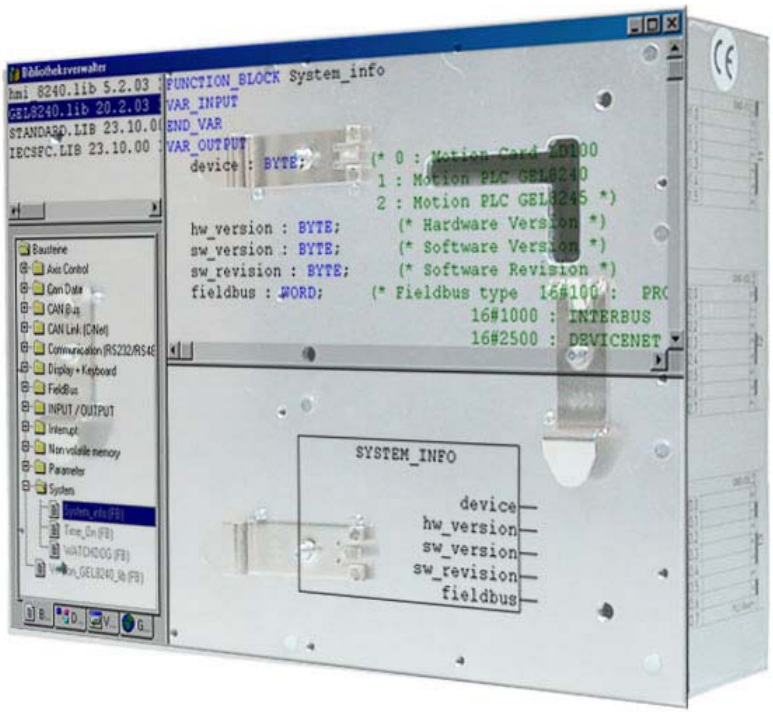
MOTIONLINE

LENORD+BAUER

MotionPLC

GEL 8240 / 8241 / 8245 / 8246

Function Library GEL8240.lib



Reference Manual
Edition: 2004-07

Editions published to date:

Edition	Remarks
2004-07	First edition Valid for firmware version ≥1.30, library version 1.08

Published by:

MOTIONLINE
LENORD+BAUER

Lenord, Bauer & Co. GmbH
Dohlenstrasse 32
46145 Oberhausen • Germany
Fon: +49 208 99 63-0 • Fax +49 208 67 62 92
Internet: http://www.lenord.de • E-Mail: info@lenord.de

Mat. No. DS12-8240/8245R

Contents

1 General

The following description deals with the GEL8240.lib library for creating PLC programs in acc. with IEC 61131-3 using the **CoDeSys** programming environment in the PC. The other libraries supplied are not dealt with here; for these, please refer to the comments enlisted with the individual function blocks.

The present description is intended to supplement the CoDeSys Manual.

All manuals are supplied in electronic form as PDF files on CD with the MotionPLC. The *Acrobat Reader* from ADOBE SYSTEMS INCORPORATED required for reading these files may be installed from the CD.

The following **knowledge** is assumed:

- Familiarity with the MS-WINDOWS operating systems
- Operating the CoDeSys programming environment
- IEC 61131-3 programming
- Operating and functionality of the GEL 824x MotionPLC

Symbols and designations used in this manual:

ℹ	identifies paragraphs providing important additional information about the subject
!	identifies paragraphs containing important statements required for proper operation.
para[123]	identifies a programmable system parameter as it is described in the MotionPLC Operating Instructions.
FB	is the abbreviation of function block (several: FBs)

ℹ **CAN axes**:

From the operating system version 1.30 four additional CAN axes may be controlled instead of the max. 4 CAN axes possible so far, using the node numbers 5...8. Thus, the total number of slave axes has increased from 7 to 11 (3x analog + 8x CAN).

For this purpose, you have to parallel both CAN interfaces on terminal block C2 thus building one single CAN bus. In this case, the following conditions are valid:

- Depending on the axes used, the number of possible CAN I/O modules (standard CAN objects) is reduced to
  4 - (number of CAN axes - 4),
  for example, 2 modules with 6 CAN axes (3rd und 4th CAN object).
- The node addresses of the CAN I/O modules must be higher than those of the CAN axes.
- The MotionPLC must be configured as CAN master (para[50]=0).
- The main shaft functionality must not be activated (para[125]=0).

- The CAN bus transmission rate will automatically be increased from 500 kBaud to 1 MBaud for the following cycle time/number of axes combinations:
  – 2 ms / 4 CAN axes (separate CAN buses; as previously)
  – 3 ms / 5 or 6 CAN axes
  – 4 ms / 5...8 CAN axes
- The transmission rate for the serial communication (BB2100K, CoDeSys) must not be set higher than 38400 if more than 4 CAN axes are used on the CAN bus.

For the additional axes the same functionality and CoDeSys function blocks are valid as for the four CAN axes used so far.

2 Utilization

The library provides predefined FBs for various applications (the table of contents provides an overview):

Utilizing FBs requires the necessary library file to be included when a new program is built (CoDeSys: *Window/Library Manager* menu, right mouse button click on *Additional Library...*).

After building the program and its successful compilation in CoDeSys, the program is transferred with the *Online/Login* command to the MotionPLC via the serial interface where it can be executed or tested (the *Online/Simulation Mode* menu item must not be active for this purpose).

The program is initially only located in the module's RAM and should be transferred after successful testing to the non-volatile flash memory (CoDeSys: *Online/Create Boot Project* menu item), from where it is reloaded into the RAM and executed after next switching on.

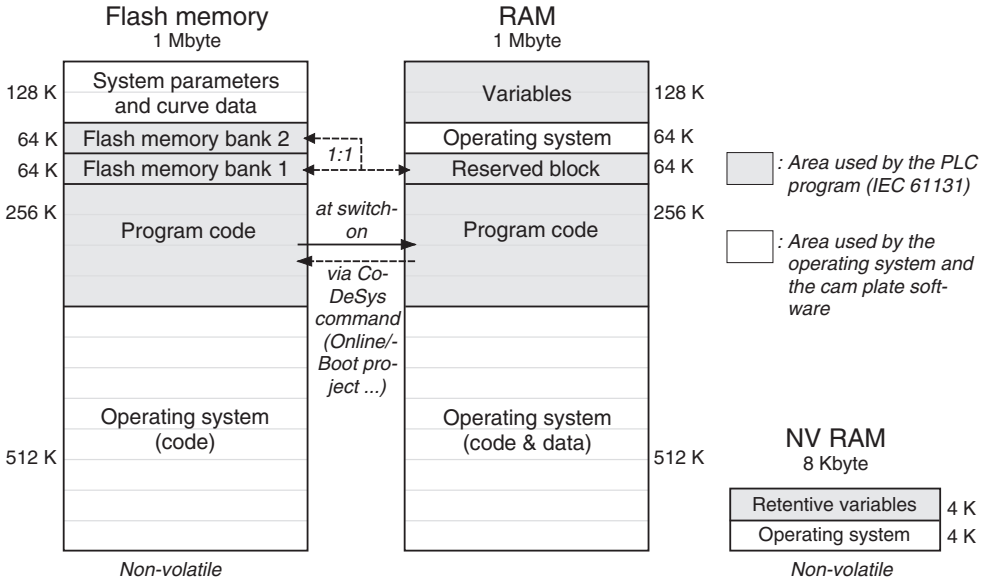
During data transfer to the flash memory the execution of the PLC program is suspended.

To start the program from CoDeSys the PLC must be activated, i.e., input I3.7 (PLC RUN) must be High (Low = stop).

When a program is stopped in CoDeSys, it can only be restarted from this environment unless the MotionPLC is switched off and on again.

3 Memory organization

Programming in acc. with IEC 61131 makes use of certain memory areas in the RAM, flash, and non-volatile (NV) RAM memories. Compare the following overview:



3.1 Flash memory

Two 64K memory banks are available for saving a certain RAM area (see below); the corresponding FBs are described in section 4.11 (from p. 88).

Data can only be written to a flash bank as an entire block, i.e. a 64K data block is required. The existing data are first removed with a delete operation.

Flash memory write operations are time-consuming requiring several seconds and are therefore critically susceptible to power failures (a reset during a delete or transfer operation results in a loss of all data to be stored). Safety precautions should therefore be taken on program level in order to be able to react to possible data loss (see example below).

The flash memory chips used have only a limited lifetime of at least 10,000 write operations. Data should therefore not be saved to the flash memory too often.

During data transfer to the flash memory the execution of the PLC program is suspended.

Possible utilization of the two flash memory banks:

- Large amounts of data can be stored safe against power failures (up to 128 Kbytes). For this purpose, it is very important to create an algorithm which permits detection of power failures having occurred while data were being stored (in analogy to the example below).
- If 64 Kbytes are sufficient for data storage, the second bank can be used for a backup copy of the first one (in the event of errors during overwriting of the

first bank – leading to total data loss – the other bank still contains the previously valid data).

The procedure described below explains roughly and **by way of example** how to react in a program to the possibility of power failures while writing data to the flash memory. The program is to change various data that were stored before in flash memory bank 1 and are then to be re-saved into flash memory bank 2 (after the next change, bank 1 is then to be used again and so on, always in alternating sequence).

0. Two flags must be reserved as retentive BYTE variables ('RETAIN'; storage in NV RAM, see below) – one for the step just performed (*step*) and the other for the number of the flash memory bank just used (*flash_no*)
1. Initialize step flag: *step* = 0 (*flash_no* is at 1)
2. Copy flash memory bank (1) into the RAM (Restore_Memory FB)
3. Increment step flag: *step* = 1
4. Copy data area from reserved RAM area into the working memory (→ variable to be changed; Memory_to_VAR FB)
5. Execute desired changes in the working memory
6. Copy variable(s) back to the reserved RAM area (Var_to_Memory FB)
7. Increment step flag: *step* = 2
8. Save reserved RAM area to the other flash memory bank (2) (Save_Memory FB). This ends the data change operation.
9. Reset the step flag and set flash flag to the current bank number: *step* = 0, *flash_no* = 2 (the current values are now in flash memory bank 2)

   Immediately after powering on, there must be a query in the program as to which of the flash memory bank contains the data last saved and whether step counter *step* is at 0. Only then has the last save operation been terminated correctly. In all other cases, *step* informs about the last successful step before switch-off.

3.2 RAM

The "normal" working memory contains a reserved 64k block that can be used as a buffer for saving any kind of data from the variables area (individual variables, arrays etc.). This RAM block can then be transferred to one of the two non-volatile flash memory banks or read back from there. Access to this area is possible only with special FBs of the *Non volatile memory* category (see section 4.11, from p. 88).

Another RAM area contains the memory locations for the system parameters organized as an array of 1000 Longs (DINT) of which only the first 500 (0…499) are specially reserved for the intrinsic system parameters. The remaining 500 Longs (500…999) can thus also be used for IEC 61131 PLC programming.

These memory locations offer the following advantages:

- With an operating system function (cam plate software) or an IEC 61131 function (*Save_Parameter*), the values can be transferred safe against power failures to the flash memory. In this way, PLC data are also saved together with the system parameters. The data can also be stored in a file on the hard disk (via the attached "LingiMon" tool, see the Operating Instructions).
- The "pseudo" system parameters 500…999 – like the normal system parameters – can be addressed individually and directly via the serial interface[1], i.e. they can be read and written without the need for a separate protocol to be created especially for this purpose.

3.3 NV RAM

This memory area is used for automatic saving when the mains voltage drops below nominal (power failure saving). The area is used for

– storing the variables defined in the CoDeSys environment as retentive by means of the RETAIN supplement and

– cyclically storing the current curve data (positions etc.) by the operating system.

The various data structures are transmitted to the NV RAM in a **byte-serial** format. If, for instance, a retentive DINT variable is to be changed, its 4 bytes are transmitted one after another (in the case of data arrays, a correspondingly larger number of bytes is involved). If a power failure occurs during the transmission of such data there is the risk that a variable has not yet been completely overwritten with new bytes. This would result in incorrect values when the device is switched on again. It is therefore strongly recommended that the program contains preventive measures to cope with such events (as shown in the example for the flash memory, see above).

! When downloading a program (CoDeSys → MotionPLC) this memory area is not initialized, i.e., all variables retain their values last defined.

[1] For instance, in the terminal mode of BB2100K or in the PLC browser of CoDeSys. You may also use the BB2100K commands in the PLC browser, but they must be preceded by the "@" character (BB2100K: rr 502 ⇒ PLC browser: @rr 502). Additionally, you may use the "n_dl_ini" and "dl_ini" commands for deactivating / activating the initialization of the variables and the disabling of the outputs when downloading a modified program (in the deactivated state you should not add or remove any variables!).

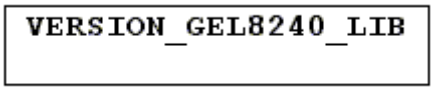
4 Function blocks

The FBs described below have been dealt with in the order as shown by the tree structure in the library window; but differing from this, existing subgroups are dealt with after the FBs in the corresponding branch.

Version_GEL8240_lib

This "pseudo" function block supplies in its declarative part only information about the current version of the library, including its history.

- Function block:

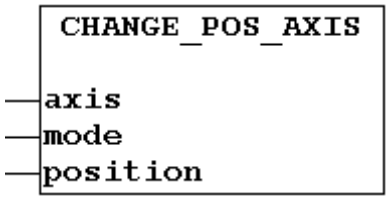


4.1 Axis Control

Change_Pos_Axis *Axis Control*

Changes the actual position of the x axis (master) or a y axis (slave)

- Function block:



- Variables:

axis WORD;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

mode WORD;
Kind of changing for all axes:
0 = absolute
1 = relative

Kind of changing for CAN axes only (8 at max.):
2 = absolute with next zero-crossing of the feed back unit, independent of the direction of movement
3 = absolute with next zero-crossing in forward direction
4 = absolute with next zero-crossing in backward direction

position DINT;
for *mode* = 0|2|3|4: new actual value
for *mode* = 1: amount of the actual value changing

- Example:

  Declaration:

```
set_nom_val: Change_Pos_Axis;
```

  Program in ST:

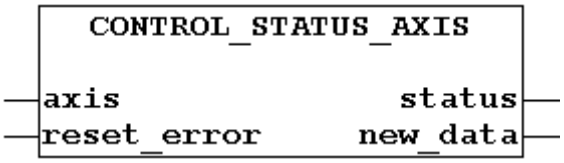
```
set_nom_val(axis:=3, mode:=1, position:=-1500);
set_nom_val(axis:=2, mode:=0, position:=500);
```

When calling the first FB the current actual value of slave 3 is reduced by 1500. The second call sets the actual value of slave 2 to 500.

Control_Status_Axis *Axis Control*

Provides information about a servo amplifier controlled via CAN bus

- Function block:



- Variables:

axis BYTE;
CAN axis only:
1 … 11 = y axis 1 ... 11 (slave)
You can activate up to 11 axes. Primarily the analog axes are numbered consecutively (up to 3), followed by the CAN axes (up to 8).

reset_error BOOL;
TRUE = faults in the servo amplifier are to be reset

status DWORD;

Bit no.	Description
0	Logic state on digital input 1 (X3.11)
1	Logic state on digital input 2 (X3.12)
2	Logic state on digital input 3 (X3.13)
3	Logic state on digital input 4 (X3.14)
26	Initializing terminated
28	Motor standstill
29	Safety relay
30	Power stage enabled
31	Fault occurred

new_data BOOL;
TRUE = communication with servo amplifier is active

◆ Example:

Declaration:

```
read_status: Control_Status_Axis;
```

Program in ST:

```
read_status(axis:=4, reset_error:=FALSE);
IF read_status.status AND 16#80000000 = 16#80000000 THEN
   read_status(reset_error:= TRUE);
END_IF;
```

The status of slave 4 is read. In the case of an error message (Bit 31 = TRUE) the fault will be reset.

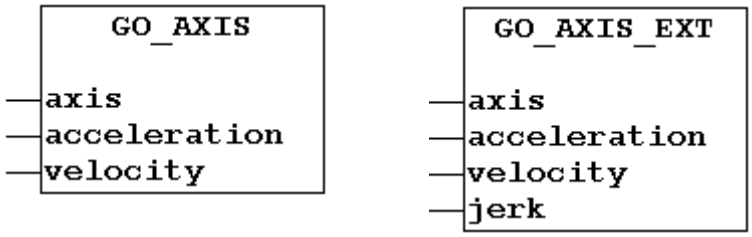
Go_Axis / Go_Axis_Ext *Axis Control*

Starts running of an axis using the specified parameters as long as no FB with different parameters or another positioning FB is called (as listed under "See also")

Virtual movement of the master:
If a curve has been selected (with active power failure security the curve number which has been eventually specified some time ago will be stored retentively) the master counting range is limited to the x range defined for the curve.

