Speicherprogrammierbare Steuerungen

Teil 3: Programmiersprachen
(IEC 61131-3:2003) Deutsche Fassung EN 61131-3:2003

DIN
EN 61131-3

ICS 25.040.40; 35.060; 35.240.50

Ersatz für
DIN EN 61131-3:1994-08
Siehe Beginn der Gültigkeit

Programmable controllers – Part 3: Programming languages
(IEC 61131-3:2003); German version EN 61131-3:2003

Automates programmables – Partie 3: Langages de programmation
(CEI 61131-3:2003); Version allemande EN 61131-3:2003

Die Europäische Norm EN 61131-3:2003 hat den Status einer Deutschen Norm.

Beginn der Gültigkeit

Die EN 61131-3 wurde am 2002-12-01 angenommen.

Daneben darf DIN EN 61131-3:1994-08 noch bis 2005-12-01 angewendet werden.

Nationales Vorwort

Für die vorliegende Norm ist das nationale Arbeitsgremium K 962 „SPS“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE zuständig.

Norm-Inhalt war veröffentlicht als E DIN IEC 65B/373/CD:1999-03.

Die enthaltene IEC-Publikation wurde vom SC 65B „Devices“ erarbeitet.

Das IEC-Komitee hat entschieden, dass der Inhalt dieser Publikation bis zum Jahr 2007 unverändert bleiben soll. Zu diesem Zeitpunkt wird entsprechend der Entscheidung des Komitees die Publikation
- bestätigt,
- zurückgezogen,
- durch eine Folgeausgabe ersetzt oder
- geändert.

Fortsetzung Seite 2
und 223 Seiten EN

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE

Änderungen

Gegenüber DIN EN 61131-3:1994-08 wurden folgende Änderungen vorgenommen:

- Doppel-Byte-Strings für die Darstellung von Zeichen in allen Fremdsprachen, auch z. B. Japanisch,
- direkte Adressierung mit noch nicht vollständigen Pfadnamen mittels „Wild-Card“,
- Deklarationen von temporären Variablen,
- Attribute RETAIN und NON_RETAIN für die Kennzeichnung von gepufferten und nicht-gepufferten Variablen,
- formale und nicht-formale Aufrufe bei Funktionen (d. h. mit und ohne Parameter-Namen auch für Funktionsargumente, wie bei Funktionsbausteinen),
- BCD-Wandlungsfunktionen, Funktionen für die Zeit-Datentypen,
- erweiterte Initialisierungsmöglichkeiten für Funktionsbausteine, Initialisierung von Konfiguration,
- außerdem gibt es einige Erweiterungen für die Anweisungsliste und bei Sprachelementen der Ablaufsprache,
- die erweiterten Spracheigenschaften führten neben den Änderungen der Semantik zu entsprechenden Änderungen in der formalen Syntax im Anhang B der Norm.

Frühere Ausgaben

DIN EN 61131-3:1994-08

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf eine Norm ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils neueste gültige Ausgabe der in Bezug genommenen Norm.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe der Norm.

Der Zusammenhang der zitierten Normen mit den entsprechenden Deutschen Normen ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

IEC hat 1997 die Benummerung der IEC-Publikationen geändert. Zu den bisher verwendeten Normnummern wird jeweils 60000 addiert. So ist zum Beispiel aus IEC 68 nun IEC 60068 geworden.

Seit der Veröffentlichung der 1. Ausgabe der IEC 61131-3 im Jahre 1993 hat sich das Umfeld der Norm stark verändert. Während dieser Zeit wurde die Norm in großem Umfang weltweit angewendet, und dabei wurden viele Erfahrungen mit der praktischen Anwendung der Norm gemacht. Eine Anzahl von Ungereimtheiten, Widersprüchen und ungelösten Fragen wurde dabei gefunden. Außerdem zeigten sich Anforderungen, die unnötig schwer zu realisieren sind. Die Nutzer der Norm, z. B. SPS-Hersteller und -Anwender, sowie Software-Häuser, die SPS-Programmiertool entwickeln, lieferten viele Änderungs- und Verbesserungsvorschläge.

Um die von IEC 61131-3 Anwendern geleisteten Investitionen zu erhalten und den Nutzen der existierenden norm-konformen Steuerungssoftware sogar zukünftig auszudehnen, wurde die nun vorliegende 2. Ausgabe erstellt. Sämtliche Änderungen und Erweiterungen sind strikt aufwärts-kompatibel, d. h. ein Anwenderprogramm, das norm-konform zur 1. Ausgabe ist, muss dies auch bezüglich 2. Ausgabe bleiben. Die einzige Ausnahme ist der Gebrauch der eckigen statt der runden Klammern bei den Datenfeldern und einiger Namensänderungen bei den Zeitdaten-Funktionen.

EUROPÄISCHE NORM

EUROPEAN STANDARD

NORME EUROPÉENNE

EN 61131-3

März 2003

ICS 25.040; 35.240.50

Ersatz für EN 61131-3:1993

Deutsche Fassung

Speicherprogrammierbare Steuerungen

Teil 3: Programmiersprachen
(IEC 61131-3:2003)

Programmable controllers –
Part 3: Programming languages
(IEC 61131-3:2003)

Automates programmables –
Partie 3: Langages de programmation
(CEI 61131-3:2003)

Diese Europäische Norm wurde von CENELEC am 2002-12-01 angenommen. Die CENELEC-Mitglieder sind gehalten, die CEN/CENELEC-Geschäftsordnung zu erfüllen, in der die Bedingungen festgelegt sind, unter denen dieser Europäischen Norm ohne jede Änderung der Status einer nationalen Norm zu geben ist.

Auf dem letzten Stand befindliche Listen dieser nationalen Normen mit ihren bibliographischen Angaben sind beim Zentralsekretariat oder bei jedem CENELEC-Mitglied auf Anfrage erhältlich.

Diese Europäische Norm besteht in drei offiziellen Fassungen (Deutsch, Englisch, Französisch). Eine Fassung in einer anderen Sprache, die von einem CENELEC-Mitglied in eigener Verantwortung durch Übersetzung in seine Landessprache gemacht und dem Zentralsekretariat mitgeteilt worden ist, hat den gleichen Status wie die offiziellen Fassungen.

CENELEC-Mitglieder sind die nationalen elektrotechnischen Komitees von Belgien, Dänemark, Deutschland, Finnland, Frankreich, Griechenland, Irland, Island, Italien, Luxemburg, Malta, den Niederlanden, Norwegen, Österreich, Portugal, Schweden, der Schweiz, der Slowakei, Spanien, der Tschechischen Republik, Ungarn und dem Vereinigten Königreich.

CENELEC

Europäisches Komitee für Elektrotechnische Normung
European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique

Zentralsekretariat: rue de Stassart, 35 B-1050 Brüssel

Ref. Nr. EN 61131-3:2003 D

Vorwort

Der Text des Schriftstücks 65B/456/FDIS, zukünftige 2. Ausgabe von IEC 61131-3, ausgearbeitet von dem SC 65B „Devices“ des IEC TC 65 „Industrial-process measurement and control“, wurde der IEC-CENELEC Parallelen Abstimmung unterworfen und von CENELEC am 2002-12-01 als EN 61131-3 angenommen.

Diese Europäische Norm ersetzt EN 61131-3:1993.

Nachstehende Daten wurden festgelegt:

– spätestes Datum, zu dem die EN auf nationaler Ebene durch Veröffentlichung einer identischen nationalen Norm oder durch Anerkennung übernommen werden muss (dop): 2003-10-01

– spätestes Datum, zu dem nationale Normen, die der EN entgegenstehen, zurückgezogen werden müssen (dow): 2005-12-01

Anhänge, die als „normativ“ bezeichnet sind, gehören zum Norminhalt.
Anhänge, die als „informativ“ bezeichnet sind, enthalten nur Informationen.
In dieser Norm sind die Anhänge A, B, C, D, E und ZA normativ und sind die Anhänge F und G informativ.
Der Anhang ZA wurde von CENELEC hinzugefügt.

Anerkennungsnotiz

Der Text der Internationalen Norm IEC 61131-3:2003 wurde von CENELEC ohne irgendeine Abänderung als Europäische Norm angenommen.

Inhalt

Bilder

Seite

Seite

Tabellen

Seite

1 Allgemeines

1.1 Anwendungsbereich

Dieser Teil der IEC 61131 legt die Syntax und Semantik von Programmiersprachen für *Speicherprogrammierbare Steuerungen* fest, wie sie in Teil 1 der IEC 61131 definiert sind.

Die Funktionen der Programmeingabe, des Tests, der Beobachtung, des Betriebssystems usw. sind in Teil 1 der IEC 61131 festgelegt.

1.2 Normative Verweisungen

Die folgenden zitierten Dokumente sind für die Anwendung dieses Dokuments erforderlich. Bei datierten Verweisungen gilt nur die in Bezug genommene Ausgabe. Bei undatierten Verweisungen gilt die letzte Ausgabe des in Bezug genommenen Dokuments (einschließlich aller Änderungen).

IEC 60050 (alle Teile), *International Electrotechnical Vocabulary (IEV).*

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems.*

IEC 60617-12:1991, *Grafical symbols for diagrams – Part 12: Binary logic elements.*

IEC 60617-13:1978, *Grafical symbols for diagrams – Part 13: Analogue elements.*

IEC 60848: 2002, *GRAFCET Specification language for sequential function charts.*

IEC 61131-1, *Programmable controllers – Part 1: General information.*

IEC 61131-5, *Programmable controllers – Part 5: Communications.*

ISO/AFNOR:1989, *Dictionary of Computer Science – The standardised vocabulary.*

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

1.3 Begriffe

Für die Anwendung dieses Teil der IEC 61131 sind die folgenden Definitionen anzuwenden. Definitionen, die für alle Teile der IEC 61131 gelten, sind in Teil 1 angegeben.

ANMERKUNG 1 Begriffe, die in diesem Abschnitt definiert sind, sind *kursiv* angegeben, wenn sie innerhalb von Definitionen auftreten.

ANMERKUNG 2 Die Angabe „(ISO)“, die einer Definition folgt, gibt an, dass die Definition aus dem ISO/AFNOR Dictionary of Computer Science entnommen ist.

ANMERKUNG 3 Das ISO/AFNOR Dictionary of Computer Science und die IEC 60050 sollten für Begriffe zu Rate gezogen werden, die nicht in dieser Norm definiert sind.

1.3.1
absolute Zeit (absolute time)
Kombination von Tageszeit und Datum

1.3.2
Zugriffspfad (access path)
Verknüpfung eines symbolischen Namens mit einer Variablen zum Zweck der offenen Kommunikation

1.3.3
Aktion (action)
Boolesche Variable oder eine Sammlung von Operationen, die zusammen mit einer zugehörigen Kontrollstruktur ausgeführt wird, wie in 2.6.4 definiert ist

1.3.4
Aktionsblock (action block)
grafisches Sprachelement, das eine boolesche Eingangsvariable benutzt, um den Wert einer booleschen Ausgangsvariablen oder die Freigabebedingung für eine *Aktion* zu bestimmen; dies erfolgt nach einer vorbestimmten Kontrollstruktur, wie in 2.6.4.5 definiert ist

1.3.5
Aggregat (aggregate)
strukturierte Sammlung von Datenobjekten, die einen *Datentyp* bilden (ISO)

1.3.6
Argument (argument)
Synonym für *Eingangsvariable*, *Ausgangsvariable* oder *Ein-Aus-Variable*

1.3.7
Feld (array)
Aggregat, das aus Datenobjekten mit identischen Attributen besteht, von denen jedes eindeutig mit einem *Index* (subscript) referenziert werden kann (ISO)

1.3.8
Zuweisung (assignment)
Mechanismus, um einer Variablen oder einem *Aggregat* einen Wert zu geben (ISO)

1.3.9
basisbezogene Zahl (based number)
Zahl, die mit einer festgelegten Basis ungleich zehn dargestellt ist

1.3.10
bistabiler Funktionsbaustein (bistable function block)
Funktionsbaustein mit zwei stabilen Zuständen, gesteuert durch einen oder mehr Eingänge

1.3.11
Bitfolge (bit string)
Datenelement, bestehend aus einem oder mehr Bits

1.3.12
Rumpf (body)
Teil einer *Programm-Organisationseinheit*, der die Operationen festlegt, die mit den deklarierten *Operanden* einer *Programm-Organisationseinheit* auszuführen sind, wenn ihre Ausführung *aufgerufen* wird

1.3.13
Aufruf (call)
Sprachkonstrukt zum *Aufruf* der Ausführung einer *Funktion* oder eines *Funktionsbausteins*

1.3.14
Zeichenfolge (character string)
Aggregat, das aus einer geordneten Folge von Zeichen besteht

1.3.15
Kommentar (comment)
Sprachkonstrukt zur Einbindung von Text in ein Programm, ohne Auswirkung auf die Programmausführung zu haben (ISO)

1.3.16
Kompilieren (compile)
Übersetzen einer *Programm-Organisationseinheit* oder einer *Datentyp-Festlegung* in ihr Äquivalent der Maschinensprache oder einer Zwischenform

1.3.17
Konfiguration (configuration)
Sprachelement, das einem *SPS-System* entspricht, wie es in IEC 61131-1 definiert ist

1.3.18
Zähler-Baustein (counter function block)
Funktionsbaustein, der einen Wert entsprechend der Anzahl der erkannten Wechsel an einem oder mehreren festgelegten *Eingängen* akkumuliert

1.3.19
Datentyp (data type)
Menge von Werten zusammen mit der Menge von zulässigen Operationen (ISO)

1.3.20
Datum und Uhrzeit (date and time)
Datum im Jahr und die Tageszeit, dargestellt als ein einzelnes Sprachelement

1.3.21
Deklaration (declaration)
Mechanismus zur Festlegung der Definition eines *Sprachelements*. Eine *Deklaration* umfasst normalerweise die Verbindung eines Bezeichners mit dem Sprachelement und ihre Zuordnung von Attributen, wie *Datentypen* und Algorithmen

1.3.22
Begrenzungszeichen (delimiter)
Zeichen oder Zeichenkombination, zur Trennung von *Sprachelementen*

1.3.23
direkte Darstellung (direct representation)
Mittel zur Darstellung einer Variablen in einem SPS-Programm, von der eine hersteller-spezifische Entsprechung zu einem physikalischen oder *logischen Speicherort* direkt bestimmt werden kann

1.3.24
Doppelwort (double word)
Datenelement, das 32 Bits enthält

1.3.25
Auswertung (evaluation)
Vorgang der Ermittlung eines Wertes für einen Ausdruck oder eine *Funktion* oder für die *Ausgänge* eines *Netzwerks* oder *Funktionsbausteins* während der Programmbearbeitung

1.3.26
Element zur Ausführungssteuerung (execution control element)
Sprachelement, das den Fluss der Programmausführung steuert

1.3.27
fallende Flanke (falling edge)
Wechsel von 1 nach 0 bei einer booleschen Variablen

1.3.28
Funktion (function)
Programm-Organisationseinheit, die bei der Ausführung genau ein Datenelement liefert und möglicherweise zusätzliche *Ausgangsvariable* (die Mehrfachwerte sein dürfen, z. B. ein *Feld* oder eine *Struktur*) und deren *Aufruf* in Textsprachen als ein *Operand* in einem Ausdruck benutzt werden kann

1.3.29
Funktionsbaustein-Instanz (Funktionsbaustein) (function block instance, function block)
Instanz eines *Funktionsbaustein-Typs*

1.3.30
Funktionsbaustein-Typ (function block type)
SPS-*Sprachelement*, bestehend aus: (1) der Definition einer Datenstruktur, geteilt in Eingangs-, Ausgangs- und interne Variablen und (2) einer Menge von Operationen, die mit den Elementen der Datenstruktur durchgeführt werden, wenn eine *Instanz* des *Funktionsbaustein-Typs aufgerufen* wird

1.3.31
Funktionsbaustein-Sprache (function block diagram)
Netzwerk, in dem die Knoten *Funktionsbaustein-Instanzen*, grafisch dargestellte *Funktionen (Prozeduren)*, *Variablen, Literale* und *Marken* sind

1.3.32
allgemeiner Datentyp (generic data type)
Datentyp, der mehr als einen Typ von Daten darstellt, wie es in 2.3.2 festgelegt ist

1.3.33
globaler Geltungsbereich (global scope)
Geltungsbereich einer Deklaration, anwendbar auf alle Programm-Organisationseinheiten innerhalb einer *Ressource* oder *Konfiguration*

1.3.34
globale Variable (global variable)
Variable, deren *Geltungsbereich global* ist

1.3.35
hierarchische Adressierung (hierarchical addressing)
direkte Darstellung eines Datenelements als ein Teil einer physikalischen oder logischen Hierarchie; z. B. der Anschlusspunkt einer Baugruppe (Modul), die in einem Rahmen steckt, der wiederum in einem Schrank eingebaut ist usw.

1.3.36
Bezeichner (identifier)
Kombination von Buchstaben, Zahlen und Unterstrich-Zeichen, wie in 2.1.2 festgelegt, die mit einem Buchstaben oder einem Unterstrich beginnt und die ein *Sprachelement* bezeichnet

1.3.37
Ein-Aus-Variable (in-out variable)
Variable, die in einem `VAR_IN_OUT … END_VAR` Block deklariert ist

1.3.38
Anfangswert (initial value)
Wert, der einer *Variablen* beim Systemanlauf zugewiesen wird

1.3.39
Eingangsparameter (Eingang) (input parameter, input)
Variable, die benutzt wird, um einer *Programm-Organisationseinheit* ein Argument zu übergeben

1.3.40
Instanz (Instanz)
individuelles, benanntes Exemplar einer Datenstruktur, verknüpft mit einem *Funktionsbaustein-Typ* oder *Programm-Typ*, das von einem *Aufruf* der zugehörigen Operationen bis zum nächsten erhalten bleibt

1.3.41
Instanz-Name (Instanz name)
Bezeichner, verknüpft mit einer festgelegten *Instanz*

1.3.42
Instanziierung (instantiation)
Erzeugung einer *Instanz*

1.3.43
ganzzahliges Literal (integer literal)
Literal, das direkt einen Wert vom Typ `SINT, INT, DINT, LINT, BOOL, BYTE, WORD, DWORD` oder `LWORD` darstellt, wie in 2.3.1 definiert

1.3.44
Aufruf (invocation)
Einleitungsvorgang zur Ausführung von Operationen, die in einer *Programm-Organisationseinheit* festgelegt sind

1.3.45
Schlüsselwort (keyword)
lexikalische Einheit, die ein *Sprachelement* charakterisiert, z. B. „`IF`“

1.3.46
Marke (label)
Sprachkonstruktion, die eine Anweisung, ein Netzwerk oder eine Gruppe von Netzwerken bezeichnet und die einen *Bezeichner* einschließt

1.3.47
Sprachelement (language element)
Element, das durch ein Symbol auf der linken Seite der Produktionsregeln in der formalen Spezifikation identifiziert wird, die in Anhang B dieser Norm angegeben ist

1.3.48
Literal (literal)
lexikalische Einheit, die direkt einen Wert darstellt (ISO)

1.3.49
lokaler Geltungsbereich (local scope)
Geltungsbereich einer *Deklaration* oder *Marke*, der nur für die *Programm-Organisationseinheit* gilt, in der die *Deklaration* oder *Marke* auftritt

1.3.50
logischer Speicherort (logical location)
Speicherort einer *hierarchisch adressierten* Variablen in einem Schema, das eine Relation zu der physikalischen Struktur der Eingänge, der Ausgänge und des Speichers der SPS besitzen darf oder auch nicht

1.3.51
lange Realzahl (long real)
Realzahl, die in einem *Langwort* dargestellt ist

1.3.52
Langwort (long word)
64-Bit-Datenelement

1.3.53
Speicher (Anwenderdatenspeicher) (memory, user data starage)
funktionale Einheit, in die das Anwenderprogramm Daten speichern kann und aus der es die gespeicherten Daten zurückgewinnen kann

1.3.54
bezeichnetes Element (named element)
Element einer *Struktur*, das durch seinen zugehörigen *Bezeichner* benannt ist

1.3.55
Netzwerk (network)
Anordnung von Knoten und miteinander verbundenen Zweigen

1.3.56
Ausschaltverzögerung (Einschaltverzögerung) (off-delay, on-delay timer function block)
Funktionsbaustein, der die *fallende (steigende) Flanke* eines booleschen *Eingangs* um eine festgelegte Dauer verzögert

1.3.57
Operand (operand)
Sprachelement, mit dem eine Operation durchgeführt wird

1.3.58
Operator (operator)
Symbol, das die Aktion darstellt, mit der eine Operation durchgeführt wird

1.3.59
Ausgangsvariable (Ausgang) (output variable, output)
Variable, die benutzt wird, um das (die) Ergebnis(se) der *Auswertung* einer *Programm-Organisationseinheit* zurückzugeben

1.3.60
Überladen (overloaded)
Fähigkeit, bei einer Operation oder einer *Funktion* Operationen mit Daten verschiedenen Typs auszuführen, wie es in 2.5.1.4 festgelegt ist

1.3.61
Stromfluss (power flow)
symbolischer Fluss des elektrischen Stroms in einem Kontaktplan, benutzt, um das fortschreitende logische Berechnen eines Algorithmus zu beschreiben

1.3.62
Pragma (pragma)
Sprachkonstrukt zur Einbindung von Text in eine *Programm-Organisationseinheit*, das die Vorbereitung des Programms zur Ausführung beeinflussen darf

1.3.63
Programmieren (program)
Entwerfen, schreiben und testen von Anwenderprogrammen

1.3.64
Programm-Organisationseinheit (program organization unit)
Funktion, Funktionsbaustein oder Programm

ANMERKUNG Dieser Begriff darf sich sowohl auf einen Typ als auch auf eine Instanz beziehen.

1.3.65
Real-Literal (real literal)
Literal, das Daten vom Typ `REAL` oder `LREAL` darstellt

1.3.66
Ressource (resource)
Sprachelement, das einer „Signalverarbeitungsfunktion“ und ihrer „Mensch-Maschine-Schnittstelle“ und den „Funktionen der Sensor- und Aktor-Schnittstelle“ entspricht, sofern sie vorhanden sind, wie es in IEC 61131-1 definiert ist

1.3.67
gepufferte Daten (retentive data)
Daten, die so gespeichert sind, dass ihr Wert nach einer Folge von Spannungsausfall/Wiederkehr unverändert erhalten bleibt

1.3.68
Rücksprung (return)
Sprachkonstruktion innerhalb einer *Programm-Organisationseinheit*, die ein Ende der Ausführungsfolgen in der Einheit bezeichnet

1.3.69
steigende Flanke (rising edge)
Wechsel einer booleschen Variablen von 0 nach 1

1.3.70
Geltungsbereich (scope)
Teil eines *Sprachelements,* in dem eine *Deklaration* oder eine *Marke* gültig ist

1.3.71
Semantik (semantics)
Beziehungen zwischen den symbolischen Elementen einer Programmiersprache und ihrer Bedeutung, Auslegung und Anwendung

1.3.72
semigrafische Darstellung (semigrafic representation)
Darstellung der grafischen Information durch die Anwendung eines eingeschränkten Zeichensatzes

1.3.73
Einzel-Datenelement (single data element)
Datenelement, das aus einem einzelnen Wert besteht

1.3.74
Einzelelement-Variable (single element variable)
Variable, die ein *Einzel-Datenelement* darstellt

1.3.75
Schritt (step)
Situation, in der das Verhalten einer *Programm-Organisationseinheit* in Bezug auf ihre *Ein-* und *Ausgänge* einer Menge von Regeln folgt, die durch die zugehörigen *Aktionen* des *Schritts* definiert sind

1.3.76
strukturierter Datentyp (structured data type)
zusammengesetzter *Datentyp* (*Aggregat*), der unter Verwendung einer *Deklaration* `STRUCT` oder `FUNCTION_BLOCK` deklariert wurde

1.3.77
Indizierung (subscribting)
Mechanismus zur Adressierung eines *Datenfeldes* durch einen Verweis auf ein Feld und einen oder mehrere Ausdrücke, die bei *Auswertung* die Position des Elementes bezeichnen

1.3.78
symbolische Darstellung (symbolic representation)
Anwendung von *Bezeichnern,* um *Variablen* mit Namen zu versehen

1.3.79
Task (task)
Element zur Ausführungssteuerung, das für eine periodische oder getriggerte Ausführung einer Gruppe von zugeordneten *Programm-Organisationseinheiten* sorgt

1.3.80
Zeitliteral (time literal)
Literal, das Daten vom Typ `TIME, DATE, TIME_OF_DAY` oder `DATE_AND_TIME` darstellt

1.3.81
Transition (transition)
Bedingung, bei der die Steuerung von einem oder mehreren *Vorgängerschritten* entlang einer gerichteten Verbindung zu einem oder mehreren Nachfolgerschritten übergeht

1.3.82
vorzeichenlose ganze Zahl (unsigned integer)
ganzzahliges Literal, das kein führendes Plus(+)- oder Minus(-)-Zeichen enthält

1.3.83
verdrahtetes ODER (wired OR)
Konstruktion zur Ausführung der booleschen `ODER`-Funktion im Kontaktplan durch ein Verbinden der rechten Enden der horizontalen Verbindungen mit vertikalen Verbindungen

1.4 Übersicht und allgemeine Anforderungen

Dieser Teil der IEC 61131 legt die Syntax und Semantik einer vereinheitlichten Reihe von Programmiersprachen für Speicherprogrammierbare Steuerungen (SPS) fest. Diese umfassen zwei Textsprachen AWL (Anweisungsliste) und ST (Strukturierter Text) und zwei grafische Sprachen KOP (Kontaktplan) und FBS (*Funktionsbaustein-Sprache*).

Die Elemente der Ablaufsprache (AS) sind zur Strukturierung der internen Organisation von SPS-*Programmen* und -*Funktionsbausteinen* definiert. Außerdem sind *Konfigurationselemente* definiert, die zur Installation von SPS-*Programmen* in die SPS-Systeme dienen.

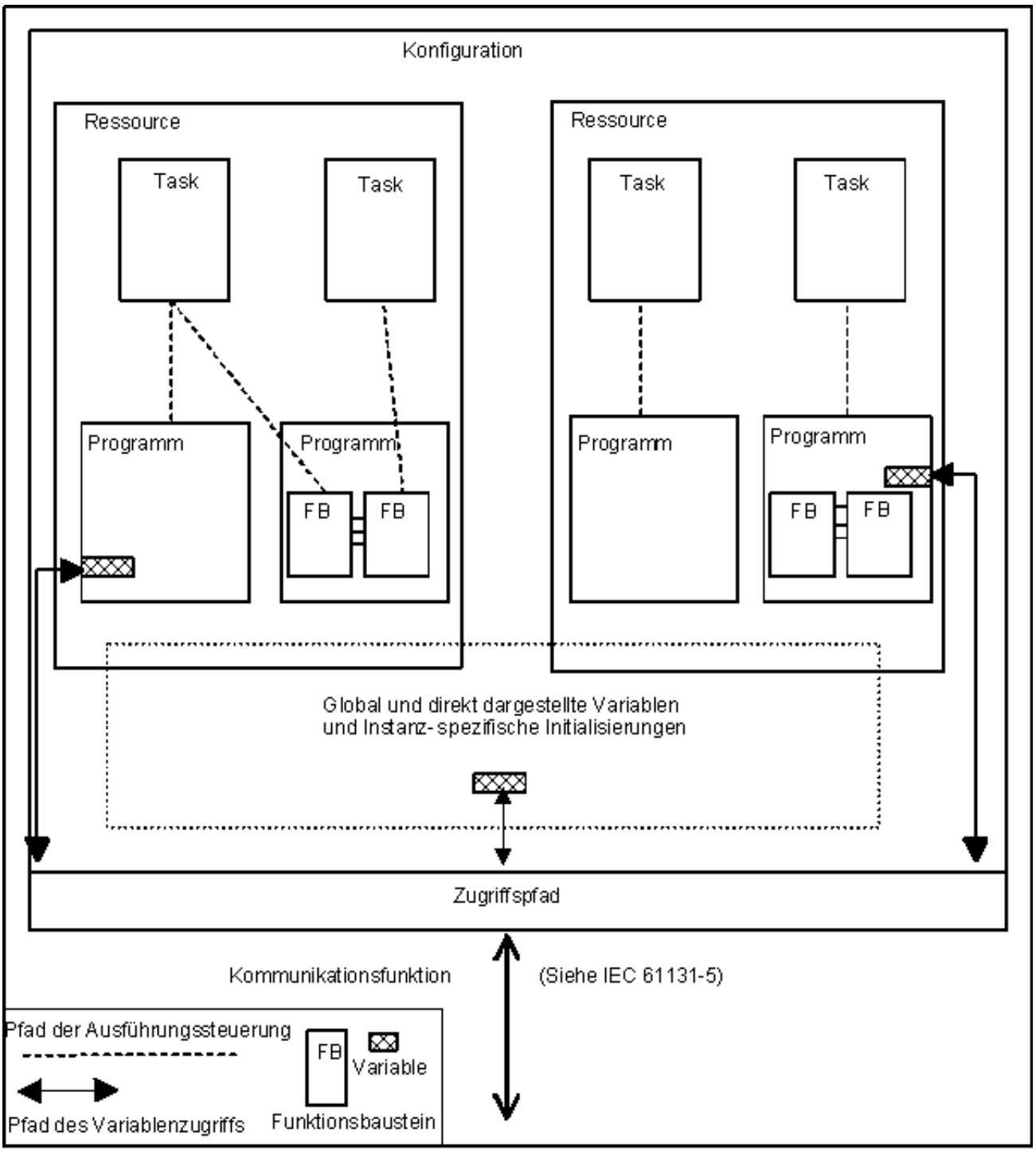
Zusätzlich sind Mittel definiert, die die Kommunikation zwischen Speicherprogrammierbaren Steuerungen und anderen Komponenten von automatisierten Systemen ermöglichen.

Die Elemente der Programmiersprachen in diesem Teil dürfen in einer interaktiven Programmierumgebung angewendet werden. Die Festlegung derartiger Umgebungen gehört nicht zum Geltungsbereich dieser Norm; eine solche Umgebung muss jedoch eine Programmdokumentation in Text- oder Grafik-Formaten erzeugen können, wie sie in diesem Teil festgelegt sind.

Der Stoff in diesem Teil ist von „unten nach oben“ („bottom up“) angelegt, d. h. die einfachen Sprachelemente werden zuerst dargestellt, um die Vorwärtsverweise im Text gering zu halten. Der Rest dieses Abschnitts bietet eine Übersicht über den Stoff, der in diesem Teil dargestellt ist, und enthält einige allgemeine Anforderungen.

1.4.1 Software-Modell

Die grundlegenden Hochsprachenelemente und ihre Beziehungen untereinander sind in Bild 1 veranschaulicht. Sie bestehen aus Elementen, die durch Anwendung der Sprachen *programmiert* werden, die in dieser Norm definiert sind, d. h. *Programme* und *Funktionsbausteine*, des Weiteren *Konfigurationselemente* wie *Konfigurationen*, *Ressourcen*, *Tasks*, *globale Variablen* und *Zugriffspfade* und Instanz-spezifische Initialisierungen, die zur Installation von SPS-*Programmen* in den SPS-Systemen dienen.



ANMERKUNG 1 Dieses Bild dient nur zur Veranschaulichung. Die grafische Darstellung ist nicht normativ.

ANMERKUNG 2 In einer Konfiguration mit einer einzigen Ressource muss die Ressource nicht explizit dargestellt werden.

Bild 1 – Software-Modell

Eine *Konfiguration* ist ein Sprachelement, das einem *SPS-System* entspricht, wie es in IEC 61131-1 definiert ist. Eine *Ressource* entspricht einer „Signalverarbeitungsfunktion“ und ihrer „Mensch-Maschine-Schnittstelle“ und sofern vorhanden den Funktionen der „Sensor- und Aktor-Schnittstelle“, wie in IEC 61131-1 definiert. Eine *Konfiguration* enthält eine oder mehrere *Ressourcen*, von denen jede ein oder mehrere *Programme* enthält, die unter der Steuerung von null oder mehr *Tasks* ausgeführt werden. Ein *Programm* kann null oder mehr *Funktionsbausteine* oder andere Sprachelemente enthalten, wie es in diesem Teil definiert ist.

Konfigurationen und *Ressourcen* können mit Hilfe der Funktionen „Bedienerschnittstelle“, „Programmierung, Testen und Beobachten“ oder „Betriebssystem“ gestartet oder gestoppt werden, wie es in IEC 61131-1 definiert ist. Das Starten einer *Konfiguration* muss die Initialisierung ihrer *globalen Variablen* bewirken, nach den Regeln, die in 2.4.2 angegeben sind, gefolgt vom Starten aller *Ressourcen* in der *Konfiguration*. Das Starten einer *Ressource* muss die Initialisierung aller *Variablen* in der *Ressource* bewirken, gefolgt von der Freigabe aller *Tasks* in der *Ressource*. Das Stoppen einer *Ressource* muss das Sperren aller ihrer *Tasks* bewirken, während das Stoppen einer *Konfiguration* das Stoppen aller ihrer *Ressourcen* bewirken muss. Die Mechanismen zur Steuerung von *Tasks* sind in 2.7.2 definiert, während die Mechanismen zum Starten und Stoppen von *Konfigurationen* und *Ressourcen* über Kommunikationsfunktionen in IEC 61131-5 definiert sind.

Programme, *Ressourcen*, *globale Variablen*, *Zugriffspfade* (und ihre zugehörigen Zugriffsberechtigungen) und *Konfigurationen* können durch die „Kommunikationsfunktion“ geladen und gelöscht werden, die in IEC 61131-1 definiert ist. Das Laden und Löschen einer *Konfiguration* oder *Ressource* muss gleichbedeutend mit dem Laden und Löschen aller der Elemente sein, die sie enthält.

Zugriffspfade und ihre zugehörigen Zugriffsberechtigungen sind in 2.7.1 definiert.

Die Abbildung der in diesem Abschnitt definierten Sprachelemente auf die Kommunikationsobjekte ist in IEC 61131-5 definiert.

1.4.2 Kommunikationsmodell

Bild 2 veranschaulicht die Wege, über die die Werte von Variablen zwischen Software-Elementen ausgetauscht werden können.

Wie in Bild 2 a) gezeigt, können die Variablenwerte innerhalb eines Programms direkt ausgetauscht werden durch die Verbindung des Ausgangs des einen Programmelements mit dem Eingang des anderen. Diese Verbindung wird in den grafischen Sprachen explizit gezeigt und ist in den Textsprachen implizit angegeben.

Variablenwerte können zwischen Programmen derselben Konfiguration über *globale Variablen* ausgetauscht werden, wie es die Variable *x* in Bild 2 b) veranschaulicht. Diese Variablen müssen als `GLOBAL` in der Konfiguration und als `EXTERNAL` im Programm deklariert werden, wie es in 2.4.3 festgelegt ist.

Wie in Bild 2 c) veranschaulicht, können die Werte von Variablen zwischen Teilen eines Programms, zwischen Programmen in derselben oder verschiedenen Konfigurationen oder zwischen einem SPS-Programm und einem Nicht-SPS-System ausgetauscht werden, indem die Kommunikationsbausteine verwendet werden, die in IEC 61131-5 definiert und in 2.5.2.3.5 beschrieben sind. Außerdem können Speicherprogrammierbare Steuerungen oder Nicht-SPS-Systeme Daten übertragen, die durch die *Zugriffspfade* zur Verfügung gestellt werden, wie es in Bild 2 d) veranschaulicht ist, indem die Mechanismen benutzt werden, die in IEC 61131-5 definiert sind.

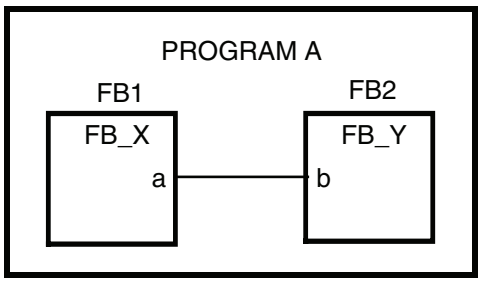


Bild 2 a) – Datenflussverbindung innerhalb eines Programms

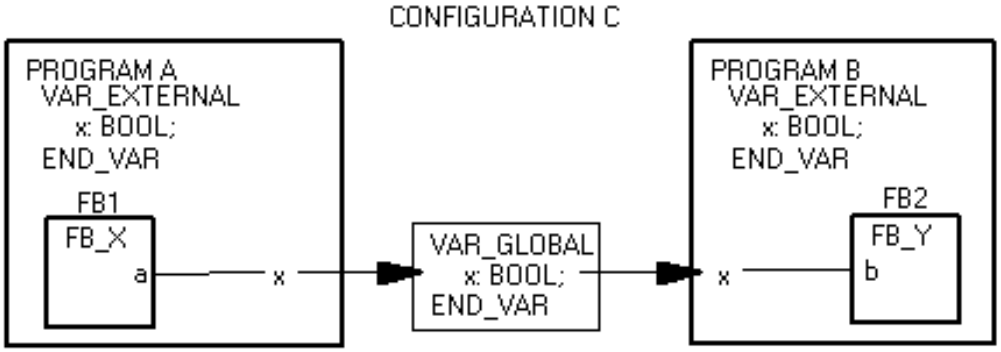


Bild 2 b) – Kommunikation über globale Variable

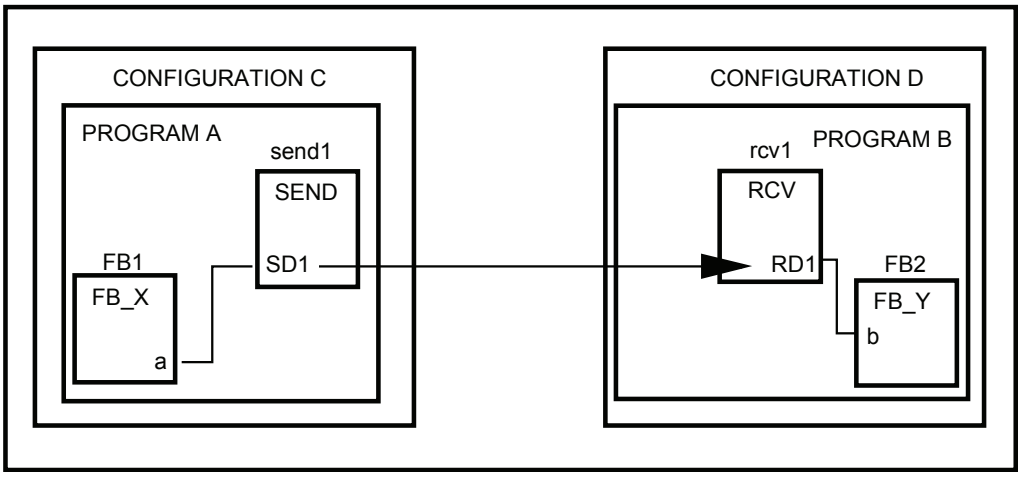


Bild 2 c) – Kommunikationsfunktionsbausteine

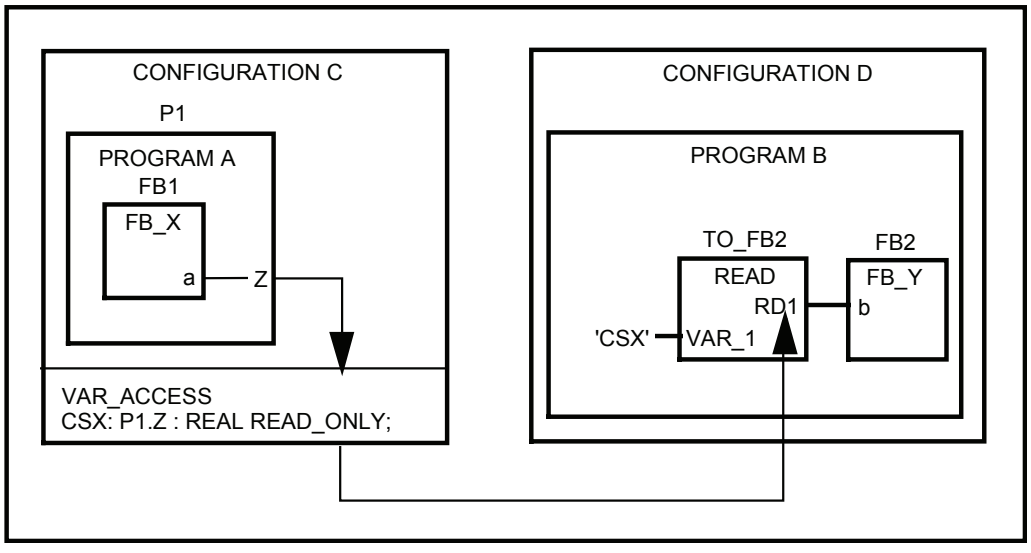


Bild 2 d) – Kommunikation über Zugriffspfade

ANMERKUNG 1 Dieses Bild dient nur zur Veranschaulichung. Die grafische Darstellung ist nicht normativ.

ANMERKUNG 2 In diesen Beispielen sollen die Konfigurationen C und D jeweils eine einzige Ressource haben.

ANMERKUNG 3 Die Einzelheiten der Kommunikationsbausteine sind in diesem Bild nicht gezeigt. Siehe 2.5.2.3.5 und IEC 61131-5.

ANMERKUNG 4 Wie in 2.5 festgelegt, können die Zugriffspfade auf direkt dargestellte Variable, globale Variable oder Eingangs-, Ausgangs- oder interne Variable von Programmen oder Funktionsbaustein-Instanzen deklariert werden.

ANMERKUNG 5 IEC 61131-5 legt die Mittel fest, mit denen sowohl SPS als auch Nicht-SPS-Systeme Zugriffspfade zum Lesen und Schreiben von Variablen benutzen können.

1.4.3 Programmiermodell

Die Elemente der SPS-Programmiersprachen und die Abschnitte, in denen sie in diesem Teil erscheinen, sind folgendermaßen eingeteilt:

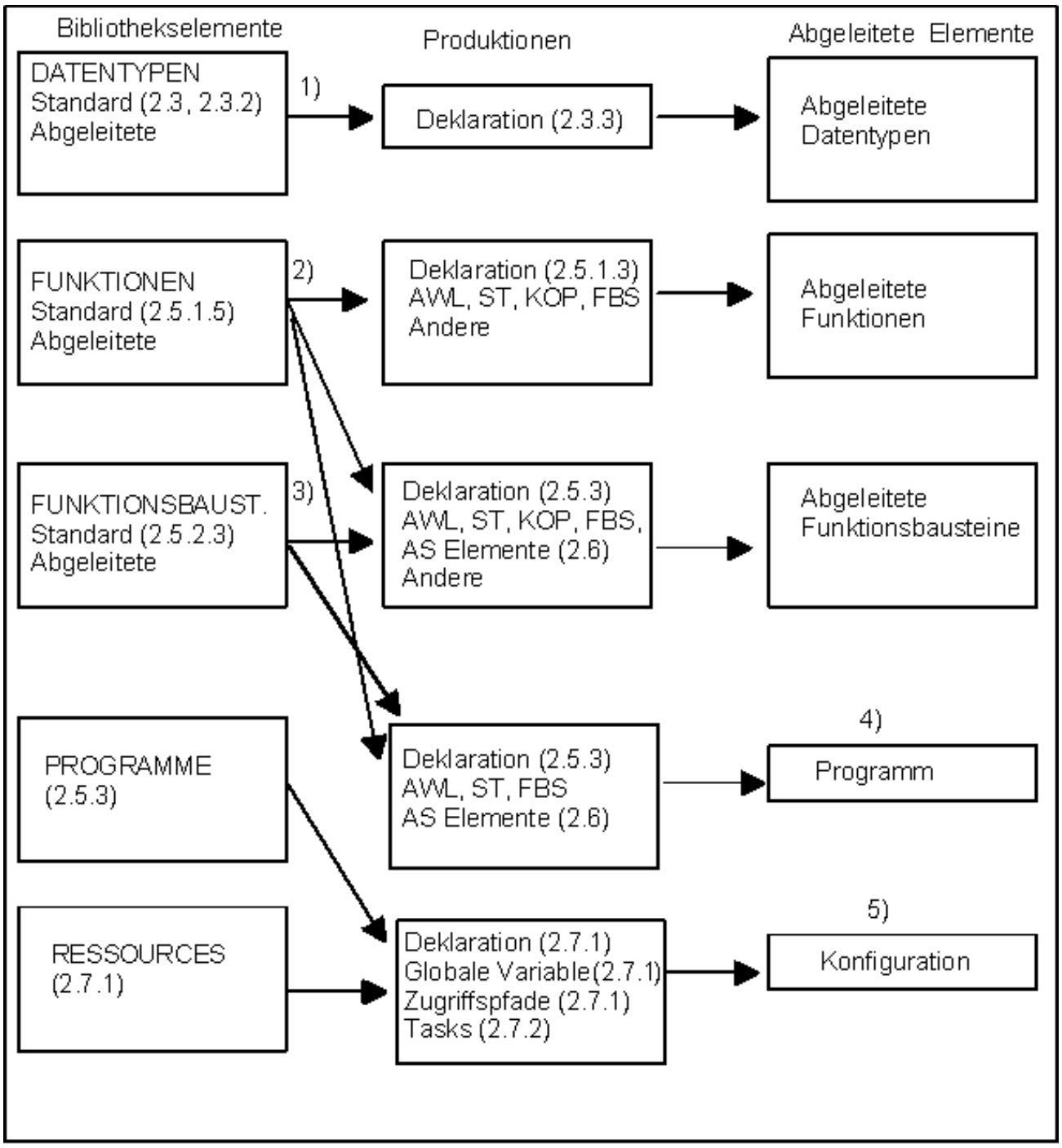
- Datentypen (2.3)
- Variable (2.4)
- Programm-Organisationseinheiten (2.5)
  - Funktionen (2.5.1)
  - Funktionsbausteine (2.5.2)
  - Programme (2.5.3)
- Ablaufsprache- (AS) Elemente (2.6)
- Konfigurationselemente (2.7)
  - Globale Variable (2.7.1)
  - Ressourcen (2.7.1)
  - Zugriffspfade (2.7.1)
  - Tasks (2.7.2)

Wie in Bild 3 gezeigt, muss die Kombination dieser Elemente die folgenden Regeln beachten:

1) Abgeleitete *Datentypen* müssen deklariert werden, wie es in 2.3.3 festgelegt ist, indem die Standard-Datentypen, die in 2.3.1 und 2.3.2 festgelegt sind, und alle vorher abgeleiteten Datentypen benutzt werden.
2) Abgeleitete *Funktionen* können deklariert werden, wie es in 2.5.1.3 festgelegt ist, indem Standard- oder abgeleitete Datentypen, Standardfunktionen, die in 2.5.1.5 definiert sind, und alle vorher abgeleiteten Funktionen benutzt werden. Diese Deklaration muss die Mechanismen für die Sprache AWL, ST, KOP oder FBS definieren.
3) Abgeleitete *Funktionsbausteine* können deklariert werden, wie es in 2.5.2.2 festgelegt ist, indem Standard- und abgeleitete Datentypen und Funktionen, Standard-Funktionsbausteine, definiert in 2.5.2.3, und alle vorher abgeleiteten Funktionsbausteine benutzt werden. Diese Deklaration muss die Mechanismen benutzen, die für die Sprache AWL, ST, KOP oder FBS definiert sind und kann die Elemente der Ablaufsprache (AS) einschließen, die in 2.6 definiert sind.
4) Ein *Programm* muss deklariert werden, wie es in 2.5.3 festgelegt ist, indem Standard- oder abgeleitete Datentypen, Funktionen und Funktionsbausteine benutzt werden. Diese Deklaration muss Mechanismen benutzen, die definiert sind für die Sprache AWL, ST, KOP oder FBS und kann die Elemente der Ablaufsprache (AS) einschließen, die in 2.6. definiert sind.
5) *Programme* können zu *Konfigurationen* kombiniert werden, indem die Elemente benutzt werden, die in 2.7 definiert sind, d. h. *globale Variable*, *Ressourcen*, *Tasks* und *Zugriffspfade*.

Der Hinweis auf „vorher abgeleitete“ Datentypen, Funktionen und Funktionsbausteine in den obigen Regeln soll bedeuten, dass, wenn ein derartiges abgeleitetes Element einmal deklariert wurde, seine Definition z. B. in einer „Bibliothek“ mit abgeleiteten Elementen zur Verfügung steht, um sie für weitere Ableitungen zu benutzen. Deshalb darf die Deklaration eines abgeleiteten Elementtyps nicht in der Deklaration eines anderen abgeleiteten Elementtyps enthalten sein.

Eine andere Programmiersprache als eine von denen, die in dieser Norm definiert sind, darf in der Deklaration von *Funktionen* oder *Funktionsbausteinen* benutzt werden. Die Mittel, mit denen ein Anwenderprogramm, das in einer der Sprachen geschrieben ist, die in dieser Norm definiert sind, die Ausführung einer derartigen abgeleiteten Funktion oder Funktionsbausteins aufruft und die Zugriffe auf die zugehörigen Daten ausführt, müssen so beschaffen sein, wie in dieser Norm definiert ist.



ANMERKUNG 1 Die eingeklammerten Zahlen (1) bis (5) verweisen auf die entsprechenden Absätze in 1.4.3.

ANMERKUNG 2 Datentypen werden in allen Produktionen benutzt. Die entsprechenden Verbindungen sind der Klarheit wegen in diesem Bild weggelassen.

Bild 3 – Kombination der SPS-Sprachelemente

KOP – Kontaktplan – 4.2
FBS – Funktionsbaustein-Sprache – 4.3
AWL – Anweisungsliste – 3.2
ST – Strukturierter Text – 3.3
ANDERE – Andere Programmiersprachen – 1.4.3

1.5 Normerfüllung

Dieser Abschnitt definiert die Anforderungen, die von SPS-Systemen und -Programmen erfüllt werden müssen, die den Anspruch auf Erfüllung dieses Teils der IEC 61131 erheben.

1.5.1 Normerfüllung des Systems

Ein SPS-System, wie in IEC 61131-1 definiert ist, das den Anspruch erhebt, vollständig oder teilweise die Anforderungen dieses Teils der IEC 61131 zu erfüllen, muss dies genau so tun, wie es unten beschrieben ist.

Eine Aussage zur Normerfüllung muss in der Dokumentation enthalten sein, die mit dem System geliefert wird oder muss vom System selbst erzeugt werden. Die Form der Aussage zur Normerfüllung muss lauten:

„Dieses System erfüllt die Anforderungen der IEC 61131-3 in folgenden Eigenschaften der Sprache:“,

gefolgt von einer Reihe von Tabellen in folgendem Format:

Tabellen-Überschrift

Tabellen-Nummer	Eigenschaftsnummer	Beschreibung der Eigenschaften
...	...	...

Die Nummern der Tabellen und Eigenschaften und die Beschreibungen sind aus den Tabellen zu entnehmen, die in den relevanten Abschnitten dieses Teils der IEC 61131 angegeben sind. Die Tabellenüberschriften sind aus der folgenden Tabelle zu entnehmen.

Tabellen-Überschrift	Für Eigenschaft in:
Gemeinsame Elemente	Abschnitt 2
Gemeinsame Text-Elemente	Abschnitt 3.1
AWL Sprach-Elemente	Abschnitten 3.2.1 bis 3.2.3
ST Sprach-Elemente	Abschnitten 3.3.1 bis 3.3.2.4
Gemeinsame Grafik-Elemente	Abschnitt 4.1
KOP-Sprach-Elemente	Abschnitt 4.2
FBS Sprach-Elemente	Abschnitt 4.3

Die Tabellen 9, 11, 13, 16a, 16b, 32, 38, 47, 48 und 51 dürfen nicht zur Normerfüllung als Eigenschaften-Tabellen betrachtet werden.

Ein SPS-System, das die Anforderungen dieses Teils bezüglich einer Sprache erfüllt, die in diesem Teil definiert ist:

a) darf nicht ersetzende oder zusätzliche Sprachelemente einschließen, um eine der Eigenschaften zu erreichen, die in diesem Teil festgelegt sind, es sei denn, solche Elemente sind unten in den Regeln e) und f) angegeben;

b) muss ein Schriftstück mitliefern, das die Daten aller **implementierungsabhängiger Parameter** angibt, wie es in Anhang D aufgelistet ist;

c) muss imstande sein, zu unterscheiden, ob ein Sprachelement des Anwenders eine Anforderung dieser Norm verletzt oder nicht; muss erkennen können, wo ein solcher Verstoß als **Fehler** nach Anhang E bezeichnet wird und das Ergebnis dieser Unterscheidung dem Anwender melden. Falls das System nicht die gesamte Programm-Organisationseinheit prüft, muss der Anwender darauf hingewiesen werden,

dass die Prüfung unvollständig ist, auch wenn keine Verletzung in dem untersuchten Teil der Programm-Organisationseinheit entdeckt wurde;

d) muss jeden Verstoß des Anwenders, der als **Fehler** in Anhang E bezeichnet ist, in mindestens einer der folgenden Weisen behandeln:

   1) es muss eine Aussage in einem mitgelieferten Schriftstück gemacht werden, dass der Fehler nicht gemeldet wird;

   2) das System muss imstande sein, während der Aufbereitung des Programms für die Ausführung melden, dass ein Auftreten dieses Fehlers möglich ist;

   3) das System muss den Fehler während der Aufbereitung des Programms für die Ausführung melden;

   4) das System muss den Fehler während der Ausführung des Programms melden und geeignete Verfahren zur Behandlung der Fehler in System- oder Anwender-definierter Form einleiten;

   und falls irgendwelche Verstöße, die als Fehler bezeichnet sind, in einer Weise behandelt werden, wie oben in Abschnitt d) 1) beschrieben ist, dann muss eine Anmerkung in einem eigenen Abschnitt des mitgelieferten Schriftstücks erscheinen, die auf jede derartige Behandlung hinweist;

e) muss mit einem Schriftstück versehen sein, das separat alle Systemeigenschaften beschreibt, die vom System akzeptiert werden und die in diesem Teil verboten oder nicht festgelegt sind. Derartige Eigenschaften sind als „Erweiterungen zur <Sprache> Sprache, die in IEC 61131-3 definiert ist“ zu beschreiben;

f) muss imstande sein, in gleicher Weise jeden Gebrauch einer derartigen Erweiterung so zu verarbeiten, wie es für Fehler festgelegt ist;

g) muss imstande sein, in einer gleichen Weise, wie es für Fehler festgelegt ist, jeden Gebrauch einer der **implementierungsabhängigen Eigenschaften**, die in Anhang D festgelegt sind, zu verarbeiten;

h) darf nicht die in dieser Norm definierten Namen für Standard-Datentypen, Funktionen oder Funktionsbausteine für hersteller-definierte Eigenschaften nutzen, deren Funktionalität sich von dem unterscheidet, was in dieser Norm beschrieben ist; es sei denn, solche Elemente sind oben in den Regeln e) und f) angegeben;

i) muss mit einem Schriftstück versehen sein, das in der Form, wie es in Anhang A festgelegt ist, die formale Syntax aller Elemente der Textsprachen definiert, die durch das System unterstützt werden;

j) muss imstande sein, Dateien zu lesen und zu schreiben, die Sprachelemente enthalten, die als Alternativen in der Erzeugung `library_element_declaration` in B.0 definiert sind, in der Syntax in der obigen Anforderung i) definiert sind oder nach der „ISO-646 IRV“ codiert sind, die als Tabelle 1 – Spalte 00 der ISO/IEC 10646-1 angeben ist.

Die Formulierung „muss imstande sein“ wird in diesem Abschnitt benutzt, um die Implementierung eines Software-Schalters zu erlauben, mit dem der Anwender die Meldung von Fehlern steuern kann.

In Fällen, in denen die Kompilation oder Programm-Eingabe wegen irgendwelcher Tabellen-Begrenzungen usw. abgebrochen wird, wird eine vollständige Angabe der Art „es wurden keine Verstöße entdeckt, die Prüfung ist jedoch unvollständig“, die Anforderungen dieses Abschnitts erfüllen.

1.5.2 Normerfüllung der Programme

Ein SPS-Programm, das die Anforderungen der IEC 61131-3 erfüllt:

a) darf nur Eigenschaften benutzen, die in diesem Teil zum Gebrauch der jeweiligen Sprache festgelegt sind;

b) darf keine Eigenschaften benutzen, die als Erweiterungen der Sprache gelten;

c) darf von keiner bestimmten Interpretation **implementierungsabhängiger Eigenschaften** abhängen.

Die Ergebnisse, die von einem normgerechten Programm erzeugt werden, müssen dieselben sein, wie von irgendeinem normgerechten System erzeugt, das die Merkmale unterstützt, die von dem Programm benutzt werden; diese Ergebnisse sind von der Programm-Ausführungszeit, der Anwendung von **implementierungsabhängigen Eigenschaften** (wie in Anhang D aufgeführt) im Programm und der Ausführung von Prozeduren zur Fehlerbehandlung beeinflusst.

2 Gemeinsame Elemente

Dieser Abschnitt definiert Text- und Grafikelemente, die für alle SPS-Programmiersprachen dieses Teils der IEC 61131 gemeinsam sind.

2.1 Gebrauch der gedruckten Zeichen

2.1.1 Zeichensatz

Die Textsprachen und die Textelemente der Grafiksprachen müssen nach der „ISO-646 IVR“ dargestellt werden, die als Tabelle 1 – Spalte 00 der ISO/IEC 10646-1 angegeben ist.

Der Gebrauch von Zeichen der zusätzlichen Zeichensätze, z. B. des „Latin-1 Supplement“, das als Tabelle 2 Spalte 00 des ISO/IEC 10646-1 angegeben ist, ist eine typische Erweiterung der Norm. Die Codierung solcher Zeichen muss mit der ISO/IEC 10646 übereinstimmen.

Der **geforderte Zeichensatz** besteht aus allen Zeichen in den Spalten 002 bis 007 der „ISO/IEC-646 IRV“, wie es oben definiert ist, ausgenommen der Kleinbuchstaben.

Tabelle 1 – Zeichensatz

Nr.	Beschreibung
2	Kleinbuchstaben [a]
3a	Nummernzeichen (#) `oder`
3b	Pfundzeichen (£)
4a	Dollarzeichen ($) `oder`
4b	Währungszeichen (¤)
5a	Senkrechter Strich (|) `oder`
5b	Ausrufungszeichen (!)
ANMERKUNG Die Nummerierung der Eigenschaften in dieser Tabelle ist so gestaltet, dass eine Übereinstimmung mit der ersten Ausgabe der IEC 61131-3 erhalten bleibt.	
[a] Wenn Kleinbuchstaben (Eigenschaft 2) unterstützt werden, darf die Groß-/Kleinschreibung in den Sprachelementen nicht signifikant sein, ausgenommen innerhalb Kommentaren wie in 2.1.5 definiert, Zeichenfolgen wie in 2.2.2 definiert und Variablen vom Typ `STRING` und `WSTRING` wie in 2.3.1 definiert.	

2.1.2 Bezeichner

Ein *Bezeichner* ist eine Folge von Buchstaben, Ziffern und Unterstrich-Zeichen, die mit einem Buchstaben oder Unterstrich-Zeichen beginnen muss.

Die Groß-/Kleinschreibung von Buchstaben darf in Bezeichnern nicht signifikant sein; z. B. müssen die Bezeichner `abcd`, `ABCD` und `aBCd` gleich interpretiert werden.

Unterstriche müssen in Bezeichnern signifikant sein; z. B. `A_BCD` und `AB_CD` müssen als unterschiedliche Bezeichner interpretiert werden. Mehrere führende und mehrfache Unterstriche sind nicht zulässig; z. B. sind die Zeichenfolgen `__LIM_SW5 und LIM__SW5` keine gültigen Bezeichner. Angehängte Unterstriche sind nicht zulässig; z. B. ist die Zeichenfolge `LIM_SW5_` kein gültiger Bezeichner.

Mindestens sechs eindeutige Zeichen müssen in allen Systemen unterstützt werden, die den Gebrauch von Bezeichnern bieten; z. B. muss `ABCDE1` als unterschiedlich zu `ABCDE2` in allen diesen Systemen interpretiert werden. Die maximale Anzahl von Zeichen, die in einem Bezeichner zulässig ist, ist ein **implementierungsabhängiger** Parameter.

Eigenschaften und Beispiele werden in Tabelle 2 gezeigt.

Tabelle 2 – Bezeichner-Eigenschaften

Nr.	Beschreibung	Beispiele
1	Großbuchstaben und Zahlen	`IW215 IW215Z QX75 IDENT`
2	Groß- und Kleinbuchstaben, Zahlen, eingebettete Unterstriche	Alle oben aufgeführten plus: `LIM_SW_5 LimSw5 abcd ab_Cd`
3	Groß- und Kleinbuchstaben, Zahlen, führende und eingebettete Unterstriche	Alle oben aufgeführte plus: `_MAIN _12V7`

2.1.3 Schlüsselwörter

Schlüsselwörter sind eindeutige Kombinationen von Zeichen, die als individuelle syntaktische Elemente angewendet werden, wie dies in Anhang B definiert ist. Alle Schlüsselwörter, die in dieser Norm benutzt werden, sind in Anhang C aufgelistet. Schlüsselwörter dürfen keine eingebetteten Leerzeichen enthalten. Die Groß-/Kleinschreibung der Zeichen darf in Schlüsselwörtern nicht signifikant sein, z. B. sind die Schlüsselwörter „FOR“ und „for“ syntaktisch gleichbedeutend. Die Schlüsselwörter, die im Anhang C aufgelistet sind, dürfen für keinen anderen Zweck genutzt werden, z. B. als Variablennamen oder Erweiterungen wie in 1.5.1 definiert.

ANMERKUNG Nationale Normungsorganisationen dürfen Übersetzungstabellen der Schlüsselwörter aus Anhang C herausgeben.

2.1.4 Gebrauch von Leerzeichen

Dem Anwender muss erlaubt sein, ein oder mehrere Zeichen als „Leerzeichen“ überall im Text des SPS-Programms einzufügen, ausgenommen innerhalb von Schlüsselwörtern, Literalen, Aufzählungswerten, Bezeichnern, direkt-dargestellten Variablen, wie in 2.4.1.1 beschrieben, oder Kombinationen von Begrenzern (z. B. für Kommentare, wie es in 2.1.5 definiert ist). „Leerzeichen“ ist sowohl als SPACE Zeichen mit dem Code-Wert 32 dezimal als auch als nicht-druckbare Zeichen wie Tabulator, Neue-Zeile usw., für die keine Kodierung in ISO/IEC 10646-1 angegeben sind, definiert.

2.1.5 Kommentare

Anwender-Kommentare müssen an Anfang und Ende mit einer besonderen Zeichenkombination „(*“bzw.“*)“ begrenzt werden, wie in Tabelle 3 gezeigt ist. Kommentare müssen überall im Programm zulässig sein, wo Leerzeichen erlaubt sind; ausgenommen innerhalb Zeichenketten-Literalen, die in 2.2.2 definiert sind. Kommentare dürfen keine syntaktische oder semantische Bedeutung für eine der in diesem Teil definierten Sprachen haben.

Der Gebrauch von geschachtelten Kommentaren, z. B. `(*(* GESCHACHTELT *)*)`, muss als ein **Fehler** nach den Regeln von 1.5.1 d) behandelt werden.

Die maximale Anzahl der Zeichen, die in einem Kommentar zulässig ist, ist ein **implementierungsabhängiger Parameter**.

Tabelle 3 – Kommentar-Eigenschaft

Nr.	Beschreibung	Beispiel

1	Kommentare	`(*********************************)`
		`(*  Ein eingerahmter Kommentar  *)`
		`(*********************************)`

ANMERKUNG Das Beispiel oben enthält drei einzelne Kommentare.		

2.1.6 Pragma

Wie in Tabelle 3a veranschaulicht, müssen Pragmas am Beginn und Ende durch geschweifte Klammern „{“ bzw. “}“ eingeschlossen werden. Die Syntax und Semantik von speziellen Pragma-Konstruktionen sind Implementierungsabhängig. Pragmas müssen überall in Programmen zulässig sein, wo Leerzeichen erlaubt sind, ausgenommen innerhalb von Zeichen folgen-Literalen, wie in 2.2.2 definiert.

ANMERKUNG Geschweifte Klammern innerhalb eines Kommentars haben keine semantische Bedeutung; Kommentare innerhalb geschweifter Klammern dürfen abhängig von der Implementierung eine semantische Bedeutung haben oder auch nicht.

Tabelle 3a – Pragma-Eigenschaften

Nr.	Beschreibung der Eigenschaften	Beispiele
1	Pragmas	{VERSION 3.1} {AUTHOR JHC} {x := 256, y := 384}

2.2 Externe Darstellung von Daten

Externe Darstellungen von Daten bei den verschiedenen SPS-Programmiersprachen müssen die numerischen Literale, Zeichenfolge und Zeitliterale umfassen.

2.2.1 Numerische Literale

Es gibt zwei Arten von numerischen Literalen: die ganzzahligen (integer) Literale und die reelen (real) Literale. Ein numerisches Literal ist als eine Dezimalzahl oder eine basisbezogene Zahl definiert. Die maximale Anzahl von Ziffern für alle Arten von numerischen Literalen muss ausreichend sein, um den gesamten Bereich und die Genauigkeit der Werte aller Datentypen auszudrücken, die durch das Literal in einer gegebenen Implementierung dargestellt werden.

Einzelne Unterstrich-Zeichen (_), die zwischen den Ziffern eines numerischen Literals eingefügt sind, dürfen nicht signifikant sein. Eine andere Anwendung von Unterstrich-Zeichen in numerischen Literalen ist nicht zulässig.

Dezimale Literale müssen in herkömmlicher Dezimal-Schreibweise dargestellt werden. Reelle Literale müssen durch die Angabe eines Dezimalpunktes gekennzeichnet werden. Ein Exponent gibt die Zehnerpotenz an, mit der die vorausgehende Zahl zu multiplizieren ist, um den dargestellten Wert zu erhalten. Dezimale Literale und ihre Exponenten können ein vorangestelltes Vorzeichen (+ oder -) haben.

Ganzzahlige Literale können auch zur Basis 2, 8 oder 16 dargestellt werden. Die Basis muss in dezimaler Angabe erfolgen. Für die Basis 16 muss ein erweiterter Satz von Zeichen benutzt werden, der aus den Buchstaben A bis F besteht, die die übliche Bedeutung der Dezimalen 10 bis 15 haben. Basisbezogene Zahlen dürfen kein Vorzeichen (+ oder -) beinhalten.

Boolesche Daten müssen durch ganzzahlige Literale mit dem Wert Null (`0`) oder Eins (`1`) oder durch die Schlüsselwörter `FALSE` oder `TRUE` dargestellt werden.

Die Eigenschaften und Beispiele für numerische Literale sind in Tabelle 4 gezeigt.

Der *Datentyp* von boolschen oder numerischen Literalen darf durch Voranstellen eines Datentyp-Präfixes festgelegt werden und erfolgt durch den Namen des Datentyps und einem „#“-Zeichen. Für Beispiele vergleiche Eigenschaft 9 in der Tabelle 4.

Tabelle 4 – Numerische Literale

Nr.	Beschreibung	Beispiele
1	Ganzzahlige (integer) Literale	`-12  0  123_456  +986`
2	Reele (real) Literale	`-12.0  0.0  0.456  3.14159_26`

3	Reele (real) Literale mit Exponenten	`-1.34E-12` oder `-1.34e-12`
		`1.0E+6` oder `1.0e+6`
		`1.234E6` oder `1.234e6`

4	Basis-2 Literale	`2#1111_1111` (`255` dezimal) `2#1110_0000` (`224` dezimal)
5	Basis-8 Literale	`8#377` (`255` dezimal) `8#340` (`224` dezimal)
6	Basis-16 Literale	`16#FF` oder `16#ff` (`255` dezimal) `16#E0` oder `16#e0` (`224` dezimal)
7	Boolesche Null und Eins	`0  1`
8	Boolesche `FALSE` und `TRUE`	`FALSE  TRUE`
9	Literale mit Typangabe	`DINT#5` (`DINT` Darstellung von `5`) `UINT#16#9AF` (`UINT Darstellung des hexadezimalen Werts 9AF`) `BOOL#0  BOOL#1  BOOL#TRUE  BOOL#FALSE`
ANMERKUNG Die Schlüsselwörter `FALSE` and `TRUE` korrespondieren mit den booleschen Werten `0` und `1`.		

2.2.2 Zeichenfolge-Literale

Zeichenfolge-Literale sind entweder Einzel-Byte- oder Doppel-Byte-Zeichen.

Ein Zeichenfolge-Literal, das ein Einzel-Byte-Zeichen ist, ist eine Sequenz von null oder mehr Zeichen aus der Reihe 00 des ISO/IEC 10646-1 Zeichensatzes, die durch Einzel-Anführungszeichen (') eingeleitet und abgeschlossen ist. In Einzel-Byte-Zeichenfolgen muss die 3-Zeichen-Kombination, bestehend aus einem Dollar-Zeichen ($) und gefolgt von zwei hexadezimalen Ziffern, als die hexadezimale Darstellung des 8-Bit-Zeichencode interpretiert werden. Dies ist in Eigenschaft 1 von Tabelle 5 gezeigt.

Ein Zeichenfolge-Literal, das ein Doppel-Byte-Zeichen ist, ist eine Sequenz von null oder mehr Zeichen aus dem ISO/IEC 10646-1 Zeichensatz, die durch Doppel-Anführungszeichen (“) eingeleitet und abgeschlossen ist. In Doppel-Byte-Zeichenfolgen muss die 5-Zeichen-Kombination, bestehend aus einem Dollar-Zeichen ($) und gefolgt von vier hexadezimalen Ziffern, als die hexadezimale Darstellung des 16-Bit-Zeichencode interpretiert werden. Dies ist in Eigenschaft 2 von Tabelle 5 gezeigt.

2-Zeichen-Kombinationen, die mit dem Dollar-Zeichen beginnen, müssen interpretiert werden, wie es in Tabelle 6 gezeigt ist, wenn sie in Zeichenfolgen auftreten.

Tabelle 5 – Zeichenfolge Literal

Nr.	Beispiel	Erläuterung
1	**Einzel-Byte-Zeichenfolge**	
	`''`	Leere Zeichenfolge (Länge null)
	`'A'`	Zeichenfolge der Länge eins mit dem einzelnen Zeichen A
	`' '`	Zeichenfolge der Länge eins mit „Leerzeichen“
	`'$''`	Zeichenfolge der Länge eins mit „Einzel-Anführungszeichen“
	`'"'`	Zeichenfolge der Länge eins mit „Doppel-Anführungszeichen“
	`'$R$L'`	Zeichenfolge der Länge zwei mit CR und LF Zeichen
	`'$0A'`	Zeichenfolge der Länge zwei mit LF Zeichen
	`'$$1.00'`	Zeichenfolge der Länge fünf, die als „$1.00“ gedruckt wird
	`'ÄË'` `'$C4$CB'`	Gleiche Zeichenfolgen der Länge 2
2	**Doppel-Byte-Zeichenfolge**	
	`""`	Leere Zeichenfolge (Länge null)
	`"A"`	Zeichenfolge der Länge eins mit dem einzelnen Zeichen A
	`" "`	Zeichenfolge der Länge eins mit „Leerzeichen“
	`" ' "`	Zeichenfolge der Länge eins mit „Einzel-Anführungszeichen“
	`"$" "`	Zeichenfolge der Länge eins mit „Doppel-Anführungszeichen“
	`"$R$L"`	Zeichenfolge der Länge zwei mit LF Zeichen
	`"$$1.00"`	Zeichenfolge der Länge fünf, die als „$1.00“ gedruckt wird
	`"ÄË"` `"$00C4$00CB"`	Gleiche Zeichenfolgen der Länge 2
3	**Einzel-Byte-Zeichenfolge-Literal mit Typangabe**	
	`STRING#'OK'`	Zeichenfolge der Länge zwei mit zwei Einzel-Byte-Zeichen
4	**Doppel-Byte Zeichenfolge-Literal mit Typangabe**	
	`WSTRING#'OK'`	Zeichenfolge der Länge zwei mit zwei Doppel-Byte-Zeichen

ANMERKUNG Wenn eine bestimmte Implementierung die Eigenschaft Nr. 4 unterstützt, jedoch nicht die Eigenschaft Nr. 2 unterstützt, darf der **Implementierer eine herstellerspezifische** Syntax und Semantik für den Gebrauch der Doppel-Anführungszeichen festlegen.

Tabelle 6 – Zwei-Zeichen-Kombinationen in Zeichenfolgen

Nr.	Kombination	Bedeutung beim Ausdruck
2	`$$`	Dollar-Zeichen
3	`$'`	Einfaches Anführungszeichen
4	`$L` oder `$l`	Zeilenvorschub (Line feed)
5	`$N` oder `$n`	Neue Zeile (New line)
6	`$P` oder `$p`	Neue Seite (Form feed, page)
7	`$R` oder `$r`	Wagenrücklauf (Carriage return)
8	`$T` oder `$t`	Tabulator (Tab)
9	`$"`	Doppel-Anführungszeichen

ANMERKUNG 1 Das Zeichen „neue Zeile" bietet ein implementierungsabhängiges Mittel, um das Ende einer Datenzeile zu definieren. Dies gilt sowohl bei physikalischen als auch bei Datei-Ein/Ausgaben. Beim Drucken bewirkt es das Beenden einer Datenzeile und das Fortsetzen der nächsten Zeile.

ANMERKUNG 2 Die Kombination `$'` ist nur innerhalb von Zeichenfolge-Literalen in Einzel-Anführungszeichen zulässig.

ANMERKUNG 3 Die Kombination `$"` ist nur innerhalb von Zeichenfolge-Literalen in Doppel-Anführungszeichen zulässig.

2.2.3 Zeitliterale

Es ist notwendig, externe Darstellungen für zwei unterschiedliche Typen von Zeitdaten zur Verfügung zu stellen: Daten für die *Zeitdauer* zur Messung oder Steuerung der verstrichenen Zeit eines Steuerungsereignisses und Daten für die *Tageszeit* (die auch Datumsinformationen beinhalten können) zur Synchronisation von Beginn oder Ende eines Steuerungsereignisses mit einer absoluten Zeitangabe.

Die Literale für Zeitdauer und Tageszeit müssen links durch die Schlüsselwörter, die in 2.2.3.1 und 2.2.3.2 definiert sind, begrenzt werden.

2.2.3.1 Zeitdauer

Die Daten für Zeitdauer müssen links durch das Schlüsselwort `T#, TIME#` begrenzt werden. Die Darstellung von Daten für Zeitdauer, wie Tage, Stunden, Minuten, Sekunden und Millisekunden oder alle Kombinationen hiervon, muss in der Weise erfolgen, wie es in Tabelle 7 gezeigt ist. Die niedrigstwertige Zeiteinheit kann als reelle Zahl ohne Exponent geschrieben werden.

Die Einheiten von Zeitdauer-Literalen können durch Unterstrich-Zeichen getrennt werden.

Der „Überlauf" der höchstwertigen Einheit eines Zeitdauer-Literals ist zulässig; z. B. ist die Angabe `T#25h_15m` erlaubt.

Zeiteinheiten, wie z. B. Sekunden, Millisekunden usw., können in Groß- oder Kleinbuchstaben dargestellt werden.

Wie in Tabelle 7 veranschaulicht, sind sowohl positive wie auch negative Werte für die Zeitdauerangaben zulässig.

Tabelle 7 – Zeitdauer Literal

Nr.	Beschreibung	Beispiele
1a	Zeitdauer ohne Unterstriche: kurzes Präfix	`T#14ms T#-14ms T#14.7s T#14.7m T#14.7h t#14.7d t#25h15m t#5d14h12m18s3.5ms`
1b	langes Präfix	`TIME#14ms TIME#-14ms time#14.7s`
2a	Zeitdauer mit Unterstrichen: kurzes Präfix	`t#25h_15m t#5d_14h_12m_18s_3.5ms`
2b	langes Präfix	`TIME#25h_15m time#5d_14h_12m_18s_3.5ms`

2.2.3.2 Tageszeit und Datum

Die vorangestellten Schlüsselwörter für die Literale Tageszeit und Datum sind in Tabelle 8 gezeigt. Wie in Tabelle 9 veranschaulicht, muss die Darstellung der Information von Tageszeit und Datum so erfolgen, wie es durch die Syntax festgelegt ist, die in B.1.2.3.2 angegeben ist.

Tabelle 8 – Literale für Datum und Tageszeit

Nr.	Beschreibung	Schlüsselwort
1	Literale für Datum (langes Präfix)	`DATE#`
2	Literale für Datum (kurzes Präfix)	`D#`
3	Literale für Tageszeit (langes Präfix)	`TIME_OF_DAY#`
4	Literale für Tageszeit (kurzes Präfix)	`TOD#`
5	Literale für Datum und Zeit (langes Präfix)	`DATE_AND_TIME#`
6	Literale für Datum und Zeit (kurzes Präfix)	`DT#`

Tabelle 9 – Beispiele für Datum und Tageszeit

In langer Präfix-Schreibweise	In kurzer Präfix-Schreibweise
`DATE#1984-06-25` `date#1984-06-25`	`D#1984-06-25` `d#1984-06-25`
`TIME_OF_DAY#15:36:55.36` `time_of_day#15:36:55.36`	`TOD#15:36:55.36` `tod#15:36:55.36`
`DATE_AND_TIME#1984-06-25-15:36:55.36` `date_and_time#1984-06-25-15:36:55.36`	`DT#1984-06-25-15:36:55.36` `dt#1984-06-25-15:36:55.36`

2.3 Datentypen

Durch diese Norm wird eine Anzahl von elementaren (vordefinierten) Datentypen zugelassen. Zusätzlich werden allgemeine (generische) Datentypen zur Anwendung bei der Definition von überladenen (typunabhängigen) Funktionen (siehe 2.5.1.4) definiert. Außerdem ist ein Verfahren für den Anwender oder den Hersteller definiert, um zusätzliche Datentypen festzulegen.

2.3.1 Elementare Datentypen

Die elementaren Datentypen, das Schlüsselwort für jeden Datentyp, die Anzahl der Bits pro Datenelement und der Wertebereich für jeden elementaren Datentyp müssen der Tabelle 10 entsprechen.

Tabelle 10 – Elementare Datentypen

Nr.	Schlüsselwort	Datentyp	N [a]
1	BOOL	boolesche	1 [h]
2	SINT	kurze ganze Zahl (short integer)	8 [c]
3	INT	ganze Zahl (integer)	16 [c]
4	DINT	doppelte ganze Zahl (double integer)	32 [c]
5	LINT	lange ganze Zahl (long integer)	64 [c]
6	USINT	vorzeichenlose kurze ganze Zahl (unsigned short integer)	8 [d]
7	UINT	vorzeichenlose ganze Zahl	16 [d]
8	UDINT	vorzeichenlose doppelte ganze Zahl	32 [d]
9	ULINT	vorzeichenlose lange ganze Zahl	64 [d]
10	REAL	reelle Zahl	32 [e]
11	LREAL	lange reelle Zahl	64 [f]
12	TIME	Zeitdauer	-- [b]
13	DATE	Datum (nur)	-- [b]
14	TIME_OF_DAY oder TOD	Uhrzeit (nur)	-- [b]
15	DATE_AND_TIME oder DT	Datum und Uhrzeit	-- [b]
16	STRING	variabel-lange Zeichenfolge	8 [i, g]
17	BYTE	Bit-Folge 8	8 [J, g]
18	WORD	Bit-Folge 16	16 [J, g]
19	DWORD	Bit-Folge 32	32 [J, g]
20	LWORD	Bit-Folge 64	64 [J, g]
21	WSTRING	Variable lange Doppel-Byte-Zeichenfolge	16 [i, g]

[a] Einträge in dieser Spalte müssen so interpretiert werden, wie es in den Fußnoten festgelegt ist.

