UNIVERSITY OF LJUBLJANA
FS
Faculty of Mechanical Engineering

Tine Jereb

# Closed-loop stepper motor control with Eaton XC204 PLC

Idrija, 2025

# Contents

1 INTRODUCTION ........ 3
2 Stepper motors ........ 4
3 Drivers ........ 4
3.1 Motion studio ........ 5
4 Controller ........ 5
4.1 Codesys V3 ........ 5
4.2 Modbus RTU ........ 6
5 Connections ........ 6
5.1 Communication level ........ 6
5.2 Driver level ........ 7
6 Setting up the drivers ........ 7
6.1 DIP Switches ........ 7
6.2 Parameters with motion studio ........ 8
6.2.1 Homing method ........ 8
6.2.2 Driver I/O terminal ........ 8
7 Programming the PLC ........ 9
7.1 Configuring PLC ........ 9
7.2 Setting up MODBUS communication ........ 10
7.2.1 Reading and writing parameters ........ 12
7.2.2 Storing parameters ........ 14
7.3 Programming function block for motor movement ........ 15
7.3.1 Use example ........ 18
7.4 Setting up web visualisation ........ 19
7.5 State machine programming ........ 21
8 Conclusion ........ 23

# Table of figures

Figure 1: Stepper motor P Series NEMA closed loop ........ 4
Figure 2: CL57RS stepper driver ........ 4
Figure 3: Eaton XC204 PLC ........ 5
Figure 4: System communication diagram ........ 6
Figure 5: Driver connections diagram ........ 7
Figure 6: DIP switch, address ........ 7
Figure 7:Homing setup ........ 8
Figure 8: I/O settings ........ 8
Figure 9: New project window ........ 9
Figure 10: Device and programing language setup ........ 9
Figure 11: PLC to PC connection ........ 10
Figure 12: Adding a COM port for Modbus RTU communication ........ 11
Figure 13: Adding slaves ........ 11
Figure 14: Slave address ........ 12
Figure 15: Values and addresses on the motor drive ........ 12
Figure 16: Modbus channel setup ........ 13
Figure 17: Channels ........ 13
Figure 18: Global variable list ........ 14
Figure 19: Defining global variables ........ 14
Figure 20: Rewriting Modbus variables to variables used in program ........ 15
Figure 21: Mapping variables to channels ........ 15
Figure 22: Input/Output variables ........ 16
Figure 23: Selecting right motor ........ 16
Figure 24: algorithm for resending the command ........ 17
Figure 25: output variables at the end of the movement ........ 17
Figure 26: Mapping variables to function block ........ 18
Figure 27: Setting variables of function block ........ 18
Figure 28: single variable access ........ 18
Figure 29: Creating visualisation ........ 19
Figure 30: Toolbox ........ 19
Figure 31: Element properties ........ 20
Figure 32: Visualization ........ 20
Figure 33: Homing ........ 21
Figure 34: state machine code ........ 22

# 1 INTRODUCTION

The project is based on controlling the drives, PLC and programing the logic of a machine used for automatically storing and supplying tools to production lines. Due to the policies of the company that is financing the project I can't include any details or photos of the complete machine. So, in this paper I will concentrate on the theoretical and practical aspects of setting up stepper motor drives, configuring the communication protocol between the PLC and drivers and executing the control code used on the machine.

# 2 Stepper motors

The system uses StepperOnline **P Series NEMA closed-loop stepper motors** with integrated **1000 PPR (4000 CPR)** incremental encoders. These motors were chosen in different sizes (Nema standard) based on torque requirements on different axis of the machine. Their affordable price, simple setup, and reliable closed-loop performance made them an alternative to servo motors, offering good accuracy and automatic position correction without the complexity or cost of full servo systems.

*Figure 1: Stepper motor P Series NEMA closed loop*

# 3 Drivers

The selected drivers were CL57RS closed-loop stepper drivers, which are compatible with NEMA 23 motors and support Modbus RTU (RS-485) communication. They provide smooth and precise motor control, automatic error correction based on encoder feedback, 7 different homing methods and built-in protection features such as over-voltage, over-current, and position deviation alarms. The drivers support a wide input voltage range (24–50V DC) and allow microstepping settings up to 256 subdivisions, making them suitable for a variety of motion control tasks while remaining cost-effective and easy to integrate with a PLC.

*Figure 2: CL57RS stepper driver*

## 3.1 Motion studio

Motion studio is the software that is used for configuring and testing the set parameters of the driver. It was used to set the running direction, jog speed, inputs and outputs of the driver and homing methods.

# 4 Controller

The controller used was the Eaton XC-204, a compact and modular PLC designed for automation tasks. It features a 400 MHz RISC processor and supports programming according to the IEC 61131-3 standard via CODESYS V3. The base unit comes with integrated communication interfaces, including Ethernet, RS-232, and RS-485, making it good for Modbus RTU or TCP-based control systems.

The PLC includes built-in digital I/O and supports modular expansion, allowing the addition of various I/O and communication modules to meet specific application requirements. It also offers real-time processing and a web-based diagnostic interface. Thanks to its flexibility and industrial-grade reliability, the XC-204 is well-suited for controlling motion systems such as stepper motors in automation environments.

*Figure 3: Eaton XC204 PLC*

## 4.1 Codesys V3

For programming I used Codesys V3 software. The PLC supports Ladder and Structured text programming, but for this task I used only ST language because the program was quite big, and it was more manageable that way. It also makes it easier to implement Modbus communication. Software and PLC also support web visualisation that is easily implemented and used for controlling the machine. For debugging PLC also has online mode so you can se the variable values in real time.

## 4.2 Modbus RTU

In this project, communication with multiple stepper motor drivers is achieved using the Modbus RTU protocol over a shared serial COM port. Each driver is assigned a unique Modbus slave ID. Control and monitoring are performed by reading from and writing to specific register offsets corresponding to driver functions — such as position, speed, acceleration, or status flags. The PLC (as Modbus master) sends function code commands (e.g., 03 for reading, 06 or 16 for writing) to these offsets, allowing control of motion parameters for each motor individually over a single RS-485 bus. Codesys uses a library that allows for simple implementation of these commands without the need to manually compose messages inside the code.

# 5 Connections

## 5.1 Communication level

Diagram in figure Figure 4 shows the communication level configuration of the system. At the top level sits the PC that is used to send commands from web visualisation HMI over ethernet to the PLC. Bellow the PLC is a Modbus master slave configuratiom between the CL57RS drivers and PLC.

