ASiM Basic Functions

TM440

Perfection in Automation

www.br-automation.com

Requirements

Training modules:	TM410 – The Basics of ASiM
Software:	Automation Studio
	Automation Runtime 2.80
	ACP10_MC Library 1.170
Hardware:	None

Table of contents

1. INTRODUCTION

The application program is the central point of development when creating a positioning application.

This is where commands are defined, transferred to the drive and signals from the process are evaluated. The program should ultimately implement an automatic sequence for controlling the drives in the process, in order to achieve the total function.

A solid overview of the available tools is the foundation for setting up a positioning task. Therefore, the first step is to obtain a clear overview of the entire range of function blocks used for controlling ACOPOS servo drives.

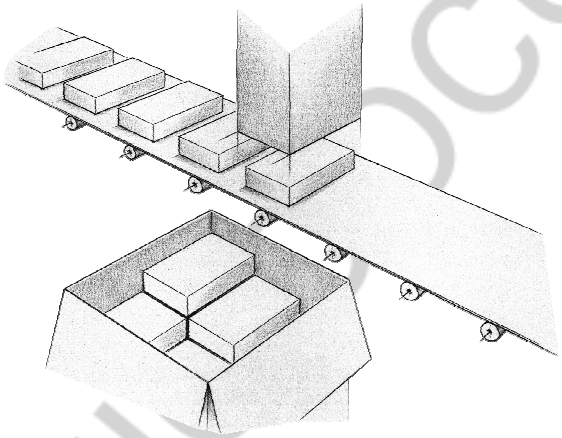


Fig. 1: Palletizing

In this training module we will also take a look at using and integrating positioning functions in an application program.

Working through different exercises will help us to understand the behavior of the function blocks and will provide us with important basic knowledge for applying these functions.

The PLCopen Motion Control Standard plays a central role in the operating concept of B&R's drive solution. We will cover this in a separate section right at the beginning.

Fig. 2: PLCopen Standard

1.1 Training guide objectives

You will become familiar with the operation of the function blocks and will get to know the structure of the corresponding function library (ACP10_MC).

You will master the usage of the basic functions for controlling and operating the ACOPOS.

You will learn how to implement specific positioning sequences in structured form using an application program.

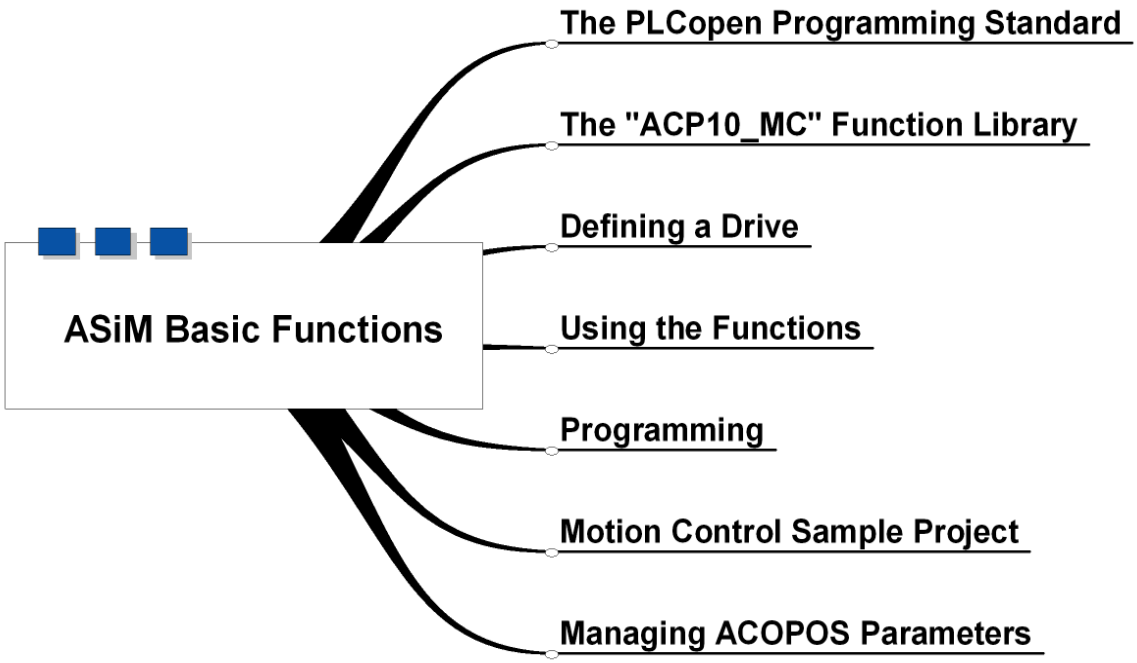


Fig. 3: Overview

2. THE PLCOPEN PROGRAMMING STANDARD

2.1 General

The amount of software available for automation systems continuously increases as more and more machine functions become more reliant on it. This growing complexity results in an increasing number of functions; software development and maintenance become that much harder. In addition, there are a number of solutions (→ products) on the market.

To **simplify** things for the user, it's necessary to have uniform, defined standards.

The PLCopen organization is busy standardizing different areas, components, and tools in the field of industrial automation engineering. In accordance with this, there are different areas of activity:

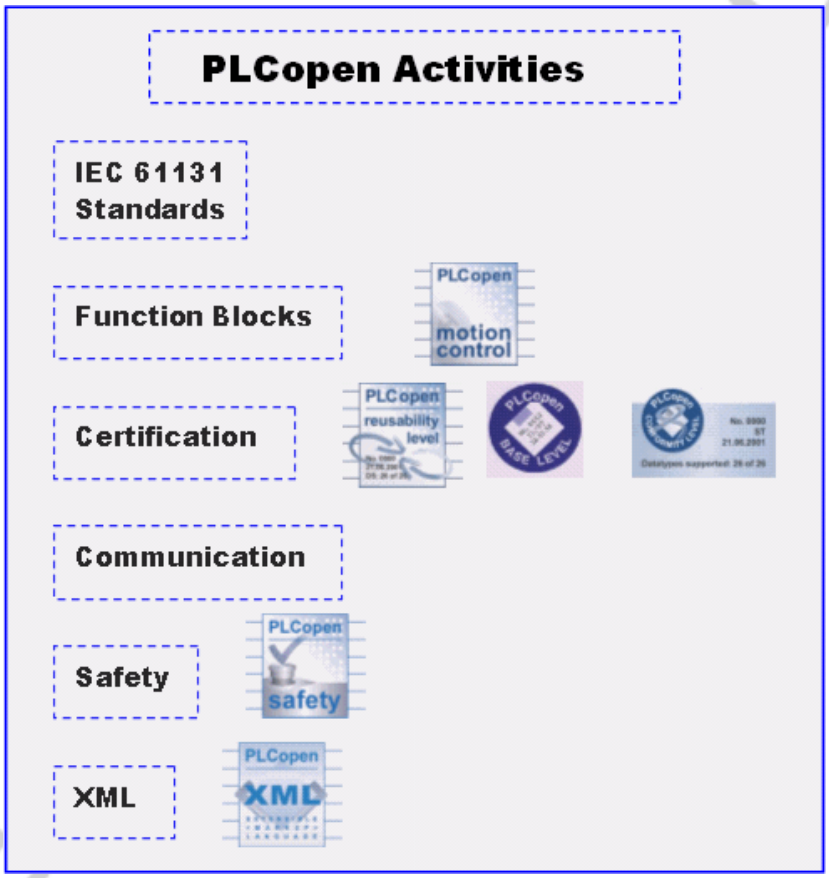


Fig. 4: PLCopen areas of activity

Detailed information about the PLCopen Organization and their activities can be found on the Internet at:

www.plcopen.org

Guidelines are being developed for a series of standard areas and applications to create a **uniform level of operation** for the different system solutions.

Every provider of automation solutions that is a member of this organization provides a uniform operating variant for his system that has been defined by PLCopen. As users, we can be sure that we can handle and work with PLCopen-supported systems as long as we are familiar with the PLCopen standard. **B&R is an active member of the PLCopen organization.**

2.2 Advantages

Manufacturer-independent software development

Because "Where you see PLCopen, you get PLCopen", and each provider in the PLCopen organization supports this standardized "pool of functions".

Shorter development times

Getting to know the specific system solution of a particular provider is no longer necessary, which essentially allows you to start implementing basic functionalities right away. Furthermore, the PLCopen standard is regarded for its simple application.

Simple program maintainability

As we can deduce from the above points, PLCopen provides a tool that offers a clearly-arranged structure in the application program thanks to its simple function blocks. This greatly simplifies the process of making corrections and updates as well as finding errors.

Full support from B&R

The PLCopen standard for positioning applications is available for the B&R drive solution. This means that project setup and configuration can be carried out quickly and easily thanks to the standardized function blocks. In addition to the standardization, special **B&R-specific functions** are also provided, which were prepared in accordance with the standard. This makes it possible to fully utilize the entire functional range of the ACOPOS. The function blocks are provided in a function library.

3. THE "ACP10_MC" FUNCTION LIBRARY

The ACP10_MC function library encompasses the function blocks for controlling the ACOPOS. In addition to the standardized function blocks, this library offers B&R-specific expansions for the special ACOPOS functions.

3.1 Structure and composition

As mentioned earlier, the PLCopen Motion Control Standard makes it much easier to operate drive systems from various manufacturers. Basic functions (e.g. "Positioning to an absolute target position" or "Homing procedure", etc.) are defined, which can be used with any of these systems.

Fig. 5: PLCopen motion control logo

This provides the user with a completely uniform user interface for a specific area of standard applications.

The actual range of ACOPOS functions goes beyond the functionalities contained in the standard. For example, various powerful tools for connecting drives are provided on the ACOPOS (see **TM441 Motion Control: Multi-Axis Functions**). Basic positioning functions are also available with advanced options.

The ACP10_MC library was expanded with a few ACOPOS-specific functions to enable the user to use these functionalities with just one comprehensive access. These advanced functions are operated exactly the same way as the standard functions.

3.2 Function groups

The difference between the PLCopen Motion Control Standard and the B&R-specific function blocks is indicated in the name of the respective function block:

The **standard function blocks** are always named using "**MC_**" at the beginning of the name, such as **MC_MoveAdditive** or **MC_ReadAxisError**. "**MC_BR_**" is used for **B&R-specific function blocks (expansions)** such as **MC_BR_BrakeOperation**, **MC_BR_AutControl**, etc.

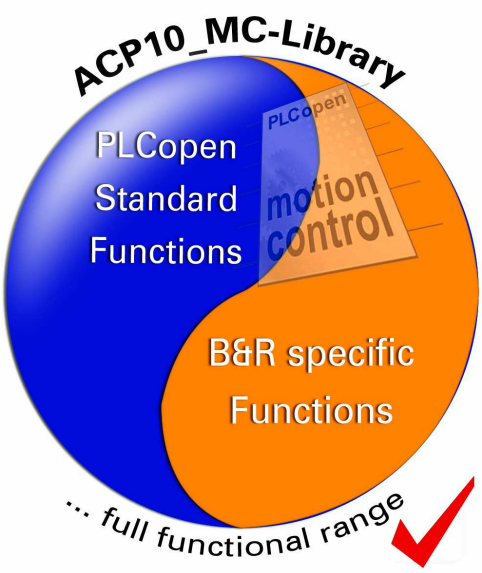


Fig. 6: ACP10_MC library

Classification of the function blocks

The ACP10_MC library has a relatively large number of function blocks. These function blocks can be divided into different groups according to their area of use:

Basic functions for:

- Preparing the drive
- **Basic movements,** such as a movement with absolute target position or relative traverse distance, etc. ...
- **Determining the drive status,** for reading position values, speed values, etc. ...
- Determining and acknowledging drive errors
- Query and control functions for **digital input and output signals**
- Position measurement

- Managing the PLCopen axis parameters
- Managing ACOPOS ParIDs

Multi-axis functions for:

- Creating an electronic gear
- Connecting drives using **cam profiles**
- Configuring and controlling the **cam profile automat**

(see **TM441 Motion Control Multi-Axis Functions**)

Note:

A complete list of the function blocks can be found in the appendix of the corresponding training documentation ("Basic Functions", "Multi-Axis Functions"). The Automation Studio online help contains extensive information about the usage and functionality of each function block.

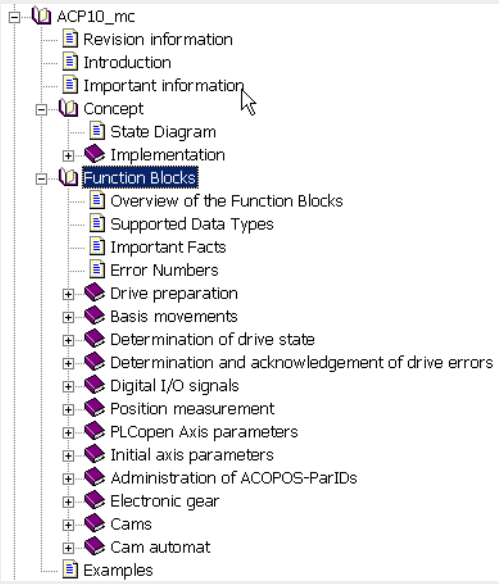


Fig. 7: List of function blocks in the Automation Studio online help

An ACOPOS servo drive must be defined before the function blocks in the ACP10_MC library can be used on the drive. This can be done using a simple routine.

