

DIN 66253

ICS 35.060

Entwurf

Einsprüche bis 2017-09-21
 Vorgesehen als Ersatz für
 die 1986-03 zurückgezogene
 Vornorm
 DIN 66253-1:1981-07,
 DIN 66253-2:1998-04 und
 DIN 66253-3:1989-01

Informationsverarbeitung – Programmiersprache PEARL – SafePEARL

Information processing –
 Programming language PEARL –
 SafePEARL

Technologies de l'information –
 Langage de programmation PEARL –
 SafePEARL

Anwendungswarnvermerk

Dieser Norm-Entwurf mit Erscheinungsdatum 2017-07-21 wird der Öffentlichkeit zur Prüfung und Stellungnahme vorgelegt.

Weil die beabsichtigte Norm von der vorliegenden Fassung abweichen kann, ist die Anwendung dieses Entwurfs besonders zu vereinbaren.

Stellungnahmen werden erbeten

- vorzugsweise online im Norm-Entwurfs-Portal von DIN unter www.din.de/go/entwuerfe bzw. für Norm-Entwürfe der DKE auch im Norm-Entwurfs-Portal der DKE unter www.entwuerfe.normenbibliothek.de, sofern dort wiedergegeben;
- oder als Datei per E-Mail an nia@din.de möglichst in Form einer Tabelle. Die Vorlage dieser Tabelle kann im Internet unter www.din.de/go/stellungnahmen-norm-entwuerfe oder für Stellungnahmen zu Norm-Entwürfen der DKE unter www.dke.de/stellungnahme abgerufen werden;
- oder in Papierform an den DIN-Normenausschuss Informationstechnik und Anwendungen (NIA), 10772 Berlin, Burggrafenstr. 6, 10787 Berlin.

Die Empfänger dieses Norm-Entwurfs werden gebeten, mit ihren Kommentaren jegliche relevanten Patentrechte, die sie kennen, mitzuteilen und unterstützende Dokumentationen zur Verfügung zu stellen.

Gesamtumfang 146 Seiten

DIN-Normenausschuss Informationstechnik und Anwendungen (NIA)

Inhalt

	Seite
Vorwort	5
Einleitung	6
1 Anwendungsbereich	7
2 Normative Verweisungen	7
3 Bezeichnung	8
4 Allgemeines zu PEARL	8
5 Sprachteilmengen für die Sicherheitsintegritätsstufen SIL0 bis SIL4.....	11
6 Regeln zum Aufbau von PEARL-Sprachformen	11
6.1 Format	11
6.2 Zeichensatz	12
6.3 Grundelemente.....	13
6.3.1 Aufbau.....	13
6.3.2 Bezeichner.....	13
6.3.3 Konstanten.....	13
6.3.4 Kommentare.....	14
6.4 Aufbau von Sprachformen.....	14
7 Programm.....	15
7.1 Aufbau.....	15
7.2 Module	15
7.3 Vereinbarungen	17
7.3.1 Deklaration	17
7.3.2 Spezifikation (SPC) und Identifikation (SPC IDENT)	18
7.4 Blöcke.....	19
7.5 Bezüge zwischen Modulen	21
7.6 Auswahl des Sicherheitsintegritätsniveaus.....	22
7.7 Variablen und Konstanten	22
7.7.1 Vereinbarung von Variablen (DCL).....	22
7.7.2 Ganze Zahlen (FIXED)	23
7.7.3 Gleitpunktzahlen (FLOAT)	24
7.7.4 Bitketten (BIT).....	25
7.7.5 Zeichenketten (CHARACTER).....	26
7.7.6 Längenvereinbarung	27
7.7.7 Uhrzeiten (CLOCK)	28
7.7.8 Zeitdauern (DURATION)	28
7.8 Referenzen (REF).....	29
7.9 Felder	32
7.10 Strukturen	34
7.11 Typvereinbarung (TYPE).....	36
7.12 Initialisierungsattribut (INITIAL)	38
7.13 Zuweisungsschutz (INV).....	39
7.14 Physikalische Einheiten (PHYSUNIT).....	41
7.15 Konstante Ausdrücke	41
8 Ausdrücke und Zuweisungen	42
8.1 Ausdrücke	42

8.2	Zuweisungen	51
8.2.1	Aufbau.....	51
8.2.2	Zuweisungen an skalare Variablen.....	51
8.2.3	Zuweisungen an Strukturen.....	51
8.3	Überlagerung von Datenstrukturen.....	52
9	Anweisungen zur Steuerung des sequentiellen Ablaufs	52
9.1	Übersicht	52
9.2	Bedingte Anweisung (IF)	53
9.3	Anweisungsauswahl (CASE) und Leeranweisung.....	53
9.4	Wiederholung (FOR-REPEAT).....	56
9.5	Sprunganweisung (GOTO).....	59
9.6	Exit-Anweisung (EXIT)	59
9.7	Ursache-Wirkungstabellen	60
9.8	Sequentielle Ablaufpläne.....	61
9.9	Prozeduren	63
9.9.1	Vereinbarung von Prozeduren (PROC)	63
9.9.2	Aufruf von Prozeduren (CALL)	66
9.9.3	Referenzen auf Prozeduren (REF PROC)	68
10	Parallele Aktivitäten	69
10.1	Vereinbarung von Tasks (TASK).....	69
10.2	Zeitschranken.....	70
10.3	Anweisungen zur Steuerung von Tasks	71
10.3.1	Starten einer Task (ACTIVATE).....	71
10.3.2	Beenden einer Task (TERMINATE).....	73
10.3.3	Anhalten einer Task (SUSPEND)	73
10.3.4	Fortsetzen einer Task (CONTINUE)	74
10.3.5	Verzögern einer Task (RESUME)	75
10.3.6	Ausplanen einer Task (PREVENT).....	76
10.4	Synchronisierung von Tasks	76
10.4.1	Semaphor-Variablen (SEMA) und -Anweisungen (REQUEST, RELEASE, TRY)	76
10.4.2	Bolt-Variablen (BOLT) und -Anweisungen (ENTER, LEAVE, RESERVE, FREE)	81
10.4.3	Betriebsmittelschutz und zeitliche Überwachung von Synchronisierungsoperationen (LOCK).....	83
10.5	Unterbrechungssignale und Unterbrechungsanweisungen	84
10.5.1	Vereinbarung von Unterbrechungssignalen und Software-Interrupts	84
10.5.2	Unterbrechungsanweisungen (TRIGGER, ENABLE, DISABLE)	85
10.6	Referenzen auf Tasks (REF TASK).....	86
10.7	Vordefinierte Prozeduren.....	87
11	Eingabe und Ausgabe	89
11.1	Systemteil.....	89
11.2	Vereinbarung von Datenstationen (DATION) im Problemteil.....	91
11.2.1	Systemdatenstationen	91
11.2.2	Benutzerdefinierte Datenstationen.....	91
11.3	Öffnen und Schließen von Datenstationen (OPEN, CLOSE)	95
11.4	Read- und Write-Anweisungen (READ, WRITE).....	97
11.5	Get- und Put-Anweisungen (GET, PUT)	101
11.5.1	Anweisungen.....	101
11.5.2	Fixed-Format (F).....	104
11.5.3	Float-Format (E).....	105
11.5.4	Zeichenkettenformate (A und S).....	106
11.5.5	Bit-Format (B).....	106
11.5.6	Zeitformat (T)	107
11.5.7	Dauerformat (D)	108
11.5.8	List-Format (LIST)	109

11.5.9 R-Format (R)	109
11.6 Convert-Anweisung (CONVERT)	110
11.7 Fehlerbehandlung in E/A-Anweisungen (RST)	110
11.8 Take- und Send-Anweisungen	111
12 Signale	111
12.1 Vereinbarung von Signalen	111
12.2 Signaleinplanung (ON) und Signalanweisung (INDUCE)	112
13 Verteilte Systeme	115
13.1 Architekturbeschreibung	115
13.2 STATION-Teil: Beschreibung von Stationen	115
13.3 NET-Teil: Beschreibung physikalischer Verbindungswege	117
13.4 SYSTEM-Teil: Beschreibung des Zugangs zur Peripherie	118
13.4.1 Zuordnung zur Station	118
13.4.2 Stationsübergreifender Gerätezugriff	119
13.5 CONFIGURATION-Teil: Beschreibung der Software-Konfiguration	119
13.5.1 COLLECTIONS: Konfigurierbare Einheiten	120
13.5.2 Beschreibung logischer Verbindungen	121
13.5.3 Beschreibung initialen Ladens und von Rekonfigurationen	123
13.6 Vordefinierte Attribute	127
13.7 Vordefinierte Prozeduren	128
14 Botschaftenaustausch	129
14.1 Grundsätzliches	129
14.2 Portspezifikation	130
14.3 Botschaftenmechanismen	132
14.3.1 Übersicht	132
14.3.2 Einfache Kommunikationsprotokolle (1 zu 1)	133
14.3.3 Protokolle mit mehreren Empfängern	135
14.3.4 Protokolle mit mehreren Sendern	137
14.3.5 Mechanismen zur Zeitüberwachung	140
14.4 Vordefinierte Signale	142
Anhang A (normativ) Liste der Schlüsselwörter mit Kurzformen	143
Anhang B (normativ) Sonstige Wortsymbole in PEARL	145
Literaturhinweise	146

Vorwort

Das zuständige nationale Normungsgremium ist der Arbeitsausschuss NA 43-01-22 AA „Programmiersprachen“ im DIN-Normenausschuss Informationstechnik und Anwendungen (NIA).

Änderungen

Gegenüber DIN 66253-1:1981-07, DIN 66253-2:1998-04 und DIN 66253-3:1989-01 wurden folgende Änderungen vorgenommen:

- a) Die in Abschnitt 6 bis 12 dieser Norm definierte Sprache entspricht i. W. PEARL90 nach DIN 66253 erweitert um neu eingeführte sicherheitsbezogene Konstrukte;
- b) die in den Abschnitten 13 und 14 definierten Sprachmittel entsprechen i. W. Mehrrechner-PEARL nach DIN 66253-3;
- c) bei der Zusammenführung dieser Normen wurden die Syntaxdarstellung vereinheitlicht, Konsistenz hergestellt und Fehler korrigiert;
- d) der neue Abschnitt 5 dieser Norm definiert in kompakter Form sicherheitsgerichtete Teilmengen von PEARL.

Einleitung

Dieses Dokument umfasst eine völlig neue und unter Programmiersprachen einzigartige Erweiterung zur Programmierung sicherheitsgerichteter eingebetteter Systeme. Insbesondere diese Erweiterung wurde im Projekt „Normung der Echtzeitprogrammiersprache PEARL hinsichtlich funktionaler Sicherheit“ erarbeitet, das vom Bundeswirtschaftsministerium im Rahmen des Programms „Transfer von Forschungs- und Entwicklungsergebnissen durch Normung und Standardisierung“ gefördert wurde.

Gepflegt wird PEARL vom gemeinsamen Fachausschuss „Echtzeitsysteme“ der Gesellschaft für Informatik (GI), der VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und der Informationstechnischen Gesellschaft im VDE (ITG). Aktuelle Informationen zu PEARL sind über den Internet-Auftritt des Fachausschusses (<http://www.real-time.de>) zu finden.

Die vorliegende Norm ersetzt die historischen (zurückgezogenen) Normen DIN 66253 Teil 1 und DIN 66253 Teil 2 sowie DIN 66253-2 und DIN 66253-3.

Im Zuge der Anpassung an die quelloffene Variante OpenPEARL gemachte Änderungen zu nennen:

- a) Die Angabe eines Modul-Bezeichners im GLOBAL-Attribut ist nun verpflichtend.
- b) Bei fehlender Formatangabe in PUT/GET-Anweisungen wird nun standardmäßig das LIST-Format angenommen und ein SKIP angefügt.
- c) Ein leeres Eingabefeld beim F-Format liefert nun eine Fehlermeldung per SIGNAL.
- d) Bei TFU kann optional eine Puffergröße spezifiziert werden.
- e) Benutzerdefinierte Signale wurden eingeführt.
- f) Die Attribute RESIDENT und REENTRANT wurden eliminiert.
- g) Die Konstrukte OPERATOR, BY TYPE, VOID und REF STRUCT[] wurden gestrichen.
- h) Geschachtelte Funktionen und Prozeduren sind nicht mehr zulässig.
- i) Das Formatelement EOF (End-Of-File) wurde als relative Positionsangabe neu eingeführt.
- j) Das Attribut EXCLUSIVE wurde als OPEN-Parameter eingeführt.
- k) In TAKE/SEND-Anweisungen wurden das Attribut CONTROL() und das S(...)-Format gestrichen.
- l) Die Signalbehandlung wurde um die INDUCE-Anweisung und das SignalFinalStatement ergänzt.
- m) Implizites RETURN/TERMINATE wurde gestrichen.
- n) Das Attribut SYSTEM wurde in DATIONS zur Kennzeichnung von Systemdatenstation eingeführt.
- o) Der Aufruf TOBIT(FIXED(n)) liefert jetzt ein Ergebnis vom Typ BIT(n+1), bzw. liefert TOFIXED(BIT(n)) jetzt ein Ergebnis vom Typ FIXED(n-1).
- p) Der Aufruf FIXED(0) ist nun zulässig.

1 Anwendungsbereich

PEARL steht für **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage und ist eine höhere Programmiersprache, die komfortable, sichere und weitgehend rechnerunabhängige Programmierung von Multitasking- und Echtzeitaufgaben erlaubt. Die Abschnitte 6 bis 12 der vorliegenden Norm beschreiben die Grundsprache zur Formulierung von für Einprozessorplattformen formulierten Automatisierungsanwendungen, während die Abschnitte 13 und 14 Sprachkonstrukte einführen, die die Programmierung verteilter Systeme ermöglichen. Den vier in DIN EN 61508 definierten Sicherheitsintegritätsstufen jeweils zugeordnete, ineinander geschachtelte und mit höheren Anforderungen an die funktionale Sicherheit immer restriktiver werdende Teilmengen von PEARL werden in Abschnitt 5 zur Programmierung sicherheitskritischer Anwendungen eingeführt. Die vorliegende Norm stellt die Grundlage für die Erstellung von Übersetzern und Anleitungen wie Hand- und Lehrbücher zur Programmierung in PEARL dar.

Die Kombination der Teile dieser Norm stellt durch ihren Fokus auf die Entwicklung eingebetteter, echtzeitfähiger Systeme unter besonderer Berücksichtigung der nötigen Kommunikationsschnittstellen im verteilten Umfeld eine geeignete Basis für die Programmierung vernetzter eingebetteter Systeme dar, die auf mittels Software implementierter Steuerung oder Regelung über (globale) drahtgebundene oder drahtlose Kommunikationsnetze beruhen. Diese sogenannten cyber-physischen Systeme zeichnen sich durch einen in der Regel hohen Vernetzungsgrad ihrer mechanischen, elektronischen und informatischen Komponenten aus und können in mannigfaltigen Bereichen wie in dynamischen Automatisierungssystemen oder in Prozess-, Verkehrs- und Energieversorgungsleitsystemen zum Einsatz kommen.

Hierbei spielen die Aspekte funktionale Sicherheit und Zuverlässigkeit eine entscheidende Rolle, denn programmierbare elektronische Systeme werden zunehmend in einer Vielzahl sicherheitsgerichteter Anwendungen eingesetzt. Sie übernehmen Aufgaben zur Überwachung oder Steuerung von medizinischen Geräten, chemischen Anlagen, Anti-Blockier-Systemen, Luft- oder Weltraumfahrzeugen, Maschinen in der Fertigungstechnik sowie Kraftwerken und Energieverteilungssystemen. Diese Überwachungs- und Steuerungssysteme müssen deshalb hohe Sicherheitsanforderungen erfüllen, da falsch erstellte Systeme oder schlichtweg Fehler darin zum Versagen der Systemfunktionen führen können. Dies kann schwere Schäden verursachen oder gar Menschenleben gefährden. Der industrielle Bedarf an sicherheitsgerichteten, programmgesteuerten Systemen ist darum hoch und wird durch die zunehmende Automatisierung von Prozessen in Zukunft noch steigen. Hinzu kommt das steigende Sicherheitsbewusstsein in der Gesellschaft. Der so umrissene Bedarf, rechnergestützte Systeme mit hinreichendem Vertrauen in ihre Verlässlichkeit erstellen zu können, der ihre Zulassung für sicherheitskritische Steuer- und Regelaufgaben durch die Aufsichtsbehörden auf der Basis formeller Abnahmen erlaubt, war Anlass zur Entwicklung des an funktionaler Sicherheit orientierten Teils dieser Norm.

