Using BASIC Scripts at a FAULHABER MotionController V3.0

Summary

This ApplicationNote is a comprehensive introduction into automation of FAULHABER MotionController V3.0 using their local scripting capabilities.

As these scripts usually will have to deal with enabling and disabling the drive as well as changes of the Mode of Operation (OpMode) the ApplicationNote starts with a compact introduction into the typical interaction with the drive control part of such a servo-drive (Chapter How to interact with the MC drive function).

Chapter The FAULHABER MC BASIC explains the capabilities and limitations of the local BASIC scripting engine

Some suggestions on how to create a program structure and some useful coding patterns are collected in chapter Patterns for embedded scripts followed by two heavily commented examples.

Some more advanced patterns are introduced in chapter Additional Patterns.

Applies To

All FAULHABER MotionController V 3.0:

MC 5004

MC 5005

MC 5010

MCS

Related FAULHABER Documents

Document	Description
Motion Manager 6	Instruction Manual for FAULHABER Motion Manager PC software
Quick start description	Description of the first steps for commissioning and operation of FAULHABER Motion Controllers
Drive functions	Description of the operating modes and functions of the drive
Programming Manual	Description of the BASIC based programming environment of the FAULHABER MotionController V3.0

How to interact with the MC drive function

Before we start coding we need to understand the basic concept of how to interact with the drive function of the FAULHABER MotionController (MC).

The purpose of the MC is to control the movement of a single motor in a closed loop. Additional sensors might be connected to the MC to limit the movement or to reset the motor position to 0 at a defined position during one of the homing sequences.

Each of the OpModes can be adjusted to the application by setting a bunch of parameters. But that's static of course. One of the reasons to use a local script might be to either change these parameters during operation or to switch between different OpModes.

Typically changing parameters can be done by a master controller connected to the MC via one of the communication interfaces. However if there is no master controller at all or if some of the changes shall be executed partly autonomously it might be an advantage to use local scripting at the MotionController directly.

Examples could be:

- Executing a homing sequence after each power up and then switching to position control with an analog reference value (**A**nalog **P**osition **C**ontrol).
- Implementing a discrete break/enable interface for a controller in stand-alone mode.
- Changing to a predefined different set of parameters during a load cycle, even if connected to a master but without having to access all the parameters from the master PLC.
- Using analog inputs to change limits of the feedback control such as maximum speed or torque limits.
- Create a predefined complex motion profile by subsequently changing profile parameters and control limits while executing the movement.

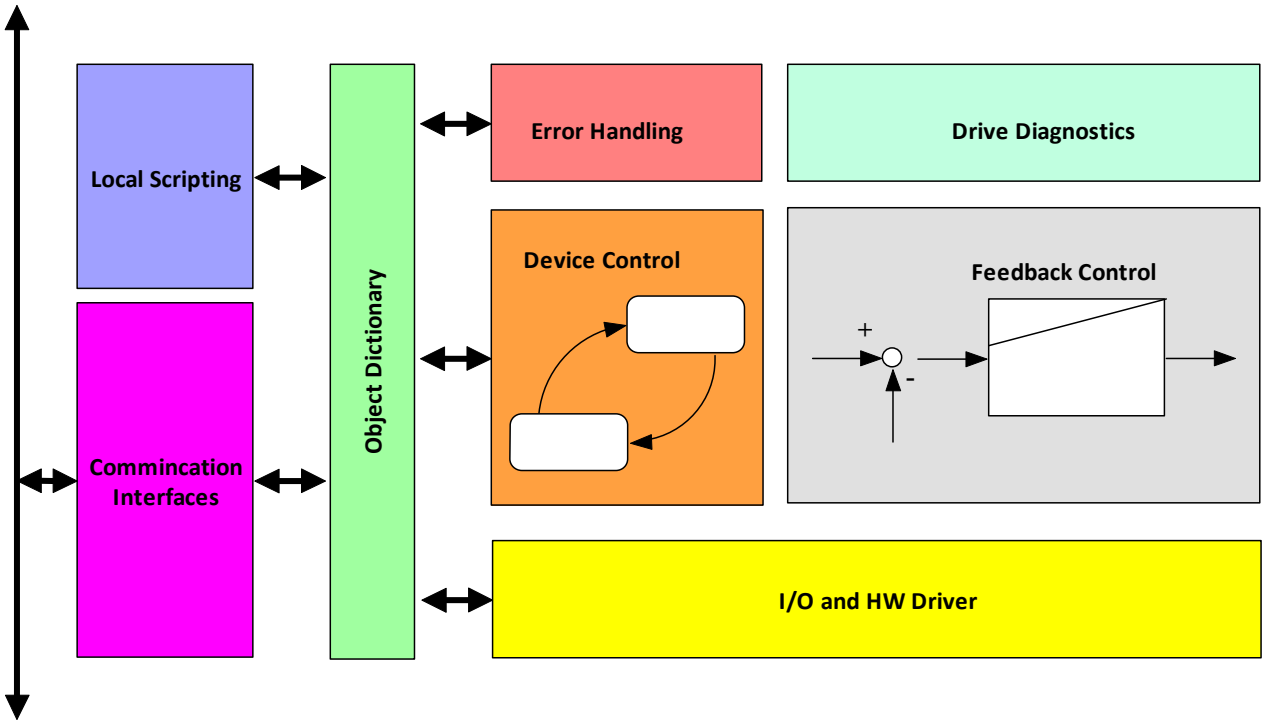


Figure 1 Main Parts of the MC internal firmware

Accessing MC parameters

The FAULHABER MotionController V3.0 is a servo-drive compatible to the CiA 402 / IEC 61800-7-200 standard used in CANopen and EtherCAT environments. All of the parameters, references, control inputs and actual values of the drive are collected in the Object Dictionary (OD).

Any access to the application is routed via one of the parameters collected in the OD. Consequently the parameters are referred as objects. The basic operation here is a read- and/or write-access to the parameters collected here – read object and write object[1]. So basically to interact with such a drive will result in a sequence of read and write commands such as:

- Set Target Velocity to xxxx
- Read Actual Velocity
- …

[1] CANopen and EtherCAT additionally provide optimized access to a predefined set of parameters for real-time data exchange by so called Process Data Objects (PDO), which define a set of parameters to cyclically exchanged during real-time operation.

All parameters/objects are identified by a 16 bit index – the parameter number and a 8 bit sub-index, allowing for structured parameters.

Simple Parameter

Idx	Sub	Parameter
0x607A	00	Target Position

Idx	Sub	Parameter
0x60FF	00	Target Velocity

Structured Parameter

Idx	Sub	Parameter
0x2311	00	Digital I/Os
	01	Logical Input State
	02	Physical Input State
	03	Output State

Table 1

Within BASIC scripts for the MotionController V3.0 the commands to access the parameters are:

SETOBJ <index>.<Sub> = <value> e.g.: **SETOBJ $607A.$00 = 10000**

a = GETOBJ <index>.<Sub> e.g.: **a = GETOBJ $2311.$01**

Parameter index and sub-index are usually denoted as hexadecimal numbers. The $ sign is used to denote a hexadecimal number within the BASIC environment.

Main units within the MC drive

To easily use the drive a general idea about the different functional parts within the MotionController might be useful (Figure 1).

Device Control

Enables or disables the motor control. Parameters are the controlword **0x6040.00** and the statusword **0x6041.00** of the servo-drive.

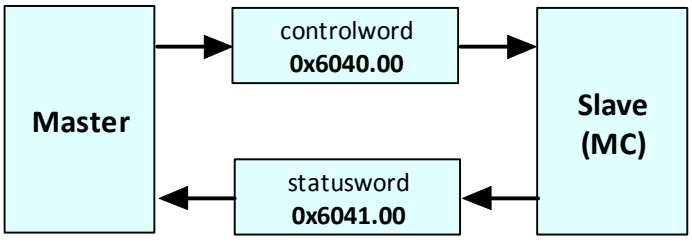


Figure 2 Interaction between master controller and servo-drive

Selection of the OpMode (see Table 4) using the parameter Modes of operation (**0x6060.00**).

Feedback Control

The motor control unit controls the torque-, velocity – or position of a motor in a closed loop. The feedback-control will try to follow the target values of the selected OpMode. Actual values are calculated. Parameters are:

Loop	Target Values	Actual Values	Scaling
Position[1)]	0x607A.00	0x6064.00	Motor Encoder increments
Velocity[1)]	0x60FF.00	0x606C.00	min^{-1}
Torque	0x6071.00	0x6077.00	nominal torque / 1000
Voltage	0x2341.00	0x2340.xx[2)]	10mV / LSB

[1)] position and velocity scaling can be changed by using the factor group

[2)] sub-index depending on the type of motor – DC or BL

Additional parameters are:

- Torque limits
- Limits for acceleration and deceleration
- Limits for the motor speed or the profile speed
- Position limits
- Filter settings for actual speed

Drive Diagnostics

Supervises the controlled motion and updates the thermal models. Drive Diagnostics will check for any limits being reached (Software Position Limits or limit switches) and will check whether target position or target speed are reached. The results are concentrated in the device status word 0x2324.01. Additional information is available via the supply voltages or the calculated internal temperatures.

Error Reaction

Conditions that are considered to be an error are collected in the FAULHABER error word 0x2320.00. Different automatic reactions to the different errors can be configured using the error masks in 0x2321.xx. If connected to the system via one of the communication systems additional EMCY messages will inform the master about the detected errors. This however does not apply to the local scripts. They don't receive error messages but can react to any combination of flags in the device status word.

I/O and HW-drivers

This unit is responsible for the update of the discrete interfaces. The type of interfaces connected to the drive is parametrized and actual values can be read or written in the case of digital outputs. Analog inputs can be pre-scaled before being used in the feedback control without any scripting involved. In addition to

these built in features scripts could read analog inputs and use their actual values to manipulate parameters of the feedback control, different from the standard ones like e.g. limits for speed or torque.

Interaction with the drive state machine

Servo-drives according to CiA 402 need to be enabled or disabled stepping through the drive state machine in Figure 5.