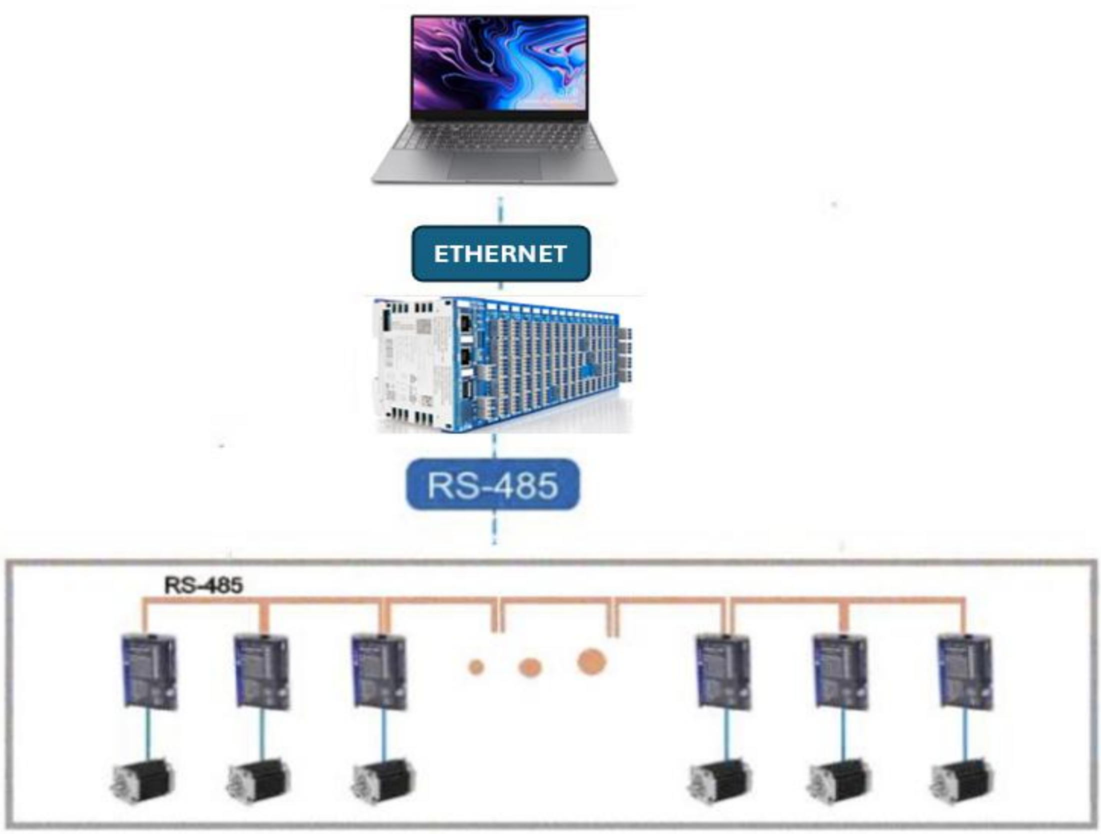



*Figure 4: System communication diagram*

## 5.2 Driver level

Figure 4 displays the connections of all the used components to the driver.

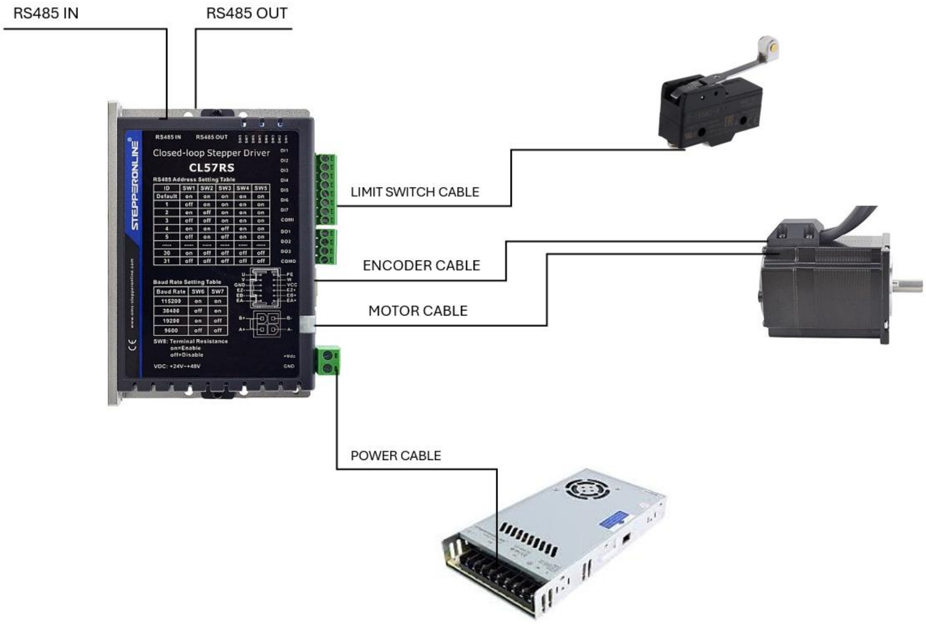
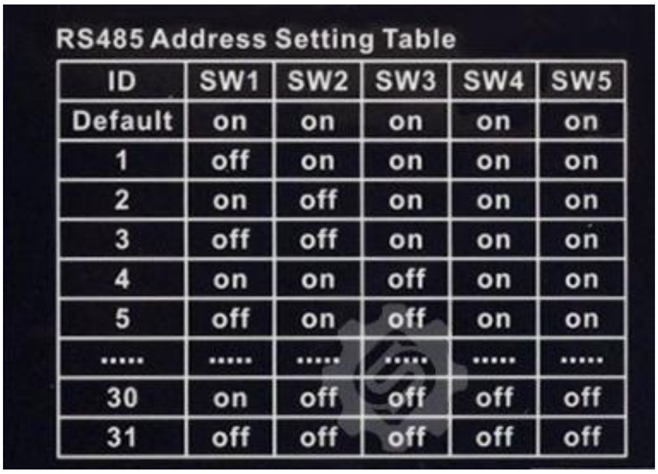



*Figure 5: Driver connections diagram*

# 6 Setting up the drivers

## 6.1 DIP Switches

First five DIP switches are used for setting the slave address of each drive according to the table displayed on the front panel of the driver. This will be important later when configuring Modbus RTU between PLC and drivers



RS485 Address Setting Table

| ID | SW1 | SW2 | SW3 | SW4 | SW5 |
|---|---|---|---|---|---|
| Default | on | on | on | on | on |
| 1 | off | on | on | on | on |
| 2 | on | off | on | on | on |
| 3 | off | off | on | on | on |
| 4 | on | on | off | on | on |
| 5 | off | on | off | on | on |
| ...... | ...... | ...... | ...... | ...... | ...... |
| 30 | on | off | off | off | off |
| 31 | off | off | off | off | off |

*Figure 6: DIP switch, address*

DIP switch 6 – 7 are used to set the baud rate of communication and the last switch 8 is terminal resistance that needs to be turned on on the last driver in the communication chain.

## 6.2 Parameters with motion studio

### 6.2.1 Homing method

Steppers use incremental encoders so after powering up the machine referencing of the motors is required. Referencing is done using the limit switch homing method. With firstly high-speed approach to the limit switch (used to cut the time used for referencing if the axis is far away from limit switch) and then axis moves back few millimetres and approaches the limit switch again with lower speed for better accuracy of homing. The limit switch is connected directly to the driver input terminal.

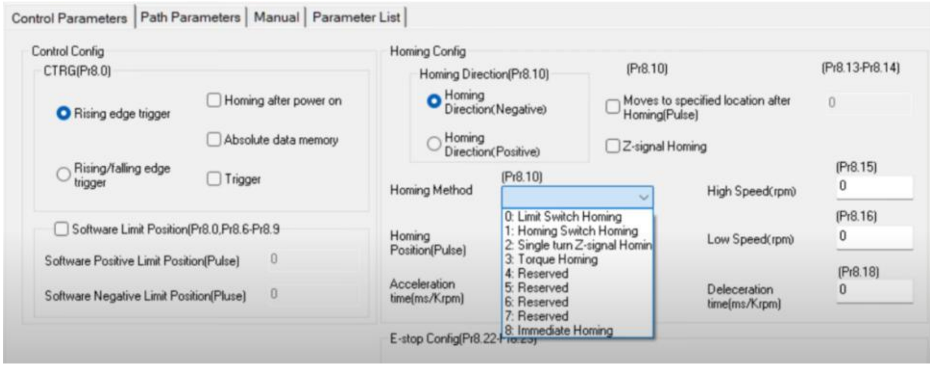
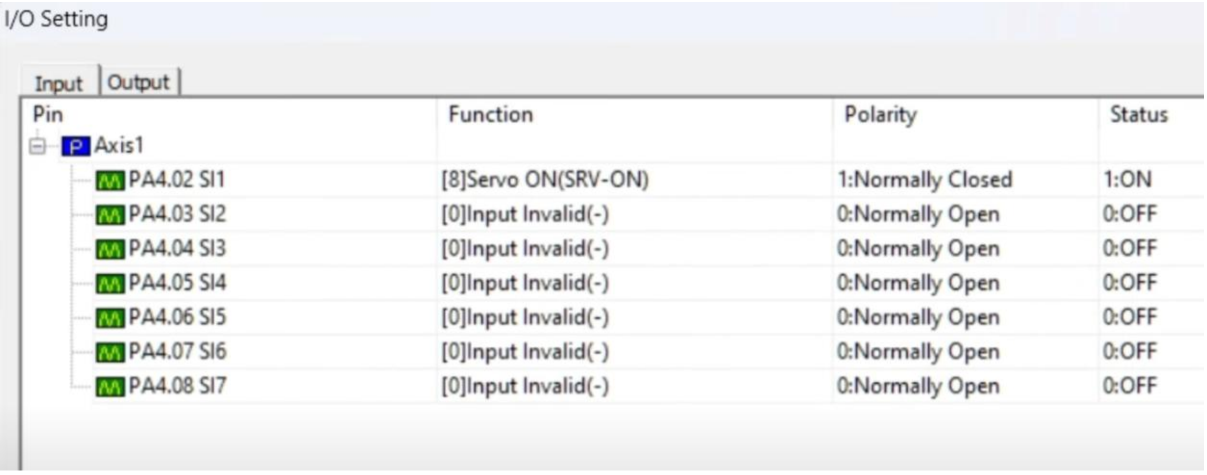



*Figure 7:Homing setup*

### 6.2.2 Driver I/O terminal

Inside the I/O settings I set up to which pin the limit switch is connected.



