PEARL 90

LANGUAGE REPORT

Version 2.2

September 1998

Ed: GI-Working Group 4.4.2 "Real-Time Programming, PEARL"

Contents

1 Introduction 11

2 Fundamental Features of PEARL 13

2.1 Multitasking Characteristics 13

2.2 Possibilities of Input and Output 14

2.3 Program Structure 15

2.4 Data Types 15

2.5 Control Structures 16

3 Rules for the Construction of PEARL Language Forms 19

3.1 Character Set 19

3.2 Basic Elements 20

3.2.1 Identifiers 21

3.2.2 Constants 21

3.2.3 Comments 21

3.3 Construction of Language Forms 21

4 Program Structure 25

4.1 Modules 25

4.2 Declarations and Specifications 26

4.2.1 Declaration (DCL) 26

4.2.2 Specification (SPC) and Identification (SPC IDENT) 27

4.3 Block Structure, Validity of Objects 28

4.4 References between Modules . . . 30
4.5 Execution of a Program . . . 31

5 Variables and Constants 33

5.1 Declaration and Specification of Variables and Constants . . . 33
5.2 Integers (FIXED) . . . 34
5.3 Floating Point Numbers (FLOAT) . . . 35
5.4 Bit Strings (BIT) . . . 36
5.5 Character Strings (CHARACTER) . . . 37
5.6 The Length Definition . . . 38
5.7 Times (CLOCK) . . . 38
5.8 Durations (DURATION) . . . 39
5.9 References (REF) . . . 39
5.9.1 Variable Character String Pointers (REF CHAR()) . . . 41
5.10 Arrays . . . 42
5.11 Structures . . . 44
5.12 Type Definition (TYPE) . . . 47
5.13 The Initialisation Attribute (INITIAL) . . . 49
5.14 Assignment Protection (INV) . . . 50
5.14.1 Named Constants (INV Constants) . . . 51
5.14.2 Constant Expressions (of Type FIXED) . . . 51

6 Expressions and Assignments 53

6.1 Expressions . . . 53
6.1.1 Monadic Operators . . . 57
6.1.2 Dyadic Operators . . . 59
6.1.3 Evaluation of Expressions . . . 64
6.2 Operator Definition (OPERATOR) . . . 65
6.3 Assignments . . . 66
6.3.1 Assignments for Scalar Variables . . . 66

6.3.2 Assignments for Structures 67
6.4 Overloading of Data Structures 68
6.4.1 The "BY TYPE" Operator 68
6.4.2 The "VOID" Data Type 69

7 Statements for the Control of Sequential Execution 71
7.1 Conditional Statement (IF) 71
7.2 Statement Selection (CASE) and Empty Statement 72
7.3 Repetition (FOR – REPEAT) 74
7.4 GoTo Statement (GOTO) 75
7.5 Exit Statement (EXIT) 76

8 Procedures 79
8.1 Declaration and Specification of Procedures (PROC) 80
8.2 Calls of Procedures (CALL) 82
8.3 References to Procedures (REF PROC) 84

9 Parallel Activities 87
9.1 Declaration and Specification of Tasks (TASK) 87
9.1.1 References to Tasks (REF TASK) 88
9.1.2 Determining Task Addresses 89
9.1.3 Determining Task Priorities 90
9.2 Statements for Controlling Tasks 90
9.2.1 Start Condition 90
9.2.2 Starting a Task (ACTIVATE) 92
9.2.3 Terminating a Task (TERMINATE) 93
9.2.4 Suspending a Task (SUSPEND) 93
9.2.5 Continuing a Task (CONTINUE) 93
9.2.6 Delaying a Task (RESUME) 95
9.2.7 De-scheduling a Task (PREVENT) 95

9.3 Synchronising Tasks 96
9.3.1 Semaphore Variables (SEMA) and Statements (REQUEST, RELEASE, TRY) 97
9.3.2 Bolt Variables (BOLT) and Statements (ENTER, LEAVE, RESERVE, FREE) 100
9.4 Interrupts and Interrupt Statements 102
9.4.1 Declarations of Interrupts and Software Interrupts 102
9.4.2 Interrupt Statements (TRIGGER, ENABLE, DISABLE) 103

10 Input and Output **105**

10.1 System Part 105
10.2 Specification and Declaration of Data Stations (DATION) 106
10.2.1 System Data Stations 106
10.2.2 User Defined Data Stations 107
10.3 Opening and Closing of Data Stations (OPEN, CLOSE) 111
10.4 The Read and Write Statements (READ, WRITE) 113
10.5 The Get and Put Statements (GET, PUT) 117
10.5.1 The Fixed Format (F) 119
10.5.2 The Float Format (E) 120
10.5.3 The Character String Formats (A) and (S) 121
10.5.4 The Bit Format (B) 123
10.5.5 The Time Format (T) 124
10.5.6 The Duration Format (D) 124
10.5.7 The List Format (LIST) 125
10.5.8 The R Format (R) 125
10.6 The Convert Statement (CONVERT) 126
10.7 The Take and Send Statements 126
10.8 Error Handling in I/O Statements (RST) 127
10.9 Interface for Additional Drivers 128

11 Signals **129**

12 Appendix **133**

12.1 Data Types and their usability 133

12.2 Predefined Functions 134

12.2.1 The Function NOW 134

12.2.2 The Function DATE 134

12.3 Syntax 135

12.3.1 Program 135

12.3.2 System Part 135

12.3.3 Basic Elements 136

12.3.4 Problem Part 138

12.3.5 Specification and Identification 142

12.3.6 Expressions 143

12.3.7 Statements 144

12.4 List of Keywords with Shortforms 150

12.5 Other Word Symbols in PEARL 152

List of Tables

4.1 Permissibility of declarations 28

6.1 Monadic operators for numerical, temporal and bit values 57

6.2 Monadic operators for explicit type conversions 58

6.3 Monadic arithmetical operators 58

6.4 Other monadic operators 59

6.5 Dyadic operators for numerical and temporal values 60

6.6 Dyadic comparison operators 62

6.7 Dyadic Boolean and shift operators 63

6.8 Dyadic character string operators 63

6.9 Other dyadic operators 63

6.10 Ranks of the operators defined in PEARL 64

Chapter 1

Introduction

PEARL stands for **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage; it is a higher programming language that allows a comfortable, safe and to a large extent computer-independent programming of multitasking and real-time tasks. PEARL was standardised by DIN in different extension steps:

- DIN 66253, part 1, Basic PEARL, 1981 (subset of Full PEARL)
- DIN 66253, part 2, Full PEARL, 1982
- DIN 66253, part 3, PEARL for Distributed Systems, 1989

Based on the experiences from hundreds of PEARL projects, a gremium of PEARL users and implementators worked out a definition of PEARL 90 in 1989 and 1990 (cf. Stieger, K.: "PEARL 90 — Die Weiterentwicklung von PEARL", in: *Informatik-Fachberichte 231*, PEARL '89 Workshop über Realzeitsysteme, Springer 1989).

PEARL 90 corresponds to Full PEARL, though some language elements not needed in practice are omitted. On the other hand, PEARL 90 contains some progressive extensions.

DIN 66253, part 1 and part 2, are now substituted for

- DIN 66253-2, PEARL 90, 1998.

The language report presented here describes the language extent of PEARL 90, first outlining the fundamental characteristics of PEARL 90. In the following, the language elements will be defined exactly.

The appendix contains a list of all data types and their possible applications, a description of the available predefined functions, a complete description of the syntax, because for didactical reasons not all syntactic possibilities are described in the respective sections, as well as a list of all keywords.

Chapter 2

Fundamental Features of PEARL

2.1 Multitasking Characteristics

A data processing program for on-line control or on-line analysis of a technical process must be able to react as soon as possible to spontaneous requests of the process or temporal events. Thus, in most cases it is not sufficient to arrange and to carry out the individual programming steps sequentially, i.e. in temporally unchangeable order. The more or less complex automation problem has rather to be split up into problem related parts with varying urgency, and the program structure has to be adapted to this problem structure. Upon this, independent program components for problem parts are created that can be worked on sequentially within other components (e.g. procedures); but also independent program components for problem parts arise that have immediately to be executed in parallel to all other components because of a temporally not necessarily fixed stimulus (e.g. a fault indication from the process). Such a program component is called task; in order to fix their urgency, tasks can be provided with priorities.

For the declaration and the working together of tasks with each other and with the technical process, PEARL offers the following possibilities:

- declaration of tasks, e.g.

  supplies: **TASK PRIORITY** 2;
  task object (declarations, statements)
  END;

- starting (activation), e.g.
  ACTIVATE supplies;

- terminating, e.g.
  TERMINATE printing;

- suspending, e.g.
  SUSPEND statistics;

- continuing, e.g.
  CONTINUE statistics;

- resuming, e.g.
  AFTER 5 SEC **RESUME**;

According to the requests of automation tasks, some of these statements can be scheduled for (repeated) execution, e.g. if a time, the end of a duration, or a message arrives:

WHEN ready **ACTIVATE** supplies;
(meaning: every time when the interrupt ready occurs, the task supplies has to be activated)

A temporal periodical start can also be scheduled:

AT 12:0:0 **ALL** 1 SEC **UNTIL** 12:15:0 **ACTIVATE** measuring;

Different tasks execute their statements independently of one another, provided relevant measures do not prevent that. However, sometimes a synchronisation of two or more tasks is necessary, e.g. if a task produces data and stores them into a buffer. Here the producer must not work faster than the consumer. More complex synchronisation problems occur, if e.g. a task has to access to a file exclusively (because writing), whereas others can access simultaneously (because reading). In order to solve these synchronisation problems, PEARL contains the data types SEMA and BOLT with corresponding statements, such as

- **REQUEST** conveyor-belt;
- **RELEASE** communication-buffer;

It is fundamental that these multitasking statements are *computer independent*, i.e. PEARL programs can run on iRMX, OS/2, UNIX or VAX/VMS systems without any changes.

In addition to this, there is the great advantage that the PEARL programmer can program his multitasking statements in a *problem oriented way* with high comfort without becoming deeply involved in the peculiarities of the various operating systems — e.g. in the handling of fork and message queue mechanisms of Unix. The conversion of problem oriented multitasking statements of PEARL into the mechanisms of multitasking operating systems is taken over by the PEARL compiler.

2.2 Possibilities of Input and Output

The transmission of data to or from devices of the standard peripherals (printer, hard disk etc.) or process peripherals (sensors, actuators, etc.), respectively, as well as the control of files take place in PEARL with the help of computer independent statements.

Devices and files are summarised by the term data station. Two kinds of data transmission are distinguished essentially:

- The transmission of data without format control, i.e. without conversion into an external representation:

  This kind of data transmission is used for file communication that allows for sequential and direct access as well as for the transmission of process data.

  Examples:

  READ record **FROM** file **BY** POS (10);
  WRITE data-set **TO** logbook;
  TAKE measured-value **FROM** TemperatureSensor;
  SEND on **TO** motor;

- The transmission of data with format control, i.e. with conversion between internal format and external representation with those possibilities available at the data station:

This means e.g. the representation in the characters of a character set of the data station.

Examples:

PUT event **TO** printer **BY** F(5);
GET receipt **FROM** terminal;

The names of the data stations can be chosen freely. This is reached by the partitioning of a PEARL program into computer dependent and mostly computer independent parts.

In order to address special devices, the compiler offers a driver interface, to which the PEARL programmer himself can connect device drivers.

2.3 Program Structure

Program systems for the solution of complex automation tasks should be structured in a modular way. PEARL meets this requirement, because a PEARL program consists of one or several separately compilable modules.

Connections between modules are possible by means of so-called global objects (e.g. variables, procedures, tasks).

In order to be able to program the statements for data transmission and to schedule the reactions to the events from the technical process (interrupts) or of the computer hardware (signals) computer independently, a module is usually structured into a system part and a problem part.

In the system part, the used hardware configuration is described. Particularly freely chosen names can here be assigned to the devices and their connections, the interrupts and signals. Thus, the following example means that a valve is connected to the connector 3 of a digital output unit called by the computer specific system name DIGOUT (1). The valve, i.e. connector 3, is to be called by the freely selectable computer independent user name “valve”:

valve: DIGOUT (1) * 3;

Using the user name introduced in the system part, the actual algorithm for the solution of the automation task is programmed computer independently in the problem part, e.g.:

SEND on **TO** valve;

In order to structure the algorithms, (named) blocks, procedures and tasks (parallel activities) are available.

2.4 Data Types

The following data types are available in PEARL:

- **FIXED** and **FLOAT** with specifyable precision
- **BIT** and **CHARACTER** strings with specifyable length
- **CLOCK** and **DURATION** for times and durations
- references **(REF)** for indirect addressing

- devices and files **(DATION)** for standard and process input and output
- interrupts **(INTERRUPT)** for external interrupts
- signals **(SIGNAL)** for internal exceptional situations
- semaphores **(SEMA)** and bolt variables **(BOLT)** for coordinating the access of tasks to shared object

From this, the user himself can build up more complex data structures like arrays, hierarchical structures (STRUCT) and lists; with the help of the TYPE definition, they can be declared as new, problem oriented data types as well. Furthermore, PEARL allows the introduction of new, problem oriented operators for any data structures with the help of the OPERATOR definition.

2.5 Control Structures

The following control structures are available:

- conditional statement

  IF expression
  THEN statement···
  ELSE statement···
  FIN;

- statement selection_1

  CASE expression
  ALT (alternative 1) statement···
  ALT (alternative 2) statement···
  ...
  OUT statement...
  FIN;

- statement selection_2

  CASE Case_Index
  ALT (Case_List) statement···
  ALT (Case_List) statement···
  ...
  OUT statement...
  FIN;

- loops

  FOR ControlVariable **FROM** start **BY** StepLength **TO** end
  REPEAT
  statement···
  END;

  WHILE condition
  REPEAT
  statement···
  END;

- exit statement

  EXIT block;

Chapter 3

Rules for the Construction of PEARL Language Forms

A PEARL program can be written free of format; particularly, it is not necessary to pay attention to the fact that a statement starts in a certain column.

All elements of a PEARL program are produced from characters of the following character set. Certainly, character string constants and comments may contain any character that is allowed by ISO 646 and national variants of ISO 646.

3.1 Character Set

The character set of PEARL is a partial set of the ISO 7 bit character set (ISO 646). It contains

- the upper-case letters A to Z
- the lower-case letters a to z
- the digits 0 to 9, and
- the special characters

_	underline
	space (for better clearness sometimes, underline _ is used)
!	exclamation mark (start of the line comment)
'	apostrophe
(	open round parenthesis
)	close round parenthesis
,	comma
.	full stop
;	semicolon
:	colon
+	plus sign (e.g. for addition, algebraic sign)
-	minus sign (e.g. for subtraction, algebraic sign)
*	asterisk (e.g. for multiplication)
/	oblique stroke (e.g. for division)
=	equals sign (e.g. for assignment)

<	less sign
>	greater sign
[	square bracket
]	square bracket
\	backslash (for control signs in character strings)

The following character combinations are interpreted as an entity (a compound symbol):

:=	assignment symbol
**	exponentiation symbol
/*	start comment
*/	end comment
//	symbol for integer division
==	equals symbol
/=	not equals symbol
<=	greater or equal symbol
>=	less or equal symbol
<>	cyclic-shift symbol
><	concatenation symbol
'\	start of a control character sequence in character string constants
\'	end of a control character sequence in character string constants

If not all symbols required for program notation are available on a device, the following character sequences can be used alternatively:

LT	for <
GT	for >
EQ	for ==
NE	for /=
LE	for <=
GE	for >=
CSHIFT	for <>
CAT	for ><
(/	for [
/)	for]

3.2 Basic Elements

A PEARL program is built up from the following basic elements:

- identifiers
- constants
- delimiters (meaning special characters and compound symbols), and
- comments

Character sequences for identifiers and constants must be followed by delimiters or comments.

3.2.1 Identifiers

Identifiers are used for constructing names of objects (e.g. numerical variables, procedures). They consist of a sequence of letters (upper-case or lower-case), the underline and/or numerals; this sequence has to start with a letter. For identifiers, PEARL distinguishes between upper-case and lower-case letters, i.e., valve, VALVE and Valve denote different objects.

Examples: counter_1, DISPO, wait

Some words have a specific meaning at prescribed positions in the PEARL program; these words are called keywords. E.g., the words BIT or GOTO are such keywords. The appendix contains a list of all keywords. They must not be used as identifiers and have always to be written with upper-case letters.

3.2.2 Constants

Constants are integer numbers, floating point numbers, bit strings, character strings, times and durations. They are described in Chapter 5 together with the corresponding variables.

3.2.3 Comments

Comments are used to explain the program and are of no importance for running of the program. There are two kinds of comments: One kind may cover several lines and is put in brackets by the compound symbols "/*" and "*/". Within these brackets, any characters may occur except the compound symbol "*/" for the end of the comment.

The other kind, the line of comment, starts with an exclamation mark "!" and is terminated by the end of the line.

Comments may be inserted wherever spaces are allowed.

Examples:

```
/* This comment is not ended
       by the end of the line */
! This comment is limited to 1 line
```

3.3 Construction of Language Forms

In the following chapters, the language forms allowed in PEARL are described. In order to make these descriptions exact and as compact as possible, some formal possibilities are needed apart from the verbal formulation:

Each language form has a name, by which it is defined with the help of the (meta) symbol "::=" :

```
NameOfTheLanguageForm ::=
       DefinitionOfTheLanguageForm
```

Example:

```
UpperCaseLetter ::=
```

A or B or ... or Z

Digit ::=
0 or 1 or ... or 9

As this example shows, the definition of a language form may contain elements that are given alternatively when constructing the language form. To shorten that, in the following the alternatively possible elements are divided by the symbol "|":

Example:

Letter ::=
A | B | ... | Z | a | b | ... | z

Digit ::=
0 | 1 | 2 | ... | 9

If one element shall occur on different occasions, but at least once, it is to be provided with three superscribed dots.

Example:

SimpleInteger ::=
Digit $^{\cdots}$

To express that an element may be missing while constructing the language form, it is given in square brackets "[" ... "]" (in case of doubt, the square brackets are printed boldly).

Example:

Identifier ::=
Letter [{ Letter | Digit | _ } $^{\cdots}$]

Here, already two further rules were used: The definition of a language form may again contain names of language forms; furthermore the braces and square brackets are also used to put together elements to new elements. Thus, the last example is equivalent to the following:

Example:

Identifier ::=
Letter [Letter | Digit | _] $^{\cdots}$

For the description of lists whose elements are separated by a certain symbol, the list element and the delimiter are given in the form

ListElement [Delimiter ListElement] $^{\cdots}$

to define the corresponding language form.

Example:

IdentifierList ::=
Identifier [, Identifier] $^{\cdots}$

For a better understanding or a more exact description of the definition of a language form, elements are often provided with an explanatory or restricting comment that is separated from the element by the symbol §.

Example:

DeviceList ::=
    Identifier§Device [, Identifier§Device] $^{\cdots}$

Chapter 4

Program Structure

4.1 Modules

A PEARL program is constructed of one or several parts, so-called modules, which are translated independently. Each module consists of a system part and/or a problem part.

The general form of a PEARL program reads as follows:

PEARL program ::=
    Module ···

Module ::=
    MODULE [(Identifier§OfTheModule)];
        { SystemPart [ProblemPart] } | ProblemPart
    MODEND;

SystemPart ::=
    SYSTEM; [UserNameDeclaration ···]

ProblemPart::=
    PROBLEM; [{ Declaration | Specification | Identification} ···]

In the system part, the connections of the projected computer are described with the elements of the technical process (sensors, actuators, etc.) and the standard peripherals (keyboard, monitor, printer, disks, tapes, etc.). The programmer can assign freely selectable names to the entries of the interrupt controller and the peripherals addressed in the I/O statements (in the problem part) to refer to these (computer independent) names in the problem part.

In the problem part, the algorithm for solving the given automation problem is described. For this, the programmer declares the following objects:

- variables and constants for integers, floating point numbers, bit strings, character strings, durations, times, references
- labels
- procedures for frequently occurring partial evaluations
- tasks for the temporarily parallel execution of tasks

- blocks for structuring procedures and tasks
- interrupts
- signals
- synchronisation variables (Sema and Bolt variables) as well as
- data stations and formats for input/output

The required statements are given in the procedures and tasks, together with other "local" declarations which are only needed there. In general, objects may not be used (in statements) until they are declared.

Example:

MODULE;

SYSTEM;
    description of the connections and introduction of names for the peripherals

PROBLEM;
    declaration of constants and variables
    declaration of interrupts
    declaration of data stations

    declaration of a task
        declaration of local constants and variables
        statements

    declaration of a procedure
        declaration of local constants and variables
        declaration of local procedures
        statements

...

MODEND;

Objects are declared at module level, i.e., outside procedures and tasks, or in procedures and tasks. Objects declared at module level are known in the entire module and can be used or, if needed, changed by each task and procedure of the module, when mentioning the identifier. A declared object in the task or procedure is only known in the respective task or procedure and can only be used or, if needed, changed there.

4.2 Declarations and Specifications

Objects are introduced by *declaration* or *specification.*

4.2.1 Declaration (DCL)

The declaration serves to introduce an object and its name, i.e., when evaluating the declaration, memory space for the object is allocated, and up from now, it can be accessed under the name given in the declaration.

At the module level or in a procedure or task, an object may be declared only once. If an identifier X is declared as object both at module level and in a procedure or task, two objects are introduced: In the

respective procedure or task, identifier X refers to the object (locally) declared there, outside the procedure or task it refers to the object declared at module level (cf. section 4.3, Block Structure).

Example:

```
PROBLEM;
    DECLARE x FLOAT;   ! 1st declaration at module level
    DECLARE x FIXED;   ! 2nd declaration at module level (wrong)

P:  PROCEDURE;
    DECLARE x FIXED;   ! declaration in procedure P (permitted)
    ...
    x := 3;            ! assignment to the local variable x
    ...
    END;               ! P

T:  TASK;
    ...
    x := 5;            ! assignment to variable x declared at module level
    ...
    END;               ! T

...
```

The different declaration forms are treated with the various objects in the subsequent chapters.

In procedures and tasks,

- tasks
- interrupts
- signals
- synchronising variables
- data stations, as well as
- formats

must not be declared or specified.

Table 4.1 shows where which objects may be declared or specified.

4.2.2 Specification (SPC) and Identification (SPC IDENT)

With a specification, an already declared object is referred to. This is meaningful for objects which are declared in a module, and which shall be used in other modules (cf. 4.4, References between Modules), but also for introducing additional names for already declared objects in general.

Table 4.1: Permissibility of declarations

object	declaration possible on module level	task level	procedure level	block level
variable, constant	x	x	x	x
label	–	x	x	x
procedure	x	x	x	–
task	x	–	–	–
block	–	x	x	x
Sema, Bolt variable	x	–	–	–
data station, format	x	–	–	–
type	x	x	x	x

Example:

Object c of type FIXED (15) shall get xx as 2nd name — or formulated in a different way: Object x shall be *identified* with name xx:

PROBLEM;
 ...
 DECLARE x FIXED;
 ...
 SPECIFY xx FIXED IDENT (x);
 ...
 xx := 7; ! assignment to object x

In general, the form of identification reads as follows:

Identification ::=
 { **SPECIFY** | **SPC** } Identifier [AllocationProtection] Type IdentificationAttribute ;

IdentificationAttribute ::=
 IDENT (Name§Object)

The given type has to correspond to the type of the named object. More details are defined when presenting the various objects.

4.3 Block Structure, Validity of Objects

Blocks are used to structure task or procedure bodies and to influence the scope and the life-span of PEARL objects. A block is a summary of declarations and statements:

Block::=
 BEGIN
 [{ Declaration | Identification } ···]
 [Statement ···]
 END [Identifier§Block] ;

Blocks are regarded as statements and may thus only occur in tasks and procedures, but there even nested. The entry into a block takes place when executing BEGIN. A block is left by the corresponding END or by a branch to an statement outside the block, e.g., by the exit statement (cf. 7.5). Jumps into a block are not allowed.

Within the blocks, no procedures may be declared!

Memory space is not allocated to the (local) objects declared in a block until the block is entered; it is abandoned when leaving the block. Like tasks, procedures and repetitions, blocks can introduce and remove objects dynamically and thus provide the opportunity to use the available memory space repeatedly.

Thus, some rules for the life-span and the scope range of these objects have to be established:

The life-span of an object is the (processing) time between the evaluation of its declaration and the execution of the end of the block (or of the repetition, procedure, task or module), where the declaration takes place.

```
MODULE;
    SYSTEM;
        printer: STDPRINT;
    PROBLEM;
        SPC printer DATION ... ;
        T: TASK;
            DCL a FIXED;
            BEGIN
                ...
                DCL x FLOAT;
                ...
            END;
            ...
            END; ! T
        ...
MODEND;
```

The scope of an object are all parts of the program where the object can be used. The following rules are to be obeyed:

- An object declared at module level is usable at the module level and in all tasks and procedures of this module (however, see 4.4), even in all encapsulated procedures, blocks and repetitions with the following exception: The scope range is restricted, if in one of the tasks or procedures another object is declared under the same name.
- An object declared in a task, procedure, repetition or block is usable in this task, procedure, repetition or block and all encapsulated procedures, repetitions and blocks with the following exception: The scope is restricted, if in one of the encapsulated procedures, repetitions or tasks another object is declared under the same name.

Example:

```
PROBLEM;
    DCL x FIXED;
    ...
    SPC xx FIXED IDENT (x);
    T: TASK;
        ...
        END; ! T
    P: PROC;
        DCL y FIXED;
        x := 2;
        BEGIN
            DCL x FLOAT;
            x :=3 ;
```

```
                ...
                BEGIN
                    DCL x DUR;
                    ...
                END;
                ...
            END;
            y := x; ! y = 2
            ...
            END; ! P
    ...
MODEND;
```

After END, blocks can be provided with identifiers, so that encapsulated blocks can be left deliberately with the help of the exit statement (cf. 7.5).

4.4 References between Modules

If a PEARL program consists of several modules, it can be necessary to use objects declared in a module and occupying memory space there (data, procedures, etc.) in other modules as well. For this reason, these (global) objects are declared with the global attribute at module level in the module where they are to occupy memory space, and specified with the global attribute at the module level in the other modules. In this way, the scope of objects declared at module level can be extended.

Example:

```
MODULE (a);                         MODULE (b);
PROBLEM;                            PROBLEM;
    ...                                 ...
    DCL x FIXED GLOBAL;                 SPC x FIXED GLOBAL (a);
    ...                                 ...
    x := 2;                             x := 3;
    ...                                 ...
MODEND;                             MODEND;
```

All data stations, interrupts and signals given in the system part of a certain module are regarded as declared (implicitly) with the global attribute. Thus, they are only specified in the problem parts of the program; here the global attribute can be omitted in the problem part of the same module, in all other modules it must be defined.

The general form of the global attribute reads:

GlobalAttribute ::=
    GLOBAL [(Identifier§OfaModule)]

When specifying an object, all attributes have to be taken over from its declaration, except for a given priority, precision or length. In the latter exceptional case, the precision or length defined in the corresponding length declaration is applied for the range of the program where the specification is valid.

Example:

```
MODULE;                         MODULE;
```

```
PROBLEM;
    T: TASK PRIO 3 GLOBAL;
            ! task body
        END; ! T

    P: PROC (A(,) FIXED IDENT)
                    GLOBAL;
            ! procedure body
        END; ! P
    ...
MODEND;
```

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

The different forms of specifying global objects are defined when presenting the objects.

4.5 Execution of a Program

After the loading of a PEARL program, the PEARL run time system automatically starts all tasks marked by the attribute MAIN according to their priority. All tasks provided with MAIN have to be declared in the same module.

Example:

```
MODULE (Main);
SYSTEM; ...
PROBLEM;
start:          TASK MAIN;
                ! activating and scheduling of other tasks
                END; start

measuring:      TASK PRIO 1;
                ! task body
                END; ! measuring
...
MODEND;
```

After the loading, the task start is started first.

Chapter 5

Variables and Constants

5.1 Declaration and Specification of Variables and Constants

Upon its execution, a PEARL program uses and modifies integers, floating point numbers, bit strings, character strings, times and durations. These data occur in form of constants or as values of variables. Constants are identified by their notations and keep their values during the entire program execution. Variables denote data (their values) which can change during the program execution.

Generally, the value range of a variable is limited to a single kind of data, e.g. bit strings, which determines the type of the variable (Bit string variables have only bit strings as values, e.g.). This type has to be defined with its identifier, when declaring or specifying a variable.

Example:

DECLARE switch **BIT**;	/* declaration of a variable switch of type bit string of length 1 */
SPECIFY status **BIT**(16) **GLOBAL**;	/* specification of a global variable status of type bit string of length 16 */

A variable denotes *one* data element, e.g. *one* integer, *one* bit string, etc.; in the following, such scalar variables are treated. The possibilities to summarise scalar variables to arrays and structures are described in 5.10 and 5.11.