Auto-enable the control and power-stage

The drive state machine is a powerful means to change the drive state from disabled to enabled or to change to a quick stop with defined reaction independent from the selected OpMode. However most stand-alone applications won't need or use this functionality. So if, for instance the application requires analog position control, a script could be used to add an initial homing sequence but other than an initial start no interaction with the state machine is involved.

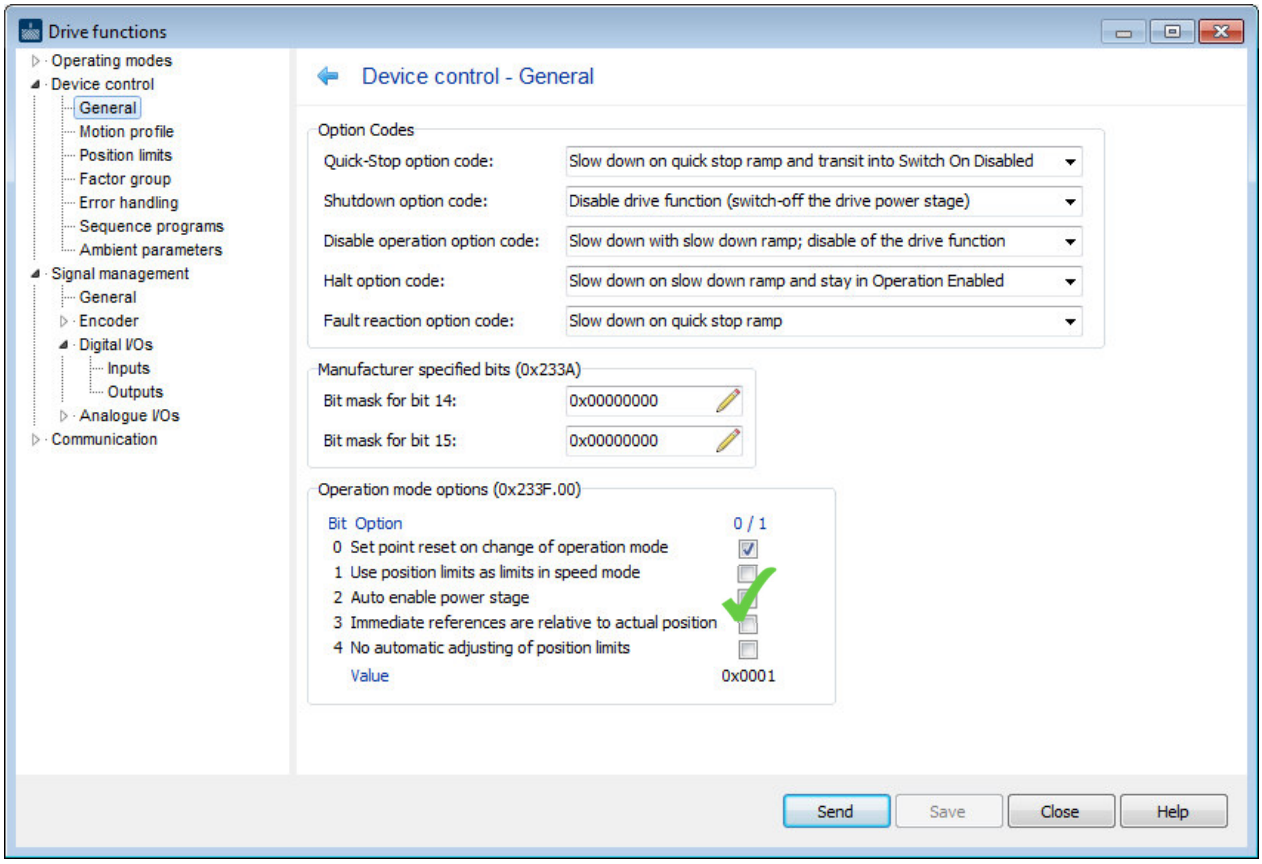


Figure 3 Auto-enable option at the General tab of the Drive functions / Device control

In these cases we might simply configure the MotionController to auto-enable the power-stage directly after reset and don't care about it at all (Figure 3).

In case of a thermal overload or in case of overvoltage the drive will protect motor and electronics by disabling the power-stage which is a direct transition from the Operation-Enabled state to Switch-On-Disabled. An auto-enabled power-stage won't react to this unless the drive is explicitly reset.

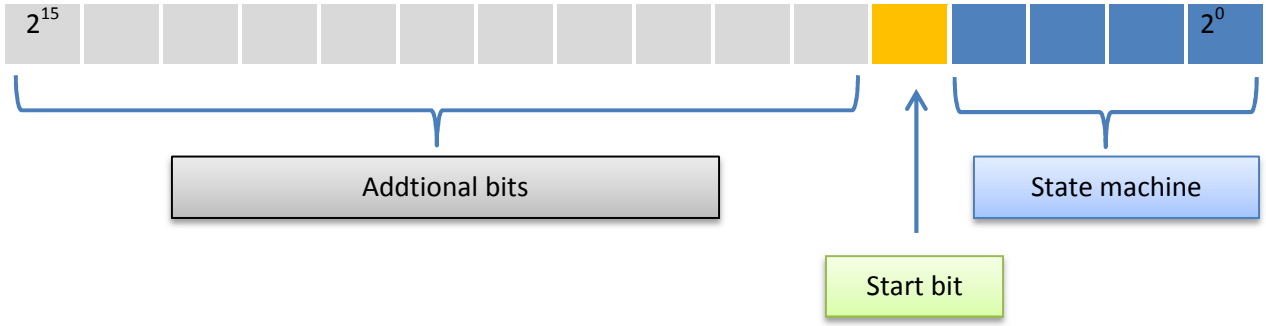
Dealing with the drive state machine

After reset the drive will reach the Switch-On-Disabled state and wait for the user to switch on the drive.

If we explicitly want to enable the control and power-stage out of our program sequence, we need to send the appropriate commands by writing to the controlword. We have to monitor the actual state by reading the statusword, each being a 16 bit unsigned parameter.

Dealing with the controlword and statusword is a little complicated because it's a mix of different flags having different tasks (Figure 4).

Structure of the controlword



Structure of the statusword

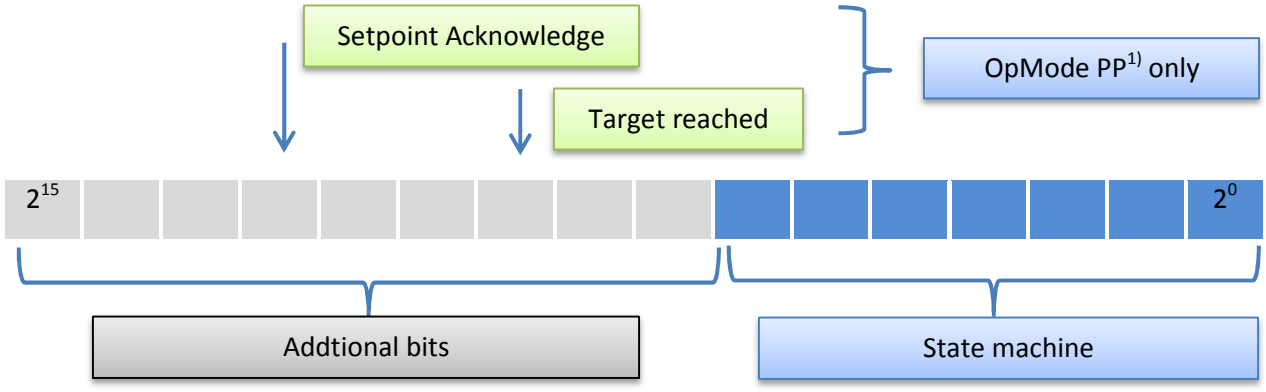


Figure 4 Contents of the statusword and control word of the servo-drive

[1] see Table 4

To interact with the state-machine we have to code the commands in the lower 4 bits of the controlword and read the actual state out of the lower 7 bits of the statusword. So most likely some kind of bit masking

using bit oriented logic will be required within the scripts (see chapter patterns below). The reason is the limited band-with of field busses. Usually the controlword and the statusword are to be exchanged cyclically between master and slave and a compact combination of the most important flags helps to increase the update rate.

The commands coded in the lower 4 bits of the controlword are listed in Table 2; the actual state coding is listed in Table 3. The numbers of the transitions in Table 2 are the same as noted in Figure 5.

Command (transition)	controlword
Shutdown (2,6,8)	0x0006
Switch on (3)	0x0007
Enable Operation (4,16)	0x000F
Disable Operation (5)	0x0007
Disable Voltage (7,9,10,12)	0x0000
Quick Stop (11)	0x0002

Table 2 command sent to the drive state machine

State	statusword	bits						
Switch on Disabled	0x..40	1	0	0	0	0	0	0
Ready to switch on	0x..21	0	1	0	0	0	0	1
Switched on	0x..23	0	1	0	0	0	1	1
Operation Enable	0x..27	0	1	0	0	1	1	1
Quick Stop	0x..07	0	0	0	0	1	1	1
Fault	0x..08	0	0	0	1	0	0	0

Table 3 state machine states coded in the statusword

Enabling the drive (→ in Figure 5)

After a reset the drive will be in Switch on Disabled state. To enable the drive, which is a transition to the Operation Enabled state, we have to subsequently:

- send the Shutdown command (0x 00 06)

- at least send the Enable Operation command (0x 00 0F)[2].

Disable the power-stage (→ in Figure 5)

To simply disable the power-stage, a Disable Voltage command is best suited. It will switch to the Switch on Disabled state out of most states.

- Send Disable Voltage (0x 00 00)

Stop the Motor and Disable the power-stage (⟶ in Figure 5)

To explicitly stop the drive and then disable the power-stage the easiest way is to switch the drive into the Quick-Stop state. The method how to stop the drive is configured using the object Quick Stop Option Code (0x605A.00). Default is: stop at quick stop ramp and disable the power-stage.

- Send Quick-Stop command (0x 00 02)
- Transition to Switch on Disabled can be done automatically

Transition between states might take some time. If the drive is going to be disabled the motor might have to be stopped and a configured brake might need some time to be activated. Therefore before sending a next command to the state-machine it is important to check the actual state. Commands will be ignored if no related transition is available in the current state.