2 Normative Verweisungen

Die folgenden Dokumente, die in diesem Dokument teilweise oder als Ganzes zitiert werden, 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).

DIN 66253-2, *Informationsverarbeitung; Programmiersprache PEARL*90

DIN 66253-3, *Informationsverarbeitung; Programmiersprache PEARL; Mehrrechner-PEARL*

ISO/IEC 646:1991, *Information technology; ISO 7-bit coded character set for information interchange*

DIN EN 61131-3, *Speicherprogrammierbare Steuerungen — Teil 3: Programmiersprachen*

DIN EN 61508 (alle Teile), *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme*

3 Bezeichnung

Bezeichnung der Programmiersprache PEARL, die den Bedingungen dieser Norm entspricht:

Programmiersprache DIN 66253 – PEARL.

4 Allgemeines zu PEARL

Die Entwicklung der deutschen Echtzeitprogrammiersprache PEARL geht auf den Wunsch zurück, automatisierungs- und kerntechnische Anwendungen angemessen und mit hoher Produktivität formulieren zu können. Schon um das Jahr 1967 herum nahmen deshalb Ingenieure und Physiker von Firmen und Forschungsstellen die Arbeit an der Sprachspezifikation auf, die im April 1973 dann als erster Forschungsbericht des vom Bund geförderten Programms Prozessdatenverarbeitung (PDV) veröffentlicht wurde. Unmittelbar danach wurden erste Übersetzer erstellt und es begann auch die Normung von PEARL, wozu 1974 der Arbeitskreis DIN/FNI AK 5.8 gegründet wurde. Aus seinen Arbeiten gingen eine einfache und eine recht komplexe Version der Sprache hervor, die in den Teilen 1 und 2 der DIN 66253 als Basic PEARL 1981 und als Full PEARL 1982 genormt wurden. Daran schloss sich im Jahre 1989 der Teil 3 Mehrrechner-PEARL an, der durch konzeptionelle Klarheit und Eleganz besticht und ein bis heute von anderen unerreichtes Muster zur Strukturierung und Programmierung verteilter Systeme darstellt.

Mit PEARL hat die deutsche Informationstechnik auf dem Gebiet der Echtzeitsysteme eine international führende Stellung erreicht, denn die Sprache ist ihren Konkurrenten wegen der Klarheit und niedrigen Komplexität ihrer Konstrukte sowie ihrer sicherheitsfördernden leichten Lesbarkeit und Verständlichkeit technisch weit überlegen, weshalb ihre Konzepte auch bei der Standardisierung von Industrial Real-Time FORTRAN, einer Sammlung von Unterprogrammen zur Erweiterung von FORTRAN um Echtzeitfähigkeit, durch den International Purdue Workshop on Industrial Computer Systems Pate gestanden haben. Allein das französische LTR reichte in etwa an PEARL heran, ist jedoch leider verschwunden. Auf dem Weltmarkt ist als einzige mit PEARL konkurrierende Sprache Ada verblieben, die sehr komplex ist und deren Echtzeitfähigkeiten nur schwach ausgeprägt sind. Die im Laufe von 25 Jahren gemachten Erfahrungen fanden 1998 in der Norm DIN 66253-2 ihren Niederschlag, mit der Basic und Full PEARL durch PEARL90 abgelöst wurden.

PEARL erfüllt die allgemeinen Anforderungen an eine strukturierte Hochsprache für den Bereich eingebetteter Echtzeitsysteme. Einerseits bietet es die üblichen Konstrukte technisch-wissenschaftlich orientierter Sprachen mit besonders strenger Typprüfung an und andererseits stellt es komplette Instrumentarien zur Definition, Einplanung und Synchronisation nebenläufiger Prozesse und zur Kommunikation mit konventioneller und Prozessperipherie bereit. Um die Entwicklung großer Software-Pakete zu unterstützen, bestehen PEARL-Programme aus separat übersetzbaren Modulen, die sich selbst wiederum aus System- und Problemteilen zusammensetzen. Zur Förderung der Portabilität kapseln erstere die notwendigen Hardware-Beschreibungen und die Zuordnung symbolischer Namen zu Geräten ein, die dann von den eigentlichen, in den Problemteilen enthaltenen Automatisierungsprogrammen verwendet werden. Möglichkeiten zur Bitverarbeitung, zur Behandlung von Ausnahmezuständen, zur Verwaltung und Maskierung von Unterbrechungen sowie zur Testunterstützung gehören ebenfalls zum Sprachumfang von PEARL.

Die nebenläufige Verarbeitung asynchroner Rechenprozesse realisiert PEARL mit dem Konzept der Task. Als Elemente paralleler Ausführung werden Tasks nach einem sehr einfachen und fast minimalen, aber trotzdem vollständigen Modell mit den vier Zuständen bekannt, bereit, laufend und zurückgestellt verwaltet. Zustandsübergänge in diesem Modell werden durch externe, interne, Synchronisations- und Zeitereignisse veranlasst. Wegen der Asynchronität der Task-Abläufe müssen sich diese z. B. beim Zugriff auf gemeinsam genutzte Betriebsmittel synchronisieren, wozu PEARL klassische und verallgemeinerte Semaphore als Sprachkonstrukte anbietet.

Von anderen höheren Programmiersprachen unterscheidet sich PEARL besonders deutlich durch explizite Unterstützung des Zeitbegriffs. Die beiden Datentypen CLOCK für Zeitpunkte und DURATION für Zeitdauern stehen zusammen mit entsprechenden arithmetischen Operationen vordefiniert zur Verfügung. Während z. B. Ada als einzigen Zeitbezug nur die inhärent ungenaue Verzögerung des Ablaufs von Tasks kennt, die diese innerhalb ihrer Rümpfe selbst verwalten müssen, werden die Aktivierungen von PEARL-Tasks im Hauptprogramm in problemorientierter Weise eingeplant, und zwar einerseits als sporadische Reaktionen auf ankommende Unterbrechungssignale, auf Erreichen absoluter Zeitpunkte oder auf Verstreichen von Wartezeiten oder andererseits periodisch mit in verschiedenen Formen anzugebenden Frequenzen und Geltungsdauern. Hier zeigt PEARL eine besondere Stärke. So benötigt man in der für die Programmierung eingebetteter Systeme z.Zt. vorherrschenden, jedoch nicht echtzeitfähigen Sprache C im Zusammenwirken mit einem Betriebssystem überflüssig komplizierte, semantisch intransparente und mehrere Dutzend Anweisungen umfassende Codesegmente, um die periodische Aktivierung einer Task einzuplanen, wozu in PEARL eine einzige, auch für Nichtfachleute unmittelbar verständliche und selbstdokumentierende Anweisung ausreicht.

Genau wegen dieser Nähe von PEARL-Code zu Klartext eignet sich PEARL nicht nur zur eigentlichen Programmierung, sondern auch schon zur Spezifikation von Automatisierungsanwendungen und insbesondere für die Lehre. In PEARL-Darstellung lassen sich Denkweise und Konzepte der Echtzeitprogrammierung sehr leicht vermitteln und verstehen. Das hat positive Konsequenzen auf Qualität und Sicherheit der später von den Absolventen erstellten Systeme. Derzeit wird PEARL an rund 50 Hochschulen in Deutschland und Europa für die Ingenieursausbildung eingesetzt. PEARL erreicht somit einen hohen Grad an Ergonomie durch Unterstützung harmonischer Mensch-Maschine-Interaktion.

Bei den Abschnitten 13 und 14 dieser Norm, den Nachfolgern der Spracherweiterung Mehrrechner-PEARL, handelt es sich um die weltweit einzige Normierung einer Sprache zur Programmierung verteilter Systeme. Sie stellt Möglichkeiten zur Beschreibung von Konfigurationen, zur Parameterversorgung von Übersetzern, Bindern und Ladern, ohne dabei ausführbaren Code zu erzeugen, zur Kommunikation zwischen Rechnerknoten mittels Nachrichtenaustausch sowie zur dynamischen Rekonfiguration zur Verfügung. Die Architekturbeschreibung eines verteilten Systems umfasst Angaben über seine konstituierenden Rechnerknoten bzw. Einzelprozessorsysteme, deren Betriebszustände, die physikalischen Netzverbindungen, die Peripherieanschlüsse an die einzelnen Knoten sowie über Konfiguration und Verteilung der Software. Das Grundelement der Software-Verteilung ist die Collection, die aus einer Gruppe von Modulen, d. h. aus einem auf einem Einprozessorsystem lauffähigen Programm nach den Abschnitten 6 bis 12 dieser Norm besteht. Einem Rechnerknoten ist als Software immer genau eine Collection zugeordnet. Mithin ist die Collection auch die Austauschereinheit bei dynamischen Rekonfigurationen, die Reparatur, Wartung und allmähliche Leistungsabsenkung im Rahmen von Fehlerbehandlungen unterstützen. Collections kommunizieren untereinander mit Hilfe von Ports, an die sie Nachrichten senden bzw. von wo sie diese empfangen. Es wird sowohl asynchrone Kommunikation als auch synchrone mit Wartezeitüberwachung und Ausweichaktionen unterstützt. Zulässige Kommunikationsstrukturen sind 1:1, 1:n und n:1.

Im Hinblick auf die Einsetzbarkeit zur Programmierung sicherheitskritischer Steuer- und Regelungsaufgaben in (verteilten) Echtzeitumgebungen und mit dem Ziel, rechnergestützte Automatisierungssysteme beherrschbarer und damit einen Schritt hin zur Ermöglichung ihrer sicherheitstechnischen Abnahme zu machen, werden in Abschnitt 5 dieser Norm bei Beschränkung auf das Notwendigste für solche Aufgaben wirklich erforderliche Sprachkonstrukte in knapper tabellarischer Form identifiziert und inhärent sichere, ineinander geschachtelte Teilmengen von PEARL nach den Abschnitten 6 bis 14 gebildet, die den jeweiligen Anforderungen der Sicherheitsintegritätsstufen SIL1, SIL2, SIL3 und SIL4 nach DIN EN 61508 genügen (siehe Bild 1). Der dabei verfolgte und bereits beim Schritt von Ada zu SPARK erfolgreich verwendete Ansatz ist, als unsicher erkannte Sprachmerkmale schrittweise auf höheren Stufen zu verbieten. So muss nicht für jede Sicherheitsintegritätsstufe eine neue Sprache definiert und gelernt werden. Außerdem können Übersetzer prüfen, ob Programme die für die jeweils gewählte Stufe geltenden Sicherheitsauflagen erfüllen. Das Prinzip, Teilmengen einer Sprache für kritische Anwendungen zu definieren, gestattet es, Programme nach bestimmten Sicherheitsanforderungen zu entwickeln und Code für sicherheitskritische und -unkritische Systemteile nahtlos miteinander zu verbinden.

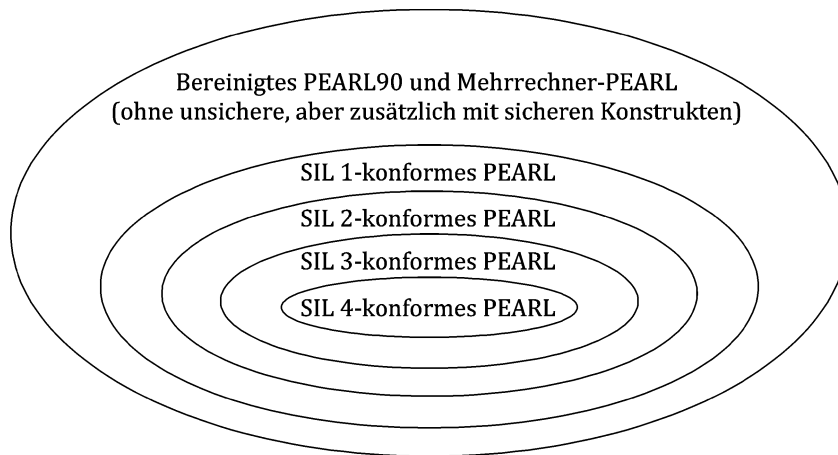


Bild 1 — Teilmengen der Echtzeitprogrammiersprache PEARL

Um den Anforderungen von SIL1 zu genügen, sind in der entsprechenden Sprachteilmenge Konstrukte nicht mehr zugelassen, deren Anwendung zu nichtdeterministischem Verhalten führt, denn nur voll deterministisches Systemverhalten kann letztendlich die sicherheitstechnische Abnahme programmgesteuerter Geräte für sicherheitskritische Aufgaben ermöglichen. Vorhersehbarkeit des Systemverhaltens ist von zentraler Bedeutung für den Echtzeitbetrieb. Sie ergänzt die Rechtzeitigkeitsforderung, da letztere nur dann garantiert werden kann, wenn das Systemverhalten exakt vorhersehbar ist, und zwar sowohl in der Zeit als auch bzgl. der Reaktionen auf externe Ereignisse. Auf der Sicherheitsintegritätsstufe SIL1 werden Sprach-elemente zur Festsetzung von Fertigstellungsfristen von Prozessabarbeitungen, zur Behandlung von Fristüberschreitungen, zur Begrenzung der Anzahl von Schleifendurchläufen, zur Zeit- und Ereignis-überwachung, zur Unterstützung der Abschätzung von Prozesslaufzeiten und Anwendung zeitgerechter Zuteilungsalgorithmen, zur allmählichen Leistungsabsenkung im Fehlerfall, zur Handhabung transients Überlast sowie zur Entdeckung und Behandlung von Fehlern mittels Diversität und allmählicher Leistungsabsenkung eingeführt. Die nur unsicher zu handhabenden Semaphore und Bolts werden durch neue Synchronisationskonstrukte zum Schutz von Betriebsmitteln ersetzt, die die Möglichkeiten bieten, maximale Wartezeiten vor kritischen Abschnitten und maximale Aufenthaltszeiten in diesen anzugeben, entsprechende Ausnahmen zu behandeln und Systemverklemmungen zu vermeiden.

Für den Einsatz von PEARL in Umgebungen, die den Anforderungen nach SIL2 genügen müssen, wird der Gebrauch dynamischer Sprachelemente wie variabler Felddimensionen und rekursiver Prozeduraufrufe zugunsten von Verlässlichkeit unterbunden, denn diese dynamischen Konstrukte führen i. A. zu Unvorhersagbarkeit im Hinblick auf Kapazität und Zeitanforderungen, was die Arbeit eines zeitgerechten Zuteilungsalgorithmus unmöglich macht.

Speicherprogrammierbarere Steuerungen spielen in der Automatisierungstechnik eine große Rolle. Für sie lässt sich auf der Grundlage vorprogrammierter, sicherheitszertifizierter Funktionsblockbibliotheken mit Hilfe der Funktionsplansprache nach DIN EN 61131-3 leicht verifizierbare Software für sicherheitskritische Anwendungen erstellen. Die der Sicherheitsintegritätsstufe SIL3 zugeordnete Teilmenge von PEARL erlaubt, Funktionspläne textuell aufzustellen. Die Teilmenge kennt i. W. nur noch Prozeduraufrufe und Parameter-übergaben, Schleifen sind z. B. nicht mehr erlaubt. Die Norm DIN EN 61131-3 definiert weiterhin die spezielle, unter Programmiersprachen einzigartige Ablaufplansprache zur Partitionierung von Programm-organisationseinheiten, d. h. Programmen und Funktionsblöcken, in Schritte und deren Verbindung entlang gerichteter Kanten mittels Transitionen. Mit jedem Schritt ist eine Menge von Aktionen und mit jeder Transition eine Übergangsbedingung assoziiert. Deshalb werden in 9.7 zusätzliche Sprachkonstrukte für PEARL zur Formulierung sequentieller Ablaufsteuerungen definiert, die aus Sicherheitsgründen die Möglichkeit paralleler Abläufe explizit ausschließen.

