Industrial Open Source

All you need to know about

Industrial Shields®

INDEX

April 2022

Open Source. The origins

History

Between the 50s and 60s, the Internet technologies had a collaborative and open environment. The **Advanced Research Projects Agency Network (ARPANET)**, which is what we now know as today's **Internet**, fostered collaborative and open work and peer groups shared source code.

Thus, by the time the modern Internet arrived, the open source initiative was already well established through collaboration and peer-to-peer work.

Software

Open Source Software is a **source code** created with the intention that everyone can freely **examine, edit or distribute it.** Therefore, any user will have access to it and will be able to read it and modify it, thus improving it.

Additionally, another concept exists. **FOSS (free and open-source software)** is software that is **both free and open-source**, meaning that anybody can use, copy, study, and modify it. The source code, on the other hand, is publicly published, encouraging everyone to **enhance the software's design.**

The **source code** is the **component of the software** that is not visible at first look; it is the code of a program or an app that programmers write to make the software run correctly and fulfill its role. Programmers with access to a program's source code can improve it by a**dding features, optimizing the code, or fixing errors or bugs** left over from prior versions.

This type of software is developed in a **decentralized and collaborative** manner, so it relies on **community review**. Furthermore, because it is produced by communities rather than by a single author or a single organization, it is frequently **less expensive or even free, more flexible, and more durable** than proprietary alternatives.

Regarding **costs, neither of the two concepts refers to this,** since both can be legally distributed for free or at a cost. But still, the main difference is not the cost, but the fact of sharing the source code itself.

In the case of **Free Software**, all changes made to the source code and subsequently distributed must be shared with the original project. But in the case of Open Source Software, it is not necessary to do so.

Open source software benefits everyone, not just programmers, because it allows many more people to generate innovations, compared to closed source models. Their communities, for example, are centered on open source projects to which anybody with programming skills can contribute code.

Main Features & Benefits

Main Features

- The program must have the **source code**. It is not allowed that the source code does not appear or that only part of it appears, it must appear completely and modifications must be allowed.
- You **cannot restrict the license to anyone**, everyone must have access to it. If for whatever reason there are any restrictions, they must be mentioned.
- The **software license** must **allow modifications and changes.**
- It can be **copied, modified or given away** to third parties at **0 cost.**
- The **integrity** of the **author's code** may be **required**, i.e. modifications must be fully visible.

Benefits of using Open Source Software

- Software can be tailored to the **specific needs** of a company.
- **Sharing, modifying and understanding** software source code.
- **Support** of forums and computer communities.
- Promotes the **collaboration of users** and the community of programmers.
- It makes the **detection, improvement and correction of errors** more efficient.
- **No license purchase costs.**
- Greater possibility of software **continuity** because it belongs to the community.

The entry of open source technology into industrial environments

The computer environment has evolved significantly in the last few years. In 2005, the launch of the [Arduino] board as an **Open Source Hardware** brought a total change in Open Source functionalities.

Open Source Software has been the key for initiatives with **different applications and uses.** Also, for **Industry 4.0** and the **Internet of Things**. A clear example is the way Industrial Shields® works; Open Source is present from the industrial automation of the production plant or the ERP platform we use, to the software and hardware of our products. At Industrial Shields®, we have adapted **Open Source technology** in the way of **automating, controlling or monitoring any process.**

The entry of open source technology into industrial environments

The increased usage of the Internet in industry is another cause for the **growth of Open Source Software**. The **Industrial Internet of Things (IIoT)** has boosted the usage of **HTML5**, the most recent version of the hypertext markup language, as well as **Open Source TCP/IP** versions. The use of the MQTT open messaging protocol has also been magnified.

The openness of this software, **allows the use of free software for individual or collective tasks or applications**. This helps to reduce the development costs and risks, it enhances flexibility in response to changing production conditions and reduces complexity by using existing solutions. When important circumstances are taken into consideration, the benefits exceed the disadvantages.

Control systems keep things running and assure the right execution of production processes for manufacturing lines, machine tools or other automated processes. Some years ago, you could only find PLCs designed for specific purposes. Moreover, their software was closed and the modifications were only allowed to a limited extent. This was sufficient for those applications that were predictable and always the same, but as we mentioned earlier, things are changing. Now, the productions are **more flexible** and there are **smaller batch sizes**, so control systems must also be flexible. Both machine makers and responsible individuals of production businesses are increasingly relying on open control platforms for this **flexibility and sustainability**.

Machine tool makers, for example, may have **complete control** over their devices **thanks to open source code**. As a result, they know exactly when and how the controller accomplishes what. There is no such insight and control for proprietary control systems. Furthermore, the code is available for an indefinite period of time and may be tailored to your specific demands and security standards. Consequently, producers avoid being reliant on control system product discontinuations.

In the meanwhile, Linux is the most widely used operating system for contemporary control systems and IoT devices. Users may accomplish control tasks using standard **IEC 61131 libraries**, as well as integrating open source packages or direct programming using **C/C++**, thanks to the open operating system. For nearly any operation or function, ready-made programs and source code for personalization are now available thanks to the open source.

In addition, the **Linux Community** is continually improving and releasing new versions of source codes, libraries and programs in openly available repositories, such as on the platform GitHub. The community supports issues, but also expects to be a part of the solution.

One of the key reasons why automation engineers are increasingly reconsidering is because open source programs have **openly accessible source code**, they often reach a significantly larger number of developers than proprietary or closed software.

Therefore, open source software typically outperforms closed software in terms of runtime stability and quality and these elements are crucial in industrial applications.

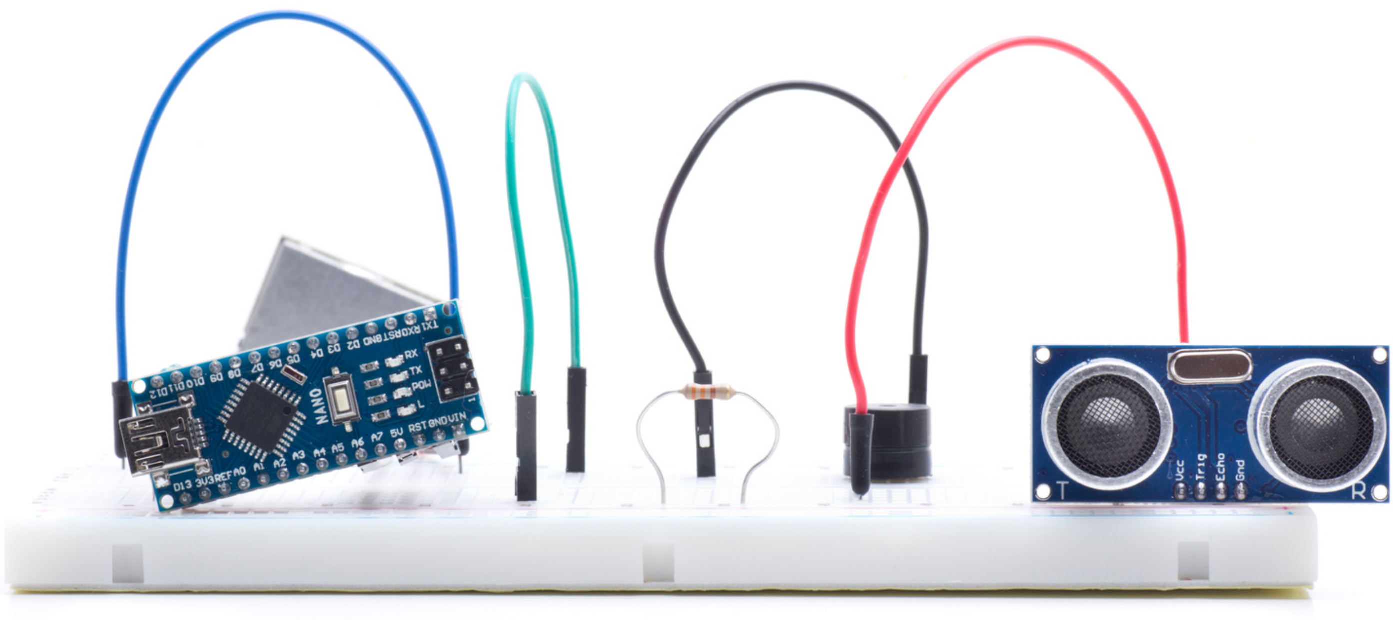
Hardware

The open source philosophy is also applicable to hardware. **Open Source Hardware** includes **machinery or electronic devices whose specifications and schematic diagrams** are **publicly** available. So, it means that anyone can study, modify, distribute and sell the design or hardware based on that design, **either free of charge or for a fee**. This means that all those original design (hardware) files must be shared. Therefore, we find that schematics, logic designs, HDL source code, CAD files, PCB design files, blueprints, G-codes, bill of materials and documentation are all shared.

OSHWA is the Open Source Hardware Association. As the name implies, it is a non-profit organization that supports open source hardware.

Industrial Shields® works with **open hardware**, but that doesn't mean that our design is made publicly available for anyone to study, modify and distribute. We build on top of **existing, well-designed, and well-tested open hardware,** giving it a more specific and narrow focus. This hardware can run a big number of proprietary programs, but it can also run a longer list of open source programs. Then, this lets consumers have a high-quality industrial product at an affordable cost and a lot of flexibility.

As we mentioned above, with the launch of **Arduino** there were many changes. Arduino is the perfect example to explain what Open Source is, **both Software and Hardware**. Arduino is a company that was born in Italy in 2005. It is a company Open Source Hardware and Software, which means that both the **source code (software) and the design (hardware)** are publicly available, in the sense that anyone should be able to view, study, understand its operation, make changes and share those changes.
Other popular examples of Open Source Hardware based products include **Raspberry Pi, RepRap, E-puck and Open Source Ecology**.



Hardware

According to the Open Source Hardware Association, the **following conditions and criteria** must be met. They are briefly explained:

- **Documentation**

Complete documentation with modification permissions must be submitted.

- **Scope**

The hardware documentation must explicitly state which parts of the design, if any, are being distributed under the license.

- **Necessary Software**

The software used must offer documentation and must be delivered under an open source license.

- **Derived Works**

Distribution of modifications and sale of products developed from the design should be allowed.

- **Free redistribution**

The license should not prevent a third party from selling or distributing project documentation. No rights may be exercised in the case of derivative works.

- **Attribution**

Intellectual property notices of developments must be respected.

- **No Discrimination Against Persons or Groups**
- **No Discrimination Against Fields of Endeavor**

The license must not impose any limitations on the use of the work in any sector or activity.

- **Distribution of License**

The license is distributed without the need to seek additional permissions.

- **License Must Not Be Specific to a Product**

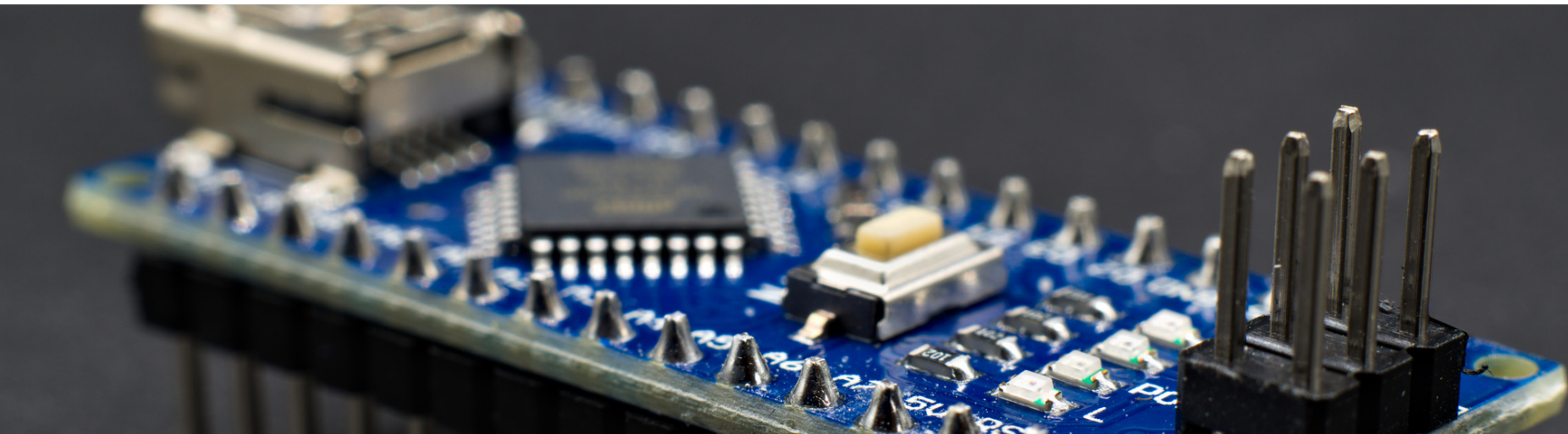
This license is extended by derivative rights.

- **License Must Not Restrict Other Hardware or Software**

There are no objections to the nature of what may be incorporated or added to this technology from the outside.

- **License Must be Technology-Neutral**

This technology's use shall not be dependent on any specific technology, part or component, material, or interface.



Hardware

As a huge benefit, Open Source Hardware **eliminates** the need to **start** from scratch in many situations. The ability to use circuit platforms that have already been established and are publicly available, such as Arduino, enables for the rapid execution of ideas that would otherwise take a long time to design.

Open source is a great illustration of how something that doesn't make sense at the beginning but then may turn out to be a game-changer in practice. Open Hardware represents a **growing movement**, potentially as important as Open software. Open Hardware applications can find a place in **different spaces to provide opportunities for developers** and solve problems that cannot be solved in conventional markets.

It has allowed us to face a completely new situation and the creation of other productive and **innovative possibilities**, which are gaining more and more popularity in all areas of society.

Open Source based PLCs Features

Discover the features of the different ranges of industrial PLCs based on Open Source CPUs such as Arduino, Raspberry Pi or ESP32.

Types of CPUs assembled in Industrial Shields PLCs

ARDUINO

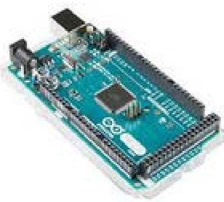
Arduino is an open source electronics creation platform based on free hardware and software, allowing anyone to use and adapt them. Thanks to that, you can find in the market several types of boards, accessories and compatible applications created by different companies or developers. All of them are different, but using the same common base, which helps the community of creators to give them different types of use.

Arduino Leonardo

Microcontroller based on the ATmega32u4. With 20 digital input / output pins (7 can be used as PWM outputs and 12 as analog inputs). Micro USB connection, power connector, an ICSP and reset button.

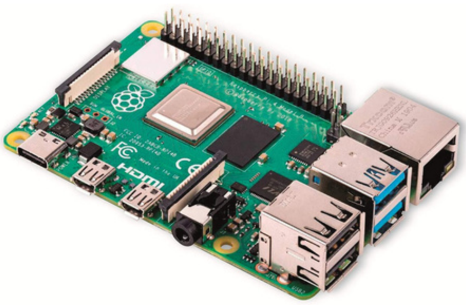
Arduino Mega

Microcontroller based on the ATmega1280. With 54 digital input / output pins (14 can be used as PWM outputs), 16 analog inputs, 4 UART, USB connection, power connector, ICSP, and reset button.



Raspberry Pi

Raspberry Pi is a low-cost, simple board computer developed in the UK by the Raspberry Pi Foundation. It is powerful enough to facilitate learning and perform basic tasks, and also allows you to program and compile programs that run on it.



ESP32

ESP32 is the name of a family of low-cost, low-power SoC chips with integrated Wi-Fi and Bluetooth dual-mode technology. It employs a Tensilica Xtensa LX6 microprocessor in its single and dual-core variants and includes antenna switches, RF balun, power amplifier, low noise receiver amplifier, filters, and power management modules.

Differences about CPUs you should know

Arduino was specifically designed so that anyone can create projects with its concept.
That is why its strenght lies in its ease of **connection with the world**, thanks to its analog and digital inputs and how easy it is to activate or deactivate with its software.

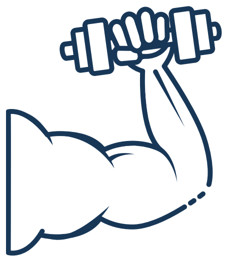
It is therefore a very versatile alternative.

However, the **Raspberry Pi** was designed as a computer itself, so it has **more computing power** than the Arduino boards.

What cannot be compared is Arduino's versatility, although it is gaining more and more in this respect thanks to the growing creation of extensions to add features.

In this sense, it is also important to talk about connectivity.
The Raspberry Pi has WiFi and Ethernet connectivity already buit into the board.

If we talk about the ESP32 board, the microcontroller is 10 times faster than the Arduino boards, and it also has a 32-bit, dual-core architecture.
The data processing speed is much faster than an ATmega board like the Arduino Mega.
As with the Raspberry Pi, the ESP32 also includes WiFi and Bluetooth.
It is also superior in the number of GPIOs and with higher resolution, 12 bits.



Inputs and Outputs. Available quantities and types

Inputs

All PLCs have analogue, digital and interrupt inputs. Those with the letter R in their description also have relay outputs.

Outputs

Inputs:

- Analogical: Min. 4, Max. 16
- Digital: Min. 9, Max. 36
- Interruption: Min 2, Max 6

Outputs:

- Analogical: Min 2, Max 8
- Digital: Min 8, Max 22
- Relay: Min 8, Max 23

What you need to know about inputs and ouputs

PLCs can be adapted to the needs of inputs and outputs by selecting one or the other equipment and also thanks to the flexibility in being able to exchange the use between inputs and outputs.

The PLCs have a **switch** similar to the one in the image, which allows the adaptation, configuration and selection of uses for the inputs and outputs. Each equipment has its own particular configuration, which is beyond the scope of this guide.

More about inputs and outputs

The computers have USB ports, which are not properly inputs but could be confused.

It is important to always check the user's manual to avoid uses that could damage the equipment. An example is not supplying power via USB, which should only be used for programming the equipment.

Comunications in PLCs

There are multiple types of communication available for use in Open Source Hardware based PLCs.
As mentioned above, the number of inputs and outputs may vary depending on the equipment, the number of inputs or outputs configured or the accessories available in the PLC ranges such as WiFi, GPRS, LoRa or Dali.

Types of communications available

- I2C
- SPI
- Serial TTL (UART)
- Ethernet
- RS485 Half / Full Duplex
- RS232
- Wi-Fi & BLE
- GPRS / GSM

Certificates

Industrial Shields PLCs were oriented since the the first moment to projects and solutions for the industrial world. One of the most important requirements for a product to be part of the industrial sector is that it complies with the guarantees and certifications that are demanded.

Conform to health, safety and environmental protection (CE)

EN61010-1
EN61010-2-201
EN61131-2:2007
(Clause 8: Zone A / B EMC and clause 11: LVD)
EN61000-6-4:2007 + A1 2011 (Emissions)
EN 61000-6-2:2005 (Immunity)

Medical Devices Directive (CE): **93/42/EEC**

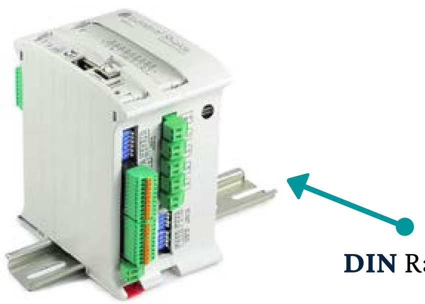
FCC Federal Code of Regulation (CFR) for Electronic Equipment: **EMC: FCC Part 15**

RoHS: **Directive 2002/95/EC | Restriction of Hazardous Substances (EEE)**

UL: STD 61010-2-201 and UL STD 61010-1

NCAGE (Commercial and Government Entity Code - Department of Defense):
NCAGE 99SGB | Commercial and Government Entity Code | Boot&Work Corp SL

Other relevant information



DIN Rail mounting

Maximum Consumption: **1.5A**
Power Supply Voltage (Vdc): **12-24**
Power consumption (VAC max.): **30**

Operating temperature: **0C-60C || 32F-140F**
Operating relative humidity % (no condensation): **10%-90%**
Moisture Sensitivity Level (MSL): **MSL 1 - Unlimited**

RoHS ¿RoHS Compliant by Exemption? **No**

Safety

Internal power supply
Galvanic isolation
Diode protected outputs
Protection against polarity reversal
Inputs protected against surges (resistance)
EMC (acording to IPC-2221)
Diferent ground planes (Single common points)
Coupling capasitors

Lead Free

Does not contain lithium

ECCN Number: **EAR99H**
STATIC Sensitive: **No**

Country of origin: **Spain**

Packaging measures (box): **13cm x 14cm x 8cm**

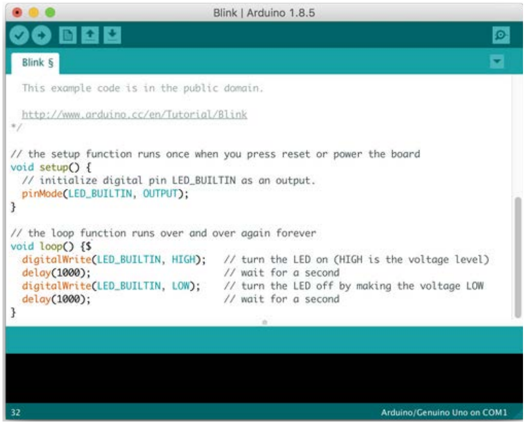
First steps – Arduino IDE and the Industrial Shields boards

¿What is Arduino IDE?

It is the Arduino Integrated Development Environment (IDE).

It is a multi platform application (for Windows, macOs, Linux) that is used to write and load programs on boards compatible with Arduino.

It can also be used with other boards, or equipment such as Industrial Shields ones, but for this it is necessary to install the boards.



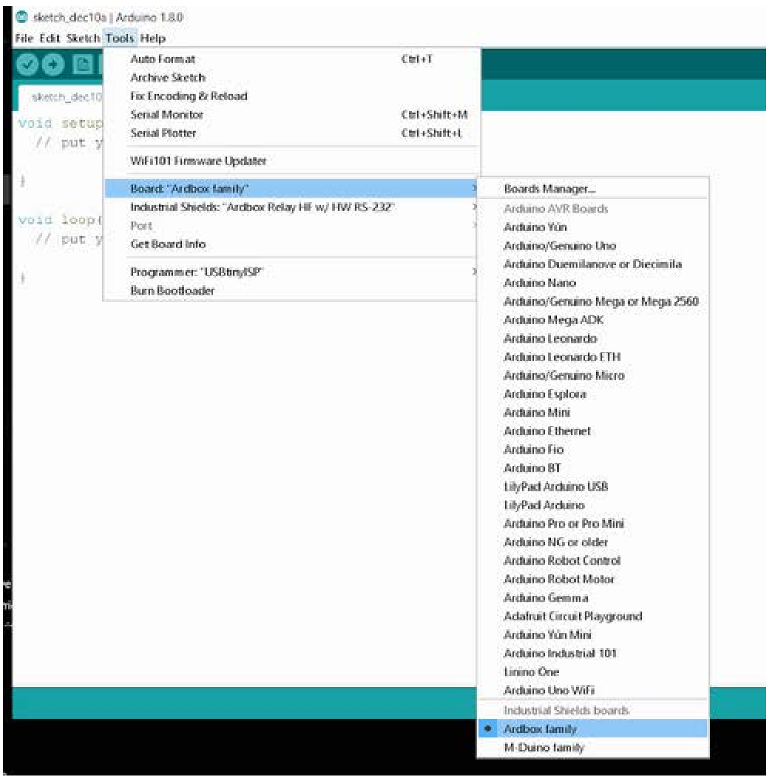
Benefits of installing the Industrial Shields Boards

The use of Industrial Shields boards simplifies the programming of the PLCs since they allow:

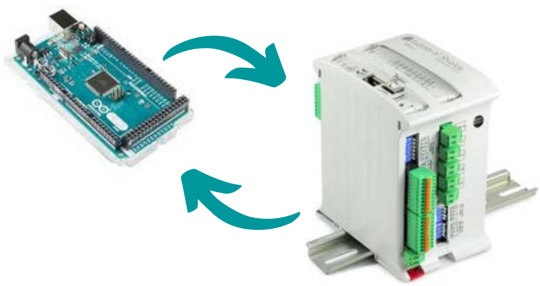
- Automatic definition / association of variables / pinmode of a pin
- Industrial Shields automatic boards (PLC features)

This is a "library collection" that is included in the Arduino IDE software, when they are selected and the Arduino board is not selected

Automatic definition / variable association / pinmode of a pin helps in pinout management.
If the sketch is not done with the boards, it cannot be expanded for future versions and for other models / teams.



Our pins (QX.X / IX.X / AX.X / RX.X) are referenced to a real Arduino pin.



Depending on the model or the equipment, these pins may be different.

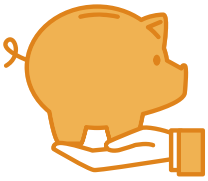
Usage examples

Once Industrial Shields boards are installed in Arduino IDE, we find different usage examples for Arduino based controller.

In Arduino IDE they can be found at:
> "File"
> "Examples"
> "MDuino Family Examples"

Benefits in using Arduino, Raspberry Pi or ESP32 controllers

Direct Impact on Costs



Different platforms can be used to program the Arduino-based equipment, the vast majority at no cost.

No license fees!

ARDUINO

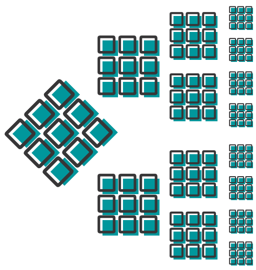
Arduino IDE, the original Arduino and the main one on the market to program Arduino boards, and therefore Industrial Shields PLCs, is free to download.

https://www.arduino.cc/en/main/software

Quantity and quality of inputs and outputs

The range of industrial PLCs based on Arduino, Raspberry Pi or ESP32, complete a range of multiple features in terms of types and quantities of inputs and outputs. There are countless applications in which to use these controllers, be it for **monitoring, control or automation solutions.**

In addition, the possibility of installation in master-slave mode must be taken into account, which greatly increases the number of available inputs and outputs.



Standard industrial communications, and more

In industrial environments, standard communications are required to facilitate the connection between all kinds of solutions, hardware or software, in the fastest, cheapest, safest and most reliable way. Industrial Shields PLCs have these requirements, although there may be manufacturers or sectors with specific solutions.

I2C	Serial TTL (UART)	Wi-Fi & BLE	RS485 Half / Full Duplex
SPI	Ethernet	GPRS / GSM	RS232

...and more

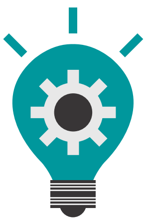
Thanks to our flexibility we have added to our range of products, specific solutions that our clients have demanded, such as:

Long Range (LoRa), An ideal technology for connections over long distances and for IoT networks where sensors that do not have mains electricity are required.

Digital Addressable Lighting Interface

DALI, It is a protocol created to control lighting systems (Digital Addressable Lighting Interface = Interface Digital de Iluminación Direccionable).

Conclusion



The benefits of the different ranges of PLC, with the particularities of each CPU, the number of inputs and outputs, or specific accessories such as GPRS, WiFi, LoRa or DALI, ensure a range of possibilities. With rare exceptions where the specifications of the solution are going to be very exclusive, Industrial Shields PLCs are a great solution for industrial applications in all sectors, be it for automation, monitoring or control.

Industrial Shields

Industrial Communications

Check the industrial communications available in the PLCs based on Open Source CPUs such as Arduino, Raspberry Pi or ESP32.

RS-232

RS-232 (Recommended Standard 232) is a standard for data transmission by serial communication. It formally defines signals connecting between a DTE (Data Terminal Equipment) such as a computer terminal, and a DCE (Data Circuit-Terminating Equipment or Data Communication Equipment), such as a modem or other industrial equipment with this port available.
The standard defines the electrical characteristics and timing of signals, the meaning of signals, and the physical size and pinout of connectors. The current version of the standard is TIA-232-F Interface Between a DTE and a DCE Employing Serial Binary Data Interchange.
The RS-232 standard had been commonly used in computer serial ports and is still widely used in industrial communication devices.

Industrial Shields PLCs include the integrated circuit MAX232

MAX232 converts signals from to TIA-232 (RS-232) serial port to signals suitable for use in TTL-compatible digital logic circuits.

The MAX232 is a dual transmitter/dual receiver used to convert the RX, TX, CTS, RTS signals.

RS-485

RS-485, also known as TIA/EIA-485, is a standard defining the electrical characteristics of drivers and receivers for use in serial communications systems. Electrical signalling is balanced, and multipoint systems are supported.

The standard is published jointly by the Telecommunications Industry Association and Electronic Industries Alliance (TIA/EIA).

Digital communications networks implementing the standard can be used effectively over long distances and in environments with electrical noise.

Multiple receivers may be connected to this network on a multidrop linear bus.

These characteristics make RS-485 useful in industrial control systems and similar applications.

RS-485

Industrial Shields PLCs include the integrated circuit MAX485

It is a low-power and slew-rate-limited transceiver used for RS-485 communication. It works at a single +5V power supply and the rated current is 300 µA.

Adopting half-duplex communication to implement the function of converting TTL level into RS-485 level, it can achieve a maximum transmission rate of 2.5Mbps.

2.5Mbps

MAX485 transceiver draws supply current of between 120µA and 500µA under the unloaded or fully loaded conditions when the driver is disabled.

ETHERNET

Ethernet is the most common technology working wth the Local Area Networks (LANs) and Wide Area Networks (WANs). The Ethernet communication uses the LAN protocol which is technically known as the IEEE 802.3 protocol.

IEEE 802.3 protocol has evolved and improved over time to transfer data at the speed of one gigabit per second.

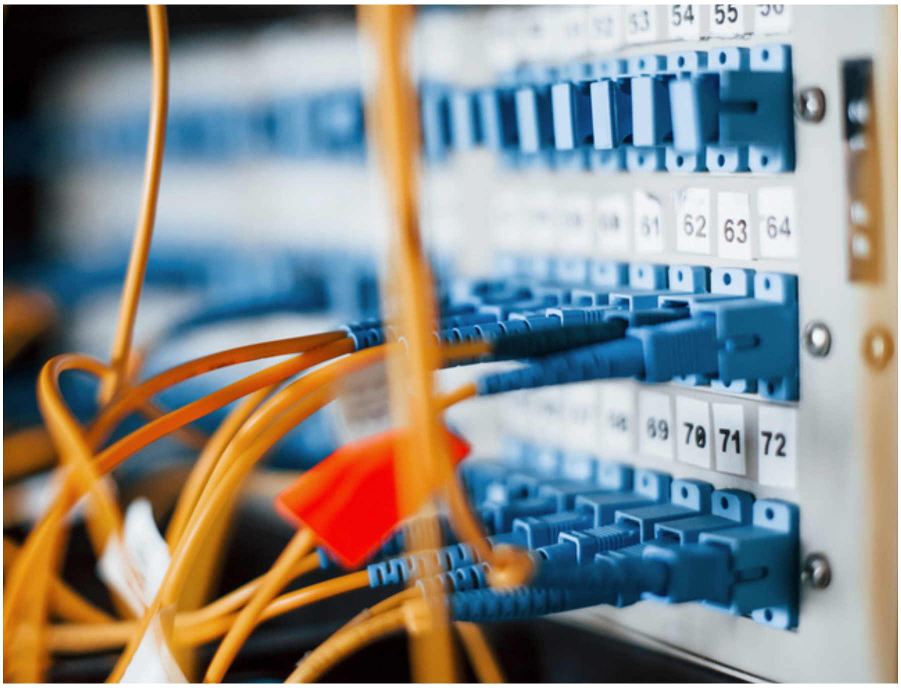
Industrial Shields PLCs incorporate the W5500 IC integrated circuit.

The W5500 is a hardwired TCP/IP embedded Ethernet controller that provides an easier Internet connection to the embedded systems. This chip alows users to have Internet connectivity in their applications by using the single chip in which TCP/IP stack, 10/100 Ethernet MAC and PHY are embedded. The W5500 chip incorporates the 32Kb of internal memory buffer for processing Ethernet packet.

With this chip users can implement the Ethernet application by using Socket Programming.

The SPI bus (Serial Peripheral Interface) is provided to facilitate the data transfer with the external microcontroller.

Ethernet uses different protocols to communicate. Some of them are HTTP, HTTPS, MQTT and Modbus protocols.



Wi-Fi

Wi-Fi is simply a trademarked phrase meaning IEEE 802.11x. Wi-Fi works off of the same principle as other wireless devices. It uses radio frequencies to send signals between devices.
To receive the information found on these waves, your radio receiver needs to be set to receive waves of a certain frequency.

In the case of WiFi, this frequency happens to be 2.4GHz and 5GHz.

In an industrial PLC controller Arduino, Wi-Fi uses multiple parts of the IEEE 802 protocol family and is designed to interwork seamlessly with its wired sibling Ethernet.

Compatible devices can be networked via wireless access points to each other, as well as to wired devices and the Internet.

The different versions of Wi-Fi are specified by various IEEE 802.11 protocol standards, with the different radio technologies determining radio bands, and the maximum ranges and speeds that may be achieved.

GPRS / GSM

GPRS (General Packet Radio Services) is a packet-based wireless communication service that promises data rates from 56 up to 114Kbps and a continuous Internet connection for mobile phone and computer users. It works on the mobile network with the help of IP (Internet Protocol) transmissions.
GPRS is the mobile data system behind 2G and some 3G.

56 to 114 Kbps

GPRS is based on Global System for Mobile (GSM) communication and complements existing services such as circuit-switched cellular phone connections and the Short Message Service (SMS).

The Industrial Arduino based PLCs with GPRS are ideal for:

remote monitoring | data logging and remote access | diagnostics and control

by using short text messages (SMS).

You can adjust the messages to be sent from device with **static** (text) or **dynamic** (text and values) content.

SMS SMS SMS SMS SMS SMS

Bluetooth Low Energy

BLE also know as **Bluetooth Low Energy** is based on the TSMC ultra-low-power 40 nm technology, as well as Wi-Fi microchip. The main specs of this kind of Bluetooth are that it is based on the 4.2 BR/EDR dual mode controller version, has +12 dBm transmitting power and a NZIF receiver with a sensitivity of -97 dBm.

The BLE is a subgroup of the 4.0 version with a whole new protocol stack to quickly devolop new links. Its objective is to cover applications with low power demand. You can consult more infromation about these specific versions in the official webpage.

+12dBm
Transmit

-97dBm
Receive

Some of the Industrial Shields PLCs can use this communication protocol. The Raspberry PLC or the M-Duino and Ardbox range with WiFi and BLE.

In the Arduino based range this feature is using the ESP32 board. The Raspberry Pi PLC range can use this feature directly with the Raspberry Pi board.

Arduino & ESP32 PLC

Both Arduino and ESP32 work with the same module. The integrated WiFi module consists of a single 2.4 GHz Wi-Fi and Bluetooth combo chip designed with the TSMC ultra-low-power 40 nm technology.

It is designed to achieve the best power and RF performance, showing robustness, versatility and reliability in a wide variety of applications and power scenarios.

Some applications are Generic Low-power IoT Sensor Hub, Generic Low-power IoT Data Loggers and Mesh Network. It is designed for Internet-of-Things (IoT) applications.

General Specifications:
- 802.11 b/g/n
- 802.11 n (2.4 GHz), up to 150 Mbps

Up to 150 Mbps

Raspberry PLC

General Specifications:
- 802.11.b/g/n/ac
- 802.11 n (2.4 GHz / 5GHz)
- 5.0 BLE

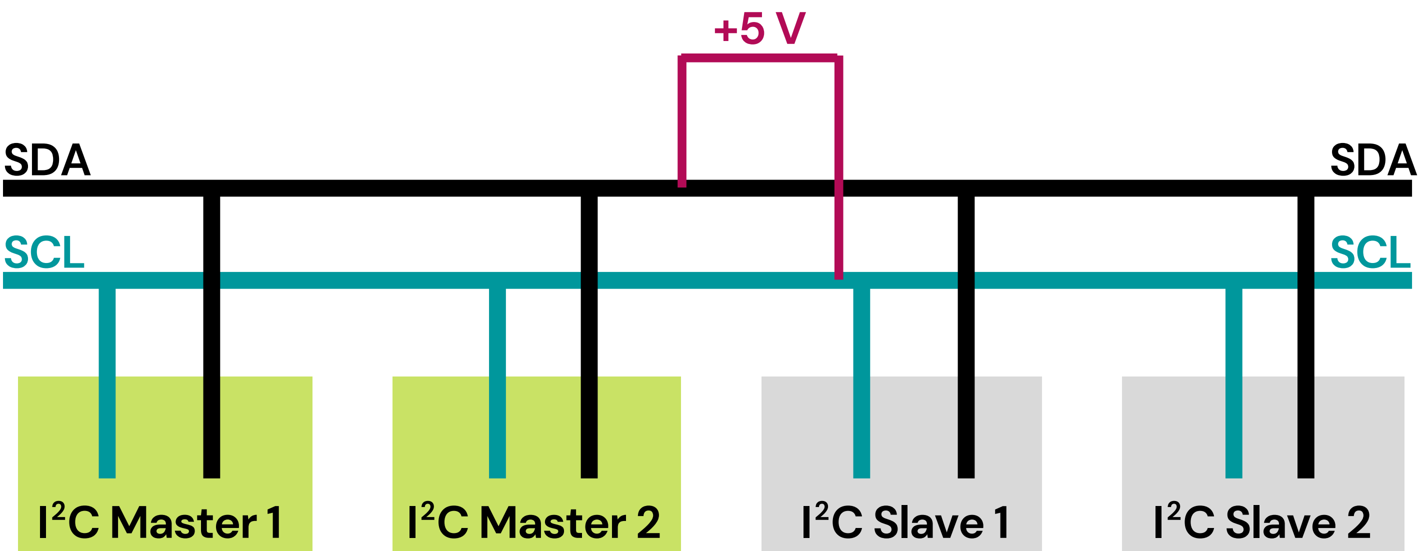
I2C

I2C (Inter-Integrated Circuit), pronounced I-squared-C, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus. It is widely used to connect lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communications.

The I2C has gradually been adopted by other manufacturers until becoming a market standard. The I2C bus requires only two cables for operation, one for the clock signal (CLK) and the other for data transmission (SDA), which is an advantage over the SPI bus. By cons, its operation is a little more complex, as well as the electronics needed to implement it.

Data is transferred bit by bit along a single cable (the SDA line). With I2C multiple slaves can be connected to a single master, and multiple masters can be controlled by one or more slaves. This is really useful when you want to have more than one microcontroller recording the data on a single memory card or displaying text on a single LCD screen. It only uses two wires to transmit data between devices:

SDA (Serial Data) – The line for the master and slave to send and receive data.
SCL (Serial Clock) – The line that carries the clock signal.



SPI

SPI (Serial Peripheral Interface) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors and SD cards. It uses separate clock and data lines, along with a select line to choose the device you want to talk to.

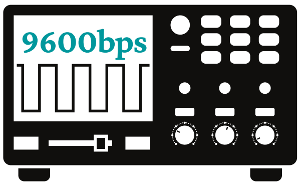
The SPI bus, which operates at full duplex (i.e. the signals carrying data can go in both directions simultaneously), is a synchronous type data link setup with a Master-Slave interface and can support up to 10Mbps of speed.

Full Duplex

Up to 10 Mbps

Both single-master and multi-master protocols can be used as SPI.

SERIAL TTL



Serial TTL (UART): UARTs (Universal Asynchronous Receivers/Transmitters) transmit one bit at a time at a specified data rate (9600bps usually). This method of serial communication is sometimes called TTL serial (transistor-transistor logic).

Serial communication at TTL level will always remain between the limits of 0V and Vcc, which is often 5V or 3.3V. It is based in two unidirectional channels; Tx to transmit and Rx to receive.

LoRa

LoRa (Long Range) is a low-power wide-area network (LPWAN) protocol developed by Semtech© that uses its own frequency modulation to communicate. This technology is based on a spread spectrum modulation technique derived from Chirp Spread Spectrum (CSS) which is historically used in military and space operations.

LoRa (Long Range modulation) is a type of wireless technology. It uses uses a radio frequency network modulation such as AM, FM or PSK, but this was created by an important radio chip manufacturer called Semtech©, but now managed by LoRa Alliance©. This modulation is named CSS (Chirp Spread Spectrum) and has been used in militar operations for many years. The benefits of this kind of communications are that it can reach long distance (cover wide areas, usually kilometers) and has a good resistance to interferences.

It can reach long distances from 10 to 20 Km.