Even if a script is planned to be auto-started directly after the reset of a drive, during development and test the drive might actually be in a state different from the Switch on Disabled when the script is started. To ensure a proper start it might be a good idea to send a Disable Voltage directly at the start of the program sequence.

[2] The switch on command is not necessary for a FAULHABER MotionController. The purpose of the switch on command in a servo drive is to enable the motor power supply. As the drives are directly connected to a low voltage power supply there is no need to control a mains contactor.

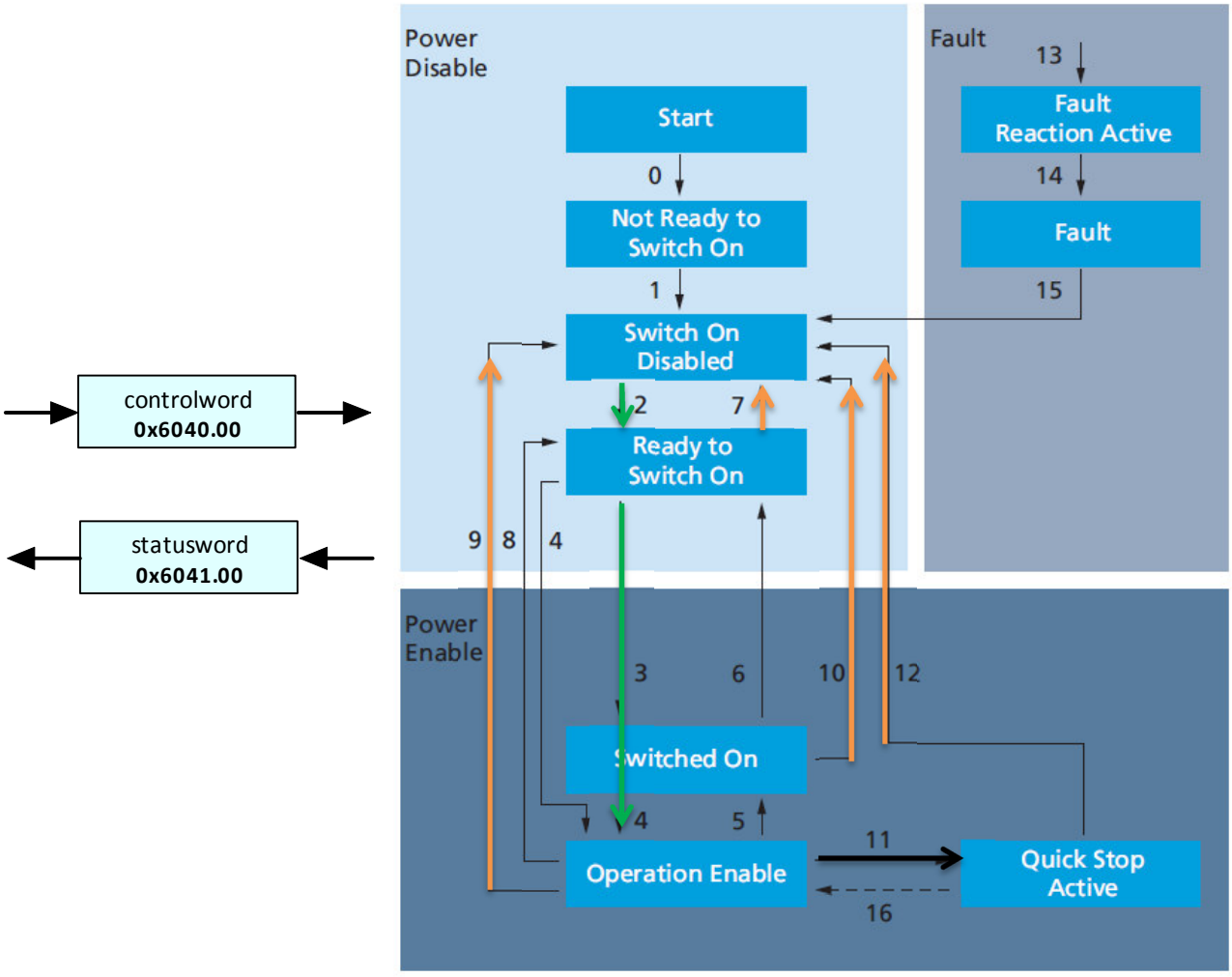


Figure 5 Drive machine state of a CiA 402 servo-drive

Control OpModes

Switching between different OpModes can be done by writing the OpMode to the Modes of Operation parameter (0x6060.00). The currently active one can be read out of the Modes of Operation Display (0x6061.00) but that's usually not necessary.

Operating Mode		Modes of Operation
ATC	**A**nalog **T**orque **C**ontrol	-4
AVC	**A**nalog **V**elocity **C**ontrol	-3
APC	**A**nalog **P**osition **C**ontrol	-2
Volt	Direct **Volt**age Mode	-1
-	Control disabled	0
PP	**P**rofile **P**osition	1

PV	**P**rofile **V**elocity	3
Homing	**Homing** Mode	6
CSP	**C**yclic **S**ynchronous **P**osition	8
CSV	**C**yclic **S**ynchronous **V**elocity	9
CST	**C**yclic **S**ynchronous **T**orque	10

Table 4 List of available OpModes

So if we want to start with a homing sequence and switch to APC afterwards the commands would be:

- (Start the drive if not done automatically)
- Set OpMode Homing: **SETOBJ $6060.00 = 6**
- Start the homing writing a positive edge to bit 4 of the controlword and wait for the homing sequence to be finished
- Set OpMode APC: **SETOBJ $6060.00 = -2**

Speed control out of a script can either be done using **CSV** or **PV** mode.

PV mode will respect the limits of acceleration and deceleration, **CSV** will not. As there is no penalty for the **PV** in terms of additional commands, consider using **PV** out of scripts.

Position control out of a script can either be done using **CSP** or **PP** mode.

PP mode will respect the limits of acceleration and deceleration and profile speed but does require the motion to be started explicitly by generating a positive edge in the start bit of the controlword (Figure 4). So **PP** usually is the more comfortable OpMode but does require additional steps (see examples).

Differences between local BASIC scripts and remote control

PLC based automation has to use one of the communication interfaces to access the parameters. After the startup CANopen and EtherCAT rely on PDOs to exchange a predefined set of parameters. Data exchange between the communication and the program is then done by means of global variables. The execution of the PLC program and the communication will not be synchronized unless an explicit SDO read or write has been implemented. So even if a command is written to the variable representing the controlword this does not imply the value is sent immediately to the slave. So if we need to send a sequence of values to a single parameter e.g. to set the start bit in the controlword and reset it again, a PLC program always needs to explicitly check the reaction of the drive in the status word.

Even using a .vbs script out of the FAULHABER MotionManager we have to check the response of the drive after each command to avoid a communication overload.

These precautions are not necessary for controller based BASIC scripts. Each write-access to a parameter using the SETOBJ will be executed at the very same time when the program line is interpreted. The next line is executed only, if the previous one is completed, so there is a strictly synchronous behavior and no communication overload.

The FAULHABER MC BASIC

The BASIC dialect

FAULHABER MotionController V3.0 uses a BASIC dialect to code scripts that can be executed directly at the controller. The MotionController interprets each line and executes the code. There is no compilation involved. However the development-environment integrated into the MotionManager implements some preprocessing of the scripts. Direct download of scripts without using the MotionManager is not supported. Debugging and single stepping are supported by the FAUHABER MotionManager using any of the supported communication interfaces (USB, RS 232, CANopen).

Please refer to the programming manual to get additional information about the debugging support of the FAULHABER MotionManager.

Main features of the scripting environment are:

- Support of standard BASIC control structures
- BASIC dialect extensions
  - to read and write of drive parameters
  - to deal with bit-wise logic
  - to add time measurement and dead-time handling
- Up to 8 programs can be stored at the MC
- One of the stored programs can be configured to be auto-started after a reset of the drive
- Access to the programs can be protected by a key-parameter
- Up to 26 global variables (a … z) can be used
  - All variables can be stored in / re-loaded out of the internal EEPROM using **SAVE** or **LOAD** and a comma separated list of variables
  - All variables can be directly accessed by a master system via the object 0x3005.xx in the Object dictionary

SAVE / LOAD stores the variables in the internal parameter EEPROM of the MotionController. It is important to keep the limited write cycles of such a EEPROM in mind. So if we assume a maximum of 10^6 write cycles and do save the counter value of an on-time counter every 1 second, the limit is reached after less than 2 weeks of 24/7 operation!

Control structures and operations

The purpose of a local script is either to execute a sequence of operations automatically (like a batch of commands) or to cyclically execute a control sequence, react to states and inputs and branch into different actions.

