openbricks documentation

Release 2.7.1

the openbricks contributors

Aug 17, 2026

GETTING STARTED

1 Installation 2

2 Hardware guide 4

3 Command-line tool 11

4 Simulator 18

5 Examples 19

6 Measuring wheel diameter & axle track 30

7 API reference 33

8 Architecture 77

9 Building the firmware 82

Python Module Index 86

Index 87

openbricks is a Pybricks-style MicroPython firmware for **open hardware** — commodity MCUs, commodity motors, commodity sensors.

Prefer offline reading? This documentation is also available as a single **PDF download**, *rebuilt on every deploy.*

Like Pybricks, openbricks is a *firmware you flash to an MCU*, not a library you `pip install` on top of stock MicroPython. The firmware owns the runtime: the 1 kHz motor scheduler, trajectory planner, state observer, and drivebase controller run as native C code inside the image, and the three-layer Python API (drivers → interfaces → robotics) is what `import openbricks` gives you out of the box.

```
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

left = ST3032Motor(servo_id=1, tx=14, rx=6)
right = ST3032Motor(servo_id=2, tx=14, rx=6, invert=True)

db = DriveBase(left, right, wheel_diameter_mm=56, axle_track_mm=114)
db.straight(500)    # mm
db.turn(90)         # degrees
```

- **Firmware images**: ESP32-S3 (primary) and classic ESP32, prebuilt on every release.
- **Host tooling**: `pipx install openbricks` gets you the *openbricks CLI* (flash / run / upload / stop / log over BLE) and, with the `[sim]` extra, a MuJoCo-backed *simulator*.
- **Source**: github.com/1e0ng/openbricks, MIT licensed.

CHAPTER

ONE

INSTALLATION

Two things get installed: the **host tooling** on your computer (a CLI that flashes firmware and talks to hubs over BLE) and the **firmware** on the hub itself.

1.1 1. Install the host tooling

```
$ pipx install 'openbricks[sim]'   # CLI + MuJoCo simulator
```

or, lighter:

```
$ pipx install openbricks          # CLI only (flash / run / log)
```

`pip install` works too; `pipx` is recommended on modern macOS / Linux to avoid the "externally managed environment" error. The package is on PyPI: https://pypi.org/project/openbricks/.

The `[sim]` extra adds `mujoco` (~50 MB, native OpenGL) and `numpy`. If you only want to flash and run code on real hardware, skip it — `openbricks sim …` will print an install hint instead of crashing.

1.2 2. Get a firmware image

Grab a prebuilt image from the Releases page:

- `openbricks-esp32s3-firmware-<version>.bin` — ESP32-S3 boards (primary target)
- `openbricks-esp32-firmware-<version>.bin` — classic ESP32 boards

(You can also *build the firmware from source*.)

1.3 3. Flash the hub

Connect the board over USB, then flash and name the hub in one step:

```
$ openbricks flash --name RobotA
```

That's the whole command: the serial port is auto-detected (when exactly one ESP device is connected), the chip type is probed, and the newest matching firmware release is downloaded automatically (cached under `~/.cache/openbricks/firmware`).

- `--name` is the BLE advertising identifier you'll use later with `openbricks run -n …`; pick a unique one per hub.

- With several serial devices connected, pass `--port` explicitly (`/dev/ttyUSB0` on Linux, `/dev/cu.usbserial-*` on macOS, `COM5` on Windows).
- To flash a specific/downloaded image instead of the newest release, pass `--firmware path/to/firmware.bin`.

Skip this step entirely if you only want to run code in the *simulator*.

1.4 4. Run your first program

```
$ openbricks list                    # find your hub over BLE
$ openbricks run -n RobotA main.py # push a script and stream its output
```

See *the CLI reference* for every command, and *Examples* for programs to start from.

1.5 Troubleshooting

- **Hub doesn't show up in `openbricks list`** — BLE may be toggled off. Short-press the Bluetooth button (GPIO 38 on the S3, GPIO 5 on the classic ESP32); on the ESP32-S3 the onboard LED turns blue when BLE is on, yellow when off. While a program is running the LED flashes that colour at 2 Hz instead of holding it solid — a flashing LED means "robot running", not a different BLE state. A brief flash is the press acknowledgment, shown the moment a program-button press is recognized: **red** for the press that starts a run, **green** for the press that stops one. See *openbricks.hub.ESP32S3DevkitHub*.
- **Serial port permission errors on Linux** — add yourself to the `dialout` group (`sudo usermod -aG dialout $USER`) and re-login.
- **Flash succeeds but the program doesn't start** — check the run log with `openbricks log -n RobotA` for the traceback of the last run.

CHAPTER

TWO

HARDWARE GUIDE

A starter parts list and wiring notes. Everything here is commodity stuff you can buy on AliExpress / Amazon / Adafruit.

2.1 Recommended starter robot

Part	Qty	Notes
ESP32-S3 DevKitC-1	1	Any ESP32-S3 board with ≥10 free GPIOs works; classic ESP32 also supported
Feetech STS3032 serial bus servo (12V)	2	Drive wheels, continuous-rotation wheel mode. 10 kg·cm, 148 RPM no-load — datasheet in `docs/datasheets/`
URT-2 serial bus adapter	1	One half-duplex bus daisy-chains every servo
3S LiPo battery (11.1V nominal) + balance charger	1	Feeds the servo rail directly. ST-3032 brown-out floor is ~9V, so a sagging 2S (7.4V) is not enough
Buck converter (12V → 5V, ≥1A)	1	Powers the ESP32-S3 and sensors
TCS34725 breakout	2–4	Colour sensor array for line following / zone detection
TCA9548A I2C multiplexer breakout	1	Required for more than one TCS34725 — its address is fixed at `0x29`
ICM-45686 breakout	1	SPI IMU, read inside the 1 kHz control tick — the heading source for `use_gyro`
ST-3215 serial bus servo	0–4	Optional; good for arms / grippers. Same bus and protocol, but do **not** share a 6V rail setup — see servo notes
Jumper wires, M3 standoffs, chassis plate	—	Your robot, your build

A DC-gear-motor build (JGB37-520 / MG370 + H-bridge) is still fully supported — see *Alternative: DC gear motors with encoders* at the bottom.

2.2 Power budget

Two rails, one battery:

```
   [ 3S LiPo 11.1V ]
        │
        ├──────────────►  URT-2 servo rail   (all ST-3032 / ST-3215 power)
        │
```

(continues on next page)

(continued from previous page)

```
        └──► Buck  ──►  5V rail
                        │
                        ├──►  ESP32-S3 VIN (5V pin)
                        └──►  Sensors via 3.3V regulator on the board
```

- Never power the servos from the ESP32's 5V pin or USB — a single ST-3032 stall pulls more than a dev board can source.
- The ST-3032's brown-out floor is ~9V. A 3S pack sags to ~9.9V near empty, which still clears it; a 2S pack does not.
- Tie all grounds together: battery, buck, URT-2, ESP32, sensor breakouts. This sounds obvious but it's the #1 reason new builds misbehave.

2.3 GPIO map (ESP32-S3)

The serial-bus build needs very few pins — that's most of its charm:

Function	GPIO(	Devices on this line
Analog sensors	1–10	The FULL ADC1 bank — the *openbricks.drivers.qtr.QTRLineSensor* window (wiring below). Buttons and UARTs deliberately live elsewhere so all ten stay analog-capable
I2C0 (SDA, SCL)	15, 16	TCA9548A mux (0x70) + colour sensors behind it, shared bus (an I2C BNO055 IMU at 0x28 fits here too)
UART1 (TX, RX)	14, 41	URT-2 serial bus — every ST-3032 / ST-3215 daisy-chained (RX was GPIO 6 until 1.71.0; it moved so the analog bank stays whole)
Program button	39	Start/stop, polled with an internal pull-up (was GPIO 4 until 1.71.0)
BLE-toggle button	38	Bluetooth on/off (was GPIO 5 until 1.66.3)
WS2812 data	21	Addressable RGB LED strip / ×8 stick DIN (*openbricks.drivers.ws2812*) — on boards that break out the header corner next to 5 V/GND, this is the free pin beside the reserved 19/20 USB pair
SPI (SCK, MOSI, MISO, CS)	12, 13, 11, 17	ICM-45686 IMU breakout (*openbricks.drivers.icm45686.ICM45686*) — 3V3 + GND + these four; INT and the other breakout pins stay unwired (the 1 kHz hard tick polls)

Pin gotchas on the ESP32-S3:

- **GPIO 22–25 don't exist** — the pin list is 0–21 then 26–48.
- **GPIO 26–32 (and 33–37 on octal-PSRAM modules) are flash/PSRAM.** Do not use them.
- **GPIO 19/20 are the native-USB D-/D+** and **0/3/45/46 are strapping pins.**
- I2C and UART route through the GPIO matrix, so none of these assignments are fixed — they're the convention the bundled examples use. On a **classic ESP32**, the usual equivalents are I2C on 21/22 and any two free pins for the UART.

The firmware enforces the hard cases at construction time: a driver asked to wire a nonexistent, flash, or USB pin — or a pin the runtime already owns, like the program button — raises *openbricks.pins.*

ReservedPinError naming the pin, the role, and the reason, instead of failing somewhere far from the mistake. See *openbricks.pins*.

2.3.1 QTRLineSensor wiring (the standard line-follow window)

openbricks.drivers.qtr.QTRLineSensor bakes the whole rig geometry into the firmware — pins, element positions, and both mode setpoints — so programs just construct it and pick a discipline:

```
from openbricks.drivers.qtr import QTRLineSensor
qtr = QTRLineSensor()
qtr.set_mode("left")            # or "right"; switchable mid-run
error = qtr.read().edge_error()
```

Ten channels of a QTRX-HD-15A (4 mm pitch) in a skip pattern, left to right as mounted, onto GPIO 1..10 **in order**:

QTR channel	1	3	4	5	7	9	11	12	13	15
GPIO	1	2	3	4	5	6	7	8	9	10
x (mm)	−28	−20	−16	−12	−4	+4	+12	+16	+20	+28

That spans a 56 mm window at spacings 8/4/4/8/8/8/4/4/8 mm (the driver's `positions_mm` carries the true coordinates, so edge interpolation is exact across the unequal gaps). The two modes:

- **`"left"`** — holds the line's LEFT edge under **channel 4** (x = −16 mm)
- **`"right"`** — holds the line's RIGHT edge under **channel 12** (x = +16 mm)

`edge_error()` is how far the mode's channel sits from the black/white boundary: that element's ambient (0 black .. 100 white) referenced to 50, so it reads 0 exactly when the channel straddles the edge, and is signed so positive steers right in both modes.

The skip pattern is a palindrome, so if the board is mounted the other way round, only these channel labels swap — GPIO order and geometry stay identical.

Either way the ~20 mm line sits inside the middle of the window with ≥3 channels of mat visible on the far side — those far-side elements are the branch watch in the bundled followers, and the whole window going dark is the intersection/ending signal.

One board-level note: **GPIO 3 (= channel 4, the left-mode setpoint channel) is a strapping pin**, and on the ESP32-S3-COREBOARD V1.4 it optionally carries a 10 kΩ pull-up through the `USB-JTAG` 0 Ω link. Harmless (per-element calibration absorbs the bias), but if `qtr_calibrate.py` shows element `[2]` with a conspicuously narrower span than its neighbours, that link is populated — desoldering it is safe if you never use pin-JTAG.

The QTRX board's CTRL (emitter enable) can stay tied high; VCC to 3V3, GND to GND.

2.3.2 If the status LED never lights

The onboard WS2812 speaks a write-only protocol — the firmware can't tell a dead LED from a working one, so a blown pixel looks like software silently failing. Run `examples/led_probe.py` on the hub: it drives full white at both timings across every plausible RGB pin (48/38/47/21) plus a raw `machine.bitstream` write. If nothing lights through all of that, the problem is physical. The decisive check is a **power-off ohmmeter reading from the LED's data pin (or its GPIO) to GND**: a WS2812 whose DI input has failed as an internal short reads ~0 Ω there and clamps the GPIO — the pin measures

millivolts even when driven high, every write "succeeds" into the short, and the symptom history is "worked once, then went dark for good". Lift the data link to free the pin. Also check the schematic for **population options in the LED circuit**. The ESP32-S3-COREBOARD V1.4 (`docs/datasheets/esp32s3_coreboard_v1.4_schematic.pdf`) routes GPIO 48 to the WS2812 through a 0 Ω `RGB` link and feeds the LED's VDD from the **5 V rail** through a second optional link — if either link is unpopulated (or the board is powered via 3.3 V only, leaving the 5 V rail dead), the LED never lights while every write succeeds. Bridge the link / feed 5 V, or wire any external WS2812 to 3.3V / GND / GPIO 48 — it becomes the status LED with no firmware change.

2.4 Sensor wiring (I2C)

The baseline build has **several colour sensors** (a line-follower / zone-detection array) on this bus — the IMU is the SPI-wired ICM-45686 (see the GPIO map above) and doesn't share it. The TCS34725's address is fixed at `0x29` — no address-select pins — so two of them collide on a bare bus. The TCA9548A multiplexer solves this: it sits at `0x70` and fans the bus out to eight isolated channels, one colour sensor per channel. (If you use the I2C BNO055 IMU instead, it has its own address, `0x28`, and connects straight to the bus.)

Wire the mux to the ESP32-S3, then each sensor to a mux channel:

TCA9548A pin	Connect to	Notes
VIN	3.3V	
GND	GND	Common ground with everything else
SDA / SCL	GPIO 15 / 16	The main bus
SD0/SC0 … SD7/SC7	one TCS34725 each	Isolated channels; every sensor sits at its own `0x29`

TCS34725 / BNO055 pin	Connect to	Notes
VIN (or VCC)	3.3V	The breakouts have onboard regulators, but the board's 3.3V is cleanest
GND	GND	
SDA / SCL	mux channel `SDn`/`SCn` (colour sensors); GPIO 15/16 directly (BNO055, if used)	

The Adafruit breakouts include ~10 kΩ SDA/SCL pull-ups, so for a handful of devices you don't need to add your own. In code, `mux[n]` behaves like an I2C bus, so the sensor driver is constructed exactly as it would be on a bare bus:

```
from machine import I2C, Pin
from openbricks.drivers.tca9548a import TCA9548A
from openbricks.drivers.tcs34725 import TCS34725

i2c = I2C(0, sda=Pin(15), scl=Pin(16), freq=400_000)    # ESP32-S3
mux = TCA9548A(i2c)                                     # 0x70 by default
sensors = [TCS34725(mux[ch]) for ch in range(3)]        # left, mid, right
```

For a complete program — a 2-sensor array that combines each sensor's `ambient()` and `rgb()` readings to name the colour under it (red / blue / green / yellow / white / black) — see `examples/color_array.py`.

Simplifications when you need fewer parts:

- **One colour sensor**: skip the mux entirely; wire the TCS34725 straight to GPIO 15/16 (and an I2C BNO055, if you use one, shares the same pins — `0x29` and `0x28` coexist fine).
- **Exactly two colour sensors**: the ESP32's second hardware I2C controller also works (`I2C(1, sda=..., scl=...)` on any two free pins), one sensor per bus — no mux.

2.4.1 TCS34725 LED pin

The colour sensor breakout has two extra pins beyond power and I2C:

- **LED** — drives the onboard white illumination LED. On Adafruit boards it defaults **on** (tied to VIN through the ADC-enable trace). To control it, wire it to a spare GPIO and drive high/low; to force it **off**, tie LED to GND. Leave it on for consistent colour readings — ambient light alone is unreliable across environments.
- **INT** — the interrupt output. The driver polls, so leave INT **unconnected**.

2.5 Serial bus servo notes (ST-3032 / ST-3215)

- **Every servo needs a unique bus ID.** Factory default is 1; re-ID one servo at a time with `examples/st3215_reid.py` (same protocol — it works for the ST-3032 too). The bundled drivebase examples assume left = 1, right = 2.
- **Speed limits.** Per the STS3032 datasheet (`docs/datasheets/`), no-load top speed at 12V is 148 RPM = 888 °/s; under the rated 3.3 kg·cm load it drops to roughly ⅔ of that. The driver's default `max_dps=600` clamps requests at the loaded operating point — construct with an explicit `max_dps=900` to chase the no-load number.
- **Mixed fleets:** the ST-3215 tolerates lower voltages, but the ST-3032 browns out below ~9V. If you daisy-chain both on one URT-2, the rail must satisfy the strictest member: 12V.
- **Drivebase:** `ST3032Motor` drops straight into `DriveBase`:

```
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

left  = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=6)
right = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=6, invert=True)
db = DriveBase(left, right, wheel_diameter_mm=65, axle_track_mm=120)
```

Bench-test a fresh build with `examples/st3032_drivebase_test.py`.

2.6 Calibrating the drivebase

`wheel_diameter_mm` and `axle_track_mm` are the two physical parameters that matter for straight-line distance and turn accuracy. Measure them with calipers or a ruler (wheel contact patch to wheel contact patch for axle track, not hub to hub). If `straight(1000)` undershoots, your wheel diameter value is too large; if `turn(360)` overshoots, your axle track is too small.

High-torque 12V servos also deliver the default launch profile much more stiffly than small DC motors — if the chassis pitches or lifts its rear when a move starts, soften the ramp:

```
db.settings(acceleration=180)    # wheel-deg/s²; default 1500
```

2.7 Alternative: DC gear motors with encoders

The original starter build — still fully supported. (Both this and the serial-servo build run in the MuJoCo simulator via the driver shim.)

Part	Qty	Notes
JGB37-520 DC motor with encoder (1:30 gearing)	2	Pick the 12V version; runs fine off 7.4V 2S LiPo
L298N dual H-bridge module	1	Cheap and robust. TB6612FNG is a better choice if you can find it
2S LiPo (7.4V) + buck converter (→5V, ≥2A)	1 each	Don't power the ESP32 from the L298N's onboard 78M05 regulator — it's good for ~300 mA and browns out the moment the motors draw current

Wiring topology:

```
    [ 2S LiPo 7.4V ]
        │
        ├──────────────►  L298N  Vmotor   (motor power)
        │
        └──► Buck  ──►  5V rail
                          │
                          ├──►  ESP32 VIN
                          ├──►  L298N  +5V (logic only)
                          └──►  Sensors via 3.3V regulator on ESP32
```

GPIO map (see `examples/esp32_drivebase.py` for the ESP32-S3 pin assignments; the classic-ESP32 equivalents are in git history):

Function	ESP32-S3 GPIO(s)	Devices on this line
Left motor dir	1, 2	L298N / TB6612 IN1, IN2
Left motor PWM	17	L298N / TB6612 ENA
Left encoder A, B	7, 8	JGB37-520 encoder channels
Right motor dir	9, 10	L298N / TB6612 IN3, IN4
Right motor PWM	11	L298N / TB6612 ENB
Right encoder A, B	12, 13	JGB37-520 encoder channels

The map deliberately leaves GPIO **39 and 38** free — those are the firmware's default **program button** and **BLE-toggle button** (see *openbricks.hub.ESP32S3DevkitHub*). The launcher polls GPIO 39 as an input, so a motor driver toggling it would read as button presses and stop your program. GPIO 15/16 (I2C) and 14/41 (serial-bus UART) are also kept free so sensors and a serial-servo arm can join the same build unchanged.

2.7.1 Calibrating encoder counts

The default in `jgb37_520.py` is `counts_per_output_rev=1320`, which is `11 CPR × 30:1 × 4 (quadrature edges)`. If you have a different gearbox variant, recompute:

```
counts_per_output_rev = encoder_CPR × gear_ratio × 4
```

Or measure empirically: rotate the output shaft by hand exactly one full turn and read `motor.angle()`. Whatever it reports is what `counts_per_output_rev` should be, scaled so that one turn = 360°.

CHAPTER

THREE

COMMAND-LINE TOOL

`pipx install openbricks` installs one console script, `openbricks`, which mirrors the `pybricksdev` workflow: flash firmware over USB, then run / upload / stop programs and pull logs over BLE. With the `[sim]` extra installed, `openbricks sim …` forwards to the *MuJoCo-backed simulator*.

A typical session:

```
$ openbricks flash --name RobotA     # port, chip and newest firmware auto-
↪detected
$ openbricks list                    # hubs in BLE range
$ openbricks run -n RobotA main.py   # push + stream output
$ openbricks upload -n RobotA main.py  # stage; start it with the hub button
$ openbricks stop -n RobotA          # Ctrl-C a running program
$ openbricks log -n RobotA           # dump the most recent run log
$ openbricks docs hardware           # open this manual offline in your␣
↪browser
```

3.1 Firmware versions and provenance

`openbricks flash` first reports the firmware already on the chip — version plus an `(official)` / `(customized)` suffix — before it looks up the newest release. Flashing the **same version** again, or an **older** one, asks for confirmation first; pass `--yes` to skip the prompt in scripts.

The default output is step-level (probe, download, erase, write, hub name, marker, reboot); pass `--verbose` / `-v` to also echo every underlying `mpremote` / `esptool` command line and the firmware cache paths — useful when reporting a flash problem.

Every firmware image published by CI is signed (Ed25519), and the CLI ships the matching public key. An image whose `.bin.sig` verifies is labeled `(official)`; anything else — a self-built image, a missing or wrong signature — is `(customized)`. Customized firmware flashes normally: the suffix is provenance, not a gate. After each flash the verdict is stored on the hub, which is how the next `openbricks flash` labels the current firmware.

The suffix follows the version everywhere it reaches you: the `firmware 1.79.0 (official)` banner at the top of every `openbricks run`, the `started:` header line in every run log (`openbricks log`), and the flash preflight above. On the hub, `openbricks.firmware_label()` returns the same string.

3.2 Programs are compiled on the host

Since 1.92.0, `openbricks run` and `openbricks upload` cross-compile your script with `mpy-cross` **before** connecting, and stage compiled bytecode instead of source (like Pybricks). Three things get better:

- **syntax errors surface in milliseconds**, on your terminal, naming your file and line and quoting the offending source line — no BLE scan, no connect, no upload round-trip;
- **programs start faster**: the hub loads bytecode directly and skips its on-device parse/compile step;
- **tracebacks name your real file and line** (`File "square.py", line 12`) instead of `File "<string>"`.

No flags, nothing to configure. Firmware older than 1.92.0 can't run compiled programs, so the CLI probes the hub's version in-session and sends plain source instead — announced on stderr, never silently. `upload --path` (custom boot flows) always stages your file verbatim, uncompiled, at the path you give.

`run` is an **upload-then-run** (since 2.7.0, deliberately different from Pybricks): it stages your script at the button's `/program.mpy` before executing it, so even a run that fails midway leaves the program on the hub — press the start button to rerun it, no BLE round trip needed. The flip side: running a calibration or a one-shot diagnostic replaces the button's program too, so re-upload your mission after such tools (`upload` alone stages without running).

`flash --with-qtr-init` additionally stores a starter QTR line-sensor calibration at `/qtr.cal` (recorded on the reference bench, default pins 1-10), so the line-follow examples work on a fresh hub out of the box. Heights, mats and lighting differ — run `examples/qtr_calibrate.py` once for a calibration measured on your own rig.

3.3 Reference

The reference below is generated from the CLI's own argument parser, so it always matches the installed version.

Host-side CLI for flashing and running code on openbricks hubs, plus a MuJoCo-backed simulator (`openbricks sim …`).

```
usage: openbricks [-h] [--version] COMMAND ...
```

3.3.1 Positional Arguments

COMMAND Possible choices: flash, run, upload, stop, list, log, servo-id, paste-probe, docs, doc, sim

3.3.2 Named Arguments

--version Print the openbricks package version and exit.

3.3.3 Sub-commands

flash

Flash a firmware image onto a hub (via esptool) and write the hub's BLE advertising name into NVS (via mpremote). –name is mandatory so every hub gets a unique identifier — two hubs with the same name can't be individually addressed over BLE.

```
openbricks flash [-h] --name NAME [--port PORT] [--firmware FIRMWARE]
                 [--chip CHIP] [--baud BAUD] [--with-qtr-init] [--skip-
↪erase]
                 [--yes] [--verbose]
```

Named Arguments

--name Hub identifier for BLE (required, <=20 chars recommended).

--port Serial port (/dev/ttyUSB0, /dev/cu.usbserial-XXXX, COM5 …). Omit to auto-detect — works when exactly ONE ESP device is connected (Espressif native USB or a CP210x/CH340/FTDI bridge).

--firmware Path to firmware.bin produced by scripts/build_firmware.sh or downloaded from the Releases page. Omit to download the newest release automatically for the detected chip (cached under ~/.cache/openbricks/firmware).

--chip esptool –chip value (esp32, esp32s3, auto). Default: auto.

Default: `'auto'`

--baud esptool flash baud rate. Default: 460800.

Default: `'460800'`

--with-qtr-init After flashing, store a starter QTR line-sensor calibration at /qtr.cal (recorded on the reference bench, default pins 1-10) so line-follow examples work out of the box. Re-run examples/qtr_calibrate.py for a calibration measured on your own mat and lighting.

Default: `False`

--skip-erase Skip erase_flash (faster dev loop; leaves stale NVS keys).

