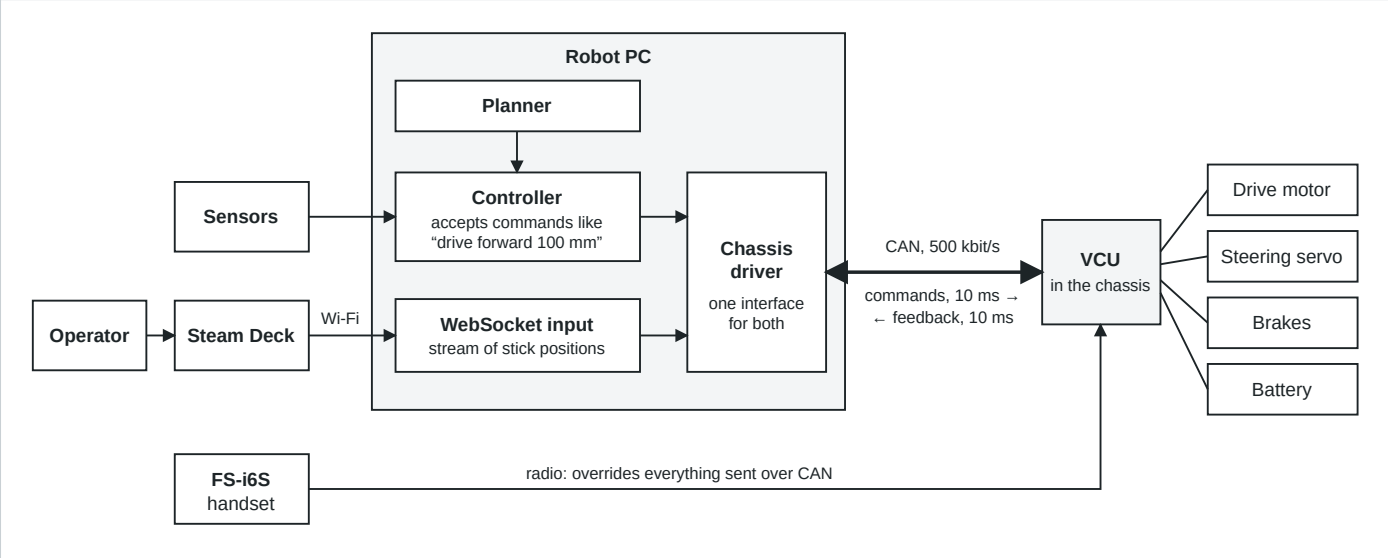
FR-max CAN control primer

FOR	Internal
DATE	10 November 2026

1 · The system

The FR-max is a car-like chassis: one 1800 W motor drives the rear axle, a 400 W servo steers the front wheels, and it brakes with a hydraulic disc brake plus an electromagnetic parking brake that clamps when unpowered. A vehicle control unit (VCU) inside the chassis runs all of this. **We never drive a motor directly**. We send the VCU a request such as "gear D, 0.2 m/s, steering 0°, brake 0 %" over a CAN bus, and the VCU reports the vehicle state back on the same bus.

