

Urs Kafader

Motion Control for Newbies.

Featuring maxon EPOS2 P.



maxon motor
driven by precision



First Edition 2014

© 2014, maxon academy, Sachseln

This work is protected by copyright. All rights reserved, including but not limited to the rights to translation into foreign languages, reproduction, storage on electronic media, reprinting and public presentation. The use of proprietary names, common names etc. in this work does not mean that these names are not protected within the meaning of trademark law. All the information in this work, including but not limited to numerical data, applications, quantitative data etc. as well as advice and recommendations has been carefully researched, although the accuracy of such information and the total absence of typographical errors cannot be guaranteed. The accuracy of the information provided must be verified by the user in each individual case. The author, the publisher and/or their agents may not be held liable for bodily injury or pecuniary or property damage.

Version 1.2, February 2014

Motion Control for Newbies

Featuring maxon EPOS2 P

Intention and approach

The basic approach of this textbook, like many, is a practical and experimental one; however, it is reversed from most. Instead of first explaining the theory of motion control and then applying it to specific examples, here we will start with hands-on experimenting on a real maxon *EPOS2 P* positioning control system by means of the *EPOS Studio* software and explain all the relevant motion control principles/features as they appear on the journey. Therefore, the text contains mainly the exercises and practical work to do. Background information can be found in the colored boxes.

This is primarily a textbook about motion control and not about programming a PLC. Therefore, the programming part should be considered as an introduction on how to program the motion control aspects. It is not a full PLC programming course.

Motion control is mechatronics. It is the combination of mechanics and electronics, of actuators (motors) and sensors all controlled by software. In this textbook, we identify and define the role, behavior and mutual interaction of the different elements of a motion control system. However, we need to know the basic internal construction and working principles of the elements only to a limited extent, but rather look at them as a black box. Given a certain input, what comes out of the box? How is this output generated and which parameter can be used to influence the output? For instance, we will not explain how feed forward control must be implemented, but we will explain the basic idea of it and how the parameters will look during axis tuning.

As a system to evaluate these things in a practical hands-on manner, we use the *EPOS2 P Starter Kit* and the maxon *EPOS Studio* software.






Table of Content

Motion Control for Newbies	3
Intention and approach.....	3
Boxes 6	
Part 1: Preparation	7
1 Setting-up the EPOS2 P Starter Kit	7
1.1 Installation of EPOS Studio software	8
1.2 Connecting the cables.....	10
1.3 What's in the black box?	11
1.4 Starting the EPOS Studio.....	14
2 Preparing the motion control system.....	19
2.1 Startup Wizard.....	19
2.2 Tuning.....	28
Part 2: The Motion Controller	35
3 Exploring the Motion Controller	36
3.1 Profile Position Mode.....	36
3.2 Homing	44
3.3 Position Mode.....	46
3.4 Profile Velocity Mode.....	48
3.5 Velocity Mode	51
3.6 Current Mode.....	52
4 Using the I/O Monitor tool	53
4.1 Outputs	53
4.2 Inputs.....	54
5 Alternative Operating Modes (optional).....	57
5.1 Analog set value	57
5.2 Master Encoder Mode: Electronic gearhead.....	59
5.3 Step Direction Mode	59
Intermezzo: CAN and CANopen.....	61
6 An introduction to CANopen and CAN.....	61
6.1 CAN	62
6.2 CANopen	62
6.3 CANopen device profile.....	63
6.4 EPOS2 [internal] Object Dictionary tool	67
6.5 EPOS2 P Object Dictionary.....	70
6.6 Parameter Up- and Download.....	70
6.7 Communication.....	71

Part 3: The PLC (Programmable Logic Controller)	73
7 Starting the programming tool	74
7.1 Sample project: <i>Simple Motion Sequence</i>	75
7.2 The project files	79
8 My first program: AxisMotion	88
8.1 Preparation work	90
8.2 Programming the basic steps.....	94
8.3 Programming more motion.....	106
9 Homing and IOs	110
9.1 Configuring the IOs	110
9.2 Homing with limit switch	112
9.3 Setting outputs according to the axis state	113
9.4 Reading inputs	115
10 Self-made Function Blocks	116
10.1 Creating a Function Block	116
10.2 Programming in Function Block Diagram (FBD)	117
10.3 Using self-made Function Blocks.....	120
Part 4: Appendices, References and Index	121
11 Appendices	121
11.1 Motor and encoder data sheets	121
12 References, Glossary	124
12.1 List of Figures	124
12.2 List of Boxes	126
12.3 Literature	128
12.4 Index.....	129

Boxes

The color-coded boxes contain additional information about motion control and programming in general and about the hardware and software at hand.

 <p>Motion Control</p>	<p>Motion Control Background Gives general information and knowledge on motion control concepts.</p>
 <p>IEC 61131</p>	<p>IEC Background Gives general information and knowledge on <i>IEC 61131-3</i> PLC programming.</p>
 <p>EPOS</p>	<p>EPOS Info Gives additional short input for better understanding of special features and behavior of the <i>EPOS2 P</i> system.</p>
 <p>OpenPCS</p>	<p>OpenPCS Info Gives additional input of special features and behavior of the <i>OpenPCS</i> software.</p>
 <p>Best Practice</p>	<p>Best Practice Gives tips and hints how to use the <i>EPOS2 P</i> system, the <i>EPOS Studio</i> and <i>OpenPCS</i> software in the most efficient way.</p>

Part 1: Preparation

These first two chapters serve to get to know the system, and to prepare the workbench.

In chapter 1, we will provide an overview of the hardware, go over the installation of the *EPOS Studio* software and briefly explain the main elements of the motion control system.

In chapter 2 the motion control system is prepared for the exercises that will follow. The properties of the motor and encoder have to be defined and the control loop tuned.

1 Setting-up the EPOS2 P Starter Kit



Figure 1: The content of the EPOS2 P Starter Kit.

In the Starter Kit box you will find a board with a “Black Box”, a motor with an encoder, and a printed circuit board with switches, LEDs, and potentiometer on it. You will also find a DC power supply and the necessary cables.

1.1 Installation of EPOS Studio software

First, prepare the computer. In the Starter Kit box, you find the *EPOS Positioning Controller* DVD with all the necessary software. Enter the DVD in your computer and install it on your computer. Just follow the installation wizard.

The installation contains all necessary information and tools (such as manuals, firmware, Windows DLLs, drivers) required for installation and operation of the *EPOS2 P Programmable Positioning Controller*. The most important tool is called *EPOS Studio*; it's the user interface that allows full access to all the features and parameters of the motion controller.

Installation

- Step 1 Insert *EPOS Positioning Controller* DVD in your computer drive. Autorun will commence automatically. If autorun should fail to start, find the installation file named *EPOS Positioning Controller.msi* on your explorer. Double click to start.
- Step 2 Follow the instructions during the installation program. Please read every instruction carefully. Indicate location of working directory when prompted.



Working directory

We recommend the following location as a working directory:

C:\Program Files\maxon motor ag

Note that the designation of the program directory may vary depending on the system language installed.

- Step 3 View the new shortcuts and icons in the start menu of your computer. The files have been copied to the menu *maxon motor ag*, where you can access the program, as well as the entire documentation set. Clicking the *EPOS Studio* shortcut on your desktop will launch the program.
- Step 4 If needed: Modify or remove the software. To change application features or to uninstall the software, start the installation program *EPOS Positioning Controller.msi* anew and follow the instructions given.



Latest *EPOS Studio*

After installation, verify that you are using *EPOS Studio 2.00* (Revision 2) or higher. Do so by clicking menu *Help*, then select menu *About EPOS Studio*. If that is not the case, download the latest version from the maxon website <http://www.maxonmotor.com>.

Component	Minimum Requirement
Operating System	Windows 8, Windows 7, Windows Vista, Windows XP SP3
Processor	Core2Duo 1.5 GHz
Drives	Hard disk drive, 1.5 GB available space DVD drive
Memory	1 GB RAM
Monitor	Screen resolution 1024 x 768 pixels at high color (16-Bit)
Web Browser	Internet Explorer IE 7.0

Table 1: Minimum System Requirements (See Release Notes.txt in the EPOS Studio installation folder)

1.2 Connecting the cables

The cabling is easy and basically mistake proof. First connect the motor (3 wires: red, black, white with a white connector) into the only white socket that fits. Then do the same with the Hall sensor cable; its socket is next to the motor lines. (Do not plug it into the socket marked *RS232*). The encoder flat ribbon cable should pose no problems either.

Next, connect the PCB to the *EPOS2 P* by means of the short cable with broad plugs. Again, it is straightforward; the plugs only fit in one socket. Then – and that is good engineering practice to do it this last - connect the power supply. Finally, connect the *USB* cable to one of the *USB* outputs on your computer and onto the *EPOS2 P Mini USB* plug.

Your setup should then look like as in Figure 2.

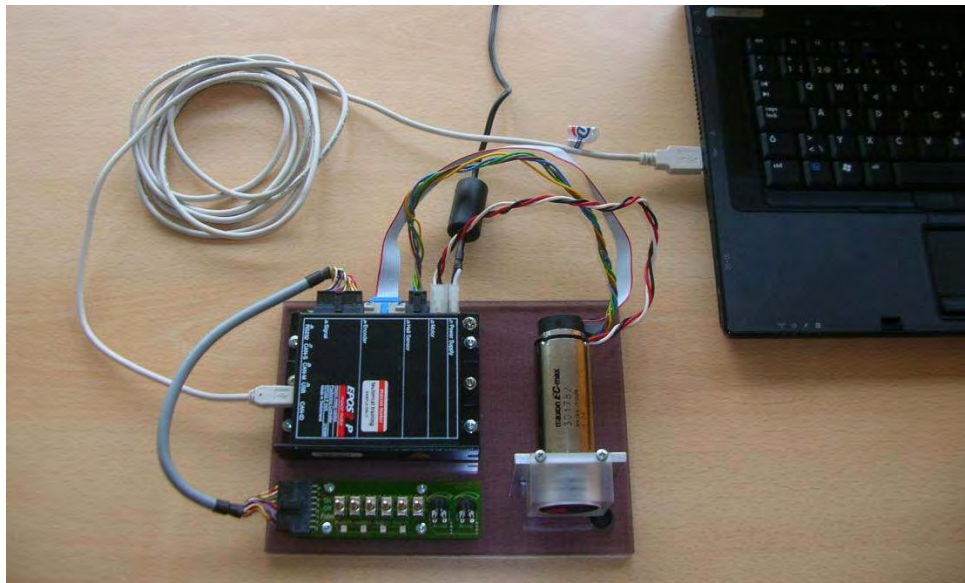


Figure 2: How to connect the EPOS2 P Starter Kit.



Finding the *USB* driver

When you first connect the *EPOS2 P* to the *USB* port your computer will announce that it has found new hardware and will ask for the corresponding drivers. Do not let your computer search automatically for the drivers. It is much faster if you tell it manually where to find them. (Sometimes it is necessary to run the *DriverPreInstaller.exe* first.)

The drivers are located in the same folder where you installed the *EPOS2 P* software: `.../maxon motor ag/ Driver Packages/ EPOS2 USB Driver` (see Figure 4). Follow the steps in the *EPOS USB Driver Installation.pdf* document. The procedure varies slightly depending on your computer's operating system.

1.3 What's in the black box?

The Black Box called *EPOS2 P* is what this textbook is all about. It is a maxon motion controller with a built-in programmable logic controller (PLC). EPOS is an acronym for **E**asy to use **P**ositioning **S**ystem, the **2** refers to the second-generation design, and the **P** denotes to the fact that it is programmable and contains a PLC. However, it's up to you to judge if it is really "EPOS" or just "POS", i.e. if it is easy to use or not.

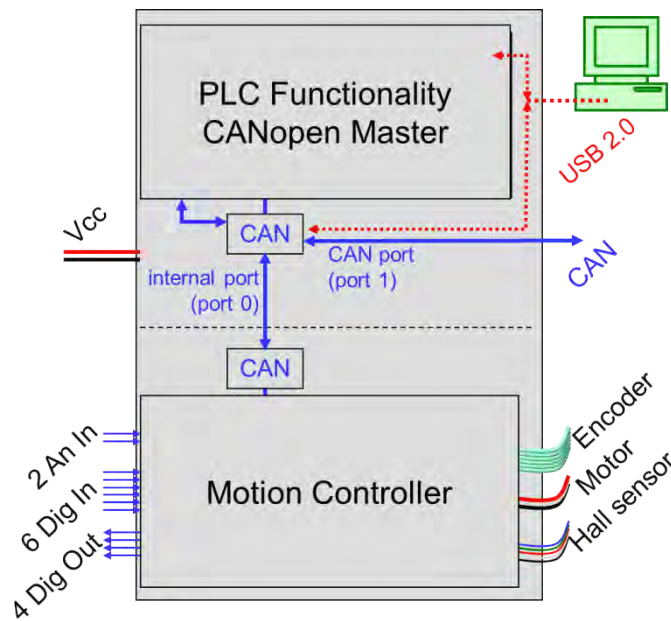


Figure 3: Schematic overview of the EPOS2 P with external connections and communication.

Essentially, the *EPOS2 P* contains two devices, a PLC and a motion controller, that are internally linked by a CAN bus. The *EPOS Studio* on your computer communicates with these two devices by means of a *USB2.0* serial connection. *USB2.0* is the default communication. Alternative ways for communication are serial *RS232* or *CAN* bus connection. Most probably, your computer does not have a *CAN* interface so we cannot use this bus.

The PLC part is the master of the device. It controls the process flow and all slave units in the network; one of the slave units being the built-in motion controller. The PLC executes an application specific program that we have a closer look at in Part 3 of this manual (starting with chapter 7).



Manuals and software documentation

The installation not only sets up the *EPOS Studio* software on your computer, but copies all the manuals and software you might need to program the device at hand or any other maxon EPOS device.

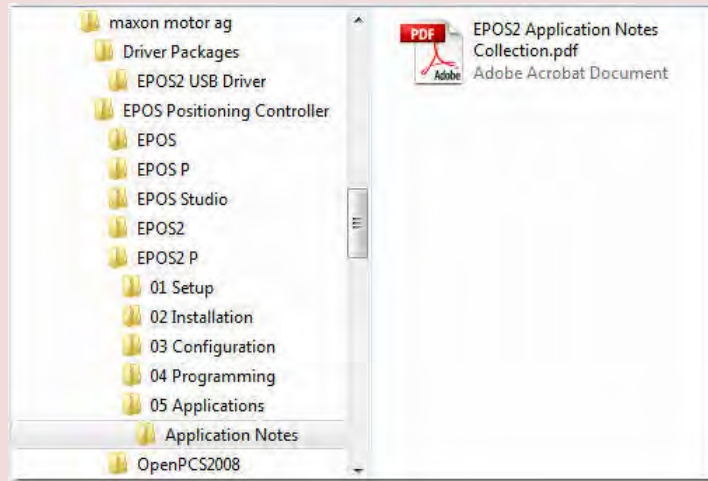


Figure 4: The file structure of the maxon EPOS software installation.

For a given EPOS product there are 5 subfolders with the documents as shown in *Figure 5*:

- In *01 Setup* you find the *Getting Started* manual which essentially contains the same set-up information as this first chapter. In addition, you can find wiring diagrams for other motor encoder combinations.
- In *02 Installation*, the hardware cabling information is listed.
- *03 Configuration* contains the actual and older *Firmware* files; that is the software running on the device.
- *04 Programming* contains important documents for the programmer: the *Firmware Specification* (see chapter 6.3). It is in there where all the properties and parameters are described in detail. The programmer finds other useful information, programming examples and libraries in the *Programming Reference* document (see chapters 7ff).
- *05 Application*: The *Application Notes* contain information on special application conditions, operation modes and situations.

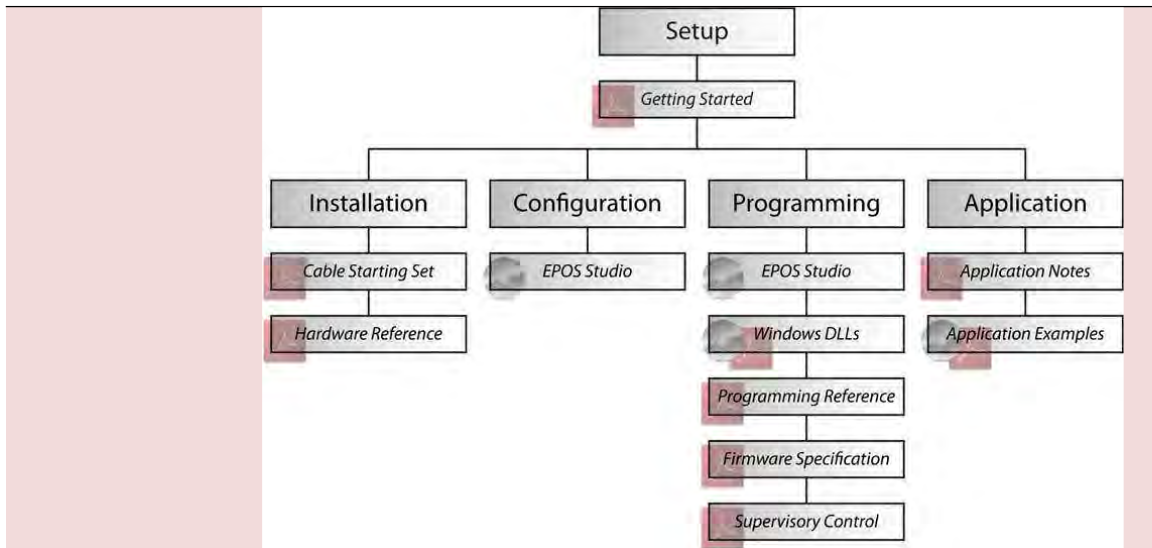


Figure 5: The document structure of the maxon EPOS2 P. Be aware that this document structure is not fully compatible with the file structure of Figure 4.

1.4 Starting the EPOS Studio

We are ready to begin. Start the *EPOS Studio* by clicking on the corresponding icon on your computer's desktop.

The *EPOS Studio* is the graphical user interface for all maxon EPOS products. In order to establish communication with the actual hardware, the correct EPOS project has to be selected. The *New Project Wizard* opens automatically. If not, click on the *New Project* icon on the menu bar.

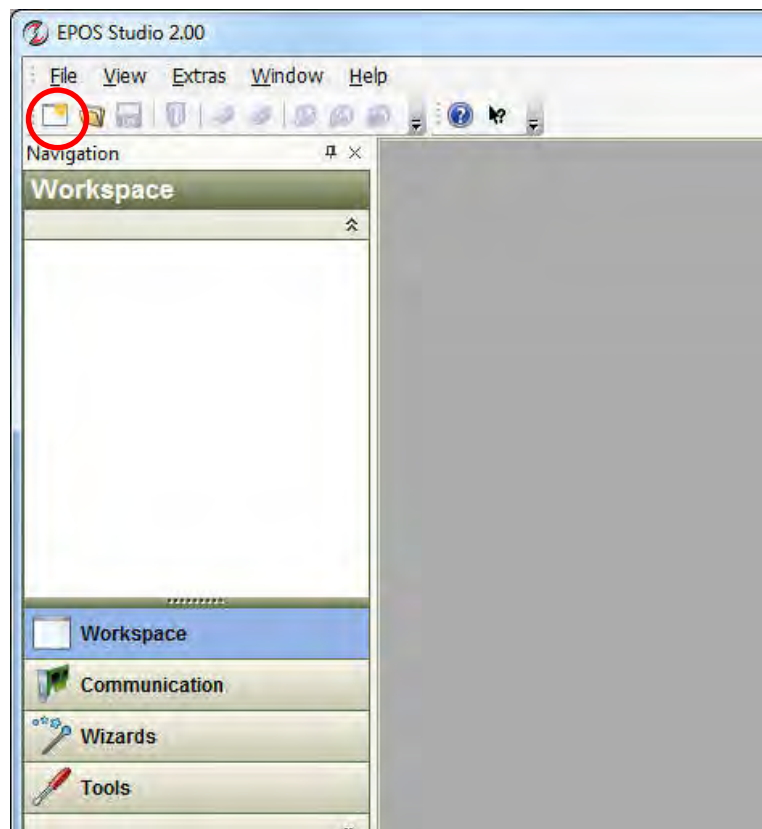


Figure 6: Where to find the New Project icon.



Projects in *EPOS Studio*

The *EPOS Studio* allows you to save the setup of your project and to open it again. The project in the *EPOS Studio* contains information about the hardware and communication channel used. If you have several EPOS linked to your computer in a network this is saved as well. Saving projects under a particular name is useful whenever you are not using a simple standard project as we have at the moment.

Step 1 Select *EPOS2 P Project* from the list. Click *Next*.

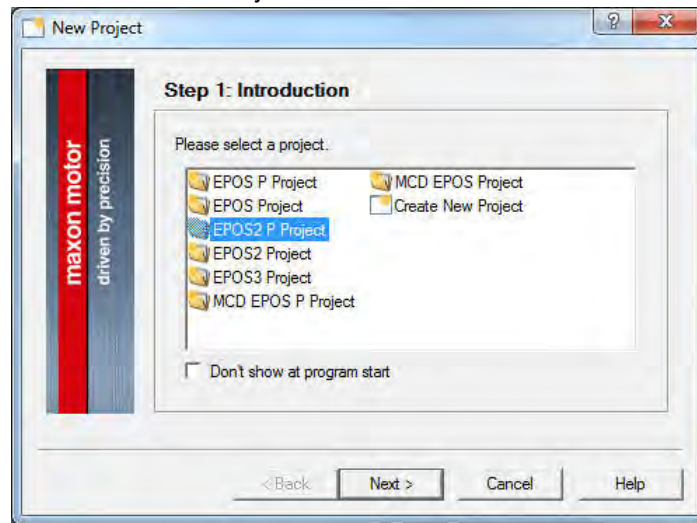


Figure 7: New Project Wizard, step 1.

Step 2 Now define a path and name for your project or use the suggested one. Click *Finish* to create a new project.

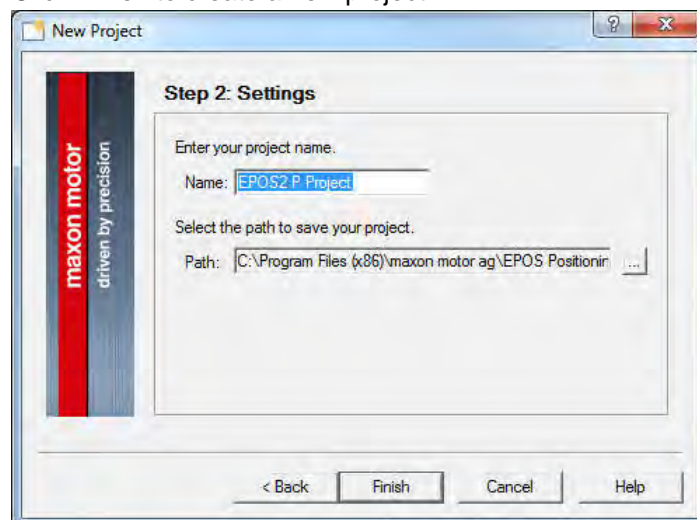


Figure 8: New Project Wizard, step 2.

If all goes well the *EPOS Studio* will establish communication with the *EPOS2 P*. This can be seen by the adjustment bars running from left to right in pop-up windows and indicating that the parameters of the *EPOS2 P* are being read.

The Workspace tab

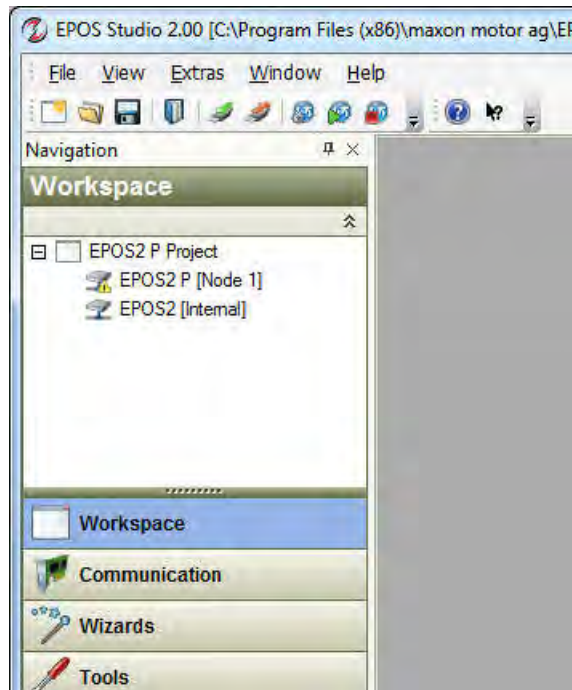


Figure 9: The EPOS Studio screen with the Workspace tab open.

The controller hardware of the project is displayed in the *Workspace* tab of the *Navigation Window* on the left of the screen. In our case, the project contains the *EPOS2 P [Node 1]*, which stands for the black box as a whole, and more specifically for the PLC inside, and the *EPOS2 [internal]*, which is the motion controller part.



Errors and warnings

Because there is no CAN communication established, the warnings *CanPassiveError on CAN-S Port* and *CanPassiveError on CAN-M Port* will appear. Clear these warnings with a right mouse click on the list and selecting *Clear All Entries*.

In case other errors or warnings appear, check the wiring and startup configuration (for details on errors and warnings see separate document *EPOS2 Firmware Specification*).

Status				
Type	Node	Code	Name	Description
Warning	EPOS2 P [Node 1]	0x8120	CAN Passive Error on CAN-M Port	CAN-M Port changed to state
Warning	EPOS2 P [Node 1]	0x8120	CAN Passive Error on CAN-S Port	CAN-S Port changed to state

Figure 10: Error and warning list in the Status window.

The Communication tab

The *Communication* tab shows the setup of your project from a communication point of view. Schematically, you can see that your computer (*Local host*) is connected by *USB2.0* to the *EPOS2 P [Node 1]*.

There is an internal *CAN* bus connection to the built-in motion controller, the *EPOS2 [internal]*. This network is called *CAN-I*, standing for *CAN internal network*. There is another *CAN* connection on the *EPOS2 P [Node 1]* called *CAN-S* (for *CAN slave network*); however there is no device connected to it.

The nice thing about the *EPOS Studio* is that it allows direct communication with any device in the network without any programming to be done. This is useful for setting up the system, training and any other exploration that you might want to do. In the rest of this textbook we use this communication first by addressing the internal *EPOS2* motion controller directly, and later for the programming the PLC of the *EPOS2 P*.

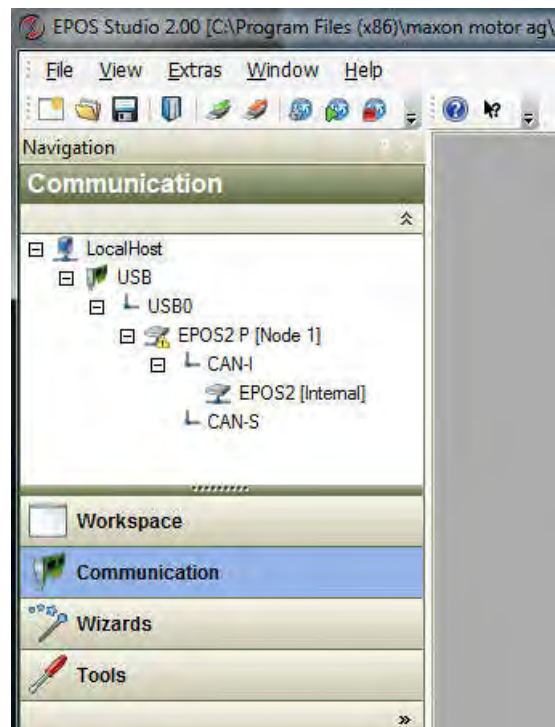


Figure 11: The Communication tab of the EPOS Studio screen.

The Wizards and Tools tabs

There is no need to look at the wizards and tools in detail at this time. Many of them will be used later on our journey and explained when they pop up.

Just one remark: Tools and wizards are assigned to the different hardware components in the project. That is why you always have to select the correct device from the drop down menu at the top. As you should know by now we have two separate devices at hand: the *EPOS2 P* with the PLC and the internal *EPOS2* motion controller.

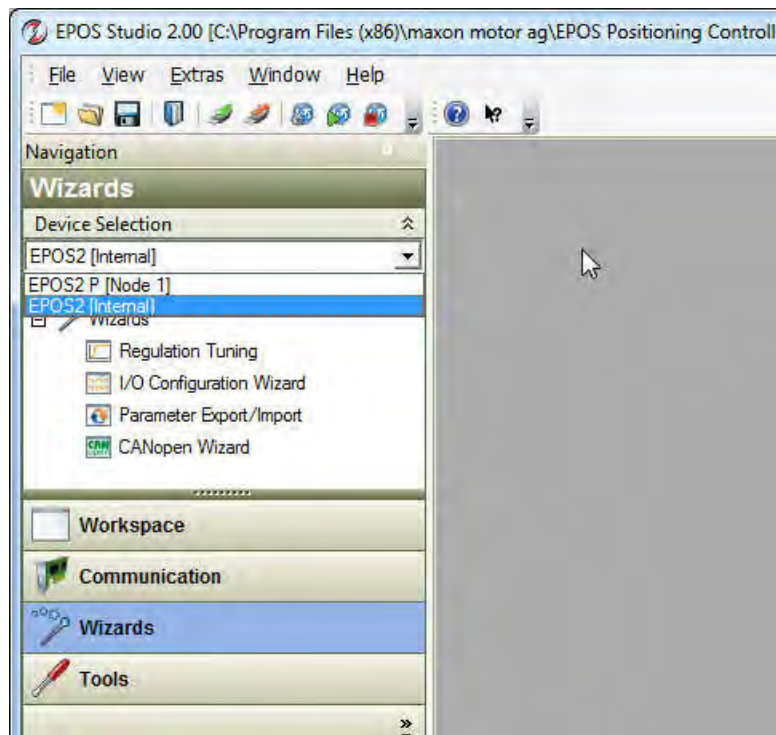


Figure 12: Selecting the device in the Wizards tab of the EPOS Studio screen.

