SYS TEC ELECTRONIC

SYS TEC spezifische Erweiterungen für OpenPCS / IEC 61131-3

User Manual

Version 4.0

Ausgabe Juli 2011

Dokument-Nr.: L-1054d_04

SYSTEC electronic GmbH   August-Bebel-Straße 29   D-07973 Greiz
Telefon: +49 (3661) 6279-0   Telefax: +49 (3661) 6279-99
Web: http://www.systec-electronic.com   Mail: info@systec-electronic.com

Status/Änderungen

Status: Freigegeben

Datum/Version	Abschnitt	Änderung	Bearbeiter
2004/11/17 1.0	alle	Erstellung	R. Sieber
2007/03/02 2.0	3	Abschnitt 3 neu eingefügt, dadurch alle nachfolgenden Abschnitte verschoben	R. Sieber
2007/07/18 3.0	1.4	Abschnitt 1.4 neu eingefügt	R. Sieber
2009/02/10 3.1	5.4	Beschreibung Rückgabewert korrigiert	R. Sieber
2011/07/15 4.0	2.2.6, 2.2.7, 3.12, 3.13, 4.5, 5.8, 5.9	Beschreibungen für FB/FUN mit Parameter vom Typ POINTER neu eingefügt	R. Sieber

Im Buch verwendete Bezeichnungen für Erzeugnisse, die zugleich ein eingetragenes Warenzeichen darstellen, wurden nicht besonders gekennzeichnet. Das Fehlen der © Markierung ist demzufolge nicht gleichbedeutend mit der Tatsache, dass die Bezeichnung als freier Warenname gilt. Ebenso wenig kann anhand der verwendeten Bezeichnung auf eventuell vorliegende Patente oder einen Gebrauchsmusterschutz geschlossen werden.

Die Informationen in diesem Handbuch wurden sorgfältig überprüft und können als zutreffend angenommen werden. Dennoch sei ausdrücklich darauf verwiesen, dass die Firma SYS TEC electronic GmbH weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgeschäden übernimmt, die auf den Gebrauch oder den Inhalt dieses Handbuches zurückzuführen sind. Die in diesem Handbuch enthaltenen Angaben können ohne vorherige Ankündigung geändert werden. Die Firma SYS TEC electronic GmbH geht damit keinerlei Verpflichtungen ein.

Ferner sei ausdrücklich darauf verwiesen, dass SYS TEC electronic GmbH weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgeschäden übernimmt, die auf falschen Gebrauch oder falschen Einsatz der Hard- bzw. Software zurückzuführen sind. Ebenso können ohne vorherige Ankündigung Layout oder Design der Hardware geändert werden. SYS TEC electronic GmbH geht damit keinerlei Verpflichtungen ein.

© Copyright 2011 SYS TEC electronic GmbH, D-07973 Greiz.
Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form ohne schriftliche Genehmigung der Firma SYS TEC electronic GmbH unter Einsatz entsprechender Systeme reproduziert, verarbeitet, vervielfältigt oder verbreitet werden.

Informieren Sie sich:

Kontakt	**Direkt**	**Ihr Lokaler Distributor**
Adresse:	SYS TEC electronic GmbH August-Bebel-Str. 29 D-07973 Greiz GERMANY	Sie finden eine Liste unserer Distributoren unter http://www.systec-electronic.com/distributors
Angebots-Hotline:	+49 (0) 36 61 / 62 79-0 info@systec-electronic.com	
Technische Hotline:	+49 (0) 36 61 / 62 79-0 support@systec-electronic.com	
Fax:	+49 (0) 36 61 / 6 79 99	
Webseite:	http://www.systec-electronic.com	

4. Auflage Juli 2011

Inhalt

1 Event-Tasks 7
1.1 Anwendung von Event-Tasks 7
1.2 Anlegen und Konfigurieren von Event-Tasks 7
1.3 Der Funktionsbaustein ETRC 8
1.4 Der Funktionsbaustein PTRC 11

2 Stringverarbeitung 13
2.1 Standard-Stringfunktionen nach IEC 61131-3 13
2.1.1 Die Funktion LEN 13
2.1.2 Die Funktion LEFT 13
2.1.3 Die Funktion RIGHT 14
2.1.4 Die Funktion MID 14
2.1.5 Die Funktion CONCAT 15
2.1.6 Die Funktion INSERT 15
2.1.7 Die Funktion DELETE 16
2.1.8 Die Funktion REPLACE 16
2.1.9 Die Funktion FIND 17
2.2 SYSTEC-spezifische Stringfunktionen und -funktionsbausteine 17
2.2.1 Der Funktionsbaustein GETSTRINFO 17
2.2.2 Die Funktion CHR 18
2.2.3 Die Funktion ASC 19
2.2.4 Die Funktion STR 19
2.2.5 Die Funktion VAL 20
2.2.6 Die Funktion BIN_TO_STR 21
2.2.7 Die Funktion STR_TO_BIN 23

3 Datenkommunikation über UDP 25
3.1 Anwendung der Datenkommunikation über UDP 25
3.2 Definitionen für UDP-Bausteine 25
3.3 Der Funktionsbaustein LAN_GET_HOST_CONFIG 26
3.4 Die Funktion LAN_ASCII_TO_INET 27
3.5 Die Funktion LAN_INET_TO_ASCII 28
3.6 Die Funktion LAN_GET_HOST_BY_NAME 28
3.7 Die Funktion LAN_GET_HOST_BY_ADDR 29
3.8 Der Funktionsbaustein LAN_UDP_CREATE_SOCKET 30
3.9 Der Funktionsbaustein LAN_UDP_CLOSE_SOCKET 31
3.10 Der Funktionsbaustein LAN_UDP_RECVFROM_STR 32
3.11 Der Funktionsbaustein LAN_UDP_SENDTO_STR 33
3.12 Der Funktionsbaustein LAN_UDP_RECVFROM_BIN 35
3.13 Der Funktionsbaustein LAN_UDP_SENDTO_BIN 36
3.14 Programmbeispiel zur Anwendung der UDP-Bausteine 37

4 Sicherung von Prozessdaten im nichtflüchtigen Speicher 41
4.1 Anwendung der nichtflüchtigen Speicherung von Prozessdaten 41
4.2 Der Funktionsbaustein NVDATA_BIT 41
4.3 Der Funktionsbaustein NVDATA_INT 44
4.4 Der Funktionsbaustein NVDATA_STR 47
4.5 Der Funktionsbaustein NVDATA_BIN 50

5 Zugriff auf Serielle Schnittstelle (SIO) 53
5.1 Verwendung der seriellen Schnittstelle 53
5.2 Der Funktionsbaustein SIO_INIT 53
5.3 Der Funktionsbaustein SIO_STATE 56
5.4 Der Funktionsbaustein SIO_READ_CHR 59
5.5 Der Funktionsbaustein SIO_WRITE_CHR 60
5.6 Der Funktionsbaustein SIO_READ_STR 63
5.7 Der Funktionsbaustein SIO_WRITE_STR 65

5.8 Der Funktionsbaustein SIO_READ_BIN 70
5.9 Der Funktionsbaustein SIO_WRITE_BIN 72

6 Zugriff auf Hardwarezähler 77
6.1 Verwendung von Hardwarezählern 77
6.2 Der Funktionsbaustein CNT_FUD 77

7 Zugriff auf Echtzeituhr (RTC) 82
7.1 Anwendung der Echtzeituhr (RTC) 82
7.2 Der Funktionsbaustein DT_CLOCK 82
7.3 Der Funktionsbaustein DT_ABS_TO_REL 85
7.4 Der Funktionsbaustein DT_REL_TO_ABS 86

8 Zugriff auf Impulsgenerator (PWM/PTO) 88
8.1 Anwendung des Impulsgenerators (PTO/PWM) 88
8.2 Der Funktionsbaustein PTO_PWM 89
8.3 Der Funktionsbaustein PTO_TAB 95

9 Verarbeitung von Prozessdaten 100
9.1 Anwendung des PID-Reglers 100
9.2 Der Funktionsbaustein PID1 102

10 Index 108

Tabellenverzeichnis

Tabelle 1: Event-Codes des Funktionsbausteines ETRC 9
Tabelle 2: Error-Codes des Funktionsbausteines ETRC 9
Tabelle 3: Start-Modes des Funktionsbausteines PTRC 11
Tabelle 4: Error-Codes des Funktionsbausteines PTRC 11
Tabelle 5: Formatspezifikationen für BIN_TO_STR 21
Tabelle 6: Formatspezifikationen für STR_TO_BIN 23
Tabelle 7: Error-Codes der Funktion STR_TO_BIN 24
Tabelle 8: Error-Codes der Funktionsbausteine LAN_Xxx 26
Tabelle 9: Aufruf-Modi für den Funktionsbaustein NVDATA_BIT 42
Tabelle 10: Error-Codes der Funktionsbausteine NVDATA_Xxx 42
Tabelle 11: Aufruf-Modi für den Funktionsbaustein NVDATA_INT 45
Tabelle 12: Aufruf-Modi für den Funktionsbaustein NVDATA_STR 48
Tabelle 13: Aufruf-Modi für den Funktionsbaustein NVDATA_BIN 51
Tabelle 14: Error-Codes des Funktionsbausteines SIO_INIT 54
Tabelle 15: Error-Codes des Funktionsbausteines SIO_STAT 57
Tabelle 16: Error-Codes des Funktionsbausteines SIO_READ_CHR 59
Tabelle 17: Error-Codes des Funktionsbausteines SIO_WRITE_CHR 61
Tabelle 18: Error-Codes des Funktionsbausteines SIO_READ_STR 64
Tabelle 19: Error-Codes des Funktionsbausteines SIO_WRITE_STR 66
Tabelle 20: Error-Codes des Funktionsbausteines SIO_READ_BIN 71
Tabelle 21: Error-Codes des Funktionsbausteines SIO_WRITE_STR 73
Tabelle 22: Error-Codes des Funktionsbausteines CNT_FUD 78
Tabelle 23: Error-Codes der Funktionsbausteine DT_Xxx 83
Tabelle 24: Error-Codes des Funktionsbausteines PTO_PWM 90
Tabelle 25: Error-Codes des Funktionsbausteines PTO_TAB 96
Tabelle 26: Error-Codes des Funktionsbausteines PID1 103

Bilderverzeichnis

Bild 1: Dialogfeld "Bearbeiten der Taskeigenschaften" 8
Bild 2: Signalverlauf der Ausgänge am Beispiel eines Vorwärtszählers 80
Bild 3: Zeitverhalten des Impulsgenerators im PTO-Modus 88
Bild 4: Zeitverhalten des Impulsgenerators im PWM-Modus 88
Bild 5: Zeitdiagramm zum Programmbeispiel "MotorCtl" 97
Bild 6: Prinizip eines Regelkreises 100
Bild 7: Aufbau der Regelstrecke zum Programmbeispiel "PidTest" 105
Bild 8: "PidTest" - Änderung Istwert beim Sprung der Führungsgröße (Sollwert) von 1V auf 6V 107

1 Event-Tasks

1.1 Anwendung von Event-Tasks

Als Event-Tasks werden SPS-Programme bezeichnet, die nur beim Auftreten eines bestimmten Ereignisses (auch "Interrupt" genannt) ausgeführt werden. Solche Ereignisse sind beispielsweise das Starten und Stoppen einer SPS sowie das Auftreten eines Laufzeitfehlers während der Programmausführung (Division durch Null oder Zugriff auf Elemente eines Datenfeldes außerhalb der definierten Feldgrenzen).

Die Aufgabe der Start-Task besteht in der einmaligen Konfiguration und Initialisierung von Steuerungs- oder Anlagenkomponenten. Dazu zählt beispielsweise die Parametrierung dezentraler Feldknoten zu Beginn der Programmausführung (z.B. Parametrierung von CANopen-Feldbusgeräten über entsprechende SDO-Zugriffe auf die Objektverzeichnisse der Geräte). Als Komplement hierzu ermöglicht die Stop-Task das definierte Abschalten der Feldknoten beim Beenden der SPS-Programmausführung. Mit Hilfe der Error-Task ist es möglich, im Fehlerfall sowohl die lokalen Ausgänge der SPS als auch die Ausgänge der Feldknoten in einen unkritischen Zustand zu setzen.

Start-Task: Die Abarbeitung der Start-Task erfolgt beim Zustandswechsel der SPS von Stop nach Run. Dies kann zum einen hardwareseitig durch Umschalten des RUN/STOP-Schalters in den Zustand RUN als auch softwareseitig durch Betätigung eines Start-Buttons in der OpenPCS-Programmierumgebung ausgelöst werden. Das eigentliche SPS-Hauptprogramm wird dann erst nach vollständiger Abarbeitung der Start-Task ausgeführt.

Stop-Task: Die Abarbeitung der Stop-Task erfolgt beim Zustandswechsel der SPS von Run nach Stop. Dies kann zum einen hardwareseitig durch Umschalten des RUN/STOP-Schalters in den Zustand STOP als auch softwareseitig durch Betätigung des Stop-Buttons in der OpenPCS-Programmierumgebung ausgelöst werden. Nach dem Beenden des eigentlichen SPS-Hauptprogrammes erfolgt die Abarbeitung der Stop-Task. Erst danach befindet sich die SPS im Zustand Stop.

Error-Task: Die Abarbeitung der Error-Task ist das Auftreten verschiedener Fehlerzustände gekoppelt (z.B. Division durch Null), die bei der Abarbeitung des SPS-Programmes auftreten können. Analog zur Stop-Task erfolgt auch nach dem Abbruch des eigentlichen SPS-Hauptprogrammes die Abarbeitung der Error-Task. Erst danach befindet sich die SPS im Zustand Stop.

1.2 Anlegen und Konfigurieren von Event-Tasks

Event-Tasks stellen aus Sicht von OpenPCS lediglich SPS-Programme mit spezifischen Eigenschaften dar. Das Anlegen einer Event-Task erfolgt daher analog zum Erstellen "normaler" Programme über den Menüpunkt "Datei ➲ Neu ➲ Programm". Beim Zuordnen der Task zur Ressource ist im Auswahlfeld "Tasktyp" der Eintrag "Interrupt" zu wählen (siehe Bild 1). Durch den im der Auswahlfeld "Interrupt" selektierten Name wird das zu verarbeitende Event definiert.

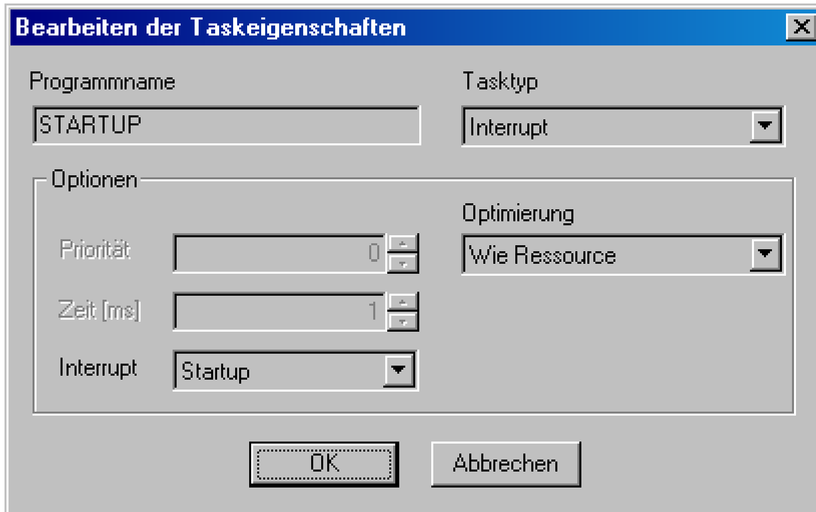


Bild 1: Dialogfeld "Bearbeiten der Taskeigenschaften"

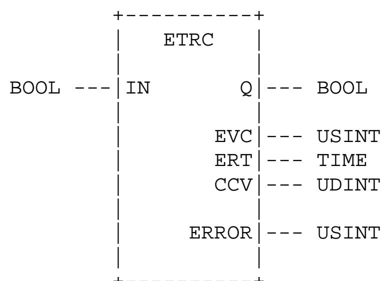
Standardmäßig wird eine Event-Task beim Eintreten des zugeordneten Ereignisses einmalig ausgeführt. Der im Abschnitt 1.3 beschriebene Funktionsbaustein *ETRC* ermöglicht eine Ausdehnung der Programmausführung auf mehrere aufeinander folgende Zyklen.

1.3 Der Funktionsbaustein ETRC

Standardmäßig wird eine Event-Task beim Eintreten des zugeordneten Ereignisses einmalig ausgeführt. Speziell beim Einsatz dezentraler Feldknoten kann es jedoch notwendig werden, die Ausführung einer Event-Task auf mehrere aufeinander folgende Zyklen auszudehnen. So erfordern beispielsweise die zur Parametrierung von CANopen-Feldbusgeräten notwendigen SDO-Zugriffe den kontinuierlichen Aufruf des SDO-Bausteins bis zum erfolgreichen Abschluss über mehrere SPS-Programmzyklen hinweg.

Mit Hilfe des Firmware-Funktionsbausteines *ETRC* (**E**vent **T**ask **R**un **C**ontrol) ist eine Event-Task in der Lage, ihre eigene Ausführung um jeweils einen weiteren Programmzyklus zu verlängern. Die an den Ausgängen des Bausteins verfügbaren Informationen über die bisherige Laufzeit und die Anzahl der bereits ausgeführten Zyklen können dabei als Abbruchkriterium verwendet werden, um im Fehlerfall nicht in einer Endlosschleife bei der Ausführung der Event-Task zu verharren.

Prototyp des Funktionsbausteines



Operandenbedeutung

IN: TRUE: Die Event-Task fordert die Ausführung für einen weiteren Zyklus an. FALSE: Die Event-Task beabsichtigt ihre Ausführung zu beenden oder erfragt nur aktuelle Statusinformationen, ohne gleichzeitig die Ausführung für einen weiteren Zyklus anzufordern.

Q: TRUE: Die Event-Task wird vom LZS noch für einen weiteren Zyklus abgearbeitet. FALSE: Die Ausführung der Event-Task wird nach dem aktuellen Zyklus beendet.

EVC: Der Event-Code beschreibt die systeminterne Ursache für den Aufruf der Event-Task, die Bedeutung der Event-Codes sind in Tabelle 1 definiert.

ERT: Die Elapsed Run Time gibt die Zeit an, die die Event-Task bereits ausgeführt wird.

CCV: Der Cycle Counter Value definiert die Anzahl der Zyklen, die die Event-Task bereits ausgeführt wird.

ERROR: Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 2 definiert.

Tabelle 1: Event-Codes des Funktionsbausteines ETRC

Event-Code	Event für den Aufruf der Task
0	aufgerufene Task ist nicht bekannt
1	Kaltstart der SPS ausgeführt
2	Warmstart der SPS ausgeführt
3	Heißstart der SPS ausgeführt
4	Start eines Einzelzyklus ausgeführt
5	SPS wurde mittels RUN/STOP Schalter in den Zustand STOP geschalten
6	SPS wurde softwareseitig in den Zustand STOP geschalten
7	SPS wechselt nach Abarbeitung eines Einzelzyklus in den Zustand STOP
8	allgemeiner Fehler bei SPS-Programmausführung (z.B. ungültiger Programmcode)
9	Division durch Null
10	Zugriff auf einen ungültigen Index eines Datenfeldes (ARRAY)
11	Fehler während der Abarbeitung eines Funktionsbausteines

Tabelle 2: Error-Codes des Funktionsbausteines ETRC

Error-Code	Bedeutung
0	der Funktionsbaustein wurde erfolgreich ausgeführt
1	der Funktionsbaustein wurde von einer Task aufgerufen, die selbst keine gültige Event-Task ist, die Ausführung des Funktionsbausteines ist in diesem Fall nicht möglich

Beschreibung

Standardmäßig wird eine Event-Tasks nur für jeweils einen einzigen Zyklus aufgerufen wird. Benötigt die Event-Task noch weitere Zyklen für ihre Ausführung, muss sie das mit Hilfe des Funktionsbausteines *ETRC* anmelden. Gleichzeitig liefert der Funktionsbaustein *ETRC* am Ausgang *EVC* den Grund für den Aufruf der Event-Task (siehe Tabelle 1). Weiterhin gibt der Ausgang *ERT* (Elapsed Run Time) die bereits verstrichene Ausführungszeit der Event-Task in Millisekunden an und der Ausgang *CCV* (Cycle Counter Value) die Anzahl der Zyklen, in denen die Event-Task bereits ausgeführt wird. Mit Hilfe der beiden Informationen *ERT* und *CCV* kann die Event-Task selbst entscheiden, ob sie noch einen weiteren Zyklus ausgeführt werden möchte oder nicht. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 2 beschrieben.

Mit Hilfe des im Abschnitt 1.4 beschriebenen Funktionsbausteines *PTRC* (**P**rogram **T**ask **R**un **C**ontrol) hat die SPS die Möglichkeit, nach einem Laufzeitfehler (z.B. Division durch Null) die reguläre Ausführung des SPS-Programmes erneut zu starten.

Das nachfolgende Programmbeispiel zeigt eine einfache Start-Task, die für insgesamt 4 Zyklen abgearbeitet wird. Dazu fordert die Task 3 mal die Ausführung für einen weiteren Zyklus durch den Aufruf des Funktionsbausteines *ETRC* an.

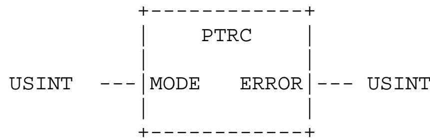
Programmbeispiel

```
PROGRAM Startup

VAR
    Out8_15 AT %QB1.0 : BYTE;
    RunState : BOOL;
    EventCode : USINT;
    RunTime : TIME;
    CycleCounter : UDINT;
    Error : USINT;

        FB_ETRC : ETRC;
END_VAR


(* get the current state only, but don't request execution time for *)
(* the next cycle yet *)
CAL     FB_ETRC (
        IN := FALSE
        |
        RunState := Q,
        EventCode := EVC,
        RunTime := ERT,
        CycleCounter := CCV,
        Error := ERROR)

LD      CycleCounter
UDINT_TO_BYTE
ST      Out8_15

(* for 1.-3. cycle request execution time for the next cycle *)
LD      CycleCounter
LE      3
CALC    FB_ETRC (IN := TRUE)

RET

END_PROGRAM
```

1.4 Der Funktionsbaustein PTRC

Mit Hilfe des Firmware-Funktionsbausteines *PTRC* (**P**rogram **T**ask **R**un **C**ontrol) ist eine SPS in der Lage, ihre eigene Programmausführung zu beenden oder nach einem Fehler wieder neu zu starten.

Prototyp des Funktionsbausteines



Operandenbedeutung

MODE: Auszuführendes Start- bzw. Stopp-Kommando gemäß Tabelle 3

ERROR: Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 4 definiert.

Tabelle 3: Start-Modes des Funktionsbausteines PTRC

Start-Mode	Bedeutung
0	Stoppen der SPS, Programmausführung beenden
1	Wiederanlauf der SPS mit Kaltstart ausführen
2	Wiederanlauf der SPS mit Warmstart ausführen
3	Wiederanlauf der SPS mit Heißstart ausführen

Tabelle 4: Error-Codes des Funktionsbausteines PTRC

Error-Code	Bedeutung
0	der Funktionsbaustein wurde erfolgreich ausgeführt
4	Ungültiger Modus (*MODE*) beim Aufruf des Funktionsbausteines

Beschreibung

Mit Hilfe des Funktionsbausteines ist eine SPS in der Lage, ihre eigene Programmausführung zu beenden oder nach einem Fehler wieder neu zu starten. Der Aufruf des Bausteines zum Restart der SPS erfolgt typischerweise aus einer Error-Task (siehe Abschnitt 1.1). Dies ermöglicht einen autarken Wiederanlauf der SPS nach einem Laufzeitfehler (z.B. Division durch Null) und ist somit Voraussetzung für einen kontinuierlichen, bedienerlosen Betrieb der Steuerung. Die als Parameter für den Eingang *MODE* unterstützen Modi sind in Tabelle 3 aufgeführt. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 4 beschrieben.

Das nachfolgende Programmbeispiel zeigt die Anwendung des Funktionsbausteines *PTRC* innerhalb einer anwenderspezifischen Error-Task (hier als Programm "Resume", das wie in Abschnitt 1.2 als Error-Task konfiguriert wurde).

Programmbeispiel

```
PROGRAM Resume

VAR CONSTANT
    PTRC_MODE_STOP       : USINT    := 0;
    PTRC_MODE_COLDSTART  : USINT    := 1;
    PTRC_MODE_WARMSTART  : USINT    := 2;
    PTRC_MODE_HOTSTART   : USINT    := 3;
END_VAR

VAR
    FB_PTRC  : PTRC;
    usiError : USINT;
END_VAR

FB_PTRC (MODE := PTRC_MODE_COLDSTART | usiError := ERROR);

RETURN;

END_PROGRAM
```

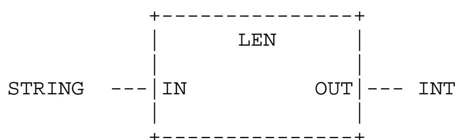
2 Stringverarbeitung

2.1 Standard-Stringfunktionen nach IEC 61131-3

Die im Folgenden aufgeführten Stringfunktionen sind Standardfunktionen gemäß IEC 61131-3 und ausführlich in der OpenPCS-Onlinehilfe beschrieben. Die Auflistung an dieser Stelle soll einen Gesamtüberblick über alle auf SYSTEC-Steuerungen verfügbaren Stringfunktionen vermitteln.

2.1.1 Die Funktion LEN

Die Funktion *LEN* ermittelt die Länge einer Zeichenfolge.



Beschreibung

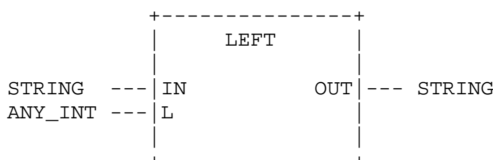
Diese Funktion ermittelt die Länge der Zeichenfolge IN.

Beispiel

```
A := LEN('ABCDEF');                    (* Result: A := 6 *)
```

2.1.2 Die Funktion LEFT

Die Funktion *LEFT* ermittelt den linken Teil einer Zeichenfolge.



Beschreibung

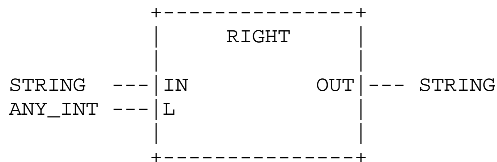
Diese Funktion ermittelt den linken Teil mit der Länge L von der Zeichenfolge IN.

Programmbeispiel

```
A := LEFT(IN:='ABCDEF', L:=3);         (* Result: A := 'ABC' *)
```

2.1.3 Die Funktion RIGHT

Die Funktion *RIGHT* ermittelt den rechten Teil einer Zeichenfolge.



<u>Beschreibung</u>

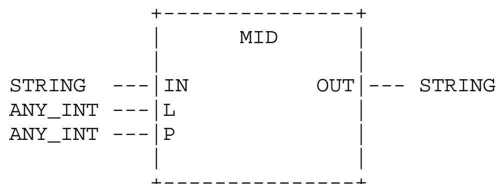
Diese Funktion ermittelt den rechten Teil mit der Länge L von der Zeichenfolge IN.

<u>Programmbeispiel</u>

```
A := RIGHT(IN:='ABCDEF', L:=3);              (* Result: A := 'DEF' *)
```

2.1.4 Die Funktion MID

Die Funktion *MID* ermittelt den mittleren Teil einer Zeichenfolge.



<u>Beschreibung</u>

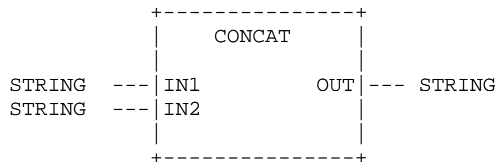
Diese Funktion ermittelt den mittleren Teil mit der Länge L von der Zeichenfolge IN, beginnend ab Position P.

<u>Programmbeispiel</u>

```
A := MID(IN:='ABCDEF', L:=3, P:=2);          (* Result: A := 'BCD' *)
```

2.1.5 Die Funktion CONCAT

Die Funktion *CONCAT* dient zum Zusammensetzen von Zeichenfolgen.



<u>Beschreibung</u>

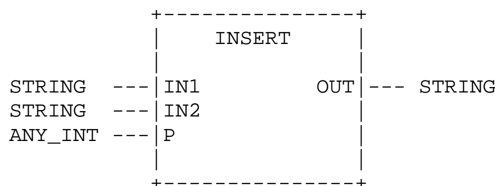
Diese Funktion ermittelt die aus den beiden Zeichenfolgen IN1 und IN2 zusammengesetzte Gesamtzeichenfolge.

<u>Programmbeispiel</u>

```
A := CONCAT(IN1:='ABC', IN2:='xyz');        (* Result: A := 'ABCxyz' *)
```

2.1.6 Die Funktion INSERT

Die Funktion *INSERT* fügt eine Zeichenfolge in eine andere Zeichenfolge ein.



<u>Beschreibung</u>

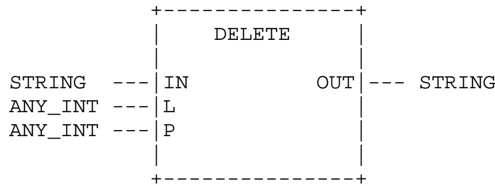
Diese Funktion fügt die Zeichenfolge IN2 nach der Position P in die Zeichenfolge IN1 ein.

<u>Programmbeispiel</u>

```
A := INSERT(IN1:='ABC', IN2:='xyz', P:=2);  (* Result: A := 'ABxyzC' *)
```

2.1.7 Die Funktion DELETE

Die Funktion *DELETE* dient zum Löschen von Zeichen aus einer Zeichenfolge.



Beschreibung

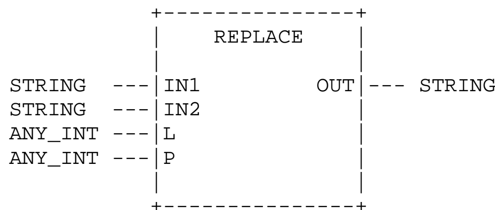
Diese Funktion löscht L Zeichen aus der Zeichenfolge IN, beginnend ab der Position P.

Programmbeispiel

```
A := DELETE(IN:='ABCDEF', L:=3, P:=2);      (* Result: A := 'AEF' *)
```

2.1.8 Die Funktion REPLACE

Die Funktion *REPLACE* dient zum Ersetzen von Teilen einer Zeichenfolge.



Beschreibung

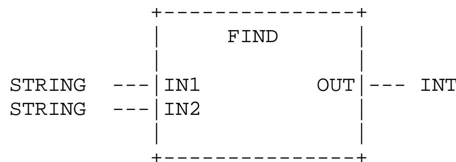
Diese Funktion ersetzt L Zeichen der Zeichenfolge IN1 durch die Zeichenfolge IN2, beginnend ab der Position P.

Programmbeispiel

```
A := REPLACE(IN1:='ABCDEF', IN2:='z',       (* Result: A := 'AzEF' *)
             L:=3, P:=2);
```

2.1.9 Die Funktion FIND

Die Funktion *FIND* dient zum Suchen einer Zeichenfolge.



Beschreibung

Diese Funktion ermittelt die Position des Anfangs für das erste Auftreten der Zeichenfolge IN2 in der Zeichenfolge IN1. Ist die Zeichenfolge IN2 nicht in der Zeichenfolge IN1 enthalten, liefert die Funktion den Wert 0.