When declaring a variable, its type has to be given in a type attribute; if different variables with the same type attribute are to be declared, this can take place in form of a list in a declaration, e.g., by

DECLARE (x,y,z) **FLOAT**;

the three variables x, y, and z are declared with the type attribute FLOAT. For simpler notation, different declarations may be formulated in a declaration by separating them by commas:

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

Summarised, scalar variables can be declared as follows:

ScalarVariableDeclaration ::=
{ **DECLARE** | **DCL** } VariableDenotation [, VariableDenotation] ··· ;

VariableDenotation ::=
IdentifierDecnotation [AllocationProtection] TypeAttribute [GlobalAttribute]
[InitialisationAttribute]

IdentifierDenotation ::=
Identifier | (Identifier [, Identifier] ···)

TypeAttribute ::=
SimpleType | TypeReference | Identifier§ForType

SimpleType ::=
TypeInteger | TypeFloatingPointNumber |
TypeBitString | TypeCharacterString |
TypeTime | TypeDuration

The general form of specifying scalar variables reads as follows:

ScalarVariableSpecification ::=
{ **SPECIFY** | **SPC** } VariableDenotationS [, VariableDenotationS] ···;

VariableDenotationS ::=
IdentifierDenotation [AllocationProtection] TypeAttribute
{ GlobalAttribute | IdentificationAttribute }

Allocation protection and initialisation attribute are described in 5.14, identification and global attributes were already defined in 4.2 and 4.4.

Subsequently, the different type possibilities are presented; variables of type Sema, Bolt, Interrupt or Signal are considered in 9.3, 9.4 and 10.

5.2 Integers (FIXED)

Integers can be written in decimal or dual representation. The decimal representation of an integer consists of a sequence of decimal digits. The dual representation consists of a sequence of the digits 0 and 1, terminated with character B.

Examples:

integer	decimal pres.	dual pres.
6	6	110B
123	123	1111011B

Furthermore, the precision of the representation can be defined for a numerical constant by noting the number of bit positions, in which it shall be represented computer internally without signs, in brackets behind the numerical constant.

Example: 123(31) Integer 123 is presented in 31 bit positions.

If no precision is given, the precision is derived from the constants.

Variables for integers are declared with the type attribute FIXED.

TypeInteger ::=
    FIXED [(Precision)]

Precision ::=
    IntegerWithoutPrecision§GreaterZero

The precision gives the number of bit positions in which the corresponding value of the variable (without signs) is represented. If the precision declaration is lacking, the precision defined in a length specification (see 5.6) is inserted. Usual precisions are 15 or 31.

Example:

DCL counter **FIXED**(31),
    (i,j,k) **FIXED**;
...
i := 2;

5.3 Floating Point Numbers (FLOAT)

Constants for floating point numbers can be represented as a sequence of

1. a point, an integer and possibly an exponent to the basis 10, where an exponent consists of a sequence of the character E, possibly a plus or minus sign, and an integer

   Example: .123 (corresponds to 0.123)
   .123E2 (corresponds to 12.3)
   .123E-1 (corresponds to 0.0123)

2. an integer followed by a sequence given in 1.

   Example: 3.123E2 (corresponds to 312.3)

3. an integer, a point, and possibly an exponent

   Example: 3. (corresponds to 3.0)

4. an integer and an exponent

   Example: 3E-2 (corresponds to 0.03)

Analogously to integers, even constants for floating point numbers can be defined with precision.

Variables of type floating point number (with integers or floating point numbers as values) are declared with type attribute FLOAT.

TypeFloatingPointNumber ::=
    FLOAT [(Precision)]

The statements of 5.2 are valid; usual precisions are here 23 and 53, respectively.

Example:

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

5.4 Bit Strings (BIT)

A bit string constant can be given in binary form (B1), in form of tetrades (B2), octades (B3), or in hexadecimal form (B4).

The form Bi (i=1,...,4) consists of an apostrophe, a sequence of

- digits 0 and 1 in case B1
- digits 0 to 3 in case B2
- digits 0 to 7 in case B3
- digits 0 to 9 and letters A to F in case B4

followed by an apostrophe and the corresponding attribute B1 or B2 or B3 or B4.

Example:

'110010100111'B1	corresponds to
'302213'B2	corresponds to
'6247'B3	corresponds to
'CA7'B4	

Instead of B1, B can be written in a shorter way. The following tables show the assignment between the binary form and the other forms:

B2	B1
0	00
1	01
2	10
3	11

B3	B1
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

B4	B1
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Variables for bit strings are to be declared with type attribute BIT.

TypeBitString ::=
    BIT [(Length)]

Length ::=
    IntegerWithoutPrecision§GreaterZero

Length gives the number of elements of the bit string. If the length option is lacking, it had either to be defined in a length definition (see 5.6), or 1 is assumed as length.

Example:

DCL X_coord **BIT**(2),
    Y_coord **BIT**(8);
...
X_coord:='01'B;
Y_coord:= 'A9'B4;

The addressing of parts of bit strings is treated in 6.1, Expressions.

5.5 Character Strings (CHARACTER)

A character string constant consists of an apostrophe, a sequence of any characters (besides of apostrophe) and an apostrophe.

Example: 'fault no.: '

If, however, the character string has to contain an apostrophe, it has to be represented by two subsequent apostrophes.

Example: 'fault"no.: '

Control characters can be inserted into character string constants with the help of the switch symbols " '\ " and " \' ". The control characters are given as pairs of sedecimal digits.

Examples:

'This string '\ 0D 0A \' contains two control characters'
"\1B\' control character at the start of the string'
'control character at the end of the string'\00\"
"\00\" /* string consisting of one single control character */

Example:

'By changing over to the control character sequence'\20
\' very long character string constants'\20
\' can be created (independently from the used editor).'

Variables for character strings are declared by the type attribute CHARACTER.

TypeCharacterString ::=
    { **CHARACTER** | **CHAR** } [(Length)]

Length gives the number of characters. If no length is indicated, it had either to be defined in a length specification (see 5.6) or length 1 is assumed.

Examples:

DCL ArticleIdentificator **CHAR**(6);
...
ArticleIdentificator:='BCD/27';

The addressing of parts of character strings is treated in 6.1, Expressions.

5.6 The Length Definition

With the length definition, the precisions and lengths for such number and string objects are defined, whose precisions and lengths are not determined by notation (for constants) or declaration (for variables).

LengthDefinition::=
  LENGTH { { **FIXED** | **FLOAT** } (precision)
    | { **BIT** | **CHARACTER** | **CHAR** } (length) };

Example:

```
PROBLEM;
    LENGTH FIXED(15);
    LENGTH FLOAT(53);
    DCL A FIXED,            /* A is of type FIXED(15) */
        X FLOAT,            /* X is of type FLOAT(53) */
        Y FLOAT(23);        /* Y is of type FLOAT(23) */
    ...
```

For a length definition, the same validity rules are valid as for variable declarations (see 4.3, Block Structure).

5.7 Times (CLOCK)

A time constant consists of a positive integer to state the hour, an integer between 0 and 59 for the minute, and a floating point number between 0 and 59.999... for the second — separated by a colon, each. The hour is interpreted modulo 24.

Example:

11:30:00	means 11.30
15:45:3.5	means 15.45 and 3.5 seconds
25:00:00	means 1.00

Variables for times are declared with type attribute CLOCK.

TypeClock ::=
  CLOCK

Example:

```
DCL time CLOCK;
...
time:=12:30:00;
```

5.8 Durations (DURATION)

A duration constant is composed of stating an hour, a minute, and a second, however, single pieces of specifications may be lacking. An hour is given by an integer and the character sequence HRS, a minute by an integer and the character sequence MIN, a second by a floating point number and the character sequence SEC.

Examples:

5 **MIN** 30 **SEC** means 5 minutes and 30 seconds
.05 **SEC** means 50 milliseconds
5 **HRS** 10 **SEC** means 5 hours and 10 seconds

The integers and floating point numbers in time constants must not contain precision specifications.

Variables for durations have to be declared with the type attribute DURATION.

TypeDuration ::=
    DURATION | **DUR**

Example:

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

5.9 References (REF)

For the indirect addressing, so-called reference variables (pointer variables, pointers) are provided in PEARL. Reference variables have the names of variables as values (they point at variables). Analogously to the variables introduced up to now, the value range of a reference variable is limited to a type of variables given when declaring the reference variables (see, however, 5.13, Interference of Data Structures).

The value of a reference variable, the referenced variable, is referred to with the help of the monadic operator CONT (“content”); this process is called “dereferentiation”.

Examples:

```
DCL (k,l) FIXED,
    x FLOAT
    (rk1,rk2) REF FIXED,   /* fixed reference variable */
    rx REF FLOAT;          /* float reference variable */
rk1:=k;                    /* rk1 points at k */
rk1:=l;                    /* rk1 points at l */
rk2:=rk1;                  /* rk2 points at l, pointer assignment */
rx:=x;                     /* rx points at x */
rx:=k;                     /* wrong, type unequal */
rx:=rk1;                   /* wrong, type unequal */
l:=2;
k:=CONT rk1;               /* k obtains value 2 */
rk2:=3;                    /* wrong, 3 is no variable */
```

```
CONT rk2:=3;          /* l obtains value 3 */
CONT rk2:=k;          /* l obtains value 2 */
```

Instead of k:=CONT rk1, k:=rk1 can be written in a simpler way; i.e., operator CONT may be omitted on the right side of an assignment ("implicit dereferentiation").

In general, it holds:

TypeReference ::=
    REF [VirtDimensionList] [AssignmentProtection]
    { SimpleType | Identifier§ForType | TypeStructure |
        TypeDation | **SEMA** | **BOLT** | **INTERRUPT** | **IRPT** | **SIGNAL** |
        TypeProcedure | **TASK** | TypeVOID | **CHAR**()
    }

VirtDimensionList ::=
    ([, ···])

Dereferentiation ::=
    CONT Name§ReferenceVariable

The definition of the language form TypeReference shows, which type a variable may have, at which a reference variable shall point. Particularly, reference variables may also point at arrays and structures (cf. 5.10 and 5.11). The number n of commas in the virtual dimension list states that the referenced variable is an (n + 1) dimensional array. On the other hand, reference variables must not point at reference variables; referenced arrays and structures, however, may possess reference variables as elements and components. This can be used, e.g., to interlink structures, or to build up lists in general. Section 5.12, Type Definition, gives an example for that.

TypeStructure and TypeDation are defined in 5.11 and 10.2, TypeProcedure and TASK in 8.3 and 9.11, TypeVOID and REF CHAR() in 6.4.2 and 5.9.1

If a reference variable R points at an array F with elements F (i,j,k,...) or at a structure S with components S.Ki, the elements of F or the components of S are addressed (taking R as starting-point) with R (i,j,k,...) or R.Ki without using CONT.

Furthermore, a reference variable is dereferentiated implicitly,

- if it is the actual parameter of a procedure call, and if the formal parameter belonging to it is no reference variable.
- if it is used as operand of a dyadic operator, as far as the latter is not defined for values of reference variables like IS (see below).

Example:

```
DCL rk REF FIXED,
    k FIXED;
    ...
    rk:=k; k:=2;
    k:=rk+1; /* equivalent k:=k+1; */
```

E.g., to mark the end of a string, a reference variable NIL (zero pointer) is provided, possessing a certain, constant value.

To compare the values of reference variables, the dyadic operators IS or ISNT can be used; IS (ISNT) provides the result '1'B ('0'B), if both the given reference variables possess the same (different) value(s), otherwise it provides the result '0'B ('1'B).

Example:

```
DCL NextOrder REF type_order;
...
IF NextOrder IS NIL THEN
    ...
FIN;
```

5.9.1 Variable Character String Pointers (REF CHAR())

Variably long character strings can easily be processed with special pointer objects, so-called variable character string pointers.

Example: Declarations of variable character string pointers

```
DCL str_ptr REF CHAR();                     /* pointer declaration */
SPC print ENTRY (REF CHAR()) GLOBAL; /* pointer as parameter */
```

A pointer of type "REF CHAR ()" contains, besides of the address of a character string, two counters storing the maximum length and the actually used length of the referenced character string variable. When assigning the address of a character string variable to a variable character string pointer, both counters are initialised with the length of the referenced character string variable.

Example: Initalisation of a variable character string pointer

```
DCL s1      CHAR(20);
DCL s2      CHAR(100);
DCL str_ptr REF CHAR();
str_ptr:=s1;                /* adr=s1, length=20, max=20 */
str_ptr:=s2;                /* adr=s2, length=100, max=100 */
```

Thus, the variable pointer differs at first not from a character string pointer with constant length. When assigning to the character string variable via this pointer, the length of the assigned character string, however, is stored in the counter for the actual length, and the otherwise usual filling of the character string with blanks up to the maximum length is omitted.

Example: Assignment to the referenced variable

```
str_ptr:=s1;                     /* adr=s1, length=20, max=20 */
CONT str_ptr:='PEARL 90';        /* adr=s1, length=8, max=20 */

str_ptr:=s2;                     /* adr=s2, length=100, max=100 */
CONT str_ptr:='Werum GmbH'; /* adr=s2, length=10, max=100 */
```

For the subsequent accesses to the character string via this pointer, only the actual length is used.

Example:

```
DCL s1       CHAR(20);
DCL s2       CHAR(100);                    /* filler */
DCL str_ptr  REF CHAR();                   /* variable pointer */

str_ptr:=s2:                               /* adr=s2, length=100, max=100 */
CONT str_ptr:='D21337 ' CAT 'Lueneburg';   /* adr=s2, length=16, max=100 */
```

After this assignment, the variable "s2" contains the characters "D21337 Lueneburg" in the first 16 positions, the other characters were not changed by the assignment. The length index of "str_ptr" was set to the actual length 16.

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

For this assignment, "CONT str_ptr" generates the character string value 'D21337 Lueneburg' (with type "CHAR(16)"). Due to the assignment rules of PEARL, the character string is extended by four blanks, before this value is assigned to variable "s1" (with type "CHAR(20)").

Variable character string pointers turn out particularly useful as formal parameters of procedures. Apart from the address of a character string variable, the compiler passes its actual length at the call position. Thus, e.g. an error routine can give out error texts of different length.

Example: Character string pointer as parameters

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

...
CALL print_error (s1);
CALL print_error (s2);
```

Even character string constants may be passed, if the formal parameter was defined with type "REF INV CHAR()".

Example:

```
SPC print_message ENTRY (REF INV CHAR()) GLOBAL;

...
CALL print_message ('text 1');
CALL print_message ('longer text 2');
```

A variable character string pointer can be used in all expressions analogously to constant character string pointers. The length check belonging to the type check takes first place at run time.

A variable character string pointer is not allowed as result value of a function. Type "CHAR()" may only be used in combination with "REF", i.e., parameter type "CHAR()IDENT" is not allowed.

5.10 Arrays

If possible, objects of the same kind are summarised in one identifier, and the various objects are addressed with indices.

Example:

An embedded controller controls three devices G(1), G(2), and G(3). For the output of the bit string '0001'B to the controller, G(1) shall be switched on, device G(2) for '0010'B, and device G(3) for '0100'B. Here it suggests itself to summarise the three switch-on signals under one identifier, e.g., switch_on, and to address them with indices:

DCL switch_on(1:3) **BIT**(4),
        i **FIXED**;
switch_on(1):='0001'B;
switch_on(2):='0010'B;
switch_on(3):='0100'B;
...

/* taking over value of i (index of the device to be switched on) from another program part
output of switch_on (i) to the embedded controller */

In general, scalar variables of the same type can be combined into n-dimensional arrays (n=1,2,3,4).

When declaring the array, an identifier is assigned to it; the various array elements (scalar variables) are addressed under this identifier and the definition of their positions within the array (the index).

Thus, with the help of

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

a 2-dimensional array is declared, where the first dimension has the lower boundary 1 and the upper boundary 2, i.e., length 2, whereas the second dimension has the lower boundary 0 and the upper boundary 3, i.e., length 4. This means that Koeff consists of the 8 scalar fixed variables

Koeff(1,0)	Koeff(1,1)	Koeff(1,2)	Koeff(1,3)
Koeff(2.0)	Koeff(2,1)	Koeff(2,2)	Koeff(2,3)

The general form of a array declaration reads as follows:

ArrayDeclaration ::=
        { **DECLARE** | **DCL** } ArrayDenotation [, ArrayDenotation] $^{\cdots}$;

ArrayDeclaration ::=
        IdentifierDenotation DimensionAttribute [AssignmentProtection] TypeAttribute
        [GlobalAttribute] [InitialisationAttribute]

DimensionAttribute ::=
        (BoundaryDenotation§FirstDimension [, BoundaryDenotation§FurtherDimension] $^{\cdots}$)

BoundaryDenotation ::=
        [Boundary§Lower:] Boundary§Upper

Boundary ::=
        [-] IntegerWithoutPrecision | Identifier§NamedConstant |
        ConstantExpression

Thus, even negative integers are allowed as dimension boundary. The upper boundary of a dimension, however, has always to be greater or equal its lower boundary.

If the lower boundary is not given, value 1 is assumed for it. By a named constant, a variable of type FIXED(15) is understood, declared with assignment protection (INV) and initial value (INIT).

Examples:

DCL switch_on(3) **BIT**(4),
Koeff(2, 0:3) **FIXED**,
message(20) **CHAR**(12);

The one-dimensional array message contains e.g. 20 error texts, so that, in the case of error i, a program can pass the message (i) to a console (i=1,...,20).

Arrays can also be declared mixed with scalar variables in a declaration, e.g.:

DCL numb_mess **INV FIXED**(15) **INIT**(20);
...

DCL message (numb_mess) **CHAR**(12),
(i,j,k) **FIXED**,
(switch_on, switch_off)(3) **BIT**(4);

The general specification form of an array reads:

ArraySpecification ::=
{ **SPECIFY** | **SPC** } ArrayDenotationS [, ArrayDenotationS] ···;

ArrayDenotationS ::=
IdentifierDenotation VirtualDimensionAttribute [AssignmentProtection]
TypeAttribute { GlobalAttribute | IdentificationAttribute }

VirtualDimensionAttribute ::=
([, ···])

The number n of commas in the virtual dimension attribute informs that the specified array has n+1 dimensions.

In Section 1 of the Appendix it is described in tabular form, which objects may be combined to arrays.

5.11 Structures

Arrays allow to combine scalar variables of the *same* type under one identifier; the various variables are addressed by this identifier and their indices.

For many automation problems, especially for those with dispositive character, however, data structures have to be described, whose components have *different* types.

Example:

The contributions for the TV news of one day are stored on magnetic tapes (VTR); the contribution sequence of a certain news broadcast of that day is composed by the editors with the help of a computer controlling the corresponding VTR. For this, a data set is necessary for each contribution with, e.g. the following structure:

- identity label of the contribution
- archive number
- notes whether the contribution already was broadcast in the first, second or third news broadcast of that day

- initial position of the contribution on the tape
- end position of the contribution on the tape
- notes whether original sound is available
- length of the text to be stored, if needed
- a text to be stored, if needed

Such data structures can be described in a problem oriented way as structures; the above structure "Contribution", e.g., is declared as follows:

DCL Contribution **STRUCT**
[(IdentNo, Archive) **FIXED**,
already_broadcast(3) **BIT**(1),
(Start, End) **FIXED**,
OriginalSound **BIT**(1),
TextLength **FIXED**,
Text **CHAR**(200)];

(A structure is encapsulated in square brackets; they are printed boldly in the text to distinguish them from the meta-linguistic characters "[" and "]". When writing down the program, only square brackets or their substitutional symbols "(/" and "/)" are used.)

In contrast to array elements, the components of a structure are not addressed via the joint identifier and an index, but via the joint identifier and the identifier denoted in their declaration, where both identifiers are separated by a full-stop.

Example:

By the statement

Contribution.Start := 1027;

value 1027 is assigned to the component Start of structure Contribution.

As the declaration of contribution shows, components of structures may be arrays; as usual, the array elements are then addressed with their indexed identifier preceded by the structure identifier, such as in the form

IF Contribution.already_broadcast(i) **THEN...FIN**;

A structure component, however, can also be a structure itself, so that not only linear, but also hierarchical data structures can be modeled.

Example:

A staff file shall contain descriptions of the staff members; these descriptions each have the structure shown in Figure 5.1.

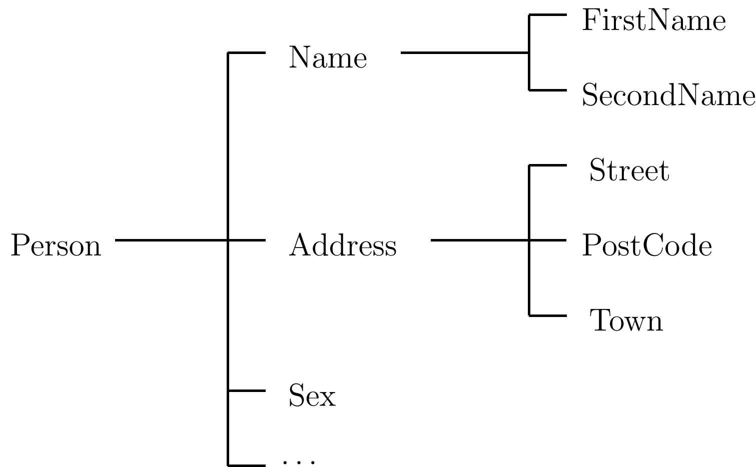

Figure 5.1:

The components Name and Address are structures themselves, namely substructures of the main structure Person which can be declared as follows:

```
DCL Person  STRUCT
          [ Name     STRUCT
                   [ FirstName CHAR(10),
                     SecondName CHAR(15),
                   ],
            Address  STRUCT
                   [ str CHAR(15);
                     PostCode FIXED,
                     Town CHAR(15)
                   ],
            Sex      CHAR(1)
            ...
          ];
...
```

Therefore, substructures can be addressed via their identifier and the preceding identifiers of the respective higher level structures.

The type of a structure is determined by the arrangement of its components and their types. Structures of the same type can be combined into arrays:

DCL Contribution(20) **STRUCT**...; /* see above */

Thus, any component of a structure is generally addressed with its (if needed, indexed) identifier and the (if needed, indexed) identifiers of all higher level structures, where the names are separated by a full-stop, each.

Example:

Person.Name.FirstName := 'OTTO';
IF Contribution(i).already_broadcast(j) **THEN...FIN**;

The general form of the structure declaration reads:

StructureDeclaration ::=
  { **DECLARE** | **DCL** } StructureDenotation [, StructurDenotation] ··· ;

StructureDenotation ::=
    IdentifierDenotation§MainStructure [DimensionAttribute][AllocationProtection]
    TypeStructure [GlobalAttribute][InitialisationAttribute]

TypeStructure ::=
    STRUCT [StructureComponent [, StructureComponent] ···]

StructureComponent ::=
    IdentifierDenotation TypeAttributeInStructureDeclaration

TypeAttributeInStructureDeclaration ::=
    [DimensionAttribute]
    { SimpleType | TypeReference | Identifier§ForType | TypeStructure }

The general form of the structure specification reads:

StructureSpecification ::=
    { **SPECIFY** | **SPC** } StructureDenotationS [, StructureDenotationS] ··· ;

StructureDenotationS ::=
    IdentifierDenotation§MainRecord [VirtualDimensionAttribute]
    [AllocationProtection] TypeRecord { GlobalAttribute | IdentificationAttribute}

5.12 Type Definition (TYPE)

A certain structure, e.g., can occur as parameter, as substructure in other structures, or as type of the transmission data of a data station. Each time, the type of this structure must be stated, which might cause extensive writing effort for complex structures. For that reason, and to increase the readability of the program, the type of a structure can be defined as new data type under a freely selectable identifier, and used under this identifier, e.g., to declare variables of this data type.

Examples:

1. **PROBLEM**;

```
...
TYPE   Message STRUCT
      [ (IdentNo, Archive) FIXED,
        already_broadcast(3) BIT(1),
        (Start, End) FIXED,
        OriginalSound BIT(1),
        TextLength FIXED,
        Text CHAR(200) ];

DCL    Content_VTR DATION INOUT Message...;

Coord: TASK;
       DCL Contribution Message;
       ...
       READ Contribution FROM Content_VTR;
       ...
       END; ! Coord
...
```

2. A task controls certain devices; it receives the control orders from another task. Since occasionally

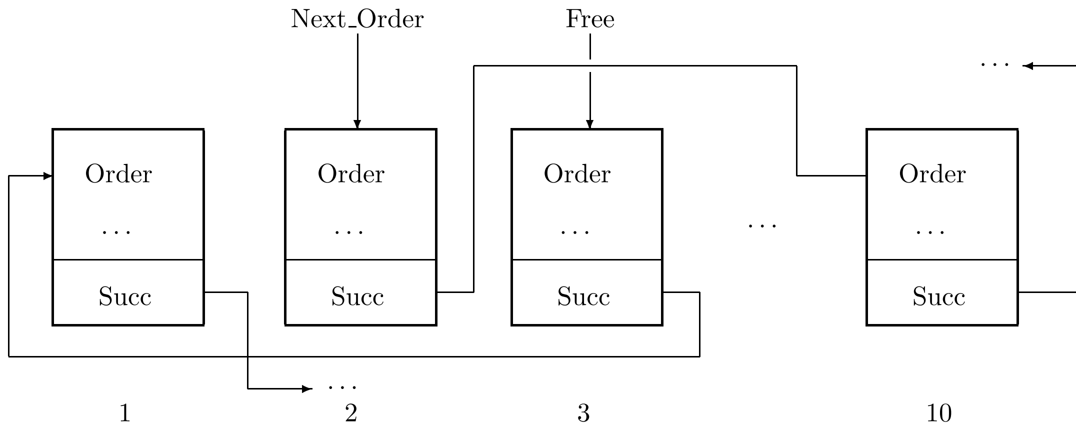

Figure 5.2:

more than one order can be existing and the orders can be of different urgency, they are buffered in form of a queue. Let the structure of the orders be of the (re-defined) type OrderType; let this even contain the urgency criteria for arranging new orders. The reference variables Next_Order and Free, respectively, as shown in Figure 5.2, point at the order to be treated next or the next free position in the queue order string, respectively; starting with Free, let the free positions be interlinked as well. Let the queue be not longer than 10 elements.

The order string can be declared as follows:

```
...
TYPE StringElement STRUCT
                [       Order OrderType,
                        ...,
                ];
DCL   OrderString (10) StringElement,
      (Next_Order, Free) REF StringElement;
...
```

After initialising and arranging various orders, a finished order can be removed from the queue as follows (let Help be a variable of type REF string element):

```
...
Help := Next_Order; Next_Order := Next_Order.Succ;
Help.Succ := Free
Free := Help
...
```

These statements are also further examples for implicit dereferentiation. The order string can be initialised as follows:

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

Generally, a new data type is defined as follows:

TypeDefinition ::=
  TYPE Identifier§ForType { SimpleType | TypeStructure }

The definition of a new data type has to take place before its use. Besides, data types defined at module

level, can be used in all tasks and procedures of the module; the usability of locally defined data types depends on the rules of the block structure.

In contrast to global objects, re-defined data types can only be used in the module where they are defined. Thus, type definitions for global objects have to be repeated in all used modules.

Example:

```
MODULE (ModuleA);
PROBLEM;
    TYPE T STRUCT [...];
    ...
    DCL x T GLOBAL;
    ...
MODEND;
```

```
MODULE (ModuleB);
PROBLEM;
    TYPE T STRUCT [...];
    ...
    SPC x T GLOBAL(ModuleA);
    ...
MODEND;
```

5.13 The Initialisation Attribute (INITIAL)

The initialisation attribute allows to assign initial values to variables upon their declaration. For variables with assignment protection (see 5.14), this is the only possibility of allocating values. The initialisation attribute is given as last attribute of the respective declaration:

InitialisationAttribute ::=
    { **INITIAL** | **INIT** } (InitElement [, InitElement] ···)

InitElement ::=
    Constant
    | Identifier§NamedConstant
    | ConstantExpression
    ...

Constant ::=
    Integer
    | FloatingPointNumber
    | BitStringConstant
    | TimeConstant
    | DurationConstant
    | **NIL**

The type of the given init element has to be compatible with the type of the declared variable in accordance with the rules for assignment (cf. 6.3):

Example:

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

Here, the variables number_devices, UGR, and OGR have the values 12, 2 and 15, i.e., the elements of a declared constant list are assigned in the sequence of writing down the declared identifier list.

Arrays and structures can also be initialised at the position of declaration. For all structure components, one initial value has to be given, each. For arrays, the initialisation list may be shorter than the number of array elements. In this case, the last initial value for the remaining elements is used repeatedly.

Example:

