PRELIMINARY

FIRST LAYER ROBOTICS

MOTION CORE

USB Motion Controller for Differential-Drive Mobile Robots

Getting Started — Python

APPLIES TO: Motion Core firmware 1.9.6 (CONFIG 23) · ORCP v1.1, conformance Level 2

AUDIENCE: Integrators and students bringing up a new two-wheeled robot platform

YOU WILL FINISH WITH: A configured, calibrated Motion Core driving your robot from Python and from the keyboard.

COMPANION GUIDE

If you intend to drive the platform from ROS 2, use Motion Core — Getting Started (ROS 2) instead. It covers the same board setup, then diverges at the software layer.

CONTENTS

1. WHAT YOU NEED

Hardware

ITEM	NOTES
Motion Core controller	Ships pre-flashed and factory-calibrated
Battery pack, 9–20V DC	Factory thresholds assume 3S Li-ion/LiPo; reconfigurable for other chemistries
Two brushed DC gearmotors with quadrature encoders	**Up to 4A continuous per side. Check whether your encoders need 5V or 3.3V** — see §2
Normally-closed e-stop switch	**Mandatory** — or a jumper link fitted to J3. See §2
Normally-closed charger-detect switch	**Mandatory** — or a jumper link fitted to J2. See §2
USB-C cable	USB 2.0 rated, data-capable. Does not power the board — see §2
A chassis	Two driven wheels plus a castor or equivalent
XT60 battery lead	To match the board's power input

Host machine

- macOS, Linux, or Windows
- Python 3.9 or newer
- No driver installation is required on any platform — the board enumerates as a standard USB CDC serial device
- A Chromium-based browser (Chrome or Edge) to run the web configurator — see §5

Bench space

Section 6 drives the robot about a metre under power. Keep a clear, flat area of roughly 3 m × 3 m, and keep the e-stop within reach.

2. WIRING IT UP

All connectors are labelled on the board silkscreen.

Connectors

REF	SILKSCREEN	FUNCTION	CONNECTOR TYPE
J1	VBAT 9V-20V DC	Battery / DC supply input	2-pin XT60
J2	CHARGER	Charger-detect interlock — mandatory	2-pin JST XH
J3	ESTOP	E-stop interlock — mandatory	2-pin JST XH
J4	—	Regulated 5V auxiliary output, 6A	2-pin Molex Mini-Fit Jr.
J5	CAN	CAN bus — factory-fit variant only	4-pin JST XH
J6	EXPANSION	Reserved — no function at this release	6-pin JST XH
J7	USB	USB-C host interface	USB-C receptacle
J8	ENCODER LEFT	Left encoder input	5-pin JST XH
J9	MOTOR LEFT	Left motor output	2-pin Molex Mini-Fit Jr.
J10	ENCODER RIGHT	Right encoder input	5-pin JST XH
J11	MOTOR RIGHT	Right motor output	2-pin Molex Mini-Fit Jr.
JP1	ENC VOLTAGE	Encoder supply select, 5V / 3.3V	3-pin header + jumper
JP2	120Ω	CAN bus termination	Header + jumper
CN1	PGM	SWD programming — factory / service use	6-way header
—	RESET	Reset pushbutton	—

Mating connectors, jumper links and reference cabling are supplied with the Rapid Integration Kit variant (FLR-MC-1-2M12-4-RIK) if you would rather not source them yourself.

> **NOTE**
> **Reading the pinouts below.** Signal names are given by **pin number**, which is authoritative. On the multi-way connectors the silkscreen abbreviations are printed in the opposite order to the pin numbering, so identify pin 1 from the board marking rather than assuming left-to-right.

Step 1 — the two mandatory interlocks (J2 and J3)

> **CAUTION**
> Read this before anything else. A board with J2 or J3 left open will not drive its motors, no matter what your software does — and nothing on the software side will look broken.

Motion Core has two independent hardware interlocks, both wired for normally-closed (NC) contacts, and both built into the drive-enable logic at hardware level. Neither can be overridden by the host.

	J3 — E-STOP	J2 — CHARGER DETECT
Intended switch	NC emergency-stop pushbutton	NC switch on the charging connector
Loop closed	Motors can be enabled	Motors can be enabled

	J3 — E-STOP	J2 — CHARGER DETECT
Loop open	Relay de-energises — motor supply physically disconnected from the H-bridges	Drive-enable inhibited at hardware level
Reported to host	estop=1, fault=ESTOP	fault=CHARGER

The e-stop is a de-energise-to-trip relay chain: opening the loop physically disconnects the H-bridge outputs from the motor terminals. It works even if the processor has crashed. The charger interlock is a parallel hardware path that stops the robot driving away while it is plugged in to charge.

> **WARNING**
>
> **If you are not fitting one of these switches, you must fit a jumper link.** Both loops are mandatory for the drive to operate:
>
> • No e-stop fitted → **link J3 pin 1 to pin 2**
>
> • No charger switch fitted → **link J2 pin 1 to pin 2**
>
> Leaving either connector open holds the motor outputs disabled at hardware level, regardless of host commands. This is the most common reason a new board appears to do nothing.