Main control structures of a BASIC sequence are therefore the **IF** / **ELSE** decisions and loops. Additionally a **FOR i = 1 To n** loop is available.

```
…
'Read Inputs
a = GETOBJ $2311.01

IF (a = 1) THEN
   'Input 1 is set
   …
ELSE
   …
ENDIF
```

```
:Init
SETOBJ …
SETOBJ …

:MainLoop
'Read Inputs and states
'processing
GOTO MainLoop
```

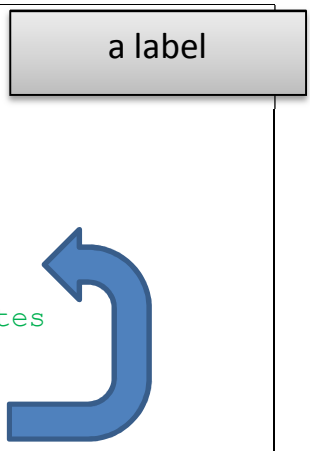


Figure 6 Control structures if/else and loop

Logic operations are

- Standard logic: AND, OR, NOT
- Bit-wise logic: &, |, ~
- Compare: < , > , <> , = , >= , <=

Simple arithmetic[3]

- +, -, /, *
- % modulus

Read and write drive parameters

- SETOBJ <index>.<Sub> = <value> e.g.: SETOBJ $607A.00 = 10000
- a = GETOBJ <index>.<Sub> e.g.: a = GETOBJ $2311.01

Restrictions

Main purpose of the MotionController is the motor control. The resources of the additional scripting engine are therefore limited. Restrictions are:

- As of now there is no support for user generated functions with local variables and return value. **GOSUB**/**RETURN** can be used though
- The up to 8 programs cannot call each other
- Variable names are a – z lower case

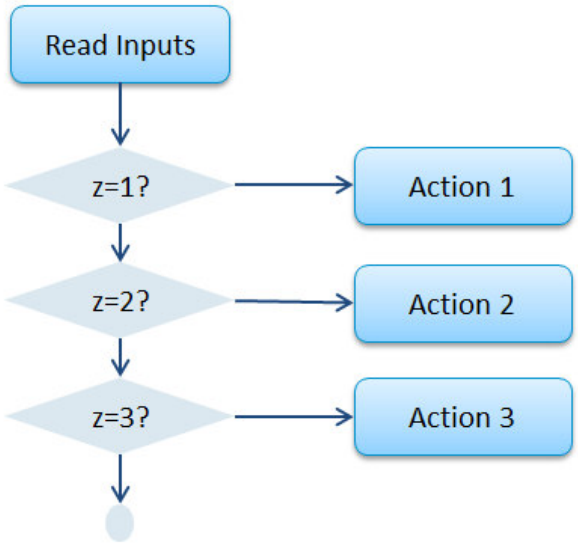
[3] All variables are treated as signed 32 bit integer numbers

- You can't use **GOTO** to jump out of a **IF**/**ELSE**, **GOSUB** is supported
- Firmware versions up to revision H do support up to 3 nested **IF**/**ELSE** levels. Firmware starting from revision I supports up to 15 levels.
- The size of a single program sequence is limited to 4kB.

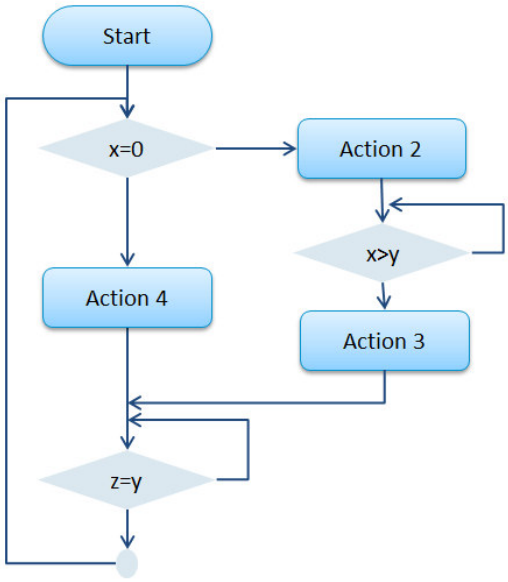
Patterns for embedded scripts

There are some recommended strategies on how to create a control flow of a program and some typical patterns like reacting to a single bit of an input that are different from standard BASIC environments. While it is not mandatory to use them, the use of these patterns is encouraged and represents the intended use of the environment.

Step Sequence vs Flow chart

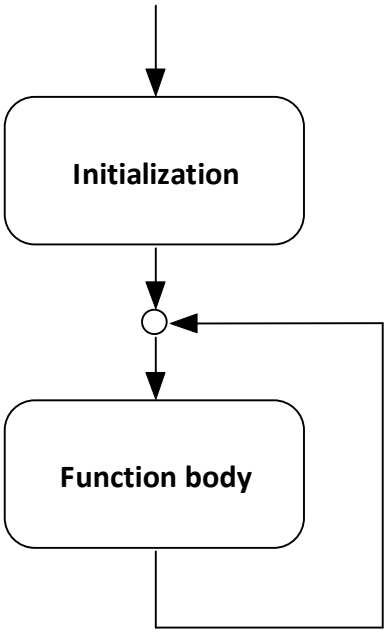


Step sequence



Flow Chart with micro loops

Figure 7 Basic control structures for a script



The overall control structure can of course be traditional flow chart type with micro looping where necessary. Such a micro loop usually will include polling one of the parameters like the statusword or a digital input and waiting for a position being reached or an input being active. Drawback however is, while being stuck in the micro loop it's difficult to react to additional inputs or changes.

PLC environments or even the popular Arduinos use a different approach which we feel is more appropriate for embedded control.

The main structure of these systems is an Initialization executed once at startup and a subsequent main loop executed either triggered by a time (PLC task) or executed continuously.

Figure 8 suggested main loop structure of a BASIC script

The recommendation is to use a continuously executed main-loop and combine it with a step sequence as in the left side of Figure 7. The main advantage compared to the flow chart on the right side of Figure 7 is there are no longer blocking loops.

In a main-loop + step-sequence you can simultaneously wait for reaching a position while checking a time-out and waiting for any of the digital inputs to change. Additionally these step-sequences are well suited to interact with the bit-wise handshake of the controlword and statusword.

Enable and Disable the power stage

Enabling the power-stage requires only a few commands to the device-control via controlword. As mentioned, we do have to check the initial state though.

So a pattern could be:

- Ensure a defined state at the beginning
- If we are in switch on disabled state (which is the default) and whatever start condition is reached:
  - Send the startup sequence
  - Start waiting for being enabled
- If we are enabled check whatever shutdown condition is defined
  - Send any of the shutdown commands
    Quickstop or
    disable voltage

```
:Init
'send a rest to the state-machine because the program could have been started in any state of the device
SETOBJ $6040.00 = 0
'y is used as a step variable for the power stage
y = 0

:MainLoop
'read the statusword
s = GETOBJ $6041.00
'and mask the bits for the state-machine
'this uses bit-wise logic and will cut the lower 7 bits
s = s & $7F

IF (y = 0) THEN
   'check for switch on disabled and a start condition
   IF(s = $40) AND (…) THEN
      'we now are in switch on disabled state
      'send the startup sequence
      'first the shutdown
      SETOBJ $6040.00 = $06
      'send the enable command next. No wait necessary here - we are 'synchronous
      SETOBJ $6040.00 = $0F
      'switch to a waiting state
      y = 1
   ENDIF
ELSEIF (y = 1) THEN
   IF(s = $27) THEN
      'enable operation is reached
      'switch out of the waiting state
      y = 2
   ENDIF
ELSEIF (y = 2) THEN
   'check whatever to disable
   IF (…) THEN
      'send the quick stop
```

```
      SETOBJ $6040.00 = $02
      'switch to wait for reset
      y = 0
   ENDIF
ENDIF

'do whatever is intended

'loop back to the start of the main-loop
GOTO MainLoop
```

Typical use of program variables

The MC BASIC offers 26 variables – lower case letters. This does not really help to create a well readable script but is very efficient in terms of implementation. But then on the other hand, typical quick hack c variables like foo, test and dummy don't help either.

To ease the work, try to establish a standard use of a basic set of variables.

We do use:

	Usage	Example
s	Used for the statusword	`s = GETOBJ $6041.00`
a	Copy of the digital inputs	`a = GETOBJ $2311.01`

z	Step-variable for the step-sequence of the application	`IF (z = 0) THEN GOSUB …`
		`ELSEIF (z = 1) THEN GOSUB …`
		`ENDIF`

Additional variables

	Usage	Example
t	Timer variable. Used either for count-down or for time measurement	`DEF_TIM_VAR t` or `DEF_CYC_VAR t`
p	Position reference	

React to flags and digital Inputs (bit-wise logic)

Another pattern typical for embedded applications is bit-wise logic. Quite often there are some flags combined in a single variable. We have already seen this for the controlword and the statusword.

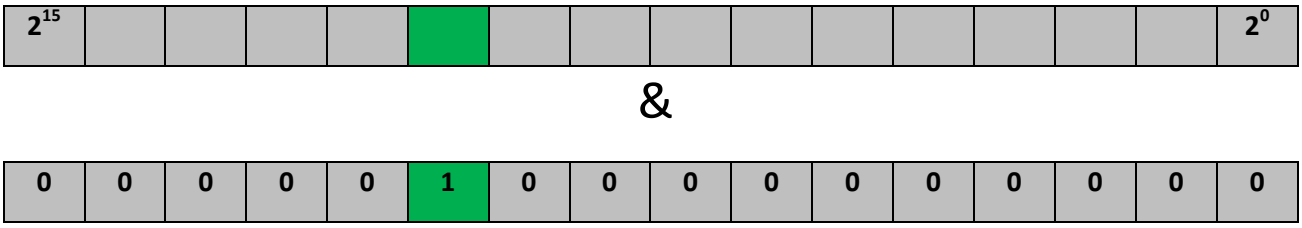
In the statusword we need to check the lower 7 bits to know in which state we are, but then there are the bit 10 and the bit 12 that are used in some of the OpModes for a hand-shake.

So e.g. in PP-mode if we want to check, whether we did reach the target position, we need to check for bit 10. We even have to be careful in sequence of positon steps the bit 10 might be set because we did reach the previous target position. If we now want to check for the next one, we do need two steps:

- Wait for bit 10 being cleared – move started
- Wait for bit 10 being set again – we reached the new position

Anyway, we need to check bit 10 and don't really care for the other bits. In order to cut the bit 10 only we use a bit-wise logic and evaluate

result.bit$_x$ = statusword.bit$_x$ & mask.bit$_x$



The code would be

```
'read statusword
s = GETOBJ $6041.00
'mask bit 10 and check
IF (s & $0400) THEN
   'do whatever is intended
ENDIF
```

React to edges: an action taken only at a rising or falling edge

Sometimes it is convenient to do actions only at the edge of an input signal – statusword or digital input. This might save some intermediate steps that would be necessary if we do only check the level and have to add a separate waiting step to avoid multiple reactions.