Die der höchsten Sicherheitsintegritätsstufe SIL4 zugeordnete Teilmenge von PEARL umfasst in dieser Norm neu eingeführte Konstrukte zur Formulierung von Ursache-Wirkungstabellen. Diese eignen sich besonders gut für die sicherheitsgerichtete Programmierung, weil sie streng, aber leicht nachvollziehbar und mit höchster Vertrauenswürdigkeit verifiziert werden können.

5 Sprachteilmengen für die Sicherheitsintegritätsstufen SIL0 bis SIL4

Die ineinander geschachtelten Sprachteilmengen für jede der Sicherheitsanforderungsklassen SIL1 bis SIL4 nach DIN EN 61508 werden in Tabelle 1 dergestalt gebildet, dass die Verwendung weniger sicherer Sprachkonstrukte ausgehend von Anwendungen ohne Sicherheitsrelevanz („SIL0“) mit steigendem Sicherheitsintegritätsniveau zunehmend restriktiver gehandhabt wird. Mit diesem Ansatz wird vermieden, für jede Sicherheitsstufe eine neue Sprache lernen zu müssen, und Übersetzer können prüfen, ob Programme vorgegebene Sicherheitsauflagen erfüllen. Das Prinzip, Teilmengen einer Sprache für kritische Anwendungen zu definieren, gestattet es, Programme nach bestimmten Sicherheitsanforderungen zu entwickeln und Code für sicherheitskritische und -unkritische Systemteile nahtlos miteinander zu verbinden. Je sicherheitskritischer ein System ist, desto restriktivere Methoden sind einzusetzen.

Tabelle 1 — Definition sicherheitsgerichteter Sprachteilmengen

Anweisung/Klausel	SIL0	SIL1	SIL2	SIL3	SIL4
GOTO, EXIT	+ ^a	— ^b	—	—	—
(bedingte) Ausdrücke und Zuweisungen	+	+	+	—	—
bedingte Anweisungen und Anweisungsauswahl	+	+	+	—	—
Angabe physikalischer Einheiten	+	+	+	+	+
Ursache-Wirkungstabellen	+	+	+	+	+
sequentielle Ablaufpläne	+	+	+	+	—
Synchronisierung mit SEMA- und BOLT-Variablen	+	—	—	—	—
Synchronisierung mit LOCK und TIMEOUT-Klausel	+	+	+	—	—
Verwendung interner Signale	+	+	+	—	—
Verwendung von Unterbrechungssignalen	+	+	+	—	—
Verwendung von Tasks	+	+	+	—	—
mit Prioritäten	+	+	—	—	—
mit Fristenangaben und Zeitüberwachung	+	+	+	—	—
(Funktions-) Prozeduraufrufe	+	+	+	+	—
Wiederholungen	+	+	—	—	—
mit MAXLOOP-Klausel	+	+	+	—	—
Verwendung von Zeigern und Referenzen	+	—	—	—	—
PUT/GET, WRITE/READ, CONVERT	+	+	+	—	—
TAKE/SEND	+	+	+	+	+
verteilte Systeme	+	+	+	+	+
dynamische Rekonfiguration	+	+	+	+	—
Botschaftenaustausch	+	+	+	—	—
^a zugelassen ^b nicht erlaubt					

6 Regeln zum Aufbau von PEARL-Sprachformen

6.1 Format

Ein PEARL-Programm kann formatfrei geschrieben werden; es muss insbesondere nicht darauf geachtet werden, dass eine Anweisung in einer bestimmten Spalte beginnt.

Alle Elemente eines PEARL-Programms werden aus Zeichen des Zeichensatzes nach 6.1 erstellt. Allerdings dürfen Zeichenkettenkonstanten und Kommentare jedes Zeichen enthalten, das ISO/IEC 646:1991 (siehe DIN 66003) und nationale Varianten von ISO/IEC 646 zulassen.

6.2 Zeichensatz

Der Zeichensatz von PEARL ist eine Teilmenge des 7-Bit-Zeichensatzes nach ISO/IEC 646:1991 (siehe DIN 66003). Er enthält

- a) die Großbuchstaben A bis Z;
- b) die Kleinbuchstaben a bis z;
- c) die Ziffern 0 bis 9 und
- d) die Sonderzeichen
 - Unterstrich;
 - Zwischenraum (zur Verdeutlichung wird manchmal auch der Unterstrich _ benutzt);
 - ! Ausrufungszeichen (Beginn des Zeilenkommentars);
 - ' Apostroph;
 - (runde Klammer auf;
 -) runde Klammer zu;
 - , Komma;
 - . Punkt;
 - ; Semikolon;
 - : Doppelpunkt;
 - + Pluszeichen (z. B. für Addition, Vorzeichen);
 - Minuszeichen (z. B. für Subtraktion, Vorzeichen);
 - * Stern (z. B. für Multiplikation);
 - / Schrägstrich (z. B. für Division);
 - = Gleichheitszeichen (z. B. für Zuweisung);
 - < Kleiner-Zeichen;
 - > Größer-Zeichen;
 - [Strukturklammerungszeichen auf;
 -] Strukturklammerungszeichen zu;
 - \ Rückschrägstrich (für Steuerzeichen in Zeichenketten).

Die folgenden Zeichenkombinationen werden jeweils als eine Einheit (zusammengesetztes Symbol) interpretiert:

- ** Exponentiationssymbol
- /* Kommentar-Beginn
- */ Kommentar-Ende
- // Symbol für ganzzahlige Division
- == Gleich-Symbol
- /= Ungleich-Symbol
- <= Kleiner-Gleich-Symbol
- >= Größer-Gleich-Symbol

<> Zyklisches-Schieben-Symbol

>< Verkettungssymbol

'\ ' Einleitung einer Steuerzeichensequenz in Zeichenkettenkonstanten

' ' Ende einer Steuerzeichensequenz in Zeichenkettenkonstanten

Falls auf dem Gerät für die Programm Niederschrift nicht alle Symbole zur Verfügung stehen, können folgende Zeichenfolgen alternativ benutzt werden:

LT für <

GT für >

EQ für ==

NE für /=

LE für <=

GE für >=

CSHIFT für <>

CAT für ><

(/ für [

/) für]

6.3 Grundelemente

6.3.1 Aufbau

Ein PEARL-Programm wird aus folgenden Grundelementen aufgebaut:

e) Bezeichner;

f) Konstanten;

g) Trennzeichen (das sind Sonderzeichen und zusammengesetzte Symbole) und

h) Kommentare.

Auf die Zeichenfolge für Bezeichner und Konstanten müssen Trennzeichen oder Kommentare folgen.

6.3.2 Bezeichner

Bezeichner werden zur Bildung von Namen für Objekte (z. B. Zahlenvariablen, Prozeduren) benutzt. Sie bestehen jeweils aus einer Folge von Buchstaben (groß und klein), dem Unterstrich und/oder Ziffern; diese Folge muss mit einem Buchstaben beginnen. PEARL unterscheidet bei Bezeichnern zwischen großen und kleinen Buchstaben, d. h. ventil, VENTIL und Ventil bezeichnen verschiedene Objekte:

BEISPIEL Zaehler_1, DISPO, warten

Einige Wörter haben an festgelegten Stellen im PEARL-Programm eine spezifische Bedeutung; diese Wörter werden Schlüsselwörter genannt. Die Wörter BIT oder GOTO sind z. B. solche Schlüsselwörter. Der Anhang enthält eine Liste aller Schlüsselwörter. Sie dürfen nicht als Bezeichner benutzt werden und müssen immer in Großbuchstaben geschrieben sein.

6.3.3 Konstanten

Konstanten sind ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern. Sie werden zusammen mit den entsprechenden Variablen in 7.6 beschrieben.

6.3.4 Kommentare

Kommentare dienen zur Erläuterung eines Programms und sind ohne Bedeutung für den Programmablauf. Es gibt zwei Arten von Kommentaren: Die eine Art kann sich über mehrere Zeilen erstrecken und wird durch die zusammengesetzten Symbole '/*' und '*/' eingeklammert. Innerhalb dieser Klammern können beliebige Zeichen mit Ausnahme des zusammengesetzten Symbols '*/' für Kommentarende vorkommen.

Die andere Art, der Zeilenkommentar, beginnt mit einem Ausrufungszeichen '!' und wird durch das Zeilenende beendet.

Kommentare dürfen überall eingefügt werden, wo Leerzeichen zulässig sind.

BEISPIEL /* Dieser Kommentar wird durch das Zeilenende
 nicht beendet */
 ! Dieser Kommentar ist auf eine Zeile begrenzt

6.4 Aufbau von Sprachformen

In den nachfolgenden Abschnitten werden die in PEARL zulässigen Sprachformen beschrieben. Damit diese Beschreibungen genau und möglichst kompakt sind, werden über die verbale Formulierung hinaus einige formale Möglichkeiten benötigt:

Jede Sprachform hat einen Namen, unter dem sie mit Hilfe des (Meta-)Symbols "::=" definiert wird:

Name-der-Sprachform ::=Definition-der-Sprachform

BEISPIEL Großbuchstabe ::=A oder B oder ... oder Z
 Ziffer ::=0 oder 1 oder ... oder 9

Wie dieses Beispiel zeigt, kann die Definition einer Sprachform Elemente enthalten, die beim Aufbau der Sprachform alternativ zueinander angegeben werden. Zur Abkürzung werden die alternativ möglichen Angaben künftig durch das Symbol "|" getrennt:

BEISPIEL Buchstabe ::=A | B | ... | Z | a | b | ... | z
 Ziffer ::=0 | 1 | 2 | ... | 9

Soll ein Element beliebig oft, aber mindestens einmal, auftreten dürfen, wird es mit drei Punkten versehen.

BEISPIEL einfache-ganze-Zahl ::=Ziffer ...

Um auszudrücken, dass ein Element beim Aufbau der Sprachform optional ist, wird es in eckigen Klammern "[...] " angegeben (Im Zweifelsfall werden die Strukturklammerungszeichen fett gedruckt).

BEISPIEL Bezeichner ::=Buchstabe [{ Buchstabe | Ziffer | } ...]

Hier wurden bereits zwei weitere Regeln benutzt: Die Definition einer Sprachform darf wieder Namen von Sprachformen enthalten; außerdem werden die geschweiften und eckigen Klammern auch dazu benutzt, Elemente zu neuen Elementen zusammenzufassen. So ist das letzte Beispiel dem folgenden äquivalent:

BEISPIEL Bezeichner ::=Buchstabe [Buchstabe | Ziffer | _] ...

Zur Beschreibung von Listen, deren Elemente durch ein bestimmtes Symbol getrennt sind, werden bei der Definition der entsprechenden Sprachform das Listenelement und das Trennsymbol in der Form

Listenelement [Trennsymbol Listenelement] ...

angegeben.

BEISPIEL Bezeichnerliste ::= Bezeichner [, Bezeichner] ...

Zum besseren Verständnis oder zur genaueren Beschreibung der Definition einer Sprachform werden häufig Elemente mit einem erläuternden oder einschränkenden Kommentar versehen, der von dem Element durch das Symbol "\$" getrennt wird.

BEISPIEL Geräteliste ::= Bezeichner\$Gerät [, Bezeichner\$Gerät] ...

7 Programm

7.1 Aufbau

Syntax: Die allgemeine Form eines PEARL-Programms lautet:

PEARL-Programm ::= Modul ...

Semantik: Ein PEARL-Programm setzt sich aus einem oder mehreren Teilen, sogenannten Modulen, zusammen, die unabhängig voneinander übersetzt werden.

7.2 Module

Syntax:

Modul ::=

MODULE [(Bezeichner\$des-Moduls)] [Safeguard-Vereinbarung] ;
 {Systemteil [Problemteil] } | Problemteil |
 { { Systemteil-für-CETABLE [Problemteil-für-CETABLE] } |
 Problemteil-für-CETABLE }
MODEND;

Systemteil ::=

SYSTEM ; [Benutzernamen-Vereinbarung ...]

Problemteil ::=

PROBLEM [Safeguard-Vereinbarung] ; [{ Deklaration | Spezifikation | Identifikation } ...]

Semantik: Jedes Modul besteht aus einem Systemteil und/oder einem Problemteil.

Erläuterungen: Im Systemteil werden die Verbindungen des projektierten Rechners mit den Elementen des technischen Prozesses (Messwertgeber, Stellglieder usw.) und der Standardperipherie (Tastatur, Monitor, Drucker, Massenspeicher usw.) beschrieben. Dabei kann der Programmierer z. B. den Eingängen eines Unterbrechungswerks und den im Problemteil in E/A-Anweisungen angesprochenen Peripherieelementen frei wählbare Namen zuordnen, um diese rechnerunabhängigen Namen im Problemteil verwenden zu können.

Im Problemteil wird der Algorithmus zur Lösung der gestellten Automatisierungsaufgabe beschrieben. Hierzu kann der Programmierer folgende Objekte durch Deklaration, Spezifikation oder Identifikation vereinbaren:

- a) Variablen und Konstanten für ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Zeitdauern, Uhrzeiten, Referenzen;
- b) Sprungmarken;
- c) Prozeduren für mehrfach auftretende Teilaufgaben;

- d) Tasks für die zeitliche parallele Abarbeitung von Aufgaben;
- e) Blöcke für die Strukturierung von Prozeduren und Tasks;
- f) Unterbrechungssignale;
- g) Signale;
- h) Synchronisierungsvariablen (Sema- und Bolt-Variablen) sowie
- i) Datenstationen und Formate für die Ein/Ausgabe.

Die erforderlichen Anweisungen werden in den Prozeduren und Tasks zusammen mit weiteren lokalen Vereinbarungen, die nur dort benötigt werden, angegeben. Dabei gilt allgemein, dass Objekte erst dann (in Anweisungen) benutzt werden dürfen, wenn sie vereinbart worden sind.

Objekte werden auf Modulebene, d. h. außerhalb von Prozeduren und Tasks, oder in Prozeduren oder in Tasks vereinbart. Auf Modulebene vereinbarte Objekte sind im ganzen Modul bekannt und können von jeder Task und Prozedur des Moduls unter Angabe des Bezeichners benutzt oder ggf. geändert werden. Ein in einer Task oder Prozedur vereinbartes Objekt ist nur in der betreffenden Task oder Prozedur bekannt und kann nur dort benutzt oder ggf. geändert werden.

BEISPIEL (Modulstruktur)

MODULE ;

SYSTEM ;

Beschreibung der Verbindungen und Vereinbarung von Namen für die Peripherieelemente

PROBLEM ;

Vereinbarung von Konstanten und Variablen

Vereinbarung von Unterbrechungssignalen

Vereinbarung von Datenstationen

Vereinbarung einer Task

Deklaration lokaler Konstanten und Variablen

Deklaration lokaler Prozeduren

Anweisungen

Vereinbarung einer Prozedur

Deklaration lokaler Konstanten und Variablen

Deklaration lokaler Prozeduren

Anweisungen

...

MODEND ;

Nach dem Laden eines PEARL-Programms startet das PEARL-Laufzeitsystem automatisch alle Tasks, die mit dem Attribut MAIN gekennzeichnet sind (sie werden lauffähig). Alle mit MAIN versehenen Tasks müssen im selben Modul deklariert sein.