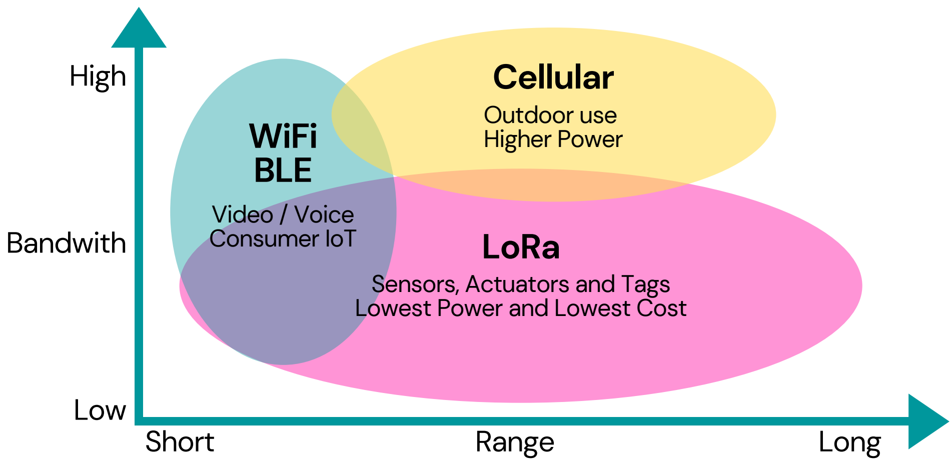
LoRa is specially useful for long distance communications and for IoT networks such as Smart Cities or agricultural holdings. LoRa communication has a high tolerance to the interferences and a huge sensibility to recive data.

Depending on the zone, it will work on a different frequency
868 MHz on Europa
915 MHz on América
433 MHz on Asia.

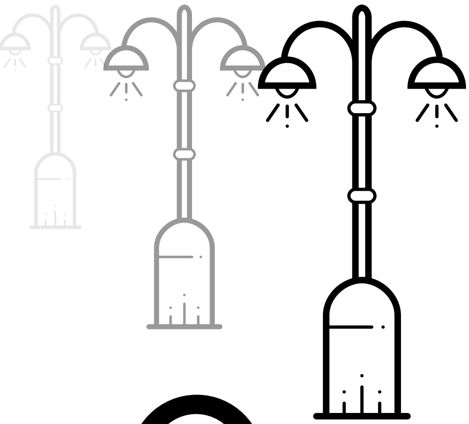
It is an ideal option when we need long range communication and IoT networks composed of sensors which are not connected to the electrical network because of its locations or its main usage. For exemple, this communication is widely used in Smart Cities or low coverage areas such as farming applications or networks of sensors/actuators which can take profit of their main characteristics.

Industrial Shields PLCs incorporate the RFM95C integrated circuit.

In the **M-Duino**, RFM95C controller communicates with the Mega board via an SPI bus (Reset is Arduino Mega pin 2, SS is Arduino Mega pin 12, interruption is Arduino Mega pin 13).

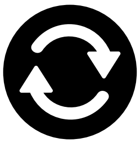


DALI



The Digital Addressable Lighting Interface (DALI) is a communication protocol designed to control light and regulate lighting systems.

It is based on an electronic system that allows you to talk bidirectionally with the connected devices, sending or receiving information. It will be very useful for controlling large lighting systems and regulating their use together with light, motion or timer sensors, allowing the automated control needed for large buildings and companies interested in industrial automation.



Bidirectional

1 DALI bus 64 devices

The DALI protocol allows a total of 64 devices to be controlled, interconnected by a DALI Bus.

The main advantages over its competitors would be easy planning and installation together with maximum flexibility when making modifications. Slave devices can be added later and, moreover, they do not need to be assigned to an initial configuration at the time of installation, as everything will be controlled digitally. As no special wiring or accessories are required, it makes the DALI protocol quick and easy to implemented.

Thanks to the possibility of regulating the lights automatically, it will be possible to meet time requirements according to energy peaks, sunshine hours or energy rates. The DALI protocol allows the professional control of various environments and configurations such as System Automation or Regulation of lights intensities.

RTC

A real-time clock (RTC) is an electronic device wich measures the passage of time and that is usually included in an integrated circuit . RTCs are present in almost any electronic device which needs to keep accurate time. RTCs are devices widely used in electronics. They are also very common in embedded systems and, in general, in a multitude of devices that require time registration.

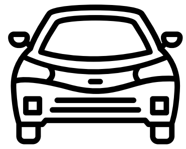
The RTC devices have an integrated crystal oscillator working at a frequency of 32.7 KHz used for take control of the time. One advantage of the real-time clock is that it uses our ways of measuring time, working with the sexagesyimal system.

- **Industrial Shields PLC uses the DS3231 chip for implementing the real-time clock**

  This chip has the advantage of incorporing a temperature measurement and compensation guaranteeing an accuracy of at least 2ppm.

CANBus

CAN (Controller Area Network) Bus is, as its name suggests, an automotive bus that allows to the microcontrollers and other devices to communicate to each other without having a host computer. The CAN Bus protocol was developed by Bosch© especially for automotive applications but nowadays is widely used in other areas. It connects individual sensors and systems as an alternative to conventional multi-wire looms.



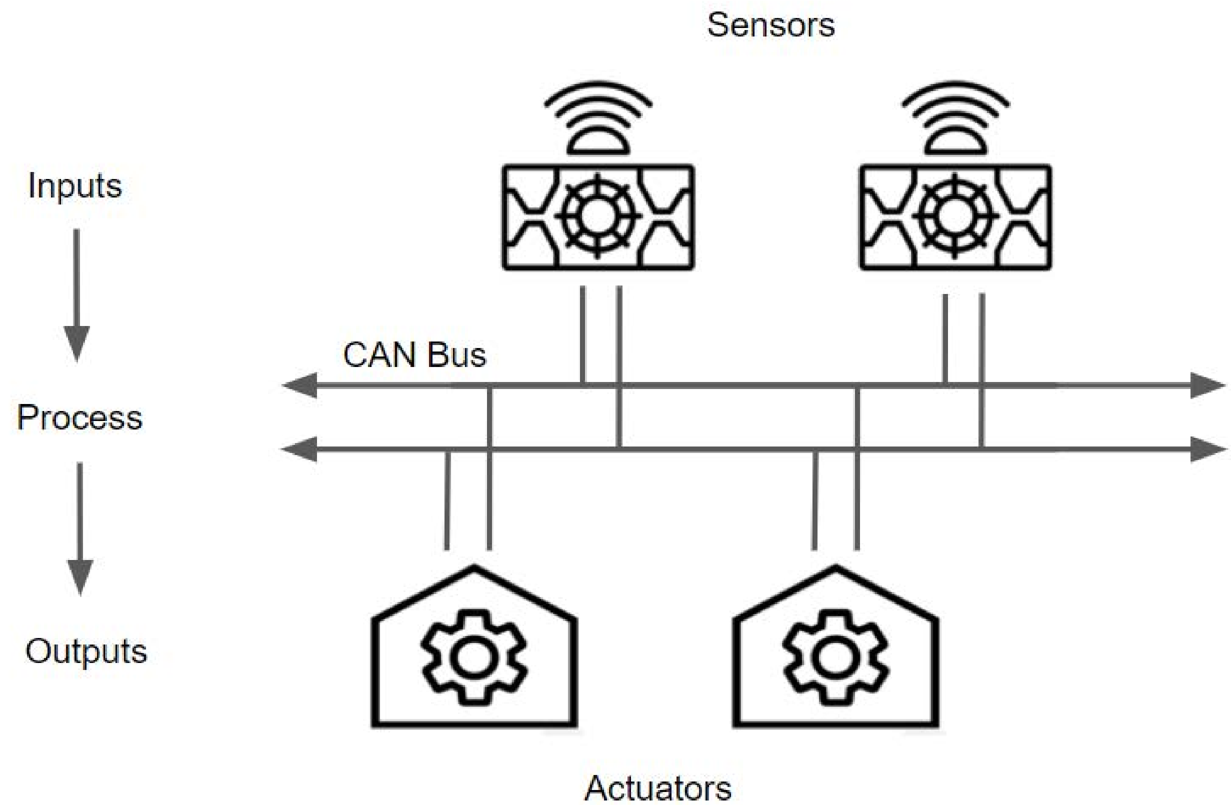
The CAN bus is implemented in Industrial Shields Raspberry PLCs using the MCP2561-E chip to make the conversion between de CAN and Serial, and the MCP2515-I to connect the serial to the SPI.

This communication follows a specific protocol; CAN Bus uses only two wires for the communication. One is called CAN High and the other is named CAN Low. The CAN controller is connected to all network components through these two wires. Each network node has an individual identifier. All the devices, also called ECU's (Electronic Control Units) are distributed in parallel so that all the nodes receive all the information on the channel each time that is sent. The node only responds when it detects its own identificator. Because of this, the individual nodes can be deleted from the network and the others will not be affected.

The working method is that, when the CAN Bus is in idle mode, both lines transport 2.5V. When data bits are transmitted, the CAN High carries 3.75V and the CAN Low 1.25V, creating a 2.5 differencial between two lines. Each line is referenced to the other, not to the Ground, so CAN Bus cannot be affected by inductive peaks, electrical fields or other noise.

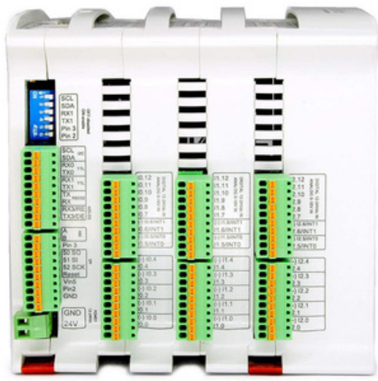
It is a reliable communication method.

The CAN can be supplied through the CAN Bus or an external power suppy. Another important factor is that all the modules can transmit and receive information from the bus and, as we have said, the data sent by one device will be received by all the others. It is important that the bandwidth of the bus is assigned first to the critical systems. Therefore, the nodes will be organised by priority.



Inputs & Outputs

Basics about digital inputs of an industrial PLC



Introduction

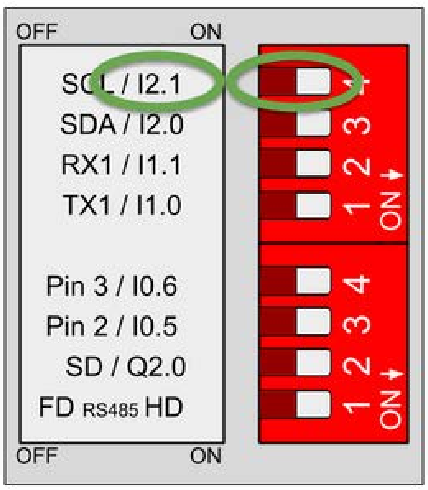
Thanks to this reading you will understand how to connect and configure the PLCs to be able to read the digital inputs correctly.

Configuring the switches

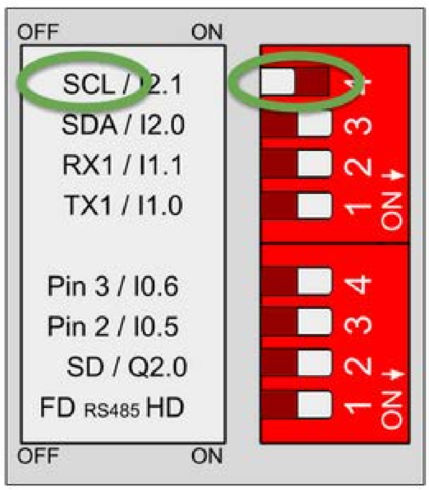
Almost all the digital inputs are always connected to the internal **Arduino**, but in a few cases, the user can choose a special peripheral configuration or a GPIO normal working. In these cases, the user can choose between two options through the switches.

Each switch can select only one configuration. For example, in this case, we are watching the **GPIOs** configuration of an M-Duino 57R+. If we put the switch to the right in the upper one, the input I2.1 will be activated and we will be able to work with this input as digital.

If the switch is in the left position, we will activate the **SCL** line which will be used for **I2C** communication. Keep in mind each switch has two different configurations: you must select the right or the left option.



I2.1 input enabled - SCL disabled



I2.1 input disabled - SCL enabled

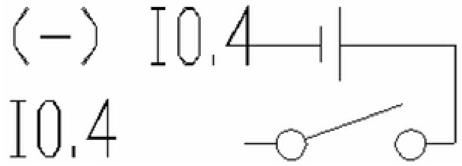
Basics about digital inputs of an industrial PLC

Input types

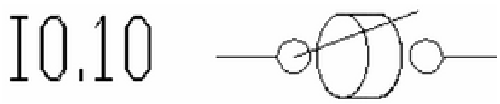
There are three different types of inputs in the **Industrial Shields** PLCs:

- 5V - 24V input
- 5V - 24V optoisolated input
- 5V input

Each one has a particular draw in the case of the **PLC**. Remember only the Pin 2 and Pin 3 are 5V compatibles:



5V - 24V optoisolated input



5V - 24V input

5V input

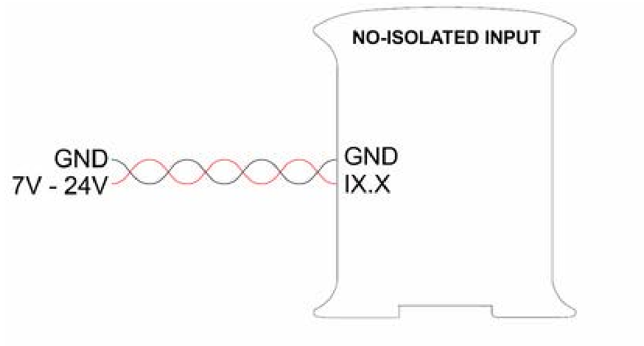
Hardware

Not all the inputs must be connected in the same way. While the non-isolated inputs must be referenced to the same ground as the **PLC**, the isolated inputs can be connected to the input grounds, allowing to isolate systems from the **PLC**.

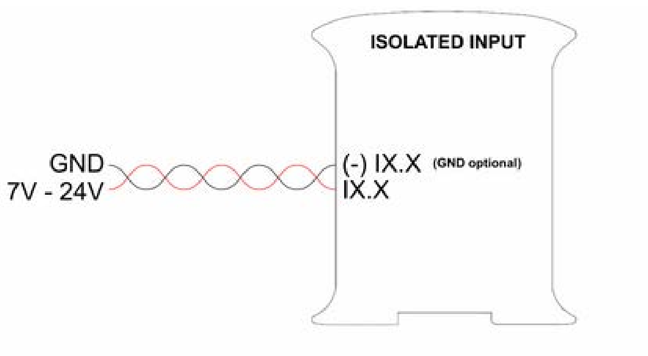
Anyway, the optoisolated input can be connected to the **PLC** ground as well.

The following images show how to connect the different inputs to the **PLC** for industrial automation:

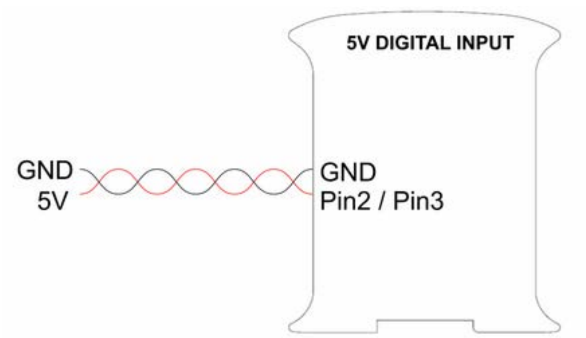
5V - 24V optoisolated input



5V - 24V input



5V input



Basics about digital inputs of an industrial PLC

Software

In order to program the digitals **GPIO**, we must keep in mind we can read the values with the following command:

```
digitalRead(GPIO);
```

This function returns "0" or "1" depending on the actual value of the input. **GPIO** is the name of the input. Imagine we want to know the state of the "I0.4" input, then, we must write this line:

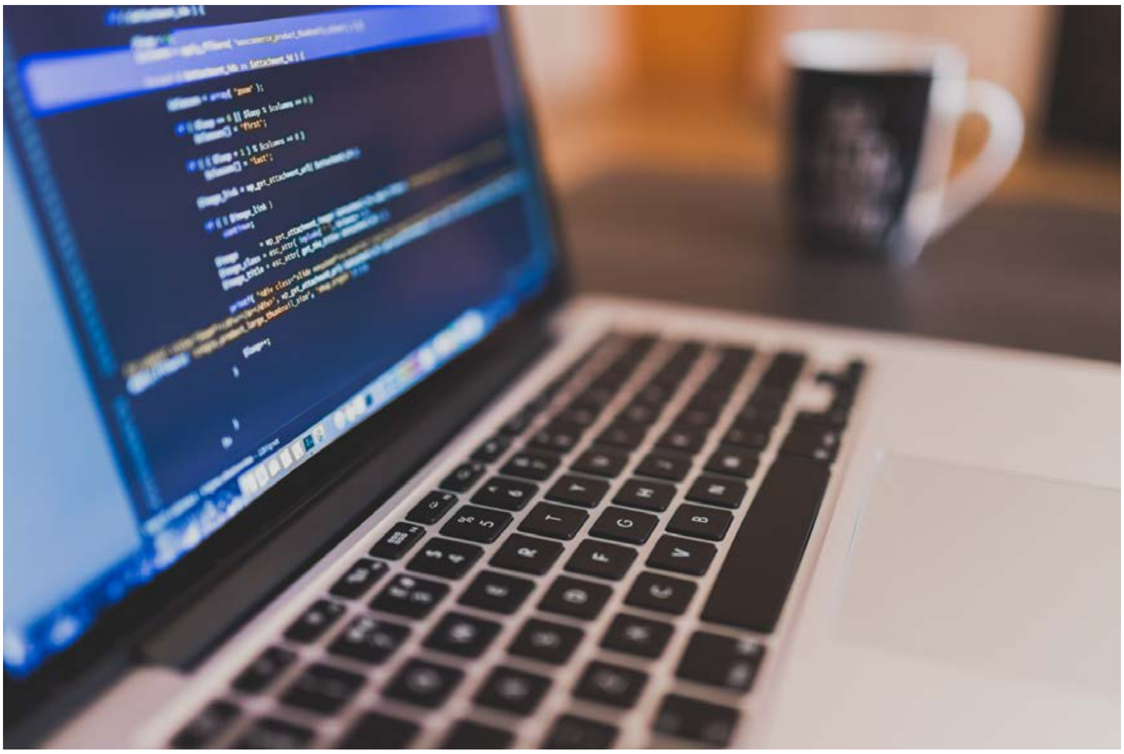
```
digitalRead(I0_4);
```

The inputs "2" and "3" does not have a special name, and to read them we must write:

```
digitalRead(2);
digitalRead(3);
```

We must keep in mind we do not need to configure the digital inputs of the **PLC** as digital ones, except with the 5V compatible inputs. It means we must configure the inputs in the setup before read them: These statements must be defined within the Setup function:

```
pinMode(2,INPUT);
pinMode(3,INPUT);
```



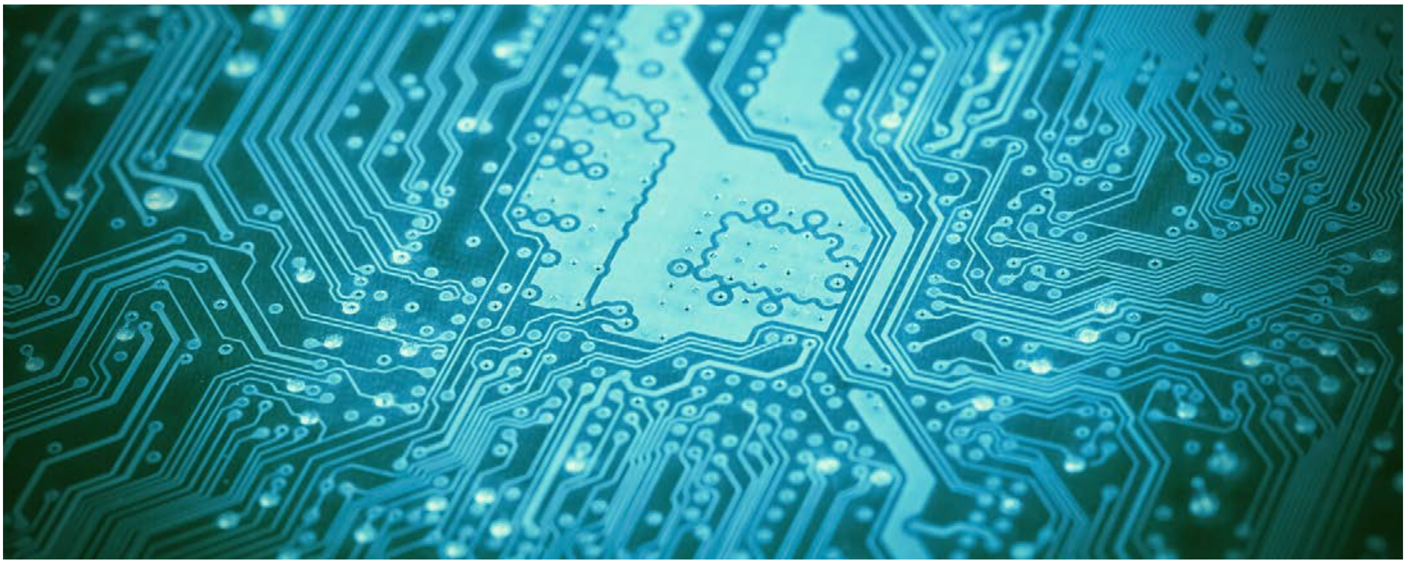
Basics about digital inputs of an industrial PLC

Software

Examples

You can see a read digital **GPIO** example in the following paragraph:

```
// Digital read example
// This example reads the I0_10, I0_2 and Pin 2 inputs, and shows via
serial if they are active

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);

    // Configure Pin 2 as a digital input
    pinMode(2, INPUT);
}

// Loop function
void loop()
{
    // Check Pin 2
    if (digitalRead(2))
        Serial.println("Pin 2 active");

    // Check I0_10
    if(digitalRead(I0_10))
        Serial.println("I0_10 active");

    // Check I0_2
    if(digitalRead(I0_2))
        Serial.println("I0_2 active");
```



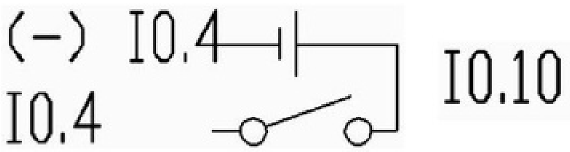
Digital inputs of a Raspberry PLC

Input types

There are two different types of inputs in the **Raspberry Pi industrial PLC** devices:

- 5 Vdc - 24 Vdc input
- 5 Vdc - 24 Vdc optoisolated input

Each one has a particular draw in the case of the PLC:



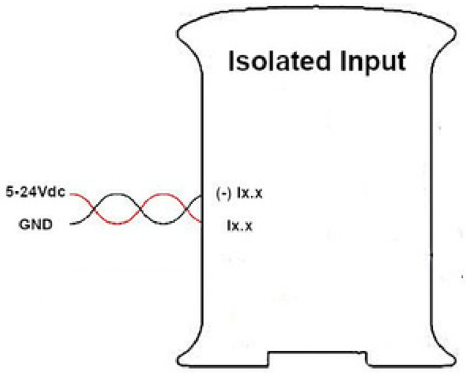
5 - 24 Vdc Optoisolated Input

5 - 24 Vdc Input

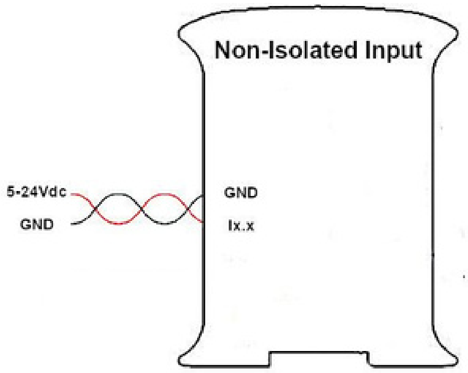
Hardware

Not all the inputs must be connected in the same way. While the non-isolated inputs must be referenced to the same ground as the **PLC**, the isolated inputs can be connected to the input grounds, allowing to isolate systems from the **PLC**. Anyway, the optoisolated input can be connected to the **PLC** ground as well.

The following images show how to connect the different inputs to the **PLC** for industrial automation:



5 - 24 Vdc Optoisolated Input



5 - 24 Vdc Input

Software

How to work with Bash Scripts
Raspberry Pi PLC has default bash scripts to work with the inputs. All the inputs and outputs scripts must be executed from the correct path.

It depends on the shield type of the I/O executed. In function of the shield of the I/O that you need to activate, you must execute the scripts from a specific path:

- Analog/Digital Shields

```
> cd /home/pi/test/analog
```

- Relay Shield

```
> cd /home/pi/test/relay
```

Digital inputs of a Raspberry PLC

The get-digital-input script will show the value of the selected input pin. It will only be provided the pin with which we are going to work. In order to call the function, we will do the following:

```
> ./get-digital-input <input>
```

Example for the I0.0 input returning a True value:

```
> ./get-digital-input I0.0
```

How to work with Python

The bash commands are the basis to work easily with the **Raspberry Pi PLC**. In order to work with python files, if you want to interact with the IOs of the PLC, you will have to call these scripts.

To edit the files you will be working with the Nano editor included by default and Python3.

```
nano digital_inputs.py
```

Python allows you to execute a shell command that is stored in a string using the subprocess library. In order to work with it, you will have to import it at the start of the file.

```
import subprocess
```

In this example, you will be reading the input given of the pin I0.0 **of the Raspberry Pi PLC**. In order to do it, you will implement a loop that will be constantly reading the input value. If it detects voltage, it will print a True value.

```
import subprocess
import time
print("Start")
while True:
        try:
                x = subprocess.run(["./get-digital-
          input","I0.0"], stdout=subprocess.PIPE, text=True)
                if '1' in x.stdout:
                        print(True)
                        time.sleep(1)
                else:
                        print(False)
                        time.sleep(1)
        except KeyboardInterrupt:
                print("\nExit")
                break
```

In order to execute the Python program, you will call it as follows:

```
> python3 analog_outputs.py
```

To exit the program, just press ^C.

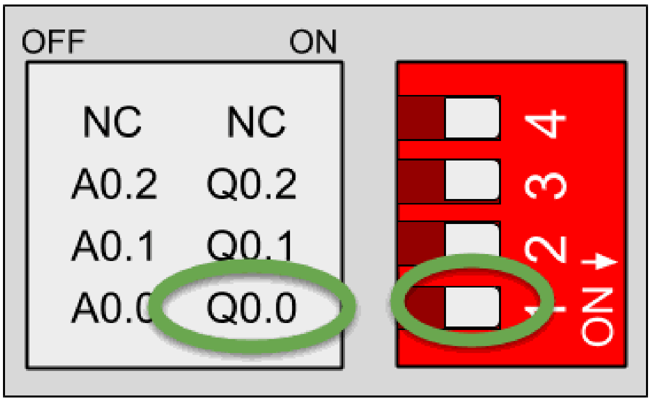
Basics about digital outputs of an industrial PLC

Reading this section, you will be able to understand how to connect and configure the digital outputs of your industrial Arduino PLC controller.

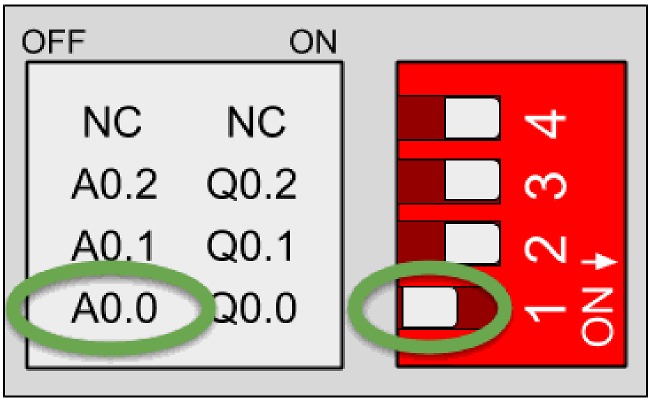
Configuring the switches

Most of the digital outputs are always connected to the internal **Arduino**, but in few cases, the user can choose between a special peripheral configuration or a **GPIO** by changing the position of the **Dip** Switches.

Each switch can select only one configuration. For example, in this case, we are watching the **GPIOs** configuration of an **M-Duino 21+**. If we put the switch to the right position (ON) in the lower one, the output **Q0.0** will be activated and we will be able to work this as digital.
If the switch is in the left position (OFF) we will activate the output as analog. Keep in mind each switch has two different configurations: you must select the right (ON) or the left (OFF) option.



Q0.0 enabled - A0.0 disabled



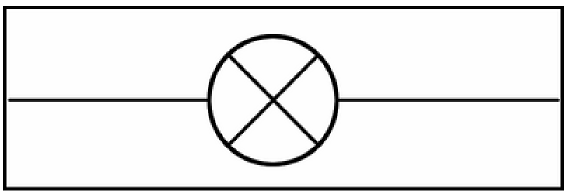
Q0.0 disabled - A0.0 enabled

Output types

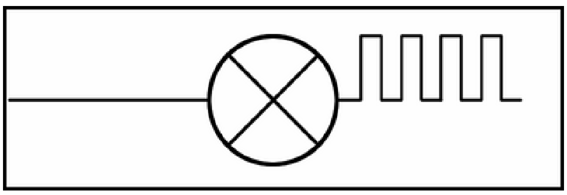
In all of the **Industrial Shields Arduino based PLCs**, digital outputs can work at:

- 5V -24V digital output

Digital outputs have a special draw in the case of the **PLC**. Keep in mind that the output that can handle PWM is the same as the other digital outputs



Digital output



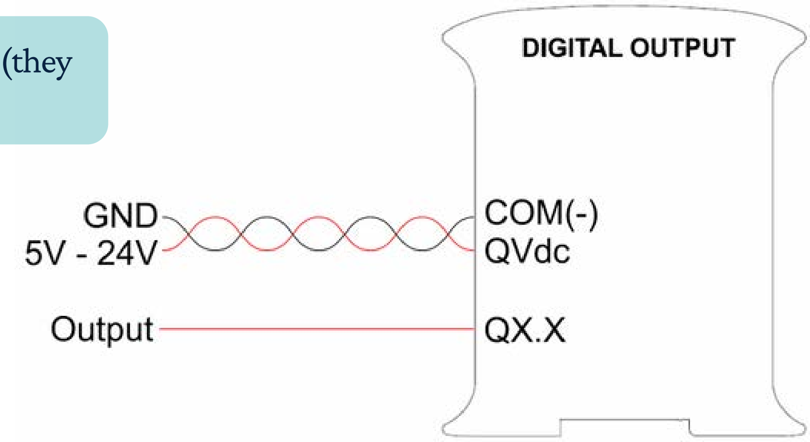
Digital output (PWM optional)

Basics about digital outputs of an industrial PLC

Hardware

All the digital outputs are optoisolated (they use the same **GNDs** as the **PLC**).

The following image shows how to connect a digital output to the **PLC**:



5Vdc - 24Vdc Digital output

Software

In order to program the digital outputs, we must keep in mind that we can write the values with the following command:

```
digitalWrite(GPIO,value);
```

This function writes a "HIGH" or "LOW" in the "GPIO" selected. Imagine we want to write a "HIGH" in the "Q0.6" output, then, we must write this line:

```
digitalWrite(Q0_6,HIGH);
```

We must know we do not need to configure the digital outputs as digital. Industrial Shields' libraries do all the work for us.
Example
You can see a digital **GPIO** written in the following paragraph:

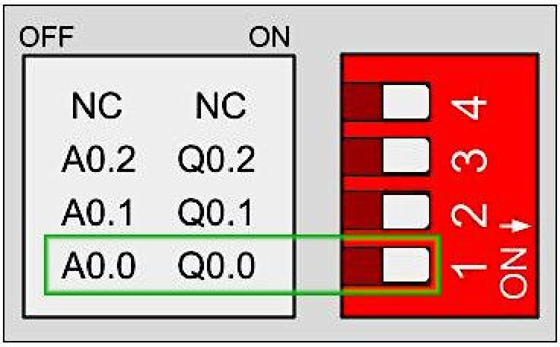
```
// Digital write example
// This example writes the Q0_0 and shows via serial the value

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);
}

// Loop function
void loop()
{
    Serial.println("1");
    digitalWrite(Q0_0, HIGH);
    Serial.println("0");
    digitalWrite(Q0_0, LOW);
}
```

Digital outputs of Raspberry PLC

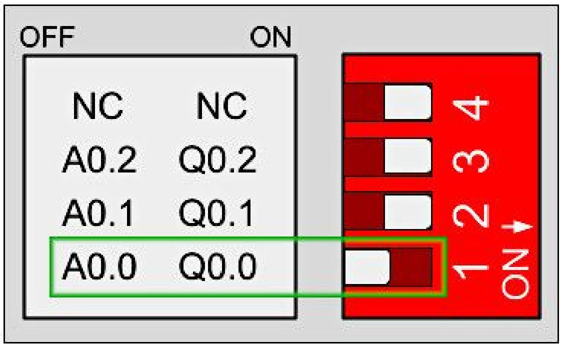
Configuring the switches

Most of the digital outputs are always connected to the internal **Raspberry Pi**, but in a few cases, the user can choose between a special peripheral configuration or a GPIO by changing the position of the Dip Switches.

Each switch can select only one configuration. For example, in this case you are watching the GPIOs configuration of an open source PLC Raspberry Pi 21+. If you put the switch to the right position (ON) in the lower one, the output Q0.0 will be activated and you will be able to work as digital. If the switch is in the left position (OFF), you will activate the output as analog. Keep in mind each switch has two different configurations: you must select the right (ON) or the left (OFF) option.



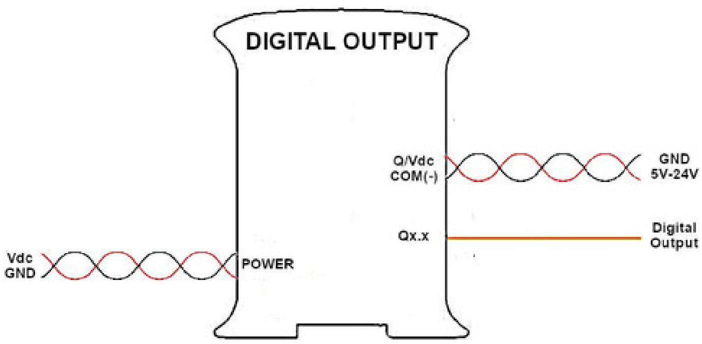
A0.0 Desactivado - Q0.0 Activado



A0.0 Activado - Q0.0 Desactivado

Hardware

Todas las salidas digitales están optoaisladas (utilizan las mismas GND que el PLC). La imagen siguiente muestra cómo conectar una salida digital a su PLC:



Salida de 5-24 Vdc

Digital outputs of Raspberry PLC

Software

How to work with Bash Scripts

Raspberry Pi PLC has default bash scripts for working with the inputs. All the inputs and outputs scripts must be executed from the correct path. It depends on the shield type of the I/O executed. In function of the shield of the I/O that you need to activate, you must execute the scripts from a specific path:

- Analog/Digital Shields

```
> cd /home/pi/test/analog
```

- Relay Shield

```
> cd /home/pi/test/relay
```

The set function will initialize the pin. You will provide the pin with which you are going to work and the value that will be set. For the digital option, a logical 1 will turn on the pin while a 0 will stop it.
By default, if not value option is provided, it will be initialized as a 1 for the Digital outputs. If any other options are chosen, an error code will warn you. In order to call the function, you will do the following:

```
> ./set-digital-output <output> <value>
```

There are some pins that both can work as digital or analog. In this case, if you have used these pins before in either digital or analogic and you want to switch their mode, you must call the set function, providing a stop to the value option; otherwise, there will be a system error. If a reboot is done, it is not necessary to do it.

The pins which can operate with both Analog/Digital configurations are:

Q0.5	Q1.5	Q2.5
Q0.6	Q1.6	Q2.6
Q0.7	Q1.7	Q2.7

Ejemplo:

```
> ./set-digital-output Q0.5 1
> ./set-digital-output Q0.5 0
> ./set-digital-output Q0.5 stop
> ./set-analog-output A0.5 50
```

Digital outputs of Raspberry PLC

How to work with Python

The bash commands are the base for easily working with the **Raspberry Pi based PLC**. In order to work with python files, if you want to interact with the IOs of the PLC, you will have to call these scripts.

To edit the files, you will be working with the Nano editor included by default and Python3.

```
nano digital_inputs.py
```

Python allows you to execute a shell command that is stored in a string using the os.system() function. In order to work with it, you will need to import its library at the beginning of the file. In addition, you will need to include the time library to summon a 2 seconds delay.

```
import os
import time
```

In this example program, you will be blinking some of the industrial **Raspberry Pi PLC** digital LEDs. In order to do it, you will implement a loop that will constantly open up and shut them off in an interval of 2 seconds.

```
import os
import time
os.system("echo Start")
while True:
          try:
                 os.system("sudo ./set-digital-output Q0.0 1")
                 os.system("sudo ./set-digital-output Q0.1 1")
                 os.system("sudo ./set-digital-output Q0.2 1")
                 os.system("sudo ./set-digital-output Q0.3 1")
                 os.system("sudo ./set-digital-output Q0.4 1")
                 os.system("sudo ./set-digital-output Q0.5 1")
                 time.sleep(2)
                 os.system("sudo ./set-digital-output Q0.0 0")
                 os.system("sudo ./set-digital-output Q0.1 0")
                 os.system("sudo ./set-digital-output Q0.2 0")
                 os.system("sudo ./set-digital-output Q0.3 0")
                 os.system("sudo ./set-digital-output Q0.4 0")
                 os.system("sudo ./set-digital-output Q0.5 0")
                 time.sleep(2)
          except KeyboardInterrupt:
                 os.system("sudo ./set-digital-output Q0.0 0")
                 os.system("sudo ./set-digital-output Q0.1 0")
                 os.system("sudo ./set-digital-output Q0.2 0")
                 os.system("sudo ./set-digital-output Q0.3 0")
                 os.system("sudo ./set-digital-output Q0.4 0")
                 os.system("sudo ./set-digital-output Q0.5 0")
                 os.system("echo End")
                 break
```

In order to execute the Python program, you will call it as the follow:

```
> python3 analog_outputs.py
```

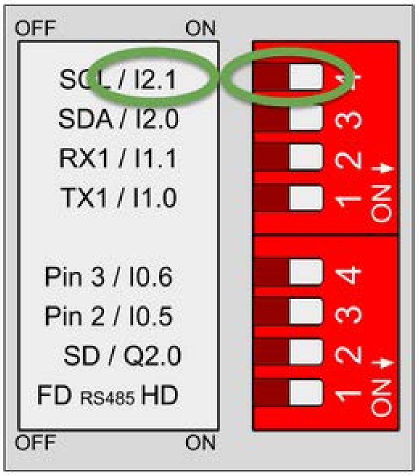
To exit the program, just press ^C.

Basics about analog inputs of an industrial PLC

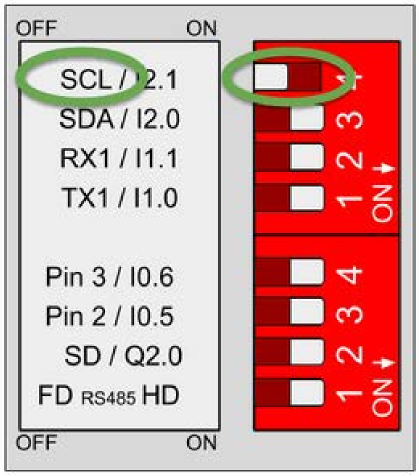
Configuring the switches

Most of the analog inputs are always connected to the internal **Arduino**, but in a few cases, the user can choose between a special peripheral configuration or a **GPIO** by changing the position of the **Dip Switches**.

Each switch can select only one configuration. For example, in this case, we are watching the **GPIOs** configuration of an **M-Duino 57R+**. If we put the switch to the right position (ON) in the upper one, the input I2.1 will be activated and we will be able to work with this as input.
If the switch is in the left position (OFF) we will activate the SCL line which will be used for I2C communication. Keep in mind that each switch has two different configurations: you must select the right (ON) or the left (OFF) option.



