PDV Berichte

Projekt Prozeßlenkung mit DV-Anlagen

KFK-PDV 120

Basic PEARL
Language Description

1977

GESELLSCHAFT FÜR KERNFORSCHUNG MBH KARLSRUHE

PDV-Berichte

Die Gesellschaft für Kernforschung mbH koordiniert und betreut im Auftrag des Bundesministers für Forschung und Technologie das im Rahmen der Datenverarbeitungsprogramme der Bundesregierung geförderte Projekt Prozeßlenkung mit Datenverarbeitungsanlagen (PDV). Hierbei arbeitet sie eng mit Unternehmen der gewerblichen Wirtschaft und Einrichtungen der öffentlichen Hand zusammen. Als Projektträger gibt sie die Schriftenreihe PDV-Berichte heraus. Darin werden Entwicklungsunterlagen zur Verfügung gestellt, die einer raschen und breiteren Anwendung der Datenverarbeitung in der Prozeßlenkung dienen sollen.

Der vorliegende Bericht dokumentiert Kenntnisse und Ergebnisse, die im Projekt PDV gewonnen wurden.

Druck und Verbreitung:
Gesellschaft für Kernforschung mbH
7500 Karlsruhe 1, Postfach 3640
Printed in Western-Germany

KFK-PDV 120

Projekt Prozeßlenkung mit DV-Anlagen

Forschungsbericht KFK-PDV 120

Basic PEARL

Language Description

(Process and Experiment Automation Realtime Language)

291 Seiten
7 Abbildungen
7 Tabellen

Basic PEARL Language Description

Forword

This document is the official common language description of Basic PEARL. Basic PEARL is the subset of Full PEARL which was defined as the minimum that each PEARL implementation must contain. At the time being, it has been accepted and included in their implementations by the following firms: AEG, BBC, Dietz, GPP, Krupp-Atlas, MBP, and Siemens.

As for Full PEARL, the reader is asked to refer to Full PEARL Language Description KFK-PDV 130, Gesellschaft für Kernforschung mbH, Karlsruhe.

This Basic PEARL Language Description was assembled by Mr. P. Hruschka from GEI (Gesellschaft für Elektronische Informationsverarbeitung), Aachen, on behalf of Project PDV under Government Contract No. DV 5.505.

(For questions, contact Projekt PDV, Gesellschaft für Kernforschung mbH, P.O. Box 3640, 7500 Karlsruhe, Germany, c/o Dr. T. Martin.)

Table of Contents: Survey

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LANGUAGE DESCRIPTION: Table of Contents

LIST OF TABLES AND FIGURES

0.1 Method of Syntax Description

0.1.0 General

The syntax of Basic PEARL is described by a set of "production-rules".
These production-rules consist of "terminals" and "non-terminals" and some special characters, the meaning of which is discussed below.

Non-terminals are represented by a sequence of (small) letters, digits and hyphens, starting with a letter. No hyphen may immediately be followed by another one.

examples for non-terminals:

identifier
prec-3-operator
schedule-1

Terminals are represented by a sequence of capital-letters, digits and special characters, that are listed in section 0.1.1.2 (character-set).

examples for terminals:

TASK
(
IF
;
F
DATION

0.1.1 Production-rules

Each production-rule consists of

- a rule-name
- the character-sequence ::= to separate the rule-name from the rule-body
- a rule-body

There is a one-to-one correspondence between the rule-names and the non-terminals, i.e. each non-terminal identifies exactly one rule-body.
Beyond that each non-terminal must a least once appear in another rule-body, except for one non-terminal, which is Basic PEARL is

basic-pearl-program

(refer to section 1.1.1).

A rule-body consists of a sequence of non-terminals and terminals and some special characters, which are used for convenience of denotation.
These special characters are:

/	used to denote alternatives
{ }	used for alternatives, too, or to enclose sequences
[ ]	used to enclose optional parts
. ...	used for repetition or repetition with delimiter

- alternatives:

  alternatives within a rule-body are either separated by / or they are aligned vertically and enclosed in braces { , } .

  example:

  A::=
  B / C / D

  A produces B or C or D. The same production-rule may be denoted as

  A::=
  { B
  { C
  { D

- options:

  if parts of a rule-body may be produced or not, these parts are enclosed in square brackets [,] .

  example:

  A::=
  B [C] D

  A produces either BCD or BD only.
  C is said to be optional.

- repetition:

  repetition of sequences is achieved by ••• following the sequence.

  example:

  A::=
  B•••

  This means, A produces B, or BB, or BBB, ...

- repetition with delimiter:

  often it is necessary to repeat a sequence, separating the single elements by a special delimiter.

  A::=

  $\{ D \cdot B \cdots \}$

  produces B or a sequence of B separated by D:

  BDB
  BDBDB
  BDBDBDB
  ⋮

To obtain a (syntactically correct) Basic PEARL program one has to start with the production-rule for 'basic-pearl-program', always replacing non-terminals by any of the alternatives listed in its production-rule until this results in a terminal sequence.

O.1.2 Character-set

The character-set of Basic PEARL is given by the following production-rule:

character-set::=

{ *letter* | *digit* | *special-character* }

letter::=

A / B / C / D / E / F /
G / H / I / J / K / L /
M / N / O / P / Q / R /
S / T / U / V / W / X /
Y / Z

digit::=

0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

For the representation of 'bit-strings' the following three subsets of the character-set are used:

binary-digit::=

0 / 1

octal-digit::=

0 / 1 / 2 / 3 / 4 / 5 / 6 / 7

hexadecimal-digit::=

0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 /
8 / 9 / A / B / C / D / E / F

In Basic PEARL the following special-characters are admitted:

special-character::=

␣	*(blank or space)*
+	*(plus-sign)*
-	*(minus-sign or hyphen)*
*	*(asterisk)*
/	*(slash)*
(	*(left parenthesis)*
)	*(right parenthesis)*
:	*(colon)*
.	*(period)*
,	*(comma)*
;	*(semicolon)*
=	*(equal-sign)*
<	*(left-angle-bracket)*
>	*(right-angle-bracket)*
'	*(single quote)*
[	*(left square bracket)*
]	*(right square bracket)*

Note: the characters /, [,] are underscored to distinguish them from the ones used within production rules.

0.1.3 Spaces, Comments and Keywords

On constructing Basic PEARL programs some more rules have to be obeyed:

- identifier, keywords and constant-denotations of objects of type FIXED, FLOAT, BIT, CHAR have to be separated by at least one blank.

  example:
  ```
  BEGIN␣IF
  7.53␣SEC
  FOR I FROMSTART BY....
           ↑
  ```
  blank missing between keyword and identifier!

- blanks may be inserted anywhere between terminals except within 'identifiers' and string-constant-denotations.
  (Note: in character-strings blanks may appear, but then they are relevant members of the string, and not separators!)

  Those sequences of special-characters denoting one terminal symbol, as for example := or // or →·, may not be broken apart, too.

- comments may be inserted anywhere where blanks are legal. Comments are bracketed by the special-character-combinations

  /* and */

  Within these brackets any text may appear except */, which would terminate the comment.
  Comments may not be nested.

  example: /* THIS IS A COMMENT */

Some of the terminal symbols are reserved keywords of Basic PEARL, i.e. they may not be used as identifiers. The following table lists all the keywords of Basic PEARL and the abbreviations (denoted in brackets) in alphabetical order.

(Note: In order to write Basic PEARL programs comptatible to Full PEARL one has to obey some more reserved keywords. They can be found in the Full PEARL Reference Manual).

ACTIVATE	DATION
AFTER	DECLARE (DCL)
ALL	DIRECT
ALPHIC	DISABLE
ALT	DURATION (DUR)
AT	DURING
BASIC	ELSE
BEGIN	ENABLE
BIT	END
BY	ENTRY
	EVERY
CALL	
CASE	FIN
CHARACTER (CHAR)	FIXED
CLOCK	FLOAT
CLOSE	FOR
CONTINUE	FORBACK
CONTROL	FORMAT
CREATED	FORWARD
CYCLIC	FROM

table 1: reserved keywords

GET
GLOBAL
GOTO

HRS

IDENT
IF
IN
INDUCE
INIT
INOUT
INTERRUPT (IRPT)
INV

LENGTH

MAX
MIN
MODEND
MODULE

NOCYCL
NOSTREAM

ON
OPEN
OUT

PRESET
PREVENT
PRIORITY (PRIO)
PROBLEM
PROCEDURE (PROC)
PUT

READ
REENT
RELEASE
REPEAT
REQUEST
RESIDENT
RESUME
RETURN
RETURNS

SEC
SEMA
SEND
SIGNAL
SPECIFY (SPC)
STREAM
STRUCT
SUSPEND
SYSTEM

TAKE
TASK
TERMINATE
TFU
THEN
TO

UNTIL

WHEN
WHILE
WRITE

table 1, continued: reserved keywords

0.2 Method of Semantics Description

The semantics of syntactically correct Basic PEARL constructs is explained in plain English. For ease of understanding examples and illustrations are added.

Part 1 of this Language Description explains the static features of Basic PEARL objects, i.e. what attributes they may be given, what restrictions have to be obeyed etc.

The dynamics of Basic PEARL programs are detailed in part 2 in form of operations on objects.

Semantics is only detailed as far as this is possible on language level, i.e. no implementation dependent features are explained in this report. Such locations still needing supplement are either marked by a reference to a special "implementation-handbook" or by referring to the "system" (meaning a virtual machine executing the Basic PEARL code, without detailing the actions of this machine precisely).

1. OBJECTS

1.0 General

1.0.0 Attributes of Objects

Objects consist of a location and a content. The content of an object can be a value or a prescription. (The contents of a FIXED-variable, for instance is a value of an integer number, the contents of a procedure is a prescription for the execution of an algorithm). Objects the content of which is a prescription are called "executable objects".

Objects may be "identifiable" or not. Identifiable objects must be equipped with a name, a socalled "identifier".
The usual way to supply objects with an identifier is a declaration (cf. section 1.0.2). But there are some objects with predefined names (and attributes), for example 'controls' (refer to section 1.3.2) or the 'system-device-identificators' in the SYSTEM-division (see section 1.3.1.2.2), that are already known to the "system" - and therefore they don't have to be declared.

An 'identifier' consists of a sequence of letters and digits, starting with a letter.

identifier::=

letter [{ *letter* / *digit* } ...]

Within its scope (refer to section 2.1.2) it must unequivocally identify one object.

Objects without identifier are called 'constant-denotations'. Such objects always represent a value, their location is not known to the programmer.

example: 3.14 may be used in a program to denote the mathematical entity "real number 3.14".

The attributes of an object serve to

- characterize the content and to
- characterize the access-right to the object.

The first group of attributes defines how the content is to be interpreted. So, for example FIXED means, the content is the value of an integer number, ALPHIC denotes, that a 'dation' may contain 'alphic symbols' etc.

In the following these attributes are often called the "type" of an object.
Examples for the second category are FORWARD, meaning an object of type 'dation' can only be accessed sequentially in positive direction - or INV, denoting "invariability". Access-right attributes given in a declaration may not be extended in specifications; they may only be restricted.

In general, the relations between these attributes (i.e. which attribute is more restricted or more extended) are set up in the sections where the corresponding objects are detailed. One attribute, however, is explained in the following section, since it applies to a variety of types: the INV-attribute.

1.0.1 Variability

"Variability" of an object means, that it may be used as left-hand-side of an assignation (refer to section 2.2.1.1), i.e. different values may be assigned to this object by the programmer.
For objects of basic type variability is default, invariability is indicated by the attribute "INV".
Objects having the INV-attribute are in the following often called 'constants', basic objects without the INV-attribute are called 'variables'.

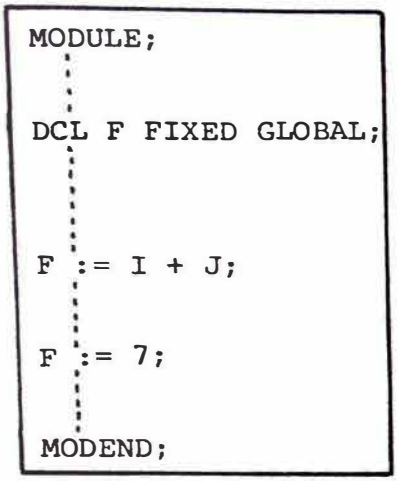
example:

```
DCL PI INV FLOAT INIT (3.14);
                    PI denotes a FLOAT-constant

DCL I FIXED;
                    I denotes a FIXED-variable
```

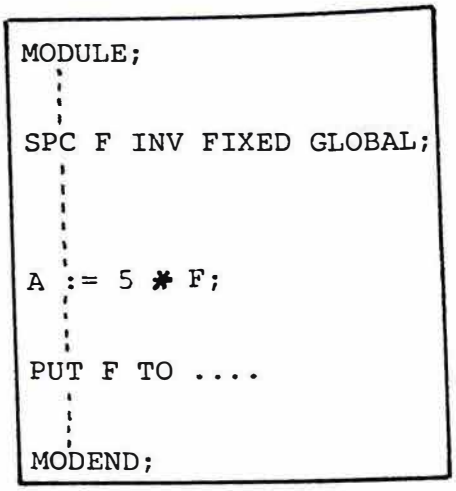
Some other Basic PEARL objects are implicitely considered of having INV-access since no assignation is defined for these objects, for example semaphores or procedures.
Variability is defined with respect to the assignation-operation, not with respect to operations in general. (For example, a semaphore, although considered as INV, will of course be changed under a REQUEST-operation).

The INV-attribute may also be used to restrict access to a global object within a specification to "read-only".

example:



F is declared as FIXED-variable, therefore F may be used as left-hand-side in assignations.



Access to F has been restricted in the specification. Therefore the value of F must not be changed in this module.

1.0.2 Declarations and Specifications

1.0.2.0 General

As already mentioned in the previous section attaching attributes to objects can either be done in a declaration or a specification.
A declaration creates an object and attaches attributes to it. Any identifiable Basic PEARL object - with the exception of labels (refer to section 1.2.4) - must be declared before it can be accessed (used in operations).

There are different syntactical forms for declarations:

- *global-declaration*
- *local-identifier-declaration*
- *r-format-declaration*
- *procedure-declaration*
- *task-declaration*
- *length-declaration*
- *precision-declaration*
- *label-declaration*

Except for 'global-declaration' they will be detailed in the corresponding sections.
The 'length-declaration' and the 'precision-declaration' do not create objects, but preset system-values needed as defaults (cf. section 1.2.1.1.0 and 1.2.1.2.0).

Specifications can only be applied to already existing objects. They are either used to make these objects known in another module (or division, resp.) and/or to restrict the access-attributes of these objects.

(Note: Extending or changing attributes is not possible in specifications, only restriction!)

In Basic PEARL specifications can only be attached at module-level, therefore they are syntactically covered by

global-specification

which is detailed in section 1.0.2.2.

1.0.2.1 Global Declaration

A variety of Basic PEARL objects is created by a 'global-declaration'. The general syntactical form is given by:

global-declaration::=

{*DECLARE* | *DCL*}

{, *one-identifier-or-list*

global-decl-attributes ...}

If several objects are to be created with equal attributes ('global-decl-attributes') their identifiers may be combined to a list:

one-identifier-or-list::=

{*identifier* | ({, *identifier* ...})}

example: creating four FIXED-variables can be achieved by

```
DCL (I,J,K,L) FIXED;
```

An equivalent denotation is

```
DCL I FIXED, J FIXED,
    K FIXED, L FIXED;
```

'global-decl-attributes' lists the attributes the object(s) is (are) to be given and provides constructions for initialization.

global-decl-attributes::=

$$\left\{\begin{array}{lll} [\textit{bound-list}]\ \textit{local-mode} & [\textit{RESIDENT}]\,[\textit{GLOBAL}] & \textit{initial} \\ \textit{SEMA} & [\textit{RESIDENT}]\,[\textit{GLOBAL}] & \textit{presetting} \\ \textit{dation-attr} & [\textit{RESIDENT}]\,[\textit{GLOBAL}] & \textit{dataway-construction} \end{array}\right\}$$

The first alternative is to denote all basic data-types (detailed in section 1.2.1), and compound objects constructed of basic objects.

local-mode::=

$$[\ \textit{INV}\ ]\quad \left\{\begin{array}{l} \textit{simple-mode} \\ \textit{structure-mode} \end{array}\right\}$$

The second alternative denotes the attributes for semaphores, the third is provided for user-defined data-stations.

The attribute RESIDENT has no semantics in the usual sense. It just indicates that the corresponding object is considered to be used frequently, therefore it would be advantageous to keep it in a fast-access part of memory. It may also be used as a hint to the compiler to do some optimization.

The attribute GLOBAL is detailed in section 1.1.3 and 1.1.4.

An initialization presets the value of an object at the time it is created. Syntactically these are three different forms:

initial::=

[*INIT*
({ , }... { [+]/- } *constant-denotation* ...)]

presetting::=

[*PRESET*
({ , }... *simple-integer-constant-denotation* ...)]

dataway-construction::=

CREATED
(dation-identifier [*integer-in-brackets*] *)*

For some objects initialization is necessary since it is the only chance to supply them with a value.
The following table lists all the objects for which initialization is possible, necessary, impossible or defaulted - and the way how this is achieved.

initialization is:	for objects of TYPE	achieved by:
possible	FIXED FLOAT BIT CHARACTER CLOCK DURATION SEMA DATION	'initial' 'presetting' 'dataway-construction'
necessary	INV FIXED INV FLOAT INV BIT INV CHARACTER INV CLOCK INV DURATION DATION	'initial' 'dataway-construction'
impossible	for all compound objects, i.e. arrays and structures	---------
defaulted	SEMA	omitting 'presetting', defaulted value is zero

table 2: initialization

Within one 'global-declaration' objects of different types may be declared.

example:

```
DECLARE F INV FLOAT INIT(7.5) GLOBAL,
        CL CLOCK, I FIXED(16) INIT(-2(16)),
        (S1,S2) (5) CHAR(10) RESIDENT GLOBAL;
```

This 'global-declaration' is equivalent to the following sequence of 'global-declaration's:

```
DCL F INV FLOAT INIT(7.5) GLOBAL;
DCL CL CLOCK;
DCL I FIXED(16) INIT(-2(16));
DCL S1 (5) CHAR(10) RESIDENT GLOBAL;
DCL S2 (5) CHAR(10) RESIDENT GLOBAL;
```

Note: In the following sections, where the objects are introduced and explained, the respective complete syntax for the declaration (and specification) for the object is given.
This is done to provide a convenient way for reading the sections and to establish an easily comprehensible connection between syntax and examples.
Nevertheless such declarations may - at the programmers pleasure - be combined to 'global-declaration's (as indicated in the example above).

1.0.2.2 Global Specification

In Basic PEARL specifications are only possible at module-level. The general syntactical form is given by:

global-specification::=

{ *SPECIFY* | *SPC* }
{ , *one-identifier-or-list*
global-spec-attributes [*GLOBAL*] ... }

As for declarations combining identifiers to lists is possible, if the corresponding objects have equal attributes.
'global-spec-attributes' provides some more alternatives than the corresponding 'global-decl-attributes', since realtime-objects and procedures are included in this syntactical construction, too.

global-spec-attributes::=

{ [([,][,])] *local-mode*
SEMA
[( )] *dation-spec-attr*
TASK
irpt-or-signal-mode
procedure-mode [*RESIDENT*][*REENT*] }

Within specifications all the bound-lists are denoted as empty lists, just indicating the (number of) dimensions. This applies to any kind of arrays (e.g. arrays of simple objects, data-stations and interrupt- or signal-arrays, too).

irpt-or-signal-mode::=

$$[(\,)] \quad \left\{ \begin{array}{l} \textit{INTERRUPT} \\ \textit{IRPT} \\ \textit{SIGNAL} \end{array} \right\}$$

For the specification of procedures a different keyword is used:

procedure-mode::=

$$\textit{ENTRY} \quad [\,(\{,\ \textit{parameter-mode}\ \cdots\})\,] \quad [\,\textit{result-attribute}\,]$$

example:

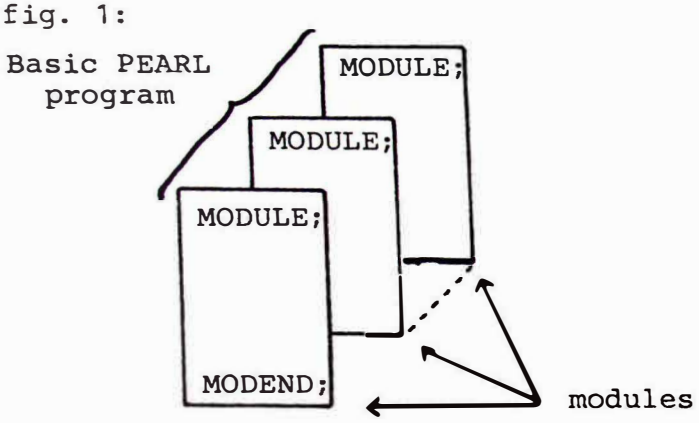
```
SPC I INV FIXED GLOBAL,
    A(,) CHAR(10) GLOBAL,
    T1 TASK GLOBAL,
    ALARM( ) IRPT, DONE SIGNAL,
    F1 ENTRY RETURNS (FIXED) REENT GLOBAL;
```

1.1 Structural Objects

1.1.0 Program Structure

In Basic PEARL a program is composed of one or several "modules".
Modules are the units of compilation of a Basic PEARL program.