2 Preparing the motion control system

2.1 Startup Wizard

An *EPOS2 P* can control many different drive configurations with a multitude of motors and possible sensors and – as we have learned earlier – with several possibilities of communication. Therefore, the first thing to do is to define the actual setup. For this purpose, we run the *Startup Wizard* that can be found in the *Wizards* tab of the *EPOS2 P*. Just double click on the corresponding icon.

Fortunately, we have quite a simple setup and only a few parameters vary from the default settings. There are not too many options to choose from. However, if you have a more complex system (e.g. a Dual Loop configuration with feedback sensors on the motor and on the load) work carefully through the *Startup Wizard*.

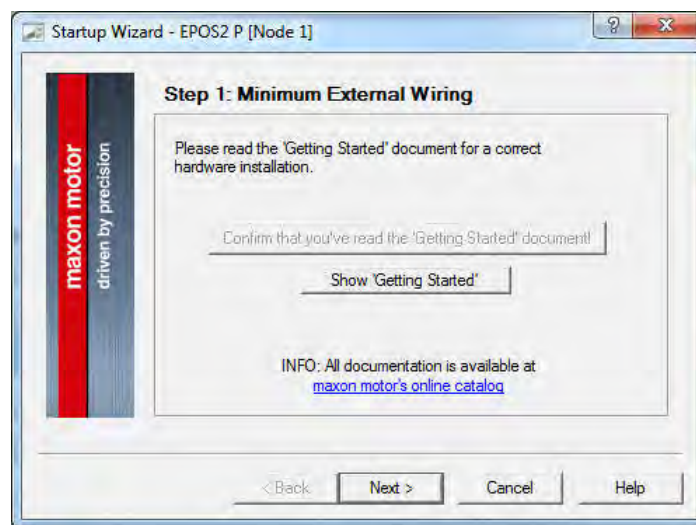


Figure 13: Startup Wizard, step 1.

- Step 1 First, maxon wants to make sure that you have proper documentation at hand and that you know what you are doing. That's why you have to *Confirm that you've read the "Getting Started" document*. Don't worry if you haven't read it. Actually, you are doing it; these first chapters contain the same information as the *Getting Started*. So relax, confirm and click on *Next*.
- Step 2 The second step is to indicate what kind of communication you are going to use. USB is set as default communication. So, there is nothing to be done but to click on *Next*. If you are wondering what other options you have; go and explore them. Only make sure that the final setting corresponds to the next figure.

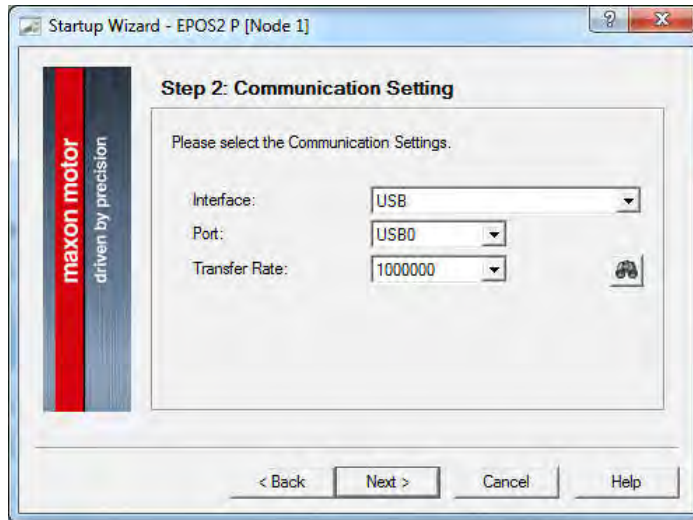


Figure 14: Startup Wizard, step 2.

- Step 3 The third step is to define the type of motor used. The motion controller can be used with brushed DC motors (maxon DC motor) or with brushless DC motors (maxon EC motor). Since the motor at hand is a maxon EC motor select the second option. Then click on *Next*.
- Step 4 For brushless motors you have to define the commutation type you would like to use. You will get the best performance with sinusoidal commutation, which requires an encoder and Hall sensors on the motor. The motor at hand has all these perfect features. Thus, let us select this option for the commutation and click on *Next*.

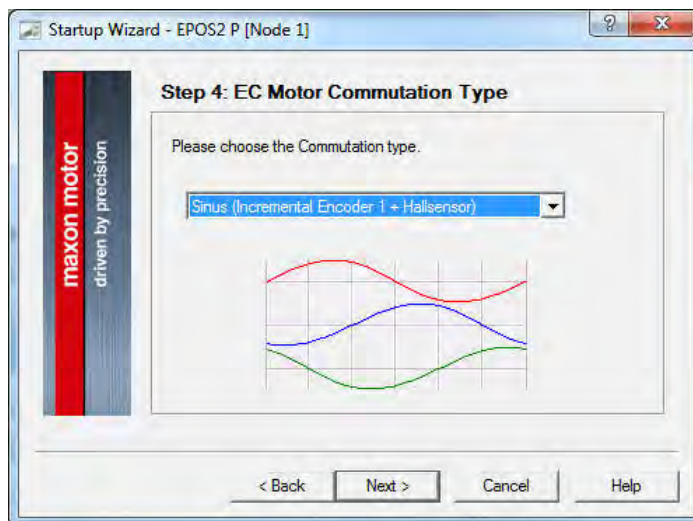


Figure 15: Startup Wizard, step 4.

maxon EC motor types (examples)	Number of pole pairs	Max. speed with block commutation	Max. speed with sinusoidal commutation
maxon EC motor, EC-max	1	100'000 rpm	25'000 rpm
EC-4pole	2	50'000 rpm	12'500 rpm
EC 20 flat, EC 32 flat	4	25'000 rpm	6'250 rpm
EC-i 40, EC 60 flat	7	14'280 rpm	3'570 rpm
EC 45 flat	8	12'500 rpm	3'125 rpm
EC 90 flat	12	8'330 rpm	2'080 rpm

Table 2: Maximum speed of EPOS systems for brushless DC motors as a function of the number of pole pairs and type of commutation. Note that some of the motors have lower maximum speed specification than what is possible with block commutation.

Step 5 In step 5 we have to define what type of feedback sensor is used to close the position or speed control loop. The standard device for maxon EPOS controllers is an incremental digital encoder. Actually, we can use the same encoder for measuring motor position and speed as we have defined for commutation purposes in the previous step. So again, just click on *Next*.

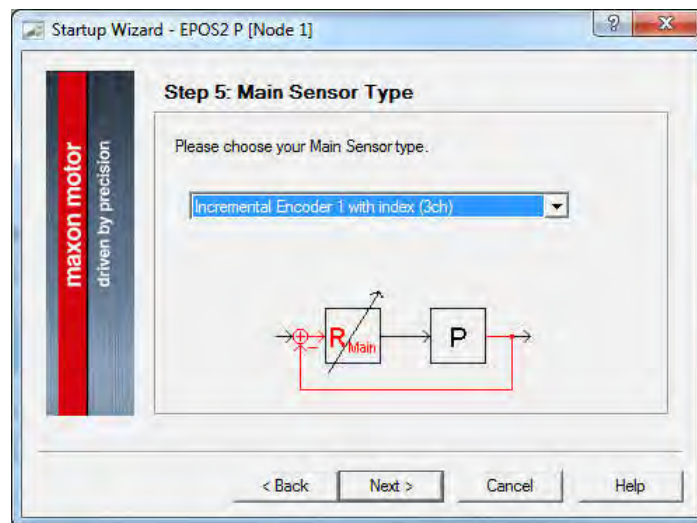


Figure 16: Startup Wizard, step 5.

Step 6 Next we have to enter some parameters needed to avoid unsafe motor operation. These parameters can be found in the data sheet of the motor (see chapter 11.1, motor part no 272763). We specify the *Maximum permissible speed* of the motor (15'000 rpm for the actual motor, or any smaller value if our application requires some lower speed limit for other reasons). *Nominal current* (2660 mA) and *Thermal time constant of the winding* (2.7 s) are needed to protect the motor from overheating. The maximum current is set to twice the nominal current automatically. (It can be changed in the *Object Dictionary*; see chapter 6.4) The *Number of pole pairs* - for our motor its value is 1 - is needed for correct commutation and speed measurement.

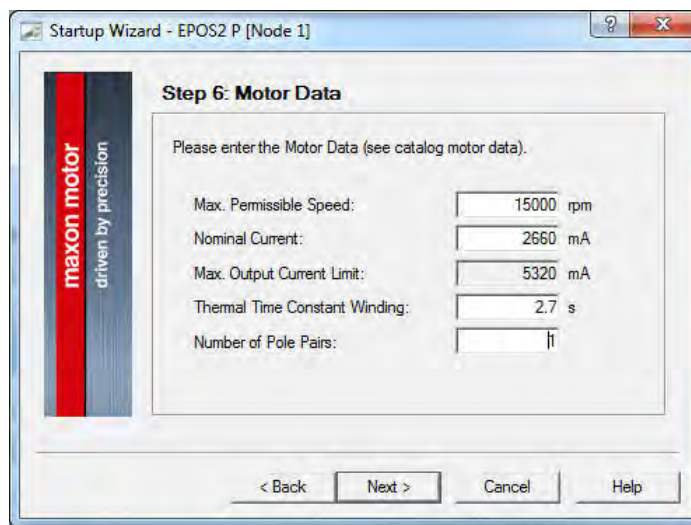


Figure 17: Startup Wizard, step 6.

Step 7 Upon *Next* we define the properties of the encoder. It is 500 pulses per turn. And *Next* again.

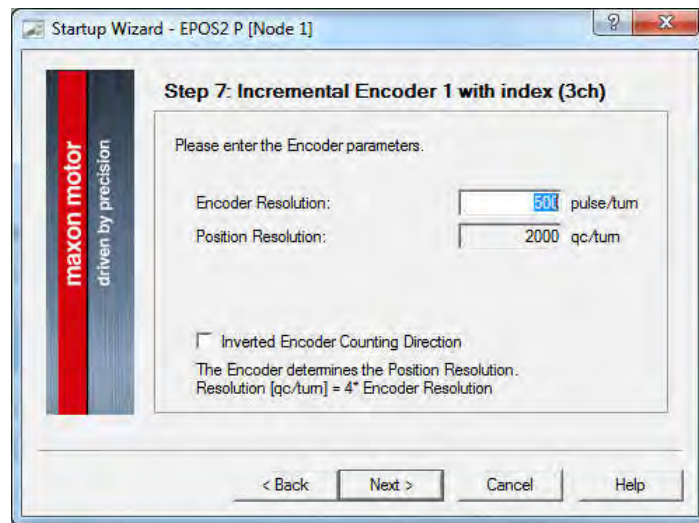


Figure 18: Startup Wizard, step 7.

Step 8 We leave the *Maximum following error* at the suggested 2000 qc (= quadrature counts, see MC background information on *Position and speed evaluation with incremental encoder*) and proceed to the last window which gives a summary of our settings. *Finish* this dialog and prompt the question if these parameters and settings should be saved with *Yes*.

This is what we mean by *Easy to use*: with a few clicks and parameters, the system is set up.



Commutation with brushed and brushless motors

The term *commutation* describes the way the power stage of the controller applies and switches the supply voltage between the different winding segments.

On brushed motors – e.g. maxon DC motor – the commutation is done in the motor by means of graphite or precious metal brushes. To make the motor rotate there is no other work to be done in the controller but to apply the voltage.

On brushed DC motors there is no additional feedback device needed for proper commutation.

On brushless DC motors (BLDC) – e.g. maxon EC motor - there are three winding segments (phases). The power stage of the controller needs to supply the voltage correctly to these three phases in accordance with the actual rotor position. The rotor position is monitored by 3 digital Hall sensors that allow the simplest type of commutation for BLDC motors, called *block commutation*. In short and without going into details, the Hall sensors are necessary to actuate the BLDC motor properly. However, Hall sensors only provide a low-level resolution feedback and block commutation has its limitation when it comes to smooth operation at low speeds.

A more sophisticated way of doing commutation is called *sinusoidal commutation*. It results in smooth motor operation even at very low speeds and gives a slightly higher motor efficiency. However, it requires the additional high-resolution feedback of an encoder. However, this is usually available on servo systems with position or speed control anyway.

Note: On brushless DC motors, feedback is needed for two separate tasks: for commutation – i.e. for motor operation – and for speed or position control.

The *EPOS2* motion controllers allow block commutation for motor speeds up to 100'000 rpm, while sinusoidal commutation has a maximum speed of 25'000 rpm. These values apply for BLDC motors with 1 magnetic pole pair. Higher numbers of pole pairs reduce the maximum speed accordingly. An EC-4pole motor with 2 pole pairs can only be operated at half these speed values. Flat motors can have even more pole pairs and – correspondingly – a lower maximum speed limit.



Motor and encoder parameters of the unit at hand

The designation of the motor used is *EC-max 30*, with order number 272763. Its exact specification can be found on the [maxon website](#) or in the [Appendix](#) (chapter 11). Important for us is the fact, that it is a brushless motor (EC) with Hall sensors.

This motor has a maximum speed limit of 15 000 rpm which is much higher than what we can achieve with the given voltage of the power supply (19 VDC). Typically, the motor is able to reach speeds of up to approximately 6500 rpm at 19 V supply (speed constant of motor times motor voltage = $393 \text{ rpm/V} * 0.9 * 19\text{V}$). Thus, there is no risk of overspeeding the motor.

The nominal current is 2.66 A. The motor can draw that current continuously without overheating. For a few seconds it can support a much higher current, possibly up to 10 A. But again, we won't be able to get as much current out of our power supply, which limits below 5 A. There is no risk of overheating the motor either.

The encoder type is *MR encoder*, order number 225778. It has 500 counts per turn on each of the two channels. This results in 2000 signal edges (states) per turn of the motor shaft which are called quad counts (qc). The qc are the internal position units. Hence, we have a nominal resolution of 1/2000 of a turn or 0.18° .

In addition, the encoder at hand has an Index channel that can be used for precise homing and is needed to double-check the Hall sensor signals.



Position and speed evaluation with incremental encoder

Incremental digital encoders are the most common sensor type for measuring speed and position in micro-drives. They can be found in various designs and with various working principles, but all of them basically subdivide one revolution into a number of steps (increments), and send signal pulses to the controller each time an increment is detected. Incremental encoders generally supply square-wave signals in two channels, A and B, which are offset by one quarter signal length (or 90 electrical degrees).

The **characteristic parameter** of the encoder is the number of square pulses N per revolution, given in cpt (counts per turn). By counting the number of state transitions in both channels, the real resolution is four times higher. These states are called quadrature counts or quad counts (qc) and are used as the position unit in EPOS systems.

Note: When talking about the resolution of incremental encoders make sure whether the counts are meant to be before or after quadrature.

The higher the resolution, the more accurately the position can be measured, and the more accurately the speed can be derived from the change in position as a function of time.

The **direction of rotation** follows from the sequence of the signal pulses of channel A and B: Channel A leads in one direction, channel B leads in the other direction.

In addition, the encoder can have an **index channel**. That's one narrow pulse per turn. The index can be used for getting a very precise position reference (see Homing, chapter 3.2). Some controllers for brushless motors need the index as an absolute rotor reference during sinusoidal commutation as well.

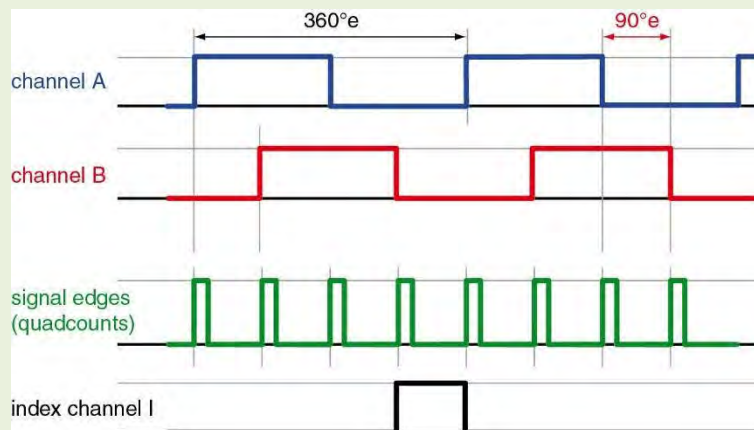


Figure 19: The signals of an incremental encoder.

Position evaluation is determined by counting the signal edges (increments) from a defined reference position (home position, see homing in chapter 3.2). The unit of positioning is quad counts (qc).

Speed evaluation is calculated from the position change per sampling interval. In our case, the natural unit of speed is quad counts per millisecond. Therefore, speed can only be expressed in steps of 1 qc/ms, which is usually translated in the more common, and practical unit rpm. The effect of speed quantization from the incremental encoder can be nicely seen on speed measurement diagrams (see chapter 3.4).

Acceleration and deceleration values are given in the practical unit rpm/s.



Encoder pulse count and control dynamics

A high encoder pulse count N not only results in a high position resolution, but also improves the control dynamics. Feedback information about a change in the actual position value is obtained more quickly and the control circuit can initiate corrective action that much faster. The controller can also be set up with more stiffness, i.e. with higher controller amplification, which reduces the risk of unstable oscillation.

With a low pulse count and high control parameter setting, the drive has more time to accelerate until a new position feedback signal, i.e. a new encoder pulse, is received. In such a case, the drive may overshoot the end position and produce a proportional correction value in the opposite direction. This again can result in an overshoot in the new direction. The motion becomes unstable and the drive oscillates.

2.2 Tuning

The next step for system preparation is tuning. The purpose is the same as tuning a sports car: We would like a quick, strong and optimized reaction of the drive system to any given motion commands. Again, we can use a wizard to make life easier.

Tuning has to do with the motion control part of our black box. Therefore, select the *EPOS2 [internal]* motion controller as the actual hardware device in the *Wizards* tab. Then start the *Regulation Tuning* wizard (double click).

Select the simpler option, which is *Auto Tuning*. And *Next* again.

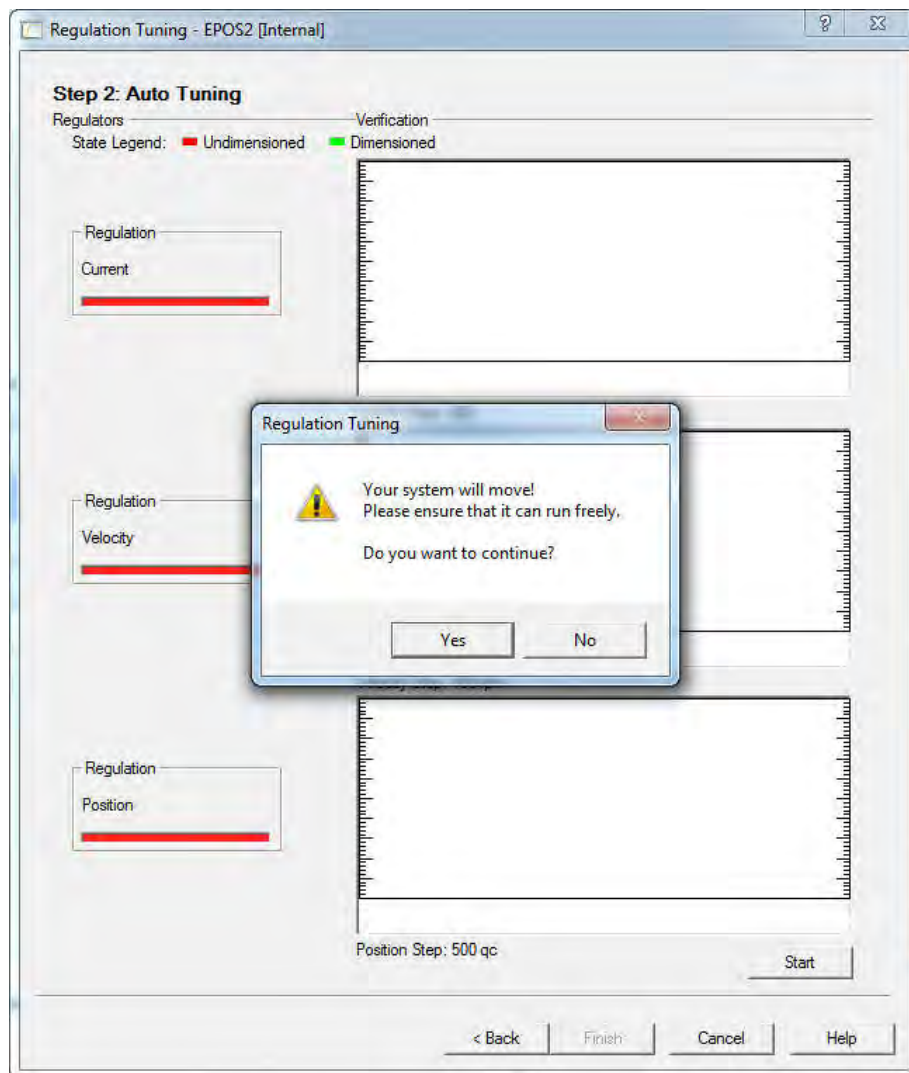


Figure 20: Regulation Tuning wizard, Auto Tuning screen.

Start the tuning and prompt the next message stating that your system will move with Yes. There is not much more you can and have to do. An automatic tuning is performed, starting with optimizing the response of the motor current control loop. The goal is that the current is applied as fast and precisely as possible whenever it is demanded from the higher control level, such as position or velocity control. A well-tuned current control loop is the prerequisite for dynamic system reaction.

The *Auto Tuning* feature starts oscillating the motor with increasing amplitude and at two different frequencies. This is the *Identification*. It serves to acquire information about the dynamic response of the system.

In a next tuning step, the *Parameterization*, the optimum feedback and feed forward tuning parameters are calculated, based on the results of the identification and on a general model of the drive system layout.

The *Verification* at the end of the *Auto Tuning* executes a motion, records the current, velocity and position reaction and displays them in the diagrams.



Diagram zoom

You can zoom into the diagrams by selecting the interesting region with the mouse. Right mouse click zooms out.

Save the newly found tuning parameters permanently when you exit the *Regulations Tuning* wizard. The parameters are applied to any future motion command. They are saved permanently and are functional, even after a restart of the system. This *Auto Tuning* process provides quite good tuning parameters for most applications with just one mouse click. There is no need for cumbersome trial and error or time-consuming evaluation of recorded motion response. Again, we think this is easy to use.

Note: Tuning should always be done on the fully established system with the original power supply and all the mechanics, inertias and friction in place. In the end, it's the full machine that we want to respond in an optimum way and not just the motor alone.



Expert Tuning and Manual Tuning

Selecting *Expert Tuning* in the start screen of the *Regulation Tuning* wizard allows more influence on the tuning process. But first you have to define the main control parameter of your system: Position, velocity or current.

At the top of the expert tuning dialog, you have access to the tuning parameter of each control loop. You can even change them manually if you are not satisfied with the tuning result of the wizard (manual tuning).

Next comes the *Identification* part, where you can set the amplitude of the oscillating movement. Once the identification has been executed, the system behavior and the effects of friction and mass inertias are known. Therefore, there is no need to repeat the identification when you just change the setting in the *Parameterization* step. And certainly, you do not want to do identification when you tune manually.

The way the actual tuning parameters are calculated during parameterization depends on the desired system behavior and your application. You might want a very stiff and fast reacting system and you don't care about overshooting the target position or speed. Or you set soft regulation stiffness with slower reaction and less overshoot.

Try different settings, observe the result in the *Verification* diagrams and compare the parameter values. For better resolution in the diagrams, adjust the *Max. Recording Time*. In our case 200ms or even less is perfect.

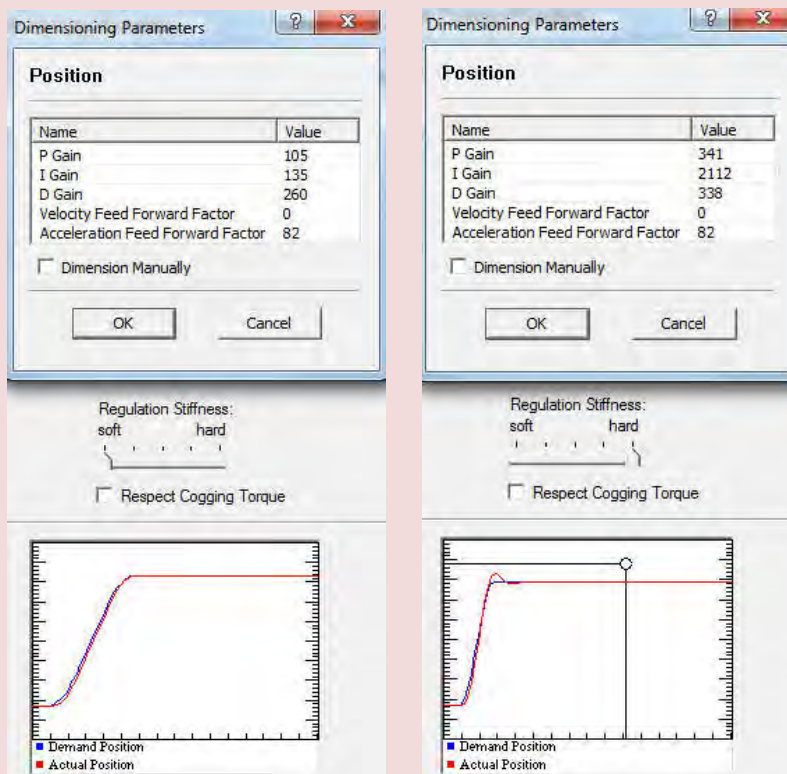


Figure 21: Regulation Tuning wizard. Comparison of soft (left) and hard (right) regulation stiffness: PID parameters (top) and system response (diagram on bottom) resulting from different regulation stiffness settings in expert tuning.



Feedback and Feed Forward

PID feedback amplification

PID stands for *Proportional*, *Integral* and *Derivative* control parameters. They describe how the error signal e (see Figure 22) is amplified in order to produce an appropriate correction. The goal is to reduce this error, i.e. the deviation between the set (or *demand*) value and the measured (or *actual*) value. Low values of control parameters will usually result in a sluggish control behavior. High values will lead to a stiffer control with the risk of overshoot and at too high an amplification, the system may start oscillating. This can easily be observed on our system by manually increasing the autotuning parameters by a factor of about 5.

On speed controllers, a closed-loop control circuit with a simple PI algorithm is usually implemented. With positioning systems, a derivative term is also necessary. The three terms may influence each other and understanding this interaction is of particular importance for the fine-tuning of a positioning system. For optimal system performance, the coefficients K_P , K_I and K_D have to be set depending on the specified motion and load inertia (Literature: Feinmess).

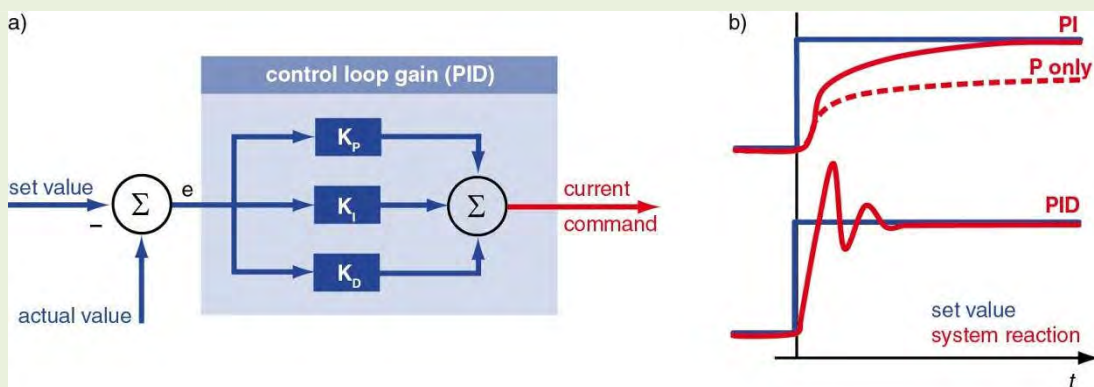


Figure 22: PID controller.

a) Schematic representation of a position or speed controller with the PID element as an amplifier for the error signal e .

b) System response to a set value change when a P, a PI and a PID element have been implemented.

Proportional controller (P): The error signal e (difference between actual and set value) is multiplied by a user-specified factor K_P and then transmitted as a new current command. An elevated K_P value accelerates error correction. If K_P is too large however, severe overshoot will occur. At even higher values of K_P the system may begin to oscillate, which in turn can result in instability if the damping in the system is insufficient. K_P cannot completely eliminate the error e , since the proportional correction value ($K_P \cdot e$) becomes smaller as the error e decreases. The result is a residual error that is particularly important in systems that require a high current simply to maintain motion (e.g. because of high friction).

Integral controller (I): Here, the error is summed up over a certain time, multiplied by a user-specified factor K_I and then added to the new current command. This eliminates the residual error of the P only amplification because a longer lasting deviation will provoke increasing corrective action.