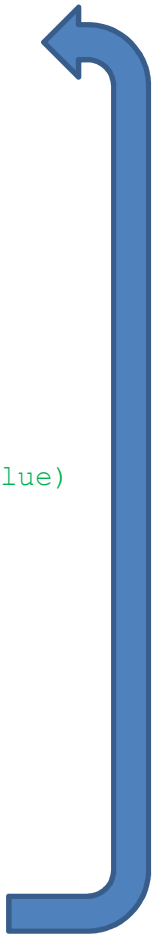
Checking the change of a bit involves bit-wise logic again but also needs a copy of the previous value of the variable to be checked. So to have an example let's enable the power-stage at the positive edge of DigIn1 and disable it again at the negative edge:

We need to read the digital inputs

rising edge	check for newly set bits and in this result mask only the one representing DigIn 1
falling edge	check for newly cleared bits and in this result mask only the one representing DigIn 1

The code would be

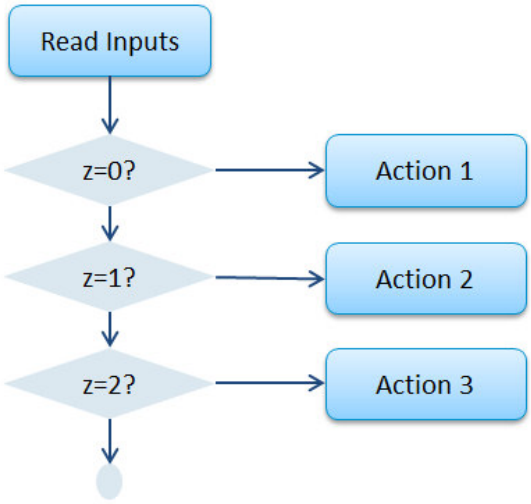
```
:Init
'variable a is used for the digital inputs
'variable b holds the previous value of the digital inputs
a = 0
b = 0

:MainLoop
'read digital inputs
a = GETOBJ $2311.01
'check for newly set bits by evaluating (new value) & ~(old value)
'and only use the lowest bit in the result
IF ((a & ~b) & $01) THEN
   'send enable sequence
ENDIF

'check for newly cleared bits by evaluating (old value) & ~(new value)
'and only use the lowest bit in the result
IF ((b & ~a) & $01) THEN
   'send disable command
ENDIF

'now save the new values in b for the next turn
b = a

GOTO MainLoop
```



Use Sub-routines

The BASIC scripting engine is based on an interpreter. No compiler is involved. So even if the MotionManager does some pre-processing the program sequence is basically a text that has to be read and interpreted line by line.

This is the main reason why we do suggest to use a step-sequence structure using sub-routines to execute the different steps. An example can be pasted into the code using the templates of the MotionManagers editor.




```
z = 1
:MainLoop
IF z = 1 THEN GOSUB Z1
IF z = 2 THEN GOSUB Z2
IF z = 3 THEN GOSUB Z3
IF z = 4 THEN GOTO Z4
GOTO MainLoop

:Z1
'Step 1: if requested state is reached, increase step counter by one
a = GETOBJ $6040.$00
IF a = $0001 THEN
   z = 2
ENDIF
RETURN

:Z2
Return

:Z3
Return
```

Figure 9 Use of sub-routines to execute the different steps of a script

In this case the time to read and interpret the main-loop is shorter compared to an implementation where the different actions are directly coded into the **IF** / **ELSEIF** / **ELSE** construct of the step-sequence in the main-loop.

Create your own program

When you start thinking about stand-alone automation don't start coding directly. BASIC or whatever scripting language is far from native for human imagination. Therefore it's much easier to start drawing.

You could start identifying the primary steps or states your application will be in (Figure 10). This is an example implementing a brake/enable function. DigIn 1 is used as an enable input; DigIn2 is the brake-input. At the positive edge of DigIn1 the drive shall be enabled, a negative edge of DigIn2 shall force the drive to be actively stopped. The drive might be reactivated out of the stop if the brake-input is back again (positive edge). The drive shall be disabled at the negative edge of the enable input.

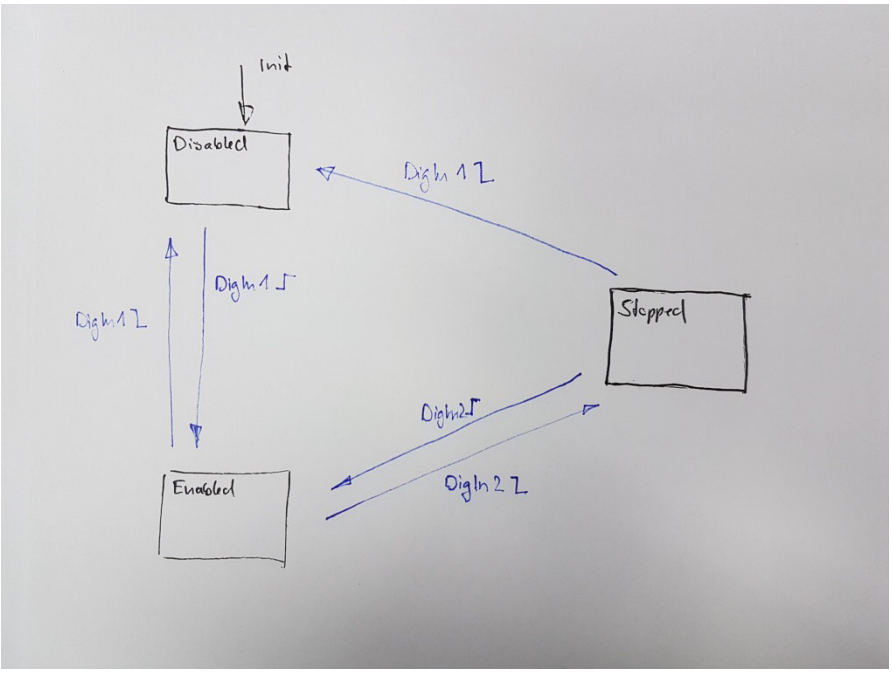


Figure 10 a first approach for a brake / enable program

Drawing here means take a sheet of paper and some colors and

- start with blocks for the steps or states of the action.
- add transitions and conditions
- add actions within the steps/states and at the transitions

If we have to implement some kind of handshake between our program and the MC: e.g. when using PP or enabling the power-stage we might need to add some intermediate steps. So add these steps to your sketch and re-apply transitions, conditions and actions (Figure 11). This is the program logic that can be executed in the main-loop of Figure 8.

What are the preconditions for the script? You might need to add some initial actions to create a defined configuration of the controller and add them to the init step of Figure 8.

Alternatively you could use a flow chart to describe the control flow of a script. Again start with simple action-blocks and branches and refine your model. Add the conditions and a text form of the actions.

These graphical representations are well suited to “simulate” the behavior.

Only if you are satisfied with the simulation, start coding. In most of the cases this will now be only a task of writing down the graphical control flow. Of course we now need to check for whatever bit-masks we might need and which numbers the used parameters do have.

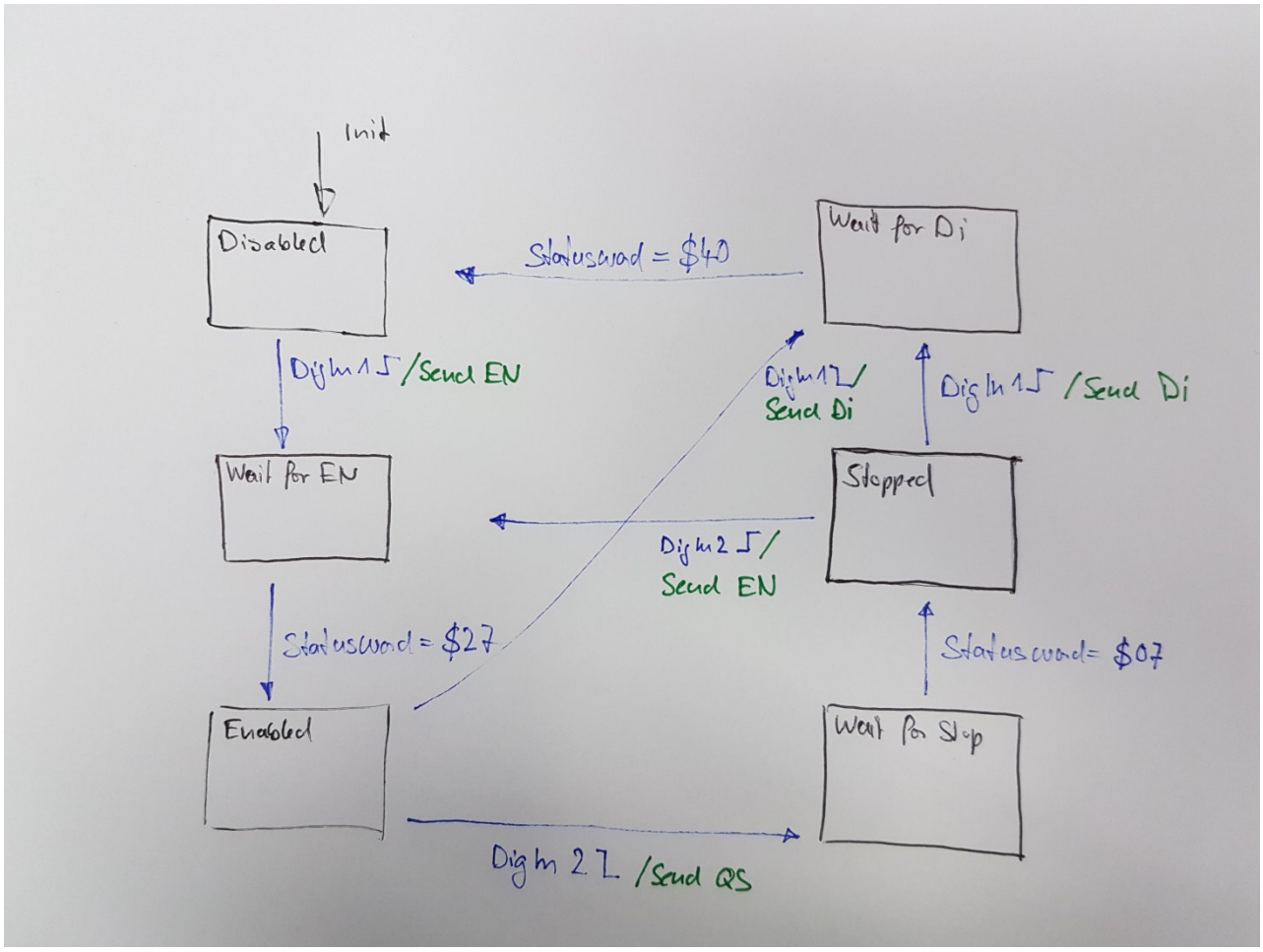


Figure 11 refined diagram having conditions, actions and intermediate steps

Example A (switch between two absolute positions)

The purpose here is to start the drive in reaction to a digital input and cyclically move between two positions while also cyclically changing the profile parameters for acceleration and deceleration. The used OpMode is ProfilePosition Mode (0x6060.00 = 1).