Programmbeispiel

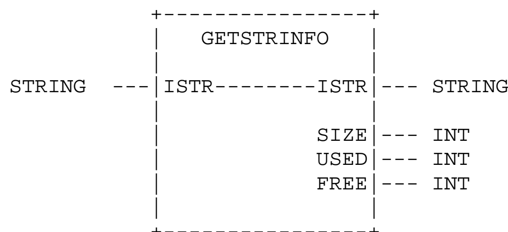
```
A := FIND(IN1:='ABCBCF', IN2:='BC');        (* Result: A := 2 *)
```

2.2 SYSTEC-spezifische Stringfunktionen und -funktionsbausteine

2.2.1 Der Funktionsbaustein GETSTRINFO

Der Funktionsbaustein *GETSTRINFO* dient zum Abfragen spezifischer Stringinformationen.

Prototyp des Funktionsbausteines



Operandenbedeutung

ISTR	String, dessen Eigenschaften zu ermitteln sind
SIZE	maximale Stringlänge (interne Größe des zur Verfügung stehenden Puffers für diese String-Variable)
USED	belegte Stringlänge (gleichbedeutend mit IEC 61131-3 Standardfunktion *LEN*, siehe Abschnitt 2.1.1)
FREE	noch verfügbare/ungenutzte Stringlänge (gleichbedeutend mit *SIZE - USED*)

Beschreibung

Dieser Funktionsbaustein ermittelt zu einem gegebenen String die Größe des intern zur Verfügung stehenden Puffers (maximale Stringlänge) und die bereits belegte sowie die noch zur Verfügung

stehende Stringlänge. Von Bedeutung ist dieser Baustein insbesondere in Verbindung mit anderen Funktionsbausteinen, die zum Auslesen oder Empfangen von Zeichenketten dienen, wie beispielsweise *NVDATA_STR* (siehe Abschnitt 4.4), *SIO_READ_STR* (siehe Abschnitt 5.6) oder *CAN_SDO_READ_STR*.

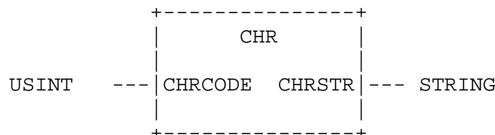
Programmbeispiel

```
VAR
    strText  : STRING(16) := 'ABCDEFGHIJ';
    iStrSize : INT;
    iStrUsed : INT;
    iStrFree : INT;
    FB_GetStrInfo : GETSTRINFO;
END_VAR

CAL     FB_GetStrInfo (
        ISTR := strText
        |
        iStrSize := SIZE,            (* iStrSize := 16 *)
        iStrUsed := USED,            (* iStrUsed := 10 *)
        iStrFree := FREE)            (* iStrFree := 6  *)

...

RET
```

2.2.2 Die Funktion CHR

Die Funktion *CHR* wandelt einen numerischen Zeichencode in das entsprechende ASCII-Zeichen.

Prototyp der Funktion



Operandenbedeutung

CHRCODE	numerischer Zeichencode, der in ein ASCII-Zeichen zu wandeln ist
CHRSTR	String mit ASCII-Zeichen entsprechend dem numerischen Zeichencode

Beschreibung

Diese Funktion liefert am Ausgang *CHRSTR* einen String zurück, dessen einziges Zeichen dem am Eingang *CHRCODE* übergebenen Zeichencode entspricht.

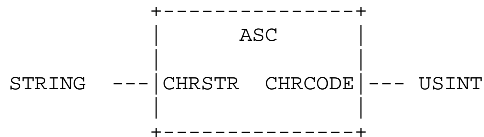
Programmbeispiel

```
CHR(16#41)      (* Result: 'A' *)
CHR(97)         (* Result: 'a' *)
CHR(60)         (* Result: '<' *)
CHR(36)         (* Result: '$' *)
```

2.2.3 Die Funktion ASC

Die Funktion *ASC* wandelt ein ASCII-Zeichen in den entsprechenden numerischen Zeichencode.

Prototyp der Funktion



Operandenbedeutung

CHRSTR String, von dessen erstem Zeichen der numerische Zeichencode zu ermitteln ist

CHRCODE numerischer Zeichencode des ersten im String enthaltenen ASCII-Zeichens

Beschreibung

Diese Funktion liefert am Ausgang *CHRCODE* den numerischen Zeichencode vom ersten Zeichen des am Eingang *CHRSTR* übergebenen Strings zurück.

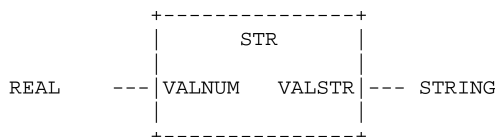
Programmbeispiel

```
ASC('A')        (* Result: 65 / 16#41 *)
ASC('a')        (* Result: 97 / 16#61 *)
ASC('ABC')      (* Result: 65 / 16#41 *)
ASC(' 123')     (* Result: 32 / 16#20 *)
```

2.2.4 Die Funktion STR

Die Funktion *STR* wandelt einen REAL-Wert in einen entsprechenden String.

Prototyp der Funktion



Operandenbedeutung

VALNUM numerischer REAL-Wert, der in einen String zu wandeln ist

VALSTR String mit Zeichenfolge entsprechend dem numerischen REAL-Wert

Beschreibung

Diese Funktion wandelt den am Eingang *VALNUM* übergebenen numerischen REAL-Wert in einen entsprechenden String und gibt diesem am Ausgang *VALSTR* zurück. Beim der Umwandlung wird immer ein führendes Leerzeichen für das Vorzeichen reserviert. Es werden keine nachfolgenden Nullen ausgegeben, bei ganzen Zahlen entfällt auch der Dezimalpunkt.

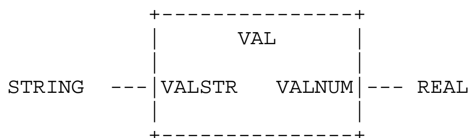
Programmbeispiel

```
STR(123)        (* Result: ' 123'      *)
STR(123.45)     (* Result: ' 123.45'   *)
STR(-123.45)    (* Result: '-123.45'   *)
STR(98.7654)    (* Result: ' 98.7654' *)
```

2.2.5 Die Funktion VAL

Die Funktion *VAL* wandelt einen String in einen entsprechenden REAL-Wert.

Prototyp der Funktion



Operandenbedeutung

VALSTR String, dessen Zeichenfolge in einen numerischen REAL-Wert zu wandeln ist

VALNUM numerischer REAL-Wert entsprechend der übergebenen Zeichenfolge

Beschreibung

Diese Funktion wandelt den am Eingang *VALSTR* übergebenen String in einen entsprechenden numerischen REAL-Wert und liefert diesem am Ausgang *VALNUM* zurück.

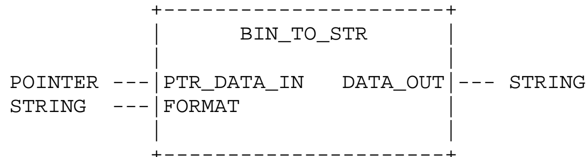
Programmbeispiel

```
VAL('123')      (* Result: 123     *)
VAL('123.45')   (* Result: 123.45  *)
VAL('-123.45')  (* Result: -123.45 *)
VAL('98.7654')  (* Result: 98.7654 *)
```

2.2.6 Die Funktion BIN_TO_STR

Die Funktion *BIN_TO_STR* wandelt einen numerischen Wert in einen entsprechenden String.

Prototyp der Funktion



Operandenbedeutung

PTR_DATA_IN Adresse eines Objektes, dessen Wert in einen String zu wandeln ist

FORMAT String mit Spezifikation des Ausgabeformates, siehe Tabelle 5

DATA_OUT formatierte Zeichenfolge entsprechend dem numerischen Eingabewert

Beschreibung

Diese Funktion wandelt ein über *PTR_DATA_IN* adressiertes numerisches Objekt unter Berücksichtigung der angegebenen Formatspezifikation in einen entsprechenden String. Die am Eingang *FORMAT* übergebene Formatspezifikation legt das Ausgabeformat des als *DATA_OUT* zurück gelieferten Strings fest, dessen Zeichenfolge dem numerischen Eingabewert entspricht. Tabelle 5 beschreibt die möglichen Formatspezifikationen.

Tabelle 5: Formatspezifikationen für BIN_TO_STR

Objekt-Typ	Formatspezifikation	Beschreibung
BOOL	'd'	Ausgabe in numerischerer Schreibweise { 0 | 1 }
	'b'	Ausgabe als Literal in Kleinbuchstaben { true | false }
	'B'	Ausgabe als Literal in Großbuchstaben { TRUE | FALSE }
BYTE, USINT, SINT WORD, UINT, INT DWORD, UDINT, DINT	'd'	Ausgabe in dezimaler Schreibweise, optional Festlegung minimal auszugebender Stellen möglich (siehe Text unten)
	'x'	Ausgabe in hexadezimaler Schreibweise mit Kleinbuchstaben, optional Festlegung minimal auszugebender Stellen möglich (siehe Text unten)
	'X'	Ausgabe in hexadezimaler Schreibweise mit Großbuchstaben, optional Festlegung minimal auszugebender Stellen möglich (siehe Text unten)
REAL	'd' oder 'f'	Ausgabe in dezimaler Schreibweise mit Nachkommastellen, optional Festlegung der minimal auszugebender Gesamtstellen sowie Nachkommastellen möglich (siehe Text unten)

Für numerische Ganzzahltypen können die Formatspezifikationen 'd', 'x' und 'X' optional um die Festlegung der minimal auszugebenden Stellenanzahl erweitert werden. Diese Mindeststellenanzahl ist an die Formatspezifikation anzuhängen (z.B. 'd4'). Eine der Mindeststellenanzahl vorangestellte '0' bewirkt, dass der Ausgabestring ggf. linksbündig mit '0'-Zeichen aufgefüllt wird, um die geforderte

Ausgabebreite zu erreichen (z.B. 'd04'). Andernfalls wird der Ausgabestring ggf. linksbündig mit Leerzeichen aufgefüllt.

Für Objekte vom Typ REAL können die Formatspezifikationen 'd', und 'f' ebenfalls optional um die Festlegung der minimal auszugebenden Stellenanzahl erweitert werden. Hierbei wird zwischen Gesamtstellenanzahl und Nachkommastellen unterschieden. Nachkommastellen sind in der Form '.y' anzugeben (z.B. 'f.4' für 4 Nachkommastellen). Optional kann in der Form 'x.y' vor dem Punkt die Mindeststellenanzahl für den gesamten Ausgabestring festgelegt werden, wobei hier der Dezimalpunkt selbst ebenfalls mitgezählt wird. So bewirkt beispielsweise die Formatspezifikation 'f9.4' die Ausgabe eines Strings mit insgesamt 9 Zeichen, von den ein Zeichen der Dezimalpunkt selbst ist, gefolgt von 4 Nachkommastellen, so dass sich damit 4 Vorkommastellen ergeben (9 total – 1 Dezimalpunkt – 4 Nachkommastellen = 4 Vorkommastellen). Eine der Gesamtmindeststellenanzahl ('x' im Format 'x.y') vorangestellte '0' bewirkt, dass der Ausgabestring ggf. linksbündig mit '0'-Zeichen aufgefüllt wird, um die geforderte Ausgabebreite zu erreichen (z.B. 'f09.4'). Andernfalls wird der Ausgabestring ggf. linksbündig mit Leerzeichen aufgefüllt.

Ist eine erfolgreiche Wandlung aufgrund ungültiger Parameter nicht möglich, enthält der Ausgabestring *DATA_OUT* eine entsprechende Fehlermeldung im Klartext (z.B. *'ERROR: data type not supported'* oder *'ERROR: format type not supported'*).

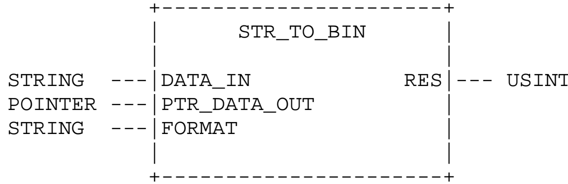
Programmbeispiel

```
VAR

    xBoolVar    : BOOL;
    iIntVar     : INT;
    rRealVar    : REAL;

    pVar        : POINTER;
    strResult   : STRING;

END_VAR

    xBoolVar := TRUE;
    pVar := &xBoolVar;
    strResult := BIN_TO_STR (pVar, 'd');        (* strResult: '1'          *)
    strResult := BIN_TO_STR (pVar, 'b');        (* strResult: 'true'       *)

    iIntVar := 123;
    pVar := &iIntVar;
    strResult := BIN_TO_STR (pVar, 'd');        (* strResult: '123'        *)
    strResult := BIN_TO_STR (pVar, 'x4');       (* strResult: '  7b'       *)
    strResult := BIN_TO_STR (pVar, 'X04');      (* strResult: '007B'       *)

    rRealVar := 123.456;
    pVar := &rRealVar;
    strResult := BIN_TO_STR (pVar, 'f.4');      (* strResult: '123.4560'  *)
    strResult := BIN_TO_STR (pVar, 'f09.4');    (* strResult: '0123.4560' *)
```

2.2.7 Die Funktion STR_TO_BIN

Die Funktion *STR_TO_BIN* wandelt einen String in einen entsprechenden numerischen Wert.

Prototyp der Funktion



Operandenbedeutung

DATA_IN String, dessen Zeichenfolge in einen numerischen Wert zu wandeln ist

FORMAT String mit Spezifikation des Eingabeformates, siehe Tabelle 6

PTR_DATA_OUT Adresse eines Objektes, in dem der gewandelte, numerische Wert abzulegen ist

RES Informationen zum Ausführungsergebnis der Wandlung, mögliche Fehlercodes sind in Tabelle 7 definiert

Beschreibung

Diese Funktion wandelt einen am Eingang *DATA_IN* übergebenen String in einen entsprechenden numerischen Wert um und legt diesen in dem über *PTR_DATA_OUT* adressierten Objekt ab. Die am Eingang *FORMAT* übergebene Formatspezifikation beschreibt das Eingabeformat des Strings, der in einen numerischen Wert zu wandeln ist. Tabelle 6 beschreibt die möglichen Formatspezifikationen.

Tabelle 6: Formatspezifikationen für STR_TO_BIN

Objekt-Typ	Formatspezifikation	Beschreibung
BOOL	'd'	Eingabestring in numerischerer Schreibweise { 0 | 1 }
	'b' oder 'B'	Eingabestring als Literal, beliebige Groß- oder Kleinbuchstaben { true | false | TRUE | FALSE }
BYTE, USINT, SINT WORD, UINT, INT DWORD, UDINT, DINT	'd'	Eingabestring in dezimaler Schreibweise
	'x' oder 'X'	Eingabestring in hexadezimaler Schreibweise, beliebige Groß- oder Kleinbuchstaben
REAL	'd' oder 'f'	Eingabestring in dezimaler Schreibweise mit Nachkommastellen

Tabelle 7: Error-Codes der Funktion STR_TO_BIN

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Pointer verweist auf ein Objekt eines nicht unterstützten Datentyps
2	Ungültige Formatspezifikation (*FORMAT*) beim Aufruf des Funktionsbausteines
4	Ungültiger Eingabestring (*DATA_IN*) beim Aufruf des Funktionsbausteines

Programmbeispiel

```
VAR

    xBoolVar    : BOOL;
    iIntVar     : INT;
    rRealVar    : REAL;

    pVar        : POINTER;
    usiRes      : USINT;

END_VAR

    pVar := &xBoolVar;
    usiRes := STR_TO_BIN ('1', pVar, 'd');
    usiRes := STR_TO_BIN ('true', pVar, 'b');

    pVar := &iIntVar;
    usiRes := STR_TO_BIN ('-123', pVar, 'd');
    usiRes := STR_TO_BIN ('ABCD', pVar, 'x');

    pVar := &rRealVar;
    usiRes := STR_TO_BIN ('123.456', pVar, 'd');
```

3 Datenkommunikation über UDP

3.1 Anwendung der Datenkommunikation über UDP

UDP (**U**ser **D**atagram **P**rotocol) ist ein minimales, verbindungsloses und paketorientiertes Netzprotokoll, das zur Transportschicht der Internetprotokollfamilie gehört. Die meisten im industriellen Umfeld eingesetzten Geräte mit Ethernet-Schnittstelle unterstützen UDP. Daher bietet sich dieses Protokoll für den Ethernet-basierten Datenaustausch zwischen der SPS und Geräten wie beispielsweise Terminals (HMI) oder Leitrechnern an.

Das Senden und Empfangen von UDP-Paketen erfolgt über Sockets. Für das Anlegen eines lokalen Sockets ist der Funktionsbaustein *LAN_UDP_CREATE_SOCKET* zuständig. Der Funktionsbaustein *LAN_UDP_SENDTO_STR* ermöglicht das Versenden von Datenpakten, äquivalent dazu dient der Baustein *LAN_UDP_RECVFROM_STR* zum Empfang von Daten. Ein nicht mehr benötigtes Socket kann mit Hilfe des Funktionsbausteines *LAN_UDP_CLOSE_SOCKET* wieder frei gegeben werden. Das Beenden des SPS-Programms führt intern automatisch zum Schließen aller belegten Sockets.

3.2 Definitionen für UDP-Bausteine

Für die Verwendung durch UDP-Bausteine sind folgende Datentypen global in OpenPCS definiert:

```
TYPE
    INETV4 : UDINT;
END_TYPE

TYPE
    SOCKID : UINT;
END_TYPE
```

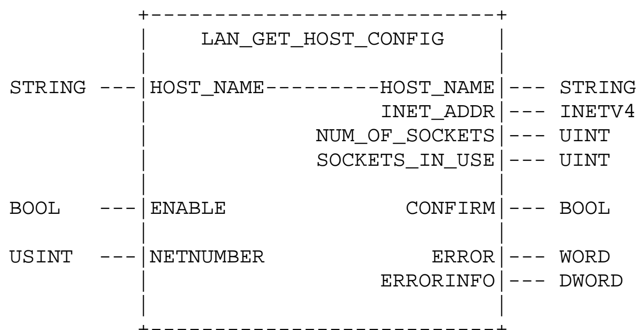
Tabelle 8: Error-Codes der Funktionsbausteine LAN_Xxx

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Die angegebene Netzwerknummer (*NETNUMBER*) wird nicht unterstützt
2	Beim Aufruf des Bausteins wurde ein ungültiger Parameter angegeben
3	Fehler bei der Initialisierung der UDP-Schicht auf der SPS
4	Beim Anlegen, Senden oder Empfangen eines Sockets meldet die UDP-Schicht auf der SPS einen Fehler
5	Es ist kein freier Socket mehr verfügbar
6	Die angegebene Socket-ID ist ungültig
7	Der Socket mit der angegebenen Socket-ID ist nicht in Benutzung
8	Der übergebene Sendepuffer ist zu groß, das Paket wurde auf die maximal mögliche Anzahl von Datenbytes begrenzt
9	Der übergebene Puffer ist zu klein, es wurden keine Daten kopiert
10	Der angegebene Host ist nicht bekannt
11	Pointer verweist auf ein Objekt eines nicht unterstützten Datentyps

3.3 Der Funktionsbaustein LAN_GET_HOST_CONFIG

Der Funktionsbaustein *LAN_GET_HOST_CONFIG* dient zum Ermitteln der lokalen Host-Konfiguration.

Prototyp des Funktionsbausteines



Operandenbedeutung

HOST_NAME	Stringvariable zur Aufnahme des lokalen Host-Namen der SPS
INET_ADDR	Lokale IP-Adresse der SPS
NUM_OF_SOCKETS	Anzahl der insgesamt für das SPS-Programm nutzbaren Sockets
SOCKETS_IN_USE	Anzahl der aktuell in Benutzung befindlichen Sockets
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)

ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum Ermitteln der lokalen Host-Konfiguration der SPS.

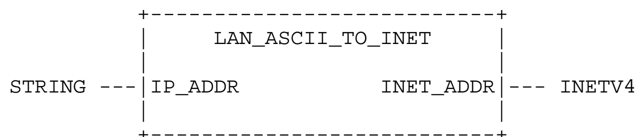
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.4 Die Funktion LAN_ASCII_TO_INET

Die Funktion *LAN_ASCII_TO_INET* wandelt eine als String in Standard "." Notation übergebene IP-Adresse in die entsprechende numerische Darstellung um.

Prototyp der Funktion


```
          +---------------------------+
          |     LAN_ASCII_TO_INET     |
          |                           |
STRING ---|IP_ADDR          INET_ADDR|--- INETV4
          |                           |
          +---------------------------+
```

Operandenbedeutung

IP_ADDR	String mit IP-Adresse in Standard "." Notation (z.B. '192.168.1.20')
INET_ADDR	numerische Darstellung der übergebenen IP-Adresse

Beschreibung

Diese Funktion wandelt den am Eingang *IP_ADDR* übergebenen String mit der IP-Adresse in Standard "." Notation (z.B. '192.168.1.20') in die entsprechende numerische Darstellung um und liefert diese am Ausgang *INET_ADDR* zurück. Die numerische Form der IP-Adresse wird von Funktionsbausteinen wie beispielsweise *LAN_GET_HOST_CONFIG*, *LAN_UDP_SENDTO_STR* oder *LAN_UDP_RECVFROM_STR* benutzt.

Die Funktion *LAN_ASCII_TO_INET* ist das Komplement zur Funktion *LAN_INET_TO_ASCII* (siehe Abschnitt 3.5).

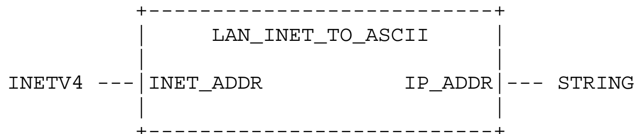
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.5 Die Funktion LAN_INET_TO_ASCII

Die Funktion *LAN_INET_TO_ASCII* wandelt eine in numerische Darstellung übergebene IP-Adresse in einen entsprechenden String mit Standard "." Notation um.

Prototyp der Funktion



```
              +---------------------------+
              |     LAN_INET_TO_ASCII     |
              |                           |
   INETV4 ---|INET_ADDR          IP_ADDR|--- STRING
              |                           |
              +---------------------------+
```

Operandenbedeutung

INET_ADDR	numerische Darstellung der IP-Adresse
IP_ADDR	String mit IP-Adresse in Standard "." Notation (z.B. '192.168.1.20')

Beschreibung

Diese Funktion wandelt die am Eingang *INET_ADDR* in numerische Darstellung übergebene IP-Adresse in einen entsprechenden String mit Standard "." Notation um und liefert diesen am Ausgang *IP_ADDR* zurück. Die numerische Form der IP-Adresse wird von Funktionsbausteinen wie beispielsweise *LAN_GET_HOST_CONFIG*, *LAN_UDP_SENDTO_STR* oder *LAN_UDP_RECVFROM_STR* benutzt. Mit Hilfe der Funktion *LAN_INET_TO_ASCII* kann diese numerische Repräsentation der IP-Adresse in einen darstellbaren String gewandelt werden.

Die Funktion *LAN_INET_TO_ASCII* ist das Komplement zur Funktion *LAN_ASCII_TO_INET* (siehe Abschnitt 3.4).

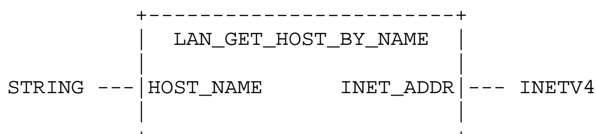
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.6 Die Funktion LAN_GET_HOST_BY_NAME

Die Funktion *LAN_GET_HOST_BY_NAME* ermittelt die IP-Adresse für den angegebenen Host-Name (nur auf Steuerungen mit DNS-Unterstützung verfügbar).

Prototyp der Funktion



```
              +------------------------+
              |  LAN_GET_HOST_BY_NAME  |
              |                        |
   STRING ---|HOST_NAME      INET_ADDR|--- INETV4
              |                        |
              +------------------------+
```

Operandenbedeutung

HOST_NAME	String mit Namen des zu suchenden Hostes
INET_ADDR	numerische Darstellung der ermittelten IP-Adresse

Beschreibung

Diese Funktion ermittelt die IP-Adresse für den am Eingang *HOST_NAME* angegebenen Host-Name und liefert diese am Ausgang *INET_ADDR* zurück. Die ermittelte IP-Adresse kann beispielsweise zum Aufruf des Funktionsbausteines *LAN_UDP_SENDTO_STR* benutzt werden.

Hinweis: Die Funktion *LAN_GET_HOST_BY_NAME* steht nur auf Steuerungen mit DNS-Unterstützung zur Verfügung.

Die Funktion *LAN_GET_HOST_BY_NAME* ist das Komplement zur Funktion *LAN_GET_HOST_BY_ADDR* (siehe Abschnitt 3.7).

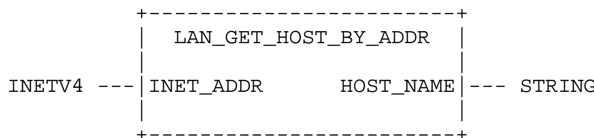
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.7 Die Funktion LAN_GET_HOST_BY_ADDR

Die Funktion *LAN_GET_HOST_BY_ADDR* ermittelt den Host-Name für die angegebene IP-Adresse (nur auf Steuerungen mit DNS-Unterstützung verfügbar).

Prototyp der Funktion



Operandenbedeutung

INET_ADDR	numerische Darstellung der aufzulösenden IP-Adresse
HOST_NAME	String mit Namen des ermittelten Hostes

Beschreibung

Diese Funktion ermittelt für die am Eingang *INET_ADDR* übergebene numerische IP-Adresse den entsprechenden Host-Namen und liefert diesen als String Ausgang *HOST_NAME* zurück. Die Funktion kann beispielsweise in Verbindung mit *LAN_UDP_RECVFROM_STR* benutzt werden, um die von diesem Funktionsbaustein zurück gelieferten IP-Adressen in Klartextnamen aufzulösen.

Hinweis: Die Funktion *LAN_GET_HOST_BY_NAME* steht nur auf Steuerungen mit DNS-Unterstützung zur Verfügung.

Die Funktion *LAN_GET_HOST_BY_ADDR* ist das Komplement zur Funktion *LAN_GET_HOST_BY_NAME* (siehe Abschnitt 3.6).

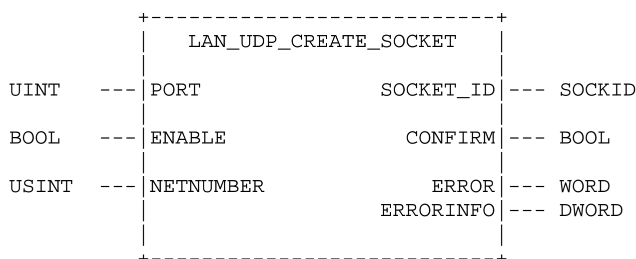
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.8 Der Funktionsbaustein LAN_UDP_CREATE_SOCKET

Der Funktionsbaustein *LAN_UDP_CREATE_SOCKET* legt ein Socket zum Senden oder Empfangen von Daten an.

Prototyp des Funktionsbausteines



Operandenbedeutung

PORT	Portnummer, an die das Socket gebunden werden soll (siehe Text)
SOCKET_ID	Socket-ID (eine von der UDP-Schicht der SPS vergebene interne Referenz auf das angelegte Socket)
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein legt ein Socket zum Senden oder Empfangen von Daten an. Ist das Socket für den Datenempfang vorgesehen, muss am Eingang *PORT* eine gültige Portnummer angegeben werden. Die SPS ruft in diesem Fall intern die Funktion *bind()* der UDP-Schicht auf und ist damit in der Lage, Datenpakete zu empfangen, die an ihre IP-Adresse mit der angegebenen Portnummer gesendet werden. Auf den meisten Systemen ist Nutzung von Portnummern kleiner 1024 nur privilegierten Prozessen erlaubt, weiterhin ist der Bereich von 1024 bis 49151 für Standard-anwendungen reserviert und wird von der IANA (Internet Assigned Numbers Authority) verwaltet. Für die UDP-Kommunikation mit der SPS sind daher nach Möglichkeit Portnummern aus dem privaten Bereich von 49152 bis 65535 zu bevorzugen.

Soll das angelegte Socket nur zum Senden von Daten verwendet werden, ist die Angabe der Portnummer optional. Ist der Eingang *PORT* auf Null gesetzt, benutzt die UDP-Schicht der SPS beim Senden intern eine freie Portnummer aus dem privaten Bereich. Der Aufruf der internen Funktion *bind()* innerhalb der UDP-Schicht entfällt dann. Die Festlegung einer definierten Portnummer kann aber auch beim Senden erforderlich sein, beispielsweise dann, wenn im Netzwerk eine Firewall aktiv ist, die den Datenverkehr nur für bestimmte Ports weiter leitet.

Bei seiner Rückkehr liefert der Funktionsbaustein *LAN_UDP_CREATE_SOCKET* am Ausgang *SOCKET_ID* eine von der UDP-Schicht der SPS vergebene interne Referenz auf das angelegte Socket zurück. Diese Socket-ID ist bei nachfolgenden Aufrufen von Funktionsbausteinen wie beispielsweise *LAN_UDP_SENDTO_STR* oder *LAN_UDP_RECVFROM_STR* zu übergeben.

Ein nicht mehr benötigtes Socket kann durch Aufruf von *LAN_UDP_CLOSE_SOCKET* wieder frei gegeben werden (siehe Abschnitt 0). Das Beenden des SPS-Programms führt intern automatisch zum Schließen aller belegten Sockets.

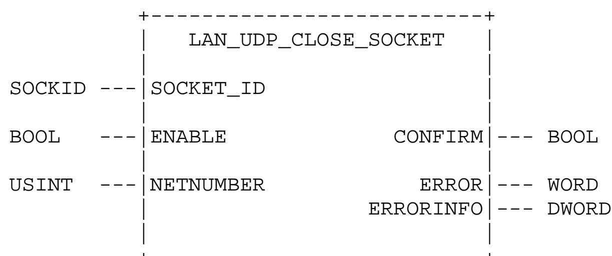
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.9 Der Funktionsbaustein LAN_UDP_CLOSE_SOCKET

Der Funktionsbaustein *LAN_UDP_CLOSE_SOCKET* dient zum expliziten Freigeben eines nicht mehr benötigten Sockets.

Prototyp des Funktionsbausteines



Operandenbedeutung

SOCKET_ID	Socket-ID des freizugebenden Sockets
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum expliziten Freigeben eines nicht mehr benötigten Sockets. Die *SOCKET_ID* ist die von der UDP-Schicht der SPS beim Aufruf von *LAN_UDP_CREATE_SOCKET* zurück gelieferte interne Referenz auf das betreffende Socket (siehe Abschnitt 3.8). Alle nicht explizit freigegebenen Sockets werden beim Beenden des SPS-Programms intern automatisch geschlossen.

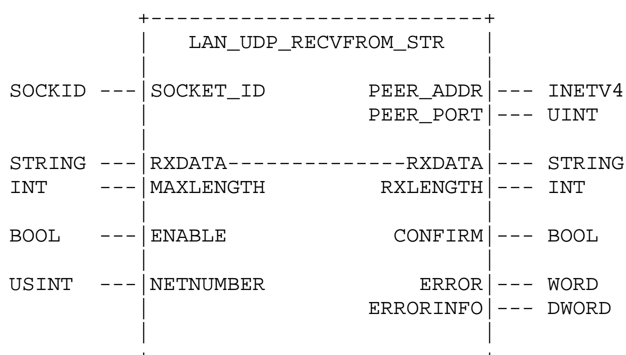
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.10 Der Funktionsbaustein LAN_UDP_RECVFROM_STR

Der Funktionsbaustein *LAN_UDP_RECVFROM_STR* dient zum Lesen von UDP-Paketen aus dem Empfangspuffer der UDP-Schicht.

