ADE9000 + ESP32-S3

Complete REST & TCP Endpoint Reference

Three-Phase 32 kSPS Power Analyzer

Device IP: 192.168.1.103

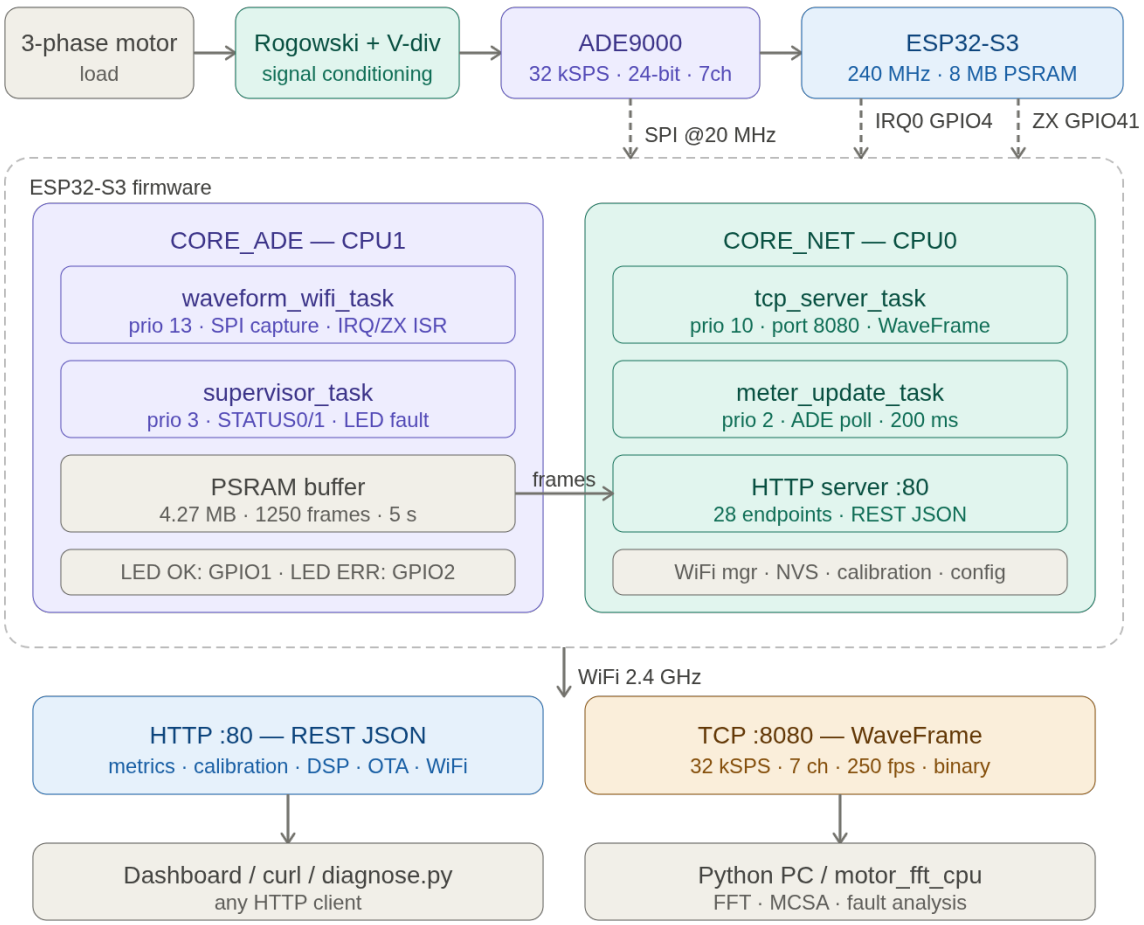
Port	Description
:80	HTTP REST — Metrics, Calibration, DSP, Protections, OTA
:8080	Binary TCP — 32 kSPS Waveform Stream (WaveFrame)

Source: verified against `http_server.cpp, ota_update.cpp, wifi_manager.cpp, waveform_wifi.cpp`

1. Architecture Overview

> **Note:** Important design decision: Full FFT analysis has been removed from the ESP32. The /api/dsp endpoint returns only what the ADE9000 hardware computes natively (RMS, THD, angles, power, imbalance, events). Full spectrum analysis is performed on the PC using raw waveforms captured via TCP:8080.

