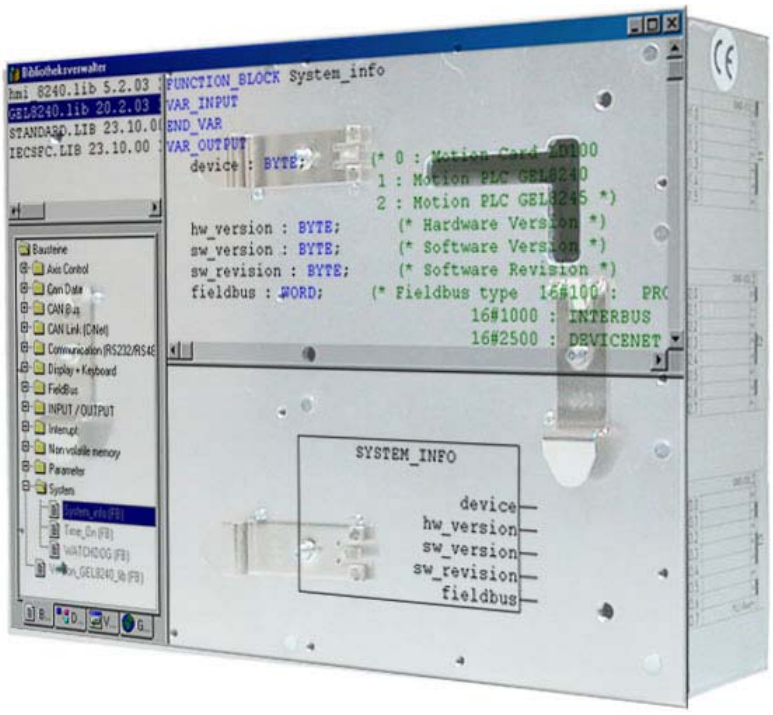
MOTIONLINE

LENORD+BAUER

MotionPLC

GEL 8240 / 8241 / 8245 / 8246

Funktionsbibliothek GEL8240.lib



Referenzhandbuch
Ausgabe: 2004-07

Bisher erschienene Ausgaben:

Ausgabe	Bemerkung
2004-07	Erstausgabe gültig für Firmware Version ≥1.30, Bibiothek-Version 1.08

Herausgeber:

MOTIONLINE
LENORD+BAUER

Lenord, Bauer & Co. GmbH
Dohlenstraße 32
46145 Oberhausen
Fon: 0208 9963-0 • Fax: 0208 676292
Internet: http://www.lenord.de • E-Mail: info@lenord.de

Mat.-Nr. DS11-8240/8245R

Inhalt

1 Allgemeines

Die folgende Beschreibung behandelt die Bibliothek GEL8240.lib, die für die Erstellung von SPS-Programmen nach IEC 61131-3 mit der PC-Programmierumgebung **CoDeSys** benötigt wird. Die anderen mitgelieferten Bibliotheken werden hier nicht behandelt; es wird dazu auf die eingetragenen Kommentare bei den einzelnen Funktionsblöcken verwiesen.

Das vorliegende Referenzhandbuch dient als Ergänzung zum CoDeSys-Handbuch.

Die Handbücher werden in elektronischer Form als PDF-Datei auf der Installations-CD zur MotionPLC mitgeliefert. Der zum Lesen der Dateien benötigte *Acrobat Reader* von ADOBE SYSTEMS INCORPORATED kann ebenfalls von der CD installiert werden.

Es werden folgende **Kenntnisse** vorausgesetzt:

- Umgang mit dem WINDOWS-Betriebssystem
- Bedienung der Programmierumgebung CoDeSys
- Programmierung nach IEC 61131-3
- Bedienung und Wirkungsweise der MotionPLC GEL 824x

Verwendete Symbole und Bezeichnungen:

ℹ	kennzeichnet Absätze, die wichtige Zusatzinformationen zum Thema liefern
!	kennzeichnet Absätze, die wichtige Aussagen für den ordnungsgemäßen Betrieb enthalten
para[123]	bezeichnet einen programmierbaren Systemparameter (Beschreibung in der Betriebsanleitung der MotionPLC).
FB	ist die Abkürzung für Funktionsblock (mehrere: FBs)

ℹ **CAN-Achsen**:

Ab der Betriebsversion 1.30 können statt der bis dahin max. 4 möglichen CAN-Achsen 4 weitere CAN-Achsen angesteuert werden (Knotennummer 5...8). Damit erhöht sich die Gesamtzahl der Slave-Achsen von 7 auf 11 (3x analog + 8x CAN).

Um diese Möglichkeit nutzen zu können, müssen die beiden CAN-Schnittstellen an Klemmleiste C2 parallel geschaltet werden, so dass nur noch 1 CAN-Bus vorhanden ist. In diesem Fall gelten folgende Bedingungen:

- Die Anzahl der möglichen CAN-I/O-Module (Standard-CAN-Objekte) verringert sich abhängig von den verwendeten Achsen auf
  4 - (Anzahl CAN-Achsen - 4),
  also z. B. bei 6 CAN-Achsen auf 2 (3. und 4. CAN-Objekt).

- Die Knoten-Adressen der CAN-I/O-Module müssen über denen der CAN-Achsen liegen.
- Die MotionPLC muss als CAN-Master konfiguriert sein (para[50]=0).
- Eine Königswelle darf nicht aktiviert sein (para[125]=0).
- Bei bestimmten Zykluszeit/Achsenzahl-Kombinationen wird die CAN-Bus-Baudrate automatisch von 500 kBaud auf 1 MBaud erhöht:
  – 2 ms / 4 CAN-Achsen (getrennte CAN-Busse, wie vorher)
  – 3 ms / 5 oder 6 CAN-Achsen
  – 4 ms / 5...8 CAN-Achsen
- Die Baudrate für die serielle Kommunikation (BB2100K, CoDeSys) sollte nicht höher als 38400 eingestellt sein, wenn mehr als 4 externe Achsen am CAN-Bus betrieben werden.

Für die zusätzlichen Achsen gelten die gleiche Funktionalität und die gleichen CoDeSys-Funktionsblöcke wie für die ersten 4 CAN-Achsen.

2 Verwendung

Die Bibliothek stellt vordefinierte FBs für verschiedene Anwendungsbereiche zur Verfügung (eine Übersicht liefert das Inhaltsverzeichnis).

Für die Nutzung der FBs muss bei der Erstellung eines neuen Programms die Bibliothek eingebunden werden (CoDeSys: Menü *Fenster/Bibliotheksverwaltung*, im Fenster rechte Maustaste *Weitere Bibliothek* ...).

Nach Erstellung des Programms und erfolgreicher Kompilierung in CoDeSys wird das Programm mit dem Befehl *Online/Einloggen* über die serielle Schnittstelle in die MotionPLC übertragen und kann dort ausgeführt bzw. getestet werden (dazu muss der Menüpunkt *Online/Simulation* inaktiv sein).

Das Programm befindet sich zunächst nur im RAM der MotionPLC und sollte nach erfolgreichem Test in den stromausfallsicheren Flash-Speicher übertragen werden (CoDeSys: Menü *Online/Bootprojekt erzeugen*), von wo es beim nächsten Einschalten wieder in das RAM geladen und dann abgearbeitet wird.

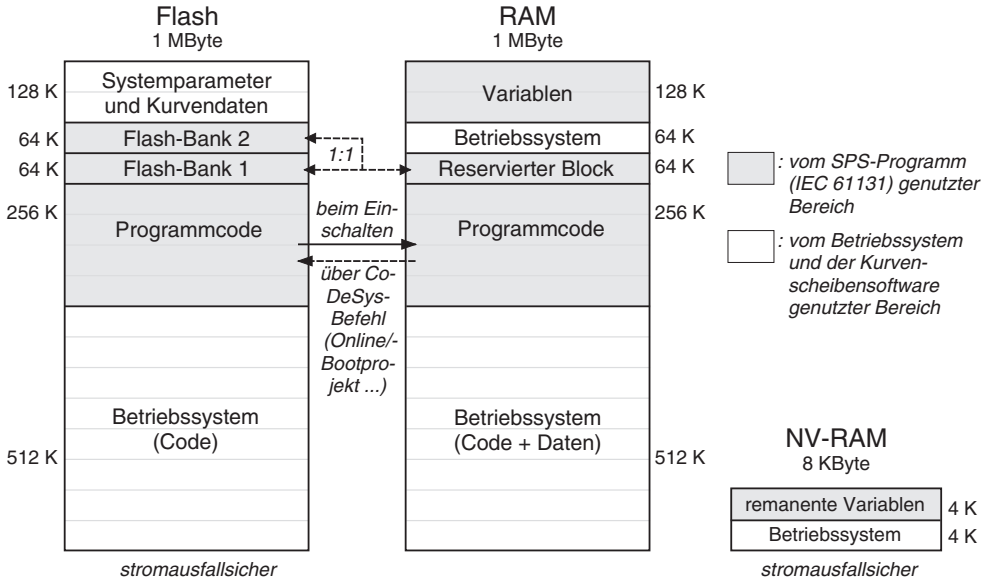
Während der Übertragung der Daten in den Flash-Speicher wird die Ausführung des SPS-Programms unterbrochen.

Für den Start des Programms über CoDeSys muss die SPS aktiviert sein d. h. der Eingang I3.7 (PLC RUN) auf High-Pegel liegen (Low = Stopp) .

Wird ein Programm aus CoDeSys heraus gestoppt, kann es auch nur von da aus wieder gestartet werden, es sei denn, man schaltet die MotionPLC aus und wieder ein.

3 Speicherorganisation

Für die Programmierung nach IEC 61131 werden bestimmte Speicherbereiche im RAM, Flash und NV-RAM genutzt. Dazu folgende Übersicht:



3.1 Flash

Es stehen zwei 64K-Speicherbänke für die Sicherung eines bestimmtem RAM-Bereichs (s. u.) zur Verfügung; die zugehörigen FBs sind im Abschnitt 4.11 (ab Seite 91) beschrieben.

Eine Flash-Bank kann immer nur komplett beschrieben werden, also mit einem 64K-Datenblock. Zuvor erfolgt ein Löschvorgang für die bereits vorhandenen Daten.

Flash-Schreiboperation sind zeitintensiv, sie nehmen einige Sekunden in Anspruch und sind somit stromausfallkritisch (ein Reset während eines Lösch- oder Übertragungsvorgangs bewirkt den Verlust aller zu speichernden Daten). Es sollten daher programmtechnisch Sicherheitsmaßnahmen getroffen werden, die auf einen möglichen Datenverlust reagieren (siehe Beispiel weiter unten).

Die eingesetzten Flash-Speicher besitzen nur eine begrenzte Lebensdauer von mindestens 10.000 Schreibvorgängen. Deshalb sollten Sicherungen in den Flash-Speicher nicht zu häufig vorgenommen werden.

Während der Übertragung der Daten in den Flash-Speicher wird die Ausführung des SPS-Programms unterbrochen.

Mögliche Verwendung der beiden Flash-Bänke:

- Größere Datenmengen können stromausfallsicher abgelegt werden (bis 128 KByte). Hierbei ist es sehr wichtig, dass ein Algorithmus geschaffen wird, mit dem erkannt werden kann, ob während einer Sicherung ein Stromausfall aufgetreten ist (analog zum Beispiel weiter unten).

- Reichen 64 KByte für die Sicherung von Daten aus, kann die zweite Bank als Sicherungskopie der ersten eingesetzt werden (sollte beim Überschreiben der ersten Bank ein Fehler auftreten – totaler Datenverlust –, enthält die andere immer noch die davor gültigen Daten).

Nachfolgend soll **beispielhaft** und in groben Zügen aufgezeigt werden, wie in einem Programm auf einen möglichen Stromausfall beim Schreiben in den Flash-Speicher reagiert werden kann. Im Programm sollen diverse Daten geändert werden, die zuletzt in der Flash-Bank 1 abgelegt wurden, und dann in die Flash-Bank 2 zurückgesichert werden (nach der nächsten Änderung soll dann wieder Bank 1 verwendet werden usw., immer im Wechsel).

0. Bei der Programmerstellung müssen zwei Merker als remanente BYTE-Variablen (RETAIN) reserviert werden (Speicherung im NV-RAM, s. u.) – einen für den jeweils durchgeführten Schritt (*step*) und den zweiten für die Nummer der verwendeten Flash-Bank (*flash_no*)
1. Merker initialisieren: *step* = 0 (*flash_no* steht auf 1)
2. Flash-Bank (1) in das RAM kopieren (FB *Restore_Memory*)
3. Schritt-Merker erhöhen: *step* = 1
4. Datenbereich aus dem reservierten RAM-Bereich in den normalen Arbeitsspeicher kopieren (→ zu ändernde Variable; FB *Memory_to_Var*)
5. Gewünschte Änderungen im Arbeitsspeicher ausführen
6. Variable(n) wieder in den reservierten RAM-Bereich kopieren (FB *Var_to_Memory*)
7. Schritt-Merker erhöhen: *step* = 2
8. Reservierten RAM-Bereich in die andere Flash-Bank (2) sichern (FB *Save_Memory*). Damit ist die Datenänderungsaktion abgeschlossen.
9. Schritt-Merker zurücksetzen und Flash-Merker auf die aktuelle Bank-Nummer setzen: *step* = 0, *flash_no* = 2 (die aktuellen Werte stehen jetzt in Flash-Bank 2)

   Direkt nach dem Einschalten muss dann im Programm abgefragt werden, welche Flash-Bank die aktuellen d. h. zuletzt gesicherten Daten enthält und ob der Schrittzähler *step* auf 0 steht, denn nur dann wurde die letzte Sicherung korrekt beendet. Andernfalls gibt *step* Auskunft über den zuletzt erfolgreich ausgeführten Schritt vor dem Abschalten.

3.2 RAM

Innerhalb des "normalen" Arbeitsspeichers befindet sich ein reservierter 64K-Block, der als Puffer für die Sicherung beliebiger Daten aus dem Variablen-Bereich verwendet werden kann (einzelne Variablen, Arrays usw.). Dieser RAM-Block kann dann in eine der beiden stromausfallsicheren Flash-Bänke übertragen oder aus diesen zurückgelesen werden. Der Zugriff auf diesen

Bereich ist nur möglich über spezielle FBs aus dem Bereich *Non volatile memory* (siehe Abschnitt 4.11, ab Seite 91).

Ein anderer RAM-Bereich enthält die Speicherplätze für die Systemparameter (siehe Betriebsanleitung zur MotionPLC), organisiert als ein Array von 1000 Longs (DINT), von denen ersten 500 (0…499) speziell für die Systemparameter reserviert sind. Die restlichen 500 Longs (500…999) können für die SPS-Programmierung nach IEC 61131 verwendet werden.

Diese Speicherplätze bieten folgende Vorteile:

- Über eine Betriebssystemfunktion (Kurvenscheiben-Software) oder eine IEC-61131-Funktion (*Save_Parameter*) können die Werte stromausfallsicher in das Flash übertragen werden. So werden also zusammen mit den Systemparametern auch SPS-Daten gesichert. Die Daten können in eine Datei auf der Festplatte gespeichert werden (über das mitgelieferte Hilfsprogramm LingiMon, siehe Betriebsanleitung).
- Die "Pseudo"-Parameter 500…999 können – wie auch die normalen Systemparameter – über die serielle Schnittstelle[1] einzeln und direkt angesprochen d. h. gelesen und geschrieben werden, ohne dass dafür extra ein eigenes Protokoll erstellt werden muss.

3.3 NV-RAM

Dieser Speicherbereich wird für die automatische Sicherung bei Spannungsunterschreitung (Stromausfallsicherung**)** verwendet. Hier werden

- die in der Entwicklungsumgebung CoDeSys durch den Zusatz RETAIN als remanent definierten Variablen gespeichert und
- vom Betriebssystem zyklisch die aktuellen Positionierdaten abgelegt.

Die Übertragung der unterschiedlichen Datenstrukturen in das NV-RAM erfolgt **byte-seriell**. Wenn also eine remanente DINT-Variable geändert werden soll, werden nacheinander deren 4 Bytes übertragen (bei Datenarrays sind dies entsprechend mehr Bytes). Dadurch besteht bei einem Stromausfall während einer solchen Datenübertragung die Gefahr, dass eine Variable nicht komplett mit neuen Bytes überschrieben wurde. Das würde nach dem Wiedereinschalten zu falschen Werten führen. Es wird daher dringend empfohlen, programmtechnisch Sicherheitsmaßnahmen vorzusehen, die diesen Fall berücksichtigen (so wie beim Flash-Speicher, siehe Beispiel weiter oben).

! Bei einem Programm-Download (CoDeSys → MotionPLC) wird dieser Speicherbereich nicht initialisiert, d. h. alle Variablen behalten ihre zuletzt definierten Werte.

[1] z. B. im Terminalbetrieb der Bediensoftware BB2100K oder im PLC-Browser von CoDeSys. Im PLC-Browser können auch die BB2100K-Kommandos verwendet werden, allerdings muss ihnen hierbei ein "@" vorangestellt werden (BB2100K: rr 502 ⇒ PLC-Browser: @rr 502); außerdem kann hier mit den Befehlen "n_dl_ini" und "dl_ini" die Initialisierung der Variablen sowie das Abschalten der Ausgänge beim Download eines geänderten Programms deaktiviert bzw. aktiviert werden (wenn deaktiviert, sollten keine Variablen eingefügt oder entfernt werden!).

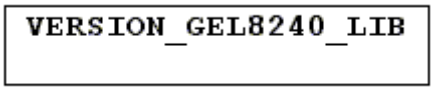
4 Funktionsblöcke

Nachfolgend werden die FBs in der Reihenfolge behandelt wie sie in der Baumstruktur des Bibliotheksfensters dargestellt sind; allerdings werden die dort an erstet Stelle angeordneten Untergruppen nachden FBs des betreffenden Zweigs behandelt.

Version_GEL8240_lib

Dieser "Pseudo“-FB liefert in seinem Deklarationsteil lediglich Informationen über die aktuelle Version der Bibliothek, inklusive der Historie.

