October 14, 2025

Advanced information may subject to change

ERQOS TECHNOLOGIES

EQSP32x

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

CAN RS232 RS485 DMX

EQSP32 is a compact, powerful, and user-friendly Industrial IoT ESP32 PLC controller (Industrial Internet of Things (IIoT) 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 ESP32S3, 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 Erqos' revolutionary Generative AI Programming Technology, 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, Erqos' AI Assistant can generate the necessary code for this ChatGPT PLC 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 future add-on modules, ensuring that the system can grow alongside the evolving needs of its users.

KEY FEATURES

- Dual-core ESP32-S3 SoC running at 240MHz with 512k RAM and 8M Flash
- Integrated WiFi and Bluetooth Low Energy (BLE) connectivity
- Ethernet connection (version dependent)
- Integrated 7V-26V power input DC/DC converter
- 5V 1A output for supplying user sensors and devices
- Continuous monitoring of power supplies voltages.
- LED indicator for power 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 support on RS485
- CAN bus interface
- Surge protection on RS232/485 and CAN Interfaces.
- Built in 16-channel PWM controller
- 16 EQUniversal Inputs/Outputs terminals, with multiple software-configurable modes:
    - High Speed 0-5V Logic-Level Digital input
    - Digital Inputs with Debounce Delay
    - High Speed Data, Logic Levels Output up to 1Mbit/s
    - Open Collector On/Off Outputs
    - Open Collector PWM Outputs
    - Open Collector Output Power-Optimized for relay coils
    - 0-5V or 0-10V Absolute Voltage Analog inputs (version dependent)
    - 4-20mA current loop sensor (version dependent)
    - Analog Inputs relative to the 5V Supply
    - NTC Temperature Sensor Analog Inputs
- Analog and Temperature are available on inputs 1 to 8 only
- Protected, 24V tolerant Inputs
- 12-bit Analog Inputs resolution
- 1A max per Open Collector Output
- Current limited and thermally protected Output drivers.
- Flyback diodes on each Output for safe and efficient switching of Inductive loads
- Built-in watchdog timer. Disables the Open Collector Outputs in case of software failure
- Expansion connector for up to 16 add-on modules for additional I/O or special function interface (ex. pH sensor, stepped driver, energy 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.
    - Connection of Real-Time Database
    - IO State and Variables Synchronization with Cloud Database.
- Library for efficient and transparent handling of pin, peripheral and connectivity control and configuration
- Compatible with EPSHome
- Generative AI Coding Assistant trained with System Supervisor and Library
- Works with Arduino IDE and VS Code
- Compact, 4SU-wide (70mm), DIN-rail mount enclosure with 24 screw terminals
- IP20 Protection
- -20 Min to 55o Max Ambient temperature

APPLICATIONS

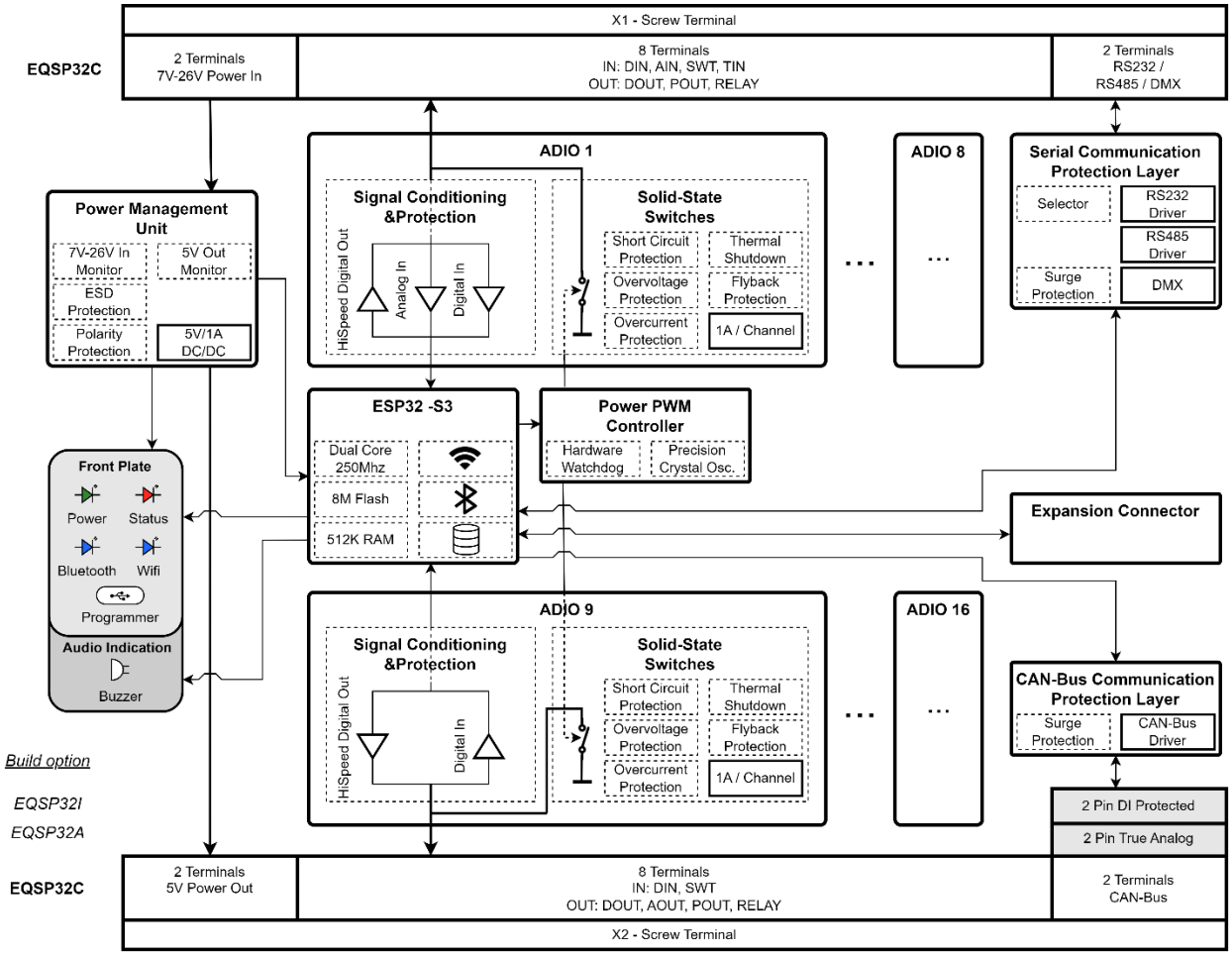
- Home Automation
- Industrial Monitoring & Control
- HVAC Systems
- IoT Automation
- Environmental Monitoring
- Agricultural Applications
- Energy Management
- Healthcare Monitoring
- Retail and Inventory Management
- Robotics and Automation
- Educational Projects
- Entertainment and Art Installations