BEISPIEL (Attribut MAIN)

MODULE (Main) ;

SYSTEM ;

PROBLEM ;

Start: **TASK MAIN ;**

! Einplanungen anderer Tasks

END ; ! Start

```
Messen : TASK PRIO 1 ;  
    ! Taskrumpf  
    END ; ! Messen  
  
MODEND ;
```

Nach dem Laden wird die Task Start automatisch gestartet.

7.3 Vereinbarungen

7.3.1 Deklaration

Syntax:

```
Deklaration ::=  
    Längen-Vereinbarung  
    | Typ-Vereinbarung  
    | Variablen-Deklaration  
    | Format-Deklaration  
    | Benutzer-SIGNAL-Deklaration  
    | Prozedur-Deklaration  
    | Task-Deklaration  
    | Ablaufplan  
    | Ursache-Wirkungstabelle
```

Semantik: Die Deklaration dient dazu, ein Objekt und seinen Namen einzuführen, d. h. bei Auswertung der Deklaration wird Speicherplatz für das Objekt angelegt, und es kann ab diesem Zeitpunkt unter dem in der Deklaration angegebenen Namen darauf zugegriffen werden.

Tabelle 2 — Zulässigkeit von Deklarationen

Objekt	Deklaration möglich auf/in			
	Modulebene	Task	Prozedur	Block
Variablen, Konstanten	x	x	x	x
Sprungmarken	—	x	x	x
Prozeduren	x	x	x	—
Task	x	—	—	—
Block	—	x	x	x
Unterbrechungssignal	x	—	—	—
Signal	x	—	—	—
Sema-, Bolt-Variablen	x	—	—	—
Dations, Formate	x	—	—	—
Typen	x	x	x	x

Auf Modulebene oder in einer Prozedur oder in einer Task darf ein Objekt nur einmal deklariert werden. Ist ein Bezeichner X sowohl auf Modulebene als auch in einer Prozedur oder Task als Objekt deklariert, so sind damit zwei verschiedene Objekte eingeführt: In der betreffenden Prozedur oder Task bezieht man sich unter dem Bezeichner X auf das dort (lokal) deklarierte Objekt, außerhalb der Prozedur oder Task auf das auf Modulebene deklarierte Objekt.

Innerhalb von Prozeduren und Tasks dürfen

- a) Tasks;
- b) Unterbrechungssignale;
- c) Signale;
- d) Synchronisierungsvariablen;
- e) Datenstationen;
- f) Formate

nicht deklariert werden.

BEISPIEL (Zulässigkeit von Deklarationen)

PROBLEM ;

DECLARE x FLOAT ; ! 1. Deklaration auf Modulebene

DECLARE x FIXED ; ! 2. Deklaration auf Modulebene (falsch)

P: **PROCEDURE ;**

DECLARE x FIXED ; ! Deklaration in der Prozedur P (zulässig)

...

x := 3 ; ! Zuweisung an die lokale Variable x

...

END ; ! P

T: **TASK ;**

...

x := 5 ; ! Zuweisung an die auf Modulebene deklarierte Variable x

END ; ! T

7.3.2 Spezifikation (SPC) und Identifikation (SPC IDENT)

Syntax:

Spezifikation ::=

{ SPC | SPECIFY } Spezifikationsangabe [, Spezifikationsangabe] ...

Identifikation ::=

{ SPECIFY | SPC } Bezeichner [Zuweisungsschutz] Typ Identifikationsattribut

[Safeguard-Vereinbarung];

Identifikationsattribut ::=

IDENT (Name\$Objekt)

Semantik: Mit einer Spezifikation bezieht man sich auf ein bereits deklariertes Objekt. Dies ist sinnvoll für Objekte, die in einem Modul deklariert und in anderen Modulen benutzt werden sollen (siehe 7.4). Mit einer Identifikation können zusätzliche Namen für bereits vereinbarte Objekte eingeführt werden.

Der bei der Identifikation angegebene Typ muss dem Typ des benannten Objekts entsprechen.

BEISPIEL Das Objekt x vom Typ FIXED (15) soll xx als zweiten Namen erhalten, d. h. das Objekt x mit dem Namen xx *identifiziert* werden können:

```
PROBLEM ;  
...  
DECLARE x FIXED ;  
...  
SPECIFY xx FIXED IDENT (x) ;  
...  
xx ::= 7 ;      ! Zuweisung an das Objekt x
```

7.4 Blöcke

Syntax:

```
Block::=  
  BEGIN  
    [ Deklaration ... ]  
    [ Anweisung ... ]  
  END [ Bezeichner$Block ] ;
```

Semantik: Ein Block ist eine Zusammenfassung von Deklarationen und Anweisungen: Blöcke dürfen nur innerhalb von Tasks und Prozeduren auftreten, dort allerdings auch ineinander geschachtelt. Der Eintritt in einen Block erfolgt mit der Ausführung von BEGIN. Verlassen wird ein Block durch Überlaufen des zugehörigen END oder durch eine Verzweigung zu einer Anweisung außerhalb des Blocks, z. B. durch eine Exit-Anweisung. Sprünge in einen Block hinein sind nicht erlaubt.

Innerhalb von Blöcken dürfen keine Prozeduren deklariert werden!

Erläuterungen: Den in einem Block deklarierten (lokalen) Objekten wird erst bei Eintritt in den Block Speicherplatz zugewiesen; dieser wird bei Verlassen des Blocks wieder freigegeben. Wie Tasks, Prozeduren und Wiederholungen können Blöcke also Objekte dynamisch einführen und entfernen und bieten somit die Möglichkeit, den zur Verfügung stehenden Speicherplatz mehrfach zu nutzen.

Die Lebensdauer eines Objekts ist die (Ausführungs-) Zeit zwischen der Auswertung seiner Deklaration und der Ausführung des Endes des Blocks (bzw. der Wiederholung, der Prozedur, der Task oder des Moduls), in dem (bzw. in der) die Deklaration erfolgt.

Als Gültigkeitsbereich eines Objekts werden alle Teile des Programms bezeichnet, in denen das Objekt benutzt werden kann. Folgende Regeln sind zu beachten:

- a) Ein auf Modulebene vereinbartes Objekt ist auf Modulebene und in allen Tasks und Prozeduren dieses Moduls sowie in allen eingeschachtelten Prozeduren, Blöcken und Wiederholungen mit folgender Ausnahme benutzbar: Der Gültigkeitsbereich wird eingeschränkt, wenn in einer der Tasks oder Prozeduren ein anderes Objekt unter demselben Namen deklariert wird.
- b) Ein in einer Task, Prozedur, Wiederholung oder einem Block vereinbartes Objekt ist in dieser Task, Prozedur, Wiederholung oder diesem Block und allen darin eingeschachtelten Prozeduren, Wiederholungen und Blöcken benutzbar mit folgender Ausnahme: Der Gültigkeitsbereich wird eingeschränkt, wenn in einer der eingeschachtelten Prozeduren, Wiederholungen oder Blöcke ein anderes Objekt unter demselben Namen deklariert wird.

Blöcke werden dazu benutzt, Task- oder Prozedurrümpfe zu strukturieren sowie um Gültigkeitsbereiche und Lebensdauern von PEARL-Objekten zu beeinflussen.

Blöcke können nach dem END mit Bezeichnern versehen werden. Dadurch können geschachtelte Blöcke mittels der Exit-Anweisung gezielt verlassen werden.

BEISPIEL (Lebensdauer von Objekten)

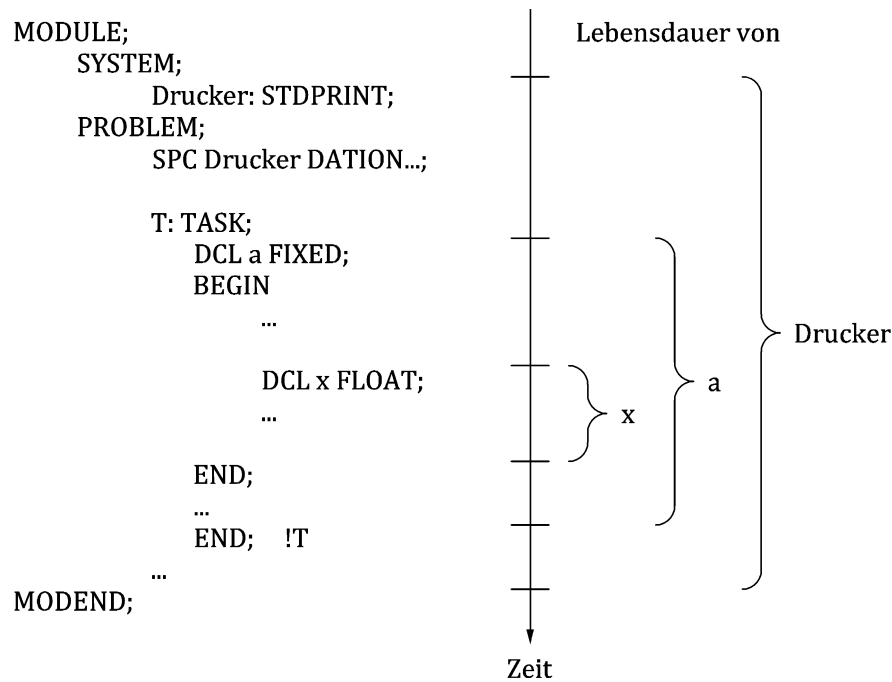


Bild 2 — Lebensdauer von Objekten

BEISPIEL (Gültigkeitsbereich von Objekten)

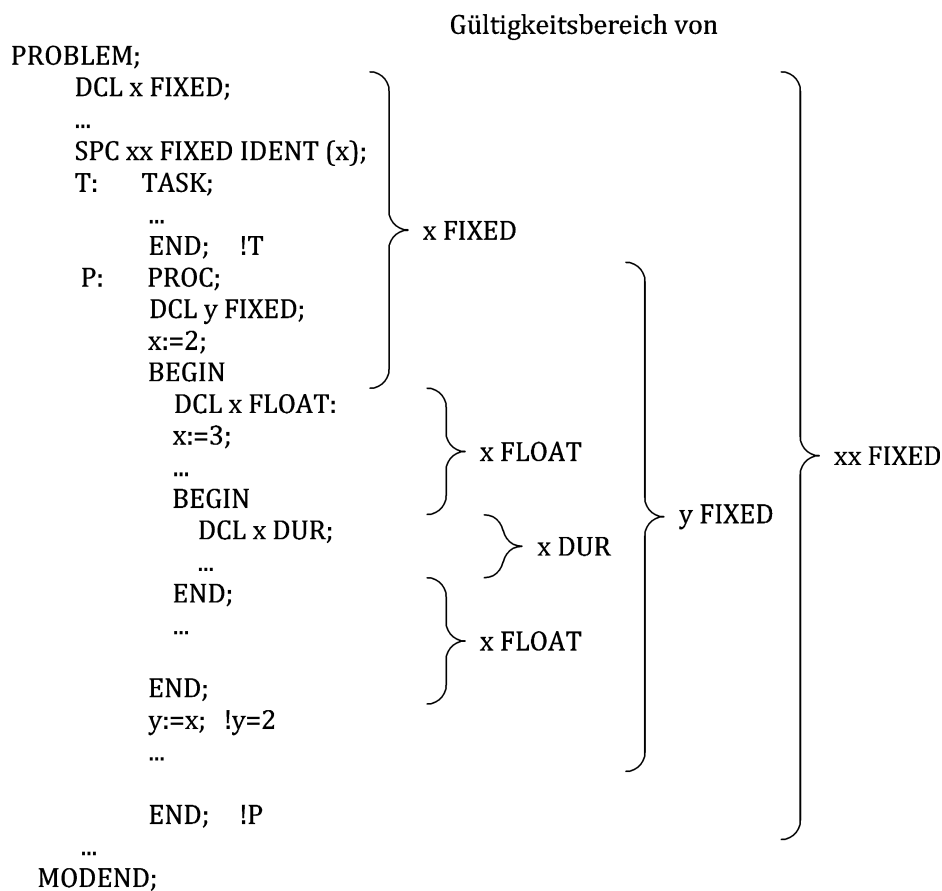


Bild 3 — Gültigkeitsbereich von Objekten

7.5 Bezüge zwischen Modulen

Syntax: Die allgemeine Form des Global-Attributs lautet:

Global-Attribut ::=
GLOBAL (Bezeichner\$eines-Moduls)

Semantik: Die Gültigkeit auf Modulebene deklarierter Objekte kann auf andere Module erweitert werden, indem diese Objekte mit dem GLOBAL-Attribut deklariert und in den anderen Modulen auf Modulebene mit dem GLOBAL-Attribut spezifiziert werden.

Bei der Spezifikation eines Objekts müssen alle Attribute aus seiner Deklaration mit Ausnahme einer angegebenen Priorität, Genauigkeit oder Länge übernommen werden. In letzterem Ausnahmefall wird für den Bereich des Programms, in dem die Spezifikation gilt, die in der entsprechenden Längenvereinbarung definierte Genauigkeit oder Länge eingesetzt.

Alle im Systemteil eines bestimmten Moduls aufgeführten Datenstationen, Unterbrechungssignale und Signale gelten als (implizit) mit dem GLOBAL-Attribut deklariert. Deshalb werden sie in den Problemteilen des Programms nur noch spezifiziert; dabei kann das GLOBAL-Attribut im Problemteil desselben Moduls entfallen, in allen anderen Modulen muss es jedoch angegeben werden.

Der gegebenenfalls angegebene Modulbezeichner des Moduls hat nur dokumentarischen Wert.

Erläuterungen: Bezüge zu in anderen Implementierungssprachen geschriebenen Modulen sind unter Einhaltung der im Benutzerhandbuch angegebenen Regeln möglich.

BEISPIEL (Variablen)

```
MODULE (a);  
PROBLEM ;  
...  
  DCL x FIXED GLOBAL ;  
...  
  x := 2 ;  
...  
MODEND ;
```

```
MODULE (b);  
PROBLEM ;  
...  
  SPC x FIXED GLOBAL (a);  
...  
  x := 3 ;  
...  
MODEND ;
```

BEISPIEL (Prozedur, Task)

```
MODULE ;  
PROBLEM ;  
  T:TASK PRIO 3 GLOBAL ;  
                                ! Taskkörper  
  END ; ! T  
  
  P: PROC ( A (,) FIXED IDENT ) GLOBAL ;  
                                ! Prozedurkörper  
  END ; ! P  
  
...  
MODEND ;
```

```
MODULE ;  
PROBLEM ;  
  SPC T TASK GLOBAL,  
    P 9 (,) FIXED IDENT ) GLOBAL ;  
  INIT: TASK ;  
    DCL TAB (10, 20) FIXED ;  
  ...  
    CALL P (TAB) ;  
  ...  
    ACTIVATE T ;  
  END ; ! INIT  
  
...  
MODEND ;
```

7.6 Auswahl des Sicherheitsintegritätsniveaus

Mittels der SAFEGUARD-Anweisung wird die auf einem gewünschten Sicherheitsintegritätsniveau zugelassene Sprachteilmenge ausgewählt:

Syntax:

Safeguard-Vereinbarung ::=
 SAFEGUARD Sicherheitsstufe

Sicherheitsstufe ::=
 { **NONE** | **SIL1** | **SIL2** | **SIL3** | **SIL4** }

Semantik: Eine SAFEGUARD-Klausel darf innerhalb des Geltungsbereichs einer anderen angegeben werden, solange damit das Sicherheitsintegritätsniveau eines Systems nicht reduziert wird. Die Gültigkeitsgrenzen eines SAFEGUARD-Bereichs dürfen nicht überschritten werden. Dies bedeutet, dass sich eine aufgerufene Prozedur innerhalb eines anderen, gleichwertigen SAFEGUARD-Bereichs befinden muss. Die einzelnen Sicherheitsbereiche sind wie folgt definiert:

SAFEGUARD 0: Auf dieser Ebene sind alle Anweisungen erlaubt und fast alle einschränkenden Klauseln wie MAXLOOP oder TIMEOUT optional. Dies ist das allgemeine, aber nicht gesicherte Niveau.