I2.1 input enabled - SCL disabled



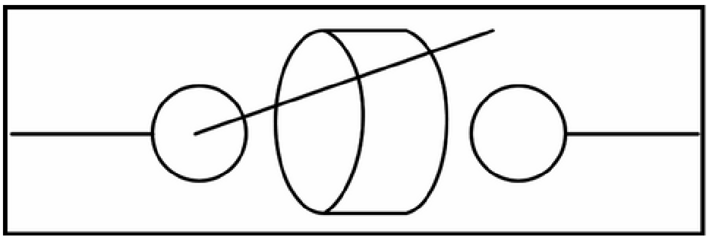
I2.1 input disabled- SCL enabled

Input types

In all of the **Industrial Shields Arduino based PLCs,** analog inputs can work at:

- 0V - 10V analog input

Analog inputs have a special draw in the case of the **PLC**:

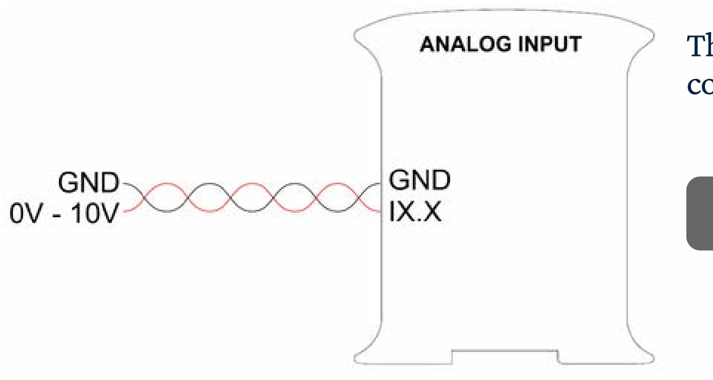


0V - 10Vdc Analog input

Basics about analog inputs of an industrial PLC

Hardware

All the analog inputs are not opto-isolated (they use the same **GNDs** as the **PLC**).



The following image shows how to connect an analog input to the **PLC**:

0V - 10Vdc Analog input

Software

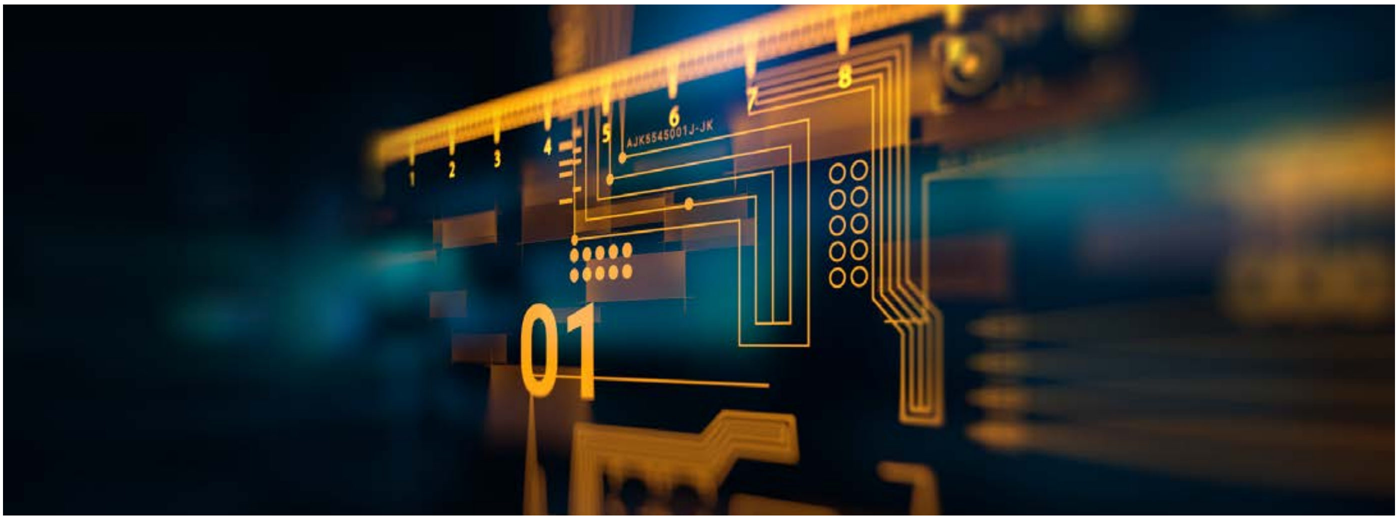
In order to program the analog **GPIOs**, we must keep in mind that we can read the values with the following command:

```
analogRead(GPIO);
```

This function returns a value between 0 and 1023 depending on the applied voltage level to the input (0V it is equal to 0, and 10V is equal to 1023).
GPIO is the name of the input. Imagine we want to know the state of the "I0.12" input, then, we must write this line:

```
analogRead(I0_12);
```

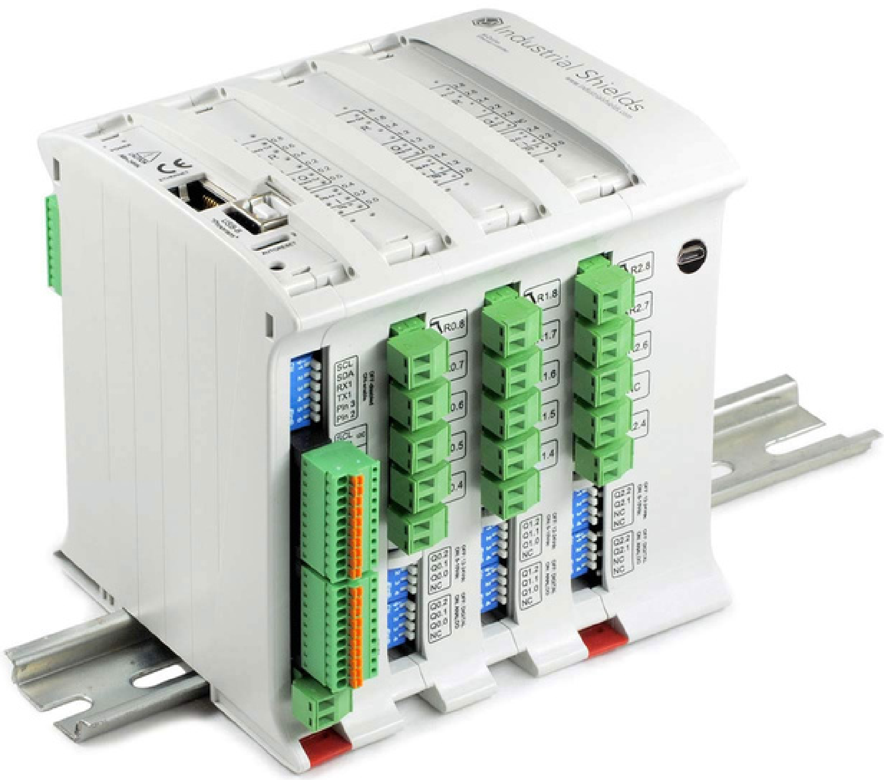
We must know that we do not need to configure the analog inputs as analog. **Industrial Shields'** libraries do all the work for us.



Basics about analog inputs of an industrial PLC

Example
You can see a read analog GPIO example in the following paragraph:

```
// Analog read example
// This example reads the I0_12 and shows via serial the value

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);
}

// Loop function
void loop()
{
    int value = analogRead(I0_12);
    Serial.println(value);
}
```



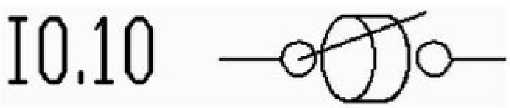
Analog inputs of Raspberry PLC

Input types

On all **Industrial Shields PLCs**, analog inputs can work at:

- 0 Vdc - 10 Vdc input

Each of them has a particular drawing in the case of the PLC:



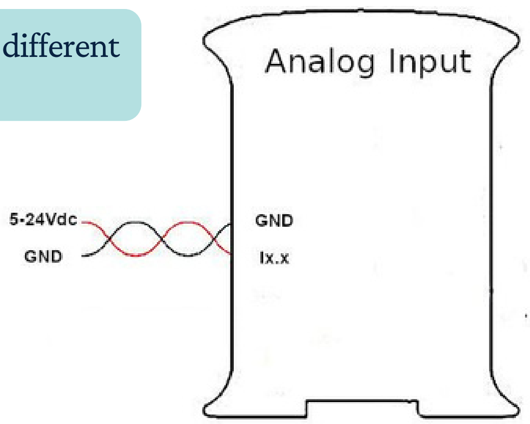
0 - 10 Vdc Analog Input

Hardware

Not all the inputs must be connected in the same way. While non-isolated inputs must be referenced to the same ground as the PLC, isolated inputs can be connected to input grounds, allowing the PLC systems to be isolated. Anyway, the optoisolated input can be connected to the PLC ground as well.

The following pictures show how to connect the different inputs to the PLC for industrial automation:

5 - 24 Vdc Input



Software

How to work with Bash Scripts

Raspberry Pi industrial PLC has default bash scripts for working with the inputs. All the inputs and outputs scripts must be executed from the correct path. It depends on the shield type of the I/O executed.

Depending on the shield of the I/O that you need to activate, you must execute the scripts from a specific path:

- Analog/Digital Shields

```
> cd /home/pi/test/analog
```

- Relay Shield

```
> cd /home/pi/test/relay
```

Analog inputs of Raspberry PLC

The get-digital-input script will show the value of the selected input pin. Only the pin we are going to work with will be provided. The return value will be in the range of 0 to 4096 (10 Vdc).
In order to call the function, we will do the following:

```
> ./get-analog-input <input>
```

Example for the I0.0 input returning a True value

```
> ./get-analog-input I0.12
4096
```

How to work with Python

The bash commands are the base for working easily with the industrial Raspberry PLC. In order to work with python files, if you want to interact with the IOs of the PLC, you will have to call these scripts.

To edit the files you will work with the Nano editor included by default and Python3.

```
nano digital_inputs.py
```

Python allows you to execute a shell command that is stored in a string using the subprocess library. In order to work with it, you will have to import it at the start of the file.

```
import subprocess
import time

def str2dec(string):
   return (string[0:-1])

def adc(value):
   return (10*int(str2dec(value)))/4096

if __name__ == "__main__":
   print("Start")
   while True:
      try:
         x = subprocess.run(["./get-analog-input","I0.12"],
         stdout=subprocess.PIPE, text=True):
         print(adc(x.stdout))
         time.sleep(1)
      except KeyboardInterrupt:
         print("\nExit")
         break
```

In order to execute the Python program, you will call it as the follow:

```
> python3 analog_outputs.py
```

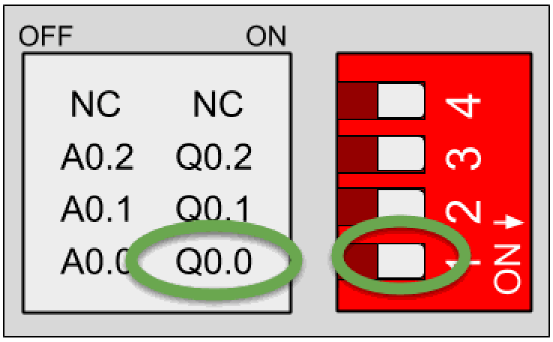
To exit the program, just press ^C.

Basics about analog outputs of an industrial PLC

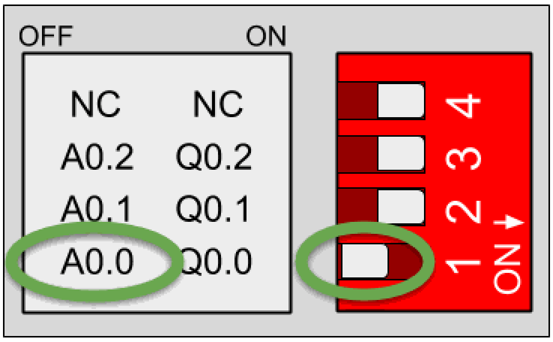
Configuring the switches

Many of the analog outputs are always connected to the internal Arduino, but in some cases, the user can choose between a special peripheral configuration or a GPIO by changing the position of the Dip switches.

Each switch can select only one configuration. For example, in this case you can see the configuration of a **GPIO** on an **M-Duino 21+**. If you put the switch in the bottom right corner (ON), the output Q0.0 will be activated and you will be able to work this digitally. If you switch to the left (OFF) position, you will activate the output as analogue. Note that each switch has two settings: you must select either the right (ON) or the left (OFF) option.



Q0.0 enabled - A0.0 disabled



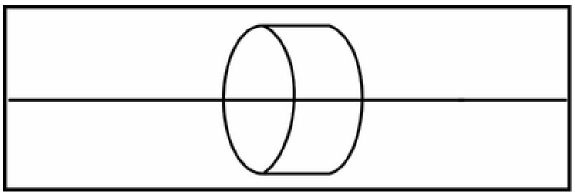
Q0.0 disabled - A0.0 enabled

Types of outputs

On all **Industrial Shields Arduino based PLCs**, analog outputs can operate on:

- Analog output 0V - 10V

Analog outputs have a special pattern on this type of PLC:

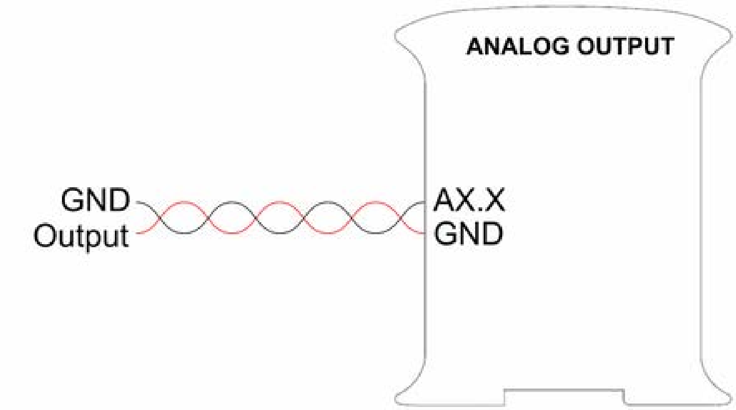


Analog output print

Hardware

The following picture shows how to connect the analog output to the **PLC**:

Analog output 0Vdc -10Vdc



Basic functions of the analog outputs of an industrial PLC

Software

To program the analog outputs, you must take into account that you can write the values with the following command:

```
analogWrite(GPIO,value);
```

This function sets the value of the analog output "A0.0" to 255 (i.e. 10V):

```
analogWrite(A0_0,255);
```

Example

You can see an analog **GPIO** written in the next paragraph:

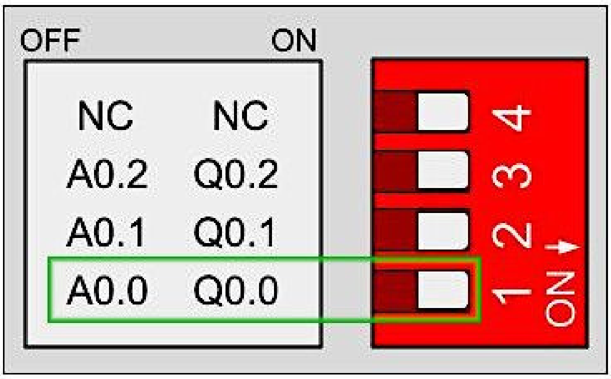
```
// Analog write example
// This example writes the A0_0 and shows via serial the value

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);
}

// Loop function
void loop()
{
    Serial.println("Value: 0");
    analogWrite(A0_0, 0);
    delay(1000);
    Serial.println("Value: 100");
    analogWrite(A0_0,100);
    delay(1000);
    Serial.println("Value: 255");
    analogWrite(A0_0,255);
    delay(1000);
}
```

Raspberry Industrial PLC analog outputs

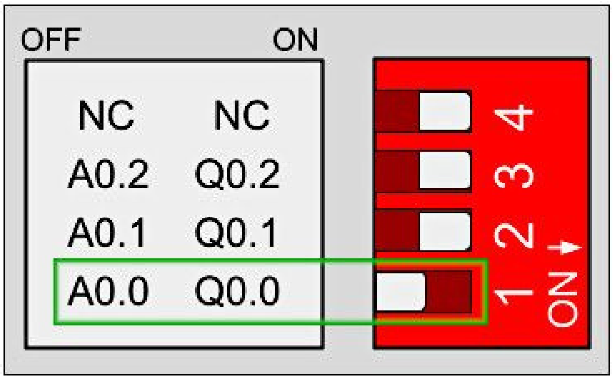
Switch configuration

Most of the digital outputs are always connected to the internal **Raspberry Pi**, but in some cases, users can choose between a special peripheral configuration or a GPIO by changing the position of the Dip Switches.

Each switch can select only one configuration. For example, in this case you can see the GPIO configuration of a **Raspberry Pi based PLC 21+**. If you set the switch to the right (ON) position at the bottom, it will activate the Q0.0 output and you will be able to work as digital. If the switch is in the left (OFF) position, it will activate the output as analog.
Note that each switch has two different settings: you must select either the right (ON) or the left (OFF) option.



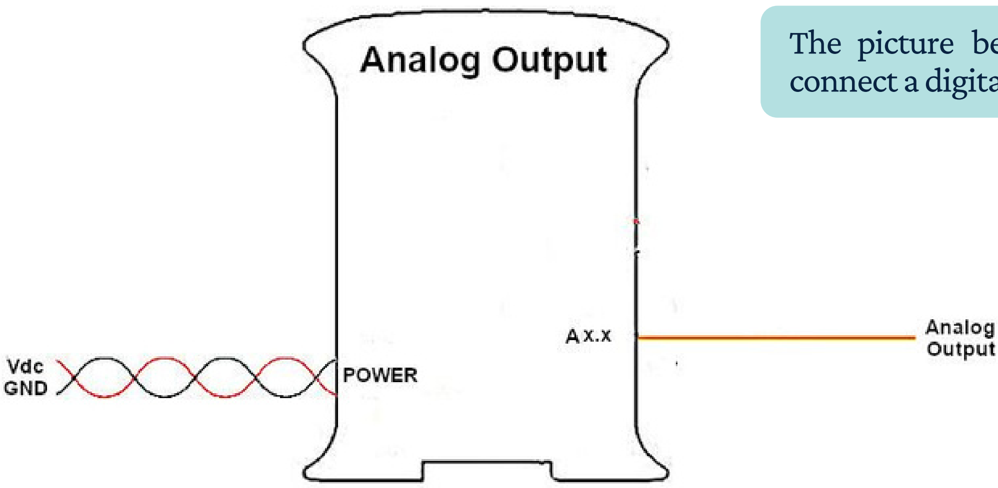
A0.0 Disabled - Q0.0 Enabled



A0.0 Enabled - Q0.0 Disabled

Hardware

The picture below shows how to connect a digital output to the PLC:



Raspberry Industrial PLC analog outputs

Software

How to work with Bash Scripts

Raspberry Pi PLC has default bash scripts to work with the inputs. All input and output scripts must be run from the correct path. It depends on the type of I/O shield executed. Depending on the area of the I/O you need to activate, you must run the scripts from a specific path:

- Analog/Digital Zone

```
> cd /home/pi/test/analog
```

- Relay Zone

```
> cd /home/pi/test/relay
```

The set function will initialise the pin. You will provide the pin you are going to work with and the value to be set. For the analog option, the value will work in a range from 0 to 4095, this being the maximum possible value (10 Vdc).
By default, if no value option is provided, it will be initialised as 50% for the analogue outputs. If any other option is chosen, an error code will warn you. To call the function, you must do the following:

```
> ./set-digital-output <output> <value>
```

By default, if no value option is provided, it will be initialised as 50% for analog outputs. If you choose any other option, an error code will warn you. To call the function, you must do the following:
The pins that can work with both analogue/digital configurations are:

Q0.5	Q1.5	Q2.5
Q0.6	Q1.6	Q2.6
Q0.7	Q1.7	Q2.7

Example:

```
> ./set-analog-output A0.5 2048
> ./set-analog-output A0.5 4096
> ./set-analog-output A0.5 0
> ./set-analog-output A0.5 stop
> ./set-digital-output Q0.5 1
```

How to work with Python

The bash commands are the basis for working easily with the **Raspberry Pi industrial PLC.**
To work with python files, if you want to interact with the PLC IOs, you will have to call these scripts.

Raspberry Industrial PLC analog outputs

To edit the files, we work with the default Nano editor and Python3.

```
> nano analog_outputs.py
```

Python allows you to execute a shell command that is stored in a string using the **os.system()** function. In order to work with it, you will have to import its library at the beginning of the file. In addition, you will include the timing library to invoke a 2 second delay.

```
import os
import time
```

In this example program, you will change the values of the A0.5 output of the **Raspberry Pi industrial PLC**. To do this, you will implement a loop that will increment the output value by 25% every 2 seconds and reset it after reaching 100%.

```
import os
import time
os.system("echo Start")
while True:
    try:
        os.system("sudo ./set-analog-output A0.5 0")
                time.sleep(2)
        os.system("sudo ./set-analog-output A0.5 1024")
                time.sleep(2)
        os.system("sudo ./set-analog-output A0.5 2048")
                time.sleep(2)
        os.system("sudo ./set-analog-output A0.5 3072")
                time.sleep(2)
        os.system("sudo ./set-analog-output A0.5 4095")
                time.sleep(2)

    except KeyboardInterrupt:
        os.system("sudo ./set-analog-output A0.5 0")
        os.system("echo End")
        break
```

To run the Python program, you will call it as follows:

```
> python3 analog_outputs.py
```

To exit the programme, simply press ^C.

Use of interrupt inputs on industrial Arduino boards

Introduction

Interrupts, as their name implies, are a method of stopping the process being executed by the processor in order to execute a smaller subroutine. This method has a lot of real-world application and is an important part of automation.

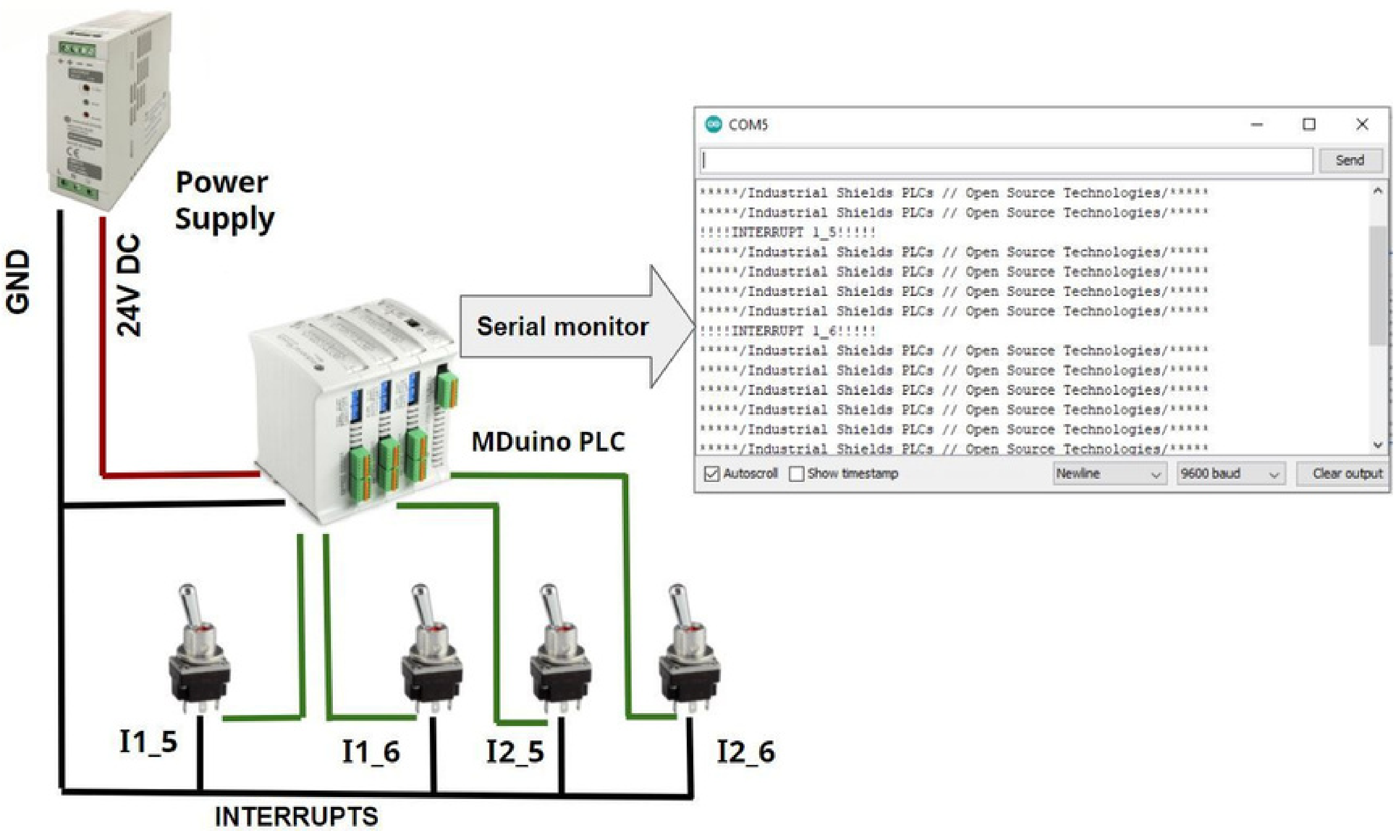
These interrupts can be generated externally with the help of hardware such as a switch or a sensor, or they can be generated by software when a particular condition is met or a set of instructions has been executed.

In this reading, you will learn how to use **hardware interrupts on an Arduino PLC**. This is an overview with example code to demonstrate the capabilities with respect to interrupt handling and execution of the boards that support this feature.

For ease of understanding and demonstration, we will loop a text string on the serial monitor and interrupt it with some hardware buttons.

Hardware

In this guide the industrial controller to be used is an **M-DUINO PLC Arduino Ethernet 58 IOs Analog/Digital PLUS**. If you are using a different board, make sure the interrupt inputs are enabled and check the DIP switch status.



Use of interrupt inputs on industrial Arduino boards

Syntax

To initialise the interrupt input on the board, you must use the attachInterrupt() function with the following parameters:

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)
```

- **digitalPinToInterrupt(pin) -->** Used to initialise the given pin and assign it as the interrupt.

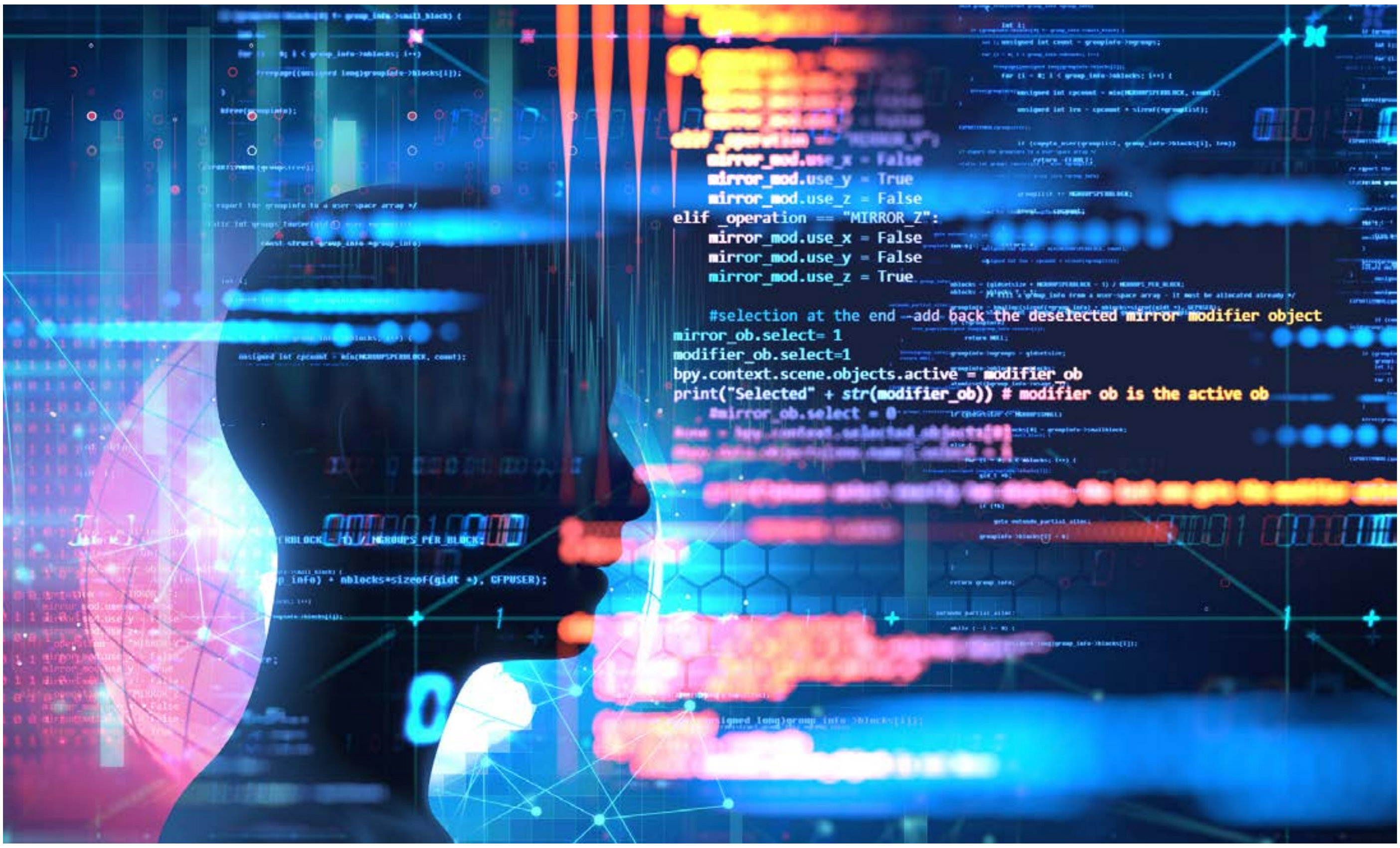
- **pin -->** In this case, you will not use the Arduino pin number, but the ones written on their respective boards. For example, "I1_5" for the board you are using.

- **ISR --> ISR -->** This stands for Interrupt Service Routine, it is a function that is called when the interrupt is triggered. This must not take any parameters and returns nothing, however, it can pass global variables.

- **mode -->** Specifies when an interrupt is to be triggered.
    - **LOW** to activate the interrupt whenever the pin is low.
    - **CHANGE** to trigger the interrupt when the pin changes value.
    - **RISING** to trigger when the pin goes from low to high.
    - **FALLING** for when the pin goes from high to low.

Due, Zero and MKR1000 plates also allow:

- **HIGH** to trigger the interrupt whenever the pin is high.



Use of interrupt inputs on industrial Arduino boards

Code

This code shows how to operate interrupts **I1_5, I1_6, I2_5 and I2_6 by applying different functions in each case.**

```
// Interrupt Example. Industrial Shields PLCs.
// Board used M-DUINO PLC Arduino Ethernet 58 IOs Analog/Digital PLUS
int val1,val2,val3,val4 = 0;
///////Setting up the board and the pins
void setup() {
  Serial.begin(9600L);

  //Initalizing interrupt I1_5
  attachInterrupt(digitalPinToInterrupt(I1_5), isrI1_5, LOW);

  //Initalizing interrupt I1_6
  attachInterrupt(digitalPinToInterrupt(I1_6), isrI1_6, CHANGE);

  //Initalizing interrupt I2_6
    attachInterrupt(digitalPinToInterrupt(I2_5), isrI2_5, RISING);

  //Initalizing interrupt I2_6
  attachInterrupt(digitalPinToInterrupt(I2_6), isrI2_6, FALLING);
}
/////// Printing a String every seconds in a loop continuously to interrupt
void loop() {
   Serial.println("*****/Industrial   Shields   PLCs   //   Open   Source
Technologies/*****");
 delay(1000);
}
///////Interrupt Service Routines
void isrI1_5(){
  Serial.println("!!!!INTERRUPT 1_5!!!!!"); //ISR for I1_5
  }
void isrI1_6 (){
  Serial.println("!!!!INTERRUPT 1_6!!!!!"); //ISR for I1_6
  }
void isrI2_5(){
  Serial.println("!!!!INTERRUPT 2_5!!!!!"); //ISR for I2_5
  }
void isrI2_6 (){
  Serial.println("!!!!INTERRUPT 2_6!!!!!"); //ISR for I2_6
  }
////////////End/////////////
```

How to program Raspberry PLC interrupt inputs in Python

How to work with Python

Code Example
For this example, you need to import the libraries that you can see at the beginning of the code, taking into account that "signal", "sys" and "RPi.gpio" are essential to work with interrupt inputs in Python with a **Raspberry Pi PLC**. The INT_GPIO must be the GPIO of the **Raspberry Pi** that you are going to configure as an interrupt input, in this case 13, which is the INT.

If you do not know the mapping between the Raspberry Pi GPIOs and the I/O of your PLC, you can take a look at these tables, also included in the Datasheet and User Guide.

Equivalence table

Pinout

Raspberry Pinout	PLC Pinout
NC	-
GPIO2	SDA
GPIO3	SCL
GPIO4	INT21
GND	-
GPIO17	INT30
GPIO27	INT20
GPIO22	IRQ SPI 485
NC	-
GPIO10	MOSI 0
GPIO9	MISO 0
GPIO11	SCLK 0
GND	-
GPIO 0	-
GPIO5	IRQ SPI CAN
GPIO6	IRQ SPI ETH
GPIO13	INT10
GPIO19	MISO 1
GPIO26	FAN CONTROL
GND	-

Digital I/Os

Digital Inputs			
PLC Pinout	Chip ADDR	Chip INDEX	GPIO
Zone A			
I0.0	ADDR = 0x21	5	-
I0.1	ADDR = 0x21	3	-
I0.2	ADDR = 0x21	2	-
I0.3	ADDR = 0x21	1	-
I0.4	ADDR = 0x21	0	-
I0.5	-	-	GPIO = 13
I0.6	-	-	GPIO = 12
Zone B			
I1.0	ADDR = 0x20	2	-
I1.1	ADDR = 0x20	1	-
I1.2	ADDR = 0x20	0	-
I1.3	ADDR = 0x21	7	-
I1.4	ADDR = 0x21	6	-
I1.5	-	-	GPIO = 27
I1.6	-	-	GPIO = 4
Zone C			
I2.0	ADDR = 0x20	6	-
I2.1	ADDR = 0x20	5	-
I2.2	ADDR = 0x20	7	-
I2.3	ADDR = 0x20	4	-
I2.4	ADDR = 0x20	3	-
I2.5	-	-	GPIO = 17
I2.6	-	-	GPIO = 16

You should also know that a signal is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to a running program.

The first function is **signal_handler**, a function that has to be called if an event that triggers a signal is anticipated, and the operating system can be told to execute it when that particular type of signal arrives.
In this case, this handler does a **GPIO.cleanup()** and a **sys.exit(0)** when it detects a CTRL+C (command that sends a SIGINT).

The second function is called **int_activated_callback** and, inside it, you can put the code you want to be executed when the interrupt is activated.

Finally, there is the **GPIO.setmode**, onfiguring it following the layout of the BCM GPIO; el GPIO.setup, configuring it with the number of the GPIO interrupt inputs, whether the trigger edge is going to be **FALLING** o el **RISING**; the trigger callback and the bouncing time (which is the period that no interrupt is going to be triggered to avoid signal bouncing). The signal.signal is the function to trigger the signal_handler, explained above.

The last infinite loop is simply to test that you can be running other processes in your code while the interrupt is ready to be triggered.

So, if you run this code, it will perform indefinite "Work" prints until the interrupt is triggered.

When a falling edge is detected on the signal, the previous prints will stop, the interrupt will trigger and you will see the "INT triggered" print once, then the "Job" prints will continue until another interrupt triggers (always respecting the 1000ms bounce time).

```
import signal
import sys
import time

import RPi.gpio as gpio
import BUTTON.GPIO as gpio

INT_GPIO = 13

def signal_handler(sif, frame):
    GPIO.cleanup()
    sys.exit(0)

def int_activated_callback(channel):
    print("INT activated")

if __name__ == '__main__':
    GPIO.setmode(GPIO.BCM)
            GPIO.setup(INT_GPIO,   GPIO.FALLING,   callback=int_activated_callback,
bouncetime=1000)
    signal.signal(signal.SIGINT, signal_handler)
    while 1:
        print ("Work")
        time.sleep(0.1)
```

The interrupt has to be triggered by an activation of the input signal, either by a rising edge or a falling edge. To test this, you can connect the GND of the sensor to the optoisolated GND of the input you are going to use ((-)IX.5) and the output of the sensor to the interrupt input signal (IX.5/INT).

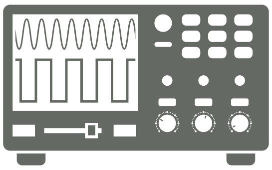
When the digital sensor is activated, you will see the activation of the interrupt as well. Here is an example of one of the PLC interrupt inputs, with the GND pin and the SIGNAL pin:

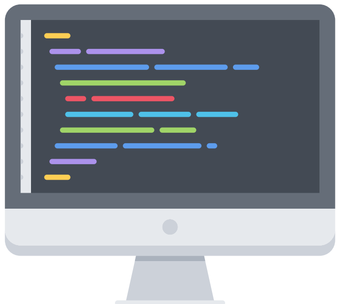
How to work with PWM outputs on the Raspberry industrial PLC

Introduction

Raspberry Pi based PLC family devices have a defined number of digital outputs. All of them can be programmed as **PWM** outputs, if necessary.

As we know, **PWM (Pulse Width Modulation)** is a type of voltage signal that is used to send information or to modify the amount of power sent by each load.





Explanation and use of the Bash Script

First of all, the Bash script you have to run to manage the PWM outputs is called "**set-analog-output**" located in the path "**/home/pi/test/analog/**". You must make sure that the output you want to configure as PWM is not configured as analogue or digital, so, to make sure, you can run the "stop" function to disable the corresponding input as digital or analogue (if it has been previously modified). To work as PWM, the DIP switch of the output in question must be in the "ON" position.

```
./set-analog-output A0.5 stop
```

Or:

```
./set-digital-output Q0.5 stop
```

To execute the script, the "**set-analog-output**" script must be called, but with a digital output and the pulse width as parameters. The pulse width is the high time period of the duty cycle and ranges from 0 to 4095 (12 bits).

For example, if you want a high time period of 25%, set 1024 and, if you want a high time period of 100%, set 4095.