4. DEFINITION OF A DRIVE

The process for creating a drive configuration in Automation Studio was described in the preceding training module. Specific commands could already be transferred to a drive using the diagnostics tool, "NC-Test".

In the following, we will take a look at how to start actions from an application task and implement commands such as movements using a certain program sequence.

To do this, we first need an **axis reference that we can use to address the axis object**. In other words, we need to use a function to let the system know that we want to use a certain axis and need it to return an axis reference which we can use.

4.1 "ncaccess" function

The **"ncaccess" function** is available for us to define the axis. This function registers the NC object (axis) with the NC manager. After the function executes successfully, the respective axis object is put into cyclic operation (i.e. it is assimilated into the manager's execution sequence and commands are passed on).

> **Note:**
>
> The **"flexible NC configuration"** in Automation Studio 2.x is required for using the **"ncaccess()"** functions to request the axis reference for the axis object (NC object) (see **TM410 Motion Components** for information about managing the NC objects in the NC deployment table).

In addition, we also receive an axis reference that can be used to access the axis object from now on.

status = ncaccess(nc_sw_id,nc_obj_name,adr(nc_object))		
Input Parameters:		
nc_sw_id	UINT	NC Software ID: **ncACP10MAN**
nc_obj_name	UDINT	Name (character address respectively string) of NC Object (corresponds to "NC Object Name" in an NC Deployment Table)
Output Parameters:		
nc_object	UDINT	NC Object (Pointer to NC Structure)
status	UINT	Function status: **ncOK** or error code

Fig. 8: Automation Studio online help – "ncaccess" parameter

In the image above (Fig. 6) we see the correct function call as well as a list of function parameters as they are shown in the Automation Studio online help documentation.

Which parameters does this function need?

→nc_sw_id
This parameter specifies the product family. In our case, the NC constant "ncACP10MAN" must be used for the ACOPOS series.

→ access_name !!!
The name that has been defined for the axis object in the axis mapping under **NC Object Name** should be used here.

Automation Studio 2.x

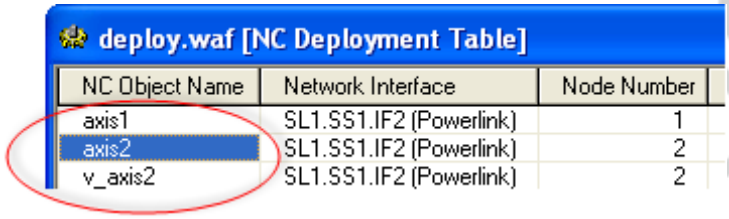


Fig. 9: Deployment table in Automation Studio 2.X

Automation Studio 3

In AS 3, axes are mapped in the NC Mapping Table (similar to the Deployment Table in AS 2). This can be opened by double-clicking on an ACOPOS in the hardware tree.

Fig. 10: NC Mapping Table in Automation Studio 3

←nc_object
The **reference for the axis object** (the memory address of the NC object's NC structure and therefore the reference for the axis object). Specifying this value allows us to apply functions to the desired axis.

← status
returns the function status and should therefore always be checked every time this function is called. A return value of "ncOK" (this corresponds to the constant value 0) indicates that the NC object (axis) was able to be addressed successfully.

Information about errors that may occur when this function is called can be

found by looking up the status error number in the Automation Studio help system.

Notes:

Error number 10600 is a special case. Although the axis was able to be registered correctly and the reference address is valid, there is still another error on the ACOPOS servo drive.

For performance reasons, the **ncaccess** function should be executed in the Init-Sp of the corresponding task. This guarantees that the function is called cleanly once.

Task: "Using the ncaccess function"

You can use this exercise to quickly learn how to use the **ncaccess** function.

Try executing "ncaccess" in your controller task's Init-subprogram and check the status value.

Example for the function call:

```
(* INIT Subprogram *)

access_status:= ncaccess(ncACP10MAN,ADR('Axis1'),ADR(Axis1Obj));
```

The received ID ("Axis1Obj") can then be used for the ACP10_MC functions after the axis has been successfully addressed ("ncOK").

5. USING THE FUNCTIONS

This section offers some basic information about using the ACP10_MC function blocks. We will look at how to operate the positioning function blocks and which possibilities are available for monitoring the procedure (action). The corresponding function library must first be added to the project.

5.1 ACP10_MC library

In order to use positioning function blocks in Automation Studio, the **ACP10_MC library** must first be integrated into the project.

The library is automatically integrated in Automation Studio as soon as an ACOPOS has been added to the hardware tree. The latest version of the **ACP10_MC library** is always integrated.

This can be changed as follows in the event that an older version should be used:

Automation Studio 2.x

To do this, you can select the desired version of the **Motion Control library (ACP10_MC)** by using the **NC properties** pull-down menu in Automation Studio.

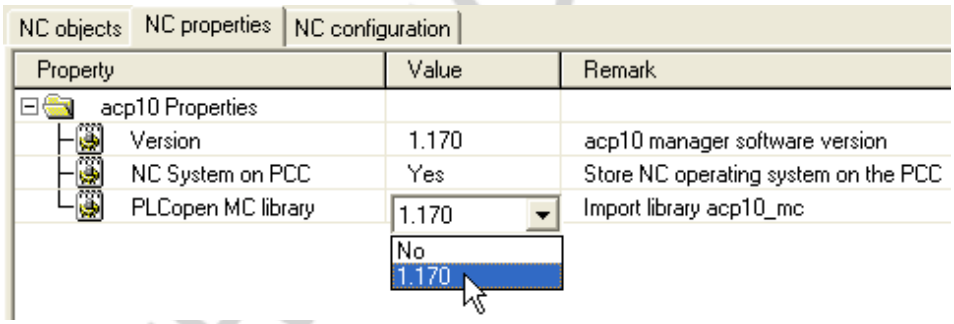


Fig. 11: Implementing the ACP10_MC library

The corresponding library version is imported into the project after it's selected. Once this is done, the motion control function blocks are available for use in the application program.

Automation Studio 3

Versions can be changed in Automation Studio by integrating an older **ACP10_MC library – Version**.

A library can be added via the shortcut menu by selecting **Append Object**.

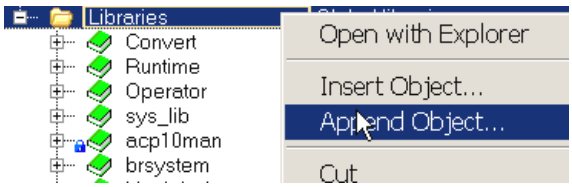


Fig. 12:Appending a new object

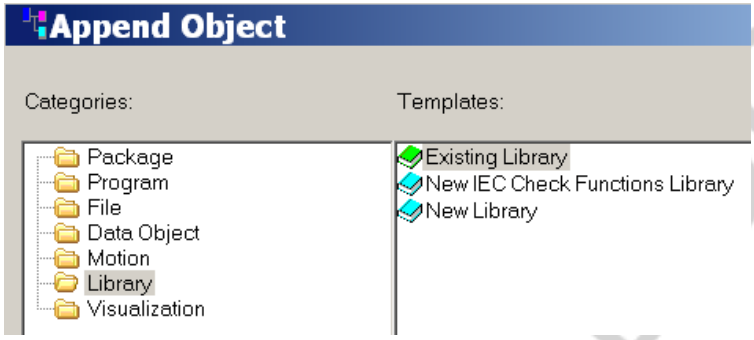


Fig. 13: Appending an already existing library

After selecting the **ACP10_MC library**, the desired version can be defined via the pull-down menu. The original version is overwritten by the new selected version after inserting the library.

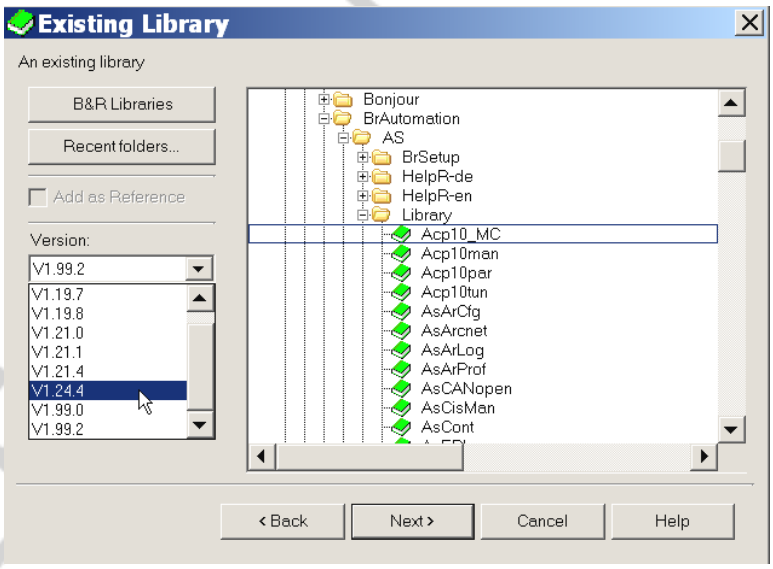


Fig. 14: Inserting another ACP10_MC library version

5.2 Important settings when using ETHERNET POWERLINK

When using **ETHERNET POWERLINK** as communication network, the following settings must be checked in the project:

System Timer / System Tick:

The System Tick determines the shortest time frame in which the task classes can be processed. It is the time basis for processing the task class system.
The System Tick can be generated by all devices that have a **timer device**. Such devices include Powerlink interface cards, X2X interface cards or even the CPU itself.

The Powerlink interface card must specify the timer in order for the application to run **synchronous** to the drives.

A settings dialog box appears in the shortcut menu on the CPU under the **Properties** selection.
The **System Timer** can be set under the **Timing** tab. The **Powerlink interface card** is selected as system timer:

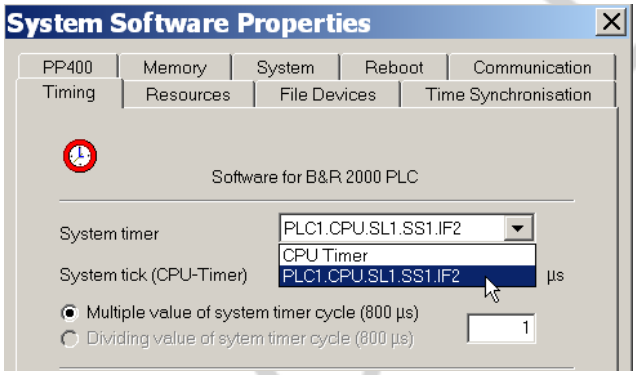


Fig. 15: Selecting the System Timer

After the Powerlink interface card has been selected as **System Timer**, the **Powerlink cycle time** is automatically displayed in the **CPU Timer** field.

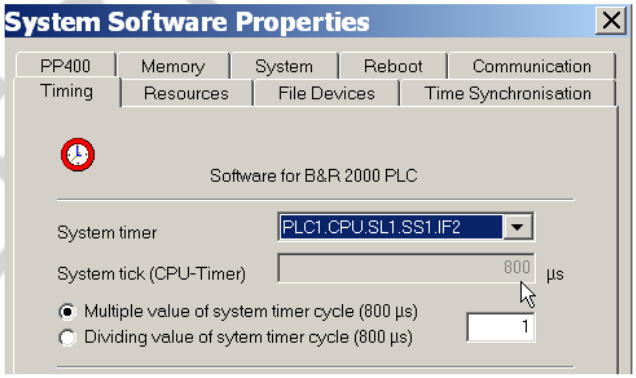


Fig. 16: System Tick is equal to the Powerlink cycle time

The **Powerlink cycle time** can be changed as follows if necessary.

Adjusting the Powerlink cycle time

The Powerlink cycle (cycle time) is set directly on the corresponding interface in the project. 400 µs is the smallest value that can be set. All other values must be a whole-number multiple of 400 µs.

The **Powerlink** folder can be used to set the **Powerlink cycle time** by right-clicking on the Powerlink interface.

