds):
    end = time.time() + seconds
    while time.time() < end:
        message, ids = comms.receive_can_messages(timeout=0.01)
        if message is not None:
            motor.UnpackData(message, ids)

for number, value in ((24, 4), (25, 1), (26, 7), (27, 2), (28, 1)):
    motor.Send_Parameter(number, value)     # A speed, B Iq, C position error; every 2nd loop;
start now
motor.Send_data_pack_1(1000, 0, 0)          # the step: to position 1000
read_all(0.5)
motor.Send_Get_Parameter(32)                # how many samples
read_all(0.1)
count = motor.parameters.get(32, 0)
motor.Send_Buffer_Read(0, 0, count)         # kind 0, from sample 0
read_all(3.0)
for i in sorted(motor.capture):
    speed, iq, position_error = motor.capture[i]
    print(i, speed, iq, position_error)
```

Trigger 2 records up to an error instead: armed with a pre-trigger, it keeps the moments before a trip, the watchdog's included, and the time after it. The capture calls (`Send_Parameter` , `Send_Get_Parameter` , `Send_Buffer_Read`) need the Spectral-BLDC library 1.21 or newer.

Memory

The drive's microcontroller has 128 KB of flash, of which the bootloader takes 12 KB, and 20 KB of RAM. Firmware 109 uses about 91.5 % of the 116 KB application flash and 65 % of the RAM for fixed data, which leaves about 7 KB for the stack and the heap.

Parameter 83 (`#Par 83` , or Send_Get_Parameter(83)) is the margin check: the bytes of that RAM the stack has never reached since start-up. The stack needs about 1.5 to 2 KB; read parameter 83 after the drive has run its hardest work (a calibration, the encoder table, the anti-cogging, sound) and see that plenty is left.

A histogram

The histogram adds one signal into 256 bins over the electrical angle or over one motor turn, every control loop, until it is stopped or a bin holds 65535 samples (about an hour): the mean per bin shows what repeats with the angle, such as the cogging current at a steady speed.

Parameter 35 picks the signal, 34 starts it (1 electrical angle, 2 motor turn) and stops it (0), 36 counts the samples, and Send_Buffer_Read with kind 1 reads the count and the sum of each bin. The anti-cogging identification uses it the same way.

Sound

From firmware 109 the driver plays sound through its motor: tones, chords, tunes from a queue it plays by itself, and recorded sound such as speech. A current at an audio frequency in the windings makes a small force at that frequency, and the motor, and above all whatever it is fixed to, radiate it as sound.

How it works

- **Where:** the driver adds the sound to its output voltage, after the current loop. By default it goes on the q axis (parameter 82 = 1), where it makes torque: the rotor rocks at the tone's pitch. On the bench that was much louder than the d axis (82 = 0), which makes no torque at all.
- **When:** only while the driver holds or moves: position, velocity, current or PD mode. Idle, or any error, stops all sound. A joint keeps holding its pose while it plays.
- **Tones:** up to 4 at once (channels 0 to 3), a chord on one motor, each with its own pitch, loudness and length.
- **Loudness** is given as a current, in mA. The coil's impedance rises with the pitch ($\sqrt{(R^2 + (2\pi fL)^2)}$, from the motor's R and L), so the same voltage would sound much quieter high than low; the driver turns the current into the voltage that pitch needs, and a note is as loud low as high.
- **Pitch:** 20 Hz up to just below 2500 Hz, half the 5000 Hz control loop. Tones are clean up to about a third of the loop rate; melodies sit well between C4 and C6 (262 to 1047 Hz).
- **The queue:** 31 events, each setting one channel's pitch and loudness and then waiting. The control loop plays them with loop-exact timing, so the host's own timing never reaches the music: it loads the queue and starts it.
- **Samples:** recorded sound, 8 bits, played from a 1024-place buffer at a rate the host sets, up to the loop rate. Speech comes through recognisable but muffled: the telephone band reaches 3400 Hz, the motor 2.5 kHz.
- **The cap:** all sound together is limited to the voltage of parameter 81, 2000 mV by default; 0 refuses sound. On a gimbal motor's high resistance the cap comes early: 250 mA in a 7 Ω coil already takes 1.75 V. None of the sound settings is saved.

A tone over UART

```
#Tone 440,250,1000      440 Hz, 250 mA, 1000 ms
#Tone 880,200,0         until stopped
#Tone off               stops all sound
```

From idle, a calibrated driver first holds the position it is at and says so (`Holding the position while it plays (#Idle releases it)`); `#Idle` releases it afterwards. Left out, the loudness is 200 mA and the length 500 ms.

`Tone: not played` means the driver is not calibrated or in an error, the pitch is outside 20 Hz to half the loop rate, or parameter 81 is 0.

Over CAN and from Python

Send_Sound (ID 44) carries every operation: a tone now, an event for the queue, play the queue, play the samples, stop. Send_Sound_Samples (ID 45) fills the sample buffer (Sound in the CAN reference). The Python calls below need the Spectral-BLDC library 1.21 or newer, as does its `music` module (note names, ringtone melodies, MIDI files, WAV files).

Over CAN the host makes the driver hold first, for example with a position frame to where the motor is:

```
import Spectral_BLDC as Spectral
from Spectral_BLDC import music

comms = Spectral.CanCommunication(bustype='slcan', channel='COM41', bitrate=1000000)
motor = Spectral.SpectralCAN(node_id=0, communication=comms)


def read_answers(seconds=0.1):
    message, ids = comms.receive_can_messages(timeout=seconds)
    while message is not None:
        motor.UnpackData(message, ids)
        message, ids = comms.receive_can_messages(timeout=0.005)


motor.Send_Respond_Encoder_data()               # where the motor is
read_answers()
motor.Send_data_pack_1(motor.position, 0, 0)    # hold it there
read_answers()

motor.Send_Tone(440, current_mA=250, duration_ms=1000)        # one tone
for channel, note in enumerate(('C4', 'E4', 'G4')):            # a chord: three channels at once
    motor.Send_Tone(music.note_hz(note), current_mA=200, duration_ms=1500, channel=channel)
```

A tune goes into the queue and plays by itself. A melody becomes two events per note (the note, then a short silence, so repeated notes are heard as two), so up to 15 notes fit at once:

```
name, melody = music.parse_rtttl('scale:d=8,o=5,b=120:c,d,e,f,g,a,b,c6')
for channel, freq, current, wait in music.melody_events(melody, current_mA=250):
    motor.Send_Sound_Event(freq, current, wait, channel=channel)
motor.Send_Sound_Play()
```

For a longer tune, keep loading events while it plays: bits 0 to 7 of parameter 80 say how many are still queued. `music.read_midi` reads a MIDI file, `music.split_voices` splits it into lines for the four channels, and `music.lines_events` makes the queue's events from them.

Recorded sound: convert a WAV file to samples at the rate the motor plays, fill the buffer, start it, and keep it filled:

```
rate = 2500
samples = music.wav_samples('hello.wav', rate, coil_tau_s=0.00149 / 7.0, clip_percentile=0.95)
motor.Send_Samples(samples[:1000])              # fill the buffer first
motor.Send_Samples_Play(rate, full_scale_mV=2000)
sent = 1000
while sent < len(samples):
    motor.Send_Get_Parameter(80)
    read_answers(0.02)
    buffered = (motor.parameters.get(80, 0) >> 8) & 0x7FF
    room = 1000 - buffered
    if room > 0:
        motor.Send_Samples(samples[sent:sent + room])
        sent += room
```

`coil_tau_s` is the coil's L ÷ R (here 1.49 mH and 7.0 Ω, the motor of the calibration example): the function raises the high frequencies the coil would otherwise swallow, and `clip_percentile` levels speech so that it plays about four times louder.

At 2500 samples a second a driver takes about 313 frames a second, some 4 % of the bus. When the buffer runs dry the playing stops and bit 27 of parameter 80 says so.

Send_Sound and Send_Sound_Samples do not reset the watchdog: with a watchdog set, keep sending a status request while the sound plays.

Limits and care

- **Fix the motor to something that rings.** A box or a table is far louder than a motor in free air, and a loose motor jumps and distorts the sound. On the STEPFOC bench the motor was taped down.
- **Keep tones short and moderate.** `#Tone` and Send_Tone default to 200 mA; the bench played notes at 250 mA. The sound current comes on top of the holding current and heats the motor like any other current; a motor is made to turn, and long loud tones are hard on it.
- **On the q axis a tone is a small torque ripple:** a held joint vibrates slightly at the tone's pitch. Choose the d axis (parameter 82 = 0) when that matters; it is quieter.
- **The cap**, parameter 81, limits all sound together; the output also clamps at the supply voltage.

Sound has not been tested on a Spectral Micro yet; the code is the same as STEPFOC's. On the STEPFOC bench (2026-11-08, a bare NEMA 17 taped down), tones, scales, a tune from the queue, four-note chords and streamed speech all played.

The q axis was much louder than the d axis, speech was audible but muffled, and there were no errors, no warming and only small vibrations.

The arm as an instrument

Every joint of a robot can play its own line, and because the sound rides on top of the control, the arm keeps holding its pose while it plays.

Load each driver's queue first, then start them one after another: the starts are meant to fall within about a millisecond of each other, and from then on each driver keeps its own time (whether they stay together over a whole song is still to be measured).

Split a MIDI file into as many lines as there are channels, four per driver, and give each driver its four. Through a gearbox the sound changes with the joint; try each joint's range. Playing on an assembled arm has not been tested yet.

Checking what plays

Parameter	Says
79, bit 24	sound is playing
80	bits 0-7 events queued, 8-18 samples buffered, 24 a tone sounds, 25 the queue plays, 26 samples play, 27 the samples ran dry
81	the sound's voltage cap, mV
82	the axis: 1 q, 0 d

Gripper mode

A driver can run a gripper instead of a joint: the same board, with a state machine that moves a jaw to a position between open and closed, at a speed, with a current limit that becomes the grip force, and that reports when it holds something. The SSG-48 gripper is built on the Spectral Micro this way.

Turning it on

```
#Gripper 1
#Save
```

`#Default` clears the flag again. Over CAN the flag is parameter 92 from firmware 110: write 1 while the driver is idle (in any other mode the write is refused), then Send_Save_config.

A driver in gripper mode still answers every other command; the gripper commands only work with the flag set, except the gripper calibration, which runs with or without it.

Calibrating the gripper

After every power-up the gripper has to find its end stops: `#Gripcal` or Send_gripper_calibrate. The jaws open slowly at 500 mA until they stop. Then the push eases to 100 mA for a quarter of a second, and the driver reads the open end where the jaws rest against the stop. The closed end is found the same way.

Position 0 (open) lies 32 ticks inside the open end, and 255 (closed) 16 ticks inside the closed end. So a move to 0 or 255 ends in free travel: it does not press on the stop and reports no object.

Firmware 109 read the ends during the 500 mA push, so 0 and 255 lay inside the stops, and a full open or close kept pressing on them.

If it has not found both ends within about 30 s it gives up, and the gripper stays uncalibrated. From firmware 110 it also stays uncalibrated when the two ends lie less than 1000 ticks apart: the jaws did not travel.

Check the result with `#Gripinfo` or the `gripper_calibrated` bit of Respond_Gripper_data_pack. From firmware 110 parameter 93 gives the stroke between 0 and 255 in ticks, and 0 while the gripper is not calibrated. The motor itself must be calibrated (`#Cal`) and saved before that, once.

While the gripper calibrates, send only empty status polls. A 5-byte Send_gripper_data_pack, Send_Clear_Error or `#Clear` stops the calibration without a message. From firmware 110 the next calibration starts from its first step, and `#Gripcal` or Send_gripper_calibrate during a calibration starts it again. Firmware 109 went on where the stopped one left off (known issue KI-7).

Moving

Over UART:

```
#Gripvel 100       speed, 0 to 255
#Gripcur 400       current limit of the move in mA, the grip force
#Grippos 200       go to position 200 (0 open, 255 closed); also activates the gripper
```

Over CAN, Send_gripper_data_pack carries position, speed and current in one frame with the activate and action bits; the driver answers with Respond_Gripper_data_pack: the position, the measured current, whether it is moving, and the object detection:

Object detection	Meaning
0	moving
1	an object was detected while closing: the jaws stopped on it and hold it with the current limit
2	an object was detected while opening
3	the position was reached

From firmware 110 an object (1 or 2) is reported once the jaws have stood still at nearly the move's current for 50 ms. Firmware 109 reported it at the first slow control cycle, also at the start of a move.

From firmware 110 a move starts at the slowest speed and picks up to its own speed within 0.1 s. It slows down over the last stretch, so it arrives slowly. A new command in the same direction keeps the speed the jaws have, so a host that streams positions is not slowed.

At the position the driver holds with a small dead zone, a quarter of a position count and at most 16 ticks, and friction keeps the jaws there: a few mA are left after a move.

On firmware 109 the current took a second or more to settle after every move. From firmware 110 the position the driver reports is also rounded to the nearest count; 109 cut it off.

An empty Send_gripper_data_pack (0 bytes) only asks for the state. The answer to a 5-byte command goes out before the driver acts on it, so its object detection and error bits are still those of the moment before (KI-6); the answers to the next frames show the new move.

The action bit in the answer is a copy of the last command's action bit and stays 1 after the move ends (KI-8): the object detection tells when it has ended.

Send_gripper_data_pack's E-stop bit raises the gripper E-stop error; `#Gripstop 1` does the same over UART. The error stays: while the E-stop bit is 1 the driver raises it again on every control cycle (KI-5). To recover, clear the bit (`#Gripstop 0`, or a Send_gripper_data_pack with the bit 0), then `#Clear` or Send_Clear_Error.

Current

From firmware 109 the current you give a gripper move stops at the board's 2800 mA; firmware 107 and older did not limit it. The safe range depends on the gripper's mechanics and motor and is lower; the SSG-48 manual gives it for that gripper.

From firmware 110 the move's current is the gripper's own limit. The driver's current limit (`#Ilim`, parameter 16) stays as you set it, and a `#Save` stores that. Firmware 109 wrote the current of each move into the driver's limit, and a `#Save` then stored the gripper's current.

Where the details are

The bytes of the two gripper frames are in the CAN reference, the UART commands in the UART reference.

7. Firmware

Firmware

The driver's firmware is open source and released as numbered builds. Release 104 is the first release built for the bootloader: a driver that runs 104 or newer carries a bootloader and updates itself over UART or CAN. A driver that runs 101 or older needs the bootloader installed once with a programming adapter. A driver that reports 101 can be either (Drivers that report 101 says how to tell).

In this chapter

- Releases and versions: how releases are numbered, how to see which one a driver runs, what the release notes contain
- The bootloader: what it is, how the driver boots, how it is entered, its protocol and tools
- Updating through the bootloader: the normal update, with Motor GUI, the bootloader's command-line tool, or a PAR6 control box
- Upgrading from older firmware: what a driver keeps when it is updated to this release, what works at once and what has to be run
- Installing the bootloader on an older board: for a driver that runs firmware 101 or older
- Building from source: PlatformIO, the memory layout, uploading with a programming adapter

The release notes of every release up to the one this site describes are in the appendix, Firmware release notes.

Releases and versions

A firmware release is a number: 101, 104, 105, … Every release is a tag `V<number>` in the firmware repository and a GitHub release that carries the image `spectral-bldc-firmware-V<number>.bin`, a manifest `firmware.json` with the image's size, SHA-256 and the commit it was built from, and the release notes. The notes are the firmware's changelog; the appendix collects them.

Which release does my driver run?

```
#Info
```

prints `Software version: <number>` among the rest; Motor GUI shows the same number in the driver's identity, and over CAN Send_Respond_Device_Info returns it as the third byte. From firmware 105 the number is the build's own version.

Before that it was read from the EEPROM, where the last `#Default` wrote it, so it names the firmware that last restored the defaults, not necessarily the one running: 101 and 104 both write 101.

Drivers that report 101

A driver that reports 101 runs either release 101 from the `main` branch, without a bootloader, or release 104 (or an earlier build of `Bootloader_main`), which also writes 101 and runs above the bootloader. Drivers sold before release 105 shipped with firmware 101 or 103.

To tell whether yours has the bootloader:

- Watch the LED as you switch the driver on: with the bootloader it blinks for about two seconds before the firmware starts.
- Or power it up while `bl_flash.py list` or Motor GUI's *Scan bootloaders* is polling: a driver with the bootloader answers in its first two seconds.

With the bootloader, update it to the current release as in Updating through the bootloader; the release it ran before does not matter then. Without one, install it first: Installing the bootloader on an older board.

Which release does this site describe?

The one in the version box on the landing page and in the version selector in the header. When a newer release changes what a page says, a new version of the site is published and the previous one stays reachable from the selector. The compatibility table says which releases go with which bootloader, Python library and PAR6 runtime.

The STEPFOC counterpart

Most Spectral Micro releases have a STEPFOC release with the same changes. From release 109 the two drivers share their version numbers and are released together: the Spectral Micro went from 107 straight to 109, so there is no Spectral Micro 108.

Before that the numbers did not always match (Spectral Micro 107 went with STEPFOC 108); the compatibility table lists the pairs.

The two branches

Releases from 104 on are built from the `Bootloader_main` branch of the repository, whose firmware is linked to run above the bootloader; 104 is the first of them.

The `main` branch holds the firmware linked to run without a bootloader: release 101 and earlier, and a few later changes that were never tagged as a release (among them the Kt request and the voltage limit message). It is not updated.

Version `101` of this site documents a board running 101. It also fits 104, which adds the Send_Respond_Kt request, Send_Voltage_Limit (in the release image) and the bootloader.

The 104 release image was built from commit `20765c9` on `Bootloader_main`, not from the source at the tag `V104`, which does not have Send_Voltage_Limit yet. To move a board without a bootloader to a current release, see Installing the bootloader on an older board.

The bootloader

Every driver that runs firmware 104 or newer (104 is the first release built for the bootloader) carries a small bootloader in the first 12 KB of its flash.

It starts the firmware on a normal boot, and it loads a new firmware over UART or CAN when a host asks, so a driver is updated without a programming adapter and without opening anything. The bootloader is open source: stm32f103c-bootloader, shared with STEPFOC.

Memory layout

Address	Contents
`0x08000000` – `0x080022FF`	the bootloader (about 9 KB)
`0x08002C00` – `0x08002FFF`	the bootloader's configuration page: the board ID, the firmware's page count and CRC, the bootloader's build version
`0x08003000` – `0x0801FFFF`	the firmware (up to 116 KB)
RAM `0x20001000`	a magic word the bootloader sets to hand over to the firmware

The firmware is linked to start at `0x08003000` and sets its vector table there (`VECT_TAB_OFFSET=0x3000`); the first `0x3000` bytes belong to the bootloader and the firmware never touches them.

What happens at boot

1. On a reset, the bootloader runs first. If the magic word is in RAM and the reset was a software reset, it checks the firmware's stack pointer and reset vector and jumps straight into the firmware: about 10 ms, the normal boot.
2. Otherwise it initialises the clocks, CAN, UART and its watchdog, and **listens for a host for 2 s**.
3. With no host, and a firmware whose stored CRC is valid, it sets the magic word and resets; the next boot takes the fast path.
4. With a host talking, it stays as long as the host keeps talking: 3 s after the last command, 60 s once a flash is in progress. Any `PING` resets the timer.
5. With no valid firmware (never flashed, or a flash that was interrupted), it stays in bootloader mode and waits. The LED is off; the driver answers nothing on `#Info` .

Entering the bootloader

- **From the firmware**: `#Reset` over UART or Send_Reset (command 14) over CAN. Since firmware 104 both reset the driver into the bootloader's 2 s window; the host tools send them for you.
- **At power-on**: the 2 s window after every power-up, if the host is already polling.
- **Always**, when there is no valid firmware.

The board ID

The bootloader has a board ID of its own, 0 to 13, stored in its configuration page. 14 and 15 do not exist for it: its command frame, identifier `0x700`, decodes as node 14 in the firmware's identifier scheme, so those two node IDs are reserved.

Give drivers node IDs 0 to 13. The firmware accepts `#CANID 14` and `15`, but the bootloader then keeps its old board ID and a firmware update over CAN cannot reach the driver.

The firmware keeps the board ID in step with its CAN node ID when the configuration is saved: `#Save` or Send_Save_config writes the node ID into the bootloader's board ID, so a saved driver answers the host tools under its node ID. Setting the node ID alone (`#CANID`, Send_CAN_ID) does not: until you save, the bootloader keeps the old ID.

A replacement board ships as ID 0; two boards with the same ID on one bus corrupt each other's flash stream, so give the new board its ID alone on the bus first (`set-id` in the host tool).

Transports and protocol

Over **CAN** (1 Mbit/s, standard 11-bit identifiers) the host commands on identifier `0x700` with 8 data bytes (`board_id`, `cmd`, two parameters). The board replies on `0x701 + board_id` with 5 bytes (`board_id`, `cmd`, error code, a 16-bit parameter).

The firmware image streams in bulk frames on `(board_id << 7) | sequence`, 16 frames of 8 bytes per chunk, with no reply per frame.

Over **UART** (256000 baud) the same commands travel in framed packets `[0xBB][CMD][LEN][BOARD_ID][DATA…][CRC16]` with the same 5-byte reply.

Commands: `WBUF` (1), `WPAGE` (2), `WCRC` (3), `PING` (4), `SET_ID` (5), `ERASE_APP` (6), `STREAM_BEGIN` (7), `STREAM_STATUS` (8). Error codes: `0` OK, `1` invalid offset, `2` invalid CRC, `3` flash write failed, `4` invalid page number.

A flash is: ping, reset the firmware if it answers, ping until the bootloader answers, `ERASE_APP` (about 3.5 s), then for each 1 KB page `STREAM_BEGIN`, 8 chunks of 16 frames, `STREAM_STATUS`, `WPAGE` with the page's CRC32; finally `WCRC` with the page count and the image's CRC32.

The bootloader stores the CRC and reboots into the new firmware. The full wire protocol is in the bootloader repository's `docs/PROTOCOL.md`.

The host side

Tool	Transport	Use
Motor GUI	serial adapter (Web Serial) or CANvas	one driver on the bench, releases fetched from GitHub
`host_tools/bl_flash.py` in the bootloader repository	slcan or gs_usb CAN adapters, or UART	scripts, several boards, `list`, `set-id`, `flash --erase-first`
the PAR6 control box's `flash_motor_firmware.py` and the dashboard's Firmware tab	the robot's own CAN bus	every joint of a robot, from the box

Updating through the bootloader has the commands.

Recovery

An interrupted flash leaves an invalid CRC; the bootloader refuses to start the firmware and waits, so the board is not lost: flash again. A board that answers nothing at all: power-cycle it with the host already polling, to catch the 2 s window.

The one thing the tools refuse, and that would need a programming adapter to undo, is flashing an image linked for `0x08000000` (a firmware from the `main` branch) over the firmware region.

Installing it

Boards that run firmware 101 or older have no bootloader. Installing the bootloader on an older board puts it there once, with a programming adapter at `0x08000000`; from then on the board updates like any other.

Updating through the bootloader

A driver with the bootloader (firmware 104 or newer) is updated over the UART connector or the CAN bus: the host tells the driver to reset, the bootloader waits two seconds for a host, and the host streams the new image into it. No programming adapter, and nothing to open.

Before you update

- Park the mechanism: the motor loses torque during the update and until the new firmware is calibrated and commanded.
- An update may clear the calibration (the release notes say so when it does); plan to calibrate and `#Save` afterwards, and note the gains `#Info` prints. An update to 109 keeps it, and the driver runs on it; calibrate again where the motor can turn freely, or apply the SSG-48 preset on an assembled gripper (over UART, with *Gripper CAN ID* set to the gripper's node): Upgrading from older firmware says why, what changes and what to run.
- On a bus with several drivers, update each node; a bus with mixed releases behaves unpredictably.

With Motor GUI

Motor GUI in Chrome or Edge, connected over the serial adapter or the CANvas adapter: choose the release, and the GUI fetches the image and its manifest from the GitHub release, verifies the size and the SHA-256, resets the driver into its bootloader and flashes it. This is the way for one driver on the bench.

With the bootloader's command-line tool

The bootloader repository has `host_tools/bl_flash.py`, which flashes an image over CAN (slcan or candleLight/gs_usb adapters) or over UART:

Windows

```
cd host_tools
pip install -r requirements.txt

# over CAN with a CANvas (slcan) adapter on COM6, driver with board ID 0
python bl_flash.py --transport can --can-interface slcan --channel COM6 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first

# over UART on COM7
python bl_flash.py --transport uart --port COM7 --baudrate 256000 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first
```

Linux

```
cd host_tools
pip install -r requirements.txt

# over CAN with a CANvas (slcan) adapter, driver with board ID 0
python3 bl_flash.py --transport can --can-interface slcan --channel /dev/ttyACM0 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first

# over UART with the serial adapter
python3 bl_flash.py --transport uart --port /dev/ttyUSB0 --baudrate 256000 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first
```

Your user needs the `dialout` group for both ports (Motor GUI: the port).

macOS

```
cd host_tools
pip install -r requirements.txt

# over CAN with a CANvas (slcan) adapter, driver with board ID 0; ls /dev/cu.usbmodem* gives the name
python3 bl_flash.py --transport can --can-interface slcan --channel /dev/cu.usbmodem1101 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first