See also: Pos_Axis / Pos_Axis_Ext, Start_Cam_Axis, Stop_Axis / Stop_Axis_Ext

◆ Function block:



◆ Variables:

axis	BYTE; 0 = x axis (master: simulation, virtual) 1 … 11 = y axis 1 … 11 (slave)
acceleration	DINT; Acceleration of the axis in increments/s^2 Negative values and 0 are internally converted to the smallest possible acceleration value.
velocity	DINT; Speed in increments/s with direction of travel (sign)

jerk DINT;
Jerk limitation of the axis in increments/s^3
Negative values and 0 result in positioning without jerk limitation (as for the Go_Axis FB).

- Example:

Declaration:

```
move_1: Go_Axis;
move_2: Go_Axis;
start_new: BYTE;
start_old: BYTE;
```

Initializing in ST:

```
move_1.axis:=1;
move_2.axis:=2;
move_1.acceleration:=100000;
move_2.acceleration:=50000;
```

Program in ST:

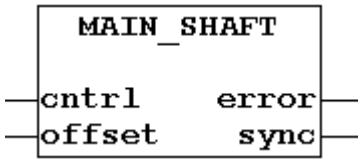
```
IF start_new AND start_old=FALSE THEN
   move_1(velocity:=-15000);
   move_2(acceleration:=120000; velocity:=28000);
END_IF;
start_old:=start_new;
```

The positive edge of `start_new` initiates the run operation for axes 1 and 2. Axis 1 runs in the backward direction using the lastly specified acceleration value which coincides with the one defined in the initialization part – if not changed afterwards.
Axis 2 is running forward with a new acceleration value that will be valid until the `move_2.acceleration` variable is changed anew.

Main_Shaft *Axis Control*

Is to be used with a main shaft slave if the master position of the main shaft is transmitted via CAN bus. The FB also returns the CAN master/slave position deviation and specifies an offset.

- Function block:



- Variables:

cntrl BYTE;
0 = read main shaft data only
1 = read main shaft data and write offset

offset DINT;
Value of the offset related to the main shaft master (≥ 0, negative values will be ignored)
If writing of an offset is enabled (*cntrl* = 1) and the value is not negative then the master reference value (para[113]) in the RAM is overwritten without restriction. The new value will be retained when powering off if the loader function is executed before (copies the parameters from RAM to flash).

error DINT;
For test purposes only (must be nearby 0 if the Sync signal is used, what is recommended)

sync BOOL;
TRUE = communication with main shaft master via CAN bus is running

◆ Example:

Declaration:

```
set_para113: Main_Shaft;
write_para_to_flash: Save_Parameter;
key_new: BOOL;
key_old: BOOL;
```

Program in ST:

```
IF key_new AND key_old=FALSE THEN
   set_para113(cntrl:=1, offset:=350);
   write_para_to_flash();
END_IF;
key_old:=key_new;
```

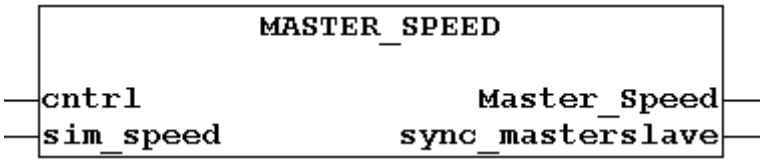
The rising edge of `key_new` sets the system parameter para[113] in the RAM to value 350 and, after that, copies the parameter RAM area to the flash, thus storing the value retentively.

Master_Speed *Axis Control*

Checks for a main shaft slave whether data are transmitted by a main shaft master, returns the actual speed of the cam plate master and optionally transmits a simulation speed if the master is virtual

See also: Go_Axis, Rd_Status_Axis

◆ Function block:



- Variables:

  cntrl BYTE;
  0 = read master speed
  1 = read master speed and write simulation speed

  sim_speed DINT;
  Simulation speed in master increments/s (nominal speed preset for the virtual cam plate master); it is ignored in a main shaft application or if the master position is generated by an incremental encoder

  master_speed DINT;
  Actual master speed in increments/s

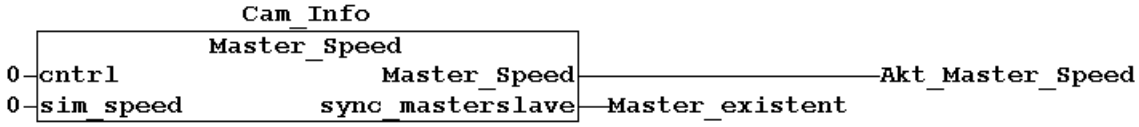
  sync_masterslave BOOL;
  TRUE = the FB is used in a main shaft slave and communication with the master is running

- Example:

  Declaration:

```
Cam_Info: Master_Speed;
Akt_Master_Speed: DINT;
master_existent: BOOL;
```

  Program in FUP:



  The network demonstrates the call of the FB. The returned data are stored in the `Akt_Master_Speed` and `master_existent` variables. A simulation speed is not specified.

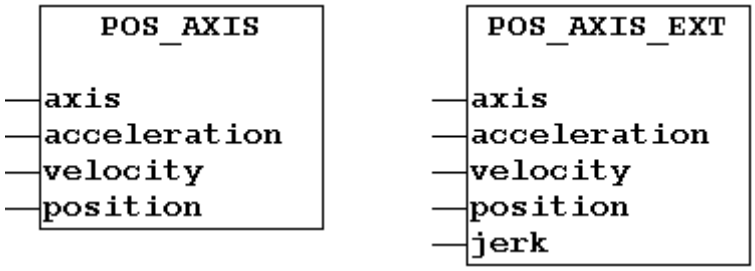
Pos_Axis / Pos_Axis_Ext *Axis Control*

Activates the positioning operation of an axis. It remains active – even when the target position has been reached – until a FB with different drive parameters or another positioning FB is called (as listed under "See also").

Virtual movement of the master:
If a curve has been selected (with active power failure security the curve number which has been eventually specified some time ago will be stored retentively) the master counting range is limited to the x range defined for the curve. However, the position will be reached correctly but the positioning value returned by the Rd_Status FB may differ from the one specified as target position.

See also: Go_Axis / Go_Axis_Ext, Start_Cam_Axis, Stop_Axis / Stop_Axis_Ext

- Function block:



- Variables:

axis BYTE;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

acceleration DINT;
Acceleration of the axis in increments/s^2
Negative values and 0 are internally converted to the smallest possible acceleration value.

velocity DINT;
Speed in increments/s, ≥ 0!

position DINT;
Target position in increments (master: see the info above)

jerk DINT;
Jerk limitation of the axis in increments/s^3
Negative values and 0 result in positioning without jerk limitation (as for the Pos_Axis FB).

- Example:

Declaration:
```
pos_0: Pos_Axis;
start_new: BOOL;
start_old: BOOL;
```

Initialization in ST:
```
pos_0.axis:=0;
```

Program in ST:
```
IF start_new AND start_old=FALSE THEN
   pos_0(acceleration:=10000, velocity:=15000, position:=
                                                    500000);
END_IF;
start_old:=start_new;
```

The one-time call of the FB starts the positioning process of the master axis (virtual axis) with the specified acceleration and speed values towards the target position of 500,000. (If a curve with a x range of 200,000 has been selected the end position will then read 100,000, because the actual position has been zeroed two times after having reached the value of 200,000. If this behaviour is undesirable you have to deactivate the power

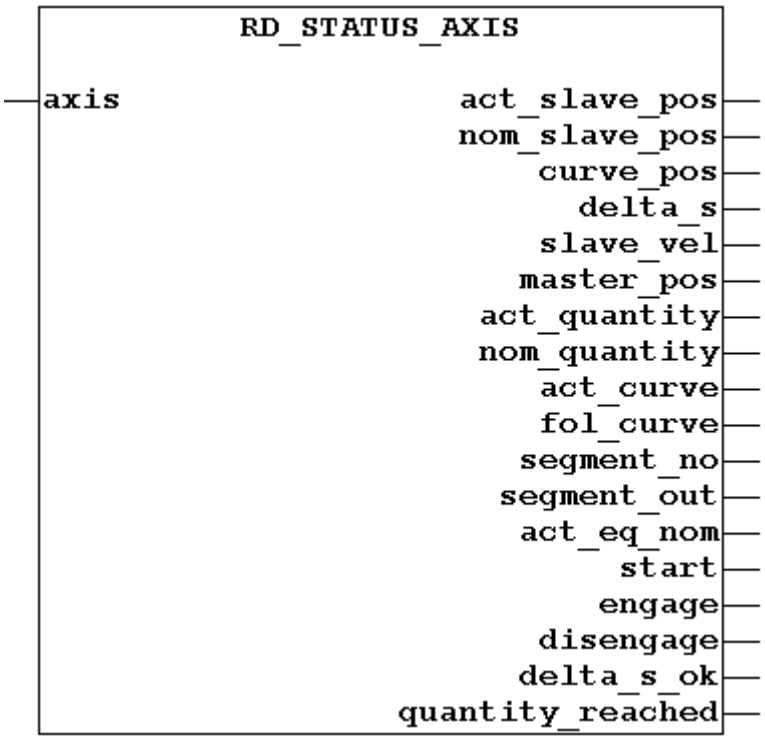
failure security in order to ensure that no curve is selected when switching on.)

Rd_Status_Axis *Axis Control*

Provides status information about the axis to be specified

See also: Go_Axis / Go_Axis_Ext, Pos_Axis / Pos_Axis_Ext, Stop_Axis / Stop_Axis_Ext

- Function block:



- Variables:

axis BYTE;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

act_slave_pos DINT;
for *axis* = 0…11:
Actual position in increments

nom_slave_pos DINT;
for *axis* = 1…11:
Current nominal position in increments

curve_pos DINT;
for *axis* = 1…11:
Current curve position in increments depending on the master position and a valid curve selected for the defined slave axis

If the curve is started and the engaging procedure terminated then this value is equal to the *nom_slave_pos* value.

! This value is only processed if the selected curve exists in the defined axis – even if it is not started – and the axis is controlled without jerk limitation. On the other side, the value is not processed as long as a speed value ≠ 0 with jerk limitation is set (Go_Axis_Ext, Pos_Axis_Ext, Stop_Axis_Ext with *jerk* ≠ *0*).

delta_s DINT;
for *axis* = 1…11:
Following distance between the current nominal position and der actual position:

delta_s = *nom_slave_pos* – *act_slave_pos*

slave_vel DINT;
for *axis* = 0…11:
Current actual speed in increments/s
If the master axis is specified and operated virtually then the simulation speed is issued.

master_pos DINT;
for *axis* = 1…11:
Actual position of the master axis in increments of the master

act_quantity DINT;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

nom_quantity DINT;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

act_curve INT;
for *axis* = 1…11:
Number of the selected curve

fol_curve INT;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

segment_no BYTE;
for *axis* = 1…11:
Number of the curve segment of the defined slave axis, where the master is actually located

! Restriction as described for *curve_pos*.

segment_out BYTE;
for *axis* = 1…11:
Informs about the OUT signals of the curve segment of the defined slave axis, where the master is actually located.

! Restriction as described for *curve_pos*.

act_eq_nom BOOL;
for *axis* = 0 only:
Inverse state of the *engage* output in the defined axis

start BOOL;
for *axis* = 0…11:
TRUE = a valid curve or a positioning operation has been started in the defined axis with Start_Cam_Axis or Pos_Axis / Pos_Axis_Ext)
FALSE = one of the Go_Axis / Go_Axis_Ext or Stop_Axis / Stop_Axis_Ext functions has been initiated in the defined axis

engage BOOL;
for *axis* = 0…11:
TRUE = a positioning operation has been started in the defined axis with Pos_Axis / Pos_Axis_Ext and the current nominal position *nom_slave_pos* (slave or master) has not yet reached the target position
additionally for *axis* = 1…11:
TRUE = a valid curve has been started in the defined axis with Start_Cam_Axis and the current nominal position *nom_slave_pos* has not yet reached the current curve position *curve_pos* (engaging process)

disengage BOOL;
for *axis* = 1…11:
TRUE = in the defined axis, a curve started with Start_Cam_Axis) has been stopped with Stop_Axis / Stop_Axis_Ext and the axis is not yet in standstill (disengaging process)

delta_s_ok BOOL;
for *axis* = 1…11:
TRUE = following error *delta_s* with active feedback position control is inside the admissible window; the maximum admissible following distance is to be specified for each axis in the corresponding system parameters for the analog and CAN axes

quantity_reached BOOL;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

◆ Example:

Declaration:

```
status_axis: Rd_Status_Axis ;
master_act_pos: DINT
slave1_act_pos: DINT
slave2_act_pos: DINT
```

Program in ST:

```
status_axis(axis:=0);
master_act_pos:=status_axis.act_slave_pos;
status_axis(axis:=1);
slave1_act_pos:=status_axis.act_slave_pos;
status_axis(axis:=2);
slave2_act_pos:=status_axis.act_slave_pos;
```

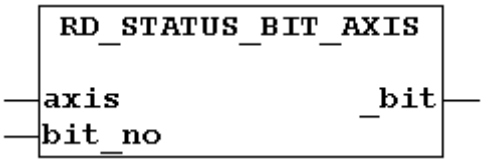
The multiple cyclic call of the same FB successively provides the actual positions of the master axis (0) and the slave axes 1 and 2. Between the calls the particular actual value is assigned different variables.

Rd_Status_Bit_Axis *Axis Control*

Informs about the status of the defined bit in the selected axis

See also: Rd_Status_Axis

◆ Function block:



◆ Variables:

axis BYTE;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

bit_no BYTE;
The bit the status of which shall be queried:

bit_no	Global variable	Valid for axis ...
0	*act_eq_nom*	0
1	*start*	0 … 11
2	*engage*	0 … 11
3	*disengage*	1 … 11
4	*delta_s_ok*	1 … 11
8	*ref_reached*	1 … 11
10	*drive_err*	CAN

bit_no	Global variable	Valid for axis ...
11	*cam_started*	1 … 11
20	*ext_achse*	1 … 11
21	*ext_LD2000*	CAN
22	*ext_LD2000_prg*	CAN
23	*ext_LD2000_init*	CAN
24	*ext_LD2000_online*	CAN
25	*ext_LD2000_operational*	CAN
30	*drive_enable*	1 … 11
35	*drive_forwards*	1 … 11
36	*drive_backwards*	1 … 11

act_eq_nom

Inverse state of the *engage* output in the defined axis

start

TRUE = a valid curve or a positioning operation has been started in the defined axis with Start_Cam_Axis or Pos_Axis / Pos_Axis_Ext)

FALSE = one of the Go_Axis / Go_Axis_Ext or Stop_Axis / Stop_Axis_Ext functions has been initiated in the defined axis (cf. *cam_started)*

engage

TRUE = a positioning operation has been started in the defined axis with Pos_Axis / Pos_Axis_Ext and the current nominal position *nom_slave_pos* (slave or master) has not yet reached the target position

additionally for *axis* = 1…11:

TRUE = a valid curve has been started in the defined axis with Start_Cam_Axis and the current nominal position *nom_slave_pos* has not yet reached the current curve position *curve_pos* (engaging process); see also Rd_Status_Axis

disengage

TRUE = in the defined axis, a curve started with Start_Cam_Axis) has been stopped with Stop_Axis / Stop_Axis_Ext and the axis is not yet in standstill (disengaging process)

delta_s_ok

TRUE = following error *delta_s* with active feedback position control is inside the admissible window; the maximum admissible following distance is to be specified for each axis in the corresponding system parameters for the analog and CAN axes

ref_reached
TRUE = reference/position value has been set (Change_Pos_-Axis FB)

drive_err
TRUE = fault in the servo amplifier
Faults in the servo amplifier can be reset using the Control_Status_Axis FB. (They can be queried using the Rw_Param_Axis FB.)

cam_started
TRUE = a curve has been started in the specified axis

ext_achse
TRUE = axis exists i.e. is activated via para[332] or para[451] (the master axis 0 is always active)

ext_LD2000
TRUE = axis is a CAN axis (servo amplifier LD 2000)

ext_LD2000_prg
TRUE = the parameters of the individual servo amplifier permit its operation as CAN axis with the MotionPLC

ext_LD2000_init
TRUE = the individual servo amplifier has been successfully initialized

ext_LD2000_online
TRUE = communication with the individual servo amplifier is running

ext_LD2000_operational
TRUE = CAN axis is operational according to CANopen

drive_enable
TRUE = MotionPLC enables the specific axis and the individual servo amplifier switches to its enabled state

drive_forwards
TRUE = axis is running forward

drive_backwards
TRUE = axis is running backward

_bit BOOL;
State of the selected bit

- Example:

Declaration:

```
status_bit_axis: Rd_Status_Bit_Axis;
online_axis: BOOL;
```

Program in ST:

```
status_bit_axis(axis:=4, bit_no:=24);
online_axis:=status_bit_axis._bit;
status_bit_axis(axis:=5);
online_axis:=online_axis AND status_bit_axis._bit;
```

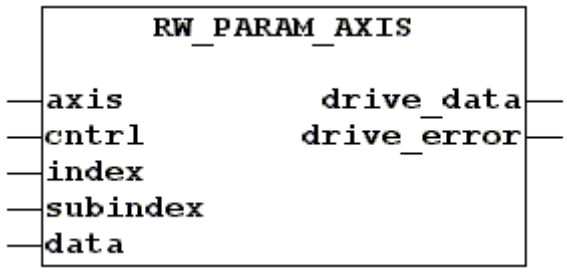
The variable `online_axis` is TRUE if axes 4 and 5 are online.