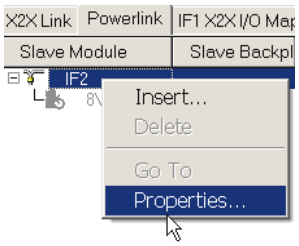


Fig. 17: Powerlink interface properties

Automation Studio 2.x

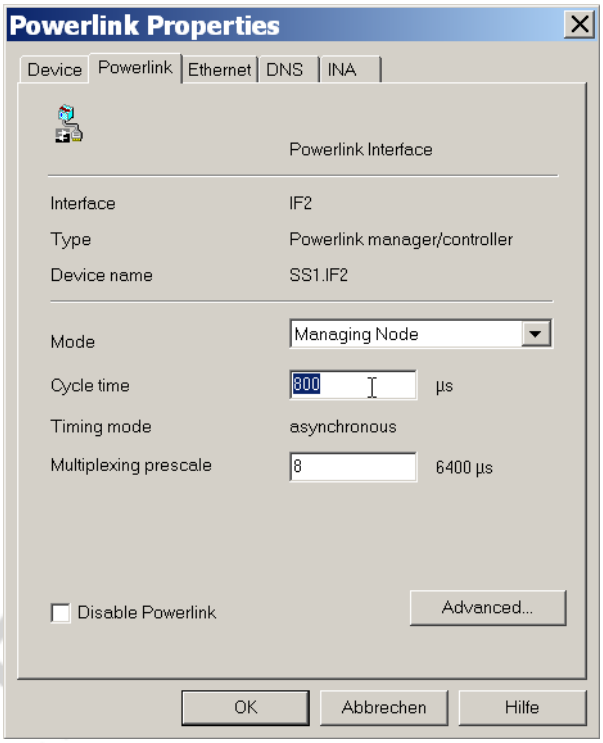


Fig. 18: Setting the Powerlink cycle time in Automation Studio 2.x

Automation Studio 3

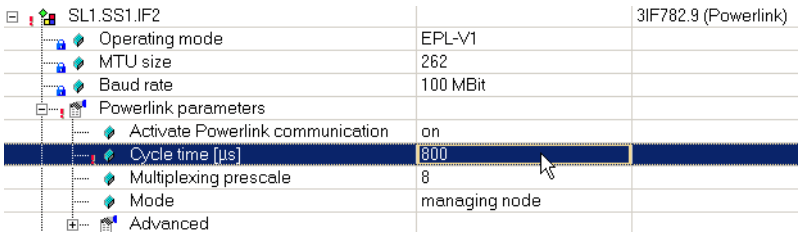


Fig. 19: Setting the Powerlink cycle in Automation Studio 3

The **task class idle time** must be a multiple of the **CPU Timer**:

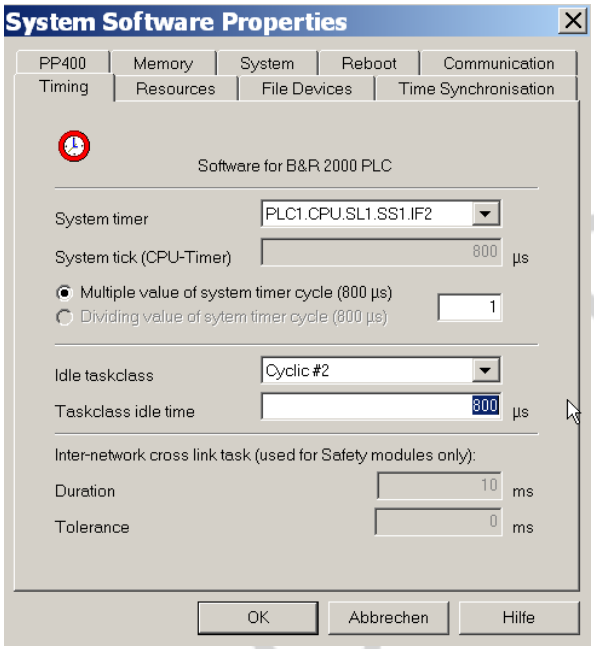


Fig. 20: Task class idle time settings

The **NC manager cycle time** (task class for operation) is defined in the **NC configuration**: This is set to **Cyclic #1** by default.

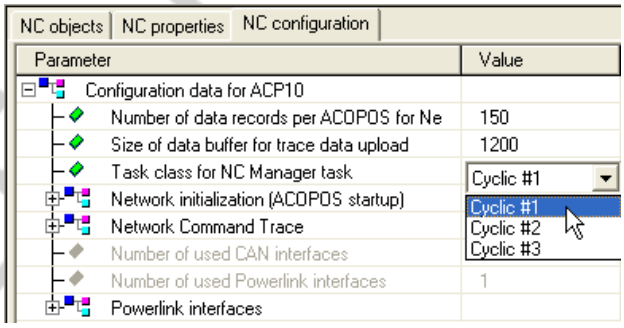


Fig. 21: Setting the NC Manager cycle time

5.3 Controlling the Function Blocks

As discussed earlier, all of the function blocks in the ACP10_MC library (i.e. the standard functions as well as the ACOPOS-specific expansions) are equipped with standard defined operating and status parameters.

This standardized operation of all function blocks makes it easier to use the program and helps to ensure a clear overview during programming.

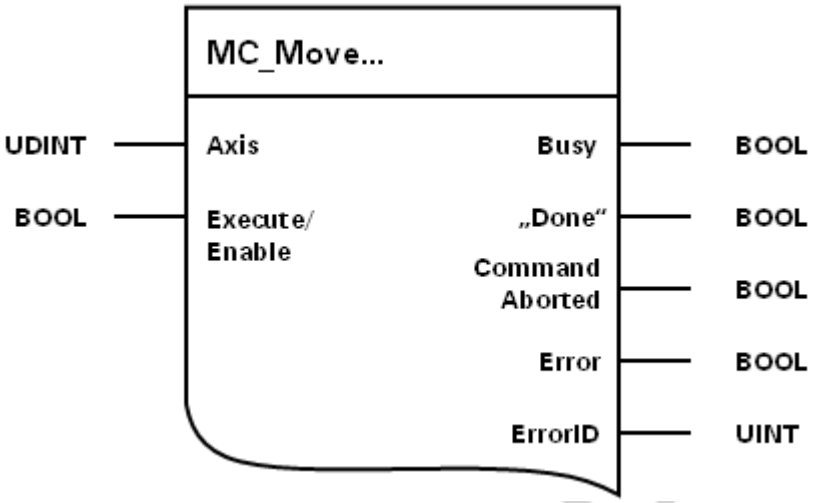


Fig. 22: Standard function parameters

→ ***Axis***
specifies the **axis object** (real or virtual axis), for which the function block should be used - as well as the **axis reference,** which we received from the "ncaccess" function.

→ ***Execute and Enable***
Used to **start the function block**.

> **Note:**
>
> Difference between **Execute** and **Enable:**
> Function blocks with **"Execute" input** read their input values and execute their tasks at a positive edge one time on the Execute input. This makes it possible e.g. to change the speed input of a single axis FB. The new value is read when a new positive edge occurs on the Execute input.
> Function blocks with **"Enable" input** execute their task each time they are called and the Enable input is TRUE. They accept changes to their inputs immediately as long as the Enable input is TRUE.

←Busy
Indicates that the **respective action is currently being executed** (i.e. was started successfully, but is not yet finished).

← Done
Each function block contains a status input that indicates the **successful completion of the action**. This output is labeled differently depending on the function block, but always serves the same purpose.

← CommandAborted
indicates that the **command was aborted** by another function block call.

← Error
indicates that an **error occurred** during the function block call.

← ErrorID
If an error occurs, the corresponding **error number** is returned here. This provides information about the cause of error. A list of error numbers can be found in the Automation Studio online help.

How does this control data behave while the action is being executed?

Successful command processing:

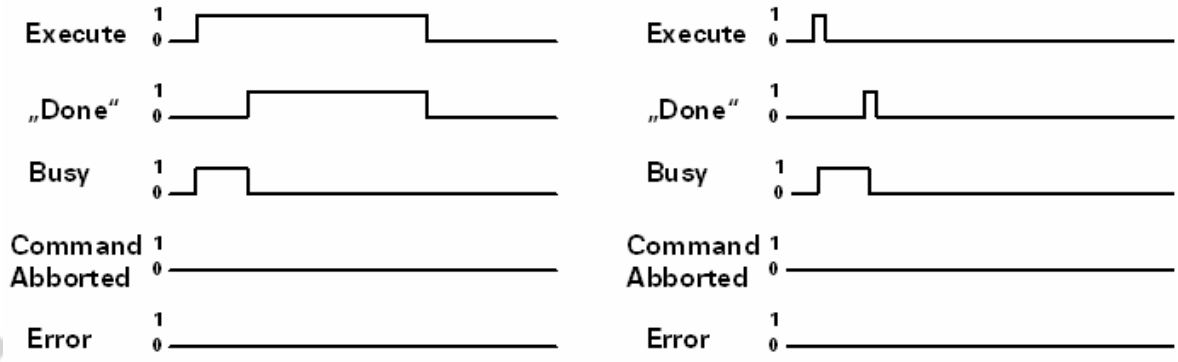


Fig. 23: Command processing successful

As illustrated above, the process is started when a positive edge arrives at the **Execute input**. Active processing of a command is indicated via the **Busy** output. **Done** (or **InVelocity**, **InGear** etc.) indicates that the action has been completed successfully.

Command processing with error:

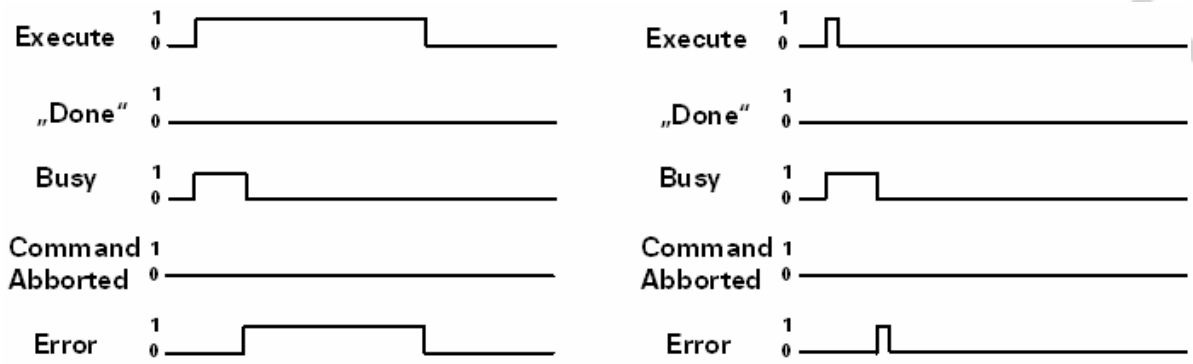


Fig. 24: : Command processing failure

In this case, an error occurs after enabling the action. If the "Error" output is active, then a corresponding error number is displayed via "ErrorID" (PLCopen Motion Control error numbers).

Command processing interrupted:

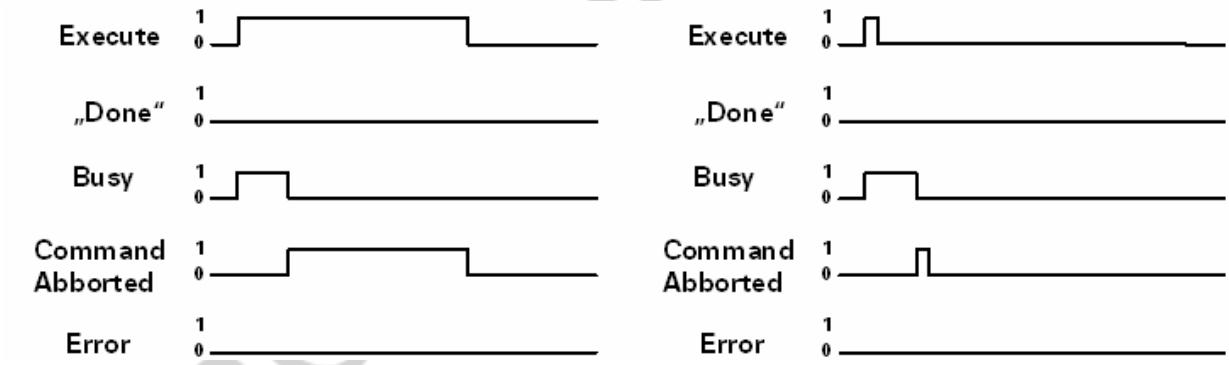


Fig. 25: : Command processing aborted

In this case, the command that is about to be processed is interrupted by another command. This is always the case when the executed function block affects an already active function block **Busy= 1**).

For example, an absolute movement (**MC_MoveAbsolute**) can be interrupted by a relative movement (**MC_MoveAdditive**).

Function blocks for reading status data (position, parameter, etc.) and commands for movements do not affect each other.

Summary:

- The status information **Done**, **Command Aborted, Error** and. **ErrorID** remain set until the **Execute input** is reset.
- The status outputs are set for the **duration of a cycle** if the **Execute input** is already disabled before this signal arrives (as seen in the diagram):

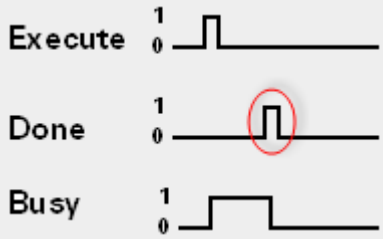


Fig. 26: Signal duration is equal to one cycle

- The **Execute input** must be set to FALSE to acknowledge the **ErrorID** output of a FB. The outputs are then reset the next time a function is called.

- The function blocks with an **Enable input** are only active as long as this input remains set. Otherwise, the action is rejected including the current status values.

Task: "Using the ACP10_MC basic functions"

The function blocks from the ACP10_MC library can be used quickly and easily in an application task. This procedure can also prove quite useful later to try out functions before integrating into a fixed sequence.

This is done as follows:

The desired ACP10_MC library version must first be integrated into the project. (→ see 5.1 ACP10_MC_Library)

A reference for the axis object must be received by using the ncaccess function before the ACP10_MC function blocks can be used in the application program.

The desired function blocks can now be consecutively inserted to the cyclic part of the program.

