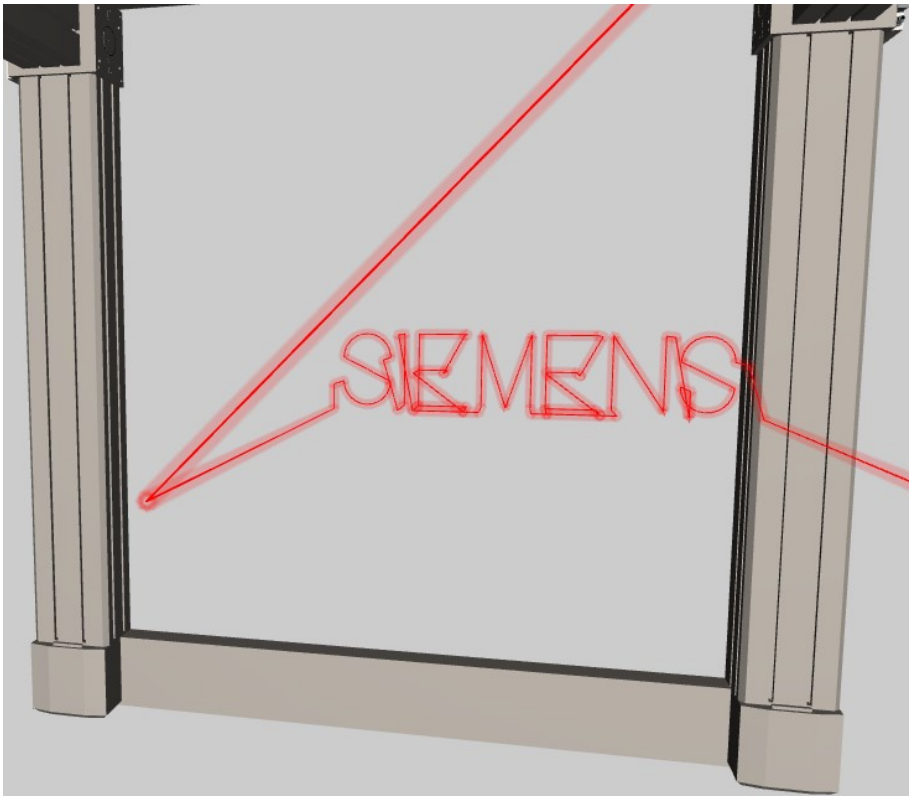
SIEMENS

Siemens NV/AG Huizingen

TIA demo project manual

Writing with G-code



Project and manual made by
Pieter Van den Bossche

Guided and supervised by
Ir. Sander Van Gorp

This project and manual was made as part of an internship during August-October 2021 at Siemens NV/AG Huizingen.

Contents

Preface

During this internship, my task was to implement additional functions on the POCFA demo machine. The most important feature to add was getting the machine to write the text that a user types on a HMI on a business card, with the requirement of using G-code to define the movement commands.

In order to accomplish this, I made this TIA application example on a demo case, based on the *LKinCtrl* and *LKinLang* library application examples. All axes are simulated, the goal was to get the portal (in TIA trace) to write the text which the user inputs in the HMI, maximizing the flexibility of the program and using a minimum of blocks and variables.

This manual was written for any future users in order to make it easier to step into TIA projects containing movement based on G-code. In this document, the set-up and working of the program and the function of each of the blocks used is explained, along with some insights on generating the G-code and the handling of the program.

Software used

The following software was used in this project:

- TIA Portal V16
- StartDrive
- WinCC Advanced
- S7-PLCSIM Advanced V3.0 Upd2

Chapter 1

Set-up project

1.1 Importing libraries

In this project, two libraries are used, one to load and convert a *.nc* file into a path data structure (*LKinLang*) and one to enable the movement from a this path data structure inside of a DB (*LKinCtrl*). These can be downloaded, along with clear manuals ([1], [2]) and example projects on LKinCtrl link and LKinLang link.

To import these libraries, open the *Libraries* tab and click on *Open global library*, see figure 1.1. Select the library and let TIA process it for a while.

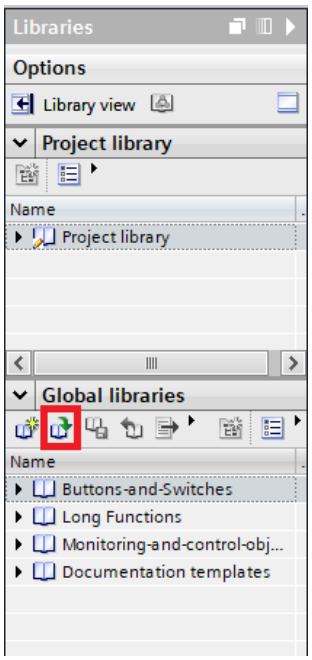


Figure 1.1: Indication of how to import the libraries

When the library is imported, the library will appear inside of the *Global libraries* list, as shown in figure 1.2.

Open the library sublist and make sure to copy the library blocks folder to the PLC's *Program blocks* folder. Also, copy the tags and types folders of the library to the tags and types of the PLC. This whole procedure is shown in figures 1.3a and 1.3b. Repeat this for both the libraries.

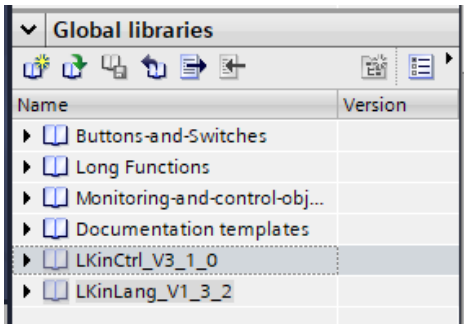
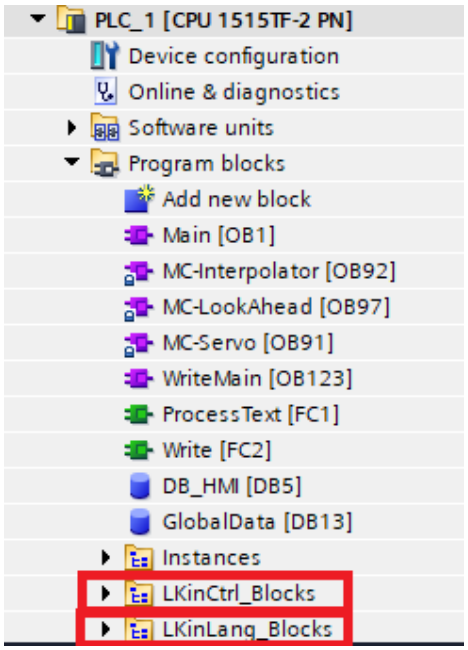


Figure 1.2: Indication of the location of the libraries after importing



(a) Copying the library blocks to the PLC blocks

(b) Copying the library tags and types to the PLC tags and types

Figure 1.3: Library block copying procedure

1.1.1 *LKinCtrl* library

The first library contains the FB *LKinCtrl_MC_MovePath*, which executes the movement of the axes if a path data structure [1] is given to its *pathData* input, see the block diagram in figure 1.4.

For a full description and more information on the usage of this block, refer to the *LKinCtrl* manual [1], which can be found in the link given in section 1.1, along with an application example. Other blocks used from this library are the FB's *LKinCtrl_MC_GroupPower*, *LKinCtrl_MC_GroupHome* and *LKinCtrl_MC_GroupReset*, which do their usual task, but for all axes at the same time.

[1] There are three types of path data structures: *LKinCtrl_typePathData_reduced*, *LKinCtrl_typePathData* and *LKinCtrl_typePathData_advanced*. The main difference is the amount of commands each path data structure can store. In this project, the *LKinCtrl_typePathData_advanced* is used, because the *LKinLang_Parser* FB can only produce this type.