However, there is an inherent disadvantage to this method particularly in cases where the error oscillates between positive and negative values. As past errors have a delayed effect, this can lead to positive feedback coupling and destabilization of the entire control circuit. Without appropriate damping, high K_I values can cause severe system oscillations.

Derivative controller (D): The D controller considers the change in the error, which is multiplied by a user-specified factor K_D and added to the current command. Sudden errors, such as set value jumps, can be very quickly corrected with this method.

Properly set, this type of control can improve control stability. It can be considered a type of electronic damping. Increasing the K_D value results in greater system stability, but too high a K_D will lead to oscillations.

Feed-forward

With the PID algorithms, corrective action only occurs if there is a deviation between the set and actual values. For positioning systems, this means that there always is – in fact, there has to be – a position error while in motion. This is called *following error*. The objective of the feed-forward control is to minimize this following error by taking into account the set value changes in advance. Energy is provided in an open-loop controller set-up to compensate friction and for the purpose of mass inertia acceleration.

Generally, there are two parameters available in feed-forward. They have to be determined for the specific application and motion task:

- *Speed feed-forward gain*: This component is multiplied by the demanded speed and compensates for speed-proportional friction.
- *Acceleration feed-forward correction*: This component is related to the mass inertia of the system and provides sufficient current to accelerate this inertia.

Incorporating the feed forward features reduces the average following error when accelerating and decelerating. By combining a feed-forward control and PID, the PID controller only has to correct the residual error remaining after feed-forward, thereby improving the system response and allowing very stiff control behavior.

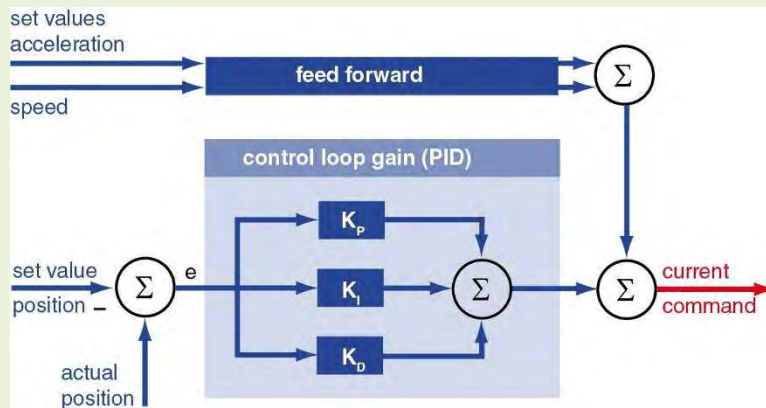


Figure 23: Schematic of feed-forward control. The feed-forward values are calculated from the required set value changes, i.e. acceleration and speed. Both values are added to the PID current command value.

Part 2: The Motion Controller

In Part 1 we have configured the system, tuned the motion controller and saved all the settings on the device. The *EPOS2 P* motion controller is now ready for further exploration.

- Chapter 3 investigates the motion controller, the *EPOS2 [internal]*, and uses the tools associated with it to study the different operating modes.
- Chapter 4 explores the inputs and outputs associated with the motion controller.
- Chapter 5 is optional. It shows some special additional operating modes that are not using on-line commanding.

In our setup, the *EPOS Studio* takes the role of the master. From the Studio, we send single motion commands (and others) and we can follow directly on the motor and in the Studio how they are executed. For the moment, there is no need to do any programming. Just play!



Master, slaves and on-line single commands

The *EPOS2* motion controllers are employed as slaves in a network. Slaves get their commands from a superior system, called *Master*.

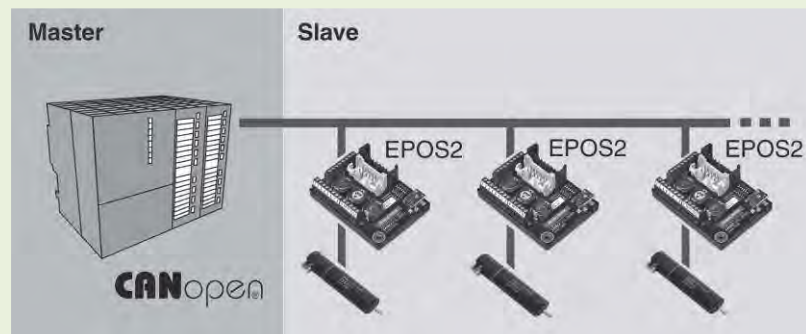


Figure 24: Master-Slave architecture with *EPOS2* slaves.

In the master runs the software that controls all the processes and slaves in the network. A master can be a PLC, a computer or a microprocessor. How the programming of the master (process control) can be done for a PLC is treated in Part 3 of this textbook (chapter 7 ff).

The master software used in this Part 2 is the *EPOS Studio*.

Important to note is that the slave *EPOS2* cannot store any command sequences; it is on-line commanded by single commands that are sent from the master and executed immediately one by one.

3 Exploring the Motion Controller

The tools for the exploration of the motion controller can be found in the *Tools* tab of the *EPOS2 [internal]*.

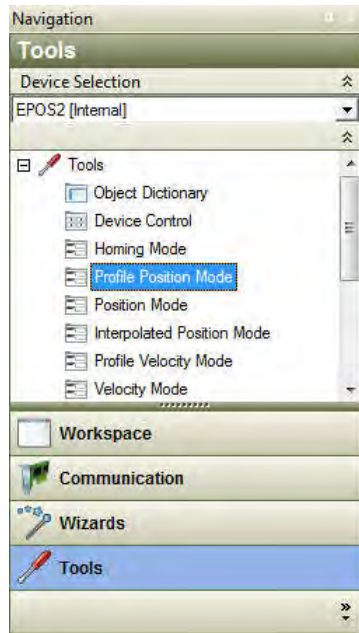


Figure 25: How to open Profile Position Mode. Select the Tools tab, then select the device EPOS2 [internal], and then double click on Profile Position Mode.

3.1 Profile Position Mode

Objectives	Executing a point-to-point movement. Monitoring the motion by the built-in <i>Data Recorder</i> .
------------	--

The standard positioning mode for most applications is the *Profile Position Mode*. Positioning is done by following a speed profile that is automatically created by the built-in *Path Generator*. The path generator has a clock rate of 1 ms, adapted to the 1 kHz sample rate of the position control loop. Every millisecond, a new *Position Demand Value* is calculated and fed into the position control loop as a new set value. The generated path takes into account motion parameters such as acceleration, velocity, target position, profile type and others.

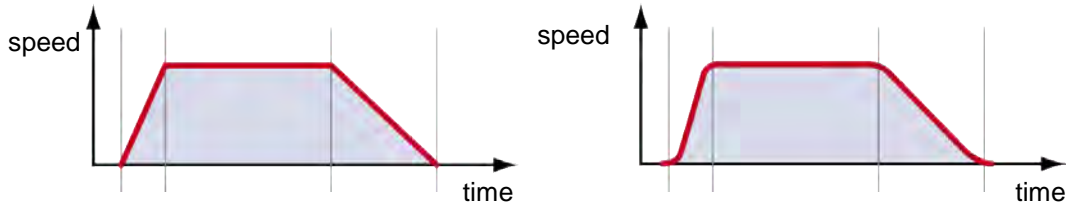


Figure 26: Speed profiles (speed vs. time) for a position move.
 Left: A trapezoidal profile with fast acceleration and slower deceleration rate.
 Right: The same profile but with smooth acceleration changes (sinusoidal profile).

Point to point movements

Here is a first step-by-step recipe how to command a movement.

- Step 1: Select the tool *Profile Position Mode* and activate the mode.
- Step 2: Set *Target Position* to 20'000 qc.
- Step 3: Set *Profile Velocity* to 500 rpm.
- Step 4: Set *Profile Acceleration* to 10000 rpm/s.
- Step 5: Set *Profile Deceleration* to 5000 rpm/s.
- Step 6: *Enable* the EPOS2.
- Step 7: Activate the movement by clicking on the *Move Relative* button and observe the motor spinning. Follow the position change also in the lower right of the dialog window.

Try other movements with different parameters (velocity, target position ...).
 Certainly, you can figure out the difference between *Move Absolute* and *Move Relative*.



Green and red LED: Indication of the internal EPOS state

The green and red LED on the EPOS2 P indicate the state of the internal EPOS2 motion controller.

- Green LED blinking Power state disabled. The motor is not powered.
- Green LED continuously ON Power stage enabled. The motor is powered and controlled.
- Red LED ON An error has occurred.

The state of these LEDs can also be seen on top of the dialog windows in the EPOS Studio.



Enable and disable the power stage

The terms *Enabled* and *Disabled* refer to the state of the output power stage of the motion controller.

- *Enabled* The output stage of the controller powers the motor. The motor receives energy and can be controlled.
- *Disabled* The power stage is switched off. The motor is not powered; there is no torque produced. The control electronics however is still powered.

Remark: The terms *Enabled* and *Disabled* will also be used to control the execution of digital and analog inputs and outputs (see chapter 4).



The control loops in motion control

In motion control, the regulated parameter is generally a force, torque, speed or position. Correspondingly, the controller is configured as a current, speed or position controller. It is often possible to switch between different types of control or operating modes.

Control of force/torque

This is the fundamental functionality of a drive in a closed-loop control system. It is the basis onto which the higher-level speed and positioning control systems build up.

Forces and torques are required to set masses in motion, to decelerate and stop their motion and to overcome friction forces. There can also be components of gravity against which work must be performed, either to lift and hold loads or to decelerate a downward motion. In some cases of torque control, it is not a question of moving anything, but simply a matter of pulling or pushing with a defined force against an obstacle.

Motor torque is a linear function of the motor current. Therefore, the basis of producing controlled forces and torques is to regulate and to control the motor current. It is the most basic control loop.

Control of velocity/speed of rotation

The speed controller attempts to match the actual measured speed to a demanded speed. Rotating or moving a body at a specified velocity requires that the body be accelerated first. Later the forces and torques that affect the speed must be compensated. To accomplish this objective, the motor has to generate the necessary torque - i.e. it needs current. Thus, the speed controller sends a corresponding command to the lower-level current control system (similar to the position controller in Figure 27).

Position control

To move an object to a specified final position, the object must first be set in motion (accelerated), moved and then decelerated. Finally, the specified position must be maintained against any interfering forces. All these operations require a motor torque that must be made available to the position control system. Ultimately, position control relies on the lower level current control loop for creating the necessary torque

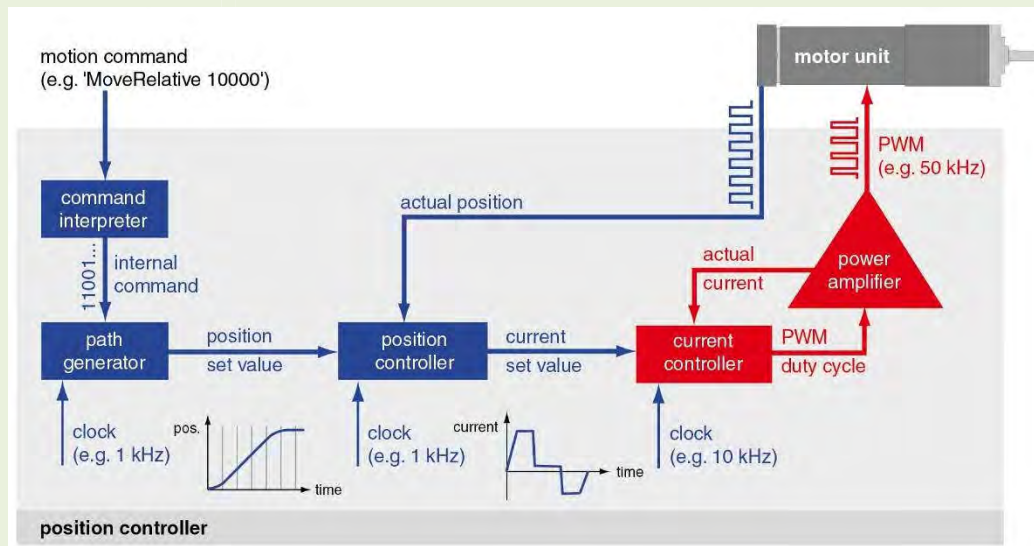


Figure 27: Diagram of a closed-loop position control system with a lower-level current controller. The master system sends motion commands to the motion controller. They are processed by the path generator that calculates intermediate positions at given time intervals (1ms on EPOS systems) on the path to reach the final position. These set values are transmitted to the position control loop, which, by comparison with the actual position, determines the set (or demand) values for the current controller.

The position and speed control loop in the EPOS2 work at a sampling frequency of 1 kHz; the control loop receives new position information every millisecond. This is fast enough for most applications, because mechanical reaction times are of the order of several milliseconds at least, as can be deduced from the mechanical time constants of the motors.

The current controller, via the output stage or driver, controls the motor current which leads to the mechanical reaction of the drive. In the EPOS the current control loop has a cycle time of 0.1 ms or 10 kHz. This is 10 times faster than the position control loop implying that the current is set almost instantly if needed from the position or speed controller.

Recorder configuration

The *Data Recording* is a useful tool for the analysis of motion behavior. It works in a similar way to a storage oscilloscope. For more information, there is a chapter regarding the use of the *Data Recording* in the document *Application Notes Collection*.

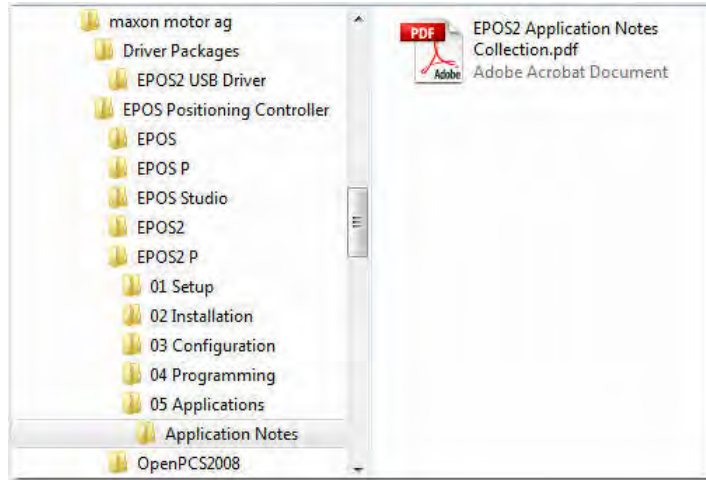


Figure 28: Where to find the Application Notes in the file structure of the maxon EPOS software installation.

We want to use the recorder to follow the movements in detail. Here is how the configuration is done step-by-step.

- Step 1: Select the tool *Data Recording*.
- Step 2: Click the button *Configure Recorder*.
- Step 3: In the *Configure Recorder* window select the button *Channel 1 Inactive* to activate channel 1 and select the button *Channel 2 Inactive* to activate channel 2.
- Step 4: Select the value *Position Actual Value* from the pull down menu for channel 1 and *Position Demand Value* from the pull down menu for channel 2.
- Step 5: Change the *Sampling Period* to 2 ms.
- Step 6: Change the trigger mode to *Single Trigger Mode* and tic the checkbox *Movement Trigger*. The next movement started triggers the recording.
- Step 7: Click on *OK*.

The first recording

Change to the tab *Profile Position Mode* and activate the mode. Execute a relative movement, e.g. with the following parameters:

- *Target Position* 4000 qc.
- *Profile Velocity* 2500 rpm.
- *Profile Acceleration* 20000 rpm/s.
- *Profile Deceleration* 8000 rpm/s.

Check the recorded data in the tab *Data Recording* and compare the set value position with the actual position.



Data Recorder

Attached cursor: With the *Attached Cursor* you can check the position values. The cursor is attached to the first displayed curve. Use the checkboxes to select which curves to show.

Zoom into the diagram by marking the interesting area with the mouse. Zoom out by right mouse click. You can zoom in and out in consecutive steps.

Save and export of recorded data to a text file (*.txt ASCII), e.g. suitable for Microsoft Excel import! (Hint: Use right mouse button on diagram). Open the exported file in the notepad and/or Excel. Other export formats are a bitmap or the special .rda file



Position accuracy

On a well-tuned system, end positions are reached to within 1 encoder quadcount (qc). As soon as the position is 1 qc off the target the position control loop will take corrective actions.

Remarks: The system reaction on deviations from the required positions depends on the torque and speed capabilities of the system, on the friction and mass inertias and on the set feedback and feed-forward parameters. The controller parameter values have been established during the tuning process based on the system properties. One recognizes again that tuning should be done with the full system in place.

Another aspect of position accuracy - besides this rather static behavior around the target position - concerns how the target position is reached. There are different parameters that describe the behavior as can be seen in

Figure 29. Again, the results depend on how the system was tuned and which aspect of accuracy was given the most weight during the tuning: e.g. no position overshoot permitted, or very fast motion regardless of overshoot.

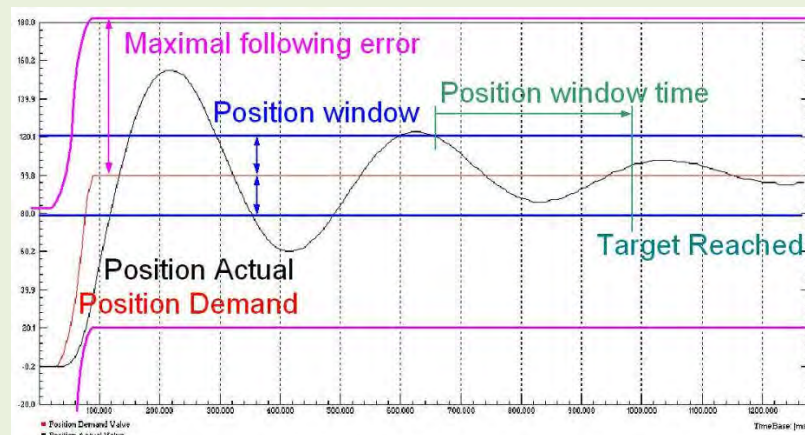


Figure 29: Target reached.

The parameters that govern the reaching of the end position of a movement (from the document EPOS 2 Firmware Specification). The black curve shows an overshoot of the actual position. The target position is judged and signaled as Reached if the actual position remains within a specified Position window around the Demand value for more than the specified Position window time.

Limiting parameters of motion profiles

Some general parameters limiting any motion command can be found in the upper right of the *Profile Position Mode* tab. These general limits can be set to protect the system from any unintended and dangerous motion.

Set the *Max Profile Velocity* to 1000 rpm and try to start a movement with a *Profile Velocity* of 1200 rpm.

Set other limitations and observe the behavior.



Maximum Following Error

The *Max Following Error* is the maximum permissible difference between demanded and actual position at any time of evaluation. It serves as a safety and motion-supervising feature.

If the following error becomes too high, this is a sign of something going wrong: Either the drive cannot reach the required speed or it is even blocked. However, allow for a certain amount of following error because feedback control will only work if there is a difference between the demanded and actual value. Hence, don't reduce the *Max Following Error* too much.

3.2 Homing

Objective	Executing a homing with current threshold. Knowing the different parameters of the homing mode.
-----------	--

Since we have not yet treated the inputs and outputs of the *EPOS2*, we execute a homing by means of a hard stop with no need of connecting any switches. (For homing with limit switches, refer to chapter 4.) Just follow these steps:

- Step 1: Select the tool *Homing Mode* and activate the mode.
- Step 2: Select the homing method *Current Threshold Positive Speed*.
- Step 3: Set the parameter *Current Threshold* to 500 mA.
- Step 4: *Enable* the *EPOS2*.
- Step 5: Start the homing with the button *Start Homing*.
- Step 6: Evoke a current peak by blocking shortly the motor shaft by hand. As a result, the *EPOS2* assumes to run into a mechanical stop and the home position is detected!

Repeat the homing exercise using different parameters and try to find answers to the following questions:

- Which mechanical positions does the disk on motor shaft stop at?
- What changes if you select the homing method *Current Threshold Negative Speed*?
- Which mechanical positions does the disk on motor shaft stop at if you select the homing methods *Current Threshold Positive Speed & Index*?
- If you execute a homing with index: What is the difference between the parameters *Speed Switch* and *Speed Index*?



What is homing and why is it needed?

The output signals of incremental encoders can only indicate relative position information or position differences. For absolute positioning, the system must be initialized to a zero or reference position first. This process is called homing, which is realized by moving the mechanism to a predefined position, relative to which all other positions are referenced. Such reference points are usually implemented by means of an inductive reference or limit switch or by means of a mechanical stop.

Homing with additional move to index channel

To increase the repeatability and accuracy of the home position, an additional motion to the first state transition of the index channel (channel I or Z of the encoder) can be performed. The index channel I delivering one pulse per revolution, this gives an absolute position within one motor turn.

By using this method of homing, it no longer matters if the reference switch actuates somewhat earlier or later; the homing position is now very precisely defined by the transition of the index channel state.

However, be careful in real systems. Using the index as a precise home position can only work, if the index pulse does not fall into the reference switch position error. That is why machine builders try to avoid using this function. If you have to change the motor or encoder, the index will not be at the same position and you will have to recalibrate the system and change the programmed positions.

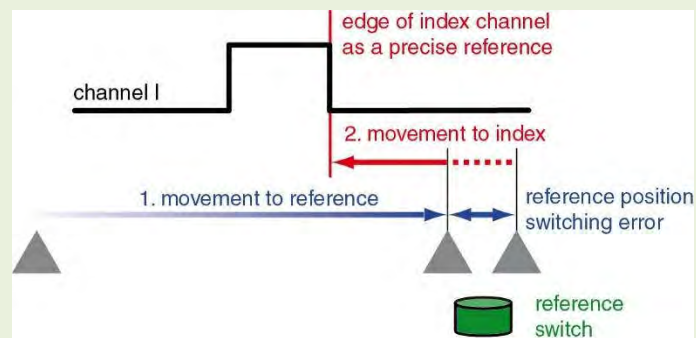


Figure 30: Principle of the homing mode with the index channel of the encoder.

The move to the reference switch (or mechanical stop) is followed by a movement back to the signal edge of the index channel. This always produces the same reference position, regardless of possible position errors during activation of the reference switch (due to fouling, for example).

3.3 Position Mode

Objective	Recognize the differences between <i>Profile Position Mode</i> and <i>Position Mode</i> .
-----------	---

The *Position Mode* allows positioning without profile, i.e. the target position is set immediately as a new set value for the position control loop. The path generator does not intervene with its nicely timed progression of position values. In the *Position Mode* only absolute positioning is possible, there is no relative motion.

Again, first a recipe for the evaluation of the *Position Mode*

- Step 1: Set the actual position to 0, e.g. by homing. This avoids troubles with too high a following error.
- Step 2: Configure the *Data Recorder*:
Select *Position Demand Value* and *Position Actual Value* as recorded parameters.
Set the *Single Trigger Mode* and *Movement Trigger*.
- Step 3: Change to Tab *Position Mode*, activate *Position Mode* and *Enable* the device.
- Step 4: Start a move of 1000 qc (*Apply Setting Value*) and observe the motor reaction.
- Step 5: Examine the move in the recorded diagram.

Reduce the maximum speed and/or velocity in the upper right *Parameters* part of the *Position Mode* tab window and observe the dampened system reaction.



Limiting motion parameters and damping system reaction

You can dampen the system reaction in *Position Mode* by limiting the *Max Profile Velocity* and the *Max Acceleration* in the upper right part of the *Position Mode* tab window. I.e. these two general limitations are still active even in positioning without profile. Large position steps can result in very strong system reactions. The needed acceleration currents can be very high because the EPOS tries to achieve the target position as fast as possible. The high acceleration currents might cause a problem for the power supply; the supply voltage might temporarily fall below the minimum supply voltage level causing a restart of the whole *EPOS2 P* device.

Remark: Such general limitations are active in all operating modes. In particular they can lead to errors if the application program sends commands that contradict these limitations; e.g. if a speed profile demands too high an acceleration rate.



Positioning without profile

The *Position Mode* is a useful operating mode for a situation where the axis acts as a slave axis commanded by progressive positions set values without large jumps. Hence, the path generator is not needed. An example for a progressive set value is an analog set value voltage. For more information, consult chapter 5.1.

Special operating modes without path generation are *Master Encoder Mode* where the slave axis follows the signal pulses from an external encoder (e.g. that of a master axis) or *Step Direction Mode*, where each pulse command from a stepper motor drive corresponds to a small rotating angle. More information in chapters 5.2 and 5.3.

Max Following Error and Position Mode

Set the *Max Following Error* to a value higher than the largest position step to be expected. Positioning steps higher than the *Max Following Error* are not allowed. The error occurs as soon as the EPOS tries to execute the command since the new set value (i.e. the target position) differs too much from the actual position (i.e. the starting position).

3.4 Profile Velocity Mode

Objectives	Setting a speed in the <i>EPOS2</i> (speed control). Monitoring the speed in the built-in <i>Data Recorder</i> .
------------	---

The next operating mode to explore is the *Profile Velocity Mode*. This mode controls motor speed and not position. The new target speed is achieved by a speed ramp - or if you like a speed profile, hence the name of the mode - with acceleration or deceleration values that can be set. The *Profile Velocity Mode* allows smooth speed changes.

As a starting point for your exploration in the *EPOS Studio* execute the following velocity profile and record the movement.

- Step 1: Select the tab *Data Recording* and configure the recorder.
Set *Channel 1, 2 and 3* active.
Channel 1: *Velocity Demand Value*, Left Scale
Channel 2: *Velocity Actual Value*, Left Scale
Channel 3: *Velocity Actual Value Averaged*, Left Scale
Select *Continuous Acquisition Mode*
Set a *Sampling Period* of 2 ms and press *OK*.
- Step 2: Select the tab *Profile Velocity Mode* and activate the mode.
- Step 3: Set *Target Velocity* to 2000 rpm. If you set a relatively low acceleration value of e.g. 200 rpm/s you can observe the disk on the motor shaft speeding up.
- Step 4: *Enable* the device and start the motion with the button *Set Velocity*.
- Step 5: Observe the velocity signals on the data recorder.



Understanding Velocity Signals on Data Recorder

Looking at the speed signals on the data recorder you might see something resembling the following picture.

Obviously, the horizontal red line is the commanded speed value of 2000 rpm.

The black line is the speed reading of the sensor. It looks quite noisy and a closer look shows that it jumps in steps of 30 rpm. Additionally, there is a certain periodicity in the signal with a period of 30 ms.

This periodic behavior can more clearly be seen in the green signal that shows the averaged velocity.

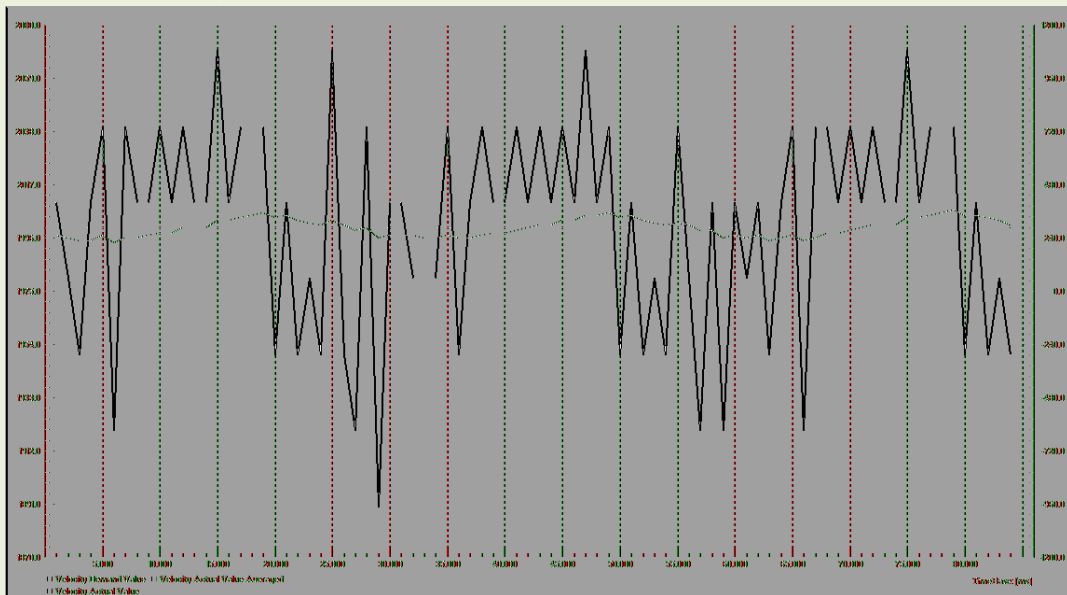


Figure 31: Velocity signals recorded on a motor with MR encoder. Screenshot from the Data Recorder.

Where do the steps of 30 rpm in the black signal come from?

Certainly, the motor is not able to change its speed in the way suggested by this signal; its mechanical time constant being too high (> 4 ms). Instead, what we observe is a quantization phenomenon in speed measurement stemming from the fact that velocity is calculated from the position change per millisecond. The position can change in steps of 1 quad count (qc) of the incremental encoder. 1 qc per millisecond is equivalent to 60'000 qc per minute and this corresponds to 30 rpm for an encoder with 2000 qc per turn.

Therefore, if you want to know the real speed of your motor it's better to look at the *Velocity Actual Value Averaged*. i.e the green signal in the diagram above. It makes more sense.

By the way, the speed accuracy of the averaged signal is about 3 rpm deviation at a speed of 2000 rpm. And this is rather good!

Where does the periodicity stem from?

The periodicity in the *Velocity Actual Value* and its average correlates with the rotation of the motor. At 2000 rpm one turn of the shaft takes exactly 30 ms, as observed in the signal.