```
./set-analog-output Q0.5 4095
```

Note: Refer to the User's Guide for PWM-compatible outputs.

How to work with PWM outputs on the Raspberry industrial PLC

The PWM parameters of the script do not have to be modified to ensure correct behaviour. Here you can see the script:

```
#!/bin/bash

# PWM period in nanoseconds
PERIOD="2000000"

case ${1} in
    A0.5) ADDR=40; INDEX=10 ;;
    A0.6) ADDR=40; INDEX=1 ;;
    A0.7) ADDR=40; INDEX=0 ;;
    A1.5) ADDR=40; INDEX=3 ;;
    A1.6) ADDR=40; INDEX=5 ;;
    A1.7) ADDR=40; INDEX=8 ;;
    A2.5) ADDR=41; INDEX=2 ;;
    A2.6) ADDR=41; INDEX=1 ;;
    A2.7) ADDR=41; INDEX=0 ;;
    Q0.0) ADDR=40; INDEX=15 ;;
    Q0.1) ADDR=40; INDEX=14 ;;
    Q0.2) ADDR=40; INDEX=13 ;;
    Q0.3) ADDR=40; INDEX=12 ;;
    Q0.4) ADDR=40; INDEX=11 ;;
    Q0.5) ADDR=40; INDEX=10 ;;
    Q0.6) ADDR=40; INDEX=1 ;;
    Q0.7) ADDR=40; INDEX=0 ;;
    Q1.0) ADDR=40; INDEX=2 ;;
    Q1.1) ADDR=40; INDEX=9 ;;
    Q1.2) ADDR=40; INDEX=6 ;;
    Q1.3) ADDR=40; INDEX=4 ;;
    Q1.4) ADDR=40; INDEX=7 ;;
    Q1.5) ADDR=40; INDEX=3 ;;
    Q1.6) ADDR=40; INDEX=5 ;;
    Q1.7) ADDR=40; INDEX=8 ;;
    Q2.0) ADDR=41; INDEX=6 ;;
    Q2.1) ADDR=41; INDEX=7 ;;
    Q2.2) ADDR=41; INDEX=5 ;;
    Q2.3) ADDR=41; INDEX=4 ;;
    Q2.4) ADDR=41; INDEX=3 ;;
    Q2.5) ADDR=41; INDEX=2 ;;
    Q2.6) ADDR=41; INDEX=1 ;;
    Q2.7) ADDR=41; INDEX=0 ;;
    *)
        echo "Output not defined" >&2
        exit 1
        ;;
esac

VALUE="${2:-50}"

if [ -z "${PWM}" ]; then
    CHIP_BASE_DIR="/sys/bus/i2c/devices/1-00${ADDR}/pwm"
    CHIP_NAME="$(ls ${CHIP_BASE_DIR})"
    CHIP_DIR="${CHIP_BASE_DIR}/${CHIP_NAME}"
    CHIP="${CHIP_NAME#pwmchip}"

    PWM="${INDEX}"
fi

if [ "${VALUE}" = "stop" ]; then
    echo "${PWM}" > ${CHIP_DIR}/unexport
    exit 0
fi

if [ ! -d ${CHIP_DIR}/pwm${PWM} ]; then
    echo "${PWM}" > ${CHIP_DIR}/export
fi

echo "${PERIOD}" > ${CHIP_DIR}/pwm${PWM}/period

DUTY_CYCLE="$((${2} * ${PERIOD} / 4095))"
echo "${DUTY_CYCLE}" > ${CHIP_DIR}/pwm${PWM}/duty_cycle
```

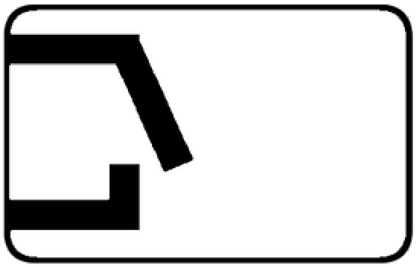
Basics of the internal relays of an industrial PLC

Introduction

Relay characteristics
There is only one type of relay in our PLCs.
These relays have the following characteristics:

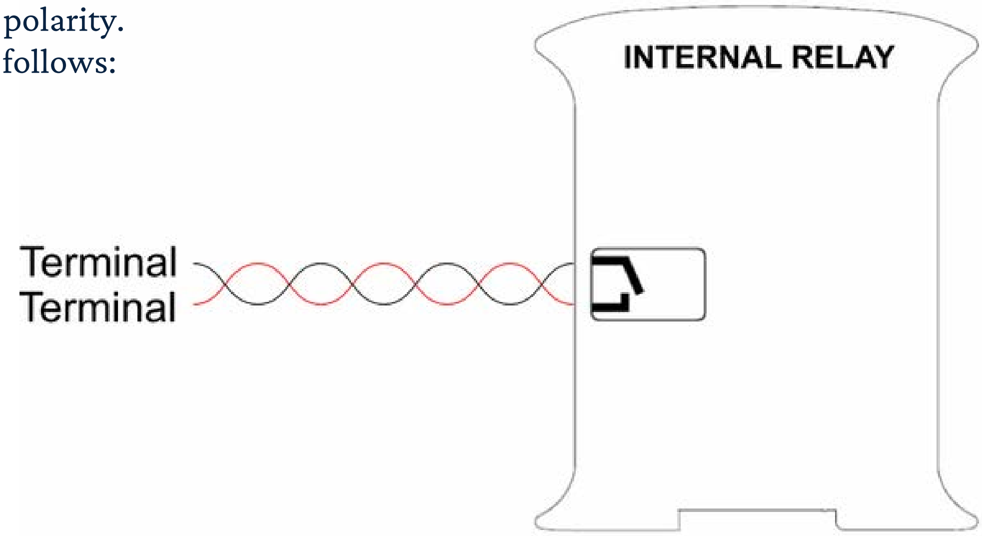
- Up to 5A working up to 250Vac
- Up to 3A working up to 30Vdc

The following illustration shows how to identify GPIO relays:



Hardware

The internal relays have no polarity.
They must be connected as follows:



Software

To program the internal relays, you should note that you can write the values using the following command:

```
digitalWrite(relay,value);
```

"Relay" has to be the reference of the target relay. The Ardbox family has the references as "R1", and for example, the M-Duino family has the reference as "R0.1" with the relay. You must write "HIGH" or "LOW" in the "value" parameter. **"HIGH" is equivalent to relay closed and "LOW" is equivalent to relay open.**

```
digitalWrite(R1,HIGH);    // Ardbox family
digitalWrite(R0_3,LOW);   // M-Duino family
```

Examples

You can see how to handle an internal relay in the Ardbox family in the example below:

```
// Internal relay example in Ardbox family
// This example writes the R1 and shows via serial the state

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);
}

// Loop function
void loop()
{
    Serial.println("Opening");
    digitalWrite(R1, HIGH);
    delay(1000);
    Serial.println("Closing");
    digitalWrite(R1, LOW);
    delay(1000);
}
```

The following example shows how to operate an internal relay with the M-Duino family:

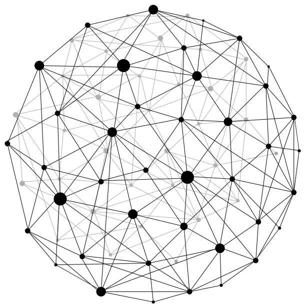
```
// Internal relay example in M-Duino family
// This example writes the R0_1 and shows via serial the state

// Setup function
void setup()
{
    // Set the speed of the serial port
    Serial.begin(9600UL);
}

// Loop function
void loop()
{
    Serial.println("Opening");
    digitalWrite(R_01, HIGH);
    delay(1000);
    Serial.println("Closing");
    digitalWrite(R_01, LOW);
    delay(1000);
}
```

Industrial Protocols

The main communication protocols, and the specific ones in Industrial Shields PLCs.

Protocol?

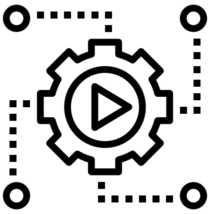
A **Communication Protocol** is a system of rules that allow **two or more entities** in a communication system to communicate with each other to **transmit information** by means of any type of variation of a physical magnitude.



It is also defined as a set of standards that allows communication between computers, establishing how they are identified on the network, how data is transmitted and how the information is processed.

The ultimate goal of these processes is the exchange of information between two or more entities, using the same previously agreed structure.

With the introduction of technology in industrial processes with monitoring, control or automation, the number and variety of elements that can communicate has increased.



In the industrial environment, there are sensors, motors, actuators, etc. which have also incorporated local or remote control through all kinds of devices or automations.

When the type and number of entities that must communicate is so large and varied, it is imperative to define and delimit how they must do so; hence the importance of communication protocols.

List of some of the most common protocols in industrial communication environments*.

- *AS-i: Actuator-Sensor interface*
- *BSAP: Bristol Standard Asynchronous Protocol*
- *CC-Link: Industrial Networks*
- *CIP: Common Industrial Protocol*
- *ControlNet*
- *DeviceNet*
- *DNP3*
- *DirectNet*
- *EtherCAT*
- *EtherNET/IP*
- *FINS: Factory Interface Network Service*
- *Foundation Fieldbus: H1 & HSE*
- *HART Protocol*
- *HTTP/HTTPS*
- *Interbus*
- *MECHATROLINK*
- *MelsecNet, and MelsecNet II, /B, and /H*
- *Modbus: RTU, TCP/IP or ASCII.*
- *MQTT: Message Queuing Telemetry Transport*
- *OSGP – The Open Smart Grid Protocol*
- *Optomux*
- *PieP: An Open Fieldbus Protocol*
- *Profibus*
- *PROFINET*
- *RAPIEnet: Real-time Automation Protocols for Industrial Ethernet*
- *SERCOS III*
- *GE SRTP*
- *SynqNet*
- *TTEthernet*
- *MPI – Multi-Point Interface*

**Some of these protocols are proprietary, although widely used in professional and industrial environments.*

HTTP & HTTPS

HTTP is the acronym for Hypertext Transfer Protocol.

When you type http:// in your address bar in front of the domain, it tells the browser to connect via HTTP.

HTTP uses **TCP (Transmission Control Protocol)**, usually through **port 80**, to send and receive data packets over the web.

HTTPS stands for Hypertext Transfer Protocol Secure (also known as HTTP over TLS or HTTP over SSL).

When you type https:// in the address bar in front of the domain, you are telling the browser to connect over HTTPS.
Generally, sites that operate over HTTPS will have a redirect in place, so even if you type http:// you will be redirected to deliver over a secure connection.

HTTPS also uses **TCP (Transmission Control Protocol)** to send and receive data packets, but it does so through **port 443**, within a Transport Layer Security (TLS) encrypted connection. (TLS).

The HTTP protocol presents a security problem in that the information flowing from a server to a browser is not encrypted, which means it can be easily stolen. HTTPS (Hypertext Transfer Protocol Secure) protocols solve this by using an SSL (secure sockets layer) certificate, which helps to create a secure encrypted connection between the server and the browser.

HTTP messages are in plain text which makes them more readable and easier to debug. This has the disadvantage of making messages longer.

Ethernet

Ethernet is the traditional technology, computer networking technologies, which are commonly used in local area networks **(LAN),** metropolitan area networks **(MAN)** and wide area networks **(WAN).**

Ethernet communication uses the **LAN protocol** which is technically known as the **IEEE 802.3 protocol**. This protocol has evolved and improved over time to transfer data at the rate of one gigabit per second.

Ethernet uses different protocols to communicate. Some of them are HTTP, HTTPS, MQTT and Modbus protocols.

The PLCs of the **M-Duino** family incorporates the W5500 IC. The W5500 is an integrated TCP/IP Ethernet controller that provides an easier Internet connection to the embedded systems.

MQTT

MQTT (Message Queuing Telemetry Transport) is an open OASIS and ISO (ISO/IEC 20922) lightweight, subscribe-and-publish network protocol that transports messages between devices.

The protocol typically runs over TCP/IP; however, any network protocol that provides orderly, lossless and bidirectional connections can support MQTT.

It is designed for connections to remote locations where a **"small code footprint"** is required or network bandwidth is limited.

MODBUS

The Modbus Protocol is a messaging framework developed by Modicon. It is used to establish master-slave/client-server communication between devices. Modbus has many protocol options, but the two most commonly used are **Modbus RTU (Remote Terminal Unit)** and **Modbus Transmission Control Protocol (TCP/IP).**

Modbus is an open standard and is a widely used network protocol in the industrial manufacturing environment. Modbus RTU mode is the most common implementation, although the Modbus TCP/IP protocol is gaining ground and is ready to overtake it.

A Modbus communication is always initiated by the master node to the slave node. Also, the slave nodes will never transmit data without receiving a request from the master node or communicate with each other.
The master node initiates only one Modbus transaction at a time.

Modbus RTU mode uses binary coding and CRC error checking.
It is an efficient protocol in which every eight bits (one byte) of a message contains two 4 bit hexadecimal characters. In addition, each message must be transmitted in a continuous flow.

The coding system of each byte (11 bits) in RTU mode is as follows:

- Bits per byte: 1 start bit, 8 data bits, least significant bit sent first, 1 bit to complete parity, 1 stop bit.

Modbus RTU packages are only intended to send data; they do not have the ability to send parameters, such as point name, resolution, units, etc.

PROFINET

PROFINET is an open standard based on **Industrial Ethernet, TCP/IP** and some communication standards belonging to the IT world.

Starting from a basic connectivity, such as the **Ethernet cable**, and established communication frames that would correspond to levels 1 and 2 of the **OSI model**, PROFINET incorporates new functionalities that would allow **modifying level 7**, of application, through a specific interpretation of the transmitted data for each case.

PROFINET is one of the most used communication standards in automation networks.

PROFINET follows the client-server model for data exchange, using three kinds:

IO Controller
It is usually the PLC in which the program with the automation is executed.
It is comparable to a Master device in Modbus.

IO Device
It is a device with multiple inputs and outputs, connected to one or more IO Controllers via PROFINET. It is comparable to a Slave device in Modbus.

IO Supervisor
Usually a programming device, such as a computer, for commissioning or diagnostic purposes.

Its main features are:

It works via Ethernet in **real time**.

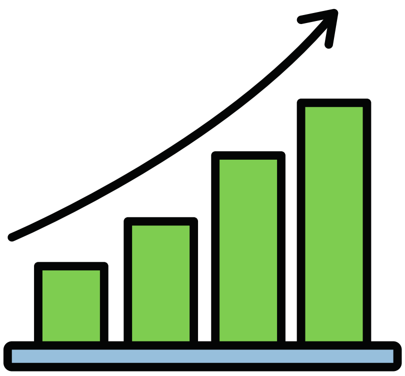
It can work on a **single network cable**, together with other industrial Ethernets also based on standard Ethernet.

Wireless communication. It is part of the PROFINET specification for two standards: Wi-Fi and Bluetooth.

FSU or Fast Start-Up: The PROFINET **Quick Start** function allows the I/O device to immediately enter the "on" state in response to signals from the I/O Controller.

In addition, **PROFINET** offers much better performance than other fieldbuses in terms of:

- Unlimited scalability
- Unlimited address space
- Larger message size (1440 vs. 244 bytes)
- Machine-to-machine communication (M2M)
- Vertical integration capabilities
- The ability to coordinate more drive shafts (32 shafts vs. > 150), with IRT updates in the range of 1 ms, with jitter of less than 1 microsecond.



Modbus RTU Master Library for industrial automation

Learn what Modbus RTU is and its applications

In the Arduino automation area, the **Modbus RTU protocol is a means of communication** that allows the exchange of data between programmable logic controllers (industrial PLC controller Arduino) and computers.

Electronic devices can exchange information through serial lines using the **Modbus protocol.**

What is Modbus RTU

Modbus is a communication protocol located at levels 1, 2, and 7 of the OSI Model, based on the master/slave architecture, designed in 1979 by Modicon for its range of PLCs.

Converted into *a de facto* **standard communications protocol in the industry**, let's see some of the main features:

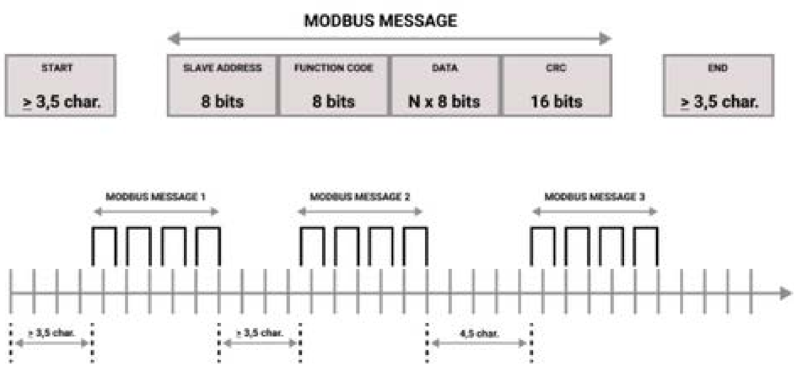
1. Designed with its use in **industrial applications** in mind.
2. It is **public** and **free**.
3. It is easy to implement and requires **little development**.
4. Handles **blocks of data** without assuming restrictions.
5. Each of the messages includes **redundant information** that ensures its **integrity at reception**.
6. The **basic** Modbus **commands** allow you to **control an RTU device** to modify the value of any of its registers or to request the content of these registers.

How does Modbus RTU work

Modbus RTU is the **most common implementation** available for Modbus.

Modbus RTU is used in **serial communication** and makes use of a **compact and binary representation** of the data for protocol communication.

Modbus messages are divided into **idle periods** as you can see in the picture.



How does Modbus RTU work

Each device on a Modbus communication has a **unique address.**

The Modbus RTU works by **RS-485** which is a single cable multi-drop network, only the node assigned as the Master may initiate a command. All the other devices are slaves and answer requests and commands.

A Modbus command contains the Modbus address of the device it is intended for. **Only the addressed device will respond and act on the command, even though other devices might receive it.**

Also, it is important to say that **all Modbus commands contain checksum information** to allow the recipient to detect transmission errors.

Let's give an example!

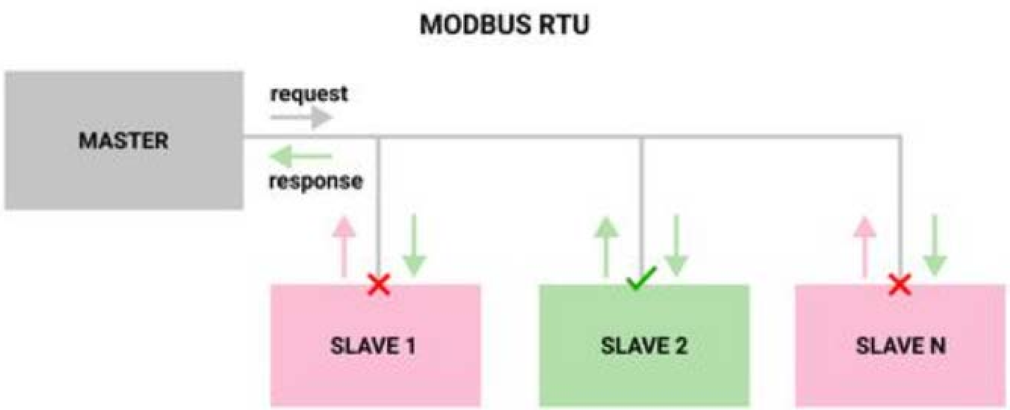
Imagine that we have a Modbus serial network, where there is a **master** and **up to 31 slaves**, each with a unique slave address.

The master only wants to **send a message to slave number 2** requesting the value of 6 input registers.

So, the master would send a message and all the slaves would receive the message, but only slave number 2 will respond and act on the command, even though other devices might receive it.

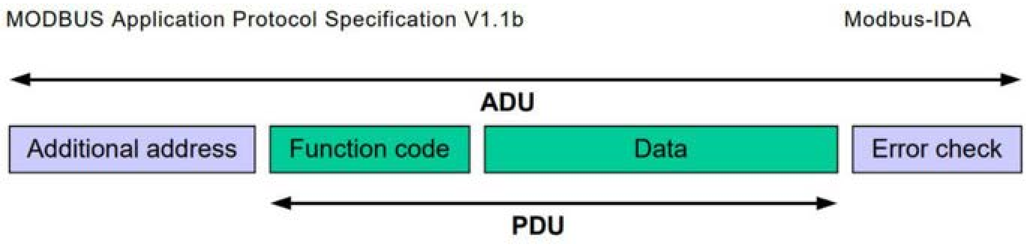
With this example, we are going to create a Modbus RTU message along with this section.

Modbus message at the moment: **02 (slave address)**.



Modbus general frame structure

The **Modbus RTU Application Data Unit (ADU)** consists of the shown elements:



- **Address:** We set the slave address for the device to which we want to send the message.
- **Function Code:** The number of the function code. You can see the table of the function codes in the "Modbus function codes" section.
- **Data:** The message itself. This can vary depending on the function code.
- **CRC:** The number of the cyclic redundancy check. It must be calculated.

Of these, The **Function Code** and **Data** constitute the **Protocol Data Unit (PDU).**

Modbus function code

Modbus is a request/reply protocol and offers services specified by **function codes**. MODBUS function codes are **elements** of MODBUS request/reply PDUs.

The function code field of a MODBUS data unit is coded in **one byte**. Valid codes are in the range of **1 to 255 decimal** (the range 128 – 255 is reserved and used for exception responses). When a message is sent from a Client to a Server device the function code field tells the server what kind of action to perform. Function code **"0"** is not valid. **Sub-function codes** are added to some function codes to define multiple actions.

Below you can find the list of function codes and their functions:

Modbus function codes

Function type			Function name	Function code	Comment
Data Access	Bit access	Physical Discrete Inputs	**Read Discrete Inputs**	2	
		Internal Bits or Physical Coils	**Read Coils**	1	
			Write Single Coil	5	
			Write Multiple Coils	15	
	16-bit access	Physical Input Registers	**Read Input Registers**	4	
		Internal Registers or Physical Output Registers	**Read Multiple Holding Registers**	3	
			Write Single Holding Register	6	
			Write Multiple Holding Registers	16	
			Read/Write Multiple Registers	23	
			Mask Write Register	22	
			Read FIFO Queue	24	
	File Record Access		Read File Record	20	
			Write File Record	21	
Diagnostics			Read Exception Status	7	serial only
			Diagnostic	8	serial only
			Get Com Event Counter	11	serial only
			Get Com Event Log	12	serial only
			Report Slave ID	17	serial only
			Read Device Identification	43	
Other			Encapsulated Interface Transport	43	

Modbus Object Types

In Modbus, the **data types** can be divided majorly into two types: **Coils and Registers.**

> The coils can be understood as digital as can only be either **ON (1) or OFF (0)**.

Some coils can represent inputs and some as outputs.

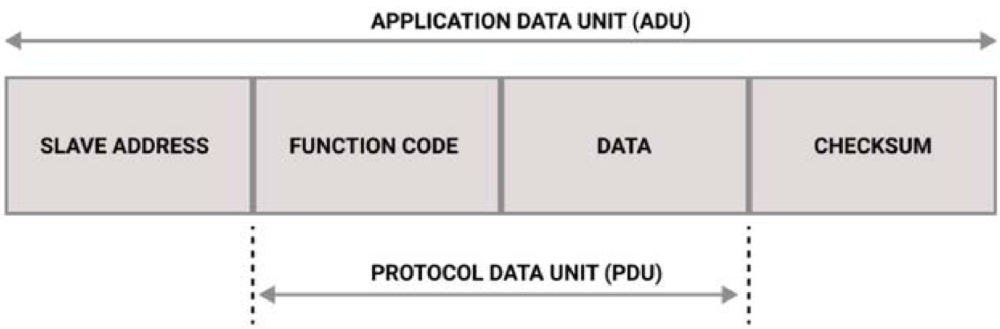
The **Registers are of 16 bits (2 bytes) unsigned registers and therefore can have values from 0 to 65535 (0 to FFFF).** Though it has its limitations such as it cannot represent negative numbers, floating-point numbers, or values with representation greater than 65535. The below table summarises the object types.

The **four primary tables** are the following:

Primary tables	Object Type	Type of	Comments
Discrete inputs (Inputs)	Single bit	Read-Only	This type of data be provided by an I/O system.
Coils (Outputs)	Single bit	Read-Write	This type of data can be alterable by an application.
Input Registers (Inputs)	16-bit word	Read-Only	This type of data can be provided by an I/O system.
Holding Registers (Outputs)	16-bit word	Read-Write	This type of data can be alterable by an application program.

Function codes descriptions

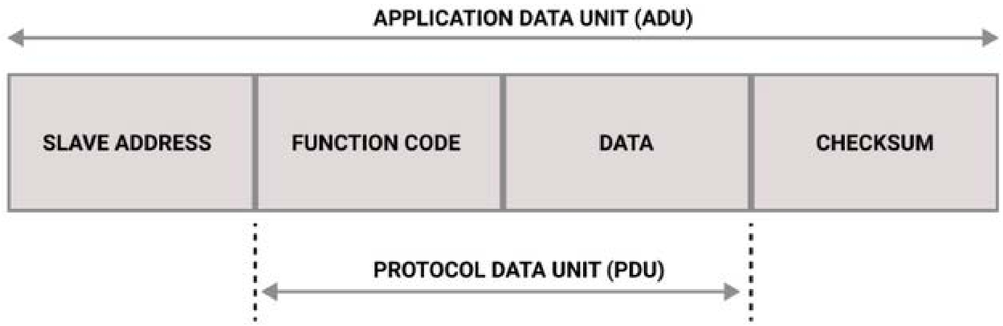
Modbus requests and responses contain an **Application Data Unit (ADU)** which contains a **Protocol Data Unit (PDU)**.



Modbus Data Format

Function codes descriptions

Modbus requests and responses contain an **Application Data Unit (ADU)** which contains a **Protocol Data Unit (PDU)**.



(0x01) Read Coils

This function code is used to read from **1 to 2000 contiguous status of coils** in a remote device.

The coils in the response message are **packed as one coil per bit of the data field**.

Status is **indicated as 1: ON and 0: OFF**.

The **LSB** of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

Request

Function Code	1 Byte	0x01
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Coils	2 Bytes	1 to 2000 (0x7D0)

Response

Function Code	1 Byte	0x01
Byte Count	2 Bytes	N*
Coil Status	n Bytes	n = N or N+1

***N = Quantity of Outputs / 8, if the remainder is different of 0 => N= N+1**

Modbus Data Format

Example of a request to read discrete outputs 20 –38

Request		Response	
Field Name	Hex	**Field name**	Hex
Function	01	**Function**	01
Starting Address Hi	00	**Byte Count**	03
Starting Address Lo	13	**Outputs Status 27-20**	CD
Quantity of Outputs Hi	00	**Outputs Status 35-28**	6B
Quantity of Outputs Lo	13	**Outputs Status**	05

***The CRC must be calculated.**

(0x02) Read Discrete Inputs

This function code is used to read from **1 to 2000 contiguous status of discrete inputs** in a remote device.

The **Request PDU** specifies the starting address, i.e. the **address of the first input specified**, and the number of inputs.

In the **PDU Discrete Inputs** are addressed starting at zero. Therefore, Discrete inputs numbered 1-16 are addressed as 0-15.

The discrete inputs in the response message are **packed as one input per bit of the data field**.

Status is **indicated as 1= ON; 0= OFF.**

The **LSB** of the first data byte contains the input addressed in the query. The other inputs follow toward the **high order end of this byte**, and from low order to high order in subsequent bytes.

Request

Function Code	1 Byte	0x02
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Inputs	2 Bytes	1 to 2000 (0x7D0)

Modbus Data Format

(0x02) Read Discrete Inputs

Response

Function Code	1 Byte	0x02
Byte Count	1 Byte	N*
Input Status	N* x 1 Byte	

Example of a request to read discrete inputs 197 – 218

Request		Response	
Field Name	Hex	**Field name**	Hex
Function	02	**Function**	02
Starting Address Hi	00	**Byte Count**	03
Starting Address Lo	C4	**Outputs Status 27-20**	AD
Quantity of Outputs Hi	00	**Outputs Status 35-28**	DB
Quantity of Outputs Lo	16	**Outputs Status**	35

(0x03) Read Holding Registers

This function code is used to **read** the **contents** of a **contiguous block** of **holding registers** in a remote device. The **Request PDU** specifies the starting register address and the number of registers. In the **PDU Registers** are a**ddressed starting at zero**. Therefore registers numbered 1-16 are addressed as 0-15.

The **register data** in the response message are **packed as two bytes per register**, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

Modbus Data Format

(0x03) Read Holding Registers

Request

Function Code	1 Byte	0x03
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D0)

Response

Function Code	1 Byte	0x03
Byte Count	1 Byte	2 x N*
Register Value	N* x 2 Bytes	

***N=Quantity of Registers**

Example of a request to read registers 108 – 110

Request		Response	
Field Name	Hex	**Field name**	Hex
Function	03	**Function**	03
Starting Address Hi	00	**Byte Count**	06
Starting Address Lo	6B	**Register value Hi (108)** **Register value Lo (108)**	02 2B
No. of Registers Hi	00	**Register value Hi (109)** **Register value Lo (109)**	00 00
No. of Registers Lo	03	**Register value Hi (110)** **Register value Lo (110)**	00 64

Modbus Data Format

(0x04) Read Input Registers

This function code is used to read from **1 to 125 contiguous input registers** in a remote device. The **Request PDU** specifies the **starting register address and the number of registers**. In the **PDU Registers** are **addressed starting at zero**. Therefore input registers numbered 1-16 are addressed as 0-15.

The **register data** in the response message are **packed as two bytes per register**, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

Request

Function Code	1 Byte	0x04
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	0x0001 to 0x007D

Response

Function Code	1 Byte	0x04
Byte Count	1 Byte	2 x N*
Input Registers	N* x 2 Bytes	***N=Quantity of Input Registers**

Example of a request to read input register 9

Request

Field Name	Hex
Function	04
Starting Address Hi	00
Starting Address Lo	08
Quantity of Input Reg. Hi	00
Quantity of Input Reg. Lo	01

Response

Field name	Hex
Function	04
Byte Count	02
Input Reg. 9 Hi	00
Input Reg. 9 Lo	0A

Modbus Data Format

(0x05) Write Single Coil

This **function code** is used to **write a single output to either ON or OFF** in a remote device.

The **requested ON/OFF state** is specified by a **constant in the request data field**. A value of **FF00 hex** requests the **output** to be **ON**. A value of **00 00 requests** it to be **OFF**. All other values are illegal and will not affect the output.

The **Request PDU** specifies the **address of the coil to be forced**. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0. The requested **ON/OFF state** is specified by a constant in the Coil Value field. A value of **0XFF00** requests the coil to be **ON**. A value of **0X0000** requests the coil to be off. All other values are illegal and will not affect the coil.

The **normal response** is an echo of the request, returned after the coil state has been written.

Request

Function Code	1 Byte	0x05
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 to 0xFF00

Response

Function Code	1 Byte	0x05
Byte Count	2 Bytes	0x0000 to 0xFFFF
Input Registers	2 Bytes	0x0000 or 0xFF00

***N=Quantity of Input Registers**

Example of a request to write Coil 173 ON

Request		**Response**	
Field Name	Hex	**Field Name**	Hex
Function	05	**Function**	05
Starting Address Hi	00	**Output Address Hi**	00

Modbus Data Format

(0x05) Write Single Coil

Example of a request to read input register 9

Request		Response	
Starting Address Lo	AC	**Output Address Lo**	AC
Quantity of Input Reg. Hi	FF	**Output Value Hi**	FF
Quantity of Input Reg. Lo	00	**Output Value Lo**	00

(0x06) Write Single Register

This **function code** is used to **write a single holding register** in a remote device.

The **Request PDU** specifies the **address of the register to be written**. **Registers** are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

The **normal response** is an **echo of the request**, returned after the register contents have been written.

Request

Function Code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Response

Function Code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 or 0xFF00

***N=Quantity of Registers**

Modbus Data Format

(0x06) Write Single Register

Example of a request to write register 2 to 00 03 hex

Request		Response	
Field Name	Hex	**Field Name**	Hex
Function	06	**Function**	06
Starting Address Hi	00	**Output Address Hi**	00
Starting Address Lo	01	**Output Address Lo**	01
Quantity of Input Reg. Hi	00	**Output Value Hi**	00
Quantity of Input Reg. Lo	03	**Output Value Lo**	03

(0x0F) Write Multiple Coils

This **function code** is used to **force each coil in a sequence of coils to either ON or OFF** in a remote device. The **Request PDU** specifies the **coil references to be forced**. Coils are **addressed starting at zero**. Therefore coil numbered 1 is addressed as 0.

The requested **ON/OFF** states are **specified by contents of the request data field**. A **logical '1'** in a bit position of the field requests the corresponding output to be ON. A **logical '0'** requests it to be OFF.

The **normal response** returns the **function code, starting address, and quantity of coils forced**.

Request

Function Code	1 Byte	0x0F
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0001 to 0x07B0
Byte Count	1 Byte	N*
Outputs Value	N* x 1 Byte	

***N = Quantity of Outputs / 8, if the remainder is different of 0 => N = N+1**

Modbus Data Format

(0x0F) Write Multiple Coils

Response

Function Code	1 Byte	0x0F
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0001 or 0x07B0

Example of a request to write register 2 to 00 03 hex

Request

Field Name	Hex
Function	0F
Starting Address Hi	00
Starting Address Lo	13
Quantity of Outputs Hi	00
Quantity of Outputs Lo	0A
Byte Count	02
Outputs Value Hi	CD
Outputs Value Lo	01

Response

Field Name	Hex
Function	0F
Starting Address Hi	00
Starting Address Lo	13
Quantity of Outputs Hi	00
Quantity of Outputs Lo	0A

(0x010) Write Multiple Registers

This **function code** is used to **write a block of contiguous registers (1 to 123 registers)** in a remote device.

The **requested written values** are specified in the request data field. **Data** is packed as **two bytes per register.**

The **normal response** returns the **function code, starting address, and quantity of registers written.**

Modbus Data Format

(0x010) Write Multiple Registers

Request

Function Code	1 Byte	0x10
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2 x N*
Registers Value	N* x 2 Bytes	Value

***N = Quantity of Registers**

Response

Function Code	1 Byte	0x10
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x123 or (0x7B)

Example of a request to write two registers starting at 2 to 00 0A and 01 02 hex

Request

Field Name	Hex
Function	10
Starting Address Hi	00
Starting Address Lo	01
Quantity of Registers Hi	00
Quantity of Registers Lo	02
Byte Count	04

Response

Field Name	Hex
Function	10
Starting Address Hi	00
Starting Address Lo	01
Quantity of Outputs Hi	00
Quantity of Outputs Lo	02

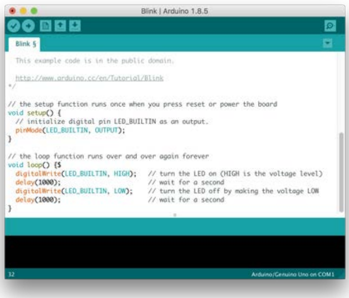
Modbus Data Format

(0x010) Write Multiple Registers

Example of a request to write two registers starting at 2 to 00 0A and 01 02 hex

Request	
Registers Value Hi	00
Registers Value Lo	0A
Registers Value Hi	01
Registers Value Lo	02

Response



Creating our Modbus RTU message

Now we already know a little bit more about Modbus RTU and its frame format, let's finish our Modbus message from the example we gave at the beginning.

We wanted the **master to send a message to slave number 2** requesting the value of 6 input registers.

Our **Modbus RTU message** looks like this at the moment: **0204 (02 (Slave Address) + 04 (Function Code))**

As our function code is number **04: Read Input Register**, the data must contain: **Starting Address Hi + Starting Address Lo + Quantity of Input Reg. Hi + Quantity of Input Reg. Lo + CRC**.

So, let's fill the **request ADU** in order to get all the messages:

Request ADU

Field Name	hex
Slave Address	02
Function Code	04
Starting Address Hi	00

Creating our Modbus RTU message

Request ADU	
Starting Address Lo	00
Quantity of Input Reg. Hi	00
Quantity of Input Reg. Lo	06
CRC	-
CRC	-

To **calculate the CRC**, just type the **Modbus message: 020400000006** on this website. Select HEX input type and get the **CRC-16 (Modbus) number**.

As it is **LSB**, we will reverse it. If the result of the **CRC is: 0x3B70**, now it will be **703B**.

Finally, this is how our Modbus message looks like:

020400000006703B

Software

Modbus RTU Master with Arduino IDE

The **Modbus RTU Master Module** implements the Modbus RTU Master capabilities. We are going to work with the **modbusrtumaster.h function**:

```
#include <ModbusRTUMaster.h>
```

There is the possibility to use **any hardware Serial Arduino** stream:

RS-485

```
#include <RS485.h>

ModbusRTUMaster master(RS485);
```

Software

RS-232

```
#include <RS232.h>

ModbusRTUMaster master(RS232);
```

Before using it, it is required to **call the begin function in the setup** for both the serial and the Modbus variable. It is a good practice to **set the baud rate (default: 19200 bps)** also in the Modbus variable to define the **Modbus internal timeouts**.