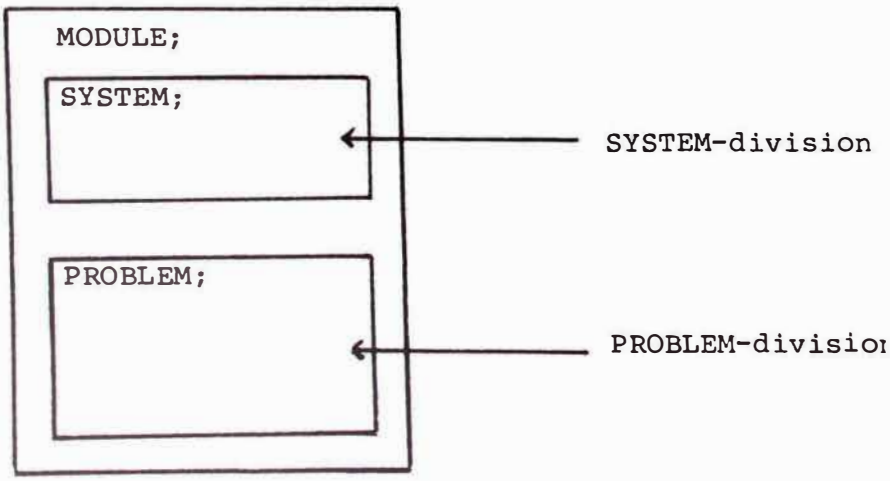
fig. 1:



A module may consist of one or two "divisions".
There are two kinds of divisions:

- SYSTEM-division
- PROBLEM-division

fig. 2: module



The SYSTEM-division describes the system-dependent parts of the program, i.e. the configuration of the devices used and the connections between them.

The PROBLEM-division contains the problem-specific parts of the program, i.e. the formulation of the algorithms used to solve the automation problem.

Modules and divisions will be briefly discussed in the following section. There the block-concept is introduced, too.

1.1.1 Modules, Divisions and Blocks

A Basic PEARL program consists of declarations and specifications partitioned into units called "modules".

basic-pearl-program::=

module ...

A module may contain one or two divisions.

module::=

MODULE;

$$\left\{\begin{matrix} \textit{system-division} & [\ \textit{problem-division}\ ] \\ \textit{problem-division} & \end{matrix}\right\}$$

MODEND;

The syntax shows, that a module may contain either

- only a system-division, or
- only a problem-division, or
- a system-division and a problem-division

Note: In a Basic PEARL program only one module may contain a system-division.
At least one module must contain a problem-division containing a declaration of a global task.
(cf. section 2.1.1)

The goals of the system-division and the specifications it may contain are further explained in section 1.3.1.2.1.
A problem-division consists of a set of declarations and specifications.

problem-division::=

PROBLEM;
declarations-and-specifications

Within 'declarations-and-specifications' the following order must be obeyed:

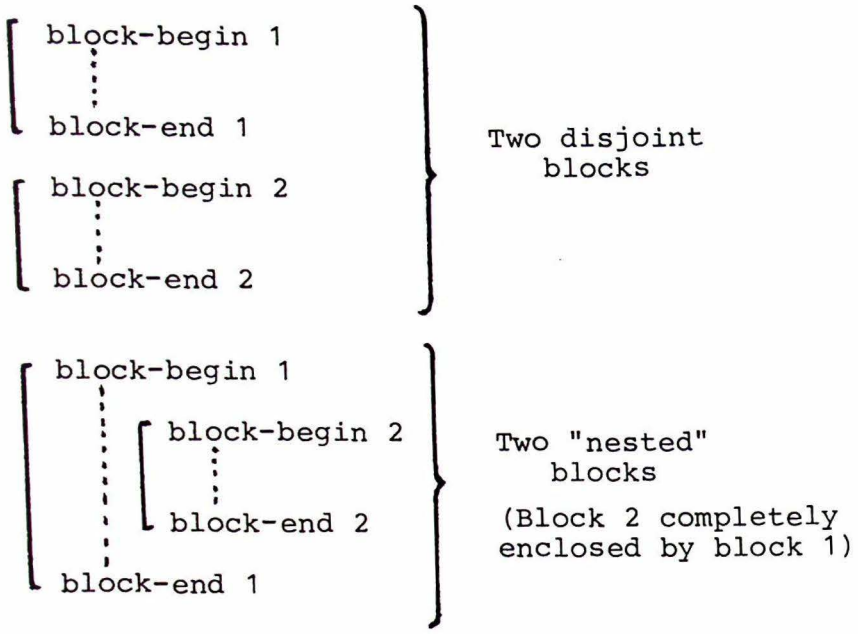
declarations-and-specifications::=

[{ *length-declaration;* / *precision-declaration;* } ...]
[*global-specification;*] ...
[*global-declaration;*] ...
[*r-format-declaration;*] ...
[*procedure-declaration;*] ...
[*task-declaration;*] ...

A further structuring of a problem-division is achieved by introducing "blocks".
Blocks are groups of declarations and/or statements. Declarations, if present, have to precede statements.

Two blocks may either be disjoint or one is completely enclosed by the other.

example:



In Basic PEARL the following types of blocks are possible:

- procedures
- tasks
- begin-blocks
- repeat-statements

In a certain respect modules or divisions, resp., may be viewed as blocks. They may only contain declarations (or specifications, resp.), but no statements.

In Basic PEARL the following rules have to be obeyed:

- at module-level (within a problem-division) only procedure- or task-blocks may be attached.

- procedures and tasks are always disjoint blocks, they may not be nested.
  (i.e. one must not declare a procedure within a task, or a procedure within a procedure etc. !)

- note the order of procedures and tasks given in the syntax: procedures always precede tasks!

- begin-blocks may only appear within procedures or tasks (not at module-level since they are no declarations). Begin-blocks will be detailed below.

The particularities of procedures and tasks will be discussed in the respective sections.

The following figure is to elucidate the rules:

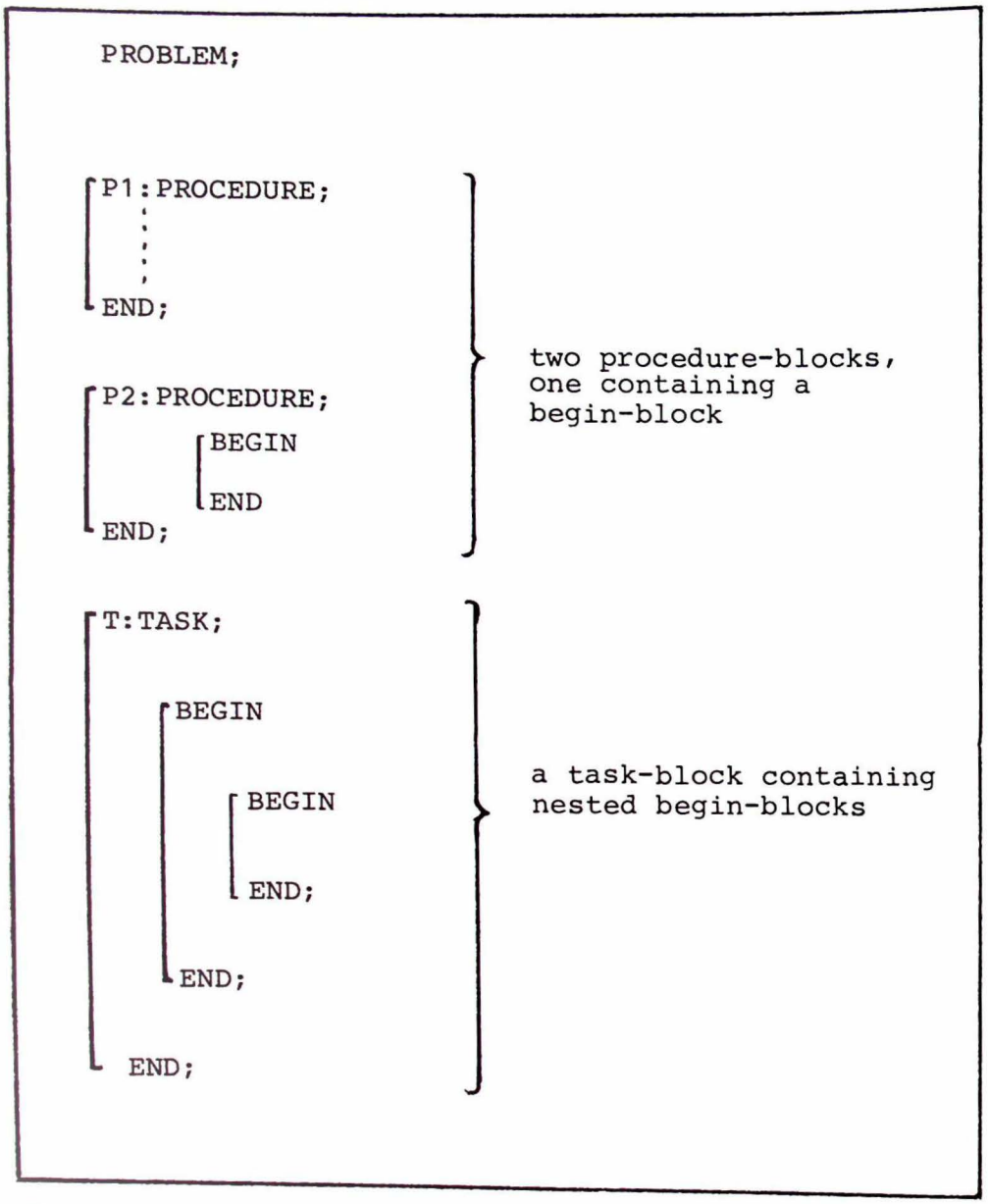


fig. 3: Block-structures

1.1.2 Begin-blocks

Syntactically, a 'begin-block' is a special form of a 'statement' (refer to section 2.0). This implies - as already mentioned in the last section - that begin-blocks may only appear within procedures or tasks (and not at module-level).

begin-block::=

BEGIN [;]
block-tail

block-tail::=

[*local-identifier-declaration;*]...
[*statement*]...
END

'statement' within the block-tail may, of course, again be a begin-block, and so nesting is achieved.
Begin-blocks are used to structure procedure- and task-blocks and to control the scope and the lifetime of objects.
This is explained in section 2.1.
A begin-block is entered as soon as its block-begin is executed. It can be left in two ways: either by executing its block-end or by transfer of control to a statement that is not part of it.
'local-identifier-declaration' serves to declare objects local to the block.
In Basic PEARL only simple objects, structures and arrays may be declared at this level.

local-identifier-declaration::=

{ *DECLARE* | *DCL* }

{ , • *one-identifier-or-list*

[*bound-list*]

local-mode

initial ••• }

examples for local declarations:

```
DCL A INV FIXED INIT(7),
    B(2,5) FLOAT;
DECLARE X STRUCT [M CHAR(5), N FIXED] ;
```

1.1.3 Inter-module communication

All objects declared at module-level may usually be accessed only within the module they are declared.
But Basic PEARL provides means to extend the scope of objects. This is achieved through the global-attribute "GLOBAL".
So, if one object (e.g. a variable or a procedure) is to be used in more than one module it must be declared with the global-attribute.
Then it can be specified in other modules.
(The usual way would be to declare this object in a more global block, enclosing all the blocks, where it is used. Since this is not possible for modules - because there is no enclosing "block" - this special mechanism for communication has been introduced).

example:

```
MODULE;
PROBLEM;
DCL F FIXED GLOBAL;
   :
   :
   :
/* here F may be used */
   :
   :
MODEND;
```

```
MODULE;
PROBLEM;
SPC F FIXED GLOBAL;
   :
   :
   :
/* here F may be used,
   too */
   :
   :
MODEND;
```

1.1.4 Inter-division communication

A similar mechanism as for modules is used between divisions.
All the devices introduced in the system-division are implicitly GLOBAL. They must be specified in problem-divisions, even if they are in the same module.
In this case no "global-attribute" is required.

example:

```
SYSTEM;
   :
   :
   TEMP: ANIN(8)   -> ADC(2);

   ALARM: ITR(3)   -> ;
    :
    :

PROBLEM;
   :
   :
   SPC TEMP DATION IN BIT(8);

   SPC ALARM INTERRUPT;
    :
    :
    :
```

1.2 Algorithmic Objects

1.2.0 General

This chapter deals with those Basic PEARL objects that are commonly considered as "algorithmic". Section 1.2.1 describes the six basic data-types, grouped into numbers, strings and times.

The declaration and specification of "procedures" are detailed in section 1.2.2. It is also explained what attributes procedures, their parameters and results may be given.
The basic data-types introduced in 1.2.1 may be combined, forming new types of objects. These "compound objects" are developped in section 1.2.3.
The final section 1.2.4 introduces "labels".

1.2.1 Basic Data Types

1.2.1.0 General

This section deals with the simple algorithmic objects used in Basic PEARL.

simple-mode::=

{ *FIXED* [*precision*]
| *FLOAT* [*precision*]
| {*CHARACTER/CHAR*} [*length*]
| *BIT* [*length*]
| *CLOCK*
| *DURATION/DUR* }

For each of these basic-types a constant-denotation exists:

constant-denotation::=

{ *integer-constant-denotation*
| *real-constant-denotation*
| *character-string-constant-denotation*
| *bit-string-constant-denotation*
| *clock-constant-denotation*
| *duration-constant-denotation* }

They are detailed in the following sections.

1.2.1.1 Numbers

1.2.1.1.0 Precision

Basic PEARL provides two data-types for the representation of numbers, that are objects of type FIXED and of type FLOAT. Associated with each FIXED- and FLOAT-object is an integer-constant called "precision".

precision::=

integer-in-brackets

integer-in-brackets::=

(simple-integer-constant-denotation)

This 'precision' defines a relation between the "internal" representation of FIXED- and FLOAT-objects and the "external", i.e. mathematical numbers to which they correspond.

For FIXED-objects precision defines a range:
Each mathematical integer with an absolute value from

$$\emptyset \text{ to } 2^{|\text{precision}|-1}$$

has an exact internal representation as FIXED-object.

For FLOAT-objects precision defines how exact the internal representation of a real number is.

The deviation between a mathematical number r and its internal representation is given by:

$$\text{deviation} < r / 2 \; |\text{precision}|$$

There are different ways of supplying the precision:

- for user-defined objects it may be given within the declaration or
  in a precision-declaration at the beginning of a problem-division.
- for constant-denotation precision can be denoted as suffix in brackets.

  example: 10 (16) denotes the integer number 10 with precision 16.

If no precision is given, neither direct, nor in a precision-declaration, it is defaulted to an implementation-dependent value.

precision-declaration::=

LENGTH { *FIXED* / *FLOAT* } *precision*

This 'precision-declaration' is valid for all FIXED- or FLOAT-objects respectively, declared in the corresponding problem-division without 'precision'.

Within specifications precision may not be changed. If it is supplied it has to be in accordance with the declaration. If it is omitted it is defaulted to the precision valid in the problem-division the specification is attached to.

In the following section the two sorts of objects, their declarations and constant-denotations are detailed.

1.2.1.1.1 FIXED

Objects of type FIXED represent integer numbers. User-defined-objects of type FIXED are introduced by the following declaration:

integer-declaration::=

{*DECLARE* | *DCL*} *one-identifier-or-list*

[*INV*] *FIXED* [*precision*]

[*RESIDENT*] [*GLOBAL*]

[*INIT* (*{, integer-constant-denotation ...}*)]

The corresponding constant-denotation consists of a sequence of decimal or binary digits.

integer-constant-denotation::=

simple-integer-constant-denotation [*precision*]

simple-integer-constant-denotation::=

{*digit* ... | *binary-digit* ... *B*}

examples:

DCL A FIXED (10);	A is an integer-variable, that may accept values: $\emptyset \leq \lvert A \rvert \leq 1023$
DCL I INV FIXED INIT (17);	I is an integer constant with the value 17
101B	This is an integer constant representing the decimal number 5

1.2.1.1.2 FLOAT

Objects of type FLOAT are used to represent rational numbers. They may be declared as follows:

float-declaration::=

{*DECLARE* / *DCL*} *one-identifier-or-list*
[*INV*] *FLOAT* [*precision*]
[*RESIDENT*] [*GLOBAL*]
[*INIT* ({,}... [+]/- *real-constant-denotation*...)]

The constant denotation for float-objects is given by the following syntax:

real-constant-denotation::=

simple-real-constant-denotation [*precision*]

simple-real-constant-denotation::=

{{[*digit*...]. *digit* / *digit*... .} [*exponent-part*] / *digit*... *exponent-part*}

The exponent is denoted in the following way:

exponent-part::=

E {[+]/-} [*digit*] *digit*

examples:

```
DCL A FLOAT (16);          A is a FLOAT-variable with
                           precision 16

DCL C INV FLOAT INIT (0.87)
                           C is a FLOAT-constant repre-
                           senting the value 0.87
```

Some examples for constant denotations, all of them denoting the same values as C above:

```
8.70E-1
 .87
0.0087E02
 .0870E+1
```

1.2.1.2 Strings

1.2.1.2.0 Length

In Basic PEARL there are two kinds of strings:

character-strings and bit-strings

Both of them are characterized by an integer number called "length", denoting the number of characters or bits, respectively, the object is able to contain.

As the precision for numbers length is denoted as:

length::=
integer-in-brackets

'Length' may be defined by a 'length-declaration' at the beginning of a problem-division (cf. section 1.1.1).

length-declaration::=

LENGTH { *BIT* / *CHARACTER/CHAR* } *length*

This 'length-declaration' is valid for all strings in this problem-division, that do not explicitly contain the length-attribute in their declaration or specification.

For user-defined strings, the length must be chosen such that it is not exceeded by any string-expression assigned to them.
This would be an error.

If 'length' is neither supplied in the declaration (or specification), nor a length-declaration is provided, it is defaulted to 1.

Within a specification, 'length', if supplied, must be the same as the one (given explicitely or implicitely) in corresponding declaration.

For string-constant-denotations length is given by the actual number of characters or bits. This is detailed in the following sections.

1.2.1.2.1 CHARACTER

Objects of type CHARACTER are used to represent "alphic symbols", i.e. all letters, digits and special characters of an implementation (refer to section 1.3.1.1.1.2).

They are declared in the following way:

character-declaration::=

{*DECLARE* / *DCL*} *one-identifier-or-list*
[*INV*] {*CHARACTER/CHAR*} [*length*]
[*RESIDENT*] [*GLOBAL*]
[*INIT* ({,'*character-string-constant-denotation*'...})]

'character-string-constant-denotation' consists of a sequence of symbols from the character-set enclosed in (single) quotation-marks:

character-string-constant-denotation::=

' [*string-character*...]'

string-character::=

letter / digit / blank /
*+ / - / * / _ / , /*
(/) / : / . / ; /
= / < / > / '' / [/]

As one can see from the syntax above, single quotes (') within a string have to be denoted as '' to distinguish them from the delimiters of a string.

The length of a character-string is given by the number of string-characters within the delimiting quotes, whereby '' count as one character.

examples:

```
'STRING'   is a string of type CHARACTER(6)
'EXAMPLE␣''2''' is a string of type CHARACTER(11)
''         is a string of type CHARACTER(0);
           it is called empty string.
```

Character-strings may also contain other alphanumeric symbols, which are not part of the Basic PEARL character-set. Those additional symbols are listed in the implementation handbook.