Both connectors share the same pinout:

PIN	SIGNAL	NOTES
1	GND	Ground
2	ESTOP / CHGDET	Dry-contact loop input, internal ~40 kΩ pull-up to 3.3V

These are **dry-contact inputs** — connect a switch, not a voltage. Maximum applied voltage is 3.6 V.

On a robot people will stand near, fit a real e-stop. The link is for bench bring-up, and for machines where the e-stop lives elsewhere in a larger safety chain.

Recovery from either interlock needs the loop closed **and** an explicit `ENABLE ON` from the host. Closing the loop alone does not restart the robot.

Step 2 — set the encoder supply voltage (JP1)

> **WARNING**
>
> **Do this before you plug an encoder in.** JP1 selects whether the encoder supply pin carries **5V or 3.3V**. Applying 5V to a 3.3V-only encoder can destroy it.

PIN	SIGNAL
1	5V
2	ENC_VCC — routed to both encoder connectors
3	3.3V

Fit the jumper between pin 2 and whichever supply your encoders need.

A jumper must be fitted for the encoders to work at all — with JP1 open, `ENC_VCC` is unpowered. JP1 feeds both J8 and J10, so you cannot mix a 5V encoder on one side with a 3.3V encoder on the other.

Step 3 — wire the encoders (J8, J10)

J8 pairs with Motor Left, J10 with Motor Right. Both share this pinout:

PIN	SIGNAL	SILKSCREEN	NOTES
1	**GND**	GND	Ground
2	**ENC_VCC**	VIN	Encoder supply — 5V or 3.3V per JP1
3	**ENC_A**	A	Quadrature channel A
4	**ENC_B**	B	Quadrature channel B
5	**Shield**	SHIELD	Optional cable-shield ground

Points worth knowing:

- **A and B channels only.** There is no index/Z input; index pulses are not used.
- Inputs are level-shifted and accept **3.3V or 5V** logic, and **open-collector or push-pull** outputs.
- Maximum input frequency is **100 kHz per channel** (400 kHz decoded at ×4).
- If your cable has no shield, leave pin 5 unconnected. Do not bond the shield at both ends.

A/B order determines the reported direction of travel. Getting it backwards is normal and expected — **§5 step 2 fixes it in software, so do not rewire to correct direction.**

Step 4 — wire the motors (J9, J11)

J9 is Motor Left, J11 is Motor Right.

PIN	SIGNAL	NOTES
1	**M+**	Motor output, positive — forward direction
2	**M−**	Motor output, negative

Rated **4A continuous per channel** across the full 0–40 °C ambient range, and **8A peak** for short acceleration events of a few seconds.

As with the encoders, polarity sets which way the wheel turns — **do not rewire to fix direction**, §5 step 2 corrects it in configuration.

Step 5 — power (J1) and the auxiliary output (J4)

J1 — battery input. 9–20V DC, XT60, polarity marked with + on the PCB.

> **WARNING**
> **USB-C is data-only.** The board draws no operating power from USB VBUS and supplies none to the host; USB Power Delivery is not implemented in either direction. **Motion Core will not power up or enumerate from USB alone — it must be supplied from J1.** If the board does not appear as a serial port, check the battery before you check the cable.

J4 — regulated 5V auxiliary output. For sensors, an SBC, or other 5V loads.

PIN	SIGNAL	NOTES
1	**+5V**	Regulated 5V, 6A maximum
2	**GND**	Ground

Output holds 4.8–5.2V across the full load and temperature range and is short-circuit protected. Its current is monitored on-board and reported to the host, so an overloaded rail or a failing sensor is visible in software. Warning thresholds are configurable (`aux5v.i_warn`, `aux5v.v_warn`) and raise `! WARN AUX5V`.

The 5V pins on J5 and J6 are limited to **500 mA each** and are not a substitute for J4.

Step 6 — power up

1. Fit JP1 for your encoder voltage.
2. Connect motors (J9/J11) and encoders (J8/J10).
3. Fit the e-stop (J3) and charger-detect (J2) loops — switch or link.
4. Connect the battery (J1).
5. Connect USB (J7).

Allow **1.5–2.5 s** from applying battery power to the board appearing as a serial port on the host.

Safe-start behaviour

Motor outputs are held disabled at power-up, regardless of anything the host is doing. They enable only after the host has connected and issued an explicit command. A board that resets mid-drive comes back stopped — it will not resume motion on its own.

Status LEDs

LED	MEANING
Red, blinking ~1 Hz	Application firmware running normally
Green, blinking ~5 Hz	Board is sitting in the bootloader

If you see the green bootloader blink on a board you have not deliberately put into update mode, see §12.

Wiring practice

These are not style preferences. **CE/UKCA compliance is conditional on following them**, and departures may require system-level EMC mitigation.

- **Twist the motor pairs** (`M+`/`M−`) to reduce radiated EMI. Near sensitive electronics, use shielded motor cable with the shield bonded to chassis ground.
- **Use twisted-pair shielded encoder cable**, particularly for runs longer than 0.5 m. Encoder noise shows up as velocity jitter and phantom stall faults.
- **Route the e-stop loop away from motor wiring**, and through any external safety circuits before it reaches J3.
- **Use a USB 2.0-rated cable.** Beyond 1 m use a cable with active or passive signal conditioning; 3 m is the specification limit.
- Size motor conductors for your actual motor current with derating headroom — gauge selection is the integrator's responsibility.
- Motion Core uses a **single ground reference**; battery, signal and shield grounds are joined on-board. In multi-controller installations, avoid creating ground loops through secondary chassis connections.

