Automation using Node-Red (iiot2k@gmail.com)

The redPlc Project

This little tutorial will tell you how to get started with Automation using Node-Red.
Of course, Node-Red is not a real-time environment and the performance
also depends on the hardware used.
Node-Red has some limitations like single task processing and the nodes only have one input.
But Node-Red has many nodes for supporting different protocols, databases and other applications.
For real-time processing there are applications like CODESYS.

Node-Red redPlc

- Implements Ladder Logic nodes according to IEC 61131-3.
- Is primarily used to easily implement automation tasks in Node-Red.
- Is written in JavaScript and works on all platforms where Node-Red runs.
- Is also ideal for implementing your ideas for automation in a short time and low cost.
- Is also ideal just for learning ladder logic diagrams in automation.

My goal was also to open a door to the automation world with redPlc for the Node-Red user.
Thinking in loops with Ladder Logic Diagrams is not easy at the beginning.
But over time you get the hang of it.
There is a lot of information and examples on the internet and on YouTube about Ladder Logic Diagram.

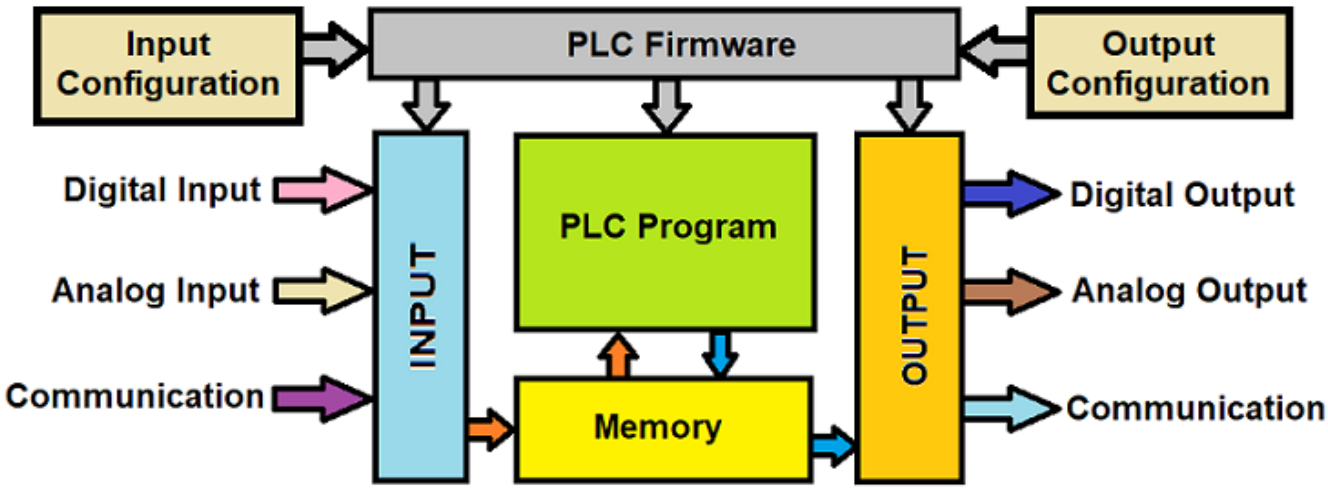
Another very interesting program is Factory I/O.
This allows you to graphically 3D simulate a real environment.
Unfortunately, the program is chargeable, but you can test it for 30 days.

How PLC works

A Programmable Logic Controller (PLC) is normally used for industrial automation.
The PLC hardware is designed for operation in harsh environments.
Special software from PLC manufacturer is required for programming.
Some industrial PLC hardware are relatively expensive and is not suitable for private use.
An alternative is a PLC based on Linux (e.g. Raspberry Pi) or Windows with Node-Red.

Wago PLC:

Here is a principle function diagram of a PLC.



These tasks are called in an endless cyclical loop (simplified):

- ***The PLC firmware reads all inputs and saves the data in memory.***
- ***The PLC firmware executes the user PLC program.***
- ***The PLC program reads the data, processes them and writes them into memory.***
- ***The PLC firmware writes the data from the memory to the outputs.***
- ***The PLC firmware performs a hardware check.***

PLC programming languages

The PLC program is not written in C/C++ or Python, but has its own programming languages.
All these PLC programming languages are defined in IEC 61131-3.

Textual PLC programming languages:

- Instuction List (IL).

```
LD  I0
ORN I1
ST  Q0
```

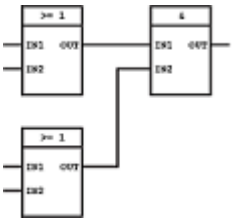
- Structured Text (ST).

```
IF Start = 1 THEN
    Motor := 1;
END_IF;
```

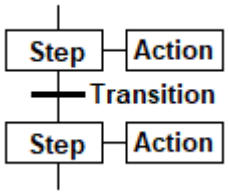
Graphic PLC programming languages:

- Ladder Logic Diagram (LD).

- Function Block Diagram (FBD)



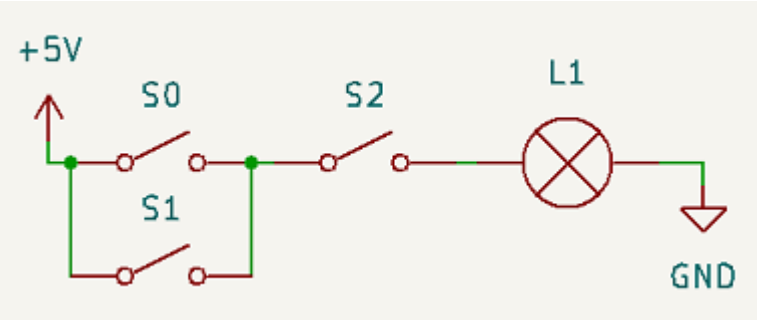
- Sequential Function Charts (SFC)



From electrical circuit to PLC

In the past, all control tasks were solved using relay circuits.
Changes and troubleshooting were very time-consuming.
Today, the control tasks are mapped in the PLC.
First we draw the circuit diagram and map it in the PLC.

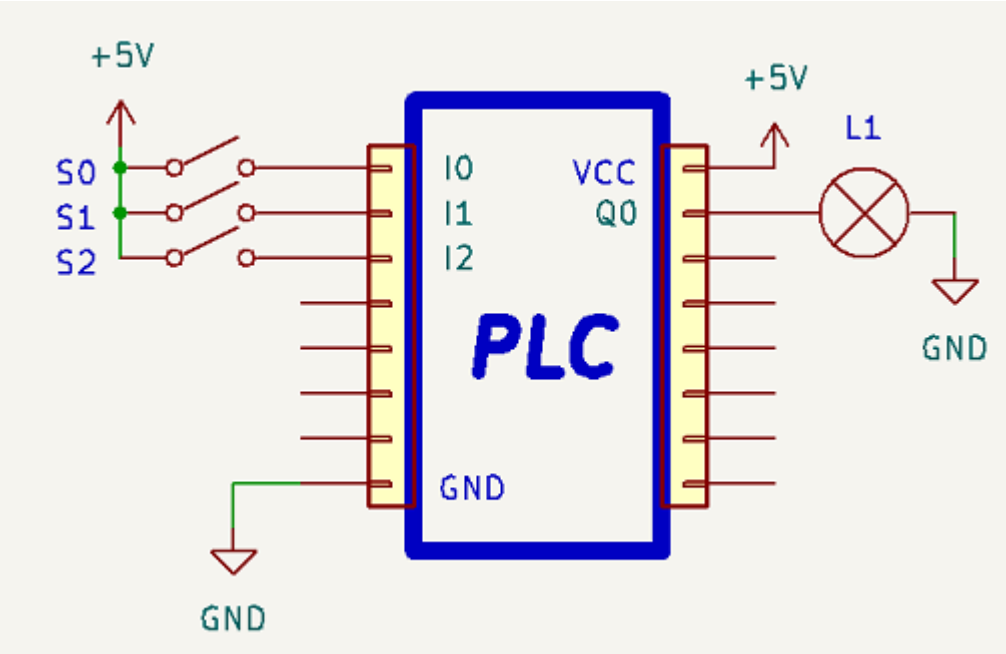
A simple circuit example:



S0, S1, S2 are simple switches.
L1 is a lamp.

Lamp L1 lights up when switches S0 **OR** S1 **AND** S2 are closed.

Connection to PLC:



Switches S0, S1, S2 are connected to the PLC inputs I0, I1, I2.
Lamp L1 is connected to PLC output Q0.
The logic when lamp L1 lights up depends on the PLC program.