```
RS485.begin(9600, HALFDUPLEX, SERIAL_8E1);

master.begin(9600);
```

The functions to **read and write slave values** are:

```
readCoils(slave_address, address, quantity);
readDiscreteInputs(slave_address, address, quantity);
readHoldingRegisters(slave_address, address, quantity);
readInputRegisters(slave_address, address, quantity);
writeSingleCoil(slave_address, address, value);
writeSingleRegister(slave_address, address, value);
writeMultipleCoils(slave_address, address, values, quantity);
writeMultipleRegisters(slave_address, address, values, quantity);
```

Modbus RTU Master with Arduino IDE

Where:

- **`#include <ModbusRTUMaster.h>`** is the Modbus RTU slave address.
- **`address`** is the **coil, digital input, holding register or input register address**. Usually, this address is the coil, digital input, holding register or input register number **minus 1**: the holding register number **`40009`** has the address **`8`** .
- **`quantity`** is the **number** of **coils, digital, holding registers or input registers to read/write.**
- **`value`** is the given value of the coil or holding registers on a write operation. Depending on the function the data type changes. A coil is represented by a bool value and a holding register is represented by a **`uint16_t`** value.

On a **multiple read/write function** the **`address`** argument is the first address. On a **multiple write function**, the **`values`** argument is an **array of values to write**.

It is important to say that these functions are **non-blocking**, so they **do not return the read value**. They return **`true`** or **`false`** depending on the **current module state**. If there is a pending Modbus request, they return **`false`**.

Software

Modbus RTU Master with Arduino IDE

```
// Read 5 holding registers from address 0x24 of slave with address 0x10if
(master.readHoldingRegisters(0x10, 0x24, 5)) {
 // OK, the request is being processed
} else {
 // ERROR, the master is not in an IDLE state
}
```

There is the **function `available ()`** to check for responses from the **slave**:

```
ModbusResponse response = master.available();
if (response) {
 // Process response
}
```

The **`ModbusResponse`** implements some functions to get the response information:

```
hasError();
getErrorCode();
getSlave();
getFC();
isCoilSet(offset);
isDiscreteInputSet(offset);
isDiscreteSet(offset);
getRegister(offset);
ModbusResponse response = master.available();
if (response) {
 if (response.hasError()) {
  // There is an error. You can get the error code with response.getErrorCode()
 } else {
  // Response ready: print the read holding registers
  for (int i = 0; i < 5; ++i) {
   Serial.println(response.getRegister(i));
  }
 }
}
```

The possible error codes are:

```
0x01 ILLEGAL FUNCTION
0x02 ILLEGAL DATA ADDRESS
0x03 ILLEGAL DATA VALUE
0x04 SERVER DEVICE FAILURE
```

[Source]

[Check out for more]

How to convert a 4-20mA to 0-10V signal in a Arduino PLC

Signal converter on Arduino Analog inputs

ARDUINO

What is a 4-20mA Signal

What is 4-20mA standard?

A 4-20mA signal is an **Analog signal** that, **if connected to an Arduino board or an industrial Arduino-based PLC controller**, **identifies** the **value** from the **sensor.** The controller receives a current from the sensor, allowing the Arduino PLC to transform that **electrical signal to an understandable value,** ready for being used on the software code. This value is used for the programmer to know the right units of measure received from that sensor and understand the Industrial project's situation.

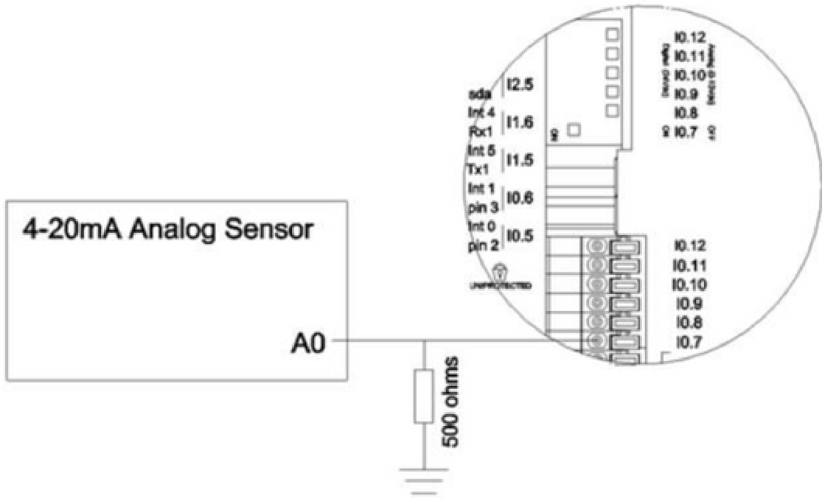
4-20mA is mainly used for **long-distance cables** because the **current signal** receives less interference from **noise or EMC issues** than a voltage signal such as 0-10Vdc Analog inputs.

Arduino is **not able to receive a current analog signal** because the Analog signals from Arduino work from **0 to 5Vdc**, so it is necessary to convert the current analog signal to a Voltage. It is important to know that Industry voltage Analog signal standards work from **0 to 10Vdc**, so the way to **transform a 4-20mA to Arduino** is quite different as transformed for Arduino-based PLC which works at 0-10Vdc.

Convert the Signal

It is necessary to convert a **4-20mA to a 0-10Vdc signal**. As you can see in the diagram of the next page, you can understand **how to transform a 4-20mA to 0-10v** using **Ohm Law**.

The product families Ardbox Analog and Ethernet industrial PLC have some **Analog / Digital configurable input signals.**



To do this it is necessary to connect an [impedance of 500 Ohm]() between the Analog 4-20mA signal and the ground signal as shown in the diagram. If you do, the **Analog value** will change from **current to Voltage.**

How can you make a 4-20mA to 0-10V signal conversion

Arduino PLC

For this reason, Industrial Shields has developed an Industrial Shield for Arduino which includes an original Arduino board inside, and it is fully compatible with Industrial Standards.

Converter module

If you need to test it using an Arduino Uno, Arduino Mega, or Arduino Leonardo Boards, you must replace the 500 Ohms to 250 Ohms to transform 4-20mA to 0-5Vdc.

Arduino IDE

You can use the Arduino IDE using the example "Analog Input" which allows the user to read the 10 bits Analog inputs from the Arduino to 0 – 1023 values.

Requirements

- [Ethernet]() or [20 I/Os]() PLC.
- **Industrial Shields boards.**

To learn how to install the boards, click [here]().

How does the input value change to Arduino

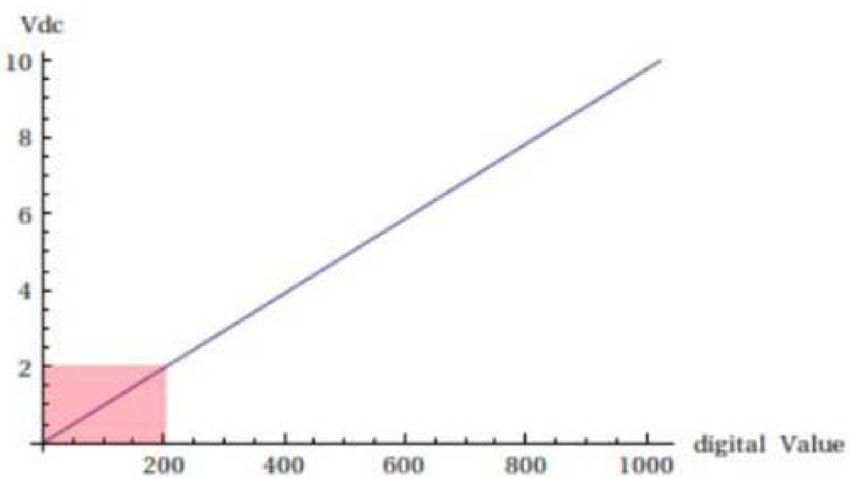
When you have a signal of 4-20mA for 10 bits, you get the following data:

4-20mA signal	10 bits
4mA	0 value
20mA	1023 value

But, if you convert this signal to 0-10Vdc, the results are:

4-20mA signal	0-10Vdc	10 bits
4mA	2Vdc	204 value* (approximately)
20mA	10Vdc	1023 value

As you can see the result of 4mA (2Vdc), is close to 204. Below is the linear graph where the results come from:



Communications and protocols used in industrial automation

Industrial network protocols in industrial automation

You are going to learn the main **industrial automation communications** of our Industrial Shields' PLC Arduino and the types of industrial networking protocols list they work with.

WiFi & Ethernet

Wi-Fi

In an industrial PLC controller Arduino, Wi-Fi uses multiple parts of the IEEE 802 protocol family and is designed to interwork seamlessly with its wired sibling Ethernet. Compatible devices can be networked via wireless access points to each other, as well as to wired devices and the Internet.

The different versions of Wi-Fi are specified by various **IEEE 802.11 protocol standards**, which is an industrial wireless network protocol, with the different radio technologies determining radio bands, and the maximum ranges, and speeds that may be achieved. Wi-Fi most commonly uses the **2.4 gigahertz (120 mm) UHF and 5 gigahertz (60 mm) SHF ISM radio bands**; these bands are subdivided into multiple channels. Channels can be shared between networks but only one transmitter can transmit locally on a channel at any time.

Ethernet

Ethernet is the most common technology working with the **Local Area Networks (LANs)** and **Wide Area Networks (WANs)**. **Ethernet communication** uses the LAN protocol which is technically known as the **IEEE 802.3 protocol**. This industrial network protocol has evolved and improved over time to transfer data at the speed of one gigabit per second.

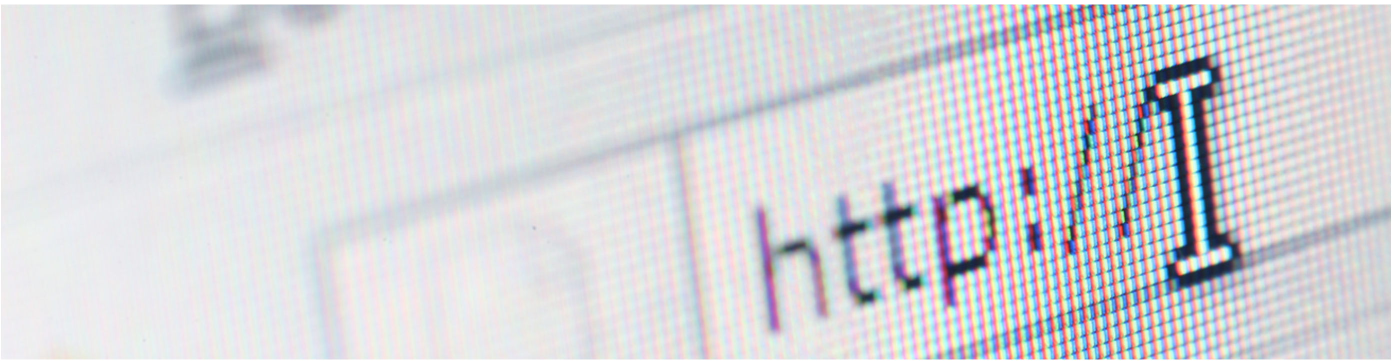
What is a 4-20mA Signal

Our M-Duino family PLCs incorporate the **W5500 IC** integrated circuit. The W5500 is a hardwired **TCP/IP** embedded Ethernet controller that provides an easier Internet connection to the embedded systems. This chip allows users to have Internet connectivity in their applications by using the single chip in which **TCP/IP stack**, **10/100 Ethernet MAC and PHY** are embedded. The W5500 chip incorporates the **32Kb** of internal memory buffer for processing Ethernet packets. With this chip, users can implement the Ethernet application by using Socket Programming. The **SPI** bus (Serial Peripheral Interface) is provided to facilitate the data transfer with the external microcontroller.

Ethernet uses **different** industrial **protocols** to communicate. Some of these are **HTTP, HTTPS, MQTT** and the **Modbus protocols.**

HTTP & HTTPS

HTTP stands for **Hypertext Transfer Protocol**. When you enter **http://** in your address bar in front of the domain, it tells the browser to connect via **HTTP. HTTP** uses **TCP (Transmission Control Protocol)**, usually, through **port 80**, to send and receive data packets over the web.

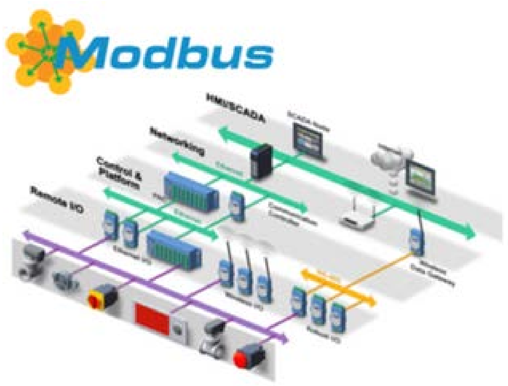


HTTPS stands for **Hypertext Transfer Protocol Secure (also known as HTTP over TLS or HTTP over SSL)**. When **https://** is entered in the address bar opposite the domain, it tells the browser to connect via HTTPS. Generally, sites that operate over HTTPS will have a redirect in place, so even if you type http:// it will be redirected to deliver over a secured connection. HTTPS also uses **TCP (Transmission Control Protocol)** to send and receive data packets, but it does so through **port 443**, within a connection encrypted by Transport Layer Security. (TLS).

MQTT

MQTT (Message Queuing Telemetry Transport) is an **open OASIS and ISO standard** (ISO/IEC 20922) lightweight, a publish-subscribe network protocol that transports messages between devices. This automation communication protocol usually **runs over TCP/IP**; however, any network protocol that provides orderly, lossless, bi-directional connections can support MQTT. It is designed for connections with remote locations where a **"small code footprint"** is required or the network bandwidth is limited.

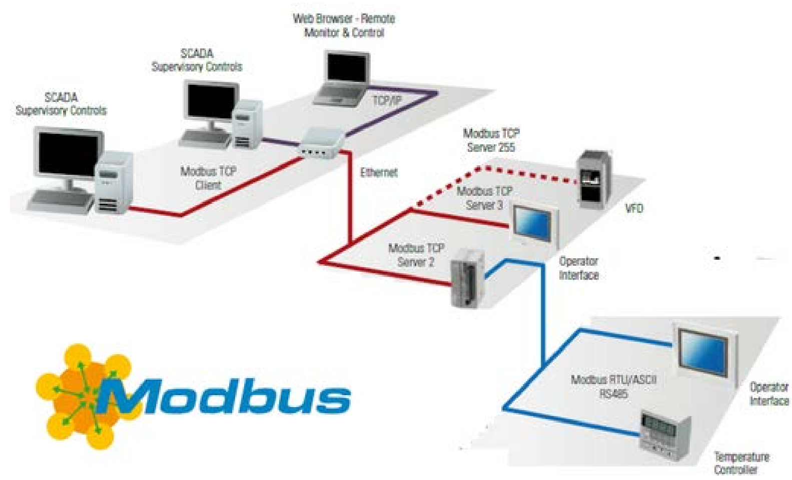
Modbus protocols



Modbus Protocol is a **messaging structure** developed by **Modicon**. It is used to establish **Master-Slave/Client-Server communication** between devices. Modbus has a lot of industrial automation protocol options. But the two most widely used are **Modbus RTU (Remote Terminal Unit)** and **Modbus (TCP/IP) Transmission Control Protocol**.

Modbus RTU

Modbus RTU mode is the most common implementation, but Modbus TCP/IP is gaining ground and is ready to overcome it. Modbus is an **open standard** and is a widely used industrial network protocol in the industrial manufacturing environment. It is a common link that has been implemented by hundreds of vendors for integration into thousands of different manufacturing devices to transfer **discrete/analog I/O** and record data between control devices. A Modbus communication is always initiated by the **master node to the slave node**. The slave nodes will **never transmit data without receiving a request** from the master node or communicating with each other. The master node initiates **only one MODBUS transaction** at the same time.



Modbus RTU mode is the most common implementation, using **binary coding** and **CRC error-checking**. RTU Protocol is an efficient binary protocol in which each **eight-bit (one byte)** in a message contains **two four-bit hexadecimal characters**. Each message must be transmitted in a continuous stream. The format for **each byte (11 bits)** in RTU mode is **Coding System: 8–bit binary, Bits per Byte: 1 start bit, 8 data bits, least significant bit sent first, 1 bit for parity completion, 1 stop bit**. Modbus RTU packets are only intended to send data; they do not have the capability to send parameters, such as point name, resolution, units, etc.

RTU is extremely popular for industrial control networks as it has been around for a long time, and there is a lot of hardware and software that support it.

Modbus protocols

Modbus TCP/IP

Modbus TCP/IP is basically the Modbus RTU protocol using the **TCP interface in an Ethernet network**. The Modbus data structure is defined using the application layer used in the TCP/IP protocol. The TCP, or transport protocol, is used to **ensure data is received and sent correctly**, and the IP is the **address and routing information**.

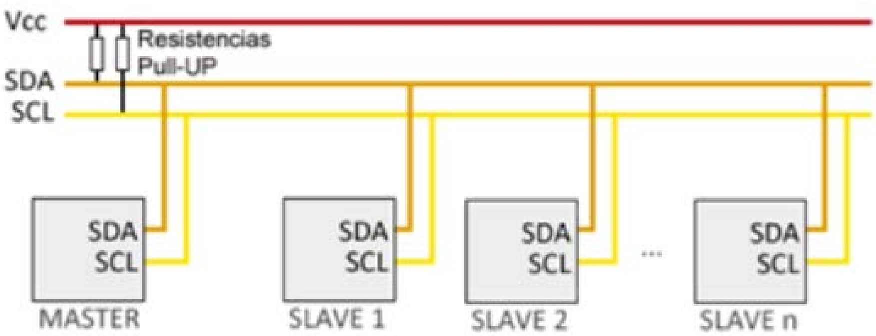
Essentially, the Modbus TCP/IP command is a Modbus RTU command included in an **Ethernet TCP/IP wrapper**. The benefit of using Modbus TCP/IP is using the existing **Ethernet network equipment** that is widely and available and cost-effective.

RS-232 & RS-485

RS-232

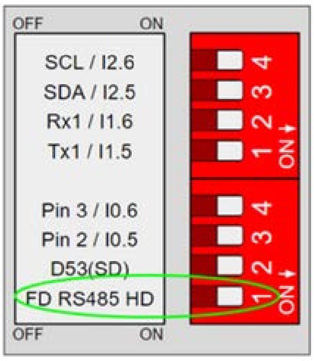
RS-232 (Recommended Standard 232) is a standard for serial communication transmission of data. It formally defines signals connecting between a **DTE (Data Terminal Equipment)** such as a computer terminal, and a **DCE (Data Circuit-Terminating Equipment or Data Communication Equipment)**, such as a modem. The standard defines the electrical characteristics and timing of signals, the meaning of signals, and the physical size and pinout of connectors. The current version of the standard is **TIA-232-F Interface** between a **DTE and a DCE** employing serial binary data interchange. The RS-232 standard had been commonly used in computer serial ports and is still widely used in industrial communication devices.

Our **Industrial Arduino Based PLCs** incorporate the integrated circuit **MAX232**. MAX232 converts signals from to **TIA-232 (RS-232)** serial port to signals suitable for use in **TTL**-compatible digital logic circuits. The MAX232 is a dual transmitter/dual receiver that is used to convert the **RX, TX, CTS, RTS signals**.



RS-232 & RS-485

RS-485



RS-485 Dip Switch configuration

RS-485, also known as **TIA/EIA-485**, is a standard defining the electrical characteristics of drivers and receivers for use in serial communications systems. Electrical signaling is balanced, and multipoint systems are supported. The standard is jointly published by the **Telecommunications Industry Association and Electronic Industries Alliance (TIA/EIA)**. Digital communications networks implementing the standard can be used effectively over **long distances** and in **electrically noisy environments**. Multiple receivers may be connected to such a network in a linear, multidrop bus. These characteristics make RS-485 useful in **industrial control systems** and similar applications.

Our Industrial Arduino-based PLCs include the integrated circuit **MAX485**. MAX485 is a low-power and slew-rate-limited transceiver used for **RS-485 communication**. It works at a single +5V power supply and the rated current is **300 µA**. Adopting Half-Duplex communication to implement the function of converting the **TTL level into RS-485 level**, can achieve a maximum transmission rate of **2.5Mbps**. MAX485 transceiver draws a supply current of between **120µA and 500µA** under the unloaded or fully loaded conditions when the driver is disabled.

GPRS

General Packet Radio Services (GPRS) is a packet-based industrial wireless communication service that promises data rates from **56 up to 114 Kbps** and continuous connection to the Internet for mobile phone and computer users. GPRS is based on **Global System for Mobile (GSM)** communication and complements existing services such as circuit-switched cellular phone connections and the **Short Message Service (SMS)**.

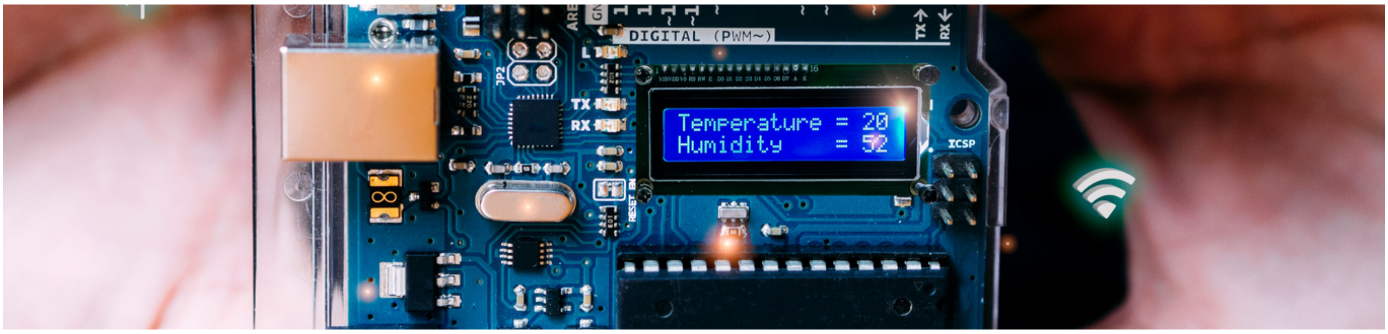
Equipment based on Arduino technology is designed for professional use. It also contains several communication ports which provide more flexibility and control. The GPRS/GSM family offers the possibility to **expand up to 127 modules** through I2C, which means that you can have **7100 Inputs / Outputs** in Master-Slave connections, additionally to sensors, etc.

The **Industrial Arduino-based PLCs with GPRS** are ideal for remote monitoring, data logging and remote access, diagnostics and control, using **short text messages (SMS)**. You can adjust the messages to send from a device with **static (text)** or **dynamic (text and values)** content.

Industrial Shields

What about Millis () vs Delay ()

Difference between Arduino Time functions



It is very common in industrial automation projects to program repetitive sequences in specific time intervals. Arduino programming language provides some time functions to control the Arduino board of your industrial PLC controller and perform computations to accomplish this.

The most common functions to work with time are **millis() and delay()**, **but what are the differences between them and which function is better to use in each case?** We will talk about them and we will see some examples of their use.

Requirements

To execute this program we will need the following:

- [Arduino board]
- **2 LEDs**
- **2 Resistors of 220 Ohms**
- **Wires**

Time functions of Millis () and Delay ()

Using [Arduino IDE] there are functions defined by default as the time functions such as Millis() and Delay(). They will allow you to control.

Millis() function

First of all, you need to know what the millis() function does. It will return the **number of milliseconds** that have passed since the PLC Arduino board started running the current program. It has a time limit of approximately **50 days.** After this time, it will **overflow** and **go back to zero**. If the program needs to run longer than this, an extra counter might be required. If a more precise amount of time is needed, there is a function called **micros()**, which has the same functionality as the millis() function but works with **microseconds**.

Delay () function

On the other hand, the delay() function will **stop the program** for a **specific amount of milliseconds** that you will have specified on the parameters. Although it is easy to use this function, it has the disadvantage that **any other action** can be performed **during its use**. If a more accurate amount of time is needed, there is a function called **DelayMicroseconds()**, which has the same functionality as Delay() but works with **microseconds.**

Delay () vs Millis ()

The first difference you can see is that **millis() has no parameter** but returns the amount of time that has passed; while the **delay() will require the number of milliseconds** we want to pause the program but will not return anything.

Even though both functions will work with milliseconds and could be used to **control time**, unlike millis(), the delay() is a **blocking function**. A blocking function is a function that **prevents a program** from doing anything else until that task has been completed. This means that by using delay() you **cannot execute any other tasks** during the time that you have specified.

Which function should I use?

Both functions can be used in most cases but sometimes one is better than the other. For example, if you want to **print a message every 5 seconds without any other conditions**, both can work perfectly:

Millis () example

```
 unsigned long time;
void setup() {
  Serial.begin(9600);
}
void loop() {
  time = millis();
  Serial.println("Hello World");
  while(millis() < time+5000);
}
```

Delay() example

```
 void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.println("Hello World");
  delay(5000);
}
```

In case you only want to **perform an action, delay()** is easier to implement as you can see on the above codes. You will only have to call the function, blocking the program for the specified time.

On the other hand, if you want to p**erform an operation** in which two actions must be executed simultaneously, the delay() function should not be used since the program will be blocked on its call. If this happens, both actions will be **stopped** for the amount of time specified in the function. In a complex program, this could cause a significant error that could spoil it.

In the following example, you can see **how to blink two LEDs** using different time intervals using the millis() function. Using the delay() function it will not be possible to do it simultaneously.

Blink LEDs using millis() function

```
int Led1,Led2;
int Period1, Period2;
unsigned long Time1, Time2;

void setup() {
    pinMode(13,OUTPUT);
    pinMode(12,OUTPUT);
    Time1 = Time2 = millis();
    Period1 = 1000;
    Period2 = 2000;
}

void loop() {
  if(millis()-Time1>=Period1){
    Led1=!Led1;
    digitalWrite(13,Led1);
    Time1=millis();
  }
  if(millis()-Time2>=Period2){
    Led2=!Led2;
    digitalWrite(12,Led2);
    Time2=millis();
  }
}
```

In conclusion, the **millis()** function is **better in general** and it is highly recommended to use **before the delay()** function.

It will allow us to **program** using **different threads** at the **same time** and is **more accurate**. The **delay()** is only recommended to be used in **simple programs** if a program **blocking action** is needed.

How to connect 7.5" E-Paper Display & ESP32

Display Applications using Arduino IDE

ARDUINO

E-Paper, or **electronic papers**, are display devices that **copy the appearance of ordinary ink on paper**. Unlike conventional flat panel displays that emit light, electronic paper **displays reflect light** like paper. This may make them more comfortable to read and provide a wider viewing angle than most light-emitting displays.

We are going to download the **GxEPD Library**, wire the **E-Paper with the** **ESP32**, and finally test some examples for industrial automation using the Arduino IDE!

Requirements

- **7.5 inch e-Paper Display (800×400 resolution, SPI interface, B/W)**
- **ESP32 devkit board**
- **WaveShare e-Paper adapter**
- **WaveShare e-Paper Driver HAT with connector for SPI interface**
- **USB micro to program ESP32**
- **Arduino IDE**

Explanation

GxEPD

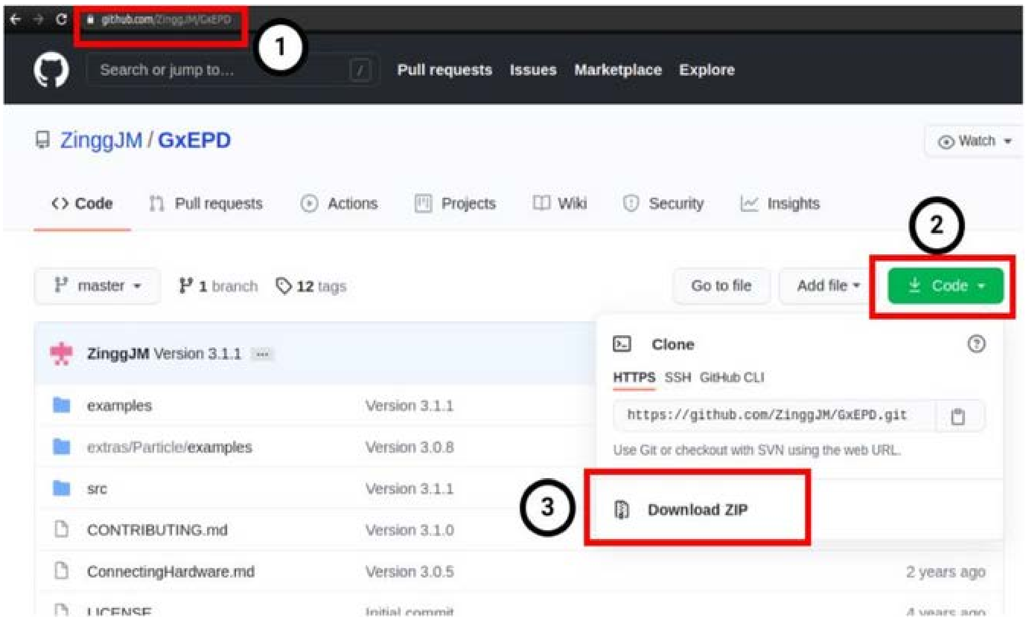
In this tutorial, we are going to download the GxEPD Library, whose author and maintainer is **Jean-Marc Zingg**. The GxEPD Library is a simple E-Paper display library with a common base class and a separate IO class for Arduino.

- **For SPI e-paper displays from Dalian Good Display**
- **SPI e-paper boards from Waveshare**
- **GxEPD2 is better suited for new users or new projects!**

Explanation

GxEPD

So, first of all, click [here](#) to get the Library for GitHub as follows:



Arduino IDE

After the Library is downloaded, there are at least **two ways** of installing the GxEPD Library.

1. Open the **Arduino IDE**, click on the **top menu Sketch >> Include Library >> Add .ZIP Library...** and **import** the **GxEPD Library** you just downloaded.

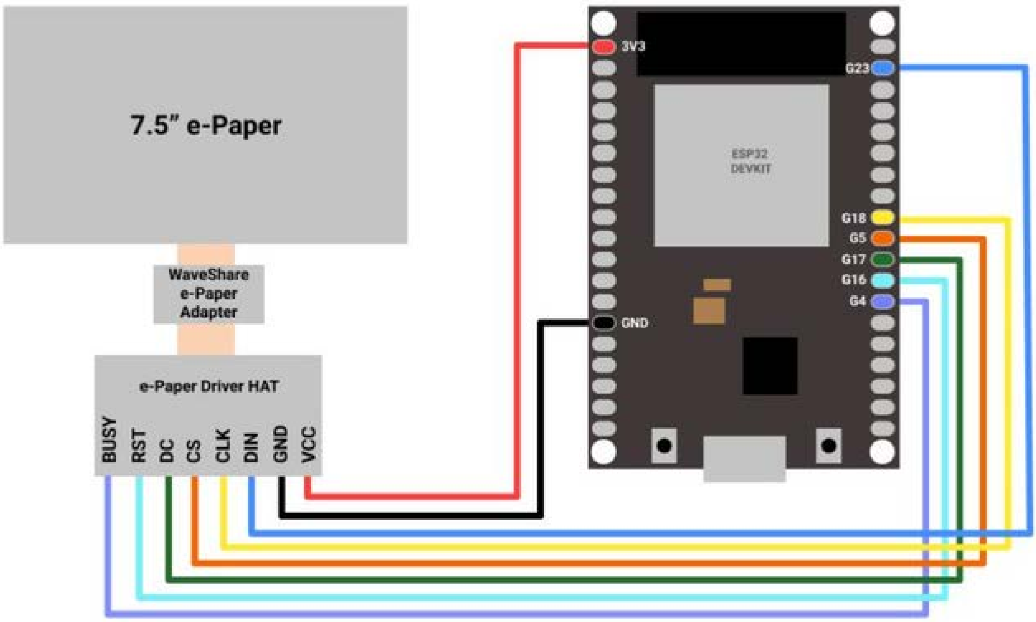
2. Open the **Arduino IDE**, go to **Sketch >> Include Library >> Manage Libraries >> type "GxEPD" >> Install.**

Once the Library is installed, you will be able to test some examples from the Library for industrial control!

If we go to **File >> Examples >> GxEPD** and click on the first GxEPD-Example, a new window will be displayed. If you take a look at the example, you will see that there is a different mapping for some hardware. Just find out your board and wire it to your e-Paper from WaveShare. In our case, we are going to do it like this line of the example:

```
// mapping suggestion for ESP32, e.g. LOLIN32, see .../variants/.../pins_arduino.h for your board
// NOTE: there are variants with different pins for SPI ! CHECK SPI PINS OF YOUR BOARD
// BUSY -> 4, RST -> 16, DC -> 17, CS -> SS(5), CLK -> SCK(18), DIN -> MOSI(23), GND -> GND, 3.3V -> 3.3V
```

7.5" e-Paper and ESP32 pinout



7.5inch e-Paper Display

Under the mapping section of the example in the Arduino IDE, you will see an include section. You must **uncomment** the one that fits your e-Paper screen. In our case, we are going to remove the // to uncomment the ***include***:

```
#include <GxGDEW075T7/GxGDEW075T7.h>      // 7.5" b/w 800x480
```

After that, you will be ready to test the example!

Go to **Tools >> Board >> Select the ESP32.**

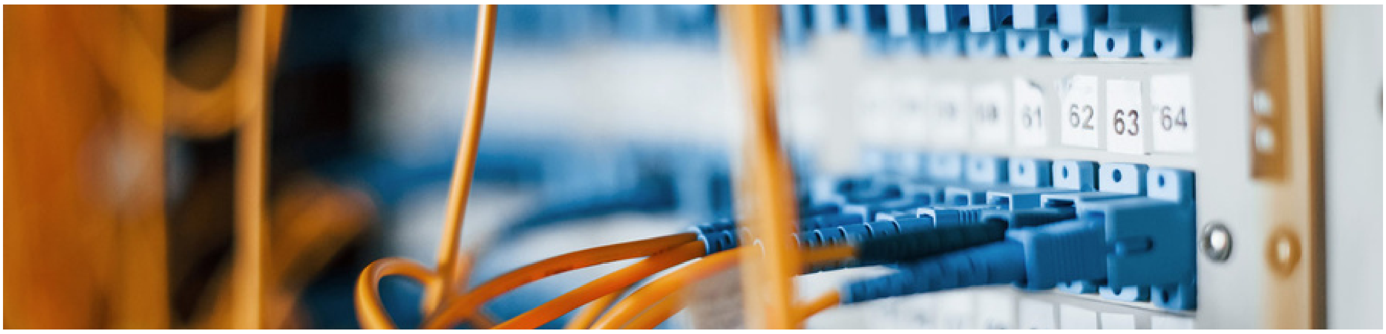
And **Tools >> Port >> Select the port where you connect your ESP32.**

Upload your sketch and enjoy the testing!

Watch the **Video** in our Youtube Channel

How to use Modbus TCP Slave library with an Arduino PLC

ARDUINO



We will introduce you to our libraries to be able to implement the **Modbus TCP/IP Slave mode** when working with programmable logic controllers. Basically, it works in the same way as the Modbus RTU with some exceptions.

- **TCP - Transmission Control Protocol.**
- **IP - Internet Protocol.**

This is a Modbus variant used for communications over **TCP/IP networks** on industrial controllers for Arduino automation, connecting over **port 502**. It does not require a checksum calculation, as lower layers already provide checksum protection.

Previous readings

To be able to follow easily the explanation of our Modbus TCP libraries for industrial PLC Arduino, first of all, it would be interesting to have a look at this previous section about Modbus RTU to understand better the **characteristics and configurations** of this type of communication on industrial Arduino devices.

Technical details

Modbus TCP encapsulates **Modbus RTU request** and **response data packets** in a TCP packet transmitted over standard **Ethernet networks**. The unit number is still included and its interpretation varies by application - the unit or slave address is not the primary means of addressing in TCP. The most important address here is the **IP address**. As we said before, the standard port for Modbus TCP is **502**, but the port number can often be reassigned if you wish.

The checksum field normally found at the end of an RTU packet is omitted from the TCP packet. Checksum and error handling is handled by Ethernet in the case of Modbus TCP.

The TCP version of Modbus follows the **OSI Network Reference Model**. Modbus TCP defines the presentation and application layers in the **OSI model**. This type of transmission makes the definition of **master and slaveless** obvious because Ethernet allows **peer-to-peer communication**.

The meaning of client and server are better-known entities in Ethernet-based networking. In this context, the slave becomes the **server** and the master becomes the **client**.

There can be more than one client obtaining data from the server. In other words, we can say that this means there can be **multiple masters** as well as **multiple slaves**. Instead of defining master and slave on a physical device-by-device basis, the designer will have to create logical associations between master and slave to define the roles.

Software

Modbus TCP Slave with Arduino IDE

The Modbus TCP Slave module implements the Modbus TCP Slave functionality.

```
#include <ModbusTCPSlave.h>

ModbusTCPSlave slave;
```

The default TCP port is the **502** but you can change it with:

```
// Set the TCP listening port to 510 instead of 502
ModbusTCPSlave slave(510);
```

To map the coils, discrete inputs, holding registers and input registers addresses with the desired variables values, the module uses four variables arrays:

```
bool coils[NUM_COILS];
bool discreteInputs[NUM_DISCRETE_INPUTS];
uint16_t holdingRegistesr[NUM_HOLDING_REGISTERS];
uint16_t inputRegisters[NUM_INPUT_REGISTERS];
```

The lengths of these arrays depend on the application and the register's usage. Obviously, the names of the arrays also depend on your preferences.

To associate the registers arrays with the library, it is possible to use their functions in the **setup**:

```
slave.setCoils(coils, NUM_COILS);
slave.setDiscreteInputs(discreteInputs, NUM_DISCRETE_INPUTS);
slave.setHoldingRegisters(holdingRegisters, NUM_HOLDING_REGISTERS);
slave.setInputRegisters(inputRegisters, NUM_INPUT_REGISTERS);
```

Modbus TCP Slave with Arduino IDE

It is not required to have all kinds of registers mapping to work, only the ones used by the application.

To start Modbus TCP server, call the **begin** function after the registers mapping. It is also possible to call the **begin** function before the registers mapping. Remember to begin the Ethernet before the Modbus TCP Slave object in the **setup**.

```
// Init the Ethernet
Ethernet.begin(mac, ip);

// Init the ModbusTCPSlave object
slave.begin();
```

At this time the Modbus TCP server is running and the only important thing to do is to update the Modbus TCP Slave object often in the lllll function and treat the registers mapping values to update variables, inputs and outputs.

```
// Update discrete inputs and input registers values
discreteInputs[0] = digitalRead(I0_7);
inputRegisters[0] = analogRead(I0_0);
// ...

// Update the ModbusTCPSlave object
slave.update();

// Update coils and holding registers
digitalWrite(Q0_0, coils[0]);
// ...
```

Modbus RTU and RS485 Arduino (Seneca Z-D-in Module)

Communication with RS485 Modbus RTU in Arduino industrial PLC controllers

The Modbus RTU protocol is a means of communication that allows the **exchange of data between programmable logic controllers (PLCs) and computers**. Electronic devices can exchange information through serial lines using the Modbus protocol.

It has been widely accepted and used in the construction of **Building Management Systems (BMS)** and **Industrial Automation Systems (IAS)**. Its adoption has been driven by its ease of use, reliability, and the fact that it is open source and can be used without royalties on any device or application.

The Modbus protocol was developed and published by **Modicon®** in 1979 for use with its programmable logic controllers. It is built using a **master/slave architecture** and is compatible with serial devices that use the **RS232 / RS485 / RS422 Arduino protocols**. Modbus is often used in scenarios where multiple control and instrumentation devices transmit signals to a controller or central system to collect and analyze data. The automation and monitoring control and data acquisition **(SCADA)** systems often use the Modbus protocol.

Requirements

- Ethernet PLC **or** 20 I/Os **industrial Arduino**
- Z-D-IN 5 Modbus RTU module
- Industrial Shields boards
- Tools40 library installed (include Modbus libraries)

Description

Z-D-In 5 Modbus RTU Module

The Z-D-IN Seneca Modbus RTU module is used to interface **5 digital signals (PUSH BUTTON, LIMIT SWITCH, REMOTE, CONTROL SWITCH, RELAYS, etc.)** with all of the control systems which are able to communicate by the protocol MODBUS RTU through the RS485 serial interface.

A **16 bit counter** is available on **each input**, the **maximum frequency input is 100 Hz** and it is possible to **set one input** as a fast counter with a **maximum frequency of 10kHz**. The module is a Modbus RTU Arduino slave and can be coupled with any Modbus Master device. **3-way galvanic isolation** among **Power supply // input // RS485 circuits** assures the integrity of your data.

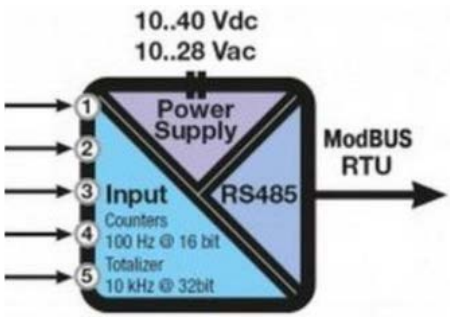
Characteristics

Power Supply	10..40 Vdc; 19..28 Vac (50-60 Hz)
Inputs	5 CH (reed, proximity, pnp, npn, contact) with common negative, self powered 24 Vdc, isolated, protected from transient up to 600 W/ms
Counters	4 @ 16 bit, max frequency 100 Hz; 1 @ 32 bit, max frequency 10 KHz
Anti rebounce filter	Settable from 5 to 250 ms
Communication	RS485 a 2 fili, Modbus RTU slave protocol
Dimesions	17,5 x 100 x 112 (mm)
Mounting	35 mm guide DIN 46277

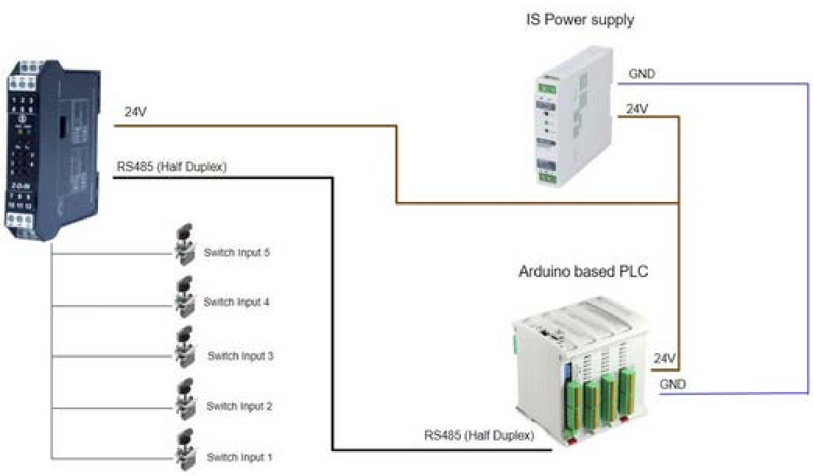
Connections

Initial connection

The following image shows the **inputs and outputs** of the Z-D-IN:



Final connection



Software

In this sketch, we are controlling **each input (5 in total)** using **digital switches**. The communication between the Arduino PLC and the Z-D-in Modbus RTU module is by **RS485 in Half Duplex**, so it's very important that you download and use the **RS485.h Library** as well as the **ModbusRTUSlave.h (Modbus RTU Arduino library)** library to work on this protocol.

The Z-D-in module acts as a slave and the Arduino controller will act as the master of the system.

In this sketch, requests for reading the entries are sent **every second** and the changes are shown on the screen. Each entry has associated a counter that will increment for each change that can be read in the entry.

The full sketch is shown below:

```
/*
   Copyright (c) 2019 Boot&Work Corp., S.L. All rights reserved
   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU Lesser General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU Lesser General Public License for more details.
   You should have received a copy of the GNU Lesser General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
#include <ModbusRTUMaster.h>
#include <RS485.h>
const uint8_t slaveAddress = 1;
const uint32_t serialRate = 38400UL;
const int serialConfig = SERIAL_8N1;
const uint32_t requestPeriod = 1000UL; // ms
const int numInputs = 5;
ModbusRTUMaster modbus(RS485);
uint8_t inputStates[numInputs];
uint16_t inputCounters[numInputs];
/////////////////////////////////////////////////////////////////////////////////
/////////////////
void setup() {
  Serial.begin(9600L);
  Serial.println("seneca-z-d-in-module started");
  RS485.begin(serialRate, HALFDUPLEX, serialConfig);
  modbus.begin(serialRate);
}
/////////////////////////////////////////////////////////////////////////////////
/////////////////
```

Software