This requires enabling the controller in a first step and then sending the target positions, waiting for being in position and updating the profile parameters. So there are several steps to be taken. A well suited solution pattern for such a problem is a step sequence. A step sequence is a pattern, where only a part of the program is executed in each update cycle, depending of the step in which the program is.

In a PLC environment there is a special diagram to design these step sequences: sequential function chart (SFC). Here however we use an **IF** / **ELSEIF** / **ELSE** construct.

```
If (StepVariable = xxx) Then
…
ELSEIF (StepVariable = xxx) Then
…
END_IF
```

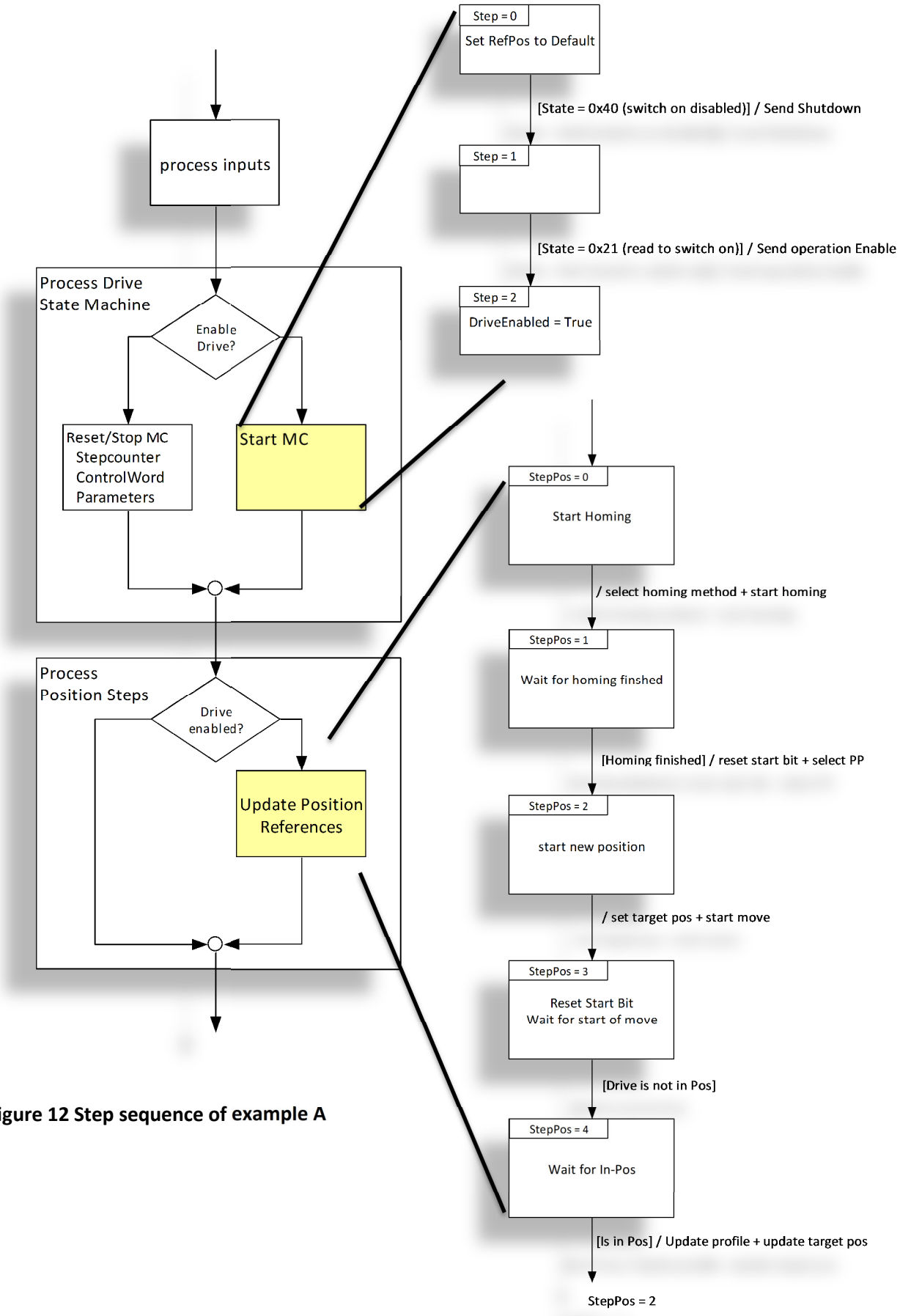


Figure 12 Step sequence of example A

```
'-------------------------------------------
'Author      : FAULHABER MCSUPPORT
'Date        :  2017-09-01
'-------------------------------------------
'Description :  Do a reference and then step positions
'-------------------------------------------


:Init

'force the settings of Dig-IOs
'config lower limit switch
'this is bit coded - here input 3
SETOBJ $2310.01 = 4
'config upper limit switch to none
SETOBJ $2310.02 = 0
'Polarity = strait
SETOBJ $2310.$10 = 0
'Threshold = TTL
SETOBJ $2310.$11 = 0

'inital target Pos = 10000
p = 10000
'initial acc/dec = 10
'if i = 1 we do increment this by 10 each step
i = 1
'initial acc/dec rate
r = 10
'initial change rate
q = 10
'write the initial values
SETOBJ $6083.00 = r
SETOBJ $6084.00 = r

'start with a homing
SETOBJ $6060.00 = 6
```

```
'use y as a step variable for the power stage EN/DI
' y = 0 - waiting for disable
' y = 1 - disabled
' y = 2 - waiting for enable
' y = 3 - enabled

'reset the drive state machine
SETOBJ $6040.00 = 0
'now state is waiting for disabled
y=0

'use z as a step variable for the behavior
'z = 0: initializing
'z = 1:
'first step = 0
z=0

:MainLoop
'read DI and statusword
a = GETOBJ $2311.01
s = GETOBJ $6041.00

'preprocess bits of stats word
'setpoint ack
m = (s & $1000) > 0
n = (s & $0400) > 0

'process the interaction with the drive state machine
IF(y = 0) THEN
   'state is waiting for disabled
   'check for beeing disabled
   IF((s & $6F) = $40) THEN
      'now we are disabled
      y = 1
   ENDIF
ELSEIF(y = 1) THEN
   'state is disabled
```

```
   'check for a change to enabled
   IF(a & $01) THEN
      'send the enable sequence. No need to wait for the
      'state changes
      SETOBJ $6040.00 = $06
      SETOBJ $6040.00 = $0F
      'change to waiting for enabled state
      y = 2
   ENDIF
ELSEIF(y = 2) THEN
   'state is waiting for enabled
   'check for status change
   IF((s & $6F) = $27) THEN
      'change to enabled
      y = 3
   ENDIF
ELSEIF(y = 3) THEN
   'state is enabled
   'check for disabled request
   IF(a & $02) THEN
      'send disable command
      SETOBJ $6040.00 = 0
      'switch to wait for disable
      y = 0
   ENDIF
ENDIF

'main processing only if enabled
IF(y = 3) THEN
   'if no successful homing - start it
   IF(z = 0) THEN
      'start homing
      'homing reference is the neg limit switch at DiIn3
      SETOBJ $6098.00 = 17
      SETOBJ $6040.00 = $1F
      'switch to wait for homing finished
      z = 1
```

```
ELSEIF(z = 1) THEN
   'state is wait for hmong finished
   IF m THEN
      'reset homing start
      SETOBJ $6040.00 = $0F
      'switch to PP
      SETOBJ $6060.00 = 1
      z = 2
   ENDIF
ELSEIF(z = 2) THEN
   'state is: set new Ref
   'write Pos ref
   SETOBJ $607A.00 = p
   'start move
   SETOBJ $6040.00 = $1F
   'switch to wait for move started
   z = 3
ELSEIF(z = 3) THEN
   'wait for move started
   SETOBJ $6040.00 = $0F
   IF(n = 0) THEN
     'switch to wait for be in Pos
      z = 4
   ENDIF
ELSEIF(z = 4) THEN
   IF n THEN
      'invert pos ref
      p = (-1)*p
      'modify acc/dec
      r = r + q
      'change of rate slope?
      IF(r = 500) OR (r = 10) THEN
         q = (-1)*q
      ENDIF
      'now write the new rate
      SETOBJ $6083.00 = r
      SETOBJ $6084.00 = r
```

```
            'switch to set new ref state
            z = 2
        ENDIF
    ENDIF
ENDIF

GOTO MainLoop
```

Example B (create a motion profile)

The purpose of this example is to enable the power stage and select PP mode. Then a motion profile is created by

Step 1:

- set an absolute target position A of 0 and
- set the target window time to 500ms
- start the move

Step 2: (if position A reached)

- set an absolute target position B of 40,000 increments
- set target window time to 20ms
- start the move

Step 3 (if position B reached)

- set an absolute target position B of 45,000 increments
- set target window time to 100ms
- start the move

Step 4:

- restart with step 1

In this example no profile parameters are changed between the different moves. But of course this could have been added easily.