Prototyp des Funktionsbausteines



Operandenbedeutung

SOCKET_ID	Socket-ID des abzufragenden Sockets
RXDATA	Stringvariable zur Aufnahme der gelesenen Zeichen
MAXLENGTH	Begrenzung der Anzahl zu lesender Zeichen, bei 0 wird intern die Pufferlänge des übergebenen Strings ermittelt und als Begrenzung der Anzahl zu lesender Zeichen verwendet (Hinweis: die Standard-Puffergröße eines Strings in OpenPCS beträgt 32 Zeichen).
RXLENGTH	Länge der gelesenen Zeichenfolge
PEER_ADDR	numerische Darstellung der IP-Adresse der Gegenstelle, von der das Paket empfangen wurde
PEER_PORT	Portnummer, die von der Gegenstelle zum Senden der Daten verwendet wurde
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)

ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum Lesen von UDP-Paketen aus dem Empfangspuffer der UDP-Schicht. Ist bei der Rückkehr des Funktionsbausteines der Ausgang *CONFIRM* auf TRUE gesetzt, enthält die als Input/Output-Parameter *RXDATA* angegebene Stringvariable die empfangene Zeichenkette. Der Ausgang *RXLENGTH* gibt die Anzahl der in *RXDATA* abgelegten Zeichen an. Ist der Ausgang *CONFIRM* bei der Rückkehr des Bausteins auf FALSE gesetzt, wurden über den angegebenen Socket keine Zeichen empfangen.

Beim Empfang von Paketen (*CONFIRM* auf TRUE gesetzt) erhalten die Ausgänge *PEER_ADDR* und *PEER_PORT* die Informationen über die IP-Adresse der Gegenstelle sowie die zum Senden von ihr verwendete Portnummer. Soll die SPS auf dieses empfangene Paket antworten, sind die Werte von *PEER_ADDR* und *PEER_PORT* als Zielangaben für den nachfolgenden Aufruf des Bausteins *LAN_UDP_SENDTO_STR* zu verwenden (siehe Abschnitt 3.11):

```
LAN_UDP_SENDTO_STR.PEER_ADDR := LAN_UDP_RECVFROM_STR.PEER_ADDR;
LAN_UDP_SENDTO_STR.PEER_PORT := LAN_UDP_RECVFROM_STR.PEER_PORT;
```

Der für den Empfang von Paketen zu benutzende Socket muss vor seiner Verwendung mit Hilfe des Funktionsbausteins *LAN_UDP_CREATE_SOCKET* unter Angabe einer gültigen Portnummer angelegt worden sein (siehe Abschnitt 0).

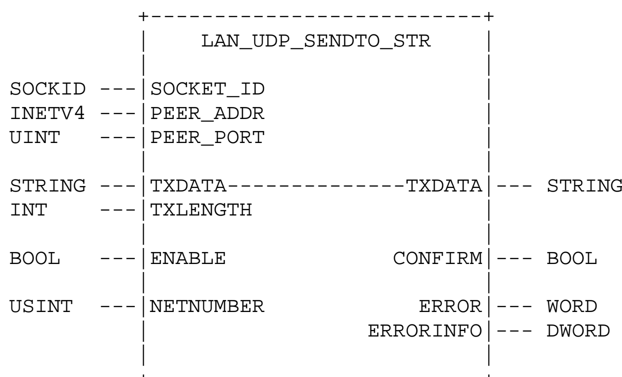
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.11 Der Funktionsbaustein LAN_UDP_SENDTO_STR

Der Funktionsbaustein *LAN_UDP_SENDTO_STR* dient zum Senden von UDP-Paketen.

Prototyp des Funktionsbausteines



Operandenbedeutung

SOCKET_ID	Socket-ID des zum Senden zu benutzenden Sockets
PEER_ADDR	numerische Darstellung der IP-Adresse der Gegenstelle, an die das Paket gesendet werden soll
PEER_PORT	Portnummer der Gegenstelle, an die das Paket gesendet werden soll
TXDATA	Stringvariable mit der zu sendenden Zeichenfolge
TXLENGTH	Anzahl der zu sendenden Zeichen, bei 0 wird intern die Länge der im String TXDATA enthaltenen Zeichenfolge ermittelt (entspricht LEN(TXDATA);) und als Anzahl der zu sendenden Zeichen verwendet.
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum Senden von UDP-Paketen. Die Eingänge *PEER_ADDR* und *PEER_PORT* enthalten die Adressinformationen der Gegenstelle an die das Paket gesendet werden soll. Ist das zu sendende Paket eine Antwort auf eine zuvor mit dem Funktionsbaustein *LAN_UDP_RECVFROM_STR* empfangene Nachricht, sind hier die Absender-Adressinformationen zu übernehmen (siehe Abschnitt 3.10).

Der für das Senden von Paketen zu benutzende Socket muss vor seiner Verwendung mit Hilfe des Funktionsbausteins *LAN_UDP_CREATE_SOCKET* angelegt worden sein (siehe Abschnitt 3.8).

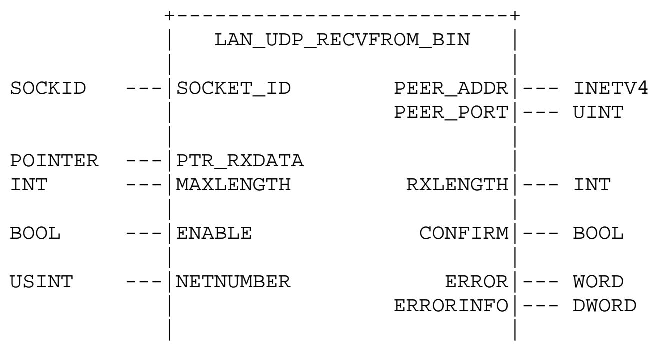
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.12 Der Funktionsbaustein LAN_UDP_RECVFROM_BIN

Der Funktionsbaustein *LAN_UDP_RECVFROM_BIN* dient zum Lesen von UDP-Paketen aus dem Empfangspuffer der UDP-Schicht.

Prototyp des Funktionsbausteines



Operandenbedeutung

SOCKET_ID	Socket-ID des abzufragenden Sockets
PTR_RXDATA	Adresse eines Objektes, in dem die empfangenen Daten abgelegt werden
MAXLENGTH	Begrenzung der Anzahl abzulegender Bytes, bei 0 wird intern die Größe des über PTR_RXDATA adressierten Objektes ermittelt und als Begrenzung der Anzahl abzulegender Bytes verwendet (es werden max. so viele Bytes abgelegt, wie das Objekt aufnehmen kann)
RXLENGTH	Anzahl der abgelegten Bytes
PEER_ADDR	numerische Darstellung der IP-Adresse der Gegenstelle, von der das Paket empfangen wurde
PEER_PORT	Portnummer, die von der Gegenstelle zum Senden der Daten verwendet wurde
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum Lesen von UDP-Paketen aus dem Empfangspuffer der UDP-Schicht. Ist bei der Rückkehr des Funktionsbausteines der Ausgang *CONFIRM* auf TRUE gesetzt, enthält das über *PTR_RXDATA* adressierte Objekt die Daten des empfangenen Paketes. Der Ausgang *RXLENGTH* gibt dabei die Anzahl der gültigen Datenbytes an. Ist der Ausgang *CONFIRM* bei der Rückkehr des Bausteins auf FALSE gesetzt, wurden über den angegebenen Socket keine Daten empfangen.

Beim Empfang von Paketen (*CONFIRM* auf TRUE gesetzt) erhalten die Ausgänge *PEER_ADDR* und *PEER_PORT* die Informationen über die IP-Adresse der Gegenstelle sowie die zum Senden von ihr

verwendete Portnummer. Soll die SPS auf dieses empfangene Paket antworten, sind die Werte von *PEER_ADDR* und *PEER_PORT* als Zielangaben für den nachfolgenden Aufruf des Bausteins *LAN_UDP_SENDTO_BIN* zu verwenden (siehe Abschnitt 3.13):

```
LAN_UDP_SENDTO_BIN.PEER_ADDR := LAN_UDP_RECVFROM_BIN.PEER_ADDR;
LAN_UDP_SENDTO_BIN.PEER_PORT := LAN_UDP_RECVFROM_BIN.PEER_PORT;
```

Der für den Empfang von Paketen zu benutzende Socket muss vor seiner Verwendung mit Hilfe des Funktionsbausteins *LAN_UDP_CREATE_SOCKET* unter Angabe einer gültigen Portnummer angelegt worden sein (siehe Abschnitt 0).

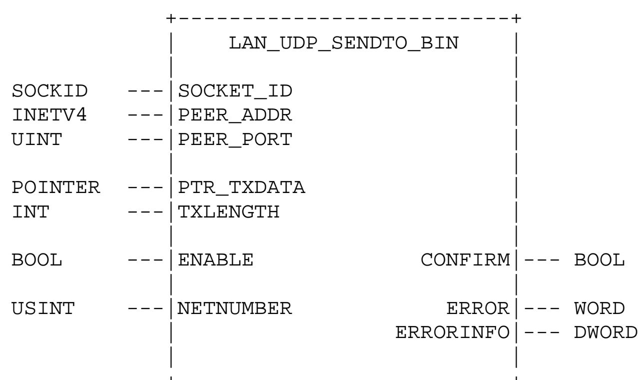
Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.13 Der Funktionsbaustein LAN_UDP_SENDTO_BIN

Der Funktionsbaustein *LAN_UDP_SENDTO_BIN* dient zum Senden von UDP-Paketen.

Prototyp des Funktionsbausteines



Operandenbedeutung

SOCKET_ID	Socket-ID des zum Senden zu benutzenden Sockets
PEER_ADDR	numerische Darstellung der IP-Adresse der Gegenstelle, an die das Paket gesendet werden soll
PEER_PORT	Portnummer der Gegenstelle, an die das Paket gesendet werden soll
PTR_TXDATA	Adresse eines Objektes, in dem die zu sendenden Daten enthalten sind
TXLENGTH	Anzahl der zu sendenden Zeichen, bei 0 wird intern die Größe des über PTR_TXDATA adressierten Objektes ermittelt und als Anzahl zu sendender Bytes verwendet
NETNUMBER	Netzwerknummer (Hinweis: falls die SPS nur ein Ethernet-Interface unterstützt, kann das Setzen dieses Eingangs entfallen, da numerische Variablen gemäß IEC61131 bereits mit dem Initialwert 0 vorbelegt sind)

ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 8 definiert.
ERRORINFO	reserviert für zusätzliche Fehlerinformationen
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB
CONFIRM	Ausgang für Fertigmeldung durch den FB

Beschreibung

Der Funktionsbaustein dient zum Senden von UDP-Paketen. Die Eingänge *PEER_ADDR* und *PEER_PORT* enthalten die Adressinformationen der Gegenstelle an die das Paket gesendet werden soll. Ist das zu sendende Paket eine Antwort auf eine zuvor mit dem Funktionsbaustein *LAN_UDP_RECVFROM_BIN* empfangene Nachricht, sind hier die Absender-Adressinformationen zu übernehmen (siehe Abschnitt 3.12).

Der für das Senden von Paketen zu benutzende Socket muss vor seiner Verwendung mit Hilfe des Funktionsbausteins *LAN_UDP_CREATE_SOCKET* angelegt worden sein (siehe Abschnitt 3.8).

Programmbeispiel

Ein ausführliches Programmbeispiel im Abschnitt 3.14 verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine.

3.14 Programmbeispiel zur Anwendung der UDP-Bausteine

Das folgende Programmbeispiel verdeutlicht die Anwendung aller im Abschnitt 3 beschriebenen UDP-Bausteine. Das Programmbeispiel realisiert einen einfachen Server, der Kommandos in Form von Strings entgegen nimmt, diese ausführt und einen entsprechenden Antwortstring mit dem Ausführungsergebnis an den Client zurück sendet. Zunächst wird der Baustein *LAN_UDP_CREATE_SOCKET* aufgerufen, um einen Socket für den Datenaustausch mit dem Client anzulegen. Anschließend verharrt der Server so lange im nachfolgenden Programmschritt, bis der Baustein *LAN_UDP_RECVFROM_STR* den Empfang eines Kommandos von einem Client signalisiert. Nach der Interpretation und Ausführung des Kommandos (im anwenderspezifischen Funktionsbaustein "*ExecCommand*", hier nicht mit dargestellt) sendet das Beispielprogramm den generierten Antwortstring mit Hilfe des Bausteins *LAN_UDP_SENDTO_STR* an den Client zurück. Als Adressinformation für den Aufruf von *LAN_UDP_SENDTO_STR* werden dabei die zuvor beim Kommandoempfang von *LAN_UDP_CREATE_SOCKET* gelieferte IP-Adresse und Portnummer übernommen.

Programmbeispiel

```
PROGRAM UdpServer
VAR CONSTANT
    NETNUMBER       : USINT     := 0;
    SVRPORT         : UINT      := 55555;
    STOP_CMD        : STRING    := 'stop';
END_VAR

VAR
    xServerRunning  : BOOL;

    FB_ExecCommand  : ExecCommand;
    strRxCommand    : STRING(128);
    strTxResult     : STRING(250);
```

```
    FB_LanGetHostConfig : LAN_GET_HOST_CONFIG;
    strPlcHostName  : STRING(64);
    inetPlcIpAddr   : INETV4;
    uiNumOfSockets  : UINT;
    uiSocketsInUse  : UINT;
    strPlcIpAddr    : STRING;

    FB_LanUdpCreateSocket : LAN_UDP_CREATE_SOCKET;
    FB_LanUdpCloseSocket  : LAN_UDP_CLOSE_SOCKET;
    SocketID        : SOCKID;

    FB_LanUdpRecvfromStr : LAN_UDP_RECVFROM_STR;
    strRxData       : STRING(128);
    inetPeerIpAddr  : INETV4;
    uiPeerPort      : UINT;
    iRxDataLen      : INT;
    strRxDataLen    : STRING;
    strPeerIpAddr   : STRING;
    strPeerPort     : STRING;

    FB_LanUdpSendtoStr : LAN_UDP_SENDTO_STR;
    strTxData       : STRING(250);

    uiProcState     : UINT := 0;
END_VAR


(* ========== Program UdpServer ========== *)

CASE uiProcState OF

    (* ----- Inititialization ----- *)
    0:
        (*-----------------------------------------------------*)
        (* The following block is not really necessary in this  *)
        (* application but it shows how to use some additional  *)
        (* LAN function(-blocks) which are maybe be helpful in  *)
        (* other projects.                                      *)
        FB_LanGetHostConfig (
            ENABLE    := TRUE,
            NETNUMBER := NETNUMBER,
            HOST_NAME := strPlcHostName
            |
            inetPlcIpAddr  := INET_ADDR,
            uiNumOfSockets := NUM_OF_SOCKETS,
            uiSocketsInUse := SOCKETS_IN_USE);

        strPlcIpAddr  := LAN_INET_TO_ASCII (inetPlcIpAddr);
        inetPlcIpAddr := LAN_ASCII_TO_INET (strPlcIpAddr);

        strPlcHostName := LAN_GET_HOST_BY_ADDR (inetPlcIpAddr);
        inetPlcIpAddr  := LAN_GET_HOST_BY_NAME (strPlcHostName);
        (*---------------------------------------------------*)

        (* ... continue with realy serious stuff for this application... *)
        FB_LanUdpCreateSocket (
            ENABLE := TRUE,
            NETNUMBER := NETNUMBER,
            PORT := SVRPORT
            |
            SocketID := SOCKET_ID);

        xServerRunning := TRUE;
        uiProcState := uiProcState + 1;     (* new state: Wait for Receipt *)
```

```
(* ----- Wait for Receipt ----- *)
1:
    (* Because this application acts as a server it is      *)
    (* necessary to save the output values PEER_ADDR and    *)
    (* PEER_PORT from the FB LAN_UDP_RECVFROM_STR. This     *)
    (* both parameters indentifies the client host from     *)
    (* which the command/request was receipt. They are used *)
    (* later to send back the answer to the peer client     *)
    (* via FB LAN_UDP_SENDTO_STR.                           *)
    FB_LanUdpRecvfromStr (
        ENABLE := TRUE,
        NETNUMBER := NETNUMBER,
        SOCKET_ID := SocketID,
        MAXLENGTH := 0,                 (* use StrAllocLen of strRxData *)
        RXDATA := strRxData
        |
        inetPeerIpAddr := PEER_ADDR,
        uiPeerPort     := PEER_PORT,
        iRxDataLen     := RXLENGTH);

    IF (FB_LanUdpRecvfromStr.CONFIRM = TRUE) THEN
        uiProcState := uiProcState + 1; (* new state: Process Command *)
    END_IF;



(* ----- Process Command ----- *)
2:
    IF (strRxData = STOP_CMD) THEN
        xServerRunning := FALSE;
    END_IF;

    IF (xServerRunning = TRUE) THEN
        (* execute command *)
        strRxCommand := strRxData;
        FB_ExecCommand (
            strCommand_i := strRxCommand
            |
            strTxResult := strResult_o);
    ELSE
        (* show good-by message *)
        strTxResult := '$NServer stopped.$N';
    END_IF;

    (* create answer string *)
    strRxDataLen  := INT_TO_STRING(iRxDataLen);
    strPeerIpAddr := LAN_INET_TO_ASCII(inetPeerIpAddr);
    strPeerPort   := UINT_TO_STRING(uiPeerPort);

    strTxData := CONCAT ('$NPLC: ', strRxDataLen, ' Byte(s) received ',
                         'from IP-Address=', strPeerIpAddr, '/',
                         'Port=', strPeerPort, ':',
                         '$N-> Command: ', strRxData,
                         '$N-> Result:  ', strTxResult);

    uiProcState := uiProcState + 1;     (* new state: Send Response *)
```

```
    (* ----- Send Response ----- *)
    3:
        (* The values PEER_ADDR and PEER_PORT identifies the    *)
        (* client host, to which the answer should be send now. *)
        (* Both values was output parameters from a previous    *)
        (* call of the FB LAN_UDP_RECVFROM_STR.                 *)
        FB_LanUdpSendtoStr (
            ENABLE := TRUE,
            NETNUMBER := NETNUMBER,
            SOCKET_ID := SocketID,
            PEER_ADDR := inetPeerIpAddr,
            PEER_PORT := uiPeerPort,
            TXDATA := strTxData,
            TXLENGTH := 0);

        IF (xServerRunning = TRUE) THEN
            uiProcState := uiProcState - 2; (* go back to reveive state *)
        ELSE
            uiProcState := uiProcState + 1; (* goto finish state *)
        END_IF;



    (* ----- Finish Server ----- *)
    4:
        FB_LanUdpCloseSocket (
            ENABLE := TRUE,
            NETNUMBER := NETNUMBER,
            SOCKET_ID := SocketID);

        uiProcState := uiProcState + 1;



    (* ----- Stop State ----- *)
    5:
        ;   (* simply do nothing *)


    (* --- unknown state --- *)
    ELSE
        uiProcState := 0;

END_CASE;


RETURN;

END_PROGRAM
```

4 Sicherung von Prozessdaten im nichtflüchtigen Speicher

4.1 Anwendung der nichtflüchtigen Speicherung von Prozessdaten

Die in einem SPS-Programm definierten Variablen können die in ihnen enthaltenen Informationen nur während der Programmlaufzeit speichern. Beim Beenden des Programmes oder beim Ausschalten der SPS gehen diese Informationen im Regelfall verloren. Der in den Abschnitten 4.2, 4.3 und 4.4 beschriebenen Funktionsbaustein vom Typ *NVDATA_Xxx* (NV = Non Volatile) bietet die Möglichkeit, Prozessdaten in einem nichtflüchtigen Speicher abzulegen.

Die Sicherung von Prozessdaten im nichtflüchtigen Speicher erlaubt einem SPS-Programm beispielsweise die kontinuierliche Fortführung von Produktionszählern auch nach dem Wiederanlauf einer Anlage. Ebenso wird die remanente Speicherung von Parametern ermöglicht, die vom Anwender z.B. über ein Bediengerät zur Laufzeit der Anlage umkonfiguriert wurden.

4.2 Der Funktionsbaustein NVDATA_BIT

Der Funktionsbaustein *NVDATA_BIT* dient zum Schreiben logischer Prozessdaten (BYTE, WORD, DWORD) in den nichtflüchtigen Speicher (EEPROM, Datei) einer SPS sowie zum Rücklesen von gespeicherten Prozessdaten aus dem nichtflüchtigen Speicher.

Prototyp des Funktionsbausteines

```
            +--------------+
            |  NVDATA_BIT  |
            |              |
BYTE    ---|DIN1      DOUT1|--- BYTE
WORD    ---|DIN2      DOUT2|--- WORD
DWORD   ---|DIN3      DOUT3|--- DWORD
            |              |
UINT    ---|ADDR       SIZE|--- UINT
USINT   ---|MODE           |
            |              |
USINT   ---|DEVICE    ERROR|--- USINT
            |              |
            +--------------+
```

Operandenbedeutung

DIN1	Dateneingang zum Schreiben eines BYTE-Wertes
DIN2	Dateneingang zum Schreiben eines WORD-Wertes
DIN3	Dateneingang zum Schreiben eines DWORD-Wertes
ADDR	Adresse innerhalb des nichtflüchtigen Speichers zum Lesen oder Schreiben von Daten (abhängig vom Parameter *MODE*)
MODE	Festlegung der auszuführenden Lese- oder Schreiboperation, eine Auflistung der unterstützten Modi enthält Tabelle 9
DOUT1	Datenausgang zum Lesen eines BYTE-Wertes
DOUT2	Datenausgang zum Lesen eines WORD-Wertes
DOUT3	Datenausgang zum Lesen eines DWORD-Wertes

SIZE — Dieser Ausgang gibt die Anzahl der geschriebenen bzw. gelesenen Bytes an (*MODE* <> *0*) oder liefert die Größe des nutzbaren nichtflüchtigen Speichers (*MODE* = *0*, siehe Text).

ERROR: — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 10 definiert.

DEVICE — Device-Nummer, dieser Parameter ist abhängig von der jeweiligen Steuerung (Hinweis: die meisten Steuerungen unterstützen nur das Device 0, somit kann das Setzen dieses Einganges entfallen, da er bereits mit dem Initialwert 0 vorbelegt ist)

Tabelle 9: Aufruf-Modi für den Funktionsbaustein NVDATA_BIT

Mode	Aktion
16#00	Größe des nutzbaren nichtflüchtigen Speichers ermitteln (siehe Text)
16#01	Lesen eines BYTE am Datenausgang *DOUT1* aus dem nichtflüchtigen Speicher
16#02	Lesen eines WORD am Datenausgang *DOUT2* aus dem nichtflüchtigen Speicher
16#03	Lesen eines DWORD am Datenausgang *DOUT3* aus dem nichtflüchtigen Speicher
16#81	Schreiben eines BYTE am Dateneingang *DIN1* in den nichtflüchtigen Speicher
16#82	Schreiben eines WORD am Dateneingang *DIN2* in den nichtflüchtigen Speicher
16#83	Schreiben eines DWORD am Dateneingang *DIN3* in den nichtflüchtigen Speicher

Tabelle 10: Error-Codes der Funktionsbausteine NVDATA_Xxx

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Ungültige Device-Nummer (*DEVICE*) beim Aufruf des Funktionsbausteines
4	Ungültiger Modus (*MODE*) beim Aufruf des Funktionsbausteines
8	Angegebene Adresse (*ADDR*) zu groß, maximal verfügbarer Speicherbereich überschritten
16	Pointer verweist auf ein Objekt eines nicht unterstützten Datentyps

Beschreibung

Mit Hilfe des Funktionsbausteines können Daten verschiedenen Typs in einen nichtflüchtigen Speicher (EEPROM, Datei) geschrieben bzw. ausgelesen werden. In Abhängigkeit der zu lesenden bzw. zu schreibenden Daten ist am Eingang *MODE* der entsprechende Modus entsprechend Tabelle 9 zu setzen. Dabei ist darauf zu achten, dass je nach gewähltem Modus die Daten am assoziierten Dateneingang angelegt bzw. vom assoziierten Datenausgang gelesen werden. Der am Eingang *ADDR* übergebene Wert legt die Basisadresse für die auszuführende Lese- oder Schreiboperation fest. Wird dabei über die maximale Speichergröße hinaus adressiert, kehrt der Funktionsbaustein mit einem entsprechenden Fehler zurück. Die Aufteilung des verfügbaren Speichers liegt vollständig in der Verantwortung des SPS-Programmes, so ist es auch die Aufgabe des SPS-Programmes sicherzustellen, dass die am Eingang *ADDR* verwendeten Werte nicht zu Überschneidungen der zu sichernden Daten führen. Am Ausgang *SIZE* wird die Anzahl der jeweils geschriebenen oder gelesenen Bytes zurückgeliefert, so dass dieser Wert zur Berechnung der nächsten freien Adresse verwendet werden kann ($ADDR_{new} := ADDR_{old} + SIZE$).

Der Aufruf des Bausteins mit *MODE = 0* dient zur Größenbestimmung des nutzbaren nichtflüchtigen Speichers. Am Ausgang *SIZE* wird dazu die verbleibende Restgröße ab dem am Eingang *ADDR* übergebenen Wert zurück geliefert ($SIZE := NVDATA_{FullSize} - ADDR$). Um die Gesamtgröße des nichtflüchtigen Speichers zu ermitteln, ist der Baustein mit *ADDR := 0* aufzurufen.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 10 beschrieben.

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *NVDATA_BIT*. Dabei wird ein Datenbyte zunächst ab Adresse 10 geschrieben und anschließend von der gleichen Adresse zurück gelesen. Da der Eingang *DEVICE* nicht durch das Anwenderprogramm gesetzt wird, bleibt die Standardeinstellung erhalten der Baustein benutzt somit implizit die Device-Nummer 0.

Programmbeispiel

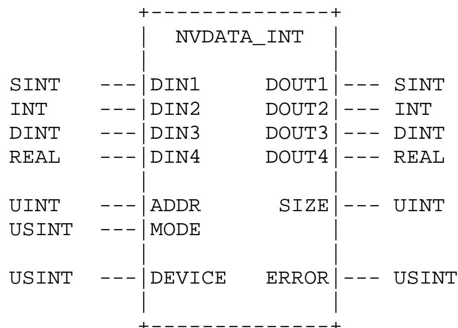
```
PROGRAM SaveDataBit

VAR CONSTANT
    NVDBIT_MODE_GET_SIZE  : USINT := 16#00;
    NVDBIT_MODE_RD_BYTE   : USINT := 16#01;
    NVDBIT_MODE_RD_WORD   : USINT := 16#02;
    NVDBIT_MODE_RD_DWORD  : USINT := 16#03;
    NVDBIT_MODE_WR_BYTE   : USINT := 16#81;
    NVDBIT_MODE_WR_WORD   : USINT := 16#82;
    NVDBIT_MODE_WR_DWORD  : USINT := 16#83;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR

VAR
    WriteDataByte : BYTE;
    WriteDataSize : UINT;
    ReadDataByte : BYTE;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataBit : NVDATA_BIT;
END_VAR
```

```
(* write a BYTE value into EEPROM *)
LD      16#10
ST      WriteDataByte

CAL     FB_NvDataBit (
        DIN1 := WriteDataByte,
        ADDR := 10,
        MODE := NVDBIT_MODE_WR_BYTE
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a BYTE value from EEPROM *)
CAL     FB_NvDataBit (
        ADDR := 10,
        MODE := NVDBIT_MODE_RD_BYTE
        |
        ReadDataByte := DOUT1,
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.3 Der Funktionsbaustein NVDATA_INT

Der Funktionsbaustein *NVDATA_INT* dient zum Schreiben arithmetischer Prozessdaten (SINT, INT, DINT, REAL) in den nichtflüchtigen Speicher (EEPROM, Datei) einer SPS sowie zum Rücklesen von gespeicherten Prozessdaten aus dem nichtflüchtigen Speicher.

Prototyp des Funktionsbausteines



Operandenbedeutung

DIN1	Dateneingang zum Schreiben eines SINT-Wertes
DIN2	Dateneingang zum Schreiben eines INT-Wertes
DIN3	Dateneingang zum Schreiben eines DINT-Wertes
DIN4	Dateneingang zum Schreiben eines REAL-Wertes
ADDR	Adresse innerhalb des nichtflüchtigen Speichers zum Lesen oder Schreiben von Daten (abhängig vom Parameter *MODE*)
MODE	Festlegung der auszuführenden Lese- oder Schreiboperation, eine Auflistung der unterstützten Modi enthält Tabelle 11.

DOUT1 Datenausgang zum Lesen eines SINT-Wertes
DOUT2 Datenausgang zum Lesen eines INT-Wertes
DOUT3 Datenausgang zum Lesen eines DINT-Wertes
DOUT4 Datenausgang zum Lesen eines RAEL-Wertes

SIZE Dieser Ausgang gibt die Anzahl der geschriebenen bzw. gelesenen Bytes an (*MODE* <> *0*) oder liefert die Größe des nutzbaren nichtflüchtigen Speichers (*MODE = 0*, siehe Text).

ERROR: Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 10 definiert (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

DEVICE Device-Nummer, dieser Parameter ist abhängig von der jeweiligen Steuerung (Hinweis: die meisten Steuerungen unterstützen nur das Device 0, somit kann das Setzen dieses Einganges entfallen, da er bereits mit dem Initialwert 0 vorbelegt ist)

Tabelle 11: Aufruf-Modi für den Funktionsbaustein NVDATA_INT

Mode	Aktion
16#00	Größe des nutzbaren nichtflüchtigen Speichers ermitteln (siehe Text)
16#01	Lesen eines SINT am Datenausgang *DOUT1* aus dem nichtflüchtigen Speicher
16#02	Lesen eines INT am Datenausgang *DOUT2* aus dem nichtflüchtigen Speicher
16#03	Lesen eines DINT am Datenausgang *DOUT3* aus dem nichtflüchtigen Speicher
16#04	Lesen eines REAL am Datenausgang *DOUT4* aus dem nichtflüchtigen Speicher
16#81	Schreiben eines SINT am Dateneingang *DIN1* in den nichtflüchtigen Speicher
16#82	Schreiben eines INT am Dateneingang *DIN2* in den nichtflüchtigen Speicher
16#83	Schreiben eines DINT am Dateneingang *DIN3* in den nichtflüchtigen Speicher
16#84	Schreiben eines REAL am Dateneingang *DIN4* in den nichtflüchtigen Speicher

Beschreibung

Mit Hilfe des Funktionsbausteines können Daten verschiedenen Typs in einen nichtflüchtigen Speicher (EEPROM, Datei) geschrieben bzw. ausgelesen werden. In Abhängigkeit der zu lesenden bzw. zu schreibenden Daten ist am Eingang *MODE* der entsprechende Modus entsprechend Tabelle 11 zu setzen. Dabei ist darauf zu achten, dass je nach gewähltem Modus die Daten am assoziierten Dateneingang angelegt bzw. vom assoziierten Datenausgang gelesen werden. Der am Eingang *ADDR* übergebene Wert legt die Basisadresse für die auszuführende Lese- oder Schreiboperation fest. Wird dabei über die maximale Speichergröße hinaus adressiert, kehrt der Funktionsbaustein mit einem entsprechenden Fehler zurück. Die Aufteilung des verfügbaren Speichers liegt vollständig in der Verantwortung des SPS-Programmes, so ist es auch die Aufgabe des SPS- Programmes sicherzustellen, dass die am Eingang *ADDR* verwendeten Werte nicht zu Überschneidungen der zu sichernden Daten führen. Am Ausgang *SIZE* wird die Anzahl der jeweils geschriebenen oder gelesenen Bytes zurückgeliefert, so dass dieser Wert zur Berechnung der nächsten freien Adresse verwendet werden kann ($ADDR_{new} := ADDR_{old} + SIZE$).

Der Aufruf des Bausteins mit *MODE = 0* dient zur Größenbestimmung des nutzbaren nichtflüchtigen Speichers. Am Ausgang *SIZE* wird dazu die verbleibende Restgröße ab dem am Eingang *ADDR* übergebenen Wert zurück geliefert ($SIZE := NVDATA_{FullSize} - ADDR$). Um die Gesamtgröße des nichtflüchtigen Speichers zu ermitteln, ist der Baustein mit *ADDR := 0* aufzurufen.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 10 beschrieben (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *NVDATA_INT*. Dabei wird ein REAL-Wert zunächst ab Adresse 20 geschrieben und anschließend von der gleichen Adresse zurück gelesen. Da der Eingang *DEVICE* nicht durch das Anwenderprogramm gesetzt wird, bleibt die Standardeinstellung erhalten der Baustein benutzt somit implizit die Device-Nummer 0.

Programmbeispiel

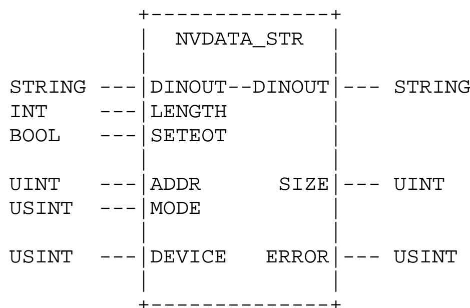
```
PROGRAM SaveDataInt

VAR CONSTANT
    NVDINT_MODE_GET_SIZE  : USINT := 16#00;
    NVDINT_MODE_RD_SINT   : USINT := 16#01;
    NVDINT_MODE_RD_INT    : USINT := 16#02;
    NVDINT_MODE_RD_DINT   : USINT := 16#03;
    NVDINT_MODE_RD_REAL   : USINT := 16#04;
    NVDINT_MODE_WR_SINT   : USINT := 16#81;
    NVDINT_MODE_WR_INT    : USINT := 16#82;
    NVDINT_MODE_WR_DINT   : USINT := 16#83;
    NVDINT_MODE_WR_REAL   : USINT := 16#84;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR

VAR
    WriteDataReal : REAL;
    WriteDataSize : UINT;
    ReadDataReal : REAL;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataInt : NVDATA_INT;
END_VAR
```

```
(* write a REAL value into EEPROM *)
LD      7.89
ST      WriteDataReal

CAL     FB_NvDataInt (
        DIN4 := WriteDataReal,
        ADDR := 20,
        MODE := NVDINT_MODE_WR_REAL
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a REAL value from EEPROM *)
CAL     FB_NvDataInt (
        ADDR := 20,
        MODE := NVDINT_MODE_RD_REAL
        |
        ReadDataReal := DOUT4,
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.4 Der Funktionsbaustein NVDATA_STR

Der Funktionsbaustein *NVDATA_STR* dient zum Schreiben zeichenkettenbasierter Prozessdaten (STRING) in den nichtflüchtigen Speicher (EEPROM, Datei) einer SPS sowie zum Rücklesen von gespeicherten Prozessdaten aus dem nichtflüchtigen Speicher.