EQSP32 PRODUCTS AND FEATURES LIST

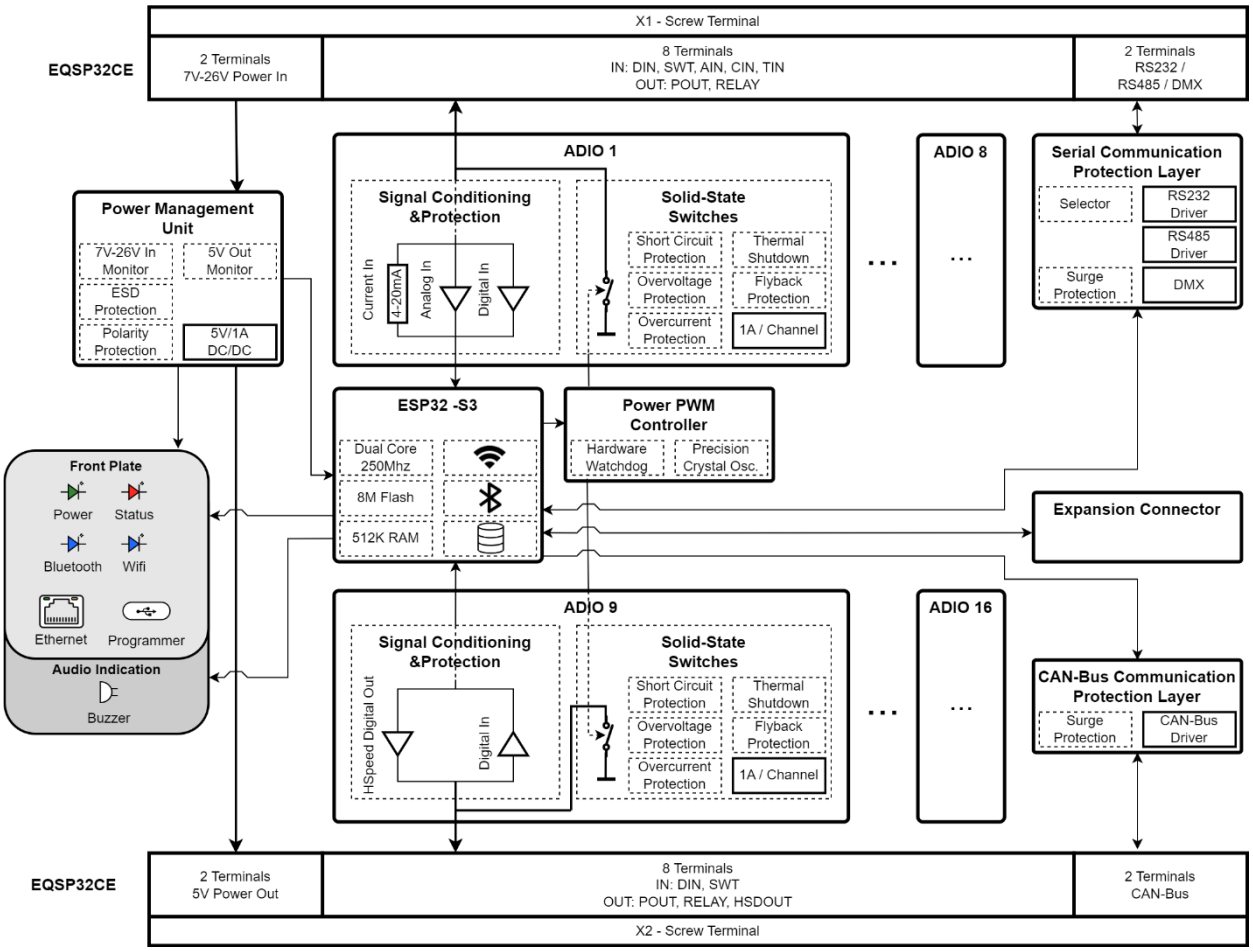
Module	EQSP32C	EQSP32C-X	EQSP32CE
Processing Unit			
MCU	ESP32-S3 Mini	ESP32-S3 Mini	ESP32-S3 Mini
Core	Dual Core x240MHz	Dual Core x240MHz	Dual Core x240MHz
RAM Memory	512 kB	512 kB	512 kB
Flash Memory	8 MB	8 MB	8 MB
I/O Capabilities			
Total I/Os	16	16	16
Digital Inputs	16	16	16
Digital Outputs (Solid State)	16 PWM (1A each)	16 PWM (1A each)	16 PWM (1A each)
Analog Inputs (Voltage)	8 (0-5V)	8 (0-5V)	**8 (0-10V)**
Analog Inputs (Current)	No	No	**8 (4-20mA)**
Protected Terminals	✓	✓	✓
Wired Connectivity			
Ethernet	No	No	✓
CAN Bus	✓	✓	✓
RS485 (Serial)	✓	✓	✓
RS232 (Serial)	✓	✓	✓
Wireless Connectivity			
Wi-Fi (2.4 GHz)	✓	✓	✓
Bluetooth (BLE)	✓	✓	✓
Antenna	Internal	**External**	Internal
Other Features			
Supply Input Monitoring	✓	✓	✓
Supply Output Monitoring	✓	✓	✓
Mounting	DIN Rail	DIN Rail	DIN Rail
Expansions	✓	✓	✓
Number of Expansions	15	15	15

EQSP32 PLC BLOCK DIAGRAMS

EQSP32C (-X)



EQSP32CE


X1 - Screw Terminal
EQSP32CE
2 Terminals
7V-26V Power In
8 Terminals
IN: DIN, SWT, AIN, CIN, TIN
OUT: POUT, RELAY
2 Terminals
RS232 /
RS485 / DMX
Power Management Unit
7V-26V In Monitor
5V Out Monitor
ESD Protection
Polarity Protection
5V/1A DC/DC
ADIO 1
Signal Conditioning &Protection
Current In
4-20mA
Analog In
Digital In
Solid-State Switches
Short Circuit Protection
Thermal Shutdown
Overvoltage Protection
Flyback Protection
Overcurrent Protection
1A / Channel
ADIO 8
Serial Communication Protection Layer
Selector
RS232 Driver
RS485 Driver
Surge Protection
DMX
ESP32 -S3
Dual Core 250Mhz
8M Flash
512K RAM
Power PWM Controller
Hardware Watchdog
Precision Crystal Osc.
Front Plate
Power
Status
Bluetooth
Wifi
Ethernet
Programmer
Audio Indication
Buzzer
Expansion Connector
ADIO 9
Signal Conditioning &Protection
HSpeed Digital Out
Digital In
Solid-State Switches
Short Circuit Protection
Thermal Shutdown
Overvoltage Protection
Flyback Protection
Overcurrent Protection
1A / Channel
ADIO 16
CAN-Bus Communication Protection Layer
Surge Protection
CAN-Bus Driver
EQSP32CE
2 Terminals
5V Power Out
8 Terminals
IN: DIN, SWT
OUT: POUT, RELAY, HSDOUT
2 Terminals
CAN-Bus
X2 - Screw Terminal