I/O Setting

Input | Output

| Pin | Function | Polarity | Status |
|---|---|---|---|
| Axis1 | | | |
| PA4.02 SI1 | [8]Servo ON(SRV-ON) | 1:Normally Closed | 1:ON |
| PA4.03 SI2 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |
| PA4.04 SI3 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |
| PA4.05 SI4 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |
| PA4.06 SI5 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |
| PA4.07 SI6 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |
| PA4.08 SI7 | [0]Input Invalid(-) | 0:Normally Open | 0:OFF |

*Figure 8: I/O settings*

# 7 Programming the PLC

## 7.1 Configuring PLC

Firstly, I opened the codesys V3 software and created and by clicking on ‘New Project’ created a new project with standard template as shown in Figure 9.

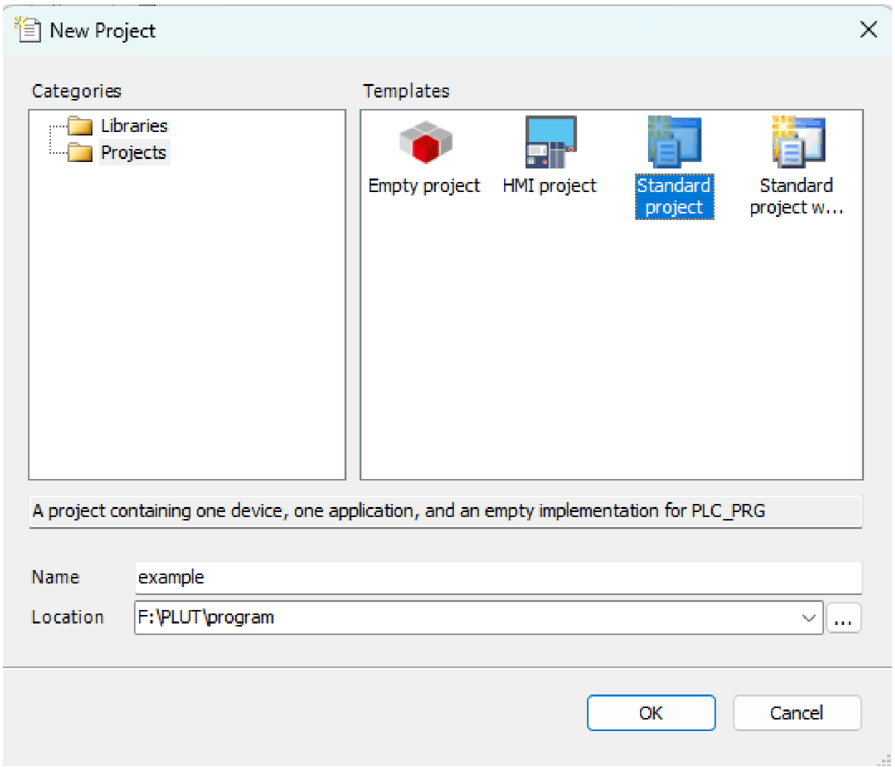



*Figure 9: New project window*

After that I selected the Eaton XC204 device and selected the programming language to be Structured Text (ST) as shown in **Error! Reference source not found.**.

*Figure 10: Device and programing language setup*

Finally, I set up the Ethernet route between the PC and PLC. I connected the PLC to a router so it can be reached from anywhere inside the local network using the correct IP address as shown in Figure 11.

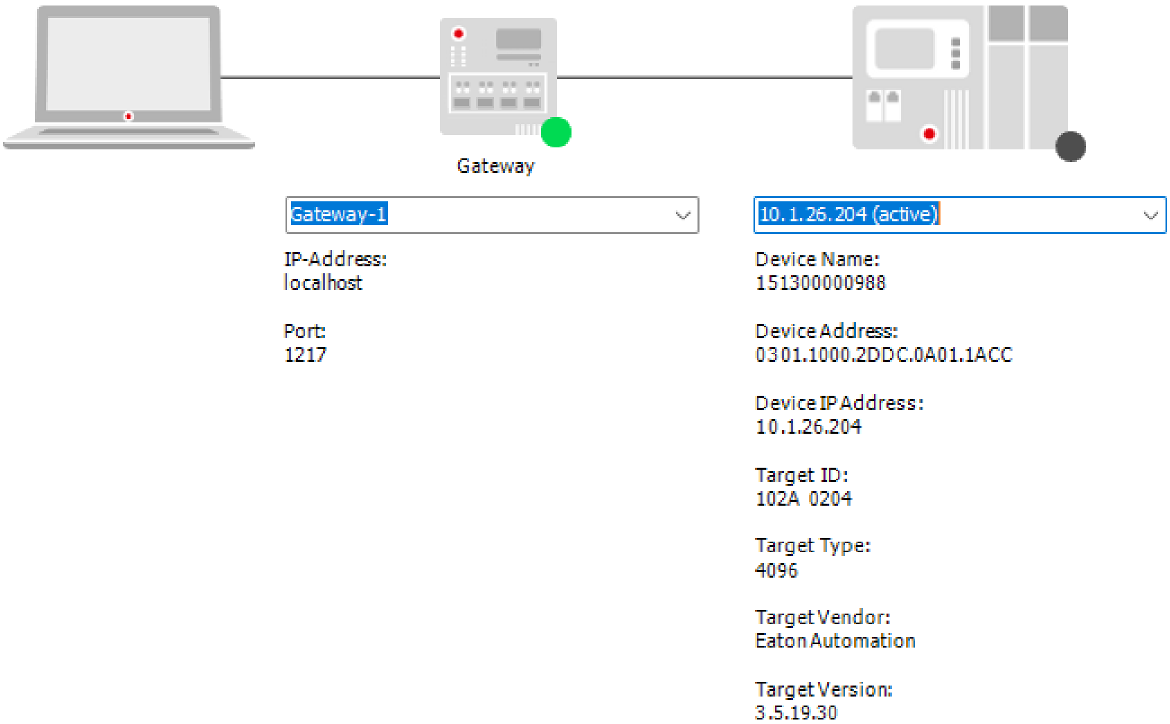



*Figure 11: PLC to PC connection*

## 7.2 Setting up MODBUS communication

As can be seen in Figure 4 the Modbus RTU communication is set so the PLC is the master, and the six motor drivers used are slaves.

Codesys features a library that can be used to simply add slaves to the network and set up the parameters that will be sent and received over the RS485 cable to the selected drives and back to PLC. To set up the COM port of the PLC used for Modbus communication I right clicked on ‘Device (XC204)’ in the tree structure window and then clicked on ‘Add device’ inside the add device window I selected the Modbus COM option.