SAFEGUARD 1: Auf diesem Niveau sind Zeitbeschränkungen wie MAXLOOP und TIMEOUT obligatorisch. Auf gemeinsam genutzte Datenobjekte muss innerhalb von LOCK-Anweisungen zugegriffen werden.

SAFEGUARD 2: Jede Prozedur muss einen Satz von Vor- und Nachbedingungen sowie Invarianten haben.

SAFEGUARD 3: Auf diesem Niveau sind keine Schleifenanweisungen mehr erlaubt. Einfache Variablen dürfen nicht lokal deklariert werden. Es ist allein möglich, auf bereits formal verifizierte Module zuzugreifen.

SAFEGUARD 4: Auf diesem Niveau sind alle Anweisungen zur Steuerung des sequentiellen Ablaufs verboten. Abläufe können ausschließlich mittels Ursache-Wirkungstabellen beschrieben werden.

7.7 Variablen und Konstanten

7.7.1 Vereinbarung von Variablen (DCL)

Syntax: Die allgemeine Form der Deklaration skalarer Variablen lautet:

Skalare-Variablen-Deklaration ::=
 { **DECLARE** | **DCL** } Variablen-Angabe [, Variablen-Angabe] ... ;

Variablen-Angabe ::=
 Bezeichner-Angabe [Zuweisungsschutz] Typ-Attribut [Global-Attribut]
 [Initialisierungsattribut] [PhysikalischeEinheit]

Bezeichner-Angabe ::=
 Bezeichner | (Bezeichner [, Bezeichner] ...)

Typ-Attribut ::=
 einfacher-Typ | Typ-Referenz | Bezeichner\$für-Typ

einfacher-Typ ::=
 Typ-ganze-Zahl | Typ-Gleitpunktzahl |
 Typ-Bitkette | Typ-Zeichenkette |
 Typ-Uhrzeit | Typ-Dauer

Die allgemeine Form der Spezifikation skalarer Variablen lautet:

Skalare-Variablen-Spezifikation ::=
 { SPECIFY | SPC } Variablen-Angabe-S [' Variablen-Angabe-S] ... ;
Variablen-Angabe-S ::=
 Bezeichner-Angabe [Zuweisungsschutz] Typ-Attribut
 { Global-Attribut | Identifikationsattribut } [PhysikalischeEinheit]

Semantik: Die von einem PEARL-Programm bei seiner Ausführung benutzten Daten (ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern) treten in Form von Konstanten oder als Werte von Variablen auf. Konstanten identifizieren sich durch ihre Schreibweise und behalten ihre Werte während des gesamten Programmablaufs bei. Die Werte von Variablen bezeichnen Daten, die sich während des Programmablaufs ändern können.

Der Wertebereich einer Variablen ist i. A. auf eine bestimmte Art von Daten, z. B. Bitketten, beschränkt, die den Typ der Variablen bestimmt (Bitkettenvariablen haben z. B. nur Bitketten als Werte.). Dieser Typ muss bei der Deklaration und Spezifikation einer Variablen zusammen mit ihrem Bezeichner angegeben werden.

Bei der Deklaration einer Variablen muss ihr Typ in einem Typ-Attribut angegeben werden; sollen verschiedene Variablen mit gleichem Typ-Attribut vereinbart werden, so kann dies in Form einer Liste und einmaliger Angabe des Typ-Attributs erfolgen. Verschiedene Vereinbarungen können in einer Deklaration formuliert werden, indem sie durch Kommata getrennt werden.

BEISPIEL (Angabe von Typen in Deklaration und Spezifikation)

DECLARE Schalter BIT ;	/* Deklaration einer Variablen Schalter vom Typ Bitkette der Länge 1 */
SPECIFY Status BIT (16) GLOBAL ;	/* Spezifikation einer globalen Variablen Status vom Typ Bitkette der Länge 16 */

BEISPIEL (Liste von Variablen unterschiedlichen Typs)

DECLARE x **FLOAT**,
 i **FIXED** ;

7.7.2 Ganze Zahlen (FIXED)

Syntax:

Typ-ganze-Zahl ::=
 FIXED [(Genauigkeit)]
Genauigkeit ::=
 ganze-Zahl-ohne-Genauigkeit\$größer-Null
ganze-Zahl ::=
 ganze-Zahl-ohne-Genauigkeit [(Genauigkeit)] [PhysikalischeEinheit]
ganze-Zahl-ohne-Genauigkeit ::=
 Ziffer ... | { 0 | 1 } ... B

Semantik: Variablen vom Typ ganze Zahl werden mit dem Typ-Attribut **FIXED** deklariert.

Die Genauigkeit legt die Anzahl der Bitstellen (ohne Vorzeichen) fest, in denen der jeweilige Wert der Variablen dargestellt wird. Fehlt die Genauigkeitsangabe, so wird die in einer Längenvereinbarung definierte Genauigkeit eingesetzt.

Ganze Zahlen können in Dezimal- oder Dualdarstellung geschrieben werden: Die Dezimaldarstellung einer ganzen Zahl besteht aus einer Folge von Ziffern. Die Dualdarstellung einer ganzen Zahl besteht aus einer Folge der Ziffern 0 und 1, die mit dem Zeichen B abgeschlossen wird.

Zudem kann für eine Zahlenkonstante die Genauigkeit ihrer Darstellung definiert werden, indem anschließend an die Zahlenkonstante die Anzahl der Bitstellen (ohne Vorzeichen) in Klammern vermerkt wird, mit der sie rechnerintern dargestellt werden soll.

Erläuterungen: Ist keine Genauigkeit angegeben, wird die Genauigkeit aus der Konstanten abgeleitet.

Gängige Genauigkeiten sind 15 oder 31.

BEISPIEL (Zahlenkonstanten vom Typ FIXED in Dezimal- und Dualdarstellung)

Dezimaldarstellung	Dualdarstellung
6	110B
123	1111011B

BEISPIEL (FIXED-Zahlenkonstante mit Genauigkeit)

: 123(31) Die ganze Zahl 123 wird in 31 Bitstellen (ohne Vorzeichen) dargestellt.

BEISPIEL (FIXED-Variablen-Deklaration mit und ohne Genauigkeitsangabe)

```
DCL Zaehler FIXED (31),  
  (i,j,k) FIXED ;  
...  
i := 2,;
```

7.7.3 Gleitpunktzahlen (FLOAT)

Syntax:

```
Typ-Gleitpunktzahl ::=  
  LOAT [ (Genauigkeit) ]  
  
Gleitpunktzahl ::=  
  Gleitpunktzahl-ohne-Genauigkeit [ ( Genauigkeit) ] [ PhysikalischeEinheit ]  
  
Gleitpunktzahl-ohne-Genauigkeit ::=  
  { Ziffer ... . [ Ziffer ... ] | . Ziffer ... } [ Exponent ]  
  | Ziffer ... Exponent  
  
Exponent ::=  
  E [ + | - ] Ziffer ...
```

Semantik: Variablen vom Typ Gleitpunktzahl (mit ganzen Zahlen oder Gleitpunktzahlen als Werten) werden mit dem Typ-Attribut FLOAT deklariert.

Konstanten für Gleitpunktzahlen können dargestellt werden als eine Folge von

- i) einem Punkt, einer ganzen Zahl und eventuell einem Exponenten zur Basis 10, wobei ein Exponent aus einer Folge des Zeichens E, eventuell einem Plus- oder Minuszeichen und einer ganzen Zahl besteht;

Beispiel: .123 (entspricht 0,123)
.123E2 (entspricht 12,3)
.123E-1 (entspricht 0,0123)

- ii) einer ganzen Zahl und der unter (i) angegebenen Folge;
Beispiel: 3.123E2 (entspricht 312,3)
- iii) einer ganzen Zahl, einem Punkt und eventuell einem Exponenten;
Beispiel: 3. (entspricht 3,0)
- iv) einer ganzen Zahl und einem Exponenten.
Beispiel: 3E-2 (entspricht 0,03)

Analog zu ganzen Zahlen können auch Gleitpunktzahlkonstanten mit Genauigkeit formuliert werden.

ANMERKUNG Gängige Genauigkeiten sind 23 bzw. 53.

BEISPIEL (FLOAT-Variablen-Deklaration mit und ohne Genauigkeitsangabe)

```
DCL (x, y, z) FLOAT ,  
    Koeff FLOAT (53) ;  
...  
x := 3.5; y := 1 ; Koeff := 3.14E-10 ;
```

7.7.4 Bitketten (BIT)

Syntax:

```
Typ-Bitkette ::=  
    BIT [ (Länge) ]  
  
Länge ::=  
    ganze-Zahl-ohne-Genauigkeit$größer-Null  
  
Bitkettenkonstante ::=  
    ' B1-Ziffer ... '{ B | B1 } | ' B2-Ziffer ... ' B2 | B3-Ziffer ... ' B3 | ' B4-Ziffer ... ' B4  
  
B1-Ziffer ::= 0 | 1  
B2-Ziffer ::= 0 | 1 | 2 | 3  
B3-Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  
B4-Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F
```

Semantik: Variablen für Bitketten werden mit dem Typ-Attribut BIT deklariert.

Länge gibt die Anzahl der Elemente einer Bitkette an. Fehlt die Längenangabe, so muss sie entweder in einer Längenvereinbarung definiert worden sein oder es wird als Länge 1 eingesetzt.

Eine Bitkettenkonstante kann binär (B1), in Form von Tetraden (B2), Oktaden (B3) oder in hexadezimaler Form (B4) angegeben werden.

Die Form Bi (i = 1, ..., 4) besteht aus einem Apostroph, einer Folge

- a) der Ziffern 0 und 1 im Fall B1,
- b) der Ziffern 0 bis 3 im Fall B2,
- c) der Ziffern 0 bis 7 im Fall B3,
- d) der Ziffern 0 bis 9 und Buchstaben A bis F im Fall B4

sowie einem Apostroph und der entsprechenden Angabe B1 oder B2 oder B3 oder B4.

Anstatt B1 kann kürzer B geschrieben werden.

Tabelle 3 — Zuordnung zwischen der Binärform und den anderen Formen

B2	B1	B3	B1	B4	B1
0	00	0	000	0	0000
1	01	1	001	1	0001
2	10	2	010	2	0010
3	11	3	011	3	0011
4	100	4	0100		
5	101	5	0101		
6	110	6	0110		
7	111	7	0111		
8	1000				
9	1001				
A	1010				
B	1011				
C	1100				
E	1110				
F	1111				

BEISPIEL (Entsprechungen von Bitkettenkonstanten)

'110010100111'B1	entspricht
'302213'B2	entspricht
'6247'B3	entspricht
'CA7'B4	

BEISPIEL (Längenangaben bei Variablendeklaration):

```
DCL    X_Koord BIT (2) ,  
        Y_Koord BIT (8) ;
```

...

```
X_Koord ::= '01'B ;
```

```
Y_Koord ::= 'A9'B4 ;
```

7.7.5 Zeichenketten (CHARACTER)

Syntax:

```
Typ-Zeichenkette ::=  
    { CHARACTER | CHAR } [(Länge)]
```

```
Zeichenkettenkonstante ::=  
    '{ Zeichen-außer-Apostroph | " | Kontrollzeichen-Sequenz } ... '
```

```
Zeichen-außer-Apostroph ::=  
    Ziffer | Buchstabe | l_l + l - l * l / l \ l ( l ) | [ ] | : | . | ; | , | = | < | > | ! |  
    /* weitere druckbare Zeichen des Maschinensatzes */
```

```
Kontrollzeichen-Sequenz ::=  
    '\ { B4-Ziffer B4-Ziffer } ... \'
```

Semantik: Variablen für Zeichenketten werden mit dem Typ-Attribut CHARACTER deklariert.

Länge gibt die Anzahl Zeichen an. Fehlt die Längenangabe, so muss sie entweder in einer Längenvereinbarung (siehe 7.6.6) definiert worden sein oder es wird die Länge 1 eingesetzt.

Eine Zeichenkettenkonstante besteht aus einem Apostroph, einer Folge von beliebigen Zeichen (außer Apostroph) und einem Apostroph.

Soll jedoch die Zeichenkette einen Apostroph enthalten, so muss dieser durch zwei aufeinander folgende Apostrophe dargestellt werden.

Steuerzeichen können mit Hilfe der Umschaltssymbole " \" und " \" in Zeichenkettenkonstanten eingefügt werden. Die Steuerzeichen werden als Paare von Hexadezimalziffern angegeben.

BEISPIEL (Zeichenkettenkonstante ohne Apostroph)

'Stoerung Nr: '

BEISPIEL (Zeichenkettenkonstante mit Apostroph)

'Stoer' 'Nr: '

BEISPIEL (Zeichenkettenkonstanten mit Steuerzeichen)

- 1) 'Dieser String *\ 0D DA \" enthält zwei Steuerzeichen'
" \"1BV' Steuerzeichen am String-Anfang'
'Steuerzeichen am String-Ende' \"00\"
" \"00\" /* String bestehend aus einem einzigen Steuerzeichen */
- 2) 'Durch die Umschaltung auf die Steuerzeichensequenz\" 20
\" können sehr lange Zeichenkettenkonstanten\" 20
\" (unabhängig vom verwendeten Editor) erstellt werden.'

BEISPIEL (Variable vom Typ CHAR mit Längenangabe)

DCL Artikelbez **CHAR** (6) ;

...

Artikelbez ::= 'BCD/27' ;

7.7.6 Längenvereinbarung

Syntax:

Längenvereinbarung ::=

LENGTH { { **FIXED** | **FLOAT** } (Genauigkeit)
| { **BIT** | **CHARACTER** | **CHAR** } (Länge) }

Semantik: Mit der Längenvereinbarung werden die Genauigkeiten bzw. Längen für solche Zahlen- und Kettenobjekte definiert, deren Genauigkeiten bzw. Längen nicht durch die Schreibweise (bei Konstanten) oder die Deklaration (bei Variablen) festgelegt sind.

Für eine Längenvereinbarung gelten die gleichen Gültigkeitsregeln wie für Variablendeklarationen.

BEISPIEL (Gültigkeit von Längenvereinbarungen)

PROBLEM ;

LENGTH **FIXED**(15) ;

LENGTH **FLOAT**(53) ;

DCL A **FIXED** , /* A hat den Typ **FIXED**(15) */
X **FLOAT** , /* X hat den Typ **FLOAT**(53) */
Y **FLOAT**(23) ; /* Y hat den Typ **FLOAT**(23) */

...

7.7.7 Uhrzeiten (CLOCK)

Syntax:

Typ-Uhrzeit ::=
CLOCK

Uhrzeitkonstante ::=
Ziffer ... : Ziffer ... : Ziffer ... [. Ziffer ...]

Semantik: Variablen für Uhrzeiten werden mit dem Typ-Attribut **CLOCK** deklariert.

Eine Uhrzeitkonstante besteht aus einer ganzen Zahl für die Stundenangabe, einer ganzen Zahl zwischen 0 und 59 für die Minutenangabe und einer Gleitpunktzahl zwischen 0 und 59.999... für die Sekundenangabe – jeweils durch Doppelpunkt getrennt. Die Stundenangabe wird modulo 24 interpretiert.

BEISPIEL (Uhrzeitkonstanten)

11:30:00	bedeutet 11.30 Uhr
15:45:3.5	bedeutet 15.45 Uhr und 3,5 Sekunden
25:00:00	bedeutet 1 Uhr

BEISPIEL (Deklaration einer Uhrzeitvariablen)

DCL Zeit **CLOCK** ;
...
Zeit ::= 12:30:00 ;

7.7.8 Zeitdauern (DURATION)

Syntax:

Typ-Dauer ::=
DURATION | DUR

Dauerkonstante ::=
Stundenangabe [Minutenangabe] [Sekundenangabe]
| Minutenangabe [Sekundenangabe]
| Sekundenangabe

Stundenangabe ::=
ganze-Zahl-ohne-Genauigkeit **HRS**

Minutenangabe ::=
ganze-Zahl-ohne-Genauigkeit **MIN**

Sekundenangabe ::=
{ ganze-Zahl-ohne-Genauigkeit | Gleitpunktzahl-ohne-Genauigkeit } **SEC**

Semantik: Variablen für Zeitdauern werden mit dem Typ-Attribut **DURATION** deklariert.