Instuction List (IL) and Ladder-Logic-Diagram (LD)

You can write our electrical circuit above as Instuction List (IL):

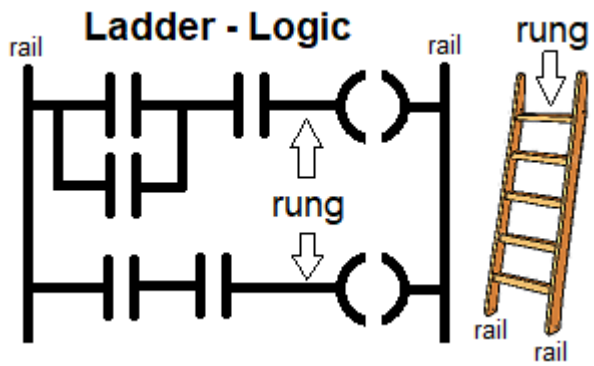
```
LD  I0
OR  I1
AND I2
ST  Q0
```

LD -> LOAD VARIABLE **I0** into accumulator
OR -> OR accumulator with VARIABLE **I1**
AND -> AND accumulator with VARIABLE **I2**
ST -> STORE accumulator to VARIABLE **Q0**

> Let's remember, the PLC program runs in an endless loop.
> The PLC program does not make any direct read/write access to inputs and outputs.
> Only the PLC memory is accessed.
> Therefore I0, I1, I2 and Q0 are the identifiers of memory.
> Symbolic names such as S0, S1, S2 and L1 can also be used for I0, I1, I2 and Q0.
> In a configuration list in the PLC, the variables are assigned to the inputs/outputs.
> At the beginning of the PLC program, the variables are updated from the inputs.
> At the end of the PLC program, the outputs are updated by the variables.

Instruction List **(IL)** is an assembly-like PLC programming language.
IL has a lot of commands.
Large IL programs are difficult to maintain and extend.
Therefore, IL will be removed next from the IEC 61131-3.
We use Ladder-Logic-Diagram **(LD)** PLC programming in this tutorial.

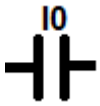
Where does the name Ladder-Logic-Diagram come from ?



Ladder-Logic-Diagram **(LD)** looks like a real ladder.
We will discuss the meaning of the symbols in the LD in the next chapters.

Ladder-Logic-Diagram (LD) Symbols

In LD, inputs are represented with this symbols and called Contact:



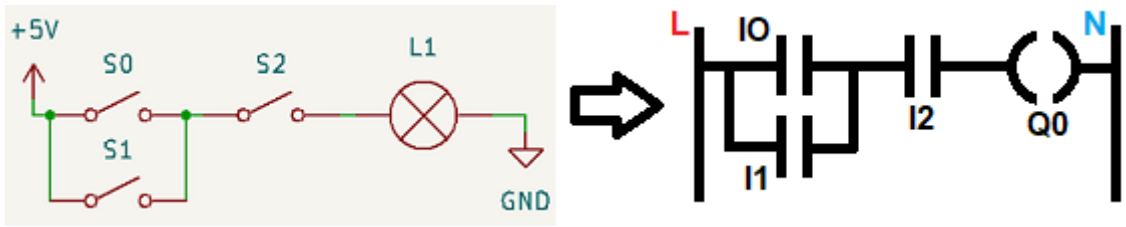
This contact works as non inverting logic **(Normaly Open NO)**.
If the variable I0 has the status 1, then the output state is same as the input state.
If the variable I0 is 0, the output is always 0.

This contact works as inverting logic **(Normaly Close NC)**.
If the variable I4 has the status 0, then the output state is same as the input state.
If the variable I4 is 1, the output is always 0.

In LD, outputs are represented with this symbol and called Coil:

The symbol shows the output variable Q0.
The variable Q0 is set with the status of the input.
Because Q0 is also a variable, it can be read like an input as a contact.

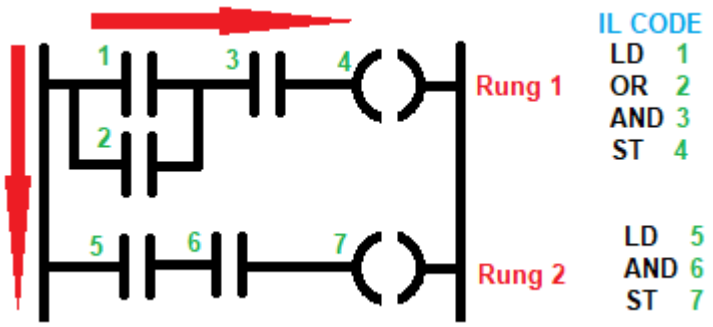
Our electrical circuit above as LD:



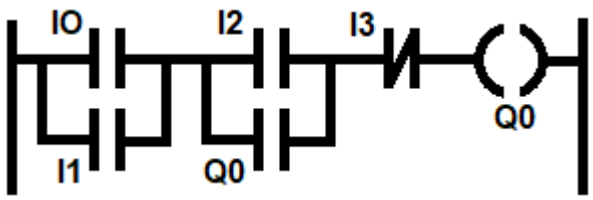
For a logical operation you have to connect the LD elements together.
As in electrical system, the LD elements are connected to voltage rails.
There is always voltage (or logical 1) on the left rail (L).
The right rail is always connected to ground or neutral conductor (N).
Each branch is called a **rung** or **network**.
A rung begins with inputs (contacts) and ends with outputs (coils).
A LD can have multiple rungs or networks.
LD symbols may vary by manufacturer.
Each manufacturer has its own LD editor and compiler.

LD Sequence of Operations (SOO)

The LD compiler analyzes the diagram and generates an intermediate code or IL code.
The compiler follows a sequence of operations.
The compiler starts the analysis from the first element in the rung and up to the last element in the rung.
After that, the compiler parses the next rung.
The PLC program runs then accordingly sequence of operations (SOO).



Example, how is this LD translated to IL?




```
LD      I0      LOAD IO in ACC0
OR      I1      OR ACC0 with I1
AND(    I2      LOAD I2 in ACC1 and remember AND operation
OR      Q0      OR ACC1 with Q0
)               AND ACC0 with ACC1
AND     I3      AND ACC0 with (NOT I3)
ST      Q0      STORE ACC0 in Q0
```

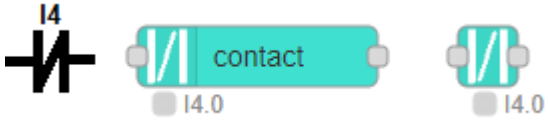
This is a typical logic to start and stop a process (e.g. a motor).
I0 **OR** I1 must be closed to start the process (can be key switches for security).
If I2 is switched on, Q0 is activated via closed I3.
If I2 then switched off, Q0 holds self on next cycle because of Q0 **OR** I2 query.
If you switch I3 on, Q0 is then deactivated.
I2 and I3 works as START and STOP and can be push button switches.
Don't forget, the program runs in a endless cycle loop.

Ladder-Logic-Diagram in Node-Red with redPlc

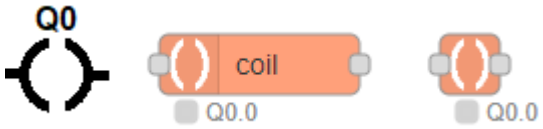
Ladder-Logic-Diagrams looks similar to Node-Red flows.
So I got the idea of implementing LD in Node-Red.
Of course, to work some challenges had to be solved in the Node-Red node programming.

Normally Open (NO) Contact Node in Node-Red (right node with hidden label):

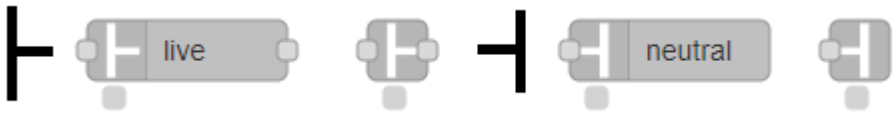
Normally Closed (NC) Contact Node in Node-Red (right node with hidden label):



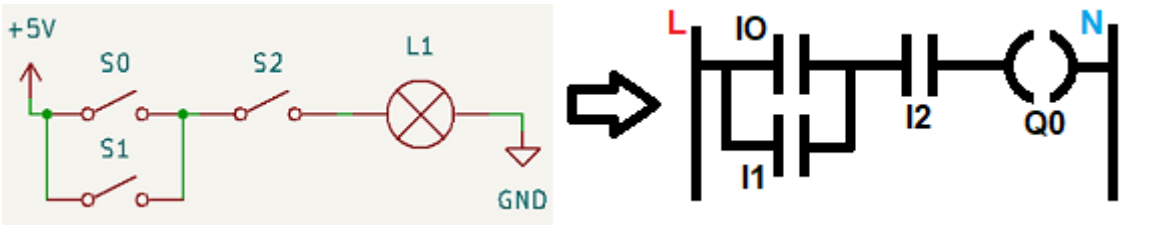
Coil Node in Node-Red (right node with hidden label):