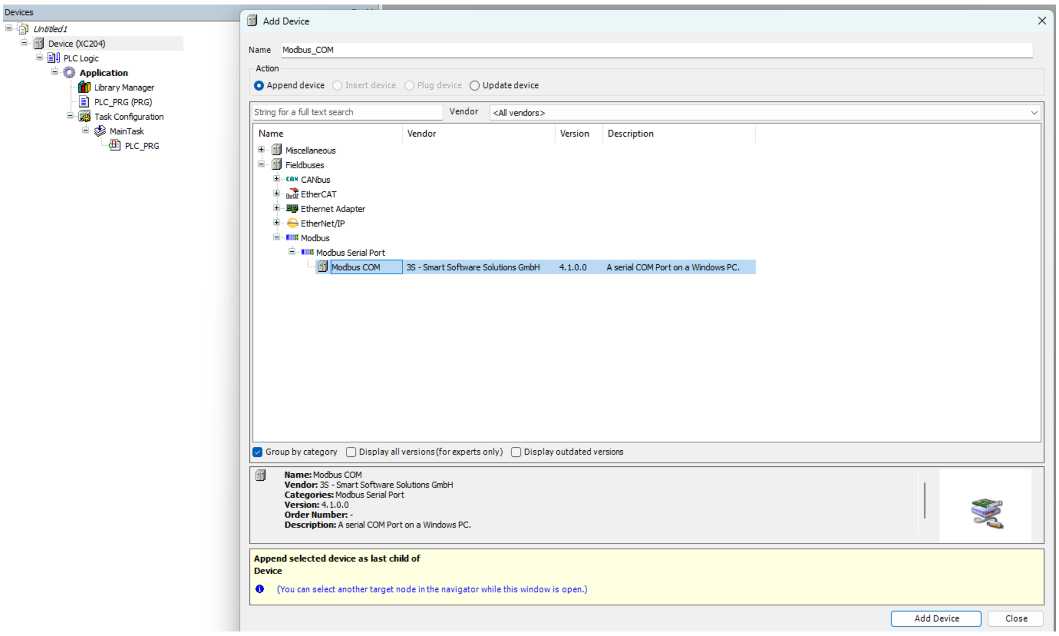



*Figure 12: Adding a COM port for Modbus RTU communication*

After that following the same procedure as above, I added a master and slaves for all six motors used. Process can be seen in Figure 13.

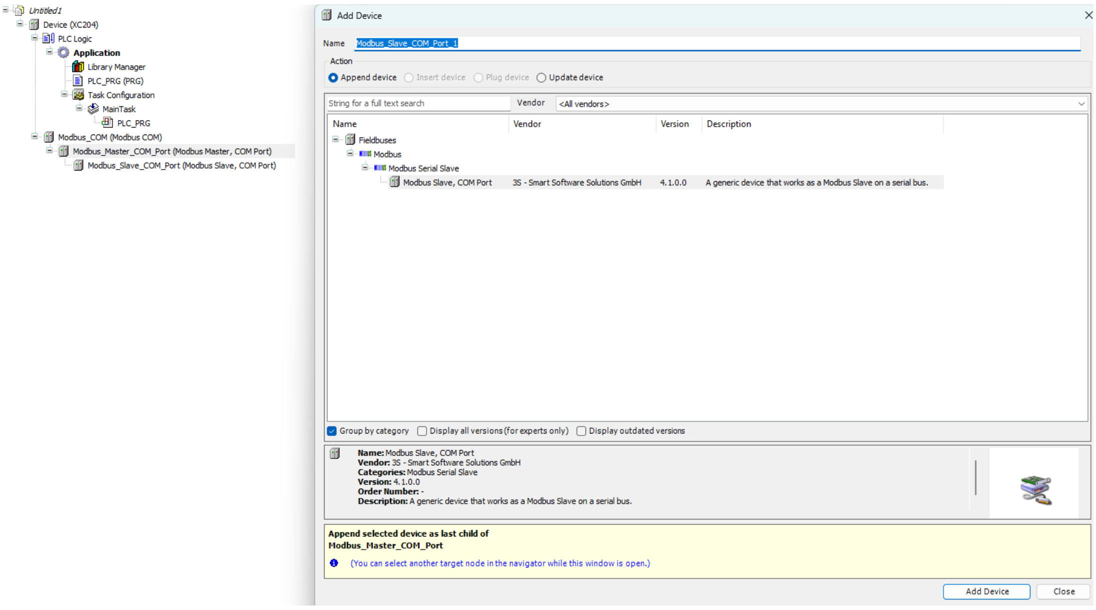



*Figure 13: Adding slaves*

Here it is also important to correctly set the address of the slaves, so it matches the addresses set on the driver using switches shown in section 6.1. The slave address is set inside the general menu of the selected slave device, Figure 14: Slave address.

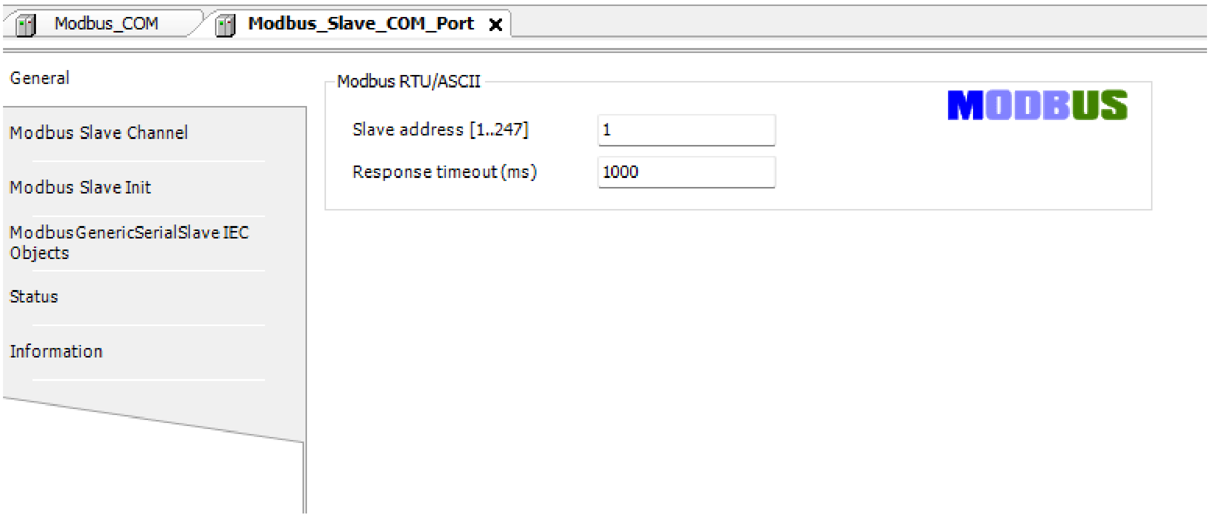



*Figure 14: Slave address*

### 7.2.1 Reading and writing parameters

The values that need to be read are set inside the Modbus Slave Channel menu. Firstly, I defined which values need to be read and sent and at what address in the CL57RS drive memory are these values stored. I got this information from the CL57RS user manual. Section of the parameters and addresses from the CL57RS user manual can be seen in Figure 15.