Power budget

CONTRIBUTOR	DRAW
Motors	Up to 2 × 4A × V_BAT — roughly 96W at 12V
5V auxiliary	Up to 6A × 5V = 30W, drawn from V_BAT via the on-board regulator
Host SBC, if powered from J4	Typically 5–15W

Mounting and cooling

The board is designed for **natural convection with no enclosure restriction**. Mount it on **5 mm standoffs with both sides clear for airflow**. Installing it in a sealed enclosure, or at elevated ambient temperature, may require derating the motor current below 4A per channel.

Board outline is 115.0 × 90.0 mm, 1.6 mm thick, with four mounting holes on a 105.0 × 80.0 mm pitch. Maximum component height is 16 mm above the top surface.

CAN (JP2)

CAN is a **factory-fit variant** (`FLR-MC-1-2M12-4-C`) and is **mutually exclusive with USB** — on CAN-variant boards the firmware brings up the CAN peripheral in place of the USB CDC interface. Standard boards leave the transceiver, connector and termination unpopulated.

If you have a CAN board sitting at one **end** of the bus, fit JP2 for the 120 Ω termination; omit it mid-bus. This guide uses USB throughout.

3. FIRST CONTACT

Connect the board over USB and find its serial port.

> **NOTE**
> **The board must be powered from the battery (J1).** USB-C is data-only — it neither powers Motion Core nor draws from it. Allow **1.5–2.5 s** after applying battery power before the serial port appears.

macOS

```
ls /dev/cu.usbmodem*
```

Use the `/dev/cu.*` entry, not `/dev/tty.*` — the `tty` variant blocks waiting for carrier detect and will appear to hang.

Linux

```
ls /dev/ttyACM*
```

If you get a permission error later, add yourself to the `dialout` group and log back in:

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

Windows

Check Device Manager under *Ports (COM & LPT)* for a `COM` port. The device identifies itself as **"FLR Motion Core"**.

Say hello

The board speaks line-oriented ASCII at **115200 baud, 8N1**. Any serial terminal works. Type `PING` and press Enter:

```
PING
OK PONG
```

Then ask what it is:

```
INFO
OK INFO fw=1.9.6 bl=1.4.0 hw=MC1 proto=ORCP/1.1 level=2 uuid=…
        build=… serial=MC1-2626-00001 vbus_v=5.02 vbus_present=1 vdda=3.31
```

- `fw` — firmware version
- `proto` — protocol version. **This is the number that matters for software compatibility**, not `fw`
- `level` — conformance level; Level 2 is what enables configuration commands
- `hw=MC1` — the board's internal hardware identifier for Motion Core
- `serial` — the board's traceability serial, or `UNSET`
- `uuid` — a unique chip ID, useful for telling two boards apart

And what it is doing:

```
STATUS
OK STATUS preset=SLOW mode=VEL en=1 fault=OK estop=0 … vbat=12.41 battery=87% …
```

`STATUS` returns a single long line — well over 256 bytes. If you are writing your own terminal integration, read to the newline; do not assume a fixed-size buffer is enough.

Two notes on the link:

- **The baud rate is nominal.** This is a USB CDC device, so baud, parity and flow control do not affect the physical link and are not enforced by the board. 115200 8N1 is recommended only because most host libraries require *some* valid setting before they will open a port.
- **Command lines are limited to 128 characters** including the newline; longer lines are rejected with `ERR TOO_LONG`. Commands are terminated with `\n`, and a preceding `\r` is accepted. Responses come back terminated with `\r\n`.

Host OS support: Linux (kernel 3.0 or later, no driver installation needed), macOS 10.15 or later, and Windows 10 or later.

If `PING` returns nothing, jump to §12.

4. THE SAFETY MODEL — READ BEFORE YOU MOVE ANYTHING

Motion Core will not let you drive a robot at full power by accident. Three mechanisms are worth understanding before you send a motion command.

Presets

The board boots in **SLOW** every time. It never remembers that you were in NORMAL.

	SLOW	NORMAL
Duty limit	30 %	90 %
Motors enabled	Automatically	Only after ENABLE ON
Stale-command timeout	Off by default	250 ms
Heartbeat required	No	Yes, every 500 ms

SLOW is the teaching and bring-up default: you can drive immediately, you cannot go fast, and nothing times out while you think. NORMAL is the operating mode, and it demands that the host prove it is still alive — both by sending motion commands regularly and by sending a separate heartbeat. If either lapses, the board brakes and latches a fault.

Switch with `PRESET SLOW` or `PRESET NORMAL`. There is no "FULL" preset.

The hardware interlocks

Two of them — e-stop (J3) and charger-detect (J2) — and both sit below software. If either loop is open, the motor outputs are disabled in hardware, whatever the host asks for; the e-stop physically disconnects the motor supply via a de-energised relay. See §2 — if you have not fitted switches, you need jumper links on both.