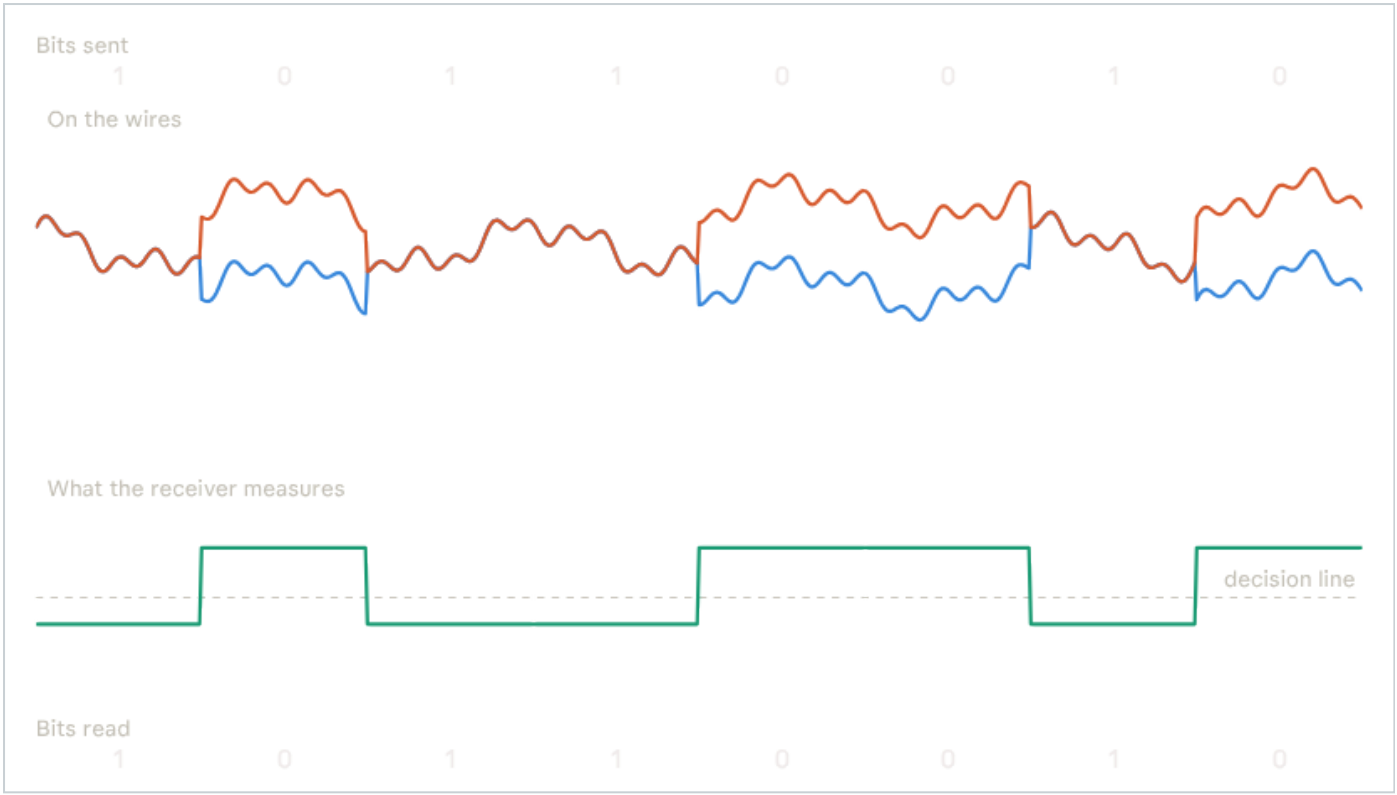


The vendor's FS-i6S radio handset talks to the VCU directly and outranks anything sent over CAN. Page numbers below refer to the manual.

2 · CAN in one page

CAN (Controller Area Network) is two wires, CAN_H and CAN_L, shared by every device ("node") on the bus. Each node can transmit, and every node hears every transmission. There are no addresses, connections or replies.

A bit is carried by the voltage difference between the two wires, not by either wire against ground. No difference is a 1; a difference of about 2 V (CAN_H near 3.5 V, CAN_L near 1.5 V) is a 0. Electrical noise hits both wires equally and drops out in the subtraction.



A transmission is a **frame**: an identifier (ID), up to 8 data bytes, and a CRC error check. The ID says what the frame contains, not who it is for; each node picks out the IDs it cares about. The FR-max uses CAN 2.0B "extended" frames, which means the ID is 29 bits long instead of 11.

The CAN controller chip in each node does the low-level work: it waits for a free bus, resolves collisions (the frame with the lowest ID wins, the other retries automatically), computes and checks the CRC, and acknowledges frames it received intact. Our software only ever sees pairs of `(ID, 8 data bytes)`.

So the "protocol" in the manual is nothing more than a list of IDs and, for each ID, what the 64 data bits mean. Command IDs sent by us end in `D0`; frames from the VCU end in `EF`.

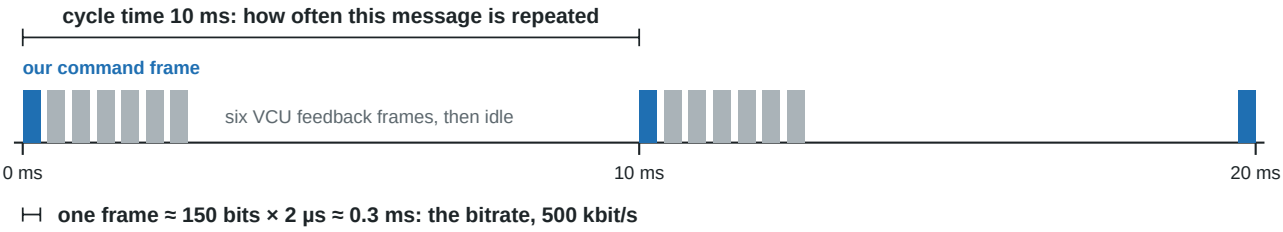
Practical facts:

- The CAN pair is on the 12-pin connector on top of the chassis: pin 1 CAN_H, pin 4 CAN_L. Pins 2 and 3 next to them are the 48 V battery (p. 12).
- **The interface is not isolated. Connecting a CAN wire to a power line can destroy the VCU (p. 22).**
- A CAN bus needs a 120 Ω resistor at each end; it absorbs the signal there so it is not reflected back down the cable. With the chassis off, a correctly terminated bus measures about 60 Ω between CAN_H and CAN_L. Whether the top connector is already terminated: not published.
- On Linux most USB-CAN adapters appear as a network interface (SocketCAN): `ip link set can0 up type can bitrate 500000` starts it, `candump can0` prints every frame. Our CANalyst-II works differently, see section 10.

3 · Bitrate and cycle time

These are two unrelated numbers.

	BITRATE (BAUD RATE)	CYCLE TIME
Value	500 kbit/s	10 ms for motion frames, 50 or 100 ms for others
Meaning	How fast bits travel on the wire: 2 µs per bit	How often one particular message is sent again
Belongs to	The whole bus; every node must use the same value	Each message separately
Set where	Once, in the adapter configuration	In our software: a loop that sends every 10 ms
If wrong	Nothing is understood; the adapter reports bus errors	The VCU decides the sender has died and stops



A frame takes about 0.3 ms on the wire, so a message repeated every 10 ms uses about 3 % of the bus. All FR-max traffic together is roughly 820 frames per second, about a quarter of the bus capacity (estimate).

Messages are repeated because a CAN frame is a statement of the desired state, not an order that gets executed and confirmed. The VCU acts on the most recent statement and treats silence as a failure. A speed request holds only while it is being repeated 100 times per second. The protocol therefore has no "drive 600 mm" message; a distance move has to be built on top (section 9).

4 · Who controls the chassis

Several sources can command the VCU. It obeys the highest-ranked one that is currently valid (p. 22):

RANK	SOURCE	HOW IT ARRIVES
1	Handset	Radio. Active when the handset is on and its SWA switch is up
2	Remote torque command	CAN frame `0x18C4D0D0`: pedal position, steering, brake
3	Auto speed command	CAN frame `0x18C4D2D0`: speed, steering, brake
4	Auto torque command	CAN frame `0x18C4D1D0`: pedal position, steering, brake
5	Parking waiting	Nothing valid: the vehicle stops and waits

A CAN source is valid while its frames keep arriving with a running alive counter and a correct checksum (section 6). When a source becomes invalid, the VCU drops to the next valid one, and to parking waiting if there is none.

CAN commands are obeyed only when the handset is off or its SWA switch is in the lower position ("command control mode"). Flipping SWA up takes the vehicle over at any moment, which is why the manual advises keeping the handset switched on during ground tests (p. 42). The feedback frame reports who is in control: 0 auto, 1 remote, 2 stop.

The modes in that feedback: auto means the VCU is obeying CAN commands from our PC, remote means it is obeying the handset, stop means parking waiting. Which value the remote torque CAN frame reports: not published.

The ranking doubles as a built-in takeover: an operator station sending the remote torque frame always overrides autonomy software sending one of the auto frames. For v1 we send a single torque frame and decide operator-over-controller in our own driver (Acceptance criteria base).

Speed mode and torque mode differ in how much control the VCU does for us. In speed mode the VCU runs its own speed controller: it accelerates to the requested speed, holds it, and works out the motor torque itself; speed 0 brings the vehicle to a stop. In torque mode the request is a share of drive torque, like an accelerator pedal: the speed is whatever results from load and slope, and 0 lets the vehicle coast with some regenerative braking. The VCU's acceleration ramp and controller tuning: not published.

5 · The speed command frame

`ctrl_cmd`, ID `0x18C4D2D0`, sent every 10 ms (p. 25):

SIGNAL	BITS	TYPE	UNIT PER COUNT	MEANING
Gear	0–3	unsigned		1 P (default), 2 R, 3 N, 4 D, 0 disable
Speed	4–19	unsigned	0.001 m/s	Magnitude only; the gear sets the direction
Steering angle	20–35	signed	0.01°	Left positive; the VCU limits it to ±32°
Brake	36–43	unsigned	1 %	0 released, 100 full braking
Alive counter	52–55	unsigned		0 to 15, +1 per frame
Checksum	56–63	unsigned		XOR of bytes 0 to 6

Speed and brake are independent fields, so a frame can request both at once. What the VCU does with a speed request and a brake request together: not published; to brake, send speed 0 with the brake value.

The manual mentions a parking request but defines no separate field for it; gear P is the only candidate.