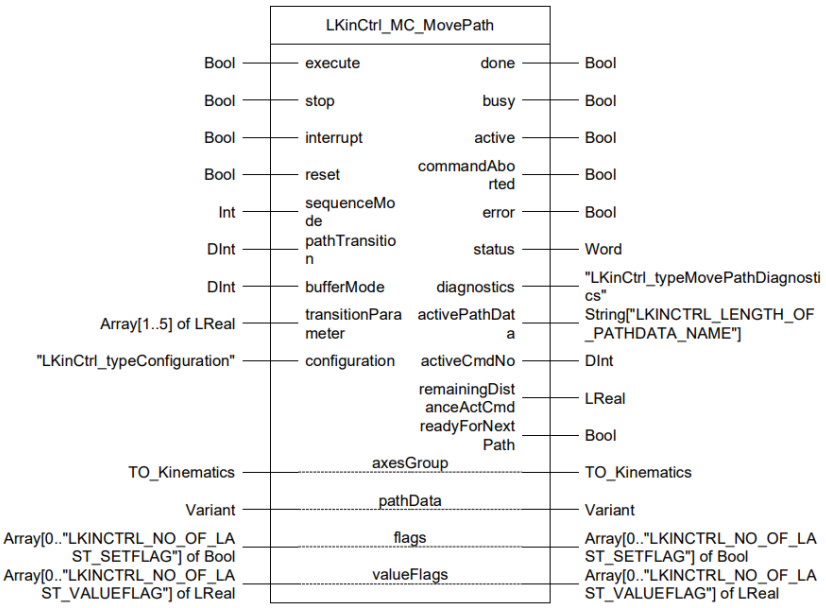


Figure 1.4: Block diagram of the FB *LKinCtrl_MC_MovePath*, found in the *LKinCtrl* manual [1]

1.1.2 *LKinLang* library

The second library includes the blocks to load the *.nc* files from the PLC memory and to convert it to a path data structure which can be read by the *LKinCtrl_MC_MovePath* FB.

The first block used in the project is the FB *LKinLang_FileHandler*, which loads the *.nc* files into the project. Its working is shown in the diagram of figure 1.5 and the block diagram of the FB is given in figure 1.6. In short, the FB searches for the file name (given in *String* format to the input *fileName* of the FB) inside of the PLC memory. When found, it reads and transfers the data line by line into the *lineBuffer* variable of the *DB_Writing* PLC data block, that is working as a data buffer.

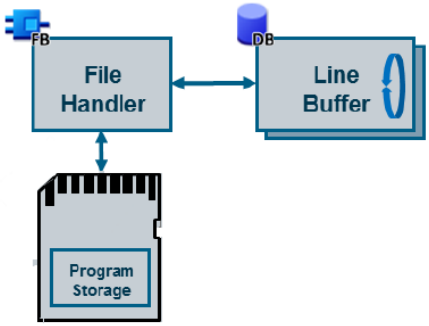


Figure 1.5: Diagram to show the working of the FB *LKinLang_FileHandler*, found in the *LKinLang* manual [2]

The second block used is the FB *LKinLang_Parser*, which converts the data transferred by the FB *LKinLang_FileHandler* into a path data structure. Its working is shown in the diagram of figure 1.7 and the block diagram of the FB is given in figure 1.8.

If desired, the library comes with its own faceplate, with which one can edit the path data on the HMI and some FB's to handle the faceplate. For further details, refer to the *LKinLang* manual [2].

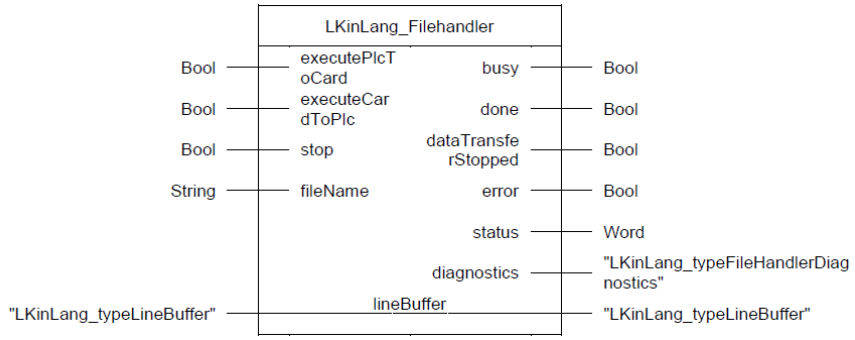


Figure 1.6: Block diagram of the FB *LKinLang_FileHandler*, found in the *LKinLang* manual [2]

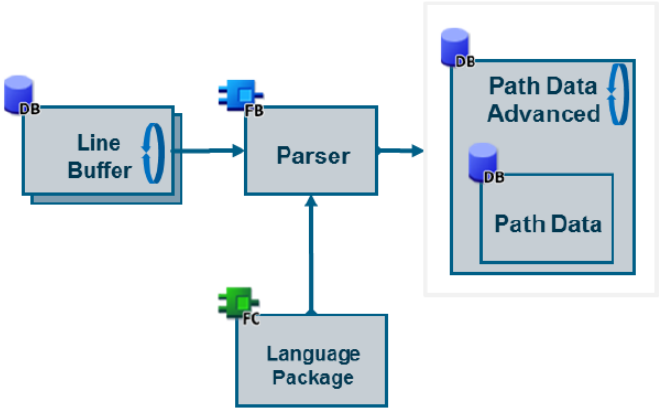


Figure 1.7: Diagram to show the working of the FB *LKinLang_Parser*, found in the *LKinLang manual* [2]

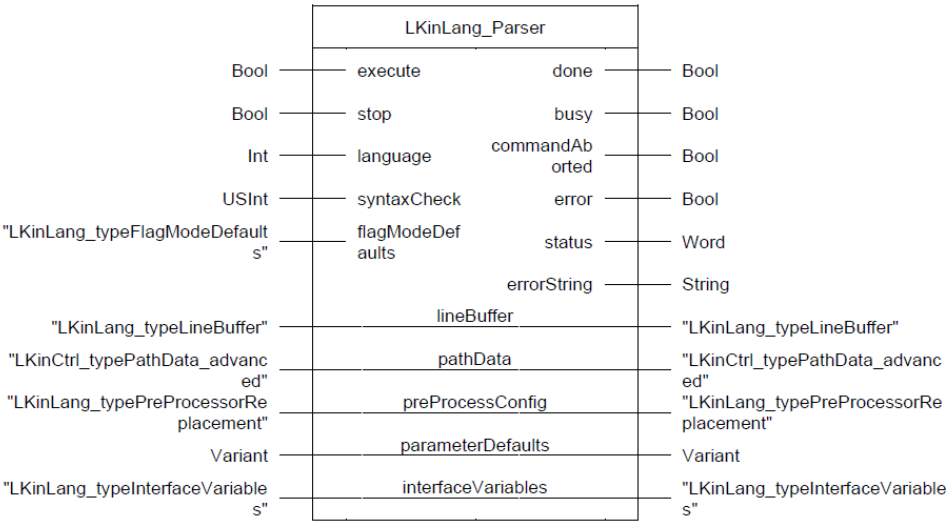


Figure 1.8: Block diagram of the FB *LKinLang_Parser*, found in the *LKinLang manual* [2]

1.2 PLC

1.2.1 Technology objects

In order to control the axes, some technology objects are needed. 3 (positioning) axes were configured and a Cartesian Portal 3D was chosen as the kinematics. All of these axes were put into simulation mode, as shown in figure 1.9.

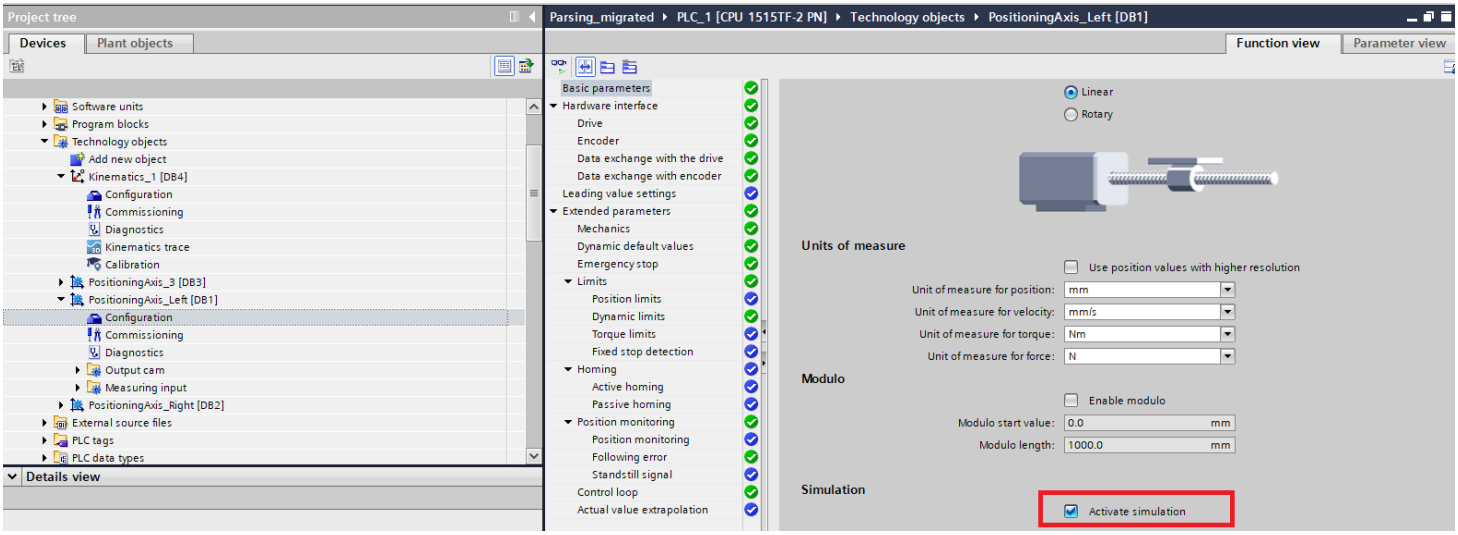


Figure 1.9: Activation of the simulation of axes

1.3 HMI

1.3.1 Overview

To avoid using watch tables all the time, a simple HMI screen was made. It contains the bare minimum to control this application, along with some status displays. In figure 1.10, the start-up screen (which is the only screen) of the HMI is shown.