Test the e-stop before every session — §5 step 1.

Safe-start

Motor outputs are held disabled at power-up regardless of host state, and enable only after the host connects and issues an explicit command. A board that resets mid-drive comes back stopped — it does not resume motion on its own.

Faults latch

This is the single most important behaviour to internalise:

> ***A fault stays latched until you send ENABLE ON.***

Motion Core does not silently recover. If the battery dips, or a wheel jams, or the e-stop is pressed, the board stops, records *why*, and stays stopped. The condition clearing is not enough — you must acknowledge it.

`ENABLE ON` is therefore the universal recovery command, and in the teleop application it is bound to the **R** key. Three faults need something done first:

- `ESTOP`, `LOWBATT`, `CHARGER` — remove the physical condition, *then* `ENABLE ON`

- `CASCADE_NOGAIN` — a configuration error; see §11
- `BOOTLOADER_MISMATCH` — requires a firmware update; see §11

Full list in §11.

5. CONFIGURING FOR YOUR PLATFORM

Motion Core ships knowing nothing about your robot. This section teaches it.

Configuration splits cleanly in two, and the distinction matters:

- **Robot configuration** — geometry, motor limits, battery chemistry. Yours to set. This section.
- **Factory calibration** — current-sense scaling and offsets, the battery divider ratio, the MCU temperature coefficients. Measured per board at manufacture. **Do not change these.** They are `current.scale`, `current.offset_left`, `current.offset_right`, `batt.divider_ratio`, `mcu.temp_v25`, `mcu.temp_slope` and `aux5v.shunt_ohms`.

The `DEFAULTS` command respects that split — it resets your robot configuration but **preserves the factory calibration group**. It is safe to use when a configuration experiment goes wrong.

> **WARNING**
> **SAVE before you power-cycle.** Configuration changes take effect immediately but live only in RAM. A power cycle before `SAVE` discards everything you configure in this section — the single most common bring-up mistake. Full save/load workflow in §7.

Two ways to configure — the web configurator is preferred

The **Motion Core Web Configurator** is the recommended way to work through this section. It runs in a browser, connects to the board over USB, and gives you:

- the full configuration surface as a grouped, validated form — no key names to type and no SET syntax to get wrong
- live status and telemetry while you work
- battery chemistry applied as a single profile selection rather than four separate values
- fault codes decoded into plain language

Use it unless you specifically need scripted or headless configuration.

> **NOTE**
> **Browser requirement.** The configurator uses the **Web Serial API**, which is supported in **Chrome, Edge and other Chromium-based browsers**. **Firefox and Safari do not implement Web Serial** and cannot connect to the board.

How to connect

The configurator is at **https://firstlayerrobotics.com/motion-core/configurator**.

Plug your Motion Core into a USB port, then click the connect button on the page. Your browser will show a picker; choose the entry labelled **FLR Motion Core** (also shown alongside a `usbmodemXXXXXX` or `usbserialXXXXXX` device path).

Only one program can hold the serial port at a time. Close the configurator before running the tuning tool in §6, or any of the software in §8 onward.

Two things the configurator does *not* cover: **velocity-loop tuning** (§6, which uses a host-side tool) and **firmware updates**.

The rest of this section gives the equivalent serial commands. They are the authoritative reference, they are what the configurator sends underneath, and they are what you need for scripted setup — so they are worth reading even if you never type one.

Command syntax

Read a value, write a value, write all values to flash:

```
GET kin.track_width
SET kin.track_width=0.28
SAVE
```

> **NOTE**
> `SET takes key=value with no spaces around the =.` `SET kin.track_width 0.28` is rejected. This trips up nearly everyone once.

Changes take effect immediately but live only in RAM until you `SAVE`. That is deliberate: you can experiment freely, and a power cycle undoes anything you have not committed.

Step 0 — snapshot what you started with

```
GET ALL
```

Copy the output somewhere. It is one long line containing all 54 keys, and it is your undo button.

Step 1 — prove the e-stop works

Before any wheel turns. Press the e-stop, then:

```
STATUS
```

You should see `estop=1` and `fault=ESTOP`. Release it, send `ENABLE ON`, and confirm the fault clears. **If this does not behave exactly as described, stop and fix the wiring** — everything downstream assumes you can stop the robot.

Step 2 — get the directions right

Put the robot on blocks so the wheels spin free. Command both wheels forward gently:

```
WHEEL l=0.2 r=0.2
```

Fix the two possible sign errors **in this order** — motor direction first, then encoder direction. Reversing them in the wrong order will chase its own tail.

First, motor direction. Does each wheel physically drive the robot forward? For any wheel that spins backwards, flip its motor invert:

```
SET motor.invert_left=1
SET motor.invert_right=0
```

Defaults are `motor.invert_left=0`, `motor.invert_right=1` — asymmetric because the two motors face opposite ways on a typical chassis.

Then, encoder direction. With both wheels now physically driving forward, check that the reported velocities are positive:

```
STATUS
```

Look at `vl` and `vr`. A wheel moving forward must report a **positive** velocity. If one is negative, flip its encoder invert:

```
SET encoder.invert_left=1
SET encoder.invert_right=0
```

Defaults are `encoder.invert_left=0`, `encoder.invert_right=1`.

