EQSP32 IOT LOGIC CONTROLLER FAMILY

Industrial IoT (IIoT) ESP32 Logic Controller (PLC) with 16 IO, Serial & CANBus Communication

1 Description

EQSP32 is a compact, powerful, and user-friendly Industrial IoT (IIoT) ESP32 PLC controller, featuring WiFi and Bluetooth for wireless connectivity and Ethernet, that is ideal for a broad range of Industrial/Home IoT Automation, Remote Monitoring/Control, and Instrumentation applications. It offers the convenience of a Mini-Programmable Logic Controller (PLC) with its DIN-rail-mount and screw terminals design, coupled with the power and flexibility of an open-architecture embedded 32-bit computer.

At its core, the EQSP32 is powered by the ESP32-S3, a widely used System on Chip known for its high-performance dual-core processor and integrated wireless radio. This open architecture allows users to leverage the extensive ecosystem of tools and software available for the ESP32, amplifying the capabilities of EQSP32.

Furthermore, with EQ-AI, Erqos' AI programming assistant, users can develop applications with only a fraction of the skills and time normally required in typical PLC or embedded computer development. By simply describing their needs in plain English, EQ-AI can generate the necessary code for the EQSP32 in seconds, enabling extremely fast IoT automation project development with minimal coding expertise.

The EQSP32 is equipped with 16 configurable I/O terminals for interfacing with all common types of sensors and actuators, RS232/485 communication, and an optional CANbus interface, making it suitable for a wide array of internet-connected automation and monitoring applications.

Erqos' comprehensive approach makes the EQSP32 a complete end-to-end solution, from interfacing with the physical environment, to offering smartphone-operated User Interfaces, like no other ESP32 PLC. This includes a variety of interfaces for different sensors and actuators, a powerful processing unit, and intuitive software for seamless Wi-Fi connectivity and tools to develop custom smartphone apps.

Lastly, the EQSP32 features an expansion connector for easy addition of various add-on modules, ensuring that the system can grow alongside the evolving needs of its users.

© 2026 Erqos Technologies P.C. · erqos.com

2 Key Features

- Dual-core ESP32-S3 SoC running at 240 MHz with 512 KB RAM and 8 MB Flash
- Integrated WiFi and Bluetooth Low Energy (BLE) connectivity
- Ethernet or LoRa connection (model dependent)
- Integrated 10-26 V power input DC/DC converter
- 5 V 1 A output for supplying user sensors and devices
- Continuous monitoring of power supply voltages
- LED indicators for power, device status and Bluetooth/WiFi connectivity status
- Built-in Buzzer audio indicator
- Front-facing USB-C connector for programming
- Software selectable RS232/RS485 serial port
- Modbus and DMX on RS485 through third-party libraries
- CAN bus interface (second RS485 port instead on EQSP32RE)
- Surge protection on RS232/485 and CAN Interfaces
- Built in 16-channel PWM controller
- 16 EQVersal® Inputs/Outputs terminals, with multiple software-configurable modes:
  - Digital inputs
  - Digital Inputs with Debounce Delay
  - Pulse counter inputs
  - Quadrature encoder inputs
  - Direct pin access for custom code and third-party libraries
  - Low-side on/off outputs (open collector)
  - Low-side PWM outputs
  - Relay driver outputs, power-optimized for relay coils
  - 0-10 V Absolute Voltage Analog inputs
  - 0-20 mA and 4-20 mA current loop sensors
  - Analog Inputs relative to the 5 V Supply
  - NTC Temperature Sensor Analog Inputs
- Analog and temperature inputs on 8 of the 16 terminals
- Protected inputs, tolerant up to VIn
- 12-bit Analog Inputs resolution
- 1 A max per output
- Current limited and thermally protected Output drivers
- Flyback diodes on each Output for safe and efficient switching of Inductive loads
- Built-in hardware watchdog: switches all outputs off if the controller stops updating them, for example after a firmware reset or crash
- Expansion connector for up to 15 add-on modules for additional I/O or special function interface (ex. relay and analog outputs, thermocouple, PT100, pH sensor etc.)
- Integrated System Supervisor Software handling core functions:
  - Initial WiFi Setup using EQConnect Smartphone App
  - Connection to WiFi
  - Clock/Calendar sync over Internet
  - Input/Output and PWM updates
  - Status Check and LED indicators update
  - MQTT connection and Home Assistant discovery
  - Over-the-air (OTA) program updates
- Library for efficient and transparent handling of pin, peripheral and connectivity control and configuration
- EQ-AI coding assistant trained with System Supervisor and Library
- Works with Arduino IDE, VS Code and OpenPLC
- Compact, 4 SU wide (71 mm), DIN-rail mount enclosure with 24 screw terminals
- IP20 Protection
- -20 °C to +55 °C ambient temperature

3 Applications

- Home Automation
- Industrial Monitoring & Control
- HVAC Systems
- IoT Automation
- Environmental Monitoring
- Agricultural Applications
- Energy Management
- Healthcare Monitoring
- Retail and Inventory Management
- Robotics and Automation
- Educational Projects
- Entertainment and Art Installations

4 Ordering Information

Part number	Description	Interfaces	Antenna
EQSP32CE	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, Ethernet, CAN, RS232/RS485	Internal Wi-Fi/BLE
EQSP32CE-X	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, Ethernet, CAN, RS232/RS485	External Wi-Fi/BLE (SMA)
EQSP32C-X	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, CAN, RS232/RS485	External Wi-Fi/BLE (SMA)
EQSP32CL-X	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, LoRa, CAN, RS232/RS485	Internal Wi-Fi/BLE, external LoRa
EQSP32RE[1]	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, Ethernet, 2x RS485, RS232	Internal Wi-Fi/BLE
EQSP32RE-X[1]	IoT Logic Controller, 16 I/O, DIN rail	Wi-Fi, BLE, Ethernet, 2x RS485, RS232	External Wi-Fi/BLE (SMA)

1 EQSP32RE and EQSP32RE-X are available on custom order only. Product pages and ordering: erqos.com/shop

© 2026 Erqos Technologies P.C. · erqos.com

Contents

1 Description....1
2 Key Features....2
3 Applications....2
4 Ordering Information....2
5 Block Diagrams....4
6 Specifications....6
6.1 Model Comparison....6
6.2 System Specifications....6
6.3 Absolute Maximum Ratings....7
6.4 Recommended Operating Conditions....7
6.5 Certification and Compliance....8
7 Powering EQSP32....8
8 Visual and Audio Indications....9
8.1 LED Indications....9
9 Pinout....11
9.1 X1 – Screw Terminal Connector 1....11
9.2 X2 – Screw Terminal Connector 2....12
10 Power Management....12
10.1 Power Distribution and Monitoring....12
10.2 Integrated 5 V Power Supply....13
11 Multipurpose I/O and Power PWM Output (ADIO)....14
11.1 Equivalent ADIO Circuit....14
11.2 ADIO Modes Overview....14
11.3 Digital Input (DIN)....15
11.4 Switch Input (SWT) – Special DIN Mode....15
11.5 Pulse Capture Counter (PCC)....16
11.6 Quadrature Encoder (QENC)....17
11.7 Analog Input (AIN)....17
11.8 Current Input (CIN)....18
11.9 Relative Analog Input (RAIN) – Special AIN Mode....19
11.10 Temperature Input (TIN) – Special AIN Mode....19
11.11 Power PWM Output (POUT)....21
11.12 Relay (RELAY) – Special POUT Mode....22
12 Industrial Communications....24
12.1 Serial – RS232....24
12.2 Serial – RS485....25
12.3 CAN-Bus....26
12.4 Third-Party Libraries – Native Pin Control....27
13 Software Architecture....29
14 Programming, Configuration and Default Parameters....29
14.1 EQSP32Configs Structure....30
15 Wireless and Wired Internet Connectivity....33
15.1 Automatic Network Management....33
15.2 Wi-Fi Connectivity....33
15.3 Ethernet Connectivity (Model-Dependent)....33
15.4 LoRa Connectivity (EQSP32CL-X)....34
16 MQTT and Home Assistant....35
16.1 Enabling MQTT....35
16.2 Device Identity and Topics....35
16.3 Home Assistant Integration....36
17 Over-the-Air (OTA) Updates....37
18 Time, Uptime and Timers....37
18.1 Clock and Calendar....37
18.2 Uptime....37
18.3 Software Timers (EQTimer)....37
19 Device Provisioning....38
19.1 Bluetooth....38
20 Stored Settings....39
21 Hardware-in-the-Loop (HIL) Test Mode....39
22 Expansion Modules....40
22.1 Supported Modules....40
23 Boot Button and Reset....41
23.1 Reset....41
23.2 Boot....41
24 Mechanical Dimensions....42
25 Notices....43
26 Revision History....43

© 2026 Erqos Technologies P.C. · erqos.com

5 Block Diagrams

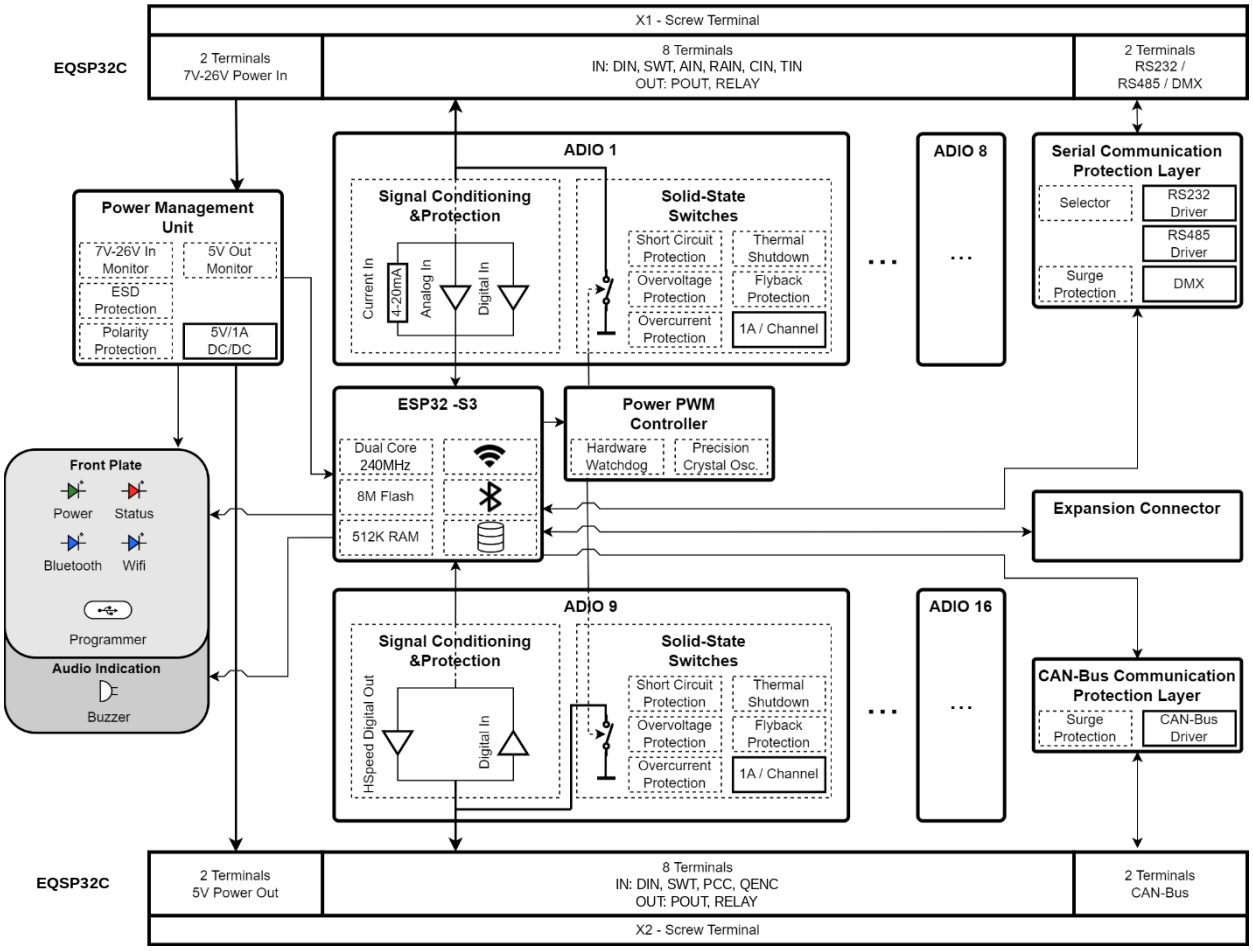

Figure 5-1. EQSP32C-X block diagram

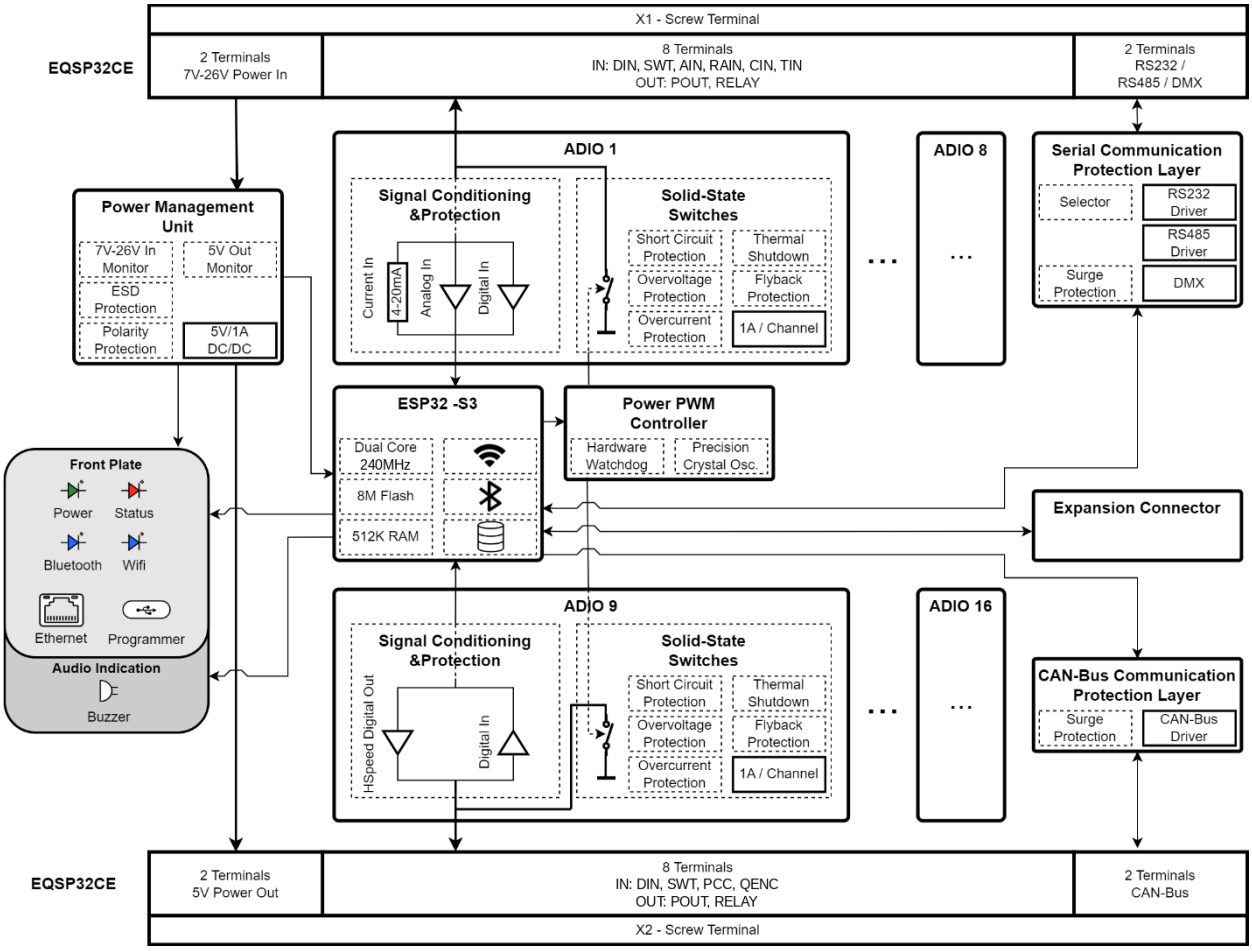

Figure 5-2. EQSP32CE block diagram

6 Specifications

6.1 Model Comparison

Table 6-1. Model Comparison

Module	EQSP32CE	EQSP32CE-X	EQSP32C-X	EQSP32CL-X	EQSP32RE *(custom order)*
Wired Connectivity					
Ethernet	✓	✓	No	No	✓
CAN Bus	✓	✓	✓	✓	No
Second RS485 port	No	No	No	No	✓
Wireless Connectivity					
LoRa	No	No	No	✓	No
Wi-Fi / Bluetooth antenna	Internal	External	External	Internal	Internal
LoRa antenna	–	–	–	External	–

EQSP32RE is available on custom order only.