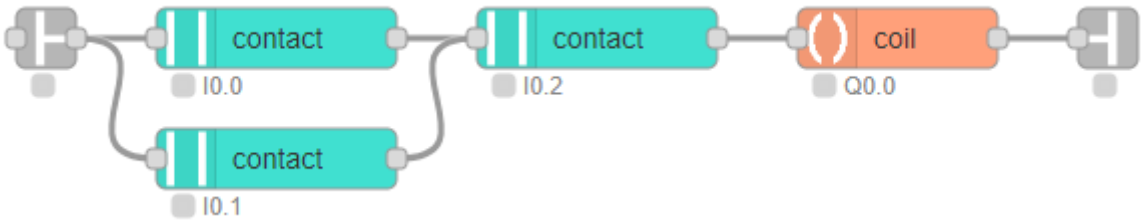


Live and Neutral Rail Node in Node-Red (right node with hidden label):



Our electrical circuit above as LD in Node-Red:



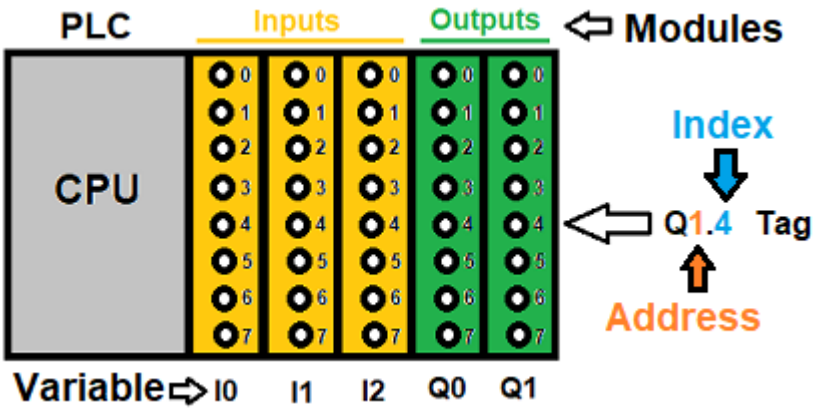


Variables in redPlc

Variable	Function	Array Of	Created by
I	Digital Input	Boolean	Digital Input Modules
Q	Digital Output	Boolean	Digital Output Modules
M	Digital Memory	Boolean	Memory Node
IA	Analog Input	Number	Analog Input Modules
QA	Analog Output	Number	Analog Output Modules
MA	Analog Memory	Number	Memory Node
C	Counter	Boolean/Number	Counter Node
T	Timer	Boolean/Number	Timer Node
FF	Flip-Flop	Boolean	Flip-Flop Node

Variable Notation in redPlc:

Let's take a look at the input/output modules of a modular plc:



> Variables are stored as arrays in Node-Red global context memory.
> Each input/output port has a unique Variable name.
> You can freely assign an Variable with select address number.
> **Q1.4** means Digital Output Variable **Q** Address **1** Index **4**.
> Maximum Address is **999** and Index is **63**.
> Unused array elements are set to undefined.
> Modules can also have multiple variables.
> This name syntax and memory handling applies to **redPlc**,
> it can be different for other PLC manufacturers.
> In some PLCs, a symbolic name can also be used for tag **I0.3** (e.g. **START**).

Array Structure of Counter, Timer and Flip-Flop

Counter:

Index	Name	Datatype	Function
0	CV	Number	Counter Value
1	PV	Number	Counter Preset
2	QU	Boolean	Counter Done Up
3	QD	Boolean	Counter Done Down
4	R	Boolean	Counter Reset
5	LD	Boolean	Counter Load Preset
6	CD	Boolean	Counter Down Input

Timer:

Index	Name	Datatype	Function
0	ET	Number	Timer Value (ms)
1	PT	Number	Timer Preset (ms)
2	Q	Boolean	Timer Done
3	R	Boolean	Timer Reset
4	TT	Boolean	Timer Timed (run)

Flip-Flop:

Index	Name	Datatype	Function
0	Q	Boolean	Flip-Flop Output
1	R	Boolean	Flip-Flop Reset
2	S	Boolean	Flip-Flop Set

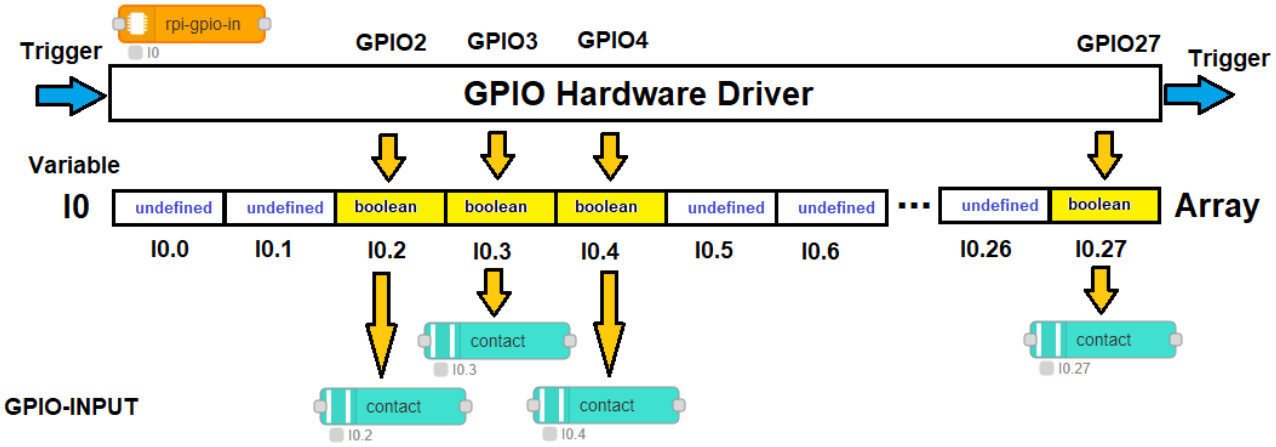
Connection to the Outside World: Modules

If you need modules for your hardware, please contact me: iiot2k@gmail.com

Nodes that read or write data from the hardware are called **module** nodes in redPlc.
This module nodes also creates the I, Q, IA and QA variables.

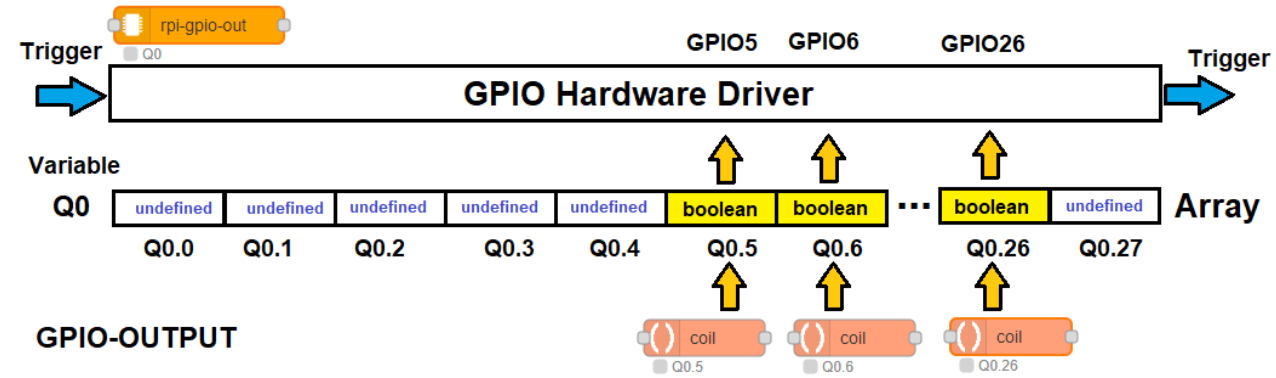
Example Raspberry Pi digital input module:

- Raspberry Pi pins 2,3,4 and 27 are selected as input.
- Module **rpi-gpio-in** creates boolean array **I0** with 28 elements (index 0..27).
- Selected inputs are set initial to *False* in the array, not selected inputs are set to **undefined**.
- In the case of an input message, the gpio inputs are read and written to the array elements.
- To prevent a high system load, the inputs are read after a time cycle.



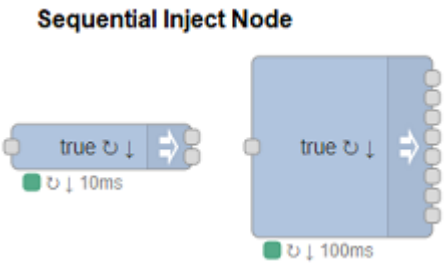
Example Raspberry Pi digital output module:

- Raspberry Pi pins 5,6 and 26 are selected as output.
- Module **rpi-gpio-out** creates boolean array **Q0** with 28 elements (index 0..27).
- Selected outputs are set initial to **false or true** in the array, not selected outputs are set to **undefined**.
- In the case of an input message, the array elements are read and written to gpio pins.
- To prevent a high system load, the outputs are written after a time cycle.
- In each cycle it is also checked whether the array elements have changed.