- Funktionsblock:

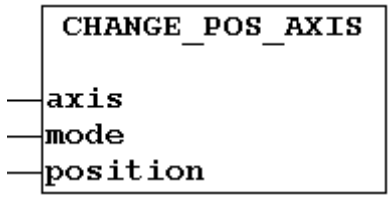


4.1 Achssteuerung (Axis Control)

Change_Pos_Axis *Axis Control*

Ändert die Istposition der x-Achse (Master) oder der y-Achsen (Slaves)

- Funktionsblock:



- Variablen:

axis WORD;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

mode WORD;
Art der Änderung für alle Achsen:
0 = absolut
1 = relativ

Art der Änderung nur für CAN-Achsen (max. 8):
2 = absolut beim nächsten Nullsignal, unabhängig von der Fahrtrichtung
3 = absolut beim nächsten Nullsignal in Vorwärtsfahrt
4 = absolut beim nächsten Nullsignal in Rückwärtsfahrt

position DINT;
für *mode* = 0|2|3|4: neuer Istwert
für *mode* = 1: Wert der Istwertänderung

◆ Beispiel:

Deklaration:

```
set_nom_val: Change_Pos_Axis;
```

Programm in ST:

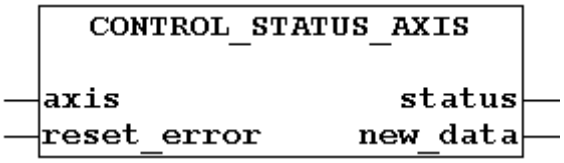
```
set_nom_val(axis:=3, mode:=1, position:=-1500);
set_nom_val(axis:=2, mode:=0, position:=500);
```

Bei Aufruf des ersten FB wird der momentane Istwert der Achse 3 um den Wert 1500 vermindert. Der zweite Aufruf setzt den Istwert der Achse 2 auf den Wert 500.

Control_Status_Axis *Axis Control*

Liefert Informationen über einen Servoumrichter am CAN-Bus.

◆ Funktionsblock:



◆ Variablen:

axis BYTE;
nur CAN-Achse:
1 … 11 = y-Achse 1 ... 11 (Slave)
Aktiviert werden können max. 11 Achsen. Nummeriert werden zunächst die analog angesteuerten Achsen (max. 3). Daran schließen sich die Achsen an, die über den CAN-Bus angesteuert werden (max. 8).

reset_error BOOL;
TRUE = Fehler im Servoumrichter werden gelöscht

status DWORD;

Bit-Nr.	Beschreibung
0	Logikzustand am Digitaleingang 1 (X3.11)
1	Logikzustand am Digitaleingang 2 (X3.12)
2	Logikzustand am Digitaleingang 3 (X3.13)
3	Logikzustand am Digitaleingang 4 (X3.14)
26	Initialisierung beendet
28	Motorstillstand
29	Sicherheitsrelais
30	Endstufe freigegeben
31	Fehler steht an

new_data BOOL;
TRUE = Kommunikation mit dem Servoumrichter läuft

◆ Beispiel:

Deklaration:
```
read_status: Control_Status_Axis;
```

Programm 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;
```

Der Status der Achse 4 wird gelesen. Im Falle einer Fehlermeldung (Bit 31 = TRUE) wird der Fehler gelöscht.

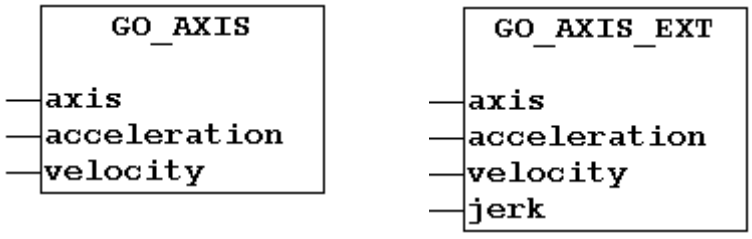
Go_Axis / Go_Axis_Ext *Axis Control*

Aktiviert das Verfahren einer Achse. Die aktivierte Achse wird solange mit den vorgegebenen Parametern verfahren, bis ein FB mit anderen Parametern für das Verfahren oder ein anderer Positionier-FB aufgerufen wird (wie unter "Siehe auch" aufgeführt).

Virtuelles Verfahren des Masters:
Wenn eine Kurve angewählt ist (bei aktivierter Stromausfallsicherheit bleibt die Nummer gespeichert, die vielleicht irgendwann einmal vorgegeben worden ist), wird der Master-Zählbereich auf den in der Kurve festgelegten x-Bereich eingeschränkt.

Siehe auch: Pos_Axis / Pos_Axis_Ext, Start_Cam_Axis, Stop_Axis / Stop_Axis_Ext

◆ Funktionsblock:



◆ Variablen:

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

acceleration DINT;
Beschleunigung der Achse in Inkrementen/s^2
Negative Werte und 0 werden intern in den kleinstmöglichen Beschleunigungswert umgesetzt.

velocity DINT;
Geschwindigkeit in Inkrementen/s mit Fahrtrichtung (Vorzeichen)

jerk DINT;
Ruckbegrenzung der Achse in Inkrementen/s^3
Negative Werte und 0 bewirken, dass ohne Ruckbegrenzung gefahren wird (wie beim FB Go_Axis).

- Beispiel:

Deklaration:

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

Initialisierung in ST:

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

Programm in ST:

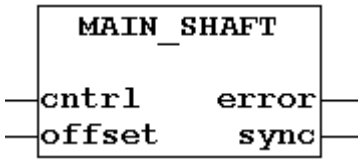
```
IF start_new AND start_old=FALSE THEN
   move_1(velocity:=-15000);
   move_2(acceleration:=120000; velocity:=28000);
END_IF;
start_old:=start_new;
```

Mit der positiven Flanke von `start_new` wird der Verfahrvorgang der Achsen 1 und 2 einmalig eingeleitet. Die Achse 1 verfährt dabei rückwärts mit der zuletzt vorgegebenen Beschleunigung. Wurde der Beschleunigungswert nach der Initialisierung nicht überschrieben, so entspricht er noch dem Wert in der Initialisierung.
Die Achse 2 wird vorwärts verfahren mit einem neuen Beschleunigungswert. Dieser Wert gilt bis die Variable `move_2.acceleration` erneut überschrieben wird.

Main_Shaft *Axis Control*

Wird in einem Königswellen-Slave verwendet, sofern die Master-Position der Königswelle über den CAN-Bus übertragen wird. Der FB liefert außerdem die Positionsabweichung zwischen CAN-Master und -Slave und schreibt einen Versatz.

- Funktionsblock:



- Variablen:

cntrl BYTE;
0 = nur Königswellen-Daten lesen
1 = Königswellen-Daten lesen und Versatz schreiben

offset DINT;
Wert des Versatzes gegenüber dem Königswellen-Master (≥ 0); negative Werte werden nicht geschrieben

Ist im Fall *cntrl* = 1 das Schreiben des Versatzes erlaubt und ist dieser nicht negativ, dann wird der Master-Referenzwert [113] im RAM ohne Einschränkung überschrieben. Folgt anschließend ein Kopiervorgang der Parameter ins Flash, so bleibt der neue Wert auch nach einem Spannungsausfall erhalten.

error DINT;
Nur für Testzwecke; sollte immer nahe 0 liegen, wenn mit Sync-Signal gearbeitet wird, was zu empfehlen ist.

sync BOOL;
TRUE = Kommunikation mit dem Königswellen-Master über den CAN-Bus läuft

- Beispiel:

Deklaration:

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

Programm 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;
```

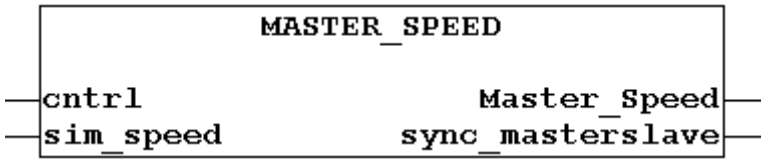
Die positive Flanke von `key_new` setzt den Systemparameter para[113] im RAM auf den Wert 350 und kopiert anschließend den RAM-Bereich für die Parameter ins Flash. Somit ist der Wert 350 dauerhaft gespeichert.

Master_Speed *Axis Control*

Prüft bei einem Königswellen-Slave, ob Daten von einem Königswellen-Master geliefert werden, liest die Istgeschwindigkeit des Kurvenscheiben-Masters und schreibt die Simulationsgeschwindigkeit, sofern der Kurvenscheiben-Master virtuell ist.

Siehe auch: Go_Axis, Rd_Status_Axis

- Funktionsblock:



- Variablen:

cntrl BYTE;
0 = Master-Geschwindigkeit lesen
1 = Master-Geschwindigkeit lesen und Simulationsgeschwindigkeit schreiben

sim_speed DINT;
Simulationsgeschwindigkeit in Master-Inkrementen/s (Sollgeschwindigkeit, Vorgabe); wird ignoriert bei einer Königswellen-Anwendung oder wenn die Master-Position über Inkrementalgeber erzeugt wird

Master_speed DINT;
Master-Geschwindigkeit in Inkrementen/s (Istgeschwindigkeit des Masters)

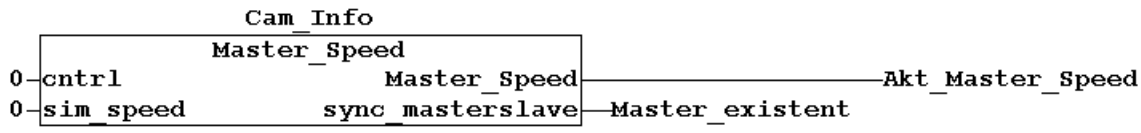
sync_masterslave BOOL;
TRUE = dieser FB wird in einem Königswellen-Slave verwendet und die Kommunikation mit dem Master läuft

- Beispiel:

Deklaration:

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

Programm in FUP:



Das Netzwerk zeigt den Aufruf des FB. Die gelesenen Daten werden den Variablen `Akt_Master_Speed` und `Master_existent` zugewiesen. Eine Simulationsgeschwindigkeit wird nicht geschrieben.

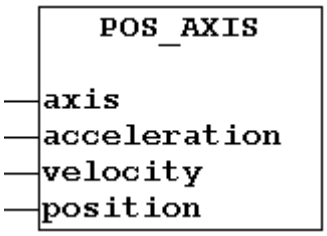
Pos_Axis / Pos_Axis_Ext *Axis Control*

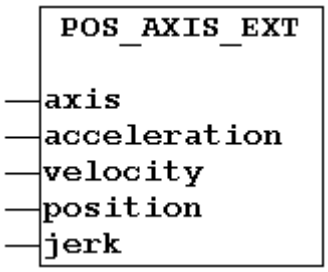
Aktiviert den Positioniervorgang einer Achse. Der Positioniervorgang bleibt auch bei erreichter Zielposition so lange aktiv, bis ein FB mit anderen Parametern für das Verfahren oder ein anderer Positionier-FB aufgerufen wird (wie unter "Siehe auch" aufgeführt).

Virtuelles Verfahren des Masters:
Wenn eine Kurve angewählt ist (bei aktivierter Stromausfallsicherheit bleibt die Nummer gespeichert, die vielleicht irgendwann einmal vorgegeben worden ist), wird der Master-Zählbereich auf den in der Kurve festgelegten x-Bereich eingeschränkt. Die Position wird jedoch korrekt angefahren – nur entspricht der mit dem FB Rd_Status ausgelesene Positionswert eventuell nicht dem vorgegebenen Wert für die Zielposition.

Siehe auch: Go_Axis / Go_Axis_Ext, Start_Cam_Axis, Stop_Axis / Stop_Axis_Ext

- Funktionsblock:





- Variablen:

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

acceleration DINT;
Beschleunigung der Achse in Inkrementen/s^2
Negative Werte und 0 werden intern in den kleinstmöglichen Beschleunigungswert umgesetzt.

velocity DINT;
Geschwindigkeit in Inkrementen/s, ≥ 0!

position DINT;
Zielposition in Inkrementen (Master: siehe Info weiter oben)

jerk DINT;
Ruckbegrenzung der Achse in Inkrementen/s^3
Negative Werte und 0 bewirken, dass ohne Ruckbegrenzung gefahren wird (wie beim FB Pos_Axis).

- Beispiel:

  Deklaration:

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

  Initialisierung in ST:

```
pos_0.axis:=0;
```

  Programm in ST:

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

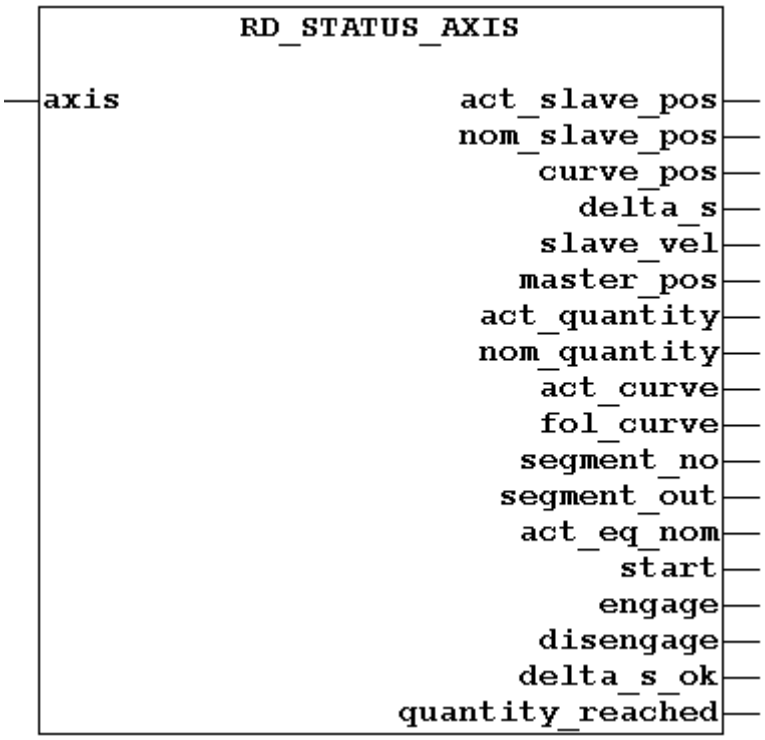
  Der einmalige Aufruf des FB startet den Positioniervorgang der Master-Achse (virtuelle Achse) mit den vorgegebenen Werten für Beschleunigung und Geschwindigkeit auf die Zielposition 500000. (Wäre eine Kurve angewählt mit einem x-Bereich von 200000, so betrüge die auslesbare Endposition 100000, wobei die Position 2× nach Erreichen des Wertes 200000 auf Null zurückgesetzt worden ist. Wenn dieses Verhalten unerwünscht ist, muss die Stromausfallsicherheit in der MotionPLC deaktiviert sein, um sicher zu gehen, dass nach dem Einschalten keine Kurve angewählt ist.)

Rd_Status_Axis *Axis Control*

Liefert Zustandsinformationen über die anzugebende Achse

Siehe auch: Go_Axis / Go_Axis_Ext, Pos_Axis / Pos_Axis_Ext, Stop_Axis / Stop_Axis_Ext

- Funktionsblock:



- Variablen:

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

act_slave_pos DINT;
für *axis* = 0 … 11:
Istposition in Inkrementen

nom_slave_pos DINT;
für *axis* = 1 … 11:
momentane Sollposition in Inkrementen

curve_pos DINT;
für *axis* = 1 … 11:
momentane Kurvenposition in Inkrementen in Abhängigkeit der Master-Position und einer für die vorgegebene Slave-Achse gewählten und gültigen Kurve
Ist die Kurve gestartet und der Einkuppelvorgang abgeschlossen, dann ist der hier vorgegebene Wert gleich dem Wert von *nom_slave_pos*.

! Der Wert wird nur aufbereitet, wenn die angewählte Kurve in der vorgegebenen Achse existiert, auch wenn sie nicht gestartet ist und die Achse ohne Vorgabe der Ruckbegrenzung verfahren wird. Dagegen wird dieser Wert nicht aufbereitet, solange für diese Achse eine Geschwindigkeit ≠ 0 mit Ruckbegrenzung ausgegeben wird (Go_Axis_Ext, Pos_Axis_Ext, Stop_Axis_Ext mit *jerk* ≠ *0*).

delta_s DINT;
für *axis* = 1 … 11:
Schleppabstand zwischen der momentanen Sollposition und der Istposition:

$$delta\_s = nom\_slave\_pos - act\_slave\_pos$$

slave_vel DINT;
für *axis* = 0 … 11:
momentane Istgeschwindigkeit in Inkrementen/s
Wird die Master-Achse vorgegeben und ist diese virtuell, so wird an dieser Stelle die Simulationsgeschwindigkeit ausgegeben.

master_pos DINT;
für *axis* = 1 … 11:
Istposition der Master-Achse in Inkrementen der Master-Achse

act_quantity DINT;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

nom_quantity DINT;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

act_curve INT;
für *axis* = 1 … 11:
Nummer der angewählten Kurve

fol_curve INT;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

segment_no BYTE;
für *axis* = 1 … 11:
Nummer des Kurvenabschnitts der vorgegebenen Slave-Achse, in dem sich der Master gerade befindet

! Einschränkung wie bei `curve_pos` beschrieben.

segment_out BYTE;
für *axis* = 1 … 11:
Liefert die OUT-Signale des Kurvenabschnitts der vorgegebenen Slave-Achse, in dem sich der Master gerade befindet.