Eine Dauerkonstante setzt sich aus einer Stundenangabe, Minutenangabe und Sekundenangabe zusammen, wobei einzelne dieser Angaben fehlen können. Eine Stundenangabe besteht aus einer ganzen Zahl und der Zeichenfolge **HRS**, eine Minutenangabe aus einer ganzen Zahl und der Zeichenfolge **MIN**, eine Sekundenangabe aus einer Gleitpunktzahl und der Zeichenfolge **SEC**.

Die ganzen Zahlen und Gleitpunktzahlen in Dauerkonstanten dürfen keine Genauigkeitsangaben enthalten.

BEISPIEL (Dauerkonstanten)

5 MIN 30 SEC	bedeutet 5 Minuten und 30 Sekunden
.05 SEC	bedeutet 50 Millisekunden
5 HRS 10 SEC	bedeutet 5 Stunden und 10 Sekunden

BEISPIEL (Deklaration einer Zeitdauervariablen)

```
DCL Verzoeigerung DUR ;  
...  
Verzoeigerung := 0.1 SEC ;
```

7.8 Referenzen (REF)

Syntax:

```
Typ-Referenz ::=  
  REF [ virtuelle-Dimensionsliste ] [ Zuweisungsschutz ]  
  { einfacher-Typ | Bezeichner$für-Typ | Typ-Struktur |  
    Typ-Dation | SEMA | BOLT | INTERRUPT | IRPT | SIGNAL |  
    Typ-Prozedur | TASK | CHAR( )  
  }
```

```
virtuelle-Dimensionsliste ::=  
  ( [ , ... ] )
```

```
Dereferenzierung ::=  
  CONT Name$Referenz-Variable
```

Semantik: Referenzvariablen haben als Werte die Namen von Variablen (sie zeigen auf Variablen). Analog zu den bisher eingeführten Variablen ist dabei der Wertebereich einer Referenzvariablen auf einen Typ von Variablen beschränkt, der bei der Deklaration der Referenzvariablen angegeben wird.

Auf den Wert der referenzierten Variablen bezieht man sich mittels des auf die Referenzvariable angewandten monadischen Operators **CONT** (von content); dieser Vorgang wird Dereferenzierung genannt.

Anstelle von $k := \text{CONT } rk1$ kann auch einfacher $k := rk1$ geschrieben werden; d. h. der Operator **CONT** darf auf der rechten Seite einer Zuweisung weggelassen werden (implizite Dereferenzierung).

Die Definition der Sprachform Typ-Referenz zeigt, welchen Typ eine Variable haben darf, auf die eine Referenzvariable zeigen soll. Insbesondere dürfen Referenzvariablen auch auf Felder und Strukturen zeigen. Die Anzahl n der Kommata in der virtuellen Dimensionsliste gibt an, dass die referenzierte Variable ein $(n + 1)$ -dimensionales Feld ist. Hingegen dürfen Referenzvariablen nicht auf Referenzvariablen zeigen; allerdings dürfen referenzierte Felder und Strukturen Referenzvariablen als Elemente und Komponenten besitzen.

Zeigt eine Referenzvariable R auf ein Feld F mit den Elementen $F(i, j, k, \dots)$ oder auf eine Struktur S mit den Komponenten $S.K_i$, so werden (ausgehend von R) die Elemente von F oder die Komponenten von S ohne Benutzung von **CONT** mit $R(i, j, k, \dots)$ oder $R.K_i$ angesprochen (implizite Dereferenzierung).

Eine Referenzvariable wird außerdem implizit dereferenziert,

- wenn sie aktueller Parameter eines Prozeduraufrufs und der zugehörige formale Parameter keine Referenzvariable ist,
- wenn sie als Operand eines dyadischen Operators benutzt wird, sofern dieser nicht genau für Werte von Referenzvariablen definiert ist wie **IS**.

Um z. B. das Ende einer Kette zu kennzeichnen, steht die Nullzeigerkonstante NIL zur Verfügung.

Zum Vergleich der Werte von Referenzvariablen können die dyadischen Operatoren IS oder ISNT benutzt werden; IS (ISNT) liefert das Ergebnis '1'B ('0'B), wenn die beiden angegebenen Referenzvariablen denselben (verschiedene) Wert(e) besitzen, andernfalls das Ergebnis '0'B ('1'B).

Variable Zeichenkettenzeiger (REF CHAR())

Variabel lange Zeichenketten können mit speziellen Zeigerobjekten, den sogenannten variablen Zeichenkettenzeigern, leicht verarbeitet werden.

Ein Zeiger vom Typ REF CHAR () enthält neben der Adresse einer Zeichenkette noch zwei Zähler, die die maximale Länge und die aktuell benutzte Länge der referenzierten Zeichenkettenvariable speichern. Bei der Zuweisung der Adresse einer Zeichenkettenvariablen an einen variablen Zeichenkettenzeiger werden beide Zähler mit der Länge der referenzierten Zeichenkettenvariablen initialisiert.

Damit unterscheidet sich der variable Zeiger zunächst nicht von einem Zeichenkettenzeiger mit einer konstanten Länge. Bei Zuweisungen an die Zeichenkettenvariable über diesen Zeiger wird jedoch die Länge der zugewiesenen Zeichenkette in dem Zähler für die aktuelle Länge gespeichert. Das bei Zeichenketten fester Länge übliche Auffüllen der Zeichenkette bis zur maximalen Länge mit Leerzeichen entfällt.

Bei lesenden Zugriffen auf Zeichenketten über einen variablen Zeichenkettenzeiger wird die aktuelle Länge benutzt.

Ein variabler Zeichenkettenzeiger ist als Ergebniswert einer Funktion nicht erlaubt. Der Typ CHAR () darf nur in Verbindung mit REF benutzt werden, d. h. der Parametertyp CHAR () IDENT ist nicht erlaubt.

Erläuterungen: Für die indirekte Adressierung stehen in PEARL sogenannte Referenzvariablen (Zeigervariablen) zur Verfügung.

```
DCL (k, l) FIXED,
  x FLOAT,
  (rk1, rk2) REF FIXED,    /* Fixed-Referenz-Variable */
  rx REF FLOAT ;          /* Float-Referenz-Variable */

rk1 := k ;                /* rk1 zeigt auf k */
rk1 := l ;                /* rk1 zeigt auf l */
rk2 := rk1 ;              /* rk2 zeigt auf l, Zeigerzuweisung */
rx := x ;                 /* rx zeigt auf x */
rx := k ;                 /* falsch, Typ ungleich */
rx := rk1 ;               /* falsch, Typ ungleich */
l := 2 ;
k := CONT rk1 ;           /* k erhält den Wert 2 */
rk2 := 3 ;                /* falsch, 3 ist keine Variable */
CONT rk2 := 3 ;           /* l erhält den Wert 3 */
CONT rk2 := k ;           /* l erhält den Wert 2 */
```

BEISPIEL (implizite Dereferenzierung)

```
DCL rk REF FIXED,
  k FIXED ;
...
rk := k ; k := " ,
k := rk + 1 ; /* äquivalent k := k + 1 ; */
```

BEISPIEL (Verwendung von IS)

```
DCL Nextauftrag REF Typ_Auftrag ;
...
IF Nextauftrag IS NIL THEN
FIN ;
```

BEISPIEL (Vereinbarungen variabler Zeichenkettenzeiger)

```
DCL str_ptr REF CHAR ( );           /* Zeigerdeklaration */
SPS print ENTRY ( REF CHAR ( ) ) GLOBAL; /*Zeiger als Parameter */
```

BEISPIEL (Initialisierung eines variablen Zeichenkettenzeigers):

```
DCL s1          CHAR(20);
DCL s2          CHAR(100);
DCL str_ptr     REF CHAR ( );
str_ptr := s1;   /* Adr=s1, Länge=20, Max=20 */
str_ptr := s2;   /* Adr=s2, Länge=100, Max=100 */
```

BEISPIEL (Zuweisungen an eine referenzierte Variable):

```
str_ptr := s1;           /* Adr=s1, Länge=20, Max=20 */
CONT str_ptr := 'PEARL 90'; /* Adr=s1, Länge=8, Max=20 */

str_ptr := s2;           /* Adr=s2, Länge=100, Max=100 */
CONT str_ptr := 'Werum GmbH'; /* Adr=s2, Länge=10, Max=100 */
```

BEISPIEL (lesender Zugriff über variablen Zeichenkettenzeiger):

```
DCL s1          CHAR(20);
DCL s2          CHAR(100);           /* platzsparende Variable */
DCL str_ptr     REF CHAR ( );       /* variabler Zeiger */

str_ptr := s2;           /* Adr=s2, Länge=100, Max=100 */
CONT str_ptr := 'D21337 ' CAT 'Lueneburg'; /* Adr=s2, Länge=16, Max=100 */
```

Nach dieser Zuweisung enthält die Variable s2 an den ersten 16 Stellen die Zeichen 'D21337 Lueneburg', die restlichen Zeichen wurden durch die Zuweisung nicht verändert. Der Längenindex von str_ptr wurde auf die aktuelle Länge 16 eingestellt.

```
s1 := CONT str_ptr;
```

Bei dieser Zuweisung erzeugt CONT str_ptr den Zeichenkettenwert 'D21337 Lueneburg' (mit dem Typ CHAR(16)). Nach den Zuweisungsregeln von PEARL wird die Zeichenkette mit vier Leerzeichen erweitert, bevor dieser Wert der Variablen s1 (mit dem Typ CHAR(20)) zugewiesen wird.

Besonders nützlich erweisen sich variable Zeichenkettenzeiger als formale Parameter von Prozeduren. Der Übersetzer übergibt an der Aufrufstelle neben der Adresse einer Zeichenkettenvariablen auch deren aktuelle Länge. So kann z. B. eine Fehleroutine unterschiedlich lange Fehlertexte ausgeben.

BEISPIEL (Zeichenkettenzeiger als Parameter)

```
SPC print_fehler ENTRY ( REF CHAR ( ) ) GLOBAL;  
DCL s1 CHAR(20) INIT ( '...' );  
DCL s2 CHAR(100) INIT ( '...' );  
  
...  
CALL print_fehler ( s1 );  
CALL print_fehler ( s2 );
```

Es dürfen auch Zeichenkettenkonstanten übergeben werden, wenn der formale Parameter mit dem Typ REF INV CHAR () definiert wurde.

BEISPIEL (Zeichenkettenkonstanten als Parameter)

```
SPC print_message ENTRY ( REF INV CHAR ( ) ) GLOBAL;  
  
...  
CALL print_message ( 'Text 1' );  
CALL print_message ( 'laengerer Text 2' );
```

Ein variabler Zeichenkettenzeiger kann in allen Ausdrücken analog zu konstanten Zeichenkettenzeigern benutzt werden. Die zur Typprüfung gehörende Längenprüfung erfolgt erst zur Laufzeit.

7.9 Felder

Syntax: Die allgemeine Form der Deklaration eines Feldes lautet:

```
Feld-Deklaration ::=  
    { DECLARE | DCL } Feld-Angabe [ , Feld-Angabe ] ... ;  
  
Feld-Angabe ::=  
    Bezeichner-Angabe Dimensions-Attribut [ Zuweisungsschutz ] Typ-Attribut  
    [ Global-Attribut ] [ Initialisierungsattribut ] [ PhysikalischeEinheit ]  
  
Dimensions-Attribut ::=  
    ( Grenzangabe$erste-Dimension [ ' Grenzangabe$weitere-Dimension ] ... )  
  
Grenzangabe ::=  
    [ Grenze$unten : ] Grenze$oben  
  
Grenze ::=  
    [ - ] ganze-Zahl-ohne-Genauigkeit | Bezeichner$benannte-Konstante |  
    Konstanter-Ausdruck
```

Die allgemeine Form der Spezifikation eines Feldes lautet:

```
Feld-Spezifikation ::=  
    { SPECIFY | SPC } Feld-Angabe-S [ , Feld-Angabe-S ] ... ;  
  
Feld-Angabe-S ::=  
    Bezeichner-Angabe virtuelles-Dimensionsattribut [ Zuweisungsschutz ]  
    Typ-Attribut { Global-Attribut | Identifikationsattribut } [ PhysikalischeEinheit ]  
  
virtuelles-Dimensionsattribut ::=  
    ( [ , ... ] )
```

Semantik: Skalare Variablen gleichen Typs können zu n-dimensionalen Feldern zusammengefasst werden (n = 1, 2, 3, 4).

Dem Feld wird bei seiner Deklaration ein Bezeichner zugeordnet; die einzelnen Feldelemente (skalare Variablen) werden unter diesem Bezeichner und der Angabe ihrer Position innerhalb des Feldes (dem Index) angesprochen.

So wird z. B. mittels

DCL Koeff (1:2, 0:3) FIXED ;

ein zweidimensionales Feld vereinbart, wobei die erste Dimension die untere Grenze 1 und die obere Grenze 2, d. h. die Länge 2 besitzt, während die zweite Dimension die untere Grenze 0 und die obere Grenze 3, d. h. die Länge 4 hat. Dies bedeutet, dass Koeff aus 8 skalaren Fixed-Variablen besteht:

Koeff(1,0)	Koeff(1,1)	Koeff(1,2)	Koeff(1,3)
Koeff(2,0)	Koeff(2,1)	Koeff(2,2)	Koeff(2,3)

Auch negative ganze Zahlen sind als Dimensionsgrenze zugelassen. Die obere Grenze einer Dimension muss jedoch immer größer oder gleich ihrer unteren Grenze sein.

Ist die untere Grenze nicht angegeben, so wird hierfür der Wert 1 angenommen. Unter einer benannten Konstante wird hier eine Variable vom Typ FIXED (15) verstanden, die mit Zuweisungsschutz (INV) und Initialwert (INIT) deklariert ist.

Felder können auch gemischt mit skalaren Variablen in einer Deklaration vereinbart werden.

Die Anzahl n der Kommata im virtuellen Dimensionsattribut gibt an, dass das spezifizierte Feld n + 1 Dimensionen besitzt.

Erläuterungen: Nach Möglichkeit werden beim Programmieren gleichartige Objekte unter einem Bezeichner zusammengefasst und die einzelnen Objekte indiziert angesprochen.

BEISPIEL Eine Steuerung steuert drei Geräte G(1), G(2) und G(3). Bei Ausgabe der Bitkette '0001'B an die Steuerung soll G(1) eingeschaltet werden, bei Ausgabe von '0010'B das Gerät G(2), bei Ausgabe von '0100'B das Gerät G(3). Hierzu bietet sich an, die drei Einschaltssignale unter einem Bezeichner, z. B. einschalten, zusammenzufassen und indiziert zu benutzen:

```
DCL    einschalten (1 : 3) BIT (4) ,  
        i FIXED ;  
    einschalten (1) := '0001'B ;  
    einschalten (2) := '0010'B ;  
    einschalten (3) := '0100'B ;  
    ...  
    /* Übernahme des Wertes von i (Index des einzuschaltenden Geräts) von einem anderen  
    Programmteil  
    Ausgabe von einschalten (i) an die Steuerung */
```

BEISPIEL (Felddeklarationen mit impliziter unterer Grenze)

```
DCL einschalten (3) BIT (4) ,  
    Koeff (2, 0:3) FIXED ,  
    Meldung (20) CHAR (12) ;
```

Das eindimensionale Feld Meldung enthält z. B. 20 Fehlertexte, damit im Fehlerfall i ein Programm die Meldung (i) an eine Konsole ausgeben kann (1 = 1, ..., 20).

BEISPIEL (Mischung von Felddeklarationen und Deklarationen skalarer Variabler)

DCL Anz_MeId **INV FIXED** (15) **INIT** (20) ;

...