RW_Param_Axis *Axis Control*

Permits reading and writing of external servo amplifier parameters via CAN bus (SDOs)

You should not call this FB cyclically in a program: It can extend the PLC cycle time by approx. 25 ms.

- Function block:



- Variables:

axis BYTE;
CAN axis only:
1 … 11 = y axis 1 ... 11 (slave)

cntrl BYTE;
Data flow direction:
34 [DOWNLOAD*] = send data to the servo amplifier
64 [UPLOAD*] = query data to from the servo amplifier

index (WORD), *subindex* (BYTE), *data* (DINT), *drive_data* (DINT): refer to the CANopen documentation of the amplifier on the attached CD (ba_can_e.pdf)

drive_error BYTE;
0 = no error
1 = communication error
2 = timeout
3 = invalid axis

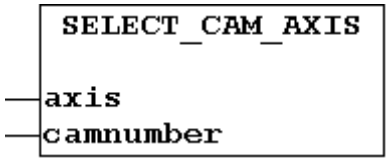
*Constant defined with the global variables in the library.

Select_Cam_Axis *Axis Control*

Selects a curve for an axis to be defined

See also: Start_Cam_Axis

- Function block:



- Variables:

axis BYTE;
1 … 11 = y axis 1 … 11 (slave)

camnumber DINT;
Curve number: 0…99

- Example:

Declaration:

```
cam_sel: Select_Cam_Axis;
new_cam: word;
status_axis: Rd_Status_Axis;
```

Program in ST:

```
new_cam:=0;
status_axis(axis:=2);
IF new_cam<>status_axis.act_curve THEN
   cam_sel(axis:=2, camnumber:=new_cam);
END_IF;
```

This program part sets a new curve number with 0 and checks whether this curve has already been selected. If this is not the case the new curve will be selected.

Start_Cam_Axis *Axis Control*

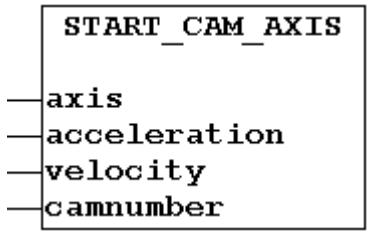
Starts the selected curve in the defined axis using the preset values for the acceleration and engaging speed (see also Rd_Status_Axis)

If a curve has already been started at this moment it will be aborted and the drive will be positioned onto the new curve (engaging process); exception: see para[181] in the Operating Instructions.

For the master, the reference value will not be set even if specified so with para[115].

See also: Select_Cam_Axis

- Function block:



- Variables:

axis	BYTE; 1 … 11 = y axis 1 … 11 (slave)
acceleration	DINT; Acceleration of the axis for the engaging process in increments/s^2 Negative values and 0 are internally converted to the smallest possible acceleration value.
velocity	DINT; Speed for the engaging process in increments/s, ≥ 0!
camnumber	DINT; Curve number: 0…99

- Example:

Declaration:

```
cam_start: Start_Cam_Axis;
pos_a: Pos_Axis;
start_new: BOOL;
start_old: BOOL;
```

Program in ST:

```
IF start_new AND start_old=FALSE THEN
   cam_start(axis:=2, acceleration:=50000, velocity:=40000,
 camnumber:=0);
   pos_a(axis:=0, acceleration:=10000, velocity:=15000,
 position:=100000);
END_IF;
start_old:=start_new;
```

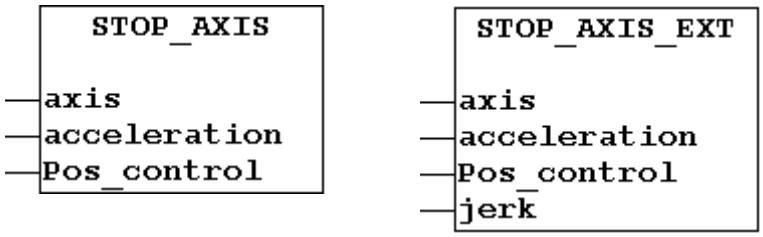
The one-time call of the FB (positive edge of `start_new`) starts curve 0 in axis 2 (slave) and positioning of axis 0 (virtual master axis) to the target position 100,000 using the acceleration and speed values specified.

Stop_Axis / Stop_Axis_Ext *Axis Control*

Stops the run operation of the defined axis

See also: Go_Axis / Go_Axis _Ext, Pos_Axis / Pos_Axis_Ext, Start_Cam_Axis

- Function block:



- Variables:

axis BYTE;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

acceleration DINT;
Deceleration of the axis in increments/s^2
Negative values and 0 are internally converted to the smallest possible deceleration value.

Pos_control BOOL;
FALSE = no feedback position control in standstill
TRUE = feedback position control in standstill remains active

jerk DINT;
Jerk limitation of the axis in increments/s^3
Negative values and 0 result in positioning without jerk limitation (as for the Stop_Axis FB)

- Example:

Declaration:

```
stop_movement: Stop_Axis;
stop_new: BOOL;
stop_old: BOOL;
I: INT
```

Initialization in ST:

```
stop_movement.acceleration:=2500000;
stop_movement.Pos_control:=FALSE;
```

Program in ST:

```
IF stop_new=FALSE AND stop_old THEN
   FOR I:=1 TO 7 BY 1 DO
      Stop_movement(axis:=I);
   END_FOR;
END_IF;
stop_old:=stop_new;
```

The slave axes 1 to 7 will be stopped with the falling edge of `stop_new` and decelerated using the same value for that. Feedback position control is inactive for all axes in standstill.

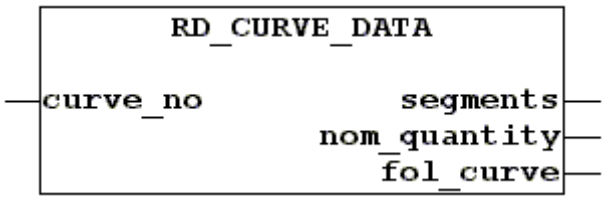
4.2 Cam Data (Curve data)

Rd_Curve_Data *Cam Data*

Provides information about a specific curve

See also: Wr_Fol_Curve, Wr_Nom_Quantity

- Function block:



- Variables:

curve_no BYTE;
Number of the desired curve: 0 … 99

segments BYTE;
Number of segments in the curve;
0 = curve does not exist

nom_quantity DINT;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

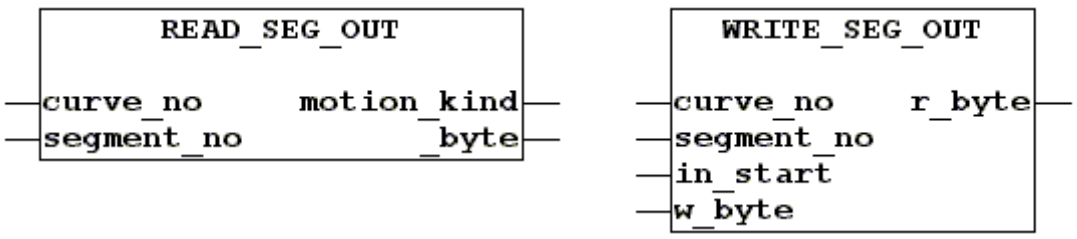
fol_curve INT;
No value is processed here. The output variable is retained for compatibility reasons with the MotionCard LD100.

Read_Seg_Out / Write_Seg_Out *Cam Data*

Provides information about the kind of motion and the status of the disposable data byte of a curve segment / Writes the disposable data byte

See also: Wr_Fol_Curve, Wr_Nom_Quantity

- Function block:



- Variables:

curve_no BYTE;
Number of the desired curve: 0 … 99

segment_no BYTE;
Number of segments in the curve

motion_kind BYTE;
Type and form of motion in the selected curve segment

For this, specific constants have been defined at the global variables, under "kind of motion".

Byte	Constant	Type	Form
0			Segment does not exist
5	V_CONST	v = const	
6	V_EQ_0	v = 0	
11	HARM_PP	P1 → P2	Harmonic
12	HARM_PV	P → v	Harmonic
13	HARM_VP	v → P	Harmonic
21	CYCL_PP	P1 → P2	Cycloidal
22	CYCL_PV	P → v	Cycloidal
23	CYCL_VP	v → P	Cycloidal
24	CYCL_VV	v1 → v2	Cycloidal

_byte BYTE;
Disposable data byte = programmed output signals in the selected curve segment (Out 1…8; 1 = High)

in_start BOOL;
Write condition:
FALSE = the data byte can only be written if the curve is not yet started
TRUE = the data byte can also be written if the curve is already started

w_byte BYTE;
Condition to be written to the disposable data byte (Out 1…8; 1 = High)

r_byte BYTE;
Condition of the disposable data byte after writing

- Example:

Declaration:

```
I: BYTE;
re_seg_byte: Read_Seg_Out;
wr_seg_byte: Write_Seg_Out;
```

Program in ST:

```
FOR I:=1 TO 10 DO
   re_seg_byte(curve_no:=3, segment_no:=I);
   wr_seg_byte.curve_no:=re_seg_byte.curve_no;
   wr_seg_byte.segment_no:=I;
   wr_seg_byte.in_start:=FALSE;
   IF re_seg_byte.motion_kind=5 THEN;
      wr_seg_byte.w_byte:=re_seg_byte._byte OR 2#10000000;
   ELSE;
      wr_seg_byte.w_byte:=re_seg_byte._byte AND 2#01111111;
   END_IF;
   wr_seg_byte();
END_FOR;
```

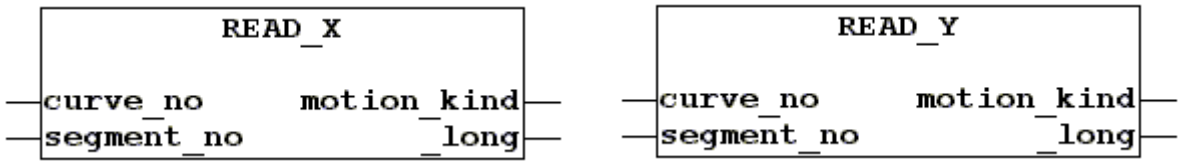
The kind of motion of each of the 10 curve segments is read. If this is defined as v = constant (fixed speed ratio between master and slave) then the MSB of the disposable data byte is set to 1 and in all other cases to 0. The other bits of the data byte are not changed.

Read_x / Read_y *Cam Data*

Provides information about the kind of motion and the end position x (master) or y (slave) in a specific segment of a selected curve

See also: Write_x, Write_y

- Function block:



- Variables:

  curve_no, *segment_no*, *motion_kind* as for Read_Seg_Out (see above)

  _long DINT;
  for *segment_no* = 1…31:
  end position x or y in the segment to be specified
  for *segment_no* = 0:
  start position x (always 0) or y in segment 1

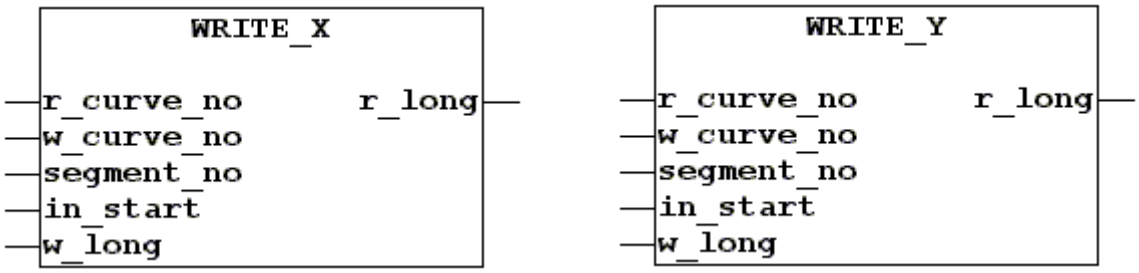
- Example: see Write_x, Write_y

Write_x / Write_y *Cam Data*

Copies a source curve to the buffer storage where it replaces the end value x / y of the specified curve segment. Then it recalculates the end value of the other axis, shifts all end values x and y of the following segments by the amount of the appropriate modifications, overwrites the specified target curve with the curve data from the buffer storage (write conditions specified in *in_start*) and returns the end position value after overwriting the target curve.

See also: Read_x, Read_y

- Function block:



- Variables:

r_curve_no BYTE;
Number of the source curve: 0 … 99

w_curve_no BYTE;
Number of the target curve: 0 … 99

segment_no BYTE;
Number of the segment in the specified curve: 0, 1 ... 30
The 0 value permits access to the start values of the curve segment no. 1.

in_start BOOL;
Write condition:
FALSE = Curve not yet started (obligatory)

Additional write conditions:
– for Write_**x**
The value specified must be bigger than the end value in the preceding curve segment.
– for Write_**x** and Write_**y**
The segment number must not exceed the number of the last segment in the specified curve and the kind of motion must not be "v1 → v2" (24).

w_long, DINT;
for *segment_no* = 1…31:
End position x / y to be written in the specified segment of the source curve

for *segment_no* = 0:
Start position x (always 0) or y of curve segment 1

r_long, DINT;
for *segment_no* = 1…31:
End position x / y in the specified segment of the target curve (for checking)
for *segment_no* = 0:
Start position x (always 0) or y of curve segment 1

◆ Example:

Declaration:

```
delta_y23: DINT;
end_y2: DINT;
end_y3: DINT;
error: BOOL;
rd_end_y: Read_y;
wr_end_y: Write_y;
```

Program in ST:

```
rd_end_y(curve_no:=0, segment_no:=3);
IF rd_end_y.motion_kind=11 OR rd_end_y.motion_kind=21 THEN
   end_y3:=rd_end_y._long;
   rd_end_y(segment_no:=2);
   end_y2:=rd_end_y._long;
   delta_y23:=end_y3 - end_y2;
   wr_end_y.r_curve_no:=0;
   wr_end_y.w_curve_no:=5;
   wr_end_y.segment_no:=3;
   wr_end_y.in_start:=FALSE;
   wr_end_y.w_long:=delta_y23*2;
   wr_end_y();
   IF wr_end_y.r_long=wr_end_y.w_long THEN
      error:=FALSE;
   ELSE;
      error:=TRUE;
   END_IF;
END_IF;
```

If the kind of motion is defined as "Pos1 → Pos2" in segment 3 of source curve 0 then the y distance in segment 3 is calculated, multiplied by 2, and transmitted to target curve 5. After that the write operation is checked; it was successful if the `error` variable is FALSE.

4.2.1 New Cam

The FBs grouped here provide the necessary tools for creating curves and the advanced handling of curves from the PLC.

! Any intervention in the curve structure is very **critical** and not easy to understand.
These FBs should therefore only be used by **experienced** and **trained** users who are **perfectly familiar** with curve applications.

The supplied file "fl_saw.pro" is to demonstrate for the Flying Saw technology function how to calculate individual curve segments, convert them to the correct format and write them into the structured curve array.