"Intel format" means little-endian: bit 0 is the lowest bit of byte 0, bit 8 the lowest bit of byte 1, and so on. A signal fills bits upward from its start bit. Because the gear takes only half of byte 0, every later signal straddles byte boundaries:

BYTE	UPPER HALF (BITS 7–4)	LOWER HALF (BITS 3–0)
0	Speed, lowest 4 bits	Gear
1	Speed, middle 8 bits	(same)
2	Steering, lowest 4 bits	Speed, highest 4 bits
3	Steering, middle 8 bits	(same)
4	Brake, lowest 4 bits	Steering, highest 4 bits
5	unused	Brake, highest 4 bits
6	Alive counter	unused
7	Checksum	(same)

The simplest way to build the frame is to treat bytes 0 to 6 as one integer and shift each signal to its start bit:

```
def ctrl_cmd(gear, speed_mps, steer_deg, brake_pct, alive):
    speed = round(speed_mps * 1000)            # unsigned
    steer = round(steer_deg * 100) & 0xFFFF    # signed 16-bit, two's complement
    word = gear | speed << 4 | steer << 20 | brake_pct << 36 | (alive & 0xF) << 52
    data = word.to_bytes(7, "little")
    checksum = 0
    for b in data:
        checksum ^= b
    return data + bytes([checksum])
```

This reproduces the manual's examples (p. 42–44). Worked through for gear D at 5 m/s with counter 0. Bytes are written in hexadecimal: each hex digit is 4 bits, so two digits are one byte.

1. Speed 5 m/s is 5000 counts, which is **1388** in hex.
2. The speed starts at bit 4, one hex digit to the left: **13880**.
3. The gear, 4, goes into the digit that came free: **13884**.
4. Cut into bytes from the right: **84**, **38**, **01**. These are bytes 0, 1 and 2; lowest byte first is what little-endian means.
5. Bytes 3 to 5 are **00**: no steering, no brake.
6. Byte 6 holds the counter in its left digit: **00** now, **10** in the next frame, then **20**.
7. Byte 7 is the XOR of the seven bytes before it: **BD**.

The frame is **84 38 01 00 00 00 00 BD**, the next one **84 38 01 00 00 00 10 AD**: only the counter and the checksum change.

The steering value is the angle of the inner front wheel α, not the single-wheel "bicycle model" angle β that planners use (p. 18). With wheelbase L = 850 mm and track w = 665 mm:

$$\cot\beta = \cot\alpha + \frac{w}{2L}, \qquad R = \frac{L}{\tan\beta}$$

At the ±32° limit β is 26.7° and the rear-axle centre turns on a 1.69 m radius; the manual lists 2.1 m as the minimum turning radius.

The torque frames (`0x18C4D0D0`, `0x18C4D1D0`) have gear in bits 0–3, acceleration mode in bits 4–5, energy-recovery mode in bits 6–7, pedal position in byte 1 (signed, 1 %), steering in bytes 2–3, brake in byte 4, and the same counter and checksum (p. 23–25). A slower frame `0x18C4D7D0`, every 50 ms, switches the turn signals and clearance lamp (p. 26).

6 · Alive counter and checksum

CAN's own CRC (Cyclic Redundancy Check) already guarantees that a received frame equals the transmitted frame. The two extra fields guard against a different failure: a sender that keeps transmitting valid-looking frames while the program behind it is no longer in control.

Alive counter (also called rolling counter or heartbeat). Four bits in the upper half of byte 6, counting 0, 1, … 15, 0, 1, … one step per frame. A counter that stops changing means the same frame is being replayed, for instance by a hung program whose last message is stuck in a buffer. A counter that skips means frames were lost.

Checksum. Byte 7 is the XOR of bytes 0 to 6. It catches a frame that was assembled incorrectly or corrupted in memory before it reached the CAN controller, and it stops unrelated traffic with the same ID from being taken as a command.

When frames stop, the counter stops, or the checksum is wrong, the VCU treats that source as offline, falls through the ranking in section 4, and ends in parking waiting. How many bad or missing frames it tolerates: not published. The fault frame shows the VCU's verdict: bit 4 of `0x18C4EAEF` is "auto speed control CAN communication error".

Two consequences for our software:

- The counter must be advanced by the code that decides the command, in the same loop. If a background thread keeps re-sending the last command with a fresh counter, a crashed controller looks alive and the protection is defeated.
- The VCU's watchdog covers only the link from our PC to the VCU. It knows nothing about the Wi-Fi link to an operator or about a planner that stopped updating. Those need their own timeouts in the chassis driver.