1.2.1.2.2 BIT

Objects of type BIT represent bit-strings.
They are declared as follows:

bit-declaration::=

{*DECLARE* / *DCL*} *one-identifier-or-list*

[*INV*] *BIT* [*length*]

[*RESIDENT*] [*GLOBAL*]

[*INIT* (*{,'bit-string-constant-denotation ···}*)]

In Basic PEARL they are three different ways to denote bit-string-constants:

bit-string-constant-denotation::=

{
' *{binary-digit ···}* ' *{B / B1}*
' *{octal-digit ···}* ' *B3*
' *{hexadecimal-digit ···}* ' *B4*
}

The length of a bit-string is given by the number of digits enclosed by ' multiplied with the integer following B (which is defaulted to 1, if B stands alone).

The following tables show the relations between binary-, octal- and hexadecimal-digits:

octal-digit	binary-digit
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

table 3: binary-octal conversion

hexadecimal-digit	binary-digit
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

table 4: binary-hexadecimal conversion

examples:

'101'B	is a bit-string of type BIT(3)
'101'B4	is a bit-string of type BIT (12)
'AE'B4	is a bit-string of type BIT (8)
'712'B3	is a bit-string of type BIT (9)

1.2.1.3 Times

1.2.1.3.0 General

In a realtime-language features for time-handling are of special importance.
Basic PEARL provides two data-types to deal with times, i.e. objects of type CLOCK and DURATION. They are mainly used for scheduling operations (cf. section 2.4.3.0.1). Other operations for objects of these types are listed in section 2.2.1.2.

1.2.1.3.1 CLOCK

Points on a time-scale are represented by Basic PEARL objects of type CLOCK.
They are declared in the following way:

clock-declaration::=

{*DECLARE* | *DCL*} *one-identifier-or-list*
[*INV*] *CLOCK*
[*RESIDENT*] [*GLOBAL*]
[*INIT* (*clock-constant-denotation* {, *clock-constant-denotation* ...})]

The corresponding constant-denotation consists of three numbers, seperated by colons.

clock-constant-denotation::=

simple-integer-constant-denotation:
simple-integer-constant-denotation:
{*simple-integer-constant-denotation* | *simple-real-constant-denotation*}

The three numbers denote hours, minutes and seconds. Minutes and seconds have to be within the range between 0 and 60, hours are always interpreted modulo 24.

Examples:

13:2:20 13:02:20.0 37:2:20.0	all three constant-denotations mean 2 minutes and 20 seconds past 1 o'clock p.m.
7:63:10	is an illegal example, since the number denoting the minutes is not within the range.

1.2.1.3.2 DURATION

Time-intervals may be handled through objects of type DURATION.
The declaration for these objects looks like:

duration-declaration::=

{*DECLARE* / *DCL*} *one-identifier-or-list*

[*INV*] {*DURATION* / *DUR*}

[*RESIDENT*] [*GLOBAL*]

[*INIT* (*{,' duration-constant-denotation ...}*)]

Duration-constants are denoted as follows:

duration-constant-denotation::=

{*hours* [*minutes*] [*seconds*] / *minutes* [*seconds*] / *seconds*}

hours::=

simple-integer-constant-denotation HRS

minutes::=

simple-integer-constant-denotation MIN

seconds::=

{*simple-integer-constant-denotation* / *simple-real-constant-denotation*} *SEC*

For the numbers contained in 'duration-constant-denotation' there are no particular restrictions.

examples:

2 HRS 10 MIN 26.7 SEC
1 HRS 10 SEC
87 MIN
320.5 SEC

1.2.2 Procedures

1.2.2.0 General

Procedures are main elements for structuring programs. A procedure is a set of declarations and statements combined to a block (refer to section 1.1.1.). This block can be executed by a special operation, a "call". Execution of procedures is detailed in section 2.2.2.

This section explains, how objects of type "procedure" are created, and what attributes they may be given. As any other Basic PEARL objects a procedure has to be declared prior to its use.

procedure-declaration::=

procedure-identifier:

{ *PROCEDURE* | *PROC* }

[({ , *one-identifier-or-list* *parameter-mode* ... })]

[*result-attribute*]
[*RESIDENT*] [*REENT*] [*GLOBAL*];
block-tail

'procedure-identifier' denotes the name of the procedure.

procedure-identifier::=

identifier

The construction following the keyword "PROCEDURE" (or "PROC") denotes the list of formal parameters. This is detailed in the following section.

'result-attribute' must only be supplied with function-procedures, i.e. with procedures returning a result to the point where they are called from. Function procedures are explained in section 1.2.2.2.
As for other objects some optional attributes may be added. One of them, the attribute 'REENT' may only be used with objects of type 'procedure', therefore it is detailed in section 1.2.2.3.

Note the sequence of the three keywords

RESIDENT REENT GLOBAL

which in Basic PEARL is compulsory!

'block-tail' contains the local declarations of the procedure and the sequence of statements that make up the procedure-body.

To use a procedure in another module it must be specified. This is done in the following way:

procedure-specification::=

$$\left\{\begin{matrix}SPECIFY\\SPC\end{matrix}\right\}\ procedure\text{-}identifier$$

$$ENTRY\ \left[\left(\left\{,\ parameter\text{-}mode\ \cdots\right\}\right)\right]$$

$$\left[result\text{-}attribute\right]$$

$$\left[RESIDENT\right]\ \left[REENT\right]\ \left[GLOBAL\right]$$

Note the different keyword "ENTRY" (instead of "PROCEDURE"). Within the formal parameter-list only the types are listed, no longer the names of the formal parameters.

example for a procedure-declaration:

```
P : PROCEDURE  (A (,) FIXED IDENT, B CLOCK)
        RESIDENT GLOBAL;
        .
        .
        .
    END;
```

The corresponding specification is:

```
SPECIFY P ENTRY ( (,) FIXED IDENT, CLOCK)
            RESIDENT  GLOBAL;
```

1.2.2.1 Parameters

There are two ways, how a procedure can communicate with its environment.
Either via global objects, i.e. objects the scope of which includes the declaration and the call of a procedure. The second way is to pass objects to a procedure, when it is called. These objects are called "actual parameters". Within a declaration (or specification, resp.) of a procedure this has to be prepared by using "formal parameters".
How the parameters are passed to a procedure is detailed in section 2.2.2.1.
In this section it is explained, which objects may be used as formal parameters and how the formal-parameter-lists are constructed within the declaration and specification of a procedure.

Within a declaration the list is denoted as follows:

({ , one-identifier-or-list
parameter-mode ... })

'one-identifier-or-list' denotes the name(s) of the formal parameter(s). 'parameter-mode' denotes the corresponding type and the way how the parameter is passed to the procedure.

parameter-mode::=

$$\left\{ \begin{array}{l} [\,(\,[,]\,[,]\,)\,]\ \textit{local-mode}\ [\,\textit{IDENT}\,] \\ [\,(\,)\,]\ \textit{dation-attr}\ \textit{IDENT} \end{array} \right\}$$

According to the syntax above the following objects may be used as formal parameters:

- all types of simple objects

  FIXED
  FLOAT
  CHARACTER
  BIT
  CLOCK
  DURATION

- structures built of simple objects
- arrays of simple objects
- arrays of structures
- dations
- arrays of dation

Parameters of the same type may be denoted in one list.

example: ((A,F,X) FIXED)

is short for

(A FIXED, F FIXED, X FIXED)

Formal parameters that are arrays are denoted with empty bound-lists:

( ) denotes a one-dimensional array,
(,) a two-dimensional array and
(,,) a three-dimensional array.

example:

```
( AR(,) FLOAT IDENT )
```

This parameter-list contains a two-dimensional array of type FLOAT.

The keyword 'IDENT' denotes that the corresponding parameter is passed to the procedure by the "identical-mechanism". If it is omitted, the parameter is passed by the "initial-mechanism".
Both mechanisms are detailed in section 2.2.2.1., where one can also find examples.

example for a complex parameter-list:

```
(D DATION IN ALPHIC IDENT, (X,Y) ( ) STRUCT [A FIXED,
 S CHAR(5)] IDENT, C INV FIXED, (R,S,T) FLOAT IDENT)
```

There are seven formal parameters:

- a dation
- two one-dimensional arrays of structures
- a FIXED-constant
- three FLOAT-variables

Within a specification of a procedure the formal-parameter-list is denoted as follows:

({ , *parameter-mode* ... })

This means, that within a specification only the types of the formal parameters are listed.
So within a specification the example above must be denoted in the following way:

```
( DATION IN ALPHIC IDENT,
  ( ) STRUCT [ A FIXED, S CHAR(5) ] IDENT,
  ( ) STRUCT [ A FIXED, S CHAR(5) ] IDENT,
  INV FIXED,
  FLOAT IDENT,
  FLOAT IDENT,
  FLOAT IDENT )
```

1.2.2.2 Function Procedures

A special group of procedures, that return a result to the point where they are called from, is called "functions" or "function-procedures".

Function-procedures are declared with the 'result-attribute'.

result-attribute::=

RETURNS (simple-mode)

The syntax shows that a function-procedure can only return an object of one of the basic types.

example:

```
FP : PROCEDURE RETURNS (FIXED) GLOBAL;
       :
     RETURN (I);
     END;
```

This is a function-procedure without parameters, returning the value of a FIXED-object.

How function-procedures are called and how the result is returned is detailed in section 2.2.2.

1.2.2.3 Reentrancy

A procedure having the attribute "REENT" may be used "reentrant".
This means, it may be called from several tasks at the same time (without explicite synchronization) and this will never lead to conflicts.
Reentrancy may be viewed as executing several "copies" of a procedure, no copy knowing about the existence of the others.
How this problem is solved is not described in Basic PEARL. The attribute "REENT" just indicates that the programmer wants to use this procedure in more than one task.
The compiler or the operating-system, respectively, have manage synchronization for such procedures.

1.2.3 Compound Objects

1.2.3.0 General

Starting with the basic data-types explained in section 1.2.1 the user can form new types by putting them together. In Basic PEARL there are two forms of compound objects that can be constructed by the programmer:

- arrays
- structures

In the following sections it is detailed how such compound objects are created and what restrictions and rules have to be obeyed.
Section 2.2.3 describes, how compound objects may be accessed and used in operations.

1.2.3.1 Arrays

An "array" is a ordered set of objects all of which have to be of the same type.
This set of objects is given one name.
The elements of an array are characterized by an index, which gives their position within the array.

array-declaration::=

{ *DECLARE* | *DCL* } *one-identifier-or-list*
bound-list *local-mode*
[*RESIDENT*] [*GLOBAL*]

'bound-list' denotes the size and the structure of an array.

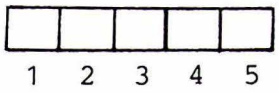
bound-list:=:

(simple-integer-constant-denotation)
[, *simple-integer-constant-denotation*
[, *simple-integer-constant-denotation*]])

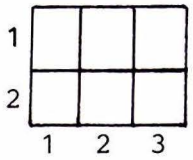
In Basic PEARL an array may have 3 dimensions, therefore 3 integers are provided in 'bound-list'.
The lower-bound is always 1, the upper-bound is given by 'simple-integer-constant-denotation'. Within this range the index can be varied.

example:

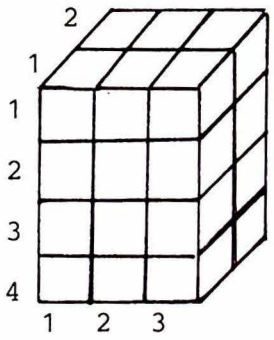
(5) denotes a vector with 5 elements



(2, 3) denotes a two-dimensional array



(2, 4, 3) denotes a three-dimensional array



The type of the elements of the array is given by 'local-mode'.

local-mode::=

$$[INV] \left\{ \begin{matrix} simple\text{-}mode \\ structure\text{-}mode \end{matrix} \right\}$$

Arrays may be constructed of all basic-types and of structures, which are detailed in the following section.

Note: Within declaration of arrays the INV-attribute must not be used!
(Since there is no way of initializing arrays, no constant arrays may be declared).

examples:

```
DCL A(7) FIXED GLOBAL;
DCL TABLE(100,10) CLOCK;
DCL CUBE(10,10,10) FLOAT RESIDENT GLOBAL;
```

An array is specified as follows:

array-specification::=

{ *SPECIFY* | *SPC* } *one-identifier-or-list*

([,] [,]) local-mode

[GLOBAL]

Within a specification of an array just an empty bound-list has to be supplied, to indicate how many dimensions the array has.
Now, 'local-mode' may contain the INV-attribute, i.e. in a specification access to an array may be restricted to "read-only".

examples:

The two global arrays in the examples above may be specified as:

```
SPC A( ) FIXED GLOBAL;
SPC CUBE(,,) INV FLOAT GLOBAL;
```

1.2.3.2 Structures

"Structures" are constructed of "components", which need not be of the same type.
A structure is given a name, and also the single components are given names (contrary to the index of arrays, which is denoted by a number).

A structure is declared as follows:

structure-declaration::=

{ *DECLARE* / *DCL* } *identifier*

STRUCT { [/ (}

{, *[one-identifier-or-list] simple-mode* ...}

{] /) }

[*GLOBAL*]

'identifier' following DECLARE (or DCL, resp.) denotes the name of the structure.

Following the keyword 'STRUCT' the components of the structure are listed; this list may be delimited by [or (/ and] or /).
The components are denoted by a name and the type, which may only be one of the basic-types. Identifiers for components of the same type may be combined to a list.
If the identifier is omitted the corresponding component cannot be selected (refer to section 2.2.3.2).
The names of the components must be unequivocal in their scope, i.e. the scope of the structure. In other words no selector may have the same 'identifier' as any other object in this block.

examples:

```
DCL S STRUCT [ (A,B) FIXED, F FLOAT ] GLOBAL;
DCL PERSON STRUCT  (/ NAME CHAR(30),
                      SEX  BIT(1),
                      AGE  FIXED,
                      ADDRESS CHAR(100) /);
```

A structure is specified in the following way:

structure-specification::=

{*SPECIFY* / *SPC*} *identifier*

[*INV*]

STRUCT {[/ (/ }

{ , [*one-identifier-or-list*]

simple-mode ... }

{] / /) }

[*GLOBAL*]

Note, that within a specification access may again - as for arrays - be restricted by the INV-attribute (which is not possible in a declaration, because there is no constant-denotation for structures).

example:

```
SPC S INV STRUCT [ (A,B) FIXED, F FLOAT ] GLOBAL;
```

1.2.4 Labels

Labels are the only Basic PEARL objects that need no explicite declaration.
They come into existence by writing a 'label-identifier' in front of a statement. (This may be considered as implicite declaration).

label-identifier::=
identifier

A 'label-identifier' may be used in a 'goto-statement' (refer to section 2.2.4.1) to transfer control to a 'labelled' statement.
The scope of a label is the block, where it is used as marking of a statement. Within this scope only one statement may, of course, be marked with this 'label-identifier'.
Separating the label from the statement is achieved by ':'.

example:

```
BEGIN
  :
  :
LØ1 : X := Y;
  :
  :
GOTO LØ1;
  :
  :
END;
```

1.3 Communication Objects

1.3.0 General

In this chapter objects are introduced, that are necessary to comply with the requests of I/O in a realtime language. Basic PEARL enables the programmer to set up his own logical communication structure, based upon the available hardware- and software components.

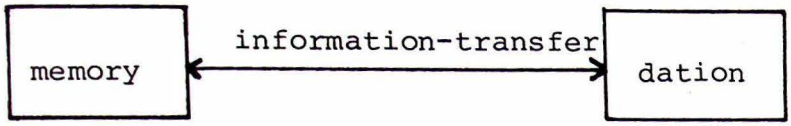
The most important notion in Basic PEARL I/O is the "data-station". A data-station is the source or the sink of a data-transfer. The properties of a data-station are either predefined by the system, or they may be specified by the programmer.
The first group is called "system-defined data-stations" (or "devices"), the second one is called "user-defined data-stations". Elements of these two groups can be connected, forming "dataways". How this is done, and how data-stations are used in I/O-operations is dealt with in section 2.3.

The second type of objects used in I/O is called "control". Controls serve to control data-stations in operation, and to transform PEARL-objects into (or out of) external representation. They are detailed in section 1.3.2.

1.3.1 Data-stations

1.3.1.0 General

Data-stations or "dations" are the objects which serve as partner for the memory in communication operations.



The objects transferred are considered to be contained in channels. The channels of a data-station are introduced in the following section.

Since communication ultimately depends upon the hardware facilities of an installation, these serve as a basis for the definition of data-stations. The data-stations which are given by the hardware are called "system-defined data-stations" or "devices". They are detailed in section 1.3.1.2.

Beyond that, the programmer may construct new data-stations, telling the system, what properties they should have. This is done within a 'dation-declaration'.

dation-declaration::=
{*DECLARE* | *DCL*}
one-identifier-or-list DATION
d-channel-attributes
[*CONTROL (ALL)*]
[*RESIDENT*] [*GLOBAL*]]
dataway-construction;

'one-identifier-or-list' denotes the name of the DATION.
'd-channel-attributes' is detailed in section 1.3.1.1.1.
The control-channel attribute 'CONTROL (ALL)' is described in 1.3.1.1.2.
'dataway-construction' is to connect this user-defined DATION to a device. (cf. section 1.3.1.1.0 linkage, and 2.3.1.1).

Before a system-defined data-station may be used in a PROBLEM-division it has to be specified.
The same sort of specification is necessary, if a global user-defined DATION is to be used within another module.

dation-specification::=

{ *SPECIFY* | *SPC* }

one-identifier-or-list [*()*] *DATION*
d-channel-spec-attr

[*CONTROL (ALL)*]
[*GLOBAL*];

The empty pair of parenthesis indicates a one-dimensional array; it may only be used with devices.
'd-channel-spec-attr' is similar to the declaration attributes; only the dimension of the dation is denoted in a different way. As for other specifications just the number of dimensions (and not the upper bounds) are denoted.

d-channel-spec-attr::=

{ *IN* | *OUT* | *INOUT* }

{ *ALPHIC* | *BASIC* | *trf-item-type* }

[*dim-spec* [*TFU* [*MAX*]]]

{ *DIRECT* | *FORWARD* | *FORBACK* }

{ [*NOCYCL*] | *CYCLIC* }

{ [*STREAM*] | *NOSTREAM* }

dim-spec::=

([,] [,])

The attribute GLOBAL has to be supplied if it is the specification of a user-defined data-station or if the SYSTEM-division is contained in another module.

1.3.1.1 User-defined Data-stations

1.3.1.1.0 General

A data-station can be seen as compound object, the constituents of which cannot be accessed directly.

There are four constituents, that are of interest to the programmer (to understand the function of a data-station):

- linkage
  This is a pointer, which is used to refer to a system-defined data-station. It can only be set once for each user-defined data-station. This is done in the declaration (cf. 2.3.1.1).

- synchronizer
  This may be seen as a counter, the value of which indicates, whether the data-station may be accessed or not (cf. section 2.3.1.2).

- data-channel
  The data-channel can hold or pass on, resp. the data-elements transferred to (or from) this data-station. In the following section all the attributes a 'd-channel' may have are described in detail.

- control-channel
  The control-channel is capable of holding sequences of controls transferred to a data-station. The function of this 'c-channel' is explained in section 1.3.1.1.2.