Prototyp des Funktionsbausteines



Operandenbedeutung

DINOUT	Datenein- und Ausgang zum Schreiben bzw. Lesen eines STRING-Wertes
LENGTH	Begrenzung der Anzahl zu lesender bzw. zu schreibender Zeichen, bei 0 wird intern die Pufferlänge des übergebenen Strings ermittelt und als Anzahl der zu lesenden bzw. zu schreibenden Zeichen verwendet (entspricht *LEN(DINOUT);*) Hinweis: die Standard-Puffergröße eines Strings in OpenPCS beträgt 32 Zeichen.
ADDR	Adresse innerhalb des nichtflüchtigen Speichers zum Lesen oder Schreiben von Daten (abhängig vom Parameter *MODE*)
MODE	Festlegung der auszuführenden Lese- oder Schreiboperation, eine Auflistung der unterstützten Modi enthält Tabelle 12.

SETEOT — TRUE: Der String wird mit Endekennzeichen gespeichert
FALSE: Die Speicherung des Endekennzeichens wird unterdrückt (Default: TRUE, siehe Text)

SIZE — Dieser Ausgang gibt die Anzahl der geschriebenen bzw. gelesenen Bytes an (*MODE* <> *0*) oder liefert die Größe des nutzbaren nichtflüchtigen Speichers (*MODE* = *0*, siehe Text).

ERROR: — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 10 definiert (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

DEVICE — Device-Nummer, dieser Parameter ist abhängig von der jeweiligen Steuerung (Hinweis: die meisten Steuerungen unterstützen nur das Device 0, somit kann das Setzen dieses Einganges entfallen, da er bereits mit dem Initialwert 0 vorbelegt ist)

Tabelle 12: Aufruf-Modi für den Funktionsbaustein NVDATA_STR

Mode	Aktion
16#00	Größe des nutzbaren nichtflüchtigen Speichers ermitteln (siehe Text)
16#08	Lesen eines STRING am Datenausgang *DINOUT* aus dem nichtflüchtigen Speicher
16#88	Schreiben eines STRING am Dateneingang *DINOUT* in den nichtflüchtigen Speicher

<u>Beschreibung</u>

Mit Hilfe des Funktionsbausteines können zeichenkettenbasierte Daten in einen nichtflüchtigen Speicher (EEPROM, Datei) geschrieben bzw. ausgelesen werden. In Abhängigkeit der zu lesenden bzw. zu schreibenden Daten ist am Eingang *MODE* der entsprechende Modus entsprechend Tabelle 12 zu setzen. Abhängig vom Modus wird der Parameter *DINOUT* dabei als Ein- oder Ausgang verwendet. Der am Eingang *ADDR* übergebene Wert legt die Basisadresse für die auszuführende Lese- oder Schreiboperation fest. Wird dabei über die maximale Speichergröße hinaus adressiert, kehrt der Funktionsbaustein mit einem entsprechenden Fehler zurück. Die Aufteilung des verfügbaren Speichers liegt vollständig in der Verantwortung des SPS-Programmes, so ist es auch die Aufgabe des SPS-Programmes sicherzustellen, dass die am Eingang *ADDR* verwendeten Werte nicht zu Überschneidungen der zu sichernden Daten führen. Am Ausgang *SIZE* wird die Anzahl der jeweils geschriebenen oder gelesenen Bytes zurückgeliefert, so dass dieser Wert zur Berechnung der nächsten freien Adresse verwendet werden kann ($ADDR_{new} := ADDR_{old} + SIZE$).

Der Eingang *LENGTH* spezifiziert beim Schreiben die Anzahl der gültigen Zeichen. Ist dieser Wert 0, wird intern die Länge der im String enthaltenen Zeichenfolge ermittelt (entspricht *LEN(DINOUT);*) und als Anzahl der zu schreibenden Zeichen verwendet. In diesem Fall wird also der gesamte belegte Stringinhalt geschrieben. Beim Lesen kann durch den Eingang *LENGTH* die Anzahl der zu verarbeitenden Zeichen auf den angegebenen Wert begrenzt werden.

Mit Hilfe des Einganges *SETEOT* wird festgelegt, ob das Endekennzeichen des String mit gespeichert wird oder nicht (Default: TRUE). Ist der String komplett mit Endekennzeichen im nichtflüchtigen Speicher abgelegt, kann beim Zurücklesen die Längenangabe am Eingang *LENGTH* entfallen (*LENGTH* = *0*), der Baustein übernimmt alle Zeichen bis zur Endekennung in den am Parameter *DINOUT* übergebenen String. Beim Aufruf des Bausteins mit *SETEOT* = *FALSE* wird die Speicherung des Endekennzeichens unterdrückt und somit ein Byte je String weniger Platz im nichtflüchtigen Speicher belegt. In diesem Fall muss aber beim Zurücklesen die Stringlänge bekannt sein und am Eingang *LENGTH* spezifiziert werden. Bei der Angabe der verarbeiteten Zeichen am Ausgang *SIZE* wird das Endekennzeichen mit berücksichtigt, falls es geschrieben wurde. Beim Aufruf des Bausteins mit *SETEOT* = *TRUE* ist daher der Wert des Ausganges SIZE gleich *LEN(DINOUT) + 1*.

Der Aufruf des Bausteins mit *MODE = 0* dient zur Größenbestimmung des nutzbaren nichtflüchtigen Speichers. Am Ausgang *SIZE* wird dazu die verbleibende Restgröße ab dem am Eingang *ADDR* übergebenen Wert zurück geliefert ($SIZE := NVDATA_{FullSize} - ADDR$). Um die Gesamtgröße des nichtflüchtigen Speichers zu ermitteln, ist der Baustein mit *ADDR := 0* aufzurufen.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 10 beschrieben (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *NVDATA_STR*. Dabei wird ein String zunächst ab Adresse 30 geschrieben und anschließend von der gleichen Adresse zurück gelesen. Da der Eingang *DEVICE* nicht durch das Anwenderprogramm gesetzt wird, bleibt die Standardeinstellung erhalten der Baustein benutzt somit implizit die Device-Nummer 0.

Programmbeispiel

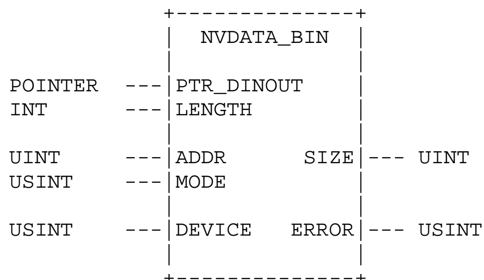
```
PROGRAM SaveDataStr

VAR CONSTANT
    NVDSTR_MODE_GET_SIZE  : USINT := 16#00;
    NVDSTR_MODE_RD_STRING : USINT := 16#08;
    NVDSTR_MODE_WR_STRING : USINT := 16#88;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR

VAR
    WriteDataString : STRING;
    WriteDataSize : UINT;
    ReadDataString : STRING;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataStr : NVDATA_STR;
END_VAR
```

```
(* write a STRING value into EEPROM *)
LD      'HelloWorld'
ST      WriteDataString

CAL     FB_NvDataStr (
        DINOUT := WriteDataString,
        LENGTH := 0,            (* save whole string            *)
        SETEOT := TRUE,         (* include termination character *)
        ADDR := 30,
        MODE := NVDSTR_MODE_WR_STRING
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a STRING value from EEPROM *)
CAL     FB_NvDataStr (
        DINOUT := ReadDataString,
        LENGTH := 0,            (* read whole string            *)
        ADDR := 30,
        MODE := NVDSTR_MODE_RD_STRING
        |
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.5 Der Funktionsbaustein NVDATA_BIN

Der Funktionsbaustein *NVDATA_BIN* dient zum Schreiben binärer Prozessdaten in den nichtflüchtigen Speicher (EEPROM, Datei) einer SPS sowie zum Rücklesen von gespeicherten Prozessdaten aus dem nichtflüchtigen Speicher.

Prototyp des Funktionsbausteines



```
                 +--------------+
                 |  NVDATA_BIN  |
                 |              |
      POINTER ---|PTR_DINOUT    |
      INT     ---|LENGTH        |
                 |              |
      UINT    ---|ADDR      SIZE|--- UINT
      USINT   ---|MODE          |
                 |              |
      USINT   ---|DEVICE   ERROR|--- USINT
                 |              |
                 +--------------+
```

Operandenbedeutung

PTR_DINOUT	Datenein- und Ausgang zum Schreiben bzw. Lesen von Daten, hier ist die Adresse eines Objektes zu übergeben, das die zu schreibenden Daten enthält bzw. die zu lesenden Daten aufnimmt
LENGTH	Begrenzung der Anzahl zu lesender bzw. zu schreibender Zeichen, bei 0 wird intern die Größe des über *PTR_TXDATA* adressierten Objektes ermittelt und als Anzahl zu lesenden bzw. zu schreibenden Zeichen verwendet

ADDR — Adresse innerhalb des nichtflüchtigen Speichers zum Lesen oder Schreiben von Daten (abhängig vom Parameter *MODE*)

MODE — Festlegung der auszuführenden Lese- oder Schreiboperation, eine Auflistung der unterstützten Modi enthält Tabelle 13.

SIZE — Dieser Ausgang gibt die Anzahl der geschriebenen bzw. gelesenen Bytes an (*MODE <> 0*) oder liefert die Größe des nutzbaren nichtflüchtigen Speichers (*MODE = 0*, siehe Text).

ERROR: — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 10 definiert (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

DEVICE — Device-Nummer, dieser Parameter ist abhängig von der jeweiligen Steuerung (Hinweis: die meisten Steuerungen unterstützen nur das Device 0, somit kann das Setzen dieses Einganges entfallen, da er bereits mit dem Initialwert 0 vorbelegt ist)

Tabelle 13: Aufruf-Modi für den Funktionsbaustein NVDATA_BIN

Mode	Aktion
16#00	Größe des nutzbaren nichtflüchtigen Speichers ermitteln (siehe Text)
16#09	Lesen von Binärdaten aus dem nichtflüchtigem Speicher, Ablage der gelesenen Daten in dem über *PTR_DINOUT* adressierten Objekt
16#89	Schreiben von Binärdaten in den nichtflüchtigen Speicher, Daten werden dem über *PTR_DINOUT* adressierten Objekt entnommen

<u>Beschreibung</u>

Mit Hilfe des Funktionsbausteines können Binärdaten in einen nichtflüchtigen Speicher (EEPROM, Datei) geschrieben bzw. ausgelesen werden. In Abhängigkeit der zu lesenden bzw. zu schreibenden Daten ist am Eingang *MODE* der entsprechende Modus entsprechend Tabelle 13 zu setzen. Abhängig vom Modus wird dabei das über den Parameter *DINOUT* adressierte Objekt als Datenquelle bzw. –ziel verwendet. Der am Eingang *ADDR* übergebene Wert legt die Basisadresse für die auszuführende Lese- oder Schreiboperation fest. Wird dabei über die maximale Speichergröße hinaus adressiert, kehrt der Funktionsbaustein mit einem entsprechenden Fehler zurück. Die Aufteilung des verfügbaren Speichers liegt vollständig in der Verantwortung des SPS-Programmes, so ist es auch die Aufgabe des SPS-Programmes sicherzustellen, dass die am Eingang *ADDR* verwendeten Werte nicht zu Überschneidungen der zu sichernden Daten führen. Am Ausgang *SIZE* wird die Anzahl der jeweils geschriebenen oder gelesenen Bytes zurückgeliefert, so dass dieser Wert zur Berechnung der nächsten freien Adresse verwendet werden kann ($ADDR_{new} := ADDR_{old} + SIZE$).

Der Eingang *LENGTH* spezifiziert die Anzahl zu verarbeitender Zeichen. Ist dieser Wert 0, wird intern die Größe des über den Parameter *DINOUT* adressierten Objektes ermittelt und als Anzahl zu lesender bzw. zu schreibenden Datenbytes verwendet.

Der Aufruf des Bausteins mit *MODE = 0* dient zur Größenbestimmung des nutzbaren nichtflüchtigen Speichers. Am Ausgang *SIZE* wird dazu die verbleibende Restgröße ab dem am Eingang *ADDR* übergebenen Wert zurück geliefert ($SIZE := NVDATA_{FullSize} - ADDR$). Um die Gesamtgröße des nichtflüchtigen Speichers zu ermitteln, ist der Baustein mit *ADDR := 0* aufzurufen.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 10 beschrieben (sie sind identisch mit den Fehlercodes des Bausteines *NVDATA_BIT*).

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *NVDATA_BIN*. Dabei wird zunächst ein Datenobjekt ab Adresse 30 geschrieben und anschließend von der gleichen Adresse zurück gelesen. Da der Eingang *DEVICE* nicht durch das Anwenderprogramm gesetzt wird, bleibt die Standardeinstellung erhalten der Baustein benutzt somit implizit die Device-Nummer 0.

Programmbeispiel

```
PROGRAM SaveDataBin

VAR CONSTANT
    NVDSTR_MODE_GET_SIZE  : USINT := 16#00;
    NVDSTR_MODE_RD_BIN    : USINT := 16#09;
    NVDSTR_MODE_WR_BIN    : USINT := 16#89;

    NVDATA_ERROR_SUCCESS        : USINT :=  0;
    NVDATA_ERROR_HW_ERROR       : USINT :=  1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT :=  2;
    NVDATA_ERROR_INVALID_MODE   : USINT :=  4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT :=  8;
    NVDATA_ERROR_PTR_TYPE       : USINT := 16;
END_VAR

VAR
    WriteDataObject : ARRAY[0..3] OF BYTE := [ 16#01, 16#02, 16#03, 16#04 ];
    ReadDataObject  : ARRAY[0..3] OF BYTE;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataBin : NVDATA_BIN;
    pDataObject  : POINTER;
END_VAR

(* write a binary data object into EEPROM *)
LD      &WriteDataObject
ST      pDataObject

CAL     FB_NvDataBin (
        PTR_DINOUT := pDataObject,
        LENGTH := 0,            (* save whole object *)
        ADDR := 30,
        MODE := NVDSTR_MODE_WR_BIN
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a binary data object from EEPROM *)
LD      &ReadDataObject
ST      pDataObject

CAL     FB_NvDataBin (
        PTR_DINOUT := pDataObject,
        LENGTH := 0,            (* read whole object *)
        MODE := NVDSTR_MODE_RD_BIN
        ADDR := 30,
        |
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

5 Zugriff auf Serielle Schnittstelle (SIO)

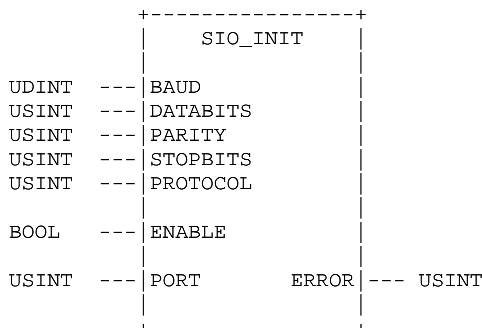
5.1 Verwendung der seriellen Schnittstelle

Die serielle Schnittstelle ermöglicht den Datenaustausch mit anderen Geräten über eine direkte Punkt-zu Punkt-Verbindung. Einsatzbeispiele hier für sind die Ausgabe von Daten auf einen Drucker oder die Ansteuerung eines Bedienterminals. Abhängig vom hardwareseitigen Ausbau der seriellen Schnittstelle kann der Datenfluss durch verschiedene Handshake-Protokolle beeinflusst werden, so z.B. hardwaremäßig über Modemsteuerleitungen (RTS, CTS, DTR, DSR) oder softwaremäßig durch XON/XOFF. Die Funktionsbausteine *SIO_INIT* und *SIO_STATE* ermöglichen Initialisierung und Steuerung der Schnittstelle sowie die Abfrage von Statusinformationen. Die Funktionsbausteine *SIO_READ_CHR* und *SIO_WRITE_CHR* dienen der Verarbeitung von Einzelzeichen, *SIO_READ_STR* und *SIO_WRITE_STR* ermöglichen im Gegensatz dazu die Übertragung von Zeichenketten.

5.2 Der Funktionsbaustein SIO_INIT

Der Funktionsbaustein *SIO_INIT* dient zum Initialisieren der seriellen Schnittstelle und zur Festlegung des Handshake-Protokolles für die Flusskontrolle.

<u>Prototyp des Funktionsbausteines</u>



<u>Operandenbedeutung</u>

BAUD	Festlegung der zu verwendenden Baudrate in Bit/s, dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Schnittstelle, gültige Werte sind z.B.: 1200, 2400, 9600, 19200, 38400, 57600, 115200
DATABITS	Festlegung der Anzahl zu verwendender Datenbits, dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Schnittstelle, gültige Werte sind z.B.: 7 = 7 Datenbits 8 = 8 Datenbits
PARITY	Festlegung der zur Sicherung der Datenübertragung zu verwendenden Parität, dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Schnittstelle, gültige Werte sind z.B.: 0 = No Parity 1 = Odd Parity 2 = Even Parity

STOPBITS — Festlegung der Anzahl zu verwendender Stopbits, dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Schnittstelle, gültige Werte sind z.B.:
1 = 1 Stopbit
2 = 2 Stopbits

PROTOCOL — Festlegung der zu verwendenden Handshake-Protokolls, dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Schnittstelle, gültige Werte sind:
0 = No Protocol
1 = XON/XOFF
2 = Hardware-Handshake (RTS/CTS Flow Control)

ENABLE — Freigeben oder Sperren der seriellen Schnittstelle
TRUE = serielle Schnittstelle initialisieren
FALSE = serielle Schnittstelle abschalten

PORT — Nummer der zu verwendenden Schnittstelle

ERROR — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 14 definiert.

Tabelle 14: Error-Codes des Funktionsbausteines SIO_INIT

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
4	Die ausgewählte Bitrate (*BAUD*) wird nicht unterstützt
8	Die ausgewählte Anzahl Datenbits (*DATABITS*) wird nicht unterstützt
16	Die ausgewählte Parität (*PARITY*) wird nicht unterstützt
32	Die ausgewählte Anzahl Stopbits wird nicht unterstützt
64	Das ausgewählte Handshake-Protokoll wird nicht unterstützt

Beschreibung

Der Funktionsbaustein initialisiert die serielle Schnittstelle mit den angegeben Parametern. Die konkrete Verfügbarkeit bzw. Unterstützung von Parametern ist abhängig von den jeweiligen Hardwareeigenschaften der Schnittstelle. Detaillierte Informationen hierzu sind dem Manual der jeweiligen Steuerung zu entnehmen. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 14 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden (z.B. 136 = 128 + 8 => ungültige Bitrate und nicht unterstütztes Protokoll).

Das folgende Programmbeispiel zeigt die Anwendung des Funktionsbausteines *SIO_INIT* zum Initialisieren der seriellen Schnittstelle mit folgenden Parametern: 9600 Baud, 8 Datenbits, keine Parität, 1 Stopbit, Software-Flusskontrolle durch XON/XOFF-Protokoll.

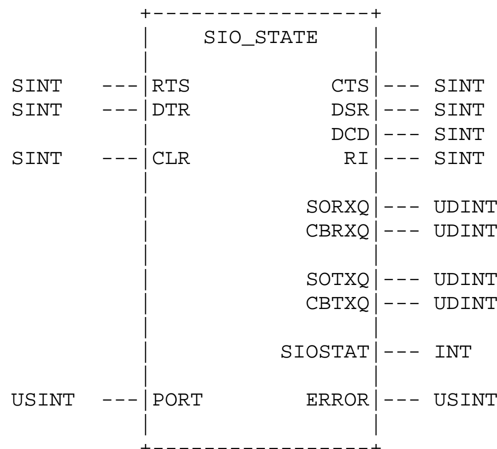
Programmbeispiel

```
VAR CONSTANT
    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT :=   0;
    SIO_INIT_PARITY_ODD             : USINT :=   1;
    SIO_INIT_PARITY_EVEN            : USINT :=   2;

    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT :=   0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT :=   1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT :=   2;

    (* Error Codes of FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT :=   0;
    SIO_INIT_ERR_HW_ERROR           : USINT :=   1;
    SIO_INIT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT :=   8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT :=  16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT :=  32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT :=  64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;

    PORTNUM : USINT := 1;
END_VAR

VAR
    FB_SioInit : SIO_INIT;
    xInitOk    : BOOL := FALSE;
END_VAR

(* ----- Init Sio ----- *)
SioInit:
(* Initialize Serial Port *)
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_XON_XOFF,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
ST      xInitOk

...

RET
```

5.3 Der Funktionsbaustein SIO_STATE

Der Funktionsbaustein *SIO_SATE* dient zum Setzen und Abfrage von Statusinformationen der seriellen Schnittstelle.

Prototyp des Funktionsbausteines



Operandenbedeutung

RTS — zu setzender Status des RTS-Signals:
-1 = aktuellen Status nicht beeinflussen
0 = Signal auf inaktiv setzen
1 = Signal auf aktiv setzen

DTR — zu setzender Status des DTR-Signals:
-1 = aktuellen Status nicht beeinflussen
0 = Signal auf inaktiv setzen
1 = Signal auf aktiv setzen

CLR — Löschen von Sende- und Empfangspuffer:
-1 = aktuellen Status nicht beeinflussen
1 = Empfangspuffer löschen
2 = Sendepuffer löschen
3 = Sende- und Empfangspuffer löschen

CTS — ermittelter Status des CTS-Signals:
-1 = Signal wird nicht unterstützt
0 = Signal ist inaktiv gesetzt
1 = Signal ist aktiv gesetzt

DSR — ermittelter Status des DSR-Signal
-1 = Signal wird nicht unterstützt
0 = Signal ist inaktiv gesetzt
1 = Signal ist aktiv gesetzt

DCD — ermittelter Status des DCD-Signals:
-1 = Signal wird nicht unterstützt
0 = Signal ist inaktiv gesetzt
1 = Signal ist aktiv gesetzt

RI	ermittelter Status des RI-Signals: -1 = Signal wird nicht unterstützt 0 = Signal ist inaktiv gesetzt 1 = Signal ist aktiv gesetzt
SORXQ	ermittelte Gesamtgröße des Empfangspuffers (Size of Rx Queue)
CBRXQ	aktuelle Anzahl von Zeichen im Empfangspuffer (Count of Bytes in Rx Queue)
SOTXQ	ermittelte Gesamtgröße des Gesamtgröße Sendepuffers (Size of Tx Queue)
CBTXQ	aktuelle Anzahl von Zeichen im Anzahl Zeichen im Sendepuffer (Count of Bytes in Tx Queue)
SIOSTAT	SIO-spezifisches Statusregister (z.B. Overrun, Frame-Error usw., siehe Manual der jeweiligen Steuerung)
PORT	Nummer der zu verwendenden Schnittstelle
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 15 definiert.

Tabelle 15: Error-Codes des Funktionsbausteines SIO_STAT

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
8	Das RTS-Signal kann bei aktiviertem Hardware-Handshake nicht beeinflusst werden (*SIO_INIT* wurde mit *PROTOCOL := 2* aufgerufen)
16	Das DTR-Signal kann bei aktiviertem Hardware-Handshake nicht beeinflusst werden (*SIO_INIT* wurde mit *PROTOCOL := 2* aufgerufen)
32	Ein direktes Setzen des RTS-Signals wird nicht unterstützt
64	Ein direktes Setzen des DTR-Signals wird nicht unterstützt
128	Das Löschen von Sende- und Empfangspuffer wird nicht unterstützt
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Setzen und Abfrage von Statusinformationen der seriellen Schnittstelle. Die konkrete Verfügbarkeit bzw. Unterstützung von Parametern ist abhängig von den jeweiligen Hardwareeigenschaften der Schnittstelle. Detaillierte Informationen hierzu sind dem Manual der jeweiligen Steuerung zu entnehmen. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 15 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden (z.B. 24 = 16 + 8 => die Signale RTS und DTR können bei aktiviertem Hardware-Handshake nicht beeinflusst werden).

Der folgende Programmausschnitt zeigt die Anwendung des Funktionsbausteines *SIO_STAT* zum ermitteln des aktuellen Status der seriellen Schnittstelle.
Hinweis: Ein vollständiges Beispielprogramm einschließlich Initialisierung und Ablaufsteuerung enthält Abschnitt 5.7.

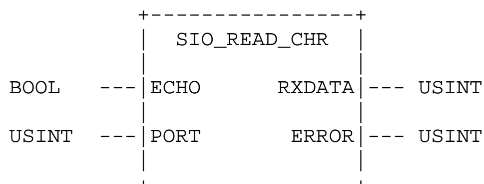
Programmbeispiel

```
VAR CONSTANT
    (* Definition of Control Codes *)
    SIO_STAT_DO_NOT_CHANGE          : SINT  :=  -1;
    SIO_STAT_CLR                    : SINT  :=   0;
    SIO_STAT_SET                    : SINT  :=   1;

    (* Error Codes of FB SIO_STAT *)
    SIO_STAT_ERR_SUCCESS            : USINT :=   0;
    SIO_STAT_ERR_HW_ERROR           : USINT :=   1;
    SIO_STAT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_STAT_ERR_RTS_SET_ERROR      : USINT :=   8;
    SIO_STAT_ERR_DTR_SET_ERROR      : USINT :=  16;
    SIO_STAT_ERR_RTS_NOT_SUPPORTED  : USINT :=  32;
    SIO_STAT_ERR_DTR_NOT_SUPPORTED  : USINT :=  64;
    SIO_STAT_ERR_CLR_NOT_SUPPORTED  : USINT := 128;
    SIO_STAT_ERR_NOT_INITIALIZED    : USINT := 255;

    PORTNUM : USINT := 1;
END_VAR

VAR
    FB_SioState : SIO_STATE;
    xStatOk     : BOOL := FALSE;

    siCts       : SINT;
    siDsr       : SINT;
    siDcd       : SINT;
    siRi        : SINT;
    udiSoRrQ    : UDINT;
    udiCbRxQ    : UDINT;
    udiSoTxQ    : UDINT;
    udiCbTxQ    : UDINT;
    iSioStat    : INT;
END_VAR

(* ----- Check Sio State ----- *)
CheckState:
(*      read current state from serial interface *)
CAL     FB_SioState (
        RTS := SIO_STAT_DO_NOT_CHANGE,
        DTR := SIO_STAT_DO_NOT_CHANGE,
        CLR := SIO_STAT_DO_NOT_CHANGE,
        PORT := PORTNUM
        |
        siCts := CTS,
        siDsr := DSR,
        siDcd := DCD,
        siRi := RI,
        udiSoRrQ := SORXQ,
        udiCbRxQ := CBRXQ,
        udiSoTxQ := SOTXQ,
        udiCbTxQ := CBTXQ,
        iSioStat := SIOSTAT)

LD      FB_SioState.ERROR
EQ      SIO_STAT_ERR_SUCCESS
ST      xStatOk

...

RET
```

5.4 Der Funktionsbaustein SIO_READ_CHR

Der Funktionsbaustein *SIO_READ_CHR* dient zum Lesen eines einzelnen Zeichens von der seriellen Schnittstelle.

Prototyp des Funktionsbausteines



Operandenbedeutung

ECHO — Zeichen-Echo ein/aus
FALSE = kein Echo
TRUE = Echo zurücksenden

RXDATA — empfangenes Zeichen (bei *ERROR := 0*)

PORT — Nummer der zu verwendenden Schnittstelle

ERROR — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 16 definiert.

Tabelle 16: Error-Codes des Funktionsbausteines SIO_READ_CHR

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
8	Es ist kein Zeichen im Empfangspuffer verfügbar
16	Ein Zeichen-Echo wird nicht unterstützt
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Lesen eines einzelnen Zeichens von der seriellen Schnittstelle. Ist bei der Rückkehr des Bausteins der Ausgang *ERROR := 8* gesetzt, war kein Zeichen im Empfangspuffer verfügbar. Bei *ERROR := 0* steht das gelesene Zeichen am Ausgang *RXDATA* zur Verfügung. Bei gesetztem Eingang *ECHO := TRUE* wird in diesem Fall das empfangene Zeichen durch den Baustein selbständig als Echo zurück gesendet. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 16 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Programmbeispiel

Der Programmausschnitt im Abschnitt 5.5 zeigt die Anwendung von *SIO_READ_CHR* in Zusammenarbeit mit dem Funktionsbaustein *SIO_WRITE_CHR*. Das Programmbeispiel ruft zunächst den Baustein *SIO_READ_CHR* um ein Zeichen von der Schnittstelle zu lesen, im Erfolgsfall wird dieses Zeichen dann mit Hilfe des Bausteins *SIO_WRITE_CHR* als Echo wieder auf dieselbe Schnittstelle zurück geschrieben.
Hinweis: Ein vollständiges Beispielprogramm einschließlich Initialisierung und Ablaufsteuerung enthält Abschnitt 5.7.