# over UART with the serial adapter; ls /dev/cu.usbserial* gives the name
python3 bl_flash.py --transport uart --port /dev/cu.usbserial-0001 --baudrate 256000 flash spectral-bldc-firmware-V109.bin --board 0 --erase-first
```

The tool sends the reset command to the running firmware (`#Reset` over UART, Send_Reset over CAN), catches the bootloader's two-second window, erases the application pages and streams the image page by page, each page verified by a CRC; `list` shows the boards in bootloader mode, `set-id` changes a board's bootloader ID. The image is the `spectral-bldc-firmware-V<number>.bin` from the GitHub release.

Board ID and node ID

The bootloader has its own board ID (0 to 13), kept in its configuration page, separate from the firmware's CAN node ID. The firmware copies the node ID into the board ID when the configuration is saved (`#Save` , Send_Save_config), so a saved driver's bootloader ID is its node ID. After `#CANID` or Send_CAN_ID without a save, `--board` still needs the old ID.

Check

After the flash, from Motor GUI or the tool, the driver starts the new firmware: `#Info` prints the release you flashed as `Software version` , and Motor GUI shows it in the driver's identity. If the driver does not answer, see When it fails.

From a PAR6 control box

A robot built on the Robot Control Box updates every joint from the box itself, over its own CAN bus, from the dashboard's Firmware tab or with `tools/flash_motor_firmware.py` ; the PAR6 documentation describes it.

What happens in the driver

1. The host sends the reset; the firmware reboots into the bootloader.
2. The bootloader listens for two seconds. With no host it starts the firmware; with a host it stays as long as the host keeps talking (60 s during a flash).
3. The application pages are erased, then written page by page; each page is checked by a CRC.
4. The bootloader checks the whole image's CRC, records it, and starts the new firmware.

When it fails

A flash that fails or is cancelled part way is not a brick: the stored CRC is invalid, so the bootloader refuses to start the firmware and waits for a host.

Run the flash again; `bl_flash.py list` (or a passive scan from the control box) finds the board. A board that answers nothing: power-cycle it with the host already polling, to catch the two-second window.

An image linked for `0x08000000` (a firmware from the `main` branch, release 101 and earlier) must never be flashed through the bootloader: it would overwrite the bootloader's own region if the tool did not refuse it. The tools validate the image first.

Upgrading from older firmware

Firmware 109 adds more to the Spectral Micro than any release before it: parameters, the watchdog actions, the identity and telemetry frames, captures, the current loop from one bandwidth, a choice of speed estimates, and the encoder table and anti-cogging.

This page says what a driver keeps when it is updated to 109, what works at once, what can be run, and how to go back. Updating through the bootloader is how to flash it.

There is no firmware 108 for the Spectral Micro: from 109 the Spectral Micro and STEPFOC share their version numbers, so a Spectral Micro goes from 107 straight to 109. The step from 109 to 110 is smaller: From 109 to 110.

From 109 to 110

Firmware 110 changes the gripper, adds `#Estop` and parameters 90 to 93, and saves brake or coast; nothing has to be measured again. A driver updated from 109 keeps everything it saved: the settings, the motor calibration, the encoder table and the anti-cogging. 110 keeps the idle behaviour in a slot of settings block 2 that was reserved.

- **Brake or coast** (parameter 18) is saved from 110 on. A driver that has not saved it yet coasts in idle, as on 109; send `#Brake 1` and `#Save` once if it should brake.
- **The gripper calibration** is kept in RAM, as it always was: calibrate the gripper after the update, as after every power-up. Positions 0 and 255 then lie a little inside the two ends (Gripper mode).
- **Motor GUI 0.7.0** is the version made for 110: its STOP sends `#Estop` over UART, and its presets write parameters 90 to 92 over CAN.

Going back to 109 is safe: 109 ignores the saved idle behaviour and coasts at every power-up again.

What is kept

Flashing never touches the EEPROM, where the configuration lives (The check at power-up). A driver updated from 106 or 107 starts with everything it had:

- the calibration: pole pairs, R, L, the angle offset, Kt, KV, the phase order and direction, the current gains;
- the CAN ID, the gains, the limits, the watchdog, the heartbeat, the thermistor and the gripper settings.

Calibrate again after the update, where the motor can turn freely

In every firmware before 109 the enable input of phase B was fed by a clock signal instead of a steady level, so phase B could drop out, also while the calibration measured the motor. A calibration made with 107 or older stored the resistance about 10 % and the inductance about 45 % too low (reference gimbal motor: about 6.4 Ω and 1.25 mH, where 109 measures about 7.0 to 7.2 Ω and 1.75 to 1.9 mH).

The driver keeps running on such a calibration: on the bench an SSG-48 updated to 109 without a new calibration reached every target and gripped as before. Only the current gains are computed from the low values. So:

- **A motor that can turn freely:** run `#Cal` once after the update. The current gains then follow the true values, and the encoder table and the anti-cogging are measured too.
- **An assembled SSG-48 gripper,** whose motor moves the jaws: do not run `#Cal`. Apply Motor GUI's SSG-48 preset over UART instead: it writes 300 Hz current gains for the gripper's motor and saves. Set *Gripper CAN ID* to the gripper's node first: it starts at 6, the PAR6 gripper's; a PAROL6 gripper is node 0. A motor calibration needs the body opened and the linear guide with the jaws off; then use `#Calonly`, because `#Cal` also runs the encoder table and the anti-cogging.

A driver from 101 to 105, saved before the configuration had checksums, is trusted and sealed at its first start of 109 when it was ever saved; a blank EEPROM gets the defaults. Release 101 and older from the `main` branch have no bootloader: such a board needs the programming adapter once (Installing the bootloader on an older board).

The new settings start from their defaults: tool id 0, the watchdog fallback 2000 ms / 0 mA / 1000 ticks/s / legacy, parameter 56 = 0.

What works at once

These need no measurement and are on from the first start of 109:

- the **current-sense zero**, measured at every start-up with the driver asleep: on the bench the current read 3.9 mA with nothing flowing, where it read −20 to +19 mA before, depending on the rotor's position;
- undervoltage is not checked for the first 500 ms after power-up, so a supply that ramps up slowly no longer leaves a Vbus error behind (overvoltage always is);
- the **watchdog actions**: the action byte the host already sends is acted on. Byte 0, what the Python library and the PAR6 runtime send, gives the legacy response, so an unchanged host sees no difference;
- phase B's enable pin is driven properly, and the control loop keeps the gate driver awake in every driving mode: either could leave a motor that took a command and did not move, or that pushed unevenly;
- the limits refuse a negative value, and Send_Limits and the gripper's current stop at 2800 mA;
- setting the watchdog time starts a fresh window, so switching it on again no longer trips it at once;
- `#Default` and a settings reset **keep the CAN ID**;
- the first motion frame after Idle is followed (the driver's wake-up now comes with it), and a position or speed frame after another mode no longer kicks with a stale velocity integral;

- the product lock, the new CAN commands and parameters, the identity frame, telemetry, the capture, `#Info`'s `In force:`, and sound through the motor (not yet tested on a Spectral Micro).

What can be run or switched on

The encoder table and the anti-cogging are off after the update until they are measured (`#Cal` measures both); the speed estimate and the delay compensation are new defaults, on from the first start:

- the **encoder table** and the **anti-cogging** (Encoder table and anti-cogging); they cannot be switched on before they are measured;
- the **direct PLL speed estimate** at 100 Hz (parameters 50 and 51) and the **commutation delay compensation** of 234 µs (parameter 64), both on by default from 109 (Current limit and loop tuning). They change how the velocity loop behaves with the gains you had: a speed step overshoots less. Check your velocity gains after the update.

What does not change by itself

- **The current gains.** The saved Kpiq and Kiiq stay as they are: on a driver calibrated before 109 they were computed for 500 Hz (parameter 49 reads about 500), from a resistance and an inductance that were too low. A calibration with 109 measures both again and computes the gains at 160 Hz. `#Cbw 160` on the old calibration does not replace it: it uses the old values and gives gains for a lower bandwidth than it names. 160 Hz is the new default; it was chosen on the STEPFOC bench.
- **The velocity gains** stay as they are; their defaults, Kpv 0.03 and Kiv 0.0003, did not change.

The whole upgrade

1. Flash 109 (Updating through the bootloader). `#Info` prints the new firmware version and ends with `In force:`.
2. Check that the current limit suits your motor (`#Ilim`): calibration puts back the limit that was in force, and the encoder table in the next step runs at 80 % of it.
3. Calibrate, with the shaft free: `#Cal`, so that the resistance, the inductance and the current gains are measured again. It goes on to the encoder table and the anti-cogging, and saves everything. `#Calonly` is the calibration alone.
4. After `#Calonly`: `#Save`; and if you want them, run the encoder table and the anti-cogging with the shaft free (`#Encal all`) and check them.

With Motor GUI's presets: apply the preset first, then calibrate. A product preset writes its own current and velocity gains, so one applied after the calibration puts its current gains back. The *Generic motor* presets set soft limits and velocity gains and leave the current gains to `#Cbw` and the calibration.

Going back

Flashing 107 or older over 109 is safe for the configuration: the calibration and the first settings block are unchanged, and older firmware ignores everything 109 saved from byte 256 on (tool id, fallback, anti-cogging, encoder table). Flash 109 again and they are all still there.

Not yet tested on the bench

A downgrade and the sound on a Spectral Micro had not been tested on the bench when this page was written; the rest of this page was.

Installing the bootloader on an older board

A driver running firmware 101 or older has no bootloader: its firmware starts at the beginning of the flash. A driver that reports 101 may run 104 and have the bootloader all the same; check first, as in Drivers that report 101.

To run any release from 104 on (104 is the first release built for the bootloader), which is linked to start above the bootloader, the bootloader must be installed once, with a programming adapter on the JTAG connector. After that, every update goes over UART or CAN.

> **Warning**
>
> Disconnect the motor supply and the UART adapter before connecting the programming adapter. The adapter powers the board at 3.3 V.

What you need

- The programming adapter (an ST-LINK with the 1.27 mm pogo-pin adapter for the JTAG pads), or any ST-LINK, J-Link or OpenOCD-compatible probe wired to 3.3 V, GND, SWDIO and SWCLK.
- STM32CubeProgrammer or the older STM32 ST-LINK Utility on the PC, with the ST-LINK drivers.
- The bootloader binary from the bootloader repository's `bin/` folder; its `README.md` names the current file.

Windows

STM32CubeProgrammer or the ST-LINK Utility, and the ST-LINK USB driver (STSW-LINK009) for the adapter.

Linux

STM32CubeProgrammer; the ST-LINK Utility runs on Windows only. The adapter needs no driver, but your user needs access to its USB device: install the udev rules that come with STM32CubeProgrammer, or run it with `sudo`.

macOS

STM32CubeProgrammer; the ST-LINK Utility runs on Windows only. The adapter needs no driver.

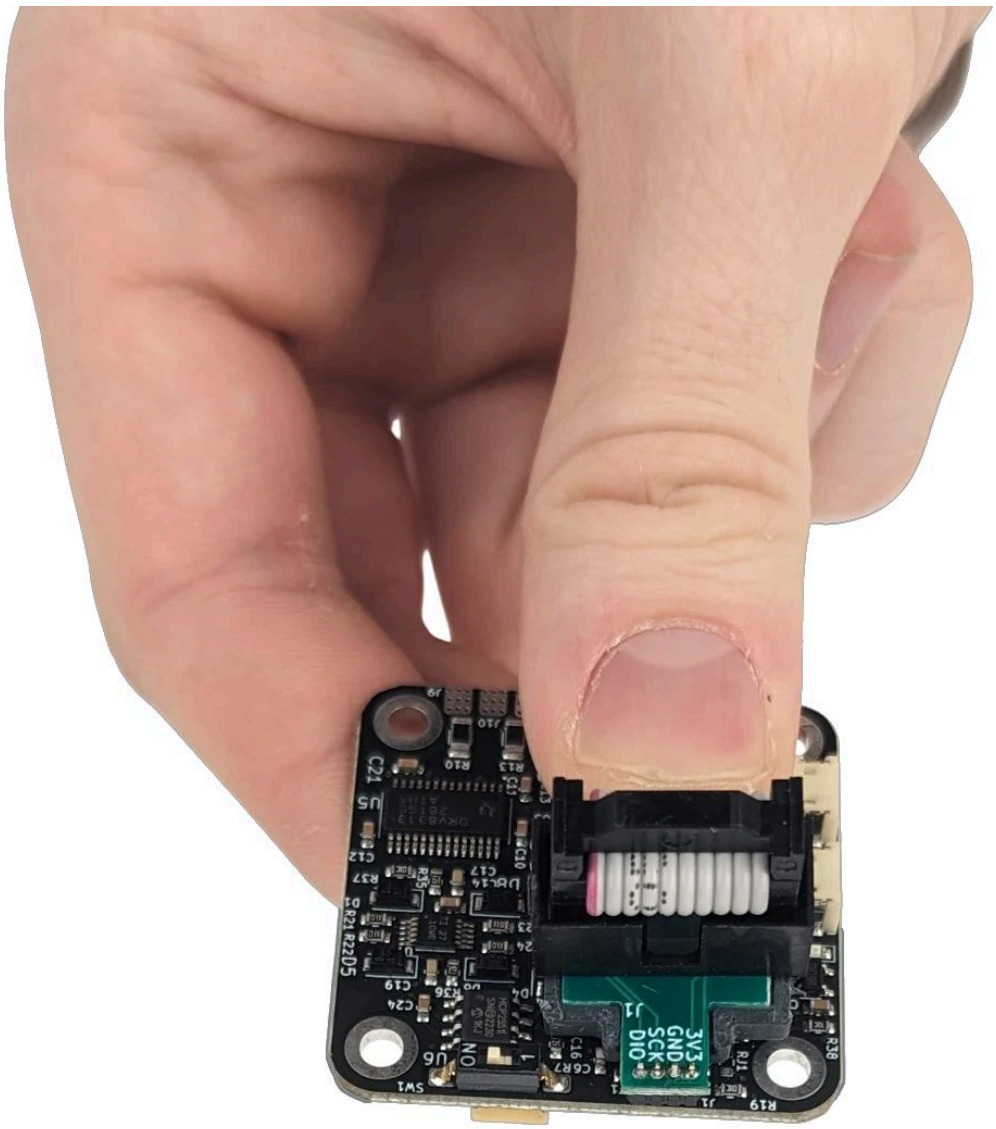


Figure 7.1: The JTAG pads. The adapter is pressed onto them; a little pressure makes the contact.

Steps

1. Connect the adapter to the JTAG pads and the PC.
2. In the programmer, connect to the target (SWD). The STM32F103CB shows up with 128 KB of flash.
3. Erase the chip, or at least the application region: the old firmware and the bootloader must not overlap.
4. Program the bootloader binary at address `0x08000000`.
5. Disconnect the adapter, power the driver. The bootloader finds no valid application and waits in bootloader mode (the LED stays off).
6. Flash the firmware release through the bootloader, over UART or CAN, as in Updating through the bootloader.

7. `#Info` shows the new release. `#Default`, then calibrate (`#Cal`), set the current limit (`#Ilim`) and `#Save`; see Calibration.

Check

`#Info` prints the release you flashed, and a driver powered up while `bl_flash.py list` or Motor GUI's *Scan bootloaders* is polling answers in its first two seconds: the bootloader is there. If the firmware does not start, see Updating through the bootloader: when it fails.

Memory layout

`0x08000000` – `0x080022FF` bootloader, `0x08002C00` its configuration page (board ID, the application's CRC), `0x08003000` – `0x0801FFFF` the application (116 KB). The bootloader keeps a magic value in RAM to hand over to the application in about 10 ms on a normal boot.

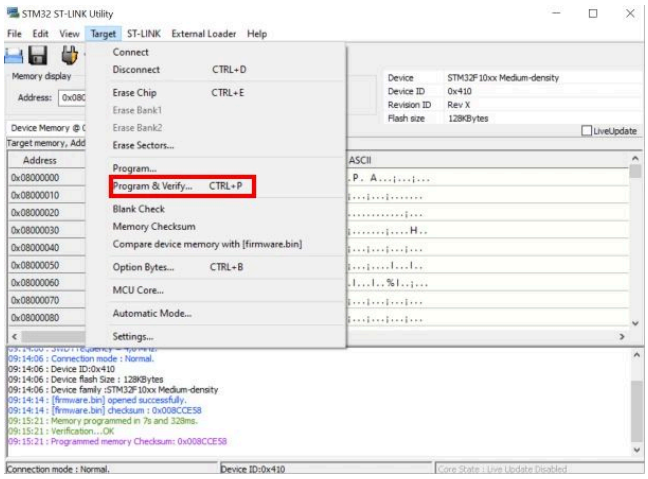
The programmer's settings

In ST-LINK Utility, *Target → Settings* with SWD and the default frequency works; *Target → Program & Verify* with the start address `0x08000000`. The pictures of the old flashing procedure apply to the connection and the settings dialog:

Settings



Programming



The same adapter uploads a firmware you build yourself, see Building from source.

Building from source

The firmware is a PlatformIO project for the Arduino core on the STM32F103CB, in the `Spectral BLDC Firmware` folder of the firmware repository on the `Bootloader_main` branch.

Building

1. Install Visual Studio Code and the PlatformIO extension.
2. Clone the repository and check out `Bootloader_main`.
3. Open the `Spectral BLDC Firmware` folder in VS Code; PlatformIO fetches the platform (`ststm32@15.4.1`) and builds with the PlatformIO *Build* button or `pio run`. The image is `.pio/build/genericSTM32F103CB/firmware.bin`.

The project is linked for the bootloader: `VECT_TAB_OFFSET=0x3000`, the linker script `ldscript_bootloader.ld`, the upload offset `0x08003000` and a maximum size of 118784 bytes, in `platformio.ini`. A build that exceeds the size fails to link.

> **Check**
>
> PlatformIO ends the build with `SUCCESS`, and `.pio/build/genericSTM32F103CB/firmware.bin` is there. If it fails, the first error above the summary names the cause.

Uploading

PlatformIO runs on Windows, Linux and macOS alike. On Linux, an upload with the programming adapter also needs PlatformIO's udev rules (`99-platformio-udev.rules`, in the PlatformIO documentation), and a serial port the `dialout` group.

- **Through the bootloader**: flash the built `firmware.bin` with `bl_flash.py` or Motor GUI as in Updating through the bootloader. This is how a release is installed, and works on any driver with the bootloader.
- **With the programming adapter**: *Upload* in PlatformIO uses ST-LINK and writes the image at `0x08003000`. The bootloader must already be on the board; the upload does not touch it. The serial monitor (the plug icon) opens the UART at 256000 with `send_on_enter`, so you can type commands to the driver from the same window.

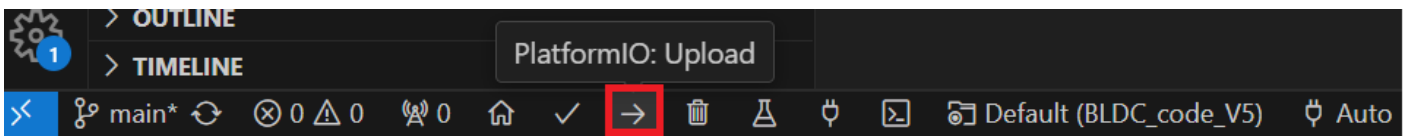


Figure 7.2: Build, upload and the serial monitor in PlatformIO's status bar.

Where things are

`src/constants.h`	`FIRMWARE_VERSION`, the CAN command IDs, the EEPROM map, the loop and PWM frequencies
`src/common.h`	the controller and PID structures with their defaults
`src/communication.cpp`	the UART command handlers
`src/communication_CAN.cpp`	the CAN receive switch and the transmit functions
`src/SerialPacketParser.cpp`	the UART parser: it splits a line into the command and its argument (up to 107 it also kept the list of names it accepted)
`src/motor_control.cpp`	the control loops, the calibration routine, the gripper state machine
`src/bootloader_config.cpp`	the vector table offset and the reset into the bootloader
`release_guide.md`	how a release is cut: version bump, build, tag, GitHub release with `firmware.json`

Making a release

`release_guide.md` in the repository is the procedure: bump `FIRMWARE_VERSION`, build, commit the source change, tag `V<number>`, create the GitHub release with the image, `firmware.json` and the release notes. This site's `tools/sync_firmware.py` reads a release by its tag; the version box names the release.

A firmware for a board without a bootloader

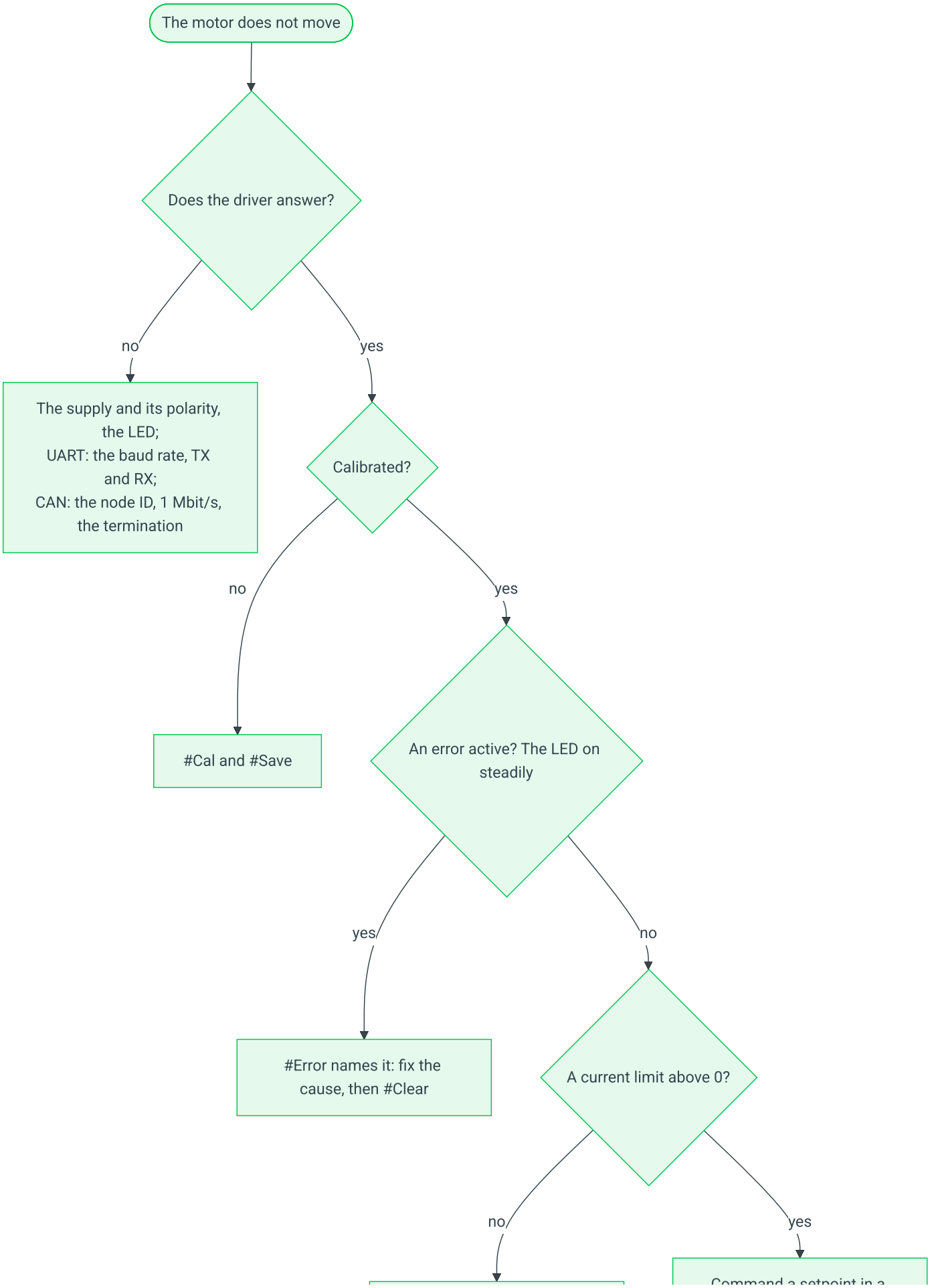
The `main` branch builds a firmware linked at `0x08000000` for boards that have no bootloader. Its last tagged release is 101; it is not maintained. Install the bootloader instead: Installing the bootloader on an older board.

8. Troubleshooting

Troubleshooting

By symptom. The error flags and what clears them are in Errors and recovery; the faults of the firmware that are known and not yet fixed, with what to do until they are, in Known issues.

The motor does not move


The motor does not move
Does the driver answer?
no
yes
The supply and its polarity, the LED; UART: the baud rate, TX and RX; CAN: the node ID, 1 Mbit/s, the termination
Calibrated?
no
yes
#Cal and #Save
An error active? The LED on steadily
yes
no
#Error names it: fix the cause, then #Clear
A current limit above 0?
no
yes

#Ilim to the motor's rated current, #Save

Command a setpoint in a closed-loop mode; #Mode and #Info show the state

The driver does not answer

Symptom	Check
No LED	the supply, DC+ and DC−, the power cable's orientation. If the cable was the wrong way round the driver is destroyed
LED on, nothing on the terminal	256000 baud; TX to RX and RX to TX; the adapter's port; 3.3 V adapter; the line ending CR+LF
LED on, nothing on CAN	the node ID; 1 Mbit/s on the adapter; termination (about 60 Ω across CAN H and CAN L with everything unpowered); the CAN cable's orientation
`Unknown command`	the name's spelling and case: `#Kpiq`, not `#kpiq`; an argument with a space in it (`#WdFb 2000,0,1000,2`, not `#WdFb 2000, 0, 1000, 2`)
`Wrong firmware for this board (product lock)`, the LED on steadily	the other product's firmware was flashed: a STEPFOC image on a Spectral Micro. Flash the Spectral Micro firmware; the configuration was not touched (The product lock)
the LED is off and nothing works after a firmware update	the bootloader has no valid firmware: flash again, see Updating through the bootloader

Motor GUI

Symptom	Check
the browser offers no port	a Chromium-based browser (Chrome, Edge, Brave, Opera); Firefox and Safari have no Web Serial
the port is listed but will not open	another program holds it: a terminal, PlatformIO's monitor, the Arduino serial monitor
the port opens, nothing appears	the driver's power; TX and RX crossed on a non-Source adapter; the baud rate on a non-Source adapter (256000)

Symptom	Check
No connection on CAN	the CANvas adapter's port; the bus termination; the node ID of the driver; 1 Mbit/s
the calibration does not start	calibration runs over the serial connection only, not over CAN
Unsaved changes after a power cycle	the settings were not saved to the driver; Save writes them
the firmware update finds no bootloader	power-cycle the driver while the GUI scans, to catch the bootloader's 2 s window. A driver with firmware 101 or older has no bootloader (one that reports 101 may have it: Drivers that report 101): Installing the bootloader on an older board

Calibration

The messages and their causes are in Calibration. The ones that come back most:

Symptom	Cause
`Magnet test: Failed`	magnet too far, axially magnetised, or missing
the motor wobbles left and right during calibration	one phase is not connected
`Open loop: Failed`	the rotor did not follow the open-loop field, or the pole pairs came out outside 1 to 49: a phase open, the shaft not free, the open-loop voltage (`#Openv`) too low for the motor
`Commutation/alignment check failed!`	the motor did not respond to the commutation check: a phase open, `#Iqoffset` too low, a magnet that slips
calibration succeeds, the motor then runs badly or in the wrong direction	the magnet slipped on the shaft; glue it and calibrate again
after `Calibration success!` the motor turns slowly by itself for about a minute	the encoder table and the anti-cogging that follow a `#Cal` ; let them finish, they save everything (What follows the calibration)

Symptom	Cause
the anti-cogging does not switch on (parameter 37 stays 0)	no set is identified for that form; run `#Cogcal save`. After an encoder table measured on its own the set is switched off until it is identified again (Encoder table and anti-cogging)
calibration succeeds but the LED stays on and `#Error` shows nothing	`#Info` shows `-1`, a value from an older release (The check at power-up): `#Default`, calibrate again

The motor

Symptom	Cause
crunching or whining at speed	the current limit or the current loop gains too high; lower `#Ilim`, lower `#Cbw` and calibrate again, or lower `#Kpiq`
humming, oscillating at rest	`#Kpv` or `#Kpp` too high; see Current limit and loop tuning
weak torque	`#Ilim` low; the supply's current limit; a calibration run at a low supply voltage (the constants are off)
the motor is hot	lower `#Ilim`, enable the thermistor; a gimbal motor held against a load at its current limit heats quickly
it stops with a Vbus error when decelerating	the supply cannot absorb the returned energy; a different supply, or a brake resistor
it stops with a velocity error	the measured speed passed the fixed limit of 20,000,000 ticks/s: an encoder fault or a slipping magnet, rarely a real runaway; `#Vlimit` does not change it
it falls behind under a load	the torque limit: raise `#Ilim` within the motor's rating, or a bigger motor
it holds position with a sawtooth noise	the position loop gain; the magnet distance

Low-quality motors

Motors with visibly bad windings, an uneven back-EMF or a resistance that differs between the phases calibrate poorly and run rough. The tested motors are known good.

The magnet and the encoder

- The distance is 1 mm. "Magnet Good" from the magnet test does not mean the distance is right, only that the field is strong enough.
- A magnet touching the chip can destroy the microcontroller; a magnet too far gives a noisy, wrong position.
- The magnet must be diametrically magnetised and centred on the shaft.

The driver gets hot or shuts down

The MOSFET driver stage shuts down on its own overtemperature or overcurrent and reports a driver error. Lower the current limit, give the board airflow, check the motor leads for a short. Near 2.8 A the driver needs cooling.

After a firmware update

`#Info` shows the new release. If the values look wrong or the LED stays on: `#Default`, `#Cal`, `#Save`. A release note says when an update clears the calibration. After an update to 109, calibrate again (`#Cal`) where the motor can turn freely; on an assembled SSG-48 apply its preset instead. Upgrading from older firmware says why and what else changes.

A Vbus error right after power-up, cleared by `#Clear`, was the supply ramping up; from firmware 109 a low supply is not checked in the first 500 ms.

Getting help

Support says where to ask and what to include; `#Info`'s output is the first thing.

Known issues

The faults in the Spectral Micro firmware that are known and not yet fixed, and what to do until they are. The pages that describe each part say the same where it matters; this page collects them in one place. Which releases work together is in Versions and compatibility.

Spectral Micro firmware

Seen in: 110.

	What happens	What to do	More
KI-3	The serial number is not read back from the EEPROM at power-up, so `#SERNUM`, Motor GUI and Send_Respond_Device_Info report 1, and every `#Save` writes that 1 over the stored number.	Tell drivers apart by their CAN node ID. To keep a serial number stored, set it with `#SERNUM` before every `#Save`.	Saving: what is not saved
KI-4	For anyone reading or changing the source, the start values in `common.h` are not the factory defaults (the gripper flag starts at 1, the pole pairs at 11, `Max_Vbus` at 29500 mV). The EEPROM replaces them at power-up, so a running driver does not show them.	Take the defaults from the Defaults page, which lists what `#Default` writes, or from `Set_Default_config()` in `EEPROM.cpp`.	Defaults
KI-5	While the E-stop bit of the last Send_gripper_data_pack (or `#Gripstop 1`) is 1, the driver raises the gripper E-stop error again on every control cycle, so Send_Clear_Error or `#Clear` alone does not clear it.	Send a gripper frame with the E-stop bit 0 (or `#Gripstop 0`) first, then clear the error.	Gripper mode: moving
KI-6	The answer to a 5-byte Send_gripper_data_pack	Judge the new move by the answers to later frames, or	Gripper mode: moving

	What happens	What to do	More
	goes out before the driver has acted on the command, so its object detection and error bits are those of the moment before, for example 3 (position reached) from the previous move.	poll with an empty Send_gripper_data_pack.	
KI-8	The action bit in Respond_Gripper_data_pack is a copy of the last command's action bit; the driver never clears it, so it stays 1 after the move has ended.	Tell the end of a move by the object detection, 3 position reached, 1 or 2 stopped on an object.	Respond_Gripper_data_pack
KI-9	Open loop (`#Openloop`) puts the voltage `#Openv` (5000 mV by default) on the phases without a current limit; `#Ilim` does not apply, so a low-resistance motor draws far more than its limit.	Lower `#Openv` before you run a low-resistance motor in open loop, and keep the test short.	Control modes
KI-10	The velocity error trips only above a fixed 20,000,000 ticks/s, about 1220 revolutions per second and far above the speeds the driver can follow; `#Vlimit` does not change it, so it does not catch a motor that runs away.	Watch the speed in your host and send Send_Idle or Send_ESTOP when it is wrong.	Errors and recovery
KI-11	Only motion frames and answered status requests reset the watchdog. Setting frames (gains, limits, Idle, Clear) and UART commands do not, so a host that only configures the driver, or a terminal session with a saved	While the watchdog runs, send a status request such as Send_Respond_Encoder_data within every watchdog period. Before you work over UART alone, set the watchdog to 0 and save (`#Par 6,0` , then `#Save`).	The watchdog

	What happens	What to do	More
	watchdog, ends in a watchdog error.		
KI-12	The current a turned motor drives through its shorted phases while the driver brakes in idle (`#Brake 1`) is not limited.	Brake only a motor whose resistance keeps that current under 2.8 A at the speeds it is turned at, and do not let a load drive a braking motor fast.	Braking in idle
KI-14	Send_Respond_Device_Info carries the batch date in one byte, its lowest 8 bits, so a host reading it over CAN gets 120 for the 24092024 that `#Default` stores.	Read the batch date with `#Info` over UART, which prints the whole number.	Send_Respond_Device_Info
KI-19	The encoder table holds a field current of 80 % of the current limit for about 20 s, also in the run that follows every `#Cal` . With a limit above the motor's rating (1000 mA after `#Default`), the run is above the rating too.	Set `#Ilim` for the motor before `#Encal` and before every `#Cal` (calibration puts the limit that was in force back before the runs that follow).	Encoder table and anti-cogging: running them

Release 104

Seen in: the image of the GitHub release V104.

	What happens	What to do	More
KI-16	For anyone who builds or reads the source of release 104, the tag `V104` is not what its image was built from. The image comes from commit `20765c9` on `Bootloader_main` , which adds Send_Voltage_Limit; the tag lacks it.	Check out `20765c9` to see or rebuild the 104 image's source; better, update the board to the current release.	Releases: the two branches

Fixed

	What happened	Fixed in	More
KI-1	The current limit of Send_Limits over CAN is used as sent, also above the driver's 2800 mA; only `#Ilim` over UART clamps it.	109: Send_Limits stops the current limit at 2800 mA and ignores a negative value	Send_Limits
KI-2	Calibration sets the current limit to 1600 mA and leaves it there, whatever it was before; a `#Save` right after calibration stores the 1600 mA.	109: calibration puts back the current limit that was in force when it started	Calibration: current limit and save
KI-7	A 5-byte Send_gripper_data_pack, Send_Clear_Error or `#Clear` during the gripper calibration stops it without a message and leaves its counters as they were, so the next calibration starts part way through; it may skip the opening or give up early.	110: a new gripper calibration starts from its first step, also after one that was stopped (a command during the calibration still stops it)	Gripper mode: calibrating
KI-13	`#Save` does not keep the idle behaviour: after every power-up the driver coasts in idle again, whatever `#Brake` or Send_Settings set before.	110: `#Save` keeps the idle behaviour (`#Brake` , Send_Settings, parameter 18)	Braking in idle
KI-15	`#Error` prints `FAULT pin is HIGH` while the driver stage reports a fault, which is when that pin is low: the text names the wrong level.	109: the line reads `FAULT pin is LOW: the gate driver reports a fault`	Errors and recovery

Something that is not here

Check the symptom tables first. If it is still not explained, Report a problem says what to include so it can be reproduced. A fixed issue stays on this page, under "Fixed", with the release that fixes it.

Appendix

Firmware release notes

What changed in each firmware release, from the release notes in the firmware repository, up to the release this site describes.

There is no release 108 for the Spectral Micro: from 109 it shares its version numbers with STEPFOC, and goes from 107 straight to 109. Upgrading from older firmware says what a driver keeps and what to run when it moves to 109.

V110 (2026-11-13)

Spectral BLDC firmware **V110** — application image for flashing via the Spectral bootloader. Mostly gripper work: an SSG-48 gripper (a Spectral Micro in gripper mode) now ends its moves cleanly and closes fully at position 255. Everything below is described in the manual: Spectral Micro docs; what a driver keeps and what to run after the update: Upgrading from older firmware.

Grippers

- **A move ends cleanly.** On 109 the current took over two seconds to settle after a move: the speed loop's integral was carried into the hold and unwound slowly through friction. Now the integral is dropped on arrival, the last stretch of a move slows down, and the hold has a small dead zone (a quarter of a position count). A few mA are left after a move.
- **A soft start.** A move picks up speed over 0.1 s instead of starting at full speed.
- **Positions 0 and 255 clear the stops, and 255 is fully closed.** On 109 the calibration read the ends while pushing at 500 mA, so 0 and 255 lay inside the stops: a full open or close pressed on the stop, drew the grip current for good and often reported an object. Each end is now read under a light 100 mA push (released to no current, the SSG-48's free-running jaws spring back about 1 mm off the stop); 0 sits 32 ticks inside the open stop and 255 16 ticks from the closed one. On an SSG-48 the jaws stop 0.12 mm short of fully closed at 255, and the calibration repeats within 4 ticks.
- **Object detection** reports an object after the jaws have stood still against it for 50 ms.
- **The gripper pushes with its own current limit.** On 109 every gripper command overwrote the driver's current limit (parameter 16) with its grip current, and a Save config then stored it.
- The gripper calibration can be started again while it runs, ends on a timeout or a stroke under 1000 ticks, and puts the current limit back. The position is rounded to the nearest count (it was cut off).

CAN bus

- **Parameters 90-93:** 90 `#Rstint`, 91 the thermistor (`#Term`), 92 gripper mode (`#Gripper`, written only while idle), each 0 or 1 and saved by Save config, so a preset over CAN can set everything (issue #1); 93, read only, the gripper's stroke between positions 0 and 255 in ticks (0 until calibrated).

UART

- `#Estop`: the E-stop, as CAN has it. The motor is released, a running calibration ends, and the E-stop latches until `#Clear`.
- `#Close` **during a motor calibration** stops the calibration (`Close: the calibration is stopped`). On 109 it switched the mode under the running calibration.

Settings and safety

- **Brake or coast when idle (parameter 18) is saved** by Save config. On 109 it was back to its default at every power-up. A driver that never saved it coasts, as before.
- **Watchdog:** a trip while the driver is idle no longer starts the hold or the descend (on 109 it did, for the whole fallback time). A trip while a loop is closed holds as before.

Compatibility

- The settings and the motor calibration of 109 are kept. The gripper calibration is kept in RAM, as always: calibrate the gripper after the update.