Figure 13 Cyclic motion profile of example B

Again a step sequence is the best suited pattern as we have to wait for the drive signaling a target reached before we start the next step.

```
'--------------------------------------------
'Author      :  FAULHABER MCSUPPORT
'Date        :  2017-09-01
'--------------------------------------------
'Description :  create am movement profile
'--------------------------------------------


:Init

'OpMode = PP
SETOBJ $6060.00 = 1
'no limit switches
SETOBJ $2310.01 = 0
SETOBJ $2310.02 = 0

'reset state machine
```

```
SETOBJ $6040.00 = 0
y = 0
'process state machine by step sequence
'y = 0: waiting for disabled
'y = 1: waiting for enabled
'y = 2: enabled

:MainLoop

s = GETOBJ $6041.00

IF(y = 0) THEN
   IF((s & $7F) = $40) THEN
      'this is disabled
      'switch to enable
      SETOBJ $6040.00 = $06
      SETOBJ $6040.00 = $0F
      'next state is wait for enabled
      y = 1
   ENDIF
ELSEIF(y = 1) THEN
   'this is wait for enabled state
   IF((s & $7F) = $27) THEN
      'switch to enabled state
      y = 2
      'reset step sequence for movement
      z = 0
   ENDIF
ENDIF

'movement:
'move to abs 0 - after 500ms
'move to abs 40000 - after 20ms
'move to abs 45000 - after 100ms

IF(y = 2) THEN
   IF(z = 0) THEN
```

```
    'set Pos REf
    SETOBJ $607A.00 = 0
    'Set Target Window Time
    SETOBJ $6068.00 = 500
    'start move
    SETOBJ $6040.00 = $1F
    z = 1
ELSEIF(z = 1) THEN
    'wait for move started
    IF((s & $0400) = 0) THEN
        z = 2
        SETOBJ $6040.00 = $0F
    ENDIF
ELSEIF(z = 2) THEN
    'wait for being in pos
    IF(s & $0400) THEN
        'start next move
        SETOBJ $607A.00 = 40000
        SETOBJ $6068.00 = 20
        SETOBJ $6040.00 = $1F
        z = 3
    ENDIF
ELSEIF(z = 3) THEN
    'wait for move started
    IF((s & $0400) = 0) THEN
        z = 4
        SETOBJ $6040.00 = $0F
    ENDIF
ELSEIF(z = 4) THEN
    'wait for being in pos
    IF(s & $0400) THEN
        'start next move
        SETOBJ $607A.00 = 45000
        SETOBJ $6068.00 = 100
        SETOBJ $6040.00 = $1F
        z = 5
    ENDIF
```

```
   ELSEIF(z = 5) THEN
      'wait for move started
      IF((s & $0400) = 0) THEN
         z = 6
         SETOBJ $6040.00 = $0F
      ENDIF
   ELSEIF(z = 6) THEN
      'wait for being in pos
      IF(s & $0400) THEN
         'start over
         z = 0
      ENDIF
   ENDIF
ENDIF

GOTO MainLoop
```

Additional Patterns

After the first successful steps seen in the examples, the expectations might increase. Some additional options and techniques might help.

Asynchronous reaction to events

In a main-loop structure according to Figure 8 it's easy to cyclically check whatever condition the drive might be in. Additionally it's possible to register one of the subroutines as a handler for any condition signaled in the device status word 0x2324.01. To check the collection of information in this 32 bit word, open the FAULHABER MotionManager Status Display or refer to the manual.

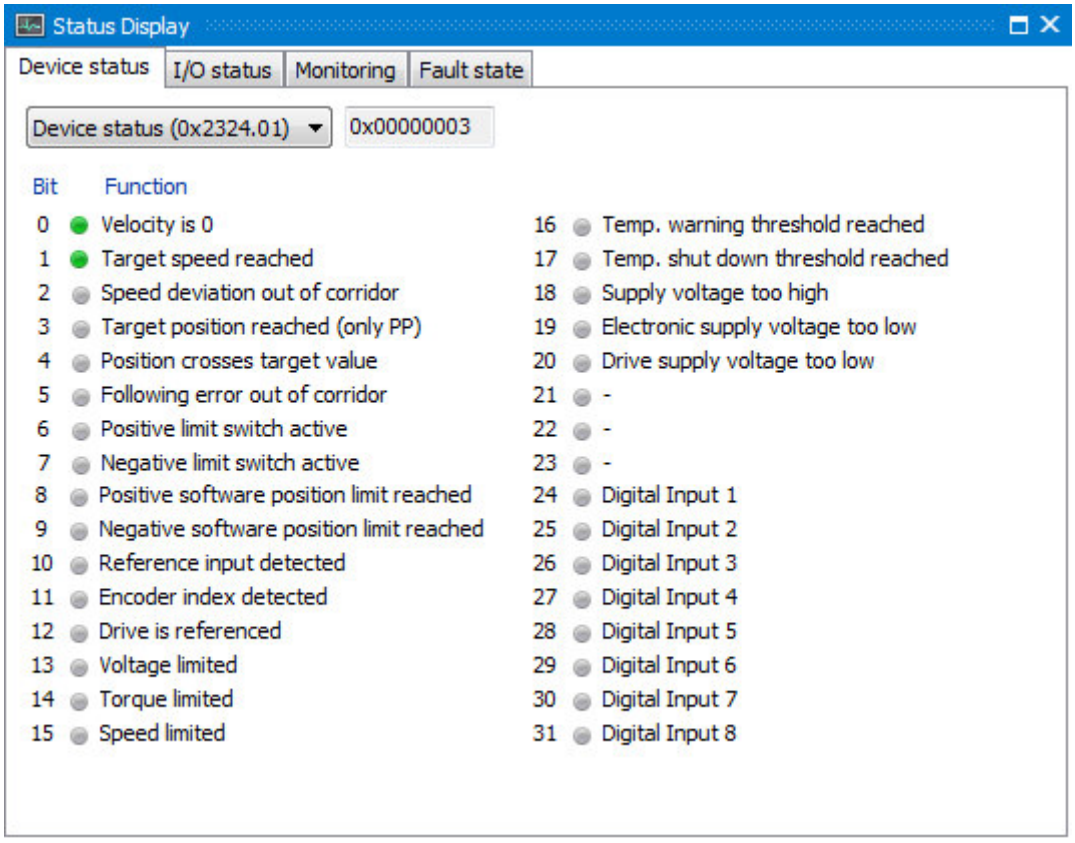


Figure 14 bits collected in the device status word 0x2324.01

A single handler for any combination of bits in this device status word can be registered:

There are four extended commands to support this:

Key word	Description
EN_EVT	Registers a subroutine identified by a label to be the handler for a certain combination of bits within the device status word. `EN_EVT $00010000, OverTemp` Will register the subroutine at the label OverTemp to be executed if the Temp warning bit is set.
DI_EVT	Disabled the event from further calls
DEF_EVT_VAR	Defines a variable to be used in the event processing. During each call, the contents of the device status word will be copied into the event variable `DEF_EVT_VAR e` Will define variable e to be the variable used by the event processing.
RET_EVT	Returns out of the event-handler back to the next line of the main program

Code example

```
:Init
…
DEF_EVT_VAR s
EN_EVT $00010000, OverTemp

:MainLoop
…
…
…
GOTO MainLoop

:OverTemp
'do whatever seems appropriate
RET_EVT
```

Add time-outs and time measurement

Time-outs

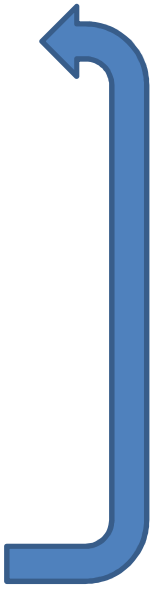
MC based scripts are executed without any specific timing behavior, simply as fast as possible. Sometimes however, a defined timing might be required e.g. in the case of a time-out.

A main-loop + step-sequences type of scripting allows for different conditions to be checked in each execution. So it would be easy to check a timeout and simultaneously check for being in position. To implement time-outs a BASIC extension can be used. Key commands are:

Key word	Description
DEF_TIM_VAR	Defines one of the variable to be used by the timer `DEF_TIM_VAR t` Will define the variable t to be used by the timer
START_TIM	Sets the defined timer variable to the given value and starts the timer. The timer will count down until reaching 0. The unit is 1ms. The value might be given by a fixed coded number or by a second variable. The defined variable and thus the elapsed time can then be evaluated. `START_TIM 10000` Will start a timer running for 10s

Example code

```
:Init
z = 0
DEF_TIM_VAR t

:MainLoop

IF(z = 0) THEN
   'start the time measurement
   START_TIM 10000
   z = 1
ELSEIF (z = 1) THEN
      'timer elapsed?
   IF (t = 0) THEN GOSUB xxxx
ENDIF
…
GOTO MainLoop
```



So different from the micro-loop structure of the flow-chart in Figure 7 it is here possible not only to check the elapsed time but also do additional checks in each step.

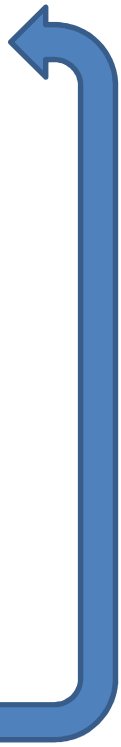
Time-measurement

The opposite approach might be the measurement of an elapsed time. This could be a transient time or any reaction time out of a supervised process. Again a timer is used, but in this case the timer starts at a value of 0 and counts the elapsed milliseconds.

Key word	Description
DEF_CYC_VAR	Defines one of the variables to be used by the timer `DEF_CYC_VAR t` Will define the variable t to be used by the timer
START_CYC	Sets the defined timer variable to 0 and start the timer. The timer variable is incremented in the background and can be evaluated by reading the defined timer variable. The unit is 1ms. `START_CYC`
STOP_CYC	Stops the timer `STOP_CYC`

Example code

```
:Init
z = 0
DEF_CYC_VAR t

:MainLoop

IF(z = 0) THEN
   'start the time measurement
   START_CYC
   Z = 1
ELSEIF (z = 1) THEN
      'timer elapsed?
      '3 different actions within 1s
   IF (t > 1000) THEN GOSUB Step1
   ESLEIF (t > 650) THEN GOSUB Step2
   ESLEIF (t > 350) THEN GOSUB Step3
   ENDIF
ENDIF
…
GOTO MainLoop
```



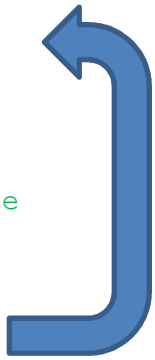
Timer variables can be evaluated within a script. Logging them via the built in logging feature of the MotionController is not supported though. If you want to visualize the down- or up-counting of a timer you need to cyclically write the actual value of the selected timer variable to a second variable – which can then be logged.