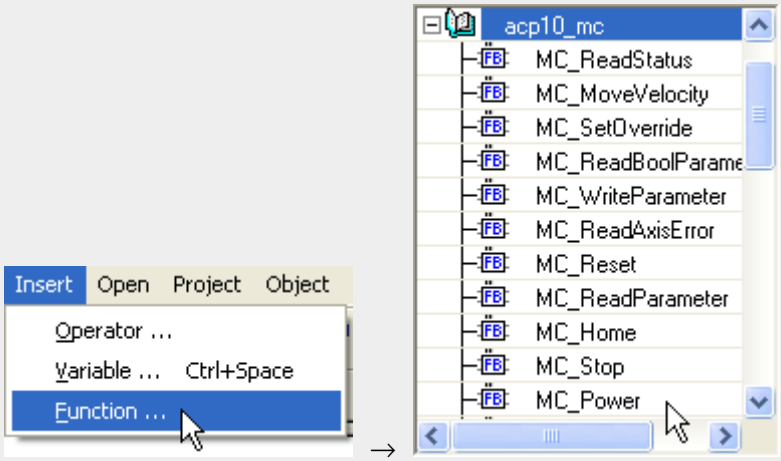


Fig. 28: Inserting new function block

Define the instances required for each function block (data structures for the function blocks):

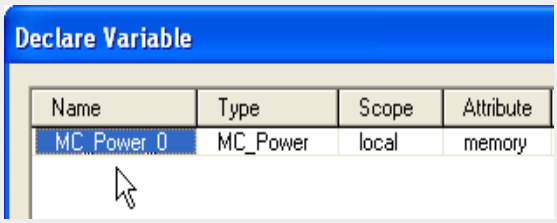


Fig. 28: Declaring a function block

After completing the insert dialog box, the function call is added to the program with the input parameters, as follows:

```
MC_Power_0(Axis:=, Enable:=)
```

To allow for a clear and organized programming environment, the input parameters should be removed from the function block and defined individually right before the function block is called (see below).

The **Axis** parameter must be provided on each function block with the axis object's reference (example above: "Axis1Obj"). All remaining parameters can be operated using the watch window.

This results in the following program structure:

```
(* Cyclic program section *)

(* Function block calls *)

(* ID assignment *)
MC_Power_0.Axis:= Axis1Obj;
(* Function call *)
MC_Power_0();

MC_Home_0.Axis:= Axis1Obj;
MC_Home_0();

MC_MoveAdditive_0.Axis:= Axis1Obj;
MC_MoveAdditive_0();
...
```

Download the project and operate the various function blocks from the **Watch** window. If necessary, you should define input parameters for the corresponding function block and activate the function block using "Execute" or "Enable":

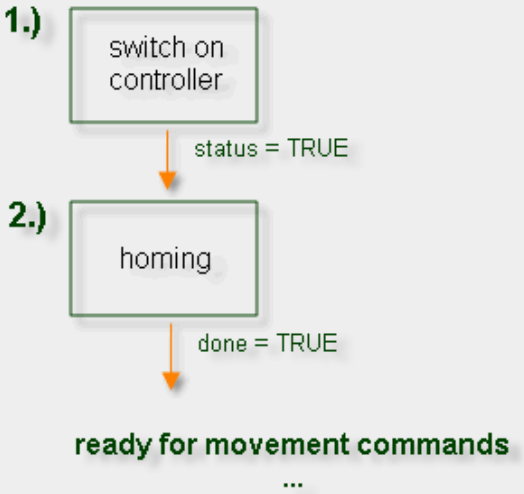
Name	Type	Scope	Force	Value
MC_MoveAdditive_0	MC_MoveAdditive	local		
Axis	UDINT			52729784
Execute	BOOL			FALSE → TRUE
Distance	REAL			1000.0
Velocity	REAL			1000.0
Acceleration	REAL			10000.0
Deceleration	REAL			10000.0
Done	BOOL			FALSE
Busy	BOOL			FALSE
CommandAborted	BOOL			FALSE
Error	BOOL			FALSE
ErrorID	UINT			0

Fig. 29: Controlling the functions via the Watch

As seen in the image, the instances for the function block (= data structure) can be placed and operated in the Watch (**MC_MoveAdditive** example).

Important:

The drive controller must first be activated (**MC_Power**) to prepare the ACOPOS for movement actions. A homing procedure is then necessary (**MC_HOME**):



Functions for preparing and moving the drive:

- **MC_Power** & **MC_Home**
- **MC_MoveAbsolute**, **MC_MoveAdditive**, **MC_Stop**
- **MC_MoveVelocity**, **MC_SetOverride**

Error handling functions:

- **MC_ReadAxisError** (for acknowledging errors on the ACOPOS)
- **MC_ReadStatus** (for outputting the current drive state, see "5.3. Drive States")
- **MC_Reset** (if the drive object is in the "Errorstop" state due to a preceding axis error).

Functions for reading axis data:

- **MC_ReadActualPosition**
- **MC_ReadActualVelocity**

Use the functions described above and pay attention to the effects on the drive and the function block status parameters.

Caution:

The initialization value of the function block variables (variable declaration in the user task when inserting the function block) must be 0. Accordingly, no variables with the "permanent" or "remanent" attribute may be used. This prevents any errors from occurring in the internal sequence after a restart.

5.4 Drive states

Specific defined states are determined for operating a drive. These states provide a simpler overview of complex movement procedures and make it easier to process error situations.

These states are:

- **Disabled**, the drive controller is switched off.
- **Standstill**, the drive is not currently executing a movement and is ready for a positioning command.
- **Homing**, the drive is executing a homing procedure.
- **Errorstop**, the drive is in standstill after an error.
- **Stopping**, the drive is stopping an active movement.
- **Discrete Motion**, the drive is executing a movement with target position. Therefore, the movement has a defined end.
- **Continuous Motion**, the drive is executing a movement without target position. The movement has no defined end.
- **Synchronized Motion**, the drive is coupled to another drive.

Transitions between these states are now executed using specific positioning commands (function blocks).

This can result in the following sequence for example:

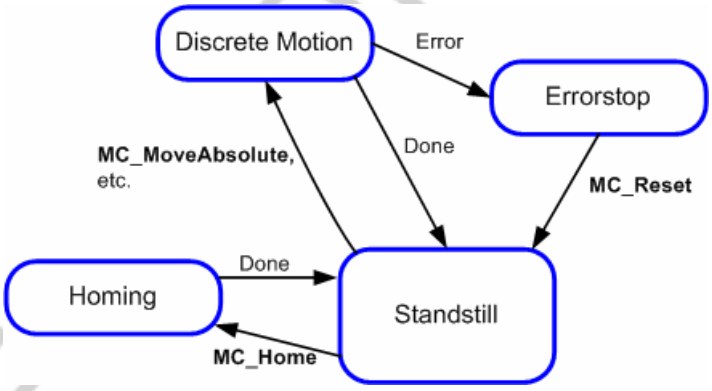


Fig. 30: Progression of states

Let's assume that the axis is in the **Standstill** state. As soon as it has successfully performed a homing procedure, the **MC_MoveAbsolute** command can be used to start a movement.

After the target position has been reached, the drive returns to its "initial state". If a drive error occurs during the positioning action, then the axis

enters error state (**Errorstop**). This can be **acknowledged** using the **MC_Reset** function (after the drive error has been corrected).

The following diagram shows how all of the states are interconnected:

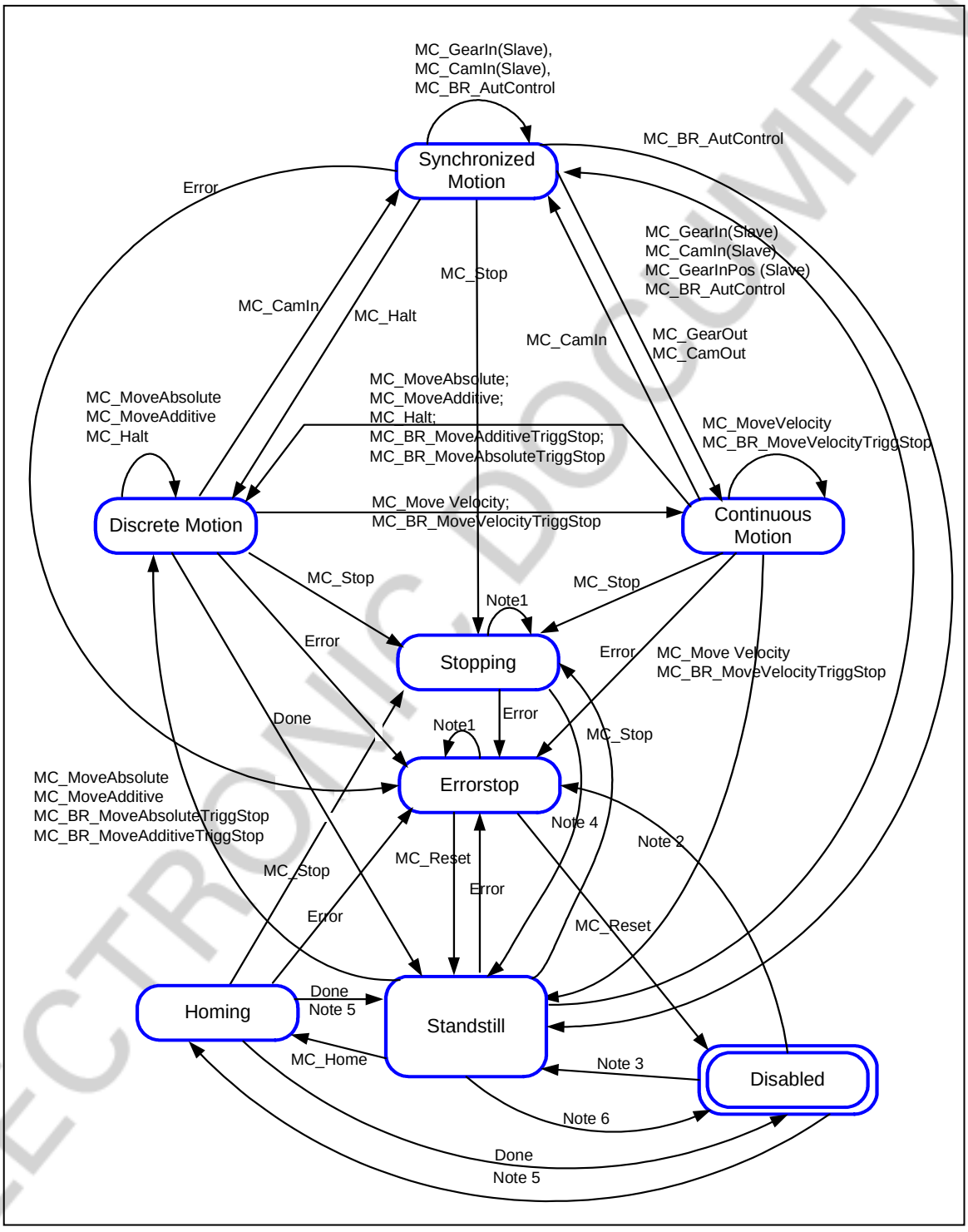


Fig. 31: PLCopen Motion Control diagram of states

Note:

A virtual axis does not have a drive controller. Therefore, unlike the real axis, the virtual axis does not have a **Disabled** status either. The NC object "virtual axis" starts in the **Standstill** state and does not necessarily need the **MC Power** function block for activation.

The most important drive states are included in the diagram. They can be used for coordinating positioning sequences. The **MC_ReadStatus** function block is used to read the current status from an axis.

Task: "Status monitoring"

The current drive state can be determined using the **MC_ReadStatus** function block. Add this function to your test program and monitor the state changes when executing different positioning actions.

5.5 Advanced settings

Additional settings in Automation Studio allow us to make adjustments to position values. Periodic position behavior as well as position scaling can be achieved using these simple configurations.

What do we need to do first?

The basic settings for **scaling** a revolution in units are made in the encoder interface parameters for the axis:

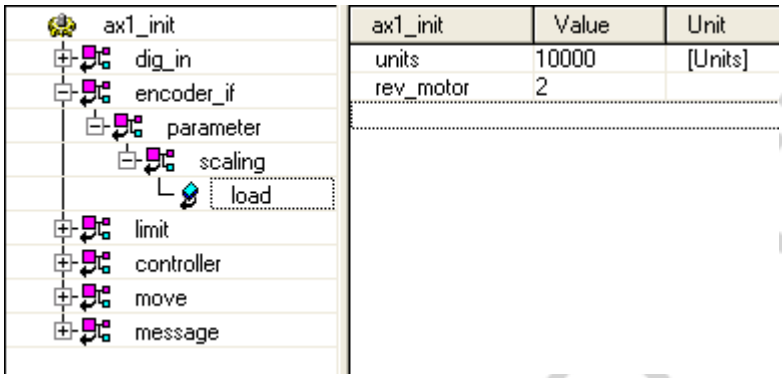


Fig. 32: Init parameter module, encoder interface

In this case (image above), the example shows **10,000 units divided into two axis revolutions**, which results in **5,000 units per revolution**. This allows us (of course keeping the maximum resolution of the encoder in mind) to divide the revolutions according to our needs. The preset data type for the position is **DINT** (double integer), i.e. a whole-number value.