! Einschränkung wie bei `curve_pos` beschrieben.

act_eq_nom BOOL;
nur für *axis* = 0 (Master):
Liefert in der vorgegebenen Achse das negierte Signal zum Ausgang *engage*

start BOOL;
für *axis* = 0 … 11:
TRUE = in der vorgegebenen Achse wurde eine gültige Kurve mit Start_Cam_Axis oder ein Positioniervorgang mit Pos_Axis / Pos_Axis_Ext gestartet
FALSE = in der vorgegebenen Achse wird eine der Anweisungen Go_Axis / Go_Axis_Ext oder Stop_Axis / Stop_Axis_Ext ausgelöst

engage BOOL;
für *axis* = 0 … 11:
TRUE = in der vorgegebenen Achse wurde ein Positioniervorgang mit Pos_Axis / Pos_Axis_Ext gestartet und die momentane Sollposition *nom_slave_pos* (Slave oder Master) hat noch nicht die Zielposition erreicht
zusätzlich für *axis* = 1 … 11:
TRUE = in der vorgegebenen Achse wurde eine gültige Kurve mit Start_Cam_Axis gestartet und die momentane Sollposition *nom_slave_pos* hat noch nicht die momentane Kurvenposition *curve_pos* erreicht (Einkuppelvorgang)

disengage BOOL;
für *axis* = 1 … 11:
TRUE = in der vorgegebenen Achse wurde eine gültige Kurve, die mit Start_Cam_Axis gestartet wurde, durch Stop_Axis / -Stop_Axis_Ext gestoppt und der Stillstand dieser Achse wurde noch nicht erreicht (Auskuppelvorgang)

delta_s_ok BOOL;
für *axis* = 1 … 11:
TRUE = Schleppabstand bei aktiver Lageregelung liegt innerhalb des zulässigen Fensters; der maximal erlaubte Schleppfehler wird achsabhängig über entsprechende Systemparameter für die Analog- und CAN-Achsen vorgegeben

quantity_reached BOOL;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

◆ Beispiel:

Deklaration:

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

Programm 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;
```

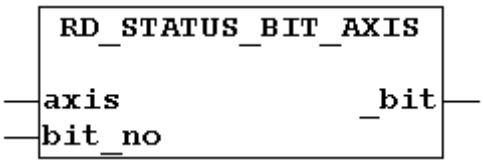
Der mehrfache, zyklische Aufruf des gleichen FB liefert nacheinander die Istpositionen der Master-Achse (0) und der Slave-Achsen 1 und 2. Zwischen den Aufrufen wird der jeweilige Istwert verschiedenen Variablen zugewiesen.

Rd_Status_Bit_Axis *Axis Control*

Liefert den Zustand des vorgegebenen Bits in der angewählten Achse

Siehe auch: Rd_Status_Axis

◆ Funktionsblock:



◆ Variablen:

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

bit_no BYTE;
Bit, dessen Zustand geliefert werden soll:

bit_no	globale Variable	gültig für Achse ...
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

bit_no	globale Variable	gültig für Achse ...
10	*drive_err*	CAN
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

Liefert in der vorgegebenen Achse das negierte Signal zum Ausgang *engage*

start

TRUE = in der vorgegebenen Achse wurde eine gültige Kurve mit Start_Cam_Axis oder ein Positioniervorgang mit Pos_Axis / Pos_Axis_Ext gestartet

FALSE = in der vorgegebenen Achse wird eine der Anweisungen Go_Axis / Go_Axis_Ext oder Stop_Axis / Stop_Axis_Ext ausgelöst

engage

TRUE = in der vorgegebenen Achse wurde ein Positioniervorgang mit Pos_Axis / Pos_Axis_Ext gestartet und die momentane Sollposition *nom_slave_pos* (Slave oder Master) hat noch nicht die Zielposition erreicht

zusätzlich für *axis* = 1 … 11:

TRUE = in der vorgegebenen Achse wurde eine gültige Kurve mit Start_Cam_Axis gestartet und die momentane Sollposition *nom_slave_pos* hat noch nicht die momentane Kurvenposition *curve_pos* erreicht (Einkuppelvorgang)

disengage

TRUE = in der vorgegebenen Achse wurde eine gültige Kurve, die mit Start_Cam_Axis gestartet wurde, durch Stop_Axis / -Stop_Axis_Ext gestoppt und der Stillstand dieser Achse wurde noch nicht erreicht (Auskuppelvorgang)

delta_s_ok

TRUE = Schleppabstand bei aktiver Lageregelung liegt innerhalb des zulässigen Fensters; der maximal erlaubte Schleppfehler wird achsabhängig über entsprechende Systemparameter für die Analog- und CAN-Achsen vorgegeben

ref_reached
TRUE = Referenz-/Positionswert wurde gesetzt (FB Change_Pos_Axis)

drive_err
TRUE = Fehler im Servoumrichter
Über den FB Control_Status_Axis können Fehler im Servoumrichter gelöscht werden. (Der FB Rw_Param_Axis ermöglicht das Lesen von Servoumrichterfehlern.)

cam_started
TRUE = Kurve ist für die gewählte Achse gestartet

ext_achse
TRUE = die Achse existiert, d. h. sie ist über para[332] oder para[451] aktiv geschaltet (die Achse 0 (Master) ist immer aktiviert)

ext_LD2000
TRUE = die Achse ist eine CAN-Achse (Umrichter LD 2000)

ext_LD2000_prg
TRUE = die Parameter des jeweiligen Umrichters erlauben einen Betrieb als CAN-Achse mit der MotionPLC

ext_LD2000_init
TRUE = die Initialisierung des jeweiligen Umrichters wurde erfolgreich abgeschlossen

ext_LD2000_online
TRUE = die Kommunikation mit dem jeweiligen Umrichter läuft

ext_LD2000_operational
TRUE = die CAN-Achse ist bezüglich CANopen betriebsbereit

drive_enable
TRUE = die Freigabe für diese Achse wird von der MotionPLC vorgegeben (bei CAN-Achsen schaltet der jeweilige Umrichter dabei in den Zustand "enabled")

drive_forwards
TRUE = die Achse befindet sich in Vorwärtsfahrt

drive_backwards
TRUE = die Achse befindet sich in Rückwärtsfahrt

_bit BOOL;
Liefert den Zustand des gewählten Bits

- Beispiel:

  Deklaration:

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

Programm 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;
```

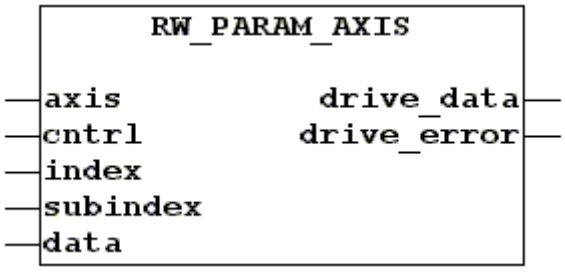
Die Variable `online_axis` ist TRUE, wenn die Achsen 4 und 5 online sind.

RW_Param_Axis *Axis Control*

Erlaubt das Lesen und Schreiben von Servoumrichter-Parametern über den CAN-Bus (SDOs)

Dieser FB sollte nicht zyklisch in einem Programm aufgerufen werden: Er kann die SPS-Zykluszeit um ca. 25 ms verlängern.

- Funktionsblock:



- Variablen:

axis BYTE;
nur CAN-Achse:
1 … 11 = y-Achse 1 ... 11 (Slave)

cntrl BYTE;
Datenrichtung:
34 [DOWNLOAD*] = Daten an den Servoumrichter senden
64 [UPLOAD*] = Daten vom Servoumrichter anfordern

index (WORD), *subindex* (BYTE), *data* (DINT), *drive_data* (DINT) siehe CANopen-Dokumentation auf der mitgelieferten CD (ba_can_d.pdf)

drive_error BYTE;
0 = kein Fehler
1 = Kommunikationsfehler
2 = Timeout
3 = ungültige Achse

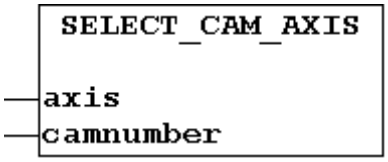
* Bei den globalen Variablen der Bibliothek definierte Konstante

Select_Cam_Axis *Axis Control*

Kurve für eine vorzugebende Achse wählen

Siehe auch: Start_Cam_Axis

- Funktionsblock:



- Variablen:

axis BYTE;
1 … 11 = y-Achse 1 … 11 (Slave)

camnumber DINT;
Kurvennummer im Bereich 0…99

- Beispiel:

Deklaration:

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

Programm 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;
```

Im Programm wird eine neue Kurvennummer mit 0 vogegeben und überprüft, ob diese Kurve bereits eingestellt ist. Ist dies nicht der Fall, wird die neue Kurve angewählt.

Start_Cam_Axis *Axis Control*

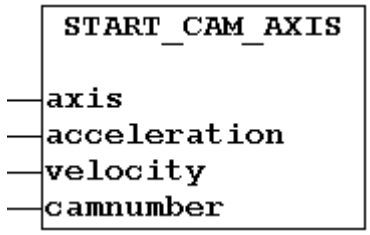
Startet die angewählte Kurve in der ausgewählten Achse mit den Vorgaben für Beschleunigung und Geschwindigkeit für das Einkuppeln (*engage*, siehe auch Rd_Status_Axis)

Ist zu diesem Zeitpunkt bereits eine Kurve gestartet, so wird diese abgebrochen und dann auf die neue Kurve eingekuppelt (Ausnahme: siehe para[181] in der Betriebsanleitung).

Ein Referenzwert beim Master wird **nicht** gesetzt, auch wenn dies mit para[115] so festgelegt wurde.

Siehe auch: Select_Cam_Axis

- Funktionsblock:



- Variablen:

axis BYTE;
1 … 11 = y-Achse 1 … 11 (Slave)

acceleration DINT;
Beschleunigung der Achse für den Einkuppelvorgang in Inkrementen/s^2
Negative Werte und 0 werden intern in den kleinstmöglichen Beschleunigungswert umgesetzt.

velocity DINT;
Geschwindigkeit für den Einkuppelvorgang in Inkrementen/s, ≥ 0!

camnumber DINT;
Kurvennummer 0…99

- Beispiel:

Deklaration:

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

Programm 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;
```

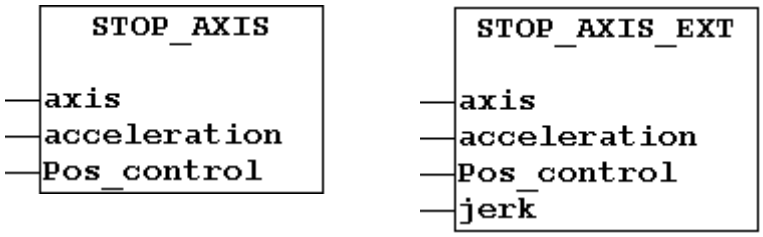
Der einmalige Aufruf (positive Flanke von `start_new`) startet die Kurve 0 in der Achse 2 (Slave) und den Positioniervorgang der Achse 0 (virtuelle Master-Achse) mit den vorgegebenen Werten für Beschleunigung und Geschwindigkeit auf die Zielposition 100000.

Stop_Axis / Stop_Axis_Ext *Axis Control*

Stoppt den Verfahrvorgang für die angegebene Achse

Siehe auch: Go_Axis / Go_Axis _Ext, Pos_Axis / Pos_Axis_Ext, Start_Cam_Axis

- Funktionsblock:



- Variablen:

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

acceleration DINT;
Verzögerung der Achse in Inkrementen/s^2
Negative Werte und 0 werden intern in den kleinstmöglichen Verzögerungswert umgesetzt.

Pos_control BOOL;
FALSE = keine Lageregelung im Stillstand
TRUE = Lageregler bleibt bei Stillstand aktiv

jerk DINT;
Ruckbegrenzung der Achse in Inkrementen/s^3
Negative Werte und 0 bewirken, dass ohne Ruckbegrenzung gefahren wird. Dies entspricht dem FB Stop_Axis.

- Beispiel:

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

Initialisierung in ST:
```
stop_movement.acceleration:=2500000;
stop_movement.Pos_control:=FALSE;
```

Programm 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;
```

Mit der negativen Flanke von `stop_new` werden die Achsen 1 bis 7 mit gleicher Verzögerung gestoppt. Der Lageregelkreis aller Achsen ist im Stillstand inaktiv.

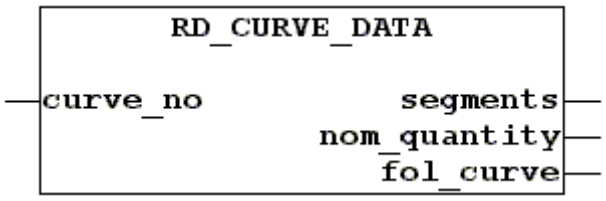
4.2 Kurvendaten (Cam Data)

Rd_Curve_Data *Cam Data*

Liefert Informationen über eine bestimmte Kurve

Siehe auch: Wr_Fol_Curve, Wr_Nom_Quantity

- Funktionsblock:



- Variablen:

curve_no BYTE;
Nummer der gewünschten Kurve im Bereich 0 … 99

segments BYTE;
Anzahl der in dieser Kurve existierenden Abschnitte;
0 = Kurve existiert nicht

nom_quantity DINT;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

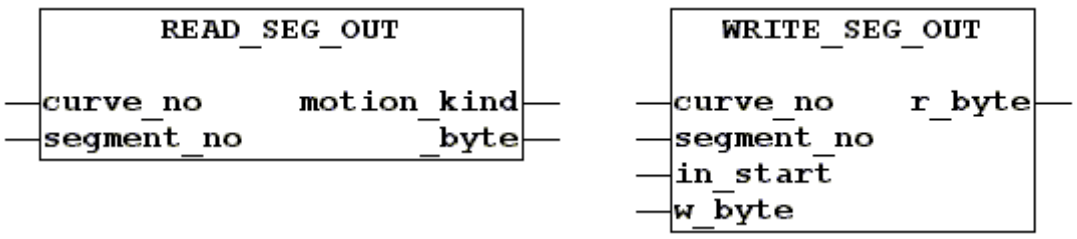
fol_curve INT;
Es wird kein Wert aufbereitet. Wegen der Kompatibilität zur MotionCard LD100 wurde der Ausgang beibehalten.

Read_Seg_Out / Write_Seg_Out *Cam Data*

Liefert die Bewegungsart und den Zustand des frei verfügbaren Datenbytes eines Kurvensegments / Schreibt das frei verfügbare Datenbyte

Siehe auch: Wr_Fol_Curve, Wr_Nom_Quantity

- Funktionsblock:



◆ Variablen:

curve_no BYTE;
Nummer der gewünschten Kurve im Bereich 0 … 99

segment_no BYTE;
Nummer des Abschnitts dieser Kurve

motion_kind BYTE;
Art und Form der Bewegung innerhalb des angewählten Kurvenabschnitts
Hierfür sind bei den globalen Variablen unter "kind of motion" bestimmte Konstanten definiert worden.

Byte	Konstante	Art	Form
0			Abschnitt existiert nicht
5	V_CONST	v = const	
6	V_EQ_0	v = 0	
11	HARM_PP	P1 → P2	harmonisch
12	HARM_PV	P → v	harmonisch
13	HARM_VP	v → P	harmonisch
21	CYCL_PP	P1 → P2	zykloid
22	CYCL_PV	P → v	zykloid
23	CYCL_VP	v → P	zykloid
24	CYCL_VV	v1 → v2	zykloid

_byte BYTE;
frei verfügbares Datenbyte = programmierte Ausgangssignale innerhalb des angewählten Kurvenabschnitts (Out 1…8; 1 = High)

in_start BOOL;
Schreibbedingung:
FALSE = Das Datenbyte kann nur geschrieben werden, wenn sich die Kurve nicht im gestarteten Zustand befindet.
TRUE = Das Datenbyte kann auch im gestarteten Zustand der Kurve geschrieben werden.

w_byte BYTE;
Zustand, der in das frei verfügbare Datenbyte geschrieben wird (Out 1…8; 1 = High)

r_byte BYTE;
Zustand des frei verfügbaren Datenbytes nach dem Schreiben

- Beispiel:

  Deklaration:

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

  Programm 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;
```

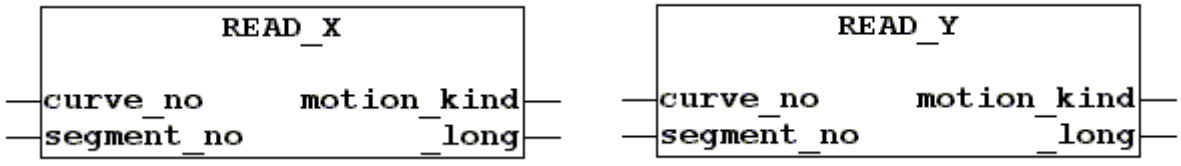
  Für jeden der 10 Kurvenabschnitte von Kurve 3 wird die Bewegungsart gelesen. Ist diese mit v = constant (festes Geschwindigkeitsverhältnis zwischen Master und Slave) vorgegeben, dann wird das höchstwertige Bit des frei verfügbaren Datenbytes auf 1 gesetzt, in allen anderen Fällen auf 0. Die anderen Bits des Datenbytes bleiben unverändert.

Read_x / Read_y *Cam Data*

Liefert Informationen über die Art der Bewegung und die Endposition x (Master) bzw. y (Slave) eines bestimmten Abschnitts aus einer angewählten Kurve

Siehe auch: Write_x, Write_y

- Funktionsblock:



- Variablen:

  curve_no, *segment_no*, *motion_kind* wie FB `Read_Seg_Out` (s. o.)

  _long DINT;
  für `segment_no` = 1…31:
  Endposition x bzw. y im vorgegebenen Kurvenabschnitt
  für `segment_no` = 0:
  Anfangsposition x (immer 0) bzw. y des Kurvenabschnitts 1

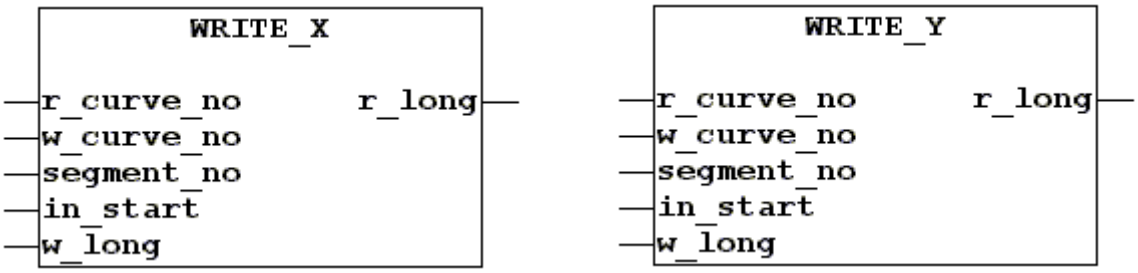
◆ Beispiel: siehe Write_x, Write_y

Write_x / Write_y *Cam Data*

Kopiert eine Quellkurve in den Zwischenspeicher, ersetzt hier den Endwert x bzw. y des vorgegebenen Kurvenabschnitts, berechnet den Endwert der jeweils anderen Achse neu, verschiebt alle Endwerte x und y der folgenden Abschnitte um die jeweiligen Änderungen, überschreibt eine vorgegebene Zielkurve mit den Kurvendaten des Zwischenspeichers (Schreibbedingungen unter *in_start*) und liefert die Endposition nach dem Überschreiben der Zielkurve

Siehe auch: Read_x, Read_y

◆ Funktionsblock:



◆ Variablen:

r_curve_no BYTE;
Nummer der Quellkurve: 0 … 99

w_curve_no BYTE;
Nummer der Zielkurve: 0 … 99

segment_no BYTE;
Nummer des Kurvenabschnitts: 0, 1 ... 30

Der Wert 0 ermöglicht den Zugriff auf die Anfangswerte von Kurvenabschnitt 1.

in_start BOOL;
Schreibbedingung:
FALSE = Kurve noch nicht gestartet (zwingend)

Weitere Schreibbedingungen:
– für Write_**x**

Der vorgegebene Wert muß größer als der Endwert des vorherigen Kurvenabschnittes sein.

