easyLadder

The PLC engine for your **Raspberry Pi**

User manual

Contents

1. Introduction

The Raspberry Pi is a powerful Linux computer that has gained wide acceptance among technicians and hobbyists, especially in automation projects.

In this field of automation, PLCs (Programmable Logic Controllers) are extensively used because program implementation is done quickly and easily. Ladder programming is perfectly suited to control automated processes. For more complex calculations, some PLC manufacturers incorporate script languages as a complement to the ladder engine.

easyLadder is a complete PLC engine for the Raspberry Pi. It was designed to work together with the RasPICER board to provide the necessary inputs and outputs. Nevertheless, the RasPICER board is not required (but highly recommended), since you can use the Raspberry GPIO ports and external I2C I/O expanders to supply inputs and outputs to the system. In addition, the system can be expanded with up to 32 additional Ethernet I/O modules.

The RasPICER board includes:

- 8 digital inputs (12-24 VDC)
- 4 relay outputs
- 4 NPN transistor outputs
- 2 analog inputs (0-20 mA)
- 2 analog outputs (0-20 mA)
- 1 RS485 serial port
- 1 RS232 serial port
- Power manager (12-24 VDC)
- Rechargeable backup battery (3000 mAh)
- Real time clock
- Watchdog functions

Using the RasPICER, the entire system is powered supplying a voltage from 12 VDC to 24 VDC to the terminals VDC and GND, or by means of the backup battery.

For more information about the RasPICER board, visit http://www.ferrariehijos.com/RasPICER.

easyLadder features include:

- Retentive devices
- Timers and retentive timers (100ms and 10ms)
- Math instructions (floating point support)
- MODBUS TCP/IP server and client functions.
- Advanced instructions, such as PID control
- Serial port support
- Online edition (program edition without stopping control)
- Free graphical ladder editor **easyLadder studio** for Microsoft Windows. This software allows programming and monitoring the PLC engine through the network connection.
- Software library for interfacing your custom program (C or Python) with the PLC engine

For support contact info@ferrariehijos.com or http://www.ferrariehijos.com/easyLadder.

2. REQUIREMENTS

easyLadder is compatible with the following Raspberry Pi versions:

- Raspberry Pi 1 A+
- Raspberry Pi 1 B+
- Raspberry Pi zero
- Raspberry Pi 2 B (recommended)
- Raspberry Pi 3 B (recommended)

A working Ethernet/Wifi connection is required for online monitoring and programming with the *easyLadder studio* for *Microsoft Windows*. You can also export your PLC program to files and transfer them to the Raspberry file system using your preferred method.

All software is compiled for the RASPBIAN JESSIE Linux distribution. Other distributions may be supported on request. Please contact us for other Linux kinds.

3. PLC ENGINE

3.1. Introduction to ladder programming

easyLadder is a simple but powerful software PLC engine. This guide explains some concepts of ladder programming, but it is not intended as a book for learning ladder. Some kind of ladder knowledge is highly recommended before reading.

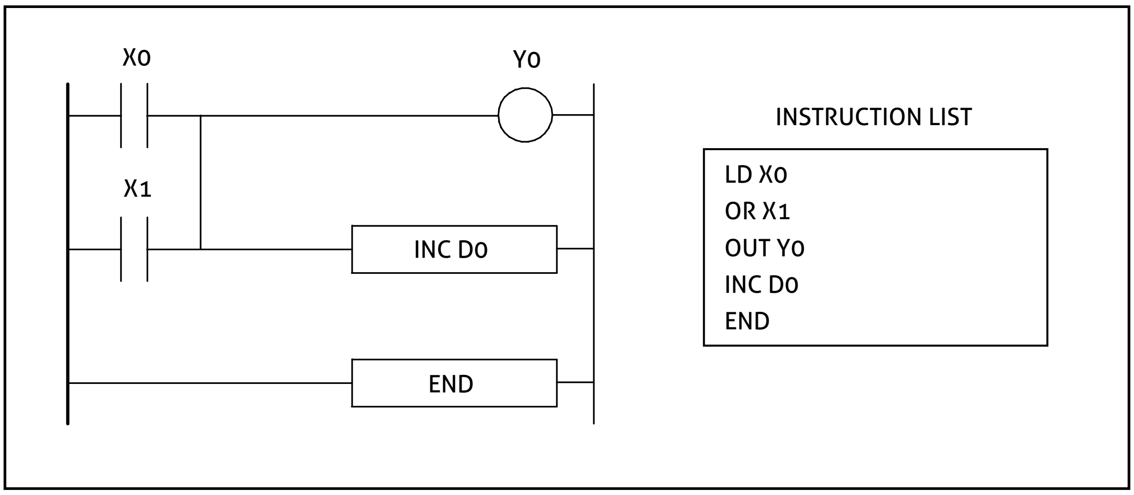
PLC machines are computers that continuously executes a ladder program. Each execution of the program is called **scan**. When the scan is complete the PLC will execute background actions like input/output refreshing and serial communication processing. When finished, the PLC will execute another program scan, repeating the cycle until the PLC is placed in STOP mode. Time elapsed between two scan cycles is called **scan time**.

The ladder program is a set of mnemonic instructions that can be easily translated to a graphic form (relay logic diagrams). These instructions are executed until a **FEND** or **END** instruction is found, meaning the end of the scan. After the **FEND** instruction you can place subroutines that can be called within the program. At the end of the code, an **END** instruction is required.

The PLC contains several memory allocations for storing data. These memory allocations are called **devices**. easyLadder includes two kind of devices according to the size of the contained value: Bit devices and Word devices. Bit devices can store boolean values (0-OFF or 1-ON). Word devices are 16 bit variables (signed or unsigned).

Instructions are generally divided into two groups: contact instructions and coil instructions. Coil instructions are executed using a condition context. This execution condition can be considered a boolean variable. Contact instructions are used to modify the condition variable for the incoming coil instructions, and are principally boolean operations (OR, AND, NOT...) against the previous condition. Coil instructions do not modify the condition, but executes according to the status of the current condition.

The following image represents a basic ladder program. At the left you can view the logic diagram, at the right the corresponding program using instruction list.



```
LD X0
OR X1
OUT Y0
INC D0
END
```

In this program, **LD X0** and **LD X1** are contact instructions and **OUT Y0**, **INC D0** and **END** are coil instructions. Coil instructions are always shown aligned to the right in a ladder diagram. X0, X1 and Y0 are bit devices, D0 is a word device. The lines can be seen as signal wires spreading the condition signal.

The line shown at the left of the ladder diagram is called **root line** or root condition, and corresponds to an always ON signal.

In order to clarify the meaning of the instructions, **LD X0** will load the value of the device X0 to the condition context, **OR X1** will execute a boolean OR between the condition context and the value X1. **OUT Y0** will transfer the value of the condition to device Y0. **INC D0** will increment the value of device D0 only when the condition value is ON. **END** represents the end of the program.

The above program will set Y0 to ON and will increment D0 when X0 OR X1 is ON. When X0 and X1 are OFF, Y0 will be reset to OFF and D0 will maintain the value.

As you can note, coil instruction behavior concerning the condition value varies between instructions.

For example, the **OUT Y0** will always refresh the value Y0. If the instruction condition is OFF, Y0 will be reset to OFF. In the same way, when the instruction condition is ON, Y0 will be set to ON. On the other hand, instruction **INC D0** will only increment the value of D0 if the instruction condition is ON. When OFF, the value of D0 **will be not changed**. The **END** instruction does not use any instruction condition.

Another very important point to note is regarding the INC D0 instruction. When X0 or X1 is TRUE, the value of D0 will increment in EVERY SCAN because, as previously noted, the PLC will execute the program cycle continuously while in RUN mode. For example, if the scan time is 1 ms, the value of D0 will increment every 1 ms when the execution condition is TRUE.

Although ladder diagrams look alike electrical schematics, you must consider that PLC programs are running step by step, in order, using instructions. For example, when setting a physical output value, you can reset and set the device value several times during a cycle scan, but this device value will be transferred to the physical output only at the end of the scan.

3.2. PLC devices

3.2.1. Introduction

PLC contains several memory allocations for storing data. These memory allocations are called **devices**.

easyLadder includes two kind of devices according to the size of the contained value: Bit devices and Word devices.

- **Bit devices** can store only boolean values (0 or 1).
- **Word devices** are 16 bit variables. These devices can store a number between -32,768 and 32,767 when signed, or between 0 and 65,535 when unsigned.

Several instructions can use two consecutive **Word devices** to form a bigger composite device:

- **Dword devices** are 32 bit variables. Can store a number between -2,147,483,648 and 2,147,483,647 when signed, or between 0 and 4,294,967,295 when unsigned.
- **Float devices** are 32 bit variables in floating point format, for storing real numbers.

All memory is organized in little-endian format, so when storing a Dword value 0x12345678 in device D0 (and D1), device D0 will store 0x5678 and device D1 will store 0x1234.

According to their volatility, easyLadder devices can be divided into:

- **Retentive devices**. These devices retain their value after a system shutdown or when PLC is placed in STOP mode.
- **Volatile devices.** Volatile devices are reset to 0 after a system shutdown or when PLC is switched to STOP mode.

3.2.2. Bit access to Word devices

easyLadder allows bit access to Word devices in every instruction. The reference is made adding a dot (.) after the Word device name and the bit number to reference (0-15).

For example **D1.2** references bit 2 of device D1, **D293.15** references bit 15 of device D293.

3.2.3. Word access to Bit devices

easyLadder allows word access to Bit devices in every instruction. The reference is made by specifying a prefix **Kn** followed by the Bit device name, where n is 1, 2, 4 or 8:

- **K1** prefix specifies the word device containing only bits 0 to 3 (nibble) from the Bit device referenced and consecutives.
- **K2** prefix specifies the word device containing only bits 0 to 7 (byte) from the Bit device referenced and consecutives.
- **K4** prefix specifies the word device containing bits 0 to 15 (word) from the Bit device referenced and consecutives.
- **K8** prefix specifies the dword device containing bits 0 to 31 (dword) from the Bit device referenced and consecutives.

For example, consider these device values: X0 (value 1), X1 (value 0), X2 (value 1), X3 (value 1), X4 (value 1). **K1X0** will contain value 0b1101 (13 decimal), **K1X1** will contain value 0b1110 (14 decimal).

3.2.4. Indexed access to devices

easyLadder allows indexed access to devices (Word devices or Bit devices) in every instruction. The reference is made by specifying the index between brackets [ ].

For example consider a value 10 in D0, **D100[D0]** will point to D110.

3.2.5. Specifying constants

When you need to use a constant value in any instruction, follow these tips:

- **Decimal constants** are specified with the **K** prefix. For example K200 references decimal 200 and K-1 references decimal -1.
- **Hexadecimal constants** are specified with the # prefix. For example #1A references hexadecimal 0x1A (decimal 26).
- **Floating constants** are specified without prefix. Use the dot (.) as decimal point.

3.2.6. easyLadder memory areas

easyLadder devices are divided in memory areas. For device designation, use the memory area name followed with the desired device number in decimal format (X10, D120...). The following table contains available memory areas accessible to the user:

Bit devices

Memory area	Type	Devices	Usage
X	-	512	Digital inputs
Y	Volatile	512	Digital outputs
W	Volatile	10000	Work bits
H	Retentive	10000	Retentive work bits
T (TC)	Volatile	256	Timer contact (100 ms precision)
TH (THC)	Volatile	256	Timer contact (10 ms precision)
C (CC)	Volatile	256	Counter contact
S	-	256	Special bit devices

Word devices

Memory area	Type	Devices	Usage
AX	-	256	Analog inputs
AY	Volatile	256	Analog outputs
D	Retentive	10000	User data devices
T (TD)	Retentive	256	Timer count value (100 ms precision)
TH (THD)	Retentive	256	Timer count value (10 ms precision)
C (CD)	Retentive	256	Counter value
SD	-	32	Special word devices

Pointer devices

Memory area	Type	Devices	Usage
P	-	256	Program pointers

3.2.7. Input / output device areas

These areas contains values for digital and analog inputs and outputs. Input areas are refreshed before any program scan. Output areas are transferred to physical output after the program scan.

X memory area (bit devices X0 – X511) contains status for the digital inputs. X0 to X7 are reserved for RasPICER digital inputs. Other inputs can be mapped to external Ethernet modules, Raspberry Pi GPIO ports or external I2C port expanders. Any unmapped input can be used as a work bit.

Y memory area (bit devices Y0 – Y511) contains status for the digital outputs. Y0 to Y7 are reserved for RasPICER digital outputs. Other outputs can be mapped to external Ethernet modules, Raspberry Pi GPIO ports or external I2C port expanders. Any unmapped output can be used as a work bit.

AX memory area (word devices AX0 – AX255) contains status for the analog inputs. AX0 to AX1 are reserved for RasPICER analog inputs. Other inputs can be mapped to external Ethernet modules. Any unmapped input can be used as a work word.

AY memory area (word devices Y0 – Y255) contains status for the analog outputs. AY0 to AY1 are reserved for RasPICER analog outputs. Other outputs can be mapped to external Ethernet modules. Any unmapped output can be used as a work word.

3.2.8. Work device areas

These areas are available to store user data and other work devices.

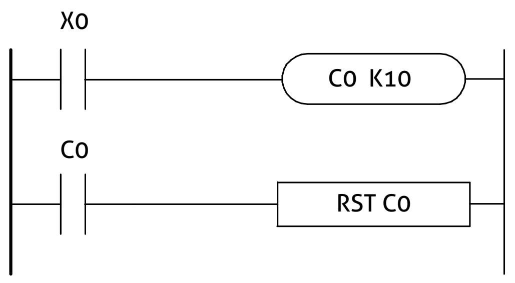
W memory area (bit devices W0 – W9999) is volatile. These devices will be reset to 0 when PLC switches to STOP mode or during engine startup.

H memory area (bit devices H0 – H9999) is retentive. Devices will retain value in when PLC switches to STOP mode or during engine startup.

D memory area (word devices D0 – D9999) is retentive. Devices will retain value in when PLC switches to STOP mode or during engine startup.

3.2.9. Counter devices

Counter management is done through **C** devices. Counters, as the name suggests, are used to count events. Counters are driven by the **OUT** instruction. When there is a rising edge in the instruction condition, the counter increments its value up to the setpoint specified in the **OUT** instruction (second parameter). Once the setpoint is reached, the correspondent counter contact turns ON. The counter value is retained on PLC STOP or shutdown.



INSTRUCTION LIST

```
LD X0
OUT C0 K10
LD C0
RST C0
```

The above example will count 10 pulses in X0 input. When the count is reached, the counter is reset to start again.

Each counter **Cn** has two associated devices **CCn** and **CDn**. **CCn** is a bit device that contains the contact state for the counter (count reached). **CDn** is a word device containing the current count value for the counter. **CDn** value starts counting from 0 to the counter setpoint. You do not need to access these **CCn** and **CDn** devices directly. You can use always the **Cn** device and the PLC will select the correct device based on the instruction executed (bit or word).

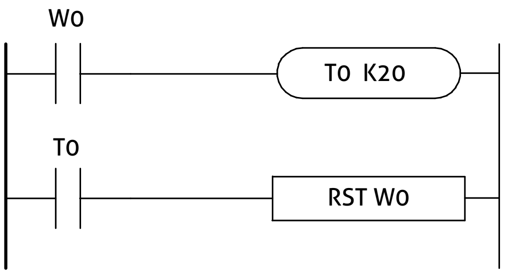
Counter status (counter contact) is refreshed during **OUT** instruction execution.

3.2.10. Timer devices

easyLadder contains several devices reserved for timer operation. There are available two timer precisions (100 ms and 10 ms). **T** devices (T0 – T255) use 100 ms unit for the timer, **TH** devices (TH0-TH255) use 10 ms unit. Both memory areas are independent, so it is possible to use T1 and TH1 simultaneously.

Normal (not retentive) timers are driven with the **OUT** instruction or **TIM** instruction. Both instructions are fully equivalent. These timers starts counting time when the instruction executes with an ON condition status, up to the setpoint specified in the instruction (second parameter). When the setpoint is reached, the corresponding timer contact turns ON.

When the instruction executes with an OFF condition, the timer is reset. You can also manually reset the timer by issuing a **RST Tn** instruction.



INSTRUCTION LIST

```
LD W0
OUT T0 K20
LD T0
RST W0
```

This example will maintain the W0 ON state for a time of 2 secs (20 * 100 ms). After this time, W0 will be set to OFF.

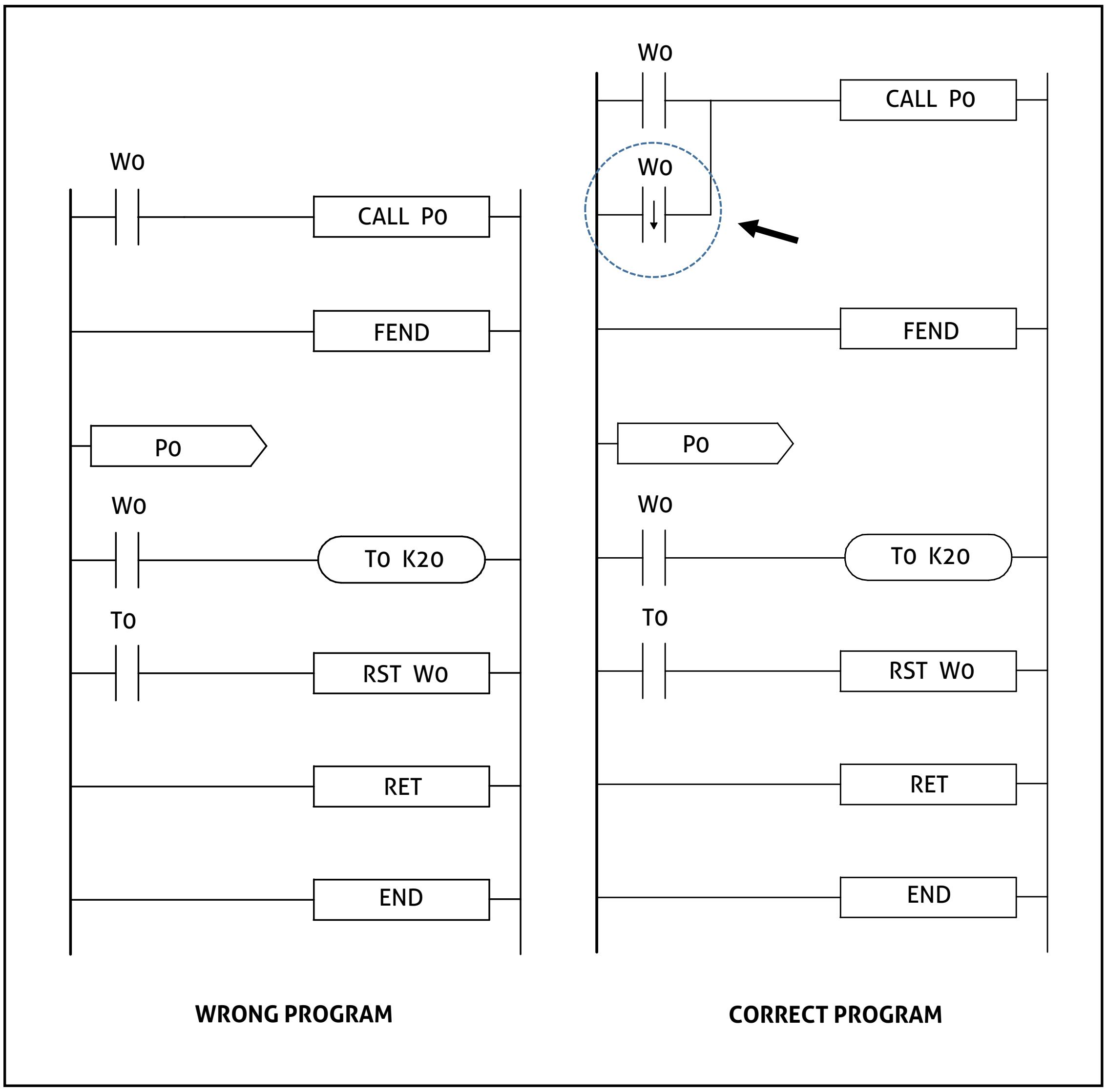
Retentive timers are driven with a **RTIM** instruction. Retentive timers count the time when the instruction executes with an ON condition status, up to the setpoint specified in the instruction. When the setpoint is reached, the corresponding timer contact turns ON.

When the instruction executes with an OFF condition, the timer is **not** reset, but only stops counting. You need to manually reset the timer by issuing a **RST Tn** instruction. The timer value is retained on PLC STOP or shutdown.