- Python library **1.22** (`pip install --upgrade Spectral-BLDC`) names parameters 90-93; 1.21 reads them by number. Motor GUI **0.7.0** uses what 110 adds: STOP over UART sends `#Estop`, brake or coast counts as saved, a preset over CAN writes 90-92.
- Bench-tested for this release on an SSG-48 gripper: the moves and the hold, moves of one count, the ends and the closed gap, the soft start, gripping objects, the calibration's repeatability, parameters 90-93, brake or coast over a save and a restart, the driver's current limit through gripper moves, the watchdog while idle and in a closed loop, an aborted gripper calibration. `#Estop` and `#Close` were tried over UART on the STEPFOC, which has the same code. Not tested on the bench: an upgrade from V103 with the ST-Link, and a downgrade.

Assets

- `spectral-bldc-firmware-V110.bin` — application firmware image (flash at `0x08003000`, target STM32F103CB).
- `firmware.json` — manifest with version, size, and SHA-256 for download verification by the web flasher.

Integrity

- SHA-256: `eae54f56b4d0eb3677155447a5ce68156bfbda16e81534513ad71cb0dbbbfb21`
- Size: 112984 bytes

Built from commit 38d210d on `Bootloader_main` (`#Info` shows it as `Build: 38D210D`). Includes bootloader entry support (CAN/UART reset-into-bootloader).

V109 (2026-11-10)

Spectral Micro BLDC firmware **V109** — application image for flashing via the Spectral bootloader. From 109 the Spectral Micro and the STEPFOC share their version numbers, so there is no 108: 109 follows 107. Everything below is described in the manual: Spectral Micro docs; what a driver keeps and what to run after the update: Upgrading from older firmware.

Phase B (the main fix)

- In every release up to 107 the gate driver's phase-B enable input was fed by the microcontroller's 8 MHz clock output instead of a steady level, so phase B could drop out: weaker or uneven torque, a calibration that failed with a wrong pole-pair count, or a motor that took a current command and did not move, with no error. 109 drives the pin properly. On the bench: 5 of 5 calibrations and full torque on every test, also over 10 restarts.
- The calibration now reports the true phase resistance and inductance (on the reference gimbal motor about 7.0-7.2 Ω and 1.75-1.9 mH, where 107 reported about 6.4 Ω and 1.25 mH); the old × 1.25 correction is gone.
- **After the update:** a calibration made with 107 keeps working. Calibrate again (`#Cal`) where the motor can turn freely, so the current gains follow the true values. An assembled SSG-48 gripper keeps its calibration: apply Motor GUI's SSG-48 preset instead, with the gripper's CAN ID set first.

New features

- **Parameters:** any setting or read-out by number over CAN (commands 36-38) or `#Par` (parameters 1-89). Identity (39) and telemetry (40) frames.
- **Capture, histogram and buffers** (41-43).
- **Watchdog actions:** idle, brake, coast, hold the position, or descend at a capped speed, with a fallback time.
- **Current loop from R, L and one bandwidth** (parameter 49, default 160 Hz; the SSG-48 preset uses 300 Hz). Measured on the bench and on by default: the direct PLL speed estimate at 100 Hz and commutation delay compensation of 234 µs (a speed step overshoots about 18 % instead of 63 %).
- **Encoder table and anti-cogging measured on the driver:** `#Cal` runs the calibration, then both, and saves; `#Calonly` is the calibration alone.
- **Sound through the motor:** tones, a queue of notes and samples (CAN 44 and 45, `#Tone`).
- **Identification:** `#Info` and parameters 84-87 report the build (git commit), the board ID and a calibration ID. Parameters 88 and 89 are an experimental current-sampling diagnostic, off by default.
- **Settings in blocks with their own checksums;** a reset keeps the CAN ID; a product lock refuses a STEPFOC image on a Spectral board.

Calibration

- Both current sensors are averaged; pole pairs 1-49; a rotor that does not follow the open-loop field ends with `Open loop: Failed` and two hint lines; an inductance that gives no gains fails the calibration instead of hanging.
- Kt and KV are measured in four spins alternating direction; a warning only when two same-direction spins differ by more than 5 %.
- The gate driver stays awake for the whole calibration. A failed calibration switches the bridge off at once and enables all three phases again. An E-stop ends a calibration, and `#Cal` is refused while one is latched. Calibrating many times without a power cycle no longer uses up memory.

Safety

- The control loop keeps the gate driver awake in every driving mode: a command arriving at the wrong moment could leave it asleep, with no torque and no error, until the next idle.

- Limits: a negative or not-a-number value is ignored by the legacy CAN frames (limits, gains, Kt, voltage limit 0-60000 mV); `#Ilim` and `#Vlimit` take the magnitude; the current of a gripper move stays within 0-2800 mA; `#PP` and parameter 78 take 1-49 pole pairs, only while idle (50 and more would read as a STEPFOC calibration at the next start). PWM duties stay between 0 and 100 %.
- The current-sense zero waits for the motor supply: started on USB before the supply, the driver used a zero about 100 mA off. It now says so in `#Info` and measures the zero by itself once the supply is there.
- `#Error` says `FAULT pin is LOW: the gate driver reports a fault` (the text said HIGH). No undervoltage error while the supply ramps up at power-up.

CAN bus

- A gripper poll right after a new move command no longer cancels it as new: object detection reports contact again (it read "at position" while pressing on an object). The same in every release before 109.
- The current and PD frames set their setpoint and mode together. Length codes above 8 are read as 8; extended (29-bit) frames are ignored.
- Watchdog: setting its time or a save starts a fresh window; a negative time means off; hold and descend no longer use the last moving loop's delay compensation.

Bug fixes

- After `#Cal`, a host that takes the driver within 3 s drops the encoder-table and anti-cogging runs. `#Encal` and `#Cogcal` start only on their own words. While such a run turns the motor, a capture is not armed and the histogram is not changed.
- A storage block whose check fails is read up to three times before its defaults are used. `#Info` says the anti-cogging is saved only while what is in RAM equals what is stored.
- The sound queue and sample buffer no longer replay old content after a stop; a pitch the driver cannot play answers `Tone: not played`.

Compatibility

- Python library **1.21 or newer** (`pip install --upgrade Spectral-BLDC`), Motor GUI **0.5.0**.
- Bench-tested for this release: calibration, current mode, restarts, the E-stop, the encoder table and anti-cogging, and the SSG-48 gripper on its 107 calibration. Not tested on the bench: an upgrade from V103 with the ST-Link, a downgrade, and on the Spectral Micro the watchdog actions, the capture and the sound (the same code passed on the STEPFOC).

Assets

- `spectral-bldc-firmware-V109.bin` — application firmware image (flash at `0x08003000`, target STM32F103CB).
- `firmware.json` — manifest with version, size, and SHA-256 for download verification by the web flasher.

Integrity

- SHA-256: `96945eec70be12f0e74c6add5fd9fafd4054677e4ebf29d0380062c77a4f6d87`
- Size: 112420 bytes

Built from commit b5cedd5 on `Bootloader_main` (`#Info` shows it as `Build: B5CEDD5`). Includes bootloader entry support (CAN/UART reset-into-bootloader).

V107 (2026-08-26)

Spectral BLDC firmware **V107** — application image for flashing via the Spectral bootloader.

CAN bus

- Fixed `IN_DATA_PACK_PD` writing the commanded speed into `Feedforward_speed` instead of `Velocity_setpoint` — PD mode's KD (velocity-damping) term was being evaluated against a stale/zero setpoint, so it silently ignored the commanded speed.
- Added a generic mask+value settings command (`IN_SETTINGS`) for CAN-unreachable boolean settings, starting with idle-mode brake/coast selection.

Usability

- Idle mode can now be set to brake (driver stays enabled, phases held shorted via a zero-voltage PWM vector) or coast (Hi-Z, previous/default behavior), via `IN_SETTINGS` over CAN or `#Brake <0|1>` over UART.

Calibration

- Fixed `#Cal` spuriously failing with "Commutation/alignment check failed!" on some individual motor units. The pre-check that confirms the motor responds to a current command was driving a fixed voltage at the raw (uncalibrated) encoder angle; whether that produced torque depended on each unit's random magnet-to-encoder mounting angle. It now retries once at a 90-degree angle bias before failing.

Assets

- `spectral-bldc-firmware-V107.bin` — application firmware image (flash at `0x08003000`, target STM32F103CB).
- `firmware.json` — manifest with version, size, and SHA-256 for download verification by the web flasher.

Integrity

- SHA-256: `2ca1f2e3ad3ee7848cf47d8733b084be2daa87fc2e49f954dd41c76405d48ada`
- Size: 95404 bytes

Built from commit `2e7febc` on `Bootloader_main`.

V106 (2026-07-06)

Spectral BLDC firmware **V106** — application image for flashing via the Spectral bootloader.

CAN bus - The controller now drains up to 8 CAN messages per pass instead of just 1, so short bursts (multiple commands sent back-to-back, or a busy multi-node bus) no longer silently drop messages once the 3-deep hardware FIFO fills. - A CAN peripheral that fails to initialize no longer bricks the board — UART stays fully usable, only the CAN bus is skipped.

Safety - Added a hardware watchdog (100ms). If the control loop ever stops running, the board resets instead of leaving the motor driven at a stale duty cycle. - Fixed a bug where `Torque_mode` could silently overwrite your commanded Iq setpoint when clamping to the current limit — reading it back after clamping now shows what you actually asked for, and raising the current limit afterward immediately restores the original commanded torque instead of requiring the command to be resent. - Fixed undefined behavior in a serial-protocol integer conversion (`fourBytesToInt`). - Synced default configuration values that had drifted between two locations in the code, so a fresh `#Default` and a freshly-flashed board's built-in defaults now agree.

EEPROM / configuration integrity - Added CRC validation over the saved configuration, split into two independent halves: calibration data (resistance, inductance, Kt/KV, angle offset, current-loop gains, etc.) and settings (PID tuning, limits, watchdog config, CAN ID, etc.). - This means a corrupted save (e.g. from a power loss mid-write) is detected and safely defaulted instead of silently running with garbage values — and critically, a future firmware update that only adds a new setting can no longer wipe your calibration, since the two are validated independently. - Boards upgrading from older, pre-CRC firmware keep their existing calibration automatically — no need to recalibrate after this update. - A few unused slots were reserved in the calibration block for upcoming features (e.g. magnet alignment), so those can be added later without another migration.

Performance - PWM output no longer re-resolves which timer/channel it's writing to on every single call (it was doing this 4x per control tick) — resolved once and cached, freeing up ISR headroom. - Velocity calculation switched from a floating-point divide to an integer multiply.

Usability - New `#Help` command — prints a summary of the available commands. - Serial now responds with `"Unknown command"` for unrecognized input instead of silently ignoring it. - Calibration now prints progress during the angle-offset search (previously up to ~40 seconds of silence that could look like the board had hung). - `#Info` now always reports live ISR execution time, useful for checking control-loop headroom on the bench. - Fixed a leftover calibration-failure message that incorrectly suggested swapping motor phases — that failure mode no longer exists (reversed wiring has been auto-detected/compensated in software since V105); the message now accurately describes what actually happened (a commutation/current-response check, not a wiring-direction problem).

Assets - `spectral-bldc-firmware-V106.bin` — application firmware image (flash at 0x08003000, target STM32F103CB). - `firmware.json` — manifest with version, size, and SHA-256 for download verification by the web flasher.

Integrity - SHA-256: `edb06ca112a7d24b0581e0b157a4524928067a46f8dc850ce22df8128092a758` - Size: 94908 bytes

Built from commit `e1f8611` on `Bootloader_main`.

V105 (2026-07-05)

Spectral BLDC firmware **V105** — reversed-wiring auto-calibration

`#Cal` now auto-detects reversed motor phase wiring during the open-loop spin and compensates it in software as a phase B<->C reflection, instead of failing and telling you to physically swap phases. Angle-offset calibration is now found automatically as part of `#Cal` itself (deterministic rotor-lock + verify + symmetry-sweep), so the standalone `#Calangle` command has been removed. `#Info` now reports the true firmware version instead of a stale EEPROM value. Existing calibrated units are unaffected and do not need `#Default`.

Assets - `spectral-bldc-firmware-V105.bin` — application firmware image (flash at 0x08003000, target STM32F103CB). - `firmware.json` — manifest with version, size, and SHA-256 for download verification by the web flasher.

Integrity - SHA-256: `0bd80ea038b8497fca8ab6a613b083d28e3473b360fa152b512b8ab0304dedb5` - Size: 91732 bytes

Built from commit `b7dded3` on `Bootloader_main`. Includes reversed-wiring auto-calibration (`commutation_dir` detection + phase B<->C reflection) and deterministic rotor-lock angle-offset calibration.

V104 (2026-06-19)

Spectral BLDC firmware **V104** — bootloader build.

Assets

- `spectral-bldc-firmware-V104.bin` — firmware image (90,968 bytes).
- `firmware.json` — manifest with version, size, and SHA-256.

Integrity

- SHA-256: `76d70a6091218d74021eeef55c31d39f19fe97f68e2b236169f84c708518a360`
- Size: 90968 bytes

Built from commit 20765c9 on `Bootloader_main`. Includes CAN Vlim command and bootloader support.

v101 (2025-09-29)

Some small fixes added based on STEPFOC code that is a clone of this code.

1. Added theta_offset calib.
2. Fixed some issues related to gripper mode where it vibrates

NOTE added another entry to EEPROM memory. Loading this code to update your previous code you will need to run #Default to load that new entry in EEPROM!

v100 (2024-03-27)

Software version 100

Versions and compatibility

Which firmware release works with which bootloader, Python library, Motor GUI, PAR6 runtime and board revision. One row per firmware release, newest first.

Firmware	Released	Bootloader	STEPFOC equivalent	Python library (Spectral-BLDC)	Motor GUI	PAR6 runtime	Spec Micr PCB
110	2026-11-13	required	110	1.22 or newer (names for parameters 90 to 93; 1.21 reads them by their numbers)	0.7.0 or newer (STOP sends #Estop over UART, presets write parameters 90 to 92 over CAN)	0.9 or newer (it sends watchdog action 0, the legacy response)	v2.0
109	2026-11-10	required	109 (from 109 the two share their version numbers; there is no Spectral Micro 108)	1.21 or newer (the calls for the commands added in 109: parameters, identity, telemetry, buffers, the watchdog's coast and descend)	0.5.0 or newer	0.9 or newer (it sends watchdog action 0, the legacy response)	v2.0
107	2026-08-26	required	108 (STEPFOC 107 has the Send_Settings and PD changes of this release, without its calibration fix)	1.20 or newer		0.9 or newer	v2.0
106	2026-07-06	required	106	1.19 or newer		0.9 or newer	v2.0

Firmware	Released	Bootloader	STEPFOC equivalent	Python library (Spectral-BLDC)	Motor GUI	PAR6 runtime	Spe Mic PCB
105	2026-07-05	required	105	1.19 or newer		0.9 or newer	v2.0
104	2026-06-19	required (the first release built for the bootloader, from Bootloader_main)	none (STEPFOC has no 104 release)	1.19 or newer (1.20 for Send_Voltage_Limit)		0.9 or newer	v2.0
101	2025-09-29	none (flashed with a programming adapter)	103	1.17 or newer		0.9 or newer	v2.0

`#Info` prints the firmware release a driver runs; `Send_Respond_Device_Info` returns it over CAN.

Hardware revisions

What changed in each revision of the Spectral Micro board, and which firmware and which revision of this manual go with it. The board reports its hardware revision in `#Info` (`Hardware version`) and in Send_Respond_Device_Info; the silkscreen on the bottom of the board carries the board's name and date.

Revision	Marked	Changes	Firmware	Manual
v2.0	Spectral motor controller micro v2.0, 10.1.2024	the production board	101 and newer; 104 and newer run on the bootloader	this manual

A board that runs firmware 101 or older has no bootloader; Installing the bootloader on an older board brings it to the current firmware.

Theory corner

The control concepts behind the driver's modes, for readers who want to know why a setting does what it does. Nothing here is needed to use the driver; the chapters say what to type.

Field-oriented control of a brushless motor

A brushless motor has three stator windings and a rotor of permanent magnets with p pole pairs: one mechanical revolution is p electrical cycles, 11 for a typical 5208 gimbal motor.

The torque the rotor receives depends on the angle between the stator field and the rotor field, the *load angle*; it is largest when the stator field stands a quarter of an electrical cycle ahead of the rotor and zero when the two are aligned.

A hobby ESC does not measure the rotor angle: it steps the field around in six fixed states and infers the rotor's position from the back-EMF of the winding it is not driving, which works at speed and fails at standstill. That is why an ESC cannot hold a position or produce a controlled torque at low speed.

Field-oriented control (FOC) measures the rotor angle instead. The encoder on the motor shaft gives the mechanical angle θ; with the pole pairs p and the angle offset θ_0 found by calibration, the electrical angle is $\theta_e = p\,\theta + \theta_0$.

The three phase currents, which sum to zero, are reduced to two (the Clarke transform) and then expressed in the rotor's frame of reference (the Park transform):

$$I_\alpha = I_U, \qquad I_\beta = \frac{I_U + 2\,I_V}{\sqrt{3}}$$

$$I_d = I_\alpha \cos\theta_e + I_\beta \sin\theta_e, \qquad I_q = -I_\alpha \sin\theta_e + I_\beta \cos\theta_e$$

I_q, the *quadrature* current, is the part of the current that stands at right angles to the rotor field and produces torque: $\tau = K_t\,I_q$. I_d, the *direct* current, is aligned with the field and produces none. Two PI controllers regulate I_q to the torque demand and I_d to zero by choosing the voltages U_q and U_d.

The inverse transforms turn those into three phase voltages, applied by PWM at 25 kHz with space-vector modulation, which gets about 15 % more phase voltage out of the supply than a plain sine would. The loop runs at 5 kHz.

The consequences are what makes a FOC drive behave like a servo:

- the current is only what the torque needs, so the motor runs cool and quiet;
- the field is always placed where it produces the most torque per ampere;
- the motor produces its full torque at standstill and holds a position;
- the torque is known from the current, which is what makes impedance control and force-limited motion possible.

Calibration supplies the constants this needs. The resistance R and inductance L of a winding are measured. The current loop gains follow from them and the chosen bandwidth $\omega_c = 2\pi f_c$ (`#Cbw`): $K_p = L\,\omega_c$ and $K_i = R\,\omega_c$ cancel the winding's electrical time constant and give a first-order response at f_c.

The driver stores them in the form its 5 kHz loop uses: `Kpiq` is close to $L\,\omega_c$, and `Kiiq` is $R\,\omega_c T$, with T the 200 µs control period. The pole pairs come from an open-loop spin: the field is turned a known number of electrical cycles and the encoder says how far the rotor went.

The angle offset is found by driving a current along the d axis and reading where the rotor settles. The velocity constant K_V comes from the back-EMF the spinning motor generates, and the torque constant follows from it: in SI units $K_t = 1/K_V$. The firmware computes $K_t = 8.27/K_V$ Nm/A, with K_V in rpm/V.

The cascade of loops

Position, velocity and current are controlled by three loops nested inside one another:

$$\text{position loop: } \omega_d = K_{pp}\,(\theta_d - \theta) \qquad \text{velocity loop: } I_{q,d} = K_{pv}\,(\omega_d - \omega) + K_{iv}\int (\omega_d - \omega)\,dt \qquad \text{currer}$$

The position loop is proportional: its output is a speed demand, limited by the velocity limit. The velocity loop is a PI controller: its output is a current demand, limited by the current limit; the integral term removes the steady-state error a load would otherwise leave. The current loop is the innermost and fastest.

Each loop must be slower than the one inside it, by a factor of five to ten, for the inner loop to look like an ideal actuator to the outer one.

That is why the loops are tuned from the inside out, and why a current loop bandwidth of 160 Hz (the default from firmware 109) leaves room for a velocity loop of a few tens of hertz and a position loop below that.

A command picks where the cascade is entered: a current setpoint uses the current loop alone, a speed setpoint the two inner loops, a position setpoint all three.

The *feedforward* terms the CAN motion frames carry are added to the demands rather than computed from an error: a speed feedforward is added to the position loop's output, a current feedforward to the velocity loop's output.

A host that knows the trajectory can command it directly and leave the loops to correct what is left, which gives a faster, smoother response than letting the loops produce the whole demand from the error.

An integrator accumulates error over time, and keeps accumulating while the output is at its limit; the overshoot this causes when the limit is released is *windup*. The velocity loop's integrator is clamped to the current limit, and `#Rstint` resets the integrators on every new setpoint, which suits a host that sends one setpoint after another in large steps.

Impedance control

Impedance control does not prescribe a motion or a force. It prescribes the *relation* between the two: how much force the motor produces when it is displaced from its setpoint and how fast. The driver's PD mode implements a spring and a damper around the setpoint:

$$\tau = K_P\,(\theta_d - \theta) + K_D\,(\dot\theta_d - \dot\theta) + K_t\,I_{ff}$$

with the stiffness K_P in Nm/rad, the damping K_D in Nm·s/rad, and a current feedforward I_{ff} for a known load such as gravity. The torque is produced by the current loop ($I_q = \tau / K_t$), which is what makes FOC a prerequisite: a drive without the rotor angle cannot produce a commanded torque at low speed.

The driver computes the same loop in its own units. Its `KP` is in mA per encoder tick and its `KD` in mA per tick/s (0.14 and 0.0028 by default), and the result is a current in mA.

The stiffness in Nm/rad is `KP` × K_t × 2.608, and the damping in Nm·s/rad is `KD` × K_t × 2.608 (16384 ticks per turn, 1000 mA per A).

Pushed off its setpoint by an external force F, the motor yields by $\Delta\theta = \tau_{ext}/K_P$ and pushes back with that force; let go, it returns, with the damper settling the motion. A small K_P makes a soft joint, a large one a stiff joint; K_D without K_P is a pure damper.

The setpoint can move, in which case the virtual spring drags the rotor along a trajectory with a bounded force — the property legged robots, hand-guided arms and anything that touches people or an uncertain environment need.

The impedance loop is also what a position loop is *not*: the position loop's integral action removes any steady-state deflection and so makes the joint as stiff as the current limit allows, which is the right thing for a machine tool and the wrong thing for a leg.

The full impedance model includes an inertia term, $M\ddot\theta$; the driver, like most joint controllers, leaves it out, so the apparent inertia is the motor's own.

Admittance control

Admittance is the inverse relation: the controller measures the force and commands a motion,

$$\dot\theta_d = \frac{\tau_{ext} - K\,(\theta - \theta_0)}{B}$$

on top of a stiff position controller. It is the natural form for a machine that has a force sensor and a position loop that cannot be made soft, such as an industrial arm; impedance control is the natural form for a machine whose actuator is a torque source, such as a FOC motor.

The driver has no force sensor and implements impedance; a host that wants admittance behaviour can estimate the external torque from $K_t I_q$ minus the known load and command positions accordingly, with the limits of that estimate (friction, cogging, the current loop's accuracy).

Compliance and stiffness

Stiffness K is the force per unit of displacement; compliance is its inverse, $C = 1/K$, the displacement per unit of force. *Passive* compliance is in the mechanism — a spring, a flexible finger, a belt; *active* compliance is in the controller — the impedance loop above.

The driver offers three points on that scale: the position mode, stiff up to the current limit; the PD mode, as stiff as K_P; and the current mode, which is no stiffness at all, a torque source the host shapes as it likes.

The current limit `#Ilim` is a force limit in every mode: the largest torque the motor will produce before it yields, whatever the loops ask. Set it for the motor's rating, and for the mechanism's.

Which mode

Mode	Behaviour under an external force	Use
Position	holds the position up to the current limit, then yields	positioning, machines, anything that must not deflect
Velocity	holds the speed up to the current limit	wheels, conveyors, spindles
PD (impedance)	deflects in proportion to the force, returns when released	legs, arms that touch things or people, hand guiding, teleoperation
Current (torque)	produces the commanded torque, moves as the load dictates	a host that closes its own loop, gravity compensation, force-limited pressing

Resources

Glossary

Activated

Two flags with the same name. The controller's flag is set by `#Activate` and reported in Send_Respond_State_of_Errors; nothing else uses it. The gripper's activate bit is set by `#Gripact`, `#Grippos` and Send_gripper_data_pack; in gripper mode a move is carried out only while it is set.

Angle offset (theta offset)

The angle between the encoder's zero and the motor's electrical zero, found by calibration and stored. The driver needs it to place the current vector relative to the rotor.

Anti-cogging

A current added to the current setpoint, a function of the electrical angle, that cancels the motor's own torque ripple (the pull of its magnets towards the stator's teeth and the ripple of current-sensing errors). Identified on the driver (`#Cogcal`, parameter 43; firmware 109 and newer).

Back-EMF

The voltage a turning motor generates in its own windings, proportional to speed. It opposes the supply, limits the top speed at a given voltage, and is what calibration measures to find KV, Kt and the flux linkage.

Board ID

The bootloader's own address, 0 to 13, stored in its configuration page. The firmware sets it to the CAN node ID when the configuration is saved. The `Board ID` line of `#Info` (parameter 86, firmware 109 and newer) is another thing: 8 hex digits from the chip's unique number, which never change.

Bootloader

The program in the first 12 KB of the driver's flash on boards that run firmware 104 or newer. It starts the firmware, and loads a new firmware over UART or CAN. See The bootloader.

Calibration

The routine (`#Cal`) that checks the magnet and the supply, measures the motor's resistance and inductance, computes the current loop gains, spins the motor in open loop to find the pole pairs and the phase order, aligns the encoder to the rotor field and measures Kt, KV and the flux linkage. Its result is kept with `#Save`.

CAN ID (node ID)

The address of the driver on the CAN bus, 0 to 13 (the field holds 0 to 15, but the bootloader reserves 14 and 15). It occupies bits 10 to 7 of the 11-bit CAN frame identifier; the command ID occupies bits 6 to 1 and the error bit is bit 0.

CANvas

Source Robotics' USB-to-CAN adapter. It appears as a serial port and speaks slcan, which `python-can` and Motor GUI use.

Clarke and Park transforms

The two steps that turn the three phase currents into the two components FOC regulates: Clarke reduces the three phases to two axes fixed to the stator, Park turns those into the frame that rotates with the rotor (the d-q frame).

Command ID

The number, 0 to 63, in bits 6 to 1 of a CAN frame identifier that says which message the frame carries. The API reference lists every command ID.

Commutation

Placing the stator current so that it produces torque at the rotor's present angle. FOC does it continuously from the encoder angle and the angle offset; a hobby ESC does it in six steps from the back-EMF, which fails at standstill.

Commutation delay compensation (parameter 64)

Turning the output voltage forward by the angle the rotor covers between the angle reading and the moment the voltage reaches the motor. 234 µs on the Spectral Micro by default.

Control mode

What the driver is doing with the motor: idle, position, velocity, current, PD (impedance), voltage torque, open loop, gripper, or calibration. A command that carries a setpoint puts the driver in the matching mode.

Current loop bandwidth (`#Cbw`, parameter 49)

How fast the current loop follows a change of its setpoint, in Hz. The current loop gains follow from it and the motor's resistance and inductance: Kpiq ≈ ω·L, Kiiq = ω·R·T (T the 200 µs control period). 160 Hz by default from firmware 109 (500 Hz before).

Current-sense zero

The reading of a current sensor with no current flowing. From firmware 109 the driver measures it at every start-up and subtracts it.

Daisy chain

Several drivers on one CAN bus and one supply, each driver's second CAN and power connector feeding the next.

d-q frame

The rotating frame of reference FOC works in: the q axis is at right angles to the rotor field and the current on it (Iq) produces torque; the d axis is along the field and its current (Id) produces none, so it is held at 0.

Diametrical magnet

A disc magnet whose poles are across its diameter, so the field turns with the shaft. The encoder needs this kind; an axially magnetised disc looks the same and does not work.

EEPROM

The non-volatile memory on the driver that holds the configuration and the calibration. `#Save` writes it, `#Default` restores the factory values.

Encoder table

A correction of the encoder's small angle error around the turn, measured on the driver (`#Encal` , parameter 52) and saved with the configuration (firmware 109 and newer).

Encoder ticks

The unit of position: the 14-bit magnetic encoder gives 16384 ticks per revolution of the motor shaft, counted over as many revolutions as the motor makes. Speed is in ticks per second.

Error bit

Bit 0 of every CAN frame identifier the driver sends: 1 while any error is active.

Feedforward

A term added to a loop's output from what is known in advance rather than from the error: the motion frames carry a speed and a current feedforward so a host that knows the trajectory can command it directly, leaving the loops to correct the rest.

Field-oriented control (FOC)

Controlling a motor by regulating the stator current as a vector in the rotor's frame, from a measurement of the rotor angle. The three phase currents are reduced to a torque-producing and a field-producing component, each regulated by its own loop, which gives smooth torque at any speed down to standstill, a known position, and the most torque per ampere.

Firmware release

One build of the firmware, numbered 101, 104, 105, … and tagged `v101` , `V104` , `V105` , … in the firmware repository. `#Info` prints the release a driver runs. This documentation describes one release; the version selector switches between them.

Flux linkage

The rotor's magnetic flux as seen by a winding, in weber; found by calibration from the back-EMF, together with KV and Kt.

Gimbal motor

A brushless motor wound with many turns of thin wire for low speed and smooth torque at low current, originally for camera gimbals: a phase resistance of a few ohms to a few tens of ohms, and many pole pairs. The kind of motor the Spectral Micro is made for.

Gripper mode

The driver running a gripper (`#Gripper 1`): jaw positions from 0 (open) to 255 (closed), a speed, a current limit that becomes the grip force, and object detection.

Heartbeat

A frame (Respond_Heartbeat) the driver sends at a set interval so the host knows it is alive. Off by default.

Iq, Id

The two components of the motor current in the d-q frame: Iq produces torque, Id is normally held at 0. The current mode's setpoint and limit are Iq values.

Impedance (PD) mode

The control mode in which the driver behaves like a spring and a damper around a position setpoint, with the stiffness `KP` and the damping `KD`, plus a current feedforward.

Kt, KV

The motor's torque constant (torque per ampere) and velocity constant (speed per volt), found by calibration. Kt turns a current into an estimated torque; KV gives the top speed at a supply voltage.

Motor GUI

Source Robotics' browser tool for the driver: setup, calibration, tuning, plots and firmware updates over a serial or CAN adapter.

Open loop

Turning the stator field at a set electrical speed without the encoder in the loop, so that the rotor follows it like a stepper: `#Openloop`, and the open-loop spin of the calibration that finds the pole pairs.