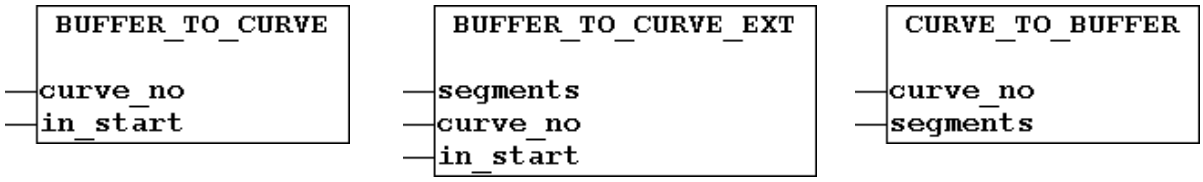
Buffer_to_Curve / Buffer_to_Curve_Ext / Curve_to_Buffer

Cam Data (New Cam)

Copies a curve or a part of it from the RAM data area to the working area (buffer): Curve_to_Buffer
Writes a curve from the working area to the RAM data area: Buffer_to_…

See also: Rd_Curve_Array, Wr_Curve_Array; Save_Parameter (p. 92)

- Function block:



- Variables:

curve_no BYTE;
Number of the curve to be copied: 0 ... 99

in_start BOOL;
Write condition:
FALSE = curve not yet started (obligatory)

segments BYTE;
Number of the curve segments to be copied or written:
for Curve_to_Buffer:
0 = All existing segments are read in. The segment number in the working area is set to the real number of segments in the data area
1 … 30 = The specified number of segments is read in. This is also the case if there are less segments in the data area than specified. In the working area, the specified segments will be installed.

for Buffer_to_Curve_Ext:
0 = the curve is to be deleted
1 … 30 = the specified curve in the data area is replaced by the specified number of segments

- Example:

  Declaration:

```
read_curve: Curve_to_Buffer;
write_curve: Buffer_to_Curve;
write_curve_ext: Buffer_to_Curve_Ext;
save: Save_Parameter;
```

  Program in ST:

```
read_curve(curve_no:=0, segments:=0);
write_curve(curve_no:=1, in_start:=FALSE);
write_curve_ext(segments:=0, curve_no:=0, in_start:=FALSE);
save();
```

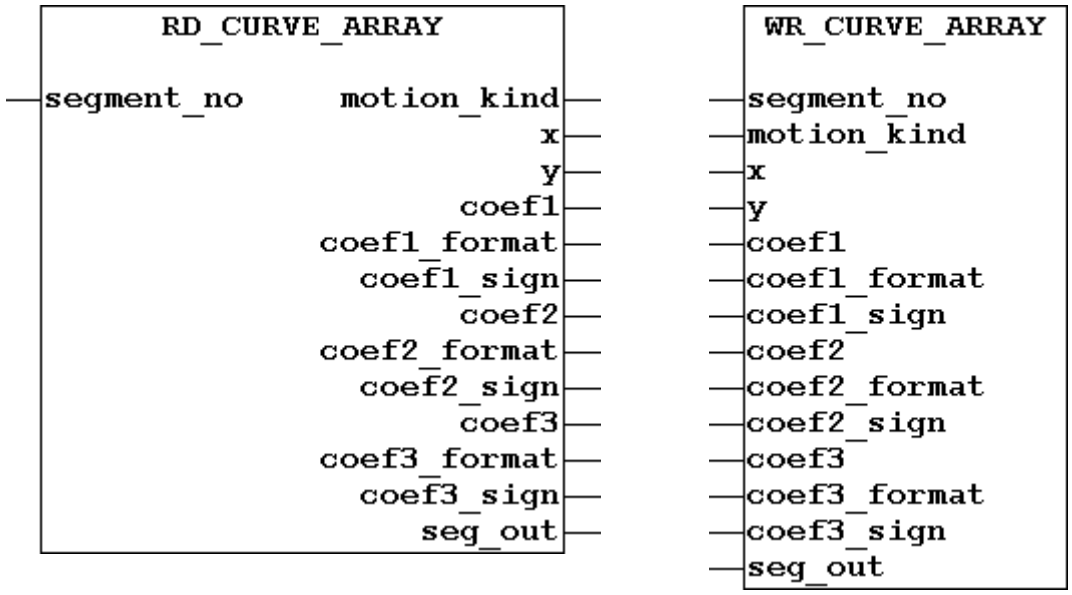
  In a single program cycle the existing segments of curve 0 are copied from the RAM data area to the working area (buffer) and are then written back into the data area as curve 1 (with the number of segments read in before). Then curve 0 is deleted in the RAM data area. Subsequently, the data area (with all curves) is copied to the flash memory for data security reasons.

Rd_Curve_Array / Wr_Curve_Array *Cam Data* (*New Cam*)

Provides / creates data for a curve segment in the working area (buffer)

See also: Buffer_to_Curve, Buffer_to_Curve_Ext, Curve_to_Buffer

- Function block:



◆ Variables:

segment_no BYTE;
Number of the segment: 0, 1 … 30
The x and y values of segment 0 provide the start positions of segment 1 (master: x = 0).

motion_kind BYTE;
Type and form of motion in the selected curve segment

For this, specific constants have been defined at the global variables, under "kind of motion".

Val.	Constant	Type	Form	Coefficients used		
				coef1	coef2	coef3
5	V_CONST	v = constant		●		
6	V_EQ_0	v = 0				
11	HARM_PP	Pos 1 → Pos 2	Harmonic			
12	HARM_PV	Pos → v	Harmonic			
13	HARM_VP	v → Pos	Harmonic			
21	CYCL_PP	Pos 1 → Pos 2	Cycloidal		●	
22	CYCL_PV	Pos → v	Cycloidal		●	
23	CYCL_VP	v → Pos	Cycloidal		●	
24	CYCL_VV	v1 → v2	Cycloidal	●	●	
50	Reserved for interpolation in the cnc_2d and cnc_3d libraries			Are used in another context by the interpolation		
51						
100	FREE_CAM	y value table[2]		●	●	●

x DINT;
x end value of the segment in increments of the master axis

y DINT;
y end value of the segment in increments of the slave axis that is used in the curve

coef1 DWORD;
– For *motion_kind* ≤ 24:
Absolute value of the start speed ratio:

$$coef1 = \left|\frac{v_y}{v_x}\right| * format \text{ (s. b.)}$$

– For *motion_kind* = 100:
Start address of the table containing the y values

[2] The values must be of DINT type and related to the end of the previous curve segment. The controller will linearly interpolate the curve shape between each two y values (you may check the curve shape in the BB2100K curve editor: load and calculate the curve). Maximum table length is 64 Kbytes. The motion type of the curve segment following the table segment must not be 24 ("v1 → v2").

coef1_sign BYTE;
Sign of the start speed ratio (for *motion_kind* ≤ 24):
0 = "+"
1 = "-" for Wr_Curve_Array
≠0 = "-" for Rd_Curve_Array

coef2 DWORD;
– For *motion_kind* ≤ 24:
Absolute value of the distance ratio:

$$coef2 = \left|\frac{\Delta y}{\Delta x}\right| * format \quad \text{(s. b.)}$$

Δy = end position y – start position y,
Δx = end position x – start position x

– For *motion_kind* = 100:
Number of master increments per y value (table interval)

coef2_sign BYTE;
Information for the display in the curve editor of the BB2100K application: 1 = ☑ (for *motion_kind* ≤ 24):

Bit	Check box	Option			
		v1 → v2		v = constant	
2^0	☐ Final position x [inc.]	1/0	1/0	0	1
2^1	☐ Final position y [inc.]	0	1	1	0
2^2	☐ Final velocity [v_y/v_x]	1	0	0	0

coef1_format,
coef2_format INT;
Formatting of coef1 or coef2 (for *motion_kind* = 1…24):

coef#_format	*format*	Utilisation
1	2^0	For $\left\|\frac{v_y}{v_x}\right\| > 2^{16}$ or $\left\|\frac{\Delta y}{\Delta x}\right\| > 2^{16}$
0	2^{16}	Standard
-1	2^{32}	For $\left\|\frac{v_y}{v_x}\right\| < 1$ or $\left\|\frac{\Delta y}{\Delta x}\right\| < 1$ only

(# = 1 or 2)

coef3,
coef3_format DWORD;
coef3_sign BYTE;
– For *motion_kind* ≤ 24: not used in the cam plate application
– For *motion_kind* = 100 (*coef3* only): table length (number of y values in the table)

seg_out BYTE;
Disposable data byte

- Example:

  Declaration:

```
write_curve_ext: Buffer_to_Curve_Ext;
write_segment: Wr_Curve_Array;
```

  Program in ST:

```
write_segment.segment_no:=0;
write_segment.x:=0;
write_segment.y:=0;
write_segment.coef1:=65536 * 33 / 47;
write_segment.coef1_format:=0;
write_segment.coef1_sign:=1;
write_segment();
write_segment.segment_no:=1;
write_segment.motion_kind:=5;
write_segment.x:=47000;
write_segment.y:=-33000;
write_segment.coef1:=65536 * 33 / 47;
write_segment.coef1_format:=0;
write_segment.coef1_sign:=1;
write_segment.coef2_sign:=2#00000010;
write_segment();
write_curve_ext(segments:=1, curve_no:=1, in_start:=FALSE);
```

  When executing this program part once, a new curve no. 1 is created that effects an anti parallel feed of the master and slave drives in a 47:33 ratio.

4.3 Cam Tracks (cam-operated switchgroup)

With the cam-operated switchgroup up to 16 tracks can be realized, each of which may contain an arbitrary number of switching points.

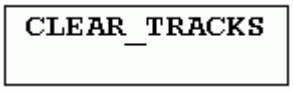
The state of operation of the individual tracks is stored bit by bit in a specific variable. Additionally, up to15 tracks can be output on the Q1.0…Q3.4 digital outputs of the MotionPLC.

The variable and outputs are set by the operating system, i.e. independently of the PLC cycle (refresh period = cycle time of the cam plate application).

Clear_Tracks *Cam Tracks*

Deletes all tracks of the cam-operated switchgroup (this FB should be called during initialization of the PLC program)

- Function block:



Init_Cam_Gear *Cam Tracks*

Initializes the cam-operated switchgroup feature

See also: Track

- Function block:

- Variables:

modus BYTE;
Characteristic of the function:
0 = inactive
1 = for circular movements, the cams are recurring beyond the limits of the specified range (example: slave in a Rotating Cutter application)
2 = for movements between 2 target points, the cams are output only in the specified range (example: slave in a Flying Saw application)

axis BYTE;
0 = x axis (master: simulation, virtual)
1 … 11 = y axis 1 … 11 (slave)

interval_start, *interval_end* DINT;
These are the positions where the counting range of the cam-operated switchgroup feature starts and ends (condition: *interval_start* < *interval_end*). This range is converted to a raster with 8192 steps at maximum.
If the *interval_end* – *interval_start* difference contains more steps than the maximum number of the raster then this difference is subdivided into 8192 steps (see the example for the Track FB further below).

out_var_pointer POINTER_TO_WORD;
Address of the variable supposed to contain the states of the cam tracks

out_hardware BYTE;
Number of the hardware outputs where the switching states are to be output automatically by the operating system (these outputs must then not be used by other FBs):
0 = no output
1…15: tracks 0…14 on terminals Q1.0…Q3.4 (consecutively; example: 2 ⇒ track 0 on Q1.0 and track 1 on Q1.1)

Time_Comp *Cam Tracks*

Allows for advancing all cams of a track group by a certain time for compensating dead times etc.

- Function block:

- Variables:

number BYTE;
Number of the track group: 0…3

start_track BYTE;
First track of the group

start_track BYTE;
Last track of the group

_time TIME;
Compensation time

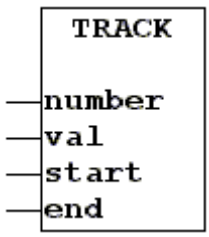
Track *Cam Tracks*

Sets/resets the switching points (cams) of a track

Tracks once defined are stored retentively. This may result in an unexpected switching behavior when defining a new cam, because the desired switching range possibly overlaps the one of a cam having been specified earlier. To ensure that no "forgotten" cam will negatively affect the desired switching range you should reset all tracks in the initialization section of the PLC program: Clear_Tracks FB.

See also: Init_Cam_Gear

- Function block:



- Variables:

number BYTE;
Assigns a track to the cam to be defined consecutively: 0…15

val BOOL;
FALSE = the switching positions of the specified track are reset
TRUE = the switching positions of the specified track are set

start DINT;
Start position:
Specifies the axis position where the cam is to switch on. When it is fallen below, the cam switches off again.

The position value is internally converted to a raster step (see the Init_Cam_Gear FB). The switching accuracy related to the specified position depends on the number of counting steps within a raster step (see the example).

end DINT;
End position:
Specifies the axis position where the cam is to switch off when the position is exceeded. When this position is reached from above the cam switches on again.

The position value is internally converted to a raster step (see the Init_Cam_Gear FB). The switching accuracy related to the specified position depends on the number of counting steps within a raster step (see the example).

- Example:

  Declaration:

```
cam_gear_out: WORD; (* contains the switching states *)
Init_Cam: Init_Cam_Gear;
Set_Track: Track;
```

  Program in ST:

```
Init_Cam.modus:= 1; (* circ. movem., closed count. range *)
Init_Cam.axis:= 0; (* master *)
Init_Cam.interval_start:= 0;
Init_Cam.interval_end:= 20000;
Init_Cam.out_var_pointer:= ADR(cam_gear_out);
Init_Cam(out_hardware:= 2); (* X23.2 and X23.3 *)
Set_Track(number:= 0, val:= TRUE, start:= 10000, end:=
                                        20000); (* Track 0 *)
Set_Track(number:= 1, val:= TRUE, start:= 5000, end:=
                                        15000); (* Track 1 *)
```

  The one-time call of this program part activates the cam-operated switchgroup feature for a circular movement (closed counting range), i.e., the signals obtained in the range from 0 to 20,000 of the master (axis 0) will be repeated in the ranges between 20,000 and 40,000, 40,000 and 60,000 etc. The terminals Q1.0 und Q1.1 are activated to directly output the tracks 0 and 1. The `cam_gear_out` variable provides the current switching states through the bits 2^0 (track 0) and 2^1 (track 1), 1 = ON.

  Because the counting range of 20,000 exceeds the maximum step number of the raster (8,192) one raster step consists of 2 or 3 counting steps (20,000/8,192 ≈ 2.4).

  In this example, the outputs provide signals as usually obtained by incremental encoders (0° and 90° signal tacks). Thus, a counter might be controlled by these outputs.

4.4 CAN Bus

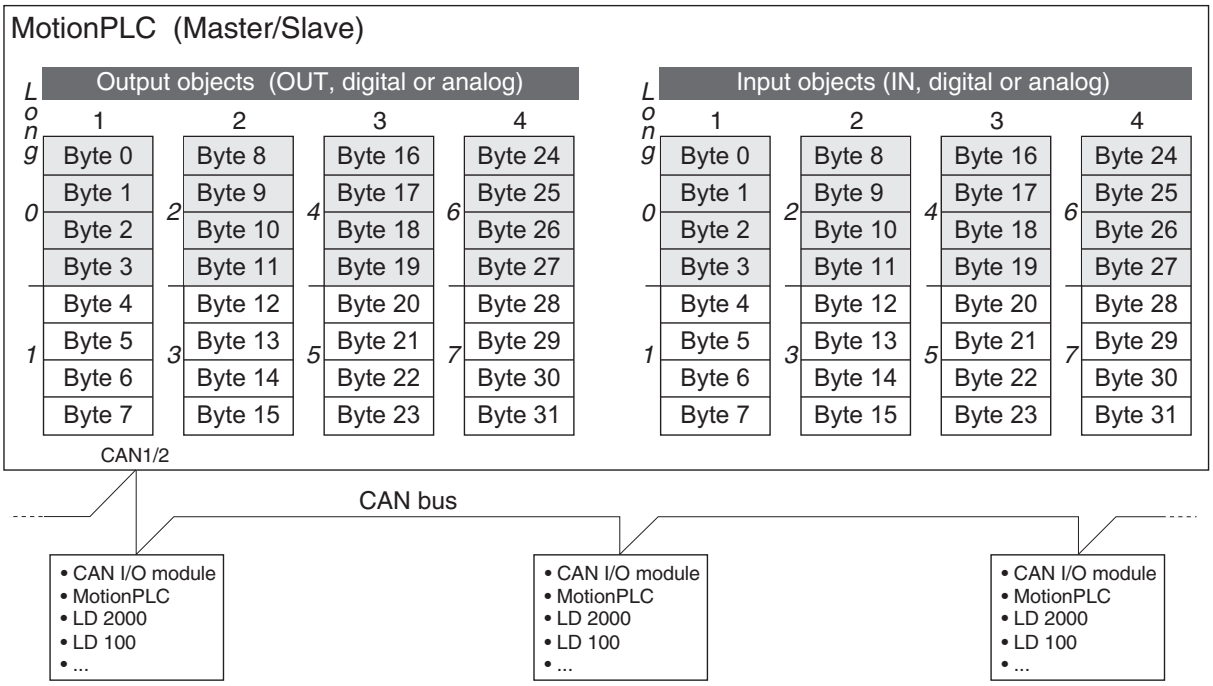
The FBs in this branch are designated for the standard communication according to the CANopen protocol using the first CAN bus (second CAN bus: see section 4.4.1). Para[205] specifies which of the two electrical connections is to be used for that (see Operating Instructions). The following operations can be performed:

- Read inputs on the CAN bus (CAN_In_…)
- Write outputs on the CAN bus (CAN_Out_…)
- Read outputs on the CAN bus (Rd_…)
- Initialize and reset CAN bus, status inquiry (CAN_Init/Reset/Status)
- Read/write SDO data channels (SDO_…)

The CAN inputs/outputs are read/written with a maximum delay time of 10 ms after calling the individual FB (exception: SDO_Request is instantly executed).