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 and VS Code (using PlatformIO)
Processor	Dual-Core XTensa LX7 MCU
	240 MHz frequency
	Dedicated user program processor
	512K RAM / 8M Flash memory
Power Management	Integrated DC/DC, 7V to 26V DC power input
	Supply output 5V @ 1A for powering external peripherals
	Input and output supply monitoring
I/O Terminals	16 x Multipurpose I/Os (digital/analog input/output)
	TTL levels
	Over voltage protection
	8 x 12-bit ADCs
	16 x Pull-Down Power PWM Outputs
Power PWM Output	16 x 1A power Pull Down PWM outputs
	Externally supplied up to 26V 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
	CAN Bus
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

CERTIFICATION AND COMPLIANCE

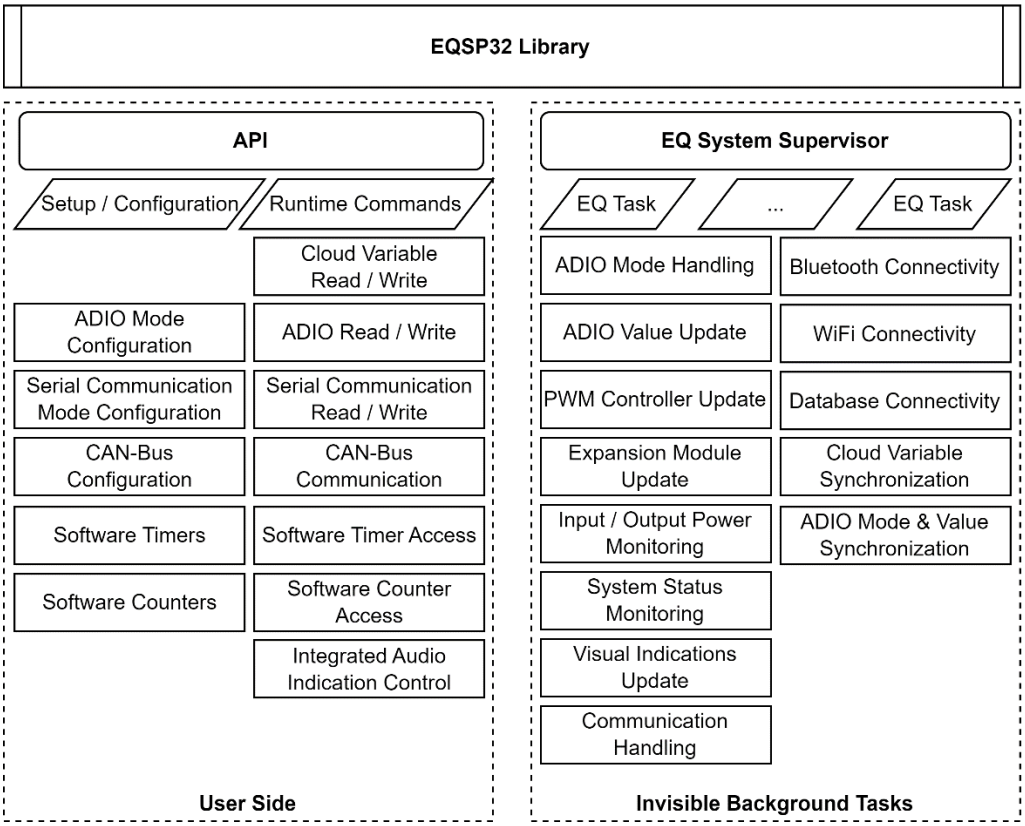
Applicable Standards for CE Compliance under Directive 2014/53/EU (RED):

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)

SOFTWARE ARCHITECTURE

EQSP32 utilizes the EQSP32 proprietary library to automatically handle all wireless connectivity and database 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 seconds all the invisible tasks running on the background, being handled by the EQ Task Manager.



EQSP32 Library Block Diagram

ELECTRICAL SPECIFICATIONS

Absolute Maximum Values

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!

Recommended Operating Conditions and Typical Values

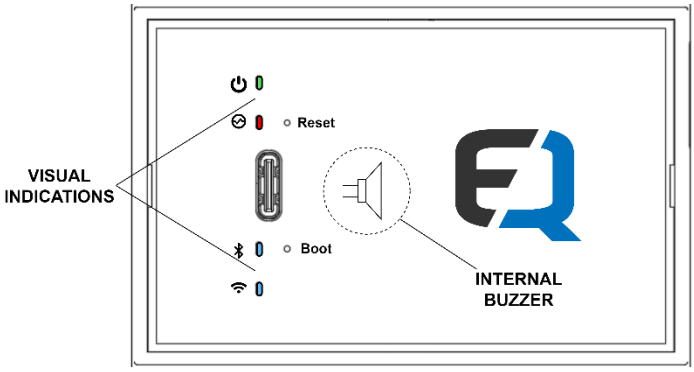
Pin		Type	Min.	Typ.	Max.	Unit
Supply In +		Power In	7	12 to 24	26	V
5V Out +		Power Out	4.98	5	5.02	V
					1	A
RS232 Tx		Out	-12		12	V
RS232 Rx		In	-12		12	
RS485 A		In/Out	0		5	V
RS485 B						
CAN H		Out (dominant)	2.75	3.5	4.5	V
CAN L		Out (dominant)	0.5	1.5	2.25	
CAN L to CAN H		In (differential)	0.5	0.7	0.9	
ADIO	High (1)	In/Out	3.3	5 to VIn	VIn (*)	V
	Low (0)		-0.5	0	1.3	
	Analog	In	0		5.5	
	Power PWM	Out			1	A

VISUAL AND AUDIO INDICATIONS

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

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

The internal buzzer will sound once at the beginning of the power cycle, indicating that the library is running. The user may utilize the audio indication in their program from the library's API.