Parameter

A setting reached by its number through Send_Parameter and Send_Get_Parameter (CAN 36 to 38) or `#Par` (firmware 109 and newer). The numbers are permanent.

Phase order

The order in which the motor's three phases are connected to U, V and W. Firmware 105 and newer learns it during calibration and compensates it; firmware 101 asks for two wires to be swapped when it is reversed.

PIN (ADDITIONAL)

The spare pin of the driver, on the three-pad header next to the UART connector, usable as an analog input or a digital input/output. The firmware's name for it is ADDITIONAL.

PLL (speed estimate)

A phase-locked loop that follows the encoder count and gives the speed with less lag than a moving average (parameter 50, firmware 109 and newer; the direct PLL at 100 Hz is the default).

Pole pairs

The number of electrical cycles per mechanical revolution, half the number of rotor magnets: 7 for a typical 2804 gimbal motor, 11 for a 5208, 14 for an 8108. Calibration finds it; `#PP` sets it by hand.

Product lock

The stamp in the EEPROM of the product that calibrated a board. A firmware that finds the other product's stamp drives nothing (firmware 109 and newer).

Remote frame

A CAN frame with the RTR bit set and no data: a request. The host asks for data with a remote frame under a command ID, and the driver answers with a data frame under the same ID.

Sound (`#Tone` , CAN 44 and 45)

Tones, a queue of notes and recorded samples the driver adds to its output voltage while it holds or moves, so the motor plays them (firmware 109 and newer).

STEPFOC

Source Robotics' stepper driver, the Spectral Micro's sister: similar hardware and firmware and practically the same commands, for two-phase stepper motors.

Termination

The 120 Ω resistor the first and the last node of a CAN bus need. The switch on the driver turns it on.

Tool id (`#Tool` , parameter 1)

What a drive is built into: 0 none, 1 SSG-48, 11 to 33 an MSG, 200 to 254 a customer's tool.

Vbus

The supply voltage at the driver, measured and printed in mV. Outside the limits the driver raises a Vbus error.

Watchdog

A timer the driver resets with every motion command and status request it receives over CAN. When it is set (Send_Watchdog) and runs out, the driver reports a watchdog error and stops; from firmware 109 it holds, brakes, coasts or descends as the watchdog's action says.

Downloads and files

Everything to download for the Spectral Micro, at the versions these pages describe. The versions come from one file, `versions.yml`, so this page changes with them; which versions work together is in Versions and compatibility.

Firmware and software

What	Version	Download	How to install
Firmware image (`spectral-bldc-firmware-V<number>.bin`), its manifest `firmware.json` and the release notes	110 (2026-11-13)	the GitHub release	Updating through the bootloader
Firmware source, a PlatformIO project	the same release	the firmware folder at the tag	Building from source
The bootloader binary, for a board without one	0.1 (2026-06-05)	bin	Installing the bootloader on an older board
The bootloader's host tool `bl_flash.py`	the same release	host_tools	Updating through the bootloader
Python library (`Spectral-BLDC`)	1.22 (2026-11-13)	`pip install Spectral-BLDC`; the source and examples on GitHub, at that version	With Python
Motor GUI	runs in the browser, always the latest	motorgui.com	Motor GUI
CAN definitions for this release: JSON, and DBC files for the motion layouts	this site's release	CAN frame tool and downloads	

What	Version	Download	How to install
ST-LINK USB driver, for the programming adapter on Windows		STSW-LINK009	Installing the bootloader on an older board
STM32CubeProgrammer, for the programming adapter		STM32CubeProgrammer	Installing the bootloader on an older board

The version links open the files at exactly that release; the dates are those of the releases on GitHub, and on PyPI for the Python library. Motor GUI has no versions to choose: the page is always the current one. Known faults of the firmware are on Known issues.

CAD and hardware files

The latest files, on the main branch of the firmware repository.

What	Where
STEP model of the board and the example mounting bracket	STEP files

More

What	Where
Source Robotics Toolbox (joints from encoder ticks)	Source-Robotics-Toolbox
SimpleFOC examples for the board	Spectral-Micro-Simple-FOC
Arduino library for controlling the driver from an Arduino	SpectralMicroArduino
CANvas USB-to-CAN adapter	CANvas-adapter
All firmware releases	Releases
This documentation	Spectral-BLDC-docs

What	Where
Shop	source-robotics.com

API reference

API reference

Every CAN message and every UART command of the firmware release this site describes, the factory defaults of its settings and which of them are saved, the error flags, and the map from the Python library's calls to the messages they send.

Where the facts come from

The command IDs, the frame lengths the firmware accepts, the responses it sends, the list of UART commands with their kind (Set/Get, Get or action), their argument type, the mode each enters and what it prints, the parameter numbers, and the defaults are read from the firmware's source at its release tag by `tools/sync_firmware.py` in the documentation repository, and the pages are built from that at every build. The descriptions, units and byte layouts are written by hand, and the build fails when they name a command the firmware does not have, or miss one it has, or when a byte layout does not fit the frame. The firmware version is in the version box on the landing page.

Conventions

- **Bytes** are numbered from 0, the first byte on the bus (`data[0]`). Multi-byte values are **big-endian**: the most significant byte first. Floats are IEEE 754 single precision in 4 bytes.
- **Bits** inside a byte are numbered from the most significant bit: bit 7 is the first flag the firmware packs into the byte, bit 0 the last.
- **Encoder ticks**: 16384 per revolution; speeds in ticks per second; currents in mA; voltages in mV.
- UART command names are **case-sensitive** and spelled exactly as the firmware compares them.

In the API reference

- CAN protocol: the identifier, the three kinds of message, every message by group, and every parameter
- UART commands: the syntax and every command by group
- Defaults
- Errors and LED states
- Python library: library call → message

CAN frame tool and downloads

Build or inspect a frame using the command definitions of this manual's firmware release. All calculations run in your browser; the tool does not connect to an adapter or send commands to hardware.

Downloads

Command definitions (JSON) · Position DBC · Velocity DBC · Current DBC

The JSON includes the firmware tag, source commit, message IDs, every documented layout and its bit order. The DBC files cover **nodes 0 to 13**, with both the clean and fault-bit identifiers of driver replies. Reserved commands are omitted. Values use the firmware's native units: ticks, ticks/s, mA and the units in the CAN reference.

Choose the DBC for the `Send_data_pack_1` layout your host sends:

File	Command layout
Position DBC	8 bytes: position, speed feedforward, current feedforward
Velocity DBC	5 bytes: speed, current feedforward
Current DBC	2 bytes: current

The three layouts share a CAN identifier. DBC cannot select a layout from the frame's length, so load the matching file; the browser decoder below handles all three lengths. Other message layouts are the same in each download.

An RTR request uses the same identifier as its data reply. DBC describes the **data reply**; it does not represent the request's RTR flag or the empty gripper status poll. Both are supported by the browser tool.

The gripper's two object-detection bits have an unusual order: bit 5 is the low bit, bit 4 the high bit. The DBC signal `gripper_object_detection_raw` retains the wire value and supplies text labels (raw 2 = object while closing, raw 1 = object while opening). The browser decoder converts that pair to the firmware's numeric status (1 = closing, 2 = opening).

Try a frame

The example `0x07A`, data `C8 1E 01 2C C0`, is a command to gripper node 0: position 200, speed 30, current 300 mA, activate and action set, E-stop and release direction clear. These are an illustration of the encoding; the device's current and motion limits still apply. Select another message above the fields to build one of its documented forms. To inspect a reply, enter its identifier and data bytes below.

Check

The example decodes to the values above. Change the identifier to `0x1FA` to see the same command addressed to node 3. An invalid length or out-of-range field produces an explanation instead of a frame.

For wiring, termination, host timing and recovery, use Over CAN. To look up a message's purpose and accepted lengths, use the CAN reference.

CAN messages

CAN protocol

CAN 2.0, 1 Mbit/s, 11-bit identifiers. The identifier carries the node ID (bits 10 to 7, 0 to 15), the command ID (bits 6 to 1, 0 to 63) and, in frames from the driver, the error bit (bit 0). Over CAN explains how the pieces are used; this page lists them.

Three kinds of message:

- **Send_…** — the host sends data to the driver. A few are answered (the motion commands with Respond_data_pack_1, the gripper command with Respond_Gripper_data_pack); most are not.
- **Send_Respond_…** — the host asks for data with a **remote frame** (RTR bit set, no data) under the command ID; the driver answers with a data frame under the same command ID.
- **Respond_…** — frames only the driver sends.

Multi-byte values are big-endian (data[0] first); floats are IEEE 754 single precision; bits inside a byte are numbered from the most significant bit (bit 7 first).

Every message

ID	Message	Direction	Length	Group
0	Send_ESTOP	host → driver	0	System
1	Send_Clear_Error	host → driver	0	System
2	Send_data_pack_1	host → driver	2, 5, 8	Motion
3	Respond_data_pack_1	driver → host	8	Motion
4	Send_data_pack_PD	host → driver	8	Motion
5	Respond_data_pack_2	driver → host	0	Motion
6	Send_data_pack_2	host → driver	not implemented	Motion
7	Respond_data_pack_3	driver → host	0	Motion
8	Send_data_pack_3	host → driver	not implemented	Motion

ID	Message	Direction	Length	Group
9	Respond_Heartbeat	driver → host	0	Status requests
10	Send_Respond_Ping	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 0	Status requests
11	Send_CAN_ID	host → driver	1	System
12	Send_Idle	host → driver	0	System
13	Send_Save_config	host → driver	0	System
14	Send_Reset	host → driver	0	System
15	Send_Watchdog	host → driver	5	System
16	Send_PD_Gains	host → driver	8	Gains and limits
17	Send_Current_Gains	host → driver	8	Gains and limits
18	Send_Velocity_Gains	host → driver	8	Gains and limits
19	Send_Position_Gains	host → driver	4	Gains and limits
20	Send_Limits	host → driver	8	Gains and limits
21	Send_Iq	host → driver	not implemented	Motion
22	Send_Kt	host → driver	4	Gains and limits
23	Send_Respond_Temperature	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 2	Status requests
24	Send_Respond_Voltage	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 2	Status requests
25	Send_Respond_Device_Info	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 7	Status requests

ID	Message	Direction	Length	Group
26	Send_Respond_State_of_Errors	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 2	Status requests
27	Send_Respond_Iq_data	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 2	Status requests
28	Send_Respond_Encoder_data	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 8	Status requests
29	Send_Cyclic_command	host → driver	not implemented	System
30	Send_Heartbeat_Setup	host → driver	4	System
33	Send_Respond_Kt	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 4	Status requests
34	Send_Voltage_Limit	host → driver	4	Gains and limits
35	Send_Settings	host → driver	2	System
36	Send_Parameter	host → driver	5	Parameters and buffers
37	Send_Get_Parameter	host → driver	1	Parameters and buffers
38	Respond_Parameter	driver → host	5	Parameters and buffers
39	Send_Respond_Identity	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 8	Status requests
40	Send_Respond_Telemetry	host → driver as a remote frame, driver → host as a data frame	0 (remote) / 8	Status requests
41	Send_Buffer_Read	host → driver	5	Parameters and buffers

ID	Message	Direction	Length	Group
42	Respond_Buffer_Data	driver → host	8	Parameters and buffers
43	Send_Buffer_Write	host → driver	8	Parameters and buffers
44	Send_Sound	host → driver	8	Sound
45	Send_Sound_Samples	host → driver	1 to 8	Sound
60	Respond_Gripper_data_pack	driver → host	4	Gripper
61	Send_gripper_data_pack	host → driver	0, 5	Gripper
62	Send_gripper_calibrate	host → driver	0	Gripper

By group

- Motion: the motion commands and their answers
- Gains and limits: the loop gains, the limits, Kt
- Status requests: temperature, voltage, device info, errors, current, encoder, Kt, identity, telemetry, the heartbeat
- System: node ID, E-stop, idle, save, reset, clear, watchdog, heartbeat setup, settings
- Parameters and buffers: the parameters behind Send_Parameter and Send_Get_Parameter, every one of them, and the buffers: capture, histogram, anti-cogging, encoder table
- Sound: tones, the queue and samples through the motor
- Gripper: the gripper command, calibration and the gripper state

The Python library

Every message the host can send has a method of the same name in `Spectral_BLDC.SpectralCAN`, listed in Python library.

Tools and downloads

Build or decode a CAN frame, and download this firmware's JSON and DBC definitions.

Motion

The commands that move the motor and the frames that come back.

Send_data_pack_1

ID 2 | host → driver | 2 or 5 or 8 bytes | Python: `Send_data_pack_1()`

The motion command. Its length decides the control mode the driver enters: 8 bytes command a position with a speed and a current feedforward, 5 bytes a speed with a current feedforward, 2 bytes a current. Send 0 for a feedforward you do not use. A frame of any other length is ignored (the driver flags it as a wrong data length) but still answered.

8 bytes — position setpoint with speed and current feedforward (position mode)

Byte	Field	Type	Meaning
0-2	Position setpoint	signed 24-bit	encoder ticks
3-5	Speed feedforward	signed 24-bit	encoder ticks/s
6-7	Current feedforward	signed 16-bit	mA

5 bytes — speed setpoint with current feedforward (velocity mode)

Byte	Field	Type	Meaning
0-2	Speed setpoint	signed 24-bit	encoder ticks/s
3-4	Current feedforward	signed 16-bit	mA

2 bytes — current setpoint (current mode)

Byte	Field	Type	Meaning
0-1	Current setpoint	signed 16-bit	mA

The driver answers with **Respond_data_pack_1**.

Every form resets the watchdog. With `Rstint` set, the integral accumulators are cleared on each new setpoint. From firmware 109 a position or speed frame that follows another mode (idle, current, PD, a watchdog trip) starts the velocity integral from zero, so a release from a wound-up mode does not kick; one that follows position or velocity mode keeps it. The new setpoint, the mode change and the driver's wake-up reach the control loop together, so the first frame after Idle is followed.

Send_data_pack_PD

ID 4 | host → driver | 8 bytes | Python: `Send_data_pack_PD()`

Puts the driver in impedance (PD) mode and sets the position, speed and current feedforward of the PD loop, whose gains are KP and KD (Send_PD_Gains).

8 bytes

Byte	Field	Type	Meaning
0-2	Position setpoint	signed 24-bit	encoder ticks
3-5	Speed setpoint	signed 24-bit	encoder ticks/s
6-7	Current feedforward	signed 16-bit	mA

The driver answers with **Respond_data_pack_1**.

Firmware 106 and older wrote the speed into the feedforward instead of the PD loop's speed setpoint, so the speed term had no effect; 107 fixed it.

Respond_data_pack_1

ID 3 | driver → host | 8 bytes

The driver's answer to Send_data_pack_1 and Send_data_pack_PD: where the motor is, how fast it turns and the current it draws. The Python library stores them as `position`, `speed` and `current`.

The frame (8 bytes)

Byte	Field	Type	Meaning
0-2	position	signed 24-bit	encoder ticks
3-5	speed	signed 24-bit	encoder ticks/s

Byte	Field	Type	Meaning
6-7	current	signed 16-bit	Iq in mA

Send_data_pack_2

ID 6 | host → driver | not implemented

Reserved. The firmware has no handler for this ID and the Python library's call does nothing.

Send_data_pack_3

ID 8 | host → driver | not implemented

Reserved. The firmware has no handler for this ID and the Python library's call does nothing.

Respond_data_pack_2

ID 5 | driver → host | 0 bytes

Reserved; the driver never sends it.

Respond_data_pack_3

ID 7 | driver → host | 0 bytes

Reserved; the driver never sends it.

Send_Iq

ID 21 | host → driver | not implemented

Reserved; the firmware has no handler for this ID. Use the 2-byte form of Send_data_pack_1 for a current setpoint.

Gains and limits

The commands that set the loop gains and the limits. None of them is answered; save with Send_Save_config.

Send_PD_Gains

ID 16 | host → driver | 8 bytes | Python: `Send_PD_Gains()`

Sets KP and KD, the gains of the impedance (PD) loop. Keep them with Send_Save_config.

8 bytes

Byte	Field	Type	Meaning
0-3	KP	float 32	
4-7	KD	float 32	

The driver does not answer this command.

Send_Current_Gains

ID 17 | host → driver | 8 bytes | Python: `Send_Current_Gains()`

Sets Kp and Ki of the PI current loop of Iq. The Id loop keeps its gains; over UART, `#Kpiq` and `#Kiiq` set both loops, and so do parameters 11 and 12.

8 bytes

Byte	Field	Type	Meaning
0-3	Kp_iq	float 32	
4-7	Ki_iq	float 32	

The driver does not answer this command.

Send_Velocity_Gains

ID 18 | host → driver | 8 bytes | Python: `Send_Velocity_Gains()`

Sets Kp and Ki of the PI velocity loop.

8 bytes

Byte	Field	Type	Meaning
0-3	Kpv	float 32	
4-7	Kiv	float 32	

The driver does not answer this command.

Send_Position_Gains

ID 19 | host → driver | 4 bytes | Python: `Send_Position_Gains()`

Sets Kp of the P position loop.

4 bytes

Byte	Field	Type	Meaning
0-3	Kpp	float 32	

The driver does not answer this command.

Send_Limits

ID 20 | host → driver | 8 bytes | Python: `Send_Limits()`

Sets the speed limit and the current limit the loops may reach. From firmware 109 a negative value, or one that is not a number, is ignored, and the current limit stops at the driver's 2800 mA, as with `#Ilim`. Firmware 107 and older stored the current limit as it arrived. Never send more than the motor's rating.

8 bytes

Byte	Field	Type	Meaning
0-3	Speed limit	float 32	encoder ticks/s
4-7	Current limit	float 32	mA

The driver does not answer this command.

Send_Kt

ID 22 | host → driver | 4 bytes | Python: `Send_Kt()`

Sets the motor's torque constant Kt, which the driver uses to estimate torque from current.

4 bytes

Byte	Field	Type	Meaning
0-3	Kt	float 32	Nm/A

The driver does not answer this command.

Send_Voltage_Limit

ID 34 | host → driver | 4 bytes | Python: `Send_Voltage_Limit()`

Sets the voltage limit of the current loops (the same value as `#Vlim`), in mV. 0 means use the supply voltage. In the 104 release image (the source at tag V104 does not have it yet) and in every release from 105. From firmware 109 it takes 0 to 60000 mV, as parameter 17 does; a value outside is ignored.

4 bytes

Byte	Field	Type	Meaning
0-3	Voltage limit	signed 32-bit	mV; 0 = Vbus

The driver does not answer this command.

Status requests

Data the host asks for with a remote frame (RTR bit set, no data); the driver answers with a data frame with the same command ID. And the heartbeat, which the driver sends on its own.

Send_Respond_Ping

ID 10 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Ping()`

Asks the driver to answer with an empty frame, to see that it is on the bus.

The frame (0 bytes) carries no data.

The driver answers with **Send_Respond_Ping**.

Send_Respond_Temperature

ID 23 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Temperature()`

Asks for the thermistor temperature. Meaningful only with the thermistor enabled (`#Term 1`).

The frame (2 bytes)

Byte	Field	Type	Meaning
0-1	temperature	signed 16-bit	°C

The driver answers with **Send_Respond_Temperature**.

Send_Respond_Voltage

ID 24 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Voltage()`

Asks for the supply voltage.

The frame (2 bytes)

Byte	Field	Type	Meaning
0-1	voltage	unsigned 16-bit	mV

The driver answers with **Send_Respond_Voltage**.

Send_Respond_Device_Info

ID 25 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Device_Info()`

Asks for the driver's identity: hardware version, batch date and serial number from the EEPROM, and the firmware release (from firmware 105 on this is the build's own version; before, a value from the EEPROM).

The frame (7 bytes)

Byte	Field	Type	Meaning
0	HARDWARE_VERSION	unsigned 8-bit	PCB revision
1	BATCH_DATE	unsigned 8-bit	production batch, the lowest 8 bits of the stored value (24092024 after #Default is sent as 120)
2	SOFTWARE_VERSION	unsigned 8-bit	firmware release, e.g. 107
3-6	SERIAL_NUMBER	unsigned 32-bit	serial number

The driver answers with **Send_Respond_Device_Info**.

Send_Respond_State_of_Errors

ID 26 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_State_of_Errors()`

Asks for every error flag and the calibrated, activated, watchdog and product-lock states.

The frame (2 bytes)

Byte	Bit	Field	Type	Meaning
0	7	error	bit	any error is active
0	6	temperature_error	bit	motor over the temperature limit
0	5	encoder_error	bit	magnet not detected or misaligned
0	4	vbus_error	bit	supply voltage outside the limits
0	3	driver_error	bit	the MOSFET driver reports undervoltage, thermal shutdown or overcurrent
0	2	velocity_error	bit	speed over the limit
0	1	current_error	bit	reserved; never set by this firmware
0	0	estop_error	bit	Send_ESTOP was received (from firmware 110 also #Estop over UART)
1	7	calibrated	bit	the motor is calibrated
1	6	activated	bit	the controller is activated
1	5	watchdog_error	bit	the watchdog ran out
1	4	wrong_product	bit	from firmware 109: this firmware is on the other product's board (the product lock) and drives nothing

The driver answers with **Send_Respond_State_of_Errors**.

Send_Respond_Iq_data

ID 27 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Iq_data()`

Asks for the measured Iq current.

The frame (2 bytes)

Byte	Field	Type	Meaning
0-1	current	signed 16-bit	mA

The driver answers with **Send_Respond_Iq_data**.

Send_Respond_Encoder_data

ID 28 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Encoder_data()`

Asks for the position and the filtered speed, as 32-bit values. Python attributes `position` and `speed`.

The frame (8 bytes)

Byte	Field	Type	Meaning
0-3	position	signed 32-bit	encoder ticks
4-7	speed	signed 32-bit	encoder ticks/s

The driver answers with **Send_Respond_Encoder_data**.

Send_Respond_Kt

ID 33 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Kt()`

Asks for the torque constant Kt the driver uses. Added in firmware 104.

The frame (4 bytes)

Byte	Field	Type	Meaning
0-3	Kt	float 32	Nm/A

The driver answers with **Send_Respond_Kt**.

Send_Respond_Identity

ID 39 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Identity()`

Asks what the drive is: the product, the tool it is built into, the firmware release, the hardware version, the pole pairs, whether it is calibrated, and eight flags. A host can check it before it commands or flashes a drive. Added in firmware 109. Python attributes `product_id`, `product_name`, `tool_id`, `firmware_version`, `identity_hardware_version`, `pole_pairs`, `identity_calibrated`, `identity_flags`.

The frame (8 bytes)

Byte	Bit	Field	Type	Meaning
0		product	unsigned 8-bit	1 STEPFOC, 2 Spectral Micro: the firmware that answers
1		tool_id	unsigned 8-bit	parameter 1: 0 none, 1 SSG-48, 11-33 MSG, 200-254 your own
2-3		firmware_version	unsigned 16-bit	the release, e.g. 109
4		hardware_version	unsigned 8-bit	PCB revision from the EEPROM
5		pole_pairs	unsigned 8-bit	the motor's pole pairs; 0 if outside 0-255
6		calibrated	unsigned 8-bit	1 calibrated, 0 not
7	0	gripper	bit	the driver is set up as a gripper (#Gripper 1)
7	1	brake_on_idle	bit	idle brakes (1) or coasts (0)
7	2	commutation_reversed	bit	calibration found the coils wired the other way round
7	3	settings_block2	bit	the EEPROM has the bank settings block 2 lives in, so parameters 1-5 can be saved
7	4	wrong_product	bit	this firmware is on the other product's board (the product lock) and drives nothing
7	5	sense_zero_measured	bit	the current-sense zero was measured at start-up and is plausible

Byte	Bit	Field	Type	Meaning
7	6	cogging_saved	bit	the anti-cogging in use is the one saved in the EEPROM
7	7	switching_zero_in_use	bit	the current-sense zero measured with the bridge switching is in use (parameter 46)

The driver answers with **Send_Respond_Identity**.

The flags in byte 7 are numbered the usual way, bit 0 the least significant (unlike the error bytes, which the firmware packs from bit 7 down).

Send_Respond_Telemetry

ID 40 | host → driver as a remote frame, driver → host as a data frame | remote frame, 0 bytes

Python: `Send_Respond_Telemetry()`

Asks for the temperature, the supply voltage, both error bytes and the current in one frame, all from the same moment: one request instead of four. Added in firmware 109. The Python library fills the same attributes as the four separate answers.

The frame (8 bytes)

Byte	Field	Type	Meaning
0-1	temperature	signed 16-bit	°C (meaningful with the thermistor enabled)
2-3	voltage	signed 16-bit	supply voltage in mV
4	errors_1	unsigned 8-bit	the error flags, as byte 0 of Send_Respond_State_of_Errors
5	errors_2	unsigned 8-bit	calibrated, activated, watchdog, wrong product, as byte 1 of Send_Respond_State_of_Errors
6-7	current	signed 16-bit	Iq in mA

The driver answers with **Send_Respond_Telemetry**.

Respond_Heartbeat

ID 9 | driver → host | 0 bytes

The frame the driver sends on its own at the heartbeat interval (Send_Heartbeat_Setup); empty. Off by default. Any other frame the driver sends resets the heartbeat timer, so a driver that answers motion commands at a high rate never sends a heartbeat.

The frame (0 bytes) carries no data.

System

Identity, saving, resetting, errors, the watchdog, the heartbeat and the settings.

Send_CAN_ID

ID 11 | host → driver | 1 bytes | Python: `Send_CAN_ID()`

Gives the driver a new node ID, 0 to 13 (the firmware accepts up to 15, but the bootloader reserves 14 and 15: a driver on one of those cannot be updated over CAN). The driver answers to the new ID at once. The ID is kept over a power cycle only after Send_Save_config, and saving is also what copies it into the bootloader's board ID.

1 bytes

Byte	Field	Type	Meaning
0	ID	unsigned 8-bit	0 to 13 (14 and 15 are accepted but reserved by the bootloader)

The driver does not answer this command.

Send_ESTOP

ID 0 | host → driver | 0 bytes | Python: `Send_ESTOP()`

Stops the motor and raises the E-stop error, which stays until Send_Clear_Error.

The driver does not answer this command.

Send_Idle

ID 12 | host → driver | 0 bytes | Python: `Send_Idle()`

Puts the driver in idle mode: the motor coasts, or brakes when Send_Settings or `#Brake 1` asked for it.

The driver does not answer this command.

Send_Save_config

ID 13 | host → driver | 0 bytes | Python: `Send_Save_config()`

Writes the current configuration (gains, limits, calibration, CAN ID, heartbeat, watchdog, gripper flag) to the EEPROM, and copies the CAN ID into the bootloader's board ID. From firmware 109 it also writes the tool id and the watchdog fallback (settings block 2), the anti-cogging set with its mode and scale, and the encoder table with its on/off flag, parameter 56 and the current-sense zero trims, each block with its own checksum; of the two large blocks only the 16-byte pages that changed are written. From firmware 110 it also writes the idle behaviour of Send_Settings (parameter 18). Not kept: the current loop bandwidth, and every parameter the parameter table marks as not kept. A drive stopped by the product lock writes nothing.

The driver does not answer this command.

Send_Reset

ID 14 | host → driver | 0 bytes | Python: `Send_Reset()`

Resets the driver. From firmware 104 the reset lands in the bootloader, which waits 2 s for a flashing host and then starts the firmware again; this is how a firmware update begins.

The driver does not answer this command.

Send_Clear_Error

ID 1 | host → driver | 0 bytes | Python: `Send_Clear_Error()`

Clears the error flags and goes to idle; it also ends a watchdog hold or descend. An error whose cause is still present comes straight back. A calibration error is cleared only by calibrating.

The driver does not answer this command.

Send_Watchdog

ID 15 | host → driver | 5 bytes | Python: `Send_Watchdog()`

Sets the watchdog. The motion commands (Send_data_pack_1, Send_data_pack_PD, Send_gripper_data_pack) and the requests the driver answers (the Send_Respond_ remote frames, Send_Get_Parameter, Send_Buffer_Read) reset it; the other frames (Send_Parameter among them) and UART commands do not. When the time runs out the driver reports a watchdog error and does what the action byte says. 0 turns it off (the default). From firmware 109 the driver acts on the action byte, but only when the watchdog is the only active error: any other error gets the legacy response, the motor disabled and coasting. Hold and descend run for the fallback time (parameter 2, 2 s by default), then the after-action (parameter 5); they start only if the driver was in position, velocity, current, PD or gripper mode, otherwise the driver gives the legacy response and the motor coasts. Setting the time starts a fresh window, and a negative time means off. The watchdog explains each action. The setting is kept by Send_Save_config. The same values are parameters 6 and 7, which `#Par` reaches over UART: to turn off a saved watchdog, send 0 over CAN or `#Par 6,0`, and save.

5 bytes

Byte	Field	Type	Meaning
0-3	Time	unsigned 32-bit	ms; 0 = off
4	Action	unsigned 8-bit	0 legacy (the motor disabled, coasting), 1 hold the position, 2 brake (not current-limited), 3 coast, 4 descend at the fallback current and speed; other values act as 0

The driver does not answer this command.

Firmware 107 and older store the action byte without acting on it. The Python library sends all five actions from 1.21 ('Idle', 'Hold', 'Brake', 'Coast', 'Descend'); 1.20 and older send only 0, 1 or 2.

Send_Heartbeat_Setup

ID 30 | host → driver | 4 bytes | Python: `Send_Heartbeat_Setup()`

Sets the heartbeat interval in ms; 0 turns the heartbeat off (the default). Kept by Send_Save_config.

4 bytes

Byte	Field	Type	Meaning
0-3	Interval	unsigned 32-bit	ms; 0 = off

Send_Settings

ID 35 | host → driver | 2 bytes | Python: `Send_Settings()`

Sets boolean settings that have no command of their own: a mask byte says which settings the frame touches, a value byte gives their new values; settings outside the mask are left alone. Bit 0 is the idle behaviour: 1 brake (phases shorted), 0 coast (the default). From firmware 110 Send_Save_config keeps it; on 109 the driver coasted again after a power cycle, as a driver still does that never saved it. Braking applies only in idle without an error; an error always disables the motor, which then coasts. The braking current is not limited, see Control modes. Added in firmware 107 (STEPFOC 107). The Python library has `Send_Settings(mask, value)` and, for this bit, `Send_Brake_Coast(brake)` from 1.21; 1.20 and older have no call for it.

2 bytes

Byte	Field	Type	Meaning
0	Mask	unsigned 8-bit	bit 0 = write the idle behaviour
1	Value	unsigned 8-bit	bit 0: 1 brake, 0 coast

The driver does not answer this command.

Send_Cyclic_command

ID 29 | host → driver | not implemented

Reserved. The firmware has no handler for this ID and the Python library's call does nothing.

Parameters and buffers

From firmware 109 a setting that has no command of its own is a **parameter**: a number from 1 to 93 (89 in firmware 109) behind one pair of commands, Send_Parameter (ID 36) to set it and Send_Get_Parameter (ID 37) to read it, answered with Respond_Parameter (ID 38). The identifier has room for only 64 command IDs, so new settings no longer use them up.

Parameter numbers are permanent: a number is never given a new meaning. Number 73 is not used and stays free. Parameters 84 to 87 are read only and say which build and which board answer: the git commit the firmware was built from (84), its build time in Unix seconds (85), the board ID (86) and the calibration ID (87). `#Info` prints them as `Build`, `Board ID` and `Calibration ID` (Status and diagnostics). Parameters 88 and 89, a diagnostic of when the current sensors are read, are the Spectral Micro's own; STEPFOC does not have them.

Firmware 110 adds 90 to 93: the three settings that only UART could set before, `#Rstint` (90), `#Term`, the thermistor (91) and `#Gripper`, gripper mode (92), and the gripper's stroke (93, read only). The Spectral-BLDC library 1.22 names them (`PARAM_RSTINT`, `PARAM_THERMISTOR`, `PARAM_GRIPPER_MODE`, `PARAM_GRIPPER_STROKE`); 1.21 sets and reads them by their numbers.

The **buffers** are the larger data a drive holds: a capture of three signals, a histogram, the anti-cogging set and the encoder table. Send_Buffer_Read (ID 41) reads them in 8-byte frames, Respond_Buffer_Data (ID 42) carries them, and Send_Buffer_Write (ID 43) writes the anti-cogging set.

Over UART, `#Par` does the same: `#Par` lists every parameter, `#Par 1` reads one, `#Par 1,200` sets it (Status and utility).

Setting and reading a parameter

A set is not answered: read the value back to know what the driver took. A value outside a parameter's range is clamped, replaced or refused, as the table says, and a read-only number ignores the write.