Signal Flow



Complete Endpoint Index

#	Endpoint	Port	Description
1	GET /api/meter	:80	Real-unit measurements (V, A, W, VAR, VA, PF)
2	GET /api/meter_codes	:80	Raw ADC codes from ADE9000
3	GET /api/calibration	:80	View calibration factors stored in NVS
4	GET /calibrate	:80	Run V/I calibration (key=1234 required)
5	GET /api/calibration/reset	:80	Delete saved calibration from NVS
6	GET /calibrate_energy_start	:80	Start energy calibration session
7	GET /calibrate_energy_finish	:80	Finish energy calibration, save wh/code
8	GET /api/energy_codes	:80	Raw 45-bit energy accumulators
9	GET /api/energy	:80	kWh imported/exported per phase
10	GET /api/dsp	:80	ADE9000 hardware metrics (RMS, THD, angles, imbalance, events, motor info)
11	GET /api/temperature	:80	ADE9000 chip temperature (TEMP_RSLT)
12	GET /api/angles	:80	All 9 ADE9000 phase angle registers
13	GET /api/peaks	:80	Current and voltage peak values (self-clear on read)
14	GET /api/fundamental	:80	Fundamental-only RMS and power per phase
15	GET /api/thd_hw	:80	ADE9000 hardware THD (5.27 fixed-point)
16	GET /api/status	:80	ADE9000 STATUS0/STATUS1 registers with decoded flags
17	GET /api/protection_config	:80	Read DIP/SWELL/OI thresholds
18	POST /api/protection_config	:80	Write protection thresholds to ADE9000 + NVS
19	GET /api/vector_current	:80	ISUMRMS: vector sum current IA+IB+IC
20	GET /api/motor_config	:80	Read motor config (poles, RPM, grid, VFD)
21	POST /api/motor_config	:80	Write motor parameters to NVS
22	GET /api/motor_thresholds	:80	Read alarm/warning thresholds
23	POST /api/motor_thresholds	:80	Write custom thresholds to NVS
24	GET /api/reg	:80	Read any ADE9000 register by address
25	POST /api/ade_reset	:80	ADE9000 soft reset + full reinit
26	POST /api/wifi/forget	:80	Remove a saved WiFi SSID from NVS
27	GET /api/ota/status	:80	OTA firmware update status
28	POST /api/ota	:80	Upload new firmware binary (OTA)
29	TCP INFO	:8080	Query stream parameters (JSON response)
30	TCP CAPTURE seconds=N	:8080	Start binary WaveFrame stream

2. Port :80 — HTTP REST Endpoints

> **Note:** All port :80 endpoints return JSON. Endpoints that modify state require key=1234 query parameter.

#1 · GET /api/meter — Real-Unit Measurements

`GET http://192.168.1.103/api/meter`

Returns electrical quantities converted using NVS calibration gains. If not calibrated, power values are estimated as V × I × PF (power_scaled=false).

```
curl "http://192.168.1.103/api/meter"
```

Field	Type	Description
`seq`	uint32	Snapshot sequence number. Increments each measurement cycle (~4 ms).
`ts_us`	uint64	Timestamp in µs since ESP32 boot (not UTC).
`freq_hz`	float	Grid frequency measured by ADE9000 in Hz.
`vrms_V[3]`	float[]	RMS voltage [A, B, C] in Volts. Zero if not calibrated.
`irms_A[4]`	float[]	RMS current [IA, IB, IC, IN] in Amperes.
`pf[3]`	float[]	Power factor per phase [PFA, PFB, PFC]. Range: [-1.0, +1.0].
`power_scaled`	bool	true = full calibration gains used. false = V×I×PF estimate.
`p_W[3]`	float[]	Active power [PA, PB, PC] in Watts.
`q_VAR[3]`	float[]	Reactive power in VAR per phase.
`s_VA[3]`	float[]	Apparent power in VA per phase.
`p_total_W`	float	Sum PA + PB + PC in Watts.
`q_total_VAR`	float	Sum QA + QB + QC in VAR.
`s_total_VA`	float	Sum SA + SB + SC in VA.
`status.zxto[3]`	bool[]	Zero-crossing timeout per phase [A, B, C].
`status.seq_error`	bool	true = incorrect phase sequence (SEQERR).