```
...
DCL Address STRUCT[ PostNumber  FIXED,
                    Town        CHAR(20),
                    Street      CHAR(20),
                    HouseNumber FIXED         ]
            INIT(21337, 'Lueneburg', 'Erbstorfer Landstr.', 14);
DCL colours  (3)    CHAR(7)     INIT ('black  ', 'white  ', 'red    ');
DCL all_sixes (6)   FIXED       INIT (6);
...
```

The initialisation list can, besides constants, even contain identifiers of named constants as well as expressions calculable by the compiler (cf. 5.15.1 and 5.15.2).

5.14 Assignment Protection (INV)

Variables can be declared with an assignment protection,i.e. the attribute INV (from "invariant"), in order to forbid — apart from initialisation — assignments to these variables. The attribute INV is noted immediately before the type attribute.

Example:

```
...
DCL Pi INV FLOAT INIT(3.141);


Pi := 3; /* results in error message of the compiler */
```

An assignment protection which once is associated with an object in its declaration cannot be revoked; thus it must be considered in specifications of this object, but also when transmitting it as actual procedure parameter or when using it as value of reference variables. On the other hand, in one of this ways an assignment protection may arise which was not associated in the declaration.

Example:

```
...
P:    PROC(A (,) INV FIXED IDENT, x FLOAT IDENT); /* procedure body */
      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); /* wrong */
...
```

```
CALL P (Tab, Pi);            /* wrong */
R1:=Pi;                      /* wrong */
R2:=Tab(5,7);
```

The specification of pi is wrong, because assignments to pi are allowed, and thus the assignment protection of Pi is circumvented.

The call of P is wrong, because the formal parameter x belonging to Pi is specified without assignment protection, and thus a (forbidden) value change of Pi could happen in the body of P via an assignment to x.

In contrast, in the same call, the parameter transmission from Tab to A is correct, because only an assignment protection for Tab arises which was not introduced yet.

The assignment R1:=Pi is not permitted, because otherwise the assignment protection for Pi via (allowed) assignments to CONT R1 could be revoked. The assignment R2:=Tab(5,7), however, is permitted; a value modification of the tabular element via CONT is not possible.

5.14.1 Named Constants (INV Constants)

The compiler treats objects of type FIXED, BIT, CHAR(1), CLOCK and DURATION declared with attribute INV as (named) constants. The names of these objects may be used after their declaration (with initialisation) for declaring and initialising further objects. For this, they may be applied as dimension boundary denotation in array declarations as well as in initialisation lists.

Example:

A surveillance program shall be adapted for various projects with different numbers of embedded controllers. The number of message buffers and synchronising variables belonging to them (cf. 9.3) can be adapted easily by changing the constant NumberControls:

```
...
DCL NumberControls INV FIXED INIT(11);


DCL MessageBuffer (NumberControls) CHAR(100);
DCL BufferAccess (NumberControls) BOLT;
...
```

For a further project with another number of controls, only the named constant NumberControls has to be adapted to the actual conditions. The memory areas dependent on that are adapted by the compiler.

Since the compiler does not generate any run time objects for named objects, they may not be assigned to a reference variable. Likewise, they cannot be used as actual parameters of a procedure, if the formal parameter was specified with the IDENTICAL attribute or as reference variable (cf. 8.2).

5.14.2 Constant Expressions (of Type FIXED)

In all positions where the compiler expects constants, also constant expressions may occur. These expressions may only contain constants (also named constants) as operands. They are evaluated by the compiler, and the results are inserted in the respective positions.

ConstantExpression ::=

{ + | - } FloatingPointNumber
| { + | - } DurationConstant
| constant-FIXED-Expression

Constant-FIXED-Expression ::=
Term [{ + | - } Term] ···

Term ::=
Factor [{ * | // | **REM** } Factor] ···

Factor ::=
[+ | -]
{ integer
| (Constant-FIXED-Expression)
| **TOFIXED** { CharacterStringConstant§OfLength1 | BitStringConstant }
| Identifier§NamedConstant
}
[**FIT** constant-FIXED-Expression]

Constant expressions of type FIXED can, e.g., be applied in the following positions:

- denotation of dimension boundaries in array declarations,
- precision declaration (for FIXED and FLOAT variables),
- length definition (for CHAR and BIT variables),
- in initialisation lists (also for PRESET),
- priority definition in task declarations and task statements
- in the constant lists of the CASE statement
- on the right side of an assignment

Often, several objects of a program (variables, constants) are dependent from one another. If this dependence can be described by a simple formula, a program can probably be configured by just adapting few constants.

Example:

A warning shall be given, if a buffer system is filled up to the last 10 %:

```
...
DCL MaxBuffer       INV FIXED INIT(1000);
DCL WarnBoundary INV FIXED INIT(MaxBuffer - MaxBuffer // 10);
DCL Buffer (MaxBuffer)    FIXED;
DCL ActualBufferIndex     FIXED INIT(1);

...
ActualBufferIndex := ActualBufferIndex+1;
IF ActualBufferIndex > WarnBoundary
THEN /* output of warning */
FIN;
...
```

Chapter 6

Expressions and Assignments

6.1 Expressions

Expressions are used for various language forms, e.g.,

- as index when addressing array elements:
  Identifier§Array(Expression,Expression)
  e.g., Tab(K,2*i)
- as reference value for the return statement in function procedures (see 8.1):
  RETURN(Expression);
  e.g., **RETURN**(No);
- as parameter when calling procedures (see 8.2):
  ListOfActualParameters ::= (Expression [, Expression] ···)
  e.g., **CALL** P(A, Tab (K, 2*i));
- as start condition for tasks (see 9.2.1):
  AT Expression§Time
  e.g., **AT** 12:00:00 **ACTIVATE** T;

These examples show that an expression at least can be

- a constant
- an identifier
- an indexed identifier or
- an arithmetical expression (e.g., 2*i).

Identifiers, indexed identifiers and names of structure components are generally summarised under the notion "Name":

Name ::= Identifier [(Index [, Index] ···)] [. Name] ···

Index ::= Expression§WithIntegerAsValue

Examples:

A, A(3), A(i,j,2*k), A.B, A.B.C, A(3).B.C(i,j)

The identifiers of the components of a structure can be selected independently from the identifiers outside the structure:

DCL Person **STRUCT**..., /* see above */
Town **CHAR**(15);
...
Person.Address.Town := Town;

Generally, the names denoted in expressions have to be names of scalar variables; however, in the expression lists of the output statements (cf. 10.5, 10,6, 10.7) even identifiers of arrays may be given.

In general, an expression has the form

Expression::=
[MonadicOperator] Operand [DyadicOperator Expression] ···

Monadic operators have only one operand, dyadic operators have two operands.

MonadicOperator ::=
+ | - | **NOT** | **ABS** | **SIGN** | **LWB** | **UPB** | **SIZEOF** | **CONT**
| **TOFIXED** | **TOFLOAT** | **TOBIT** | **TOCHAR** | **ENTIER** | **ROUND**
| Identifier§UserDefinedMonadicOperator

DyadicOperator ::=
+ | - | * | / | // | **REM** | ** | < | **LT** | > | **GT** | <= | **LE** | >= | **GE** | == | **EQ** | /= | **NE**
| **AND** | **OR** | **EXOR** | >< | **CAT** | <> | **CSHIFT** | **SHIFT** | **LWB** | **UPB** | **IS** | **ISNT** | **FIT**
| Identifier§UserDefinedDyadicOperator

Operand ::=
Constant | Name | FunctionCall | ConditionalExpression | Dereferentiation |
StringExcerpt | (Expression)

Examples:

- -A + B * C - D/E**2
- (A - B) / (A + B)
- F(Tab(K, 2*i)) / (F(i) - 3)
- A < B **OR** A < C
- ProcessImage_new **AND NOT** ProcessImage_old
  (the result has 1 in *that* bit place where ProcessImage_old has 0 and ProcessImage_new has 1)
- Xcoord >< Ycoord >< Zcoord
  (the three bit strings are concatenated to one bit string)

The conditional expression can be useful, e.g., in assignments or function procedures (see 8); it has the following form:

ConditionalExpression ::=
IF Expression§WithValueOfType-BIT(1)
THEN Expression **ELSE** Expression **FIN**

If the calculation of the expression following IF yields value '1'B (true), the expression following THEN is executed; if the value equals '0'B (wrong), the expression after ELSE is executed.

Examples:

1. The function procedure max shall state and return the greater one of two floating point numbers.

   max: **PROC**((X,Y)**FLOAT**) **RETURNS** (**FLOAT**);
   RETURN(**IF** X>Y **THEN** X **ELSE** Y **FIN**);
   END;
   ...
   A := max(B,C)/2.0 ;

2. Equivalent with this assignment is the following one:

   A := (**IF** B>C **THEN** B **ELSE** C **FIN**)/2.0;

The importance of dereferentiation was treated in 5.9, References. The general form reads:

Dereferentiation::=
CONT { Name§Reference | FunctionCall }

The i-th bit of a bit string can be addressed with the help of the standard attribute BIT(i) which — separated by a full-stop — is mentioned behind the name of the bit string. Therefore, a bit string B of length lg is considered as structure B, which as only component has a one-dimensional array BIT of length lg, whose elements are of type BIT(1). Analogously, the i-th character of a character string Z can be addressed with Z.CHAR(i) or Z.CHARACTER(i). The bits or characters of a bit or character string are numbered from the left to the right, starting with 1.

Example:

```
...
DCL Byte BIT(8),
    Bt BIT(1),
    i FIXED;
...
Byte:='11101111'B;
Bt:=Byte.BIT(4);   /* Bt obtains value '0'B */
Byte.BIT(2):='0'B; /* Byte has value '10101111'B */
i:=8;
Byte.BIT(i):=Bt;   /* Byte has value '10101110'B */
```

Furthermore, segments of bit or character strings containing several bits or characters can be addresses analogously; the general form of addressing is:

StringPart::=
Name§String. { **BIT** | **CHAR** | **CHARACTER** } (Expression§Start [: Expression§End])

The expressions Start and End must have integer values. End must be greater than or equal to Start, and both values must be within the declared string length.

The result of such a string segment expression has type BIT(lg) or CHAR(lg), resp., where "lg := End - Start + 1" is holds. For bit strings, the compiler must be able to calculate length "lg", so that further type checks are possible when using the string segment expression.

The following cases are distinguished by the compiler:

1. Name.{ **BIT** | **CHAR** }(Expression)
2. Name.{ **BIT** | **CHAR** }(constant-FIXED-Expression: constant-FIXED-Expression)
3. Name.{ **BIT** | **CHAR** }(Expression1: Expression2 + FIXED-Constant)
4. Name.**CHAR**(Expression§Start: Expression§End)

In case (1), the length of the string segment equals 1, "Expression" denotes the bit number or the character position, respectively.

In case (2), the compiler calculates the values of the two constant expressions and determines the length "lg := Constant§End - Constant§Start + 1".

In case (3), Expression1 and Expression2 must be equal. Thus, the length results from the additive constant: "lg := FIXED-Constant + 1".

In case (4), the compiler cannot calculate the length of the string segment! This is not allowed for bit strings. Case (4) provides a variable character string, i.e., the length is not calculable until run time. Variable character strings can be used anywhere, besides in conjunction with the CAT operator.

Example:

Let a transport vehicle in a warehouse be connected to a digital input via 16 consecutive positions. Let the meaning of the connectors be as follows:

connector 1–8	: X coordinate
connector 9–12	: Y coordinate
connector 13	: Z coordinate
connector 14–16	: further parameters

When positioning has been completed, the actual position shall be retrieved and checked.

```
MODULE;
SYSTEM;
    RFZ : DIGE(1)*0*1, 16;
    ...
PROBLEM;
    SPC RFZ DATION IN BIT(16);
    DCL Rfz DATION IN BIT(16) CREATED(RFZ);
    ...
    Control:TASK;
        DCL Rfz_State BIT(16),
            Xcoord BIT(8), Ycoord BIT(4), Zcoord BIT(1);
        ...
            /* positioning */
        TAKE Rfz_State FROM Rfz;
        Xcoord:=Rfz_State.BIT(1:8);
        Ycoord:=Rfz_State.BIT(9:12);
        Zcoord:=Rfz_State.BIT(13);
            /* checking of the actual position */
        ...
    END; ! Control
    ...
```

This program part can be programmed more flexibly by additionally recording the initial positions of the partial strings Xcoord, Ycoord and Zcoord in variables Init_X, Init_Y, Init_Z.

Table 6.1: Monadic operators for numerical, temporal and bit values

Expression	Operand type	Result type	Meaning of operator
+a	FIXED(g) FLOAT(g) DURATION	FIXED(g) FLOAT(g) DURATION	identity
-a	FIXED(g) FLOAT(g) DURATION	FIXED(g) FLOAT(g) DURATION	changing the sign of a
NOT a	BIT(lg)	BIT(lg)	inverting all bit positions of a
ABS a	FIXED(g) FLOAT(g) DURATION	FIXED(g) FLOAT(g) DURATION	absolute value of a
SIGN a	FIXED(g) FLOAT(g) DURATION	FIXED(1)	determining the sign of a 1 for a > 0 0 for a = 0 -1 for a < 0

```
Control: TASK;
    ...
    DCL (Init_X, Init_Y, Init_Z) INV FIXED INIT(1, 9, 13);
    ...
    Xcoord := Rfz_State.BIT(Init_X:Init_X+7);
    Ycoord := Rfz_State.BIT(Init_Y:Init_Y+3);
    Zcoord := Rfz_State.BIT(Init_Z);
    ...
END; ! Control
```

Assignments to string segments are possible, too; furthermore, they may occur as operands in expressions.

Parts of an expression may be written in brackets to influence the order of the expression calculation (cf. 6.3), e.g.

A - (B + C)

6.1.1 Monadic Operators

Table 6.1 describes for each listed monadic operator

- which type the operand may have
- which type the result (of the operation) has, and
- the meaning of the operator.

In Tables 6.1 to 6.4, “a” stands for any operand, g for the precision, and lg for the length of the operand.

Examples (Table reftab71):

```
DCL (X, Y) FLOAT,
    B      BIT(4);
```

Table 6.2: Monadic operators for explicit type conversions

Expression	**Operand type**	**Result type**	**Meaning of operator**
TOFIXED a	CHARACTER(1)	FIXED(7)	ASCII code for operand character
	BIT(lg)	FIXED(g)	interpreting the bit pattern of the operand as an integer, with g = lg
TOFLOAT a	FIXED(g)	FLOAT(g)	converting the operand into a floating point number
TOBIT a	FIXED(g)	BIT(lg)	interpreting the operand as bit pattern, with lg = g
TOCHAR a	FIXED	CHARACTER(1)	character for the ASCII code of the operand
ENTIER a	FLOAT(g)	FIXED(g)	greatest integer less or equal than the operand
ROUND a	FLOAT(g)	FIXED(g)	next integer according to DIN
CONT a	REF type	type	explicit de-referentiation

Table 6.3: Monadic arithmetical operators

Expression	**Operand type**	**Result type**	**Meaning of operator**
SQRT a	FIXED(g) or FLOAT(g)	FLOAT(g)	square root of operand
SIN a			sine of operand
COS a			cosine of operand
EXP a			e^a, with e=2.718281828459
LN a			natural logarithm of operand
TAN a			tangent of operand
ATAN a			arcus tangent of operand
TANH a			tangent hyperbolicus of operand

```
X := 3;
B := '1001'B
Y := -X;            /* Y has value -3 */
B := NOT B;         /* B has value '0110'B */
```

In general, in an assignment, the type of the variable given to the left of the assignment character must match the type of the value of the assigned expression (cf. section 6.3). Particularly, only values of type integer may be assigned to variables of integers. Furthermore, the types of operands of dyadic operators must be compatible.

Thus, several operators are provided which cause a possibly needed conversion of the type of objects.

Examples (Table 6.2):

```
DCL A FIXED(15),
    X FLOAT(31),
    (B, C) BIT(15);
... A := ENTIER X//2;       /* corresponds to A := (Entier X)//2; */
A := ENTIER (X/2.0);
C := TOBIT A AND B;
A := ROUND X + TOFIXED B;
```

Table 6.4: Other monadic operators

Expression	**Operand type**	**Result type**	**Meaning of operand**
LWB a	array	FIXED(31)	lower boundary of the first dimension of the operand array
UPB a	array	FIXED(31)	upper boundary of the first dimension of the operand array
	CHARACTER(lg)	FIXED(15)	result := Lg
SIZEOF a	identifier	FIXED(31)	memory size of the identified object at run time in bytes
	simple type		
SIZEOF a MAX	REF CHAR( )	FIXED(31)	maximum length of the referenced character string variable
SIZEOF a LENGTH	REF CHAR( )	FIXED(31)	actual length of the referenced character string variable
TRY a	SEMA	BIT(1)	try of a REQUEST, ’1’B if successful

6.1.2 Dyadic Operators

Tables 6.5, 6.6, 6.7 and 6.8 describe for each listed dyadic operator

- which type the operands may have
- which type the result (of the operation) has, and
- the meaning of the operator,

where

- “op1” and “op2”, denote the first and second operand, resp.
- g1, g2, ..., and lg1, lg2, ..., denote the precision and lengths,resp., of the operands and the results.

Since the precision of the result of an addition, subtraction, multiplication or division equals the maximum of the precisions of both operands, it may happen that the overflows arising in these operations are cut off. Hence, the dyadic operator FIT is provided. It causes the conversion from the precision of operand “a” into the precision of operand b. FIT has rank 1.

Example:

```
...
DCL (A, B) FIXED(15),
        C FIXED(31);
DCL D BIT(16);
A:=32767;
B:=4;
C:=(A FIT C)*B; /* C obtains value 131068 */
A:=A FIT C;     /* assignment not permitted */
A:=C FIT A;     /* OK, but lack of information of C by C FIT A */
```

Table 6.5: Dyadic operators for numerical and temporal values

Expression	**Type operand 1**	**Type operand 2**	**Result type**	**Meaning of operator**
op1 + op2	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) DURATION DURATION CLOCK	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) DURATION CLOCK DURATION	FIXED(g3) FLOAT(g3) FLOAT(g3) FLOAT(g3) DURATION CLOCK CLOCK	addition of the values of the operands op1 and op2 g3 = max (g1, g2)
op1 - op2	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) DURATION CLOCK CLOCK	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) DURATION DURATION CLOCK	FIXED(g3) FLOAT(g3) FLOAT(g3) FLOAT(g3) DURATION CLOCK DURATION	subtraction of the values of the operands op1 and op2 g3 = max (g1, g2)
op1 * op2	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) FIXED(g1) DURATION FLOAT(g1) DURATION	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) DURATION FIXED(g2) DURATION FLOAT(g2)	FIXED(g3) FLOAT(g3) FLOAT(g3) FLOAT(g3) DURATION DURATION DURATION DURATION	multiplication of the values of the operands op1 and op2 g3 = max (g1, g2)
op1 / op2	FIXED(g1) FLOAT(g1) FIXED(g1) FLOAT(g1) DURATION DURATION DURATION	FIXED(g2) FIXED(g2) FLOAT(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) DURATION	FLOAT(g3) FLOAT(g3) FLOAT(g3) FLOAT(g3) DURATION DURATION FLOAT(g4)	division of the values of the operands op1 and op2, if op2 $\neq$ 0 g3 = max (g1, g2) g4 = 31 (dependent on implementation)
op1 // op2	FIXED(g1)	FIXED(g2)	FIXED(g3)	integer division of the values of the operands op1 and op2 g3 = max (g1, g2)
op1 REM op2	FIXED(g1)	FIXED(g2)	FIXED(g3)	remainder of the integer division of the values of the operands op1 and op2
op1 ** op2	FIXED(g1) FLOAT(g1)	FIXED(g2) FIXED(g2)	FIXED(g1) FLOAT(g1)	exponentiation of the values of the operands op1 and op2
op1 FIT op2	FIXED(g1) FLOAT(g1)	FIXED(g2) FLOAT(g2)	FIXED(g2) FLOAT(g2)	changing the precision of operand op1 into the precision of operand op2

```
/* conversion of FIXED and BIT objects */
/* without conversion of the internal presentation */
A:=(TOFIXED D) FIT A;
D:=TOBIT (A FIT 1(16));
```

In order to enable the user to reconstruct the result of a comparison operation of two character strings, the comparison algorithm is given here (the shorter character string is filled with blanks on the right side).

Algorithm to compare two character strings, where it holds “lg3 = max(lg1, lg2)”:

```
TYPE StringCheck FIXED; DCL (less, equal, greater) INV FIXED INIT(-1, 0, 1);
    string_comparison: PROC((string1, string2) REF INV CHAR(lg3))RETURNS(StringCheck);
        FOR i TO lg3 REPEAT
            IF TOFIXED string1.CHAR(i) < TOFIXED string2.CHAR(i)
            THEN RETURN(less);
            ELSE IF TOFIXED string1.CHAR(i) > TOFIXED string2.CHAR(i)
                THEN
                RETURN(greater);
                FIN;
            FIN;
        END; ! loop
        RETURN(equal);
    END; ! PROC string_comparison
```

Two further dyadic standard operators relating to array bounds are defined in Table 6.9.

Table 6.6: Dyadic comparison operators

Expression	**Type operand 1**	**Type operand 2**	**Result type**	**Meaning of operator**
op1 < op2 or op1 LT op2	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) CLOCK DURATION	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) CLOCK DURATION	BIT(1)	“less than” If op1 is less than op2, the result has value ’1’B, otherwise ’0’B.
	CHAR(lg1)	CHAR(lg2)		character string comparison (for comparison algorithm, cf. explanations)
op1 > op2 or op1 GT op2	compare op1 < op2	compare op1 < op	BIT(1)	“greater than”: If op1 is greater than op2, the result has value ’1’B, otherwise ’0’B.
op1 <= op2 or op1 LE op2	compare op1 < op2	compare op1 < op2	BIT(1)	“less or equal” If op1 is less or equal op2, the result has value ’1’B, otherwise ’0’B.
op1 >= op2 or op1 GE op2	compare op1 < op2	compare op1 < op2	BIT(1)	“greater or equal” If op1 is greater or equal op2, the result has value ’1’B, otherwise ’0’B.
op1 == op2 or op1 EQ op2	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) CLOCK DURATION CHAR(lg1) BIT(lg1)	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) CLOCK DURATION CHAR(lg2) BIT(lg2)	BIT(1)	“equal” If op1 is equal op2, the result has value ’1’B, otherwise ’0’B. If lg2 $\neq$ lg1, the shorter character or bit string, resp., is padded with blanks or zeros, resp., on the right side to match the length of the longer string.
op1 /= op2 or op1 NE op2	compare op1 < op2	compare op1 < op2	BIT(1)	“not equal” If op1 is not equal op2, the result has value ’1’B, otherwise ’0’B. If lg2 $\neq$ lg1, the shorter character or bit string, resp., is padded with blanks or zeros, resp., on the right side to match the length of the longer string.
op1 IS op2	REF type	REF type	BIT(1)	Comparison for euqality of the values of references variables. The result upon equality is ’1’B.
op1 ISNT op2	REF type	REF type	BIT(1)	Comparison for euqality of the values of references variables. The result upon non-equality is ’1’B.

Table 6.7: Dyadic Boolean and shift operators

Expression	**Type operand 1**	**Type operand 2**	**Result type**	**Meaning of operator**
op1 AND op2	BIT(lg1)	BIT(lg2)	BIT(lg3)	Boolean bitwise conjunction, disjunction, or antivalence, resp., of the operands lg3 = max (lg1, lg2). The shorter operand is padded with zeros on the right side to match the length of the longer operand.
op1 OR op2	BIT(lg1)	BIT(lg2)	BIT(lg3)	
op1 EXOR op2	BIT(lg1)	BIT(lg2)	BIT(lg3)	
op1 <> op2 or op1 CSHIFT op2	BIT(lg)	FIXED(g)	BIT(lg)	Cyclic shift of operand op1 for the number of positions determined by operand op2. If operand op2 > 0, op1 is shifted to the left, otherwise to the right.
op1 SHIFT op2	BIT(lg)	FIXED(g)	BIT(lg)	Shifting of operand op1 for the number of positions determined operand op2. If operand op2 > 0, op1 is shifted to the left, otherwise to the right. In both cases, zeros are padded.

Table 6.8: Dyadic character string operators

Expression	**Type operand 1**	**Type operand 2**	**Result type**	**Meaning of operator**
op1 >< op2 or op1 CAT op2	CHAR(lg1) BIT(lg1)	CHAR(lg2) BIT(lg2)	CHAR(lg3) BIT(lg3)	Concatenation of the strings op1 and op2. Op2 is linked to op1 on the right side. lg3 = lg1 + lg2.

Table 6.9: Other dyadic operators

Expression	**Type operand 1**	**Type operand 2**	**Result type**	**Meaning of operator**
op1 LWB op2	FIXED(g)	array	FIXED(31)	lower boundary of the dimension (given by op1) of the array (determined by op2), if existing
op1 UPB op2	FIXED(g)	array	FIXED(31)	upper boundary of the dimension (given by op1) of the array (determined by op2), if existing

Table 6.10: Ranks of the operators defined in PEARL

rank	dyadic operators	evaluation order
1	**, FIT, LWB, UPB	from the right to the left
2	*, /, ><, //, REM	from the left to the right
3	+, -, <>, SHIFT	from the left to the right
4	<, >, <=, >=	from the left to the right
5	==, /=, IS, ISNT	from the left to the right
6	AND	from the left to the right
7	OR, EXOR	from the left to the right

Example:

```
...
P: PROC(A(,) FIXED IDENT);
    ...
    FOR i FROM LWB A TO UPB A
        REPEAT
            FOR k FROM 2 LWB A TO 2 UPB A
                REPEAT
                    ...
                END; ! for k
        ...
        END; ! for i
    ...
    END; ! proc p
...
DCL Tab1(5,10) FIXED,
    Tab2(-1:2, 3:5) FIXED;
...
CALL P(Tab1) ;
...
CALL P(Tab2);
```

6.1.3 Evaluation of Expressions

In the sequel, lower-case letters a, b, c ... denote constants or scalar variables.

According to the rules of arithmetics, the order of calculating an expression is dependent on the (pre-)rank of the various operators in the expression. The dyadic operator “*”, e.g., has a higher rank than the dyadic operator “+”: In the expression “a+b*c”, hence, “b*c” is calculated first, and the product is added to a.

The ranking of dyadic operators is defined in Table 6.10. Lower numbers correspond to higher ranks.

All monadic standard operators have rank 1.

The order of calculating an expression is furthermore influenced in the common way of putting parentheses around parts of the expression; e.g., in the expression

a*(b-(c-d))

first c-d is calculated, this first intermediate result is subtracted from b, and then this second intermediate

result is multiplied by a.

In general, an expression is calculated according to following rules:

- The partial expression with the highest rank operator is calculated first, unless one of the two following rules is violated.
- If several operators of the same rank occur, the calculation takes place
  - from the left to the right in case 2≥rank≥7
  - from the right to the left in case rank=1
    Example: -a**b corresponds to -(a**b)
- parenthesised partial expressions are completely calculated due to the rules above, before they are combined with another partial expression.

6.2 Operator Definition (OPERATOR)

The operator definition allows to define new operators with freely selectable identifiers, or to extend the meaning of the previously introduced standard operators.

OperatorDefinition ::=
    OPERATOR OpName ([OpParameter ,] OpParameter)
    RETURNS (ResultType);
    ProcedureBody
    END;

OpName ::=
    Identifier | + | − | ∗ | ∗∗ | / | // |==| / =|<=|>=|<|>|<>|><

OpParameter ::=
    Identifier [VirtDimensionList] ParameterType [**IDENTICAL** | **IDENT**]

ResultType ::=
    SimpleType | StructuredType | TypeReference

The meaning is analogous to the one of a function procedure (cf. 8).

Example:

The standard operator + shall be extended for complex numbers.

PROBLEM;

TYPE Complex **STRUCT**
    [Real **FLOAT**, Imag **FLOAT**];

OPERATOR + (A Complex **IDENT**, B Complex **IDENT**) **RETURNS** (Complex):
    DCL Sum Complex;
    Sum.Real := A.Real + B.Real;
    Sum.Imag := A.Imag + B.Imag;
    RETURN (Sum);
    END; ! Operator +

DCL (XX, YY, ZZ) Complex,
(X, Y, Z) **FLOAT**;
...
ZZ := XX + YY;
Z := X + Y;
...

This example shows the possibility to define various operators with the same operator name, if the operands are of different types. If an expression is evaluated, in which such an operator name occurs, that operation is executed where the parameter types are identical with the operand types in the expression.

If an operator declaration serves for extending the meaning of a standard operator, the re-declared operator has the rank of the standard operator. For an operator declared with a new operator name not introduced in a standard way, a rank between 1 and 7 can be determined with the help of a rank declaration:

PrecedenceDefinition ::=
PRECEDENCE OpName ({ 1 | 2 | 3 | 4 | 5 | 6 | 7 });

If a precedence shall be defined for a new operator, this must take place before the operator definition; if no precedence definition is stated, the new operator is given precedence 7.

Example:

... **PRECEDENCE** INDEX(1);
OPERATOR INDEX...;

6.3 Assignments

Assignments are implemented for scalar variables and for structures, but not for arrays.

Assignment ::=
ScalarAssignment | StructureAssignment

6.3.1 Assignments for Scalar Variables

Assignments for scalar variables are defined as follows:

ScalarAssignment ::=
Name§ScalarVariable { := | = } Expression;

The statement causes that the value of the expression on the right side is assigned to the variable given on the left of the assignment sign (":=" or "="), i.e., after the execution of the assignment, this name can be used to refer to the value determined by this expression and, if needed, calculated before the actual assignment:

Result(i) := Koeff * SIN((X(i+1) - X(i)) / X(i));

The type of the variable given to the left of the assignment sign has to match the type of the value of the expression, with the following exceptions:

- The value of a FIXED variable or an integer, resp., may be assigned to a FLOAT variable.
- The precision of a numeric variable to the left of an assignment sign may be greater than the precision

of the value of the expression.

- A bit or character string, resp., to the left may have a greater length than the value to be assigned; if needed, the latter is extended by zeros or spaces, resp., on the right.

Operators for needed type conversions are described in 6.1.2.

Examples:

```
DCL (I, J) FIXED(15),
      K FIXED(31),
      (X, Y) FLOAT,
      Bit8 BIT(8),
      Bit12 BIT(12),
      Text4 CHAR(4),
      Text10 CHAR(10),
      Duration(2) DURATION,
      Time(2) CLOCK;

I := 2.0;                          ! wrong
J := 3;
X := J+5; Y := 0;
K := J;
J := K;                            ! wrong
Text10 := 'Result';                ! Text10 has value 'Result__'
Bit8 := 'A9F'B4;                   ! wrong, because too long
Duration(1) := 1 HRS;
Duration(2) := 30 MIN;
Time(1) := 11:00:00;
Time(2) := Time(1) + (IF   Time(1) < 12:00:00
                           THEN Duration(1)
                           ELSE Duration(2)
                      FIN);
Bit8 := '10001100'B;
Bit12 := Bit8 >< '11'B;            /* Bit12 has value '100011001100'B */
Bit8 := Bit8 CSHIFT 3;             /* Bit8 has value '01100100'B */
Bit12 := Bit8 SHIFT -6;            /* Bit12 has value '000000100011'B */
```

It is possible to declare variables with an attribute for assignment protection (cf. 5.14). Assignments to such variables result in error messages.

6.3.2 Assignments for Structures

The values of all components of a structure can be assigned to another structure in one single assignment:

```
StructureAssignment ::=
      Name§Structure_1 { := | = } ExpressionStructure_2;
```

The values of the components of Structure_2 are assigned to the corresponding components of Structure_1. Both structures must have the same type; i.e., the number of components and their types must match; an implicit type adaptation like for scalar variables takes not place.

Example:

```
TYPE  Type_Measurement STRUCT
     [ Time_Stamp,
       Value FLOAT(53)];

DCL   Workpiece STRUCT
     [ Ident CHAR(8),
       Quality Type_Measurement,
       ... ];

DCL   Measurement Type_Measurement;
...
Workpiece.Quality := Measurement;
```

6.4 Overloading of Data Structures

To exclude programming errors to a large extent, the compiler checks the use of variables and values in accordance with types.

Upon assignments, the types of the left and right side have to match each other (see 6.3: Assignments), and when calling a procedure, the actual parameters have to match the formal parameters in type (see 8.2: Call of Procedures).

However, there are often situations where the required type compatibility ("strong typing") proves to be a hindrance. As an example, we refer to a data base interface: the data base shall store user data of different types with different length. For this, the routines only need the address and the length of the structures to be stored. For strong typing, a new data base interface (with the desired user types) had to be programmed for each application.

PEARL 90 provides two different syntactic forms for the overloading of different data types. Both forms allow exclusively the type conversion of the objects' addresses when assigning or identifying, resp. A data conversion does not take place in any case. For converting basic types, there is a sufficient number of standard operations in PEARL (see 6.1: Monadic and Dyadic Operators).

6.4.1 The "BY TYPE" Operator

With the "BY TYPE" operator, the type of a variable address can be converted to any other pointer type.

TypeConvertingExpression ::=
    Name **BY TYPE** Type

The type of the variable "Name" is converted into the given Type by operator "BY TYPE". The result of the type converting expression is the address of "Name" with type "Type". This type conversion is a compiler internal action preventing an error message of the compiler. At run time, no action is executed; especially the content of "Name" stays unchanged.

Example:

Overloading of data objects upon assignment

```
DCL var        TYPE_A;
DCL ptr REF TYPE_B;
...
```

ptr := var **BY TYPE** TYPE_B;

The address of variable "var" receives type "TYPE_B"; thus, the assignment to pointer "ptr" is correct. It is now possible to access a data object of type "TYPE_B", which overloads variable "var", via pointer "ptr".

Example:

Overloading of data objects upon procedure call

SPC p1 **ENTRY** (**REF** TYPE_B) **GLOBAL**;
SPC p2 **ENTRY** (TYPE_B **IDENT**) **GLOBAL**;
DCL var TYPE_A;
...
CALL p1 (var **BY TYPE** TYPE_B);
CALL p2 (var **BY TYPE** TYPE_B);

The type of the formal parameter "var" is converted to type "TYPE_B" by operator "BY TYPE". In both cases, the address of variable "var" is passed on to an object of type "TYPE_B" to the procedures as pointer. Thus, both calls are correct.

By switching off the type control, the compiler cannot indicate errors any longer. On no account it is allowed to read or write beyond the end of the basic memory space with the help of an overloaded data structure, because severe errors can arise upon program execution (a data loss is just the smallest problem).

The programmer should not make use of any assumptions about data storage within the overloaded memory area, either. Storing data is generally dependent on the target machine. Programs converting data with the help of overloaded data structures are hardware dependent and only portable with great effort. Access to data should always be executed via the same data structure, both for writing and for reading later on. Since in this case, the data are always processed in accordance with their types, such programs are portable, if the space requirements of the overloading data structure do not go beyond the supported filler.

6.4.2 The "VOID" Data Type

In contrast to the active type conversion by the "BY TYPE" operator, the "VOID" data type provides a passive possibility to overload different data objects. Similar to the constant "NIL", which can be assigned to pointer variables of any data types, the compiler accepts the assignment of any data addresses to a pointer variable of type "VOID".

The "VOID" data type is written as structure description without components, and may only be used in combination with REF.

VOID-DataType ::=
    STRUCT [ ]

A reference variable with this type can admit any variable address, the variable, however, cannot be altered directly. First, the address must be assigned to a reference variable with the needed type, before the data object can be manipulated via the reference variable.

Example:

General memory administration routine

DCL c_max_buffer **INV** **FIXED**(31) **INIT**(10000);
DCL memory_block (c_max_buffer) **CHAR**(1);

```
DCL free                    FIXED(31)  INIT(0);
DCL memory_protection       SEMA       PRESET(1);


Malloc: PROC(size FIXED(31))RETURNS(REF STRUCT[ ])GLOBAL;
        DCL ptr REF CHAR(1);

        REQUEST memory_protection;          /* synchronize access to global variables */

        IF free + size >= c_max_buffer THEN
                ptr := NIL;          /* no space left */
        ELSE
                ptr := memory_block(free + 1);
                free := free + size;
        FIN;

        RELEASE memory_protection;
        RETURN(ptr);
END; ! Malloc

t: TASK;
        DCL ptr_var_a REF TYPE_A;
        DCL ptr_var_b REF TYPE_B;

        /* require memory range for different types */
        ptr_var_a := Malloc (SIZEOF TYPE_A);
        ptr_var_b := Malloc (SIZEOF TYPE_B);
END; ! t
```

This second form is particularly suitable for polymorphic procedures, i.e., procedures called with actual parameters of different types. The “VOID” data type serves on the one hand as indication in the procedure specification that the procedure works with different parameter types, on the other hand, an explicit type conversion with the “BY TYPE” operator can be omitted in the call position.

Example (Data base interface):

```
SPC read_record  ENTRY(adr REF         STRUCT[ ], size FIXED(31)) GLOBAL;
SPC write_record ENTRY(adr REF INV STRUCT[ ], size FIXED(31)) GLOBAL;


...
DCL user_data TYPE_A;

/* add record with user data into data base */
CALL write_record (user_data, SIZEOF TYPE_A);

/* read user record from data base */
CALL read_record (user_data, SIZEOF TYPE_A);
```

The data base routines “read_record” and “write_record” are called by various user programs. They can store and read records of different length. For this, the data base routines need only the addresses and lengths of the records, but not the user specific data types.

Chapter 7

Statements for the Control of Sequential Execution

A task or procedure declaration defines a sequence of statements which are processed sequentially in the defined order when executing the task or the procedure, unless control statements designated for this influence the order of processing.

Such control statements are

- the conditional statement
- the statement selection
- the empty statement
- the repetition
- the jump statement
- the exit statement

7.1 Conditional Statement (IF)

With the help of the conditional statement, it is determined depending on the result of an expression with which statement the program execution is to continued.

ConditionalStatement ::=
    IF Expression **THEN** Statement$^{\cdots}$ [**ELSE** Statement$^{\cdots}$] **FIN**;

The result of the expression must be of type BIT(1). If the expression provides value '1'B (true), the statements following THEN are evaluated; otherwise the statements following ELSE are evaluated.

If the execution of the statements following THEN or ELSE, resp., does not result in a jump out of the conditional statement, the statement following FIN is evaluated subsequently.

Example:

IF gradient > degree_bound
    THEN alarm;

```
        ELSE IF gradient > degree_threshold
                THEN ALL 1 SEC ACTIVATE measurement; ! measure more often
            FIN;
FIN;
...
```

7.2 Statement Selection (CASE) and Empty Statement

Let us assume that a (function) procedure control shall be used for controlling several devices of the same kind, after each call returning a number between 1 and 4 meaning:

- returned value = 1: request carried out
- returned value = 2: request data wrong
- returned value = 3: device not addressable
- returned value = 4: device does not work correctly

The task supply shall perform a measure planned for each case.

For the programming of such case distinctions, the statement selection provided in two (historically caused) forms is suitable: The older form allows only integers as distinction criteria; the newer one permits also characters. At first, the older form is described:

```
StatementSelection1 ::=
    CASE Expression
        { ALT Statement··· }···
        [ OUT Statement··· ]
    FIN;
```

The number 1 is assigned to the statement sequence following the first ALT (alternative 1), the number 2 to the statement sequence following the second ALT (alternative 2), etc.

Upon execution of the statement selection, the given expression is evaluated; it must result in a value of type FIXED. If the integer value is between 1 and the number of given alternatives, the associated statement sequence is executed; otherwise the statement sequence following OUT (if given) is executed.

If the selected statement sequence does not contain a jump out of the statement selection, the statement following FIN is evaluated subsequently.

Example:

The above problem can be programmed as follows:

```
supply:  TASK PRIO 7;

control: PROC (No FIXED,          ! device
               order BIT(8))      ! order inf.
         RETURNS (FIXED);
         ! procedure body for carrying out the control order
         END; ! control
         ...
```

```
        ! creating an order for the device with index no

again:  CASE control (no, order)
        ALT   ! order carried out
              ;
        ALT   ! order inf. wrong
              CALL error(2); GOTO end;
        ALT   ! device dead
              ACTIVATE device_breakdown PRIO 2;
              CALL error(3); GOTO end;
        ALT   ! device works incorrectly
              CALL device_check; GOTO again;
        OUT   ! result out of range
              CALL error(5);
        FIN;
        ...
end:    END;  ! supply
```

In this example, the empty statement is used. It consists of one semicolon only and has no effects. The keyword ALT must be followed by one statement at least; this may also be an empty statement. It has no effects and is only of interest in conditional statements and statement selections.

In the example, the empty statement results in the immediate execution of the statement following FIN in the case of success ("request carried out").

The general form of the empty statement reads:

EmptyStatement ::=
    ;

The second form of the statement selection has the following form:

StatementSelection2 ::=
    CASE CaseIndex
    { **ALT** (CaseList) Statement$^{\cdots}$}$^{\cdots}$
    [**OUT** Statement$^{\cdots}$]
    FIN;

CaseIndex ::=
    Expression§WithValueOfType-FIXED-or-CHAR(1)

CaseList ::=
    IndexRange [, Index-Range]$^{\cdots}$

IndexRange ::=
    Constant [: Constant]

All given constants must be of type CaseIndex expression; CHAR(1) or FIXED are allowed.

When executing the StatementSelection2, the CaseIndex is evaluated. If the value is contained in one of the given case lists, the associated statement sequence is executed; otherwise the statement sequence following OUT (if given) is executed.

If the selected statement sequence does not contain a jump out of the statement selection, the statement following FIN is evaluated subsequently.

Examples:

```
DCL   (Operator, chr)CHAR(1), (x, y) FIXED;
...
CASE Operator
      ALT ('+')    x := x + y;
      ALT ('-')    x := x - y;
      ALT ('*')    x := x * y;
      ALT ('/')    CASE y
                   ALT (0) CALL Error;
                   OUT x := x//y;
                   FIN;
FIN;

CASE chr
      ALT ('A':'Z') CALL uppercase;
      ALT ('a':'z') CALL lowercase;
FIN;

CASE chr
      ALT ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u') CALL Vocal(chr);
      ...
FIN;
```

It is not intended to mix both forms of the statement selection. Thus, e.g., the following is not correct:

```
CASE ErrNum
      ALT /* 1 */ CALL ok;
      ALT (0)     CALL nothing_done;
      ALT (-99:-1) CALL ErrorMsg (ErrNum);
      ...
```

The selection must be deterministic.

Jumps into a statement selection are forbidden.

7.3 Repetition (FOR – REPEAT)

Often, a statement sequence must be executed repeatedly, with only one parameter changing. E.g., various devices are to be checked; let num_devices be the number of devices:

```
FOR i FROM 1 BY 1 TO num_devices
REPEAT
     checking of device(i)
END;
```

In general, such “program loops” are constructed like this:

```
repetition ::=
     [ FOR Identifier§ControlVariable ]
     [ FROM Expression§InitialValue ]
```

[**BY** Expression§Increment]
[**TO** Expression§FinalValue]
[**WHILE** Expression§Condition]
REPEAT
[Declaration]$^{\cdots}$ [Statement]$^{\cdots}$
END [Identifier§Loop] ;

The declarations and statements following REPEAT, i.e., the loop body, are run so often as specified by the clauses given in front of them; the statement following END is carried out subsequently. However, it is also possible to leave the loop body prematurely by a jump statement or the exit statement (cf. 7.5). Jumps into the loop body are not permitted.

In the loop body, all statements are permitted; thus, particularly repetitions can be nested:

```
FOR i TO 10
REPEAT
    FOR k TO 10
    REPEAT
        c (i,k) := a (i,k) + b(i,k);
    END;
END;
```

If InitialValue or Increment are missing, they are assumed as 1. If FinalValue is lacking, the loop body can be repeated unrestrictedly.

The ControlVariable may neither be declared nor changed; it has implicitly type FIXED. The values of the expressions for InitialValue, Increment and FinalValue have to be of type FIXED, the value of the expression for the Condition must be of type BIT(1).

The ControlVariable may not be used in the given expressions, except for Expression§Condition, but in the statements to be repeated.

Besides, all rules for blocks are valid for the loop body (cf. 4.4).

The flow chart depicted in Figure 7.1 is an equivalent representation of the statement

```
FOR Indicator§ControlVariable
BY Expression§InitialValue
TO Expression§Increment
WHILE Expression§Condition
REPEAT
    LoopBody
END;
```

7.4 GoTo Statement (GOTO)

GoToStatement ::=
GOTO Identifier§Label;

This statement has the result that program processing is continued at the program position determined by the label identifier. This program position must be a statement and may not be outside the body of the task or procedure executing the GoTo statement.

Example:

```
                ...
measure: read:  READ value FROM device;
                ...
                GOTO read;
```

In general, statements can (several times) be marked with lables; i.e., the label is given immediately before the (possibly already marked) statement, separated by a colon.

7.5 Exit Statement (EXIT)

The exit statement serves to exit blocks and loops deliberately. With EXIT, also blocks and loops nested several times can be exited deliberately, which must have an identifier (the jump target) at the corresponding end.

ExitStatement ::=
    EXIT [Identifier§BlockOrLoop] ;

If the identifier is missing, program processing is continued with the statement following the end of the block or the loop containing the exit statement.

If the identifier is given, program processing is continued with the statement following the end of the indicated block or loop, the exit statement being in an internal block or loop.

The exit statement may not serve for exiting procedures or tasks.

Example:

```
...
BEGIN                                   /* analysis */
    ...
    TO number REPEAT                    /* comparison */
        ...
        IF MeasuredValue < BoundaryValue
        THEN EXIT analysis;
        ELSE ...
        FIN;
        ...
    END comparison;
    ...
END analysis;
RETURN (OK);
    ...
```

The execution of “EXIT analysis;” would immediately be followed by “RETURN (OK);”.

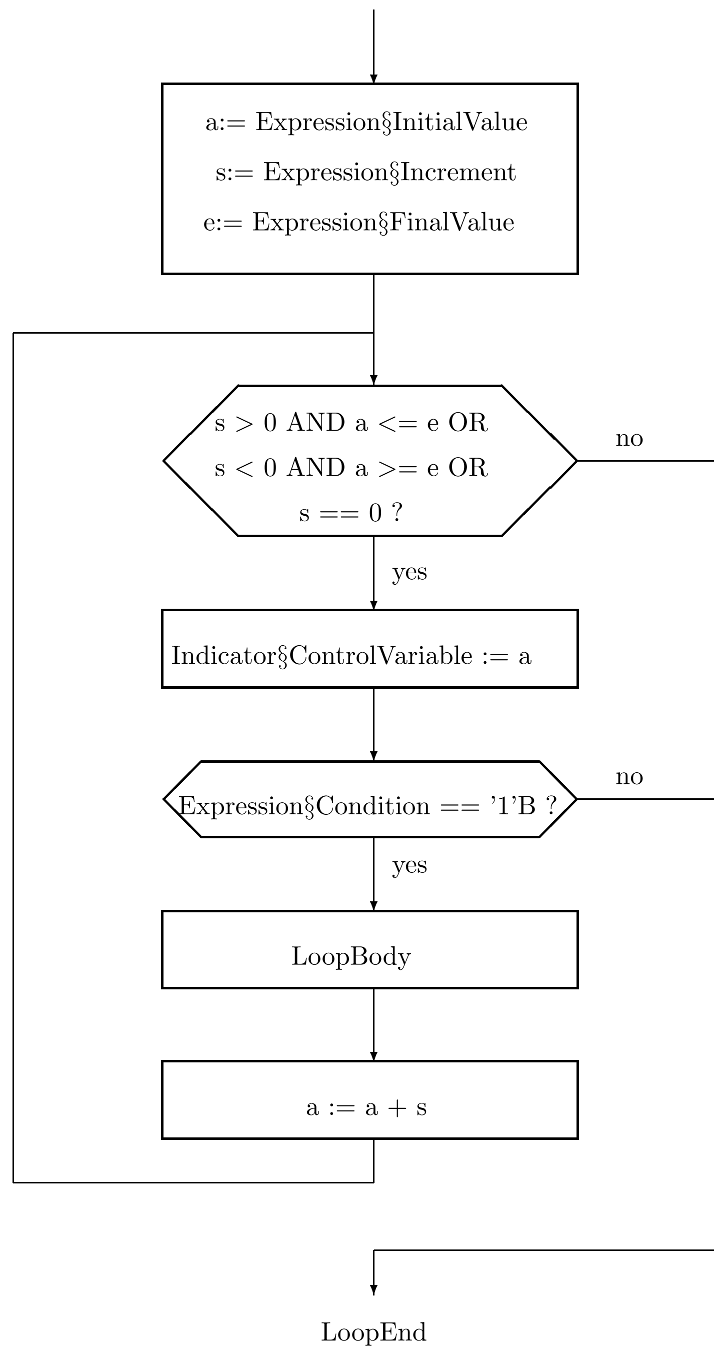

Figure 7.1: Flow chart for evaluating a repetition statement

Chapter 8

Procedures

When solving an automation problem, in the sense of structured programming an independent program part is formulated and named for a logically independent algorithm, particularly if the processing of the algorithm is needed in several parts of the program, eventually only changing the arguments of the algorithm, its parameters. The execution of such a program part is initiated by calling its name — possibly provided with actual parameters.

If this call shall have the same effect as executing the proper program part instead of it, in PEARL this program part is declared and called as a procedure. Otherwise — namely, if the statements following the call are to be executed simultaneously with the called program part — the program part is declared and started as task. Tasks are treated in Chapter 9, Parallel Activities.

Procedures returning a result to their call position are called function procedures, all other ones are called subprogram procedures.

Example for a subprogram procedure:

Let the procedure Output convert a position indication of type FIXED into a bit string BinPos and pass it to a machine to be positioned and marked by number Mach_No of type FIXED. Let Output be called by the task Control.

```
PROBLEM;
    Output: PROC ((Position, Mach_No) FIXED);
        DCL BinPos BIT(8);
        ! transmission of Position into BinPos
        ! output of BinPos to machine Mach_No
        END; ! declaration of Output

    Control: TASK;
        DCL (Pos, /* actual nominal position */
            No /* no of the device */ ) FIXED;
        ...
        /* assignments to Pos and No */
        CALL Output (Pos, No);
        ...
        END; ! declaration of Control
    ...
```

Position and Mach_No are the formal parameters of Output; Pos and No are actual parameters. BinPos is a local variable of Output only known within this procedure.

Example for a function procedure:

Due to a schedule Occ_Plan, procedure Next_Machine shall determine the number of the machine to be occupied next among all available machines. Occ_Plan shall not be passed on as parameter. Let the number to be returned be of type FIXED. Next_Machine shall be declared and called within the task Supply.

```
PROBLEM;
    DCL Occ_Plan ...;

    Supply: TASK;
            DCL Mach_No FIXED;
            ...
            Next_Machine: PROCEDURE RETURNS (FIXED);
                    DCL No FIXED; ! No of the next machine
                        ! establishing of No with the help of Occ_Plan
                    RETURN (No);
                END ! Declaration of Next_Machine
            ...

            Mach_No := Next_Machine;
            ...
        END; ! Declaration of Supply
    ...
```

Since the variable Occ_Plan is declared at module level, it can be used and, if needed, changed by all procedures and tasks of the module.

8.1 Declaration and Specification of Procedures (PROC)

The statement sequence to be executed when calling a procedure is prescribed in a procedure declaration, defining a procedure identifier. The statements of the procedure can use data

- which are declared at module level or in a higher level relativ to the procedure (see 4.1),
- which are specified as formal parameters, i.e., as representatives for those expressions or variables, which, upon calling, are passed to the procedure as actual parameters, or
- which are locally declared in the procedure.

The local declarations and the statements of the procedure form the procedure body.

```
ProcedureDeclaration ::=
    Identifier: { PROCEDURE | PROC } [ ListOfFormalParameters ]
    [ ResultAttribute ]
    [ GlobalAttribute ] ;
    ProcedureBody
    END;

ProcedureBody ::=
    [ Declaration··· ] [ Statement··· ]

ListOfFormalParameters ::=
    (FormalParameter [ , FormalParameter ]···)
```

FormalParameter ::=
    Identifier_or_IdentifierList [VirtualDimensionList] [AssignmentProtection]
    ParameterType [**IDENTICAL** | **IDENT**]

VirtualDimensionList ::=
    ([, ···])

ParameterType ::=
    SimpleType | TypeReference | TypeStructure
    | Identifier§ForType | TypeDation | TypeRealTimeObject

TypeRealTimeObject ::=
    SEMA | **BOLT** | **IRPT** | **INTERRUPT** | **SIGNAL**

ResultAttribute ::=
    RETURNS(ResultType)

ResultType ::=
    SimpleType | TypeReference | TypeStructure | Identifier§ForType

The general form of the specification of a procedure reads as follows:

ProcedureSpecification ::=
    { **SPECIFY** | **SPC** } Identifier§Procedure { **ENTRY** | [**:**] **PROC** }
    [ListOfParametersFor-SPC] [ResultAttribute] GlobalAttribute ;

ListOfParametersFor-SPC ::=
    (ParameterSpecification [, ParameterSpecification]···)

ParameterSpecification ::=
    [Identifier] [VirtualDimensionList] [AssignmentProtection]
    ParameterType [**IDENTICAL** | **IDENT**]

In the procedure specification, the (optionally) definable list of parameters is only of documentary importance; however, it is thus possible to copy the head of a procedure declaration into another module and to generate a correct specification of the procedure by adding the keyword SPECIFY.

Subprogram procedures are declared without and function procedures with result attribute. The result type determines the type of the calculated result which is returned to the call position. With the help of the return statement, this return takes place in the form

RETURN (Expression);

Thus, the value of the expression must have the type specified by the result attribute.

Execution of the procedure body of a function procedure is terminated by executing a return statement. Function procedures may only be terminated this way and no other.

Execution of a subprogram procedure is terminated by

- executing the return statement in the form
  RETURN;

- executing the last statement of the procedure body.

The procedure body can contain declarations, e.g., declarations of local variables which are only known within the procedure body. However, further procedures, so-called nested procedures, may also be declared; the occurring aspects of unambiguity of names, also occurring when declaring procedures in task bodies, are described in 4.3 within the context of blocks.

Due to the call, variables or expressions are associated with the specified formal parameters of the procedure as actual parameters. How this association takes place (two possibilities), is determined by the fact whether the attribute IDENTICAL is given or not. Both ways are explained in 8.2, Call of Procedures.

The number n of commas in the virtual dimension list indicates that the parameter is an (n+1) dimensional array. Formal array parameters (virtual dimension list is present) may only be specified together with the IDENTICAL attribute. If, e.g., the one-dimensional array “A(10) FIXED” shall be passed on to a procedure P with the corresponding formal parameter Array, Array is to be specified like this: “Array() FIXED IDENTICAL”.

Procedures declared at module level, are translated by the compiler with the *re-entrancy* capability, so that they can be used simultaneously by several tasks (see 9). The recursive call of procedures is allowed for all — even for nested — procedures. However, since for each task only limited memory space for the local data of the called procedures (stack) is provided, the programmer should avoid (or suitably restrict) recursion in the sense of safe programs.

8.2 Calls of Procedures (CALL)

Subprogram procedures are called with the help of the keyword CALL or only with their identifiers:

CallStatement ::=
    [**CALL**] Name§SubprogramProcedure [ListOfActualParameters] ;

ListOfActualParameters ::=
    (Expression [, Expression]$^{\cdots}$)

Example:

SPC Output **PROC** (P **FIXED**, N **FIXED**) **GLOBAL**;
DCL (Pos, No) **FIXED**;
...
! Assignments to Pos and No
CALL Output (Pos, No);

The call statement results in associating the given actual parameters to the formal parameters of the indicated procedure in the order of writing down them, and then executing the procedure body. Subsequently, the statement following the call statement is executed.

The call of a function procedure does not take place as an independent statement, but within expressions upon stating the identifier and the actual parameters:

FunctionCall ::=
    Name§FunctionProcedure [ListOfActualParameters]

Example:

The function procedure Ari shall calculate the arithmetic average of an array of n FLOAT variables. This average shall then be printed together with the text “Arith.Average”.