5.5 Der Funktionsbaustein SIO_WRITE_CHR

Der Funktionsbaustein *SIO_WRITE_CHR* dient zum Schreiben eines einzelnen Zeichens auf die serielle Schnittstelle.

Prototyp des Funktionsbausteines

```
            +------------------+
            |  SIO_WRITE_CHR   |
            |                  |
USINT   ---|TXDATA            |
            |                  |
BOOL    ---|EXPANDCR          |
BOOL    ---|EXPANDLF          |
            |                  |
USINT   ---|PORT        ERROR|--- USINT
            |                  |
            +------------------+
```

Operandenbedeutung

TXDATA	Eingang für das zu sendende Zeichen
EXPANDCR	automatische Erweiterung von Wagenrücklauf ein/aus FALSE: keine automatische Erweiterung von Wagenrücklauf TRUE: automatische Erweiterung von Wagenrücklauf, CR ('$R'=13) wird automatisch zu CR+LF ('RL'=13+10) erweitert
EXPANDLF	automatische Erweiterung von Zeilenumbruch ein/aus FALSE: keine automatische Erweiterung von Zeilenumbruch TRUE: automatische Erweiterung von Zeilenumbruch, LF ('$L'=10) wird automatisch zu CR+LF ('RL'=13+10) erweitert
PORT	Nummer der zu verwendenden Schnittstelle
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 17 definiert.

Tabelle 17: Error-Codes des Funktionsbausteines SIO_WRITE_CHR

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
8	Es ist kein Platz im Sendepuffer verfügbar, das Zeichen wurde ignoriert
16	Die automatische Erweiterung von Wagenrücklauf wird nicht unterstützt
32	Die automatische Erweiterung von Zeilenumbruch wird nicht unterstützt
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Schreiben eines einzelnen Zeichens auf die serielle Schnittstelle. Ist bei der Rückkehr des Bausteins der Ausgang *ERROR := 1* gesetzt, war kein Platz im Sendepuffer verfügbar, das Zeichen konnte nicht gesendet werden. Bei *ERROR := 0* wurde das Zeichen vom Eingang *TXDATA* erfolgreich auf die Schnittstelle geschrieben. Bei gesetztem Eingang *EXPANDCR := TRUE* wird geprüft, ob das gesendete Zeichen dem ASCII-Code für Wagenrücklauf entspricht. In diesem Fall Erweitert der Baustein dieses Zeichen selbständig zur Zeichenfolge Wagenrücklauf+Zeilenumbruch. Analog dazu wird bei gesetztem Eingang *EXPANDLF := TRUE* ein Zeilenumbruch durch den Baustein ebenfalls intern selbständig zur Zeichenfolge Wagenrücklauf+Zeilenumbruch ergänzt. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 17 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Der folgende Programmausschnitt zeigt die Anwendung der Funktionsbausteine *SIO_READ_CHR* (siehe Abschnitt 5.4) und *SIO_WRITE_CHR* zum Lesen und Schreiben von Zeichen über die serielle Schnittstelle. Das Programmbeispiel ruft zunächst den Baustein *SIO_READ_CHR* um ein Zeichen von der Schnittstelle zu lesen, im Erfolgsfall wird dieses Zeichen dann mit Hilfe des Bausteins *SIO_WRITE_CHR* als Echo wieder auf dieselbe Schnittstelle zurück geschrieben.
Hinweis: Ein vollständiges Beispielprogramm einschließlich Initialisierung und Ablaufsteuerung enthält Abschnitt 5.7.

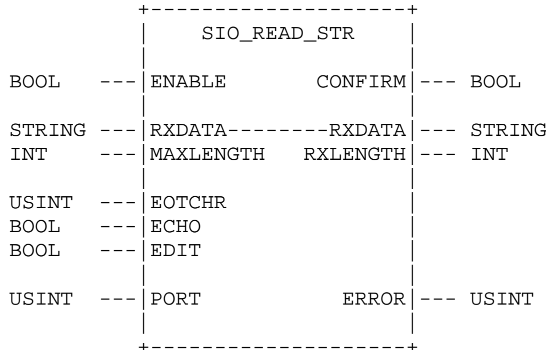
Programmbeispiel

```
VAR CONSTANT
    (* Error Codes of FB SIO_READ_CHR *)
    SIO_RCHR_ERR_SUCCESS             : USINT :=   0;
    SIO_RCHR_ERR_HW_ERROR            : USINT :=   1;
    SIO_RCHR_ERR_INVALID_PORT        : USINT :=   2;
    SIO_RCHR_ERR_NO_CHAR             : USINT :=   8;
    SIO_RCHR_ERR_ECHO_NOT_SUPPORTED : USINT :=  16;
    SIO_RCHR_ERR_NOT_INITIALIZED     : USINT := 255;

    (* Error Codes of FB SIO_WRITE_CHR *)
    SIO_WCHR_ERR_SUCCESS             : USINT :=   0;
    SIO_WCHR_ERR_HW_ERROR            : USINT :=   1;
    SIO_WCHR_ERR_INVALID_PORT        : USINT :=   2;
    SIO_WCHR_ERR_TXBUFFER_OVERFLOW   : USINT :=   8;
    SIO_WCHR_ERR_EXCR_NOT_SUPPORTED : USINT :=  16;
    SIO_WCHR_ERR_EXLF_NOT_SUPPORTED : USINT :=  32;
    SIO_WCHR_ERR_NOT_INITIALIZED     : USINT := 255;

    PORTNUM : USINT := 1;
END_VAR

VAR
    xEcho            : BOOL := FALSE;
    FB_SioReadChr    : SIO_READ_CHR;
    usiRxData        : USINT;
    xRdCharSuccess   : BOOL := FALSE;

    xExpandCR        : BOOL := FALSE;
    xExpandLF        : BOOL := FALSE;
    FB_SioWriteChr   : SIO_WRITE_CHR;
    xWrCharOk        : BOOL := FALSE;
END_VAR

(* ----- Read Char ----- *)
CAL     FB_SioReadChr (
        ECHO := xEcho,
        PORT := PORTNUM
        |
        usiRxData := RXDATA)

(*      check receive result *)
LD      FB_SioReadChr.ERROR     (* character received ? *)
EQ      SIO_RCHR_ERR_SUCCESS
ST      xRdCharSuccess
RETCN

(* ----- Write Char ----- *)
CAL     FB_SioWriteChr (
        TXDATA := usiRxData,
        EXPANDCR := xExpandCR,
        EXPANDLF := xExpandLF,
        PORT := PORTNUM)

LD      FB_SioWriteChr.ERROR
EQ      SIO_WCHR_ERR_SUCCESS
ST      xWrCharOk

...

RET
```

5.6 Der Funktionsbaustein SIO_READ_STR

Der Funktionsbaustein *SIO_READ_STR* dient zum Lesen einer Zeichenkette von der seriellen Schnittstelle.

Prototyp des Funktionsbausteines



Operandenbedeutung

RXDATA	Stringvariable zur Aufnahme der gelesenen Zeichen
MAXLENGTH	Begrenzung der Anzahl zu lesender Zeichen, bei 0 wird intern die Pufferlänge des übergebenen Strings ermittelt und als Begrenzung der Anzahl zu lesender Zeichen verwendet (Hinweis: die Standard-Puffergröße eines Strings in OpenPCS beträgt 32 Zeichen).
EOTCHR	Zeichen für Endekennung des Strings (Default: 10='$L'), z.B.: 0 (NUL), 10 ('$L'=Zeilenumbruch), 13 ('$R'=Wagenrücklauf)
ECHO	Zeichen-Echo ein/aus FALSE = kein Echo TRUE = Echo zurücksenden
EDIT	Edit-Modus ein/aus FALSE = BS (8) wird als normales Zeichen im Empfangsstring abgelegt TRUE = BS (8) wird als Korrekturzeichen interpretiert
RXLENGTH	Länge der gelesenen Zeichenfolge (bei *ERROR := 0*)
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB (siehe Text)
CONFIRM	Ausgang für Fertigmeldung durch den FB (siehe Text) FALSE = Empfang noch nicht erfolgreich abgeschlossen oder nach Fehler abgebrochen TRUE = Empfang erfolgreich abgeschlossen, *RXLENGTH* Zeichen sind im Empfangspuffer *RXDATA* verfügbar
PORT	Nummer der zu verwendenden Schnittstelle
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 18 definiert.

Tabelle 18: Error-Codes des Funktionsbausteines SIO_READ_STR

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
8	Es wurde kein Zeichen für Endekennung empfangen, Empfangsabbruch nach *MAXLENGHTH* Zeichen
16	Ein Zeichen-Echo wird nicht unterstützt
32	Der Edit-Modus wird nicht unterstützt
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Lesen einer Zeichenkette von der seriellen Schnittstelle. Die gelesenen Zeichen werden in dem am Eingang *RXDATA* übergebenen String abgelegt. Das Lesen der Zeichenkette wird beendet, wenn entweder das am Eingang *EOTCHR* definierte Zeichen für die Endekennung empfangen wurde oder wenn der String mit der als *MAXLENGTH* festgelegten Anzahl von Zeichen gefüllt ist (unter Berücksichtigung von *EDIT*; bei *MAXLENGTH := 0* wird intern die Pufferlänge des übergebenen Strings ermittelt und als Begrenzung verwendet). In beiden Fällen zeigt der Baustein bei der Rückkehr durch Setzen des Ausganges *CONFIRM* auf TRUE an, dass der Empfang beendet ist und der am Eingang *RXDATA* übergebenen String die gelesene Zeichenkette beinhaltet. Der Ausgang *RXLENGTH* gibt dabei die Anzahl der im Empfangspuffer enthaltenen Zeichen an (entspricht *LEN(RXDATA);*). Ist gleichzeitig der Ausgang *ERROR := 0* gesetzt, wurde das am Eingang *EOTCHR* definierte Zeichen für die Endekennung empfangen. Bei *ERROR := 8* wurde der Empfang nach Einlesen der als *MAXLENGTH* festgelegten Anzahl von Zeichen beendet.

Der Zeichenempfang wird durch den Baustein gestartet, wenn dieser am Eingang *ENABLE* eine steigende Flanke erkennt (erster Aufruf mit *ENABLE := TRUE*). Bis zum Abschluss des Zeichenempfangs (Endekennung oder *MAXLENGTH* Zeichen) ist der Funktionsbaustein wiederholt durch das SPS-Programm aufzurufen. Dabei muss der Eingang *ENABLE* auch weiterhin auf TRUE gesetzt bleiben, um Empfang von Zeichen zu ermöglichen. Der Baustein signalisiert seinerseits den erfolgreichen Empfangsabschluss durch Setzen des Ausganges *CONFIRM* auf TRUE. Nach der Verarbeitung der empfangenen Zeichenkette muss das SPS-Programm den Baustein zunächst mit *ENABLE := FALSE* aufrufen, um den Baustein intern wieder in seinen Ausgangszustand zurück zu versetzen. Anschließend kann ein weiterer Zeichenempfang gestartet werden, indem der Eingang *ENABLE* erneut auf TRUE gesetzt und so wiederum eine steigende Flanke erkannt wird. Ein aktiver Empfang kann jederzeit durch den Aufruf das Bausteins mit *ENABLE := FALSE* abgebrochen werden.

Bei gesetztem Eingang *ECHO := TRUE* wird jedes empfangene Zeichen durch den Baustein selbständig als Echo zurück gesendet. Bei gesetztem Eingang *EDIT := TRUE* wird das Zeichen Backspace (BS=8) nicht als normales Zeichen im Empfangspuffer abgelegt sondern als Korrekturzeichen interpretiert. Dies führt zum Löschen des zuletzt empfangenen Zeichens und der Verringerung der Ausgang *RXLENGTH* gemeldeten Anzahl bereits empfangener Zeichen.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 18 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Das folgende Programmbeispiel zeigt die Anwendung des Funktionsbausteines *SIO_READ_STR* zum Lesen einer Zeichenkette von der seriellen Schnittstelle.

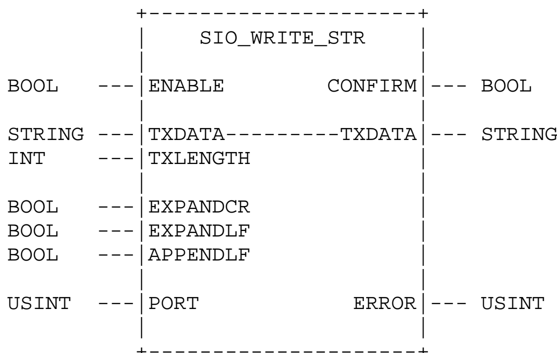
Programmbeispiel

Das Programmbeispiel im Abschnitt 5.7 zeigt die Anwendung von *SIO_READ_STR* in Zusammenarbeit mit dem Funktionsbaustein *SIO_WRITE_STR*. Das Programmbeispiel ruft zunächst den Baustein *SIO_READ_STR* um eine Zeichenkette von der Schnittstelle zu lesen. Nach vollständigem Einlesen der Zeichenkette wird diese mit Hilfe des Bausteins *SIO_WRITE_STR* wieder auf die Schnittstelle zurück geschrieben.

5.7 Der Funktionsbaustein SIO_WRITE_STR

Der Funktionsbaustein *SIO_WRITE_STR* dient zum Schreiben einer Zeichenkette auf die serielle Schnittstelle.

Prototyp des Funktionsbausteines



Operandenbedeutung

TXDATA Stringvariable mit der zu schreibenden Zeichenfolge
TXLENGTH Anzahl der zu schreibenden Zeichen, bei 0 wird intern die Länge der im String *TXDATA* enthaltenen Zeichenfolge ermittelt (entspricht *LEN(TXDATA);*) und als Anzahl der zu schreibenden Zeichen verwendet.

EXPANDCR automatische Erweiterung von Wagenrücklauf ein/aus
FALSE: keine automatische Erweiterung von Wagenrücklauf
TRUE: automatische Erweiterung von Wagenrücklauf, CR ('$R'=13) wird automatisch zu CR+LF ('RL'=13+10) erweitert

EXPANDLF automatische Erweiterung von Zeilenumbruch ein/aus
FALSE: keine automatische Erweiterung von Zeilenumbruch
TRUE: automatische Erweiterung von Zeilenumbruch, LF ('$L'=10) wird automatisch zu CR+LF ('RL'=13+10) erweitert

APPENDLF automatische Anfügen von Zeilenumbruch ein/aus
FALSE: keine automatisches Anfügen eines Zeilenumbruchs
TRUE: automatisches Anfügen eines Zeilenumbruchs, bei *EXPANDLF:=FALSE* wird LF ('$L'=10) angefügt bei *EXPANDLF:=TRUE* wird CR+LF ('RL'=13+10) angefügt

ENABLE — Eingang zur Freigabe bzw. Sperrung des FB (siehe Text)

CONFIRM — Ausgang für Fertigmeldung durch den FB (siehe Text)
FALSE = Sendung noch nicht erfolgreich abgeschlossen oder nach Fehler abgebrochen
TRUE = Sendung erfolgreich beendet

PORT — Nummer der zu verwendenden Schnittstelle

ERROR — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 19 definiert.

Tabelle 19: Error-Codes des Funktionsbausteines SIO_WRITE_STR

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
16	Die automatische Erweiterung von Wagenrücklauf wird nicht unterstützt
32	Die automatische Erweiterung von Zeilenumbruch wird nicht unterstützt
64	Das automatische Anfügen von Zeilenumbruch wird nicht unterstützt
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Schreiben einer Zeichenkette auf die serielle Schnittstelle. Der String mit der zu sendenden Zeichenfolge ist am Eingang *TXDATA* zu übergeben. Der Eingang *TXLENGTH* spezifiziert dabei die Anzahl der gültigen Zeichen. Ist dieser Wert 0, wird intern die Länge der im String *TXDATA* enthaltenen Zeichenfolge ermittelt (entspricht *LEN(TXDATA);*) und als Anzahl der zu schreibenden Zeichen verwendet. In diesem Fall wird also der gesamte belegte Stringinhalt geschrieben.

Das Schreiben der Zeichenfolge wird durch den Baustein gestartet, wenn dieser am Eingang *ENABLE* eine steigende Flanke erkennt (erster Aufruf mit *ENABLE := TRUE*). Bis zum Abschluss der Zeichenübertragung ist der Funktionsbaustein wiederholt durch das SPS-Programm aufzurufen. Dabei muss der Eingang *ENABLE* auch weiterhin auf TRUE gesetzt bleiben, um das Senden von Zeichen zu ermöglichen. Der Baustein signalisiert seinerseits den erfolgreichen Abschluss durch Setzen des Ausganges *CONFIRM* auf TRUE. Im weiteren Verlauf muss das SPS-Programm den Baustein zunächst mit *ENABLE := FALSE* aufrufen, um den Baustein intern wieder in seinen Ausgangszustand zurück zu versetzen. Anschließend kann eine weitere Zeichenübertragung gestartet werden, indem der Eingang *ENABLE* erneut auf TRUE gesetzt und so wiederum eine steigende Flanke erkannt wird. Eine aktive Sendung kann jederzeit durch den Aufruf das Bausteins mit *ENABLE := FALSE* abgebrochen werden.

Bei gesetztem Eingang *EXPANDCR := TRUE* wird jedes Zeichen mit dem ASCII-Code für Wagenrücklauf vom Baustein intern selbständig zur Zeichenfolge Wagenrücklauf+Zeilenumbruch erweitert. Analog dazu wird bei gesetztem Eingang *EXPANDLF := TRUE* ein Zeilenumbruch durch den Baustein ebenfalls intern selbständig zur Zeichenfolge Wagenrücklauf+Zeilenumbruch ergänzt. Bei gesetztem Eingang *APPENDLF := TRUE* fügt der Baustein nach dem vollständigen Senden der am Eingang *TXDATA* übergeben Zeichenfolge intern einen Zeilenumbruch an. In Abhängigkeit vom Eingang *APPENDLF* wird dieser Zeilenumbruch gegebenenfalls als Zeichenfolge bestehend aus Wagenrücklauf+Zeilenumbruch gesendet.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 19 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Das folgende Programmbeispiel zeigt die Anwendung der Funktionsbausteine *SIO_READ_STR* (siehe Abschnitt 5.6) und *SIO_WRITE_STR*. Zunächst wird der Baustein *SIO_READ_STR* aufgerufen, um eine Zeichenkette von der Schnittstelle zu lesen. Nach vollständigem Einlesen der Zeichenkette wird diese mit Hilfe des Bausteins *SIO_WRITE_STR* wieder auf die Schnittstelle zurück geschrieben. Vervollständigt wird dieses Beispielprogramm durch Initialisierung der seriellen Schnittstelle und Ablaufsteuerung der Programmausführung.

Programmbeispiel

```
PROGRAM SioRwStr
VAR CONSTANT

    (* Definition of specail ASCII-Codes *)
    strDollar       : STRING := '$$';   (* Dollar         *)
    strApostroph    : STRING := '$'';   (* Apostroph      *)
    strLF           : STRING := '$L';   (* LineFeed       *)
    strCR           : STRING := '$R';   (* CarriageReturn *)
    strNL           : STRING := '$N';   (* NewLine        *)
    strFF           : STRING := '$P';   (* NewPage        *)
    strTab          : STRING := '$T';   (* Tabulator      *)

    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT :=   0;
    SIO_INIT_PARITY_ODD             : USINT :=   1;
    SIO_INIT_PARITY_EVEN            : USINT :=   2;
```

```
    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT :=   0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT :=   1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT :=   2;

    (* Error Codes for FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT :=   0;
    SIO_INIT_ERR_HW_ERROR           : USINT :=   1;
    SIO_INIT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT :=   8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT :=  16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT :=  32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT :=  64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;


    (* Error Codes for FB SIO_READ_STR *)
    SIO_RSTR_ERR_SUCCESS            : USINT :=   0;
    SIO_RSTR_ERR_HW_ERROR           : USINT :=   1;
    SIO_RSTR_ERR_INVALID_PORT       : USINT :=   2;
    SIO_RSTR_ERR_NO_EOT_CHAR        : USINT :=   8;
    SIO_RSTR_ERR_ECHO_NOT_SUPPORTED : USINT :=  16;
    SIO_RSTR_ERR_EDIT_NOT_SUPPORTED : USINT :=  32;
    SIO_RSTR_ERR_NOT_INITIALIZED    : USINT := 255;


    (* Error Codes for FB SIO_WRITE_STR *)
    SIO_WSTR_ERR_SUCCESS            : USINT :=   0;
    SIO_WSTR_ERR_HW_ERROR           : USINT :=   1;
    SIO_WSTR_ERR_INVALID_PORT       : USINT :=   2;
    SIO_WSTR_ERR_TXBUFFER_OVERFLOW  : USINT :=   8;
    SIO_WSTR_ERR_EXCR_NOT_SUPPORTED : USINT :=  16;
    SIO_WSTR_ERR_EXLF_NOT_SUPPORTED : USINT :=  32;
    SIO_WSTR_ERR_APLF_NOT_SUPPORTED : USINT :=  64;
    SIO_WSTR_ERR_NOT_INITIALIZED    : USINT := 255;


    PORTNUM : USINT := 1;

END_VAR

VAR

    FB_SioInit       : SIO_INIT;
    xInitDone        : BOOL := FALSE;

    strRxText        : STRING(32);       (* set string length := 32 *)
    usiEotChr        : USINT := 16#0D;   (* ==> '$R' = CR *)
    xEcho            : BOOL  := TRUE;
    xEdit            : BOOL  := TRUE;
    FB_SioReadStr    : SIO_READ_STR;
    iRxDataSize      : INT;
    xRdStrConfirm    : BOOL := FALSE;
    xWaitForReceipt  : BOOL := FALSE;

    strTxText        : STRING;
    xExpandCR        : BOOL := TRUE;
    xExpandLF        : BOOL := FALSE;
    xAppendLF        : BOOL := TRUE;
    FB_SioWriteStr   : SIO_WRITE_STR;
    xWrStrConfirm    : BOOL := FALSE;
    xTransmitting    : BOOL := FALSE;

END_VAR
```

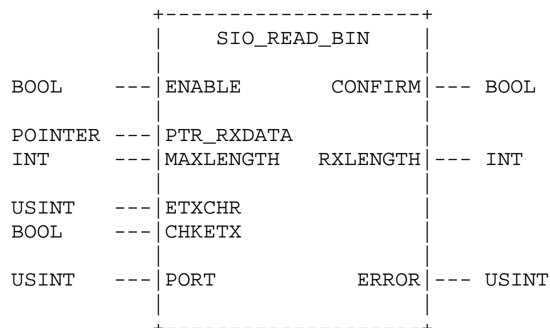
```
LD      xTransmitting
JMPC    WriteStringCont
LD      xRdStrConfirm
JMPC    WriteStringStart
LD      xWaitForReceipt
JMPC    ReadStringCont
LD      xInitDone
JMPC    ReadStringStart


(* ----- Init Sio ----- *)
SioInit:
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_NO,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
RETCN
LD      TRUE
ST      xInitDone


(* ----- Read String ----- *)
ReadStringStart:
CAL     FB_SioReadStr (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        RXDATA := strRxText,
        PORT := PORTNUM)

LD      FB_SioReadStr.ERROR
EQ      SIO_RSTR_ERR_SUCCESS
RETCN
LD      TRUE
ST      xWaitForReceipt

ReadStringCont:
CAL     FB_SioReadStr (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        RXDATA := strRxText,
        MAXLENGTH := 0,         (* no limit, use whole string length *)
        EOTCHR := usiEotChr,
        ECHO := xEcho,
        EDIT := xEdit,
        PORT := PORTNUM
        |
        xRdStrConfirm := CONFIRM,
        iRxDataSize := RXLENGTH)

LD      xRdStrConfirm
RETCN

LD      strCR
CONCAT  '-> '
CONCAT  strRxText
ST      strTxtext
```

```
(* ----- Write String ----- *)
WriteStringStart:
CAL     FB_SioWriteStr (
        ENABLE := FALSE,         (* Step 1: Reset FB *)
        TXDATA := strTxText,
        PORT := PORTNUM)

LD      FB_SioWriteStr.ERROR
EQ      SIO_WSTR_ERR_SUCCESS
RETCN
LD      TRUE
ST      xTransmitting

WriteStringCont:
CAL     FB_SioWriteStr (
        ENABLE := TRUE,          (* Step 2: Start FB *)
        TXDATA := strTxText,
        TXLENGTH := 0,           (* no limit, transmit whole string *)
        EXPANDCR := xExpandCR,
        EXPANDLF := xExpandLF,
        APPENDLF := xAppendLF,
        PORT := PORTNUM
        |
        xWrStrConfirm := CONFIRM)

LD      xWrStrConfirm
RETCN


(* ----- Reset Flow Control Logic ----- *)
LD      FALSE
ST      xTransmitting
ST      xWrStrConfirm
ST      xWaitForReceipt
ST      xRdStrConfirm
RET


END_PROGRAM
```

5.8 Der Funktionsbaustein SIO_READ_BIN

Der Funktionsbaustein *SIO_READ_BIN* dient zum Lesen einer Binärzeichnfolge von der seriellen Schnittstelle.

Prototyp des Funktionsbausteines



```
               +--------------------+
               |    SIO_READ_BIN    |
               |                    |
BOOL    ---|ENABLE       CONFIRM|--- BOOL
               |                    |
POINTER ---|PTR_RXDATA          |
INT     ---|MAXLENGTH   RXLENGTH|--- INT
               |                    |
USINT   ---|ETXCHR              |
BOOL    ---|CHKETX              |
               |                    |
USINT   ---|PORT           ERROR|--- USINT
               |                    |
               +--------------------+
```

Operandenbedeutung

PTR_RXDATA	Adresse eines Objektes, in dem die empfangenen Daten abgelegt werden
MAXLENGTH	Begrenzung der Anzahl abzulegender Bytes, bei 0 wird intern die Größe des über PTR_RXDATA adressierten Objektes ermittelt und als Begrenzung der Anzahl abzulegender Bytes verwendet (es werden max. so viele Bytes abgelegt, wie das Objekt aufnehmen kann)
EOTCHR	Zeichen für Endekennung des Binärdaten-Strings (Zeichen wird nur bei CHKETX := TRUE ausgewertet)
CHKETX	Überprüfung Zeichen für Endekennung des Binärdaten-Strings ein/aus FALSE = Zeichen für Endekennung wird nicht überprüft TRUE = Zeichen für Endekennung wird überprüft
RXLENGTH	Anzahl der abgelegten Bytes (bei *ERROR := 0*)
ENABLE	Eingang zur Freigabe bzw. Sperrung des FB (siehe Text)
CONFIRM	Ausgang für Fertigmeldung durch den FB (siehe Text) FALSE = Empfang noch nicht erfolgreich abgeschlossen oder nach Fehler abgebrochen TRUE = Empfang erfolgreich abgeschlossen, *RXLENGTH* Zeichen sind in dem über *PTR_RXDATA* adressierten Objekt verfügbar
PORT	Nummer der zu verwendenden Schnittstelle
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 20 definiert.

Tabelle 20: Error-Codes des Funktionsbausteines SIO_READ_BIN

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
8	Es wurde kein Zeichen für Endekennung empfangen, Empfangsabbruch nach *MAXLENGHTH* Zeichen
128	Pointer verweist auf ein Objekt eines nicht unterstützten Datentyps
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

Beschreibung

Der Funktionsbaustein dient zum Lesen einer Binärzeichnfolge von der seriellen Schnittstelle. Die gelesenen Zeichen werden in dem über den Eingang *PTR_RXDATA* adressierten Objekt abgelegt. Wenn der Eingang *CHKETX* auf TRUE gesetzt ist, wird die von der seriellen Schnittstelle gelesene Binärzeichnfolge auf das Auftreten des am Eingang *EOTCHR* definierten Endezeichens überwacht. Mit Erkennung des definierten Endezeichens wird das Lesen beendet und der Baustein kehrt mit TRUE am Ausgang *CONFIRM* zurück. Ist der Eingang *CHKETX* auf FALSE gesetzt oder in der gelesenen Binärzeichnfolge wird kein definiertes Endezeichen erkannt, dann beendet der Baustein das Lesen, wenn die maximale Zeichenanzahl verarbeitet wurde (entweder interne Größe des über *PTR_RXDATA* adressierten Objekts oder *MAXLENGTH* Zeichen). Auch hier kehrt der Baustein kehrt mit TRUE am Ausgang *CONFIRM* zurück. Der Ausgang *RXLENGTH* gibt die Anzahl der in dem über *PTR_RXDATA* adressierten Objekt abgelegten Zeichen an. Ist gleichzeitig der Ausgang *ERROR := 0* gesetzt, wurde das am Eingang *EOTCHR* definierte Zeichen für die Endekennung empfangen. Bei *ERROR := 8* wurde der Empfang nach Einlesen der maximalen Anzahl von Zeichen beendet.

Der Zeichenempfang wird durch den Baustein gestartet, wenn dieser am Eingang *ENABLE* eine steigende Flanke erkennt (erster Aufruf mit *ENABLE := TRUE*). Bis zum Abschluss des Zeichenempfangs (Endekennung oder *MAXLENGTH* Zeichen) ist der Funktionsbaustein wiederholt durch das SPS-Programm aufzurufen. Dabei muss der Eingang *ENABLE* auch weiterhin auf TRUE gesetzt bleiben, um Empfang von Zeichen zu ermöglichen. Der Baustein signalisiert seinerseits den erfolgreichen Empfangsabschluss durch Setzen des Ausganges *CONFIRM* auf TRUE. Nach der Verarbeitung der empfangenen Zeichenkette muss das SPS-Programm den Baustein zunächst mit *ENABLE := FALSE* aufrufen, um den Baustein intern wieder in seinen Ausgangszustand zurück zu versetzen. Anschließend kann ein weiterer Zeichenempfang gestartet werden, indem der Eingang *ENABLE* erneut auf TRUE gesetzt und so wiederum eine steigende Flanke erkannt wird. Ein aktiver Empfang kann jederzeit durch den Aufruf das Bausteins mit *ENABLE := FALSE* abgebrochen werden.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 20 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Das folgende Programmbeispiel zeigt die Anwendung des Funktionsbausteines *SIO_READ_BIN* zum Lesen einer Binärzeichenfolge von der seriellen Schnittstelle.

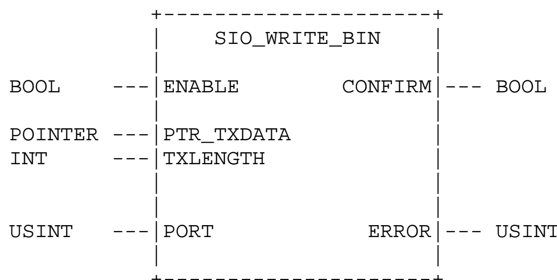
Programmbeispiel

Das Programmbeispiel im Abschnitt 5.9 zeigt die Anwendung von *SIO_READ_BIN* in Zusammenarbeit mit dem Funktionsbaustein *SIO_WRITE_BIN*. Das Programmbeispiel ruft zunächst den Baustein *SIO_READ_BIN* um eine Binärzeichenfolge von der Schnittstelle zu lesen. Nach vollständigem Einlesen der Zeichenfolge wird diese mit Hilfe des Bausteins *SIO_WRITE_BIN* wieder auf die Schnittstelle zurück geschrieben.