Default: `False`

--yes Skip the confirmation prompt when the target firmware is the same version as (or older than) the current one.

Default: `False`

--verbose, -v Echo every subprocess command line (mpremote/esptool) and cache paths. Default output is step-level only.

Default: `False`

run

Connect to the named hub over BLE, push SCRIPT to its REPL (via paste mode), and stream stdout/stderr back to this terminal until the script finishes. Ctrl-C interrupts the remote program.

```
openbricks run [-h] -n NAME [-c CODE] [--scan-timeout SCAN_TIMEOUT] [--
↪debug]
               [SCRIPT]
```

Positional Arguments

SCRIPT Path to the local Python script to run on the hub. Mutually exclusive with -c.

Named Arguments

-n, --name Hub name baked in at flash time (`openbricks flash --name`).

-c, --code Inline Python code to run on the hub (analogous to `python -c CODE`). Useful for quick diagnostics — e.g. `openbricks run -n ls -c 'import openbricks; print(openbricks.__version__)'`. Mutually exclusive with the SCRIPT positional.

--scan-timeout How long to scan for the named hub before giving up. Default: 5.0 s.

Default: `5.0`

--debug Print every BLE notify packet (timestamp + hex + ascii) to stderr as it arrives. Use to diagnose 'timed out reading from hub' errors — tells you whether the hub is sending anything at all.

Default: `False`

upload

Upload SCRIPT to the hub's filesystem (default path `/program.py`). The uploaded code does NOT run automatically — the hub's frozen main.py watches the hub button and exec's the staged script on each short press. Second short-press stops a running program. (Pybricks calls this same operation `download` from the hub's perspective; we name by direction-of-data-travel — bytes flow *up* to the hub.)

```
openbricks upload [-h] -n NAME [--path PATH] [--scan-timeout SCAN_TIMEOUT]
                  SCRIPT
```

Positional Arguments

SCRIPT Path to the local Python script to stage.

Named Arguments

-n, --name Hub name baked in at flash time.

--path Destination path on the hub's filesystem; the file is staged VERBATIM there (no compilation) for custom boot flows. Default: compile with mpy-cross and stage /program.mpy (which the frozen launcher runs; older firmware gets the source at /program.py).

--scan-timeout BLE scan timeout. Default: 5.0 s.

Default: `5.0`

stop

Connect to the named hub over BLE and send a single Ctrl-C, which MicroPython surfaces as KeyboardInterrupt. Use when a long-running `openbricks run` has already ended and you just want the hub to idle again.

```
openbricks stop [-h] -n NAME [--scan-timeout SCAN_TIMEOUT]
```

Named Arguments

-n, --name Hub name.

--scan-timeout BLE scan timeout. Default: 5.0 s.

Default: `5.0`

list

Run a BLE scan and print every device found, sorted by RSSI (strongest first). Unnamed devices are shown with a placeholder so you can still spot a hub whose name wasn't flashed.

```
openbricks list [-h] [--timeout TIMEOUT] [--all]
```

Named Arguments

--timeout Scan duration in seconds. Default: 5.0.

Default: `5.0`

--all Show every BLE device, not just those with names. Useful when debugging a hub that came up without a flashed name.

Default: `False`

log

Every program executed via the launcher (button press OR `openbricks run`) gets its stdout / stderr tee'd to a flash file under `/openbricks_logs/`. Ten rotating slots are kept. With no flags this prints the most recent run; `--list` shows the index; `--run N` selects a specific slot. Useful for post-mortem on an untethered run where no live console was attached.

```
openbricks log [-h] -n NAME [--list] [--run RUN] [--scan-timeout SCAN_
↪TIMEOUT]
```

Named Arguments

-n, --name Hub name baked in at flash time.

--list List the available run indices + their on-flash size, instead of dumping a run's contents.

Default: `False`

--run Specific run index to dump. Defaults to the most recent.

--scan-timeout BLE scan timeout. Default: 5.0 s.

Default: `5.0`

servo-id

Scans the bus (IDs 0..253), rewrites the servo's EEPROM ID register, and verifies the result. With several servos attached, –old-id is required so the tool never guesses which one to re-ID. Wire the servo to a USB half-duplex adapter (e.g. the URT-2 board) — this talks directly to the adapter's serial port, no hub involved.

```
openbricks servo-id [-h] [-p PORT] [-n NAME] [--tx TX] [--rx RX]
                    [--scan-timeout SCAN_TIMEOUT] [--scan] [--old-id OLD_ID]
                    [--baudrate BAUDRATE] [--timeout TIMEOUT]
                    [new_id]
```

Positional Arguments

new_id Bus ID to assign (0..253). Omit with –scan.

Named Arguments

-p, --port Serial port of the USB adapter, e.g. /dev/cu.usbmodem123. Omitted (and no -n): auto-detected when exactly one USB serial device is connected.

-n, --name Hub name: run the scan/re-ID THROUGH THE HUB over BLE instead of a USB adapter — the servo stays wired to the robot. Mutually exclusive with -p.

--tx Hub path only: servo-bus TX pin (default 14).

Default: `14`

--rx Hub path only: servo-bus RX pin (default 41).

Default: `41`

--scan-timeout Hub path only: BLE scan timeout. Default: 5.0 s.

Default: `5.0`

--scan Just list the IDs that answer on the bus; change nothing.

Default: `False`

--old-id Current ID of the servo to re-ID. Required when more than one servo is attached; otherwise auto-detected.

--baudrate Bus baudrate. Default: 1000000 (Feetech factory).

Default: `1000000`

--timeout Per-ping serial read timeout in seconds. Default: 0.02 (a full 254-ID scan takes ~5 s).

Default: `0.02`

paste-probe

Pastes padded no-op programs of increasing size through the real raw-paste path and reports the largest that completes, plus how each failure presents (truncated vs hung). Use before changing the firmware's MICROPY_REPL_STDIN_BUFFER_MAX: two windows may be in flight at once, so that setting is only safe at or below the measured limit.

```
openbricks paste-probe [-h] -n NAME [--scan-timeout SCAN_TIMEOUT] [--max␣
↪MAX]
                       [--timeout TIMEOUT]
```

Named Arguments

-n, --name	Hub BLE name.
--scan-timeout	BLE scan timeout in seconds. Default: 5.
	Default: `5.0`
--max	Largest size to try, bytes. Default: 8192.
	Default: `8192`
--timeout	Per-size timeout in seconds. Default: 15.
	Default: `15.0`

docs (doc)

Opens the full manual in your browser — the same Sphinx build as docs.openbricks.dev, API reference included, bundled and served from disk so no internet is needed. Pass a topic to jump straight to that page.

```
openbricks docs [-h] [topic]
```

Positional Arguments

topic	Page to open (e.g. install, hardware, robotics). Guides and API pages both work. Omit for the index.

sim

Forwards all remaining arguments to the MuJoCo-backed simulator's CLI. Use `openbricks sim --help` to see the sim's own subcommand list.

```
openbricks sim
```

CHAPTER

FOUR

SIMULATOR

The `[sim]` extra ships a MuJoCo-backed physics simulator, so you can develop robot programs without a hub on the desk:

```
$ pipx install 'openbricks[sim]'
$ openbricks sim run examples/full_robot.py --viewer
```

The sim runs the **same script you'd push to the hub** — a driver shim maps the `openbricks` API onto simulated motors and sensors, so `ST3032Motor`, `DriveBase`, color sensors, and distance sensors behave like their hardware counterparts.

4.1 Commands

```
$ openbricks sim preview [--world WORLD] [--x X] [--y Y] [--headless] [--
↪duration S] [--seed N]
```

Loads the named world (an alias or a path to an MJCF file), splices in the default chassis, and opens the MuJoCo viewer so you can inspect the scene. `--headless` steps the physics for `--duration` seconds without opening a window — useful as a smoke test.

```
$ openbricks sim run SCRIPT [--world WORLD] [--x X] [--y Y] [--viewer] [--
↪no-shim] [--seed N]
```

Loads the world plus the default chassis and executes `SCRIPT` against the simulated robot. `--viewer` opens the interactive MuJoCo window; without it the sim runs headless (CI-friendly). `--seed` makes randomized worlds reproducible.

Run `openbricks sim --help` for the full, always-current option list.

4.2 Notes

- The sim needs the `[sim]` extra (`mujoco`, `numpy`). Without it, `openbricks sim …` prints an install hint instead of crashing.
- Firmware-only users never need the simulator — it's strictly host-side tooling.

CHAPTER

FIVE

EXAMPLES

The repo ships runnable example programs in `examples/`. Push any of them to a hub with `openbricks run -n <name> <script>` or try them in the simulator with `openbricks sim run <script>`.

Two representative ones are reproduced below.

5.1 Drive a square (ST-3032 drivebase)

```
# SPDX-License-Identifier: MIT
"""
Drive a square with the ST-3032 drivebase.

Four sides of ``SIDE_MM`` straight + ``+90°`` turn each. Demonstrates
``DriveBase.straight`` / ``turn`` composing into a closed loop — if
the chassis geometry (``WHEEL_DIAMETER_MM`` / ``AXLE_TRACK_MM``) is
calibrated correctly, the robot returns to within a few cm of its
starting pose after one lap.

Uses the new ``then="coast"`` default end-state (1.6.7), so the
wheels free-wheel briefly between segments. If your bench has
significant momentum carryover and you'd rather pin the wheels
between sides, pass ``then="brake"`` to the ``straight`` / ``turn``
calls (or ``then="hold"`` for active position lock — ST-3032
supports it).

Hardware: identical to ``examples/st3032_drivebase_test.py``.

Run with:
    openbricks run -n ls examples/st3032_drivebase_square.py
"""

from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase


LEFT_ID, RIGHT_ID = 2, 1
UART_ID, TX, RX   = 1, 14, 41

WHEEL_DIAMETER_MM = 88
```

(continues on next page)

(continued from previous page)

```
AXLE_TRACK_MM     = 136

SIDE_MM    = 200
NUM_LAPS   = 1

STRAIGHT_SPEED_MM = 150
TURN_RATE_DPS     = 200


def main():
    print("--- ST-3032 drivebase square (%d mm sides × %d laps) ---" %
          (SIDE_MM, NUM_LAPS))

    left  = ST3032Motor(servo_id=LEFT_ID,  uart_id=UART_ID, tx=TX, rx=RX,␣
↪invert=True)
    right = ST3032Motor(servo_id=RIGHT_ID, uart_id=UART_ID, tx=TX, rx=RX)

    db = DriveBase(left, right,
                   wheel_diameter_mm=WHEEL_DIAMETER_MM,
                   axle_track_mm=AXLE_TRACK_MM)
    db.settings(straight_speed=STRAIGHT_SPEED_MM, turn_rate=TURN_RATE_DPS)

    for lap in range(NUM_LAPS):
        for side in range(4):
            print("  lap %d side %d: straight(%d) → turn(+90)" %
                  (lap + 1, side + 1, SIDE_MM))
            db.straight(SIDE_MM)
            db.turn(90)

    print("--- done ---")


main()
```

5.2 Gyro-corrected square (ICM-45686)

With an ICM-45686 attached, `use_gyro(True)` moves heading control off the encoders and onto measured body rotation — corrected every millisecond inside the firmware's 1 kHz control tick, with no Python in the loop. This script drives the same square twice, encoders-only and gyro-corrected, and prints each pass's heading drift so the difference is a number, not an impression. On the reference bench the gyro pass returns within ~0.6° over all four turns; wheel slip that would bend the encoder pass simply gets steered back out.

```
# SPDX-License-Identifier: MIT
"""
Gyro-corrected square — ICM-45686 on the hard tick.

The ICM-45686 is read INSIDE the 1 kHz control tick over SPI, so
with ``use_gyro(True)`` the DriveBase corrects heading every
```

(continues on next page)

(continued from previous page)

```
millisecond in C — no Python in the loop. This script drives a
square twice, encoder-only and then gyro-corrected, and prints how
far each pass drifts from its starting heading. Bench 2026-08-10:
the gyro pass returned within +0.6 degrees over all four turns.

The learned gyro bias is persisted to NVS after the stillness
lock, so later boots start corrected without the ~0.5 s wait.

Wiring: breakout on 3V3/GND plus the four SPI pins below; INT
stays unwired (the tick polls). Needs ~0.5 m of clear floor.

Run:
    openbricks run -n <hub> examples/icm45686_square.py
"""

import time

from openbricks.drivers.icm45686 import ICM45686
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

LEFT_ID, RIGHT_ID = 2, 1
UART_ID, TX, RX = 1, 14, 41
SCK, MOSI, MISO, CS = 12, 13, 11, 17

WHEEL_DIAMETER_MM = 88
AXLE_TRACK_MM = 136

STRAIGHT_SPEED = 300
TURN_RATE = 200
SIDE_MM = 300


def square_drift(db, imu):
    start = imu.heading()
    for side in range(4):
        db.straight(SIDE_MM)
        db.turn(90)
        print("corner %d: heading %.1f" % (side + 1, imu.heading() - start))
    return imu.heading() - start - 360


print("hold still for gyro bias lock ...")
imu = ICM45686(sck=SCK, mosi=MOSI, miso=MISO, cs=CS)
while not imu.calibrated():
    time.sleep_ms(100)
imu.save_calibration()

left = ST3032Motor(servo_id=LEFT_ID, uart_id=UART_ID, tx=TX, rx=RX,
```

(continues on next page)

(continued from previous page)

```
                    invert=True)
right = ST3032Motor(servo_id=RIGHT_ID, uart_id=UART_ID, tx=TX, rx=RX)
db = DriveBase(left, right, wheel_diameter_mm=WHEEL_DIAMETER_MM,
               axle_track_mm=AXLE_TRACK_MM, imu=imu)
db.settings(straight_speed=STRAIGHT_SPEED, turn_rate=TURN_RATE)

print("pass 1: encoders only")
print("drift: %+.1f deg" % square_drift(db, imu))

db.use_gyro(True)
print("pass 2: gyro-corrected at 1 kHz")
print("drift: %+.1f deg" % square_drift(db, imu))
db.stop(then="brake")
```

Wiring for the IMU is four SPI pins plus power — see *Hardware guide*. The first still half-second learns the gyro bias; `save_calibration()` persists it so later boots skip the wait.

5.3 Rounded square (DriveBase.curve)

`curve(radius, angle)` follows the Pybricks contract — positional order, parameter names, and sign semantics: positive `angle` arcs right (clockwise), a negative `radius` drives the arc backward, and `curve(0, angle)` degrades to a turn in place. The forward and turn profiles run with proportionally scaled speed *and* acceleration, so the path is a true circle even through the ramps, and the outer wheel is automatically capped at the `straight_speed` setting. (One deviation: `then` defaults to `"coast"` like every openbricks move; pass `then="hold"` for the Pybricks end state.)

```
# SPDX-License-Identifier: MIT
"""Drive a rounded square with DriveBase.curve().

Four straights joined by four quarter-circle arcs — the robot never
stops to pivot, so the lap is faster and smoother than the
straight+turn square. curve(radius, angle) follows Pybricks: positive
angle arcs right, the radius sign picks forward/backward, and the
outer wheel is automatically capped at the straight_speed setting.

Run with:
    openbricks run -n ls examples/st3032_drivebase_curve.py
"""

from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

LEFT_ID, RIGHT_ID = 2, 1
UART_ID, TX, RX = 1, 14, 41

WHEEL_DIAMETER_MM = 88
AXLE_TRACK_MM = 136

SIDE_MM = 150
```

(continues on next page)

(continued from previous page)

```
RADIUS_MM = 60
NUM_LAPS = 1

left = ST3032Motor(servo_id=LEFT_ID, uart_id=UART_ID, tx=TX, rx=RX,
                   invert=True)
right = ST3032Motor(servo_id=RIGHT_ID, uart_id=UART_ID, tx=TX, rx=RX)
db = DriveBase(left, right,
               wheel_diameter_mm=WHEEL_DIAMETER_MM,
               axle_track_mm=AXLE_TRACK_MM)
db.settings(straight_speed=150, turn_rate=200)

print("rounded square: %d mm sides, %d mm corner radius" %
      (SIDE_MM, RADIUS_MM))
for lap in range(NUM_LAPS):
    for side in range(4):
        db.straight(SIDE_MM)
        db.curve(radius=RADIUS_MM, angle=90)
print("done")
```

5.4 Square up on a line (QTR sensor bar)

The classic align move on the `QTRLineSensor` window: each half of the ten-element bar acts as one virtual corner sensor, in two passes. Seek: drive slowly toward the line — the wheel whose half reaches it first stops while the other keeps rolling, pivoting the chassis square. Edge: servo each wheel proportionally — the follower's KP discipline — until its half reads ambient of about 50, the elements straddling the black/white boundary, parked right ON the line's edge. Calibrate once with `examples/qtr_calibrate.py` first; mount the bar ahead of the wheels.

```
# SPDX-License-Identifier: MIT
"""Square up on the edge of a perpendicular line, QTRLineSensor.

The classic FLL/WRO align move, on one sensor bar instead of two
corner sensors: each half of the window is one virtual corner
sensor, and each wheel servos proportionally until its half's mean
ambient reads 50 — the elements straddling the black/white
boundary. The wheel whose half arrives first holds while the other
pivots the chassis on; overshoot backs up. Both halves end ON the
edge, so the bar — and the chassis — is square right at it.

Run ``examples/qtr_calibrate.py`` once first. The bar must be
mounted ahead of the wheels; the farther ahead, the finer the final
heading.
"""

import time

from openbricks.drivers.qtr import QTRLineSensor
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase
```

(continues on next page)

(continued from previous page)

```
# --- control law (pure logic, unit-tested in tests/test_qtr_align.py) ---

KP = 1.3
EDGE_TOLERANCE = 8
SIDE_COUNT = 5


def side_ambient(elements):
    total = 0
    for e in elements:
        total += e.ambient()
    return total // len(elements)


def edge_dps(elements, target):
    error = side_ambient(elements) - target
    if abs(error) <= EDGE_TOLERANCE:
        return 0
    return int(KP * error)


def edge_wheel_speeds(reading):
    left = edge_dps(reading.elements[:SIDE_COUNT], 50)
    right = edge_dps(reading.elements[-SIDE_COUNT:], 50)
    if left == 0 and right == 0:
        return None
    return (left, right)

# --- end control law ---


qtr = QTRLineSensor()
qtr.load_calibration("/qtr.cal")

left_motor = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=41,
                         invert=True)
right_motor = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=41)
db = DriveBase(left_motor, right_motor,
               wheel_diameter_mm=88, axle_track_mm=136)

print("aligning on the line ...")
while True:
    speeds = edge_wheel_speeds(qtr.read())
    if speeds is None:
        break
    db.move_wheels(speeds[0], speeds[1])
    time.sleep_ms(5)
db.stop(then="brake")
```

(continues on next page)

(continued from previous page)

```
print("aligned - square on the edge")
```

5.5 Square up on a line (two color sensors)

The same maneuver with a corner-mounted color sensor per side, for rigs without the QTR bar.

```
# SPDX-License-Identifier: MIT
"""
Demo: square the robot up on a dark line using two colour sensors.

The classic FLL/WRO "align on a line" move: drive slowly toward a
black line with one colour sensor mounted near each front corner,
ahead of the wheels. The moment a sensor crosses onto the line its
wheel brakes — the other wheel keeps rolling, pivoting the chassis
until *its* sensor reaches the line too. When both have stopped, the
sensor pair (and therefore the chassis) is parallel to the line, no
matter how crooked the approach was.

Geometry matters: the sensors must sit ahead of the axle and be
mounted symmetrically. The farther apart they are, the more accurate
the final heading.

Line detection reuses the idea from ``color_array.py``: a black line
on a light mat is simply "too dark to be the mat" — ``ambient()``
below a threshold. Print ``sensor.ambient()`` over your own mat and
line and put ``LINE_AMBIENT`` between the two readings.

Hardware (same bus layout as ``color_array.py`` / ``full_robot.py``):
    * ESP32-S3 (I2C on 15/16; serial bus UART on 14/6)
    * 2x ST-3032 wheel servos, IDs 1 (left) / 2 (right)
    * TCA9548A mux, one TCS34725 per channel: 0 = left, 1 = right,
      both facing the mat at the front of the chassis
"""

from machine import I2C, Pin
from openbricks.tools import wait

from openbricks.drivers.st3032 import ST3032Motor
from openbricks.drivers.tca9548a import TCA9548A
from openbricks.drivers.tcs34725 import TCS34725
from openbricks.robotics import DriveBase


left_motor = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=41)
right_motor = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=41, invert=True)
db = DriveBase(left_motor, right_motor,
               wheel_diameter_mm=88, axle_track_mm=136)

i2c = I2C(0, sda=Pin(15), scl=Pin(16), freq=400_000)
```

(continues on next page)

(continued from previous page)

```
mux = TCA9548A(i2c)
left_sensor = TCS34725(mux[1])
right_sensor = TCS34725(mux[0])


LINE_AMBIENT = 20


def align_on_line():

    approach_dps = 100
    poll_ms = 10
    timeout_ms = 8000

    left_done = False
    right_done = False
    left_ambient = None
    right_ambient = None
    speeds = [approach_dps, approach_dps]
    db.move_wheels(speeds[0], speeds[1])
    try:
        for _ in range(max(1, timeout_ms // poll_ms)):
            if not left_done:
                left_ambient = left_sensor.ambient()
                if left_ambient < LINE_AMBIENT:
                    speeds[0] = 0
                    db.move_wheels(speeds[0], speeds[1])
                    left_done = True
                    print('left wheel stopped (ambient=%d)' % left_ambient)
            if not right_done:
                right_ambient = right_sensor.ambient()
                if right_ambient < LINE_AMBIENT:
                    speeds[1] = 0
                    db.move_wheels(speeds[0], speeds[1])
                    right_done = True
                    print('right wheel stopped (ambient=%d)' % right_
↪ambient)

            if left_done and right_done:
                return
            wait(poll_ms)
    finally:
        db.stop(then="brake")
    raise RuntimeError(
        "no line found within %d ms (last ambient: left=%r right=%r) — "
        "is the line in reach, and is LINE_AMBIENT calibrated for "
        "your mat?" % (timeout_ms, left_ambient, right_ambient))
```

(continues on next page)

(continued from previous page)

```
def main():

    print("aligning on the line ...")
    align_on_line()
    print("aligned — square to the line.")
    wait(500)
    left_motor.coast()
    right_motor.coast()


if __name__ == "__main__":
    main()
```

5.6 Full robot (drivebase + IMU + color sensor array)

```
# SPDX-License-Identifier: MIT
"""
Example: a small robot that rolls forward until its colour sensor sees red,
then demos a servo wave.

Hardware:
    * ESP32-S3 DevKitC-1 (or classic ESP32 DevKitC-V4)
    * 2× JGB37-520 DC gear motors (with Hall-effect quadrature encoders)
        driven by a shared L298N (or TB6612FNG) H-bridge
    * 1× ICM-45686 IMU on SPI
    * 1× TCS34725 RGB colour sensor on the same I2C bus
    * 1× ST-3215 serial bus servo on UART (optional — the servo demo
        at the bottom just prints a message if it isn't attached)

Edit the GPIOs at the top to match your wiring.
"""

import time

from machine import I2C, Pin, UART

from openbricks.drivers.icm45686 import ICM45686
from openbricks.drivers.jgb37_520 import JGB37Motor
from openbricks.drivers.st3215 import ST3215
from openbricks.drivers.tcs34725 import TCS34725
from openbricks.robotics import DriveBase


I2C_SDA  = 15
I2C_SCL  = 16

SCK, MOSI, MISO, CS = 12, 13, 11, 17
```

(continues on next page)

(continued from previous page)

```
LEFT_IN1,  LEFT_IN2,  LEFT_PWM  = 4, 5, 6
LEFT_EA,   LEFT_EB              = 7, 8

RIGHT_IN1, RIGHT_IN2, RIGHT_PWM = 9, 10, 11
RIGHT_EA,  RIGHT_EB             = 12, 13

SERVO_TX, SERVO_RX = 17, 18
SERVO_ID = 1

WHEEL_DIAMETER_MM = 56
AXLE_TRACK_MM     = 114


i2c   = I2C(0, sda=Pin(I2C_SDA), scl=Pin(I2C_SCL), freq=400_000)
imu   = ICM45686(sck=SCK, mosi=MOSI, miso=MISO, cs=CS)
color = TCS34725(i2c)

left  = JGB37Motor(in1=LEFT_IN1,  in2=LEFT_IN2,  pwm=LEFT_PWM,
                   encoder_a=LEFT_EA,  encoder_b=LEFT_EB)
right = JGB37Motor(in1=RIGHT_IN1, in2=RIGHT_IN2, pwm=RIGHT_PWM,
                   encoder_a=RIGHT_EA, encoder_b=RIGHT_EB)

drivebase = DriveBase(left, right,
                      wheel_diameter_mm=WHEEL_DIAMETER_MM,
                      axle_track_mm=AXLE_TRACK_MM)

try:
    uart = UART(1, baudrate=1_000_000, tx=SERVO_TX, rx=SERVO_RX)
    arm  = ST3215(uart, servo_id=SERVO_ID)
except Exception as e:
    print("no servo attached:", e)
    arm = None


while True:
    r, g, b = color.rgb()
    heading = imu.heading()

    print("rgb=({:3d},{:3d},{:3d})  heading={:6.1f}".format(r, g, b,␣
↪heading))

    if r > g and r > b and r > 120:
        print("Red detected — stopping.")
        drivebase.stop()
        break

    drivebase.drive(speed_mm_s=150, turn_rate_dps=0)
    time.sleep_ms(50)
```

(continues on next page)

(continued from previous page)

```
if arm is not None:
    arm.move_to(180, speed=500)
    time.sleep_ms(500)
    arm.move_to(0, speed=500)
```

CHAPTER

SIX

MEASURING WHEEL DIAMETER & AXLE TRACK

`DriveBase` converts your commands from millimeters and body-degrees into wheel rotations using exactly two numbers:

```
db = DriveBase(left, right, wheel_diameter_mm=88, axle_track_mm=138)
```

- **`wheel_diameter_mm`** — how far one wheel rotation moves the robot. Every `straight()` distance scales linearly with it.
- **`axle_track_mm`** — the distance between the two wheels' *contact points* with the floor. Every `turn()` angle scales linearly with it.