**5.4.1 PR Parameters**

Usually it is recommended using the PTP window of the STEPPERONLINE tuning software to configure the PR path parameters, but it can also use the following objects:

| Par. # in software | Register Address | Definition | Description |
|---|---|---|---|
| Pr9.00 | 0x6200 | PR path 0 | The corresponding functions can be selected for different bit<br>Bit0-3: Operation mode<br>=0---- no action<br>=1---- position mode<br>=2---- velocity mode<br>=3---- homing mode;<br>Bit4: INS,<br>=0---- No interrupt<br>=1---- interrupt(all the current ones are 1.);<br>Bit5: OVLP,<br>=0---- Non overlapping<br>=1---- Overlapping<br>Bit6:<br>=0----absolute position<br>=1----relative position<br>Bit8-13: Jump to the corresponding PR path 0-15;<br>bit14: JUMP,<br>=0---- No jump<br>=1---- jump |
| Pr9.01 | 0x6201 | Position H | High 16 bit, |
| Pr9.02 | 0x6202 | Position L | Low 16 bit |
| Pr9.03 | 0x6203 | velocity | Unit: rpm |
| Pr9.04 | 0x6204 | Acc | Unit: ms/1000rpm |
| Pr9.05 | 0x6205 | Dec | Unit: ms/1000rpm |
| Pr9.06 | 0x6206 | Pause time | Pause time after the command is stopped |
| Pr9.07 | 0x6207 | Special parameter | PR Path 0 maps directly to Pr8.02, Others are reserved |
| Pr9.08 | 0x6208 | PR path 1 | --- |
| Pr9.09 | 0x6209 | Position | --- |
| Pr9.10 | 0x620A | Position | --- |
| Pr9.11 | 0x620B | velocity | --- |
| Pr9.12 | 0x620C | Acc | --- |
| Pr9.13 | 0x620D | Dec | --- |
| Pr9.14 | 0x620E | Pause time | --- |
| Pr9.15 | 0x620F | Special parameter | --- |
| Pr9.16 | 0x6210 | PR path 2 | --- |
| Pr9.17 | 0x6211 | Position | --- |
| Pr9.18 | 0x6212 | Position | --- |
| Pr9.19 | 0x6213 | velocity | --- |
| Pr9.20 | 0x6214 | Acc | --- |
| Pr9.21 | 0x6215 | Dec | --- |
| Pr9.22 | 0x6216 | Pause time | --- |

*Figure 15: Values and addresses on the motor drive*

After defining the values and addresses I set them up inside the program using the 'add channel' function.

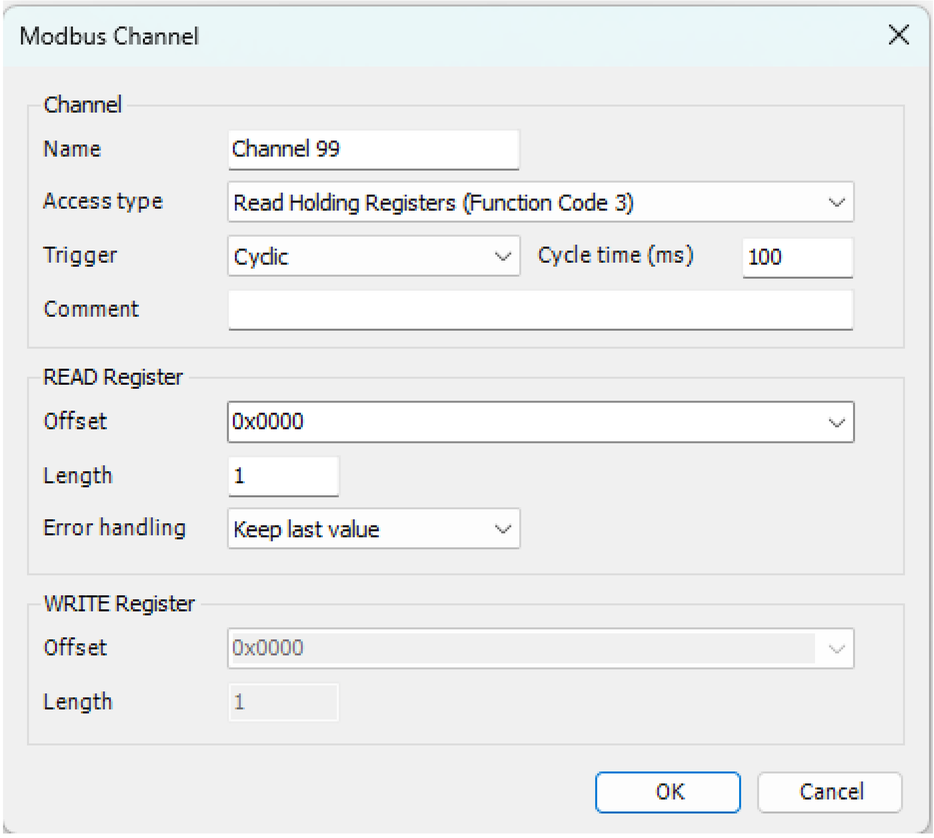


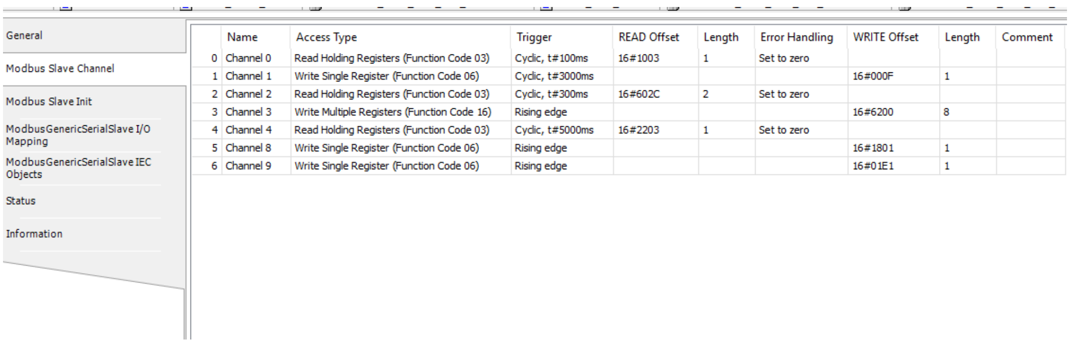

*Figure 16: Modbus channel setup*

Inside the Modbus channel set for each parameter, I wanted to read or write I configured a trigger which can be:

- Cylic – reads or writes in set time intervals (used for reading statuses and position of the motor)
- Rising edge – reads or writes when a rising edge is detected on later set trigger value (used for sending position, jog, enable/disable commands)

Then I set the offset value of the parameter according to the addresses found in the user manual. One channel allows you to read multiple parameters at once by setting the length value this reads or writes form offset value forward as many as specified with length and stores them in an array.

Figure 17 shows the setup of the channels for a single slave. I repeated this process for all the slaves.




| | Name | Access Type | Trigger | READ Offset | Length | Error Handling | WRITE Offset | Length | Comment |
|---|---|---|---|---|---|---|---|---|---|
| 0 | Channel 0 | Read Holding Registers (Function Code 03) | Cyclic, t#100ms | 16#1003 | 1 | Set to zero | | | |
| 1 | Channel 1 | Write Single Register (Function Code 06) | Cyclic, t#3000ms | | | | 16#000F | 1 | |
| 2 | Channel 2 | Read Holding Registers (Function Code 03) | Cyclic, t#300ms | 16#602C | 2 | Set to zero | | | |
| 3 | Channel 3 | Write Multiple Registers (Function Code 16) | Rising edge | | | | 16#6200 | 8 | |
| 4 | Channel 4 | Read Holding Registers (Function Code 03) | Cyclic, t#5000ms | 16#2203 | 1 | Set to zero | | | |
| 5 | Channel 8 | Write Single Register (Function Code 06) | Rising edge | | | | 16#1801 | 1 | |
| 6 | Channel 9 | Write Single Register (Function Code 06) | Rising edge | | | | 16#01E1 | 1 | |

*Figure 17: Channels*

### 7.2.2 Storing parameters

After that the values from the channel need to be mapped to variables to be used later in the control program. I did this using the ModbusGenericSerialSlave I/O Mapping menu. But before that I needed to create global variables inside the program. This is done by right clicking the 'Application' object inside the tree structure and under 'Add object 'selecting 'Global variable list'.