A wheel that drives forward but reports negative velocity creates a positive feedback loop in closed-loop control — the robot will run away. This step is not optional.

Step 3 — teach it your scale

Three numbers convert between encoder counts and real-world motion. Set them in this order, because each measurement depends on the one before it.

`kin.counts_per_rev` — encoder counts per wheel revolution, after gearing.

```
counts_per_rev = gearbox_ratio × encoder_CPR × 4
```

The ×4 is quadrature decoding. The factory default of **2249** corresponds to the reference platform: a 46.85:1 gearmotor with a 48-count encoder (46.85 × 48 = 2249). For a 30:1 gearbox with a 12-count encoder you would use 30 × 12 × 4 = 1440.

```
SET kin.counts_per_rev=1440
```

If your platform has **no encoders**, set this to `0`. The board switches to open-loop and velocity commands return `NO_FEEDBACK` — you can still drive in duty mode, but nothing in §6 applies.

`kin.wheel_radius` — verify by driving, not by measuring the wheel. Command the robot to travel 1.0 m in a straight line, then measure what it actually did. If it travelled 0.92 m, your radius is 8 % too small:

```
SET kin.wheel_radius=0.053
```

Repeat until commanded and actual agree. This catches tyre compression and counts-per-rev errors together.

`kin.track_width` — the distance between the two wheel contact patches. Verify by rotation: command a full 360° spin in place and measure the heading error. Overshooting means the configured track width is too small.

```
SET kin.track_width=0.28
```

Get these three right and everything downstream — odometry, `cmd_vel`, teleop speeds — is correct. Get them wrong and every layer above inherits the error.

Step 4 — set your motion envelope

```
SET kin.max_v=1.0          # max linear speed, m/s
SET kin.max_w=3.0          # max angular speed, rad/s
SET kin.max_accel=25.0     # acceleration limit, rad/s² (0 disables)
SET motor.max_rads=20.9    # max motor speed, rad/s — from your motor's rating
```

`kin.max_v` and `kin.max_w` are the envelope the board clamps `cmd_vel` to. Set them to what your robot can actually do, not what you hope for.

Step 5 — set your motor limits

```
SET motor.v_max=12.0          # nominal motor voltage
SET motor.i_max=4.0           # rated continuous current per side, A
SET motor.i_fault=5.0         # absolute hard trip per side, A
SET motor.i_max_time_ms=500   # how long i_max may be exceeded before tripping
```

Current protection is two-tier. Exceeding `motor.i_max` for longer than `motor.i_max_time_ms` trips an overcurrent fault — that is the thermal protection. Exceeding `motor.i_fault` trips immediately — that is the "something is badly wrong" limit. Set `i_max` from your motor's continuous rating, and keep `i_fault` above it but within the board's 4A per-side continuous rating.

Step 6 — set your battery

Defaults assume a 3S Li-ion/LiPo pack. Set four values to match your chemistry:

```
SET batt.full_v=12.6       # 100% reference
SET batt.low_v=10.2        # warning threshold
SET batt.critical_v=9.6    # LOWBATT fault — motors stop
SET batt.hyst_v=0.2        # hysteresis, stops threshold chatter
```

`batt.critical_v` is a real cutoff: crossing it stops the motors and latches `LOWBATT`, and the board will refuse `ENABLE ON` until the voltage recovers. Set it above the point where your pack is damaged by over-discharge.

Do **not** touch `batt.divider_ratio` — that is factory calibration of the voltage sensing itself, not a property of your battery.

What you should leave alone

Beyond the factory calibration group, these are advanced or bench-only and are correctly set to their defaults for normal use:

- `current_loop.enable`, `pid.kp_i`, `pid.ki_i` — the inner current cascade
- `ff.static`, `ff.visc`, `ff.inertia`, `pid.kp_c`, `pid.ki_c` — cascade feed-forward
- `encoder.obs_enable`, `encoder.obs_alpha`, `encoder.obs_beta` — velocity observer
- `motor.stall_duty`, `motor.stall_vel`, `motor.stall_ms` — stall detection thresholds

The cascade is **disabled by default and is not yet validated for production use**. Enabling it with zero gains raises a `CASCADE_NOGAIN` fault that blocks all motion. Leave `current_loop.enable=0`.

A complete per-parameter reference, including valid ranges, is in `CONFIG_PARAMETERS.md`.

6. TUNING THE VELOCITY LOOP

With geometry and directions correct, tune the velocity controller to your motors. The board ships with `pid.kp=0.5`, `pid.ki=7.0`, `pid.kd=0.0`, tuned for the reference platform — a good starting point, rarely optimal for yours.

An automated tuner is provided:

```
python3 tools/mc1_optimise.py <PORT> --scenario autotune-all
```

This sweeps `pid.kp`, then `pid.ki`, and **saves both** to flash. It is the customer-facing tuning path. Expect it to drive the wheels through a range of speeds for a minute or two.

Useful diagnostic scenarios, which change nothing:

SCENARIO	WHAT IT DOES
open	Open-loop duty ramp. Exposes cogging, deadband, and supply sag
closed	Closed-loop velocity ramp. Exposes oscillation and tracking error
all	Both, in sequence (the default)

The tuner temporarily raises the SLOW duty limit so it can reach test speeds, and restores it when it exits.