One can think of two possible reasons for these oscillations: Either it's some irregularity in the motor, e.g. an enhanced friction on one side of the motor, or it comes from imperfections of the measuring device, i.e. the encoder.

Here, I think it's rather the second reason. MR encoders are known to have this sort of waviness. A comparison with a very accurate optical encoder on the same motor clearly shows the difference (Figure 32).

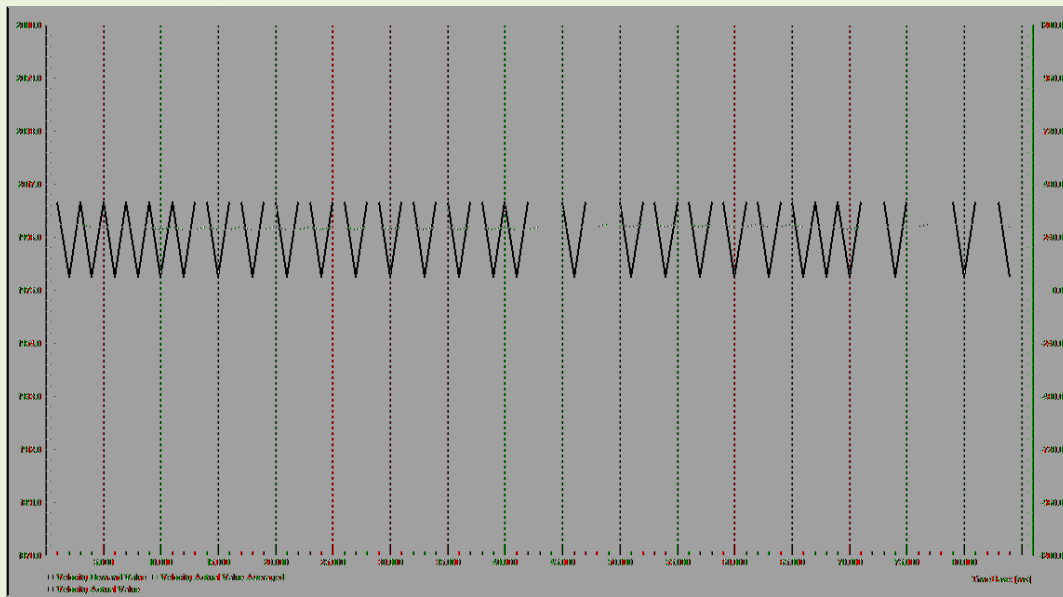


Figure 32: Velocity signals recorded on a motor with optical encoder. Screenshot from the Data Recorder.

3.5 Velocity Mode

Objective	Recognize the differences between <i>Profile Velocity Mode</i> and <i>Velocity Mode</i> .
-----------	---

The *Velocity Mode* is very similar to the *Profile Velocity Mode* but there is no predefined ramp. It's just the general limitations that are still active. Try the following exercise as a starting point for your own exploration of this operating mode.

- Step 1: Select the tab *Data Recording* and configure the recorder.
 - Set Channel 1, 2 and 3 active.
 - Channel 1: *Velocity Demand Value*, Left Scale
 - Channel 2: *Velocity Actual Value Averaged*, Left Scale
 - Channel 3: *Current Actual Value Averaged*, Right Scale
 - Select *Single Trigger Mode* and *Movement Trigger*
 - Set a sampling time of 2 ms and press *OK*.
- Step 2: Select the tab *Velocity Mode* and activate the mode.
- Step 3: Change *Setting Value* to 2000 rpm.
- Step 4: *Enable* the device and start the motion with the button *Apply Setting Value*.
- Step 5: Look at the velocity and current signals on the data recorder. Observe how the current increases whenever the speed error becomes larger.

3.6 Current Mode

Objective	Learning how the motor behaves upon current control only.
-----------	---

Explore the *Current Mode*, e.g. with the following steps.

- Step 1: Change to tool *Current Mode* and activate it.
- Step 2: *Enable* the device.
- Step 3: Block the motor shaft with your hand! (Why?)
- Step 4: Set the current *Setting Value* to 100 mA and *Apply Setting Value*. What do you observe, what do you feel on the blocked shaft.
- Step 5: Increase the current, e.g. to 500 mA or higher.



Runaway in current control mode

If you don't block the motor shaft in current control the motor speed goes up to very high levels. Why does this happen?

Current control means torque control. The controller tries to maintain the current (torque) at the set value. Torque on the motor usually means acceleration => Higher speed results in a higher induced voltage (back EMF) that counteracts the applied voltage. => More applied voltage is needed to maintain the current at the set value. => Higher voltage means the motor accelerates further => higher back EMF => more applied voltage needed => resulting higher speed =>

...

As a result, the motor speed runs away up to the maximum speed that is possible with the given supply voltage. Such a runaway condition can occur unless it is limited by a higher-level control loop (velocity or position control), a mechanical stop or an additional safety speed limitation (as the *Max Speed* in the limitation section of the current mode interface screen).

With the *Current Mode* we have finished the exploration of the basic motion control functionality.

4 Using the I/O Monitor tool

Objectives	Using the <i>I/O Monitor</i> and practical understanding of the I/O-functionality and masks.
------------	--

There are 6 digital and 2 analog inputs and 4 digital outputs available on the *EPOS2 [internal]*. These I/Os are primarily intended to be used for special tasks in the periphery of the corresponding motion axis. One example for an input is a limit switch that indicates that the mechanical drive has left the save operation area on a linear drive. Moving the device into the limit switch will cause the motion controller to signal this specific error to the master system. Limit switches might also be used as a reference for homing.

The I/Os of the EPOS are freely configurable. You can use them for the predefined functionalities or for any purpose you like (*General Purpose*). You might use one of the digital inputs to start a particular part of your program in the master PLC, or read the signal of a temperature sensor with an analog input, or signal the end of a subroutine on one of the digital outputs.

In the EPOS Starter Kit on your table, switches can activate the digital inputs, the analog inputs read the voltage of a potentiometer and the digital outputs illuminate colored LEDs. Open the *I/O Monitor* tool and try the following exercises to learn how the inputs and outputs work and how they can be configured. For more details, refer to the corresponding chapters in the documents *Application Notes Collection* and *Firmware Reference*.

4.1 Outputs

As a first exercise let us set a digital output and learn how we can mask the physical output.

- Step 1: Click on *Purpose* column in the table and set the digital outputs 1 to 4 as *General A* to *General D*. Set the *Mask* to *Enabled* and the *Polarity* to *High Active*.
- Step 2: Set the *State* of some of the digital outputs to *Active* and observe the reaction of the LED outputs on the PCB.
- Step 3: Change the *Mask* to *Disabled*. How does this affect the setting of the outputs and the effect on the LEDs?
- Step 4: Change the *Polarity* to *Low Active*. How does this affect the LED output?

Now we assign a specific functionality to a digital output.

- Step 1: Set the *Purpose* of the *Digital Output 4* (or any other) to *Ready*.
- Step 2: Watch the behavior of this output on the PCB during the next steps, e.g. when an error occurs.

4.2 Inputs

Here are some exercises for the inputs.

First, we read the analog inputs: Turn the potentiometers on the PCB and observe the value reading on the *I/O Monitor*.

Remark: There is a delay between the activation of the input switches or potentiometers on the PCB and the reaction of the *I/O Monitor*. There is a lot of information to be transmitted for the refreshment of the *I/O Monitor* display in the *EPOS Studio*; hence, it takes a while. Don't worry, the internal state of the inputs are changed immediately.

Then, we configure and read some digital inputs.

- Step 1: Set the purposes of digital inputs 1 to 6 as *General A* to *General F*. Set the *Mask* to *Enabled* and the *Polarity* to *High Active*.
- Step 2: Activate one of the digital inputs switches on the PCB and observe the reaction in the *I/O Monitor*.
- Step 3: Change *Mask* to *Disabled*. How does this affect the reading of the input?
- Step 4: Change *Polarity* to *Low Active*. How does this affect the reading of the input?

Now, let's assign functionality to a digital input.

- Step 1: Set the purpose of the digital input 1 to *Negative Limit Switch* and *ExecMask* to *Enable*.
- Step 2: Activate digital input switch 1 on the PCB and observe the reaction of the EPOS and the *Ready* output.
- Step 3: Clear the error in the *EPOS Studio*.
- Step 4: What is the difference between *Mask* and *ExecMask* (execution mask)?

Finally, we apply this functionality: Use the digital input 1 to perform a homing onto the *Negative Limit Switch*. Since you learned to perform a homing in chapter 3.2 you should be able to do this without step-by-step instructions. How about the reaction of the *Ready* output?



Save parameters

Save the changed I/O parameters by right-clicking on the header of the *I/O Monitor* window. This avoids the loss of your settings in case of a power shut down.



Axis related input functionalities

Limit Switches and Home Switch

We have already seen what *Limit Switches* can be used for. They limit the mechanical working range on both sides. Very often inductive proximity switches are used as limit switches; if a metallic part comes in close proximity they change their output state. If a limit switch is activated the drive is assumed to have moved out of the permitted area. An error signal is generated; the drive comes to a stop and is disabled.

You can use one of the limit switches for homing after power up. No extra *Home Switch* is needed and it is clear where to look for it; at the corresponding positive or negative end of the mechanical travel. When using the limit switch for homing no error is generated of course.

For homing, you can also use a specially designed *Home Switch* and its input functionality.

Device Enable and Quick Stop

In most cases the enable and disable of the power stage in the controller is governed by the commands from the program in the master that controls the process. The *Enable* input functionality offers an additional way to control the power stage by an external digital signal: A rising edge with high active polarity on this input will enable the device; a falling edge will disable it.

This functionality might be useful in situations where you want to remove the power from the motor without referring to the program in the master. However, be careful, this feature does not comply with safety regulations!

The *Quick Stop* works in a similar way, but it brings the axis to a stop and holds the final position. The motor is still powered, i.e. the power stage remains enabled.

Capture Input: Position Marker

The *Position Marker* input records the actual position when this digital input is activated. A typical situation for this functionality is the following: When an object on a conveyor belt passes a certain point (e.g. a light curtain) this is signaled to the motion controller on the *Position Marker* input and the corresponding position of the conveyor is stored. The process control program may now add the offset for the object on the belt to stop at a defined position.

Axis related output functionalities

Ready/Fault

The *Ready/Fault* output can be used to signal the correct operation of the device. Think of the common green and red light accessory on production machinery.

Trigger Output: Position Compare

Position Compare allows sending signals at a predefined position or position intervals. This might be useful to activate another device or process by a digital signal if your axis moves through a specific position.

Best practice: If you want to see the output LED blinking on the EPOS Starter Kit, set the *Pulse Width* as long as possible, i.e. 64 000 μs .

Holding Brake

Holding brakes are used to maintain a position without the motor being powered. This is useful to save energy in applications where the drive has to rest at the same position for a long time or where holding a position requires a lot of torque (e.g. on a vertical drive). Holding brakes hold position when they are not powered. Therefore, holding brakes may also be used as an emergency stopping device upon power shut down.

The powerful *Holding Brake* output allows the direct activation of brakes without additional power supply, taking into consideration the reaction time of the brake activation.

5 Alternative Operating Modes (optional)

Alternative operating modes allow running the motion controller without receiving on-line commands. Instead, analog voltage signals or signal pulses are used as command values. These incoming signals are converted to set values that are directly fed into the corresponding control loop without a profile or trajectory being generated.

However, keep in mind that these operating modes do not correspond to the original intention of the EPOS product family, which is to be commanded on-line via a CAN-bus system from a programmed master device.

5.1 Analog set value

In the analog set value modes, on one of the analog inputs reads the position, speed or current demand value that is converted and applied to the corresponding control loop. Let's try this for speed and position control.



Hardware enable

Since these alternative operating modes are not on-line commanded we have to establish a possibility to enable and disable the power stage independently. Therefore, we should design one of the digital inputs as enable. Use the *I/O Monitor* tool to configure the purpose of one of the digital input switches as *Drive Enable*. Do not forget to set the masks correctly.

Analog speed control

Objectives	Setting up the analog speed control. Operating without serial on-line commanding.
------------	--

First, we have to tell the *EPOS2 [internal]* that it gets speed set value from one of the analog inputs. Use the *I/O Monitor* to configure one of the analog input purposes as *Velocity Setpoint*.

- Step 1: Change to Tab *I/O Monitor*.
- Step 2: Change the purpose of *Analog Input 1* to *Velocity Setpoint* (only possible if drive is disabled).
- Step 3: Click on the *Show Attributes* button at the bottom.
- Step 4: Enter a *Setpoint Scaling* factor between the analog input voltage and speed (e.g. 1000 rpm/V).
- Step 5: Optional: Save parameters (e.g. context menu).

Now perform a speed control with the analog set value.

- Step 1: Change to the tab *Velocity Mode* and activate it.
- Step 2: Set the analog potentiometer to the minimum. Observe the change in the lower right part of the tab window.
- Step 3: Enable the *Execution Mask* of the analog set value by activating the corresponding checkbox.
(After this configuration steps you may now remove the USB connection. However, if you want to follow the actual and demanded potentiometer and speed setting values on the screen leave the USB in place.)
- Step 4: Enable the EPOS, set different speeds with the potentiometer and observe the motor reaction.
- Step 5: Try different settings of *Setpoint Scaling* and *Setpoint Offset*.



Limiting and damping system reaction

You can dampen the system reaction by limiting the *Max Profile Velocity* and the *Max Acceleration* in the upper right of the tab window. Particularly recommended with noisy analog setvalues.

Analog position control

Objectives	Setting up an analog position control. Operating without serial on-line commanding.
------------	--

Use the *I/O Monitor* to configure one of the analog inputs as the analog *Position Setpoint* (only possible if drive is disabled) and one of the digital input switches as *Drive Enable*.

- Step 1: Perform a homing: Set actual position to 0.
- Step 2: Change to the tab *Position Mode* and activate it.
- Step 3: Set the analog potentiometer to the minimum. Observe the change in the lower right part of the tab window.
- Step 4: Enter a *Setpoint Scaling* factor between the analog input voltage and speed (e.g. 2000 qc/V).
- Step 5: Enable the *Execution Mask* of the analog set value mode by clicking on the corresponding checkbox. It is recommended to save your parameters.
(After this configuration steps you may now remove the USB connection. However, if you want to follow the actual and demanded potentiometer and position settings on the screen leave the USB in place.)
- Step 6: Enable the drive, increase the position by turning the potentiometer and observe the motor reaction.
- Step 7: Try different settings of *Setpoint Scaling* and *Setpoint Offset*.

5.2 Master Encoder Mode: Electronic gearhead

In the *Master Encoder Mode* the signal pulses of an external incremental encoder (channel A and B) are interpreted as position set values. Being set up in this mode, the *EPOS2 [internal]* tries to follow the motion of the external encoder signal.

It's possible to define a fixed ratio between the encoder count of the external and the internal encoder, as well as the relative direction. Concerning speed, this results in a behavior similar to the input and output of a gearhead: That's where the term *Electronic Gearing* stems from.

Sorry, there is no practical work possible with the hardware at hand. We do not have an external encoder.

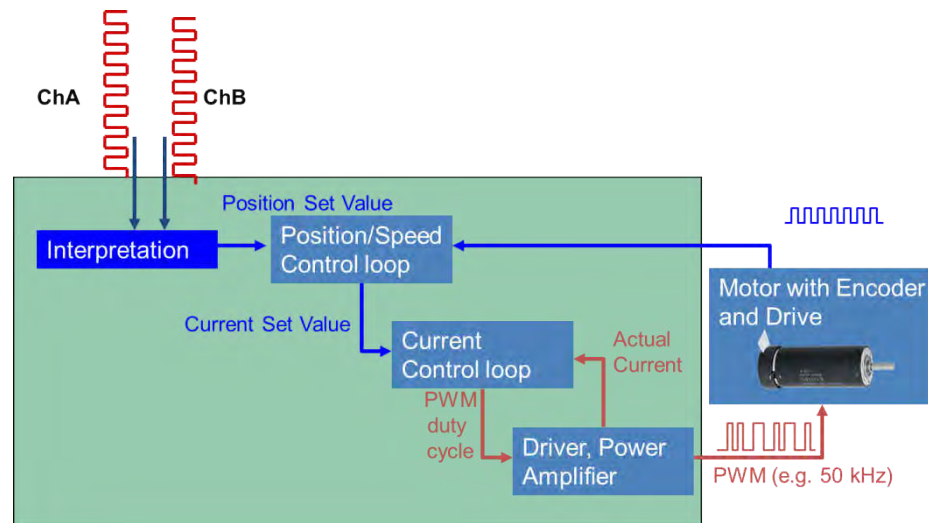


Figure 33: Schematic representation of the Master Encoder Mode.

5.3 Step Direction Mode

In the *Step Direction Mode* incoming signal pulses are interpreted as position increments. The state of a second digital signal gives the direction of rotation. The *EPOS2 [internal]* follows the incoming steps. It's possible to define a fixed value of how many encoder quadcounts should correspond to each step pulse.

Step and direction signals can be generated by many PLCs and are usually needed to drive stepper motors. With an EPOS2 motion controller and DC motor it is possible to replace a stepper motor with its drive electronics. This might be useful if the stepper drive system is too weak, too slow or too bulky.

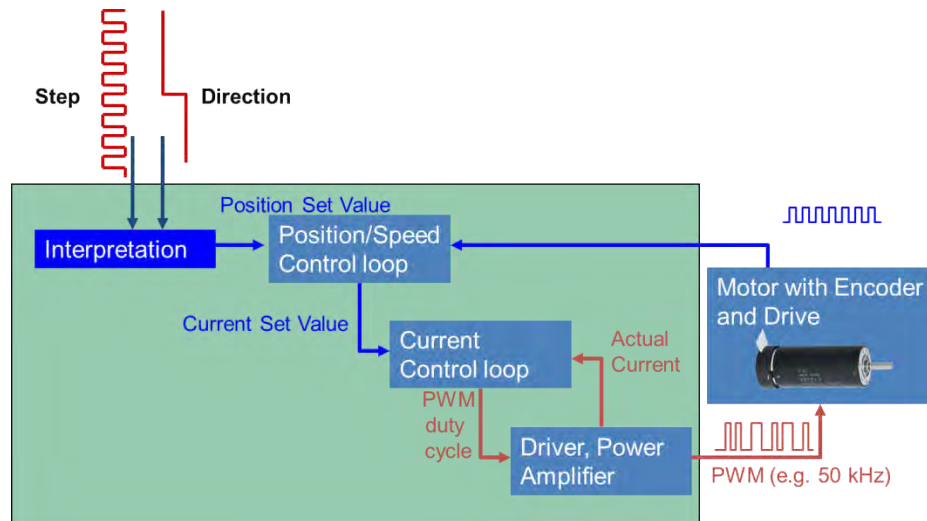


Figure 34: Schematic representation of the Step Direction Mode.

Simulation of Step Direction Mode with I/Os

Objectives	Setting up the <i>Step Direction Mode</i> using the digital inputs for step and direction signals. Manual operation without serial on-line commanding.
Step 1:	Use the <i>I/O Monitor</i> to configure the digital inputs 2 and 3 as General Purpose A and B, respectively. Save the parameters.
Step 2:	Configure one of the remaining digital input switches as <i>Drive Enable</i> . Set <i>Mask</i> and <i>Exec Mask</i> to <i>Enabled</i> .
Step 3:	Open the tool <i>Step Direction Mode</i> and activate the mode.
Step 4:	Set a <i>Scaling factor</i> , i.e. how many quadcounts should the motor axis move per signal pulse on digital input 3. E.g. set 200 qc per pulse, corresponding to a tenth of a motor turn in our case.
Step 6:	Reduce <i>Max Acceleration</i> (e.g. < 100 000 rpm/s) to dampen system response and avoid the power supply to shut down temporarily because of too high a current draw.
Step 5:	<i>Enable</i> the axis with the digital input switch and observe the motor behavior upon activating digital input 3 (the <i>Step</i> input) and the setting of digital input 2 (the <i>Direction</i> input). Follow the position on the screen. Try different settings of the different parameters on the screen.

Further description can be found in chapter 5 of the document *EPOS2 Application Notes Collection*.

Intermezzo: CAN and CANopen

The *EPOS2 P* is a *CANopen Device*, or more accurately, there are two *CANopen Devices*, an *IEC-61131 Programmable Device* and a *Motion Controller Device*. The purpose of this chapter is to give you the most important features and ideas connected with *CANopen Devices*. This provides you with a better understanding of how communication is organized. This will later be useful for programming.

However, this is just an introduction and not a full description of what CANopen is. Please refer to the CiA (*CAN in Automation*) website for further information (www.can-cia.org).

6 An introduction to CANopen and CAN

CANopen is a communication protocol applied to a field bus that is usually CAN. CAN covers the lower communication levels focusing on technical aspects, while CANopen defines the meaning of the data and devices in the network. In other, rather simplistic words: the CAN bus describes the vehicle how data are transmitted, and the CANopen protocol describes what kind of data are transmitted.

CANopen is a standard maintained and supervised by the independent user organization *CAN in Automation*.

Here, we don't have to bother about the details of how messages are transmitted and received. All we need to know is that CANopen and CAN provide a reliable, low cost bus framework for sending messages between different devices.

Some limitations include:

- Maximum numbers of nodes per bus 127 nodes
- Maximum bit rate 1 Mbit/s up to 40 m total bus length
All nodes need to have the same bit rate!
- Typical length of a CAN telegram 100 – 130 microseconds at maximum bit rate

The most important concept we have to deal with is the *Device Model* and the *Object Dictionary* therein. But first a few words on CAN and CANopen.

6.1 CAN

CAN stands for *Controller Area Network*. It is a field bus designed originally for automotive applications but is now also used in other areas such as industrial automation and medical equipment.

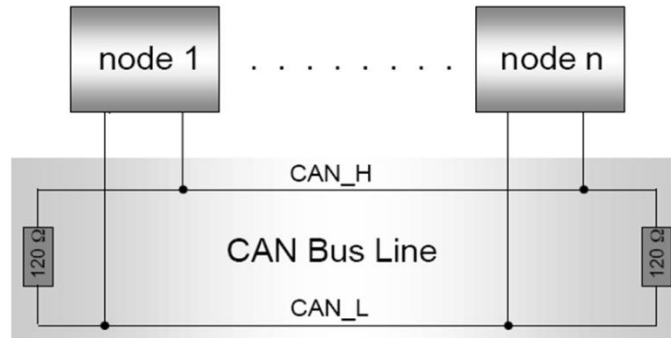


Figure 35: Physical layout of the CAN bus.

On a *physical level*, each device - called a node and given its distinctive address (node number) - is mounted in parallel on the bus. The bus transmits the signals on a differential line which needs proper termination on both sides to avoid signal reflections (120 Ohm resistances). Signal bits require a certain time to spread over the whole bus. Therefore, the transmission rate depends on the bus length. The maximum bit rate of 1 Mbit/s can only be achieved up to approx. 40 m bus length.

"The *Transfer Layer* represents the kernel of the CAN protocol." It is where the structure, transmission and reception of messages are defined. "The transfer layer is responsible for bit timing and synchronization, message framing, arbitration, acknowledgement, error detection and signaling, and fault confinement." (Wikipedia)

In short, CAN gives a framework for microcontrollers and devices to communicate in a safe and reliable way with each other.

6.2 CANopen

"In terms of the OSI model, CANopen implements the higher layers above and including the network layer. The CANopen standard consists of an addressing scheme, several small communication protocols and an application layer defined by a device profile. The communication protocols have support for network management, device monitoring and communication between nodes, including a simple transport layer for message segmentation/desegmentation. The lower level protocol implementing the data link and physical layers is usually Controller Area Network (CAN), although devices using some other means of communication (such as Ethernet Powerlink, EtherCAT) can also implement the CANopen device profile." (Wikipedia)

In short, CANopen uses the CAN framework for sending meaningful messages in between specified nodes (devices) in a network.

6.3 CANopen device profile

The CANopen device profile is one of the central elements of CANopen. It describes how the nodes in a network must be structured. You can think of a CANopen Device as consisting of three sections: a *communication unit*, an *object dictionary* and the *application part* (Figure 36). In addition, there has to be a state machine implemented for starting and resetting the device. It must contain the states Initialization, Pre-operational, Operational and Stopped. Typically, the pre-operational state is used for configuration while real time communication is restricted to the operational state.

The *communication unit* is responsible for communication with the other nodes in the network. It contains all CAN and CANopen communication features: Receiving and sending network management information as well as writing and reading data in or from the object dictionary.

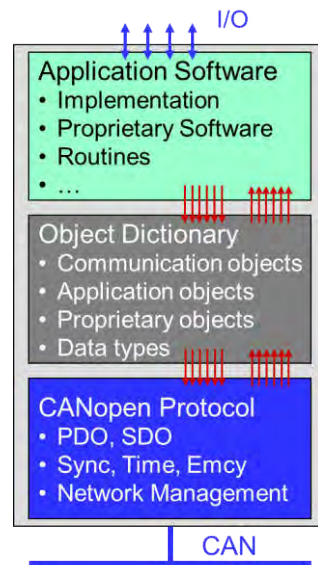


Figure 36: CANopen device profile

The *Application* part actually performs the desired function of the device. This can be a motion controller as the *EPOS2 [internal]*, a programmable logic controller as the *EPOS2 P* or any other functionality. The application is configured and controlled by parameters and variables in the *Object Dictionary*.

Object dictionary

The *Object Dictionary* is the heart of the device. It represents the link between the CANopen communication world and the application. The application uses the data from the object dictionary as inputs to perform its task and also writes its results and outputs to the same object dictionary. On the other side, any communication between devices in the CANopen network is based on data exchange between corresponding object dictionaries. Hence, communication with a CANopen device comes down to writing or reading parameters from the corresponding object dictionary.

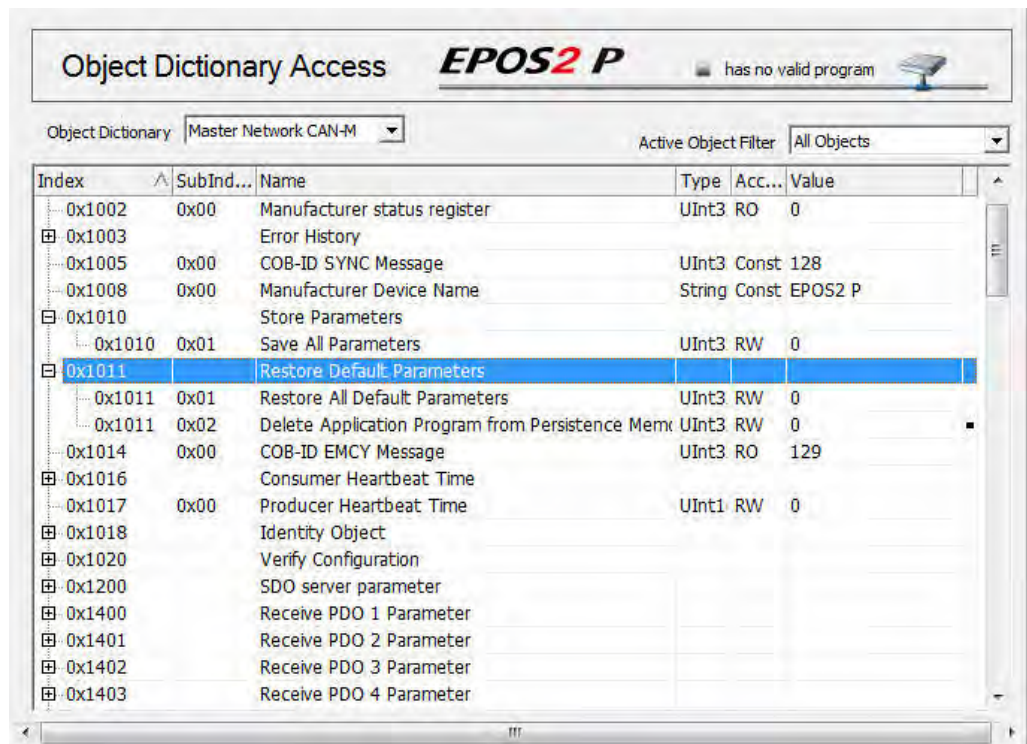


Figure 37: Access to the Object Dictionary of the EPOS2 P in the EPOS Studio Tools tab.

The *Object Dictionary* consists basically of a table of parameters describing all the properties of the device: its communication channels, its configuration and settings, but also the application configuration, input and output data.

An entry in the object dictionary is defined by

- Address (16-bit index and 8-bit subindex)
- Name a string describing the parameter
- Type the data type of the parameter
- Access rights read/write, read-only, write-only or read-only constant
- Value of the variable

The basic data types for object dictionary values such as Booleans, integers and floating numbers are defined in the CANopen standard, as well as composite data types such as arrays, records and strings.



Object Dictionaries of EPOS2 P

The complex structure of the EPOS2 P is reflected in the different object dictionaries and CAN ports. The built-in PLC has the CAN-I connection to the EPOS2 [internal] slave, the CAN-S port may be used for other slave devices, and the CAN-M port for commands from an even higher level of control hierarchy. Each of these three different CAN interfaces has its own object dictionary as well as the EPOS2 [internal].