– für Write_**x** und Write_**y**

Die Nummer des Kurvenabschnittes darf nicht größer sein als die Nummer des letzten Abschnittes der gewählten Kurve. Die Bewegungsart darf nicht "v1 → v2" (24) sein.

w_long DINT;
für *segment_no* = 1…31:
zu schreibende Endposition x bzw. y im vorgegebenen Kurvenabschnitt der Quellkurve
für *segment_no* = 0:
Anfangsposition x (immer 0) bzw. y des Kurvenabschnitts 1

r_long DINT;
für *segment_no* = 1…31:
gelesene Endposition x bzw. y im vorgegebenen Kurvenabschnitt der Zielkurve
für *segment_no* = 0:
Anfangsposition x (immer 0) bzw. y des Kurvenabschnitts 1

◆ Beispiel:

Deklaration:

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

Programm 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;
```

Ist in Abschnitt 3 der Quellkurve 0 die Bewegungsart "Pos1 → Pos2", dann wird die y-Strecke im Abschnitt 3 ermittelt, mit dem Faktor 2 multipliziert und in die Zielkurve 5 übertragen. Der Schreibvorgang wird anschließend überprüft. Ist die Variable *error* FALSE, dann war der Schreibvorgang erfolgreich.

4.2.1 Neue Kurven (New Cam)

Die hier zusammengefassten FBs liefern die nötigen Werkzeuge für die Erstellung und erweiterte Bearbeitung von Kurven aus der SPS heraus.

> **!** Der Eingriff in die Kurvenstruktur ist sehr **kritisch** und nicht ganz einfach zu verstehen.
>
> Deswegen sollten diese FBs nur von **versierten** und **geschulten** Anwendern genutzt werden, die mit der Kurvenproblematik **bestens vertraut** sind.

Die auf der CD mitgelieferte Datei 'fl_saw.pro' zeigt am Beispiel einer Fliegenden Säge, wie einzelne Kurvenabschnitte berechnet, in das korrekte Format gewandelt und in das strukturierte Kurvenarray geschrieben werden.

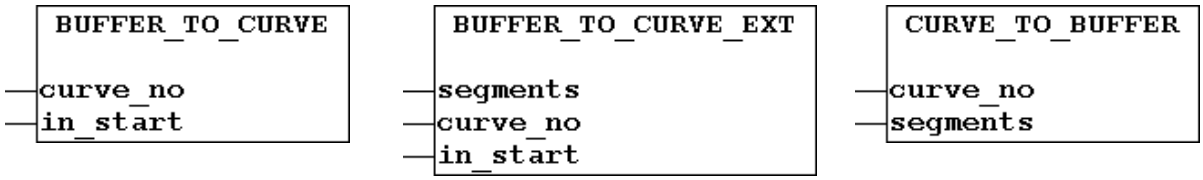
Buffer_to_Curve / Buffer_to_Curve_Ext / Curve_to_Buffer	*Cam Data (New Cam)*

Kopiert eine Kurve bzw. einen Kurventeil aus dem RAM-Datenbereich in den Bearbeitungsbereich (Buffer): Curve_to_Buffer

Schreibt eine Kurve aus dem Bearbeitungsbereich in den RAM-Datenbereich: Buffer_to_…

Siehe auch: Rd_Curve_Array, Wr_Curve_Array; Save_Parameter

- Funktionsblock:



- Variablen:

curve_no BYTE;
zu kopierende Kurve im Bereich von 0 ... 99

in_start BOOL;
FALSE = Kurve noch nicht gestartet (zwingend)

segments BYTE;
Anzahl der zu kopierenden bzw. zu schreibenden Kurvensegmente:

für Curve_to_Buffer:
0 = Alle vorhandenen Segmente werden gelesen. Die Segmentanzahl wird im Bearbeitungsbereich auf die vorhandene Anzahl Segmente gesetzt
1 … 30 = Die vorgegebene Anzahl von Segmenten wird gelesen. Dies ist auch der Fall, wenn die im Datenbereich vorhandene Anzahl der Segmente kleiner ist als die vorge-

gebene Anzahl. Die Segmentanzahl wird im Bearbeitungsbereich auf die vorgegebene Anzahl Segmente gesetzt.

für Buffer_to_Curve_Ext:
0 = Kurve wird gelöscht
1 … 30 = die vorgegebene Kurve im Datenbereich wird mit der angegebenen Anzahl von Segmenten überschrieben

- Beispiel:

  Deklaration:

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

  Programm 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();
```

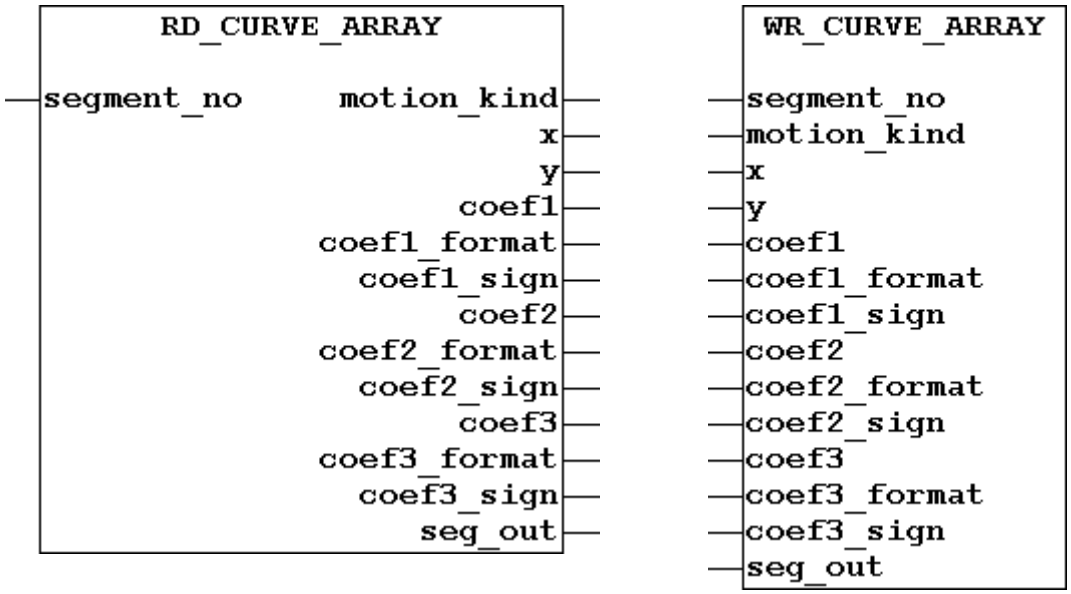
  Ein einmaliger Programmdurchlauf liest die Kurve 0 mit den vorhandenen Segmenten aus dem RAM-Datenbereich in den Bearbeitungsbereich (Buffer) und schreibt diesen dann zurück in den Datenbereich als Kurve 1 mit der zuvor gelesenen Anzahl von Segmenten. Anschließend wird die Kurve 0 im RAM-Datenbereich gelöscht. Um die Änderungen dauerhaft zu sichern, wird anschließend der Datenbereich mit allen Kurven in den Flash-Speicher kopiert.

Rd_Curve_Array / Wr_Curve_Array *Cam Data (New Cam)*

Liefert / erstellt Daten für ein Kurvensegment im Bearbeitungsbereich (Buffer)

Siehe auch: Buffer_to_Curve, Buffer_to_Curve_Ext, Curve_to_Buffer

- Funktionsblock:



- Variablen:

Segment_no BYTE;
Vorgabe eines Segments: 0, 1 … 30
Die x- und y-Werte des Segments 0 liefern die Start-positionen des Segments 1 (x = 0: Master).

Motion_kind BYTE;
Bewegungsart und Kurvenform
Hierfür sind bei den globalen Variablen unter "kind of motion" bestimmte Konstanten definiert worden.

Wert	Konstante	Art	Form	verwendete Koeffizienten		
				coef1	coef2	coef3
5	V_CONST	v = constant		•		
6	V_EQ_0	v = 0				
11	HARM_PP	Pos 1 → Pos 2	harmonisch			
12	HARM_PV	Pos → v	harmonisch			
13	HARM_VP	v → Pos	harmonisch			
21	CYCL_PP	Pos 1 → Pos 2	zykloid		•	
22	CYCL_PV	Pos → v	zykloid		•	
23	CYCL_VP	v → Pos	zykloid		•	
24	CYCL_VV	v1 → v2	zykloid	•	•	

Wert	Konstante	Art	Form	verwendete Koeffizienten		
				coef1	coef2	coef3
50	reserviert für die Interpolation der Bibliotheken cnc_2d.lib und cnc_3d.lib			werden in der Interpolation mit anderer Bedeutung genutzt		
51						
100	FREE_CAM	Tabelle mit y-Werten[2]		•	•	•

x DINT;
x-Endwert des Segments in Inkrementen der Master-Achse

y DINT;
y-Endwert des Segments in Inkrementen der Slave-Achse, in der die Kurve verwendet wird

coef1 DWORD;
– für *motion_kind* ≤ 24:
Absolutwert des Startgeschwindigkeits-Quotienten:

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

– für *motion_kind* = 100:
Startadresse der Tabelle mit y-Werten

coef1_sign BYTE;
Vorzeichen des Geschwindigkeitsquotienten:
0 = "+"
für Rd_Curve_Array: ≠0 = "-"
für Wr_Curve_Array: 1 = "-"

coef2 DWORD;
– für *motion_kind* ≤ 24:
Absolutwert des Wegdifferenzquotienten:

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

Δy = Endposition y – Startposition y,
Δx = Endposition x – Startposition x

– für *motion_kind* = 100:
Anzahl der Masterinkremente pro y-Wert (Tabellenintervall)

coef2_sign BYTE;
Information für die Darstellung im Kurveneditor des Programms BB2100K (1 = ☑), für *motion_kind* ≤ 24:

[2] Die Werte müssen vom Typ DINT sein, bezogen auf den Endwert des vorherigen Kurvenabschnitts. Zwischen zwei y-Werten wird linear interpoliert (der Kurvenverlauf kann im BB2100K-Kurveneditor überprüft werden: Kurve laden und berechnen lassen). Die Tabelle darf maximal 64 KByte groß sein. Die Bewegungsart des folgenden Kurvenabschnitts darf nicht 24 sein ("v1 → v2").

Bit	Check Box	Wahlmöglichkeit			
		v1 → v2		v = constant	
2^0	☐ Endposition x [Inc.]	1/0	1/0	0	1
2^1	☐ Endposition y [Inc.]	0	1	1	0
2^2	☐ Endgeschw. [v_y/v_x]	1	0	0	0

coef1_format, *coef2_format* INT;
Formatierung für coef1 bzw. coef2 (für *motion_kind* ≤ 24):

coef#_format	*format*	Verwendung
1	2^0	bei $\left\|\frac{v_y}{v_x}\right\| > 2^{16}$ bzw. $\left\|\frac{\Delta y}{\Delta x}\right\| > 2^{16}$
0	2^{16}	Standard
-1	2^{32}	nur bei $\left\|\frac{v_y}{v_x}\right\| < 1$ bzw. $\left\|\frac{\Delta y}{\Delta x}\right\| < 1$

(# = 1 oder 2)

coef3, *coef3_format* DWORD;
coef3_sign BYTE;
– für *motion_kind* ≤ 24:
Werden in Verbindung mit der Kurvenscheibenfunktion nicht verwendet.
– für *motion_kind* = 100 (nur *coef3*):
Tabellenlänge (Anzahl der y-Werte in der Tabelle)

seg_out BYTE;
Datenbyte zur freien Verfügung

◆ Beispiel:

Deklaration:

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

Programm 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);
```

Der einmalige Aufruf des Programms erzeugt die Kurve 1, die einen gegensinnigen Gleichlauf zwischen Master und Slave im Verhältnis von 47:33 bewirkt.

4.3 Nockenschaltwerk (Cam Tracks)

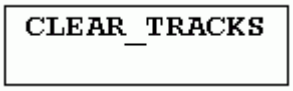
Mit der Nockenschaltwerksfunktion können bis zu 16 Bahnen mit beliebig vielen Nockenschaltpunkten realisiert werden. Die Schaltzustände der einzelnen Bahnen werden bitweise in einer Variablen abgelegt. Bis zu 15 Bahnen können zusätzlich an den Digitalausgängen Q1.0…Q3.4 ausgegeben werden.

Das Setzen der Variablen und Ausgänge wird vom Betriebssystem vorgenommen, erfolgt also unabhängig vom SPS-Zyklus mit der Zykluszeit der Kurvenscheibenanwendung.

Clear_Tracks *Cam Tracks*

Löscht alle Bahnen des Nockenschaltwerks (sollte in der Initialisierungsphase des SPS-Programms aufgerufen werden)

- Funktionsblock:



Init_Cam_Gear *Cam Tracks*

Initialisiert die Nockenschaltwerksfunktion

Siehe auch: Track

- Funktionsblock:

- Variablen:

modus BYTE;
Eigenschaft der Nockenschaltwerksfunktion:
0 = ohne
1 = für Rundbewegungen, die Nocken wiederholen sich außerhalb des festgelegten Bereiches (Beispiel: Slave als rotierendes Messer)
2 = für Bewegungen zwischen 2 Endpunkten, die Nocken werden nur innerhalb des festgelegten Bereiches

ausgegeben (Beispiel: Slave als fliegende Säge)

axis BYTE;
0 = x-Achse (Master: Simulation, virtuelle Achse)
1 … 11 = y-Achse 1 … 11 (Slave)

interval_start, *interval_end* DINT;
Markieren die Positionen, an denen der Zählbereich der Nockenschaltwerksfunktion beginnt bzw. endet (Bedingung: *interval_start* < *interval_end*):
Der Zählbereich wird für die Nockenschaltwerksfunktion auf ein Raster mit maximal 8192 Schritten umgesetzt. Liefert die Differenz *interval_end* – *interval_start* mehr Zählschritte als die maximale Schrittzahl des Rasters, so wird diese Differenz auf die 8192 Schritte verteilt (siehe Beispiel beim FB Track weiter unten).

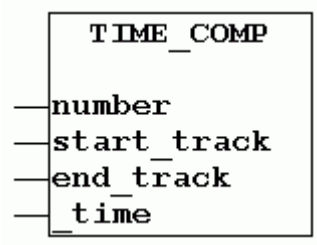
out_var_pointer POINTER_TO_WORD;
Gibt die Adresse der Variablen an, in die die Zustände der Nockenbahnen geschrieben werden.

out_hardware BYTE;
Gibt die Anzahl der Hardwareausgänge vor, an denen die Schaltzustände der Bahnen automatisch vom Betriebssystem ausgegeben werden sollen (diese Ausgänge dürfen dann nicht von anderen FBs verwendet werden):
0 = keine Ausgabe an Klemmen
1…15: Bahn 0…14 an Klemmen Q1.0…Q3.4 (fortlaufend; Beispiel: 2 ⇒ Bahn 0 an Q1.0 und Bahn 2 an Q1.1)

Time_Comp *Cam Tracks*

Hiermit können für eine Gruppe von Bahnen alle Schaltpunkte um eine bestimmte Zeit vorverlegt werden, um so z. B. Totzeiten zu kompensieren.

- Funktionsblock:



- Variablen:

number BYTE;
Nummer der Gruppe: 0…3

start_track BYTE;
erste Bahn der Gruppe

start_track BYTE;
letzte Bahn der Gruppe

_time TIME;
Kompensationszeit

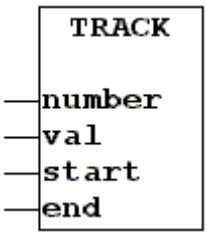
Track *Cam Tracks*

Setzen / Rücksetzen der Schaltpunkte einer Bahn

Einmal definierte Nocken bleiben dauerhaft gespeichert. Dies kann dazu führen, dass bei Definition einer neuen Nocke nicht der gewünschte Schaltbereich erscheint, weil sie sich vielleicht mit bereits vorher einmal definierten Nocken überschneidet. Um sicher zu gehen, dass keine "vergessene" Nocke den gewünschten Schaltbereich negativ beeinflusst, sollten in der Initialisierungsphase des Programms alle Bahnen zurückgesetzt werden: FB Clear_Tracks.

Siehe auch: Init_Cam_Gear

- Funktionsblock:



- Variablen:

number BYTE;
Nummer der Bahn, auf die die nachfolgend definierte Nocke wirkt: 0…15

val BOOL;
FALSE = Rücksetzen der Schaltpositionen in der vorgegebenen Bahn
TRUE = Setzen der Schaltpositionen in der vorgegebenen Bahn

start DINT;
Anfangsposition:
Gibt die Position einer Achse vor, an der eine Nocke einschaltet. Wenn diese Position unterschritten wird, schaltet die Nocke wieder aus.

Die Anfangsposition wird intern auf einen Rasterschritt umgerechnet (siehe FB Init_Cam_Gear). Die Schaltgenauigkeit bezogen auf

die vorgegebene Position hängt von der Anzahl der Zählschritte innerhalb eines Rasterschrittes ab (siehe Beispiel).

end DINT;
Endposition:
Gibt die Position einer Achse vor, bei deren Überschreiten die Nocke ausschaltet. Sie schaltet wieder ein, wenn die Achsposition die Endposition von oben erreicht.

Die Anfangsposition wird intern auf einen Rasterschritt umgerechnet (siehe FB Init_Cam_Gear). Die Schaltgenauigkeit bezogen auf die vorgegebene Position hängt von der Anzahl der Zählschritte innerhalb eines Rasterschrittes ab (siehe Beispiel).

- Beispiel:

  Deklaration:

```
cam_gear_out: WORD; (* enthält die Schaltzustände *)
Init_Cam: Init_Cam_Gear;
Set_Track: Track;
```

  Programm in ST:

```
Init_Cam.modus:= 1; (* Rundbewegung, geschl. Zählbereich *)
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); (* Klemmen Q1.0&1 *)
Set_Track(number:= 0, val:= TRUE, start:= 10000, end:=
                                        20000); (* Bahn 0 *)
Set_Track(number:= 1, val:= TRUE, start:= 5000, end:=
                                        15000); (* Bahn 1 *)
```

  Der einmalige Aufruf dieses Programms schaltet die Nockenschaltwerksfunktion für eine Rundbewegung (geschlossener Zählbereich) ein, d. h. die im Bereich 0 bis 20000 vorgegebenen Signale werden in den Bereichen zwischen 20000 und 40000, 40000 und 60000 usw. der Masterachse (Achse 0) wiederholt. Für die direkte Ausgabe der Bahnen 0 und 1 werden die beiden Ausgänge Q1.0 und Q1.1 aktiviert. Die Variable `cam_gear_out` enthält die Schaltzustände in den Bits 2^0 (Bahn 0) und 2^1 (Bahn 1), 1 = Ein.

  Da der Zählbereich mit 20000 größer ist als die maximale Schrittzahl des Rasters mit 8192, enthält ein Rasterschritt 2 oder 3 Zählschritte (20000/8192 ≈ 2,4).

  In diesem Beispiel liefern die Ausgänge Signale wie sie üblicherweise auch von inkrementalen Gebern vorgegeben werden (0°- und 90°-Spur). Über die Ausgänge könnte somit ein Zähler angesteuert werden.

4.4 CAN-Bus

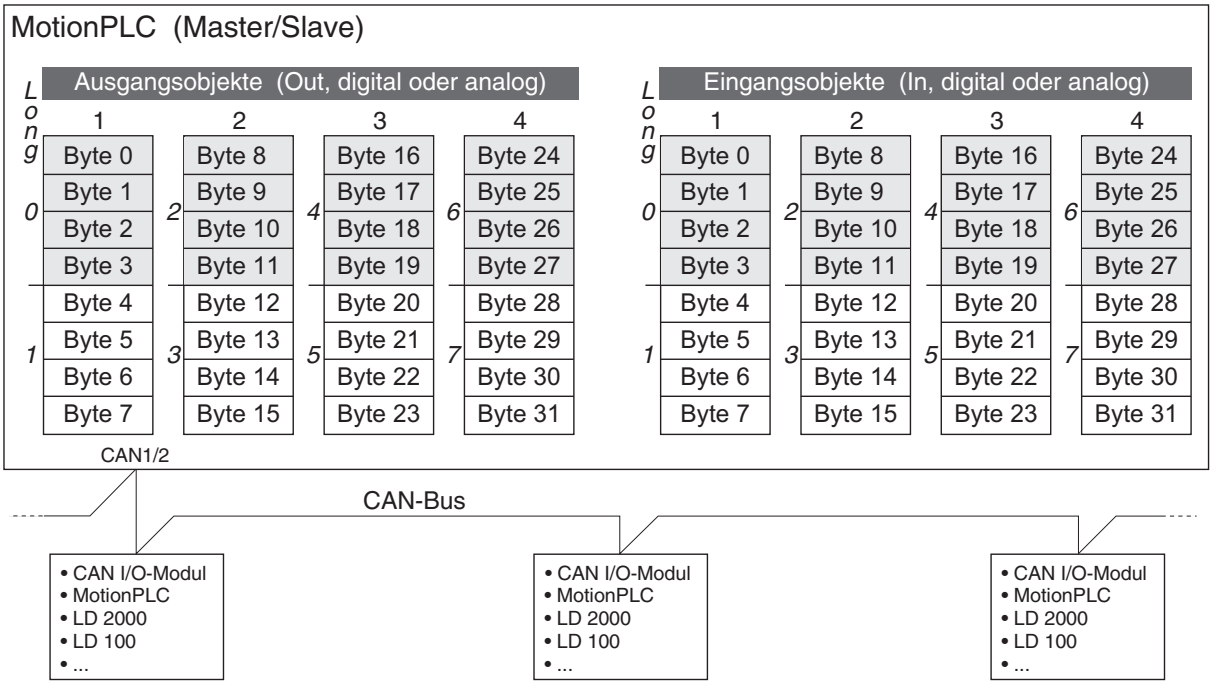
Die FBs in diesem Zweig sind für die Standardkommunikation nach dem CANopen-Protokoll über den ersten CAN-Bus vorgesehen (zweiter CAN-Bus: siehe Abschnitt 4.4.1). Welcher der beiden elektrischen Anschlüsse dazu verwendet wird, ist in para[205] festgelegt (siehe Betriebsanleitung). Es können folgende Aktionen ausgeführt werden:

- Eingänge über CAN-Bus lesen (CAN_In_…)
- Ausgänge über CAN-Bus schreiben (CAN_Out_…)
- Ausgänge über CAN-Bus lesen (Rd_…)
- CAN-Bus initialisieren, zurücksetzen und Zustandsabfrage (CAN_Init/-Reset/Status)
- SDO-Datenkanäle lesen/schreiben (SDO_…)

Die CAN-Ein/Ausgänge werden mit einer maximalen Verzögerung von 10 ms nach dem Aufruf des jeweiligen FBs gelesen/geschrieben (SDO_Request wird allerdings sofort ausgeführt).

Die MotionPLC arbeitet mit einer festen Baudrate von **500 kBaud** (Ausnahme: siehe Betriebsanleitung; CAN Link: 250 kBaud). Die angeschlossenen CAN-Bausteine müssen ebenfalls auf diese Baudrate eingestellt sein und es muss eine Objekt-Adresse eingestellt werden (1…127, Standard: 1 … 4; siehe Betriebsanleitung zur MotionPLC: CAN-Parameter para[55] ff).

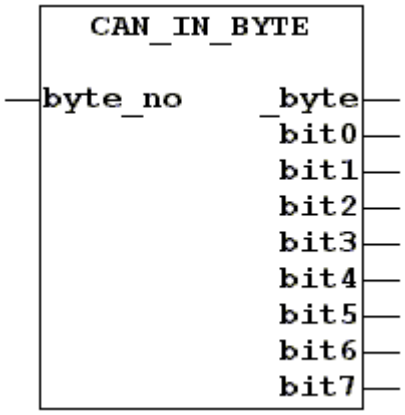
Die folgende Grafik informiert über die Zuordnung der Ein- und Ausgangsbytes zu den einzelnen Objekten:



CAN_In_Byte *CAN Bus*

Liefert Zustände eines Eingangsobjekt-Bytes als Byte und Bits

- Funktionsblock:



- Variablen:

byte_no BYTE;
Wahl des Eingangs-Bytes
0 … 7: Objekt 1
8 … 15: Objekt 2
16 … 23: Objekt 3
24 … 31: Objekt 4

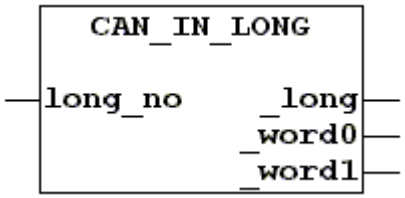
_byte BYTE;
Inhalt des Eingangs-Bytes

bit0…7 BOOL;
Zustände der Bits 0…7 innerhalb von *_byte*

CAN_In_Long *CAN Bus*

Liefert Zustände eines Eingangsobjekt-Longs als Long und Words

- Funktionsblock:



- Variablen:

long_no BYTE;
Wahl des Eingangs-Longs
0, 1: Objekt 1

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

_long DINT;
Inhalt des Eingangs-Longs

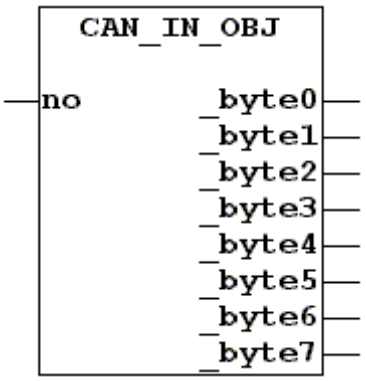
word0 WORD;
Inhalt des niederwertigen Words (LSW) in *_long*

word1 WORD;
Inhalt des höherwertigen Words (MSW) in *_long*

CAN_In_Obj *CAN Bus*

Liefert die Zustände der 8 Bytes eines Eingangsobjekts

- Funktionsblock:



- Variablen:

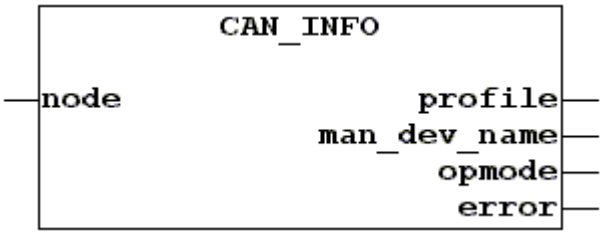
no BYTE;
Wahl des Eingangs-Objekts
1 … 4: Objekt 1…4

byte0…7 BYTE;
Inhalt der Bytes 0…7 innerhalb des Eingangsobjekts

CAN_Info *CAN Bus*

Informationen zu einem angeschlossenen Gerät abfragen

- Funktionsblock:



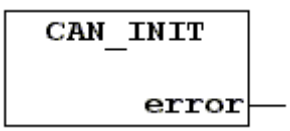
- Variablen:

node	BYTE; 1 … 16: CAN-Adresse
profile	WORD; Gerätetyp (festgelegt nach CANopen) 191h: CAN Remote I/O 192h: Servoumrichter
man_dev_name	DWORD; Hersteller-Gerätename
opmode	BYTE; Betriebsart (nur bei LD 2000: Parameter OPMODE)
error	BYTE; CAN-Bus-Fehler während der Initialisierungsphase (0 = kein Fehler)

CAN_Init *CAN Bus*

Initialisierung des CAN-Controllers nach einer Umprogrammierung von Daten (z. B. Adresse); es wird der CAN-Loader aufgerufen (erfolgt auch automatisch mit Aufruf des FBs Loader). Das Bit `new_dat` des geänderten Objekts wird zurückgesetzt (siehe CAN_Status weiter unten).

- Funktionsblock:



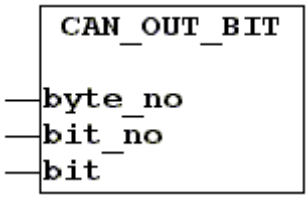
- Variablen:

error	BYTE; 0 = kein Fehler, andernfalls Nummer des fehlerhaften Parameters

CAN_Out_Bit *CAN Bus*

Setzt ein bestimmtes Ausgangsobjekt-Bit auf 0 (FALSE) oder 1 (TRUE)

- Funktionsblock:



- Variablen:

byte_no BYTE;
Wahl des Ausgangs-Bytes
0 … 7: Objekt 1
8 … 15: Objekt 2
16 … 23: Objekt 3
24 … 31: Objekt 4

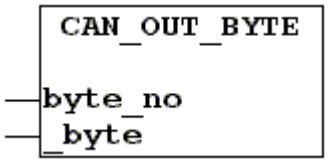
bit_no BYTE;
Bit innerhalb des Ausgangs-Bytes: 0…7

bit0…7 BOOL;
auszugebende Information: TRUE/1 oder FALSE/0

CAN_Out_Byte *CAN Bus*

Setzt ein bestimmtes Ausgangsobjekt-Byte

- Funktionsblock:



- Variablen:

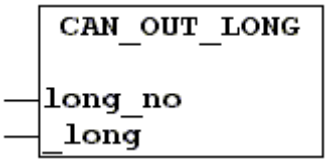
byte_no BYTE;
Wahl des Ausgangs-Bytes
0 … 7: Objekt 1
8 … 15: Objekt 2
16 … 23: Objekt 3
24 … 31: Objekt 4

_byte BYTE;
auszugebende Information

CAN_Out_Long *CAN Bus*

Setzt ein bestimmtes Ausgangsobjekt-Long

- Funktionsblock:



- Variablen:

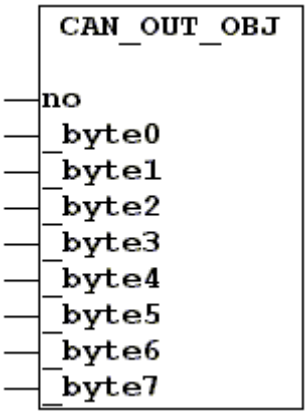
long_no BYTE;
Wahl des Ausgangs-Longs
0, 1: Objekt 1
2, 3: Objekt 2
4, 5: Objekt 3
6, 7: Objekt 4

_long DINT;
auszugebende Information

CAN_Out_Obj *CAN Bus*

Setzt ein bestimmtes Ausgangsobjekt

- Funktionsblock:



- Variablen:

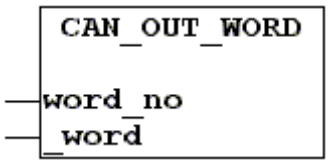
no BYTE;
Wahl des Ausgangsobjekts
1 … 4: Objekt 1…4

byte0…7 BYTE;
auszugebende Information

CAN_Out_Word *CAN Bus*

Setzt ein bestimmtes Ausgangsobjekt-Word

- Funktionsblock:



- Variablen:

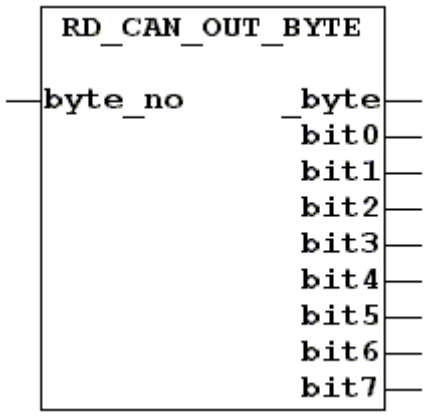
word_no BYTE;
Wahl des Ausgangs-Word
0 … 3: Objekt 1
4 … 7: Objekt 2
8 … 11: Objekt 3
12 … 15: Objekt 4

_word WORD;
auszugebende Information

Rd_CAN_Out_Byte *CAN Bus*

Liest die Zustände bestimmter Ausgangsobjekte als Byte und Bits

- Funktionsblock:



- Variablen:

byte_no BYTE;
Wahl des zu lesenden Ausgangs-Bytes
0 … 7: Objekt 1
8 … 15: Objekt 2
16 … 23: Objekt 3
24 … 31: Objekt 4

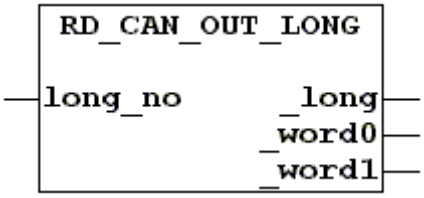
_byte BYTE;
zu lesende Information

bit0…7 BOOL;
Zustände der Bits 0…7 von *_byte*

Rd_CAN_Out_Long *CAN Bus*

Liest die Zustände bestimmter Ausgangsobjekte als Long und Words

- Funktionsblock:



- Variablen:

long_no BYTE;
Wahl des zu lesenden Ausgangs-Longs
0, 1: Objekt 1
2, 3: Objekt 2
4, 5: Objekt 3
6, 7: Objekt 4

_long DINT;
Inhalt des gewählten Ausgangs-Longs

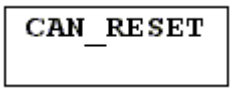
word0 WORD;
Inhalt des niederwertigen Words (LSW) von *_long*

word1 WORD;
Inhalt des höherwertigen Words (MSW) von *_long*

CAN_Reset *CAN Bus*

Das Bit *new_dat* aller CAN-Objekte wird zurückgesetzt (siehe CAN_Status).
Dieser FB sollte nach Erkennen eines Busfehlers aufgerufen werden.

- Funktionsblock:



CAN_Status *CAN Bus*

Liefert Informationen über einen erfolgten Datentransfer. Der Status wird nach Aufruf des FBs zurückgesetzt.

- Funktionsblock:

- Variablen:

no BYTE;
0 = Synchronisationstelegramm (CAN-SYNC)
1…4 = Eingangsdaten (CAN-IN)
5…8 = Ausgangsdaten (CAN-OUT)

new_dat BOOL;
TRUE = neue Daten angekommen (CAN-IN) / ausgegeben (CAN-OUT)

bus_off BOOL;
TRUE = Busfehler liegt vor (siehe auch Betriebsanleitung unter Terminalbetrieb: Befehl "re"); es muss eine elektrische Verbindung bestehen. Bei TRUE kann z. B. ein Kurzschluss vorliegen, oder es sind mehrere CAN-Module mit identischen Adressen bei CAN-Out-Objekten konfiguriert oder gleichzeitig als Master.

SDO_Request *CAN Bus*

Abruf von Daten eines Servers (Slave), hier Servoumrichter

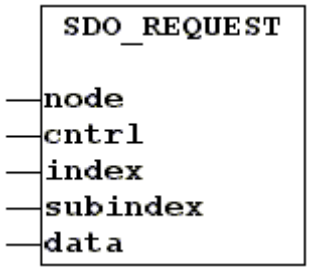
Für die CANopen-Kommunikation mit dem Servoumrichter LD 2000 wurden unter CoDeSys spezielle FBs in der mitgelieferten Bibliothek **ld2000.lib** zusammengefasst. Diese ist als interne Bibliothek aufgebaut und kann – wenn über *Datei/Öffnen* geladen – verändert und an die eigenen Bedürfnisse angepasst werden.

Wichtig: Vor jeder Neukompilierung von ld2000.lib müssen die Bibliotheken standard.lib und gel8240.lib über den Bibliotheksverwalter eingebunden werden.

Die FBs sind weitgehend selbsterklärend, so dass auf weitere Erläuterungen an dieser Stelle verzichtet wird. Ein Beispielprogramm für die Anwendung der verschiedenen FBs befindet sich im CoDeSys-Verzeichnis unter *Examples_Ld100*: ld2000_demo.pro.

Für eine Erweiterung der Bibliothek ist die genaue Kenntnis der ASCII-Objektcodes des Servoumrichters und deren Bedeutung erforderlich (Dokumentation dazu auf der mitgelieferten CD: ba_can_d.pdf).

- Funktionsblock:



- Variablen:

node BYTE;
CAN-Adresse, 1…127

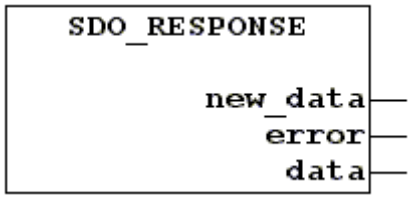
cntrl BYTE;
Datenrichtung:
34 [DOWNLOAD*] = Daten an den Server (CAN-Slave) senden
64 [UPLOAD*] = Daten vom Server anfordern

index (WORD), *subindex* (BYTE), *data* (DINT) siehe CANopen-Dokumentation zum Servoumrichter

SDO_Response *CAN Bus*

Rückmeldung vom Server (Slave)

- Funktionsblock:



- Variablen:

new_data BOOL;
TRUE = neue Daten vorhanden bzw. Server hat geantwortet

error BYTE;
0 = kein Fehler

data DINT *(siehe CANopen-Dokumentation)*

* Bei den globalen Variablen definierte Konstante

4.4.1 CAN2

Die FBs dieser Kategorie sind speziell für die Steuerung von Servoumrichtern des Typs LD 2000 über den zweiten CAN-Bus vorgesehen. Welcher der beiden elektrischen Anschlüsse dazu verwendet wird, ist in para[205] festgelegt (siehe Betriebsanleitung). Der CAN-Bus steht dann nicht mehr für die CAN-Link Funktionen (C-Net) des Abschnitts 4.5 zur Verfügung.

CAN2_BusInit *CAN Bus (CAN2)*

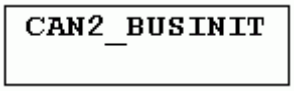
Initialisiert den für die Regelung externer Achsen verwendeten CAN-Bus erneut (wie dies sonst beim Einschalten der MotionPLC erfolgt). Damit kann im Falle eines mit einem Reset verbundenen Fehlers in einem Umrichter (wie er häufiger gerade während der Inbetriebnahmephase auftreten kann) ein erneutes Aus- und Einschalten der MotionPLC umgangen werden.

Nach einer solchen Initialisierung muss der betreffende Servoumrichter zuerst referenziert werden, wenn das SPS-Programm diesen Vorgang nach dem Einschalten auch durchführt.

Beim Aufruf des Funktionsblocks sollten alle CAN-Achsen gesperrt d. h. disabled sein.

Der FB darf nicht verwendet werden, wenn sich Achsen mit absoluter Istwerterfassung am CAN-Bus befinden.

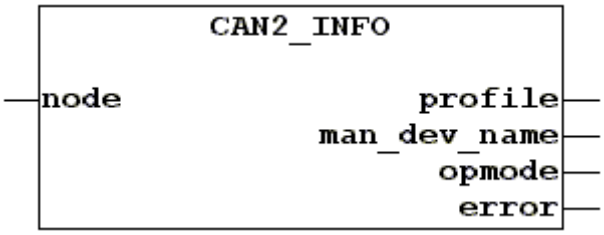
- Funktionsblock:



CAN2_Info *CAN Bus (CAN2)*

Wie FB CAN_Info (→ S. 48)

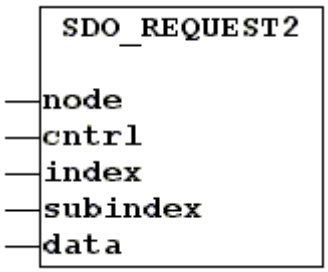
- Funktionsblock:



SDO_Request2 *CAN Bus (CAN2)*

Funktionalität wie SDO_Request weiter oben (siehe dort)

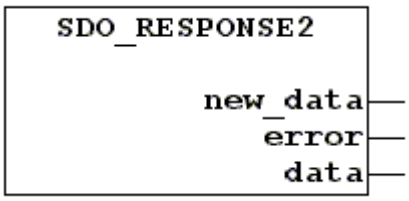
- Funktionsblock:



SDO_Response2 *CAN Bus (CAN2)*

Funktionalität wie SDO_Response weiter oben (siehe dort)

- Funktionsblock:



4.5 CAN Link (C-Net)

Mit den FBs dieser Kategorie kann ein CAN-Netzwerk (System-CAN-Bus) mit einem freien Protokoll aufgebaut werden, das also nicht den CANopen-Konventionen unterliegt. Hardwaremäßig wird dazu entweder das CAN1- oder das CAN2-Interface verwendet, je nach Programmierung des Systemparameters para[205] (siehe Betriebsanleitung).

Es stehen 14 Ein-/Ausgabekanäle zur Verfügung. Es darf immer nur jeweils **1 Gerät pro Kanal senden**, wogegen die Zahl der Empfänger pro Kanal beliebig ist. Es kann auch 1 Gerät auf mehreren Kanälen senden.

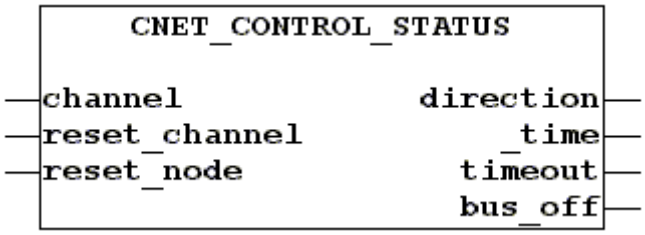
Es können folgende Aktionen ausgeführt werden:

- Bus-Status lesen (CNET_Control_Status)
- Zyklischen Datenaustausch initiieren (CNET_Start)
- Datenobjekte senden und empfangen ("Basic Jobs": CNET_Out_Obj und CNET_In_Ob)

CNET_Control_Status *CAN Link*

Gibt Auskunft über den Zustand des CAN-Netzwerks und steuert dieses in Verbindung mit dem FB CNET_Start

◆ Funktionsblock:



◆ Variablen:

channel	BYTE; Ein-/Ausgabekanal: 1…14
reset_channel	BOOL; TRUE = angegebener Kanal wird deaktiviert
reset_node	BOOL; TRUE = CAN-Netzwerk-Teilnehmer wird zurückgesetzt (Aufruf kann z. B. nach Erkennen eines Busfehlers sinnvoll sein)
direction	BYTE; 0 = inaktiv 1 = senden 2 = empfangen
_time	TIME; Zeitdauer seit der letzten Übertragung

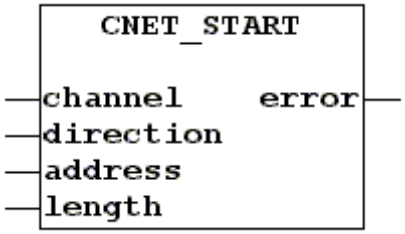
timeout BOOL;
TRUE = Zeit für eine (weitere) Übertragen wurde überschritten: 200 ms (die zyklische Übertragung erfolgt typisch alle 70 ms)

bus_off BOOL;
TRUE = CAN-Netzwerk-Busfehler liegt vor
Es muss eine elektrische Verbindung bestehen. Bei TRUE kann z. B. ein Kurzschluss vorliegen.

CNET_Start *CAN Link*

Durch einmaligen Aufruf des FB kann ein ständig und im Hintergrund laufender Datenaustausch gestartet werden

- Funktionsblock:



- Variablen:

channel BYTE;
Ein-/Ausgabekanal: 1…14

direction BYTE;
1 = senden
2 = empfangen

address POINTER TO BYTE;
Startadresse eines Speicherbereichs (Variable, Array), in den z. B. die Sollpositionen, Istpositionen, Istgeschwindigkeiten etc. verschiedener Antriebe abgelegt werden

length BYTE;
Anzahl der zu übertragenen Bytes: max 70

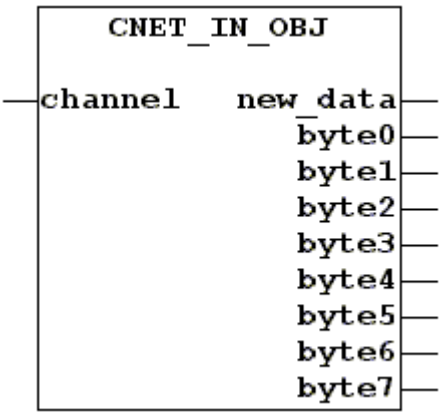
error BYTE;
Fehlerzustände:
0 = kein Fehler
1 = ungültige Ein-/Ausgabekanal-Nummer
2 = Kanal benutzt
3 = ungültige Datenrichtung
4 = ungültige Adresse oder Datenlänge

4.5.1 Elementare Funktionen (Basic Jobs)

CNET_In_Obj *CAN Link (Basic Jobs)*

Liest ein Datenobjekt (8 Bytes) vom CAN-Netzwerk

- Funktionsblock:



- Variablen:

channel BYTE;
Ein-/Ausgabekanal: 1…14

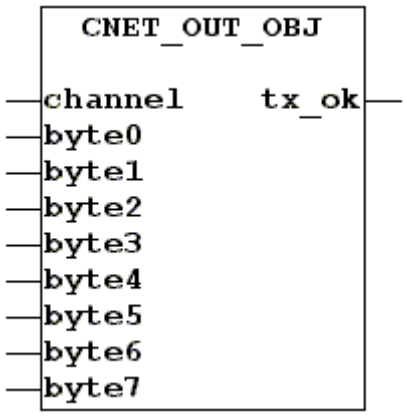
new data BOOL;
TRUE = neue Daten angekommen; wird beim Lesen der Daten zurückgesetzt (FALSE)

byte0…7 BYTE;
zu lesende Daten

CNET_Out_Obj *CAN Link (Basic Jobs)*

Schreibt ein Datenobjekt (8 Bytes) ins CAN-Netzwerk

- Funktionsblock:



- Variablen:

  channel BYTE;
  Ein-/Ausgabekanal: 1…14

  byte0…7 BYTE;
  zu sendende Daten

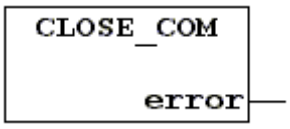
  tx_ok BOOL;
  FALSE = Timeout (keine Empfangsquittierung auf dem Bus); das Objekt konnte nicht innerhalb von 10 ms gesendet werden

4.6 Kommunikation (RS232/RS485)

Close_COM *Communication*

Schließt die serielle Schnittstelle; dabei werden deren Kommunikationsparameter auf die Standardwerte für die Arbeit mit dem PC und CoDeSys gesetzt, wie sie mit para[207/208] festgelegt sind (siehe Betriebsanleitung).

- Funktionsblock:



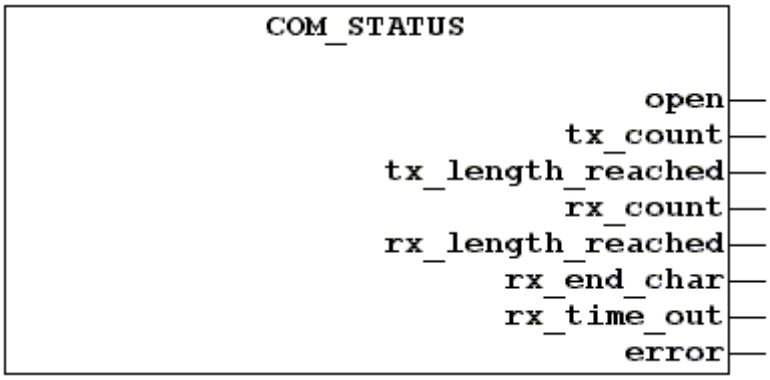
- Variablen:

error BYTE;
Fehlerzustand:
0 = kein Fehler
4 = Schnittstelle ist bereits geschlossen

COM_Status *Communication*

Über diesen FB erfolgt die Steuerung des Sende- und Empfangsablaufs. Er sollte immer jeweils vor einem Sende- bzw. nach einem Empfangsbefehl aufgerufen werden, um dann gezielt auf die gesetzten Variablen zu reagieren.

- Funktionsblock:



- Variablen:

open BOOL;
TRUE = geöffnet
FALSE = geschlossen

tx_count BYTE;
Anzahl der bisher gesendeten Zeichen;
wird mit jedem Aufruf des FB Transmit zurückgesetzt

tx_length_reached	BOOL; TRUE = Anzahl der gesendeten Zeichen hat den im FB Transmit vorgegebenen Wert *length* erreicht
rx_count	BYTE; Anzahl der bisher empfangenen Zeichen; wird mit jedem Aufruf des FB Receive zurückgesetzt
rx_length_reached	BOOL; TRUE = Anzahl der empfangenen Zeichen hat den im FB Receive vorgegebenen Wert *length* erreicht
rx_end_char	BOOL; TRUE = zuletzt empfangenes Zeichen ist das im FB Receive vorgegebene Endzeichen, falls aktiviert (siehe dort); wenn dies nicht der Fall ist, aber ein Endzeichen unter *end_char* angegeben wurde, dann bedeutet TRUE, dass dieses Zeichen mindestens einmal übertragen worden ist
rx_time_out	BOOL; TRUE = die im FB Receive vorgegebene Time-out-Zeit ist abgelaufen
error	BYTE; Fehlerzustand: 0 = kein Fehler 1 = Parity error 2 = Framing error 3 = Overflow error 4 = Schnittstelle nicht geöffnet

Open_COM *Communication*

Dieser FB muss vor dem ersten Kommunikationsvorgang aufgerufen werden (gefolgt vom FB Receive). Es werden die Schnittstellenparameter auf das verwendete Peripheriegerät eingestellt (nach Schließen der Schnittstelle wird diese wieder auf Standardwerte eingestellt, siehe Close_COM).

- Funktionsblock:

- Variablen:

baudrate WORD;
Übertragungsgeschwindigkeit:
2400 / 4800 / 9600 / 19200 / 38400 / 57600 [Baud]

data_bits BYTE;
Anzahl der Datenbits: 7 / 8

parity BYTE;
Parität:
0 = keine (none)
1 = ungerade (odd)
2 = gerade (even)

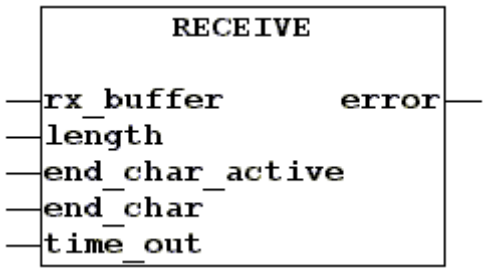
profile BYTE;
Kommunikationsprotokoll:
0 = freies Protokoll an COM1
1 = freies Protokoll an COM2

error BYTE;
Fehlerzustand:
0 = kein Fehler
4 = ungültiger Schnittstellenparameter

Receive *Communication*

Dieser FB sollte direkt nach dem Öffnen der Schnittstelle aufgerufen werden, um die Kommunikation für den zu erwartenden Datenempfang vorzubereiten.

- Funktionsblock:



- Variablen:

rx_buffer POINTER TO BYTE;
Anfangsadresse des als Lesepuffer zu verwendenden Speicherbereichs (Variable, Array)

length BYTE;
Anzahl der erwarteten Zeichen (Datenlänge); alternativ oder zusätzlich zur Verwendung eines Daten-Endzeichens oder der Definition eines Time-out

Der interne Kommunikationspuffer ist auf 255 Zeichen begrenzt!

end_char_active BOOL;
Daten-Endzeichen verwendet? – TRUE / FALSE (alternativ oder zusätzlich zur Nutzung einer Datenlänge oder eines Time-out; Endzeichen hat Vorrang);
bei FALSE und Angabe eines *end_char* kann über den FB COM_Status der empfangene Datenblock auf das Vorhandensein des angegebenen Endzeichens untersucht werden

end_char BYTE;
ASCII-Code des verwendeten Endzeichens (z. B. Carriage Return [CR] = 13)

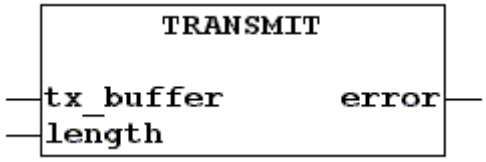
time_out WORD;
maximale Übertragungsdauer in ms (alternativ oder zusätzlich zur Nutzung einer Datenlänge oder eines Endzeichens); 0 = inaktiv

error BYTE;
Fehlerzustand:
0 = kein Fehler
1 = Parity error
2 = Framing error
3 = Overflow error
4 = Schnittstelle nicht geöffnet
5 = ungültige Adresse (Pointer)

Transmit *Communication*

Daten senden

- Funktionsblock:



- Variablen:

tx_buffer POINTER TO BYTE;
Anfangsadresse des zu sendenden Speicherbereichs (Variable, Array)

length BYTE;
Anzahl der zu sendenden Zeichen (Datenlänge, ≤ 255 Zeichen)

error	BYTE; Fehlerzustand: 0 = kein Fehler 1 = Parity error 2 = Framing error 3 = Overflow error 4 = Schnittstelle nicht geöffnet

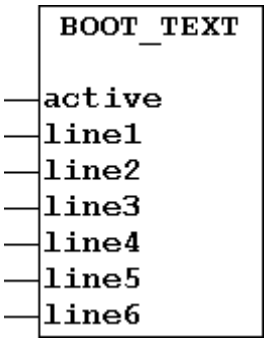
4.7 Anzeige und Tastatur (Display and Keyboard)

Boot_Text *Display and Keyboard*

Speichert den definierten Textblock stromausfallsicher für die Anzeige beim nächsten Boot-Vorgang

Siehe auch: Clr_Tscreen, Write_BStr, Write_Str

- Funktionsblock:



- Variablen:

active BOOL;
Freigabe des Textblocks für den Boot-Vorgang:
TRUE = Textblock wird beim Einschalten des Gerätes angezeigt;
FALSE = Textblock wird beim Einschalten des Gerätes nicht angezeigt, sondern der werksseitig programmierte Standard-Boot-Block, der aus Grafik- und Textelementen besteht

line1; *line2* STRING(20);
Zeilen 1 und 2 des Textblocks
Diese beiden Zeilen werden in großer Schrift (12x16 Pixel) mit maximal 20 Zeichen angezeigt.

line3…6 STRING(40);
Zeilen 3 bis 6 des Textblocks
Diese vier Zeilen werden in kleiner Schrift (6x8 Pixel) mit maximal 40 Zeichen angezeigt.

- Beispiel:

Deklaration:

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

Programm 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;
```