6.2 System Specifications

Table 6-2. System Specifications

Module	Features
Software	Precompiled proprietary library for pin and peripheral modes and configurations Transparent connectivity handling using BLE, WiFi and Ethernet Interfacing entities for easy MQTT integrations
Compatibility	Home assistant integration using MQTT (Automatic MQTT device discovery) Compatible with Arduino IDE, VS Code (using PlatformIO) and OpenPLC
Processor	Dual-Core XTensa LX7 MCU 240 MHz frequency Wi-Fi, Bluetooth and MQTT run on a separate, dedicated core 512 KB RAM / 8 MB Flash memory
Power Management	Integrated DC/DC, 10 V to 26 V DC power input Supply output 5 V @ 1 A for powering external peripherals Input and output supply monitoring
I/O Terminals	16 x Multipurpose I/Os (digital/analog input/output) 5 V logic levels / 24 V tolerant Over voltage protection 8 x 12-bit ADCs
Power PWM Output	16 x 1 A low-side (open-collector) power PWM outputs Externally supplied up to VIn for more flexibility Integrated flyback diodes to VInput for inductive loads (like solenoids & relay coils) Short circuit, over current, over voltage and over temperature protection
Communication	RS232 RS485 (Half duplex); supports Modbus and DMX integrations CAN Bus (second RS485 port instead on EQSP32RE)

© 2026 Erqos Technologies P.C. · erqos.com

Module	Features
Indications	Green LED for power indication
	Red LED for device status
	2 x Blue LEDs for BLE and WiFi connectivity status
	Buzzer for audio indications
Mounting	Easy standard DIN rail push-fit mounting with spring mechanism
Expandability	Push fit expansion module mounting
	Specialized expansion modules for niche applications and system extension
	Automatic module enumeration, detection and characterization

6.3 Absolute Maximum Ratings

Table 6-3. Absolute Maximum Ratings

Pin	Type	Min.	Typ.	Max.	Unit
Supply In + to GND	Power In			28	V
5V Out + to GND	Power Out			2	A
Tx / A to GND	In/Out	-15		15	V
Rx / B to GND					
CAN H to GND	In/Out	-21		21	V
CAN L to GND					
ADIO to GND	In/Out	-0.5		VIn (*)	V
	Out			1.3	A

> **WARNING**
> (*) ADIO voltage should never exceed the VIn voltage of EQSP for steady states, or else the flyback diodes will conduct continuously!

6.4 Recommended Operating Conditions

Table 6-4. Recommended Operating Conditions

Pin	Type	Min.	Typ.	Max.	Unit
Supply In +	Power In	10	12 to 24	26	V
5V Out +	Power Out	4.98	5	5.02	V
				1	A
RS232 Tx	Out	-12		12	V
RS232 Rx	In	-12		12	
RS485 A	In/Out	0		5	V
RS485 B					
CAN H	Out (dominant)	2.75	3.5	4.5	V
CAN L	Out (dominant)	0.5	1.5	2.25	
CAN H – CAN L	In (differential)	0.5	0.7	0.9	

© 2026 Erqos Technologies P.C. · erqos.com

Pin		Type	Min.	Typ.	Max.	Unit
ADIO	High (1), terminals 1-8[4] High (1), terminals 9-16[4]	In/Out	3 5	5 to VIn 5 to VIn	VIn (*) VIn (*)	V
	Low (0)[4]		-0.5	0	0.6	
	Analog	In	0		10	
	Power PWM	Out			1	A
	AIN impedance	In		51[1]		kΩ
	CIN input resistance	In		470[2]		Ω
DIO	PCC / QENC frequency	In			200[3]	kHz

[1] 0-10 V. Above about 10.5 V a protection diode conducts and the input impedance drops.

[2] 9.4 V across the input at 20 mA. The loop supply must cover this plus the transmitter's own minimum voltage.

[3] Per channel, 50 % duty cycle, in PCC and QENC modes.

[4] In DIN and SWT modes, and in PCC and QENC modes on terminals 9-16. For native access see Native Pin Control.

6.5 Certification and Compliance

Applicable Standards for CE Compliance under Directive 2014/53/EU (RED):

Table 6-5. Certification and Compliance

Area	Applicable Standard(s)
Safety (Electrical)	EN IEC 62368-1:2020 + A11:2020
Outdoor Safety	EN 62368-1 Annex Y
RF Exposure / Health	EN IEC 62311:2020 EN 50566:2017 EN 62479:2010 EN 50663:2017 EN 50665
EMC (General/IT)	EN 55032 EN 55024 (or EN 55035)
EMC (Wireless)	ETSI EN 301 489-1 V2.2.3 ETSI EN 301 489-17 V3.2.4
Industrial EMC	ETSI EN 303 446-1 V1.2.1 (references EN 61000-6-2, EN 61000-6-4, and IT EMC standards)
Radio (WiFi/BLE)	ETSI EN 300 328 V2.2.2 (Full + Spurious only) ETSI EN 300 440 V2.2.1 (Full + Spurious only) ETSI EN 301 893 V2.1.1 (Spurious only)
Environmental	Directive 2011/65/EU (RoHS)

7 Powering EQSP32

To power the EQSP32 from the grid an AC/DC converter is needed. It is recommended that minimum of 5 W AC/DC power supply is used to power EQSP32 main module.

For driving additional devices and loads either directly from the AC/DC or from EQSP32's 5VDC out, the power supply's rating must be calculated accordingly to meet or exceed the total power demand of the application.

For safety, switch off VIn before connecting the USB-C cable for programming, and program the EQSP32 on USB power only. Also switch off any separately powered loads: a new or untested program could switch them unexpectedly.

For safety and ease of use, it is strongly recommended that a main circuit breaker is used to completely isolate the AC/DC from the grid.

© 2026 Erqos Technologies P.C. · erqos.com

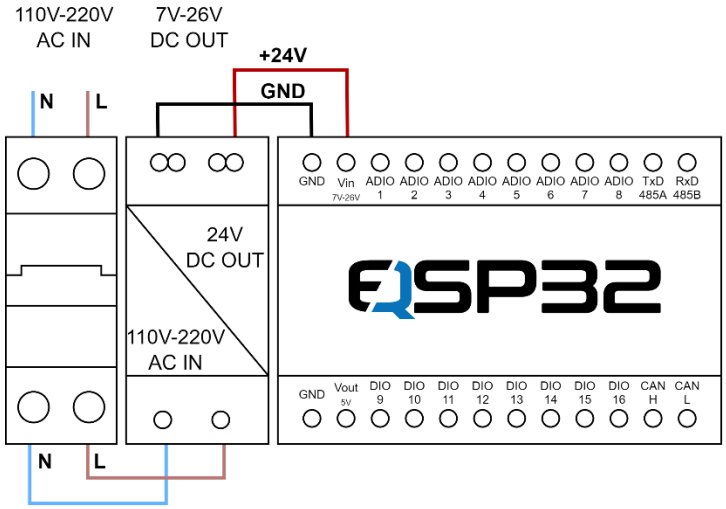

Figure 7-1. Circuit breaker and AC/DC wiring

> **WARNING**
> Make sure that the appropriate ground fault circuit breaker, fuses or circuit breakers and any other necessary protective components are used if there is any risk of user electrocution.

8 Visual and Audio Indications

While the proprietary library is running, the EQ System Supervisor will monitor EQSP32's status and Bluetooth/WiFi status.

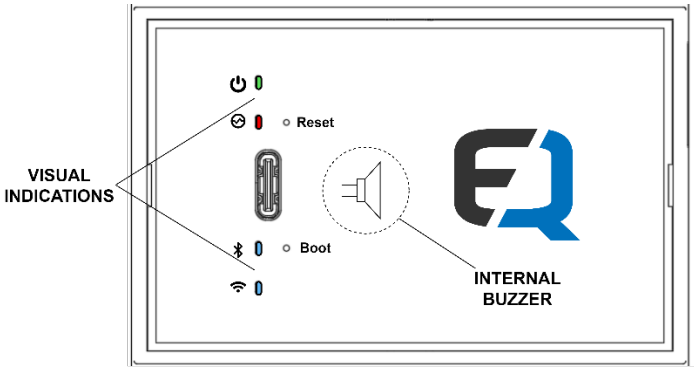

Figure 8-1. Front panel: LEDs, buttons and buzzer

The visual indicator (LEDs) will be automatically updated to reflect the device's state.

The internal buzzer will sound two short tones at the end of start-up, indicating that the library is running. The user may utilize the audio indication in their program from the library's API. During supply over-voltage the buzzer sounds an alarm tone.

8.1 LED Indications

Table 8-1. LED indications

State	Meaning	Pattern
Power LED		
On	Powered on	
Off	No power	

State	Meaning	Pattern
Status LED		
Off	No errors	
Blink	Supply under-voltage	
Flashing	Supply over-voltage	
Blink	Internal fault	
Long + 2	Expansion module not compatible	
On	Timezone not configured	
Bluetooth LED		
Off	Bluetooth disconnected	
Flashing	Bluetooth advertising	
On	Bluetooth connected	
WiFi LED		
Off	WiFi disconnected	
Flashing	WiFi scanning or trying to connect	
On	WiFi connected	

The Status LED shows one condition at a time. If several are present, it shows the first of: supply over-voltage, expansion module not compatible, time zone not configured, supply under-voltage, internal fault.

The buzzer and, when the library does not manage them, the Bluetooth and WiFi LEDs can be driven by the application:

Table 8-2. Library reference: LED Indications

Usage	
`eqsp32.buzzerOn(ff, dd)`	Turns the buzzer on ff: frequency in Hz, 50 to 20000 (optional, default = 500) dd: duration in ms (optional, default = 0: on until `eqsp32.buzzerOff()`)
`eqsp32.buzzerOff()`	Turns the buzzer off
`eqsp32.setBleLed()`, `eqsp32.resetBleLed()`, `eqsp32.toggleBleLed()`	Turn the Bluetooth LED on, off, or toggle it Only with `disableErqosIoT` = true, when the library does not manage the LED
`eqsp32.setWifiLed()`, `eqsp32.resetWifiLed()`, `eqsp32.toggleWifiLed()`	Turn the WiFi LED on, off, or toggle it Only with `disableErqosIoT` = true, when the library does not manage the LED

9 Pinout

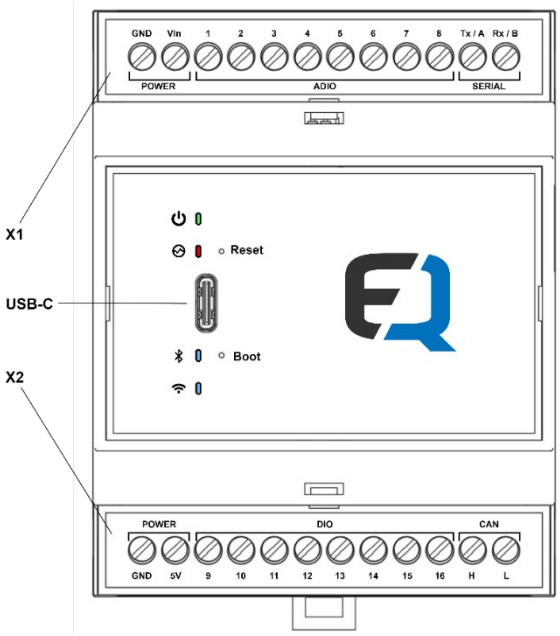

Figure 9-1. EQSP32 terminals and front panel

X1 – Screw terminal connector 1

USB-C – Programmer port

X2 – Screw terminal connector 2

9.1 X1 – Screw Terminal Connector 1

X1 connector accommodates the power supply input pins, the serial communication and 8 ADIO pins.

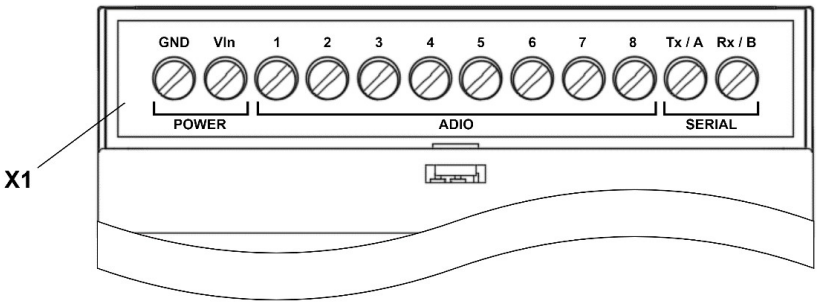

Figure 9-2. X1 terminals

Table 9-1. X1 – Screw Terminal Connector 1

Name	Pin	Function	Type
X1	GND	DC Power Input - (GND) (*)	POWER IN
	VIn	DC Power Input + (10-26 V)	POWER IN
	1, …, 8	Multipurpose I/O (**)	ADIO
	Tx / A	RS232 Tx / RS485 A	SERIAL
	Rx / B	RS232 Rx / RS485 B	SERIAL

NOTE
(*) DC Power Input GND is common with DC Power Output GND
(**) X1 ADIO pins also support analog input (AIN) functionality.

9.2 X2 – Screw Terminal Connector 2

X2 connector accommodates the power supply output pins, the CAN-Bus communication and 8 ADIO pins.

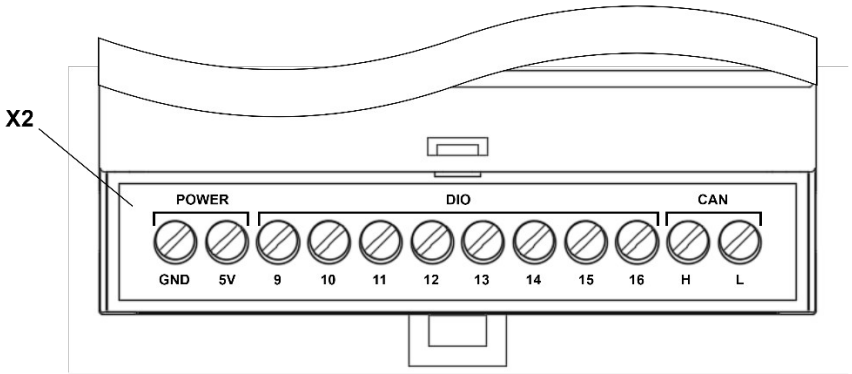

Figure 9-3. X2 terminals

Table 9-2. X2 – Screw Terminal Connector 2

Name	Pin	Function	Type
X2	GND	DC Power Output - (GND) (*)	POWER OUT
	5V	DC Power Output + (5 V)	POWER OUT
	9, …, 16	Multipurpose I/O (**)	DIO
	H	CAN H (***)	CAN-Bus
	L	CAN L (***)	CAN-Bus

NOTE

(*) DC Power Output GND is common with DC Power Input GND

(**) X2 DIO pins also support pulse counting (PCC), and quadrature encoder inputs (QENC) on the pairs 9/10, 11/12, 13/14 and 15/16.