The MotionPLC works with a fixed baud rate of **500 kBaud** (exception: see the Operating Instructions manual; CAN Link: 250 kBaud). The CAN modules connected must also be set to this baud rate and an object address be selected (1…127, standard: 1 … 4; see Operating Instructions: CAN parameters para[55] and following).

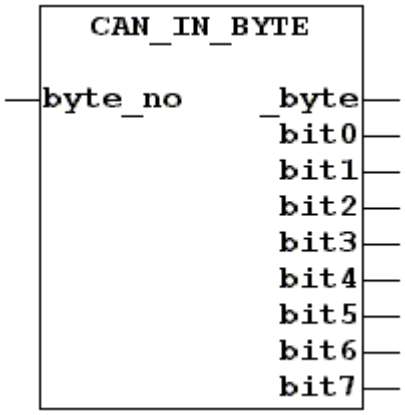
The following illustration shows the input and output bytes and their assignment to the individual objects:



CAN_In_Byte *CAN Bus*

Provides the states of an input object byte in a byte and in individual bits

- Function block:



- Variables:

byte_no BYTE;
Specifies the input byte:
0 … 7: Object 1
8 … 15: Object 2
16 … 23: Object 3
24 … 31: Object 4

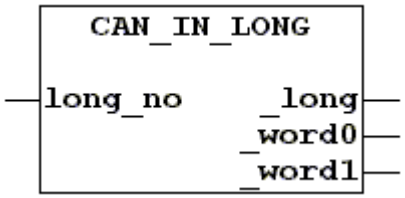
_byte BYTE;
Content of the input byte

bit0…7 BOOL;
Logic states of the bits 0…7 of *_byte*

CAN_In_Long *CAN Bus*

Provides the states of an input object Long in a Long and in two Words

- Function block:



- Variables:

long_no BYTE;
Specifies the input long
0, 1: Object 1

2, 3: Object 2
4, 5: Object 3
6, 7: Object 4

_long DINT;
Content of the input Long

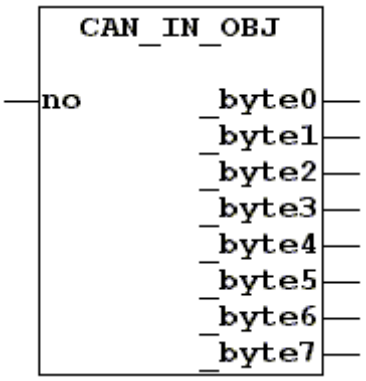
word0 WORD;
Content of the least significant Word (LSW) of *_long*

word1 WORD;
Content of the most significant Word (MSW) of *_long*

CAN_In_Obj *CAN Bus*

Provides the states of the 8 bytes of an input object

- Function block:



- Variables:

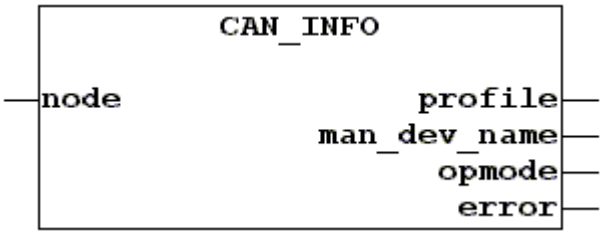
no BYTE;
Specifies the input object
1 … 4: Object 1…4

byte0…7 BYTE;
Content of the bytes 0…7 of the input object

CAN_Info *CAN Bus*

Informs about the connected device

- Function block:



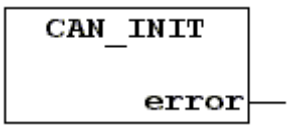
- Variables:

node	BYTE; 1 … 16: CAN address
profile	WORD; Device type (specified according to CANopen) 191h: CAN Remote I/O 192h: servo amplifier
man_dev_name	DWORD; Manufacturer device name
opmode	BYTE; Operating mode (with LD 2000 only: OPMODE parameter)
error	BYTE; CAN bus error while initializing (0 = no error)

CAN_Init *CAN Bus*

Initialization of the CAN controller after changing of a parameter (e.g. address); the CAN loader is called (will be done automatically when calling the Loader FB). The *new_dat* bit of the changed object is reset (see the CAN_-Status FB further below).

- Function block:



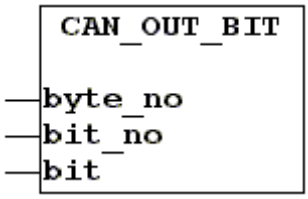
- Variables:

error	BYTE; 0 = no error, otherwise: number of the erroneous parameter

CAN_Out_Bit *CAN Bus*

Sets a specific bit in an output object to 0 (FALSE) or 1 (TRUE).

- Function block:



- Variables:

byte_no BYTE;
Specifies the output byte
0 … 7: Object 1
8 … 15: Object 2
16 … 23: Object 3
24 … 31: Object 4

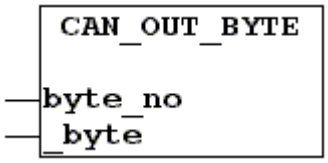
bit_no BYTE;
Bit in the output byte: 0…7

bit BOOL;
Logic state to be output: TRUE = 1 or FALSE = 0

CAN_Out_Byte *CAN Bus*

Sets a specific byte in an output object

- Function block:



- Variables:

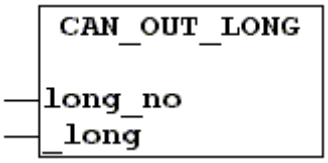
byte_no BYTE;
Specifies the output byte
0 … 7: Object 1
8 … 15: Object 2
16 … 23: Object 3
24 … 31: Object 4

_byte BYTE;
Data to be output

CAN_Out_Long *CAN Bus*

Sets a specific Long in an output object

- Function block:



- Variables:

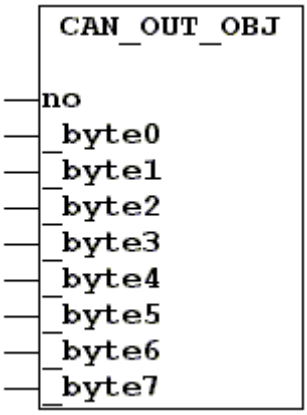
long_no BYTE;
Specifies the output Long
0, 1: Object 1
2, 3: Object 2
4, 5: Object 3
6, 7: Object 4

_long DINT;
Data to be output

CAN_Out_Obj *CAN Bus*

Sets an output object

- Function block:



- Variables:

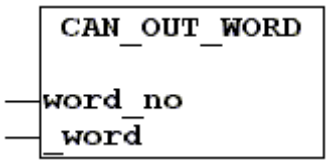
no BYTE;
Specifies the output object
1 … 4: Object 1…4

byte0…7 BYTE;
Data to be output

CAN_Out_Word *CAN Bus*

Sets a specific Word in an output object

- Function block:



- Variables:

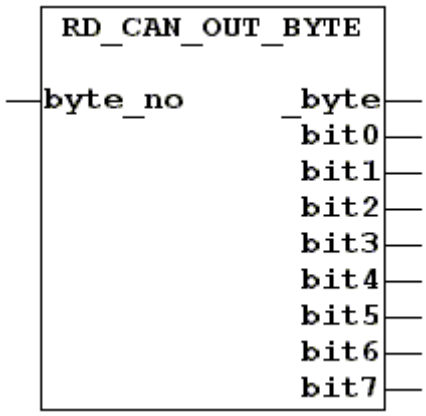
word_no BYTE;
Specifies the output Word
0 … 3: Object 1
4 … 7: Object 2
8 … 11: Object 3
12 … 15: Object 4

_word WORD;
Data to be output

Rd_CAN_Out_Byte *CAN Bus*

Provides the states of a specific output byte in a byte and in individual bits

- Function block:



- Variables:

byte_no BYTE;
Specifies the output byte
0 … 7: Object 1
8 … 15: Object 2
16 … 23: Object 3
24 … 31: Object 4

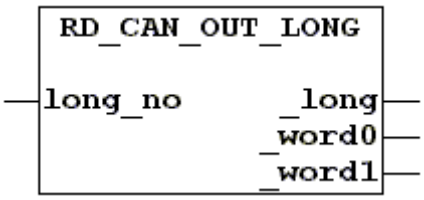
_byte BYTE;
Data to be read

bit0…7 BOOL;
Logic states of the bits 0…7 of *_byte*

Rd_CAN_Out_Long *CAN Bus*

Provides the states of a specific output object Long in a Long and in two Words

- Function block:



- Variables:

long_no BYTE;
Specifies the output Long
0, 1: Object 1
2, 3: Object 2
4, 5: Object 3
6, 7: Object 4

_long DINT;
Content of the output Long

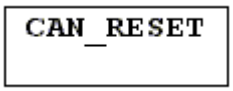
word0 WORD;
Content of the least significant Word (LSW) of *_long*

word1 WORD;
Content of the most significant Word (MSW) of *_long*

CAN_Reset *CAN Bus*

The *new_dat* bit in all CAN objects is reset (see CAN_Status). This FB should be executed after a bus error has been detected.

- Function block:



CAN_Status *CAN Bus*

Provides information about a data transfer operation. The status is reset after calling the FB.

- Function block:

- Variables:

no BYTE;
0 = synchronisation telegram (CAN-SYNC)
1…4 = input data (CAN-IN)
5…8 = output data (CAN-OUT)

new_dat BOOL;
TRUE = new data arrived (CAN-IN) / output (CAN-OUT)

bus_off BOOL;
TRUE = Bus error occurred (see the Operating Instructions manual at terminal operation: command "re"); an electrical connection must exist. At TRUE a short-circuit may exist, or several CAN modules have been configured with identical addresses in the CAN-Out objects or as masters at the same time.

SDO_Request *CAN Bus*

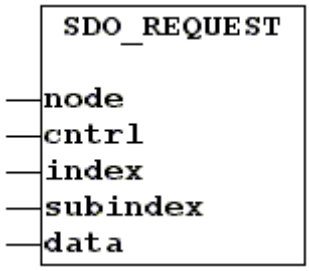
Retrieves data from a server (slave), here the servo amplifier

For the CANopen communication with servo amplifiers LD 2000 special function blocks have been grouped under CoDeSys in the attached library **ld2000.lib**. This library is structured as an internal library and can – if loaded with *File/Open* – be modified and adapted to personal requirements.

Important: Before each recompilation of ld2000.lib, the standard.lib and gel8240.lib libraries must be linked again.

The function blocks of the ld2000.lib are basically self-explaining so that no further explanations are given in this manual. An example program for the use of the various function blocks can be found in the CoDeSys folder: ld2000_demo.pro. Expanding the library requires exact knowledge of the ASCII object codes of the servo amplifier and their meaning (documentation: ba_can_e.pdf).

- Function block:



- Variables:

node BYTE;
CAN address: 1…127

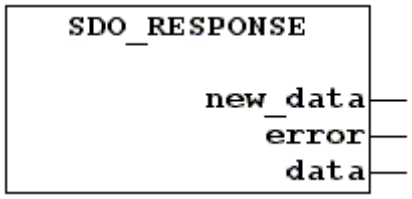
cntrl BYTE;
Data direction:
34 [DOWNLOAD*] = send data to the server (CAN slave)
64 [UPLOAD*] = retrieve data from the server

index (WORD), *subindex* (BYTE), *data* (DINT): see the CANopen documentation of the servo amplifier.

SDO_Response *CAN Bus*

Server (slave) response

- Function block:



- Variables:

new_data BOOL;
TRUE = new data available or server has responded

error BYTE;
0 = no error

data DINT *(see CANopen documentation)*

4.4.1 CAN2

The FBs in this branch are exclusively designated for controlling type LD 2000 servo amplifiers via the second CAN bus. Para[205] specifies which of the two

* Constant defined at the global variables.

electrical connections is to be used for that (see Operating Instructions). This CAN bus is then no longer available for the CAN Link functions (C-Net) listed in section 4.5.

CAN2_BusInit *CAN Bus (CAN2)*

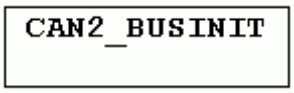
Re-initializes the CAN bus used for controlling of external axes (as executed when energizing the MotionPLC). So you can evade switching off and on the MotionPLC when a fault has occurred in the servo amplifier causing a reset (as it is often the case just during the setup procedure).

After such an initialization the corresponding servo amplifier must first be referenced if the PLC program executes this procedure after starting, too.

When calling this FB all CAN axes should be disabled.

You must not use this FB if there are axes with absolute actual value acquisition on the CAN bus.

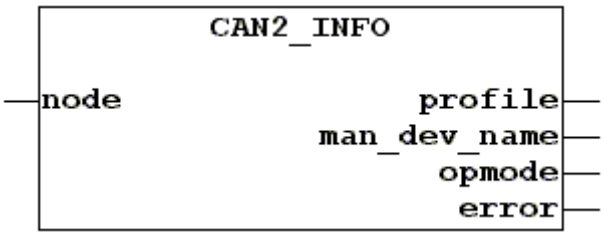
- Function block:



CAN2_Info *CAN Bus (CAN2)*

As for the CAN_Info FB (→ p. 45)

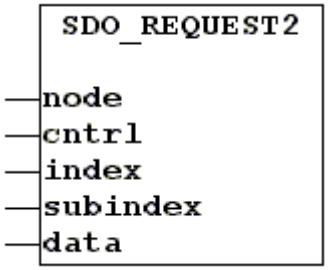
- Function block:



SDO_Request2 *CAN Bus (CAN2)*

Functionality as for SDO_Request further above

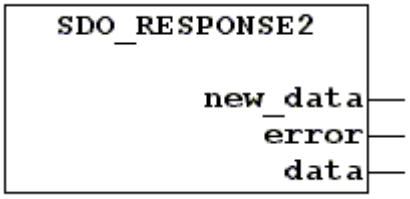
- Function block:



SDO_Response2 *CAN Bus (CAN2)*

Functionality as for SDO_Response further above

- Function block:



4.5 CAN Link (C-Net)

With the FBs in this branch a CAN network (system CAN bus) can be established using a free protocol which is not to be subject to the CANopen conventions. On the hardware side, either the CAN1 or the CAN2 interface is used depending on the programming of system parameter para[205] (see Operating Instructions).

14 I/O channels are available. Only **one device per channel at a time is allowed to transmit**, whereas the number of receivers per channel is arbitrary. It is also possible that 1 device transmits on several channels.

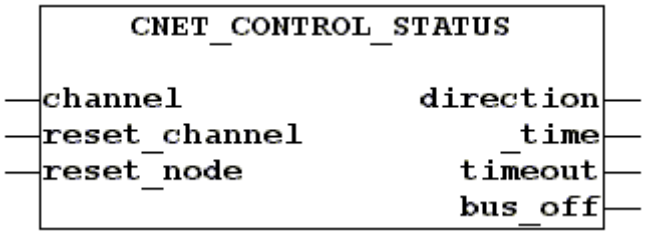
The following operations can be performed:

- Read bus status (CNET_Control_Status)
- Initialize cyclical data exchange (CNET_Start)
- Transmit and receive data objects ("Basic Jobs": CNET_Out_Obj and CNET_In_Ob)

CNET_Control_Status *CAN Link*

Informs about the state of the CAN network and controls it in conjunction with the CNET_Start FB

◆ Function block:



◆ Variables:

channel	BYTE; I/O channel: 1 … 14
reset_channel	BOOL; TRUE = specified channel is deactivated
reset_node	BOOL; TRUE = CAN network participant reset (execution may make sense, for instance, after detection of a bus error)
direction	BYTE; 0 = inactive 1 = transmit 2 = receive
_time	TIME; Time span since last transmission

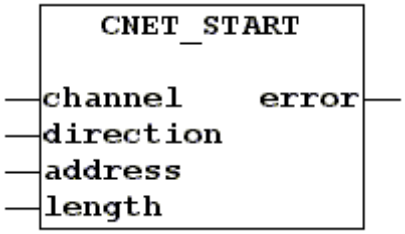
timeout BOOL;
TRUE = time for (another) transmission has been exceeded: 200 ms (cyclical transmission typically takes place every 70 ms)

bus_off BOOL;
TRUE = CAN network bus error occurred
There must be a physical connection. At TRUE a short-circuit may exist.