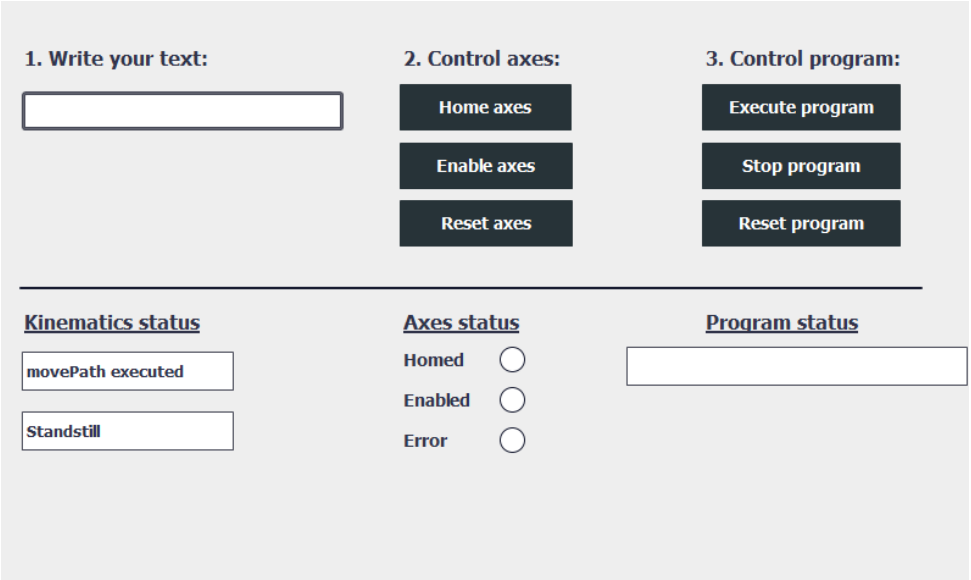


Figure 1.10: Begin screen of the HMI

The user first has to write the desired text, home the axes and enable them and finally execute the program. After doing so, the portal will write the text which was put into the HMI.

Then, click on an object in the screen and go to its properties to know which tag is coupled to the state of the object.

1.3.2 HMI tags

The HMI tags are used to couple the state of buttons and text input/output fields on the HMI to. These tags are then coupled to tags in the PLC itself, so that they can be processed there. The HMI tags and the PLC tags to which they are coupled are shown in figure 1.11.

Default tag table

Name	Data type	Connection	PLC name	PLC tag
AxesEnabled	Bool	HMI_Connection_1	PLC_1	DB_Writing.AxesEnabled
AxesError	Bool	HMI_Connection_1	PLC_1	DB_Writing.AxesError
AxesHomed	Bool	HMI_Connection_1	PLC_1	DB_Writing.AxesHomed
EnableAxes	Bool	HMI_Connection_1	PLC_1	DB_Writing.EnableAxes
executeProgram	Bool	HMI_Connection_1	PLC_1	DB_Writing.executeProgram
HomeAxes	Bool	HMI_Connection_1	PLC_1	DB_Writing.HomeAxes
movePath_active	Bool	HMI_Connection_1	PLC_1	DB_Writing.movePath_active
movePath_status	Word	HMI_Connection_1	PLC_1	DB_Writing.movePath_status
programStatusString	String	HMI_Connection_1	PLC_1	DB_Writing.programStatusString
ResetAxes	Bool	HMI_Connection_1	PLC_1	DB_Writing.ResetAxes
resetProgram	Bool	HMI_Connection_1	PLC_1	DB_Writing.resetProgram
stopProgram	Bool	HMI_Connection_1	PLC_1	DB_Writing.stopProgram
text	String	HMI_Connection_1	PLC_1	DB_Writing.textInput
triggerTextbox	Bool	HMI_Connection_1	PLC_1	DB_Writing.triggerTextbox
<Add new>				

Figure 1.11: HMI tags

If one wants to know which object on the HMI is coupled to the HMI tag, cross-reference on the tag or go to the HMI *Screens* tab, and double click on the *Writing_controls* screen.

1.4 Drives

Since all axes are put into simulation mode, no communication with any drives is needed. However, somehow TIA doesn't want to run with an empty network, so the drives of the demo case were left in the project. They are not used for anything, but they have to be connected correctly in the network and in the topology.

Chapter 2

Program overview

2.1 Program structure

The program is very minimalistic. As one can see in figure 2.1, only 2 OB's, 2 FB's and 1 DB are used, along with the usual instance DB's of blocks and standard OB's. In the following sections, the blocks will be shortly discussed.

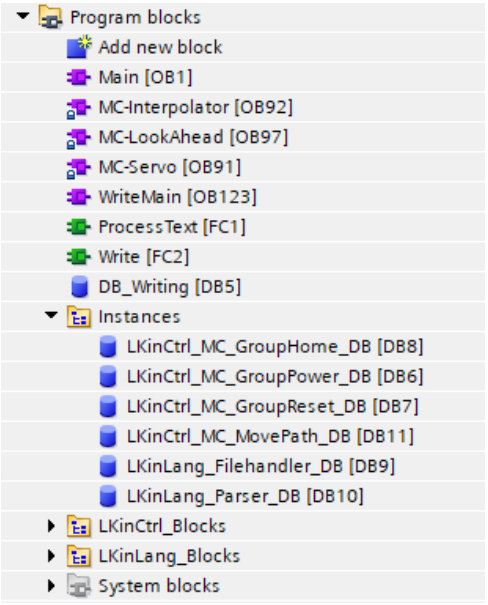


Figure 2.1: General program overview

2.2 OB *Main*

In this OB, the basic motion commands are given to the kinematics technology objects. These are the powering, homing and resetting commands for all axes at the same time, using the FB's *LKinCtrl_MC_GroupPower*, *LKinCtrl_MC_GroupHome* and *LKinCtrl_MC_GroupReset*. Also, the error and homing states of the axes are retrieved.

2.3 OB *WriteMain*

First, some program logic is added in the first two networks to stop the whole program (*LKinLang_FileHandler*, *LKinLang_Parser* and *LKinCtrl_MC_MovePath* FB) at the same time and to reset the axes and the *LKinCtrl_MC_MovePath* FB at the same time.

Network 1: Program logic to stop all FBs at once

Comment

```
// Stop all FBs at once with "stopProgram"
IF "DB_Writing".stopProgram THEN
    "DB_Writing".fileHandler_stop := TRUE;
    "DB_Writing".parser_stop := TRUE;
    "DB_Writing".movePath_stop := TRUE;
END_IF;

// Falling edge of stop program
"FTrigStopProgram_DB"(CLK:="DB_Writing".stopProgram);

// Reset stop bits when falling edge is detected
IF "FTrigStopProgram_DB".Q THEN
    "DB_Writing".fileHandler_stop := FALSE;
    "DB_Writing".parser_stop := FALSE;
    "DB_Writing".movePath_stop := FALSE;
END_IF;
```

Network 2: Program logic to reset axes and movePath FB at once

Comment

```
// Reset all FBs at once with "resetProgram"
IF "DB_Writing".resetProgram THEN
    "DB_Writing".movePath_reset := TRUE;
    "DB_Writing".ResetAxes := TRUE;
END_IF;

// Falling edge of reset program
"FtrigResetProgram_DB"(CLK:="DB_Writing".resetProgram);

// Reset stop bits when falling edge is detected
IF "FtrigResetProgram_DB".Q THEN
    "DB_Writing".movePath_reset := FALSE;
    "DB_Writing".ResetAxes := FALSE;
END_IF;
```

This is followed by a reset of the *programDone* boolean on a rising edge of the *triggerTextbox* boolean (set high every time the user finishes the text input in the HMI). The *programDone* boolean is obviously set high every time the program finishes and serves to keep track of the program status.