(***) For the EQSP32RE and RE-X, CAN H and CAN L are replaced by the second RS485 port's terminals, 485A2 and 485B2 respectively.

10 Power Management

10.1 Power Distribution and Monitoring

The EQSP32, continuously monitors the power supply's input voltage and the internally generated power supply 5 V output. The 5 V output supply is also used as a voltage reference, for example when connecting a thermistor, so internal monitoring results in more accurate measurements.

Supply under-voltage (VIn below 7 V) and over-voltage (VIn above 26 V) are shown on the Status LED; over-voltage also sounds the buzzer.

EQSP32 is also powered from the USB 5 V but this voltage is only internally used. When programming over USB-C, switch off VIn first (see Powering EQSP32). The 5V Out supply is powered only when VIn is present: on USB power alone, 5V Out and the devices connected to it are off.

© 2026 Erqos Technologies P.C. · erqos.com

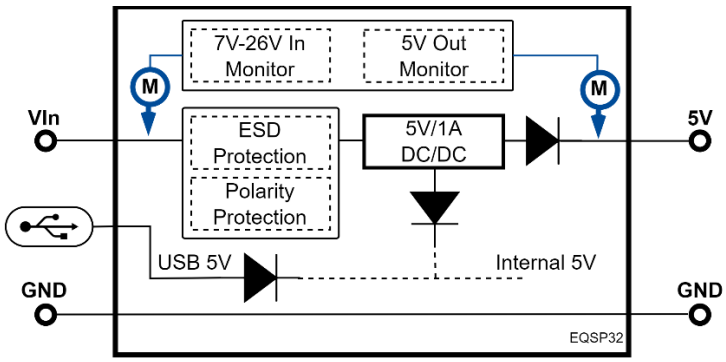

Figure 10-1. Power distribution and monitoring points (M)

Table 10-1. Library reference: Power Distribution and Monitoring

Usage	
`eqsp32.readInputVoltage()`	Returns the measured voltage on VIn in mV; -1 if the application holds the ESP32's ADC1 at start-up
`eqsp32.readOutputVoltage()`	Returns the measured voltage on VOut in mV; -1 if the application holds the ESP32's ADC1 at start-up

10.2 Integrated 5 V Power Supply

Any number of external devices, loads and sensors may be powered directly from the integrated 5 V power supply, as long as the combined continuous current consumption is up to 1 A.

The integrated 5 V power supply is heavily filtered and monitored. The user has access to the internal 5 V analog readings and it is recommended for best accuracy that this supply is used as reference.

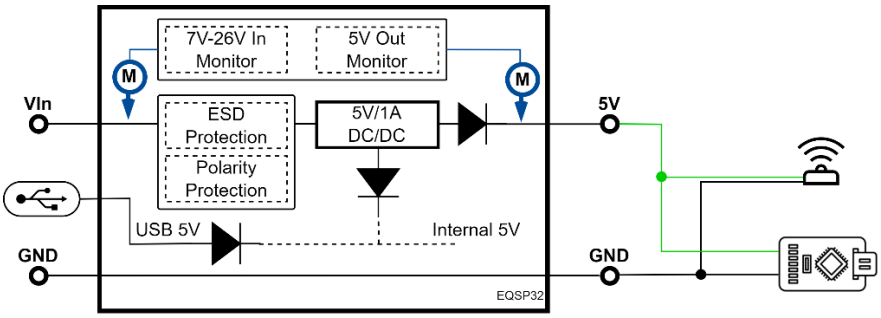

Figure 10-2. Example: Powering external devices and sensors using integrated 5 V power supply

11 Multipurpose I/O and Power PWM Output (ADIO)

11.1 Equivalent ADIO Circuit

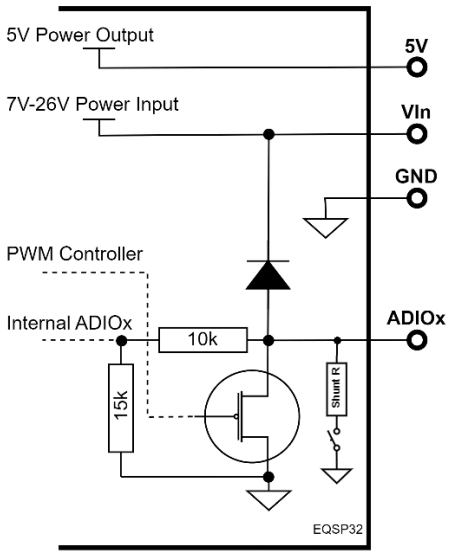

Figure 11-1. Equivalent ADIO Circuit

On terminals 1–8 in CIN mode, a shunt resistor connects the terminal to GND. The over-current protection disconnects it.

11.2 ADIO Modes Overview

All ADIO pins support DIN and POUT modes. In X1, pins 1-8 also support AIN mode.

In addition to the main/core modes, there are some extra special modes, which extend the pin's functionality. For the pin to support each special mode, it has to support the corresponding main mode.

Table 11-1. ADIO Modes Overview

Mode	Type	Function
Digital Input (DIN)	Input	Digital
Switch Input (SWT) – {Special DIN}	Input	Digital
Pulse Capture Counter (PCC) – {DIO 9, …, 16}	Input	Digital
Quadrature Encoder (QENC) – {DIO 9/10, 11/12, 13/14, 15/16}	Input	Digital
Analog Input (AIN) – {ADIO 1, …, 8}	Input	Analog
Current Input (CIN) – {ADIO 1, …, 8}	Input	Analog
Relative Analog Input (RAIN) – {Special AIN}	Input	Analog
Temperature Input (TIN) – {Special AIN}	Input	Analog
Power PWM Output (POUT)	Output	PWM
Relay (RELAY) – {Special POUT}	Output	PWM

After `eqsp32.begin()` no terminal has a mode: outputs are off and inputs are not read. `eqsp32.readPin()` returns -1 until a mode is set with `eqsp32.pinMode()`.

`eqsp32.pinMode()` returns false and does not apply the mode when the mode is not available on that terminal: the analog modes (AIN, RAIN, TIN, CIN) on terminals 9-16, PCC on terminals 1-8, and QENC on terminals other than 9, 11, 13 and 15. Every change of mode switches the terminal's output off. `eqsp32.readMode(nn)` returns the mode a terminal is in.

The EQ System Supervisor samples all inputs in the background. `eqsp32.readPin()` returns the most recent value and does not wait for a new measurement.

© 2026 Erqos Technologies P.C. · erqos.com

For direct control of a terminal's pin by the application, for example with a third-party library, use `eqsp32.getPin()` (see Third-Party Libraries – Native Pin Control).

11.3 Digital Input (DIN)

When in digital input mode (DIN), the pin reads 1 (HIGH) if a voltage between 3 V and VIn is applied on terminals 1-8, or between 5 V and VIn on terminals 9-16, and 0 (LOW) if the voltage is between 0 V and 0.6 V.

When a trigger mode other than STATE is used, the read value is latched and automatically cleared after being read.

- `ON_RISING` – latches true when at least one rising edge is detected since the last read; resets to false after being read.
- `ON_FALLING` – latches true when at least one falling edge is detected since the last read; resets to false after being read.
- `ON_TOGGLE` – latches true when any edge (rising or falling) is detected since the last read; resets to false after being read.

Table 11-2. Library reference: Digital Input (DIN)

Configuration	
`eqsp32.pinMode(nn, DIN)`	nn: pin index
Usage	
`eqsp32.readPin(nn, mm)`	Returns HIGH/LOW based on trigger mode nn: pin index mm: trigger mode (optional, default = STATE)

11.4 Switch Input (SWT) – Special DIN Mode

This is a special DIN mode. Since all pins support digital input (DIN), any pin may operate in SWT mode. In this mode, the input is debounced using the specified debounce time. If no debounce time is provided, the default value of 100 ms is used.

For a change in state to be registered, the pin value must remain stable for at least the debounce time. If the pin value fluctuates (bounces) more frequently than the debounce time, the last stable logic level is returned by the `eqsp32.readPin(nn, mm)` call.

The configured trigger mode also applies in SWT mode:

- STATE – returns the current debounced logic level.
- `ON_RISING` – latches true when the debounced input goes from low to high; resets to false after being read until the next rising edge.
- `ON_FALLING` – latches true when the debounced input goes from high to low; resets to false after being read until the next falling edge.
- `ON_TOGGLE` – latches true whenever the debounced input changes state (rising or falling); resets to false after being read until the next transition.

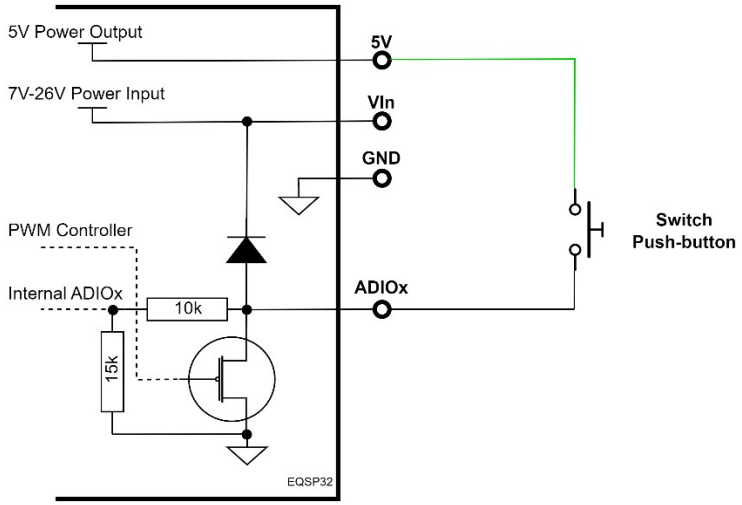

Figure 11-2. Example: Push-button on EQSP32's 5V out

Table 11-3. Library reference: Switch Input (SWT) – Special DIN Mode

Configuration	
`eqsp32.pinMode(nn, SWT)`	nn: pin index
`eqsp32.configSWT(nn, tt) (*)`	nn: pin index tt: debounce ms (optional, default = 100)
Usage	
`eqsp32.readPin(nn, mm)`	Returns HIGH/LOW based on trigger mode nn: pin index mm: trigger mode (optional, default = STATE)

> **NOTE**
> (*) If this function is not called, the default values are applied, meaning the SWT pin's debounce time will be set to 100 ms. Call it after `eqsp32.pinMode(nn, SWT)`: setting the mode loads the default value again.

nn: Is the pin index and it can be in range [1, 16].

tt: Is the debounce time in ms. This is an optional parameter with default value of 100 ms, if omitted.

mm: This parameter sets the trigger mode while reading the pin. It is an optional parameter and the default value is STATE (returning the pin HIGH/LOW state) if omitted.

11.5 Pulse Capture Counter (PCC)

The pulse capture count mode (PCC) measures the number of digital pulses on supported terminal pins (9–16) using hardware pulse counters. The function `eqsp32.readPin(pinIndex)` returns the number of pulses detected since the last read, after which the counter automatically resets to zero (default). Up to 250,000,000 pulses are counted between two reads. Maximum input frequency: 200 kHz (50 % duty cycle). Pulses shorter than 2 µs are rejected as noise.

With `eqsp32.configPCC(nn, mm, false)` the terminal keeps a running total instead: reading does not clear it, the total counts from 0 to 2,147,483,647 and then starts again from 0, and `eqsp32.pinValue(nn, vv)` sets it to zero or to a preset value.

Pulse capture counter (PCC) mode is supported on up to four of terminals 9–16 at the same time. The four hardware pulse counters are shared with the quadrature encoder mode (QENC), which uses one counter per terminal pair. When no counter is free, `eqsp32.pinMode()` returns false and the mode is not applied. A refused `eqsp32.pinMode(nn, PCC)` has already switched the terminal's output off. Application code that uses the pulse counters itself must read them by polling: if it installs the pulse-counter interrupt, `eqsp32.pinMode()` refuses PCC and QENC.

The counting behavior depends on the selected trigger mode:

- `ON_RISING` – counts rising edges (default).
- `ON_FALLING` – counts falling edges.
- `ON_TOGGLE` – counts both rising and falling edges.

If an unsupported or invalid mode is specified, the configuration fails and returns false.

Power the EQSP32 through VIn when using this mode. On USB power alone, the first counts after an idle period may be inaccurate.

Table 11-4. Library reference: Pulse Capture Counter (PCC)

Configuration	
`eqsp32.pinMode(nn, PCC)`	nn: pin index
`eqsp32.configPCC(nn, mm, cc)`	nn: pin index mm: pulse capture counter trigger mode, `ON_RISING` (default), `ON_FALLING` or `ON_TOGGLE`. cc: true (default) clears the count on every read; false keeps a running total. Call it after `eqsp32.pinMode(nn, PCC)`: switching a terminal to PCC from another mode sets `ON_RISING` and cc true again. It returns false if the terminal is not in PCC mode. Changing cc keeps the count.

© 2026 Erqos Technologies P.C. · erqos.com

Usage	
`eqsp32.readPin(nn)`	Returns the accumulated pulse counts since last read. With cc false, returns the running total without clearing it. nn: pin index
`eqsp32.pinValue(nn, vv)`	Sets the running total to vv; 0 zeroes it. Only with cc false. nn: pin index vv: running total, 0 to 2,147,483,647

11.6 Quadrature Encoder (QENC)

The quadrature encoder mode (QENC) counts the position of an incremental encoder on a pair of terminals: 9/10, 11/12, 13/14 or 15/16. `eqsp32.pinMode()` is called on the first terminal of the pair (A input), and the second terminal becomes the B input. `eqsp32.readPin()` on either terminal returns the signed position; reading does not clear it. `eqsp32.pinValue()` on either terminal sets the position to zero or to a preset value, without losing counts. A reading of -1 is a valid position, not an error.

Each pair uses one of the four hardware pulse counters, which are shared with the PCC mode. The position wraps around at the end of its range, and the difference of two readings stays correct across the wrap. Setting another mode on either terminal of the pair, including CUSTOM through `eqsp32.getPin()`, releases the encoder and leaves the other terminal in DIN mode. If the application has installed the pulse-counter interrupt, `eqsp32.pinMode(nn, QENC)` is refused; the second terminal is then already in DIN mode and the first terminal's output is off. Maximum input frequency: 200 kHz per channel (50 % duty cycle). Signal levels: High 5 V to VIn, Low 0 V to 0.6 V.

The counting format is set with `eqsp32.configQENC()`:

- `QENC_X4` – encoder A/B signals, 4 counts per signal period (default).
- `QENC_X2` – encoder A/B signals, 2 counts per signal period.
- `QENC_X1` – encoder A/B signals, 1 count per signal period.
- `QENC_PULSE_DIR` – pulses on the first terminal, direction level on the second (high counts up).
- `QENC_UP_DOWN` – pulses on the first terminal count up, pulses on the second count down.