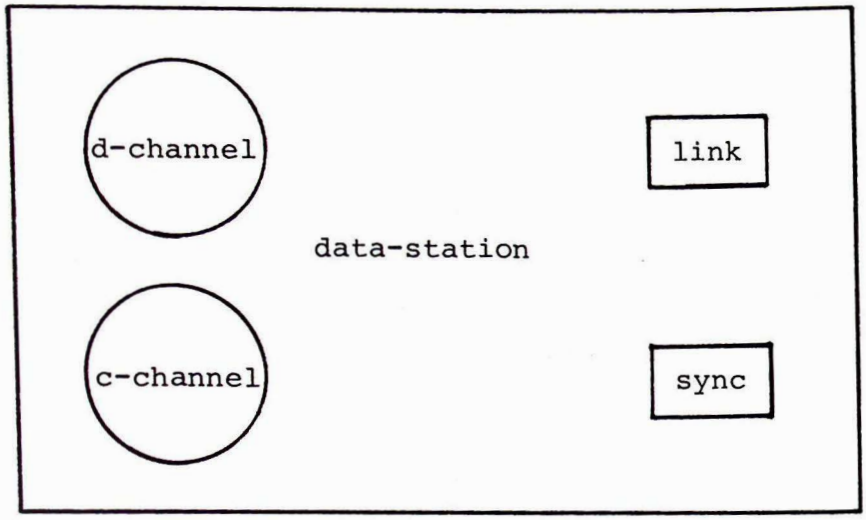


fig. 4: constituents of a data-station

1.3.1.1.1 Data-channel Attributes

The most important attributes of a data-station are summarized in 'd-channel-attributes'.

d-channel-attributes::=

usage
class
[*dimension*
access]

.1 usage

As already mentioned in section 1.3.1.0 memory is always one of the "end-points" of an I/O-operation.
The usage-attribute qualifies a data-station as a transfer-partner for the memory. It indicates the possible transfer-direction.

usage::=

{*IN* | *OUT* | *INOUT*}

With INOUT, both transfer-directions are possible. IN restricts the direction to input from a data-station to the memory (e.g. card-reader, digital input); with OUT output from memory to a data-station is possible (e.g. line-printer).
Restricting the usage within a specification is possible.
So a DATION declared as INOUT and GLOBAL may be specified either as IN or OUT in another module.
But a DATION declared with the attribute OUT may neither be specified IN nor INOUT.

.2 class

The class-attribute describes the items a data-station can accept. These items may be represented in internal or external mode. Internal means that the DATION can hold PEARL-objects in the same way the "system" does.
External means the DATION is capable of changing PEARL-objects into objects external to PEARL (or the opposite direction). This transformation is called "formatting" (cf. section 1.3.2.2).

class::=

$$\left\{\begin{array}{l} \textit{ALPHIC} \\ \textit{BASIC} \\ \textit{trf-item-type} \end{array}\right\}$$

ALPHIC and BASIC are used for DATIONS with external access.
ALPHIC means the DATION transforms PEARL-objects into or out of "alphic-symbols".
"Alphic-symbols" are all characters used to represent information in a special implementation.
A DATION declared ALPHIC can be accessed by a 'get-statement' or a 'put-statement'.

BASIC means that PEARL-objects are transformed into or out of "basic-symbols". These "basic-symbols" are used to represent binary information.
For instance "basic-symbols" can be represented by

- the characters '0' and '1'
- two voltages or two magnetizations etc.

For a DATION declared BASIC the 'take-statement' or the 'send-statement' is possible.

If 'trf-item-type' is used the DATION has internal access.

trf-item-type::=

{*simple-mode* | *structure-mode*}

Note, that only simple objects (as FIXED or DURATION) and structures may be specified as transfer items, but no arrays. A transfer-operation to or from a DATION with internal access transports data in internal representation without transforming them. The 'read-statement' and the 'write-statement' are used for this transport.

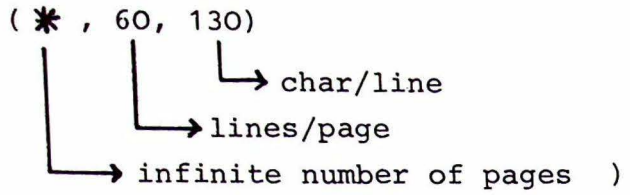
.3 dimension

The d-channel of a DATION may be dimensioned, i.e. it may be constructed as one-, two- or three-dimensional array of transfer-items. (These transfer-items may be "alphic-symbols", "basic-symbols" or one of 'trf-item-type' detailed in the previous section).

dimension::=

({✱ | *simple-integer-constant-denotation*}
[, *simple-integer-constant-denotation*
[, *simple-integer-constant-denotation*]])
[*TFU* [*MAX*]]

The integer numbers must be positive. They specify the number of pages, the number of lines and the number of items.
The highest order bound must not be finite. In this case an asterisk (✱) replaces the integer number.
(e.g. the pages of a line-printer may be considered infinite, one would specify the following dimension:



The following combinations are possible:

(pages, lines, items)
(* , lines, items)
(lines, items)
(* , items)
(items)
(*)

Additionally a transfer-unit may be specified. The transfer-unit indicates the smallest number of items transported in one I/O-operation.

Transfer-unit is indicated by TFU; three case have to be considered:

- TFU omitted: this means that the transfer-unit is one item.
- TFU supplied: this means the transfer-unit is one line.
- TFU MAX supplied: this means that the maximum length of the transfer-unit is one line. The actual length is undefined, it depends on the implementation of the DATION.

If TFU is supplied and the actual number of items in an I/O-operation is less than the number of item per line, the line is implicitly completed by appropriate items. For a DATION with the attribute ALPHIC these might be blanks, for BASIC these might be zeroes or "low voltage" etc..

example:

```
DCL PRINT DATION OUT ALPHIC (*, 60, 130) TFU.....
```

The transfer-unit of this line-printer is one line (130 characters). If the following I/O-statement is written:

```
PUT 'PEARL' TO PRINT;
```

this results in the transport of the 5 characters 'P', 'E', 'A', 'R', 'L' and 125 blanks to the d-channel of PRINT.

.4 access

The following attributes determine the access facilities to the d-channel of a DATION.

access::=

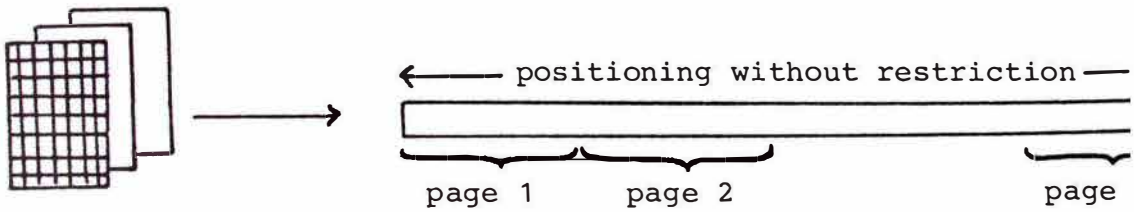
{ *DIRECT* | *FORWARD* | *FORBACK* }

{ [*NOCYCL*] | *CYCLIC* } { [*STREAM*] | *NOSTREAM* }

The attribute DIRECT allows unrestricted access to all transfer-items within the array defined by 'dimension'. Absolute positioning is possible (cf. section 1.3.2.1, where the controls for absolute positioning are explained).

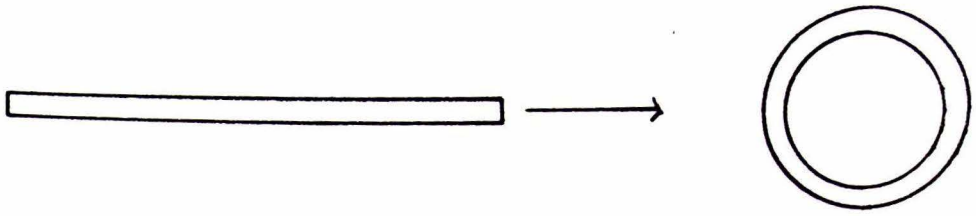
Sequential access is indicated by the attributes FORWARD and FORBACK. FORWARD restricts the access to the positive direction, FORBACK permits sequential access forwards and backwards.
Sequential access only allows relative positioning. The controls therefore are detailed in section 1.3.2.1, too.

The following attributes indicate, what has to be done, if an I/O-operation exceeds one of the bounds given in 'dimension'.
If the DATION has the STREAM-attribute (which is default), positioning beyond the bounds in the array of the d-channel is possible, but not beyond the highest bound. (One can image a three-dimensional array displayed in linear form, without marking the inner bounds).



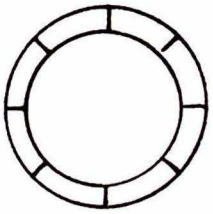
NOSTREAM indicates that positioning beyond one bound would result in a signal.

CYCLIC means that also the outermost bounds may be exceeded, in which case positioning starts again at the lowest (or highest) value. (Imagine the linearized form shown above put together to a ring. Then you cannot see the beginning and the end anymore).



NOCYCL means, that the lowest and the highest value within 'dimension' have to be obeyed, otherwise a signal is raised.

Note, that a DATION containing an asterisk in 'dimension' may, of course, never be CYCLIC.
But a DATION need not necessarily be STREAM to admit the CYCLIC-attribute. (Imagine this as a ring with the inner bounds marked).



Within a specification of a DATION the access attributes may be restricted in the following way:

- a DATION declared DIRECT may be specified FORBACK or FORWARD
- a DATION declared FORBACK may be specified FORWARD
- CYCLIC may be restricted to NOCYCL
- STREAM may be restricted to NOSTREAM

1.3.1.1.2 Control-channel Attribute

In order to have a control-channel in addition to the d-channel, a data-station must have been declared with the attribute:

CONTROL (ALL)

(cf. section 1.3.1.0).

A control-channel must always be supplied when positioning or explicite transformation is intended.
In other words when any transfer-operation uses objects of type "control" (i.e. contains the phrase "BY c-list") the corresponding data-station must have been declared (or specified) with the attribute 'CONTROL (ALL)'.

1.3.1.2 System-defined Data-stations

1.3.1.2.0 General

There is a set of intrinsic data-stations, called "devices", the names and types of which are known to the system without explicite declaration.
Devices must be introduced in the SYSTEM-division, before they can be used in a PROBLEM-division.
The information contained in the SYSTEM-division is based on the implementation handbooks. There, the properties of all devices available within an installation are registered.
Within a PROBLEM-division, all the devices used must be specified. How this is done has been explained in section 1.3.1.0.

Syntax and - as far as defined on language level - semantics of the SYSTEM-division is treated in the following section.

1.3.1.2.1 System-division

.0 General

The SYSTEM-division is the implementation- and installation-dependent part of a PEARL-program. Here restrictions are formulated with respect to the devices listed in the implementation handbook. The SYSTEM-division has two main goals:

- to describe the configuration used within a special process. That means, a selection of devices has to be made, and the connections between the selected ones have to be specified.

- to identify a device with a user-chosen name. This has to be done, because within the PROBLEM-division one only refers to these user-names, and no longer to system-names.

The term "device" in SYSTEM-division-context is used in a slightly more general way than within the PROBLEM-division, because it is not distinguished between data-stations, interrupts and signals. This distinction is not made till the specification in the PROBLEM-division. It is a matter of the implementation handbook to determine, which device is to be specified as INTERRUPT, which as SIGNAL and which as DATION. This may not be inferred from the SYSTEM-division due to the equal treatment of the three.

The syntax of the SYSTEM-division is given by the following:

system-division::=

SYSTEM;
{[*connection;*]...}

Note: Within a Basic PEARL-program there may only be one SYSTEM-division!

Due to the different goals of the SYSTEM-division a 'connection' can be a 'device-specification' (combining a name and a device) or a 'connection-specification' (in general specifying a connection between two devices).

connection::=

{*device-specification* / *connection-specification*}

'connection' is detailed in the following sections.

.1 Device Specifications

.1.0 General

As already mentioned in the previous section, those SYSTEM-division devices, that are intended to be used in a PROBLEM-division (as DATION, INTERRUPT or SIGNAL) must be given programmer-names. This may either be done in conjunction with a connection-specification (cf..2), or independently, in form of a device-specification. The latter is mainly to provide user-names for standard peripheral devices. The system implicitly knows the connections; the user does not have to write them down.

device-specification::=

$$\left\{\begin{array}{l} \textit{simple-dev-spec} \\ \textit{array-dev-spec} \end{array}\right\}$$

.1.1 Simple Device-Specification

A 'simple-dev-spec' attaches one or several new identifiers (chosen by the programmer) to one device.

simple-dev-spec::=
{*user-device-identificator:* ...}
system-device-identificator [*index*]

user-device-identificator::=
identifier

system-device-identificator::=
identifier

'user-device-identificator' is the name for one device chosen by the programmer. Within the PROBLEM-division the object denoted by this name may be specified as DATION, INTERRUPT or SIGNAL.

index::=
integer-in-brakcets

'system-device-identificator' and 'index' must be taken from the implementation-handbook.

examples: PRINTER:LISTING:LPR53;

(two names (PRINTER and LISTING) are attached to the device LPR53)

OVERFLOW:OFL;

(the name OVERFLOW is attached to the signal OFL)

.1.2 Device-array Specification

In the implementation-handbook a series of devices with identical properties may be grouped into an array.
In the SYSTEM-division such arrays (or slices of such arrays) may be identified with one or several user-named dation-arrays.

array-dev-spec::=

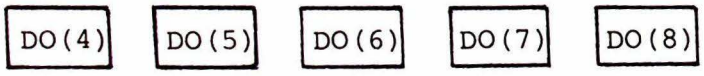
{*user-device-identificator array-bounds:*•••}
system-device-identificators array-bounds

array-bounds::=

(simple-integer-constant-denotation
: simple-integer-constant-denotation)

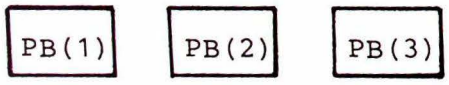
The array-bounds following the user-name and the system-name do not have to be identical, but the width (i.e. upper-bound - lower-bound) must be equal.

examples: • LAMP(1:5) : DO(4:8);



LAMP(1) LAMP(2) LAMP(3) LAMP(4) LAMP(5)

• SWITCH(1:3) : IN(5:7) : PB(1:3);



SWITCH(1) SWITCH(2) SWITCH(3)
IN(5) IN(6) IN(7)

.2 Connection Specification

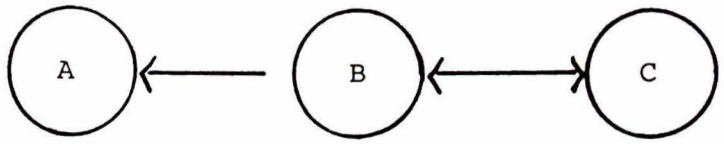
.2.0 General

As already mentioned in .0 one goal of the SYSTEM-division is to describe the hardware configuration used in the special program system.
This is achieved by specifying connections between SYSTEM-division devices.
A 'connection-specification' establishes a relation between two "end-points".
The direction for the flow of information is indicated by 'transfer-direction'.

transfer-direction::=

{ -> | <-> | <- }

example:



In this example the flow of information between device A and device B is only possible from device B to device A, even if bidirectional exchange of information is provided in the implementation handbook.

A <- B;

Between device B and device C bidirectional exchange is possible. This must be provided in the implementation handbook.

B <-> C;

The end-points of a 'connection-specification' may be single devices, interrupts, device-arrays etc..
As in the 'device-specification', the programmer may identify a device with a user-name within a 'connection-specification', but only on the left side of a connection (i.e. left of 'transfer-direction').

Due to the different complexity of the end-points, there is a simple and a compound specification.

connection-specification::=

{ *simple-con-spec* | *compound-con-spec* }

.2.1 Simple Connection Specification

A 'simple-con-spec' specifies a possible transfer-direction between two devices.

simple-con-spec::=

[{*user-device-identificator:* ···}]
system-device-identificator [*index*]

transfer-direction
[*simple-connection-point-description*]

The left side is similar to a 'simple-dev-spec' (cf. .1.1) with the exception that the 'user-device-identificator' may be missing.
The right side ('simple-connection-point-description') must not contain a user-name, but more detailed information about a device is possible.

simple connection-point-description::=

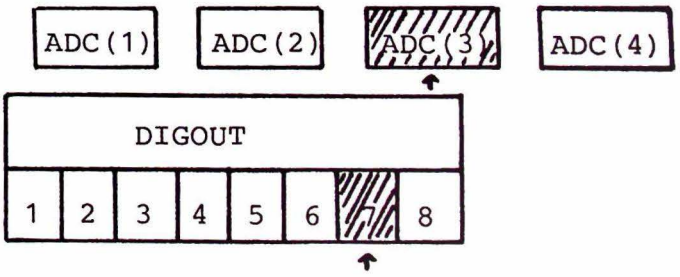
system-device-identificator [*index*]
[✱*simple-integer-constant-denotation*]

If a device has more than one connection-point '✱ simple-integer-constant-denotation' allows to select one of them.

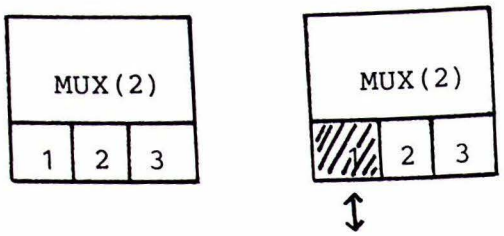
examples for 'simple-connection-point-description':

(1) ADC(3)

(2) DIGOUT ✱7

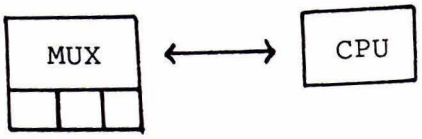


(3) MUX(2) ✱ 1

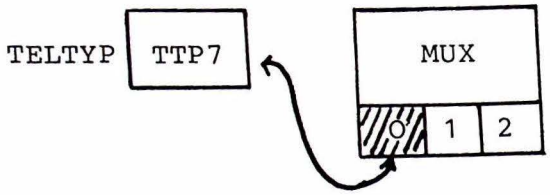


examples for 'simple-connection-specification':

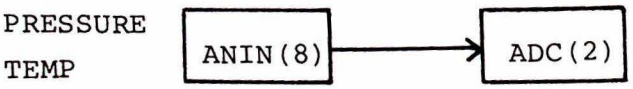
(1) MUX <-> CPU;



(2) TELTYP:TTP7 <-> MUX ✱0;

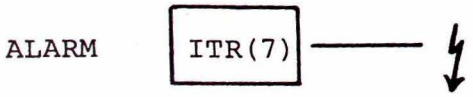


(3) PRESSURE:TEMP:ANIN(8) -> ADC(2);



'simple-connection-point-description' may be omitted, if the end-point cannot be described on language-level. (e.g. an interrupt source located somewhere within the process).

example: ALARM:ITR(7) -> ;



.2.2 Compound Connection Specification

.2.2.0 General

The 'compound-con-spec' enables the programmer to specify connections between several devices, device-arrays or groups of devices within a single specification.

compound-con-spec::=

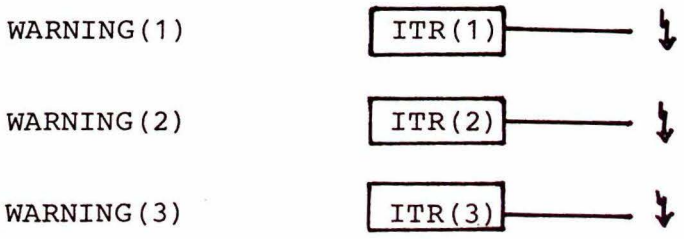
[{*user-device-identificator* [*array-bounds*] : ```}]
[*system-device-identificator* [*array-bounds*]]

transfer-direction

[{+ *connection-point-description* ```}]

At least one of the outermost pairs of square brackets must be omitted (i.e. 'transfer-direction' must not stand alone).
On the left side of 'transfer-direction' the description of a single device or an array may stand combined with one or several usernames; or user-names may stand alone.
On the right side either an array-description may stand (cf. .2.2.1) or a group-description (cf. .2.2.2).
The right side may be completely omitted, in which case it is assumed, that the end-points are situated within the process (e.g. an interrupt-array).

example: WARNING(1:3) : ITR(1:3) -> ;



'connection-point-description' has the following syntax:

connection-point-description::=

system-device-identificator
[*index-or-array-bounds*]
[*{ {simple-integer-constant-denotation [,simple-integer-constant-denotation]} | array-bounds }*]

Note that no user-name may be specified within 'connection-point-description'.

index-or-array-bounds::=

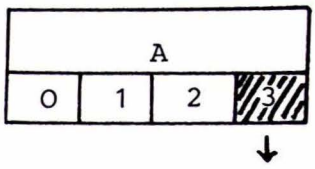
(simple-integer-constant-denotation
[*: simple-integer-constant-denotation*])

'index-or-array-bounds' allows to denote

- a single element of an array
  (e.g.: A(5))
- or a slice
  (e.g.: A(3:6))
- or a whole array, respectively
  (e.g.: A(1:8)).