Network 3: Reset programDone bool on rising edge

Comment

```
// Act when rising edge of trigger for processing text (set in HMI) is detected
IF "DB_Writing".triggerTextbox AND NOT "DB_Writing".oldTriggerTextbox THEN
    "DB_Writing".programDone := FALSE;
END_IF;

"DB_Writing".oldTriggerTextbox := "DB_Writing".triggerTextbox;
```

Then, the *ProcessText* FC, (which will be discussed more in detail in section 2.4) is called on a rising edge of the *executeProgram* boolean (set high every time the user presses the *Execute Program* button on the HMI). In short, this FC selects the correct *.nc* file for each char of the input string. On the same rising edge, the pointer used for the main loop of the program is reset, the *programDone* boolean is reset and the *String*

array *gcodeFileNames* is set to a copy. This is needed for example when executing the program twice without changing the input text, the original array will be processed and the copy remains untouched.

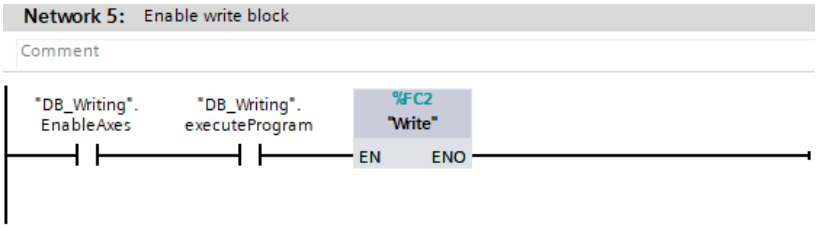
Network 4: (Re-)initialize step value when pressing executeProgram/Get the letters from the input text block into array ...

Comment

```
// Act when rising edge of trigger executeProgram is detected
IF "DB_Writing".executeProgram AND NOT "DB_Writing".oldTriggerExecuteProgram THEN
    "ProcessText"(fileNameCapsTemp := "DB_Writing".fileNameCapsTemp,
                  fileNameTemp := "DB_Writing".fileNameTemp,
                  beginPosFileName := "DB_Writing".beginPosFileName,
                  endPosFileName := "DB_Writing".endPosFileName,
                  textInput := "DB_Writing".textInput,
                  textCharArray := "DB_Writing".TextCharArray,
                  spaceString := "DB_Writing".spaceString,
                  gcodeFileNames => "DB_Writing".gcodeFileNames,
                  programLength => "DB_Writing".programLength);
    "DB_Writing".statActualStep := 10;
    "DB_Writing".gcodeFileNames := "DB_Writing".gcodeFileNames_copy; // reset the commands to an original copy
    "DB_Writing".programDone := FALSE;
END_IF;

"DB_Writing".oldTriggerExecuteProgram := "DB_Writing".executeProgram;
```

Finally, this OB contains the initialisation of the main loop of the program in its last network.



2.4 FC *ProcessText*

In this FC, the input string first gets converted to an array of chars. For each char, the appropriate *.nc* file name is put into an array of strings. Uppercase letters have to be distinguished from lowercase letters, so the ASCII value is compared in order to do so: if the ASCII value is between 65 and 90, the letter is an uppercase letter. If not, the letter is a lowercase letter. Also, spaces are recognized by comparing them to an empty string: " ". Moreover, if the text length is too long (longer than the *enterLength* int), an enter command is inserted at the proper place.

The SCL code of this FC is shown in figure 2.2:

Network 1: Construction of the array of commands, select the right g code files

Comment

```
// Go to begin position first before writing (letters are defined relative to begin position)
"DB_Writing".gcodeFileNames[0] := #beginPosFileName;

// Convert input string to array of chars
Strg_TO_Chars(Strg := #textInput,
              pChars := 0,
              Cnt => "DB_Writing".textLength,
              Chars := #textCharArray);

// Loop over array of chars, make the file names
#counter := 0;
#numberofenters := 0;
FOR #i := 1 TO "DB_Writing".textLength BY 1 DO
    #counter := #counter + 1;

    Chars_TO_Strg(Chars := #textCharArray,
                  pChars := #i - 1,
                  Cnt := 1,
                  Strg => #TempString);
    #stringInAscii := #textCharArray[#i - 1];

    // Ascii value between 65 and 90 --> uppercase letter
    IF (#stringInAscii >= 65 AND #stringInAscii <= 90) THEN
        #TempString2 := CONCAT(IN1 := #fileNameCapsTemp, IN2 := #TempString);

    // Not uppercase
    ELSE

        // A space is detected
        IF (#TempString = #spaceString) THEN
            #TempString2 := 'space';

        // Lowercase
        ELSE
            #TempString2 := CONCAT(IN1 := #fileNameTemp, IN2 := #TempString);
        END_IF;

    END_IF;

    #FileName := CONCAT(IN1 := #TempString2, IN2 := '.nc');
    "DB_Writing".gcodeFileNames[#counter] := #FileName;

    // An enter is detected or the text is going out of bounds
    IF (#stringInAscii = 10) OR (#i MOD "DB_Writing".enterLenght = 0) AND NOT (#i = "DB_Writing".textLength) THEN
        #counter := #counter + 1;
        "DB_Writing".gcodeFileNames[#counter] := 'enter.nc';
        #numberofenters := #numberofenters + 1;
    END_IF;

END_FOR;

// Go to end position
#programLength := "DB_Writing".textLength + 2 + #numberofenters;
"DB_Writing".gcodeFileNames[#programLength - 1] := #endPosFileName;

// Reset rest of array
FOR #i := #programLength TO "DB_Writing".maxTextLength + 1 + #numberofenters BY 1 DO
    "DB_Writing".gcodeFileNames[#i] := '';
END_FOR;

// Give as output
#gcodeFileNames := "DB_Writing".gcodeFileNames;

// Make a copy, because other array will be processed
"DB_Writing".gcodeFileNames_copy := #gcodeFileNames;

```

Figure 2.2: FB *ProcessText* SCL code

2.5 FC *Write*

This FC contains in the first place the main loop of the program. This is a case statement, because a for-loop cannot be used (it takes more than a cycle to complete the operations done in this network). A pointer (the int *statActualStep*) is used to go to the next case, when the operations in the current case are correctly executed. In the first case, the *.nc* file is loaded by the FB *LKinLang_FileHandler*, in the second case the *.nc* file is parsed by the FB *LKinLang_Parser* and in the third case the movement is executed by the FB *LKinCtrl_MC_MovePath*. The last case is used to initiate the same procedure for the next letter of the word, by shifting the array of command strings by one and setting the pointer back to 10.

The SCL code of the main loop is shown in figure 2.3:

Network 1: Loading/parsing/writing, main loop of program

Comment

```
"DB_Writing".filename := "DB_Writing".gcodeFileNames[0];

IF LEN("DB_Writing".filename) > 0 THEN

    CASE "DB_Writing".statActualStep OF

        10: // Load NC file from memory
            "DB_Writing".loadFile := TRUE;
            "DB_Writing".programStatusString := CONCAT(IN1 := 'Loading ', IN2 := "DB_Writing".filename);

            IF "DB_Writing".fileHandler_done AND NOT "DB_Writing".fileHandler_error THEN
                "DB_Writing".statActualStep := 20;
                "DB_Writing".loadFile := FALSE;
            END_IF;

        20: // Parse NC file
            "DB_Writing".parser_execute := TRUE;
            "DB_Writing".programStatusString := CONCAT(IN1 := 'Parsing ', IN2 := "DB_Writing".filename);

            IF "DB_Writing".parser_done AND NOT "DB_Writing".parser_error THEN
                "DB_Writing".statActualStep := 30;
                "DB_Writing".parser_execute := FALSE;
            END_IF;

        30: // Move
            "DB_Writing".movePath_execute := TRUE;
            "DB_Writing".programStatusString := CONCAT(IN1 := 'Writing ', IN2 := "DB_Writing".filename);

            IF "DB_Writing".movePath_done AND NOT "DB_Writing".movePath_error THEN
                "DB_Writing".statActualStep := 40;
                "DB_Writing".movePath_execute := FALSE;
            END_IF;

        40: // Initiate new array, shifted by one place, delete first entry of previous array
            "DB_Writing".programStatusString := 'Making array';

            FOR #i := 1 TO "DB_Writing".programLength - 1 DO
                #dummyArray[#i - 1] := "DB_Writing".gcodeFileNames[#i];
            END_FOR;

            #dummyArray["DB_Writing".programLength - 1] := '';
            "DB_Writing".gcodeFileNames := #dummyArray;
            "DB_Writing".statActualStep := 10;

    END_CASE;

ELSE
    "DB_Writing".programStatusString := 'Program finished';
    "DB_Writing".programDone := TRUE;
END IF;
```

Figure 2.3: Program main loop

The *Write* FC contains also the blocks itself needed for the writing functionality: the FB *LKinLang_FileHandler*, the FB *LKinLang_Parser* and then the FB *LKinCtrl_MC_MovePath*.