Das einmalige Ausführen dieses Programms bewirkt, dass die String-Variablen im stromausfallsicheren Variablenbereich gespeichert und bei jedem erneuten Einschalten des Gerätes direkt angezeigt werden. Die Anzeige erfolgt auch dann, wenn sich das Programm nicht mehr im Gerät befindet.

Erst nach einigen Sekunden ist der Boot-Vorgang abgeschlossen und das gespeicherte SPS-Programm freigegeben, über das dann bei Bedarf die Anzeige verändert werden kann.

Clr_GScreen / Clr_TScreen *Display and Keyboard*

Löscht alle Grafikelemente / Textelemente im Display

Siehe auch: Boot_Text, Clr_Point; Line, Put_Point, Write_BGStr; Write_BStr; Write_Str

- Funktionsblock:

- Beispiel:

  Deklaration:

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

  Programm in ST:

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

  Mit der positiven Flanke von `key_new` werden die FBs einmalig aufgerufen. Dabei bewirkt der FB `Clr_GScreen` das Löschen aller mit `Line`, `Put_Point` und `Write_BGStr` erstellten Elemente. Der FB `Clr_TScreen` löscht alle mit `Boot_Text`, `Write_BStr` und `Write_Str` erstellten Texte.

  Der Standard-Startbildschirm besteht aus Grafik- und Textelementen. Um das Display komplett zu löschen, müssen also wie im Beispiel beide FBs aufgerufen werden.

Clr_Point / Put_Point *Display and Keyboard*

Löscht / schreibt einzelne Grafik-Punkte im Display

Siehe auch: Clr_GScreen, Clr_TScreen

- Funktionsblock:

- Variablen:

  x BYTE;
  Spaltenposition für den zu löschenden/schreibenden Punkt im Bereich von 0 (links) bis 239 (rechts)

  y BYTE;
  Zeilenposition für den zu löschenden/schreibenden Punkt im Bereich von 0 (oben) bis 63 (unten)

- Beispiel:

  Deklaration:

```
clr_grafic: Clr_GScreen;
clr_txt: Clr_TScreen;
axis: Line; (* Null-Linie *)
display_line: BYTE;
i: BYTE;
c_point: Clr_Point;
p_point: Put_Point;
```

  Programm 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;
```

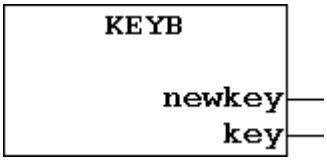
  Ein Programmdurchlauf zeichnet eine Sinuskurve. Kurvenpunkte, die auf der Null-Linie liegen, werden gelöscht.

Keyb *Display and Keyboard*

Liefert ein Byte für eine gedrückte Taste und meldet jeweils über einen SPS-Zyklus eine Zustandsänderung an den Tasten.

Siehe auch: Keyb_Val

- Funktionsblock:



- Variablen:

newkey BOOL;
Meldung bezogen auf den vorhergehenden SPS-Zyklus:
FALSE = keine Tastenzustandsänderung
TRUE = Tastenzustand geändert

key BYTE;
Tastenwert entsprechend folgender Tabelle:

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

Die grau hinterlegten Felder stehen für die 4 Tasten, die oberhalb der Anzeige hinter dem Logo-Fenster verdeckt in den gleichen Spalten wie die Tasten F1 bis F4 angeordnet sind.
Werden mehrere Tasten betätigt, liefert *key* den Wert 255.

- Beispiel:

Deklaration:

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

Programm in ST:

```
keyboard_byte();
IF keyboard_byte.newkey AND keyboard_byte.key=255 THEN
```

```
    counter:= counter + 1;
END_IF;
```

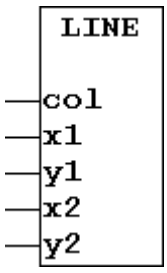
Die Variable `counter` zählt die Ereignisse, bei denen mehr als eine Taste betätigt wurden.

Line *Display and Keyboard*

Zeichnet oder löscht eine Linie

Siehe auch: Clr_GScreen, Clr_Point; Put_Point

- Funktionsblock:



- Variablen:

col BYTE;
Modus:
0 = Löschen einer Linie
1 = Zeichnen einer Linie

x1 INT;
Spaltenposition für den Linienanfang im Bereich von 0 (links) bis 239 (rechts)

y1 INT;
Zeilenposition für den Linienanfang im Bereich von 0 (oben) bis 63 (unten)

x2 INT;
Spaltenposition für das Linienende im Bereich von 0 bis 239

y2 INT;
Zeilenposition für das Linienende im Bereich von 0 bis 63

- Beispiel:

Deklaration:

```
FLine: Line;
```

Programm 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);
```

Ein Programmdurchlauf zeichnet Beschriftungsfelder für die Funktionstasten F1 bis F5.