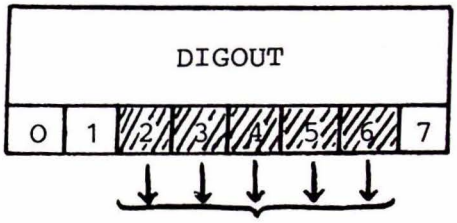
The construction following '✱' allows to denote sub-divided connection-points of a single device (or an element out of an array).
The first integer is the number of the subdivided connection-point.

e.g. A ✱ 3



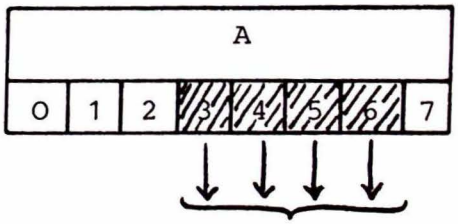
If the second integer (after the comma) is supplied, it denotes the width of a slice of subdivided connection-points, starting with the number specified by the first integer.

e.g. DIGOUT ✱2,5



Alternatively 'array-bounds' may be specified after '✱'.

e.g. A ✱ (3:6)

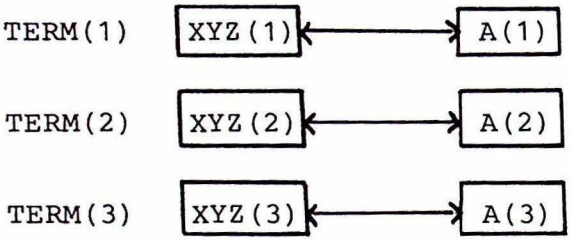


.2.2.1 Array Connection Specification

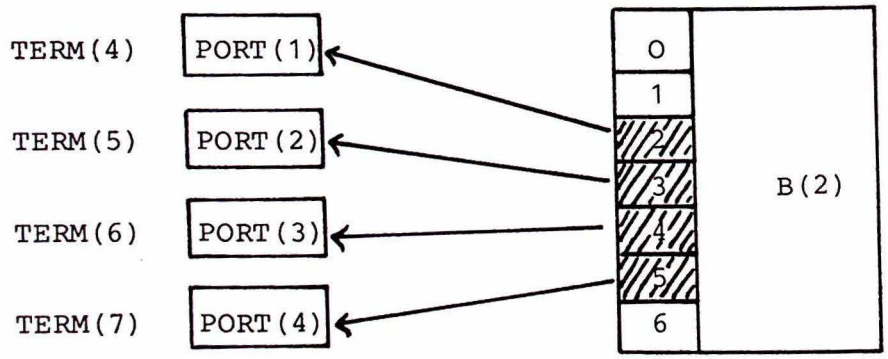
Array connections are just a shorthand notation for several simple connections. This implies that the number of connection points must be the same on both sides of transfer-direction.

The following examples are to illustrate this:

(1) TERM(1:3) : XYZ(1:3) <-> A(1:3);



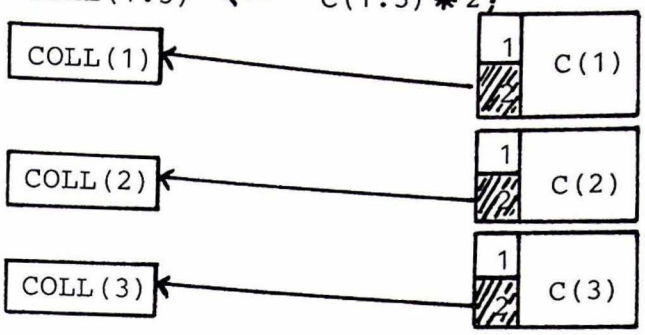
(2) TERM(4:7) : PORT(1:4) <- B(2)*(2:5);



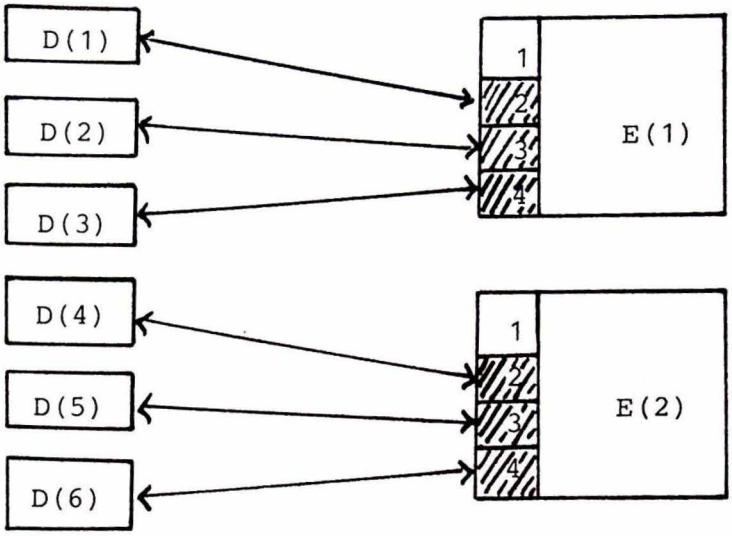
This example can also be denoted in the following way:

TERM(4:7) : PORT(1:4) <- B(2)*2,4;

(3) COLL(1:3) <- C(1:3) * 2;



(4) D(1:6) <-> E(1:2) * (2:4);

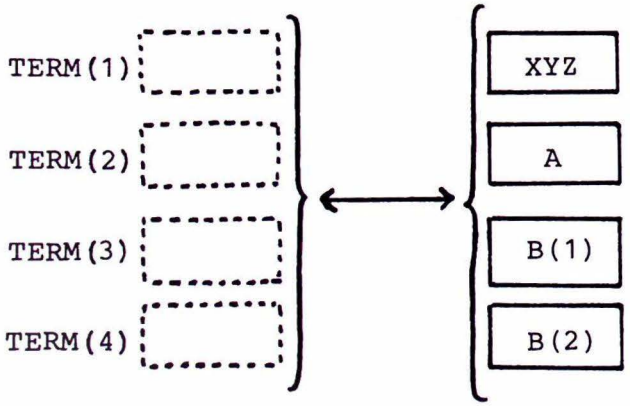


.2.2.2 Group Connection Specification

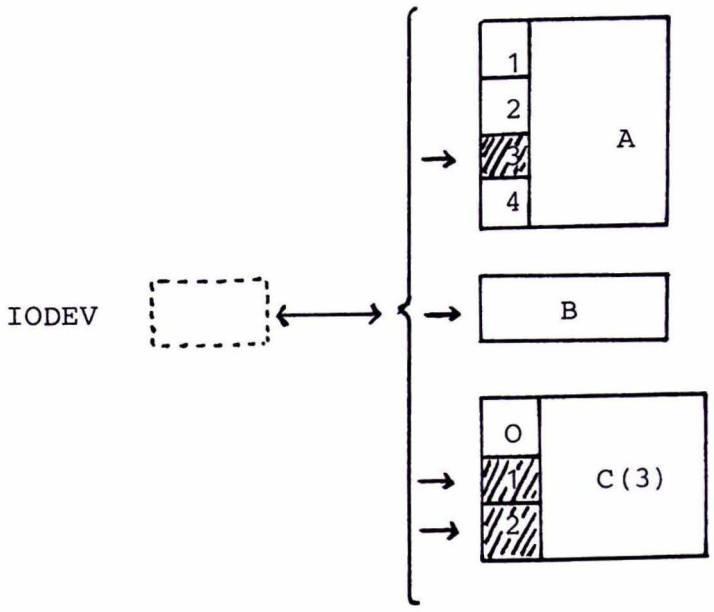
A device-group is an ordered sequence of devices and/or connection-points. The programmer can connect such a device-group with an array or a single device. This implies that the attributes of the elements must be compatible, as defined in the implementation handbook. The sequence within the group is important, because it determines the logical position of the connection-points. Syntactically the elements of a group are separated by the symbol '+'.
All the examples for 'simple-connection-point-description' (cf..2.1) and for 'connection-point-description' (cf..2.2.0) may be used to illustrate the elements of a device-group.

examples for group connections:

(1) TERM(1:4): <-> XYZ + A + B(1:2);



(2) IODEV: <-> A*3 + B + C(3) * (1:2);



1.3.2 Controls

1.3.2.0 General

Due to the importance of the c-channel (see section 1.3.1.1.2) for the semantics of communication-statements, there is a special type of objects called "controls".
In Basic PEARL only "predefined" controls exist, i.e. the names, the parameters, the semantics etc. of these objects are determined. (The programmer may not declare new controls).

There are two different kinds of control elements: such that match an item in the data-sequence of a DATION, and such that do not. The first group is called "matching controls", the second one "non-matching controls".

Matching controls control the transformation of a PEARL-object into (or out of) a non-PEARL-object. This transformation is called "formatting", therefore matching controls are called "formats", too. They are detailed in section 1.3.2.2.

Non-matching controls control a data-station in operation, either during it is established or during it is used. This kind of controls is explained in section 1.3.2.1.

Controls may be combined in lists, when they are used in communication operations. How these lists are constructed is detailed in section 1.3.2.3.
For the purpose of abbreviation these lists may also be predefined and supplied with a name. This is called "remote-format" and detailed in section 1.3.2.4.

1.3.2.1 Non-matching Controls

1.3.2.1.0 General

There are two different groups of non-matching controls. The first group influences the physical dataset belonging to a DATION. They are called "open-controls".
The second group permits explicite positioning within a DATION. Therefore they are called "positioning-controls".

non-matching-control::=

{ *open-control* }
{ *pos-control* }

1.3.2.1.1 Open Controls

These open controls are to handle DATIONS with identifiable datasets. They influence the access, the identification and the disposition of the datasets.

open-control::=

{ *IDF (symbol-or-constant)* }
{ *OLD* }
{ *NEW* }
{ [*ANY*] }

- access:

In Basic PEARL only shared access to a DATION is possible, i.e. several tasks may use the DATION at the same time.
No task may reserve a DATION exclusively.
Therefore there is no need for a special control, "shared access" is defaulted.

- identification:

To identify a dataset the control IDF is provided.
The parameter following IDF (in brackets) specifies the name of the dataset.
This parameter may be a constant or a variable of type CHAR(n). The length and the interpretation of this parameter depend on the implementation.

```
e.g.:    IDF ('FILEØ1')

     or DCL A CHAR INIT ('FILEØ2');
          :
          :
          :
        IDF (A)
```

If IDF is omitted, an implementation-dependent name is used for the dataset.

- disposition:

There are three open controls to specify the disposition of a dataset.

OLD: a dataset with the name specified in IDF must exist. If not, or if IDF is omitted, a signal is raised.

NEW: a new dataset with the name given in IDF is to be established. If it already exists, or if IDF is omitted, a signal is raised.

ANY: if a dataset already exists, it is reopened; if not, a new dataset is established.
If IDF is omitted an implementation-dependent name is used for the dataset.

If no disposition is specified, ANY is defaulted. Commonly for all disposition controls after the 'open-statement' the transfer-item-pointer refers to the beginning of the dataset.

1.3.2.1.2 Positioning Controls

Explicite positioning can be done in two ways:

- absolute
- relative

Absolute means the transfer-item-pointer can be set to any position within 'dimension' without referring to the actual position. Relative positioning always specifies a distance from the actual position.

pos-control::=

{*abs-pos-ctrl* | *rel-pos-ctrl*}

1.3.2.1.2.1 Absolute Positioning

Absolute positioning is only possible if the DATION has been declared with the attribute DIRECT.

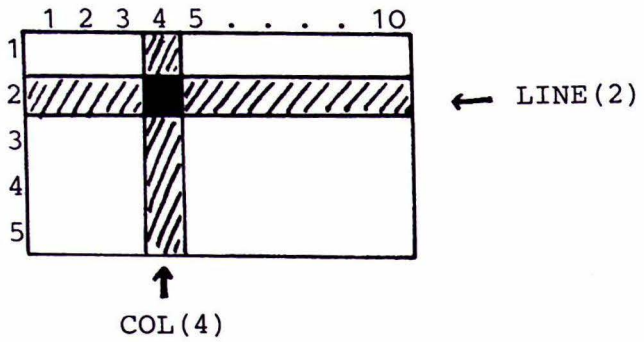
abs-pos-ctrl::=

{ {COL | LINE} (*simple-expr*)

| POS (*simple-expr* [, *simple-expr* [, *simple-expr*]]) }

'simple-expr' may be a positive integer or a variable denoting such an integer. The value must be within the range of the corresponding dimension.
COL refers to the lowest dimension (to the items), LINE refers to the second dimension (to the lines).

example: dimension = (5, 10)



POS may have three parameters, referring to the three dimensions. Leading parameters may be omitted (- the corresponding commas, too); they are defaulted to the actual value.

example: POS (7, 10, 3) sets the transfer-item-pointer to the seventh page, tenth line into the third column.

If in the following I/O-statement POS (2, 9) appears, the transfer-item-pointer keeps pointing to the seventh page, but now refers to the second line and the ninth column.

1.3.2.1.2.2 Relative Positioning

If a DATION has been declared with the attribute FORWARD or FORBACK only relative controls may be used.

rel-pos-ctrl::=

{ { *X* / *SKIP* / *PAGE* } [*(simple-expr)*]

{ *ADV (simple-expr* [*, simple-expr* [*, simple-expr*]] *)* } }

'simple-expr' must be an integer, or an identifier denoting an integer. It may be positive or negative.

X refers to the lowest dimension (the items). So X(n) means: advance n items.

SKIP refers to the second dimension (the lines). If the actual position of the transfer-item-pointer is within line a, SKIP(n) sets the pointer to the beginning of line (a+n).

PAGE refers to the third dimension (the pages). If the actual position is somewhere on page a, PAGE(n) sets the pointer to the top of page (a+n).

If the parameter of X, SKIP or PAGE is missing it is defaulted to 1.
So, PAGE without parameter means: go to the top of the next page.

The parameter must be positive if the DATION concerned has the attribute FORWARD, and may also be negative if the attribute is FORBACK or DIRECT.

examples: let the dimension be (10, 10, 10)

actual position	control	new position
(4, 3, 7)	X	(4, 3, 8)
	X(-5)	(4, 3, 2)
	SKIP(2)	(4, 5, 1)
	PAGE(4)	(8, 1, 1)
	PAGE(-2)	(2, 1, 1)

ADV may have three parameters, affecting the corresponding dimensions. Leading parameters (and the corresponding commas) may be omitted, in which case they are defaulted to zero.
(Note the difference in defaulting X, SKIP and PAGE!)

If a DATION has the attribute FORWARD the leftmost nonzero parameter must be positive.

examples: dimension = (10, 10, 10)

actual position	control	new position
(3, 5, 6)	ADV(1, 0, 3)	(4, 5, 9)
	ADV(2)	(3, 5, 8)
	ADV(-3, 0)	(3, 2, 6)
	ADV(-1, -5)	(3, 4, 1)

If the bounds given in dimension are exceeded during relative positioning, this is in error unless the DATION has the attribute STREAM (resp. CYCLIC).
Positioning in continued within the next dimension (or starts again at the lowest or highest value after reaching the outermost bound).

examples: DCL D DATION IN ALPHIC (10, 10, 10) FORWARD CYCLI

actual position	control	new position
(4, 8, 7)	X(5)	(4, 9, 2)
	SKIP(7)	(5, 5, 1)
	ADV(7, 0, 5)	(1, 9, 2)
	PAGE(9)	(3, 1, 1)

1.3.2.2 Matching Controls

.0 General

If a DATION has either the attribute ALPHIC or BASIC (i.e. the 'class' is external, cf. section 1.3.1.1.1) PEARL-objects have to be transformed to or from non-PEARL-objects. This transformation is called "formatting". It is controlled by "matching-controls" or - how they are often called - by "formats".

matching-control::=

$$\left\{\begin{array}{l} \textit{number-format} \\ \textit{string-format} \\ \textit{time-format} \\ \textit{list-format} \end{array}\right\}$$

Each 'matching-control' matches exactly one data-element. Syntax and semantics will be detailed in the following sections.

.1 Number Format

.1.0 General

In Basic PEARL two formats are provided for the external representation of numbers (i.e. objects of type FIXED or FLOAT).

number-format::=

$$\left\{\begin{array}{l} \textit{f-format} \\ \textit{e-format} \end{array}\right\}$$

.1.1 Fixed-point Format

The fixed-point format "F" can be used to transform objects of type FIXED or FLOAT.

f-format::=

F (width [*, decim* [*, scale*]] *)*

width::=

simple-expr

decim::=

simple-expr

scale::=

simple-expr

(1) output

The object that matches to a fixed-point format is transformed into the following sequence of alphic symbols:

_---_sX---X [.X---X]

where "_" stands for the alphic symbol "blank", "." for "decimal point", "s" for "plus" or "minus" and "X" for a member of the set of digits "zero" to "nine". The brackets [,] enclose options.

"---" stands for repetition of the symbols enclosing, a number of times given by 'width', 'decim' and 'scale' in the following way:

. the total number of alphics produced is given by 'width',
. X following the decimal point is repeated 'decim' times,

. the digits preceding the decimal point are given by the corresponding significant digits of the FLOAT or FIXED to be formatted, after a shift according to 'scale' , i.e., the output corresponds to a FIXED or FLOAT multiplied by 1Ø**'scale'.

  If due to 'scale' or the absolute value of the source the number of alphics needed exceeds the number available this is in error and a SIGNAL is raised. In the opposite case zeros are produced.

If 'decim' is omitted the alphics produced are

_---_sX---X

with "X---X" corresponding to the integer part of the source (proper rounding implied).

If Ø < 'width' < 'decim', an implementation-dependent character, e.g. the alphic "asterisk" (*) is produced 'width' times.

If 'width' = Ø, no alphic is produced, the source is skipped.

Leading zeros are replaced by blanks except for the zero preceding the decimal point.

(2) input

The sequence of alphics to be transformed into memory has the form

_---_[[s]x---x[.[x---x]]]_---_

The alphics admitted and the number of repetitions are given as for output. Alphics not admitted lead to a SIGNAL.

Only the total number of blanks being fixed, the sequence

[s]x---x[.[x---x]]

may be located anywhere within the field of 'width' alphics.

If this sequence is empty (i.e. consists of blanks only) the value is defaulted to "zero".

s is defaulted to "plus" (+).

'decim' and 'scale' are defaulted to "Ø".

If the alphic sequence contains a decimal point,

[s]x---x.[x---x]

'decim' - if supplied - is overridden. The effect of 'scale' is to shift the decimal point in the alphic sequence, i.e. prior to assignation to memory.

If 'width' = Ø, no assignment is made. The object in the data-list is skipped.
If due to the 'precision' a transformation is not possible, a SIGNAL is raised.

examples:

value	format	output
13.5	F(7, 2)	␣␣13.5Ø
275.2	F(4, 5)	****
22.8	F(5)	␣␣␣23
212.73	F(9, 2, 2)	␣21273.ØØ

.1.2 Exponential Format

The exponential format "E" can be matched by objects of type FLOAT or FIXED.

e-format::=

E (width [*, decim* [*, signif*]]*)*

signif::=

simple-expr

(1) output

The source is transformed into the following sequence of alphics:

_---_[-]X---X.X---XEsXX

E stands for the alphic "E".

The total number of alphics produced is given by 'width'.

The number of digits "X" following the decimal point is given by 'decim'.

The total number of digits of the mantissa (the digits preceding "E") is given by 'signif'.

The exponent (the digits following "Es") is chosen such that the first digit preceding the decimal point is different from "zero", except for a source-value Ø.

If 'width' is such that no mantissa-digit can be produced, i.e. if 'width' $<$ 6 for positive source-values or 'width' $<$ 7 for negative source-values, a SIGNAL is raised and an implementation dependent sequence of alphics is produced, for example "asterisks".

If 'decim' = 'signif' there is no digit preceding the decimal point.

If 'decim' $>$ 'width' $>$ Ø a SIGNAL is raised.

If Ø $<$ 'width' $>$ 'decim' $>$ 'signif' the mantissa is chosen such that

$$10^{\text{'signif'-'decim'-1}} \leq |\text{mantissa}| < 10^{\text{'signif'-'decim'}}$$

If 'width' = Ø, no alphics are produced, the source is skipped.