[b] Der Wertebereich und die Genauigkeit der Darstellung in diesen Datentypen ist **implementierungs-abhängig**.

[c] Der Wertebereich für Variablen dieses Datentyps reicht von $-(2^{N-1})$ bis $(2^{N-1})-1$.

[d] Der Wertebereich für Variablen dieses Datentyps reicht von 0 bis $(2^{N})-1$.

[e] Der Wertebereich für Variablen dieses Datentyps muss festgelegt sein, wie es in IEC 60559 für das Basis-Gleitpunkt-Format mit einfacher Länge definiert ist.

[f] Der Wertebereich für Variablen dieses Datentyps muss festgelegt sein, wie es in IEC 60559 für das Basis-Gleitpunkt-Format mit doppelte Länge definiert ist.

[g] Ein numerischer Wertebereich ist für diesen Datentyp nicht anwendbar .

[h] Die möglichen Werte für Variablen dieses Datentyps müssen 0 und 1 sein, entsprechend den Schlüsselwörtern FALSE und TRUE.

[i] Der Wert N gibt die Anzahl der Bits/Zeichen für diesen Datentyp an.

[J] Der Wert N gibt die Anzahl der Bits in der Bitfolge für diesen Datentyp an.

2.3.2 Allgemeine (generische) Datentypen

Zusätzlich zu den Datentypen, die in Tabelle 10 gezeigt sind, kann die Hierarchie der allgemeinen Datentypen, die in Tabelle 11 gezeigt ist, bei der Festlegung von Eingängen und Ausgängen von Standardfunktionen und Funktionsbausteinen angewendet werden (siehe 2.5.1.4). Die allgemeinen Datentypen werden durch die Vorsilbe „ANY“ identifiziert. Der Gebrauch von allgemeinen Datentypen ist in den folgenden Regeln beschrieben:

1) Allgemeine Datentypen dürfen nicht in anwender-deklarierten Programm-Organisationseinheiten, die in 2.5 definiert sind, verwendet werden.
2) Der allgemeine Typ eines abgeleiteten *Unterbereichstyps* (Merkmal 3 von Tabelle 12) muss ANY_INT sein.
3) Der allgemeine Typ eines *direkt abgeleiteten* Typs (Eigenschaft 1 der Tabelle 12) muss derselbe sein wie der elementare Typ, von dem er abgeleitet ist.
4) Der allgemeine Typ von allen anderen abgeleiteten Typen, die in Tabelle 12 definiert sind, muss ANY_DERIVED sein.

Tabelle 11 – Hierarchie der allgemeinen Datentypen

```
ANY
        ANY_DERIVED (Abgeleitete Datentypen – siehe vorausgegangener Text)
        ANY_ELEMENTARY
           ANY_MAGNITUDE
               ANY_NUM
                   ANY_REAL
                       LREAL
                       REAL
                   ANY_INT
                       LINT, DINT, INT, ULINT, SINT
                       ULDINT, UDINT, UINT, USINT
               TIME
           ANY_BIT
               LWORD, DWORD, WORD, BYTE, BOOL
           ANY_STRING
               STRING
               WSTRING
           ANY_DATE
               DATE_AND_TIME
               DATE, TIME_OF_DAY
```

2.3.3 Abgeleitete Datentypen

2.3.3.1 Deklarationen (Vereinbarungen)

Abgeleitete (d. h. anwender- oder herstellerdefinierte) Datentypen können durch die Anwendung der textuelen Konstruktion TYPE...END_TYPE, die in Tabelle 12 gezeigt ist, deklariert werden. Diese abgeleiteten Datentypen können dann zusätzlich zu den elementaren Datentypen, die in 2.3.1 definiert sind, in Deklarationen von Variablen angewendet werden, wie in 2.4.3 gezeigt ist.

Eine Deklaration für den Datentyp *Aufzählung* legt fest, dass der Wert jedes Datenelements dieses Typs nur einen der Werte annehmen kann, die in der zugehörigen Liste der Bezeichner angegeben sind, wie in Tabelle 12 veranschaulicht ist. Die Aufzählungsliste definiert eine geordnete Menge von aufgezählten Werten, beginnend mit dem ersten Bezeichner der Liste und endend mit dem letzten. Verschiedene Aufzählungstypen dürfen die gleichen Bezeichner für aufgezählte Werte verwenden. Die maximal zulässige Anzahl der aufgezählten Werte ist ein **implementierungsabhängiger Parameter**.

Um eine eindeutige Identifikation in einem bestimmten Kontext zu gewährleisten, dürfen die aufgezählten Literale durch ein Präfix beschrieben werden, das aus dem zugehörigen Datentypnamen und den '#'-Zeichen besteht, so wie die Literale mit Typangabe, die in 2.2.1 definiert sind. Ein derartiges Präfix darf nicht innerhalb einer Aufzählungsliste verwendet werden. Es ist ein **Fehler**, wenn für ein aufgezähltes Literal nicht ausreichend Information zur Verfügung gestellt wird, um seinen Wert eindeutig zu bestimmen.

Eine Deklaration für den *Unterbereich* legt fest, dass der Wert jedes Datenelement dieses Typs nur Werte annehmen kann, die innerhalb und einschließlich der festgelegten Ober- und Untergrenze liegen, wie dies in Tabelle 12 angegeben ist. Es ist ein **Fehler**, wenn der Wert eines Unterbereichstyps außerhalb der festgelegten Bereichswerte liegt.

Eine Deklaration für Struktur (STRUCT) legt fest, dass Datenelemente dieses Typs Unterelemente von festgelegten Typen enthalten müssen, auf die mit festgelegten Namen zugegriffen werden kann. Z. B. enthält ein Element des Typs ANALOG_CHANNEL_CONFIGURATION, wie in Tabelle 12 deklariert, ein Unterelement RANGE vom Typ ANALOG_SIGNAL_RANGE, ein Unterelement MIN_SCALE vom Typ ANALOG_DATA und ein Element MAX_SCALE vom Typ ANALOG_DATA. Die maximale Anzahl von Strukturelementen, die maximale Menge von Daten, die in einer Struktur einhalten sein kann, und die maximale Anzahl von geschachtelten Ebenen von zugehörigen Strukturelementen sind **implementierungsabhängige** Parameter.

Eine Deklaration für Feld (ARRAY) legt fest, dass eine ausreichende Menge von Datenspeicher für jedes Element dieses Typs zugewiesen werden muss, um alle die Daten zu speichern, auf die mit dem (den) festgelegten Index-Unterbereich(en) gezeigt werden kann. So muss jedes Element vom Typ ANALOG_16_INPUT_CONFIGURATION, wie in Tabelle 12 gezeigt, ausreichend Speicherplatz für 16 Elemente CHANNEL vom Typ ANALOG_CHANNEL_CONFIGURATION (neben weiteren Elementen) enthalten. Mechanismen für den Zugriff auf Feldelemente sind in 2.4.1.2 definiert. Die maximale Anzahl der Feldindizes, die maximale Feldgröße und der maximale Bereich von Index-Werten sind **implementierungsabhängige** Parameter.

2.3.3.2 Initialisierung

Der voreingestellte Wert eines Datentyps für *Aufzählung* muss der erste Bezeichner in der zugehörigen Aufzählungsliste sein oder ein Wert, der durch den Zuweisungsoperator „:=“ festgelegt ist. Z. B. haben, wie in den Tabellen 12, Nr. 2 und 14, Nr. 2 gezeigt, die voreingestellten Anfangswerte der Elemente der Datentypen ANALOG_SIGNAL_TYPE und ANALOG_SIGNAL_RANGE die Werte SINGLE_ENDED bzw. UNIPOLAR_1_5V.

Für Datentypen mit *Unterbereichen* muss der voreingestellte Anfangswert die erste (untere) Grenze des Unterbereichs sein, falls dies nicht anders durch einen Zuweisungsoperator festgelegt ist. Wie in Tabelle 12 deklariert, ist z. B. der voreingestellte Anfangswert der Elemente vom Typ ANALOG_DATEN gleich -4095, während der voreingestellte Anfangswert für das Unterelement FILTER_PARAMETER der Elemente vom Typ ANALOG_16_INPUT_CONFIGURATION gleich null ist. Im Gegensatz dazu ist der voreingestellte Anfangswert der Elemente vom Typ ANALOG_DATAZ, wie in Tabelle 14 deklariert, gleich Null.

Für weitere abgeleitete Datentypen müssen die voreingestellten Anfangswerte gleich denen der unterlagerten Datentypen sein, wie es in Tabelle 13 definiert ist; falls sie nicht anders durch die Anwendung des Zuweisungsoperators „:=“ in der TYP-Deklaration festgelegt sind. Weitere Beispiele für die Anwendung des Zuweisungsoperators bei der Initialisierung sind in 2.4.2 angegeben.

Die voreingestellte maximale Länge von Elementen des Typs STRING und WSTRING muss ein **implementierungsabhängiger** Wert sein, falls sie nicht anders durch eine in Klammern angegebene Maximallänge in der zugehörigen Deklaration festgelegt ist. (Diese Länge darf die implementierungsabhängige Voreinstellung nicht überschreiten.) Wenn z. B. der Typ STR10 wie folgt deklariert ist

```
TYPE STR10 : STRING[10] := 'ABCDEF' ; END_TYPE
```

dann ist die Maximallänge, der voreingestellte Anfangswert und die voreingestellte Anfangslänge von Datenelementen des Typs STR10 gleich 10 Zeichen, 'ABCDEF' bzw. 6 Zeichen. Die maximal zulässige Länge von STRING und WSTRING ist ein **implementierungsabhängiger Parameter**.

Tabelle 12 – Eigenschaften der Datentyp-Deklarationen

Nr.	Eigenschaft/Beispiel in Textform
1	Direkte Ableitung von elementaren Typen, z. B.: `TYPE RU_REAL : REAL ; END_TYPE`
2	Datentypen für Aufzählung, z. B.: `TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ; END_TYPE`
3	Datentypen für Bereich, z. B.: `TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE`
4	Datentypen für Feld, z. B.: `TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ; END_TYPE`

5	Datentypen für Strukturen, z. B.:
	`TYPE`
	`  ANALOG_CHANNEL_CONFIGURATION :`
	`  STRUCT`
	`  RANGE : ANALOG_SIGNAL_RANGE ;`
	`  MIN_SCALE : ANALOG_DATA ;`
	`  MAX_SCALE : ANALOG_DATA ;`
	`  END_STRUCT ;`
	`  ANALOG_16_INPUT_CONFIGURATION :`
	`  STRUCT`
	`  SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ;`
	`  FILTER_PARAMETER : SINT (0..99) ;`
	`  CHANNEL : ARRAY [1..16] OF ANALOG_CHANNEL_CONFIGURATION ;`
	`  END_STRUCT ;`
	`END_TYPE`

ANMERKUNG Beispiele für den Gebrauch dieser Typen in Deklarationen von Variablen siehe 2.3.3.3, 2.4.1.2 und Tabelle 17.	

Tabelle 13 – Voreingestellte Anfangswerte

Datentypen	Initialisierungswert
BOOL, SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0S
DATE	D#0001-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#0001-01-01-00:00:00
STRING	' ' (die leere Zeichenfolge)
WSTRING	“ “ (die leere Zeichenfolge)

Tabelle 14 – Eigenschaften der Deklaration der Anfangswerte von Datentypen

Nr.	Eigenschaft/Beispiel in Textform
1	Initialisierung von direkt abgeleiteten Typen, z. B.: `TYPE FREQ : REAL := 50.0 ; END_TYPE`

2	Initialisierung von Datentypen für Aufzählung, z. B.:
	`TYPE ANALOG_SIGNAL_RANGE :`
	`(BIPOLAR_10V, (* -10 to +10 VDC *)`
	`UNIPOLAR_10V, (* 0 to +10 VDC *)`
	`UNIPOLAR_10V, (* 0 to +10 VDC *)`
	`UNIPOLAR_1_5V, (* + 1 to + 5 VDC *)`
	`UNIPOLAR_0_5V, (* 0 to + 5 VDC *)`
	`UNIPOLAR_4_20_MA, (* +4 to +20 mADC *)`
	`UNIPOLAR_0_20_MA (* 0 to +20 mADC *)`
	`) := UNIPOLAR_1_5V ;`
	`END_TYPE`

3	Initialisierung von Datentypen für Bereich, z. B.: `TYPE ANALOG_DATAZ : INT (-4095..4095) := 0 ; END_TYPE`

4	Initialisierung von Datentypen für Feld, z. B.:
	`TYPE ANALOG_16_INPUT_DATAI :`
	`ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)] ;`
	`END_TYPE`

5	Initialisierung von Datentypen für Struktur, z. B.:
	`TYPE ANALOG_CHANNEL_CONFIGURATIONI :`
	`STRUCT`
	`RANGE : ANALOG_SIGNAL_RANGE ;`
	`MIN_SCALE : ANALOG_DATA := -4095 ;`
	`MAX_SCALE : ANALOG_DATA := 4095 ;`
	`END_STRUCT ;`
	`END_TYPE`

6	Initialisierung von Datentypen für abgeleitete Struktur, z. B.:
	`TYPE ANALOG_CHANNEL_CONFIGZ :`
	`ANALOG_CHANNEL_CONFIGURATIONI`
	`:= (MIN_SCALE := 0, MAX_SCALE := 4000);`
	`END_TYPE`

2.3.3.3 Gebrauch

Der Gebrauch von Variablen, die als abgeleitete Datentypen deklariert sind (wie in 2.4.3.1 definiert ist), muss folgenden Regeln folgen:

1) Eine Einzelelement-Variable, wie in 2.4.1.1 definiert, mit einem abgeleiteten Typ kann überall benutzt werden, wo eine Variable vom Typ ihrer „Eltern“ benutzt werden kann; z. B. können Variable mit den Typen `RU_REAL` und `FREQ`, wie in den Tabellen 12 und 14 gezeigt, überall benutzt werden, wo eine Variable vom Typ `REAL` benutzt werden kann und Variable mit dem Typ `ANALOG_DATA` können überall benutzt werden, wo eine Variable vom Typ `INT` benutzt werden kann.

   Diese Regel kann rekursiv angewendet werden. Sind z. B. die folgenden Deklarationen gegeben, dann kann die Variable `R3` vom Typ `R2` überall angewendet werden, wo eine Variable vom Typ `REAL` angewendet werden kann:

```
TYPE R1  : REAL := 1.0 ; END_TYPE
TYPE R2  : R1;           END_TYPE
VAR  R3  : R2;           END_VAR
```

2) Ein Element einer Mehrelement-Variablen, wie in 2.4.1.2 definiert, kann überall benutzt werden, wo der „Elterntyp“ benutzt werden kann. Sind z. B. die Deklarationen von `ANALOG_16_INPUT_DATA` aus der Tabelle 12 und die Deklaration

   ```
   VAR INS : ANALOG_16_INPUT_DATA ; END_VAR
   ```

   gegeben, dann können die Variablen `INS [1]` bis `INS [16]` überall benutzt werden, wo eine Variable vom Typ `INT` benutzt werden könnte.

   Diese Regel kann auch rekursiv angewendet werden; sind z. B. die Deklarationen `ANALOG_16_INPUT_CONFIGURATION`, `ANALOG_CHANNEL_CONFIGURATION` und `ANALOG_DATA` aus Tabelle 12 und die Deklaration

   ```
   VAR CONF : ANALOG_16_INPUT_CONFIGURATION ; END_VAR
   ```

   gegeben, dann kann die Variable `CONF.CHANNEL [2].MIN_SCALE` überall benutzt werden, wo eine Variable vom Typ `INT` benutzt werden könnte.

2.4 Variablen

Im Gegensatz zu den externen Darstellungen von Daten, wie in 2.2 beschrieben, liefern *Variablen* die Mittel zur Identifizierung von Datenobjekten, deren Inhalt sich ändern darf; z. B. Daten, die Eingängen, Ausgängen oder Speicherplatz der SPS zugeordnet sind. Eine Variable kann als eine der elementaren Typen, die in 2.3.1 definiert sind, deklariert werden, oder als eine der abgeleiteten Typen, wie in 2.3.3.1 definiert.

2.4.1 Darstellung

2.4.1.1 Einzelelement-Variablen

Eine *Einzelelement*-Variable ist als eine Variable definiert, die ein einzelnes Datenelement von einem der elementaren Typen, wie in 2.3.1 definiert, darstellt, oder die einen abgeleiteten Typ für Aufzählung oder Unterbereich, wie in 2.3.3.1 definiert, darstellt, oder die einen abgeleiteten Typ, dessen „Elternschaft“, wie in 2.3.3.3 rekursiv definiert, darstellt, der zu einem elementaren Aufzählungs- oder Unterbereichstyp zurückverfolgt werden kann. Dieser Abschnitt definiert die Mittel zur Darstellung derartiger Variablen auf *symbolische* Art oder alternativ in einer Weise, die *direkt* die Zuordnung des Datenelements zu den physikalischen oder logischen Speicherorten in der SPS, nämlich Eingang, Ausgang und Speicherstruktur, darstellt.

Die Bezeichner, wie sie in 2.1.2 definiert sind, müssen zur symbolischen Darstellung von Variablen angewendet werden.

Die direkte Darstellung einer Einzelelement-Variablen muss durch ein besonderes Symbol ausgedrückt werden, welches durch folgende Aneinanderreihung gebildet wird: Das Prozentzeichen „%“ (Zeichencode 037 dezimal in Tabelle 1 – Reihe 00 von ISOI/IEC 10646-1), ein *Präfix für den Speicherort* und ein *Präfix für die Größe* nach Tabelle 15 und eine oder mehrere vorzeichenlose ganze Zahlen, die durch Punkte (.) getrennt sind.

Im dem Fall, dass eine direkt dargestellte Variable in einer Speicherortzuweisung zu einer internen Variablen im Deklarationsteil eines *Programm*- oder eines *Funktionsbausteintyps*, wie in 2.4.3.1 definiert, verwendet wird, muss ein Stern (Asterisk) „*“ anstelle des Präfix für die Größe und eine oder mehrere vorzeichenlose ganze Zahlen in der Aneinanderreihung verwendet werden, um anzuzeigen, dass die direkte Darstellung noch nicht vollständig spezifiziert ist. Das Prozentzeichen und das Präfix für den Speicherort `I`, `Q` oder `M` aus Tabelle 15 muss immer in der direkten Darstellung vorhanden sein.

In beiden Fallen erfordert die Anwendung dieser Eigenschaft, dass der so deklarierte Speicherort der Variablen vollständig innerhalb der Konstruktion `VAR_CONFIG...END_VAR` der Konfiguration, wie in 2.7.1 definiert, für alle Instanzen des Typs, der das Konstrukt enthält, festgelegt sein muss.

Es ist in einer Instanz von Programm- oder Funktionsbausteintypen, die solche unvollständigen Festlegungen enthalten, ein **Fehler**, wenn eine der vollständigen Festlegungen in der `VAR_CONFIG...END_VAR` Konstruktion für eine unvollständige Adressenfestlegung fehlt, die durch die Stern-Schreibweise ausgedrückt ist.

Beispiele

`%QX75` und `%Q75`	Ausgangsbit 75
`%IW215`	Eingangswort Speicherort 215
`%QB7`	Ausgangsbyte Speicherort 7
`%MD48`	Merker-Doppelwort Speicherort 48
`%IW2.5.7.1`	siehe Erläuterung unten
`%Q*`	Ausgang an einem noch nicht festgelegten Speicherort

Der Hersteller muss den Zusammenhang zwischen der direkten Darstellung einer Variablen und dem physikalischen oder logischen Speicherort des angesprochenen Punkts im Speicher, Eingang oder Ausgang festlegen. Wenn eine direkte Darstellung mit zusätzlichen ganzzahligen Feldern erweitert ist, die durch Punkte getrennt sind, muss dies als *hierarchische* physikalische oder logische Adresse interpretiert werden; dabei stellt das linke Feld die höchste Ebene der Hierarchie dar, die folgenden niederen Ebenen erscheinen auf der rechten Seite. Z. B. darf die Variable `%IW2.5.7.1` den ersten „Kanal“ (Wort) des siebten „Moduls“ im fünften „Baugruppenträger“ des zweiten „E/A-Busses“ eines SPS-Systems darstellen.

Der Gebrauch der hierarchischen Adressierung, um einem Programm in einer SPS den Datenzugriff in einer weiteren SPS zu erlauben, muss in einer Spracherweiterung betrachtet werden.

Der Gebrauch von direkt dargestellten Variablen ist nur in *Funktionsbausteinen*, wie in 2.5.2 definiert, in *Programmen*, wie in 2.5.3 definiert, und in *Konfigurationen* und *Ressourcen*, wie in 2.7.1 definiert, zugelassen. Die maximale Anzahl der Ebenen einer hierarchischen Adressierung ist ein **implementierungsabhängiger Parameter**.

Tabelle 15 – Eigenschaften der Präfixe für Speicherort und Größe bei direkt dargestellten Variablen

Nr.	Präfix	Bedeutung	Voreingestellter Datentyp
1	`I`	Speicherort Eingang	
2	`Q`	Speicherort Ausgang	
3	`M`	Speicherort Merker	
4	`X`	(Einzel-) Bit-Größe	`BOOL`
5	kein	(Einzel-) Bit-Größe	`BOOL`
6	`B`	Byte- (8 Bits) Größe	`BYTE`
7	`W`	Wort- (16 Bits) Größe	`WORD`
8	`D`	Doppelwort- (32 Bits) Größe	`DWORD`
9	`L`	Langwort- (64 Bits) Größe	`LWORD`
10	Der Gebrauch eines Stern (Asterisk, *) zur Anzeige eines noch nicht festgelegten Speicherorts (ANMERKUNG 2)		

ANMERKUNG 1 Nationale Normungsorganisationen können Tabellen mit Übersetzungen dieser Präfixe herausgeben.

ANMERKUNG 2 Der Gebrauch der Eigenschaft 10 in dieser Tabelle erfordert die Eigenschaft 11 in Tabelle 49 und umgekehrt.

2.4.1.2 Multielement-Variable

Die Typen der *Multielement*-Variablen, die in dieser Norm definiert sind, sind die *Felder* und *Strukturen*.

Ein *Feld* ist eine Sammlung von Datenelementen des gleichen Datentyps, die durch einen oder mehrere *Feldindizes* angesprochen werden; diese Indizes sind in eckigen Klammern eingeschlossen und durch Kommas getrennt. In der ST-Sprache, die in 3.3 definiert ist, muss ein Index ein Ausdruck sein, der einen

Wert entsprechend einem der Untertypen des allgemeinen Typs ANY_INT, wie in Tabelle 11 definiert, liefert. Die Form von Feldindizes in der AWL Sprache, definiert in 3.2, und in den grafischen Sprachen, definiert in Abschnitt 4, ist auf *Einzelelement-Variablen* oder *ganzzahlige Literale* beschränkt.

Ein Beispiel für die Anwendung von Feldvariablen in ST-Sprache, definiert in 3.3, ist:

```
OUTARY[%MB6,SYM] := INARY[0] + INARY[7] - INARY[%MB6] * %IW62 ;
```

Eine *strukturierte Variable* ist eine Variable, die als ein Typ deklariert ist, der vorher bereits als eine *Datenstruktur* festgelegt wurde, d. h. als ein Datentyp, der aus einer Sammlung von bezeichneten Elementen besteht.

Ein Element einer strukturierten Variablen muss durch zwei oder mehr Bezeichner oder durch Feldzugriffe dargestellt werden, die durch einzelne Punkte (.) getrennt sind. Der erste Bezeichner stellt den Namen des strukturierten Elements dar und die folgenden Bezeichner stellen die Folge von Komponentennamen zum Zugriff auf das bestimmte Datenelement innerhalb der Datenstruktur dar.

Wenn z. B. die Variable MODULE_5_CONFIG als Typ ANALOG_16_INPUT_CONFIGURATION, wie in Tabelle 12 gezeigt, deklariert worden ist, dann würden die folgenden Anweisungen in ST-Sprache, die in 3.3 definiert ist, bewirken, dass der Wert SINGLE_ENDED dem Element SIGNAL_TYPE aus der Variable MODULE_5_CONFIG zugewiesen wird. Außerdem würde der Wert BIPOLAR_10V dem Unterelement RANGE des fünften Elements CHANNEL von MODULE_5_CONFIG zugewiesen:

```
MODULE_5_CONFIG.SIGNAL_TYP := SINGLE_ENDED;
MODULE_5_CONFIG.CHANNEL[5].RANGE := BIPOLAR_10V;
```

2.4.2 Initialisierung

Wenn ein Konfigurationselement (*Ressource* oder *Konfiguration*), wie in 1.4.1 definiert, „gestartet“ wird, kann jede der Variablen, die mit dem Konfigurationselement und seinen *Programmen* verbunden ist, einen der folgenden Anfangswerte annehmen:

– den Wert, den die Variable hatte, als das Konfigurationselement „gestoppt“ wurde (ein *gepufferter* Wert);

– einen anwender-spezifizierten Anfangswert;

– den voreingestellten Anfangswert für den der Variablen zugeordneten Datentyp.

Der Anwender kann deklarieren, dass die Variable *gepuffert* sein soll, indem er das Bestimmungszeichen RETAIN benutzt, das in Tabelle 16 a) festgelegt ist, sofern diese Eigenschaft von der Implementierung unterstützt wird.

Der Anfangswert einer Variablen beim Starten seines zugehörigen Konfigurationselements muss nach folgenden Regeln ermittelt werden:

1) Falls die Startoperation ein „Warmstart“ ist, der in IEC 61131-1 definiert ist, müssen die Anfangswerte von *gepufferten* Variablen ihre *gepufferten* Werte wie oben definiert annehmen.

2) Falls die Operation ein „Kaltstart“ ist, wie in IEC 61131-1 definiert, müssen die Anfangswerte der gepufferten Variablen die anwender-spezifizierten Anfangswerte oder den voreingestellten Wert annehmen, wie in 2.3.3.2 für den zugehörigen Datentyp für jede Variable definiert ist, für die kein Anfangswert durch den Anwender spezifiziert wurde.

3) Nicht-gepufferte Variable müssen mit den anwender-spezifizierten Anfangswerten oder mit dem voreingestellten Wert initialisiert werden, wie in 2.3.3.2 für den zugehörigen Datentyp für jede Variable definiert ist, für die kein Anfangswert durch den Anwender spezifiziert wurde.

4) Variable, die *Eingänge* eines *SPS-Systems,* wie in IEC 61131-1 definiert, darstellen, müssen in einer **implementierungsabhängigen** Weise initialisiert werden.

2.4.3 Deklaration

Jede Typdeklaration eines Typs einer Programm-Organisationseinheit (d. h. jede Deklaration eines *Programms*, einer *Funktion* oder eines *Funktionsbausteins*, wie in 2.5 definiert) muss an seinem Anfang mindestens einen *Deklarationsteil* enthalten, der die Typen (und falls notwendig, den physikalischen oder

logischen Speicherort) der Variablen festlegt, die in der Organisationseinheit benutzt werden. Dieser Deklarationsteil muss die Textform eines der Schlüsselwörter VAR, VAR_INPUT oder VAR_OUTPUT, wie in Tabelle 16 a) definiert, besitzen; dabei folgt im Falle von VAR null mal oder einmal die Bestimmungszeichen RETAIN, NON_RETAIN oder das Bestimmungszeichen CONSTANT; und im Falle von VAR_INPUT oder VAR_OUTPUT folgen null mal oder einmal die Bestimmungszeichen RETAIN oder NON_RETAIN, gefolgt von einer oder mehreren Deklarationen, die durch Semikolons getrennt sind und mit dem Schlüsselwort VAR_END abgeschlossen sind. Wenn eine SPS die Deklaration von Anfangswerten von Variablen durch den Anwender unterstützt, dann muss diese Deklaration im Deklarationsteil (bzw. in Deklarationsteilen) geschehen, wie es in diesem Abschnitt definiert ist.

Tabelle 16 a) – Schlüsselwörter für Variablen-Deklaration

Schlüsselwort	Gebrauch der Variablen
VAR	Innerhalb der Organisationseinheit
VAR_INPUT	Von außerhalb geliefert, nicht innerhalb der Organisationseinheit änderbar
VAR_OUTPUT	Von der Organisationseinheit nach außen geliefert
VAR_IN_OUT	Von außerhalb geliefert – kann innerhalb der Organisationseinheit geändert werden
VAR_EXTERNAL	Von der Konfiguration geliefert via VAR_GLOBAL (2.7.1) – kann innerhalb der Organisationseinheit geändert werden
VAR_GLOBAL	Deklaration von globalen Variablen (2.7.1)
VAR_ACCESS	Deklaration von einem Zugriffspfad (2.7.1)
VAR_TEMP	Temporäre Speicherung von Variablen in Funktionsbausteinen und Programmen (2.4.3)
VAR_CONFIG	Instanz-spezifische Initialisierung und Zuweisung des Speicherorts
RETAIN[b, c, d, e]	Gepufferte Variablen (siehe vorangehenden Text)
NON_RETAIN[b, c, d, e]	Nicht-gepufferte Variablen (siehe vorangehenden Text)
CONSTANT[a]	Konstante (Variable kann nicht geändert werden)
AT	Zuweisung des Speicherorts (2.4.3.1)

ANMERKUNG 1 Der Gebrauch dieser Schlüsselwörter ist eine Eigenschaft der Programm-Organisationseinheit oder des Konfigurationselementes, in dem sie benutzt werden. Normative Anforderungen des Gebrauchs dieser Schlüsselwörter sind in 2.5 und 2.7 angegeben.

ANMERKUNG 2 Beispiele für den Gebrauch der VAR_IN_OUT Variablen sind in Bild 11 b und Bild 12 angegeben.

[a] Der Bezeichner CONSTANT darf nicht zur Deklaration von *Funktionsbaustein-Instanzen* verwendet werden, die in 2.5.2.1 beschrieben sind.

[b] Die Bezeichner RETAIN und NON_RETAIN dürfen für *Variable* verwendet werden, die in VAR, VAR_INPUT, VAR_OUTPUT und VAR_GLOBAL Blöcken deklariert sind, jedoch nicht in VAR_IN_OUT Blöcken und nicht für einzelne Elemente von Strukturen.

[c] Der Gebrauch von RETAIN und NON_RETAIN für *Funktionsbausteine* und *Programm-Instanzen* ist erlaubt. Die Wirkung ist, dass alle Elemente der Instanz als RETAIN oder NON_RETAIN behandelt werden, ausgenommen, wenn:
- das Element ausdrücklich als RETAIN oder NON_RETAIN in der Definition des Funktionsbausteins oder Programms deklariert ist;
- das Element selbst ein *Funktionsbaustein* ist.

[d] Der Gebrauch von RETAIN und NON_RETAIN für *Instanzen* von strukturierten Datentypen ist erlaubt. Die Wirkung ist, dass alle Strukturelemente, auch die von geschachtelten Strukturen, als RETAIN oder NON_RETAIN behandelt werden.

[e] Sowohl RETAIN als auch NON_RETAIN sind Eigenschaften. Wenn eine Variable weder ausdrücklich als RETAIN noch als NON_RETAIN deklariert ist, ist das „Warmstart"-Verhalten der Variablen **implementierungsabhängig**.

Innerhalb von *Funktionsbausteinen* und *Programmen* können Variablen in einer VAR_TEMP...END_VAR Konstruktion deklariert werden. Diese Variablen werden bei jedem Aufruf einer Instanz der Programm-Organisationseinheit zugeordnet und initialisiert und bleiben nicht über Aufrufe hinweg bestehen.

Der *Geltungsbereich* (Bereich der Gültigkeit) der Deklarationen, die im Deklarationsteil enthalten sind, muss *lokal* zu der Programm-Organisationseinheit sein, in der der Deklarationsteil enthalten ist. D. h. die deklarierten Variablen dürfen nicht von anderen Organisationseinheiten aus erreichbar sein; ausgenommen ist die explizite Parameterübergabe über Variable, die vorher als *Eingänge* und *Ausgänge* dieser Einheiten deklariert wurden. Die einzige Ausnahme von dieser Regel ist der Fall von Variablen, die als *global* deklariert wurden, wie in 2.7.1 definiert ist. Solche Variablen sind von einer Programm-Organisationseinheit aus nur über eine VAR_EXTERNAL Deklaration erreichbar. Der Typ einer Variablen, die in einem VAR_EXTERNAL Block deklariert ist, muss mit dem Typ übereinstimmen, der im VAR_GLOBAL Block des zugehörigen *Programms*, der *Konfiguration* oder der *Ressource* deklariert ist.

Es muss ein **Fehler** sein, wenn:

– eine Programm-Organisationseinheit versucht, den Wert einer Variablen zu verändern, die mit dem Bestimmungszeichen CONSTANT deklariert wurde;

– eine Variable, die als VAR_GLOBAL CONSTANT in einem Konfigurationselement oder einer Programm-Organisationseinheit (das „enthaltende Element“) in einer VAR_EXTERNAL Deklaration (ohne das CONSTANT Bestimmungszeichen) eines Elements verwendet wird, das im enthaltenden Element eingeschlossen ist, wie unten veranschaulicht ist.

Die maximale Anzahl von Variablen, die in einen Variablen-Deklarationsblock erlaubt sind, ist ein **implementierungsabhängiger Parameter**.

Tabelle 16 b) – Gebrauch von VAR_GLOBAL, VAR_EXTERNAL und CONSTANT Deklarationen

Deklaration von enthaltenden Elementen	Deklaration von enthaltenen Elementen	Erlaubt?
VAR_GLOBAL X ...	VAR_EXTERNAL CONSTANT X…	Ja
VAR_GLOBAL X ...	VAR_EXTERNAL X…	Ja
VAR_GLOBAL CONSTANT X ...	VAR_EXTERNAL CONSTANT X…	Ja
VAR_GLOBAL CONSTANT X ...	VAR_EXTERNAL X…	Nein

2.4.3.1 Typzuweisung

Wie in Tabelle 17 gezeigt, muss die Konstruktion VAR...END_VAR zur Festlegung von Datentypen und Pufferungsverhalten von direkt dargestellten Variablen angewendet werden. Diese Konstruktion muss auch zur Festlegung von Datentypen, Pufferverhalten und (falls nötig, nur in *Programmen* und VAR_GLOBAL Deklarationen) den physikalischen und logischen Speicherort für symbolisch dargestellte Einzel- und Multi-Elementvariable benutzt werden. Der Gebrauch der Konstruktionen VAR_INPUT, VAR_OUTPUT und VAR_IN_OUT ist in 2.5 definiert.

Die Zuordnung einer physikalischen oder logischen Adresse zu einer symbolisch dargestellten Variablen muss durch Anwendung des Schlüsselworts AT durchgeführt werden. Wo eine derartige Zuordnung nicht gemacht wird, muss eine automatische Zuweisung der Variablen zu einem geeigneten Speicherort im SPS-Speicher erfolgen.

Die Schreibweise mit Stern (Asterisk) (Eigenschaft Nr. 10 in Tabelle 15) kann bei Adresszuweisungen innerhalb von Typen von Programmen und Funktionsbausteinen verwendet werden, um noch nicht vollständig festgelegte Speicherorte für direkt dargestellte Variablen anzugeben.

Tabelle 17 – Eigenschaften der Typzuweisung für Variablen

Nr.	Eigenschaften/Beispiele	
1[a]	**Deklaration von direkt dargestellten Variablen**	

	`VAR`
	`  AT %IW6.2 : WORD;`
	`  AT %MW6  : INT ;`
	`END_VAR`
		16-Bit-Folge (Anmerkung 2)
		16-Bit-Integer, Anfangswert = 0

2[a]	**Deklaration von direkt dargestellten gepufferten Variablen**	

	`VAR RETAIN`
	`  AT %QW5 : WORD ;`
	`END_VAR`
		Beim Kaltstart, wird mit einer 16-Bit-Folge mit dem Wert `16#0000` initialisiert.

3[a]	**Deklaration von Speicherorten bei symbolischen Variablen**	
	`VAR_GLOBAL` ` LIM_SW_S5 AT %IX27 : BOOL;`	Weist Eingangsbit 27 der boolesche Variablen `LIM_SW_5` zu. (Anmerkung 2)
	`CONV_START AT %QX25 : BOOL;`	Weist Ausgangsbit 25 der boolesche Variablen `CONV_START` zu.
	`TEMPERATURE AT %IW28 : INT;`	Weist Eingangswort 28 der ganzzahligen Variablen `TEMPERATURE` zu. (Anmerkung 2)
	`VAR C2 AT %Q* : BYTE ;` `END_VAR`	Weist noch nicht zugeordnetes Ausgangsbyte der Bit-Folge-Variable `C2` der Länge 8 Bit zu.
4[a]	**Zuweisung von Speicherort bei Feld**	

	`VAR`
	`INARY AT %IW6 :`
	` ARRAY [0..9] OF INT;`
	`END_VAR`
		Deklariert ein Feld von 10 ganzen Zahlen, das dem zusammenhängenden Eingangsspeicherort beginnend ab `%IW6` zugeordnet ist. (Anmerkung 2)

5	**Automatische Speicherzuteilung für symbolische Variablen**	

	`VAR`
	`  CONDITION_RED : BOOL;`
	`  IBOUNCE : WORD ;`
	`  MYDUB : DWORD ;`
	`  AWORD, BWORD, CWORD : INT;`
	`  MYSTR: STRING[10] ;`
	`END_VAR`
		Teilt ein Speicher-Bit der booleschen Variablen `CONDITION_RED` zu.
		Teilt ein Speicher-Wort der 16-Bit-Folge-Variablen `IBOUNCE` zu.
		Teilt ein Speicher-Doppelwort der 32-Bit-Folge-Variablen `MYDUB` zu.
		Teilt 3 separate Speicher-Wörter den ganzzahligen Variablen `AWORD`, `BWORD` und `CWORD` zu.
		Teilt Speicher zu, der eine Zeichenfolge mit einer Maximallänge von 10 Zeichen enthält. Nach der Initialisierung hat die Folge die Länge 0 und enthält die leere Folge `' '`.

6	**Deklaration für Feld**	

	`VAR THREE :`
	`ARRAY[1..5,1..10,1..8] OF INT;`
	`END_VAR`
		Teilt für ein drei-dimensionales Feld von ganzen Zahlen 400 Speicher-Wörter zu.

7	**Deklaration für gepuffertes Feld**	

	`VAR RETAIN RTBT:`
	`  ARRAY [1..2,1..3] OF INT;`
	`END_VAR`
		Deklariert das gepufferte Feld `RTBT` mit den „Kaltstart“-Anfangswerten `0` für alle Elemente.

Nr.	Eigenschaften/Beispiele	
8	**Deklaration für strukturierte Variablen**	
	`VAR MODULE_8_CONFIG : ANALOG_16_INPUT_CONFIGURATION; END_VAR`	Deklaration einer Variablen mit abgeleitetem Datentyp (siehe Tabelle 12).
ANMERKUNG 1 Die Initialisierung von Systemeingängen ist **implementierungsabhängig**, siehe 2.4.2. ANMERKUNG 2 Die Anmerkungen zu Tabelle 16 a) gelten auch für diese Tabelle.		
[a] Wenn direkt dargestellten Variablen explizit der Speicherort zugewiesen wird, können die Eigenschaften 1 bis 4 nur in `PROGRAM` und `VAR_GLOBAL` Deklarationen angewendet werden, wie in 2.5.3 bzw. 2.7.1 definiert. Wenn die Schreibweise mit Stern von Eigenschaft 10 in Tabelle 15 verwendet wird, um die Instanz-spezifische Speicherort-Zuweisung für eine teilweise festgelegte direkt repräsentierte Variable anzugeben, können die Eigenschaften 1 und 2 nicht verwendet werden, und die Eigenschaften 3 und 4 können nur in Deklarationen von internen Variablen von Funktionsbausteinen und Programmen verwendet werden, wie in 2.5.2 bzw. 2.5.3 definiert.		

2.4.3.2 Anfangswert-Zuweisung

Die `VAR...END_VAR` Konstruktion, die in Tabelle 18 gezeigt ist, kann benutzt werden, um die Anfangswerte von direkt dargestellten Variablen oder symbolisch dargestellten Einzel- oder Multi-Element-Variablen darzustellen.

Anfangswerte können auch durch Anwenden der Instanz-spezifischen Initialisierung festgelegt werden, die durch das `VAR_CONFIG...END_VAR` Konstrukt geboten wird, das in 2.7.1 beschrieben wird (Tabelle 49, Eigenschaft 11). Instanz-spezifische Anfangswerte überschreiben immer die Typ-spezifischen Anfangswerte.

ANMERKUNG Der Gebrauch der Konstruktionen `VAR_INPUT`, `VAR_OUTPUT` und `VAR_IN_OUT` ist in 2.5 definiert.

Die Anfangswerte können nicht in `VAR_EXTERNAL` Deklarationen angegeben werden.

Während der Initialisierung von Feldern, wenn das Feld mit einer Liste von Initialisierungsvariablen gefüllt wird, muss sich der Feldindex ganz rechts am schnellsten ändern.

Runde Klammern können als ein Wiederholungsfaktor in Initialisierungslisten für Felder benutzt werden; z. B. ist `2 (1, 2, 3)` gleichbedeutend mit der Initialisierungsfolge `1, 2, 3, 1, 2, 3.`

Falls die Anzahl von Anfangswerten, die in der Initialisierungsliste angegeben ist, die Anzahl der Feldeinträge überschreitet, muss die Überschreitung der (rechten) Anfangswerte ignoriert werden. Falls die Anzahl der Anfangswerte geringer als die Anzahl der Feldeinträge ist, müssen die übrig gebliebenen Feldeinträge mit den voreingestellten Werten des zugehörigen Datentyps aufgefüllt werden. In beiden Fällen muss der Anwender wegen dieses Vorgangs während der Aufbereitung des Programms gewarnt werden.

Wenn eine Variable als ein abgeleiteter strukturierter Datentyp nach 2.3.3.1 deklariert wird, können Anfangswerte für die Elemente der Variablen in einer eingeklammerten Liste mit nachfolgendem Datentyp-Bezeichner deklariert werden, wie in Tabelle 18 gezeigt ist. Elemente, für die keine Anfangswerte in der Liste der Anfangswerte angegeben sind, müssen die voreingestellten Anfangswerte besitzen, die für diese Elemente in der Datentyp-Deklaration deklariert sind.

Wenn eine Variable als *Funktionsbaustein-Instanz,* wie in 2.5.2.2 definiert, deklariert ist, können die Anfangswerte für die Eingangs- und die zugänglichen Variablen des Funktionsbausteins in einer eingeklammerten Liste deklariert werden, die auf den Zuweisungsoperator folgt, der dem Bezeichner des Funktionsbausteintyps folgt, wie es in Tabelle 18 gezeigt ist. Elemente, für die keine Anfangswerte in der Liste angegeben sind, müssen die voreingestellten Anfangswerte besitzen, die für diese Elemente in der Funktionsbaustein-Deklaration deklariert sind.

Tabelle 18 – Zuweisung von Anfangswerten für Variablen

Nr.	Eigenschaft/Beispiele	
1 [a]	**Initialisierung von direkt dargestellten Variablen**	

	`VAR AT %QX5.1 : BOOL := 1;`
	`  AT %MW6 : INT := 8;`
	`END_VAR`
		Boolescher Typ, Anfangswert = `1`
		Initialisiert ein Speicher-Wort mit der ganzen Zahl `8`

2 [a]	**Initialisierung von direkt dargestellten gepufferten Variablen**	

	`VAR RETAIN`
	`  AT %QW5 : WORD := 16#FF00 ;`
	`END_VAR`
		Beim Kaltstart werden die 8 höchstwertigen Bits der 16-Bit-Folge des Ausgangsworts 5 mit 1 und die 8 niederwertigen Bits mit 0 initialisiert.

3 [a]	**Zuweisung von Speicherort und Anfangswert für symbolische Variablen**	

	`VAR`
	`  VALVE_POS AT %QW28 : INT := 100;`
	`END_VAR`
		Weist das Ausgangswort 28 der ganzzahligen Variablen `VALVE_POS` mit einem Anfangswert 100 zu.

4 [a]	**Zuweisung von Speicherort und Initialisierung für Feld**	

	`VAR OUTARY AT %QW6 :`
	` ARRAY [0..9] OF INT := [10(1)];`
	`END_VAR`
		Deklariert ein Feld von 10 ganzen Zahlen, die dem zusammenhängenden Ausgangsbereich beginnend ab `%QW6` jeweils mit einem Anfangswert von 1 zugeteilt werden.

5	**Initialisierung von symbolischen Variablen**	

	`VAR`
	`  MYBIT : BOOL := 1 ;`
	`  OKAY : STRING[10] := 'OK';`
	`END_VAR`
		Weist ein Speicher-Bit der booleschen Variablen `MYBIT` mit einem Anfangswert von 1 zu.
		Weist Speicher zu, der eine Zeichenfolge mit einer Maximallänge von 10 enthält. Nach Initialisierung hat die Folge die Länge 2 und enthält die 2-Byte-Zeichenfolge '`OK`' /dezimal 79 bzw. 75), in einer für das Ausdrucken als Zeichenfolge geeigneten Anordnung.

<table>
<tr><th>Nr.</th><th colspan="2">Eigenschaft/Beispiele</th></tr>
<tr><td rowspan="2">6</td><td colspan="2">Initialisierung für Feld</td></tr>
<tr><td>

```
VAR
 BITS : ARRAY [0..7] OF BOOL
   := [1, 1, 0, 0, 0, 1, 0, 0];
 TBT : ARRAY [1..2,1..3]
          OF INT
     := [1, 2, 3(4), 6];
END_VAR
```

</td><td>Weist 8 Speicher-Bits die Anfangswerte zu

```
  BITS[0] := 1, BITS[1]
:= 1,...,
  BITS[6] := 0, BITS[7]
:= 0.
```

Weist einem 2-mal-3 Feld TBT von ganzen Zahlen die Anfangswerte zu

```
  TBT[1,1] := 1,
TBT[1,2] := 2,
  TBT[1,3] := 4,
TBT[2,1] := 4,
  TBT[2,2] := 4,
TBT[2,3] := 6.
```

</td></tr>
<tr><td rowspan="2">7</td><td colspan="2">Deklaration und Initialisierung für gepuffertes Feld</td></tr>
<tr><td>

```
VAR RETAIN
RTBT :
  ARRAY
(1..2, 1..3)
OF INT
     := [1,
2, 3(4)];
END_VAR
```

</td><td>Deklariert das gepufferte Feld RTBT mit den „Kaltstart“-Anfangswerten

```
  RTBT[1,1] := 1, RTBT[1,2] := 2,
  RTBT[1,3] := 4, RTBT[2,1] := 4,
  RTBT[2,2] := 4, RTBT[2,3] := 0.
```

</td></tr>
<tr><td rowspan="2">8</td><td colspan="2">Initialisierung von strukturierten Variablen</td></tr>
<tr><td>

```
VAR
MODULE_8_CONF
IG :

ANALOG_16_INP
UT_CONFIGURAT
ION :=

(SIGNAL_TYPE
:=
DIFFERENTIAL,
     CHANNEL
        :=
[4((RANGE :=
UNIPOLAR_1_5_
V)),

(RANGE :=
BIPOLAR_10_V,

MIN_SCALE :=
0,

MAX_SCALE :=
500)]) ;
END_VAR
```

</td><td>Initialisierung einer Variablen mit abgeleitetem Datentyp (siehe Tabelle 12)

Dieses Beispiel veranschaulicht die Deklaration eines nicht-voreingestellten Werts für das fünfte Element des Feldes <code>CHANNEL</code> der Variablen <code>MODULE_8_CONFIG</code>.</td></tr>
<tr><td rowspan="2">9</td><td colspan="2">Initialisierung von Konstanten</td></tr>
<tr><td colspan="2">

```
VAR CONSTANT PI : REAL := 3.141592 ; END_VAR
```

</td></tr>
</table>

Nr.	Eigenschaft/Beispiele	
10	**Initialisierung von Funktionsbaustein-Instanzen**	

	`VAR TempLoop`
	`:`
	`  PID :=`
	`  (ProbBand`
	`:= 2.5,`
	`  Integral`
	`:= T#5S);`
	`END_VAR`
		Weist die Anfangswerte den Eingängen und Ausgängen einer Funktionsbaustein-Instanz zu.

[a] Die Eigenschaften 1 bis 4 können nur in Deklarationen von `PROGRAM` und `VAR_GLOBAL` benutzt werden, wie in 2.5.3 bzw. 2.7.1 definiert ist.		

2.5 Programm-Organisationseinheiten

Die in diesem Teil von IEC 61131 definierten Programm-Organisationseinheiten sind die *Funktion*, der *Funktionsbaustein* und das *Programm*. Diese Programm-Organisationseinheiten können vom Hersteller geliefert werden oder vom Anwender mit den Mitteln programmiert werden, die in diesem Teil der Norm definiert sind.

Programm-Organisationseinheiten dürfen nicht *rekursiv* sein; d. h. der Aufruf einer Programm-Organisationseinheit darf nicht den Aufruf einer weiteren Programm-Organisationseinheit desselben Typs verursachen.

Die zur Bestimmung der Programm-Ausführungszeiten notwendige Information darf aus einem oder mehreren **implementierungsspezifischen Parametern** bestehen.

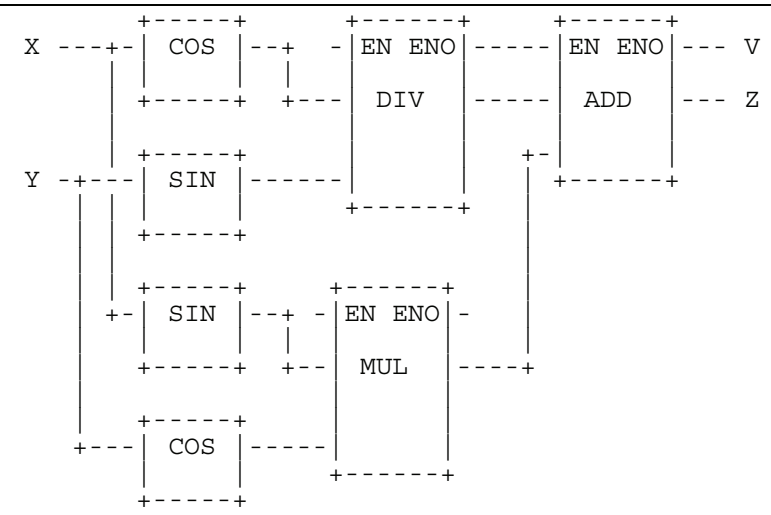
2.5.1 Funktionen

Für die Zwecke der SPS-Programmiersprachen ist eine *Funktion* als eine Programm-Organisationseinheit definiert, die bei Ausführung genau ein Datenelement, das als das Funktionsergebnis betrachtet werden kann, und beliebig viele zusätzliche Ausgangselemente (`VAR_OUTPUT` und `VAR_IN_OUT`) liefert. Das Funktionsergebnis kann wie bei jedem Datenelement mehrere Werte haben, z. B. ein Feld oder eine Struktur. Der Aufruf einer Funktion kann in Textsprachen als ein Operand in einem Ausdruck verwendet werden. Z. B. könnten die `SIN`- und `COS`-Funktionen, wie in Bild 4 gezeigt, angewendet werden.

a)

```
VAR X,Y,Z,RES1,RES2 : REAL; EN1,V : BOOL; END_VAR


RES1 := DIV(IN1 := COS(X), IN2 := SIN(Y), ENO => EN1);
RES2 := MUL (SIN(X), COS(Y));
Z: = ADD(EN := EN1, IN1 := RES1, IN2 := RES2, ENO => V);
```

b)



a) Strukturierter Text (ST) – siehe 3.3

b) Funktionsbaustein Sprache (FBS) – siehe 4.3

ANMERKUNG Dieses Bild zeigt zwei verschiedene Darstellungen derselben Funktionalität. Es ist nicht erforderlich, eine automatische Umformung zwischen diesen zwei Darstellungsformen zu unterstützen.

Bild 4 – Beispiele für den Gebrauch von Funktionen

Funktionen dürfen keine internen Zustandsinformationen beinhalten; d. h. der Aufruf einer Funktion mit denselben Argumenten (Eingangsvariablen `VAR_INPUT` und Ein-Aus-Variablen `VAR_IN_OUT`) muss immer dieselben Werte (Ausgangsvariable `VAR_OUTPUT`, Ein-Aus-Variable `VAR_IN_OUT` und das Funktionsergebnis) liefern. Es muss ein **Fehler** sein, wenn externe Variable, wie in 2.4.3 definiert, eine Verletzung dieser Regel verursachen.

Jeder Funktionstyp, der bereits deklariert wurde, kann in der Deklaration einer weiteren Programm-Organisationseinheit benutzt werden, wie in Bild 3 gezeigt.

2.5.1.1 Darstellung

Funktionen und ihr Aufruf können entweder grafisch oder in Textform dargestellt werden.

In den Textsprachen, die in Abschnitt 3 dieser Norm definiert sind, muss der Aufruf von Funktionen nach den folgenden Regeln erfolgen:

1) Die Zuweisung von Eingangsargumenten muss den Regeln in Tabelle 19 a) folgen.
2) Die Zuweisungen von Ausgangsvariablen von Funktionen müssen entweder leer sein oder an Variable erfolgen.
3) Zuweisungen an `VAR_IN_OUT` Argumente müssen Variable sein.
4) Zuweisungen an `VAR_INPUT` Argumente dürfen leer sein (Eigenschaft 1 von Tabelle 19 a)) oder sind Konstante, Variable oder Funktionsaufrufe. In letzterem Fall wird das Funktionsergebnis als das aktuelle Argument verwendet.

In den grafischen Sprachen, die in Abschnitt 4 dieser Norm definiert sind, müssen Funktionen als grafische Schaltzeichen (Block) nach den folgenden Regeln dargestellt sein:

1) Die Form des Schaltzeichens muss rechteckig oder quadratisch sein.

2) Die Größe und das Seitenverhältnis des Schaltzeichens darf in Abhängigkeit von der Anzahl der Eingänge, Ausgänge und weiteren Informationen, die anzuzeigen sind, variieren.

3) Die Richtung der Bearbeitung muss von links nach rechts durch das Schaltzeichen verlaufen (Eingangsvariable auf der linken Seite und Ausgangsvariable auf der rechten Seite).

4) Der Name oder das Symbol der Funktion, die wie im Folgenden festgelegt sind, müssen innerhalb des Schaltzeichens angegeben sein.

5) Es muss die Möglichkeit bestehen, dass die Namen der formalen Eingangs- bzw. Ausgangsvariablen innen auf den linken bzw. rechten Seiten des Schaltzeichens erscheinen, wenn das Schaltzeichen Folgendes darstellt:

   – Eine der Standardfunktionen, die in Abschnitt 2.5.1.5 definiert sind, wenn die angegebene grafische Form die Variablennamen enthält oder

   – eine zusätzliche Funktion, die wie in 2.5.1.3 festgelegt ist.

   Dieser Gebrauch unterliegt den folgenden Bedingungen:

   a) Wenn keine Namen für die Eingangsvariablen in Standardfunktionen angegeben sind, müssen die voreingestellten Namen `IN1, IN2, ...` von oben nach unten gelten.

   b) Wenn eine Standardfunktion einen einzelnen Eingang ohne Namen hat, muss der voreingestellte Name `IN` gelten.

   c) Die voreingestellten Namen, die oben beschrieben sind, können, aber müssen nicht links innerhalb der grafischen Darstellung erscheinen.

6) Es darf ein zusätzlicher Eingang `EN` und/oder Ausgang `ENO`, wie in 2.5.1.2 festgelegt, verwendet werden. Sie müssen, falls vorhanden, an den obersten Positionen auf der linken bzw. rechten Seite des Schaltzeichens gezeigt werden.

7) Das Funktionsergebnis muss an der obersten Position auf der rechten Seite des Schaltzeichens gezeigt werden, ausgenommen, es gibt einen `ENO` Ausgang. Dann muss das Funktionsergebnis an der nächsten Position unter dem `ENO` Ausgang gezeigt werden. Da der Name der Funktion für die Zuweisung ihrer Ausgangswerte, wie in 2.5.1.3 festgelegt, verwendet wird, darf kein Ausgangsvariablen-Name auf der rechten Seite des Schaltzeichens gezeigt werden.

8) Die Verbindungen der Argumente (einschließlich des Funktionsergebnisses) müssen durch Signalfluss-Linien angezeigt werden.

9) Die Negation von booleschen Signalen muss durch Angabe eines offenen Kreises direkt außerhalb der Verbindung der Eingangs- oder Ausgangslinie mit dem Schaltzeichen gezeigt werden. Dies muss in dem Zeichensatz, der in 2.1.1 definiert ist, durch den Großbuchstaben `"O"` dargestellt werden, wie in Tabelle 19 gezeigt ist.

10) Alle Eingänge und Ausgänge (einschließlich dem Funktionsergebnis) einer grafisch dargestellten Funktion müssen als eine einfache Linie außen an der entsprechenden Seite des Schaltzeichens dargestellt werden; das gilt auch, wenn das Datenelement ein Mehrfach-Element ist.

11) Funktionsergebnisse und Funktionsausgänge (`VAR_OUTPUT`) können mit einer Variablen verbunden werden, als Eingang mit anderen Funktionsbausteinen oder Funktionen verbunden werden oder unverbunden bleiben.

12) Es muss ein **Fehler** sein, wenn eine `VAR_IN_OUT` Variable eines Funktionsbausteinausrufs oder eines Funktionsaufrufs in einer POE nicht „korrekt abgebildet“ ist. Eine `VAR_IN_OUT` Variable ist „korrekt abgebildet“, wenn sie grafisch auf der linken Seite mit einer Variablen verbunden ist oder bei einem Aufruf in Textsprache unter Verwendung des „`:=`“ Operators einer Variablen zugewiesen ist, die in einem `VAR_IN_OUT`, `VAR, VAR_OUT` oder `VAR_EXTERNAL` Block (ohne das Bestimmungszeichen `CONSTANT`) der enthaltenden Programm-Organisationseinheit deklariert ist, oder wenn sie mit einer „korrekt abgebildeten“ `VAR_IN_OUT` Variablen einer anderen Funktionsbaustein-Instanz oder eines anderen Funktionsaufrufs verbunden ist.

13) Eine „korrekt abgebildete“ (siehe oben Regel 12) `VAR_IN_OUT` Variable einer Funktionsbaustein-Instanz oder eines Funktionsaufrufs kann grafisch rechts mit einer Variablen verbunden sein oder unter Verwendung des „`:=`“ Operators in einer Zuweisung in Textsprache einer Variablen zugewiesen sein, die in einem `VAR`, `VAR_OUTPUT` oder `VAR_EXTERNAL` Block einer enthaltenden Programm-Organisations-

einheit deklariert ist. Es muss ein **Fehler** sein, wenn eine solche Verbindung zu einem mehrdeutigen Wert der so verbundenen Variablen führt.

Tabelle 19 – Grafische Negation von booleschen Signalen

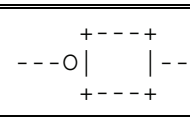
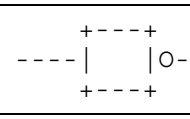
Nr.	Eigenschaft [a, b]	Darstellung
1	Negierter Eingang	
2	Negierter Ausgang	
[a] Falls eine dieser Eigenschaften für Funktionen unterstützt wird, muss sie auch für Funktionsbausteine, die in 2.5.2 definiert sind, unterstützt werden und umgekehrt. [b] Der Gebrauch dieser Konstrukte ist für Ein-Aus-Variable verboten.		

Bild 5 veranschaulicht sowohl den grafischen als auch den gleichbedeutenden Gebrauch in Textform von Funktionen, einschließlich des Gebrauchs einer Standardfunktion (ADD), ohne definierte formale Argument-Namen; es veranschaulicht eine Standardfunktion (SHL) mit definierten Argumentnamen, die gleiche Funktion mit zusätzlichem Gebrauch von EN Eingang und negiertem ENO Ausgang und eine anwender-definierte Funktion (INC) mit definierten formalen Argumentnamen.

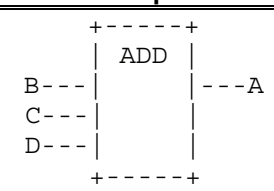
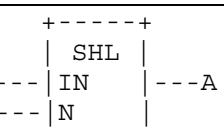
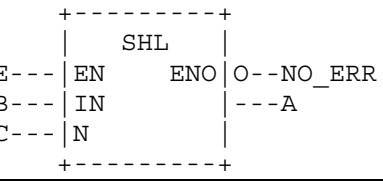
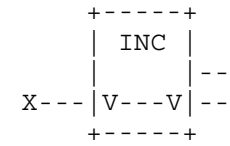
Beispiel	Erläuterung
	Grafischer Gebrauch der ADD Funktion (Siehe 2.5.1.5.2) (FBS Sprache; siehe 4.3) (Keine formalen Variablennamen)
`A := ADD(B,C,D);`	Textueller Gebrauch der ADD Funktion (ST Sprache; siehe 3.3)
	Grafischer Gebrauch der SHL Funktion (Siehe 2.5.1.5.3) (FBS Sprache; siehe 4.3) (Formale Argumentnamen)
`A := SHL(IN := B,N := C);`	Textueller Gebrauch der SHL Funktion (ST Sprache; siehe 3.3)
	Grafischer Gebrauch der SHL Funktion (Siehe 2.5.1.5.3) (FBS Sprache; siehe 4.3) (Formale Argumentnamen; Gebrauch von EN Eingang und negiertem ENO Ausgang)
`A := SHL(EN:=ENABLE, IN:=B, N:=C,` `NOT ENO => NO_ERR);`	Gebrauch der SHL Funktion in Textform (ST Sprache; siehe 3.3)
	Grafischer Gebrauch der anwender-definierten INC Funktion (FBD Sprache, siehe 4.3) (Formale Argumentnamen für VAR_IN_OUT)
`A := INC(V := X) ;`	Gebrauch der INC Funktion in Textform (ST Sprache, siehe 3.3)

Bild 5 – Gebrauch von formalen Argumentnamen

Eigenschaften für den Aufruf von Funktionen in Textform sind in Tabelle 19 a) definiert. Der Aufruf einer Funktion in Textform muss aus den Funktionsnamen, gefolgt von einer Liste von Argumenten bestehen. In

der ST Sprache, die in 3.3 definiert ist, müssen die Argumente durch Kommas getrennt sein, und diese Liste muss links und rechts durch runde Klammern begrenzt sein.

In Eigenschaft 1 von Tabelle 19 a) (formaler Aufruf) hat die Argumentenliste die Form einer Menge von Zuweisungen von aktuellen Werten an die formalen Argumentnamen (formale Argumentenliste), das heißt:

1) Wertzuweisungen an Eingangs- und Ein-Aus-Variablen verwenden den „:=“ Operator und

2) Wertzuweisungen an Ausgangsvariable verwenden den „=>“ Operator.

Die Reihenfolge der Argumente in der Liste darf keine Bedeutung haben. In Eigenschaft 1 von Tabelle 19 a) muss jede Variable, der kein Wert in der Liste zugewiesen ist, den eventuell vorhandenen voreingestellten Wert haben, der in der Funktionsfestlegung zugewiesen ist, oder sie muss den voreingestellten Wert für den zugehörigen Datentyp haben.

In Eigenschaft 2 von Tabelle 19 a) (nicht-formaler Aufruf) muss die Argumentenliste genau dieselbe Anzahl von Argumenten in genau derselben Reihenfolge und mit denselben Datentypen haben, wie in der Funktionsdefinition angegeben ist, ausgenommen der Argumente zur Ausführungssteuerung EN und ENO.

Tabelle 19 a) – Aufruf in Textform von Funktionen mit formalen und nicht-formalen Argumentenliste

Nr.	Eigenschaft				Beispiel In Strukturierter Text (ST) Sprache – siehe 3.3
	Aufruftyp	**Variablen-zuweisung**	**Variablen-reihenfolge**	**Anzahl der Varablen**	
1	formal	ja	beliebig	beliebig	`A := LIMIT(EN:=COND,IN:=B, MX:=5, ENO => TEMPL);`
2 [a]	nicht-formal	nein	fest	fest	`A := LIMIT(1, B, 5);`

[a] Eigenschaft Nr. 2 ist **erforderlich** zum Aufruf bei jeder der Standardfunktionen, die in 2.5.1.5 ohne formale Namen oder Eingangsvariablen definiert sind, es muss jedoch Eigenschaft Nr. 1 verwendet werden, wenn EN/ENO bei Funktionsausrufen notwendig sind.

ANMERKUNG 1 In dem Beispiel, das in Eigenschaft Nr. 1 angegeben ist, wird die Variable MN den voreingestellten Wert 0 (Null) haben.

ANMERKUNG 2 Das Beispiel, das in Eigenschaft Nr. 2 angegeben ist, ist semantisch gleichbedeutend mit dem folgenden Aufruf mit formalen Variablenzuweisungen (Eigenschaft Nr. 1):

```
A := LIMIT(EN := TRUE,MN := 1, IN := B, MX := 5);
```

2.5.1.2 Ausführungssteuerung

Wie in Tabelle 20 gezeigt, können ein zusätzlicher boolescher (Freigabe-) Eingang EN (Enable) oder Ausgang ENO (Enable Out) oder beide vom Hersteller oder vom Anwender zur Verfügung gestellt werden. Dabei gelten folgende Deklarationen:

```
VAR_INPUT EN: BOOL := 1 ; END_VAR
VAR_OUTPUT ENO: BOOL ; END_VAR
```

Wenn diese Variablen benutzt werden, muss die Ausführung der Operationen, die durch die Funktion definiert ist, nach folgenden Regeln gesteuert werden:

1) Falls der Wert von EN gleich FALSE (0) ist, wenn die Funktion aufgerufen wird, dürfen die Operationen, die durch den Funktionsrumpf definiert sind, nicht ausgeführt werden, und der Wert von ENO muss durch das SPS-System auf FALSE (0) zurückgesetzt werden.

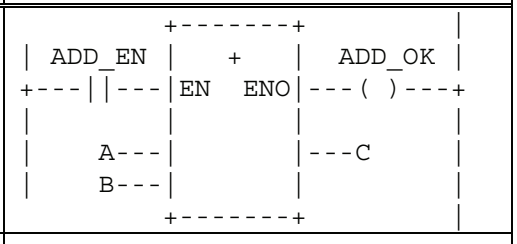
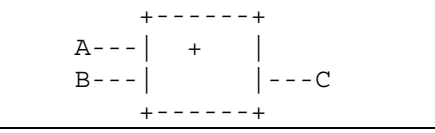
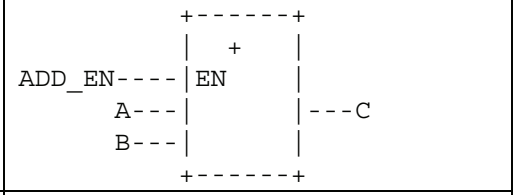
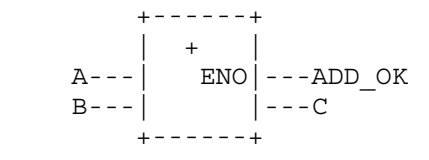
2) Andernfalls muss der Wert von ENO durch das SPS-System auf TRUE (1) gesetzt werden, und die Operationen, die durch den Funktionsrumpf definiert sind, müssen ausgeführt werden. Diese Operationen können die Zuweisung eines booleschen Wertes an ENO einschließen.

3) Falls einer der Fehler, die in Tabelle E.1 für 2.5.1.5 definiert sind, während der Ausführung einer der Standardfunktionen, die in 2.5.1.5 definiert sind, auftritt, muss der ENO Ausgang dieser Funktion durch das SPS-System auf FALSE (0) rückgesetzt werden oder der Hersteller muss eine andere Bedeutung eines solchen **Fehlers** nach den Bestimmungen von 1.5.1 festlegen.

4) Wenn der ENO Ausgang mit FALSE (0) ausgewertet wird, müssen die Werte aller Funktionsausgänge (VAR_OUTPUT, VAR_IN_OUT und Funktionsergebnisse) als **implementierungsabhängig** betrachtet werden.

ANMERKUNG Aus diesen Regeln folgt, dass der ENO Eingang einer Funktion explizit nötigenfalls durch die aufrufende Einheit geprüft werden muss, um mögliche Fehlerbedingungen zu erkennen.

Tabelle 20 – Der Gebrauch des EN Eingangs und ENO Ausgangs

Nr.	Eigenschaft	Beispiel
1	Gebrauch von EN und ENO Gezeigt bei KOP (Kontaktplan Sprache); siehe 4.2	
2	Gebrauch ohne EN und ENO Gezeigt bei FBS (Funktionsbaustein-Sprache); siehe 4.3	
3	Gebrauch mit EN und ohne ENO Gezeigt bei FBS (Funktionsbaustein-Sprache); siehe 4.3	
4	Gebrauch ohne EN und mit ENO Gezeigt bei FBS (Funktionsbaustein-Sprache); siehe 4.3	
[a] Die grafischen Sprachen, die zum Zeigen der Eigenschaften oben gewählt wurden, sind nur als Beispiele angegeben. Eigenschaften müssen, wenn sie von einem Hersteller ausgewählt sind, für alle Sprachen gelten, die durch den Hersteller unterstützt werden (AWL, ST, KOP, FBS).		

2.5.1.3 Deklaration

Eigenschaften für die Deklaration von Funktionen in Text- und in grafischer Form sind in Tabelle 20 a) aufgeführt.

Wie in Bild 6 veranschaulicht, muss die Text-Deklaration einer Funktion aus folgenden Elementen bestehen:

1) Das Schlüsselwort FUNCTION, gefolgt von einem Bezeichner, der den Namen der zu deklarierenden Funktion festlegt, einem Doppelpunkt (:) und dem Datentyp des Wertes, der von der Funktion zurückgegeben wird;

2) Ein Konstrukt VAR_INPUT ... END_VAR, wie in 2.4.2 definiert, das die Namen und Typen der Eingangsvariablen der Funktion festlegt;

3) Falls erforderlich, Konstrukte VAR_IN_OUT...END_VAR und VAR_OUTPUT...END_VAR (siehe F.11 mit einem Beispiel für den Gebrauch des letztgenannten Konstrukts), wie in 2.4.3 definiert, das die Namen und Typen der Ein-Aus-Variablen und Ausgangsvariablen der Funktion festlegt;

4) Falls erforderlich ein VAR...END_VAR Konstrukt, das die Namen und Typen der internen Variablen der Funktion festlegt;

5) Ein *Funktionsrumpf*, geschrieben in einer der in dieser Norm definierten Sprachen oder einer weiteren Programmiersprache, wie in 1.4.3 definiert, die Operationen festlegt, die auf der (die) Variable(n) ausgeführt werden, um Werte abhängig von der Semantik der Funktion einer Variablen zuzuweisen, die denselben Namen wie die Funktion hat, und die das Funktionsergebnis darstellt, das von der Funktion (Funktionsergebnis) zurückgegeben wird, und auch Werte den Ein-Aus-Variablen oder Ausgangsvariablen zuzuweisen;

6) Das abschließende Schlüsselwort END_FUNCTION.

Wenn die allgemeinen Datentypen, die in Tabelle 11 angegeben sind, in der Deklaration von Standardfunktions-Variablen verwendet werden, dann müssen die Regeln zur Beeinflussung der aktuellen Typen der Argumente von solchen Funktionen ein Teil der Funktionsdeklaration sein.

Die Konstrukte zur Variableninitialisierung, die in 2.4.3.2 definiert sind, können für die Deklaration von voreingestellten Werten von Funktionseingängen und von Anfangswerten ihrer internen Ausgangswerte verwendet werden.

Die Variablenwerte, die an die Funktion mittels eines VAR_IN_OUT Konstrukts übergeben werden, können innerhalb der Funktion verändert werden.

Wie in Bild 6 veranschaulicht, muss die grafische Deklaration einer Funktion aus folgenden Elementen bestehen:

1) Die einklammernden Schlüsselwörter FUNCTION und END_FUNCTION oder ein grafisches Äquivalent.

2) Eine grafische Spezifikation des Funktionsnamens und der Namen und Typen der Eingänge und möglicher Anfangswerte von Ergebnis und Variablen (Eingang, Ausgang und Ein-Aus) der Funktion.

3) Eine Spezifikation der Namen, Typen und der möglichen Anfangswerte der internen Variablen, die in der Funktion benutzt werden, z. B. kann dazu das VAR...END_VAR Konstrukt benutzt werden.

4) Ein Funktionsrumpf wie oben definiert.

Die maximale Anzahl von Funktionsfestlegungen, die in einer bestimmten *Ressource* zulässig ist, ist ein **implementierungsabhängiger Parameter**.

Tabelle 20 a) – Eigenschaften der Funktion

Nr.	Beschreibung	Beispiel
1	Deklaration von Ein-Aus-Variablen (Textform)	`VAR_IN_OUT A: INT; END_VAR`
2	Deklaration von Ein-Aus-Variablen (grafisch)	Siehe Bild 6 b)
3	Grafische Verbindung einer Ein-Aus-Variablen zu anderen Variablen (grafisch)	Siehe Bild 6 d)

a)

```
FUNCTION SIMPLE_FUN : REAL
  VAR_INPUT
    A,B : REAL ;              (* Festlegung der externen Schnittstelle *)
      C : REAL := 1.0;
  END_VAR
  VAR_IN_OUT COUNT : INT ; END_VAR
  VAR COUNTP1 : INT ; END_VAR
  COUNTP1 := ADD(COUNT,1);  (*Festlegung des Funktionsrumpfs *)
  COUNT := COUNTP1 ;
     SIMPLE_FUN := A*B/C;
  END_FUNCTION
```

ANMERKUNG Im Beispiel oben erhält die Ausgangsvariable einen voreingestellten Wert von 1.0, wie in 2.4.3.2 festgelegt, um einen **Fehler** „Division durch Null“ zu vermeiden, wenn die Funktion aufgerufen wird, falls z. B. ein grafischer Eingang unverbunden gelassen ist.

b)

```
FUNCTION
                +-------------+   (*Festlegung von externer Schnittstelle *)
                | SIMPLE_FUN  |
       REAL----|A            |----REAL
       REAL----|B            |
       REAL----|C            |
       INT-----|COUNT---COUNT|----INT
                +-------------+
(* Festlegung von Funktionsrumpf *)
              +----+
              |ADD |---          +----+
       COUNT--|    |---COUNTP1--| := |---COUNT
           1--|    |             +----+
              +----+
                        +---+
                   A----| * |   +---+
                   B----|   |---| / |---SIMPLE_FUN
                        +---+   |   |
                   C------------|   |
                                +---+
END_FUNCTION
```

c)

```
  ....
VAR X,Y,Z,RESULT : REAL:
VAR COUNT1,COUNT2 : INT;
   ...
RESULT := SIMPLE_FUN(A:=X, B:=Y, C:=Z, COUNT:=COUNT1);
COUNT2 := COUNT1;
   ...
```

d)

```
            +-------------+
            | SIMPLE_FUN  |
      X----|A            |----RESULT
      Y----|B            |
      Z----|C            |
 COUNT1---|COUNT---COUNT|----COUNT2
            +-------------+
```

ANMERKUNG Die Wirkung dieses Aufrufs dieser Funktion ist identisch zu der in Bild 6 c).

a) Deklaration in Textform in der Sprache ST (siehe 3.3).
b) Grafische Deklaration in der Sprache FBS (siehe 4.3).
c) Gebrauch einer Funktion in ST Sprache.
d) Gebrauch einer Funktion in FBS Sprache (4.3).

Bild 6 – Beispiel für Deklarationen und Gebrauch der Funktion

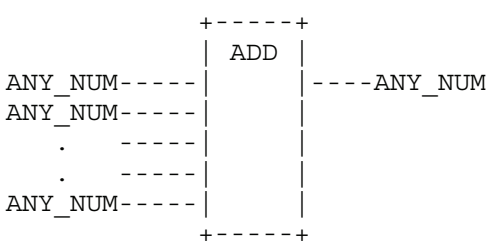
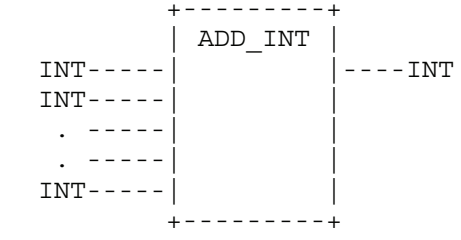
2.5.1.4 Typangabe, Überladen und Typwandlung

Eine Standardfunktion, ein Funktionsbausteintyp, ein Operator oder eine Anweisung nennt man *überladen*, wenn sie mit Eingangsdatenelementen verschiedenen Typs arbeiten kann, und zwar innerhalb eines allgemeinen Typbezeichners, wie in 2.3.2 definiert. Z. B. kann eine überladene Additionsfunktion vom Typ `ANY_NUM` mit Daten vom Typ `LREAL`, `REAL`, `DINT`, `INT` und `SINT` arbeiten.

Wenn ein SPS-System eine überladene Standardfunktion, einen Funktionsbausteintyp, eine Operation oder eine Anweisung unterstützt, muss diese Standardfunktion, dieser Funktionsbausteintyp, diese Operation oder diese Anweisung auf alle Datentypen eines allgemeinen Typs anwendbar sein, die von diesem System unterstützt werden. Wenn z. B. ein SPS-System die überladene Funktion `ADD` und die Datentypen `SINT`, `INT` und `REAL` unterstützt, dann muss das System die Funktion `ADD` mit den Eingängen vom Datentyp `SINT`, `INT` und `REAL` unterstützen.

Wenn für eine Funktion, die normalerweise einen überladenen Operator darstellt, ein bestimmter Typ angegeben werden soll, d. h. wenn die Typen ihrer Eingänge und Ausgänge auf einen bestimmten elementaren oder abgeleiteten Datentyp beschränkt sein sollen, wie in 2.3 definiert, dann muss dies durch Anhängen eines „Unterstrich“-Zeichens getan werden, auf das der gewünschte Typ folgt, wie in Tabelle 21 gezeigt ist.

Tabelle 21 – Typangabe und überladene Funktionen

Nr.	Eigenschaft	Beispiel
1	Überladene Funktionen	
2 [a]	Funktionen mit Typangabe	
ANMERKUNG Das Überladen von Funktionen oder Funktionsbausteinen, die Nicht-Standard sind, liegt außerhalb des Geltungsbereich dieser Norm.		
[a] Falls Eigenschaft 2 unterstützt wird, muss der Hersteller eine Tabelle zur Verfügung stellen, die angibt, welche Funktionen in der Implementierung überladen sind und welche mit dem Typ angegeben werden.		

Wenn der Typ des Ergebnisses einer Standardfunktion, die in 2.5.1.5 definiert ist, ein allgemeiner Typ ist, dann müssen die aktuellen Typen aller Eingangsvariablen desselben allgemeinen Typs vom selben Typ sein, wie der aktuelle Typ des Funktionswertes in einem gegebenen *Aufruf* der Funktion. Falls nötig, können die Typ-Umwandlungsfunktionen, die in 2.5.1.5.1 angegeben sind, benutzt werden, um diese Anforderung zu erfüllen. Beispiele für die Anwendung dieser Regel sind in den Bildern 7 und 8 angegeben.