DCL Meldung (Anz_Meld) **CHAR** (12) ,
(i, k, k) **FIXED** ,
(einschalten, ausschalten) (3) **BIT** (4) ;

7.10 Strukturen

Syntax: Die allgemeine Form der Strukturdeklaration lautet:

Struktur-Deklaration ::=

{ **DECLARE** | **DCL** } Struktur-Angabe [, Struktur-Angabe] ... ;

Struktur-Angabe ::=

Bezeichner-Angabe\$Hauptstruktur [Dimensionsattribut] [Zuweisungsschutz]
Typ-Struktur [Global-Attribut] [Initialisierungsattribut]

Typ-Struktur ::=

STRUCT [Strukturkomponente [, Strukturkomponente] ...]

Strukturkomponente ::=

Bezeichner-Angabe Typ-Attribut-in-Struktur-Vereinbarung [PhysikalischeEinheit]

Typ-Attribut-in-Struktur-Vereinbarung ::=

[Dimensionsattribut]
{ einfacher-Typ | Typ-Referenz | Bezeichner\$für-Typ | Typ-Struktur }

Die allgemeine Form der Strukturspezifikation lautet:

Struktur-Spezifikation ::=

{ **SPECIFY** | **SPC** } Struktur-Angabe-S [, Struktur-Angabe-S] ... ;

Struktur-Angabe-S ::=

Bezeichner-Angabe\$Hauptstruktur [virtuelles-Dimensionsattribut]
[Zuweisungsschutz] Typ-Struktur { Global-Attribut | Identifikationsattribut }
[PhysikalischeEinheit]

Eine Struktur wird in eckige Klammern eingeschlossen; diese sind hier fett gedruckt, um sie von den metasprachlichen Zeichen [und] zu unterscheiden.

Semantik: Anders als Feldelemente werden die Komponenten einer Struktur nicht über den gemeinsamen Bezeichner und einen Index, sondern über den gemeinsamen Bezeichner und ihren in der Vereinbarung angegebenen Bezeichner angesprochen, wobei die beiden Bezeichner durch einen Punkt getrennt werden.

Komponenten von Strukturen dürfen Felder sein; die Feldelemente werden dann wie gewohnt mit ihrem indizierten Bezeichner angesprochen, dem der Strukturbezeichner vorangestellt ist.

Eine Strukturkomponente kann jedoch auch selbst wieder eine Struktur sein, wodurch nicht nur lineare, sondern auch hierarchische Datenstrukturen nachgebildet werden können. Unterstrukturen können über ihren Bezeichner und die vorangestellten Bezeichner der jeweils übergeordneten Strukturen angesprochen werden.

Der Typ einer Struktur wird durch die Anordnung ihrer Komponenten und deren Typ bestimmt. Strukturen gleichen Typs können zu Feldern zusammengefasst werden. Allgemein wird deshalb eine beliebige Komponente einer Struktur unter ihrem (ggf. indizierten) Bezeichner und den vorangestellten (ggf. indizierten) Bezeichnern aller übergeordneten Strukturen angesprochen, wobei diese Namen jeweils durch einen Punkt getrennt werden.

Erläuterungen: Bei vielen Automatisierungsaufgaben, insbesondere bei solchen mit dispositivem Charakter, müssen Datenstrukturen beschrieben werden, deren Komponenten *unterschiedliche* Typen besitzen.

BEISPIEL Die Beiträge für die Fernsehnachrichtensendungen eines Tages sind auf Magnetaufzeichnungsgeräten (MAZ) gespeichert; die Beitragsfolge einer bestimmten Nachrichtensendung dieses Tages wird von Redakteuren mit Hilfe eines Rechners zusammengestellt, der die entsprechenden MAZ steuert. Dazu ist für jeden Beitrag ein Datensatz nötig, der z. B. folgende Struktur besitzt:

- a) Identitätskennzeichen des Beitrages;
- b) Archivnummer;
- c) Hinweise, ob der Beitrag bereits in der ersten, zweiten oder dritten Nachrichtensendung dieses Tages gesendet wurde;
- d) Anfangsposition des Beitrages auf dem Band;
- e) Endeposition des Beitrages auf dem Band;
- f) Kennzeichen, ob Originalton vorhanden ist;
- g) Länge des ggf. zu speichernden Texts;
- h) ggf. zu speichernder Text.

Solche Datenstrukturen lassen sich problemorientiert als Struktur beschreiben; die obige Struktur Beitrag wird z. B. wie folgt vereinbart:

DCL Beitrag STRUCT
[(Identnr, Archiv) **FIXED**,
schon_gesendet (3) **BIT** (1),
(Anfang, Ende) **FIXED**,
Originalton **BIT** (1),
Textlaenge **FIXED**,
Text **CHAR** (200)] ;

BEISPIEL (Zuweisung an eine Strukturkomponente): Durch die Anweisung

Beitrag.Anfang := 1027;

erhält die Komponente Anfang der Struktur Beitrag den Wert 1027 zugewiesen.

BEISPIEL (Zugriff auf das Element eines Feldes, das Komponente einer Struktur ist):

IF Beitrag.schon_gesendet (i) **THEN ... FIN** ;

BEISPIEL (Strukturen als Komponenten einer Struktur): Eine Personaldatei enthalte die Beschreibungen von Mitarbeitern; diese Beschreibungen haben jeweils folgende Struktur:

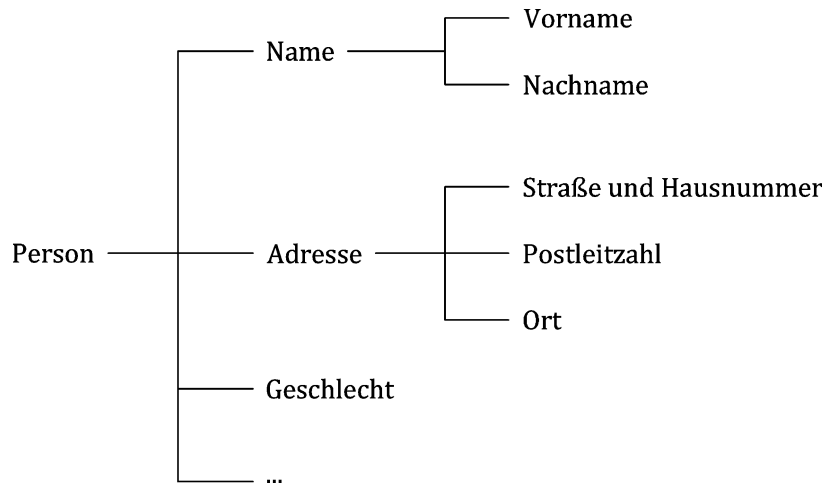


Bild 4 — Strukturen als Komponenten einer Struktur: Person

Die Komponenten Name und Adresse sind selbst Strukturen, nämlich Unterstrukturen der Hauptstruktur Person; diese kann so vereinbart werden:

```
DCL Person STRUCT
  [ Name STRUCT
    [ Vorname CHAR (10),
      Nachname CHAR (15)
    ],
    Adresse STRUCT
    [ Str CHAR (15);
      PLZ FIXED,
      Ort CHAR (15)
    ],
    Geschlecht CHAR (1),
    ...
  ];
```

BEISPIEL (Feld von Strukturen)

```
DCL Beitrag(20) STRUCT ... ;
```

BEISPIEL (Zugriffe auf Komponenten komplexer Strukturen)

```
Person.Name.Vorname := 'OTTO' ;
IF Beitrag(i).schon_gesendet (j) THEN ... FIN ;
```

7.11 Typvereinbarung (TYPE)

Syntax: Ein neuer Datentyp kann wie folgt vereinbart werden:

```
Typ-Vereinbarung ::=
  TYPE Bezeichner$für-Typ { einfacher-Typ | Typ-Struktur }
```

Semantik: Ein neuer Datentyps muss vor seiner Benutzung vereinbart werden. Im Übrigen können auf Modulebene vereinbarte Datentypen in allen Tasks und Prozeduren des Moduls benutzt werden; die Benutzbarkeit lokal vereinbarter Datentypen richtet sich nach den Regeln der Blockstruktur.

Im Gegensatz zu globalen Objekten können neu vereinbarte Datentypen nur in dem Modul benutzt werden, in dem sie vereinbart sind. Typvereinbarungen für globale Objekte müssen also in allen diese Objekte benutzenden Modulen wiederholt werden.

Erläuterungen: Eine bestimmte Struktur kann z. B. als Parameter oder als Unterstruktur in anderen Strukturen oder als Typ der Übertragungsdaten einer Datenstation auftreten. Dabei muss jedes Mal der Typ dieser Struktur angegeben werden, was bei komplexen Strukturen umfangreichen Schreibaufwand erfordern kann. Deshalb und zur Erhöhung der Lesbarkeit des Programms kann der Typ einer Struktur unter einem frei wählbaren Bezeichner als neuer Datentyp vereinbart und unter diesem Bezeichner z. B. dazu benutzt werden, Variablen dieses Datentyps zu vereinbaren.

BEISPIEL (Vereinbarung von Datentypen)

1) **PROBLEM ;**

```
TYPE   Nachricht STRUCT
[       (Identnr, Archiv) FIXED,
        schon_gesendet (3) BIT (1),
        (Anfang, Ende) FIXED,
        Originalton BIT (1),
        Textlaenge FIXED,
        Text CHAR (200) ] ;
```

```
DCL Inhalt_MAZ DATION INOUT Nachricht ... ;
```

```
Koord: TASK ;
```

```
    DCL Beitrag Nachricht ;
```

```
    ...
```

```
    READ Beitrag FROM Inhalt_MAZ ;
```

```
    ...
```

```
    END ; ! Koord
```

- 2) Eine Task steuert bestimmte Geräte; die Steuerungsaufträge erhält sie von einer anderen Task. Weil zeitweise mehr als ein Auftrag vorliegen kann und die Aufträge unterschiedliche Dringlichkeit haben können, werden sie in Form einer Warteschlange gepuffert. Die Struktur der Aufträge sei vom (neu vereinbarten) Typ **Auftragstyp**; dieser enthalte auch das Dringlichkeitskriterium, nach dem neue Aufträge eingeordnet werden. Die Referenzvariablen **Next_Auftrag** bzw. **Frei** zeigen auf den nächsten zu bearbeitenden Auftrag bzw. den nächsten freien Platz in der Warteschlange Auftragskette; ausgehend von **Frei** seien die freien Plätze ebenfalls verkettet. Die Warteschlange sei höchstens 10 Elemente lang.

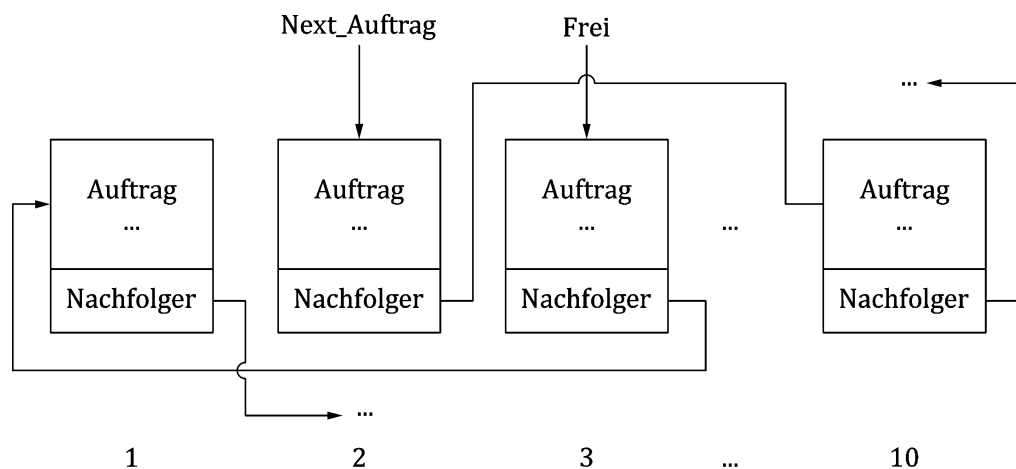


Bild 5 — Warteschlange

Die Auftragskette kann folgendermaßen vereinbart werden:

```
TYPE Kettenelement STRUCT
    [   Auftrag Auftragstyp,
        ...,
    ];
DCL Auftragskette ( 10 ) Kettenelement,
    (Next_Auftrag, Frei) REF Kettenelement;
```

Nach erfolgter Initialisierung und dem Einordnen verschiedener Aufträge kann ein fertig bearbeiteter Auftrag wie folgt aus der Warteschlange entfernt werden (Hilf sei dabei eine Variable vom Typ REF Kettenelement):

```
Hilf := Next_Auftrag ;
Next_Auftrag := Next_Auftrag.Nachf ;
Hilf.Nachf := Frei ;
Frei := Hilf ;
```

Diese Anweisungen sind auch weitere Beispiele für implizites Dereferenzieren. Die Auftragskette kann so initialisiert werden:

```
FOR i TO 9
    REPEAT
        Auftragskette (i).Nachf := Auftragskette (i + 1) ;
    END ;
Auftragskette (10).Nachf := NIL ;
```

BEISPIEL (Verwendung eines vereinbarten Datentyps in verschiedenen Modulen)

<pre>MODULE (a); PROBLEM ; TYPE T STRUCT [...] ; ... DCL x T GLOBAL ; ... MODEND ;</pre>		<pre>MODULE (b); PROBLEM ; TYPE T STRUCT [...] ; ... SPC x T GLOBAL ; ... MODEND ;</pre>
--	--	--

7.12 Initialisierungsattribut (INITIAL)

Syntax:

```
Initialisierungsattribut ::=
    { INITIAL | INIT } ( Init-Element [ , Init-Element ] ... )

Init-Element ::=
    Konstante
    | Bezeichner$benannte-Konstante
    | konstanter-Ausdruck
    ...
```

Konstante ::=

- ganze-Zahl
- | Gleitpunktzahl
- | Bitkettenkonstante
- | Zeichenkettenkonstante
- | Uhrzeitenkonstante
- | Dauerkonstante
- | **NIL**

Semantik: Das Initialisierungsattribut erlaubt, Variablen bei ihrer Deklaration Anfangswerte zuzuweisen. Für Variablen mit Zuweisungsschutz ist dies die einzige Möglichkeit der Wertzuweisung. Das Initialisierungsattribut wird bei der jeweiligen Deklaration als letztes Attribut angegeben.

Der Typ des angegebenen Init-Elements muss mit dem Typ der deklarierten Variablen gemäß den Regeln für die Zuweisung verträglich sein.

Felder und Strukturen können ebenfalls an der Deklarationsstelle initialisiert werden. Für alle Strukturkomponenten muss jeweils ein Initialwert angegeben werden. Bei Feldern darf die Initialisierungsliste kürzer als die Anzahl der Feldelemente sein. In diesem Fall wird der letzte Initialwert für die restlichen Elemente wiederholt benutzt.

Die Initialisierungsliste kann außer Konstanten auch Bezeichner benannter Konstanten sowie vom Übersetzer berechenbare Ausdrücke enthalten.

BEISPIEL (Zuordnung der Elemente der Initialisierungsliste)

```
DCL Anzahl_Geraete FIXED INIT(12),  
      (UGR, OGR) FIXED INIT(2,15) ;  
...  
FOR i FROM 1 TO Anzahl_Geraete
```

Hier haben die Variablen Anzahl_Geraete, UGR und OGR die Werte 12, 2 und 15, d. h. die Elemente einer angegebenen Konstantenliste werden in der Reihenfolge der Niederschrift der angegebenen Bezeichnerliste zugeordnet.