5.9 Der Funktionsbaustein SIO_WRITE_BIN

Der Funktionsbaustein *SIO_WRITE_STR* dient zum Schreiben einer Binärzeichenfolge auf die serielle Schnittstelle.

Prototyp des Funktionsbausteines


```
                +---------------------+
                |    SIO_WRITE_BIN    |
                |                     |
BOOL      ---|ENABLE         CONFIRM|--- BOOL
                |                     |
POINTER ---|PTR_TXDATA             |
INT       ---|TXLENGTH               |
                |                     |
                |                     |
USINT     ---|PORT             ERROR|--- USINT
                |                     |
                +---------------------+
```

Operandenbedeutung

PTR_TXDATA	Adresse eines Objektes, in dem die zu schreibenden Daten enthalten sind
TXLENGTH	Anzahl der zu schreibenden Zeichen, bei 0 wird intern die Größe des über PTR_TXDATA adressierten Objektes ermittelt und als Anzahl zu schreibender Bytes verwendet

ENABLE Eingang zur Freigabe bzw. Sperrung des FB (siehe Text)
CONFIRM Ausgang für Fertigmeldung durch den FB (siehe Text)
FALSE = Sendung noch nicht erfolgreich abgeschlossen oder nach Fehler abgebrochen
TRUE = Sendung erfolgreich beendet

PORT Nummer der zu verwendenden Schnittstelle

ERROR Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 21 definiert.

Tabelle 21: Error-Codes des Funktionsbausteines SIO_WRITE_STR

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Die ausgewählte Schnittstelle (*PORT*) wird nicht unterstützt
128	Pointer verweist auf ein Objekt eines nicht unterstützten Datentyps
255	Die ausgewählte Schnittstelle (*PORT*) ist nicht initialisiert

<u>Beschreibung</u>

Der Funktionsbaustein dient zum Schreiben einer Binärzeichenfolge auf die serielle Schnittstelle. Am Eingang *PTR_TXDATA* ist die Adresse eines Objektes anzugeben, das die zu schreibenden Daten beinhaltet. Der Eingang *TXLENGTH* spezifiziert dabei die Anzahl der gültigen Zeichen. Ist dieser Wert 0, wird intern die Größe des über *PTR_TXDATA* adressierten Objektes ermittelt und als Anzahl zu schreibender Bytes verwendet.

Das Schreiben der Zeichenfolge wird durch den Baustein gestartet, wenn dieser am Eingang *ENABLE* eine steigende Flanke erkennt (erster Aufruf mit *ENABLE := TRUE*). Bis zum Abschluss der Zeichenübertragung ist der Funktionsbaustein wiederholt durch das SPS-Programm aufzurufen. Dabei muss der Eingang *ENABLE* auch weiterhin auf TRUE gesetzt bleiben, um das Senden von Zeichen zu ermöglichen. Der Baustein signalisiert seinerseits den erfolgreichen Abschluss durch Setzen des Ausganges *CONFIRM* auf TRUE. Im weiteren Verlauf muss das SPS-Programm den Baustein zunächst mit *ENABLE := FALSE* aufrufen, um den Baustein intern wieder in seinen Ausgangszustand zurück zu versetzen. Anschließend kann eine weitere Zeichenübertragung gestartet werden, indem der Eingang *ENABLE* erneut auf TRUE gesetzt und so wiederum eine steigende Flanke erkannt wird. Eine aktive Sendung kann jederzeit durch den Aufruf das Bausteins mit *ENABLE := FALSE* abgebrochen werden.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden in Form einer Bitmaske am Ausgang *ERROR* angezeigt und sind in Tabelle 21 beschrieben. Durch gleichzeitiges Setzen verschiedener Bits können mehrere Fehler signalisiert werden.

Das folgende Programmbeispiel zeigt die Anwendung der Funktionsbausteine *SIO_READ_BIN* (siehe Abschnitt 5.8) und *SIO_WRITE_BIN*. Zunächst wird der Baustein *SIO_READ_BIN* aufgerufen, um eine Binärzeichenfolge von der Schnittstelle zu lesen. Nach vollständigem Einlesen der Zeichenfolge wird diese mit Hilfe des Bausteins *SIO_WRITE_STR* wieder auf die Schnittstelle zurück geschrieben. Vervollständigt wird dieses Beispielprogramm durch Initialisierung der seriellen Schnittstelle und Ablaufsteuerung der Programmausführung.

Programmbeispiel

```
PROGRAM SioRwBin
VAR CONSTANT

    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT :=   0;
    SIO_INIT_PARITY_ODD             : USINT :=   1;
    SIO_INIT_PARITY_EVEN            : USINT :=   2;

    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT :=   0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT :=   1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT :=   2;

    (* Error Codes for FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT :=   0;
    SIO_INIT_ERR_HW_ERROR           : USINT :=   1;
    SIO_INIT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT :=   8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT :=  16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT :=  32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT :=  64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;


    (* Error Codes for FB SIO_READ_BIN *)
    SIO_RBIN_ERR_SUCCESS            : USINT :=   0;
    SIO_RBIN_ERR_HW_ERROR           : USINT :=   1;
    SIO_RBIN_ERR_INVALID_PORT       : USINT :=   2;
    SIO_RBIN_ERR_NO_EOT_CHAR        : USINT :=   8;
    SIO_RBIN_ERR_ECHO_NOT_SUPPORTED : USINT :=  16;
    SIO_RBIN_ERR_EDIT_NOT_SUPPORTED : USINT :=  32;
    SIO_RBIN_ERR_PTR_TYPE           : USINT := 128;
    SIO_RBIN_ERR_NOT_INITIALIZED    : USINT := 255;


    (* Error Codes for FB SIO_WRITE_BIN *)
    SIO_WBIN_ERR_SUCCESS            : USINT :=   0;
    SIO_WBIN_ERR_HW_ERROR           : USINT :=   1;
    SIO_WBIN_ERR_INVALID_PORT       : USINT :=   2;
    SIO_WBIN_ERR_TXBUFFER_OVERFLOW  : USINT :=   8;
    SIO_WBIN_ERR_PTR_TYPE           : USINT := 128;
    SIO_WBIN_ERR_NOT_INITIALIZED    : USINT := 255;


    PORTNUM : USINT := 1;

END_VAR

VAR

    FB_SioInit      : SIO_INIT;
    xInitDone       : BOOL := FALSE;

    abDataBuffer    : ARRAY[0..127] OF BYTE;
    pDataObject     : POINTER;

    FB_SioReadBin   : SIO_READ_BIN;
    iRxDataSize     : INT;
    xRdBinConfirm   : BOOL := FALSE;
    xWaitForReceipt : BOOL := FALSE;
```

```
    FB_SioWriteBin  : SIO_WRITE_BIN;
    xWrBinConfirm   : BOOL := FALSE;
    xTransmitting   : BOOL := FALSE;

END_VAR

LD      xTransmitting
JMPC    WriteBinCont
LD      xRdBinConfirm
JMPC    WriteBinStart
LD      xWaitForReceipt
JMPC    ReadBinCont
LD      xInitDone
JMPC    ReadBinStart


(* ----- Init Sio ----- *)
SioInit:
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_NO,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
RETCN

LD      &abDataBuffer
ST      pDataObject

LD      TRUE
ST      xInitDone


(* ----- Read Binary Data Stream ----- *)
ReadBinStart:
CAL     FB_SioReadBin (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        PORT := PORTNUM)

LD      FB_SioReadBin.ERROR
EQ      SIO_RBIN_ERR_SUCCESS
RETCN
LD      TRUE
ST      xWaitForReceipt

ReadBinCont:
CAL     FB_SioReadBin (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        PTR_RXDATA := pDataObject,
        MAXLENGTH := 0,         (* no limit, use whole object size *)
        CHKETX := FALSE,
        PORT := PORTNUM
        |
        xRdBinConfirm := CONFIRM,
        iRxDataSize := RXLENGTH)

LD      xRdBinConfirm
RETCN
```

```
(* ----- Write Binary Data Stream ----- *)
WriteBinStart:
CAL     FB_SioWriteBin (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        PORT := PORTNUM)

LD      FB_SioWriteBin.ERROR
EQ      SIO_WBIN_ERR_SUCCESS
RETCN
LD      TRUE
ST      xTransmitting

WriteBinCont:
CAL     FB_SioWriteBin (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        PTR_TXDATA := pDataObject,
        TXLENGTH := 0,          (* no limit, transmit whole object data *)
        PORT := PORTNUM
        |
        xWrBinConfirm := CONFIRM)

LD      xWrBinConfirm
RETCN


(* ----- Reset Flow Control Logic ----- *)
LD      FALSE
ST      xTransmitting
ST      xWrBinConfirm
ST      xWaitForReceipt
ST      xRdBinConfirm
RET

END_PROGRAM
```

6 Zugriff auf Hardwarezähler

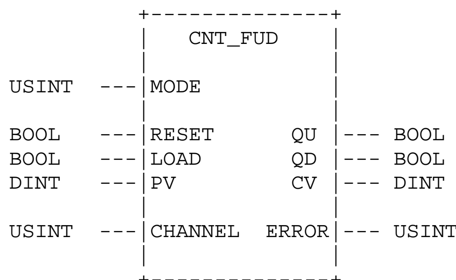
6.1 Verwendung von Hardwarezählern

Hardwarezähler ermöglichen die Erfassung von schnellen, digitalen Signalen, deren Periodendauer kleiner ist als die Zykluszeit des SPS-Programmes. Durch Hardwarezähler können somit auch schnell aufeinander folgende Signalwechsel erkannt und registriert werden. Im Gegensatz dazu erlauben Softwarezähler wie z.B. die Standard-Funktionsbausteine *CTU*, *CTD* und *CTUD* nur die Verarbeitung von Eingangssignalen, deren Änderungsgeschwindigkeit größer ist als die Zykluszeit des SPS-Programmes. Mit Hilfe des Funktionsbausteines *CNT_FUD* (Counter for Fast Up Down) können die Hardwarezähler für verschiedene Betriebsarten konfiguriert werden (Vorwärts-/Rückwärtszähler, zählen von steigenden, fallenden oder beiden Flanken usw.). Gleichzeitig dient der Baustein zum Abfragen des aktuellen Zählerstandes sowie zum Prüfen von Grenzwertüber- bzw. -unterschreitungen.

6.2 Der Funktionsbaustein CNT_FUD

Der Funktionsbaustein *CNT_FUD* dient zum Konfigurieren der Betriebsart (Zählrichtung, Zählflanke), zum Abfragen des Zählerstandes sowie zum Prüfen von Grenzwertüber- bzw. -unterschreitungen.

<u>Prototyp des Funktionsbausteines</u>



<u>Operandenbedeutung</u>

MODE

Auswahl der Betriebsart für den selektierten Kanal, der Wertebereich hängt von den durch die Hardware unterstützten Betriebsarten ab

0 = Sperrung des ausgewählten Kanal, die Ausgänge werden zurückgesetzt

Vorwärts-/Rückwärtszähler, softwaregesteuert:
1 = Vorwärtszähler, steigende Flanke
2 = Vorwärtszähler, fallende Flanke
3 = Vorwärtszähler, beide Flanken

4 = Rückwärtszähler, steigende Flanke
5 = Rückwärtszähler, fallende Flanke
6 = Rückwärtszähler, beide Flanken

Vorwärts-/Rückwärtszähler, hardwaregesteuert:
7 = steigende Flanke
8 = fallende Flanke
9 = beide Flanken
Die Richtungssteuerung erfolgt durch den zugehörigen digitalen Steuereingang:
Steuereingang = 0 → Vorwärtszähler
Steuereingang = 1 → Rückwärtszähler

Vorwärts-/Rückwärtszähler, hardwaregesteuert:
10 = steigende Flanke
11 = fallende Flanke
12 = beide Flanken
Die Richtungssteuerung erfolgt durch den zugehörigen digitalen Steuereingang:
Steuereingang = 0 → Rückwärtszähler
Steuereingang = 1 → Vorwärtszähler

RESET Der Eingangswert TRUE führt zum Rücksetzen des internen Zählerstandes auf Null. Die Eingänge LOAD und PV haben dabei keinen Einfluss. Solange der Eingang den Wert TRUE beibehält, werden keine Zählimpulse verarbeitet. Mit der fallenden Flanke wechselt der Baustein in die am Eingang MODE gewählte Betriebsart.

LOAD Der Eingangswert TRUE führt zur Übernahme des am Eingang PV angegebenen Anfangswertes in den Zähler

PV Der am Eingang angegebene Wert wird mit LOAD = TRUE in den Zähler übernommen, durch die daraus resultierende Gleichheit von aktuellem Zählerstand und PV wird der Ausgang QU auf TRUE gesetzt

CHANNEL Kanalnummer des Zählers

QU TRUE: Der erreichte Zählerstand ist größer oder gleich PV

QD TRUE: Der erreichte Zählerstand ist kleiner oder gleich Null

CV aktueller Zählerstand

ERROR Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind Tabelle 22 definiert.

Tabelle 22: Error-Codes des Funktionsbausteines CNT_FUD

Error-Code	Bedeutung
0	Es ist kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Der ausgewählte Kanal (*CHANNEL*) wird nicht unterstützt
4	Die ausgewählte Betriebsart (*MODE*) wird nicht unterstützt

Beschreibung

Der Funktionsbaustein dient zum Konfigurieren der Betriebsart (Zählrichtung, Zählflanke, Hard- oder Softwaresteuerung), zum Abfragen des Zählerstandes sowie zum Prüfen von Grenzwertüber- bzw. -unterschreitungen. Die nutzbaren Betriebsarten des Bausteines sind abhängig von der Unterstützung durch die konkret verwendete Hardware. Detaillierte Informationen hierzu sind dem Manual der jeweiligen Steuerung zu entnehmen.

Die Auswahl der jeweiligen Betriebsart erfolgt mit Hilfe des Einganges *MODE*. Hierbei werden die Zählrichtung (Vorwärts-/Rückwärtszähler) und die zu verarbeitenden Zählflanken (steigende, fallende oder beiden Flanken) konfiguriert. In Abhängigkeit der verwendeten Hardware besteht die Möglichkeit, für die Umschaltung der Zählrichtung einen weiteren digitalen Eingang zu benutzen, der dann als Steuereingang für den Zähler fungiert.

Die Eingänge *LOAD* und *RESET* ermöglicht das Setzen des Zählers auf einen beliebigen Startwert bzw. das Löschen des aktuellen Zählerstandes. Beim Aufruf des Funktionsbausteines mit auf TRUE gesetztem Eingang *LOAD* wird der am Eingang *PV* angegebene Startwert als neuer Zählerwert *CV* in den internen Zähler übernommen. Der Aufruf des Bausteines mit auf TRUE gesetztem Eingang *RESET* führt zum Rücksetzen des internen Zählerstandes auf Null. Der Zustand der Eingänge *LOAD* und *PV* werden dabei ignoriert. Solange der Eingang *RESET* den Wert TRUE beibehält, werden keine Zählimpulse verarbeitet. Mit der fallenden Flanke wechselt der Baustein in die am Eingang *MODE* gewählte Betriebsart.

Am Ausgang *CV* des Funktionsbausteines liefert den aktuellen Zählerstand. Ein gesetzter Ausgang *QU := TRUE* zeigt an, dass der erreichte Zählerstand größer oder gleich PV ist (Überlauf), analog dazu zeigt ein gesetzter Ausgang *QD := TRUE* zeigt an, dass der erreichte Zählerstand kleiner oder gleich Null ist. Befindet sich der aktuelle Zählerwert *CV* im Intervall $0 < CV < PV$, so sind beide Ausgänge *QU* und *QV* inaktiv.

Jedem Zählerkanal ist ein spezifischer digitaler Zählereingang zugeordnet (Informationen hierzu sind dem Manual der jeweiligen Steuerung zu entnehmen). Abhängig vom konfigurierten Betriebsmodus (Eingang *MODE*) erfolgt die Auswahl der Zählrichtung entweder implizit, fest vorgegeben durch den ausgewählten Modus (softwaregesteuert, *MODE := 1...6*) oder flexibel zur Laufzeit über zweiten digitalen Steuereingang (hardwaregesteuert, *MODE := 7...12*). Dies ist bei der Verwendung der digitalen Eingänge zu berücksichtigen. Unabhängig von der gewählten Betriebsart des Zählers sind der Momentanwerte des digitalen Zählereinganges sowie ggf. des Steuereinganges stets auch im Prozessabbild der digitalen Eingänge hinterlegt.

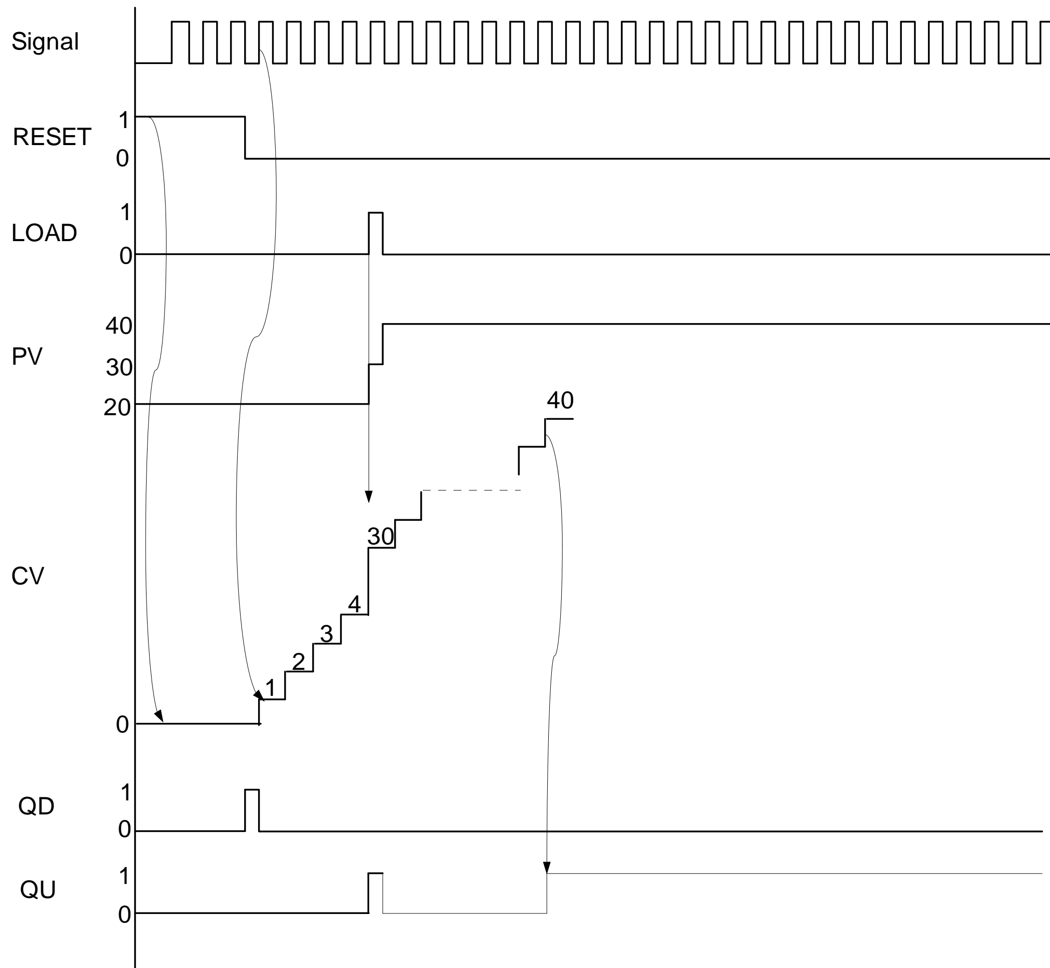


Bild 2: Signalverlauf der Ausgänge am Beispiel eines Vorwärtszählers

Bild 2 verdeutlicht die Verläufe der einzelnen Signale am Beispiel eines Vorwärtszählers (*MODE := 1*, zählen steigender Flanke). Bei aktivem *RESET := TRUE* bleiben die Eingänge *LOAD* und *PV* unberücksichtigt, die Ausgänge sind inaktiv. Mit der fallenden Flanke am Eingang *RESET* wird die gewählte Betriebsart eingestellt und der Zählerwert *CV* auf Null zurückgesetzt. Bei aktivem *LOAD := TRUE* wird der Wert am Eingang *PV* als Anfangswert nach *CV* übernommen. Der Ausgang *QD* wechselt nach TRUE, sobald der Zählerstand kleiner oder gleich Null ist; der Ausgang *QU* wechselt nach TRUE, sobald der Zählerstand größer oder gleich *PV* ist.

Programmbeispiel

```
PROGRAM CntDemo

VAR CONSTANT

    (* Error Codes of FB CNT_FUD *)
    CNT_FUD_ERROR_SUCCESS          : USINT := 0;
    CNT_FUD_ERROR_HW_ERROR         : USINT := 1;
    CNT_FUD_ERROR_UNKNOWN_CHANNEL : USINT := 2;
    CNT_FUD_ERROR_INVALID_MODE     : USINT := 4;

END_VAR
```

```
VAR
    xOvflUp      : BOOL;
    xOvflDwn     : BOOL;

    usiProcState : USINT := 0;
    ausiError    : ARRAY[0..3] OF USINT;

    FB_CntFUD   : CNT_FUD;
END_VAR

(* ----- Select current program step ----- *)
LD      usiProcState
EQ      0
JMPC    CounterInit
LD      usiProcState
EQ      1
JMPC    CounterRead
LD      0
ST      usiProcState


(* ----- Init Counter ----- *)
CounterInit:
CAL     CntFUD (              (* Reset Counter *)
        CHANNEL := 0,
        RESET := TRUE
        |
        ausiError[0] := ERROR)

CAL     CntFUD (             (* Set Mode and StartValue *)
        MODE := 1,
        RESET := FALSE,
        LOAD := TRUE,
        PV := 30
        |
        ausiError[1] := ERROR)

CAL     CntFUD (             (* Clear Input LOAD to start Counter *)
        LOAD := FALSE
        |
        ausiError[2] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Read Counter Value ----- *)
CounterRead:
CAL     CntFUD (
        PV := 40
        |
        xOvflUp := QU,
        xOvflDwn := QD,
        ausiError[3] := ERROR)


(* ----- Cycle End ----- *)
ProgExit:
RET

RET
```

7 Zugriff auf Echtzeituhr (RTC)

7.1 Anwendung der Echtzeituhr (RTC)

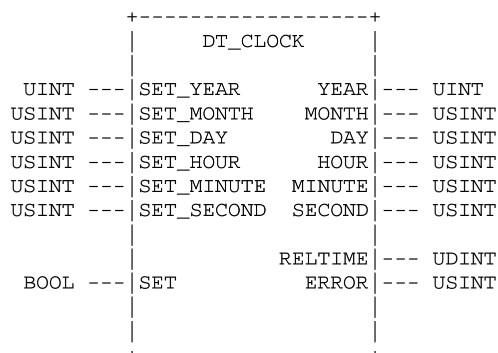
Die Echtzeituhr (**RTC** = **R**eal **T**ime **C**lock) ist ein spezieller Hardwarebaustein, der über einer Batterie mit Spannung versorgt, um auch im ausgeschaltenen Zustand der SPS arbeiten zu können. Es verfügen jedoch nur einige wenige Steuerungsbaugruppen über einen derartigen Baustein. Die Echtzeituhr stellt einem SPS-Programm die absolute Uhrzeit sowie das aktuelle Datum zur Verfügung. Diese Informationen können beispielsweise zur Steuerung datums- und uhrzeitabhängiger Prozesse sowie zur Protokollierung von Ereignissen mit Zeitstempel benutzt werden.

Der Funktionsbaustein *DT_CLOCK* ermöglicht das Stellen der Echtzeituhr sowie die Abfrage von Datum und Uhrzeit (siehe Abschnitt 7.2). An seinen Ausgängen stehen Datum und Uhrzeit sowohl in absoluter Form (Jahr/Monat/Tag, Stunde/Minute/Sekunde) als auch in relativer Form (Sekunden seit dem 01.01.1980) zur Verfügung. Der Funktionsbaustein *DT_ABS_TO_REL* dient zur Umrechnung einer absoluten Zeit- und Datumsangabe in die entsprechende relative Darstellung (siehe Abschnitt 7.3). In dieser Form vereinfachen sich arithmetische Operationen wie z.B. die Berechnung von Zeitdifferenzen oder Festlegung einer neuen Schaltzeit auf die einfache Subtraktion bzw. Addition von ganzzahligen UDINT-Variablen. Der Funktionsbaustein *DT_REL_TO_ABS* konvertiert bei Bedarf das Berechnungsergebnis anschließend wieder aus der relativen Darstellung zurück in seine absolute Form (siehe Abschnitt 7.4).

7.2 Der Funktionsbaustein DT_CLOCK

Der Funktionsbaustein *DT_CLOCK* dient zum Stellen der Echtzeituhr sowie zum Auslesen von Datum und Uhrzeit aus der Echtzeituhr der SPS. Dieser Baustein ist nur auf Steuerungen verfügbar, die mit einem RTC-Baustein ausgestattet sind.

<u>Prototyp des Funktionsbausteines</u>



<u>Operandenbedeutung</u>

SET_YEAR
SET_MONTH
SET_DAY — Jahr/Monat/Tag des zu setzenden Datums

SET_HOUR
SET_MINUTE
SET_SECOND — Stunde/Minute/Sekunde der zu setzenden Uhrzeit

SET	TRUE: Das an den Eingängen *SET_YEAR*, *SET_MONTH* und *SET_DAY* anliegende Datum sowie die an den Eingängen *SET_HOUR*, *SET_MINUTE* und *SET_SECOND* anliegende Uhrzeit werden beim Aufruf des Bausteins in die RTC geschrieben. Gleichzeitig können an den entsprechenden Ausgängen das gesetzte Datum sowie die gesetzte Uhrzeit zurück gelesen werden. FALSE: Beim Aufruf des Bausteins werden nur die aktuelle Uhrzeit sowie das aktuelle Datum aus der RTC gelesen, die RTC wird jedoch nicht neu gestellt.
YEAR MONTH DAY	Jahr/Monat/Tag des gelesenen Datums
HOUR MINUTE SECOND	Stunde/Minute/Sekunde der gelesenen Uhrzeit
RELTIME	relativer Form des gelesenen Datums sowie der gelesenen Uhrzeit (Sekunden seit dem 01.01.1980)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 23 definiert.

Tabelle 23: Error-Codes der Funktionsbausteine DT_Xxx

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
4	Ungültigen Modus (*MODE*) beim Aufruf des Funktionsbausteines
8	Power-Fail, gelesene Zeit ungültig (siehe Text)
16	Übergebene absolute Zeitangabe ungültig

Beschreibung

Beim Aufruf des Funktionsbausteines mit auf TRUE gesetztem Eingang *SET* werden das an den entsprechenden Eingängen anliegende Datum (*SET_YEAR*, *SET_MONTH* und *SET_DAY*) sowie die anliegende Uhrzeit (*SET_HOUR*, *SET_MINUTE* und *SET_SECOND*) in die RTC der SPS übernommen. Gleichzeitig können an den entsprechenden Ausgängen das gesetzte Datum sowie die gesetzte Uhrzeit zurück gelesen werden. Ein Aufruf des Funktionsbausteines mit auf FASLE gesetztem Eingang *SET* führt dagegen nur zum Auslesen des aktuellen Datums sowie der aktuellen Uhrzeit, ohne jedoch die RTC dabei zu beeinflussen. Die Werte der Setzeingänge werden dabei ignoriert. Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 23 beschrieben.

Ist nach Ausführung des Funktionsbausteines *DT_CLOCK* der Ausgang *ERROR* = 3, war die Stromversorgung der RTC unterbrochen (Power-Fail, Batterie leer) und die gelesene Zeit ist ungültig. Dieser Fehlerzustand bleibt erhalten, bis die RTC der SPS neu gestellt wird (Aufruf des Funktionsbausteines mit Eingang *SET := TRUE*) oder die SPS per Reset-Taster erneut zurückgesetzt wird.

Das folgende Programmbeispiel zeigt die Anwendung des Funktionsbausteines *DT_CLOCK* sowohl zum Stellen als auch zum Lesen der Echtzeituhr.

Programmbeispiel

```
PROGRAM RtcTest

VAR
        Year : UINT;
        Month : USINT;
        Day : USINT;
        Hour : USINT;
        Minute : USINT;
        Second : USINT;
        RelTime : UDINT;

        ErrorCode : ARRAY [0..1] OF USINT;

        FB_DtClock : DT_CLOCK;
END_VAR

LD  0
ST      ErrorCode[0]
ST      ErrorCode[1]

(* setup RTC with new time/date *)
CAL     FB_DtClock (
        SET_YEAR := 2003,
        SET_MONTH := 8,
        SET_DAY := 6,
        SET_HOUR := 12,
        SET_MINUTE := 3,
        SET_SECOND := 0,
        SET := TRUE
        |
        Error[0] := ERROR)

(* read absolute and relative time from RTC *)
CAL     FB_DtClock (SET :=FALSE)

LD      FB_DtClock.YEAR
ST      Year
LD      FB_DtClock.MONTH
ST      Month
LD      FB_DtClock.DAY
ST      Day
LD      FB_DtClock.HOUR
ST      Hour
LD      FB_DtClock.MINUTE
ST      Minute
LD      FB_DtClock.SECOND
ST      Second
LD      FB_DtClock.RELTIME
ST      RelTime
LD      FB_DtClock.ERROR
ST      ErrorCode[1]

RET

END_PROGRAM
```

7.3 Der Funktionsbaustein DT_ABS_TO_REL

Der Funktionsbaustein *DT_ABS_TO_REL* dient zur Umrechnung einer absoluten Zeit- und Datumsangabe in die entsprechende relative Darstellung (Sekunden seit dem 01.01.1980). In dieser Form vereinfachen sich arithmetische Operationen wie z.B. die Berechnung von Zeitdifferenzen oder Festlegung einer neuen Schaltzeit auf die einfache Subtraktion bzw. Addition von ganzzahligen UDINT-Variablen.

Prototyp des Funktionsbausteines

```
             +-----------------+
             |  DT_ABS_TO_REL  |
             |                 |
     UINT ---|YEAR             |
    USINT ---|MONTH            |
    USINT ---|DAY              |
    USINT ---|HOUR             |
    USINT ---|MINUTE           |
    USINT ---|SECOND           |
             |                 |
             |          RELTIME|--- UDINT
             |            ERROR|--- USINT
             |                 |
             +-----------------+
```

Operandenbedeutung

YEAR MONTH DAY	Jahr/Monat/Tag des zu wandelnden Datums
HOUR MINUTE SECOND	Stunde/Minute/Sekunde der zu wandelnden Uhrzeit
RELTIME	relativer Form des gewandelten Datums sowie der gewandelten Uhrzeit (Sekunden seit dem 01.01.1980)
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 23 definiert.