```
void loop() {
 static uint32_t lastRequestTime = millis();
 // Send a request every 1000ms
 if (!modbus.isWaitingResponse()) {
 if (millis() - lastRequestTime > requestPeriod) {
 // Send a Read Holding Registers request to the slave with address 1
 // IMPORTANT: all read and write functions start a Modbus transmission, but they
are not
 // blocking, so you can continue the program while the Modbus functions work. To
check for
 // available responses, call modbus.available() function often.
 if (!modbus.readHoldingRegisters(slaveAddress, 1, 6)) {
 // TODO Failure treatment
 }
 lastRequestTime = millis();
 }
 }
 // Check available responses often
 ModbusResponse response = modbus.available();
 if (response) {
 if (response.hasError()) {
 // TODO Response failure treatment. You can use response.getErrorCode()
 // to get the error code.
 } else {
 uint16_t states = response.getRegister(0);
 for (int i = 0; i < numInputs; ++i) {
 inputStates[i] = (states >> i) & 0x01;
 inputCounters[i] = response.getRegister(i + 1);
 }
 printInputs();
 }
 }
}
/////////////////////////////////////////////////////////////////////////////////
/////////////////
void printInputs() {
 Serial.println();
 Serial.print("Inputs: ");
 for (int i = 0; i < numInputs; ++i) {
 Serial.print(inputStates[i] ? "HIGH" : "LOW ");
 Serial.print(' ');
 }
 Serial.println();
 Serial.print("Counters: ");
 for (int i = 0; i < numInputs; ++i) {
 Serial.print(inputCounters[i]);
 Serial.print(' ');
 }
 Serial.println();
}
```

How to Use the Software Serial library in Arduino PLC industrial controller

Arduino Software Serial example

ARDUINO

What is Serial communication with Arduino? How is the Library used in the Arduino IDE?

There are different types of Serial Communications. When you use an Arduino board on a project you can choose the standard Serial pins as Arduino software serial **Rx Tx, from the UART** inside the Arduino board, so it is called Serial TTL. In that case, you will use the Hardwareserial.h Library, but some additional pins can work as a Rx or Tx. For example, the **SPI communication** pins can work as a **MISO, MOSI and Select (SC)**, but they are also pins that **can work as a digital input or digital output**, or if you need, you could use those pins **as Rx, Tx** using the softwareserial.h Library.

Serial communications allow you to connect two different devices sending and receiving data between them.

The **Serial TTL** port can be transformed as required on Industry **as an RS232 and as an RS485**. When you use RS232 the functionality is quite similar to working as Serial TTL but if you work using RS485 you can configure a network using a Master device that can connect with Slave devices. So the number of devices has been increased from 2 to 32 devices (nodes). And the max distance between them can be up to 1220m if the wiring is well done and in compliance with EMC and the electrical noise is avoided.

To sum up, if you are using the UART serial port from the Arduino or the Arduino-based PLC for Industrial projects, the use of other pins working as a Serial TTL can help you with the success of the development of your project. So, that additional serial port must be programmed using **softwareserial.h** library.

If the **HardwareSerial** Library can not be used, because you need to use a communication protocol that needs the use of a physical UART instead of a virtual serial port. Then, you could convert the standard RS232 or RS485 from the device to a **Serial TTL**.

This section explains the advantage of the SoftwareSerial Library to simulate a serial port through Software (virtual serial TTL) using the Arduino IDE.

Serial ports on Arduino PLCs

Arduino Uno includes **1 serial TTL port** from pin 0 and pin 1. It works from UART, but there is the possibility to use it as a **TTL with other pins**. For example, all digital pins can work as a Tx, but the SPI communication pins can work as an Rx, so you could be able to add at least 3 additional SoftwareSerial ports. See on technical specifications from **Arduino UNO** that Rx and Tx are shared with a USB port, so if you need to use Rx and Tx from UART the USB port can not be used. Other characteristics affect the use of Arduino UNO so we recommend the use of **Arduino Leonardo and Arduino Mega** if you need to use **both ports**, USB and Serial TTL.

If you see Arduino Mega, it includes **4 UART serial TTL ports** and you can also use other pins as Rx. Using Arduino Leonard the UART is not shared from the USB so you could use USB and Serial TTL to work together

Anyway, we **don't recommend** the use of the **USB port** to **send and receive data** because it is designed just to program the Arduino board, and sometimes the configuration of the UART chip can be misconfigured if the volume and/or the velocity of the communication is quite high. So, for safety reasons on Industrial projects, **we recommend using RS232** instead of USB.

Using an Arduino Industrial controller

Using an Arduino Industrial Shield, the standard configuration has already been done. So you can connect **directly to RS232 and RS485 communication ports** but only one of them can run under hardware serial, so, the other must be configurated as SoftwareSerial.

You need to **configure** the **internal jumpers** to define which port should work as hardware serial and which one should work as SoftwareSerial, because the PLC allows you to configure which one can run as a hardware serial.

Example

Requirements

- Ethernet or 20 I/Os PLC
- SoftwareSerial Library

Example

Configuration

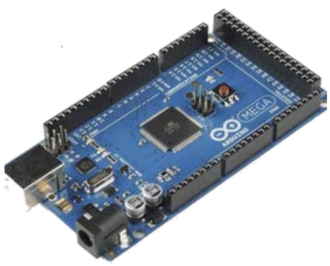
Arduino Leonardo

All pins can be used for Tx on the Arduino LEONARDO, while for Rx the following can be used:

- **Pin 14 of Leonardo (SO)**
- **Pin 15 of Leonardo (SCK)**
- **Pin 16 of Leonardo(SI)**

We recommend using one of the Rx-enabled pins for Tx.

Arduino Mega



On the Arduino MEGA, all pins can be used for Tx, while for the Rx the following can be used:

- **Pin 50 of the Arduino Mega 2560 (SO)**
- **Pin 51 of the Arduino Mega (SI)**
- **Pin 52 of the Arduino Mega (SCK)**

We recommend using one of the Rx-enabled pins for Tx. Using an Arduino board (this example has been done using Arduino Leonardo, Arduino Uno and Arduino Mega. If you need to use other Arduino boards, you can check the technical specification from that board).

Software

This sketch is very simple and shows how the library works. First, the **Tx and Rx pins** of the equipment for the **SoftwareSerial** must be defined (we are trying it on an [Ardbox Family model]). See the SoftwareSerial source code:

```
SoftwareSerial mySerial(14, 15); // RX, TX
```

Rx is digital pin 14 (SO), connected to TX of the other device.
Tx is digital pin 15 (SCK), connected to RX of the other device.

Software

Receives from the HardwareSerial, send to SoftwareSerial.
Receives from SoftwareSerial, send to HardwareSerial.

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(14, 15); // RX (MISO), TX (SCK)

void setup()
{
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }


  Serial.println("Goodnight moon!");

  // set the data rate for the SoftwareSerial port

  mySerial.begin(9600);
  mySerial.println("Hello, world?");
}

void loop() // run over and over

{
  if (mySerial.available())

    Serial.write(mySerial.read());

  if (Serial.available())

    mySerial.write(Serial.read());
}
```

This other sketch **sends instructions through the SoftwareSerial**:

```
#include <SoftwareSerial.h>
SoftwareSerial mySerial(14, 15);  // Rx (MISO, Tx (MOSI)
void setup() {
  // put your setup code here, to run once:
mySerial.begin(9600);
Serial.begin(9600);
}
void loop() {
  // put your main code here, to run repeatedly:
delay(500);
mySerial.println("Instruction 1");
Serial.println("1st instruction sended");
delay(500);
mySerial.println("Instruction 2");
Serial.println("2nd instruction sended");
}
```

How to send and receive SMS with Raspberry Pi automation

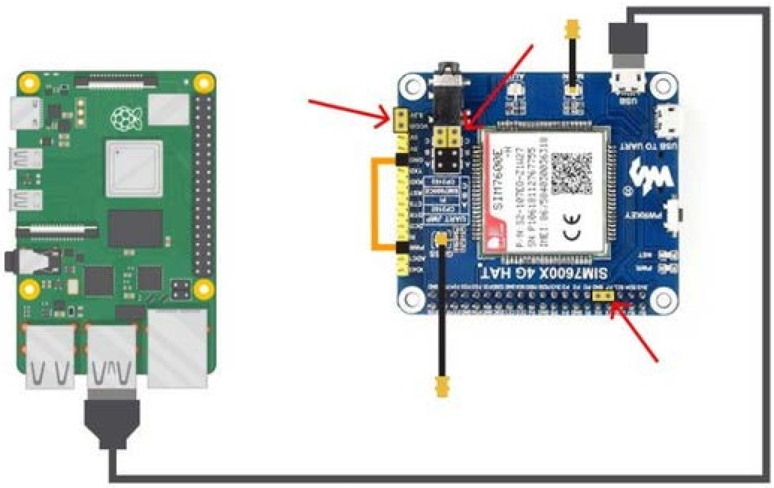
Use Raspberry Pi and the modules SIM7600E 4G HAT to send and receive SMS between GSM module and mobile phone

You will know more about **SIM7600E-H 4G HAT for Raspberry Pi**, LTE Cat-4 4G / 3G / 2G, GNSS, for Europe, Southeast Asia, West Asia and Africa, and you are going to test it.

For the testing it, you are going to send SMS **from your Toucherry Pi 10" Panel PC** monitoring for industrial automation with the SIM7600E 4G Hat module to your mobile phone.

Connections

In this case, you can see a Raspberry Pi 4 connected to the SIM7600E-H 4G Hat module through the USB port as shown:



SIM7600E-H 4G HAT

GPRS (General Packet Radio Service) is an extension of GSM based on packet transmission that offers a more efficient service for data communications, especially in the case of Internet access.

GSM (Global System for global Communications) is the communication system most widely used in mobile phones, and its first functionality is voice transmission. However, it also allows data transmission like SMS or internet, at a very low speed.

The **SIM7600E-H 4G HAT** is **multi-band LTE-TDD/LTE-FDD/HSPA+** and **GSM/GPRS/EDGE** module solution in an SMT type which supports LTE CAT1 up to 10Mbps for downlink data transfer. It has strong extension capability with rich interfaces including UART, USB2.0, I2C, GPIO etc. With abundant application capability like TCP/UDP/FTP/FTPS/HTTP/HTTPS/DNS, the module provides much flexibility and ease of integration for customer's applications, such as open source projects.

Sending SMS

The **AT+CMGF command** is used to instruct the **GSM / GPRS modem** to operate in **SMS text mode**. So, insert your SIM Card and into the SIM7600X board and let's start!

First, ensure that the hardware is right connected. So, connect open up a terminal window and type "sudo raspi-config" > Go to Interface Options > Serial Port > Reject the login shell to be accessible over serial > Accept the serial port hardware to be enabled.

Now, **open the Serial Port** by installing the screening tool:

```
sudo apt update
sudo apt install screen
screen /dev/ttyUSB2 115200 (you can use USB0, USB1 or USB2)
```

Once in the Serial Port, let's send some **AT commands:**

AT	*The device recognizes the SIM7600E module*
OK	*Response*
AT+CFUN=1	*Set phone functionality*
OK	*Response*
AT+CMGF=1	*Set the GSM modem in SMS Text Mode*
OK	*Response*
AT+CMGS="+34666XXXX66"	<-- *Add a phone number*
> *Type here your text message*	*<-- (Do not finish it with an Enter, but with **Ctrl + Z**)*
OK	*Response*

Receiving SMS

The AT + CMGL command lists messages received on the GSM / GPRS modem. It can be used to get all received messages, all unread messages or all read messages.

Finally, let's set the GSM modem to Text Mode SMS and read all received messages:

AT+CMGF=1	*Set the GSM modem in SMS Text Mode*
OK	*Response*
AT+CMGL="ALL"	<-- *List all received messages*

+ CMGL: 0, "REC READ", "+346XXXXXXX6", "", "21/05/21,12:32:09+08"

HIGH Temperature

OK	*Response*

How to communicate Raspberry Pi 3 B+ with a MAX485 module

Learn how to connect a Raspberry Pi 3 B+ with an M-DUINO by RS485

Raspberry Pi

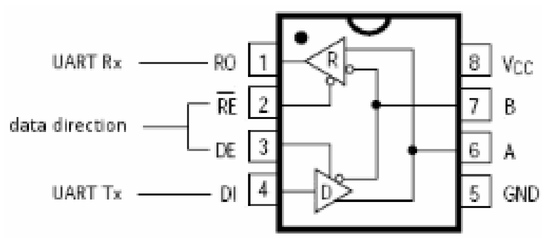
Requirements for communicating Raspberry Pi 4 B+ with a MAX485 module

- **M-DUINO/ Ardbox Family**
- **Touchberry Pi 10.1" w/ UPS & RTC & RS485**
- **MAX485 module**

MAX485 Module

On-board MAX485 chip is a low-power and slew-rate-limited transceiver used for RS485 communication. By adopting half-duplex communication to implement the function of converting TTL level information into RS485 level, it can reach a maximum transmission rate of 2.5 Mbps.

Pinout



Depending on RE and DE connection, the module works as a receiver or transmitter. Connected to VCC, it transmits data and connected to GND it receives data.

It is a cheap module. Batches of 5 units can be found for less than 1€.

Connections

Raspberry Pi to MAX485

Raspberry Pi 3 B+ Pins	MAX485 Module Pins
UART_TXD	RX
UART_RXD	TX
GPIO 17	RE
GPIO 27	D

MAX485 to M-Duino/Ardbox

MAX485 Module Pins	M-Duino / Ardbox Pin
VCC	3.3 V
B	B-
A	A+
GND	GND

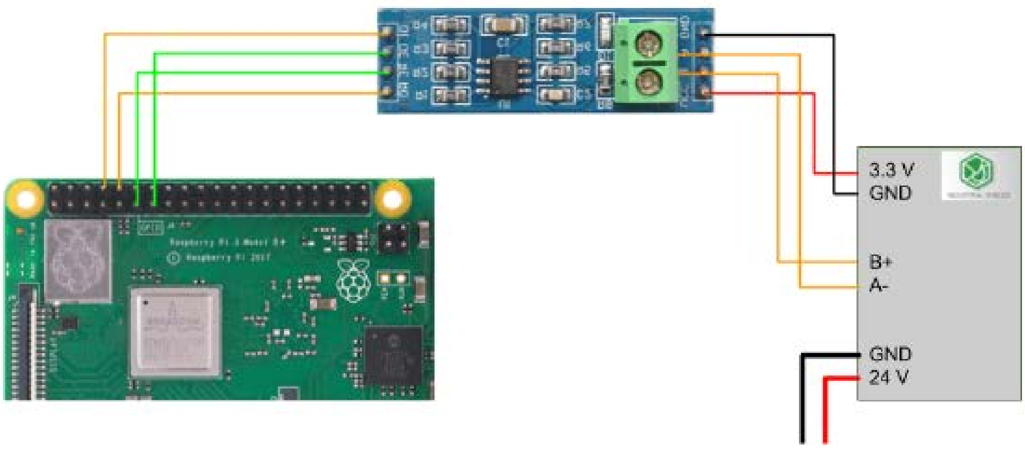
Connections

The GPIO 17/27 digital pins are used to establish the type of communication. Receiver or transmitter mode.

Raspberry Pi 4 B+ Pinout

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I2C)	DC Power 5v	04
05	GPIO03 (SCL1 , I2C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)	(I2C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Diagram



Python Code Example

In this example of code, we **send a character from the PLC to the Raspberry**, print it, and answer to the PLC with the same character, which is an "echo". Note that in the case of receiving data, pins 17 and 27 are deactivated and activated in case of transmitting data.

```
#!/usr/bin/env python3

' IMPORTANT: remember to add "enable_uart=1" line to /boot/config.txt

from gpiozero import OutputDevice
from time import sleep
from serial import Serial

' RO  <-> GPIO15/RXD
' RE  <-> GPIO17
' DE  <-> GPIO27
' DI  <-> GPIO14/TXD

' VCC <-> 3.3V
' B   <-> RS-485 B
' A   <-> RS-485 A
' GND <-> GND

re = OutputDevice(17)
de = OutputDevice(27)

' enable reception mode
de.off()
re.off()

with Serial('/dev/ttyS0', 19200) as s:
 while True:
  ' waits for a single character
  rx = s.read(1)

  ' print the received character
  print("RX: {0}".format(rx))

  ' wait some time before echoing
  sleep(0.1)

  ' enable transmission mode
  de.on()
  re.on()

  ' echo the received character
  s.write(rx)
  s.flush()

  ' disable transmission mode
  de.off()
  re.off()
```

Node-RED tutorial: How to get GPS coordinates with a Maps Widget

Learn how to get GPS coordinates from your SIM7600E module and put them into a Google Maps Widget in Node-RED Dashboard.

Raspberry Pi

Node-RED is a programming tool to wire together hardware devices, APIs and online services in new and interesting ways.

It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

As it is quite interesting to use for open source projects, you are going to learn how to get GPS coordinates from your **SIM7600E** module and put them in a **Google Maps Widget** in Node-RED Dashboard.

Explanation

You will learn how to get GPS coordinates with a Map Widget from Node-RED. Let's continue the post below:

[How to get GPS location on Panel PC]

Installing and Upgrading Node-RED

In order to start your Node-RED application, you must **check** if you already have N**ode-RED installed** in your Industrial Panel PC. If you do not have it already installed, **run the following command** to download and run the script to install Node.js, npm and Node-RED onto a Raspberry Pi.

```
sudo apt install build-essential git

bash <(curl -sL https://raw.githubusercontent.com/node-red/linux-installers/master/deb/update-nodejs-and-nodered)
```

Python Code Example

Installing and Upgrading Node-RED

The script above will:

- Remove the **pre-packaged version** of Node-RED and Node.js if present.
- Install the **current Node.js LTS** release using the NodeSource. If it detects Node.js is already installed from NodeSource, it will ensure it is at least Node 8, but otherwise it will leave it alone.
- Install the **latest version** of Node-RED using **npm**.
- Optionally, **install** a **collection** of useful **Pi-specific nodes.**
- **Setup Node-RED** to run as a service and provide a set of commands to work with the service.

Autostart on boot

If you want Node-RED to run when the Pi is turned on, or re-booted, you can enable the service to autostart by running the command:

```
sudo systemctl enable nodered.service
```

To disable the service, run the command:

```
sudo systemctl disable nodered.service
```

To know more about Node-RED installation, check out the following link:

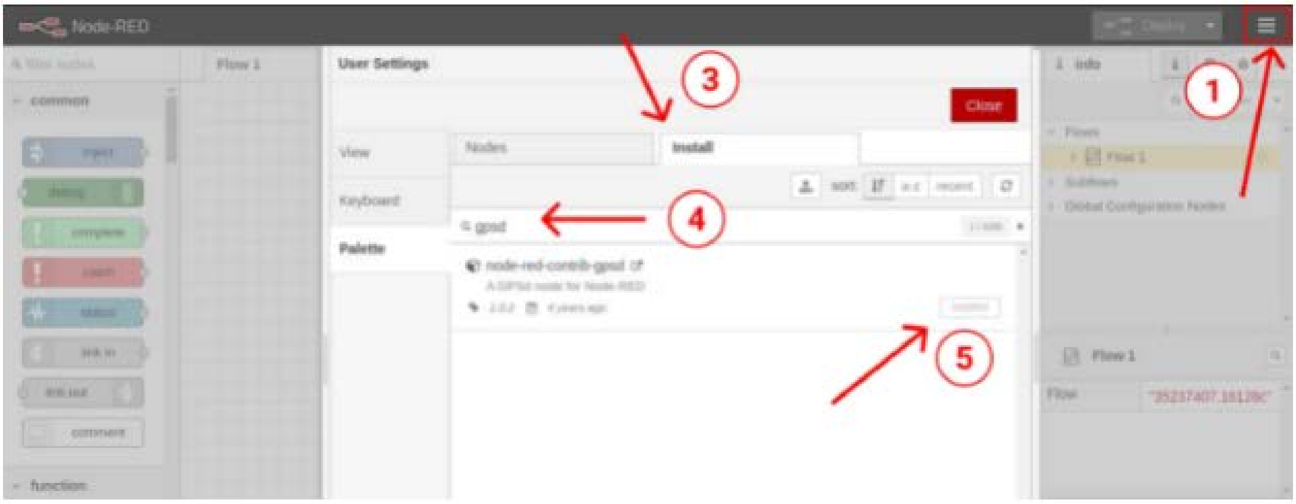
[Running on Raspberry Pi](#)

GPSD

GPSD is a monitor daemon that collects information from GPS, differential-GPS radios, or AIS receivers attached to the host machine. Each GPS, DGPS radio, or AIS receiver is expected to be connected directly to the host via a USB or RS232 serial device. So, to link the data you get from your Serial Port (See how here >>) to Node-RED, just download some gpsd nodes: **node-red-contrib-gpsd.**

Go to the **Side Tab Menu > Click on Manage Palette > Install > Type "gpsd" > Install the node-red-contrib-gpsd package:**

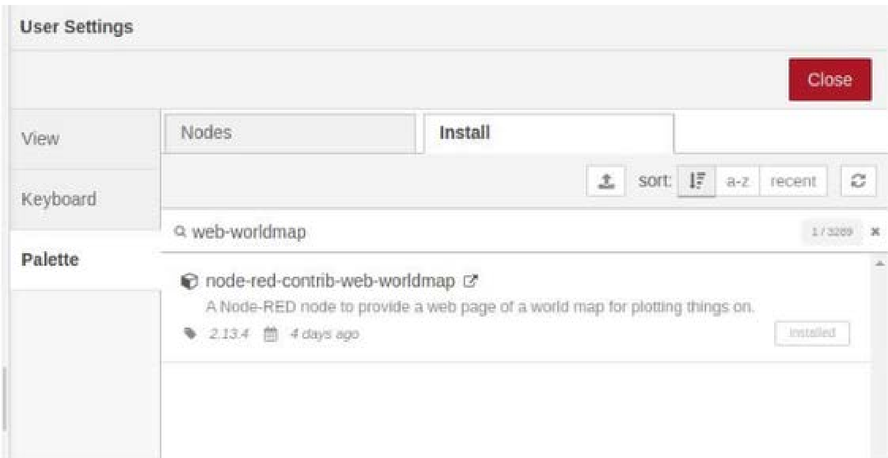
GPSD



Install Map nodes

Now, you are going to install some nodes for the Map Widget.

Go to the Side Tab Menu > Click on Manage Palette > Install > Type "web-worldmap" > Install the ***node-red-contrib-web-worldmap*** package:



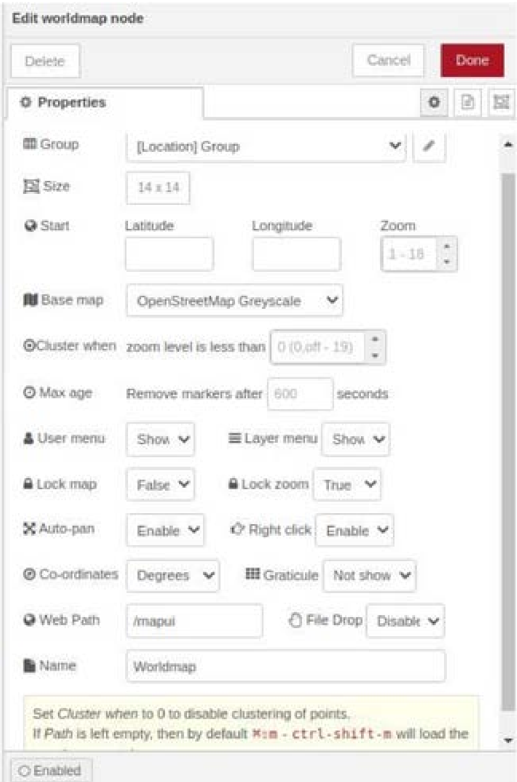
Getting GPS Coordinates

Once installed, type: gpsd in the filter nodes search bar, and drag and drop the **Adafruit Ultimate GPS node** to your flow. Add a **switch node** and evaluate the property: **msg.payload.class == (string) TPV** and connect it to a **debug node** like this:

You will see that you will start receiving data immediately, every second. So, if you want to control that time, just add a Delay node between the Adafruit Ultimate GPS node and the switch node, with the configuration you want. For example, try receiving 1 msg every 5 seconds, and drop intermediate messages.

Add the WorldMap node

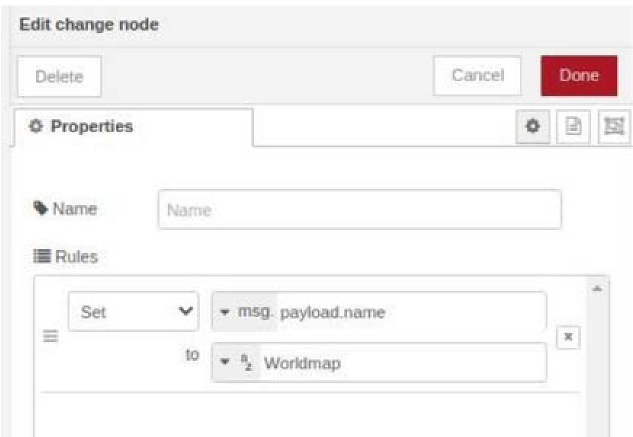
Once you get all the **GPS data**, you are going to **add two worldmap nodes** so that it displays in the Dashboard correctly:



If you take a look at the node help documentation, the minimum msg.payload from the worldmap nodes, must contain **name, lat** and **lon** properties.

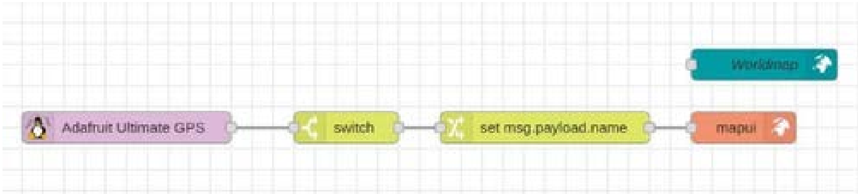
```
E.g. { "name": "John", "lat": 41.45, "lon": 1.53 }
```

So, as you already get the lat and lon properties from the GPS, you need the msg.payload.name. So, let's add a change node, and set msg.payload.name to "Worldmap" (or any name):



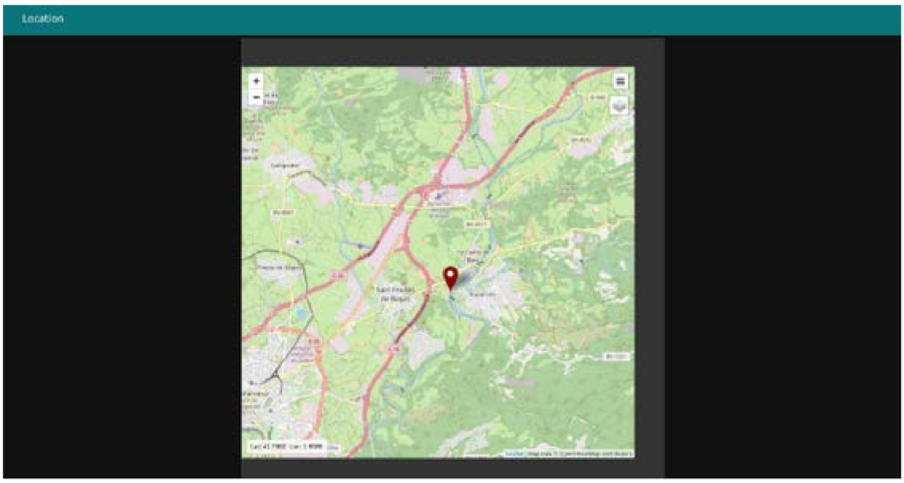
Add the WorldMap node

Connect this to the world map node as shown below:



And go to your **Dashboard** to check your map with **your location**! Your flow should something be like this:


```
[{"id":"6e545ede.d4925","type":"tab","label":"Flow 1","disabled":false,"info":""},
{"id":"7a97b4c5.876cfc","type":"gpsd","z":"6e545ede.d4925","name":"Adafruit Ultimate
GPS","hostname":"0.0.0.0","port":"2947","tpv":true,"sky":true,"info":false,"device":t
rue,"gst":false,"att":false,"x":140,"y":200,"wires":[["6560f5f6.c0ed8c"]]},
{"id":"6560f5f6.c0ed8c","type":"switch","z":"6e545ede.d4925","name":"","property":"pa
yload.class","propertyType":"msg","rules":
[{"t":"eq","v":"TPV","vt":"str"}],"checkall":"true","repair":false,"outputs":1,"x":35
0,"y":200,"wires":[["4d22a0bd.63dda"]]},
{"id":"2cf3ed7c.bd6ad2","type":"ui_worldmap","z":"6e545ede.d4925","group":"88f77589.0
122d8","order":2,"width":"14","height":"14","name":"Worldmap
","lat":"","lon":"","zoom":"","layer":"OSM
grey","cluster":"","maxage":"","usermenu":"show","layers":"show","panit":"true","panl
ock":"false","zoomlock":"true","hiderightclick":"false","coords":"deg","showgrid":"fa
lse","allowFileDrop":"false","path":"/mapui","x":710,"y":120,"wires":[]},
{"id":"4d22a0bd.63dda","type":"change","z":"6e545ede.d4925","name":"","rules":
[{"t":"set","p":"payload.name","pt":"msg","to":"Worldmap","tot":"str"}],"action":"","
property":"","from":"","to":"","reg":false,"x":540,"y":200,"wires":
[["23b5e03.345882"]]},
{"id":"23b5e03.345882","type":"worldmap","z":"6e545ede.d4925","name":"","lat":"","lon
":"","zoom":"","layer":"Esri
Satellite","cluster":"","maxage":"","usermenu":"show","layers":"show","panit":"false"
,"panlock":"false","zoomlock":"false","hiderightclick":"false","coords":"none","showg
rid":"false","allowFileDrop":"false","path":"/mapui","x":730,"y":200,"wires":[]},
{"id":"88f77589.0122d8","type":"ui_group","name":"","tab":"c4c17961.4519f8","order":1
,"disp":true,"width":15,"collapse":false},
{"id":"c4c17961.4519f8","type":"ui_tab","name":"Location","icon":"dashboard","disable
d":false,"hidden":false}]
```

PROFINET & Raspberry PLC tutorial: How to set communication on Linux

Learn how to run the p-net Profinet device stack and its sample application on a Raspberry PLC!

PROFINET (**Pro**cess **F**ield **Net**work) is a network standard for Industrial Automation based on open Ethernet and non-proprietary for automation.

The industrial **Raspberry PLC** has two **Ethernet** ports, and its own OS from the Raspberry, Raspbian, **Linux**.

Requirements

- **1x Raspberry Pi**
- **1x [Raspberry Pi based PLC]**
- **Ethernet/USB Hub**
- **Raspberry power cable - USB-C type**
- **2x Ethernet cables**

PROFINET

PROFINET is the international open **Ethernet** industrial standard of PROFIBUS & PROFINET (pi) for **automation**.

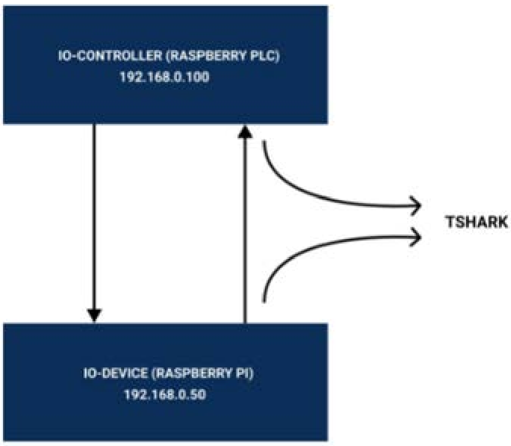
- PROFINET uses the **TCP/IP** and LO standards, is PROFI real-time Ethernet, and allows investment protection with the integration of **fieldbus** systems.

- **ProFiNet IO** (Input Output): Developed real-time (RT) and **real-time** isochronous (IRT) **communication** with the decentralized periphery. The names RT and IRT are limited to describing the properties in time for communication in ProFiNet IO.

The PROFINET concept offers modular mode structure so that users can select cascade connection themselves. They differ essentially due to the type of data exchange to meet the requirements, partly very high speed.

With PROFINET, communication without discontinuities from the management level to the field level is possible. On the other hand, it meets the great demands imposed by industry, e.g., ex. wiring and connection system suitable for **industrial environment**, real time, motion control in **isochronous mode**, non-proprietary engineering, etc.

PROFINET



PROFINET was developed with the aim of fostering a process of convergence between **industrial automation** and the information management platform for corporate management and global corporate networks. It applies to Ethernet-based distributed automation systems that integrate existing field bus systems, for example **PROFIBUS**, but without modifying them.

Notes to advanced users

The IO-device sample application can be running on:

- **Raspberry Pi**
- **Any Linux OS**
- **An embedded board running an RTOS, such as RT-kernel**

Files

The **sample_app** directory in the p-net repository contains the source code for this tutorial. It also contains a GSD file which tells the IO-controller how to communicate with the IO-device. Those parts of the sample application that are dependent on whether you run Linux or an RTOS are located in **src/ports**.

Install dependencies

1. First of all, **connect** your Raspberry Pi **to Internet** via LAN or Wi-Fi to be able to download some packages.

See how

2. Then, in order to **compile p-net** on Raspberry Pi, you need a recent version of cmake. Install it:

```
sudo apt update
sudo apt install snapd
sudo reboot
sudo snap install cmake --classic
```

3. Verify the installed version:

```
cmake --version
```

Download and compile p-net

1. Install git to download p-net:

```
sudo apt install git
```

2. Create a directory called ***profinet*:**

```
mkdir /home/pi/profinet
cd /home/pi/profinet
```

3. Clone the repository with submodules. Then create and **configure** the **build:**

```
git clone --recurse-submodules https://github.com/rtlabs-com/p-net.git
cmake -B build -S p-net
cmake --build build --target install
```

Notes to advanced users

If you have already **cloned the repository without** the ***--recurse-submodules* flag,** then run in the **p-net folder**.

```
git submodule update --init --recursive.
```

Alternate ***cmake*** command to also adjust some settings:

```
cmake -B build -S p-net -DCMAKE_BUILD_TYPE=Debug -DBUILD_TESTING=OFF -
DBUILD_SHARED_LIBS=ON -DUSE_SCHED_FIFO=ON
```

Depending on how you installed ***cmake***, you might need to run ***snap run cmake*** instead of ***cmake***.

Run the sample application

1. Go to the **/home/pi/profinet/build directory**:

```
cd /home/pi/profinet/build
```

Run the sample application

2. Enable the **Ethernet interface** and set the **initial IP address**:

```
sudo ifconfig eth0 192.168.0.50 netmask 255.255.255.0 up
```

3. Execute the **sample application:**

```
sudo ./pn_dev -v -v -v -v
```

From the Raspberry PLC

1. Configure the **Ethernet interface from the Raspberry Pi** industrial PLC and set the **initial IP address:**

```
sudo ifconfig eth0 192.168.0.100 netmask 255.255.255.0 up
```

2. Download the **tshark package:**

```
sudo apt update
sudo apt install -y tshark
```

3. Add a new **system group** called ***tshark*:**

```
sudo groupadd tshark
```

4. Add **current user** to a tshark group:

```
sudo usermod -a -G tshark $USER
```

- To make changes to **take effect, logout and login to Raspberry Pi.** After you are reconnected, **check tshark version:**

```
tshark --version
```

- To start **capturing packets** on the **default network interface with tshark**, simply execute this command:

```
sudo tshark
```

From the Raspberry PLC

- To identify which **network interfaced** are available to the tshark, run the following command:

```
sudo ifconfig eth0 192.168.0.100 netmask 255.255.255.0 up
```

You can use -i option to capture packets on specific network interface

```
sudo tshark -i eth0
```

Get the results

```
151 232.320974055 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
152 237.318528588 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
153 242.316080714 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
154 247.313632729 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
155 252.311323003 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
156 257.308732778 Raspberr_db:27:27 →LLDP_Multicast LLDP 173 TTL = 20 RTClass3 Port Status = OFF
```

```
728 1651.588778480 192.168.0.50 →224.0.0.22   IGMPv3 60 Membership Report / Join group 224.0.0.251 for any sources
729 1651.603858534 Raspberr_db:27:27 →PN-MC_00:00:01 PN-DCP 90 Hello Req, Xid:0x1, NameOfStation:"rt-labs-dev", IP, Dev-ID, OEM-Dev-ID, DeviceInitiative
730 1651.758899803 Raspberr_db:27:27 →Broadcast    ARP 60 Who has 192.168.0.50? Tell 0.0.0.0
731 1652.408350768 192.168.0.50 →224.0.0.22   IGMPv3 60 Membership Report / Join group 224.0.0.251 for any sources
732 1652.834707184 192.168.0.50 →224.0.0.251  MDNS 140 Standard query 0x0000 ANY 50.0.168.192.in-addr.arpa, "QM" question ANY myraspberrypi.local, "QM" question A 192.168.0.50 PTR myraspberrypi.local
733 1653.084593653 192.168.0.50 →224.0.0.251  MDNS 140 Standard query 0x0000 ANY 50.0.168.192.in-addr.arpa, "QM" question ANY myraspberrypi.local, "QM" question A 192.168.0.50 PTR myraspberrypi.local
734 1653.334613655 192.168.0.50 →224.0.0.251  MDNS 140 Standard query 0x0000 ANY 50.0.168.192.in-addr.arpa, "QM" question ANY myraspberrypi.local, "QM" question A 192.168.0.50 PTR myraspberrypi.local
735 1653.535448048 192.168.0.50 →224.0.0.251  MDNS 128 Standard query response 0x0000 PTR, cache flush myraspberrypi.local A, cache flush 192.168.0.50
```

LLDP

LLDP or Link Layer Discovery Protocol is used by PROFINET to determine and manage neighborhood relationships **between PROFINET devices.** LLDP uses the special multicast **MAC address: 01-80-C2-00-00-0E** and the **Link layer Ethernet II and IEEE 802. 1Q and Ethertype 0x88CC (PROFINET)**. Finally, the port number is non relevant:

No	Time	Source	Destination	Protocol	Length	Info
151	232.320974055	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF
152	237.318528588	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF
153	242.316080714	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF
154	247.313632729	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF
155	252.311323003	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF
156	257.308732778	Raspberr_db:27:27	→LLDP_Multicast	LLDP	173	TTL = 20 RTClass3 Port Status = OFF

Autostart on boot

- If you want **pnet** to run when the Pi is turned on, or re-booted, first enable the serial port console by writing the following line in the file **/boot/config.txt** :

```
enable_uart=1
```

Autostart on boot

1. Go to **/etc/dhcpcd.conf** and include these lines to set a static IP:

```
interface eth0
static ip_address=192.168.0.100/24
```

2. Then, you can enable the service to autostart by running the following commands:

```
sudo        cp        /home/pi/profinet/p-net/src/ports/linux/pnet-sampleapp.service
/lib/systemd/system
sudo systemctl daemon-reload
sudo systemctl enable pnet-sampleapp.service
```

Start service

```
sudo systemctl start pnet-sampleapp.service
```

- To see the status of the process, and the log output:

```
sudo systemctl status pnet-sampleapp.service
journalctl -fu pnet-sampleapp
```

Finally, **reboot** to apply the changes and get your service running!

Node-RED & Raspberry tutorial: How to capture data from sensor

Learn capturing values from weight sensor with Industrial Raspberry PLC and Node-Red applications

Raspberry Pi

Node-RED is a **programming tool** for wiring together **hardware devices, APIs and online services** in new and interesting ways.

It provides a **browser-based editor** that makes it easy to wire together flows using the wide range of node in the palette that can be deployed to its runtime in a single-click.

You will be learn how to develop the **Node-RED application** that was shown in **this post.**

To know more about Node-RED, please visit **https://nodered.org/**

Node-Red basics

As we said in the introduction, Node-RED provides a browser-based editor that makes it easy to **wire together flows** using the wide range of nodes in the palette that can be deployed to its runtime in a single click.

So, let's go to discover the basics:

Node-RED has a wide range of nodes that offers you a lot of possibilities. If you go to the nodes menu on the left, you will find the nodes that come by default. They are easy to use; you just have to drag and drop them in your flow so that you can start using them.

1. Moreover, if you already know what node you want, there is a search bar to filter nodes and find exactly the one you want.

2. If you double-click on the Flow 1 tab, a configuration window will be displayed, where you can change its name, or disable it, for example. In the same bar, there is a + tab which adds another Flow tab, so that you can use as many as you want.

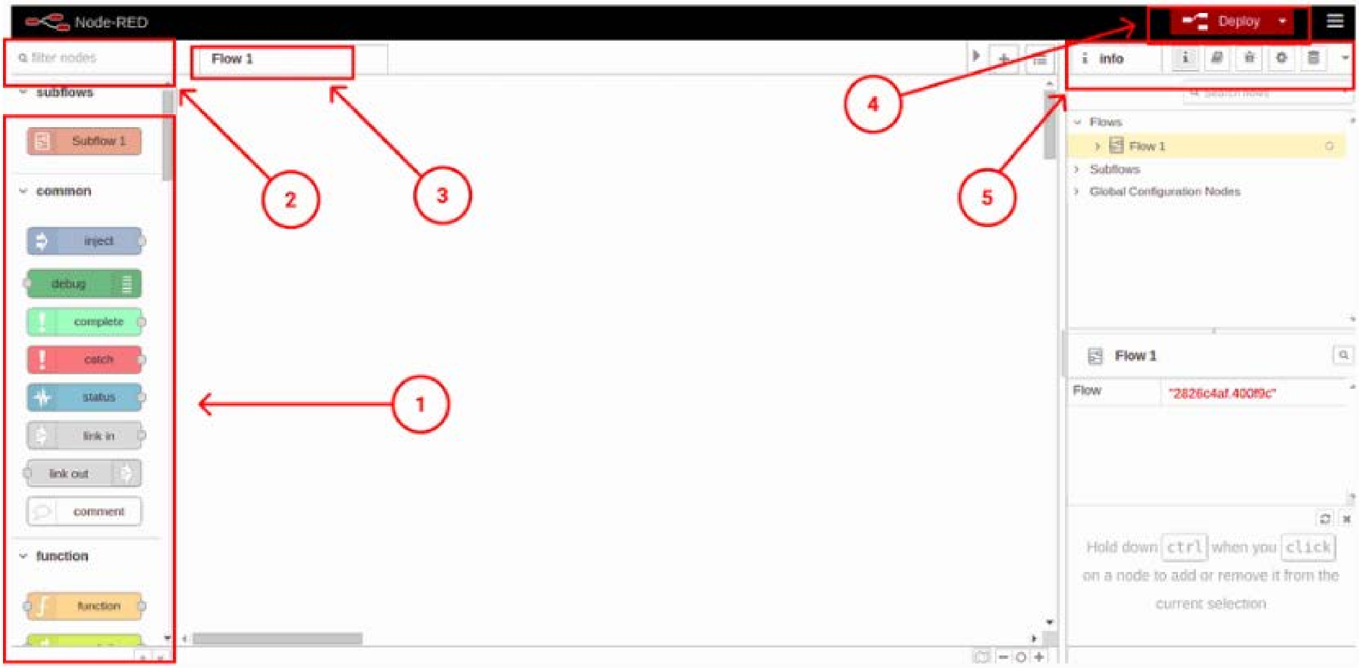
3. Once you have your nodes connected and you want to Deploy your changes, click on the Deploy button.