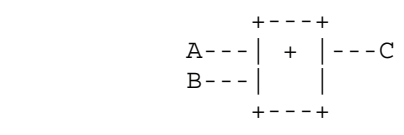
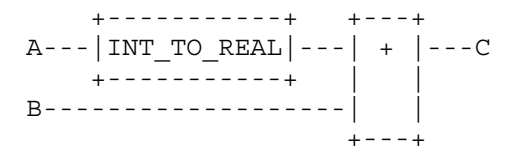
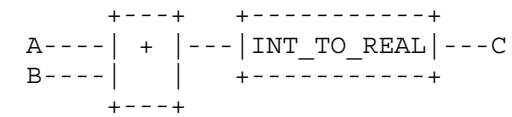
Typdeklaration (ST Sprache – siehe 3.3)	Gebrauch (FBS Sprache – siehe 4.3) (ST Sprache – siehe 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	 C := A+B;
ANMERKUNG Typumwandlung ist in dem oben gezeigten Beispiel nicht erforderlich.	
VAR A : INT ; B : REAL ; C : REAL; END_VAR	 C := INT_TO_REAL(A)+ B;
VAR A : INT ; B : INT ; C : REAL; END_VAR	 C := INT_TO_REAL(A + B) ;

Bild 7 – Beispiele für explizite Typumwandlung bei überladenen Funktionen

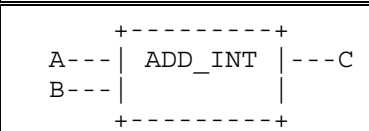
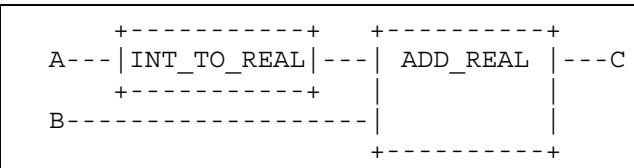
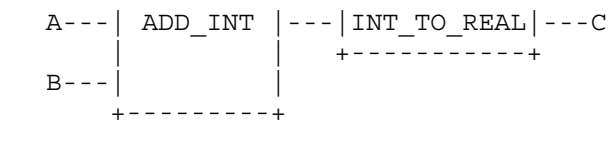
Typdeklaration (ST Sprache – siehe 3.3)	Gebrauch (FBS Sprache – siehe 4.3) (ST Sprache – siehe 3.3)
VAR A : INT ; B : INT ; C : INT ; END_VAR	 C := ADD_INT(A,B) ;
ANMERKUNG Typumwandlung ist im ersten Beispiel nicht erforderlich.	
VAR A : INT ; B : REAL ; C : REAL; END_VAR	 C := ADD_REAL(INT_TO_REAL(A),B);
VAR A : INT ; B : INT ; C : REAL; END_VAR	 C := INT_TO_REAL(ADD_INT(A,B));

Bild 8 – Beispiele von expliziter Typumwandlung bei Funktionen mit Typangabe

2.5.1.5 Standardfunktionen

In diesem Abschnitt sind die Definitionen der Funktionen angegeben, die allen SPS-Programmiersprachen gemeinsam sind. Wo grafische Darstellungen der Standardfunktionen in diesem Abschnitt gezeigt werden, dürfen gleichwertige Deklarationen in Textform geschrieben werden, wie in 2.5.1.3 festgelegt ist.

Eine Standardfunktion, die in diesem Abschnitt als *erweiterbar* festgelegt ist, darf zwei oder mehr Eingänge haben, auf die die angezeigte Operation anzuwenden ist; z. B. muss die erweiterbare Addition an ihrem Ausgang die Summe aller ihrer Eingänge liefern. Die maximale Anzahl von Eingängen einer erweiterbaren Funktion ist ein **implementierungsabhängiger Parameter**. Die aktuelle Anzahl von Eingängen in einem formalen Aufruf einer erweiterbaren Funktion ist durch den formalen Eingangsnamen mit der höchsten Position in der Folge von Parameternamen bestimmt.

BEISPIEL 1 Die Anweisung

```
X := ADD(Y1,Y2,Y3);
```

ist gleichbedeutend mit

```
X := ADD(IN1 := Y1, IN2 := Y2, IN3 := Y3);
```

BEISPIEL 2 Die folgenden Anweisungen sind gleichbedeutend:

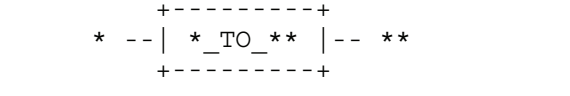
```
I := MUX_INT(K:=3,IN0 := 1, IN2 := 2, IN4 := 3);
I := 0;
```

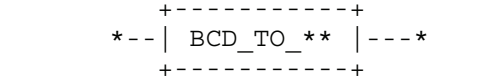
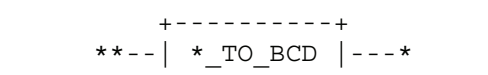
2.5.1.5.1 Funktionen zur Typumwandlung

Wie in Tabelle 22 gezeigt, müssen die Funktionen zur Typumwandlung die Form `*_TO_**` haben, wobei „`*`“ der Typ der Eingangsvariablen `IN` und „`**`“ der Typ der Ausgangsvariablen `OUT` ist; z. B. `INT_TO_REAL`. Die Wirkungen der Typumwandlungen auf die Genauigkeit und die Typen von **Fehlern**, die während der Ausführung von Typumwandlungsoperationen auftreten können, sind **implementierungsabhängige Parameter**.

Tabelle 22 – Funktion zur Typumwandlung

Nr.	Grafische Form	Beispiel

1 [a,b,e]	
	`(*)` Eingangsdatentyp, z. B. `INT`
	`(**)` Ausgangsdatentyp, z. B. `REAL`
	`*_TO_**)` Funktionsname, z. B. `INT_TO_REAL`
		`A := INT_TO_REAL(B) ;`

2 [c]		`A := TRUNC(B) ;`
3 [d]		`A := WORD_BCD_TO_INT(B) ;`
4 [d]		`A := INT_TO_BCD_WORD(B) ;`

ANMERKUNG Die Anwendungsbeispiele sind in der ST-Sprache angegeben, die in 3.3 definiert ist.

[a] Eine Aussage bezüglich Übereinstimmung mit Eigenschaft 1 dieser Tabelle muss eine Liste der spezifischen unterstützten Typumwandlungen und eine Aussage der Wirkungen, die bei der Bearbeitung der jeweiligen Umwandlung auftreten, einschließen.

[b] Die Wandlung vom Typ `REAL` oder `LREAL` nach `SINT`, `INT`, `DINT` oder `LINT` muss nach den Festlegungen in IEC 60559 runden. Danach muss, falls die zwei nächsten ganzen Zahlen gleich nah sind, das Ergebnis die nächste gerade ganze Zahl sein, z. B.:

`REAL_TO_INT (1.6)` ist gleich `2`
`REAL_TO_INT (-1.6)` ist gleich `-2`

`REAL_TO_INT (1.5)` ist gleich `2`
`REAL_TO_INT (-1.5)` ist gleich `-2`

`REAL_TO_INT (1.4)` ist gleich `1`
`REAL_TO_INT (-1.4)` ist gleich `-1`

`REAL_TO_INT (2.5)` ist gleich `3`
`REAL_TO_INT (-2.5)` ist gleich `-3`

[c] Die Funktion `TRUNC` muss zum Abschneiden Richtung Null von `REAL` oder `LREAL` benutzt werden; dabei wird einer der ganzzahligen Typen geliefert; z. B.:

`TRUNC (1.6)` ist gleich `1`
`TRUNC (-1.6)` ist gleich `–1`

`TRUNC (1.4)` ist gleich `1`
`TRUNC (-1.4)` ist gleich `-1`

[d] Die Wandlungsfunktionen `*_BCD_TO_**` und `**TO_BCD_TO_*` müssen die Umwandlungen zwischen Variablen vom Typ `BYTE`, `WORD`, `DWORD` und `LWORD` und Variablen vom Typ `USINT`, `UINT`, `DINT` und `ULINT` durchführen (durch „`*`“ bzw. „`**`“ dargestellt), wenn die entsprechenden Bit-Folge-Variablen Daten beinhalten, die in BCD-Format kodiert sind. Z. B. wird der Wert von `USINT_TO_BCD_BYTE(25)` gleich `2#0010_0101` sein, und der Wert von `WORD_BCD_TO_UINT (2#0011_0110_1001)` wird gleich `369` sein.

[e] Wenn ein Eingang oder Ausgang einer Typ-Wandlungsfunktion vom Typ `STRING` oder `WSTRING` ist, müssen die Daten der Zeichenfolge mit der äußeren Darstellung der entsprechenden Daten übereinstimmen, wie es in 2.2 festgelegt ist, im Zeichensatz in 2.1.1 definiert.

2.5.1.5.2 Numerische Funktionen

Die standardmäßige grafische Darstellung, die Funktionsnamen, die Typen der Eingangs- und Ausgangsvariablen und die Funktionsbeschreibungen der Funktionen mit einer einzelnen numerischen Variablen müssen nach Tabelle 23 angegeben werden. Diese Funktionen müssen mit den definierten allgemeinen Typen überladen sein und sie können mit Typangabe nach 2.5.1.4 definiert sein. Für diese Funktionen müssen die Typen von Eingang und Ausgang gleich sein.

Die standardmäßige grafische Darstellung, die Funktionsnamen und -Symbole und die Beschreibung der arithmetischen Funktionen mit zwei oder mehr Variablen müssen nach Tabelle 24 angegeben werden. Diese Funktionen müssen mit allen numerischen Typen überladen sein und können mit Typen angegeben werden, wie in 2.5.1.4 definiert ist.

Die Genauigkeit von numerischen Funktionen muss mittels eines oder mehrerer **implementierungsabhängiger Parameter** ausgedrückt werden.

Es ist ein **Fehler**, wenn das Ergebnis der Auswertung einer dieser Funktionen den Wertebereich überschreitet, der für den Datentyp des Funktionsausgangs festgelegt ist oder wenn eine Division durch Null versucht wird.

Tabelle 23 – Standardfunktionen mit einer numerischen Variablen

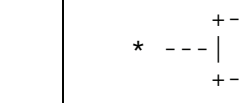
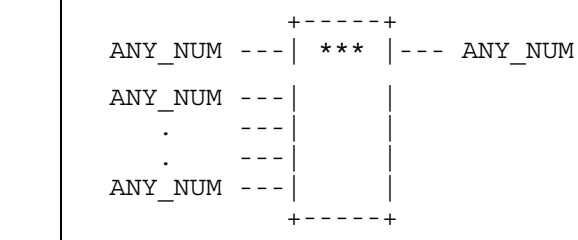
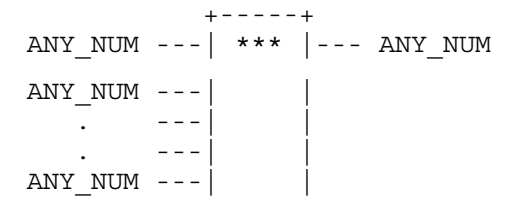
Grafische Form			Anwendungsbeispiel
 (*) Eingang/Ausgang (E/A)-Typ (**) Funktionsname			A := SIN(B) ; (ST Sprache – siehe 3.3)
Nr.	**Funktionsname**	**E/A Typ**	**Beschreibung**
	Allgemeine Funktionen		
1	ABS	ANY_NUM	Absolutwert
2	SQRT	ANY_REAL	Quadratwurzel
	Logarithmische Funktionen		
3	LN	ANY_REAL	Natürlicher Logarithmus
4	LOG	ANY_REAL	Logarithmus zur Basis 10
5	EXP	ANY_REAL	Exponentialfunktion
	Trigonometrische Funktionen		
6	SIN	ANY_REAL	Sinus, mit Eingang im Bogenmaß
7	COS	ANY_REAL	Cosinus
8	TAN	ANY_REAL	Tangens
9	ASIN	ANY_REAL	Arcsin, Hauptwert
10	ACOS	ANY_REAL	Arccos, Hauptwert
11	ATAN	ANY_REAL	Arctan, Hauptwert

Tabelle 24 – Standardmäßige arithmetische Funktionen

Grafische Form				Anwendungsbeispiel
 (***) Name oder Symbol				A := ADD(B,C,D) ; oder A := B+C+D ;
Nr. [d,e]	**Name**	**Symbol**	**Beschreibung**	
	Erweiterbare arithmetische Funktionen			
12[g]	ADD	+	OUT := IN1 + IN2 + ... + INn	
13	MUL	*	OUT := IN1 * IN2 * ... * INn	
	Nicht-erweiterbare arithmetische Funktionen			
14[g]	SUB	-	OUT := IN1 - IN2	
15[c]	DIV	/	OUT := IN1 / IN2	
16[a]	MOD		OUT := IN1 modulo IN2	
17[b]	EXPT	**	Exponentiation: OUT := $IN1^{IN2}$	

Grafische Form	Anwendungsbeispiel
 (***) Name oder Symbol	A := ADD(B,C,D) ; oder A := B+C+D ;

Nr. [d,e]	Name	Symbol	Beschreibung
18[f]	MOVE	:=	OUT := IN

ANMERKUNG 1 Nicht leere Einträge in der **Symbol**-Spalte sind für den Gebrauch als Operatoren in textuellen Sprachen verwendbar, wie es in den Tabellen 52 und 55 gezeigt ist.

ANMERKUNG 2 Die Angaben IN1, IN2, ..., INn beziehen sich auf die Eingänge in der Reihenfolge von oben nach unten; OUT bezieht sich auf den Ausgang.

ANMERKUNG 3 Anwendungsbeispiele und Beschreibungen sind in der ST-Sprache angegeben, die in 3.3 definiert ist.

[a] IN1 und IN2 müssen für diese Funktion den allgemeinen Typ ANY_INT haben. Das Ergebnis der Bearbeitung dieser Funktion ist gleich der Bearbeitung der folgenden Anweisungen in der Sprache ST, wie in 3.3 definiert:

```
IF (IN2 = 0) THEN OUT := 0 ;  ELSE OUT := IN1 - (IN1/IN2) * IN2 ;
END_IF
```

[b] Bei dieser Funktion muss IN1 den Typ ANY_REAL und IN2 den Typ ANY_NUM haben. Der Ausgang muss denselben Typ wie IN1 haben.

[c] Das Ergebnis der Division von ganzen Zahlen muss eine ganze Zahl vom selben Typ sein, mit Abschneiden Richtung Null, z. B. 7/3 = 2 und (-7) / 3 = -2.

[d] Wenn die Darstellung einer Funktion mit Namensangabe unterstützt wird, muss dies durch das Suffix „n“ in der Aussage zur Normerfüllung angegeben werden, z. B. repräsentiert „12n“ die Schreibweise „ADD“.

[e] Wenn die Darstellung einer Funktion mit Symbol unterstützt wird, muss dies durch das Suffix „s“ in der Aussage zur Normerfüllung angegeben werden, z. B. repräsentiert „12s“ die Schreibweise „+“.

[f] Die Funktion MOVE hat genau einen Eingang (IN) vom Type ANY und einen Ausgang (OUT) vom Typ ANY.

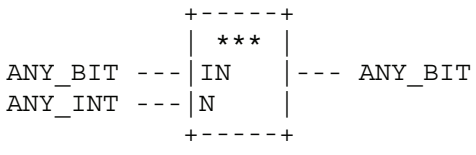
[g] Der allgemeine Typ der Eingänge und Ausgänge dieser Funktionen ist ANY_MAGNITUDE.

2.5.1.5.3 Bit-Folge-Funktionen

Die standardmäßige grafische Darstellung, die Funktionsnamen und Beschreibungen von Schiebefunktionen für eine einzelne Bit-Folge-Variable müssen angegeben werden, wie es in Tabelle 25 definiert ist. Diese Funktionen müssen mit allen Bit-Folge-Typen überladen sein und sie können mit Typen angegeben werden, wie in 2.5.1.4 definiert ist.

Die standardmäßige grafische Darstellung, die Funktionsnamen, Symbole und Beschreibungen der Bit-weisen booleschen Funktionen müssen angegeben werden, wie es in Tabelle 26 definiert ist. Diese Funktionen müssen erweiterbar sein, ausgenommen ist NOT, und sie müssen mit allen Bit-Folge-Typen überladen sein. Sie können mit Typ angegeben werden, wie es in 2.5.1.4 definiert ist.

Tabelle 25 – Standardmäßige Bitschiebe-Funktionen

Grafische Form	Anwendungsbeispiele [a]
 (***) Funktionsname	A := SHL(IN:=B, N:=5) ; (ST Sprache – siehe 3.3)

Nr.	Name	Beschreibung
1	SHL	OUT := IN links schieben um N Bits, rechts mit null füllen
2	SHR	OUT := IN rechts schieben um N Bits, links mit null füllen
3	ROR	OUT := IN rechts rotieren um N Bits, im Kreis
4	ROL	OUT := IN links rotieren um N Bits, im Kreis
ANMERKUNG	Die Angabe OUT bezieht sich auf den Funktionsausgang.	
[a] Es muss ein **Fehler** sein, wenn der Wert des Eingangs N kleiner als null ist.		

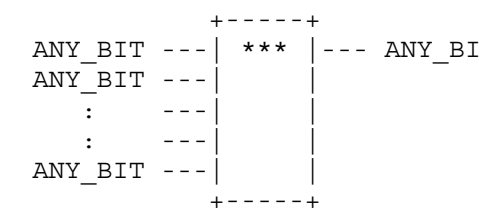
2.5.1.5.4 Funktionen für Auswahl und Vergleich

Die Funktionen für Auswahl und Vergleich müssen mit allen Datentypen überladen sein. Die standardmäßigen grafischen Darstellungen, Funktionsnamen und Beschreibungen von Auswahlfunktionen müssen nach Tabelle 27 erfolgen.

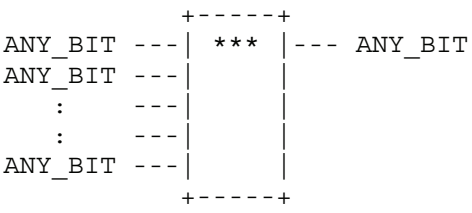
Die standardmäßige grafische Darstellung, die Funktionsnamen und Symbole und die Beschreibungen der Vergleichsfunktionen müssen nach Tabelle 28 erfolgen. Alle Vergleichsfunktionen (ausgenommen NE) müssen erweiterbar sein.

Vergleiche von Bit-Folgen müssen bitweise vom höchstwertigen zum niederwertigen Bit erfolgen, und kürzere Bit-Folgen müssen betrachtet werden, als seien sie links mit Nullen aufgefüllt, wenn sie mit längeren Bit-Folgen verglichen werden, d. h. der Vergleich von Bitvariablen muss das gleiche Ergebnis haben, wie der Vergleich von vorzeichenlosen ganzzahligen Variablen.

Tabelle 26 – Bitweise boolesche Standardfunktionen

Grafische Form	Beispiel
 (***) Name oder Symbol	A := AND(B,C,D) ; oder A := B & C & D ;

Nr. [a,b]	Name	Symbol	Beschreibung
5	AND	& (ANMERKUNG 1)	OUT := IN1 & IN2 & ... & INn
6	OR	>=1 (ANMERKUNG 2)	OUT := IN1 OR IN2 OR ... OR INn
7	XOR	=2k+1 (ANMERKUNG 2)	OUT := IN1 XOR IN2 XOR ... XOR INn
8	NOT		OUT := NOT IN1 (ANMERKUNG 4)

Grafische Form	Beispiel
 (***) Name oder Symbol	A := AND(B,C,D) ; oder A := B & C & D ;

Nr. [a,b]	Name	Symbol	Beschreibung

ANMERKUNG 1 Dieses Symbol ist für den Gebrauch als Operator in Textsprachen geeignet, wie in den Tabellen 52 und 55 gezeigt.

ANMERKUNG 2 Dieses Symbol ist nicht für den Gebrauch als Operator in Textsprachen geeignet.

ANMERKUNG 3 Die Angabe IN1, IN2, ..., INn bezieht sich auf die Eingänge in der Reihenfolge von oben nach unten; OUT bezieht sich auf den Ausgang.

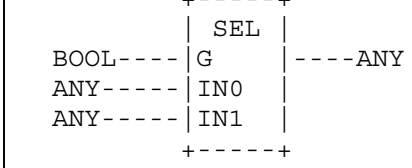
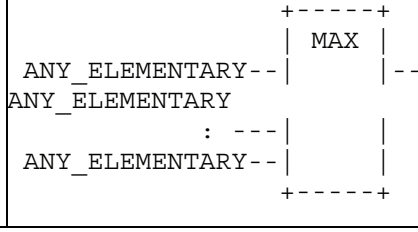
ANMERKUNG 4 Die grafische Negation von Signalen mit dem Typ BOOL kann auch, wie es in Tabelle 19 gezeigt ist, erreicht werden.

ANMERKUNG 5 Der Gebrauch von Beispielen und die Beschreibungen sind in der ST-Sprache angegeben, die in 3.3 definiert ist.

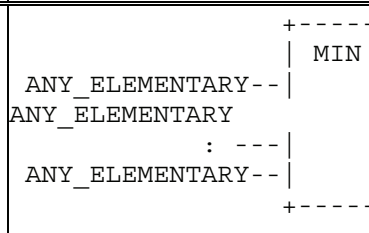
[a] Wenn die Darstellung einer Funktion mit Namensangabe unterstützt wird, muss dies durch das Suffix „n“ in der Aussage zur Normerfüllung gezeigt werden. Z. B. stellt „5n“ die Angabe „AND“ dar.

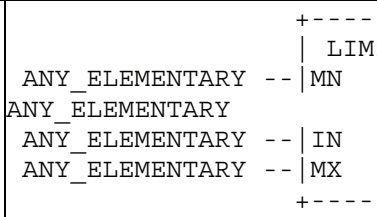
[b] Wenn die Darstellung einer Funktion mit Symbolangabe unterstützt wird, muss dies durch das Suffix „s“ in der Aussage zur Normerfüllung gezeigt werden. Z. B. stellt „5s“ die Angabe „&“ dar.

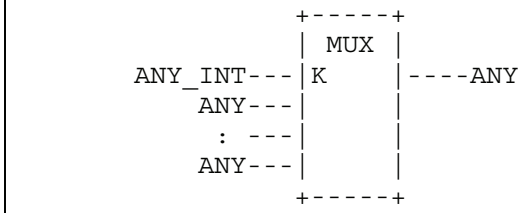
Tabelle 27 – Standardfunktionen für Auswahl [d]

Nr.	Grafische Form	Erläuterung/Beispiel
1		Binäre Auswahl [c]: OUT := IN0 falls G = 0 OUT := IN1 falls G = 1 BEISPIEL: A := SEL(G:=0, IN0:=X, IN1:=5);
2a		Erweiterbare Maximum Funktion: OUT := MAX (IN1, IN2, ...,INn) BEISPIEL: A := MAX(B,C,D) ;

Nr.	Grafische Form	Erläuterung/Beispiel

2b		Erweiterbare Minimum Funktion:
		`OUT := MIN (IN1, IN2, ...,INn)`
		BEISPIEL:
		`A := MIN(B,C,D) ;`

3		Begrenzer:
		`OUT := MIN(MAX(IN, MN),MX)`
		BEISPIEL:
		`A := LIMIT(IN:=B,MN:=0,MX:=5);`

4[e]		Erweiterbarer Multiplexer [a, b, c]:
		Wählt einen von `N` Eingängen aus abhängig vom Eingang `K`
		BEISPIEL:
		`A :=MUX(0, B, C, D);`
		würde dieselbe Wirkung haben wie
		`A := B ;`

ANMERKUNG 1 Die Angaben `IN1, IN2, ..., INn` beziehen sich in der Reihenfolge von oben nach unten auf die Eingänge; `OUT` bezieht sich auf den Ausgang.

ANMERKUNG 2 Die Anwendungsbeispiele und Beschreibungen sind in der ST Sprache angegeben, die in 3.3 definiert ist.

[a] Die Eingänge ohne Namen in der `MUX` Funktion müssen die voreingestellten Namen `IN0`, `IN1`, `...`, `INn-1` in der Reihenfolge von oben nach unten haben, wobei `n` die Gesamtanzahl der Eingänge ist. Diese Namen dürfen, aber müssen nicht, in der grafischen Darstellung gezeigt werden.

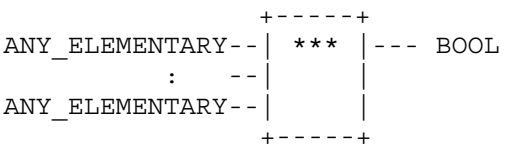
[b] Die `MUX` Funktion kann mit *Typ* angegeben werden, wie es in 2.5.1.4 in der Form `MUX_*_**` definiert ist, wobei `*` der Typ der Eingangs `K` und `**` der Typ der anderen Eingänge und Ausgänge ist.

[c] Es ist erlaubt, aber nicht verlangt, dass der Hersteller eine Auswahl unter Variablen vom *abgeleiteten Datentyp* unterstützt, wie es in 2.3.3 definiert ist, um eine Normerfüllung bei dieser Eigenschaft zu beanspruchen.

[d] Es ist ein **Fehler**, wenn die Eingänge und Ausgänge von einer dieser Funktionen nicht alle vom gleichen aktuellen Datentyp sind, mit Ausnahme des Eingangs `G` der `SEL` Funktion und des Eingangs `K` der `MUX` Funktion.

[e] Es ist ein **Fehler**, wenn der aktuelle Wert des Eingangs `K` der `MUX` Funktion nicht innerhalb des Bereichs `{0..n-1}` liegt.

Tabelle 28 – Standardfunktionen für Vergleich

Grafische Form	Beispiele
 (***) Name oder Symbol	A := GT(B,C,D) ; oder A := (B>C) & (C>D) ;

Nr.	Name [a]	Symbol [b]	Beschreibung
5	GT	>	Fallende Folge: OUT := (IN1>IN2) & (IN2>IN3) & ... & (INn-1 > INn)
6	GE	>=	Monotone Folge: OUT := (IN1>=IN2) & (IN2>=IN3) & ... & (INn-1 >= INn)
7	EQ	=	Gleichheit: OUT := (IN1=IN2) & (IN2=IN3) & ... & (INn-1 = INn)
8	LE	<=	Monotone Folge: OUT := (IN1<=IN2) & (IN2<=IN3) & ... & (INn-1 <= INn)
9	LT	<	Steigende Folge: OUT := (IN1<IN2) & (IN2<IN3) & ... & (INn-1 < INn)
10	NE	<>	Ungleichheit (nicht erweiterbar): OUT := (IN1 <> IN2)

ANMERKUNG 1 Die Angaben IN1, IN2, ..., INn beziehen sich in der Reihenfolge von oben nach unten auf die Eingänge; OUT bezieht sich auf den Ausgang.

ANMERKUNG 2 Alle Symbole, die in dieser Tabelle gezeigt sind, sind für den Gebrauch als Operatoren in Textsprachen geeignet, wie in den Tabellen 52 und 55 gezeigt ist.

ANMERKUNG 3 Der Gebrauch der Beispiele und die Beschreibungen sind in der ST Sprache angegeben, die in 3.3 definiert ist.

[a] Wenn die Darstellung einer Funktion mit Namensangabe unterstützt wird, muss dies durch den Suffix „n“ in der Aussage zur Normerfüllung angegeben werden. Zum Beispiel stellt „5n“ die Angabe „GT“ dar.

[b] Wenn die Darstellung einer Funktion als Symbol unterstützt wird, muss dies durch das Suffix „s“ in der Aussage zur Normerfüllung angegeben werden. Zum Beispiel stellt „5s“ die Angabe „>“ dar.

2.5.1.5.5 Funktionen für Zeichenfolge

Alle Funktionen, die in 2.5.1.5.4 definiert sind, müssen auf Zeichenfolgen anwendbar sein. Zum Zweck des Vergleiches von zwei Folgen ungleicher Länge muss die kürzere Folge betrachtet werden, als wäre sie rechts auf die Länge der längeren Folge um Zeichen mit dem Wert Null erweitert. Der Vergleich muss von links nach rechts erfolgen, basierend auf den numerischen Werten der Zeichen-Codes, der in 2.1.1 definiert ist. Die Zeichenfolge 'Z' muss z. B. größer sein als die Zeichenfolge 'AZ', und 'AZ' muss größer sein als 'ABC'.

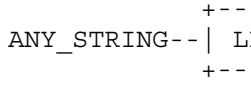
Die standardmäßigen grafischen Darstellungen, Funktionsnamen und Beschreibungen von zusätzlichen Funktionen für Zeichenfolgen müssen erfolgen, wie es in Tabelle 29 definiert ist. Zum Zweck dieser Operationen müssen die Zeichenpositionen innerhalb der Folge als nummeriert mit 1, 2, ..., L betrachtet werden, beginnend mit der linken Zeichenposition, wobei L die Länge der Folge darstellt.

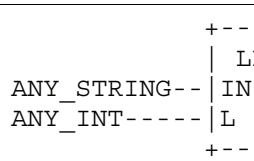
Es muss ein **Fehler** sein, wenn:

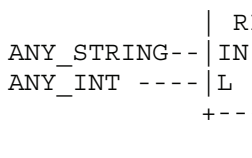
– der aktuelle Wert eines Eingangs, der als ANY_INT in Tabelle 29 bezeichnet ist, kleiner als Null ist;

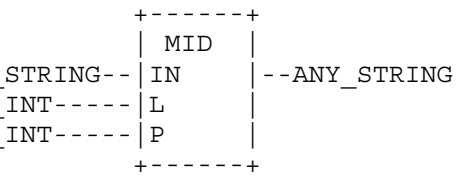
– die Auswertung der Funktionsergebnisse in einem Versuch (1) auf eine nicht vorhandene Zeichenposition in einer Folge zuzugreifen oder (2) eine Folge zu erzeugen, die länger als die implementierungsabhängige maximale Länge der Zeichenfolge ist.

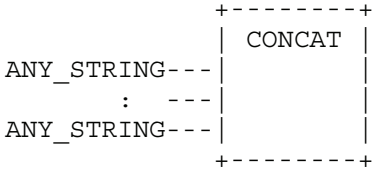
Tabelle 29 – Standardfunktionen für Zeichenfolgen

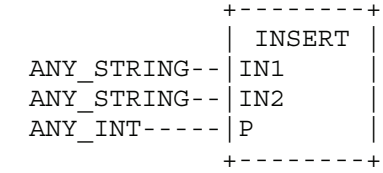
Nr.	Grafische Form[a]	Erläuterung/Beispiele
1		Funktion Länge der Zeichenfolge Beispiel: `A := LEN('ASTRING');` ist gleichwertig mit `A := 7;`

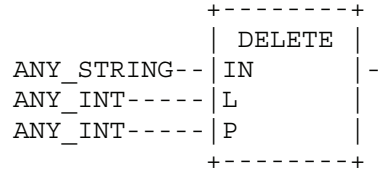
2		Linksstehende L Zeichen von IN
		Beispiel:
		`A := LEFT(IN:='ASTR', L:=3);`
		ist gleichwertig mit
		`A := 'AST' ;`

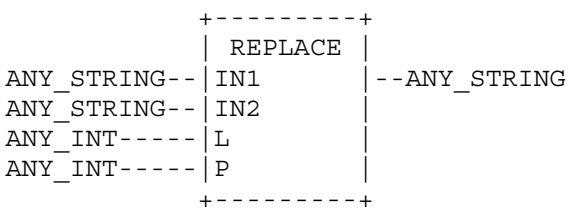
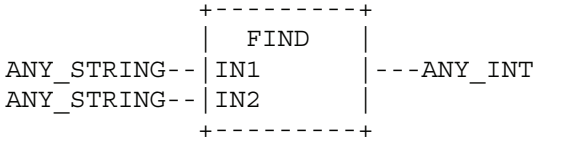
3		Rechtsstehende L Zeichen von IN
		Beispiel:
		`A := RIGHT(IN:='ASTR', L:=3);`
		ist gleichwertig mit
		`A := 'STR' ;`

4		L Zeichen von IN, beginnend mit dem P-ten
		Beispiel:
		`A := MID(IN:='ASTR',L:=2,P:=2);`
		ist gleichwertig mit
		`A := 'ST' ;`

5		Erweiterbare Aneinanderreihung
		Beispiel:
		`A := CONCAT('AB','CD', E') ;`
		ist gleichwertig mit
		`A := 'ABCDE' ;`

6		Fügt IN2 in IN1 nach der P-ten Zeichenposition ein
		Beispiel:
		`A:=INSERT(IN1:='ABC', IN2:='XY', P=2);`
		ist gleichwertig mit
		`A := 'ABXYC' ;`

7		Löscht L Zeichen von IN, beginnend mit der P-ten Zeichenposition
		Beispiel:
		`A := DELETE(IN:='ABXYC', L:=2, P:=3) ;`
		ist gleichwertig mit
		`A := 'ABC' ;`

Nr.	Grafische Form[a]	Erläuterung/Beispiele
8		Ersetzt L Zeichen von IN1 durch IN2, beginnend ab der P-ten Zeichenposition Beispiel: A := REPLACE(IN1:='ABCDE', IN2:='X', L:=2, P:=3) ; ist gleichwertig mit A := 'ABXE' ;
9		Ermittelt die Zeichenposition des Anfangs für das erste Auftreten von IN2 in IN1. Falls kein Auftreten von IN2 gefunden wird, dann ist OUT := 0. Beispiel: A := FIND(IN1:='ABCBC', IN2:='BC') ; ist gleich A := 2 ;
ANMERKUNG Die Beispiele in dieser Tabelle sind in der Sprache Strukturierter Text (ST) angegeben, die in 3.3 definiert ist.		

2.5.1.5.6 Funktionen für Datentypen der Zeit

Zusätzlich zu den Funktionen für Vergleich und Auswahl, die in Abschnitt 2.5.1.5.4 definiert sind, müssen die Kombinationen von Eingang und Ausgang der Datentypen der Zeit, die in Tabelle 30 gezeigt sind, bei den zugehörigen Funktionen erlaubt sein.

Es muss ein **Fehler** sein, wenn das Ergebnis der Auswertung einer dieser Funktionen den **implementierungsabhängigen** Bereich der Werte für den Ausgangsdatentyp überschreitet.

Tabelle 30 – Funktionen für Datentypen der Zeit

	Numerische und Aneinanderreihungsfunktionen				
Nr.	**Name**	**Symbol**	**IN1**	**IN2**	**OUT**
1a [c,d]	ADD	+	TIME	TIME	TIME
1b [c,d]	ADD_TIME	+	TIME	TIME	TIME
2a	ADD [b]	+ [b]	TIME_OF_DAY	TIME	TIME_OF_DAY
2b	ADD_TOD_TIME	+ [b]	TIME_OF_DAY	TIME	TIME_OF_DAY
3a	ADD [b]	+ [b]	DATE_AND_TIME	TIME	DATE_AND_TIME
3b	ADD_DT_TIME	+ [b]	DATE_AND_TIME	TIME	DATE_AND_TIME
4a [c,d]	SUB	-	TIME	TIME	TIME
4b [c,d]	SUB_TIME	-	TIME	TIME	TIME
5a	SUB [b]	- [b]	DATE	DATE	TIME
5b	SUB_DATE_DATE	- [b]	DATE	DATE	TIME
6a	SUB [b]	- [b]	TIME_OF_DAY	TIME	TIME_OF_DAY
6b	SUB_TOD_TIME	- [b]	TIME_OF_DAY	TIME	TIME_OF_DAY
7a	SUB [b]	- [b]	TIME_OF_DAY	TIME_OF_DAY	TIME
7b	SUB_TOD_TOD	- [b]	TIME_OF_DAY	TIME_OF_DAY	TIME
8a	SUB [b]	- [b]	DATE_AND_TIME	TIME	DATE_AND_TIME

	Numerische und Aneinanderreihungsfunktionen				
Nr.	**Name**	**Symbol**	**IN1**	**IN2**	**OUT**
8b	SUB_DT_TIME	- [b]	DATE_AND_TIME	TIME	DATE_AND_TIME
9a	SUB [b]	- [b]	DATE_AND_TIME	DATE_AND_TIME	TIME
9b	SUB_DT_TIME	- [b]	DATE_AND_TIME	DATE_AND_TIME	TIME
10a	MUL [b]	* [b]	TIME	ANY_NUM	TIME
10b	MULTIME	* [b]	TIME	ANY_NUM	TIME
11a	DIV [b]	/ [b]	TIME	ANY_NUM	TIME
11b	DIVTIME	/ [b]	TIME	ANY_NUM	TIME
12	CONCAT_DATE_TOD		DATE	TIME_OF_DAY	DATE_AND_TIME
	Funktionen für Typwandlung				
13 [a]	DT_TO_TOD				
14 [a]	DT_TO_DATE				

ANMERKUNG 1 Nicht-leere Einträge in der **Symbol**-Spalte sind für den Gebrauch als Operatoren in Textsprachen, wie in Tabellen 52 und 55 gezeigt, anwendbar.

ANMERKUNG 2 Die Schreibweise IN1, IN2, ..., INn verweist auf die Eingänge in der Anordnung von oben nach unten; OUT verweist auf den Ausgang.

ANMERKUNG 3 Es ist möglich, bei den Funktionen MULTIME und DIVTIME den Typ anzugeben; z. B. würden die Operatoren von MULTIME_REAL den Typ TIME bzw. REAL haben.

ANMERKUNG 4 Die Wirkungen der Wandlung zwischen Zeitdatentypen und Typen STRING und WSTRING sind in der Fußnote (e) von Tabelle 22 definiert.

ANMERKUNG 5 Die Wirkungen der Typwandlungen zwischen Zeitdatentypen und anderen Datentypen, die nicht in dieser Tabelle definiert sind, sind **implementierungsabhängig**.

[a] Die Funktionen zur Typwandlung müssen die Wirkung des „Herausgreifens“ von entsprechenden Daten haben. Z. B. müssen die Anweisungen in ST Sprache

```
X := DT#1986-04-28-08:40:00 ;
Y := DT_TO_TOD(X) ;
W := DT_TO_DATE(X) ;
```

dasselbe Ergebnis haben, wie die Anweisungen

```
X := DT#1986-04-28-08:40:00 ;
W := DATE#1986-04-28 ;
Y := TIME_OF_DAY#08:40:00;
```

[b] Von diesem Gebrauch wird **abgeraten**. Er wird in zukünftigen Ausgaben dieser Norm nicht enthalten sein.

[c] Wenn die Darstellung einer Funktion mit **N**amen unterstützt wird, muss dies durch das Suffix „n“ in der Normerfüllungserklärung angegeben werden, z. B. stellt „1n“ die Angabe „ADD“ dar.

[d] Wenn die Darstellung einer Funktion mit **S**ymbol unterstützt wird, muss dies durch das Suffix „s“ in der Normerfüllungserklärung angegeben werden, z. B. stellt „1s“ die Angabe „+“ dar.

2.5.1.5.7 Funktionen für Datentypen der Aufzählung

Die Funktionen für Auswahl und Vergleich, die in Tabelle 31 aufgezählt sind, können mit Eingängen verwendet werden, die einen Datentyp Aufzählung haben, die in 2.3.3.1 definiert ist.

Tabelle 31 – Funktionen für Datentypen der Aufzählung

Nr.	Name	Symbol	Nummer der Eigenschaft in Tabellen 27 und 28
1	SEL		1
2	MUX		4
3 [a]	EQ	=	7
4 [a]	NE	<>	10
ANMERKUNG Für diese Tabelle gelten die Bestimmungen der ANMERKUNGEN 1 und 2 von Tabelle 28.			
[a] Für diese Eigenschaft gelten die Bestimmungen der Fußnoten a und b von Tabelle 28.			

2.5.2 Funktionsbausteine

Für die Anwendungen in den SPS-Sprachen ist ein *Funktionsbaustein* eine Programm-Organisationseinheit, die bei der Ausführung einen oder mehrere Werte liefert. Es können mehrere mit Namen versehene *Instanzen* (Kopien) eines Funktionsbausteins erzeugt werden. Jede Instanz muss einen zugehörigen Bezeichner (*Instanz-Name*) und eine Datenstruktur besitzen, die ihre Ausgangs- und Internvariablen beinhaltet und abhängig von der Implementierung der Werte oder Verweise auf ihre Eingangsparameter ist. Alle Werte der Ausgangsvariablen und die notwendigen internen Variablen dieser Datenstruktur müssen von einer Bearbeitung des Funktionsbausteins zur nächsten erhalten bleiben; deshalb muss der Aufruf eines Funktionsbausteins mit denselben Argumenten (Eingangsparametern) nicht immer dieselben Ausgangswerte liefern.

Nur die Eingangs- und Ausgangsparameter dürfen von außerhalb einer Instanz eines Funktionsbausteins erreichbar sein; d. h. die internen Variablen des Funktionsbausteins müssen für den Anwender des Funktionsbausteins verborgen sein.

Die Ausführung der Operationen eines Funktionsbausteins muss aufgerufen werden, wie dies in Abschnitt 3 für die Textsprachen definiert ist, und zwar nach den Regeln der Netzwerkauswertung, die in Abschnitt 4 für grafische Sprachen angegeben sind oder unter der Kontrolle von Elementen zur Ablaufsprache (AS), wie es in 2.6 definiert ist.

Jeder Funktionsbausteintyp, der bereits deklariert wurde, kann in der Deklaration eines weiteren Funktionsbausteintyps oder Programmtyps, wie in Bild 3 gezeigt, benutzt werden.

Der Anwendungsbereich einer Instanz eines Funktionsbausteins muss lokal zu der Programm-Organisationseinheit sein, in der sie instanziiert wird, sofern sie nicht als global in einem VAR_GLOBAL Abschnitt deklariert ist, wie es in 2.7.1 definiert ist.

Wie in 2.5.2.2 veranschaulicht, kann der Instanz-Name einer Funktionsbaustein-Instanz als Eingang für eine Funktion oder einen Funktionsbaustein benutzt werden, wenn er in einer VAR_INPUT Deklaration als Eingangsvariable deklariert ist oder als eine Eingangs-/Ausgangsvariable eines Funktionsbausteins in einer Deklaration VAR_IN_OUT, wie in 2.4.3 definiert, benutzt werden.

Die maximale Anzahl von Funktionsbausteintypen und Instanziierungen für eine bestimmte *Ressource* sind **implementierungsabhängige Parameter**.

2.5.2.1 Darstellung

Wie in Bild 9 veranschaulicht, kann eine Instanz eines Funktionsbausteins *textuell* erzeugt werden, indem man ein Datenelement durch Verwenden eines bereits deklarierten Funktionsbausteintyps in der VAR...END_VAR Konstruktion deklariert. Dies geschieht in der gleichen Weise wie der Gebrauch eines strukturierten Datentyps, wie in 2.4.3 definiert ist.

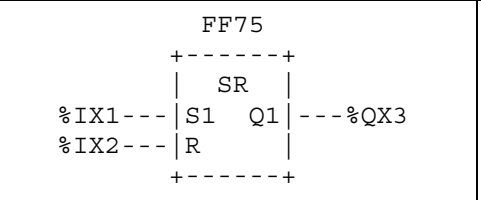
Wie des Weiteren in Bild 9 veranschaulicht, kann eine Instanz eines Funktionsbausteins *grafisch* erzeugt werden, indem man eine grafische Darstellung eines Funktionsbausteins mit dem Namen des Funktionsbau-

steintyps innerhalb des Blocks und dem Instanznamen oberhalb des Blocks benutzt. Dabei folgt man den Darstellungsregeln von Funktionen, die in 2.5.1.1 angegeben sind.

Wie in Bild 9 gezeigt, können die Eingangs- und Ausgangsvariablen einer Instanz eines Funktionsbausteins als Elemente von strukturierten Datentypen dargestellt werden, wie es in 2.3.3.1 definiert ist.

Wenn eine der beiden Eigenschaften zur grafischen Negation, die in Tabelle 19 gezeigt sind, für Funktionsbausteine unterstützt wird, dann muss sie auch für Funktionen unterstützt werden, die in 2.5.1 definiert sind und umgekehrt.

Grafische (FBS-Sprache)	Textform (ST-Sprache)

	`VAR FF75: SR; END_VAR (* Deklaration *)`
	`FF75 (S1:=%IX1, R:=%IX2);(* Aufruf *)`
	`%QX3 := FF75.Q1; (* Ausgangszuweisung *)`

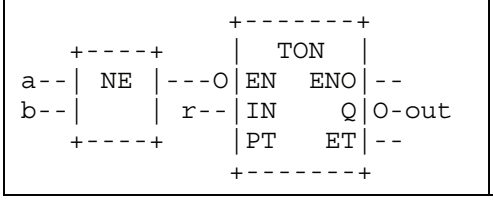
	`VAR a, b, r, out : BOOL; MyTon : TON; END_VAR`
	`MyTon(EN := NOT (a <> b),`
	`IN := r,`
	`NOT Q => out) ;`

Bild 9 – Beispiele für Instanziierung eines Funktionsbausteins

Die Zuweisung eines Wertes an eine Ausgangsvariable eines Funktionsbausteins ist unzulässig, außer von innerhalb des Funktionsbausteins. Die Zuweisung eines Wertes an einen Eingang eines Funktionsbausteins ist nur als Teil des Aufrufs des Funktionsbausteins gestattet. Nicht zugewiesene oder nicht verbundene Eingänge eines Funktionsbausteins müssen ihre initialisierten Werte oder gegebenenfalls die Werte vom letzten vorhergehenden Aufruf behalten. Die zulässigen Anwendungen von Eingängen und Ausgängen von Funktionsbausteinen sind in Tabelle 32 zusammengefasst. Die Beispiele sind in der ST-Sprache gezeigt.

Tabelle 32 – Beispiele für den Gebrauch der E/A-Variablen von Funktionsbausteinen

Gebrauch	Im Funktionsbaustein	Außerhalb des FB
Eingang lesen	`IF IN1 THEN ...`	Nicht zulässig (Anmerkungen 1 und 2)
Eingang zuweisen	Nicht zulässig (Anmerkung 1)	`FB_INST(IN1:=A, IN2:=B);`
Ausgang lesen	`OUT := OUT AND NOT IN2;`	`C := FB_INST.OUT;`
Ausgang zuweisen	`OUT := 1;`	Nicht zulässig (Anmerkung 1)
Ein-Aus lesen	`IF INOUT THEN...`	`IF FB1.INOUT THEN …`
Ein-Aus zuweisen	`INOUT := OUT OR IN1;` (Anmerkung 3)	`FB_INST(INOUT:=D);`

ANMERKUNG 1 Die Anwendungen, die als „nicht zulässig“ in dieser Tabelle angegeben sind, könnten zu implementierungsabhängigen, unvorhersehbaren Seiteneffekten führen.

ANMERKUNG 2 Das Lesen und Schreiben von Eingangs-, Ausgangs- und internen Variablen eines Funktionsbausteins kann mittels der „Kommunikationsfunktion“, „Bediener-Schnittstellenfunktion“ oder den „Funktionen für Programmierung, Testen und Überwachen“ durchgeführt werden, definiert in IEC 61131-1.

ANMERKUNG 3 Wie in 2.5.2.2 veranschaulicht, ist ein Ändern einer Variablen, die in einem `VAR_IN_OUT` Block deklariert ist, innerhalb des Funktionsbausteins erlaubt.

2.5.2.1 a) Der Gebrauch von EN und ENO in Funktionsbausteinen

Wie in Tabelle 20 für Funktionen gezeigt, kann auch für Funktionsbausteine ein zusätzlicher boolescher Eingang EN (Enable) oder Ausgang ENO (Enable out) oder beide vom Hersteller oder Anwender entsprechend den folgenden Deklarationen zur Verfügung gestellt werden:

```
VAR_INPUT   EN:  BOOL := 1;  END_VAR
VAR_OUTPUT  ENO:BOOL;        END_VAR
```

Wenn diese Variablen verwendet werden, muss die Ausführung der Operationen, die durch den Funktionsbaustein definiert sind, nach den folgenden Regeln gesteuert werden:

1) Falls der Wert von EN gleich FALSE (0) ist, wenn der Funktionsbaustein aufgerufen wird, dürfen die Zuweisungen der aktuellen Werte an die Funktionsbausteineingänge **implementierungsabhängig** ausgeführt werden oder auch nicht ausgeführt werden, die Operationen, die durch den Funktionsbausteinrumpf definiert sind, dürfen nicht ausgeführt werden und der Wert von ENO muss durch das SPS-System auf FALSE (0) zurückgesetzt werden.
2) Anderenfalls muss der Wert von ENO vom SPS-System auf TRUE (1) gesetzt werden, die Zuweisungen der aktuellen Werte an die Funktionsbausteineingänge muss ausgeführt werden und die Operationen, die durch den Funktionsbausteinrumpf definiert sind, müssen ausgeführt werden. Diese Operationen können die Zuweisung eines booleschen Werts an ENO einschließen.
3) Falls der Ausgang ENO mit FALSE (0) ausgewertet wird, behalten die Werte der Funktionsbausteinausgänge (VAR_OUTPUT) ihren Zustand vom vorhergehenden Aufruf.

ANMERKUNG Es folgt aus diesen Regeln, dass der Ausgang ENO eines Funktionsbausteins explizit durch die aufrufende Einheit geprüft werden muss, falls es notwendig ist, die möglichen Fehlerbedingungen zu ermitteln.

BEISPIELE Die Bilder unten veranschaulichen den Gebrauch von EN und ENO in Verbindung mit den Standardbausteinen TP, TON, TOF (durch T** dargestellt), die in 2.5.2.3.4 definiert sind, und den Bausteinen CTU und CTD (durch CT* dargestellt), die in 2.5.2.3.3 definiert sind. In Übereinstimmung mit den obigen Regeln darf ein Wert FALSE des Eingangs EN verwendet werden, um die Ausführung des zugehörigen Funktionsbausteins „einzufrieren"; d. h. die Ausgangswerte ändern sich nicht, auch bei der Änderung an einem der anderen Eingangswerte. Wenn der EN Eingangswert von EN TRUE wird, kann die normale Ausführung des Funktionsbausteins fortgesetzt werden. Nach der Ausführung des Funktionsbausteins, bei dem der Eingang EN FALSE ist, ist der Wert des Ausgangs ENO FALSE. Wenn EN TRUE ist, zeigt ein Wert TRUE an ENO eine normale Ausführung des Bausteins, und ein Wert FALSE an ENO kann zur Anzeige einer implementierungsabhängigen Fehlerbedingung verwendet werden.

2.5.2.2 Deklaration

Wie in Bild 10 veranschaulicht, muss ein Funktionsbaustein in Text oder Grafik in derselben Weise deklariert werden, wie dies für Funktionen in 2.5.1.3 definiert ist, mit den Unterschieden, die unten beschrieben und in Tabelle 33 zusammengefasst sind:

1) Die begrenzenden Schlüsselwörter für die Deklaration von Funktionsbausteinen müssen FUNCTION_BLOCK...END_FUNCTION_BLOCK sein.
2) Das Bestimmungszeichen RETAIN, das in 2.4.3 definiert ist, kann für interne und Ausgangsvariablen eines Funktionsbausteins benutzt werden, wie in den Eigenschaften 1, 2 und 3 in Tabelle 33 gezeigt ist.

3) Die Werte von Variablen, die dem Funktionsbaustein mittels eines Konstrukts VAR_EXTERNAL übergeben werden, können innerhalb des Funktionsbausteins verändert werden, wie in Eigenschaft 10 der Tabelle 33 gezeigt ist.

4) Auf die Ausgangswerte einer Instanz eines Funktionsbausteins, dessen Name in den Funktionsbaustein mittels eines Konstrukts VAR_INPUT, VAR_IN_OUT oder VAR_EXTERNAL übergeben wird, kann zugegriffen werden; sie können aber nicht innerhalb des Funktionsbausteins verändert werden; dies ist in den Eigenschaften 5, 6 und 7 der Tabelle 33 gezeigt.

5) Ein Funktionsbaustein, dessen Instanzname in den Funktionsbaustein mittels einer Konstruktion VAR_EXTERNAL übergeben wird, kann innerhalb des Funktionsbausteins aufgerufen werden; dies ist in den Eigenschaften 6 und 7 der Tabelle 33 gezeigt.

6) In Deklarationen in Textform können die Bestimmungszeichen R_EDGE und F_EDGE benutzt werden, um eine Flankenerkennung an booleschen Eingängen anzuzeigen. Dies muss die implizite Deklaration eines Funktionsbausteins vom Typ R_TRIG bzw. F_TRIG bewirken, wie in 2.5.2.3.2 definiert ist, um die geforderte Flankenerkennung auszuführen. Als ein Beispiel dieser Konstruktion siehe die Eigenschaften 8a und 8b der Tabelle 33 und die zugehörige ANMERKUNG.

7) Die Konstruktion, die in den Eigenschaften 9a und 9b von Tabelle 33 veranschaulicht ist, muss in grafischen Deklarationen zur Erkennung von steigenden und fallenden Flanken benutzt werden. Wenn der Zeichensatz, der in 2.1.1 definiert ist, benutzt wird, muss das Zeichen „größer als“ (>) oder „kleiner als“ (<) in der Außenlinie des Funktionsbausteins liegen. Wenn grafische oder semigrafische Darstellungen verwendet werden, muss die Schreibweise der IEC 60617-12 für dynamische Eingänge benutzt werden.

8) Falls die allgemeinen Datentypen, die in Tabelle 11 angegeben sind, in der Deklaration von Eingängen und Ausgängen der Standard-Funktionsbausteine benutzt werden, dann müssen die Regeln für die Auswirkung der aktuellen Typen von Ausgängen solcher Funktionsbaustein-Typen Teil der Funktionsbaustein-Definition sein. In textuellen Aufrufen solcher Funktionsbausteine müssen die Zuweisungen von Ausgängen auf Variablen direkt in der Aufrufanweisung (durch Verwenden des Operators „=>“) gemacht werden.

9) Die Angabe des Stern (Asterisk, Eigenschaft 10 in Tabelle 15) kann in der Deklaration von internen Variablen eines Funktionsbausteins verwendet werden.

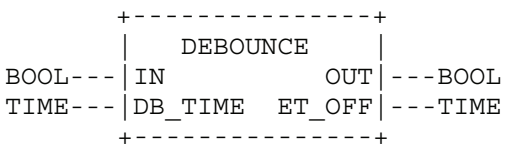
10) Die Eingänge EN und Ausgänge ENO müssen, wie in 2.5.1.2a) beschrieben, deklariert und verwendet werden.

11) Es muss ein **Fehler** sein, wenn kein Wert festgelegt ist für: (i) eine Ein-Aus-Variable einer Funktionsbaustein-Instanz; (ii) eine Funktionsbaustein-Instanz, die als eine Eingangsvariable einer anderen Funktionsbaustein-Instanz verwendet wird.

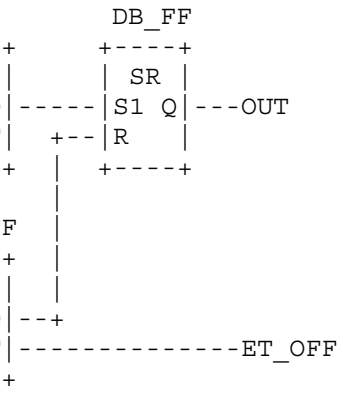
Wie in Bild 12 veranschaulicht, können nur Variablen oder Funktionsbaustein-Instanznamen in einen Funktionsbaustein mittels des Konstrukts VAR_IN_OUT übergeben werden; d. h. Ausgänge von Funktionen oder Funktionsbausteinen können nicht mittels dieser Konstruktion übergeben werden. Dies soll die versehentlichen Änderungen solcher Ausgänge verhindern. Das „Kaskadieren“ von VAR_IN_OUT Konstruktionen, wie in Bild 12c) gezeigt, ist jedoch zugelassen.

```
(* a) Textuelle Deklaration in ST-Sprache (siehe 3.3) *)
FUNCTION_BLOCK DEBOUNCE
(*** External Interface ***)
VAR_INPUT
  IN      : BOOL ;                  (* Voreinstellung = 0 *)
  DB_TIME : TIME := t#10ms ;        (* Voreinstellung = t#10ms *)
END_VAR
VAR_OUTPUT OUT    : BOOL ;          (* Voreinstellung = 0    *)
    ET_OFF : TIME ;                 (* Voreinstellung = t#0s *)
END_VAR
VAR DB_ON   : TON  ;                (* Interne Variablen *)
    DB_OFF  : TON  ;                (* und FB-Instanzen  *)
    DB_FF   : SR   ;
END_VAR
(** Funktionsbaustein-Rumpf **)
DB_ON(IN:=IN, PT:=DB_TIME) ;
DB_OFF(IN:=NOT IN, PT:=DB TIME) ;
DB_FF(S1:=DB_ON.Q, R:=DB_OFF.Q) ;
OUT := DB_FF.Q ;
ET_OFF := DB_OFF.ET ;
END_FUNCTION_BLOCK
```

```
(* b) Grafische Deklaration in FBD-Sprache (siehe 4.3) *)
FUNCTION_BLOCK
(* Außen-Schnittstelle *)
```




```
(* Funktionsbaustein-Rumpf *)
```




```
END_FUNCTION_BLOCK
```

Bild 10 – Beispiele für Deklarationen von Funktionsbausteinen

Tabelle 33 – Funktionsbaustein Deklaration und Verwendung

Nr.	Beschreibung	Beispiel
1a	RETAIN Bestimmungszeichen für interne Variablen	`VAR RETAIN X : REAL;  END_VAR`
1b	NON_RETAIN Bestimmungszeichen für interne Variablen	`VAR NON_RETAIN X : REAL;  END_VAR`
2a	RETAIN Bestimmungszeichen für Ausgangsvariablen	`VAR_OUTPUT RETAIN X : REAL ;END_VAR`
2b	RETAIN Bestimmungszeichen für Eingangsvariablen	`VAR_INPUT RETAIN X : REAL ; END_VAR`
2c	NON_RETAIN Bestimmungszeichen für Ausgangsvariablen	`VAR_OUTPUT NON_RETAIN X : REAL ;END_VAR`
2d	NON_RETAIN Bestimmungszeichen für Eingangsvariablen	`VAR_INPUT NON_RETAIN X : REAL ; END_VAR`
3a	RETAIN Bestimmungszeichen für interne Funktionsbausteine	`VAR RETAIN TMR1: TON ;  END_VAR`
3b	NON_RETAIN Bestimmungszeichen für interne Funktionsbausteine	`VAR NON_RETAIN TMR1: TON ;  END_VAR`
4a	VAR_IN_OUT Deklaration (Textform)	`VAR_IN_OUT A: INT ;  END_VAR`
4b	VAR_IN_OUT Deklaration (grafisch)	Siehe Bild 12
4c	VAR_IN_OUT Deklaration mit Zuweisung an verschiedene Variablen (grafisch)	Siehe Bild 12d)
5a	Funktionsbaustein-Instanzname als Eingang (Textform)	`VAR_INPUT I_TMR: TON ;  END_VAR` `EXPIRED := I_TMR.Q; (* Anmerkung 1 *)`
5b	Funktionsbaustein-Instanzname als Eingang (grafisch)	Siehe Bild 11a)

6a	Funktionsbaustein-Instanzname als VAR_IN_OUT (Textform)	`VAR_IN_OUT IO_TMR: TOF ;  END_VAR`
		`IO_TMR(IN:=A_VAR, PT:=T#10S);`
		`EXPIRED := IO_TMR.Q; (* Anmerkung 1 *)`

6b	Funktionsbaustein-Instanzname als Eingang/Ausgang (grafisch)	Siehe Bild 11b)

7a	Funktionsbaustein-Instanzname als externe Variable (Textform)	`VAR_EXTERNAL EX_TMR : TOF ; END_VAR`
		`EX_TMR(IN:=A_VAR,  PT:=T#10S);`
		`EXPIRED := EX_TMR.Q; (* Anmerkung 1*)`

7b	Funktionsbaustein-Instanzname als externe Variable (grafisch)	Siehe Bild 11c

<table>
<tr><th>Nr.</th><th>Beschreibung</th><th>Beispiel</th></tr>
<tr><td>8a

8b</td><td>Textuelle Deklaration von:
Eingängen mit steigender Flanke

Eingängen mit fallender Flanke</td><td><pre>
FUNCTION_BLOCK AND_EDGE (* Anmerkung 2 *)
VAR_INPUT X : BOOL R_EDGE;
           Y : BOOL F_EDGE;
END_VAR
VAR_OUTPUT Z : BOOL ; END_VAR
Z := X AND Y ; (* ST-Sprache *)
END_FUNCTION_BLOCK (* - siehe 3.3 *)
</pre></td></tr>
<tr><td>9a
9b</td><td>Grafische Deklaration von:
Eingängen mit steigender Flanke
Eingängen mit fallender Flanke</td><td><pre>
       FUNCTION_BLOCK
       +------------+ (* Anmerkung 2 *)
       | AND_EDGE |
BOOL--->X   Z|---BOOL
       | |
BOOL---<Y |
       | |
       +-------------+

       +---+   (* FB-Rumpf   *)
   X---| & |---Z (* FBS Sprache *)
   Y---| | (* - siehe 4.3 *)
       +---+
   END_FUNCTION_BLOCK
</pre></td></tr>
<tr><td>10a</td><td colspan="2">VAR_EXTERNAL Deklarationen innerhalb von Funktionsbausteintyp-Deklarationen</td></tr>
<tr><td>10b</td><td colspan="2">VAR_EXTERNAL CONSTANT Deklarationen innerhalb von Funktionsbausteintyp-Deklarationen</td></tr>
<tr><td>11</td><td colspan="2">VAR_TEMP Deklarationen (siehe 2.4.3) innerhalb von Funktionsbausteintyp-Deklarationen</td></tr>
<tr><td colspan="3">ANMERKUNG 1 Es wird in diesen Beispielen angenommen, dass die Variablen EXPIRED und A_VAR mit dem Typ BOOL deklariert wurden.

ANMERKUNG 2 Die Deklaration des Funktionsbausteins AND_EDGE in den obigen Beispielen entspricht der Folgenden:

<pre>
FUNCTION_BLOCK AND_EDGE
    VAR_INPUT X : BOOL; Y : BOOL; END_VAR
    VAR X_TRIG: R_TRIG; Y_TRIG: F_TRIG; END_VAR
    X_TRIG(CLK := X) ;
    Y_TRIG(CLK := Y) ;
    Z := X_RIG.Q AND Y_TRIG.Q;
END_FUNCTION_BLOCK
</pre>
Siehe 2.5.2.3.2 für die Definition der Funktionsbausteine zur Flankenerkennung R_TRIG und F_TRIG.</td></tr>
</table>

```
FUNCTION_BLOCK

        +--------------+     (* Außen-Schnittstelle *)
        |   INSIDE_A   |
  TON---|I_TMR  EXPIRED|---BOOL
        +--------------+
                         (* Funktionsbaustein Rumpf *)

             +------+
             | MOVE |
  I_TMR.Q--- |      |---EXPIRED
             +------+
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK

         +--------------+      (* Außen-Schnittstelle *)
         |  EXAMPLE_A   |
  BOOL---|GO        DONE|---BOOL
         +--------------+

              E_TMR            (* Funktionsbaustein Rumpf*)

             +-----+                    I_BLK

             | TON |              +--------------+
       GO---|IN  Q|              |   INSIDE_A   |

   t#100ms---|PT ET|     E_TMR---|I_TMR  EXPIRED|---DONE

             +-----+              +--------------+
END_FUNCTION_BLOCK
```

ANMERKUNG `I_TMR` ist in diesem Bild nicht grafisch dargestellt, da dies einen Aufruf von `I_TMR` innerhalb von `INSIDE_A` bedeuten würde, der durch die Regeln 4) und 5) von 2.5.2.2 verboten ist. Siehe auch die Eigenschaft 5a) von Tabelle 33.

Bild 11 a) – Grafischer Gebrauch eines Funktionsbaustein-Namens als eine Eingangsvariable (Tabelle 33, Eigenschaft 5b)

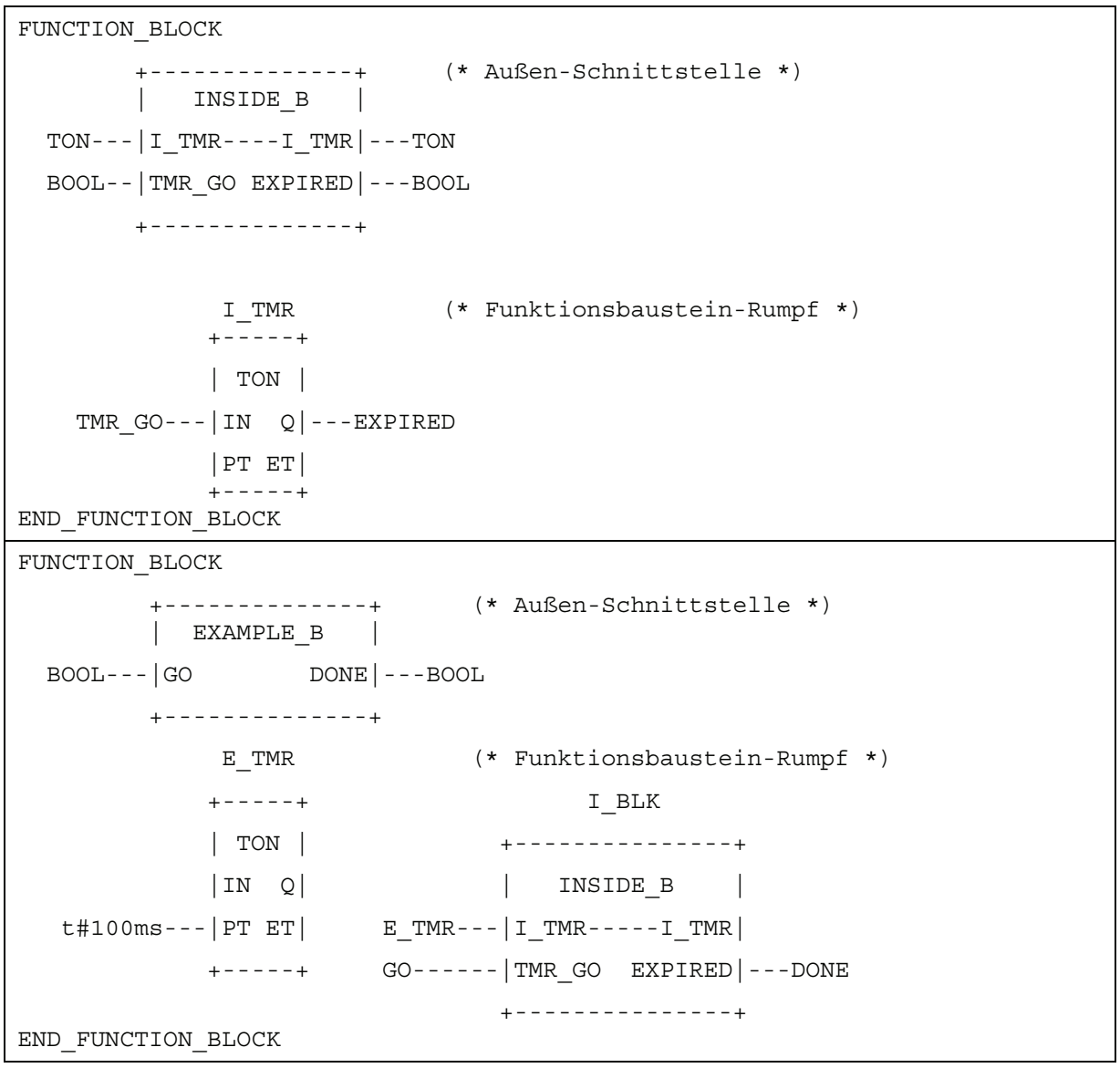


Bild 11 b) – Grafischer Gebrauch eines Funktionsbaustein-Namens als eine Eingangs-/Ausgangsvariable (Tabelle 33, Eigenschaft 6b)

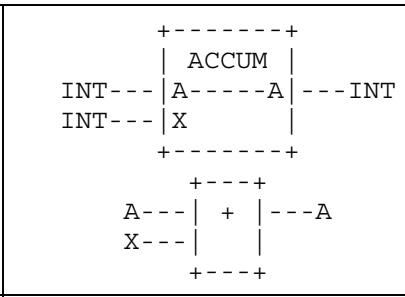
```
FUNCTION_BLOCK
      +--------------+         (* Außen-Schnittstelle   *)
      |   INSIDE_C   |
BOOL--|TMR_GO EXPIRED|---BOOL
      +--------------+
VAR_EXTERNAL X_TMR: TON; END_VAR

              X_TMR            (* Funktionsbaustein-Rumpf *)
             +-----+
             | TON |
    TMR_GO---|IN  Q|---EXPIRED
             |PT ET|
             +-----+
END_FUNCTION_BLOCK
```

```
PROGRAM
         +--------------+      (* Außen-Schnittstelle *)
         |  EXAMPLE_C   |
  BOOL---|GO        DONE|---BOOL
         +--------------+
  VAR_GLOBAL X_TMR: TON; END_VAR

               I_BLK           (* Programm-Rumpf *)
         +---------------+
         |   INSIDE_C    |
 GO------|TMR_GO  EXPIRED|---DONE
         +---------------+
END_PROGRAM
```

ANMERKUNG Die Deklaration von `PROGRAM` ist in 2.5.3 definiert.

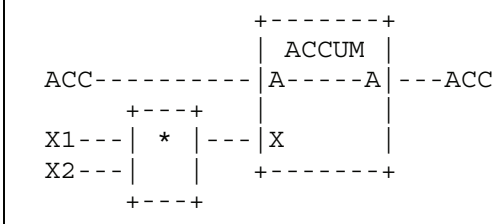
Bild 11 c) – Grafischer Gebrauch eines Funktionsbaustein-Namens als eine externe Variable (Tabelle 33, Eigenschaft 7b)

12a)




```
FUNCTION_BLOCK ACCUM
   VAR_IN_OUT A : INT ; END_VAR
   VAR_INPUT  X : INT ; END_VAR
   A := A + X;
END_FUNCTION_BLOCK
```

12b)



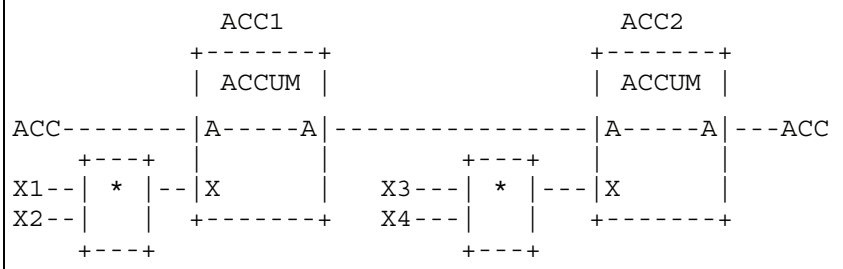
Bei einer Deklaration wie

```
VAR
       ACC : INT ;
       X1  : INT ;
       X2  : INT ;
END_VAR
```

ist die Wirkung:

```
ACC := ACC+X1*X2 ;
```

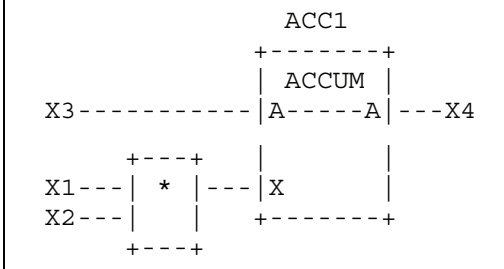
12c)



Bei Deklarationen wie in **12b)** für `ACC`, `X1`, `X2`, `X3` und `X4` ist die Wirkung der Ausführung

```
ACC :=
ACC+X1*X2+X3*X4 ;
```

12d)



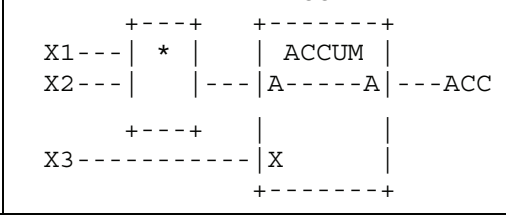
Bei einer Deklaration wie

```
VAR
    X1 : INT ;
    X2 : INT ;
....X3 : INT ;
 ...X4 : INT ;
END_VAR
```

ist die Wirkung:

```
X3 := X3+X1*X2 ;
X4 := X3 ;
```

12e)



UNZULÄSSIGER GEBRAUCH!!!

Die Verbindung zur Ein-Aus-Variablen `A` ist keine Variable oder kein Funktionsbaustein-Name (siehe vorausgehender Text).

Bild 12 – Beispiele für den Gebrauch von Eingangs/Ausgangsvariablen in Funktionsbausteinen
a) Deklarationen in grafischer und Textform
b), c), d) Zulässiger Gebrauch; e) Unzulässiger Gebrauch

2.5.2.3 Standard-Funktionsbausteine

Die Definitionen von Funktionsbausteinen, die für alle SPS-Programmiersprachen gemeinsam sind, werden in diesem Abschnitt angegeben.

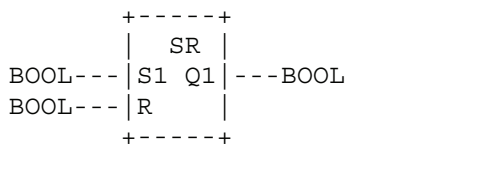
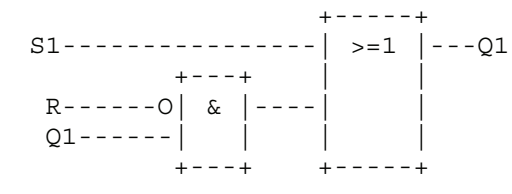
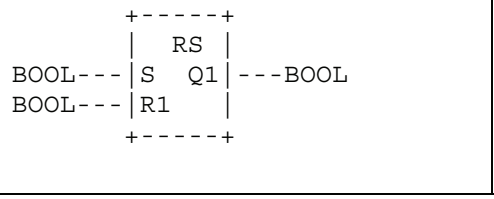
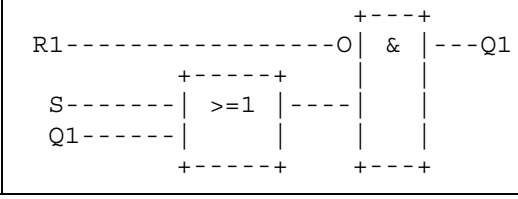
Überall, wo in diesem Abschnitt grafische Deklarationen gezeigt werden, können auch gleichartige Deklarationen, wie in 2.5.2.2 festgelegt, in Textform geschrieben werden; dies ist als Beispiel in Tabelle 35 gezeigt.

Standardfunktionsbausteine dürfen *überladen* sein und können *erweiterbare* Ein- und Ausgänge haben. Die Definition solcher Typen von Funktionsbausteinen muss Begrenzungen hinsichtlich Anzahl und Datentypen solcher Ein- und Ausgänge beschreiben. Die Verwendung dieser Möglichkeiten bei Nicht-Standard-Funktionsbausteinen überschreitet die Gültigkeit dieses Standards.

2.5.2.3.1 Bistabile Elemente

Die Darstellung und die *Funktionsbaustein-Rümpfe* für die standardmäßigen bistabilen Elemente sind in Tabelle 34 gezeigt. Die Schreibweise für diese Elemente ist so übereinstimmend wie möglich mit den Symbolen 12-09-01 und 12-09-02 der IEC 60617-12 gewählt worden.

Tabelle 34 – Bistabile Funktionsbausteine [a]

Nr.	Grafische Form	Funktionsbaustein-Rumpf
1	**Bistabiler Funktionsbaustein (vorrangig Setzen)**	
		
2	**Bistabiler Funktionsbaustein (vorrangig Rücksetzen)**	
		
ANMERKUNG Der Rumpf des Funktionsbausteins ist in FBS Sprache spezifiziert, die in 4.3 definiert ist.		
[a] Der Anfangszustand der Ausgangsvariablen `Q1` muss der normale voreingestellte Wert Null für boolesche Variablen sein.		

2.5.2.3.2 Flankenerkennung

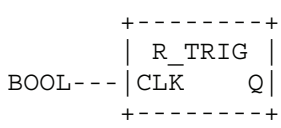
Die grafische Darstellung von Standard-Funktionsbausteinen zur Erkennung von steigender und fallender Flanke muss, wie in Tabelle 35 gezeigt, erfolgen. Das Verhalten dieser Bausteine muss nach den Definitionen, die in dieser Tabelle angegeben sind, geschehen. Dieses Verhalten entspricht den folgenden Regeln:

1) Der Ausgang `Q` eines Funktionsbausteins `R_TRIG` muss von einer Ausführung des Funktionsbausteins bis zur nächsten auf dem booleschen Wert `BOOL#1` bleiben; dabei folgt er dem Übergang des Eingangs `CLK` von `0` nach `1`, und er muss bei der nächsten Ausführung nach `0` zurückkehren.

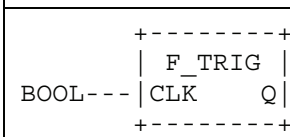
2) Der Ausgang `Q` eines Funktionsbausteins `F_TRIG` muss von einer Ausführung des Funktionsbausteins bis zur nächsten auf dem booleschen Wert `BOOL#1` bleiben; dabei folgt er dem Übergang des Eingangs `CLK` von `1` nach `0`, und er muss bei der nächsten Ausführung nach `0` zurückkehren.

Tabelle 35 – Standard-Funktionsbausteine Flankenerkennung

Nr.	Grafische Form	Definition (ST Sprache – siehe 3.3)
1	**Erkennung der steigenden Flanke**	

		`FUNCTION_BLOCK R_TRIG`
		`VAR_INPUT CLK: BOOL; END_VAR`
		`VAR_OUTPUT Q: BOOL; END_VAR`
		`VAR M: BOOL; END_VAR`
		`Q := CLK AND NOT M;`
		`M := CLK;`
		`END_FUNCTION_BLOCK`

2	**Erkennung der fallenden Flanke**	

		`FUNCTION_BLOCK F_TRIG`
		`VAR_INPUT CLK: BOOL; END_VAR`
		`VAR_OUTPUT Q : BOOL; END_VAR`
		`VAR M: BOOL; END_VAR`
		`Q := NOT CLK AND NOT M;`
		`M := NOT CLK;`
		`END_FUNCTION_BLOCK`

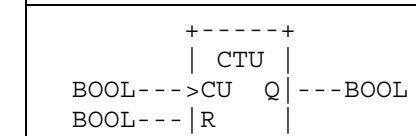
ANMERKUNG Wenn der Eingang `CLK` einer Instanz von Typ `R_TRIG` mit einem Wert `BOOL#1` verbunden ist, wird sein Ausgang `Q` nach seiner ersten Ausführung, die auf einen „Kaltstart“, wie er in 2.4.2 beschrieben ist, folgt, auf `BOOL#1` stehen bleiben. Der Ausgang `Q` wird alle folgenden Ausführungen auf `BOOL#0` stehen bleiben. Das gleiche gilt für eine Instanz `F_TRIG` , deren Eingang `CLK` unverbunden ist oder mit den Wert `FALSE` verbunden ist.		