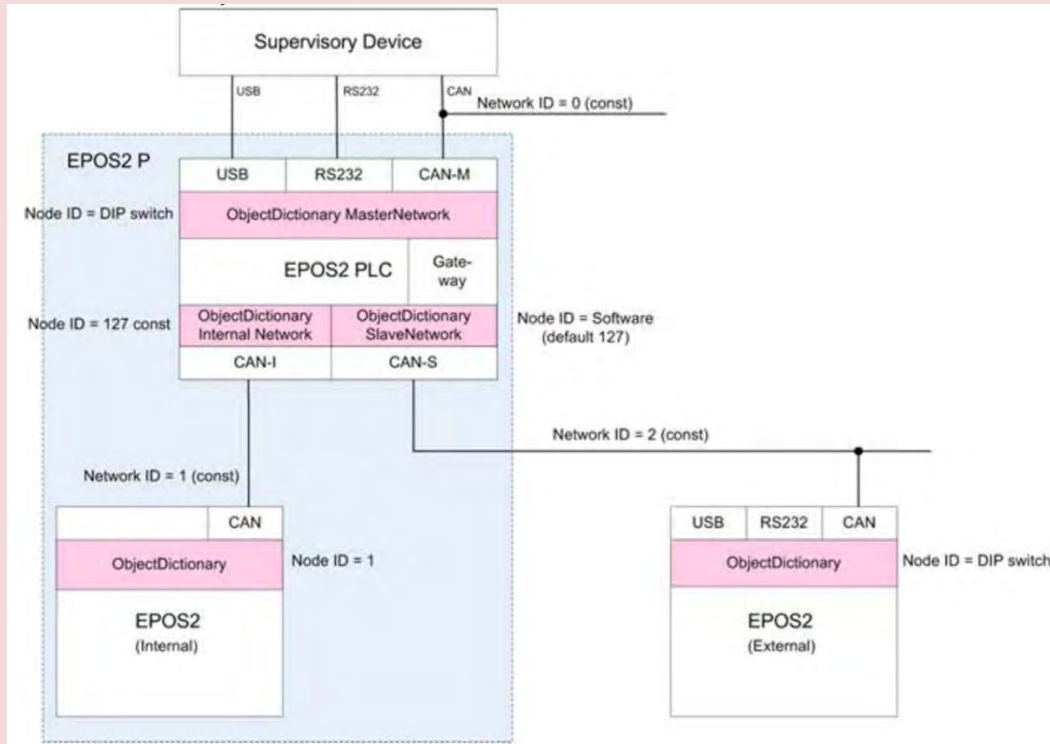


Figure 38: The different Object Dictionaries in an EPOS2 P Network.

Restrictions on variable types of EPOS systems

The EPOS2 systems does not contain any float type variables. Numeric variables can only be integers of different size.

Firmware Specification

All the objects on the EPOS2 P and the EPOS2 [internal] are described in the corresponding Firmware Specification. In there, you find as well information about the value range and the meaning of the parameter values (units).

Standard CANopen Device Profiles

Object dictionaries cannot be defined at will. First, there are certain rules to be observed regarding which data can be found where in the object dictionary. Second, there are standard device profiles for a variety of applications. In our situation, the most important profiles are the numbers 402 and 405 for motion controller and programmable devices.

Profile number	Device class
CiA 401	Generic I/O Modules
CiA 402	Drives and Motion Control
CiA 404	Measuring devices and Closed Loop Controllers
CiA 405	IEC 61131-3 Programmable Devices
CiA 406	Rotating and Linear Encoders
CiA 408	Hydraulic Drives and Proportional Valves
CiA 410	Inclinometers
CiA 412	Medical Devices
CiA 413	Truck Gateways
CiA 414	Yarn Feeding Units (Weaving Machines)
CiA 415	Road Construction Machinery
CiA 416	Building Door Control
CiA 417	Lift Control Systems
CiA 418	Battery Modules

Figure 39: Some standard CANopen Device Profiles (from CiA website).

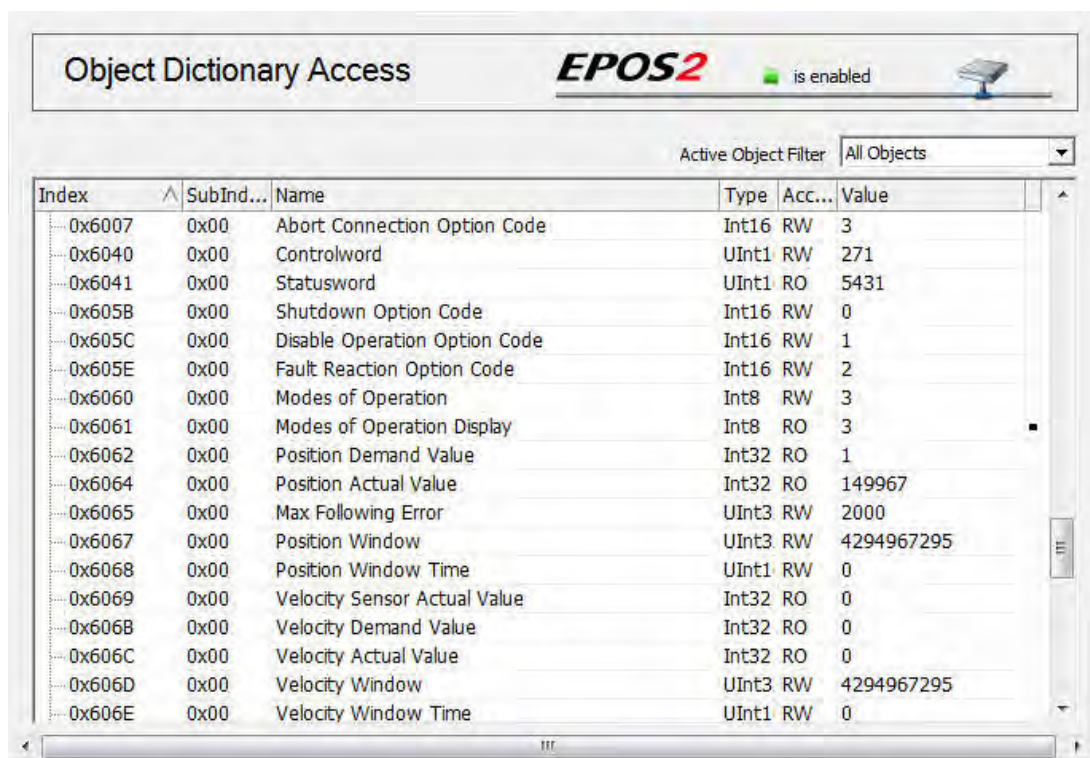
It is important to note that the standard is *open* in the sense that there is a certain degree of freedom which of the objects and functionality must be implemented. For a device to comply with the standard, it must contain certain objects but not all of them. Moreover, the manufacturer is free to add extra objects and functionality as well.

6.4 EPOS2 [internal] Object Dictionary tool

Objectives	Editing system parameters in the object dictionary. Generating an object filter.
------------	---

Let's take, as an example, the object dictionary of the *EPOS2 [internal]*. The motion control application uses data from the object dictionary (e.g. target position) as inputs to perform a motion and writes the results (e.g. target position reached) to the same object dictionary.

The object dictionary of the *EPOS2 [internal]* is accessible as a tool in the *EPOS Studio*. Just open the *Object Dictionary* and activate the Object Filter *All Objects* in the upper right corner.



Index	SubInd...	Name	Type	Acc...	Value
0x6007	0x00	Abort Connection Option Code	Int16	RW	3
0x6040	0x00	Controlword	UInt1	RW	271
0x6041	0x00	Statusword	UInt1	RO	5431
0x605B	0x00	Shutdown Option Code	Int16	RW	0
0x605C	0x00	Disable Operation Option Code	Int16	RW	1
0x605E	0x00	Fault Reaction Option Code	Int16	RW	2
0x6060	0x00	Modes of Operation	Int8	RW	3
0x6061	0x00	Modes of Operation Display	Int8	RO	3
0x6062	0x00	Position Demand Value	Int32	RO	1
0x6064	0x00	Position Actual Value	Int32	RO	149967
0x6065	0x00	Max Following Error	UInt3	RW	2000
0x6067	0x00	Position Window	UInt3	RW	4294967295
0x6068	0x00	Position Window Time	UInt1	RW	0
0x6069	0x00	Velocity Sensor Actual Value	Int32	RO	0
0x606B	0x00	Velocity Demand Value	Int32	RO	0
0x606C	0x00	Velocity Actual Value	Int32	RO	0
0x606D	0x00	Velocity Window	UInt3	RW	4294967295
0x606E	0x00	Velocity Window Time	UInt1	RW	0

*Figure 40: Object Dictionary of the motion controller EPOS2.
Shown are the first entries of the standard motion control profile.*

Scrolling through the list we find the following groups of entries (the index and subindex are given as hexadecimal numbers, hence the prefix 0x):


- Index 0x1000 to 0x1FFF communication profile entries (in the CANopen standard)
- Index 0x2000 to 0x5FFF maxon EPOS specific entries (not in the CANopen standard)
- Index 0x6000 to 0x9FFF standard device profile entries (in the CANopen standard)

The maxon specific entries contain - among others - the objects related to non-CANopen communication as well as objects related to the digital and analog inputs and output

configuration.

In the standard device profile entries, you find all the information related to motion control.

The *Object Dictionary* tool gives you access to the full list of objects. You can change the value of any entry provided it is writable. Simply double click on the object or use the context menu (right mouse click on the list).



Refreshing rate of the *Object Dictionary* in the *EPOS Studio*
USB connection is not extremely fast and it is not real-time, i.e. it does not have the highest priority on your computer and you cannot predict how long it takes for the object dictionary values to refresh. You can accelerate the response time to some extent by minimizing the *Refresh Rate* in the menu bar.

||

Figure 41: Where to change the Refresh Rate.



Best Practice

Define your own object filter

Looking at the full object dictionary with its many entries can be quite overwhelming. Often you are interested in just a few entries. For this purpose, you can create your own object filter, similar to the predefined *System Parameter* filter.

Here is a recipe for watching the PID gains:

- Step 1: Right-click to the dialog window and select *Define ObjectFilter*.
- Step 2: Select *New* to generate a new object filter.
- Step 3: Appoint the object filter with *MyRegGains*.
- Step 4: *Add* the objects:
 - *0x60F6 (Current Control Parameter Set)*
 - *0x60F9 (Velocity Control Parameter Set)*
 - *0x60FB (Position Control Parameter Set)*
- Step 5: *Save* the object filter and *Exit*.



Save parameters

Changes in the Object Dictionary tool will only be permanently stored in the EPOS after a *Save All Parameters* command (right mouse click to access the context menu of the Object dictionary).

Basically there are three sets of *Object Dictionaries*:

- the *actual parameters* in the working memory (RAM) which are lost after a power down.
- the *permanent parameters* saved in the EEPROM which are active after a power up.
- the *default parameters* (i.e. the factory setting) which is always in the EEPROM as a last back-up.

6.5 EPOS2 P Object Dictionary

As can be seen in Figure 38, the PLC part of the *EPOS2 P* has three different object dictionaries – one for each CAN network. Refer also to Figure 37.

Besides the usual CAN communication and device information, the object dictionaries of the PLC contain essentially the input and output process variables. This is the information the PLC works with. The PLC application uses data from the object dictionaries as input variables to perform a motion – e.g. the signal from a position compare output of an axis that a predefined position has been reached. The PLC then writes the results of the actual program run to the relevant object dictionaries – e.g. an output signal that is used to stop an axis.

Most of our PLC programming in the following chapters, however, does not use these process I/Os. Instead, we read and write the relevant information directly in and from the slave object dictionaries.



PLC background information

PLC stands for *Programmable Logic Controller*. Essentially, it is a computer (or better a central control unit) with one or several programs running on it. The PLC contains an input module where information from the process (from sensors) comes in and an output module, where the outcome of the process is set (to drive actors).

PLC programs run in cycles: They read the actual process inputs, perform a program run with these input values and set the new process outputs. This cycle is repeated according to the *task* settings (see page 96). A program cycle can be of any length, but typically, it is of the order of a few milliseconds.

6.6 Parameter Up- and Download

The Object Dictionary contains all the parameters describing a CANopen device. It is a fingerprint of the device. The parameter list can be saved in a special electronic data sheets called *Device Configuration File* (parameter export). Thus, it is possible to configure different devices in the same way simply by downloading the corresponding electronic data sheet in the *EPOS2 P* (parameter import).

In the *EPOS Studio*, there are *Parameter Export/Import* wizards for this purpose. They also allow resetting the default parameters.

6.7 Communication

Communication in the basic CAN layers is based on broadcast communication: Each node can send messages that can be picked up by any other node. In order to avoid data collision each telegram has a defined priority. If two messages start at the same time the one with lower priority value wins and is transmitted first.

In CANopen the communication is more specific in the sense that there are predefined communication channels (each with a certain priority level) describing which node is the sender and which node is the receiver of the message (peer-to-peer communication).

Network Management and special messages

The lowest priority values are assigned to messages that ascertain that the network operates correctly and that the communication is reliable. In this category, we find error control and emergency messages, synchronization and time stamp messages as well as network management messages such as booting and changing the operational state of a node. The exact priority sequence within these categories is not of importance for the purpose of this textbook.

Process Data Objects (PDO)

The next priority is given to *Process Data Objects* (PDO). PDOs are for real-time data exchange of small objects. Typically, these are data that change a lot in the process (sic the name!) and need to be updated frequently as in real time applications.

In order to speed up transmission, the telegrams are kept small by avoiding unnecessary overhead information. To reduce the busload, the correct transmission of a PDO telegram is not confirmed.

The content of each PDO message is defined in advance, as well as the location from where it is sent (which object of which node) and the target location. This configuration is called *mapping*, which also specifies how often a PDO message is sent: periodically or triggered by some event. The PDO mapping can be found in the object dictionary of the devices involved.

The CAN PDO communication is typically used to update the process inputs of the PLC and there can be several PDO communication channels between two devices. Be aware that the cycle time of the PLC task and the update rate of data sent by PDO are two different things. In certain applications, they might need synchronization.

Remark: All the commands sent by the *EPOS Studio* are essentially based on SDO communication, as well as the PLC programming in the next chapter. Please refer to the *Programming Reference* document (Chapter 4.4.2 *Communication via Network Variables*) to gain more information on the mapping of PDO in EPOS Systems.

Service Data Objects (SDO)

The lowest priority – the highest priority values – is given to *Service Data Objects (SDO)*. SDO permit the transmission of any data size. Large messages (> 4 bytes) are automatically segmented, i.e. split into several telegrams that are sent consequently.

As in PDO communication, SDO transmission is defined between two specified nodes. There are two communication channels needed, one for each direction. The telegrams contain information which entry in the object dictionary is to be written or read. Half of the useful data content is already consumed by this overhead information.

SDO is a relatively slow communication and it is primarily intended for setting up the device. Since the motion of an individual axis takes typically several hundred milliseconds, the speed of SDO communication is sufficient for many motion control applications. The situation may be different when several axes need to be closely coordinated or synchronized.



PDO and SDO in EPOS systems

PDO channels in EPOS systems are only defined between the slave and the master. There are no predefined communication channels between slaves.

- There is one SDO channel in each direction between the *EPOS2 P* and the *EPOS2 [internal]*.
- There are 4 PDO channels to transmit and 4 PDO channels receive process data on the *EPOS2 [internal]*.
- There are 32 PDO channels to transmit and 32 PDO channels receive process data on the *EPOS2 P* CAN slave network. This allows up to 32 axis on the network.

Part 3: The PLC (Programmable Logic Controller)

In Part 2 we have explored the internal motion controller, the *EPOS2 [internal]*, with the help of the *EPOS Studio* software tool. The *EPOS studio* is made for setting up and configuring the system. This is done by applying single commands, one at a time. There is no possibility to send a sequence of commands and the *EPOS2* motion controller has no memory to store such a sequence. The *EPOS2* is what we call *online commanded*. Single motion and I/O commands are transmitted from a superior system (master) and immediately executed by the slave controller. Up to this point, the *EPOS Studio* served as the master.

The master we would like to use now is the built-in PLC. In the PLC, an application program takes care of the process control, i.e. it coordinates all the actions in the different slaves. The program reads input information, sends motion commands and sets outputs. Writing a simple process control is the task of the third part of this tutorial.

- In chapter 7 we have a first look at the programming tool and the PLC programming standard IEC 61131-3.
- In chapter 8 we write our first little PLC motion program.
- Chapter 9 deepens the knowledge of how to use the additional inputs and outputs, e.g. for homing.
- In chapter 0 we program self-defined function block and include it into the PLC motion program.

7 Starting the programming tool

The goal of this chapter is to get a first glimpse of the software tool that is used to write PLC programs. The tool is called *OpenPCS*® and made from the German company *infoteam*®. *OpenPCS* is an editor for creating programs according to the international PLC programming standard *IEC 61131-3* which is the most widely used PLC programming standard in the world. *IEC 61131-3* is developed, maintained and promoted by an organization called *PLCopen*. Thus, if you are familiar with this standard, all you have to learn is how to use the *OpenPCS* editor; if not you will learn the basics in the chapters that follow.

Since EPOS systems are CANopen devices, the first step is to set up the CANopen network with all its properties in the *EPOS Studio*. Only then should we proceed to the PLC programming. Therefore, open the *OpenPCS* from within the *EPOS Studio* (and not separately) to make sure that all the relevant information about the devices involved and about the CANopen networks are available for the *EPOS2 P* PLC programming. For instance, the programming tool needs to know about axis numbers, I/O assignation and process data objects (PDO).

For the purpose of this chapter, however, there is no particular set-up needed. The standard settings in the object dictionary of the *EPOS2* motion controller are fine, even with the modifications done in the previous chapters. Just make sure that there is no input activated by accident that creates an error, such as a limit switch.

So all we have to do is to select the *IEC-61131 Programming* tool of the *EPOS2 P*.

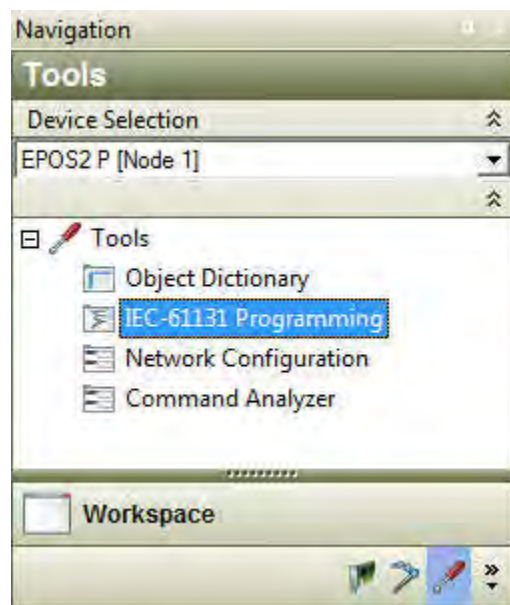


Figure 42: Open the IEC-61131 Programming tool in the EPOS Studio.

7.1 Sample project: Simple Motion Sequence

Before we start to program ourselves, we have a look at an existing program, a project called *Simple Motion Sequence*. For this purpose highlight *SimpleMotionSequence* in the *Sample Project* list and open it by clicking on the button *Open Sample Project* on the right.

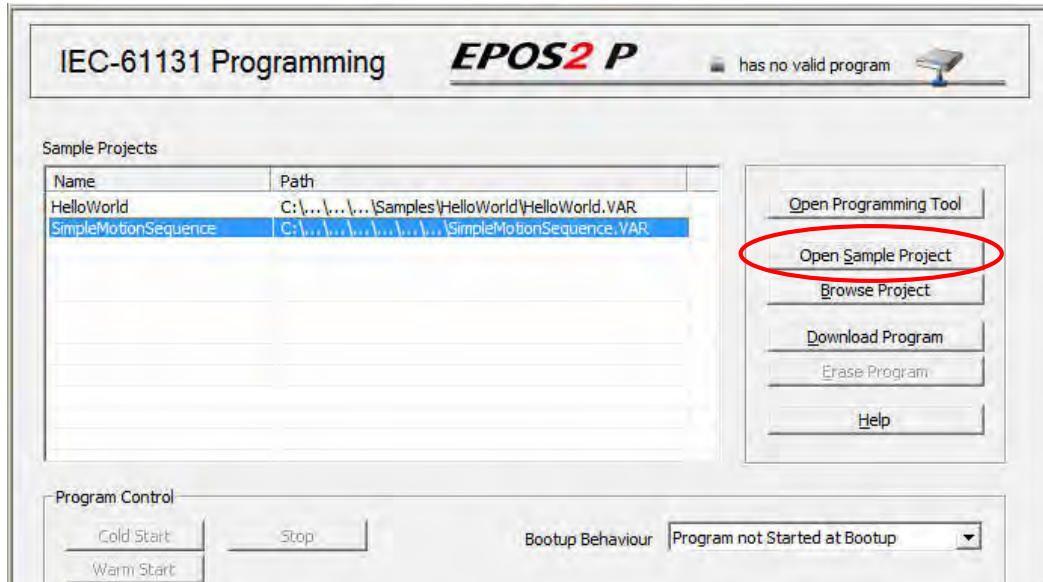


Figure 43: Open the OpenPCS tool with an existing Sample Project. Activate the corresponding project in the list and click on the button Open Sample Project on the right.

The *OpenPCS* software opens in a new window displaying the file structure of the project in the project window on the left.



Blue LED: Indication of the PLC program state

The blue LED on the *EPOS2 P* indicates the state of the program on the PLC.

- | | |
|------------------------|---|
| - fast blinking | no program is available on the PLC |
| - slowly blinking | a program is loaded on the PLC, but not running |
| - continuously shining | a program is running |

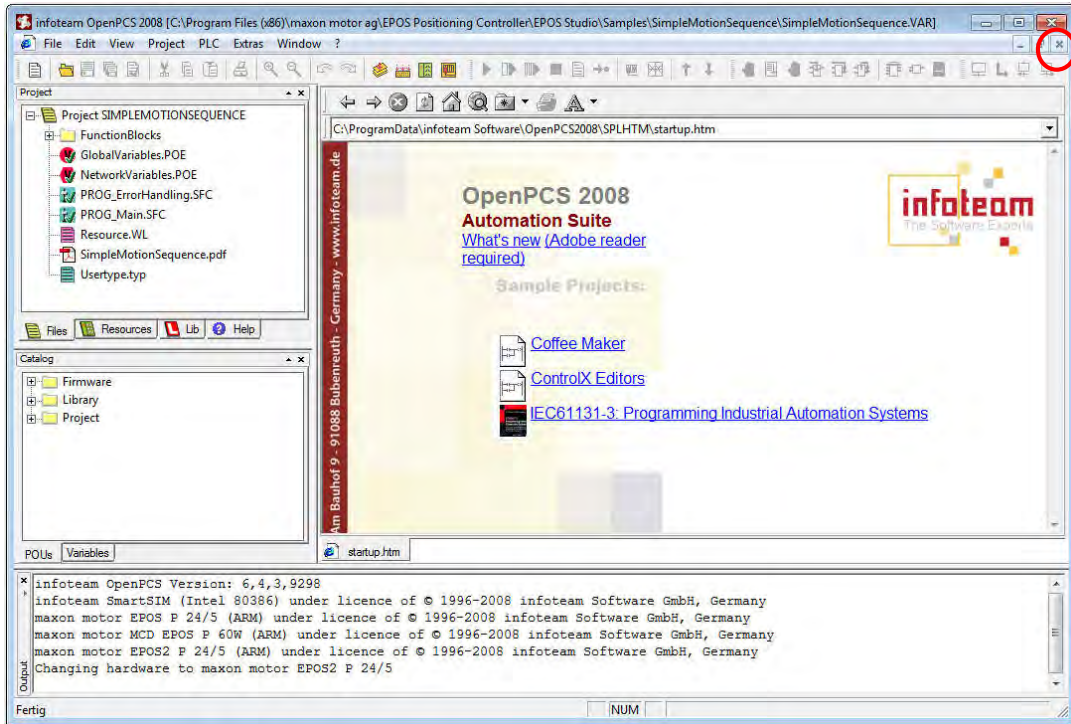


Figure 44: The main screen of the SimpleMotionSequence project. The file structure is given on the upper left. The main window displays the OpenPCS start screen which is not needed and can be closed by clicking on the small x button on the upper right (not the program!).

Before we analyze the different files we start the program and see what it does. Just follow these simple steps:

- Step 1 Compiling or *Build Active Resource*: Click on the corresponding command in the PLC menu or on the short-cut icon.

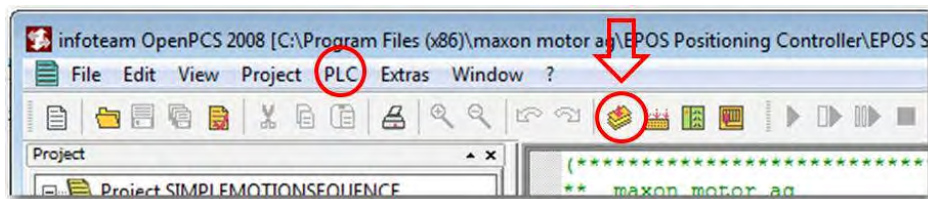


Figure 45: Build Active Resource

The result of the compilation should be a *0 error(s) 0 warning(s)* on the last line of the output section at the bottom of the screen.

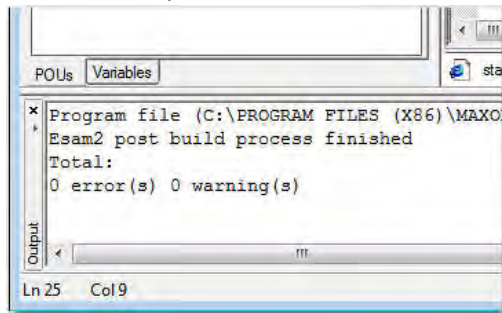


Figure 466: Compiling errors and warnings

- Step 2 Connect to the PLC and download the program on the PLC. Press the *Go Online/Offline* button in the *PLC* menu or on the short-cut icon and accept the prompting message that you want to download the program to the PLC.

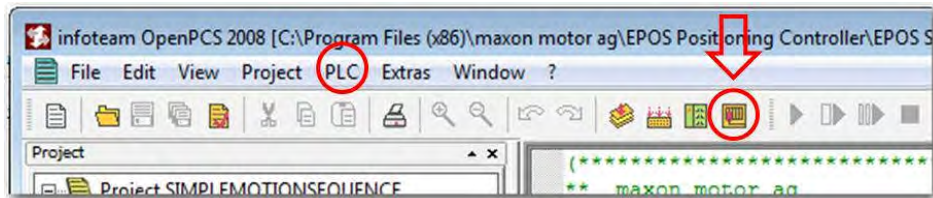


Figure 47: Go Online/Offline. Connect to the PLC.

- Step 3 Start the program. Click on *Coldstart* button in the *PLC* menu or on the short-cut icon.



Figure 48: Coldstart of the PLC program

- Step 4 Observe what the motor does. After a few seconds, the motor slowly rotates counterclockwise (CCW) followed by a fast motion in clockwise (CW) direction. Then, there is a short dwell and the motion sequence is repeated on and on.
- Step 5 Stop the program. Click on *Stop* in the *PLC* menu or on the short-cut icon.



PLC menu and short-cuts of the OpenPCS

The *PLC* menu contains all commands related to the PLC part of the *EPOS2 P* device, like establishing a connection (Go Online/Offline), compiling a program (*Build Active Resource*), up- and downloading programs, starting, stopping, and monitoring the execution of programs.

For many of these commands there is a short-cut icon just below the menu bar.

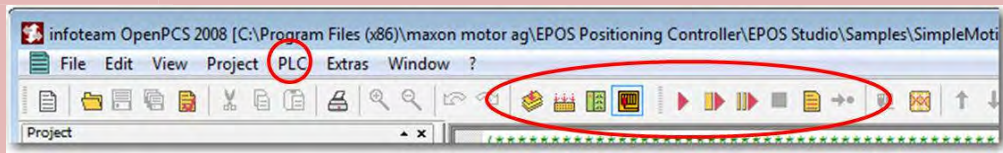


Figure 49: Menu short cuts of OpenPCS.

7.2 The project files

Figure 50 shows the file structure of the *SimpleMotionSequence* project. A particular icon represents each file type.

In short, there are files declaring variables or variable types that are used in the entire project: *GlobalVariables.POE*, *NetworkVariables.POE* and *USERTYPE.TYP*. There are two program files (*PROG_ErrorHandling.SFC* and *PROG_Main.SFC*) making use of some user defined *Function Blocks* that can be found in the corresponding folder. A project description is given in the *SimpleMotionSequence.pdf*. In addition, there is a file called *Resource.WL* that serves to follow the values of variables during operation (WL stands for watch list).

With the exception of *Resource.WL* just double-click on the corresponding file to open it in the main window.

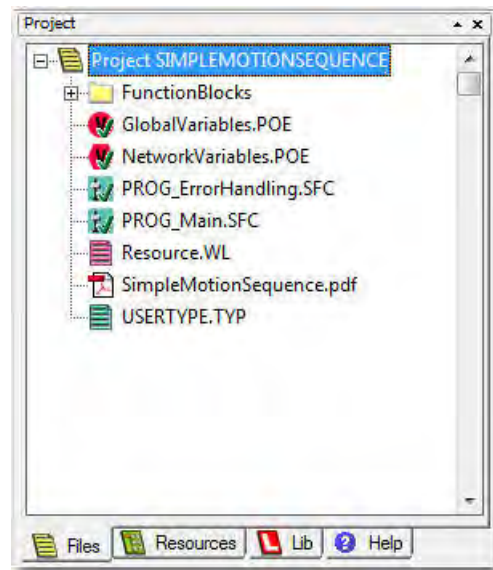


Figure 50: The project files window.