Getting these two right is worth more than any controller tuning: a 2 % diameter error is 20 mm of error on a 1 m drive, and a 2 % track error is ~7° of error on a full spin. A ruler gets you close; the two test drives below get you to a few tenths of a percent.

Calibrate **wheel diameter first, then axle track** — the turn calculation uses the wheel diameter, so a diameter error contaminates the track measurement.

6.1 Step 1 — wheel diameter, from a straight drive

Start from the nominal value (caliper across the tire, or the manufacturer's spec). Then:

1. Put a strip of masking tape on the floor and align a marked point of the robot (e.g. the axle center) with its edge.
2. Run a long straight — the longer the drive, the better the resolution:

   ```
   db.settings(straight_speed=150)
   db.straight(1000)
   ```

3. Measure the distance actually traveled, from tape edge to the same marked point, in millimeters.
4. Scale the diameter by how far the robot *really* went:

   ```
   new_diameter = old_diameter × measured_mm / 1000
   ```

   Traveled 1023 mm with `wheel_diameter_mm=88`? Then `88 × 1023 / 1000 = 90.0` is your real diameter.

Repeat once with the new value: the measured distance should now land within a few millimeters of the command. Soft tires compress under the robot's weight, so the *effective* diameter is often ~1-3 % smaller than the caliper says — the test drive measures reality, load included.

6.2 Step 2 — axle track, from an in-place spin

With the wheel diameter calibrated, the fastest way is to let an attached ICM-45686 measure the spin for you: `examples/icm45686_axle_track.py` commands ten encoder-only turns (the drivebase's gyro stays off — the point is to measure what the *encoders* produce), reads the true rotation off the IMU, and prints the corrected `axle_track_mm` to paste into your `DriveBase(...)` call. Ten turns make each 0.1° of gyro error only 0.003 % of track.

No IMU on the robot? The manual version:

1. Align the robot against a straightedge (a wall or ruler touching both wheels works well) and note its exact heading.
2. Command several full spins in place — more turns amplify the error so it's easier to measure:

   ```
   db.settings(turn_rate=120)
   db.turn(3600)         # ten full clockwise spins
   ```

3. Measure the final heading error `err_deg` against your straightedge: positive if the robot rotated *past* the start alignment, negative if it stopped short. A protractor or a phone compass app is plenty — over ten spins, each degree of final error is only 0.1 % of track.
4. Scale the track by how far the robot really rotated:

   ```
   new_track = old_track × 3600 / (3600 + err_deg)
   ```

   Overshot by 18° with `axle_track_mm=138`? Then `138 × 3600 / 3618 = 137.3` is your real track.

Note the direction: if the robot turns **too far**, the real track is **smaller** than configured (each wheel-degree of travel produced *more* body rotation than the math assumed), so the correction *decreases* the configured value. Stopping short means the opposite — increase it.

The contact *points* matter, not the wheel centers: wide, soft tires effectively touch the ground inboard of their centerline, so the real track is usually a few millimeters less than what you measure center-to-center with a ruler.

6.3 Checking the result

Drive a square and see how close the robot returns to its start pose:

```
for _ in range(4):
    db.straight(300)
    db.turn(90)
```

With both values calibrated, the return error on a 300 mm square is typically under 1 cm and a few degrees. Do the calibration on the same surface the robot will compete on — carpet, foam mats, and wood all load the tires differently.

6.4 What about the gyro?

`use_gyro(True)` (with an `imu=` attached) makes turns terminate on *measured* body rotation, so heading no longer depends on the axle track being exact — wheel slip included. Calibrate the track anyway: the

controller still uses it to shape the commanded wheel speeds, and encoder-only operation (gyro off, or no IMU on the robot) depends on it entirely.

How much the gyro buys depends on which IMU: an ICM-45686 read inside the 1 kHz control tick returned a four-turn square within **+0.6° total** on the reference bench, versus +0.5° to +1.8° *per turn* for a Python-pumped BNO055 and a few degrees encoder-only. `examples/icm45686_square.py` measures all of this on your own robot — run it and read the two drift numbers.

CHAPTER

SEVEN

API REFERENCE

Everything below ships frozen into the firmware image — `import openbricks` on the hub gives you all of it. The pages are generated from the package's docstrings.

The API is layered:

- **Drivers** (`openbricks.drivers.*`) — one module per supported component. Construct these with your wiring's pins / IDs.
- **Interfaces** (`openbricks.interfaces`, `openbricks.distance`) — the abstract contracts drivers implement. Higher layers depend only on these, which is what makes the system plug-and-play.
- **Robotics** (`openbricks.robotics`) — building blocks composed from interfaces, like the two-wheel `DriveBase`.
- **Hub & runtime** — board-level peripherals, BLE, program launching, and log capture.

7.1 `robotics` — DriveBase

Drive a robot, not two motors: `DriveBase` couples a left and a right motor into one chassis with moves in millimeters and body-degrees.

```
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

left  = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=41)
right = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=41, invert=True)

db = DriveBase(left, right, wheel_diameter_mm=88, axle_track_mm=138)
db.settings(straight_speed=250, turn_rate=200,
            acceleration=1000, turn_acceleration=800)

for _ in range(4):           # a 300 mm square
    db.straight(300)
    db.turn(90)
```

Moves block by default. Pass `wait=False` to return immediately and poll `done()` — the Pybricks pattern for driving while reading sensors; any new move command supersedes the pending one:

```
db.straight(600, wait=False)
while not db.done():
```

(continues on next page)

(continued from previous page)

```
    if bumper_pressed():
        db.stop()
        break
    time.sleep_ms(10)
```

For direct control of each wheel — line-following, tank-style teleop, or any controller that computes its own per-wheel outputs — `move_wheels` takes two speeds in wheel-deg/s:

```
db.move_wheels(200, 120)      # gentle right-hand arc
time.sleep_ms(500)
db.stop()
```

Both setpoints leave in a single sync-write packet on serial-bus motors, so the wheels change speed at the same packet boundary. Reach for this rather than a *SyncServoGroup* over the wheels: a DriveBase hands their UART to the native bus driver when it adopts them, so a SyncServoGroup can't drive them at all.

Moves take a `then=` end state: `"stop"` (alias `"coast"`, the default) decelerates to rest and free-wheels; `"brake"` / `"hold"` end actively. `then="continue"` on `straight` and `curve` — Pybricks `Stop.NONE` — does NOT decelerate at the end: the move finishes at cruise speed and the wheels keep it until the next command, so chained segments flow through their seams:

```
db.straight(300, then="continue")    # ends AT cruise
db.curve(150, 90, then="continue")   # picks the speed up
db.straight(300)                     # decelerates to rest
```

Move endings are SHAPED all the way down (2.6.0): the controller runs position integral action (pbio's integrator rules — the same control law Pybricks uses) so tracking error is squeezed out near the target, and any residual left when a profile expires is closed by a small landing trajectory under the same acceleration limit as every other motion — never a raw feedback step. A robot that ends a mission simply comes to rest on its mark; a genuinely stuck robot still refuses to report `done()` and the stall watchdog raises.

`stop()` is Pybricks parity: it coasts and returns immediately. `then` picks the end state (`"coast"`, `"brake"`, `"hold"`). Short moves armed while the robot is already fast raise their own deceleration to land at rest exactly on target, so you rarely need more — but `wait=True` is available to block until both wheels' measured speeds read ~0 (the decel ramp plus settle for brake/hold, the physical freewheel decay for coast). It raises `ValueError` on open-loop pairs (no measured speed) and, if the wheels never settle within 5 s, `RuntimeError` naming the measured speeds — a stopped robot that is still moving is a fault, not a detail to hide.

A wheel that stops answering the bus — no power, a knocked-loose TX/RX wire, the wrong `servo_id` — raises instead of quietly doing nothing, and the error names the motor:

```
OSError: motor is not responding on the bus: right wheel
(servo id 1, slot 1) on UART1 tx=14 rx=6 — 0 replies, 137 failed
reads (137 in a row). Check the servo's power and TX/RX wiring,
and that it really has that bus id — `openbricks servo-id --scan`
lists the ids actually answering on the bus.
```

(`openbricks servo-id` talks through the URT-2's USB port. With the servo already wired to the hub, `openbricks run -n NAME examples/servo_set_id.py` scans and re-IDs through the hub instead — same safety contract.)

Both wheels are verified when the DriveBase is constructed, and on every move afterwards. If one

goes silent mid-move the controller halts immediately rather than winding that wheel's command to the rail — a frozen odometry reading would otherwise look like "infinite error" to the heading loop. `db.check_motors()` runs the same check on demand.

With an IMU attached, `use_gyro(True)` steers by measured body rotation instead of the encoder differential — immune to wheel slip. The preferred IMU is the *ICM45686*: it is read *inside* the 1 kHz control tick over SPI, so the heading correction runs every millisecond in C with no Python in the loop (bench: +0.6° total drift over a four-turn square):

```
from openbricks.drivers.icm45686 import ICM45686

imu = ICM45686(sck=12, mosi=13, miso=11, cs=17)
db = DriveBase(left, right, wheel_diameter_mm=88,
               axle_track_mm=138, imu=imu)
db.use_gyro(True)
```

(A legacy *BNO055* on I2C still works — its fused heading is pumped from Python between ticks, which corrects noticeably slower, typically +0.5° to +1.8° per turn. New builds should use the ICM-45686.)

To re-zero the heading frame mid-mission (say, after squaring up on a line), call `db.reset()` between moves — afterwards the robot's CURRENT pose is heading zero for both the drive base and `imu.heading()`, atomically:

```
db.straight(100)
db.turn(-90)
db.reset()           # here, now = heading zero
db.straight(130)     # drives straight along the NEW zero
```

`imu.reset_heading()` refuses (`OSError`) while a drive base steers by the gyro — same rule as Pybricks ("can't reset heading while gyro in use"): zeroing the integrator under an armed heading controller shifts the measurement out from under the held target, and the next move veers chasing the old frame. Use `db.reset()`, or `use_gyro(False)` first. `db.reset()` itself raises while a move is in progress — stop first.

Accurate `wheel_diameter_mm` / `axle_track_mm` values matter more than any tuning — calibrate both with two short test drives: *Measuring wheel diameter & axle track*. Two-wheel differential drivebase.

Thin Python wrapper over `_openbricks_native.DriveBase` — the C implementation at `native/user_c_modules/openbricks/drivebase.c` that runs 2-DOF coupled control at 1 kHz. Both motors are driven by a single forward-progress trajectory and a heading-hold trajectory; a heading-error feedback term keeps them in sync even when one wheel has more friction than the other.

Public API matches the M1 Python version so existing code and tests don't need to change:

> db = DriveBase(left, right, wheel_diameter_mm=56, axle_track_mm=114) db.settings(straight_speed=200, turn_rate=180) # deg/s at wheels db.straight(500) # mm, blocking db.turn(90) # deg body heading, blocking db.drive(100, 0) # non-blocking kinematic mapping

Serial-bus motors (ST-3215 / ST-3032) are adopted transparently onto the hard-tick engine (firmware) or the emulated bus (sim) — same class, same code, one controller. There is no Python control loop: motor pairs with neither a native servo nor a serial-bus adoption path get open-loop `drive()`/`stop()` only, and `straight()`/`turn()` raise.

Open-loop `drive()` bypasses the coupled controller; it just maps (speed_mm_s, turn_rate_dps) → (left_dps, right_dps) and hands them to each servo's `run_speed`. Useful for interactive control where

profile-based moves would feel sluggish.

class openbricks.robotics.drivebase.**DriveBase**(*left*, *right*, *wheel_diameter_mm*, *axle_track_mm*, *imu=None*, *drive='duty'*)

Bases: object

A two-wheel differential drive robot: two motors, one chassis.

Pybricks-compatible surface: `straight(distance_mm)`, `turn(angle_deg)`, `drive(speed_mm_s, turn_rate_dps)`, `stop(then=...)`, `settings(...)`, `use_gyro(True)` and non-blocking moves via `wait=False` + `done()`. Positive `turn` is right/clockwise viewed from above.

Give it any two closed-loop motors and it picks the right controller automatically:

- **Encoder servos** (`JGB37Motor`, `MG370Motor`) — the native C 2-DOF coupled controller at 1 kHz.
- **Serial-bus servos** (`ST3032Motor`, `ST3215Motor`) — the motors are *adopted* onto the hard-tick native bus engine (~220 Hz odometry per wheel, immune to Python stalls). Their wheel-mode motor API keeps working after adoption.
- **Open-loop motors** (`L298NMotor`) — kinematic `drive()` / `stop()` only; moves by distance need feedback and raise.

Example:

```
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

left  = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=6)
right = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=6,
                    invert=True)
db = DriveBase(left, right, wheel_diameter_mm=88,
               axle_track_mm=138)
db.straight(300)      # forward 300 mm
db.turn(90)           # turn right 90 degrees
```

Accurate `wheel_diameter_mm` / `axle_track_mm` values matter more than any controller gain — see *Measuring wheel diameter & axle track* for how to calibrate both in two short test drives.

settings(*straight_speed=None*, *turn_rate=None*, *acceleration=None*, *turn_acceleration=None*)

Tune cruise + ramp parameters for subsequent moves.

Parameters

- **straight_speed** – cruise speed for `straight()`, wheel-deg/s.
- **turn_rate** – cruise rate for `turn()`, wheel-deg/s.
- **acceleration** – STRAIGHT (and curve) trajectory acceleration, wheel-deg/s^2 — Pybricks' `straight_acceleration`. Serial-engine default 1500 (Pybricks parity: their 2000 dps^2 motor accel × 3/4) — lower it if the robot pitches or lifts its rear on launch. In mm/s^2 that's `acceleration * wheel_circumference / 360`. Applies on both paths: the native (encoder-servo) controller arms its C trajectory with it, and the serial-bus engine forwards it to the hard-tick controller.

- **turn_acceleration** – `turn()` ramps' own acceleration, wheel-deg/s^2, independent of `acceleration` — Pybricks parity (serial default 1500). Serial-bus drivebase only; passing it on an encoder/DC pair raises.

use_gyro(*enable*)

Switch the heading feedback source between encoder-diff (default) and the attached IMU (when True). Pybricks-style.

Requires an `imu=` argument to the constructor. With the gyro, heading is slip-immune — wheel slip or wildly asymmetric friction won't throw the robot off course, because the IMU sees actual body rotation regardless of what the wheels did. Works on both the native (encoder-servo) path and the serial-bus engine.

reset()

Re-zero the heading frame: after `reset()`, the robot's CURRENT pose is heading zero — for the drive base's controller and `imu.heading()` together (Pybricks `DriveBase.reset()`). Call it between moves; it raises while a move is active.

This is the supported way to re-zero mid-mission. `imu.reset_heading()` refuses while a drive base steers by the gyro, because zeroing the integrator under an armed controller shifts the measurement out from under the held target — the next `straight()` then veers chasing the old frame.

drive(*speed_mm_s*, *turn_rate_dps*)

Start driving at a given forward speed + body turn rate.

Kinematic one-shot — no coupled feedback. Call again (or `stop()`) to change. Positive turn rate = right turn (clockwise viewed from above), Pybricks convention.

Speed changes ramp at `settings(acceleration=...)` (the uniform-accel rule, 1.94.0) — proportionally across the two wheels, so an arc keeps its radius through the ramp.

check_motors()

Raise if a wheel has stopped answering the bus.

Serial-bus wheels are checked at construction and on every move, so you rarely need this directly — reach for it in a long-running open-loop control loop (`move_wheels` in a `while` loop already calls it for you), or to verify the chassis before a run. The error names the motor: side, bus id, slot, UART and pins.

move_wheels(*left_wheel_speed*, *right_wheel_speed*)

Drive the two wheels at independent speeds, in wheel-deg/s.

Positive is forward on both sides (each motor's `invert` is already applied), so `move_wheels(200, 200)` drives straight and `move_wheels(200, -200)` spins in place.

Non-blocking and continuous, like `drive()`: the wheels hold these speeds until you call it again, issue another move, or `stop()`. It supersedes any move in flight.

Use this instead of building a `SyncServoGroup` over the wheels. On serial-bus motors both setpoints leave in a single sync-write packet, so the wheels change speed at the same packet boundary — and a `SyncServoGroup` could not drive them anyway, because adopting them into a DriveBase hands their UART to the native driver. On encoder servos both targets are set and both servos subscribed inside one native call.

Where `drive(speed_mm_s, turn_rate_dps)` speaks chassis kinematics, this speaks wheels directly — the right tool for line-following, tank-style teleop, or any controller that

computes per-wheel outputs itself.

Example:

```
db.move_wheels(200, 120)      # gentle right-hand arc
time.sleep_ms(500)
db.stop()
```

Open-loop motor pairs (no encoder, no serial bus) are supported but cannot batch: the two speeds are written one after the other.

stop(*then='coast'*, *wait=False*)

Halt both wheels. Also clears any pending `wait=False` move (new command supersedes, pybricks-style). `then` selects the end-state:

- `"coast"` (default) — both motors free-wheel.
- `"brake"` — both motors actively resist motion at zero velocity.
- `"hold"` — both motors actively hold their current angle. Requires motors that implement `hold()` (e.g. `ST3215Motor`); open-loop drivers raise `NotImplementedError`.

Both wheels are always commanded together, never one motor at a time. On serial-bus (adopted) motors the whole stop is staged atomically in the C engine and reaches the wheels at the same bus-packet boundary. `brake` and `hold` DECELERATE at `settings(acceleration=...)` first (the uniform-accel rule) — hold anchors where the robot actually stops; `coast` releases torque immediately (a freewheel has no controlled deceleration). On encoder servos `coast` / `brake` likewise apply to both bridges inside one native call, so the second wheel's 1 kHz control tick can't keep driving while the first is already released.

Pybricks parity by default: the call returns immediately (their stop/brake do too — verified from their source). Since 2.4.0 short moves armed at speed raise their own deceleration to land at rest on target, so waiting is rarely needed; pass `wait=True` to BLOCK until both wheels' MEASURED speeds read ~0 — for `brake`/`hold` the decel ramp finishing plus settle, for `coast` the physical freewheel decay. `wait=True` raises `ValueError` on open-loop pairs (no measured speed to wait on) and `RuntimeError` if the wheels never settle within the timeout.

done()

Pybricks-style status check for in-flight `straight(wait=False)` / `turn(wait=False)`. Returns `True` if no move is pending or the active move has reached its target (and `stop(then=…)` has run). Returns `False` while the move is still progressing.

The controller runs the trajectory independently on the hard tick (native path: 1 kHz C scheduler; serial path: the st_bus pump); `done()` checks a flag — plus, on the serial path with the gyro enabled, feeds the IMU heading into the hard-tick heading hold. The natural polling cadence is `time.sleep_ms(10)`.

straight(*distance_mm*, *then='coast'*, *wait=True*)

Drive forward by `distance_mm`. 2-DOF coupled.

`then` is forwarded to `stop()` — see its docstring for coast/brake/hold semantics.

`wait=True` (default) blocks until the move completes. `wait=False` returns immediately after arming the move; the caller polls `done()` to check completion, and the `then=` dispatch is deferred until `done()` reports the target was reached. Concurrent use with another wait=False

move on a separate `DriveBase` (or with motor `run_angle(wait=False)` calls) is the intended pattern.

Any subsequent move command supersedes the previous pending wait=False move (pybricks "new command wins").

`then="continue"` (Pybricks `Stop.NONE`) does not decelerate at the end: the move finishes AT cruise speed and the wheels keep it until the next command — chain `straight`/`curve` segments without stopping between them. `"stop"` is accepted as an alias of the default coast end state.

Raises `RuntimeError` for open-loop motor pairs — moves by distance need feedback; use `drive()`/`stop()`.

turn(*angle_deg*, *then='coast'*, *wait=True*)

Turn in place by `angle_deg` body heading (positive = right/clockwise viewed from above, Pybricks convention).

Same `then` / `wait` semantics as `straight()` — see its docstring — except `then="continue"`: a turn in place ends facing its target heading, so there is no speed worth carrying.

curve(*radius*, *angle*, *then='coast'*, *wait=True*)

Drive an arc along a circle of `|radius|` mm, changing heading by `angle` degrees — Pybricks `DriveBase.curve()`, including the parameter names, so Pybricks-style keyword calls (`curve(radius=150, angle=90)`) work verbatim. The one deviation: our `then` defaults to `"coast"` like every openbricks move (Pybricks defaults to hold) — pass `then="hold"` for the Pybricks end state.

Positive `angle` turns right (clockwise from above, the system-wide sign convention, same as `turn()`); the SIGN of `radius` picks the travel direction along the arc (positive = forward, negative = backward). `curve(150, 90)` sweeps a forward quarter-circle to the right around a centre 150 mm to the robot's right; `curve(150, -90)` the mirror to the left.

The forward and turn profiles run simultaneously with proportional speed AND acceleration, so heading stays proportional to distance at every instant — the path is a true circle through the accel/decel ramps, not just at the endpoints. The centre speed is the `straight_speed` setting scaled by `|R| / (|R| + track/2)` so the OUTER wheel never exceeds `straight_speed`. `curve(0, angle)` degrades to a turn in place.

Same `then` / `wait` semantics as `straight()`, including `then="continue"` — the arc hands its full speed to the next command.

7.2 Motor drivers

Every motor class implements the same `Motor` contract (*Interfaces*), so higher layers — and your code — swap motor types without changes. Speeds are output-shaft degrees per second, angles are degrees, `dc()` duty is -100..100.

```
from openbricks.drivers.st3032 import ST3032Motor

m = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=6)
m.run_speed(120)          # wheel mode: spin at 120 deg/s
print(m.angle())          # multi-turn accumulated degrees
```

(continues on next page)

(continued from previous page)

```
m.coast()

from openbricks.drivers.jgb37_520 import JGB37Motor

e = JGB37Motor(in1=12, in2=14, pwm=27, encoder_a=18, encoder_b=19)
e.run_angle(360, 720)     # two turns at 360 deg/s, blocking
```

7.2.1 Serial bus servos (recommended)

ST-3032

ST-3032 (Feetech STS3032) serial bus servo.

Smaller sibling of the ST-3215 — same SCS protocol, same 4096-count 12-bit magnetic encoder, same operating modes (0 = position, 1 = wheel), same default 1 Mbps bus. The only differences are mechanical / electrical.

Full datasheet: `docs/datasheets/feetech_sts3032.pdf` (model ST-3032-C062, Edition A/0 2025-11-30). Headline specs at typical 12 V operation, all ±10% per Feetech:

```
Working voltage      9–14 V (typ 12 V)        ST-3215 typ 6–12.6 V
No-load speed        0.067 s/60° = 148 RPM = 888 °/s
                     (``ST3032Motor`` defaults ``max_dps`` to
                     this — the ST-3215's protective 600 default
                     capped the wire command well below what the
                     servo can do)
Stall torque         10 kg·cm (139 oz·in)     ST-3215 ~30 kg·cm
Stall current        1.6 A
Rated torque         3.3 kg·cm (1/3 of stall)
Rated current        500 mA
No-load current      100 mA
Idle current         20 mA
Kt                   6.3 kg·cm/A
Size                 23 × 12 × 27.5 mm        ST-3215 larger
Weight               20.6 g
Gear                 1/205, coreless motor
Backlash             ≤ 1.0°
Over-temp shutdown   80 °C
Over-current         > 80 % stall for 2 s
Max position update  1 ms
```

Because the wire protocol is identical, `ST3032` and `ST3032Motor` are thin marker subclasses of `ST3215` / `ST3215Motor` with no behavioural override. Use them when your bench actually carries an ST-3032 — the typed name documents the hardware and gives us a place to specialise defaults later (lower max_dps, torque caps) if hardware testing surfaces a real difference.

ST-3032 and ST-3215 instances on the same UART share the bus registry, so mixing them on one daisy chain is fine — each servo is addressed by its 1-byte ID regardless of model.