Most feedback frames carry the same counter and checksum, so the driver can detect stale feedback the same way.

7 · Feedback frames

FRAME	ID	EVERY	CONTENTS
`ctrl_fb`	`0x18C4D2EF`	10 ms	Same layout as `ctrl_cmd` with actual gear, speed (always positive), steering angle and brake; control mode in bits 44–45
`lr_wheel_fb`, `rr_wheel_fb`	`0x18C4D7EF`, `0x18C4D8EF`	10 ms	Left and right rear wheel: speed (signed, 0.001 m/s, bits 0–15) and pulse count (signed 32-bit, bits 16–47, 400 pulses per wheel turn)
`Drive_fb_MCUEcoder`	`0x18C4DCEF`	10 ms	Drive motor encoder count (signed 32-bit, 4096 pulses per motor turn) and motor torque (unit not published)
`odo_fb`	`0x18C4DEEF`	10 ms	Distance travelled, signed 32-bit, 0.001 m; resets at power-on
`Veh_fb_Diag`	`0x18C4EAEF`	10 ms	Fault level, CAN communication errors, steering, brake and motor fault codes, e-stop state, handset state
`io_fb`	`0x18C4DAEF`	50 ms	Lamps, front and rear bumper-strip switches
`bms_Infor`, `bms_flag_Infor`	`0x18C4E1EF`, `0x18C4E2EF`	100 ms	Battery voltage, current, capacity; state of charge, protection flags, temperatures

Pulse counts rise when driving forward and fall in reverse (p. 43). Fault levels: 1 warns, 2 also reduces drive power, 3 shuts the drive down (p. 10); the code tables are on p. 38–40.

8 · Operator control from a Steam Deck

The Steam Deck is not on the CAN bus. It sends stick positions over Wi-Fi to the robot PC, where the chassis driver turns them into `ctrl_cmd` frames at 100 Hz.

OPERATOR INPUT	FRAME FIELD	NOTES
Throttle 0 to 1	Speed = throttle × speed cap	Speed mode: releasing the throttle requests speed 0, so the vehicle stops
Forward / reverse	Gear D or R	Speed stays positive. Change gear at standstill; whether the VCU enforces this is not published
Steering −1 to 1	Steering = stick × 32°	Left positive. Steering rate: not published
Brake	Brake %	Speed 0 and brake together for a firm stop
Park	Gear P	Engages the parking brake

The same mapping works with the torque frames. Throttle then sets the accelerator value instead of a speed: a percentage of drive torque, since the chassis has no physical pedals. Releasing the throttle lets the vehicle coast, so stopping, and holding on a slope, needs the brake.

The driver has to add what the VCU does not provide:

- **Acceleration limit.** Ramp the speed request. A 1 m/s² limit is a change of at most 10 mm/s per 10 ms frame. The VCU's own acceleration and braking ramps are not published.
- **Deadman.** If the newest operator packet is older than a fixed time (about 0.2 s), request speed 0 and brake. Without this, a Wi-Fi drop leaves the last throttle value being repeated to the VCU with a perfectly valid counter.
- **Source selection.** One place decides whether operator or controller is driving, with operator input always winning. The VCU's own ranking offers an alternative: operator commands sent as the remote torque frame `0x18C4D0D0` outrank controller commands sent as `0x18C4D2D0`, at the price of pedal control instead of speed control for the operator.

Ways to stop, from softest to hardest: request speed 0; request brake 100 %; gear P; stop sending frames so the VCU falls to parking waiting; take over with the handset; press the emergency-stop button on the rear panel. Only the last one is hardware. A "remote e-stop" sent over Wi-Fi and CAN is one of the first four and depends on software working. The manual states the chassis has none of the safety functions an autonomous vehicle needs (p. 2); a real remote e-stop means a wireless e-stop relay wired into the e-stop circuit.

9 · Driving 600 mm to within 10 mm

Because the VCU accepts speed and not distance, the distance move is a position loop in the controller, the component between the planner and the chassis driver. The controller reads the feedback frames and its own sensors and sends the driver the same kind of request an operator would:

1. Read the encoder count at standstill and store it as the start.
2. Every 10 ms: distance travelled = (count − start) × mm per pulse; remaining = target − travelled.
3. Request a speed that ramps up at the acceleration limit, holds a cruise value, and falls as $\sqrt{2 \cdot \text{deceleration} \cdot \text{remaining}}$ down to a slow creep speed.
4. When remaining is within tolerance: speed 0 and brake, wait until the count stops changing, gear P.
5. Report the distance actually travelled from the final count.