SimpleMotionSequence.pdf

We start with looking at the *SimpleMotionSequence.pdf*. This file gives a description of the project together with additional exercises for variable monitoring.

The core information about the project is contained in section 3.1. The diagram and the tables therein tell us about the structure of the main program which is constructed as a state machine. For simplicity, we ignore the error handling program all together with the error handling states in the main program and just look at the real operational motion states. Essentially, the program does the following.




Figure 51: The state machine of the SimpleMotionSequence project Modified representation highlighting the main states and the wait conditions for the transitions between the states. There is a 5s break after the initialization and a 2s dwell after each position sequence.

After program start – which is symbolized by the black dot – there is an initialization state (*Init*) that resets the axis, i.e. clears the errors, as we will later see. After a break of 5 seconds the transition to the next state (*Standstill*) is made. *Standstill* enables the axis and adds a dwell of 2 seconds before doing the transition to the state *Position Sequence*. In *Position Sequence* the CCW and CW motions are executed followed by a transition back to the state *Standstill*.

In chapter 4 of the *SimpleMotionSequence.pdf* document you find additional information about how to follow the values of variables and states and how to provoke an error and observe the error handling. However, these are topics we will not follow up any further at this point.

A remark before we continue with a closer look at the other files: *SimpleMotionSequence* is programmed in a very rigorous way and there is some effort needed to fully understand the programming structure and hierarchy especially for people that are not familiar with IEC 61131 PLC programming. The objective of this chapter, however, is not to dive deep into programming details but get to know the main elements of a PLC project.

Program files

A first element that we can recognize on the file structure is that a PLC project can have more than one program. There is a main program (*PROG_Main.SFC*) and in parallel a program that takes over whenever an error should occur (*PROG_ErrorHandling.SFC*). Both programs are written in a programming language called *Sequential Function Chart* (hence the file extension *.SFC*). SFC allows easily structuring the program as a state machine. Here, the individual states or *steps* are programmed in *Structured Text (ST)*, another programming language defined in the IEC 61131-3 standard.


```

(*****
** maxon motor ag
** All rights reserved
*****
** Project   : SimpleMotionSequence
** File      : Main State Machine
** Description: Implementation of Main State Machine
** Creation  : 15.02.2008, BRE
** Modification: 15.02.2008, BRE, Initial Version
*****

VAR_EXTERNAL
  Axis0           : AXIS_REF;
  eStateMain      : MAIN_STATE;
  eStateErrorHandling : EH_STATE;

```

```

eStateMain := MAIN_Init;
DoTrans_InitDone           := FALSE; (* Reset all Tr
DoTrans_StateStandstillDone := FALSE;
DoTrans_StatePositionSequenceDone := FALSE;
DoTrans_StateStandstillErrorDetected := FALSE;
DoTrans_StateStandstillErrorReaction := FALSE;
DoTrans_Error              := FALSE;
DoTrans_ErrorRecovery      := FALSE;
DoTrans_StatePositionSequenceErrorDetected := FALSE;
DoTrans_StatePositionSequenceErrorReaction := FALSE;

```

Figure 52: The PROG_Main.SFC file of SimpleMotionSequence.

Top: Variable declaration part.

Middle: The sequential function chart structure of the program with Steps and Transitions. The initial step is highlighted in grey.

Bottom: The program code of the initial step in Structured Text language.

We have a closer look at *PROG_Main.SFC*. Double click on the corresponding file. The main screen of the *OpenPCS* is shown in Figure 52. In the center part of the screen one finds the same state machine structure as in Figure 51. The states are represented as boxes, called *Steps*, separated by *Transitions* that contain conditions when to proceed to the following state (or step). Clicking on a step or transition displays the corresponding programming code at the bottom part of the window.

In order to explain the most important features of programming, look at the *Init* step which itself is programmed in ST (Figure 53). The first 12 lines contain organizational resets to make sure that all the possible transitions between steps are switched off. I.e. variables governing these transitions are set to FALSE. This guarantees that everything is set up properly, regardless how you enter the *Init* state, e.g. from an error recovery state.

The main action of the *Init* step is put into a customized function block called *fbInit*. The last IF...END_IF statement organizes the transition to the next state: the transition is done (i.e. the variable *DoTrans_InitDone* becomes true) if the *fbInit* has finished properly (i.e. if *fbInit.done* is true).

In summary: The code of the *Init* step shows us a lot of state machine organization and a function block *fbInit* where the real action is hidden.

```
eStateMain := MAIN_Init;
DoTrans_InitDone := FALSE;
DoTrans_StateStandstillDone := FALSE;
DoTrans_StatePositionSequenceDone := FALSE;
DoTrans_StateStandstillErrorDetected := FALSE;
DoTrans_StateStandstillErrorReaction := FALSE;
DoTrans_Error := FALSE;
DoTrans_ErrorRecovery := FALSE;
DoTrans_StatePositionSequenceErrorDetected := FALSE;
DoTrans_StatePositionSequenceErrorReaction := FALSE;

oExecuteStatePositionSequence := FALSE;
oExecuteStateStandstill := FALSE;

fbInit(Axis := Axis0, Execute := oExecuteInit);
oExecuteInit := TRUE;
IF fbInit.Done THEN
  oExecuteInit := FALSE;
  DoTrans_InitDone := TRUE;
END_IF;
```

Figure 53: Program code (without comments) of the *Init* step written in the programming language Structured Text (ST).

Function Blocks

Where can we find detailed information on *fbInit*? Since this function block contains variables it must be declared in a similar way as a variable. In the variable declaration part we notice that *fbInit* is an instance of a function block called *FB_MAIN_Init* (Figure 54). The declaration and definition of the latter can be found among others in the directory *FunctionBlocks* of the file structure.



Figure 54: *fbInit*

Top: Declaration of *fbInit* as an instance of *FB_MAIN_Init* in the variable declaration of the *PROG_Main.SFC*.

Bottom: Where to find *FB_MAIN_Init.FBD* in the file structure.

The file extension of *FB_MAIN_Init* is FBD, which stands for *Function Block Diagram*. FBD is a graphical programming language within the IEC 61131-3 standard. Opening the *FB_MAIN_Init.FBD* file, we recognize that FBD emphasizes programming by lining up the function blocks in a sequence (Figure 55). In this example, predefined function blocks are used to create a new function block. A timer function block *TON* (included in the IEC 61131-3 standard) is started once the axis is successfully reset. *MC_Reset* is a standard motion control function block that resets the axis (clears the errors and disables the power stage).

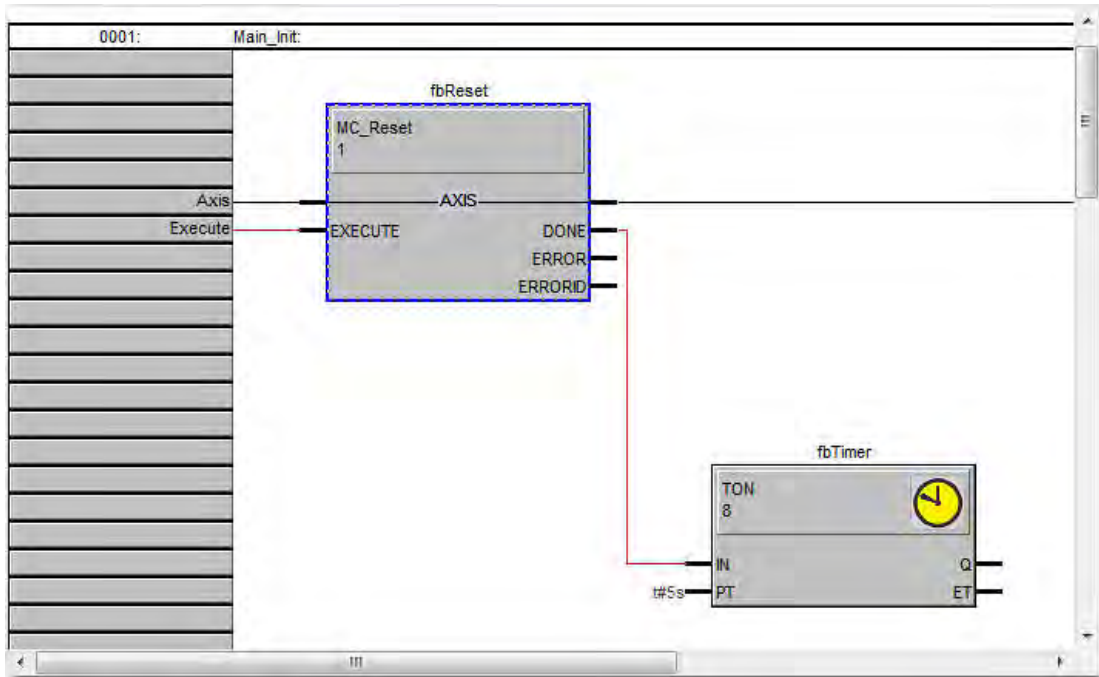


Figure 55: Program code of *FB_MAIN_Init.FBD*.

You can think of function blocks as building blocks and subroutines in a program. Function blocks have input and output parameters as can very nicely be seen in the FBD graphical programming language (Figure 55). *fbInit* is an example of a self-made function block. There are a lot of predefined function blocks that can be used as well. You can find them in the *Catalog* part of the *OpenPCS* screen on the left (Figure 56).



Color codes of programming text in *OpenPCS*

The *OpenPCS* editor identifies the reserved key words, expressions, predefined functions and function blocks from the IEC 61131-3 standards and highlights them in **blue**.

Green color is used to mark comments, which are to be set in brackets with asterisks, (* ... *).

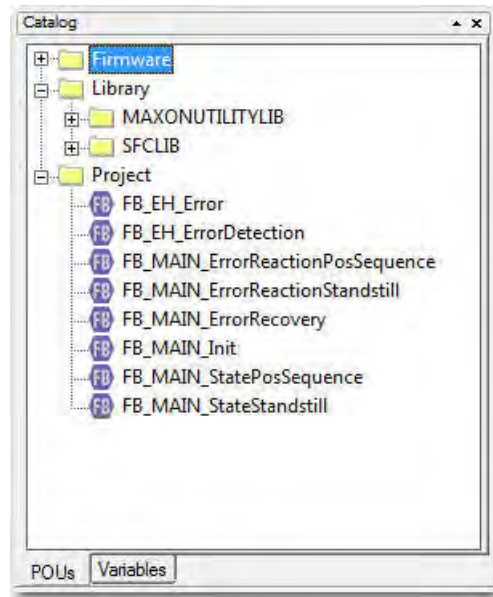


Figure 56: Catalog of available function blocks (and functions) for programming.

So, that's the end of our little tour of the *OpenPCS* software. You should now be able to ...

- know and understand the purpose of the different file types in a *OpenPCS* project.
- open and look at the different files in an existing *OpenPCS* project.
- download, start and stop an existing PLC program on the *EPOS2 P*.



Variables

Data types

The IEC 61131-3 programming standard is based on variables that are used to initialize, process and store user data. Variables need to be declared and they need to be assigned a certain data type, such as BYTE or INTEGER. The standard offers also the possibility to define your own data types. For example, you can create arrays and complex structures or – as you can find in the file *Usertype.typ* – enumeration types, i.e. data types where the variables can only take values out of a finite list.

Starting values

It is also possible to assign starting values to variables. This looks like these two examples:

- IntVar : DINT := 5;
- BoolVar : BOOL := False;

```
VAR_EXTERNAL
END_VAR

VAR_INPUT
  Execute          : BOOL;
END_VAR

VAR_IN_OUT
  Axis             : Axis_REF;
END_VAR

VAR_OUTPUT
  Done             : BOOL;
  Error            : BOOL;
  ErrorID          : DINT;
END_VAR

VAR
  fbReset          : MC_Reset;    (* Create Instance of MC_Reset *)
  fbTimer          : TON;         (* Create Instance of TON *)
  fbSelection      : MU_Selection; (* Create Instance of MU_Selection *)
END_VAR
```

Figure 57: The variable declaration part of the function block FB_Main_Init.FBD.

There are no external variables declared. The execution of this function block is started with a rising edge on the input variable Execute. As a convention, the axis number is given as an input and also as an output. The termination of the function block is signaled at the output variable Done. There are other possible output signals: The occurrence of an error during execution and an identification number for it.

The last set of declarations is for local variables, here three instances of function blocks are declared that are used in FB_Main_Init.FBD.

Variable declaration

Variables are declared in a separate section at the top of the programming window of a program or function block (Figure 52 and top part of Figure 54). They are grouped into different sets according to their use in the project.

Variables that are used globally – i.e. in different programs of a project – need to be declared in a separate file: In our *SimpleMotion Sequence* project this file is called *GlobalVariables.POE*. It contains axis numbers that are valid all over the project and variables that govern the interaction between the two programs in the project. If such global variables are to be used in a specific program they need to be “imported”, i.e. declared as external variables as can be seen at the top of Figure 54.

In function blocks, one typically finds interface variables, i.e. input variables as starting parameters when the function block is called and output variables that give the result.

And of course, there are the local variables and instances of other function blocks that are used and valid just within the particular function block. Refer to Figure 57 as an example.



EPOS Data Types

EPOS2 P systems do not support all basic data types possible in the IEC 61131-3 standard. In particular, there are no REAL types or types based on REAL possible.

For representing numbers, only INTEGER based types are allowed.

For motion control applications two special data types are predefined:

- *AXIS_REF* specifies an axis number.
- *MC_Direction* defines the direction of rotation in speed control with the two values *MCpositive* and *MCnegative*.

These two types are automatically made available (no type definition needed) if you start the *OpenPCS* software from within the EPOS Studio or if you select the *OpenPCS* project to be of the *maxon motor ag EPOS* type.

8 My first program: AxisMotion

In this chapter, you are guided through all the necessary steps to create a first PLC program for the *EPOS2 P*. We program a motion sequence with the axis at hand. As in the *SimpleMotionSequence* sample project of the previous chapter, we select *Sequential Function Chart (SFC)* as the programming language to give the general structure and use the *Structured Text (ST)* language to elaborate the individual steps. We apply predefined function blocks from the standard IEC 61131-3 library (e.g. *TON*) and from the motion control (MC) library: *MC_Reset*, *MC_Power*, *MC_MoveRelative*, *MC_MoveVelocity*, *MC_Stop*

However, be aware that our programming just concerns the motion part. There is no error handling included. In real applications, the reaction upon errors should be dealt with – a broken motor or encoder cable or any other malfunctioning can always occur. The *SimpleMotionSequence* example of the previous chapter gives an idea how error handling could be implemented.



Programming Reference

The most important document in the *EPOS Studio* file structure for programming is the *Programming Reference*; refer to Figure 4 and Figure 5 how to find it.

The *Programming Reference* contains the description of the basic programming steps, the system preparation and of the additional function block libraries which are not part of IEC 61131-3 standard. In particular, the additional libraries contain function blocks for motion control (MC), for easy handling of EPOS2 functionalities (MU = *maxon utilities*) and of general CANopen commands such as reading and writing objects and managing the CAN network.

The AxisMotion program overview

The task of the *AxisMotion* program is to perform a series of axis movements and stops. A possible sequence of steps is sketched in Figure 58.

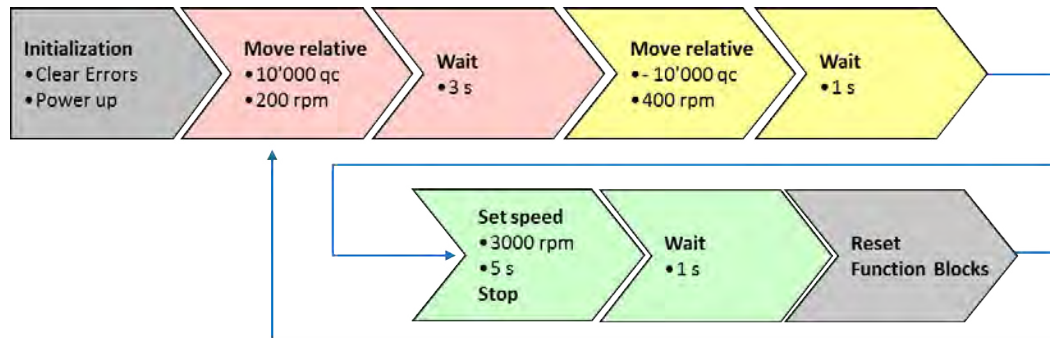


Figure 58: A possible sequence of AxisMotion serving as the base line for the programming in Sequential Function Chart. The different colors represent the different stages of completion of the programming in the next chapters.

8.1 Preparation work

Objectives	Preparing the system for programming; including axis definition, opening a new PLC programming project and defining global variables.
------------	---

Defining the Axis Number

Defining the axis number is important, since most of the function blocks in motion control refer to this number when being called. This section follows the steps described in the *Programming Reference* document (chapter 4.3.1).

Setting axis numbers is part of the *Network Configuration* and is done in the *EPOS Studio*. The information about network configuration is transferred from the *EPOS Studio* to the *OpenPCS* when you open the latter. Hence, first close the *OpenPCS* software and follow these steps to define the axis number of the internal *EPOS2*.

- Step 1 Open the *EPOS2 P* tool *Network Configuration*. (This takes a short while because a lot of information needs to be read.)
- Step 2 On the top left panel, select the *Internal Network CAN-I*.
- Step 3 On the device panel, select the *EPOS2 [internal]*.
- Step 4 Select the *Axis Number* as *Axis 1* and *Axis Type* as *Standard*.

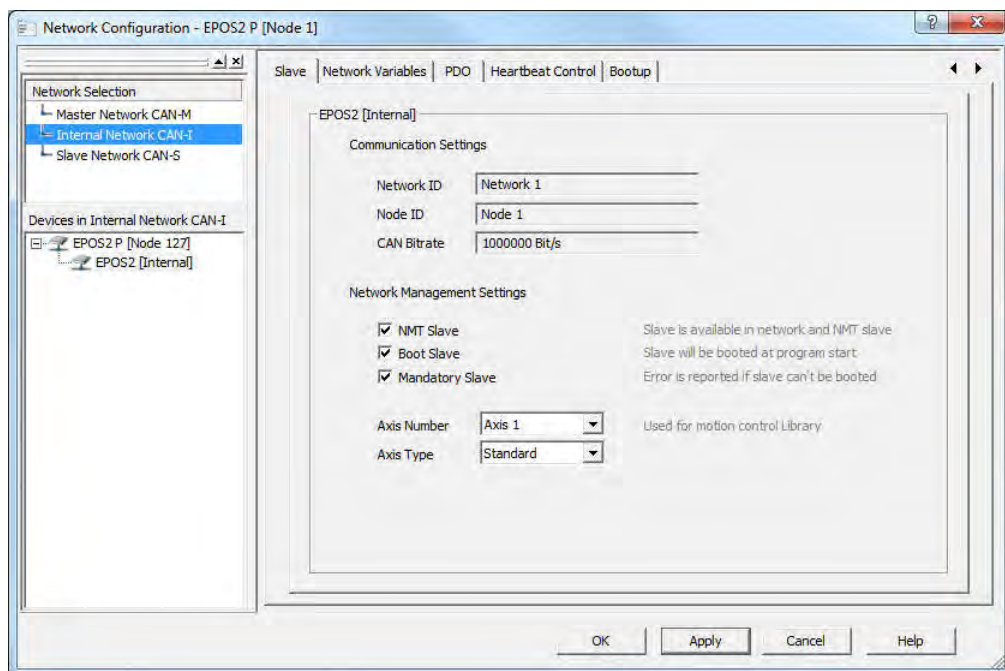


Figure 59: Configuration of the EPOS2 [internal] as axis number 1 in the Network Configuration tool of the EPOS Studio.

- Step 5 Confirm with OK. (Again, writing the new configuration takes a little bit of time.)

Creating a new OpenPCS project

Now we return to the PLC programming and create a new project. This is done in the *OpenPCS* programming tool. The subsequent recipe follows chapter 3.4 of the *Programming Reference* document.

- Step 1 In the *EPOS Studio*, open the tools for the *EPOS2 P*. Select the *IEC-61131-3 Programming* tool.
- Step 2 Open the *OpenPCS* software by clicking on the *Open Programming Tool* button.
- Step 3 Create a new *OpenPCS* project: Select *New...* in the *Project* menu.
- Step 4 In the dialog window select the file type *maxon motor ag* and the template project *EPOS2 P*. Give the project a name (e.g. *Axis_Motion*) and define the path where to save it. Confirm by pressing *OK*.

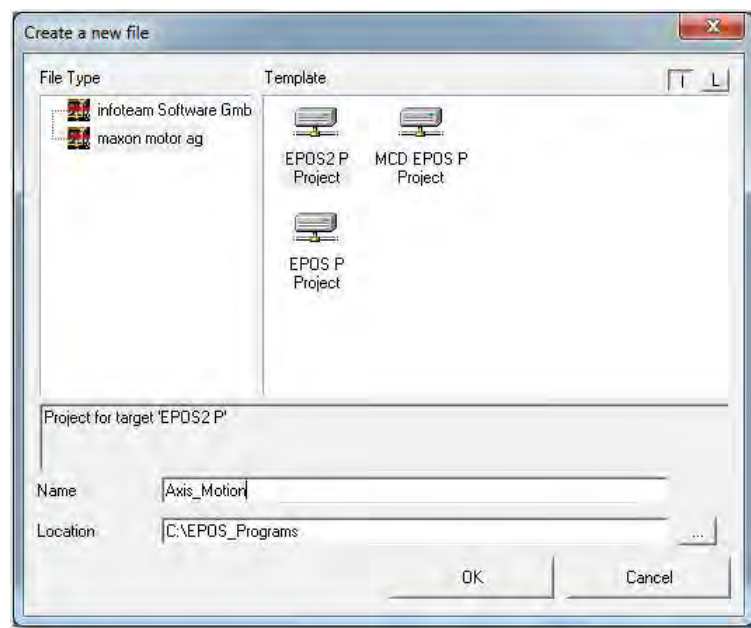


Figure 60: Creating a new PLC project *Axis_Motion* in *OpenPCS* for the *maxon EPOS2 P*.

In the project window on the upper left a new project appears with just one file: An empty variable type declaration file.

Adding Global Variables

Next we define the axis number as a global variable in a corresponding *Global variable* file. (See also chapter 3.5 of the *Programming Reference* document.)



Global variables

It is useful to declare variables that may be used in several programs as *global*. However, it is not necessary. We could have declared the axis in the variable declaration section of the program itself.

- Step 1 From the menu *File / New...* select *Declarations* and then *Global*. Give the file a name, e.g. *GlobalVar*.

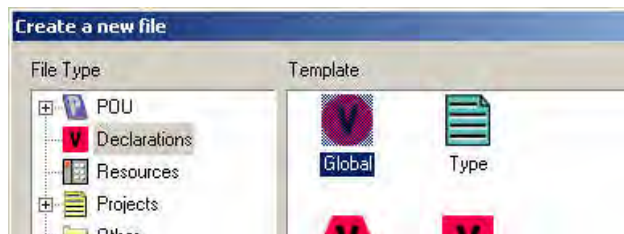


Figure 61: Creating a new variable declaration file.

Prompt the dialog with yes and add the file to the *Active Resource*; so, it will be included in the compilation process. In our case, the *Active Resource* is the PLC on the *EPOS2 P*. The *GlobalVar.POE* file is added to the project and opened in the main screen.

- Step 2 Define an axis variable, e.g. *Axis1*. This variable must be of the complex type called *AXIS_REF* which is a preset type for axes in EPOS systems. Assign the axis number 1 (as defined in the *EPOS Studio* network setup) to the variable. The syntax must look as:

```
VAR_GLOBAL
  Axis1      : AXIS_REF := (AxisNo := 1);
END_VAR
```

- Step 3 Save the file and check syntax. You find the corresponding commands in the *File* menu or as short-cut icons in the menu bar. (Checking the syntax always saves the file first).

The result of the syntax check is displayed on the lower right of the screen. If you are lucky, it reads *0 warnings, 0 errors*. If not, compare carefully with the sample code above.

A signpost with a grey vertical post and an orange arrow pointing to the right. The text "Best Practice" is written in white on the orange arrow.

Best Practice

Syntax check

Check the syntax every now and then. This avoids that you have to search any error in too large a section of program code. When writing in the *Structured Text* (ST) programming language, the individual commands need to be separated by a semicolon. Returns, spaces and indentations are ignored but they are very useful for structuring the program and make it easier to read. ST is not case sensitive. Thus, you can use upper or lower case at will for further improving the readability of the program code.

Syntax errors

Sometimes a long list of errors appears although only one mistake was made and it can be difficult to find it. Scroll to the first error in the list and double click on it. The cursor jumps to the program code line where the compiler found the error. Check the lines before and after for the syntax error.

After correcting and repeating the syntax check, the error list should be empty (except there is some more mistake).

8.2 Programming the basic steps

Objectives	Programming a simple axis motion and handling of variables and function blocks.
------------	---

Adding a new program file

(See also chapter 3.5 of the *Programming Reference* document.)

Open the *Create a new file* dialog from the menu *File / New....* Select *Program* from the *POU-Type* and *SFC* as the *IEC Language*. Give the program file a name, e.g. *MyAxisMotion*.

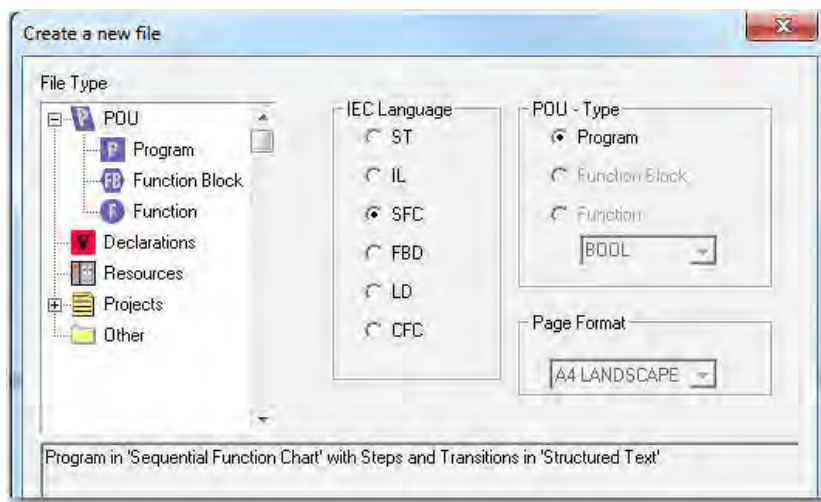


Figure 62: Creating a new file.

Here a program file is created with IEC Language Sequential Function Chart (SFC) with steps and transitions programmed in Structured Text (ST).

Prompt the dialog with yes and add the file to the active resource. The *MyAxisMotion.SFC* file is added to the project and opened on the main screen. The SFC program window already contains a first step, called *Init*, and a transition.



Libraries of Function Blocks for Motion control with *EPOS2 P*

As noted earlier, many predefined functions and function blocks for different purposes are made available for programming. Besides the general libraries standardized within IEC 61131-3 and some CAN related function blocks, there are two sets of special function blocks for motion control with *EPOS2 P*.

Motion Control Library (MC_)

The function blocks in this library start with MC_ and are described in detail in the document *Programming Reference*. The MC_library is consistent with the PLCopen standard. The most important function blocks are

- *MC_Reset* clears errors and disables the power stage
- *MC_Power* enables the power stage
- *MC_MoveRelative* moves the axis relative to the actual position
- *MC_MoveAbsolute* moves the axis to an absolute position
- *MC_Home* executes a homing
- *MC_MoveVelocity* used for speed control
- *MC_Stop* used for stopping the axis

Maxon Utility Library (MU_)

The function blocks in this library start with MU_ and are described in detail in the document *Programming Reference*. The MU_library contains function blocks that are specially made for *EPOS2* controllers and using them in the project needs special activation (see Figure 63).

The most important MU_function blocks are used for

- setting homing parameters,
- reading inputs, setting outputs; particularly for special IO functionalities,
- special operation modes, such as analog set value, *Interpolated Position Mode*, *Master Encoder Mode*.

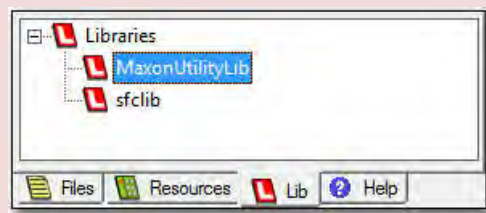


Figure 63: How to make the maxon Utilities Library available. Open the Lib tab, activate the use of the MaxonUtilityLib in the context menu.



Program Organization Units POU

POU are the building blocks for structuring a PLC project. There are three types of POU: *Programs* (PROG), *Function Blocks* (FB) and *Functions* (FUN). Common to all three POU is that they are self-contained. They have a closed shell and “communication with the outer world” is done by interface variables, that can be defined. POU can be organized in libraries, made available for re-use and allow the modularization of the program.

Program (PROG)

PROG is the main program where all the allocation to the periphery of the PLC, e.g. hardware inputs and outputs is done. On a PLC resource – i.e. the CPU – different programs can run at the same time. Each program is given a task, i.e. a definition of how the program is to be executed and with which priority. There are three possible settings of program execution

- **cyclic**: Command after command is executed until the end. Then the program restarts using the variable values from the previous run. Cyclic execution is the default setting.
- **timer**: The timer execution restarts the program at fixed intervals, e.g. every 10ms. For stable operation make sure that the previous run has finished in all situations.
- **interrupt**: Interrupt tasks only start in special situations. For example when the PLC is started or stopped, when an error occurs or when a CAN sync command is detected.