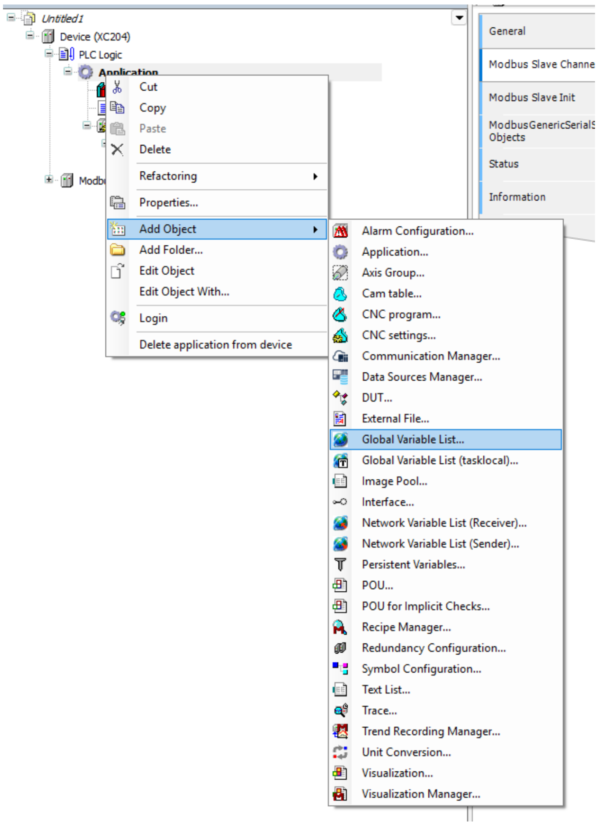



*Figure 18: Global variable list*

This creates a space where all global variables for storing Modbus values are defined. In Figure 19 is an example for how I defined the variables.

```
wModBusRead_MotorZ_Status: WORD;
waModBusRead_MotorZ_ActualPosition   :ARRAY[0..1] OF WORD;
wModBusWrite_MotorZ_Enable: WORD;
waModBusWrite_MotorZ_MotionParameters   :ARRAY[0..7]OF WORD;
xModBusWrite_MotorZ_Triger:BOOL;
wModBusRead_MotorZ_Error: WORD;
wModBusWrite_MotorZ_ResetCurrentAlarm:WORD;
wModBusWrite_MotorZ_JogCommand:WORD;
xModBusWrite_MotorZ_JogTrigger:BOOL;
wModBusWrite_MotorZ_JogVelocity:WORD;
```

*Figure 19: Defining global variables*

As seen in Figure 15 position needs to be sent and received in high and low byte format to two registers which allows the sending of bigger numbers than only 16bit. This is implementation can be also seen in third and fourth line of Figure 20.

```
//MotionParameters and MotionCommands
    waMotorZ_MotionParameters[0] := uiMotorZ_OperationMode;     //OperationMode: 1= position mode, 3=Homing mode
    waMotorZ_MotionParameters[1] := UDINT_TO_WORD(udiMotorZ_NewPosition/65535);                            //Position_H
    waMotorZ_MotionParameters[2] := UDINT_TO_WORD(udiMotorZ_NewPosition-(65535*waMotorZ_MotionParameters[1]));  //Position_L
    waMotorZ_MotionParameters[3] := uiMotorZ_Velocity;  //Velocity_RPM
    waMotorZ_MotionParameters[4] := uiMotorZ_Acc;       //Acc ms/1000RPM
    waMotorZ_MotionParameters[5] := uiMotorZ_Dec;       //Dec ms/1000RPM
    waMotorZ_MotionParameters[6] := 0;                  //Pause time after the command is stopped
    waMotorZ_MotionParameters[7] := uiMotorZ_Mode;      //PR Path 0 maps directly to Pr8.02,  16= run, 32= Homing, 33= manually set to zerc


    // Prepis v ModBus
    wModBusWrite_MotorZ_Enable := BOOL_TO_WORD(xWEB_MotorZ_Enable) OR BOOL_TO_WORD(xMotorZ_Enable);
    waModBusWrite_MotorZ_MotionParameters := waMotorZ_MotionParameters;
    xModBusWrite_MotorZ_Triger := xMotorZ_StartTriger OR xMotorZ_HomingTriger OR xMotorZ_CyclicStartTriger
    wModBusWrite_MotorZ_JogVelocity:= wAllMotors_JogVelocity;
    xModBusWrite_MotorZ_JogTrigger:= xMotorZ_JogTrigger;
    wModBusWrite_MotorZ_JogCommand:= wMotorZ_JogCommand;
```

*Figure 20: Rewriting Modbus variables to variables used in program*

Finally, the Modbus channels can be mapped to these variables inside the previously mentioned ModbusGenericSerialSlave I/O Mapping menu. As shown in Figure 21. Channels that I previously defined rising edge trigger also have a trigger variable.

| Variable | Mapping | Channel | Address | Type | Default Value | Unit | Description |
|---|---|---|---|---|---|---|---|
| Application.wModBusRead_MotorL_Status | | Channel 0 | ~~%IW62~~ | ARRAY [0..0] OF WORD | | | Read Holding Registers |
| Application.wModBusWrite_MotorL_Enable | | Channel 1 | ~~%QW114~~ | ARRAY [0..0] OF WORD | | | Write Single Register |
| Application.waModBusRead_MotorL_ActualPosition | | Channel 2 | ~~%IW64~~ | ARRAY [0..1] OF WORD | | | Read Holding Registers |
| Application.xModBusWrite_MotorL_Triger | | Channel 1 | ~~%QX116.0~~ | BIT | | | Trigger variable |
| Application.waModBusWrite_MotorL_MotionParameters | | Channel 3 | ~~%QW118~~ | ARRAY [0..7] OF WORD | | | Write Multiple Registers |
| Application.wModBusRead_MotorL_Error | | Channel 4 | ~~%IW68~~ | ARRAY [0..0] OF WORD | | | Read Holding Registers |
| Application.xModBusWrite_MotorL_JogTrigger | | Channel 8 | ~~%QX134.0~~ | BIT | | | Trigger variable |
| Application.wModBusWrite_MotorL_JogCommand | | Channel 8 | ~~%QW136~~ | ARRAY [0..0] OF WORD | | | Write Single Register |
| Application.xWEB_AllMotors_JogVelocityTrigger | | Channel 9 | ~~%QX138.0~~ | BIT | | | Trigger variable |
| Application.wModBusWrite_MotorL_JogVelocity | | Channel 9 | ~~%QW140~~ | ARRAY [0..0] OF WORD | | | Write Single Register |

*Figure 21: Mapping variables to channels*

## 7.3 Programming function block for motor movement

For simpler and more readable implementation of the state machine I decided to make a function block. Function block takes next input arguments:

- motorName – Here I specify which motor I want to move (Example input: “MotorX”)
- position – new target absolute position of the motor in encoder increments (Example input: 10000)
- acc – acceleration of the motor during movement (Example input: 100)
- dec – declaration of the motor during movement (Example input: 100)
- velocity – velocity in RPM during movement ((Example input: 50)

Function block returns next values as outputs:

- Done – Value is True when the movement of the motor finishes without an error.
- Inprogress – Value is True when the motor is moving towards a new position.
- Inposition - Value is True when the motor reaches targeted position.
- Error – True when an error occurs.

Inside the function block there is also an algorithm that detect is the motor hasn’t started to move after PLC send the command, so that in this case the function block tries to send the command again after 200ms. If the motor still doesn't move after sending the command ten times, there is probably something wrong with the hardware and function block returns an error.

Below I will include some of the most important parts of the function block code.