#2 · GET /api/meter_codes — Raw ADC Codes

`GET http://192.168.1.103/api/meter_codes`

Returns raw ADE9000 register values with no conversion. Includes total RMS, fundamental-only RMS, powers, THD, angles, events, and hardware-reference freshness plane.

```
curl "http://192.168.1.103/api/meter_codes"
```

Field	Type	Description
`seq, ts_us, freq_hz`	—	Same meaning as /api/meter.
`calibrated`	bool	true if v_gain_a and i_gain_a are loaded in NVS.
`note_codes`	string	Status message indicating calibration state.
`vrms_codes[3]`	uint32[]	AVRMS/BVRMS/CVRMS registers in counts.
`irms_codes[4]`	uint32[]	AIRMS/BIRMS/CIRMS/NIRMS registers in counts.
`ifrms_codes[3]`	uint32[]	AIFRMS/BIFRMS/CIFRMS (fundamental-only current).
`vfrms_codes[3]`	uint32[]	AVFRMS/BVFRMS/CVFRMS (fundamental-only voltage).
`p_codes[3]`	int32[]	AWATT/BWATT/CWATT active power registers.
`q_codes[3]`	int32[]	AVAR/BVAR/CVAR reactive power registers.
`s_codes[3]`	int32[]	AVA/BVA/CVA apparent power registers.
`pf_codes[3]`	int32[]	APF/BPF/CPF power factor (s.23 format).
`fp_codes[3]`	int32[]	AFWATT/BFWATT/CFWATT fundamental active power.
`fq_codes[3]`	int32[]	AFVAR/BFVAR/CFVAR fundamental reactive power.
`fs_codes[3]`	uint32[]	AFVA/BFVA/CFVA fundamental apparent power.

`ithd_pct[3]`	float[]	Hardware current THD [%] from AITHD/BITHD/CITHD.
`vthd_pct[3]`	float[]	Hardware voltage THD [%] from AVTHD/BVTHD/CVTHD.
`angles_deg.*`	float	9 ADE9000 angle registers converted to degrees (va_vb, vb_vc, va_vc, va_ia, vb_ib, vc_ic, ia_ib, ib_ic, ia_ic).
`status.*`	bool	ADE9000 STATUS0/STATUS1 decoded flags: zxto[3], seq_error, dip_any, swell_any, overcurrent, mismatch, thd_pf_rdy, coh_wfb_full.
`hw_freshness.valid_thd/angle/fundamental[3]`	bool[]	Whether each hardware register type is currently valid.
`hw_freshness.age_thd/angle/fundamental[3]`	uint[]	1-second windows elapsed since last valid reading.
`hw_freshness.cycle_reference_valid`	bool	true if freq_hz is in valid range (40–70 Hz or 5–200 Hz for VFD).
`hw_freshness.samples_per_cycle`	float	Fs / f_grid (~533 at 60 Hz).

#3 · GET /api/calibration — View Saved Calibration

GET `http://192.168.1.103/api/calibration`

```
curl "http://192.168.1.103/api/calibration"
```

```
{
  "valid": true, "version": 3,
  "voltage_gain": {"a": 4.55e-06, "b": 4.54e-06, "c": 4.56e-06},
  "current_gain": {"a": 4.19e-07, "b": 4.21e-07, "c": 4.18e-07, "n": 0.0},
  "power_gain":   {"p": {"a": 0.0, "b": 0.0, "c": 0.0}, ..., "calibrated": true},
  "energy":       {"flags": 1, "calibrated": true, "wh_per_code": {"a": 2.3e-09, ...}}
}
```

Field	Type	Description
`valid`	bool	true if calibration data is loaded in RAM cache.
`version`	uint32	Calibration schema version (currently 3).
`voltage_gain.a/b/c`	float	V/code factor: V = code × v_gain. Typical: ~4.5×10^{-6}.
`current_gain.a/b/c/n`	float	A/code factor: I = code × i_gain. Typical: ~4.2×10^{-7}.
`power_gain.p/q/s.a/b/c`	float	W, VAR, VA per code. 0 = not calibrated (uses V×I×PF).
`power_gain.calibrated`	bool	true if valid V/I or explicit power gains are present.
`energy.calibrated`	bool	true if wh_per_code values are valid.
`energy.wh_per_code.a/b/c`	float	Wh per AWATTHR/BWATTHR/CWATTHR code.