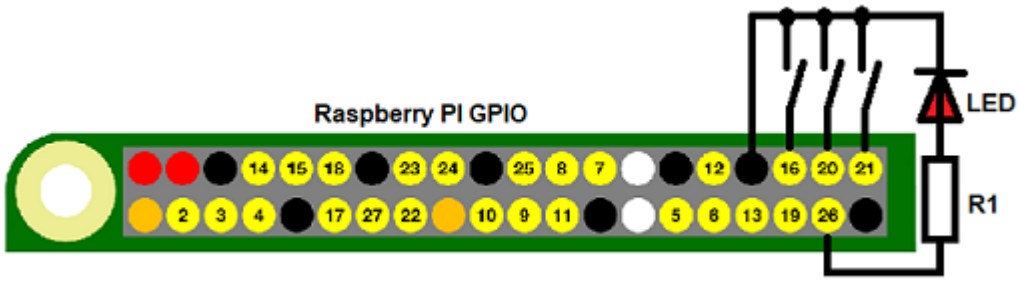
Trigger Engine for redPLC

The LD nodes shown must be triggered cyclically with *True* for processing.
This can be done with a standard Node-Red inject node.
As explained in previous chapter, Sequence of Operations (SOO) is used in LD.
We need also a special inject node for this:



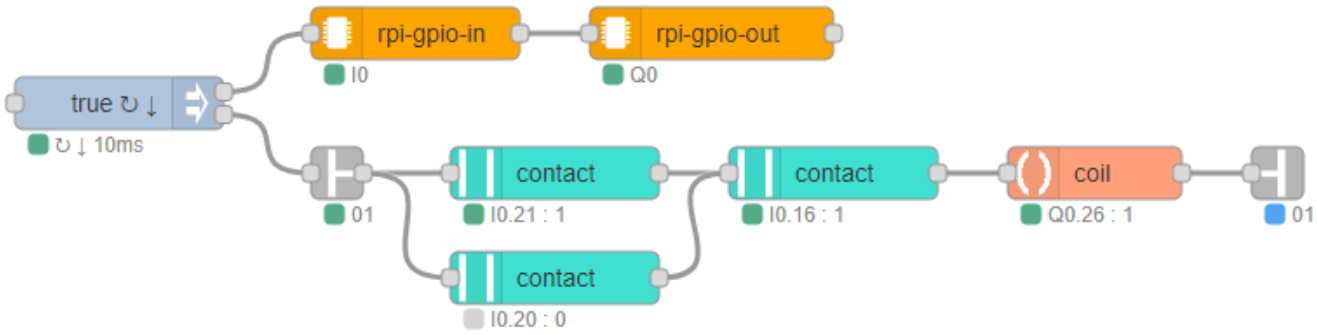
This sequential inject node has multiple outputs (selectable 1..45).
True is sent cyclically to the outputs sequentially, starting with the first.
The cycle time can be selected (default 10ms) and should be adjusted depending on the scope of the flow.
The cycle can be stopped with a *False* message at the node input.
A delayed start of the cycle can also be selected (default 200ms).

I have changed the index numbers (pin numbers) according to my test circuit:



Select in module node **rpi-gio-in** GPIO16, GPIO20 and GPIO21 as pullup inputs.
Select in module node **rpi-gio-out** GPIO26 as output (R1 depends on used LED, ~220 Ohms).

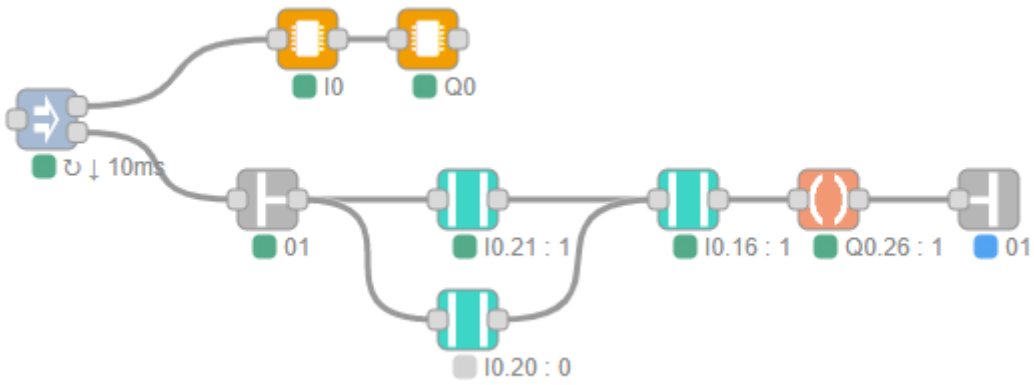
The complete LD with sequential inject node and Raspberry PI module nodes:



Complete redPlc Example and Compact Nodes

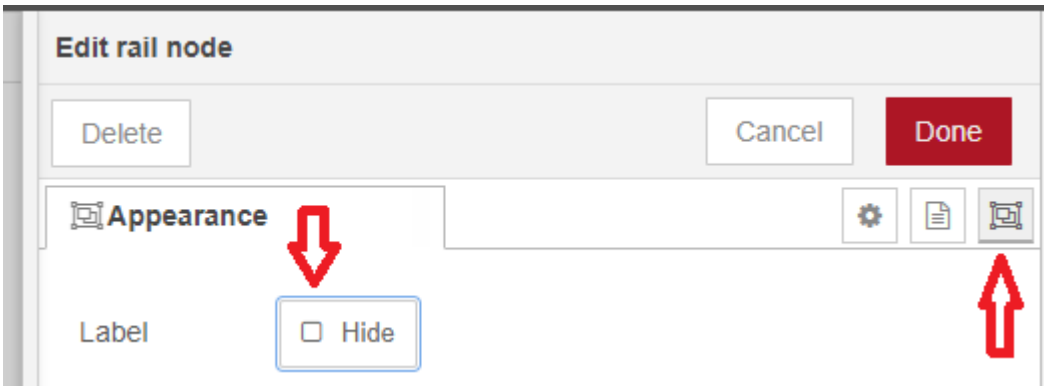
The modules **rpi-gio-in** and **rpi-gio-out** are connected in series at first output of sequential inject node.
The LD nodes are connected to second output of sequential inject node.
If more rungs are required, the number of outputs in the sequential inject node can be increased.
This ensures the Sequence of Operations (SOO).
The LD nodes shows the signal flow with the status icons (grey for false, green for true, red for error).
The states of the variables are displayed in the status text or the cause of error.
The live and neutral rail nodes have no function and is for documentary purposes only.
But I recommend using these to indicate that it is an LD.

In Node-Red it is also possible to display the nodes without a label:



This representation is more compact.
To display the label as tooltip, move the cursor above the node.

Displays icons without label, click in node **Properties** dialog **Appearance** icon
and deselect checkbox **Label**.



In Node-Red settings, displaying labels can be switched off in general.
Deselect in settings checkbox **Show label of newly added nodes**.

The next chapters show all redPlc nodes with their variants.
At the end are some redPlc examples.
If the ladder logic element has multiple inputs/outputs, the asterisk (*) indicates them in the node.

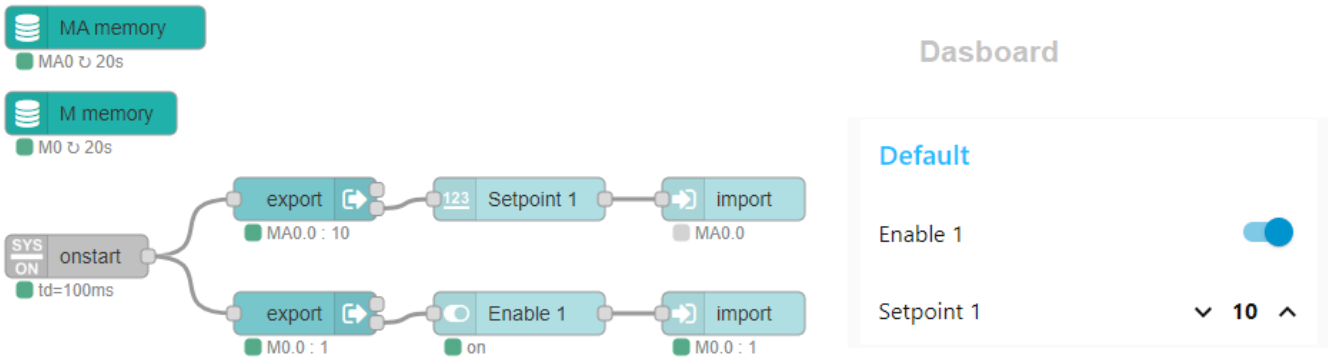
Memory Node

Creates memory variable.