Autostart on boot

3. Next to the **Deploy button**, there is a menu that allows you to import or export your flows, or if you go to **Manage palette > Install**, and you type the nodes you want to install, you will be able to download as many nodes as you want, like the **node-red-dashboard** or the **node-red-contrib-ui-media**, for example.

4. Finally, in the **right bar** where the info tab is displayed, there are more important tabs such as:

- **Information:** to get general information about the flows.
- **Help tab:** This gives you information about the node you clicked on.
- **Debug messages:** this is a really useful tab to know the errors you got, or to display the debug node messages.
- **Configuration nodes:** It shows the configuration nodes from the flows.
- **Dashboard:** This tab allows you to set the dashboard layout, tsite configuration and theme.



Our nodes

So, now you know the basics, let's introduce the nodes we are going to use:

- **Ui_button node:** Adds a button to the user interface. Clicking the button generates a message with **msg.payload** set to the Payload field. If no payload is specified, the node id is used.
- **Function node:** A JavaScript function to run against the messages being received by the node. The messages are passed in as a JavaScript object called msg. By convention, it will have a msg.payload property containing the body of the message.
- **Exec node:** Runs a system command and returns its output. The node can be configured to either wait until the command completes, or to send its output as the command generates it. The command that is run can be configured in the node or provided by the received message.
- **Change node:** Set, change, delete or move properties of a message, flow context or global context. The node can specify multiple rules that will be applied in the order they are defined.
- **Switch node:** Route messages based on their property values or sequence position.

Our nodes

- **Ui_chart node:** Plots the input values on a chart. This can either be a time based line chart, a bar chart (vertical or horizontal), or a pie chart.
- **Ui_gauge node:** Adds a gauge type widget to the user interface. The msg.payload is searched for a numeric value and is formatted in accordance with the defined Value Format.
- **Ui_media node:** Displays media files and URLs on the Dashboard.
- **Status node:** Report status messages from other nodes on the same tab.

Getting the weight value

What we are going to do is to **start getting** the **values** from a **weight sensor**, and when the application founds the value we set, the **USB camera** will take a picture.

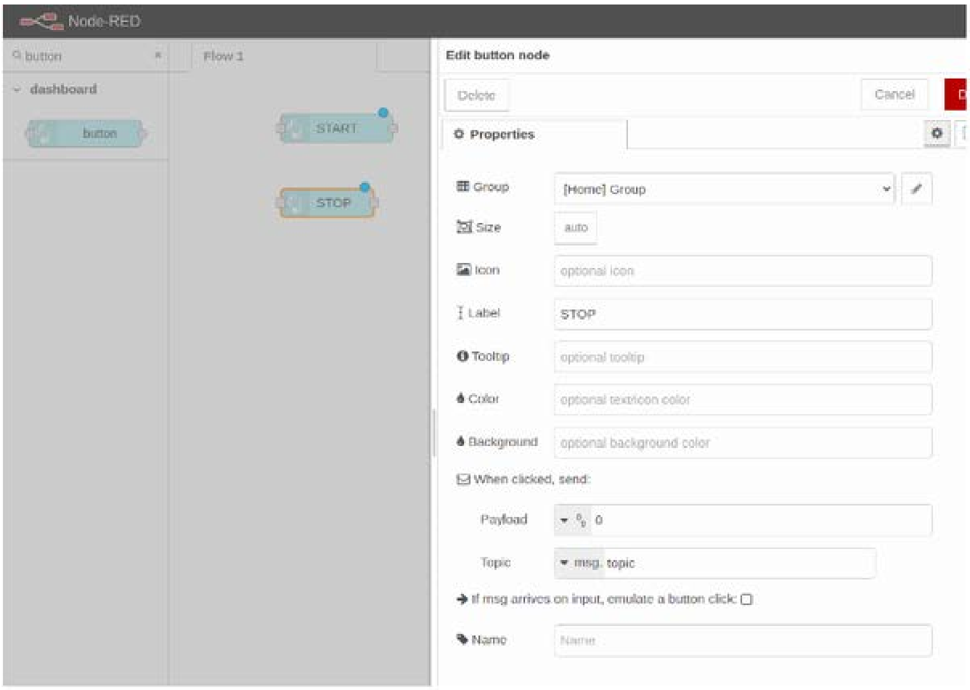
So, let's start developing our application!

1. First of all, you are going to add **two dashboard buttons**: the first one to start the application, and the other one to stop it.

So, go the filter nodes search bar and type: **button**. Add two buttons to the flow, and double-click to edit them.

In the first one, you must create a **UI Group and a UI Tab** to display our dashboard. Once done, it will work for all the Dashboard nodes, so it is only necessary once. After that, you will type a label to be displayed, in our case: **START.**

Likewise, the stop button will have the same configuration; you will select the group and tab where you want to display it, you will type: **STOP as a label** and we will add a **0 to the payload**, so that the value for the gauge sets to 0 when the application stops, instead of stopping in the last value.



Getting the weight value

2. You are going to **add a function node** next to the start button and wire it.

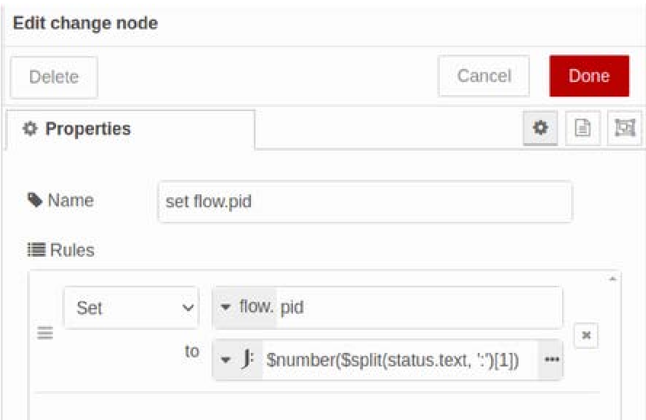
In the start node, you are going to **initialize a flow variabl**e named count to 0, which you are going to use later on when you name the pictures, and you are going to send the message with the command to execute for the app to start.

```
var count = flow.get('count')||0;
flow.set('count', count);
var newMsg = {payload: "python -u /home/pi/hx711py/example.py"};
return newMsg;
```

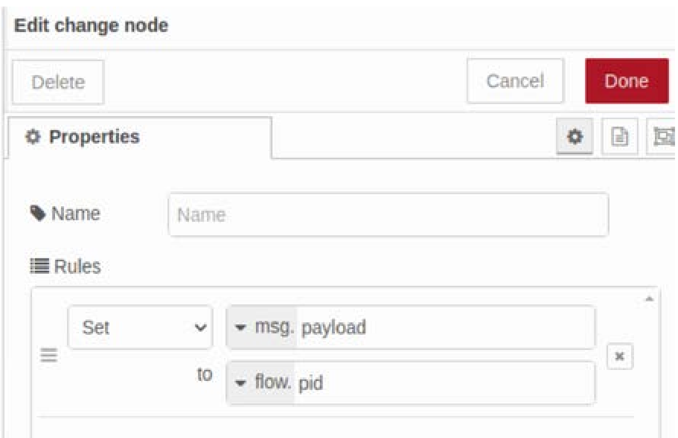
You can give a name to the function node as you would like to see it in your flow. In this case: start **flow.count** and send **python cmd**. Finally, wire an **Exec node** and edit it. Select the output: "while the command is running **- spawn mode**", and click on the checkbox to append the msg.payload.

3. When there is an exec node running as **spawn mode**, that generates a pid of the **running process**, you will have to get to be able to kill it. So that is what you are going to do right now.

Add a **status node**, go to **"Report status from"** and select **"Selected nodes"**. Choose the **exec node**, and click on Done. After that, wire a **Change node**, and edit it to set the **flow.pid** as shown below:

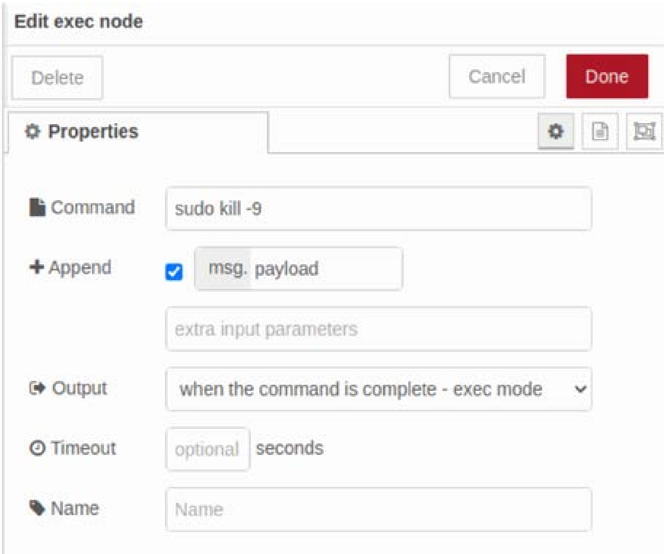


Finally, **add another change** node next to the stop button and connect them. As we set the **flow.pid** in the previous change node, now we are going to set the **msg.payload to flow.pid**. By doing this, when you click on the Stop button, the msg.payload will be sent through the node.



Getting the weight value

So, now the **pid** is the **msg.payload**. Add an exec node as an exec mode to kill the pid, and edit it:



At the moment, your flow will look like:


```
[{"id":"2826c4af.400f9c","type":"tab","label":"Flow 1","disabled":false,"info":""},
{"id":"bce0df4f.bc788","type":"ui_button","z":"2826c4af.400f9c","name":"","group":"c4
c1bcc1.49c24","order":16,"width":"7","height":"2","passthru":false,"label":"START
","tooltip":"","color":"","bgcolor":"","icon":"","payload":"","payloadType":"str","to
pic":"topic","topicType":"msg","x":140,"y":140,"wires":[["882b392c.ab71b8"]]},
{"id":"222e70bc.56f6","type":"ui_button","z":"2826c4af.400f9c","name":"","group":"c4c
1bcc1.49c24","order":15,"width":0,"height":0,"passthru":false,"label":"STOP","tooltip
":"","color":"","bgcolor":"","icon":"","payload":"0","payloadType":"num","topic":"top
ic","topicType":"msg","x":130,"y":220,"wires":[["63e42d5b.dee384"]]},
{"id":"882b392c.ab71b8","type":"function","z":"2826c4af.400f9c","name":"start
flow.count and send python cmd","func":"var count =
flow.get('count')||0;\nflow.set('count', count);\n\nvar newMsg = {payload: \"python -
u /home/pi/hx711py/example.py\"};\nreturn
newMsg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":
[],"x":410,"y":140,"wires":[["9628a2eb.2a5d3"]]},
{"id":"2abcf1ce.f1931e","type":"status","z":"2826c4af.400f9c","name":"","scope":
["9628a2eb.2a5d3"],"x":140,"y":60,"wires":[["b7fab428.f4fb78"]]},
{"id":"9628a2eb.2a5d3","type":"exec","z":"2826c4af.400f9c","command":"","addpay":"pay
load","append":"","useSpawn":"true","timer":"","oldrc":false,"name":"","x":690,"y":14
0,"wires":[[],[],[]]},
{"id":"b7fab428.f4fb78","type":"change","z":"2826c4af.400f9c","name":"","rules":
[{"t":"set","p":"pid","pt":"flow","to":"$number($split(status.text, ':')
[1])","tot":"jsonata"}],"action":"","property":"","from":"","to":"","reg":false,"x":4
10,"y":60,"wires":[[]]},
{"id":"63e42d5b.dee384","type":"change","z":"2826c4af.400f9c","name":"","rules":
[{"t":"set","p":"payload","pt":"msg","to":"pid","tot":"flow"}],"action":"","property"
:"","from":"","to":"","reg":false,"x":420,"y":220,"wires":[["46ba8b75.815004"]]},
{"id":"46ba8b75.815004","type":"exec","z":"2826c4af.400f9c","command":"sudo kill
-9","addpay":"payload","append":"","useSpawn":"false","timer":"","oldrc":false,"name"
:"","x":690,"y":220,"wires":[[],[],[]]},
{"id":"c4c1bcc1.49c24","type":"ui_group","name":"Group","tab":"cbda5f28.c75ad","order
":1,"disp":true,"width":"20","collapse":false},
{"id":"cbda5f28.c75ad","type":"ui_tab","name":"Home","icon":"dashboard","disabled":fa
lse,"hidden":false}]
```

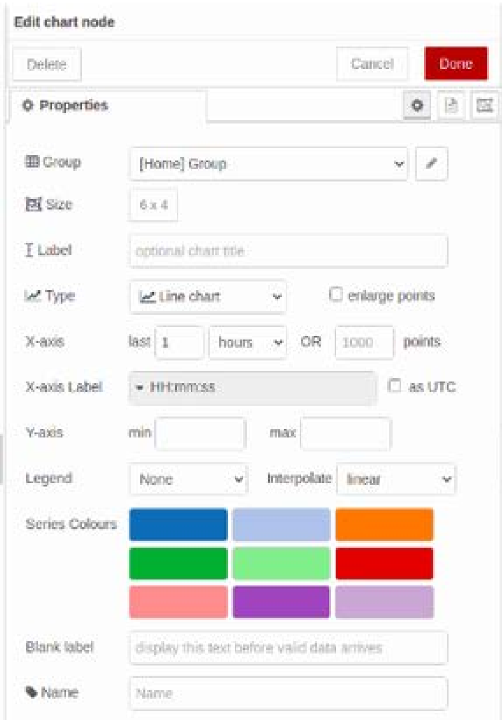
Getting the weight value

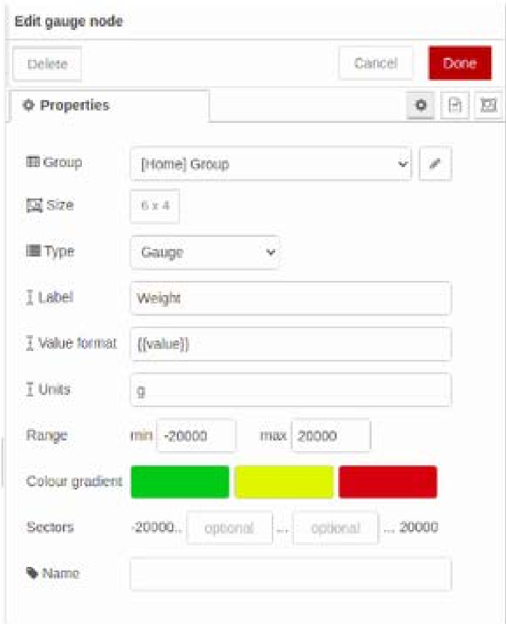
4. Now, you are going to see the **values** from the **last 1 hour** in a line chart, and also in real time in a gauge.

Then, drag and drop a chart node and a gauge node, and let's edit them.

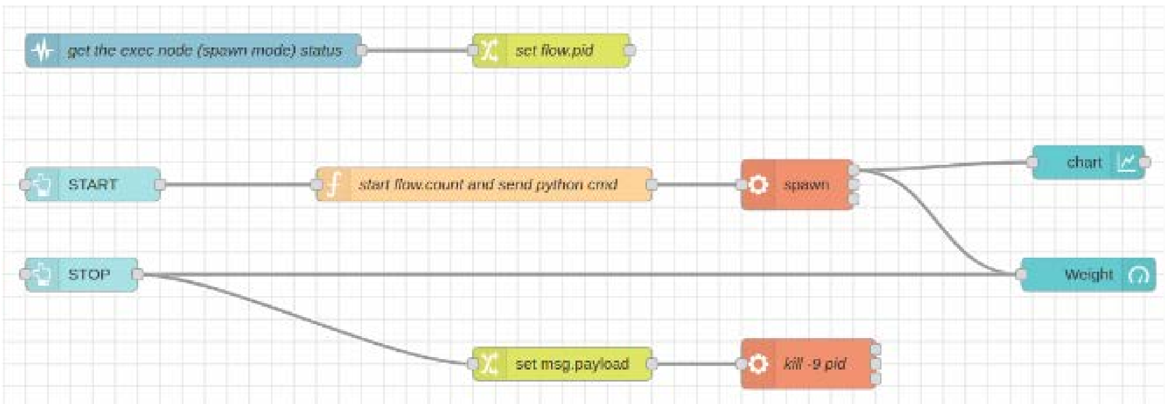
In the chart node, set de **X-axis to the last 1 hour**, or the time you would like to register, add the Tab and Group you would like to display in and click on Done.

Edit the gauge node by choosing the same **Tab and Group** and setting a label to display as its title, also type the units. Finally, set the **minimum and the maximum value** to set the range:





Finally, connect them as shown below:



Weight! Picture this!

Once you got the values of our Raspberry scale and you displayed them to your Dashboard, it is time to **take some photos.**

For the following steps, it is necessary to install the **node-red-contrib-ui-media**, so if you did not do it yet, please go to the last post to know how.

5. Now, you are going to **add a switch node** and set if a property is between **50 and 100** to take a picture. The **values are up to you**, just choose the value rules, choose the number field, and add the number you want to feature. Connect this node to the **spawn node**.

6. Connected to the output of the last change node, add a function node to send the **fswebcam command** and set the **flow.count** to name the pictures with a counter as follows:

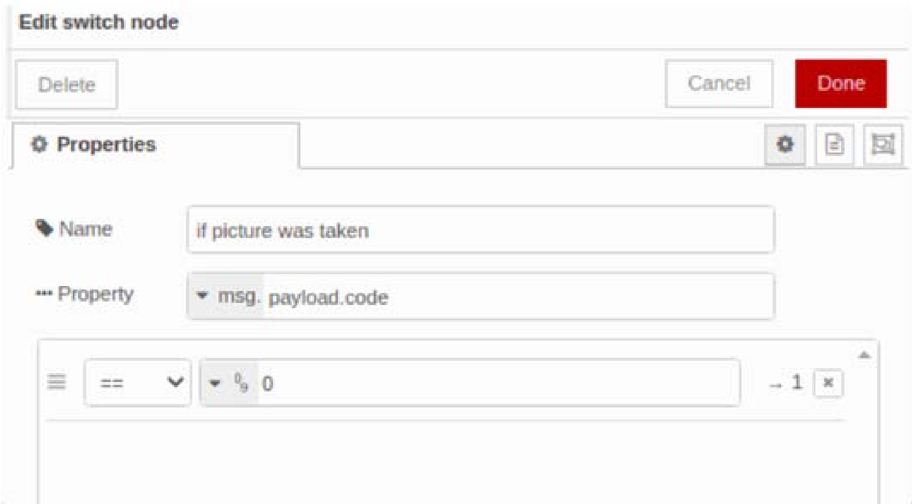
```
var count = flow.get('count'); count++;
msg.payload = "fswebcam -r 1280x720 --no-banner /home/pi/images/image" + count +
".jpg";
flow.set('count', count); return msg;
```

You should add **three parameters** to the **fswebcam** command:
a. - r to set the photo resolution.
b. --no-banner to skip the camera banner
c. The path to say where to save the images, and how are they be going to be named.

7. The function node will send a msg.payload, so you are going to add an exec node appending the msg.payload to execute the command in your industrial Raspberry Pi PLC controller.

8. The exec mode has three outputs. The first one returns the stdout, the second one returns the stderr and the last one, returns the return code. So in this case, connect the third output, the return code, to a switch node to continue with the flow if there was no error.

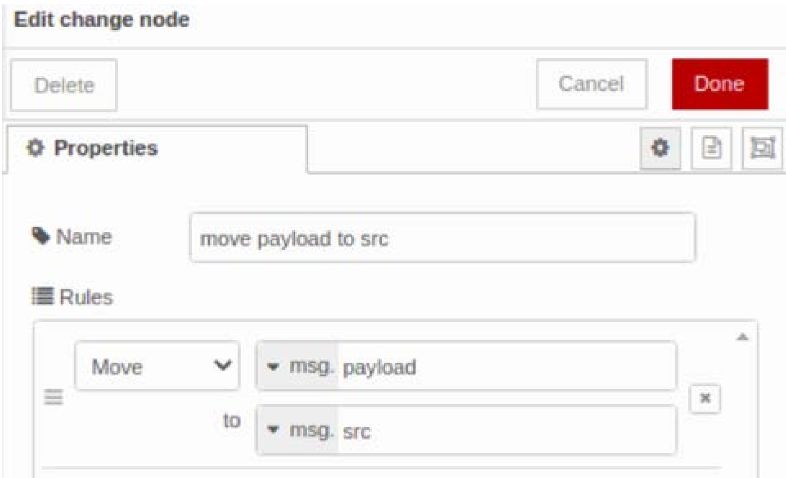
So, in the switch node, set the property to **msg.payload.code** and set the value rule to **equal number 0**, to be sure that the fwwebcam command was executed with **no errors.**



Weight! Picture this!

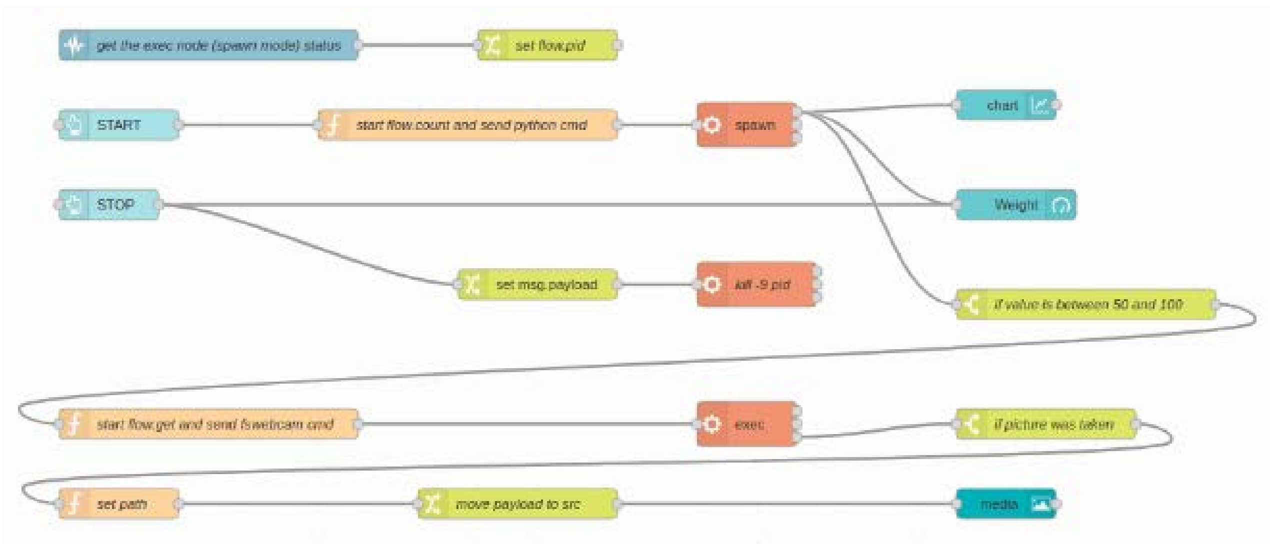
9. After that, connect a function node to send the name of the picture it was just taken, so that it can be displayed in the Node-RED Dashboard. Once edited as shown below, connect a change node to move the **msg.payload to msg.src**:

```
let count = flow.get('count');
msg.payload = "/image" + count + ".jpg";
return msg;
```



10. Finally, **add the media node** and just add a Group to it, or configure the layout as you want.

Now, your Node-RED application should look like this (you have the code after the picture):

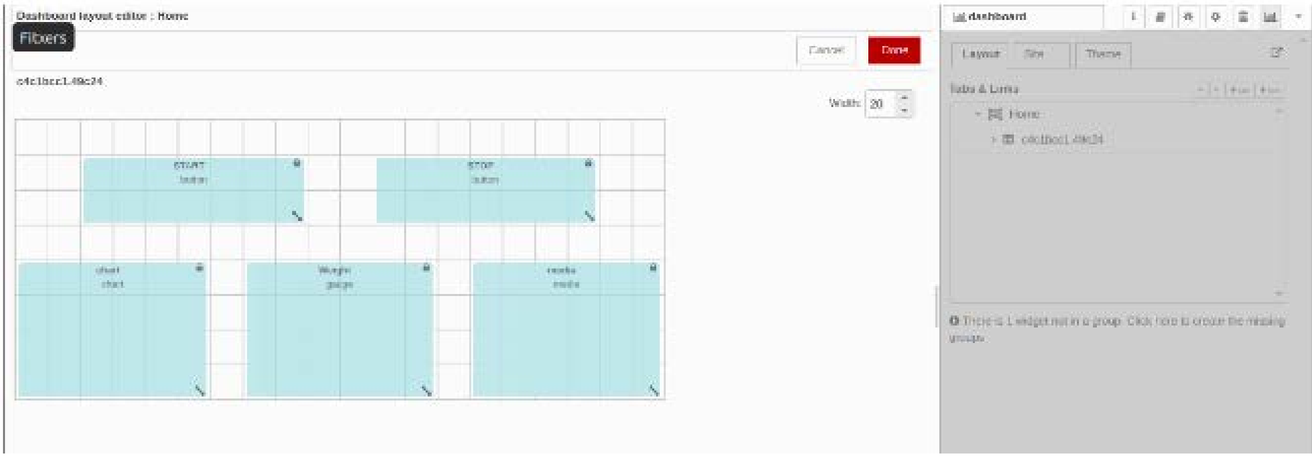


Weight! Picture this!

[{"id":"2826c4af.400f9c","type":"tab","label":"Flow 1","disabled":false,"info":""},
{"id":"9b234a13.0256e8","type":"exec","z":"2826c4af.400f9c","command":"","addpay":"payload","append":"","
useSpawn":"false","timer":"","oldrc":false,"name":"","x":510,"y":260,"wires":[[],[],
["3f27e8b4.d02378"]]},
{"id":"ec5481a.4fbf28","type":"exec","z":"2826c4af.400f9c","command":"","addpay":"payload","append":"","u
seSpawn":"true","timer":"","oldrc":false,"name":"","x":870,"y":60,"wires":
[["36307784.3144e8","372e8f7b.f9752","d4e34cd5.f423e"],[],[]]},
{"id":"36307784.3144e8","type":"switch","z":"2826c4af.400f9c","name":"if value is between 50 and
100","property":"payload","propertyType":"msg","rules":
[{"t":"btwn","v":"50","vt":"num","v2":"100","v2t":"num"}],"checkall":"true","repair":false,"outputs":1,"x
":990,"y":160,"wires":[["fb666c7f.c2ff9"]]},
{"id":"3f27e8b4.d02378","type":"switch","z":"2826c4af.400f9c","name":"if picture was
taken","property":"payload.code","propertyType":"msg","rules":
[{"t":"eq","v":"0","vt":"num"}],"checkall":"true","repair":false,"outputs":1,"x":1030,"y":260,"wires":
[["7646b60e.83a318"]]},{"id":"7646b60e.83a318","type":"function","z":"2826c4af.400f9c","name":"set
path","func":"let count = flow.get('count');\nmsg.payload = \"/image\" + count + \".jpg\";\nreturn
msg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":[],"x":140,"y":360,"wires":
[["a4078c82.e803a"]]},{"id":"a4078c82.e803a","type":"change","z":"2826c4af.400f9c","name":"move payload
to src","rules":
[{"t":"move","p":"payload","pt":"msg","to":"src","tot":"msg"}],"action":"","property":"","from":"","to":"
","reg":false,"x":560,"y":360,"wires":[["3385b59c.06c81a"]]},
{"id":"8771f8be.e44f68","type":"ui_button","z":"2826c4af.400f9c","name":"","group":"c4c1bcc1.49c24","orde
r":5,"width":7,"height":2,"passthru":false,"label":"STOP LOAD
CELL","tooltip":"","color":"","bgcolor":"","icon":"","payload":"0","payloadType":"num","topic":"","topicT
ype":"str","x":170,"y":160,"wires":[["e053ecae.bca31","d4e34cd5.f423e"]]},
{"id":"d4e34cd5.f423e","type":"ui_gauge","z":"2826c4af.400f9c","name":"","group":"c4c1bcc1.49c24","order"
:13,"width":6,"height":4,"gtype":"gage","title":"Weight","label":"g","format":"
{{value}}","min":"-2000","max":"2000","colors":
["#00b500","#e6e600","#ca3838"],"seg1":"","seg2":"","x":1070,"y":100,"wires":[]},
{"id":"372e8f7b.f9752","type":"ui_chart","z":"2826c4af.400f9c","name":"","group":"c4c1bcc1.49c24","order"
:11,"width":6,"height":4,"label":"","chartType":"line","legend":"false","xformat":"HH:mm:ss","interpolate
":"linear","nodata":"","dot":false,"ymin":"","ymax":"","removeOlder":1,"removeOlderPoints":"","removeOlde
rUnit":"3600","cutout":0,"useOneColor":false,"useUTC":false,"colors":
["#1f77b4","#aec7e8","#ff7f0e","#2ca02c","#98df8a","#d62728","#ff9896","#9467bd","#c5b0d5"],"outputs":1,"
useDifferentColor":false,"x":1070,"y":60,"wires":[[]]},
{"id":"99be7e29.78696","type":"ui_button","z":"2826c4af.400f9c","name":"","group":"c4c1bcc1.49c24","order
":3,"width":7,"height":2,"passthru":false,"label":"START LOAD
CELL","tooltip":"","color":"","bgcolor":"","icon":"","payload":"","payloadType":"str","topic":"","topicTy
pe":"str","x":180,"y":60,"wires":[["615b1ef6.53963"]]},
{"id":"615b1ef6.53963","type":"function","z":"2826c4af.400f9c","name":"start flow.count and send python
cmd","func":"var count = flow.get('count')||0;\nflow.set('count', count);\n\nvar newMsg = {payload:
\"sudo python -u /home/pi/hx711py/example.py\"};\nreturn
newMsg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":[],"x":610,"y":60,"wires":
[["ec5481a.4fbf28"]]},
{"id":"3385b59c.06c81a","type":"ui_media","z":"2826c4af.400f9c","group":"c4c1bcc1.49c24","name":"","width
":6,"height":4,"order":15,"category":"","file":"","layout":"expand","showcontrols":true,"loop":true,"onst
art":false,"scope":"local","tooltip":"","x":1070,"y":360,"wires":[[]]},
{"id":"16de31a2.e4a6de","type":"status","z":"2826c4af.400f9c","name":"get the exec node status","scope":
["ec5481a.4fbf28"],"x":190,"y":600,"wires":[["9bb820b3.87fbe"]]},
{"id":"9bb820b3.87fbe","type":"change","z":"2826c4af.400f9c","name":"set flow.pid","rules":
[{"t":"set","p":"pid","pt":"flow","to":"$number($split(status.text, ':')
[1])","tot":"jsonata"}],"action":"","property":"","from":"","to":"","reg":false,"x":410,"y":600,"wires":
[["b99c988c.86bf98"]]},{"id":"b99c988c.86bf98","type":"function","z":"2826c4af.400f9c","name":"set kill
cmd","func":"let pid = flow.get('pid');\nvar kill = \"kill -9 \" + pid;\nflow.set('kill', kill);\nreturn
msg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":[],"x":590,"y":600,"wires":[[]]},
{"id":"e053ecae.bca31","type":"function","z":"2826c4af.400f9c","name":"killall
python","func":"msg.payload = \"sudo killall python\";\nreturn
msg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":[],"x":530,"y":160,"wires":
[["ec5481a.4fbf28"]]},{"id":"fb666c7f.c2ff9","type":"function","z":"2826c4af.400f9c","name":"start
flow.get and send fswebcam cmd","func":"var count = flow.get('count');\ncount++;\n\nmsg.payload =
\"fswebcam -r 1280x720 --no-banner /home/pi/images/image\" + count + \".jpg\";\n\nflow.set('count',
count);\n\nreturn msg;\n\n","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":
[],"x":230,"y":260,"wires":[["9b234a13.0256e8"]]},
{"id":"5771d86a.220b58","type":"comment","z":"2826c4af.400f9c","name":"In case you want to kill the flow
pid and not the python processes, replace the \"killall python\" function node, for the \"killall pid\"
function node -->","info":"","x":550,"y":540,"wires":[]},
{"id":"b88b67eb.03f068","type":"function","z":"2826c4af.400f9c","name":"killall pid","func":"msg.payload
= flow.get('kill');\nreturn msg;","outputs":1,"noerr":0,"initialize":"","finalize":"","libs":
[],"x":1100,"y":540,"wires":[[]]},
{"id":"c4c1bcc1.49c24","type":"ui_group","name":"","tab":"cbda5f28.c75ad","order":1,"disp":true,"width":"
20","collapse":false},
{"id":"cbda5f28.c75ad","type":"ui_tab","name":"Home","icon":"dashboard","disabled":false,"hidden":false}]

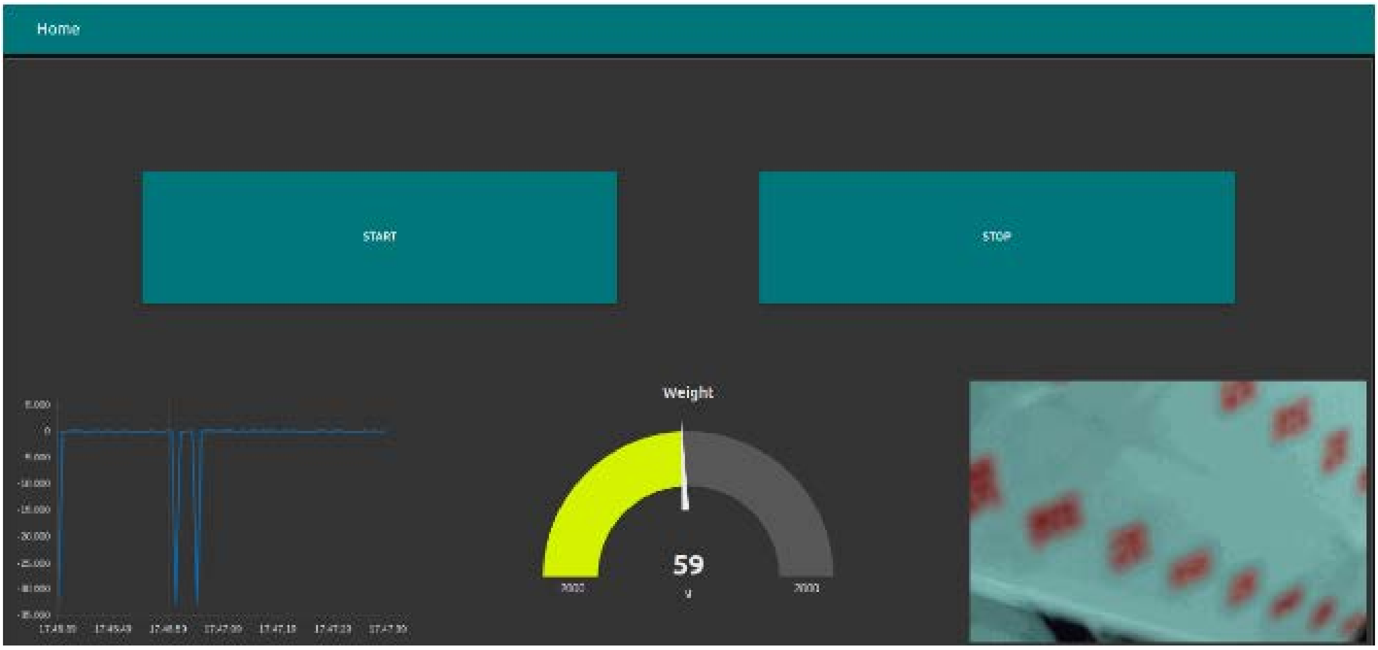
Tips

If you go to the **right menu**, in the Dashboard tab, and you hover on your tab, you will see appear **three buttons: group, edit and layout**. So, if you click on layout, you will see the Dashboard layout editor, where it is possible to display your **ui nodes** as you want.



If you see that you cannot resize the widgets, go to **every Dashboard node**, and in the **Size section,** you will see that is set as auto, so just **set any manual size**, go back to the Dashboard layout editor, where the changes will be applied.

Finally, go to http://10.10.10.20:1880/ui/ to check out **your Node-RED dashboard!**



I. InfluxDB & Node-RED & MQTT Tutorial: How to install InfluxDB

Store your data with Influx by using Node-RED and the MQTT!

Raspberry Pi

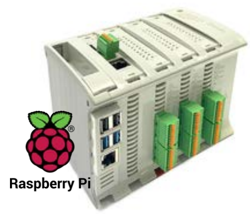
influxdb

InfluxDB is an **open source time-series database (TSDB)**, written in Go and optimised for **fast, high-availability storage and retrieval of time series data** in fields such as operations **monitoring, application metrics, Internet of Things sensor data,** and **real-time analytics.**

You will learn **how to install InfluxDB and create an InfluxDB database** in order to store data coming from **MQTT through Node-RED.**

Requirements

- Raspberry Pi PLC
- Power Supply
- **Any industrial Raspberry PLC access:**
    - Remote Raspberry Pi based PLC access
    - Raspberry Pi PLC controller access by HDMI



What is Node-RED and how does it work

Node-RED is a **programming tool** for connecting together **hardware devices, APIs and online services** in new and interesting ways.

It provides a **browser-based editor** that makes it easy to wire flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

What is InfluxDB used of

InfluxDB is a **time series platform** that enables developers to **build IoT, analytics and monitoring software.**

It is designed to handle the massive volumes and countless sources of time-stamped data produced by **sensors, applications and infrastructure.**

What is MQTT

MQTT is an **OASIS standard messaging protocol** for the **Internet of Things (IoT)**.

It is designed as an **extremely lightweight publish/subscribe messaging transport** that is ideal for connecting remote devices with a **small code footprint** and **minimal network bandwidth.**

MQTT today is used in a wide variety of industries, such as **automotive, manufacturing, telecommunications, oil and gas, etc.**

Install InfluxDB

Now that you know a little bit more about the tools that you are going to use, you are going to install **InfluxDB** on your Raspberry Pi industrial PLC.

1. First of all, update the **aptitude package**:

```
sudo apt update
```

2. Then, add the **InfluxDB repository key** to your industrial Raspberry Pi PLC in order to allow the package manager on **Raspbian** to search the repository and verify the packages installed.

```
wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add -
```

3. Add the repository to the **sources list.**

```
echo "deb https://repos.influxdata.com/debian $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/influxdb.list
```

4. Update the **package list** again.

```
sudo apt update
```

5. Install **InfluxDB** onto your open source **PLC Raspberry Pi:**

```
sudo apt install influxdb
```

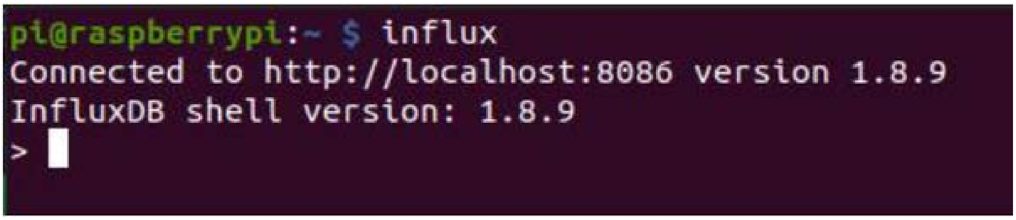
6. Install **InfluxDB** onto your open source **PLC Raspberry Pi:**

```
sudo systemctl unmask influxdb
sudo systemctl enable influxdb
sudo systemctl start influxdb
```

Install InfluxDB

Finally, start **InfluxDB** by typing ***influx*** in the commandline of your Raspberry PLC:

```
influx
```



How to create an InfluxDB database

As InfluxDB comes **without any database by default**, you are going to create the **first one**, called ***test.*** Into this **influx prompt**, follow the next steps:

1. Create a **database** called ***test***.

```
create database test
```

2. Go into the database:

```
use test
```

3. Now, you will show everything from the database with the **command below.** You will see that obviously is **empty**. You just do this step in order to make sure what it contains right now.

```
select * from test
```

Now that you have created your first database with InfluxDB, let's insert data from Node-RED!