Set_TP / Write_Str *Display and Keyboard*

Setzt die Anfangsmarke für die anschließende Textausgabe mit Write_Str an die vorgegebene Position im Display. / Schreibt einen String im Textmodus mit kleiner Schriftgröße (6x8 Pixel) in das Display ab der mit Set_TP festgelegten Position.

Siehe auch: Clr_TScreen, Write_BStr

- Funktionsblock:

- Variablen:

x BYTE;
Spaltenposition für den Textanfang im Bereich von 0 (links) bis 39 (rechts)

y BYTE;
Zeilenposition für den Text im Bereich von 0 (oben) bis 7 (unten)

_string STRING[160];
Im Display darzustellende Zeichenfolge
Überschreitet eine Zeichenfolge das Ende einer Zeile, dann wird sie in der nächsten Zeile fortgesetzt. Zeichen, die über das Ende der letzten Zeile hinaus geschrieben werden, sind nicht sichtbar.

- Beispiel:

Deklaration:

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

Programm 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);
```

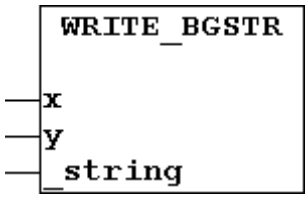
Ein Programmdurchlauf schreibt über der Funktionstaste F4 den Text 'Stop'.

Write_BGStr *Display and Keyboard*

Schreibt einen String in das Display ab der definierten Anfangsmarke. Die Ausgabe erfolgt im (langsameren) Grafikmodus mit großer Schriftgröße (12x16 Pixel).

Siehe auch: Clr_GScreen, Write_BStr, Write_Str

- Funktionsblock:



- Variablen:

x BYTE;
Spaltenposition für den Textanfang im Bereich von 0 (links) bis 38 (rechts)
Das Raster für die Spaltenposition ist die halbe Zeichenbreite von 6 Pixeln. Maximal können in eine Zeile 20 Zeichen geschrieben werden.

y BYTE;
Zeilenposition für den Text im Bereich von 0 (oben) bis 6 (unten)
Das Raster für die Zeilenposition ist die halbe Zeichenhöhe von 8 Pixeln. Maximal können in einer Spalte 4 Zeichen untereinander stehen.

_string STRING[80];
Im Display darzustellende Zeichenfolge
Überschreitet eine Zeichenfolge das Ende einer Zeile, dann wird die Anzeige der Zeichenfolge am linken Rand um ein halbes Zeichen nach unten verschoben fortgesetzt. Zeichen, die über das Ende der letzten Zeile hinaus geschrieben werden, erscheinen nur mit der oberen Zeichenhälfte am unteren Rand des Displays.

- Beispiel:

Deklaration:

```
clr_grafic: Clr_GScreen;
clr_txt: Clr_TScreen;
column: BYTE;
Text_BG: Write_BGStr;
```

```
txt_length: BYTE;
txt_str: STRING[20];
```

Programm 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);
```

Ein Programmdurchlauf schreibt mittig in die obere Zeile den Text 'Lenord + Bauer'.

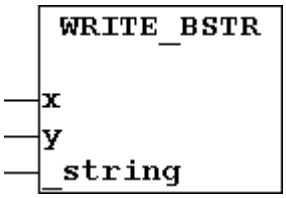
Write_BStr *Display and Keyboard*

Schreibt einen String in das Display ab der definierten Anfangsmarke. Die Ausgabe erfolgt im (schnelleren) Textmodus mit großer Schriftgröße (12x16 Pixel). Der Zeichensatz ist eingeschränkt auf folgende Zeichen:

```
! " # % & ( ) * + , - . / ?
0 1 2 3 4 5 6 7 8 9 : ; < = >
```

Siehe auch: Clr_TScreen, Write_BGStr, Write_Str

- Funktionsblock:



- Variablen: wie Write_BGStr (siehe dort)
- Beispiel:

Deklaration:

```
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;
```

Programm 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);
```

Das Programm schreibt den Istwert der Achse 1 mit zwei Stellen nach dem Dezimalpunkt und einer Länge von insgesamt 6 Zeichen einschließlich Punkt und ggf. Minuszeichen rechtsbündig in die Anzeige. Werden mehr als 6 Zeichen für die Anzeige des Wertes benötigt, dann werden nur die 6 linken (höherwertigen) Zeichen angezeigt.

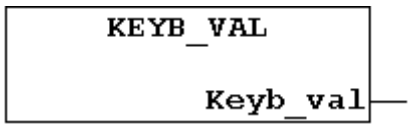
4.7.1 Elementare Funktionen (Basic Jobs)

Keyb_Val *Display and Keyboard (Basic Jobs)*

Stellt gedrückte Tasten entsprechend der Anordnung in der Tastaturmatrix dar

Siehe auch: Keyb

- Funktionsblock:



- Variablen:

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 grau hinterlegten Felder stehen für die 4 Tasten, die oberhalb der Anzeige hinter dem Logo-Fenster verdeckt in den gleichen Spalten wie die Tasten F1 bis F4 angeordnet sind.
Werden mehr als zwei Tasten betätigt, dann kann dies zu einem falschen Abbild führen.

- Beispiel:

Deklaration:

```
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;
```

Programm 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
```

Zunächst erfolgt die Abfrage der Tasten durch Aufruf des FB. Alle sichtbaren Tasten werden in der Variablen `keyb_val_long` zusammengeführt. Jede Tastenänderung führt dazu, dass für einen Zyklus die Variable `key_puls` den Zustand der Tasten und die Variable `key_changed` die Information TRUE liefert. Die Variable `key_puls_0` liefert nur dann den Zustand der Tasten, wenn zuvor keine Taste gedrückt war.

Die Variable `counter1` wird mit der Tastenkombination F1+▲ inkrementiert bzw. mit F1+▼ dekrementiert. Die Variable `counter2` wird auf die gleiche Weise mittels F2+▲/▼ verändert.

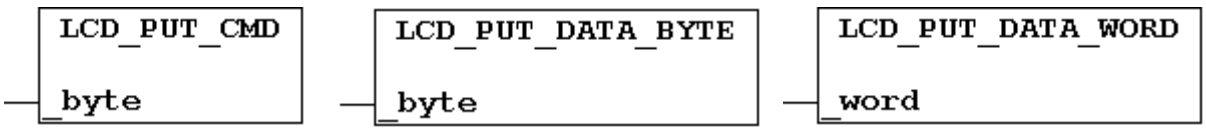
Lcd_Put_Cmd / Lcd_Put_Data_Byte / Lcd_Put_Data_Word

Display and Keyboard (Basic Jobs)

Direkte Ansteuerung des 240x64-Displays über den Controller T6963C der Firma Toshiba

Siehe auch: Boot_Text, Clr_GScreen, Clr_Point, Clr_TScreen, Line, Put_Point, Set_TP, Write_BGStr, Write_BStr, Write_Str

- Funktionsblock:



Diese FBs bieten dem versierten Anwender die Möglichkeit, annähernd alle Funktionen des Displays zu nutzen, die in der Spezifikation des Controllerherstellers aufgeführt werden.

4.8 Feldbus (FieldBus)

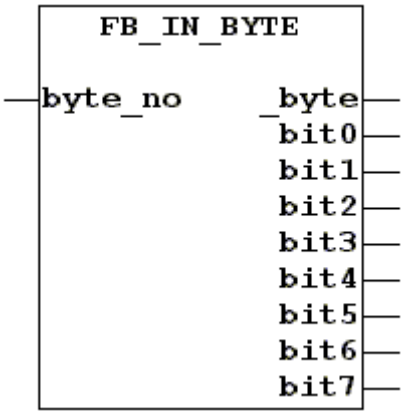
Mit den FBs dieser Kategorie können Daten über ein in die MotionPLC eingesetztes Feldbus-Modul übertragen werden (PROFIBUS-DP, InterBus-S, DeviceNet oder Ethernet). Das Datenformat und die Datengröße werden bei den Feldbus-Systemparametern festgelegt (siehe Betriebsanleitung).

Ethernet-Modul:
Für die in diesem Abschnitt behandelten FBs wird der Modbus-Server verwendet, Port 502. Datenformat und -größe werden mit den DeviceNet-Parametern (para[321+322]) festgelegt.
Die FBs in Unterabschnitt 4.8.1 verwenden den FTP-Server auf dem Modul. Die erforderliche IP-Adresse muss den eigenen Bedürfnissen entsprechend festgelegt werden (siehe Dokumentation zu den Feldbus-Modulen).

FB_In_Byte *FieldBus*

Datenbyte lesen

- Funktionsblock:



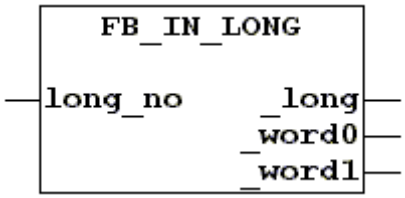
- Variablen:

byte_no	WORD; Wahl des Daten-Bytes: 0 … max
_byte	BYTE; Inhalt des gewählten Daten-Bytes
bit0…7	BOOL; Zustände der Bits 0…7 in *_byte*

FB_In_Long *FieldBus*

Daten-Long lesen

- Funktionsblock:



- Variablen:

long_no WORD;
Wahl des Daten-Longs: 0 … max

_long DINT;
Inhalt des gewählten Daten-Longs

word0 WORD;
Inhalt des niederwertigen Words (LSW) von *_long*

word1 WORD;
Inhalt des höherwertigen Words (MSW) von *_long*

FB_Out_Bit *FieldBus*

Datenbit schreiben

- Funktionsblock:

- Variablen:

byte_no WORD;
Wahl des Daten-Bytes: 0 … max

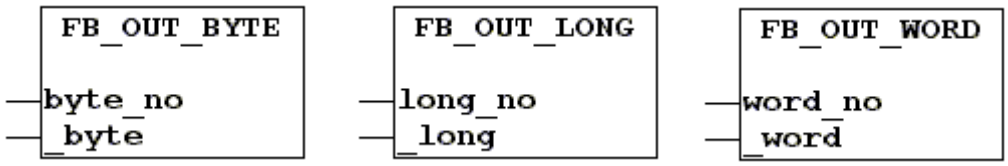
bit_no BYTE;
Bit-Position im gewählten Daten-Byte

bit BOOL;
Zustand des gewählten Datenbits

FB_Out_Byte / FB_Out_Long / FB_Out_Word *FieldBus*

Daten-Byte/Long/Word schreiben

- Funktionsblock:



- Variablen:

byte_no, *long_no*, *word_no* WORD;
Wahl des Daten-Bytes/Longs/Words: 0 … max

_byte BYTE;
Inhalt des gewählten Daten-Bytes

_long *DINT;*
Inhalt des gewählten Daten-Longs

_word *WORD;*
Inhalt des gewählten Daten-Words

4.8.1 Ethernet

Mit den hier aufgeführten FBs können einige Dateioperationen auf dem FTP-Server des Moduls ausgeführt werden. Temporäre Dateien können im RAM abgelegt werden (1 MByte, Verzeichnis "\RAM_DISC"). Daneben existiert ein 1,4 MByte großer Flash-Speicher, in dem beliebige Verzeichnisse erstellt (z. B. über Telnet oder einen FTP-Client wie den Internet Explorer) und Dateien dauerhaft abgelegt werden können.

Del_File *FieldBus (Ethernet)*

Datei im Modul löschen

- Funktionsblock:

- Variablen:

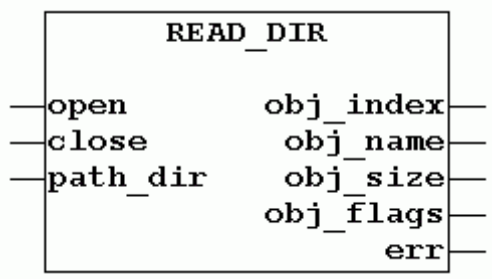
file_name STRING[255];
Pfad\Name der zu löschenden Datei

err BYTE;
Fehlernummer:
0 = kein Fehler
≠0 = Fehler

Read_Dir *FieldBus (Ethernet)*

Ein Verzeichnis im Modul auslesen

- Funktionsblock:



- Variablen:

open BOOL;
beim ersten Aufruf des FBs muss diese Variable auf TRUE gesetzt sein, um das Verzeichnis zu öffnen; bei weiteren Aufrufen sollte sie auf FALSE stehen

close BOOL;
beim vorzeitigen Beenden des Auslesevorgangs sollte diese Variable auf TRUE gesetzt sein, um das Verzeichnis wieder zu schließen; wenn kein weiterer Dateieintrag mehr vorhanden ist (*obj_index* = 0), erfolgt dies automatisch

path_dir STRING[255];
kompletter Verzeichnispfad (Pfad\Verzeichnisname)

obj_index WORD;
laufende Nummer des Dateieintrags im Verzeichnis; sie wird mit jedem weiteren FB-Aufruf inkrementiert, außer wenn kein weiterer Eintrag mehr vorhanden ist (*obj_index* = 0); auch ein "leeres" Verzeichnis enthält immer die beiden Systemeinträge "**.**" (*obj_index* = 1) und "**..**" (*obj_index* = 2)

obj_name STRING[255];
Datei- oder Unterverzeichnisname

obj_size *DWORD*;
Größe der Datei in Bytes

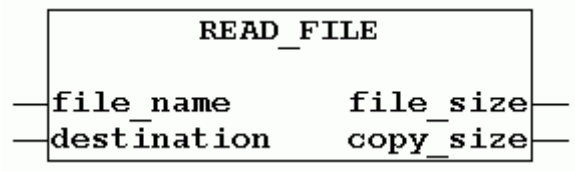
obj_flags BYTE;
Attribute der Datei:
Bit 0 = Verzeichnis
Bit 1 = Schreibschutz
Bit 2 = versteckt
Bit 3 = Systemdatei
Bit 4…7 = Reserve

err BYTE;
Fehlernummer:
0 = kein Fehler
1/2 = Verzeichnis konnte nicht geöffnet/geschlossen werden
3/4 = Verzeichnis bereits geöffnet/geschlossen

Read_File *FieldBus (Ethernet)*

Datei aus dem Modul in den reservierten RAM-Bereich der MotionPLC übertragen (siehe Abschnitt 4.11)

- Funktionsblock:



- Variablen:

file_name STRING[255];
Pfad\Name der zu lesenden Datei

destination WORD;
Zieladresse (Offset) im RAM der MotionPLC (0…FFFFh = 64 KByte)

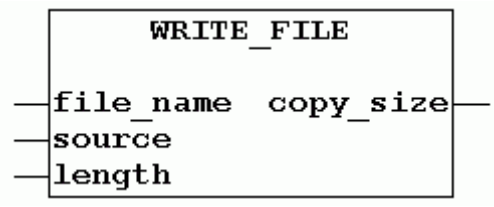
file_size *DWORD*;
Größe der Datei in Bytes

copy_size *DWORD*;
Anzahl der tatsächlich geschriebenen Bytes (zur Überprüfung einer Bereichsüberschreitung)

Write_File *FieldBus (Ethernet)*

Datei aus dem reservierten RAM-Bereich der MotionPLC (siehe Abschnitt 4.11) in das Modul schreiben, dynamisch in \RAM_DISC oder statisch in den Flash-Speicher

- Funktionsblock:



- Variablen:

file_name STRING[255];
Pfad\Name der zu erstellenden Datei im Modul

source WORD;
Quelladresse (Offset) im RAM der MotionPLC (0…FFFFh = 64 KByte)

length WORD;
Anzahl der zu schreibenden Bytes (Dateigröße)

copy_size DWORD;
Anzahl der übertragenen Bytes

4.9 Ein-/Ausgänge (Input / Output)

Mit den FBs dieser Kategorie können folgende Funktionen ausgeführt werden:

- Analogeingänge lesen (Ana_In)
- Analogausgänge lesen und schreiben (Rd/Wr_Ana_Out)
- Digitaleingänge lesen (Dig_In_Byte)
- Digitalausgänge lesen und schreiben (Rd_Dig_Out_Byte, Dig_Out_Bit)

! Die FBs greifen direkt auf die Hardware zu. Das bedeutet, dass Lese- und Schreibvorgänge schon beim Ausführen des FB erfolgen (mit einer maximalen Verzögerung von 300 µs) und nicht – wie sonst üblich bei der SPS-Programmierung – für alle Ein- und Ausgänge komplett am Anfang bzw. Ende des Programmzyklus. Wenn letzteres gewünscht wird, muss anstelle der FBs mit **%-Variablen** gearbeitet werden (siehe CoDeSys-Hilfe). Wenn von beiden Möglichkeiten in einem Programm Gebrauch gemacht werden soll, ist darauf zu achten, dass sie nicht gleichzeitig für einen Ausgang angewendet werden. Es könnten sich sonst unerwartete Schaltzustände ergeben (die von Betriebssystem, FB und %-Variable ausgegebenen Zustände für einen Ausgang werden ODER-verknüpft).