Power LED

On	- Powered On	
Off	- No Power	

Status LED

Off	- No errors	
Blink	- Supply under voltage	
Flashing	- Supply over voltage	
Blink	- Internal fault	
On	- Default timezone configs	

Bluetooth LED

Off	- Bluetooth disconnected	
Flashing	- Bluetooth advertising	
On	- Bluetooth connected	

WiFI LED

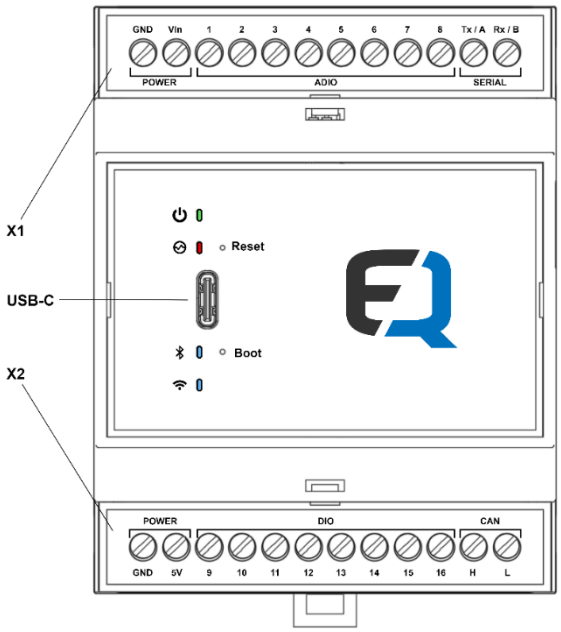
Off	- WiFi disconnected	
Flashing	- WiFi scanning/trying to connect	
On	- WiFi connected	

PINOUT

X1 – Screw terminal connector 1

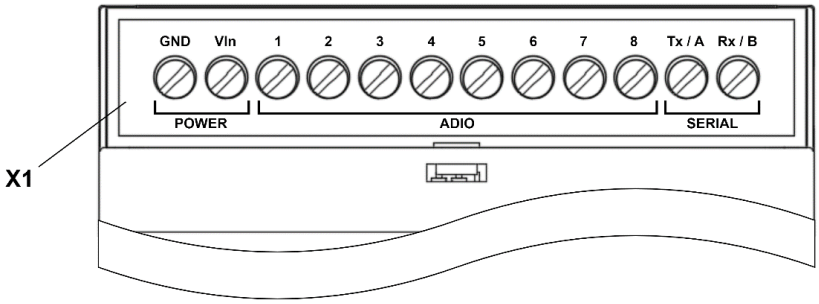
USB-C – Programmer port

X2 – Screw terminal connector 2



X1 – Screw terminal connector 1

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



Name	Pin	Function	Type
X1	GND	DC Power Input - (GND) (*)	POWER IN
	VIn	DC Power Input + (10V-24V)	POWER IN
	1, …, 8	Multipurpose I/O (**)	ADIO
	Tx / A	RS232 Tx / RS485 A	SERIAL
	Rx / B	RS232 Rx / RS485 B	SERIAL

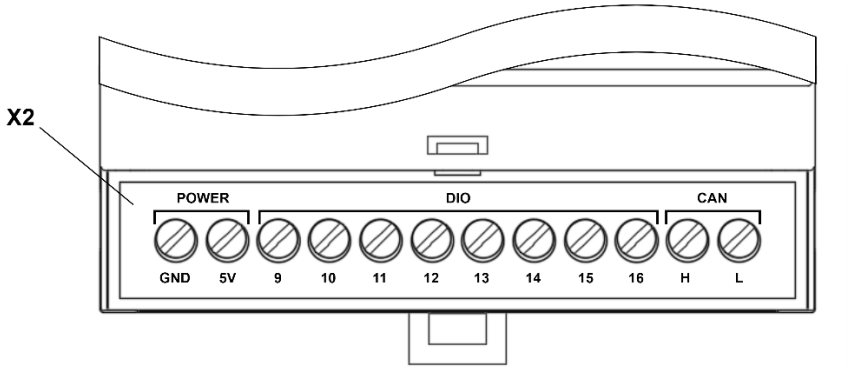
Notes:

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

(**) X1 ADIO pins also support analog input (AIN) functionality.

X2 – Screw terminal connector 2

X2 connector accommodates the power supply output pins, the CAN-Bus communication (or optional analog output) and 8 ADIO pins.



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

Notes:

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

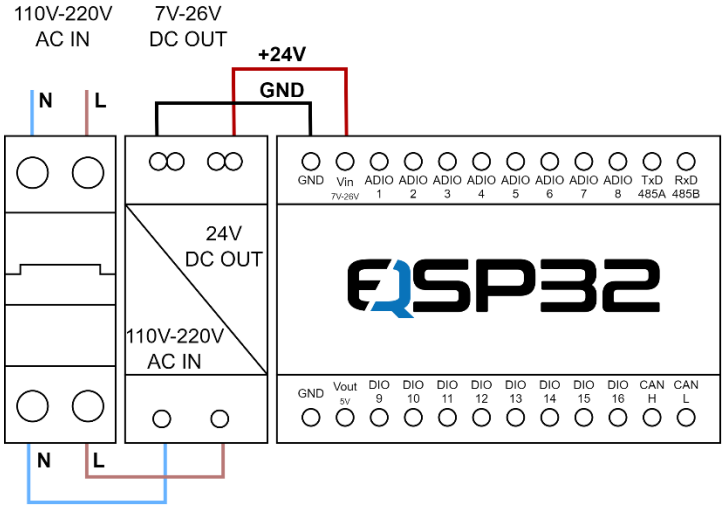
POWERING EQSP32

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

When programming the EQSP32, it is important that the VIn power is cut and EQSP32 is running only on USB-C's 5V.

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.



Circuit breaker and AC/DC wiring

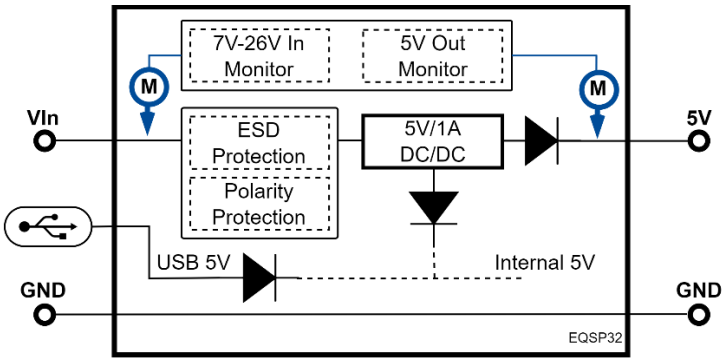
Warning: Make sure that the appropriate ground fault circuit breaker, fuses or circuit breakers and any other necessary protective components are used if there is any risk of user electrocution.