⚠ Voltage rail: an ST-3032 will brown out below ~9 V. Don't share a 6 V bus with ST-3215s; budget a separate 12 V rail.

class openbricks.drivers.st3032.**ST3032**(*servo_id*, *uart_id=1*, *tx=17*, *rx=16*, *baud=1_000_000*, *dir_pin=None*, *min_raw=0*, *max_raw=4095*, *range_deg=360*)

Bases: *ST3215*

One ST-3032 in position-servo mode (Servo interface).

Behaves identically to `ST3215` — see that class for the full API. Subclassed only to let user code spell the actual hardware in place.

class openbricks.drivers.st3032.**ST3032Motor**(*servo_id*, *uart_id=1*, *tx=17*, *rx=16*, *baud=1_000_000*, *dir_pin=None*, *invert=False*, *steps_per_dps=_DEFAULT_STEPS_PER_DPS*, *max_dps=ST3032_NO_LOAD_DPS*, *accel_dps2=1500.0*, *raise_on_stall=False*, *stall_idle_ms=1000*)

Bases: *ST3215Motor*

One ST-3032 in wheel/continuous-rotation mode (Motor interface).

Behaves identically to `ST3215Motor` — see that class for `run_speed` / `angle` / `run_angle` semantics — except for ONE specialised default: `max_dps` matches the ST-3032's datasheet no-load speed (888 °/s) instead of the ST-3215's 600, so the driver clamp can't cap the servo below its own spec.

STALL_TORQUE_MNM = 980.0

ST-3215

ST-3215 (Waveshare/FeeTech) serial bus servo.

These servos daisy-chain on a half-duplex UART bus: one TX line shared by host and all servos, each servo addressed by a 1-byte ID. Packet format (as in the FeeTech SCServo / Dynamixel-style protocol):

0xFF 0xFF ID LEN INSTR PARAM… CHECKSUM

CHECKSUM = ~(ID + LEN + INSTR + sum(PARAM)) & 0xFF LEN = number of params + 2

Common instructions:

```
0x01 PING            ->  probe the servo
0x02 READ            ->  READ  reg len
0x03 WRITE           ->  WRITE reg value...
```

Key registers (ST-3215):

```
0x21  Operation Mode — 0 = position, 1 = wheel/continuous
0x28  Torque Switch  — 1 = enable, 0 = coast
0x2A  Goal position (low, high) — int16, 0..4095 over ~360°
0x2E  Goal speed (low, high) — sign-magnitude; in wheel mode this
      is the velocity setpoint (bit 15 of high byte = direction)
0x38  Present position (low, high) — read only. 12-bit absolute
      angle within one revolution: 0..4095 over 360°. Wraps to 0
```

(continues on next page)

(continued from previous page)

```
    at every full turn; we accumulate multi-turn revolutions in
    software via a wrap heuristic in ``ST3215Motor.angle``.
```

Half-duplex wiring: most ST-3215 boards use a single data line driven by a TX/RX switching circuit, but MicroPython UART pins are usually separate. If your adapter exposes TX and RX separately, just wire them normally. If you have a true half-duplex bus, you'll need a driver chip or a direction-enable GPIO — set `dir_pin` below.

Two classes here:

- `ST3215` — position-mode (Servo interface), for grippers / lifts / sensor turrets. `move_to(angle)` blocks until reached.
- `ST3215Motor` — continuous-rotation mode (Motor interface), for drivebase wheels. Switches the servo into mode=1 at construction and exposes `run_speed(dps)` / `angle()` / `brake()` so it drops into `DriveBase` the same way `MG370Motor` does.

The ST-3032 (smaller sibling — 12 V, ~10 kg·cm, same SCS protocol) ships as marker subclasses in `openbricks.drivers.st3032`.

Only a minimal subset of the protocol is implemented here. PR welcome.

class openbricks.drivers.st3215.**ST3215**(*servo_id*, *uart_id=1*, *tx=17*, *rx=16*, *baud=1_000_000*, *dir_pin=None*, *min_raw=0*, *max_raw=4095*, *range_deg=360*)

Bases: *Servo*

One ST-3215 servo on a shared bus.

move_to(*angle_deg*, *speed=None*, *wait=True*)

Move to `angle_deg` within the configured position range.

`speed` (raw goal-speed register units) is optional; with `wait=True` polls until within 2% of target or a 3 s timeout.

angle()

Current shaft angle in degrees within the position range, or `None` if the bus read timed out.

ping()

`True` if the servo answers on the bus.

class openbricks.drivers.st3215.**ST3215Motor**(*servo_id*, *uart_id=1*, *tx=17*, *rx=16*, *baud=1_000_000*, *dir_pin=None*, *invert=False*, *steps_per_dps=_DEFAULT_STEPS_PER_DPS*, *max_dps=600.0*, *accel_dps2=1500.0*, *raise_on_stall=False*, *stall_idle_ms=1000*)

Bases: *Motor*

One ST-3215 in wheel/continuous-rotation mode.

Implements the openbricks `Motor` interface so it drops directly into `DriveBase`. The servo's internal velocity loop handles closed-loop speed tracking — we just write the setpoint and read the multi-turn accumulated angle.

`angle()` accumulates in software because the servo's Present-Position register is a 16-bit signed counter that wraps every ~3 turns (4096 counts/rev × ~8 turns to wrap at ±32767). Same wrap-correction shape as `PCNTEncoder`.

STALL_TORQUE_MNM = 2940.0

STALL_LOAD_PCT = 80

STALL_SPEED_DPS = 20.0

dc(*duty*)

True Pybricks `Motor.dc()`: raw duty, no speed regulation. Switches the servo to its open-loop mode (2) and writes duty straight to the output stage — speed sags under load, exactly like a plain DC motor at fixed voltage.

On a motor adopted by the native engine the bus belongs to the hard tick, so duty still maps onto `run_speed` scaled to `max_dps` there (transitional until the engine grows its own duty drive).

speed()

Measured shaft speed in deg/s from the present-speed register (sign-magnitude, bit 15; steps/s scaled by `steps_per_dps`). Returns `None` if the bus is silent, matching `angle()`. On an adopted motor (native DriveBase) the value comes from the hard-tick pump's widened feedback read — no extra bus traffic.

load()

Estimated shaft torque in mNm — Pybricks `Motor.load()` shape. The present-load register reports 0.1 %-of-stall units (sign in bit 10, per the Feetech SCServo SDK); scaled by the model's datasheet stall torque (`STALL_TORQUE_MNM`), so treat it as an estimate, not a measurement. `None` if the bus is silent. On an adopted motor the value comes from the hard-tick pump's widened feedback read (user frame — the slot carries this motor's invert).

stalled()

`True` when the servo is pushing hard (load magnitude at least `STALL_LOAD_PCT` percent of stall) but barely moving (speed magnitude at most `STALL_SPEED_DPS`) — the Pybricks `Motor.stalled()` contract, from the servo's own feedback registers. Raises `OSError` if the bus is silent (a silent bus must not read as "not stalled").

run_speed(*deg_per_s*)

Set continuous wheel velocity in degrees per second.

brake()

Ramp to zero velocity and hold (servo's internal loop).

Deliberately RAMPED at `accel_dps2` like every other speed change (user decision, reverting 1.18.1's instant bypass): one uniform rule — the acceleration default governs all commanded speed transitions, brake included. The SAFETY stop is unaffected: the e-stop and `coast()` cut the torque register, which the ramp does not govern, and remain instant. A hard local stop without torque-off is `accel_dps2=0` at construction.

coast()

Disable torque — wheel free-wheels.

classmethod migrate_bus_to_native(*bus*, *skip=()*)

Move every motor still on `bus` onto native slots.

Called by DriveBase adoption right after it takes the UART: the wheels become slots 0/1 by their own path, and anything else that was sharing the MicroPython bus has to come across too or it is left talking into a closed UART.

hold()

Actively hold the current shaft angle so the position PID resists rotation. Subsequent `run_speed` / `brake` / `coast` calls transparently restore wheel mode.

On an adopted motor (native DriveBase) the hold is a per-slot position lock on the hard tick (st_move_core) — it corrects disturbances continuously at the bus feedback rate.

Holding never crosses a turn boundary (the shaft is meant to stay put), so the mechanism is chosen to avoid disturbing the servo's turn model:

- If this motor has already been switched to multi-turn step mode (a prior `run_angle`), hold with a zero-count step — step mode keeps holding its current target.
- Otherwise (e.g. a `DriveBase` wheel that has only ever run in velocity mode), hold in single-turn position mode with goal=present. This deliberately does NOT zero the angle-limit registers, so a wheel motor keeps its stock single-turn present-position reads and its odometry stays intact.

done()

Pybricks-style status check for an in-flight `run_angle(wait=False)` move. Returns `True` if no move is in flight (the normal case) or the active move has parked (its remaining-to-target register has counted down to within tolerance). Returns `False` while the move is still running.

Calling `done()` is what advances the move: each call reads the step-mode remaining register once. For a move larger than 7 turns (issued as back-to-back steps), `done()` writes the next step once the current one parks, and on the final step runs the end-of-move `then=` dispatch and clears the pending state. So polling cadence matters for >7-turn moves — the wheel sits idle at each step boundary until the next `done()` advances it. With a typical `time.sleep_ms(10)` poll that's a single-tick gap; if you never poll, a multi-step move stalls after the first step.

On an adopted motor the C controller advances the move on the hard tick; `done()` just checks the arrival flag and runs the deferred `then=` dispatch.

angle()

Return shaft angle in degrees, multi-turn accumulated.

In STEP mode the present-position register reads remaining-to- target rather than absolute position, so we can't derive the angle from it — return the software accumulator, which `run_angle` advances by the executed step counts as each step parks. In wheel/position mode, rebuild the accumulator from the encoder via the wrap heuristic.

reset_angle(*angle=0*)

Set the current shaft angle to `angle` (degrees).

run_angle(*deg_per_s*, *target_angle*, *wait=True*, *tolerance_deg=0.5*, *kp=None*, *poll_ms=None*, *debug=False*, *then='coast'*)

Rotate by `target_angle` degrees at up to `deg_per_s`, ending within `tolerance_deg` of the target.

On an adopted motor (native DriveBase) the move runs as a per-slot trapezoid position move on the hard tick (st_move_core) — same trajectory + arrival semantics as the drivebase's own moves; `tolerance_deg` is fixed at the C core's arrival tolerance there.

`target_angle` is RELATIVE and UNBOUNDED — `run_angle(200, 360)` rotates one full turn forward, `run_angle(200, 1080)` three turns, `run_angle(200, -540)` one and a half turns back. Direction is the sign of `target_angle`.

Implementation: the servo is driven in **step mode** (op_mode=3) for the move. In step mode the goal-position register is a *signed relative step* — write N counts and the shaft advances N counts from where it is, with no single-turn 0/4095 wrap to cross (the prerequisite is angle limits = 0, set once on the first call by `_ensure_step_limits`). This is what makes moves past 180° / past one full turn work: the older single-turn position mode (op_mode=0) clamps to 0..4095 and a target across the boundary was executed the *wrong way round*, capping real motion at roughly half a turn.

A single step write covers up to ±7 turns (the STS multi-turn envelope); larger moves are issued as back-to-back ±7-turn steps, each parking before the next — still no boundary to cross. The servo's internal PID handles convergence (≈0.088° per encoder count); completion is detected by reading the step-mode *remaining-to-target* register and waiting for it to count down to ~0 (see `_read_step_remaining`). The shaft-angle accumulator is advanced by the counts actually travelled so `angle()` / `reset_angle` stay correct across the move.

`then` selects the end-state, pybricks-style:

- `"coast"` (default) — cut torque; wheel free-wheels. The next `run_speed` / `brake` / `run_angle` transparently re-enables torque and restores the mode it needs.
- `"brake"` — restore wheel mode and write goal_speed=0 so the servo's velocity loop actively holds zero rotation rate.
- `"hold"` — leave the servo in step mode; its position PID is already holding the target and resisting rotation.

`wait=False` kicks off the move and returns immediately without blocking. Use it for concurrent multi-motor moves, pybricks-style:

```
left.run_angle(60, 720, wait=False)
right.run_angle(60, 720, wait=False)
while not (left.done() and right.done()):
    time.sleep_ms(10)
```

Multi-revolution targets are supported in `wait=False` mode too. For a move within ±7 turns the whole thing is one step write and `done()` simply reports convergence; for a larger move `done()` issues each subsequent ±7-turn step once the previous one parks, so you must keep polling. The end-state `then=` dispatch is deferred until `done()` reports the final step has converged.

Any subsequent motion command (`run`, `run_speed`, `brake`, `coast`, `hold`, `run_angle`) supersedes a pending `wait=False` move — the new command takes over and the pending state is dropped (pybricks "new command wins").

The legacy `kp` / `poll_ms` / `debug` arguments are accepted for back-compat with the velocity-mode implementation but no longer apply — the PID lives on the servo, not in Python.

ping()

`True` if the servo answers on the bus.

class openbricks.drivers.st3215.**SyncServoGroup**(*servos*)

Bases: `object`

Coordinated multi-servo writes via SCServo SYNC WRITE.

All servos must share one `_SCServoBus` (same UART). Mixed servo types (`ST3215`, `ST3215Motor`, `ST3032`, `ST3032Motor`) are fine since they all speak the same SCS protocol — SYNC WRITE just blasts the same register on all listed IDs.

Use this whenever you have multiple servos that should apply a setpoint at the same packet boundary (a multi-finger gripper, a multi-axis arm) — each servo receives its byte slot of the broadcast packet at the same instant, instead of N serialised individual writes.

NOT for drivebase wheels: use `DriveBase.move_wheels(left, right)` instead. It gives the same one-packet guarantee, and a SyncServoGroup cannot drive adopted wheels at all — a DriveBase hands their UART to the native bus driver, so the MicroPython bus this class writes through is closed.

Example

```
from openbricks.drivers.st3215 import ST3215Motor, SyncServoGroup

thumb = ST3215Motor(servo_id=1)
index = ST3215Motor(servo_id=2)
group = SyncServoGroup([thumb, index])

# Both fingers start moving at the same packet boundary —
# one SYNC WRITE instead of two individual writes.
group.set_goal_speeds([200, 200])
```

set_goal_speeds(*speeds_dps*)

Write goal-speed on every servo in one SYNC WRITE packet.

`speeds_dps` is a list parallel to the servos given at construction. Each servo's own `_encode_goal_speed` is used, so per-servo `invert` / `steps_per_dps` / `max_dps` are respected.

Servos that don't expose `_encode_goal_speed` (i.e. the position-mode `ST3215` class) raise `TypeError`.

7.2.2 DC gear motors with encoders

JGB37-520

JGB37-520 geared DC motor with magnetic quadrature encoder.

This is a 6-wire motor: two power leads (into an H-bridge, typically L298N), Vcc and GND for the hall sensors, and two encoder channels A and B.

Typical spec sheet values (varies by gear ratio variant):

- Encoder CPR at motor shaft: 11
- Gear ratio (example): 1:30 -> output shaft CPR = 11 * 30 * 4 = 1320 edges (multiply by 4 because we count both edges on both channels)

This class is a thin Python wrapper over the C `Servo` type in `_openbricks_native` (see `native/user_c_modules/openbricks/servo.c`). The wrapper's job is to build the hardware handles (Pin / PWM / encoder) from pin numbers and pass them to the native servo — the closed-loop control tick runs in C at the `motor_process` tick rate (1 kHz default).

class openbricks.drivers.jgb37_520.**JGB37Motor**(*in1*, *in2*, *pwm*, *encoder_a*, *encoder_b*, *counts_per_output_rev=1320*, *invert=False*, *encoder_invert=False*, *kp=_DEFAULT_KP*)

Bases: *Motor*

JGB37-520 DC gear motor with quadrature encoder, closed-loop.

Runs the full `Motor` contract on the native 1 kHz servo core: `run_speed(deg_per_s)`, `run_angle(deg_per_s, target_angle)`, `dc(duty)`, `brake()` / `coast()`, `angle()` / `reset_angle()`, `speed()`. Pair two of them in a *DriveBase* for the coupled 2-DOF chassis controller.

Parameters

- **in1** – H-bridge pins (e.g. L298N IN1/IN2/EN).
- **in2** – H-bridge pins (e.g. L298N IN1/IN2/EN).
- **pwm** – H-bridge pins (e.g. L298N IN1/IN2/EN).
- **encoder_a** – quadrature encoder channel pins.
- **encoder_b** – quadrature encoder channel pins.
- **counts_per_output_rev** – encoder edges per output-shaft revolution (4 × PPR × gear ratio; 1320 for the common 11-PPR 1:30 variant).
- **invert** – flip motor command AND encoder together — for a motor wired backwards end-to-end (typically the mirrored side of a drivebase).
- **encoder_invert** – flip ONLY the encoder reading — for mirror-mounted encoders that count down on motor-forward.
- **kp** – speed-loop proportional gain (native servo core).

Example:

```
from openbricks.drivers.jgb37_520 import JGB37Motor

m = JGB37Motor(in1=12, in2=14, pwm=27,
               encoder_a=18, encoder_b=19)
m.run_angle(360, 720)       # two turns at 360 deg/s, blocking
print(m.angle())
```

dc(*duty*)

Run at a fixed raw duty cycle (-100..100), open loop — Pybricks `Motor.dc()`. Non-blocking. This is the pre-1.21.0 `run()`.

speed()

Measured shaft speed in degrees per second — Pybricks `Motor.speed()`. Closed-loop drivers implement it (encoder observer / servo present-speed register).

brake()

Stop with active braking (both terminals shorted).

coast()

Stop by cutting drive power (motor free-wheels).

angle()

Return the current shaft angle in degrees.

reset_angle(*angle=0*)

Set the current angle to `angle` degrees.

run_speed(*deg_per_s*)

Enter closed-loop speed control with the given target (deg/s).

run_angle(*deg_per_s*, *target_angle*, *wait=True*, *accel_dps2=1500.0*)

Rotate by `target_angle` degrees, following a trapezoidal profile at up to `deg_per_s`.

The native servo builds the profile once and the scheduler samples it at 1 kHz, so acceleration / cruise / deceleration phases are smooth and the move stops cleanly at the target without overshoot (apart from small integration error).

With `wait=False` the scheduler keeps running the profile; the caller can poll `self._servo.is_done()` or eventually call `brake()` / `coast()`.

MG370

MG370 geared DC motor with GMR (giant magnetoresistance) quadrature encoder.

The GMR variant has a 500-PPR encoder on the motor shaft, giving massively more resolution than a standard Hall-effect encoder — at a 1:34 gearbox the output shaft sees `500 * 4 * 34.014 ≈ 68028 CPR`. That edge rate is too fast for the software `QuadratureEncoder` (Pin.irq() tops out around 5-10 kHz), so this driver uses the native `PCNTEncoder` — a C wrapper over the ESP32 PCNT peripheral — baked into the firmware image.

Every MG370Motor instance needs its own PCNT unit — ESP32 has 8, ESP32-S3 has 4. Pick unit=0 for the first motor, unit=1 for the second, etc.

Apart from the encoder layer, MG370Motor is identical to JGB37Motor: same native `Servo` underneath, same control tick, same closed-loop API.

class openbricks.drivers.mg370.**MG370Motor**(*in1*, *in2*, *pwm*, *encoder_a*, *encoder_b*, *pcnt_unit=0*, *pcnt_filter=1023*, *counts_per_output_rev=_DEFAULT_CPR*, *invert=False*, *encoder_invert=False*, *kp=_DEFAULT_KP*)

Bases: *Motor*

MG370 DC gear motor with high-resolution GMR encoder, closed-loop.

Same `Motor` contract and native 1 kHz servo core as *JGB37Motor*; the difference is the encoder path — the 500-PPR GMR encoder is counted by the ESP32 PCNT hardware peripheral (`pcnt_unit`), not GPIO interrupts, because its edge rate exceeds what `Pin.irq` can service.

Parameters

- **in1** – H-bridge pins.
- **in2** – H-bridge pins.
- **pwm** – H-bridge pins.
- **encoder_a** – encoder channel pins (into PCNT).
- **encoder_b** – encoder channel pins (into PCNT).

- **pcnt_unit** – PCNT peripheral unit, unique per motor (ESP32 has 8, ESP32-S3 has 4 — first motor 0, second 1, …).
- **pcnt_filter** – PCNT glitch filter in APB cycles (default 1023 ≈ 12.8 µs; lower it only for encoders faster than ~35 kHz edge rate).
- **counts_per_output_rev** – encoder edges per output-shaft revolution (default matches the 1:34 GMR variant).
- **kp** (*invert / encoder_invert /*) – as in `JGB37Motor`.

dc(*duty*)

Run at a fixed raw duty cycle (-100..100), open loop — Pybricks `Motor.dc()`. Non-blocking. This is the pre-1.21.0 `run()`.

speed()

Measured shaft speed in degrees per second — Pybricks `Motor.speed()`. Closed-loop drivers implement it (encoder observer / servo present-speed register).

brake()

Stop with active braking (both terminals shorted).

coast()

Stop by cutting drive power (motor free-wheels).

angle()

Return the current shaft angle in degrees.

reset_angle(*angle=0*)

Set the current angle to `angle` degrees.

run_speed(*deg_per_s*)

Hold a target speed (closed loop).

run_angle(*deg_per_s*, *target_angle*, *wait=True*, *accel_dps2=1500.0*)

Rotate by `target_angle` degrees at `deg_per_s`. Blocks if `wait`; otherwise returns immediately and the caller polls `done()` to advance the move and detect completion.

7.2.3 H-bridge drivers (open loop)

L298N

L298N H-bridge motor driver.

The L298N drives a brushed DC motor via two direction pins (IN1/IN2) and one PWM pin (EN-A or EN-B). It's open-loop: the class knows nothing about the physical motor speed or position. For closed-loop use, wrap one of these in `jgb37_520.JGB37Motor` (which adds an encoder).

Pinout recap for one channel:

```
IN1  = direction bit A
IN2  = direction bit B
PWM  = enable / speed (tie to VCC for 100%, PWM for speed control)

IN1  IN2  result
---  ---  ----------
```

(continues on next page)

(continued from previous page)

```
0    0   coast
0    1   reverse
1    0   forward
1    1   brake
```

class openbricks.drivers.l298n.**L298NMotor**(*in1*, *in2*, *pwm*, *invert=False*, *pwm_freq=_PWM_FREQ_HZ*)

Bases: *Motor*

Open-loop brushed DC motor on an L298N H-bridge channel.

No encoder, so only the open-loop subset of the `Motor` contract works: `dc(duty)`, `brake()`, `coast()`. Closed-loop methods (`run_speed`, `run_angle`, `angle`, `hold`) raise `NotImplementedError` — wrap the same H-bridge channel in *JGB37Motor* when the motor has an encoder.

dc(*duty*)

Duty is -100..100 — Pybricks `Motor.dc()`. (Open-loop H-bridge: there is no closed-loop `run(speed)` here; the inherited `run` surfaces run_speed's NotImplementedError.)

brake()

Short both terminals — active brake.

coast()

Cut drive entirely — motor spins freely.

TB6612FNG

TB6612FNG dual MOSFET H-bridge — re-exposes `L298NMotor` under the TB6612 name.

Both chips drive each motor channel with the same three signals:

IN1 — direction bit 1 IN2 — direction bit 2 PWM — speed (PWM duty cycle)

…so the openbricks driver is identical. TB6612 is generally the better commodity choice — MOSFETs instead of Darlingtons, ~0.3 V drop instead of ~1.8 V, 3.3 V-logic compatible directly from an ESP32 GPIO, and 3.2 A peak per channel vs L298N's 2 A.

Wiring difference to be aware of: TB6612 has a chip-level **STBY** pin that must be held high for either channel to operate. Tie it to 3.3 V on your breakout, or drive a spare GPIO high at boot — openbricks doesn't model STBY because most breakout modules already pull it up.

7.3 Sensor drivers

All sensors construct against a `machine.I2C` bus (or a *TCA9548A* mux channel, which quacks the same) except two: the GPIO-driven HC-SR04, and the ICM-45686 IMU, which wires over SPI so the firmware can read it inside the 1 kHz control tick.

```
from machine import I2C, Pin
from openbricks.drivers.icm45686 import ICM45686
from openbricks.drivers.tcs34725 import TCS34725
from openbricks.drivers.tca9548a import TCA9548A
```

(continues on next page)

(continued from previous page)

```
i2c = I2C(0, sda=Pin(15), scl=Pin(16), freq=400_000)  # ESP32-S3 pins
mux = TCA9548A(i2c)

color = TCS34725(mux[0])          # two same-address sensors ...
color2 = TCS34725(mux[1])         # ... on separate mux channels
imu = ICM45686(sck=12, mosi=13, miso=11, cs=17)   # SPI, not I2C

print(color.rgb())                # (r, g, b) each 0-255
print(color.ambient())            # clear channel, 0-100
print(imu.heading())              # degrees, CW-positive
```

7.3.1 ICM-45686 (raw IMU, hard-tick heading)

TDK ICM-45686 raw 6-axis IMU — the hard-tick heading source.