Figure 22 shows previously mentioned inputs and outputs of the block.

```
FUNCTION_BLOCK MotorControl

VAR_INPUT
    motorName: STRING(7);
    velocity: UINT;
    acc: UINT;
    dec: UINT;
    position: UDINT;
    request: BOOL;
END_VAR

VAR_OUTPUT
    InPosition : BOOL;
    InProgress : BOOL;
    Done : BOOL;
    Error : BOOL;
END_VAR
```

*Figure 22: Input/Output variables*

Figure 23 shows a part of the code that select variables of which motor need to be written to based on the motorName input.

```
        IF motorName ='MotorX' THEN
            motor:=1;
        ELSIF motorName ='MotorY' THEN
            motor:=2;
        ELSIF motorName ='MotorZ' THEN
            motor:=3;
        ELSIF motorName ='MotorL'  THEN
            motor:=4;
        ELSIF motorName ='MotorR' THEN
            motor:=5;
        ELSIF motorName ='MotorP' THEN
            motor:=6;
        END_IF
    END_IF

    CASE motor OF
        1: //x
            uiMotorX_Mode := Mode;
            uiMotorX_Velocity := velocity;
            udiMotorX_NewPosition := position;
            uiMotorX_Acc := acc;
            uiMotorX_Dec := dec;
            MotorX_Triger_MotorControl := trigger;
            actual_position := udiMotorX_ActualPosition;
            motorRunning := xMotorX_Running;
            pathCompleted := xMotorX_PathCompleted;
            xBusy := Modbus_Slave_COM_Port_MotorX.xBusy;
        2: //y
            uiMotorY_Mode := Mode;
            uiMotorY_Velocity := velocity;
            udiMotorY NewPosition := position;
```

*Figure 23: Selecting right motor*

Figure 24 is the algorithm that runs after the command to move has been sent to the motor. If the motor is running the program advances to the case 150. Second option is if the motor reaches targeted position very fast (before receiving back the running status) or if it already is in that position code also advances to case 150. Third option happens if neither of the about happens inside 200ms the timer runs out and code enters if statements that send it back to case 50 where the command is sent again.

```
100://
    trigger := FALSE;

    IF motorRunning  THEN
        trigger := FALSE;
        koraki_movement := 150;
    ELSIF WithinTolerance(actual_position, position) THEN
        trigger := FALSE;
        koraki_movement := 150;
    ELSIF  timerTrigger.Q AND NOT xBusy AND xInput2 THEN

        IF retryCount <= 10 THEN
            trigger:=FALSE;
            timerZakasnitevObNapaki(IN:=TRUE , PT:= , Q=> , ET=> );
            IF timerZakasnitevObNapaki.Q THEN
                retryCount := retryCount + 1;
                koraki_movement := 50;
                napake := napake+1;
                timerZakasnitevObNapaki(IN:=FALSE , PT:= , Q=> , ET=> );
                timerTrigger(IN:=FALSE , PT:= , Q=> , ET=> );

            END_IF

        ELSE
            koraki_movement := 999; //error
            trigger:=FALSE;
        END_IF

    END_IF
```

*Figure 24: algorithm for resending the command*

Figure 25: Shows output variables at the end of the movement.

```
 200://konec
     InProgress := FALSE;
     Done := TRUE;
     koraki_movement := 0;
     timerTrigger(IN:=FALSE , PT:= , Q=> , ET=> );
     timerGlobal(IN:=FALSE , PT:= , Q=> , ET=> );
     retryCount := 0;

 999://error
     InProgress := FALSE;
     Error := TRUE;
     timerTrigger(IN:=FALSE , PT:= , Q=> , ET=> );
     timerGlobal(IN:=FALSE , PT:= , Q=> , ET=> );
END_CASE
```

*Figure 25: output variables at the end of the movement*

### 7.3.1 Use example

Firstly, I created a separated FB for each motor so program can run multiple function blocks at the same time, one for each motor.

```
MotorP_RUN : MotorControl;
MotorX_RUN : MotorControl;
MotorY_RUN : MotorControl;
MotorZ_RUN : MotorControl;
MotorL_RUN : MotorControl;
MotorR_RUN : MotorControl;
```

*Figure 26: Mapping variables to function block*

Here is an example of the input for motor to move to a specific position with given parameters Figure 27.

```
MotorZ_RUN(
    motorName:='MotorZ' ,
    velocity:=100 ,
    acc:=500 ,
    dec:=500 ,
    position:=1000 ,
    request:=TRUE ,
    InPosition=> ,
    InProgress=> ,
    Done=> ,
    Error=> );
```

*Figure 27: Setting variables of function block*

Program also allows accessing single inputs and outputs. So, I don't need to input all the parameters each time I can set the velocity, acc, dec and motorName parameters at the start of the program and execute movement only by changing the position and setting the trigger to TRUE during later movements as shown in Figure 28.

```
600://odmakne Y
    motorY_run.position := udiMovePosY;
    MotorY_RUN.request:=TRUE;
    MotorY_RUN();
    IF MotorY_RUN.Done  THEN
        MotorY_RUN.request := FALSE;
        UiKorakiAvtoPospravi :=700;
    END_IF

700://Z na lokacijo prevzemnega mesta
```

*Figure 28: single variable access*

## 7.4 Setting up web visualisation

Eaton XC-204 has a built-in web visualization feature that allows users to monitor and control the automation system through a standard web browser. It works similarly to an HMI (Human-Machine Interface), displaying real-time data, buttons, status indicators, and process graphics. Instead of using a physical HMI panel, the interface is hosted on the PLC and accessed remotely via IP address using a web browser.

Web visualization is set up inside the object tree by clicking on ‘Add object’ and selecting ‘Visualization’

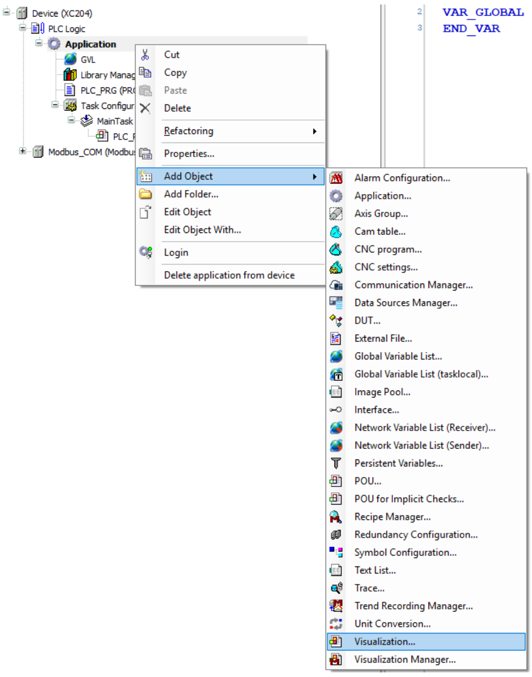



*Figure 29: Creating visualisation*

Elements of the visualisation are added from the toolbox Figure 30.

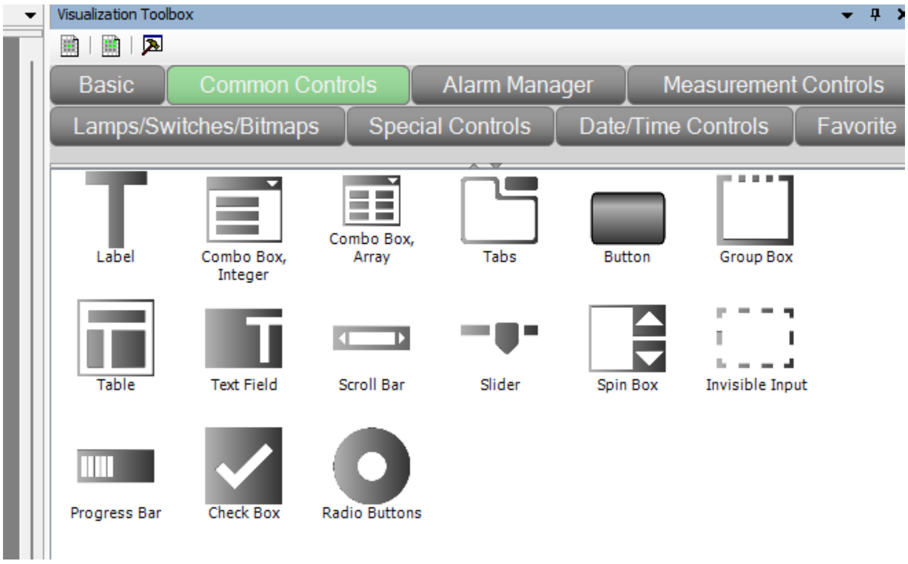



*Figure 30: Toolbox*

After adding the elements such as buttons, textboxes etc. I set the variables which they are mapped to in the properties box of each element.