Want to know how?

Click here

Industrial Shields

II. InfluxDB & Node-RED & MQTT Tutorial: Sending data to InfluxDB

Learn how to store data from Node-RED to InfluxDB with a Raspberry PLC

Raspberry Pi

As **InfluxDB** is an **open-source time series database (TSDB)**, it is designed so that it is easy to store and access information. A good database is crucial to any company or organization. This is because the database stores all the pertinent details about what the company wants to store, like salaries, transactional records or any valuable information.

We will continue this post and we will learn how to **send data** to the database, only installing an InfluxDB node on Node-RED.

Node-red-contrib-influxdb

Now, we are going to open Node-RED and install node-red-contrib-influxdb nodes from InfluxDB.

1. So, go to the **Menu > Manage Palette > Install > Type: node-red-contrib-influxdb** and click on Install.

2. Then, go to the **nodes sections**, filter your search by ***influx***, and drop an Influx out node to the flow.

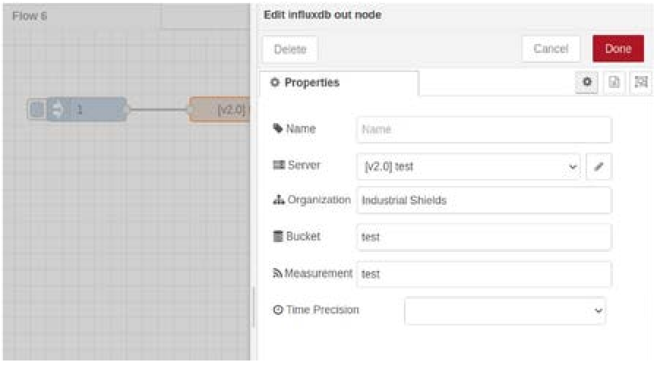
3. Then configure your Influx out node with a **server, organisation, bucket and measurement**. We did it like this:


```
[{"id":"0e17644c4b3628b4","type":"influxdb out","z":"b716fdc48724e610","influxdb":"bc4ab5cb2a050021","name":"","measurement":"test","precision":"","retentionPolicy":"","database":"","retentionPolicyV18Flux":"","org":"Industrial          Shields","bucket":"test","x":340,"y":100,"wires":[]},{"id":"bc4ab5cb2a050021","type":"influxdb","hostname":"127.0.0.1","port":"8086","protocol":"http","database":"test","name":"test","usetls":true,"tls":"d50d0c9f.31e858","influxdbVersion":"2.0","url":"http://localhost:8086","rejectUnauthorized":false,"credentials":{}},{"id":"d50d0c9f.31e858","type":"tls-config","name":"","cert":"","key":"","ca":"","certname":"","keyname":"","caname":"","servername":"","verifyservercert":false}]
```


Note: In order to import this **json file**, go to **Node-RED > Menu > Import > Paste this JSON file,** and click on Import.

Node-red-contrib-influxdb

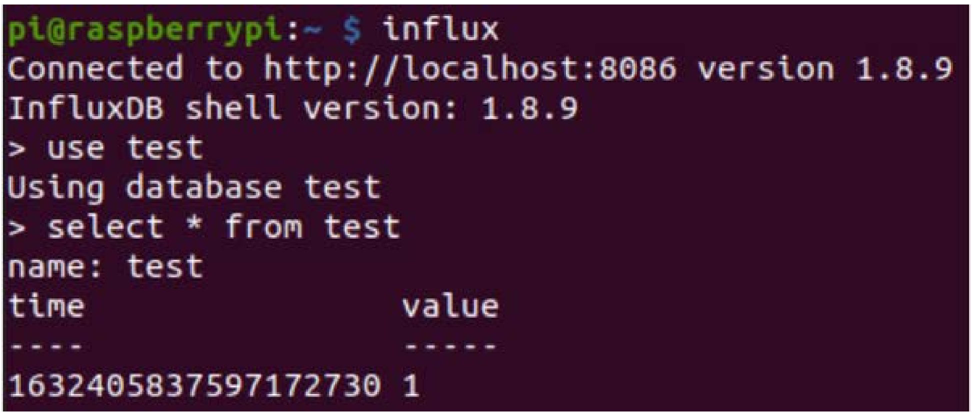
4. Finally, add an **inject node with a number** to run this example.



Get data from InfluxDB

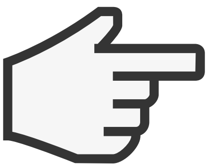
Now, deploy your application and click on the button from the inject node to send the data to the database.

So, let's see what we got in the database now!



```
pi@raspberrypi:~ $ influx
Connected to http://localhost:8086 version 1.8.9
InfluxDB shell version: 1.8.9
> use test
Using database test
> select * from test
name: test
time                value
----                -----
1632405837597172730 1
```

As you can see, we have **stored the information** in InfluxDB! But, **what if the information would not come from an inject node, but from MQTT data?**



Click here to continue

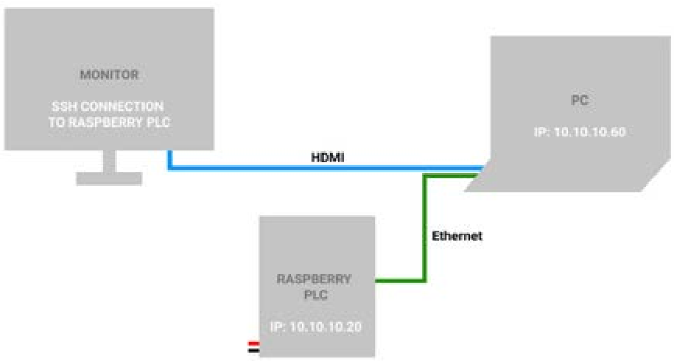
III. InfluxDB & Node-RED & MQTT Tutorial: Getting data from MQTT

Learn Node-RED & InfluxDB. And MQTT: the best protocol for the IoT industry

Raspberry Pi

In the previous parts, we learned how to install InfluxDB, and how to send any number from an inject node to the Influx database that we created. In this section, we will learn how to get data from MQTT, and send it to the database!

Connections



First of all, we have to make sure about the **connections**. Right now we have our **laptop connected** through **Ethernet and SSH** to the industrial Raspberry PLC, and we have an extra monitor to work better with the information from the Raspberry Pi based PLC.

Publish MQTT message from PC

What we are going to do now, is to publish a MQTT message from our PC. And we will subscribe to it from the industrial Raspberry Pi PLC.

1. So, the first thing we are going to do it is to **open Node-RED from your PC:**

```
localhost:1880
```

2. Add an inject node with a number msg.payload and we will send it to a MQTT out node with the **topic: /industrial/shields:**

Publish MQTT message from PC

3. Import our flow in order to compare it with yours by going to: **Menu > Import > Paste this JSON > Click on Import**


```
[{"id":"f04655a6.2e0548","type":"tab","label":"Flow  1","disabled":false,"info":""},
{"id":"18eca158.c30bef","type":"ui_spacer","name":"spacer","group":"","order":3,"wid
th":3,"height":1},
{"id":"66b1ac62.f204c4","type":"ui_spacer","name":"spacer","group":"","order":5,"wid
th":2,"height":1},
{"id":"57e7a054.1d3d","type":"ui_spacer","name":"spacer","group":"","order":6,"width
":1,"height":1},
{"id":"3d2f0a19.b50e16","type":"ui_spacer","name":"spacer","group":"","order":7,"wid
th":3,"height":1},
{"id":"2818fa86.78b246","type":"ui_spacer","name":"spacer","group":"","order":8,"wid
th":2,"height":1},
{"id":"b7071d44.aac77","type":"ui_spacer","name":"spacer","group":"","order":9,"widt
h":14,"height":1},
{"id":"76d851d4.191b1","type":"ui_spacer","name":"spacer","group":"","order":10,"wid
th":1,"height":1},
{"id":"2b8a21d5.7e913e","type":"ui_spacer","name":"spacer","group":"","order":12,"wi
dth":1,"height":1},
{"id":"8ff507a5.bebcf8","type":"ui_spacer","name":"spacer","group":"","order":13,"wi
dth":1,"height":1},
{"id":"4e0db42.3a7d74c","type":"ui_spacer","name":"spacer","group":"","order":14,"wi
dth":1,"height":1},
{"id":"6ef46fcc.54f8d","type":"ui_spacer","name":"spacer","group":"","order":15,"wid
th":14,"height":1},
{"id":"ff69b32a.fafbb","type":"ui_spacer","name":"spacer","group":"","order":16,"wid
th":1,"height":1},
{"id":"a4c5a26b.0ba75","type":"ui_spacer","name":"spacer","group":"","order":18,"wid
th":1,"height":1},
{"id":"aba323e0.729b9","type":"ui_spacer","name":"spacer","group":"","order":19,"wid
th":1,"height":1},
{"id":"caa68f8b.b56ca","type":"ui_spacer","name":"spacer","group":"","order":20,"wid
th":1,"height":1},
{"id":"a9e2309bbb6821ba","type":"ui_spacer","name":"spacer","group":"","order":2,"wi
dth":1,"height":1},
{"id":"21ea384a01fa887e","type":"ui_spacer","name":"spacer","group":"","order":3,"wi
dth":1,"height":1},
{"id":"d5d81a9f0d428697","type":"ui_spacer","name":"spacer","group":"","order":4,"wi
dth":1,"height":1},
{"id":"da4a771ac31dcf3d","type":"ui_spacer","name":"spacer","group":"","order":5,"wi
dth":1,"height":1},
{"id":"fe4d904b8bb40878","type":"ui_spacer","name":"spacer","group":"","order":6,"wi
dth":1,"height":1},
{"id":"6b3d8da4e9a96852","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":1,"height":1},
{"id":"74821225a7ca51ca","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":1,"height":1},
{"id":"e91dbd2767a763f6","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":1,"height":1},
{"id":"93d3502802dcc1a4","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":2,"height":1},
{"id":"9fc8c3278ee9b9c1","type":"ui_spacer","name":"spacer","group":"","order":10,"w
idth":1,"height":1},
{"id":"f774fbfaae54eb0e","type":"ui_spacer","name":"spacer","group":"","order":12,"w
idth":10,"height":1},{"default":"#0094CE","value":"#0094CE","edited":false},
```

Publish MQTT message from PC

{"id":"08e6b009b586964d","type":"ui_spacer","name":"spacer","group":"","order":17,"w
idth":1,"height":1},
{"id":"b6232aa234a57cb1","type":"ui_spacer","name":"spacer","group":"","order":20,"w
idth":1,"height":1},
{"id":"a60378ac9f87fc5e","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":1,"height":1},
{"id":"260c6340f6f14ad3","type":"ui_spacer","name":"spacer","group":"","order":12,"w
idth":9,"height":1},
{"id":"2b980305848cfeef","type":"ui_spacer","name":"spacer","group":"","order":19,"w
idth":1,"height":1},
{"id":"22a9f72987124d95","type":"ui_spacer","name":"spacer","group":"","order":5,"wi
dth":5,"height":1},
{"id":"5e9d977e96083094","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":9,"height":1},
{"id":"9110a4597689ae8b","type":"ui_spacer","name":"spacer","group":"","order":10,"w
idth":9,"height":1},
{"id":"459ab95d2d2a5ddb","type":"ui_spacer","name":"spacer","group":"","order":11,"w
idth":9,"height":1},
{"id":"90e1410acc8c38b9","type":"ui_spacer","name":"spacer","group":"","order":16,"w
idth":5,"height":1},
{"id":"7ad2d307c987a7df","type":"ui_spacer","name":"spacer","group":"","order":18,"w
idth":2,"height":1},
{"id":"e5e9feb0763ec81e","type":"ui_spacer","name":"spacer","group":"","order":20,"w
idth":1,"height":1},
{"id":"093ef67678864b13","type":"ui_spacer","name":"spacer","group":"","order":1,"wi
dth":6,"height":1},
{"id":"b94afa82cf8ecc1f","type":"ui_spacer","name":"spacer","group":"","order":3,"wi
dth":6,"height":1},
{"id":"faf9e9fde02df27d","type":"ui_spacer","name":"spacer","group":"","order":4,"wi
dth":2,"height":1},
{"id":"a8cbdcf68248d1b9","type":"ui_spacer","name":"spacer","group":"","order":6,"wi
dth":2,"height":1},
{"id":"1dfce3d8addd0131","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":2,"height":1},
{"id":"add5f326020a30a1","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":2,"height":1},
{"id":"0c6c63f5118690bf","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":2,"height":1},
{"id":"73f373a14ff151ec","type":"ui_spacer","name":"spacer","group":"","order":11,"w
idth":2,"height":1},
{"id":"ce026db7ddc0b2d0","type":"ui_spacer","name":"spacer","group":"","order":13,"w
idth":2,"height":1},
{"id":"f8b17415c15a6072","type":"ui_spacer","name":"spacer","group":"","order":15,"w
idth":2,"height":1},
{"id":"8f4603f38cd344ad","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":2,"width":1,"height":1},
{"id":"562468767aea6755","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":4,"width":2,"height":1},
{"id":"6a6113cdfd6d877e","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":6,"width":2,"height":1},
{"id":"4d89a801118d61ef","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":7,"width":2,"height":1},
{"id":"72e7fd9dec97d5ce","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":8,"width":2,"height":1},
{"id":"4a4b3e4669027be4","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":9,"width":2,"height":1},e":"spacer","group":"78426f95.8f95a","order":8,"wid
th":2,"height":1},

Publish MQTT message from PC


```
{"id":"79bd20907ddb50c4","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":10,"width":3,"height":1},
{"id":"c1fed531493ab6fe","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":12,"width":3,"height":1},
{"id":"3e3030c3709fe31c","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":13,"width":3,"height":1},
{"id":"bf367c6bb97ffc0a","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":14,"width":3,"height":1},
{"id":"79689c89c65ac453","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":15,"width":3,"height":1},
{"id":"26078b1caf5a4049","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":17,"width":3,"height":1},
{"id":"03d10337489964ad","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":18,"width":3,"height":1},
{"id":"a3a820bc7fb53497","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":19,"width":3,"height":1},
{"id":"9f9ae7141861e1c8","type":"ui_spacer","name":"spacer","group":"","order":2,"wi
dth":1,"height":1},
{"id":"d9cb8337360336d6","type":"ui_spacer","name":"spacer","group":"","order":3,"wi
dth":1,"height":1},
{"id":"a86c41aef3f1182e","type":"ui_spacer","name":"spacer","group":"","order":4,"wi
dth":1,"height":1},
{"id":"9666ccbf2e136ba9","type":"ui_spacer","name":"spacer","group":"","order":5,"wi
dth":1,"height":1},
{"id":"d6403fae75f2ce72","type":"ui_spacer","name":"spacer","group":"","order":6,"wi
dth":1,"height":1},
{"id":"1577da1ad00267c8","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":1,"height":1},
{"id":"85a6633f734a397f","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":1,"height":1},
{"id":"6d0451543273b8e9","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":1,"height":1},
{"id":"a096632e7e8c7fcc","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":2,"height":1},
{"id":"01c034e071a22d06","type":"ui_spacer","name":"spacer","group":"","order":10,"w
idth":1,"height":1},
{"id":"a41773576044e9e2","type":"ui_spacer","name":"spacer","group":"","order":12,"w
idth":10,"height":1},
{"id":"38b3ea45b44de95a","type":"ui_spacer","name":"spacer","group":"","order":17,"w
idth":1,"height":1},
{"id":"4f7c5bfb900e811b","type":"ui_spacer","name":"spacer","group":"","order":20,"w
idth":1,"height":1},
{"id":"2bf13987b7d07c68","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":1,"height":1},
{"id":"925897d6382cfc1e","type":"ui_spacer","name":"spacer","group":"","order":12,"w
idth":9,"height":1},
{"id":"f9dbd340829a20ab","type":"ui_spacer","name":"spacer","group":"","order":19,"w
idth":1,"height":1},
{"id":"84efe78acd668123","type":"ui_spacer","name":"spacer","group":"","order":5,"wi
dth":5,"height":1},
{"id":"7403c4408d998eb9","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":9,"height":1},
{"id":"a72cd60a04683869","type":"ui_spacer","name":"spacer","group":"","order":10,"w
idth":9,"height":1},
{"id":"27a841b63ebf78df","type":"ui_spacer","name":"spacer","group":"","order":11,"w
idth":9,"height":1},
```

Publish MQTT message from PC


```
{"id":"2507a7e17f059640","type":"ui_spacer","name":"spacer","group":"","order":16,"w
idth":5,"height":1},
{"id":"dd648ea2b4fc993e","type":"ui_spacer","name":"spacer","group":"","order":18,"w
idth":2,"height":1},
{"id":"f6f18b0119a0c7a8","type":"ui_spacer","name":"spacer","group":"","order":20,"w
idth":1,"height":1},
{"id":"6f20eee74019c663","type":"ui_spacer","name":"spacer","group":"","order":1,"wi
dth":6,"height":1},
{"id":"7e797c689f839bd5","type":"ui_spacer","name":"spacer","group":"","order":3,"wi
dth":6,"height":1},
{"id":"159875d396fd2e1d","type":"ui_spacer","name":"spacer","group":"","order":4,"wi
dth":2,"height":1},
{"id":"09f15455e01dc3bd","type":"ui_spacer","name":"spacer","group":"","order":6,"wi
dth":2,"height":1},
{"id":"00b530d543c6533d","type":"ui_spacer","name":"spacer","group":"","order":7,"wi
dth":2,"height":1},
{"id":"9b68b46d65334e55","type":"ui_spacer","name":"spacer","group":"","order":8,"wi
dth":2,"height":1},
{"id":"2b103ab7680be12c","type":"ui_spacer","name":"spacer","group":"","order":9,"wi
dth":2,"height":1},
{"id":"a45e230894b860f4","type":"ui_spacer","name":"spacer","group":"","order":11,"w
idth":2,"height":1},
{"id":"d96da445e3f00f3b","type":"ui_spacer","name":"spacer","group":"","order":13,"w
idth":2,"height":1},
{"id":"8c8b1dc3d2ecd22e","type":"ui_spacer","name":"spacer","group":"","order":15,"w
idth":2,"height":1},
{"id":"ad65b0982b9515d2","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":2,"width":1,"height":1},
{"id":"8572d6bead68d3fd","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":4,"width":2,"height":1},
{"id":"7535d92b622832e2","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":6,"width":2,"height":1},
{"id":"acc331cf74b68edf","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":7,"width":2,"height":1},
{"id":"d8a017de905c275c","type":"ui_spacer","nam{"id":"30e75722341fe68d","type":"ui_
spacer","name":"spacer","group":"78426f95.8f95a","order":9,"width":2,"height":1},
{"id":"b1a10de362b4fc16","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":10,"width":3,"height":1},
{"id":"6edfbbcaa60fa199","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":12,"width":3,"height":1},
{"id":"15600e26f2233629","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":13,"width":3,"height":1},
{"id":"90ddb4d963776480","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":14,"width":3,"height":1},
{"id":"43c7e25f67aadcb1","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":15,"width":3,"height":1},
{"id":"dd2ed8c042a0b788","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":17,"width":3,"height":1},
{"id":"d3b14202340ff557","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":18,"width":3,"height":1},
{"id":"da0c9031a45eeb31","type":"ui_spacer","name":"spacer","group":"78426f95.8f95a"
,"order":19,"width":3,"height":1},
{"id":"a4e70dca39a11812","type":"ui_tab","name":"Home","icon":"dashboard","disabled"
:false,"hidden":false},{"id":"6902ce18c912334f","type":"ui_base","theme":
{"name":"theme-light","lightTheme":
```

Publish MQTT message from PC


```
{"default":"#0094CE","baseColor":"#0094CE","baseFont":"-apple-
system,BlinkMacSystemFont,Segoe     UI,Roboto,Oxygen-Sans,Ubuntu,Cantarell,Helvetica
Neue,sans-serif","edited":true,"reset":false},"darkTheme":
{"default":"#097479","baseColor":"#097479","baseFont":"-apple-
system,BlinkMacSystemFont,Segoe     UI,Roboto,Oxygen-Sans,Ubuntu,Cantarell,Helvetica
Neue,sans-serif","edited":false},"customTheme":{"name":"Untitled               Theme
1","default":"#4B7930","baseColor":"#4B7930","baseFont":"-apple-
system,BlinkMacSystemFont,Segoe     UI,Roboto,Oxygen-Sans,Ubuntu,Cantarell,Helvetica
Neue,sans-serif"},"themeState":{"base-color":"page-titlebar-backgroundColor":
{"value":"#0094CE","edited":false},"page-backgroundColor":
{"value":"#fafafa","edited":false},"page-sidebar-backgroundColor":
{"value":"#ffffff","edited":false},"group-textColor":
{"value":"#1bbfff","edited":false},"group-borderColor":
{"value":"#ffffff","edited":false},"group-backgroundColor":
{"value":"#ffffff","edited":false},"widget-textColor":
{"value":"#111111","edited":false},"widget-backgroundColor":
{"value":"#0094ce","edited":false},"widget-borderColor":
{"value":"#ffffff","edited":false},"base-font":{"value":"-apple-
system,BlinkMacSystemFont,Segoe     UI,Roboto,Oxygen-Sans,Ubuntu,Cantarell,Helvetica
Neue,sans-serif"}},"angularTheme":
{"primary":"indigo","accents":"blue","warn":"red","background":"grey","palette":"lig
ht"}},"site":{"name":"Node-RED
Dashboard","hideToolbar":"false","allowSwipe":"false","lockMenu":"false","allowTempT
heme":"true","dateFormat":"DD/MM/YYYY","sizes":
{"sx":48,"sy":48,"gx":6,"gy":6,"cx":6,"cy":6,"px":0,"py":0}}},
{"id":"0a51eb25daf41663","type":"mqtt-
broker","name":"","broker":"localhost","port":"1883","clientid":"","usetls":false,"p
rotocolVersion":"4","keepalive":"60","cleansession":true,"birthTopic":"","birthQos":
"0","birthPayload":"","birthMsg":
{},"closeTopic":"","closeQos":"0","closePayload":"","closeMsg":
{},"willTopic":"","willQos":"0","willPayload":"","willMsg":{},"sessionExpiry":""},
{"id":"5b73900a1aadda9c","type":"mqtt
out","z":"f04655a6.2e0548","name":"","topic":"/industrial/shields","qos":"","retain"
:"","respTopic":"","contentType":"","userProps":"","correl":"","expiry":"","broker":
"0a51eb25daf41663","x":470,"y":100,"wires":[]},
{"id":"36cdcf455932b604","type":"inject","z":"f04655a6.2e0548","name":"","props":
[{"p":"payload"},
{"p":"topic","vt":"str"}],"repeat":"","crontab":"","once":false,"onceDelay":0.1,"top
ic":"","payload":"2","payloadType":"num","x":180,"y":100,"wires":
[["5b73900a1aadda9c"]]}]
```


Subscribe to MQTT message from Raspberry Pi PLC controller

As we want to subscribe to the MQTT topic: /industrial/shields from the Raspberry Pi industrial PLC, we are going to open a new tab in our browser and type:

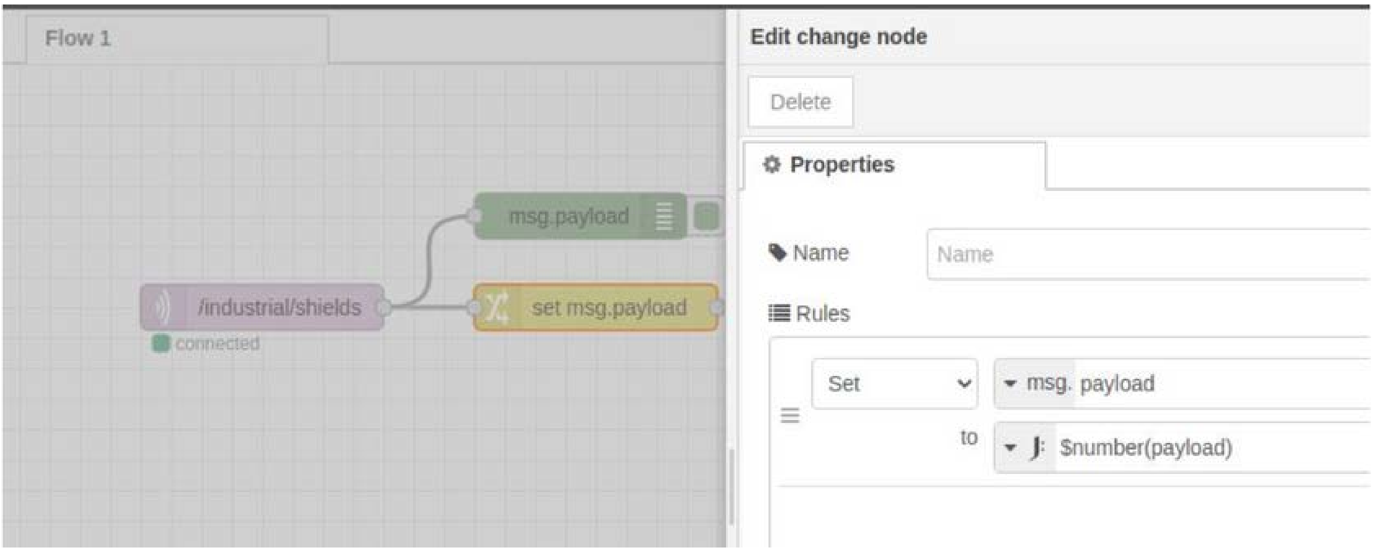
```
10.10.10.20:1880
```

Now, follow these steps to subscribe to the MQTT topic and send the information to the Influx database:

Subscribe to MQTT message from Raspberry Pi PLC controller

1. Add an IN MQTT node, set the **server** to your PC IP address, in our case: **10.10.10.60**, and the MQTT port by default: **1883**. Leave the rest of the parameters by default. Update. And Subscribe to the **topic: /industrial/shields.**

2. If we add a debug node just right after the MQTT node, you will realize that the message you receive is a string, not a number. So now, we are going to parse it to a number by adding a **change node** and **setting the msg.payload to a expression: $number(payload)**

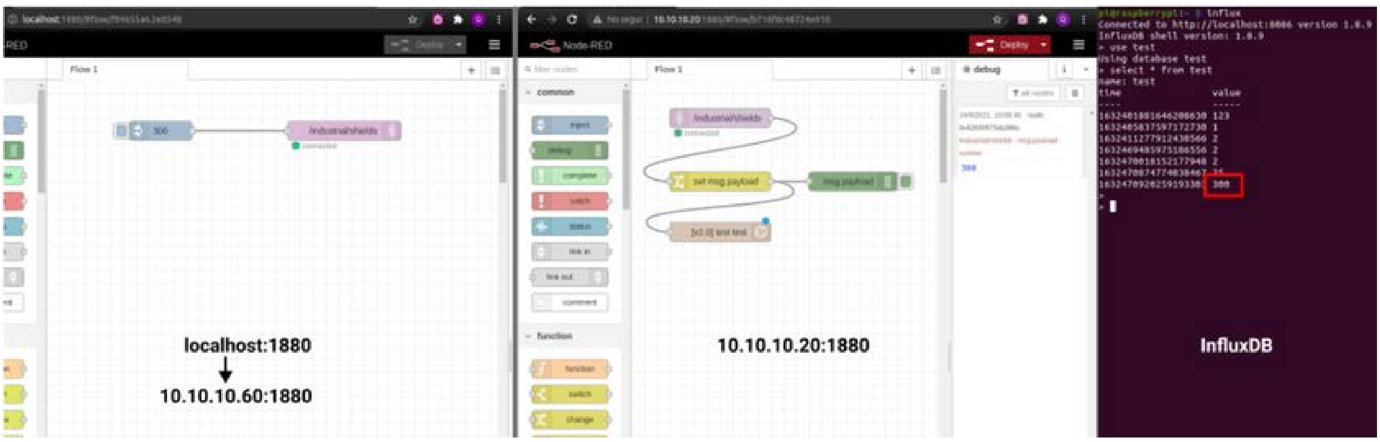


3. Wire a debug node after the change node, click on the inject node on the flow of our laptop in order to publish the **MQTT message**. And check that now you receive the information as a number, not a string.

4. Connect this to the **influxdb out node** that we already configured and send the information straight to the database!

5. Finally, go back to the **terminal window** and check that you got the information in the database. Your flow should be something like the picture below.

Now, it is your time! Import the **flow** and start playing with Node-RED!



Subscribe to MQTT message from Raspberry Pi PLC controller


```
[{"id":"b716fdc48724e610","type":"tab","label":"Flow 1","disabled":false,"info":""},
{"id":"d50d0c9f.31e858","type":"tls-
config","name":"","cert":"","key":"","ca":"","certname":"","keyname":"","caname":"",
"servername":"","verifyservercert":false},
{"id":"bc4ab5cb2a050021","type":"influxdb","hostname":"127.0.0.1","port":"8086","pro
tocol":"http","database":"test","name":"test","usetls":true,"tls":"d50d0c9f.31e858",
"influxdbVersion":"2.0","url":"http://localhost:8086","rejectUnauthorized":false},
{"id":"562ec1085cc9dbf1","type":"mqtt-
broker","name":"","broker":"10.10.10.60","port":"1883","clientid":"","usetls":false,
"protocolVersion":"4","keepalive":"60","cleansession":true,"birthTopic":"","birthQos
":"0","birthPayload":"","birthMsg":
{},"closeTopic":"","closeQos":"0","closePayload":"","closeMsg":
{},"willTopic":"","willQos":"0","willPayload":"","willMsg":{},"sessionExpiry":""},
{"id":"0e17644c4b3628b4","type":"influxdb
out","z":"b716fdc48724e610","influxdb":"bc4ab5cb2a050021","name":"","measurement":"t
est","precision":"","retentionPolicy":"","database":"","retentionPolicyV18Flux":"","
org":"Industrial                Shields","bucket":"test","x":680,"y":80,"wires":[]},
{"id":"0c42600975da386c","type":"debug","z":"b716fdc48724e610","name":"","active":tr
ue,"tosidebar":true,"console":false,"tostatus":false,"complete":"false","statusVal":
"","statusType":"auto","x":670,"y":120,"wires":[]},
{"id":"65e33f58d20fee33","type":"mqtt
in","z":"b716fdc48724e610","name":"","topic":"/industrial/shields","qos":"2","dataty
pe":"auto","broker":"562ec1085cc9dbf1","nl":false,"rap":true,"rh":0,"x":160,"y":80,"
wires":[["69f73915a01982c3","1ba5715c13107a23"]]},
{"id":"69f73915a01982c3","type":"change","z":"b716fdc48724e610","name":"","rules":
[{"t":"set","p":"payload","pt":"msg","to":"$number(payload)\t","tot":"jsonata"}],"ac
tion":"","property":"","from":"","to":"","reg":false,"x":400,"y":80,"wires":
[["0e17644c4b3628b4","0c42600975da386c"]]},
{"id":"1ba5715c13107a23","type":"debug","z":"b716fdc48724e610","name":"","active":tr
ue,"tosidebar":true,"console":false,"tostatus":false,"complete":"false","statusVal":
"","statusType":"auto","x":390,"y":120,"wires":[]}]
```

How to send WhatsApp messages with an industrial Raspberry PLC

Learn how to send WhatsApp messages using a Raspberry PLC and Node-RED

Can you imagine receiving an **alarm** from your industrial Raspberry Pi PLC to your phone via **WhatsApp** in **real time?**

That is possible because of the **Open-Source Raspberry Pi 4**. So in this tutorial, we are going to teach you how to **develop** a very simple **low-code program** using **Node-RED** for an open source PLC programming, so that you can be more competitive by streamlining your business processes.

Requirements

- Raspberry Pi PLC
- Either Ethernet cable or HDMI cable with an extra monitor.

Node-RED

Setting an alarm with a Raspberry Pi industrial PLC can be a very useful functionality to take control of your industrial environment. With our Open-Source Hardware, you will be able to **get WhatsApp messages** and **take control** of your company.

The first thing we need to do is to open the Node-RED from our Raspberry PLC. Install it if you do not have it yet from here.

As we can access either through **SSH or HDMI**, we will open our browser and just type:

```
localhost:1880 <--- If you are connected through HDMI
```

Node-RED

or

```
YOUR-IP-ADDRESS:1880 <--- If you are connected through SSH.
```

Once you are into Node-RED, let's develop our **alarm application** using **WhatsApp**!

Node-red-contrib-whatsapp-cmb

As we are going to use the **node-red-contrib-cmb nodes** from Node-RED to develop our **alarm system** for industrial control, we first need to **install the nodes**.

1. So, once in Node-RED go to the **top right hamburger menu > click on Manage Palette > Install > Type** and install it:

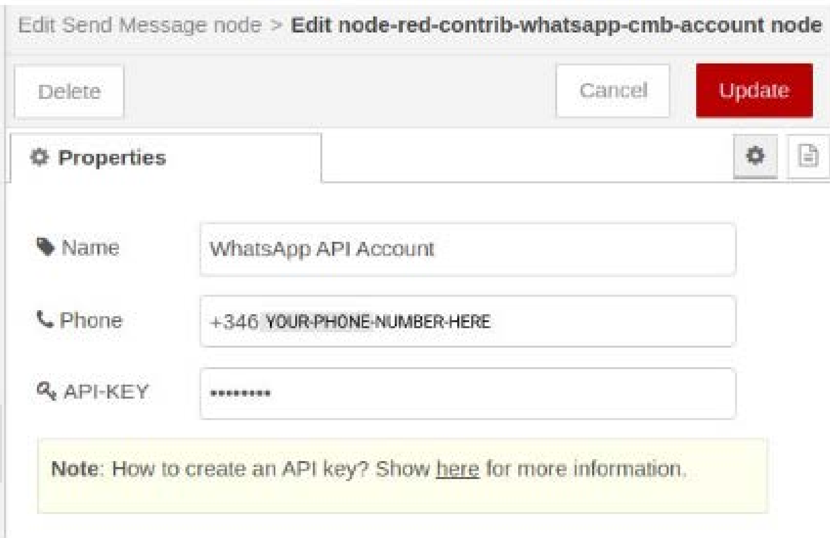
```
node-red-contrib-whatsapp-cmb
```

2. If you go to the **filter nodes search bar**, and **search 'WhatsApp',** you will see a new green node called ***Send Message*.** Drag and drop the node to the flow, and double click to explore it.

3. If any field of a Node-RED node is **red**, means that **it must be configured.** As the account field is red, click on the pen to edit.

The **properties' configuration node** will be displayed. You will need to fill in your phone and **API-KEY**. Follow the steps from [here] to create an **API-KEY**.

4. Once the **API-KEY** is created and you can interact with the **WhatsApp Bot**, it is very easy to test the example. First, fill in **your phone and API-KEY** in the ***Send Message*** node and leave it configured.

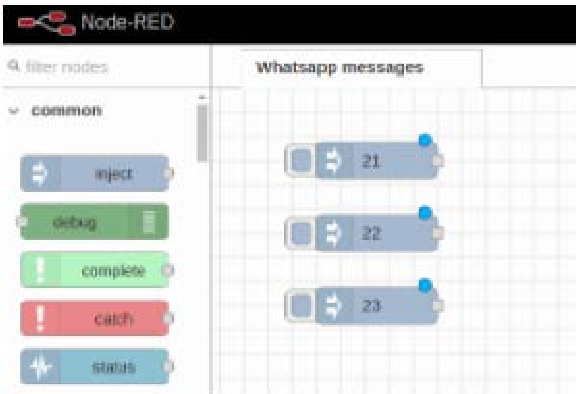


Getting inputs

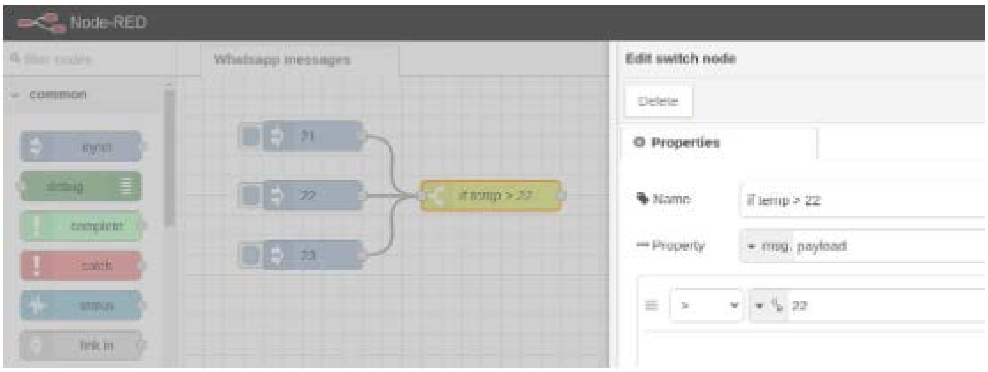
This application can be applied for multiple purposes and inputs can come from different places. The Raspberry PLC from Industrial Shields, as you can see here, can have up to **36 inputs**. That is perfect for our application, as we could get the values like this:

Even though it is a very simple way of getting a value from an input, we are going to do it easier with inject nodes.

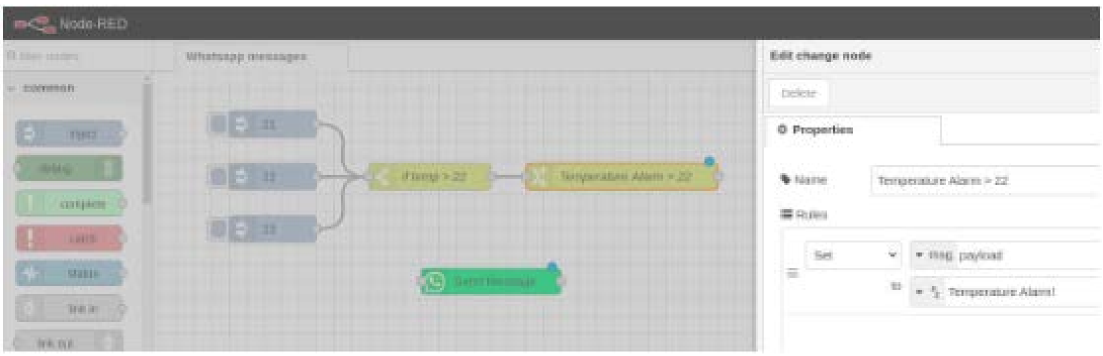
5. Add **three inject nodes** with **sample values**, like 21, 22 and 23 to send a **WhatsApp message** as a temperature alarm if the value is higher than 22.



6. Now, **add a switch node** to get the value if is **higher than 22** like this:

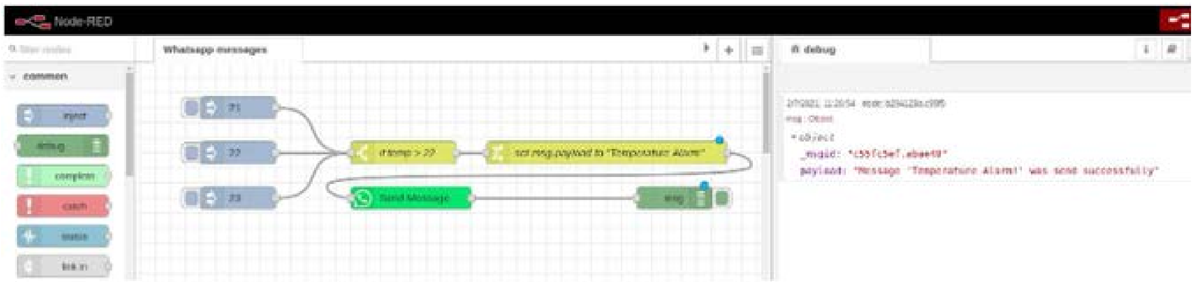


7. Then, **add a change node** and set the **msg.payload** to the message you want to be sent to your WhatsApp.



Getting inputs

8. Finally, wire the ***Send Message*** node to the change node, and add a debug node to get the debug messages.



9. Inject the **21, 22 and 23 messages,** and **get your alarm in your phone!**

Thank you very much for your time

Industrial Shields®

(+34) 938 760 191

info@industrialshields.com

Fabrica del Pont, 1-11
08272 Sant Fruitós de Bages (Barcelona)
España