#4 · GET /calibrate — Run V/I Calibration

GET `/calibrate?key=1234&va=V&vb=V&vc=V&ia=A&ib=A&ic=A&in=A`

Reads current ADE9000 codes and computes: v_gain = V_ref / AVRMS_code, i_gain = A_ref / AIRMS_code. Saves to NVS key cal_data.

```
curl "http://192.168.1.103/calibrate?
key=1234&va=120.0&vb=119.8&vc=120.2&ia=10.05&ib=9.98&ic=10.12&in=0.0"
```

Param	Unit	Constraint	Description
`key`	—	1234	Mandatory security key.
`va/vb/vc`	Vrms	>10 V	RMS voltage measured with reference multimeter.

`ia/ib/ic`	Arms	>0 A	RMS current per phase.
`in`	Arms	≥0 A	Neutral RMS current. Can be 0 for balanced load.

Response fields: ok, saved, nvs_key, ref_input, adc_codes, gains_saved, next_step.

#5 · GET /api/calibration/reset

GET `/api/calibration/reset?key=1234`

Deletes all calibration factors from NVS. All gains set to 0. /api/meter returns zeros until recalibration.

```
curl "http://192.168.1.103/api/calibration/reset?key=1234"
```

#6–7 · Energy Calibration (two-step)

Step 1: `/calibrate_energy_start?key=1234`

Step 2: wait 120–300 s with constant resistive load

Step 3: `/calibrate_energy_finish?key=1234&pa=W&pb=W&pc=W`

> **Atención:** calibrate_energy_finish rejects calls where Δt < 30 s (dt_too_short error). Recommended: 120–300 s.

Response from finish: ok, saved, dt_s, pref_w[3], delta_codes[3], wh_ref[3], wh_per_code[3], warnings[] (optional).

#8–9 · Energy Endpoints

GET `/api/energy_codes`

Returns raw 45-bit signed accumulators: watthr_codes[3], varhr_codes[3], vahr_codes[3].

GET `/api/energy`

Returns valid, import_kwh[3], export_kwh[3], wh_per_code[3]. Requires energy calibration. Values accumulate since last reset/calibration.

#10 · GET /api/dsp — ADE9000 Hardware Metrics

GET `http://192.168.1.103/api/dsp`

```
curl "http://192.168.1.103/api/dsp"
```

> **Atención:** This endpoint contains only what the ADE9000 hardware computes natively. No Goertzel, no FFT, no bearing analysis, no MHI. All advanced analysis is performed externally on the PC using TCP:8080 raw waveforms.


```
{
  "valid": true, "timestamp_us": 48270000, "frequency_hz": 59.982,
  "voltage":      {"vrms_a": 26000000, "vrms_b": 26010000, "vrms_c": 25990000},
  "current":      {"irms_a": 23000000, "irms_b": 23100000, "irms_c": 22900000, "irms_n":
95000},
  "power":        {"p_a": 21160000, ..., "p_total": 62950000, ...},
  "power_factor": {"pf_a": 0.9231, "pf_b": 0.9187, "pf_c": 0.9054},
  "fundamental":  {"ifrms_a": 22900000, ..., "valid": [true,true,true], ...},
  "thd":          {"thd_i_a": 2.59, "thd_i_b": 3.55, "thd_i_c": 3.68, "thd_v_a": 1.2, ...},
  "angles":       {"va_ia_deg": 22.7, "va_vb_deg": 120.1, ..., "valid_angle":
[true,true,true]},
  "imbalance":    {"current_pct": 0.87, "voltage_pct": 0.11, "zero_seq_pct": 0.41},
  "events":       {"dip": false, "swell": false, "overcurrent": false, ...},
```

```
  "motor":        {"rpm": 1797.4, "rpm_source": "estimated_from_grid_freq", "poles": 4, ...},
  "note": "ESP32 raw metrologia only -- DSP on client"
}
```