Power the EQSP32 through VIn when using this mode. On USB power alone, the first counts after an idle period may be inaccurate.

Table 11-5. Library reference: Quadrature Encoder (QENC)

Configuration	
`eqsp32.pinMode(nn, QENC)`	nn: first terminal of the pair: 9, 11, 13 or 15
`eqsp32.configQENC(nn, ff, rr)`	nn: either terminal of the pair ff: format, `QENC_X4` (default), `QENC_X2`, `QENC_X1`, `QENC_PULSE_DIR` or `QENC_UP_DOWN` rr: true reverses the counting direction (default false) Call it after `eqsp32.pinMode(nn, QENC)`: a newly formed pair starts at `QENC_X4`, not reversed. It returns false if the pair is not in QENC mode. Changing between an encoder format and a pulse format keeps the position; changing the resolution (`QENC_X4`, `QENC_X2`, `QENC_X1`) rescales it.
Usage	
`eqsp32.readPin(nn)`	Returns the signed position nn: either terminal of the pair
`eqsp32.pinValue(nn, vv)`	Sets the position to vv; 0 zeroes it nn: either terminal of the pair vv: position, within the range of the format set with `eqsp32.configQENC()`

11.7 Analog Input (AIN)

The analog input mode (AIN), measures the analog value of the pin from 0 V to 10 V and returns the analog value in mV.

© 2026 Erqos Technologies P.C. · erqos.com

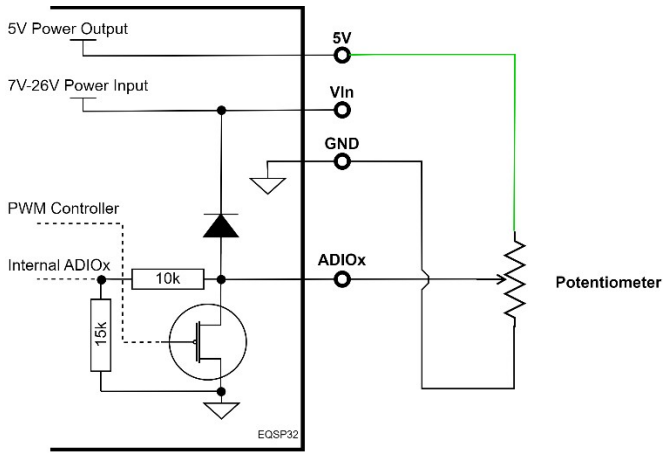

Figure 11-3. Example: Potentiometer with EQSP32's 5V out as reference voltage

Table 11-6. Library reference: Analog Input (AIN)

Configuration	
`eqsp32.pinMode(nn, AIN)`	nn: pin index
Usage	
`eqsp32.readPin(nn)`	Returns analog value in mV nn: pin index
Macros	
mV = `eqsp32.readPin(nn)`	
`CONVERT_AIN( mV )`	Converts the int analog input value mV from mV to float V mV: Is the read AIN in mV

11.8 Current Input (CIN)

The current input mode (CIN) measures 0–20 mA and 4–20 mA current loop signals and returns the value in units of mA x100 (for example, 1234 corresponds to 12.34 mA).

CIN mode is supported on pins 1–8.

If the input current stays above 21 mA for an accumulated 250 ms, the automatic overcurrent protection disconnects the shunt resistor to prevent damage and the function returns -1 (`CIN_OC_ERROR`). After 2.25 seconds the shunt resistor is reconnected and the measurement resumes. If the current is still above 21 mA, the cycle repeats until normal operating conditions are restored.

Table 11-7. Library reference: Current Input (CIN)

Configuration	
`eqsp32.pinMode(nn, CIN)`	nn: pin index
Usage	
`eqsp32.readPin(nn)`	Returns analog value in 100x mA nn: pin index
Macros	
mA100 = `eqsp32.readPin(nn)`	
`CONVERT_CIN( mA100 )`	Converts the int current input value mA100 from 100x mA to float mA mA100: Is the read CIN in 100x mA

11.9 Relative Analog Input (RAIN) – Special AIN Mode

This is a special AIN mode. Only pins 1-8 may operate as RAIN, since only these pins support analog input mode (AIN). In this mode the analog value is automatically converted into 0-1000 range with 0 being the GND value and 1000 being the 100% of the reference 5 V output supply voltage. While the 5 V output is below 4.5 V, the reading is 0.

This is a very helpful mode to directly convert the analog input into a usable value. A simple example would to have 3 potentiometers to control the PWM values of an RGB LED strip. By configuring the potentiometers' pins to RAIN mode, the values may be directly passed as control values for the 3 POUT pins to drive the RGB strip.

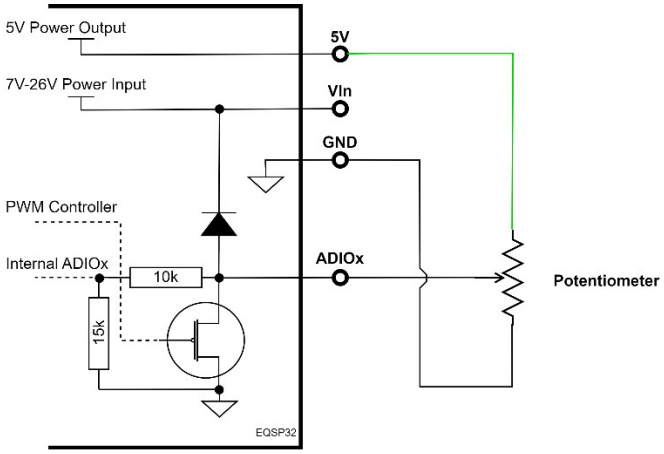

Figure 11-4. Example: Potentiometer with EQSP32's 5V out as reference voltage

Table 11-8. Library reference: Relative Analog Input (RAIN) – Special AIN Mode

Configuration	
`eqsp32.pinMode(nn, RAIN)`	nn: pin index
Usage	
`eqsp32.readPin(nn)`	Returns a 0 to 1000 value representing 0% to 100% of the 5 V supply out reference nn: pin index

11.10 Temperature Input (TIN) – Special AIN Mode

This is a special AIN mode. Only pins 1-8 may operate as TIN, since only these pins support analog input mode (AIN). In this mode the analog value is automatically converted into temperature in degrees Celsius.

Although the `eqsp32.configTIN()` call is optional in pin setup, the temperature measurement will not be accurate unless the respective NTC thermistor parameters are configured using `eqsp32.configTIN()`.

It is strongly recommended to call `eqsp32.configTIN()`, configuring the NTC beta and reference resistance values, when setting up the pin in TIN mode.

Besides the Beta model, `eqsp32.configTIN()` also accepts the Steinhart-Hart coefficients (A, B, C), or the normalized Steinhart-Hart coefficients (nA, nB, nC) with the reference resistance, as given in the thermistor datasheet. The temperature reading is updated about every 0.3 s.

© 2026 Erqos Technologies P.C. · erqos.com

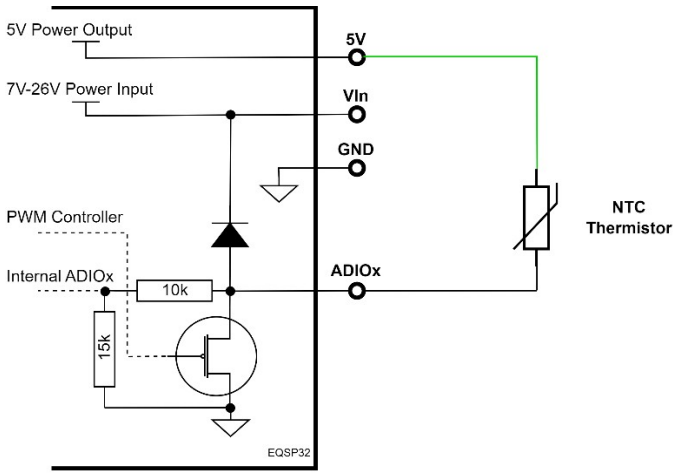

Figure 11-5. Example: NTC Thermistor using EQSP32's 5V out as reference voltage

Table 11-9. Library reference: Temperature Input (TIN) – Special AIN Mode

Configuration	
`eqsp32.pinMode(nn, TIN)`	nn: pin index
`eqsp32.configTIN(nn, bb, rr) (*)`	nn: pin index bb: NTC beta value (optional, default = 3435) rr: NTC resistance at 25C (optional, default = 10000)
`eqsp32.configTIN(nn, A, B, C)`	Steinhart-Hart form A, B, C: Steinhart-Hart coefficients from the thermistor datasheet
`eqsp32.configTIN(nn, nA, nB, nC, rr)`	Normalized Steinhart-Hart form: $1/T = nA + nB \cdot \ln(R/rr) + nC \cdot (\ln(R/rr))^3$, T in K nA, nB, nC: normalized coefficients from the thermistor datasheet rr: NTC resistance at 25C
Usage	
`eqsp32.readPin(nn)`	Returns the temperature in 10x C nn: pin index
Macros	
`t = eqsp32.readPin(nn)`	
`IS_TIN_VALID( t )`	Returns true if read temperature is valid and no sensor errors have been detected. Read temperature may equal to TIN_OPEN_CIRCUIT or TIN_SHORT_CIRCUIT if not valid. t: Is the read TIN temperature (or error value)
`CONVERT_TIN( t )`	Converts the int temperature t from 10x C to float C t: Is the read TIN temperature
Constants	
`t = eqsp32.readPin(nn)`	
`TIN_OPEN_CIRCUIT`	Read temperature will equal to TIN_OPEN_CIRCUIT if open circuit sensor error is detected.
`TIN_SHORT_CIRCUIT`	Read temperature will equal to TIN_SHORT_CIRCUIT if shorted sensor error is detected.

NOTE
(*) If this function is not called, the default values are applied, meaning for the TIN the NTC beta is set to 3435 and its reference resistance to 10000.

nn: Is the pin index and it can be in range [1, 8].

bb: Is the NTC's beta value. This is an optional parameter with default value of 3435, if omitted.

rr: Is the NTC's reference resistance parameter. It is an optional parameter and the default value is 10000, if omitted.

11.11 Power PWM Output (POUT)

The power PWM output mode (POUT) is meant to be used to drive up to 1 A loads per terminal. In this mode the pin operates as a low-side (open-collector) PWM output. The duty cycle is set from 0 to 1000, in steps of 0.1 %. All outputs are off after `eqsp32.begin()` and after every change of mode.

Inductive and Non-Inductive loads

EQSP32 incorporates protective diodes for protecting the mosfets from the overvoltage spikes induced by the switching of relays or solenoids. Thus, inductive and non-inductive loads may be connected on the power PWM outputs in the same way.

The following diagrams demonstrate various ways to use the ADIO in power PWM mode.

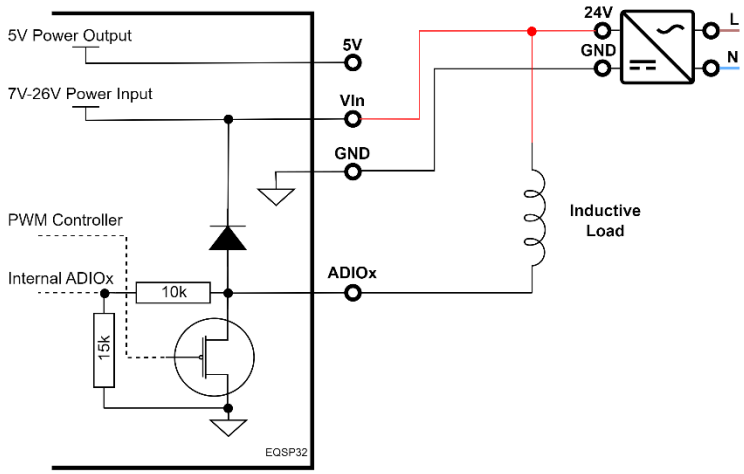

Figure 11-6. Example 1: Driving an inductive load on a common power supply

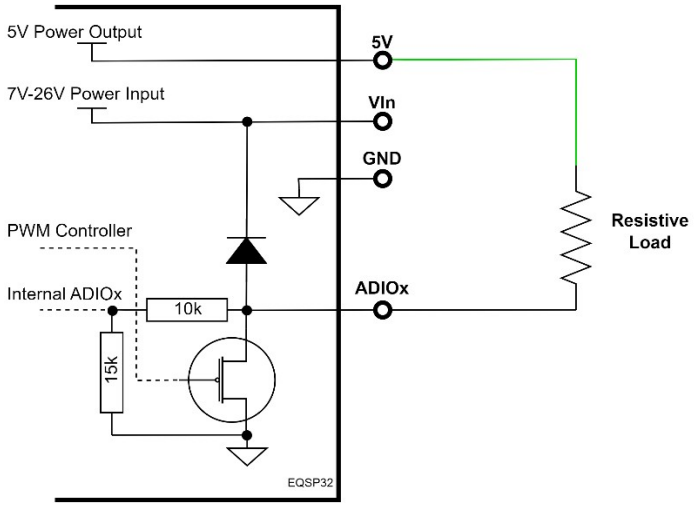

Figure 11-7. Example 2: Driving a resistive load from the integrated 5V Out power supply

Driving LED Strips

On terminals 1–8, an optional turn-off resistor across the LED strip, from the strip's + supply to the output terminal, keeps the strip fully off, as in the example below. The input circuit of these terminals lets a small current flow through the strip while the output is off, which can make LEDs glow faintly. Terminals 9–16 do not need it.

The value of the resistor depends on the efficiency of the LEDs, their number and other factors. A 4.7 kΩ, 0.25 W resistor suits most strips up to 24 V.

Example: In the following example a 4.7 kΩ resistor is used. Assuming our system runs on 24 V but our LED strip is rated for 12 V, we may use a separate power supply for our LEDs.

© 2026 Erqos Technologies P.C. · erqos.com

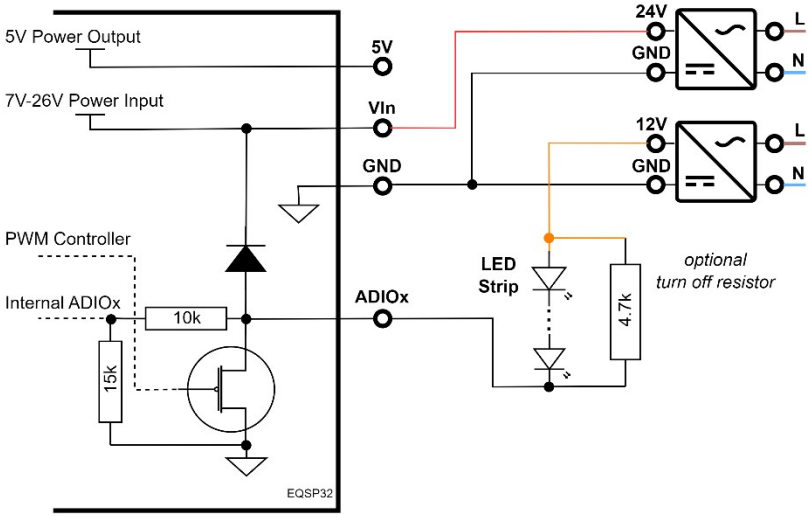

Figure 11-8. Example 3: Driving a 12 V LED strip from separate 12 V power supply

Table 11-10. Library reference: Power PWM Output (POUT)

Configuration	
`eqsp32.pinMode(nn, POUT)`	nn: pin index
`eqsp32.configPOUTFreq(ff) (*)`	ff: frequency in Hz for all power PWM output (POUT) channels (and special POUT like RELAY). Set frequency may range from 50 to 3000 Hz. Default is 1000 Hz.
Usage	
`eqsp32.pinValue(nn, pp)`	nn: pin index pp: power per 1000 (500 for 50%, 1000 is for 100%)
`eqsp32.readPin(nn)`	Returns the last power value written with `eqsp32.pinValue()` (0 to 1000) nn: pin index

NOTE
(*) Configuring the POUT frequency affects all POUT and special POUT (ex. RELAY) channels. All pins in POUT or special POUT modes share the same PWM frequency.