The Memory Node creates the **M** and **MA** variables.
These variables store the intermediate values of operations (e.g. coil, math, export/import...).
By default, these memory variables are initialized to false (M) and 0 (MA).
These default values can be changed in the node properties dialog.
If persistent mode is not switched on, the stored values are lost when Node-Red stops.
The persistent mode saves digital or analog memory variables to file.
This allows you to make variables persistent.
On start of Node-Red the variables are created and initialized from file.
On stop of Node-Red the variables are saved to file.
If the node is deleted, the file is also deleted.
Frequent updates can shorten the life of SD memory.
In order to reduce write access, the variables are saved after a time cycle and if they change.
Update cycle time can be selected from 1s to 200s (default 20s).
The variables are stored in the home directory `~/.redplc/persistent/`.

Example:

This example stores the values of the dashboard elements to persistent storage.



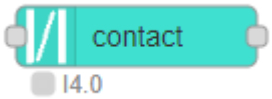
How it works ?

1. The **persistent nodes** initialize the MA0 and M0 variables from file on start.
2. **onstart node** sends once *True* to all **export nodes** on start.
3. **export nodes** reads the variables and sends the data to the Dashbord elements.
4. The Dashbord elements initialize own data and view.
5. If the user changes the Dashbord element, data is send to **import** nodes.
6. The **import** nodes saves the data form Dashbord element to variables.
7. On update cycle the **persistent nodes** recognize a change and save the variable to file.

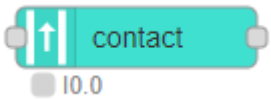
Contact Node

Normally Open (NO) Contact Node (also called *Examine IF Closed XIC*):
Input is passed through to output if variable is *True*. Else *False* is send to output.

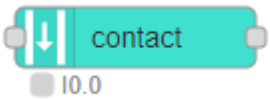
Normally Closed (NC) Contact Node (also called *Examine IF Open XIO*):
Input is passed through to output if variable is *False*. Else *False* is send to output.



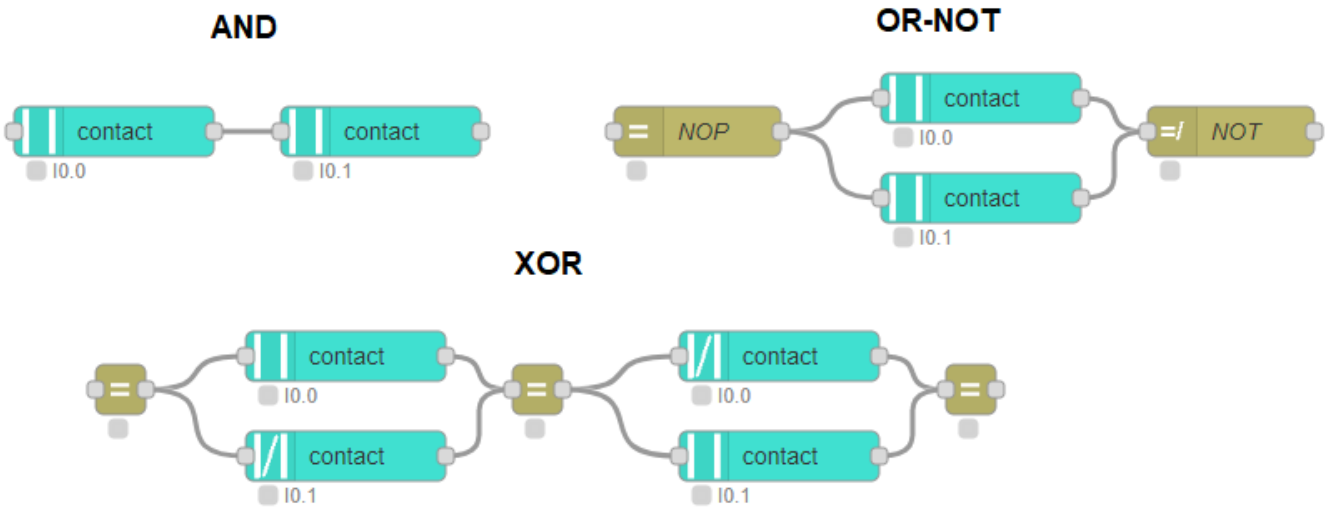
Positve Transition Sense Contact Node:
Input is passed through to output if variable changes from *False* to *True*. Else *False* is send to output.



Negative Transition Sense Contact Node:
Input is passed through to output if variable changes from *True* to *False*. Else *False* is send to output.

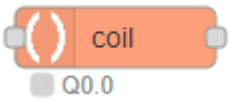


Example (function node are explained below):

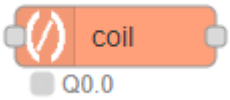


Coil Node

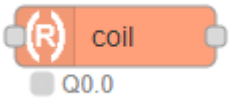
Store Coil Node:
Input is stored in variable. Input is also send to output.



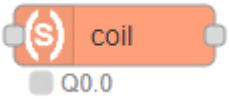
Store negated Coil Node:
Input is stored negated in variable. Input is also send to output.



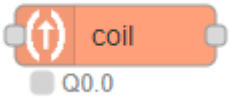
Reset Coil Node:
Variable is set to *False* in input is *True*. Input is also send to output.



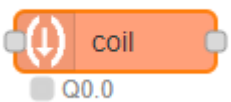
Set Coil Node:
Variable is set to *True* in input is *True*. Input is also send to output.



Positve Transition Coil Node:
Variable is set to *True* if input changes from *False* to *True*. Input is also send to output.



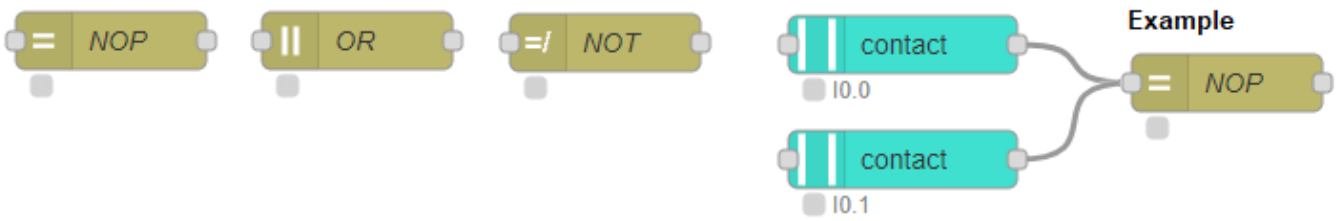
Negative Transition Coil Node:
Variable is set to *True* if input changes from *True* to *False*. Input is send to output.



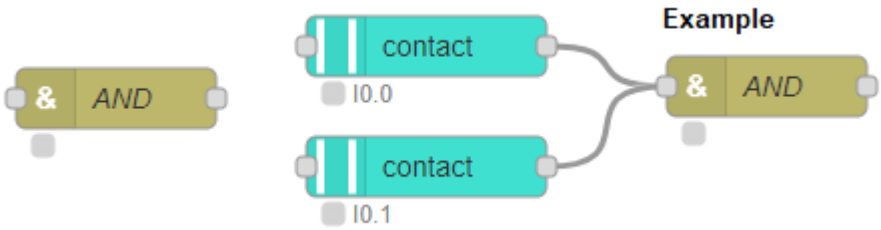
Digital Function Node

Digital Function Nodes works like Function Block Diagram (FBD).
You can connect one or more LD nodes to input.

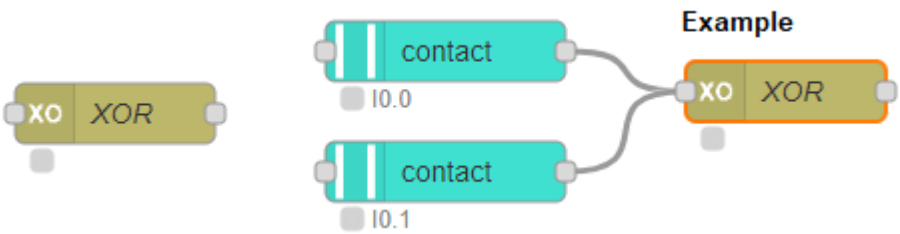
NOP (No Operation), OR and NOT:
Logical OR inputs. Output payload is *True* if any inputs are *True*.
NOP and OR have the same function, use NOP for conjunction of nodes.
NOT negates OR result.



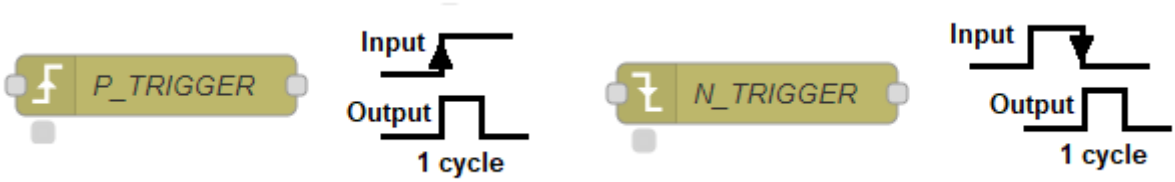
AND:
Logical AND inputs. Output payload is *True* if all inputs are *True*.
Needs min. 2 nodes wired on input for valid work.