(2) input

The general form of the sequence of alphics to be transformed into the sink is the following:

```
_---_ [[s] [X---X.X---X] [E[s] [X]X]]_---_
```

The alphics admitted and the number of repetitions are given as for output. Alphics not admitted lead to a SIGNAL.

The sequence

```
[[s] [X---X.X---X] [E[s] [X]X]]
```

may contain no blanks and may be located anywhere within the field of 'width' alphics.

If the mantissa or the whole sequence are empty they are defaulted to "zero".

The exponent-part is defaulted to "E+ØØ".

Let m be the number of mantissa-digits supplied, i.e., the total number of "X" in the alphic sequence "X---X.X---X".

. If m < 'signif', 'signif' is overridden by m.
. If 'signif' < m, precision may be lost.

If the alphic sequence contains a decimal point, 'decim' - if supplied - is overridden.

s is defaulted to "plus" (+).

If due to the exponent-part of the alphic sequence a transformation is not possible, a SIGNAL is raised.

If 'width' = Ø, no assignment is made to the sink. The element is skipped in the data sequence.

examples:

value	format	output
-0.07	E(9, 1)	␣-7.ØE-Ø2
2713.5	E(11, 2, 4)	␣␣27.13E+Ø2
2721	E(8)	␣␣␣2E+Ø3

.2 String Format

.2.0 General

String formats are used to transform objects of type CHARACTER(n) or BIT(n), respectively.

string-format::=

$$\left\{\begin{array}{l}\textit{character-string-format}\\ \textit{bit-string-formats}\end{array}\right\}$$

.2.1 Character-string Format

The character-string format "A" is used to transform objects of type CHARACTER(n).

character-string-format::=

A [*(width)*]

On input, 'width' must be supplied.

(1) output

The objects of type CHARACTER matching to this format is transformed into a sequence of 'width' alphic symbols and put to the external d-channel.

If ('width') is omitted, the number of alphic symbols produced conforms to the declaration of the source (cf. 'length', section 1.2.1).

. if 'width' < 'length', the CHARACTERs in excess are lost (right truncation).
. if 'width' > 'length', the alphic symbols in excess are blanks.
. if 'width' = Ø, no alphic symbols are produced.

(2) input

'width' alphic symbols are transformed to objects of type CHARACTER and assigned to the corresponding matching object.

. if 'width' < 'length', the CHARACTERs in excess are blanks.
. if 'width' > 'length', the alphic symbols in excess are lost (right truncation).
. if 'width' = Ø, blanks are assigned to all CHARACTERs.

examples:

(1) output of 'ABCDE' with the format

A(5) results in ABCDE,
A(7) results in ABCDE␣␣,
A(2) results in AB

(2) input of 'ABCDE␣␣' to a variable STR, declared as CHAR(5) with the format

A(5): STR: = 'ABCDE'
A(7): STR: = 'ABCDE'
A(2): STR: = 'AB␣␣␣'

.2.2 Bit-string Formats

The bit-string formats "B", "B3", "B4" match objects of type BIT(n).

bit-string-formats::=

$$\left\{\begin{matrix} B \\ B3 \\ B4 \end{matrix}\right\} \quad [(width)]$$

The semantics of the previous section essentially remains valid if "CHARACTER" is replaced by "BIT" and "blank" by "Ø".

(1) output

According to the alternatives 'B', 'B3' and 'B4', the alphics produced are chosen from one out of three alphabets conventionally used to represent binary, octal or hexadecimal digits, respectively.

binary-digit::=

$$\{ Ø/1 \}$$

octal-digit::=

$$\{ Ø/1/2/3/4/5/6/7 \}$$

hexadecimal-digit::=

$$\{ Ø/1/2/3/4/5/6/7/8/9/A/B/C/D/E/F \}$$

(2) input

Any alphic symbol not member of the appropriate alphabet is in error. An exception are leading and trailing blanks, which are ignored.

examples:

(1) output of the bit-string '0101110' with the format

B(5) results in 01011
B3(3) results in 270
B4(3) results in 7C␣

(2) input to a variable BSTR, declared as BIT(8):

input-string	format	value assigned to BSTR
11111	B(5)	11111000
235	B3(3)	01001110
AB	B4(2)	10101011
201	B(3)	- (error, "2" is no binary-digit)

.3 Time Format

.3.0 General

For Basic PEARL objects of type CLOCK and DURATION the corresponding formats are provided :

time-format ::=

{ *t-format* }
{ *d-format* }

.3.1 Clock Format

The clock format "T" matches objects of type CLOCK.

t-format ::=
T (width [,decim])

(1) output

Objects of type CLOCK are transformed into the following sequence of alphic symbols :

_---_[X]X:XX:XX[.X---X]

The total number of alphics produced is given by 'width'. The number of digits following the decimal point is given by 'decim'.

example: printing with T(12,1) may result in

␣␣14:07:25.3

(2) input

On input a valid 'clock-constant-denotation' has to be located anywhere within 'width' positions.
'decim' - if supplied - is overridden.
(Note: there may be some blanks "surrounding" the colons)

example: input with T (15.2) may be

␣␣7␣:28:␣20␣␣␣␣

.3.2 Duration Format

Objects of type DURATION are matched by the duration format "D".

d-format::=
D (width [,decim])

(1) output

Output of DURATION objects results in the following sequence of alphics:

_---_[X]X_HRS_XX_MIN_XX[.X---X]_SEC

The total number of alphics produced is given by 'width'. 'decim' denotes the number of digits following the decimal point.

example: output with D(24,1) may be

␣␣18␣HRS␣10␣MIN␣59.9␣SEC

(2) input

As for CLOCK objects a valid 'duration-constant-denotation' has to be located within 'width' positions. 'decim'-if supplied-is overridden.
(Note : At least one blank has to be used to separate the keywords HRS, MIN, SEC from the neighbouring numbers !)

example: input with T (26) may be

␣7␣␣HRS␣02␣MIN␣50␣␣SEC␣␣␣␣

.4 List Format

The 'list-format' LIST matches all basic types, i.e. variables or constants of type FIXED, FLOAT, CLOCK, DURATION, BIT and CHAR.

list-format::=

LIST

The transformation or the external representation resp. is implementation-dependent.

example:

```
DCL (A,B) FIXED,
      C CHAR,
      D FLOAT;

DCL DEV DATION OUT ALPHIC(*) CONTROL (ALL);

PUT A, B, C, D TO DEV BY(4) LIST;
```

1.3.2.3 Control Lists

As already mentioned in section 1.3.2.0 single control-elements may be put together to lists, which may be used in transfer-statements.
A 'standard-c-list' may contain one or several 'c-list-element', separated by commas.

standard-c-list::=

$$\{,^{\bullet}\ \textit{c-list-element}\ \cdots\}$$

c-list-element::=

$$\left\{\begin{array}{l}[\textit{mult}]\ \left\{\begin{array}{l}\textit{pos-control}\\ \textit{matching-control}\end{array}\right\}\\ \\ \textit{mult}\quad \textit{(standard-c-list)}\end{array}\right\}$$

'mult' is the denotation for a multiplier. This must be a 'simple-integer-constant-denotation' enclosed in brackets.

mult::=

(simple-integer-constant-denotation)

'mult' means, that the following control (or list of controls) has to be repeated 'mult'-times.

example: (4) A (7) is equivalent to A(7), A(7), A(7), A(7)

(2) (F(7, 2), X(3)) is equivalent to
F(7,2), X(3), F(7,2), X(3)

Some examples may illustrate the complex variety of 'standard-c-list':

```
(3) F (A, B, C)
A(2), SKIP, (5) B4 (12)
X, (2O) (F(8), X(3), F(5, 1))
COL (ST)
(4) (LINE(2), T(17, 2), X, (3) A (L))
```

1.3.2.4 Remote Format

For convenience 'standard-c-list' may be "predefined" by the programmer within a 'r-format-declaration'.

r-format-declaration::=

format-identifier: FORMAT
(standard-c-list);

This is similar to a procedure declaration. The "body" is denoted by 'standard-c-list' (which has been explained in the previous section).
'format-identifier' is the name for this list. It is used in transfer-operations as a parameter for the standard-function "R". This standard-function returns the 'standard-c-list' denoted by format-identifier.

remote-format::=

R (format-identifier)

example:

```
DCL LIN  DATION OUT ALPHIC(130) CONTROL (ALL);
DCL A, B, C, X, Y, Z FIXED;
  :
LINFOR : FORMAT(X, F(7, 2), (2) (X(5), F(8)));
  :
PUT A, B, C TO LIN  BY R (LINFOR);
  :
PUT X, Y, Z TO LIN  BY R (LINFOR);
  :
```

1.4 Realtime Objects

1.4.0 General

If an industrial process is to be controlled by a program system, a special set of language elements is required. The program system has to react upon "messages" coming out of the process, it has to start or stop activities depending on states of the process and so on. The language facilities provided therefore are called realtime-features.
The execution of a program system is influenced by "events" arising from the process. These events are detailed in section 1.4.1.
In section 1.4.3 the basic notion for concurrent program parts, a "task", is introduced. Concurrency means that these tasks can execute their statements independently of each other in a quasi-parallel manner.
If these tasks interact, i.e. if there are logical connections between them or if parts have to be sequentialized, it is necessary to introduce coordinating objects. These are called "synchronizers" and are treated in section 1.4.2.

1.4.1 Events

1.4.1.0 General

An "event" can be seen as a message originating in the environment of a program system. An event is always combined with a condition. When the condition is satisfied (set true) the event "occurs".
(e.g. let the condition be "reaching the end of a tape". As soon as EOF is reached, i.e. the condition is satisfied, an event occurs. A message is sent to the program).

An event can arise implicitly from a system-division device (e.g. a device reports "not ready", an AD-converter signals "done"); explicitly an occurrence can be caused by PEARL-statements (cf. section 2.4.1.2) or by errors during the execution of statements (e.g. overflow, zero-divide etc.).
Reactions on events can be specified by the programmer within the program system. They are discussed in section 2.4.1.3 and in section 2.4.3.0.1.

In PEARL two kinds of events are distinguished. An event concerning one particular activity is called a "signal". An event that may concern more tasks is called an "interrupt".
An interrupt is a priori an independent event occurring asynchronously ("unexpected"). There is no connection to the statement executed the very moment of the occurrence of the interrupt.
It effects all the tasks which have specified a reaction on that interrupt (refer to section 2.4.3.0.1).

A signal is always in connection with exactly one activity. It is caused by a PEARL-statement. The system directs the signal to the task having executed this statement.

1.4.1.1 Interrupts

In PEARL interrupt-handling is based on objects of type INTERRUPT. In the implementation-handbook one can find all the interrupts that can occur.
In the system-division they can be identified with user-names (cf. section 1.3.1.2.2). Interrupts are always GLOBAL and need only specification within a problem-division at module level.

interrupt-specification::=

{*SPECIFY* | *SPC*}
one-identifier-or-list [*()*]
{*INTERRUPT* | *IRPT*}
[*GLOBAL*];

Interrupt may be grouped into arrays, therefore an empty pair of parenthesis is provided.

examples:
```
SPC ALARM IRPT;
SPECIFY ITR() INTERRUPT GLOBAL;
SPC (M1,ITR,INT) IRPT;
```

In Basic PEARL interrupt reactions may only be scheduled task-operations (refer to section 2.4.3.0.1).
For masking interrupts refer to section 2.4.1.1.

1.4.1.2 Signals

During the execution of some selected PEARL statements exceptional conditions may arise. They require a special treatment (e.g. overflow is indicated during the execution of an arithmetic operation).
Signals are used to report the exceptional conditions to the task executing the statement in question.

Similar to interrupts all possible signals can be found in the implementation-handbook. There the system-names are listed and the meaning is explained. Signals may be given user-names within the system-division (cf. section 1.3.1.2.2). Within a problem-division they have to be specified at module level.

signal-specification::=

{ *SPECIFY* | *SPC* }

one-identifier-or-list [*()*]

SIGNAL

[*GLOBAL*];

Analogous to interrupts also signals can be grouped into arrays, therefore an empty pair of parenthesis is provided.

Signals may be stimulated explicitly by the induce-statement (cf. section 2.4.1.2).
Syntax and semantics of signal-reactions are discussed in section 2.4.1.3.
Typical examples for signals are overflow, end-of-file, zerodivide, conversion error etc.

Examples for signal-specification:

```
SPC OFL SIGNAL;
SPECIFY PRINT(3) SIGNAL GLOBAL;
```

1.4.2 Synchronizers

Synchronizers are objects for explicit synchronization of statements in different tasks.
They allow to force a time-ordering upon parts of a program.
This may for instance be necessary if one task produces results and another consumes them. Then the consumer must not be faster than the producer.
In Basic PEARL synchronizers are called "semaphores".
On language level semaphores are realized by objects of type SEMA.
Semaphores are characterized by their states. They may be occupied (requested) or free (released). Changes of states can only be achieved by special operations, one to request a semaphore and one to release it (cf. section 2.4.2 synchronization). The states are displayed by integer numbers. Zero represents the state 'requested', any positive number the state 'released'.

In Basic PEARL semaphores can only be declared at module level.

sema-declaration::=

{ *DECLARE* | *DCL* }

one-identifier-or-list SEMA

[*GLOBAL* [*RESIDENT*]]
[*PRESET* ({ , *integer-constant-denotation* ... })] ;

'one-identifier-or-list' denotes the name of one or more semaphores.
The programmer may assign an initial value for a semaphore by using the PRESET-clause.
The list of numbers following PRESET must contain as many items as 'one-identifier-or-list'.
The initial value may only be zero or a positive integer number. If the PRESET-clause is omitted the initial value is defaulted to zero.

Prior to the use in another module a global semaphore has to be specified.

sema-specification::=

{*SPECIFY* / *SPC*}

one-identifier-or-list SEMA

[*GLOBAL*] ;

Note: Within a specification no initialization is permitted.

examples:
```
DCL RESULT SEMA GLOBAL PRESET 1;
SPC RESULT SEMA GLOBAL;

DCL (S1,VALVE,DONE) SEMA PRESET 3,0,1;
DCL S SEMA GLOBAL;
```

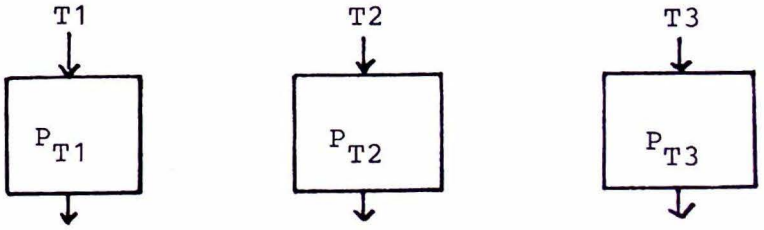
1.4.3 Tasks

In a program system for the control of a technical process, it is not always possible to execute the programs in a predefined sequential way.
Programs have to run at times required by the process, often at the same time as (concurrent to) other programs.
For parts of a program which have to be executed in a concurrent or "quasi-parallel" manner the PEARL object "task" was introduced.

A task consists of declarations and statements. These are worked off sequentially.

To simplify the description of the execution of a task we can assume that each task has its own "virtual processor". All the processors can do their work independently of each other, simultaneously and parallel.

example: a program system consists of 3 tasks T1, T2, T3.



T1 is executed on the virtual processor P_{T1}, T2 on P_{T2} and T3 on P_{T3}. So each task is executed on its own processor, even if they don't overlap in time.

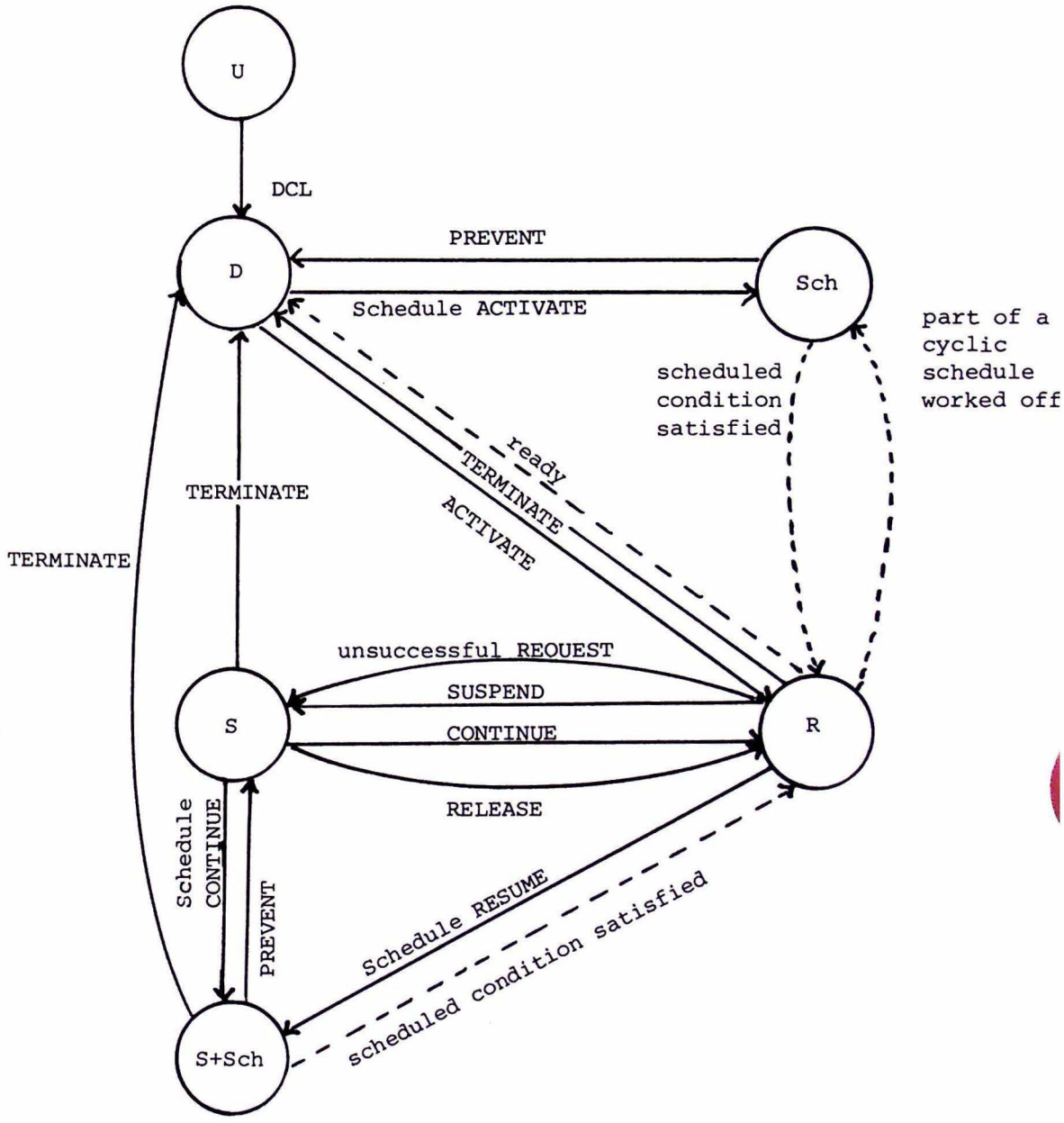
To execute a task it has to be "activated" (this is described in section 2.4.3.1). An activation creates an "activity". One task can have several activities at the same time. (This can happen, if there are more activations of this task before the first activity has finished its execution). But only one activity of a task (the one that was activated first in time) can occupy the virtual processor of this task and execute its statements.
All the other activities of this task have to queue up and wait for their turn (they are worked off sequentially).

A task can have different states. Prior to its declaration for instance it is 'unknown', if an activity is executing its statements it is 'runnable' etc.
The following diagram shows the states of a task (drawn as circles). The transitions between the states are represented by lines between the circles. Every line is marked with the event causing the transition.
The diagram cannot give a complete description of all possible sequences of transitions. It shall just be helpful to understand the task-operations in principle.

More detailed explanations are given in the following description of the states and in section 2.4.3. where the task-operations are described in detail.

fig. 5:

States and transitions of tasks:



Solid lines: transitions activated directly by PEARL-statements

Dotted lines: transitions initialized by the system

Explanation of the states of a task:

- unknown (U) — the operating system does not know the task, therefore no operations are allowed.

- dormant (D) — the task has been declared. It is known to the operating system, but at the moment no activity exists.