```
import Spectral_BLDC as Spectral

comms = Spectral.CanCommunication(bustype='slcan', channel='COM41', bitrate=1000000)
motor = Spectral.SpectralCAN(node_id=0, communication=comms)

motor.Send_Parameter(1, 200)              # tool id 200, a custom tool
motor.Send_Get_Parameter(1)               # read it back
message, ids = comms.receive_can_messages(timeout=0.1)
if message is not None:
    motor.UnpackData(message, ids)
print(motor.parameters.get(1))
motor.Send_Save_config()                  # keep it over a power cycle
```

`Send_Parameter`, `Send_Get_Parameter` and the buffer calls need the Spectral-BLDC library 1.21 or newer. `Send_Parameter` takes the value itself: a gain or a limit (8 to 16 and 20) is sent as a float, so `motor.Send_Parameter(16, 1500)` sets a current limit of 1500 mA; every other parameter takes a whole number.

The commands

Send_Parameter

ID 36 | host → driver | 5 bytes | Python: `Send_Parameter()`

Sets one parameter by its number (the parameter table). Firmware 109 gives new settings a parameter number instead of a command ID of their own, since the identifier has room for only 64 command IDs. A value outside a parameter's range is clamped, replaced or refused as the table says; a number the firmware does not know, or a read-only one, is ignored. The driver does not answer: read the value back with Send_Get_Parameter. A value set this way lives in RAM until Send_Save_config, like every other setting; the table says which parameters Save keeps.

5 bytes

Byte	Field	Type	Meaning
0	Parameter	unsigned 8-bit	the parameter number, 1-93
1-4	Value	signed 32-bit	the new value, in the parameter's unit; for the gains, limits and Kt (8-16, 20) a float's bit pattern

The driver does not answer this command.

A data frame of another length, or a remote frame, is ignored and flagged as a wrong data length. It does not reset the watchdog.

Send_Get_Parameter

ID 37 | host → driver | 1 bytes | Python: `Send_Get_Parameter()`

Asks for the value in force of one parameter. The answer, Respond_Parameter, comes under ID 38, not 37: a host and a driver must never send data frames with the same identifier, because two sent at the same moment collide. Resets the watchdog. The Python library stores the value in `parameters[number]`.

1 bytes

Byte	Field	Type	Meaning
0	Parameter	unsigned 8-bit	the parameter number

The driver answers with **Respond_Parameter**.

Respond_Parameter

ID 38 | driver → host | 5 bytes

The answer to Send_Get_Parameter: the number and the value in force. For a number the firmware does not know the frame carries the number alone (1 byte; the Python library sets `unknown_parameter`). The gains, limits and Kt (parameters 8 to 16 and 20) travel as the bit pattern of an IEEE-754 float, the others as integers.

The frame (5 bytes)

Byte	Field	Type	Meaning
0	parameter	unsigned 8-bit	the parameter number
1-4	value	signed 32-bit	the value in force; a float's bits for 8-16 and 20

Send_Buffer_Read

ID 41 | host → driver | 5 bytes | Python: `Send_Buffer_Read()`

Asks for entries of one of the drive's buffers: the capture, the histogram, the anti-cogging harmonics or table, or the encoder table. The driver answers with one Respond_Buffer_Data frame per entry (for the two tables, one per three values), sent from its main loop whenever a transmit mailbox is free, so a long read-out never holds up the control loop or other answers. A new request replaces one still in progress; count 0 cancels it. Entries past the end are not sent. Resets the watchdog.

5 bytes

Byte	Field	Type	Meaning
0	Kind	unsigned 8-bit	0 capture (up to 1024 samples: parameter 32 says how many), 1 histogram (256 bins), 2 anti-cogging harmonics (8 slots), 3 anti-cogging table (256 values), 4 encoder table (128 values)
1-2	First	unsigned 16-bit	the first index
3-4	Count	unsigned 16-bit	how many entries; 0 cancels a read-out

The driver does not answer this command.

Respond_Buffer_Data

ID 42 | driver → host | 8 bytes

One entry of a buffer, the answer to Send_Buffer_Read. The first two bytes say which: the kind in the top three bits, the index in the other thirteen (kind × 8192 + index). The six bytes after them depend on the kind, as the table below the frame says.

The frame (8 bytes)

Byte	Field	Type	Meaning
0-1	header	unsigned 16-bit	kind × 8192 + index
2-3	word_a	signed 16-bit	capture: channel A; histogram: the count (unsigned); harmonics: the harmonic number (0 = slot unused); tables: entry index
4-5	word_b	signed 16-bit	capture: channel B; histogram: the sum's upper 16 bits; harmonics: a, the cosine term in mA; tables: entry index + 1
6-7	word_c	signed 16-bit	capture: channel C; histogram: the sum's lower 16 bits; harmonics: b, the sine term in mA; tables: entry index + 2 (0 past the end)

Kind	Index	Bytes 2-7
0 capture	the sample, oldest first, 0 to parameter 32 − 1	channels A, B and C, each in its signal's unit (the parameter table lists the signals)
1 histogram	the bin, 0-255 over one electrical turn or one motor turn	the count (unsigned 16-bit), then the sum (signed 32-bit); the bin's mean is sum ÷ count
2 anti-cogging harmonics	the slot, 0-7	the harmonic of the electrical angle (1-32, 0 = unused), a and b in mA: the slot adds a·cos(h·θ) + b·sin(h·θ)
3 anti-cogging table	the first of three values, 0-255	three values in mA, 64 angle counts apart (256 per electrical turn)
4 encoder table	the first of three values, 0-127	three corrections in 1/16 encoder count, 128 per motor turn

Send_Buffer_Write

ID 43 | host → driver | 8 bytes | Python: `Send_Cogging_Harmonic(), Send_Cogging_Table()`

Writes the anti-cogging set: kind 2 one harmonic slot per frame, kind 3 three table values per frame, in the Respond_Buffer_Data layout. Other kinds are ignored (the encoder table cannot be written), and so is a harmonic above 32. The set lives in RAM until Send_Save_config; switch it on with parameter 37. The driver does not answer.

8 bytes

Byte	Field	Type	Meaning
0-1	Header	unsigned 16-bit	kind × 8192 + index; kind 2 or 3
2-3	Word A	signed 16-bit	kind 2: the harmonic (1-32, 0 = unused); kind 3: value at index
4-5	Word B	signed 16-bit	kind 2: a in mA; kind 3: value at index + 1
6-7	Word C	signed 16-bit	kind 2: b in mA; kind 3: value at index + 2

The driver does not answer this command.

Every parameter

The number and the name are read from the firmware; the rest is checked against its source. *Access*: a read-only parameter ignores a write; a run parameter starts or stops a run when written and gives its state when read.

Watchdog and tool

Parameters 1 to 5 live in settings block 2 of the EEPROM, 6 and 7 in the first settings block beside the other settings; Send_Save_config keeps all seven. A value outside a range is clamped, replaced or ignored, as the Value column says.

No.	What it is	Value
1	What the drive is built into. For an MSG the tens digit is the rail (1 = 100, 2 = 150, 3 = 200 mm) and the units digit the stepper (1 = 21.5, 2 = 40, 3 = 60 mm). Also `#Tool`; reported in the identity frame. `PARAM_TOOL_ID` · Access: set, read · Kept by Save: yes, settings block 2	0 none, 1 SSG-48, 11-33 MSG, 200-254 your own tools; a value outside 0-255 is ignored. Default 0

No.	What it is	Value
2	How long watchdog hold or descend runs before the after-action takes over. `PARAM_WD_FALLBACK_TIME` · Access: set, read · Kept by Save: yes, settings block 2	ms, 0-60000 (clamped). Default 2000
3	The current watchdog descend applies. 0 lets gravity lower a joint; a gripper gets its grip current. `PARAM_WD_FALLBACK_CURRENT` · Access: set, read · Kept by Save: yes, settings block 2	mA, signed, -32768 to 32767 (outside: 0). Default 0
4	The speed watchdog descend never exceeds, in either direction. `PARAM_WD_FALLBACK_SPEED` · Access: set, read · Kept by Save: yes, settings block 2	ticks/s, 0-65535 (outside: 1000). Default 1000
5	What follows hold or descend when the fallback time has run out. `PARAM_WD_AFTER_ACTION` · Access: set, read · Kept by Save: yes, settings block 2	0 legacy, 2 brake, 3 coast (any other value: 0). Default 0
6	The watchdog time, the same value Send_Watchdog sets. `PARAM_WD_TIME` · Access: set, read · Kept by Save: yes, settings block 1	ms, 0 = off (a negative value: 0). Default 0
7	What a trip of the watchdog alone does, the same value as byte 4 of Send_Watchdog. `PARAM_WD_ACTION` · Access: set, read · Kept by Save: yes, settings block 1	0 legacy, 1 hold, 2 brake, 3 coast, 4 descend (outside 0-255: 0; 5-255 act as legacy). Default 0

Gains, limits and settings

The values in force, readable and writable by parameter (their own commands, given in each row, still work). A float travels as its IEEE-754 bit pattern, big-endian, the way the gain commands send it; over UART `#Par 9,0.02` sets Kpv. A write goes to RAM like the commands' own; Send_Save_config keeps it. Parameter 23, the product stamp, stays read only.

No.	What it is	Value
8	Kp of the position loop (Send_Position_Gains, `#Kpp`). `PARAM_KPP` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)

No.	What it is	Value
9	Kp of the velocity loop (Send_Velocity_Gains, `#Kpv`). `PARAM_KPV` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
10	Ki of the velocity loop (Send_Velocity_Gains, `#Kiv`). `PARAM_KIV` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
11	Kp of the current loops; writing it sets the Iq and the Id loop, as `#Kpiq` does (Send_Current_Gains sets the Iq loop only). A calibration and parameter 49 set it too. `PARAM_KPIQ` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
12	Ki of the current loops; writing it sets the Iq and the Id loop, as `#Kiiq` does (Send_Current_Gains sets the Iq loop only). A calibration and parameter 49 set it too. `PARAM_KIIQ` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
13	KP, the stiffness of the impedance (PD) loop (Send_PD_Gains). `PARAM_KP` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
14	KD, the damping of the impedance (PD) loop (Send_PD_Gains). `PARAM_KD` · Access: set, read · Kept by Save: yes	float; 0 or more, below 10^9 (negative or not a number: refused)
15	The velocity limit (Send_Limits, `#Vlimit`). `PARAM_VELOCITY_LIMIT` · Access: set, read · Kept by Save: yes	float, ticks/s; 0 or more, below 10^9 (otherwise refused)
16	The current limit (Send_Limits, `#Ilim`). `PARAM_CURRENT_LIMIT` · Access: set, read · Kept by Save: yes	float, mA; 0-2800 (negative or above 2800: refused)
17	The voltage limit of the current loops (Send_Voltage_Limit, `#Vlim`). `PARAM_VOLTAGE_LIMIT` · Access: set, read · Kept by Save: yes	mV, 0-60000 (outside: refused); 0 = the supply voltage

No.	What it is	Value
18	The idle behaviour (Send_Settings bit 0, `#Brake`). `PARAM_BRAKE_COAST` · Access: set, read · Kept by Save: yes, settings block 2, from firmware 110 (109 coasted at every power-up). Never saved, and after `#Default`: coast	1 brake, 0 coast (any value but 0: brake)
19	The heartbeat interval (Send_Heartbeat_Setup). `PARAM_HEARTBEAT_MS` · Access: set, read · Kept by Save: yes	ms, 0-60000 (outside: refused); 0 = off
20	The torque constant (Send_Kt, `#Kt`, calibration). `PARAM_KT` · Access: set, read · Kept by Save: yes	float, Nm/A; 0 or more, below 10^9 (otherwise refused)
23	The product lock: the product that calibrated this board. Stamped at the first start-up of a calibrated board and at every save. A firmware that finds the other product's stamp drives nothing (see Errors). `PARAM_BOARD_PRODUCT` · Access: read only · Kept by Save: yes, settings block 2 (stamped by the firmware)	2 Spectral Micro, 1 STEPFOC, 0 none
90	Whether the velocity and current integrators are reset on every new setpoint command (`#Rstint`). Added in firmware 110. `PARAM_RSTINT` · Access: set, read · Kept by Save: yes	0 or 1 (anything else: refused). Default 0
91	The thermistor measurement and the temperature error (`#Term`). Added in firmware 110. `PARAM_THERMISTOR` · Access: set, read · Kept by Save: yes	0 off, 1 on (anything else: refused). Default 0
92	Gripper mode (`#Gripper`), in force at once: the driver then reads its commands as a gripper, or as a motor. It is written only while the driver is idle, so a move is never cut over half-way. Added in firmware 110. `PARAM_GRIPPER_MODE` · Access: set, read · Kept by Save: yes	0 or 1 (anything else, or a write while the driver is not idle: refused). Default 0

The motor

No.	What it is	Value
76	The phase resistance, the same value as `#R` (in Ω there). Calibration measures it; set by hand, the current gains follow	mΩ, 1-100000 (outside: refused)

No.	What it is	Value
	when parameter 49 is written. `PARAM_R_MOHM` · Access: set, read · Kept by Save: yes, with the calibration	
77	The phase inductance, the same value as `#L` (in H there). Calibration measures it; set by hand, the current gains follow when parameter 49 is written. `PARAM_L_UH` · Access: set, read · Kept by Save: yes, with the calibration	µH, 1-1000000 (outside: refused)
78	The motor's pole pairs, the same value as `#PP`. Calibration finds them. `PARAM_POLE_PAIRS` · Access: set, read · Kept by Save: yes, with the calibration	1-49, only while the driver is idle (otherwise refused); 50 and more would read as a STEPFOC calibration

Current sensing and the current loop

The current-sense zero is the ADC reading with no current flowing, in counts from 2048 (1 count is about 1.6 mA). The driver measures it at every start-up with the bridge off; the trim is added to it.

No.	What it is	Value
21	The current-sense zero of channel 1 in use. Measured at start-up (the trim, 59, added); writable for tests, until the next start-up. `PARAM_SENSE1_OFFSET` · Access: set, read · Kept by Save: no: measured again at every start-up	counts from 2048, -400 to 400 (outside: refused)
22	The current-sense zero of channel 2 in use, as 21. `PARAM_SENSE2_OFFSET` · Access: set, read · Kept by Save: no: measured again at every start-up	counts from 2048, -400 to 400 (outside: refused)
40	The zero of channel 1 measured with the bridge off, the trim added. `PARAM_SENSE1_OFFSET_OFF` · Access: read only · Kept by Save: no: measured at every start-up	counts from 2048
41	The zero of channel 2 measured with the bridge off, the trim added. `PARAM_SENSE2_OFFSET_OFF` · Access: read only · Kept by Save: no: measured at every start-up	counts from 2048

No.	What it is	Value
42	The second zero measurement, made in idle after start-up with the bridge switching at 50 % duty. On the bench it made the current ripple worse, so it is measured and kept but not used unless 46 asks for it. `PARAM_SENSE_ZERO_SWITCHING` · Access: read only · Kept by Save: no	0 not run, 1 measuring, 2 in use, 3 rejected (implausible), 4 interrupted (a command or an error came first), 5 measured, not in use
46	Which zero is in use. 1 takes effect only while 42 reads 5 (measured, not in use). `PARAM_SENSE_ZERO_USE_SWITCHING` · Access: set, read · Kept by Save: no	1 the zero measured with the bridge switching, 0 the bridge-off zero. Default 0
47	The zero of channel 1 measured with the bridge switching. `PARAM_SENSE1_OFFSET_SW` · Access: read only · Kept by Save: no	counts from 2048
48	The zero of channel 2 measured with the bridge switching. `PARAM_SENSE2_OFFSET_SW` · Access: read only · Kept by Save: no	counts from 2048
49	Writing it sets Kp_iq and Ki_iq at once from R, L and this bandwidth, as `#Cbw` does. Reading gives the bandwidth the gains in force amount to with the R and L in force (0 without them). `PARAM_CURRENT_BANDWIDTH` · Access: set, read · Kept by Save: no; the gains it sets are kept by Save	Hz, 10-500 (outside, or with R or L not set: refused). 160 at every start-up
58	The d-axis current setpoint, as `#Id`. It makes no torque, only more current in the coils. `PARAM_ID_SETPOINT` · Access: set, read · Kept by Save: no	mA, -3000 to 3000 (outside: refused). Default 0
59	The trim added to the zero of channel 1 measured at every start-up; writing it moves the zero in use at once. The Spectral Micro firmware 109 skips the trim step of the anti-cogging identification, so only a write sets it. `PARAM_ZERO_TRIM1` · Access: set, read · Kept by Save: yes, encoder block	counts, -60 to 60 (outside: refused). Default 0
60	The trim of channel 2, as 59. `PARAM_ZERO_TRIM2` · Access: set, read · Kept by Save: yes, encoder block	counts, -60 to 60 (outside: refused). Default 0
61	The electrical offset between the encoder and the motor, found by calibration (`#Offset` in rad). `PARAM_THETA_OFFSET` · Access: read only · Kept by Save: yes, with the calibration	0.1° electrical

No.	What it is	Value
62	The 1st harmonic of the anti-cogging set found by an identification with the trim (43 = 4 or 5). `PARAM_TRIM_H1_BEFORE` · Access: read only · Kept by Save: no	0.1 mA
63	After a trim, the 1st harmonic again; the Spectral Micro skips the trim, so it equals 62. `PARAM_TRIM_H1_AFTER` · Access: read only · Kept by Save: no	0.1 mA
64	Commutation delay compensation: the output voltage is turned forward by the angle the rotor covers in this time. Applied in the closed-loop modes above 100 ticks/s. 234 µs is the Spectral Micro's own delay, measured on the bench. `PARAM_COMM_DELAY_US` · Access: set, read · Kept by Save: no: 234 at every start-up	µs, 0-500 (outside: refused), 0 = off. Default 234
65	Where the transforms of every control loop take sin and cos from. On the STEPFOC bench the table was not faster. `PARAM_FAST_TRIG` · Access: set, read · Kept by Save: no	1 from the integer sine table, 0 from the float routine. Default 0
88	An experimental diagnostic of when the current sensors are read, off by default. They are read by a software start, wherever the code happens to be in the 25 kHz centre-aligned PWM cycle, so the moment of a reading depends on the code before it. With a count set, the driver waits until the PWM counter is within 24 counts of it (at most about 60 µs), then reads. Leave it at -1 unless you are investigating the current sensing. `PARAM_SENSE_SAMPLE_AT` · Access: set, read · Kept by Save: no: -1 at every start-up	-1: read wherever the code is (the default); 0 to the PWM period (the timer's reload value): wait for that count first (outside: refused)
89	Where in the PWM cycle the last current reading happened; it shows where parameter 88 puts the readings. `PARAM_SENSE_SAMPLE_SEEN` · Access: read only · Kept by Save: —	the PWM counter at the last reading, +65536 while it counts down

Speed estimate and the velocity loop

No.	What it is	Value
50	The speed the loops, the watchdog descend, telemetry and the capture use. Calibration and the gripper's stop detection always use the moving average. `PARAM_SPEED_ESTIMATOR` · Access: set, read · Kept by Save: no: 2 at every start-up	0 moving average, 1 PLL, 2 direct PLL (outside: refused). Default 2

No.	What it is	Value
51	The natural frequency of both PLLs. `PARAM_SPEED_PLL_BANDWIDTH` · Access: set, read · Kept by Save: no: 100 at every start-up	Hz, 10-250 (outside: refused). Default 100
74	The position input filter, position and PD modes: a critically damped filter between the host's position setpoints and the loops, which also makes the speed from them. Only for hosts that send positions without the speed; about the host's setpoint rate ÷ 12. `PARAM_INPUT_FILTER_HZ` · Access: set, read · Kept by Save: no	Hz, 0-200 (outside: refused), 0 = off. Default 0
75	Position mode: the velocity integral sees the position error less this band, so a holding motor stops pushing against a small error. It trades holding current for accuracy. `PARAM_VEL_INT_DEADBAND` · Access: set, read · Kept by Save: no	ticks, 0-2000 (outside: refused), 0 = off. Default 0

Anti-cogging

No.	What it is	Value
37	Which form of the stored anti-cogging is added to the current setpoint in velocity and position modes. 1 or 2 is refused while the stored set holds nothing for that form: read it back. `PARAM_COGGING_MODE` · Access: set, read · Kept by Save: yes, anti-cogging block	0 off, 1 harmonics, 2 table. Default 0
38	How much of the stored anti-cogging is applied. `PARAM_COGGING_SCALE` · Access: set, read · Kept by Save: yes, anti-cogging block	percent, 0-200 (clamped). Default 100
43	The anti-cogging identification on the drive: one motor turn forward and one back in velocity mode at the speed of 44. The shaft must be free to turn. A command from a host or an error aborts it. On the Spectral Micro 4 and 5 identify like 1 and 2 (the trim step is skipped); not yet	Write 0 abort, 1 identify, 2 identify and save, 3 encoder table, then anti-cogging, then save all, 4 identify with the current-sense zero trim, 5 the same and save. Read 0 idle, 1 turning forward, 2 turning backward, 3 done (harmonics on), 4 aborted, 5 failed (an error, or not calibrated), 6 failed (the shaft did not turn), 7 a combined run is measuring the encoder table, 8 the combined run's encoder table failed (52

No.	What it is	Value
	tested on its bench. `PARAM_COGCAL` · Access: set starts or stops a run; read gives its state · Kept by Save: the result, when started with 2, 3 or 5	says why), 9 not used, 10 switched off (the encoder table was measured again on its own), 11 done without the trim
44	The speed the identification turns at. `PARAM_COGCAL_SPEED` · Access: set, read · Kept by Save: no	ticks/s, 100-20000 (clamped). Default 1500
45	The cogging current the last identification found, rms. `PARAM_COGCAL_RIPPLE` · Access: read only · Kept by Save: no	0.1 mA rms

Encoder table

No.	What it is	Value
52	Measures the encoder table: the field turns the rotor open loop one shaft turn forward and one back at 2048 ticks/s (about 20 s). The shaft must be free. `PARAM_ENCAL` · Access: set starts or stops a run; read gives its state · Kept by Save: the table, when started with 2 or 3	Write 0 abort, 1 measure and use, 2 measure, use and save, 3 then the anti-cogging and save all. Read 0 idle, 1 turning forward, 2 turning backward, 3 done (in use), 4 aborted, 5 failed (an error, or not calibrated), 6 failed (the rotor did not follow the field), 7 rejected (over 4° peak to peak, or an offset move over 60° electrical)
53	Corrects the encoder reading with the table. `PARAM_ENCAL_USE` · Access: set, read · Kept by Save: yes, encoder block	1 on, 0 off; 1 only when a table was measured; refused while one is being measured
54	The table's error, peak to peak (0 without a table). `PARAM_ENCAL_PP` · Access: read only · Kept by Save: with the table	0.01° of the shaft

No.	What it is	Value
55	The once-per-turn part of the error, its amplitude (mostly a magnet off the axis). `PARAM_ENCAL_ONCE` · Access: read only · Kept by Save: with the table	0.01° of the shaft
56	No effect on the Spectral Micro: `#Cal` always runs the encoder table and the anti-cogging, and `#Calonly` never does. It stays so that the parameter numbers match STEPFOC. `PARAM_AFTER_CAL` · Access: set, read · Kept by Save: yes, encoder block	0, 1 or 2 (outside: refused). Default 0
57	How far the last encoder-table run moved the electrical offset. `PARAM_ENCAL_OFFSET` · Access: read only · Kept by Save: no	0.1° electrical

Capture and histogram

The signal numbers: 0 none, 1 measured Iq (mA), 2 measured Id (mA), 3 Iq setpoint with the anti-cogging (mA), 4 speed (ticks/s, saturates at ±32767), 5 speed ÷ 16, 6 position (ticks, the low 16 bits), 7 position error (ticks), 8 electrical angle (0-16383 per electrical turn, from the encoder alone), 9 encoder (0-16383 per motor turn), 10 Vbus (10 mV), 11 and 12 the two current-sense readings (counts after the zero), 13 the control interrupt's time (µs), 14 the anti-cogging feedforward (mA), 15 the Iq setpoint without the anti-cogging (mA), 16 the Uq setpoint (mV), 17 and 18 the d and q voltage the output transform received (mV). None of these settings is saved.

No.	What it is	Value
24	The signal of channel A, which is also the trigger channel. `PARAM_CAPTURE_SOURCE_A` · Access: set, read · Kept by Save: no	signal 0-18 (outside: 0). Default 4, speed
25	The signal of channel B. `PARAM_CAPTURE_SOURCE_B` · Access: set, read · Kept by Save: no	signal 0-18 (outside: 0). Default 1, Iq
26	The signal of channel C. `PARAM_CAPTURE_SOURCE_C` · Access: set, read · Kept by Save: no	signal 0-18 (outside: 0). Default 8, electrical angle

No.	What it is	Value
27	One sample every N control loops of 200 µs. `PARAM_CAPTURE_DIVIDER` · Access: set, read · Kept by Save: no	control loops per sample, 1-65535 (clamped). Default 1
28	1 to 4 clear the buffer and arm a capture; 0 stops one and keeps what was recorded. The error trigger fires when the error flag rises, a watchdog trip included; an error already present does not fire it. `PARAM_CAPTURE_TRIGGER` · Access: set, read · Kept by Save: no	0 stop, 1 now, 2 on an error, 3 channel A rising through the level, 4 falling through it (outside: 0)
29	The level of triggers 3 and 4. `PARAM_CAPTURE_LEVEL` · Access: set, read · Kept by Save: no	channel A's unit, -32768 to 32767 (saturated). Default 0
30	How many samples from before the trigger the capture keeps. `PARAM_CAPTURE_PRETRIGGER` · Access: set, read · Kept by Save: no	samples, 0-1023 (clamped). Default 0
31	Where the capture stands. `PARAM_CAPTURE_STATE` · Access: read only · Kept by Save: no	0 idle, 1 armed, 2 recording, 3 done
32	The samples readable with buffer kind 0. `PARAM_CAPTURE_COUNT` · Access: read only · Kept by Save: no	samples, 0-1024
33	Which sample of the read-out is the trigger sample. `PARAM_CAPTURE_TRIGGER_INDEX` · Access: read only · Kept by Save: no	sample index; -1 before a trigger
34	1 or 2 clear the histogram and start it; 0 stops it and keeps the data readable. `PARAM_HIST_X` · Access: set, read · Kept by Save: no	0 off, 1 electrical angle, 2 encoder (256 bins each; outside: 0)
35	The signal summed per bin. Set it before 34. `PARAM_HIST_Y` · Access: set, read · Kept by Save: no	signal 0-18 (outside: 0). Default 3, Iq setpoint
36	Samples summed since the start. The histogram stops by itself when one bin holds 65535. `PARAM_HIST_SAMPLES` · Access: read only · Kept by Save: no	samples
39	The samples per channel a capture holds. `PARAM_CAPTURE_LENGTH` · Access: read only · Kept by Save: —	1024

Sound

None of the sound settings is saved; see Sound.

No.	What it is	Value
80	What the sound is doing; a host that refills the queue or the sample buffer reads it. `PARAM_SOUND_STATE` · Access: read only · Kept by Save: —	bits 0-7 events queued, 8-18 samples buffered, 24 a tone sounds, 25 the queue plays, 26 samples play, 27 the samples ran dry
81	The sound's voltage at most, all tones and samples together. `PARAM_SOUND_LIMIT_MV` · Access: set, read · Kept by Save: no: 2000 at every start-up	mV, 0-6000 (outside: refused). Default 2000; 0 refuses sound and stops what plays
82	Which axis carries the sound. The q axis makes torque, a small ripple at the tone's pitch, and was much louder on the bench; the d axis makes no torque and is quieter. `PARAM_SOUND_AXIS` · Access: set, read · Kept by Save: no: 1 at every start-up	1 the q axis (default), 0 the d axis (outside: refused)

Status

No.	What it is	Value
79	What is in force and what is running, one bit each. `PARAM_FEATURES` · Access: read only · Kept by Save: —	bits, see What is in force
83	The stack margin: RAM between the heap and the stack that the stack has not used since start-up (setup() fills it with a pattern at its end, and this counts the bytes the pattern is still in). The stack needs about 1.5 to 2 KB of the about 7 KB there are; a value far below that means the drive is short of RAM. `PARAM_STACK_UNUSED` · Access: read only · Kept by Save: —	bytes; -1 if it was not measured
84	The git commit the firmware was built from. `#Info` prints it as `Build:`, with a `+` for bit 28. `PARAM_BUILD_COMMIT` · Access: read only · Kept by Save: —	the commit's first 7 hex digits as a number (bits 0-27); bit 28 set: built with uncommitted changes; 0 when git was not available at the build

No.	What it is	Value
85	When the firmware was built. `PARAM_BUILD_TIME` · Access: read only · Kept by Save: —	Unix seconds, UTC; 0 when the build did not set it
86	This board: the chip's factory-unique 96-bit number folded into 32 bits. It never changes, also not with a new firmware or `#Default`. `#Info` prints it as `Board ID:`, 8 hex digits. It is not the bootloader's board ID (0 to 13). `PARAM_BOARD_ID` · Access: read only · Kept by Save: —	32 bits, read as unsigned
87	A new number at every successful calibration, so a host can tell one calibration from the next. Send_Save_config keeps it, and `#Default` sets it back to 0. `#Info` prints it as `Calibration ID:`, 8 hex digits. `PARAM_CALIBRATION_ID` · Access: read only · Kept by Save: yes, settings block 2	32 bits, read as unsigned; 0 before the first calibration
93	The gripper's stroke between positions 0 and 255, from its last calibration (Gripper mode). Added in firmware 110. `PARAM_GRIPPER_STROKE` · Access: read only · Kept by Save: no: the gripper calibrates after every power-up	ticks; 0 while the gripper is not calibrated since power-up

What is in force: parameter 79

Parameter 79 says in one read what the drive runs with. `#Info` prints the same at its end, under `In force:`. Bit 0 is the least significant.

Bit	Set when
0	the motor is calibrated
1	an encoder table is measured (in RAM)
2	the encoder table is in use (parameter 53)
3	an anti-cogging set is in RAM (harmonics or table)
4	anti-cogging is on (parameter 37 not 0)
5	anti-cogging is on as the table (otherwise as the harmonics)

Bit	Set when
6	the anti-cogging set in RAM is the one saved in the EEPROM
7	a current-sense zero trim is applied (parameter 59 or 60 not 0)
8	the current-sense zero measured with the bridge switching is in use (parameter 46)
9	commutation delay compensation is on (parameter 64 not 0)
10	the speed estimate is a PLL (parameter 50 is 1 or 2)
11, 12	0 on the Spectral Micro (the STEPFOC's fast reading and fast PWM)
13	sin and cos come from the integer table (parameter 65)
14	the position input filter is on (parameter 74 not 0)
15	the velocity-integral deadband is on (parameter 75 not 0)
16-17	parameter 56 (no effect on the Spectral Micro)
20	a calibration is running
21	the encoder table is being measured
22	the anti-cogging is being identified
23	the runs after a calibration are about to start
24	sound is playing (Sound)

Bits 18, 19 and 25 to 31 are 0.

Reading the capture

A capture records three signals inside the control loop, one sample every *divider* loops, 1024 samples per channel. It starts when parameter 28 is written: at once (1), when the error flag rises (2, a watchdog trip included), or when channel A rises (3) or falls (4) through the level of parameter 29. Parameter 30 keeps that many samples from before the trigger. Status and diagnostics shows a whole capture.

1. Set the signals (24, 25, 26), the divider (27), the pre-trigger (30) and, for 3 and 4, the level (29).
2. Write the trigger (28). Parameter 31 reads 1 (armed), 2 (recording), then 3 (done).
3. Read parameter 32 (samples readable) and 33 (the trigger sample's index, −1 if none).
4. Send_Buffer_Read with kind 0, first 0, count = parameter 32: one Respond_Buffer_Data frame per sample, oldest first.

The histogram works the same way with parameters 34 to 36 and kind 1: it adds a signal into 256 bins by the electrical angle or by the encoder, every control loop, until it is stopped or one bin holds 65535 samples.

Sound

Sound through the motor: tones, a queue of timed events and samples, added to the output voltage while the driver holds or moves. Sound explains how to use them; parameters 80 to 82 are in the parameter table.

Send_Sound

ID 44 | host → driver | 8 bytes | Python: `Send_Sound()`

Plays sound through the motor: a tone now, a queue of timed events, or samples. The sound is added to the output voltage, on the q axis by default (parameter 82), and only while the driver holds or moves (position, velocity, current or PD mode); idle or an error stops all of it. Up to 4 tones sound at once (channels 0 to 3), from 20 Hz to below 2500 Hz, half the control-loop rate. Loudness is given as a current: the driver turns it into a voltage with the coil's impedance at that pitch, so low and high notes sound alike. The total voltage is limited to parameter 81 (2000 mV by default; 0 refuses sound). Added in firmware 109. The driver does not answer; parameter 80 says what plays. The Python library wraps the operations as Send_Tone, Send_Sound_Stop, Send_Sound_Event, Send_Sound_Play, Send_Sound_Clear, Send_Samples_Play and Send_Samples_Stop.

8 bytes

Byte	Field	Type	Meaning
0	Operation	unsigned 8-bit	0 stop everything (tones, queue, samples); 1 a tone now; 2 add an event to the queue; 3 play the queue; 4 play the samples; 5 samples off, buffer cleared; 6 clear the queue (tones that sound keep sounding)
1	Channel	unsigned 8-bit	the tone channel, 0-3 (operations 1 and 2)
2-3	A	unsigned 16-bit	1, 2: the pitch in Hz (2: 0 silences the channel); 4: the sample rate in Hz, at most the loop rate
4-5	B	unsigned 16-bit	1, 2: the loudness in mA; 4: the voltage of a full-scale sample in mV
6-7	C	unsigned 16-bit	1: the duration in ms, 0 = until stopped; 2: the wait in ms before the next event, 0 = the next one starts with it

The driver does not answer this command.

The queue holds 31 events and is played by the control loop with loop-exact timing: an event sets one channel's pitch and loudness, then waits. A chord is three events with wait 0 and a fourth with the chord's length. When the queue runs out, every channel falls silent. An event sent to a full queue is dropped. Samples: send them first with Send_Sound_Samples, then operation 4; the driver plays them at the given rate and stops when the buffer runs dry. A frame of another length, or a remote frame, is ignored. It does not reset the watchdog.

Send_Sound_Samples

ID 45 | host → driver | 1 to 8 bytes | Python: `Send_Samples()`

Appends 1 to 8 samples (signed 8-bit, one per byte, −127 to 127 = full scale) to the driver's 1024-place sample buffer, which holds 1023 samples. Samples that do not fit are dropped; parameter 80 says how many are buffered. Send_Sound with operation 4 plays them. Added in firmware 109. The driver does not answer.

8 bytes — eight samples; a shorter frame carries fewer

Byte	Field	Type	Meaning
0	Sample 1	signed 8-bit	the next sample
1	Sample 2	signed 8-bit	
2	Sample 3	signed 8-bit	
3	Sample 4	signed 8-bit	
4	Sample 5	signed 8-bit	
5	Sample 6	signed 8-bit	
6	Sample 7	signed 8-bit	
7	Sample 8	signed 8-bit	

The driver does not answer this command.

Gripper

The commands of gripper mode (`#Gripper 1` , saved).

Send_gripper_data_pack

ID 61 | host → driver | 0 or 5 bytes | Python: `Send_gripper_data_pack()`

The gripper command, for a driver set up as a gripper (`#Gripper 1` , saved). Position, speed and the current limit of the move, plus the activate, action and E-stop bits. A move is carried out only while the gripper is calibrated, activated and the action bit is 1. An empty frame (0 bytes) only asks for the state.

5 bytes

Byte	Bit	Field	Type	Meaning
0		Position	unsigned 8-bit	0 fully open, 255 fully closed
1		Speed	unsigned 8-bit	0 slowest, 255 fastest
2-3		Current	signed 16-bit	current limit of the move in mA; from firmware 109 its magnitude is used and it stops at 2800 mA; see the gripper manual for the safe range; from firmware 110 the gripper's own limit, the driver's current limit (parameter 16) stays as set
4	7	Activate	bit	1 activate the gripper, 0 deactivate
4	6	Action	bit	1 go to the position, 0 idle
4	5	E-stop	bit	1 raises the gripper E-stop error
4	4	Release direction	bit	read by the firmware but not used

The driver answers with **Respond_Gripper_data_pack**.

Send_gripper_calibrate

ID 62 | host → driver | 0 bytes | Python: `Send_gripper_calibrate()`

Starts the gripper calibration (the jaws find their end stops). Needed after every power-up. From firmware 110 each end is read under a light push and positions 0 and 255 lie a few ticks inside the two ends; this frame sent during a calibration starts it again from its first step. Parameter 93 gives the stroke found.

The driver does not answer this command.

Respond_Gripper_data_pack

ID 60 | driver → host | 4 bytes

The driver's answer to Send_gripper_data_pack: position, measured current, and the state bits. Python attributes `gripper_position`, `gripper_current`, `gripper_activated`, `gripper_action_status`, `gripper_object_detection`, `gripper_temperature_error`, `gripper_timeout_error`, `gripper_estop_error`, `gripper_calibrated`. The Python library 1.20 and older combine the two object-detection bits the wrong way round, so their `gripper_object_detection` reads 2 for an object found while closing and 1 for one found while opening; 1.21 and newer read them as the table says.

The frame (4 bytes)

Byte	Bit	Field	Type	Meaning
0		gripper_position	unsigned 8-bit	0 fully open, 255 fully closed
1-2		gripper_current	signed 16-bit	measured current in mA
3	7	gripper_activated	bit	1 activated
3	6	gripper_action_status	bit	the action bit of the last Send_gripper_data_pack (1 after #Grippos); the driver never clears it, so it stays 1 after the move ends: the object detection tells when the move has ended
3	5-4	gripper_object_detection	2 bits	bit 5 is the low bit, bit 4 the high bit: 0 moving, 1 object detected while closing, 2 object detected while opening, 3 at the position
3	3	gripper_temperature_error	bit	the driver's temperature error
3	2	gripper_timeout_error	bit	the driver's watchdog error: the CAN watchdog ran out
3	1	gripper_estop_error	bit	

Byte	Bit	Field	Type	Meaning
3	0	gripper_calibrated	bit	1 calibrated

UART commands

UART commands

Every command the firmware release this site describes understands over the serial port. The kind, the argument type, the mode a command enters and what it prints are read from the firmware; the descriptions are written by hand. Over UART explains the protocol.

Syntax