Each timer **Tn** (and **THn**) has two associated devices **TCn** and **TDn** (**THCn** and **THDn**). **TCn** is a bit device that contains the contact state for the timer (timer elapsed). **TDn** is a word device containing the current count value for the timer. **TDn** value starts counting from 0 to the timer setpoint, so this device contain the elapsed time (in timer units) from the start of the timed event. You do not need to access these **TCn** and **TDn** devices directly. You can use always the **Tn** device and the PLC will select the correct device based on the instruction executed (bit or word).

Timer status is refreshed during **OUT**, **TIM** or **RTIM** instruction execution. It is not required to execute the timer coil in every scan, but it must be executed at least every second when the timer is ON, in order to obtain correct timings. For this reason, care must be taken when using timers in subroutines or using **JMP** instructions.

Consider the example shown at the begin of this section. When trying to convert this program using subroutines we get the following two programs:



The program at the left initially seems to be correct. When W0 is ON, the P0 subroutine will be continuously called. This subroutine will count W0 ON time and, after 2 seconds, the value of W0 will be reset to OFF, and the subroutine will not be called anymore. What happens when we set W0 to ON again? The subroutine will be called again, but the timer **T0 coil was not executed in the OFF state, so the timer was NOT reset**. So, the W0 will be set to OFF immediately. The program shown at the right solves this running the subroutine again on W0 falling edge, so the timer will be reset when W0 switches from ON to OFF state. You can also correct the left program resetting the timer manually using RST T0 instruction when T0 contact sets ON.

Please consider this tip when using timers in subroutines.

For more information about timers, refer to the **TIM** and **RTIM** instructions.

3.2.11. Special devices

easyLadder provides two memory areas (**S** and **SD**) for storing status and system variables. **S** memory area contains special bit devices and **SD** contains special word devices.

S devices:

Device	Name	Description
S0	OFF	Always OFF bit
S1	ON	Always ON bit
S2	1st ON	Bit ON for the first scan cycle. Other cycles OFF.
S3	1st OFF	Bit OFF for the first scan cycle. Other cycles ON.
S4	10 ms	10 millisecond clock oscillator
S6	100 ms	100 millisecond clock oscillator
S5	1 s	1 second clock oscillator
S7	1 min	1 minute clock oscillator
S10	EQUAL	EQUAL flag for CMP instructions
S11	GREAT	GREAT flag for CMP instructions
S12	LESS	LESS flag for CMP instructions
S13	DIV ZERO	Division by zero in DIV instruction
S16	POWER	Power present in VDC terminals (RasPICER board only)
S17	EOC	End of battery charge (RasPICER board only)
S18	CHARGE	Battery charging (RasPICER board only)
S19	BTN PRESS	Power button pressed (RasPICER board only). Reset manually
S20	DBLBTN PRESS	Power button double pressed (RasPICER only). Reset manually
S24	16BIT MODE	16 bit mode for selected instructions. See **section 3.3.3.**
S25	RX1 READY	RS485 data ready for receive with RXD instruction (RasPICER only)
S26	RX2 READY	RS232 data ready for receive with RXD instruction (RasPICER only)
S27	RX3 READY	ttyAMA0 data ready for receive with RXD instruction
S28	TX1 READY	RS485 write buffer ready for TXD instruction (RasPICER only)
S29	TX2 READY	RS232 write buffer ready for TXD instruction (RasPICER only)
S30	TX3 READY	ttyAMA0 write buffer ready for TXD instruction

SD devices:

Device	Name	Description
SD0	SCAN TIME	Current scan time in 0,1 millisecond units
SD1	MIN SCAN TIME	Minimum scan time in 0,1 millisecond units
SD2	MAX SCAN TIME	Maximum scan time in 0,1 millisecond units
SD10	RTC YEAR	System clock current year (full year format)
SD11	RTC MONTH	System clock current month (1-12)
SD12	RTC DAY	System clock current day (1-31)
SD13	RTC HOUR	System clock current hour (0-23)
SD14	RTC MIN	System clock current minute (0-59)
SD15	RTC SEC	System clock current second (0-59)
SD16	PORT1 CFG	Configuration word for RS485 port (RasPICER board only)
SD17	PORT2 CFG	Configuration word for RS232 port (RasPICER board only)
SD18	PORT3 CFG	Configuration word for ttyAMA0 port
SD20	EXTERR0	Extension module error bitmask (modules 1-16)
SD21	EXTERR1	Extension module error bitmask (modules 17-32)
SD22	I2CERR0	I2C expander error bitmask (modules 1-16)
SD23	I2CERR1	I2C expander error bitmask (modules 17-32)
SD25	RX1 BYTES	RS485 num. bytes ready for RXD instruction (RasPICER board only)
SD26	RX2 BYTES	RS232 num. bytes ready for RXD instruction (RasPICER board only)
SD27	RX3 BYTES	ttyAMA0 num. bytes ready for RXD instruction
SD28	RX1 READ	RS485 num. bytes read in last RXD instruction (RasPICER only)
SD29	RX2 READ	RS232 num. bytes read in last RXD instruction (RasPICER only)
SD30	RX3 READ	ttyAMA0 num. bytes read in last RXD instruction

3.2.12. Pointer devices

easyLadder provides a special memory area (**P**) for storing up to 256 (**P0** to **P255)** pointers to the program memory. These devices are not accessible to the user.

A pointer is basically a label that designates a program location. When the PLC engine reaches a pointer during execution, it simply bypasses the pointer without taking any action. When this program location needs to be called, you can issue a **CALL Pn** or **CJ Pn** instruction, so the program flow will continue at the pointer position.

3.3. PLC instructions

3.3.1. Introduction

Instructions are divided mainly into two groups: *contact instructions* and *coil instructions*. Contact instructions are used to modify the condition context for coil instructions. Coil instructions do not modify the condition, but executes according to the status of the current condition.

It is generally not necessary to encode contact instructions directly, since **easyLadder studio** software will do it for you through the graphical contact representation.

Some instructions require operands. Operands are parameters passed to the instruction during execution. Operands are appended after the instruction name using a space as separator to form the complete instruction sentence.

INSTRUCTION *OP1 OP2 ...*

Example: ***ADD*** *D0 D1 D100*

Most instructions need an execution condition to work, that is, cannot connect directly to the ladder **root line**. If you need to run any instruction without condition, you can add the ALWAYS ON device (special device **S1**) as the condition for the instruction.

Other instructions does not require conditions (**END** for example) and must connect directly to the **root line**. These instructions are specified in the instruction reference.

3.3.2. Instruction modifiers

Some easyLadder instructions can be modified using prefixes and suffixes. You can find available modifiers for each command in the instruction reference section. Using modifiers you can easily select several variants for the same instruction.

Example: D**ADD**P ➔ D (Prefix) + **ADD** (Instruction) + P (Suffix)

Offered modifiers for coil instructions are:

Valid prefixes:

U This prefix indicates UNSIGNED WORD operation. The instruction function is modified to operate with UNSIGNED WORD data. Please note that most instructions can operate over signed or unsigned data without differences, so no U prefix will be available. Others, like comparison instructions, offer this modifier.

D This prefix indicates DOUBLE WORD operation. The instruction function is modified to operate with DOUBLE WORD data. When accessing devices, two consecutive devices will be used to form a 32 bit signed number.

UD This prefix indicates UNSIGNED DOUBLE WORD operation. The instruction function is modified to operate with UNSIGNED DOUBLE WORD data. When accessing devices, two consecutive devices will be used to form an unsigned 32 bit signed number.

F This prefix indicates FLOATING POINT operation. The instruction function is modified to operate with FLOATING POINT data. When accessing devices, two consecutive devices will be used to form a 32 bit floating point number.

Valid suffixes:

P This prefix indicates RISING EDGE operation. The instruction will be executed only on the rising edge of the operation condition, that is, when the execution condition changes from OFF to ON state. This modifier is available in almost all instructions.

F This prefix indicates FALLING EDGE operation. The instruction will be executed only on the falling edge of the operation condition, that is, when the execution condition changes from ON to OFF state. This modifier is available in almost all instructions.

For example, instruction D**ADD**P indicates ADD instruction with DOUBLE WORD data, and executed on rising edge.

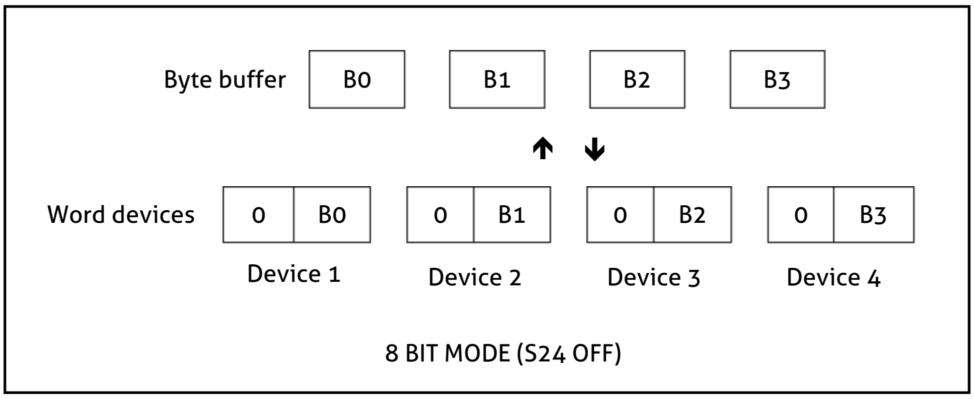
3.3.3. 16 bit mode

Some PLC instructions that operate on byte buffers operate differently regarding to the state of the special bit device **S24** (16BIT MODE). Serial instructions are examples of these instructions.

All word PLC devices are 16 bit values. When operating byte buffers easyLadder offers two options to manage word-to-byte and byte-to-word data movement:

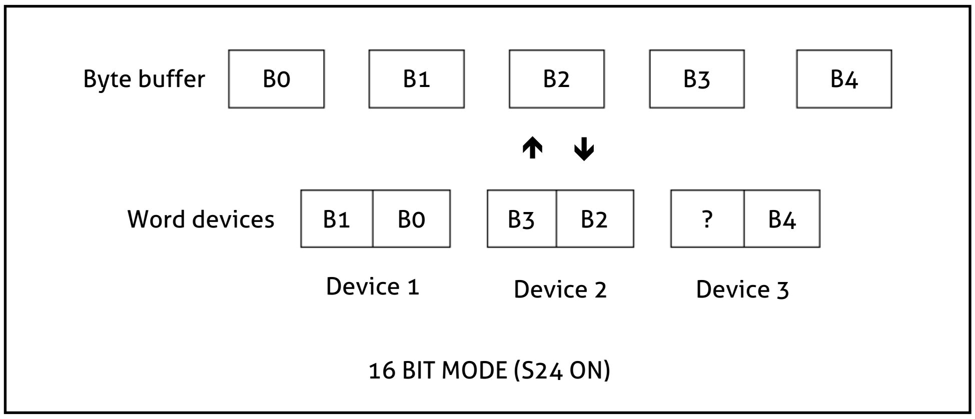
8BIT MODE Using this mode, each word device is considered a byte device. Byte-to-device and device-to-byte transfer is done using the lower byte of the device only. Higher byte of the device is ignored in word to byte operations, and is set to 0 in byte to word operations.

This method eases program coding but wastes memory space. You can set this mode by resetting to OFF **S24** device. This is the default mode when the PLC engine is switched to RUN mode.



16BIT MODE Using this mode word devices are fully used. Byte-to-device and device-to-byte transfer is done using the lower byte first and then the higher byte of the device.

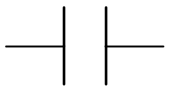
This method requires more effort during program coding but takes advantage of the full memory space. **BTOW** and **WTOB** instructions can help managing byte packet devices. You can set this mode by setting to ON **S24** device.

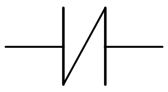


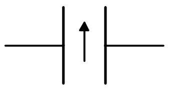
3.3.4. Instruction summary

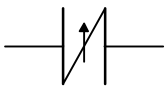
3.3.5. Contact instructions

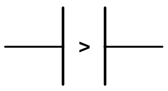
These instructions modify the execution condition. As you will note, every instruction is offered in three variants LD, OR and AND, sharing the same symbol. LD variant is used to start the logic operations and load the execution condition. OR and AND variants operates with this condition to obtain the desired ladder logic. When programming using the easyLadder studio software it is not required to encode AND or OR instructions since the software will compute these boolean operations to encode your graphic logic.

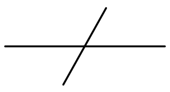
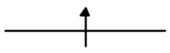
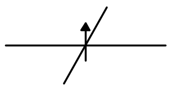
 **LD** **OR** **AND**	LD. Load condition, OR condition, AND condition *OP1*: Operand (Bit device)
	This instruction loads, ORs or ANDs the bit device *OP1* to the execution condition.

 **LDI** **ORI** **ANDI**	LDI. Load NOT condition, OR NOT condition, AND NOT condition *OP1*: Operand (Bit device)
	This instruction loads, ORs or ANDs the inverted value (NOT) of the bit device *OP1* to the execution condition.

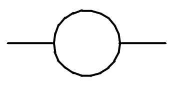
 **LDP** **ORP** **ANDP** **LDF** **ORF** **ANDF**	LDP. Load, OR, AND pulse condition *OP1*: Operand (Bit device)
	This instruction loads, ORs or ANDs the value of the rising edge (☐P instructions) or falling edge (☐F instructions) of the bit device *OP1* to the execution condition. NOTE: The image shows the RISING EDGE instruction symbol.

 **LDIP** **ORIP** **ANDIP** **LDIF** **ORIF** **ANDIF**	**LDIP. Load NOT, OR NOT, AND NOT pulse condition** *OP1*: Operand (Bit device) This instruction loads, ORs or ANDs the inverted value (NOT) of the rising edge (☐P instructions) or falling edge (☐F instructions) of the bit device *OP1* to the execution condition. NOTE: The image shows the RISING EDGE instruction symbol.

 **LD>** **OR>** **AND>** **LD<** **OR<** **AND<** **LD=** **OR=** **...**	**LD>. Load compare, OR compare, AND compare** *OP1*: Operand 1 (Word device / Constant) *OP2*: Operand 2 (Word device / Constant) Modifiers: UNSIGNED WORD: **U☐**; DOUBLE WORD: **D☐**; UNSIGNED DOUBLE: **UD☐**; FLOATING POINT: **F☐**; RISING EDGE: **☐P**; FALLING EDGE: **☐F** This instruction loads, ORs or ANDs the result of the comparison between *OP1* and *OP2* to the execution condition. *OP1* and *OP2* can be word values, unsigned word values, dword values, unsigned dword values o float values, depending on the prefix indicated. Valid comparisons are GREATER THAN (**>**), LESS THAN (**<**), EQUAL TO (**=**), GREATER THAN OR EQUAL TO (**>=**), LESS THAN OR EQUAL TO (**<=**), NOT EQUAL TO (**<>**). For example, for NOT EQUAL TO comparison instructions are LD<>, OR<>, AND<>.

 **CNDI**	## CNDI. NOT condition *No operands* This instruction inverts the execution condition.
 **CNDP** **CNDF**	## CNDP. Pulse condition *No operands* This instruction detects a RISING EDGE (☐P instruction) or FALLING EDGE (☐F instruction) in the input execution condition and puts the result to the output execution condition. NOTE: The image shows the RISING EDGE instruction symbol.
 **CNDIP** **CNDIF**	## CNDIP. NOT pulse condition *No operands* This instruction detects a RISING EDGE (☐P instruction) or FALLING EDGE (☐F instruction) in the input execution condition and puts the inverted (NOT) of the result to the output execution condition. NOTE: The image shows the RISING EDGE instruction symbol.

3.3.6. Bit management instructions



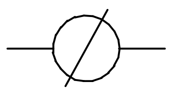
OUT

OUT. Output bit

OP1: Destination (Bit device / Timer device / Counter device)
OP2: Setpoint (Word device / Constant) (*when OP1 is a Timer or Counter device*)

When *OP1* references a counter or timer device, this instruction drives the counter or timer. The *OP2* parameter is required and designates the setpoint for the timer or counter. Refer to sections **3.2.8. Counter devices** and **3.2.9. Timer devices** for additional details.

For other bit devices in *OP1*, this instruction transfers the execution condition value to the *OP1* device. Parameter *OP2* is not required. The OUT instruction updates *OP1* value in every execution. When the execution condition is ON, *OP1* will be set to ON; when execution condition is OFF, *OP1* will be reset to OFF.



OUTI

OUTI. Output NOT bit

OP1: Destination (Bit device)

This instruction transfers the inverted (NOT) execution condition value to the *OP1* device. The OUTI instruction updates *OP1* value in every execution. When the execution condition is ON, *OP1* will be reset to OFF; when execution condition is OFF, *OP1* will be set to ON.

SET

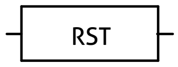
SET. Set bit or word device

OP1: Destination (Bit device / Word device)

Modifiers				
	DOUBLE WORD:	**D□**	RISING EDGE:	**□P**
			FALLING EDGE:	**□F**

When the execution condition is ON, this instruction sets to ON the bit device *OP1*. If *OP1* is a word or double word device, the value is set to 1.

When the execution condition is OFF, no action is performed.



RST

RST. Reset bit or word device

OP1: Destination (Bit device / Word device)

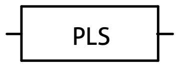
Modifiers

DOUBLE WORD:	**D**☐	RISING EDGE:	☐**P**
		FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction sets to OFF the bit device *OP1*. If *OP1* is a word or double word device, the value is set to 0.

When *OP1* is a timer or counter device, the corresponding timer or counter is manually reset.

When the execution condition is OFF, no action is performed.

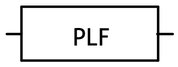


PLS

PLS. Pulse bit device (rising edge)

OP1: Destination (Bit device)

This instruction detects a RISING EDGE (OFF to ON transition) in the execution condition and outputs the result to the *OP1* bit device. When a RISING EDGE is detected *OP1* turns ON, in all other cases the *OP1* will be reset to OFF.



PLF

PLF. Pulse bit device (falling edge)

OP1: Destination (Bit device)

This instruction detects a FALLING EDGE (ON to OFF transition) in the execution condition and outputs the result to the *OP1* bit device. When a FALLING EDGE is detected *OP1* turns ON, in all other cases the *OP1* will be reset to OFF.

ALT

ALT. Invert bit device

OP1: Destination (Bit device)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction inverts (NOT) *OP1* value and stores the result back to *OP1*. If *OP1* is ON, device *OP1* will be reset to OFF; if *OP1* is OFF, device *OP1* will be set to ON.

When the execution condition is OFF, no action is performed.

3.3.7. Comparison instructions

CMP

CMP. Data comparison

OP1: Operand 1 (Word device / Constant)
OP2: Operand 2 (Word device / Constant)

Modifiers

UNSIGNED WORD:	**U**☐	RISING EDGE:	☐**P**
DOUBLE WORD:	**D**☐	FALLING EDGE:	☐**F**
UNSIGNED DOUBLE:	**UD**☐		
FLOATING POINT:	**F**☐		

When instruction condition is ON, this instruction compares *OP1* data against *OP2* data, according to the supplied modifiers, and signals the result of the comparison by setting or resetting the special devices EQUAL (**S10**), GREAT (**S11**) and LESS (**S12**).

For example, when issuing **CMP K10 K20**, EQUAL (**S10**) will be reset, GREAT (**S11**) will be reset and LESS (**S12**) will be set.

When the execution condition is OFF, no action is performed.

3.3.8. Data movement instructions

 MOV	MOV. Data move *OP1*: Source (Word device / Constant) *OP2*: Destination (Word device)			
	Modifiers	DOUBLE WORD: FLOATING POINT:	**D☐** **F☐**	RISING EDGE: **☐P** FALLING EDGE: **☐F**
	When the execution condition is ON, this instruction moves *OP1* data to *OP2* device, according to the supplied modifiers. When modifier **D☐** or **F☐** is used, two consecutive words will be used for getting and storing the value. When the execution condition is OFF, no action is performed.			

 BMOV	BMOV. Block move *OP1*: Source (Word device / Bit device) *OP2*: Destination (Word device / Bit device) *OP3*: Count (Word device / Constant)			
	Modifiers			RISING EDGE: **☐P** FALLING EDGE: **☐F**
	When the execution condition is ON, this instruction moves (copies) a block of *OP3* devices starting from *OP1* device to *OP2* and following devices. Both *OP1* and *OP2* must have the same type (Bit or Word). For example, **BMOV D0 D100 K3** will copy D0 to D100, D1 to D101 and D2 to D102. When the execution condition is OFF, no action is performed.			