This loop relies on the VCU regulating speed. If the controller uses a torque frame instead, it has to regulate speed itself: the same accelerator value gives a different speed with a different load or slope, and zero does not hold the vehicle on a ramp.

Three signals can measure distance:

SIGNAL	RESOLUTION	MEASURES
Wheel pulse counts	3.3 mm	Each rear wheel separately; their difference shows heading change
Motor encoder count	≈ 0.025 mm	The average of both rear wheels through the differential; includes gear backlash
Odometer	1 mm	Source not published

Resolution follows from the 420 mm tyre: one wheel turn is 1319 mm, divided by 400 pulses, or by 4096 × 13 motor pulses. The manual gives the gear ratio as 13:1 on p. 33 and 10:1 on p. 42; the true mm per pulse has to be measured. A 600 mm move is about 182 wheel pulses or about 24,000 motor pulses, so the motor encoder is the one to control on, with the wheel counts as a cross-check.

10 mm on 600 mm is 1.7 %. Where the error comes from:

SOURCE	SIZE	REMEDY
Scale: the rolling radius of an air-filled tyre changes with load and pressure	1 % radius error = 6 mm, every move, same direction	Calibrate mm per pulse with the full robot mounted, over a few metres against a tape or laser
Stopping: the vehicle keeps moving during the reaction delay	speed × delay; 5 mm at 50 mm/s and 0.1 s (estimate)	Arrive at creep speed; measure the delay. Lowest speed the VCU holds steadily: not published
Slip: encoders measure wheel rotation, not floor travel	Depends on floor and braking	Gentle acceleration and braking
Sideways: steering accuracy is ≤ 0.5° (p. 6)	0.5° over 600 mm = 5 mm	Cannot be corrected at standstill; a car-like chassis only moves sideways while driving

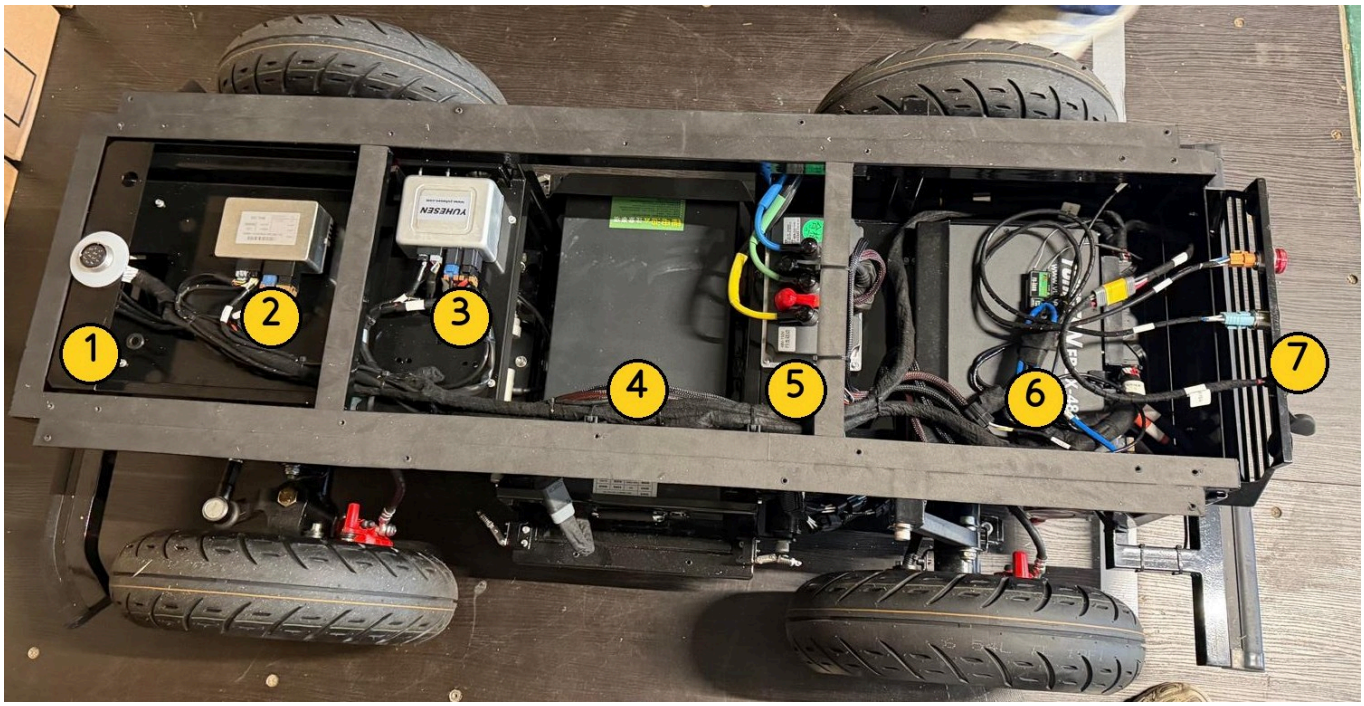
Accuracy and precision map onto these. Accuracy (the average landing point is on target) is set by the scale calibration. Precision (repeated moves land in the same place) is set by how repeatable the stop is and by slip.