XOR:
Logical XOR inputs. Output payload is *True* if inputs are different.
Needs min. 2 nodes wired on input for valid work.



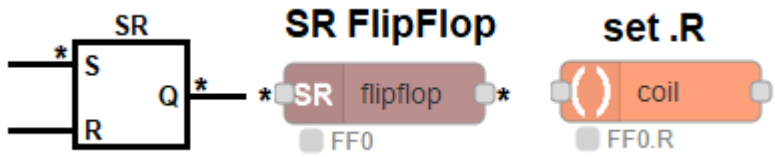
P_TRIGGER, N_TRIGGER:
Logical OR inputs and sense for positive or negative transition.



Flip-Flop Node

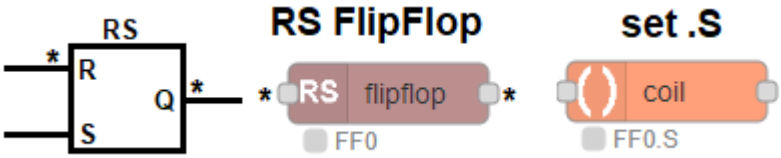
This node is used for store digital states.
Each FlipFlop has its order of operation sequence priority.

SR FlipFlop (set dominant):



1. **Set:** If input payload is *True*, output payload and flag **Q** is set to *True*.
2. **Reset:** If flag **R** is *True*, output payload and flag **Q** is set to *False*.

RS FlipFlop (reset dominant):

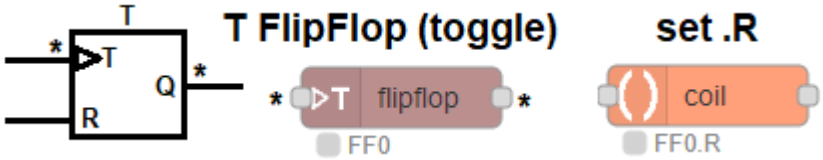


1. **Reset:** If input payload is *True*, output payload and flag **Q** is set to *False*.
2. **Set:** If flag **S** is *True*, output payload and flag **Q** is set to *True*.

L FlipFlop (latch):

1. **Reset:** If flag **R** is *True*, output payload and flag **Q** is set to *False*.
2. **Set:** If input payload changes from *False* to *True*, output payload and flag **Q** is set to *True*.

T FlipFlop (toggle):

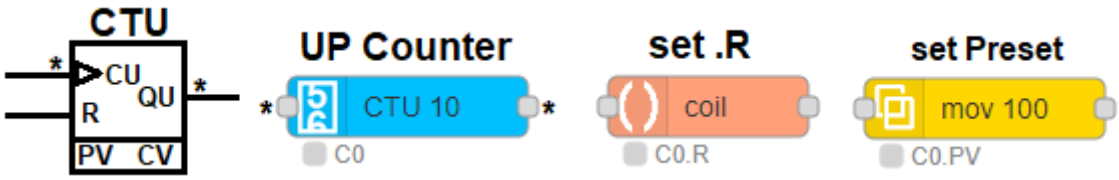


1. **Reset:** If flag **R** is *True*, output payload and flag **Q** is set to *False*.
2. **Toggle:** If input payload changes from *False* to *True*, output payload and flag **Q** toggles.

Counter Coil Node

This node counts on input pulse up/down (**PV** = Preset Value, **CV** = Counter Value).
If preset **PV** is changed to <= **0** then preset **PV** is set to entered default preset.

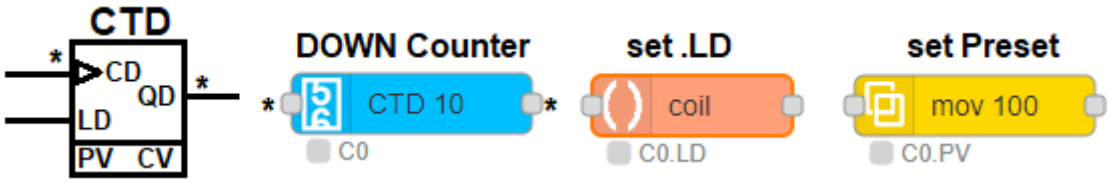
CTU Up-Counter:



1. **Reset:** If flag **R** is *True*, counter **CV** is reset to **0**.
2. **Count up:** If input payload changes from *False* to *True*, the counter **CV** increments by one.

Output payload and flag **QU** is set *True* if counter **CV** >= preset **PV**.

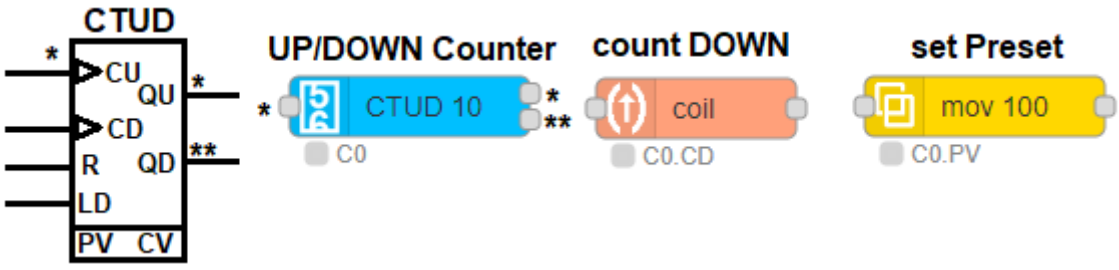
CTD Down-Counter:



1. **Load:** If flag **LD** is *True*, counter **CV** is load with preset **PV**.
2. **Count down:** If input payload changes from *False* to *True*, the counter **CV** decrements by one.

Output payload and flag **QD** is set to *True* if counter **CV** <= **0**.

CTUD Up-Down-Counter:



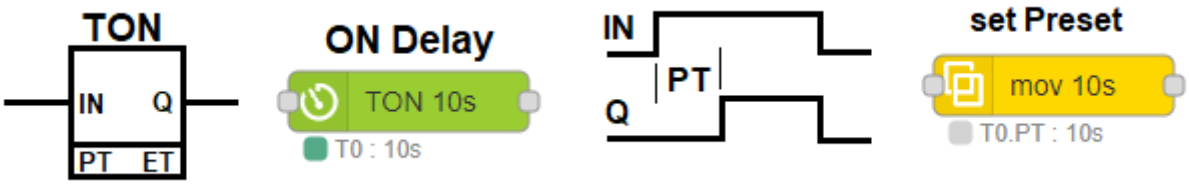
1. **Reset:** If flag **R** is *True*, counter **CV** is reset to **0**.
2. **Load:** If flag **LD** is *True*, counter **CV** is load with preset **PV**.
3. **Count up:** If input payload changes from *False* to *True*, the counter **CV** increments by one.
4. **Count down:** If flag **CD** changes from *False* to *True*, the counter **CV** decrements by one.

Output-1 payload and flag **QU** is set *True* if counter **CV** >= preset **PV**.
Output-2 payload and flag **QD** is set *True* if counter **CV** <= **0**.

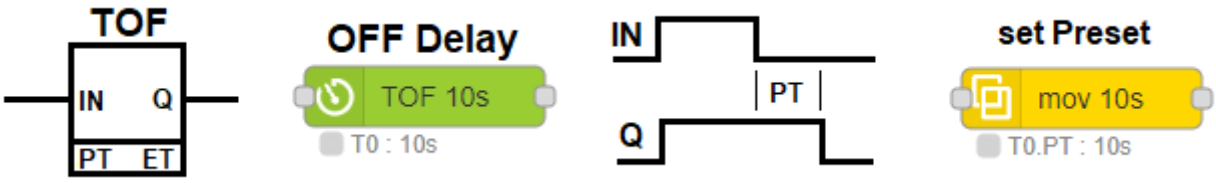
Timer Coil Node

Timers delays input signals or generate pulses (**PT** = Preset Time, **ET** = Elapsed Time).
Changes to **PT** efects on start/restart of timer.
If preset **PT** is changed to <= **0** then preset **PT** is set to entered default preset.