BSET

BSET

BSET. Block fill

OP1: Source (Word device / Bit device / Constant)
OP2: Destination (Word device / Bit device)
OP3: Count (Word device / Constant)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction fills a block of *OP3* devices starting from *OP2* device with the value indicated in *OP1*. Both *OP1* and *OP2* must have the same type (Bit or Word), unless *OP2* is a bit device. In this case *OP1* can have any kind, so destination bits will be set if *OP1* is not 0.

For example, **BSET D0 D100 K3** will set D100, D101 and D102 with the contents of D0.

When the execution condition is OFF, no action is performed.

BTOW

BTOW

BTOW. Byte to word conversion

OP1: First source byte (Word device)
OP2: First destination word (Word device)
OP3: Byte count (Word device / Constant)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction packs a count of OP3 devices starting at *OP1* (containing byte values) to word devices starting at *OP2*. Lower byte of *OP1* is copied to the lower byte of *OP2*, and lower byte of *OP1+1* is copied to the higher byte of *OP2* and so on. Higher bytes of source devices are ignored. For example, when *OP3* is 3:

X B0	X B1	X B2	➔	B1 B0	? B2
OP1	*OP1+1*	*OP1+2*		*OP2*	*OP2+1*

When the execution condition is OFF, no action is performed.

This instruction is useful when working in 16BIT MODE (see **section 3.3.3.**)

WTOB

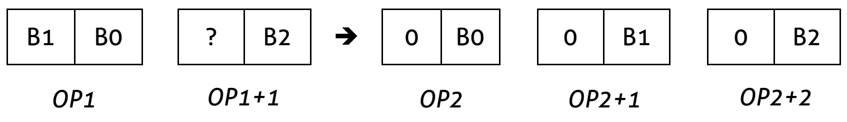
WTOB. Word to byte conversion

OP1: First source word (Word device)
OP2: First destination byte (Word device)
OP3: Byte count (Word device / Constant)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction extracts a total count of *OP3* bytes starting at *OP1* (containing packed byte values) to devices starting at *OP2*. Lower byte of *OP1* is copied to the lower byte of *OP2*, higher byte of *OP1* is copied to the lower byte of *OP2+1* and so on. Higher bytes of destination devices are set to 0. For example, when *OP3* is 3.



When the execution condition is OFF, no action is performed.

This instruction is useful when working in 16BIT MODE (see **section 3.3.3.**)

3.3.9. Arithmetic instructions

INC

INC. Increment data

OP1: Operand (Word device)

Modifiers

DOUBLE WORD:	**D**☐	RISING EDGE:	☐**P**
FLOATING POINT:	**F**☐	FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction increments (+1) *OP1* data value, according to the supplied modifiers, storing the result back to *OP1* device. When modifier **D**☐ or **F**☐ is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

DEC

DEC. Decrement data

OP1: Operand (Word device)

Modifiers

DOUBLE WORD:	**D☐**	RISING EDGE:	**☐P**
FLOATING POINT:	**F☐**	FALLING EDGE:	**☐F**

When the execution condition is ON, this instruction decrements (-1) *OP1* data value, according to the supplied modifiers, storing the result back to *OP1* device. When modifier **D☐** or **F☐** is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

ADD

ADD. Data addition

OP1: Operand 1 (Word device / Constant)
OP2: Operand 2 (Word device / Constant)
OP3: Result (Word device)

Modifiers

DOUBLE WORD:	**D☐**	RISING EDGE:	**☐P**
FLOATING POINT:	**F☐**	FALLING EDGE:	**☐F**

When the execution condition is ON, this instruction sums *OP1* data value to *OP2* data value, according to the supplied modifiers, storing the result to *OP3* device (*OP3 = OP1 + OP2*). When modifier **D☐** or **F☐** is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

SUB

SUB. Data subtraction

OP1: Minuend (Word device / Constant)
OP2: Subtrahend (Word device / Constant)
OP3: Result (Word device)

Modifiers

DOUBLE WORD:	**D☐**	RISING EDGE:	**☐P**
FLOATING POINT:	**F☐**	FALLING EDGE:	**☐F**

When the execution condition is ON, this instruction subtracts *OP1* data value minus *OP2* data value, according to the supplied modifiers, storing the result to *OP3* device (*OP3 = OP1 – OP2*). When modifier **D☐** or **F☐** is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

MUL

MUL. Data multiplication

OP1: Multiplicand (Word device / Constant)
OP2: Multiplier (Word device / Constant)
OP3: Result (Word device)

Modifiers

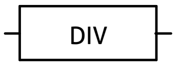
UNSIGNED WORD:	**U☐**	RISING EDGE:	**☐P**
DOUBLE WORD:	**D☐**	FALLING EDGE:	**☐F**
UNSIGNED DOUBLE:	**UD☐**		
FLOATING POINT:	**F☐**		

When the execution condition is ON, this instruction multiplies *OP1* data value by *OP2* data value, according to the supplied modifiers, storing the result to *OP3* device (*OP3 = OP1 * OP2*).

When using single words (no modifier or **U☐**) input devices will use one word, but result will use two words (**DOUBLE WORD result**).

When using double words or float modifiers (**U☐**, **UD☐** or **F☐** modifiers) two consecutive words will be used for getting and storing the result in the corresponding format.

When the execution condition is OFF, no action is performed.



DIV

DIV. Data division

OP1: Dividend (Word device / Constant)
OP2: Divisor (Word device / Constant)
OP3: Result (Word device)

Modifiers

UNSIGNED WORD:	**U☐**	RISING EDGE:	**☐P**
DOUBLE WORD:	**D☐**	FALLING EDGE:	**☐F**
UNSIGNED DOUBLE:	**UD☐**		
FLOATING POINT:	**F☐**		

When the execution condition is ON, this instruction divides *OP1* data value by *OP2* data value, according to the supplied modifiers, storing the result to *OP3* device (*OP3 = OP1 / OP2*).

When using single words (no modifier or **U☐**) input devices will use one word, but result will use two words. First word will contain division result (quotient), next word will contain the remainder of the operation.

When using double words (**U☐** or **UD☐** modifiers) two consecutive words will be used for getting parameters, and four consecutive words will be used for storing the result. First two result words will contain the quotient in double word format; next two words will contain the remainder in double word format.

When using floating point (**F☐** modifier) two consecutive words will be used for getting parameters and storing result in floating point format. No remainder is returned.

When the execution condition is OFF, no action is performed.

3.3.10. Data logic instructions

WAND

WAND. Word / double word logic AND

OP1: Operand 1 (Word device / Constant)
OP2: Operand 2 (Word device / Constant)
OP3: Result (Word device)

Modifiers				
	DOUBLE WORD:	**D☐**	RISING EDGE: FALLING EDGE:	☐**P** ☐**F**

When the execution condition is ON, this instruction computes the logical AND between *OP1* and *OP2* data values storing the result to *OP3* device (*OP3 = OP1 AND OP2*). When modifier **D☐** is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

WOR

WOR. Word / double word logic OR

OP1: Operand 1 (Word device / Constant)
OP2: Operand 2 (Word device / Constant)
OP3: Result (Word device)

Modifiers				
	DOUBLE WORD:	**D☐**	RISING EDGE: FALLING EDGE:	☐**P** ☐**F**

When the execution condition is ON, this instruction computes the logical OR between *OP1* and *OP2* data values storing the result to *OP3* device (*OP3 = OP1 OR OP2*). When modifier **D☐** is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

WXOR

WXOR. Word / double word logic OR

OP1: Operand 1 (Word device / Constant)
OP2: Operand 2 (Word device / Constant)
OP3: Result (Word device)

Modifiers				
	DOUBLE WORD:	**D**☐	RISING EDGE: FALLING EDGE:	☐**P** ☐**F**

When the execution condition is ON, this instruction computes the logical XOR (eXclusive OR) between *OP1* and *OP2* data values storing the result to *OP3* device (*OP3 = OP1 XOR OP2*). When modifier **D**☐ is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

WNOT

WNOT. Word / double word logic NOT

OP1: Source (Word device / Constant)
OP2: Result (Word device)

Modifiers				
	DOUBLE WORD:	**D**☐	RISING EDGE: FALLING EDGE:	☐**P** ☐**F**

When the execution condition is ON, this instruction computes the logical NOT of *OP1* data value storing the result into *OP2* device (*OP2 = NOT OP1*). When modifier **D**☐ is used, two consecutive words will be used for getting and storing the value.

When the execution condition is OFF, no action is performed.

ROR

ROR. Word / double word shift right

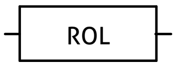
OP1: Operand (Word device)
OP2: Positions (Word device / Constant)

Modifiers				
	DOUBLE WORD:	**D**☐	RISING EDGE: FALLING EDGE:	☐**P** ☐**F**

When the execution condition is ON, this instruction shifts *OP1* data value to the right *OP2* bit positions, storing the result back to *OP1* device (*OP1 = OP1>>OP2*). No carry is used in this operation. When modifier **D**☐ is used, two consecutive words will be used for getting and storing the value.

For example, if D0 is binary 0b1010111110001001, when executing **ROR D0 K2**, device D0 will contain binary 0b0010101111100010.

When the execution condition is OFF, no action is performed.

 **ROL**	ROL. Word / double word shift left *OP1*: Operand (Word device) *OP2*: Positions (Word device / Constant)
	Modifiers — DOUBLE WORD: **D☐** — RISING EDGE: **☐P** — FALLING EDGE: **☐F**
	When the execution condition is ON, this instruction shifts *OP1* data value to the left *OP2* bit positions, storing the result back to *OP1* device (*OP1 = OP1<<OP2*). No carry is used in this operation. When modifier **D☐** is used, two consecutive words will be used for getting and storing the value. For example, if D0 is binary 0b1010111110001001, when executing **ROL D0 K2**, device D0 will contain binary 0b1011111000100100. When the execution condition is OFF, no action is performed.

3.3.11. Special timer instructions

 TIM	TIM. Timer control *OP1*: Timer (Timer device) *OP2*: Setpoint (Word device / Constant)
	This instruction drives the timer *OP1*. The *OP2* parameter designates the setpoint for the timer. Refer to section **3.2.9. Timer devices** for additional details. This instruction is fully equivalent to the **OUT Tn** instruction.

 RTIM	RTIM. Retentive timer control *OP1*: Timer (Timer device) *OP2*: Setpoint (Word device / Constant)
	This instruction drives the timer *OP1*. The *OP2* parameter designates the setpoint for the timer. Retentive timers do not reset when execution condition is OFF. You must reset manually these timers using **RST Tn** instruction. Refer to section **3.2.9. Timer devices** for additional details.

3.3.12. Data conversion instructions

DOUBLE

DOUBLE

DOUBLE. Single word to double word

OP1: Source (Word device / Constant)
OP2: Result (Word device)

Modifiers

UNSIGNED WORD:	**U□**	RISING EDGE:	**□P**
		FALLING EDGE:	**□F**

When the execution condition is ON, this instruction converts *OP1* data value, according to the supplied modifiers, to a double word value storing the result in *OP2* and next device.

When the execution condition is OFF, no action is performed.

FIX

FIX

FIX. Floating point to word / double word

OP1: Source (Word device / Constant - float value)
OP2: Result (Word device)

Modifiers

UNSIGNED WORD:	**U□**	RISING EDGE:	**□P**
DOUBLE WORD:	**D□**	FALLING EDGE:	**□F**
UNSIGNED DOUBLE:	**UD□**		

When the execution condition is ON, this instruction converts *OP1* floating point value to a word or double word value in *OP2*, signed or unsigned, according to the supplied modifiers.

When using double words (**U□** or **UD□** modifiers) two consecutive words will be used for storing the result.

When the execution condition is OFF, no action is performed.

FLT

FLT. Word / double word to floating point

OP1: Source (Word device / Constant)
OP2: Result (Word device - float value)

Modifiers

UNSIGNED WORD:	**U□**	RISING EDGE:	**□P**
DOUBLE WORD:	**D□**	FALLING EDGE:	**□F**
UNSIGNED DOUBLE:	**UD□**		

When the execution condition is ON, this instruction converts *OP1* value, according to the supplied modifiers, to a floating point value in *OP2*.

When using double words (**U□** or **UD□** modifiers) two consecutive words will be used for getting *OP1* parameter.

When the execution condition is OFF, no action is performed.

3.3.13. Floating-point math instructions

FSIN

FSIN. Sine calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE:	**□P**
FALLING EDGE:	**□F**

When the execution condition is ON, this instruction computes the sine of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP1* is expressed in radians.

When the execution condition is OFF, no action is performed.

FCOS

FCOS

FCOS. Cosine calculation

OP1: Word device / Constant
OP2: Word device

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction computes the cosine of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP1* is expressed in radians.

When the execution condition is OFF, no action is performed.

FTAN

FTAN

FTAN. Tangent calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction computes the tangent of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP1* is expressed in radians.

When the execution condition is OFF, no action is performed.

FASIN

FASIN

FASIN. Arc sine calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction computes the arc sine of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP2* is given in radians.

When the execution condition is OFF, no action is performed.

FACOS

FACOS

FACOS. Arc cosine calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction computes the arc cosine of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP2* is given in radians.

When the execution condition is OFF, no action is performed.

FATAN

FATAN

FATAN. Arc tangent calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction computes the arc tangent of *OP1* value and stores the result in *OP2*. Both parameters are floating-point data, requiring two consecutive word devices. *OP2* is given in radians.

When the execution condition is OFF, no action is performed.

FSQRT

FSQRT

FSQRT. Square root calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE: ☐**P**
FALLING EDGE: ☐**F**

When the execution condition is ON, this instruction calculates the square root of *OP1* value and stores the result in *OP2* (*OP2 = √OP1)*. Both parameters are floating-point data, requiring two consecutive word devices.

When the execution condition is OFF, no action is performed.

FEXP

FEXP

FEXP. Exponent (base e) calculation

OP1: Exponent (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction calculates the natural (base e) exponent of *OP1* value and stores the result in *OP2* (*$OP2 = e^{OP1}$*). Both parameters are floating-point data, requiring two consecutive word devices.

When the execution condition is OFF, no action is performed.

FLOG

FLOG

FLOG. Logarithm (base e) calculation

OP1: Operand (Word device / Constant)
OP2: Result (Word device)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction calculates the natural (base e) logarithm of *OP1* value and stores the result in *OP2* (*$OP2 = log_e OP1$*). Both parameters are floating-point data, requiring two consecutive word devices.

When the execution condition is OFF, no action is performed.

FPWR

FPWR

FEXP. Exponent (base e) calculation

OP1: Base (Word device / Constant)
OP2: Exponent (Word device / Constant)
OP3: Result (Word device)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction raises *OP1* value to the power of *OP2* value and stores the result in *OP3* (*$OP3 = OP1^{OP2}$*). All parameters are floating-point data, requiring two consecutive word devices.

When the execution condition is OFF, no action is performed.

3.3.14. Program flow instructions

CJ

CJ

CJ. Conditional jump

OP1: Jump destination (Pointer)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction makes the program jump to the pointer specified in *OP1*. The program will continue running instructions at this pointer position. Any instruction between **CJ** instruction location and the jump destination pointer will not be executed.

When the execution condition is OFF, no action is performed.

CALL

CALL

CALL. Subroutine call

OP1: Call destination (Pointer)

Modifiers

RISING EDGE:	☐**P**
FALLING EDGE:	☐**F**

When the execution condition is ON, this instruction makes the program jump to the pointer specified in *OP1*. The program will continue running instructions at the pointer position, until a **RET** instruction is found. When **RET** is found, the program will return to the next instruction just after **CALL**.

Any instruction between the **CALL** instruction and the jump destination, and between the **RET** position and the **CALL** location will not be executed.

NOTE: when doing the **CALL,** current condition stack is saved into memory and restored after **RET**, so program status after return remains unchanged. Due to memory restrictions, you can nest up to 100 calls. Take this into account when doing recursive calls.

When the execution condition is OFF, no action is performed.

RET

RET

RET. Return from subroutine

No operands

This instruction makes the program return to the next instruction after **CALL**. For more information see **CALL** instruction.

RET instruction runs without execution condition. Must be connected directly to the **root line**.

FOR

FOR. Loop execution

OP1: Repetitions (Word device / Constant)

This instruction is used together with the **NEXT** instruction to create program loops, and references the start of the loop.

FOR makes the program to execute the code between **FOR** and the correspondent **NEXT** instruction *OP1* times. When the **FOR** instruction is found, the PLC engine will store the value of *OP1* in an internal (loop count) variable continuing to execute instructions until **NEXT** is found. When found, loop count variable will be decremented. If not zero, the program will jump again to the next instruction after **FOR**, repeating the loop. When zero, the program will continue after **NEXT**.

You can nest up to 100 FOR-NEXT loops.

NOTE: Even if the **FOR** instruction is executed with an *OP1* value of 0, the loop will be executed 1 time, so an *OP1* value of 0 is equivalent to an *OP1* value of 1.

FOR instruction runs without execution condition. It must be connected directly to the **root line**.

NEXT

NEXT. End of loop execution

No operands

This instruction is used together with the **FOR** instruction to create program loops, and references the end of the loop. Refer to the **FOR** instruction for details.

NEXT instruction runs without execution condition. It must be connected directly to the **root line**.

END

END. End of program

No operands

There are two instructions related to the end of the program: **END** and **FEND**. The **END** instruction indicates the end of the program code. Like the **FEND** instruction, when the PLC engine reaches this instruction, the scan cycle ends.

If your program contains subroutines out of the main program cycle, these subroutines must be placed after the **FEND** instruction. So the **FEND** instruction indicates the end of the main scan cycle while the **END** instruction indicates the end of the program code.

The **END** instruction is required and must be the LAST instruction at the end of the entire code. The **FEND** instruction is only required when there is code after the main program cycle.

END instruction runs without execution condition. It must be connected directly to the **root line**.

FEND

FEND. End of scan cycle

No operands

There are two instructions related to the end of the program: **END** and **FEND**. The **FEND** instruction indicates the end of the scan cycle. Like the **END** instruction, when the PLC engine reaches this instruction, the scan cycle ends.

If your program contains subroutines out of the main program cycle, these subroutines must be placed after the **FEND** instruction. So the **FEND** instruction indicates the end of the main scan cycle while the **END** instruction indicates the end of the program code.

The **END** instruction is required and must be the LAST instruction at the end of the entire code. The **FEND** instruction is only required when there is code after the main program cycle.

FEND instruction runs without execution condition. It must be connected directly to the **root line**.

SHUTDOWN

SHUTDOWN. System shutdown

No operands

When the execution condition is ON, this instruction shutdowns the system by issuing a **poweroff** system command.

This command is especially useful when controlling the RasPICER power. When configuring the PLC to not power down automatically when external power supply is lost, you can do your required tasks before shutting down the system with this command. When power is restored, the RasPICER board will power the PLC again to recover functions.

When the execution condition is OFF, no action is performed.

3.3.15. Serial port instructions

RXD

RXD. Serial port receive

OP1: Port number 1,2 or 3 (Word device / Constant)
OP2: First buffer device (Word device)
OP3: Byte count (Word device / Constant)

This instruction is used to receive bytes from any serial port. *OP1* references the port number. Use a value of 1 for the RasPICER RS485 port, a value of 2 for the RasPICER RS232 port, and a value of 3 for the Raspberry ttyAMA0 TTL port.

Port must be open prior to use this instruction by writing port configuration to special device **SD16**, **SD17** or **SD18**, depending on the port used.

When **RXD** is executed with an ON condition, available received port data up to *OP3* bytes will be copied to *OP2* user buffer. After that, **SD28**, **SD29** or **SD30** will contain the number of bytes copied to the user buffer, and the entire receive buffer will be discarded. For example, when there are 12 bytes available for read and you execute **RXD K1 D0 K5**, a total of 5 bytes will be copied to **D0** device, **SD28** will be set to 5 and all 12 bytes will be removed for the serial buffer.

The byte count parameter *OP3* must be less or equal than 1024, because this is the size of the internal receive buffer.

Data is copied to the user buffer according to the status of the 16BIT MODE special device **S24**. When **S24** is OFF (default) bytes will be copied to independent word devices (in the above example, first byte to D0, second byte to D1…). When **S24** is ON bytes will be copied using the entire word device (first byte to the lower byte of D0, second byte to the higher byte of D0, third byte to the lower byte of D1…). More information on section **3.3.2. 16 bit instruction mode**.