Beschreibung

Beim Aufruf des Funktionsbausteines werden das an den entsprechenden Eingängen anliegende Datum (*YEAR*, *MONTH* und *DAY*) sowie die anliegende Uhrzeit (*HOUR*, *MINUTE* und *SECOND*) aus der absoluten Zeit- und Datumsangabe in die entsprechende relative Darstellung (Sekunden seit dem 01.01.1980) umgerechnet. In dieser Form vereinfachen sich arithmetische Operationen wie z.B. die Berechnung von Zeitdifferenzen oder Festlegung einer neuen Schaltzeit auf die einfache Subtraktion bzw. Addition von ganzzahligen UDINT-Variablen. Der Funktionsbaustein *DT_REL_TO_ABS* konvertiert bei Bedarf das Berechnungsergebnis anschließend wieder aus der relativen Darstellung zurück in seine absolute Form (siehe Abschnitt 7.4). Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 23 beschrieben.

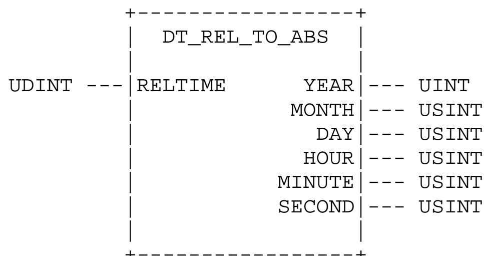
Programmbeispiel

```
PROGRAM DtConv1

VAR
        RelTime : UDINT;
        ErrorCode : USINT;

        FB_DtAbsToRel : DT_ABS_TO_REL;
END_VAR

CAL     FB_DtAbsToRel(
        YEAR := 2003,
        MONTH := 7,
        DAY := 23,
        HOUR := 15,
        MINUTE := 10,
        SECOND := 20
        |
        Error := ERROR)

LD      FB_DtAbsToRel.RELTIME
ST      RelTime
RET

END_PROGRAM
```

7.4 Der Funktionsbaustein DT_REL_TO_ABS

Der Funktionsbaustein *DT_REL_TO_ABS* dient zur Umrechnung einer relativen Zeitangabe (Sekunden seit dem 01.01.1980) in die entsprechende absolute Form (Jahr/Monat/Tag, Stunde/Minute/Sekunde)

Prototyp des Funktionsbausteines



```
             +----------------+
             |  DT_REL_TO_ABS |
             |                |
    UDINT ---|RELTIME     YEAR|--- UINT
             |           MONTH|--- USINT
             |             DAY|--- USINT
             |            HOUR|--- USINT
             |          MINUTE|--- USINT
             |          SECOND|--- USINT
             |                |
             +----------------+
```

Bedeutung der Operanden

RELTIME	relativer Form des zu wandelten Datums sowie der zu wandelten Uhrzeit (Sekunden seit dem 01.01.1980)
YEAR MONTH DAY	Jahr/Monat/Tag des gewandelten Datums
HOUR MINUTE SECOND	Stunde/Minute/Sekunde der gewandelten Uhrzeit

Beschreibung

Beim Aufruf des Funktionsbausteines wird die am Eingang *RELTIME* anliegende relative Zeit (Sekunden seit dem 01.01.1980) in die zugehörige absolute Darstellungsform umgerechnet und an den entsprechenden Ausgängen für Datum (*YEAR*, *MONTH* und *DAY*) und Uhrzeit (*HOUR*, *MINUTE* und *SECOND*) zur Verfügung gestellt. Zur Berechnung einer relativen Zeitangabe aus der absoluten Form steht der Baustein *DT_ABS_TO_REL* zur Verfügung (siehe Abschnitt 7.3).

Programmbeispiel

```
PROGRAM DtConv2

VAR
        Year : UINT;
        Month : USINT;
        Day : USINT;
        Hour : USINT;
        Minute : USINT;
        Second : USINT;

        RelTime : UDINT := 743440220;     (* = 23.07.2003, 15:10:20 *)

        FB_DtRelToAbs : DT_REL_TO_ABS;
END_VAR

CAL     FB_DtRelToAbs (RELTIME := RelTime)

LD      FB_DTRelToAbs.YEAR
ST      Year
LD      FB_DTRelToAbs.MONTH
ST      Month
LD      FB_DTRelToAbs.DAY
ST      Day
LD      FB_DTRelToAbs.HOUR
ST      Hour
LD      FB_DTRelToAbs.MINUTE
ST      Minute
LD      FB_DTRelToAbs.SECOND
ST      Second

RET

END_PROGRAM
```

8 Zugriff auf Impulsgenerator (PWM/PTO)

8.1 Anwendung des Impulsgenerators (PTO/PWM)

Der Impulsgenerator (**PTO** = **P**ulse **T**imer **O**utput / **PWM** = **P**ulse **W**idth **M**odulation) ermöglicht sowohl die Generierung einmaliger Impulsfolgen (PTO-Modus) als auch kontinuierlicher Impulsfolgen (PWM-Modus). Einsatzbeispiele sind die verlustarme Leistungsregelung ohmscher Lasten wie Heizstäbe oder Lampen (PWM-Modus) sowie die Ansteuerung von Schrittmotoren mit Einzelimpulsfolgen (PTO-Modus). Der Funktionsbaustein *PTO_PWM* erlaubt die Benutzung des Impulsgenerators sowohl im PTO- als auch im PWM-Modus mit direkter Parametrierung des Generators. Der Funktionsbaustein *PTO_TAB* ermöglicht die Definition komplexer Einzelimpuls-Folgen in Form einer Parameter-Tabelle, wie sie beispielsweise zur Realisierung von Rampenfunktionen bei der Ansteuerung von Schrittmotoren benötigt werden.

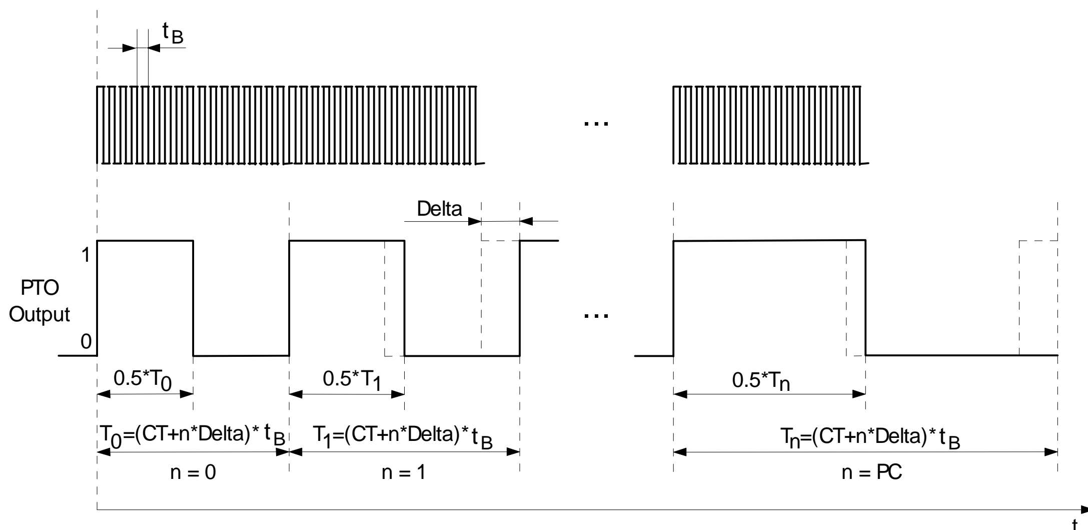


Bild 3: Zeitverhalten des Impulsgenerators im PTO-Modus

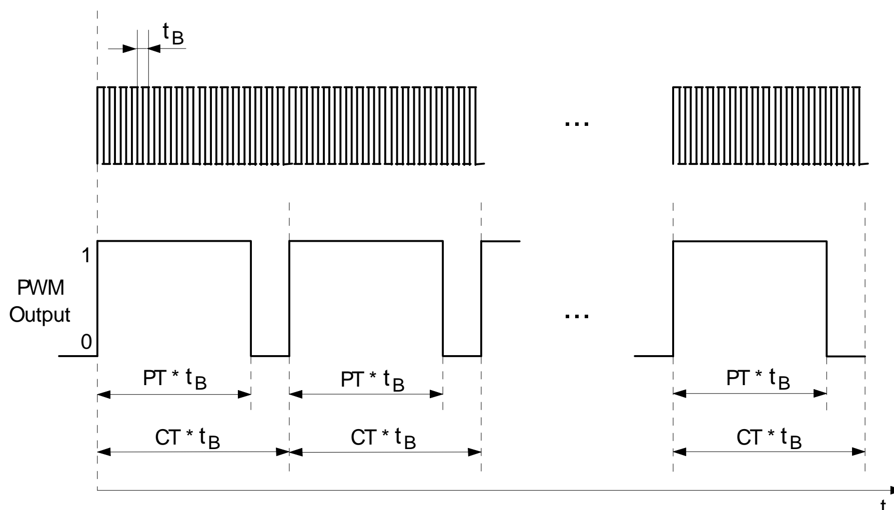


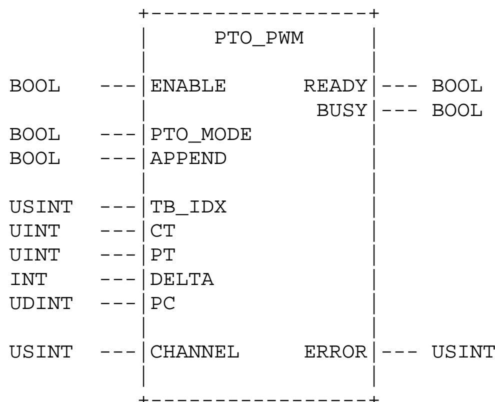
Bild 4: Zeitverhalten des Impulsgenerators im PWM-Modus

Bild 3 verdeutlicht das Zeitverhalten des Impulsgenerators in der Betriebsart PTO, Bild 4 zeigt das Zeitverhalten des Impulsgenerators in der Betriebsart PWM.

8.2 Der Funktionsbaustein PTO_PWM

Der Funktionsbaustein *PTO_PWM* dient zum direkten Parametrieren des Impulsgenerators sowohl im PTO-Modus (Impulsfolgeausgang) als auch im PWM-Modus (Impulsdauerausgang). Dieser Baustein ist nur auf Steuerungen verfügbar, die über PWM-Ausgänge verfügen.

Prototyp des Funktionsbausteines



```
          +-----------------+
          |     PTO_PWM     |
          |                 |
BOOL   ---|ENABLE      READY|--- BOOL
          |             BUSY|--- BOOL
BOOL   ---|PTO_MODE         |
BOOL   ---|APPEND           |
          |                 |
USINT  ---|TB_IDX           |
UINT   ---|CT               |
UINT   ---|PT               |
INT    ---|DELTA            |
UDINT  ---|PC               |
          |                 |
USINT  ---|CHANNEL     ERROR|--- USINT
          |                 |
          +-----------------+
```

Operandenbedeutung

PTO_MODE	Auswahl der Betriebsart TRUE = PTO-Generator (Impulszählerausgang, einmalige Impulsfolge) FALSE = PWM-Generator (Impulsdauerausgang, kontinuierliche Impulsfolge) Eine Änderung des Mode-Einganges bei Eingang *ENABLE := TRUE* führt zum Beenden der vorher eingestellten Funktion
APPEND	Steuereingang zum Anfügen eines Parametersatzes TRUE = die aktuell konfigurierten Parameter werden als weiterer Parametersatz übernommen FALSE = nur Aktualisierung der Status-Ausgänge des Bausteins, die konfigurierten Parameter werden ignoriert
TB_IDX	Index zur Festlegung des Basistaktes für den Impulsgenerator dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Steuerung, gültige Werte sind z.B.: 0 = 800ns Basistakt 1 = 1ms Basistakt Der Basistakt wird nur mit der steigenden Flanke am Eingang *ENABLE* übernommen.
CT	PTO-Modus: Periodendauer PWM-Modus: Zykluszeit die Periodendauer bzw. die Zykluszeit sind abhängig von dem am Eingang *TB_IDX* festgelegten Basistakt TB_IDX := 0 : 125 ... 65535 (100µs - 52428 µs) TB_IDX := 1 : 2 ... 65535 (2ms - 65535ms)

PT	PTO-Modus: keine Verwendung PWM-Modus: Impulsdauer, Wertebereich: 0 .. 65535
DELTA	PTO-Modus: Änderung der Periodendauer zwischen zwei Impulsen, Wertebereich: -32768 ... +32767 PWM-Modus: keine Verwendung
PC	PTO-Modus: Anzahl der Impulse, Wertebereich: 1 ... 4294967295 PWM-Modus: keine Verwendung
ENABLE	Freigeben oder Sperren des Impulsgenerators TRUE = Aktivieren des Impulsgenerators, der Generator übernimmt die Steuerung des zugeordneten digitalen Ausganges FALSE = Deaktivieren des Impulsgenerators, die Steuerung des zugeordneten digitalen Ausganges erfolgt durch das Prozessabbild (direktes Beeinflussen des Ausganges durch das SPS-Programm) Mit der steigenden Flanke am Eingang *ENABLE* übernimmt der Funktionsbaustein den Index zur Festlegung des Basistaktes (Eingang *TB_IDX*).
READY	Status-Ausgang des Impulsgenerators TRUE = der Impulsgenerator wurde vollständig parametriert, der Generator ist betriebsbereit FALSE = der Impulsgenerator wurde noch nicht parametriert oder der Baustein wurde mit einem Fehler beendet, der Generator ist nicht betriebsbereit
BUSY	Status-Ausgang des Impulsgenerators TRUE = der Impulsgenerator ist aktiv (Impulsfolge wird generiert), die Steuerung des digitalen Ausganges erfolgt durch den Generator FALSE = der Impulsgenerator ist inaktiv (Impulsfolge abgeschlossen), die Steuerung des digitalen Ausganges erfolgt durch das Prozessabbild (direktes Beeinflussen des Ausganges durch das SPS-Programm)
CHANNEL	Nummer des zu verwendenden Kanals
ERROR	Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 24 definiert.

Tabelle 24: Error-Codes des Funktionsbausteines PTO_PWM

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Der ausgewählte Kanal (*CHANNEL*) wird nicht unterstützt
8	Der ausgewählte Index für den Basistakt (*TB_IDX*) wird nicht unterstützt
16	Bereichsüberschreitung bei der Neuberechnung der Periodendauer unter Einbeziehung von *DELTA* (Periodendauer ist größer 65535 oder kleiner 0)
32	Es ist kein Platz im Datensatzpuffer verfügbar, der Datensatz wurde ignoriert

Beschreibung

Der Funktionsbaustein ermöglicht ein direktes Parametrieren des Impulsgenerators im PTO-Modus (Impulsfolgeausgang) und im PWM-Modus (Impulsdauerausgang). Die Betriebsart Impulsausgang ist eine Alternativfunktion der digitalen Ausgänge. Ist der Eingang *ENABLE := FALSE*, so wird der zugehörige digitale Ausgang über das Prozessabbild beeinflusst. Bei *ENABLE := TRUE* erfolgt dagegen die Steuerung des Ausganges durch den Impulsgenerator.

PTO-Generator (PDO_MODE := TRUE, Impulszählerausgang, einmalige Impulsfolge):
Der PTO-Generator erzeugt eine einmalige Impulsfolge und steuert damit den digitalen Ausgang. Die Impulsfolge wird durch einen Parametersatz beschrieben, der aus Periodendauer (Anfangswert), Delta der Periodendauer (Wert der Änderung zwischen zwei aufeinander folgenden Impulsen) sowie der Anzahl zu generierender Impulse besteht. Die Impulsbreite wird fest auf 50% der Periodendauer eingestellt (Tastverhältnis 1:1). Die Periodendauer T_n berechnet sich unter Einbeziehung von DELTA wie folgt (vergl. auch Bild 3):

$T_n = (CT + n*DELTA) * t_B$ (mit 0 <= n <= PC)

Bei t_B = 1 ms (*TB_IDX := 1*) und *CT := 1000* ergibt sich eine Anfangs-Periodendauer (n = 0) von:
T_n = 1 ms * 1000 = 1 Sekunde

Der Funktionsbaustein unterstützt das Aneinanderreihen von Impulsfolgen mit unterschiedlichen Werte für die Periodendauer, Delta (Änderung der Periodendauer) und Anzahl der Impulse. Dazu ist der Baustein für jeden anzufügenden Parametersatz mit *APPEND := TRUE* aufzurufen. Auf diese Art können bis zu 255 Parametersätze aneinander gehängt werden. Den Versuch, weitere Parametersätze zu definieren, quittiert der Baustein mit *ERROR := 32* (kein Platz im Datensatzpuffer verfügbar, der Datensatz wurde ignoriert). Sämtliche Parametersätze basieren auf ein und derselben Zeitbasis. Ein Wechsel der Zeitbasis (Eingang *TB_IDX*) ist nur im deaktivieren Zustand des Impulsgenerators möglich. Der Index für den Basistakt wird nur mit der steigenden Flanke am Eingang *ENABLE* übernommen. Ist eine Impulsfolge vollständig übertragen und kein weiterer Parametersatz mehr vorhanden, schaltet sich der PTO-Generator selbständig ab und die Steuerung des digitalen Ausganges erfolgt wieder durch das Prozessabbild. Vom SPS-Programm ist daher im Prozessabbild ist der geforderte Zustand für den digitalen Ausgang zu hinterlegen, den dieser nach Deaktivierung des PTO-Generators annehmen soll. Der PTO-Generator schaltet sich ebenfalls selbständig ab, wenn bei der Berechnung der Periodendauer für den nachfolgenden Impuls ein Bereichsüber- bzw. -unterlauf auftritt. Das ist der Fall, wenn bei der Berechnung von T_n (siehe oben) ein Ergebnis größer 65535 oder kleiner 0 entsteht. Dieser Fehler wird vom Baustein mit *ERROR := 32* signalisiert (Bereichsüberschreitung bei der Neuberechnung der Periodendauer). Aufgrund der kumulativen Einbeziehung von *DELTA* kann dieser Fehler auch erst nach einer Reihe von vorangegangenen erfolgreichen Berechnungen entstehen.

PWM-Generator (PDO_MODE := FALSE, Impulsdauerausgang, kontinuierliche Impulsfolge):
In der Funktion als PWM-Generator wird eine kontinuierliche Impulsfolge am digitalen Ausgang erzeugt. Dabei sind sowohl die Periodendauer als auch die Impulsdauer als Anzahl der Basistakte einstellbar. Bei Eingang *ENABLE := TRUE* wird der PWM-Generator unmittelbar nach der Übergabe der Parameter aktiviert. Ist dabei der Wert für die Impulsdauer *PT* gleich 0, verharrt der zugehörige Ausgang für die gesamte Periodendauer in seinen inaktiven Zustand. Wird dagegen für die Impulsdauer *PT* ein Wert größer oder gleich der Periodendauer übergeben, so ist Ausgang für die gesamte Periodendauer aktiv. Das Ändern der Periodendauer erfolgt stets asynchron. Die laufende Periode wird unterbrochen, um den neuen Wert zu übernehmen. Das Ändern der Impulsdauer erfolgt synchron, die Impulsdauer wird beim Starten der nächsten Periodendauer übernommen. Zum Ändern von Parametern ist der Baustein mit Eingang *APPEND := TRUE* aufzurufen. Die Generierung der kontinuierlichen Impulsfolge wird bei Aufruf des Bausteins mit *ENABLE := FALSE* beendet. Tritt während der Ausführung ein Fehler auf, so schaltet sich der Funktionsbaustein selbständig ab. Eine erneute Verwendung des Bausteins ist erst nach dem Rücksetzen des Bausteins durch den Aufruf mit *ENABLE := FALSE* möglich.

Der Ausgang *READY* signalisiert, dass der Baustein vollständig parametriert und somit betriebsbereit ist. Eine Änderung des Parameters *TB_IDX* (Index zur Festlegung des Basistaktes für den

Impulsgenerator) ist nicht mehr möglich (*TB_IDX* wird nur bei steigender Flanke am Eingang *ENABLE* eingelesen). Beim Aufruf des Bausteins mit *ENABLE := FALSE* kehrt der Ausgang READY zurück nach FALSE.

Der Funktionsbaustein signalisiert bei seiner Rückkehr mit *BUSY := TRUE*, dass der Generator aktiv ist und die Steuerung des zugehörigen digitalen Ausganges übernommen hat (PTO-Modus: eine parametrierte Impulsfolge wird übertragen, PWM-Modus: eine kontinuierliche Impulsfolge wird generiert). Mit *BUSY := FALSE* wird angezeigt, dass der Generator inaktiv ist und die Beeinflussung des zugehörigen digitalen Ausgang über das Prozessabbild direkt durch das SPS-Programm erfolgt.

Die bei der Ausführung des Funktionsbausteines möglichen Fehler werden am Ausgang *ERROR* angezeigt und sind in Tabelle 24 beschrieben.

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *PTO_PWM* sowohl zum Generieren einmaliger Impulsfolgen im PTO-Modus als auch zum Generieren kontinuierlicher Impulsfolgen im PWM-Modus. Die Parametersätze sind so gewählt, dass sich die Impulsfolgen an der Status-LED des PWM-Ausganges ohne zusätzliche technische Hilfs- bzw. Messmittel beobachten lassen. Ein Zyklus im PTO-Modus wird durch eine positive Flanke am Eingang *xStartButtonPto* gestartet. Die Impulsfolge beginnt mit einem Impuls von 1 Sekunde (CT * TB = 1000 * 1 ms = 1 sec), jeder Folgeimpuls verkürzt sich im 50 ms (Delta = -50). Es werden insgesamt 15 Impulse generiert (PC = 15). Der PWM-Modus wird durch eine positive Flanke am Eingang *xStartButtonPwm* gestartet. Die generierte Impulsfolge hat eine Periodendauer von 500 ms (CT * TB = 500 * 1 ms = 0.5 sec -> 2 Hz), die Einschaltdauer je Impuls beträgt 150 ms (PT * TB = 150 * 1 ms = 150 ms).

Programmbeispiel

```
PROGRAM PtoPwm

VAR CONSTANT

    (* Definition of TimeBase-Index *)
    PTO_TB_IDX_800_US : USINT := 0;     (* TimeBase-Index 800us *)
    PTO_TB_IDX_1_MS   : USINT := 1;     (* TimeBase-Index 1ms   *)

    (* Error Codes of FB PTO_TAB *)
    PTOTAB_ERROR_SUCCESS         : USINT :=  0;
    PTOTAB_ERROR_HW_ERROR        : USINT :=  1;
    PTOTAB_ERROR_UNKNOWN_CHANNEL : USINT :=  2;
    PTOTAB_ERROR_UNKNOWN_TB_IDX  : USINT :=  8;
    PTOTAB_ERROR_DELTA_OVERFLOW  : USINT := 16;
    PTOTAB_ERROR_INVALID_TAB     : USINT := 64;

PTO_PWM_CHANNEL : USINT := 0;
END_VAR

VAR
    xStartButtonPto AT %IX0.0 : BOOL;   (* DI0 at PmC14/phyPS-412 *)
    xStartButtonPwm AT %IX0.1 : BOOL;   (* DI1 at PmC14/phyPS-412 *)
    xPtoPwmOut      AT %QX2.4 : BOOL;   (* P0  at PmC14/phyPS-412 *)

    FB_RTrigPto     : R_TRIG;
    FB_RTrigPwm     : R_TRIG;

    usiPtoTbIdx     : USINT := 1;           (* PTO_TB_IDX_1_MS *)
    uiPtoCt         : UINT  := 1000;
    iPtoDelta       : INT   := -50;
    udiPtoPc        : UDINT := 15;

    usiPwmTbIdx     : USINT := 1;           (* PTO_TB_IDX_1_MS *)
    uiPwmCt         : UINT  := 500;
    uiPwmPt         : UINT  := 150;
```

```
    xPtoAppend       : BOOL  := TRUE;
    xPtoReady        : BOOL  := FALSE;
    xPtoBusy         : BOOL  := FALSE;

    xPwmAppend       : BOOL  := TRUE;
    xPwmReady        : BOOL  := FALSE;
    xPwmBusy         : BOOL  := FALSE;

    FB_PtoPwm        : PTO_PWM;
    usiPtoPwmError   : USINT;
END_VAR

(* ----- Wait for Start ----- *)
WaitForStart:
CAL     FB_RTrigPto (CLK := xStartButtonPto)
LD      FB_RTrigPto.Q
JMPC    StartPtoMode

CAL     FB_RTrigPwm (CLK := xStartButtonPwm)
LD      FB_RTrigPwm.Q
JMPC    StartPwmMode

LD      xPtoBusy
JMPC    RunPtoMode

LD      xPwmBusy
JMPC    RunPwmMode

JMP     ProgExit


(* ----- Run PTO Mode ----- *)
StartPtoMode:
LD      FALSE           (* preset output state, this state is    *)
ST      xPtoPwmOut      (* used when PTO Generator isn't running *)

LD      FALSE           (* reset state flags *)
ST      xPtoReady
ST      xPtoBusy
ST      xPwmReady
ST      xPwmBusy

CAL     FB_PtoPwm (
        ENABLE := FALSE,
        CHANNEL := PTO_PWM_CHANNEL)

CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := TRUE,
        APPEND := xPtoAppend,
        TB_IDX := usiPtoTbIdx,
        CT := uiPtoCt,
        DELTA := iPtoDelta,
        PC := udiPtoPc,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPtoReady := READY,
        xPtoBusy := BUSY,
        usiPtoPwmError := ERROR)
```

```
RunPtoMode:
CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := TRUE,
        APPEND := FALSE,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPtoReady := READY,
        xPtoBusy := BUSY,
        usiPtoPwmError := ERROR)

JMP     ProgExit


(* ----- Run PWM Mode ----- *)
StartPwmMode:
LD      FALSE           (* preset output state, this state is    *)
ST      xPtoPwmOut      (* used when PTO Generator isn't running *)

LD      FALSE           (* reset state flags *)
ST      xPtoReady
ST      xPtoBusy
ST      xPwmReady
ST      xPwmBusy

CAL     FB_PtoPwm (
        ENABLE := FALSE,
        CHANNEL := PTO_PWM_CHANNEL)

CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := FALSE,
        APPEND := xPwmAppend,
        TB_IDX := usiPwmTbIdx,
        CT := uiPwmCt,
        PT := uiPwmPt,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPwmReady := READY,
        xPwmBusy := BUSY,
        usiPtoPwmError := ERROR)

RunPwmMode:
CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := FALSE,
        APPEND := FALSE,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPwmReady := READY,
        xPwmBusy := BUSY,
        usiPtoPwmError := ERROR)

JMP     ProgExit


(* ----- Cycle End ----- *)
ProgExit:
RET

END_PROGRAM
```

8.3 Der Funktionsbaustein PTO_TAB

Der Funktionsbaustein *PTO_TAB* dient zum indirekten Parametrieren des Impulsgenerators im PTO-Modus (Impulsfolgeausgang) mit Hilfe einer Parametersatz-Tabelle. Dieser Baustein ist nur auf Steuerungen verfügbar, die über PWM-Ausgänge verfügen.

Prototyp des Funktionsbausteines

```
              +-----------------+
              |     PTO_TAB     |
              |                 |
BOOL     ---|ENABLE      READY|--- BOOL
              |             BUSY|--- BOOL
              |                 |
USINT    ---|TB_IDX            |
              |                 |
[...]    ---|TABLE-------TABLE|--- [...]
UINT     ---|RECORDS           |
              |                 |
USINT    ---|CHANNEL     ERROR|--- USINT
              |                 |
              +-----------------+
```

Operandenbedeutung

TB_IDX	Index zur Festlegung des Basistaktes für den Impulsgenerator dieser Parameter ist abhängig von den Eigenschaften der jeweiligen Steuerung, gültige Werte sind z.B.: 0 = 800ns Basistakt 1 = 1ms Basistakt Der Basistakt gilt für alle Parametersätze der Tabelle und wird nur mit der steigenden Flanke am Eingang *ENABLE* übernommen.
TABLE	Parametersatz-Tabelle, enthält die Parametersätze der zu generierenden Impulsfolgen (siehe Text)
RECORDS	Anzahl belegter Einträge der in der am Eingang *TABLE* übergebenen Parametersatz-Tabelle (siehe Text)
ENABLE	Freigeben oder Sperren des Impulsgenerators TRUE = Aktivieren des Impulsgenerators, die am Eingang *TABLE* angegebene Parametersatz-Tabelle wird eingelesen und der Generator übernimmt die Steuerung des zugeordneten digitalen Ausganges FALSE = Deaktivieren des Impulsgenerators, die Steuerung des zugeordneten digitalen Ausganges erfolgt durch das Prozessabbild (direktes Beeinflussen des Ausganges durch das SPS-Programm) Mit der steigenden Flanke am Eingang *ENABLE* übernimmt der Funktionsbaustein den Index zur Festlegung des Basistaktes (Eingang *TB_IDX*).
READY	Status-Ausgang des Impulsgenerators TRUE = der Impulsgenerator wurde vollständig parametriert, der Generator ist betriebsbereit FALSE = der Impulsgenerator wurde noch nicht parametriert oder der Baustein wurde mit einem Fehler beendet, der Generator ist nicht betriebsbereit
BUSY	Status-Ausgang des Impulsgenerators TRUE = der Impulsgenerator ist aktiv (Impulsfolge wird generiert), die Steuerung des digitalen Ausganges erfolgt durch den Generator FALSE = der Impulsgenerator ist inaktiv (Impulsfolge abgeschlossen), die Steuerung des digitalen Ausganges erfolgt durch das Prozessabbild (direktes Beeinflussen des Ausganges durch das SPS-Programm)

CHANNEL Nummer des zu verwendenden Kanals

ERROR Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 25 definiert.

Tabelle 25: Error-Codes des Funktionsbausteines PTO_TAB

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
1	Hardwarefehler bei der Ausführung des Funktionsbausteines aufgetreten
2	Der ausgewählte Kanal (*CHANNEL*) wird nicht unterstützt
8	Der ausgewählte Index für den Basistakt (*TB_IDX*) wird nicht unterstützt
16	Bereichsüberschreitung bei der Neuberechnung der Periodendauer unter Einbeziehung von *DELTA* (Periodendauer ist größer 65535 oder kleiner 0)
64	Die am Eingang *TABLE* übergebene Parametersatz-Tabelle ist ungültig

Beschreibung

Der Funktionsbaustein ermöglicht ein indirektes Parametrieren des Impulsgenerators im PTO-Modus (Impulsfolgeausgang) mit Hilfe einer Parametersatz-Tabelle. Die Tabelle ist im SPS-Programm wie folgt zu definieren:

```
PTO_TABLE : ARRAY [0..255] OF PTO_RECORD;
```

Die Struktur *PTO_RECORD* ist in OpenPCS global definiert und besitzt folgenden Aufbau:

```
    PTO_RECORD : STRUCT
        CT    : UINT;
        DELTA : INT;
        PC    : UDINT;
    END_STRUCT;
```

Die Bedeutung der Parameter *CT*, *DELTA* und *PC* entspricht denen des Funktionsbausteines *PTO_PWM* (siehe Abschnitt 8.2). Gemäß den Definitionen der Norm IEC 61131-3 sind Typen von Variablen und Parametern exakt einzuhalten. Daher ist die Parametersatz-Tabelle im SPS-Programm immer mit 256 verfügbaren Einträgen anzulegen (*ARRAY [0..255] OF PTO_RECORD*). Die Anzahl der davon wirklich mit gültigen Parametern konfigurierten Datensätze ist am Eingang *RECORDS* zu spezifizieren. Sämtliche Parametersätze basieren auf ein und derselben Zeitbasis. Ein Wechsel der Zeitbasis (Eingang *TB_IDX*) ist nur im deaktivierten Zustand des Impulsgenerators möglich.