> **NOTE**
> **Bench-only scenarios.** `autotune-inner` and `autotune-vehicle` tune the current cascade, which is not validated for production. Do not run them on a customer platform.

Judge the result by ear and by eye. A well-tuned loop holds commanded speed without audible hunting and without visible oscillation when you push against the wheel. If the tuner leaves the robot buzzing or twitching, reduce `pid.ki` and re-test.

7. SAVE IT AND PROVE IT

Nothing so far survives a power cycle. Commit it:

```
SAVE
```

Then **prove it stuck** — power the board down completely, bring it back up, and verify:

```
GET ALL
```

Compare against what you configured. This step catches the single most common bring-up mistake, which is a session of careful configuration that was never saved.

Three commands manage persisted configuration:

COMMAND	EFFECT
SAVE	Write current settings to flash
LOAD	Discard RAM changes, reload from flash
DEFAULTS	Reset robot configuration to factory values — preserves calibration

Your board is now configured. Everything below is software.

8. INSTALLING THE PYTHON LIBRARY

```
pip install orcp
```

Verify:

```
python -c "from orcp import ORCP; print('orcp installed OK')"
```

Requires Python 3.9+. The only runtime dependency is `pyserial`.

For the keyboard teleop application on **Windows**, install the extra that pulls in curses support:

```
pip install "orcp[teleop]"
```

On macOS and Linux the base install is sufficient — the extra is a no-op.

9. DRIVING WITH THE TELEOP APPLICATION

Before writing any code, prove out the whole platform end-to-end with the keyboard driving application that ships with the library. If teleop can drive your robot, everything from the battery to the encoders to the ORCP link is working — and any problem you hit next is in your code, not your hardware.

Point it at your port:

```
orcp-drive /dev/cu.usbmodem1234
```

It works equally against the simulator:

```
orcp-drive /tmp/orcp
```

It takes **one argument — the port — and no flags.** Baud is always 115200, and it always starts in SLOW.

Controls

KEY	ACTION
W / ↑	Forward
S / ↓	Reverse
A / ←	Spin left
D / →	Spin right
Q	Arc forward-left
E	Arc forward-right
Space	Stop
+ / =	Speed up one step
− / _	Slow down one step
M	Toggle SLOW ↔ NORMAL
R	Re-enable after a fault
Esc	Quit

What you see

The display shows the current mode and duty limit, the selected speed step, the live `v` and `w` being commanded, battery voltage, and either `Status: OK` or a highlighted fault banner.

Speed steps

Six steps in each mode, selected with **+** and **−**:

- **SLOW** — 0.05 to 0.30 m/s (*Crawl, Slow, Steady, Brisk, Fast, Max*)
- **NORMAL** — 0.10 to 1.00 m/s (*Gentle, Easy, Cruise, Quick, Fast, Full*)

Spin rate is 2.0 rad/s in SLOW and 4.0 rad/s in NORMAL. The **Q**/**E** arcs follow a fixed 0.40 m turning radius, so the curve keeps the same shape at any speed rather than pivoting on one wheel when you are going slowly.

Switching to NORMAL

Pressing **M** does everything the mode change requires: stops the robot, switches preset, enables the motors, and starts the heartbeat. Pressing it again reverses all of that. You do not need to manage enable or heartbeat yourself.

Recovering from a fault

When the board faults, the robot stops and the banner tells you why. Clear the physical cause if there is one — release the e-stop, free the jammed wheel — then press **R**.

If **R** is rejected, the fault is not yet clearable; the reason appears in the message line. See §11.

On exit

Esc always leaves the robot stopped, the heartbeat stopped, and the board back in SLOW — even if you quit mid-motion. The same cleanup runs if the application crashes.

10. YOUR FIRST MOTION IN CODE

Teleop is fine for driving the robot around, but the point of Motion Core is to build your own robotic behaviours in code. This section walks the same commands through the Python library.

Connecting

```
from orcp import ORCP

robot = ORCP('/dev/cu.usbmodem1234')
```

Two things to know:

- **There is no `connect()` method.** The constructor opens the port.
- **Opening a USB port takes about two seconds.** The library waits out the microcontroller's reset-on-connect, then flushes stale input. This is normal; do not treat it as a hang.

The address argument accepts more than a device path:

FORM	USE
/dev/cu.usbmodem1234	USB on macOS
/dev/ttyACM0	USB on Linux
COM3	USB on Windows
/tmp/orcp	The simulator — see Appendix A
socket://192.168.4.1:3333	A network-bridged board

Prefer the context manager, which guarantees the robot is stopped and the port closed even if your code raises:

```
with ORCP('/dev/cu.usbmodem1234') as robot:
    print(robot.info())
```

Check in

```
from orcp import ORCP

with ORCP('/dev/cu.usbmodem1234') as robot:
    assert robot.ping()

    info = robot.info()
    print(f"firmware {info.fw}, protocol {info.proto}, level {info.level}")

    status = robot.status()
    print(f"battery {status.vbat} V, fault {status.fault}")
```

Drive, in SLOW

The board boots in SLOW with motors already enabled, so you can move immediately:

```
import time
```

```
from orcp import ORCP

with ORCP('/dev/cu.usbmodem1234') as robot:
    robot.cmd_vel(0.2, 0.0)      # 0.2 m/s forward
    time.sleep(2)

    robot.cmd_vel(0.0, 0.5)      # spin left at 0.5 rad/s
    time.sleep(1)

    robot.stop()
```

`cmd_vel(v, w)` takes linear velocity in m/s and angular velocity in rad/s, and relies on the kinematics you configured in §5.