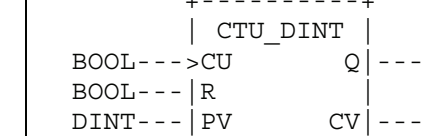
2.5.2.3.3 Zähler

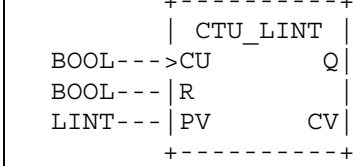
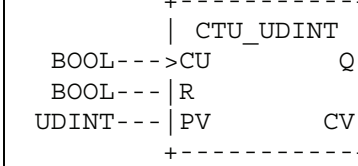
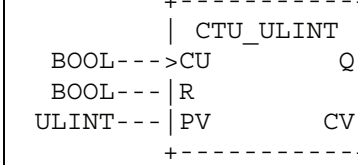
Die grafischen Darstellungen von Standard-Funktionsbausteinen für Zähler, mit den Typen der zugehörigen Eingänge und Ausgänge, sind in Tabelle 36 gezeigt. Die Arbeitsweise dieser Funktionsbausteine muss, wie in den entsprechenden Rümpfen der Funktionsbausteine festgelegt ist, erfolgen.

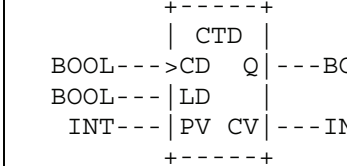
Tabelle 36 – Standard-Funktionsbausteine Zähler

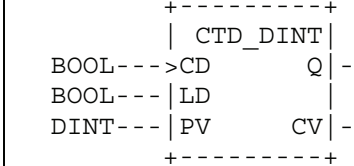
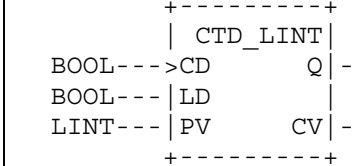
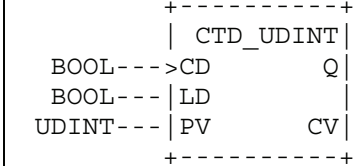
Nr.	Grafische Form	Funktionsbaustein-Rumpf (ST Sprache – siehe 3.3)
1a	**Aufwärts-Zähler**	

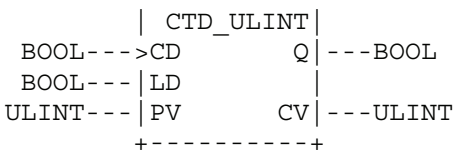
		`IF R THEN CV := 0 ;`
		`ELSIF CU AND (CV < PVmax)`
		`THEN CV := CV+1;`
		`END_IF ;`
		`Q := (CV >= PV) ;`

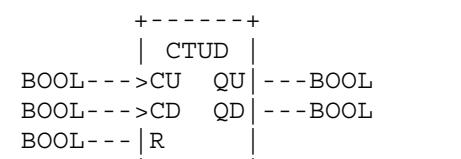
1b		Wie bei 1a

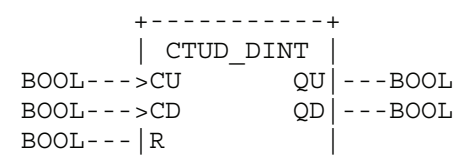
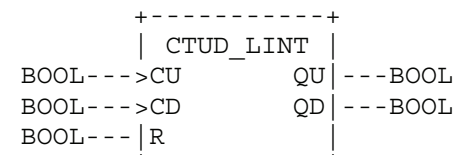
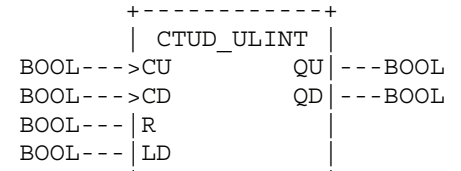
Nr.	Grafische Form	Funktionsbaustein-Rumpf (ST Sprache – siehe 3.3)
1c		Wie bei 1a
1d		Wie bei 1a
1e		Wie bei 1a
2a	**Abwärts-Zähler**	

		`IF LD THEN CV := PV ;`
		`ELSIF CD AND (CV > PVmin)`
		`  THEN CV := CV-1;`
		`END_IF ;`
		`Q := (CV <= 0) ;`

2b		Wie bei 2a
2c		Wie bei 2a
2d		Wie bei 2a

Nr.	Grafische Form	Funktionsbaustein-Rumpf (ST Sprache – siehe 3.3)
2e		Wie bei 2a
	Aufwärts-Abwärts-Zähler	

3a		`IF R THEN CV := 0 ;`
		`ELSIF LD THEN CV := PV ;`
		`ELSE`
		`  IF NOT (CU AND CD) THEN`
		`  IF CU AND (CV < PVmax)`
		`  THEN CV := CV+1;`
		`  ELSIF CD AND (CV > PVmin)`
		`  THEN CV := CV-1;`
		`  END_IF ;`
		`  END_IF;`
		`END_IF;`
		`QU := (CV >= PV) ;`
		`QD := (CV <= 0) ;`

3b		Wie bei 3a
3c		Wie bei 3a
3d		Wie bei 3a
ANMERKUNG Die numerischen Grenzwerte `PVmin` und `PVmax` sind **implementierungsabhängig**.		

2.5.2.3.4 Zeitgeber

Die Standard-Funktionsbausteine für Zeitgeber müssen die Form haben, die in Tabelle 37 gezeigt ist. Die Arbeitsweise dieser Funktionsbausteine muss erfolgen, wie in den Zeitdiagrammen der Tabelle 38 definiert ist.

Tabelle 37 – Standard-Funktionsbaustein Zeitgeber

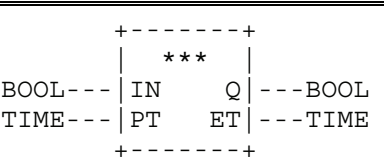
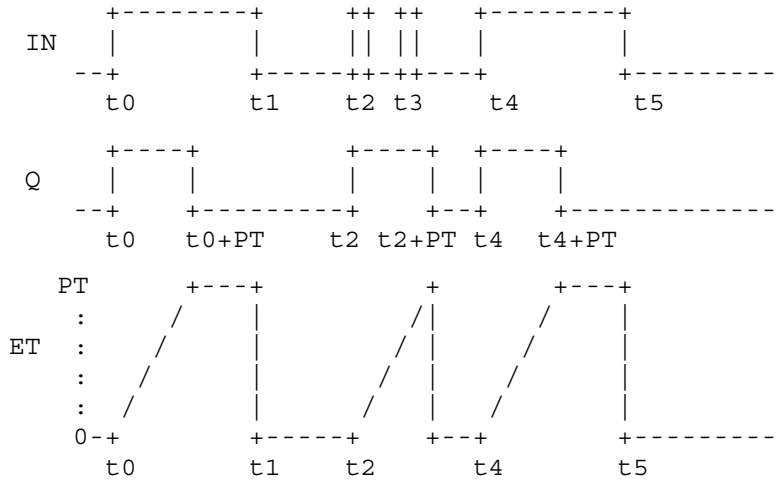
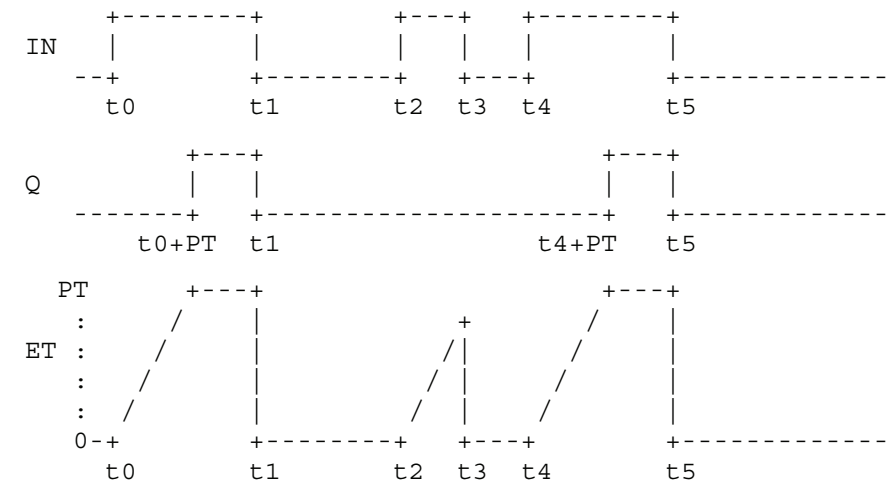
Nr.	Beschreibung	Grafische Form
1	*** bedeutet: `TP` (Puls)	
2a	`TON` (Einschalt-Verzögerung)	
2b [a]	`T---0` (Einschalt-Verzögerung)	
3a	`TOF` (Ausschalt-Verzögerung)	
3b [a]	`0---T` (Ausschalt-Verzögerung)	
ANMERKUNG Die Wirkung bei einem Wechsel des Wertes des Eingangs `PT` während der Zeitbearbeitung, z. B. das Setzen von `PT` auf `t#0s`, um die Bearbeitung einer Instanz von `TP` zurückzusetzen, muss ein **implementierungsabhängiger Parameter** sein.		
[a] In Text-Sprachen dürfen die Eigenschaften 2b und 3b nicht benutzt werden.		

Tabelle 38 – Standard-Funktionsbausteine Zeitgeber – Zeitdiagramme

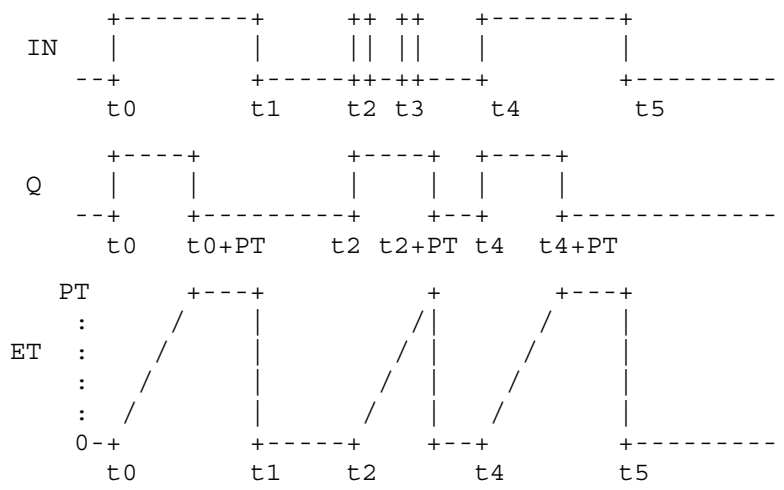
Puls (TP) Zeitverhalten



Einschalt-Verzögerung (TON)

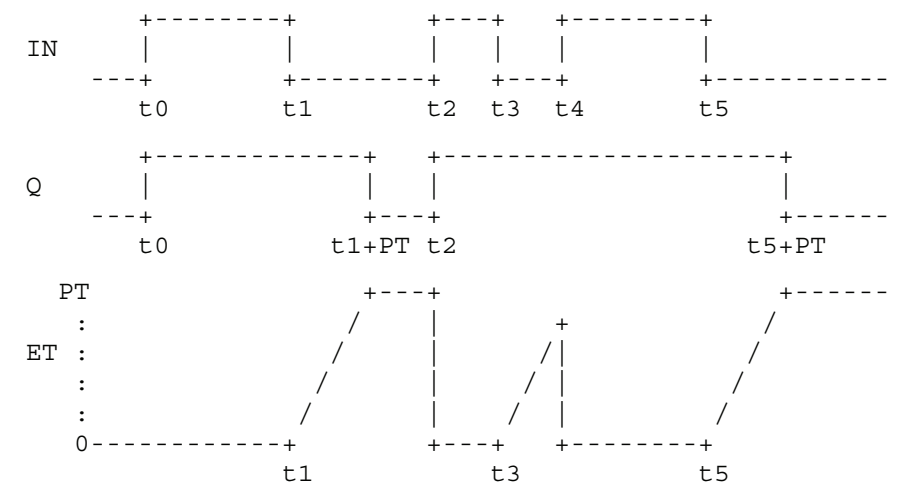


Puls (TP) Zeitverhalten



Einschalt-Verzögerung (TON)

Ausschalt-Verzögerung (TOF)



2.5.2.3.5 Funktionsbausteine für Kommunikation

Die Standard-Funktionsbausteine für SPS sind in IEC 61131-5 definiert. Diese Funktionsbausteine liefern programmierbare Kommunikationsfunktionalität wie Geräte-Verifikation, zyklische Datenerfassung, programmierte Datenerfassung, parametrierte Steuerung, verzahnte Steuerung, programmierte Alarmmeldung sowie Verbindungsmanagement und Schutz.

2.5.3 Programme

Ein *Programm* ist in IEC 61131-1 definiert als „eine logische Sammlung von allen Programmiersprachen-Elementen und -Konstrukten, die notwendig für eine gewünschte Signalverarbeitung sind, die zur Steuerung einer Maschine oder eines Prozesses durch ein SPS-System erforderlich ist.“

Der Abschnitt 1.4.1 dieses Teils beschreibt die Stellung von Programmen in dem übergreifenden Software-Modell einer SPS, der Abschnitt 1.4.2 beschreibt die verfügbaren Mittel für die Kommunikation zwischen Programmen und innerhalb eines Programms und der Abschnitt 1.4.3 beschreibt den übergreifenden Prozess der Programmentwicklung.

Die Deklaration und der Gebrauch von *Programmen* ist identisch mit denen der *Funktionsbausteine*, die in 2.5.2.1 und 2.5.2.2 definiert sind, mit den zusätzlichen Eigenschaften, die in Tabelle 39 gezeigt sind und den folgenden Unterschieden:

1) Die begrenzenden Schlüsselwörter für die Programmdeklarationen müssen PROGRAM...END_PROGRAM sein.

2) Ein Programm kann eine Konstruktion VAR_ACCESS...END_VAR enthalten, die die Möglichkeit zur Festlegung von Variablen mit Namen bietet, auf die durch einige der Kommunikationsdienste zugegriffen werden kann, die in IEC 61131-5 festgelegt sind. Ein Zugriffspfad verknüpft jede derartige Variable mit einem Eingang, Ausgang oder einer internen Variablen des *Programms*. Das Format und der Gebrauch dieser Deklaration muss so sein, wie es in 2.7.1 und IEC 61131-5 beschrieben ist.

3) Programme können nur innerhalb von *Ressourcen* instanziiert werden, wie in 2.7.1 definiert ist, während *Funktionsbausteine* nur innerhalb von *Programmen* oder anderen *Funktionsbausteinen* instanziiert werden können.

4) Ein Programm kann Zuweisungen des Speicherorts, wie in 2.4.3.1 und 2.4.3.2. beschrieben, in den Deklarationen ihrer globalen und internen Variablen enthalten. Zuweisungen des Speicherorts mit nicht vollständig spezifizierter direkter Darstellung, wie in 2.4.1.1 und 2.4.3.1 beschrieben, können nur in Deklarationen von internen Variablen eines Programms verwendet werden.

Die Deklaration und der Gebrauch von Programmen sind in Bild 19 sowie in den Beispielen F.7 und F.8 veranschaulicht.

Begrenzungen der Größe von Programmen in einer bestimmtem Ressource sind **implementierungsabhängige Parameter**.

Tabelle 39 – Eigenschaften der Programm-Deklaration

Nr.	Beschreibung
1 bis **9b**	Dieselben Eigenschaften 1 bis 9b der Tabelle 33
10	Formale Eingangs- und Ausgangsvariablen
11 bis **14**	Dieselben Eigenschaften 1 bis 4 der Tabelle 17
15 bis **18**	Dieselben Eigenschaften 1 bis 4 der Tabelle 18
19	Gebrauch von direkt dargestellten Variablen (Abschnitt 2.4.1.1)
20	VAR_GLOBAL...END_VAR Deklaration innerhalb eines PROGRAM (siehe 2.4.3 und 2.7.1)
21	VAR_ACCESS...END_VAR Deklaration innerhalb eines PROGRAM
22a	VAR_EXTERNAL Deklarationen innerhalb von PROGRAM Typdeklarationen
22b	VAR_EXTERNAL CONSTANT Deklarationen innerhalb von PROGRAM Typdeklarationen
23	VAR_GLOBAL CONSTANT Deklarationen innerhalb von PROGRAM Typdeklarationen
24	VAR_TEMP Deklarationen (siehe 2.4.3) innerhalb von PROGRAM Typdeklarationen

2.6 Elemente der Ablaufsprache (AS)

2.6.1 Allgemeines

Dieser Abschnitt definiert die Elemente der *Ablaufsprache* (AS) als Mittel zur Gliederung der internen Organisation einer SPS-Programm-Organisationseinheit, die in einer der in dieser Norm definierten Sprachen geschrieben ist; der Zweck ist dabei die Ausführung von Ablaufsteuerungsfunktionen. Die Definitionen in diesem Abschnitt sind aus IEC 60848 abgeleitet. Dabei waren Änderungen bei den Darstellungen notwendig, um eine *Dokumentationsnorm* in eine Menge von *Elementen* zur *Ausführungssteuerung* für eine SPS-Programm-Organisationseinheit umzuwandeln.

Die Elemente der Ablaufsprache bieten Hilfsmittel zur Aufteilung einer SPS-Programm-Organisationseinheit in eine Menge von *Schritten* und Transitionen, die durch *gerichtete Verbindungen* miteinander verbunden sind. Zugeordnet zu jedem Schritt ist eine Menge von *Aktionen* und jeder Transition ist eine *Transitionsbedingung* zugehörig.

Da die AS-Elemente Speicher für Zustandsinformationen benötigen, kommen als einzige Programm-Organisationseinheiten, die unter Verwendung dieser Elemente organisiert werden können, *Funktionsbausteine* und *Programme* in Betracht.

Wenn ein Teil einer Programm-Organisationseinheit in AS-Elemente gegliedert ist, muss die gesamte Programm-Organisationseinheit so gegliedert sein. Wenn keine AS-Gliederung für eine Programm-Organisationseinheit vorgegeben ist, dann muss die gesamte Programm-Organisationseinheit als eine einzelne *Aktion* betrachtet werden, die unter Steuerung der aufrufenden Einheit abläuft.

2.6.2 Schritte

Ein *Schritt* stellt eine Situation dar, in der das Verhalten einer Programm-Organisationseinheit in Bezug auf ihre Eingänge und Ausgänge einer Menge von Regeln folgt, die durch die dem Schritt zugehörigen *Aktionen* definiert sind. Ein Schritt ist entweder *aktiv* oder *inaktiv*. Zu jedem gegebenen Zeitpunkt ist der Zustand der Programm-Organisationseinheit durch die Menge der aktiven Schritte und die Werte ihrer internen und Ausgangsvariablen definiert.

Wie in Tabelle 40 gezeigt, muss ein Schritt grafisch durch einen Block dargestellt werden, der einen *Schrittnamen* in Form eines Bezeichners enthält, wie dies in 2.1.2 definiert ist oder in Textform durch eine Konstruktion `STEP...END_STEP`. Die auf den Schritt gerichtete(n) Verbindung(en) kann/können grafisch durch eine vertikale Linie dargestellt werden, die oben am Schritt angeknüpft ist. Die vom Schritt weg gerichtete(n) Verbindung(en) kann/können durch eine vertikale Linie dargestellt werden, die unten am Schritt angeknüpft ist. Alternativ dazu können die gerichteten Verbindungen durch die Textkonstruktion `TRANSITION...END_TRANSITION` dargestellt werden, die in 2.6.3 definiert ist.

Der *Schrittmerker* (aktiver oder inaktiver Zustand eines Schritts) kann durch den logischen Wert eines booleschen Strukturelements `***.X` dargestellt werden; wobei `***` der Schrittname ist, wie es in Tabelle 40 gezeigt ist. Diese boolesche Variable hat den Wert `1`, wenn der entsprechende Schritt aktiv ist und `0`, wenn er inaktiv ist. Der Zustand dieser Variablen steht zur grafischen Verbindung auf der rechten Seite des Schrittes zur Verfügung, wie dies in Tabelle 40 gezeigt ist.

Wie in Tabelle 40 gezeigt, kann die Zeit `***.T`, die nach Initiierung eines Schritts verstrichen ist, in ähnlicher Weise durch ein Strukturelement vom Typ `TIME` repräsentiert werden. Wenn ein Schritt deaktiviert wird, bleibt der Wert der verstrichenen Zeit des Schritts auf dem Wert, der vorlag, als der Schritt rückgesetzt wurde. Wenn ein Schritt aktiviert ist, muss der Wert der verstrichenen Zeit auf `t#0s` zurückgesetzt werden.

Der *Geltungsbereich* von Schrittnamen, Schrittmerkern und Schrittzeiten muss *lokal* zu der Programm-Organisationseinheit sein, in der die Schritte auftreten.

Der Anfangszustand der Programm-Organisationseinheit wird durch die Anfangswerte ihrer internen und Ausgangsvariablen und durch ihre Menge von *Anfangsschritten* dargestellt, d. h. durch die Schritte, die zu Beginn aktiv sind. Jedes AS-*Netzwerk* oder seine äquivalente Textform muss genau einen Anfangsschritt haben.

Ein Anfangsschritt kann grafisch mit Doppellinien an den Rändern gezeichnet werden. Wenn der Zeichensatz für die Zeichnung verwendet wird, der in 2.1.1 definiert ist, muss der Anfangsschritt gezeichnet werden, wie es in Tabelle 40 gezeigt ist.

Zur Systeminitialisierung, wie in 2.4.2 definiert, ist der voreingestellte Anfangswert der verstrichenen Zeit für Schritte `t#0s` und der voreingestellte Anfangszustand für gewöhnliche Schritte ist `BOOL#0` bzw. `BOOL#1` für Anfangsschritte. Wenn aber eine Instanz eines Funktionsbausteins oder eines Programms als gepuffert deklariert ist (wie z. B. in Eigenschaft 3 der Tabelle 33), müssen die Zustände (und falls unterstützt) die verstrichenen Zeiten aller Schritte, die im Programm oder Funktionsbaustein enthalten sind, als gepuffert für die Systeminitialisierung behandelt werden, wie in 2.4.2 definiert ist.

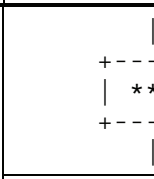
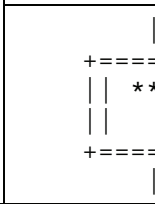
Die maximale Anzahl von Schritten je Ablaufsprache-Diagramm und die Genauigkeit der verstrichenen Zeiten der Schritte sind **implementierungsabhängige Parameter**.

Es muss ein Fehler sein, wenn

1) ein AS-Netzwerk nicht genau einen Anfangsschritt besitzt;

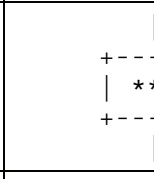
2) ein Anwenderprogramm versucht, einem Schrittmerker oder der verstrichenen Zeit einen Wert direkt zuzuweisen.

Tabelle 40 – Eigenschaften des Schritts

Nr.	Darstellung	Beschreibung
1		Schritt – grafische Form mit gerichteten Verbindungen „***“ = Schritt-Name
		Anfangsschritt – grafische Form mit gerichteten Verbindungen „***“ = Anfangsschritt-Name

2	`STEP *** :`
	`  (* Schritt-Rumpf *)`
	`END_STEP`
		Schritt – Textform
		ohne gerichtete Verbindung (siehe 2.6.3)
		„***“ = Schritt-Name

	`INITIAL_STEP *** :`
	`  (* Schritt-Rumpf *)`
	`END_STEP`
		Anfangsschritt – Textform
		ohne gerichtete Verbindungen (siehe 2.6.3)
		„***“ = Anfangsschritt-Name

3a [a]	`***.X`	Schritt-Merker – allgemeine Form „***“ = Schritt-Name `***.X = BOOL#1` wenn `***` aktiv ist, sonst `BOOL#0`
3b		Schritt-Merker – direkte Verbindung der booleschen Variablen `***.X` mit der rechten Seite des Schritts „***“
4 [a]	`***.T`	Verstrichene Schritt-Zeit – allgemeine Form „***“ = Schritt-Name `***.T` = eine Variable von Typ `TIME` (siehe 2.6.2)
ANMERKUNG Es gibt keine obere gerichtete Verbindung zu einem Anfangsschritt, wenn er keine Vorgänger hat.		

[a] Wenn Eigenschaft 3a, 3b oder 4 unterstützt wird, muss es ein **Fehler** sein, wenn das Anwenderprogramm versucht, die zugehörige Variable zu verändern. Wenn z. B. `S4` ein Schritt-Name ist, dann wären die folgenden Anweisungen Fehler in der ST-Sprache, die in 3.3 definiert ist:
`S4.X := 1 ;  (* FEHLER *)`
`S4.T := t#100ms ;  (* FEHLER *)`		

2.6.3 Transitionen (Übergänge)

Eine *Transition* stellt die Bedingung dar, durch die die Steuerung von einem oder mehreren Schritten, die der Transition vorausgehen, entlang der entsprechenden gerichteten Verbindung auf einen oder mehrere Nachfolgerschritte weitergegeben wird. Die Transition muss durch eine horizontale Linie dargestellt werden, die die vertikale gerichtete Verbindung kreuzt.

Die Richtung des Ablaufs, die den gerichteten Verbindungen folgt, muss von der Unterseite des Vorgängerschritts (oder mehrerer) zur Oberseite des Nachfolgerschritts (oder mehrerer) verlaufen.

Jede Transition muss eine zugehörige *Transitionsbedingung* haben, die das Ergebnis der Auswertung eines einzelnen booleschen Ausdrucks ist. Eine Transitionsbedingung, die immer wahr ist, muss durch das Symbol `1` oder das Schlüsselwort `TRUE` dargestellt werden.

Eine Transitionsbedingung kann durch eines der folgenden Mittel, die in Tabelle 41 gezeigt sind, einer Transition zugeordnet sein:

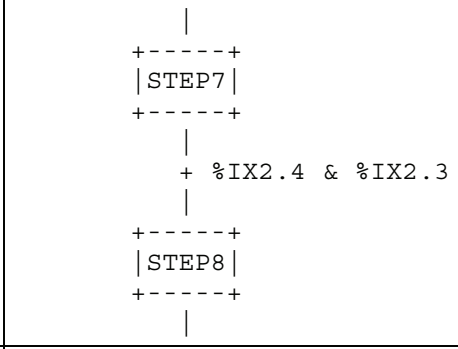
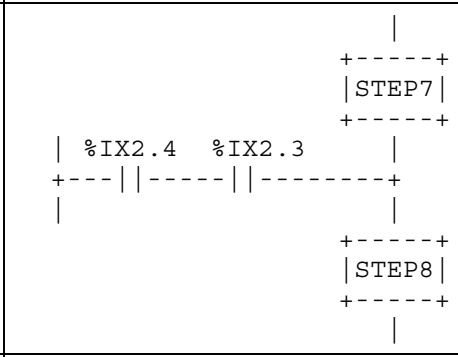
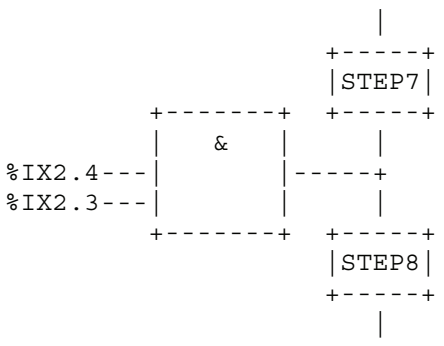
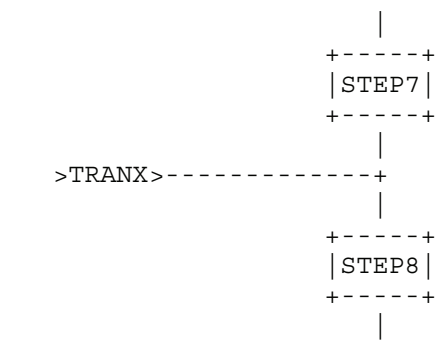
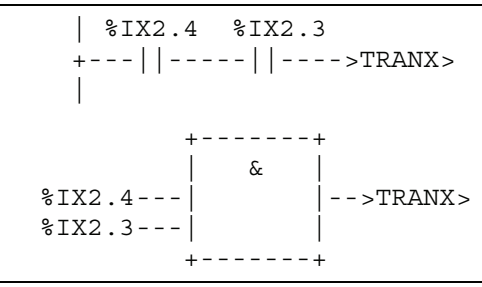
1) Durch die Angabe des geeigneten booleschen Ausdrucks in der ST-Sprache, die in 3.3 definiert ist, der physikalisch oder logisch der vertikalen gerichteten Verbindung zugeordnet ist.
2) Durch ein Kontaktplan-Netzwerk in der KOP-Sprache, die in 4.2 definiert ist, das physikalisch oder logisch der vertikal gerichteten Verbindung zugeordnet ist.
3) Durch ein Netzwerk in der Sprache FBS, die in 4.3 definiert ist, das physikalisch oder logisch der vertikalen gerichtete Verbindung zugeordnet ist.
4) Durch ein KOP- oder FBS-Netzwerk, dessen Ausgang mit der vertikalen gerichteten Verbindung mittels eines Konnektors, der in 4.1.1 definiert ist, verbunden ist.
5) Durch ein Konstrukt `TRANSITION...END_TRANSITION` in der ST-Sprache. Diese muss bestehen aus:
   – den Schlüsselwörtern `TRANSITION FROM`, gefolgt vom Schrittnamen des Vorgänger-Schritts (oder falls es mehr als einen Vorgänger gibt, durch eine eingeklammmerte Liste der Vorgänger-Schritte);
   – dem Schlüsselwort `TO`, gefolgt vom Schrittnamen des Nachfolger-Schritts (oder falls es mehr als einen Nachfolger gibt, durch eine geklammerte Liste der Nachfolger-Schritte);
   – dem Zuweisungsoperator (`:=`), gefolgt von einem booleschen Ausdruck in ST-Sprache, der die Transitionsbedingung festlegt;
   – dem abschließenden Schlüsselwort `END_TRANSITION`.
6) Durch ein Konstrukt `TRANSITION...END_TRANSITION` in der Sprache AWL, die in 3.2 definiert ist. Diese muss bestehen aus:
   – den Schlüsselwörtern `TRANSITION FROM`, gefolgt vom Schrittnamen des Vorgänger-Schritts (oder falls es mehr als einen Vorgänger gibt, durch eine eingeklammmerte Liste der Vorgänger-Schritte), gefolgt durch einen Doppelpunkt (`:`);
   – dem Schlüsselwort `TO`, gefolgt vom Schrittnamen des Nachfolger-Schritts (oder falls es mehr als einen Nachfolger gibt, durch eine geklammerte Liste der Nachfolger-Schritte);
   – beginnend in einer neuen Zeile, einer Liste von Steuerungsanweisungen in der Sprache AWL, deren Auswertungsergebnis die Transitionsbedingung bestimmt;
   – dem abschließenden Schlüsselwort `END_TRANSITION` in einer separaten Zeile.
7) Durch die Anwendung eines *Transitionsnamens* in der Form eines Bezeichners auf der rechten Seite der gerichteten Verbindung. Dieser Bezeichner muss auf ein Konstrukt `TRANSITION...END_TRANSITION` verweisen, das eine der folgenden Einheiten definiert, deren Auswertung mit der Zuweisung eines booleschen Wertes an den Transitionsnamen endet:
   – ein Netzwerk in den Sprache KOP oder FBS;
   – eine Liste von Steuerungsanweisungen in der AWL-Sprache;
   – eine Zuweisung eines booleschen Ausdrucks in der ST-Sprache.

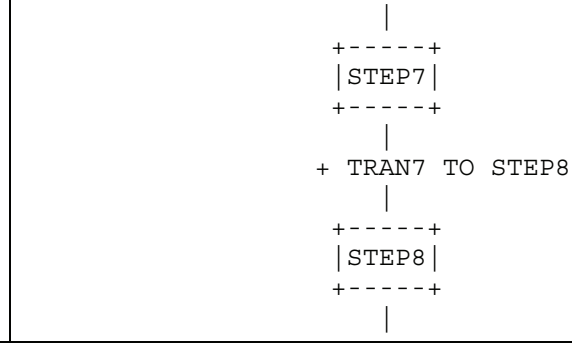
Der *Geltungsbereich* eines Transitionsnamens muss *lokal* zu der Programm-Organisationseinheit sein, in der sich die Transition befindet.

Es muss im Sinne von 1.5.1 ein **Fehler** sein, falls ein „Seiteneffekt“ (wie z. B. die Zuweisung eines Wertes an eine Variable, die nicht der Transitionsname ist) während der Auswertung der Transitionsbedingung auftritt.

Die maximale Anzahl der Transitionen pro Ablaufsprache-Diagramm und pro Schritt sind **implementierungsabhängige Parameter**.

Tabelle 41 – Transitionen und Transitionsbedingungen

Nr.	Beispiel	Beschreibung
1 [a]		Vorgänger-Schritt Transitionsbedingung physikalisch oder logisch der Transition zugeordnet in ST-Sprache (siehe 3.3) Nachfolger-Schritt
2 [a]		Vorgänger-Schritt Transitionsbedingung physikalisch oder logisch der Transition zugeordnet in KOP-Sprache (siehe 4.2) Nachfolger-Schritt
3 [a]		Vorgänger-Schritt Transitionsbedingung physikalisch oder logisch der Transition zugeordnet in FBS-Sprache (siehe 4.3) Nachfolger-Schritt
4 [a]		Gebrauch des Konnektors: Vorgänger-Schritt Transitionskonnektor Nachfolger-Schritt
4a **4b**		Transitionsbedingung KOP-Sprache (siehe 4.2) in FBS-Sprache (siehe 4.3)

Nr.	Beispiel	Beschreibung
5 [b]	STEP STEP7: END_STEP TRANSITION FROM STEP7 TO STEP8 := %IX2.4 & %IX2.3 ; END_TRANSITION STEP STEP8: END_STEP	In Textform wie Eigenschaft 1 in ST-Sprache (siehe 3.3)
6 [b]	STEP STEP7: END_STEP TRANSITION FROM STEP7 TO STEP8: LD %IX2.4 AND %IX2.3 END_TRANSITION STEP STEP8: END_STEP	In Textform wie Eigenschaft 1 in AWL-Sprache (siehe 3.2)
7 [a]		Gebrauch des Transitionsnamens: Vorgänger-Schritt Transitionsname Nachfolger-Schritt
7a	TRANSITION TRAN78 FROM STEP7 TO STEP8: | | | %IX2.4 %IX2.3 TRAN78 | +---||-----||------( )---+ | | END_TRANSITION	Transitionsbedingung in KOP-Sprache (siehe 4.2)
7b	TRANSITION TRAN78 FROM STEP7 TO STEP8: +-------+ | & | %IX2.4---| |--TRAN78 %IX2.3---| | +-------+ END_TRANSITION	Transitionsbedingung in FBS-Sprache (siehe 4.3)
7c	TRANSITION TRAN78 FROM STEP7 TO STEP8: LD %IX2.4 AND %IX2.3 END_TRANSITION	Transitionsbedingung in AWL-Sprache (siehe 3.2)
7d	TRANSITION TRAN78 FROM STEP7 TO STEP8 := %IX2.4 & %IX2.3 ; END_TRANSITION	Transitionsbedingung in ST-Sprache (siehe 3.3)

[a] Falls Eigenschaft 1 der Tabelle 40 unterstützt wird, müssen eine oder mehrere der Eigenschaften 1, 2, 3, 4 oder 7 dieser Tabelle unterstützt werden.

[b] Falls Eigenschaft 2 der Tabelle 40 unterstützt wird, dann müssen eine oder beide der Eigenschaften 5 oder 6 dieser Tabelle unterstützt werden.

2.6.4 Aktionen

Einem Schritt müssen null oder mehrere *Aktionen* zugeordnet sein. Ein Schritt, der keine zugeordneten Aktionen hat, muss betrachtet werden, als habe er eine WARTE-Funktion, d. h. er wartet, bis eine Nachfolger-Transition erfüllt ist.

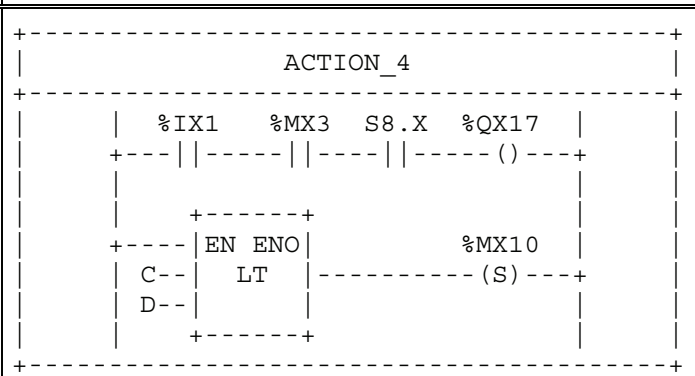
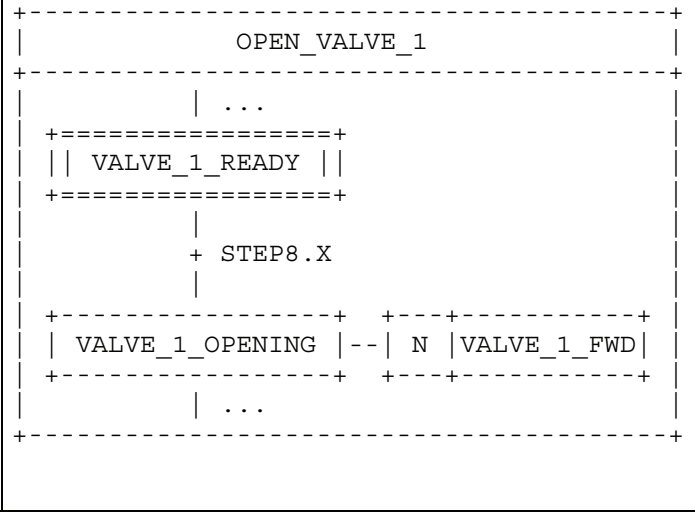
Eine Aktion kann eine boolesche Variable sein, eine Folge von *Anweisungen* in der AWL-Sprache sein, die in 3.2 definiert ist, eine Folge von *Anweisungen* in der ST-Sprache sein, die in 3.3 definiert ist, eine Sammlung von *Strompfaden* in der KOP-Sprache sein, die in 4.2 definiert ist, eine Sammlung von *Netzwerken* in der FBS-Sprache sein, die in 4.3 definiert ist, oder ein *Ablaufsprache-Diagramm* (AS) sein, wie es in 2.6 definiert ist.

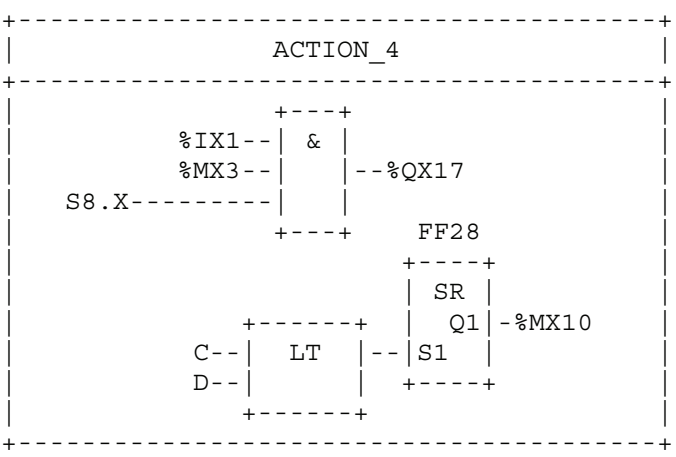
Aktionen müssen mittels einem oder mehreren der Mechanismen deklariert werden, die in 2.6.4.1 definiert sind und sie müssen den *Schritten* mittels Schrittrümpfen in Textform oder grafischen *Aktionsblocks* zugeordnet werden, wie in 2.6.4.2 definiert ist. Die Einzelheiten der Darstellung des Aktionsblocks sind in 2.6.4.3 definiert. Die Steuerung der Aktionen muss durch die *Aktionsbestimmungszeichen* ausgedrückt werden, die in 2.6.4.4 definiert sind.

2.6.4.1 Deklaration

Eine SPS-Implementierung, die AS-Elemente unterstützt, muss einen oder mehrere der Mechanismen bieten, die in Tabelle 42 zur Deklaration von Aktionen definiert sind. Der *Geltungsbereich* der Deklaration einer Aktion muss *lokal* zu der Programm-Organisationseinheit sein, die die Deklaration enthält.

Tabelle 42 – Deklaration von Aktionen [a, b]

<table>
<tr><th>Nr.</th><th colspan="2">Eigenschaft</th></tr>
<tr><td>1</td><td colspan="2">Jede boolesche Variable, die in einem <code>VAR</code> oder <code>VAR_OUTPUT</code> Block deklariert ist oder ihren grafischen Äquivalenten, kann eine Aktion sein.</td></tr>
<tr><th>Nr.</th><th>Beispiel</th><th>Eigenschaft</th></tr>
<tr><td>2l</td><td>  </td><td>Grafische Deklaration in KOP-Sprache (siehe 4.2)</td></tr>
<tr><td>2s</td><td>  </td><td>Einschluss von AS-Elementen in einer Aktion</td></tr>
</table>

Nr.	Eigenschaft	
2f		Grafische Deklaration in FBS-Sprache (siehe 4.3)

3s	`ACTION ACTION_4:`
	`  %QX17 := %IX1 & %MX3 & S8.X ;`
	`  FF28(S1 := (C<D));`
	`  %MX10 := FF28.Q;`
	`END_ACTION`
		Textuelle Deklaration in ST-Sprache (siehe 3.3)

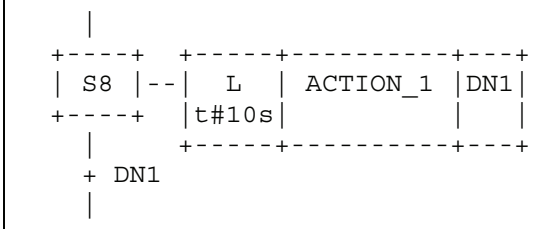
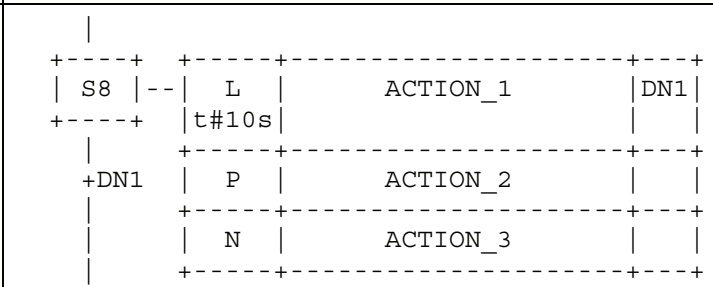
3i	`ACTION  ACTION_4:`
	`  LD  S8.X`
	`  AND  %IX1`
	`  AND  %MX3`
	`  ST  %QX17`
	`  LD  C`
	`  LT  D`
	`  S1  FF28`
	`  LD  FF28.Q`
	`  ST  %MX10`
	`END_ACTION`
		Textuelle Deklaration in AWL-Sprache (siehe 3.2)

ANMERKUNG Der Schrittmerker `S8.X` wird in diesen Beispielen benutzt, um als gewünschtes Ergebnis Folgendes zu erhalten, wenn `S8` deaktiviert wird, ist `%QX17 := 0`.		
[a] Falls Eigenschaft 1 der Tabelle 40 unterstützt wird, müssen eine oder mehrere der Eigenschaften in dieser Tabelle oder Eigenschaft 4 der Tabelle 43 unterstützt werden. [b] Falls Eigenschaft 2 der Tabelle 40 unterstützt wird, dann müssen eine oder mehrere der Eigenschaften 1, 3s oder 3i dieser Tabelle unterstützt werden.		

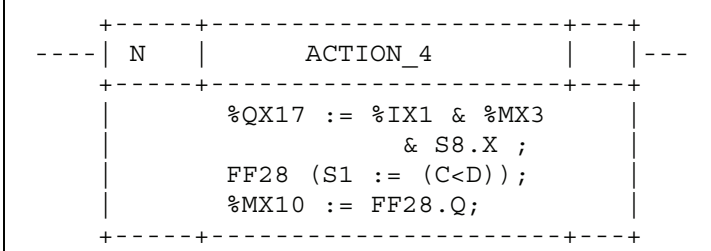
2.6.4.2 Verknüpfung mit Schritten

Eine SPS-Implementierung, die AS-Elemente unterstützt, muss einen oder mehrere der Mechanismen besitzen, die in Tabelle 43 zur Verknüpfung von Aktionen mit Schritten definiert sind. Die maximale Anzahl von Aktionsblocks pro Schritt ist ein **implementierungsabhängiger Parameter**.

Tabelle 43 – Verknüpfung Schritt/Aktion

Nr.	Beispiel	Eigenschaft
1		Aktionsblock physikalisch oder logisch dem Schritt zugeordnet (siehe 2.6.4.3)
2		Aneinander gereihte Aktionsblöcke physikalisch oder logisch dem Schritt zugeordnet

3	`STEP S8:`
	`  ACTION_1(L,t#10s,DN1) ;`
	`  ACTION_2(P) ;`
	`  ACTION_3(N) ;`
	`END_STEP`
		Schritt-Rumpf in Textform

4 [a]		„d“-Feld des Aktionsblock (siehe 2.6.4.3)

[a] Wenn Eigenschaft 4 benutzt wird, kann der entsprechende Aktionsname in keinem anderen Aktionsblock benutzt werden.

2.6.4.3 Aktionsblöcke

Wie in Tabelle 44 gezeigt, ist ein *Aktionsblock* ein grafisches Element zur Kombination einer booleschen Variablen mit einem der *Aktionsbestimmungszeichen*, die in 2.6.4.4 festgelegt sind, um eine Freigabebedingung für eine zugehörige Aktion nach den Regeln in 2.6.4.5 zu erzeugen.

Der Aktionsblock liefert ein Mittel zur optionalen Festlegung von booleschen „Anzeige“-Variablen, die durch das `"c"`-Feld in Tabelle 44 angezeigt sind; diese können durch die festgelegte Aktion gesetzt werden, um deren Abschluss, Zeitüberschreitung, Fehlerbedingungen usw. anzuzeigen. Falls das `"c"`-Feld nicht vorhanden ist und das `"b"`-Feld festlegt, dass die Aktion eine boolesche Variable sein muss, dann muss diese Variable erforderlichenfalls als `"c"`-Variable interpretiert werden. Falls `"c"`-Feld nicht definiert ist und das `"b"` keine boolesche Variable festlegt, dann wird der Wert der „Anzeige“-Variablen als ständig `FALSE` betrachtet.

Wenn Aktionsblöcke grafisch aneinandergereiht sind, wie es in Tabelle 43 veranschaulicht ist, können solche Aneinanderreihungen mehrfache Anzeige-Variablen haben; sie dürfen aber nur eine einzelne gemeinsame boolesche Eingangsvariable haben, die gleichzeitig auf alle aneinander gereihten Blöcke wirken muss.

Ein Aktionsblock kann sowohl mit einem Schritt verknüpft sein als auch als ein grafisches Element in den KOP- und FBS-Sprachen benutzt werden, die in Abschnitt 4 festgelegt sind. In diesem Fall muss der Signal- oder Stromfluss durch einen Aktionsblock den Regeln folgen, die in 4.1.1 festgelegt sind.

Tabelle 44 – Eigenschaften des Aktionsblock

<table>
<tr><th>Nr.</th><th>Eigenschaft</th><th>Grafische Form</th></tr>
<tr><td>1 [a]

2

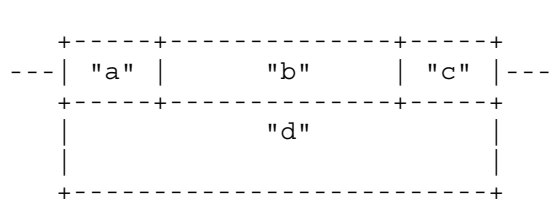
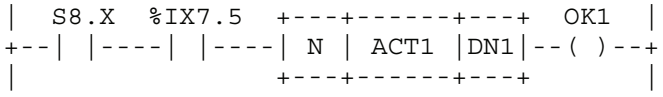
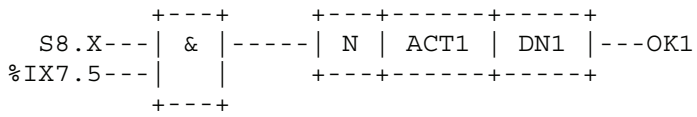
3 [b]

4
5
6
7</td><td>„a“ : Bestimmungszeichen nach 2.6.4.4

„b“ : Aktionsname

„c2“: boolesche „Anzeige“-Variable

„d“ : Aktion in:

– AWL-Sprache (3.2)
– ST-Sprache (3.3)
– KOP-Sprache (4.2)
– FBS-Sprache (4.3)</td><td>  </td></tr>
<tr><th>Nr.</th><th colspan="2">Eigenschaft/Beispiel</th></tr>
<tr><td>8</td><td colspan="2">Gebrauch von Aktionsblöcken in Kontaktplan (siehe 4.2)</td></tr>
<tr><td></td><td colspan="2">  </td></tr>
<tr><td>9</td><td colspan="2">Gebrauch von Aktionsblöcken in Funktionsbausteinsprache (siehe 4.3)</td></tr>
<tr><td></td><td colspan="2">  </td></tr>
<tr><td colspan="3">[a] Feld „a“ kann entfallen, wenn das Bestimmungszeichen „N“ ist.
[b] Feld „c“ kann entfallen, wenn keine Anzeige-Variable benutzt wird.</td></tr>
</table>

2.6.4.4 Aktionsbestimmungszeichen

Zugehörig zu jeder Verknüpfung von Schritt/Aktion, die in 2.6.4.2 definiert ist, oder jedem Auftreten eines Aktionsblocks, wie in 2.6.4.3 definiert, muss ein *Bestimmungszeichen* für Aktionen verwendet werden. Der Wert dieses Bestimmungszeichen muss einer der Werte sein, die in Tabelle 45 aufgeführt sind. Zusätzlich müssen die Bestimmungszeichen L, D, SD, DS und SL eine zugehörige Zeitdauer vom Typ TIME haben.

ANMERKUNG IEC 60848 gibt informative Definitionen und Beispiele für den Gebrauch dieser Bestimmungszeichen. Diese Norm gibt diese Definitionen formal an, dabei wird das Bestimmungszeichen S neu definiert und das Zeichen R eingeführt. Die Steuerung von Aktionen, die diese Bestimmungszeichen benutzen, ist im folgenden Abschnitt definiert, und zusätzliche Beispiele für ihren Gebrauch sind in Anhang F angegeben.

Tabelle 45 – Aktionsbestimmungszeichen

Nr.	Bestimmungszeichen	Erläuterung
1	kein	Nicht-gespeichert (kein Zeichen)
2	N	Nicht-gespeichert (**N**on-stored)
3	R	vorrangiges Rücksetzen (overriding **R**eset)
4	S	Setzen (gespeichert) (**S**et stored)
5	L	Zeitbegrenzt (time **L**imited)
6	D	Zeitverzögert (time **D**elayed)
7	P	Impuls (Flanke) (**P**ulse)
8	SD	gespeichert und zeitverzögert (**S**tored and time **D**elayed)
9	DS	verzögert und gespeichert (**D**elayed and **S**tored)
10	SL	gespeichert und zeitbegrenzt (**S**tored and time **L**imited)
11	P1	Puls (Steigende Flanke) (**P**ulse rising edge)
12	P0	Puls (Fallende Flanke) (**P**ulse falling edge)

2.6.4.5 Steuerung von Aktionen

Die Steuerung von Aktionen muss funktional der Anwendung folgender Regeln entsprechen:

1) Jede Aktion muss mit dem funktionalen Äquivalent einer Instanz des Funktionsbausteins ACTION_CONTROL, der in den Bildern 14 und 15 definiert ist, verknüpft sein. Falls die Aktion als boolesche Variable deklariert ist, wie in 2.6.4.1 definiert ist, muss der Ausgang Q dieses Blocks dem Zustand dieser booleschen Variablen entsprechen. Wenn die Aktion als eine Sammlung von Anweisungen oder Netzwerken deklariert ist, wie in 2.6.4.1 definiert, dann muss diese Sammlung so lange ausgeführt werden, wie der Ausgang A (Aktivierung) des Funktionsbausteins ACTION_CONTROL den Wert BOOL#1 behält. In diesem Fall kann auf den Zustand des Ausgangs Q („Aktionsmerker“ genannt) innerhalb der Aktion durch Lesen einer Read-only-Variablen zugegriffen werden, die die Form eines Verweises auf den Q Ausgang einer Funktionsbaustein-Instanz hat, deren Instanzname der gleiche wie der entsprechende Aktionsname, z. B. ACTION1.Q, ist.

ANMERKUNG 1 Die Bedingung Q=FALSE wird gewöhnlich von einer Aktion benutzt, um zu bestimmen, dass sie das letzte Mal während ihrer Aktivierung ausgeführt wird.

ANMERKUNG 2 Der Wert von Q wird immer während der Ausführung von Aktionen FALSE sein, die durch die Bestimmungszeichen P0 und P1 aufgerufen werden.

ANMERKUNG 3 Der Wert von A wird für nur eine Ausführung einer Aktion TRUE sein, die durch die Bestimmungszeichen P0 und P1 aufgerufen werden. Für alle anderen Bestimmungszeichen wird A während eines zusätzlichen Aufrufs nach der fallenden Flanke von Q den Wert TRUE haben.

ANMERKUNG 4 Der Zugriff auf das funktionale Äquivalent der Ausgänge Q oder A eines Funktionsbausteins ACTION_CONTROL von außerhalb der zugehörigen Aktion ist eine **implementierungsabhängige** Eigenschaft.

ANMERKUNG 5 Der Hersteller kann eine einfachere Implementierung wählen als in Bild 15 b) gezeigt ist. In diesem Fall, wenn die Aktion als eine Sammlung von Anweisungen oder Netzwerken deklariert ist, wie es in 2.6.4.1 definiert ist, dann muss diese Sammlung ständig bearbeitet werden, solange der Ausgang Q des Funktionsbausteins ACTION_CONTROL auf BOOL#1 steht. In jedem Fall muss der Hersteller festlegen, welche der Eigenschaften, die in Tabelle 45 a) angegeben sind, unterstützt werden.

2) Ein boolescher Eingang am Baustein ACTION_CONTROL einer Aktion muss sozusagen eine *Verknüpfung* mit einem Schritt, wie in 2.6.4.2 definiert, oder mit einem Aktionsblock haben, wie in 2.6.4.3 definiert, falls das entsprechende Bestimmungszeichen mit dem Eingangsnamen (N, R, S, L, D, P, P0, P1, SD, DS oder SL) übereinstimmt. Die Verknüpfung wird *aktiv* genannt, falls der zugehörige Schritt aktiv ist oder falls der Eingang des zugehörigen Aktionsblocks den Wert BOOL#1 hat. Die *aktiven Ver-*

knüpfungen einer *Aktion* sind gleichbedeutend mit der Menge der *aktiven Verknüpfungen* aller Eingänge mit ihrem Funktionsbaustein ACTION_CONTROL.

Ein boolescher Eingang des Bausteins ACTION_CONTROL muss den Wert BOOL#1 haben, falls dieser mindestens eine aktive Verknüpfung hat, und andernfalls den Wert BOOL#0.

3) Der Wert des Eingangs T eines Bausteins ACTION_CONTROL muss den Wert der Zeitdauer eines zeitabhängigen Bestimmungszeichens (L, D, SD, DS oder SL) einer aktiven Verknüpfung haben. Falls eine solche Verknüpfung nicht existiert, muss der Eingang T den Wert t#0s haben.

4) Es muss ein **Fehler** im Sinne des Abschnitts 1.5.1 sein, falls eine oder mehrere der folgenden Bedingungen auftreten:

   a) Mehr als eine *aktive Verknüpfung* einer Aktion hat ein zeitabhängiges Bestimmungszeichen (L, D, SD, DS oder SL).

   b) Der Eingang SD eines ACTION_CONTROL Bausteins hat den Wert BOOL#1, wenn der Q1 Ausgang seines SL_FF Bausteins den Wert BOOL#1 hat.

   c) Der Eingang SL eines ACTION_CONTROL Bausteins hat den Wert BOOL#1, wenn der Q1 Ausgang seines SD_FF Bausteins den Wert BOOL#1 hat.

5) Es ist nicht erforderlich, dass der ACTION_CONTROL Baustein selbst implementiert ist, sondern nur, dass die Steuerung der Aktionen gleichwertig mit den vorhergehenden Regeln ist. Nur diejenigen Teile der Aktionssteuerung, die für eine bestimmte Aktion passend sind, müssen instanziiert werden, wie in Bild 16 veranschaulicht ist. Insbesondere ist zu bemerken, dass einfache Funktionen mit MOVE (:=) oder booleschem OR zur Steuerung von booleschen Aktionsvariablen ausreichen, sofern die Verknüpfungen der letzteren nur "N" als Bestimmungszeichen besitzen.

a) **b)**

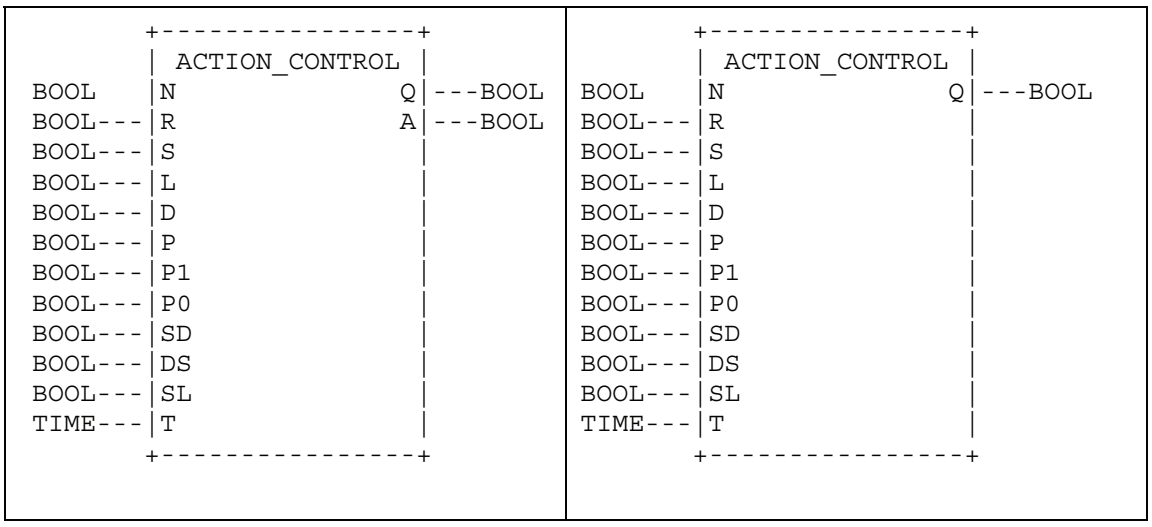
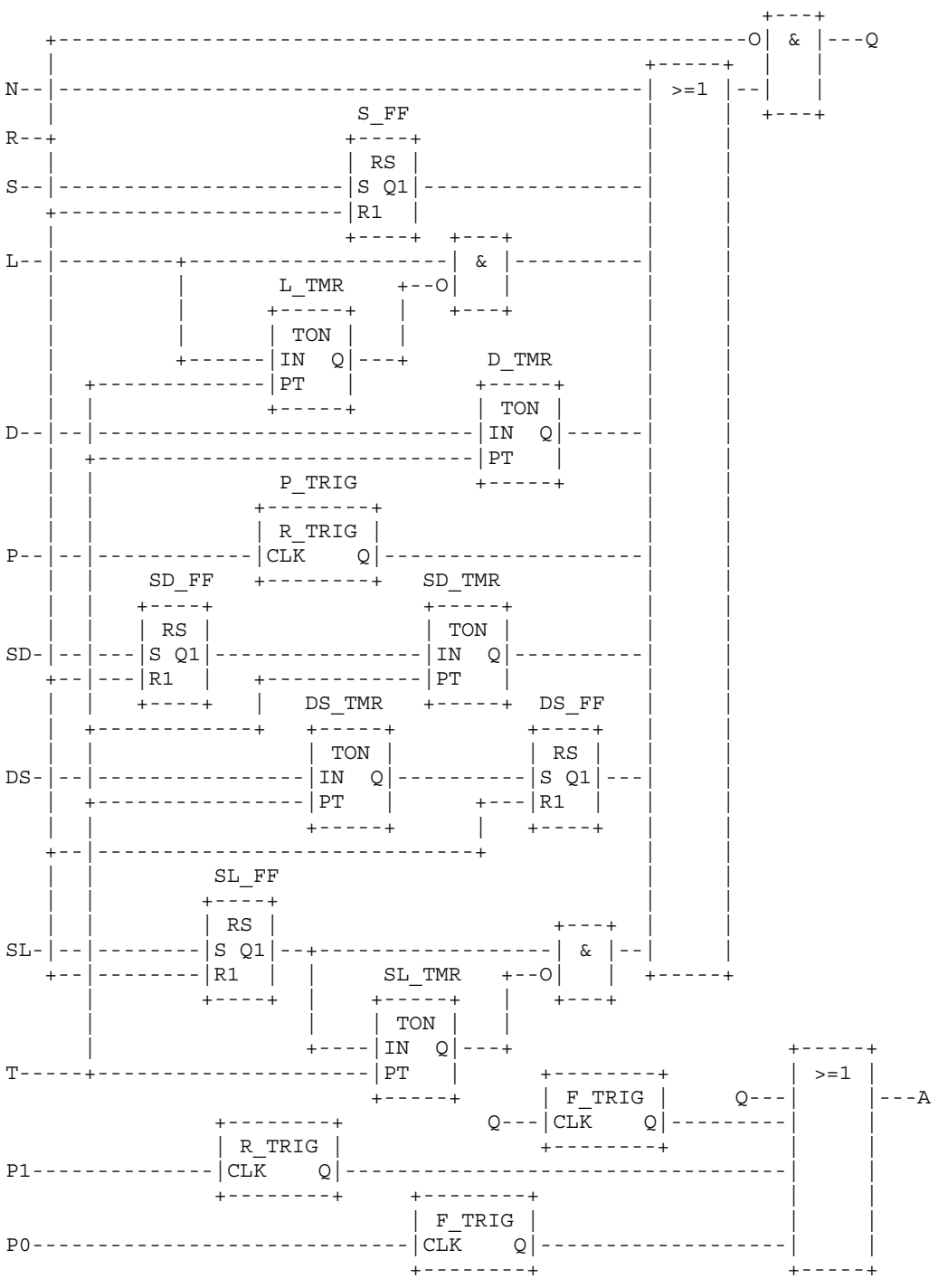


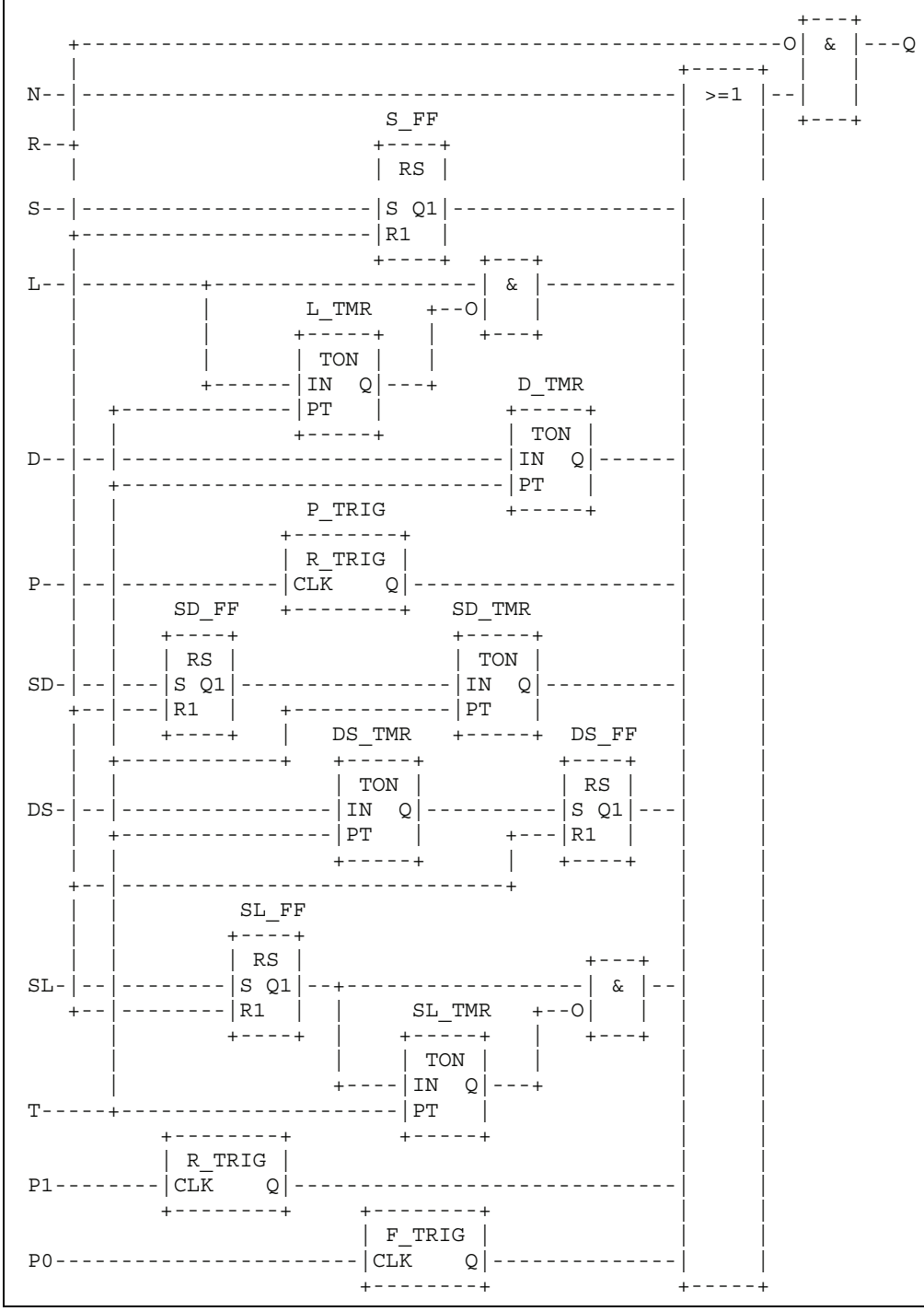
Bild 14 – Funktionsbaustein ACTION_CONTROL – Externe Schnittstelle (Unsichtbar für den Anwender)
a) Mit „letztem Durchlauf“ – siehe Bild 15 a); b) Ohne „letzten Durchlauf“ – siehe Bild 15 b)



ANMERKUNG 1 Die Instanzen dieses Funktionsbausteins sind für den Anwender nicht sichtbar.

ANMERKUNG 2 Die Außenschnittstelle dieses Funktionsbaustein-Typs ist in Bild 14 a) angegeben.

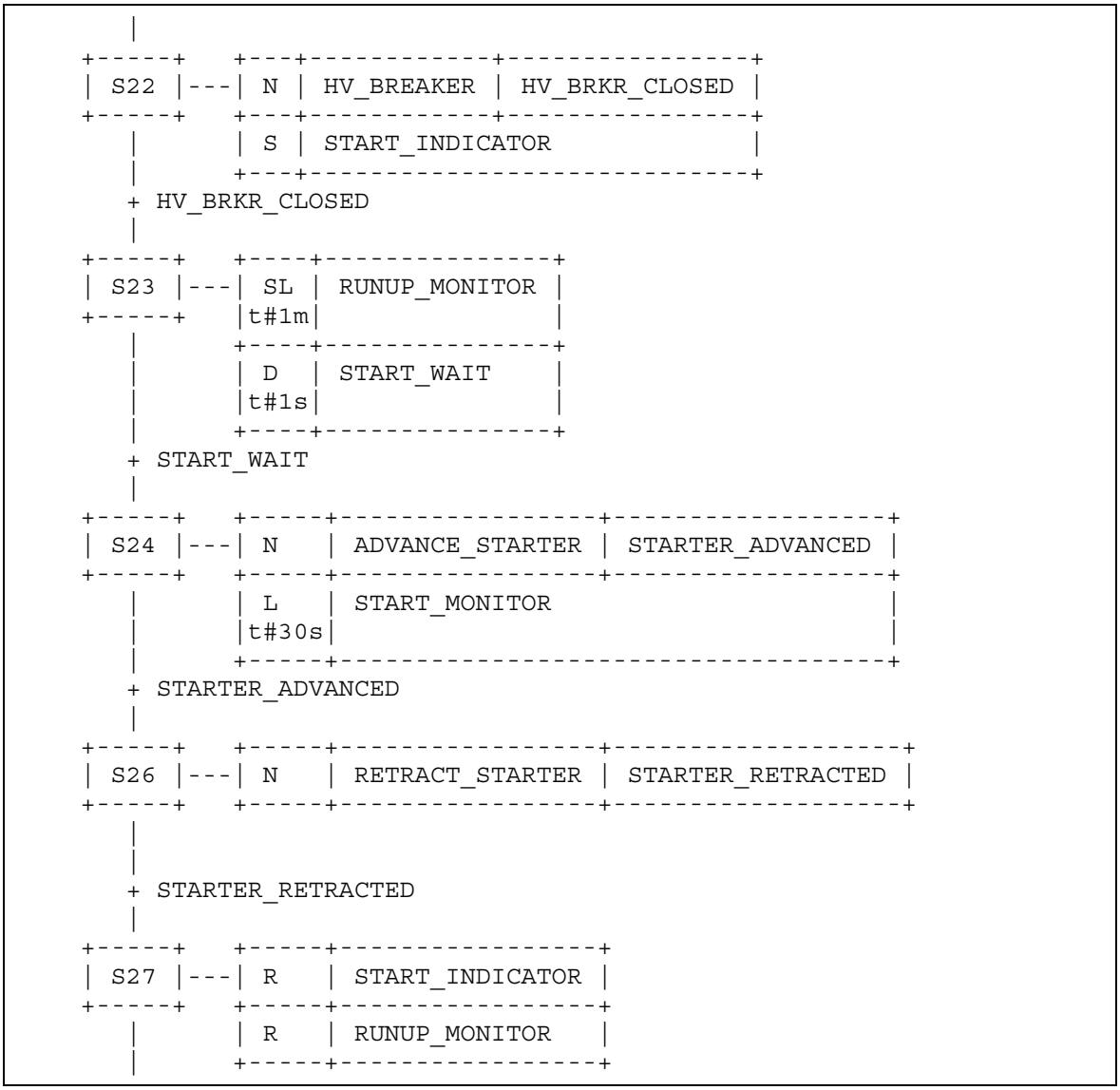
Bild 15 a) – `ACTION_CONTROL` Funktionsbaustein-Rumpf mit „letztem Durchlauf“



ANMERKUNG 1
Die Instanzen dieses Funktionsbausteins sind für den Anwender nicht sichtbar.

ANMERKUNG2
Die Außenschnittstelle dieses Funktionsbaustein-Typs ist in Bild 14 b) angegeben.

Bild 15 b) – `ACTION_CONTROL` Funktionsbaustein-Rumpf ohne „letzten Durchlauf”



ANMERKUNG Das vollständige AS-Netzwerk und seine zugehörigen Deklarationen sind in diesem Beispiel nicht gezeigt.

Bild 16 a) – Beispiel für Aktionssteuerung – AS-Darstellung

Tabelle 45 a) – Eigenschaften der Aktionssteuerung

Nr.	Beschreibung
1	für Bilder 14 a) und 15 a)
2	für Bilder 14 b) und 15 b)

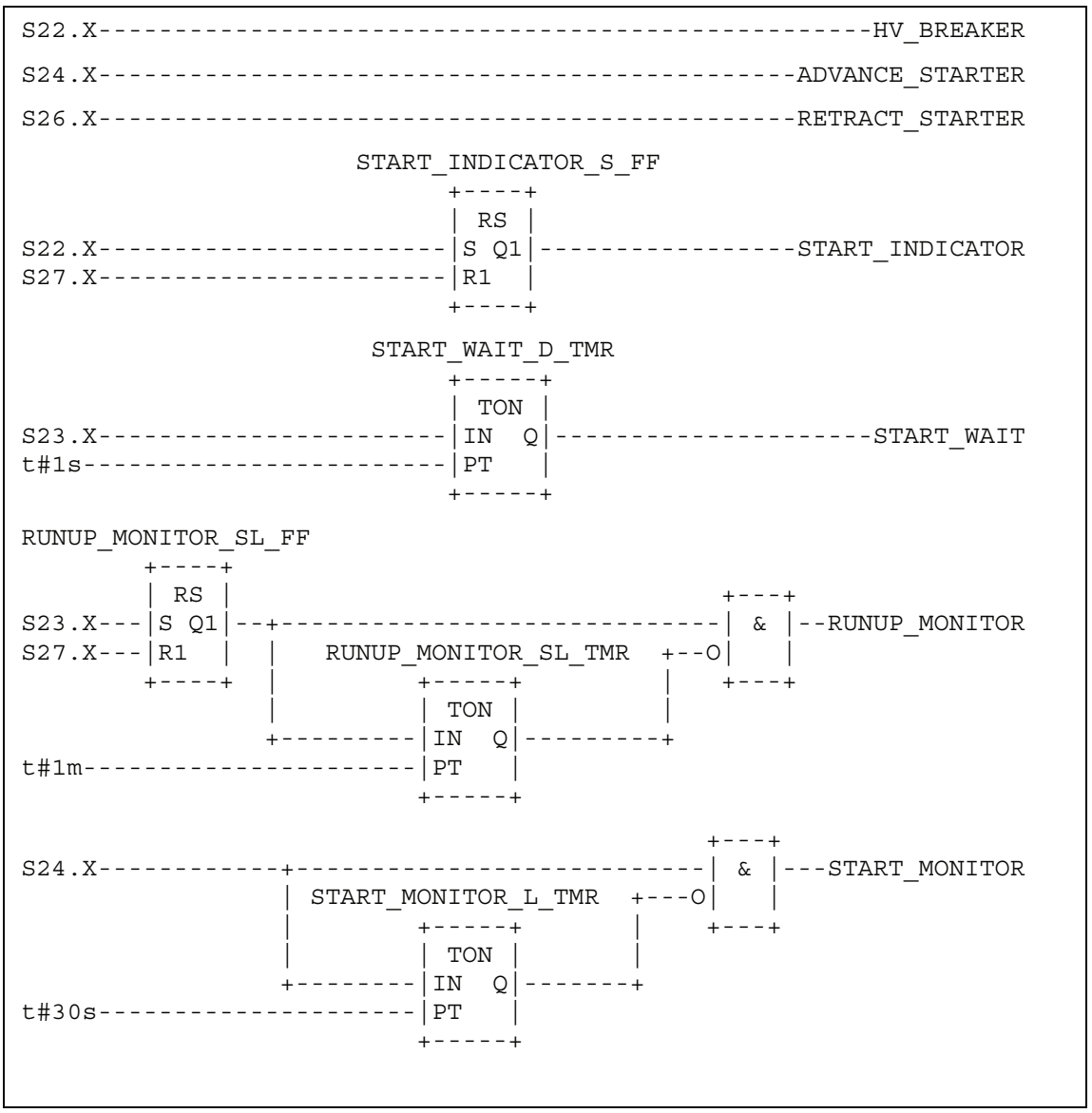


Bild 16 b) – Beispiel für Aktionssteuerung – Funktional gleichartig

2.6.5 Ablaufregeln

Der *Anfangszustand* eines AS-Netzwerks ist durch den *Anfangsschritt* charakterisiert, der bei Initialisierung des Programms oder Funktionsbausteins, welche das Netzwerk enthalten, in aktivem Zustand ist.

Die *Abläufe* der aktiven Schrittzustände finden längs der *gerichteten Verbindungen* statt, ausgelöst durch das *Schalten* einer oder mehrerer *Transitionen.*

Eine Transition ist *freigegeben*, wenn alle vorausgehenden Schritte aktiv sind, die mit dem entsprechenden Transitionssymbol durch eine gerichtete Verbindung verbunden sind. Das Weiterschalten einer Transition geschieht dann, wenn die Transition freigegeben ist und wenn die zugehörige Transitionsbedingung erfüllt ist.

Das Weiterschalten einer Transition führt zur *Deaktivierung* (oder „Rücksetzen“) aller unmittelbar vorangehenden Schritte, die über eine gerichtete Verbindung mit dem entsprechenden Transitionssymbol verbunden sind, gefolgt von der *Aktivierung* aller unmittelbar folgenden Schritte.

Der Wechsel Schritt/Transition und Transition/Schritt muss für die Verbindungen von AS-Elementen immer folgendermaßen ablaufen:

– Zwei Schritte dürfen nie direkt verbunden sein; sie müssen immer durch eine Transition getrennt sein.
– Zwei Transitionen dürfen nie direkt verbunden sein; sie müssen immer durch einen Schritt getrennt sein.

Wenn das Weiterschalten einer Transition zur Aktivierung von mehreren Schritten zur gleichen Zeit führt, dann werden die Ablaufketten, zu denen diese Schritte gehören, *Simultan-Ketten* genannt. Nach ihrer gleichzeitigen Aktivierung erfolgt der Ablauf jeder dieser Ketten unabhängig voneinander. Um die besondere Art derartiger Konstrukte hervorzuheben, muss die Verzweigung und Zusammenführung von Simultanketten durch eine doppelte horizontale Linie angezeigt werden.

Es muss ein **Fehler** sein, wenn die Möglichkeit bestehen kann, dass nicht-priorisierte Transitionen in einer Auswahl-Verzweigung, wie in Eigenschaft 2a der Tabelle 46 gezeigt ist, gleichzeitig erfüllt sind. Der Anwender darf Vorkehrungen treffen, um diesen Fehler, wie in den Eigenschaften 2b und 2c in Tabelle 46 gezeigt, zu vermeiden.

Tabelle 46 definiert die Syntax und Semantik der möglichen Kombinationen von Schritten und Transitionen.

Die Weiterschaltzeit einer Transition kann man theoretisch als so kurz wie möglich betrachten, sie kann aber niemals null werden. In der Praxis wird die Weiterschaltzeit durch die Implementierung der SPS bestimmt. Aus dem gleichen Grund kann die Dauer einer Schritt-Aktivierung niemals als null betrachtet werden.

Mehrere Transitionen, die gleichzeitig schalten können, müssen gleichzeitig innerhalb der vorgegebenen Zeitanforderungen der jeweiligen SPS-Implementierung und der Prioritätsregeln schalten, die in Tabelle 46 definiert sind.

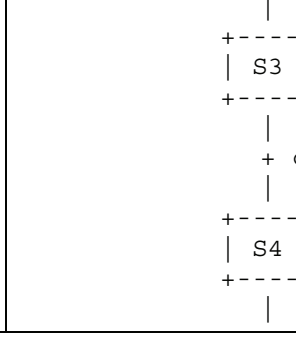
Die Prüfung der Bedingung(en) der Nachfolger-Transition(en) eines aktiven Schritts darf so lange nicht durchgeführt werden, bis sich die Wirkungen der Schrittaktivierung über die Programm-Organisationseinheit verbreitet haben, in der der Schritt deklariert ist.