2.6 DB *DB_Writing*

This DB contains all PLC tags to which the HMI tags are coupled, along with all the non-instance variables of all blocks used in the program.

Chapter 3

G-code

3.1 Creating G-code files of letters

Different methods of creating a G-code file for each letter were researched.

3.1.1 Online tools

To avoid the tedious work of writing G-code by hand, some online tools were investigated and tested, such as:

- CNC-apps: Text to G-CODE (linear path): paying software, would be the best option. Syntax not completely right, but possible to generate relative G-code.
- InkScape: free, can be used to go from images to G-code. Syntax not completely right.
- DeskEngrave: syntax wrong, only absolute G-code possible.
- Convert *.stl* files of letters to *.dxf* format, and then to G-code using some online converters

All these methods were not suited for this project, as the FB *LKinLang_Parser* has a certain syntax. The G-code generated by all online tools listed above does not comply to this syntax, so sadly this way of generating G-code is not appropriate for the application.

The syntax of the parser demands for example that there is *M30* written at the end of the *.nc file*. Also, the order in which initial commands are given is important. Hereunder, one can find as an example the code inside of a G-code file to write a capital B.

```
G17
G91
F100
G1 Z-10
G1 Y15
G1 X2
G2 Y-7.5 I0 J-3.75
G1 X-2
G1 X2
G2 Y-7.5 I0 J-3.75
```

```
G1 X-2
G1 Z10
G1 X8.75
M30
```

It can be noted that the *G17* command (select plane in which circular movements will be carried out) has to be set before the *G91* command (set relative movement), otherwise the parser will throw syntax errors. As mentioned, an *M30* command has to be given to indicate the end of the *.nc* file. This syntax makes it an almost impossible task to find online software that makes G-code that is compatible with the parser.

3.1.2 Writing by hand

To be able to test the program, I wrote a provisional alphabet of uppercase letters by hand in G-code. 15mm was used as the height of each uppercase letter and the width varies between 8-12mm, depending on the shape of the letter. This was done on sight, without sticking to a certain font, so these letters are not the prettiest and improvement can be made.

Since multiple letters have to be written after each other, not in a fixed order (since the user can write any word he wants) the best way to write the G-code is to work with relative movement to allow for maximal flexibility. Using relative positioning means that every letter can be written separately from each other, without having the need of knowing the absolute position.

The G-code files were written in such a way that the machine has to move 10mm to the negative z direction to begin to write, and ends at the same y position with the z position again 10mm above the paper and with the x position spaced some mm's with respect to the just written letter. This is illustrated in figure 3.1. When moving on to the next letter, the relative coordinate system is moved to the current position, and the next letter is written in the same way.

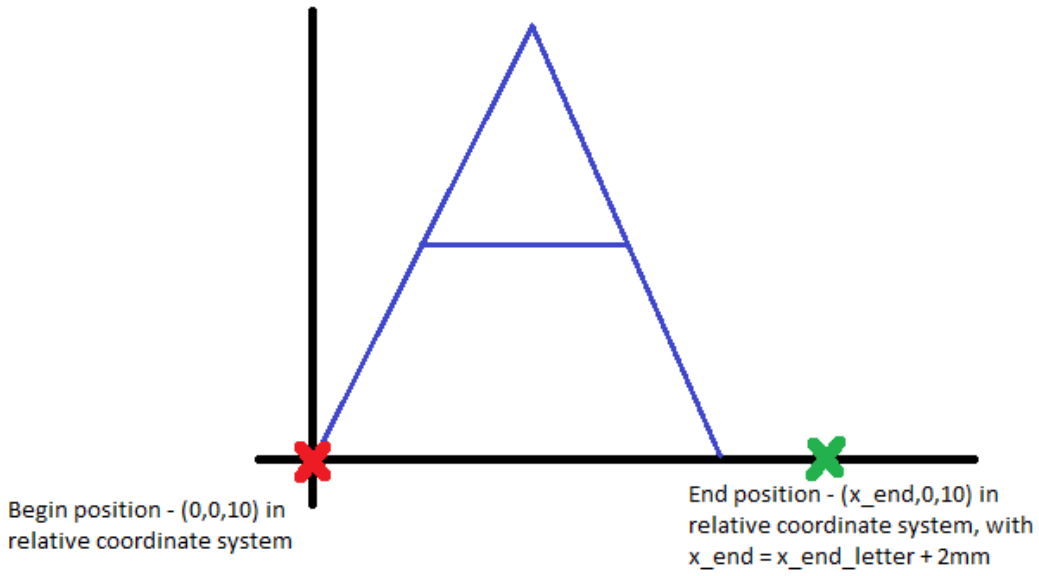


Figure 3.1: Procedure of writing of a letter in a relative coordinate system

An absolute begin position was added so the portal starts to write from there. Also, an absolute end position was added to which the portal moves after writing, the letters themselves are all relative. The adding of the begin- and end positions can be seen in the code given in section 2.4.

3.1.3 NX

Apparently, it is possible to make a CAD model of a letter in NX, and to convert it into G-code. I did try this method, so this might be a viable option, provided that relative G-code can be generated instead of absolute.

3.2 Putting G-code files on memory card

To be able to access the G-code files in the program, they have to be transferred to the memory of the PLC.

Note: If one wants to run this program on a PLCSIM Advanced instance, refer to subsection 4.2.3, because the method used to put G-code files in the memory of the PLC (instance) differs.

For a physical PLC, the most convenient method is using the web server. However, this function has to be enabled first. To do so, go to the *Online & diagnostics* tab of the PLC and click on the *Properties* panel, there one can find the *Web server* menu. Make sure that the *Activate web server on this module* option is activated and the *Permit access only with HTTPS* is deactivated. This procedure is shown and indicated in figure 3.2.

Additionally, one must make sure to enable the last three options in the *Access level* drop-down menu in the *User management* table at the bottom of the *Web server* menu. This is again illustrated in figure 3.3.

The last step is to check if the option *Enable Web server via IP address of this interface* is enabled under the menu *PROFINET interface [X1]*, in submenu *Web server access*. This is visualised in figure 3.4.

At last, one can access the file browser by typing the IP-address of the PLC into the browser search bar, for instance: `http://192.168.1.11/`. There, one can add or delete files in the PLC memory.