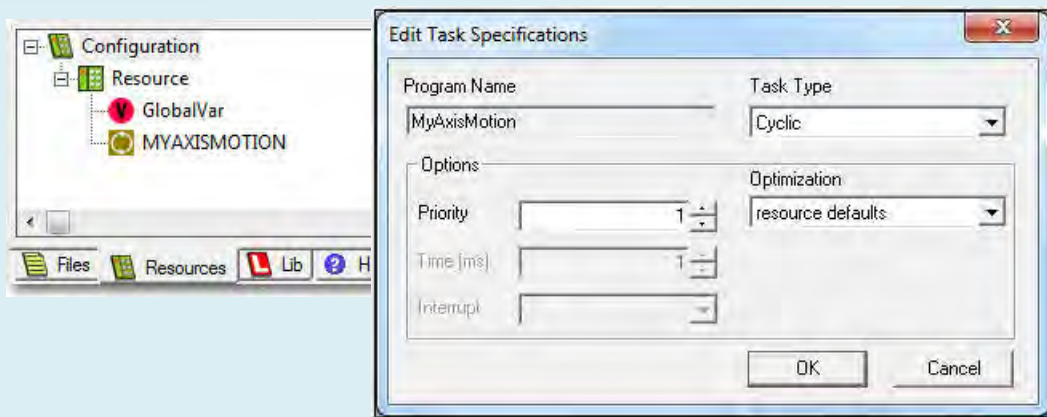


Figure 64: How to edit the task specifications of a program. Open the Resources tab, select Properties in the context menu of the program (right mouse click on MYAXISMOTION).

Important to note here is the inherently cyclic manner of PLC program run. Program inputs are read at the start of every cycle. Outputs are set at the end of each run. How long it takes to finish a program cycle depends strongly on the programming language used and the way it is programmed. In particular, the use of conditions (if..then..else constructions, case statements, jumps or steps in SFC) can limit the amount of code to be executed and speed up the run of each cycle.

Function Block (FB)

Function Blocks are probably the most widely used POUs. They act in a similar way as subroutines. FB work with their own variable set (they have a memory) and need to be declared in a similar way as variables. This is called creating an *instance* of a FB. FB are called with input variables and the outcome of the FB is stored in output variables that can be used in the calling POU.

FB can be created by the programmer himself or he uses predefined function blocks from standard or self-made libraries. Standard FB can be found in the *Catalog* part of the *OpenPCS* screen on the left (see also Figure 50).

A function block that starts execution upon the **rising edge** of one of the input variables (e.g. *Execute*) is started only if this variable was false on the previous cycle run and now is true. Be aware that the PLC continues with the execution of the next command not waiting for the FB to be finished. Some FB take quite a long time to finish, think of a positioning move that can take several seconds. During this time, the PLC program can run through many cycles. Whenever it comes back to the function block started previously, it will not reexecute or finish it, as long as the execute input variable remains true. Hence, it is important that the function block *Execute* input remains at a high level until one of the FB output signals that the execution of the FB is finished (usually called *Done* output). If the same instance of the FB is to be used again it must be reset, i.e. called with *Execute* set to *False*. This will also reset the output *Done* to false. Most of the predefined Motion Control and Maxon Utilities function blocks are started with a rising edge, e.g. *MC_MoveRelative* (positioning).

There are other predefined FB that react on the **state** of one of the input variables (e.g. called *Enable*). They are executed in each program cycle when the *Enable* state is high. Important examples for this behaviour are *MC_Power* (enables the power stage) and FB that read physical inputs, e.g. *MU_GetDigitalInput*.

The execution of self-defined FB depends on the way they are programmed.

Function (FUN)

Function (FUN) have no memory. They are executed in every cycle of the PLC program and immediately return a result value. Functions are often used for mathematical operations. Typical examples are trigonometrical functions, comparison of variable values or conversion functions that change variable types.

Programming the step Init

As suggested by its name, we would like to use this first step for axis initialization. The axis has to be reset, i.e. the power stage is disabled and any errors are cleared. This is good practice, because this clears the field for subsequent programming. There is a predefined motion control function block that performs this task: *MC_Reset*.

Still in the *Init* step the axis is enabled. Again, we can use a predefined motion control function block called *MC_Power*.

Here is a step-by-step recipe for the *Init* step starting with variable declaration.

- Step 1 Declare the *Axis1* as an external variable. That's like telling the *MyAxisMotion* program that *Axis1*, which was declared as a global variable, is going to be used. References to global variables have to be declared in the *VAR_EXTERNAL ... END_VAR* section. The declaration needs the variable name (*Axis1*) and the type (*AXIS_REF*). Do not assign an axis name; this has already been done in the global variable declaration!
- Step 2 In the *VAR ... END_VAR* section, declare instances of the function blocks that we use: *MC_Reset* and *MC_Power*. A good practice is to start the instance names with *fb* to indicate that these are function blocks and not ordinary variables. In the next figure, you can see an example. Don't forget the semicolons at the end of the lines!
- Step 3 Click on the *Init* step in the central part of the programming window. Now you can write the program code for this step in the lower part. It's simple: First call the *fbReset* and once the reset is finished power the axis with *fbPower*.
fbReset is started with a rising edge at the input variable *Execute*. It makes sense to start *fbPower* only if *fbReset* has finished properly which is signaled at the output *Done*.
- Step 4 At last the transition to the next step is prepared with the help of a Boolean variable named, for example, *DoTransInit*. Only if the motor is powered, i.e. the *Status* output of *fbPower* is at a high level, the program should proceed to the next step.
Don't forget to declare *DoTransInit* as a Boolean variable. (If you like, you can assign a starting value *false* to it, but it is not necessary since that's the default starting value anyway.)
- Step 5 Maybe it's time we had the syntax checked.

```

VAR_EXTERNAL
  Axis1      : AXIS_REF;
END_VAR

VAR_GLOBAL

END_VAR

VAR
  fbPower    : MC_Power;
  fbReset    : MC_Reset;

  DoTransInit : bool := false;
END_VAR

```

```

fbReset(Axis := Axis1, Execute := true);
if fbReset.done then
  fbPower(Axis := Axis1, Enable := true);
end_if;

if fbPower.status then
  DoTransInit := true;
end_if;

```

Figure 65: Variable declaration and program code of the Init Step.



Best Practice

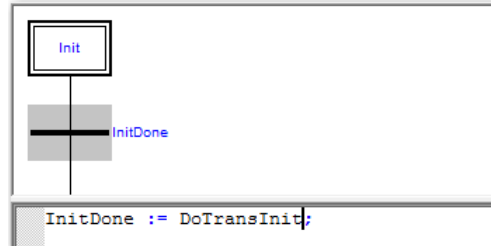
Catalog of Function Blocks and Functions

As mentioned earlier you can find the predefined functions (FUN) and function blocks (FB) in the *Catalog* tab on the left (see Figure 56). It is worth for a good programmer to spend some time studying the available predefined function blocks and functions. It may save you a lot of time later. You can get a description of most FUN and FB in the catalog simply by double clicking on it in the catalog tab. This will open the corresponding help entry.

An easy way to introduce FUN and FB is to drag them from the catalog and drop them at the place in the program code where you need it. This gives you a complete interface with all input and output parameters: remove the ones that you do not need. Don't forget to declare and name FB instances properly. (With FUN you do not need to do this since they are not instantiated.)

And here is the recipe for programming the transition:

- Step 1 First, rename the transition, e.g. *InitDone*, in order to make the program easier to read. Double click on the transition symbol and change the name. You can even add a comment.
- Step 2 Formulate the condition for the transition to be done. In our case, this should happen when *DoTransInit* has become true.



- Step 3 Save and check syntax again!

With this, you have finished programming your first step and transition.

Step Move1 and Transition

The goal of this step is to move by 10'000 quadcounts (equals 5 turns of the motor shaft) at a speed of 200 rpm. Add a short break interval after the motion has finished. This makes it easier to follow the different movements of the motor.

- Step 1 Add a step and transition:
Mark the line below the transition *InitDone*.
Select from the menu *Insert* the command *Step/Transition*.

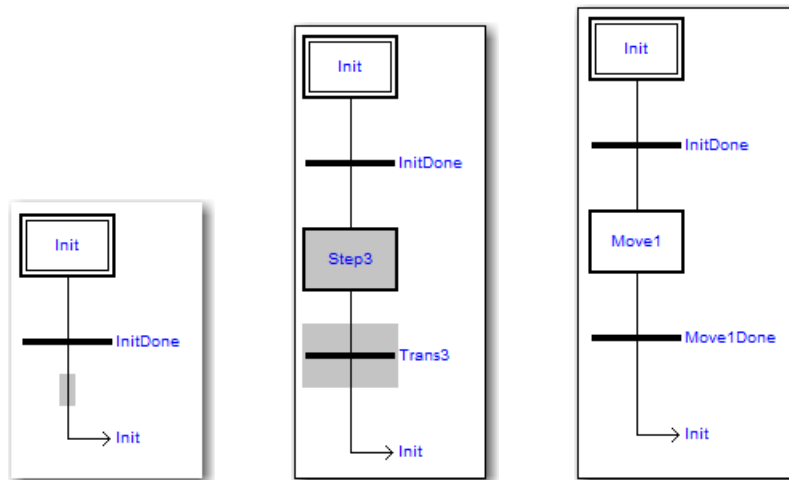


Figure 66: Adding steps and transitions in SFC.

Mark the location where you want to enter a new element. Select from the Insert menu the corresponding element. Rename the elements and add comments by double clicking on them.

- Step 2 Name the step, e.g. *Move1*, and the transition, e.g. *Move1Done*.
- Step 3 First, we write some organizational program code in *Move1* to clean the table, so to say.
If ever we jump back to the *Init* step we would like the step to work as if we started the program anew. For this purpose reset the *fbReset* function block of the previous step; i.e. call it with *Execute := False*. Also reset the Boolean variable needed for the transition *DoTransInit := False*.
- Step 4 Program *Move1*:
Add a position move. Declare an instance of the motion control function block *MC_MoveRelative* in the variable section. Name it *fbMove1* for example.
Add a short break after the motion has finished, i.e. *fbMove1.done*. Use one of the predefined timer function blocks of the IEC 61131-3 programming standard, e.g. *TON*. Name it *fbWait1* for example. You find *TON* in the catalog on the left; double clicking gives you the information how it works. Note the format of the time variable, *#3s*.

```

(* Reset the function blocks and transition flags *)
fbReset(Axis := Axis1, Execute := false);
DoTransInit := false;

(* Move and dwell *)
fbMove1(Execute := true, Axis := Axis1 , Distance := 10000,
        Velocity := 200, Acceleration := 40000 , Deceleration := 40000);

If fbMove1.done then
    fbWait1(IN := true, PT := t#3s);
end_if;

(* Set transition flag *)
DoTransMove1 := fbWait1.Q;

```

Figure 67: Program code of the step Move1.

- Step 5 Program the transition *Move1Done*. The transition is only to be executed when *fbWait1* has elapsed, i.e. the output Q has switched to a high state.
- Step 6 Save and check syntax!



Best Practice

Limits of Motion Parameters

While programming motions, be careful not to exceed the maximum settings of velocity, acceleration and position of each axis. These limitations are defined in the *Object Dictionary* of the EPOS2 [internal]: *Min Position Limit, Max Position Limit, Max Profile Velocity, Max Acceleration*.

Testing

We can try to start our little program and see what happens. Follow the steps described in chapter 7.

- Step 1 *Build Active Resource*: Click on the corresponding command in the PLC menu or on the short-cut icon. The result of the compilation should be a *0 error(s) 0 warning(s)*
- Step 2 *Go Online/Offline*: Press button in the PLC menu or on the short-cut icon.
- Step 3 *Coldstart* the program.
- Step 4 Observe what the motor does!

At first, everything goes as expected. The axis is enabled and the motor shaft rotates 5 turns. After a few seconds standstill however, the axis is continuously disabled and enabled in a very fast cycle.

Explanation: This behavior can easily be understood. Once the motion and the dwell are over for the first time, the program jumps back to the *Init* step. There, a reset is done (with power off) followed by power on. Then the program enters the step *Move1* for the second time. However, nothing happens! The function block instances *fbMove1* and *fbWait1* are not started again because they have not been reset (*Execute* is true all the time). Even the *Move1Done* transition is always true and the program jumps directly back to the *Init*, where the axis is reset and powered again, and on and on. This is exactly what we observe.

Note: In opposition to the FBs in the step *Move1*, the *fbReset* in the step *Init* is executed every time, because it is reset at the beginning of the step *Move1*.

- Step 5 So, you had better stop the program with the *Stop* button!

Resetting the function blocks

If *Move 1* is to be repeated again and again, we need to reset the function block instances *fbMove1* and *fbWait1*. This is done best in a subsequent step. Hence,

- Step 1 Add another step and transition after the *Move1Done* transition.
- Step 2 Name the step, e.g. *ResetFB*, and the transition, e.g. *ResetFBDone*.
- Step 3 Programming of the step *ResetFB*:
Reset the function blocks of the previous step by calling them with execution *false*.
Also, reset the Boolean variable needed for the previous transition.
Define a Boolean variable that governs the next transition, e.g. *DoTransResetFB*. The transition is to be done if all the function blocks are reset.
- Step 4 Program the transition *ResetFBDone*.
- Step 5 Modify the code so that only the motion is repeated:
Set the *Jump* to point to step *Move1*. Double click on the arrow after the last transition and rename the *Jump* to *Move1*.
In *Move1*, don't forget to reset the Boolean variable needed for the transition from *ResetFB*.
- Step 6 Save and Check syntax.
- Step 7 *Build Active Resource, Go Online/Offline* and *Coldstart*. Observe the motor motion.
- Step 8 Stop the program with the *Stop* button.

```
(* Reset the function blocks and transition flags *)
fbReset(Axis := Axis1, Execute := false);
DoTransInit := false;
DoTransResetFB := false;

(* Move and dwell *)
fbMove1(Execute := true, Axis := Axis1, Distance := 10000,
        Velocity := 200, Acceleration := 40000, Deceleration := 40000);

If fbMove1.done then
    fbWait1(IN := true, PT := t#3s);
end_if;

(* Set transition flag *)
DoTransMove1 := fbWait1.Q;
```

```
(* Reset the function blocks and transition flags *)
fbMove1(Execute := false, Axis := Axis1);
fbWait1(IN := false);
DoTransMove1 := false;

(* Set transition flag *)
DoTransResetFB := not fbWait1.Q and not fbMove1.done;
```

Figure 68: Program codes of the step *Move1* (top) and *ResetFB* (bottom).

8.3 Programming more motion

Objectives	Programming a position sequence and using the function blocks for speed control.
------------	--

Back and forth motion

Let us extend the program *MyAxisMotion* so that the axis moves back to the initial position and waits there for 1s. There are two ways to achieve this: Either we introduce a step and transition after *Move1* with similar content, e.g. called *Move2*. Alternatively, we can include the second motion into the step *Move1*; that's what we are going to do here.

For either case, we need second instances of *MC_MoveRelative* and *TON*, which we declare as *fbMove2* and *fbWait2*, respectively.

Program modifications step by step:

- Step 1 Declare second instances of *MC_MoveRelative* and *TON*. Name them as *fbMove2* and *fbWait2*, respectively.
- Step 2 In step *Move1*, add the second motion (negative distance, different velocity ...) after the first dwell time has elapsed.
- Step 3 In step *Move1*, add the second dwell after the second motion has finished.
- Step 4 In step *Move1*, modify the condition for the transition *Move1Done*: Dwell time 2 has elapsed.
- Step 5 In step *ResetFB*, reset the additional function blocks of the previous step. And reprogram the condition for the transition: All function blocks to be properly reset.
- Step 6 Save and check syntax.
- Step 7 *Build Active Resource*, *Go Online/Offline* and *Coldstart*. Observe the motor motion.

```

VAR
    fbReset          : MC_Reset;
    fbPower          : MC_Power;
    fbMove1, fbMove2 : MC_MoveRelative;
    fbWait1, fbWait2 : TON;

    DoTransInit      : bool := false;
    DoTransMove1     : bool := false;
    DoTransResetFB   : bool := false;
END_VAR

Move1

----- MovesDone

(* Reset the function blocks and transition flags *)
fbReset(Axis := Axis1, Execute := false);
DoTransInit := false;
DoTransResetFB := false;

(* Move forth and dwell *)
fbMove1(Execute := true, Axis := Axis1 , Distance := 10000,
        Velocity := 200, Acceleration := 40000 , Deceleration := 40000);
If fbMove1.done then
    fbWait1(IN := true, PT := t#3s);
end_if;

(* Move back and dwell *)
If fbWait1.Q then
    fbMove2(Execute := true, Axis := Axis1 , Distance := -10000,
            Velocity := 400, Acceleration := 40000 , Deceleration := 40000);
end_if;
If fbMove2.done then
    fbWait2(IN := true, PT := t#1s);
end_if;

(* Set transition flag *)
DoTransMove1 := fbWait2.Q;

```

Figure 69: How the modified program code of step Move1 could look like.

Motion at constant speed and stop.

The next extension of the *MyAxisMotion* program illustrates how more axis motion can be included by means of additional steps and transitions. It also introduces two more predefined motion control function blocks: one for setting a constant velocity (speed control) and another one for stopping the axis. The speed control section is to take place after the back and forth positioning of *Move1*.

Here are the possible programming steps.

- Step 1 Insert a new step and transition after *Move1*. Name it e.g. *Move2* and *Move2Done*, respectively.
- Step 2 Define a new transition flag variable, e.g. *DoTransMove2*, and adjust the resetting of the different transition flags wherever it is necessary.
- Step 3 In step *Move2*, set a speed to axis 1:
Declare and use an instance of the predefined motion control function block *MC_MoveVelocity*. Name it e.g. *fbSpeed1*.
Remark: The velocity value is given by an unsigned double integer variable. The direction of the movement has to be given by a predefined enumeration type of variable: *MCpositive* or *MCnegative*.
- Step 4 In step *Move2*, stop the axis after a certain amount of time has elapsed.
For stopping, declare and use an instance of the predefined motion control function block *MC_Stop* (named e.g. *fbStop*)
Remark: You could also use another instance of *MC_MoveVelocity* with the velocity set to 0 rpm.
- Step 5 In step *Move2*, add another dwell after the axis has come to a stop.
- Step 6 Don't forget to reset all the used function blocks in step *ResetFB*.
- Step 7 Save and check syntax.
- Step 8 *Build Active Resource, Go Online/Offline* and *Coldstart*. Observe the motor motion.

```

VAR
    fbReset          : MC_Reset;
    fbPower           : MC_Power;
    fbMove1, fbMove2 : MC_MoveRelative;
    fbSpeed1          : MC_MoveVelocity;
    fbStop            : MC_Stop;
    fbWait1, fbWait2 : TON;
    fbWait3, fbWait4 : TON;

    DoTransInit      : bool := false;
    DoTransMove1     : bool := false;
    DoTransMove2     : bool := false;
    DoTransResetFB   : bool := false;
END_VAR

Move2
-----
Move2Done

(* Reset the transition flags *)
Move1Done := false;

(* Set a speed *)
fbSpeed1(Execute := true, Axis := Axis1, Direction := MCnegative,
         Velocity := 3000, Acceleration := 60000, Deceleration := 60000);
If fbSpeed1.InVelocity then
    fbWait3(IN := true, PT := t#5s);
end_if;

(* Stop the axis *)
If fbWait3.Q then
    fbStop(Execute := true, Axis := Axis1, Deceleration := 80000);
end_if;
If fbStop.done then
    fbWait4(IN := true, PT := t#1s);
end_if;

(* Set transition flag *)
DoTransMove2 := fbWait4.Q;

```

Figure 70: How the program code of step Move2 could look like.

9 Homing and IOs

Objectives	An introduction to useful library function blocks for homing and input/output handling.
------------	---

In this chapter, we extend the *MyAxisMotion* program with I/O functionality and a homing procedure. The goal is to activate the axis stop by reading one of the digital inputs. The movement of the axis should be signaled by one of the outputs; the standstill of the axis by another. The homing method is *Negative Limit Switch & Index* (see chapter 4). All this involves using function blocks from *maxon utilities* (MU) library for homing and handling the EPOS2 inputs and outputs.

9.1 Configuring the IOs

First, we have to set up the digital inputs and outputs of the *EPOS2 [internal]* to our needs. As described in chapter 4, the *I/O Monitor* tool can be used. The proper operation of the *MyAxisMotion* program relies on this correct configuration.

An alternative way is to do this configuration at program start (e.g. in the *Init* step) by writing all the necessary CANopen object dictionary entries.



Slave configuration and programming

The slaves in any network need to have a suitable configuration for correct functioning within a PLC program. For instance, we have configured the *EPOS2 [internal]* in our case to match the motor and encoder type. Moreover, we have found good tuning parameters for precise load operation. In addition, the inputs and outputs can be assigned special purposes.

All this information has to be set up in the object dictionary of the slave. There are essentially two ways to do such a configuration:

- **Configuration before** mounting on the network. This can be done by downloading the correct configuration file (see chapter 6.6) on the slave.
- **Configuration at program start** as part of the. The correct parameters are written to all the slaves object dictionaries.

The second alternative has as an advantage that the correct parameter set is available, even if the slave hardware had to be replaced. For multi-axis applications, this is the preferred solution. However, there is some more programming to be done and the initialization takes longer.

In our example, we have a very simple situation with just one axis and we can keep the I/O configuration that was established at the end of chapter 4. It should look like as in Figure 71.

I/O Monitor
The **EPOS2** is disabled

HW	State	Digital Input	Purpose	Mask	Polarity	Exec Mask	Exec Trigger
<input type="radio"/>	Inactive	Digital Input 1	Negative Limit Switch	Enabled	High Active	Enabled	Rising Edge
<input type="radio"/>	Inactive	Digital Input 2	General B	Enabled	High Active		
<input type="radio"/>	Inactive	Digital Input 3	General C	Enabled	High Active		
<input type="radio"/>	Inactive	Digital Input 4	General D	Enabled	High Active		
<input type="radio"/>	Inactive	Digital Input 5	General E	Enabled	High Active		
<input type="radio"/>	Inactive	Digital Input 6	General F	Enabled	High Active		

HW	State	Digital Out...	Purpose	Mask	Polarity
<input type="radio"/>	Inactive	Digital Output 1	General A	Enabled	High Active
<input type="radio"/>	Inactive	Digital Output 2	General B	Enabled	High Active
<input type="radio"/>	Inactive	Digital Output 3	General C	Enabled	High Active
<input checked="" type="radio"/>	Active	Digital Output 4	Ready/Fault	Enabled	High Active

Value	Analog Input	Purpose	Exec Mask
4621 mV	Analog Input 1	General A	
4241 mV	Analog Input 2	General B	

Value	Analog Out...

Figure 71: The configuration of the digital inputs and outputs for the programming in this chapter.



General Purpose inputs and outputs

As noted earlier, the physical inputs and outputs belong to the EPOS2 [internal], i.e. to the motion controller. If they are not needed for the control or a functionality related to this particular axis (e.g. the limit switch in the example above), these inputs and outputs can be made available to the master program. In such a case, they have to be set as *General Purpose*.

9.2 Homing with limit switch

Since homing is to be done only once after start-up of the machine, it is reasonable to include the homing procedure in the *Init* step. First, the homing mode has to be defined, unless the *EPOS2 [internal]* has not been configured with the correct homing mode already. Then, the homing procedure can start.

Here is the necessary programming.

- Step 1 Declare instances of the function blocks *MU_SetHomingParameter* and *MC_Home*, and call the instances *fbSetHoming* and *fbHoming*, respectively.
- Step 2 Modify the sequence of operations in the *Init* step to: Reset => set homing mode => enable the power stage => execute the homing => transition to the next step. Make sure that each of the operations in the sequence is finished before calling the next.
Hint: Drag and drop the new function blocks for the homing from the *Catalog* on the left. Double click on the function blocks in the *Catalog* to get detailed information.
- Step 3 Adjust the parameters of *fbSetHoming* according to your needs. Most importantly set the *Method* to 1, which will execute a homing with the *Negative Limit Switch & Index* as you can see from the description of this function block.
- Step 4 Reset the newly introduced function blocks for homing in the step *Move1*.
- Step 6 Check the syntax and test the program. You will have to activate the limit switch on the I/O PCB during the homing.

9.3 Setting outputs according to the axis state

We would like to have the digital output 1 activated whenever the axis is in motion. The digital output 2 should signal that the axis is at standstill.

For output handling, we use the MU library function block *MU_SetAllDigitalOutput*, which sets all the outputs each time it is executed. According to the description, the states of the digital outputs are not affected by a reset of this function block; the outputs keep the digital state they are in. This allows to reset this function block immediately after use and in the next program cycle it is ready to set the outputs again. Therefore, the output states update with every program cycle.

The predefined function block *MC_ReadStatus* offers an easy access to the motion state of the axis. *MC_ReadStatus* gives the information whether the axis is at standstill or in different motion conditions (homing, discrete or continuous motion, stopping and others). In each program cycle, we have an instance of *MC_ReadStatus* to read the actual axis status information. We concentrate this information into two Boolean variables, *InMotion* and *Standstill*, and use them to trigger the settings of the digital outputs.

Here are the necessary programming steps.

- Step 1 Declare instances *fbSetDigOut* and *fbStatus* of the function blocks *MU_SetAllDigitalOutputs* and *MC_ReadStatus*, respectively.
 Declare the Boolean variables *InMotion* and *Standstill*.
- Step 2 Use *InMotion* to sum up all the states of *fbStatus* that signal the axis in some kind of motion (Hint: Look up the possible outputs of *MC_ReadStatus* in the help). Assign *fbStatus.Standstill* to *Standstill*. Activate digital output 1 whenever the axis is in motion. Activate digital output 2 whenever the axis is at standstill.
 Hint: Reset the *fbSetDigOut* immediately after calling, in order to have it updated in every program cycle. (Figure 72 shows how this could be done for the *Init* step.)
- Step 3 Check the syntax and copy this piece of code into all steps that contain motion (*Init*, *Move1* and *Move2*) to ascertain that the outputs are always correctly set.
- Step 4 Check the syntax again and test the program. Observe how the LED outputs on the PCB change when the axis is in motion or at rest.

```

(* Setting the outputs according to the Axis state *)
fbStatus(Axis := Axis1, Enable := true);
Standstill := fbStatus.standstill;
InMotion:= fbStatus.DiscreteMotion or fbStatus.Homing
           or fbStatus.ContinuousMotion or fbStatus.Stopping;
fbSetDigOut (Axis := Axis1, Execute := true,
            GenPurpA := InMotion, GenPurpB := Standstill);
fbSetDigOut (Axis := Axis1, Execute := false); (* Reset of fbSetDigOut *)

(* Reset the axis, and Homing *)
fbReset(Axis := Axis1, Execute := true);
if fbReset.done then
    fbSetHoming(Axis := Axis1, Execute := true, Method := 1, Offset := 0,
               Speedswitch := 100, Speedindex := 20, Acceleration := 500);
end_if;

if fbSetHoming.done then
    fbPower(Axis := Axis1, Enable := true);
end_if;

if fbPower.status then
    fbHoming(Axis := Axis1, Execute := true, Position := 0);
end_if;

(* Wait for DigIn2 to be activated *)
if fbHoming.done then
    fbReadDigIn2 (Axis := Axis1, Enable := true, Purpose := 14);
end_if;

(* Set transition flag *)
DoTransInit := fbReadDigIn2.state;

```

Figure 72: The Init step with homing, setting of the outputs and reading of digital input 2.

9.4 Reading inputs

We expand our program so that the program flow is triggered by physical inputs.

After the initialization, the program should wait for a digital input signal on input 2 for further execution. Similarly, the velocity mode of step *Move2* should be stopped by a rising edge of input 2 and not by an elapse of time.

Here are the necessary programming steps (compare also with Figure 72)

- Step 1 Declare *fbReadDigIn2* as an instance of the function blocks *MU_GetDigitalInput*.
- Step 2 Modify the sequence of operations in the *Init* step to: Reset => Set homing mode => enable the power stage => execute the homing => read digital input 2 => transition to the next step.
- Step 3 Adjust the *Purpose* input parameter of *fbReadDigIn2* so that the physical input 2 is read. According to our setting in Figure 71 this input is *General Purpose B*, which means that the *Purpose* input parameter of *fbReadDigIn2* must be set to 14.
- Step 4 Use the output parameter *fbReadDigIn2.state* to trigger the transition from the *Init* to the *Move1* step.
- Step 5 Reset *fbReadDigIn2* in the step *Move1*.
- Step 6 Use *fbReadDigIn2* to stop the velocity mode in step *Move2*. Essentially, it is about replacing the *fbWait3*.
Trigger the transition to the next step with *fbReadDigIn2.state*.
- Step 7 Don't forget to reset *fbReadDigIn2* in the *ResetFB* step.
- Step 8 Check the syntax and test the program.