Field	Type	Description
`voltage.vrms_a/b/c`	uint32	AVRMS/BVRMS/CVRMS registers in counts.
`current.irms_a/b/c/n`	uint32	AIRMS/BIRMS/CIRMS/NIRMS in counts.
`power.p/q/s_a/b/c`	int32	AWATT/AVAR/AVA per phase (total).
`power.p/q/s_total`	int64	Sum across three phases.
`power_factor.pf_a/b/c`	float	APF/BPF/CPF converted from s.23 format.
`fundamental.*`	mixed	AIFRMS, AVFRMS, AFWATT, AFVAR, AFVA per phase + validity + age.
`thd.thd_i_a/b/c`	float	Hardware THD current [%] (AITHD 5.27 fixed-point).
`thd.thd_v_a/b/c`	float	Hardware THD voltage [%] (AVTHD 5.27 fixed-point).
`thd.valid_thd[3]`	bool[]	Whether THD_PF_RDY flag was set for each phase.
`angles.va_ia/vb_ib/vc_ic_deg`	float	Power angle (V-I) per phase in degrees.
`angles.va_vb/vb_vc/va_vc_deg`	float	Line-to-line voltage angles. Balanced: ~120°.
`angles.ia_ib/ib_ic/ia_ic_deg`	float	Current inter-phase angles.
`imbalance.current_pct`	float	NEMA MG-1 current unbalance: max|Ix-Ī|/Ī×100.
`imbalance.voltage_pct`	float	Voltage unbalance [%].
`imbalance.zero_seq_pct`	float	Zero-sequence current: IN/(IA+IB+IC)×100.
`events.*`	bool	ADE9000 real-time event flags from STATUS0/STATUS1: dip, swell, overcurrent, seq_error, mismatch, zx_timeout.
`motor.rpm`	float	Effective RPM currently used for analysis.
`motor.rpm_source`	string	"tachometer" or "estimated_from_grid_freq".
`motor.poles`	int	Motor poles configured in NVS.
`motor.grid_freq_hz`	float	Grid frequency used for ADE9000 ACCMODE.
`motor.is_vfd`	bool	true if VFD mode is active.
`motor.vfd_freq_hz`	float	VFD output frequency (0 if not VFD mode).
`motor.cycle_reference_valid`	bool	true if freq_hz is in valid range for ADE9000 angle plane.
`motor.samples_per_cycle`	float	Fs / f_grid.
`note`	string	Always "ESP32 raw metrologia only — DSP on client".

#11 · GET /api/temperature

GET `http://192.168.1.103/api/temperature`

Reads TEMP_RSLT (12-bit). Formula UG-1098 Table 52: T_C = (raw - 111.4) / 1.169. Accuracy: ±5°C.

Response: ok, raw (12-bit uint), temp_c (float), note.

```
curl "http://192.168.1.103/api/temperature"
```

#12 · GET /api/angles

GET `http://192.168.1.103/api/angles`

Returns all 9 ANGL_* registers. Formula: deg = raw × 360 × f_Hz / 1,024,000.

Response: ok, freq_hz, angles[] array with {name, raw, deg, ok} per entry.

```
curl "http://192.168.1.103/api/angles"
```

Names: va_vb, vb_vc, va_vc, va_ia, vb_ib, vc_ic, ia_ib, ib_ic, ia_ic.

#13 · GET /api/peaks

`GET http://192.168.1.103/api/peaks`

> **Atención:** IPEAK and VPEAK registers self-clear on read. Each call consumes accumulated peaks.

Response: ok, current.{code, phase, phase_name, valid}, voltage.{code, phase, phase_name, valid}.

Bits[26:25] of IPEAK/VPEAK give the source phase: 0=A, 1=B, 2=C. Bits[24:0] give the peak magnitude.

```
curl "http://192.168.1.103/api/peaks"
```

#14 · GET /api/fundamental

`GET /api/fundamental` or `GET /api/fundamental?units=1`

Returns AIFRMS, AVFRMS, AFWATT, AFVAR, AFVA per phase. Without ?units=1: raw codes. With ?units=1: converts using calibration gains.

Response per phase (A, B, C): irms_code/irms_A, vrms_code/vrms_V, p_code/p_W, q_code/q_VAR, s_code/s_VA, p_calibrated.

```
curl "http://192.168.1.103/api/fundamental"
```

#15 · GET /api/thd_hw

`GET http://192.168.1.103/api/thd_hw`

Reads AITHD/BITHD/CITHD and AVTHD/BVTHD/CVTHD. Updated by ADE9000 every ~1.024 s.

Conversion: THD% = raw_s.27 / 2^{27} × 100.