11.12 Relay (RELAY) – Special POUT Mode

This is a special POUT mode. All pins may operate as RELAY, since all pins support Power PWM output mode (POUT). The unique feature of this mode is the automatic power derating to a specified value after a specified time. This drastically reduces the heat and power consumption on the relays, allowing for more relays to be driven from a smaller power supply.

If the minimum holding voltage (MHV) of the relay is not provided and only the nominal or trigger voltage is, then the holding voltage may be experimentally found.
One way to achieve this is by connecting the relay's coil on an adjustable power supply and apply the nominal voltage. Once the relay is triggered, we slowly reduce the applied voltage until the relay disengages. After finding the voltage on which the relay disengages, we calculate the MHV by the following equation:

Minimum holding PWM ratio = (Vdisengage / Vsupply)

NOTE
Since the PWM range is from 0 to 1000, reflecting to 0% to 100% we need to multiply the MHV ratio with 1000.

Minimum holding PWM value = (Vdisengage / Vsupply) * 1000

Example: We have a 24 V relay, so for our test we will also use a 24 V supply. We measure that the relay disengages at 6 V after slowly reducing from 24 V.

Minimum holding PWM value = (6 V / 24 V) * 1000 = 250

© 2026 Erqos Technologies P.C. · erqos.com

To be on the safe side, we don't want to be exactly on the limit, so we will set the relay's holding voltage at 50% more than the disengaging voltage.

Desired holding PWM value = 250 + 250 x ½ = 375

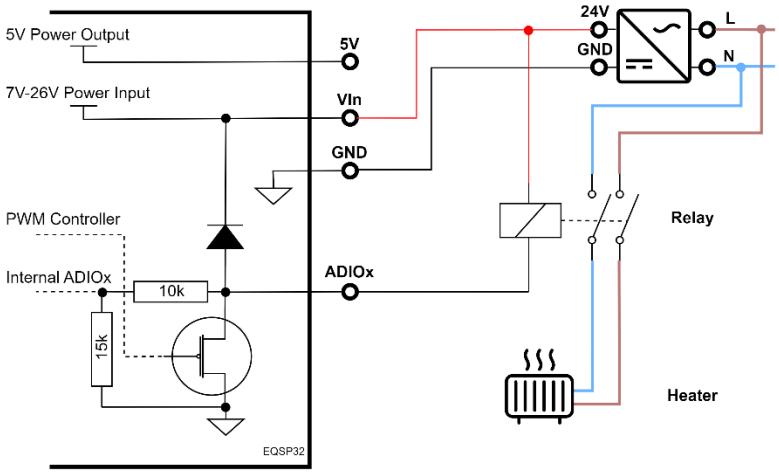

Figure 11-9. Example: EQSP32 controlling a 24 V Relay to drive a 220 V heater

Table 11-11. Library reference: Relay (RELAY) – Special POUT Mode

Configuration	
`eqsp32.pinMode(nn, RELAY)`	nn: pin index
`eqsp32.configRELAY(nn, hh, tt) (*)`	nn: pin index hh: hold power (optional, default = 500) tt: derate time ms (optional, default = 1000)
Usage	
`eqsp32.pinValue(nn, pp)`	nn: pin index pp: initial power (1000 is for 100%)

NOTE
(*) If this function is not called, the default values are applied, meaning the RELAY pin's hold power will be set to 50% (hh = 500) and the derate time will be set to 1000 (1 second). Call it after `eqsp32.pinMode(nn, RELAY)`: setting the mode loads the default values again.

nn: Is the pin index and it can be in range [1, 16].

hh: Is the holding power of the relay after the derate time has passed. The value's range is [0, 1000] and it corresponds to 0% - 100%. This is an optional parameter with default value of 500.

tt: Is the time after which the relay's power will drop to holding power in ms. This is an optional parameter with default value of 1000 ms, if omitted.

pp: This parameter sets the trigger power of the relay. This parameter's range is [0,1000]. The relay will be triggered at this percentage of its power and it will keep this value until derate time has passed. After the derate time has passed, the relay's power will drop to holding power until it is set to 0 by the user's program.

© 2026 Erqos Technologies P.C. · erqos.com

12 Industrial Communications

12.1 Serial – RS232

Standard RS232

For standard RS232 communication, connect EQSP32's RS232 Tx pin with receiver's RS232 Rx pin and vice versa. The serial port uses 8 data bits, no parity and 1 stop bit (8N1).

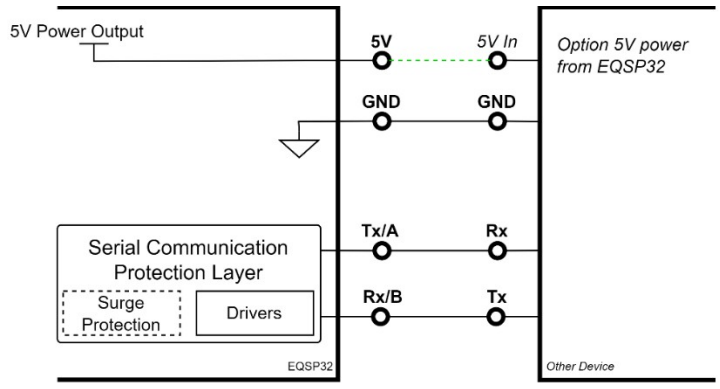

Figure 12-1. RS232 wiring

RS232 to TTL

To communicate using EQSP32's RS232 port with a TTL level device, like an Arduino, the RS232 Tx signal must be clipped between GND and 5 V.

Use external clipping diodes connecting the RS232 Tx signal to TTL device's 5 V and GND like shown in the following diagram. To restrict the current flow from the RS232 driver going to the clipping diodes, an in-series resistor must be used.

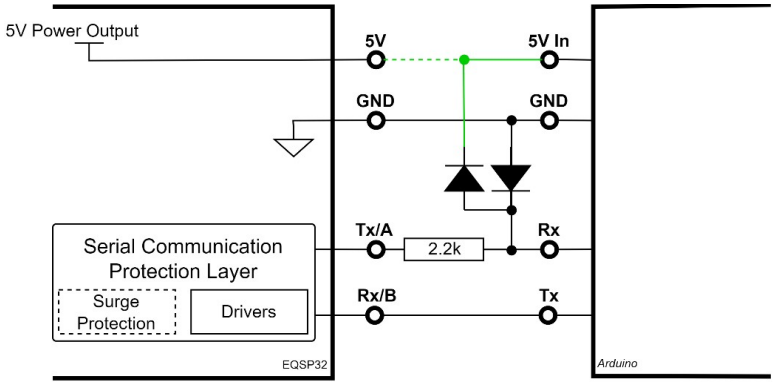

Figure 12-2. RS232 to TTL wiring

> **NOTE**
> RS232 signals are inverted by the RS232 driver.
> In order for communication to work, apart from clipping the signals to TTL levels, it is also needed to setup the communication on either the EQSP32 or TTL device as INVERTED. For EQSP32 use `RS232_INV` as serial mode to invert the signal polarity.

Table 12-1. Library reference: Serial – RS232

Configuration	
`eqsp32.configSerial(mm, bb)`	mm: serial mode RS232 or `RS232_INV` for inverted polarity bb: baud rate (optional, default = 115200) ex. `eqsp32.configSerial(RS232, 9600)`

Usage	
`eqsp32.Serial.{Serial function}`	Call Serial from eqsp32 class and use any Serial.{function} like you would normally do

12.2 Serial – RS485

Half-Duplex RS485

For standard RS485 half-duplex communication, the wiring is as shown in the following diagram. A termination resistor may be needed depending on the wire length.

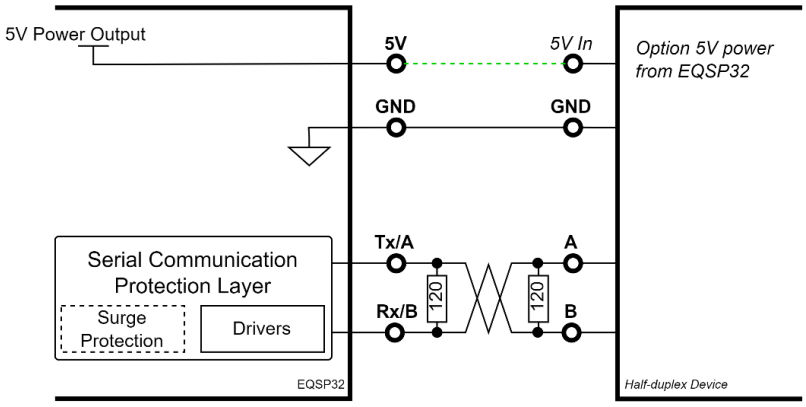

Figure 12-3. Half-duplex RS485 wiring

Generally, the right way to terminate an RS485 bus depends on the network layout but the standard practice is to use **differential termination** and **fail-safe biasing.**

Differential Termination

To ensure signal integrity on an RS-485 network, the bus must be terminated at both physical ends with a 120 Ω resistor across the A and B differential lines.
This matches the characteristic impedance of the twisted-pair cable (typically 120 Ω) and prevents signal reflections that can cause data corruption at higher baud rates.

Only the two devices located at the ends of the bus should include termination resistors; intermediate nodes should remain unterminated.

Use **differential termination** when:

- Operating at baud rates above 9600 bps (especially 19200 bps and higher).
- Bus cable lengths are greater than 10 m.
- Three or more nodes are connected over the twisted-pair wiring.

Fail-Safe Biasing

Fail-safe biasing ensures that the RS-485 bus remains in a known logic state when no transmitter is active (idle condition). This is achieved by adding bias resistors that slightly drive the differential lines to a defined level.

Typical configuration:

- **Pull-up resistor (560 Ω, 680 Ω at most) from A → 5V Out terminal**
- **Pull-down resistor** (same value) from **B → GND**

These resistors maintain a small voltage difference between A and B, preventing random noise from being interpreted as data.

Fit the bias resistors at one node only, usually the master. With 120 Ω terminators at both ends of the bus, 560 Ω gives about 250 mV between A and B at idle, above the 200 mV that RS-485 receivers need; 1 kΩ gives too little. Do not connect the pull-up to VIn: on a 24 V system it drives the lines outside the RS-485 voltage range.

© 2026 Erqos Technologies P.C. · erqos.com

Use **Fail-Safe Biasing** when:

- Operating at baud rates above 9600 bps (especially 19200 bps and higher).
- The bus is long (over 50 m) or operates in electrically noisy environments.
- The network includes many passive (listening-only) nodes.

RS485 is half-duplex and the application sets the direction: call `eqsp32.configSerial(RS485_TX, bb)` before sending, and `eqsp32.configSerial(RS485_RX, bb)` after the last byte has been sent (for example after `eqsp32.Serial.flush()`). The frame format is 8N1.

On the EQSP32RE the second RS485 port is started as Serial1 at 9600 baud, 8N1, in receive direction. Its pins are returned by `eqsp32.getPin()` with `EQ_RS485_2_TX`, `EQ_RS485_2_RX` and `EQ_RS485_2_EN`, for use with third-party libraries.

Table 12-2. Library reference: Serial – RS485

Configuration	
`eqsp32.configSerial(mm, bb)`	mm: serial mode `RS485_TX` or `RS485_RX` for Tx or Rx bb: baud rate (optional, default = 115200)
Usage	
`eqsp32.Serial.{Serial function}`	Call Serial from eqsp32 class and use any Serial.{function} like you would normally do.

12.3 CAN-Bus

For reliable CAN-Bus communication, it is essential to use two 120 Ω termination resistors on the network — one at each physical end of the CAN bus.
These resistors are connected across the CAN_H and CAN_L lines and are required to match the characteristic impedance of the twisted-pair cable (typically 120 Ω).

Proper termination prevents signal reflections, ensures correct voltage levels on the differential lines, and allows stable communication between all nodes on the bus.

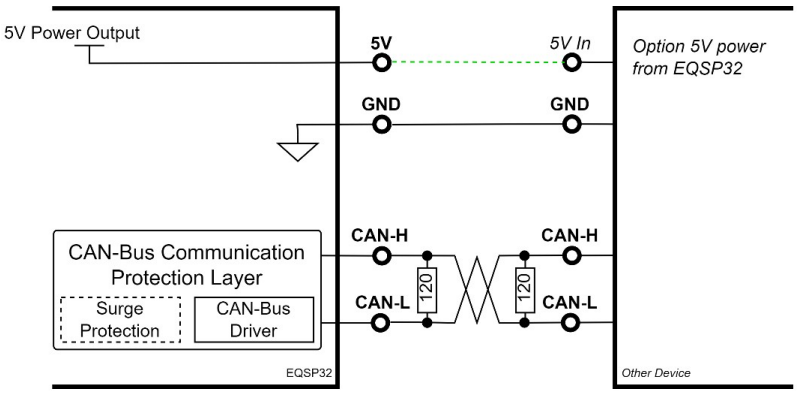

Figure 12-4. CAN-Bus wiring

> **NOTE**
> When setting up a CAN network, verify if any of the devices already have internal termination resistors before adding any external termination.

On models with CAN, the CAN controller starts in `eqsp32.begin()` at 250 kbit/s and accepts all frames. `eqsp32.configCAN()` sets the bit rate, from 25 kbit/s to 1 Mbit/s (CAN_25K, CAN_50K, CAN_100K, CAN_125K, CAN_250K, CAN_500K, CAN_800K, CAN_1000K), and an optional acceptance filter. The transmit and receive queues hold 16 frames each. Frames use standard 11-bit identifiers.

© 2026 Erqos Technologies P.C. · erqos.com

Table 12-3. Library reference: CAN-Bus

Configuration	
`eqsp32.configCAN(bb, id, lb)`	Configures the CAN bus interface on the EQSP32 for accepting all messages or filtering for a single message ID. bb: baud rate of type CanBitRates , ex. CAN_250K, CAN_500K id: acceptance id (optional, default = 0, accepts all) lb: loopback (optional, default = false, no loopback)
`eqsp32.configCANNode(bb, nid)`	Configures the CAN bus to filter messages by Node ID only (CANopen-style filtering). Accepts all Function Codes for the specified Node ID but no broadcast messages like NMT (0x000) or SYNC (0x080), these are filtered out. bb: baud rate of type CanBitRates , ex. CAN_250K, CAN_500K nid: node id (ex. 0x21 to accept all PDOs, SDOs and EMCY for node ID 0x21). Range: 1 to 127.
Usage	
`eqsp32.transmitCANFrame(mm)`	Transmits a CAN message using the EQSP32 CAN interface. This function sends a standard 11-bit CAN message using the built-in TWAI controller. The message structure must be pre-filled with an identifier, data payload, and data length. mm: a CanMessage struct containing the CAN ID, data bytes, and data length
`eqsp32.receiveCANFrame(mm)`	Receives a CAN message using the EQSP32 CAN interface. This function attempts to retrieve a standard 11-bit CAN frame from the TWAI receive queue. It operates in a non-blocking manner; if no message is available, the function returns immediately with false. Returns true if a message was successfully received. mm: a reference to a CanMessage struct where the received data will be stored.