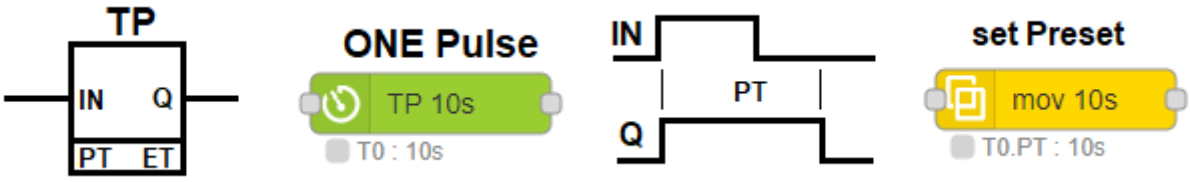
TON On Delay:



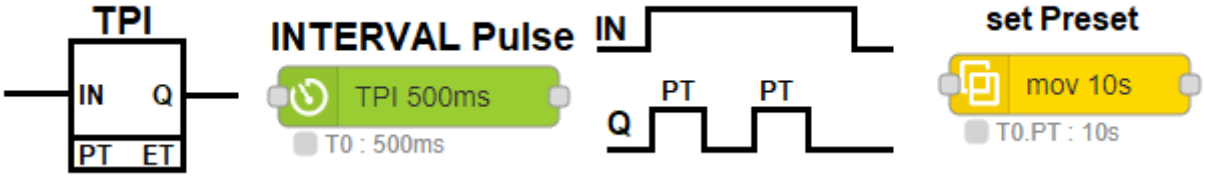
TOF Off Delay:



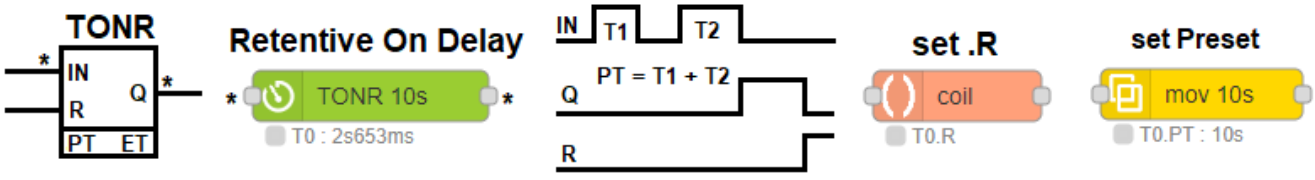
TP One Pulse:
If timer is started, input is ignored until preset time **PT** is reached.



TPI Interval Pulse:

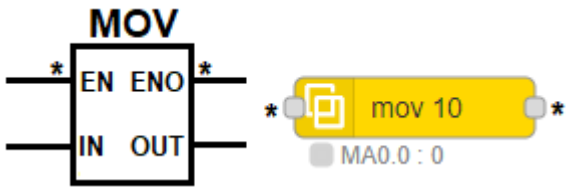


TONR Retentive On Delay:
R sets output **Q** to *False* and resets elapsed time **ET** to 0.



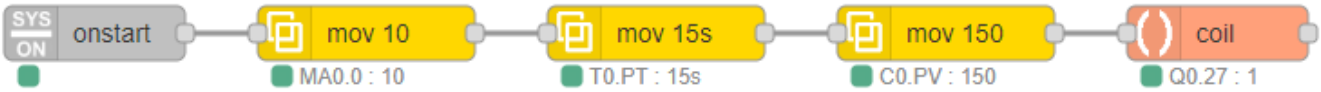
Move Analog Node

Moves (copies) source variable/constant values to target variables.



Input of move can be **IA, QA, MA, T, C, Number-Constant, Math-Constant, Timer-Constant**.
Output of move can be **QA, MA, Timer-Preset and Counter-Preset**.
Move operation is made on enable input **EN** is *True*.
Enable input **EN** is send to enable output **ENO**.
Input **Timer-Constant** is converted to ms.

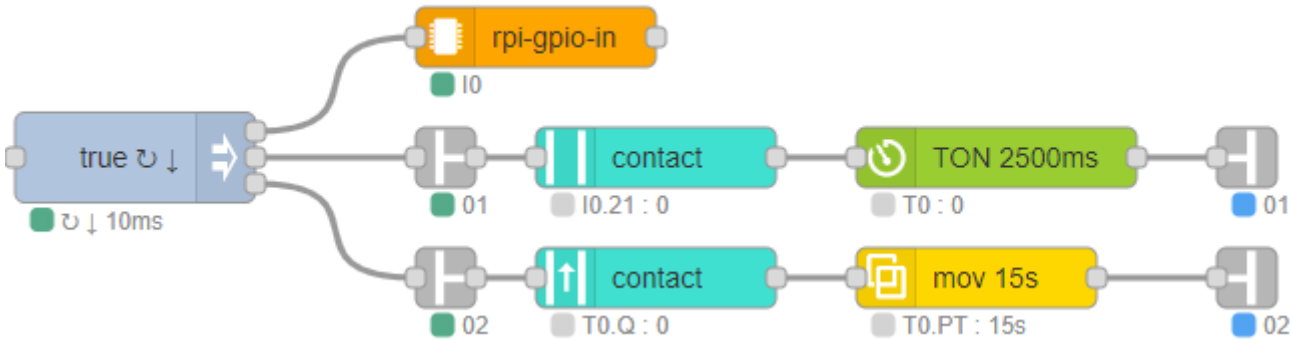
Example 1:



The redPlc node **onstart** sends on start once a *True* message.
This can be used to initialize.

1. Analog memory **MA0.0** is set with 10.
2. Timer **T0.PT** preset is set to 15s.
3. Counter **C0.PV** preset is set to 150.
4. Digital output **Q0.27** is set.

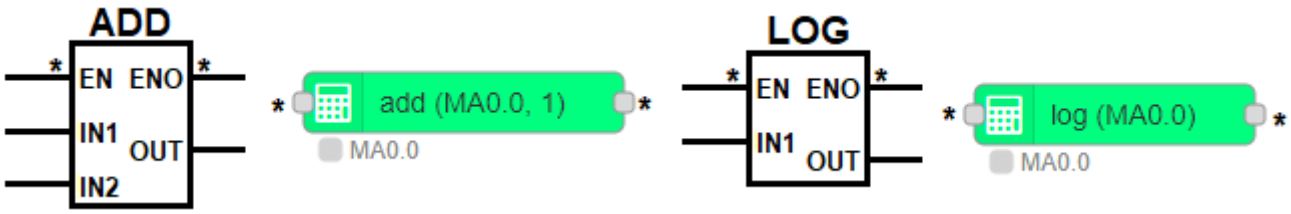
Example 2:



Timer **T0** (On Delay) has a default preset of 2500ms.
Rung 01: The Raspberry Pi digital input GPIO27 starts timer T0 on closed.
Rung 02: If timer T0 timed out, contact T0.Q sends *True* to move node and sets **T0** preset to 15s.
Contact T0.Q on rung 2 is a pulse type, because only once *True* is send on cycle.
If input GPIO27 closed again, the timer **T0** preset time is 15s.

Math Analog Node

Calculates variables/constant values with other variables/constant values and stores in variable.



Depending on the operation, one or two operands are used.
Inputs of math can be **IA, QA, MA, T, C, Number-Constant, Math-Constant, Timer-Constant**.
Output of math can be **QA, MA, Timer-Preset and Counter-Preset**.
Calculation operation is made on enable input **EN** is *True*.
Enable input **EN** is send to enable output **ENO**.
On calculation error, *False* is send to output **ENO**.
If you connect math nodes in series, the arithmetic operations are executed in this order.
Before the arithmetic operations, the variables can be set with the move node.

Arithmetic Operations:
add-addition, **sub**-subrtact, **mul**-multiple, **div**-divide, **mod**-remainder, **inc**-increment, **dec**-decrement.

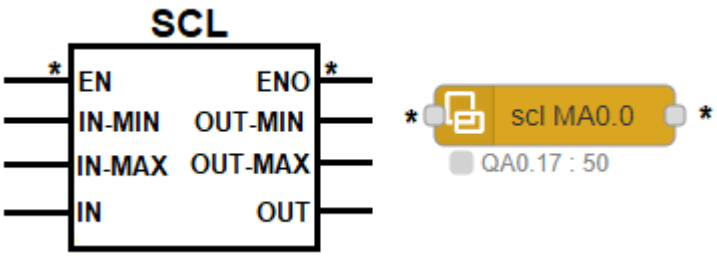
Trigonomic Operations:
pow-power, **acos**-arccosine, **acosh**-hyperbolic arccosine, **asin**-arcsine, **asinh**-hyperbolic arcsine,
atan-hyperbolic arctangent, **cbrt**-cube root, **cos**-cosine, **cosh**-hyperbolic cosine,
exp-exponent to euler's constant, **log**-natural logarithm, **log10**-base-10 logarithm, **log2**-base-2 logarithm,
sin-sine, **sinh**-hyperbolic sine, **sqrt**-square, **tan**-tangent, **tanh**-hyperbolic tangent,
torad-convert to radians, **todeg**-convert to degrees,

Statistical Operations:
abs-absolute, **min**-minimum, **max**-maximum,
round-rounded to the nearest integer, **ceil**-smallest integer greater than or equal,
floor-largest integer less than or equal, **trunc**-removing any fractional digits,
random1-random number one range (0..n), **random2**-random number two range (m..n).

Mathematical Constants:
PI - This is the ratio of the circumference of a circle to its diameter.
E - This is Euler's number, the base of natural logarithms.
LN2 - The natural logarithm of 2.
LN10 - The natural logarithm of 10.
LOG2E - The base-2 logarithm of e.
LOG10E - The base-10 logarithm of e.
SQRT2 - The square root of 2.
SQRT1_2 - The square root of 0.5, or, equivalently, one divided by the square root of 2.