Response: ok, note, ithd_a/b/c [%], vthd_a/b/c [%].

```
curl "http://192.168.1.103/api/thd_hw"
```

#16 · GET /api/status

`GET http://192.168.1.103/api/status`

```
curl "http://192.168.1.103/api/status"
```

Field	Type	Description
`status0_raw`	uint32	Raw STATUS0 register value.
`status1_raw`	uint32	Raw STATUS1 register value.
`flags0.page_full`	bool	Waveform buffer half-page full (STATUS0 bit17).
`flags0.coh_wfb_full`	bool	Coherent waveform buffer full (STATUS0 bit23).
`flags0.wfb_trig`	bool	Waveform buffer trigger fired (STATUS0 bit22).
`flags0.thd_pf_rdy`	bool	New THD/PF results available (STATUS0 bit21).
`flags0.pwrrdy`	bool	Power registers updated (STATUS0 bit18).
`flags0.mismatch`	bool	ISUMRMS exceeds ISUMLVL (STATUS0 bit24).
`flags1.dip_a/b/c`	bool	Active voltage dip per phase (STATUS1 bits 23–25).
`flags1.swell_a/b/c`	bool	Active overvoltage per phase (STATUS1 bits 20–22).

`flags1.oi`	bool	Overcurrent detected (STATUS1 bit17).
`flags1.seqerr`	bool	Incorrect phase sequence (STATUS1 bit18).
`flags1.zxto_a/b/c`	bool	Zero-crossing timeout per phase (STATUS1 bits 6–8).

#17–18 · GET/POST /api/protection_config

GET `/api/protection_config`

```
curl "http://192.168.1.103/api/protection_config"
```

POST `/api/protection_config?key=1234&dip_lvl=N&swell_lvl=N&oi_lvl=N&dip_cyc=N&swell_cyc=N`

POST writes to ADE9000 registers immediately and saves to NVS. Clears STATUS0/STATUS1 latched flags (W1C).

```
curl -X POST "http://192.168.1.103/api/protection_config?
key=1234&dip_lvl=N&swell_lvl=N&oi_lvl=N&dip_cyc=N&swell_cyc=N"
```

Field	Type	Description
`dip_lvl`	uint32	DIP_LVL register. ADE activates dip when AVRMS < this value.
`swell_lvl`	uint32	SWELL_LVL. ADE activates swell when AVRMS > this value.
`oi_lvl`	uint32	OILVL. ADE activates overcurrent when AIRMS > this value.
`dip_cyc`	uint16	DIP_CYC. Minimum half-cycles before DIP is declared valid.
`swell_cyc`	uint16	SWELL_CYC. Minimum half-cycles before SWELL is declared valid.

#19 · GET /api/vector_current

GET `http://192.168.1.103/api/vector_current`

Reads ISUMRMS (register 0x0074): RMS of vector sum IA + IB + IC. Zero in a perfectly balanced system.

```
curl "http://192.168.1.103/api/vector_current"
```

Response: ok, isumrms_code (uint32), isumrms_A (float, 0 if not calibrated), calibrated, i_gain_avg, note.

#20–21 · GET/POST /api/motor_config

GET `/api/motor_config`

POST `/api/motor_config?poles=4&rpm=1742&grid_freq=60&vnom=120&is_vfd=1&vfd_freq=45`

All parameters persisted in NVS. Applied live without reboot.

Field / Param	Type	Description
`poles`	int	Motor poles: 0 (reset) or even 2–12.
`rpm_measured`	float	Tachometer RPM. 0 = use estimated value.
`rpm_source`	string	Read-only. "tachometer" or "estimated_from_grid_freq".
`rpm_effective`	float	Read-only. RPM currently used by the system.
`rpm_sync`	float	Read-only. Synchronous speed: 120 × f_Hz / poles.
`rpm_estimated`	float	Read-only. Estimated with nominal slip.
`grid_freq_hz`	float	Grid frequency: 0 (auto-detect) or 10–200 Hz. Used for ADE9000 ACCMODE.
`vnom_v`	float	Nominal phase-neutral voltage [Vrms]. 0 = auto (120 V at 60 Hz, 230 V at 50 Hz).