Unlike the BNO055 (fused heading over I2C, pumped from Python at ~50-100 Hz during moves), this part is read INSIDE the 1 kHz hard tick over SPI (~13 µs per burst): gyro-Z integrates into a continuous heading in C (`imu_yaw_core` — stationarity-gated bias learning, Pybricks-Prime architecture), and a gyro `DriveBase` consumes it every millisecond with no Python in the loop.

Wiring: 4 free GPIOs to the breakout's SPI pins (SCLK/MOSI(SDI)/ MISO(SDO)/CS) — the I2C bus, mux and color sensors are untouched. INT/FSYNC/CLKIN stay unwired: the hard tick polls, nothing interrupts.

Example:

```
from openbricks.drivers.icm45686 import ICM45686
from openbricks.drivers.st3032 import ST3032Motor
from openbricks.robotics import DriveBase

imu = ICM45686(sck=12, mosi=13, miso=11, cs=17)
left  = ST3032Motor(servo_id=2, uart_id=1, tx=14, rx=41, invert=True)
right = ST3032Motor(servo_id=1, uart_id=1, tx=14, rx=41)
db = DriveBase(left, right, wheel_diameter_mm=88,
               axle_track_mm=138, imu=imu)
db.use_gyro(True)      # heading now corrects at 1 kHz in C
```

Calibration: gyro bias learns automatically during stillness (fast at boot rest, slow tracking after). `save_calibration()` persists the learned bias to NVS; the next construction seeds from it (the pbio trick), so boot-and-immediately-run starts corrected.

class openbricks.drivers.icm45686.**ICM45686**(*sck*, *mosi*, *miso*, *cs*, *hz=8_000_000*, *mode=3*, *scale=-1.0*)

Bases: `object`

heading()

Continuous body heading in degrees, CW-positive — the hard-tick integrator's value. Same contract as `BNO055.heading()` except unwrapped (multi-turn) rather than [-180, 180); `DriveBase` accepts both.

reset_heading()

Zero the heading frame (calibration is kept).

Refused (`OSError` from the yaw binding) while a DriveBase steers by this gyro — Pybricks parity: "Can't reset heading while gyro in use". The controller's measured frame would shift under its held target and the next move veers chasing the difference (bench 2026-08-13: straight() after turn(-90) + reset_heading() pivoted left). Use `db.reset()` — it re-bases the controller and the heading together — or `use_gyro(False)` first.

gyro()

(x, y, z) rates in dps from the last hard-tick sample.

acceleration()

(x, y, z) in g from the last hard-tick sample.

calibrated()

True once the bias estimator has locked (robot has been still for ~0.5 s since boot, or a saved calibration was loaded).

save_calibration()

Persist the learned gyro bias to NVS so the next boot starts corrected instead of waiting for stillness.

stats()

(reads_ok, read_errors, configured) — hard-tick health.

7.3.2 BNO055 (IMU)

BNO055 — re-export of the native `BNO055` C type.

The implementation lives in `native/user_c_modules/openbricks/bno055.c` so the drivebase tick can read `imu.heading()` every ms (when `use_gyro=True`) without a Python frame on the hot path. This module exists so existing user imports stay the same:

from openbricks.drivers.bno055 import BNO055

The class itself is implemented in C (so the drivebase can read the heading on its 1 kHz tick); its Python-facing API:

class openbricks.drivers.bno055.**BNO055**(*i2c*, *address=0x28*)

Bosch BNO055 9-axis IMU in 6-DOF fusion mode (accelerometer + gyro; the magnetometer is deliberately unused — motor magnets and steel in floors bend the local field, and a drive robot only needs *relative* heading). Heading zeroes where the robot points at construction.

Parameters

- **i2c** – `machine.I2C` (or a mux channel).
- **address** – 0x28, or 0x29 on breakouts whose ADR pin straps high.

heading()

Body heading in degrees, wrapped to [-180, 180). CW-positive: turning right (clockwise viewed from above) increases it — compass and Pybricks convention. This is what `DriveBase(imu=...)` reads.

euler()

`(heading, roll, pitch)` tuple in degrees.

angular_velocity()

`(x, y, z)` gyro rates in deg/s.

acceleration()

`(x, y, z)` accelerometer in m/s^2.

7.3.3 TCS34725 (color)

AMS TCS34725 RGB + clear light-to-digital sensor.

The TCS34725 returns four 16-bit channels (clear, red, green, blue) over I2C at address 0x29. There's an onboard LED that we leave under user control — some breakout boards wire it to the LED pin on reset, others require GPIO control.

Reference: TCS34725 datasheet (AMS / ams-OSRAM), sections 2.4 and 3.

I2C command byte format (from datasheet):
bit 7 (CMD) = 1 (always for command byte) bits 6:5 (TYPE) = 01 (auto-increment) or 00 (single) bits 4:0 (ADDR) = register address

class openbricks.drivers.tcs34725.**TCS34725**(*i2c*, *address=_ADDR*, *integration_ms=2.4*, *gain=16*)

Bases: *ColorSensor*

RGB + clear color sensor, fixed at I2C address 0x29.

Implements the `ColorSensor` contract: `rgbc()` raw 16-bit channels, `reflection()` and the calibrated helpers built on it. Two or more on one robot need a *TCA9548A* mux (the address is not configurable).

raw()

Return the raw (clear, red, green, blue) 16-bit readings.

rgb()

Return `(r, g, b)` scaled to 0..255 using the clear channel.

Dividing by the clear channel normalizes for ambient brightness, so a white object reports roughly (255, 255, 255) at any light level within the sensor's range.

ambient()

Return clear-channel brightness scaled to 0..100.

100 means the clear ADC is saturated *for the configured integration time* — 1024 counts per 2.4 ms cycle, capped at 65535 (datasheet "MAX COUNT"). Scale non-linearly would be nicer but keep it simple.

7.3.4 QTR / QTRX (reflectance array)

Pololu QTR / QTRX reflectance sensor arrays (analog outputs).

A row of IR emitter/phototransistor pairs a few millimetres above the mat: dark line = high reading, light mat = low. Unlike a pair of colour sensors, the array gives a CONTINUOUS line position — a weighted centroid across all elements — so a follower steers on a real analog error instead of edge-crossings.

Wiring (QTRX-HD-15A on ESP32-S3): the analog outputs go to ADC1 pins (GPIO 1..10 — ADC2 fights the radios). Any subset of the array's channels works; pass the pins left-to-right as mounted, and set `pitch_mm` to the spacing of the channels you actually wired (4 mm for adjacent QTRX-HD channels, 8 mm if every other one). CTRL (emitter enable) may be tied high or given a pin.

Readings are ratiometric to whatever height and mat you mounted over, so the array MUST be calibrated once per session: sweep it across the line while `calibrate()` runs. Reading before calibration raises — an uncalibrated centroid is a plausible-looking wrong number.

Example (bench: channels 15..9 left-to-right on ADC1; GPIO 4/6 belong to the stop button and servo bus):

```
from openbricks.drivers.qtr import QTRArray, QTRChannel

line   = QTRArray(pins=(1, 2, 3, 7, 8, 9, 10), pitch_mm=4.0)
branch = QTRChannel(pin=5)          # array channel 1, far right
line.calibrate(duration_ms=3000)    # sweep across the line now
branch.calibrate(duration_ms=3000)
while True:
    pos = line.position()           # mm, +right of centre, or None
```

class openbricks.drivers.qtr.**QTRElement**(*value*, *threshold*)

Bases: `object`

One calibrated element reading, as returned by *QTRArray.read()*.

`value` is the 0 (mat) .. 1000 (line) int; `dark()` / `white()` answer per element exactly like *QTRChannel.dark()*. Deliberately NOT an int subclass: MicroPython cannot reflect-compare `int` against one, so `max()` over such readings raises on the hub while passing on the desktop — numeric code uses `.value` instead (`max(e.value for e in readings)`).

dark()

True when this element is over the line.

white()

True when this element is over the mat.

ambient()

Reflected brightness as 0 (black) .. 100 (white) — the Pybricks scale, inverted from `value`.

class openbricks.drivers.qtr.**QTRReading**(*elements*, *array*)

Bases: `object`

One snapshot of the whole array, as returned by *QTRArray.read()*.

Behaves like the list of *QTRElement* it holds (index, iterate, `len`) and carries every aggregate view of exactly this snapshot — so a control tick reads once and derives everything from the same coherent sample:

```
r = qtr.read()
r[0].dark()              # per element
r.position()             # global centroid, mm
r.left_edge_position()   # the line's left edge, mm
r.right_edge_position()  # the line's right edge, mm
r.edge_error()           # setpoint element's ambient vs 50
r.leftmost_position()    # fork clusters
r.rightmost_position()
r.dark_count()           # how many elements on the line
r.all_dark()             # whole array on the line
r.max()                  # brightest value (phantom gate)
```

values()

The plain 0..1000 ints, left to right.

max()

The brightest calibrated value in this snapshot — the follower's off-mat phantom gate.

dark_count()

all_dark()

Every element over the line — with the branch flag dark too, the whole crossing is under the robot (the follower's stop condition).

position()

left_edge_position()

right_edge_position()

edge_error()

leftmost_position()

rightmost_position()

class openbricks.drivers.qtr.**QTRArray**(*pins*, *pitch_mm=4.0*, *ctrl=None*, *dark_threshold=300*, *positions_mm=None*)

Bases: object

Analog QTR/QTRX reflectance array on ESP32 ADC pins.

Parameters

- **pins** – ADC-capable GPIO numbers, LEFT to RIGHT as mounted.
- **pitch_mm** – physical spacing between the wired channels.
- **ctrl** – optional emitter-control GPIO (QTRX CTRL). Driven high at construction (emitters on). None = tied high.
- **dark_threshold** – calibrated value (0..1000) above which an element counts as "over the line" for position() / dark_count().

calibrate(*duration_ms=3000*, *poll_ms=5*)

Learn each element's mat/line extremes.

Sweep the array across the line while this runs (rotate the robot, or slide it by hand). Extends any previous calibration rather than replacing it, so repeated calls refine.

save_calibration(*path*)

Persist the calibration to a file on the hub filesystem, so one sweep (examples/qtr_calibrate.py) serves every later run. The wiring is stored with it: a calibration is per- element min/max, so loading it onto different pins would silently mis-scale every reading.

load_calibration(*path*)

Load a calibration saved by *save_calibration()*.

Raises with the remedy when the file is missing (run the calibrate script), corrupt, or was recorded for DIFFERENT wiring. Calibration is height- and mat-dependent — resweep after remounting the array or changing mats.

read()

One calibrated snapshot: a *QTRReading* holding a *QTRElement* per array element, left to right — `r[i].value` 0 (mat) .. 1000 (line), `r[i].dark()` / `r[i].white()`, and the aggregate views (`r.position()`, `r.dark_count()`, …) computed from exactly this sample.

dark_count(*readings=None*)

How many elements are over the line — the intersection / stop-bar signal (a full-width bar darkens most of the array, a branch stub only one side).

position(*readings=None*)

Line centre in mm relative to the array centre; positive = line is to the RIGHT. `None` when no element sees the line — use *last_side()* to know which way it escaped.

left_edge_position(*readings=None*)

Millimetre position of the line's LEFT edge — the white→black boundary on the left side of the leftmost dark cluster, in the same frame as *position()* (positive = right of the array centre). `None` when nothing is dark.

The crossing is interpolated between the last white element and the first dark one: the point where the calibrated value passes `dark_threshold`. An edge follower keeps THIS at 0, so the robot straddles the boundary — half the array over mat, half over line — instead of centring on the line itself. That also makes line width irrelevant to steering.

When the leftmost element is itself dark the true edge is off-array to the left; the estimate saturates half a pitch beyond that element, so the error keeps its sign and magnitude instead of vanishing.

right_edge_position(*readings=None*)

Mirror of *left_edge_position()*: the line's RIGHT edge — the black→white boundary on the right side of the rightmost dark cluster, in the same mm frame. `None` when nothing is dark; a dark rightmost element saturates half a pitch off-array to the right.

A right-edge follower keeps THIS at 0 — the mirror-image track discipline of the left-edge follower, same sign convention (the P law is symmetric between the two).

leftmost_position(*readings=None*)

Centroid of the LEFTMOST contiguous dark cluster, in mm (same frame as *position()*), or `None` when nothing is dark.

At a branch the array sees TWO dark regions and the global centroid lands between them — steering into the gap. The leftmost cluster is the left line's own centre: a follower that must take the left fork steers on this while its branch flag says a second line is present. Computed the same way every tick, so switching between the two is jump-free.

rightmost_position(*readings=None*)

Mirror of *leftmost_position()*: the RIGHTMOST contiguous dark cluster's centre — the right fork, for a route policy that takes it.

LEFT_SETPOINT_MM = 0.0

RIGHT_SETPOINT_MM = 0.0

set_mode(*mode*)

Select the edge-following discipline: `"left"` holds the line's LEFT edge at `LEFT_SETPOINT_MM`, `"right"` the RIGHT edge at `RIGHT_SETPOINT_MM`. Takes effect on the next *edge_error()* — call it again any time to switch disciplines mid-run.

mode()

The selected discipline, or `None` before set_mode.

edge_error(*readings=None*)

Signed steering error for the discipline selected with *set_mode()*: how far the mode's setpoint element sits from the black/white boundary, as its ambient (0 black .. 100 white) referenced to 50. Zero exactly when that element straddles the edge; range -50 .. +50. Positive = steer right, same sign convention as *position()* — drifting onto the mat side pushes the error toward the line, and vice versa, in both modes.

last_side()

+1 if the line was last seen right of centre, -1 left, 0 if it has never been off-centre. The recovery hint for a follower that lost the line entirely.

emitters(*on*)

Drive the CTRL pin (no-op when CTRL is tied high).

class openbricks.drivers.qtr.**QTRChannel**(*pin*, *dark_threshold=300*)

Bases: *QTRArray*

One reflectance element with the array's calibrate/read contract — for a DETECTOR channel wired apart from the line cluster (a branch / marker flag).

Kept out of *QTRArray* on purpose: a flag element folded into the steering centroid would yank the position toward every marker it passes. Steer on the array; DECIDE on this.

Example (bench: QTRX channel 1, far right, on GPIO 9):

```
branch = QTRChannel(pin=9)
branch.calibrate(duration_ms=3000)    # same sweep as the array
if branch.dark():
    ...   # marker under the flag channel
```

value()

Calibrated reading, 0 (mat) .. 1000 (marker/line).

dark()

True when the element is over a marker/line.

white()

True when the element is over the mat.

class openbricks.drivers.qtr.**QTRLineSensor**(*dark_threshold=300*)

Bases: *QTRArray*

THE bench line sensor: one QTRX-HD-15A window of ten skip- pattern channels, pre-wired geometry included — construct it, pick a discipline, follow:

```
qtr = QTRLineSensor()
qtr.set_mode("left")              # or "right", any time
error = qtr.read().edge_error()
```

Wiring (detailed table in docs/hardware.md): QTRX channels 1, 3, 4, 5, 7, 9, 11, 12, 13, 15 left-to-right onto GPIO 1..10 in order — a 56 mm window at spacings 8/4/4/8/8/8/4/4/8 mm (the pattern is a palindrome, so board orientation only changes the channel LABELS, never the geometry). `"left"`

mode holds the line's LEFT edge under channel 4 (-16 mm); `"right"` holds the RIGHT edge under channel 12 (+16 mm) — both derived from the positions table, not repeated by hand. Different wiring: use *QTRArray* directly with your own pins/positions/setpoints.

```
PINS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

POSITIONS_MM = (-28.0, -20.0, -16.0, -12.0, -4.0, 4.0, 12.0, 16.0, 20.0,
28.0)

LEFT_SETPOINT_MM = -16.0

RIGHT_SETPOINT_MM = 16.0
```

7.3.5 HC-SR04 (ultrasonic distance)

HC-SR04 ultrasonic distance sensor.

The HC-SR04 has two pins: `trig` (input — we drive it) and `echo` (output — it pulls high for as long as the round-trip echo took). Sequence:

1. Drive `trig` high for ~10 µs.
2. Read the duration of the resulting pulse on `echo`.
3. distance_mm = pulse_us × speed_of_sound_mm_per_us / 2.

Speed of sound in air at room temperature is ~0.343 mm/µs, so a 60 cm target gives ~3.5 ms of echo pulse — well within the 30 ms default timeout.

This driver is pure Python — there's no closed-loop control on a range sensor, so the 1 kHz hot-path concern doesn't apply. The `machine.time_pulse_us` builtin does the actual measurement; typical call latency is ~1 ms (pulse round-trip) which is fine for the cold-path use cases (line-following, wall avoidance, mission "approach until N mm" loops at <100 Hz).

class openbricks.drivers.hcsr04.**HCSR04**(*trig*, *echo*, *timeout_us=_DEFAULT_TIMEOUT_US*)

Bases: *DistanceSensor*

HC-SR04 ultrasonic distance sensor.

Parameters

- **trig** – GPIO pin number wired to the sensor's `TRIG` pin.
- **echo** – GPIO pin number wired to `ECHO`.
- **timeout_us** – how long to wait for the echo. `time_pulse_us` returns `-1` past this; we map that to `-1` from *distance_mm()* to mean "no return".

distance_mm()

Return the round-trip distance to the nearest reflector ahead, in millimetres, or `-1` if no echo arrived inside `timeout_us`.

7.3.6 VL53L0X (laser distance)

ST VL53L0X laser time-of-flight distance sensor.

The VL53L0X talks I2C at default address 0x29. A measurement cycle:

1. Write 0x01 to `SYSRANGE_START` (register 0x00).

2. Poll `RESULT_INTERRUPT_STATUS` (0x13) bit 0 until set (typical 33 ms at 33 Hz default rate).
3. Read `RESULT_RANGE_STATUS + 10` (0x14 + 10 = 0x1E) as a 16-bit big-endian integer — the raw distance in millimetres.
4. Write 0x01 to `SYSTEM_INTERRUPT_CLEAR` (0x0B).

Out of range / blocked targets show as 8190 mm in the raw register; we surface that as `-1` to match the `DistanceSensor` contract.

This driver implements the minimum register sequence to get usable single-shot readings on a power-on-default VL53L0X. ST's reference API ships an extensive calibration / VHV / measurement-budget setup that improves accuracy on tuned hardware — that's a follow-up patch when we have boards to validate against. The default-init ranging here is what most off-the-shelf MicroPython VL53L0X drivers use, and is good for the WRO use cases (line-of-sight, 30–1500 mm).

class openbricks.drivers.vl53l0x.**VL53L0X**(*i2c*, *address=_DEFAULT_ADDR*, *timeout_ms=200*)

Bases: *DistanceSensor*

ST VL53L0X laser time-of-flight distance sensor.

Parameters

- **i2c** – a `machine.I2C` (or compatible) instance.
- **address** – 7-bit I2C address (default 0x29). XSHUT-strapping multiple sensors onto one bus requires assigning each a unique address before constructing — outside this driver's scope.
- **timeout_ms** – how long to poll for a measurement to finish. Default 200 ms; the chip is typically done in 33–50 ms.

distance_mm()

Distance ahead in millimetres; -1 if no echo / out of range.

7.3.7 VL53L1X (laser distance, long range)

ST VL53L1X laser time-of-flight distance sensor.

Successor to the VL53L0X with 4 m range (vs 2 m on the L0X) and a different register map. I2C default address 0x29 (same as L0X — XSHUT strapping needed if both share a bus).

The chip-ID lives at 16-bit register 0x010F and reads back 0xEACC for a VL53L1X (or 0xEBAA for VL53L4CD, an L1X-pin-compatible variant).

Like the L0X driver this module ships the minimum register sequence to get usable single-shot ranging on a power-on-default chip. ST's reference API does an extensive calibration / VHV / SPAD-array selection pass for tuned operation; that lives behind a separate `calibrate()` method we'll add when we have hardware to validate against.

Measurement cycle:

1. Write 0x40 to `SYSTEM_INTERRUPT_CLEAR` (16-bit reg 0x0086).
2. Write 0x40 to `SYSTEM_MODE_START` (16-bit reg 0x0087) — kicks a single-shot ranging.
3. Poll `GPIO_TIO_HV_STATUS` (16-bit reg 0x0031) bit 0 until clear.
4. Read `RESULT_FINAL_CROSSTALK_CORRECTED_RANGE_MM_SD0` (16-bit reg 0x0096) as a 16-bit big-endian value — distance in mm.

5. Write 0x01 to `SYSTEM_INTERRUPT_CLEAR` (16-bit reg 0x0086).

VL53L1X registers are 16-bit (vs L0X's 8-bit) — addresses go through two byte-swaps on the I2C wire. The driver wraps that in `_read_u8_16` / `_write_u8_16` helpers.

class openbricks.drivers.vl53l1x.**VL53L1X**(*i2c*, *address=_DEFAULT_ADDR*, *timeout_ms=200*)

Bases: *DistanceSensor*

ST VL53L1X laser time-of-flight distance sensor (4 m range).

Parameters

- **i2c** – a `machine.I2C` (or compatible) instance.
- **address** – 7-bit I2C address (default 0x29).
- **timeout_ms** – how long to poll for a measurement to finish. Default 200 ms; the chip is typically done in ~50 ms.

distance_mm()

Distance ahead in millimetres; -1 if no echo / out of range.

7.4 Display & bus drivers

7.4.1 SSD1306 (OLED display)

SSD1306 OLED driver — thin wrapper around `micropython-lib`'s `ssd1306.SSD1306_I2C`.

The micropython-lib driver is already battle-tested and subclasses `framebuf.FrameBuffer`. We re-expose its core methods (`text`, `pixel`, `fill`, `show`) explicitly and delegate the rest (`rect`, `line`, `hline`, `scroll`, `contrast`, …) via `__getattr__`. An extra `clear()` convenience erases the buffer and pushes a blank frame.

The `ssd1306` module is frozen into each board's firmware image (see `native/boards/*/manifest.py`), so `import` works out of the box on flashed hardware.

A Display is an external I2C component, not part of the hub — instantiate one directly wherever you want to draw text / pixels.

class openbricks.drivers.ssd1306.**SSD1306**(*i2c*, *addr=0x3C*, *width=128*, *height=64*)

Bases: `object`

128×64 (or 128×32) SSD1306 OLED on I2C.

text(*s*, *x*, *y*, *c=1*)

Draw string `s` with the 8x8 font at pixel (x, y); `c=1` lit, `c=0` dark. Buffered — call `show()`.

pixel(*x*, *y*, *c=None*)

Set pixel (x, y) to `c` (1 lit / 0 dark), or return its current value when `c` is None.

fill(*c*)

Fill the whole buffer with `c` (1 lit / 0 dark).

show()

Push the buffer to the panel (nothing appears until this).

clear()

Blank the display immediately (fill 0 + show).

7.4.2 WS2812 / WS2812B (RGB LED strip)

WS2812 / WS2812B RGB LED strip driver (NeoPixel protocol).

Targets the common addressable-LED modules — the 8-LED "stick" (WS2812B x8), rings, and cut-to-length strips — on a single data GPIO. The bit-banged 800 kHz protocol itself comes from MicroPython's built-in `neopixel` module; this wrapper adds what user code actually wants on top of it:

- **Brightness scaling.** Raw WS2812s at full duty are dazzling and hot; `brightness=0.2` is a comfortable indoor default (same convention as the hub's onboard status LED). Colors are stored unscaled and multiplied only at `show()`, so `strip[i]` reads back exactly what you assigned and changing `brightness` later re-scales everything on the next `show()`.
- **Buffered updates.** Item assignment only touches the buffer; `show()` pushes the whole strip in one wire transaction — an animation frame is N assignments + one `show()`, not N flickery writes. `fill()` / `clear()` are one-call conveniences that push immediately.

Usage:

```
from openbricks.drivers.ws2812 import WS2812

strip = WS2812(pin=21, n=8)          # WS2812B x8 stick
strip.fill((0, 60, 0))               # everything green (pushed)
strip[0] = (255, 0, 0)               # buffer only...
strip[7] = (0, 0, 255)
strip.show()                         # ...pushed together
```

Wiring the x8 stick: DIN → the data GPIO, 5V → 5 V supply, GND → common ground. The ESP32's 3.3 V data line is out of spec for a 5 V-supplied WS2812B (V_IH = 0.7 × VDD = 3.5 V) but works with virtually every module in practice; if you see glitches, power the stick from 3.3 V (fine for the small x8 boards) or add a level shifter.

class openbricks.drivers.ws2812.**WS2812**(*pin*, *n=8*, *brightness=0.2*)

Bases: `object`

A strip of `n` WS2812/WS2812B RGB LEDs on one data pin.

show()

Push the buffered colors to the strip (one transaction), applying the current `brightness` scale.

fill(*color*)

Set every LED to `color` (an `(r, g, b)` tuple) and push immediately.

clear()

All LEDs off, pushed immediately.

property brightness

Global brightness scale 0.0 – 1.0. Assigning re-scales the whole strip on the next `show()` (or right now via `show()` — colors are stored unscaled).