The number of available bytes on the serial buffer can be monitored through special word devices **SD25**, **SD26** and **SD27** (number of available bytes), or bit devices **S25**, **S26** and **S27** (any byte available)**,** depending on the port used. You can call the **RXD** instruction without data available, but is a good practice to check the status of **S25**, **S26** and **S27** before execution.

For more information refer to section **3.4 Serial port programming**.

When the execution condition is OFF, no action is performed.

TXD

TXD. Serial port transmit

OP1: Port number 1,2 or 3 (Word device / Constant)
OP2: First buffer device (Word device)
OP3: Byte count (Word device / Constant)

This instruction is used to transmit bytes to any serial port. *OP1* references the port number. Use a value of 1 for the RasPICER RS485 port, a value of 2 for the RasPICER RS232 port, and a value of 3 for the Raspberry ttyAMA0 TTL port.

Port must be open prior to use this instruction by writing port configuration to special device **SD16**, **SD17** or **SD18**, depending on the port used.

When **TXD** is executed with an ON condition, *OP3* bytes will be copied from *OP2* user buffer to the internal transmit buffer of the port. The system will transfer these buffer to the physical serial port as a background operation. When **TXD** is executed, special bit devices **S28**, **S29** or **S30** (depending on the port used) will be reset to OFF to indicate the internal transmit buffer is busy (there is some data in the buffer).

The byte count parameter *OP3* must be less than 1024, because this is the size of the internal transmit buffer. Care must be taken when using the **TXD** instruction to not overflow the internal transmit buffer. You can avoid this problem by checking that special bit device **S28**, **S29** or **S30** is ON before issuing the **TXD** instruction. These especial bits indicate that the internal transmit buffer is empty and ready to contain up to 1024 bytes.

Data is copied from the user buffer according to the status of the 16BIT MODE special device **S24**. Consider the instruction **TXD K1 D0 K10**. When **S24** is OFF (default) bytes will be copied from independent word devices (first byte from D0, second byte from D1…). When **S24** is ON bytes will be copied using the entire word device (first byte from the lower byte of D0, second byte from the higher byte of D0, third byte from the lower byte of D1…). More information on section **3.3.2. 16 bit instruction mode**.

For more information refer to section **3.4 Serial port programming**.

When the execution condition is OFF, no action is performed.

3.3.16. MODBUS TCP/IP instructions (from v1.4)

MODBUS

MODBUS

MODBUS. MODBUS TCP/IP connection setup

OP1: First configuration device (Word device)

This instruction is used to configure a MODBUS TCP/IP client connection to a MODBUS TCP/IP device.

OP1 designates the first device of a buffer used to configure the MODBUS connection. A total of **8 consecutive word** devices will be used in this area. You need to set these parameters before instruction execution.

OP1	**Status**	MODBUS connection status
OP1+1	**Error**	MODBUS connection error
OP1+2	**IP1**	First byte of peer IP address
OP1+3	**IP2**	Second byte of peer IP address
OP1+4	**IP3**	Third byte of peer IP address
OP1+5	**IP4**	Forth byte of peer IP address
OP1+6	**TCP port**	TCP port (usually 502)
OP1+7	**Unit**	MODBUS destination unit

Word device *OP1* contains the **Status** of the MODBUS connection. This word is a mask of several bits according to the following table:

Bit 0 [OP1].0	**Ready**	MODBUS connection ready
Bit 1 [OP1].1	**Complete**	MODBUS transfer complete
Bit 2 [OP1].2	**Error**	MODBUS error detected
	...	
Bit 6 [OP1].6	**Enabled**	MODBUS connection enabled
Bit 7 [OP1].7	**Connected**	Connected to the MODBUS device

You can use up to 64 **MODBUS** instructions to connect to up to 64 MODBUS devices. These instructions cannot share the same *OP1* device buffer, just because this *OP1* device is used to reference the MODBUS connection in other MODBUS related instructions.

When the MODBUS instruction is executed (with an ON or OFF condition), value of Status (*OP1*) and Error (*OP1+1*) words are refreshed according to the status of the MODBUS connection. The **Ready** bit indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction.

Continues

$\boxed{\text{MODBUS}}$

MODBUS

Continued

MODBUS read and write operations are performed as a background process. When any MODBUS read or write operation is complete and the **MODBUS** instruction is executed, the **Complete** bit or **Error** bit will be set according to the success of the operation. On MODBUS read operations, destination devices will be refreshed only when the **MODBUS** instruction is executed. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When any error is detected during connection, read or write operation, the **Error** bit (*OP1.2*) will be set and the **Error** word (*OP1+1*) will contain information about this error. The following table contains possible error values:

0	No error
1	MODBUS exception 1 (illegal function)
2	MODBUS exception 2 (illegal data address)
3	MODBUS exception 3 (illegal data value)
4	MODBUS exception 4 (slave device failure)
5	MODBUS exception 5 (acknowledge)
6	MODBUS exception 6 (slave device busy)
8	MODBUS exception 8 (memory parity error)
10	MODBUS exception 10 (gateway path unavailable)
11	MODBUS exception 11 (gateway target failed to respond)
20	Timeout contacting MODBUS device
21	Frame error from MODBUS device
22	Unable to connect to MODBUS device
23	No free connections (more than 64 MODBUS instructions)
24	R/W operation without matching MODBUS instruction
25	Exceeded max R/W devices in MODBUS Read/Write

When **MODBUS** instruction is executed with an ON condition, easyLadder will try to connect to the device using the IP / port specified in the *OP1* area. When connection is lost or unsuccessful, easyLadder will retry the connection indefinitely. During instruction execution, IP address and port can be modified, but new values will be used only when an OFF to ON condition is detected in the **MODBUS** instruction.

When **MODBUS** instruction is executed with an OFF condition, easyLadder will close the connection to the MODBUS device.

For more information refer to section **3.5 MODBUS functions**.

MODBUSRDI

MODBUSRDI

MODBUSRDI. MODBUS read discrete inputs

OP1: First configuration device (Word device)
OP2: Starting MODBUS address (Word device / Constant)
OP3: Number of inputs (Word device / Constant)
OP4: First destination device (Bit device)

This instruction is used to read discrete inputs from remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSRDI** is executed with an ON condition, a MODBUS read discrete inputs operation (function code 2) is started as a background operation. A number of *OP3* inputs (up to 2000), starting at MODBUS address *OP2* will be transferred to easyLadder bit devices starting at *OP4.*

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word, error word and destination devices are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSRDI** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

MODBUSRC

MODBUSRC. MODBUS read coils

OP1: First configuration device (Word device)
OP2: Starting MODBUS address (Word device / Constant)
OP3: Number of coils (Word device / Constant)
OP4: First destination device (Bit device)

This instruction is used to read coils from remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSRC** is executed with an ON condition, a MODBUS read coils operation (function code 1) is started as a background operation. A number of *OP3* coils (up to 2000), starting at MODBUS address *OP2* will be transferred to easyLadder bit devices starting at *OP4.*

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word, error word and destination devices are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSRC** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

MODBUSRIR

MODBUSRIR

MODBUSRIR. MODBUS read input registers

OP1: First configuration device (Word device)
OP2: Starting MODBUS address (Word device / Constant)
OP3: Number of registers (Word device / Constant)
OP4: First destination device (Word device)

This instruction is used to read input registers from remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSRIR** is executed with an ON condition, a MODBUS read input registers operation (function code 4) is started as a background operation. A number of *OP3* registers (up to 125), starting at MODBUS address *OP2* will be transferred to easyLadder word devices starting at *OP4*.

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word, error word and destination devices are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSRIR** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

MODBUSRHR

MODBUSRHR

MODBUSRHR. MODBUS read holding registers

OP1: First configuration device (Word device)
OP2: Starting MODBUS address (Word device / Constant)
OP3: Number of registers (Word device / Constant)
OP4: First destination device (Word device)

This instruction is used to read holding registers from remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSRHR** is executed with an ON condition, a MODBUS read holding registers operation (function code 3) is started as a background operation. A number of *OP3* registers (up to 125), starting at MODBUS address *OP2* will be transferred to easyLadder word devices starting at *OP4*.

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word, error word and destination devices are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSRHR** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

MODBUSWC

MODBUSWC

MODBUSWC. MODBUS write coils

OP1: First configuration device (Word device)
OP2: First source device (Bit device)
OP3: Number of registers (Word device / Constant)
OP4: First destination MODBUS address (Word device / Constant)

This instruction is used to write coils to remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSWC** is executed with an ON condition, a MODBUS write multiple coils operation (function code 15) is started as a background operation. A number of *OP3* devices (up to 1968), starting at source device *OP2* will be transferred to MODBUS address starting at *OP4.*

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word and error word are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSWC** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

$\boxed{\text{MODBUSWR}}$

MODBUSWR

MODBUSWR. MODBUS write registers

OP1: First configuration device (Word device)
OP2: First source device (Word device)
OP3: Number of registers (Word device / Constant)
OP4: First destination MODBUS address (Word device / Constant)

This instruction is used to write registers to remote MODBUS devices. A valid call to **MODBUS** instruction with the same *OP1* parameter must be performed prior to use any MODBUS related instruction.

As a tip, the **MODBUS** instruction must be executed in every scan, prior to any read or write instruction. Read and write instructions can be used in subroutines.

Refer to **MODBUS** instruction for information about *OP1* parameter.

When **MODBUSWR** is executed with an ON condition, a MODBUS write multiple registers operation (function code 16) is started as a background operation. A number of *OP3* devices (up to 123), starting at source device *OP2* will be transferred to MODBUS address starting at *OP4*.

The **Ready** bit in *OP1* indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

Updates to status word and error word are performed when the corresponding **MODBUS** operation is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When **MODBUSWR** is executed with an OFF condition no operation is performed.

For more information refer to section **3.5 MODBUS functions**.

3.3.17. Other instructions

 CRC16	**CRC16. CRC16 calculation** *OP1*: Source (Word device) *OP2*: Byte count (Word device / Constant) *OP3*: Result (Word device)	
	Modifiers	RISING EDGE: ☐**P** FALLING EDGE: ☐**F**
	When the execution condition is ON, this instruction computes the 16 bit Cyclic Redundancy Check (CRC16) for the buffer starting at *OP1* device. A number of *OP2* bytes will be used for calculation. The result word is stored in *OP3* device. This instruction uses the CRC-16-IBM polynomial ($x^{16} + x^{15} + x^2 + 1$) used for MODBUS CRC calculation. Data in the source buffer is used according to the status of the 16BIT MODE special device **S24**. When **S24** is OFF (default) bytes are taken from independent word devices (first byte from first device, second byte from second device). When **S24** is ON bytes will be taken using the entire word device (first byte from the lower byte of first device, second byte from the higher byte of first device, third byte from the lower byte of second device). More information on section **3.3.2. 16 bit instruction mode**. When the execution condition is OFF, no action is performed.	

SCL

SCL. Linear scaling

OP1: Source (Word device / Constant)
OP2: Control data (Word device)
OP3: Result (Word device)

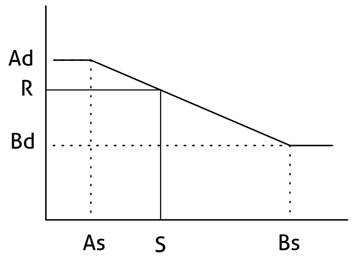
Modifiers

UNSIGNED WORD:	**U☐**	RISING EDGE:	**☐P**
		FALLING EDGE:	**☐F**

When the execution condition is ON, this instruction converts *OP1* source value into *OP3* value according to the specified linear function. **SCL** is useful for converting sensor output analog values to a useful technical scale.

The linear function is defined through two points A and B, using *OP2* control data devices:

OP2	**As (source for A)**
OP2 + 1	**Ad (result for A)**
OP2 + 2	**Bs (source for B)**
OP2 + 3	**Bd (result for B)**



Converting value As will result in value Ad, and converting Bs will result Bb. When the source value is out of the limits As to Bs, the output will be limited to Ad or Bd.

When the execution condition is OFF, no action is performed.

AVG

AVG. Moving average

OP1: Value (Word device / Constant)
OP2: Result buffer (Word device)
OP3: Input control word (Word device / Constant)

Modifiers

UNSIGNED WORD: **U□**

This instruction calculates the moving average for a series of **N** sample values (word values). The moving average is a special kind of average that computes continuously the mean value of the last added **N** numbers. When the sample buffer is full and another value is added to the series, the oldest value will be discarded to accommodate the new value.

OP1 references the word value to add to the series.

OP3 is the input control word for the average. The lower byte of this word specifies the number of sample values (**N**) in the series, so the maximum number of averaged values is 255. The bit **8** of this control word is used to inhibit new value addition. Whenever this bit is ON, the value *OP1* will not be added to the series.

OP2 references a word device buffer. This buffer is used for storing the average result (*OP2*), an output control word (*OP2*+1) and the last **N** devices added to the series (*OP2*+2 to *OP2*+2+**N**-1). A total consecutive 2 + **N** devices will be used in this result buffer. Do not modify these devices during instruction execution.

OP2	**Result**
OP2+1	**Output control word**
OP2+2	**Sample 1**
OP2+...	**. . .**
OP2+2+**N**-1	**Sample N**

When **AVG** is executed with an ON condition, the *OP1* value will be added to the series if bit **8** of *OP3* is OFF. The average result will be stored in *OP2* device. When the number of sample values reaches **N**, bit 8 of output control word (*OP2*+1) will be set to ON. When the number of sample values is less than **N**, the instruction calculates the average based on available sample values. You can use bit 8 of *OP2*+1 to know when the average is calculated with the complete series (**N** values).

When **AVG** is executed with an OFF condition, the average will be reset discarding current sample values.

For example, instruction **AVG D0 D100 K5** will put in D100 the average of 5 D0 values adding D0 to the series each execution cycle. Instruction **AVG D0 D100 D50** with a value 5 in D50 will be equivalent to the previous instruction, but by controlling bit **D50.8** (with a timer, for example) you can control when to add D0 to the average. Executing the **AVG** instruction with an OFF condition the average will be reset.

$-\boxed{\text{PID}}-$

PID

PID. PID control

OP1: Process variable (Word device)
OP2: First buffer word (Word device)
OP3: Manipulated variable (Word device)

This instruction executes a PID control loop. A Proportional Integral Differential controller (PID controller) is a control loop feedback mechanism commonly used in industrial control systems. This controller attempts to guide the value of the process variable (PV) near to a setpoint value (SP), modifying continuously the Manipulated variable (MV).

The **PID** instruction uses equations commonly used in discrete controllers. You can select between Type **B** and Type **C** general equations through bit Tp of the control word. The main difference between equations is that **C** type removes setpoint from proportional (P) term of the PID sentence. Both equations behave similarly during normal operation, showing differences only during setpoint changes. Type **C** results generally in less overshoot on response due to a large setpoint step, while type **B** acts faster on setpoint changes, and so will tend to overshoot slightly the output.

When driving the **PID** instruction with an ON condition, the control calculation happens according to the cadence t_s specified in the control area parameter. For example, consider a t_s parameter of 3 ms and a scan time of 2 ms. When the instruction is first executed, manipulated variable will be calculated. On the second execution, no calculation will be done since a total time of 2 ms has elapsed. On the third execution, when elapsed time is 4 ms, another calculation will be made, and the remaining 1 ms will be added to the count for next time calculations.

When driving the **PID** instruction with an OFF condition, no calculation is made and the **PID** calculation data will be reset.

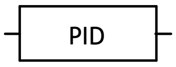
OP1 parameter designates the process variable (PV), or the variable you need to adjust. For example, if you are trying to regulate an oven temperature with a proportional valve, the process variable (PV) is the acquired temperature. *OP3* parameter designates the manipulated variable (MV), or the calculated action for the control loop, the proportional valve position. Both variables are treated as **signed words** (range -32,768 to 32,767).

You can modify the calculated MV variable at any time, before or during **PID** operation, so it is possible to limit MV value or change rate. Next **PID** operation will be based on new MV value.

OP2 designates the first device of a buffer used to accommodate control words and other PID calculation data. A total of **32 consecutive word** devices will be used in this area. First part of this buffer contain control constants to configure the **PID** operation. You need to set these constants in order to tune your control loop. Remaining devices are for internal use only. Do not modify these devices or incorrect **PID** operation will occur.

You can use any number of **PID** instructions in your program, but these instructions cannot share the same *OP2* device buffer, unless instructions are not executed simultaneously.

Continues



PID

Continued

This is the list of parameters found in the *OP2* control word area:

OP2	**SP**	Process setpoint
OP2+1	**t_s**	Sample time
OP2+2	**K_p**	Proportional constant
OP2+3	**T_i**	Integral constant
OP2+4	**T_d**	Derivative constant
OP2+5	**K_{lpf}**	Low pass filter constant
OP2+6	**Control**	Control word (see below)
OP2+7	**MV_{max}**	Max MV value
OP2+8	**MV_{min}**	Min MV value
OP2+9	**Reserved**	Do not use
OP2+...	**...**	
OP2+31	**Reserved**	Do not use

The process setpoint (SP) is the desired value for the process variable (PV). The control loop will modify the manipulated variable (MV) to guide the PV close to the SP. You can modify this value at any time.

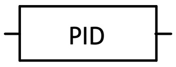
The sample time (t_s) is the time desired between **PID** calculations, in millisecond units. When the instruction is driven ON, this time is required to elapse between MV calculations. This value must be greater than 0. You can change this value at any time, but new value will be used when a rising edge on the **PID** instruction condition is detected, unless you set to ON Rc bit in the control word.

The proportional constant (K_p) defines the strength of the proportional action for the loop. This constant, divided by 100, is multiplied by the error value (PV – SP) to obtain the proportional term action. The larger this constant, the greater the effect of the proportional action. A value of 0 disables proportional action. You can change this value at any time, but new value will be used when a rising edge on the **PID** instruction condition is detected, unless you set to ON Rc bit in the control word.

The integral constant (T_i) defines the strength of the integral action. This term integrates the error over time, trying to reset the loop error by adding some amount to the MV variable as time elapses. This constant is given in 10 milliseconds units. The larger this constant, the weaker the effect of the integral action. A value greater of 30000 disables integral action. You can change this value at any time, but new value will be used when a rising edge on the **PID** instruction condition is detected, unless you set to ON Rc bit in the control word.

The differential constant (T_d) also known as derivative constant defines the strength of the differential action. This term tries to reset the error by adding some amount to the MV variable based on the rate change of PV, thus compensating future errors in the control loop. This constant is given in 10 milliseconds units. The larger this constant, the greater the effect of the derivative action. A value of 0 disables derivative action. You can change this value at any time, but new value will be used when a rising edge on the **PID** instruction condition is detected, unless you set to ON Rc bit in the control word.

Continues



PID

Continued

The low pass filter constant (K_{lpf}) controls the strength of the low pass filter applied to the process variable (PV) on the derivative term only. This filter tries to minimize derivative action due to noises in the process variable. The larger this constant, the greater the effect of the filter. A value of 0 disables the filter. You can change this value at any time, but new value will be used when a rising edge on the **PID** instruction condition is detected, unless you set to ON Rc bit in the control word.

The control word at *OP2* + 6 contains control bits used to configure some features in the **PID** action.

Control word	b15 … b4	b3	b2	b1	b0
OP2 + 6	Reserved	**Rc**	**Lm**	**Tp**	**Di**

Di (direction bit, bit 0 of control word) sets the direction for the loop. When ON (1) the reverse action is selected. Reverse action means that the control must increase MV to increase the process variable (heating type). When OFF (0) the forward action is selected. Control must increase MV to decrease the process variable (cooling type).

Tp (equation selection, bit 1 of control word) selects the desired equation. When ON (1) PID type B is selected. When OFF (0) PID type C is selected. Main difference between types is found during large changes in the setpoint. Type B control reaction is fastest, but tends to overshoot. Type C reaction is slower, but eliminates overshoot.

Lm (limit MV, bit 2 of control word) is used to limit MV values between a maximum and minimum value. These limits are set in MV_{max} (*OP2*+7) and MV_{min} (*OP2*+8) parameters. If no limit is set (**Lm** to OFF) a minimum value of 0 and a maximum value of 10,000 will be used and transferred to MV_{max} (*OP2*+7) and MV_{min} (*OP2*+8) parameters.

Rc (parameter recalculation, bit 3 of control word) selects the desired behavior when modifying t_s, K_p, T_i, T_d and K_{lpf} constants. If **Rc** is set to ON (1) new constants will be updated in every instruction execution. If reset to OFF (0) new constants will be updated only on rising edge of the execution condition.