```
#<Name> [number]
```

- A **Set/Get** command sets the value when a number follows and prints the current value either way.
- A **Get** command only prints.
- An **Action** command does something; a number after it is ignored.
- Names are case-sensitive, at most 19 characters; the argument is one word of at most 19 characters, without spaces, so a list is written with commas (`#WdFb 2000,0,1000,2` , `#Par 1,200`); firmware 107 and older allowed 20 characters for name and number together. The line ends with LF (CR before it is fine). 256000 baud, 8N1, 3.3 V.
- A name the firmware does not know is answered `Unknown command` .

Every command

Command	Kind	Argument	Group
`#Activate`	Action		Status and utility
`#Brake`	Set/Get	int	Motion
`#Cal`	Action		Calibration and motor parameters
`#Calibrated`	Set/Get	int	Calibration and motor parameters
`#Calonly`	Action		Calibration and motor parameters
`#Calpwr`	Set/Get	float	Calibration and motor parameters
`#CANID`	Set/Get	int	Status and utility
`#Cbw`	Set/Get	float	Gains and limits

Command	Kind	Argument	Group
`#Clear`	Action		Status and utility
`#Close`	Action		Motion
`#Cogcal`	Set/Get		Calibration and motor parameters
`#Cyc`	Set/Get	int	Status and utility
`#Cyca`	Set/Get	int	Status and utility
`#Default`	Action		Status and utility
`#Dir`	Set/Get	int	Calibration and motor parameters
`#Encal`	Set/Get		Calibration and motor parameters
`#Error`	Get		Status and utility
`#Estop`	Action		Motion
`#Flux`	Set/Get	float	Calibration and motor parameters
`#Gripact`	Set/Get	int	Gripper
`#Gripcal`	Action		Gripper
`#Gripcur`	Set/Get	int	Gripper
`#Gripinfo`	Get		Gripper
`#Gripper`	Set/Get	int	Gripper
`#Grippos`	Set/Get	int	Gripper
`#Gripstop`	Set/Get	int	Gripper
`#Gripvel`	Set/Get	int	Gripper
`#Help`	Action		Status and utility

Command	Kind	Argument	Group
`#Iabc`	Get		Status and utility
`#Id`	Set/Get	float	Motion
`#Idle`	Action		Motion
`#Ilim`	Set/Get	int	Gains and limits
`#Info`	Action		Status and utility
`#Iq`	Set/Get	float	Motion
`#Iqoffset`	Set/Get	int	Calibration and motor parameters
`#KD`	Set/Get	float	Gains and limits
`#Kiid`	Set/Get	float	Gains and limits
`#Kiiq`	Set/Get	float	Gains and limits
`#Kiv`	Set/Get	float	Gains and limits
`#KP`	Set/Get	float	Gains and limits
`#Kpid`	Set/Get	float	Gains and limits
`#Kpiq`	Set/Get	float	Gains and limits
`#Kpp`	Set/Get	float	Gains and limits
`#Kpv`	Set/Get	float	Gains and limits
`#Kt`	Set/Get	float	Calibration and motor parameters
`#KV`	Set/Get	float	Calibration and motor parameters
`#L`	Set/Get	float	Calibration and motor parameters
`#LED`	Set/Get	int	Status and utility

Command	Kind	Argument	Group
`#Magnet`	Get		Calibration and motor parameters
`#Mode`	Get		Status and utility
`#Offset`	Set/Get	float	Calibration and motor parameters
`#Openloop`	Set/Get	float	Motion
`#Opens`	Set/Get	float	Calibration and motor parameters
`#Openv`	Set/Get	int	Calibration and motor parameters
`#P`	Set/Get	int	Motion
`#Par`	Set/Get	int	Status and utility
`#Param`	Get		Status and utility
`#PD`	Set/Get	int	Motion
`#PDI`	Set/Get	int	Motion
`#PDV`	Set/Get	int	Motion
`#Phase`	Get		Calibration and motor parameters
`#PP`	Set/Get	int	Calibration and motor parameters
`#Product`	Action		Status and utility
`#Pullconfig`	Get		Status and utility
`#PWM`	Get		Status and utility
`#R`	Set/Get	float	Calibration and motor parameters
`#Reset`	Action		Status and utility
`#Resv`	Set/Get	int	Calibration and motor parameters

Command	Kind	Argument	Group
`#Rstint`	Set/Get	int	Gains and limits
`#Save`	Action		Status and utility
`#SERNUM`	Set/Get	int	Status and utility
`#Temp`	Get		Status and utility
`#Term`	Set/Get	int	Status and utility
`#Tlim`	Set/Get	int	Status and utility
`#Tone`	Set/Get		Status and utility
`#Tool`	Set/Get	int	Status and utility
`#Ud`	Set/Get	int	Motion
`#Uq`	Set/Get	int	Motion
`#V`	Set/Get	int	Motion
`#Vbus`	Get		Status and utility
`#Vlim`	Set/Get	int	Gains and limits
`#Vlimit`	Set/Get	int	Gains and limits
`#Vmax`	Set/Get	int	Status and utility
`#Vp`	Set/Get	int	Calibration and motor parameters
`#WdFb`	Set/Get	int list	Status and utility

By group

- Motion: setpoints and modes, idle and the E-stop (`Estop`, from firmware 110)
- Gains and limits: the loop gains, the limits, the integrator reset, the current loop bandwidth

- Calibration and motor parameters: `Cal` and `Calonly`, the motor's R, L and pole pairs, the constants calibration finds, the calibration settings, the encoder table (`Encal`) and the anti-cogging (`Cogcal`)
- Status and utility: save, reset, defaults, identity, errors, temperature, the cyclic output, any parameter (`Par`), the tool id, the watchdog fallback, the product lock
- Gripper: gripper mode

Motion

`#Iq` `#Id` `#V` `#P` `#PD` `#PDV` `#PDI` `#Uq` `#Ud` `#Openloop` `#Idle` `#Estop` `#Brake` `#Close`

`#Iq`

Set/Get | Argument float: current setpoint in mA | Enters **current** mode

Current (torque) control with the given Iq setpoint.

Response: the measured Iq in mA

`#Id`

Set/Get | Argument float: current setpoint in mA

Sets the Id setpoint; normally left at 0. Also parameter 58.

Response: the measured Id in mA

`#V`

Set/Get | Argument int: speed setpoint in encoder ticks/s | Enters **velocity** mode

Velocity control with the given speed.

Response: the measured speed in ticks/s

`#P`

Set/Get | Argument int: position setpoint in encoder ticks | Enters **position** mode

Position control to the given position.

Response: the measured position in ticks

`#PD`

Set/Get | Argument int: position setpoint in encoder ticks | Enters **PD** mode

Impedance (PD) control around the given position with the gains KP and KD.

Response: the measured position in ticks

`#PDV`

Set/Get | Argument int: speed setpoint in encoder ticks/s

The speed setpoint of the PD loop.

Response: the PD speed setpoint

`#PDI`

Set/Get | Argument int: current feedforward in mA

The current feedforward of the PD loop.

Response: the PD current feedforward

`#Uq`

Set/Get | Argument int: voltage in mV | Enters **voltage torque** mode

Voltage-torque mode with the given q-axis voltage (no current loop).

Response: the Uq setpoint in mV

`#Ud`

Set/Get | Argument int: voltage in mV

The d-axis voltage used in voltage-torque mode; does not change the mode.

Response: the Ud setpoint in mV

`#Openloop`

Set/Get | Argument float: electrical speed in rad/s | Enters **open loop** mode

Turns the stator field at the given electrical speed without the encoder in the loop, with the voltage set by Openv. Without a number it still starts the motor, at the stored speed: `#Openloop` alone is not a query. The speed is the same setting as `#Opens`.

Response: the open-loop speed

`#Idle`

Action | Enters **idle** mode

Goes to idle; the motor coasts or brakes as set by Brake.

Response: `Idle`

`#Estop`

Action

The E-stop, as Send_ESTOP over CAN: it raises the E-stop error, the driver disables the motor, which coasts, and a calibration, gripper calibration or encoder-table or anti-cogging run that is running ends. The error stays until Clear. Added in firmware 110.

Response: `E-stop`

`#Brake`

Set/Get | Argument int: 0 or 1

What the motor does in idle: 1 brakes (the phases are shorted through the driver), 0 coasts (the phases are open, the default). Takes effect once the driver is idle, so send Idle first. From firmware 110 Save keeps it; on 109 the driver coasted again after a power cycle. Also parameter 18. An error always disables the motor, which then coasts. The braking current is not limited, see Control modes. Added in firmware 107.

Response: the setting

`#Close`

Action | Enters **idle** mode

Leaves open loop and goes back to closed-loop idle. From firmware 110, while a calibration (Cal, Calonly) runs, it stops the calibration instead: it ends as a failed one (`Calibration failed!`), with the motor released and the driver not calibrated until it calibrates again. Firmware 109 switched the mode under the running calibration.

Response: nothing; `Close: the calibration is stopped` during a calibration (firmware 110)

Gains and limits

`#Kpp` `#Kpv` `#Kiv` `#Kpiq` `#Kiiq` `#Kpid` `#Kiid` `#KP` `#KD` `#Vlimit` `#Ilim` `#Vlim` `#Rstint` `#Cbw`

`#Kpp`

Set/Get | Argument float: gain

Kp of the position loop.

Response: the current value

`#Kpv`

Set/Get | Argument float: gain

Kp of the velocity loop.

Response: the current value

`#Kiv`

Set/Get | Argument float: gain

Ki of the velocity loop.

Response: the current value

`#Kpiq`

Set/Get | Argument float: gain

Kp of the Iq current loop; also sets Kpid.

Response: the current value

`#Kiiq`

Set/Get | Argument float: gain

Ki of the Iq current loop; also sets Kiid.

Response: the current value

`#Kpid`

Set/Get | Argument float: gain

Kp of the Id current loop.

Response: the current value

`#Kiid`

Set/Get | Argument float: gain

Ki of the Id current loop.

Response: the current value

`#KP`

Set/Get | Argument float: gain

KP of the impedance (PD) loop.

Response: the current value

`#KD`

Set/Get | Argument float: gain

KD of the impedance (PD) loop.

Response: the current value

`#Vlimit`

Set/Get | Argument int: speed in encoder ticks/s

The speed limit: the speed the position loop may command, and the largest velocity setpoint. (The velocity integrator is clamped to the current limit.) The magnitude of a negative number is taken.

Response: the current limit

`#Ilim`

Set/Get | Argument int: current in mA

The Iq current limit. Set it to the rated phase current of your motor; the firmware clamps it to 2800 mA and takes the magnitude of a negative number.

Response: the current limit

`#Vlim`

Set/Get | Argument int: voltage in mV

The voltage limit of the current loops (Ud and Uq); 0 means use the supply voltage.

Response: the current limit in mV

`#Rstint`

Set/Get | Argument int: 0 or 1

Whether the integral accumulators are reset on every new setpoint. From firmware 110 also parameter 90.

Response: the setting

`#Cbw`

Set/Get | Argument float: bandwidth in Hz, 10 to 500

The current loop bandwidth. From firmware 109 setting it computes Kpiq and Kiiq at once from R and L, the way calibration does, and prints them (the same as parameter 49); a value outside 10-500, or with R or L at 0, changes nothing. Not saved: 160 Hz after a power cycle, but the gains it set are kept by Save.

Response: the bandwidth, then `Kp_iq = …, Ki_iq = …`

Calibration and motor parameters

`#Cal` `#Calonly` `#Encal` `#Cogcal` `#R` `#L` `#PP` `#Kt` `#KV` `#Flux` `#Offset` `#Iqoffset` `#Magnet` `#Dir` `#Phase` `#Resv` `#Calpwr` `#Openv` `#Opens` `#Vp` `#Calibrated`

`#Cal`

Action | Enters **idle** mode

Runs the calibration routine (see Calibration): it measures R and L, finds the pole pairs, the phase order and the angle offset, and measures Kt, KV and the flux linkage. It uses its own 1600 mA current limit for its steps; from firmware 109 it puts the limit that was in force back when it ends. From firmware 109, when it succeeds, the encoder table and then the anti-cogging follow by themselves, and everything is saved (about a minute; keep the shaft free).

Response: the calibration log

`#Calonly`

Action | Enters **idle** mode

The calibration alone: like Cal, but nothing follows it. Save afterwards with Save. Added in firmware 109.

Response: the calibration log

`#Encal`

Set/Get | Argument nothing, or save, all, stop, on, off, state

The encoder table (see Encoder table and anti-cogging). Alone it measures the table and uses it: the field turns the rotor open loop one shaft turn each way at 2048 ticks/s, about 20 s, with the shaft free. `save` also saves; `all` then identifies the anti-cogging and saves both; `stop` aborts; `on` and `off` switch the correction (on only with a measured table, and not while a run turns the motor); `state` or any other word only prints. The same as parameters 52 and 53. Added in firmware 109; not yet tested on a Spectral Micro.

Response: `Encoder table: <state>`, then the table's error in degrees

`#Cogcal`

Set/Get | Argument nothing, or save, trim, stop, state

The anti-cogging identification on the drive (see Encoder table and anti-cogging): one motor turn forward and one back in velocity mode at parameter 44's speed (1500 ticks/s, about 22 s), the shaft free to turn. `save` also saves; `stop` aborts, also the encoder part of a combined run; `state` or any other word only prints. `trim` identifies like the plain command on the Spectral Micro: its firmware skips the current-sense trim. Added in firmware 109; not yet tested on a Spectral Micro.

Response: `Anti-cogging identification: <state>`, the speed and the identified cogging in mA rms, and the harmonics when done

---

`#R`

Set/Get | Argument float: resistance in Ω

The phase resistance of the motor, measured by Cal; it can also be set by hand. Also parameter 76, in mΩ.

Response: the current value

`#L`

Set/Get | Argument float: inductance in H

The phase inductance of the motor, measured by Cal; it can also be set by hand. Also parameter 77, in µH.

Response: the current value

`#PP`

Set/Get | Argument int: pole pairs

The motor's pole pairs, found by Cal; they can also be set by hand. Also parameter 78: 1-49, only while idle (50 and more would read as a STEPFOC calibration); from 109 `#PP` answers `PP: not taken (1-49, and only while idle)` otherwise.

Response: the current value

`#Kt`

Set/Get | Argument float: Nm/A

The torque constant, found by calibration.

Response: the current value

`#KV`

Set/Get | Argument float: rpm/V

The velocity constant, found by calibration.

Response: the current value

`#Flux`

Set/Get | Argument float: Wb

The flux linkage, found by calibration. Works over UART from firmware 109; the parser of 107 and older accepted only the lowercase `#flux`, which had no handler.

Response: the current value

`#Offset`

Set/Get | Argument float: angle in rad

The angle offset between encoder zero and electrical zero, found by calibration.

Response: the current value

`#Iqoffset`

Set/Get | Argument int: current in mA

The current used by the angle-offset search and the commutation check of Cal, 300 mA by default. Not saved.

Response: the current value

`#Magnet`

Get

Reads the encoder's magnet status.

Response: `Magnet Good` or `Magnet Bad`

`#Dir`

Set/Get | Argument int: 0 or 1

The base direction of the encoder and phase mapping. Calibration always sets it to 1 (0 after `#Default`); the direction it finds is the commutation direction, which `#Info` prints.

Response: the current value

`#Phase`

Get

Prints the phase order found by calibration.

Response: `Phase order is: N`

`#Resv`

Set/Get | Argument int: voltage in mV

The voltage the resistance test of Cal starts at; it rises until the power reaches Calpwr. Not saved.

Response: the current value

`#Calpwr`

Set/Get | Argument float: power in W

The power the resistance and inductance tests of Cal may dissipate in the motor. Not saved.

Response: the current value

`#Openv`

Set/Get | Argument int: voltage in mV

The voltage of the open-loop spin, in calibration and with Openloop. Not saved.

Response: the current value

`#Opens`

Set/Get | Argument float: electrical speed in rad/s

The speed of the open-loop spin in calibration; the same setting as the speed of `#Openloop`.

Response: the current value

`#Vp`

Set/Get | Argument int: voltage in mV

The voltage of the phase-order search in calibration. Not saved.

Response: the current value

`#Calibrated`

Set/Get | Argument int: 0 or 1

The calibrated flag. `#Calibrated 1` clears a calibration error without calibrating; the motor may still be uncalibrated.

Response: the flag

Status and utility

`#Save` `#Reset` `#Default` `#Pullconfig` `#Info` `#Tone` `#Par` `#Tool` `#WdFb` `#Product` `#Param` `#Help` `#Error` `#Clear` `#Mode` `#Vbus` `#Vmax` `#Temp` `#Term` `#Tlim` `#Iabc` `#PWM` `#LED` `#CANID` `#SERNUM` `#Activate` `#Cyc` `#Cyca`

`#Save`

Action

Writes the configuration to the EEPROM. Do it after calibration and after tuning.

Response: `Save`

`#Reset`

Action

Resets the driver. From firmware 104 the reset lands in the bootloader, which waits 2 s for a flashing host and then starts the firmware again.

Response: nothing

`#Default`

Action

Writes the factory defaults to the EEPROM and loads them: the calibration is erased (Not calibrated), the gripper flag, heartbeat and watchdog are cleared, the hardware version and batch date are set to 1 and 24092024, and 1111 is stored as the serial number; from firmware 109 also the tool id (0), the watchdog fallback, and the anti-cogging set, the encoder table, its trims and parameter 56 (cleared). From firmware 109 the CAN node ID is kept (0 only when no valid one is stored). Note the values first.

Response: `Loaded default config!`

`#Pullconfig`

Get

Prints the whole configuration on one line, for Motor GUI.

Response: the configuration

`#Info`

Action

Prints the driver's identity and the motor: hardware version, batch date, firmware version, ISR time, CAN ID, calibrated or not, temperature, Vbus, Dir, phase order, commutation direction, the speeds of the calibration spin, pole pairs, R, L, Kt and KV. From firmware 109 also the build (`Build:`, the git commit, with `+` when built with uncommitted changes), the `Board ID` (this board, from the chip's unique number; not the bootloader's board ID), the `Calibration ID` (new at every successful calibration), the CAN receive overruns and dropped transmits, the product, the current-sense zeros, the tool id, the watchdog and its fallback, settings block 2, the anti-cogging, and at the end the `In force:` section (see Status and diagnostics). The serial number is `#SERNUM`, the gains `#Param`, the errors `#Error`.

Response: the report

`#Tone`

Set/Get | Argument f,mA[,ms], or off | Enters **position** mode

A tone through the motor (see Sound): `#Tone 440,250,1000` plays 440 Hz at a loudness of 250 mA for 1000 ms, on the axis parameter 82 picks. mA left out: 200; ms left out: 500; ms 0: until `#Tone off`. 20 Hz up to below 2500 Hz. It plays only while the driver holds or moves; from idle a calibrated driver first holds the position it is at (`#Idle` releases it). `#Tone` alone or `#Tone off` stops all sound. Added in firmware 109.

Response: `Tone 440 Hz, 250 mA, 1000 ms`; from idle first `Holding the position while it plays (#Idle releases it)`; `Tone: not played (…)` when it cannot; `Tone off`

`#Par`

Set/Get | Argument int: n, or n,value

Any parameter by its number (see Parameters and buffers): alone it lists every parameter, `#Par 1` reads one, `#Par 1,200` sets it (no spaces); a gain or limit (8-16, 20) takes a decimal, e.g. `#Par 9,0.02` sets Kpv. Added in firmware 109.

Response: `P<n> = <value>`; `refused: out of range, read only, not now, or unknown` when the driver did not take it; `P<n> unknown`

`#Tool`

Set/Get | Argument int: 0 to 255

The tool id: what the drive is built into, 0 none, 1 SSG-48, 11-33 MSG (tens = rail 1/2/3 = 100/150/200 mm, units = stepper 1/2/3 = 21.5/40/60 mm), 200-254 your own. The same as parameter 1. Kept by Save. Added in firmware 109.

Response: `Tool ID: N`, or `Invalid tool id (0-255)`

`#WdFb`

Set/Get | Argument int list: time_ms,current_mA,speed_ticks_s,after_action

The watchdog fallback, four numbers at once without spaces, e.g. `#WdFb 2000,0,1000,2`: how long hold or descend runs, descend's current and speed cap, and what follows (0 legacy, 2 brake, 3 coast). The same as parameters 2 to 5, clamped the same way; fewer than four numbers change nothing. Kept by Save. Added in firmware 109.

Response: `Watchdog fallback: … ms, … mA, … ticks/s, then …`

`#Product`

Action | Argument nothing, or claim, clear

The product lock: prints this firmware and the board's stamp. `claim` stamps the board as this product (a repaired board, a motor the stamp rule misjudges), `clear` removes the stamp; power-cycle after either. Over UART only. Added in firmware 109.

Response: `Firmware: Spectral Micro; board stamp: …`

`#Param`

Get

Prints every gain of every loop.

Response: the gains

`#Help`

Action

Prints a short command reference. Added in firmware 106.

Response: the reference

`#Error`

Get

Prints which errors are active.

Response: `No errors!` or `In error state!`, then one line per active error (see Errors and LED states), and `FAULT pin is LOW: the gate driver reports a fault` while the driver stage's fault pin is active

---

`#Clear`

Action | Enters **idle** mode

Clears the errors and goes to idle. An error whose cause is still present comes straight back; a calibration error is cleared only by calibrating (or `#Calibrated 1`).

Response: `Clear errors`

`#Mode`

Get

Prints the control mode. In voltage-torque mode (`#Uq`) it prints nothing.

Response: `Idle mode`, `Position control`, `Speed control`, `Current control`, `PD control`, `Open loop`, `Gripper mode`, `Gripper calib mode`

`#Vbus`

Get

Prints the supply voltage.

Response: mV

`#Vmax`

Set/Get | Argument int: voltage in mV

The supply voltage above which a Vbus error is raised: 29000 mV after `#Default`.

Response: `Vmax in mV: N`

`#Temp`

Get

Prints the thermistor temperature.

Response: °C, or `Temp Off` while the thermistor is disabled

`#Term`

Set/Get | Argument int: 0 or 1

Enables the thermistor measurement and the temperature error (off by default). From firmware 110 also parameter 91.

Response: the setting

`#Tlim`

Set/Get | Argument int: temperature in °C

The temperature at which the temperature error is raised; 75 by default.

Response: `Temp limit N`

`#Iabc`

Get

Prints the phase currents.

Response: `Ia`, `Ib`, `Ic` in mA

`#PWM`

Get

Prints the PWM duty values.

Response: the duties

`#LED`

Set/Get | Argument int: 0 or 1

Turns the status LED on or off.

Response: the LED state

`#CANID`

Set/Get | Argument int: 0 to 13

The CAN node ID. Use 0 to 13: the firmware accepts 14 and 15, but the bootloader reserves them and a driver on one of those cannot be updated over CAN. Kept over a power cycle after Save, which also copies it into the bootloader's board ID; from firmware 109 Default and a settings reset keep it too.

Response: the ID, or `Invalid CAN ID`

`#SERNUM`

Set/Get | Argument int: number

The serial number. Save writes it to the EEPROM, but the firmware does not read it back at power-up: after a power cycle the driver reports 1.

Response: `Serial number is: N`

`#Activate`

Action

Sets the controller's activated flag (bit 6 of byte 1 of Send_Respond_State_of_Errors). It is not the gripper's activate bit, which `#Gripact` and Send_gripper_data_pack set.

Response: `Controller activated`

`#Cyc`

Set/Get | Argument int: interval in ms (0 = off)

Sends position, speed and Iq on its own every interval: `$ <position> <speed> <Iq>` ; for BetterSerialPlotter.

Response: `Cyclic disabled` for `#Cyc` without a number; `#Cyc 0` stops the output without an answer

---

`#Cyca`

Set/Get | Argument int: interval in ms (0 = off)

Like Cyc, in the Arduino plotter format `Position:x,Speed:y,Iq:z`.

Response: `Cyclic disabled` for `#Cyca` without a number; `#Cyca 0` stops the output without an answer

---

Gripper

`#Gripper` `#Gripcal` `#Grippos` `#Gripvel` `#Gripcur` `#Gripact` `#Gripstop` `#Gripinfo`

`#Gripper`

Set/Get | Argument int: 0 or 1

Whether this driver is a gripper. Save it; `#Default` clears it. From firmware 110 also parameter 92, which takes a write only while the driver is idle.

Response: the flag

`#Gripcal`

Action | Enters **gripper calibration** mode

Starts the gripper calibration (the jaws find their end stops). From firmware 110 each end is read under a light 100 mA push after the 500 mA one, and 0 and 255 lie a few ticks inside the two ends; the gripper stays uncalibrated when the ends lie less than 1000 ticks apart, and Gripcal during a calibration starts it again from its first step. Parameter 93 then gives the stroke in ticks.

Response: `Gripper calibration!`

`#Grippos`

Set/Get | Argument int: 0 (open) to 255 (closed) | Enters **gripper** mode

The position setpoint; also activates the gripper, sets the action bit and goes to gripper mode.

Response: the current position

`#Gripvel`

Set/Get | Argument int: 0 to 255

The speed setpoint of gripper moves.

Response: the setpoint

`#Gripcur`

Set/Get Argument int: current in mA

The current limit of gripper moves. From firmware 109 a gripper move uses its magnitude and stops at 2800 mA; see the gripper manual for the safe range. From firmware 110 it is the gripper's own limit: the driver's current limit (Ilim) stays as set.

Response: the setpoint

`#Gripact`

Set/Get Argument int: 0 or 1

The activate bit.

Response: the bit

`#Gripstop`

Set/Get Argument int: 0 or 1

The gripper E-stop bit.

Response: the bit

`#Gripinfo`

Get

Prints the gripper state: whether the driver is a gripper, calibrated, activated, the position, speed and current setpoints, the E-stop, action and object-detection status, and the end stops.

Response: the report

Defaults

The settings of the firmware, read from its source at the release this site describes. The table leaves out the firmware's internal state (counters, flags, scratch variables).

- **Start value** is the value in the firmware's structures (`common.h`). At power-up the firmware loads the saved settings from the EEPROM over it, so for a setting that `#Save` keeps the start value only shows until the EEPROM is read.
- **After `#Default`** is what the driver runs with after `#Default`, which writes the factory values to the EEPROM and loads them. A fresh board, and a board whose saved configuration fails its check at power-up, gets the same values.
- **Kept by `#Save`** says whether the setting survives a power cycle. *No* means the driver starts with the start value again after every power-up.

`#Default` also erases the calibration; from firmware 109 it keeps a valid stored CAN node ID (`existing_id` in the table; 0 when none is stored); see Saving the configuration. A name after a value, such as `160 (CURRENT_BANDWIDTH_DEFAULT)`, is the constant in `constants.h` the value comes from. The settings of firmware 109 that are parameters, the anti-cogging and the encoder table among them, are in the parameter table with their defaults.

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`SERIAL_NUMBER`	1	1111 in the EEPROM, not read back	written, not read back
`HARDWARE_VERSION` Version of the Spectral BLDC driver PCB	1	1	yes
`BATCH_DATE` Date the batch was produced	1	24092024	yes
`SOFTWARE_VERSION` Software release	1	110	no: set by the firmware
`I_AM_GRIPPER` Is motor controller gripper (1) or not (0)	1	0	yes

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`LED_ON_OFF` Use LED for status indication	1	1	yes
`theta_offset` will be determined at calibration	0.0f	0	yes
`pole_pairs` Number of pole pairs of your BLDC motor	11	7	yes
`current_control_bandwidth` Hz: the current PI gains come from R, L	160 (CURRENT_BANDWIDTH_DEFAULT)	—	no
`DIR_` When looking at the magnetic encoder counter-clockwise is POSITIVE rotation.	1	0	yes
`Phase_order`	2	0	yes
`commutation_dir` +1 or -1, auto-detected at calibration so +Iq/+velocity/+rotation map to +encoder rotation	1	1	yes
`Thermistor_on_off` Are we taking termistor mesurements? 0 is off, 1 is on	0	0	yes
`brake_coast` Idle-mode (Controller_mode == 0) phase behavior: 0 = coast (driver disabled, Hi-Z), 1 = brake (driver stays enabled, zero-voltage PWM vector shorts the phases together)	0	0 (BRAKE_COAST_DEFAULT)	yes

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`Resistance` Resistance of SINGLE PHASE of your BLDC motor	0	0	yes
`Total_Resistance` = 2 * Resistance + 2 * R_sense + 2 *Rdson	0	0	yes
`Inductance` Inductance of SINGLE PHASE of your BLDC motor	0	0	yes
`Kt` Torque constant of your BLDC motor	0	0	yes
`flux_linkage` Flux linkage of your BLDC motor	0	0	yes
`KV`	0	0	yes
`Max_temperature` Max temperature in degrees	75	75	yes
`Min_temperature` Min temperature in degrees	-20	—	no
`Max_Vbus` Max vbus voltage in mV	29500	29000	yes
`Min_Vbus` Min vbus voltage in mV	9000	—	no
`watchdog_time_ms`	0	0	yes
`watchdog_action` 0 legacy, 1 hold, 2 brake, 3 coast, 4 descend (watchdog_action.h)	0	0	yes

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`wd_fallback_time_ms` how long hold/descend may run before the after-action	2000	2000	yes
`wd_fallback_current` mA, descend: the current applied (signed)	0	0	yes
`wd_fallback_speed` ticks/s, descend: the speed cap in either direction	1000	1000	yes
`wd_after_action` after hold/descend: 0 legacy, 2 brake, 3 coast	0	0 (WD_ACTION_LEGACY)	yes
`calibration_id` parameter 87: new at every successful calibration	0	0 in the EEPROM, not read back	written, not read back
`tool_id` what the drive is built into: 0 none, 1 SSG-48, 11-33 MSG, 200-254 custom	0	0	yes
`board_product` product lock stamp in block 2: 1 STEPFOC, 2 Spectral, 0 none	0	0	yes
`sense1_offset` current-sense zero of channel 1, ADC counts from 2048 (start-up)	0	—	no
`sense2_offset` current-sense zero of channel 2	0	—	no
`comm_delay_us` parameter 64: commutation delay compensation, us (0 = off)	234 (COMM_DELAY_DEFAULT_US)	—	no

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`input_filter_hz` parameter 74: the position input filter, Hz (0 = off)	0	—	no
`vel_int_deadband` parameter 75: the velocity integral's deadband, ticks (0 = off)	0	—	no
`sound_limit_mv` parameter 81: the sound's voltage at most (batch 22)	2000 (SOUND_LIMIT_DEFAULT_MV)	—	no
`sound_axis` parameter 82: 1 the q axis (default: much louder on the bench), 0 the d axis	1	—	no
`fast_trig` parameter 65: 1 the transforms' sin/cos from the integer table (it was	0	—	no
`speed_estimator` parameter 50	2 (SPEED_ESTIMATOR_DEFAULT)	—	no
`speed_pll_bandwidth` parameter 51, Hz	100 (SPEED_PLL_BANDWIDTH_DEFAULT)	—	no
`Heartbeat_rate_ms` Motor controller will send data every Heartbeat_rate_ms	0	0	yes
`Calibrated` 0 if motor is not calibrated, 1 if it is calibrated	0	0	yes
`Resistance_calc_voltage` Voltage used to calculate resistance //1000	1000	—	no

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`Calibration_power` Max power that can dissipate during inductance test.	12	—	no
`Open_loop_voltage` Voltage to use when spinning the motor open loop. In mV	5000	—	no
`Open_loop_speed` Speed to be used when spinning the motor open loop. In RAD/s	20.0	—	no
`Phase_voltage` Max voltage to be used during Kv calculation	5000	—	no
`cyclic_period` We send messages every cyclic_period	0	—	no
`CAN_ID` CAN id of THIS motor driver! No 2 motor drivers can share the same ID.	0	existing_id	yes
`calibration_offset_current`	300	—	no
`Voltage_limit` Voltage limit for Ud and Uq voltages. If 0 value of controller.VBUS_mV will be used	0	0	yes
`Reset_integral_accumulator`	0	0	yes
`Kp_p`	11	11	yes
`Position_setpoint`	0	—	no

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`Kp_v` 0.006 basic values that work for most motors but not optimal	0.03	0.03	yes
`Ki_v` 0.0001 basic values that work for most motors but not optimal	0.0003	0.0003	yes
`Velocity_setpoint` -1.5	0	—	no
`Velocity_limit` Clamp integrals to this [TICKS/s]	800000	800000	yes
`Velocity_limit_error` [TICKS/s] ; Velocity when we will report error	20000000	—	no
`Kp_id`	3	3	yes
`Ki_id`	1.5	1.5	yes
`Id_setpoint`	0	—	no
`Kp_iq`	3	3	yes
`Ki_iq`	1.5	1.5	yes
`Iq_current_limit`	1000	1000	yes
`Id_current_limit`	0	0	yes
`Feedforward_current`	0	—	no
`KD` [Nm*s/rad] KD 0.002800	0.002800	0.0028	yes

Variable and firmware comment	Start value	After `#Default`	Kept by `#Save`
`KP` [Nm/rad] KP 0.14000	0.14000	0.1400	yes
`Uq_setpoint`	0	—	no
`Ud_setpoint`	0	—	no
`position_setpoint`	0	—	no
`speed_setpoint`	20	—	no
`current_setpoint`	200	—	no
`action_status` -0 Stopped(or performing activation /automatic release).	0	—	no
`activated`	0	—	no
`release_direction`	0	—	no
`estop_status`	0	—	no

Errors and LED states

The LED

Pattern	State
three short flashes, a pause	normal operation: calibrated, no error
fast flashing, about 10 times a second	calibration running
blinking for about 2 s after power-up	the bootloader waits for a host before it starts the firmware
on, steady	error mode, or not calibrated
off	no power, or `#LED 0`

Normal operation	Calibration	Error mode
		

The error flags

The flags as the firmware keeps them, the line `#Error` prints for each, and where each sits in the two bytes of Send_Respond_State_of_Errors (bits numbered from the most significant).

Flag	`#Error` prints	Byte, bit	Raised when
`Error`	`In error state!`	0, 7	any of the flags below is set (also the error bit of every CAN identifier)

Flag	`#Error` prints	Byte, bit	Raised when
`temperature_error`	`Temp error`	0, 6	the thermistor reads above `#Tlim` or below −20 °C (with `#Term 1`)
`encoder_error`	`Encoder error`	0, 5	the magnet is not detected or the reading fails its check
`Vbus_error`	`Vbus error`	0, 4	the supply is below 9 V or above `#Vmax` (29 V after `#Default`); from firmware 109 a low supply is not checked in the first 500 ms after power-up
`Driver_error`	`Drv error`	0, 3	the MOSFET driver reported a fault on its fault pin: undervoltage, thermal shutdown or overcurrent. `#Error` also prints `FAULT pin is LOW: the gate driver reports a fault` while the fault output is still active
`Velocity_error`	`Velocity error`	0, 2	the measured speed exceeds 20,000,000 ticks/s (about 1220 revolutions per second), a fixed limit: a runaway or an encoder fault, not `#Vlimit`
`Current_error`	`Current error`	0, 1	never in this firmware; reserved
`ESTOP_error`	`Received ESTOP command`	0, 0	Send_ESTOP was received, or from firmware 110 `#Estop` over UART
`Calibrated` (0)	`Not calibrated`	1, 7 (1 = calibrated)	the motor has not been calibrated
`Activated`	—	1, 6	the activated flag, not an error
`Watchdog_error`	`Watchdog error`	1, 5	the watchdog time passed without a motion command or an answered request over CAN
`wrong_product`	`Wrong firmware for this board (product lock)`	1, 4	from firmware 109: the firmware found the other product's stamp, or a calibration its product cannot make, at start-up; it drives nothing

Gripper mode has its own flags in Respond_Gripper_data_pack: a temperature error (the driver's temperature error), a timeout error (the driver's watchdog error: the CAN watchdog ran out) and an E-stop error; see the Gripper reference.

Clearing

`#Clear` and Send_Clear_Error clear every flag except `Calibrated` and go to idle; a flag whose cause persists is set again at once. While any flag is set the driver disables the motor, which coasts whatever `#Brake` says. The watchdog, E-stop and velocity errors clear and stay clear; the Vbus, encoder, driver and temperature errors come back until the cause is gone. Calibration clears the calibration error. A wrong product stays until the right firmware is flashed. Clearing a watchdog error also ends a watchdog hold or descend.

Python library

The `Spectral-BLDC` package (Spectral-BLDC-Python, `pip install Spectral-BLDC`) has one method per CAN message in `Spectral_BLDC.SpectralCAN`, and `Spectral_BLDC.CanCommunication` for the bus. With Python shows how they are used.

Calls and messages

Library call	Sends	Answered with
`Send_data_pack_1()`	Send_data_pack_1 (ID 2) Motion	Respond_data_pack_1
`Send_data_pack_PD()`	Send_data_pack_PD (ID 4) Motion	Respond_data_pack_1
`Send_PD_Gains()`	Send_PD_Gains (ID 16) Gains and limits	—
`Send_Current_Gains()`	Send_Current_Gains (ID 17) Gains and limits	—
`Send_Velocity_Gains()`	Send_Velocity_Gains (ID 18) Gains and limits	—
`Send_Position_Gains()`	Send_Position_Gains (ID 19) Gains and limits	—
`Send_Limits()`	Send_Limits (ID 20) Gains and limits	—
`Send_Kt()`	Send_Kt (ID 22) Gains and limits	—
`Send_Voltage_Limit()`	Send_Voltage_Limit (ID 34) Gains and limits	—
`Send_Respond_Ping()`	Send_Respond_Ping (ID 10) Status requests	Send_Respond_Ping (data frame)

Library call	Sends	Answered with
`Send_Respond_Temperature()`	Send_Respond_Temperature (ID 23) Status requests	Send_Respond_Temperature (data frame)
`Send_Respond_Voltage()`	Send_Respond_Voltage (ID 24) Status requests	Send_Respond_Voltage (data frame)
`Send_Respond_Device_Info()`	Send_Respond_Device_Info (ID 25) Status requests	Send_Respond_Device_Info (data frame)
`Send_Respond_State_of_Errors()`	Send_Respond_State_of_Errors (ID 26) Status requests	Send_Respond_State_of_Errors (data frame)
`Send_Respond_Iq_data()`	Send_Respond_Iq_data (ID 27) Status requests	Send_Respond_Iq_data (data frame)
`Send_Respond_Encoder_data()`	Send_Respond_Encoder_data (ID 28) Status requests	Send_Respond_Encoder_data (data frame)
`Send_Respond_Kt()`	Send_Respond_Kt (ID 33) Status requests	Send_Respond_Kt (data frame)
`Send_Respond_Identity()`	Send_Respond_Identity (ID 39) Status requests	Send_Respond_Identity (data frame)
`Send_Respond_Telemetry()`	Send_Respond_Telemetry (ID 40) Status requests	Send_Respond_Telemetry (data frame)
`Send_CAN_ID()`	Send_CAN_ID (ID 11) System	—
`Send_ESTOP()`	Send_ESTOP (ID 0) System	—
`Send_Idle()`	Send_Idle (ID 12) System	—

Library call	Sends	Answered with
`Send_Save_config()`	Send_Save_config (ID 13) System	—
`Send_Reset()`	Send_Reset (ID 14) System	—
`Send_Clear_Error()`	Send_Clear_Error (ID 1) System	—
`Send_Watchdog()`	Send_Watchdog (ID 15) System	—
`Send_Heartbeat_Setup()`	Send_Heartbeat_Setup (ID 30) System	—
`Send_Settings()`	Send_Settings (ID 35) System	—
`Send_Parameter()`	Send_Parameter (ID 36) Parameters and buffers	—
`Send_Get_Parameter()`	Send_Get_Parameter (ID 37) Parameters and buffers	Respond_Parameter
`Send_Buffer_Read()`	Send_Buffer_Read (ID 41) Parameters and buffers	—
`Send_Cogging_Harmonic()`, `Send_Cogging_Table()`	Send_Buffer_Write (ID 43) Parameters and buffers	—
`Send_Sound()`	Send_Sound (ID 44) Sound	—
`Send_Samples()`	Send_Sound_Samples (ID 45) Sound	—
`Send_gripper_data_pack()`	Send_gripper_data_pack (ID 61) Gripper	Respond_Gripper_data_pack

Library call	Sends	Answered with
`Send_gripper_calibrate()`	Send_gripper_calibrate (ID 62) Gripper	—

The calls for the messages added in firmware 109 (Send_Parameter, Send_Get_Parameter, Send_Respond_Identity, Send_Respond_Telemetry, Send_Buffer_Read, Send_Cogging_Harmonic, Send_Cogging_Table, Send_Sound, Send_Samples), Send_Settings and Send_Brake_Coast for the idle behaviour, and the actions 'Coast' and 'Descend' of Send_Watchdog, come with library 1.21. It also adds calls built on the parameters, among them Send_Watchdog_Fallback, Send_Tool_ID, Send_Cogging_Mode, Send_Capture_Setup, Send_Capture_Trigger, Send_Histogram, Send_Cogging_Calibrate and Send_Encoder_Calibrate; the sound calls Send_Tone, Send_Sound_Stop, Send_Sound_Event, Send_Sound_Play, Send_Sound_Clear, Send_Samples_Play and Send_Samples_Stop on top of Send_Sound and Send_Samples; and the module `Spectral_BLDC.music` (note names, ringtone melodies, MIDI and WAV files as queue events and samples, see Sound).

The same release adds `Spectral_BLDC.helpers`, blocking functions that send a request and wait for the answer: `get_parameter` and `set_parameter` (which reads the value back), `read_identity`, `read_telemetry`, `features`, `read_buffer`, `capture`, `wait_for_state` (for the encoder table, the anti-cogging and other runs), and `play_events` and `stream_samples` for sound. They are for setting up and diagnostics, not for the inside of a fast control loop. The library's `examples` folder shows them at work: `parameters_and_status.py`, `capture.py`, `calibration_features.py`, `watchdog.py` and `sound.py`.

The same release is stricter about what it sends and reads:

- `Send_Parameter` takes the value itself. Parameters 8 to 16 and 20 are sent as floats (`Send_Parameter(16, 1500)` is 1500 mA); every other parameter takes a whole number, and anything else raises `ValueError`.
- A position or a speed that does not fit 24 bits, or a current that does not fit 16 bits, raises `ValueError`; 1.20 wrapped it round. `Send_Voltage_Limit` raises `ValueError` outside 0 to 60000 mV.
- A node ID must be 0 to 15.
- `UnpackData` ignores frames from other nodes, remote frames and extended frames.
- Parameters 79, 80 and 84 to 87 read as unsigned numbers.
- A send that fails returns `False` and is logged once.

Receiving

`CanCommunication.receive_can_messages(timeout)` returns `(message, ids)`: the raw `python-can` message and a tuple `ids` with `node_id`, `command_id` and `error_bit` from the identifier, or `(None, None)` after the timeout. `SpectralCAN.UnpackData(message, ids)` decodes the message into the object's attributes:

Message	Attributes
Respond_data_pack_1	`position`, `speed`, `current`
Send_Respond_Encoder_data	`position`, `speed`
Send_Respond_Iq_data	`current`
Send_Respond_Temperature	`temperature`
Send_Respond_Voltage	`voltage`
Send_Respond_Device_Info	`HARDWARE_VERSION`, `BATCH_DATE`, `SOFTWARE_VERSION`, `SERIAL_NUMBER`
Send_Respond_State_of_Errors	`error`, `temperature_error`, `encoder_error`, `vbus_error`, `driver_error`, `velocity_error`, `current_error`, `estop_error`, `calibrated`, `activated`, `watchdog_error`; from 1.21 also `wrong_product`
Send_Respond_Identity (1.21 and newer)	`product_id`, `product_name`, `tool_id`, `firmware_version`, `identity_hardware_version`, `pole_pairs`, `identity_calibrated`, `identity_flags`
Send_Respond_Telemetry (1.21 and newer)	`temperature`, `voltage`, the error flags as above, `current`
Respond_Parameter (1.21 and newer)	`parameters[number]` (floats for 8-16 and 20), `unknown_parameter`; `tool_id` and `watchdog_fallback` when those parameters come in
Respond_Buffer_Data (1.21 and newer)	`capture[index]`, `histogram[bin]`, `cogging_harmonics[slot]`, `cogging_table[index]`, `encoder_table[index]`
Send_Respond_Kt	`Kt`
Respond_Gripper_data_pack	`gripper_position`, `gripper_current`, `gripper_activated`, `gripper_action_status`, `gripper_object_detection`, `gripper_temperature_error`, `gripper_timeout_error`, `gripper_estop_error`, `gripper_calibrated`

Object detection in the library

Library 1.20 and older assemble `gripper_object_detection` with the two bits swapped relative to the firmware, so they report 1 where the gripper sends 2 (object detected while opening) and 2 where it sends 1 (object detected while closing); 0 and 3 are unaffected. 1.21 and newer read them right.

Guides

Guides

Worked examples with code, from a first Python program to a mobile robot, a leg, a gravity-compensated arm and two motors coupled over a network, and the SimpleFOC library as an alternative firmware. The CAN commands and the Python library are the same on STEPFOC, so the code runs there unchanged.

- Python: read a driver's position and speed over CAN
- Mobile robot: two wheels driven from a game controller
- Quadruped leg: the kinematics and control of a two-motor leg
- Gravity compensation: hold a link against gravity with a current feedforward
- Bilateral teleoperation: two motors that follow each other, with force feedback
- SimpleFOC: running the SimpleFOC library on the Spectral Micro instead of the Spectral firmware

More examples are in the Spectral-BLDC-Python examples.

Python guide

More examples are in the Spectral-BLDC-Python examples folder.

Examples

Get_encoder_data.py