7.4.3 TCA9548A (I2C multiplexer)

TI TCA9548A 8-channel I2C multiplexer / switch.

Some I2C sensors have a fixed address with no way to change it — the TCS34725 colour sensor is stuck at `0x29`, for instance — so you cannot put two of them on one bus without an address collision. The

TCA9548A sits between the host and the sensors and fans one bus out to eight electrically-isolated channels (SD0/SC0 .. SD7/SC7); each channel can host its own copy of the same-address device.

The mux itself answers at `0x70` by default (`0x70`..``0x77`` via the A0/A1/A2 address pins). Channel selection is a single-byte write to that address: bit *N* enables channel *N*. Multiple bits enable several channels at once; `0x00` disables all of them.

The drop-in path is `mux[n]`: it returns an object that quacks like a `machine.I2C` bus but transparently selects channel `n` before every operation, so existing drivers construct against it unchanged:

```
from machine import I2C, Pin
from openbricks.drivers.tca9548a import TCA9548A
from openbricks.drivers.tcs34725 import TCS34725

i2c = I2C(0, sda=Pin(21), scl=Pin(22), freq=400_000)
mux = TCA9548A(i2c)
left  = TCS34725(mux[0])    # 0x29 on channel 0
right = TCS34725(mux[1])    # 0x29 on channel 1
```

Reference: TCA9548A datasheet (Texas Instruments), section 8.3.

class openbricks.drivers.tca9548a.**TCA9548A**(*i2c*, *address=_DEFAULT_ADDR*)

Bases: `object`

8-channel I2C multiplexer for same-address sensors.

Index it like a list: `mux[n]` returns an I2C-compatible handle for channel *n* (0-7) that selects the channel transparently before every transaction — pass it to any driver in place of the real `machine.I2C`.

Parameters

- **i2c** – the upstream `machine.I2C` bus the mux sits on.
- **address** – mux's own I2C address, 0x70-0x77 via A0/A1/A2 straps (default 0x70).

select(*channel*)

Enable exactly `channel` (0..7), disabling the rest.

disable()

Disable every channel (control byte `0x00`).

7.5 Interfaces

7.5.1 Core interfaces

Abstract interfaces for openbricks components.

MicroPython doesn't ship full `typing.Protocol` support, so these are plain base classes. Drivers should subclass the appropriate interface and fill in every method. The higher-level modules (`robotics`, `config`) only depend on these interfaces, never on concrete drivers — that's what makes the system plug-and-play.

If you add a new category of component (e.g. a distance sensor), add its interface here.

class openbricks.interfaces.**Motor**

Bases: object

A bidirectional motor.

Implementations range from an open-loop H-bridge driver (L298N) to a closed-loop geared motor with quadrature encoder (JGB37-520).

The method names and semantics follow the Pybricks Prime `Motor` API (1.21.0): `run(speed)` is degrees per second, closed loop; the raw-duty command is `dc(duty)`. Additional openbricks methods (`coast`, `run_speed`) remain as aliases.

Units

- `duty` is -100..100 (percent duty cycle, sign = direction).
- `speed` is degrees per second at the output shaft (closed-loop only).
- `angle` is degrees at the output shaft (closed-loop only).

run(*speed*)

Run at `speed` degrees per second, closed loop — Pybricks `Motor.run()`. Non-blocking. Concrete: delegates to `run_speed()` (the openbricks alias), so open-loop drivers surface its `NotImplementedError`.

BREAKING (1.21.0): before Pybricks parity this method took percent power. That command is now `dc(duty)` — a script still calling `run(30)` for power gets 30 deg/s instead (slow, not dangerous) or `NotImplementedError` on open-loop drivers.

dc(*duty*)

Run at a fixed raw duty cycle (-100..100), open loop — Pybricks `Motor.dc()`. Non-blocking. This is the pre-1.21.0 `run()`.

stop()

Stop and let the motor spin freely; it gradually stops from friction. Pybricks `Motor.stop()` semantics — the default of the three stop flavours (stop/brake/hold), and a concrete method: it delegates to `coast()`, so every driver gets it for free.

brake()

Stop with active braking (both terminals shorted).

coast()

Stop by cutting drive power (motor free-wheels).

hold()

Stop and actively hold the current shaft angle via closed-loop control. Only motors with position-mode hardware (e.g. ST-3215) or a software position loop implement this; open-loop drivers raise `NotImplementedError` — pick `brake` or `coast` instead.

angle()

Return the current shaft angle in degrees.

reset_angle(*angle=0*)

Set the current angle to `angle` degrees.

run_speed(*deg_per_s*)

Hold a target speed (closed loop).

run_angle(*deg_per_s*, *target_angle*, *wait=True*)

Rotate by `target_angle` degrees at `deg_per_s`. Blocks if `wait`; otherwise returns immediately and the caller polls `done()` to advance the move and detect completion.

done()

Return `True` if no non-blocking move is in flight or the active `run_angle(wait=False)` move has reached its target. Drivers that don't support non-blocking moves always return `True` (a wait=True call is finished before returning to the caller, by definition).

speed()

Measured shaft speed in degrees per second — Pybricks `Motor.speed()`. Closed-loop drivers implement it (encoder observer / servo present-speed register).

load()

Measured torque at the shaft in mNm — Pybricks `Motor.load()`. Drivers with load feedback (serial servos) implement it; the value is derived from the servo's load register and its datasheet stall torque, so treat it as an estimate.

stalled()

`True` when the motor is pushing as hard as it can but cannot reach its commanded speed — Pybricks `Motor.stalled()`. Drivers with load feedback implement it.

run_time(*speed*, *time_ms*, *then='hold'*, *wait=True*)

Run at `speed` deg/s for `time_ms` ms, then stop with the `then` flavour — Pybricks `Motor.run_time()`. `wait=False` is not supported (no background timer is allocated for it); pass `wait=True` or sequence it yourself.

run_target(*speed*, *target_angle*, *then='hold'*, *wait=True*)

Run to the ABSOLUTE `target_angle` (degrees, in the `reset_angle` frame) at up to `speed` deg/s — Pybricks `Motor.run_target()`. Built on the relative `run_angle`: the delta is measured from `angle()` at call time.

run_until_stalled(*speed*, *then='coast'*, *duty_limit=None*)

Run at `speed` deg/s until `stalled()`, apply the `then` flavour, and return the angle where it stalled — Pybricks `Motor.run_until_stalled()` (its default `then` is Stop.COAST).

`duty_limit` (percent, 0 < limit <= 100) caps the motor's torque for the duration of the run — the Pybricks gripper- homing pattern: drive gently into the end stop without crushing it. The cap is applied before the motion starts and restored afterwards, stall or not. Drivers opt in via `_duty_limit_push` / `_duty_limit_pop` (the ST3215/ ST3032 serial servos implement it as a temporary torque- limit register write; their stall detection scales to the cap).

class openbricks.interfaces.**Servo**

Bases: `object`

A position-controlled servo (angle-addressable).

move_to(*angle_deg*, *speed=None*, *wait=True*)

Move to absolute angle in degrees.

angle()

Read back the current angle.

class openbricks.interfaces.**IMU**

Bases: object

A 3-axis inertial measurement unit.

The expected unit convention is:

- heading/yaw/pitch/roll in degrees
- angular_velocity in degrees / second
- acceleration in m / s^2

heading()

Return heading (yaw) in degrees, wrapped to [-180, 180).

angular_velocity()

Return (wx, wy, wz) in deg/s.

acceleration()

Return (ax, ay, az) in m/s^2.

class openbricks.interfaces.**ColorSensor**

Bases: object

An RGB-ish color sensor.

rgb()

Return `(r, g, b)` each in 0..255.

ambient()

Return ambient / clear-channel intensity in 0..100.

7.5.2 Distance sensors

Distance-sensor interface, kept separate from `interfaces.py`.

Why split: `openbricks/__init__.py` eagerly imports the four core interfaces (Motor / Servo / IMU / ColorSensor) so that every `import openbricks` pays them. The observer's variance test runs close to the MicroPython heap ceiling, so adding even a small class to `interfaces.py` tips it over. Distance-sensor users explicitly import this module.

class openbricks.distance.**DistanceSensor**

Bases: object

A forward-facing range sensor (HC-SR04 / VL53L0X).

distance_mm()

Distance ahead in millimetres; -1 if no echo / out of range.

7.6 Hubs

Board-level peripherals — the onboard LED and buttons — behind one `Hub` object per supported devkit.

```
from openbricks.hub import ESP32S3DevkitHub

hub = ESP32S3DevkitHub()
```

(continues on next page)

(continued from previous page)

```
hub.led.rgb(0, 80, 0)          # onboard NeoPixel: dim green
if hub.bluetooth_button.pressed():
    print("button held at boot")
```

Hub — board-level peripherals (status LED, user button) for each supported MCU devkit.

Concrete hubs:

- `ESP32DevkitHub` — ESP32 DevKitC-V4: blue LED on GPIO 2.
- `ESP32S3DevkitHub` — ESP32-S3 DevKitC-1: onboard WS2812 RGB LED on GPIO 48.

Both hubs expect **two** momentary pushbuttons, each wired between a GPIO and GND:

- `bluetooth_button_pin` (classic ESP32 default **GPIO 5**; ESP32-S3 default **GPIO 38** — the S3 has only ten analog-capable pins, ADC1 = GPIO 1-10, and a button doesn't need one) — short-press toggles BLE on/off. Watched by *openbricks.bluetooth_button.BluetoothToggleButton* which the hub auto-wires when `bluetooth=True` (the default).
- The **program button** (default **GPIO 39**) — short-press starts or stops `/program.py`. Watched by *openbricks.launcher* directly, which the frozen default `main.py` starts via `launcher.run()`.

Two pins → no duration-based dispatch. Each button does one thing on every short press.

We deliberately avoid GPIO 0 (BOOT) for either role: holding it during reset puts the ESP32 into the ROM bootloader, so sharing that pin with an application button would turn an accidental power-glitch into a flash-mode drop.

Both hubs auto-wire the BLE toggle button by default (`bluetooth=True`): **constructing a hub** restores the persisted BLE state, installs a short-press handler on the `bluetooth_button_pin`, and — on the S3 — paints the WS2812 blue (BLE on) or yellow (BLE off). While a user program runs, the same LED **flashes** its state colour at 1 Hz as a "robot is running" indicator (plain on/off blink on the classic ESP32's single-colour LED), returning to the solid idle colour when the program stops. Put `hub = ESP32S3DevkitHub()` in your `main.py` to turn that on; omit the call (or pass `bluetooth=False`) to keep the button free for your own use.

External I2C components like SSD1306 OLEDs are **not** part of the hub — they're wired to any I2C bus the user chooses, instantiated directly from `openbricks.drivers.ssd1306` or similar, and used alongside the hub rather than through it.

class openbricks.hub.**StatusLED**

Bases: `object`

A user-visible status LED on the hub.

Plain single-colour LEDs implement `on` / `off`. Addressable RGB LEDs (WS2812 and friends) additionally implement `rgb`.

on()

Turn the LED on (RGB LEDs restore their last colour).

off()

Turn the LED off.

rgb(*r*, *g*, *b*)

Set colour (each channel 0..255). Raises on non-addressable LEDs.

class openbricks.hub.**Button**

Bases: object

A momentary pushbutton on the hub.

pressed()

Return True while the button is held down.

class openbricks.hub.**Hub**

Bases: object

Board-level peripherals baked into a specific MCU devkit.

led = None

bluetooth_button = None

class openbricks.hub.**SimpleLED**(*pin*, *active_high=True*)

Bases: *StatusLED*

Single-colour LED driven by one digital pin.

on()

Turn the LED on (RGB LEDs restore their last colour).

off()

Turn the LED off.

class openbricks.hub.**NeoPixelLED**(*pin*, *brightness=0.2*)

Bases: *StatusLED*

Single-pixel WS2812 / NeoPixel driver — the onboard LED on ESP32-S3 DevKitC-1 and most modern ESP32-S3 compact boards.

Unlike SimpleLED this one implements rgb(r, g, b) too, which is what the Bluetooth-toggle button uses for blue / yellow feedback.

brightness scales every channel on write (0.0 – 1.0). 0.2 is a comfortable indoor default — the raw WS2812 at full brightness is dazzling.

on()

Turn the LED on (RGB LEDs restore their last colour).

off()

Turn the LED off.

rgb(*r*, *g*, *b*)

Set colour (each channel 0..255). Raises on non-addressable LEDs.

class openbricks.hub.**PushButton**(*pin*, *active_low=True*)

Bases: *Button*

Digital pushbutton with configurable active level and internal pull.

pressed()

Return True while the button is held down.

class openbricks.hub.**ESP32DevkitHub**(*led_pin=2*, *bluetooth_button_pin=5*, *bluetooth=True*)

Bases: *Hub*

ESP32 DevKitC-V4 onboard hub: blue LED on GPIO 2, BLE-toggle button on GPIO 5.

`bluetooth` default True wires the button to a short-press BLE toggle and restores the persisted state at boot (see `openbricks.bluetooth_button` / `openbricks.bluetooth`). The onboard LED is single-colour so no colour feedback on toggle, but it still blinks at 1 Hz while a user program runs (dark at idle). Pass `bluetooth=False` to skip the wiring entirely, or `bluetooth_button_pin=<N>` to use a different GPIO. Wire the button between the chosen GPIO and GND; an internal pull-up is enabled.

class openbricks.hub.**ESP32S3DevkitHub**(*led_pin=48*, *bluetooth_button_pin=38*, *brightness=0.2*, *bluetooth=True*)

Bases: *Hub*

ESP32-S3 DevKitC-1 onboard hub: WS2812 RGB LED on GPIO 48, BLE-toggle button on GPIO 38.

`led_pin` defaults to 48 (the DevKitC-1 onboard WS2812). Pass `led_pin=None` to disable the LED entirely, or any other GPIO for a WS2812 wired elsewhere. `brightness` (0.0 – 1.0) scales channel values on write; default 0.2 is comfortable indoors.

`bluetooth_button_pin` defaults to 38 (was 5 until 1.66.3): the S3 has exactly ten analog-capable pins (ADC1, GPIO 1-10) and a button needs none of them — parking the default there cost an analog channel the moment a sensor array arrived. GPIO 38 is free, non-strapping and digital-only. Wire a momentary switch between GPIO 38 and GND; the pin is configured `Pin.IN` with `PULL_UP` so no external resistor is needed. Override via `bluetooth_button_pin=<N>` (a hub wired to the old default passes 5).

7.7 Tools

Small user-facing utilities mirroring `pybricks.tools`. Small utilities mirroring the `pybricks.tools` module.

openbricks.tools.**wait**(*milliseconds*)

Pause for `milliseconds` ms.

class openbricks.tools.**StopWatch**

Elapsed-time stopwatch in milliseconds.

time()

reset()

pause()

resume()

7.8 Runtime services

These modules run behind the scenes on the hub — the frozen `main.py` wires them up at boot. They're documented here because their behavior (button semantics, log rotation, BLE persistence) is user-visible.

7.8.1 Reserved-GPIO guard

Reserved-GPIO guard: fail pin misuse at construction time, loudly.

Drivers that take raw GPIO numbers call *check()* before touching `machine.Pin`. Two kinds of mistakes are caught:

- **Chip-reserved pins** — pins that can never work for user wiring on the running chip: nonexistent numbers, the SPI-flash pins, the ESP32-S3's native-USB pair, and (for output roles) the classic ESP32's input-only range. Claiming these either crashes the chip, bricks the USB connection, or fails somewhere far from the real mistake; the guard names the pin and the reason instead.
- **Pins the firmware runtime already owns** — the program button, the Bluetooth-toggle button, and the status LED register themselves via *claim()* when the hub / launcher wire them at boot. A driver constructed on one of those pins gets an error naming the owner ("in use as the program button") plus the constructor argument that moves the owner elsewhere.

Chip detection reads `os.uname().machine` and recognizes the two supported targets (ESP32, ESP32-S3). Off-chip — CPython tests, the MuJoCo sim, unix MicroPython — no chip is detected and the chip-reserved rules are skipped (the runtime-claims check still applies); pins there are fakes, not wiring. Tests pin a chip explicitly with *set_chip()*.

Strapping pins (0/3/45/46 on the S3) are deliberately *not* blocked: they work as regular GPIOs after boot and are routinely used. The wiring guides in `docs/hardware.md` cover the soft cases.

exception openbricks.pins.**ReservedPinError**

A GPIO was requested that can't (or shouldn't) be user-wired.

openbricks.pins.**set_chip**(*chip*)

Force the chip model the guard validates against.

`chip` is `"esp32"`, `"esp32s3"`, or `None` to return to autodetection. Used by tests; harmless elsewhere.

openbricks.pins.**claim**(*pin*, *role*, *hint=''*)

Record that the firmware runtime owns `pin` (e.g. a button).

openbricks.pins.**release**(*pin*)

Forget a claim — for callers that hand a pin back (e.g. after `hub.bluetooth_toggle.stop()`). Unknown pins are a no-op.

openbricks.pins.**check**(*pin*, *role*, *output=True*)

Validate that `pin` is usable for `role` on this chip.

Parameters

- **pin** – GPIO number the caller is about to wire.
- **role** – human-readable purpose ("L298N IN1", "HC-SR04 echo") — appears in the error message.
- **output** – the pin must drive (True) or only read (False). Only matters on the classic ESP32, where GPIO 34-39 are input-only.

Raises

ReservedPinError – with the pin, the role, and the reason.

7.8.2 Program launcher

Button-gated user-program launcher.

Pybricks-style workflow:

- `openbricks upload` stages a script at `/program.py` but does not run it — the user presses the program button to launch.
- `openbricks run` stages the same script and triggers the launcher immediately. Output streams back to the client; pressing the program button stops the program; when the program stops, the terminal exits.

Each press is a full press-release cycle. The program button has its own GPIO (default `39`), separate from the BLE-toggle button watched by *`openbricks.bluetooth_button`* (default `38`). Two pins → no duration-based dispatch — every press on the program pin means start-or-stop, and every press on the BLE pin means toggle-BLE.

Wiring:

- Press while idle → start `/program.py` (on release, with a post-stop lockout against bounce).
- Press while running → the stop fires on press-DOWN: the e-stop latch engages (motors halt + motion commands raise — see `openbricks.estop`), and a `KeyboardInterrupt` injection is requested and *retried* until the program is actually dead.

The watcher runs off a `machine.Timer` kept alive for the whole hub uptime (we never `deinit` it), so button-press-to-run survives `openbricks run` interrupting the main idle loop.

Typical `main.py` (the firmware ships a frozen default; users can override by writing to `/main.py` in VFS):

```
from openbricks import bluetooth, launcher bluetooth.apply_persisted_state() launcher.run()
# installs watcher + blocks on the idle loop
```

class openbricks.launcher.**Launcher**(*button*, *program_path=DEFAULT_PROGRAM_PATH*, *poll_ms=DEFAULT_POLL_MS*)

Shared state for the program-button watcher.

Tests instantiate this directly and drive `_tick` with a fake Pin; production code uses `_ensure_launcher()` below, which installs a singleton + `machine.Timer`.

START_LOCKOUT_MS = 500

STOP_RETRY_MS = 300

STARVE_NOTE_MS = 5000

DEBOUNCE_TICKS = 2

RUN_START_GRACE_MS = 400

START_PRESS_OPEN_MS = 600

RELEASE_CHATTER_MS = 500

note_external_stop()

Attribute a stop delivered OUTSIDE this watcher's own machinery — the hard-button path, or a REPL Ctrl-C.

The 1.48.2 start gates check state (*_lockout_until_ms*, press lifecycle) that only the watcher's stop path armed. When the hard path wins the race (stop in ~2 ms, before the watcher's next 50 ms tick), that state never arms: the stopping press's echoes — PCNT edges, the hard latch's post-disarm confirmation — read as fresh idle presses and dispatch a phantom start (bench 2026-08-03, second occurrence: gates in place, lockout never armed, next BLE session dead again). Called from the program teardown, so EVERY interrupt-unwound run arms the same suppression.

`openbricks.launcher.`**`dump_events`**`()`

Print the launcher button-event ring, oldest first.

`openbricks.launcher.`**`program_running`**`()`

True while a user program is executing — whatever started it (button press, `openbricks run`, or a scheduled start; all paths maintain the same flag).

This is the hub-wide "robot is running" signal. The BLE toggle watcher polls it from its 50 ms tick to flash the status LED while a program runs (`openbricks.bluetooth_button`), which is why it must stay cheap: one attribute read, no allocation.

`openbricks.launcher.`**`run`**`(`*program_path=DEFAULT_PROGRAM_PATH*, *button_pin=DEFAULT_BUTTON_PIN*, *poll_ms=DEFAULT_POLL_MS*, *timer_id=0*`)`

Install the button watcher and block on the cooperative drain loop. Called from the frozen `main.py`.

`timer_id=0` is the first ESP32-S3 hardware timer. The previous default `-1` (virtual timer) was supported by older MicroPython builds but raises `ValueError: invalid Timer number` on the v1.27+ MP we vendor — esp32-s3 only exposes hardware timers 0..3.

Intentionally blocks forever. If `openbricks run` later sends a Ctrl-C over the REPL to interrupt this loop, the Timer stays alive (we never `deinit` it) so subsequent `run_program` / button-press start continue to work.

`openbricks.launcher.`**`run_program`**`(`*program_path=DEFAULT_PROGRAM_PATH*`)`

Client-triggered entry for `openbricks run`.

Sets the `_running` flag, then exec's the program in the main thread (`_exec_program_raw` arms the native stop button for the duration). Propagates `KeyboardInterrupt` so the raw-REPL disconnect signals "stopped" back to the client (which then exits).

The program-boundary native-state wipe (motor_process callbacks + st_bus runtime) happens inside `_exec_program_raw`, shared with the button-press paths.

7.8.3 Run logs

Per-run log capture: tee `print(...)` output to a file on flash so untethered runs can be inspected later via `openbricks log`.

The launcher wraps every program execution with `log.session()` so the user's `print` output streams to *both* the live USB / BLE console (when one's listening) and a rotating file on flash. With nobody listening on the live channel, the file is the only record. `openbricks log` reads the most recent files back over BLE.

Storage layout:

```
/openbricks_logs/run_0.log
/openbricks_logs/slot_1.log
/openbricks_logs/slot_2.log
```

Each run gets the next index; the index lives in the file's header line ("<epoch> -- run_N --"), NOT the filename. The MAX_RUNS slot files are reused in place (run N overwrites slot_(N % MAX_RUNS)) — the earlier delete+create rotation churned littlefs directory metadata until every commit's allocator traversal cost ~400 ms (bench 2026-08-09); truncate-reuse keeps commits at fresh-filesystem cost. Flash usage stays bounded and indices grow monotonically, exactly as before.

Every line is prefixed with a raw int64 **UTC Unix epoch in milliseconds** (e.g. 1783950123456 left ambient: 33). No formatting, no timezone on the hub — the host CLI converts to the user's local time at display. The ESP32 RTC starts at 2000-01-01 on power-up; the CLI syncs it from the host clock on every connect, so runs started after any openbricks run / log / upload carry real wall-clock stamps (an unsynced run shows year-2000 dates, which is self-diagnosing).

The session is also bytes-capped: once a run's log file passes MAX_BYTES bytes, further writes are dropped from the file (the live console still gets them). This keeps a runaway while True: print(...) from filling the entire flash partition.

Implementation note: MicroPython doesn't expose sys.stdout as a re-bindable attribute on every port, so we tee at the builtins.print level instead. This catches every print(...) call — including ones with file=sys.stderr — but does not catch direct sys.stdout.write() calls. User code on the firmware path overwhelmingly goes through print(), so this trade-off is fine. The launcher additionally calls log.write_text(...) from its exception handler so tracebacks are captured.

openbricks.log.**note**(*text*)

Write one stamped line to the active run's log file.

No-op when no program is running (there is no file to write to). File only — the live console is deliberately not touched; callers that want a console message print one themselves. Used by the launcher so every button press that starts or stops a run leaves a timestamped entry in that run's log.

openbricks.log.**worst_write_ms**()

Slowest single filesystem call of the active session, in ms. 0 when nothing has been written or no session is active. The launcher includes this in its tick-starvation note: if the two numbers are of the same order, littlefs (block erase under a write) owns the starvation; if this stays small while the gaps are large, the blocker is elsewhere.

openbricks.log.**pump**()

Drain the active session's buffered output to its file.

Called from the launcher's Timer tick, which is what makes logging asynchronous: print appends to RAM and returns, this moves the bytes to flash off the program's hot path. No-op when no program is running. Never raises — the tick that calls this also owns the stop button.

openbricks.log.**flush**()

Commit the active session's buffered output NOW.

For the moments durability beats speed: the stop button firing, or any other point where the next thing that happens might be a reset. No-op when no program is running.

openbricks.log.**session**()

Construct a fresh _LogSession.