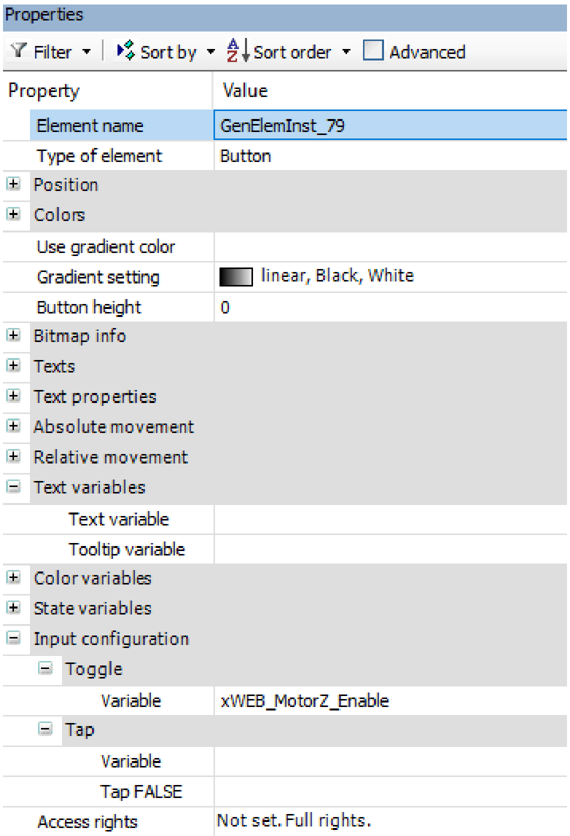



*Figure 31: Element properties*

Figure 32 shows one of the pages of the web visualisation.

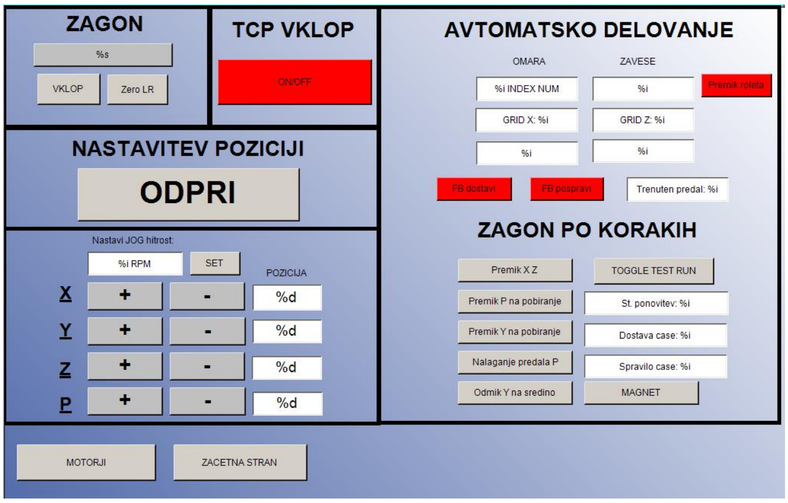



*Figure 32: Visualization*

## 7.5 State machine programming

The automatic movement of the machine is achieved by using a state machine based code. On power on the machine all motors need to be referenced this is achieved by homing all the motors, this is done by setting the mode and operation mode of the motors to homing and triggering it. Part of the power on code is shown below.

```
500: sKorakiVklop:= 'Homing Y';
    uiMotorY_Mode := 32;
    uiMotorY_OperationMode := 3;
    xMotorY_HomingTriger := TRUE;
    IF xMotorY_Running  THEN
        xMotorY_HomingTriger := FALSE;
        uiKorakiVklopNaprave := 510;
    END_IF
510:
    IF xMotorY_PathCompleted = TRUE AND xMotorY_CommandCompleted AND xMotorY_HomingCompleted THEN
        uiKorakiVklopNaprave := 600;
    END_IF

600: sKorakiVklop:= 'Homing X';
    uiMotorX_Mode := 32;
    uiMotorX_OperationMode := 3;
    xMotorX_HomingTriger := TRUE;
    IF xMotorX_Running  THEN
        xMotorX_HomingTriger := FALSE;
        uiKorakiVklopNaprave := 610;
    END_IF
610:
    IF xMotorX_PathCompleted = TRUE AND xMotorX_CommandCompleted AND xMotorX_HomingCompleted THEN
        uiKorakiVklopNaprave := 700;
    END_IF
```

*Figure 33: Homing*

After machine is powered on and all motors are homed the automatic sequence can begin. First few lines of the code of automatic sequence are displayed in Figure 34

```
50://nastavi motorje
    finishedFBSEQ:=FALSE;
    MotorX_RUN(motorName:='MotorX', velocity:=350 , acc:=4000 , dec:=4000);
    MotorY_RUN(motorName:='MotorY', velocity:=15 , acc:=4000 , dec:=4000);
    MotorZ_RUN(motorName:='MotorZ', velocity:= MotorZ_fast ,acc:=4000 , dec:=4000);
    MotorL_RUN(motorName:='MotorL', velocity:=30 , acc:=1000 , dec:=1000);
    MotorR_RUN(motorName:='MotorR', velocity:=30 , acc:=1000 , dec:=1000);
    MotorP_RUN(motorName:='MotorP', velocity:=MotorP_Fast , acc:=5000 , dec:=5000);

    //TCP-----
    bStatusOmare := 3;
    UiKorakiAvtoDostava := 80;

80: //prepis poziciji
    StepCountXSave:= udiStepCountX;
    StepCountZSave:= udiStepCountZ;
    uiDrawerIndexSave := uiDrawerIndex;
    UiKorakiAvtoDostava := 100;

100://preveri Y, če je odmaknjen
    motorY_run.position := udiMovePosY;
    MotorY_RUN.request:=TRUE;
    MotorY_RUN();

    IF MotorY_RUN.Done  THEN
        MotorY_RUN.request := FALSE;
        UiKorakiAvtoDostava :=200;
    END_IF

200://premakne X na lokacijo
    UiKorakiAvtoDostava :=210;
    MotorZ_RUN.request:=TRUE;
    MotorX_RUN.request:=TRUE;
210://premakne Z na lokacijo
    MotorZ_run.position := StepCountZSave;
    MotorX_run.position := StepCountXSave;
    MotorZ_RUN();
    MotorX_RUN();

    IF MotorX_RUN.Done THEN
        MotorX_RUN.request := FALSE;
    END_IF

    IF MotorZ_RUN.Done THEN
        MotorZ_RUN.request := FALSE;
    END_IF

    IF MotorX_RUN.Done AND MotorZ_RUN.Done THEN
        UiKorakiAvtoDostava := 250;
    END_IF
```

*Figure 34: state machine code*

# 8 Conclusion

In this project, I successfully implemented closed-loop stepper motor control using the Eaton XC-204 PLC and CL57RS drivers, communicating over Modbus RTU. The system allows for precise and reliable motion control, with automatic correction of positioning errors via encoder feedback. Using CODESYS V3 and structured text programming, I developed a modular and maintainable control program, including a state machine and function blocks for scalable control. The built-in web visualization feature on the XC-204 provided an effective alternative to a traditional HMI, allowing remote control and monitoring through a standard web browser. The experience gained through configuring drivers, programming Modbus communication, and building the control logic has deepened my understanding of industrial motion control systems and PLC-based automation.

# References

[1] STEPPERONLINE, *CL57RS Closed Loop Stepper Driver User Manual*. [Online]. Available: https://www.omc-stepperonline.com/modbus-rs485-closed-loop-stepper-motor-driver-0-5-7-0a-24-48vdc-cl57rs

[2] CODESYS GmbH, *CODESYS Examples Documentation Portal*. [Online]. Available: https://content.helpme-codesys.com/en/CODESYS%20Examples/_ex_start_page.html

[3] EATON, *Modular PLCs XControl Manual*. [Online]. Available: https://www.eaton.com/us/en-us/catalog/machinery-controls/xc-modular-programmable-logic-controllers--plcs-.html#tab-2