CNET_Start *CAN Link*

One-time execution of the FB starts up a permanent exchange of data in the background

- Function block:



- Variables:

channel BYTE;
I/O channel: 1 … 14

direction BYTE;
1 = transmit
2 = receive

address POINTER TO BYTE;
Start address of a memory area (variable, array) where data of different drives are stored, such as the nominal positions, actual positions, actual speeds, etc.

length BYTE;
Number of bytes to be transmitted: max. 70

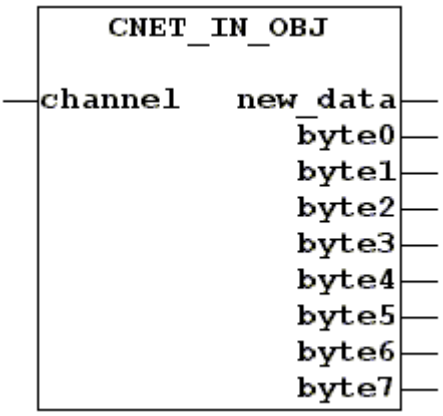
error BYTE;
Error conditions:
0 = no error
1 = invalid I/O channel number
2 = channel in use
3 = invalid direction of data
4 = invalid address or data length

4.5.1 Basic Jobs

CNET_In_Obj *CAN Link (Basic Jobs)*

Reads a data object (8 bytes) from the CAN network

- Function block:



- Variables:

channel BYTE;
I/O channel: 1 … 14

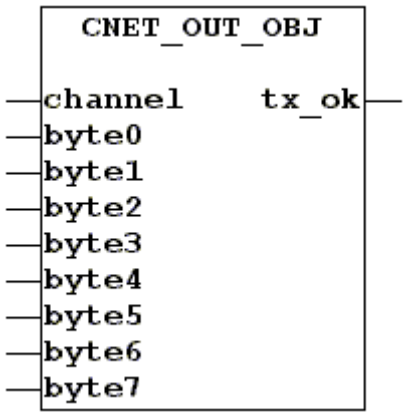
new data BOOL;
TRUE = new data available; being reset (FALSE) on reading of the data

byte0…7 BYTE;
Data to be read

CNET_Out_Obj *CAN Link (Basic Jobs)*

Writes a data object (8 bytes) into the CAN network

- Function block:



- Variables:

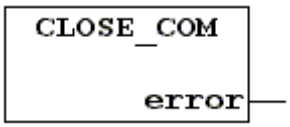
channel	BYTE; I/O channel: 1 … 14
byte0…7	BYTE; Data to be transmitted
tx_ok	BOOL; FALSE = timeout (no reception acknowledgement on the bus); object could not be transmitted within 10 ms

4.6 Communication (RS232/RS485)

Close_COM *Communication*

Closes the serial interface and resets its communication parameters to the default values for working with a PC and CoDeSys as specified in para[207/-208] (see Operating Instructions).

- Function block:



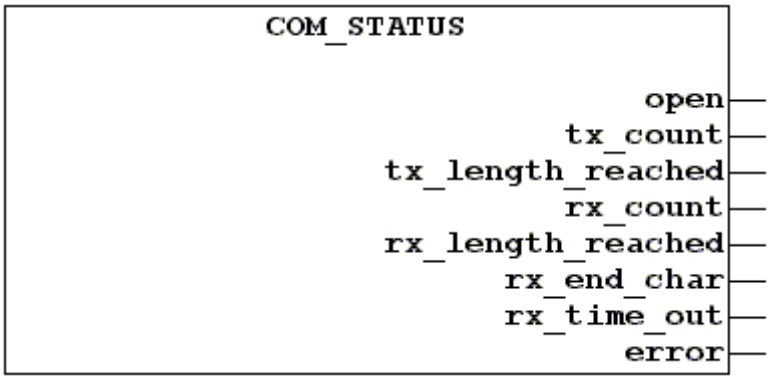
- Variables:

error BYTE;
Error condition:
0 = no error
4 = interface already closed

COM_Status *Communication*

This FB is used to control the transmit and receive sequence. It should always be executed before a transmit command or after a receive command in order to be able to react especially to the specified variables.

- Function block:



- Variables:

open BOOL;
TRUE = opened
FALSE = closed

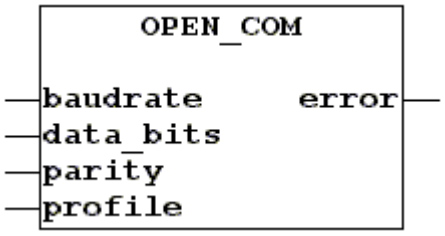
tx_count BYTE;
Number of characters transmitted so far; is being reset every time the Transmit FB is called

tx_length_reached	BOOL; TRUE = number of characters transmitted has reached the value *length* specified in the Transmit FB
rx_count	BYTE; Number of characters received so far; is being reset every time the Receive FB is called
rx_length_reached	BOOL; TRUE = number of characters received has reached the value *length* specified in the Receive FB
rx_end_char	BOOL; TRUE = the last received character is the end character specified in the Receive FB – if activated (see there); if this is not the case although an end character was specified in *end_char* then TRUE means that this character has been transmitted at least once
rx_time_out	BOOL; TRUE = the timeout period specified in the Receive FB has elapsed
error	BYTE; Error condition: 0 = no error 1 = parity error 2 = framing error 3 = overflow error 4 = interface not opened

Open_COM *Communication*

This FB must be called up before the first communication takes place (followed by the Receive FB). The interface parameters are set for the peripheral device in use (after closing of the interface, they are reset to their default values, see Close_COM).

- Function block:



◆ Variables:

baudrate WORD;
Transmission speed:
2400 / 4800 / 9600 / 19200 / 38400 / 57600 [baud]

data_bits BYTE;
Number of data bits: 7 / 8

parity BYTE;
Parity:
0 = none
1 = odd
2 = even

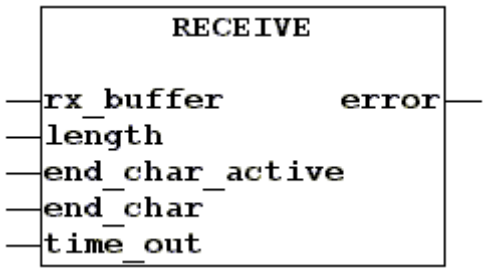
profile BYTE;
Communication protocol: 0 = free protocol

error BYTE;
Error condition:
0 = no error
4 = invalid interface parameter

Receive *Communication*

This FB should be called up immediately after opening of the interface (Open_COM FB) to prepare the communication for the expected reception of data.

◆ Function block:



◆ Variables:

rx_buffer POINTER TO BYTE;
Start address of the memory area to be used as read buffer (variable, array)

length BYTE;
Number of characters expected (data length); as an alternative or in addition to using a data end character or defining a timeout period

The internal communication buffer is limited to 255 characters!

end_char_active BOOL;
Data end character used? – TRUE / FALSE (as an alternative or in addition to using the data length or timeout specification; end character has priority);
if FALSE and *end_char* specified, the received data block can be checked by means of the COM_Status FB for the presence of the specified end character

end_char BYTE;
ASCII code of the end character used (e.g. *carriage return* [CR] = 13)

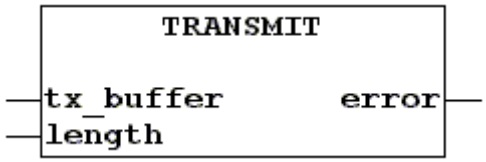
time_out WORD;
Maximum duration of transmission in milliseconds (as an alternative or in addition to using the data length or end character specification); 0 = inactive

error BYTE;
Error condition:
0 = no error
1 = parity error
2 = framing error
3 = overflow error
4 = interface not opened
5 = invalid address (pointer)

Transmit *Communication*

Transmits data

- Function block:



- Variables:

tx_buffer POINTER TO BYTE;
Start address of the memory area to be transmitted (variable, array)

length BYTE;
Number of characters to be transmitted (data length ≤ 255 characters)

error	BYTE; Error condition: 0 = no error 1 = parity error 2 = framing error 3 = overflow error 4 = interface not opened

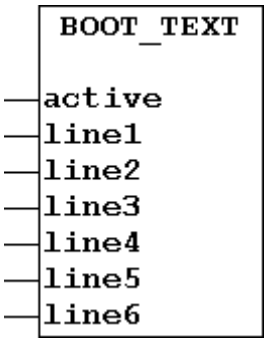
4.7 Display and Keyboard

Boot_Text *Display and Keyboard*

Stores the defined text block in the non volatile RAM to be displayed when booting next time

See also: Clr_Tscreen, Write_BStr, Write_Str

- Function block:



- Variables:

active BOOL;
Enables the text block:
TRUE = text block is displayed when energizing the device
FALSE = text block is not displayed when energizing the device but the factory default boot block containing graphic and text elements

line1, line2 STRING(20);
Lines 1 and 2 of the text block
These two lines can show up to 20 characters each using the big font (12x16 pixels).

line3…6 STRING(40);
Lines 3 to 6 of the text block
These four lines can show up to 40 characters each using the small font (6x8 pixels).

- Example:

Declaration:

```
Customer_Text: Boot_Text;
init: BOOL:=TRUE;
```

Program in ST:

```
IF init THEN
   Customer_Text.line1:='   LENORD + Bauer   ';
   Customer_Text.line2:='     MotionLine     ';
   Customer_Text.line3:='                                    ';
   Customer_Text.line4:='           Dohlenstrasse 32         ';
   Customer_Text.line5:='           46145 Oberhausen         ';
   Customer_Text.line6:='           +49 208 9963 0           ';
   Customer_Text(active:=TRUE);
```

```
   Init:=FALSE;
END_IF;
```

When executing this small program once, the string variables are stored retentively so that they will be displayed each time the device is powered on (even if the program is no longer residing in memory).

The boot process will take a few seconds. As soon as it is completed, the PLC program loaded starts running and may then alter the display for your special application.

Clr_GScreen / Clr_TScreen *Display and Keyboard*

Deletes all graphic / text elements in the display

See also: Boot_Text, Clr_Point; Line, Put_Point, Write_BGStr; Write_BStr; Write_Str

- Function block:

- Example:

  Declaration:

```
clr_grafic: Clr_GScreen;
clr_text: Clr_TScreen;
key_new: BYTE;
key_old: BYTE;
```

  Program in ST:

```
IF key_new AND key_old=FALSE THEN
   clr_grafic();
   clr_text();
END_IF;
key_old:=key_new;
```

  The FBs are called up once with the positive edge of `key_new` where the `Clr_GScreen` FB clears all elements created with `Line`, `Put_Point`, and `Write_BGStr` and the `Clr_TScreen` FB clears all texts created with `Boot_Text`, `Write_BStr` and `Write_Str`.

  The default boot screen contains graphic and text elements. So you have to call both FBs to clear the display completely, as shown in the example.

Clr_Point / Put_Point *Display and Keyboard*

Deletes / draws individual (graphic) dots on the display

See also: Clr_GScreen

- Function block:

- Variables:

  x BYTE;
  Column of the dot position: 0 (left) to 239 (right)

  y BYTE;
  Row of the dot position: 0 (top) to 63 (bottom)

- Example:

  Declaration:

```
clr_grafic: Clr_GScreen;
clr_txt: Clr_TScreen;
axis: Line; (* base line *)
display_line: BYTE;
i: BYTE;
c_point: Clr_Point;
p_point: Put_Point;
```

  Program in ST:

```
clr_grafic();
clr_txt();
axis(col:=1, x1:=0, y1:=31, x2:=239, y2:=31);
FOR i:=0 TO 239 DO
   display_line:=REAL_TO_BYTE(-31.*SIN(BYTE_TO_REAL(I)
                                                /38.))+31;
   IF display_line=31 THEN
      c_point(x:=i, y:=display_line);
   ELSE
      p_point(x:=i, y:=display_line);
   END_IF
END_FOR;
```

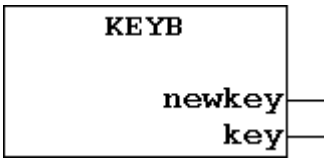
  One program cycle draws a sinusoid on the display. Curve dots placed on the base line are cleared.

Keyb *Display and Keyboard*

Issues one byte for a key pressed down and messages a change in state of the keys for one PLC cycle

See also: Keyb_Val

- Function block:



- Variables:

newkey BOOL;
State message related to the previous PLC cycle:
FALSE = no change in key state
TRUE = key state changed

key BYTE;
Key value corresponding to the following table:

Key										
Value	1	2	3	4						
Key	F1	F2	F3	F4	F5		M1	M2	M3	M4
Value	17	18	19	20	21		28	29	30	31
Key	0	1	2	3	4	5	6	7	8	9
Value	48	49	50	51	52	53	54	55	56	57
Key	+/−	.	⇨	Enter	ESC	///	◀	▶	▲	▼
Value	45	46	13	6	27	127	8	9	11	22

Die grayed fields are assigned four keys which are arranged invisibly behind the logo field in the same columns like the F1 to F4 keys.
The value 255 is returned if several keys are pressed simultaneously.

- Example:

Declaration:
```
keyboard_byte: Keyb;
counter: DINT;
```

Program in ST:
```
keyboard_byte();
IF keyboard_byte.newkey AND keyboard_byte.key=255 THEN
   counter:= counter + 1;
END_IF;
```

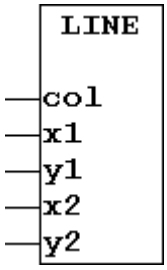
The `counter` variable counts all events where at least two key have been pressed simultaneously.

Line *Display and Keyboard*

Draws or deletes a line

See also: Clr_GScreen, Clr_Point; Put_Point

- Function block:



- Variables:

col BYTE;
Mode:
0 = delete
1 = draw

x1 INT;
Column of the line start: 0 (left) to 239 (right)

y1 INT;
Row of the line start: 0 (top) to 63 (bottom)

x2 INT;
Column of the line end: 0 (left) to 239 (right)

y2 INT;
Row of the line end: 0 (top) to 63 (bottom)

- Example:

Declaration:

```
FLine: Line;
```

Program in ST:

```
FLine(col:=1, x1:=0, y1:=47, x2:=239, y2:=47);
FLine(col:=1, x1:=0, y1:=48, x2:=0, y2:=63);
FLine(col:=1, x1:=47, y1:=48, x2:=47, y2:=63);
FLine(col:=1, x1:=95, y1:=48, x2:=95, y2:=63);
FLine(col:=1, x1:=143, y1:=48, x2:=143, y2:=63);
FLine(col:=1, x1:=191, y1:=48, x2:=191, y2:=63);
FLine(col:=1, x1:=239, y1:=48, x2:=239, y2:=63);
```

This program part draws caption fields for the function keys F1 to F5 in a single pass.

Set_TP / Write_Str *Display and Keyboard*

Specifies the starting point for the consecutive text output with Write_Str / writes a string in text mode using the small font (6x8 pixels) from the position specified with Set_TP

See also: Clr_TScreen, Write_BStr

- Function block:

- Variables:

x BYTE;
Column of the text start: 0 (left) to 39 (right)

y BYTE;
Row of the text start: 0 (top) to 7 (bottom)

_string STRING[160];
String to be displayed
If a string exceeds the end of a row it will be continued in the next row. Characters that are written beyond the end of the last row are not visible.

- Example:

Declaration:

```
Text_Pointer: Set_TP;
Text: Write_Str;
txt_str: STRING[8];
txt_length: BYTE;
column: BYTE;
Fkey_loc: BYTE;
```

Program in ST:

```
Fkey_loc:=4;
txt_str:='Stop';
txt_length:=INT_TO_BYTE(LEN(txt_str));
column:=Fkey_loc * 8 - 4 - txt_length/2;
Text_Pointer(x:=column, y:=7);
Text(_string:=txt_str);
```

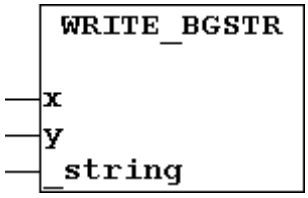
This program part writes the text "Stop" above the F4 function key.