```
import Spectral_BLDC as Spectral
import time

# Init comms channel
Communication1 = Spectral.CanCommunication(bustype='slcan', channel='COM41', bitrate=1000000)
# Init motor object
Motor1 = Spectral.SpectralCAN(node_id=0, communication=Communication1)

while True:

    # Send this msg to the motor controller, it will respond with position and speed data
    Motor1.Send_Respond_Encoder_data()

    # Get CAN msg, timeout tells us to continue if there is no msg after in this case 0.2 s
    message, UnpackedMessageID = Communication1.receive_can_messages(timeout=0.2)

    # If there was a msg unpack it
    if message is not None:
        print(f"Message is: {message}")
        print(f"Node ID is : {UnpackedMessageID.node_id}")
        print(f"Message ID is: {UnpackedMessageID.command_id}")
        print(f"Error bit is: {UnpackedMessageID.error_bit}")
        print(f"Message length is: {message.dlc}")
        print(f"Is is remote frame: {message.is_remote_frame}")
        print(f"Timestamp is: {message.timestamp}")

        # Here we unpack the data from the packet
        Motor1.UnpackData(message,UnpackedMessageID)
        # We are interested in position and speed data that was requested by calling
Send_Respond_Encoder_data
        print(f"Motor position is: {Motor1.position}")
        print(f"Motor speed is: {Motor1.speed}")

    else:
        print("No message after timeout period!")
    print("")
    time.sleep(1 )
```

Mobile robot guide

Intro

This guide is a really simple example on how you can control mobile robot based on **spectral micro BLDC drives** using xbox controller!

By following this guide you will be able to make something like this: Link!

> **Python API**
>
> This guide leverages Spectral BLDC python API

What you will need to create a simple mobile robot is:

- 2 x spectral micro BLDC - used for 2 wheels
- Wires to connect everything up: CAN wires, Power wires
- 2 x BLDC motors
- 1 x CAN adapter
- 1 x Drill battery adapter: Example - This is great way to power your BLDC motors with cheap and affordable batteries
- A Laptop or raspberry pi
- Xbox controller

> **Building the robot**
>
> In this guide we only recommend you the hardware you can use to build the robot you designed. We do not offer already made designs you can build!

Setup

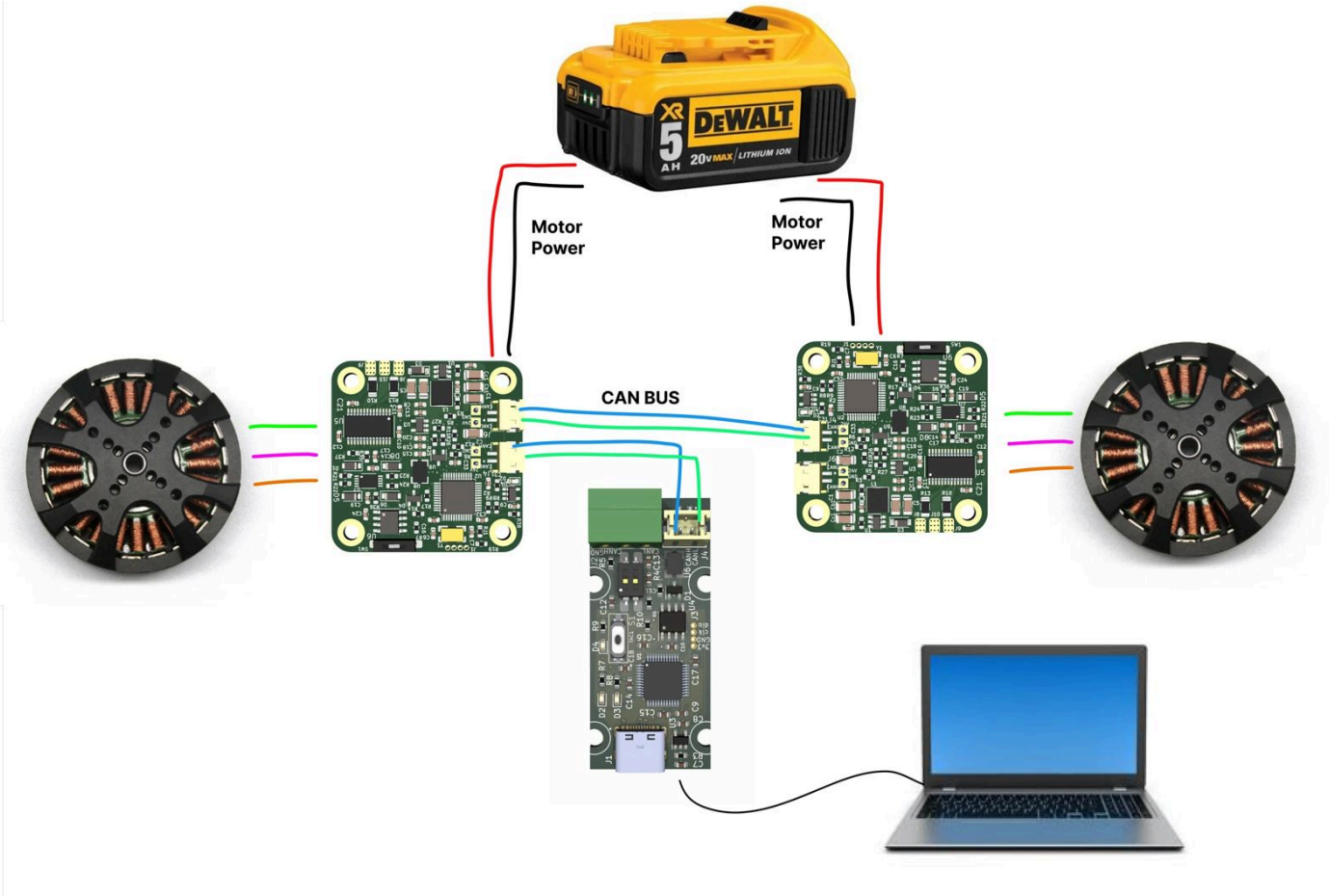


Figure 1: Two drivers powered from a drill battery, each with a gimbal motor, on one CAN bus with the CAN adapter and a laptop.

The setup will be the same for any kind of mobile robot. You will need to follow the diagram above to wire everything up.

- First you need to calibrate your Spectral BLDC drivers with the motors you are using.
- After that change the CAN ids of one of the drivers to CAN 1. You can do it by using UART commands: #CANID 1 and after that #Save
- Now connect CAN adapters CAN bus to one of the drivers and then from that driver connect to the second driver. Make sure that last driver in chain has its CAN termination resistor in "ON" state
- Connect the power to both drivers. You can daisy chain it or use 2 separate wires.
- You will also need to connect your xbox controller to the PC/laptop via Bluetooth.

The code below is really simple example that drives the robot using the left joystick of the controller. It sets a watchdog on both drivers, so the wheels stop if the program or the laptop stops; clear the watchdog error (`#Clear` or Send_Clear_Error) before the next run, as the code does at its start.

Spectral_mobile_robot_xbox.py

```
from inputs import get_gamepad
import math
import threading
import time
import Spectral_BLDC as Spectral

Communication1 = Spectral.CanCommunication(bustype='slcan', channel='COM3', bitrate=1000000)
Motor = []

# We are using 2 motors with IDs 0 and 1
Motor.append(Spectral.SpectralCAN(node_id=0, communication=Communication1))
Motor.append(Spectral.SpectralCAN(node_id=1, communication=Communication1))

# Stop the wheels when this program stops sending: a watchdog of 500 ms on each driver.
# Every Send_data_pack_1 below resets it; if none arrives for 500 ms, the driver stops the motor.
for m in Motor:
    m.Send_Clear_Error()
    m.Send_Watchdog(500)

timeout_setting = 0.00005

MAX_JOY_VAL = math.pow(2, 15)
DEADBAND_THRESHOLD = 0.1

# Read xbox controller and output left joystick x and y positions
def read_left_joystick():
    left_joystick_x = 0
    left_joystick_y = 0

    def monitor_left_joystick():
        nonlocal left_joystick_x, left_joystick_y
        while True:
            events = get_gamepad()
            for event in events:
                if event.code == 'ABS_Y':
                    left_joystick_y = event.state / MAX_JOY_VAL  # normalize between -1 and 1
                elif event.code == 'ABS_X':
                    left_joystick_x = event.state / MAX_JOY_VAL  # normalize between -1 and 1

                  # Apply deadband
                if abs(left_joystick_x) < DEADBAND_THRESHOLD:
                    left_joystick_x = 0
                if abs(left_joystick_y) < DEADBAND_THRESHOLD:
                    left_joystick_y = 0

    monitor_thread = threading.Thread(target=monitor_left_joystick)
    monitor_thread.daemon = True
    monitor_thread.start()

    while True:
        yield left_joystick_x, left_joystick_y

# Robot control
def control_robot():
    for x, y in read_left_joystick():
        # Assuming the robot has two wheels and speed can be controlled independently
        left_wheel_speed = y + x
        right_wheel_speed = y - x
```

```
        # Limit the speed between -1 and 1
        left_wheel_speed = max(min(left_wheel_speed, 1), -1)
        right_wheel_speed = max(min(right_wheel_speed, 1), -1)

        # Use the calculated speeds to control the robot's wheels
        print("Left Wheel Speed:", left_wheel_speed)
        print("Right Wheel Speed:", right_wheel_speed)

        # Send data to all motors
        Motor[0].Send_data_pack_1(None,int(left_wheel_speed * 1000000),0)
        Motor[1].Send_data_pack_1(None,int(right_wheel_speed * -1000000),0)

        for i in range(1, 4):  # read up to 3 replies
            message, UnpackedMessageID =
Communication1.receive_can_messages(timeout=timeout_setting)
            print(f"unpack{i} is: {UnpackedMessageID}")

            # Check if UnpackedMessageID is not None
            if UnpackedMessageID is not None:

                Motor[UnpackedMessageID[0]].UnpackData(message,UnpackedMessageID)
                print(f"Motor {UnpackedMessageID[0]}, speed is:
{Motor[UnpackedMessageID[0]].speed}")

        time.sleep(0.05)

if __name__ == '__main__':
    control_robot()
```

Quadruped robot guide

Intro

In this guide we will show you how you can control a leg of any quadruped robot using Spectral micro BLDC drivers. If you can control one of them you can control all the rest!

After you finish this guide you will be able to do something like this: video

Python API and toolbox

This guide leverages Spectral BLDC python API and Source robotics toolbox

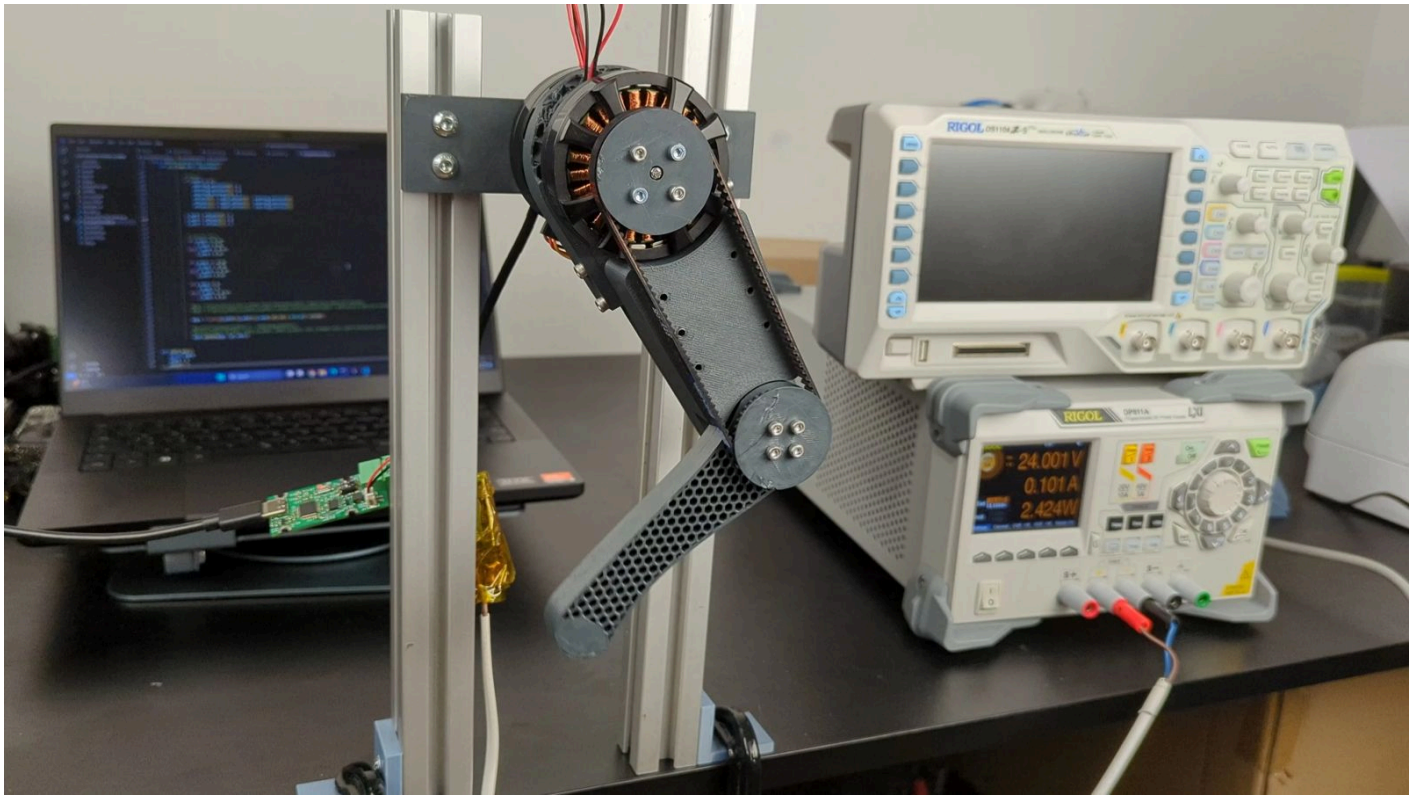


Figure 1: The robot leg: two motors, each with its driver, on one CAN bus.

Building the LEG

In this guide we only recommend you the hardware you can use to build the robot you designed. We do not offer already made designs you can build!

This guide will cover:

- How to define the robot leg kinematic chain
- How to execute simple trajectories with the leg

What you will need:

- 2 x spectral micro BLDC
- Wires to connect everything up: CAN wires, Power wires
- 2 x BLDC motors
- 1 x CAN adapter
- 1 x 24V power supply
- A Laptop or raspberry pi

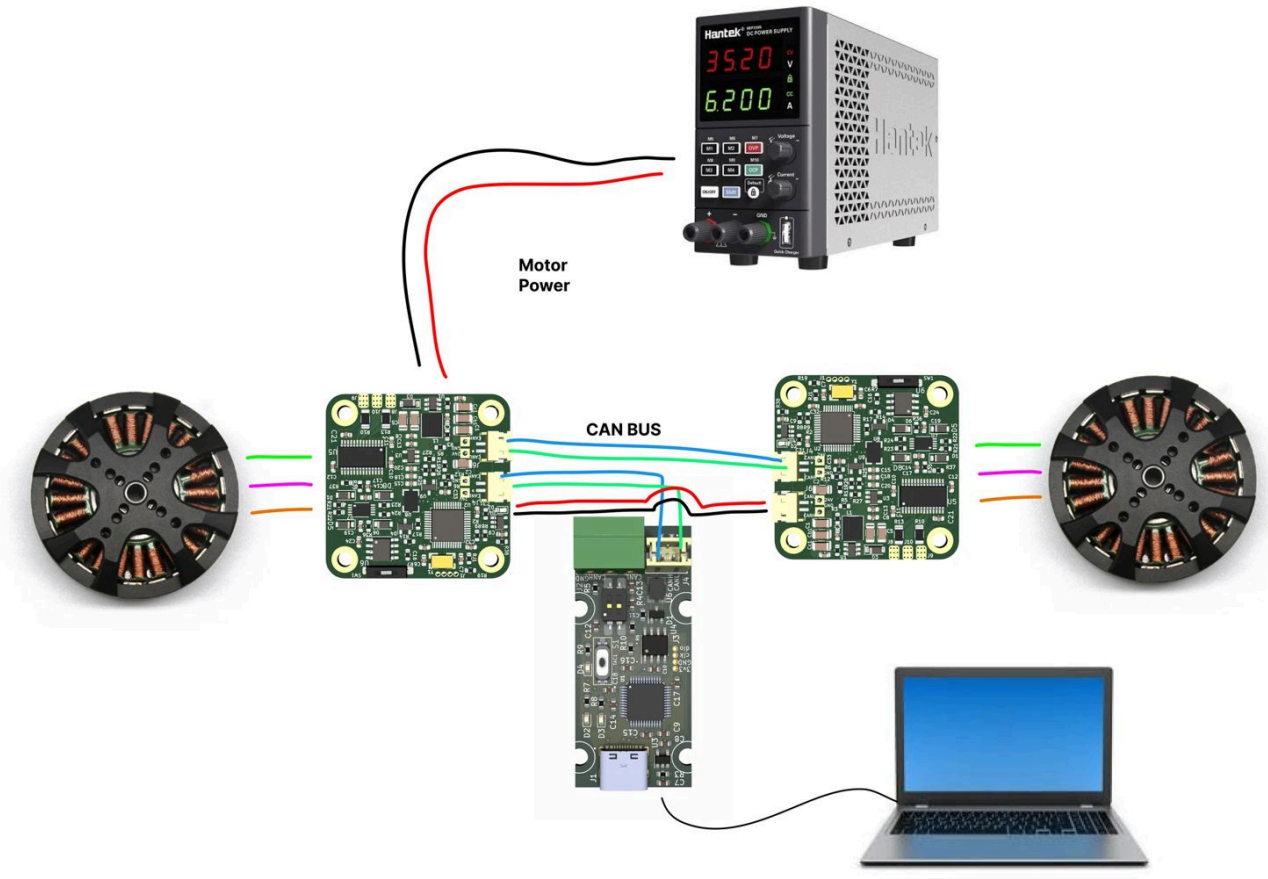
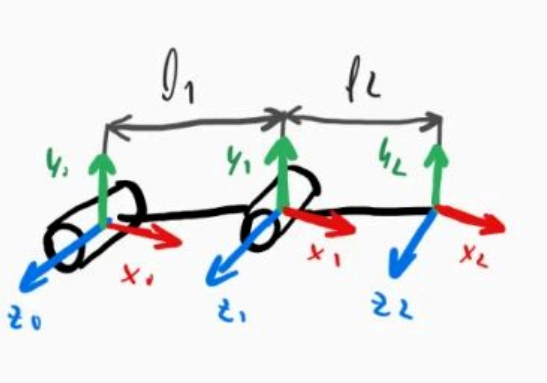


Figure 2: Two drivers daisy-chained on power and CAN, with the CAN adapter to the PC.

The setup will be the same for any kind of quadruped leg. You will need to follow the diagram above to wire everything up.

- First you need to calibrate your Spectral BLDC drivers with the motors you are using.
- After that change the CAN ids of one of the drivers to CAN 1. You can do it by using UART commands: #CANID 1 and after that #Save
- Now connect CAN adapters CAN bus to one of the drivers and then from that driver connect to the second driver. Make sure that last driver in chain has its CAN termination resistor in "ON" state
- Connect the power to both drivers. (Use daisy chain)

Leg kinematics



θ_j	d_j	a_j	α_j
q1	0	0.112	0.0°
q2	0	0.1	0.0°

Figure 3: The coordinate frames of the leg and its DH table.

First you will need to assign coordinate frames to your leg model. We already did it but if you want to learn more on how and why check this Link

Now we will need to find DH (Denavit-Hartenberg) table values. You can learn more on how and why we did it Here!

value l1 is 0.112 meters and value l2 is 0.1 meters. The frame 2 is the end / foot of our leg.

Testing / Calibrating the leg

Before running the main code lets check if the comms are working and get initial position values of our robot leg. This code will try to get data from the 2 motors of our robot leg.

We are most interested in position data of those 2 motors. Position the leg like shown in this image. Run the leg_calib code and write down position values of each motor.

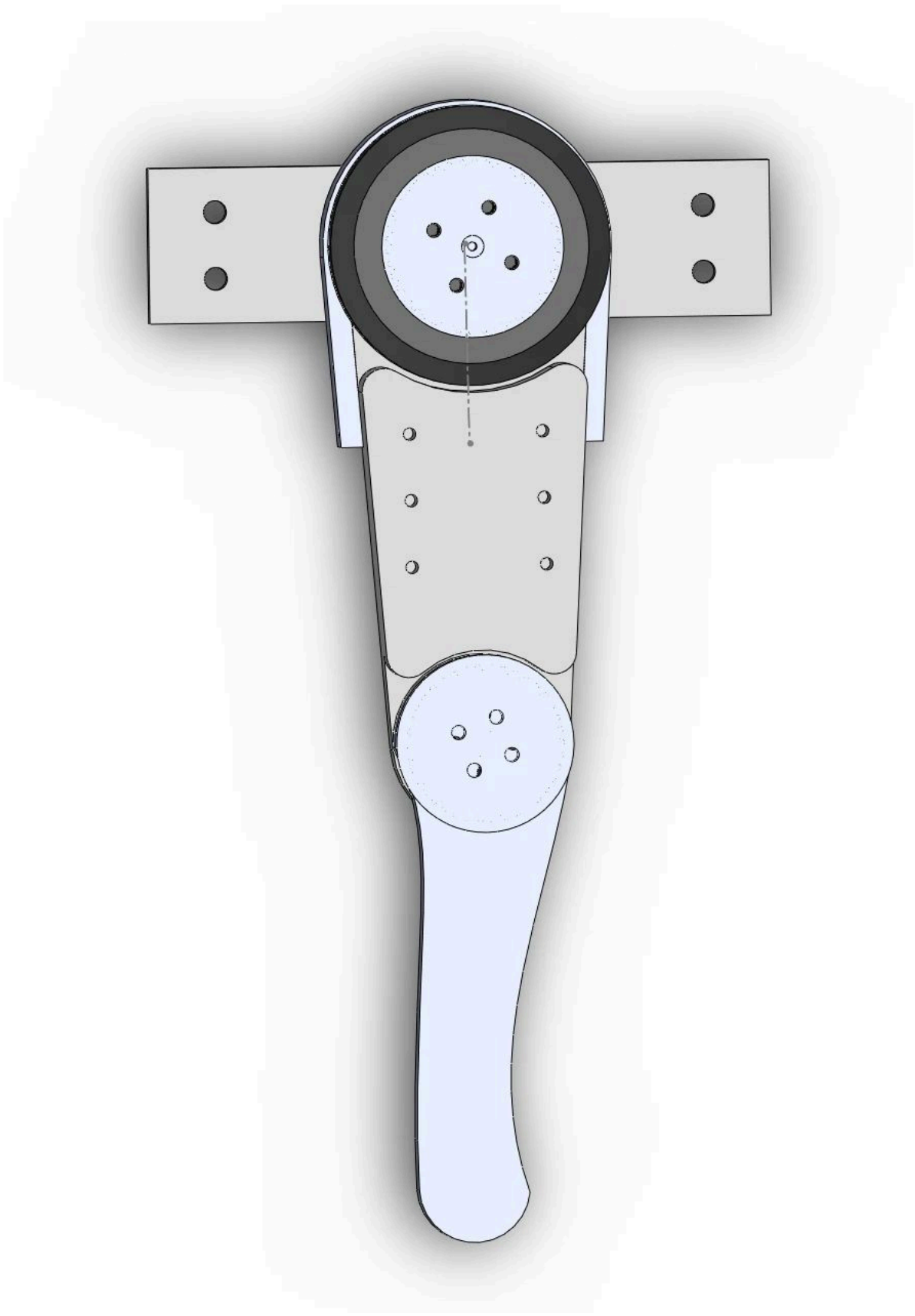


Figure 4: The leg in its reference pose and the positions leg_calib.py prints.

leg_calib.py

```
import Spectral_BLDC as Spectral
import time

Communication1 = Spectral.CanCommunication(bustype='slcan', channel='COM6', bitrate=1000000)
Motor = []

Motor.append(Spectral.SpectralCAN(node_id=0, communication=Communication1))
Motor.append(Spectral.SpectralCAN(node_id=1, communication=Communication1))

timeout_setting = 0.00005

while True:

    Motor[0].Send_Respond_Encoder_data()
    Motor[1].Send_Respond_Encoder_data()

    for i in range(1, 7):  # read up to 6 replies
        message, UnpackedMessageID = Communication1.receive_can_messages(timeout=timeout_setting)
        print(f"unpack {i} is: {UnpackedMessageID}")