PLCopen function blocks uniformly use the **REAL data type** for position specifications. This makes the following configuration options possible and sensible:

Values for the **position period and factor** can be entered for the **PLCopen_ModPos="<Period>,<Factor>"** entry in the axis mapping table to adjust the position value:

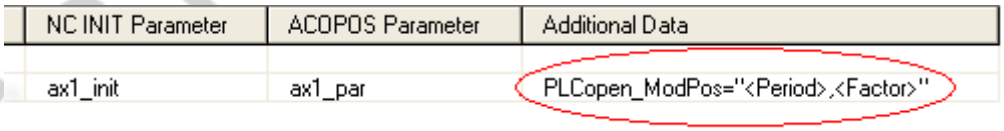


Fig. 33: NC mapping table, Advanced settings

→ **Period**

For continuous axis movements, a **position value determined periodically** is frequently needed. If a value greater than 0 is being used for the period, then the position value is adjusted according to this entry. It refers directly to the scaling of the axis in the encoder parameters (see above). All PLCopen function blocks "work" with this periodic position.

For example, if the value <Period> = 1000 is used, then the position value always increases from 0 towards 999 during positive movement, resets again to 0, increases up to 999, etc.

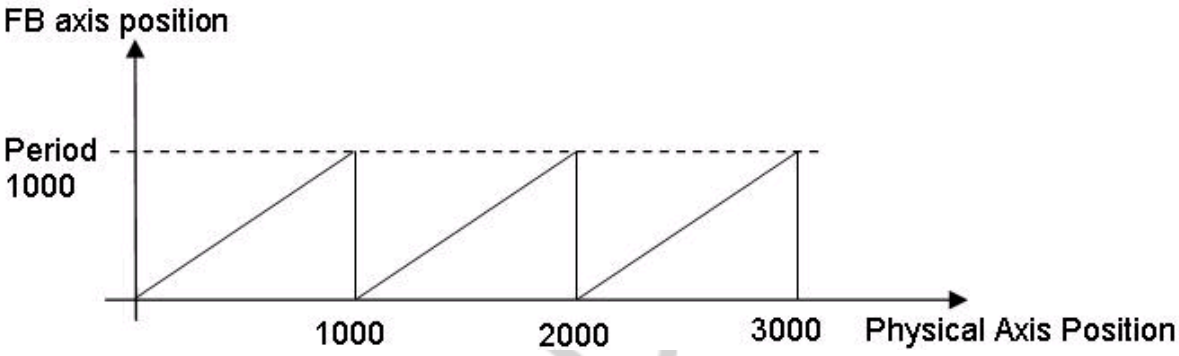


Fig. 34: Periodic axis position

Periodic position conversion can also be explicitly disabled by entering *<Period>*=0 if for example, periodic behavior is not required, (but a specific factor has to be used). This once again limits the position to the value range of the *REAL* data type.

→ **Factor**

PLCopen function blocks use the REAL data type for the axis position. This data type allows us to use decimal places, which in turn makes **simplified scaling** interesting.

What exactly is "scaling"?

"Scaling" is nothing more than "converting". The best way to look at this is to use an example:

A certain application requires positioning to be accurate down to 1 µm. Let's assume that the way we've configured the encoder parameters allowed us to set up a 1 µm distance per positioning unit.

Now it might be beneficial if we could specify the distance in millimeters for our positioning task with PLCopen function blocks. And that's exactly what we achieve with this factor.

The equation looks like this:

$$PLCopenUnits = \frac{AxisParameterUnits}{Factor}$$

So if we set the factor to a value of 1000, we will get our scaling to millimeters. If we now start a movement for a distance of value 1 with the corresponding PLCopen function block, our axis will actually travel 1000 axis parameter units.

<Factor>	PLCopen units [REAL value]	Axis parameter units [DINT value]
1	1	1
1000	1	1000
1000	0.001	1

Note:

The velocity and acceleration values will also refer to this scaling in the future.

Task: "Settings for position adjustment"

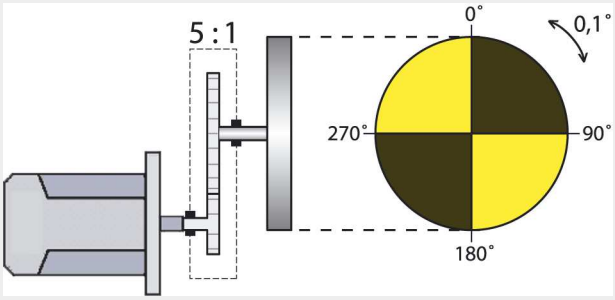
This simple example should implement the settings shown above and demonstrate the possibilities.

Default:

A pivoting carrier must move a product to different stations for processing (specific angular positions within the 360° of a full rotation). Positioning must be within 0.1° of accuracy. The **MC_MoveAbsolute** function block is used for approaching the positions.

To simplify this procedure a little, the position is specified in degrees, with one decimal place, e.g:

```
..
MC_MoveAbsolute_0.Position:= 135.0;     (* Goal: 135° *)
...
```



The carrier is driven by a gear (gear ratio= 5:1) using a servo motor.

Find the appropriate settings for the position value (encoder setting and advanced settings) based on these specifications.

Solution:

The basic settings for the encoder are made in the Init parameter module. Therefore, for the position resolution in **0.1° steps**, **3600 units** are required for one revolution of the rotating carrier. These units are distributed over **5 motor revolutions**.

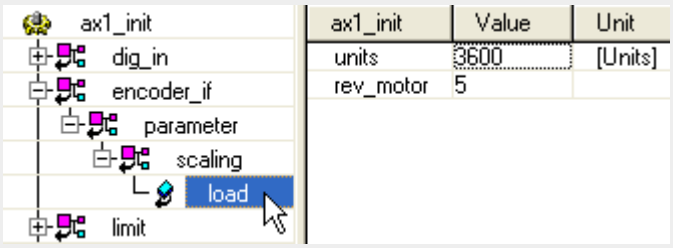
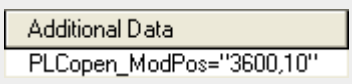


Fig. 35: Init parameter module, encoder interface, setting

If a movement of 3600 units is now executed, then the motor performs exactly 5 revolutions and the carrier is rotated exactly 360°. The gear has now been fully set.

To now get the desired scaling for the positioning function, the value 10 must be defined for the factor and the value 3600 for the period (based directly on the encoder setting):



6. PROGRAMMING

The application program should establish an automatic sequence for controlling the ACOPOS. When doing this, it is important to implement the function blocks into the program in a clear and organized manner.

Furthermore, error events must be taken into consideration and accordingly handled if necessary.

6.1 Creating an automatic sequence

How can we implement function blocks of this type into a **program structure** optimally and as clearly as possible?

Each function block **allows for targeted execution**. In other words, we can start our commands at a certain position and at a desired time in our program code.

Afterwards, the function block provides **feedback about the status** via the corresponding output parameters:

- Was the function block able to be executed successfully?
- If not, which errors (**ErrorID**) occurred?
- How does the "physical" process look? Is the axis already in the target position, running at the intended speed, able to be referenced successfully, etc.?

We can use this information as a **basis for making decisions regarding the following steps in the program sequence**. If an error would occur in a program, we would have to respond differently depending on what that error is.

A control structure that is especially well-suited for managing these types of function sequences is the **step sequencer**.

This type of structure allows the implementation of individual steps whose sequence can be determined by the use of a step index.

The following simplified diagram shows an example of what this type of step-by-step function execution could look like.

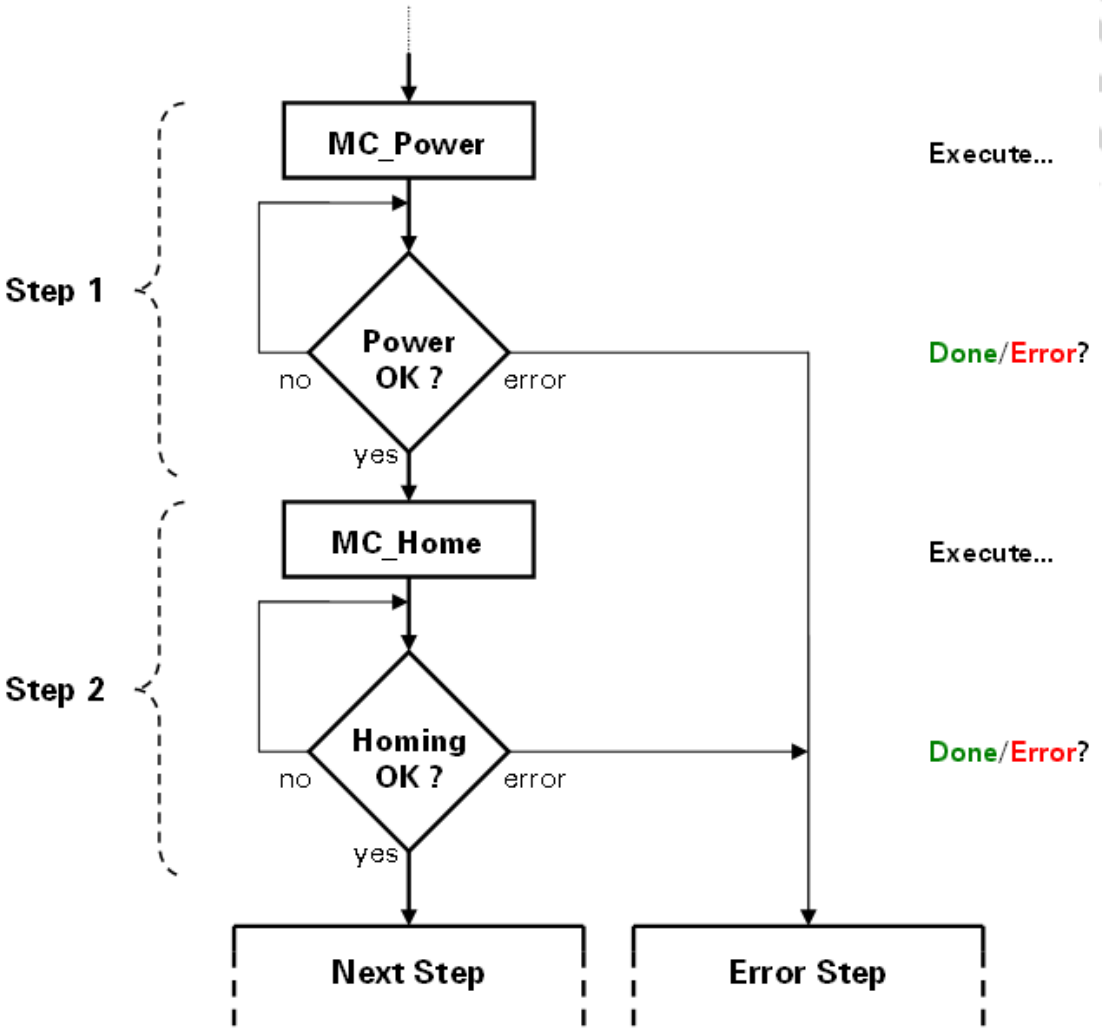


Fig. 36: Sample sequence, structured programming

We can now activate our function blocks (**Execute**) in the individual sequence steps and use the parameter values (**Done**, **CommandAborted**, **Error**, **ErrorID** etc.) to determine what the next step would be.

At the same time, we are providing our application with a clearly arranged structure so that any future updates or modifications can be implemented relatively easily.

Task: "Structured programming"

This task should clarify how the function blocks in the ACP10_MC library can be used in an application program to achieve an **automatic sequence**.

The function blocks, which in the previous example were solely executed using the watch window, should now be controlled using a step sequencer.

Default:

Create the following sequence:

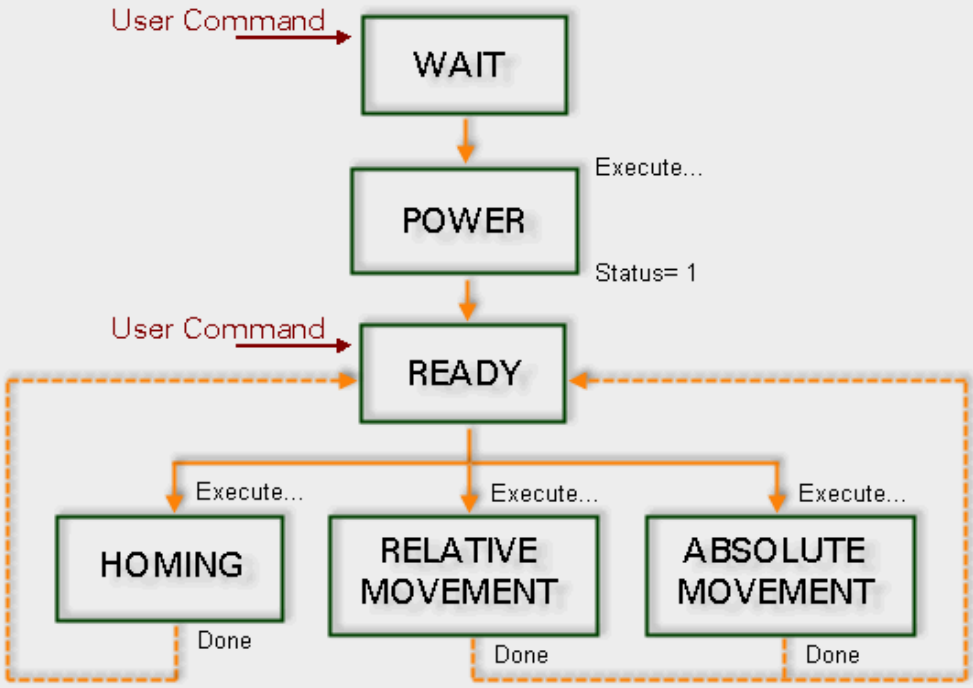


Fig. 37: Task, sequence diagram

- After starting up the system, the system should wait for the signal to activate the drive controller in a waiting step (**WAIT**). If a corresponding signal arrives (from the user), then the controller is activated with the **MC_Power** function. The program should then go into another waiting step (**READY**).
- It should be possible to start other actions, a homing procedure or a relative or absolute movement from the waiting step. After the action is complete, the program should return to the waiting step.