Write_BGStr *Display and Keyboard*

Writes a string from the specified starting position in graphics mode (slower than text mode) using the big font (12x16 pixels)

See also: Clr_GScreen, Write_BStr, Write_Str

- Function block:



- Variables:

x BYTE;
Column of the text start position: 0 (left) to 38 (right)
The raster for the column position is determined by the half character width of 6 pixels. Up to 20 characters may be displayed in one row.

y BYTE;
Row position of the text start: 0 (top) to 6 (bottom)
The raster for the row position is determined by the half character height of 8 pixels. Up to 4 characters may be placed one under another in a column.

_string STRING[80];
String to be displayed
If the string exceeds the end of a row then display is continued at the left border but shifted down by half a character height.
Characters that are written beyond the end of the last row are shown with its upper half at the bottom border of the display screen.

- Example:

Declaration:

```
clr_grafic: Clr_GScreen;
clr_txt: Clr_TScreen;
column: BYTE;
Text_BG: Write_BGStr;
txt_length: BYTE;
txt_str: STRING[20];
```

Program in ST:

```
clr_grafic();
clr_txt();
txt_str:='Lenord + Bauer';
txt_length:=INT_TO_BYTE(LEN(txt_str));
```

```
column:=20 - txt_length;
Text_BG(x:=column, y:=0, _string:=txt_str);
```

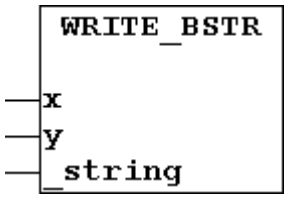
This program part writes the text "Lenord + Bauer" centred in the upper row of the display screen.

Write_BStr *Display and Keyboard*

Writes a string from the specified starting position in text mode (faster than graphics mode) using the big font (12x16 pixels). The character set is reduced to

See also: Clr_TScreen, Write_BGStr, Write_Str

- Function block:



- Variables: see Write_BGStr
- Example:

Declaration:

```
fix_dint: DINT;
fix_str: STRING(20);
frac_dint: DINT;
frac_len: INT;
frac_str: STRING(20);
IN_dint: DINT;
length: BYTE;
mult: DINT;
OUT_len: BYTE;
OUT_str: STRING(20);
stat_axis: Rd_Status_Axis;
Text_B: Write_BStr;
```

Program in ST:

```
stat_axis(axis:=1);
IN_dint:=stat_axis.act_slave_pos;
length:=6;
frac_len:=2;
IF frac_len>0 THEN
   mult:=REAL_TO_DINT(EXPT(10,frac_len));
   fix_dint:=IN_dint / mult;
```

```
   fix_str:=DINT_TO_STRING(fix_dint);
   frac_dint:=IN_dint MOD mult;
   frac_str:=DINT_TO_STRING(ABS(frac_dint));
   IF frac_dint<0 AND fix_dint=0 THEN
      fix_str:=CONCAT('-',fix_str);
   END_IF;
   frac_str:=RIGHT(CONCAT('0000',frac_str),frac_len);
   OUT_str:=CONCAT(CONCAT(fix_str,'.'),frac_str);
ELSE;
   OUT_str:=DINT_TO_STRING(IN_dint);
END_IF;
OUT_len:=INT_TO_BYTE(LEN(OUT_str));
IF OUT_len>length THEN
   OUT_str:=LEFT(OUT_str,length);
ELSE;
   OUT_str:=RIGHT(CONCAT('          ',OUT_str),length);
END_IF;
Text_B(x:=(20-length)*2, y:=2, _string:=OUT_str);
```

This program writes the actual value of axis 1 on the display, right-aligned with two decimals and a length of 6 characters (decimal point and sign included). If the value consists of more than 6 characters, only the left 6 high-order characters will be displayed.

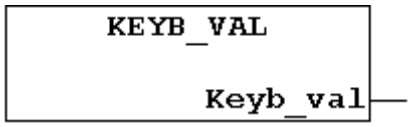
4.7.1 Basic Jobs

Keyb_Val *Display and Keyboard (Basic Jobs)*

Informs about keys pressed down according to their arrangement in the key matrix

See also: Keyb

- Function block:



- Variables:

Keyb_val ARRAY[0..4] OF BYTE;

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Keyb_val[0]					–	–	–	–
Keyb_val[1]	·	3	2	1	⇨	▼	M2	M1
Keyb_val[2]	+/–	6	5	4	ESC	▶	Enter	◀
Keyb_val[3]	0	9	8	7	//	–	–	▲
Keyb_val[4]	F4	F3	F2	F1	F5	–	M3	M4

Die grayed fields are assigned four keys which are arranged invisibly behind the logo field in the same columns like the F1 to F4 keys.
Pressing several keys simultaneously will result in a wrong image.

- Example:

Declaration:

```
keyboard: Keyb_val;
keyb_val_long: UDINT;
keyb_val_long_old: UDINT;
key_puls: UDINT;
key_puls_0: UDINT;
key_changed: BOOL;
counter1: DINT;
counter2: DINT;
```

Program in ST:

```
keyboard();
keyb_val_long:=keyboard.Keyb_val[4];
keyb_val_long:=keyb_val_long * 16#100 + keyboard.Keyb_val[3];
```

```
keyb_val_long:=keyb_val_long * 16#100 + keyboard.Keyb_val[2];
keyb_val_long:=keyb_val_long * 16#100 + keyboard.Keyb_val[1];
IF keyb_val_long_old<>keyb_val_long THEN
   key_puls:=keyb_val_long;
   key_changed:=TRUE;
   IF keyb_val_long_old=0 THEN
      key_puls_0:=keyb_val_long;
   ELSE;
      key_puls_0:=0;
   END_IF;
ELSE;
   key_puls:=0;
   key_puls_0:=0;
   key_changed:=FALSE;
END_IF;
keyb_val_long_old:=keyb_val_long;
CASE key_puls OF
   16#10000004:
     counter1:=counter1 - 1;
   16#10010000:
     counter1:=counter1 + 1;
   16#20000004:
     counter2:=counter2 - 1;
   16#20010000:
     counter2:=counter2 + 1;
END_CASE
```

At first keys are queried by calling the FB. Then all visible keys are merged in the `keyb_val_long` variable. When the key state has changed, the `key_puls` variable provides the key state and the `key_changed` variable reports TRUE for one program cycle each. The `key_puls_0` variable only provides the key state if no key was pressed before.

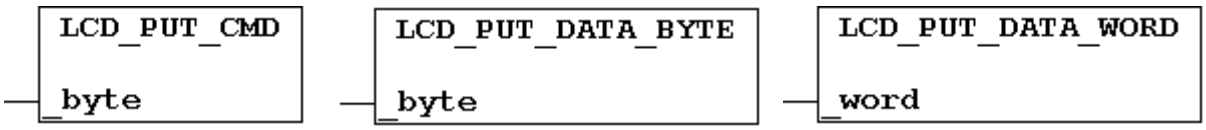
The `counter1` variable is incremented by pressing F1+▲ and decremented by pressing F1+▼. The `counter2` variable is changed in the same way by means of F2+▲/▼.

Lcd_Put_Cmd / Lcd_Put_Data_Byte / Lcd_Put_Data_Word

Display and Keyboard (Basic Jobs)

Direct control of the 240x64 display via the T6963C Toshiba controller

- Function block:



These FBs enable the experienced operator to make use of nearly all the display functions the manufacturer specifies in his data sheet.

4.8 Field bus

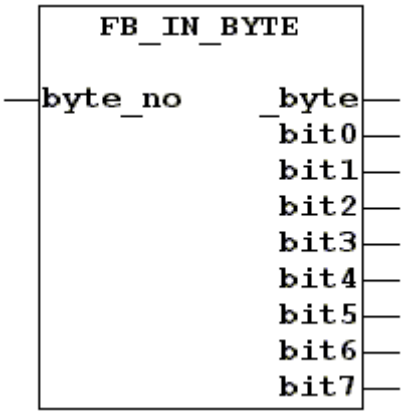
The FBs in this branch permit reading/writing of data via a field bus module installed on the MotionPLC (PROFIBUS-DP, InterBus-S, DeviceNet or Ethernet). The data format and data size are specified in the field bus system parameters (see Operating Instructions).

Ethernet module:
For the FBs listed in this section the Modbus server is used: Port 502. The data format and size are to be specified in the parameters for the DeviceNet (para[321+322]).
The FBs in subsection 4.8.1 make use of the FTP server in the module. The necessary IP address must be specified according to your own needs (see the field bus module documentation).

FB_In_Byte *FieldBus*

Reads a data byte

- Function block:



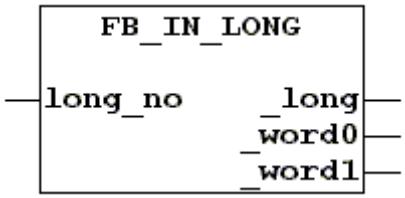
- Variables:

byte_no	WORD; Specifies the data byte: 0 … max
_byte	BYTE; Content of the data byte
bit0…7	BOOL; States of the bits 0…7 of *_byte*

FB_In_Long *FieldBus*

Reads a data Long

- Function block:



- Variables:

long_no	WORD; Specifies the data Long: 0 … max
_long	DINT; Content of the data Long
word0	WORD; Content of the least significant Word (LSW) of *_long*
word1	WORD; Content of the most significant Word (MSW) of *_long*

FB_Out_Bit *FieldBus*

Writes data bit

- Function block:

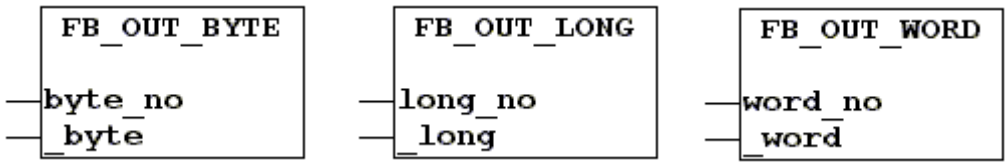
- Variables:

byte_no	WORD; Specifies the data byte: 0 … max
bit_no	BYTE; Bit position in the data byte
bit	BOOL; Logic state of the data bit

FB_Out_Byte / FB_Out_Long / FB_Out_Word *FieldBus*

Writes a data byte/Long/Word

- Function block:



- Variables:

byte_no, *long_no*, *word_no* WORD;
Specifies the data byte/Long/Word: 0 … max

_byte BYTE;
Content of the specified data byte

_long *DINT;*
Content of the specified data Long

_word *WORD;*
Content of the specified data Word

4.8.1 Ethernet

With the FBs listed here file operations can be executed on the FTP server of the module. Files may be temporarily stored in the RAM ("\RAM_DISC" folder with a capacity of 1 Mbytes). In addition, there is a flash memory of 1.4 Mbytes where you can create any directories you like (e.g. using Telnet or a FTP client like the Internet Explorer) and save files retentively.

Del_File *FieldBus (Ethernet)*

Deletes a file in the module's flash memory

- Function block:

- Variables:

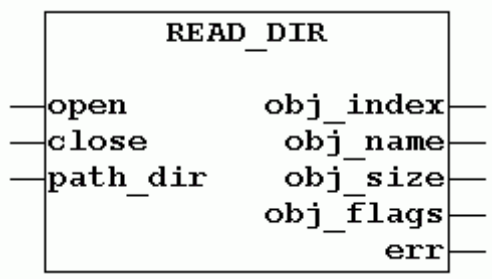
file_name STRING[255];
Path\name of the file to be deleted

err BYTE;
Error number:
0 = no error
≠0 = error

Read_Dir *FieldBus (Ethernet)*

Reads a directory in the module

- Function block:



- Variables:

open BOOL;
When calling the FB first time you must set this variable to TRUE in order to open the directory; it should be FALSE for further calls.

close BOOL;
When aborting the read operation this variable should be set to TRUE in order to close the directory; if there is no further entry (*obj_index* = 0) this is done automatically.

path_dir STRING[255];
Complete directory path (path\directory_name)

obj_index WORD;
Number of the file entry in the directory; it is incremented with any further FB call unless there is no further entry (*obj_index* = 0); even an empty subdirectory contains the two system entries "**.**" (*obj_index* = 1) and "**..**" (*obj_index* = 2)

obj_name STRING[255];
File or directory name

obj_size *DWORD*;
File size in bytes

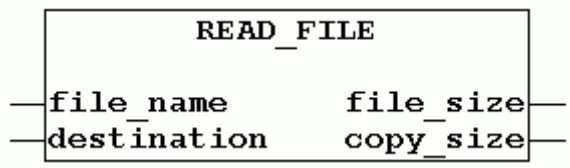
obj_flags BYTE;
File attributes:
Bit 0 = directory
Bit 1 = write protected
Bit 2 = hidden
Bit 3 = system file
Bit 4…7 = reserved

err BYTE;
Error number:
0 = no error
1/2 = directory could not be opened/closed
3/4 = directory is already open/closed

Read_File *FieldBus (Ethernet)*

Loads a file from the module into the RAM of the MotionPLC

- Function block:



- Variables:

file_name STRING[255];
Path\name of the file to be read

destination WORD;
Target address (offset) in the RAM of the MotionPLC
(0…FFFFh = 64 Kbytes)

file_size *DWORD*;
File size in bytes

copy_size *DWORD*;
Number of bytes actually written (check for range crossing)

Write_File *FieldBus (Ethernet)*

Copies a file from the RAM of the MotionPLC to the module, dynamically to the \RAM_DISC or statically to the flash memory

- Function block:

- Variables:

file_name STRING[255];
Path\name of the file to be created in the module

source WORD;
Source address (offset) in the RAM of the MotionPLC
(0…FFFFh = 64 Kbytes)

copy_size *DWORD*;
Number of bytes to be transferred

4.9 Input / Output

With the FBs in this branch you can execute the following functions:

- Read analog inputs (Ana_In)
- Read/write analog outputs (Rd/Wr_Ana_Out)
- Read digital inputs (Dig_In_Byte)
- Read/write digital outputs (Rd_Dig_Out_Byte, Dig_Out_Bit)

! The FBs access the hardware directly. This means that read and write operations take place as soon as the FB is being executed (with maximum delay of 300 µs) and that they – as usual in PLC programming – do not take place completely for all inputs and outputs at the beginning or at the end of the program cycle. If this is desired, the **% variables** must be used instead of the FBs (see CoDeSys help). If a program is to make use of both possibilities, it must be ensured that they are not applied simultaneously to one output since this might otherwise result in unexpected switching conditions (the statuses returned by servo amplifier, cam plate, FB and % variable are ORed).

%-Variables:

The number of variables is factory-limited to the following values and should not be changed:

inputs (%I…):	128
outputs (%Q…):	128
flags (%M…):	4096

The variables are assigned to the respective inputs and outputs as follows:

- Digital inputs (terminal blocks I1 to I4):

Terminal	Variable	
	Bit	Word
I1.0	%IX1.0	%IW1
I1.1	%IX1.1	
I1.2	%IX1.2	
I1.3	%IX1.3	
I1.4	%IX1.4	
I1.5	%IX1.5	
–	–	
–	–	
I2.0	%IX2.0	%IW2
I2.1	%IX2.1	
I2.2	%IX2.2	
I2.3	%IX2.3	
I2.4	%IX2.4	
I2.5	%IX2.5	
I2.6	%IX2.6	
I2.7	%IX2.7	

Terminal	Variable	
	Bit	Word
I3.0	%IX3.0	%IW3
I3.1	%IX3.1	
I3.2	%IX3.2	
I3.3	%IX3.3	
I3.4	%IX3.4	
I3.5	%IX3.5	
I3.6	%IX3.6	
I3.7	%IX3.7	
I4.0	%IX4.0	%IW4
I4.1	%IX4.1	
I4.2	%IX4.2	
I4.3	%IX4.3	
I4.4	%IX4.4	
I4.5	%IX4.5	
I4.6	%IX4.6	
I4.7	%IX4.7	

- Analog inputs (terminal blocks 5 and I6):

Terminal	Variable (Word)
I5.4+/-	%IW54
I5.5+/-	%IW55
I5.6+/-	%IW56
I5.7+/-	%IW57

Terminal	Variable (Word)
I6.1+/-	%IW61
I6.2+/-	%IW62
I6.3+/-	%IW63
–	–

▶ Digital and analog outputs (terminal blocks Q1 to Q3):

Terminal	Variable	
	Bit	Word
Q1.0	%QX1.0	%QW1
Q1.1	%QX1.1	
Q1.2	%QX1.2	
Q1.3	%QX1.3	
Q1.4	%QX1.4	
Q1.A+/-	–	%QW10
Q2.0	%QX2.0	%QW2
Q2.1	%QX2.1	
Q2.2	%QX2.2	
Q2.3	%QX2.3	
Q2.4	%QX2.4	
Q3.A+/-	–	%QW30

Terminal	Variable	
	Bit	Word
Q3.0	%QX3.0	%QW3
Q3.1	%QX3.1	
Q3.2	%QX3.2	
Q3.3	%QX3.3	
Q3.4	%QX3.4	
Q2.A+/-	–	%QW20

▶ Encoder I/O (terminal blocks E1 to E3):

Terminal	Variable (Bit)
E1.A	%IX1.8
E1.B	%IX1.9
E1.N	%IX1.10
E1.C	%QX0.0
E2.A	%IX2.8
E2.B	%IX2.9
E2.N	%IX2.10
E2.C	%QX0.1

Terminal	Variable (Bit)
E3.A	%IX3.8
E3.B	%IX3.9
E3.N	%IX3.10
E3.C	%QX0.2

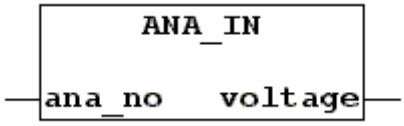
This assignment can also be read in the *Resources* window of CoDeSys under *PLC Configuration*; if not, the corresponding configuration file must be linked (in logged out mode, click the right mouse button inside the window).