`is_vfd`	0/1	1 = VFD mode: accepts 5–200 Hz, disables SEQERR fault.
`vfd_freq_hz`	float	VFD output frequency [5–200 Hz]. Used as seed for first cycle.

```
# Standard 60 Hz, 4-pole motor
curl -X POST "http://192.168.1.103/api/motor_config?poles=4&rpm=1742"


# VFD at 45 Hz output, 60 Hz grid
curl -X POST "http://192.168.1.103/api/motor_config?is_vfd=1&grid_freq=60&vfd_freq=45"


# European: 50 Hz grid, 230 V
curl -X POST "http://192.168.1.103/api/motor_config?grid_freq=50&vnom=230"
```

#22–23 · GET/POST /api/motor_thresholds

GET `/api/motor_thresholds`

```
curl "http://192.168.1.103/api/motor_thresholds"
```

POST `/api/motor_thresholds?key=1234&thd_warn=4.5&thd_alarm=7.0&...`

Partial updates supported: only fields present in the query string are modified.

```
curl -X POST "http://192.168.1.103/api/motor_thresholds?
key=1234&thd_warn=4.5&thd_alarm=7.0&..."
```

Field	Type	Default	Description
`is_custom`	bool	false	true after first POST.
`thd_warn`	float	5.0%	THD warning threshold [0–50%].
`thd_alarm`	float	8.0%	THD alarm threshold. Must be > thd_warn.
`unbalance_warn`	float	5.0%	Current unbalance warning [%].
`unbalance_alarm`	float	10.0%	Current unbalance alarm [%].
`bearing_warn`	float	0.3	Bearing risk warning [0–1].
`bearing_alarm`	float	0.6	Bearing risk alarm. Must be > bearing_warn.
`mhi_thd_factor`	float	1.5	THD weight in MHI: penalty = THD% × factor.
`mhi_unbal_factor`	float	2.0	Unbalance weight in MHI.
`mhi_bearing_factor`	float	20.0	Bearing weight in MHI.
`thd_baseline`	float	0.0	Learned baseline THD for this motor.
`bearing_baseline`	float	0.0	Learned baseline bearing amplitude.
`startup_irms_codes_threshold`	float	2000.0	Start-up detection threshold in ADC codes.
`note`	string	—	Indicates whether factory or custom thresholds are active.

#24 · GET /api/reg — Read ADE9000 Register

GET `/api/reg?key=1234&addr=0xXXXX`

Reads any ADE9000 register by SPI address. Response: addr, val32 (uint32), hex32 (string). For 16-bit registers also returns val16, hex16.

```
curl "http://192.168.1.103/api/reg?key=1234&addr=0xXXXX"
```

#25 · POST /api/ade_reset

POST `/api/ade_reset?key=1234`

Performs ADE9000 soft reset (CONFIG1 bit 0) then calls full init_ade9000() to reapply calibration, protections, WFB, and all configuration. Does not restart the ESP32.

Response: ok, message.

```
curl -X POST "http://192.168.1.103/api/ade_reset?key=1234"
```

#26 · POST /api/wifi/forget

POST `/api/wifi/forget?ssid=NetworkName`

Removes the SSID from NVS saved networks and triggers reconnection. If no other saved networks are available, ESP32 starts the MCSA-SETUP-XXYY access point at 192.168.4.1.

```
curl -X POST "http://192.168.1.103/api/wifi/forget?ssid=OldNetworkName"
```

Response: ok, forgotten (SSID string), note or error.

#27–28 · OTA Firmware Update

GET `/api/ota/status`

#27 — Returns current OTA state.

```
curl "http://192.168.1.103/api/ota/status"
```

POST `/api/ota`

#28 — Upload new firmware binary. Performs OTA partition swap and reboot.

```
curl -X POST \
  --data-binary @build/MCSA_V1.6.bin \
"http://192.168.1.103/api/ota?key=1234"
```

3. Port :8080 — Binary TCP Waveform Stream

Raw TCP socket (not HTTP). Send ASCII command ending with \n. Only 1 simultaneous client.

Stream Parameters

Parameter	Value
Sampling frequency	32,000 Hz
Channels per set	7: IA, VA, IB, VB, IC, VC, IN
Sets per frame	128 (1024 locations ÷ 8 slots)
Bytes per frame	3,610 = 26 header + 3,584 payload
Frames per second	250 fps
Payload endianness	Big-Endian (MSB first, direct from SPI)
Data type	`uint32 → shift >>4 for 24-bit signed ADC`
Max clients	1

WaveFrame Header Structure (26 bytes)

Offset	Bytes	Field	Description
0	1	0xA5	Sync byte 1
1	1	0x5A	Sync byte 2
2	1	ver	Version
3	1	flg	Flags
4	4	seq	Sequence number (uint32 BE)
8	8	ts_us	Timestamp in µs (uint64 BE)
16	4	fs	Sampling frequency (uint32 BE)
20	2	spc	Sets per channel (uint16 BE)
22	1	ch	Channel count
23	1	str	Stride (words per set)
24	2	pb	Payload bytes (uint16 BE)

Payload (3,584 bytes): [IA_0, VA_0, IB_0, VB_0, IC_0, VC_0, IN_0] × 128 sets. Big-Endian uint32.