The routines must be controlled by the user (watch window). To do this, the following structure should be created in the project:

Control structure

AxisBasic_typ:

```
Power                 | BOOL     | Controller on / off
Home                  | BOOL     | Start search home
MoveAbsolute          | BOOL     | Start absolute movement
MoveAdditive          | BOOL     | Start relative movement
ParaPosition          | REAL     | Target position absolute Motion
ParaDistance          | REAL     | Traverse distance rel. movement
ParaVelocity          | REAL     | Movement velocity
ParaAcceleration      | REAL     | Movement acceleration
ParaDeceleration      | REAL     | Movement deceleration
```

Create this structure in the project and a global variable ("gAxisBasic" type *AxisBasic_typ*). The individual elements can be used to control the routines.

Implementing the first routine ("Power"):

The **MC_Power** function block is required for this routine. The function block call can be placed in the cyclic section of the control task ("Function Block Calls"). Fixed input parameters (e.g. axis reference) can also be set here.

The function block controller inputs are linked in the step sequencer:

```
(* Cyclic program *)
...
(* AxisStep = Step index *)
CASE AxisStep OF

      (* Wait for "Controller on" command *)
      STATE_WAIT:
            MC_Power_0.Enable:= 0;

            (* "Controller on" command *)
            IF (gAxis.Power = 1) THEN
                  AxisStep:= STATE_POWER_ON;
            ...

      STATE_POWER_ON:
            (* Start action *)
            MC_Power_0.Enable:= 1;

            (* Controller action enabled? Action complete? *)
            IF (MC_Power_0.Status = 1) THEN
                  AxisStep:= STATE_READY;
            END_IF
            ...
END_CASE
```

```
(* Function block calls *)
      MC_Power_0.Axis:= Axis1Obj; (* Axis reference assignment *)
      MC_Power_0();               (* Cyclic call *)
```

The sequencer in the step sequencer decides if and when a function block is enabled.

The following querry can be set in the cyclic section of the program to switch off the controller (possibly before the step sequencer):

```
IF (gAxis.Power = 0) THEN
    AxisStep:= STATE_WAIT;
END_IF
```

As a result, resetting the "**gAxis.Power**" structure element will cause the step **STATE_WAIT** to be executed cyclically. The controller is then deactivated (**Enable= 0**).

Implementing the other routines:

The following actions can now be executed the same way starting from the **STATE_READY** step. You must also consider which parameter values must be taken from the operating structure. This would appear as follows for the action, "absolute movement":

```
STATE_READY:
     (* Command? *)
     IF (gAxis.MoveAbsolute = 1) THEN
          gAxis.MoveAbsolute:= 0;
          AxisStep:= STATE_MOVE_ABSOLUTE;
     END_IF
     ...
STATE_MOVE_ABSOLUTE:
     (* Transfer of the relevant parameters *)
     MC_MoveAbsolute_0.Position:= gAxis.ParaPosition;
     MC_MoveAbsolute_0.Velocity:= gAxis.ParaVelocity:
     ...
     (* Start action *)
     MC_MoveAbsolute_0.Execute:= 1;

     (* Target position reached *)
     IF (MC_MoveAbsolute_0.Done = 1) THEN
          MC_MoveAbsolute_0.Execute:= 0;
          AxisStep:= STATE_READY;
     END_IF
```

As you can see, we must not forget to transfer the corresponding parameter values to the function block before activation.

Function calls and fix configurations can be added "below" in the program ("Function Block Calls").

Perform the necessary programming and test the routines using the control structure.

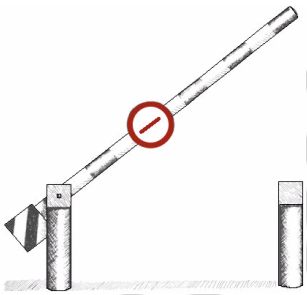
6.2 Error handling

We have already partially implemented automatic routines in the previous example, but we have not taken potential error events into consideration.

Which errors have to be taken into consideration in the application program?

We can differentiate between two different "error cases":

- Errors that occur when a function block is called (e.g. because of an incorrect configuration)
- Drive errors



How can these errors be monitored?

PLCopen function blocks provide direct feedback about their status. An error that occurs during execution is displayed on the **Error** status output. In this case, a corresponding error number is output on the function block's **ErrorID** output, which can be used to more precisely localize the error (motion control, error numbers).

Caution:

If a function block is called with an error, it's absolutely necessary to reset it before the next application so that the error status can be left. This is done by executing the function block with **Enable=0** or **Execute=0**.

The **MC_ReadAxisError** function block is available to monitor errors on the ACOPOS device.

These options allow the following approach to be used as an error handling routine:

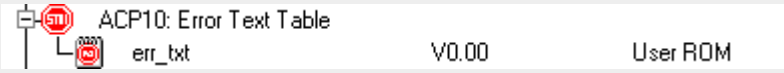
We recommend using the **ReadAxisError** function block cyclically in the program to monitor ACOPOS errors. The function block must be constantly active for this (**Enable = 1**). The necessary query (**error?**) can be placed in a cyclic part of the program (possibly before the step sequencer).

If a drive error is registered, then the regular program execution should be stopped and a step for handling the error should be entered.

Note:

Additionally, the **MC_ReadAxisError** function block also offers the possibility to output **plain text about the current ACOPOS error** (a brief description of the error) using a STRING variable (type *STRING(79)[4]*).

In order to use this option, the NC software object **error text module** must be added to the project in the desired language via the known method.



The name of this module must be connected to the function block. Once this is done, the text for the current error number is then sent from the function block to the specified STRING variable.

Detailed information about the configuration can be found in the Automation Studio online help.

The same is true when an error occurs during the actual function block call. There should also be an "error step" implemented to handle this situation.

The following flow chart illustrates this structure once again:

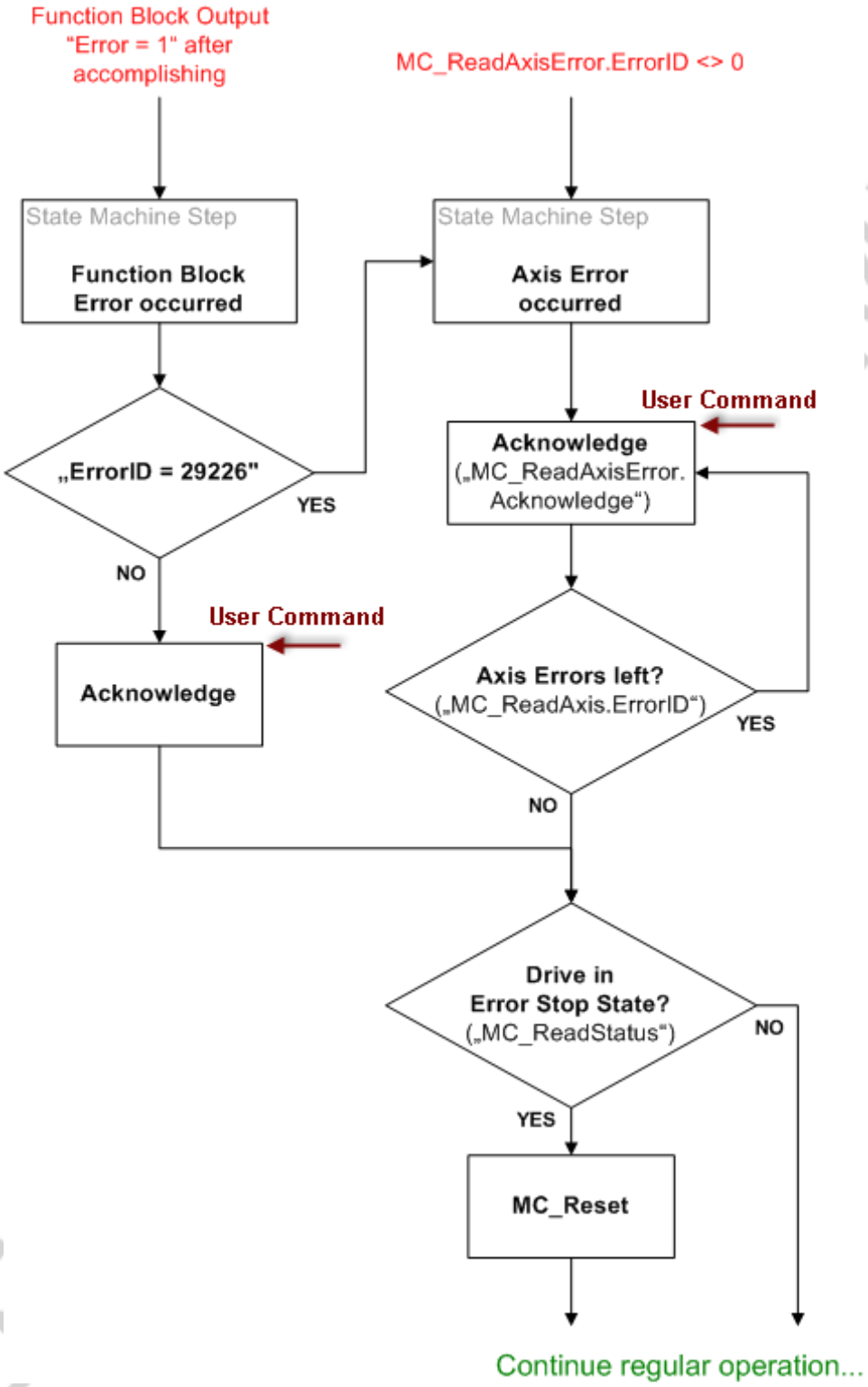


Fig. 38: Sequence for error handling in a positioning task

Starting from the top, we deal with both types of error events (drive error or function block error):

- If the problem is a drive error, we can begin carrying out the **step-by-step analysis** (ACOPOS error numbers) and acknowledge the pending error. The function block returns the respective ACOPOS error number until the error that occurred last is acknowledged.

- If a function block error message is output, an immediate check should take place to see whether the behavior was caused by an ACOPOS error (**ErrorID = 29226** on the function block). If this is the case, you can switch directly to the routine to **analyze and acknowledge the drive errors**. If not, this type of error can also be acknowledged and the error step can be exited after the error event (PLCopen motion control, error numbers) is analyzed.

- After both routines have been run through, the drive status should be checked as well. In certain cases, the ACOPOS device can be set to an **Error stop** status. This status remains in effect until a status **reset** (**MC_Reset**) is carried out (see the motion control status diagram).

The application can then be continued once all errors have been corrected.

Note:

This program sequence should in no way be seen as a guideline for implementing error handling. Instead, it should merely serve as a universal approach or basis for implementing this type of routine.

Task: "Implementing the routines for error handling"

Implement error monitoring and error evaluation in your application program based on the diagram shown above.

Additional specifications:

It should be possible to acknowledge errors using a common variable. In the event of an error, the error number should be transferred to a separate element of the control structure (see previous example) and the error text should be displayed.

The control structure (*AxisBasic_typ*) for the task must be expanded to include the following elements:

```
ErrorAcknowledge       | BOOL                | Acknowledge error
ErrorID                | UINT                | Error number
ErrorText              | STRING (79) [4]     | Error text
```

Tips for implementation

To start with, the function block **MC_ReadAxisError** is required for cyclic monitoring of ACOPOS errors. This can be placed in the cyclic section of the program ("Function Block Calls"), → **Enable" set permanently to 1** ("TRUE"). The error text string can also be connected here (see Automation Studio online help).

The return status (drive status) must also be evaluated cyclically. E.g.:

```
(* Cyclic program section *)

(* Checking for drive errors *)
IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
    (* Error handling *)
    AxisStep:= STATE_ERROR_AXIS;
END_IF
```

Additional queries for function errors must be implemented to evaluate the function states within the step sequencer.

For example:

```
(* Cyclic program section, step sequencer *)

(* Absolute movement *)
STATE_MOVE_ABSOLUTE:
    (* Transfer of the relevant parameters *)
    MC_MoveAbsolute_0.Position:= gAxis.ParaPosition;
    ...
    (* Start action *)
    MC_MoveAbsolute_0.Execute:= 1;

    (* Target position reached *)
    IF (MC_MoveAbsolute_0.Done = 1) THEN
        MC_MoveAbsolute_0.Execute:= 0;
        AxisStep:= STATE_READY;
    END_IF

    (* Function block error *)
    IF (MC_MoveAbsolute_0.ErrorID <> 0) THEN
        gAxis.ErrorID:= MC_MoveAbsolute_0.ErrorID;
        (* Error handling *)
        AxisStep:= STATE_ERROR;
    END_IF
```

The routines for acknowledging the error must still be created (function errors and drive errors, see diagram above). After the acknowledge has been completed, the task returns to its initial state (**STATE_WAIT**).