10 Self-made Function Blocks

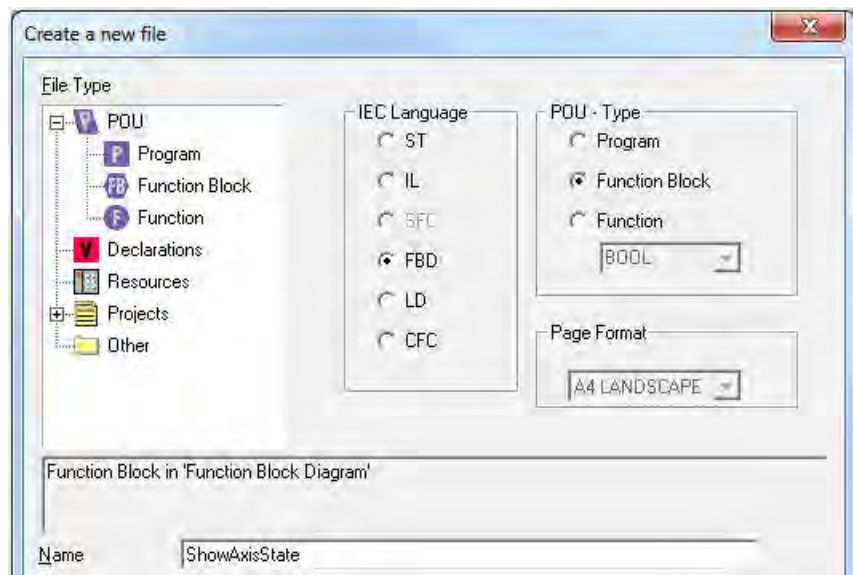
Objectives	Creating and using self-defined function blocks. Programming in <i>Function Block Diagram</i> FBD language
------------	---

The code sequence for reading the axis state and setting the digital outputs (see chapter 9.3) is used in several places in the program; actually, in each step that contains motion. Instead of copying it multiple times (with the danger of having it to modify in as many places) we can put all this action in a self-defined function block that we can call whenever needed.

As an exercise, we program the function block not as *Structured Text*, but in a graphical language called *FBD*, *Function Block Diagram*.

10.1 Creating a Function Block

Open the *Create a new file* dialog from the menu *File / New*. Select *Function Block* from the *POU-Type* and *FBD* as the *IEC Language*. Give the function block a name, e.g. *ShowAxisState*.



*Figure 73: Creating a new file in OpenPCS.
Here a function block file is created with IEC Language Function Block Diagram (FBD).*

10.2 Programming in Function Block Diagram (FBD)

Here are the steps for the programming of *ShowAxisState*. A lot of it is about handling the FBD editor.

- Step 1 Variable declaration of the function block:
As usual, the axis number is declared as an Input/Output variable.
Define an input variable *Enable*. Calling *ShowAxisState* with *Enable* in *True* state should initiate a reading of the axis status and then a setting of the outputs. Hence, we need instances of the corresponding function blocks from the libraries.
For completeness sake, we define the standard output variables *Done*, *Error*, and *ErrorID* as well, even if we do not make use of them.
Check the syntax.

```
VAR_IN_OUT
  Axis          : AXIS_REF;
END_VAR
|
VAR_INPUT
  Enable        : bool;
END_VAR

VAR_OUTPUT
  Done          : bool; (* indicates that the FB is ready again*)
  Error         : bool; (* no error management set up yet *)
  ErrorID       : DINT;
END_VAR

VAR
  fbSetDigOut   : MU_SetAllDigitalOutputs;
  fbStatus      : MC_ReadStatus;
END_VAR
```

Figure 74: Variable declaration of *ShowAxisState*.

- Step 2 Insert a *MC_ReadStatus* function block in the programming window.
Select the corresponding command from the *Insert* menu or from the context menu (right mouse click on the programming window) or use the short-cut icon at the top.
- Step 3 Give *MC_ReadStatus* the correct name as declared above, here *fbStatus*.
- Step 4 Assign the input parameters *Axis* and *Enable* to two of the fields in the left rail. Use the right mouse button on a particular field and *Insert variable....*
- Step 5 Connect these input variables with *fbStatus*.
Highlight the two end points of the connection and *Insert Connection*. Use the *Insert* menu or use the short-cut icon at the top or the short-cut *ctrl B*.

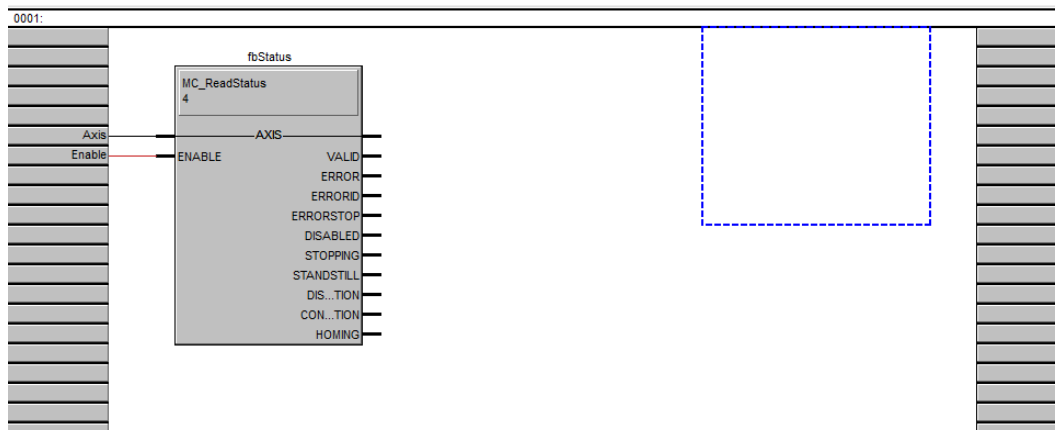


Figure 75: Programming window after Step 5 (FBD).

- Step 6 Insert an instance *fbSetDigOut* of *MU_SetAllDigitalOutputs* in a similar way as in step 2 and 3.
 Connect the *Axis* output of *fbStatus* with the *Axis* input of *fbSetDigOut*. With this connection *fbSetDigOut* is called for the correct axis.
 Connect the *Valid* output of *fbStatus* with the *Execute* input of *fbSetDigOut*. With this connection *fbSetDigOut* is called only if the axis state has been read correctly. Since *fbSetDigOut* works only with a rising edge on the *Execute* input we have to reset this function block after every use (see Step 9).
- Step 7 Connect the *Standstill* output of *fbStatus* with the *GenPurpB* input of *fbSetDigOut*. As required, the digital output 2 is set whenever the axis is at standstill.
- Step 8 Whenever the axis is in one of the states *Stopping*, *Discrete Motion*, *Continuous Motion* or *Homing* the digital output 1 should be set. For this purpose, insert the predefined function *OR_BOOL_FBD* that corresponds to a Boolean OR.
 Connect the moving state outputs of *fbStatus* with the input of the OR function. You can generate additional inputs on *OR_BOOL_FBD* with the *Append Input* command (right mouse on function).
 Connect the output of *OR_BOOL_FBD* with the *GenPurpA* input of *fbSetDigOut*.
 It might be time to check the syntax again.

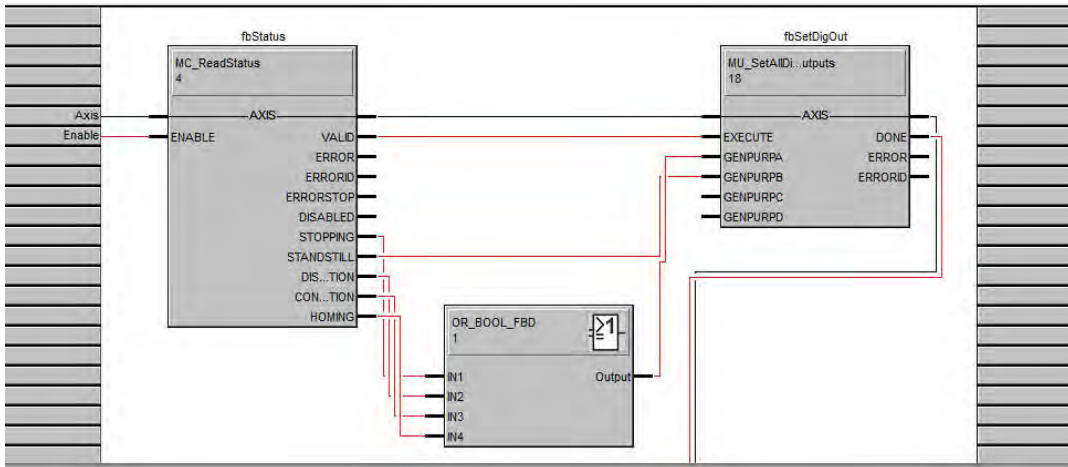


Figure 76: Programming window after Step 8 (FBD).

- Step 9 Once the outputs have been set correctly (i.e. *fbSetDigOut.done* is true) the *fbSetDigOut* must be reset, i.e. called with *Execute = False*. Insert a copy of *fbSetDigOut* and connect the *fbSetDigOut.done* with the *Execute* input of the copy and *Negate Input*. Do not forget to connect the axis as well. Check the syntax.

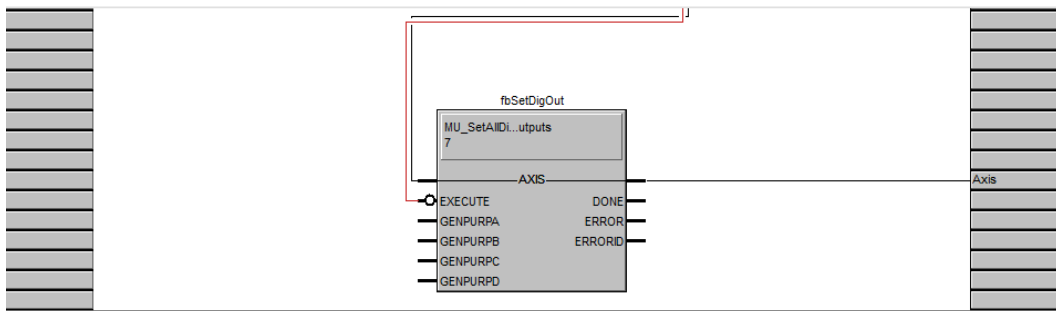


Figure 77: Programming the reset (Lower part of Figure 76).

10.3 Using self-made Function Blocks

Next, we need to adjust the main program code. Here are the necessary steps.

- Step 1 In the variable declaration of the main program, declare *fbShowAxisState* as an instance of the newly defined function block *ShowAxisState*.
- Step 2 Delete all the variables and function block instances that are not needed anymore: *fbSetDigOut*, *fbStatus*, *InMotion*, *Standstill*.
- Step 3 In all the motion steps, replace the piece of program code that sets the outputs according to the axis state with the function call of *fbShowAxisState*. The *Init* step now looks similar to Figure 78.
- Step 4 Check syntax, compile, download and see if everything works as it should.

```
(* Setting the outputs according to the Axis state *)
fbShowAxisState (Axis := Axis1, Enable := true);

(* Reset the axis, and Homing *)
fbReset(Axis := Axis1, Execute := true);
if fbReset.done then
    fbSetHoming(Axis := Axis1, Execute := true, Method := 1, Offset := 0,
               Speedswitch := 100, Speedindex := 20, Acceleration := 500);
end_if;

if fbSetHoming.done then
    fbPower(Axis := Axis1, Enable := true);
end_if;

if fbPower.status then
    fbHoming(Axis := Axis1, Execute := true, Position := 0);
end_if;

(* Wait for DigIn2 to be activated *)
```

Figure 78: Setting the axis state outputs with the function block in the Init step.

(Compared with Figure 72, only the first lines of the code are shown).

Part 4: Appendices, References and Index

11 Appendices

11.1 Motor and encoder data sheets

The maxon motor-encoder combination with part number 301782 consists of

- EC-max 30 motor with part number **272763**
- MR encoder with part number **225778**

Moto

- Stock program
- Standard program
- Special program (on request)

		Part Numbers				
		272762	272763	272764	272765	
Motor Data						
Values at nominal voltage						
1	Nominal voltage	V	12	24	36	48
2	No load speed	rpm	7980	9340	9490	9350
3	No load current	mA	302	191	130	95.4
4	Nominal speed	rpm	6590	8040	8270	8130
5	Nominal torque (max. continuous torque)	mNm	63.6	60.7	63.7	64.1
6	Nominal current (max. continuous current)	A	4.72	2.66	1.88	1.4
7	Stall torque	mNm	381	458	522	519
8	Starting current	A	26.8	18.8	14.5	10.7
9	Max. efficiency	%	80	81	82	82
Characteristics						
10	Terminal resistance phase to phase	Ω	0.447	1.27	2.48	4.49
11	Terminal inductance phase to phase	mH	0.049	0.143	0.312	0.573
12	Torque constant	mNm/A	14.2	24.3	35.9	48.6
13	Speed constant	rpm/V	672	393	266	197
14	Speed/torque gradient	rpm/mNm	21.2	20.6	18.4	18.2
15	Mechanical time constant	ms	4.86	4.73	4.21	4.17
16	Rotor inertia	gcm ²	21.9	21.9	21.9	21.9

r data sheet

1/10 maxon 30 Ø30 mm, brushless, 60 Watt

Figure 79: Motor data sheet EC-max 30.

Specifications

Thermal data

17	Thermal resistance housing-ambient	7.4 K/W
18	Thermal resistance winding-housing	0.5 K/W
19	Thermal time constant winding	2.76 s
20	Thermal time constant motor	1000 s
21	Ambient temperature	-40...+100°C
22	Max. permissible winding temperature	+155°C

Mechanical data (preloaded ball bearings)

23	Max. permissible speed	15000 rpm
24	Axial play at axial load < 6.0 N	0 mm
	> 6.0 N	0.14 mm
25	Radial play	preloaded
26	Max. axial load (dynamic)	5 N
27	Max. force for press fits (static)	98 N
	(static, shaft supported)	1300 N
28	Max. radial loading, 5 mm from flange	25 N

Other specifications

29	Number of pole pairs	1
30	Number of phases	3
31	Weight of motor	305 g

Values listed in the table are nominal.

Connection motor (Cable AWG 20)

red	Motor winding 1	Pin 1
black	Motor winding 2	Pin 2
white	Motor winding 3	Pin 3
	N.C.	Pin 4

Connector Part number

Molex 39-01-2040

Connection sensors (Cable AWG 26)

yellow	Hall sensor 1	Pin 1
brown	Hall sensor 2	Pin 2
grey	Hall sensor 3	Pin 3
blue	GND	Pin 4
green	V_{Hall} 3...24 VDC	Pin 5
	N.C.	Pin 6

Connector Part number

Molex 430-25-0600

Wiring diagram for Hall sensors see p. 35

Figure 80: Extract from Motor data sheet EC-max 30: Specifications.

Encoder data sheet

MR Encoder Type MI, 128-1000 CPT, 3 Channels, with Line Driver

	Part Numbers																												
	225771	225773	225778	225805	225780																								
<div style="display: flex; justify-content: space-between;"> <div style="width: 30%;"> <p> Stock program</p> <p> Standard program</p> <p> Special program (on request)</p> </div> <div style="width: 70%;"> <p>Type</p> <table border="1"> <tr> <td>Counts per turn</td> <td>128</td> <td>256</td> <td>500</td> <td>512</td> <td>1000</td> </tr> <tr> <td>Number of channels</td> <td>3</td> <td>3</td> <td>3</td> <td>3</td> <td>3</td> </tr> <tr> <td>Max. operating frequency (kHz)</td> <td>80</td> <td>160</td> <td>200</td> <td>320</td> <td>200</td> </tr> <tr> <td>Max. speed (rpm)</td> <td>37500</td> <td>37500</td> <td>24000</td> <td>37500</td> <td>12000</td> </tr> </table> </div> </div>	Counts per turn	128	256	500	512	1000	Number of channels	3	3	3	3	3	Max. operating frequency (kHz)	80	160	200	320	200	Max. speed (rpm)	37500	37500	24000	37500	12000					
Counts per turn	128	256	500	512	1000																								
Number of channels	3	3	3	3	3																								
Max. operating frequency (kHz)	80	160	200	320	200																								
Max. speed (rpm)	37500	37500	24000	37500	12000																								
Technical Data																													
Supply voltage V_{CC}	5 V \pm 5%																												
Output signal	TTL compatible																												
Phase shift Φ	90°e \pm 45°e																												
Index pulse width	90°e \pm 45°e																												
Operating temperature range	-25...+85°C																												
Moment of inertia of code wheel	\leq 0.7 gcm ²																												
Output current per channel	max. 5 mA																												
Pin Allocation																													
1 N.C. 2 V_{CC} 3 GND 4 N.C. 5 Channel \bar{A} 6 Channel A 7 Channel \bar{B} 8 Channel B 9 Channel I (Index) 10 Channel I (Index)																													
DIN Connector 41651/ EN 60603-13 flat band cable AWG 28																													

Figure 81: Extracts from MR Encoder data sheet.

12 References, Glossary

12.1 List of Figures

Figure 1: The content of the EPOS2 P Starter Kit.....	7
Figure 2: How to connect the EPOS2 P Starter Kit.....	10
Figure 3: Schematic overview of the EPOS2 P.....	11
<i>Figure 4: The file structure of the maxon EPOS software installation.....</i>	<i>12</i>
<i>Figure 5: The document structure of the maxon EPOS2 P.....</i>	<i>13</i>
Figure 6: Where to find the New Project icon.....	14
Figure 7: New Project Wizard, step 1.....	15
Figure 8: New Project Wizard, step 2.....	15
Figure 9: The EPOS Studio Screen with the Workspace tab open.....	16
<i>Figure 10: Error and Warning list in the Status window.....</i>	<i>16</i>
Figure 11: The Communication Tab of the EPOS Studio Screen.....	17
Figure 12: Selecting the device in the Wizards tab of the EPOS Studio screen.....	18
Figure 13: Startup Wizard, step 1.....	19
Figure 14: Startup Wizard, step 2.....	20
Figure 15: Startup Wizard, step 4.....	20
Figure 16: Startup Wizard, step 5.....	21
Figure 17: Startup Wizard, step 6.....	22
Figure 18: Startup Wizard, step 7.....	23
<i>Figure 19: The signals of an incremental encoder.....</i>	<i>26</i>
Figure 20: Regulation Tuning Wizard, Auto Tuning screen.....	28
<i>Figure 21: Regulation Tuning Wizard.....</i>	<i>31</i>
<i>Figure 22: PID controller.....</i>	<i>32</i>
<i>Figure 23: Schematic of feed-forward control.....</i>	<i>34</i>
<i>Figure 24: Master-Slave architecture with EPOS2 slaves.....</i>	<i>35</i>
Figure 25: How to open Profile Position Mode.....	36
Figure 26: Speed profiles (speed vs. time) for a position move.....	37
<i>Figure 27: Diagram of a closed-loop position control system.....</i>	<i>39</i>
Figure 28: Where to find the Application Notes.....	40
<i>Figure 29: Target reached.....</i>	<i>42</i>
<i>Figure 30: Principle of the homing mode with the index channel of the encoder.....</i>	<i>45</i>
<i>Figure 31: Velocity signals recorded on a motor with MR encoder.....</i>	<i>49</i>
<i>Figure 32: Velocity signals recorded on a motor with optical encoder.....</i>	<i>50</i>
Figure 33: Schematic representation of the Master Encoder Mode.....	59
Figure 34: Schematic representation of the Step Direction Mode.....	60
Figure 35: Physical layout of the CAN bus.....	62
Figure 36: CANopen device profile.....	63
Figure 37: Access to the Object Dictionary of the EPOS2 P in the EPOS Studio Tools tab.....	64
<i>Figure 38: The different Object Dictionaries in an EPOS2 P Network.....</i>	<i>65</i>
Figure 39: Some Standard CANopen Device Profiles (from CiA website).....	66
Figure 40: Object Dictionary of the motion controller EPOS2.....	67
<i>Figure 41: Where to change the Refresh Rate.....</i>	<i>68</i>
Figure 42: Open the IEC 61131 programming tool in the EPOS Studio.....	74
Figure 43: Open the OpenPCS tool with an existing Sample Project.....	75

Figure 44: The main screen of the SimpleMotionSequence project.	76
Figure 45: Build Active Ressource	76
Figure 46: Compiling Errors and Warnings	77
Figure 47: Go Online/Offline. Connect to the PLC.	77
Figure 48: Coldstart of the PLC program	77
Figure 49: Menu short cuts of OpenPCS.	78
Figure 50: The project files window.....	79
Figure 51: The state machine of the SimpleMotionSequence project	80
Figure 52: The PROG_Main.SFC file of SimpleMotionSequence.	81
Figure 53: Program code (without comments) of the Init step	82
Figure 54: fbInit	83
Figure 55: Program code of FB_MAIN_Init.FBD.....	84
Figure 56: Catalog of available function blocks (and functions) for programming.	85
Figure 57: The variable declaration part of the function block FB_Main_Init.FBD.	86
Figure 58: A possible sequence of AxisMotion	89
Figure 59: Configuration of the EPOS2 [internal] as axis number 1	90
Figure 60: Creating a new PLC project Axis_Motion in OpenPCS for the maxon EPOS2 P.	91
Figure 61: Creating a new variable declaration file.	92
Figure 62: Creating a new file.	94
Figure 63: How to make the maxon Utilities Library available.	95
Figure 64: How to edit the task specifications of a program.	96
Figure 65: Variable declaration and program code of the Init Step.	99
Figure 66: Adding steps and transitions in SFC.....	102
Figure 67: Program code of the step Move1.....	103
Figure 68: Program codes of the step Move1 (top) and ResetFB (bottom).....	105
Figure 69: How the modified program code of step Move1 could look like.	107
Figure 70: How the program code of step Move2 could look like.	109
Figure 71: The configuration of the digital inputs and outputs	111
Figure 72: The Init step with homing, setting of the outputs and reading of digital input 2. .	114
Figure 73: Creating a new file in OpenPCS.	116
Figure 74: Variable declaration of ShowAxisState.	117
Figure 75: Programming window after Step 5 (FBD).	118
Figure 76: Programming window after Step 8 (FBD).	119
Figure 77: Programming the reset (Lower part of Figure 76).....	119
Figure 78: Setting the axis state outputs with the function block in the Init step.	120
Figure 79: Motor data sheet EC-max 30.	121
Figure 80: Extract from Motor data sheet EC-max 30: Specifications.	122
Figure 81: Extracts from MR Encoder data sheet.	123

12.2 List of Boxes



EPOS Info

- Latest EPOS Studio 9
- Manuals and software documentation 12
- Projects in EPOS Studio 14
- Errors and warnings 16
- Motor and encoder 25
- Expert Tuning and Manual Tuning 30
- Green and red LED 37
- Restrictions on variable types of EPOS systems 65
- Object Dictionaries of EPOS2 P 65
- Firmware Specification 65
- Save parameters 69
- PDO and SDO in EPOS systems 72
- Blue LED: Indication of the PLC program state 75
- EPOS Data Types 87
- Programming Reference 88
- General Purpose inputs and outputs 111



Motion Control Background

- Commutation with brushed and brushless motors 24
- Position and speed evaluation with incremental encoder 26
- Encoder pulse count and control dynamics 27
- Feedback and Feed Forward 32
- Master, slaves and on-line single commands 35
- Enable and disable the power stage 38
- The control loops in motion control 38
- Position accuracy 42
- Maximum Following Error 43
- What is homing and why is it needed? 45
- Positioning without profile 47
- Understanding Velocity Signals on Data Recorder 49
- Runaway in current control mode 52
- Limit Switches and Home Switch 55
- Device Enable and Quick Stop 55
- Capture Input: Position Marker 55
- Ready/Fault 56
- Trigger Output: Position Compare 56
- Holding Brake 56



IEC Background

- PLC background information 70
- Variables 86
- Program Organization Units POU 96



OpenPCS Info

- PLC menu and short-cuts of the OpenPCS 78
- Color codes of programming text 84



Best Practice

- Working directory 8
- Finding the USB driver 10
- Diagram zoom 29
- Data Recorder 41
- Limiting motion parameters and damping system reaction 46
- Define your own object filter 69
- Refreshing rate of the Object Dictionary 68
- Save parameters 54
- Hardware enable 57
- Limiting and damping system reaction 58
- Global variables 92
- Syntax check 93
- Syntax errors 93
- Catalog of Function Blocks and Functions 100
- Limits of Motion Parameters
103
- Slave configuration and programming 110

12.3 Literature

- Feinmess www.feinmess.de/mt_all/glossar.htm
- John K.-H. John, M. Tiegelkamp "SPS-Programmierung mit IEC 61131-3", 3rd Edition, Springer Verlag 2000. ISBN 3-540-66445-9
- Wikipedia www.wikipedia.org
- CAN in Automation www.can-cia.org

12.4 Index

A

Acceleration
 maximum, 46
 Unit, 27
Accuracy
 position accuracy, 42
 speed accuracy, 49
Analog position control, 58, 60
Analog set value, 57, 95
Analog speed control, 57
AXIS_REF, 87

B

Brake, 56
 Holding brake, 56

C

Cabling, 10
CAN, 62
 CANopen, 62
 CANopen Device, 63
 CANopen Device Profile, 66
 CiA, CAN in Automation, 61
 Communication, 71
 Process Data Object PDO, 71
 Service Data Object SDO, 72
Capture Input, 55, 126
Closed-loop, 39
Communication, 11
Commutation, 24, 126
 Block, 24
 Sinusoidal, 24
Current
 Nominal current, 22
Current control
 Current Mode, 52

D

Data Recording, 40
Deceleration
 Unit, 27
Device Configuration File. see Electronic data sheet
Disabled, 38
Documentation, 12

E

Electronic data sheet, 70
Enable
 Device Enable, 55, 126
 Enabled, 38
 Hardware enable, 57
Encoder
 Incremental encoder, 26
EPOS Studio
 Communication Tab, 17
 Tools, 18
 Wizards, 18
 Workspace Tab, 16

F

Fault, 56
Feed-forward, 34
Firmware Specification, 65
Following error, 34
 Max Following Error, 43
Function, 97
Function Block, 97
 Enable, 97
 Execute, 97
 Resetting, 105

G

Gearhead
 Electronic gearhead, 59

H

Home Switch, 55
Homing, 44
 Current threshold, 44
 With index channel, 45

I

I/O
 I/O Monitor, 53
IEC 61131-3, 74
IEC programming language
 Function Block Diagram FBD, 84, 116
 Sequential Function Chart SFC, 80

- Structured Text ST, 80
- Index channel, 26
- Infoteam, 74
- Input
 - Analog inputs, 54
 - Digital inputs, 54
 - General Purpose*, 111, 126
- Input/Output
 - I/O Monitor, 53
- Installation
 - EPOS Studio, 8
- Instance, 97
- Interpolated Position Mode, 95

L

- LED
 - Blue LED, 75, 126
 - green LED, 37
 - red LED, 37
- Library
 - MC motion control, 95
 - MU maxon utility, 95
- Limit Switch*, 55, 126

M

- Manual, 12, 126
 - Application Notes Collection, 40
 - Firmware Specification, 12
 - Programming Reference, 12
- Mask, 53, 54
 - Execution Mask (ExecMask), 54
- Master, 11, 35, 126
- Master Encoder Mode, 59, 95
- MC_Direction, 87
- MC_library, 95
 - MC_Home, 95, 112
 - MC_MoveAbsolute, 95
 - MC_MoveRelative, 95
 - MC_MoveVelocity*, 95, 108
 - MC_Power, 95
 - MC_Reset, 95
 - MC_Stop, 95, 108
- MCnegative, 87, 108
- MCpositive, 87, 108
- Monitor
 - I/O Monitor, 53
- Motor
 - Brushed (DC), 24
 - Brushless (BLDC, EC), 24

- MU_library, 95
 - MU_GetDigitalInput, 115
 - MU_SetDigitalOutput, 113
 - MU_SetHomingParameter, 112

N

- Network Configuration, 90

O

- Object Dictionary, 64
 - EPOS2 [internal], 67
 - EPOS2 P, 65, 70
 - Index, 64
- Object filter, 69
- Online commanded, 73
- On-line commanded, 35
- OpenPCS, 74
- OSI model, 62
- Output
 - Digital outputs, 53
 - General Purpose*, 111, 126
- Output stage, 38, 39

P

- Parameter
 - Save parameters, 69
- Path generator, 39
- PLC, 11, 70
- PLCopen, 74
- Polarity of IOs, 53
- Position Compare, 56
- Position control
 - Position Mode, 46
 - Profile Position Mode, 36
- Position Marker, 55
- Power stage. *see* Output stage
- Process Data Object, 71
 - Mapping, 71
- Program, 96
 - cyclic task, 96
 - interrupt task, 96
 - task, 96
 - timer task, 96
- Program Organization Unit, 96, 127
- Programming Reference, 88, 94

Q

Quad counts, 26
Quick Stop, 55

R

Ready, 56, 126
Real-time, 68, 71
Resource
 Active resource, 92

S

Sample Project
 Simple Motion Sequence, 75
Service Data Object, 72
Setpoint Offset, 58
Setpoint Scaling, 58
Slave, 11, 35
Speed
 Maximum permissible speed, 22
Speed control
 Profile Velocity Mode, 48
 Velocity Mode, 51
State machine
 CANopen, 63
 PLC program as a, 79, 80
Step Direction Mode, 59
Stepper motor, 59
Structured Text (ST), 80
 Syntax, 93
Syntax check, 93

Syntax errors, 93

T

task. see Program
Time constant
 Thermal time constant, 22
Trigger Output, 56, 126
Tuning, 28
 Auto Tuning, 28
 Expert Tuning, 30, 126
 Manual tuning, 30

U

USB driver, 10

V

Variable, 86
 EPOS variable types, 87
 Global variables, 92
Velocity
 max. profile velocity, 46

W

Wizard
 Installation Wizard, 8
 New Project Wizard, 14
 Parameter Export/Import wizard, 70
 Regulation Tuning Wizard, 28
 Startup Wizard, 19

academy.maxonmotor.com

maxon motor
driven by precision