Ana_In *Input / Output*

Returns the value measured on a certain analog input (for the value ranges see the Connections chapter in the Operating Instructions)

See also: Rd_Ana_Out, Wr_Ana_Out

- Function block:



- Variables:

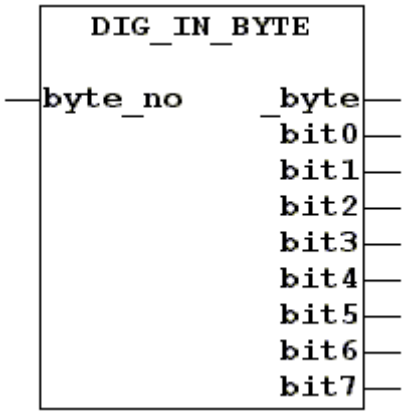
ana_no BYTE;
Specifies the analog input:
1…3 = current/voltage input 1…3 on terminal block I6
4…7 = PT100 input 4…7 on terminal block I5

voltage INT;
Converted digital value:
0…1023 for *ana_no* = 1…3
0…1564 for *ana_no* = 4…7

Dig_In_Byte *Input / Output*

Returns the statuses of specific digital input terminals

- Function block:



- Variables:

byte_no BYTE;
Specifies the digital inputs:
1…4 = input terminal blocks I1…4
10, 20, 30 = encoder input terminal blocks E1, E2, E3

_byte BYTE;
Status byte of the specified input terminals

bit0…7 BOOL;
Logic state of a specific digital input; High = 1 (TRUE), Low = 0 (FALSE):

bit	I1…4	E1…3
0	Ix.0	Ex.A
1	Ix.1	Ex.B
2	Ix.2	Ex.N
3	Ix.3	0
4	Ix.4	0
5	Ix.5	0
6	Ix.6*	0
7	Ix.7*	0

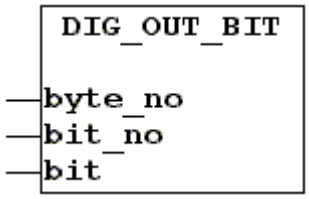
* always 0 for I1

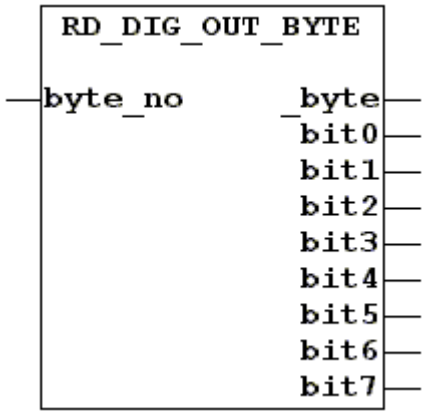
Dig_Out_Bit / Rd_Dig_Out_Byte *Input / Output*

Sets / returns the status at a specific digital output

See also: Dig_In_Byte

- Function block:





- Variables:

byte_no BYTE;
Specifies the digital outputs:
0 = clock output (C, /C) on terminal blocks E1…3 (precondition: no SSI encoder activated)
1…3 = terminal blocks Q1…3

bit_no BYTE;
Specifies the terminal:

bit	Q1…3	E
0	Qx.0	E1.C
1	Qx.1	E2.C
2	Qx.2	E3.C
3	Qx.3	0
4	Qx.4	0
5	0	0
6	0	0
7	0	0

bit BOOL;
Logic state to be output; High = 1 (TRUE), Low = 0 (FALSE)

_byte BYTE;
Status byte of the specified output terminals

bit0…7 BOOL;
Status of a specific digital output; High = 1 (TRUE), Low = 0 (FALSE); assignment as for *bit_no*

Rd_Ana_Out / Wr_Ana_Out *Input / Output*

Returns / writes a voltage value of / to a specific analog output (writing only if the output is not occupied by an analog axis)

See also: Ana_Out

- Function block:

- Variables:

ana_no BYTE;
Specifies the analog output: 1…3 = terminal block Q1…3, (terminals A+/-)

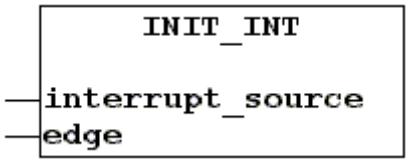
voltage INT;
Voltage value in mV

4.10 Interrupt

Init_Int *Interrupt*

Enables a specific input on terminal blocks I1 or E1 to E3 for interrupts. On detection of an interrupt, the current actual positions of the master and slave axes are latched.

- Function block:



- Variables:

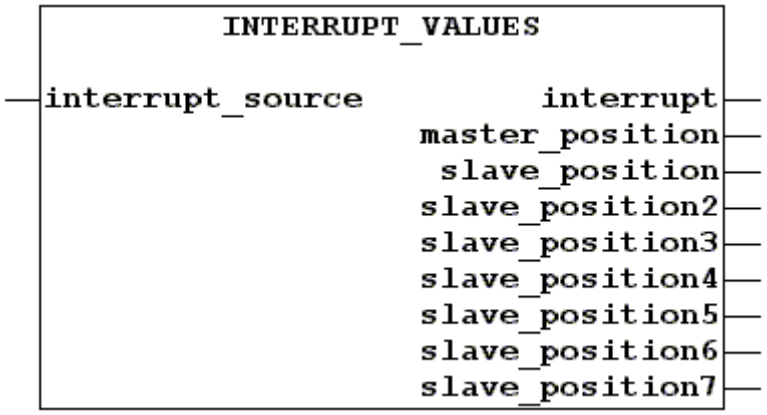
interrupt_source BYTE;
Interrupt input:
0…5 = digital input I1.0…5
6…8 = encoder reference input E1.N…E3.N

edge BYTE;
Evaluation of the signal edge:
0 = inactive
1 = rising edge
2 = trailing edge
3 = both edges

Interrupt_Values *Interrupt*

Returns the interrupt status and the actual positions of the master and slave axes at the time of the interrupt event

- Function block:



- Variables:

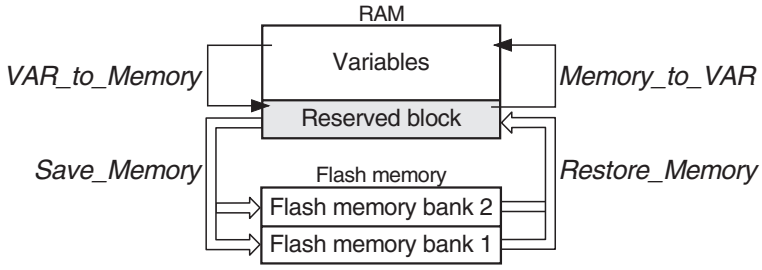
interrupt_source	BYTE; Interrupt input: 0…5 = digital input I1.0…5 6…8 = encoder reference input E1.N…E3.N
interrupt	BOOL; Interrupt status (after calling this FB the variable is reset): 0 = no interrupt detected 1 = interrupt occurred
master_position	DINT; Actual position of the master axis
slave_position…7	DINT; Actual position of the slave axes 1…7

4.11 Non-volatile memory

The FBs in this branch can be used to saving certain data from the RAM to the flash memory (where it is safe against power failures) and writing these data back (see also section 3).

The following functions are available:

- Copy variable data from normal main memory to a special 64K RAM area and reload (VAR_to_Memory, Memory_to_VAR)
- Write 64K data block of the special RAM area in the flash memory and read back (Save_Memory, Restore_Memory)

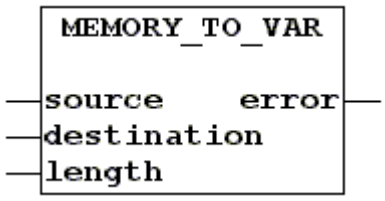


Memory_to_VAR *Non volatile memory*

Replaces variable(s) in main memory with value(s) from the reserved RAM buffer, which has normally been filled by means of Restore_Memory with the data from a flash bank

See also: Restore_Memory

◆ Function block:



◆ Variables:

source	WORD; Address (offset) in reserved 64K RAM area, where the data to be read are located: 0…FFFFh
destination	POINTER TO BYTE; Start address of the variables to be replaced in the main memory
length	WORD; Number of bytes to be transmitted

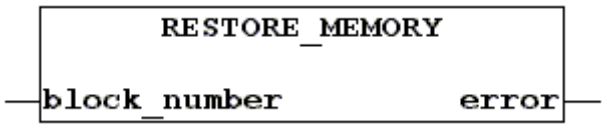
error BOOL;
Error condition:
0 = no error
1 = memory overrun: *source* + *length* > FFFFh

Restore_Memory *Non volatile memory*

Replaces the 64K data block reserved in the RAM with the data from one of the two flash banks. This FB is usually called before executing Memory_-to_VAR.

See also: Memory_to_VAR

◆ Function block:



◆ Variables:

block_number BYTE;
1 = flash bank 1
2 = flash bank 2

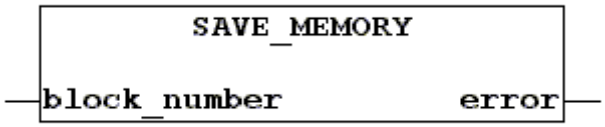
error BYTE;
Error condition:
0 = no error
2 = invalid value for *block_number*

Save_Memory *Non volatile memory*

Replaces the data in one of the two flash banks with the data from the reserved 64K block in the RAM. This operation will take a few seconds (execution of the PLC program is suspended during this time). This FB is usually called after executing VAR_to_Memory (see below).

See also: VAR_to_Memory

◆ Function block:



◆ Variables:

block_number BYTE;
1 = flash bank 1
2 = flash bank 2

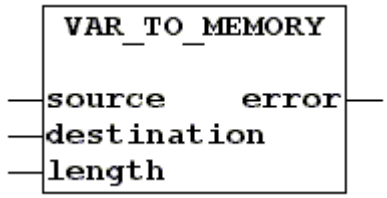
error BYTE;
Error condition:
0 = no error
1 = flash error (write operation not executed)
2 = invalid value for *block_number*

VAR_to_Memory *Non volatile memory*

Copies variable(s) from the variables area of the main memory to the reserved RAM block (to be saved from here to the flash memory by means of Save_-Memory)

See also: Save_Memory

◆ Function block:



◆ Variables:

source POINTER TO BYTE;
Start address of the variables to be copied in the main memory

destination WORD;
Address (offset) in reserved 64K RAM area, where the variable is to be stored: 0…FFFFh

length WORD;
Number of bytes to be written: ≤ FFFFh - *destination*

error BOOL;
Error condition:
0 = no error
1 = memory overrun: *destination* + *length* > FFFFh

4.12 Parameter

Loader *Parameter*

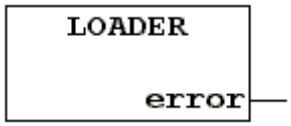
Checks modified parameters and activates them if there is no error

Some parameters will become immediately active i.e. without calling the loader; others however require switching off and on again the MotionPLC (see the Operating Instructions).

A CAN init is executed, too (see FB in section 4.4).

This FB should not be called up cyclically in a program, but only if needed when one or two parameters have been changed. Otherwise, the program is slowed down unnecessarily. Moreover, all parameters in the RAM are replaced by those from the flash. This is not always desired and may result in faults and errors not easily detectable.

- Function block:



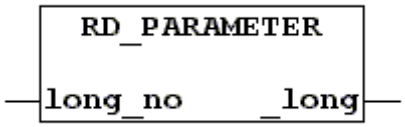
- Variables:

error INT;
0 = no error ⇒ parameters accepted; else number of the faulty parameter

Rd_Parameter *Parameter*

Returns a parameter value.

- Function block:



- Variables:

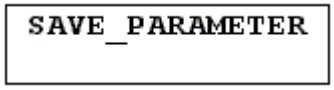
long_no WORD;
Number of parameter to be queried: 000…999 (000…499 = operating system parameters – see Operating Instructions – and 500…999 = for free use – e.g. by the PLC program)

_long DINT;
Content of the associated memory location (in RAM)

Save_Parameter *Parameter*

Copies parameters and curve data from the RAM to the non-volatile flash memory. During this data transfer, execution of the PLC program is suspended.

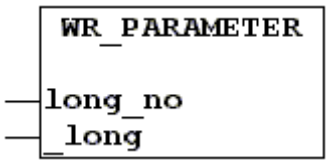
- Function block:



Wr_Parameter *Parameter*

Copies a parameter to the RAM of the MotionPLC. If the operating system is to make use of the new value or modified property of the parameter, it is necessary to first execute the Loader FB (not valid for parameters which become immediately active, see Operating Instructions; this also applies to the master correction parameters: para[343].../[463].../[263]... ⇒ modification causes an engaging operation if a curve has already been started).

- Function block:



- Variables:

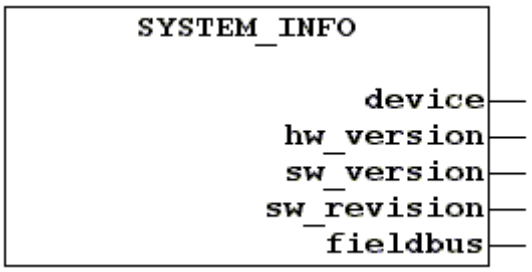
long_no	WORD; Number of parameter to be queried: 000…999 (000…499 = operating system parameters – see Operating Instructions – and 500…999 = for free use – e.g. by the PLC program)
_long	DINT; Value of the parameter

4.13 System

System_info *System*

Provides device information

- Function block:



- Variables:

device BYTE;
0 = Motion Card LD100
1 = MotionPLC GEL 8240
2 = MotionPLC GEL 8245

hw_version BYTE;
Hardware version number (e.g. 3 for HW3)

sw_version BYTE;
Firmware version number (e.g. 1 for SW1.30)

sw_revision BYTE;
Firmware revision number (e.g. 30 for SW1.30)

fieldbus WORD;
Field bus type:
100h = PROFIBUS-DP
1000h = InterBus
2500h = DeviceNet
8200h = Ethernet

Time_On *System*

Returns the cumulated on time

- Function block:

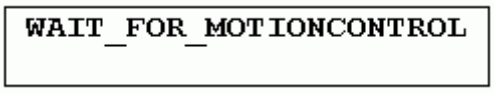
◆ Variables:

_time TIME;
Time since last powering on
The measuring time is limited to 49 days 17 hours 2 minutes 47 seconds 295 milliseconds; after this time the timer is reset to zero.

Wait_for_MotionControl *System*

This FB suspends execution of the PLC program until a motion control cycle has passed. You may use the function when changing control parameters and immediately after that querying the status of the axis to check if the change already shows the desired effect. This only makes sense when using a single slave axis or when working with an alternative cycle time (para[211] ≠ 0).

◆ Function block:



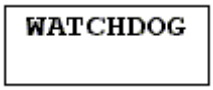
Watchdog *System*

The watchdog function of the operating system prevents the axis control from being blocked by an error occurring in the PLC program (infinite loops etc.).

For that purpose, the operating system operates a so-called watchdog every 400 ms. If this operation fails to appear – because the PLC program claims the processor for a time longer than approx. 400 ms – a reset is triggered. This results in fault F20 being displayed in the LD 2000 servo amplifier.

The FB also enables the PLC program to operate the watchdog. However, this should be only done in exceptional cases, e.g., if the (one-time) calculation of many curves in an interpolation will result in a cycle time longer than 400 ms. But the FB must not be called periodically in the program sequence, because otherwise the safety aspect would be undermined.

◆ Function block:



Index

(Terms printed in *italics* identify function blocks.)