POWER MANAGEMENT

Power distribution and monitoring

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

EQSP32 is also powered from the USB 5V but this voltage is only internally used. When programming EQSP32 it is recommended that the external VIn is shut down and only the USB 5V is used.



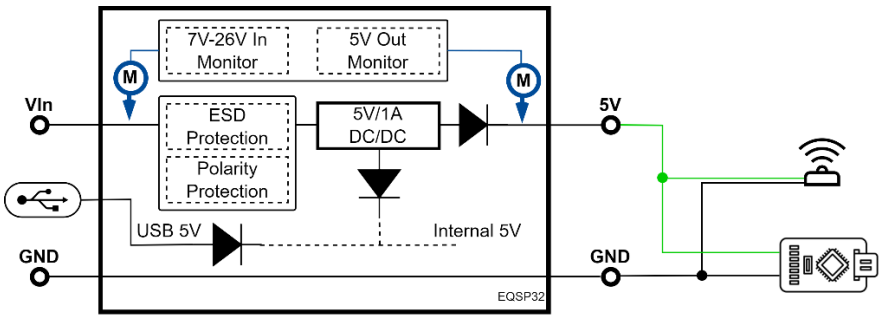
Power distribution and monitoring points (M)

Library Reference	
Usage	
eqsp32.readInputVoltage()	Returns the measured voltage on VIn (7V-26V) in mV
eqsp32.readOutputVoltage()	Returns the measured voltage on VOut (5V) in mV

Integrated 5V power supply

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

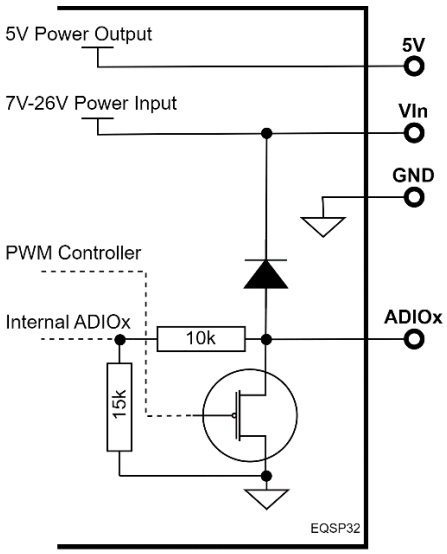
The integrated 5V power supply is heavily filtered and monitored. The user has access to the internal 5V analog readings and it is recommended for best accuracy that this supply is used as reference.



Example: *Powering external devices and sensors using integrated 5V power supply*

MULTIPURPOSE I/O & POWER PWM OUTPUT – ADIO

Equivalent ADIO Circuit



Equivalent ADIO Circuit

ADIO Modes Overview

All ADIO pins support DIN, DOUT 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.

Mode	Type	Function
Digital Input (DIN)	Input	Digital
Switch Input (SWT) – {Special DIN}	Input	Digital
Pulse Capture Counter (PCC) – {DIO 9, …, 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

Digital Input (DIN)

When in digital input mode (DIN), the pin reads 1 (HIGH) if a voltage between 2.5 V and 24 V is applied, or 0 (LOW) if the voltage is between 0 V and 1.3 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.

Library Reference	
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)

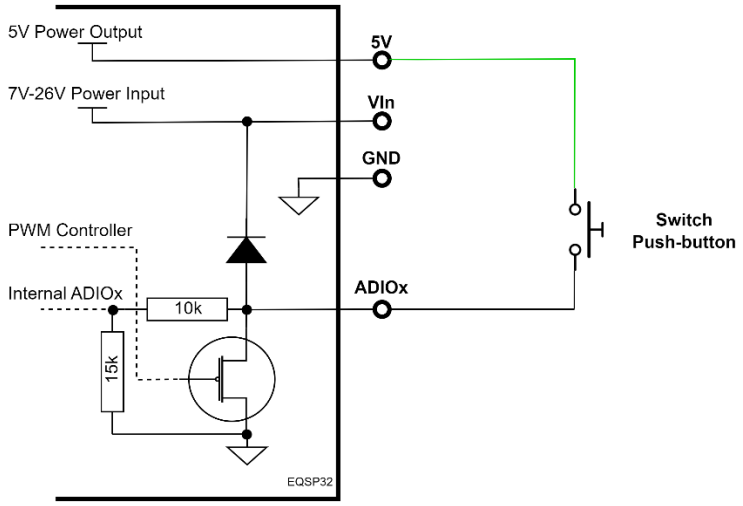
Switch Input (SWT) – {Special (DIN) mode}

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.



Example: *Push-button on EQSP32's 5V out*

Library Reference	
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)

(*) If this function is not called, the default values are applied, meaning the SWT pin's debounce time will be set to 100ms.

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 100ms, 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.

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.

Pulse capture counter (PCC) mode is supported for up to any four pins of terminals 9–16 simultaneously. If a fifth pin is configured, the configuration function returns false and the PCC setup is ignored.

The counting behavior depends on the selected trigger mode:

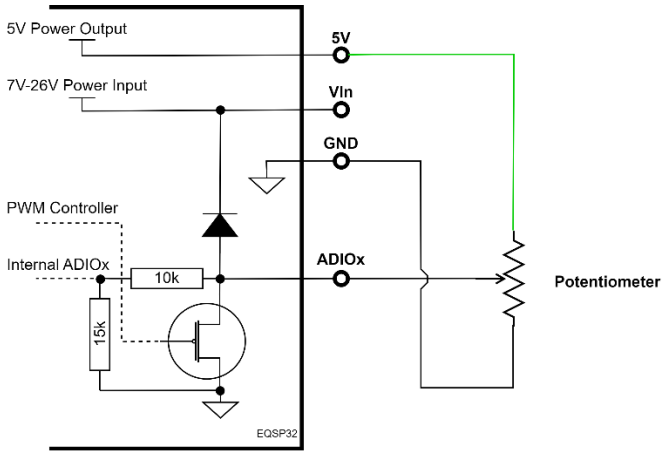
- ON_RISING – counts rising edges (default).
- ON_FALLING – counts falling edges.
- ON_TOGGLE – counts both rising and falling edges.

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

Library Reference	
Configuration	
eqsp32.pinMode(nn, PCC)	nn: pin index
eqsp32.configPCC (nn, mm)	nn: pin index mm: pulse capture counter trigger mode, ON_RISING (default), ON_FALLING or ON_TOGGLE.
Usage	
eqsp32.readPin(nn)	Returns the accumulated pulse counts since last read. nn: pin index

Analog Input (AIN)

The analog input mode (AIN), measures the analog value of the pin from 0V to 5V and returns the analog value in mV.