When trying to tune PID constants for your loop, some kind of experimentation is required to find a good constant set. This process generally requires a compromise between control reaction time and loop stability. It is a very good practice to start with a slow reacting but stable control loop without derivative action. Once achieved, you can increase K_p and decrease T_i to get a fastest control loop. Using a K_p of 50, a T_i of 100 and a T_d of 0 can be a good starting point in most cases.

A good knowledge of constant meaning is very important to ease the tuning process. You can consider proportional action as the responsible of turning the PV to the SP when the error is big. An only proportional loop will generally put the PV at a stable point close to the SP, but not at the SP. The integral action will slowly increase the action to correct this error over time. The derivative action act quickly to fast changes in the PV due to perturbations. It is not advisable to use the derivative term to guide normal control because can make your loop very unstable.

3.4. Serial port programming

easyLadder lets you control 3 serial ports for receiving and transmitting serial data. These serial ports are referenced using a number (1-3):

Port 1 This port is the **RS485** included in the RasPICER board. RasPICER board is required to use this port.

Port 2 This port is the **RS232** included in the RasPICER board. RasPICER board is required to use this port.

Port 3 This port is the **ttyAMA0 (serial0)** port included in the Raspberry, and use TTL 3V3 level signals. **If you want to use this port you must free the linux terminal console that runs on system startup by default** on some linux distributions, and disable boot information sent to this port. You can do so by editing /etc/inittab and /boot/cmdline.txt and remove references to the ttyAMA0 (or serial0) port. To use a different device for this port, see parameters in section **4.2. The easyLadder engine**.

Important note: On Raspberry Pi model 3 B, serial port **ttyAMA0** is routed to the Bluetooth module. The TTL port included in GPIO header is referred as **ttyS0** and it is disabled by default. You can enable this port by editing /boot/config.txt and adding **enable_uart=1** option. Please note that **ttyS0** serial port has limited functions. You can disable the Raspberry internal Bluetooth module to route the complete ttyAMA0 serial port back to the GPIO header, as previous Raspberry models. This procedure is not included in this manual, but you can easily find information searching in Internet.

Serial receive and transfer is done through **RXD** and **TXD** instructions. Before calling these instructions you must open the target port by setting values to the corresponding PORTn CFG special device (**SD16**, **SD17** or **SD18**).

PORTn CFG special devices (**SD16**, **SD17** and **SD18**) specify configuration data for each port. A value of 0 in this device closes the port. You have to write a value resulting from adding the chosen baudrate code and the serial format code as shown in the table below. All values are

Baudrates

Code	Baudrate
0x01	110 baud
0x02	300 baud
0x03	600 baud
0x04	1200 baud
0x05	2400 baud
0x06	4800 baud
0x07	9600 baud
0x08	14400 baud
0x09	19200 baud
0x0A	28800 baud
0x0B	38400 baud
0x0C	56000 baud
0x0D	57600 baud
0x0E	115200 baud

Serial format

Code		Serial format
0x00	8N1	8 bits, no parity, 1 stop
0x10	8E1	8 bits, even parity, 1 stop
0x20	8O1	8 bits, odd parity, 1 stop
0x30	8S1	8 bits, space parity, 1 stop
0x40	8M1	8 bits, mark parity, 1 stop
0x80	7E1	7 bits, even parity, 1 stop
0x90	7O1	7 bits, odd parity, 1 stop
0xA0	7S1	7 bits, space parity, 1 stop
0xB0	7M1	7 bits, mark parity, 1 stop
0xB0	7N2	7 bits, no parity, 2 stop
0xC0	7E2	7 bits, even parity, 2 stop
0xD0	7O2	7 bits, odd parity, 2 stop
0xE0	7S2	7 bits, space parity, 2 stop
0xF0	7M2	7 bits, mark parity, 2 stop

indicated in hexadecimal format because you can simply use first digit to indicate serial format and second digit to select baudrate:

For example, writing hexadecimal **#87** to device **SD16** selects 9600 baud, 7 bits, even parity, 1 stop bit for port 1 (RS485 RasPICER port).

Once the port is open, incoming data from this port will be stored in an internal buffer. User has to read received data from this buffer using the **RXD** instruction. **RXD** instruction requires three parameters *OP1, OP2 and OP3. OP1* references the port number. *OP2* designates the destination of the received data and *OP3* is the maximum count of bytes to be copied to the destination.

When **RXD** is executed with an ON condition, available received port data (up to *OP3* bytes) will be copied to *OP2* user buffer. After that, **SD28**, **SD29** or **SD30** will contain the number of bytes copied to the user buffer, and the entire receive buffer will be discarded. For example, when there are 12 bytes available for read and you execute **RXD K1 D0 K5**, a total of 5 bytes will be copied to **D0** device, **SD28** will be set to 5 and all 12 bytes will be removed for the serial buffer.

The byte count parameter *OP3* must be less or equal than 1024, because this is the size of the internal receive buffer.

Data is copied to the user buffer according to the status of the 16BIT MODE special device **S24**. When **S24** is OFF (default) bytes will be copied to independent word devices (in the above example, first byte to D0, second byte to D1...). When **S24** is ON bytes will be copied using the entire word device (first byte to the lower byte of D0, second byte to the higher byte of D0, third byte to the lower byte of D1...). More information on section **3.3.2. 16 bit instruction mode**.

The number of available bytes on the serial buffer can be monitored through special word devices **SD25**, **SD26** and **SD27** (number of available bytes), or bit devices **S25**, **S26** and **S27** (any byte available)**,** depending on the port used. You can call the **RXD** instruction without data available, but is a good practice to check the status of **S25**, **S26** and **S27** before execution.

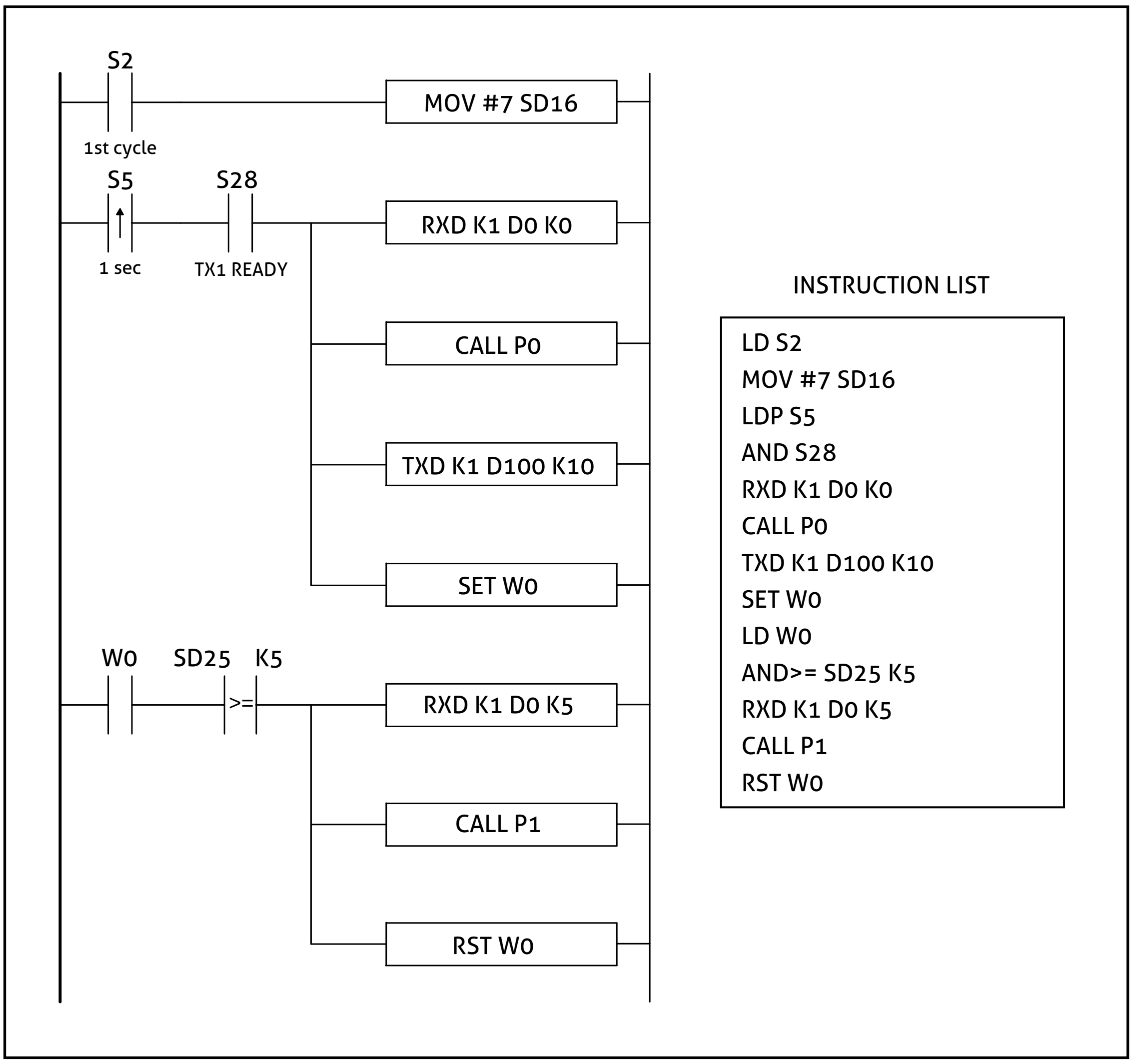
Data transfer to the serial port is done using the **TXD** instruction. **TXD** instruction requires three parameters *OP1, OP2 and OP3. OP1* references the port number. *OP2* designates the source of the data and *OP3* is the count of bytes to be to be transferred to the port.

When **TXD** is executed with an ON condition, *OP3* bytes will be copied from *OP2* user buffer to the internal transmit buffer of the port. The system will transfer these buffer to the physical serial port as a background operation. When **TXD** is executed, special bit devices **S28**, **S29** or **S30** (depending on the port used) will be reset to OFF to indicate the internal transmit buffer is busy (there is some data in the buffer).

The byte count parameter *OP3* must be less than 1024, because this is the size of the internal transmit buffer. Care must be taken when using the **TXD** instruction to not overflow the internal transmit buffer. You can avoid this problem by checking that special bit device **S28**, **S29** or **S30** is ON before issuing the **TXD** instruction. These especial bits indicate that the internal transmit buffer is empty and ready to contain up to 1024 bytes.

Data is copied from the user buffer according to the status of the 16BIT MODE special device **S24**. Consider the instruction **TXD K1 D0 K10**. When **S24** is OFF (default) bytes will be copied from independent word devices (first byte from D0, second byte from D1...). When **S24** is ON bytes will be copied using the entire word device (first byte from the lower byte of D0, second byte from the higher byte of D0, third byte from the lower byte of D1...). More information on section **3.3.2. 16 bit instruction mode**.

Look at the following example:



INSTRUCTION LIST

```
LD S2
MOV #7 SD16
LDP S5
AND S28
RXD K1 D0 K0
CALL P0
TXD K1 D100 K10
SET W0
LD W0
AND>= SD25 K5
RXD K1 D0 K5
CALL P1
RST W0
```

This code is a simple example to implement a basic message interchange between easyLadder and an external serial device. This program uses port 1 to send a message every second and then receives the response. Code at subroutines P0 and P1 is deliberately omitted.

The first line of code configures port 1 to 9600 baud 8N1 on first execution cycle.

On rising edge of S5 oscillator (rising edge occurs every 1 second) and serial port ready to transmit data (S28) the RXD instruction with a byte count of 0 is called. This instruction does not receive any data, just clears the receive buffer to start over for a new message response. This is a good practice because it clears any data received after the last message and before sending new message, decreasing possible faults due to unexpectedly long message responses. After that, subroutine P0 is called to fill message data in D100. Instruction TXD sends this data (10 bytes long) to the serial port. Bit W0 is set to signal that we have sent a message.

When bit W0 is ON (message sent) and at least 5 bytes are received from the serial device (SD25 >= 5), we can receive the data with the RXD instruction and process the message in the P1 subroutine. Bit W0 is reset to indicate receipt of the message.

3.5. MODBUS TCP/IP functions

easyLadder (from version 1.4) includes several MODBUS TCP/IP server/client capabilities, permitting easy communication with other standard MODBUS TCP/IP devices. These capabilities include an embedded MODBUS TCP/IP server and convenient MODBUS TCP/IP client instructions, covering any MODBUS connection scenario.

Using the embedded MODBUS TCP/IP server it is possible to read and write easyLadder devices from external machines, like HMIs (Human-Machine Interfaces), PCs (Personal Computers) and other easyLadder machines.

Using the included MODBUS TCP/IP client instructions you can begin connections to industrial MODBUS devices like distributed IO systems, PLCs, other easyLadder machines and so on.

3.5.1. MODBUS TCP/IP server

The MODBUS TCP/IP server allows sharing device data with standard MODBUS TCP/IP devices like HMIs, Pcs or other easyLadder machines, for example.

This server listens to TCP port 502 (MODBUS standard), waiting connections from external machines. Up to 64 simultaneous connections are supported.

Starting from version 1.7, you can restrict connections to the MODBUS TCP/IP server to only a number of allowed IP ranges. Using this feature it is possible to protect the PLC engine from unwanted remote accesses. See section **5.8.5. TCP/IP security** for details.

Standard MODBUS data model divides sharing registers into four kinds:

- Discrete Inputs (read only bit memory)
- Coils (read/write bit memory)
- Input registers (read only word memory)
- Holding registers (read/write word memory)

The entire easyLadder memory is shared through the MODBUS TCP/IP server. This MODBUS implementation makes no difference between Discrete Inputs and Coils (bit memory) and between Input Registers and Holding Registers (word memory). Nevertheless, bit memory and word memory uses a fully isolated addressing, meaning that easyLadder bit devices (W, H, X, Y...) are addressed through MODBUS bit addresses (Discrete Inputs or Coils) and easyLadder word devices (D, AX, AY...) are addressed through MODBUS word addresses (Input Registers or Holding Registers).

The MODBUS TCP/IP server implements MODBUS data access functions 02 (Read Discrete Inputs), 01 (Read Coils), 05 (Write Single Coil), 15 (Write Multiple Coils), 04 (Read Input Register), 03 (Read Holding Registers), 06 (Write Single Register) and 16 (Write Multiple Registers).

The following tables illustrates the MODBUS memory mapping:

Bit devices (access through Discrete Inputs or Coils MODBUS memory)

Memory area	First MODBUS address	Devices	Usage
W	0	10000	Work bits
H	10000	10000	Retentive work bits
X	20000	512	Digital inputs
Y	25000	512	Digital outputs
T (TC)	30000	256	Timer contact (100 ms precision)
TH (THC)	35000	256	Timer contact (10 ms precision)
C (CC)	40000	256	Counter contact
S	45000	256	Special bit devices

Word devices (access through Input Registers or Holding Registers MODBUS memory)

Memory area	First MODBUS address	Devices	Usage
D	0	10000	User data devices
AX	10000	256	Analog inputs
AY	15000	256	Analog outputs
T (TD)	20000	256	Timer count value (100 ms precision)
TH (THD)	25000	256	Timer count value (10 ms precision)
C (CD)	30000	256	Counter value
SD	35000	32	Special word devices

3.5.2. MODBUS TCP/IP client instructions

The included MODBUS TCP/IP client instructions give the power to connect to industrial MODBUS devices like distributed IO systems, PLCs, other easyLadder machines and so on.

The MODBUS TCP/IP client connection is managed through the **MODBUS** instruction. This instruction is used to configure and connect to a MODBUS TCP/IP device.

The **MODBUS** instruction requires one parameter *OP1*. This parameter *OP1* designates the first device of a buffer used to configure and reference the MODBUS connection. A total of **8 consecutive word** devices will be used in this area. You need to set these parameters before instruction execution.

OP1	**Status**	MODBUS connection status
OP1+1	**Error**	MODBUS connection error
OP1+2	**IP1**	First byte of peer IP address
OP1+3	**IP2**	Second byte of peer IP address
OP1+4	**IP3**	Third byte of peer IP address
OP1+5	**IP4**	Forth byte of peer IP address
OP1+6	**TCP port**	TCP port (usually 502)
OP1+7	**Unit**	MODBUS destination unit

Word device *OP1* contains the **Status** of the MODBUS connection. This word is a mask of several bits according to the following table:

Bit 0 [OP1].0	**Ready**	MODBUS connection ready
Bit 1 [OP1].1	**Complete**	MODBUS transfer complete
Bit 2 [OP1].2	**Error**	MODBUS error detected
	...	
Bit 6 [OP1].6	**Enabled**	MODBUS connection enabled
Bit 7 [OP1].7	**Connected**	Connected to the MODBUS device

You can use up to 64 **MODBUS** instructions to connect to up to 64 MODBUS devices. These instructions cannot share the same *OP1* device buffer, just because this *OP1* device is used to reference the MODBUS connection in other MODBUS related instructions.

When the MODBUS instruction is executed (with an ON or OFF condition), value of Status (*OP1*) and Error (*OP1+1*) words are refreshed according to the status of the MODBUS connection. The **Ready** bit indicates that the MODBUS connection is ready for starting MODBUS Read or Write operations. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction.

MODBUS read and write operations are performed as a background process. When any MODBUS read or write operation is complete and the **MODBUS** instruction is executed, the **Complete** bit or **Error** bit will be set according to the success of the operation. On MODBUS read operations, destination devices will be refreshed only when the **MODBUS** instruction is executed. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

When any error is detected during connection, read or write operation, the **Error** bit (*OP1.2*) will be set and the **Error** word (*OP1+1*) will contain information about this error. The following table contains possible error values:

0	No error
1	MODBUS exception 1 (illegal function)
2	MODBUS exception 2 (illegal data address)
3	MODBUS exception 3 (illegal data value)
4	MODBUS exception 4 (slave device failure)
5	MODBUS exception 5 (acknowledge)
6	MODBUS exception 6 (slave device busy)
8	MODBUS exception 8 (memory parity error)
10	MODBUS exception 10 (gateway path unavailable)
11	MODBUS exception 11 (gateway target failed to respond)
20	Timeout contacting MODBUS device
21	Frame error from MODBUS device
22	Unable to connect to MODBUS device
23	No free connections (more than 64 MODBUS instructions)
24	R/W operation without matching MODBUS instruction
25	Exceeded max R/W devices in MODBUS Read/Write

When **MODBUS** instruction is executed with an ON condition, easyLadder will try to connect to the device using the IP / port specified in the *OP1* area. When connection is lost or unsuccessful, easyLadder will retry the connection indefinitely. During instruction execution, IP address and port can be modified, but new values will be used only when an OFF to ON condition is detected in the **MODBUS** instruction.

When **MODBUS** instruction is executed with an OFF condition, easyLadder will close the connection to the MODBUS device.

Once the **MODBUS** instruction opens the connection with the external device, and the **Ready** bit of *OP1* device (Bit 0) is ON indicating connection ready, you can issue any of the MODBUS read or write instruction. Is a good practice to check the status of this bit before starting a MODBUS register Read or Write instruction. Executing any Read or Write instruction when the **Ready** bit is OFF is allowed, but the instruction will not be executed (no error indication), so the user cannot know if the read or write operation is started.

The **MODBUS** instruction must be executed in every scan (with an ON or OFF condition), prior to any read or write instruction. By the other way, read and write instructions can be used in subroutines.