```
Ari: PROC (Array() FLOAT IDENTICAL) RETURNS (FLOAT);
            DCL Sum FLOAT;
            DCL (LowerBound, UpperBound) FIXED;
            Sum := 0;
            LowerBound := LWB Array;
            UpperBound := UPB Array;
            FOR i FROM LowerBound BY 1 TO UpperBound
            REPEAT
                Sum := Sum + Array(i);
            END; ! loop
            RETURN (Sum/(UpperBound - LowerBound + 1));
     END; ! Ari

     DCL MeasuredValue(10) FLOAT;
     ...          /* Acquisition of the measured values */
     PUT Ari (MeasuredValue), 'Arith.Average' TO Printer BY LIST;
     ...
```

When evaluating a function call, the given actual parameters are associated with the formal parameters of the indicated function procedure in the order of writing down them; then the procedure body is executed. Subsequently, the evaluation of the expression where the function call took place is continued — in the above example the evaluation of the expression 'Arith.Average' in the put statement.

Both in the call statement and in the function call, the types of the actual parameters must match the types of the formal parameters corresponding to them.

The association of the actual parameters with the formal parameters can take place in two ways: If the specification of a formal parameter has the attribute IDENTICAL or IDENT, the association takes places with the help of identification, otherwise by value transmission.

In the case of value transmission (also called *call by value*), a new object, having the type of the formal parameter and being local to the procedure body, is declared for each defined formal parameter when invoking the procedure; i.e., the formal parameters become local variables of specified types. Then, the values of the actual parameters are assigned to the corresponding formal parameters. An assignment to a formal parameter by a statement in the procedure body, hence, does not result in a change of the actual parameter. Furthermore, in this case any expressions may be passed as actual parameters.

When associating with the help of identification (also called *call by reference*), a formal parameter is identified with the corresponding actual parameter; i.e., in the procedure body, the data of the actual parameter are referred to under the name of the formal parameter. An assignment to a formal parameter in the procedure body means, thus, an assignment to the variable passed as corresponding actual parameter. Hence, in this case, not expressions, but only names (of variables) may be passed as actual parameters.

Example:

```
PROBLEM;

P1: PROC (pi FIXED, pj FLOAT IDENT);
    ...
    pi := 3; pj := 5.0;
    END; ! P1
P2: PROC ...;
    DCL (i, j) FIXED, a(100) FLOAT;
    ...
    i := 2 ; a(i) := 2.5;
    CALL P1 (i, a(i));
```

...
END; ! P2
...

After the call of P1 in P2 i (still) has the value 2, but a(i) has the value 5.0.

As the language form of the procedure declaration already shows (see 8.1), the values of the actual parameters may be of type

- Integer or FloatingPointNumber, or
- BitString or CharacterString, or
- Time or Duration, or
- Structure or Identifier§For_Type, or
- TypeReference.

No explicit values are assigned to objects of types

- DATION, SEMA, BOLT, INTERRUPT and SIGNAL.

Such objects may only be passed to a procedure via identification, i.e., the formal parameter may only be defined with the IDENTICAL attribute.

8.3 References to Procedures (REF PROC)

The opportunity to use procedure reference variables is a first step towards object oriented programming. With the help of it, e.g., data structures and the necessary procedures for the controlled manipulation of these structures can be combined into new, abstract data structures.

A declaration of reference variables for procedures contains the description of all parameter types, as well as the type of the result.

ProcedureReferenceDeclaration ::=
    { **DECLARE** | **DCL** } Identifier_or_IdentifierList [DimensionAttribute] [**INV**]
    REF TypeProcedure [GlobalAttribute] [InitialisationAttribute] ;

TypeProcedure ::=
    PROC [ListOfParametersFor-SPC] [ResultAttribute]

The general form of the specification of procedure reference variables reads:

ProcedureReferenceSpecification ::=
    { **SPECIFY** | **SPC** } Identifier_or_IdentifierList [VirtualDimensionList] [**INV**]
    REF TypeProcedure GlobalAttribute;

The value assignment to a procedure reference variable takes place with the assignment:

Assignment ::=
    Name§RefProcVariable { := | = } Identifier§Procedure;

Only procedures declared at module level may be assigned here.

The required matching of types means in this case, that the number of parameters, all parameter types and also the type of the result attributes match.

Calling a procedure via a procedure reference variable takes simply place by giving the reference variable, followed by a list of the actual parameters. For procedures without parameters, CALL, or in the case of function procedures, also CONT, can be used.

Examples:

1. **DCL** ProcPointer **REF PROC** (a **FIXED**, b **FIXED**, c **FIXED IDENT**);
   add: **PROC** (a **FIXED**, b **FIXED**, c **FIXED IDENT**);
       c := a + b;
       END;
       DCL (A, B, C) **FIXED**;
   ProcPointer := add;
   ProcPointer (A, B, C);

2. **DCL** FuncPointer **REF PROC RETURNS(CLOCK)**;
   time: **PROC RETURNS(CLOCK)**;
       RETURN(NOW);
       END;

   DCL(A, B) **CLOCK**;
   FuncPointer := time;
   A := FuncPointer;
   B := **CONT** FuncPointer;

Chapter 9

Parallel Activities

Typical for a program to control a technical process are

- asynchronous processes as program parts which run parallel in time and independent from one another, and which are initiated by spontaneous events or at certain (scheduled) times, as well as
- the synchronisation of such processes at certain program points, e.g., to be able to exchange data with one another.

To program such processes, in PEARL tasks, interrupts and synchronisation objects are used.

A task is of a program part which is executed under the control of the operating system. The task body, like a procedure body consists of PEARL declarations and statements. Before using it, a task must be declared; upon this, an identifier is assigned to it, which is then mentioned in tasking operations such as start or resumption.

Since usually only one processor is available to the tasks of a program, they must compete for its use — but also for the access to other resources (such as I/O devices) which have to be shared. The operating system, assigns the resources taking account of the tasks' priority. Thus, a positive integer can be allocated to a task as priority, smaller numbers meaning higher priority. The thus determined task priorities are used by the operating system to control the allocation of resources. If a task, e.g., possesses the only processor and requests another, exclusively occupied resource, the operating system withdraws the processor from this task and allocates it to the task of highest priority among all runable tasks waiting for the processor. To runnable tasks of the same priority, the processor is allocated due to the Round-Robin-Strategy.

Such a priority controlled re-allocation of a processor takes place each time, when one of its operating system functions is called, e.g., upon the occurrence of an interrupt or upon executing statements for task control, for synchronisation and for input and output.

9.1 Declaration and Specification of Tasks (TASK)

The declaration of tasks takes place analogously to the declaration of procedures. In contrast to procedures, tasks may only be declared at module level — i.e., not within procedure or task bodies. Furthermore, parameters are not permitted. However, since in a task body all PEARL objects declared at module level may be used and — as far as it is possible — changed, the data exchange with a task can take place via data which are declared at module level. Particularly the access of several tasks to the same data, however,

should carefully be synchronised with the means described in 9.3. In the case of the data exchange by I/O statements, the input/output functions already take over the synchronisation.

Example:

A task Protocol deposits protocol texts in the variable Text to be delivered to a terminal via the task Output. (The needed synchronisation statements are explained in 9.3.)

```
PROBLEM;
    DCL Text CHAR(60);
    Protocol: TASK;
              Text := ProtocolText(27);
              ...
              END; ! Protocol

    Output:   TASK;
              PUT Text TO Printer BY LIST;
              ...
              END; ! Output
    ...
```

The general form of a task declaration reads:

TaskDeclaration ::=
    Identifier: **TASK** [PriorityAttribute] [**MAIN**] [GlobalAttribute] ;
    TaskBody
    END;

PriorityAttribute ::=
    { **PRIORITY** | **PRIO** } IntegerWithoutPrecision§GreaterZero

TaskBody ::=
    [Declaration···] [Statement···]

If no priority is given in a task declaration, priority 255 is assumed.

The general form of a task specification reads:

TaskSpecification ::=
    { **SPECIFY** | **SPC** } Identifier§Task **TASK** GlobalAttribute;

Upon program or system start, the tasks marked with MAIN are started according to their priority. All MAIN tasks must be declared in the same module.

The global attribute is explained in 4.4.

9.1.1 References to Tasks (REF TASK)

The control of all task activities is also possible via references to objects of type TASK. The declaration of corresponding reference variables can take place like this:

TaskReferenceDeclaration ::=
    { **DECLARE** | **DCL** } Identifier_or_IdentifierList [DimensionAttribute]
    [AllocationProtection] **REF TASK** [GlobalAttribute] [InitialisationAttribute];

The general form of the specification of task reference variables reads:

TaskReferenceSpecification ::=
  { **SPECIFY** | **SPC** } Identifier_or_IdentifierList [VirtualDimensionList]
  [AllocationProtection] **REF TASK** GlobalAttribute;

The following short example shows an application possibility for task reference variables:

Example:

Both segments from two PEARL programs show how various tasks can be started from a procedure, once via their names, once via references.

```
PROBLEM;
    SPC (Consumer_1, Consumer_2, Consumer_3) TASK GLOBAL;
    ...
    StartTask:  PROC (Index FIXED);
                CASE Index
                ALT   ACTIVATE Consumer_1;
                ALT   ACTIVATE Consumer_2;
                ALT   ACTIVATE Consumer_3;
                FIN;
        END; ! StartTask
    ...
```

```
PROBLEM;
    SPC (Consumer_1, Consumer_2, Consumer_3) TASK GLOBAL;
    DCL Consumer (3) INV REF TASK GLOBAL
                     INIT (Consumer_1, Consumer_2, Consumer_3);
    ...
    StartTask:  PROC (Index FIXED);
                IF Index < UPB Consumer THEN
                    ACTIVATE Consumer (Index);
                FIN;
        END; ! StartTask
    ...
```

9.1.2 Determining Task Addresses

The address of a task can be obtained by calling the predefined function TASK or simply by assigning a task identifier to a task reference variable.

Example:

```
SPC TemperatureMeasurement TASK GLOBAL;
DCL PtrTask REF TASK;
...
PtrTask := TASK (TemperatureMeasurement);
PtrTask := TemperatureMeasurement;
```

Both assignments are equivalent, however, the first form should be used, because it documents the address assignment in a better way.

The address of the running task is received by invoking the function TASK without stating a parameter.

Example:

...
PtrTask := **TASK**; ! provides address of the task containing this statement.
...

Besides using task reference variables in task statements, they can also be used for identifying tasks (cf. IS and ISNT operators).

Example:

IF PtrTask **IS TASK**(hello) **THEN**
...
FIN;
IF PtrTask **IS TASK THEN**
/* points to running task */
...
FIN;

9.1.3 Determining Task Priorities

The (current) priority of a task can be obtained by the predefined function PRIO. Without stating a parameter, it provides the priority of the running task, and with a task name as parameter, it returns the priority of that task.

Example:

DCL CurrPrio **FIXED**(15);
DCL CurrTaskA **FIXED**(15);
SPC TaskA **TASK GLOBAL**;
...
CurrPrio := **PRIO**;
PrioTaskA := **PRIO** (TaskA);
CONTINUE TaskA **PRIO** CurrPrio + 1;

9.2 Statements for Controlling Tasks

A task can be started, terminated, suspended, continued, resumed and descheduled.

9.2.1 Start Condition

Unlike procedures, tasks are not executed immediately by calls, but in dependence of time instants and interrupts. Thus, a task must first be **scheduled** for the start by a task control statement of the syntax

Task_Start ::=
[StartCondition] **ACTIVATE** Name§Task;

The start itself is managed by a real time operating system. If the StartCondition defined in the control statement is fulfilled, the task first becomes "runnable"; but it is not started, i.e., transferred into the state

"running", until it has become the runnable task of highest priority (cf. also 9.2.2).

Examples:

AT 20:0:0 **ACTIVATE** Statistics;
  means that the task "statistics" is not to be started until the future time "8 o'clock p.m.".

ACTIVATE statistics;
  in contrast, means that the task "statistics" is to be started immediately.

With the start condition, tasks can be scheduled for cyclic (repeated) start, in accordance with the syntax definition

StartCondition ::=
  AT Expression§Time [Frequency]
  | **AFTER** Expression§Duration [Frequency]
  | **WHEN** Name§Interrupt [**AFTER** Expression§Duration] [Frequency]
  | Frequency

Frequency ::=
  ALL Expression§Duration [{ **UNTIL** Expression§Time }
  | { **DURING** Expression§Duration }]

Examples:

ALL T **ACTIVATE** Regulator;
  means that task Regulator becomes runnable at each integer multiple of the time interval T.

WHEN Alarm **ACTIVATE** ShutDown;
  means that the task ShutDown becomes runnable each time when the interrupt Alarm occurs.

AT Expression§Time determines, at which time the task shall be executed for the first time, AFTER Expression§Duration determines, from when on this is to happen relative to the execution of the task control statement, and WHEN Name§Interrupt determines that this is to happen when the denoted interrupt occurs, possibly delayed by the duration denoted in AFTER Expression§Duration. If AT, AFTER and WHEN are lacking, the task becomes runnable immediately after the execution of the task control statement.

If the task is to be repeated after equal time intervals, the time between two executions is to be determined by ALL Expression§Duration. To limit the repeated executions, a time can be determined by UNTIL Expression§Time, or a duration with DURING Expression§Duration, respectively, after which no more task starts are repeated.

If the start condition commences with WHEN, the task becomes runnable each time the denoted interrupt occurs, considering further components of the start condition. A schedule determined by ALL becomes re-effective, relative to the occurrence of the interrupt, whereas the previous schedule becomes ineffective.

The start condition becomes ineffective (i.e., the task associated with it is not executed any longer),

- if it commences with AFTER or ALL and an end condition determined by UNTIL or DURING is reached, or if it has the form AFTER Expression§Duration and the denoted Duration reckoning from the execution of the task control statement has passed,
- because a statement to deschedule a task is executed (see 9.2.7),
- upon execution of a new task control statement of the same type. The previous start condition is replaced by the new one, or, if no new condition is denoted, deleted.

Example:

...
AT 12:0:0 **ACTIVATE** Protocol;
ALL 2 HRS **ACTIVATE** Protocol;
...

These two schedulings cannot be valid at the same time. On the contrary, start condition ALL 2 HRS replaces start condition AT 12:0:0, unless time "12 o'clock" does not occur before executing the second statement.

9.2.2 Starting a Task (ACTIVATE)

The general form of the statement to start a task (i.e., to schedule a task to be started) reads:

TaskStart ::=
    [StartCondition] **ACTIVATE** Name§Task [Priority];

Priority ::=
    { **PRIORITY** | **PRIO** } Expression§WithPositiveIntegerAsValue

Executing such a start statement causes that the indicated task applies for the allocation of a processor immediately (form without start condition) or at the instant determined by the start condition — competing with all other tasks being runable at the moment of the start (being runable) as well. Thus, upon an immediate start, the started task competes particulary with the starting task, if only one processor is provided.

Possibly, the indicated task has already been started and not been terminated when executing the start statement. If in this case no StartCondition is denoted, an error message occurs. However, if the start statement contains a StartCondition, the indicated task is continued, and its re-start is scheduled (buffered) according to the StartCondition, replacing a possibly existing scheduling by the new one.

A possibly stated priority overwrites the priority stated in the declaration of the indicated task.

A task is terminated

- when it reaches the terminating END statement of its body, or
- by executing a statement to terminate tasks related to the task (see 9.2.3).

Example:

The task Pressure_measurement shall measure all 5 seconds the pressure in a tank and transfer it to the task Checking. If the pressure increases significantly fast, the measurement shall by taken all second with increased priority. The task Checking is started immediately by task Initial.

PROBLEM;
    Initial: **TASK MAIN**;
                    ACTIVATE Checking;
                        /* further initialisations */

                    END; ! Initial

```
Checking:              TASK PRIORITY 6;
                       ALL 5 SEC ACTIVATE Pressure_measurement;
                            /* taking over the measured values
                            if the pressure rises: */
                       ALL 1 SEC ACTIVATE Pressure_measurement PRIO 2;
                       ...
                       END; ! Checking

Pressure_measurement:  TASK PRIO 5;
                       /* measuring and passing over to checking */
                       END; ! Pressure_measurement
...
```

9.2.3 Terminating a Task (TERMINATE)

The premature termination of a task is achieved by following statement:

TaskTermination ::=
  TERMINATE [Name§Task] ;

If the denotation Name§Task is lacking, the statement refers to the task in whose body it is contained.

All resources occupied by the terminated task (including processor) are withdrawn from it. Synchronisation variables blocked by the task, however, are not released automatically.

9.2.4 Suspending a Task (SUSPEND)

By executing the statement

TaskSuspension ::=
  SUSPEND [Name§Task] ;

the indicated task — or the executing task, respectively, if Name§Task is lacking — is suspended, i.e., its execution is postponed. The processor allocated to it is withdrawn from it — but not all the other resources occupied by it.

A suspended task can only be continued by executing a continue statement (see 9.2.5).

9.2.5 Continuing a Task (CONTINUE)

A suspended task can be continued immediately, at a certain point in time, after a certain duration or after the occurrence of an interrupt by the following statement:

TaskContinuation ::=
  [SimpleStartCondition] **CONTINUE** [Name§Task] [Priority]

SimpleStartCondition ::=
  AT Expression§Time | **AFTER** Expression§Duration | **WHEN** Name§Interrupt

If Name§Task is stated, the statement causes that the indicated task competes for the processor immediately (form without start condition) or at the instant determined by the start condition — possibly with the given

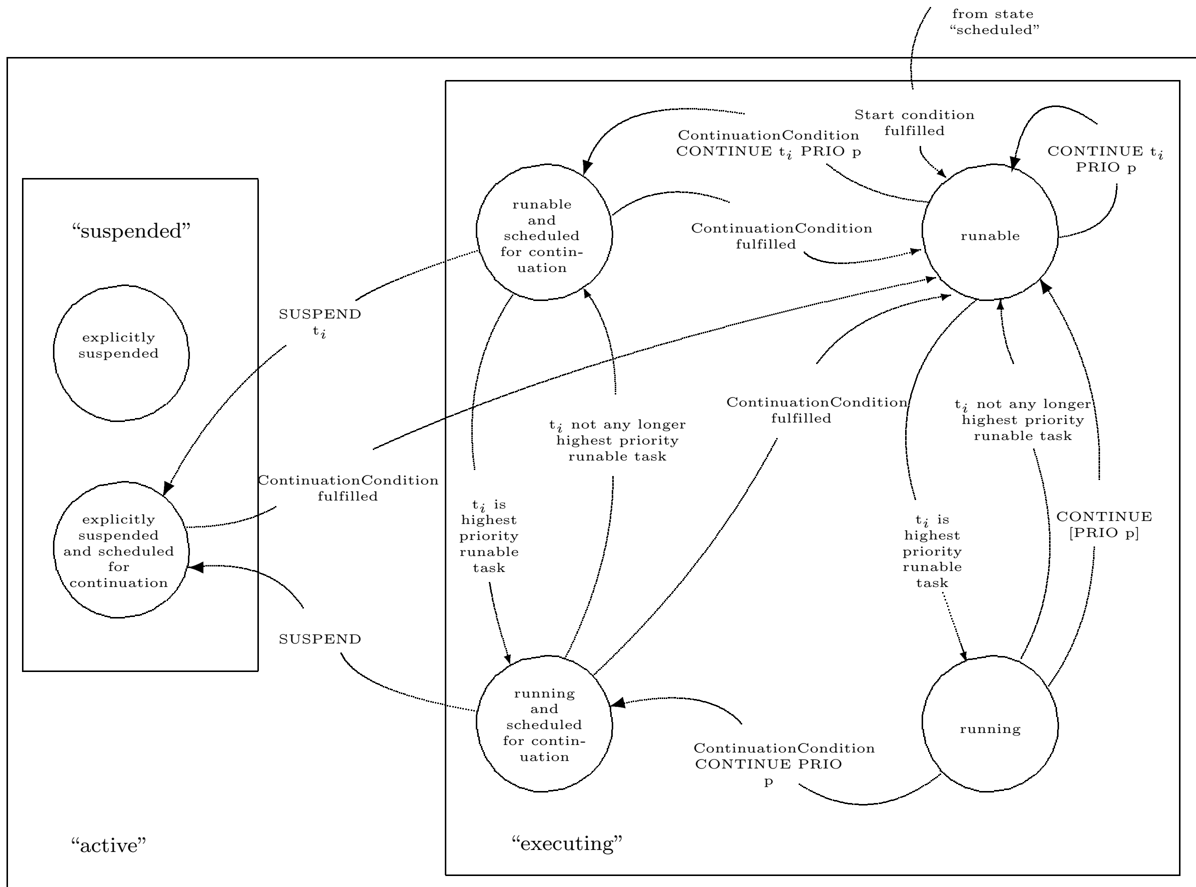

Figure 9.1: State transition diagram of task t_i when executing CONTINUE

priority replacing the declared one.

The form without Name§Task causes that the executing task re-competes for the processor at the instant determined by the start condition — possibly with the given priority replacing the declared one.

Example:

The task Acquisition shall cause an output and not continue until the occurrence of the interrupt Continuation, but then with higher priority.

```
Acquisition: TASK PRIO 8;
                 ! Acquisition of data
             WHEN Continuation CONTINUE PRIO 5;
                 ! Output
             SUSPEND;
                 ! Acquisition of data
         END; ! Acquisition
```

In Figure 9.1, the state transitions of a task possible by executing CONTINUE instructions are illustrated graphically. For the case of “scheduled continuations”, the state transitions for SUSPEND statements are given additionally.

9.2.6 Delaying a Task (RESUME)

If the running task shall release the processor allocated to it for a certain duration or until the occurrence of a certain point in time or an interrupt, respectively, (if it is to be delayed), the following statement is to be executed:

TaskDelay ::=
    SimpleStartCondition **RESUME**;

This statement if equivalent to the non-devisable combination of the statements

SimpleStartCondition **CONTINUE**;
SUSPEND;

After having executed the statement TaskDelay, the start condition is ineffective, i.e., the execution is once.

Example:

The task Control shall turn on a device and check 10 seconds later, whether the device works as desired.

```
Control: TASK;
                ! turning on the device
            AFTER 10 SEC RESUME;
                ! ckecking the function of the device
            ...
        END; ! Control
```

9.2.7 De-scheduling a Task (PREVENT)

Sometimes it is necessary to cancel the schedules existing for a task, i.e., to take care that the start conditions related to this task become ineffective. This can be achieved with the following statement:

TaskPrevent ::=
    PREVENT [Name§Task] ;

This statement does not terminate the corresponding task; if Name§Task is not given, the statement effects the running task.

Example:

The procedure Control called by a higher level task, gives a move request to an AGV (Automatic Guided Vehicle) which has to send the ready message Ready within 30 seconds in order to start the task Supply. If the ready message has not arrived within 30 seconds, the task Malfunction shall be started, and the probably still possible, but delayed start of Supply shall be de-scheduled. In the normal case — Ready arrives within 30 seconds — the already scheduled start of Malfunction must be de-scheduled.

```
PROBLEM:
    SPECIFY  Ready INTERRUPT;

    Control:    PROC (X FIXED, /* X coordinate */
                      Y FIXED); /* Y coordinate */
                /* converting the coordinates taken over into a bit string,
                output of the bit string to the AGV */
```

```
                WHEN Ready ACTIVATE Supply;
                AFTER 30 SEC ACTIVATE Malfunction;
                ...
           END; ! Control

Supply:         TASK PRIO 3;
                PREVENT Malfunction;
                ...
           END; ! Supply

Malfunction:    TASK PRIO 2;
                PREVENT Supply;
                ...
           END; ! Malfunction
```

9.3 Synchronising Tasks

Generally, tasks are run independently from one another. However, it may happen that several tasks work on parts of a more complex, overall problem and have to use certain resources, particularly data, jointly.

Example:

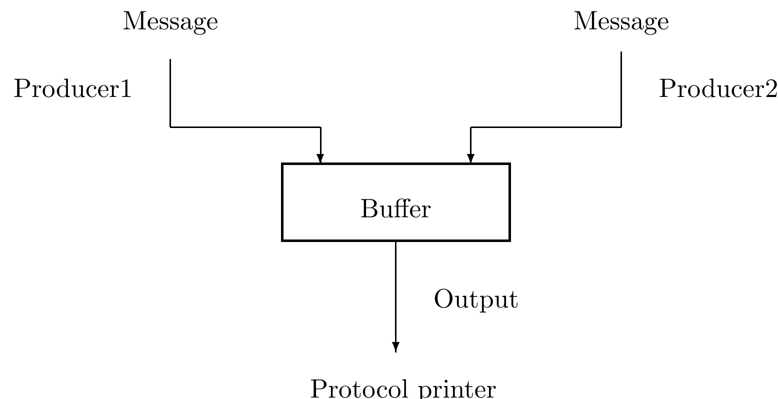

Figure 9.2: Example for task co-ordination

According to Figure 9.2 the tasks Producer1 and Producer2 produce messages which they want to store in a common buffer; by the task output, a message is taken from the buffer and displayed in a formatted way on a protocol printer.

For a correct execution of the operations, they have to be co-ordinated as follows:

- The task Output creates only an output, if a message was buffered, which means that it possibly waits that the buffer is filled by Producer1 or Producer2.
- Producer1 and Producer2 can only buffer a message, if no previous message is being sent out by Output, i.e., they must possibly be suspended until Output has issued the previous message completely.
- The tasks Producer1 and Producer2 to buffer a message have to be mutually exclusive.

To control such co-ordination tasks, two types of synchronisation variables are provided, namely semaphore and bolt variables. If the access to jointly used resources (e.g., data, procedures, devices) is to be co-ordinated, one synchronisation variable is associated to each resource; then the using tasks execute a locking statement on the associated synchronisation variable before and a release statement after using a resource.

The association of synchronisation variables to resources does not take place by PEARL statements; it is only assumed and is realised by using a certain synchronisation variable always in combination with a certain resource, only. Such an association is not restricted to data, procedures, devices, etc., sometimes it must also be done for program parts (e.g., for executing sub-tasks), which shall all be terminated until a certain other program part (e.g., the main task) is to be continued.

9.3.1 Semaphore Variables (SEMA) and Statements (REQUEST, RELEASE, TRY)

Semaphore variables can have non-negative integers as values. These numbers represent the states “free” or “locked”: Zero means state “locked” (the semaphore variable “locks”), positive numbers mean the state “free”. These states can only be changed by special request or release statements.

Semaphore variables are declared as follows:

DCL Identifier_or_IdentifierList **SEMA**;

Example:

DCL (On, Off) **SEMA**;

After its declaration, a semaphore variable has the state “locked”.

If a semaphore variable explicitly is to assume the state “locked”, the following locking statement is to be executed:

REQUEST Name§Sema;

The effect of this statement depends on the actual value of the indicated semaphore variable:

- If the value is greater than zero, it is decremented by 1.
- If the value equals zero, it stays the same; the executing task, however, is suspended and placed in a queue which is (implicitly) associated to the semaphore variable (the task is blocked).

The execution of the release statement

RELEASE Name§Sema;

causes that the value of the indicated semaphore variable is incremented by 1. Furthermore, the tasks in the queue of the semaphore variable are released; they repeat their request statement in the order of their priorities.

Example:

With these means, the problem of buffering messages can be solved as follows:

For entering messages into the buffer, a semaphore variable Into_buffer, for the output out of the buffer, a semaphore variable Out_of_buffer is declared. Upon declaration, Into_buffer and Out_of_buffer are implicitly initialised with “locked”. The task Consumer is started from outside. Before it starts the tasks Producer_1 and Producer_2, Into_buffer must receive the state “free”.

PROBLEM;
    DCL (Into_buffer, Out_of_buffer) **SEMA**;

```
Consumer:   TASK;
                RELEASE Into_buffer;
                ACTIVATE Producer_1;
                ACTIVATE Producer_2;
                REPEAT
                    REQUEST Out_of_buffer;
                        ! Output to the protocol printer
                    RELEASE Into_buffer;
                END; ! Loop
            END; ! Consumer

Producer_1: TASK;
                REPEAT
                        ! Preparing the message
                    REQUEST Into_buffer;
                        ! Buffering
                    RELEASE Out_of_buffer;
                END; ! Loop
            END; ! Producer_1

Producer_2: TASK;
                REPEAT
                        ! Preparing the message
                    REQUEST Into_buffer;
                        ! Buffering
                    RELEASE Out_of_buffer;
                END; ! Loop
            END; ! Producer_2
```

Explanation:
Output waits first for the release of Out_of_buffer, i.e., the buffering of a message, because Out_of_buffer can only be released by Producer_1 or Producer_2. After the output has taken place, Consumer releases Into_buffer, i.e., admits buffering, and waits for a new request. Producer_1 and Producer_2 can only buffer a message, if state “free” is defined for Into_buffer; after buffering a message, they release Out_of_buffer resulting in the continuation of Consumer.

The state of a semaphore variable can be obtained with the monadic operator

TRY Name§Sema;

As result, the operator provides a BIT(1) value to be used in all logical expressions. If the semaphore variable is “free”, the operator executes a REQUEST statement and provides value '1'B as result. If the semaphore variable has already state “locked”, no REQUEST statement is executed, and value '0'B is provided. Therefores a possible blocking of the running task can be avoided when using the TRY operator!

Example:

In a dialogue system the function Function1 may not be called twice at the same time, although the blocking of a task calling Function1 has to be avoided.