Example: *Potentiometer with EQSP32's 5V out as reference voltage*

Library Reference	
Configuration	
eqsp32.pinMode(nn, AIN)	nn: pin index
Usage	
eqsp32.readPin(nn)	Returns analog value in mV nn: pin index

Current Input (CIN)

The current input mode (CIN) measures 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 of compatible hardware.

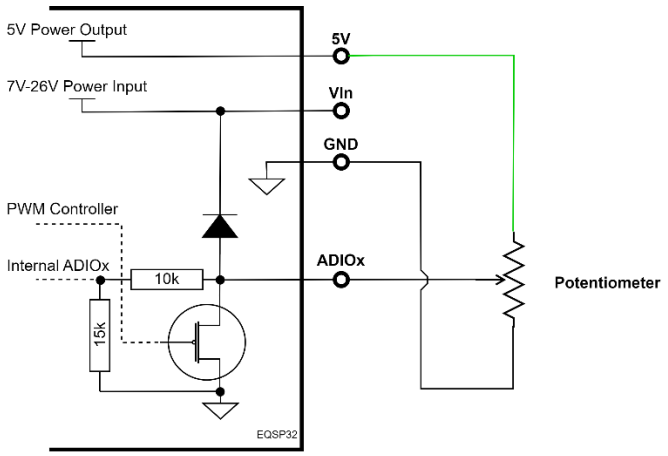
If the input current exceeds 21 mA, the system enables automatic overcurrent protection and the function returns -1. In this state, the shunt resistor is disabled to prevent damage. At regular intervals (every few milliseconds), the system briefly reconnects the shunt resistor to check whether the current has returned to a safe range. If the current remains above 21 mA, protection stays active until normal operating conditions are restored.

Library Reference	
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

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 5V output supply voltage.

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.



Example: *Potentiometer with EQSP32's 5V out as reference voltage*

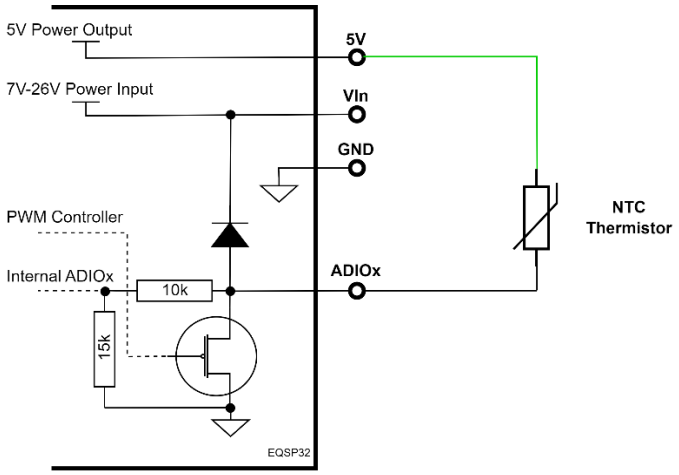
Library Reference	
Configuration	
eqsp32.pinMode(nn, RAIN)	nn: pin index
Usage	
eqsp32.readPin(nn)	Returns a 0 to 1000 value representing 0% to 100% of the 5V supply out reference nn: pin index

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.



Example: *NTC Thermistor using EQSP32's 5V out as reference voltage*

Library Reference	
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)
Usage	
eqsp32.readPin(nn)	Returns the temperature in 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.

(*) 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 3988, if omitted.

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

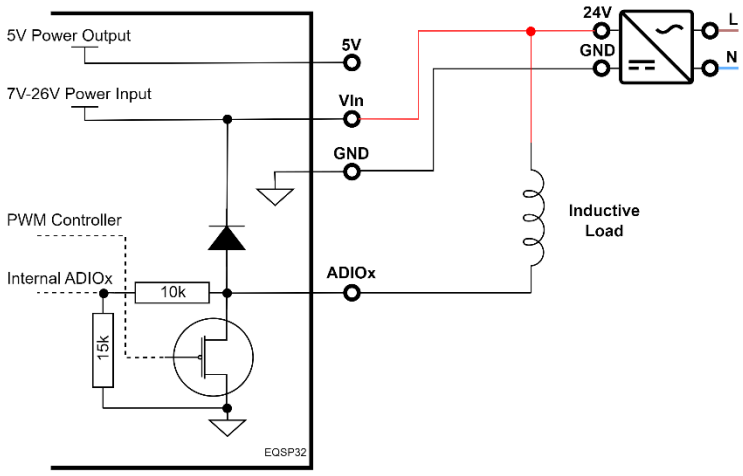
Power PWM Output (POUT)

The power PWM output mode (POUT) is meant to be used to drive up to 1A loads per terminal. In this mode the pin operates in pull-down (open collector) PWM.

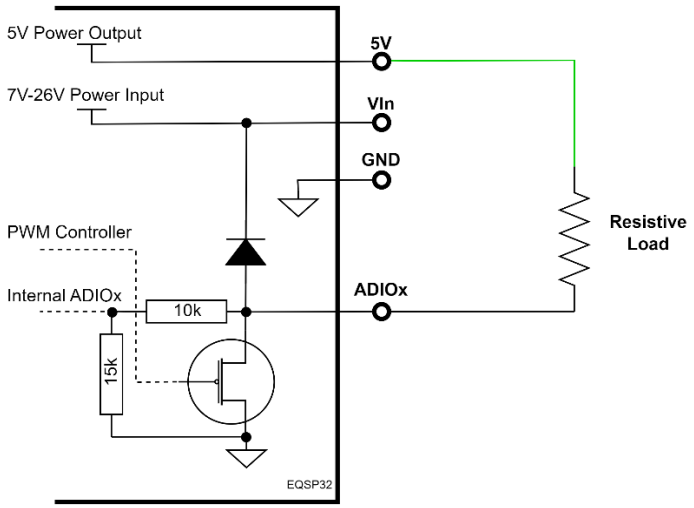
Inductive and Non-Inductive loads

EQSP32 incorporates protective diodes for protecting the mosfets from the overvoltage spikes induced by the switching of relays or solenoids. Thus, indictive 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.