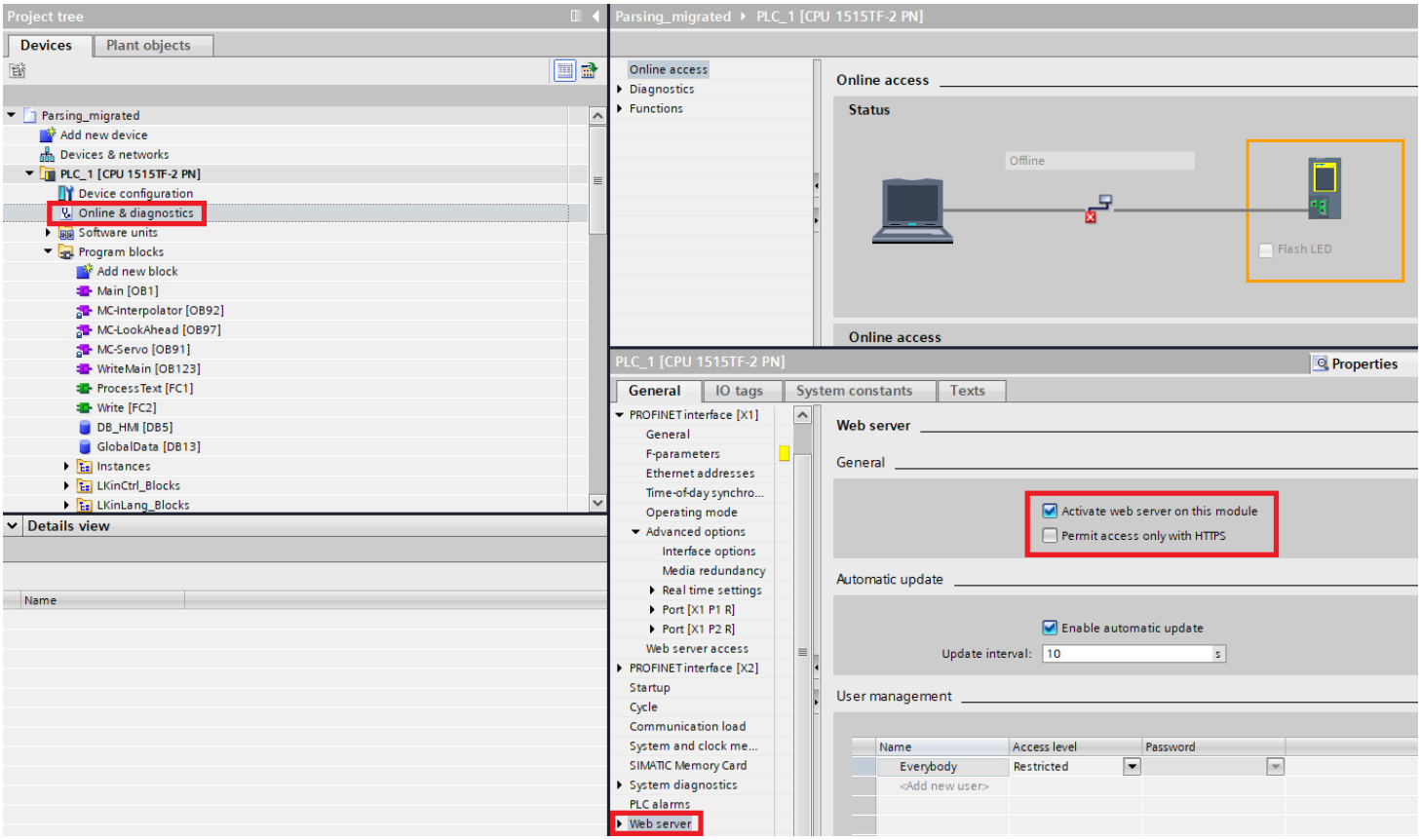


Figure 3.2: Indication of how to enable the web server

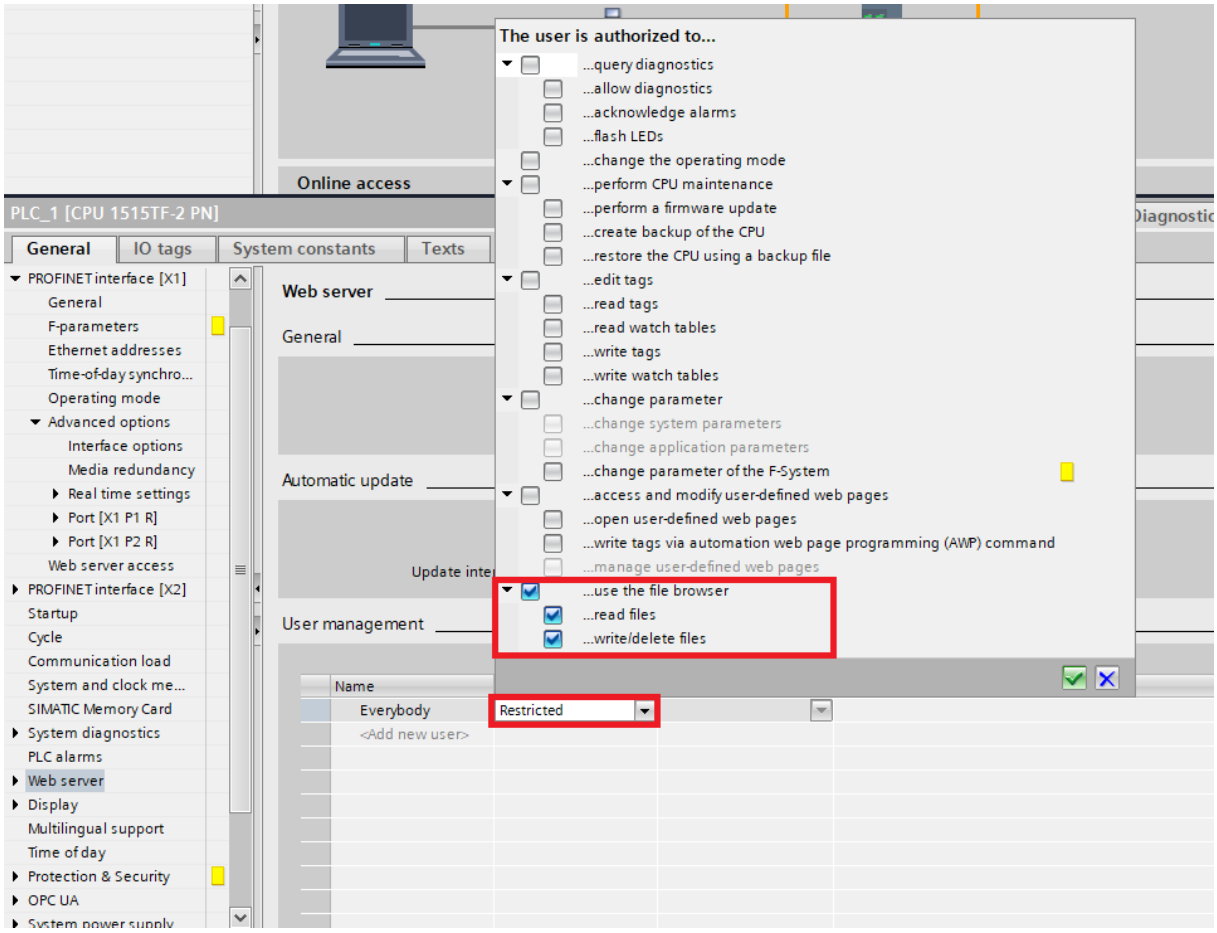


Figure 3.3: Indication of how to enable the web server

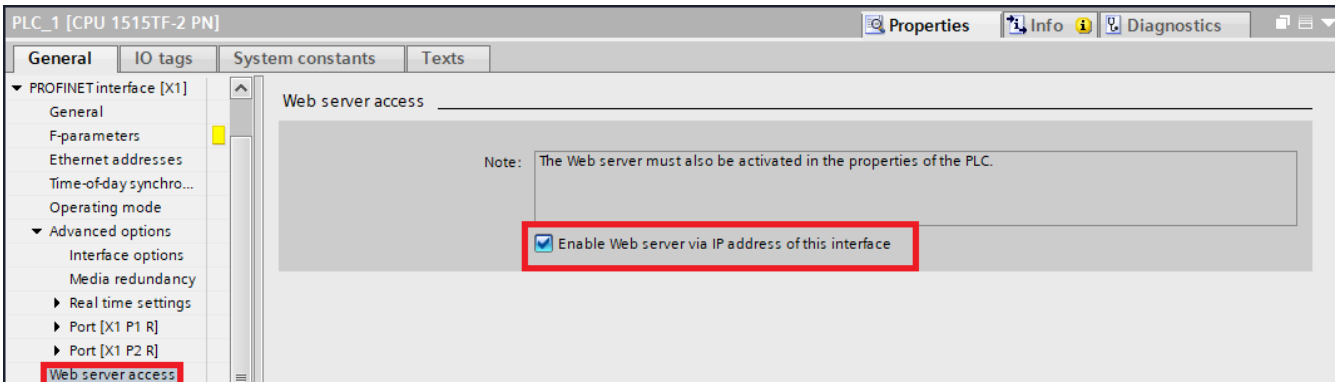


Figure 3.4: Indication of how to enable the web server

Chapter 4

Running the program

4.1 Running on physical hardware

After the steps discussed in the following subsections are completed, the program should be able to be downloaded and run by the PLC.

4.1.1 Changing the PLC type

This program was initially developed using a demo case, which had a SIMATIC S7-1500T, CPU 1515TF-2 PN. If one wants to implement this program on another type of PLC, the device has to be changed. This can be done by right clicking on the PLC, and clicking on *"Change device"*, as indicated in figure 4.1. A pop-up window will appear, you can choose the proper type of PLC there.

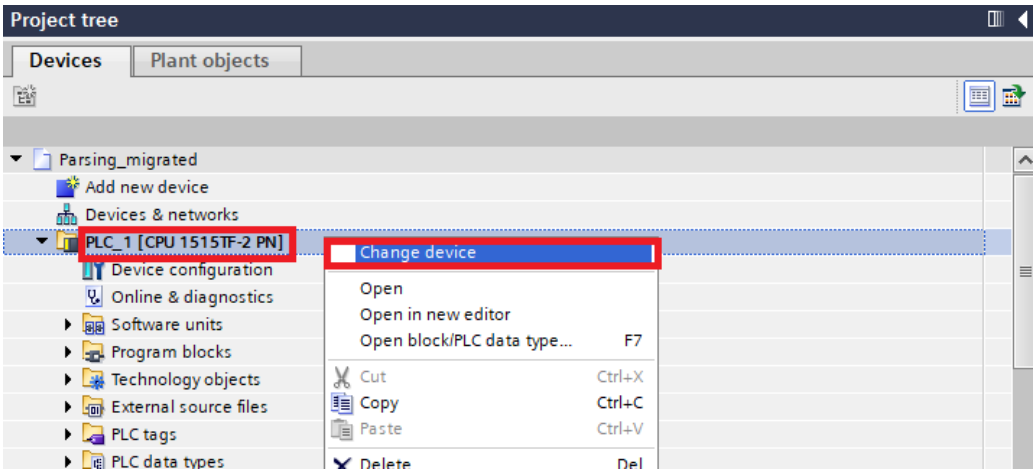


Figure 4.1: Indication of how to change the PLC type

4.1.2 Ethernet addresses

To run the program on the new PLC, one must first check if the Ethernet address of the PLC and the computer where TIA is opened on are in the same range. If this is not the case, make sure to change this address such that they can properly communicate. In figure 4.2, this is indicated.