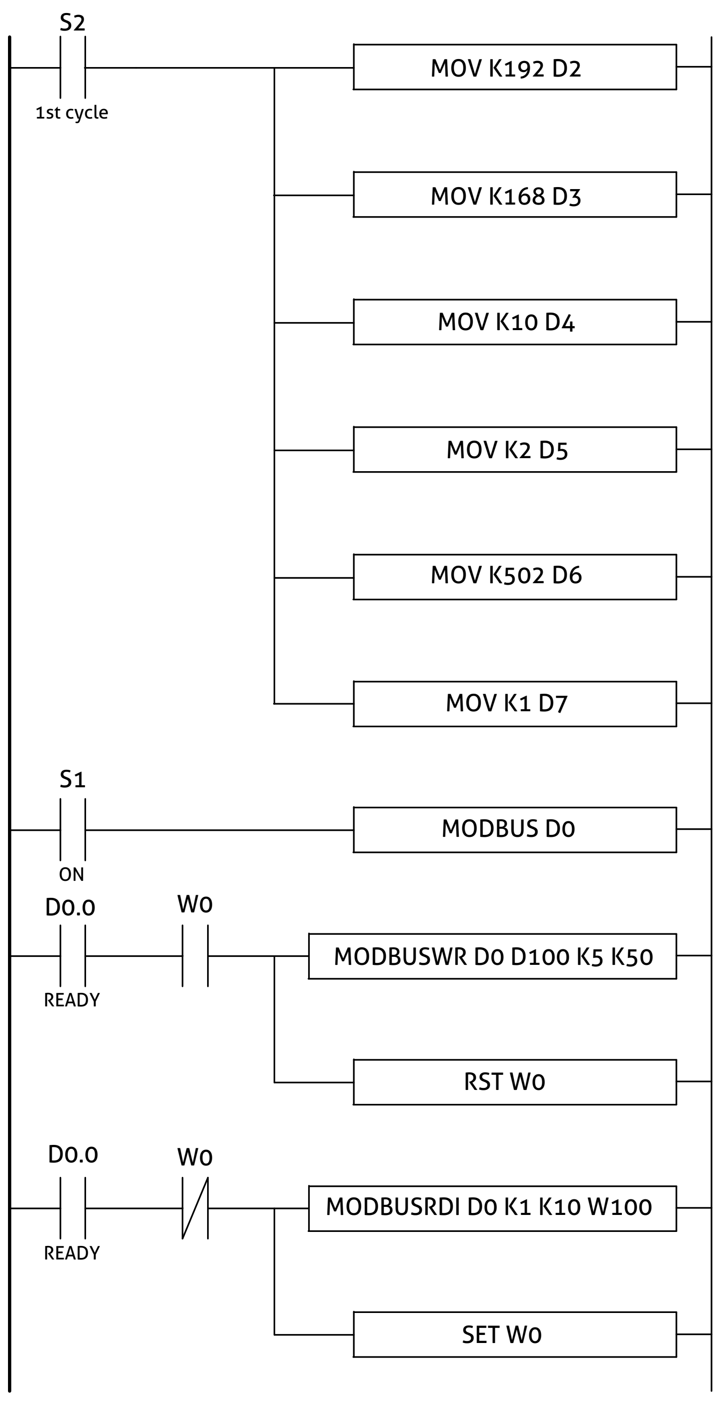
Updates to status word, error word and destination devices are performed when the corresponding **MODBUS** instruction is executed after operation completes. On completion, the **Complete** bit or **Error** bit in *OP1* will be set according to the success of the operation. Please note that the **Complete** bit will be set only 1 cycle after the completion of the read or write operation. After this cycle, the **MODBUS** instruction will reset automatically the **Complete** bit.

Available MODBUS read and write instructions are:

Instruction	**Function**
MODBUSRDI	Read Discrete Inputs
MODBUSRC	Read Coils
MODBUSRIR	Read Input Registers
MODBUSRHR	Read Holding Registers
MODBUSWC	Write Coils
MODBUSWR	Write Registers

These instructions are used to transfer bit devices or word devices to or from the connected MODBUS server. Please refer to the above instruction reference to obtain details about these instructions.

The following example illustrates a basic method to write devices D100-D104 to the holding registers addresses 50-54, and read discrete inputs 1-10 to devices W100-W109. The MODBUS device is found at address 192.168.10.2, port 502, unit 1.



INSTRUCTION LIST

```
LD S2
MOV K192 D2
MOV K168 D3
MOV K10 D4
MOV K2 D5
MOV K502 D6
MOV K1 D7
LD S1
MODBUS D0
LD D0.0
AND W0
MODBUSWR D0 D100 K5 K50
RST W0
LD D0.0
ANDI W0
MODBUSRDI D0 K1 K10 W100
SET W0
```

In the above example, we are using D0 as the MODBUS configuration buffer. In the first scan (S2) the configuration data buffer is filled with the IP address, port number and Unit.

The MODBUS instruction is then executed in every scan, creating the connection to the MODBUS device.

Once the connection is ready for read and write operations, bit D0.0 (**Ready** bit) will be set to ON. When this happens, we use bit W0 to select between read and write operation. If W0 is set, the MODBUSWR (MODBUS write register) instruction is executed, writing 5 registers starting from D100 to the MODBUS address 50. Then, the W0 bit is reset to OFF, so the next operation will be the other read operation.

When operation completes, bit D0.0 (**Ready** bit) will be set to ON again. If W0 is OFF, the MODBUSRDI (MODBUS Read Discrete Input) instruction is executed, reading 10 registers starting from MODBUS address 1 and storing values to devices W100 to W109. Then, the W0 bit is set to ON, so the next operation will be again the write operation.

Please note that when any MODBUS read or write instruction is executed, the **Ready** bit (D0.0 in this case), will be immediately reset to OFF, indicating that the operation is in progress.

3.6. RasPICER, GPIO and I2C expanders I/O allocation

The RasPICER board provides a set of inputs and outputs with fixed device mapping. Even if you are not using this board, you must respect this device reservation.

Assigned devices for the RasPICER board are found in the following table:

Devices	Description
X0 – X7	Digital inputs
Y0 – Y3	Digital outputs (transistor outputs)
Y4 – Y7	Digital outputs (relay outputs)
AX0 – AX1	Analog inputs (0...20 mA)
AY0 – AY1	Analog outputs (0...20 mA)

Raspberry Pi provides a number or input and output pins in the GPIO connector. These GPIO ports can be used to expand your available digital IOs for the easyLadder PLC. Please note that these ports use 3V3 level signals, so it must require generally some kind of signal conditioning to interact with outer systems.

You can allocate X and Y PLC devices to the ports using **easyLadder studio** software. In order to configure easyLadder for your expansion modules, refer to section **5.8. PLC configuration**.

In the following image you can view the list of available GPIOs for device assignment. All pins are configurable as output or input (with or without pullup and pulldown resistors). GPIO7, GPIO8, GPIO9, GPIO10 and GPIO11 are used for SPI communication. **You cannot use these pins if the RasPICER board is present** or you need to use the SPI port.

	1	2	
	3	4	
	5	6	
GPIO4	7	8	
	9	10	
GPIO17	11	12	**GPIO18**
GPIO27	13	14	
GPIO22	15	16	**GPIO23**
	17	18	**GPIO24**
(SPI) GPIO10	19	20	
(SPI) GPIO9	21	22	**GPIO25**
(SPI) GPIO11	23	24	*GPIO8 (SPI)*
	25	26	*GPIO7 (SPI)*
	27	28	
GPIO5	29	30	
GPIO6	31	32	**GPIO12**
GPIO13	33	34	
GPIO19	35	36	**GPIO16**
GPIO26	37	38	**GPIO20**
	39	40	**GPIO21**

Additionally, you can provide other I/Os to your system, by using up to 32 external I2C I/O port expanders. Raspberry Pi includes an I2C serial port in the GPIO header. There are a number of Raspberry HAT boards in the market including I2C devices. These devices are connected to the Raspberry I2C bus and can provide a number of 8 or 16 I/O ports per device.

You can allocate X and Y PLC devices to these expanders using **easyLadder studio** software. In order to configure easyLadder for your expanders, refer to section **5.8. PLC configuration**.

3.7. Extension I/O units

In addition to the inputs and outputs provided by the RasPICER board and the GPIO ports on the Raspberry Pi, you can use Ethernet extension modules to expand your system I/O capabilities.

You can connect up to 32 Ethernet I/O modules. At this moment easyLadder PLC supports Ethernet modules from **SHJ Electronic Co, Ltd.** (http://www.shjelectronic.com). We selected these modules due to the high quality / cost ratio. Other brands may be supported in a near future.

Currently available SHJ Electronic modules are:

- S6301 (8 Digital In, 8 Digital Out)
- S6302 (16 Digital In)
- S6303 (16 Digital Out)
- S6305 (5 Power Relay Out)
- S6316 (16 Relay Out)
- S6116 (16 Analog In - 12 bit ADC)
- S6216 (16 Analog In - 16 bit ADC)

Modules must be properly configured before use on easyLadder. Configuration requires the **UDP server** mode, responding to UDP port 502 (MODBUS). Additionally you must configure IP parameters (IP address, Subnet mask and Gateway) according to your network.

You can allocate input and output devices for each module using **easyLadder studio** software. In order to configure easyLadder for your expansion modules, refer to section **5.8. PLC configuration**.

The system behavior under extension module failure can be configured. You can select whether to STOP the PLC when any module stops responding. Module communication failure can be monitored through special SD devices:

SD20	EXTERR0	Extension module error bitmask (modules 1-16)
SD21	EXTERR1	Extension module error bitmask (modules 17-32)

These devices contain the error bit mask for all extension units. A value of 1 in the corresponding bit indicates that module is not responding to queries. **SD20** contains error bits for modules 1 to 16, and **SD21** for modules 17 to 32.

3.8. PLC errors

easyLadder can detect many error conditions. Depending on the error severity (fatal error or warning), the PLC will switch to STOP mode, requiring a manual RUN or clear error command to restart normal operation.

You can monitor the error status of your PLC using the easyLadder studio software. Refer to **section 5.6. Online operation** for more details.

This is the list of available errors:

Error	Description	STOP
Instruction error	An invalid (unknown) instruction was found.	YES
FOR / NEXT error	Invalid FOR and NEXT nesting	YES
FOR without NEXT	A NEXT not found for the a FOR instruction	YES
No END instruction	The program finished without an END instruction	YES
Program pointer	Used program pointer (Pn) is not found	YES
Call stack error	RET instruction not found or exceeded CALL nestings	YES
Hardware IO error	Error contacting RasPICER board	YES
Scan time too long	Instruction watchdog error. Executed instructions in the scan exceeds the configured watchdog value.	YES
Invalid CRC	Program is corrupted	YES
Extension module	Extension module not responding	Configurable
I2C expander	I2C I/O expander not responding	Configurable
Device out of range	An instruction parameter contains a reference to a device out of range.	NO

4. Raspberry Pi software

4.1. Installing easyLadder

If your Raspberry Pi has connection to the Internet, you can download our software installer to ease the entire process. To get the installer, open a local terminal or a remote SSH session to your Raspberry and type the following commands:

```
cd
wget http://www.ferrariehijos.com/easyladdersetup
chmod +x easyladdersetup
sudo ./easyladdersetup
```

The installer will query for your desired configuration and download the required files. Files will be installed in **/opt/effesoftware**.

All files are compiled for the RASPBIAN JESSIE, our preferred Linux distribution. Please contact us if you want to use other Linux distribution.

If your Raspberry is not connected to the Internet, or you want to do a manual setup, you can download the PLC engine pack with all files and sample source code from:

http://www.ferrariehijos.com/easyLadder

Copy the downloaded **easyladderX.tar** (X is 1 or 2 depending on your raspberry model) file to your home directory, and open a local terminal or a remote SSH session to your Raspberry and type the following commands (do not copy **bold** commands, are only for help):

```
sudo sh
cd
tar -xvf easyladderX.tar -C /opt  (replace X with 1 or 2)
cp /opt/effesoftware/easyladderd /etc/init.d/easyladderd (with RasPICER board, or)
cp /opt/effesoftware/easyladderdn /etc/init.d/easyladderd (without RasPICER board)
update-rc.d easyladderd defaults
service easyladderd start
exit
```

This procedure will copy the software package to /opt/effesoftware directory and will run easyLadder engine as a daemon, so it will be automatically loaded on system startup.

4.2. The easyLadder engine

The PLC engine is managed through the **easyLadder** program. When using the **easyladdersetup** installer, this file is trasferred to the **/opt/effesoftware** directory.

easyLadder uses the **plcdata** directory (found in the **/opt/effesoftware/plcdata** by default) to store PLC program files, retentive device files and engine log file. Sometimes it is very useful to view the engine log file to monitor system working. You can find this log file in the **plcdata** directory, **/opt/effesoftware/plcdata/easyladder.log** by default.

Normally, the **easyLadder** program is loaded as an *init.d* service (*daemon)* during system startup. This is done using the **easyladderd** script found at **/etc/init.d** directory. You can also run the engine manually by executing **easyLadder** directly with superuser privileges (using *sudo* command, for example).

You can start or stop manually the PLC engine service using:

```
sudo service easyladderd start
-or-
sudo service easyladderd stop
```

When using the RasPICER board with the watchdog function active, a PLC engine stop can result in a system poweroff due to a watchdog timeout. Disable the watchdog before stopping using **/opt/effesoftware/raspicer -w0** command or doing three power button presses on the RasPICER board. The watchdog LED indicator in the RasPICER board will blink when watchdog is temporally disabled.

IMPORTANT NOTE: If using the RasPICER board, do not load the **raspicer** *daemon* (**raspicerd init.d** script). The easyLadder PLC engine includes his own *daemon*, replacing the general **raspicer** *daemon*. This *daemon* lets you to access every RasPICER feature and additionally gives control of the embedded PLC variables (devices). You can use the **raspicer** utility to control the board, but do not execute it with the **--startdaemon** option.

The **easyLadder** program can be executed using parameters. When using the **init.d** script to run the engine, you can set parameters by editing **/etc/init.d/easyladderd** script and modifying DAEMON_ARGS variable at the start of the file. Available command line parameters are:

Usage: **easyladder** [option ...]

-v, --verbose	When using this option, engine log messages are directed to stdout, in addition to the log file. This option is useful for debug purposes.
-n, --noraspicer	Disable RasPICER board support. Use this option when not using the RasPICER board, otherwise engine will shutdown with an *unable to contact the RasPICER board* error message.
-l KEY, --license KEY	This option is used to register easyLadder. This license key is generated using the system key for your system. The system key is shown in the log file during engine startup. More information about registering is found at http://www.ferrariehijos.com/easyLadder. Without license, easyLadder engine will run without limitations for a period of 15 minutes. After this time, the engine will be placed in STOP mode, requiring a manual RUN command to start it again.

-s DEV, --serial DEV	Use this option to specify device used for serial Port 3. This device is **/dev/serial0** by default.
-d DIR, --basedir DIR	Use this option to specify the location of the **plcdata** directory, where program, data, configuration and log files are stored. By default, when this parameter is not present, the **plcdata** is located in the easyLadder executable directory. Set this parameter if you need to store writable data in other location.
-h, --help	Shows command line help information.

4.3. The raspicer utility

The **raspicer** utility is provided as a part of the easyLadder and the RasPICER board software packages. This is a command line utility used to control features of the RasPICER board, such as monitor the status of the board, set outputs, adjust clock and configure hardware values, but also to control the PLC engine through the command line.

IMPORTANT NOTE: If using the RasPICER board, do not load the **raspicer** *daemon* (**raspicerd init.d** script). The easyLadder PLC engine includes his own *daemon*, replacing the general **raspicer** *daemon*. This *daemon* lets you to access every RasPICER feature and additionally gives control of the embedded PLC variables (devices). You can use the **raspicer** utility to control the board, but do not execute it with the **--startdaemon** option.

The available command line parameters for the raspicer utility are:

Usage: **raspicer** [option ...]

General options:

-s, --status	Displays board I/O or PLC status information. **RasPICER required.**
-q, --quiet	Runs raspicer in quiet mode. No text is displayed.
-h, --help	Shows command line help information.
-w0, --watchdog0	Disables RasPICER shutdown due to watchdog timeout. When the watchdog is active and you need to stop the control application or daemon, you can temporarily disable the watchdog engine by using this command. When the watchdog is temporarily disabled the watchdog LED will blink. You can reactivate the watchdog issuing the –w1 command. Refer to section **3.11. Power manager** (RasPICER user manual) for more information. **RasPICER required.**
-w1, --watchdog1	Re-enables RasPICER shutdown due to watchdog timeout. Refer to section **3.11. Power manager** (RasPICER user manual) for more information. **RasPICER required.**
-r, --reloadwd	Reloads RasPICER special watchdog counter. Refer to section **3.11. Power manager** (RasPICER user manual) for more information. **RasPICER required.**
-p PAR, --powerctrl PAR	Sets RasPICER power control options. **Do not use with easyLadder.**
-c, --viewconfig	Displays RasPICER flash configuration parameters. See parameters bellow. **RasPICER required.**

-f CFG, --setconfig CFG

Sets RasPICER flash configuration parameters. New parameters are saved to flash memory. **RasPICER required.**

Use '--setconfig parameter1=value1,parameter2=value2, ...' format.

Valid parameters are:

DisableAutoPowerOn (0-1). This parameter configures the RasPICER board to power the Raspberry when external VDC power is applied to the terminals (**AUTO POWER ON** power manager function). A value of 1 disables the function.

DisableAutoPowerOff (0-1). This parameter configures the RasPICER to remove power to the Raspberry when the **poweroff** state is detected (**AUTO POWER OFF** power manager function). A value of 1 disables the function.

DisableWatchdog (0-1). This parameter configures the RasPICER to remove power to the Raspberry when a watchdog timeout occurs (**WATCHDOG** power manager function). A value of 1 disables the function.

UseWatchdogCommand (0-1). This parameter configures the preferred method to reload the watchdog. Using a value of 0, any SPI command reloads the watchdog. Using a value of 1, the special watchdog reload command is required.

StartupTime (0-65535). This parameter specifies the time (in 100 ms units) needed for the Raspberry Pi system to start, thus the maximum time after a power up required to issue the first watchdog reload. If no watchdog reload occurs passed this time from the power up, the system will restart. This time must be set according to your system, depending on the time required to launch your control application or daemon.

WatchdogTime (0-65535). This parameter specifies the maximum time (in 100 ms units) required between watchdog reloads, after the first (startup) reload. If no watchdog reload occurs passed this time, the system will restart.

InputFilter (0-255). This parameter specifies the time filter value (in 1,54 ms units) for digital inputs.

CalibOutCurrent0 and **CalibOutCurrent1** (0-65535). These parameters are used to calibrate analog current outputs AY0 and AY1. In order to calibrate the outputs, set the analog output to generate 20 mA. Increasing the parameter decreases the output current.

CalibInCurrent0 and **CalibInCurrent1** (0-65535). These parameters are used to calibrate analog current inputs AX0 and AX1. In order to calibrate the inputs, connect a 20mA analog source to the input. Increasing the parameter decreases the read current.

--startdaemon

Starts RasPICER controller daemon. **Do not use with easyLadder.**

--stopdaemon

Stops RasPICER controller daemon. **Do not use with easyLadder.**

RasPICER I/O options:

--yN VAL	Set value for RasPICER board digital output N. **Do not use with easyLadder. Use --plcsetbit parameter instead.**
--ayN VAL	Set value for RasPICER board analog output N. **Do not use with easyLadder. Use --plcsetword parameter instead.**
--rcxN	Reset counter value for RasPICER board digital input N. **Do not use with easyLadder.**
-d VAL, --do VAL	Sets value for entire RasPICER board digital output. **Do not use with easyLadder. Use --plcsetbit parameter instead.**

RasPICER RTC options:

-v, --viewrtc	View current clock from RasPICER RTC. **RasPICER required.**
-g, --getrtc	Transfer RasPICER RTC to system clock. **RasPICER required.**
-t, --setrtc	Transfer system clock to RasPICER RTC. **RasPICER required.**
-y, --syncrtc	Automatic RTC synchronization. **This function is always ON with easyLadder. Do not use.**

easyLadder options (a running easyLadder PLC engine is required):

--plcrun	Places easyLadder PLC in RUN mode.
--plcstop	Places easyLadder PLC in STOP mode.
--plcsetbit PAR	Sets PLC bit device value.
	Use '--plcsetbit Device1=Value1,Device2=Value2, ...' format.
--plcsetword PAR	Sets PLC word device value.
	Use '--plcsetword Device1=Value1,Device2=Value2, ...' format.
--plcgetbit PAR	Gets PLC bit device value.
	Use '--plcgetbit Device1,Device2, ...' format.
--plcgetword PAR	Gets PLC word device value.
	Use '--plcgetword Device1,Device2, ...' format.
--unblocktcpip	Disables PLC TCP/IP security. When you restricted the allowed incoming TCP/IP addresses for easyLadder Studio, and the easyLadder engine is no longer accessible, you can execute this option, so the TCP/IP security will be disabled.

4.4. Optimizing the PLC engine.

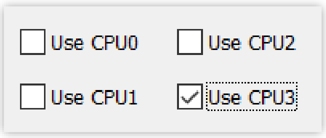
When using multicore processors (**Raspberry Pi2 and Pi3**), the PLC engine performance can be optimized using CPU isolation. You can isolate a single CPU for the PLC engine only.

It is possible to configure your Linux system to isolate one CPU during boot. When this is done, the system will not assign processes to this core. easyLadder lets you select which CPU are assigned to the PLC engine thread, so selecting ONLY the isolated core in the PLC parameter dialog (**section 5.8.1**), the PLC engine will boost performance being the unique process for the CPU and, additionally, you will obtain a more constant scan cycle time.