```
PROBLEM;
    DCL Func_1 SEMA;

    Dialogue: TASK;
                  ...
                  IF TRY Func1 THEN
```

```
            ! Semaphore variable Func1 had state "free"
            CALL Function1;          ! execute function
            RELEASE Func1;           ! release semaphore
        ELSE
            ! Semaphore variable func_1 had state "locked"
            PUT 'Function 1 not possible at present' TO Terminal;
        FIN;
        ...
    END; ! Dialogue
```

Due to erroneous synchronisations, tasks can constantly block one another; such a deadlock, e.g., could arise due to the following program organisation:

Task T1	Task T2
REQUEST S1;	**REQUEST** S2;
1st segment	1st segment
REQUEST S2;	**REQUEST** S1;
2nd segment	2nd segment
RELEASE S2;	**RELEASE** S1;
RELEASE S1;	**RELEASE** S2;

If tasks T1 and T2 are in their first segments at the same time, a deadlock occurs, for on the one hand, T1 is blocked by the execution of statement "REQUEST S2;", because T2 has already built up the lock with the semaphore variable S2, and on the other hand, T2 is blocked by the execution of statement "REQUEST S1;". If no other task executes a release, the blockings remain permanently.

To avoid such situations, lists of semaphore variables can be stated for locking statements.

By executing the statement

REQUEST Name§Sema [, Name§Sema]$^{\cdots}$;

the running task is suspended, if at least one of the indicated semaphore variables is in state "locked"; the task is not continued until none of the semaphore variables is locking any longer. If none of the indicated semaphore variables is locking, the executing task is continued after decrementing the values of each of these semaphore variables by 1.

The statement

RELEASE Name§Sema [, Name§Sema]$^{\cdots}$;

has the effect, as if a release statement were executed for each of the indicated semaphore variables without any interruption.

In the last example, the deadlock could be avoided by the following program organisation:

Task T1	Task T2
REQUEST S1, S2;	**REQUEST** S1, S2;
1st segment	1st segment
2nd segment	2nd segment
RELEASE S1, S2;	**RELEASE** S1, S2;

The order of the semaphore variables in the list is of no significance.

After the TRY operator, only one single semaphore variable may be given, a list of semaphore variables is not allowed.

The general forms for the declaration of semaphore variables as well as locking and release statements read:

SemaDeclaration ::=
{ **DECLARE** | **DCL** } Identifier_or_IdentifierList [DimensionAttribute] **SEMA** [GlobalAttribute]
[**PRESET** (IntegerWithoutPrecision [, IntegerWithoutPrecision] ···)];

RequestStatement ::=
REQUEST Name§Sema [, Name§Sema] ···;

ReleaseStatement ::=
RELEASE Name§Sema [, Name§Sema] ···;

TRY-Operator ::=
TRY Name§Sema

Thus, also arrays of semaphore variables are possible; the various elements are addressed by indices in the request and release statements.

Semaphore variables must be declared at module level. Upon declaration, semaphore variables can also be initialised explicitly by PRESET statement; the given values are assigned to the corresponding semaphore variables according to their order.

Example:

Upon declaration, values 3 and 5 shall be assigned to the semaphore variables S1 and S2, respectively.

DCL (S1, S2) **SEMA** **PRESET**(3, 5);

The general form for the specification of semaphore variables reads:

SemaSpecification ::=
{ **SPECIFY** | **SPC** } Identifier_or_IdentifierList
[VirtualDimensionAttribute] **SEMA**
{ GlobalAttribute | IdentificationAttribute };

9.3.2 Bolt Variables (BOLT) and Statements (ENTER, LEAVE, RESERVE, FREE)

We assume that in different tasks the same data are used for calculation, but not modified (e.g., comparison quantities for supervision processes); additionally, another task shall re-establish these data (e.g., establishing new supervision data). It shall be granted that the processes for modifying and using data mutually exclude each other; in contrast, the simultaneous use of data (by various tasks) is desired.

These problems can principally be solved with the described locking and release statements; the formulation, however, is relatively complicated, and the run time for the execution considerable. Thus, four more statements working on variables of data type bolt are provided.

A bolt variable can take on the states “locked”, “lock possible”, or “lock not possible”, depending on whether the associated resource is used exclusively, is free or is used simultaneously.

Bolt variables, e.g., are declared as follows:

DCL Identifier_or_IdentifierList **BOLT**;

After its declaration, a bolt variable has the state "lock possible".

Let the bolt variable B be associated to a resource. When entering a critical section for the exclusive use of this resource, the access by other tasks is locked by executing the statement

RESERVE B;

When leaving this critical section, the release takes place by the statement

FREE B;

The critical sections for the simultaneous use of the resource are initiated and terminated, respectively, by executing the statements

ENTER B; or **LEAVE** B;

The exact description of the effect of theses statements considers the state and the modification of the bolt variables:

Effect of RESERVE B; If B has state "lock possible", B assumes state "locked"; otherwise, the executing task is suspended and placed in a queue associated with B.

Effect of FREE B; B assumes state "lock possible". Furthermore, all tasks waiting in the queue of B due to a RESERVE statement are released; the tasks repeat their locking statements in the order of their priorities. If no task is waiting due to a RESERVE statement, all other waiting tasks, due to their ENTER-instruction, are released, in turn repeating the ENTER statement in the order of their priorities.

Effect of ENTER B; If B has state "locked", or if a task is in the queue of B due to a RESERVE statement, the executing task is suspended and placed in the queue of B. Otherwise, B assumes state "lock not possible", to prohibit exclusive access; furthermore, the (internally noted) number Z of "using" tasks is incremented by 1.
(I.e., tasks executing "ENTER B" are continued not before the tasks requiring exclusive access to the resource.)

Effect of LEAVE B; If Z = 1, this statement has the same effect as "FREE B;". Otherwise, Z is decremented by 1, and B keeps the state "lock not possible".

Example:

A task Measurement continuously acquires values of comparison quantities needed by the tasks Control and Disposition for calculations from a process to be supervised. It shall be granted that Measurement changes the comparison quantities only if they are not used; in contrast, Control and Disposition shall use the comparison quantities simultaneously.

For this, a bolt variable Vvalue is declared; in the bodies of the three tasks the critical sections of the modification or the use of the comparison quantities are initiated or terminated, respectively, as follows:

In the body of Measurement:

```
...
RESERVE Vvalue;
    ! modification
FREE Vvalue;
...
```

In the bodies of Control and Disposition:

...
ENTER Vvalue;
    ! use
LEAVE Vvalue;
...

All statements for bolt variables are also defined for lists of bolt variables — analogously to the request and release statements for semaphore variables.

Generally, bolt variables can be declared and used as follows:

BoltDeclaration ::=
    { **DECLARE** | **DCL** } Identifier_or_IdentifierList [DimensionAttribute] **BOLT**
    [GlobalAttribute] ;

BoltSpecification ::=
    { **SPECIFY** | **SPC** } Identifier_or_IdentifierList [VirtualDimensionAttribute]
        BOLT { GlobalAttribute | IdentificationAttribute } ;

BoltStatement ::=
    RESERVE Name§Bolt [, Name§Bolt] ··· ;
    | **FREE** Name§Bolt [, Name§Bolt] ··· ;
    | **ENTER** Name§Bolt [, Name§Bolt] ··· ;
    | **LEAVE** Name§Bolt [, Name§Bolt] ··· ;

Therefore, also arrays of bolt variables can be declared.

Bolt declarations must take place at module level.

9.4 Interrupts and Interrupt Statements

9.4.1 Declarations of Interrupts and Software Interrupts

An interrupt is a message of the controlled process via an interrupt input line of the interrupt controller to the operating system, which, after occurrence of the interrupt has to initiate the reaction planned by the programmer for that case, e.g., "Upon occurrence of interrupt Ready, task Supply is to be started".

The interrupt channels provided in a computer system are described in the user manual, giving their (system) names. The interrupt channels required by a PEARL program are declared in the system part, where user names can be associated to them. Under these user names, they are specified as interrupts in the problem part, to be able to use them in the task control statements (see 9.2) and the interrupt statements.

Example:

MODULE;
SYSTEM;
    Ready: Hard_Int(7); ! Hard_Int is the system name

PROBLEM;

```
SPECIFY Ready INTERRUPT;
...
Initialisation: TASK;
                ...
                WHEN Ready ACTIVATE Supply;
                ...
                END; ! Initialisation

...
Supply:         TASK PRIORITY 2;
                ! task body
                END; ! Supply
...
```

The general form of an interrupt specification reads:

InterruptSpecification ::=
  { **SPECIFY** | **SPC** } Identifier_or_IdentifierList { **INTERRUPT** | **IRPT** }
  [GlobalAttribute];

Most modern computers and operating systems, however, do not provide the possibility to address interrupt channels anymore, because hardware oriented device control and supervision are taken over by programmable logic controllers. The possibilities for interrupts provided by PEARL, however, are also usable for so-called *software interrupts* introduced into the system part under their own system names. The user manual gives information about the system name and usage possibilities.

9.4.2 Interrupt Statements (TRIGGER, ENABLE, DISABLE)

First, interrupts are disabled by their declarations and have to be enabled by an enable statement.

If the effect of an enabled interrupt, e.g., upon malfunctions in the technical process, shall be suppressed, i.e., start conditions for task statements related to it shall stay ineffective although the interrupt occurs, a disable statement is to be executed for this purpose, which in turn can be revoked again by executing an enable statement.

DisableStatement ::=
  DISABLE Name§Interrupt;

EnableStatement ::=
  ENABLE Name§Interrupt;

The validity range of the disable statement starts with its execution and ends upon execution of an enable statement.

Example:

```
MODULE;
SYSTEM;
    Alarm: INT;
    ...
PROBLEM;

    SPC Alarm INTERRUPT;
    ...
```

```
    Init: TASK MAIN;
        WHEN Alarm ACTIVATE Recovery;
        ENABLE Alarm;
        ...
        END; ! Init

    Recovery: TASK PRIO 1;
        DISABLE Alarm;
            ! checking the reason
            ! initiate reaction
        ENABLE Alarm;
        END; ! Recovery
    ...
MODEND;
```

The test of real time programs requires sometimes to simulate the effect of interrupts, particularly if the technical process has not been connected to the computer, yet. For such simulations, the trigger statement is provided:

TriggerStatement :=
TRIGGER Name§Interrupt;

Example:

```
MODULE;                         MODULE(Test);
SYSTEM;                         PROBLEM;

    Ready: Soft_Int;                SPC Ready IRPT GLOBAL;
    ...
PROBLEM;                            ...
    ...
    WHEN Ready                      /* e.g., at random times */
        ACTIVATE Control;           TRIGGER Ready;
    ...                             ...
MODEND;                         MODEND;
```

Chapter 10

Input and Output

The input and output statements enable the transfer of data from the working memory of the computer to an external data station (output) and vice versa, the transfer of data from an external data station to the working memory (input). Primarily, data stations are standard peripheral devices (printer, console, disk, magnetic tape, keyboard, etc.) or process peripherals (sensors, actuators, embedded controllers, etc.).

On these *system defined* data stations, *user defined* data stations can be created in the program to store data, e.g., on disks, magnetic tapes, printers, etc.

In the I/O statements of the problem part, the data stations are addressed under freely selectable, logical user names; the features of the data stations are to be specified before mentioning these names.

In conventional programming languages, the association of the logical identifiers (user names) for data stations used in the program to the devices of a particular computer system is done by additional control definitions when setting up a compiler program (job control definitions). In PEARL, these associations take place in standardised notation by declarations in the system part.

10.1 System Part

The system part serves for describing the used I/O configurations of a PEARL program. A system name must be associated with all devices of a computer system which can directly be addressed with PEARL I/O statements. A list of these devices and their system names can be found in the PEARL user manual of the corresponding target system. In the system part, freely selectable user names must be assigned to the system names of the needed devices. The devices can only be addressed in I/O statements of the problem part via these user names. If there are several devices of the same type, they can also be distinguished by an index added to the system name. Thus, a system name has the following general form:

SystemName ::=
    Identifier [(nni§index)]

nni ::=
    IntegerWithoutPrecision§NonNegative

For most of the process devices, further details are necessary, such as the channel number, and for digital I/O channels, even the selection of single bits can be desired. Thus, the following extension after the system name is possible for process devices:

ExtensionProcessDevice ::=
    *nni§ChannelNumber [*nni§Position [, nni§Width]]

With “Position”, e.g., the initial bit position of the device connection of a digital input/output, with “Width”, the number of bits of this device connection can be given.

Thus, a system part consists of a number of device associations of the following form:

DeviceAssociation ::=
    UserName: SystemName [ExtensionProcessDevice]

UserName ::=
    Identifier

For the example of a system part as given below, a fictive computer system with the following system names is assumed:

SystemName	Device
STDIN	standard input (console)
STDOUT	standard output (console)
SERIAL	for I/O with serial interfaces
DISC	for reading and writing files (disk, floppy,...)
DIGIO	digital I/O

PEARL example for the system part:

```
MODULE (example);
SYSTEM;
    termin:   STDIN;
    termout:  STDOUT;
    file:     DISC;
    tty_1:    SERIAL(1);
    tty_4:    SERIAL(4);
    counter:  DIGIO*0;
    switch:   DIGIO*1*16,1;
    motor:    DIGIO*1*1,4;
    ...
PROBLEM;
    ...
```

The available devices and the associated system names must be taken from the corresponding user manual of a PEARL implementation. There it is also described, whether (and, if yes, how) system names can be changed. For most PEARL systems there is an additional possibility to integrate device drivers and their system names created by the user into the PEARL system and to address them with PEARL I/O statements.

10.2 Specification and Declaration of Data Stations (DATION) in the Problem Part

10.2.1 System Data Stations

Before using system defined data stations, they must be specified in the problem part, defining their associated user names. For the declarations from the above example, this can take place as:

SPC	termin	**DATION IN**	**ALPHIC**,
	termout	**DATION OUT**	**ALPHIC**,
	file	**DATION INOUT**	**ALL**,
	tty_1	**DATION IN**	**ALPHIC**,
	tty_4	**DATION INOUT**	**ALL**,
	counter	**DATION IN**	**BASIC**,
	switch	**DATION OUT**	**BASIC**,
	motor	**DATION OUT**	**BASIC**;

The general form reads:

DationSpecification ::=
{ **SPECIFY** | **SPC** } IdentifierDenotation TypeDation [GlobalAttribute] ;

The different attributes in a system dation specification describe the fundamental features of the physical device to be communicated with. The features of a device and the possible attributes are described for all devices in the user manual.

10.2.2 User Defined Data Stations

By a dation declaration, a logical, so-called user defined data station (or user dation) is created on a physical device (system dation). The allocation to a device is determined by the CREATED attribute.

All input/output statements described in the following refer to user dations — the direct declaration of system dations leads to run time errors.

DationDeclaration ::=
{**DECLARE** | **DCL**} IdentifierDenotation TypeDation [GlobalAttribute]
CREATED (Name§SystemDefDation);

Hence, no arrays of data stations may be declared. However, it is possible to declare arrays of references to data stations to enable also indexed addressing of data stations.

The various attributes enable the detection of contradictions between the features of data stations and their way of using in I/O statements already at compilation time.

TypeDation ::=
DATION SourceSinkAttribute ClassAttribute
[Structure] [AccessAttribute]

Each data station is source and/or sink of a data transmission. The corresponding feature must be given upon declaration:

SourceSinkAttribute ::=
IN | **OUT** | **INOUT**

IN means that this data station is a source for data, i.e., it may only appear in such data transmission statements which transmit these data *into* the working memory (e.g., digital inputs, keyboard).

Data stations specified with OUT may only be used as sinks for outputs from the working memory (e.g., printer).

A data station with attribute INOUT allows for data transmissions in both directions (e.g., disk).

The data transmissions take place with the computer internal format of the data or by means of conversion

between computer internal and external format. For this, PEARL provides three different kinds of I/O statements:

- The READ/WRITE statements for transmission in computer internal format (e.g., for disk data, see 10.4.
- The PUT/GET statements for transmission with conversion between internal format and representation in the character set available on the data station (e.g., for printer output, see 10.5).
- The TAKE/SEND statements for transmitting process data (see 10.7).

The data transmission to or from a data station can only take place in one of the given ways.

The selection is made when declaring the data station by means of the class attribute stating to which of the three classes the data belong:

- If the READ/WRITE statements are to be used, i.e., if the data station records data in computer internal form, the type of data to be transmitted is stated as class attribute, e.g., FIXED, or FLOAT (53), or BIT (16), or ALL (for various types).
- If the data are represented alphanumerically on the data stations (case PUT/GET), the data station gets the class attribute ALPHIC.
- Data stations for transmissions with the TAKE/SEND statements have class attribute BASIC.

The general form of the class attribute reads:

ClassAttribute ::=
    ALPHIC | **BASIC** | TypeOfTransmissionData

TypeOfTransmissionData ::=
    ALL | SimpleType | CompoundType

SimpleType ::=
    TypeInteger | TypeFloatingPointNumber | TypeBitString |
    TypeCharacterString | TypeTime | TypeDuration

CompoundType ::=
    IO-Structure | Identifier§ForNewTypeFromSimpleTypes

IO-Structure ::=
    STRUCT [IO-StructureComponent [, IO-StructureComponent] ···]

IO-StructureComponent :=
    IdentifierDenotation
    { SimpleType | IO-Structure | Identifier§ForNewTypeFromSimpleTypes }

Hence, the type of the transmission data may also be a multiply structured structure or a newly defined type, but no component being of type reference.

Example:

...
TYPE Kind_Structure **STRUCT**
    [(No, Number) **FIXED**,

Weight **FLOAT**, ...];
DCL Kind_File **DATION INOUT** Kind_Structure ...,
Tab **DATION INOUT FIXED** ...;

The attribute ALL includes all other possibilities of type of transmission data.

Example:

On a disk memory drive with system name PSP31 and user name Disk, a file File1 for the input of FIXED quantities and a file File2 for the input/output of FLOAT quantities with computer internal format are to be created.

SYSTEM;
Disk: PSP31;

PROBLEM;
SPC Disk **DATION INOUT ALL** ...;
DCL File1 **DATION IN FIXED** ... **CREATED** (Disk);
DCL File2 **DATION INOUT FLOAT** ... **CREATED** (Disk);

The source/sink attribute and the class attribute must be defined for each data station, in contrast to the now described attributes for the structure and access possibilities of a data stations.

The smallest data set transmitted to or from a data station is called data element. Its type is determined by the class attribute. Several data elements can be combined into a record (synonymously, line), and several records to a segment (syonymously, page), i.e., all elements make up a 1-, 2-, or 3-dimensional array. For this, the number of data elements in a line, the number of lines in a page, and the number of pages must be denoted in the structure attribute:

Typology ::=
DIM ({ * | pi } [, pi [, pi]]) [**TFU** [**MAX**]]

The pi denotation furthest to the right always denotes the number of elements per line, the next (possibly lacking) pi denotation denotes the number of lines per page, and the following (possibly lacking) pi denotation denotes the number of pages. The denotation * means that the corresponding number is not limited. E.g., a data station Printer with 120 characters per line, 60 lines per page, and any number of pages can obtain the structure (*, 60, 120).

The following combinations are possible:

- 3-dimensional typology
  DIM (number of pages, number of lines, number of elements), or
  DIM (*, number of lines, number of elements)
- 2-dimensional typology
  DIM (number of lines, number of elements), or
  DIM (*, number of elements)
- 1-dimensional typology
  DIM (number of elements), or
  DIM (*)

Furthermore, the typology indicates, how many data elements are transmitted *at least* upon the execution of *one* data transmission statement:

- If the attributes TFU and MAX are lacking, single data elements, lines or pages can be transmitted.

- Attribute TFU means that only lines or pages are transferable.
  If the actual number of data elements in a data transmission statement is less than the number of data elements of a line or page, this line or page, respectively, is implicitly padded with spaces (ALPHIC data stations) or zeros (BASIC data stations).

  Example:

  DCL Printer **DATION OUT ALPHIC DIM**(*, 60, 120) **TFU** ...;
  ...
  PUT 'PEARL' **TO** Printer;

  This PUT statement has the effect that the five characters P, E, A, R, L are written out in one (new) line by Printer.

- TFU MAX corresponds to TFU.

The possible ways of accessing a data station are determined by the access attribute:

AccessAttribute ::=
  { DIRECT | **FORWARD** | **FORBACK }** [**NOCYCL** | **CYCLIC**]
  [**STREAM** | **NOSTREAM**]

DIRECT means that (based on a transmitted data element) any data element can directly be accessed, giving the position of the element (see 10.4, 10.5).

The attributes FORWARD and FORBACK mean sequential access; i.e., the access may (based on a transmitted element) only take place in the order determined by the structure — for FORWARD only forward, for FORBACK in both directions —, possibly giving the relative position of the wanted element to the element just transmitted (see 10.4, 10.5).

NOCYCL, CYCLIC, STREAM, and NOSTREAM are treated in the context of the I/O statements in 10.4 and 10.5.

Example:

On a disk drive with system name PSP31 and user name Disk a file for a table with 300 lines and 5 columns (elements per line) is to be created. Let the table elements be floating point numbers; the access takes place to elements directly and only reading.

MODULE;

SYSTEM;
  Disk: PSP31;
  ...
PROBLEM;
  SPC Disk **DATION INOUT ALL**;
  DCL Table **DATION IN FLOAT DIM**(300, 5) **DIRECT**
    GLOBAL CREATED (Disk);
  ...

Examples for the declaration of data stations for inputs/outputs in character form are described in 10.4 and 10.5.

10.3 Opening and Closing of Data Stations (OPEN, CLOSE)

Before a data station may be used for the first time in a data transmission statement, it must be opened by the open statement:

OpenStatement ::=
  OPEN Name§Dation [**BY** OpenParameter [, OpenParameter] ···] ;

When executing the open statement, a data station with typology is positioned at its beginning.

The open parameters serve to handle data stations containing identifyable files. E.g., a system defined data station Disk can possess a file TAB1, which is also maintained after terminating the program under this name. Later on, the same or another program can create a user defined data station Table on Disk, identified with file TAB1 in the open statement.

OpenParameter ::=
  IDF ({Name§CharacterVariable | CharacterStringConstant }) |
  RST (Name§ErrorVariable-FIXED) |
  { **OLD** | **NEW** | **ANY** } |
  { **CAN** | **PRM** }

The open parameters of the open statement must belong to different subsets.

Meaning of the parameters:

- IDF (Name§CharacterVariable | CharacterStringConstant)
  The value of the specified character variable or the specified character string constant is the name of the file to be identified with the data station named Name§Dation.
- RST (Name§ErrorVariable-FIXED)
  If an error occurs during the OPEN execution, the specified variable is assigned an error number unequal to zero; in the error-free case, it is set to zero (see 10.8).
- OLD
  If there is a file with IDF name, it its allocated on the notated data station. Otherwise, or if IDF is lacking, an error message is given, or the RST variable is set with the error number.
- NEW
  A file with IDF name is created and identified with the notated data station. If there is already a file with this name, or if IDF is lacking, an error message is given, or the RST variable is set with the error number.
- ANY
  If there is already a file with IDF name, it is identified with the notated data station. Otherwise, a new file is created for it. If IDF is lacking, a new file is created under a name determined by the system and identified with the denoted data station.
- CAN (from "cancel")
  The file is no longer to be made accessible after executing the close statement (see below).
- PRM (from "permanent")
  The file is still there after executing the close statement, and again accessible with the same name after re-executing an open statement.

If open parameters are lacking, ANY and PRM are assumed.

When executing the close statement, a file is closed; i.e., it is usable again not before the execution of an open statement.

CloseStatement ::=
  CLOSE Name§Dation [**BY** CloseParameter [, CloseParameter] ···] ;

The settings for closing a data station made in the open statement can be overwritten by a close statement:

CloseParameter ::=
  CAN | **PRM** | **RST** (Name§ErrorVariable-FIXED)

Generally, the following rules hold:

- Not every task executing an access to a data station must execute an open or close statement.
- However, at least one open statement must be executed concerning the access to a data station.
- The same number of close statements and open statements must be executed to close the data station.
- Corresponding open and close statements need not be executed by the same task.
- Upon lacking parameters, ANY and PRM are assumed, unless earlier executed close or open statements have made explicit settings.

Example:

```
MODULE;
SYSTEM;
    Printer: DRUA;
    Disk: PSP31;
    ...
PROBLEM;
    SPC Printer DATION OUT ALPHIC;
    DCL Tab_Prot DATION OUT ALPHIC DIM(*, 50, 30) FORWARD
        GLOBAL CREATED(Printer);
    SPC Disk DATION INOUT ALL;
    DCL Table DATION IN FLOAT DIM(300, 5) DIRECT
        GLOBAL CREATED(Disk);

Start: TASK MAIN;
    OPEN Tab_Prot;
    OPEN Table BY IDF('TAB-1'), OLD;
    ACTIVATE Prot;
    ...
    END; ! Start

Prot: TASK;
    ! Data transmission statements with Table and Tab_Prot
    ...
    CLOSE Tab_Prot;
    CLOSE Table;
    END; ! Prot
...
MODEND;
```

10.4 The Read and Write Statements (READ, WRITE)

The read statement serves for the input, the write statement for the output of data without converting the computer internal representation (binary input/output). Data can be transmitted to or from the connected devices (e.g., a file on disk or magnetic tape). The corresponding data stations must be declared with the class attribute "TypeOfTransmissionData".

Examples:

1. Columns 4 and 5 in the dation Table (cf. the example on page 110) are to be replaced by re-calculated values.

```
...
DCL (x, y, z) FLOAT;
...
FOR line FROM 1 TO 300
REPEAT
    ! calculation of x, y, z
    WRITE x, SIN(y+z) TO Table BY POS(line, 4);
END;
```

2. A task Measurement periodically acquires 14 temperature values (Procedure Get_Temp), processes them, and writes them sequentially in blocks with 14 values in a logbook on the disk.