To address wheels individually — useful for diagnostics — use `wheel()`, which takes rad/s per side:

```
robot.wheel(2.0, 2.0)            # both wheels forward
robot.wheel(2.0, -2.0)           # spin in place
```

`stop()` is deliberately forgiving: it never raises. You can always call it in a cleanup path without wrapping it.

Going faster — NORMAL mode

NORMAL raises the duty limit to 90 %, but requires you to enable the motors and to keep sending a heartbeat:

```
import time
from orcp import ORCP

with ORCP('/dev/cu.usbmodem1234') as robot:
    robot.preset('NORMAL')
    robot.enable()
    robot.start_heartbeat(interval=0.08)   # background thread, 80 ms

    try:
        robot.cmd_vel(0.5, 0.0)
        time.sleep(3)
    finally:
        robot.stop()
        robot.stop_heartbeat()
        robot.disable()
        robot.preset('SLOW')
```

`start_heartbeat()` runs a daemon thread that sends `HB` on your behalf. Pick an interval comfortably under the board's `hb.timeout_ms` (500 ms by default); 80 ms gives generous margin. **Stop the heartbeat before you stop caring about the robot** — a heartbeat thread outliving your control logic defeats the purpose of a dead-man switch.

Note that NORMAL also enforces a 250 ms command timeout. Sending `cmd_vel` once and sleeping for three seconds, as above, works because the library resends — but if you write your own loop, keep commands flowing.

Reacting to faults

Register a callback and the board will tell you the moment something goes wrong, without polling:

```
def on_fault(event):
    print(f"FAULT: {event.code}")

def on_warn(event):
    print(f"warning: {event.type}")

robot.on_fault(on_fault)
robot.on_warn(on_warn)
```

When a fault arrives, the motors have **already stopped** — the board acts first and reports second. By the time your callback runs, `robot.fault` is already updated, so you do not need to call `status()` to find out what happened.

Keep callbacks short. They run on the library's reader thread, and blocking there delays every subsequent message. Exceptions raised inside a callback are swallowed, so a broken callback fails silently — log rather than raise.

Recover the same way the board expects:

```
robot.enable()      # ENABLE ON — clears the latched fault
```

Live telemetry

```
def on_data(d):
    print(f"vl={d.vl:.2f} vr={d.vr:.2f} vbat={d.vbat:.1f}")

robot.stream_on(rate=10, callback=on_data)
# ... drive ...
robot.stream_off()
```

Reading and writing configuration

Everything from §5 is available programmatically:

```
print(robot.get('kin.wheel_radius'))

robot.set('pid.kp', 0.4)
robot.save()

all_params = robot.get_all()        # dict of all 54 keys
```

`robot.defaults()` performs the calibration-preserving reset described in §5.

Handling errors

```
from orcp import ORCP, CommandError, TimeoutError, ConnectionError

try:
    with ORCP('/dev/cu.usbmodem1234') as robot:
        robot.cmd_vel(0.2, 0.0)
```

```
except CommandError as e:
    print(f"board rejected the command: {e.code}")
except TimeoutError:
    print("no response — check the cable and the port")
except ConnectionError:
    print("could not open the port")
```

`CommandError.code` carries the board's reason — `BAD_ARG`, `NO_FEEDBACK`, `NOT_ENABLED` and so on.

11. FAULT REFERENCE

Every fault latches. `ENABLE ON` — the **R** key in teleop, `robot.enable()` in code — is the universal recovery unless noted otherwise.

CODE	MEANING	WHAT TO DO
OK	No fault	—
TIMEOUT	No motion command within the preset's timeout window	Resume sending commands, then ENABLE ON
HEARTBEAT	No HB within hb.timeout_ms	Start or fix the heartbeat, then ENABLE ON
NOT_ENABLED	Motion commanded while disabled	ENABLE ON
ESTOP	E-stop loop open — button pressed, wiring broken, or J3 has no switch and no link fitted	Release the e-stop and confirm wiring, then ENABLE ON. Motor supply is physically disconnected until the loop closes
LOWBATT	Pack fell below batt.critical_v	Charge or replace the pack. The board refuses ENABLE ON until voltage recovers
CHARGER	Charger-detect loop open — charger connected, or J2 has no switch and no link fitted	Disconnect the charger, or fit a link across J2 pins 1–2. Then ENABLE ON. Drive is inhibited in hardware until the loop closes
MOTOR_LEFT / MOTOR_RIGHT / MOTOR_BOTH	Motor driver reported a hardware fault	Check motor wiring for shorts; look for overheating. Then ENABLE ON
OVERCURRENT_LEFT / _RIGHT / _BOTH	Current exceeded motor.i_max for longer than motor.i_max_time_ms, or exceeded motor.i_fault outright	Find the mechanical cause. Verify motor.i_max matches your motor's rating. Then ENABLE ON
ENCODER_STALL	Duty is being applied but the wheel is not turning	Check for a jam, a disconnected encoder, or broken encoder wiring. Then ENABLE ON
CASCADE_NOGAIN	current_loop.enable=1 with both inner gains at zero	SET current_loop.enable=0 and SAVE. See §5
BOOTLOADER_MISMATCH	Firmware requires a newer bootloader than the one installed	Motion is blocked until the bootloader is updated. Contact support