12.4 Third-Party Libraries – Native Pin Control

Direct pin control

There might be cases that a custom or an external library is needed for an application. For example, to control an addressable LED panel, run DMX master/slave protocol, Modbus, CANOpen, or other protocols, there are plenty available libraries in the esp32/Arduino community.

To take direct control of any of the ADIO pins or communication peripherals (CAN-Bus, RS232 and RS485), the system needs to be notified to release these pins from the internal control and setup the corresponding hardware for the specific process.

For this purpose, the `eqsp32.getPin()` function may be used, which will disassociate the corresponding pin from the EQ System Supervisor, prepare the corresponding peripherals and return the native pin mapping for the requested ADIO pin number.

A terminal handed over with `eqsp32.getPin()` is set to CUSTOM mode: the library no longer drives it, and `eqsp32.readPin()` returns -1 for it. A pulse counter (PCC) or encoder pair (QENC) on the terminal is released; the other terminal of the pair is left in DIN mode. This is the way to control a terminal's pin directly from the application. Requesting an RS232 or RS485 pin stops `eqsp32.Serial`, and requesting a CAN pin stops the library's CAN driver. For a pin that does not exist on the model, `eqsp32.getPin()` returns -1.

With direct access, the input thresholds are: terminals 1-8, HIGH from 9 V and LOW up to 2.5 V; terminals 9-16, HIGH from 5 V and LOW up to 0.6 V.

Table 12-4. Library reference: Third-Party Libraries – Native Pin Control

Usage	
`eqsp32.getPin(pp)`	Returns the native pin number (the actual MCU pin number for this ADIO or peripheral pin) pp: ADIO pin number, can be from `EQ_PIN_1`, …, `EQ_PIN_16` or `EQ_RS232_TX`, `EQ_RS232_RX`, …

To use `getPin()` for any of the ADIO pins, the available input values are **`EQ_PIN_xx`**, where xx is from 1 to 16, for all 16 ADIOs.

To use RS232, RS485 and CAN peripherals with external/custom libraries the inputs values are:

`EQ_RS232_TX` (prepares EQSP32 RS232 peripheral and returns the native TX pin number)

`EQ_RS232_RX` (prepares EQSP32 RS232 peripheral and returns the native RX pin number)

`EQ_RS485_TX` (prepares EQSP32 RS485 peripheral in receive mode and returns the native TX pin number)

`EQ_RS485_RX` (prepares EQSP32 RS485 peripheral in receive mode and returns the native RX pin number)

`EQ_RS485_EN` (prepares EQSP32 RS485 peripheral in receive mode and returns the native RS485 Enable pin number, when `RS485_EN` is HIGH, RS485 will enter transmit mode, when `RS485_EN` is LOW, it will enter receive mode)

`EQ_CAN_TX` (prepares EQSP32 CAN peripheral and returns the native TX pin number)

`EQ_CAN_RX` (prepares EQSP32 CAN peripheral and returns the native RX pin number)

`EQ_RS485_2_TX`, **`EQ_RS485_2_RX`**, **`EQ_RS485_2_EN`** (EQSP32RE only: return the native pin numbers of the second RS485 port)

Example:

```
int rs232Tx = eqsp32.getPin(EQ_RS232_TX); // Gets the ESP32 pin number for RS232 transmission.
int rs232Rx = eqsp32.getPin(EQ_RS232_RX); // Gets the ESP32 pin number for RS232 reception.
// Put your custom setup and usage of RS232 by using rs232Tx and rs232Rx pins directly
```

© 2026 Erqos Technologies P.C. · erqos.com

13 Software Architecture

EQSP32 utilizes the EQSP32 proprietary library to automatically handle all wireless connectivity tasks. The library provides a user API for easy and straightforward configuration and usage of all ADIOs and communications.

The following block diagram demonstrates the software architecture of the library. The first section corresponds to the application programming interface (API) and the second to all the invisible tasks running on the background, being handled by the EQ System Supervisor.

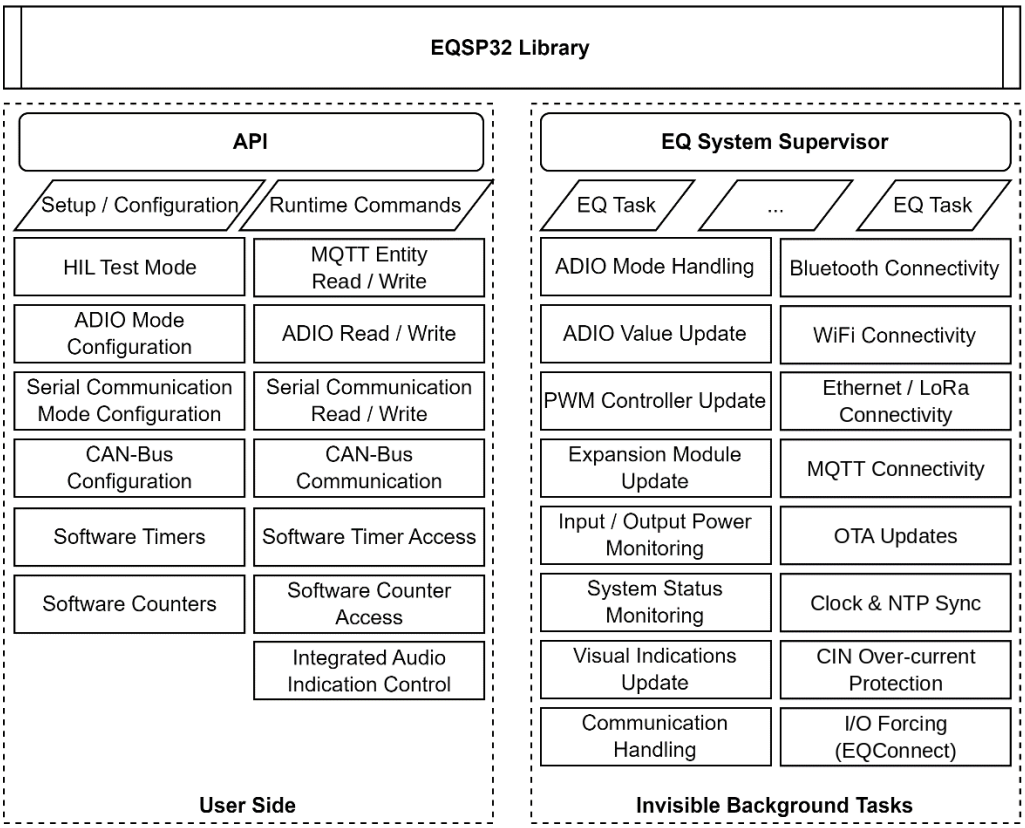

Figure 13-1. EQSP32 Library Block Diagram

14 Programming, Configuration and Default Parameters

The EQSP32 comes with a precompiled proprietary library that provides the user with an API to access all ADIO modes, communications and internal monitoring functionality.

When the program is deployed and the library is running, EQ System Supervisor takes care of all background tasks related pin and peripheral update and Bluetooth and WiFi connectivity.

To program EQSP32 connect it with a USB-C and open Visual Studio Code, Arduino IDE or any other ESP32 compatible IDE. After installing EQSP32 library, the user should describe their wiring and preferred functionality to EQ-AI, start developing the code themselves or a combination of both. Then, simply download and power EQSP32 from the external power supply.

It is important that the `begin()` function is called in the `setup()` section of the user program before any other EQSP32 operation is used.

© 2026 Erqos Technologies P.C. · erqos.com

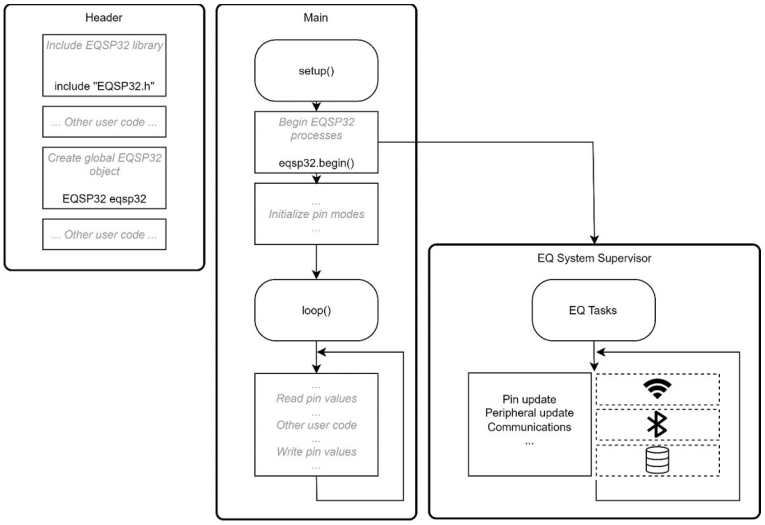

Figure 14-1. Program flow with EQSP32 library

Table 14-1. Library reference: Programming, Configuration and Default Parameters

Configuration	
`eqsp32.begin(vv)`	Starts EQSP32 tasks using default library configuration preferences, allows for verbose option control vv: enable verbose (optional, default = false)
`eqsp32.begin(cc, vv)`	Starts EQSP32 tasks using developer defined default parameters and/or EQSP32 system configuration preferences, allows for verbose option control cc: configuration struct `EQSP32Configs` to define system ID and other EQSP32 system preferences (see example below) vv: enable verbose (optional, default = false)
`eqsp32.begin()`	Starts EQSP32 tasks using default library configurations and preferences with verbose disabled
Usage	
`eqsp32.initStatus()`	Returns the start-up state: INIT_NOT_STARTED, INIT_STARTED, or INIT_OK once start-up is complete

Differences between begin() variations

- **`begin()`:** This is the easiest way to get started. It initializes EQ supervisor to run EQSP32 system processes without defining any custom default parameters and options. Having the system messages disabled (verbose = false), allows for easier monitoring of developer's system messages.
- **begin(cc) or `eqsp32.begin(cc, false)`:** Using `begin()` in either of these ways is suitable for production builds. This disables the verbose option, so no EQSP32 system messages will be printed on the terminal by EQ supervisor tasks.
- **begin(cc, true):** It enables the verbose option. This is a helpful tool to monitor the EQSP32 system's internal processes and verify the correct operation for connectivity, peripheral status, network parameters and other system tasks.

Initial system configurations: EQSP32Configs

```
#include "EQSP32.h"
EQSP32 eqsp32;
void setup() {
  EQSP32Configs myEQSP32Configs;
  myEQSP32Configs.devSystemID = "SystemID";  // Developer's system code, versioning etc.
  eqsp32.begin(myEQSP32Configs, true);   // Verbose enabled
  // EQSP32 pin mode initializations and other initialization code
}
```

14.1 EQSP32Configs Structure

The `EQSP32Configs` structure defines all **initialization and runtime configuration parameters** for the EQSP32 controller. It provides a unified interface for setting up the device's **networking**, **MQTT communication**, **device identity**, and **system behavior** before initialization.

These parameters are applied when calling the `eqsp32.begin()` function. The Wi-Fi credentials and the device name can also be changed at runtime, and the MQTT login and enable can be stored for the next restart (see Wireless and Wired Internet Connectivity, and MQTT and Home Assistant). Settings stored on the device through EQConnect take priority over these values until the network parameters are reset with the boot button.

When no configuration is provided, all fields default to safe baseline values — enabling the EQSP32 to start in **standalone mode** and automatically handle provisioning through the **EQConnect mobile app**.
By defining custom settings within this structure, developers can override defaults to integrate the EQSP32 seamlessly into existing networks, MQTT brokers, or third-party IoT systems.

Table 14-2. Library reference: EQSP32Configs Structure

Field	Type	Default Value	Description
`mqttBrokerIp`	std::string	"homeassistant.local"	IP address or hostname of the MQTT broker (e.g., Home Assistant).
`mqttBrokerPort`	int	1883	MQTT broker port. Use 8883 for SSL/TLS.
`mqtt_broker_ca`	std::string	""	(Optional) MQTT CA certificate, used to verify the broker when TLS is used (port 8883). Leave empty to skip verification.
`mqttDiscovery`	bool	false	Enables MQTT operations on EQSP32 (Pub/sub on interfacing topics and device discovery for Home Assistant integration).
`devSystemID`	std::string	""	Developer-assigned system ID and/or versioning (read-only for external apps).
`userDevName`	std::string	""	Default device name. Editable by user from EQConnect.
`wifiSSID`	std::string	""	(Optional) Default Wi-Fi SSID, leave empty for no default SSID. Stored SSID is reset to this value if network credentials are erased by pressing and holding the 'boot' button.
`wifiPassword`	std::string	""	(Optional) Default Wi-Fi password, leave empty for no default password. Stored password is reset to this value if network credentials are erased by pressing and holding the 'boot' button.
`staticIP`	std::string	"0.0.0.0"	Default static IP address configuration. Use 0.0.0.0 to enable DHCP. Stored IP configuration is reset to this value if network credentials are erased by pressing and holding the 'boot' button.
`gateway`	std::string	"0.0.0.0"	Default gateway IP address (used when static IP is enabled). Stored IP configuration is reset to this value if network credentials are erased by pressing and holding the 'boot' button.
`subnet`	std::string	"0.0.0.0"	Default subnet mask (used when static IP is enabled). Stored IP configuration is reset to this value if network credentials are erased by pressing and holding the 'boot' button.
`DNS`	std::string	"0.0.0.0"	Default DNS server (used when static IP is enabled). Stored IP configuration is reset to this value if network credentials are erased by pressing and holding the 'boot' button.

© 2026 Erqos Technologies P.C. · erqos.com

Field	Type	Default Value	Description
`relaySequencer`	bool	false	Enables the **internal relay sequencing feature**. When multiple relays are commanded **ON simultaneously**, the EQSP32 automatically activates them **sequentially in ascending pin order** rather than all at once. This behavior prevents high inrush currents and minimizes **power surges on the power supply**. Each relay waits until the previous one completes its **derate-to-hold power phase** before the next relay is triggered. **Example:** If relays on pins **9**, **10**, and **12** are all commanded to turn ON at the same time: Relay on **pin 9** activates first and transitions to holding power, Then **pin 10** activates, Finally, **pin 12** turns on last. This ensures smoother load transitions and increased power system stability during multi-relay activation events.
`disableErqosIoT`	bool	false	Disables EQSP32's built-in IoT functions: Wi-Fi and BLE provisioning through EQConnect, MQTT, the boot button network actions and the automatic Bluetooth/WiFi LED indications. **Use only if Wi-Fi and BLE will be handled by user code or third-party solutions.**
`disableNetSwitching`	bool	false	Prevents automatic switching between Ethernet and Wi-Fi when both are connected. When this feature is enabled, static IP configurations only apply on Ethernet connectivity and WiFi always operates on DHCP.
`disableSystemMQTTEntities`	bool	false	Disables the system entities the device publishes over MQTT: heartbeat, uptime, input and output voltage and Wi-Fi signal strength.
`mqttAutoUpdateStateFromCommand`	bool	true	When a command is received on an entity's command topic, the same value is published on its state topic. Set to false to update the state only from the application (for example with updateControl_Switch()).
`bleBroadcastingMins`	int	3	Minutes of Bluetooth advertising after power-up or after a short press of the Boot button, while no client is connected. 0 keeps advertising on.