Use as a context manager:

```
with log.session() as sess:
    run_user_program()
    # sess.write_text(extra) for non-print output if needed.
```

openbricks.log.**list_runs**()

Return a list of `(index, full_path)` tuples, oldest first. Used by the on-hub helper that `openbricks log` invokes via raw-paste to enumerate available runs. Slot files are listed by their header index; legacy `run_N.log` files (pre-slot firmware) still appear until the next run's migration removes them.

openbricks.log.**read_run**(*index*)

Read a single run's log by index. Raises `OSError` if no such run exists (the slot was reused by a newer run, or the index never happened).

7.8.4 Bluetooth (BLE REPL & console)

Persistent Bluetooth on/off state for the hub.

The firmware ships with BLE compiled in (so `mpremote`-over-BLE code transfer works). At runtime we let the user toggle the advertising stack on and off; the choice is persisted in NVS so reboots don't lose the state. Default (no key in NVS yet) is **on**.

Typical usage from `main.py` right after the firmware boots:

from openbricks import bluetooth bluetooth.apply_persisted_state()

Then toggle programmatically (or from the hub's button — see `openbricks.hub.ESP32DevkitHub` once that integration lands):

bluetooth.toggle() # flip current state, persist, apply bluetooth.set_enabled(False) # explicit off

Activating BLE requires a hub name (`openbricks.HUB_NAME`) — flash the image with `scripts/flash_firmware.py --name NAME` first. We refuse to advertise with no name rather than defaulting to a shared value, since two hubs with the same advertising name can't be individually addressed.

`esp32.NVS` and `bluetooth.BLE` are imported lazily so desktop tests running under unix MicroPython don't need to have the firmware- only modules present — they install fakes in `tests/_fakes_ble.py`.

exception openbricks.bluetooth.**HubNameNotSetError**

Raised when the firmware was flashed without `--name NAME`.

openbricks.bluetooth.**is_enabled**()

Return the persisted flag (`True` if no value has ever been written).

openbricks.bluetooth.**set_enabled**(*enabled*)

Persist `enabled` and apply it to the BLE stack immediately.

Raises *HubNameNotSetError* when turning BLE on with no hub name flashed. Turning BLE off is always allowed — a nameless hub can still be silenced.

When enabling, also starts the NUS REPL bridge (`openbricks.ble_repl`) so `openbricks run` / `stop` can push scripts over BLE. When disabling, the bridge is torn down first so dupterm isn't left pointing at an inactive stack.

openbricks.bluetooth.**add_state_listener**(*callback*)

Register `callback(enabled)` to run after each state change. Adding the same callback twice is a no-op.

openbricks.bluetooth.**remove_state_listener**(*callback*)

Unregister a callback added with *add_state_listener()*. Unknown callbacks are a no-op.

openbricks.bluetooth.**toggle**()

Flip the current state. Returns the new (post-toggle) value.

openbricks.bluetooth.**apply_persisted_state**()

Read the persisted flag and apply it to the BLE stack.

Call this at boot (e.g. top of `main.py`) so the BLE radio reflects whatever the user last chose before the reboot — or comes up enabled on the very first boot.

Tolerant of a missing hub name: if the persisted state says BLE-on but no hub name is flashed (typical on a freshly-flashed chip before the first `openbricks flash --name ...` run), prints a one-line warning and skips activation. Better than raising `HubNameNotSetError` from `main.py`, which (per the 1.0.4 silent-boot bug) could lose its traceback to USB-Serial-JTAG timing and brick the boot.

Nordic UART Service bridge for the MicroPython REPL over BLE.

Vendored from upstream MicroPython's `examples/bluetooth/ble_uart_peripheral.py` (the `BLEUART` class) and `ble_uart_repl.py` (the `BLEUARTStream` dupterm adapter). Both files are MIT-licensed under MicroPython's project LICENSE. The one structural change is the advertising payload — upstream advertises only the device name + appearance; we add the NUS 128-bit service UUID so clients filtering by service can find us. Everything else (connection-set tracking, append-mode rx buffer, scheduled-flush write batching, `io.IOBase` inheritance) is the upstream pattern, unmodified.

Why vendored: writing this from scratch using only the docs (the 1.0.0-1.1.1 history) produced a long parade of "invisible until hardware" bugs — each fix required a hardware reflash + paste diagnostic round. Starting from upstream's working example would have caught all of them at once.

Public surface (what `openbricks.bluetooth.apply_persisted_state` calls):

- `start()` — bring up the NUS bridge. Idempotent.
- `stop()` — tear down. Idempotent.
- `is_running()` — query.

openbricks.ble_repl.**log_entries**()

The recorded events, oldest first (unwraps the ring).

openbricks.ble_repl.**dump_log**()

Print the in-memory BLE event log over USB-Serial-JTAG.

Why a helper rather than `print(_LOG)`: `print` routes through dupterm which routes through this module's stream, so a long log print adds entries to itself while running. Iterating + printing item by item is bounded; the bytes added during the print sit in _tx_buf until later and don't grow the log.

openbricks.ble_repl.**clear_log**()

openbricks.ble_repl.**is_running**()

openbricks.ble_repl.**pump_tx**()

Re-arm the TX flush if bytes are sitting in the buffer.

Liveness backstop, called from the launcher's poll tick. A flush chain dies in two ways: `micropython.schedule`'s queue was full at re-schedule time, or a notify failed and the retry is deliberately paced (see `_flush`). In both cases the buffered bytes wait for the *next* `write()` —

which never comes when the writer already finished. That was the `openbricks log` end-of-dump stall: the file content and the raw-REPL terminator sat in `_tx_buf` forever while the host timed out. The pump gives the chain a fresh start at tick cadence; it is idempotent and no-ops when the buffer is empty or the bridge is down.

`openbricks.ble_repl.`**`start`**`()`

Bring up the NUS bridge. Idempotent — second call is a no-op if already running.

Caller must have `ble.active(True)` before calling. Hub name comes from `openbricks.HUB_NAME` (NVS-backed); we refuse to advertise without one.

`openbricks.ble_repl.`**`stop`**`()`

Tear down the NUS bridge. Idempotent.

7.8.5 BLE toggle button

Short press on the BLE-toggle button toggles BLE on/off.

Wires a `machine.Timer`-driven poll loop (default 50 ms) against a `Button`-conformant object and calls `openbricks.bluetooth.toggle()` once per press-release cycle. State is persisted via NVS by the `bluetooth` module, so the new value survives reboots.

The same poll loop doubles as the **run indicator**: while a user program is executing (`launcher.program_running()`), the status LED flashes at 2 Hz instead of holding a solid colour — blue when BLE is on, yellow when it's off, plain on/off blinking on single-colour LEDs. When the program stops, the LED returns to its idle state (solid state colour on RGB hubs, dark on single-colour hubs). And it renders the **press acknowledgment**: every program-button press flashes the LED for a moment — red for the press that starts a run, green for the press that stops one (`notify_press()`, called by the launcher's press detectors).

This is a different physical button from the one `openbricks.launcher` watches for program start/stop — the BLE toggle lives on its own GPIO (default 5, see *openbricks.hub.Hub*) while the program button is on GPIO 4. Two pins → no duration-based dispatch, every press on this pin means "flip BLE".

Usage from `main.py`:

from openbricks import bluetooth from openbricks.bluetooth_button import BluetoothToggleButton from openbricks.hub import ESP32DevkitHub

bluetooth.apply_persisted_state() hub = ESP32DevkitHub() BluetoothToggleButton(hub.bluetooth_button).start()

The helper is deliberately standalone (not baked into `Hub`) so tests can exercise it in isolation, and so boards without a button — or users who want to drive the toggle from something other than a physical press — can skip it.

`openbricks.bluetooth_button.`**`notify_press`**`(`*`stop=False`*`)`

Record a program-button press. `stop=True` marks it as the press that stops a run (green flash); the default is a start press (red). The active *BluetoothToggleButton* renders it on its next poll tick. Safe from any context — it only sets two module variables.

`class` `openbricks.bluetooth_button.`**`BluetoothToggleButton`**`(`*`button`*`, `*`led=None`*`, `*`poll_ms=DEFAULT_POLL_MS`*`, `*`timer_id=1`*`, `*`color_on=DEFAULT_COLOR_ON`*`, `*`color_off=DEFAULT_COLOR_OFF`*`, `*`program_running=None`*`)`

Polls a button and toggles BLE on each press-release cycle.

Optional RGB LED feedback (blue = BLE on, yellow = off). Call `start()` to begin polling on a `machine.Timer`; `stop()` releases the timer. The toggled state persists across reboots.

start()

Begin polling. Safe to call repeatedly — the second call is a no-op.

On first call, paints the LED (if one was provided) to reflect the current persisted BLE state, and registers with `bluetooth.add_state_listener` so the LED also follows state changes made *without* the button — `bluetooth.set_enabled` from user code or a tool over the REPL.

stop()

Stop polling, release the timer, and unregister the LED state listener.

CHAPTER

EIGHT

ARCHITECTURE

A short tour of how `openbricks` is organized and why. If you've read Pybricks' `pbio` codebase, a lot of this will look familiar — the layering is borrowed directly, and for the same reason: openbricks ships as a **custom MicroPython firmware**, not a library you install on top of stock MicroPython. Pybricks does exactly this for LEGO hubs; we do it for commodity MCUs.

Owning the firmware shapes several decisions:

- Background control loops (`MotorProcess`) can run always-on off a hardware timer — nobody else is contending for that peripheral.
- Platform selection means picking which firmware image to flash, not runtime-dispatching between adapters.
- Hot control code can be compiled in as a native C extension later without a separate install step.
- We can extend or add `machine`-level primitives (custom timers, a hub abstraction) because we build the `machine` module.

8.1 Four layers

```
┌──────────────────────────────────────────────────────────┐
│  User code     (main.py, robotics.DriveBase, …)          │
├──────────────────────────────────────────────────────────┤
│  Abstract interfaces   (Motor, Servo, IMU, ColorSensor)  │
├──────────────────────────────────────────────────────────┤
│  Concrete drivers      (st3032, icm45686, tcs34725, …)   │
├──────────────────────────────────────────────────────────┤
│  MicroPython HAL       (machine.Pin, I2C, UART, PWM)     │
├──────────────────────────────────────────────────────────┤
│  openbricks firmware image — custom MicroPython build    │
│  for this specific MCU, with all the above baked in      │
└──────────────────────────────────────────────────────────┘
```

The two middle layers are what make this different from "a pile of MicroPython scripts." Interfaces (`openbricks/interfaces.py`) define the contract each family of component obeys; drivers implement that contract; everything above the interface line depends only on interfaces, not on specific chips. That's why swapping a JGB37-520 DC motor for an ST-3032 serial servo only changes the driver you instantiate — the `DriveBase` class asks for "a `Motor`" and doesn't know or care what's underneath.

This is the same split Pybricks has: `pbio/include/pbio/*.h` is the interface, `pbio/src/*.c` is the library, `pbio/drv/*` is the driver layer. We take the same approach in C — `native/user_c_modules/`

`openbricks/` holds the hot control code that runs at the scheduler tick rate. Targeted pbio-parity on control quality is the reason that code is C and not Python.

8.2 Pybricks-parity control, in C

All four of the big pbio control-quality items are ported and shipped in `native/user_c_modules/openbricks/`. Each corresponds to a pbio source file and keeps its structure close — `pbio` is MIT-licensed so the ports are direct where they can be.

1. **State observer** (pbio `observer.c`) — our `observer.c` is a two-state α-β filter. Less capable than pbio's full model-based observer (no motor model, no PWM coupling, no current/flux estimation) but a ~60× variance reduction over raw finite-differencing for little code. Upgrading to a model-based observer is later roadmap work.

2. **Trajectory planning** (pbio `trajectory.c`) — our `trajectory.c` computes trapezoidal (and triangular fall-through) speed profiles with explicit accel / cruise / decel phases. `servo.run_target()` and `DriveBase.straight()` / `.turn()` sample it each tick.

3. **Cooperative multitasking** (pbio `motor_process.c` + `os.c`) — our `motor_process.c`. Always-on 1 kHz tick off a `machine.Timer`. Native subscribers (`Servo`, `DriveBase`) register via a fast C-function-pointer path (~1 µs/tick); Python callables are still accepted on a slower dispatch path for user extensibility. Honesty note: on esp32, `machine.Timer` callbacks are dispatched through `micropython.schedule`'s bounded queue — a main thread blocked in one long C call delays or silently drops ticks, so 1 kHz is nominal, not guaranteed (unlike pbio, whose loop runs under the VM). Since 1.37.0 the controllers' **clock** is immune to that: on real hardware the tick advances `now_ms` by measured `mp_hal_ticks_ms` deltas (wall time, enabled by the frozen `boot.py`), so dropped ticks cost control updates but no longer dilate trajectory time. Since 1.38.0 the firmware also carries a **hard tick** (`native/patches/esp32-openbricks-hard-tick.patch`): a periodic C hook on the esp_timer service task, below the Python scheduler entirely — verified on hardware via `motor_process.hard_tick_selftest()` / `hard_tick_count()`. Existing controllers still dispatch through the scheduler (their encoder reads call into Python objects, which the hard context must never do); the serial-bus motor path lives there: since 1.45.0 `DriveBase` (`openbricks.robotics`) **adopts** serial-bus Motor objects transparently onto the hard-tick engine — it releases their `machine.UART` (explicit ownership handover, no peripheral double-claim) and runs the 2-DOF controller inside the hard tick on serial-bus servo slots (~220 Hz odometry per wheel, floor-verified 0.3% odometry closure on a square), with `use_gyro(True)` fed by the ICM-45686 read inside the hard tick itself (1 kHz heading correction, no Python in the loop; bench-verified +0.6° over a four-turn square) — or, for an I2C IMU like the BNO055, by a Python outer loop at ~50-100 Hz. Since 1.89.0 the wheels themselves run "dumb mode" by default: the servo is switched to its open-loop duty mode and the engine's own integer FF+PI closes the speed loop over raw duty sync-packets — every layer of the drive loop is openbricks code (`DriveBase(..., drive="wheel")` restores the servo's internal speed controller). There is exactly ONE drivebase class and NO Python control loop: on a runtime with neither the native bus nor the sim's emulated bus, constructing a serial-bus `DriveBase` raises instead of silently degrading. The sim emulates the `st_bus` surface (`_SimStBus` over MuJoCo wheels), so the same controller code path runs everywhere.

4. **Drivebase coupling** (pbio `drivebase.c`) — our `drivebase.c` runs two coupled controllers in (sum, diff) coordinates with position feedback on both. Exit criterion: asymmetric-friction test (one wheel at 0.9× commanded speed) keeps heading error under 5% of forward distance — the pure-Kp M1 fallback fails it.

8.3 Host tooling

Everything above describes what runs on the hub. There's a parallel host-side surface — a single PyPI package called `openbricks` that ships:

- A console CLI: `openbricks flash | list | run | upload | stop | log` for hub interaction over BLE / USB. See `tools/openbricks/openbricks_dev/`.
- A MuJoCo-backed simulator: `openbricks sim {preview, run}` opens a physics sim with the same C control cores as the firmware (`*_core.c` files compile into both targets, so the sim's hot-path math is byte-identical). Lives under `tools/openbricks/openbricks_sim/`. Optional via `pip install openbricks[sim]`.
- A driver shim that lets `from openbricks.drivers.st3032 import ST3032Motor` (and ST3215Motor / JGB37Motor / BNO055 / TCS34725 / HC-SR04 / VL53L0X / VL53L1X) run unchanged in MuJoCo — `openbricks sim run main.py` installs no-op `machine` fakes and replaces the I2C driver classes with sim-aware versions.
- Per-run log capture on the hub: every program execution tee'd to `/openbricks_logs/slot_N.log` (10 slot files reused in place, 64 KB each; run `N` overwrites slot `N % 10` and carries its run index in the file's header line — truncate-reuse instead of delete+create keeps littlefs commits at fresh-filesystem cost. Ten slots are enough that a few diagnostic `openbricks run -c` sessions don't rotate away the failing run they're investigating). Each line is prefixed with a raw UTC Unix-epoch-milliseconds stamp; `openbricks log -n NAME` reads the file back over BLE and renders the stamp as `[YYYY-MM-DD HH:MM:SS.mmm]` in your local timezone — useful for untethered runs where no live console was attached. The CLI syncs the hub's RTC from the host clock on every connect (run / upload / log), so runs started after any connect carry real wall-clock time; a hub that powered up and never saw a connect stamps from 2000-01-01, which is self-diagnosing. Button presses leave stamped entries too: a run's log opens with a header line — `started: button press | firmware 1.12.0 | program /program.py | uptime N ms | free N B` — and a stop press writes `button pressed -> stop` the moment it lands, followed by `estop engaged` and a final `stopped: KeyboardInterrupt (N ms after press, M retries)` debrief, so a misbehaving stop chain is diagnosable from the log alone. Since 1.44.0 the stop chain is also **bounded**: the program button is sampled on the hard tick (core 0) — a debounced press while a program runs fires the `KeyboardInterrupt` injection and the native-bus torque-off from C within ~2 ms, regardless of scheduler state (the Python watcher, which stays as defence in depth and as the classic-bus e-stop, measured gaps to 981 ms under load). An uncaught exception writes its **full traceback**, not just the exception's repr — on an untethered run the file is the only record, and a bare `OSError(19,)` doesn't say which call failed.
- Log writes are **asynchronous**. `print` only appends to a RAM buffer; the bytes reach flash from the launcher's Timer tick (`log.pump()`). A `flush()` on littlefs forces a metadata commit measured at ~60-90 ms on the ESP32 bench, and doing that per line ran synchronously on the main thread between the user program's own bytecodes — logging cost more than the work it logged and distorted the timing of whatever the robot was controlling. Real commits are paid where durability matters: program end, the stop button, and every `write_text` (the `started:` / `stopped:` / `Exception:` lines and button notes). So a hard reset can lose recent `print` output — never the run's framing. Measure it on your own hub with `openbricks run -n NAME examples/log_write_benchmark.py`.
- The **wired UART console is asynchronous too** (a build-time patch, `native/patches/esp32-uart-repl-tx-nonblocking.patch`). Upstream's UART stdout busy-waits until every byte has left the wire — ~5.1 ms for a typical line at 115200, paid by `print()` on the calling thread even with nothing attached to the UART pins. With the patch, `print` copies into a 2 KB ring and the

UART interrupt drains it in the background; a print storm deeper than the ring drops the remainder **on the wired console only** — BLE and the run log (both already asynchronous) keep every line. Net effect: a `print` costs string formatting plus three RAM buffer appends, ~1 ms, regardless of what is or isn't listening on any transport.

The Python module names on the host are deliberately split (`openbricks_dev` for the CLI, `openbricks_sim` for the sim) so they don't shadow the firmware-side `openbricks` package, which is sometimes imported on the host by the sim's driver shim.

8.4 Status

All foundational milestones are landed. Roadmap items completed:

- **M1** — always-on 1 kHz scheduler in C (`motor_process.c`).
- **M2** — observer + trajectory + servo state machine, all in C.
- **M3** — 2-DOF coupled drivebase in C, with optional gyro-feedback (`use_gyro(True)`) for slip-immune heading via an attached IMU.
- **M4** — `hub` abstraction (status LED, user button) + SSD1306 OLED. ESP32 + ESP32-S3 firmware images both build from the same codebase.
- **M5** — per-platform firmware images auto-published on every push to `main` (rolling `latest`) and on `v*` tags (versioned).

Sim phases (host-side): A (chassis + worlds) → B (shared C cores) → C (runtime + driver shim) → D (sensors + scenario reset / scoring) all landed. Phase E1 — pixel-accurate colour-sensor texture sampling — landed via CPU-side sampling: the sensor reads `model.tex_data` directly, computes UV from the geom-local hit point, and indexes the texel. No offscreen GL context, no platform divergence, works on macOS / Linux / Windows. Originally scoped as "Linux EGL headless rendering" but the EGL machinery is only needed for scenes with shadows / lighting / overlays over the textured plane — the WRO use case is a flat printed mat where the texture IS the answer.

Phase F (WRO 2026 RoboMission, 0.10.8 → 0.10.12) is feature-complete:

- **F1** — high-fidelity mat textures rasterised at 150 dpi (~14000×6750 px) from the official "Game Mat Printing File" PDFs. Drives Phase E1's sensor sampling against the real printed artwork. `scripts/regen-wro-mat-textures.sh` re-fetches and re-rasterises when WRO updates the source PDFs.
- **F2** — every visible LEGO prop in all three age categories (Elementary, Junior, Senior) modelled as LDraw assemblies. Per- prop `.ldr` files are the source of truth; `world.py` expands `<lego_prop ldr=".../*.ldr"/>` placeholders into MJCF bodies at load time via `openbricks_sim.lego_mjcf`. 13 LDraw part types in the registry today; new parts plug in by adding one `_PartSpec` entry. Senior also wires the WRO-published 3D-printed "mosaic frame" STL as a static MuJoCo `<mesh>`.
- **F3** — per-round randomization (WRO General Rules glossary "Robot Round" definition). Same seed → same layout. Specs are per-world tuples of `_RandomizationSpec` driven by one shared seeded RNG, so a Senior round shuffles all four cement colour groups deterministically from a single `seed=N`.
- **F4 + F5** — closed the F2 deferreds (mosaic frame mesh, dual- colour Senior barriers) and lifted Junior + Senior randomization slot coordinates from estimates to mat-extracted positions (same pixel-inspection flow Elementary used in 0.10.10).

Remaining in Phase E: broader worlds library, more example walkthroughs. EGL offscreen rendering would unlock simulation of scenes more complex than a printed mat (e.g. coloured 3D obstacles that cast shadows onto the colour sensor's view); not yet prioritised.

Upgrading the α-β observer to a pbio-style model-based observer (voltage/current coupling + motor model) is on the longer-term list — a precision lift we pick up once we have real hardware to measure against.

CHAPTER

NINE

BUILDING THE FIRMWARE

openbricks is a custom MicroPython firmware. Users flash the resulting image to their MCU; the openbricks Python package is frozen into the image and the motor-control hot path (scheduler, trajectory, observer, servo, drivebase) lives in a compiled C extension.

See `native/README.md` for the directory layout.

9.1 One-time setup

9.1.1 1. Initialise all submodules

```
git submodule update --init --recursive
```

The MicroPython source tracks a master commit under `native/micropython/`. Bumps should be deliberate and tested on real hardware — `motor_process.c` and the other native modules assume a specific MP ABI. The current pin is documented in `.gitmodules` / `native/micropython` HEAD.

9.1.2 2. Install the target toolchain

ESP32 / ESP32-S3 (current targets)

Install **ESP-IDF v5.5.4** (or newer in the 5.5 line). MicroPython master supports ESP-IDF 5.3 through 5.5; we recommend 5.5.4 because that's what CI builds against and what the maintainer tests on hardware.

```
mkdir -p ~/esp && cd ~/esp
git clone -b v5.5.4 --recursive https://github.com/espressif/esp-idf.git␣
↪esp-idf-v5.5
cd esp-idf-v5.5 && ./install.sh esp32,esp32s3
source export.sh          # sets $IDF_PATH; needs to happen in every shell
```

Verify:

```
echo $IDF_PATH       # /Users/<you>/esp/esp-idf-v5.5
idf.py --version     # ESP-IDF v5.5.4
```

> **Note.** If you're running an older MicroPython pin (e.g. v1.28.0), you'll need ESP-IDF v5.2.x — v1.28 pre-dates the MP commits that add 5.4+ support. Check `git -C native/micropython log --oneline -1` first.

9.2 Building

From the repo root:

```
./scripts/build_firmware.sh esp32      # original ESP32 (Xtensa LX6)
./scripts/build_firmware.sh esp32s3    # ESP32-S3  (Xtensa LX7, native USB)
```

The script checks `native/micropython` is populated and `$IDF_PATH` is set, then builds `mpy-cross` once and produces the image for the selected target with `BOARD=openbricks_<target>` and `USER_C_MODULES=$(pwd)/native/user_c_modules`.

Output tree: `native/micropython/ports/esp32/build-openbricks_<target>/`

File	Purpose
`firmware.bin`	complete flash image (everything below combined)
`bootloader/bootloader.bin`	second-stage bootloader
`partition_table/partition-table.bin`	partition map
`micropython.bin`	application partition only

9.3 Flashing

Use `openbricks flash`. It drives `esptool.py` to write the image and then writes the hub's BLE advertising name into NVS. The name is **per-hub**, set at flash time (not build time) — one firmware image is reused across every hub, and each hub gets its own identity here. `--name` is mandatory: two hubs that answer to the same name can't be individually addressed over BLE.

```
pipx install openbricks             # one-time; or `pip install openbricks`

openbricks flash \
    --name RobotA \
    --firmware native/micropython/ports/esp32/build-openbricks_esp32s3/
↪firmware.bin
```

`--port` is auto-detected when exactly one ESP device is connected; pass it explicitly with several devices attached. (`--firmware` is optional too — omitted, the newest **released** image for the detected chip is downloaded — but when flashing a local build like here, point it at your build output.)

The command erases flash, writes `firmware.bin` at the offset the image was built for — `0x0` on the S3, `0x1000` on the classic ESP32, detected from the partition-table position inside the image itself — waits for the device to boot, then pokes the name into `esp32.NVS("openbricks").hub_name` via `mpremote` and reads it back to verify. Cross-platform — works on macOS, Linux, Windows (use `COM5` etc. for `--port`).