Scale Analog Node

Scales analog values by given input/output limits and stores in variable.

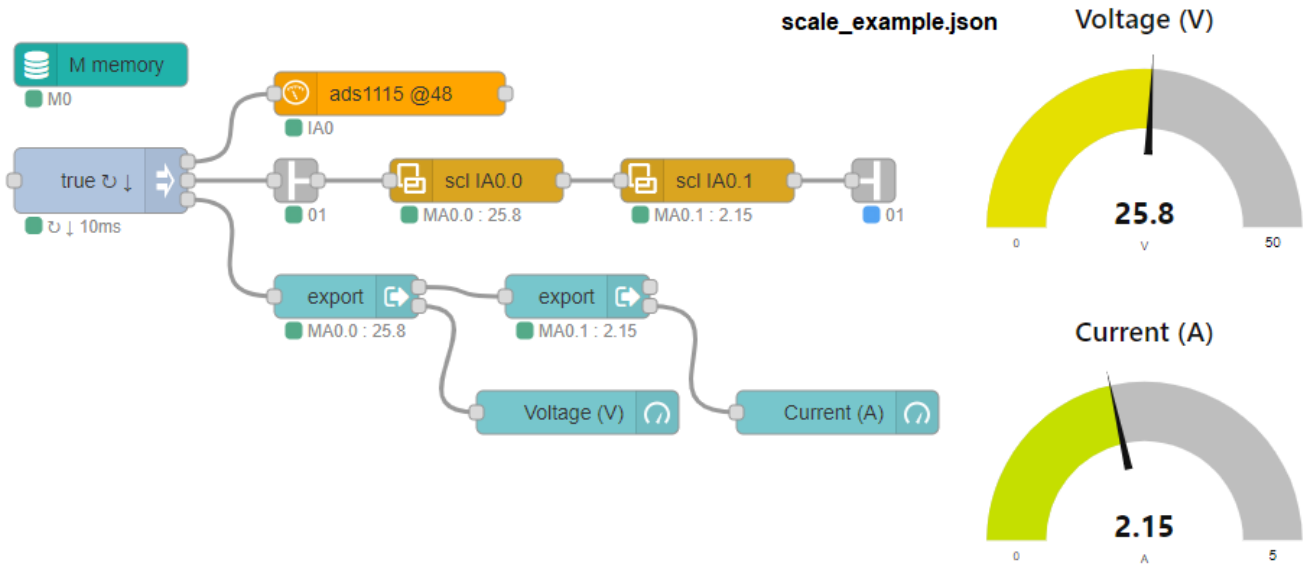


Input of scale can be **IA, QA, MA**.
Output of scale can be **QA, MA**.
Scale operation is made if enable input **EN** is *True*.
Enable input **EN** is send to enable output **ENO**.
On scale error, *False* is send to output **ENO**.
Input-Max must greater then Input-Min.
Output-Max must greater then Output-Min.
Input value is also called raw value. Output value is also called engineering value.

> **Scale Calculation Formula:**
>
> **Factor** = (Output-Max - Output-Min) / (Input-Max - Input-Min)
> **Offset** = Output-Min - (Input-Min * **Factor**)
> **Output** = **Input** * **Factor** + **Offset**

Example:

This example reads voltage and current from the analog to digital converter,
scales the values to engineering values.
In Node-Red Dashboard, these are displayed with the export node in a gauge.



Compare Analog Contact Node

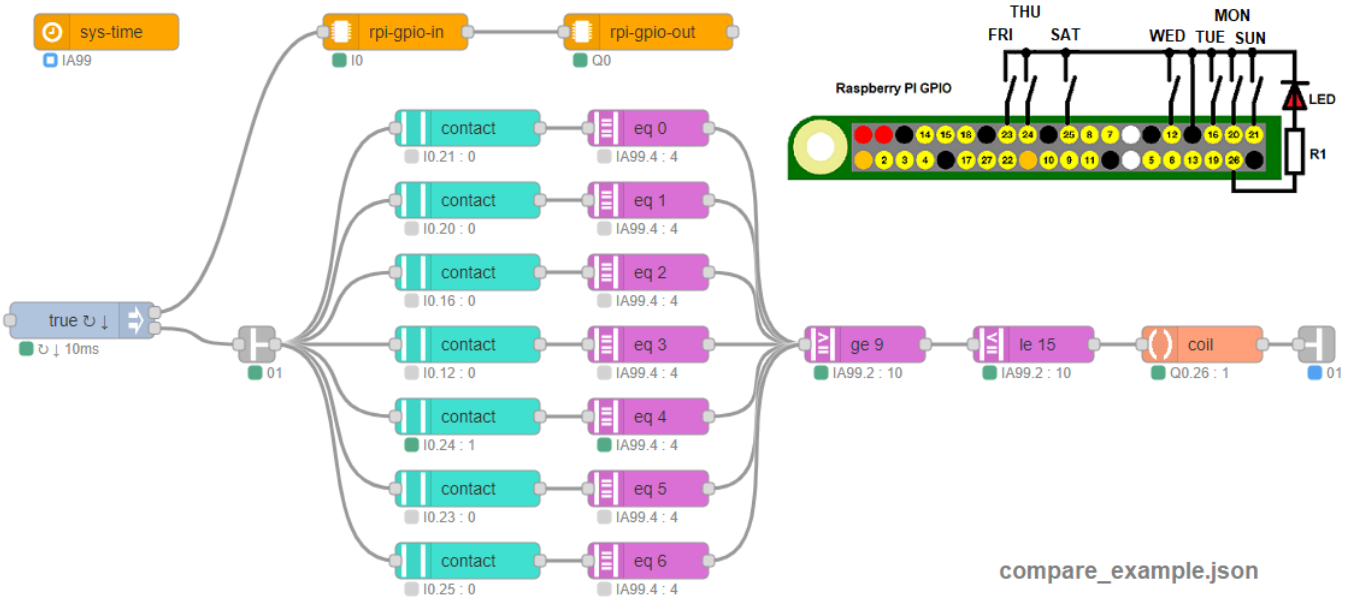
Compares analog variables and acts as a contact.

Variable **IA, QA, MA, T, C**,
can be compared with **IA, QA, MA, T, C, Number-Constant, Timer-Constant**.
Compare Operation:
== equal, **<>** not equal, **<** lower then, **>** greater then, **<=** lower-equal, **>=** greater-equal.
increase increase by, **decrease** decrease by.

Input is passed through to output if compare is *True*. Else *False* is send to output.

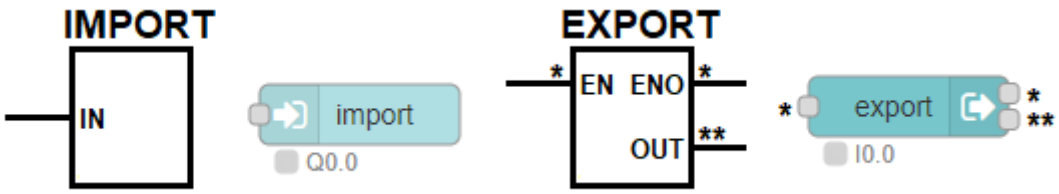
Example:

This example switches on a consumer (e.g. lamp) with switches on selectable days of the week
and between hour 9 and 15.
The redPlc module **sys-time** reads the system time and saves it in the variable IA99.
The day of the week is stored in index 4 (0 = sunday, 1 = monday ..) and the hours in index 2.
The days of the week are selected with the switches on the Raspberry Pi input
and the days of the week are queried with the compare node.
After that, the start and end hours are queried with the compare nodes in series.
If all queries are fulfilled, the Raspberry Pi output is switched on.
Select in module node **rpi-gio-in** GPIO12 .. GPIO25 as pullup inputs.
Select in module node **rpi-gio-out** GPIO26 as output (R1 depends on used LED, ~220 Ohms).



Import/Export Node

Imports or exports data from other Node-Red nodes.



External data can come from other nodes.
RedPlc variables can also be sent to other nodes.
These other nodes can be for example Dashboard, MQTT and Databases.
It is possible to include data from other nodes in the ladder logic process.
The import node stores external data asynchronously in redPlc variables.
The export node sends redPlc variables synchronously to external nodes.
The export is started with input message *True* at the EN input.
To connect export nodes in series, input EN is send to output ENO.

Example:

In this example voltage, current and power is imported via MQTT and saved in MA variable elements.
Rung 00 checks current if >= 2.1 and triggers timer T0. If limit more then 2 seconds exceeded, M0.0 is set.
Rung 01 checks voltage if >= 30.5 and triggers timer T1. If limit more then 3 seconds exceeded, M0.1 is set.
Rung 02 checks power if >= 15.5 and triggers timer T2. If limit more then 1 seconds exceeded, M0.2 is set.
The export nodes sends M0.0, M0.1, M0.2 to Dashboard slider for indicate alarms.