```
SYSTEM;
    File: DISC;
PROBLEM;
    SPC Get_Temp PROC(i FIXED) RETURNS (FIXED(15)) GLOBAL;
    SPC File DATION INOUT ALL;
    DCL Logbook DATION OUT FIXED(15) DIM(*, 14) TFU FORWARD
        CREATED (File);
    DCL Num_Temp INV FIXED(15) INIT(14);
Start: TASK MAIN;
    OPEN Logbook;        ! Positioning to beginning
    ALL 10 SEC ACTIVATE Measurement;
    ...
    END; ! Start
Measurement: TASK;
    DCL Temperature (Num_Temp) FIXED(15);
    FOR i TO Num_Temp
    REPEAT
        Temperature(i) := Get_Temp(i);
    END;
    ! Processing the measured values
    WRITE Temperature TO Logbook;
    END; ! Measurement
```

The general forms of the read and write statements read:

```
ReadStatement ::=
    READ [ { Name§Variable | Segment } [ , { Name§Variable | Segment } ] ··· ]
    FROM Name§Dation [ BY Position [ , Position ] ··· ] ;

WriteStatement ::=
    WRITE [ { Expression | Segment } [ , { Expression | Segment } ] ··· ]
    TO Name§Dation [ BY Position [ , Position ] ··· ] ;
```

Segment ::=
    Name§Field ([Index ,] ··· Index : Index)

Index ::=
    Expression§WithIntegerAsValue

Position ::=
    AbsolutePosition | RelativePosition | **RST** (Name§ErrorVariable-FIXED)

AbsolutePosition ::=
    { **COL** | **LINE** } (Expression) |
    POS ([[Expression ,] Expression ,] Expression) |
    SOP ([[Name ,] Name ,] Name)

RelativePosition ::=
    { **X** | **SKIP** | **PAGE** } [(Expression)] |
    ADV ([[Expression ,] Expression ,] Expression)

Upon entry with the read statement, the addressed data elements are read one after the other and correspondingly assigned to the variables in the variable list. The variables are assigned according to the general rules for assignments. If an element of the variable list is an array, the addressed data are assigned by rows; if it is a structure, the data are assigned to the structure components in the order determined by the structure declaration.

For simplicity in writing, the elements (of the last dimension) of an array following one another in the variable list can be declared in form of a segment. Let list be an array with ten elements list(1), ... , list(10); then the two following statements are equivalent:

READ list(2), list(3), list(4) ... ;
READ list (2 : 4), ... ;

All position expressions are completely evaluated, before values are assigned to the data elements.

Example:

Read X from position 3 and Y from position 5 from the dation file:

READ X **FROM** File **BY** **POS**(3) ;
READ Y **FROM** File **BY** **POS**(5) ;

The execution of the statement

READ X, Y **FROM** File **BY** **POS**(3), **POS**(5) ;

however, results in that X being read from position 5 and Y from the subsequent position.

The RST attribute (cf. 10.8) can occur anywhere in the position list. However, it does not become effective until it is evaluated. A position list is elaborated one after the other, starting from the left. If an error occurs, the elaboration of the I/O statement is aborted in this position, and the error reaction valid at this moment (error assignment to an RST variable or system reaction) is executed.

These statements analogously hold for the write statement.

The type of the variables in the variable list of the read statement must be compatible with the class attribute of the given data station; this holds analogously for the results of the expressions in the expression list of the write statement.

The variable or expression list of the read or write statement may be lacking, if these statements shall only be used for positioning in the denoted data stations. In this case, however, a position must be given.

The values in the position list refer to the structure of the data station and determine the data elements to be transmitted. Thus, the values of the expressions must be of type FIXED and must be compatible with the structure.

When using an absolute position, i.e., a position independent from the actual data element, the data station must have the access attribute DIRECT. A relative position denotes the distance of the data element to be transmitted from the current data element; in this case, the data station must have the access attribute FORWARD, FORBACK or DIRECT.

In detail, the possible position attributes have the following meanings:

- COL (Expression)
  refers to the first dimension (from the right) of the structure and determines the i-th element in the current line of the data station, if i equals the value of the expression.

- LINE (Expression)
  refers to the second dimension of the structure and determines the i-th line of the current page of the data station, if i equals the value of the expression.

  Example:

  A 2-dimensional array corresponds to the structure (5,10):

  1 2 3 4 5 6 7 8 9 10

  1 2 3 4 5

  LINE(2)

  COL(4)

- POS ([[Expression§Page ,] Expression§Row ,] Expression§Column)
  gives the position of a data element in the n-dimensional structure (n = 1, 2, 3) of a data station. Lacking expressions are replaced by the respective current value.

  Example:

  With the statement

  READ x **FROM** File1 **BY** **POS**(3,2,8);

  the eighth data element of the second row of the third page of File1 is read into x. If this is followed by the statement

  READ x **FROM** File1 **BY** **POS**(4,5);

  the fifth data element of the fourth row of the third page of File1 is read into x.

- SOP ([[Name§Page ,] Name§Row ,] Name§Column)
  is the counterpart to POS. With SOP, the current positions of a dation can be assigned to the given program variables. The number of names in SOP may not exceed the number of dimensions in the addressed dation.

In the following, let i be the value of the respective expression.

- X [(Expression)]
  refers to the first dimension of the structure and determines the i-th element behind (i positive) or before (i negative), respectively, the current element in the current row of the data station; i = 0 denotes the current element.
- SKIP [(Expression)]
  refers to the second dimension of the structure and determines the start (the first element) of the i-th row behind (i positive) or before (i negative), respectively, the current row of the current page of the data station; i = 0 denotes the current row.
- PAGE [(Expression)]
  refers to the third dimension of the structure and determines the start of the i-th page behind (i positive) or before (i negative), respectively, the current page of the data station; i = 0 denotes the current page.

If the Expression denotation is not present for X, SKIP, or PAGE, respectively, value 1 is assumed. The value of a given expression must be positive, if the data station has the attribute FORWARD.

- ADV ([[Expression§Page ,] Expression§Row ,] Expression§Column)
  gives the distance of the data element to be transmitted from the current data element. Missing expressions are replaced by the value zero. If the data station has the attribute FORWARD, the value for the expression furthest to the left must be positive or zero.

Examples:

Let the corresponding data station File1 have the structure (10, 10, 10), let the actual position be (5,3,8).

position statement	new position	
X	(5,3,9)	
X (-5)	(5,3,3)	
X (4)	(5,4,2)	(1)
SKIP (2)	(5,5,1)	
SKIP (-1)	(5,2,1)	
PAGE	(6,1,1)	
PAGE (6)	(1,1,1)	(2)
PAGE (-4)	(1,1,1)	
ADV (2,5,1)	(7,8,9)	
ADV (1,0)	(5,4,8)	
ADV (-3,-2,1)	(2,1,9)	
ADV (1,8,0)	(7,1,8)	(3)

With a relative positioning, dimension boundaries may not be exceeded (cf. the marked examples), unless the data station possesses the access attribute STREAM or CYCLIC, respectively. STREAM allows to exceed the internal dimension boundary (examples (1) and (3)), but not to exceed the boundary of the highest dimension (example (2)). For this, a data station must possess the attribute CYCLIC. If error reactions shall occur upon exceeding the corresponding boundaries, the attributes NOSTREAM or NOCYCL, respectively, must be given. By default, STREAM and NOCYCL are assumed.

Example:

DCL File1 **DATION INOUT FIXED DIM**(10,10,10) **FORWARD CYCLIC**
    GLOBAL CREATED(Disk);

Let each time the current position be (5,3,8). Then it holds:

position statement	new position
X (6)	(5,4,4)
SKIP (8)	(6,1,1)
PAGE (7)	(2,1,1)
ADV (9,0,3)	(4,4,1)

10.5 The Get and Put Statements (GET, PUT)

The get statement serves for the input, the put statement for the output of data with conversion between computer internal and external, character oriented representations on ALPHIC data stations. To control this conversion, formats can be declared.

Examples:

1. The following text is to appear on the monitor of a storekeeper:

   ```
   ___article no:__4711
   ___stock:_______1281
   ```

   Program steps required:

   ...
   SPC monitor **DATION INOUT ALPHIC**;
   DCL stock_monitor **DATION INOUT ALPHIC DIM**(*,20,80)
   FORWARD GLOBAL CREATED (monitor);
   DCL (artno, stock) **FIXED**;
   ...
   PUT 'article no:', artno, 'stock:', stock **TO** stock_monitor
   BY X(3), **A**(13), **F**(4), **SKIP**, **X**(3), **A**(13), **F**(4);
   ...

2. Output of two values in standard format on a new page of the printer.

   ...
   SPC thermo_print **DATION OUT ALPHIC**;
   DCL printer **DATION OUT ALPHIC DIM**(*,50,120) **FORWARD**
   GLOBAL CREATED (thermo_print);
   DCL a **FIXED**(15), x **FLOAT**(31);
   ...
   a := 5;
   x := 2.33;
   ...
   PUT TO printer **BY PAGE**;
   PUT a, x **TO** printer **BY LIST**;

   The execution results in the following printer layout:

   ```
   _____5___2.33000E+00
   ```

3. Let data be stored in an input file on floppy disk in the following form:

   column 1 – 10 :article identifier (CHARACTER)
   column 12 – 20:quantity (FIXED)
   column 22 – 30:price per unit right-justified (e.g., ___124.57)

   They are to be read into the variables article_id, quantity, price:

```
...
SPC floppy_disk DATION INOUT ALL;
DCL input_file DATION IN ALPHIC DIM(*,80) TFU FORWARD
            GLOBAL CREATED (floppy_disk),
      article_id CHAR(10),
      quantity FIXED,
      price FLOAT;
...
GET article_id, quantity, price FROM input_file BY
      A(10), X, F(9), X, E(9), SKIP;
```

The general forms of the get and put statements read:

GetStatement ::=
  GET [{ Name§Variable | Segment } [, { Name§Variable | Segment }] ···]
  FROM Name§Dation [**BY** FormatPosition [, FormatPosition] ···] ;

PutStatement ::=
  PUT [{ Expression | Segment } [, { Expression | Segment }] ···]
  TO Name§Dation [**BY** FormatPosition [, FormatPosition] ···] ;

FormatPosition ::=
  [Factor] { Format | Position } |
  Factor (FormatPosition [, FormatPosition] ···)

Factor ::=
  (Expression§IntegerGreaterZero) | IntegerWithoutPrecision§GreaterZero

Format ::=
  FixedFormat | FloatFormat | CharacterStringFormats | BitFormat |
  TimeFormat | DurationFormat | ListFormat | R-Format | **RST** (Name)

Upon input with the get statement, the addressed data elements are read one after the other and correspondingly assigned to the variables in the variable list (analogously to the read statement). The assignment to the variables takes place according to the general assignment rules.

The input is terminated, when the the variable list is worked off. If there still are list elements, but no data elements, an error message is displayed.

When executing the put statement, the values of the expressions following PUT in the list are written out in the given order.

In the order of notation, a format in the format position list is associated with each variable in the variable list of the get statement, describing the external representation of the data on the named data station and used for converting into computer internal representation. The kind of format is determined by the type of the variable. Besides formats, the list of the format positions can also contain position statements (cf. 10.4) for positioning in the data station. If the variable list is not exhausted, yet, the further positionings are executed and the just mentioned association is continued with the next following format. If, on the contrary, the variable list is already exhausted, the following position attributes are elaborated, until a format is encountered or the list is worked off.

If the variable list contains an array or a segment, the following formats are associated with the array elements one after the other.

The number of the transmitted data elements is only determined by the variable list, not by the format position list. If there are more formats than variables, the surplus formats are neglected. If there are still

variables left, when the format position list is worked off, it is re-started with the first element of the format position list. In each case, the transmission is terminated, when the variable list is exhausted.

The above analogously holds for the put statement with “variable” replaced by “expression”.

The data station must possess the class attribute ALPHIC, a structure, and an access attribute. The selection of the access attribute possibly limits the positioning possibilities (cf. 10.4).

The format position list consists of format and position statements. To simplify notation, repetition factors may be used in the list. For instance, the format position list

X(2), F(12,3), X(2), F(12,3), X(2), F(12,3)

can be written in an easier way:

(3) (X(2), F(12,3))

The following table shows the permitted associations between formats and types of the data elements to be transferred:

format	data type
fixed format	FIXED, FLOAT
float format	FIXED, FLOAT
bit format	BIT
character string format	CHARACTER
time format	CLOCK
duration format	DURATION
list format	all given data types

In detail, the formats have the following forms and meanings (the position statements were explained in 10.4).

10.5.1 The Fixed Format (F)

FixedFormat ::=
    F (FieldWidth [, DecimalPositions [, ScaleFactor]])

FieldWidth ::=
    Expression§WithPositiveIntegerAsValue

DecimalPositions ::=
    Expression§WithNonNegativeIntegerAsValue

ScaleFactor ::=
    Expression§WithIntegerAsValue

The fixed format describes the external representation of decimal fixed point numbers. The field width w is the complete number of characters available for the decimal number; decimal positions d denote the number of digits behind the decimal point. The scale factor p can be both positive or negative; it causes that not the number itself, but its value multiplied by 10**p is transferred.

It is important that only integers can be transferred with the fixed format, but also the processing of fractional fixed point numbers is possible by scaling. These are converted to integers upon input and to fractional numbers again upon output.

1. **Output**

   (a) The decimal number is stored right-justified in a field of length w in the form
   [-] pi [. pi]
   where pi means positive integer. If the number does not occupy the entire field, the left part is padded with spaces.

   (b) If $0 > d$ or $w < d$, the character * is stored w times.

   (c) In case $w \leq 0$, no character is stored; an error is reported.

   (d) If $d = 0$ or is not given, only the integer part of the decimal number rounded without decimal point is written out.

   (e) Except for the zero immediately in front of the decimal point, leading zeros are suppressed.

2. **Input**

   (a) A field of length w is read, containing a decimal fixed point number in the following representation:
   [[+ | -] pi [. [pi]]]

   (b) Spaces preceding or following the number are ignored.

   (c) If the entire field is empty, value 0 is read in.

   (d) If no decimal point occurs in the representation, the last d digits are interpreted as positions following a decimal point. It must be $p \geq d$.

   (e) If a decimal point occurs in front of the last b digits in the representation, then it has priority over the specification by d. In this case, the statement of d has no meaning. It must be $p \geq b$.

   (f) If $w \leq 0$, no assignment takes place; an error is reported.

Example:

value	format	output	
13.5	F(7,2)	__13.50	
275.2	F(4,1)	****	error message!
22.8	F(5)	___23	
212.73	F(9,2,2)	_21273.00	
212.73	F(9,2)	___212.73	

10.5.2 The Float Format (E)

FloatFormat ::=
  E (FieldWidth [, DecimalPositions [, Significance]])

Significance ::=
  Expression§WithIntegerAsValue

The float format describes the external representation of decimal floating point numbers of the form

[+ | -] FloatingPointNumber

where the exponent consists of two digits. Field width and decimal positions have the same meaning as in the fixed format; significance s denotes the number of significant digits; i.e., the length of the mantissa.

E(w,d) is of equivalent to E(w,d,d+1), and E(w) to E(w,0).

1. **Output**
   The floating point number is stored right-justified in a field of length w. Otherwise, 1(a) in 10.5.1 holds.

   If 0 < w < d, an error reaction takes place.

   In case 0 < w > d > s, the mantissa is chosen so that it holds:

   $10^{s-d-1} \leq |$ mantissa $| < 10^{s-d}$

   For w = 0, no character is stored; the respective expression in the expression list is skipped.

   If d > 0, the number has the form

   [-] s-d digits . d digits E { + | - } exponent.

   The exponent is determined in such a way that the leading digit of the mantissa does not equal zero, if the number is different from zero.

   If d = 0, the number has the form

   [-] s digits E { + | - } exponent

   If w is too small to mention a digit of the mantissa, the character * is written out w times, followed by an error reaction.

2. **Input**
   A field of length w is read, containing a decimal floating point number in one of the possible representations (cf. 5.3).

   The statements 2(b) to 2(f) in 10.5.1 hold analogously.

value	format	output
-0.07	E(9,1)	␣-7.0E-02
2713.5	E(11,2,4)	␣␣27.13E+02
2721	E(8)	␣␣␣2E+03

10.5.3 The Character String Formats (A) and (S)

CharacterStringFormat ::=
  A [(Expression§NumberCharacters)] | **S** (Name§NumberCharactersVariableFixed)

The character formats describe the external representation of character strings (character quantities) of the form

Character$^{\cdots}$

Character String Format (A)

The value of the expression in character string format means the total number w of the character positions available for the representation.

1. **Output**
   If the format has form "A (Expression)", the character string is written out left-justified in the above presented form in a field of length w. If the character string consists of more than w characters, it is truncated on the right; if it consists of less than w characters, the field is padded with spaces on the right. If w = 0, no characters are written out, and the expression in the expression list of the put statement is skipped.

   If the expression is not given in the format, i.e., the format has the form "A", the string is written out in a field, whose length equals the string length.

2. **Input**
   Characters up to the maximum of w or until encountering the next record delimiter (e.g., CR) are read in. A record delimiter is not transferred into the character string variable.

   If w is smaller than length lg of the associated character string variable, the right part is padded with spaces; in case w > lg, the right part is truncated. If w = 0, a string of lg spaces is assigned to the variable.

Examples:

The output of the character string 'PEARL' in format

- A results in PEARL
- A(5) results in PEARL
- A(7) results in PEARL␣␣
- A(2) results in PE

The input of the character string 'PEARL␣␣' to a CHAR(5) variable text in format

- A is equivalent to text := 'PEARL';
- A(5) is equivalent to text := 'PEARL';
- A(7) is equivalent to text := 'PEARL';
- A(2) is equivalent to text := 'PE␣␣␣';

Character String Format (S)

The variable in character string format must be of type FIXED.

1. **Output**
   Identical with A format: the value of the given variable determines the width of the output field.
2. **Input**
   Characters up to the maximum lg (length of the associated character string variables) or until encountering the next record delimiter are read in; otherwise, the same rules like for the A format hold. Additionally, the number of characters (without record delimiter) read is assigned to the variable of the S format. Which record delimiters are defined for which devices is to be found in the respective PEARL user manual.

Example:

Command lines are to be read in from the terminal.

```
DCL buffer CHAR(80);
DCL length FIXED;
...
GET buffer FROM terminal BY S(length);
...
```

After the entry "abc<RETURN>" on the keyboard (the key <RETURN> creates a record delimiter), the variable "buffer" contains the characters "abc" and the variable "length" the value 3.

10.5.4 The Bit Format (B)

BitFormat ::=
{ **B** | **B1** | **B2** | **B3** | **B4** } [(Expression§NumberCharacters)]

The bit format describes the external representation of bit strings (bit quantities), namely (cf. 5.4)

- in binary form by the format: { B | B1 } [(Expression)] ,
- in form of tetrades by the format: B2 [(Expression)] ,
- in form of octades by the format: B3 [(Expression)] ,
- in hexadecimal form by the format: B4 [(Expression)] .

The value of the expression in bit format means the total number w of the character positions available for the representation.

1. **Output**
   If the expression is declared in the format, the bit string is written out left-justified in the above represented form in a field of length w. If the bit string consists of more than w characters, it is truncated on the right; if it consists of less than w characters, the field is padded with zeros on the right. If w is not given, and if the bit format thus has the form B | B1 | B2 | B3 | B4, the string is written out in a field, whose length equals the length of the string.

2. **Input**
   The expression must be given.

   A field of length w is read in, which must contain a bit string of the above described form. The field must not exclusively consist of spaces. Spaces preceding or following the string are ignored. The statements on the input with the A format hold likewise.

Example:

The output of the bit string '0101110' in the format

- B(5) results in 01011
- B2(3) results in 113
- B3(3) results in 270
- B4(2) results in 5C

Let the variable bit string be of type BIT(8); the input, there are the following possibilities:

data element to be entered	format	value of bit string
11111	B(5)	11111000
201	B2(3)	10000100
235	B3(3)	01001110
AB	B4(2)	10101011

10.5.5 The Time Format (T)

TimeFormat ::=
    T (FieldWidth [, DecimalPosition])

The time format describes the external representation of time data. The field width means the total number w of the character positions available for the representation; decimal positions stands for the number d of the digits for the fractional parts of seconds of the clock time.

1. **Output**
   The time is written out rigth-justified in a field of length w in the form

   [Digit] Digit : Digit Digit : Digit Digit [. pi]

   If the first digit is zero, it is replaced by spaces. In case d = 0, decimal point and fractional parts of seconds are not written out.

   If the output value does not occupy the entire field, the left part is padded up with spaces.

2. **Input**
   A field of length w is read in, which must contain a time in a permitted representation (see 1). Preceding and following spaces are ignored.

Examples:

value	format	output
12.30 hours 5.2 sec	T(12,1)	__12:30:05.2
8 hours	T(8)	_8:00:00

10.5.6 The Duration Format (D)

DurationFormat ::=
    D (FieldWidth [, DecimalPosition])

The duration format describes the external representation of durations. The value of the field width means the total number w of the character positions available for representation, the value of decimal positions means the number d of the digits for the fractional parts of seconds of the duration.

1. **Output**
   The duration is written out right-justified in a field of length w in the form

   [Digit] Digit_HRS_Digit Digit_MIN_Digit Digit [. pi]_SEC

   The rules from 10.5.5 (1) are valid.

2. **Input**
   A field of length w is read in, which must contain a duration in a permitted representation (see 1). Preceding and following spaces are ignored.

Examples:

value	format	output
11 hours 15 minutes	D(20)	11_HRS_15_MIN_00_SEC
100 milliseconds	D(24,3)	_0_HRS_00_MIN_00.100_SEC

The character _ means a space.

10.5.7 The List Format (LIST)

ListFormat ::=
LIST

The list format serves for the input/output of fixed, float, bit, char, clock, and dur quantities.

1. **Output**
   Subsequent output data are separated by two spaces, each. The data are written out in such a way, as if for a quantity of type

CHAR(k)	format	A(k)
BIT(k)	format	B(k)
FIXED(k)	format	F(n)
FLOAT(k)	format	E(m,m-7,m-6)
CLOCK	format	T(8)
DUR	format	D(20)

   with n = ENTIER(k/3.32) + 2, m = ENTIER (k/3.32) + 3 were declared.

2. **Input**
   The input data can have any form permitted for the representation of constants. They are separated by a comma or at least two spaces. If no constant is between two commas, the corresponding element of the variable list remains unchanged.

Examples:

data type	value	implicit format	output
FIXED(15)	127	F(6)	___127
FLOAT(31)	3.28E+28	E(12,5,6)	_3.28000E+28
BIT(8)	'EF'B4	B(8)	11101111

10.5.8 The R Format (R)

Sometimes, the same format position lists are used in more than one get or put statement. The R format serves to describe these lists only once. For this, a list with the so-called format declaration is introduced.

FormatDeclaration ::=
Identifier : **FORMAT** (FormatPosition [, FormatPosition] $^{\cdots}$) ;

Example:

Ftab : **FORMAT** (**X**(2), **F**(8,3), (3) (**X**(2), **E**(10,3))) ;

A format declared in such a way can be used in a get or put statement, stating its identifier:

R-Format ::=
R (Identifier§Format)

When transmitting data, the R format is replaced by the format position list contained in the indicated format declaration.

Example:

PUT a, x, y, z **TO** printer **BY** (Ftab) ;

The format position list in the format declaration must not contain any R format referring directly or indirectly (via another format declaration) to its own format declaration.

10.6 The Convert Statement (CONVERT)

The comfortable conversion of number values in character strings and vice versa is very important for many applications, e.g., for the creation of display masks or for data exchange via communication interfaces which cannot transfer binary data. The PUT and GET statements enable that in connection with data stations. Following that, the CONVERT statement is defined, performing formatted data exchange with a character string or character string variable, respectively, instead of a data station.

The general forms of the convert statement are:

ConvertToStatement ::=
 CONVERT Expression [, Expression] ··· **TO** Name§CharacterString
 [**BY** FormatPositionConvert [FormatPositionConvert] ···] ;

ConvertFromStatement ::=
 CONVERT Name§Variable [, Name§Variable] ··· **FROM**
Expression§CharacterString
 [**BY** FormatPositionConvert [, FormatPositionConvert] ···] ;

FormatPositionConvert ::=
 [Factor] { Format | PositionConvert } |
 Factor (FormatPositionConvert [, FormatPositionConvert] ···)

PositionConvert ::=
 RST (Name§ErrorVariable-FIXED) | **X** (Expression) | **ADV** (Expression) |
 POS (Expression) | **SOP** (Name§PositionVariable-FIXED)

All permitted formats have the same meaning as for the PUT and GET statements. The only exception is the S format. In the convert statement, the number of characters, which at this point in time was read from Expression§CharacterString or written into Name§CharacterString, respectively, is assigned to the variable in S format.

Example:

DCL (index, conv_error, number_of_bytes) **FIXED**,
 value **FLOAT**;

DCL string_out **CHAR**(40),
 string_in **CHAR**(20);
...
CONVERT 'Index =', index **TO** string_out **BY A**, **F**(4), **S**(number_of_bytes);
CONVERT index, value **FROM** string_in **BY RST**(conv_error), **F**(4), **E**(10,2);

10.7 The Take and Send Statements

The take statement serves for the input, the send statement for the output of data. These statements are provided for the transmission of process data, and for the data exchange with user specific drivers,

respectively. The data station must possess the class attribute BASIC.

TakeStatement ::=
  TAKE [Name§Variable] **FROM** Name§Dation
    [**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ···] ;

SendStatement ::=
  SEND [Expression] **TO** Name§Dation
    [**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ···] ;

RST-S-CTRL-Format ::=
  RST (Name§ErrorVariable-FIXED)
  | **S** (Name§Variable-FIXED)
  | **CONTROL** (Expression [, Expression [, Expression]])

The types of the "variable" in the take statement or the "expression" in the send statement, respectively, are implementation dependent.

The attribute definitions RST, S and CONTROL may be stated in any order, but only once, each. The meanings of the CONTROL and S formats are implementation dependent and, thus, to be found in the respective user manual of a PEARL implementation.

Example:

SYSTEM;
  motor: DIGEA *1*1,4;

PROBLEM;
  SPC motor **DATION OUT BASIC**;
  DCL on **INV BIT**(4) **INIT**('1010'B1);

  SEND on **TO** motor;

10.8 Error Handling in I/O Statements (RST)

Usually, errors are recognised when the execution of I/O statements lead to the termination of the responsible task, and an error message is written out. This standard reaction of the PEARL system is suppressed by defining

RST (Name§ErrorVariable-FIXED)

in the parameter list of the OPEN/CLOSE statement, or as format or position element in the other data transmission statements, respectively.

"Name" must denote a variable of type FIXED, in which an error number not equal to zero is written in the case of error. Upon error free execution of the I/O statement, the variable is set to zero. Possible errors and their identification are to be found in the PEARL user manual of the respective computer system.

The RST definition can be located in any position of the format or position list of PUT, GET, WRITE, READ and CONVERT statements; also multiple definitions with different variables are permitted. The RST definition does not change the error reaction before the RST element is evaluated in the format or position list. Upon recognition of an error, the I/O statement is immediately aborted, and the error reaction valid at this point in time (assignment of an error number to an RST variable or the system reaction) is carried out.

10.9 Interface for Additional Drivers

The diversity of — particularly in the world of PCs — existing I/O controllers and devices does not allow the compiler supplier to account for all I/O devices in the system part and create system names for them. To be able to address special I/O devices from PEARL in spite of that fact, a driver interface can be provided for certain operating systems, to which the PEARL programmer himself can connect drivers. This interface is described in the respective user manual (in the chapter "Open Driver Interfaces").

Chapter 11

Signals

When executing certain statements, internal events, so-called signals, can occur, leading to an interruption of the running task; such signals may, e.g., be an overflow in course of an arithmetic operation, a division by zero, or reaching the end of a file.

A program abortion can be avoided by enabling a corresponding error handling upon the occurrence of a signal.

The signals needed for a program are declared in the system part, where freely selectable user names can be assigned to them. Additionally, an error list can be defined to limit the signal scheduling to one or several particular error numbers.

UserNameDeclaration§For-SIGNAL ::=
    Identifier§UserName : Identifier§SIGNAL-SystemName
        [(Identifier§ErrorNumber [, Identifier§ErrorNumber] ···)]

The signals and their associated error numbers possible on a certain computer are described in the respective user manual, giving their system names and meanings.

Before signals are used, they must be specified under their user names in the problem part at module level.

Example:

IO_SIGNAL and ENDF are system names; c_error_open is a pre-defined constant.

MODULE;
    SYSTEM;
        OPEN_ERR : IO_SIGNAL (c_error_open);
        EOF : ENDF;
        ...
    PROBLEM;
        SPC (OPEN_ERR, EOF) **SIGNAL**;
        ...
MODEND;

The general form of the specification of signals reads:

SIGNAL-Specification ::=
    { **SPECIFY** | **SPC** } Identifier_or_IdentifierList **SIGNAL** [GlobalAttribute] ;

The reaction planned for the occurrence of a signal is scheduled with the following statement:

SchedulingSignalReaction ::=
    ON Name§Signal { [**RST** (Name§ErrorVariable-FIXED)] :
        SignalReaction | **RST** (Name§ErrorVariable-FIXED) };

SignalReaction ::=
    UnlabeledStatement

Instead of UnlabeledStatement, all Statements besides the Statement SchedulingSignalReaction are permitted, particularly blocks or procedure calls.

The Statement SchedulingSignalReaction is not permitted within BEGIN and REPEAT blocks, or as SignalReaction.

- Validity range of scheduling a signal reaction:
  The validity range of scheduling a signal reaction reaches from the execution of the corresponding ON statement to the end of the scheduling task or procedure, respectively.
  If another scheduling for the same signal (e.g., in a procedure called by the scheduling task) is run, it hides the preceding scheduling until the end of the validity range of the new schedule (in the example until the end of the scheduling procedure).
- Leaving a procedure (task) after the execution of a signal reaction:
  If a signal occurs within the validity range of scheduling a signal reaction, the procedure (task) in which the signal reaction is scheduled is left by an implicit RETURN (TERMINATE) after having executed the corresponding signal reaction, unless the signal reaction is left by a GOTO statement.
- Validitiy of signal schedulings during the execution of a signal reaction:
  If a signal reaction is scheduled at procedure level, only those ON schedulings are valid during the execution of this signal reaction which were already executed before calling this procedure.
  If the signal reaction is scheduled at task level, no ON schedulings are valid during the execution of this signal reaction.
- Validity of signal schedulings after leaving a signal reaction by GOTO:
  If the execution of a signal reaction scheduled at procedure or task level is left by GOTO, the signal schedulings which were already executed before triggering the signal, particularly those of the triggered signal, are valid again.

If the syntax variant

    ON Name§Signal **RST** (Name§ErrorVariable-FIXED);

is used the scheduling task (procedure) will not be terminated but continued after setting the ErrorVariable upon the occurrence of the corresponding signal.

To test the reaction scheduled for a signal, the occurrence of a signal can be simulated analogously to the occurrence of an interrupt:

InduceStatement ::=
    INDUCE Name§Signal [**RST** (Expression§ErrorNumber)];

Whereas it is possible to react to asynchronous exception situations (i.e., influences effecting from outside) by interrupts, the signal treatment serves exclusively for reacting to synchronous error states (i.e., cause and treatment of the error state originate from the same task).

Example:

The procedure Analysis shall sequentially analyse a logbook created in the course of a day; the single data elements of the LogBook are of type Event.