When doing so, the ethernet address of the HMI has to be changed too, also in the same range. This is done in the same manner, indicated in figure 4.3.

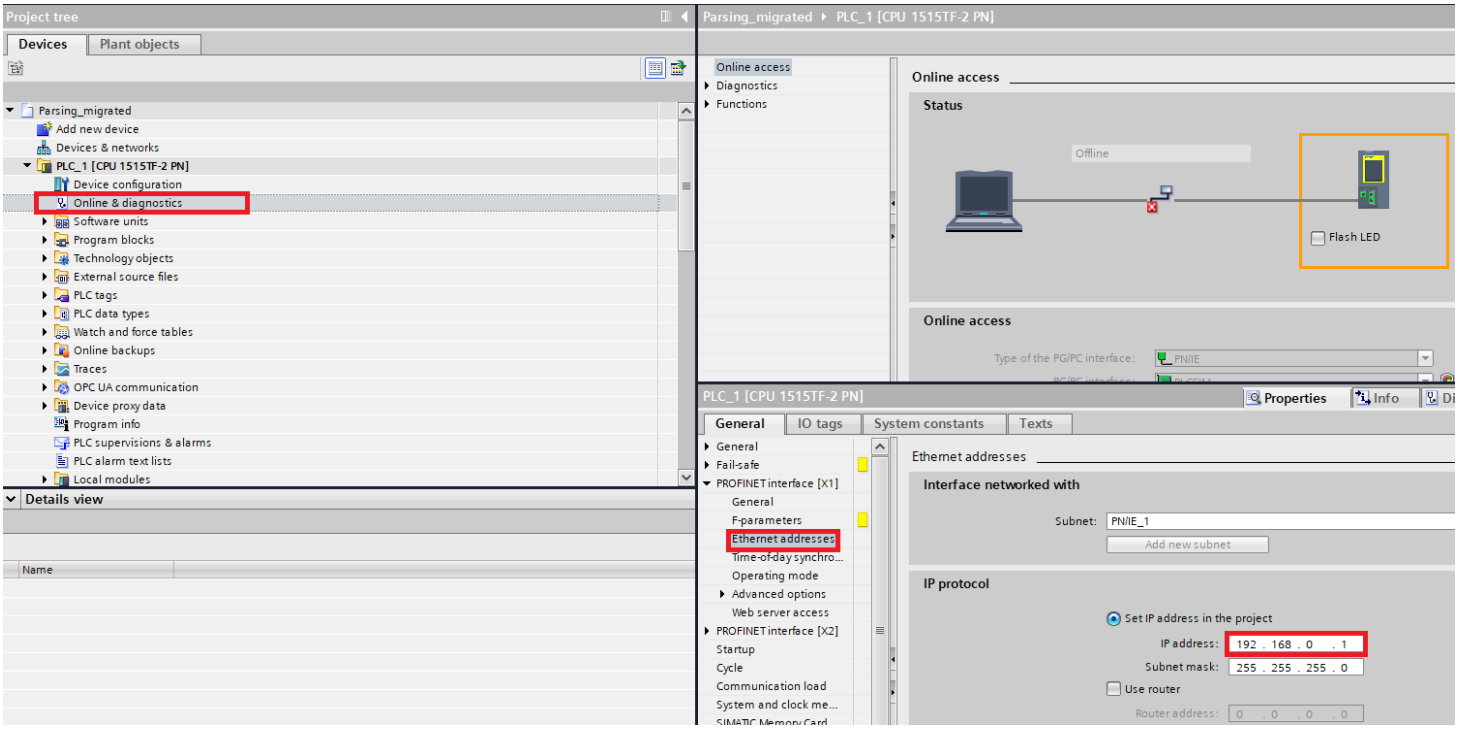


Figure 4.2: Indication of how to change the ethernet address of a PLC

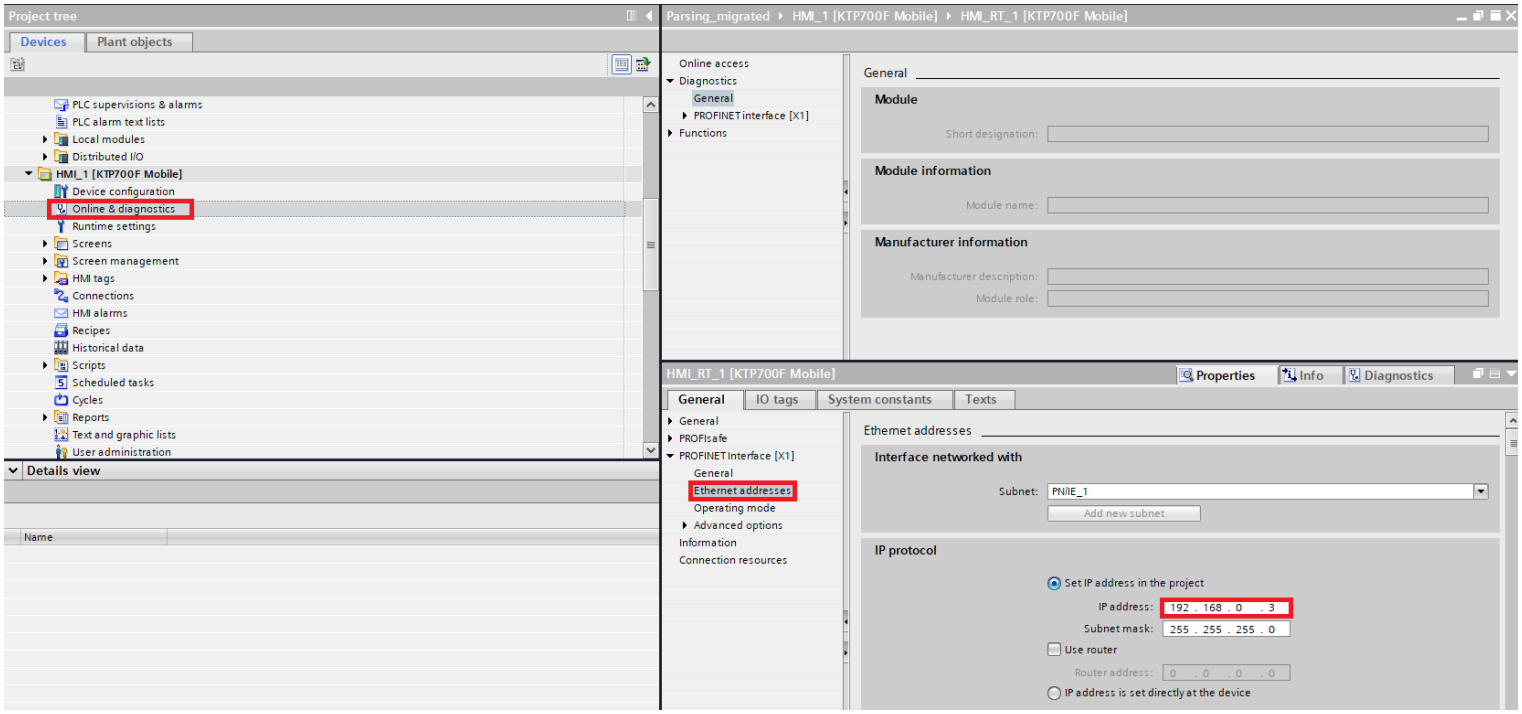


Figure 4.3: Indication of how to change the ethernet address of a HMI

4.2 Running on PLCSIM Advanced

After the steps discussed in the following subsections are completed, the program should be able to be downloaded and run by the PLC instance.

4.2.1 Ethernet adresses

To run the program on a PLCSim Advanced instance, one must first check if the ethernet address of the PLC instance, the HMI and the computer where TIA is opened on are in the same range. If this is not the case, make sure to change this address such that they can properly communicate. Instructions on how to do this are found in subsection 4.1.2.

4.2.2 Turning on simulation support

The next issue to be solved is turning on simulation support in the project properties. Without this, the PLCSIM instance cannot simulate write-protected blocks such as library blocks, which were used in this program. Firstly, right-click on the project name and click on *"Properties..."*.