The PLC engine is single threaded, so you will not obtain more performance by isolating more than one core.

In order to isolate the CPU for the PLC engine, you must edit **/boot/cmdline.txt** file in your Raspberry Pi and add at the end of the **root=** line the sentence **isolcpus=3**. This way CPU3 will be reserved for the PLC engine. This setting will be effective after a system reboot.

Once the CPU is isolated in the Linux kernel, you can select ONLY this CPU in the PLC parameter dialog (**section 5.8.1**).


☐ Use CPU0 ☐ Use CPU2
☐ Use CPU1 ☑ Use CPU3

5. EASYLADDER STUDIO PROGRAMMING SOFTWARE

5.1. Introduction

easyLadder studio is a powerful but lightweight ladder editor for the easyLadder PLC. With this application you can easily manage your PLC programs using an easy to use graphical environment.

With easyLadder studio you can:

- Write your PLC program.
- Transfer the program to the PLC using a network connection or export program files to copy manually to the Raspberry Pi.
- Read program currently running on the PLC
- Protect your PLC program with password
- Monitor and modify PLC device values in real time
- Modify a part of the program and transfer changes without stopping control (Online edit)
- Assign comments and descriptions to used PLC devices to ease program readability
- Divide the program in sections to ease program readability
- Assign input and output devices to external expansion modules, GPIOs and I2C expanders
- Configure running parameters for the PLC

Starting from version 1.7, you can restrict remote connections to the PLC to only a number of allowed IP ranges. Using this feature is possible to protect the PLC engine from unwanted remote accesses. See section **5.8.5. TCP/IP security** for details.

5.2. Installing easyLadder studio

easyLadder is designed for Microsoft Windows. Windows XP, Vista, 7, 8, 8.1 and 10 are supported.

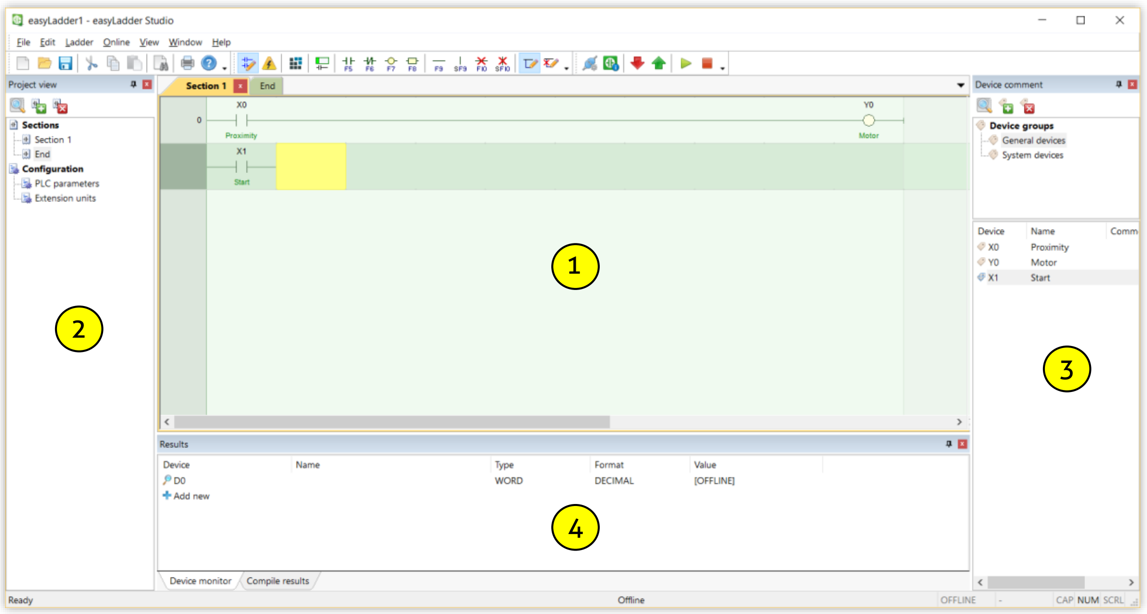
You can download easyLadder installer at http://www.ferrariehijos.com/easyLadder.

Installation process is straightforward. Simply execute the installer and follow on-screen indications. Setup program will create shortcuts for calling easyLadder studio.

5.3. easyLadder studio overview

This section provides basic information about software features. Some kind of general knowledge about Microsoft Windows applications is required before using this software. Concepts as project saving and loading will not be covered in this guide.

Main easyLadder studio window is divided into several areas:



Device	Name	Comm
X0	Proximity	
Y0	Motor	
X1	Start	

Device	Name	Type	Format	Value
D0		WORD	DECIMAL	[OFFLINE]
Add new				

- **Editor view** (Pos. 1). This window is used to view and modify the PLC program. Each section is shown in a different tab. You can switch between open sections by pressing the corresponding tab.
- **Project view** (Pos. 2). This tree view contains the section list for your program and configuration parameters.
- **Device comment view** (Pos. 3). This view contains comments for the PLC devices. You can assign a friendly name and a description for your used devices. Comments for system devices (**S**, **SD**) are automatically assigned when creating a new project. Device comments are arranged into groups to facilitate device management. You can create your own device group by right clicking device group window.
- **Results view** (Pos. 4). This view contains two tabs: Device monitor tab and compile results tab. Using device monitor tab you can monitor your desired PLC devices in real time. This tab contains the list of devices to monitor during online sessions. You can add or remove devices to this tab. Compile results tabs is filled with errors and warnings during program compilation.

5.4. Writing your program

An easyLadder program is a set of instructions ended with an **END** instruction. As noted in section **3.1. Introduction to ladder programming**, the scan cycle is finished when an **END** or **FEND** instruction is executed. When you need to include subroutine code that it is not executed in the main scan cycle, you need to put a **FEND** instruction at the end of the main scan code. After this **FEND** you can place every subroutine code and, at the end of all the code, you place the **END** instruction.

To ease program reading, you can divide your program in sections. Sections are parts of code with a customizable name. These parts are joined together in the order specified in the project view tree to form the entire program. Sections have no influence on the easyLadder PLC engine function. Sections are only provided for program organization.

When you start with a blank project, the framework will create two sections. A blank section and a section named End. The End section, as the name suggests, includes only the END instruction. You are free to delete, rename or modify this section.

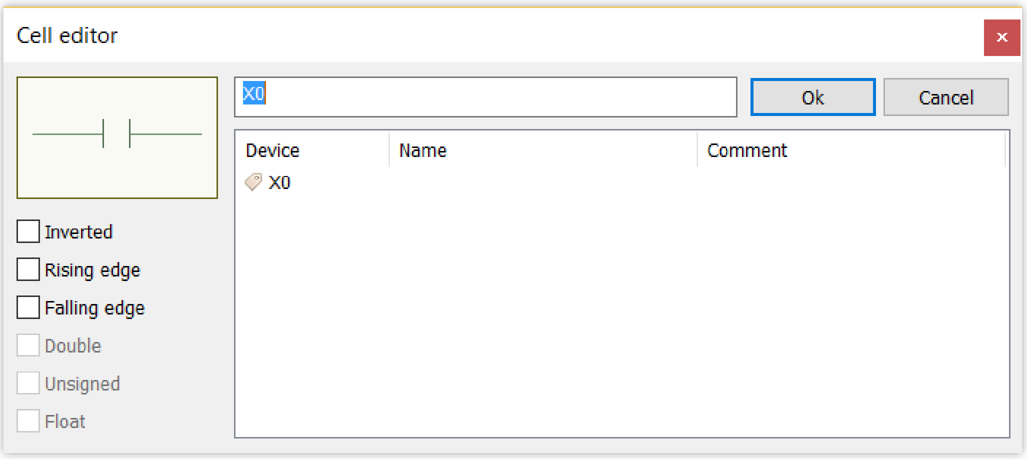
Program sections are shown in the Project view. You can view, add, delete, rename, move up and move down any section using the context menu. Right click the section tree to open the context menu.

The ladder program is composed by contact instructions and coil instructions. Contact instructions form a logic block to command a number of coil instructions. Each logic block with their corresponding coil instructions is called **network**. A section can contain any number of networks.

The editor space is divided in **cells**. Instructions are placed in these cells. Each instruction can use one or more cells depending on the instruction size.

Edit mode. In order to modify your program, easyLadder must be set to **Edit mode**. When this mode is not selected, any modification of the program is forbidden. You can enable or disable this mode using the Edit menu or clicking the **Edit mode** button in the toolbar.

Cell edition. When the editor is in edit mode, you can place instructions by selecting the destination cell and using the toolbar shortcuts provided. Also you can edit any cell by double clicking it. When editing the cell, a cell edition dialog will appear. Using this dialog you can choice the desired base instruction and select modifiers for the instruction. You can also enter instruction code directly.



Device	Name	Comment
X0		

Signal wires. Once your logic block is ready, you need to wire signals between ladder cells. The most practical way is to use the **Draw wire / wire delete mode** by selecting the icon shown at the left. This mode lets you to draw wires using the mouse. To draw or delete a wire you must click at the start point of the wire and release the click at the end of the wire. A sample wire will be shown while dragging. You can also use single horizontal wire and vertical wire icons in the toolbar. All commands are accessible from the Edit menu.

Network comments. The editor allows network comments. These comments are shown at the top of the network and gives information about the network to increase program readability. To place a network comment you must select any cell in the network and click the network comment toolbar icon. A popup window with the network comment will be shown.

Program compile. When you add or modify any network in your program, the editor will show this network using a different background color to indicate that this network was modified. After any change, the resulting network must be **compiled** to check for errors and translate to PLC code. Compilation is made through the Compile button in the toolbar, the Edit menu or using the context menu shown right clicking the editor. You can also undo the changes before compile using the menu. If any error is detected during code compilation a popup window will indicate the source of the error. Additionally, the compile results tab will show additional information about compilation process.

5.5. Device comments

easyLadder supports device comments. Device comments are useful to ease program interpretation. Any PLC device can have a name and a comment. This name is shown on the ladder editor in addition to the PLC device name.

Device comments are arranged in groups. You can create several groups and include any number of comments in each group.

The device comment view lets you manage these comments. At the top of this view is the device group tree. By clicking a device group, you can view all devices in that group at the bottom of the view. Right clicking the tree you can add, remove or rename the group through the context menu.

In order to add a device comment to a group, you can select the group from the tree and right click the bottom list to popup the context menu. Also, you can use the add icon located at the top of the view.

Comments for system devices (**S**, **SD**) are automatically assigned when creating a new project.

Device comments can be transferred to the PLC, if needed. This way these device comments will be recovered when getting the program from the PLC.

5.6. Online operation

easyLadder studio can connect to the PLC through the network connection. Acting this way it is possible to transfer the program to or from the PLC, monitor device data, change PLC mode to RUN or STOP and view status of the PLC.

Online mode. You can switch to online mode pressing the icon shown at the left or using the main menu. In this mode, easyLadder studio connects to the PLC through the network connection to monitor PLC devices. The editor window will display status and values for all devices in your code. Additionally, you can use the device monitor tab to monitor other devices. You can change device values by right clicking any device in the editor window and selecting the corresponding option from the context menu.

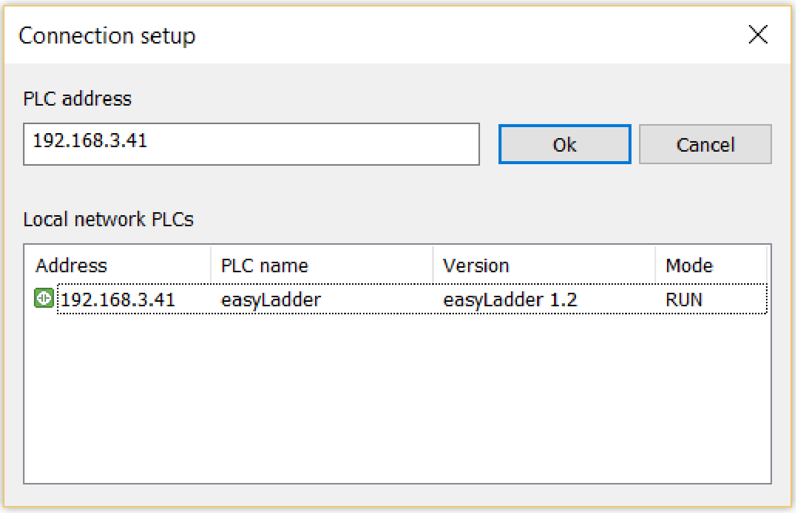
Online edit. When easyLadder studio is in online mode, you can switch to online edit mode by clicking the **edit mode** button. When both online and edit modes are active, the system enters the **Online edit** mode. In this mode, any change to the program will be transferred to the PLC during compile. The PLC will not be switched to STOP mode and program will continue with the new code. A red background in editor windows indicates this mode.

Care must be taken when using this mode with pulsed instruction or contacts. New network code with pulse conditions will be executed (in the first cycle after modify) considering that last **executing condition was OFF**.

Before entering **Online edit** mode, easyLadder studio will compare editor program with the program on the PLC. When programs are not exactly the same, Online edit mode will not be activated.

Connection setup. The first time you use any online option, a connection dialog will appear. You can also access to this dialog by clicking the icon shown at the left or selecting the option in the main menu. You must enter the IP of the easyLadder PLC in the PLC address field.

For your convenience, easyLadder attempts an automatic detection of any easyLadder PLCs in your local network. The bottom list includes detected PLCs in your network. You can click any detected PLC to use its IP address.


Connection setup

PLC address: 192.168.3.41

Local network PLCs

Address	PLC name	Version	Mode
192.168.3.41	easyLadder	easyLadder 1.2	RUN

PLC status. This option is used to monitor status of the PLC. You can view current PLC mode (RUN, STOP), system clock, PLC errors and other status related to the RasPICER board, if present.

At the bottom of the dialog you can find the error list. Errors in this list can be active errors or inactive errors. Active errors represent error conditions not yet solved. Inactive errors signify the past error history, or failures not currently present in the system. When switching from STOP to RUN mode the system will try to clear existing errors, but these errors will remain as inactive until a system shutdown or a manual clear error command in used.

Write to PLC. Using this command you can transfer your program to the PLC. When executing the command, you can specify which areas you want to transfer (program memory, parameter data or device comments).

WARNING: When writing the program memory, the PLC must be switched to STOP mode. Verify that is safe to STOP the control before issuing the write.

Read from PLC. Using this command you can transfer the current PLC program to easyLadder studio. When executing the command, you can specify which areas you want to transfer (program memory, parameter data or device comments). If the PLC program is protected by password, you must enter this password before data transfer.

Switch to RUN mode. You can remotely place the PLC in RUN mode using this button. When PLC is in RUN mode, your program is executed. When your PLC is stopped due to some PLC error, switching to RUN mode will try to reset the error condition. When the PLC is powered, it is automatically placed in RUN mode.

Switch to STOP mode. You can remotely place the PLC in STOP mode using this button. When PLC is in STOP mode, no program is executed. Even if you manually STOP the control, the PLC will be automatically placed in RUN mode when PLC is powered.

5.7. Offline program transfer to the PLC

When your Raspberry Pi does not have a working network connection, you can transfer your program manually to the PLC.

To do so, you must write your program using the easyLadder studio software, including comments and parameters. When done, you need to export the project using **Export PLC files** option in the File menu. This option will generate four files in your selected directory:

program.bin	This file contains your compiled program.
sections.bin	This file contains information about section organization and network comments. This file is optional.
comments.bin	This file includes device comments for the program. This file is optional.
settings.bin	This file contains PLC configuration parameters.

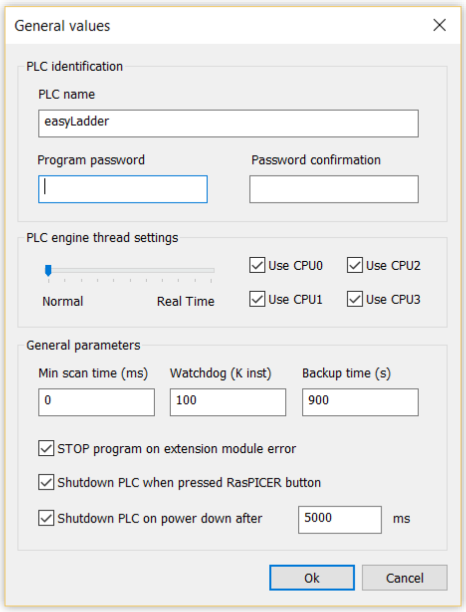
After that, you must close the easyLadder PLC engine in your Raspberry and copy these files using your favorite method. The target for the files is the **plcdata** directory (by default **/opt/raspicer/plcdata**). Once done, you can restart the PLC engine and your program will be up and running.

5.8. PLC configuration

In the project view, bellow program sections, you can access several PLC configuration parameters. These configurations are transferred to the PLC or from the PLC using Write to PLC or Read from PLC icons. You can transfer only the configuration or together with the program. It is not required to STOP the PLC to transfer parameter area.

Configuration parameters are divided into three groups: PLC parameters, Raspberry GPIO definitions, Ethernet modules, I2C I/O expanders and TCP/IP security. You can setup any group by double clicking the item in the project view.

5.8.1. PLC parameters



PLC identification

PLC name: easyLadder

Program password:

Password confirmation:

PLC engine thread settings

☑ Use CPU0 ☑ Use CPU2
☑ Use CPU1 ☑ Use CPU3

General parameters

Min scan time (ms)	Watchdog (K inst)	Backup time (s)
0	100	900

☑ STOP program on extension module error

☑ Shutdown PLC when pressed RasPICER button

☑ Shutdown PLC on power down after 5000 ms

This group contain general configuration values for the PLC.

At the top of the configuration dialog you can find the PLC identification section. Using the PLC name field, you can tag your PLC to ease identification in the network. Also it is possible to protect the program with a password to prohibit unauthorized users to download your program with easyLadder studio.

The PLC engine thread settings lets you optimize the PLC process. You can set the thread priority for the engine and specify the CPUs the engine will use (for multicored Raspberry Pi2 or Pi3). The most important usage for this CPU assignment is to isolate a single CPU for the PLC engine only. You can configure your Linux system to isolate one CPU during boot. When this is done, the system will not assign processes to this core, so selecting ONLY this core in the PLC parameter dialog, the PLC engine will boost performance being the unique process for the CPU. The PLC engine is single threaded, so you will not obtain more performance by isolating more than one core. For more information about CPU isolation, refer to section **4.4. Optimizing the PLC engine**.

The general parameter section contain the following items:

Min scan time selects the minimum scan time for the PLC program, in milliseconds. When setting this field a value greater than 0, the PLC engine will try to adjust the cycle time to this value. For example, if your program cycle without **min scan time** setting is 2 ms, and you set this time to 10 ms, the PLC will execute the cycle and then sleep for 8 ms before running a new scan. This time is approximate, specially when you are not using CPU isolation, due to the Linux system thread scheduler. Use this parameter to reduce processor load if your program timing allows it.

Watchdog field sets the value for the cycle watchdog function, in 1000 instructions units. This feature prevents PLC engine hanging due to bugs in your PLC program. When more than Watchdog x 1000 instructions are executed in a single scan cycle, the PLC program switches to STOP to indicate that the program has crashed, probably due to an infinite loop or similar.

Backup time is the time between automatic retentive data backups, in seconds. Retentive data is saved when PLC engine closes, but data might be lost if the Raspberry Pi loses power. For this reason, you can set a time for doing regular retentive device backups.

STOP program on extension module / I2C device error option selects the desired behavior when any Ethernet extension module or I2C device is not reachable.