%-Variablen:

Die Anzahl der Variablen wurde werkseitig auf folgende Werte begrenzt und sollte nicht verändert werden:

Eingänge (%I…):	128
Ausgänge (%Q…):	128
Merker (%M…):	4096

Die Zuordnung zu den jeweiligen Ein- und Ausgängen wurde wie folgt vorgenommen:

▶ Digitaleingänge (Klemmleisten I1…4):

Klemme	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	

Klemme	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	

▶ Analogeingänge (Klemmleisten I5 und I6):

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

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

▶ Digital- und Analogausgänge (Klemmleisten Q1…3):

Klemme	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

Klemme	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

▶ Geber-Ein- und Ausgänge (Klemmleisten E1…3):

Klemme	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

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

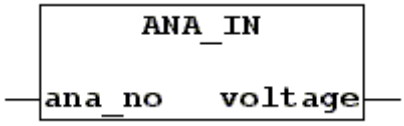
Diese Zuordnung kann auch unter CoDeSys im Register *Resourcen* unter *Steuerungskonfiguration* abgelesen werden. Falls nicht, muss die entsprechende Gerätedatei "angehängt" werden (über die rechte Maustaste innerhalb des Fensters).

Ana_In *Input / Output*

Liefert den Messwert an einem bestimmten Analogeingang (Wertebereiche siehe Anschlussbelegung in der Betriebsanleitung)

Siehe auch: Rd_Ana_Out, Wr_Ana_Out

- Funktionsblock:



- Variablen:

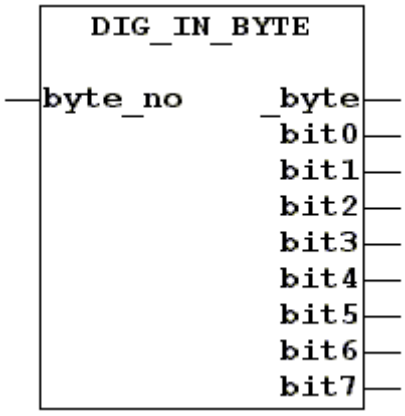
ana_no BYTE;
Nummer des Analogeingangs:
1…3 = Strom-/Spannungseingang 1…3 an Klemmleiste I6
4…7 = PT100-Eingang 4…7 an Klemmleiste I5

voltage INT;
digital gewandelter Messwert:
0…1023 für *ana_no* = 1…3
0…1564 für *ana_no* = 4…7

Dig_In_Byte *Input / Output*

Liefert die Zustände an bestimmten digitalen Eingangsklemmen

- Funktionsblock:



- Variablen:

byte_no BYTE;
Nummer des Digitaleingangs:
1…4 = Eingangsklemmleisten I1…4
10, 20, 30 = Geber-Eingangsklemmleisten E1, E2, E3

_byte BYTE;
Zustands-Byte der gewählten Eingangsklemmen

bit0…7 BOOL;
Schaltzustand eines Digitaleingangs; High = 1 (TRUE), Low = 0 (FALSE);
Zuordnung:

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

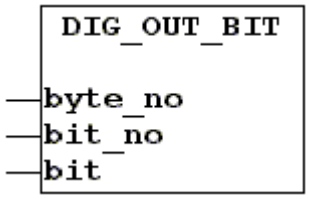
* bei I1 immer 0

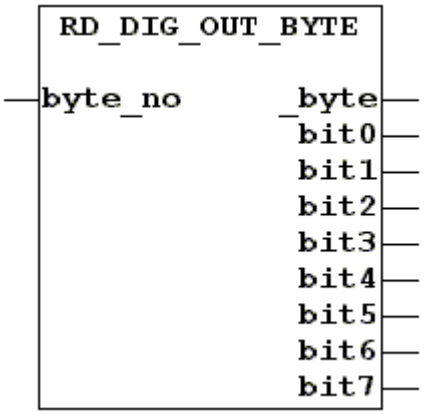
Dig_Out_Bit / Rd_Dig_Out_Byte *Input / Output*

Setzt den Zustand an einem bestimmten Digitalausgang / Liest den Zustand

Siehe auch: Dig_In_Byte

- Funktionsblock:





- Variablen:

byte_no BYTE;
Nummer des Digitalausgangs:
0 = Taktausgang (C, /C) an den Klemmleisten E1…3 (Voraussetzung: kein SSI-Geber aktiviert)
1…3 = Klemmleisten Q1…3

bit_no BYTE;
Position (Klemme) innerhalb der gewählen Klemmleiste;
Zuordnung:

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;
auszugebender Schaltzustand; High = 1 (TRUE), Low = 0 (FALSE)

_byte BYTE;
Zustands-Byte der gewählten Ausgangsklemmen

bit0…7 BOOL;
Schaltzustand eines Digitalausgangs; High = 1 (TRUE), Low = 0 (FALSE); Zuordnung wie bei *bit_no*

Rd_Ana_Out / Wr_Ana_Out *Input / Output*

Liest den Spannungswert an einem bestimmten Analogausgang / Schreibt einen Spannungswert (nur wenn der entsprechende Ausgang nicht von einer Analogachse genutzt wird)

Siehe auch: Ana_Out

- Funktionsblock:

- Variablen:

ana_no BYTE;
Nummer des Analogausgangs: 1…3 = Klemmleiste Q1…3 (Klemmen A+/-)

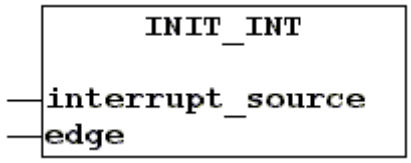
voltage INT;
Spannungswert in mV

4.10 Interrupt

Init_Int *Interrupt*

Macht einen bestimmten Eingang an Klemmleiste I1 oder Klemmleiste E1…3 interruptfähig. Bei Erkennung eines Interrupts werden die aktuellen Istpositionen von Master und Slave zwischengespeichert.

- Funktionsblock:



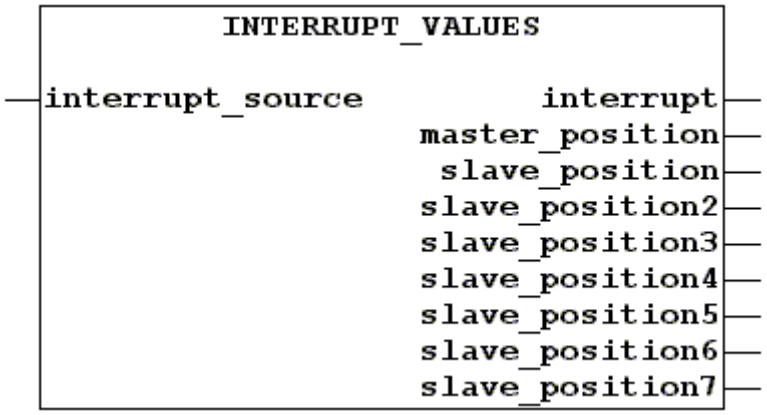
- Variablen:

interrupt_source	BYTE; Interrupteingang: 0…5 = Digitaleingang I1.0…5 6…8 = Geber-Nullsignaleingang E1.N…E3.N
edge	BYTE; Signalflanken-Auswertung: 0 = inaktiv 1 = positive Flanke 2 = negative Flanke 3 = beide Flanken

Interrupt_Values *Interrupt*

Liefert den Interruptstatus und die Istpositionen der Master- und Slave-Antriebe zum Zeitpunkt des Interruptereignisses

- Funktionsblock:



- Variablen:

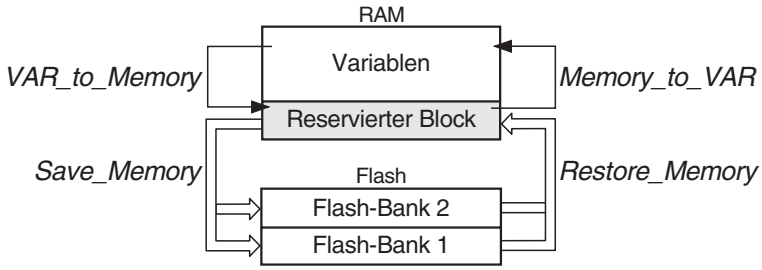
interrupt_source	BYTE; Interrupteingang: 0…5 = Digitaleingang I1.0…5 6…8 = Geber-Nullsignaleingang E1.N…E3.N
interrupt	BOOL; Interruptstatus; nach Aufruf dieses FB wird die Variable auf 0 gesetzt: 0 = kein Interrupt aufgetreten 1 = Interrupt ist aufgetreten
master_position	DINT; Istposition der Master-Achse
slave_position…7	DINT; Istposition der Slave-Achsen 1…7

4.11 Nichtflüchtiger Speicher (Non volatile memory)

Mit den hier zusammengefassten FBs können bestimmte Daten aus dem RAM stromausfallsicher in den Flash-Speicher gesichert und auch wieder zurückgeschrieben werden (siehe auch Kapitel 3).

Folgende Funktionen stehen zur Verfügung:

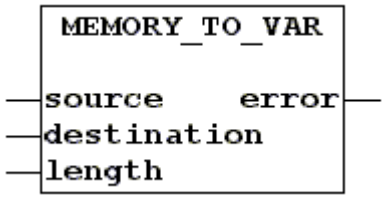
- Variablendaten aus dem normalen Arbeitsspeicher in einen speziellen 64K-RAM-Bereich kopieren und zurückladen (VAR_to_Memory, Memory_to_VAR)
- 64K-Datenblock des speziellen RAM-Bereichs in den Flash-Speicher schreiben und zurücklesen (Save_Memory, Restore_Memory)



Memory_to_VAR *Non volatile memory*

Überschreibt Variable(n) im Arbeitsspeicher mit Wert(en) aus dem reservierten RAM-Puffer, der üblicherweise zuvor mittels Restore_Memory (s. u.) mit den Daten aus einer Flash-Bank gefüllt wurde

◆ Funktionsblock:



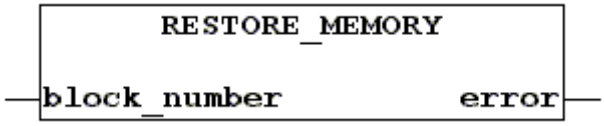
◆ Variablen:

source	WORD; Adresse (Offset) im reservierten 64K-RAM-Bereich, ab der sich die zu lesenden Daten befinden: 0…FFFFh
destination	POINTER TO BYTE; Startadresse der zu überschreibenden Variablen im Arbeitsspeicher
length	WORD; Anzahl der zu übertragenden Bytes
error	BOOL; Fehlerzustand: 0 = kein Fehler 1 = Speicherüberschreitung: *source* + *length* > FFFFh

Restore_Memory *Non volatile memory*

Überschreibt den im RAM reservierten 64K-Datenblock mit den Daten aus einer der beiden Flash-Bänke. Dieser FB wird üblicherweise vor Memory_to_-VAR (s. o.) aufgerufen.

- Funktionsblock:



- Variablen:

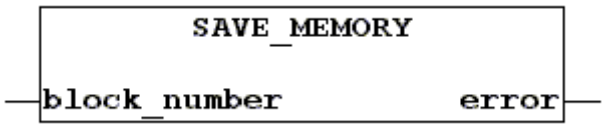
block_number BYTE;
1 = Flash-Bank 1
2 = Flash-Bank 2

error BYTE;
Fehlerzustand:
0 = kein Fehler
2 = unerlaubter Wert für *block_number*

Save_Memory *Non volatile memory*

Überschreibt eine der beiden Flash-Bänke mit den Daten aus dem reservierten 64K-Datenblock im RAM. Der Vorgang dauert einige Sekunden (während dieser Zeit ist das SPS-Programm unterbrochen). Dieser FB wird üblicherweise nach VAR_to_Memory (s. u.) aufgerufen.

- Funktionsblock:



- Variablen:

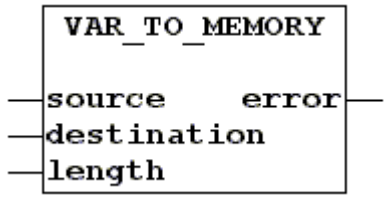
block_number BYTE;
1 = Flash-Bank 1
2 = Flash-Bank 2

error BYTE;
Fehlerzustand:
0 = kein Fehler
1 = Flash-Fehler (es erfolgte kein Schreibvorgang)
2 = unerlaubter Wert für *block_number*

VAR_to_Memory *Non volatile memory*

Kopiert Variable(n) aus dem Variablenbereich im Arbeitsspeicher in den reservierten RAM-Block (um von hier mittels Save_Memory in den Flash-Speicher gesichert zu werden).

- Funktionsblock:



- Variablen:

source	POINTER TO BYTE; Startadresse der zu kopierenden Variablen im Arbeitsspeicher
destination	WORD; Adresse (Offset) im reservierten 64K-RAM-Bereich, ab der die Variable gespeichert werden soll: 0…FFFFh
length	WORD; Anzahl der zu schreibenden Bytes: ≤ FFFFh - *destination*
error	BOOL; Fehlerzustand: 0 = kein Fehler 1 = Speicherüberschreitung: *destination* + *length* > FFFFh

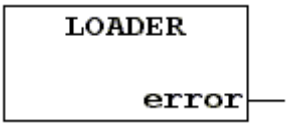
4.12 Parameter

Loader *Parameter*

Überprüft geänderte Parameter und aktiviert sie, falls kein Fehler vorliegt
Einige Parameter werden auch ohne Loader sofort aktiv, andere erfordern ein Aus- und Wiedereinschalten (siehe Parametereditor in der Betriebsanleitung).
Es wird auch ein CAN-Init ausgeführt (siehe FB in Abschnitt 4.4).

Dieser FB sollte nicht zyklisch in einem Programm aufgerufen werden, sondern nur gezielt, wenn ein oder mehrere Parameter geändert wurde(n). Andernfalls verlangsamt sich das Programm unnötig. Außerdem werden alle Parameter im RAM durch diejenigen aus dem Flash ersetzt, was ja nicht immer erwünscht ist und zu nicht so leicht erkennbaren Fehlern führen kann.

- Funktionsblock:



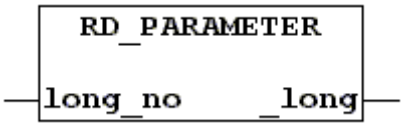
- Variablen:

error INT;
0 = Parameter wurden ohne Fehler übernommen, sonst Nummer des Parameters, der den Fehler verursacht hat

Rd_Parameter *Parameter*

Parameter auslesen

- Funktionsblock:



- Variablen:

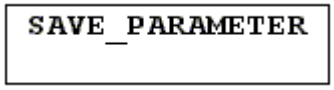
long_no WORD;
Nummer des Parameters, der gelesen werden soll: 000…999 (000…499 = Betriebssystem-Parameter – siehe Betriebsanleitung – und 500…999 = frei belegbar – z. B. von einem SPS-Programm)

_long DINT;
Inhalt des zugehörigen Speicherplatzes (im RAM)

Save_Parameter *Parameter*

Kopiert die Parameter und Kurvendaten aus dem RAM in den nicht-flüchtigen Flash-Speicher. Während dieser Übertragung wird ein gestartetes SPS-Programm angehalten.

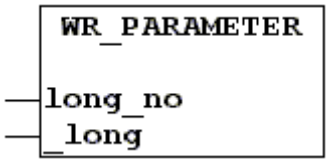
- Funktionsblock:



Wr_Parameter *Parameter*

Überträgt einen Parameter in das RAM der MotionPLC. Für die Nutzung des neuen Wertes bzw. der veränderten Eigenschaft des Parameters durch das Betriebssystem muss dann zuerst der FB Loader aufgerufen werden (nicht bei Parametern, die direkt aktiv werden; siehe Betriebsanleitung; hierzu gehören auch die Parameter für die Masterkorrektur: para[343].../[463].../[263]... ⇒ eine Änderung führt bei einer bereits gestarteten Kurve zu einem Einkuppelvorgang).

- Funktionsblock:



- Variablen:

long_no WORD;
Nummer des Parameters, der gelesen werden soll: 000…999 (000…499 = Betriebssystem-Parameter – siehe Betriebsanleitung – und 500…999 = frei belegbar – z. B. von einem SPS-Programm)

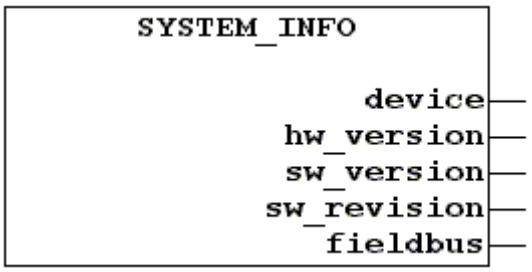
_long DINT;
Wert des Parameters

4.13 System

System_Info *System*

Auslesen von Systeminformationen

- Funktionsblock:



- Variablen:

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

hw_version BYTE;
Hardware-Versionsnummer (z. B. 3 bei HW3)

sw_version BYTE;
Software-Versionsnummer (z. B. 1 bei SW1.30)

sw_revision BYTE;
Software-Revisionsnummer (z. B. 30 bei SW1.30)

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

Time_On *System*

Einschaltdauer auslesen

- Funktionsblock:

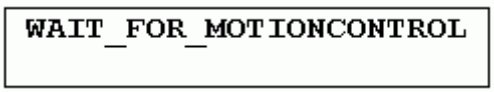
- Variablen:

_time TIME;
Zeit seit dem letzten Einschalten
Die Messzeit ist begrenzt auf 49 Tage 17 Stunden 2 Minuten 47 Sekunden 295 Millisekunden; nach dieser Zeit wird der Timer wieder auf Null gesetzt.

Wait_for_MotionControl *System*

Dieser FB unterbricht die Abarbeitung des SPS-Programms bis ein Achsregelzyklus erfolgt ist. So kann eine Parameteränderung mit direkt anschließender Statusabfrage erfolgen, um sich zu vergewissern, dass regelspezifische Parameter schon gewirkt haben. Dies ist allerdings ist nur sinnvoll, wenn mit einer einzigen Slave-Achse gearbeitet wird oder mit einer definierten Zykluszeit (para[211] $\neq$ 0).

- Funktionsblock:

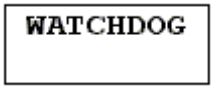


Watchdog *System*

Die Watchdog-Funktion des Betriebsystems verhindert, dass auftretende Fehler im SPS-Programm (Endlosschleifen etc.) die Regelung blockieren. Dazu bedient das Betriebssystem ca. alle 400 ms einen sogenannten Watchdog. Bleibt dieser Vorgang aus, weil das SPS-Programm den Prozessor für eine längere Zeit als ca. 400 ms beansprucht, so wird ein Reset ausgelöst. Dies führt beim Servoumrichter LD 2000 zum Fehler F20 (Anzeige).

Der FB ermöglicht auch einem SPS-Programm, den Watchdog zu bedienen. Dies sollte aber nur in Ausnahmefällen geschehen, wenn z. B. die (einmalige) Berechnung vieler Kurven in einer Interpolation zu einer längeren Zykluszeit als 400 ms führt. Der Aufruf des FB darf aber keinesfalls periodisch im Programmablauf erfolgen, da sonst der Sicherheitsaspekt untergraben würde.

- Funktionsblock:



Index

(*kursiv* dargestellte Begriffe bezeichnen Funktionsblöcke)