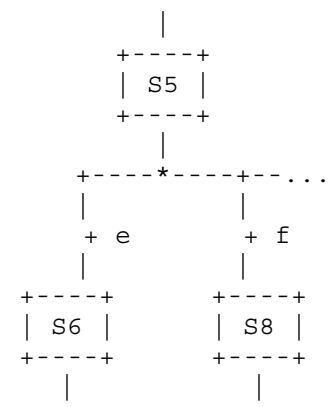
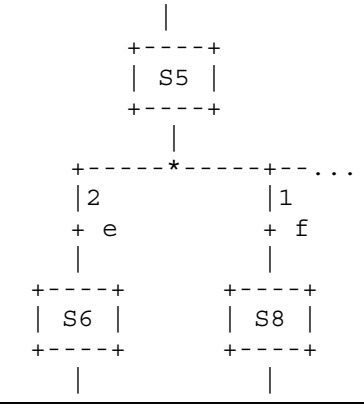
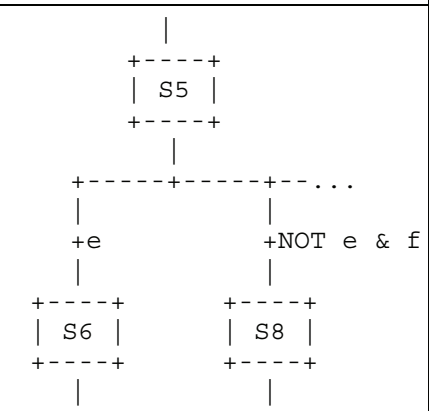
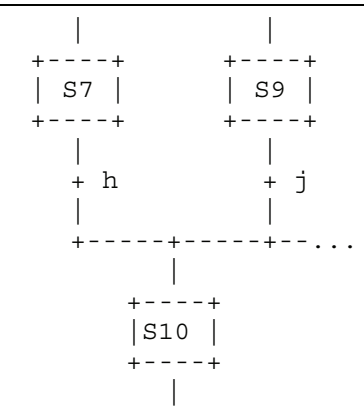
Bild 17 veranschaulicht die Anwendung dieser Regeln. In diesem Bild ist der aktive Zustand eines Schrittes durch einen Stern (*) im entsprechenden Block angezeigt. Diese Schreibweise ist nur zur Veranschaulichung benutzt und ist keine geforderte Eigenschaft der Sprache.

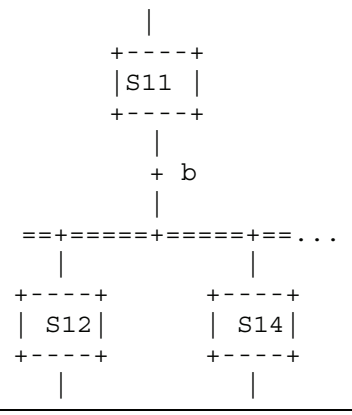
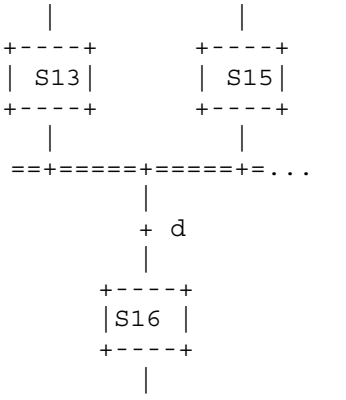
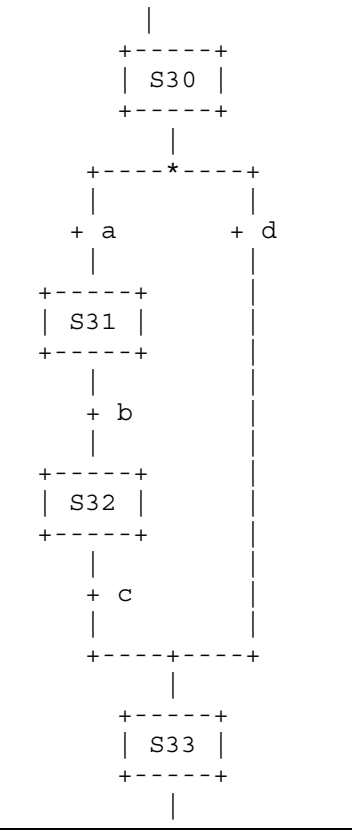
Die Anwendung der Regeln, die in diesem Abschnitt angegeben sind, können nicht die Formulierung von „unsicheren“ Ablaufketten verhindern, wie diejenige, die in Bild 18 a) gezeigt ist, die eine unkontrollierte Verbreitung von Tokens darstellen soll. Ebenso kann die Anwendung dieser Regeln nicht die Formulierung von „unerreichbaren“ Ketten verhindern, wie die in Bild 18 b) gezeigte, die ein „verklemmtes“ Verhalten darstellen soll. Das SPS-System muss die Existenz derartiger Bedingungen als **Fehler**, wie in 1.5.1 definiert, behandeln.

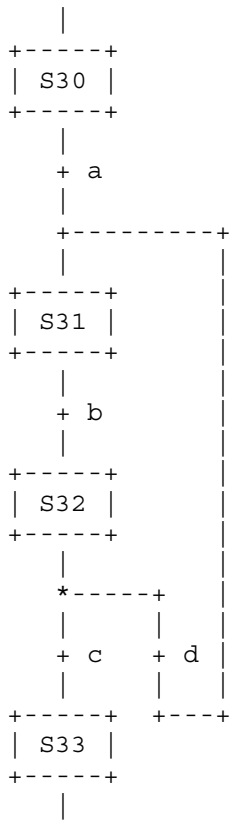
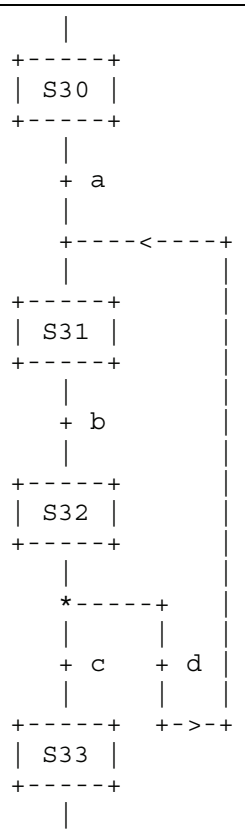
Die maximal erlaubte Breite von Konstrukten der „Verzweigungen“ und „Zusammenführungen“ in Tabelle 46 sind **implementierungsabhängige Parameter**.

Tabelle 46 – Kettenablauf

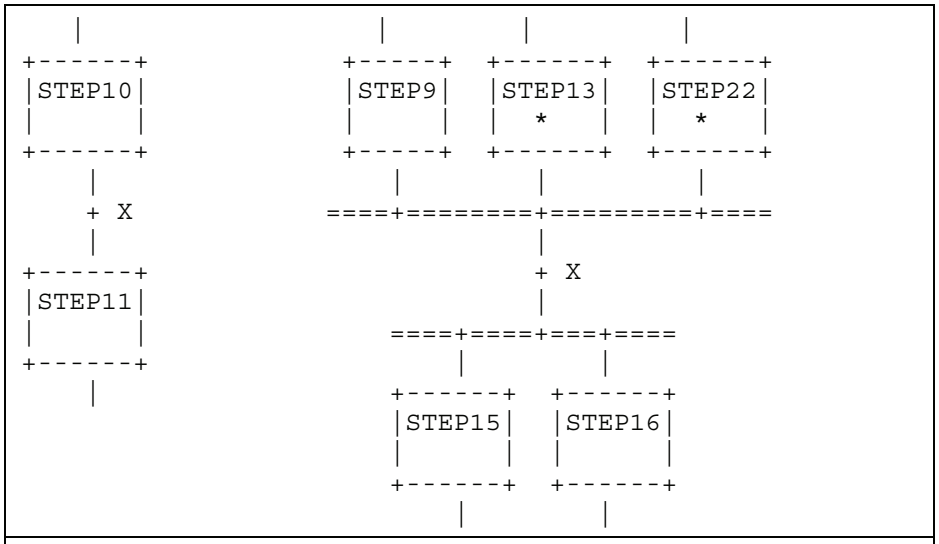
Nr.	Beispiel	Regel
1		**Einfache Kette:** Der Wechsel Schritt-Transition wird als Folge wiederholt. **Beispiel:** Ein Ablauf von Schritt `S3` zu Schritt `S4` findet nur dann statt, wenn Schritt `S3` in aktivem Zustand ist und die Transitionsbedingung `C` wahr ist.

Nr.	Beispiel	Regel
2a		**Verzweigung bei Kettenauswahl:** Eine Auswahl zwischen mehreren Ketten wird durch so viele Transitionssymbole *unter* der horizontalen Linie dargestellt, wie es unterschiedliche mögliche Abläufe gibt. Der Stern (*) gibt an, dass die Transitionen von links nach rechts bearbeitet werden. **Beispiel:** Ein Ablauf von S5 nach S6 findet nur statt, wenn S5 aktiv ist und die Transitionsbedingung e wahr ist oder von S5 nach S8, nur wenn S5 aktiv ist und f wahr und e falsch ist.
2b		**Verzweigung bei Kettenauswahl:** Der Stern gefolgt von nummerierten Zweigen, zeigt eine anwender-definierte Priorität der Transitionsauswertung an, dabei hat der Zweig mit der niedersten Nummer die höchste Priorität. **Beispiel:** Ein Ablauf von S5 nach S8 findet nur statt, wenn S5 aktiv ist und die Transitionsbedingung f wahr ist oder von S5 nach S6 nur, wenn S5 aktiv ist und e wahr ist und f falsch ist.
2c		**Verzweigung einer Kettenauswahl:** Der Verbindungspunkt der Verzweigung zeigt an, dass der Anwender selbst dafür sorgen muss, dass die Transitionsbedingungen sich gegenseitig ausschließen, wie es in IEC 60848 festgelegt ist. **Beispiel:** Ein Ablauf von S5 nach S6 findet nur statt, wenn S5 aktiv ist und die Transitionsbedingung e wahr ist oder von S5 nach S8, nur wenn S5 aktiv ist und e falsch ist und f wahr ist.
3		**Zusammenführung einer Kettenauswahl:** Das Ende einer Kettenauswahl wird durch so viele Transitionssymbole *über* der horizontalen Linie dargestellt, wie es Auswahlpfade gibt, die zu beenden sind. **Beispiel:** Ein Ablauf von S7 nach S10 findet nur statt, wenn S7 aktiv ist und die Transitionsbedingung h wahr ist oder von S9 nach S10 nur, wenn S9 aktiv ist und j wahr ist.

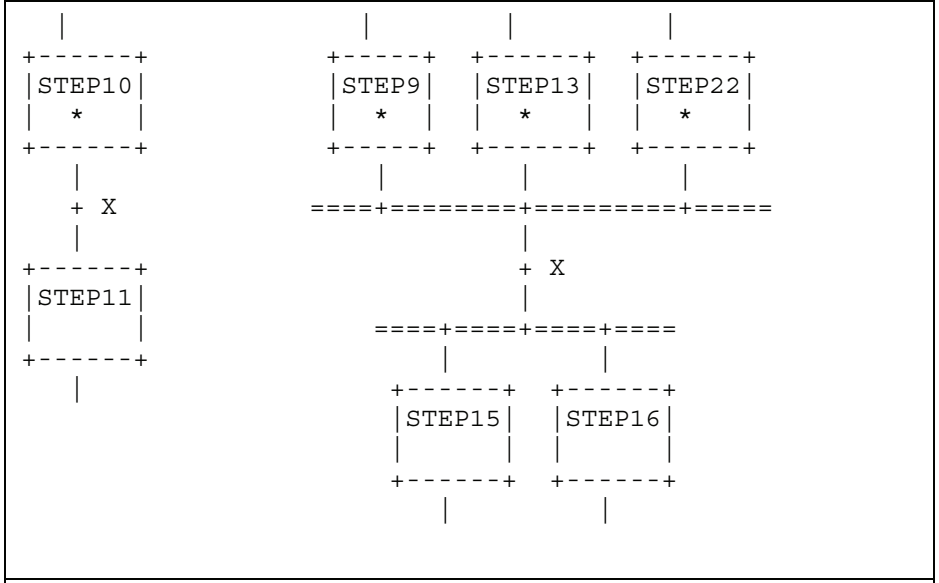
Nr.	Beispiel	Regel
4		**Verzweigung von Simultanketten:** Es ist nur ein gemeinsames Transitionssymbol unmittelbar *über* der doppelten horizontalen Synchronisationslinie möglich. **Beispiel:** Ein Ablauf von `S11` nach `S12`, `S14...` findet nur statt, wenn `S11` aktiv ist und die Transitionsbedingung `b`, die zur gemeinsamen Transition gehört, ebenfalls wahr ist. Nach der simultanen Aktivierung von `S12`, `S14` usw. laufen die Ketten unabhängig voneinander ab.
		**Zusammenführung von Simultanketten:** Es ist nur ein gemeinsames Transitionssymbol unmittelbar *unter* der doppelten horizontalen Synchronisationslinie möglich. **Beispiel:** Ein Ablauf von `S13`, `S15...` nach `S16` findet nur statt, wenn alle Schritte aktiv sind, die oberhalb und verbunden mit der doppelten horizontalen Linie sind und die Transitionsbedingung `d`, die zur gemeinsamen Transition gehört, wahr ist.
5a **5b** **5c**		**Ketten-Sprung:** Ein „Ketten-Sprung“ ist ein Sonderfall der Kettenauswahl (Eigenschaft 2), bei der einer oder mehrere Zweige keine Schritte enthalten. Die Eigenschaften 5a, 5b und 5c entsprechen den Darstellungsmöglichkeiten, die in 2a, 2b bzw. in 2c angegeben sind. **Beispiel:** (Eigenschaft 5a ist gezeigt) Ein Ablauf von `S30` nach `S33` findet nur statt, wenn `a` falsch und `d` wahr ist, d. h. die Kette (`S31`, `S32`) wird übersprungen.

Nr.	Beispiel	Regel
6a 6b 6c		**Ketten-Schleife:** Eine Ketten-Schleife ist ein Sonderfall einer Kettenauswahl (Eigenschaft 2), in der ein oder mehrere Zweige zu einem Vorgängerschritt zurückführen. Die Eigenschaften 6a, 6b und 6c entsprechen den Darstellungsmöglichkeiten, die in 2a, 2b bzw. 2c angegeben sind. **Beispiel:** (Eigenschaft 6a ist gezeigt) Ein Ablauf von `S32` nach `S31` findet statt, wenn `c` falsch und `d` wahr ist; d. h. die Kette (`S31`, `S32`) wird wiederholt.
7		**Richtungspfeile:** Wenn es zur Klarheit nötig ist, kann das „kleiner als“ (`<`) Zeichen des Zeichensatzes, der in 2.1.2 definiert ist, benutzt werden, um einen Steuerfluss von rechts nach links anzuzeigen und das „größer als“ (`>`) Zeichen, um einen Steuerfluss von links nach rechts darzustellen. Wenn diese Eigenschaft benutzt wird, muss das entsprechende Zeichen zwischen zwei `"-"` Zeichen angeordnet sein; d. h. als Zeichenfolge `"-<-"` oder `"->-"`, wie im zugehörigen Beispiel gezeigt.

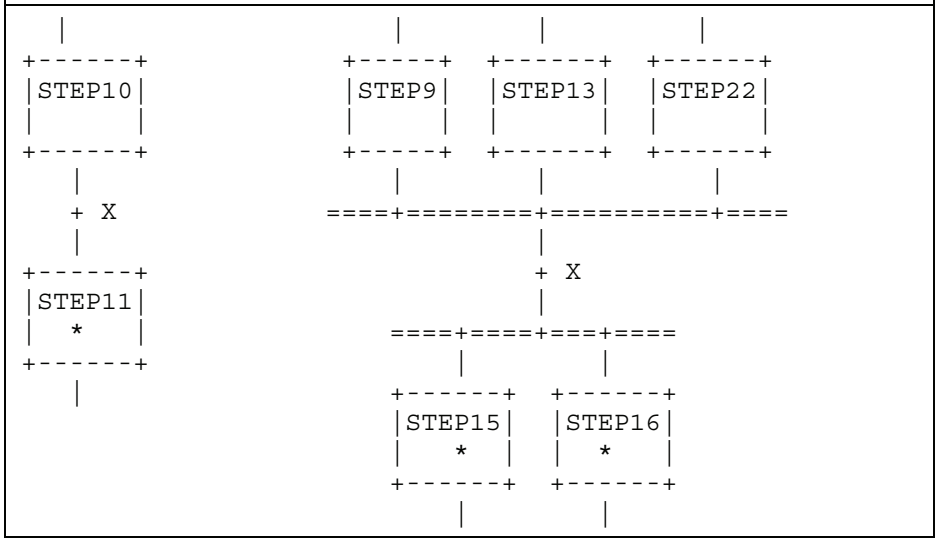
a) Transition nicht freigegeben (X = 0 oder 1)



b) Transition freigegeben, aber nicht weitergeschaltet (X = 0)



c) Transition weitergeschaltet (X = 1)



ANMERKUNG In diesem Bild ist der aktive Zustand eines Schrittes durch einen Stern (*) im entsprechenden Block angezeigt. Diese Schreibweise wird nur zur Veranschaulichung benutzt und ist keine geforderte Eigenschaft der Sprache.

Bild 17 – AS-Ablaufregeln

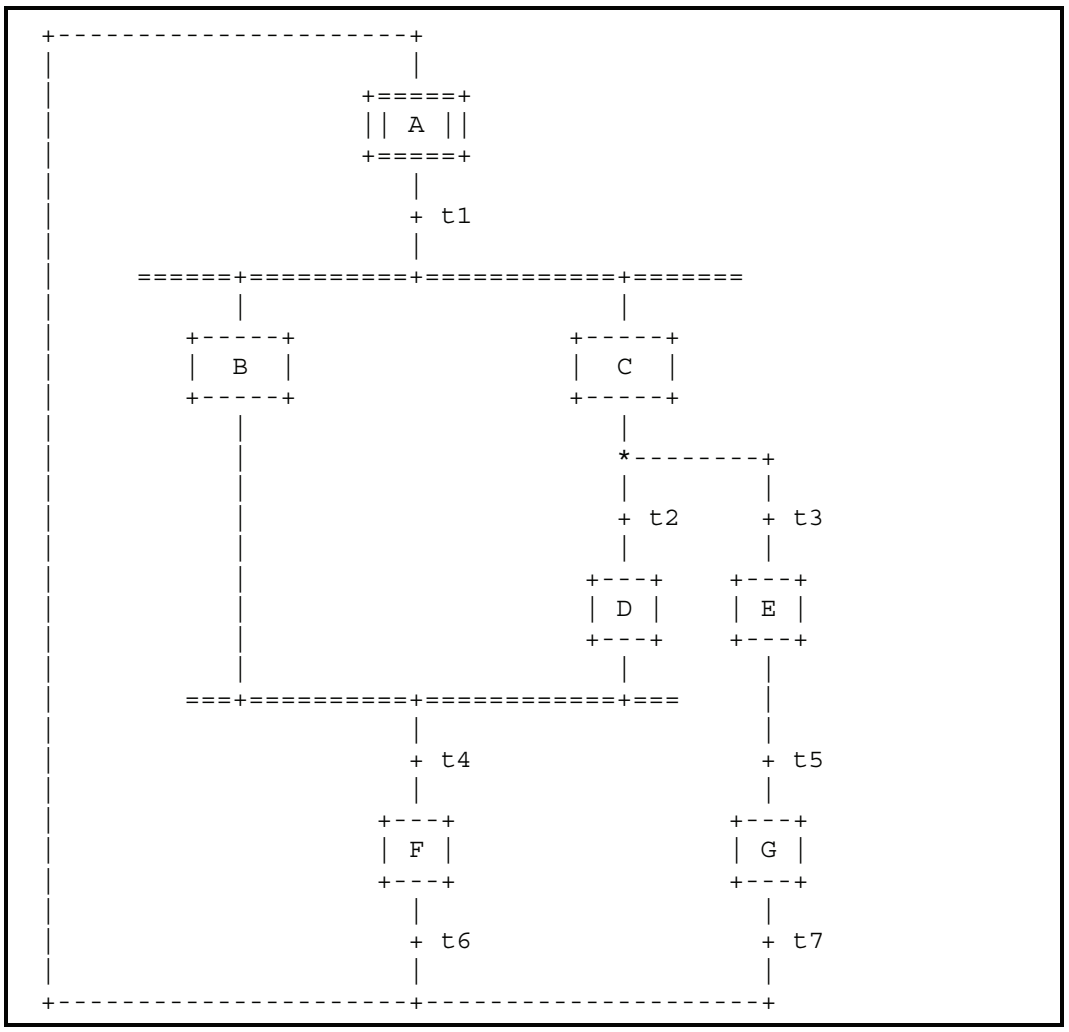


Bild 18 a) – Beispiele für Fehler in Anlaufsprache: Eine „unsichere“ Ablaufkette (siehe 2.6.5)

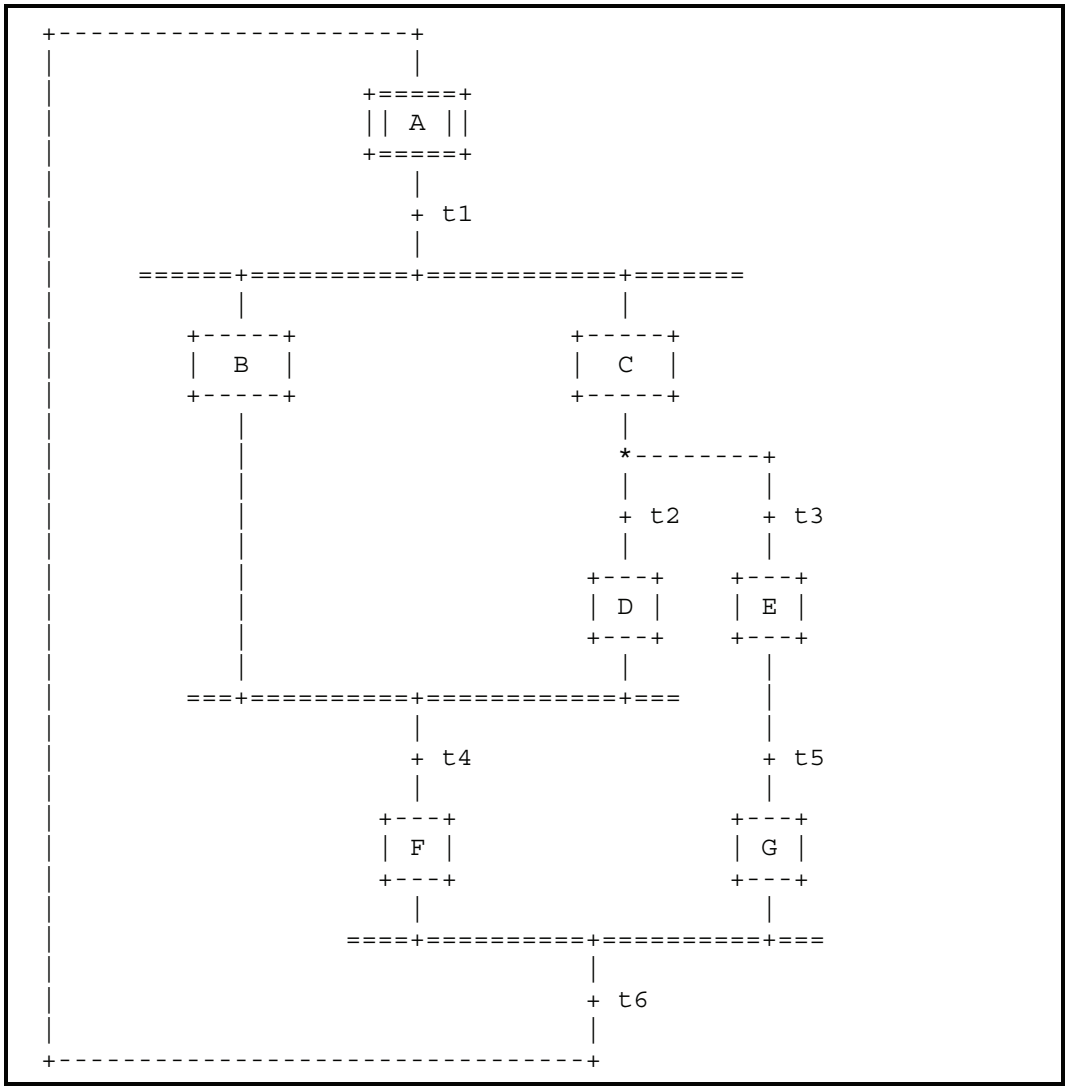


Bild 18 b) – Beispiele für Fehler in der Ablaufsprache: Eine „unerreichbare“ Ablaufkette (siehe 2.6.5)

2.6.6 Kompatibilität von AS-Elementen

Ablaufketten können grafisch oder als Text durch Anwenden der oben definierten Elemente dargestellt werden. Tabelle 47 fasst zur Übersicht die Elemente zusammen, die grafisch bzw. in Textdarstellung zusammenpassen.

Tabelle 47 – Kompatibilität der Eigenschaften der Ablaufkette

Tabelle	Grafische Darstellung	Text-Darstellung
40	1, 3a, 3b, 4	2, 3a, 4
41	1, 2, 3, 4, 4a, 4b, 7, 7a, 7b	5, 6, 7c, 7d
42	1, 2l, 2s, 2f	3s, 3i
43	1, 2, 4	3
44	1 bis 9	--
45	1 bis 10	1 bis 10 (in Text äquivalent)
46	1 bis 7	1 bis 6
57	Alle	--

2.6.7 Anforderungen an die Normerfüllung

Um den Anspruch auf Normerfüllung bei den Anforderungen von 2.6 zu erfüllen, müssen die Elemente unterstützt werden, die in Tabelle 48 gezeigt sind und die Kompatibilitätsanforderungen beachtet werden, die in 2.6.6 definiert sind.

Tabelle 48 – Mindestanforderungen der Normerfüllung

Tabelle	Grafische Darstellung	Text-Darstellung
40	1	2
41	1 oder 2 oder 3 oder (4 und (4a oder 4b)) oder (7 und (7a oder 7b oder 7c oder 7d))	5 oder 6
42	1 oder 2l oder 2f	3s oder 3i
43	1 oder 2 oder 4	3
45	1 oder 2	1 oder 2
46	1 und (2a oder 2b oder 2c) und 3 und 4	gleich (Text äquivalent)
57	(1 oder 2) und (3 oder 4) und (5 oder 6) und (7 oder 8) und (9 oder 10) und (11 oder 12)	Nicht erforderlich

2.7 Konfigurationselemente

Wie in 1.4.1 beschrieben, besteht eine *Konfiguration* aus *Ressourcen*, *Tasks* (die innerhalb von *Ressourcen* definiert sind), *globalen Variablen* und *Zugriffspfaden* und Instanz-spezifischen Initialisierungen. Jedes dieser Elemente ist im Einzelnen in diesem Abschnitt definiert.

Ein grafisches Beispiel einer einfachen Konfiguration ist in Bild 19 a) gezeigt. Das Gerippe der Deklarationen für die entsprechenden Funktionsbausteine und Programme ist in Bild 19 b) angegeben. Dieses Bild dient als Bezugspunkt für die Beispiele der Konfigurationselemente, die im Weiteren dieses Abschnitts in Bild 20 angegeben sind.

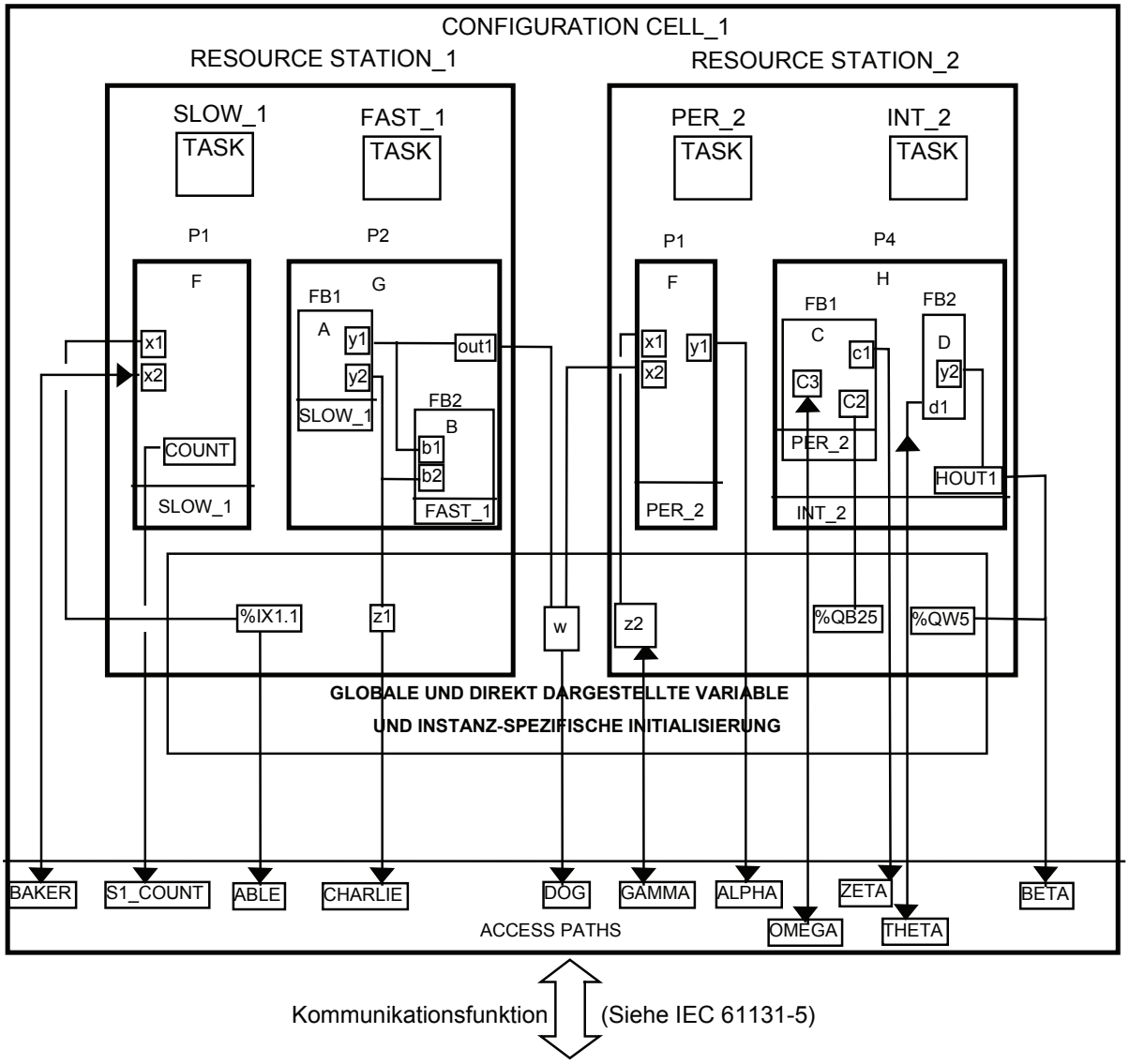


Bild 19 a) – Grafisches Beispiel für eine Konfiguration

```
FUNCTION_BLOCK A
  VAR_OUTPUT
    y1 : UINT ; y2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK B
  VAR_INPUT
    b1 : UINT ; b2 : BYTE ;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK C
  VAR_OUTPUT c1 : BOOL ; END_VAR
  VAR C2 AT %Q* : BYTE;
      C3 : INT;
  END_VAR
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK D
   VAR_INPUT d1 : BOOL ; END_VAR
   VAR_OUTPUT y2 : INT ; END_VAR
END_FUNCTION_BLOCK
```

```
PROGRAM F
   VAR_INPUT  x1 : BOOL ;  x2 : UINT ; END_VAR
   VAR_OUTPUT y1 : BYTE ; END_VAR
   VAR COUNT: INT; TIME1: TON; END_VAR
END_PROGRAM
```

```
PROGRAM G
  VAR_OUTPUT out1 : UINT ;  END_VAR
  VAR_EXTERNAL z1 : BYTE ;  END_VAR
  VAR  FB1 : A ;  FB2 : B ; END_VAR
  FB1(...);  out1 := FB1.y1;  z1 := FB1.y2;
  FB2(b1 := FB1.y1, b2 := FB1.y2) ;
END_PROGRAM
```

```
PROGRAM H
  VAR_OUTPUT HOUT1: INT ;  END_VAR
  VAR  FB1 : C ;  FB2 : D ; END_VAR
  FB1(...) ;
  FB2(...);  HOUT1 := FB2.y2;
END_PROGRAM
```

Bild 19 b) – Gerüst der Deklarationen von Funktionsbausteinen und Programmen als Beispiel für eine Konfiguration

2.7.1 Konfigurationen, Ressourcen und Zugriffspfade

Tabelle 49 zählt die Spracheigenschaften für die Deklaration von *Konfigurationen*, *Ressourcen*, *globalen Variablen* und *Zugriffspfaden* und Instanz-spezifischen Initialisierungen auf. Es ist auch eine teilweise Aufzählung der Eigenschaften der `TASK` Deklaration angegeben; weitere Informationen über *Tasks* sind in 2.7.2 zu finden. Die formale Syntax dieser Eigenschaften ist in B.1.7 angegeben. Bild 20 gibt Beispiele dieser Eigenschaften, entsprechend der in Bild 19 a) gezeigten Beispiel-Konfiguration und den dazugehörigen Deklarationen in Bild 19 b).

Das Bestimmungszeichen `ON` in der Konstruktion `RESOURCE...ON...END_RESOURCE` verwendet man zur Festlegung des Typs der „Verarbeitungsfunktion“, seiner Funktionen als „Mensch-Maschine-Schnittstelle“ und seiner Funktionen als „Sensor- und Aktor-Schnittstelle“, auf denen aufbauend die *Ressource* und ihre zugehörigen *Programme* und *Tasks* zu implementieren sind. Der Hersteller muss eine **implementierungsabhängige** *Ressource-Bibliothek* mit solchen Funktionen liefern, wie es in Bild 3 veranschaulicht ist. Jedem Element in dieser Bibliothek muss ein Bezeichner zugeordnet sein (der *Ressource-Typ-Name*), der in der Ressource-Deklaration benutzt wird.

ANMERKUNG Die Konstruktion `RESOURCE...ON...EN_RESOURCE` ist in einer Konfiguration mit einer einzelnen *Ressource* nicht erforderlich. Siehe dazu die Produktion `single_resource_declaration` in B.1.17 für die Syntax, die in diesem Fall zu verwenden ist.

Der *Geltungsbereich* einer `VAR_GLOBAL` Deklaration muss auf die *Konfiguration* oder *Ressource* begrenzt sein, in der sie deklariert ist, mit der Ausnahme, dass ein *Zugriffspfad* zu einer *globalen Variablen* in einer *Ressource* deklariert werden kann, indem man die Eigenschaft 10d in Tabelle 49 benutzt.

Die Konstruktion VAR_ACCESS...END_VAR bietet ein Mittel zur Festlegung von Variablen mit Namen, auf die durch die Kommunikationsdienste zugegriffen werden kann, die in IEC 61131-5 festgelegt sind. Ein *Zugriffspfad* verknüpft jede derartige Variable mit einer *globalen* Variablen, mit einer *direkt dargestellten* Variablen, wie in 2.4.1.1 definiert ist, oder mit einer *Eingangs-*, *Ausgangs-* oder internen Variablen eines *Programms* oder *Funktionsbausteins*.

Die Verknüpfung muss erreicht werden, indem man den Variablennamen durch die vollständige hierarchische Aneinanderreihung der Instanznamen festlegt; dies beginnt mit dem Namen der Ressource (sofern vorhanden), gefolgt vom Namen der Programm-Instanz (sofern vorhanden), gefolgt von den Namen der Funktionsbaustein-Instanzen (sofern vorhanden). Der Name der Variable wird am Ende der Kette angehängt. Alle Namen in der Aneinanderreihung müssen durch Punkte getrennt sein. Falls eine Variable eine *Multi-Element-Variable* (*Struktur* oder *Feld*) ist, kann ein Zugriffspfad auch zu einem Element der Variablen festgelegt werden.

Es darf nicht möglich sein, *Pfadnamen* zu Variablen zu definieren, die in VAR_TEMP, VAR_EXTERNAL oder VAR_IN_OUT Deklarationen deklariert sind.

Die Richtung des Zugriffspfads kann als READ_WRITE oder READ_ONLY definiert werden, um damit anzuzeigen, dass die Kommunikationsdienste im ersten Fall den Variablenwert sowohl lesen und verändern können oder im zweiten Fall den Wert lesen, aber nicht verändern können. Falls keine Richtung festgelegt ist, ist die voreingestellte Richtung READ_ONLY.

Der Zugriff auf Variablen, die als CONSTANT deklariert sind oder auf Funktionsbaustein-Eingänge, die außen mit anderen Variablen verbunden sind, muss READ_ONLY sein.

ANMERKUNG Die Wirkung der Anwendung der READ_WRITE Zugriffs auf Funktionsbaustein-Ausgangsvariable ist **implementierungsabhängig**.

Die VAR_CONFIG...END_VAR Konstruktion gibt die Möglichkeit, den symbolisch dargestellten Variablen Instanz-spezifische Ortsangaben zuzuweisen, die für den besonderen Zweck durch die Benutzung der Stern-Angabe bezeichnet werden, die in 2.4.1.1 und 2.4.3.1 beschrieben ist, oder den symbolisch dargestellten Variablen Instanz-spezifische Anfangswerte zuzuweisen, oder beides zu tun.

Die Verknüpfung muss durch die Angabe des Namen des Objektes geschehen, welches durch die vollständige hierarchische Aneinanderreihung der Instanznamen zugeordnet oder initialisiert werden muss, beginnend mit dem Namen der Ressource (sofern vorhanden), gefolgt vom Namen der Programm-Instanz (sofern vorhanden), gefolgt von den Namen der Funktionsbaustein-Instanzen (sofern vorhanden). Der Name des zuzuordnenden oder zu initialisierenden Objektes wird am Ende der Kette angehängt. Alle Namen in der Aneinanderreihung müssen durch Punkte getrennt sein. Die Positionszuordnung oder die Zuweisung eines Initialwertes folgt der Syntax und Semantik, wie in 2.4.3.1 und 2.4.3.2 beschrieben.

Durch VAR_CONFIG...END_VAR bereitgestellte, Instanz-spezifische Initialwerte überschreiben typspezifische Initialwerte immer. Es darf nicht möglich sein, Instanz-spezifische Initialisierungen von Variablen zu definieren, die durch VAR_TEMP, VAR_EXTERNAL, VAR CONSTANT oder VAR_IN_OUT Deklarationen deklariert wurden.

Tabelle 49 – Deklaration von Konfigurations- und Ressource-Eigenschaften

Nr.	Beschreibung
1	CONFIGURATION...END_CONFIGURATION Konstruktion
2	VAR_GLOBAL...END_VAR Konstruktion innerhalb CONFIGURATION
3	RESOURCE...ON...END_RESOURCE Konstruktion
4	VAR_GLOBAL...END_VAR Konstruktion innerhalb RESOURCE
5a	Periodische TASK Konstruktion (siehe Anmerkung 1)
5b	Nicht-periodische TASK Konstruktion (siehe Anmerkung 1)
6a	WITH Konstruktion für die Zuordnung von PROGRAM zu TASK (siehe Anmerkung 1)
6b	WITH Konstruktion für die Zuordnung von Funktionsbaustein zu TASK (siehe Anmerkung 1)
6c	PROGRAM Deklaration ohne TASK Zuordnung (siehe Anmerkung 1)
7	Deklaration von direkt dargestellten Variablen in VAR_GLOBAL (siehe Anmerkung 2)
8a	Verbindung von direkt dargestellten Variablen mit PROGRAM Eingängen
8b	Verbindung von GLOBAL Variablen mit PROGRAM Eingängen
9a	Verbindung von PROGRAM Ausgängen mit direkt dargestellten Variablen
9b	Verbindung von PROGRAM Ausgängen mit GLOBAL Variablen
10a	VAR_ACCESS...END_VAR Konstruktion
10b	Zugriffspfade zu direkt dargestellten Variablen
10c	Zugriffspfade zu PROGRAM Eingängen
10d	Zugriffspfade zu GLOBAL Variablen in RESOURCEs
10e	Zugriffspfade zu GLOBAL Variablen in CONFIGURATIONs
10f	Zugriffspfade zu PROGRAM Ausgängen
10g	Zugriffspfade zu PROGRAM internen Variablen
10h	Zugriffspfade zu Funktionsbaustein-Eingängen
10i	Zugriffspfade zu Funktionsbaustein-Ausgängen
11	VAR_CONFIG ... END_VAR Konstruktion [a]
12a	VAR_GLOBAL CONSTANT in RESOURCE Deklarationen [a]
12b	VAR_GLOBAL CONSTANT in CONFIGURATION Deklarationen
13a	VAR_EXTERNAL in RESOURCE Deklarationen
13b	VAR_EXTERNAL CONSTANT in RESOURCE Deklarationen

ANMERKUNG 1 Siehe 2.7.2 für weitere Beschreibungen der TASK Eigenschaften.

ANMERKUNG 2 Siehe 2.4.3.1 für weitere Beschreibungen der zugehörigen Eigenschaften.

[a] Diese Eigenschaft muss unterstützt werden, wenn die Eigenschaft 10 in Tabelle 15 unterstützt wird.

Nr.	Beispiel
1	CONFIGURATION CELL_1
2	VAR_GLOBAL w: UINT; END_VAR
3	RESOURCE STATION_1 ON PROCESSOR_TYPE_1
4	VAR_GLOBAL z1: BYTE; END_VAR
5a	TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2) ;
5a	TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
6a	PROGRAM P1 WITH SLOW_1 :
8a	F(x1 := %IX1.1) ;
9b	PROGRAM P2 : G(out1 => w,
6b	FB1 WITH SLOW_1,
6b	FB2 WITH FAST_1) ;
3	END_RESOURCE
3	RESOURCE STATION_2 ON PROCESSOR_TYPE_2
4	VAR_GLOBAL z2 : BOOL ;
7	AT %QW5 : INT ;
4	END_VAR
5a	TASK PER_2(INTERVAL := t#50ms, PRIORITY := 2) ;
5b	TASK INT_2(SINGLE := z2, PRIORITY := 1) ;
6a	PROGRAM P1 WITH PER_2 :
8b	F(x1 := z2, x2 := w) ;
6a	PROGRAM P4 WITH INT_2 :
9a	H(HOUT1 => %QW5,
6b	FB1 WITH PER_2);
3	END_RESOURCE
10a	VAR_ACCESS
10b	ABLE : STATION_1.%IX1.1 : BOOL READ_ONLY ;
10c	BAKER : STATION_1.P1.x2 : UINT READ_WRITE ;
10d	CHARLIE : STATION_1.z1 : BYTE ;
10e	DOG : w : UINT READ_ONLY ;
10f	ALPHA : STATION_2.P1.y1 : BYTE READ_ONLY ;
10f	BETA : STATION_2.P4.HOUT1 : INT READ_ONLY ;
10d	GAMMA : STATION_2.z2 : BOOL READ_WRITE ;
10g	S1_COUNT : STATION_1P1.COUNT : INT ;
10h	THETA : STATION_2.P4.FB2.d1 : BOOL READ_WRITE ;
10i	ZETA : STATION_2.P4.FB1.c1 : BOOL READ_ONLY ;
10k	OMEGA : STATION_2.P4.FB1.C3 : INT READ_WRITE ;
10a	END_VAR

Nr.	Beispiel
11	VAR_CONFIG STATION_1.P1.COUNT : INT := 1; STATION_2.P1.COUNT : INT := 100; STATION_1.P1.TIME1 : TON := (PT := T#2.5s); STATION_2.P1.TIME1 : TON := (PT := T#4.5s); STATION_2.P4.FB1.C2 AT %QB25 : BYTE; VAR_END
1	END_CONFIGURATION

ANMERKUNG 1 Die grafische und semigrafische Darstellung dieser Eigenschaften ist zulässig; sie liegt jedoch außerhalb des Geltungsbereichs dieses Teils der IEC 61131.

ANMERKUNG 2 Es ist ein **Fehler**, wenn der Datentyp, der für eine Variable in einer VAR_ACCESS Anweisung deklariert ist, nicht derselbe ist, wie der Datentyp der Variablen an anderer Stelle; z. B. wenn die Variable BAKER vom Typ WORD in obigen Beispielen deklariert wäre.

Bild 20 – Beispiele von Deklaration der Eigenschaften von CONFIGURATION und RESOURCE

2.7.2 Tasks

Für diesen Teil der IEC 61131-3 ist eine *Task* als ein Element der Ausführungssteuerung definiert, das imstande ist, entweder auf einer periodischen Basis oder auf Grund des Auftretens einer steigenden Flanke einer bestimmten boolesche Variablen die Ausführung einer Menge von Programm-Organisationseinheiten aufzurufen, die *Programme* und *Funktionsbausteine* einschließen können, deren Instanzen in der Deklaration von *Programmen* festgelegt sind.

Die maximale Anzahl von Tasks pro *Ressource* und die Auflösung des Task-Intervalls sind **implementierungsanhängige Parameter**.

Tasks und ihre Verknüpfung mit Programm-Organisationseinheiten können grafisch oder in Textform mit Hilfe der WITH Konstruktion, wie in Tabelle 50 gezeigt ist, als ein Teil von *Ressourcen* innerhalb von *Konfigurationen* dargestellt werden. Eine Task wird implizit durch ihre zugehörige Ressource nach den Mechanismen, die in 1.4.1 definiert sind, freigegeben oder nicht-freigegeben. Die Steuerung der Programm-Organisationseinheiten durch freigegebene Tasks muss den folgenden Regeln entsprechen:

1) Die zugehörigen Programm-Organisationseinheiten müssen bei jeder steigenden Flanke am Task-Eingang SINGLE zur Ausführung aufgerufen werden.

2) Falls der Eingang INTERVAL ungleich null ist, müssen die zugehörigen Programm-Organisationseinheiten zur periodischen Ausführung im festgelegten Intervall aufgerufen werden, solange der Eingang SINGLE auf null (0) steht. Falls der Eingang INTERVAL gleich null ist (der Voreinstellungswert), darf kein periodisches Aufrufen der zugehörigen Programm-Organisationseinheiten erfolgen.

3) Der Eingang PRIORITY einer Task legt die Aufrufpriorität der zugehörigen Programm-Organisationseinheiten fest, wobei null (0) die höchste Priorität ist und die aufeinander folgenden niederen Prioritäten aufeinanderfolgende höhere numerische Werte haben. Wie in Tabelle 50 gezeigt, kann die Priorität einer Programm-Organisationseinheit (d. h. die Priorität ihrer zugehörigen Task) zum *bevorrechtigten* (preemptive) oder *nicht-bevorrechtigten* (non-preemptive scheduling) Aufrufen benutzt werden.

   a) Beim *nicht-bevorrechtigten* Aufrufen (non-preemptive scheduling) wird einer *Ressource* Rechenzeit zur Verfügung gestellt, sobald die Ausführung einer Programm-Organisationseinheit oder einer Betriebssystem-Funktion vollendet ist. Wenn Rechenzeit verfügbar ist, muss die Programm-Organisationseinheit mit der höchsten zugeteilten Priorität zur Ausführung kommen. Falls mehr als eine Programm-Organisationseinheit mit der höchsten zugeteilten Priorität wartend ist, muss die Programm-Organisationseinheit mit der längsten Wartezeit in der höchsten Priorität ausgeführt werden.

   b) Beim *vorberechtigten* Aufrufen kann eine auszuführende Programm-Organisationseinheit die Ausführung einer Programm-Organisationseinheit mit niederer Priorität in der gleichen *Ressource unterbrechen,* d. h. die Ausführung einer niederprioren Einheit kann aufgeschoben werden, bis die

Ausführung der höherprioren Einheit vollendet ist. Eine Programm-Organisationseinheit darf die Ausführung einer anderen Einheit nicht unterbrechen, die gleiche oder höhere Priorität besitzt.

ANMERKUNG Abhängig von den Aufruf-Prioritäten könnte eine Programm-Organisationseinheit die Ausführung nicht zu dem Zeitpunkt beginnen, an dem sie aufgerufen wird. In den Beispielen, die in Tabelle 50 gezeigt sind, liegen jedoch alle Programm-Organisationseinheiten innerhalb ihrer *Zeitfristen*, d. h. sie beenden alle die Ausführung, bevor sie erneut zur Ausführung aufgerufen werden. Damit der Anwender bestimmen kann, ob alle Zeitfristen in einer gegebenen Konfiguration eingehalten werden, muss der Hersteller entsprechende Informationen bereitstellen.

4) Ein *Programm* ohne Task-Verknüpfung muss die niederste Priorität haben. Jedes derartige Programm muss mit dem „Starten“ seiner *Ressource* zur Ausführung aufgerufen werden, wie in 1.4.1 definiert ist, und muss erneut zur Ausführung aufgerufen werden, sobald seine Ausführung beendet ist.

5) Wenn eine *Funktionsbaustein-Instanz* mit einer Task verknüpft ist, muss ihre Ausführung durch die ausschließliche Kontrolle der Task erfolgen, unabhängig von den Regeln zur Auswertung der Programm-Organisationseinheit, in der die Task-zugehörige Funktionsbaustein-Instanz deklariert ist.

6) Die Ausführung einer *Funktionsbaustein-Instanz*, die nicht direkt mit einer Task verknüpft ist, muss den normalen Regeln über die Reihenfolge der Auswertung von Sprachelementen für die Programm-Organisationseinheit folgen (die selbst unter der Steuerung einer Task stehen kann), in der die Funktionsbaustein-Instanz deklariert ist.

7) Die Ausführung von Funktionsbausteinen innerhalb eines Programms muss synchronisiert werden, um sicherzustellen, dass die Datenkonsistenz nach den folgenden Regeln gewährleistet ist:

   a) Wenn ein Funktionsbaustein mehr als eine Eingabe von einem anderen Funktionsbaustein empfängt, dann müssen alle Eingänge vom letzteren die Ergebnisse derselben Auswertung darstellen, wenn der erstere ausgeführt wird. Z. B. müssen in dem Beispiel, das in Bild 21 a) dargestellt ist, wenn `Y2` ausgewertet wird, die Eingänge `Y2.A` und `Y2.B` gleich den Ausgängen `Y1.C` und `Y1.D` von derselben (nicht zwei unterschiedlichen) Auswertung von `Y1` sein.

   b) Wenn zwei oder mehr Funktionsbausteine Eingaben vom gleichen Funktionsbaustein empfangen und wenn die „Empfänger“-Bausteine alle explizit oder implizit mit der selben Task verknüpft sind, dann müssen die Eingänge von allen diesen „Empfänger“-Bausteinen zum Zeitpunkt ihrer Auswertung die Ergebnisse der selben Auswertung des „Erzeuger“-Bausteins darstellen. Z. B. müssen in den Beispielen, die durch die Bilder 21 b) und 21 c) dargestellt sind, wenn `Y2` und `Y3` in einem normalen Lauf des Programms `P1` ausgewertet werden, die Eingänge `Y2.A` und `Y2.B` die Ergebnisse der gleichen Auswertung von `Y1` darstellen wie die Eingänge `Y3.A` und `Y3.B`.

Es muss eine Speicherung für die Ausgänge von Funktionen und Funktionsbausteinen gewährleistet sein, die explizite Task-Verknüpfungen haben oder die als Eingänge von Programm-Organisationseinheiten benutzt werden, die explizite Task-Verknüpfungen haben, wie es für die Erfüllung der oben gegebenen Regeln notwendig ist.

Es muss eine **Fehler** im Sinne des Unterabschnitts 1.5.1 sein, falls eine Task wegen überhöhter Ressourcen-Anforderungen oder anderer Task-Aufrufkonflikte nicht aufgerufen wird oder ihre Zeitfrist für die Ausführung nicht einhält.

Tabelle 50 – Task-Eigenschaften

<table>
<tr><th>Nr.</th><th>Beschreibung/Beispiele</th></tr>
<tr><td>1a</td><td>Text-Deklaration von periodischer <code>TASK</code> (Eigenschaft 5a von Tabelle 49)</td></tr>
<tr><td>1b</td><td>Text-Deklaration von nicht-periodischer <code>TASK</code> (Eigenschaft 5b von Tabelle 49)</td></tr>
<tr><td colspan="2">Grafische Darstellung von <code>TASKs</code> (allgemeine Form)</td></tr>
<tr><td colspan="2"><pre>
                    TASKNAME
                      +---------+
                      | TASK |
                BOOL---|SINGLE |
                TIME---|INTERVAL |
                UINT---|PRIORITY |
                      +---------+
</pre></td></tr>
<tr><td rowspan="2">2a</td><td>Grafische Darstellung von periodischen <code>TASKs</code></td></tr>
<tr><td><pre>
          SLOW_1   FAST_1
        +---------+   +---------+
        | TASK | | TASK |
     ---|SINGLE | ---|SINGLE |
t#20ms---|INTERVAL | t#10ms---|INTERVAL |
    2---|PRIORITY | 1---|PRIORITY |
        +---------+   +---------+
</pre></td></tr>
<tr><td rowspan="2">2b</td><td>Grafische Darstellung von nicht-periodischer <code>TASK</code></td></tr>
<tr><td><pre>
                      INT_2
                     +---------+
                     | TASK |
                  --|SINGLE |
                %IX--|INTERVAL |
                  1--|PRIORITY |
                     +---------+
</pre></td></tr>
<tr><td>3a</td><td>Textuelle Zuordnung zu <code>PROGRAMs</code> (Eigenschaft 6a der Tabelle 49)</td></tr>
<tr><td>3b</td><td>Textuelle Zuordnung zu Funktionsbausteinen (Eigenschaft 6b der Tabelle 49)</td></tr>
<tr><td rowspan="2">4a</td><td>Grafische Zuordnung zu <code>PROGRAMs</code></td></tr>
<tr><td><pre>
RESOURCE STATION_2

    P1   P4
+-------+   +-------+
F		H
+-------+   +-------+
| PER_2 | | INT_2 |
+-------+   +-------+

END_RESOURCE
</pre></td></tr>
</table>

Nr. 4b — Grafische Zuordnung zu Funktionsbausteinen innerhalb PROGRAMs

```
RESOURCE STATION_1
   P2
   +-----------------------------------------------------+
   |                           G                         |
   |                                                     |
   |              FB1                  FB2               |
   |           +------+             +------+             |
   |           |   A  |             |   B  |             |
   |           |      |             |      |             |
   |           |      |             |      |             |
   |           +------+             +------+             |
   |           |SLOW_1|             |FAST_1|             |
   |           +------+             +------+             |
   +-----------------------------------------------------+
END_RESOURCE
```

Nr. 5a — Nicht-bevorrechtigter Zeitplan (non-preemptive scheduling)

BEISPIEL 1:

– `RESOURCE STATION_1` wie in Bild 20 konfiguriert

– Ausführungszeiten: `P1` = 2 ms; `P2` = 8 ms;

– `P2.FB1 = P2.FB2` = 2 ms (siehe Anmerkung 3)

– `STATION_1` startet bei t = 0

ZEITPLAN (wiederholt sich alle 40 ms)

t (ms)	In Ausführung	Wartend
0	P2.FB2@1	P1@2, P2.FB1@2, P2
2	P1@2	P2.FB1@2, P2
4	P2.FB1@2	P2
6	P2	
10	P2	P2.FB2@1
14	P2.FB2@1	P2
16	P2	(Neustart von P2)
20	P2	P2.FB2@1, P1@2, P2.FB1@2
24	P2.FB2@1	P1@2, P2.FB@2, P2
26	P1@2	P2.FB1@2, P2
28	P2.FB1@2	P2
30	P2.FB2@1	P2
32	P2	
40	P2.FB2@1	P1@2, P2.FB1@2, P2

Nr.	Beschreibung/Beispiele		
	BEISPIEL 2:		
	– RESOURCE STATION_2 wie in Bild 20 konfiguriert		
	– Ausführungszeiten: P1 = 30 ms; P4 = 5 ms; P4.FB1 = 10 ms (siehe Anmerkung 4)		
	– INT_2 wird ausgelöst bei t = 25, 50, 90... ms		
	– STATION_2 startet bei t = 0		
	ZEITPLAN		
	t (ms)	**In Ausführung**	**Wartend**
	0	P1@2	P4.FB1@2
	25	P1@2	P4.FB1@2, P4@1
	30	P4@1	P4.FB1@2
	35	P4.FB1@2	
	50	P4@1	P1@2, P4.FB1@2
	55	P1@2	P4.FB1@2
	85	P4.FB1@2	
	90	P4.FB1@2	P4@1
	95	P4@1	
	100	P1@2	P4.FB1@2
5b	**Bevorrechtigter Zeitplan (preemptive scheduling)**		
	BEISPIEL 3:		
	– RESOURCE STATION_1 wie in Bild 20 konfiguriert		
	– Ausführungszeiten: P1 = 2 ms; P2 = 8 ms, P2.FB1 = P2.FB2 = 2 ms (siehe Anmerkung 3)		
	– STATION_1 startet bei t = 0		
	ZEITPLAN		
	t (ms)	**In Ausführung**	**Wartend**
	0	P2.FB2@1	P1@2, P2.FB1@2, P2
	2	P1@2	P2.FB1@2, P2
	4	P2.FB1@2	P2
	6	P2	
	10	P2.FB2@1	P2
	12	P2	
	16	P2	P2 Neustart)
	20	P2.FB2@1	P1@2, P2.FB1@2, P2
	BEISPIEL 4:		
	– RESOURCE STATION_2 wie in Bild 20 konfiguriert		
	– Ausführungszeiten: P1 = 30 ms; P4 = 5 ms; P4.FB1 = 10 ms (siehe Anmerkung 4)		
	– INT_2 ist ausgelöst bei = 25, 50, 90... ms		
	– STATION_2 startet bei t = 0		

Nr.	Beschreibung/Beispiele		
	ZEITPLAN		
	t (ms)	**In Ausführung**	**Wartend**
	0	P1@2	P4.FB1@2
	25	P4@1	P1@2, P4.FB1@2
	30	P1@2	P4.FB1@2
	35	P4.FB1@2	
	50	P4@1	P1@2, P4.FB1@2
	55	P1@2	P4.FB1@2
	85	P4.FB1@2	
	90	P4@1	P4.FB1@2
	95	P4.FB1@2	
	100	P1@2	P4.FB1@2

ANMERKUNG 1 Einzelheiten der Deklarationen von RESOURCE und PROGRAM sind nicht gezeigt; siehe 2.7 und 2.7.1.

ANMERKUNG 2 Die Schreibweise X@Y zeigt an, dass die Programm-Organisationseinheit X mit der Priorität Y aufgerufen ist oder ausgeführt wird.

ANMERKUNG 3 Die Ausführungszeiten von P2.FB1 und P2.FB2 sind nicht in der Ausführungszeit von P2 eingeschlossen.

ANMERKUNG 4 Die Ausführungszeit von P4.FB1 ist nicht in der Ausführungszeit von P4 eingeschlossen.

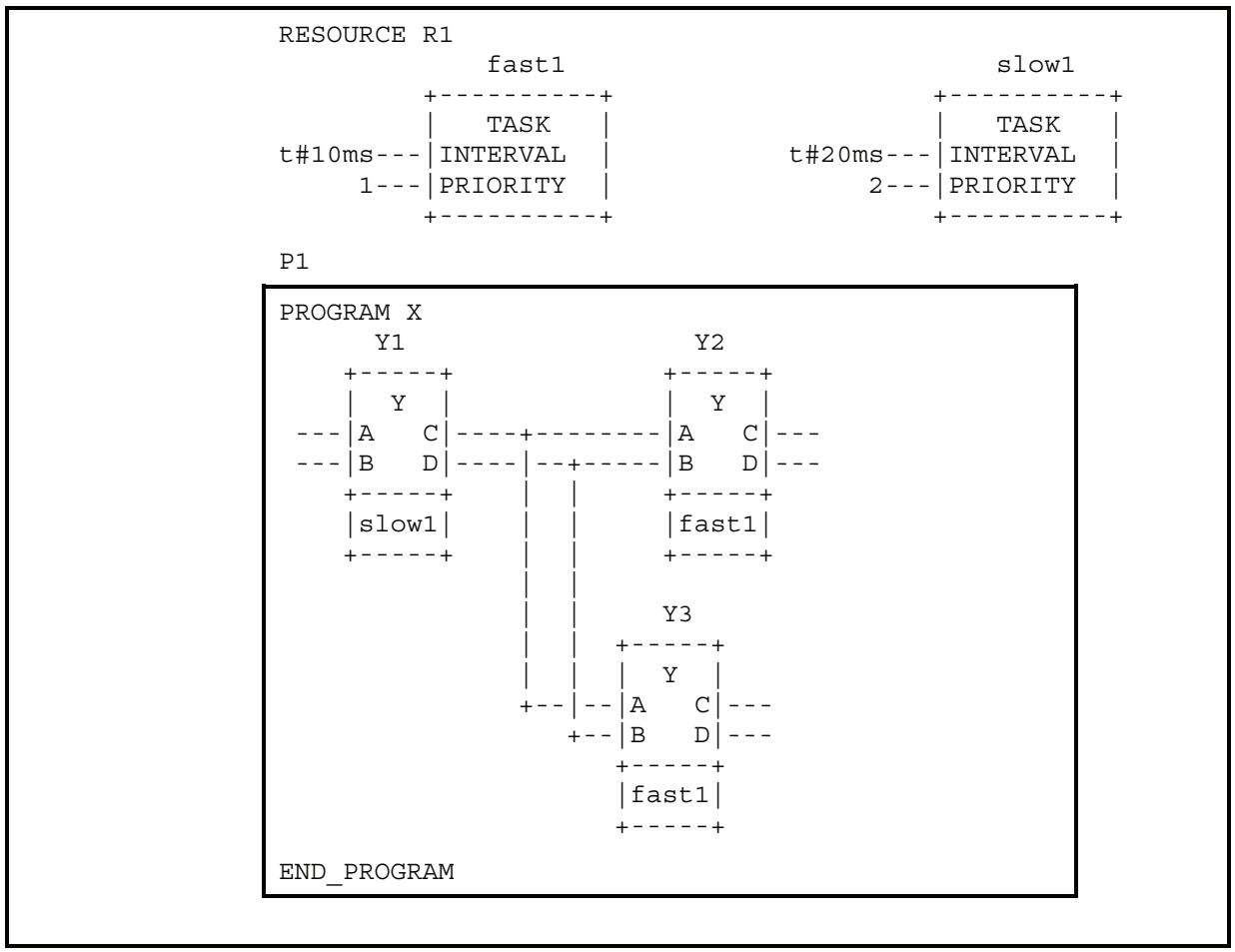


Bild 21 a) – Synchronisation von Funktionsbausteinen bei expliziten Task-Zuordnungen

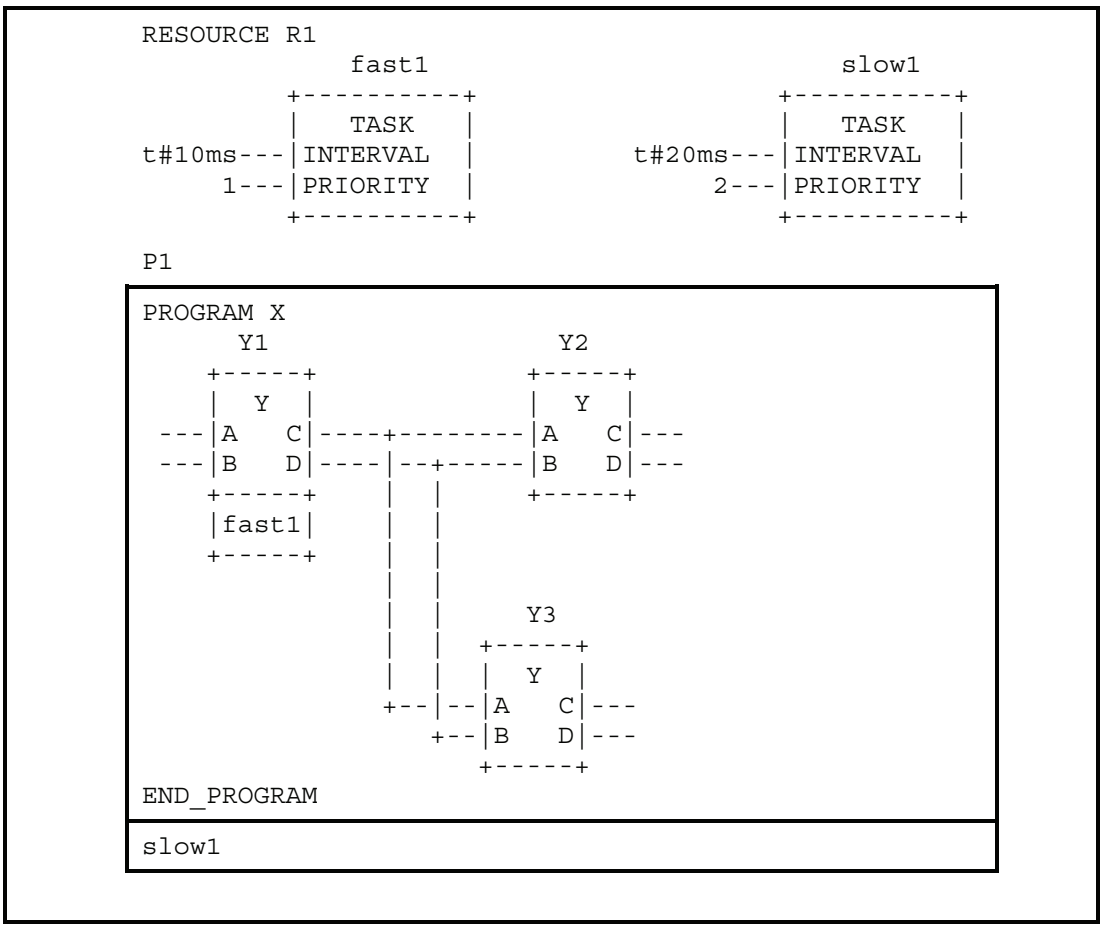


Bild 21 b) – Synchronisation von Funktionsbausteinen bei impliziten Task-Zuordnungen

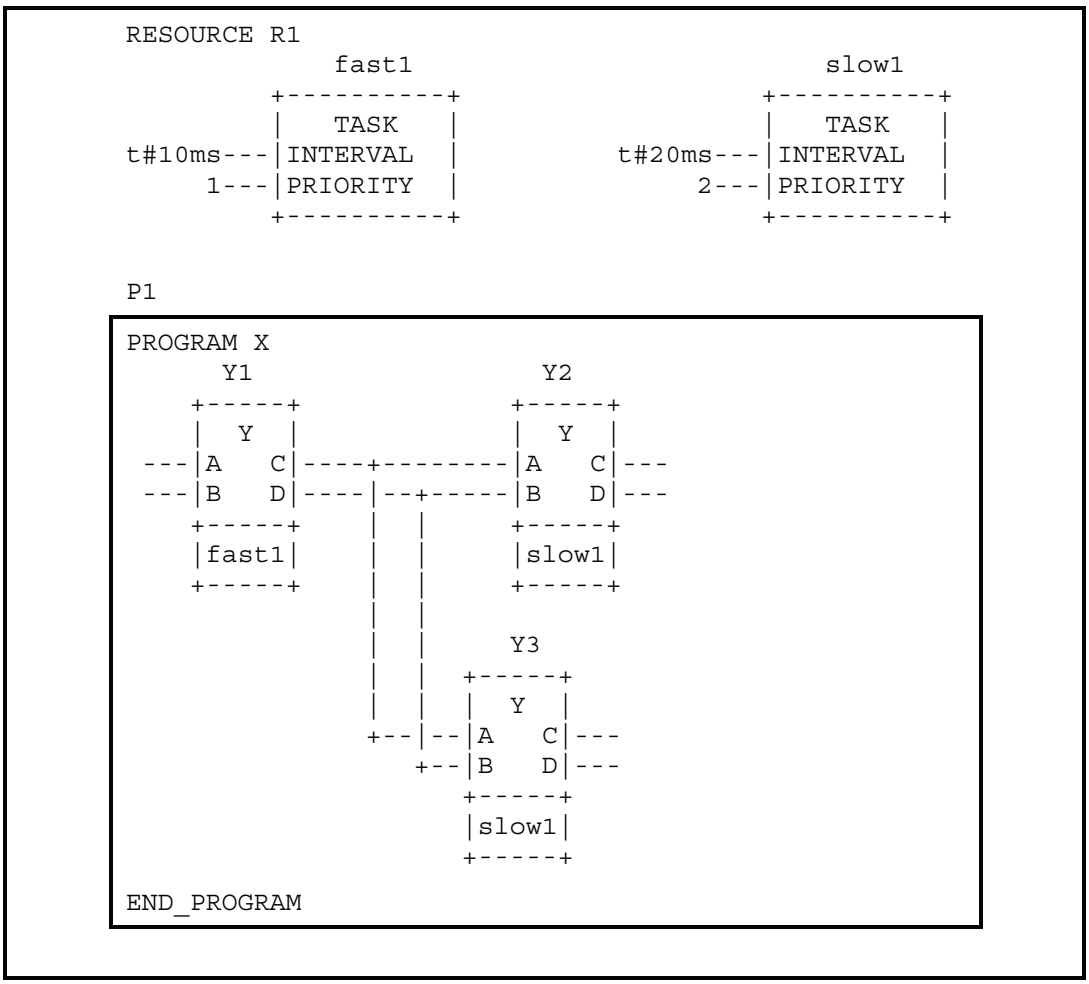


Bild 21 c) – Explizite Task-Zuordnungen äquivalent mit Bild 21 b)

3 Textsprachen

Die Textsprachen, die in dieser Norm definiert werden, sind AWL (Anweisungsliste) und ST (Strukturierter Text). Die Elemente der Ablaufsprache (AS), die in 2.6 definiert sind, können in Verbindung mit jeder dieser Sprachen benutzt werden.

3.1 Gemeinsame Elemente

Die Textelemente, die in Abschnitt 2 festgelegt sind, müssen in den Textsprachen (AWL und ST), die in diesem Abschnitt definiert sind, gemeinsam gelten. Im Einzelnen müssen die folgenden Elemente zur Programm-Strukturierung für die Textsprachen gemeinsam gelten:

TYPE...END_TYPE	(2.3.3)
VAR...END_VAR	(2.4.3)
VAR_INPUT...END_VAR	(2.4.3)
VAR_OUTPUT...END_VAR	(2.4.3)
VAR_IN_OUT...END_VAR	(2.4.3)
VAR_EXTERNAL...END_VAR	(2.4.3)
VAR_TEMP...END_VAR	(2.4.3)
VAR_ACCESS...END_VAR	(2.4.3)
VAR_GLOBAL...END_VAR	(2.4.3)
VAR_CONFIG...END_VAR	(2.4.3)
FUNCTION...END_FUNCTION	(2.5.1.3)
FUNCTION_BLOCK...END_FUNCTION_BLOCK	(2.5.2.2)
PROGRAM...END_PROGRAM	(2.5.3)
STEP...END_STEP	(2.6.2)
TRANSITION...END_TRANSITION	(2.6.3)
ACTION...END_ACTION	(2.6.4)

3.2 Anweisungsliste (AWL)

Dieser Abschnitt definiert die Semantik der Sprache AWL (Anweisungsliste), deren formale Syntax in B.2 angegeben ist.

3.2.1 Anweisungen

Wie in Tabelle 51 veranschaulicht, setzt sich eine *Anweisungsliste* aus einer Folge von *Anweisungen* zusammen. Jede Anweisung muss in einer neuen Zeile beginnen und muss einen *Operator* mit optionalen *Modifizierern* und, falls erforderlich für die jeweilige Operation, einen oder mehrere *Operanden* enthalten, die durch Kommas getrennt sind. Operanden können in allen Datendarstellungen auftreten, die in 2.2 für Literale, in 2.3.3 für aufgezählte Werte und in 2.4 für Variablen definiert sind.

Der Anweisung kann eine identifizierende *Marke* vorangehen, der ein Doppelpunkt (`:`) folgt. Leere Zeilen können zwischen den Anweisungen eingefügt werden.

Tabelle 51 a) – Beispiele für Anweisungsfelder

Marke	Operator	Operand	Kommentar
START:	LD	%IX1	(* Druck-Taster *)
	ANDN	%MX5	(* Nicht-gesperrt *)
	ST	%QX2	(* Lüfter an *)

3.2.2 Operatoren, Modifizierer und Operanden

Standardoperatoren mit ihren zulässigen Modifizierern und Operanden müssen dargestellt werden, wie es in Tabelle 52 aufgezählt ist. Die Typangabe der Operatoren muss den Konventionen von 2.5.1.4 folgen.

Falls nicht anders in Tabelle 52 definiert, muss die Semantik der Operatoren folgendermaßen sein

Ergebnis := Ergebnis OP Operand

Das heißt, der Wert des Ausdrucks, der ausgerechnet wird, wird durch den aktuellen Wert ersetzt, der auf Grund des Operators mit dem Operand berechnet wurde. Z. B. ist die Anweisung `AND %IX2` interpretiert als

Ergebnis := Ergebnis AND %IX2

Die Vergleichsoperatoren müssen mit dem aktuellen Ergebnis auf der linken Seite des Vergleichs und dem Operanden auf der rechten Seite ausgeführt werden und ein boolesches Ergebnis liefern. Z. B. wird die Anweisung „`GT %IW10`“ das boolesche Ergebnis 1 haben, falls das aktuelle Ergebnis größer ist als der Wert des Eingangswortes 10, und andernfalls das boolesche Ergebnis 0.

Der Modifizierer „N“ zeigt die boolesche Negation des Operanden an. Z. B. ist die Anweisung `ANDN%IX2` zu verstehen als

Ergebnis := Ergebnis AND NOT %IX2

Es muss ein **Fehler** im Sinne des Abschnitts 1.5.1 sein, falls das aktuelle Ergebnis und der Operand nicht denselben Datentyp besitzen oder falls das Ergebnis einer numerischen Operation den Wertebereich seines Datentyps überschreitet.

Der Modifizierer linke Klammer „(“ zeigt an, dass die Auswertung des Operators zurückgestellt werden muss, bis ein Operator rechte Klammer „)“ erscheint. In Tabelle 51 b) sind zwei gleichwertige Formen einer geklammerten Folge von Anweisungen gezeigt. Beide Eigenschaften in Tabelle 51 b) müssen interpretiert werden wie

Ergebnis := Ergebnis AND (%IX1 OR %IX2)

Tabelle 51 b) – Eigenschaften des geklammerter Ausdrucks in Anweisungsliste

Nr.	BESCHREIBUNG/BEISPIEL
1	**Geklammerter Ausdruck, der explizit mit einen Operator beginnt:**

	`AND(`
	`LD  %IX1  (ANMERKUNG 1)`
	`OR  %IX2`
	`)`

2	**Geklammerter Ausdruck (Kurzform):**

	`AND(  %IX1`
	`OR  %IX2`
	`)`

ANMERKUNG In der Form 1 darf der `LD` Operator modifiziert werden oder der `LD` Operator darf durch einen anderen Operator oder Funktionsaufruf ersetzt werden.	

Der Modifizierer „C“ zeigt an, dass die zugehörige Anweisung nur durchgeführt werden darf, falls der Wert des gerade ausgewerteten Ergebnisses eine boolesche 1 ist (oder eine boolesche 0, falls der Operator mit einem Modifizierer „N“ verknüpft ist).

Tabelle 52 – Operatoren der Anweisungsliste (AWL)

Nr.	OPERATOR [a]	MODIFIZIERER (Anmerkung 1)	BEDEUTUNG
1	`LD`	`N`	Setzt aktuelles Ergebnis dem Operanden gleich
2	`ST`	`N`	Speichert aktuelles Ergebnis auf die Operanden-Adresse
3	`S`[e] `R`[e]		Setzt booleschen Operator auf 1, falls das aktuelle Ergebnis boolesche 1 ist Setzt booleschen Operator auf 0 zurück, falls das aktuelle Ergebnis boolesche 1 ist
4	`AND`	`N, (`	Logisches UND

Nr.	OPERATOR [a]	MODIFIZIERER (Anmerkung 1)	BEDEUTUNG
5	`&`	`N, (`	Logisches UND
6	`OR`	`N, (`	Logisches ODER
7	`XOR`	`N, (`	Logisches Exklusiv-ODER
7a	`NOT`[d]		Logische Negation (Einer-Komplement)
8	`ADD`	`(`	Addition
9	`SUB`	`(`	Subtraktion
10	`MUL`	`(`	Multiplikation
11	`DIV`	`(`	Division
11a	`MOD`	`(`	Modulo-Division
12	`GT`	`(`	Vergleich: >
13	`GE`	`(`	Vergleich: >=
14	`EQ`	`(`	Vergleich: =
15	`NE`	`(`	Vergleich: <>
16	`LE`	`(`	Vergleich: <=
17	`LT`	`(`	Vergleich: <
18	`JMP`[b]	`C, N`	Sprung zur Marke
19	`CAL`[c]	`C, N`	Aufruf Funktionsbaustein (siehe Tabelle 53)
20	`RET`[f]	`C, N`	Rücksprung von aufgerufener Funktion, Funktionsbaustein oder Programm
21	`)`[f]		Bearbeitung zurückgestellter Operation

ANMERKUNG Siehe vorausgehenden Text zur Erklärung von Modifizierern und Auswertung von Ausdrücken.

[a] Falls nicht anders angegeben, müssen diese Operatoren entweder überladen oder typisiert sein, wie es in 2.5.1.4 und 2.5.1.5.6 definiert ist.

[b] Der Operand einer `JMP` Anweisung muss die Marke einer Anweisung sein, zu der die Ausführung übergehen soll. Wenn eine Sprung-Anweisung in einem `ACTION...END_ACTION` Konstrukt enthalten ist, muss der Operand eine Marke in demselben Konstrukt sein.

[c] Der Operand dieser Anweisung muss der Name einer Funktionsbaustein-*Instanz* sein, die *aufgerufen* werden soll.

[d] Das Ergebnis dieser Anweisung muss eine Bit-weise Negation (Einer-Komplement) des aktuellen Ergebnisses sein.

[e] Der Typ des Operanden dieser Anweisung muss `BOOL` sein.

[f] Diese Anweisung darf keinen Operanden haben.

3.2.3 Funktionen und Funktionsbausteine

Funktionen, wie in 2.5.1 definiert, müssen aufgerufen werden, indem der Funktionsname in das Operatorfeld eingetragen wird. Wie in den Eigenschaften 4 und 5 von Tabelle 53 gezeigt, kann dieser Aufruf eine von zwei Formen annehmen. Der Wert, der von einer Funktion auf Grund einer erfolgreichen Ausführung einer `RET` Anweisung oder bei Erreichen des physikalischen Endes der Funktion zurückgegeben wird, muss das „aktuelle Ergebnis“ werden, wie es in 3.2.2 beschrieben ist.

Die Argumenten-Liste von Funktionen (Eigenschaft 4 von Tabelle 53) ist gleichbedeutend mit Eigenschaft 1 von Tabelle 19 a). Es gelten die Regeln und Eigenschaften, die in 2.5.1.1 und Tabelle 19 a) für Funktionsaufrufe definiert sind.