If you'd rather use `esptool.py` directly (e.g. mass-flashing with a fixture, no name needed yet), the raw commands:

```
# Classic ESP32 — bootloader at 0x1000
esptool.py --chip esp32 --port /dev/tty.usbserial-XXXX erase_flash
esptool.py --chip esp32 --port /dev/tty.usbserial-XXXX --baud 460800 \
    write_flash -z 0x1000 openbricks-esp32-firmware-v0.9.0.bin
```

(continues on next page)

(continued from previous page)

```
# ESP32-S3 — bootloader at 0x0
esptool.py --chip esp32s3 --port /dev/tty.usbmodemXXXX erase_flash
esptool.py --chip esp32s3 --port /dev/tty.usbmodemXXXX --baud 460800 \
    write_flash -z 0x0 openbricks-esp32s3-firmware-v0.9.0.bin
```

A hub flashed this way boots fine but has no name; `openbricks.bluetooth.set_enabled(True)` will raise `HubNameNotSetError` until you write one via `mpremote`:

```
mpremote connect PORT exec "
import esp32
nvs = esp32.NVS('openbricks')
nvs.set_blob('hub_name', b'RobotA')
nvs.commit()
"
```

> **Common footgun.** Flashing the S3 `firmware.bin` at `0x1000` (classic-ESP32 offset) leaves `0x0..0xFFF` untouched; the S3 ROM boots from `0x0` and fails with `Invalid image block, can't boot`. Always match the offset to the chip.

Or, if you built with `idf.py`:

```
cd native/micropython/ports/esp32/build-openbricks_esp32
idf.py -p /dev/tty.usbserial-XXXX flash monitor
```

9.4 Verifying the image at the REPL

Connect over the USB UART (e.g. `mpremote connect /dev/tty.usbserial-XXXX`) and at the REPL:

```
from openbricks._native import motor_process, Servo, TrapezoidalProfile,␣
↪Observer, DriveBase
motor_process.is_running()    # False initially; True once a motor attaches
```

The openbricks Python package is frozen into the image, so `import openbricks.*` works without copying any files. The first time a closed-loop motor calls `run_speed`, it registers its control step with the scheduler and the 1 kHz timer ISR comes online.

9.5 Running tests

Tests exercise the real C module under the unix MicroPython binary — the same build as firmware, minus the ESP32 port:

```
make -C native/micropython/mpy-cross -j
make -C native/micropython/ports/unix \
    VARIANT=standard \
    USER_C_MODULES=$(pwd)/native/user_c_modules -j
./native/micropython/ports/unix/build-standard/micropython tests/run.py
```

The runner spawns one MP subprocess per test module for state isolation (the module list lives in `tests/run.py::_TEST_MODULES`). Expected final line: `all modules passed.`

9.6 CI

GitHub Actions runs several job groups on every push / PR (see `.github/workflows/ci.yaml`):

- **`test`** — builds the unix MP binary with the `_openbricks_native` user_c_module and runs `tests/run.py`. No ESP-IDF needed. This is the merge gate for firmware changes.
- **`cpython-tests` / `openbricks-py`** — the CPython-compatible subset of the firmware tests under stock CPython (catches MP/CPython interpreter divergence), the latter with coverage upload.
- **`openbricks-host`** — the host CLI + sim test suite (`tools/openbricks/tests`) with coverage, plus a `--help` smoke test of every subcommand and a headless preview of every shipped sim world.
- **`coverage`** — the firmware suite again on the gcov-instrumented unix MP build; uploads C-core coverage.
- **`firmware`** — a matrix job (targets: `esp32`, `esp32s3`) that builds each image inside the `espressif/idf:v5.5.4` container and uploads `firmware.bin` + bootloader + partition-table per target as a workflow artifact.
- **`qemu-smoke`** — boots the just-built ESP32-S3 image in Espressif's QEMU and asserts the bootloader reaches app entry with no panic markers.
- **`release`** / **`build-openbricks-sdist`** / **`build-openbricks-wheels`** / **`publish-openbricks`** — firmware GitHub Releases (rolling `latest` on main pushes, versioned on `v*` tags) and the PyPI sdist + cibuildwheel wheels published on `cli/v*` tags (releases up to 0.10.24 used `openbricks/v*`).

Successful PRs produce a flashable image downloadable from the Actions run.

9.7 Troubleshooting

"IDF_PATH is not set" — source `export.sh` in the shell that runs the build script. Each new shell needs it.

Build fails at `modbluetooth_nimble.c` — BLE (NimBLE) is *enabled* on both boards via `boards/sdkconfig.ble` in each board's `mpconfigboard.cmake` `SDKCONFIG_DEFAULTS` list; it's what `openbricks run` / `upload` / `stop` ride on, so don't disable it. A failure here usually means an ESP-IDF / MicroPython version mismatch — see the `WIFI_AUTH_MAX` entry below. (WiFi, by contrast, *is* deliberately disabled in `sdkconfig.board` + `mpconfigboard.h`.)

Build fails at `network_wlan.c` with a `_Static_assert` about `WIFI_AUTH_MAX` — ESP-IDF / MP version mismatch. Either bump ESP-IDF into the supported range (5.3–5.5 for current MP master) or bump the MP submodule.

Submodule is empty — `git submodule update --init --recursive`.

Link error "undefined reference to `mp_register_module__openbricks_native`" — the user_c_module isn't being built into the image. Check `./scripts/build_firmware.sh` is passing `USER_C_MODULES=$(pwd)/native/user_c_modules` to `idf.py`.

PYTHON MODULE INDEX

O

openbricks.ble_repl, 74
openbricks.bluetooth, 73
openbricks.bluetooth_button, 75
openbricks.distance, 65
openbricks.drivers.bno055, 52
openbricks.drivers.hcsr04, 58
openbricks.drivers.icm45686, 51
openbricks.drivers.jgb37_520, 46
openbricks.drivers.l298n, 49
openbricks.drivers.mg370, 48
openbricks.drivers.qtr, 53
openbricks.drivers.ssd1306, 60
openbricks.drivers.st3032, 40
openbricks.drivers.st3215, 41
openbricks.drivers.tb6612, 50
openbricks.drivers.tca9548a, 61
openbricks.drivers.tcs34725, 53
openbricks.drivers.vl53l0x, 58
openbricks.drivers.vl53l1x, 59
openbricks.drivers.ws2812, 61
openbricks.hub, 66
openbricks.interfaces, 62
openbricks.launcher, 70
openbricks.log, 71
openbricks.pins, 69
openbricks.robotics.drivebase, 35
openbricks.tools, 68

INDEX

A

acceleration() (*openbricks.drivers.bno055.BNO055 method*), 52
acceleration() (*openbricks.drivers.icm45686.ICM45686 method*), 52
acceleration() (*openbricks.interfaces.IMU method*), 65
add_state_listener() (*in module openbricks.bluetooth*), 73
all_dark() (*openbricks.drivers.qtr.QTRReading method*), 55
ambient() (*openbricks.drivers.qtr.QTRElement method*), 54
ambient() (*openbricks.drivers.tcs34725.TCS34725 method*), 53
ambient() (*openbricks.interfaces.ColorSensor method*), 65
angle() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 47
angle() (*openbricks.drivers.mg370.MG370Motor method*), 49
angle() (*openbricks.drivers.st3215.ST3215 method*), 42
angle() (*openbricks.drivers.st3215.ST3215Motor method*), 44
angle() (*openbricks.interfaces.Motor method*), 63
angle() (*openbricks.interfaces.Servo method*), 64
angular_velocity() (*openbricks.drivers.bno055.BNO055 method*), 52
angular_velocity() (*openbricks.interfaces.IMU method*), 65
apply_persisted_state() (*in module openbricks.bluetooth*), 74

B

bluetooth_button (*openbricks.hub.Hub attribute*), 67
BluetoothToggleButton (*class in openbricks.bluetooth_button*), 75
BNO055 (*class in openbricks.drivers.bno055*), 52
brake() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 47
brake() (*openbricks.drivers.l298n.L298NMotor method*), 50
brake() (*openbricks.drivers.mg370.MG370Motor method*), 49
brake() (*openbricks.drivers.st3215.ST3215Motor method*), 43
brake() (*openbricks.interfaces.Motor method*), 63
brightness (*openbricks.drivers.ws2812.WS2812 property*), 61
Button (*class in openbricks.hub*), 66

C

calibrate() (*openbricks.drivers.qtr.QTRArray method*), 55
calibrated() (*openbricks.drivers.icm45686.ICM45686 method*), 52
check() (*in module openbricks.pins*), 69
check_motors() (*openbricks.robotics.drivebase.DriveBase method*), 37
claim() (*in module openbricks.pins*), 69
clear() (*openbricks.drivers.ssd1306.SSD1306 method*), 60
clear() (*openbricks.drivers.ws2812.WS2812 method*), 61
clear_log() (*in module openbricks.ble_repl*), 74
coast() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 47
coast() (*openbricks.drivers.l298n.L298NMotor method*), 50
coast() (*openbricks.drivers.mg370.MG370Motor method*), 49
coast() (*openbricks.drivers.st3215.ST3215Motor method*), 43
coast() (*openbricks.interfaces.Motor method*), 63

ColorSensor (*class in openbricks.interfaces*), 65
curve() (*openbricks.robotics.drivebase.DriveBase method*), 39

D

dark() (*openbricks.drivers.qtr.QTRChannel method*), 57
dark() (*openbricks.drivers.qtr.QTRElement method*), 54
dark_count() (*openbricks.drivers.qtr.QTRArray method*), 56
dark_count() (*openbricks.drivers.qtr.QTRReading method*), 55
dc() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 47
dc() (*openbricks.drivers.l298n.L298NMotor method*), 50
dc() (*openbricks.drivers.mg370.MG370Motor method*), 49
dc() (*openbricks.drivers.st3215.ST3215Motor method*), 43
dc() (*openbricks.interfaces.Motor method*), 63
DEBOUNCE_TICKS (*openbricks.launcher.Launcher attribute*), 70
disable() (*openbricks.drivers.tca9548a.TCA9548A method*), 62
distance_mm() (*openbricks.distance.DistanceSensor method*), 65
distance_mm() (*openbricks.drivers.hcsr04.HCSR04 method*), 58
distance_mm() (*openbricks.drivers.vl53l0x.VL53L0X method*), 59
distance_mm() (*openbricks.drivers.vl53l1x.VL53L1X method*), 60
DistanceSensor (*class in openbricks.distance*), 65
done() (*openbricks.drivers.st3215.ST3215Motor method*), 44
done() (*openbricks.interfaces.Motor method*), 64
done() (*openbricks.robotics.drivebase.DriveBase method*), 38
drive() (*openbricks.robotics.drivebase.DriveBase method*), 37
DriveBase (*class in openbricks.robotics.drivebase*), 36
dump_events() (*in module openbricks.launcher*), 71
dump_log() (*in module openbricks.ble_repl*), 74

E

edge_error() (*openbricks.drivers.qtr.QTRArray method*), 57
edge_error() (*openbricks.drivers.qtr.QTRReading method*), 55
emitters() (*openbricks.drivers.qtr.QTRArray method*), 57
ESP32DevkitHub (*class in openbricks.hub*), 67
ESP32S3DevkitHub (*class in openbricks.hub*), 68
euler() (*openbricks.drivers.bno055.BNO055 method*), 52

F

fill() (*openbricks.drivers.ssd1306.SSD1306 method*), 60
fill() (*openbricks.drivers.ws2812.WS2812 method*), 61
flush() (*in module openbricks.log*), 72

G

gyro() (*openbricks.drivers.icm45686.ICM45686 method*), 52

H

HCSR04 (*class in openbricks.drivers.hcsr04*), 58
heading() (*openbricks.drivers.bno055.BNO055 method*), 52
heading() (*openbricks.drivers.icm45686.ICM45686 method*), 51
heading() (*openbricks.interfaces.IMU method*), 65
hold() (*openbricks.drivers.st3215.ST3215Motor method*), 44
hold() (*openbricks.interfaces.Motor method*), 63
Hub (*class in openbricks.hub*), 67
HubNameNotSetError, 73

I

ICM45686 (*class in openbricks.drivers.icm45686*), 51
IMU (*class in openbricks.interfaces*), 64
is_enabled() (*in module openbricks.bluetooth*), 73
is_running() (*in module openbricks.ble_repl*), 74

J

JGB37Motor (*class in openbricks.drivers.jgb37_520*), 46

L

L298NMotor (*class in openbricks.drivers.l298n*), 50
last_side() (*openbricks.drivers.qtr.QTRArray method*), 57
Launcher (*class in openbricks.launcher*), 70
led (*openbricks.hub.Hub attribute*), 67
left_edge_position() (*openbricks.drivers.qtr.QTRArray method*), 56
left_edge_position() (*openbricks.drivers.qtr.QTRReading method*), 55
LEFT_SETPOINT_MM (*openbricks.drivers.qtr.QTRArray attribute*), 56
LEFT_SETPOINT_MM (*openbricks.drivers.qtr.QTRLineSensor attribute*), 58
leftmost_position() (*openbricks.drivers.qtr.QTRArray method*), 56
leftmost_position() (*openbricks.drivers.qtr.QTRReading method*), 55
list_runs() (*in module openbricks.log*), 73
load() (*openbricks.drivers.st3215.ST3215Motor method*), 43
load() (*openbricks.interfaces.Motor method*), 64
load_calibration() (*openbricks.drivers.qtr.QTRArray method*), 55
log_entries() (*in module openbricks.ble_repl*), 74

M

max() (*openbricks.drivers.qtr.QTRReading method*), 55
MG370Motor (*class in openbricks.drivers.mg370*), 48
migrate_bus_to_native() (*openbricks.drivers.st3215.ST3215Motor class method*), 43
mode() (*openbricks.drivers.qtr.QTRArray method*), 57
module
    openbricks.ble_repl, 74
    openbricks.bluetooth, 73
    openbricks.bluetooth_button, 75
    openbricks.distance, 65
    openbricks.drivers.bno055, 52
    openbricks.drivers.hcsr04, 58
    openbricks.drivers.icm45686, 51
    openbricks.drivers.jgb37_520, 46
    openbricks.drivers.l298n, 49
    openbricks.drivers.mg370, 48
    openbricks.drivers.qtr, 53
    openbricks.drivers.ssd1306, 60
    openbricks.drivers.st3032, 40
    openbricks.drivers.st3215, 41
    openbricks.drivers.tb6612, 50
    openbricks.drivers.tca9548a, 61
    openbricks.drivers.tcs34725, 53
    openbricks.drivers.vl53l0x, 58
    openbricks.drivers.vl53l1x, 59
    openbricks.drivers.ws2812, 61
    openbricks.hub, 66
    openbricks.interfaces, 62
    openbricks.launcher, 70
    openbricks.log, 71
    openbricks.pins, 69
    openbricks.robotics.drivebase, 35
    openbricks.tools, 68
Motor (*class in openbricks.interfaces*), 62
move_to() (*openbricks.drivers.st3215.ST3215 method*), 42
move_to() (*openbricks.interfaces.Servo method*), 64
move_wheels() (*openbricks.robotics.drivebase.DriveBase method*), 37

N

NeoPixelLED (*class in openbricks.hub*), 67
note() (*in module openbricks.log*), 72
note_external_stop() (*openbricks.launcher.Launcher method*), 70
notify_press() (*in module openbricks.bluetooth_button*), 75

O

off() (*openbricks.hub.NeoPixelLED method*), 67
off() (*openbricks.hub.SimpleLED method*), 67
off() (*openbricks.hub.StatusLED method*), 66
on() (*openbricks.hub.NeoPixelLED method*), 67
on() (*openbricks.hub.SimpleLED method*), 67
on() (*openbricks.hub.StatusLED method*), 66
openbricks.ble_repl
    module, 74

openbricks.bluetooth
module, 73
openbricks.bluetooth_button
module, 75
openbricks.distance
module, 65
openbricks.drivers.bno055
module, 52
openbricks.drivers.hcsr04
module, 58
openbricks.drivers.icm45686
module, 51
openbricks.drivers.jgb37_520
module, 46
openbricks.drivers.l298n
module, 49
openbricks.drivers.mg370
module, 48
openbricks.drivers.qtr
module, 53
openbricks.drivers.ssd1306
module, 60
openbricks.drivers.st3032
module, 40
openbricks.drivers.st3215
module, 41
openbricks.drivers.tb6612
module, 50
openbricks.drivers.tca9548a
module, 61
openbricks.drivers.tcs34725
module, 53
openbricks.drivers.vl53l0x
module, 58
openbricks.drivers.vl53l1x
module, 59
openbricks.drivers.ws2812
module, 61
openbricks.hub
module, 66
openbricks.interfaces
module, 62
openbricks.launcher
module, 70
openbricks.log
module, 71
openbricks.pins
module, 69
openbricks.robotics.drivebase
module, 35
openbricks.tools
module, 68

P

pause() (*openbricks.tools.StopWatch method*), 68
ping() (*openbricks.drivers.st3215.ST3215 method*), 42
ping() (*openbricks.drivers.st3215.ST3215Motor method*), 45
PINS (*openbricks.drivers.qtr.QTRLineSensor attribute*), 58
pixel() (*openbricks.drivers.ssd1306.SSD1306 method*), 60
position() (*openbricks.drivers.qtr.QTRArray method*), 56
position() (*openbricks.drivers.qtr.QTRReading method*), 55
POSITIONS_MM (*openbricks.drivers.qtr.QTRLineSensor attribute*), 58
pressed() (*openbricks.hub.Button method*), 67
pressed() (*openbricks.hub.PushButton method*), 67
program_running() (*in module openbricks.launcher*), 71
pump() (*in module openbricks.log*), 72
pump_tx() (*in module openbricks.ble_repl*), 74
PushButton (*class in openbricks.hub*), 67

Q

QTRArray (*class in openbricks.drivers.qtr*), 55
QTRChannel (*class in openbricks.drivers.qtr*), 57
QTRElement (*class in openbricks.drivers.qtr*), 54
QTRLineSensor (*class in openbricks.drivers.qtr*), 57
QTRReading (*class in openbricks.drivers.qtr*), 54

R

raw() (*openbricks.drivers.tcs34725.TCS34725 method*), 53
read() (*openbricks.drivers.qtr.QTRArray method*), 55
read_run() (*in module openbricks.log*), 73
release() (*in module openbricks.pins*), 69
RELEASE_CHATTER_MS (*openbricks.launcher.Launcher attribute*), 70
remove_state_listener() (*in module openbricks.bluetooth*), 74
ReservedPinError, 69
reset() (*openbricks.robotics.drivebase.DriveBase method*), 37
reset() (*openbricks.tools.StopWatch method*), 68

reset_angle() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 48
reset_angle() (*openbricks.drivers.mg370.MG370Motor method*), 49
reset_angle() (*openbricks.drivers.st3215.ST3215Motor method*), 44
reset_angle() (*openbricks.interfaces.Motor method*), 63
reset_heading() (*openbricks.drivers.icm45686.ICM45686 method*), 51
resume() (*openbricks.tools.StopWatch method*), 68
rgb() (*openbricks.drivers.tcs34725.TCS34725 method*), 53
rgb() (*openbricks.hub.NeoPixelLED method*), 67
rgb() (*openbricks.hub.StatusLED method*), 66
rgb() (*openbricks.interfaces.ColorSensor method*), 65
right_edge_position() (*openbricks.drivers.qtr.QTRArray method*), 56
right_edge_position() (*openbricks.drivers.qtr.QTRReading method*), 55
RIGHT_SETPOINT_MM (*openbricks.drivers.qtr.QTRArray attribute*), 56
RIGHT_SETPOINT_MM (*openbricks.drivers.qtr.QTRLineSensor attribute*), 58
rightmost_position() (*openbricks.drivers.qtr.QTRArray method*), 56
rightmost_position() (*openbricks.drivers.qtr.QTRReading method*), 55
run() (*in module openbricks.launcher*), 71
run() (*openbricks.interfaces.Motor method*), 63
run_angle() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 48
run_angle() (*openbricks.drivers.mg370.MG370Motor method*), 49
run_angle() (*openbricks.drivers.st3215.ST3215Motor method*), 44
run_angle() (*openbricks.interfaces.Motor method*), 64
run_program() (*in module openbricks.launcher*), 71
run_speed() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 48
run_speed() (*openbricks.drivers.mg370.MG370Motor method*), 49
run_speed() (*openbricks.drivers.st3215.ST3215Motor method*), 43
run_speed() (*openbricks.interfaces.Motor method*), 63
RUN_START_GRACE_MS (*openbricks.launcher.Launcher attribute*), 70
run_target() (*openbricks.interfaces.Motor method*), 64
run_time() (*openbricks.interfaces.Motor method*), 64
run_until_stalled() (*openbricks.interfaces.Motor method*), 64

S

save_calibration() (*openbricks.drivers.icm45686.ICM45686 method*), 52
save_calibration() (*openbricks.drivers.qtr.QTRArray method*), 55
select() (*openbricks.drivers.tca9548a.TCA9548A method*), 62
Servo (*class in openbricks.interfaces*), 64
session() (*in module openbricks.log*), 72
set_chip() (*in module openbricks.pins*), 69
set_enabled() (*in module openbricks.bluetooth*), 73
set_goal_speeds() (*openbricks.drivers.st3215.SyncServoGroup method*), 46
set_mode() (*openbricks.drivers.qtr.QTRArray method*), 56
settings() (*openbricks.robotics.drivebase.DriveBase method*), 36
show() (*openbricks.drivers.ssd1306.SSD1306 method*), 60
show() (*openbricks.drivers.ws2812.WS2812 method*), 61

SimpleLED (*class in openbricks.hub*), 67
speed() (*openbricks.drivers.jgb37_520.JGB37Motor method*), 47
speed() (*openbricks.drivers.mg370.MG370Motor method*), 49
speed() (*openbricks.drivers.st3215.ST3215Motor method*), 43
speed() (*openbricks.interfaces.Motor method*), 64
SSD1306 (*class in openbricks.drivers.ssd1306*), 60
ST3032 (*class in openbricks.drivers.st3032*), 40
ST3032Motor (*class in openbricks.drivers.st3032*), 41
ST3215 (*class in openbricks.drivers.st3215*), 42
ST3215Motor (*class in openbricks.drivers.st3215*), 42
STALL_LOAD_PCT (*openbricks.drivers.st3215.ST3215Motor attribute*), 43
STALL_SPEED_DPS (*openbricks.drivers.st3215.ST3215Motor attribute*), 43
STALL_TORQUE_MNM (*openbricks.drivers.st3032.ST3032Motor attribute*), 41
STALL_TORQUE_MNM (*openbricks.drivers.st3215.ST3215Motor attribute*), 43
stalled() (*openbricks.drivers.st3215.ST3215Motor method*), 43
stalled() (*openbricks.interfaces.Motor method*), 64
start() (*in module openbricks.ble_repl*), 75
start() (*openbricks.bluetooth_button.BluetoothToggleButton method*), 76
START_LOCKOUT_MS (*openbricks.launcher.Launcher attribute*), 70
START_PRESS_OPEN_MS (*openbricks.launcher.Launcher attribute*), 70
STARVE_NOTE_MS (*openbricks.launcher.Launcher attribute*), 70
stats() (*openbricks.drivers.icm45686.ICM45686 method*), 52
StatusLED (*class in openbricks.hub*), 66
stop() (*in module openbricks.ble_repl*), 75
stop() (*openbricks.bluetooth_button.BluetoothToggleButton method*), 76
stop() (*openbricks.interfaces.Motor method*), 63
stop() (*openbricks.robotics.drivebase.DriveBase method*), 38
STOP_RETRY_MS (*openbricks.launcher.Launcher attribute*), 70
StopWatch (*class in openbricks.tools*), 68
straight() (*openbricks.robotics.drivebase.DriveBase method*), 38
SyncServoGroup (*class in openbricks.drivers.st3215*), 45

T

TCA9548A (*class in openbricks.drivers.tca9548a*), 62
TCS34725 (*class in openbricks.drivers.tcs34725*), 53
text() (*openbricks.drivers.ssd1306.SSD1306 method*), 60
time() (*openbricks.tools.StopWatch method*), 68
toggle() (*in module openbricks.bluetooth*), 74
turn() (*openbricks.robotics.drivebase.DriveBase method*), 39

U

use_gyro() (*openbricks.robotics.drivebase.DriveBase method*), 37

V

value() (*openbricks.drivers.qtr.QTRChannel method*), 57
values() (*openbricks.drivers.qtr.QTRReading method*), 54
VL53L0X (*class in openbricks.drivers.vl53l0x*), 59
VL53L1X (*class in openbricks.drivers.vl53l1x*), 60

W

wait() (*in module openbricks.tools*), 68
white() (*openbricks.drivers.qtr.QTRChannel method*), 57
white() (*openbricks.drivers.qtr.QTRElement method*), 54
worst_write_ms() (*in module openbricks.log*), 72
WS2812 (*class in openbricks.drivers.ws2812*), 61