`ENCODER_STALL` firing when you bump a wall is **correct behaviour**, not a bug — it is motor protection. If it trips too eagerly on your platform, the thresholds are tunable via `motor.stall_duty`, `motor.stall_vel` and `motor.stall_ms`.

Warnings

Warnings do not stop the robot. They arrive as `! WARN` push events and reach your `on_warn` callback.

WARNING	MEANING
BATT	Pack fell below batt.low_v — still running, but plan to charge
AUX5V	The auxiliary 5V rail exceeded aux5v.i_warn or dropped below aux5v.v_warn

12. TROUBLESHOOTING

The robot does nothing — no motion at all, and no fault that explains it.

Check the two hardware interlocks first (§2). If J3 (e-stop) or J2 (charger-detect) is open — no switch and no jumper link — the motor outputs are disabled in hardware and no amount of software will move the robot. `STATUS` will show `fault=ESTOP` or `fault=CHARGER`.

No serial port appears.

Check the battery first — USB does not power the board, so an unpowered Motion Core never enumerates no matter how good the cable is. Try another USB-C cable — charge-only cables are the usual culprit. Confirm the board is powered and the red LED is blinking. On Linux, check `dmesg | tail` immediately after plugging in.

The port exists but PING returns nothing.

On macOS, confirm you are using `/dev/cu.*` and not `/dev/tty.*`. Check the baud rate is 115200. Close any other program holding the port — only one process can own it.

Permission denied on Linux.

Add yourself to `dialout` and log out and back in (see §3).

The green LED is blinking at 5 Hz.

The board is in its bootloader. Usually this means an interrupted firmware update, or the board failed to start the application three times running and fell back deliberately. Power-cycle first. If it persists, the application firmware needs reflashing — contact support.

Commands are rejected with NO_FEEDBACK.

`kin.counts_per_rev` is `0`, so the board has no encoder feedback and cannot do closed-loop velocity control. Set it (§5, step 3), or drive in duty mode.

The robot runs away when I command a small speed.

An encoder is reading backwards. Go back to §5 step 2 and verify each wheel reports **positive** velocity when driving forward. This is the classic symptom of an inverted encoder in a closed loop.

The robot drives, but distances and turns are wrong.

Your kinematics are off. Re-do §5 step 3 in order: `counts_per_rev`, then `wheel_radius` by driving a measured straight line, then `track_width` by spinning 360°.

The robot buzzes, hunts, or oscillates in place.

The velocity loop is over-tuned. Reduce `pid.ki` and re-test, or re-run `--scenario autotune-all` (§6).

It faults with TIMEOUT or HEARTBEAT as soon as I switch to NORMAL.

NORMAL requires both a motion command every 250 ms and a heartbeat every 500 ms. Call `start_heartbeat()` and keep commands flowing (§10).

My settings vanished after a power cycle.

You did not `SAVE`. Configuration lives in RAM until committed (§7).

Everything is misconfigured and I want to start over.

`DEFAULTS` resets robot configuration while preserving factory calibration. Follow with `SAVE`.

WHERE TO GO NEXT

- `CONFIG_PARAMETERS.md` — complete per-parameter reference with valid ranges
- `BATTERY_PROFILES.md` — supported battery chemistries and their threshold values
- *Motion Core — Getting Started (ROS 2)* — if you want to drive this platform from ROS 2

APPENDIX A — TRYING IT WITHOUT HARDWARE

You do not need a board to learn the workflow. The ORCP reference simulator presents a virtual Motion Core on a serial device, and everything in sections 8–10 works against it unchanged.

```
pip install orcp orcp-sim
orcp-sim --profile mc1 --link /tmp/orcp
```

It prints the device it created:

```
ORCP Reference Simulator — proto ORCP/1.1, fw 1.8.0, Level 2
Serial device: /dev/ttys004
```

Leave it running and use `/tmp/orcp` anywhere this guide says "your port". Every later example — including the teleop application — accepts it:

```
orcp-drive /tmp/orcp
```

Two limitations to know about:

- **The simulator requires macOS or Linux.** It uses a pseudo-terminal, which Windows does not provide. Windows users need hardware, or WSL.
- **The bundled mc1 profile models firmware 1.8.0 with 46 configuration keys**, while current firmware is 1.9.6 with 54. The command surface and the workflow are identical; a handful of the newer keys in section 5 are absent. Treat the simulator as a way to learn the flow, not as a configuration rehearsal.

The symlink takes a moment to appear after launch. If a connection attempt fails immediately, wait a second and retry.