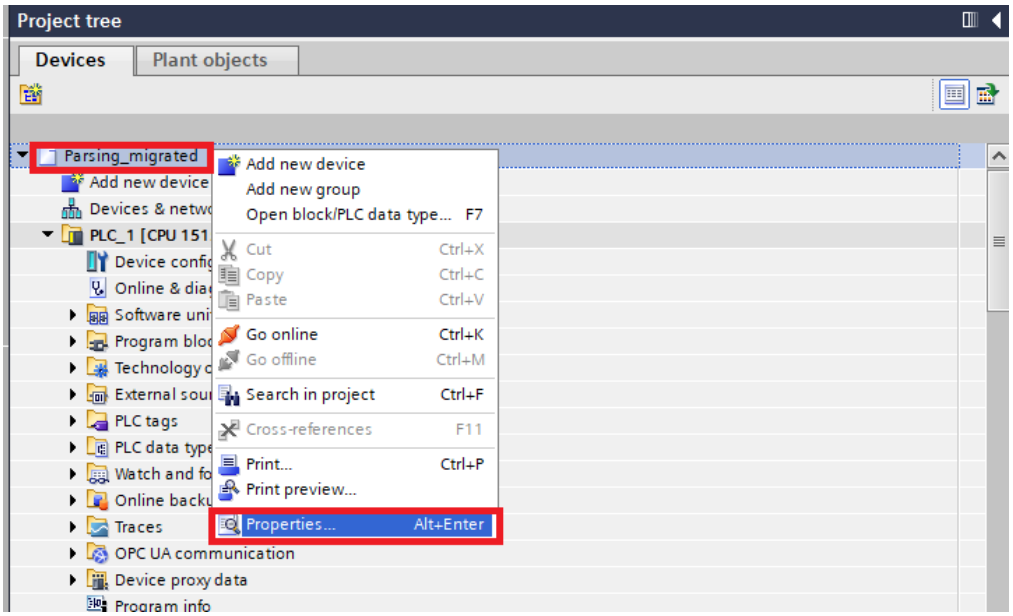


Figure 4.4: Right-click on project name

Then, a new window will pop up. Go to the *"Protection"* tab and turn on the *"Support simulation during block compilation"* option, as shown in figure 4.5.

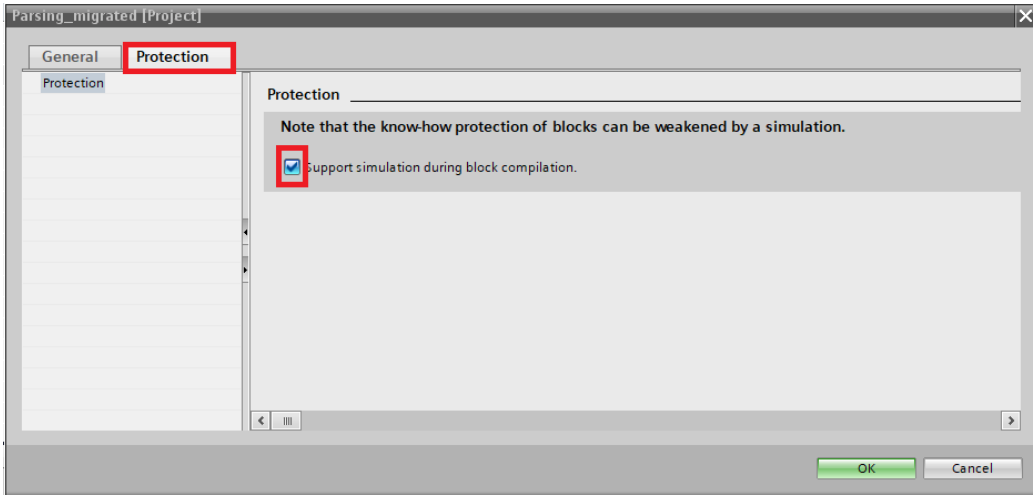


Figure 4.5: Turning on support for simulation

4.2.3 Getting G-code files in PLCSIM memory

To get G-code files into the PLCSIM Advanced memory, click open the PLCSIM application and double click on *Virtual SIMATIC Memory Card*, as indicated on figure 4.6.

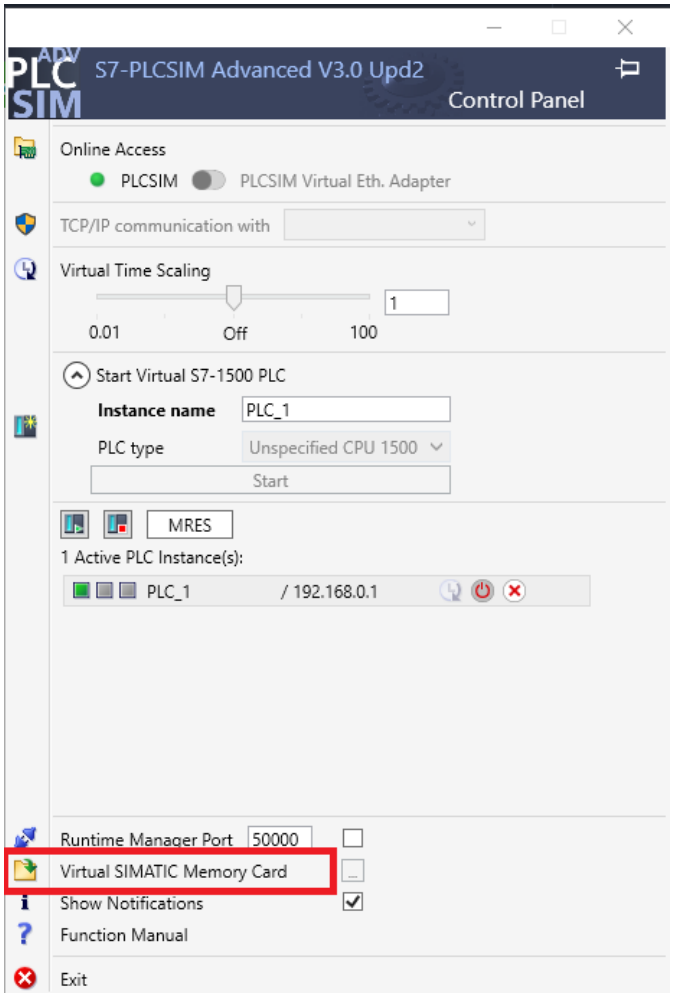


Figure 4.6: Opening the *Virtual SIMATIC Memory Card*

Then, a Windows Explorer window will pop up. Select the correct PLC and click open the folder. Then, click on *SIMATIC_MC*. The only thing left to do is to create the folder *UserFiles* if it is not already there and to copy the *.nc* files in this folder. The folder structure is shown in image 4.7

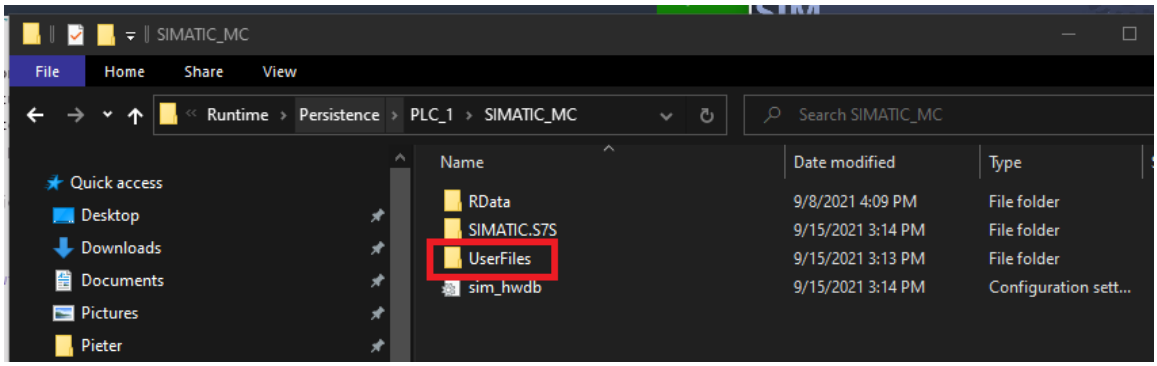


Figure 4.7: Folder structure of the *Virtual SIMATIC Memory Card*

After doing the previous steps, put the PLC in STOP mode and then again in RUN. The G-code files should be inside of the PLCSIM memory now and be ready for use.

Bibliography

[1] Siemens Industry Online Support. *SIMATIC S7-1500T Kinematics Control manual*. Aug. 2021. URL: https://cache.industry.siemens.com/dl/files/891/109755891/att_1017999/v1/109755891_LKinCtrl_DOC_v2_3_0_en.pdf (visited on 09/23/2021).

[2] Siemens Industry Online Support. *SIMATIC S7-1500T Kinematics Language manual*. Jan. 2021. URL: https://cache.industry.siemens.com/dl/files/009/109767009/att_1062345/v1/109767009_LKinLang_DOC_V1_3_1_en.pdf (visited on 09/23/2021).