- scheduled (Sch) — a condition for an activate-statement has been set up. But this condition is not yet satisfied. The task has to wait until the condition comes true. Then a new activity of this task is created.

- runnable (R) — all the conditions for the start or continuation of an activity are satisfied. Several activities of one or more tasks can be in that state.
  Note: From each task only the first activity has occupied the virtual processor, i.e. the first activity is running (executing its statements). The others are waiting until the first one is finished although they are 'runnable' too.

- suspended (S) — an activity has been stopped. It has to wait for a certain time. There are two ways for an activity to reach this state: either it stops itself (cf. section 2.4.2.2) to wait for an event or a point of time or it is forced to wait for the release of a semaphore (cf. section 2.4.2).

- suspended and scheduled (S+Sch) — analogous to the state 'suspended' an activity is still waiting but a continuation for this activity has already been scheduled.

Declaration of a task:

As any other PEARL objects (except labels) a task has to be declared prior to its use. The declaration takes the task from the state 'unknown' to the state 'dormant'. In Basic PEARL a task has to be declared at module-level.

task-declaration::=

task-identifier: TASK

[{*PRIORITY* / *PRIO*} *simple-integer-constant-denotation*]

[*RESIDENT*] [*GLOBAL*];

block-tail

'task-identifier' denotes the name of the task. The priority clause is detailed in section 2.4.3.0.2.
'block-tail' contains all task-specific declarations and statements.

A task can be used in another module after it has been specified:

task-specification::=

{*SPECIFY* | *SPC*}

one-identifier-or-list *TASK*

[*GLOBAL*]

Note: Changing the priority is not permitted within a specification.

2. OPERATIONS

2.0 General

2.0.0 Operators and Operands

In part 1 all the Basic PEARL objects have been introduced. This part deals with the dynamic features of these objects. It is explained how they may be used in operations.

Some of the Basic PEARL objects are considered to be "executable", for instance procedures or tasks.
This means their declaration contains a sequence of operations which is worked off sequentially as soon as the object is "executed".

In general an operation consists of an operator and one or more operands.

In Basic PEARL all the operators are predefined, i.e. the number and the types of the operands required are determined.

examples:
- the monadic operator ROUND requires exactly one operand of type FLOAT.
- REQUEST requires an operand of type SEMA.

The number of operands may be used to classify operations. In particular this is done for the variety of algorithmic operations, where the terms "monadic" and "dyadic" are used to refer to operators requiring one and two operands, respectively.

Another criterion for the classification of operations is, whether they have a result or not.
An operations is said to have a result if its execution may be replaced by a value.

example: 2 + 3 * 4 is an operation with a result,
it may be replaced by this result 14.

Such operations are called "expressions".

Operations without result are just worked off. Of course, during this execution there may be changes of the contents of some objects that are accessed. But such an operation stands for "itself", it does not represent a value.

example: REQUEST S;
This is an operation without result, but it changes the value of the semaphore S.

Operations of this kind are called "statements".

2.0.1 Expressions

Expressions are defined as operations returning a result. In Basic PEARL this result is always a value.
Hence, expressions may be used in many parts of a Basic PEARL program, for example in

- assignations,
- parameter-lists,
- I/O-lists,
- repeat-statements,
- schedules ....

How expressions are evaluated is detailed in section 2.2.1.3. Sometimes an operation demands a special type of the result of an expression, or in other words - not every expressions is legal in this operation. This will syntactically be denoted by adding the type of the result to the general notation "expression-seven".
So, for example

bit-one-expression-seven

denotes an expression, the result of which has to be of type BIT(1).

2.0.2 Statements

Statements have been introduced as operations returning no result.
In Basic PEARL statements can only be part of an executable object like a procedure or a task.
Statements are executed in the sequence they are denoted unless this sequence is explicitely modified by a special category of statements, called "transfer-of-control-statements" (refer to section 2.2.4).
These "transfer-of-control" operations require "labelling" of statements, therefore the general syntax of statement is given by:

statement::=

[{ *label-identifier* :}$^{\cdots}$]
[*unlabelled-statement*] ;

'label-identifier' is viewed as implicite declaration of an object of type LABEL (refer to section 1.2.4). Any unlabelled statement may be preceded by an arbitrary number of labels.
'unlabelled-statement' is further classified as follows:

unlabelled-statement::=

{ *assignment*
begin-block
call-statement
return-statement
trf-of-ctrl-statement
realtime-statement
communication }

The different categories will be discussed in the following sections.
If 'label-identifier' and 'unlabelled-statement' are omitted, the semicolon remains, denoting a "no-operation", a "dummy-statement" with the semantics: do nothing, pass the control to the next statement in the sequence.

2.1 Structural Operations

2.1.0 General

In section 1.1 the structural objects module, division and block have been introduced.
This section deals with some properties of these objects during program execution.
In 2.1.1 a general outline is given, how a Basic PEARL program is executed.
Section 2.1.2 details how long objects attached to the different levels exist and where they may be used.
How the different types of blocks (procedures, tasks and begin-blocks) are executed is explained in the corresponding sections.

2.1.1 Program Execution

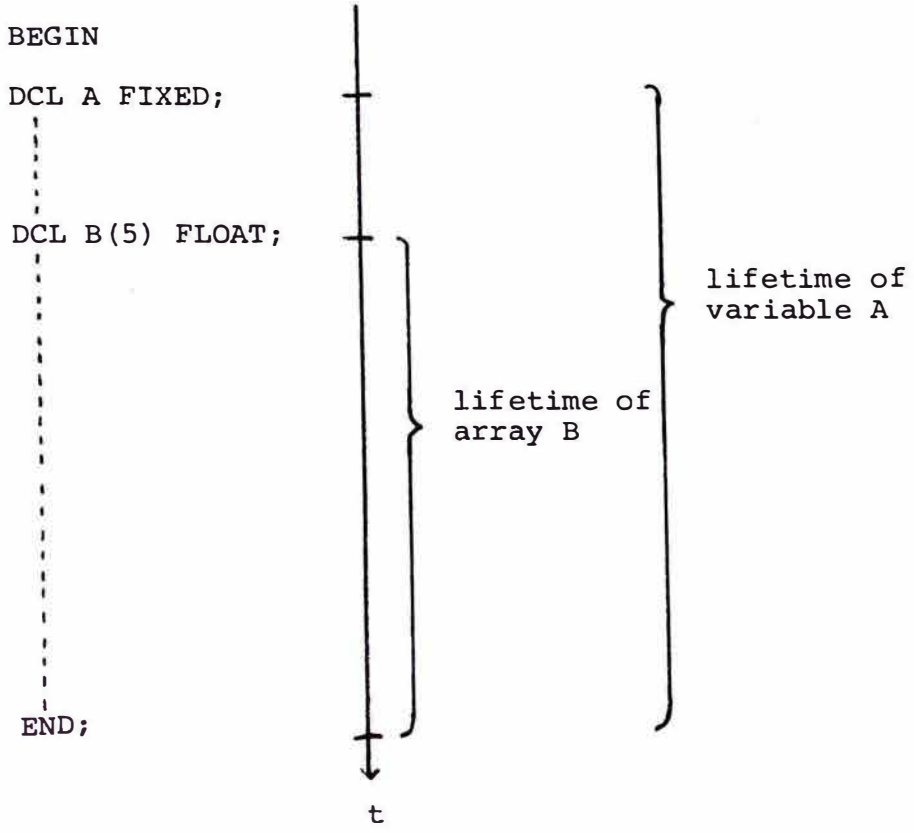
If a Basic PEARL program is to be executed, the SYSTEM-division is considered first.
All the specifications given in it are evaluated.
Then the PROBLEM-divisions are executed in an arbitrary-order.
In section 1.1.1 it has been stated that at least one module has to contain a declaration of a global task.
So there is one task to be activated.
Since there is no language feature permitting to start a task when none is active, the "initial" start (the start of the first task) has to be done by an external stimulus (e.g. by an operator).

2.1.2 Lifetime and Scope

In section 1.0 it has been described, how objects are generated by declarations or specifications. In this section it is to be detailed during which time the objects exist and from which part of the program they may be accessed (used).

The lifetime of an object is the time between (the execution of) its declaration and the execution of the block-end of the block, in which its declaration is contained.

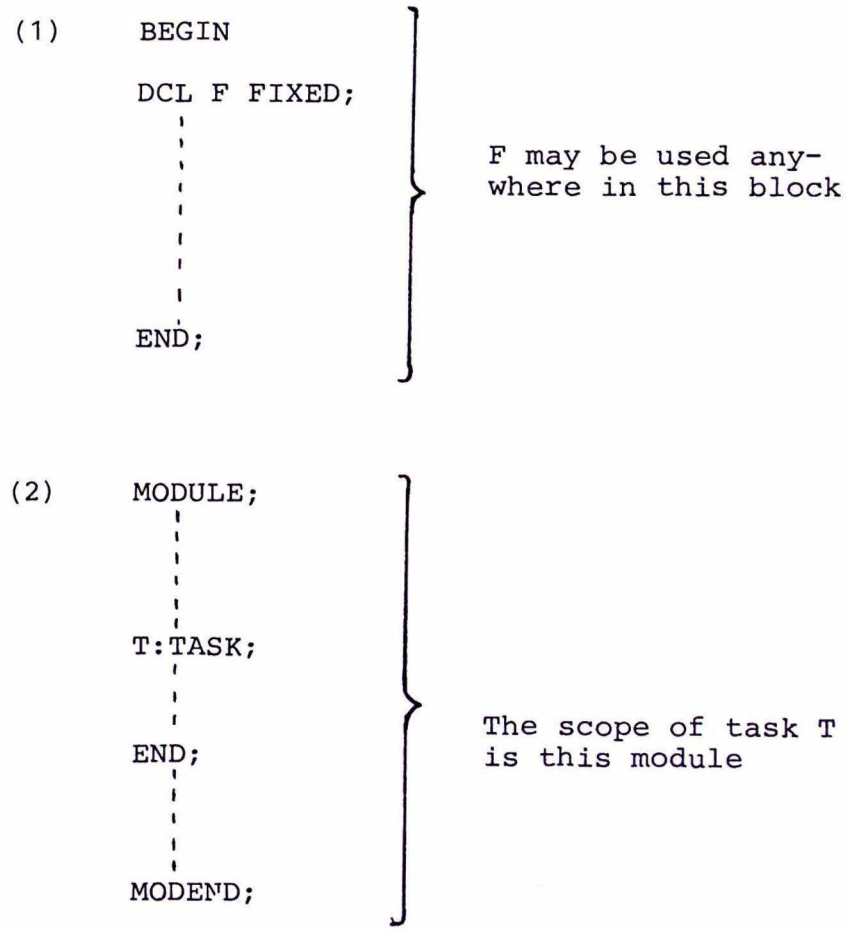
example:



All parts of the program (declarations, specifications, statements) where it is possible to access an object, are called the scope of this object.

In general the scope of an object is the block, in which it is declared.

example:



Some rules have to be obeyed:

- Except controls within the same scope,
  no two objects may have the same identification.

  example:

  (1)
```
MODULE;
DCL A CHAR(1);
   ¦
   ¦
A:TASK;
   ¦
   ¦
MODEND;
```

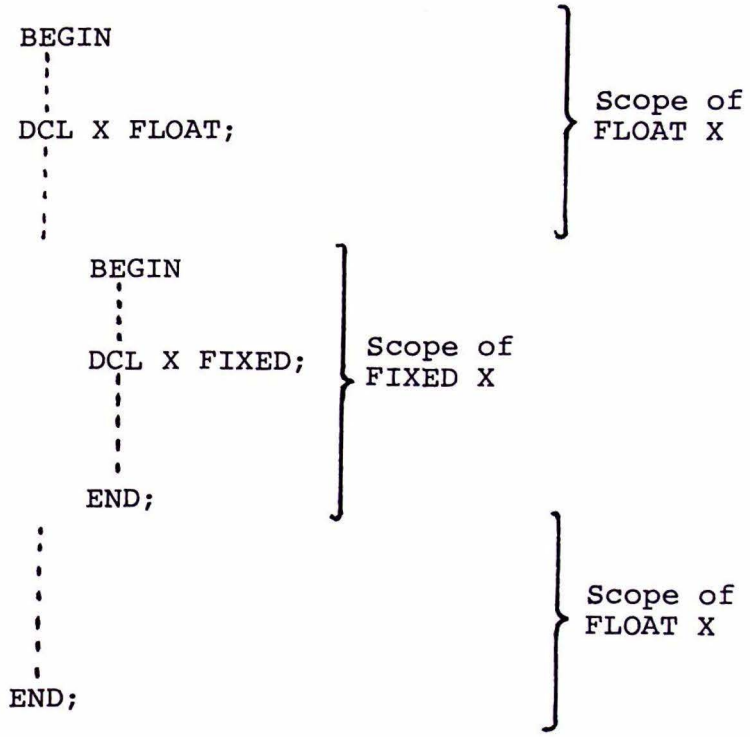
  This example is illegal: there must not be a variable A and a task A within one module!

  (2)
```
BEGIN
DCL (E,F) FIXED;
   ¦
   ¦
   ¦
PUT E TO LPRINT BY F(F);
   ¦
   ¦
END;
```

  This example is legal, since the first "F" in the put-statement denotes a control, the second one a variable of type FIXED.

- The scope of an object may be restricted, if there is an enclosed block containing (a declaration of) another object with the same identification.

example:



The scope of the FIXED-variable X is the inner block. The scope of the FLOAT-variable X is the outer block except the inner one, since there is another object with the same name.

- The scope of objects declared at module-level may be extended by using the global-attribute "GLOBAL" (cf. section 1.1.3).
  The scope of an object declared at module-level is a priori the module. If this object has the global-attribute the scope may be extended to other modules by specifying this object in the other modules.

example:

```
MODULE;
   :
PROBLEM;
   :
DCL F FIXED GLOBAL;
   :
   :
MODEND;
```

a priori scope of F

```
MODULE;
   :
PROBLEM;
   :
SPC F FIXED GLOBAL;
   :
MODEND;
```

extended scope of F

2.2 Algorithmic Operations

2.2.0 General

In this chapter all the languages features will be discussed that are commonly considered as "algorithmic". Section 2.2.1 deals with operations on objects of basic types, especially the fundamental operation of assigning a value to an object.
In section 2.2.2 the call of a procedure is explained together with the mechanisms, how to get data into and out of a procedure.
Section 2.2.3 details, how compound objects (refer to section 1.2.3) may be accessed.
A very important group of statements is summarized by the term 'transfer of control'. These operations control the sequence of execution of statements. They are detailed in section 2.2.4.

2.2.1 Basic Operations

2.2.1.0 General

The operations detailed in this section are essentially operations using objects of the basic types (refer to section 1.2.1).
Section 2.2.1.1 discusses the assignment-statement, which is a fundamental operation in each programming language.
Section 2.2.1.2 lists all the standard operators of Basic PEARL and explains, how they may be used in expressions.
The final section 2.2.1.3 details, how complex expressions are evaluated.

2.2.1.1 Assignation

Assignation is one of the fundamental operations. It explicitely allows to modify the contents of an object. In Basic PEARL assignation is only defined for objects of basic types.
However, assignation, even if defined for an object, may be inhibited, if access to this object is restricted by the INV-attribute (either by declaration or specification).

assignment::=

symbol { := / = } *expression-seven*

The value of 'expression-seven' is assigned to the 'symbol' left of the assignation-operator ':=' (or '=').
Some rules for compatibility have to be obeyed:

- the result of 'expression-seven' and 'symbol' must be of the same type. (There is only one exception: FIXED-objects may be assigned to FLOAT-objects).

- as already mentioned above, 'symbol' must not have the INV-attribute.

- assignation of strings: if the string on the left side is longer than the one on the right side it is filled up with blanks or zeroes, respectively. If it is shorter, it results in an error.

example:

```
DCL S CHAR(9);
 ⋮
S := 'EXAMPLE';
```

This results in assignation of
'EXAMPLE␣␣' to S

- assignation of numbers: The precision of the left side must not be smaller than the precision of the right side.

Some examples for assignations:

```
DCL (I,J,K) FIXED INIT (1,2,3),
    PI INV FLOAT,
    B BIT(8),
    B1 BIT(4) INIT ('1111'B),
    F FLOAT;
P : PROC (A1 FIXED, A2 FLOAT) RETURNS (FIXED);
     ⋮
    END;
    ⋮
I := J * 3;            /* I becomes 6  */
B := B1 CAT '01'B;     /* B becomes '11110100'B  */
PI:= 3.14;             /* illegal, since PI has the INV-
                          attribute  */
K := P(I,F);           /* The result of procedure P is assigned
                          to K  */
```

2.2.1.2 Standard Operations

2.2.1.2.0 General

In this section the operators are described, that may be used in Basic PEARL expressions.
They are predefined and can only be used with the types of operands given in the following tables.
Using other types of operands results is an error, just as using a wrong type for the result.
If the precision of the result is not compatible with the one needed, in general a signal is raised.

2.2.1.2.1 Monadic Operators

An operator having exactly one operand is called a "monadic operator".
In Basic PEARL the following monadic operators are provided:

monadic-operator::=

{ + | - | *NOT* | *ABS* | *SIGN* | *ROUND* | *TOFIXED* | *TOFLOAT* | *TOBIT* | *TOCHAR* | *ENTIER* }

The following table lists for each monadic operator the type(s) that its operand must have, the corresponding type of the result and the semantics.

table 5: monadic operators

syntax	type of operand a	type of result c	semantics
+ a	FIXED (p) FLOAT (p) DURATION	FIXED (p) FLOAT (p) DURATION	c := +a (monadic plus)
- a	see +	see +	c := -a (monadic minus)
NOT a	BIT (k)	BIT (k)	inversion of all bits
ABS a	FIXED (p) FLOAT (p) DURATION	FIXED (p) FLOAT (p) DURATION	c := |a| (absolute value)
SIGN a	FIXED (p) FLOAT (p) DURATION	FIXED (1)	$c := \begin{cases} +1 & \text{if } a > \emptyset \\ \emptyset & \text{if } a = \emptyset \\ -1 & \text{if } a < \emptyset \end{cases}$
ROUND a	FLOAT (p)	FIXED (p)	rounding to the nearest integer

table 5, continued: monadic operators

syntax	type of operand a	type of result c	semantics
TOFIXED a	CHARACTER (1)	FIXED (p)	c := integer associated with a CHARACTER; that integer and p are implementation dependent
	BIT (k)	FIXED (p)	c = interpretation of a bit-string as an integer
TOFLOAT a	FIXED (p)	FLOAT (p)	c becomes the real number corresponding to a
TOBIT a	FIXED (k)	BIT (p)	conversion of a to BIT (p); p is implementation dependent
TOCHAR a	FIXED	CHARACTER (1)	conversion of a to a CHARACTER (1) value, if possible
ENTIER a	FLOAT (p)	FIXED (p)	c := nearest integer not greater than a

2.2.1.2.2 Dyadic Operators

A "dyadic operator" is an operator with two operands. Dyadic operators are classified according to their "precedence".

prec-1-operator::=

{ ** / UPB / FIT }

prec-2-operator::=

{ * / / / >< / CAT / // }

prec-3-operator::=

{ + / - / <> / CSHIFT / SHIFT }

prec-4-operator::=

{ < / LT / > / GT / <= / LE / >= / GE }

prec-5-operator::=

{ == / EQ / /= / NE }

prec-6-operator::=

AND

prec-7-operator::=

{ OR / EXOR }

The precedence of an operator becomes important when several operators are used within one expression. Then it is one criterium to determine how such an expression is worked off. This is further detailed in section 2.2.1.3. As for monadic operators the following table lists the syntax of dyadic operators, the types of operands and the corresponding type of the result. Besides this a short explanation of semantics is given.