Wheel odometry cannot see its own error, and the errors of successive moves add up along a container. Reporting the measured displacement after each move matters more than landing exactly on target, and only a measurement against the surroundings (camera or lidar on the container) bounds the accumulated error.

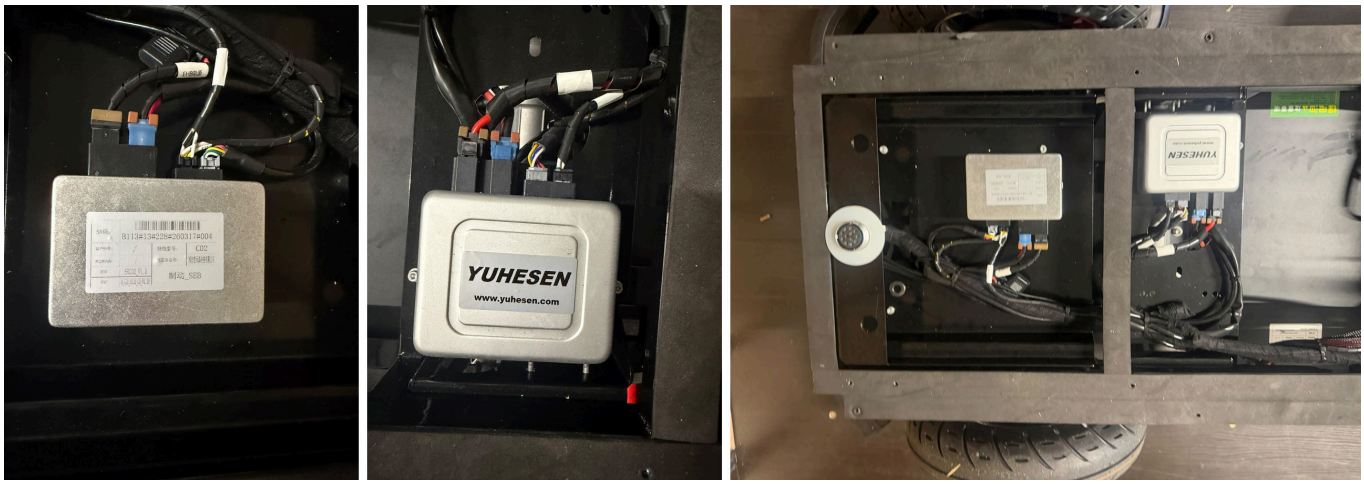
10 · Our chassis and connecting a laptop

What is inside

With the top plate off, front on the left:



NO.	PART	HOW IT IS IDENTIFIED
1	12-pin top connector: CAN and the 48, 24 and 12 V supplies for our equipment	Manual p. 11
2	Brake controller. Drives the hydraulic brake; the "EHB" in the fault codes	Label reads 制动_SEB (brake), its supply lead EHB电源 (EHB power)
3	Yuhesen controller, most likely the steering controller ("EPS")	Same housing and connector set as the brake controller, heavy supply leads, mounted over a motor at the front axle. No type label visible
4	48 V lithium battery with its battery management system (BMS)	Green battery warning label
5	Drive motor controller ("MCU")	Label 48V 150A 行走驱动 (travel drive); yellow, green and blue motor phase cables
6	Electrical box: power distribution and the 24 V and 12 V converters, likely also the VCU; the small unit on top with the thin antenna leads is the handset receiver	Marked YUHESEN FR-MAX-48; contents not opened
7	Rear panel: emergency stop, power switch, charging port	Manual p. 12



Brake controller (2), Yuhesen controller (3), and both in the front bays next to the 12-pin connector.

Each of these boxes is a node on a CAN bus inside the chassis. The VCU sends them setpoints and collects their state; what we receive on the top connector is the VCU's summary. That is why the fault frame has separate "offline" bits for the brake (EHB), steering (EPS), motor controller (MCU) and battery (BMS). Whether the top connector is wired to that same internal bus or to a separate VCU port: not published; the cable label "CAN1" suggests a separate port.