Mit der steigenden Flanke am Eingang *ENABLE* übernimmt der Baustein die Parametersatz-Tabelle am Eingang *TABLE* (wobei nur die als *RECORDS* spezifizierte Anzahl von Paramtersätzen berücksichtigt wird) und Startet die Generierung der Impulsfolgen. Ist die Tabelle vollständig abgearbeitet, schaltet sich der PTO-Generator selbständig ab und die Steuerung des digitalen Ausganges erfolgt wieder durch das Prozessabbild. Vom SPS-Programm ist daher im Prozessabbild ist der geforderte Zustand für den digitalen Ausgang zu hinterlegen, den dieser nach Deaktivierung des PTO-Generators annehmen soll. Der PTO-Generator schaltet sich ebenfalls selbständig ab, wenn bei der Berechnung der Periodendauer für den nachfolgenden Impuls ein Bereichsüber- bzw. -unterlauf auftritt. Das ist der Fall, wenn bei der Berechnung von T_n (siehe Beschreibung im Abschnitt 8.2) ein Ergebnis größer 65535 oder kleiner 0 entsteht. Dieser Fehler wird vom Baustein mit *ERROR := 32* signalisiert (Bereichsüberschreitung bei der Neuberechnung der Periodendauer). Aufgrund der kumulativen Einbeziehung von *DELTA* kann dieser Fehler auch erst nach einer Reihe von vorangegangenen erfolgreichen Berechnungen entstehen.

Das folgende Programmbeispiel zeigt die Verwendung des Funktionsbausteines *PTO_TAB* zum Generieren einmaliger Impulsfolgen mit Hilfe einer Parametersatz-Tabelle. Dabei wird die in Bild 5 dargestellte Motoransteuerung mit 3 Phasen simuliert (Anfahren, kontinuierlicher Lauf, Anhalten). Zur Simulation sind die Parametersätze sind so gewählt, dass sich die Impulsfolgen an der Status-LED des PWM-Ausganges ohne zusätzliche technische Hilfs- bzw. Messmittel beobachten lassen. Ein Zyklus wird durch eine positive Flanke am Eingang *xStartButton* gestartet.

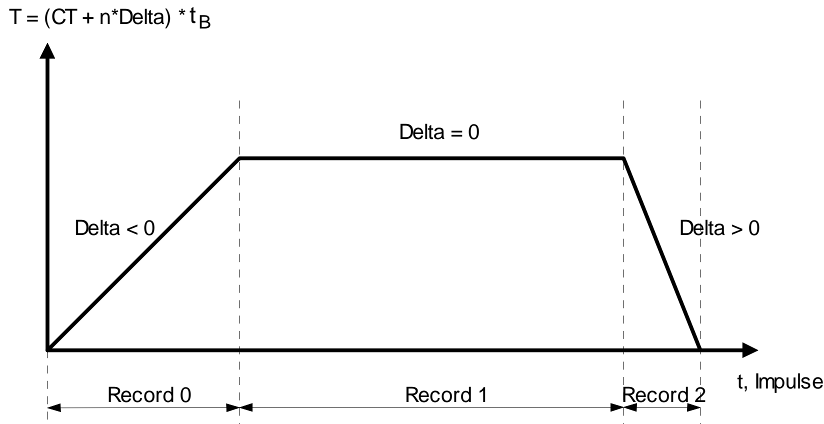


Bild 5: Zeitdiagramm zum Programmbeispiel "MotorCtl"

<u>Programmbeispiel</u>

```
PROGRAM MotorCtl
VAR CONSTANT
    (* Definition of TimeBase-Index *)
    PTO_TB_IDX_800_US : USINT := 0;     (* TimeBase-Index 800us *)
    PTO_TB_IDX_1_MS   : USINT := 1;     (* TimeBase-Index 1ms   *)

    (* Error Codes of FB PTO_TAB *)
    PTOTAB_ERROR_SUCCESS         : USINT :=  0;
    PTOTAB_ERROR_HW_ERROR        : USINT :=  1;
    PTOTAB_ERROR_UNKNOWN_CHANNEL : USINT :=  2;
    PTOTAB_ERROR_UNKNOWN_TB_IDX  : USINT :=  8;
    PTOTAB_ERROR_DELTA_OVERFLOW  : USINT := 16;
    PTOTAB_ERROR_INVALID_TAB     : USINT := 64;

    PTO_CHANNEL : USINT := 0;
END_VAR
```

```
VAR
    aPdoTab : ARRAY[0..255] OF PTO_RECORD :=
    [
        (* CT :  UINT   DELTA :  INT    PC :  UDINT *)
        (  CT := 1000,  DELTA := -100,  PC := 9      ),
        (  CT :=  100,  DELTA := 0,     PC := 50     ),
        (  CT :=  100,  DELTA := 200,   PC := 5      )
    ];
    uiRecords       : UINT  := 3;

    usiProcState    : USINT := 0;

    FB_PtoTab       : PTO_TAB;
    ausiError       : ARRAY[0..2] OF USINT;

    xStartButton AT %IX0.0 : BOOL;  (* DI0 at PmC14/phyPS-412 *)
    xMotorOut    AT %QX2.4 : BOOL;  (* P0  at PmC14/phyPS-412 *)

    FB_RTrig : R_TRIG;
END_VAR

(* ----- Select current program step ----- *)
LD      usiProcState
EQ      0
JMPC    WaitForStart
LD      usiProcState
EQ      1
JMPC    PtoInit
LD      usiProcState
EQ      2
JMPC    PtoSetTab
LD      usiProcState
EQ      3
JMPC    PtoRun
LD      0
ST      usiProcState


(* ----- Wait for Start ----- *)
WaitForStart:
CAL     FB_RTrig (CLK := xStartButton)
LD      FB_RTrig.Q
JMPCN   ProgExit

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Init PTO Generator ----- *)
PtoInit:
LD      FALSE           (* preset output state, this state is    *)
ST      xMotorOut       (* used when PTO Generator isn't running *)

CAL     FB_PtoTab (
        ENABLE := FALSE,
        CHANNEL := PTO_CHANNEL,
        TABLE := aPdoTab,
        RECORDS := 0
        |
        ausiError[0] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit
```

```
(* ----- Set Table ----- *)
PtoSetTab:
CAL     FB_PtoTab (
        ENABLE := TRUE,
        CHANNEL := PTO_CHANNEL,
        TB_IDX := PTO_TB_IDX_1_MS,
        TABLE := aPdoTab,
        RECORDS := uiRecords
        |
        ausiError[1] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Run PTO Generator ----- *)
PtoRun:
CAL     FB_PtoTab (
        ENABLE := TRUE,
        CHANNEL := PTO_CHANNEL,
        TB_IDX := PTO_TB_IDX_1_MS,
        TABLE := aPdoTab,
        RECORDS := uiRecords
        |
        ausiError[2] := ERROR)
LD      FB_PtoTab.BUSY
JMPC    ProgExit

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Cycle End ----- *)
ProgExit:
RET

END_PROGRAM
```

9 Verarbeitung von Prozessdaten

9.1 Anwendung des PID-Reglers

Ein Regler findet dann Anwendung, wenn Aufgrund zeitlich nicht vorhersagbarer Störgrößen das zeitliche Verhalten der Ausgangsgröße einer Strecke nicht unmittelbar durch die Eingangsgröße geführt werden kann. Die Aufgabe eines Reglers besteht darin, die Ausgangsgröße (Istwert, Process Variable PV) zu beobachten, mit der Führungsgröße (Sollwert, Setpoint SP, Regeldifferenz = Sollwert - Istwert) zu verglichen und über eine Stelleinrichtung die Eingangsgröße der Strecke zu korrigieren (siehe Bild 6). Im Ergbenis stellt sich eine neue, korrigierte Ausgangsgröße ein. Das System ist rückgekoppelt. Eine Regelung setzt die Beobachtung der Ausgangsgröße voraus. Unter Umständen sind deshalb geeignete Größen zu finden bzw. zu schaffen, um die Strecke zu beobachten.

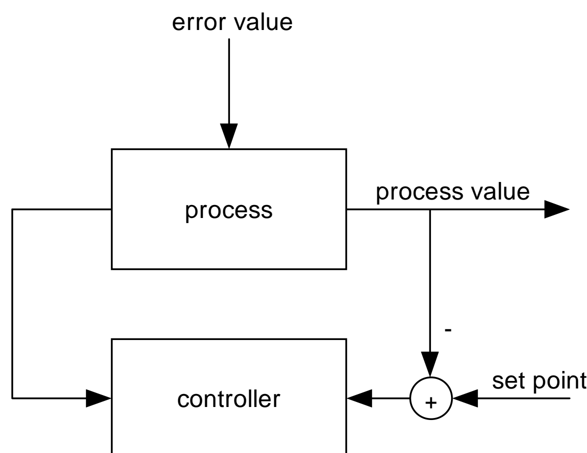


Bild 6: Prinizip eines Regelkreises

Im Gegensatz dazu fordert der bekannte Zusammenhang zwischen Ausgangsgröße, Störgröße und Eingangsgröße einer Strecke und das bekannte Verhalten der Störgröße keine Beobachtung der Ausgangsgröße, denn es ist zu jeder Zeit möglich durch gezielte Beeinflussung der Eingangsgröße entsprechend einer Sollgröße die Ausgangsgröße zu führen. In diesem Fall spricht man von einer Steuerung. Das System ist nicht rückgekoppelt.

Der Funktionsbaustein *PID1* (siehe Abschnitt 9.2) berechnet die Stellgröße CO (Controller Output) nach dem Verfahren der quasikontinuierlichen PID-Regelung (Proportional-, Integral-, Differentialregler) aus den Eingangswerten Sollwert SP (Setpoint) und Istwert PV (Process Variable). Die Eigenschaften des PID-Reglers bezüglich des Frequenz- und Phasenganges werden durch seine Parameter Reglerverstärkung KR, Differentialzeit (Vorhaltezeit) TD und Integralzeit (Nachstellzeit) TI und Abtastzeit T0 beschrieben.

Für die korrekte Funktion der Regelung ist der Baustein in gleich bleibenden Intervallen der Länge der Abtastzeit T0 aufzurufen.

Grundlagen (PID-Algorithmus)

Beim analogen Regler ist die Stellgröße y(t) das Ergebnis der Summe aus Proportionalanteil $y_P(t)$, Integralanteil $y_I(t)$ und Differentialanteil $y_D(t)$:

$$y(t) = y_P(t) + y_I(t) + y_D(t) + y_0$$

$$y(t) = K_R e(t) + \frac{K_R}{T_I}\int e(t)dt + K_R T_D \frac{de(t)}{dt} + y_0$$

$$e(t) = SP(t) - PV(t)$$

Durch Abtasten der Regeldifferenz kann diese Gleichung in eine quasikontinuierlichen PID-Regler überführt werden. Der Integralanteil wird hierbei durch eine Summe aller Regeldifferenzen und der Differentialanteil durch eine Differenz der letzten beiden Regeldifferenzen ersetzt.

$$y(kT0) = K_R[e(kT_0) + \frac{T_0}{T_I}\sum_{n=0}^{k-1} e(kT_0) + \frac{T_D(e(kT_0) - e((k-1)T_0)}{T_0} + y_0$$

Das Berechnen des Integralanteils kann dahingehend weiter vereinfacht werden, dass sich der neue Werte des Integralanteils aus dem Ergebnis des letzten Wertes plus der neuen Regeldifferenz ermitteln lässt. Dadurch ist es nicht erforderlich, sämtliche Werte der Regeldifferenz seit dem Start des Reglers zu speichern. Die Summe aller vorherigen Werte wird als Integralsumme, der Anfangswert y_0 der Integralsumme als Bias bezeichnet.

$$y_I(kT0) = K_R \frac{T_0}{T_I} e(kT_0) + y_I((k-1)T_0) + y_0$$

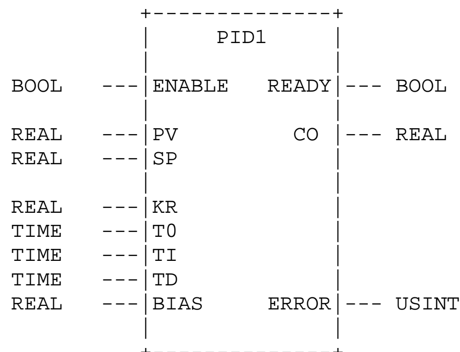
Der Differentialanteil wird durch die Differenz der beiden letzten Regeldifferenzen ersetzt. Um bei einer Änderung des Sollwertes keine Sprünge in der Stellgröße zu erzeugen, wird für den D-Anteil angenommen, dass der Sollwert zwischen zwei Abtastzeitpunkten stets konstant ist. Dadurch beschränkt sich die Berechnung des D-Anteils auf die Differenz der letzten beiden Istwerte ($PV(kT_0)$, $PV((k-1)T_0)$).

$$y_D(kT0) = K_R \frac{T_D(PV(kT_0) - PV((k-1)T_0)}{T_0}$$

9.2 Der Funktionsbaustein PID1

Der Funktionsbaustein PID1 realisiert einen PID-Reglerbaustein nach dem im Abschnitt 9.1 beschriebenen Regelalgorithmus.

Prototyp des Funktionsbausteines



Operandenbedeutung

PV	Normierter Istwert (Prozess Variable) der Regelstrecke gültige Werte liegen im Bereich von 0.0 ... 1.0, der Baustein selbst prüft diesen Parameter jedoch nicht auf Bereichsüberschreitung
SP	Normierter Sollwert (Führungsgröße, Setpoint) für die Regelung gültige Werte liegen im Bereich von 0.0 ... 1.0, der Baustein selbst prüft diesen Parameter jedoch nicht auf Bereichsüberschreitung
KR	Normierte Reglerverstärkung Die Verstärkung kann positiv oder negativ gewählt werden. Bei einer positiven Verstärkung zeigt der Regler Vorwärtsverhalten, bei einer negativen Verstärkung Rückwärtsverhalten. Bei einer Verstärkung von 0 wird bei der PID-Berechnung der P-Anteil zu Null und bleibt damit unberücksichtigt. Da die Verstärkung ebenfalls mit dem I- und D-Anteil verknüpft ist, wird in diesem Fall für den I- und D-Anteil eine Verstärkung von 1 verwendet.
T0	Abtastzeit des Reglers (gültiger Wertebereich T0 > 0)
TI	Integralzeit (Nachstellzeit)
TD	Differentialzeit (Vorhaltezeit)
BIAS	Anfangswert der Stellgröße bzw. Integralsumme beim Starten der PID-Berechnung (gültiger Wertebereich 0.0 ... 1.0)
ENABLE	Mit der steigenden Flanke werden die Reglerparameter KR, T0, TI, TD übernommen und die Integralsumme auf den Anfangswert BIAS gesetzt. Mit ENABLE = FALSE werden die Ausgänge READY, ERROR und CO zurückgesetzt. Der Baustein prüft bei der Übernahme der Parameter T0 und BIAS den Gültigkeitsbereich und signalisiert gegebenenfalls eine Bereichsüberschreitung am Fehlerausgang.
CO	Reglerausgang, berechnete Stellgröße des Reglers (Wertebereich 0.0 ... 1.0)

READY — Status-Ausgang des PID-Reglers
TRUE = der Reglerbaustein wurde vollständig parametriert und ist betriebsbereit
FALSE = der Reglerbaustein wurde noch nicht oder fehlerhaft parametriert (die Reglerparameter liegen außerhalb des Gültigkeitsbereichs), der Reglerbaustein ist nicht betriebsbereit

ERROR — Der Error-Code liefert Informationen über das Ausführungsergebnis des Funktionsbausteines, die möglichen Fehlercodes sind in Tabelle 26 definiert.

Tabelle 26: Error-Codes des Funktionsbausteines PID1

Error-Code	Bedeutung
0	Kein Fehler bei der Ausführung des Funktionsbausteines aufgetreten
8	Der angegebene Wert für den Parameter *BIAS* ist ungültig (kleiner 0 oder größer 1)
16	Der angegebene Wert für den Parameter *T0* ist ungültig (Zeit gleich 0)

Beschreibung

Der Funktionsbaustein realisiert einen PID-Reglerbaustein nach dem im Abschnitt 9.1 beschriebenen Regelalgorithmus. Mit der steigenden Flanke am Eingang *ENABLE* übernimmt der Baustein die Reglerparameter *KR*, *T0*, *TD*, *TI* sowie *BIAS* und startet den Regler. Für die erste Berechnung werden der Sollwert und der letzte Istwert auf den aktuellen Istwert gesetzt. Dadurch ergibt sich für die Stellgröße bei der ersten Berechnung stets der Wert *BIAS*, da sich für den Proportional-, Integral- und Differentialanteil Null einstellt. Durch die Wahl der Reglerparameter kann das Verhalten des Reglers beeinflusst werden. Wird für die Zeit *TI = t#0ms* angegeben, so wird der Integral-Anteil des Reglers nicht berechnet und zu Null gesetzt. Ist die Zeit *TD = t#0ms*, so ergibt sich für den Differential-Anteil der Wert Null. Wird die Verstärkung *KR* gleich Null gewählt, so entfällt der Proportional-Anteil. Da die Verstärkung *KR* ebenfalls mit dem Integral- und Differential-Anteil verknüpft ist, wird in diesem Fall für den I- und D-Anteil eine Verstärkung von 1 verwendet.

P-Regler — $TI = TD = 0, KR \neq 0$
PI-Regler — $TD = 0, KR, TI \neq 0$
PID-Regler — $KR, TI, TD \neq 0$

Für das Ermitteln der Reglerparameter werden in der Literatur verschiedene Verfahren beschrieben (Wendetangenten-Verfahren, Schwingungsversuch). Mit Hilfe von Simulationstools können ebenfalls die Parameter ermittelt werden. Dafür ist es jedoch erforderlich, dass die Strecke bezüglich des Frequenz- und Phasengangs beschrieben werden kann. Aus dem bekannten Verhalten kann dann für die relevanten Glieder der Strecke ein Modell erstellt werden, um in der Simulation die Parameter zu bestimmen.

Sollwert, Istwert als auch Bias und die Stellgröße werden als normierte Größen verwendet. Der Wertebereich für die normierten Größen liegt zwischen 0.0 und 1.0. Für den Bias erfolgt beim Starten des Bausteins eine Überprüfung des Gültigkeitsbereichs. Liegt der Wert außerhalb, dann wird das als Fehler am Ausgang *ERROR* signalisiert. Sollwert und Istwert werden aus Laufzeitgründen nicht geprüft. Die Stellgröße als auch die Integralsumme werden durch den Baustein begrenzt. Entsteht im Ergebnis der Berechnung ein negativer Wert, so wird die Größe auf 0.0 gesetzt, übersteigt das Ergebnis den Wert 1, so erfolgt die Begrenzung auf 1.0. Weiterhin wird die Integralsumme in Abhängigkeit von der Stellgröße nach folgender Vorschrift begrenzt:

- Ist das Ergebnis der Stellgröße größer 1.0, so wird die Integralsumme wie folgt berechnet:

  Integralsumme = 1.0 - (Proportionalanteil + Differentialanteil)

- Ist das Ergebnis der Stellgröße kleiner 0.0, so wird die Integralsumme wie folgt berechnet:

  Integralsumme = - (Proportionalanteil + Differentialanteil)

Normieren der Eingangswerte

Beim Normieren wird der Wertebereich einer Größe auf einen anderen Zahlenbereich abgebildet. Ein analoger Eingang besitzt einen Wertebereich von 0 ... 32767. Dieser Zahlenbereich ist auf den Wertebereich des Regler (0.0 ... 1.0) abzubilden:

$$y_{nom} = y / 32767$$

Für eine bipolare Eingangsgröße verdoppelt sich der abzubildende Wertebereich (-32768 ... +32767). Die Abbildung im positiven Zahlenbereich der normierten Größe wird hierbei mit einem Offset von 0.5 berücksichtigt:

$$y_{nom} = y / 65535 + 0.5$$

<u>Beispiel:</u>

(1) Der Istwert der Strecke wird mit einem analogen Eingang 0-10V aufgenommen. Der aktuelle Istwert beträgt 7.5V. Der Istwert wird als 15Bit-Wert = 7.5 * 32767 / 10 = 24575 im Prozessabbild hinterlegt:

$$y_{nom} = y / 32767 = 24575 / 32767 = 0.7499924$$

(2) Der Istwert der Strecke wird mit einem analogen Eingang ±10V aufgenommen. Der aktuelle Istwert beträgt 7.5V. Der Istwert wird als vorzeichenbehafteter 15Bit-Wert = 7.5 * 32767 / 10 = 24575 im Prozessabbild hinterlegt:

$$y_{nom} = y / 65535 + 0.5 = 24575 / 65535 + 0.5 = 0.87499$$

(3) Der Istwert der Strecke wird mit einem analogen Eingang ±10V aufgenommen. Der aktuelle Istwert beträgt -7.5V. Der Istwert wird als vorzeichenbehafteter 15Bit-Wert = -7.5 * 32767 / 10 = -24575 im Prozessabbild hinterlegt:

$$y_{nom} = y / 65535 + 0.5 = -24575 / 65535 + 0.5 = 0.12501$$

Normieren der Ausgangswerte

Für den Ausgangswert erfolgt das Entnormieren in umgekehrter Reihenfolge. Ein analoger Ausgang besitzt einen Wertebereich von 0 ... 32767. Dieser Zahlenbereich ist auf den Wertebereich des Reglerausganges (0.0 ... 1.0) abzubilden:

$$y = y_{nom} \cdot 32767$$

Für eine bipolare Ausgangsgröße verdoppelt sich der abzubildende Wertebereich (-32768 ... +32767). Die Abbildung im positiven Zahlenbereich der normierten Größe wird hierbei mit einem Offset von 0.5 berücksichtigt:

$$y = (y_{nom} - 0.5) \cdot 65535$$

Das nachfolgende Programmbeispiel zeigt die Ausführung einer Regelstrecke mit Hilfe des PLCmodule-C14. Die Strecke setzt sich aus mehreren Übertragungsgliedern 1.Ordnung zusammen:

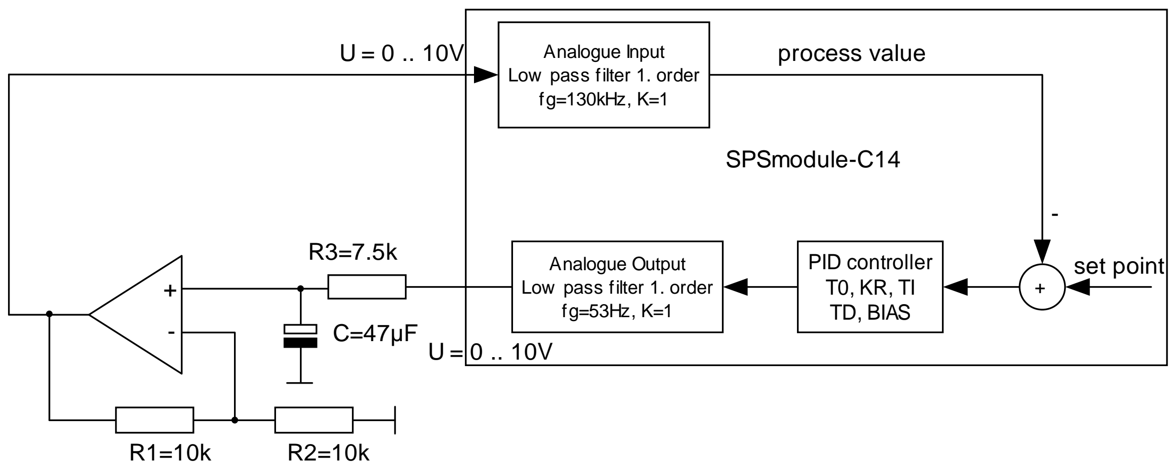


Bild 7: Aufbau der Regelstrecke zum Programmbeispiel "PidTest"

Der Istwert der Strecke wird über den Analogeingang AI0 gelesen, die Stellgröße wird mit dem Analogausgang AO0 erzeugt.

Programmbeispiel

```
PROGRAM PidTest

VAR CONSTANT
    (* Error Codes of FB PID1 *)
    PID1_ERR_SUCCESS              : USINT :=   0;
    PID1_ERR_INVALID_BIAS         : USINT :=   8;
    PID1_ERR_INVALID_T0           : USINT :=  16;
END_VAR

VAR_GLOBAL
    (* Prozess Variables *)
    ADC_Result    AT %IW8.0 : UINT;
    ControlOutput AT %QW8.0 : UINT;
END_VAR
```

```
VAR
    SetPoint_V   : REAL := 1.0;
    ProcessVar_V : REAL;
    Bias         : REAL;
    FB_PID       : PID1;
    FB_Timer     : TON;
END_VAR


(* to get periodical time stamps start an TON timer         *)
FB_Timer(IN := TRUE, PT := t#25ms);
IF (FB_Timer.Q = FALSE) THEN
      (* the timer intervall is not left                    *)
      RETURN;
END_IF;

(* The timer intervall is left. Restart the timer for next  *)
(* periode.                                                 *)
FB_Timer(IN := FALSE);
FB_Timer(IN := TRUE, PT := t#25ms);

(*----------------------------------------------------------*)
(* Prepare calculating PID algorithm                        *)
(* Scale the result of AD converter to a REAL number        *)
ProcessVar_V := UINT_TO_REAL(ADC_Result) * 10.0 / 32767.0;

(*----------------------------------------------------------*)
(* calculating PID algorithm                                *)
(* The inputs must scalled by 10.0V                         *)
FB_PID(ENABLE := TRUE,
       PV     := ProcessVar_V  / 10.0,
       SP     := SetPoint_V    / 10.0,
       KR     := 1.5,
       T0     := t#25ms,    (* sample time is 25ms          *)
       TI     := t#20ms,    (* integral time is 20ms        *)
       TD     := t#6ms,     (* derivative time is 6ms       *)
       BIAS   := Bias
      );

(* The result is scalled to unsigned integer value.         *)
ControlOutput := REAL_TO_UINT (FB_PID.CO * 32767.0);
(* The control output is stored to bias to prevent high     *)
(* steps in the reaction curve of controler output if a re- *)
(* start (the PLC was stopped and starts again) is happend. *)
Bias          := FB_PID.CO;
RETURN;

END_PROGRAM
```

Bild 8 verdeutlicht die Regelwirkung des oben aufgeführten Programmbeispiels anhand der Änderung des Istwertes beim Sprung der Führungsgröße (Sollwert) von 1V auf 6V.

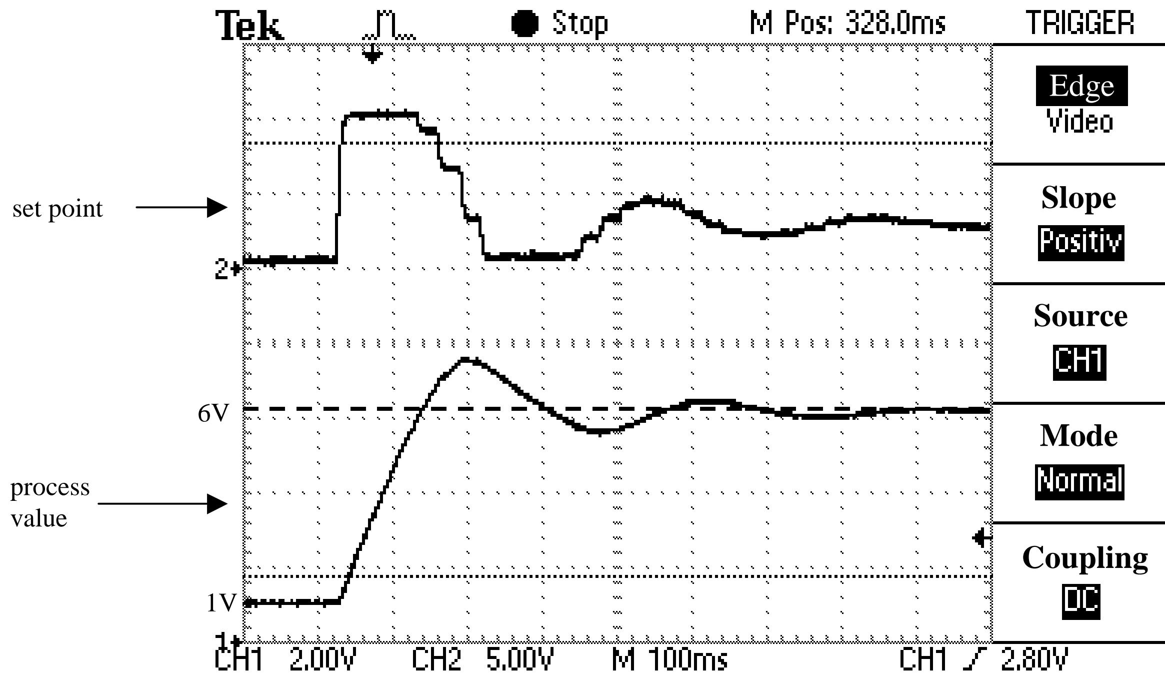


Bild 8: "PidTest" - Änderung Istwert beim Sprung der Führungsgröße (Sollwert) von 1V auf 6V

10 Index

A

ASC 19

B

BIN_TO_STR 21

C

CHR 18
CNT_FUD 77
CONCAT 15
Counter
  Übersicht 77

D

DELETE 16
DT_ABS_TO_REL 85
DT_CLOCK 82
DT_REL_TO_ABS 86

E

Echtzeituhr
  Übersicht 82
Error-Task 7
ETRC 8
Event-Task
  Übersicht 7

F

FIND 17

G

GETSTRINFO 17

I

Impulsgenerator
  Übersicht 88
INSERT 15

L

LAN_ASCII_TO_INET 27
LAN_GET_HOST_BY_ADDR 29
LAN_GET_HOST_BY_NAME 28
LAN_GET_HOST_CONFIG 26
LAN_INET_TO_ASCII 28
LAN_UDP_CLOSE_SOCKET 31
LAN_UDP_CREATE_SOCKET 30
LAN_UDP_RECVFROM_BIN 35
LAN_UDP_RECVFROM_STR 32
LAN_UDP_SENDTO_BIN 36
LAN_UDP_SENDTO_STR 33
LEFT 13
LEN 13

M

MID 14

N

NVDATA
  Übersicht 41
NVDATA_BIN 50
NVDATA_BIT 41
NVDATA_INT 44
NVDATA_STR 47

P

PID1 102
PID-Regler
  Normierung der Ausgangswerte 105
  Normierung der Eingangswerte 104
  Übersicht 100
PTO_PWM 89
PTO_TAB 95
PTRC 11
PWM,PTO
  Übersicht 88

R

REPLACE 16
RIGHT 14
RTC
  Übersicht 82

S

Serielle Schnittstelle
  Übersicht 53
SIO
  Übersicht 53
SIO_INIT 53
SIO_READ_BIN 70
SIO_READ_CHR 59
SIO_READ_STR 63
SIO_STAT 56
SIO_WRITE_BIN 72
SIO_WRITE_CHR 60
SIO_WRITE_STR 65
Start-Task 7
Stop-Task 7
STR 19
STR_TO_BIN 23

V

VAL 20

Z

Zähler
  Übersicht 77

Dokument: **SYS TEC spezifische Erweiterungen für OpenPCS / IEC 61131-3**
Dokumentnummer: **L-1054-04, Juli 2011**

Wie würden Sie dieses Handbuch verbessern?

Haben Sie in diesem Handbuch Fehler entdeckt? Seite

Eingesandt von:

Kundennummer: ______

Name: ______

Firma: ______

Adresse: ______

Einsenden an:

SYS TEC electronic GmbH
August-Bebel-Str. 29
D-07973 Greiz, Germany
Fax : +49 (0) 36 61 / 6279-99