Measuring the cycle time

What about the timing performance of an application. There is no built in feature to assess the cycle time of an application as there is no predefined behavior – executing loops is a recommendation only.

Measuring the cycle time of a cyclic application can easily implemented without any special services though. Simply use a free variable and initialize it to 1. Then in each cycle multiply the variable by -1 and add log the contents. As the execution time of many scripts will only be a few ms it might even be necessary to use the recorder and trigger the variable crossing 0.

```
:Init
'here k is used as the trigger variable for loop time
k = 1

:MainLoop
    ' do whatever

   'invert k for logging purpose
   k = -1 * k

GOTO MainLoop
```



Mixed operation of Master and local scripts

Even a combined operation of a master controller and a set of local scripts is possible. The key here is the shared access to the 26 global variables. All of them are available in object 0x3005.

So let's assume a drive shall be used for position control by a master but shall execute a homing at the beginning. Homing method is using a block as a reference condition in the beginning. In such a case it's likely and recommended to use reduced current -/torque-limits during the initial homing and restore the original limits afterwards.

This can be done by a master directly but as the master now needs to read and write the torque limit, they have to be added to the process image or a dedicated SDO read/write access has to be implemented.

Alternatively you could create a local script executing the homing which restores the limits after the homing. Of course even with this approach some exchange variable will be necessary but the overall solution might be simpler.

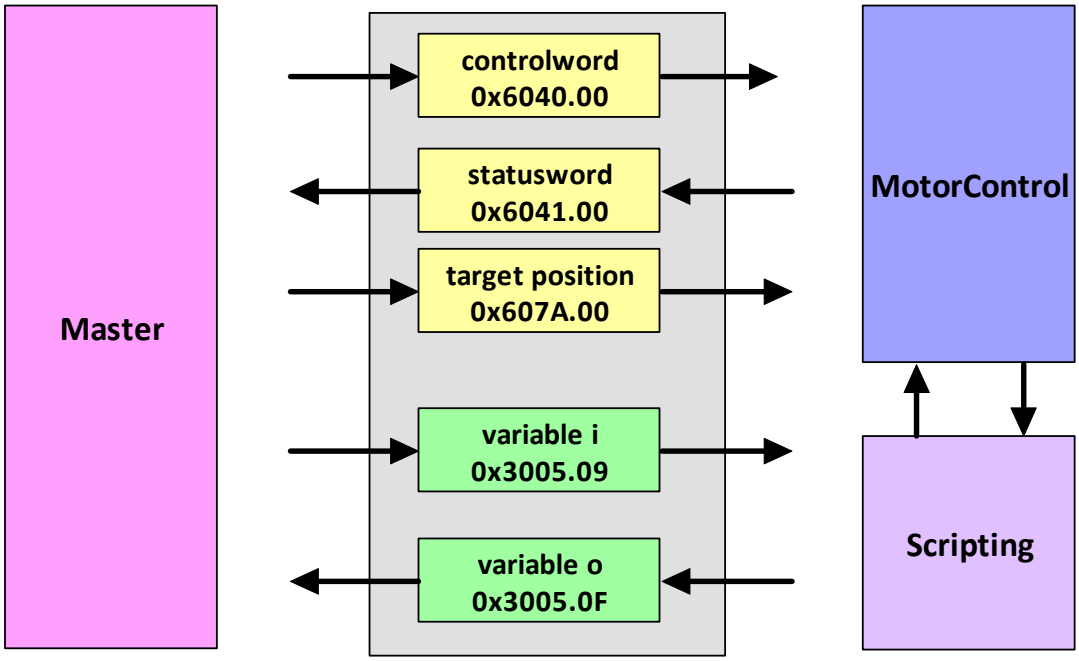


Figure 15 combination of a master PLC and local scripts

We could now create a local sub-routine executing a step-sequence to:

- save the original torque limits to variables
- configure the homing
  - select homing method
  - configure reduced torque limits
- start the homing and wait for homing finished
- restore the original torque limits and switch to PP or CSP mode
- signal the drive being ready by writing to the variable o.

So let's assume the variable z is once again used as the internal step variable and the processing starts if z = 1. The variable i is used by the master to start a local script. The main-loop of such a solution might look like:

```
:Init
...
:MainLoop
'read status word
s = GETOBJ $6041.00
'eval the input variable
IF((i = 1) AND (z = 0)) THEN
   z = 1
ENDIF

'Standard step-sequence
IF (z = 1) THEN GOSUB Z1
ELSEIF(z = 2) THEN GOSUB Z2
ELSEIF(z = 3) THEN GOSUB Z3
ENDIF

GOTO MainLoop

:Z1
'signal the sub-routine to be started
o = 1
'save positive and negative torque limit
p = GETOBJ $60E0.00
n = GETOBJ $60E1.00
'reduce torque limit for now
SETOBJ $60E0.00 = 500
SETOBJ $60E1.00 = 500
 'select homing method, OpMode homing and start the homing
SETOBJ $6098.00 = -1
SETOBJ $6060.00 = 6
SETOBJ $6040.00 = $1F
'switch to waiting step
z = 2
RETURN
```

Requires the power-stage has already been started and the control word has been 0x000F after the last write. Otherwise no rising edge in bit 4

```
:Z2
'wait for being started
IF ((s & $1000) = 0) THEN
   z = 3
ENDIF

:Z3
'wait for homing finished
IF (s & $1000) THEN
'reset the OpMode and parameters
   SETOBJ $6060.00 = 1
   SETOBJ $60E0.00 = p
   SETOBJ $60E1.00 = n
   'signal to be finshed
   o = 2
   'stop the action
   z = 0
ENDIF
RETURN
```

In this case the contents of the output variable o is used as a feedback to the PLC with the values

`o = 0:` idle – no sequence is active

`o = 1:` sequence is started

`o = 2:` sequence is finished

Several subroutines could be stored in a single BASIC script and the contents of i could be used to select the one to be started. In this case the master would have to add a step to its sequence where the homing is started by writing a 1 to the exchange variable i and then waiting for the exchange variable o to become 2. Looks a little complicated but saves a lot of interaction between the PLC and the MotionController.

Additional Resources

FAULHABER Application Notes	
App-Note 164	control a FAULHABER MC V3.0 ET out of a CODESYS environment

FAULHABER manuals at www. faulhaber.com/manuals

FAULHABER demo systems at youtube. Some of them using local scripting to control the behavior.

Rechtliche Hinweise

Urheberrechte. Alle Rechte vorbehalten. Ohne vorherige ausdrückliche schriftliche Genehmigung der Dr. Fritz Faulhaber & Co. KG darf insbesondere kein Teil dieser Application Note vervielfältigt, reproduziert, in einem Informationssystem gespeichert oder be- oder verarbeitet werden.

Gewerbliche Schutzrechte. Mit der Veröffentlichung der Application Note werden weder ausdrücklich noch konkludent Rechte an gewerblichen Schutzrechten, die mittelbar oder unmittelbar den beschriebenen Anwendungen und Funktionen der Application Note zugrunde liegen, übertragen noch Nutzungsrechte daran eingeräumt.

Kein Vertragsbestandteil; Unverbindlichkeit der Application Note. Die Application Note ist nicht Vertragsbestandteil von Verträgen, die die Dr. Fritz Faulhaber GmbH & Co. KG abschließt, soweit sich aus solchen Verträgen nicht etwas anderes ergibt. Die Application Note beschreibt unverbindlich ein mögliches Anwendungsbeispiel. Die Dr. Fritz Faulhaber GmbH & Co. KG übernimmt insbesondere keine Garantie dafür und steht insbesondere nicht dafür ein, dass die in der Application Note illustrierten Abläufe und Funktionen stets wie beschrieben aus- und durchgeführt werden können und dass die in der Application Note beschriebenen Abläufe und Funktionen in anderen Zusammenhängen und Umgebungen ohne zusätzliche Tests oder Modifikationen mit demselben Ergebnis umgesetzt werden können.

Keine Haftung. Die Dr. Fritz Faulhaber GmbH & Co. KG weist darauf hin, dass aufgrund der Unverbindlichkeit der Application Note keine Haftung für Schäden übernommen wird, die auf die Application Note zurückgehen.

Änderungen der Application Note. Änderungen der Application Note sind vorbehalten. Die jeweils aktuelle Version dieser Application Note erhalten Sie von Dr. Fritz Faulhaber GmbH & Co. KG unter der Telefonnummer +49 7031 638 688 oder per Mail von mcsupport@faulhaber.de.

Legal notices

Copyrights. All rights reserved. No part of this Application Note may be copied, reproduced, saved in an information system, altered or processed in any way without the express prior written consent of Dr. Fritz Faulhaber & Co. KG.

Industrial property rights. In publishing the Application Note Dr. Fritz Faulhaber & Co. KG does not expressly or implicitly grant any rights in industrial property rights on which the applications and functions of the Application Note described are directly or indirectly based nor does it transfer rights of use in such industrial property rights.

No part of contract; non-binding character of the Application Note. Unless otherwise stated the Application Note is not a constituent part of contracts concluded by Dr. Fritz Faulhaber & Co. KG. The Application Note is a non-binding description of a possible application. In particular Dr. Fritz Faulhaber & Co. KG does not guarantee and makes no representation that the processes and functions illustrated in the Application Note can always be executed and implemented as described and that they can be used in other contexts and environments with the same result without additional tests or modifications.

No liability. Owing to the non-binding character of the Application Note Dr. Fritz Faulhaber & Co. KG will not accept any liability for losses arising in connection with it.

Amendments to the Application Note. Dr. Fritz Faulhaber & Co. KG reserves the right to amend Application Notes. The current version of this Application Note may be obtained from Dr. Fritz Faulhaber & Co. KG by calling +49 7031 638 688 or sending an e-mail to mcsupport@faulhaber.de.