© 2026 Erqos Technologies P.C. · erqos.com

15 Wireless and Wired Internet Connectivity

The EQSP32 controller supports both Wi-Fi and Ethernet connectivity *(on respective versions)*, providing flexible communication options for a wide range of industrial and IoT environments.
Depending on the model, the EQSP32 may feature Wi-Fi only or Wi-Fi and Ethernet hardware capability.
By default, both interfaces are managed by the internal EQSP32 system supervisor network task, which automatically handles provisioning, active connection maintenance and interface switching without the need for user code.

15.1 Automatic Network Management

During startup, the EQSP32 automatically initializes all available network interfaces.
The internal network manager continuously monitors connection status and gives priority to Ethernet.

Automatic Interface Switching and Priority Behavior

By default, `disableNetSwitching` is set to **false**, meaning that **only one network interface is active at a time**.
When both Ethernet and Wi-Fi are available, **Ethernet automatically takes priority**, and Wi-Fi is automatically disconnected and turned off.

If the Ethernet link is disconnected or unavailable, the EQSP32 seamlessly switches to Wi-Fi to maintain connectivity.
This automatic **failover** ensures reliable operation without requiring user intervention.

If a fixed network configuration is required — where the EQSP32 should not switch between interfaces — the automatic selection can be disabled by setting `disableNetSwitching = true` in the `EQSP32Configs` structure.

When enabled, the device may be connected to both interfaces simultaneously.
This is particularly useful especially when Ethernet connection is part of the system integration, for example the EQSP32 is connected via the Ethernet as part of the Modbus TCP integration and uses the WiFi for internet connection.

15.2 Wi-Fi Connectivity

All EQSP32 versions include integrated 2.4 GHz Wi-Fi (802.11 b/g/n) capability.
Wi-Fi can be configured through code or provisioned via the EQConnect mobile app over BLE.

Key Features:

- **Provisioning:**
  Custom default Wi-Fi credentials can be set in the `EQSP32Configs` structure (`wifiSSID`, `wifiPassword`).
  End user may adjust the network credentials using EQConnect provisioning.
- **Automatic Reconnection:**
  EQSP32 automatically retries connection when Wi-Fi becomes temporarily unavailable.
- **Status Monitoring:**
  Use `eqsp32.getWiFiStatus()` to retrieve the current connection state:
  `EQ_WF_DISCONNECTED`, `EQ_WF_CONNECTED`, `EQ_WF_RECONNECTING`, or `EQ_WF_SCANNING`.
- **Runtime control:** `eqsp32.startWiFi()` connects, optionally with new credentials that are then stored; `eqsp32.stopWiFi()` disconnects; `eqsp32.clearWiFi()` erases the stored credentials.

15.3 Ethernet Connectivity (Model-Dependent)

On EQSP32 models equipped with Ethernet hardware, the controller provides a wired network interface over a standard RJ45 connector.
The Ethernet interface is detected and started in `eqsp32.begin()`, and the connection comes up when a network cable is connected.

Key Features:

- **Plug-and-Play Operation:**
  DHCP is enabled by default. To assign a fixed IP address, configure `staticIP`, `gateway`, `subnet` and `DNS` in the `EQSP32Configs` structure. The network host name is the device name, with spaces and underscores replaced by hyphens.

© 2026 Erqos Technologies P.C. · erqos.com

- **Priority Handling:**
  When both interfaces are active, Ethernet always takes priority unless `disableNetSwitching` is manually set to true.
- **Status Monitoring:**
  Retrieve the link state using `eqsp32.getEthernetStatus()`:
  `EQ_ETH_DISCONNECTED`, `EQ_ETH_PLUGGED_IN`, `EQ_ETH_CONNECTED`, or `EQ_ETH_STOPPED`.

Table 15-1. Library reference: Ethernet Connectivity (Model-Dependent)

Usage	
`eqsp32.isDeviceOnline()`	Checks if the EQSP32 is currently online via either Wi-Fi or Ethernet. Type: Bool Returns true if: • Wi-Fi status = `EQ_WF_CONNECTED`, or • Ethernet status = `EQ_ETH_CONNECTED`. Returns false if both interfaces are disconnected.
`eqsp32.getWiFiStatus()`	Retrieves the current Wi-Fi connection status of the EQSP32 module. Type: EQ_WifiStatus Possible return values: • `EQ_WF_DISCONNECTED` (0): Not connected to any Wi-Fi network. • `EQ_WF_CONNECTED` (1): Successfully connected. • `EQ_WF_RECONNECTING` (2): Attempting to reconnect. • `EQ_WF_SCANNING` (3): Scanning for networks.
`eqsp32.getEthernetStatus()`	Retrieves the current Ethernet connection status of the EQSP32 module. Type: EQ_EthernetStatus Possible return values: • `EQ_ETH_DISCONNECTED` (0): No Ethernet cable detected or no link. • `EQ_ETH_CONNECTED` (1): Ethernet connected and online. • `EQ_ETH_PLUGGED_IN` (2): Cable detected but IP not yet acquired. • `EQ_ETH_STOPPED` (3): Ethernet interface disabled or not present (Wi-Fi-only model).
`eqsp32.localIP()`	Returns the current IP address of the EQSP32 device. Type: String Returns: • Ethernet IP if Ethernet is connected. • Wi-Fi IP if Wi-Fi is connected and Ethernet is not. • '0.0.0.0' if the device is offline.
`eqsp32.ethernetIP()`	Returns the current Ethernet IP address of the EQSP32. Type: String Returns: • Valid IP if Ethernet is connected. • '0.0.0.0' if Ethernet is not connected.
`eqsp32.wifiIP()`	Returns the current Wi-Fi IP address of the EQSP32. Type: String Returns: • Valid IP if Wi-Fi is connected. • '0.0.0.0' if Wi-Fi is not connected.
`eqsp32.startWiFi(ss, pw)`	Starts Wi-Fi with the stored credentials, or with new ones, which are stored ss, pw: network name and password (optional)
`eqsp32.stopWiFi()`	Disconnects from the Wi-Fi network
`eqsp32.clearWiFi()`	Erases the stored Wi-Fi credentials and disconnects; the `EQSP32Configs` defaults apply again

15.4 LoRa Connectivity (EQSP32CL-X)

The EQSP32CL-X carries an SX1262 LoRa radio with a power amplifier and an external antenna. The radio is detected in `eqsp32.begin()`, and `eqsp32.isLoRaAvailable()` reports whether it is ready. LoRa and Ethernet are not available on the same unit.

The radio is used through `eqsp32LoRa`, a RadioLib SX1262 object, with the RadioLib functions for sending and receiving. The default arguments of `eqsp32LoRa.begin()` are 868.0 MHz, 125 kHz bandwidth, spreading factor 7, coding rate 4/5, the private sync word, a power setting of 10 dBm and a preamble of 8 symbols.

> **NOTE**
> The frequency band and the output power are set by the application. The system developer must configure the band, output power and duty cycle allowed in the region where the product is used.

Table 15-2. Library reference: LoRa Connectivity (EQSP32CL-X)

Usage	
`eqsp32.isLoRaAvailable()`	Returns true if the LoRa radio was found and started in `eqsp32.begin()`
`eqsp32LoRa.begin(fr, bw, sf, cr, sw, pw, pl)`	Starts LoRa mode (all arguments optional) fr: frequency in MHz (868.0); bw: bandwidth in kHz (125.0); sf: spreading factor (7); cr: coding rate denominator (5) sw: sync word (private); pw: power setting in dBm (10); pl: preamble length (8)
`eqsp32LoRa.{RadioLib function}`	Any RadioLib SX1262 function, for example transmit() and receive()

16 MQTT and Home Assistant

EQSP32 has a built-in MQTT client that can be used in two ways. As a plain MQTT device, it works with any MQTT broker, for integrations with Node-RED, custom applications or other platforms: the application creates entities (switches, numeric values, texts, binary sensors and sensors) with a few function calls, and the library publishes their states and receives commands on the device's own topics (Enabling MQTT, Device Identity and Topics). With Home Assistant, the same entities are also announced through MQTT discovery, and Home Assistant finds the device automatically (Home Assistant Integration).

16.1 Enabling MQTT

- **Enable:** set `mqttDiscovery` = true in `EQSP32Configs`, switch MQTT on from the EQConnect app, or call `eqsp32.updateMQTT(true)`. The setting is stored on the device; a change made from EQConnect or with `eqsp32.updateMQTT()` applies after a restart.
- **Broker:** host name or IP address and port from `EQSP32Configs` (`mqttBrokerIp`, `mqttBrokerPort`; default `homeassistant.local`, port 1883).
- **Encryption:** port 8883 selects an encrypted (TLS) connection; the broker's CA certificate is given in `mqtt_broker_ca`.
- **Login:** the MQTT user name and password are set from the EQConnect app or with `eqsp32.updateMQTT()`. They are used when both are set.

16.2 Device Identity and Topics

The device ID is EQ_ followed by the device name, with spaces replaced by underscores and special characters removed: the device name "My EQSP32!" gives `EQ_My_EQSP32`. `eqsp32.getDeviceID()` returns it. All topics are built from it:

Table 16-1. Device Identity and Topics

Topic	Use
state/<device ID>/<entity>	Entity state, published by the device
cmd/<device ID>/<entity>	Commands to the device
status/<device ID>	Availability: online or offline (retained)
homeassistant/...	Home Assistant discovery messages

Commands ON, OPEN and PRESS are read as true; OFF and CLOSE as false. With `mqttAutoUpdateStateFromCommand` (default true) a received command is also published on the entity's state topic.

Unless `disableSystemMQTTEntities` is set, the device also publishes system entities: a heartbeat and its uptime every second, and its input voltage, output voltage and Wi-Fi signal strength every 10 seconds.

If the broker connection is lost, the library reconnects automatically. `eqsp32.getMQTTStatus()` returns the connection state.

Table 16-2. Library reference: Device Identity and Topics

Configuration	
`createControl_Switch(nm, ic)`	Creates a switch entity (on/off command and state) nm: entity name; ic: Home Assistant icon (optional)
`createControl_Value(nm, mn, mx, dc, ic)`	Creates a numeric value entity mn, mx: range (optional, default = 0 to 1000); dc: decimals (optional, default = 0)
`createDisplay_BinarySensor(nm, ic, cl)`	Creates a binary sensor (on/off display) cl: Home Assistant binary sensor class (optional)
`createDisplay_Sensor(nm, dc, un, ic, cl)`	Creates a sensor (numeric display) un: unit (optional); cl: Home Assistant sensor class (optional)
`createControl_Switches(it, n)`, `createControl_Values(it, n)`, `createControl_Texts(it, n)`, `createDisplay_BinarySensors(it, n)`, `createDisplay_Sensors(it, n)`	Create n entities from an array it of entity definitions Recommended for more than 20 entities. Text entities are created this way only.
`createGlobal_Topic(tp)`	Subscribes to a plain MQTT topic tp, outside the device's own topics
Usage	
`readControl_Switch(nm)`, `readControl_Value(nm)`, `readControl_Text(nm)`	Return the current value of a control entity
`updateControl_Switch(nm, vv, rt)`, `updateControl_Value(nm, vv, rt)`, `updateControl_Text(nm, vv, rt)`	Publish a new state vv: value; rt: retain (optional, default = false)
`readDisplay_BinarySensor(nm)`, `readDisplay_Sensor(nm)`	Return the last value published for a display entity
`updateDisplay_BinarySensor(nm, vv, rt)`, `updateDisplay_Sensor(nm, vv, rt)`	Publish a new sensor value
`readGlobal_Topic(tp)`, `readGlobal_TopicBOOL(tp)`, `readGlobal_TopicFLOAT(tp)`	Return the last payload received on tp, as text, true/false or a number
`updateGlobal_Topic(tp, pl, rt)`	Publishes payload pl on topic tp; returns false if nothing was sent
`eqsp32.getMQTTStatus()`	Returns `EQ_MQTT_CONNECTED`, `EQ_MQTT_CONNECTING`, `EQ_MQTT_DISCONNECTED` or `EQ_MQTT_NO_INIT`
`eqsp32.updateMQTT(us, pw, en)`, `eqsp32.updateMQTT(en)`	Store the MQTT user name and password and the MQTT enable; applied after a restart
`eqsp32.getDeviceID()`	Returns the device ID used in the topics

NOTE
The entity functions (create..., read..., update...) are global functions and are called without the eqsp32. prefix.

16.3 Home Assistant Integration

When the EQSP32 uses the broker of a Home Assistant installation, it appears in Home Assistant as a device with every entity the application has created. The library announces the device information and its entities for Home Assistant device discovery.

Icons and Home Assistant classes are optional arguments of the create functions (ic; cl for binary sensors and sensors). The default broker address, `homeassistant.local` on port 1883, is the Home Assistant host.

© 2026 Erqos Technologies P.C. · erqos.com

17 Over-the-Air (OTA) Updates

A new program can be loaded over the network, without a USB connection, in two ways:

- **Arduino OTA:** the EQSP32 appears as a network port, under its device name, in the Arduino IDE and in other tools that support Arduino OTA uploads.
- **Web page:** the maintenance page at http://<device IP address>:8000 accepts a compiled program file (.bin).

Both are protected by the OTA password. Until you set your own, the password is "admin"; it can be changed at any time from the EQConnect app (IoT Configuration). The EQSP32 restarts about 1 second after the upload has completed and runs the new program.

18 Time, Uptime and Timers

18.1 Clock and Calendar

Once the EQSP32 is online, its clock is set over the internet from public time servers (NTP: pool.ntp.org, time.google.com, time.nist.gov). The local time zone, including daylight saving, is set from the EQConnect app and stored on the device. `eqsp32.isLocalTimeSynced()` returns true once the clock has been set.

18.2 Uptime

The uptime is counted from the last start and does not depend on the clock.

18.3 Software Timers (EQTimer)

EQTimer provides non-blocking timers for periodic actions, delays and time-outs, with a preset in milliseconds.

Table 18-1. Library reference: Software Timers (EQTimer)

Clock	
`eqsp32.isLocalTimeSynced()`	Returns true once the clock has been set over the internet
`eqsp32.getLocalYear()`, getLocalMonth(), getLocalMonthDay(), getLocalYearDay(), getLocalWeekDay()	Local date: year, month (1-12), day of month (1-31), day of year (1-366), weekday (`EQ_SUNDAY` to `EQ_SATURDAY`)
`eqsp32.getLocalHour()`, getLocalMins(), getLocalSecs()	Local time: hour (0-23), minutes and seconds (0-59)
`eqsp32.getLocalUnixTimestamp()`, `eqsp32.getUnixTimestamp()`	Local or UTC time as a Unix timestamp (seconds since 1 January 1970)
`eqsp32.getFormattedLocalTime()`, `eqsp32.getFormattedUnixTimestamp()`	Local or UTC time as text, "MM-DD-YYYY HH:MM:SS"
`eqsp32.printLocalTime()`	Prints the local time on the serial monitor
Uptime	
`eqsp32.getUptimeDays()`, getUptimeHours(), getUptimeMinutes(), getUptimeSeconds()	Uptime in full days, then hours (0-23), minutes and seconds (0-59)
`eqsp32.printUptime()`	Prints the uptime on the serial monitor
EQTimer	
`EQTimer tt(pp)`	Creates a stopped timer pp: preset in ms (optional, default = 0)
`tt.start(pp)`	Starts or resumes the timer pp: new preset in ms (optional)
`tt.pause()`, `tt.stop()`	Pauses the timer and keeps the elapsed time; stops it and clears the elapsed time
`tt.value()`	Returns the elapsed time in ms

© 2026 Erqos Technologies P.C. · erqos.com

`tt.isElapsed(ar)`	Returns true when the running timer has reached its preset ar: restart the timer automatically (optional, default = false)
`tt.isRunning()`	Returns true while the timer is running

19 Device Provisioning

Once EQSP32 library is running, use “EQConnect” phone app to connect controller via Bluetooth. From the EQConnect mobile app you may configure the WiFi network’s credentials, rename the device, setup local timezone with daylight savings, wirelessly monitor EQSP32 I/O modes, states and all connected expansion modules. For commissioning and testing, the app can also force inputs and outputs; every force ends when the app disconnects. Pins named in the application with `eqsp32.pinName()` appear with their names.