For example:

```
(* Cyclic program section, step sequencer *)

(* Function error handling *)
STATE_ERROR:
    (* Checking for drive errors *)
    IF (gAxis.ErrorID = 29226) THEN
        AxisStep:= STATE_ERROR_AXIS;
    ELSE
        (* Acknowledge by user *)
        IF (gAxis.ErrorAcknowledge = 1) THEN
            gAxis.ErrorAcknowledge:= 0;
            gAxis.ErrorID:= 0;
            (* Check status *)
            IF (MC_ReadStatus_0.Errorstop = 1) THEN
                AxisStep:= STATE_ERROR_RESET;
            ELSE
                AXIS:= STATE_WAIT;
            END_IF
        END_IF
    END_IF
    ...


(* Drive error handling *)
STATE_ERROR_AXIS:

    gAxis.ErrorID:=  MC_ReadAxisError_0.AxisErrorID;
    MC_ReadAxisError_0.Acknowledge:= 0;

    (* Acknowledge by user *)
    IF (gAxis.ErrorAcknowledge = 1) THEN
        gAxis.ErrorAcknowledge:= 0;
        (* Execute acknowledge *)
        IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
            MC_ReadAxisError_0.Acknowledge:= 1;
        END_IF
    END_IF

    (* No more errors on the drive *)
    IF (MC_ReadAxisError_0.AxisErrorCount = 0) THEN
        gAxis.ErrorID:= 0;
        (* Check status *)
        IF (MC_ReadStatus_0.Errorstop = 1) THEN
            AxisStep:= STATE_ERROR_RESET;
        ELSE
            AXIS:= STATE_WAIT;
        END_IF
    END_IF
    ...
```

```
(* Reset status *)
STATE_ERROR_RESET:
     (* Start action *)
     MC_Reset_0.Execute:= 1;
     (* Exit error status *)
     IF (MC_Reset_0.Done = 1) THEN
          MC_Reset_0.Execute:= 0;
          AxisStep:= STATE_WAIT;
     END_IF
     ...
```

The corresponding function block calls (& fixed configurations) must be available in the cyclic program section ("Function Block Calls"):

- **MC_ReadAxisError** with configurations for displaying error text
- **MC_ReadStatus**
- **MC_Reset**

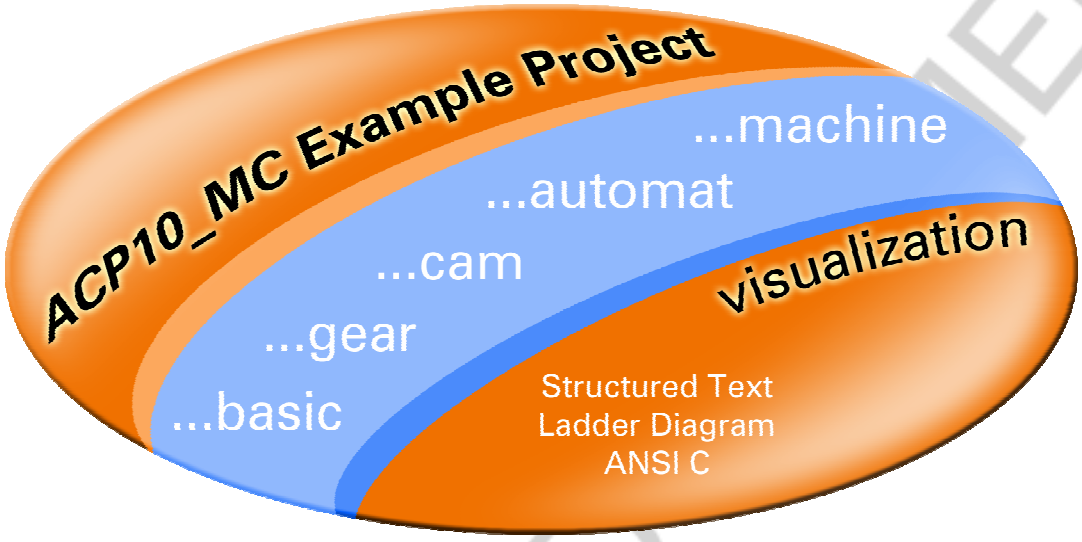
You can test your routines using error configurations (e.g. movement with a speed value set too high, etc.)

Note:

The notes for implementing the previous programming examples were taken from the Motion Control sample project. The routines for ACOPOS control are implemented the same way in this reference project. To keep things simple, some confirmation messages were not included in the examples.

7. MOTION CONTROL SAMPLE PROJECT

A sample project for Motion Control applications, which demonstrates selected functionalities from the ACP10_MC library, is included in the Automation Studio installation program. This makes it possible to use this project as a template for complete implementation of the axis functions.



The sample project is available in multiple variations, programmed in Structured Text (ST), Ladder Diagram (LD) and in ANSI C.

The corresponding project versions can be found in the Automation Studio installation directory under:

"...\BR_AS_25x_Lxxx\Samples\Motion\..."

7.1 Components

Which components does this project include?

The project is based on a SG4-CPU (CP360) for controlling two ACOPOS modules. Powerlink (interface card F787) is used as communication network.

The **drive components** (ACOPOS modules) in the project are **managed using the NC deployment table**. Furthermore, INIT parameter modules are also provided in the project for the NC objects "real axis" and "virtual axis" (see **TM410 Motion Components**).

The core functions of the ACP10_MC library are shown in a total of four **control tasks** (in the respective programming language):

"basic": Various **basic functions for drive control** are structured here in a complete sequence. Additionally, this task also contains a routine for handling potential error situations.

The other tasks also contain the basic functions and the error handling from the "basic" task, but with various expansions for multi-axis functionalities as well (see **TM441 Motion Control Multi-Axis Functions**):

"gear": **Function blocks for linking the axis** (slave axis) **via an electronic gear** to a corresponding linking signal (e.g. master axis, etc).

"cam": Function blocks for connecting the axis via a cam profile to a corresponding connection signal.

"automat": Function blocks for configuring and controlling the **cam profile automat**.

The control programs mentioned above provide a complete picture of how the functions are used with the help of the necessary routines. They can be considered as a reference for setting up and configuring positioning tasks using the ACP10_MC function blocks. Control of the sequences is grouped together into a separate operating structure for each task.

Additionally, a variation is displayed in the **"machine"** task that combines the task controller for a master-slave connection ("basic" for the master axis, "automat" for the slave axis) in a central operating structure. Corresponding commands (start, stop, etc.) are distributed on both of the "basic tasks" and important status information is displayed in the operating structure.

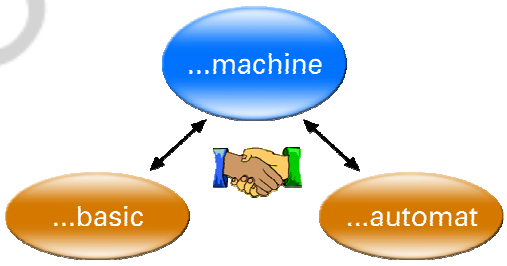


Fig. 39: Machine functionality

7.2 Start-up

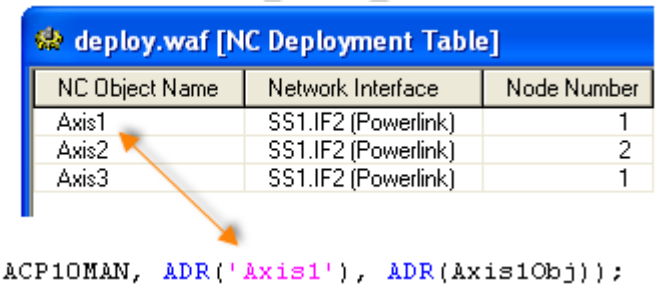
The tasks from the sample project can be used for immediate commissioning of the corresponding function blocks.

> **Note:**
>
> The specific hardware being used doesn't really matter. There only has to be a correct drive configuration ("flexible NC configuration in AS 2.x" with NC deployment table) in a project.

The desired task can be exported from the sample project using the known methods and imported into a target project. Now, the "**ncaccess**" function must simply be adjusted in the Init subprogram in order to use the control task for an ACOPOS in the project:

The NC object name from the axis mapping must be entered in the function.

Automation Studio 2.x




```
ncaccess(ncACP10MAN, ADR('Axis1'), ADR(Axis1Obj));
```

Automation Studio 3

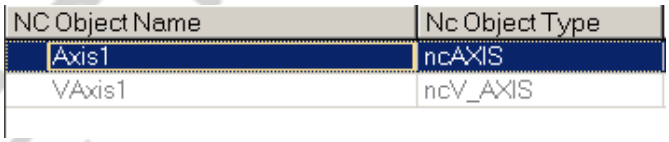


Fig. 40: NC Mapping Table in Automation Studio 3

Therefore, it is possible to use the task to control the virtual or real axes on an ACOPOS as needed.

A control structure ("gAxis...") is provided to operate the sequences in each task. This can be done from the watch window.

The NC software object "error text module" is required in the project to display error texts. The **MC_ReadAxisError** function block uses a corresponding module ("module name").

Notes:

The reference of the slave axis as well as the master axis is required for the tasks with multi-axis applications ("gear", "cam" or "automat"). These three tasks are each controlled by a global structure variable "gAxisSlave". As a result, **only one of the three tasks can be active in the project at a time**.
The "basic" task for controlling a master axis can be used to test multi-axis functions on a drive

The NC software object "cam profile" (in the sample project: "profile") is required in the project for connecting via a cam profile (tasks "cam" and "automat"). However, this profile can also be taken from the sample project.

Task: "Implementing the sample task for basic functions"

Import the "basic" task from the sample project into your application project. Make the necessary adjustments to control your ACOPOS with this task.

Get familiar with the sequence (identical to the previous programming exercises) and get the application running. Test the individual functionalities.

8. MANAGING ACOPOS PARAMETERS

A few function blocks are provided by the ACP10_MC library for managing the parameters on the ACOPOS (selective reading and setting).

The software structure of the ACOPOS has already been looked at closely in previous training material.

A brief review:

The ACOPOS has a large number of parameters. These parameters are used to set the ACOPOS to the connected **hardware**, and to control and **check positioning sequences**. The parameters are generally managed using the NC operating system, which is connected to the the NC manager on the controller via the common network.

However, direct access (read and write) to different parameters is also required for specific applications during runtime. The ACP10_MC library supplies the corresponding tools.

> **Note:**
>
> This is how the ACOPOS configuration can be adjusted to meet special circumstances by making specific changes. The "Smart Process Technology" functions are also operated using separate ParIDs on the ACOPOS.

The following section will provide a brief overview of the possibilities for managing ACOPOS parameters.

> **Note:**
>
> These functions are only offered by the B&R drive solution. These are solely B&R-specific function blocks. The use of these function blocks (operation, status check, etc.) is however identical to the PLCopen Motion Control standard.

The different function blocks for ACOPOS parameter management optimally cover various demands.

8.1 Initializing and reading individual parameters

Individual parameters can be initialized and read one time or cyclically.

Function blocks for initializing/reading one time:

- **MC_BR_WriteParID**
- **MC_BR_ReadParID**

Individual ParIDs can be initialized and read automatically with each "manager cycle" (cycle time of the NC manager) by setting up a cyclic transfer:

- **MC_BR_InitCyclicWrite**
- **MC_BR_InitCyclicRead**

Note:

Typical applications for these two functions would include value monitoring or a control loop. The control loop can run on the controller. "Actual values" are obtained cyclically. The set values determined here can then be cyclically transferred to the ACOPOS.

Note:

Reading ParIDs is a clear-cut procedure. When doing this, the value of the resepective parameter on the ACOPOS is determined.

Initializing means making a specific value "operational" for a parameter on the ACOPOS. Initialization is not always automatic after transfer. Certain functions only transfer parameter values to the ACOPOS. In these cases, the initialization can be carried out at a later point.

A further function block can be used to establish independent cyclic communication for a ParID between individual ACOPOS units:

- **MC_BR_InitMasterParIDTransfer**

This function bock is used for the cam profile automat's additive axes and for **Smart Process Technology** functions.

8.2 Transfer and initialization of parameter sets

All other function blocks for managing ACOPOS parameters support the transfer and initialization of multiple ParIDs.

There are a few different ways to establish these groups:

- The parameters entered in an **ACOPOS parameter table** are individually transferred to the ACOPOS with each NC Manager task cycle and **immediately initialized** by using the **MC_BR_InitParTableObj** function block.

AP par01.waf [ACOPOS Table]

Parameters	Define Name	ID	Value
Parameters			
Resolver Slot 2			
Encoder1: Type	ENCOD_TYPE	97	5
Encoder1: Encoder scaling: increments pe	SCALE_ENCOD_INCR	109	16384
Encoder1: Resolver polepairs per encoder	ENCOD_POLEPAIRS	203	1

Fig. 41: ACOPOS parameter table

- In the "parameter list", the parameter configuration and the value assignment can be easily adjusted during runtime. There is an additional data type for this purpose that contains an element for the parameter ID and an element for the parameter value. You can use this data type to create an array (▫ list) in the task. Entries in this "list" can then be changed from the application program during runtime.