Eine nicht-formale Eingangsliste von Funktionen (Eigenschaft 5 von Tabelle 53) ist gleichbedeutend mit Eigenschaft 2 von Tabelle 19 a). Es gelten die Regeln und Eigenschaften, die in 2.5.1.1 und Tabelle 19 a) für Funktionsaufrufe definiert sind. Im Gegensatz zu den Beispielen, die in Tabelle 19 a) für die ST-Sprache angegeben sind, ist das erste Argument in der nicht-formalen Eingangsliste in AWL nicht enthalten, es muss jedoch das aktuelle Ergebnis als erstes Argument der Funktion verwendet werden. Gegebenenfalls müssen

zusätzliche Argumente (beginnend mit dem zweiten) im Operanden-Feld getrennt durch Kommas und in der Reihenfolge ihrer Deklaration angegeben werden.

Funktionsbausteine, wie in 2.5.2 definiert, können bedingt oder unbedingt durch den Operator CAL (Aufruf) aufgerufen werden, der in Tabelle 52 angegeben ist. Wie in den Eigenschaften 1a, 1b, 2 und 3 von Tabelle 53 gezeigt, kann dieser Aufruf eine von vier Formen annehmen.

Eine formale Argumenten-Liste eines Funktionsbaustein-Aufrufs (Eigenschaft 1a in Tabelle 53) ist gleichbedeutend mit Eigenschaft 1 in Tabelle 19 a). Eine nicht-formale Argumenten-Liste eines Funktionsbaustein-Aufrufs (Eigenschaften 1b in Tabelle 53) ist gleichbedeutend mit Eigenschaft 2 in Tabelle 19 a). Die Regeln und Eigenschafen, die in 2.5.1.1 und Tabelle 19 a) für Funktionsaufrufe definiert sind, gelten entsprechend, wenn man jeweils den Begriff „Funktion“ durch den Begriff „Funktionsbaustein“ in diesen Regeln ersetzt.

Alle Zuweisungen in einer Argumenten-Liste eines bedingten Funktionsbaustein-Aufrufs dürfen nur zusammen mit dem Aufruf ausgeführt werden, wenn die Bedingung erfüllt ist.

Tabelle 53 – Eigenschaften des Funktionsbaustein-Aufrufs und Funktionsaufrufs in der Sprache AWL

Nr.	BESCHREIBUNG/BEISPIEL
1a	Aufruf (CAL) von Funktionsbausteinen mit nicht-formaler Argumenten-Liste:
	`CAL C10(%IX10, FALSE, A, OUT, B)` `CAL CMD_TMR(%IX5, T#300ms, OUT, ELAPSED)`
1b	Aufruf (CAL) von Funktionsbausteinen mit formaler Argumenten-Liste:

	`CAL C10(`
	`  CU := %IX10,`
	`  Q => OUT)`
	`CAL CMD_TMR(`
	`  IN := %IX5,`
	`  PT := T#300ms,`
	`  Q => OUT,`
	`  ET => ELAPSED,`
	`  ENO => ERR)`

2	Aufruf (CAL) von einem Funktionsbaustein mit Laden/Speichern von Argumenten (Anmerkung 2):

	`LD A`
	`ADD 5`
	`ST C10.PV`
	`LD %IX10`
	`ST C10.CU`
	`CAL C10`

3	Gebrauch von Funktionsbaustein-Eingangsoperatoren:

	`LD A`
	`ADD 5`
	`PV C10`
	`LD %IX10`
	`CU C10`

4	Funktionsaufruf mit formaler Argumenten-Liste:

Nr.	BESCHREIBUNG/BEISPIEL

	`LIMIT(`
	` EN:=  COND,`
	` IN:=  B,`
	` MN:=  1,`
	` MX:=  5,`
	` ENO =>  TEMPL`
	`)`
	`ST  A`

5	Funktionsaufruf mit nicht-formaler Argumenten-Liste:

	`LD  1`
	`LIMIT  B, 5`
	`ST  A`

ANMERKUNG 1 Eine Deklaration wie

```
VAR
  C10 : CTU;
  CMD_TMR : TON;
  A, B : INT;
  ELAPSED : TIME;
  OUT, ERR, TEMPL, COND : BOOL;
END_VAR
```

ist in den obigen Beispielen vorausgesetzt.

ANMERKUNG 2 Dieser Gebrauch ist eine Ausnahme von der Regel, die in 2.5.2.1 angegeben ist: „Die Zuweisung eines Wertes an die Eingänge eines Funktionsbausteins ist nur zulässig als Teil des Aufrufs eines Funktionsbausteins."

Die Eingangsoperatoren, die in Tabelle 54 gezeigt werden, können in Verbindung mit der Eigenschaft 3 von Tabelle 53 verwendet werden. Diese Aufruf-Methode ist gleichbedeutend mit einem Aufruf (`CAL`) mit Argumenten-Liste, die nur eine Variable mit dem Namen des Eingangsoperators enthält. Argumente, die nicht zur Verfügung gestellt werden, werden von der letzten Zuweisung genommen oder wenn diese nicht vorhanden ist, von der Initialisierung. Diese Eigenschaft unterstützt Problemfälle, bei denen Ereignisse vorhersagbar sind und daher sich nur eine Variable von einen Aufruf zum nächsten ändern kann.

BEISPIEL 1:

Zusammen mit der Deklaration

```
VAR C10: CTU;
```

liefert die Anweisungsfolge

```
LD 15
PV C10
```

dasselbe Ergebnis wie

```
CAL C10 (PV:=15)
```

Die fehlenden Eingänge `R` und `CU` haben Werte, die ihnen vorher zugewiesen wurden. Da der Eingang `CU` eine steigende Flanke erkennt, wird nur der `PV` Eingang durch diesen Aufruf gesetzt werden; es kann kein Zählen erfolgen, weil ein nicht-geliefertes Argument sich nicht ändern kann. Im Gegensatz dazu bewirkt die Folge

```
LD %IX10
CU C10
```

das Zählen bis zum Maximum in jedem zweiten Aufruf, abhängig von der Änderungsrate des Eingangs `%IX10`. Jeder Aufruf verwendet die vorher gesetzten Werte von `PV` und `R`.

BEISPIEL 2

Bei den bistabilen Funktionsbausteinen mit einer Deklaration

```
VAR FORWARD: SR; END_VAR
```

bewirkt dies ein implizites bedingtes Verhalten. Die Folge

```
LD FALSE
S1 FORWARD
```

ändert nicht den Zustand des bistabilen Blocks `FORWARD`. Eine weitere Folge

```
LD TRUE
R FORWARD
```

setzt den bistabilen Block zurück.

Tabelle 54 – Eingangsoperatoren von Standard-Funktionsbausteinen für die Sprache AWL

Nr.	Operatoren	FB Typ	Verweis
4	`S1,R`	`SR`	2.5.2.3.1
5	`S,R1`	`RS`	2.5.2.3.1
6	`CLK`	`TRIGGER`	2.5.2.3.2
8	`CU,R,PV`	`CTU`	2.5.2.3.3
9	`CD,PV`	`CTD`	2.5.2.3.3 (Anmerkung 1)
10	`CU,CD,R,PV`	`CTUD`	2.5.2.3.3 (Anmerkung 1)
11	`IN,PT`	`TP`	2.5.2.3.4
12	`IN,PT`	`TON`	2.5.2.3.4
13	`IN,PT`	`TOF`	2.5.2.3.4

ANMERKUNG 1 `LD` ist als Standard-Funktionsbaustein-Eingang nicht notwendig, weil die `LD` Funktionalität in `PV` eingeschlossen ist.

ANMERKUNG 2 Die Nummerierung der Eigenschaften in dieser Tabelle soll die Konsistenz mit der ersten Ausgabe der IEC 61131-3 gewährleisten.

3.3 Sprache ST (Strukturierter Text)

Dieser Abschnitt legt die Semantik der Sprache ST (Strukturierter Text) fest, deren Syntax in B.3 definiert ist. In dieser Sprache muss das Ende einer Textzeile in derselben Weise behandelt werden, wie ein Leerzeichen (SP), wie es in 2.1.4 definiert ist.

3.3.1 Ausdrücke

Ein *Ausdruck* ist ein Konstrukt, das bei Auswertung einen Wert liefert, der einem der Datentypen entspricht, die in 2.3.1 und 2.3.3 definiert sind. Die maximal erlaubte Länge von Ausdrücken ist ein **implementierungsabhängiger Parameter**.

Ausdrücke bestehen aus Operatoren und Operanden. Ein *Operand* muss ein Literal sein, wie es in 2.2 definiert ist, eine Variable, wie in 2.4 definiert, ein Funktionsaufruf wie in 2.5 definiert, oder ein anderer Ausdruck sein.

Die *Operatoren* der Sprache ST sind in Tabelle 55 zusammengefasst. Die Auswertung eines Ausdrucks besteht aus Anwenden der Operatoren auf die Operanden, in der Reihenfolge, die durch die Rangfolge der Operatoren definiert ist, die in Tabelle 55 gezeigt ist. Der Operator mit der höchsten Rangfolge in einem Ausdruck muss zuerst angewendet werden, gefolgt vom Operator der nächst-niederen Rangfolge usw., bis die Auswertung vollendet ist. Operatoren mit gleichem Rang müssen angewendet werden, wie sie im Ausdruck von links nach rechts beschrieben sind. Z. B., wenn `A`, `B`, `C` und `D` den Typ `INT` und entsprechend die Werte 1, 2, 3 und 4 haben, dann muss

```
A+B-C*ABS(D)
```

als -9 errechnet werden und

```
(A+B-C)*ABS(D)
```

0 ergeben.

Wenn ein Operator zwei Operanden hat, muss der linke Operand zuerst ausgewertet werden. Z. B. muss im Ausdruck

```
SIN(A)*COS(B)
```

der Ausdruck `SIN(A)` zuerst ausgewertet werden, gefolgt von `COS(B)`, gefolgt von der Auswertung des Produkts.

Die folgenden Bedingungen bei der Ausführung von Operatoren müssen im Sinne von 1.5.1 als **Fehler** behandelt werden:

1) Es wird ein Versuch gemacht, durch Null zu dividieren.
2) Operanden haben nicht den korrekten Datentyp für die Operation.
3) Das Ergebnis einer numerischen Operation überschreitet den Wertebereich seines Datentyps.

Boolesche Ausdrücke brauchen nur in dem Umfang ausgewertet werden, der notwendig ist, um den resultierenden Wert zu bestimmen. Wenn z. B. `A<=B` ist, dann genügt es, nur den Ausdruck `(A>B)` auszuwerten, um zu entscheiden, dass der Wert des Ausdrucks

```
(A>B)&(C<D)
```

den booleschen Wert 0 hat.

Funktionen müssen als Elemente von Ausdrücken aufgerufen werden, die aus dem Funktionsnamen bestehen, dem eine eingeklammerte Liste von Argumenten folgt, wie in 2.5.1.1 definiert ist.

Wenn ein Operator in einem Ausdruck eine der überladenen Funktionen, wie in 2.5.1.5 definiert, darstellt, dann müssen die Konvertierungen von Operanden und Ergebnissen den Regeln und Beispielen wie in 2.5.1.4 entsprechen.

Tabelle 55 – Operatoren der Sprache ST

Nr.	Operation [a]	Symbol	Rang
1	Klammerung	(Ausdruck)	HÖCHSTER
2	Funktions-Bearbeitung BEISPIELE:	Bezeichner (Argument-Liste) `LN(A), MAX(X,Y), etc.`	
4	Negation	-	
5	Komplement	`NOT`	
3	Potenzierung [b]	**	
6	Multiplizieren	*	
7	Dividieren	/	
8	Modulo	`MOD`	
9	Addieren	+	
10	Subtrahieren	-	
11	Vergleich	`< , > , <= , >=`	
12	Gleichheit	=	
13	Ungleichheit	<>	

Nr.	Operation [a]	Symbol	Rang
14	Boolesches UND	&	
15	Boolesches UND	AND	
16	Boolesches Exklusiv ODER	XOR	
17	Boolesches ODER	OR	NIEDERSTER
ANMERKUNG Die Nummerierung der Eigenschaften in dieser Tabelle soll die Konsistenz mit der ersten Ausgabe der IEC 61131-3 gewährleisten.			
[a] Dieselben Einschränkungen gelten sowohl für die Operanden dieser Operatoren als auch für die Eingänge der entsprechenden Funktionen, die in 2.5.1.5 definiert sind.			
[b] Das Ergebnis der Auswertung des Ausdrucks A**B muss dasselbe sein, wie das Ergebnis der Auswertung von EXPT(A,B), wie in Tabelle 24 definiert ist.			

3.3.2 Anweisungen

Die Anweisungen der Sprache ST sind in Tabelle 56 zusammengefasst. Anweisungen müssen durch Semikolons abgeschlossen werden, wie in der Syntax von B.3 festgelegt ist. Die maximal erlaubte Länge von Anweisungen ist ein **implementierungsabhängiger Parameter**.

Tabelle 56 – Anweisungen der Sprache ST

Nr.	Anweisungstyp/Verweis	Beispiele
1	Zuweisung (3.3.2.1)	`A := B;  CV := CV+1; C := SIN(X);`
2	Funktionsbaustein-Aufruf und Gebrauch des FB-Ausgang (3.3.2.2)	`CMD_TMR(IN:=%IX5, PT:=T#300ms) ;` `A := CMD_TMR.Q;`
3	RETURN (3.3.2.2)	`RETURN ;`

4	IF (3.3.2.3)	`D := B*B - 4*A*C ;`
		`IF D < 0.0 THEN NROOTS := 0 ;`
		`ELSIF D = 0.0 THEN`
		`  NROOTS := 1 ;`
		`  X1 := - B/(2.0*A) ;`
		`ELSE`
		`  NROOTS := 2 ;`
		`  X1 := (- B + SQRT(D))/(2.0*A) ;`
		`  X2 := (- B - SQRT(D))/(2.0*A) ;`
		`END_IF ;`

5	CASE (3.3.2.3)	`TW := BCD_TO_INT(THUMBWHEEL);`
		`TW_ERROR := 0;`
		`CASE TW OF`
		`  1,5:  DISPLAY := OVEN_TEMP;`
		`  2:  DISPLAY := MOTOR_SPEED;`
		`  3:  DISPLAY := GROSS - TARE;`
		`  4,6..10: DISPLAY:= STATUS(TW-4);`
		`ELSE  DISPLAY  := 0 ;`
		`  TW_ERROR := 1;`
		`END_CASE;`
		`QW100 := INT_TO_BCD(DISPLAY);`

Nr.	Anweisungstyp/Verweis	Beispiele

6	FOR (3.3.2.4)	`J := 101 ;`
		`FOR I := 1 TO 100 BY 2 DO`
		`  IF WORDS[I] = 'KEY' THEN`
		`  J := I ;`
		`  EXIT ;`
		`  END_IF ;`
		`END_FOR ;`

7	WHILE (3.3.2.4)	`J := 1;`
		`WHILE J <= 100 & WORDS[J] <> 'KEY' DO`
		`  J := J+2 ;`
		`END_WHILE ;`

8	Wiederholung REPEAT (3.3.2.4)	`J := -1 ;`
		`REPEAT`
		`  J := J+2 ;`
		`UNTIL J = 101 OR WORDS[J] = 'KEY'`
		`END_REPEAT ;`

9	Ausgang EXIT (3.3.2.4) [a]	`EXIT ;`
10	Leer-Anweisung	`;`
[a] Wenn die Anweisung EXIT (9) unterstützt wird, dann muss sie für alle Wiederholungsanweisungen (FOR, WHILE, REPEAT) unterstützt werden, die in der Implementierung unterstützt werden.		

3.3.2.1 Zuweisungen

Die Zuweisung ersetzt den aktuellen Wert einer Einzel- oder Multi-Elementvariablen durch das Ergebnis der Auswertung eines Ausdrucks. Eine Zuweisung muss aus einer Variablen-Angabe auf der linken Seite bestehen, gefolgt von einem *Zuweisungsoperator* ":=", gefolgt vom Ausdruck, der auszuwerten ist. Z. B. wird die Anweisung

```
A := B ;
```

benutzt werden, um den einzelnen Datenwert der Variablen A durch den aktuellen Wert der Variablen B zu ersetzen, falls beide den Typ INT haben. Falls aber A und B den Typ ANALOG_CHANNEL_CONFIGURATION haben, wie in Tabelle 12 beschrieben ist, dann werden die Werte von allen Elementen der strukturierten Variablen A durch die aktuellen Werte der entsprechenden Elemente der Variablen B ersetzt werden.

Wie in Bild 6 veranschaulicht, muss die Zuweisung auch benutzt werden, um den Wert zuzuweisen, der von einer Funktion zurückzugeben ist; dies geschieht durch Platzieren des Funktionsnamens auf die linke Seite des Zuweisungsoperators im Rumpf der Funktionsdeklaration. Der Wert, der von der Funktion zurückgegeben wird, muss das Ergebnis der letzten Auswertung einer solchen Zuweisung sein. Es ist ein **Fehler**, von der Auswertung einer Funktion mit einem ENO Ausgang mit einem Wert TRUE zurückzukehren oder mit einem nicht-vorhandenen ENO Ausgang, wenn nicht mindestens eine solche Zuweisung gemacht wurde.

3.3.2.2 Steueranweisungen für Funktion und Funktionsbaustein

Steueranweisungen für Funktionen und Funktionsbausteine bestehen aus Mechanismen zum Aufrufen von Funktionsbausteinen und zum Steuern der Rückkehr zur aufrufenden Einheit, bevor das physikalische Ende einer Funktion oder eines Funktionsbausteins erreicht ist.

Die Auswertung der Funktion muss als Teil der Auswertung des Ausdrucks aufgerufen werden, wie es in 3.3.1 festgelegt ist.

Funktionsbausteine müssen durch eine Anweisung aufgerufen werden, die aus dem Namen des Funktionsbausteins besteht, dem eine eingeklammerte Liste von Argumenten folgt, wie es in Tabelle 56 veranschaulicht ist. Die Regeln und Eigenschaften, die in 2.5.1.1 und Tabelle 19 a) für Funktionsaufrufe definiert sind,

gelten entsprechend, wenn man in diesen Regeln immer den Begriff „Funktion“ durch den Begriff „Funktionsbaustein“ ersetzt.

Die Anweisung RETURN (Rücksprung) muss ein vorzeitiges Verlassen einer Funktion, eines Funktionsbausteins oder eines Programms bewirken (z. B. als das Ergebnis der Auswertung einer IF Anweisung).

3.3.2.3 Auswahlanweisungen

Auswahlanweisungen umfassen die IF und CASE Anweisungen. Eine Auswahlanweisung wählt aufgrund einer angegebenen Bedingung eine (oder eine Gruppe) ihrer Anweisungen zur Auswertung aus. Beispiele für Auswahlanweisungen sind in Tabelle 56 angegeben.

Die IF Anweisung legt fest, dass eine Gruppe von Anweisungen nur ausgeführt wird, wenn der zugehörige boolesche Ausdruck den Wert 1 (TRUE) liefert. Falls die Bedingung den Wert FALSE hat, darf entweder keine Anweisung ausgeführt werden oder die Anweisungsgruppe ist auszuführen, die dem Schlüsselwort ELSE folgt (oder dem Schlüsselwort ELSIF, falls dessen zugehörige boolesche Bedingung den Wert TRUE hat).

Die CASE Anweisung besteht aus einem Ausdruck, der als eine Variable vom Typ ANY_INT oder von einem Aufzählungsdatentyp (der „Selektor“) berechnet werden muss und einer Liste von Anweisungsgruppen, wobei jede Gruppe mit einer Marke versehen ist, die aus einer oder mehreren ganzen Zahlen, Aufzählungswerten oder Bereichen von ganzzahligen Werten besteht. Dies legt fest, dass die erste Gruppe von Anweisungen ausgeführt werden muss, die einen der Bereiche hat, in dem der errechnete Wert des Selektors enthalten ist. Falls der Wert des Selektors nicht in einem der Bereichsfälle vorkommt, muss die Anweisungsfolge ausgeführt werden, die dem Schlüsselwort ELSE folgt (falls es in der CASE Anweisung auftritt). Andernfalls darf keine der Anweisungsfolgen ausgeführt werden.

Die maximal erlaubte Anzahl von Auswahlanweisungen in CASE Anweisungen ist ein **implementierungsabhängiger Parameter**.

3.3.2.4 Wiederholungsanweisungen

Wiederholungsanweisungen legen fest, dass eine Gruppe von zugehörigen Anweisungen wiederholt ausgeführt werden muss. Die FOR Anweisung wird benutzt, falls die Anzahl der Wiederholungen im voraus bestimmt werden kann; andernfalls werden die Konstrukte WHILE und REPEAT benutzt.

Die EXIT Anweisung muss benutzt werden, um Wiederholungen zu beenden, bevor die Ende-Bedingung erfüllt ist.

Wenn die EXIT Anweisung innerhalb eines geschachtelten Wiederholungskonstrukts liegt, muss die innerste Schleife verlassen werden, in der EXIT liegt; d. h. die Steuerung muss zur nächsten Anweisung nach dem ersten Schleifen-Ende übergehen (END_FOR, END_WHILE oder END_REPEAT), das der EXIT Anweisung folgt. Z. B. muss nach Ausführen der Anweisungen, die in Bild 22 gezeigt sind, der Wert der Variablen SUM 15 sein, falls der Wert der booleschen Variablen FLAG = 0 ist und 6, wenn FLAG = 1 ist.

```
SUM := 0 ;
FOR I := 1 TO 3 DO
  FOR J := 1 TO 2 DO
    IF FLAG THEN EXIT ; END_IF
    SUM := SUM + J ;
  END_FOR ;
  SUM := SUM + I ;
END_FOR ;
```

Bild 22 – Beispiel für die Anweisung EXIT

Die FOR Anweisung zeigt an, dass eine Anweisungsfolge bis zum Schlüsselwort END_FOR wiederholt ausgeführt werden muss; dabei wird der Steuervariablen der FOR Schleife ein fortschreitender Wert zugewiesen.

Die Steuervariable, der Anfangswert und der Endwert müssen Ausdrücke desselben ganzzahligen Typs sein (z. B. SINT, INT oder DINT) und dürfen nicht durch eine der wiederholten Anweisungen verändert werden. Die FOR Anweisung inkrementiert die Steuervariable von einem Anfangswert auf- oder abwärts zu einem Endwert in Inkrementen, die durch den Wert des Ausdrucks bestimmt sind; dieser Wert ist auf 1 voreingestellt. Die Prüfung der Ende-Bedingung wird am Anfang jeder Wiederholung durchgeführt, so dass die Anweisungsfolge nicht ausgeführt wird, falls der Anfangswert den Endwert überschreitet. Der Wert der Steuervariablen nach der Vollendung der FOR Schleife ist **implementierungsabhängig**.

Ein Beispiel für den Gebrauch der FOR Anweisung ist in Eigenschaft 6 der Tabelle 56 angegeben. In diesem Beispiel wird die FOR Schleife benutzt, um den Index J zu bestimmen, der das erste Auftreten (falls vorhanden) angibt für die Zeichenfolge „KEY“ unter den ganzzahligen Elementen eines Feldes von Folgen WORDS mit einem Indexbereich von (1..100). Falls kein Auftreten gefunden wird, wird J den Wert 101 haben.

Die WHILE Anweisung bewirkt, dass die Folge von Anweisungen bis zum Schlüsselwort END_WHILE wiederholt ausgeführt wird, bis der zugehörige boolesche Ausdruck falsch ist. Falls der Ausdruck von Anfang an falsch ist, wird die Gruppe von Anweisungen überhaupt nicht ausgeführt. Z. B. kann das Beispiel FOR...END_FOR, das in Tabelle 56 angegeben ist, auch mit der Konstruktion WHILE...END_WHILE geschrieben werden, die in Tabelle 56 gezeigt ist.

Die REPEAT Anweisung bewirkt, dass die Folge von Anweisungen bis zum Schlüsselwort UNTIL wiederholt (und mindestens einmal) ausgeführt wird, bis die zugehörige boolesche Bedingung wahr ist. Z. B. kann das Beispiel WHILE...END_WHILE, das in Tabelle 56 angegeben ist, auch mit der Konstruktion REPEAT...END_REPEAT geschrieben werden, die in Tabelle 56 gezeigt ist.

Die Anweisungen WHILE und REPEAT dürfen nicht benutzt werden, um Synchronisation zwischen Prozessen vorzunehmen, z. B. als eine „Warteschleife“ mit einer extern bestimmten Ende-Bedingung. Zu diesem Zweck müssen die AS-Elemente benutzt werden, die in 2.6 definiert sind.

Es muss ein **Fehler** im Sinne von 1.5.1 sein, wenn eine WHILE oder REPEAT Anweisung in einem Algorithmus benutzt wird, für den die Erfüllung der Schleifen-Ende-Bedingung oder der Ausführung einer EXIT Anweisung nicht garantiert werden kann.

4 Grafische Sprachen

Die grafischen Sprachen, die in dieser Norm definiert sind, sind KOP (Kontaktplan) und FBS (Funktionsbaustein-Sprache). Die Elemente der Ablaufsprache (AS), die in 2.6 definiert sind, können in Verbindung mit jeder dieser Sprachen benutzt werden.

4.1 Gemeinsame Elemente

Die Elemente, die in diesem Abschnitt definiert sind, gelten für die beiden grafischen Sprachen in dieser Norm; d. h. für KOP (Kontaktplan) und FBS (Funktionsbaustein-Sprache) und auch für die grafische Darstellung der Elemente der Ablaufsprache (AS).

4.1.1 Darstellung von Linien und Blöcken

Die grafischen Sprachelemente, die in diesem Abschnitt definiert sind, werden mit Linienelementen gezeichnet, die im Zeichensatz, der in 2.1.1 definiert ist, gezeichnet werden oder benutzen grafische oder semigrafische Elemente, wie es in Tabelle 57 gezeigt ist.

Linien können durch den Gebrauch von *Konnektoren* erweitert werden, wie in Tabelle 57 gezeigt ist. Es darf keine Datenspeicherung oder Zuordnung von Datenelementen mit dem Gebrauch von Konnektoren verbunden sein; deswegen muss es, um Mehrdeutigkeiten zu vermeiden, ein **Fehler** sein, wenn der Bezeichner, der als Konnektor-Marke benutzt wird, derselbe ist, wie der Name eines anderen mit Namen versehenen Elements innerhalb derselben Programm-Organisationseinheit.

Alle Beschränkungen der Netzwerktopologie in einer bestimmten Implementierung müssen als **implementierungsabhängige Parameter** ausgedrückt werden.

4.1.2 Flussrichtung in Netzwerken

Ein *Netzwerk* ist definiert als eine maximale Menge von grafisch miteinander verknüpften Elementen; dabei sind die linke und rechte Stromschiene im Fall der KOP-Sprache, die in 4.2 definiert ist, ausgenommen. Es muss dafür gesorgt werden, dass zu jedem Netzwerk oder jeder Netzwerkgruppe in einer grafischen Sprache eine *Netzwerk-Marke* zugeordnet werden kann, die rechts durch einen Doppelpunkt (:) abgeschlossen ist. Diese Marke muss die Form eines Bezeichners oder einer vorzeichenlosen dezimalen ganzen Zahl haben, wie es in Abschnitt 2 dieses Teils definiert ist. Der *Geltungsbereich* eines Netzwerks oder deren Marke muss *lokal* zur Programm-Organisationseinheit sein, in der sich das Netzwerk befindet. Beispiele von Netzwerken und Netzwerk-Marken sind im Anhang F gezeigt.

Grafische Sprachen werden verwendet, um den Fluss einer konzeptionellen Größe durch ein oder mehr Netzwerke darzustellen, die einen Steuerungsplan darstellen, das bedeutet:

– „Stromfluss“, analog zum Fluss von elektrischem Strom in einem elektromechanischen Relais-System, wird typischerweise in Kontaktplänen verwendet;

– „Signalfluss“, analog zum Fluss von Signalen zwischen Elementen eines Signalverarbeitungssystems, wird typischerweise in Funktionsbaustein-Diagrammen verwendet;

– „Aktivitätsfluss“, analog zum Steuerfluss zwischen Organisationselementen oder zwischen den Schritten eines elektromechanischen Schrittschaltwerks, wird typischerweise in Ablaufdiagrammen (Ablaufsprache) verwendet.

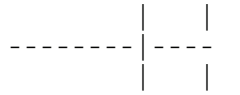
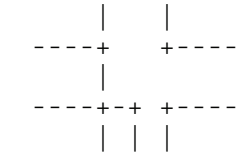
Die entsprechende konzeptionelle Größe muss entlang der Linien zwischen den Elementen eines Netzwerks nach den folgenden Regeln fließen:

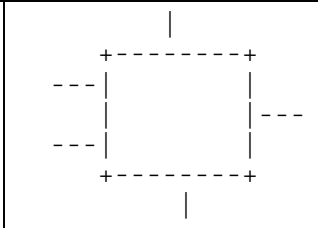
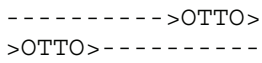
1) Der Stromfluss in der Sprache KOP muss von links nach rechts fließen.

2) Der Signalfluss in der Sprache FBS muss von der Ausgangsseite (rechts) einer Funktion oder eines Funktionsbausteins zur Eingangsseite (links) einer verbundenen Funktion oder eines Funktionsbausteins (bzw. mehrerer) fließen.

3) Der Aktivitätsfluss zwischen den AS-Elementen, die in 2.6 definiert sind, muss von der Unterseite eines Schritts durch die zugehörige Transition zur Oberseite des entsprechenden Nachfolger-Schritts (bzw. mehrerer) erfolgen.

Tabelle 57 – Darstellung von Linien und Blöcken

Nr.	Eigenschaft	Beispiel
	Horizontale Linien:	
1	ISO/IEC 10646-1 „Minus“-Zeichen	
2	Grafisch oder semigrafisch	
	Vertikale Linien:	
3	ISO/IEC 10646-1 „vertikale Linie“-Zeichen	
4	Grafisch oder semigrafisch	
	Horizontale/vertikale Verbindung:	
5	ISO/IEC 10646-1 „Plus“-Zeichen	
6	Grafisch oder semigrafisch	
	Linienkreuzung ohne Verbindung:	
7	ISO/IEC 10646-1 Zeichen	
8	Grafisch oder semigrafisch	
	Verbundene und nicht-verbundene Ecken:	
9	ISO/IEC 10646-1 Zeichen	
10	Grafisch oder semigrafisch	

Nr.	Eigenschaft	Beispiel
	Blöcke mit Verbindungslinien:	
11	ISO/IEC 10646-1 Zeichen	
12	Grafisch oder semigrafisch	
13	**Konnektoren mit ISO/IEC 10646-1 Zeichen:**	
	Konnektor	
	Fortsetzung einer verbundenen Linie	
14	Grafische oder semigrafische Konnektoren	

4.1.3 Auswertung von Netzwerken

Die Reihenfolge, in der Netzwerke und ihre Elemente ausgewertet werden, ist nicht notwendigerweise dieselbe wie die Reihenfolge, in der sie mit Marken versehen und dargestellt werden. Genauso ist es nicht notwendig, dass alle Netzwerke ausgewertet werden, bevor die Auswertung eines gegebenen Netzwerks wiederholt werden kann. Wenn aber der Rumpf einer Programm-Organisationseinheit aus einem oder mehreren Netzwerken besteht, müssen die Ergebnisse der Netzwerk-Auswertung innerhalb des besagten Rumpfs funktional äquivalent zur Befolgung der folgenden Regeln sein:

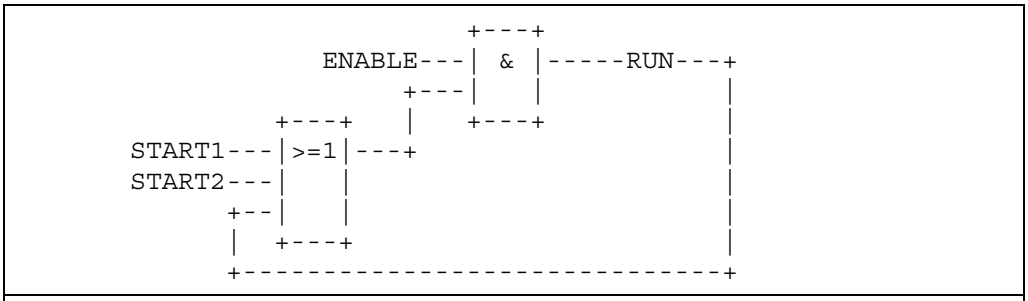
1) Kein Element eines Netzwerks darf ausgewertet werden, bevor die Zustände aller seiner Eingänge ausgewertet wurden.
2) Die Auswertung eines Netzwerk-Elements darf nicht abgeschlossen werden, bevor die Zustände aller seiner Ausgänge ermittelt wurden.
3) Die Auswertung eines Netzwerks ist nicht abgeschlossen, bevor die Ausgänge aller seiner Elemente ermittelt wurden, auch wenn das Netzwerk eines der Elemente zur Ausführungssteuerung enthält, die in Abschnitt 4.1.4 definiert sind.
4) Die Reihenfolge, in der Netzwerke ausgewertet werden, muss mit den Bestimmungen von 4.2.6 für die Sprache KOP und Abschnitt 4.3.3 für die Sprache FBS übereinstimmen.

Man sagt, es gibt einen *Rückkopplungspfad* in einem Netzwerk, wenn der Ausgang einer Funktion oder eines Funktionsbausteins als der Eingang einer Funktion oder eines Funktionsbausteins benutzt wird, der ihm im Netzwerk vorausgeht; die zugehörige Variable wird als eine *Rückkopplungsvariable* bezeichnet. Z. B. ist die boolesche Variable `RUN` die Rückkopplungsvariable im Beispiel, das in Bild 23 gezeigt ist. Eine Rückkopplungsvariable kann auch ein Ausgangselement einer Datenstruktur eines Funktionsbausteins sein, wie in 2.5.2 definiert ist.

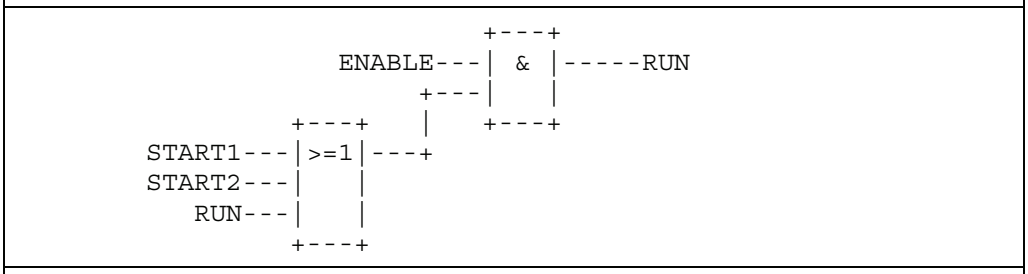
Rückkopplungspfade können in den grafischen Sprachen verwendet werden, die in 4.2 und 4.3 definiert sind; es gelten die folgenden Regeln:

1) Explizite Schleifen, wie die in Bild 23 a) gezeigten, dürfen nur in der Sprache FBS auftreten, die in 4.3 definiert ist.
2) Es muss für den Anwender möglich sein, **implementierungsabhängige** Mittel zu verwenden, um die Reihenfolge der Ausführung von Elementen in einer expliziten Schleife zu definieren, z. B. durch Wahl der Rückkopplungsvariablen, um eine implizite Schleife zu bilden, wie es in Bild 23 b) gezeigt ist.
3) Rückkopplungsvariablen müssen durch einen der Mechanismen initialisiert werden, die in Abschnitt 2 definiert sind. Der Anfangswert muss während der ersten Auswertung des Netzwerks benutzt werden. Es muss ein **Fehler** sein, wenn die Rückkopplungsvariable nicht initialisiert ist.
4) Sobald das Element mit einer Rückkopplungsvariablen als Ausgang einmal ermittelt wurde, muss der neue Wert der Rückkopplungsvariablen bis zur nächsten Auswertung des Elements benutzt werden.

a)



b)



c)

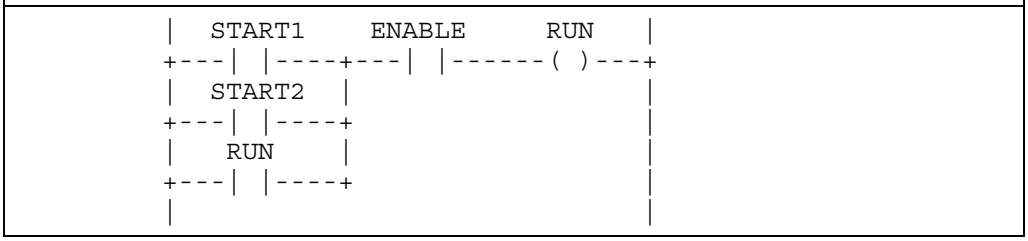


Bild 23 – Beispiel eines Rückkopplungspfads
a) Explizite Schleife
b) Implizite Schleife
c) KOP Sprache-Äquivalent

4.1.4 Elemente der Ausführungssteuerung

Die Übergabe der Ausführungssteuerung in den Sprachen KOP und FBS muss durch die grafischen Elemente dargestellt werden, die in Tabelle 58 gezeigt sind.

Sprünge müssen durch eine boolesche Signallinie gezeigt werden, die in einer Doppelpfeilspitze endet. Die Signallinie für eine Sprungbedingung muss bei einer booleschen Variablen, bei einem booleschen Ausgang einer Funktion oder eines Funktionsbausteins oder an einer Stromflusslinie eines Kontaktplans beginnen. Die Übergabe der Programmsteuerung an die bestimmte Netzwerk-Variable muss geschehen, wenn der boolesche Wert der Signallinie 1 (`TRUE`) ist; somit ist der unbedingte Sprung ein Sonderfall des bedingten Sprungs.

Das Ziel eines Sprungs muss eine Netzwerk-Marke innerhalb der Programm-Organisationseinheit sein, in der der Sprung auftritt. Falls der Sprung innerhalb eines Konstrukts `ACTION...END_ACTION` auftritt, muss das Sprungziel im selben Konstrukt liegen.

Bedingte Rücksprünge von Funktionen und Funktionsbausteinen müssen mit einer `RETURN` Konstruktion implementiert werden, wie in Tabelle 58 gezeigt ist. Die Programmausführung muss an die aufrufende Einheit zurückgegeben werden, wenn der boolesche Eingang 1 (`TRUE`) ist und muss in normaler Weise fortgesetzt werden, wenn der boolesche Eingang 0 (`FALSE`) ist. Unbedingte Rücksprünge müssen durch das physikalische Ende der Funktion oder des Funktionsbausteins erfolgen oder durch ein `RETURN` Element, das mit der linken Stromschiene in der KOP-Sprache verbunden ist, wie in Tabelle 58 gezeigt ist.

Tabelle 58 – Beispiele von grafischen Elementen zur Ausführungssteuerung

Nr.	Symbol/Beispiel	Erläuterung
1 2	1---->>LABELA | +---->>LABELA |	Unbedingter Sprung: Sprache FBS Sprache KOP
3	X---->>LABELB +---+ %IX20---| & |--->>NEXT %MX50---| | +---+ NEXT: +---+ %IX25---|>=1|---%QX100 %MX60---| | +---+	Bedingter Sprung: (Sprache FBS) Beispiel: Bedingter Sprung Sprungziel
4	| X +-| |---->>LABELB | | | %IX20 %MX50 +---| |-----| |--->>NEXT | | NEXT: | %IX25 %QX100 | +----| |----+----( )---+ | %MX60 | | +----| |----+ | | |	Bedingter Sprung: (Sprache KOP) Beispiel: Sprungbedingung Sprungziel
5 6	| X +--| |---<RETURN> | X---<RETURN>	Bedingter Rücksprung: Sprache KOP Sprache FBS
7 8	END_FUNCTION END_FUNCTION_BLOCK | +---<RETURN> |	Unbedingter Rücksprung: aus FUNCTION aus FUNCTION_BLOCK Alternative Darstellung in der Sprache KOP

4.2 Sprache Kontaktplan (KOP)

Dieser Abschnitt definiert die Sprache KOP für die Kontaktplan-Programmierung von Speicherprogrammierbaren Steuerungen.

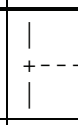
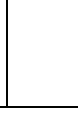
Ein KOP-Programm ermöglicht es, die Speicherprogrammierbare Steuerung mit den Mitteln der genormten grafischen Symbole Daten zu testen und zu verändern. Diese Symbole werden in Netzwerken in einer Weise

ähnlich den „Strompfaden“ eines Relais-Kontaktplans dargestellt. KOP-Netzwerke sind auf der rechten und linken Seite durch die *Stromschienen* begrenzt.

4.2.1 Stromschienen

Wie in Tabelle 59 gezeigt ist, muss ein KOP-Netzwerk links durch eine vertikale Linie begrenzt werden, die als *linke Stromschiene* bekannt ist, und rechts durch eine vertikale Linie, die als *rechte Stromschiene* bekannt ist. Die rechte Stromschiene darf explizit oder impliziert angegeben sein.

Tabelle 59 – Stromschienen

Nr.	Symbol	Beschreibung
1		**Linke Stromschiene** (mit angebundener horizontaler Verbindung)
2		**Rechte Stromschiene** (mit angebundener horizontaler Verbindung)

4.2.2 Verbindungselemente und Zustände

Wie in Tabelle 60 gezeigt, dürfen die Verbindungselemente horizontal und vertikal sein. Der Zustand des Verbindungselements wird als „`EIN`“ oder „`AUS`“ bezeichnet, entsprechend der booleschen Werte 1 oder 0. Der Begriff *Verbindungszustand* muss synonym mit dem Begriff *Stromfluss* angewendet werden.

Der Zustand der linken Stromschiene wird immer als `EIN` betrachtet. Für die rechte Stromschiene ist kein Zustand definiert.

Ein horizontales Verbindungselement wird durch eine horizontale Linie angezeigt. Ein horizontales Verbindungselement überträgt den Zustand des Elements auf seiner unmittelbaren linken Seite zum Element auf seiner unmittelbaren rechten Seite.

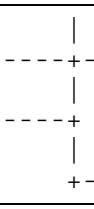
Das vertikale Verbindungselement besteht aus einer vertikalen Linie, die sich mit einem oder mehreren horizontalen Elementen auf beiden Seiten kreuzt. Der Zustand der vertikalen Verbindung muss das inklusive `OR` der `ON`-Zustände der horizontalen Verbindungen auf ihrer linken Seite darstellen; d. h., der Zustand der vertikalen Verbindung muss sein:

– `AUS`, falls die Zustände aller verbundenen horizontalen Verbindungen auf ihrer linken Seite `AUS` sind;

– `EIN`, falls der Zustand von einer oder mehrerer der verbundenen horizontalen Verbindungen auf ihrer linken Seite `EIN` ist.

Der Zustand der vertikalen Verbindung muss auf alle verbundenen Verbindungen auf ihrer rechten Seite kopiert werden. Der Zustand der vertikalen Verbindung darf nicht auf die verbundenen horizontalen Verbindungen auf der linken Seite kopiert werden.

Tabelle 60 – Verbindungselemente

Nr.	Symbol	Beschreibung
1		**Horizontale Verbindung**
2		**Vertikale Verbindung** (mit angebundenen horizontalen Verbindungen)

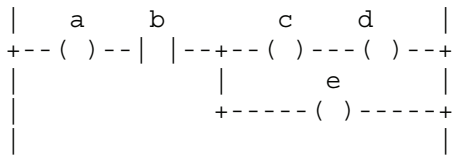
4.2.3 Kontakte

Ein *Kontakt* ist ein Element, das einen Zustand an die horizontale Verbindung auf seiner rechten Seite übergibt; dieser Zustand ergibt sich aus der booleschen UND-Verknüpfung des Zustands der horizontalen Verbindung auf seiner linken Seite mit einer entsprechenden Funktion einer zugehörigen booleschen Eingangs-, Ausgangs- oder Speicher-Variablen. Ein Kontakt verändert nicht den Wert der zugehörigen booleschen Variablen. Die genormten Kontaktsymbole sind in Tabelle 61 angegeben.

4.2.4 Spulen

Eine *Spule* kopiert den Zustand der Verbindung auf ihrer linken Seite ohne Veränderung auf die Verbindung auf ihrer rechten Seite und speichert eine entsprechende Funktion des Zustandes oder des Übergangs der linken Verbindung in die boolesche Variable. Die genormten Spulensymbole sind in Tabelle 62 angegeben.

BEISPIEL Im Strompfad, der unten gezeigt ist, ist der Wert des booleschen Ausgangs a immer TRUE, während der Wert der Ausgänge c, d und e nach Vollendung der Auswertung der Strompfads gleich dem Wert des Eingangs b ist.



4.2.5 Funktionen und Funktionsbausteine

Die Darstellung der Funktionen und Funktionsbausteine in der Sprache KOP muss so sein, wie es in Abschnitt 2 definiert ist, mit den folgenden Ausnahmen:

1) Die Verbindungen der aktuellen Variablen dürfen optional durch die Angabe der entsprechenden Daten oder Variable außerhalb des Blocks gezeigt werden; dabei sind sie benachbart zu formalen Namen der Variable auf der Innenseite anzugeben.
2) Mindestens ein boolescher Eingang und ein boolescher Ausgang müssen an jedem Block gezeigt werden, um den Stromfluss durch den Block zu ermöglichen.

4.2.6 Reihenfolge der Netzwerk-Auswertung

Innerhalb einer Programm-Organisationseinheit, die in KOP geschrieben ist, müssen die Netzwerke so, wie sie im Kontaktplan erscheinen, in einer Reihenfolge von oben nach unten ausgewertet werden, ausgenommen, diese Reihenfolge ist durch Elemente der Ausführungssteuerung verändert, die in 4.1.4 definiert sind.

Tabelle 61 – Kontakte [a]

Statische Kontakte		
Nr.	**Symbol**	**Beschreibung**
1 **2**	*** --| |-- oder *** --! !--	**Schließer** Der Zustand der linken Verbindung wird auf die rechte Verbindung kopiert, wenn der Zustand der zugehörigen booleschen Variablen (gekennzeichnet durch "***") EIN ist. Anderenfalls ist der Zustand der linken Verbindung AUS.
3 **4**	*** --|/|-- oder *** --!/!--	**Öffner** Der Zustand der linken Verbindung wird auf die rechte Verbindung kopiert, wenn der Zustand der zugehörigen booleschen Variablen (gekennzeichnet durch "***") AUS ist. Anderenfalls ist der Zustand der linken Verbindung AUS.
Kontakte zur Erkennung von Übergängen (Flanken)		
5 **6**	*** --|P|-- oder *** --!P!--	**Kontakt zur Erkennung von positivem Übergang** Der Zustand der linken Verbindung ist von einer Auswertung dieses Elements zur nächsten EIN, wenn ein Übergang der zugehörigen Variablen von AUS nach EIN erkannt wird, und zwar zur selben Zeit, in der der Zustand der linken Verbindung EIN ist. Der Zustand der linken Verbindung muss in allen anderen Zeiten AUS sein.
7 **8**	*** --|N|-- oder *** --!N!--	**Kontakt zur Erkennung von negativem Übergang** Der Zustand der linken Verbindung ist von einer Auswertung dieses Elements zur nächsten EIN, wenn ein Übergang der zugehörigen Variablen von EIN nach AUS erkannt wird, und zwar zur selben Zeit, in der der Zustand der linken Verbindung EIN ist. Der Zustand der linken Verbindung muss in allen anderen Zeiten AUS sein.
[a] Wie in 2.2.2 festgelegt, muss das Ausrufungszeichen "!" benutzt werden, wenn ein nationaler Zeichensatz den vertikalen Balken "|" nicht unterstützt.		

Tabelle 62 – Spulen (Abschlussoperationen)

Nr.	Symbol	Beschreibung
Spulen		
1	*** --( )--	**Spule** Der Zustand der linken Verbindung wird auf die zugehörige boolesche Variable und die rechte Verbindung kopiert.
2	*** --(/)--	**Negative Spule** Der Zustand der linken Verbindung wird auf die rechte Verbindung kopiert. Die Invertierung des Zustands der linken Verbindung wird auf die zugehörige boolesche Variable kopiert; d. h., falls der Zustand der linken Verbindung AUS ist, dann ist der Zustand der zugehörigen Variablen EIN und umgekehrt.
Speichernde Spulen		
3	*** --(S)--	**S Spule (Setzen)** Die zugehörige boolesche Variable wird auf den Zustand EIN gesetzt, wenn die linke Verbindung im EIN Zustand ist und bleibt gesetzt, bis sie durch die R Spule zurückgesetzt wird.

Nr.	Symbol	Beschreibung
4	*** --(R)--	**R Spule Rücksetzen** Die zugehörige boolesche Variable wird auf den Zustand AUS gesetzt, wenn die linke Verbindung im EIN Zustand ist und bleibt zurückgesetzt, bis sie durch die S Spule gesetzt wird.
Spulen zur Erkennung von Übergängen (Flanke)		
8	*** --(P)--	**Spule zur Erkennung von positivem Übergang** Der Zustand der zugehörigen booleschen Variablen ist von einer Auswertung des Elements zur nächsten EIN, wenn ein Übergang der linken Verbindung von AUS nach EIN erkannt wird. Der Zustand der linken Verbindung wird immer auf die rechte Verbindung kopiert.
9	*** --(N)--	**Spule zur Erkennung von negativem Übergang** Der Zustand der zugehörigen booleschen Variablen ist von einer Auswertung des Elements zur nächsten EIN, wenn ein Übergang der linken Verbindung von EIN nach AUS erkannt wird. Der Zustand der linken Verbindung wird immer auf die rechte Verbindung kopiert.
ANMERKUNG Die Eigenschaften 5, 6 und 7 der ersten Ausgabe sind in dieser Ausgabe gelöscht.		

4.3 Funktionsbaustein-Sprache (FBS)

4.3.1 Allgemeines

Dieser Abschnitt definiert FBS, eine grafische Sprache zur Programmierung von Speicherprogrammierbaren Steuerungen, die mit der Dokumentationsnorm IEC 60617-12 soweit wie möglich übereinstimmt. Wo Widersprüche zwischen dieser Norm und der IEC 60617-12 bestehen, müssen die Festlegungen dieser Norm für die Programmierung von SPS in der Sprache FBS gelten.

Die Festlegungen von Abschnitt 2 und Abschnitt 4.1 müssen auf die Erstellung und Interpretation von SPS-Programmen der Sprache FBS angewendet werden.

Beispiele für den Gebrauch der Sprache FBS sind in Anhang F angegeben.

4.3.2 Kombination von Elementen

Elemente der Sprache FBS müssen durch Signalfluss-Linien verbunden werden, die den Vereinbarungen von 4.1.2 folgen.

Ausgänge von Funktionsbausteinen dürfen nicht miteinander verbunden werden. Insbesondere ist das KOP-Konstrukt „Verdrahtetes-ODER“ nicht in der Sprache FBS zulässig; stattdessen ist ein impliziter boolescher „ODER“-Block erforderlich, wie in Bild 24 gezeigt ist.

a) **b)**

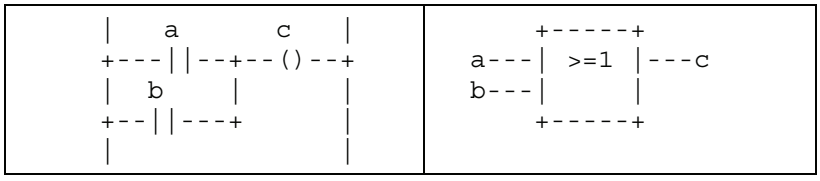


Bild 24 – Beispiele für boolesches ODER
a) „Verdrahtetes ODER“ in KOP
b) Funktion in FBS

4.3.3 Reihenfolge der Netzwerk-Auswertung

Wenn eine Programm-Organisationseinheit, die in der Sprache FBS geschrieben ist, mehr als ein Netzwerk enthält, muss der Hersteller **implementierungsabhängige** Mittel bieten, mit denen der Anwender die Reihenfolge der Ausführung der Netzwerke ermitteln kann.

Anhang A
(normativ)

Spezifikationsmethode für Textsprachen

Programmiersprachen werden mit Hilfe einer *Syntax* spezifiziert, die die zulässigen Kombinationen der Symbole festlegt, die zur Programmdefinition benutzt werden dürfen und mit Hilfe einer Anzahl von *semantischen* Regeln, die die Beziehung zwischen den programmierten Operationen und den Symbolkombinationen festlegt, die durch die Syntax definiert sind.

A.1 Syntax

Eine Syntax ist durch eine Menge von *terminalen Symbolen* definiert, die zur Programm-Spezifikation benutzt werden, weiterhin durch eine Menge von *nicht-terminalen Symbolen*, die mit Hilfe der terminalen Symbole definiert werden, und schließlich durch eine Anzahl von *Produktionsregeln*, die derartige Definitionen spezifizieren.

A.1.1 Terminale Symbole

Die terminalen Symbole für SPS-Programme in Textsprache müssen aus Zeichenkombinationen mit dem Zeichensatz bestehen, der in 2.1.1 definiert ist.

Zum Zweck dieses Teils bestehen terminale Textsymbole aus einer geeigneten Zeichenfolge, die in paarweise einfache oder doppelte Anführungszeichen eingeschlossen ist. Z. B. kann ein terminales Symbol, das durch die Zeichenfolge ABC dargestellt ist, sowohl durch

```
"ABC"
```

als auch durch

```
'ABC'
```

repräsentiert sein.

Dies erlaubt die Darstellung von Zeichenfolgen, die sowohl einfache als auch doppelte Anführungszeichen beinhalten; z. B. würde ein terminales Symbol, das selbst aus einem doppelten Anführungszeichen besteht, durch `'"'` dargestellt werden.

Ein besonderes terminales Symbol, das in dieser Syntax benutzt wird, ist das Begrenzungssymbol Zeilenende, das durch die Zeichenfolge `EOL` ohne Anführungszeichen dargestellt wird. Dieses Symbol muss standardmäßig aus dem Zeichen „Paragraph Separator“ bestehen, das als hexadezimaler Code 2029 durch ISO/IEC 10646-1 definiert ist.

Ein zweites besonderes terminales Symbol, das in dieser Syntax benutzt wird, ist die „Null-Folge“; das ist eine Zeichenfolge, die keine Zeichen enthält. Sie wird durch das terminale Symbol `NIL` dargestellt.

Die Groß-/Kleinschreibung von Buchstaben ist in terminalen Symbolen nicht signifikant.

A.1.2 Nicht-terminale Symbole

Nicht-terminale Symbole müssen durch eine Folge von Kleinbuchstaben, Zahlen und dem Unterstrich-Zeichen `(_)` dargestellt werden; dabei wird mit einem Kleinbuchstaben begonnen. Z. B. sind die Folgen

```
nonterm1
```

und

```
non_term_2
```

gültige nicht-terminale Symbole, während die Folgen

```
3nonterm
```

und

```
_nonterm4
```

es nicht sind.

A.1.3 Produktionsregeln

Die Produktionsregeln für textuelle SPS-Programmiersprachen müssen eine *erweiterte Grammatik* bilden, in der jede Regel folgende Form hat:

```
non_terminal_symbol ::= extended_structure
```

Diese Regel darf gelesen werden als:

„Ein non_terminal_symbol darf aus einer extended_structure bestehen."

Erweiterte Strukturen dürfen nach den folgenden Regeln gebildet werden:

1) Die Null-Folge `NIL` ist eine erweiterte Struktur.
2) Ein terminales Symbol ist eine erweiterte Struktur.
3) Ein nicht-terminales Symbol ist eine erweiterte Struktur.
4) Wenn `S` eine erweiterte Struktur ist, dann sind folgende Ausdrücke auch erweiterte Strukturen:

   `(S)`, bedeutet `S` selbst;

   `{S}`, *closure* (Einschluss), bedeutet null oder mehrere Aneinanderreihungen von `S`;

   `[S]`, *option* (Auswahl), bedeutet null oder ein Auftreten von `S`.
5) Sind `S1` und `S2` erweiterte Strukturen, dann sind die folgenden Ausdrücke erweiterte Strukturen:

   `S1|S2`, *alternation* (Auswahl), bedeutet die Auswahl von `S1` oder `S2`;

   `S1 S2`, *concatenation* (Aneinanderreihung), bedeutet `S1` gefolgt von `S2`.
6) Concatenation hat *Vorrang* vor alternation; d. h. `S1|S2 S3` ist äquivalent mit `S1|(S2 S3)` und `S1 S2|S3` ist äquivalent mit `(S1 S2)|S3`.

A.2 Semantische Regeln

Die semantischen Regeln von SPS-Programmiersprachen in Textform sind in diesem Teil von IEC 61131 definiert durch entsprechende Erklärung in natürlicher Sprache; sie ist zusammen mit den Produktionsregeln angegeben, die auf die Beschreibungen in den entsprechenden Abschnitten verweisen. In diesen semantischen Regeln sind die standardmäßigen Wahlmöglichkeiten festgelegt, die dem Anwender und dem Hersteller zu Verfügung stehen.

In einigen Fällen ist es angebracht, die semantische Information in eine erweiterte Struktur einzubetten. In diesen Fällen ist diese Information in paarigen spitzen Klammern eingeschlossen, z. B. `<semantische Information>`.

Anhang B
(normativ)

Formale Festlegungen der Sprachelemente

B.0 Programmiermodell

Der Inhalt dieses Anhangs ist normativ in dem Sinne, dass ein Compiler, der in der Lage ist, die ganze Syntax dieses Anhangs zu erkennen, in der Lage sein muss, die Syntax jeder Implementierung der Textsprache zu erkennen, die diese Norm erfüllt.

PRODUKTIONSREGELN:

```
library_element_name ::= data_type_name | function_name
        | function_block_type_name |program_type_name
        | resource_type_name | configuration_name

library_element_declaration ::= data_type_declaration
        | function_declaration | function_block_declaration
        | program_declaration | configuration_declaration
```

SEMANTIK: Diese Produktionen spiegeln das grundlegende Programmiermodell wider, das in 1.4.3 definiert ist; dort sind die *Deklarationen* die grundlegenden Mechanismen zur Produktion von mit Namen versehenen *Bibliothekselementen*. Die Syntax und Semantik der oben angegebenen nicht-terminalen Symbole sind in den unten aufgeführten Abschnitten definiert.

Nicht-terminales Symbol	Syntax	Semantik
data_type_name data_type_declaration	B.1.3	2.3
function_name function_declaration	B.1.5.1	2.5.1
function_block_type_name function_block_declaration	B.1.5.2	2.5.2
program_type_name program_declaration	B.1.5.3	2.5.3
resource_type_name configuration_name configuration_declaration	B.1.7	2.7

B.1 Gemeinsame Elemente

B.1.1 Buchstaben, Ziffern und Bezeichner

PRODUKTIONSREGELN:

```
letter  ::=  'A' | 'B' | <...> | 'Z' | 'a' | 'b' | <...> | 'z'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
octal_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
hex_digit ::= digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
```

```
identifier ::= (letter | ('_' (letter | digit)))  {['_'] (letter | digit)}
```

SEMANTIK:

Die spitzen Klammern < . . . > bedeuten hier die Folge der 26 Buchstaben der ISO/IEC 10646-1.

Zeichen aus den nationalen Zeichensätzen dürfen benutzt werden, doch kann in diesem Fall die internationale Portierbarkeit der gedruckten Darstellung von Programmen nicht gewährleistet werden.

B.1.2 Konstanten

PRODUKTIONSREGEL:

```
constant ::= numeric_literal | character_string | time_literal
     | bit_string_literal | boolean_literal
```

SEMANTIK:

Die äußeren Darstellungen von Daten, die in 2.2 beschrieben sind, werden in diesem Anhang als „constants“ bezeichnet.

B.1.2.1 Numerische Literale

PRODUKTIONSREGELN:

```
numeric_literal ::= integer_literal | real_literal

integer_literal ::= [ integer_type_name '#' ]
     ( signed_integer | binary_integer | octal_integer | hex_integer)

signed_integer ::= ['+' |'-'] integer

integer ::= digit {['_'] digit}

binary_integer ::= '2#' bit {['_'] bit}

bit ::= '1' | '0'

octal_integer ::= '8#' octal_digit {['_'] octal_digit}

hex_integer ::= '16#' hex_digit {['_'] hex_digit}

real_literal ::= [ real_type_name '#' ]
     signed_integer '.' integer [exponent]

exponent ::= ('E' | 'e') ['+'|'-'] integer

bit_string_literal ::=
     [ ('BYTE' | 'WORD' | 'DWORD' | 'LWORD') '#' ]
     ( unsigned_integer | binary_integer | octal_integer | hex_integer)

boolean_literal ::=
     ( [ 'BOOL#' ] ( '1' | '0' )  )| 'TRUE' | 'FALSE'
```

SEMANTIK: Siehe 2.2.1.

B.1.2.2 Zeichenfolgen

PRODUKTIONSREGELN:

```
character_string ::=
     single_byte_character_string | double_byte_character_string

single_byte_character_string ::=
     "'" {single_byte_character_representation} "'"
```

```
double_byte_character_string ::=
     '"' {double_byte_character_representation} '"'

single_byte_character_representation ::= common_character_representation
     | "$'"| '"' | '$' hex_digit hex_digit

double_byte_character_representation ::= common_character_representation
     |'$"' | "'"| '$' hex_digit hex_digit hex_digit hex_digit

common_character_representation ::=
     <any printable character except '$', '"' or "'">
     | '$$' | '$L' | '$N' | '$P' | '$R' | '$T'
     | '$l' | '$n' | '$p' | '$r' | '$t'
```

SEMANTIK: Siehe 2.2.2.

B.1.2.3 Zeitliterale

PRODUKTIONSREGEL:

```
time_literal ::= duration | time_of_day | date | date_and_time
```

SEMANTIK: Siehe 2.2.3.

B.1.2.3.1 Zeitdauer

PRODUKTIONSREGELN:

```
duration ::= ('T' | 'TIME') '#' ['-'] interval

interval ::= days | hours | minutes | seconds | milliseconds

days ::= fixed_point ('d') | integer ('d') ['_'] hours

fixed_point ::= integer [ '.' integer]

hours ::= fixed_point ('h') | integer ('h') ['_'] minutes

minutes ::= fixed_point ('m')  | integer ('m') ['_'] seconds

seconds ::= fixed_point ('s') | integer ('s') ['_'] milliseconds

milliseconds ::= fixed_point ('ms')
```

SEMANTIK: Siehe 2.2.3.1.

ANMERKUNG Die Semantik von 2.2.3.1 beschränkt zusätzlich die erlaubten Werte von `hours`, `minutes`, `seconds` und `milliseconds`.

B.1.2.3.2 Uhrzeit und Datum

PRODUKTIONSREGELN:

```
time_of_day ::= ('TIME_OF_DAY' | 'TOD')  '#' daytime

daytime ::= day_hour ':' day_minute ':' day_second

day_hour ::= integer

day_minute ::= integer

day_second ::= fixed_point

date ::= ('DATE' | 'D') '#' date_literal

date_literal ::= year '-' month '-' day

year ::= integer

month ::= integer
```

```
day ::= integer

date_and_time ::= ('DATE_AND_TIME' | 'DT') '#' date_literal '-' daytime
```

SEMANTIK: Siehe 2.2.3.2.

ANMERKUNG Die Semantik von 2.2.3.2 beschränkt zusätzlich die erlaubten Werte von `day_hour`, `day_minute`, `day_second`, `year`, `month` und `day`.

B.1.3 Datentypen

PRODUKTIONSREGELN:

```
data_type_name ::= non_generic_type_name | generic_type_name

non_generic_type_name ::=  elementary_type_name | derived_type_name
```

SEMANTIK: Siehe 2.3.

B.1.3.1 Elementare Datentypen

PRODUKTIONSREGELN:

```
elementary_type_name ::= numeric_type_name | date_type_name
     | bit_string_type_name | 'STRING' | 'WSTRING' | 'TIME'

numeric_type_name ::= integer_type_name | real_type_name

integer_type_name ::= signed_integer_type_name
     | unsigned_integer_type_name

signed_integer_type_name ::= 'SINT' | 'INT' | 'DINT' | 'LINT'

unsigned_integer_type_name ::= 'USINT' | 'UINT' | 'UDINT'  | 'ULINT'

real_type_name ::= 'REAL' | 'LREAL'

date_type_name ::= 'DATE' | 'TIME_OF_DAY' | 'TOD'  | 'DATE_AND_TIME'
     | 'DT'

bit_string_type_name ::= 'BOOL' | 'BYTE' | 'WORD' | 'DWORD' | 'LWORD'
```

SEMANTIK: Siehe 2.3.1.

B.1.3.2 Allgemeine Datentypen

PRODUKTIONSREGEL:

```
generic_type_name ::= 'ANY' | 'ANY_DERIVED' | 'ANY_ELEMENTARY'
     | 'ANY_MAGNITUDE' | 'ANY_NUM' | 'ANY_REAL' | 'ANY_INT' | 'ANY_BIT'
     | 'ANY_STRING' | 'ANY_DATE'
```

SEMANTIK: Siehe 2.3.2.

B.1.3.3 Abgeleitete Datentypen

PRODUKTIONSREGELN:

```
derived_type_name ::= single_element_type_name | array_type_name
     | structure_type_name | string_type_name

single_element_type_name ::= simple_type_name | subrange_type_name
     | enumerated_type_name

simple_type_name ::= identifier

subrange_type_name ::= identifier

enumerated_type_name ::= identifier
```

```
array_type_name ::= identifier

structure_type_name ::= identifier

data_type_declaration ::=
     'TYPE' type_declaration ';'
     {type_declaration ';'}
     'END_TYPE'

type_declaration ::= single_element_type_declaration
     | array_type_declaration
     | structure_type_declaration | string_type_declaration

single_element_type_declaration ::= simple_type_declaration
     | subrange_type_declaration | enumerated_type_declaration

simple_type_declaration ::= simple_type_name ':' simple_spec_init

simple_spec_init ::= simple_specification [':=' constant]

simple_specification ::= elementary_type_name | simple_type_name

subrange_type_declaration ::= subrange_type_name ':' subrange_spec_init

subrange_spec_init ::= subrange_specification [':=' signed_integer]

subrange_specification ::= integer_type_name '(' subrange')'
     | subrange_type_name

subrange ::= signed_integer '..' signed_integer

enumerated_type_declaration ::=
     enumerated_type_name ':' enumerated_spec_init

enumerated_spec_init ::= enumerated_specification [':=' enumerated_value]

enumerated_specification ::=
     ( '(' enumerated_value {',' enumerated_value} ')' )
     | enumerated_type_name

enumerated_value ::= [enumerated_type_name '#'] identifier

array_type_declaration ::= array_type_name ':' array_spec_init

array_spec_init ::= array_specification [':=' array_initialization]

array_specification ::= array_type_name
     | 'ARRAY' '[' subrange {',' subrange} ']' 'OF' non_generic_type_name

array_initialization ::=
     '[' array_initial_elements {',' array_initial_elements} ']'

array_initial_elements ::=
     array_initial_element | integer '(' [array_initial_element] ')'

array_initial_element ::= constant | enumerated_value
     | structure_initialization | array_initialization

structure_type_declaration ::=
     structure_type_name ':' structure_specification

structure_specification ::= structure_declaration | initialized_structure

initialized_structure ::=
     structure_type_name [':=' structure_initialization]

structure_declaration ::=
     'STRUCT' structure_element_declaration ';'
     {structure_element_declaration ';'}
     'END_STRUCT'
```

```
structure_element_declaration ::= structure_element_name ':'
     (simple_spec_init | subrange_spec_init  | enumerated_spec_init
     | array_spec_init | initialized_structure)

structure_element_name ::= identifier

structure_initialization ::=
     '(' structure_element_initializatio
     {','  structure_element_initialization} ')'

structure_element_initialization ::=
     structure_element_name ':=' (constant | enumerated_value
     | array_initialization | structure_initialization)

string_type_name ::= identifier

string_type_declaration ::= string_type_name ':'
     ('STRING'|'WSTRING') ['[' integer ']'] [':=' character_string]
```

SEMANTIK: Siehe 2.3.3.

B.1.4 Variablen

PRODUKTIONSREGELN:

```
variable ::= direct_variable | symbolic_variable

symbolic_variable ::= variable_name | multi_element_variable

variable_name ::= identifier
```

SEMANTIK: Siehe 2.4.1.

B.1.4.1 Direkt dargestellte Variablen

PRODUKTIONSREGELN:

```
direct_variable ::= '%' location_prefix size_prefix integer {'.' integer}

location_prefix ::= 'I' | 'Q' | 'M'

size_prefix ::= NIL | 'X' | 'B' | 'W' | 'D' | 'L'
```

SEMANTIK: Siehe 2.4.1.1.

B.1.4.2 Multi-Element-Variablen

PRODUKTIONSREGELN:

```
multi_element_variable ::= array_variable | structured_variable

array_variable ::= subscripted_variable subscript_list

subscripted_variable ::= symbolic_variable

subscript_list ::= '[' subscript {',' subscript} ']'

subscript ::= expression

structured_variable ::= record_variable '.' field_selector

record_variable ::= symbolic_variable

field_selector ::= identifier
```

SEMANTIK: Siehe 2.4.1.2.

B.1.4.3 Deklaration und Initialisierung

PRODUKTIONSREGELN:

```
input_declarations ::=
     'VAR_INPUT' ['RETAIN' | 'NON_RETAIN']
     input_declaration ';'
     {input_declaration ';'}
     'END_VAR'

input_declaration ::= var_init_decl | edge_declaration

edge_declaration ::= var1_list ':' 'BOOL' ('R_EDGE' | 'F_EDGE')

var_init_decl ::= var1_init_decl | array_var_init_decl
     | structured_var_init_decl | fb_name_decl | string_var_declaration

var1_init_decl ::=  var1_list ':'
     (simple_spec_init | subrange_spec_init | enumerated_spec_init)

var1_list ::= variable_name {',' variable_name}

array_var_init_decl ::= var1_list ':' array_spec_init

structured_var_init_decl ::= var1_list ':' initialized_structure

fb_name_decl ::= fb_name_list ':' function_block_type_name
     [ ':=' structure_initialization ]

fb_name_list ::= fb_name {',' fb_name}

fb_name ::= identifier

output_declarations ::=
     'VAR_OUTPUT' ['RETAIN' | 'NON_RETAIN']
       var_init_decl ';'
       {var_init_decl ';'}
     'END_VAR'

input_output_declarations ::=
     'VAR_IN_OUT'
     var_declaration ';'
     {var_declaration ';'}
     'END_VAR'

var_declaration ::=  temp_var_decl | fb_name_decl

temp_var_decl ::= var1_declaration | array_var_declaration
     | structured_var_declaration | string_var_declaration

var1_declaration ::=  var1_list  ':' (simple_specification
     | subrange_specification | enumerated_specification)

array_var_declaration ::= var1_list ':' array_specification

structured_var_declaration ::= var1_list ':' structure_type_name

var_declarations ::=
     'VAR' ['CONSTANT']
     var_init_decl ';'
     {(var_init_decl ';')}
     'END_VAR'

retentive_var_declarations ::=
     'VAR' 'RETAIN'
     var_init_decl ';'
     {var_init_decl ';'}
     'END_VAR'
```

```
located_var_declarations ::=
     'VAR' ['CONSTANT' | 'RETAIN' | 'NON_RETAIN']
       located_var_decl ';'
       {located_var_decl ';'}
     'END_VAR'

located_var_decl ::= [variable_name] location ':' located_var_spec_init

external_var_declarations ::=
     'VAR_EXTERNAL' ['CONSTANT']
     external_declaration ';'
     {external_declaration ';'}
     'END_VAR'

external_declaration ::= global_var_name ':'
     (simple_specification | subrange_specification
     | enumerated_specification | array_specification
     | structure_type_name | function_block_type_name)

global_var_name ::= identifier

global_var_declarations ::=
     'VAR_GLOBAL' ['CONSTANT' | 'RETAIN']
     global_var_decl ';'
     {global_var_decl ';'}
     'END_VAR'

global_var_decl ::= global_var_spec ':'
     [ located_var_spec_init | function_block_type_name ]

global_var_spec ::= global_var_list | [global_var_name] location

located_var_spec_init ::= simple_spec_init | subrange_spec_init
     | enumerated_spec_init | array_spec_init | initialized_structure
     | single_byte_string_spec | double_byte_string_spec

location ::= 'AT' direct_variable

global_var_list ::= global_var_name {',' global_var_name}

string_var_declaration ::= single_byte_string_var_declaration
     | double_byte_string_var_declaration

single_byte_string_var_declaration ::=
     var1_list ':' single_byte_string_spec

single_byte_string_spec ::=
     'STRING' ['[' integer ']'] [':=' single_byte_character_string]

double_byte_string_var_declaration ::=
     var1_list ':' double_byte_string_spec

double_byte_string_spec ::=
     'WSTRING' ['[' integer ']'] [':=' double_byte_character_string]

incompl_located_var_declarations ::=
      'VAR' ['RETAIN'|'NON_RETAIN']
         incompl_located_var_decl ';'
         {incompl_located_var_decl ';'}
      'END_VAR'

incompl_located_var_decl ::= variable_name incompl_location ':' var_spec

incompl_location ::= 'AT' '%' ('I' | 'Q' | 'M') '*'

var_spec ::= simple_specification
     | subrange_specification | enumerated_specification
     | array_specification | structure_type_name
     | 'STRING' ['[' integer ']'] | 'WSTRING' ['[' integer ']']
```

SEMANTIK: Siehe 2.4.2. Das Nicht-Terminal `function_block_type_name` ist in B.1.5.2 definiert.

B.1.5 Programm-Organisationseinheiten

B.1.5.1 Funktionen

PRODUKTIONSREGELN:

```
function_name ::= standard_function_name | derived_function_name

standard_function_name ::= <as defined in 2.5.1.5>

derived_function_name ::= identifier

function_declaration ::=
     'FUNCTION' derived_function_name ':'
           (elementary_type_name | derived_type_name)
        { io_var_declarations | function_var_decls }
        function_body
     'END_FUNCTION'

io_var_declarations ::= input_declarations | output_declarations |
     input_output_declarations

function_var_decls ::= 'VAR' ['CONSTANT']
     var2_init_decl ';' {var2_init_decl ';'} 'END_VAR'

function_body ::= ladder_diagram | function_block_diagram
     | instruction_list | statement_list | <andere Sprachen>

var2_init_decl ::= var1_init_decl | array_var_init_decl
     | structured_var_init_decl | string_var_declaration
```

SEMANTIK: Siehe 2.5.1.

ANMERKUNG 1 Die Syntax spiegelt nicht die Tatsache wider, dass jede Funktion mindestens eine Deklaration der Eingänge hat.

ANMERKUNG 2 Die Syntax spiegelt nicht die Tatsache wider, dass Deklarationen von Flanken, Funktionsbaustein-Verweisen und Aufrufen nicht in Funktionsrümpfen erlaubt sind.

ANMERKUNG 3 Kontaktpläne und Funktionsbaustein-Diagramme werden grafisch dargestellt, wie in Abschnitt 4 definiert ist. Die Nicht-Terminale `instruction_list` und `statement_list` sind in B.2.1 bzw. B.3.2 definiert.

B.1.5.2 Funktionsbausteine

PRODUKTIONSREGELN:

```
function_block_type_name ::= standard_function_block_name
     | derived_function_block_name

standard_function_block_name ::= <wie in 2.5.2.3 definiert>

derived_function_block_name ::= identifier

function_block_declaration ::=
     'FUNCTION_BLOCK' derived_function_block_name
        { io_var_declarations | other_var_declarations }
        function_block_body
     'END_FUNCTION_BLOCK'

other_var_declarations ::= external_var_declarations | var_declarations
     | retentive_var_declarations | non_retentive_var_declarations
     | temp_var_decls | incompl_located_var_declarations
```

```
temp_var_decls ::=
     'VAR_TEMP'
       temp_var_decl ';'
       {temp_var_decl ';'}
     'END_VAR'

non_retentive_var_decls ::=
     'VAR' 'NON_RETAIN'
       var_init_decl ';'
       {var_init_decl ';'}
     'END_VAR'

function_block_body ::= sequential_function_chart | ladder_diagram
     | function_block_diagram | instruction_list | statement_list
     | <andere Sprachen>
```

SEMANTIK: Siehe 2.5.2.

ANMERKUNG 1 Kontaktpläne und Funktionsbaustein-Diagramme werden grafisch dargestellt, wie in Abschnitt 4 definiert ist.

ANMERKUNG 2 Die Nicht-Terminale `sequential_function_chart`, `instruction_list` und `statement_list` sind in B.1.6, B.2 bzw. B.3.2 definiert.

B.1.5.3 Programme

PRODUKTIONSREGELN:

```
program_type_name :: = identifier

program_declaration ::=
     'PROGRAM' program_type_name
       { io_var_declarations | other_var_declarations
         | located_var_declarations | program_access_decls }
       function_block_body
     'END_PROGRAM'

program_access_decls ::=
     'VAR_ACCESS' program_access_decl ';'
        {program_access_decl ';' }
     'END_VAR'

program_access_decl ::= access_name ':' symbolic_variable ':'
     non_generic_type_name [direction]
```

SEMANTIK: Siehe 2.5.3.

B.1.6 Elemente der Ablaufsprache

PRODUKTIONSREGELN:

```
sequential_function_chart ::= sfc_network {sfc_network}

sfc_network ::= initial_step {step | transition | action}

initial_step ::=
     'INITIAL_STEP' step_name ':' {action_association ';'} 'END_STEP'

step ::= 'STEP' step_name ':' {action_association ';'} 'END_STEP'

step_name ::= identifier

action_association ::=
     action_name '(' [action_qualifier] {',' indicator_name} ')'

action_name ::= identifier
```

```
action_qualifier ::=
     'N' | 'R' | 'S' | 'P' |  'P0' | 'P1' | timed_qualifier ',' action_time

timed_qualifier ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'

action_time ::= duration | variable_name

indicator_name ::= variable_name

transition ::= 'TRANSITION'
     [transition_name] ['(' 'PRIORITY' ':=' integer ')']
     'FROM' steps 'TO' steps
     transition_condition
     'END_TRANSITION'

transition_name ::= identifier

steps ::= step_name | '(' step_name ',' step_name {',' step_name} ')'

transition_condition ::= ':' simple_instruction_list | ':=' expression ';'
     | ':' (fbd_network | rung)

action ::= 'ACTION' action_name ':'
              function_block_body
            'END_ACTION'
```

SEMANTIK: Siehe 2.6. Der Gebrauch von Netzwerken in Funktionsbaustein-Sprache und in Kontaktplan, die durch die Nicht-Terminale `fbd_network` bzw. `rung` bezeichnet sind, für die Darstellung von Übergangsbedingungen (`transition_conditions`) muss der Definition von 2.6.3 folgen.

ANMERKUNG 1 Die Nicht-Terminale `simple_instruction_list` und `expression` sind in B.2.1 und B.3.1 definiert.

ANMERKUNG 2 Der Term [`transition_name`] darf nur in der Produktion für `transition` verwendet werden, wenn die Eigenschaft 7 von Tabelle 41 unterstützt wird. Die sich daraus ergebende Produktion ist in Textform mit dieser Eigenschaft gleichbedeutend.