The 12-pin connector and its cable



The plug that mates with the top connector, and the far end of its cable.

The supplied cable has the 12-pin plug on one end; the pin numbers are moulded next to the pins. The other end fans out into loose, labelled wires. Pin assignment from the manual (p. 12):

PIN	SIGNAL	WIRE LABEL ON OUR CABLE
1	CAN_H	CAN1_H
4	CAN_L	CAN1_L
2, 3	Battery + and −, 48 V, up to 20 A	not photographed
5, 6	24 V + and −, up to 15 A	24V+, 24V−
9, 10	12 V + and −, up to 15 A	not photographed
7, 8, 11, 12	USB, reserved	not photographed

The blue 9-pin D-sub connector on the same cable is not described in the manual. It may carry the reserved USB pins or a second copy of CAN; a continuity test from its pins to the 12-pin plug settles it.

Two things to do before the cable goes on the chassis:

- With a multimeter on continuity, confirm that the wire labelled CAN1_H goes to plug pin 1 and CAN1_L to pin 4.
- Insulate every loose supply wire that is not in use. They are live as soon as the chassis is switched on, and the 48 V pair comes straight from the battery.

CANalyst-II to the laptop



CANalyst-II: two independent CAN channels on one screw terminal, each with a pair of termination switches.

The CANalyst-II is a USB-to-CAN adapter with two channels and supports 500 kbit/s and 29-bit IDs, so it fits. One channel is enough.

Wiring: CAN1_H to the H terminal and CAN1_L to the L terminal of the adapter's CAN1 group (the three terminals under the CAN1 marking on the top face). The middle terminal of each group (S or G, shield or ground) stays empty at first; if the adapter reports bus errors, connect it to 12V−. Run the laptop on battery for the first connection. H and L swapped does no damage but nothing is received.

Termination: measure the resistance between CAN1_H and CAN1_L on the cable with the chassis off.

READING	MEANING	ADAPTER SWITCHES FOR THAT CHANNEL
About 60 Ω	Both ends already terminated	Both off
About 120 Ω	One end terminated	One on
No connection	Not terminated	One on; the VCU end is then unknown

The switch pair R1, R2 next to each channel connects the adapter's 120 Ω resistors. Check the result the same way: with the adapter unplugged from the cable, one switch on should read about 120 Ω between its H and L.

Software on Linux: this adapter has no kernel driver, so it does not appear as **`can0`** and **`candump`** does not work with it. The Python library python-can talks to it directly over USB:

```
pip install "python-can[canalystii]"
python -m can.viewer -i canalystii -c 0 -b 500000
```

`-c 0` is the adapter's CAN1. The viewer lists every ID with its cycle time and latest bytes. In a script the same bus is **`can.Bus(interface="canalystii", channel=0, bitrate=500000)`**, and a frame is sent with **`bus.send(can.Message(arbitration_id=0x18C4D2D0, is_extended_id=True, data=ctrl_cmd(...)))`**. If the adapter is not found, USB permissions are the usual cause: add a udev rule for its USB ID (shown by **`lsusb`**, normally **`04d8:0053`**) or run once with sudo. On Windows the vendor's USB-CAN Tool does the same job as the viewer.

First session: chassis on, handset on, nothing sent from the laptop. With the right bitrate and wiring the viewer immediately fills with the 10 ms frames ending in **`EF`** from section 7. That one observation confirms wiring, termination, bitrate and the IDs.

11 · Not in the manual

Each of these decides a design parameter and can be measured on the vehicle. The manual asks for the first motion tests to be done with the wheels off the ground (p. 41).

UNKNOWN	DECIDES	TEST
Bus termination at the top connector	Whether the adapter needs its 120 Ω resistor	Resistance between pins 1 and 4, chassis off
Actual IDs and cycle times	The decoder	Bus viewer with nothing sent (section 10)
Timeout after frames stop; reaction to a frozen counter or wrong checksum	Safety margins, send-loop jitter budget	Stop sending, freeze the counter, break the checksum; watch **`ctrl_fb`** mode and diagnostic bit 4
Handset takeover while the PC is sending	Operating procedure	Flip SWA during a CAN-commanded run
VCU acceleration and braking ramps, lowest steady speed, stop delay	Speed profile and creep speed of the position loop	Speed steps, logged against wheel speed feedback
Gear change rules and parking-brake engage time	Sequencing of D, R and P	Request changes at standstill and while rolling
mm per pulse under load; odometer source	Accuracy	Drive a few metres against a laser, compare all three signals
Repeatability of the 600 mm move	Whether 10 mm is met	Twenty moves forward and back, each measured externally