Example 1: *Driving an inductive load on a common power supply*



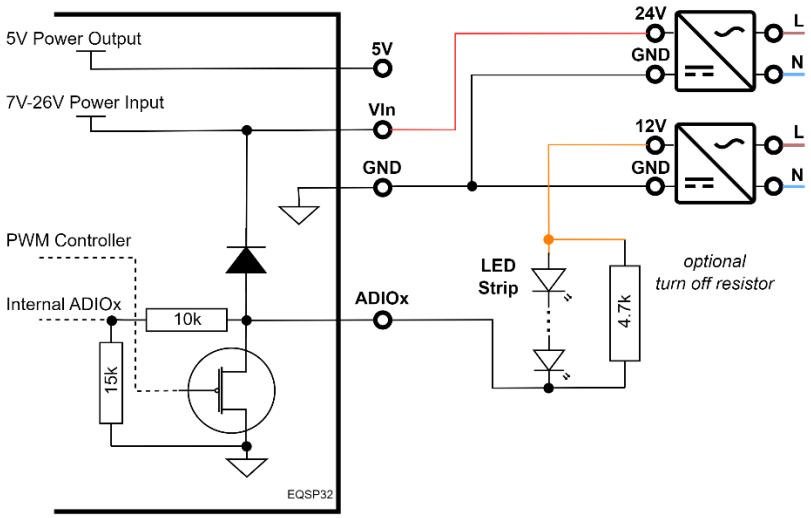
Example 2: *Driving a resistive load from the integrated 5V Out power supply*

Driving LED Strips

Due to LEDs lighting up with even a very small current, it is recommended that an optional external pull up is used between the anode and the cathode of the LED strip.

The value of the resistor depends on the efficiency of the LEDs, their number and other factors. Generally, an 1k, 0.5W resistor would be appropriate for most cases.

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



Example 3: *Driving a 12V LED strip from separate 12V power supply*

Library Reference	
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.
Usage	
eqsp32.pinValue(nn, pp)	nn: pin index pp: power per 1000 (500 for 50%, 1000 is for 100%)

(*) 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.

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.

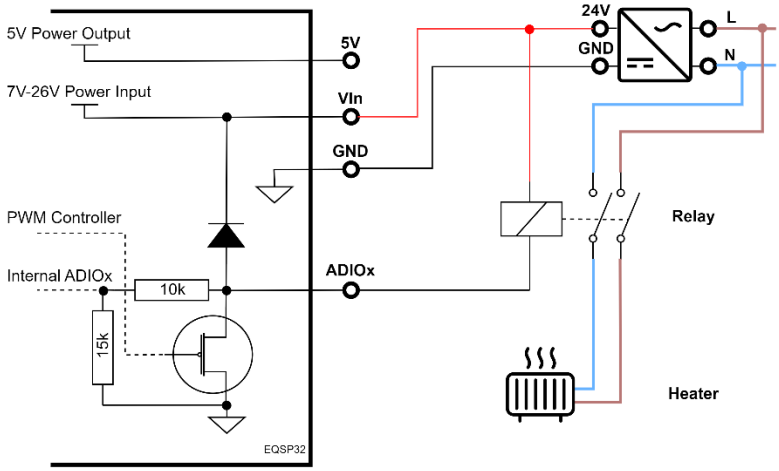
Minimum holding PWM value = (Vdisengage / Vsupply) * 1000

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

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

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 that the disengaging voltage.

Desired holding PWM value = 250 + 250 x $\frac{1}{2}$ = 375



Example: *EQSP32 controlling a 24V Relay to drive a 220V heater*

Library Reference	
Configuration	
eqsp32.pinMode(nn, RELAY)	nn: pin index
eqsp32.configRELAY(nn, hh, tt) (*)	nn: pin index hh: hold power (optional, default = 250) tt: derate time ms (optional, default = 1000)
Usage	
eqsp32.pinValue(nn, pp)	nn: pin index pp: initial power (1000 is for 100%)

(*) If this function is not called, the default values are applied, meaning the RELAY pin's hold power will be set to 25% (hh = 250) and the derate time will be set to 1000 (1 second).

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 250.

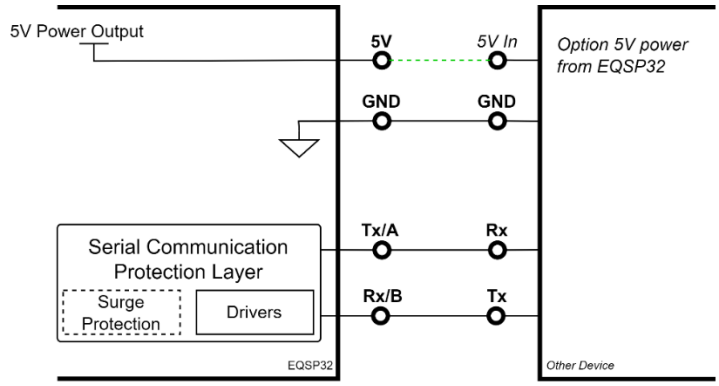
tt: Is the time after which the relay's power will drop to holding power in ms. This is an optional parameter with default value of 1000ms, if omitted.

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.

Serial - RS232

Standard RS232

For standard RS232 communication, connect EQSP32's RS232 Tx pin with receiver's RS232 Rx pin and vice versa.

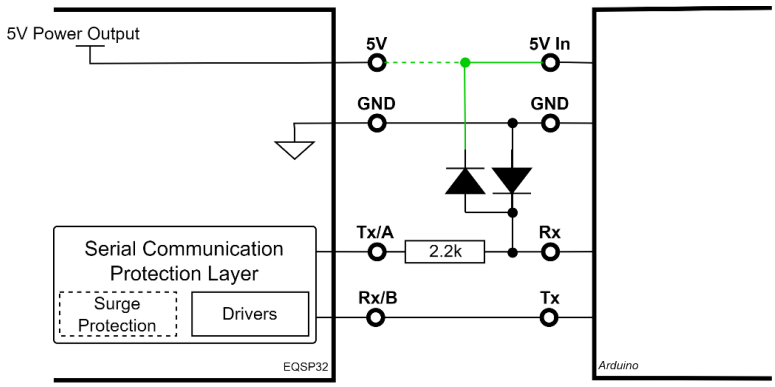


RS232 wiring

RS232 to TTL

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

Use external clipping diodes connection the RS232 Tx signal to TTL device's 5V 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 is must be used.