B.1.7 Konfigurationselemente

PRODUKTIONSREGELN:

```
configuration_name ::= identifier

resource_type_name ::= identifier

configuration_declaration ::=
     'CONFIGURATION' configuration_name
        [global_var_declarations]
        (single_resource_declaration
           | (resource_declaration {resource_declaration}))
        [access_declarations]
        [instance_specific_initializations]
     'END_CONFIGURATION'

resource_declaration ::=
     'RESOURCE' resource_name 'ON' resource_type_name
        [global_var_declarations]
        single_resource_declaration
      'END_RESOURCE'

single_resource_declaration ::=
     {task_configuration ';'}
     program_configuration ';'
     {program_configuration ';'}

resource_name ::= identifier
```

```
access_declarations ::=
     'VAR_ACCESS'
      access_declaration ';'
      {access_declaration ';'}
     'END_VAR'

access_declaration ::= access_name ':' access_path ':'
     non_generic_type_name [direction]

access_path ::= [resource_name '.'] direct_variable
     | [resource_name '.'] [program_name '.']
     {fb_name'.'} symbolic_variable

global_var_reference ::=
     [resource_name '.'] global_var_name ['.' structure_element_name]

access_name ::= identifier

program_output_reference ::= program_name '.' symbolic_variable

program_name ::= identifier

direction ::= 'READ_WRITE' | 'READ_ONLY'

task_configuration ::= 'TASK' task_name task_initialization

task_name := identifier

task_initialization ::=
     '(' ['SINGLE' ':=' data_source ',']
         ['INTERVAL' ':=' data_source ',']
         'PRIORITY' ':=' integer ')'

data_source ::= constant | global_var_reference
     | program_output_reference | direct_variable

program_configuration ::=
     'PROGRAM' [RETAIN | NON_RETAIN]
       program_name ['WITH' task_name] ':' program_type_name
       ['(' prog_conf_elements ')']

prog_conf_elements ::= prog_conf_element {',' prog_conf_element}

prog_conf_element ::= fb_task | prog_cnxn

fb_task ::= fb_name 'WITH' task_name

prog_cnxn ::= symbolic_variable ':=' prog_data_source
     | symbolic_variable '=>' data_sink

prog_data_source ::=
     constant | enumerated_value | global_var_reference | direct_variable

data_sink ::= global_var_reference | direct_variable

instance_specific_initializations ::=
     'VAR_CONFIG'
       instance_specific_init ';'
       {instance_specific_init ';'}
     'END_VAR'

instance_specific_init ::=
     resource_name '.' program_name '.' {fb_name '.'}
     ((variable_name [location] ':' located_var_spec_init) |
      (fb_name ':' function_block_type_name ':='
     structure_initialization))
```

SEMANTIK: Siehe 2.7.

ANMERKUNG Diese Syntax spiegelt nicht die Tatsache wider, dass die Zuweisungen der Ortsangaben nur als Verweise auf Variablen erlaubt sind, die durch die Angabe des Sterns auf der Ebene der Typdeklaration gekennzeichnet sind.

B.2 Sprache AWL (Anweisungsliste)

B.2.1 Anweisungen und Operationen

PRODUKTIONSREGELN:

```
instruction_list ::= il_instruction {il_instruction}

il_instruction ::= [label':'] [ il_simple_operation
      | il_expression
      | il_jump_operation
      | il_fb_call
      | il_formal_funct_call
      | il_return_operator ] EOL {EOL}

label ::= identifier

il_simple_operation ::= ( il_simple_operator [il_operand] )
     | ( function_name [il_operand_list] )

il_expression ::= il_expr_operator '(' [il_operand] EOL {EOL}
     [simple_instr_list] ')'

il_jump_operation ::= il_jump_operator label

il_fb_call ::= il_call_operator fb_name ['('
     (EOL {EOL} [ il_param_list ]) | [ il_operand_list ] ')']

il_formal_funct_call ::= function_name '(' EOL {EOL} [il_param_list] ')'

il_operand ::= constant | variable | enumerated_value

il_operand_list ::= il_operand {',' il_operand}

simple_instr_list ::= il_simple_instruction {il_simple_instruction}

il_simple_instruction ::=
     (il_simple_operation | il_expression | il_formal_funct_call)
     EOL {EOL}

il_param_list ::= {il_param_instruction} il_param_last_instruction

il_param_instruction ::= (il_param_assignment | il_param_out_assignment)
     ',' EOL {EOL}

il_param_last_instruction ::=
     ( il_param_assignment | il_param_out_assignment ) EOL {EOL}

il_param_assignment ::= il_assign_operator ( il_operand | ( '(' EOL {EOL}
     simple_instr_list ')' ) )

il_param_out_assignment ::= il_assign_out_operator variable
```

B.2.2 Operatoren

PRODUKTIONSREGELN:

```
il_simple_operator ::=   'LD' | 'LDN' | 'ST' | 'STN' | 'NOT' | 'S'
     | 'R' | 'S1' | 'R1' | 'CLK' | 'CU' | 'CD' | 'PV'
     | 'IN' | 'PT' | il_expr_operator

il_expr_operator ::=  'AND' | '&' | 'OR' | 'XOR' | 'ANDN' | '&N' | 'ORN'
     | 'XORN' | 'ADD' | 'SUB' | 'MUL' | 'DIV' | 'MOD' | 'GT' | 'GE' | 'EQ '
     | 'LT' | 'LE' | 'NE'
```

```
il_assign_operator ::= variable_name':='
il_assign_out_operator ::= ['NOT'] variable_name'=>'
il_call_operator ::= 'CAL' | 'CALC' | 'CALCN'
il_return_operator ::= 'RET' | 'RETC' | 'RETCN'
il_jump_operator ::= 'JMP' | 'JMPC' | 'JMPCN'
```

SEMANTIK: Siehe 3.2. Diese Syntax spiegelt nicht die Möglichkeit wider, dass IL Operatoren mit Typen versehen sein können, wie es in Tabelle 52 angegeben ist.

B.3 Sprache ST (Strukturierter Text)

B.3.1 Ausdrücke

PRODUKTIONSREGELN:

```
expression ::= xor_expression {'OR' xor_expression}
xor_expression ::= and_expression {'XOR' and_expression}
and_expression ::= comparison {('&' | 'AND') comparison}
comparison ::= equ_expression { ('=' | '<>') equ_expression}
equ_expression ::= add_expression {comparison_operator add_expression}
comparison_operator ::= '<' | '>' | '<=' | '>=' '
add_expression ::= term {add_operator term}
add_operator ::= '+' | '-'
term ::= power_expression  {multiply_operator power_expression}
multiply_operator ::= '*' | '/' | 'MOD'
power_expression ::= unary_expression {'**' unary_expression}
unary_expression ::= [unary_operator] primary_expression
unary_operator ::= '-' | 'NOT'
primary_expression ::=
     constant | enumerated_value | variable | '(' expression ')'
     | function_name '(' param_assignment {',' param_assignment} ')'
```

SEMANTIK: Diese Definitionen sind so angeordnet, dass sie eine top-down-Ableitung der Struktur des Ausdrucks zeigen. Die Vorrang-Regeln der Operationen werden durch ein „bottom-up“-Lesen der Definitionen der verschiedenen Arten von Ausdrücken deutlich. Eine weitere Beschreibung der Semantik dieser Definitionen sind in 3.3.1 zu finden. Siehe 2.5.1.1 für Einzelheiten der Semantik von Funktionsaufrufen.

B.3.2 Anweisungen

PRODUKTIONSREGELN:

```
statement_list ::= statement ';' {statement ';'}
statement ::= NIL | assignment_statement | subprogram_control_statement
     | selection_statement | iteration_statement
```

SEMANTIK: Siehe 3.3.2.

B.3.2.1 Zuweisungsanweisungen

PRODUKTIONSREGEL:

```
assignment_statement ::= variable ':=' expression
```

SEMANTIK: Siehe 3.3.2.1.

B.3.2.2 Unterprogramm-Steueranweisungen

PRODUKTIONSREGELN:

```
subprogram_control_statement ::= fb_invocation | 'RETURN'

fb_invocation ::= fb_name '(' [param_assignment {',' param_assignment}] ')'

param_assignment ::= ([variable_name ':='] expression)
     | (['NOT'] variable_name '=>' variable)
```

SEMANTIK: Siehe 3.3.2.2.

B.3.2.3 Auswahl-Anweisungen

PRODUKTIONSREGELN:

```
selection_statement ::= if_statement | case_statement

if_statement ::=
     'IF' expression 'THEN' statement_list
       {'ELSIF' expression 'THEN' statement_list}
       ['ELSE' statement_list]
     'END_IF'

case_statement ::=
     'CASE' expression 'OF'
       case_element
       {case_element}
       ['ELSE' statement_list]
     'END_CASE'

case_element ::= case_list ':' statement_list

case_list ::= case_list_element {',' case_list_element}

case_list_element ::= subrange | signed_integer | enumerated_value
```

SEMANTIK: Siehe 3.3.2.3.

B.3.2.4 Wiederholungsanweisungen

PRODUKTIONSREGELN:

```
iteration_statement ::=
     for_statement | while_statement | repeat_statement | exit_statement

for_statement ::=
     'FOR' control_variable ':=' for_list 'DO' statement_list 'END_FOR'

control_variable ::= identifier

for_list ::= expression 'TO' expression ['BY' expression]

while_statement ::= 'WHILE' expression 'DO' statement_list 'END_WHILE'

repeat_statement ::=
     'REPEAT' statement_list 'UNTIL' expression 'END_REPEAT'

exit_statement ::= 'EXIT'
```

SEMANTIK: Siehe 3.3.2.4.

Anhang C
(normativ)

Begrenzungszeichen und Schlüsselwörter

Der Gebrauch der Begrenzungszeichen und Schlüsselwörter in IEC 61131-3 ist in den Tabellen C.1 und C.2 zusammengefasst. Nationale Normungsorganisationen können Übersetzungstabellen für den Text-Anteil der Begrenzungszeichen herausgeben, die in Tabelle C.1 aufgeführt sind und für die die Schlüsselwörter in Tabelle C.2 aufgeführt sind.

Tabelle C.1 – Begrenzungszeichen

Begrenzungszeichen	Abschnitt	Gebrauch
Leer	2.1.4	Wie in 2.1.4. festgelegt
(*	2.1.5	Kommentar-Anfang
*)		Kommentar-Ende
+	2.2.1	Führendes Vorzeichen von Dezimal-Literal
	3.3.1	Additionsoperator
-	2.2.1	Führendes Vorzeichen von Dezimal-Literal
	2.2.3.2	Jahr-Monat-Tag-Trennzeichen
	3.3.1	Subtraktion, Negationsoperator
	4.1.1	Horizontale Line
#	2.2.1	Basiszahl-Trennzeichen
	2.2.3	Zeit-Literal-Trennzeichen
.	2.2.1	Ganzzahl/Bruch-Trennzeichen
	2.4.1.1	Trennzeichen Hierarchische Adresse
	2.4.1.2	Trennzeichen Strukturiertes Element
	2.5.2.1	Trennzeichen Funktionsbaustein-Struktur
e oder E	2.2.1	Begrenzungszeichen Real-Exponent
'	2.2.2	Anfang und Ende von Zeichenfolge
$	2.2.2	Anfang von Sonderzeichen in Folgen
2.2.3 – Zeit-Literal-Begrenzungszeichen einschließlich: T#, D, H, M, S, MS, DATE#, D#, TIME_OF_DAY#, TOD#, tod#, DATE_AND_TIME#, DT#		
:	2.2.3.2	Uhrzeit-Trennzeichen
	2.3.3.1	Trennzeichen Typname-Festlegung
	2.4.2	Trennzeichen Variable/Typ
	2.6.2	Schrittnamen-Trennzeichen
	2.7	Trennzeichen RESOURCE Name/Typ
	2.7	Trennzeichen PROGRAM Name/Typ
	2.7	Trennzeichen Zugriffsname/Pfad/Typ
	3.2.1	Trennzeichen Anweisungsmarke
	4.1.2	Trennzeichen Netzwerk-Marke

Begrenzungszeichen	Abschnitt	Gebrauch
	2.3.3.1	Initialisierungsoperator
:=	2.7.1	Eingangsverbindungsoperator
	3.3.2.1	Anweisungsoperator
( )	2.3.3.1	Begrenzungszeichen Aufzählungsliste
( )	2.3.3.1	Begrenzungszeichen Bereich
[ ]	2.4.1.2	Begrenzungszeichen Feldindex
[ ]	2.4.2	Begrenzungszeichen Folgenlänge
( )	2.4.2	Mehrfach-Initialisierung
( )	3.2.2	Operator Anweisungsliste
( )	3.3.1	Funktionsargumente
( )	3.3.1	Unterausdruck-Hierarchie
( )	3.3.2.2	Begrenzungszeichen Funktionsbaustein-Eingangsliste
,	2.3.3.1	Trennzeichen Aufzählungsliste
	2.3.3.2	Trennzeichen Anfangswert
	2.4.1	Trennzeichen Feldindex
	2.4.2	Trennzeichen Deklarierte Variable
	2.5.2.1	Trennzeichen Funktionsbaustein-Anfangswert
	2.5.2.1	Trennzeichen Funktionsbaustein-Eingangsliste
	3.2.1	Trennzeichen Operandenliste
	3.3.1	Trennzeichen Funktionsargumentliste
	3.3.2.3	Trennzeichen CASE Werteliste
;	2.3.3.1	Trennzeichen Typdeklaration
	3.3	Trennzeichen Anweisung
..	2.3.3.1	Trennzeichen Bereich
	3.3.2.3	Trennzeichen CASE Bereich
%	2.4.1.1	Präfix für Direkt-Darstellung
=>	2.7.1	Ausgangsverbindungsoperator
3.3.1 – Infix-Operatoren einschließlich: ** , NOT, * , / , MOD, + , - , < , > , <= >= , = , <> , &, AND, XOR, OR		
| oder !	4.1.1	Vertikale Linien

Tabelle C.2 – Schlüsselwörter

Schlüsselwörter	Abschnitt
ACTION...END_ACTION	2.6.4.1
ARRAY...OF	2.3.3.1
AT	2.4.3
CASE...OF...ELSE...END_CASE	3.3.2.3
CONFIGURATION...END_CONFIGURATION	2.7.1
CONSTANT	2.4.3
Datentyp-Namen	2.3
EN, ENO	2.5.1.2, 2.5.2.1a)
EXIT	3.3.2.4
FALSE	2.2.1
F_EDGE	2.5.2.2
FOR...TO...BY...DO...END_FOR	3.3.2.4
FUNCTION...END_FUNCTION	2.5.1.3
Funktionsnamen	2.5.1
FUNCTION_BLOCK...END_FUNCTION_BLOCK	2.5.2.2
Funktionsbaustein-Namen	2.5.2
IF...THEN...ELSIF...ELSE...END_IF	3.3.2.3
INITIAL_STEP...END_STEP	2.6.2
NOT, MOD, AND, XOR, OR	3.3.1
PROGRAM...WITH...	2.7.1
PROGRAM...END_PROGRAM	2.5.3
R_EDGE	2.5.2.2
READ_ONLY, READ_WRITE	2.7.1
REPEAT...UNTIL...END_REPEAT	3.3.2.4
RESOURCE...ON...END_RESOURCE	2.7.1
RETAIN, NON_RETAIN	2.4.3
RETURN	3.3.2.2
STEP...END_STEP	2.6.2
STRUCT...END_STRUCT	2.3.3.1
TASK	2.7.2
TRANSITION...FROM...TO...END_TRANSITION	2.6.3
TRUE	2.2.1
TYPE...END_TYPE	2.3.3.1
VAR...END_VAR	2.4.3
VAR_INPUT...END_VAR	2.4.3
VAR_OUTPUT...END_VAR	2.4.3

Schlüsselwörter	Abschnitt
VAR_IN_OUT...END_VAR	2.4.3
VAR_TEMP...END_VAR	2.4.3
VAR_EXTERNAL...END_VAR	2.4.3
VAR_ACCESS...END_VAR	2.7.1
VAR_CONFIG...END_VAR	2.7.1
VAR_GLOBAL...END_VAR	2.7.1
WHILE...DO...END_WHILE	3.3.2.4
WITH	2.7.1

Anhang D
(normativ)

Implementierungsabhängige Parameter

Die implementierungsabhängigen Parameter, die in dieser Norm definiert sind, und der für sie jeweils wichtigste Abschnitt sind in Tabelle D.1 aufgezählt.

ANMERKUNG Weitere implementierungsabhängige Parameter, wie die Fehlerfreiheit, Genauigkeit und Reproduzierbarkeit von Eigenschaften des Zeitverhaltens und der Ausführungs-Steuerung, können signifikante Wirkungen auf die Portierbarkeit von Programmen haben; sie gehören aber nicht zum Geltungsbereich dieses Teils der IEC 61131.

Tabelle D.1 – Implementierungsabhängige Parameter

Abschnitt	Parameter
2.1.2	Maximale Längen von Bezeichnern
2.1.5	Maximale Kommentarlänge
2.1.6	Syntax und Semantik von Pragmas
2.2.2	Syntax und Semantik für den Gebrauch des Doppel-Anführungszeichens, wenn ein bestimmte Implementierung die Eigenschaft 4, aber nicht die Eigenschaft 5 der Tabelle 5 unterstützt
2.3.1	Wertebereich und Genauigkeit der Darstellung von Variablen vom Typ `TIME`, `DATE`, `TIME_OF_DAY` und `DATE_AND_TIME` Genauigkeit der Darstellung von Sekunden in den Typen `TIME_OF_DAY` und `DATE_AND_TIME`
2.3.3.1	Maximale Anzahl von aufgezählten Werten Maximale Anzahl von Feldindizes Maximale Feldgröße Maximale Anzahl der Struktur-Elemente Maximale Struktur-Größe Maximale Anzahl von Variablen pro Deklaration Maximale Anzahl von Ebenen von geschachtelten Strukturen
2.3.3.2	Voreingestellte maximale Länge von `STRING` und `WSTRING` Variablen Maximal zulässige Länge von `STRING` und `WSTRING` Variablen
2.4.1.1	Maximale Anzahl von Hierarchieebenen Logische oder physikalische Abbildung
2.4.2	Initialisierung der Systemeingänge
2.4.3	Maximale Anzahl der Variablen pro Deklaration Wirkung der Anwendung des `AT` Bezeichners in Deklarationen von Funktionsbaustein-Instanzen Warmstart-Verhalten, wenn die Variable weder als `RETAIN` noch als `NON_RETAIN` deklariert ist
2.5	Information zur Bestimmung der Ausführungszeiten von Programm-Organisationseinheiten
2.5.1.2	Werte der Ausgänge, wenn `ENO FALSE` ist

Abschnitt	Parameter
2.5.1.3	Maximale Anzahl der Funktionsspezifikation
2.5.1.5	Maximale Anzahl von Eingängen bei erweiterbaren Funktionen
2.5.1.5.1	Wirkungen der Typkonvertierungen auf die Genauigkeit Fehlerbedingungen während der Typkonvertierungen
2.5.1.5.2	Genauigkeit von numerischen Funktionen
2.5.1.5.6	Wirkungen der Typkonvertierungen zwischen Zeit-Datentypen und anderen Datentypen, die nicht in Tabelle 30 definiert sind
2.5.2	Maximale Anzahl von Funktionsbaustein-Festlegungen und Instanziierungen
2.5.2.1a)	Zuweisung der Funktionsbaustein-Eingangsvariable, wenn `EN FALSE` ist
2.5.2.3.3	`PVmin`, `PVmax` von Zählern
2.5.2.3.4	Wirkung einer Änderung des Werts eines `PT` Eingangs während einer Zeit-Operation
2.5.3	Begrenzungen der Programmgröße
2.6.2	Genauigkeit der verstrichenen Schrittzeit Maximale Anzahl von Schritten pro AS
2.6.3	Maximale Anzahl von Transitionen pro AS und pro Schritt
2.6.4.2	Maximale Anzahl von Aktionsblocks pro Schritt
2.6.4.5	Zugriff auf das funktionale Äquivalent des `Q` oder `A` Ausgangs
2.6.5	Transitionsschaltzeit Maximale Breite der Konstrukte Verzweigung/Zusammenführung
2.7.1	Inhalt der `RESOURCE` Bibliotheken
2.7.1	Wirkung der Verwendung des `READ_WRITE` Zugriffs auf Funktionsbaustein-Ausgänge
2.7.2	Maximale Anzahl von Tasks Auflösung des Task-Intervalls
3.3.1	Maximallänge von Ausdrücken
3.3.2	Maximallänge von Anweisungen
3.3.2.3	Maximale Anzahl von `CASE` Auswahl
3.3.2.4	Wert der Steuervariable bei die Beendigung der `FOR` Schleife
4.1.1	Einschränkungen der Netzwerk-Topologie
4.1.3	Auswertungsreihenfolge von Rückkopplungsschleifen

Anhang E
(normativ)

Fehlerursachen

Die Fehlerursachen, die in dieser Norm definiert sind, und der für sie jeweils wichtigste Abschnitt sind in Tabelle E.1 aufgezählt. Diese Fehler können während der Aufbereitung des Programms für die Ausführung oder während der Ausführung des Programms entdeckt werden. Der Hersteller muss die Behandlung dieser Fehler nach den Bestimmungen von Abschnitt 1.5.1 dieses Teils der IEC 61131 festlegen.

Tabelle E.1 – Fehlerursachen

Abschnitt	Fehlerursachen
2.1.5	Geschachtelte Kommentare
2.3.3.1	Aufgezählter Wert ist mehrdeutig
2.3.3.1	Wert einer Variable überschreitet den festgelegten Bereich (Subrange)
2.4.1.1	Fehlende Konfiguration einer unvollständigen Adress-Festlegung (`"*"` Angabe)
2.4.3	Versuch einer Programm-Organisationseinheit, eine Variable zu verändern, die als `CONSTANT` deklariert ist
2.4.3	Deklaration einer Variable `VAR_GLOBAL CONSTANT` in einem enthaltenen Element, das ein Element enthält, in der dieselbe Variable als `VAR_EXTERNAL` ohne das Bestimmungszeichen `CONSTANT` deklariert ist
2.5.1	Inkorrekter Gebrauch von direkt dargestellten oder externen Variablen in Funktionen
2.5.1.1	Eine `VAR_IN_OUT` Variable ist nicht „korrekt abgebildet“
2.5.1.1	Mehrdeutiger Wert verursacht durch eine `VAR_IN_OUT` Verbindung
2.5.1.5.1	Typwandlungsfehler
2.5.1.5.2	Numerisches Ergebnis überschreitet den Bereich des Datentyps Division durch Null
2.5.1.5.3	`N` Eingang ist kleiner Null bei einer Bit-Schiebe-Funktion
2.5.1.5.4	Gemischte Eingangsdatentypen bei einer Auswahlfunktion Selektor (`K`) außerhalb des Bereichs der `MUX` Funktion
2.5.1.5.5	Ungültige festgelegte Zeichenposition Ergebnis überschreitet maximale Länge der Zeichenfolge `ANY_INT` Eingang ist kleiner Null bei einer Zeichenfolge-Funktion
2.5.1.5.6	Ergebnis überschreitet den Bereich für Datentyp
2.5.2.2	Für eine Funktionsbaustein-Instanz, die als Eingangsvariable verwendet wird, ist kein Wert festgelegt
2.5.2.2	Für eine Ein-Aus-Variable ist kein Wert festgelegt
2.6.2	Null oder mehr als ein Anfangsschritt in AS-Netzwerk Anwenderprogramm versucht, den Schritt-Status oder die -Zeit zu verändern
2.6.3	Seiteneffekte bei der Auswertung der Transitionsbedingung
2.6.4.5	Fehler bei der Aktionssteuerung

Abschnitt	Fehlerursachen
2.6.5	Gleichzeitig erfüllte nicht-priorisierte Transitionen in einer Auswahl-Verzweigung Unsichere oder nicht-erreichbare AS
2.7.1	Datentypkonflikt in `VAR_ACCESS`
2.7.2	Eine Task wird nicht aufgerufen oder überschreitet die Ausführungsgrenzzeit
3.2.2	Numerisches Ergebnis überschreitet den Bereich für Datentyp Aktuelles Ergebnis und Operand haben nicht denselben Datentyp
3.3.1	Division durch Null Numerisches Ergebnis überschreitet den Bereich für Datentyp Ungültiger Datentyp für Operation
3.3.2.1	Rücksprung von Funktion ohne zugewiesenen Wert
3.3.2.4	Wiederholung erreicht kein Ende
4.1.1	Gleicher Bezeichner als Konnektor-Marke und Elementname benutzt
4.1.3	Nicht-initialisierte Rückkopplungsvariable

Anhang F
(informativ)

Beispiele

F.1 Funktion WIEGEN

Die Beispielfunktion WIEGEN (WEIGH) beinhaltet die Wandlung einer Eingabe von einer Waage, wobei das Bruttogewicht von BCD in binär gewandelt wird, die binäre ganzzahlige Subtraktion des Tara-Gewichts, das vorher gewandelt und im SPS-Speicher abgelegt ist und die Wandlung des resultierenden Nettogewichts zurück in die BCD-Form, z. B. für eine Ausgabe auf einer Anzeige. Der „EN“-Eingang wird benutzt, um anzuzeigen, dass die Waage bereit ist, die Wägeaufgaben durchzuführen.

Der „ENO“-Ausgang zeigt an, dass ein entsprechender Auftrag vorliegt (z. B. von einem Bedienschalter), dass die Waage in Bereitstellung ist, um das Gewicht abzulesen und dass die jeweilige Funktion ein korrektes Ergebnis liefert.

Die Text-Form der Deklaration dieser Funktion lautet:

```
FUNCTION WEIGH : WORD     (* BCD kodiert *)
  VAR_INPUT  (* "EN"-Eing. wird benutzt, um "Waage bereit" anzuzeigen*)
    weigh_command : BOOL;
    gross_weight : WORD ; (* BCD kodiert*)
    tare_weight : INT ;
  END_VAR
(* Funktionsrumpf *)
END_FUNCTION               (* Implizit "ENO" *)
```

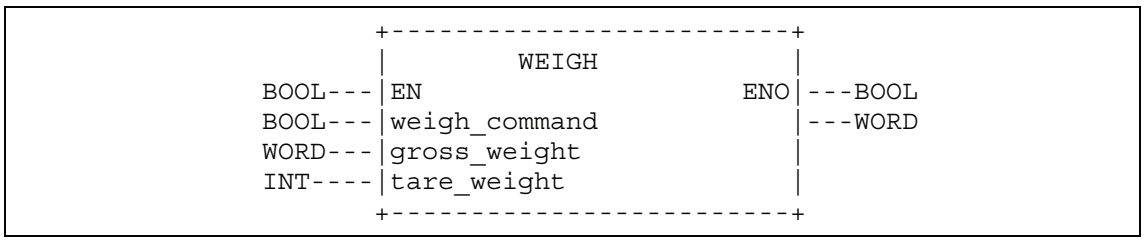
Der Rumpf der Funktion WIEGEN in der AWL Sprache:

```
            LD          weigh_command
            JMPC        WEIGH_NOW
            ST          ENO             (* nicht Wiegen, 0 nach "ENO" *)
            RET
WEIGH_NOW:  LD          gross_weight
            BCD_TO_INT
            SUB         tare_weight
            INT_TO_BCD                  (* Rückgabe berechn. Gewicht *)
            ST          WEIGH
```

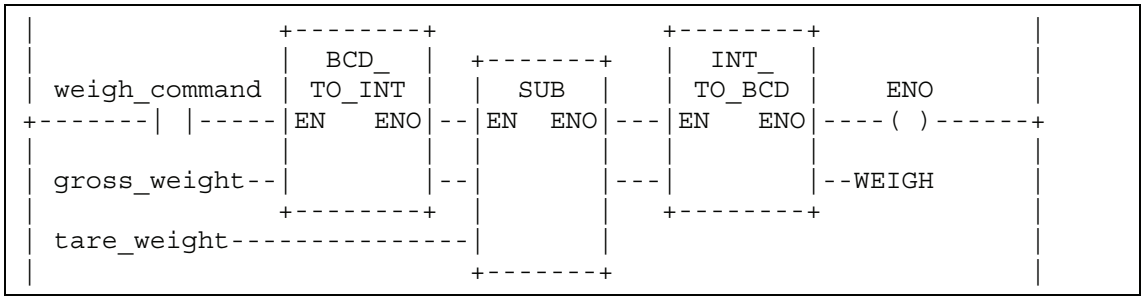
Der Rumpf der Funktion WIEGEN in der ST Sprache:

```
IF weigh_command THEN
  WEIGH := INT_TO_BCD (BCD_TO_INT(gross_weight) - tare_weight);
END_IF ;
```

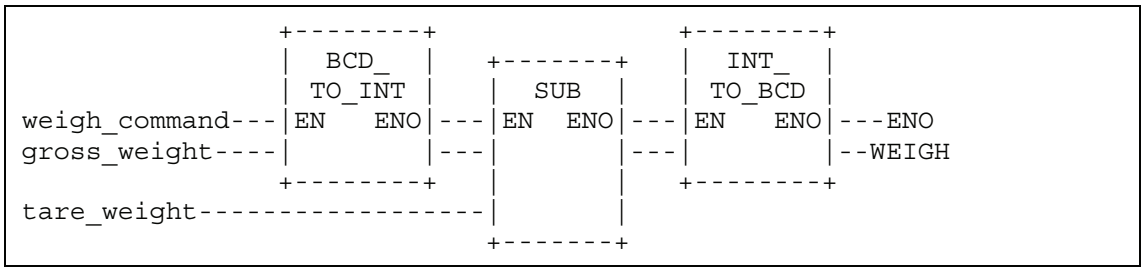
Eine äquivalente grafische Deklaration der Funktion WIEGEN:



Der Funktionsrumpf in der KOP Sprache:



Der Funktionsrumpf in der FBS Sprache lautet:



F.2 Funktionsbaustein KOMMANDO_ÜBERWACHUNG

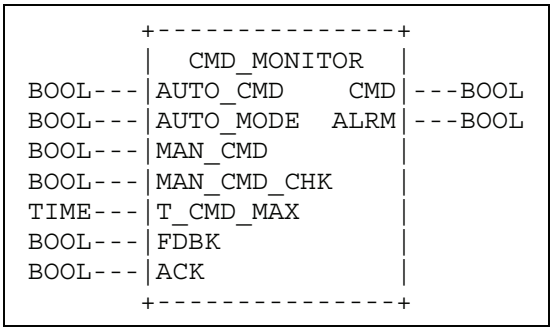
Das Beispiel des Funktionsbausteins KOMMANDO_ÜBERWACHUNG (CMD_MONITOR) veranschaulicht die Steuerung einer Einheit, die in der Lage ist, auf ein boolesches Kommando (dem CMD Ausgang) zu antworten und ein boolesches Rückkoppelsignal (den FDBK Eingang) zurückzugeben, um die erfolgreiche Vollendung der gewünschten Aktion anzuzeigen. Der Funktionsbaustein bietet eine manuelle Steuerung über den Eingang MAN_CMD oder eine automatische Steuerung über den Eingang AUTO_CMD in Abhängigkeit vom Zustand des Eingangs AUTO_MOD (0 bzw. 1). Eine Prüfung des Eingangs MAN_CMD wird mittels des Eingangs MAN_CMD_CHK durchgeführt, der 0 sein muss, um den Eingang MAN_CMD freizugeben.

Wenn die Bestätigung der Kommandoausführung nicht innerhalb einer mit T_CMD_MAX vorbestimmten Zeit am Eingang FDBK eintrifft, wird das Kommando zurückgenommen und ein Alarm am ALRM Ausgang signalisiert. Der Alarm kann durch den ACK Eingang (Bestätigung) gelöscht werden, womit der weitere Ablauf des Kommandozyklus freigegeben wird.

Die Text-Form der Deklaration des Funktionsbausteins KOMMANDO_ÜBERWACHUNG lautet:

```
FUNCTION_BLOCK CMD_MONITOR
 VAR_INPUT AUTO_CMD : BOOL ;(* Automatisches Kommando *)
     AUTO_MODE : BOOL ;      (* AUTO_CMD freigeben     *)
     MAN_CMD : BOOL ;        (* Manuelles Kommando     *)
     MAN_CMD_CHK : BOOL ;    (* Negiertes MAN_CMD zum Entprellen  *)
     T_CMD_MAX : TIME ;      (* Max. Zeit von CMD bis FDBK        *)
     FDBK : BOOL ;           (* Bestätigung der CMD Ausführung durch Gerät *)
     ACK : BOOL ;            (* Quittung/Rücknahme ALRM       *)
 END_VAR

 VAR_OUTPUT CMD : BOOL ;     (* Kommando an Gerät               *)
           ALRM : BOOL ;     (* T_CMD_MAX verstrichen ohne FDBK *)
 END_VAR

 VAR CMD_TMR : TON ;(* CMD bis FDBK Zeitglied *)
     ALRM_FF : SR ; (* Vorrangiger S-Eingang *)
 END_VAR     (* Kommando muss zurückgenommen werden bevor ACK" den Alarm
löschen kann *)

 (* Funktionsbaustein-Rumpf *)

END_FUNCTION_BLOCK
```

Eine äquivalente grafische Deklaration lautet:



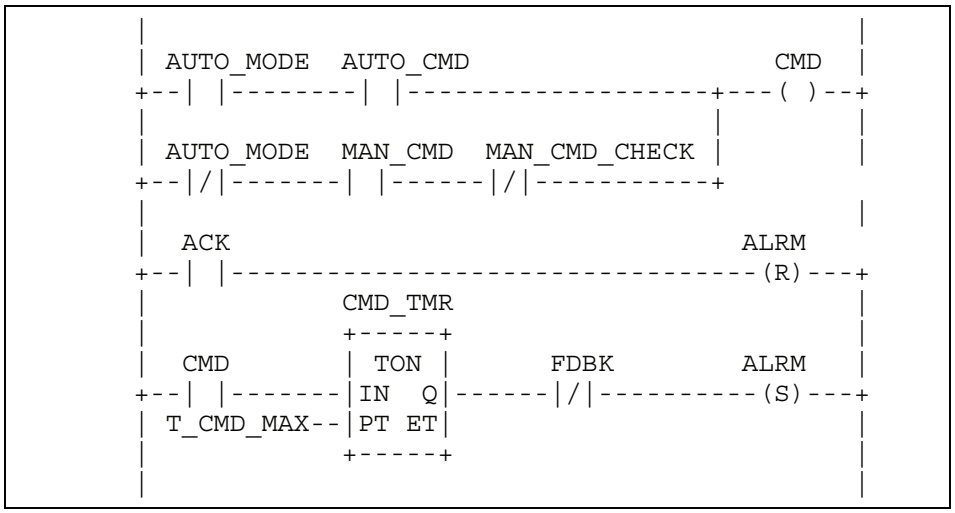
Der Rumpf des Funktionsbausteins KOMANDO_ÜBERWACHUNG in der ST Sprache lautet:

```
CMD := AUTO_CMD & AUTO_MODE
       OR MAN_CMD & NOT MAN_CMD_CHK & NOT AUTO_MODE ;
CMD_TMR (IN := CMD, PT := T_CMD_MAX);
ALRM_FF (S1 := CMD_TMR.Q & NOT FDBK, R := ACK);
ALRM := ALRM_FF.Q1;
```

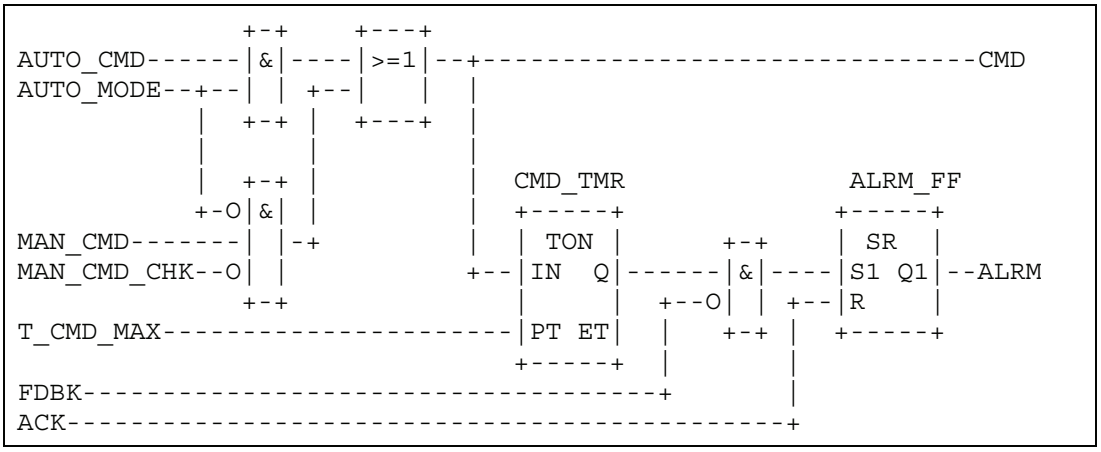
Der Rumpf des Funktionsbausteins KOMMANDO_ÜBERWACHUNG in der AWL Sprache lautet:

```
LD        T_CMD_MAX
ST        CMD_TMR.PT          (* Speichern einer Eingabe des FB TON *)
LD        AUTO_CMD
AND       AUTO_MODE
OR(       MAN_CMD
ANDN      AUTO_MODE
ANDN      MAN_CMD_CHK
)
ST        CMD
IN        CMD_TMR             (* Aufrufen des FB TON *)
LD        CMD_TMR.Q
ANDN      FDBK
ST        ALRM_FF.S1          (* Speichern einer Eingabe des FB SR *)
LD        ACK
R         ALRM_FF             (* Aufrufen des FB SR *)
LD        ALRM_FF.Q1
ST        ALRM
```

Der Rumpf des Funktionsbausteins KOMMANDO_ÜBERWACHUNG in der KOP Sprache lautet:



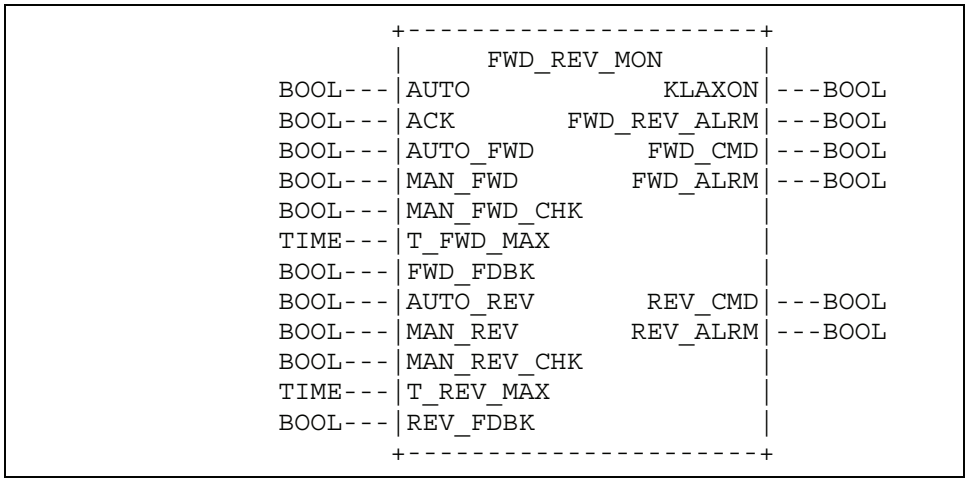
Der Rumpf des Funktionsbausteins KOMMANDO_ÜBERWACHUNG in der FBS Sprache lautet:



F.3 Funktionsbaustein VORW_RÜCKW_ÜBERWACHUNG

Das Beispiel des Funktionsbausteins VOR_RÜCK_ÜBERWACHUNG (FWD_REV_MON) zeigt die Steuerung einer Einheit, die eine Zwei-Weg-Positionierung durchführen kann; z. B ein motorgetriebenes Ventil. Sowohl automatischer als auch handgesteuerter Modus ist möglich, dabei ist für jede Bewegungsrichtung eine Meldemöglichkeit gegeben, wie dies für den oben stehenden Funktionsbaustein CMD_MONITOR beschrieben ist. Zusätzlich bewirkt ein Konflikt zwischen Vorwärts- und Rückwärtskommandos das Zurücknehmen von beiden Kommandos und die Anzeige eines Alarmfalles. Das boolesche ODER aller Alarmfälle wird als Hupen-Ausgang (KLAXON) zur Verfügung gestellt, um dem Bediener eine Meldung anzuzeigen.

Die grafische Form der Deklaration dieses Funktionsbausteins lautet:



Eine Textform der Deklaration des Funktionsbausteins VORW_RÜCKW_ÜBERWACHUNG lautet:

```
FUNCTION_BLOCK FWD_REV_MON
VAR_INPUT AUTO : BOOL ;(* Freigeben des automatischen Kommandos  *)
  ACK : BOOL ;          (* Quittierung/Löschen aller Alarme       *)
  AUTO_FWD : BOOL ;     (* Automatisches Vorwärtskommando         *)
  MAN_FWD : BOOL ;      (* Manuelles Vorwärtskommando             *)
  MAN_FWD_CHK : BOOL ; (* Negiertes MAN_FWD zum Entprellen       *)
  T_FWD_MAX : TIME ;    (* Max. Zeit von FWD_CMD bis FWD_FDBK     *)
  FWD_FDBK : BOOL ;     (* Bestätigung der FWD_CMD Ausführung     *)
                        (*                      durch das Gerät   *)
  AUTO_REV : BOOL ;     (* Automatisches Rückwärts-Kommando       *)
  MAN_REV :  BOOL ;     (* Manuelles Rückwärts-Kommando           *)
  MAN_REV_CHK : BOOL ; (* Negiertes MAN_REV zum Entprellen       *)
  T_REV_MAX : TIME ;    (* Max. Zeit von REV_CMD bis REV_FDBK     *)
  REV_FDBK :  BOOL ;    (* Bestätigung der REV_CMD Ausführung     *)
END_VAR                 (* durch das Gerät                        *)
VAR_OUTPUT KLAXON : BOOL ;      (* Irgendein Alarm aktiv          *)
  FWD_REV_ALRM : BOOL; (* Vorw./Rückw. Kommando-Konflikt         *)
  FWD_CMD :  BOOL ;     (* "Vorwärts"-Kommando an Gerät           *)
  FWD_ALRM : BOOL ;     (* T_FWD_MAX abgelaufen ohne FWD_FDBK     *)
  REV_CMD :  BOOL ;     (* "Rückwärts"-Kommando an Gerät          *)
  REV_ALRM : BOOL ;     (* T_REV_MAX abgelaufen ohne REV_FDBK     *)
END_VAR
VAR FWD_MON : CMD_MONITOR; (* "Vorwärts"-Kommando Überwachung  *)
  REV_MON : CMD_MONITOR;   (* "Rückwärts"-Kommando Überwachung *)
  FWD_REV_FF : SR ;        (* Vorw./Rückw.-Überlappung Merker  *)
END_VAR
  (* Funktionsbaustein-Rumpf *)
END_FUNCTION_BLOCK
```

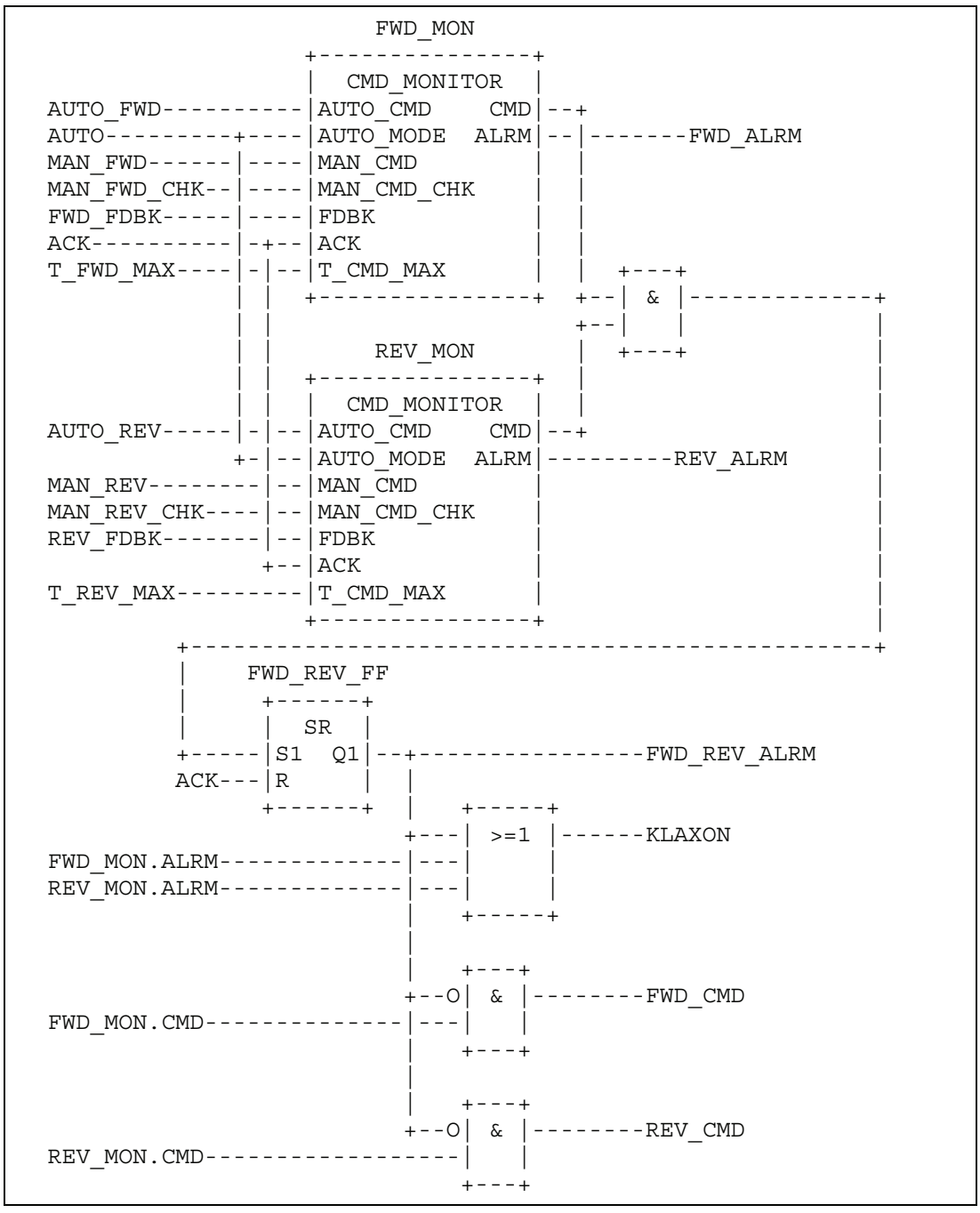
Der Rumpf des Funktionsbaustein VORW_RÜCKW_ÜBERWACHUNG kann in der ST Sprache folgendermaßen geschrieben werden:

```
(* Auswerten interner Funktionsbausteine *)
  FWD_MON    (AUTO_MODE   := AUTO,
              ACK         := ACK,
              AUTO_CMD    := AUTO_FWD,
              MAN_CMD     := MAN_FWD,
              MAN_CMD_CHK := MAN_FWD_CHK,
              T_CMD_MAX   := T_FWD_MAX,
              FDBK        := FWD_FDBK);
  REV_MON    (AUTO_MODE   := AUTO,
              ACK         := ACK,
              AUTO_CMD    := AUTO_REV,
              MAN_CMD     := MAN_REV,
              MAN_CMD_CHK := MAN_REV_CHK,
              T_CMD_MAX   := T_REV_MAX,
              FDBK        := REV_FDBK);
  FWD_REV_FF (S1 := FWD_MON.CMD & REV_MON.CMD, R := ACK);
(* Transfer der Daten zu den Ausgängen *)
  FWD_REV_ALRM := FWD_REV_FF.Q1;
  FWD_CMD := FWD_MON.CMD & NOT FWD_REV_ALRM;
  FWD_ALRM := FWD_MON.ALRM;
  REV_CMD := REV_MON.CMD & NOT FWD_REV_ALRM;
  REV_ALRM := REV_MON.ALRM;
  KLAXON := FWD_ALRM OR REV_ALRM OR FWD_REV_ALRM;
```

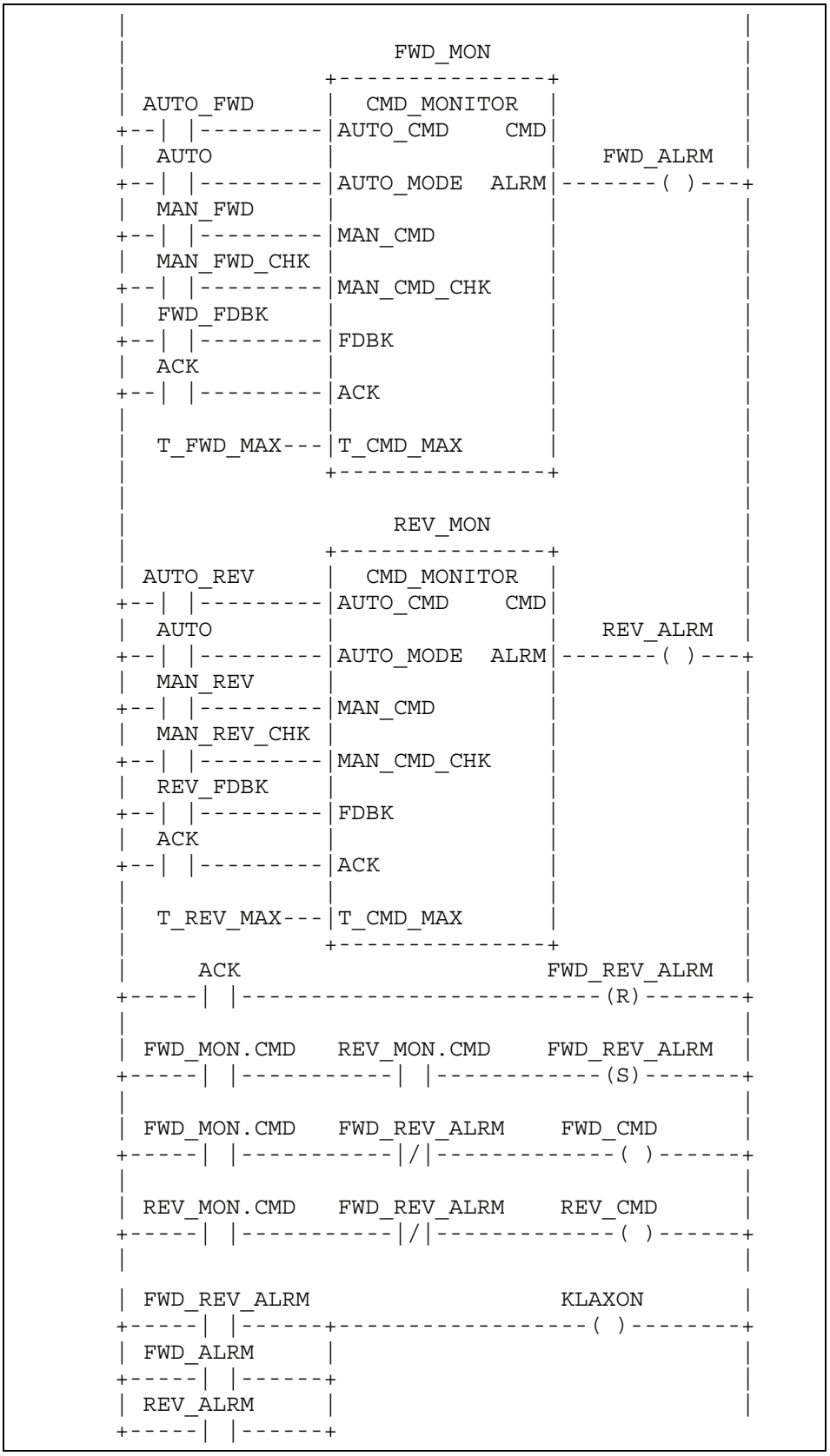
Der Rumpf des Funktionsbaustein VORW_RÜCKW_ÜBERWACHUNG in der AWL Sprache lautet:

```
(* Auswerten der internen Funktionsbausteine *)
CAL   FWD_MON(
      AUTO_MODE:= AUTO,
      ACK:= ACK,
      AUTO_CMD:= AUTO_FWD,
      MAN_CMD:= MAN_FWD,
      MAN_CMD_CHK:= MAN_FWD_CHK,
      T_CMD_MAX:= T_FWD_MAX,
      FDBK:= FWD_FDBK
)
CAL   REV_MON(
      AUTO_MODE:= AUTO,
      ACK:= ACK,
      AUTO_CMD:= AUTO_REV,
      MAN_CMD:= MAN_REV,
      MAN_CMD_CHK:= MAN_REV_CHK,
      T_CMD_MAX:= T_REV_MAX,
      FDBK:= REV_FDBK
)
CAL   FWD_REV_FF(
      S1:=(
         LD FWD_MON.CMD
         AND REV_MON.CMD
         ),
      R:= ACK,
      Q => FWD_REV_ALRM   (* Konflikt Alarm *)
)
(* Daten an die Ausgänge transferieren *)
LD    FWD_MON.CMD (* „Vorwärts“ Befehl und Alarm *)
ANDN  FWD_REV_ALRM
ST    FWD_CMD
LD    FWD_MON.ALRM
ST    FWD_ALRM
LD    REV_MON.CMD  (*„Rückwärts“ Befehl und Alarm *)
ANDN  FWD_REV_ALRM
ST    REV_CMD
LD    REV_MON.ALRM
ST    REV_ALRM
OR    FWD_ALRM  (* OR von allen Alarme *)
OR    FWD_REV_ALRM
ST    KLAXON
```

Der Rumpf des Funktionsbaustein VORW_RÜCKW_ÜBERWACHUNG in der FBS Sprache lautet:



Der Rumpf des Funktionsbaustein VORW_RÜCKW_ÜBERWACHUNG in der KOP Sprache lautet:



F.4 Funktionsbaustein STACK_INT

Dieser Funktionsbaustein bildet ein Stack („Stapelspeicher“) für maximal 128 ganze Zahlen (integer). Die üblichen Stack-Operationen PUSH („einkellern“) und POP („auskellern“) werden durch flankengesteuerte boolesche Eingänge zur Verfügung gestellt. Es ist ein Eingang mit vorrangigem Rücksetzen (R1) vorhanden; die maximale Stack-Tiefe (N) wird zum Zeitpunkt des Rücksetzens festgelegt. Zusätzlich zum oberen Wert des Stacks (OUT) gibt es boolesche Ausgänge, die die Zustände „Stack leer“ und „Stack Überlauf“ anzeigen.

Die Text-Form der Deklaration dieses Funktionsbausteins lautet:

```
FUNCTION_BLOCK STACK_INT
  VAR_INPUT PUSH, POP: BOOL R_EDGE;    (* Stack-Grundoperationen *)
            R1 : BOOL ;              (* Vorrangiges Rücksetzen    *)
            IN : INT ;               (* Eingabe                   *)
            N  : INT ;               (* Maximale Tiefe nach Rücks.*)
  END_VAR
  VAR_OUTPUT EMPTY : BOOL := 1 ;       (* Stack leer     *)
             OFLO  : BOOL := 0 ;       (* Stack-Überlauf *)
             OUT   : INT  := 0 ;       (* Ausgabe        *)
  END_VAR
  VAR STK : ARRAY[0..127] OF INT;   (* Interner Stack             *)
      NI : INT :=128  ;          (* Speicher für N nach Rücks. *)
      PTR : INT := -1 ;          (* Stack-Zeiger               *)
  END_VAR
      (* Funktionsbaustein-Rumpf *)
END_FUNCTION_BLOCK
```

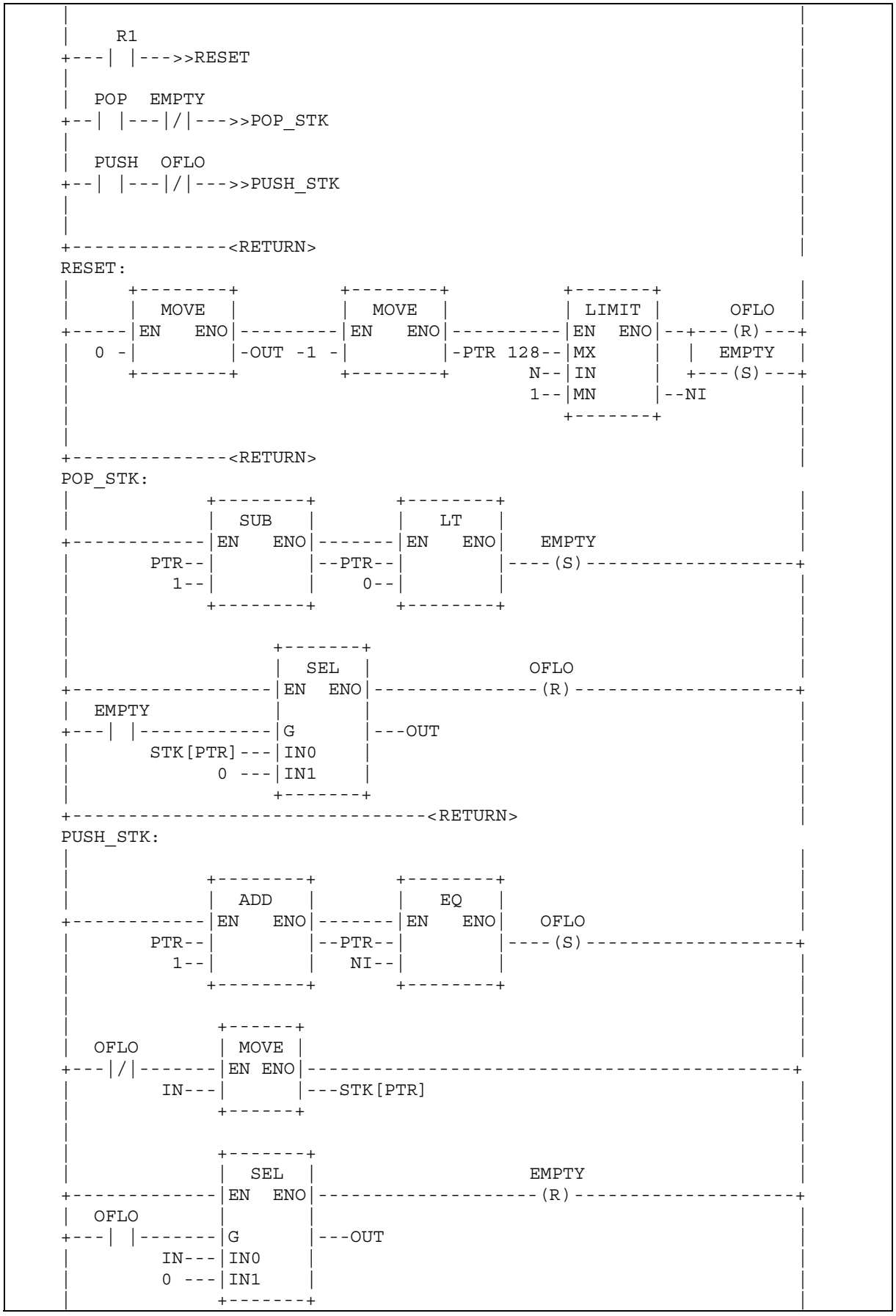
Eine grafische Deklaration des Funktionsbausteins STACK_INT lautet:

```
                        +------------+
                        | STACK_INT  |
                  BOOL--->PUSH  EMPTY|---BOOL
                  BOOL--->POP    OFLO|---BOOL
                  BOOL---|R1       OUT|---INT
                  INT----|IN          |
                  INT----|N           |
                        +------------+

(* Interne Variablen-Deklarationen *)
  VAR STK : ARRAY[0..127] OF INT;   (* Interner Stack             *)
      NI : INT :=128  ;             (* Speicher für N nach Rücks. *)
      PTR : INT := -1 ;             (* Stack-Zeiger               *)
  END_VAR
```

Der Funktionsbaustein-Rumpf in der Sprache ST lautet:

```
   IF R1 THEN
      OFLO := 0; EMPTY := 1; PTR := -1;
      NI := LIMIT (MN:=1,IN:=N,MX:=128); OUT := 0;
   ELSIF POP & NOT EMPTY THEN
      OFLO := 0; PTR := PTR-1; EMPTY := PTR < 0;
      IF EMPTY THEN OUT := 0;
      ELSE OUT := STK[PTR];
      END_IF ;
   ELSIF PUSH & NOT OFLO THEN
      EMPTY := 0; PTR := PTR+1; OFLO := (PTR = NI);
      IF NOT OFLO THEN OUT := IN ; STK[PTR] := IN;
      ELSE OUT := 0;
      END_IF ;
   END_IF ;
```

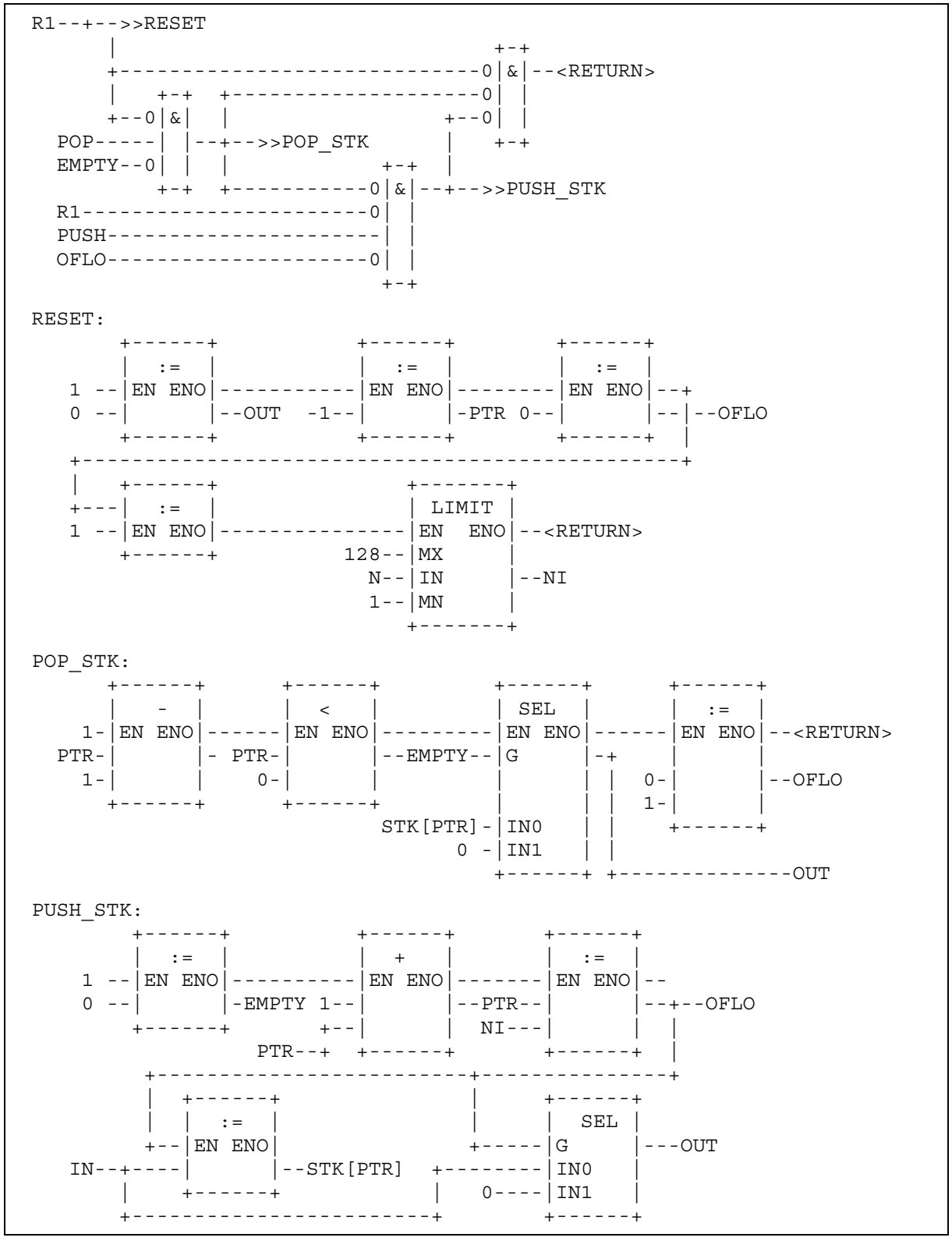
Der Funktionsbaustein-Rumpf in der Sprache KOP lautet:



Der Rumpf des Funktionsbaustein STACK_INT in der Sprache AWL lautet:

```
             LD       R1            (* Führt Operationen aus *)
             JMPC     RESET
             LD       POP
             ANDN     EMPTY         (* Kein Pop des leeren Stack *)
             JMPC     POP_STK
             LD       PUSH
             ANDN     OFLO          (* Kein Pusch des übergelaufenen Stack *)
             JMPC     PUSH_STK
             RET                    (* Zurück, wenn keine Operationen aktiv
                                    *)
RESET:       LD       0             (* Stack Rücksetzen *)
             ST       OFLO
             LD       1
             ST       EMPTY
             LD       -1
             ST       PTR
             LD       1
             LIMIT    N, 128
             ST       NI
             JMP      ZRO_OUT
POP_STK:     LD       0
             ST       OFLO          (* „popped" Stack hat keinen Überlauf *)
             LD       PTR
             SUB      1
             ST       PTR
             LT       0             (* Leer, wenn  PTR < 0 *)
             ST       EMPTY
             JMPC     ZRO_OUT
             LD       STK[PTR]
             JMP      SET_OUT
PUSH_STK:    LD       0
             ST       EMPTY         (* „pusched" Stack ist nicht leer *)
             LD       PTR
             ADD      1
             ST       PTR
             EQ       NI            (* Überlauf wenn PTR = NI *)
             ST       OFLO
             JMPC     ZRO_OUT
             LD       IN
             ST       STK[PTR]      (* Push IN in STK *)
             JMP      SET_OUT
ZRO_OUT:     LD       0             (* OUT=0 für EMPTY oder OFLO *)
SET_OUT:     ST       OUT
```

Der Funktionsbaustein `STACK_INT` in der Sprache FBS:



F.5 Funktionsbaustein MIX_2_ZIEGEL

Der Funktionsbaustein MIX_2_ZIEGEL (`MIX_2_BRIX`) steuert das Mischen von zwei Ziegeln aus festem Material, die einzeln über ein Transportband befördert werden, mit den abgewogenen Mengen von zwei flüssigen Stoffen A und B. Dies ist in Bild F.1 gezeigt. Ein „Start“ Kommando (`ST`), das manuell oder automatisch sein kann, leitet einen Mess- und Mischzyklus ein; dabei wird folgendermaßen mit dem gleichzeitigen Wiegen und Ziegeltransport begonnen:

– Flüssigkeit A wird bis zur Marke „a“ der Wiegeeinrichtung gewogen, dann wird Flüssigkeit B bis zur Marke „b“ gewogen; darauf folgt das Füllen des Mischers aus der Wiegeeinrichtung C;

– zwei Ziegel werden vom Band in den Mischer transportiert.

Der Zyklus endet mit dem Drehen des Mischers und dann seinem Kippen nach einer vorbestimmten Zeit `"t1"`. Das Drehen des Mischers läuft während des Entleerens weiter.

Die Anzeige „`WC`“ (Waage C) wird als vierzifferige BCD-Zahl angegeben und in den Typ `INT` für die internen Operationen gewandelt. Es wird angenommen, dass das Tara-Gewicht „z“ (Leergewicht) bereits vorher bestimmt wurde.

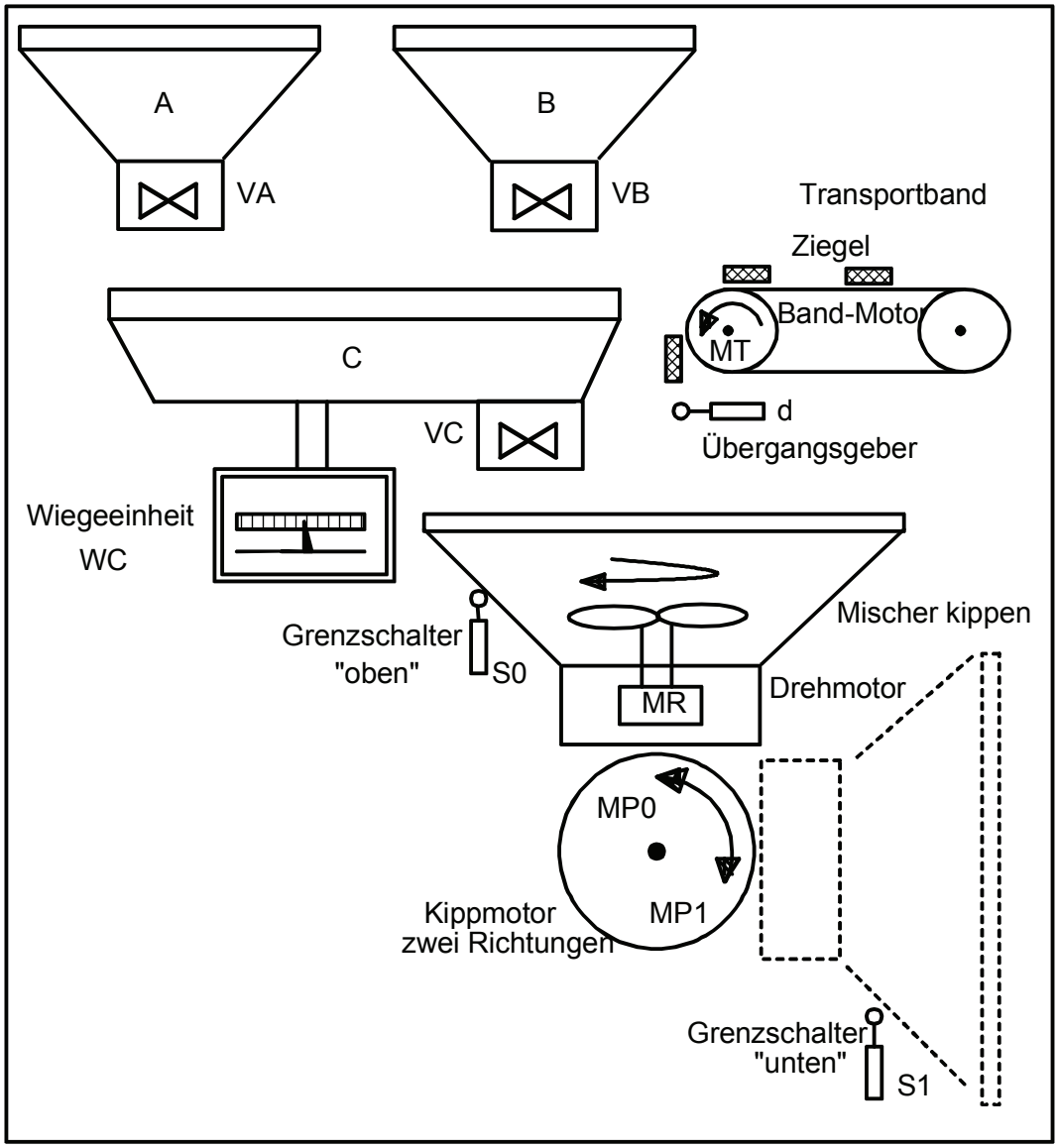
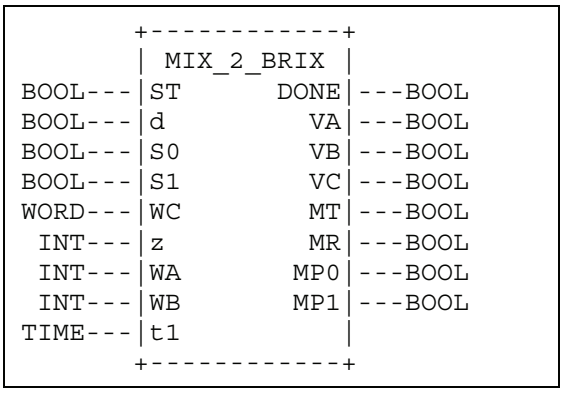


Bild F.1 – Funktionsbaustein MIX_2_ZIEGEL – Physikalisches Modell

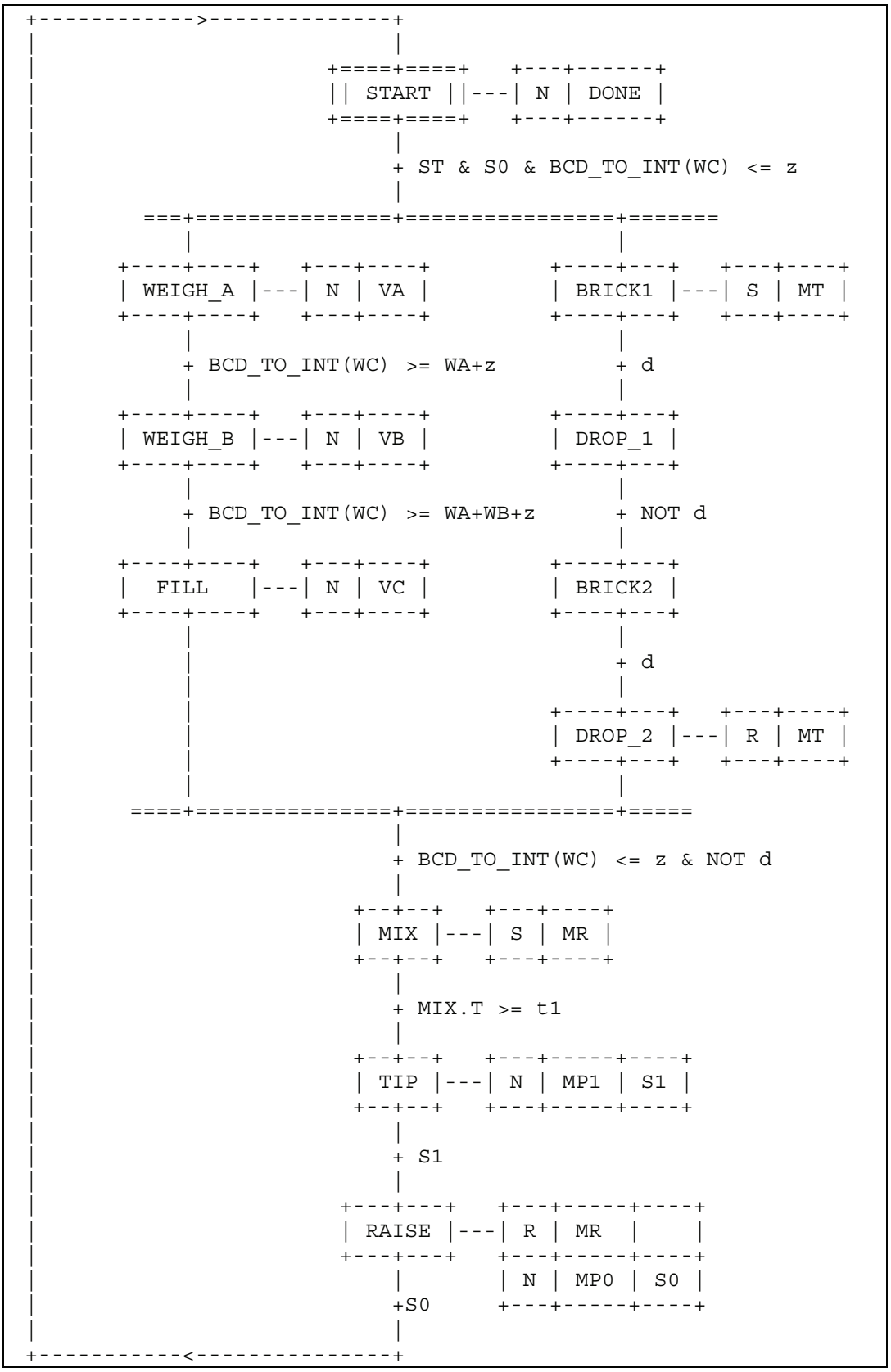
Die Text-Form der Deklaration dieses Funktionsbausteins lautet:

```
FUNCTION_BLOCK MIX_2_BRIX
       VAR_INPUT
  ST : BOOL ;       (* "Start" Befehl                *)
  d  : BOOL ;       (* Übergangsgeber                *)
  S0 : BOOL ;       (* Grenzschalter "Mischer oben"  *)
  S1 : BOOL ;       (* Grenzschalter "Mischer unten" *)
  WC : WORD;        (* Aktuelle Anzeige in BCD       *)
  z  : INT ;        (* Tara-(Leer) Gewicht           *)
  WA : INT ;        (* Gewünschtes Gewicht von A     *)
  WB : INT ;        (* Gewünschtes Gewicht von B     *)
  t1 : TIME ;       (* Mischzeit                     *)
VAR_END
VAR_OUTPUT
  DONE ,
  VA   ,        (* Ventil "A": 0 - geschlossen, 1 - offen *)
  VB   ,        (* Ventil "B": 0 - geschlossen, 1 - offen *)
  VC   ,        (* Ventil "C": 0 - geschlossen, 1 - offen *)
  MT   ,        (* Motor Transportband                    *)
  MR   ,        (* Motor Mischer drehen                   *)
  MP0  ,        (* Kippbefehl Motor "aufwärts"            *)
  MP1  : BOOL; (* Kippbefehl Motor "abwärts"              *)
END_VAR
(* Funktionsbaustein-Rumpf *)
END_FUNCTION_BLOCK
```

Eine grafische Deklaration lautet:



Der Rumpf des Funktionsbaustein MIX_2_Ziegel in der FBS Sprache mit Transitionsbedingungen in ST Sprache ist unten gezeigt:



Der Rumpf von des Funktionsbausteins MIX_2_ZIEGEL in einer AS Text-Darstellung mit ST Sprachelementen lautet:

```
INITIAL_STEP START: DONE(N); END_STEP

TRANSITION FROM START TO (WEIGH_A, BRICK1)
   := ST & S0 & BCD_TO_INT(WC) <= z;
END_TRANSITION

STEP WEIGH_A: VA(N); END_STEP
TRANSITION FROM WEIGH_A TO WEIGH_B := BCD_TO_INT(WC) >= WA+z ;
END_TRANSITION

STEP WEIGH_B: VB(N); END_STEP
TRANSITION FROM WEIGH_B TO FILL := BCD_TO_INT(WC) >= WA+WB+z ;
END_TRANSITION

STEP FILL: VC(N); END_STEP
STEP BRICK1: MT(S); END_STEP
TRANSITION FROM BRICK1 TO DROP_1 := d ; END_TRANSITION

STEP DROP_1: END_STEP
TRANSITION FROM DROP_1 TO BRICK2 := NOT d ; END_TRANSITION

STEP BRICK2: END_STEP
TRANSITION FROM BRICK2 TO DROP_2:= d ; END_TRANSITION

STEP DROP_1: MT(R); END_STEP

TRANSITION FROM (FILL,DROP_2) TO MIX
    := BCD_TO_INT(WC) <= z & NOT d ;
END_TRANSITION

STEP MIX: MR(S); END_STEP
TRANSITION FROM MIX TO TIP MIX.T >= t1 ; END_TRANSITION

STEP TIP: MP1(N); END_STEP
TRANSITION FROM TIP TO RAISE := S1 ; END_TRANSITION

STEP RAISE: MR(R); MP0(N); END_STEP
TRANSITION FROM RAISE TO START := S0 ; END_TRANSITION
```

F.6 Analogsignal-Verarbeitung

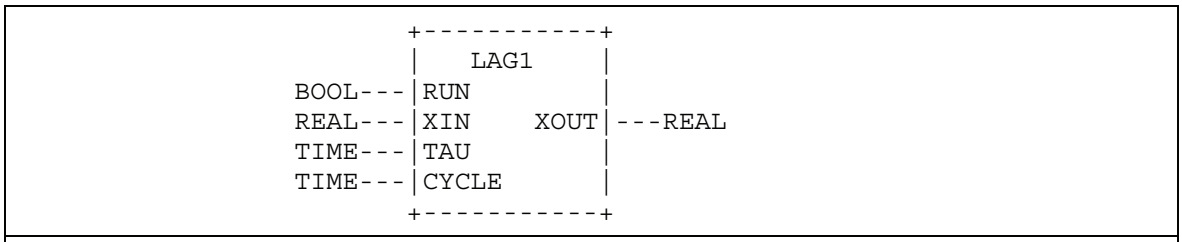
Dieser Teil des Anhangs soll veranschaulichen, wie die Programmiersprachen dieser Norm angewendet werden können, um die Grundfunktionen Messen, Steuern und Regeln der prozessrechnergestützten Automatisierung zu realisieren. Die unten gezeigten Bausteine sind nicht auf Analogsignale beschränkt; sie dürfen angewendet werden, um alle Variablen mit den geeigneten Typen zu verarbeiten. Gleichermaßen können weitere Funktionen und Funktionsbausteine aus dieser Norm (z. B. mathematische Funktionen) benutzt werden, um die Variablen zu verarbeiten, die als Analogsignale an den E/A-Klemmen der SPS erscheinen können.