#29 · TCP INFO Command

TCP `echo "INFO" | nc 192.168.1.103 8080`

Returns JSON then closes connection. Fields: fs_hz, channels, stride_words, frame_bytes, payload_bytes, buffer_frames, buffer_count, buffer_full, state, endian ("big"), dtype ("i4"), event_capture.latched (bool), accumulated_bursts.

#30 · TCP CAPTURE Command

TCP `CAPTURE seconds=N\n → text ACK line, then binary WaveFrame stream`

```
# ACK format:
OK fs=32000 channels=7 stride=7 frame_bytes=3610 frames=1300 bursts=4
```

seconds=0 = infinite stream (until disconnection).

Python Decode Example

```
import struct, numpy as np

FRAME_SIZE = 3610
HDR_SIZE   = 26
SETS       = 128
CHANNELS   = 7        # IA VA IB VB IC VC IN
FSC        = 8_388_608.0   # 2^23

def decode_frame(raw_bytes):
    assert raw_bytes[0] == 0xA5 and raw_bytes[1] == 0x5A, 'sync error'
    seq    = struct.unpack_from('>I', raw_bytes, 4)[0]
    ts     = struct.unpack_from('>Q', raw_bytes, 8)[0]
    words  = np.frombuffer(raw_bytes[HDR_SIZE:], dtype='>u4').astype(np.int32)
    matrix = words.reshape(SETS, CHANNELS)
    adc    = (matrix << 8) >> 12   # sign-extend + >>4
    pct    = adc / FSC * 100.0    # % of full scale
    return seq, ts, pct
    # columns: 0=IA  1=VA  2=IB  3=VB  4=IC  5=VC  6=IN
```

4. Rogowski Coil Turns Multiplier

When the motor current is too low for the Rogowski coil to produce adequate signal, the cable is passed through the coil N times. The calibration endpoint handles this automatically:

```
i_gain = I_real / code_ADE = (I_measured / N) / code_ADE
```

Set ia, ib, ic in /calibrate to the real current (not the amplified value seen by the ADE9000). The firmware computes the gain that already embeds the 1/N correction.

> **Note:** For currents below ~1 A, use 10 turns through the Rogowski coil. The effective sensitivity increases 10× while the stored i_gain compensates automatically. No firmware changes are needed.

5. Quick Reference

```
IP="192.168.1.103"

# Measurements
curl "$IP/api/meter"
curl "$IP/api/dsp"

# Calibration (V/I)
curl "$IP/calibrate?key=1234&va=120.0&vb=119.8&vc=120.2&ia=10.05&ib=9.98&ic=10.12&in=0.0"

# Energy calibration
curl "$IP/calibrate_energy_start?key=1234"
# ... wait 120-300 s with constant load ...
curl "$IP/calibrate_energy_finish?key=1234&pa=1200.0&pb=1150.0&pc=1180.0"

# Motor config: VFD at 45 Hz, 60 Hz grid, 4-pole
curl -X POST "$IP/api/motor_config?is_vfd=1&grid_freq=60&vfd_freq=45&poles=4"

# Custom thresholds
curl -X POST "$IP/api/motor_thresholds?key=1234&thd_warn=4.5&thd_alarm=7.0&bearing_warn=0.12"

# Forget WiFi
curl -X POST "$IP/api/wifi/forget?ssid=OldNetwork"

# ADE9000 reset
curl -X POST "$IP/api/ade_reset?key=1234"

# TCP waveform capture
python3 -c "
import socket
s = socket.create_connection(('$IP', 8080))
s.send(b'CAPTURE seconds=5\n')
ack = b""
while b"\n" not in ack: ack += s.recv(256)
print('ACK:', ack.decode().strip())
data = b""
while True:
    c = s.recv(65536)
    if not c: break
    data += c
open('capture.bin','wb').write(data)
print(f'{len(data)//3610} frames received')
"
```