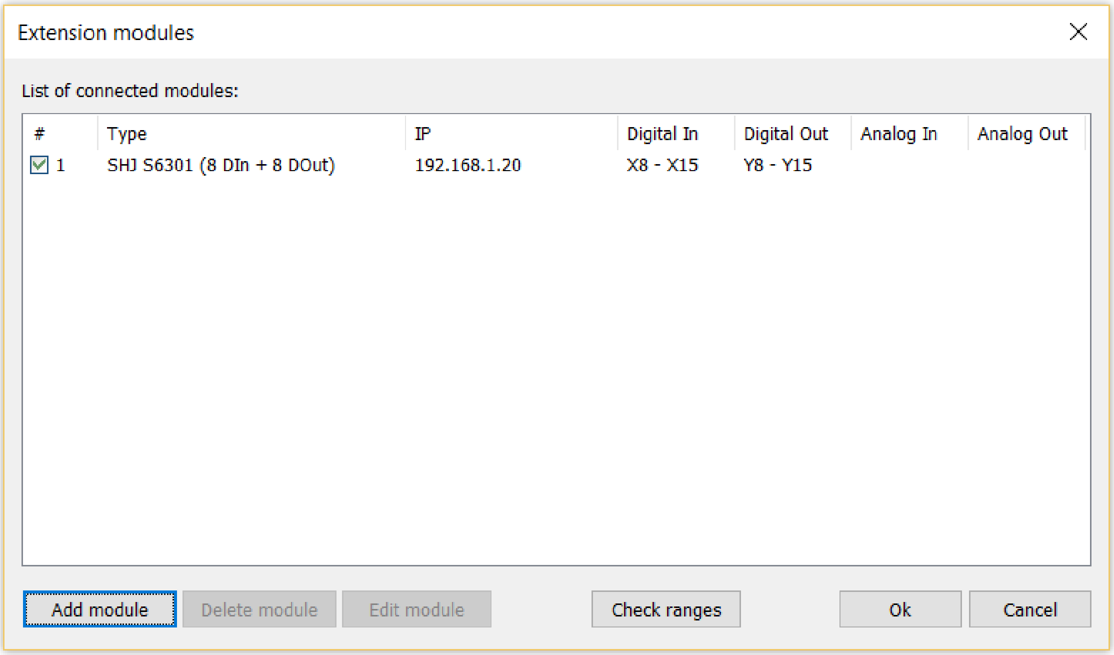
Shutdown PLC when pressed RasPICER button enables the power button feature for the RasPICER board. A **poweroff** command is issued when pressing the power button. This option is only valid when using the RasPICER board.

Shutdown PLC on power down after a configured time, in milliseconds, causes the PLC engine to issue a **poweroff** command when no power is detected on the RasPICER board during the time specified. While this time elapses, the Raspberry Pi is powered through the RasPICER battery. This option is only valid when using the RasPICER board.

5.8.2. Ethernet unit definitions

This configuration group contains information about extension modules connected to the system. Unit management is done through the extension modules dialog. In this dialog you can add, delete or edit modules.

The dialog include a list of connected units with configuration data (module type, Ethernet address and device allocation).

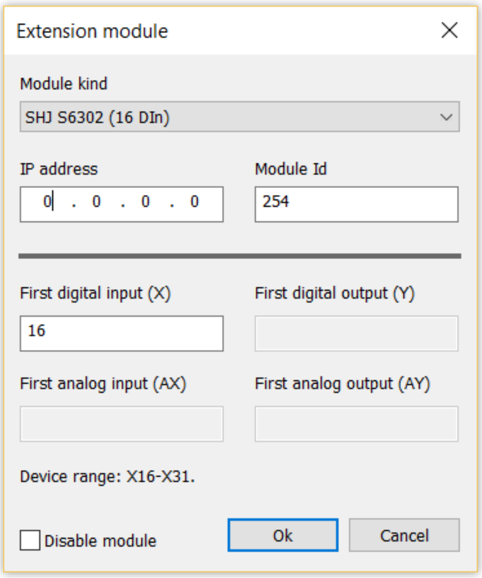


#	Type	IP	Digital In	Digital Out	Analog In	Analog Out
☑ 1	SHJ S6301 (8 DIn + 8 DOut)	192.168.1.20	X8 - X15	Y8 - Y15		

When adding a new module, you must specify the module kind, IP address, Module Id and PLC devices allocated to the unit. You can also disable the module for maintenance purposes.

Modules must be properly configured before use on easyLadder. Configuration requires the **UDP server** mode, responding to UDP port 502 (MODBUS). Additionally you must configure IP parameters (IP address, Subnet mask and Gateway) according to your network.

Care must be taken when configuring PLC device ranges for the module, because devices must not overlap between modules, RasPICER IOs, GPIO and I2C definitions. For this reason, a **check ranges** button is present in the extension modules dialog. This button checks all device allocation to prevent device overlapping.



Extension module

Module kind: SHJ S6302 (16 DIn)

IP address: 0 . 0 . 0 . 0

Module Id: 254

First digital input (X): 16

First digital output (Y):

First analog input (AX):

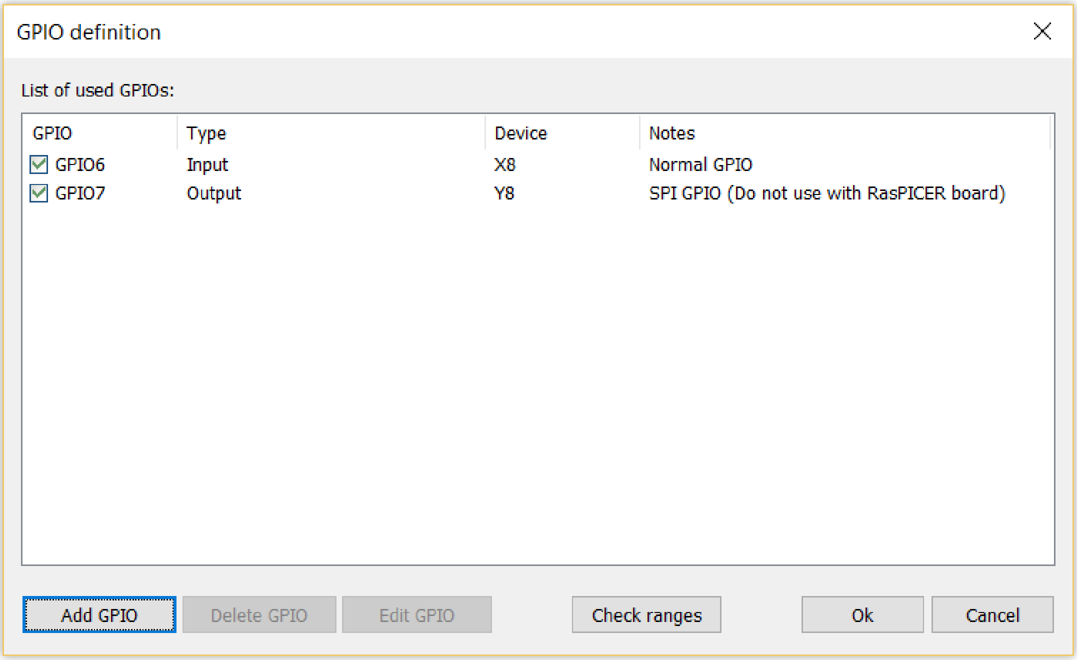
First analog output (AY):

Device range: X16-X31.

5.8.3. GPIO definitions

Raspberry Pi provides a number or input and output pins in the GPIO connector. These GPIO ports can be used to expand your available digital IOs for the easyLadder PLC.

This configuration group lets you configure assigned X or Y devices to GPIO pins. You can view, add, delete or modify GPIO definitions using the GPIO definition dialog.



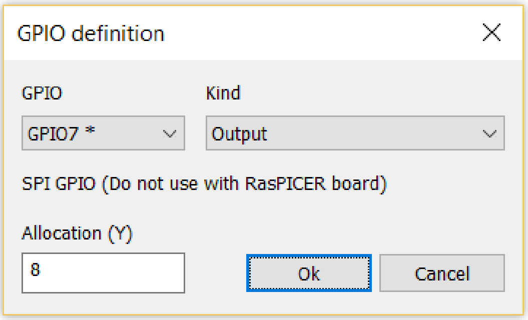
GPIO definition

List of used GPIOs:

GPIO	Type	Device	Notes
✓ GPIO6	Input	X8	Normal GPIO
✓ GPIO7	Output	Y8	SPI GPIO (Do not use with RasPICER board)

When adding a new GPIO, you must select the desired GPIO number. You can get more information about available GPIOs in **section 3.6. RasPICER, GPIO and I2C expanders I/O allocation**. Please note that when using the RasPICER board some GPIOs (SPI related) are not available. Do not use them.

Each GPIO can be configured as an **output**, **input**, **input with pullup** or **input with pulldown**. Care must be taken when configuring assigned device for the GPIO, because devices must not overlap between GPIOs, RasPICER IOs, I2C expanders and extension units. For this reason, a **check ranges** button is present in the GPIO definition dialog. This button checks all device allocation to prevent device overlapping.


GPIO definition

GPIO: GPIO7 *

Kind: Output

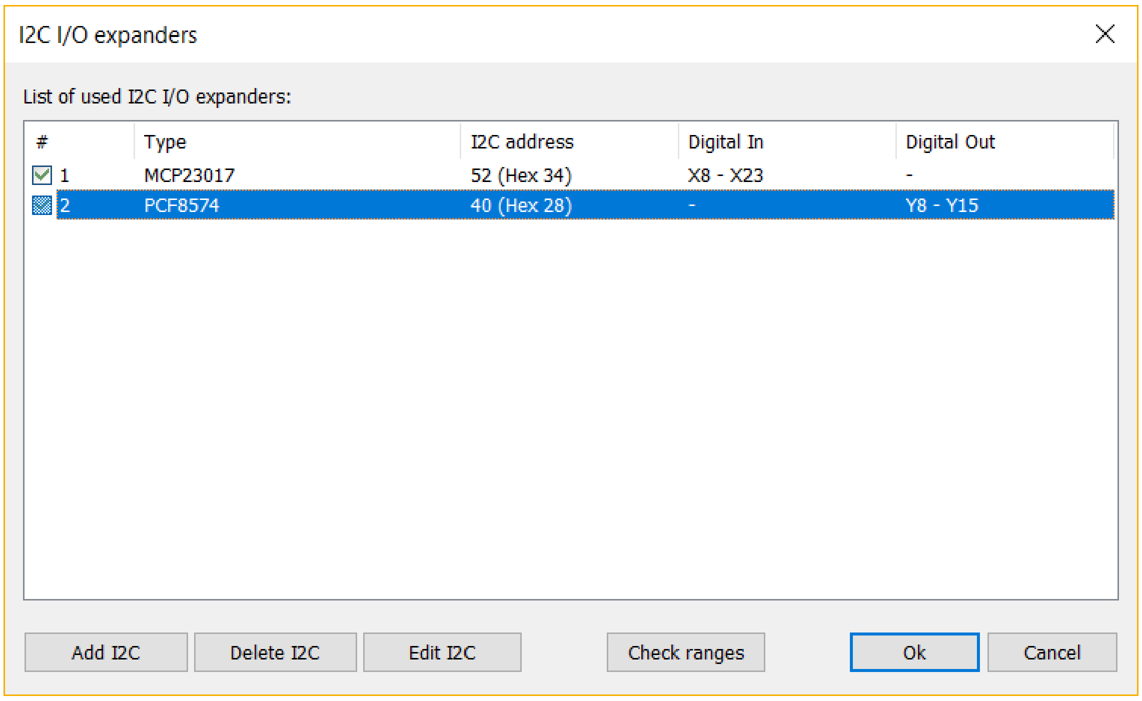
SPI GPIO (Do not use with RasPICER board)

Allocation (Y): 8

5.8.4. I2C I/O expanders

Raspberry Pi includes an I2C serial port in the GPIO header. Using this port you can provide additional low cost inputs and outputs to your system, by simply connecting compatible I2C port expanders to the Raspberry Pi I2C serial bus.

This configuration group lets you configure assigned X or Y devices to I2C expanders. You can view, add, delete or modify I2C modules using the I2C I/O expanders dialog.


I2C I/O expanders

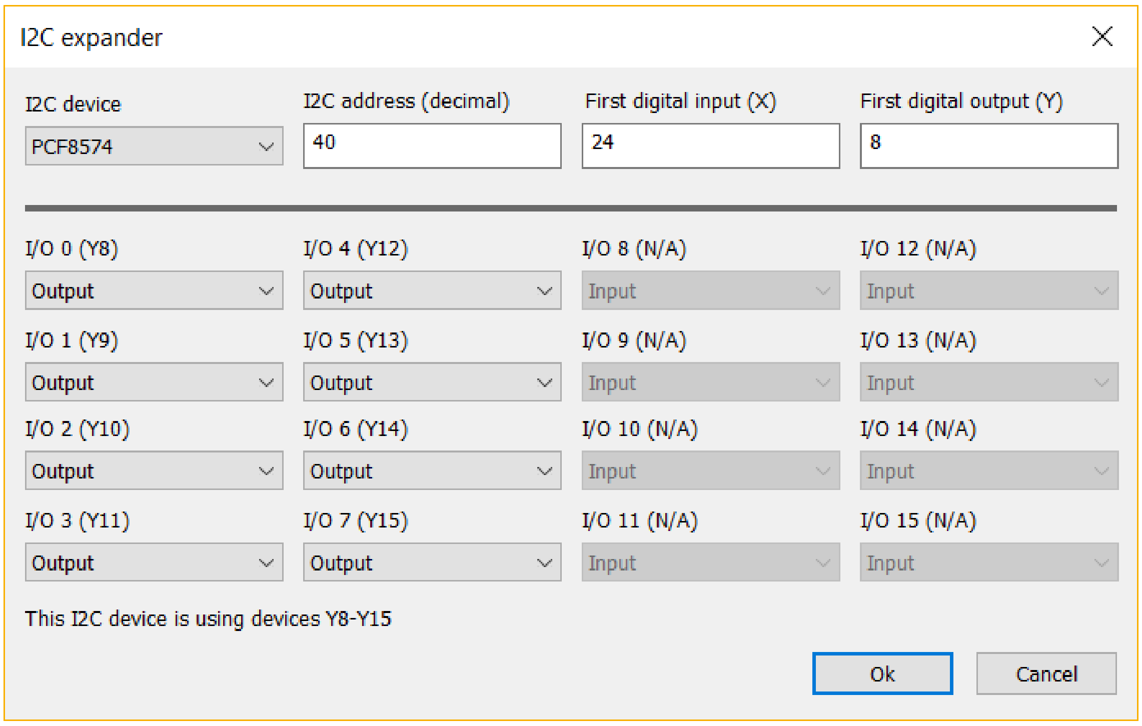
List of used I2C I/O expanders:

#	Type	I2C address	Digital In	Digital Out
✓ 1	MCP23017	52 (Hex 34)	X8 - X23	-
▨ 2	PCF8574	40 (Hex 28)	-	Y8 - Y15

When adding a new I2C device, you must select your connected device model and the I2C address. Compatible I2C devices are shown as a drop box when adding an expander in the easyLadder studio software. Please consult this list before designing your system. I2C device address is designated without the less significant bit, which indicates the read or write operation (R/W). Please note that when using the I2C bus, some GPIOs (I2C related) are not available. Do not use them.

Each I2C I/O can be configured as an **output**, **input** or **input with pullup**. For each expander, X and Y devices are sequentially allocated from first digital input (X) and first digital output (Y) fields.

For your convenience, you can view assigned devices for each I/O in the I/O caption.



I2C device	I2C address (decimal)	First digital input (X)	First digital output (Y)
PCF8574	40	24	8

I/O 0 (Y8): Output	I/O 4 (Y12): Output	I/O 8 (N/A): Input	I/O 12 (N/A): Input
I/O 1 (Y9): Output	I/O 5 (Y13): Output	I/O 9 (N/A): Input	I/O 13 (N/A): Input
I/O 2 (Y10): Output	I/O 6 (Y14): Output	I/O 10 (N/A): Input	I/O 14 (N/A): Input
I/O 3 (Y11): Output	I/O 7 (Y15): Output	I/O 11 (N/A): Input	I/O 15 (N/A): Input

This I2C device is using devices Y8-Y15

Care must be taken when configuring assigned devices for the I2C expander, because devices must not overlap between GPIOs, RasPICER IOs and extension units. For this reason, a **check ranges** button is present in the I2C definition dialog. This button checks all device allocation to prevent device overlapping.

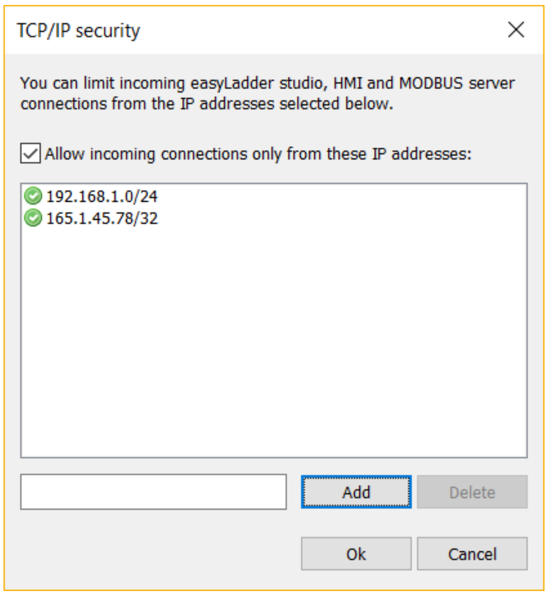
5.8.5. TCP/IP security

Starting from version 1.7, you can restrict remote connections to the PLC (HMI protocol, easyLadder Studio and MODBUS TCP/IP server) to only a number of allowed IP ranges.

With this feature, you can protect the PLC engine from unwanted remote accesses.

Using this configuration option, you can set rules for remote accesses to the PLC.

The checkbox provided lets you to enable the access rules to the PLC. When unchecked, all accesses are allowed. When checked, remote access will be valid only when remote IP address is included in the list.



You can limit incoming easyLadder studio, HMI and MODBUS server connections from the IP addresses selected below.

[✓] Allow incoming connections only from these IP addresses:

- ✓ 192.168.1.0/24
- ✓ 165.1.45.78/32

Addresses are entered using the CIDR notation (X.X.X.X/N), so it is possible to specify an address with a network mask. For example, when setting an address of 192.168.1.0/24, addresses in the range 192.168.1.1 to 192.168.1.254 are allowed to access the engine. When setting 192.168.1.45/32, only access from 192.168.1.45 is allowed. Please note that 127.0.0.1 (localhost) access is always permitted.

When access to the PLC engine is lost, due to an error in this setting or other problems, you can disable access rules by running the RasPICER utility in the Raspberry Pi with a special parameter: **raspicer --unblocktcpip**. This way you will gain access to the PLC without losing configurations.

6. HMI AND EASYLADDER PLC

When building the easyLadder PLC system, you probably will need some kind of Human Machine Interface (HMI) to monitor PLC status, operate manual PLC commands, configure working parameters or manage process data (logging, graphs and so on). For this purpose, you can use any available industrial HMI providing a MODBUS TCP driver. Nevertheless, when creating cost-sensitive applications or having special HMI requirements, the best solution is to develop your own HMI application using a powerful language like C++, executed on the same Raspberry Pi. Doing so, you get an embedded HMI/PLC system with unlimited possibilities.

When creating your HMI application, you can use the general Programming library (**section 7. Programming library**) or use our ready to use **easyLadderHMI** library.

easyLadderHMI is a free library used to build your own HMI application to interface with the easyLadder PLC. Using this library you can easily develop the HMI interface for your PLC program using the power of the Qt platform and C++ language, without worrying about knowledge of easyLadder PLC communication internals. A complete sample is provided with the library.

The sample provided is designed for the official Raspberry Pi 7" touch screen, but can be easily translated to other LCD. In this case, it connects to the local easyLadder engine (127.0.0.1), but it can also connect to a remote PLC simply changing this IP in the source code.

Using the Qt platform, this library can be compiled in any Linux platform, or even in Windows machines with small modifications. So it is possible to get your HMI working remotely in any desktop PC, for example.

For more information and library download, refer to http://www.ferrariehijos.com/easyLadder.

7. PROGRAMMING LIBRARY

The easyLadder software package contains libraries and source code to ease your custom development project. **C** and **Python** languages are supported. This library is shared with the RasPICER board support library. Nevertheless, the RasPICER board is not needed to interface with the easyLadder PLC.

With these libraries, you can monitor the easyLadder engine through the PLC *daemon* using UNIX domain sockets.

Writing your customized program to interact with the PLC gives an additional power to the PLC engine. For instance, you can use your program to do advanced calculations, custom logging, remote messaging, or provide a user interface to your PLC (HMI or Web).

When using **C** language, you will need to add the **raspicer.c** and **easyladder.c** source file to your project and include **raspicer.h** and **easyladder.h** headers. The only requirement is to call **raspicerInit ()** function before using the library.

When using **Python**, you will find the source code for the Python module, written in C language. To build and install this module for the current user follow these instructions:

```
cd /opt/effesoftware/python
python setup.py install --user
```

In order to build and install this module for all users follow these instructions:

```
cd /opt/effesoftware/python
sudo python setup.py install
```

To use the raspicer/easyLadder module, simply **import raspicer** in your Python program.

Refer to http://www.ferrariehijos.com/easyLadder for more details about available functions.

ferrari e hijos, s.a.

http://www.ferrariehijos.com/easyLadder

info@ferrariehijos.com