Diese Funktionsbausteine dürfen Typangaben bei den Eingangs- und Ausgangsvariablen haben, die unten als `REAL` (z. B. `XIN`, `XOUT`) gezeigt sind; dabei ist der geeignete Datentypname anzuhängen, z. B. `LAG1_LREAL`. Der voreingestellte Datentyp für diese Variablen ist `REAL`.

Diese Beispiele sind nur zum Zweck der Veranschaulichung angegeben. Hersteller dürfen verschiedenartige Implementierungen von Signal-Verarbeitungselementen haben. Die Angabe dieser Beispiele hat nicht die Absicht, der Normung solcher Elemente durch die entsprechenden Gremien vorzugreifen.

F.6.1 Funktionsbaustein FILT1

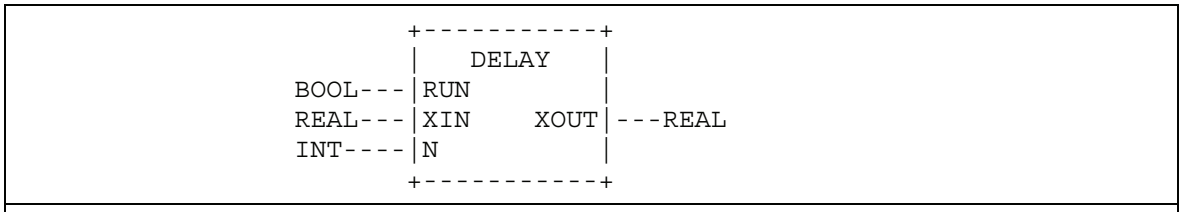
Dieser Funktionsbaustein (LAG1) führt die Filterfunktion 1. Ordnung durch.




```
FUNCTION_BLOCK LAG1
  VAR_INPUT
    RUN : BOOL ;      (* 1 = ein, 0 = rücksetzen  *)
    XIN : REAL ;      (* Eingangsvariable      *)
    TAU : TIME ;      (* Filter-Zeit-Konstante       *)
    CYCLE : TIME ;    (* Abtast-Interval          *)
  END_VAR
  VAR_OUTPUT XOUT : REAL ; END_VAR  (* Gefilterter Ausgang *)
  VAR K : REAL ;      (* Glättungskonstante, 0.0<=K<1.0 *)
  END_VAR
  IF RUN THEN XOUT := XOUT + K * (XIN - XOUT) ;
  ELSE XOUT := XIN ;
       K := TIME_TO_REAL(CYCLE) / TIME_TO_REAL(CYCLE + TAU) ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.2 Funktionsbaustein TOTZEIT

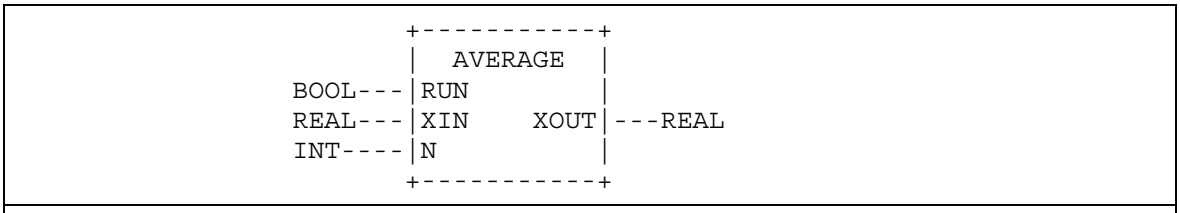
Dieser Funktionsbaustein (DELAY) ermittelt eine Totzeit (Verzögerung) für N Messwerte.




```
FUNCTION_BLOCK DELAY    (* N-Messwerte Verzögerung *)
  VAR_INPUT
    RUN : BOOL ;      (* 1 = ein, 0 = rücksetzen       *)
    XIN : REAL ;
    N   : INT         (* 0 <= N < 128 oder hersteller-spezifischer Maximalwert *)
  END_VAR
  VAR_OUTPUT XOUT : REAL; END_VAR      (* Verzögerte Ausgabe *)
  VAR X : ARRAY [0..127]               (* N-Element-Schlange in FIFO-Technik *)
          OF REAL;
      I, IXINT, IXOUT  :  INT  :=  0;
  END_VAR
  IF RUN THEN IXIN := MOD(IXIN + 1, 128) ; X[IXIN] := XIN ;
              IXOUT := MOD(IXOUT + 1, 128) ; XOUT := X[IXOUT];
  ELSE XOUT := XIN ; IXIN := N ; IXOUT := 0;
       FOR I := 0 TO N DO X[I] := XIN; END_FOR;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.3 Funktionsbaustein MITTEL

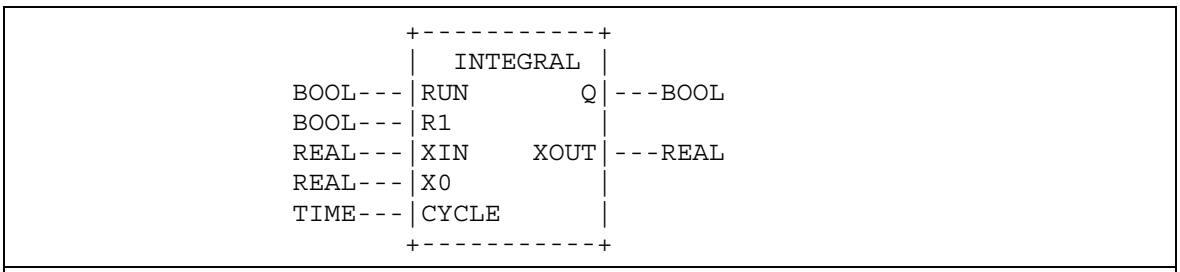
Dieser Funktionsbaustein (AVERAGE) führt eine laufende Mittelwertbildung über N Messwerte durch.




```
FUNCTION_BLOCK AVERAGE
  VAR_INPUT
    RUN : BOOL ;      (* 1 = ein, 0 = rücksetzen *)
    XIN : REAL ;      (* Eingangsvariable *)
    N   : INT ;       (* 0 <= N < 128 *)
  END_VAR             (* oder hersteller-spezifische Maximalwert *)
 VAR_OUTPUT XOUT : REAL ; END_VAR (* Gemittelter Ausgang *)
  VAR SUM  : REAL := 0.0; (* Laufende Summe *)
      FIFO : DELAY ;      (* N-Element FIFO *)
  END_VAR
  SUM := SUM - FIFO.XOUT ;
  FIFO (RUN := RUN , XIN := XIN, N := N) ;
  SUM := SUM + FIFO.XOUT ;
  IF RUN THEN XOUT := SUM/N ;
  ELSE SUM := N*XIN ; XOUT := XIN ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.4 Funktionsbaustein INTEGRAL

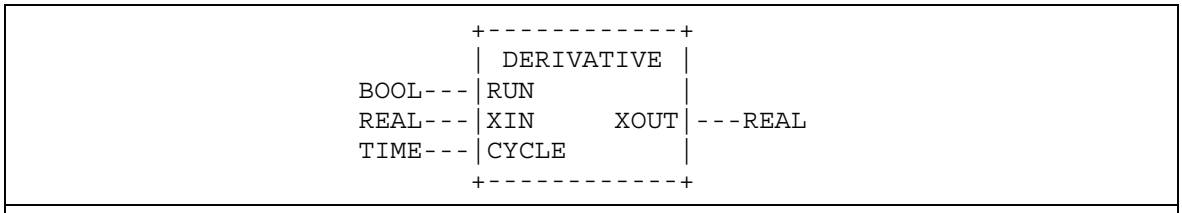
Dieser Funktionsbaustein (INTEGRAL) führt eine Integration über die Zeit durch.




```
FUNCTION_BLOCK INTEGRAL
  VAR_INPUT
    RUN : BOOL ;       (* 1 = integriert, 0 = halt *)
    R1 : BOOL ;        (* Vorrangiges Rücksetzen *)
    XIN : REAL ;       (* Eingabe-Variable *)
    X0  : REAL ;       (* Initial value *)
    CYCLE : TIME ;     (* Abtastperiode *)
  END_VAR
  VAR_OUTPUT
    Q : BOOL ;         (* NOT R1 *)
    XOUT : REAL ;      (* Integrierte Ausgabe *)
  END_VAR
  Q := NOT R1 ;
  IF R1 THEN XOUT := X0 ;
  ELSIF RUN THEN XOUT := XOUT + XIN * TIME_TO_REAL(CYCLE);
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.5 Funktionsbaustein GRADIENT

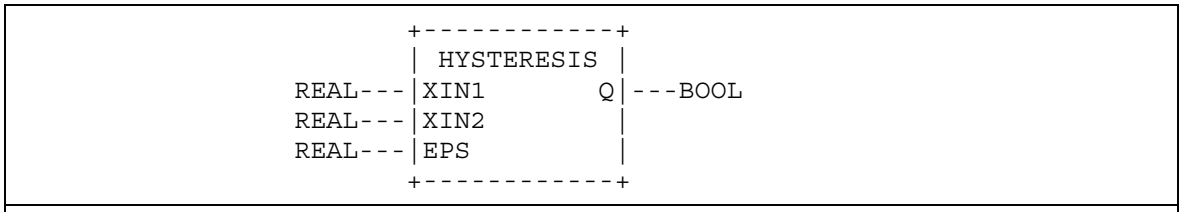
Dieser Funktionsbaustein (DERIVATIVE) führt die Differenzierung in Bezug auf die Zeit durch.




```
FUNCTION_BLOCK DERIVATIVE
  VAR_INPUT
    RUN : BOOL ;           (* 0 = Rücksetzen *)
    XIN : REAL ;           (* Eingabe, wird differenziert *)
    CYCLE : TIME ;         (* Abtastperiode *)
  END_VAR
  VAR_OUTPUT
    XOUT : REAL ;          (* Differenzierter Ausgang *)
  END_VAR
  VAR X1, X2, X3 : REAL ; END_VAR
  IF RUN THEN
     XOUT := (3.0 * (XIN - X3) + X1 - X2)
             / (10.0 * TIME_TO_REAL(CYCLE)) ;
     X3 := X2 ; X2 := X1 ; X1 := XIN ;
  ELSE XOUT := 0.0; X1 := XIN ; X2 := XIN ; X3 := XIN ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.6 Funktionsbaustein HYSTERESE

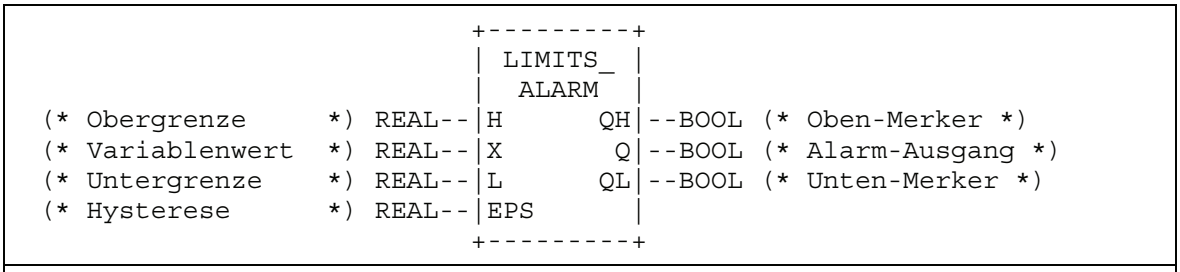
Dieser Funktionsbaustein (HYSTERESIS) führt eine boolesche Hysterese-Funktion auf die Differenz von REAL Eingängen durch.



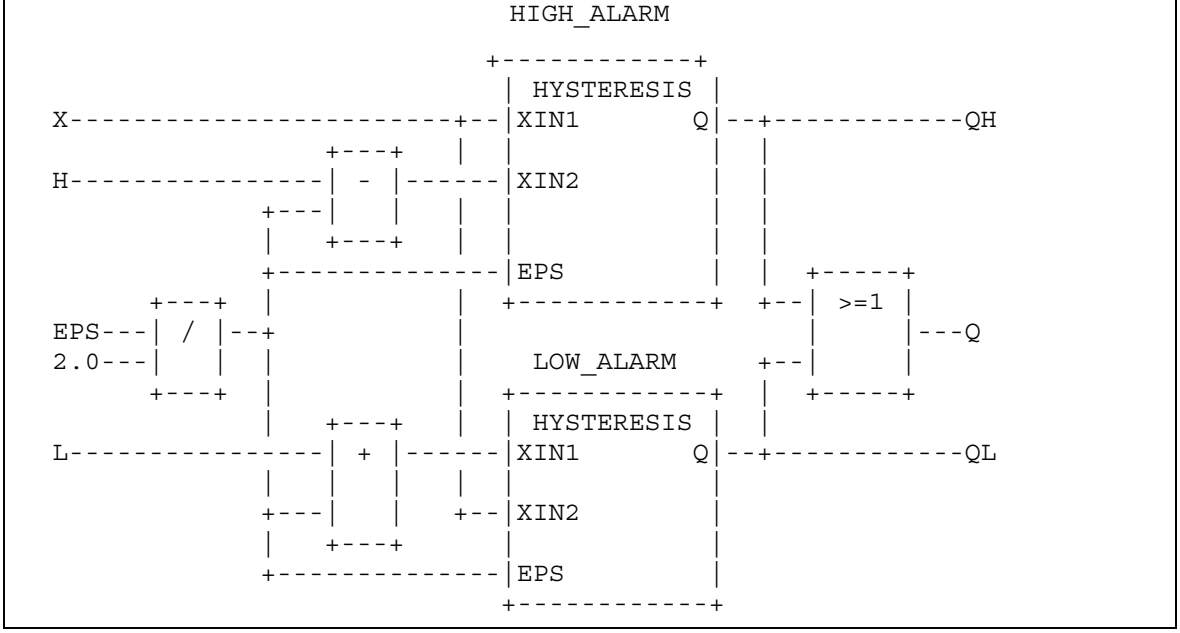

```
FUNCTION_BLOCK HYSTERESIS
    (* Boolesche Hysterese auf die Differenz von REAL Eingängen XIN1 - XIN2 *)
  VAR_INPUT XIN1, XIN2, EPS : REAL; END_VAR
  VAR_OUTPUT Q : BOOL := 0; END_VAR
  IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0; END_IF ;
  ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1 ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.7 Funktionsbaustein GRENZ_ALARM

Dieser Funktionsbaustein (LIMITS_ALARM) führt eine Überwachung der Ober-/Untergrenze mit Hysterese an den beiden Ausgängen durch.




```
            (* Funktionsbaustein-Rumpf in FBS Sprache *)
```



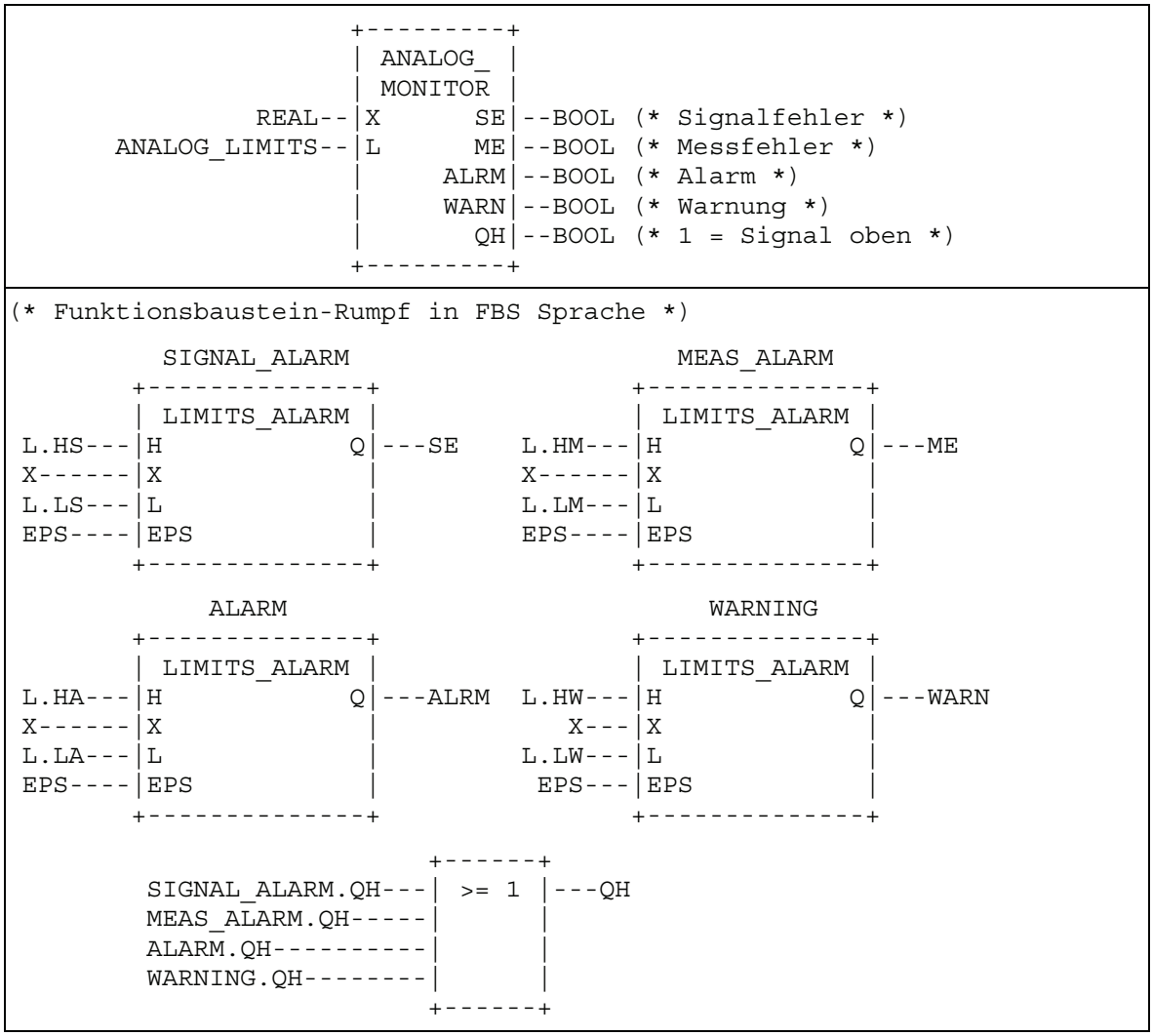
F.6.8 Struktur ANALOG_GRENZEN

Dieser Datentyp (ANALOG_LIMITS) deklariert die Parameter für die Analogsignal-Überwachung.

```
TYPE ANALOG_LIMITS :
      STRUCT
         HS : REAL ;      (* Oberes Ende des Signalbereichs *)
         HM : REAL ;      (* Oberes Ende des Messbereichs *)
         HA : REAL ;      (* Obere Alarmgrenze *)
         HW : REAL ;      (* Obere Warngrenze *)
         NV : REAL ;      (* Nennwert *)
         EPS : REAL ;     (* Hysterese *)
         LW : REAL ;      (* Untere Warngrenze *)
         LA : REAL ;      (* Untere Alarmgrenze *)
         LM : REAL ;      (* Unteres Ende des Messbereichs *)
         LS : REAL ;      (* Unteres des Signalbereichs *)
      END_STRUCT
END_TYPE
```

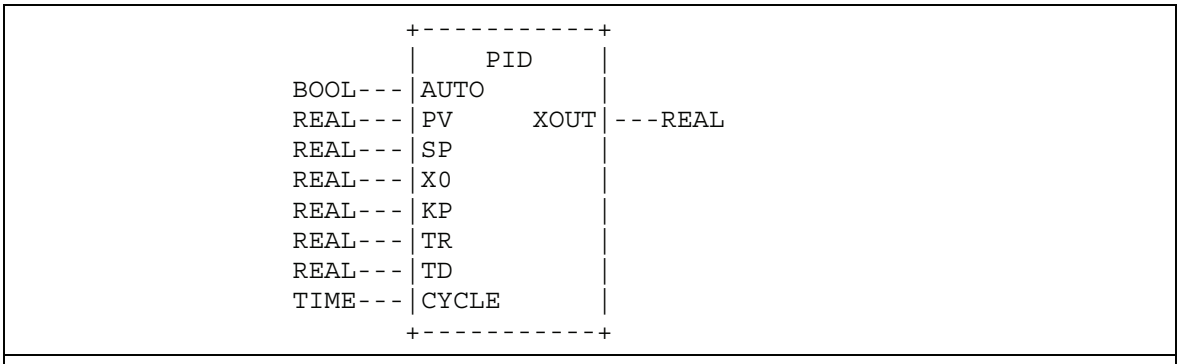
F.6.9 Funktionsbaustein ANALOG_ÜBERWACHUNG

Dieser Funktionsbaustein (ANALOG_MONITOR) führt die Überwachung eines Analogsignals durch.



F.6.10 Funktionsbaustein PID

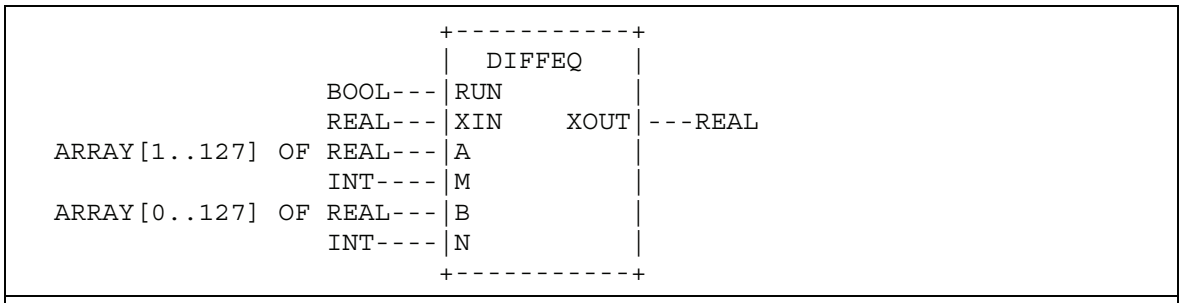
Dieser Funktionsbaustein (PID) führt die Proportional- und Integral- und Differenzial-Regelung durch. Die Funktionalität wird durch funktionales Zusammenfügen der oben deklarierten Funktionsbausteine erreicht.




```
FUNCTION_BLOCK PID

  VAR_INPUT
    AUTO : BOOL ;         (* 0 - Hand , 1 - Automatik *)
    PV : REAL ;           (* Ist-Wert X *)
    SP : REAL ;           (* Soll-Wert W *)
    X0 : REAL ;           (* Manuelle Ausgangsnachführung - *)
                          (* typischerweise von Transferstation *)
    KP : REAL ;           (* Proportionalbeiwert *)
    TR : REAL ;           (* Nachstellzeit TN *)
    TD : REAL ;           (* Vorhaltezeit TV *)
    CYCLE : TIME ;        (* Abtastperiode *)
  END_VAR

  VAR_OUTPUT XOUT : REAL; END_VAR

  VAR ERROR : REAL ;         (* PV - SP *)
      ITERM : INTEGRAL ;     (* FB für I-Glied *)
      DTERM : DERIVATIVE ;   (* FB für D-Glied *)
  END_VAR

  ERROR := PV - SP ;
      (*** ITERM so anpassen, dass  XOUT := X0, wenn AUTO = 0 ***)
  ITERM (RUN := AUTO, R1 := NOT AUTO, XIN := ERROR,
         X0 := TR * (X0 - ERROR), CYCLE := CYCLE) ;
  DTERM (RUN := AUTO, XIN := ERROR, CYCLE := CYCLE) ;
  XOUT := KP * (ERROR + ITERM.XOUT/TR + DTERM.XOUT*TD) ;

END_FUNCTION_BLOCK
```

F.6.11 Funktionsbaustein DIFFGLEICH

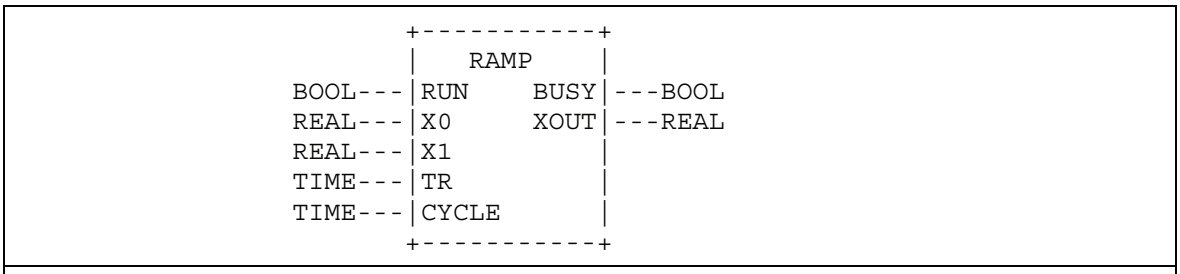
Dieser Funktionsbaustein (DIFFEQ) führt eine allgemeine Differenzen-Gleichung durch.




```
FUNCTION_BLOCK DIFFEQ

  VAR_INPUT
    RUN : BOOL ;                (* 1 = ein, 0 = rücksetzen *)
    XIN : REAL ;
    A : ARRAY[1..127] OF REAL ; (* Eingangskoeffizienten *)
    M : INT ;                   (* Länge d. Eingangshistorie *)
    B : ARRAY[0..127] OF REAL ; (* Ausgangskoeffizienten *)
    N : INT ;                   (* Länge der Ausgangshistorie *)
  END_VAR

  VAR_OUTPUT XOUT : REAL := 0.0 ; END_VAR
  VAR  (* ANMERKUNG: Der Hersteller darf andere Feldgrößen festlegen *)
    XI : ARRAY [0..127] OF REAL ;   (* Eingangshistorie *)
    XO : ARRAY [0..127] OF REAL ;   (* Ausgangshistorie *)
    I : INT ;
  END_VAR

  XO[0] := XOUT ; XI[0] := XIN ;
  XOUT := B[0] * XIN ;
  IF RUN THEN
     FOR I := M TO 1 STEP -1 DO
         XOUT := XOUT + A[I] * XO[I] ; XO[I] := XO[I-1];
     END_FOR;

     FOR I := N TO 1 STEP -1 DO
         XOUT := XOUT + B[I] * XI[I] ; XI[I] := XI[I-1];
     END_FOR;

  ELSE
     FOR I := 1 TO M DO XO[I] := 0.0; END_FOR;
     FOR I := 1 TO N DO XI[I] := 0.0; END_FOR;
  END_IF ;

END_FUNCTION_BLOCK
```

F.6.12 Funktionsbaustein RAMPE

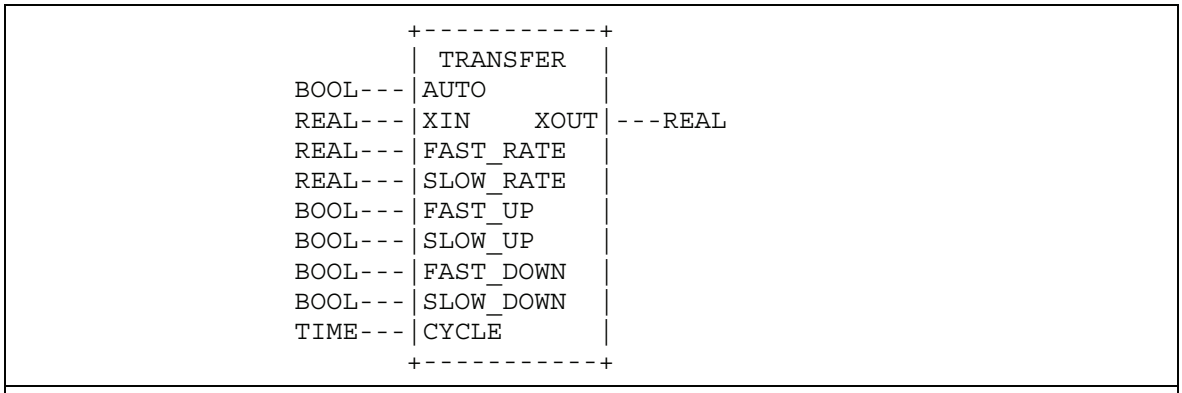
Dieser Funktionsbaustein (RAMP) führt eine Rampenfunktion auf Zeitbasis durch.




```
FUNCTION_BLOCK RAMP

  VAR_INPUT
    RUN : BOOL ;       (* 0 - Spur X0, 1 - Rampe/Spur X1 *)
    X0,X1 : REAL ;
    TR : TIME ;        (* Rampendauer *)
    CYCLE : TIME ;     (* Abtastperiode *)
  END_VAR

  VAR_OUTPUT
    BUSY : BOOL ;       (* BUSY = 1 während Hochlauf *)
    XOUT : REAL := 0.0 ;
  END_VAR

  VAR XI : REAL ;        (* Eingangswert *)
      T : TIME := T#0s; (* Verstrichene Rampen-Zeit *)
  END_VAR

  BUSY := RUN ;
  IF RUN THEN
     IF T >= TR THEN BUSY := 0 ; XOUT := X1 ;
     ELSE XOUT := XI + (X1-XI) * TIME_TO_REAL(T)
                     / TIME_TO_REAL(TR) ;
          T := T + CYCLE ;
     END_IF ;
  ELSE XOUT := X0 ; XI := X0 ; T := t#0s ;
  END_IF ;

END_FUNCTION_BLOCK
```

F.6.13 Funktionsbaustein TRANSFER

Dieser Funktionsbaustein (TRANSFER) führt die Funktion eines Leitgeräts mit stoßfreien Übergang aus.




```
FUNCTION_BLOCK TRANSFER

 VAR_INPUT
   AUTO : BOOL ;      (* 1 - Spur X0, 0 - Rampe oder Halt *)
   XIN : REAL ;       (* Typischerweise vom Funktionblock PID *)
   FAST_RATE, SLOW_RATE : REAL ;    (* Auf/Ab - Rampenanstieg *)
   FAST_UP, SLOW_UP,      (* Typischerweise Drucktaster *)
   FAST_DOWN, SLOW_DOWN : BOOL;
   CYCLE : TIME ; (* Abtastperiode *)
 END_VAR

 VAR_OUTPUT XOUT : REAL ; END_VAR

 VAR XFER_RAMP : INTEGRAL ;
     RAMP_RATE : REAL ;
 END_VAR

 RAMP_RATE := 0.0 ;
 IF NOT AUTO THEN
  IF FAST_UP THEN RAMP_RATE := FAST_RATE; END_IF;
  IF SLOW_UP THEN RAMP_RATE := RAMP_RATE + SLOW_RATE; END_IF;
  IF FAST_DOWN THEN RAMP_RATE := RAMP_RATE - FAST_RATE; END_IF;
  IF SLOW_DOWN THEN RAMP_RATE := RAMP_RATE - SLOW_RATE; END_IF;
 END_IF ;

 XFER_RAMP (RUN := 1, CYCLE := CYCLE, R1 := AUTO,
            XIN := RAMP_RATE, X0 := XIN) ;
 XOUT := XFER_RAMP.XOUT;

END_FUNCTION_BLOCK
```

F.7 Programm SCHÜTTGUT

Ein Steuerungssystem (GRAVEL) dient zur Dosierung einer Schüttgutmenge, die von einem Bediener vorgegeben wird. Diese Schüttgutmenge wird aus einem Silo in einen Behälter gefüllt und nach der Dosierung vom Behälter in einen LKW geladen.

Die Menge des zu transportierenden Schüttguts wird über einen Zahleneinsteller innerhalb eines Bereichs von 0 bis 99 Einheiten vorgegeben. Die Schüttgutmenge im Behälter wird an einer Ziffernanzeige dargestellt.

Aus Sicherheitsgründen muss sofort optisch und akustisch gemeldet werden, wenn das Silo leer ist. Diese Meldefunktionen sind im Steuerungsprogramm zu lösen.

Eine grafische Darstellung dieser Steuerungsaufgabe ist in Bild F.2 gezeigt; die Deklarationen der Variablen für das Steuerungsprogramm sind in Bild F.3 angegeben.

Wie in Bild F.4 gezeigt ist, besteht die Arbeitsweise des Systems aus einer Anzahl von Schritten, beginnend mit dem Füllen des Behälters auf Grund des Befehls mit dem Druck-Taster FILL. Nach Füllen des Behälters beginnt der Ladevorgang des LKWs auf Grund des Befehls durch den LOAD Drucktaster, sobald sich der LKW auf der Rampe befindet. Das Laden besteht aus einer „Vorlauf"-Phase für das Starten des Transportbandes, gefolgt vom Entleeren des Behälters auf das Band. Sobald der Behälter geleert ist, hat das Band eine vorbestimmte Zeitdauer einen „Nachlauf", um zu gewährleisten, dass alles Schüttgut auf den LKW gefördert wird. Der Ladevorgang wird angehalten, sobald die Automatik durch den OFF Taster angehalten wird.

Bild F.5 zeigt sowohl den OFF/ON Ablauf als auch die Erzeugung der Blinkanzeige-Takte und der Bandmotor-Verriegelung bei ON geschalteter Steuerung.

Die Überwachung des Füllstands im Behälter, die Bedienungsschnittstelle und die Anzeigefunktionen sind in Bild F.6 definiert.

Eine Text-Version des Programmrumpfs für SCHÜTTGUT (GRAVEL) unter Anwendung der ST-Sprache mit AS-Elementen ist in Bild F.7 angegeben.

Eine Beispiel-Konfiguration für das Programm SCHÜTTGUT ist in Bild F.8 angegeben.

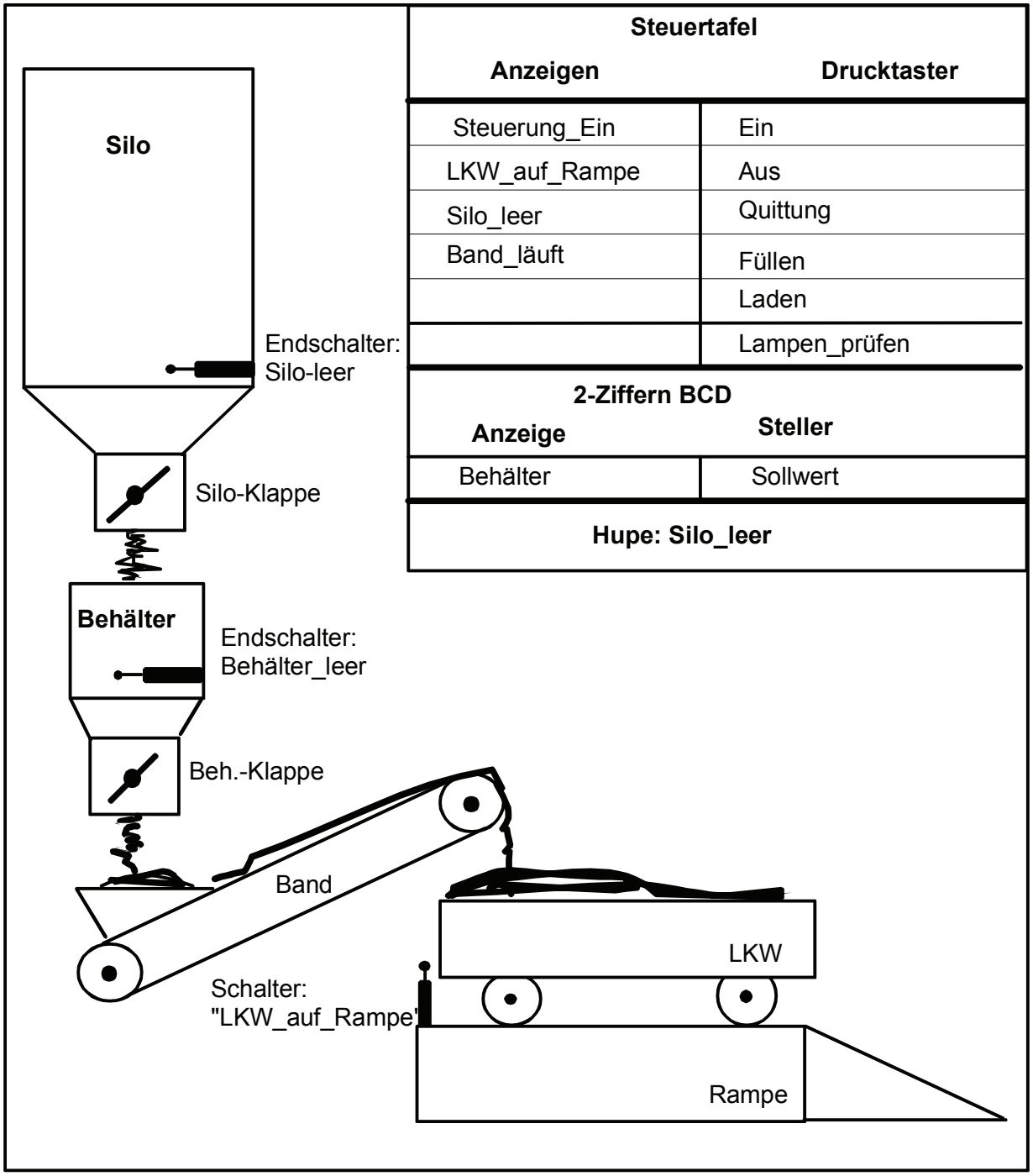


Bild F.2 – Schüttgut Mess- und Verladesystem

```
PROGRAM GRAVEL  (* Schüttgut Mess- und Verladesystem *)

VAR_INPUT
   OFF_PB          : BOOL ;
   ON_PB           : BOOL ;
   FILL_PB         : BOOL ;
   SIREN_ACK       : BOOL ;
   LOAD_PB         : BOOL ; (* LKW aus Behälter laden *)
   JOG_PB          : BOOL ;
   LAMP_TEST       : BOOL ;
   TRUCK_ON_RAMP   : BOOL ; (* Lichtschranke *)
   SILO_EMPTY_LS   : BOOL ;
   BIN_EMPTY_LS    : BOOL ;
   SETPOINT        : BYTE ; (* 2-Ziffern-BCD *)
END_VAR

VAR_OUTPUT
   CONTROL_LAMP    : BOOL ;
   TRUCK_LAMP      : BOOL ;
   SILO_EMPTY_LAMP : BOOL ;
   CONVEYOR_LAMP   : BOOL ;
   CONVEYOR_MOTOR  : BOOL ;
   SILO_VALVE      : BOOL ;
   BIN_VALVE       : BOOL ;
   SIREN           : BOOL ;
   BIN_LEVEL       : BYTE ;
END_VAR

VAR
 BLINK_TIME : TIME;    (* BLINK ON/OFF Zeit *)
 PULSE_TIME : TIME;    (* LEVEL_CTR Inkrement-Interval *)
 RUNOUT_TIME: TIME;    (* Bandlaufzeit nach Laden *)
 RUN_IN_TIME: TIME;    (* Bandlaufzeit vor Laden *)
 SILENT_TIME: TIME;    (* Hupe still nach SIREN_ACK *)
 OK_TO_RUN  : BOOL;    (* 1 = Band darf laufen *)

(* Funktionsbausteine *)
 BLINK: TON;        (* Blinker OFF Zeitdauer / ON Ausgang *)
 BLANK: TON;        (* Blinker ON Zeitdauer / aus-Puls *)
 PULSE: TON;        (* LEVEL_CTR Takt *)
 SIREN_FF: RS;
 SILENCE_TMR: TP;  (* Hupe-Stille-Periode *)
END_VAR

VAR RETAIN LEVEL_CTR : CTU ; END_VAR

  (* Programmrumpf *)

END_PROGRAM
```

Bild F.3 – Deklarationen für das Programm SCHÜTTGUT

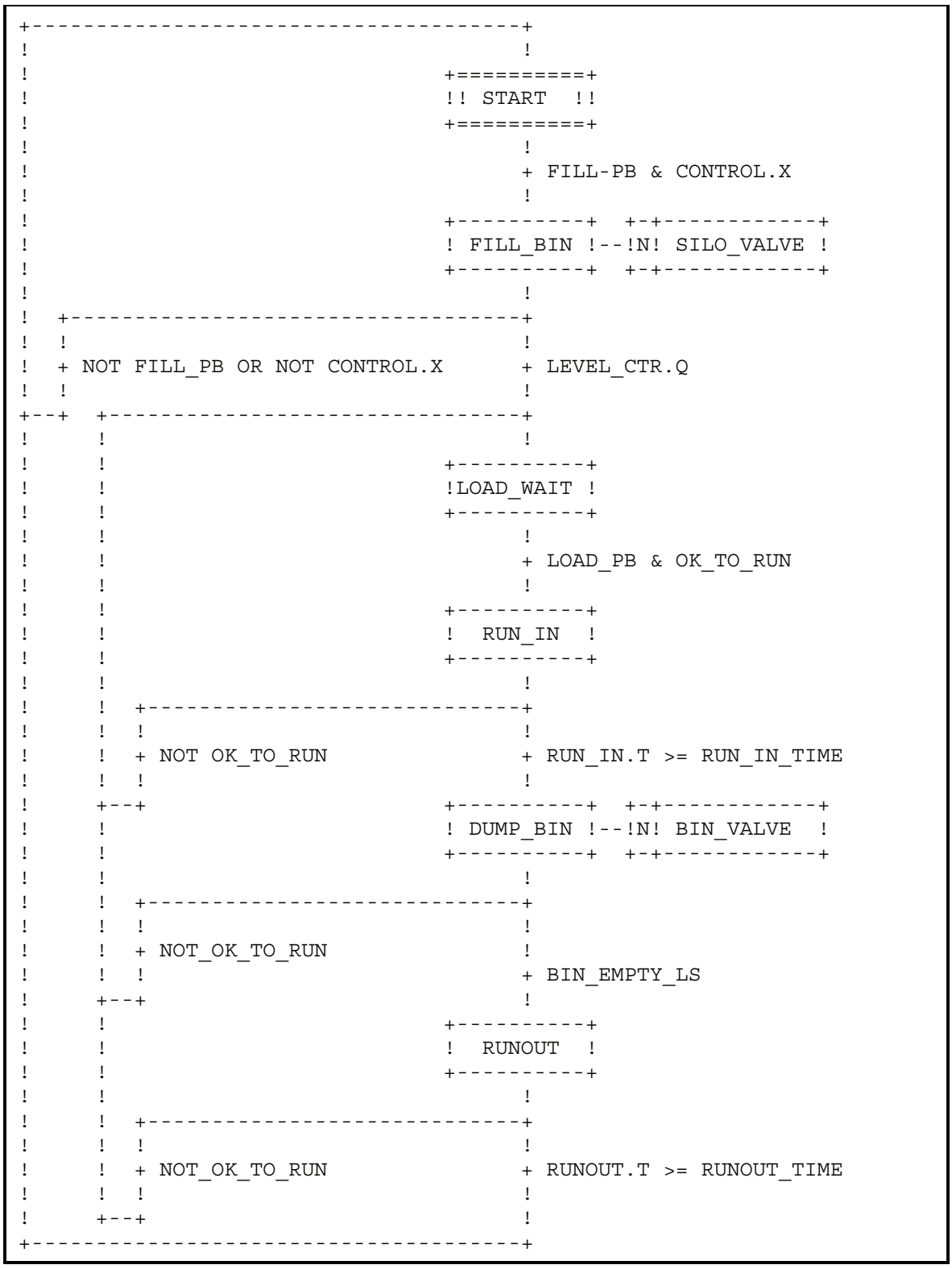


Bild F.4 – AS für den Programm-Rumpf von SCHÜTTGUT

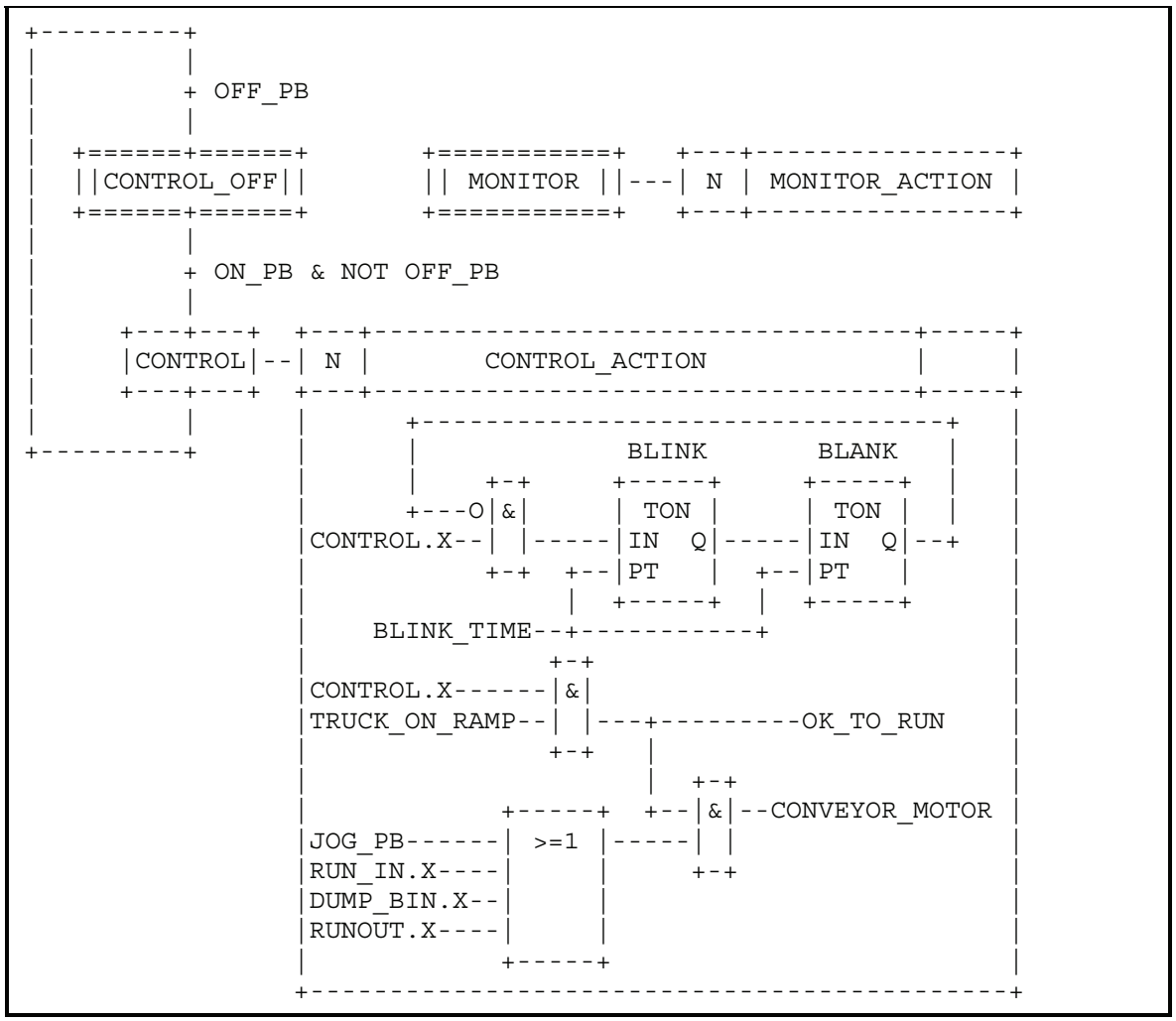


Bild F.5 – Programm-Rumpf von SCHÜTTGUT (Fortsetzung) – Ablauf und Überwachung des Steuerungszustands

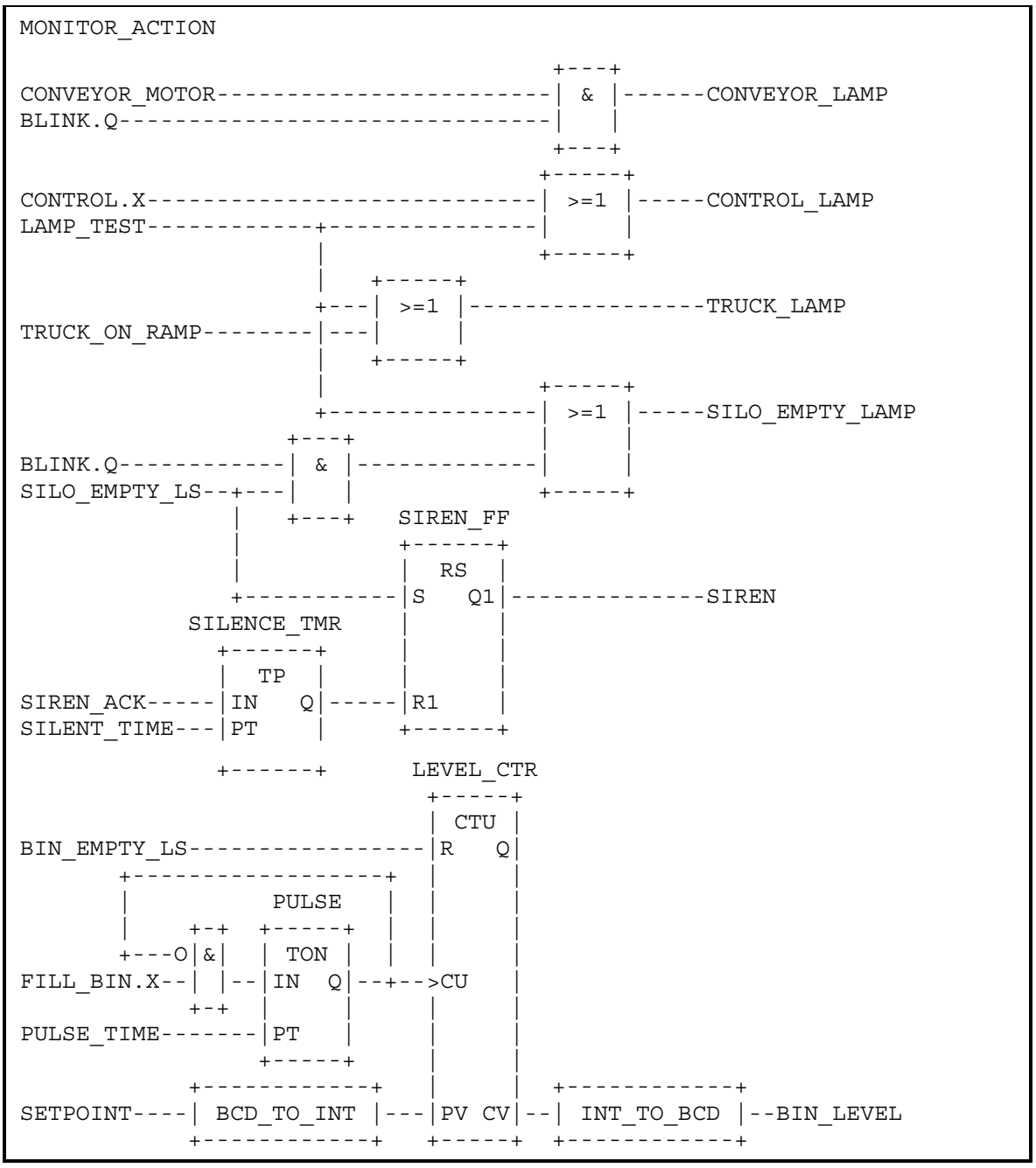


Bild F.6 – Aktionsrumpf von `MONITOR_ACTION` in FBS-Sprache

```
(* Zustände des Hauptablaufs *)

INITIAL_STEP START : END_STEP
TRANSITION FROM START TO FILL_BIN
   := FILL_PB & CONTROL.X ; END_TRANSITION

STEP FILL_BIN: SILO_VALVE(N); END_STEP
TRANSITION FROM FILL_BIN TO START
   := NOT FILL_PB OR NOT CONTROL.X ; END_TRANSITION
TRANSITION FROM FILL_BIN TO LOAD_WAIT := LEVEL_CTR.Q ;
END_TRANSITION

STEP LOAD_WAIT : END_STEP
TRANSITION FROM LOAD_WAIT TO RUN_IN
   := LOAD_PB & OK_TO_RUN ; END_TRANSITION

STEP RUN_IN : END_STEP
TRANSITION FROM RUN_IN TO LOAD_WAIT := NOT OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM RUN_IN TO DUMP_BIN
   := RUN_IN.T > RUN_IN_TIME;
END_TRANSITION

STEP DUMP_BIN: BIN_VALVE(N);  END_STEP
TRANSITION FROM DUMP_BIN TO LOAD_WAIT := NOT OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM DUMP_BIN TO RUNOUT := BIN_EMPTY_LS ;
END_TRANSITION

STEP RUNOUT : END_STEP
TRANSITION FROM RUNOUT TO LOAD_WAIT := NOT OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM RUNOUT TO START
   := RUNOUT.T >= RUNOUT_TIME;  END_TRANSITION
```

```
(* Ablauf der Steuerzustände *)

INITIAL_STEP CONTROL_OFF : END_STEP
TRANSITION FROM CONTROL_OFF TO CONTROL
   := ON_PB & NOT OFF_PB ; END_TRANSITION

STEP CONTROL: CONTROL_ACTION(N); END_STEP

ACTION CONTROL_ACTION:

       BLINK(EN:=CONTROL.X & NOT BLANK.Q, PT := BLINK_TIME) ;
       BLANK(EN:=BLINK.Q, PT := BLINK_TIME) ;
       OK_TO_RUN := CONTROL.X & TRUCK_ON_RAMP ;
       CONVEYOR_MOTOR :=
         OK_TO_RUN & OR(JOG_PB, RUN_IN.X, DUMP_BIN.X, RUNOUT.X);
END_ACTION

TRANSITION FROM CONTROL TO CONTROL_OFF := OFF_PB ;  END_TRANSITION

(* Überwachungslogik *)

INITIAL_STEP MONITOR: MONITOR_ACTION(N); END_STEP

ACTION MONITOR_ACTION:
  CONVEYOR_LAMP   := CONVEYOR_MOTOR & BLINK.Q ;
  CONTROL_LAMP    := CONTROL.X OR LAMP_TEST ;
  TRUCK_LAMP      := TRUCK_ON_RAMP OR LAMP_TEST ;
  SILO_EMPTY_LAMP := BLINK.Q & SILO_EMPTY_LS OR LAMP_TEST ;
  SILENCE_TMR(EN:=SIREN_ACK, PT:=SILENT_TIME) ;
  SIREN_FF   (S:=SILO_EMPTY_LS, R1:=SILENCE_TMR.Q) ;
  SIREN := SIREN_FF.Q1 ;
  PULSE    (EN:=FILL_BIN.X & NOT PULSE.Q, PT:=PULSE_TIME) ;
  LEVEL_CTR(EN := BIN_EMPTY_LS, CU := PULSE.Q,
            PV := BCD_TO_INT(SETPOINT)) ;
  BIN_LEVEL := INT_TO_BCD(LEVEL_CTR.CV) ;
END_ACTION
```

Bild F.7 – Rumpf des Programms SCHÜTTGUT in Textdarstellung mit AS unter Verwendung von ST-Sprachelementen

```
CONFIGURATION GRAVEL_CONTROL
  RESOURCE PROC1 ON PROC_TYPE_Y
    PROGRAM G : GRAVEL
     (* Eingänge *)
      (OFF_PB       := %I0.0 ,
       ON_PB         := %I0.1 ,
       FILL_PB       := %I0.2 ,
       SIREN_ACK     := %I0.3 ,
       LOAD_PB       := %I0.4 ,
       JOG_PB        := %I0.5 ,
       LAMP_TEST     := %I0.7 ,
       TRUCK_ON_RAMP := %I1.4 ,
       SILO_EMPTY_LS := %I1.5 ,
       BIN_EMPTY_LS  := %I1.6 ,
       SETPOINT      := %IB2  ,
     (* Ausgänge *)
       CONTROL_LAMP    => %Q4.0,
       TRUCK_LAMP      => %Q4.2,
       SILO_EMPTY_LAMP => %Q4.3,
       CONVEYOR_LAMP   => %Q5.3,
       CONVEYOR_MOTOR  => %Q5.4,
       SILO_VALVE      => %Q5.5,
       BIN_VALVE       => %Q5.6,
       SIREN           => %Q5.7,
       BIN_LEVEL       => %B6) ;
       END_PROGRAM
  END_RESOURCE
END_CONFIGURATION
```

Bild F.8 – Beispiel-Konfiguration für das Programm SCHÜTTGUT

F.8 Programm AGV

Wie in Bild F.9 veranschaulicht, soll ein Programm ein „Automatisch Gelenktes Vehikel“ (AGV) steuern. Das AGV soll zwischen den beiden Endpositionen pendeln: links (angezeigt durch den Endschalter S3) und rechts (angezeigt durch den Endschalter S4). Die Normalposition des AGV ist auf der linken Seite.

Das AGV soll einen Zyklus einer Links/Rechtsbewegung und zurück ausführen, wenn der Bediener den Druckschalter S1 betätigt und zwei Zyklen, wenn der Bediener den Druckschalter S2 betätigt. Es ist auch möglich, vom Einzel-Zyklus zum Doppel-Zyklus überzugehen, indem man den Druckschalter S2 während eines Einzel-Zyklus betätigt. Wenn S1 oder S2 betätigt bleiben, soll ein Verriegeln (Nicht-Wiederholen) erreicht werden.

Bild F.10 zeigt die grafische Deklaration des Programms AGV, während Bild F.11 eine typische Konfiguration für dieses Programm zeigt. Bild F.12 zeigt den Programm-Rumpf von AGV, der aus einem Hauptsteuer-Ablauf und einem Einzel-Zyklus-Steuerablauf besteht.

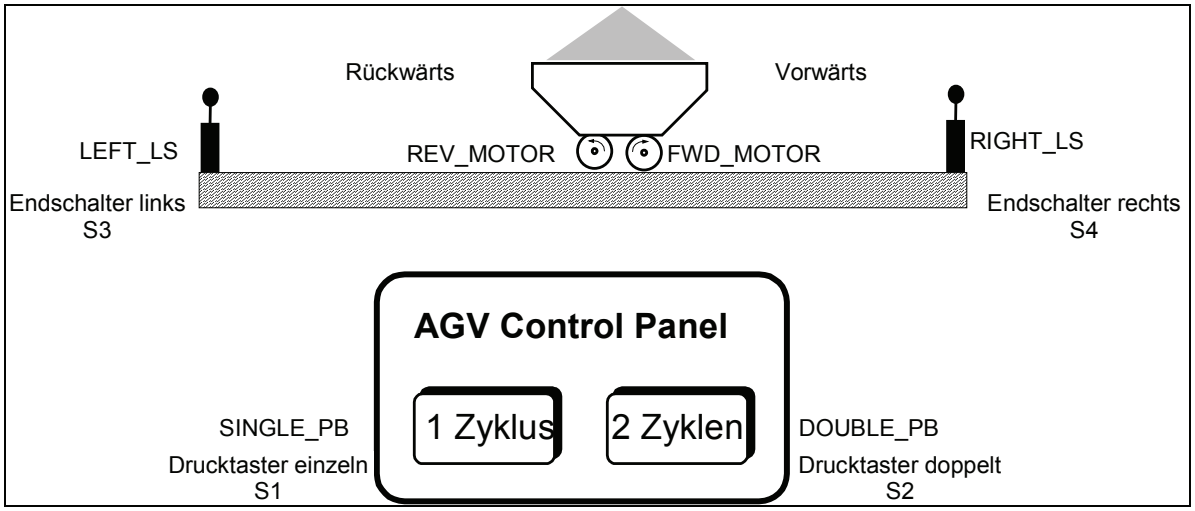


Bild F.9 – Physikalisches Modell für das Programm AGV

```
        +----------------------+
        |         AGV          |
BOOL---|SINGLE_PB    FWD_MOTOR|---BOOL
BOOL---|DOUBLE_PB    REV_MOTOR|---BOOL
BOOL---|LEFT_LS               |
BOOL---|RIGHT_LS              |
        +----------------------+
```

Bild F.10 – Grafische Deklaration des Programms AGV

```
CONFIGURATION AGV_CONTROL
    RESOURCE AGV_PROC: SMALL_PC
                          AGV_1
                  +--------------------+
                  |        AGV         |
          %IX1---|SINGLE_PB  FWD_MOTOR|---%QX1
          %IX2---|DOUBLE_PB  REV_MOTOR|---%QX2
          %IX3---|LEFT_LS             |
          %IX4---|RIGHT_LS            |
                  +--------------------+
```

Bild F.11 – Eine grafische Konfiguration des Programms AGV

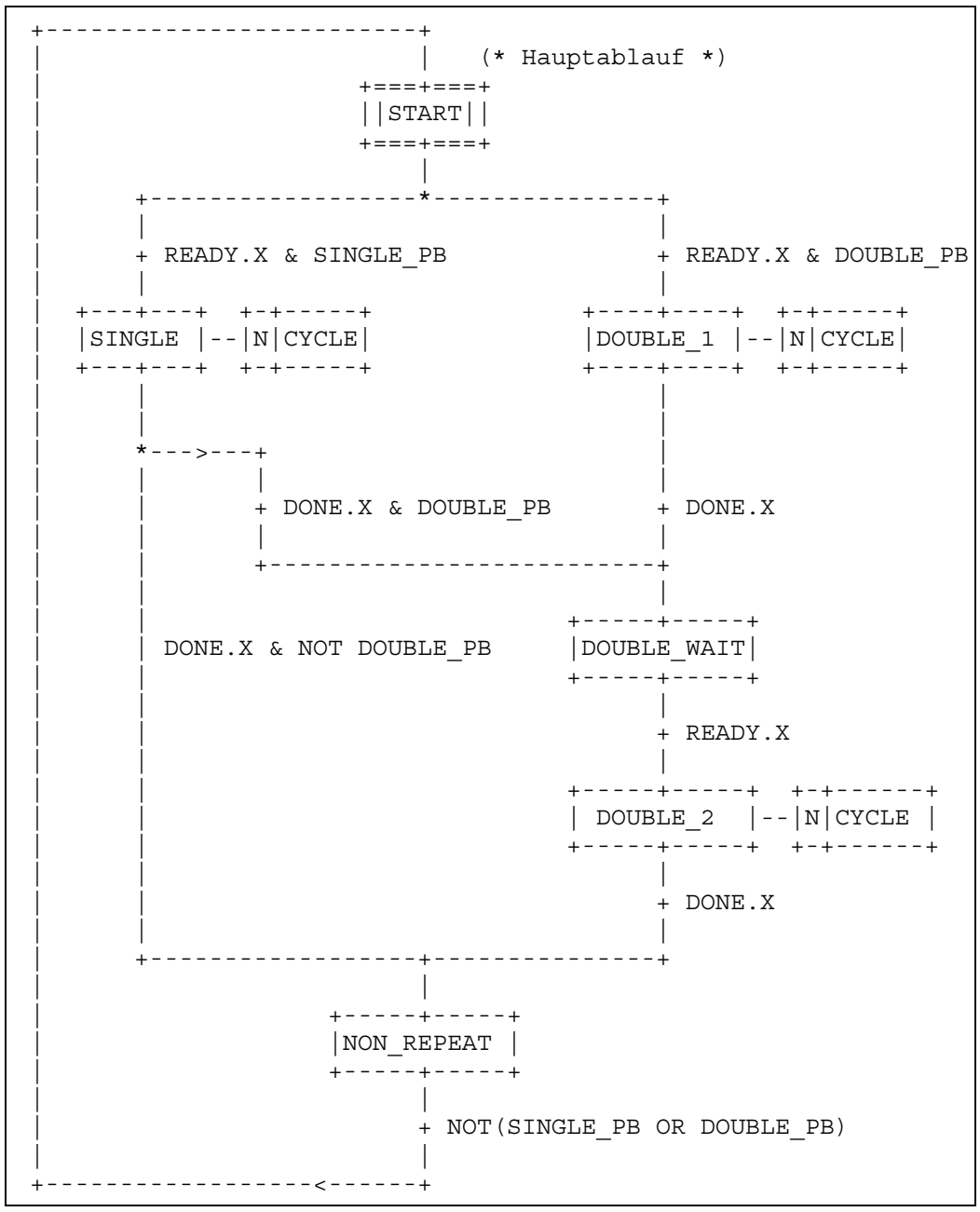


Bild F.12 – Programm-Rumpf von AGV

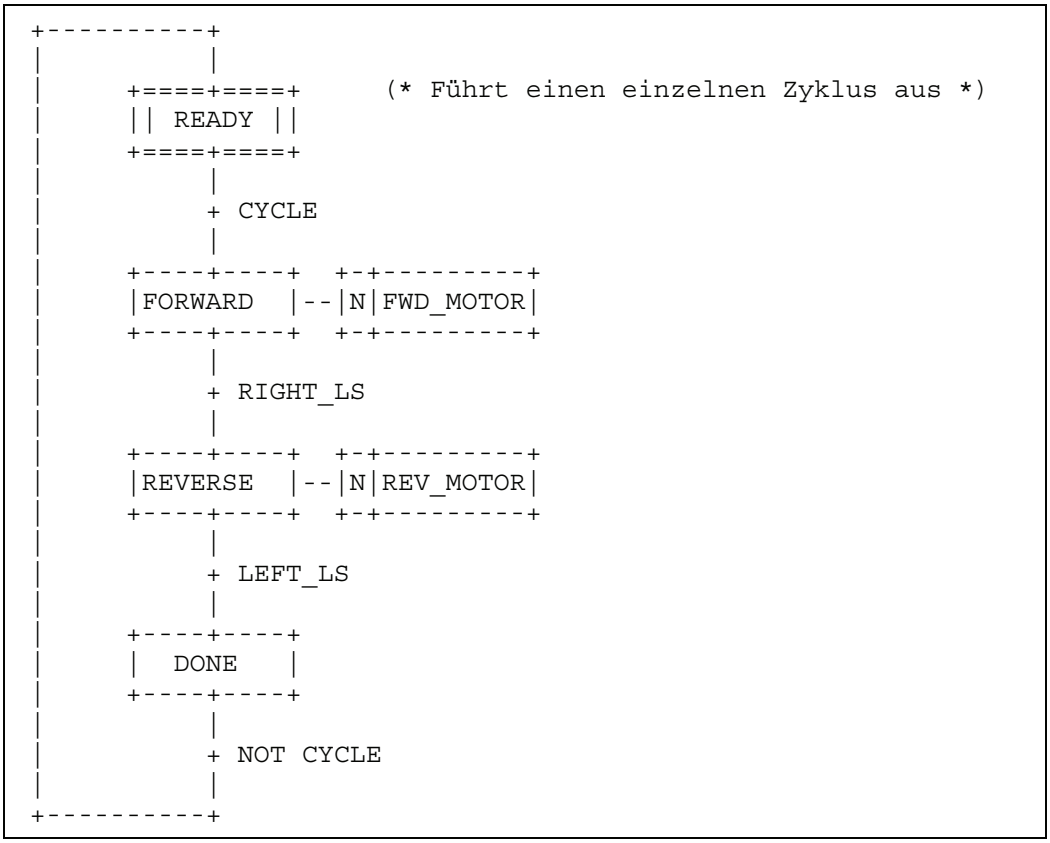


Bild F.12 – Programm-Rumpf von AGV *(Fortsetzung)*

F.9 Gebrauch von aufgezählten Datentypen

Das folgende Beispiel veranschaulicht den Gebrauch von aufgezählten Datentypen in Anweisungen der ST Sprache und Anweisungsliste. Angenommen ein aufgezählter Datentyp wurde durch die folgende Deklaration definiert:

```
TYPE SPEED: (SLOW, MEDIUM, FAST, VERY_FAST); END_TYPE
```

Zusätzlich angenommen ein Eingang und ein Ausgang eines Funktionsbaustein-Typs ist deklariert durch:

```
VAR_INPUT MOTOR_SPEED: SPEED; END_VAR
VAR_OUTPUT SPEED_OUT: SPEED; END_VAR
```

Dann könnte eine `CASE` Anweisung, falls der Rumpf des Funktionsbaustein-Typs in ST Sprache definiert ist, folgendermaßen verwendet werden:

```
CASE MOTOR_SPEED OF
  SLOW:   (* langsamer *);
  MEDIUM: (* hält die aktuelle Geschwindigkeit *);
  FAST:   (* schneller *);
ELSE      (* Sonderfall *);
END_CASE;
```

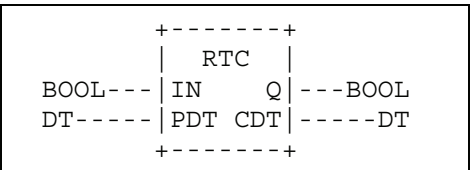
Falls der Rumpf des Funktionsbaustein-Typs in der Sprache AWL definiert ist, könnten die folgenden Anweisungen verwendet werden:

```
LD SPEED#SLOW (* aufgezählte Wert wird durch den Datentyp bestimmt *)
ST SPEED_OUT
```

F.10 Funktionsbaustein Echtzeituhr (RTC)

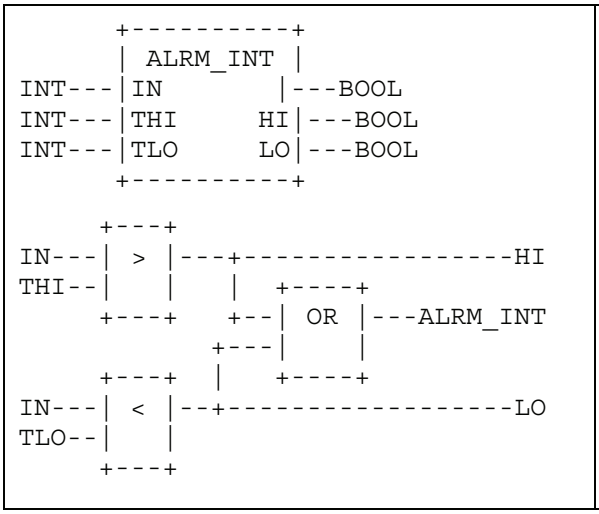
Der unten gezeigte Funktionsbaustein Echtzeituhr (Real time clock – RTC) setzt bei der nächsten Auswertung des Funktionsbausteins, die einer Transition von 0 nach 1 am Eingang IN folgt, den Ausgang CDT auf den Wert des Eingangs PDT.

Funktionsbaustein Echtzeituhr

PDT = (Preset date and time) Voreingestelltes Datum und Uhrzeit, geladen bei steigender Flanke von IN CDT = (Current date and time) aktuelles Datum und Uhrzeit, gültig wenn IN=1 Q = Kopie von EN	

F.11 Funktionsbaustein ALRM_INT

Dieser Funktionsbaustein liefert eine einfache Alarmmeldung für Unter- und Obergrenzwert an einem Eingang vom Typ INT, und er veranschaulicht den Gebrauch der VAR_OUTPUT Deklaration bei Funktionen. Der Funktionsausgang ist TRUE, wenn die obere oder untere Grenze überschritten wird, und separate Ausgänge liefern die Alarmmeldungen für die obere und untere Bedingung.




```
FUNCTION ALRM_INT : BOOL

VAR_INPUT
  IN : INT ;
  THI : INT ; (* Obergrenze   *)
  TLO : INT ; (* Untergrenze *)
END_VAR

VAR_OUTPUT
  HI: BOOL; (* Oberer Alarm  *)
  LO: BOOL; (* Unterer Alarm *)
END_VAR

HI := IN > THI ;
LO := IN < THL ;
ALRM_INT := THI OR THL ;

END_FUNCTION
```

Anhang G
(informativ)

Zeichensatz

ANMERKUNG 1 Der Inhalt der letzten Ausgabe von „Tabelle 1, Spalte 00: ISO-646 IRV“ ist normativ für diese Norm. Der hier angegebene Zeichensatz dient nur zur Information.

ANMERKUNG 2 Bei Variablen vom Type `STRING` sind die individuellen Byte-Codierungen der Zeichen in dem hier angegebenen Zeichensatz in Tabelle G.2 angegeben. Bei Variablen vom Type `WSTRING` ist das numerische Äquivalent der individuellen 16-Bit-Wort Codierungen auch in Tabelle G.2 angegeben.

Tabelle G.1 – Zeichen-Darstellung

	Erste hexadezimale Ziffer					
Zweite hexadezimale Ziffer	**2**	**3**	**4**	**5**	**6**	**7**
0		0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	\	l	|
D	-	=	M	]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	

Tabelle G.2 – Zeichen-Codierung

Dez.	Hexa	Name	Dez.	Hexa	Name
032	20	SPACE	080	50	LATIN CAPITAL LETTER P
033	21	EXCLAMATION MARK	081	51	LATIN CAPITAL LETTER Q
034	22	QUOTATION MARK	082	52	LATIN CAPITAL LETTER R
035	23	NUMBER SIGN	083	53	LATIN CAPITAL LETTER S
036	24	DOLLAR SIGN	084	54	LATIN CAPITAL LETTER T
037	25	PERCENT SIGN	085	55	LATIN CAPITAL LETTER U
038	26	AMPERSAND	086	56	LATIN CAPITAL LETTER V
039	27	APOSTROPHE	087	57	LATIN CAPITAL LETTER W
040	28	LEFT PARENTHESIS	088	58	LATIN CAPITAL LETTER X
041	29	RIGHT PARENTHESIS	089	59	LATIN CAPITAL LETTER Y
042	2A	ASTERISK	090	5A	LATIN CAPITAL LETTER Z
043	2B	PLUS SIGN	091	5B	LEFT SQUARE BRACKET
044	2C	COMMA	092	5C	REVERSE SOLIDUS
045	2D	HYPHEN-MINUS	093	5D	RIGHT SQUARE BRACKET
046	2E	FULL STOP	094	5E	CIRCUMFLEX ACCENT
047	2F	SOLIDUS	095	5F	LOW LINE
048	30	DIGIT ZERO	096	60	GRAVE ACCENT
049	31	DIGIT ONE	097	61	LATIN SMALL LETTER A
050	32	DIGIT TWO	098	62	LATIN SMALL LETTER B
051	33	DIGIT THREE	099	63	LATIN SMALL LETTER C
052	34	DIGIT FOUR	100	64	LATIN SMALL LETTER D
053	35	DIGIT FIVE	101	65	LATIN SMALL LETTER E
054	36	DIGIT SIX	102	66	LATIN SMALL LETTER F
055	37	DIGIT SEVEN	103	67	LATIN SMALL LETTER G
056	38	DIGIT EIGHT	104	68	LATIN SMALL LETTER H
057	39	DIGIT NINE	105	69	LATIN SMALL LETTER I
058	3A	COLON	106	6A	LATIN SMALL LETTER J
059	3B	SEMICOLON	107	6B	LATIN SMALL LETTER K
060	3C	LESS-THAN SIGN	108	6C	LATIN SMALL LETTER L
061	3D	EQUALS SIGN	109	6D	LATIN SMALL LETTER M
062	3E	GREATER-THAN SIGN	110	6E	LATIN SMALL LETTER N
063	3F	QUESTION MARK	111	6F	LATIN SMALL LETTER O
064	40	COMMERCIAL AT	112	70	LATIN SMALL LETTER P
065	41	LATIN CAPITAL LETTER A	113	71	LATIN SMALL LETTER Q
066	42	LATIN CAPITAL LETTER B	114	72	LATIN SMALL LETTER R
067	43	LATIN CAPITAL LETTER C	115	73	LATIN SMALL LETTER S
068	44	LATIN CAPITAL LETTER D	116	74	LATIN SMALL LETTER T
069	45	LATIN CAPITAL LETTER E	117	75	LATIN SMALL LETTER U

Dez.	Hexa	Name	Dez.	Hexa	Name
070	46	LATIN CAPITAL LETTER F	118	76	LATIN SMALL LETTER V
071	47	LATIN CAPITAL LETTER G	119	77	LATIN SMALL LETTER W
072	48	LATIN CAPITAL LETTER H	120	78	LATIN SMALL LETTER X
073	49	LATIN CAPITAL LETTER I	121	79	LATIN SMALL LETTER Y
074	4A	LATIN CAPITAL LETTER J	122	7A	LATIN SMALL LETTER Z
075	4B	LATIN CAPITAL LETTER K	123	7B	LEFT CURLY BRACKET
076	4C	LATIN CAPITAL LETTER L	124	7C	VERTICAL LINE
077	4D	LATIN CAPITAL LETTER M	125	7D	RIGHT CURLY BRACKET
078	4E	LATIN CAPITAL LETTER N	126	7E	TILDE
079	4F	LATIN CAPITAL LETTER O			

Stichwortverzeichnis

Die Hauptverweise der *Begrenzungszeichen* und *Schlüsselwörter* sind in Anhang C angegeben.

Anhang ZA
(normativ)

Normative Verweisungen auf internationale Publikationen mit ihren entsprechenden europäischen Publikationen

Diese Europäische Norm enthält durch datierte oder undatierte Verweisungen Festlegungen aus anderen Publikationen. Diese normativen Verweisungen sind an den jeweiligen Stellen im Text zitiert, und die Publikationen sind nachstehend aufgeführt. Bei datierten Verweisungen gehören spätere Änderungen oder Überarbeitungen dieser Publikationen zu dieser Europäischen Norm nur, falls sie durch Änderung oder Überarbeitung eingearbeitet sind. Bei undatierten Verweisungen gilt die letzte Ausgabe der in Bezug genommenen Publikation (einschl. Änderungen).

ANMERKUNG Wenn internationale Publikationen durch gemeinsame Abänderungen geändert wurden, durch (mod) angegeben, gelten die entsprechenden EN/HD.

Publikation	Jahr	Titel	EN/HD	Jahr
IEC 60050	Reihe	International Electrotechnical Vocabulary	–	–
IEC 60559	1989	Binary floating-point arithmetic for microprocessor systems	HD 592 S1	1991
IEC 60617-12	1997	Graphical symbols for diagrams Part 12: Binary logic elements	EN 60617-12	1998
IEC 60617-13	1993	Part 13: Analogue elements	EN 60617-13	1993
IEC 60848	2002	GRAFCET specification language for sequential function charts	EN 60848	2002
IEC 61131-1	– [1)]	Programmable controllers Part 1: General information	EN 61131-1	1994 [2)]
IEC 61131-5	– [1)]	Part 5: Communications	EN 61131-5	2001 [2)]
ISO/AFNOR	1989	Dictionary of computer science – The standardised vocabulary	–	–
ISO/IEC 10646-1	1993	Information technology – Universal Multiple-Octet Coded Character set (UCS) – Part 1: Architecture and Basic Multilingual Plane	–	–

[1)] Undatierte Verweisung.

[2)] Zum Zeitpunkt der Veröfftenlichung dieser Norm gültige Ausgabe.