RS232 to TTL wiring

Note: ***RS232 signals are inverted by the RS232 driver.***

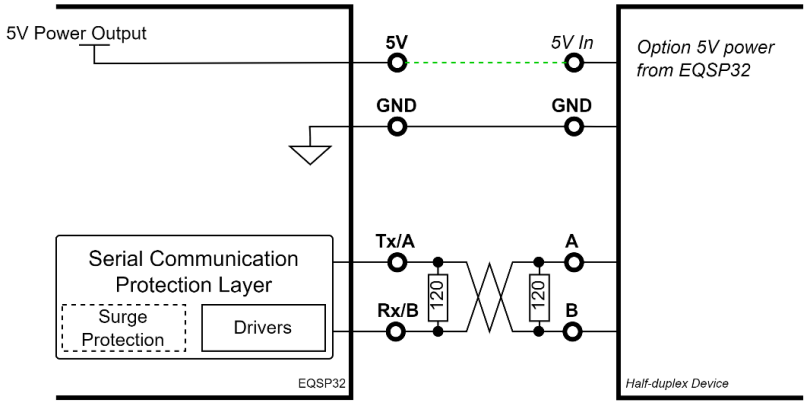
In order for communication to work, apart from clipping the signals to TTL levels, it is also needed to setup the communication on either the EQSP32 or TTL device as INVERTED. For EQSP32 use RS232_INV as serial mode to invert the signal polarity.

Library Reference	
Configuration	
eqsp32.configSerial(mm, bb)	mm: serial mode RS232 or RS232_INV for inverted polarity bb: baude 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

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.



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** (680 Ω – 1 kΩ) from **A → VCC**
- **Pull-down resistor** (680 Ω – 1 kΩ) from **B → GND**

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

Use **Fail-Safe Biasing** when:

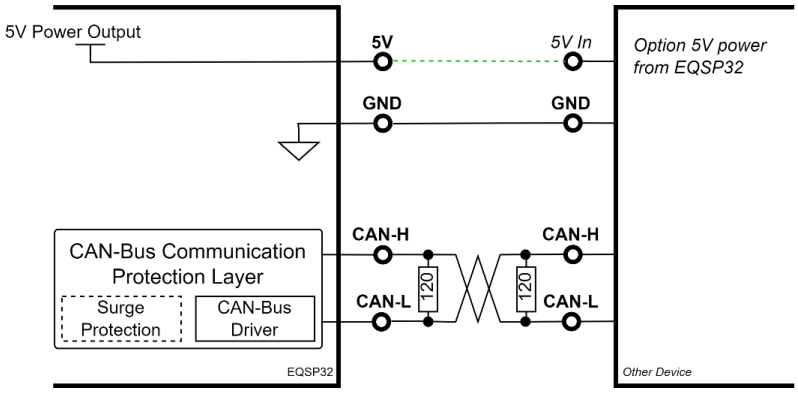
- Operating at baud rates above 9600 bps (especially 19200 bps and higher).
- The bus is long (over 50 m) or operates in electrically noisy environments.
- The network includes many passive (listening-only) nodes.

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

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.



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.

Library Reference	
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)
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.

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.

Library Reference	
	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 TX pin number)

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 transmission.*

// Put your custom setup and usage of RS232 by using rs232Tx and rs232Rx pins directly

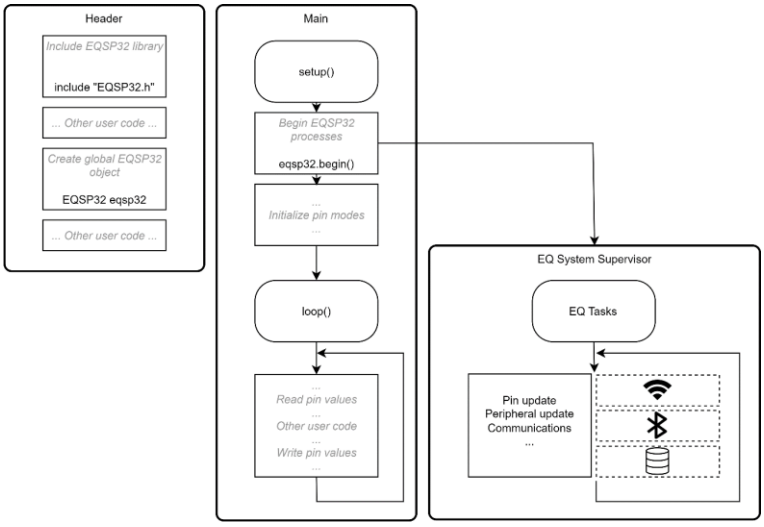
PROGRAMMING, EQSP32 CONFIGURATIONS AND DEFAULT PARAMETERS

The EQSP32 comes with a precompiled proprietary library that provides the user with an API to access all ADIO modes, communications, internal monitoring and database 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, Bluetooth and WiFi connectivity, database connection and synchronization.

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 pal, 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.



Program flow with EQSP32 library

Library Reference	
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

Differences between begin() variations

1. **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.

2. **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.

3. **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

}
```

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 and may not be directly changed by the developer during system runtime operation.

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.

Library Reference			
EQSP32Configs			
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 (leave empty for unencrypted).
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.
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 provisioning and cloud management. **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.

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.

Automatic Network Management

During startup, the EQSP32 automatically initializes all available network interfaces.
The internal network manager continuously monitors connection status and selects the most stable and available interface with priority on 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 the .disableNetSwitching = true; from the EQSP32Configs struct.

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.

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.

Ethernet Connectivity (Model-Dependent)

On EQSP32 models equipped with Ethernet hardware, the controller provides a wired network interface over a standard RJ45 connector.
Ethernet is initialized automatically when a cable is detected and functions seamlessly with the Wi-Fi interface.

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.

- **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.

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 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, setup local timezone with daylight savings, wirelessly monitor EQSP32 I/O modes, states and all connected expansion modules.

The "IoT Configuration" section offers additional setup options for defining a static IP or DHCP, updating the OTA (Over-The-Air) updates' password 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.

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.

Notes: 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.

BOOT & RESET

Reset

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

Boot

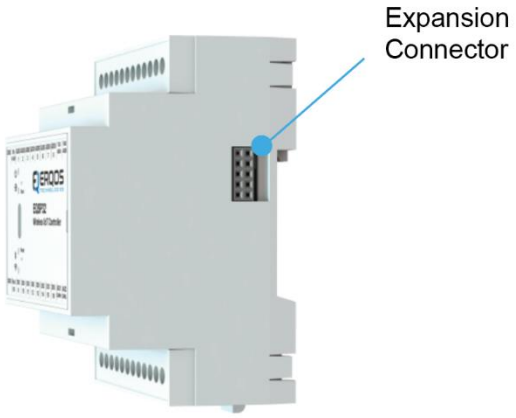
A short tap on “Boot” button enables BLE advertising and EQSP32 will be discoverable from EQConnect app.

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).

EXPANSION MODULES

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



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 and in the database.

EQSP32 Daisy-Chained Expansion Modules

MECHANICAL DIMENSIONS

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