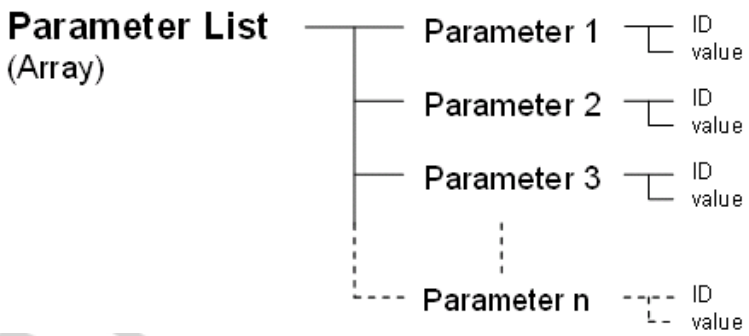


Fig. 42: Data array as parameter list

All parameters in the **parameter list**, as for the ACOPOS parameter tables themselves, are transferred to the ACOPOS with each NC Manager task cycle and **immediately initialized** using the **MC_BR_InitParList** function block.

Note:

Unlike the initialization variant, it's possible to change the parameters that are available, as well as their values, in the predefined ACOPOS parameter tables while the application program is running.

- The "**parameter sequence**" uses the same array as the "parameter list".

  Unlike the ACOPOS parameter tables or parameter lists, the parameters in the **parameter sequence** are not transferred individually, but as a data block in the NC Manager idle time task. Once transferred to ACOPOS, they are **not immediately initialized, but rather saved as a "recipe"**. The **initialization** takes place independent of the data transfer and **only after the command "initialize parameter sequence"** is run.

  For the transfer, the **MC_BR_DownloadParSequ** function block is required with which a sequence is placed at a specific index on the ACOPOS. Initialization for a specific sequence (index) is possible using the **MC_BR_InitParSequ** function block.

  As a result of the index assignment, it is also possible to place **multiple parameter configurations together on the drive and initialize them selectively**.

Note:

Detailed information regarding the operation of these functions during program execution is available, as usual, in the Automation Studio online help.

8.2.1 Additional information

Each of these function blocks is equipped with an input parameter "DataAddress" for connecting the parameter groups "parameter list" and "parameter sequence". An **ACP10DATBL_typ** structure variable still must be provided here as intermediary:

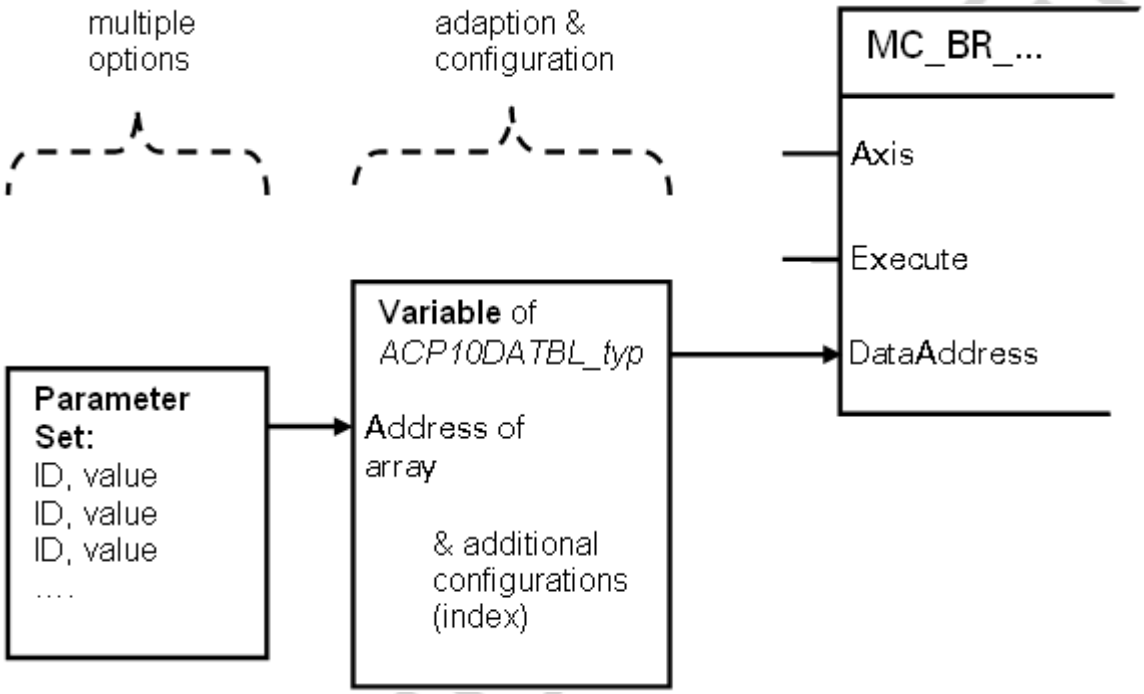


Fig. 43: Connecting a parameter group via an ACP10_DATBL_typ variable

In the schematic above we see a **simplified example of the procedure**. The **parameter group** is located on the left as a **parameter array**. The group is connected to the function block via the variable for transfer configuration (**ACP10DATBL_typ**). The parameter download can be made once this assignment has been made correctly ("Execute= 1").

Note:

Detailed information about operating these functions can be found in the Automation Studio online help.

9. SUMMARY

Power function blocks are provided for controlling the B&R drive solution. These are designed based on the PLCopen Motion Control standard and feature a uniform design regarding functional usage. Once the programmer is familiar with this standardized operation, he can then begin to combine the corresponding function blocks for the process from the Motion Control library's pool of functions (ACP10_MC).

The automatic sequence can be optimally implemented using a step sequencer. Safety routines expand the sequence to a complete positioning application.

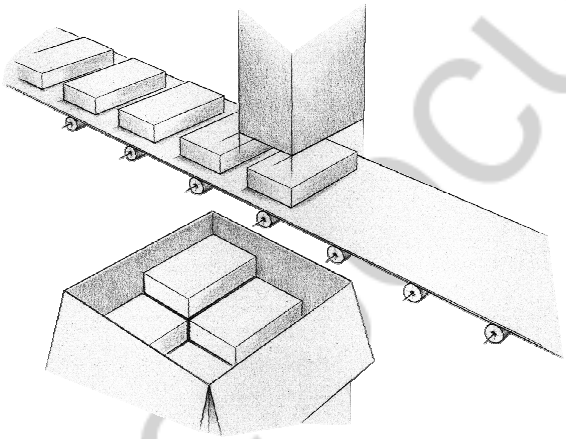


Fig. 44: Palletizing

In addition to the standard, specific function blocks are also provided for controlling the drives. These are used to operate special ACOPOS functions. This enables the programmer to utilize the full functional capability for solving positioning tasks.

10. APPENDIX

Motion Control Basic Functions (ACP10_MC):

Drive preparation:
MC_Power
MC_Home
MC_BR_BrakeOperation
MC_BR_InitModPod
MC_BR_LoadAxisPar
MC_BR_SaveAxisPar
MC_BR_InitAxisPar
MC_BR_InitAxisSubjectPar

Basis movements:
MC_MoveAbsolute
MC_MoveAdditive
MC_MoveVelocity
MC_BR_MoveAbsoluteTriggStop
MC_BR_MoveAdditiveTriggStop
MC_BR_MoveVelocityTriggStop
MC_BR_EventMoveAbsolute
MC_BR_EventMoveAdditive
MC_BR_EventMoveVelocity
MC_Stop
MC_Halt
MC_SetOverride

Determining the drive status
MC_ReadStatus
MC_ReadActualPosition
MC_ReadActualVelocity
MC_ReadActualTorque

Determining and acknowledging drive errors
MC_ReadAxisError
MC_Reset

Digital input/output signals
MC_ReadDigitalInput
MC_ReadDigitalOutput
MC_WriteDigitalOutput
MC_DigitalCamSwitch

Position measurement
MC_TouchProbe
MC_BR_TouchProbe
MC_AbortTrigger

Notes

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM241 – Function Block Diagram (FBD)
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB)
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I
TM261 – Closed Loop Control with LOOPCONR

TM400 – The Basics of Motion Control
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors

TM500 – The Basics of Integrated Safety Technology
TM510 – ASiST SafeDESIGNER

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 – Automation Net PVI
TM710 – PVI Communication
TM711 – PVI DLL Programming
TM712 – PVIServices
TM730 – PVI OPC

TM800 – APROL System Concept
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming
TM890 – The Basics of LINUX

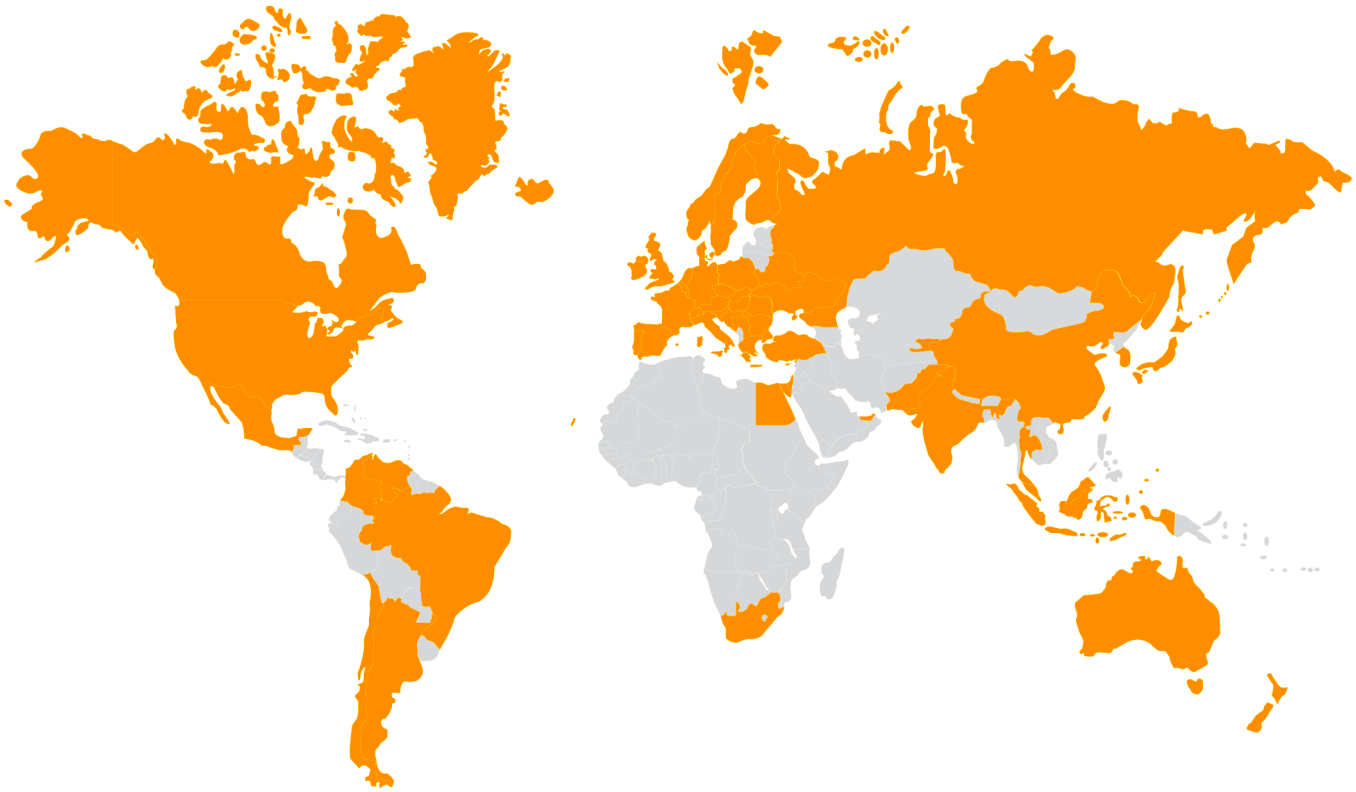
**) see Product Catalog

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Strasse 1
5142 Eggelsberg
Austria
Tel.: +43 (0) 77 48 / 65 86 - 0
Fax: +43 (0) 77 48 / 65 86 - 26
info@br-automation.com
www.br-automation.com

TM440TRE.00-ENG 0907

140 offices in more than 55 countries - **www.br-automation.com/contact**



Australia • Argentina • Austria • Belarus • Belgium • Brazil • Bulgaria • Canada • Chile • China • Colombia • Croatia • Cyprus
Czech Republic • Denmark • Egypt • Emirates • Finland • France • Germany • Greece • Hungary • India • Indonesia
Ireland • Israel • Italy • Japan • Korea • Luxemburg • Kyrgyzstan • Malaysia • Mexico • The Netherlands • New Zealand
Norway • Pakistan • Poland • Portugal • Romania • Russia • Serbia • Singapore • Slovakia • Slovenia • South Africa
Spain • Sweden • Switzerland • Taiwan • Thailand • Turkey • Ukraine • United Kingdom • USA • Venezuela • Vietnam