BEISPIEL (Initialisierung von Strukturen und Feldern)

```
DCL Adresse      STRUCT [ Postleitzahl FIXED,  
                           Ort           CHAR(20),  
                           Strasse       CHAR(20),  
                           HausNr        FIXED ]  
      INIT ( 21337, 'Lueneburg', 'Erbstorfer Landstr.', 14 ) ;  
DCL Farben      ( 3 ) CHAR (7) INIT ( 'schwarz', 'weiss', 'rot' ) ;  
DCL alles_Sechsen ( 6 ) FIXED INIT ( 6 ) ;
```

7.13 Zuweisungsschutz (INV)

Syntax:

Zuweisungsschutz ::=

INV

Semantik: Skalare Variablen, Felder und Strukturen können mit einem Zuweisungsschutz mittels des Attributs INV (von invariant), deklariert werden, um – abgesehen von der Initialisierung – Zuweisungen an diese Objekte zu verbieten. Das Attribut INV wird unmittelbar vor dem Typ-Attribut angegeben.

Ein einmal für ein Objekt in seiner Deklaration vereinbarter Zuweisungsschutz kann nicht aufgehoben werden; er muss deshalb in Spezifikationen dieses Objekts berücksichtigt werden, aber auch bei Übertragungen als aktueller Prozedurparameter oder bei der Benutzung als Wert von Referenzvariablen. Andererseits darf auf eine dieser Arten ein bei der Deklaration noch nicht vereinbarter Zuweisungsschutz neu entstehen.

Benannte Konstanten (INV-Konstanten)

Mit dem Attribut INV deklarierte Objekte vom Typ FIXED, BIT, CHAR(1), CLOCK und DURATION behandelt der Übersetzer als (benannte) Konstanten. Die Namen dieser Objekte dürfen nach ihrer Deklaration (mit Initialisierung) bei der Deklaration und Initialisierung weiterer Objekte benutzt werden. Dabei dürfen sie sowohl als Dimensionsgrenzenangabe bei Felddeklarationen als auch in Initialisierungslisten eingesetzt werden.

Da der Übersetzer für die benannten Konstanten keine Laufzeitobjekte erzeugt, dürfen sie nicht einer Referenzvariablen zugewiesen werden. Ebenso können sie nicht als aktuelle Parameter einer Prozedur benutzt werden, wenn die formalen Parameter mit dem IDENTICAL-Attribut oder als Referenzvariablen spezifiziert wurden.

BEISPIEL (Deklaration einer skalaren Variablen mit Zuweisungsschutz)

```
DCL      Pi INV FLOAT INIT (3.141) ;  
Pi := 3 ; /* Ergibt Fehlermeldung des Übersetzers */
```

BEISPIEL (Inkonsistenzen bei der Verwendung von INV)

```
P: PROC ( A (,) INV FIXED IDENT , x FLOAT IDENT ) ;  
    /* Prozedurrumpf */  
END ; ! P  
  
DCL Tab (10,20) FIXED,  
    Pi INV FLOAT INITIAL (3.141),  
    R1 REF FLOAT,  
    R2 REF INV FIXED ;  
  
SPC pi FLOAT IDENT (Pi) ;      /* Falsch */  
CALL P (Tab, Pi) ;            /* Falsch */  
R1 := Pi ; 1                  /* Falsch */  
R2 := Tab(5, 7) ;
```

Die Spezifikation von pi ist falsch, weil Zuweisungen an pi erlaubt werden und somit der Zuweisungsschutz von Pi unterlaufen wird.

Der Aufruf von P ist falsch, weil der zu Pi gehörige formale Parameter x ohne Zuweisungsschutz INV vereinbart ist und somit im Rumpf von P über eine Zuweisung an x eine (verbotene) Wertänderung von Pi erfolgen könnte.

Dagegen ist im selben Aufruf die Parameterübertragung von Tab an A korrekt, da lediglich ein bisher noch nicht vereinbarter Zuweisungsschutz für Tab entstanden ist.

Die Zuweisung Ri := Pi ist unzulässig, da andernfalls der Zuweisungsschutz für Pi durch (erlaubte) Zuweisungen an CONT R1 aufgehoben werden könnte. Die Zuweisung R2 := Tab(5,7) ist dagegen zulässig; eine Wertänderung des Tabellenelementes über CONT R2 ist nicht möglich.

BEISPIEL (Benannte Konstanten): Ein Überwachungsprogramm soll für verschiedene Projekte mit unterschiedlichen Anzahlen unterlagerter Steuerungen angepasst werden. Die Anzahl der Nachrichtenpuffer und zugehörigen Synchronisierungsvariablen lässt sich durch Ändern der Konstante AnzahlSteuerungen leicht anpassen:

```
DCL    AnzahlSteuerungen INV FIXED INIT (11) ;  
DCL    NachrichtenPuffer ( AnzahlSteuerungen ) CHAR(100);  
DCL    Pufferzugriff ( AnzahlSteuerungen ) BOLT;
```

Für ein weiteres Projekt mit einer anderen Anzahl von Steuerungen muss nur die benannte Konstante AnzahlSteuerungen den aktuellen Gegebenheiten angepasst werden. Die davon abhängigen Speicherbereiche werden durch den Übersetzer angepasst.

7.14 Physikalische Einheiten (PHYSUNIT)

Syntax:

```
PhysikalischeEinheit ::=  
    { PHYSUNIT } ( PhysUnit-Element [ , PhysUnit-Element ] ... )  
  
PhysUnit-Element ::=  
    Zeichenkettenkonstante
```

Semantik: Mittels dieser Klausel lassen sich Variablen bei ihrer Spezifikation und Deklaration physikalische Einheiten zuweisen. Der Übersetzer kann dann anhand der gemachten Angaben bei Zuweisungen und der Berechnung von Ausdrücken die Verträglichkeit der Einheiten prüfen.

7.15 Konstante Ausdrücke

Syntax:

```
konstanter-Ausdruck ::=  
    { + | - } Gleitpunktzahl  
    | { + | - } Dauerkonstante  
    | konstanter-FIXED-Ausdruck  
  
konstanter-FIXED-Ausdruck ::=  
    Term [ { + | - } Term ] ...  
  
Term ::=  
    Faktor [ { * | // | REM } Faktor ] ...  
  
Faktor ::=  
    [ + | - ]  
    {  
        ganze-Zahl  
        | ( konstanter-FIXED-Ausdruck )  
        | TOFIXED { Zeichenkettenkonstante$der-Länge-1 | Bitkettenkonstante }  
        | Bezeichner$benannte-Konstante  
    }  
    [ FIT konstanter-FIXED-Ausdruck ]
```

Semantik: Konstante Ausdrücke vom Typ FIXED können z. B. an folgenden Stellen verwendet werden:

- a) bei der Angabe von Dimensionsgrenzen in Felddeklarationen;
- b) bei Genauigkeitsangaben (für FIXED- und FLOAT-Objekte);
- c) bei Längenangaben (für CHAR- und BIT-Objekte);

- d) in Initialisierungslisten (auch bei PRESET);
- e) bei Prioritätsangaben in Task-Deklarationen/Anweisungen;
- f) in den Konstantenlisten der CASE-Anweisung und
- g) auf der rechten Seite einer Zuweisung.

Erläuterungen: An allen Stellen, an denen der Übersetzer Konstanten erwartet, dürfen auch konstante Ausdrücke stehen. Diese Ausdrücke dürfen als Operanden nur Konstanten (auch benannte Konstanten) enthalten. Sie werden durch den Übersetzer ausgewertet und das Ergebnis an den entsprechenden Stellen eingesetzt.

Oft sind mehrere Objekte eines Programms (Variablen, Konstanten) voneinander abhängig. Lässt sich diese Abhängigkeit durch eine einfache Formel beschreiben, kann ein Programm evtl. durch Anpassen einiger weniger Konstanten konfiguriert werden.

BEISPIEL (Anwendung konstanter Ausdrücke): Es soll eine Warnung ausgegeben werden, wenn ein Puffersystem bis auf die letzten 10 % gefüllt ist.

```
...
DCL MaxPuffer          INV      FIXED INIT ( 1000 );
DCL WarnGrenze         INV      FIXED INIT ( MaxPuffer - MaxPuffer // 10 );
DCL Puffer ( MaxPuffer)  FIXED ;
DCL aktuellerPufferIndex  FIXED INIT ( 1 );
...
aktuellerPufferIndex := aktuellerPufferIndex + 1;
IF aktuellerPufferIndex > WarnGrenze
THEN /* Warnung ausgeben */
FIN ;
```

8 Ausdrücke und Zuweisungen

8.1 Ausdrücke

Syntax:

```
Ausdruck ::=
    [ monadischer-Operator | Operand [ dyadischer-Operator Ausdruck ] ...

monadischer-Operator ::=
    + | - | NOT | ABS | SIGN | LWB | UPB | SIZEOF | CONT |
    TOFIXED | TOFLOAT | TOBIT | TOCHAR | ENTIER | ROUND |
    Bezeichner$Benutzer-definierter-monadischer-Operator

dyadischer-Operator ::=
    + | - | * | / | // | REM | ** | FIT |
    < | LT | > | GT | <= | LE | >= | GE | == | EQ | /= | NE |
    >< | CAT | <> | CSHIFT | SHIFT | LWB | UPB | IS | ISNT |
    Bezeichner$Benutzer-definierter-dyadischer-Operator

Operand ::=
    Konstante | Name | Funktionsaufruf | bedingter-Ausdruck | Dereferenzierung |
    Kettenausschnitt | ( Ausdruck ) | ( Zuweisung )

Name ::=
    Bezeichner [ ( Index [ , Index ] ... ) ] [ . Name ] ...
```

Index ::=
 Ausdruck\$mit-ganzer-Zahl-als-Wert

Funktionsaufruf ::=
 Bezeichner\$Funktionsprozedur [Liste-aktueller-Parameter];

Liste-aktueller-Parameter ::=
 (Ausdruck [, Ausdruck] ...)

bedingter-Ausdruck ::=
 IF Ausdruck\$mit-einem-Wert-vom-Typ-BIT(1)
 THEN Ausdruck **ELSE** Ausdruck **FIN**

Dereferenzierung ::=
 CONT { Name\$Referenz | Funktionsaufruf }

Kettenausschnitt ::=
 Bitkettenausschnitt | Zeichenkettenausschnitt

Bitkettenausschnitt ::=
 Name\$Bit-Kette **.BIT**
 ({ Ausdruck\$konstanter-FIXED-Ausdruck : Ausdruck\$konstanter-FIXED-Ausdruck |
 Ausdruck\$FIXED-Ausdruck [: Ausdruck\$FIXED-Ausdruck + ganze-Zahl] })

Zeichenkettenausschnitt ::=
 Name\$Zeichen-Kette { **CHAR** | **CHARACTER** }
 ({ Ausdruck\$konstanter-FIXED-Ausdruck : Ausdruck\$konstanter-FIXED-Ausdruck |
 Ausdruck\$FIXED-Ausdruck [: Ausdruck\$FIXED-Ausdruck + ganze-Zahl] })

Semantik:

- a) Namen und Konstanten: In seiner einfachsten Form ist ein Ausdruck eine Konstante oder ein Name. Unter der syntaktischen Einheit Name werden Bezeichner, indizierte Bezeichner und Namen von Strukturkomponenten zusammengefasst.
- b) bedingter-Ausdruck: Weiterhin kann als Ausdruck ein bedingter-Ausdruck auftreten. Der bedingte-Ausdruck dient zur alternativen Berechnung eines Wertes in Abhängigkeit von einer Bedingung. Die Bedingung wird durch den Ausdruck hinter dem Symbol IF formuliert.

Zur Auswertung von bedingter-Ausdruck wird zuerst der Ausdruck hinter dem Symbol IF berechnet. Liefert die Berechnung den Wert '1'B (den logischen Wert wahr), wird der bedingte-Ausdruck durch den Ausdruck hinter dem Symbol THEN ersetzt. Liefert die Berechnung den Wert '0'B (den logischen Wert falsch), wird der bedingte-Ausdruck durch den Ausdruck hinter dem Symbol ELSE ersetzt.

- c) Kettenausschnitte: Kettenausschnitte bezeichnen Teilketten von Bit- oder Zeichenketten.

Die Symbole BIT und CHAR bzw. CHARACTER geben an, um welchen Datentyp der Original- und der Ergebniskette es sich handelt. BIT bezeichnet Bitketten. CHAR bzw. CHARACTER bezeichnen Zeichenketten.

Mit der in runden Klammern eingeschlossenen Ausdrucksform wird der Ausschnitt der Originalkette beschrieben, der die Ergebniskette darstellt. Folgende Konstruktionen sind als Kettenausschnitte zulässig:

- 1) Name. { BIT | CHAR } (Ausdruck): Hier wird als Ergebnis des Gesamtausdrucks aus der durch Name bezeichneten Bit- oder Zeichenkette ein Bit oder ein Zeichen geliefert. Der in den runden Klammern eingeschlossene Ausdruck bezeichnet die Position des Bits bzw. des Zeichens in der Ausgangskette, die von links nach rechts, beginnend mit 1, nummeriert ist. Das Ergebnis ist ein Wert vom Typ BIT(1) bzw. CHAR(1).
- 2) Name. { BIT | CHAR } (konstanter-FIXED-Ausdruck : konstanter-FIXED-Ausdruck)

- 3) Name. CHAR (Ausdruck : Ausdruck): Hier wird als Ergebnis des Gesamtausdrucks aus der durch Name bezeichneten Bit- oder Zeichenkette eine Bit- oder eine Zeichenteilkette geliefert. Die in runden Klammern eingeschlossenen Ausdrücke bezeichnen die Anfangs- und die Endposition der Bit- bzw. der Zeichenteilkette in der Ausgangskette.

Der Ausdruck vor dem Doppelpunkt bezeichnet die Anfangsposition und der nach dem Doppelpunkt bezeichnet die Endposition. Beide Ausdrücke müssen ganzzahlige Werte als Ergebnis liefern. Der Ausdruck für die Anfangsposition muss einen Wert liefern, der kleiner oder gleich dem durch den Ausdruck für die Endposition gelieferten Wert ist. Beide Werte müssen innerhalb der deklarierten Kettenlänge liegen.

Das Ergebnis ist eine Bit- bzw. Zeichenkette vom Typ BIT (lg) bzw. CHAR (lg). Der Wert für lg ergibt sich aus: Wert der Endposition – Wert der Anfangsposition + 1.

Im Falle, dass die Positionen durch konstante Ausdrücke angegeben werden (Ausdrücke, in denen nur dem Übersetzer ihrem Wert nach bekannte Konstanten als Operanden vorkommen), wird zur Übersetzungszeit die Länge des Ergebniskettenausschnitts berechnet. Im anderen Fall handelt es sich um variable Ketten, deren Länge erst zur Laufzeit bestimmt wird. Diese Form der variablen Ketten ist nur für Ketten des Typs CHAR erlaubt.

- 4) Name. { BIT | CHAR } (Ausdruck: Ausdruck + ganze-Zahl): Die Ausdrücke müssen beide gleich sein und ganzzahlige Werte als Ergebnisse liefern. Das Ergebnis ist eine Bit- bzw. Zeichenkette vom Typ BIT (lg) bzw. CHAR (lg). Der Wert für lg ergibt sich aus: Wert der ganzen Zahl + 1.

Die bezeichneten Teilketten müssen vollständig in der deklarierten Originalkette liegen.

Zuweisungen an Kettenausschnitte sind möglich; außerdem dürfen sie als Operanden in Ausdrücken auftreten.

Teile eines Ausdrucks können geklammert werden, um die Reihenfolge der Ausdrucksberechnung zu beeinflussen.

Monadische Operatoren

Tabelle 4 — Bedeutung der monadischen Operatoren für numerische, zeitliche und Bitgrößen

Ausdruck	Operanden-Typ	Ergebnis-Typ	Bedeutung des Operators
+a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	Identität
–a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	Umkehrung des Vorzeichens von a
NOT a	BIT (lg)	BIT (lg)	Invertieren aller Bitstellen von a
ABS a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	Absolutbetrag von a
SIGN a	FIXED (g) FLOAT (g) DURATION	FIXED (1)	Ermitteln des Vorzeichens von a 1 für a > 0 0 für a = 0 –1 für a < 0