        # Check if UnpackedMessageID is not None
        if UnpackedMessageID is not None:
            #print(Motor)
            # Update received id index; meaning that we received response from that CAN ID
            Motor[UnpackedMessageID[0]].UnpackData(message,UnpackedMessageID)
            print(f"Motor {UnpackedMessageID [0]}, speed is: {Motor[UnpackedMessageID[0]].speed},
current is {Motor[UnpackedMessageID[0]].current },pos is {Motor[UnpackedMessageID[0]].position}")
            """
            print(f"Motor {UnpackedMessageID[0]}, speed is: {Motor[UnpackedMessageID[0]].speed}")
            print(f"Error is: {Motor[UnpackedMessageID[0]].error}")
            print(f"Temperature rror is: {Motor[UnpackedMessageID[0]].temperature_error}")
            print(f"Encoder error is: {Motor[UnpackedMessageID[0]].encoder_error}")
            print(f"Vbus error is: {Motor[UnpackedMessageID[0]].vbus_error}")
            print(f"Driver error is: {Motor[UnpackedMessageID[0]].driver_error}")
            print(f"Velocity error is: {Motor[UnpackedMessageID[0]].velocity_error}")
            print(f"Current error is: {Motor[UnpackedMessageID[0]].current_error}")
            print(f"Estop error is: {Motor[UnpackedMessageID[0]].estop_error}")
            print(f"Watchdog error is: {Motor[UnpackedMessageID[0]].watchdog_error}")
            print(f"Calibrated is: {Motor[UnpackedMessageID[0]].calibrated}")
            print(f"Activated is: {Motor[UnpackedMessageID[0]].activated}")
            """

    time.sleep(0.5)
```

Follow a specific trajectory

Joint level control involves directly controlling the angles or positions of the robot's individual joints. Each joint is treated as an independent actuator, and the control system sends commands to each joint to achieve the desired angles.

For this example we will not use joint level control but a different method called Task/Cartesian Level Control. Task or cartesian level control focuses on controlling the position and orientation of the robot's end-effector in a Cartesian coordinate system (X, Y, Z coordinates and orientations).

In our robot leg case we don't have Z axes (or orientation) and our end-effector is foot (end of our robot leg).

In this example we will follow a circle trajectory i X,Y plane. We defined that trajectory with our function generate_circle_coords and with parameters:

- center - Tuple representing the center coordinates (x, y) in meters
- radius - Radius of the circle in meters
- speed - Speed of the end-effector in meters per second
- time_var - Elapsed time in seconds

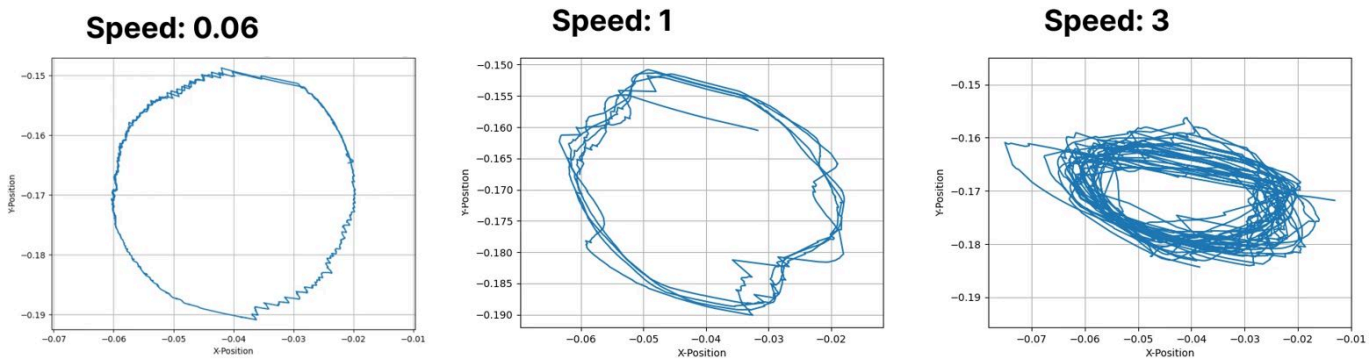


Figure 5: The circle the foot follows at a speed of 1 and of 3.

How precise you follow the trajectory depends on the speed you set. With low speeds you can track the circle perfectly. With larger speeds the system can't keep up and overshoots. This is due multiple reaseons like:

- Inertia and Momentum: At higher speeds, the inertia and momentum of the robot leg increase, making it harder to change directions quickly and accurately.
- Centrifugal and Coriolis Forces
- Trajectory Planning and Execution - at higher speeds our trajectory has less points to create a full circle. This is due a fixed rate we send data to our motors eg. every 20ms.
- Feedforward Control: Lack of proper feedforward control can lead to errors in trajectory tracking, as the controller might not anticipate the required accelerations and decelerations.

You can see that from the example with speed set to 1 and 3.

Example code

This code will make your robot leg draw the above circles. It might look complicated but it is not. Once you understand how this example works you can use it as a base for building more complicated projects.

2DOF_leg_demo.py

```
import Spectral_BLDC as Spectral
import SourceRoboticsToolbox as SourceRoboticsToolbox
import time
import numpy as np
# We are using v1.1.0
from roboticstoolbox import DHRobot, RevoluteDH, ERobot, ELink, ETS
from math import pi, sin, cos
from spatialmath import *
from oclock import Timer, loop, interactiveloop
import math
from typing import Union, Any, List, Optional, cast
INTERVAL_S = 0.02

Needed_joint_pos = np.array([14000,2430]) # These are the positions we got from calib code
Joint_reduction_ratio = [1, 0.90476190] # Reduction ratio we have on our joints

# Setup your CAN coms. Change the channel to match your CAN adapter
Communication1 = Spectral.CanCommunication(bustype='slcan', channel='COM6', bitrate=1000000)

# Define our motors
Motor: list[Union[Spectral.SpectralCAN,Spectral.SpectralCAN]] = []
Motor.append(Spectral.SpectralCAN(node_id=0, communication=Communication1))
Motor.append(Spectral.SpectralCAN(node_id=1, communication=Communication1))

# Define the joints
Joint: list[Union[SourceRoboticsToolbox.Joint,SourceRoboticsToolbox.Joint]] = []
Joint.append(SourceRoboticsToolbox.Joint(encoder_resolution = 14,
master_position=Needed_joint_pos[0], gear_ratio = Joint_reduction_ratio[0], offset = -np.pi/2,
dir = 0))
Joint.append(SourceRoboticsToolbox.Joint(encoder_resolution = 14,
master_position=Needed_joint_pos[1], gear_ratio = Joint_reduction_ratio[1], offset = 0, dir = 0))

timeout_setting = 0.001

initial = 0
initial_setup = [0,0]

testing = 5

# Initialize position values
position_values =  np.array([0.0,0.0])
Motor_values = np.array([0,0])
received_ids = [0,0]

# robot length values (metres)
l = [0.112, 0.1]

alpha_DH = [0,0]

# Create robot model using DH params
L1 = RevoluteDH(a=l[0],d = 0,alpha=alpha_DH[0])
L2 = RevoluteDH(a=l[1],d = 0,alpha=alpha_DH[1])
robot = DHRobot([L1,L2], name="Two link")

prev_pos = np.array([0,0])
speed_var = np.array([0,0])

def generate_circle_coords(center, radius, speed, time):
```

```
    """
    Generates x,y coordinates for a circle at a given time, considering constant speed.

    Args:
        center: Tuple representing the center coordinates (x, y) in meters.
        radius: Radius of the circle in meters.
        speed: Speed of the end-effector in meters per second.
        time: Elapsed time in seconds.

    Returns:
        Tuple containing the x and y coordinates in meters.
    """

    # Calculate angular displacement based on speed and time
    angular_displacement = speed * time

    # Use parametric equations for a circle to generate coordinates
    x = center[0] + radius * math.cos(angular_displacement)
    y = center[1] + radius * math.sin(angular_displacement)

    return (x, y)

def save_xy_data(filename, xy_data):
    """
    Saves a list of x,y coordinates to a text file.

    Args:
        filename (str): The name of the text file to save the data to.
        xy_data (list): A list of tuples containing x and y coordinates.
    """

    with open(filename, 'a') as file:
      for x, y in xy_data:
        file.write(f"{x},{y}\n")

# Example usage with update loop (adjust update_interval for your needs)
center = (-0.04, -0.17)
radius = 0.03
speed = 0.8
time_var = 0

# We are using oclock module to generate stable interval defined with INTERVAL_S
timer = Timer(interval=INTERVAL_S, warnings=False, precise=True)
while timer.elapsed_time < 1100000:

    t1 = time.perf_counter()
    #Motor[0].Send_Respond_Encoder_data()
    #Motor[1].Send_Respond_Encoder_data()
    Motor[0].Send_data_pack_1(Position = Needed_joint_pos[0],Speed = speed_var[0], Current = 0)
    Motor[1].Send_data_pack_1(Position = Needed_joint_pos[1],Speed = speed_var[1], Current = 0)

    # Handles data we get from the motors
    for i in range(1, 3):  # read up to 2 replies
        message, UnpackedMessageID = Communication1.receive_can_messages(timeout=timeout_setting)
        #print(f"unpack{i} is: {UnpackedMessageID}")

        # Check if UnpackedMessageID is not None
        if UnpackedMessageID is not None:
```

```
            # Update received id index; meaning that we received response from that CAN ID
            received_ids[UnpackedMessageID[0]] = 1
            Motor[UnpackedMessageID[0]].UnpackData(message,UnpackedMessageID)
            #print(f"Motor {UnpackedMessageID[0]}, position is:
{Motor[UnpackedMessageID[0]].position}")
            unwrapped_position_raw =
Joint[UnpackedMessageID[0]].unwrap_position(Motor[UnpackedMessageID[0]].position)
            Motor_values[UnpackedMessageID[0]] = Motor[UnpackedMessageID[0]].position
            position_values[UnpackedMessageID[0]] =
Joint[UnpackedMessageID[0]].get_joint_position(Motor[UnpackedMessageID[0]].position)

            if initial_setup[UnpackedMessageID[0]] == 0:
                initial_setup[UnpackedMessageID[0]] = 1

Joint[UnpackedMessageID[0]].determine_sector(Motor[UnpackedMessageID[0]].position)

    # Generate desired x,y coordinates with respect to time_var
    x, y = generate_circle_coords(center, radius, speed, time_var)

    # Create a matrix with those x,y coordinates
    T2 = SE3(x, y, 0)
    # Perform inverse kinematics (get joint data from x,y data)
    q1 = robot.ik_LM(T2, q0 = [-0.8,-1],mask = [1,1,0,0,0,0])
    Needed_joint_pos[0] = Joint[0].get_encoder_position(q1[0][0])
    Needed_joint_pos[1] = Joint[1].get_encoder_position(q1[0][1])
    speed_var[0] = ((Needed_joint_pos[0] - prev_pos[0]) / INTERVAL_S)
    speed_var[1] = ((Needed_joint_pos[1] - prev_pos[1]) / INTERVAL_S)
    print(speed_var)
    prev_pos[0] = Needed_joint_pos[0]
    prev_pos[1] = Needed_joint_pos[1]
    # Process or visualize the generated coordinates (x, y)
    #print(f"Time: {time_var:.2f}s, Coordinates: ({x:.4f}, {y:.4f})")

    # Update time
    time_var += INTERVAL_S

    t2 = time.perf_counter()
    #print(f"total time is {t2-t1}")

    T = robot.fkine(position_values)
    xy = T.t
    #print(f"{xy[0]},{xy[1]}")
    xy_data = [(xy[0], xy[1])]  # Assuming xy is a single pair of coordinates
    # Save data to a txt file
    save_xy_data('my_data.txt', xy_data)
    #time.sleep(0.05)
```

Plot data

Script that will plot the data of your leg:

plot_leg_data.py

```
import matplotlib.pyplot as plt

def plot_data(filename):
  """
  Plots data from a text file in the format:
  x1, y1
  x2, y2
  ...

  Args:
      filename (str): The name of the text file containing the data.
  """

  try:
    with open(filename, 'r') as file:
      data = file.readlines()

    x_data = []
    y_data = []
    for line in data:
      # Split the line based on comma (",")
      values = line.strip().split(',')

      # Check if there are exactly two values per line
      if len(values) != 2:
        raise ValueError(f"Invalid line format in file '{filename}': {line}")

      try:
        x = float(values[0])
        y = float(values[1])
      except ValueError:
        raise ValueError(f"Invalid data type in file '{filename}': {line}")

      x_data.append(x)
      y_data.append(y)

    # Ensure equal x and y scales
    plt.axis('equal')

    # Create the plot
    plt.plot(x_data, y_data)

    # Customize the plot (optional)
    plt.xlabel('X-Position')
    plt.ylabel('Y-Position')
    plt.title('Data Plot from ' + filename)

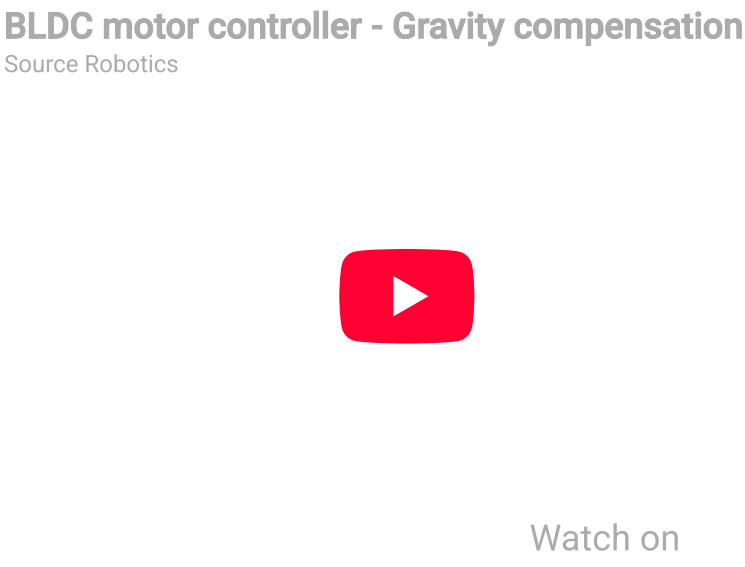
    plt.grid(True)
    plt.show()

  except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
  except (ValueError, TypeError) as e:
    print(f"Error: {e}")

# Example usage
filename = 'C:\\Users\\xx\\Desktop\\test_code\\env\\Robot_leg\\data_2.txt'  # Replace with your
actual filename
```

```
plot_data(filename)
```

Gravity compensation



What is Gravity Compensation?

Gravity compensation is a fundamental concept in robotic control that helps counteract the effects of gravity on a robot's joints and links. By applying the necessary motor current to balance this torque, the robotic joint can remain stationary at any position without additional external forces.

Why is it so important?

In robotics, a significant portion of motor power is consumed just fighting against gravity. Gravity compensation algorithms are standard in all industrial robots and collaborative robot arms because they provide: improved position accuracy and control, smoother motion planning, and more precise force control applications.

The gravity compensation algorithm works as a feedforward component to the motor PID control loops, reducing their computational load and improving overall performance.

The Approach

In this example, we use the Robotics Toolbox for Python to model a "robotic arm" with a single revolute joint. We compute the required torque using the Recursive Newton-Euler Algorithm (RNEA) and convert this torque into a motor current/torque that the motor controller will command the motor.

For more info check the full guide here: https://source-robotics.com/blogs/blog/gravity-compensation-in-robotics

Github repos with examples are here: Github repo, Github repo2

Bilateral teleoperation guide

What is bilateral teleoperation?

Bilateral teleoperation is a concept that Involves two-way communication where both the operator and the remote device send and receive feedback. The operator controls the device remotely and receives sensory feedback (such as force) from the device to adjust their actions.

The two grippers are virtually coupled to each other, allowing either to act as the master (the haptic interface) or the slave (the manipulator) in the system. We are not using a force sensor at the grippers but are relying on proprioception.

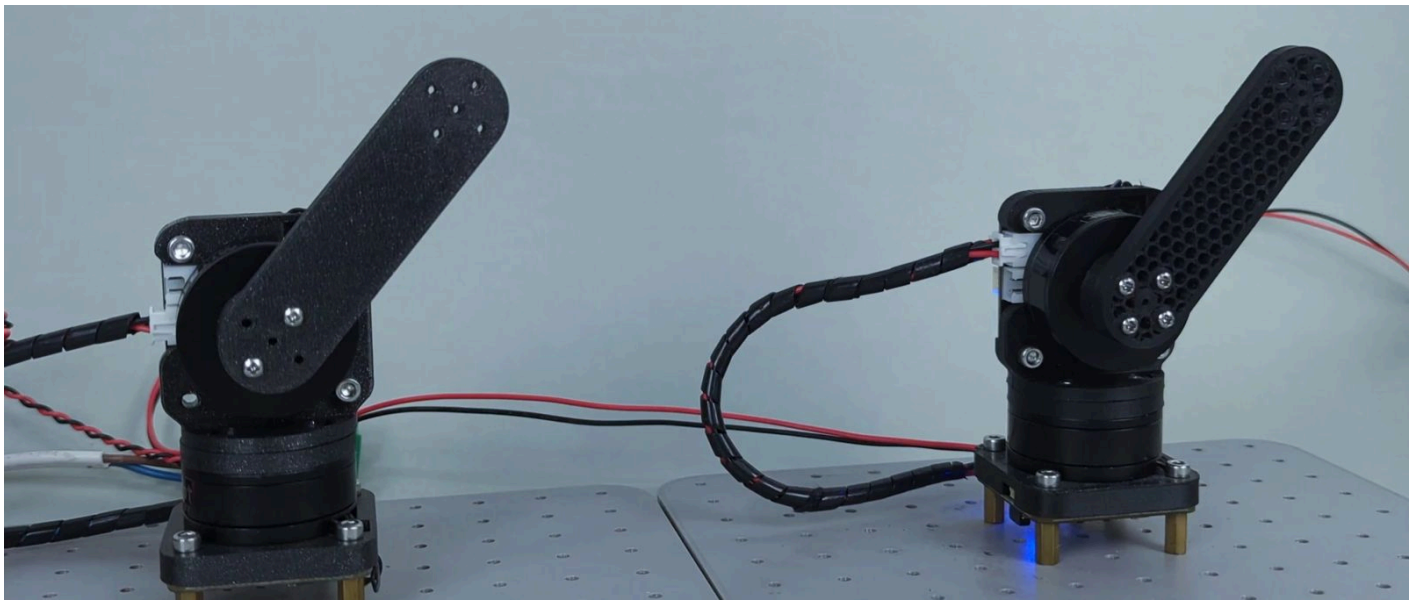


Figure 1: Two motors coupled by bilateral teleoperation: each follows the other and feels its force.

Watch these example videos to get a "feel" of the concept:

- Gripper bilateral
- 2DOF arm bilateral

What is this used for?

These kind of setup is perfect for: Teleoperation, Imitation learning, Virtual Environments...

The algorithm

The magic of this setup is done with these 2 simple formulas.

We got them from amazing Ben Katz paper, creator of mini cheetah quad: Link to the paper

$$\tau 1,i = Kp(\theta 2,i - \theta 1,i) + Kd(\dot{\theta} 2,i - \dot{\theta} 1,i) - K(\dot{\theta} 1,i)$$

$$\tau 2,i = Kp(\theta 1,i - \theta 2,i) + Kd(\dot{\theta} 1,i - \dot{\theta} 2,i) - K(\dot{\theta} 2,i)$$

Learn more!

You can read more on how it works in our blog post!

What you will need

We recommend these items to create a simple test jig:

- 2 x spectral micro BLDC - one for each motor or gripper
- Wires to connect everything up: CAN wires, Power wires
- 2 x BLDC motors or SSG48 grippers
- 1 x CAN adapter
- 1 x 24V power supply
- A Laptop or raspberry pi
- UART adapter

Setup

Here we are showing you the setup for any 2 motors, for example we are using 2 x ssg48 grippers that are connected in exactly the same setup.

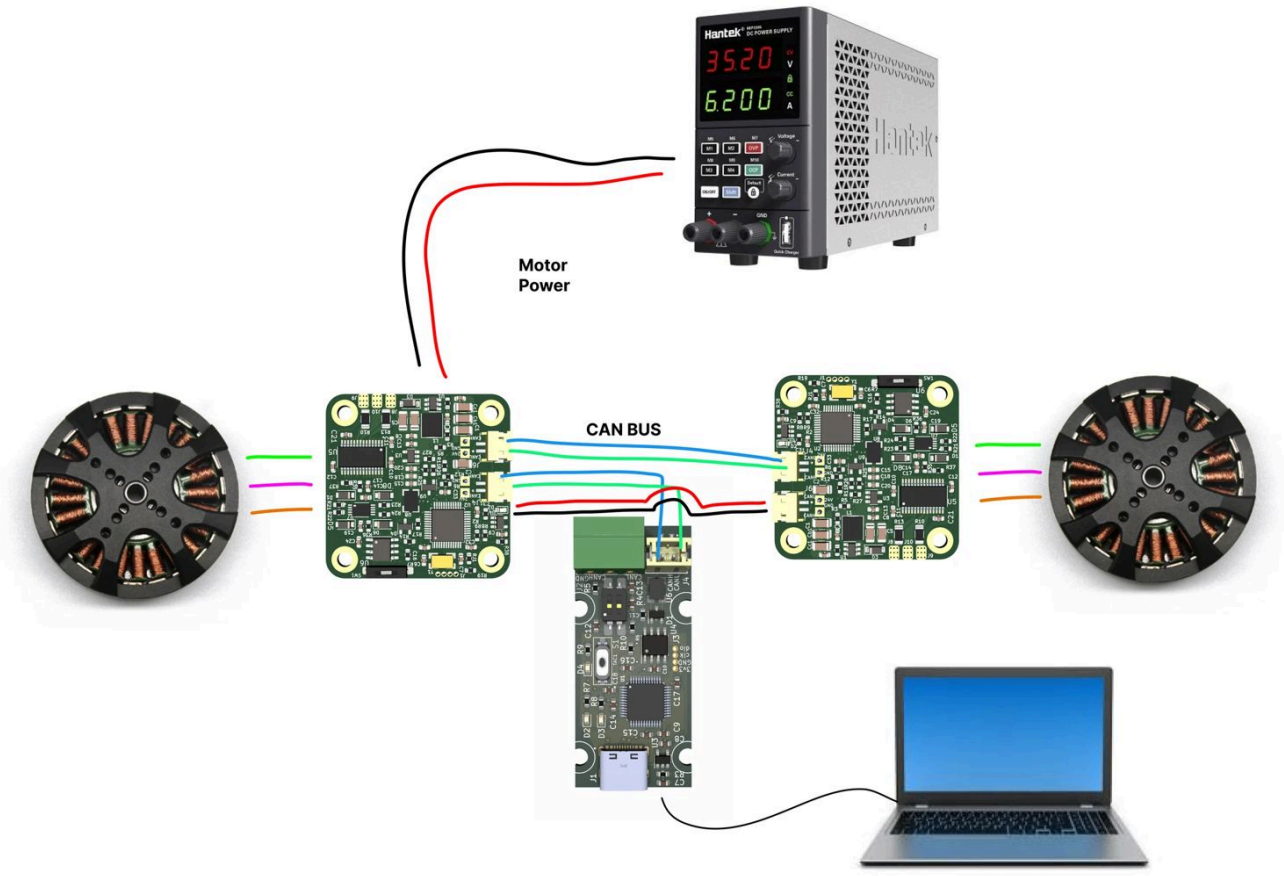


Figure 2: Two drivers daisy-chained on power and CAN, with the CAN adapter to the PC.

You will need to follow the diagram above to wire everything up.

- First you need to calibrate your Spectral BLDC drivers with the motors you are using.
- After that change the CAN ids of one of the drivers to CAN 1. You can do it by using UART commands: #CANID 1 and after that #Save
- Now connect CAN adapters CAN bus to one of the drivers and then from that driver connect to the second driver. Make sure that last driver in chain has its CAN termination resistor in "ON" state
- Connect the power to both drivers. You can daisy chain it or use 2 separate wires.

The code

This code will run on your PC and it will communicate with the drivers from the CAN adapter.

Advanced example

More advanced code that allows communication over UDP can be found here: Link

bilateral_teleoperation.py

```
import Spectral_BLDC as Spectral
import SourceRoboticsToolbox as SourceRoboticsToolbox
import time
import numpy as np
from math import pi, sin, cos
from spatialmath import *
from typing import Union, Any, List, Optional, cast

Needed_joint_pos = np.array([100,100]) # If using gripper you will need to adjust this
Joint_reduction_ratio = [1, 1] # Reduction ratio we have on our joints

Communication1 = Spectral.CanCommunication(bustype='slcan', channel='COM123', bitrate=1000000)

Motor: list[Union[Spectral.SpectralCAN,Spectral.SpectralCAN]] = []
Motor.append(Spectral.SpectralCAN(node_id=0, communication=Communication1))
Motor.append(Spectral.SpectralCAN(node_id=1, communication=Communication1))

Joint: list[Union[SourceRoboticsToolbox.Joint,SourceRoboticsToolbox.Joint]] = []
Joint.append(SourceRoboticsToolbox.Joint(encoder_resolution = 14,
master_position=Needed_joint_pos[0], gear_ratio = Joint_reduction_ratio[0], offset = 0, dir = 0))
Joint.append(SourceRoboticsToolbox.Joint(encoder_resolution = 14,
master_position=Needed_joint_pos[1], gear_ratio = Joint_reduction_ratio[1], offset = 0, dir = 0))

timeout_setting = 0.001

initial = 0
initial_setup = [0,0]

# Initialize position values
position_values =  np.array([0.0,0.0])
Motor_values = np.array([0,0])

speed_values = np.array([0,0])

position_send = np.array([0,0])
speed_send = np.array([0,0])
T0 = 0
T1 = 0

received_ids = [0,0]
Kp = 650
Kd = 0.00550 #  0.0060
Kk = 0.0003 #0.0003

while(1):
    t1 = time.perf_counter()
    Motor[0].Send_data_pack_1(Position = None,Speed = None, Current = int(T0))
    Motor[1].Send_data_pack_1(Position = None,Speed = None, Current = int(T1))

    for i in range(1, 3):  # read up to 2 replies
        message, UnpackedMessageID = Communication1.receive_can_messages(timeout=timeout_setting)
        #print(f"unpack{i} is: {UnpackedMessageID}")

        # Check if UnpackedMessageID is not None
        if UnpackedMessageID is not None:

            # Update received id index; meaning that we received response from that CAN ID
            received_ids[UnpackedMessageID[0]] = 1
```

```
            Motor[UnpackedMessageID[0]].UnpackData(message,UnpackedMessageID)
            #print(f"Motor {UnpackedMessageID[0]}, position is:
{Motor[UnpackedMessageID[0]].position}")
            unwrapped_position_raw =
Joint[UnpackedMessageID[0]].unwrap_position(Motor[UnpackedMessageID[0]].position)
            Motor_values[UnpackedMessageID[0]] = Motor[UnpackedMessageID[0]].position
            position_values[UnpackedMessageID[0]] =
Joint[UnpackedMessageID[0]].get_joint_position(Motor[UnpackedMessageID[0]].position)

            if initial_setup[UnpackedMessageID[0]] == 0:
                initial_setup[UnpackedMessageID[0]] = 1

Joint[UnpackedMessageID[0]].determine_sector(Motor[UnpackedMessageID[0]].position)

    # The magic
    T0 = Kp*( position_values[1] -  position_values[0]) + Kd*(Motor[1].speed - Motor[0].speed) -
Kk*(Motor[0].speed)
    T1 = Kp*( position_values[0] -  position_values[1]) + Kd*(Motor[0].speed - Motor[1].speed) -
Kk*(Motor[1].speed)
    # There are no delays here, we send as fast as possible!
```

SimpleFOC guide

Tested with Simple FOC@^2.3.2 release

Main firmware that runs on Spectral micro is Spectral firmware.

SimpleFOC is alternative to Spectral Firmware.
The library is intended both for:

- Beginners searching for a simple and user-friendly way to learn how to control BLDC and Stepper motors
- Advanced users ready to dig deeper into the FOC algorithm and optimize the code for their particular application/hardware.

More info about simple FOC:

- Forum
- DOCS
- Discord

SimpleFOC forum and discord is best place to look for help when developing with SimpleFOC library

How to install

Follow this guide

This replaces the bootloader

This project has no upload offset, so an upload with the programming adapter writes SimpleFOC at `0x08000000`, over the bootloader. Firmware updates over UART or CAN stop working. To go back to the Spectral firmware, install the bootloader again as in Installing the bootloader on an older board, then flash a release.

Your platformio.ini should look like this:

platformio.ini

```
[env:genericSTM32F103CB]
platform = ststm32
board = genericSTM32F103CB
framework = arduino
upload_protocol = stlink
build_flags =
debug_tool = stlink
monitor_speed = 115200
lib_deps =
    askuric/Simple FOC@^2.3.2
lib_archive = false
```

SimpleFOC guides

If you want to develop more advanced applications with SimpleFOC you can follow these tutorials:

- Getting started
- Writing the code
- SimpleFOC Driver and Support Library

Basic example

This example will spin the motor in open loop and enable serial monitor.
Use this code as a base for more advanced code you will develop with SimpleFOC.
The Spectral-Micro-Simple-FOC repository has this project and more examples in its `examples` folder, among them voltage-torque FOC with the encoder.

main.cpp

```
#include "Arduino.h"
#include "SimpleFOC.h"

// Driver PWM pins
#define PWM_CH1 PA_2
#define PWM_CH2 PA_1
#define PWM_CH3 PA_0

// Encoder pins
#define MOSI PB_5
#define MISO PB_4
#define CLK PB_3
#define CSN PA_15

#define SENSE1 PA_4 // Inline current sense
#define SENSE2  PA_3 // Inline current sense

#define TX_COM PB_6
#define RX_COM PB_7

// Current sense resistors
#define SENSE_RESISTOR 0.025f
#define CURRENT_AMP_GAIN 20

// Serial pins
#define Serial Serialx
HardwareSerial Serialx(RX_COM, TX_COM); // PA3, PA2 RX,TX

// BLDC motor & driver instance
// BLDCMotor motor = BLDCMotor(pole pair number, phase resistance (optional) );
BLDCMotor motor = BLDCMotor(11);
//  BLDCDriver3PWM( pin_pwmA, pin_pwmB, pin_pwmC )
BLDCDriver3PWM driver = BLDCDriver3PWM(PWM_CH1, PWM_CH2, PWM_CH3);

//target variable
float target_velocity = 0.1;

void Init_pins();

void setup() {

  // Init driver enable pins and LED
  Init_pins();

  // pwm frequency to be used [Hz]
  driver.pwm_frequency = 20000;
  // power supply voltage [V]
  driver.voltage_power_supply = 24;
  // Max DC voltage allowed - default voltage_power_supply
  driver.voltage_limit = 5;
  // driver init
  driver.init();
    // link the motor and the driver
  motor.linkDriver(&driver);
  motor.controller = MotionControlType::velocity_openloop;

  // init motor hardware
  motor.init();
```

```
    // use monitoring with the BLDCMotor
  Serial.begin(115200);
  Serial.println("Motor ready!");
  _delay(1000);
  // monitoring port
  motor.useMonitoring(Serial);

}

void loop() {
  // open loop velocity movement
  // using motor.voltage_limit and motor.velocity_limit
  motor.move(target_velocity);
  // monitoring function outputting motor variables to the serial terminal
  motor.monitor();

}

void Init_pins(){
  pinMode(PA_9, OUTPUT);
    pinMode(PA_8, OUTPUT);
    pinMode(PA_10, OUTPUT);
    pinMode(PB_15, OUTPUT);
    pinMode(PB_14, OUTPUT);
    pinMode(PB_13, OUTPUT);

  digitalWrite(PA_9,HIGH);
  digitalWrite(PA_8,HIGH);
  digitalWrite(PA_10,HIGH);
  digitalWrite(PB_14, HIGH);
  digitalWrite(PB_15, HIGH);
  digitalWrite(PB_13, HIGH);
}
```

Support

Getting help

- **Questions and discussion**: the Source Robotics Discord.
- **A problem with the driver or the firmware**: an issue in the firmware repository; see Report a problem for what to include.
- **A mistake on a page of this site**: every page ends with a link to edit it and one to report it; see Contributing.
- **Orders and hardware**: source-robotics.com.

Before you ask, Troubleshooting goes through the common symptoms, and `#Info` tells most of what anyone will ask you for.

Report a problem

Open an issue in the firmware repository for the driver and its firmware, or in Spectral-BLDC-Python for the Python library. Include:

1. **The output of `#Info`, `#Error` and `#Param`**, copied from the terminal. They carry the hardware and firmware version, the motor parameters, the calibration state, the errors and the gains.
2. **The motor**: type, and the resistance, inductance and pole pairs calibration found (`#Info` prints them).
3. **The supply**: voltage, and whether it has a current limit.
4. **What you did and what happened**: the commands, in order, and the driver's answers; the calibration log if it is about calibration.
5. **How it is wired**: a photo of the board on the motor and of the cables helps more than a description.

For a problem with this documentation, the *Report it* link at the end of the page opens an issue in the documentation repository with the page already named.

Contributing

This documentation lives in Source-Robotics/Spectral-BLDC-docs. A small fix can be made from the browser: the *Edit it on GitHub* link at the end of every page opens the page's source, and GitHub turns your change into a pull request. For anything larger, open an issue first.

How the site is built

- The pages are Markdown, built with MkDocs and the Material theme; the repository's `README.md` says how to preview it locally.
- The site describes one firmware release, named in `versions.yml`. The command IDs, frame lengths, the UART command list and the defaults in the Reference are read from the firmware's source at that release's tag by `tools/sync_firmware.py`; do not edit those facts in the pages. The descriptions of the commands are in `docs/includes/can_commands.yml` and `uart_commands.yml`.
- Every push runs a strict build (a broken link fails it) and the tool tests. A push to `main` publishes the site under its firmware version with mike.

House rules

- Facts come from the firmware. When a page and the firmware disagree, the firmware is right; fix the page, or report the firmware.
- Spell commands exactly as the firmware does, even where the spelling looks wrong.
- Write for someone who has the driver in front of them and wants to do one thing: say what to do, in order, then why.
- Three kinds of boxes: a note, a notice (damage) and a danger (injury). Use them for those, not for emphasis.
- Images: under 2400 px on the long side, with a caption that says what the reader should see.

Licence

Text and images: CC BY-SA 4.0: copy, adapt and share them, also commercially, if you credit Source Robotics and share your changes under the same licence. Code: the MIT licence, which covers the code examples in the pages and the site's own code. The Source Robotics name, the product names and the logos are not covered by either. By contributing you agree that your contribution is published under these licences.