> **WARNING**
> Forcing overrides the application program and drives the real outputs, and the program is not told that a pin is forced. Force only when the connected equipment is in a safe state. Anyone who connects with EQConnect while the EQSP32 is advertising can force I/O, so keep `bleBroadcastingMins` short on installed units.

The "IoT Configuration" section offers additional setup options for defining a static IP or DHCP, updating the OTA (Over-The-Air) updates' password, switching MQTT on or off and setting up the MQTT user name and password.

> **NOTE**
> For EQSP32 to be discoverable via Bluetooth, the Bluetooth LED must be flashing. If the Bluetooth LED is off, a short press on the boot button will enable BLE advertising for 3 minutes (`bleBroadcastingMins` in `EQSP32Configs`).

The WiFi network credentials will be stored in the internal flash. Each time EQSP32 starts or restarts, it will automatically try to reconnect on the saved network.

Even if a new program is uploaded in the device, if no explicit full flash erase is selected during programming, the wifi credentials will be retained.

> **NOTE**
> To force EQSP32 to disconnect from the WiFi network and erase the WiFi credentials from flash, press and hold the boot button for 3 seconds.
> It is important to remember that the "erase network parameters" operation practically resets the user networking parameters to defaults.
> If the developer has initialized the system with custom default network parameters, like credentials, static IP, etc., these will be applied after pressing the boot button for 3 seconds instead.

19.1 Bluetooth

Bluetooth advertising starts at every power-up under the name "[EQSP32] <device name>" and stops after `bleBroadcastingMins` minutes (default 3) if no client has connected; 0 keeps it on. A short press on the Boot button or the device web page starts it again.

While the library manages Bluetooth, the application can also exchange text over Bluetooth through `bleSerial` (or `bleStream`), a serial-style stream on the Nordic UART Service.

Table 19-1. Library reference: Bluetooth

Usage	
`eqsp32.getBleStatus()`	Returns `EQ_BLE_ADVERTISING`, `EQ_BLE_CONNECTED`, `EQ_BLE_SUBSCRIBED`, `EQ_BLE_DISCONNECTED` or `EQ_BLE_NO_INIT`
`eqsp32.stopBle()`	Stops Bluetooth advertising; returns false while a client is connected
`bleSerial.{Serial function}`	Serial-style text stream over Bluetooth; use it like Serial
`eqsp32.getDeviceName()`	Returns the device name

© 2026 Erqos Technologies P.C. · erqos.com

`eqsp32.setDeviceName(nm)`	Sets and stores a new device name nm: device name
`eqsp32.pinName(nn, nm)`	Names a pin so that EQConnect shows the name; kept until restart; can be called before or after `eqsp32.begin()`. Returns false and keeps the previous name if the name is refused nn: pin index, or an expansion-module pin such as EQXIO(1, 3) nm: name, 1 to 64 bytes of UTF-8 text

20 Stored Settings

The EQSP32 stores the following settings in its internal flash. They are kept across restarts and program uploads, unless the flash is fully erased during programming. With `disableErqosIoT` set, the library does not use them.

Table 20-1. Stored Settings

Group	Settings	Changed by
Wi-Fi	Network name (SSID) and password	EQConnect, `eqsp32.startWiFi()`, `eqsp32.clearWiFi()`, Boot button (3 s hold)
Device	Device name; country, city and time zone	EQConnect (all); `eqsp32.setDeviceName()` (device name only)
IoT	MQTT enable, MQTT user name and password OTA password Static IP, gateway, subnet mask and DNS	EQConnect, `eqsp32.updateMQTT()`; Boot button 3 s hold (MQTT enable and IP settings)

Values stored on the device take priority over the `EQSP32Configs` defaults. Pin modes, output values, the PWM frequency and all `eqsp32.config...()` settings are not stored: the application sets them at every start.

21 Hardware-in-the-Loop (HIL) Test Mode

In HIL test mode a test program on a PC sets the EQSP32's input values, and reads values back, over the USB serial port while the application runs unchanged on the controller. An application can thus be tested, also by automated test tools, without wiring sensors or a process.

Call `eqsp32.beginHil()` before `eqsp32.begin()`. The mode stays active until the next restart. Commands are text lines on USB serial at 115200 baud:

- -!EQ.1=1 sets the input of terminal 1; -?EQ.1 reads it back.
- Several items are joined with /, for example -!EQ.1=1/XAI.1.2=2450.
- -?#500/EQ.1 repeats a query every 500 ms; commands can repeat the same way.

Inputs that can be simulated: the 16 terminals, the Boot button state read by `eqsp32.readUserButton()`, the inputs of EQXIO, EQXAI, EQXTC, EQXPT, EQXPH and EQXHI modules, the clock and the EQTimer time. By default the outputs stay off while HIL test mode runs (safe mode); with `eqsp32.beginHil(false)` they are driven normally.

Table 21-1. Library reference: Hardware-in-the-Loop (HIL) Test Mode

Configuration	
`eqsp32.beginHil(ss)`	Enables HIL test mode; call it before `eqsp32.begin()` Returns true if HIL mode was enabled; false if called after `eqsp32.begin()` ss: true (default) keeps the outputs off; false drives them

© 2026 Erqos Technologies P.C. · erqos.com

22 Expansion Modules

EQSP32 has an expansion connector for adding additional functionality for niche and tailor-made applications.

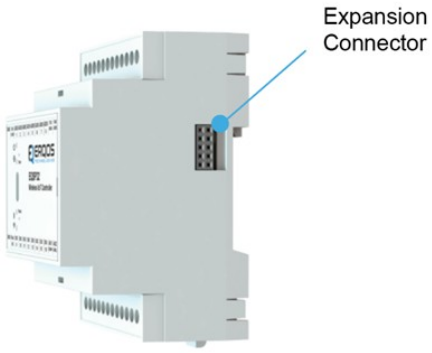

Figure 22-1. Expansion connector

> **WARNING**
> Before connecting any expansion module, make sure that EQSP32 is powered off.

To add an expansion module, mount it on the DIN rail next to EQSP32 and gently slide it until the expansion module's connector is completely inserted in the main EQSP32 controller.

After powering up the EQSP32, the expansion module detection will run automatically on boot by the proprietary EQSP32 library. Once boot is done, all additional values will be available locally.

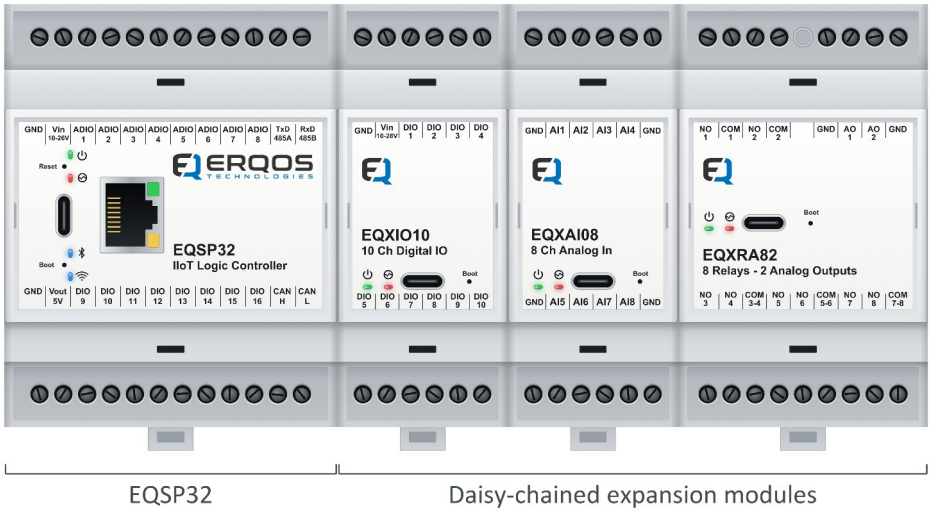

Figure 22-2. EQSP32 with daisy-chained expansion modules

22.1 Supported Modules

Table 22-1. Supported Modules

Module	Function	Inputs / outputs
EQXIO	Digital inputs and power outputs (DIN, SWT, POUT, RELAY)	10
EQXAI	Analog inputs: voltage, current and NTC temperature (AIN, CIN, TIN)	8
EQXHI	Isolated AC/DC voltage-presence inputs with frequency measurement	10
EQXPH	pH input	1

© 2026 Erqos Technologies P.C. · erqos.com

Module	Function	Inputs / outputs
EQXTC	Thermocouple inputs	6
EQXPT	PT100 inputs, 2/4-wire or 3-wire	2
EQXRA	Relay outputs and 0-10 V analog outputs	8 relays, 2 analog outputs
EQX2G	GSM/GPRS modem (one per system)	-

Up to 15 modules, in any mix, can be connected. The modules are detected at start-up. Each module type is numbered separately in the order found: the first EQXIO is `EQXIO(1)` wherever it sits in the chain.

Module channels use the same functions as the main unit's terminals, with the module and channel number, for example `eqsp32.pinMode(EQXIO(1, 3), DIN)` and `eqsp32.readPin(EQXAI(1, 2))`. The EQX2G modem is used through `eqx2gSerial` (or `eqx2gStream`) with a standard GSM library. The datasheet of each module gives its modes and values.

Table 22-2. Library reference: Supported Modules

Usage	
`eqsp32.isModuleDetected(mm)`	Returns true if the module is detected mm: module code, for example EQXPH(1) for the first EQXPH

23 Boot Button and Reset

23.1 Reset

Press and release the “Reset” button to restart the device.

23.2 Boot

A short tap on “Boot” button enables BLE advertising and EQSP32 will be discoverable from EQConnect app.

Press and hold “Boot” button for at least 3 seconds to force the device to return to default network configurations.

If no custom default configurations are set by the system's developer, the EQSP32 device will disconnect and forget the currently configured WiFi network and any static IP/DHCP parameters. Otherwise, the EQSP32 system will overwrite any user configurations with the system's developer default parameters (default network credentials and static IP/DHCP option). The MQTT enable setting also returns to its default. The device name, timezone, MQTT user name and password, and the OTA password are kept.

With `disableErqosIoT` set, the Boot button has no system function and is free for the application.

Table 23-1. Library reference: Boot

Usage	
`eqsp32.readUserButton()`	Returns true while the Boot button is pressed

© 2026 Erqos Technologies P.C. · erqos.com

24 Mechanical Dimensions

All dimensions are measured in mm with a +/-0.1 tolerance.

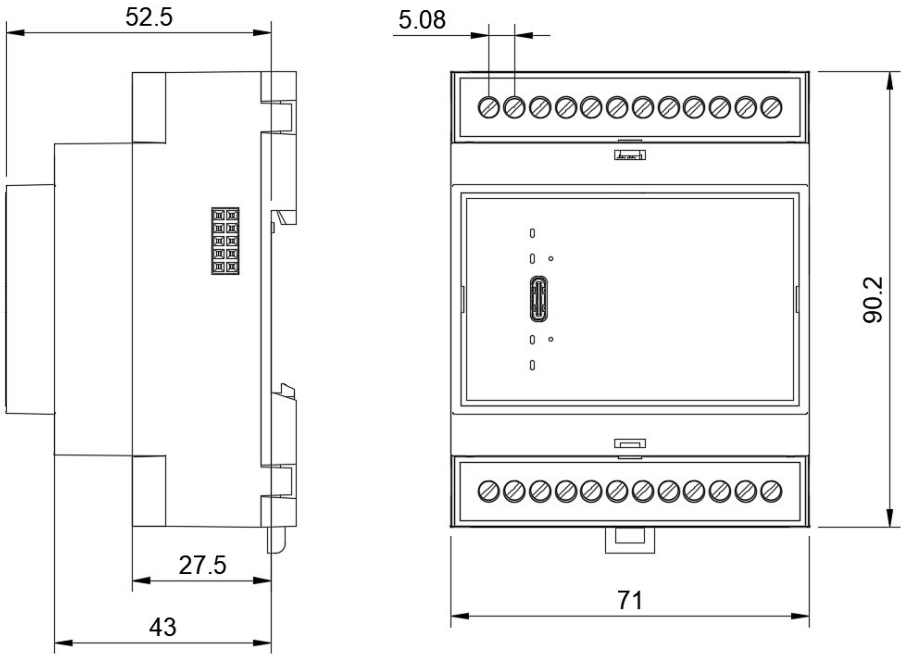

Figure 24-1. Mechanical dimensions: side and front views

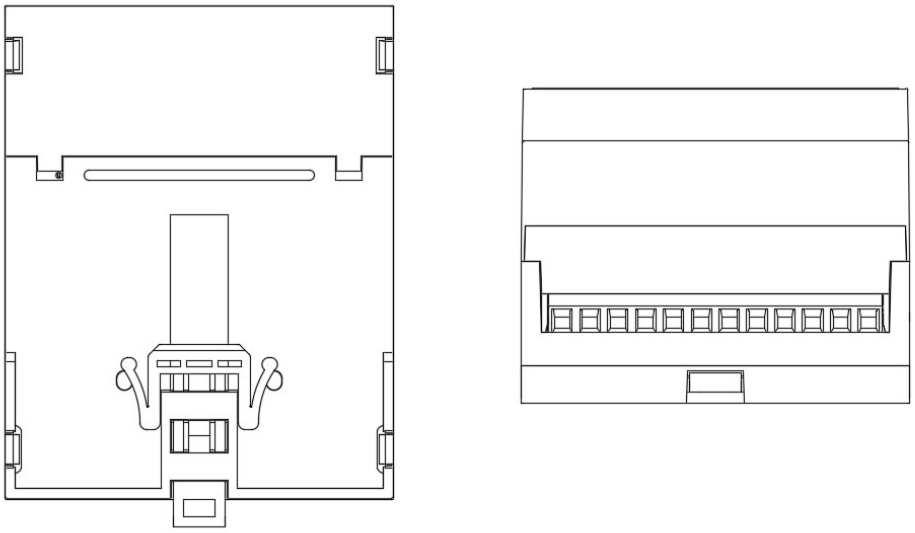

Figure 24-2. Mechanical dimensions: top and bottom views

25 Notices

Erqos Technologies P.C. may change products, specifications and this document at any time without notice. Always use the latest revision, available at erqos.com.

These products and this document are intended for qualified professionals. The information in this document is provided as is. Erqos Technologies P.C. assumes no liability for its use. Customers are responsible for designing and testing their own applications, including the protection of connected equipment and persons.

26 Revision History

VERSION	REVISION	STATUS
v1.1	**Rev102626**	**Released**

Table 26-1. Revision history

Version	Revision	Date	Status	Changes
v1.1	Rev102626	26 Oct 2026	Released	Added QENC, forcing and HIL. Completed operating conditions (analog range, input impedance, CIN resistance, PCC/QENC frequency); corrected digital input levels.
v1.0	Rev101425	14 Oct 2025	Released	First released version.

Erqos Technologies P.C.
Krinou 6, 194 00 Koropi, Greece
erqos.com