```
...
PROBLEM;
    SPC EOF SIGNAL,
        Tape DATION INOUT ALL;
    TYPE Event ...;
    DCL LogBook DATION IN event DIM(*) FORWARD CREATED(tape);

Analysis: PROC;
    DCL Input Event;
    ...
    OPEN LogBook;
    ON EOF:
        BEGIN
            CLOSE LogBook;
        END; ! ON EOF
    ...
    REPEAT
        READ Input FROM LogBook;
        ...
    END;
    END; ! Analysis
```

For testing, the statement

```
INDUCE EOF;
```

could be executed sporadically instead of the read statement.

If a signal is induced, whether by an error state or an induce statement, and no signal reaction is scheduled for it, the system reaction is triggered (i.e., generally an error message of the run time system and termination of the causing task).

By stating a variable after RST in the signal scheduling, the programmer gets access to the error number (error cause). In this case, the signal SignalName can be induced for the error with the number ErrorNumber by

```
INDUCE SignalName RST ( ErrorNumber ) ;
```

Example:

The signal TaskSignal shall be simulated with the error number “1010”, and the causing task shall react by writing out the error number on the console:

```
PROBLEM;
    SPC TaskSignal SIGNAL;
    DCL ErrorNummer FIXED;

Regulator: TASK PRIO 20;
    ON TaskSignal RST (ErrorNumber):
        PUT ErrorNumber TO Console;
        ...
    Test;
```

```
        ...
    END; ! Regulator

Start: TASK MAIN;
    ALL Ta ACTIVATE Regulator;
    END; ! Start

Test: PROC;
    INDUCE TaskSignal RST (1010);
    ...
    END; ! Test
```

Chapter 12

Appendix

12.1 Data Types and their usability

The following overview shows for each of the available data types, whether objects of this type may

- be summarised to arrays,
- occur as structure components,
- be formal procedure parameters,
- be results of a function procedure,
- be values of a reference variable,
- be transmitted to or from data stations,
- be provided with allocation protection,
- be global, or
- be provided with the initialisation attribute.

Objects of type SEMA, BOLT, IRPT, SIGNAL, DATION or array may only be passed on by means of identification (IDENT) as procedure parameters.

type	**usage**								
	array	struct.	para-meter	result type	ref. value	dation class	INV	GLOBAL	INIT
FIXED	x	x	x	x	x	x	x	x	x
FLOAT	x	x	x	x	x	x	x	x	x
BIT	x	x	x	x	x	x	x	x	x
CHAR	x	x	x	x	x	x	x	x	x
CLOCK	x	x	x	x	x	x	x	x	x
DUR	x	x	x	x	x	x	x	x	x
SEMA	x	–	x	–	x	–	–	x	–
BOLT	x	–	x	–	x	–	–	x	–
IRPT	x	–	x	–	x	–	–	x	–
SIGNAL	x	–	x	–	x	–	–	x	–
DATION	x	–	x	–	x	–	–	x	–
array	–	x	x	–	x	–	x	x	x
STRUCT	x	x	x	x	x	x	x	x	x
new type	x	x	x	x	x	x	x	x	x
REF	x	x	x	x	–	–	x	x	x
procedure	–	–	–	–	x	–	–	x	–
TASK	–	–	–	–	x	–	–	x	–
FORMAT	–	–	–	–	–	–	–	–	–
REF CHAR ( )	x	x	x	–	–	–	x	x	x
REF PROC	x	x	x	–	–	–	x	x	x
REF TASK	x	x	x	x	–	–	x	x	x
REF STRUCT [ ]	x	x	x	x	–	–	x	x	x

12.2 Predefined Functions

This appendix describes the functions known to the PEARL compiler. They can be used in the single modules without specifying them before. If one of the functions is specified in a module, no object may exist with the functions' name at module level.

12.2.1 The Function NOW

The function procedure NOW passes back the actual time or system time, resp., as value of type CLOCK. A specification of the function looks like this:

SPC NOW **PROC RETURNS (CLOCK) GLOBAL** ;

12.2.2 The Function DATE

The actual date can be received by invoking the function procedure DATE. The function result is a character string constant of length 10, containing the date in the form "year–month–day". Here an example for December 5, 1991: "1991–12-05". The function can be specified like this:

SPC DATE **PROC RETURNS (CHAR(10)) GLOBAL** ;

12.3 Syntax

Following meta characters are used in the syntax description:

meta character	meaning
::=	introduction of a Name (nonterminal symbol) for a language form
[ ]	bracketing of optional parts of a language form
|	separation of alternative parts of a language form
{ }	putting together several elements to a new element
···	one or multiple repetition of the preceding element (or of several elements bracketed by { } or [ ])
§	separates an explaining comment from a language form Name
/* */	comment brackets: includes an explaining text, possibly explaining the language form in detail instead of a formal description

All other elements occurring in the syntax rules are either Names of language forms or terminal symbols. Examples for terminal symbols are the PEARL keywords (printed boldly) or the characters semicolon ";", opening round bracket "(" and closing round bracket ")", or the apostrophe " ' "; the terminal symbols opening square bracket "**[**" and closing square bracket "**]**" are printed boldly to distinguish them from the meta symbols for optional parts. Attention: the round brackets are no meta characters and have thus no grouping effect!

The symbol PEARL-Program is the initial symbol of the following syntax description.

12.3.1 Program

PEARL-Program ::=
    Module ···

Module ::=
    MODULE [[( ] Identifier§OfTheModule [ )]] ;
    { SystemPart [ProblemPart] | ProblemPart }
    MODEND ;

12.3.2 System Part

SystemPart ::=
    SYSTEM ;
        [UserNameDeclaration§ForDationInterruptOrSignal] ···

UserNameDeclaration ::=
    Identifier§UserName: Identifier§SystemName [(nngz§Index)]
        [* nngz§Channel [* nngz§Position [, nngz§Width]]];
    | Identifier§UserName: Identifier§SIGNAL-SystemName
        [(Identifier§ErrorNumber [, Identifier§ErrorNumber] ···)] ;

nngz ::=

IntegerWithoutPrecision§NonNegative

12.3.3 Basic Elements

Digit ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Letter ::=
A | B | C | ... | Z | a | b | c | ... | z

Identifier ::=
Letter [Letter | Digit | _] ···

Constant ::=
Integer | FloatingPointNumber
| BitStringConstant | CharacterStringConstant
| TimeConstant | DurationConstant
| **NIL**

Integer ::=
IntegerWithoutPrecision [(Precision)]

IntegerWithoutPrecision ::=
Digit ··· | { 0 | 1 } ··· **B**

Precision ::=
IntegerWithoutPrecision

FloatingPointNumber ::=
FloatingPointNumberWithoutPrecision [(Precision)]

FloatingPointNumberWithoutPrecision ::=
{ Digit ··· . [Digit ···] | . Digit ··· } [Exponent]
| Digit ··· Exponent

Exponent ::=
E [+ | -] Digit ···

BitStringConstant ::=
' B1-Digit ··· ' { **B** | **B**1 } | ' B2-Digit ··· ' **B**2 | ' B3-Digit ··· ' **B**3 | ' B4-Digit··· ' **B4**

B1-Digit ::= 0 | 1

B2-Digit ::= 0 | 1 | 2 | 3

B3-Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

B4-Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

CharacterStringConstant ::=
    ' { CharacterBesidesApostrophe | '' | ControlCharacterSequence } ··· '

CharacterBesidesApostrophe ::=
    Digit | Letter | | _ | + | - | * | / | \ | (|) | [|] | : | . | ; | , | = | < | > | !
    | /* more printable characters of the machine character set */

ControlCharacterSequence ::=
    '\ { B4-Digit B4-Digit } ··· \'

TimeConstant ::=
    Digit ··· : Digit ··· : Digit ··· [. Digit ···]

DurationConstant ::=
    Hours [Minutes] [Seconds]
    | Minutes [Seconds]
    | Seconds

Hours ::=
    IntegerWithoutPrecision **HRS**

Minutes ::=
    IntegerWithoutPrecision **MIN**

Seconds ::=
    { IntegerWithoutPrecision | FloatingPointNumberWithoutPrecision } **SEC**

ConstantExpression ::=
{ + | - } FloatingPointNumber
    | { + | - } DurationConstant
    | Constant-FIXED-Expression

Constant-FIXED-Expression ::=
    Term [{ + | - } Term] ···

Term ::=
    Factor [{ * | // | **REM** } Factor] ···

Factor ::=
[+ | -]
{ Integer
| (Constant-FIXED-Expression)
| **TOFIXED** { CharacterStringConstant§OfLength1 | BitStringConstant }
| Identifier§Named-FIXED-Constant
}
[**FIT** Constant-FIXED-Expression]

12.3.4 Problem Part

ProblemPart ::=
PROBLEM; [{ Declaration | Specification | Identification } ···]

Declaration

Declaration ::=
LengthDefinition
| TypeDefinition
| VariableDeclaration
| FormatDeclaration
| ProcedureDeclaration
| TaskDeclaration
| OperatorDefinition
| PredenceDefinition

LengthDefinition ::=
LENGTH { **FIXED** | **FLOAT** | **BIT** | **CHARACTER** | **CHAR** } (Precision) ;

TypeDefinition ::=
TYPE Identifier§ForType { SimpleType | TypeStructure } ;

VariableDeclaration ::=
{ **DECLARE** | **DCL** } DeclareSentence [, DeclareSentence] ··· ;

DeclareSentence ::=
OneIdentifierOrList [DimensionAttribute]
{ ProblemDataAttribute | SemaAttribute | BoltAttribute | DationAttribute }

OneIdentifierOrList ::=
Identifier | (Identifier [, Identifier] ···)

DimensionAttribute ::=
(DimensionBoundaries [, DimensionBoundaries] ···)

DimensionBoundaries ::=
    [Constant-FIXED-Expression§ForLowerBoundary :]
    Constant-FIXED-Expression§ForUpperBoundary

ProblemDataAttribute ::=
    [**INV**] { SimpleType | StructuredType | TypeReference }
    [GlobalAttribute] [InitialisationAttribute]

SimpleType ::=
    { **FIXED** | **FLOAT** | **BIT** | **CHAR** | **CHARACTER** }
        [(Constant-FIXED-Expression§PrecisionResp.Length)]
    | **CLOCK**
    | { **DUR** | **DURATION** }

StructuredType ::=
    TypeStructure | Identifier§ForNewDefinedType

TypeStructure ::=
    STRUCT [StructureComponent [, StructureComponent] ···]

StructureComponent ::=
    OneIdentifierOrList§ForStructureComponent [DimensionAttribute]
    TypeAttributeInStructuremComponent

TypeAttributeInStructureComponent ::=
    [**INV**] { SimpleType | StructuredType | TypeReference }

TypeReference ::=
    REF
    { [VirtualDimensionList] [**INV**] { SimpleType | StructuredType }
        | [VirtualDimensionList] { TypeDation | **SEMA** | **BOLT** }
        | TypeProcedure | **TASK** | **INTERRUPT** | **IRPT** | **SIGNAL**
        | Type-VOID | **CHAR**( )
    }

VirtualDimensionList ::=
    ([, ···])

Type-VOID ::=
    STRUCT [ ]   /* only allowed in combination with REF */

SemaAttribute ::=
    SEMA [GlobalAttribute]
    [**PRESET** (Constant-FIXED-Expression [, Constant-FIXED-Expression] ···)]

BoltAttribute ::=

BOLT [GlobalAttribute]

GlobalAttribute ::=
GLOBAL [(Indicator§Module)]

InitialisationAttribute ::=
{ **INITIAL** | **INIT** } (InitElement [, InitElement] ···)

InitElement ::=
Constant
| Identifier§NamedConstant
| ConstantExpression

DationAttribute ::=
TypeDation
[GlobalAttribute]
CREATED (Indicator§UserNameForSytemDation)

TypeDation ::=
DATION SourceSinkAttribute ClassAttribute
[Topology] [AccessAttribute]

SourceSinkAttribute ::=
IN | **OUT** | **INOUT**

ClassAttribute ::=
ALPHIC | **BASIC** | TypeOfTransmissionData

TypeOfTransmissionData ::=
ALL | SimpleType | Indicator§ForRedeclaredType | TypeStructure
/* A Structure as transfer type may not contain any reference variable */

Topology ::=
DIM ({ * | Constant-FIXED-Expression }
[, { * | Constant-FIXED-Expression }
[, { * | Constant-FIXED-Expression }]]) [**TFU** [**MAX**]]
| **DIM** ([, [,]]) /* Virtual dimension declaration only permitted for SPC */

AccessAttribute ::=
{ **DIRECT** | **FORWARD** | **FORBACK** }
[**NOCYCL** | **CYCLIC**]
[**STREAM** | **NOSTREAM**]

FormatDeclaration ::=
Identifier : **FORMAT** (FormatOrPosition [, FormatOrPosition] ···) ;

ProcedureDeclaration ::=
  Identifier : { **PROC** | **PROCEDURE** } [ListOfFormalParameters]
  [ResultAttribute] [GlobalAttribute] ;

    ProcedureBody

  END;

ProcedureBody ::=
  [{ Declaration | Identification } ···] [Statement ···]

ListOfFormalParameters ::=
  (FormalParameter [, FormalParameter] ···)

FormalParameter ::=
  OneIdentifierOrList [VirtualDimensionList]
    ParameterType [**IDENT** | **IDENTICAL**]

ParameterType ::=
  [**INV**] { SimpleType | StructuredType | TypeReference } |
    TypeRealTimeObject | TypeDation

TypeRealTimeObject ::=
  SEMA | **BOLT** | { **INTERRUPT** | **IRPT** } | **SIGNAL**

ResultAttribute ::=
  RETURNS (ResultType)

ResultType ::=
  SimpleType | StructuredType | TypeReference

TaskDeclaration ::=
  Identifier : **TASK** [PriorityAttribute] [**MAIN**]
  [GlobalAttribute] ;

    TaskBody

  END ;

TaskBody ::=
  [{ Declaration | Identification }···] [Statement···]

PriorityAttribute ::=
  { **PRIORITY** | **PRIO** } Constant-FIXED-Expression§GreaterZero

OperatorDefinition ::=
OPERATOR OpName (OpParameter [, OpParameter]) ResultAttribute ;

ProcedureBody

END ;

OpName ::=
Identifier | + | − |*| / | // |**|==| / =|<=|>=|<|>|<>|><

OpParameter ::=
Identifier [VirtualDimensionList] ParameterType [**IDENT** | **IDENTICAL**]

PrecedenceDefinition ::=
PRECEDENCE OpName ({ 1 | 2 | 3 | 4 | 5 | 6 | 7 }) ;

12.3.5 Specification and Identification

Specification ::=
{ **SPC** | **SPECIFY** } SpecificationDefinition [, SpecificationDefinition] ··· ;

SpecificationDefinition ::=
OneIdentifierOrList
{ SpecificationAttribute | ProcedureUsageAttribute | TaskUsageAttribute }

SpecificationAttribute ::=
[VirtualDimensionList]
{ [**INV**] { SimpleType | StructuredType | TypeReference }
| **SEMA** | **BOLT** | **INTERRUPT** | **IRPT** | **SIGNAL** | TypeDation
}
[GlobalAttribute]

ProcedureUsageAttribute ::=
{ **ENTRY** | [:] **PROC** } [ParameterListFor-SPC] [ResultAttribute]
GlobalAttribute

ParameterListFor-SPC ::=
(FormalParameterFor-SPC [, FormalParameterFor-SPC] ···)

FormalParameterFor-SPC ::=
[Identifier§OnlyForDocumentation] [VirtualDimensionList]
ParameterType [**IDENTICAL** | **IDENT**]

TypeProcedure ::=
  PROC [ParameterListFor-SPC] [ResultAttribute] /* for REF PROC */

TaskUsageAttribute ::=
  TASK GlobalAttribute

Identification ::=
  { **SPC** | **SPECIFY**} Identifier [**INV**] Type IdentificationAttribute ;

IdentificationAttribute ::=
  IDENT (Name§Object)

12.3.6 Expressions

Expression ::=
  [MonadicOperator] Operand [DyadicOperator Expression] ···

MonadicOperator ::=
  + | - | Identifier§MonadicOperator

DyadicOperator ::=
  + | − |*| / | // |**|<|>|<=|>=|==| / =|><|<>|
  Identifier§DyadicOperator

Operand ::=
  Constant | Name | FunctionCall | ConditionalExpression
  | Dereferentiation | StringSelection | (Expression) | (Assignment)
  | **PRIO** [(Name§Task)] | **TASK** [(Name§Task)] | **TRY** Name§Sema

Name ::=
  Identifier [(Index [, Index] ···)] [. Name]

Index ::=
  Expression§WithIntegerAsValue

FunctionCall ::=
  Identifier§FunctionProcedure [ListOfActualParameters]

ListOfActualParameters ::=
  (Expression [, Expression] ···)

ConditionalExpression ::=
  IF Expression§OfType-BIT(1) **THEN** Expression **ELSE** Expression **FIN**

Dereferentiation ::=
  CONT { Name§Reference | FunctionCall }

StringSelection ::=
  Name§String . { **BIT** | **CHAR** | **CHARACTER** }
  ({ Constant-FIXED-Expression [: Constant-FIXED-Expression]
    | Expression [: Expression + Constant-FIXED-Expression]
    | Expression [: Expression] })

12.3.7 Statements

Statement ::=
  [Identifier§Label :] ··· UnlabelledStatement

UnlabelledStatement ::=
  EmptyStatement | Assignment | Block | SequentialControlStatement
  | RealTimeStatement | ConvertStatement | I/O-Statement

EmptyStatement ::=
  ;

Assignment ::=
  { { Name§Variable | Dereferentiation | StringSelection | Name§Structure }
  { := | = } } ··· Expression ;

Block ::=
  BEGIN
    [{ Declaration | Identification }···]
    [Statement ···]
  END [Identifier§Block] ;

SequentialControlStatement ::=
  IfStatement | CaseStatement | LoopStatement | ExitStatement
  | ProcedureCall | ReturnStatement | GoToStatement

IfStatement ::=
  IF Expression§OfType-BIT(1)
    THEN [Statement ···]
    [**ELSE** [Statement ···]]
  FIN ;

CaseStatement ::=
  StatementSelection1 | StatementSelection2

CaseStatement1 ::=
  CASE Expression§WithIntegerAsValue
    { **ALT** [Statement ···] } ···
    [**OUT** [Statement ···]]
  FIN ;

CaseStatement2 ::=
  CASE CaseIndex
    { **ALT** (CaseList) [Statement ···] } ···
    [**OUT** [Statement ···]]
  FIN ;

CaseIndex ::=
  Expression§WithValueOfType-FIXED-Or-CHAR(1)

CaseList ::=
  IndexSection [, IndexSection] ···

IndexSection::=
  Constant-FIXED-Expression [: Constant-FIXED-Expression]
  | CharacterStringConstant§OfLength1 [: CharacterStringConstant§OfLength1]

LoopStatement ::=
  [**FOR** Indicator§ControlVariable]
  [**FROM** Expression§InitialValue]
  [**BY** Expression§StepLength]
  [**TO** Expression§EndValue]
  [**WHILE** Expression§Condition]
  REPEAT
    [{ Declaration | Identification }···]
    [Statement ···]
  END [Identifier§Loop] ;

ExitStatement ::=
  EXIT [Identifier§BlockOrLoop] ;

ProcedureCall ::=
  [**CALL**] Name§SubprogramProcedure [ListOfActualParameter] ;

ReturnStatement ::=
  RETURN [(Expression)] ;

GoToStatement ::=
  GOTO Identifier§Label ;

RealTimeStatement ::=
  TaskControlStatement | TaskCoordination

| InterruptStatement | SignalStatement

TaskControlStatement ::=
TaskStarting | TaskTerminating
| TaskSuspending | TaskContinuing
| TaskResuming | TaskPreventing

TaskStarting ::=
[StartCondition] **ACTIVATE** Name§Task [PriorityExpression] ;

PriorityExpression ::=
{ **PRIORITY** | **PRIO** } Expression§PositiveInteger

StartCondition ::=
AT Expression§Time [Frequency]
| **AFTER** Expression§Duration [Frequency]
| **WHEN** Name§Interrupt [**AFTER** Expression§Duration] [Frequency]
| Frequency

Frequency ::=
ALL Expression§Duration
[{ **UNTIL** Expression§Time } | { **DURING** Expression§Duration }]

TaskTerminating ::=
TERMINATE [Name§Task] ;

TaskSuspending ::=
SUSPEND [Name§Task] ;

TaskContinuing ::=
[**AT** Expression§Time | **AFTER** Expression§Duration | **WHEN** Name§Interrupt]
CONTINUE [Name§Task] [PriorityAttribute] ;

TaskResuming ::=
{ **AT** Expression§Time | **AFTER** Expression§Duration | **WHEN** Name§Interrupt }
RESUME;

TaskPreventing ::=
PREVENT [Name§Task] ;

TaskCoordiniationStatement ::=
{ **REQUEST** | **RELEASE**} Name§Sema [, Name§Sema] ··· ;
| { **RESERVE** | **FREE** | **ENTER** | **LEAVE** } Name§Bolt [, Name§Bolt] ··· ;

InterruptStatement ::=

{ **ENABLE** | **DISABLE** | **TRIGGER** } Name§Interrupt ;

SchedulingSignalReaction ::=
ON Name§Signal { [**RST** (Name§ErrorVariable-FIXED)] :
SignalReaction | **RST** (Name§ErrorVariable-FIXED) };

SignalReaction::= UnlabeledStatement

InduceStatement ::=
INDUCE Name§Signal [**RST** (Expression§ErrorNumber-FIXED)] ;

ConvertStatement ::=
ConvertToStatement | ConvertFromStatement

ConvertToStatement ::=
CONVERT Expression [, Expression] ··· **TO** Name§CharacterStringVariable
[**BY** FormatOrPositionConvert [, FormatOrPositionConvert] ···] ;

ConvertFromStatement ::=
CONVERT Name§Variable [, Name§Variable] ··· **FROM** Expression§CharacterString
[**BY** FormatOrPositionConvert [, FormatOrPositionConvert] ···] ;

FormatOrPositionConvert ::=
[Factor] { Format | PositionConvert }
| Factor (FormatOrPositionConvert [, FormatOrPositionConvert] ···)

PositionConvert ::=
RST (Name§ErrorVariable-FIXED)
| **X** [(Expression)]
| { **POS** | **ADV** } (Expression)
| **SOP** (Name§PositionVariable-FIXED)

I/O-Statement ::=
OpenStatement | CloseStatement
| PutStatement | GetStatement
| WriteStatement | ReadStatement
| SendStatement | TakeStatement

OpenStatement ::=
OPEN Name§Dation [**BY** OpenParameter [, OpenParameter] ···] ;

OpenParameter ::=
RST (Name§ErrorVariable-FIXED)
| **IDF** ({ Name§CharacterStringVariable | CharacterStringConstant })
| { **OLD** | **NEW** | **ANY** }
| { **CAN** | **PRM** }

CloseStatement ::=
CLOSE Name§Dation [**BY** CloseParameter [, CloseParameter] ···] ;

CloseParameter ::=
RST (Name§ErrorVariable-FIXED)
| { **CAN** | **PRM** }

PutStatement ::=
PUT [{ Expression | Slice } [, { Expression | Slice }] ···]
TO Name§Dation [**BY** FormatOrPosition [, FormatOrPosition] ···] ;

GetStatement ::=
GET [{ Name | Slice } [, { Name | Slice }] ···]
FROM Name§Dation [**BY** FormatOrPosition [, FormatOrPosition] ···] ;

WriteStatement ::=
WRITE [{ Expression | Slice } [, { Expression | Slice }] ···]
TO Name§Dation [**BY** Position [, Position] ···] ;

ReadStatement ::=
READ [{ Name | Slice } [, { Name | Slice }] ···]
FROM Name§Dation [**BY** Position [, Position] ···] ;

SendStatement ::=
SEND [Expression] **TO** Name§Dation
[**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ···] ;

TakeStatement ::=
TAKE [Name] **FROM** Name§Dation
[**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ···] ;

RST-S-CTRL-Format ::=
RST (Name§ErrorVariable-FIXED)
| **S** (Name§Variable-FIXED)
| **CONTROL** (Expression [, Expression [, Expression]])

Slice ::=
Name§Array ([Index ,] ··· Index : Index)
/* Instead of the last array-index an index-slice is given to address the respective one-dimensional array-slice which is processed by the I/O in bottom-up order. */

FormatOrPosition ::=
[Factor] { Format | Position }
| Factor (FormatOrPosition [, FormatOrPosition] ···)

Factor ::=
(Expression§IntegerGreaterZero) | IntegerWithoutPrecision§GreaterZero

Format ::=
{ **F** | **E**} (Expression [, Expression [, Expression]])
| { **B** | **B1** | **B2** | **B3** | **B4** | **A** } [(Expression)]
| { **T** | **D** } (Expression [, Expression])
| **LIST**
| **R** (Identifier§Format)
| **S** (Name§LengthVariable-FIXED)

Position ::=
RST (Name§ErrorVariable-FIXED)
| { **X** | **SKIP** | **PAGE** } [(Expression)]
| { **POS** | **ADV** } (Expression [, Expression [, Expression]])
| { **COL** | **LINE** } (Expression)
| **SOP** (Name [, Name [, Name]] /* PositionVariables-FIXED */)

12.4 List of Keywords with Shortforms

The denotation behind the keyword refers to the paragraph where it is introduced.

ACTIVATE 9.2.2
AFTER 9.2.1
ALL 9.2.1
ALPHIC 10.2
ALT 7.2
AT 9.2.1

BASIC 10.2
BEGIN 4.4
BIT 5.4
BOLT 9.3.2
BY 7.3, 10.3

CALL 8.2
CASE 7.2
CHARACTER, **CHAR** 5.5
CLOCK 5.7
CLOSE 10.3
CONT 5.9
CONTINUE 9.2.5
CONTROL 10.2
CONVERT 10.2
CREATED 10.2
CYCLIC 10.2

DATION 10.2
DECLARE, **DCL** 4.1
DIM 10.2
DIRECT 10.2
DISABLE 9.4.2
DURATION, **DUR** 5.8
DURING 9.2.1

ELSE 7.1
ENABLE 9.4.2
END 4.4, 8.1, 9.1
ENTER 9.3.2
ENTRY 8.1
EXIT 7.5

FIN 7.1
FIXED 5.2
FLOAT 5.3
FOR 7.3
FORBACK 10.2
FORMAT 10.5.8
FORWARD 10.2
FREE 9.3.2
FROM 7.3, 10.4

GET 10.5
GLOBAL 4.1
GOTO 7.4

HRS 5.8

IDENTICAL, **IDENT** 4.1, 8.1
IF 7.1
IN 10.2
INDUCE 11
INITIAL, **INIT** 5.14
INOUT 10.2
INTERRUPT, **IRPT** 9.4.1
INV 5.14

LEAVE 9.3.2
LENGTH 5.6

MAIN 9.1
MAX 10.2
MIN 5.8
MODEND 4.1
MODULE 4.1

NIL 5.9
NOCYCL 10.2
NOSTREAM 10.2

ON 11
OPEN 10.3
OPERATOR 6.2
OUT 7.2, 10.2

PRECEDENCE 6.2, 12.3.4
PRESET 9.3.1
PREVENT 9.2.7
PRIORITY, **PRIO** 9.1
PROBLEM 4.3
PROCEDURE, **PROC** 8.1 8.1
PUT 10.5

READ 10.4
REF 5.9
RELEASE 9.3.1
REPEAT 7.3
REQUEST 9.3.1
RESERVE 9.3.2
RESUME 9.2.6
RETURN 8.2
RETURNS 8.1

SEC 5.8
SEMA 9.3.1
SEND 10.6
SIGNAL 11
SPECIFY, **SPC** 4.1
STREAM 10.2
STRUCT 5.11
SUSPEND 9.2.4
SYSTEM 4.2

TAKE 10.6
TASK 9.1
TERMINATE 9.2.3
TFU 10.2
THEN 7.1
TO 7.3, 10.4
TRIGGER 9.4.2
TYPE 5.12

UNTIL 9.2.1

WHEN 9.2.1
WHILE 7.3
WRITE 10.4

12.5 Other Word Symbols in PEARL

The denotation behind the word symbol refers to the paragraph where it is introduced.

A 10.5.3
ABS 6.1.1, table 6.1
ADV 10.4
AND 6.1.2
ANY 10.3
ATAN 6.1.1, table 6.3

B 10.5.4
B1 10.5.4
B2 10.5.4
B3 10.5.4
B4 10.5.4

CAN 10.3
CAT 6.1.2
COL 10.4
CONT 5.9
COS 6.1.1, table 6.3
CSHIFT 6.1.2

D 10.5.6
DATE 12.2.2

E 10.5.2
ENTIER 6.1.1
EQ 6.1.2
EXOR 6.1.2
EXP 6.1.1, table 6.3

F 10.5.1
FIT 6.1.2

GE 6.1.2
GT 6.1.2

IDF 10.3
IS 5.9
ISNT 5.9

LE 6.1.2
LINE 10.4
LIST 10.5.7
LN 6.1.1, table 6.3
LT 6.1.2
LWB 6.1.1, 6.1.2

NE 6.1.2
NEW 10.3
NOT 6.1.1, table 6.1
NOW 12.2.1

OLD 10.3
OR 6.1.2

PAGE 10.4
POS 10.4
PRM 10.3

R 10.5.8
REM 6.1.2
ROUND 6.1.1
RST 10.3, 10.8

S 10.5.3
SHIFT 6.1
SIGN 6.1.1, table 6.1
SIN 6.1.1, table 6.3
SIZEOF 6.1
SKIP 10.4
SOP 10.4
SQRT 6.1.1, table 6.3

T 10.5.5
TAN 6.1.1, table 6.3
TANH 6.1.1, table 6.3
TOBIT 6.1.1
TOCHAR 6.1.1
TOFIXED 6.1.1
TOFLOAT 6.1.1
TRY 9.3.1

UPB 6.1.1, 6.1.2

X 10.4