table 6: dyadic operators

syntax	type of operand a	type of operand b	type of result c	semantics
a ** b	FIXED (p)	FIXED (q)	FIXED (p)	c:= {a*a*...*a (b times), if b > Ø {1 , if b = Ø
	FLOAT (p)	FIXED (q)	FLOAT (p)	c:= {a*a*...*a (b times), if b > Ø {1.Ø , if b = Ø {1/(a*a*...*a)(|b| times), if b < Ø
a UPB b	FIXED (p)	array	FIXED	c := upper bound of the a-th dimension of b, if it exists
a FIT b	FIXED (p) FLOAT (p)	FIXED (q) FLOAT (q)	FIXED (q) FLOAT (q)	The precision of a is changed to that one of b.
a * b	FIXED (p) FIXED (p) FLOAT (p) FLOAT (p) FIXED (p) DURATION	FIXED (q) FLOAT (q) FIXED (q) FLOAT (q) DURATION FIXED (p)	FIXED (r) FLOAT (r) FLOAT (r) FLOAT (r) DURATION DURATION	c := a * b (multiplication) r := max (p,q)

table 6, continued: dyadic operators

syntax	type of operand a	type of operand b	type of result c	semantics
a / b	FIXED (p) FLOAT (p) FIXED (p) FLOAT (p) DURATION	FIXED (q) FLOAT (q) FLOAT (q) FIXED (q) FIXED	FLOAT (r) FLOAT (r) FLOAT (r) FLOAT (r) DURATION	$c := \frac{a}{b}$, if b ≠ ∅ (division) r := max (p,q)
a // b	FIXED (p)	FIXED (q)	FIXED (r)	c := ENTIER ABS (a/b) * SIGN (a/b) if b ≠ ∅ (integer division) r := max (p,q)
a { > < CAT } b	CHAR (n) BIT (n)	CHAR (m) BIT (m)	CHAR (k) BIT (k)	concatenation of two strings, c contains string a, followed by string b. k := m+n
a + b	FIXED (p) FIXED (p) FLOAT (p) FLOAT (p) DURATION DURATION CLOCK	FIXED (q) FLOAT (q) FIXED (q) FLOAT (q) DURATION CLOCK DURATION	FIXED (r) FLOAT (r) FLOAT (r) FLOAT (r) DURATION CLOCK CLOCK	c := a + b (addition) r := max (p,q)

table 6, continued: dyadic operators

syntax	type of operand a	type of operand b	type of result c	semantics
a - b	FIXED (p) FIXED (p) FLOAT (p) FLOAT (p) DURATION CLOCK CLOCK	FIXED (q) FLOAT (q) FIXED (q) FLOAT (q) DURATION DURATION CLOCK	FIXED (r) FLOAT (r) FLOAT (r) FLOAT (r) DURATION CLOCK DURATION	c := a - b (subtraction) r := max (p,q)
a SHIFT b	BIT (k)	FIXED (p)	BIT (k)	a is shifted b steps to the left, if b > Ø, and to the right, if b < Ø. zeroes are pulled after
a { < > / CSHIFT } b	BIT (k)	FIXED (p)	BIT (k)	cyclic shift of a b steps to the left, if b > Ø b steps to the right, if b < Ø
a { < / LT } b	FIXED (q) FIXED (q) FLOAT (q) FLOAT (q) CLOCK DURATION	FIXED (r) FLOAT (r) FIXED (r) FLOAT (r) CLOCK DURATION	BIT (1)	comparison-operator "less than" c := { '1'B, if true 'Ø'B, if false comparison-precision: p = max (q,r) If one operand is FLOAT, the other is converted to FLOAT

table 6, continued: dyadic operators

syntax	type of operand a	type of operand b	type of result c	semantics
a { > / GT } b	see <	see <	BIT (1)	comparison-operator "greater than"
a { <= / LE } b	see <	see <	BIT (1)	comparison-operator "less than or equal to"
a { >= / GE } b	see <	see <	BIT (1)	comparison-operator "greater than or equal to"
a { == / EQ } b	see < CHAR (q) BIT (q)	see < CHAR (r) BIT (r)	BIT (1)	comparison-operator "equal to"
a { /= / NE } b	see < CHAR (q) BIT (q)	see < CHAR (r) BIT (r)	BIT (1)	comparison-operator "not equal to"

table 6, continued: dyadic operators

syntax	type of operand a	type of operand b	type of result c	semantics
a AND b	BIT (k)	BIT (m)	BIT (m)	if m > k, a is extended on the right end by '0'B.
a OR b	BIT (k)	BIT (m)	BIT (m)	Then corresponding single bits are logically connected according to the following table:
a EXOR b	BIT (k)	BIT (m)	BIT (m)	

a	b	a AND b	a OR b	a EXOR b
'1'B	'1'B	'1'B	'1'B	'0'B
'1'B	'0'B	'0'B	'1'B	'1'B
'0'B	'1'B	'0'B	'1'B	'1'B
'0'B	'0'B	'0'B	'0'B	'0'B

2.2.1.3 Evaluation of Expressions

Since the operands of an expression may in term be expressions, rules have to be established, how such a complex expression is evaluated, i.e. in which order partial operations are to be considered.
There are two criteria, familiar from ordinary algebra, to achieve ordering of operations:

- explicitly by enclosing partial operations in brackets
- implicite bracketing by introducing precedence-levels for operators.

There are seven precedence-levels, numbered from 1 to 7. Number 1 denotes the highest precedence, number 7 the lowest one. Operations containing operators with higher precedence are seen to be (implicitly) enclosed in brackets before operations with lower-precedence-operators.

example: a + b * c

* is of precedence two, + is of precedence three, therefore the partial operation b * c is implicitly enclosed in brackets first, then the operation with + :

(a + (b * x))

If there are several operations with the same precedence, ordering is achieved by the following rules:

- for precedence 2 to 6 implicite bracketing is done from left to right.

  example at precedence-level 3:

  a + b - c - d

  is implicitly enclosed in brackets in the following way:

  (((a + b) - c) - d)

- for precedence 1 implicite bracketing is done from right to left, due to mathematical conventions.

  example:

  a**b**c

  is equivalent to

  (a**(b**c))

- monadic operators are considered to be at the same level as prec-1-operators. The same rules for bracketing are applied, too.

  example:

  -a**b

  is equivalent to

  (- (a**b))

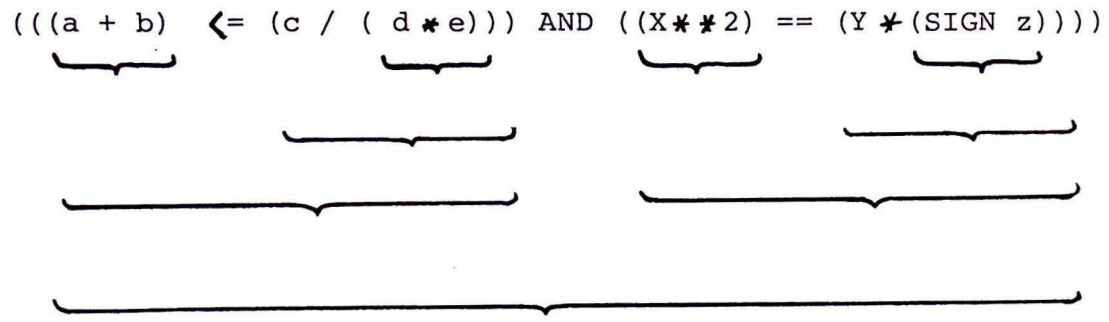
Now, considering all the (explicite and implicite) brackets, the rule for evaluation of a complex expression is quite simple:

a complex expression is worked off, starting with evaluation of the partial operation(s) enclosed in the innermost pair(s) of brackets.

example:

a + b <= c / (d * e) AND X **2 == Y * SIGN z

After adding all the implicite pairs of brackets this expression looks like:



The four lines of braces denote the steps of execution. Braces in one line indicate that these partial operations may be executed in parallel.

The syntax of expressions reflects all the rules given above. Since the "most loosely bound" expression contains an operator of precedence 7 the most general form of an expression is called 'expression-seven'.

expression-seven::=

[*expression-seven prec-7-operator*]

expression-six

expression-six::=

[*expression-six prec-6-operator*]
expression-five

expression-five::=

[*expression-five prec-5-operator*]
expression-four

expression-four::=

[*expression-four prec-4-operator*]
expression-three

expression-three::=

[*expression-three prec-3-operator*]
expression-two

expression-two::=

[*expression-two prec-2-operator*]
expression-one

The syntax of 'expression-one' contains the different rules for (implicite) bracketing of 'prec-1-operator' and 'monadic-operators'.

expression-one::=

{ *primitive-expression* [*prec-1-operator expression-one*] }
monadic-operator expression-one

primitive-expression::=

{ *symbol-or-constant* }
(*expression-seven*)

The rule for 'primitive-expression' contains inturn 'expression-seven', but explicitly enclosed in brackets.

symbol-or-constant::=

$$\left\{\begin{matrix}\textit{symbol} \\ \textit{constant-denotation}\end{matrix}\right\}$$

symbol::=

identifier [*expression-seven-pack*]
[. *identifier*]
[. *BIT integer-in-brackets*]

'symbol' may be:

- a simple identifier
- an element of an array
- an element of a structure
- a bit out of a bit-string

2.2.2 Procedure Calls

2.2.2.0 General

A procedure-block is executed through a special operation, a "call".
A call may be performed in two ways:

- either explicitly through a 'call-statement'
- or implicitly as operand in an expression.

(1) A procedure returning no result is called through a 'call-statement'.

call-statement::=

CALL procedure-identifier
[expression-seven-pack]

'expression-seven-pack' denotes the actual parameters. How they are passed to a procedure is discussed in the next section.

example:

```
P:PROCEDURE (A FIXED, B( ) FLOAT IDENT);
    :
  RETURN;
  END;
```

This procedure P may be called through:

```
CALL P (X,Y);
```

(2) Calling a procedure within an expression is only possible for procedures returning a result, i.e. function-procedures.
In this case the call is denoted by

procedure-identifier [*expression-seven-pack*]

whereby 'expression-seven-pack' denotes the actual parameter-list.

example:

```
FP:PROCEDURE (B( ) FLOAT IDENT) RETURNS (FLOAT);
      ¦
      ¦
   RETURN (B(I));
   END;
```

This procedure returns a value of type FLOAT.
So it may be called within an expression:

```
X:=X + FP(Y);
```

As already indicated in the example above, a procedure-block is left through execution of a 'return-statement'.

return-statement::=

RETURN [*(expression-seven)*]

With procedures returning no result just the keyword 'RETURN' is used. This indicates:

- stop the execution of the procedure,
- return to the point where it was called from.

For function-procedures (declared with the result-attribute 'RETURNS (simple-mode)') 'expression-seven' in brackets has to be supplied.
After evaluation 'expression-seven' must agree in type with 'simple-mode'.
The effect of the statement is the following:

- stop the execution of the function-procedure
- pass the result to the point of the call
- return the control to this point.

2.2.2.1 Passing Actual Parameters

2.2.2.1.0 General

In a procedure-call the list of actual parameters is denoted by 'expressions-seven-pack'.

expression-seven-pack::=

({ , ' expression-seven ''' })

The elements of this list have to be compatible with the elements of the formal-parameter-list in the respective procedure-declaration.
The relation between formal and actual parameters may be established in two different ways:

- either by the initial-mechanism
- or by the identical-mechanism

The initial-mechanism is default. It passes the value of an object to the procedure.
The identical-mechanism is characterized by the keyword IDENT. It passes a reference to the object to the procedure. Both mechanism are detailed in the following sections.

2.2.2.1.1 Initial-Mechanism

The initial-mechanism of passing actual parameters is commonly known as "call by value".
For each parameter passed with this mechanism a new object local to the procedure is created. This new object is initialized with the current value (i.e. the value at the moment of the procedure-call) of the actual parameter.
In Basic PEARL the initial-mechanism may only be used for simple-objects, i.e. objects of type:

FIXED
FLOAT
CLOCK
DURATION
BIT
CHARACTER

with any precision or length, respectively.
For objects of these types the initial-mechanism is defaulted. This means, if the formal-parameter-list contains elements of these types without further keyword (i.e. without "IDENT"), the corresponding actual parameters are passed to the procedure by the initial-mechanism.

example:

```
P:PROCEDURE (A FLOAT,
             B INV FIXED,
             C CLOCK,
             D CHAR(5));
  :
END;
```

This procedure may be called, for example, as:

```
BEGIN
  :
DCL (E,F) FLOAT INIT (7.32,14.08);
DCL I INV FIXED INIT (4);
DCL T CLOCK INIT (10:15:28);
DCL STR CHAR(5) INIT ('PEARL');
  :
  :
CALL P(F/E,18+I * 4,T,STR);
  :
END;
```

In this example all parameters are passed with the initial-mechanism.

Note, that the actual parameters may be expressions.

2.2.2.1.2 Identical-Mechanism

With the identical-mechanism no new object is created, but an existing object is made accessible.
In other words, similar to the global-mechanism the scope of an object is extended by this mechanism (since, in general, the procedure-block is not part of the scope of the actual parameters).
The identical-mechanism is the only way to change the values of objects out of the procedure.
(Note: since such an object may be accessed from other parts of the program, too, side-effects have to be obeyed).
Syntactically formal parameters have to be marked with the keyword 'IDENT' if this mechanism is to be used.
The following objects may be passed with the identical-mechanism:

- all simple types as listed in the previous section
- arrays (built up of simple types)
- structures
- arrays of structures
- dations
- arrays of dations

Note: the actual parameters must be identifiable objects. Contrary to the initial-mechanism no constant-denotations (e.g. 4.ϕ) or 'expression-seven' containing operators (e.g. 4 * I + K) must be used.

example:

```
R:PROCEDURE (F FIXED IDENT,
             S STRUCT [A CHAR, B FLOAT] IDENT,
             D DATION INOUT ALPHIC IDENT);
  :
END;
```

In this example all parameters are passed by the identical-mechanism. A call may be:

```
DCL J INV FIXED INIT(3);
DCL S1 STRUCT [K1 CHAR, K2 FLOAT];
DCL T DATION OUT ALPHIC CREATED(LO3);
  :
CALL R(J,S1,T);
```

The relation between formal and actual parameters is established in a similar way as between declaration and specification of global objects, i.e. they must agree in their type. If the parameter is a structure the components must agree. For arrays the number of dimensions must be the same, the upper bounds are taken from the actual parameter.
As for specifications access attributes may be restricted in the call (for J and T this is indicated in the example above). The coercion rules are given in the sections where the objects are introduced.

2.2.2.2 Returning a Result

A function-procedure returns a result to the point, where it was called from.
As detailed in section 1.2.2 a function-procedure must be declared with the result-attribute

RETURNS (simple-mode)

and must be left by executing a return-statement

RETURN (expression-seven)

'simple-mode' in the result-attribute indicates that only simple objects (refer to section 1.2.1) may be returned. The types of 'simple-mode' and (the result of) 'expression-seven' must coincide.

The effect of the call of a function-procedure - as far as the result is concerned - may be viewed as replacing the call by 'expression-seven' of the return-statement. This implies that syntactically the call may appear anywhere, where 'expression-seven' is legal.

example:
```
DCL LOW,SCALE FIXED;
DCL A(25) FIXED;
  :
MIN:PROCEDURE (F ( ) FIXED IDENT) RETURNS (FIXED);
    DCL RES FIXED;
      :
    RETURN(RES);
    END;
      :
    LOW:=MIN(A)* SCALE;
      :
```

2.2.3 Operations on Compound Objects

2.2.3.0 General

In section 1.2.3 compound objects have been introduced. In the following sections it is detailed how these objects or parts of these objects, respectively, may be accessed. Section 2.2.3.1 describes access to arrays, section 2.2.3.2 deals with access to structures. In Basic PEARL bit-strings, although considered as simple objects, may be accessed in a similar way as structures. Therefore bit-string-selection is explained in section 2.2.3.3.

2.2.3.1 Access to Arrays

In Basic PEARL there are three ways of "using" an array:

- accessing single components,
- using slices or
- the whole array

(1) Single components are denoted in the following way:

identifier expression-seven-pack

'identifier' denotes the name of an array.
'expression-seven-pack' may contain one, two or three expression - seven enclosed in brackets.
They represent the indices. Evaluation of expression-seven has to result in a positive integer.

expression-seven-pack::=

(, ' expression-seven ' ' ')

examples for selections of single components:

A(5) denotes the 5th element of an one-dimensional array

B(I+J,3,J * 2)
denotes one element out of a three-dimensional array

(2) Slices

A slice is an abbreviated denotation for a continous part of an array. In Basic PEARL slices are only legal within transfer-operations and only for one-dimensional arrays.

They are denoted by

identifier (simple-integer-constant-denotation
:simple-integer-constant-denotation)

'identifier' is again the name of an array. The two 'simple-integer-constant-denotation's denote the first and the last index of the slice.

example: A(2:6) is a slice with 5 elements

In a transfer-operation A(2), A(3), A(4), A(5), A(6) is an equivalent notation for this slice.

(3) If the array is to be accessed as a whole it is sufficient to use the 'identifier' denoting the name of the array. In Basic PEARL the array as a whole may be used in a transfer-operation, in the parameter-list of a procedure-call or as operand for the dyadic operator "UPB" (cf. section 2.2.1.2.2).

example: a procedure declared

```
MAX:PROCEDURE (A(,)FIXED IDENT);
    :
```

may be called by

```
CALL MAX(F);
```

if F has been declared

```
DCL F(5,7) FIXED;
```

2.2.3.2 Access to Structures

Accessing a structure can only be done by selection, i.e. by choosing one component out of the structure.

Selection of a component of a structure is achieved by

identifier [*expression-seven-pack*]
.*identifier*

The first 'identifier' denotes the name of the structure. Since arrays of structures are legal indexing is provided by 'expression-seven-pack'.
The second 'identifier' following "." is the name of the chosen component.

examples:

- DCL S STRUCT [I FIXED,(X,Y) FLOAT,D DURATION];

  The following selections are possible:

  S.I
  S.X
  S.Y
  S.D

- DCL F(3,10) STRUCT [A CHAR(5),(B,C) FIXED];

  Selections may be:

  F(1,1).A
  F(1,7).B
  F(3,1).C

2.2.3.3 Bit-string-selection

Although bit-strings are simple objects a selection-operation is provided for them. This selection is to single out individual bits of the string. Syntactically the selection is similar to the selection of components of structures.

Bits are selected by

identifier [*expression-seven-pack*]
[*.identifier*]
.BIT integer-in-brackets

The first 'identifier' either denotes the name of a bit-string or the name of a structure containing a bit-string.
'expression-seven-pack' is provided for indexing.
The second (optional) 'identifier' serves to select a bit-string which is component of a structure.
'integer-in-brackets' following ".BIT" is the number of the bit that is to be selected. (The left-most bit in the string is number 1, the right-most bit has the highest number).

examples:

- DCL BSTR BIT(5) INIT ('11101'B);
  Selection of
  BSTR.BIT(4)
  results in '0'B.
- DCL B3STR BIT(9) INIT ('205'B3);
  Selection of
  B3STR.BIT(2)
  results in '1'B.
- DCL S STRUCT [A BIT(3), B FIXED] ;
  The first bit of the component A is selected by S.A.BIT(1)

2.2.4 Transfer of Control

2.2.4.0 General

Within a task- or procedure-body statements are usually executed in the sequence they are written down. But there are some statements that allow to modify this sequence. They are called "transfer-of-control statements".

trf-of-ctrl-statement::=

{ *goto-statement*
conditional-statement
case-statement
repeat-statement }

2.2.4.1 Goto-Statement

A 'goto-statement' serves to unconditionally transfer control to a labelled statement.

goto-statement::=

GOTO label-identifier

'label-identifier' denotes the target of the jump, i.e. control is transferred to the statement which is labelled with this identifier.
With a 'goto-statement' one can transfer control within the block the statement is attached to, or to an enclosing (outer) block. One must not jump into (inner) blocks.
Leaving a procedure- or task-body by a 'goto-statement' is not possible.

2.2.4.2 Conditional-Statement

The 'conditional-statement' allows different continuation of the statement sequence depending on a condition.

conditional-statement::=

IF bit-one-expression-seven
THEN {*statement* ...}
[*ELSE* {*statement* ...}]
FIN

'bit-one-expression-seven' is an expression, the result of which has to be of type BIT(1).

bit-one-expression-seven::=

expression-seven

Together with IF it denotes the condition. The condition is said to be true if the expression results in '1'B, it is false, if the result is 'Ø'B.

If the condition is true the statement(s) between THEN and ELSE is (are) executed. (If ELSE is missing, between THEN and FIN, resp.).

If the condition is false the statement(s) between ELSE and FIN is (are) executed. If this branch is empty (i.e. does not contain any statement) ELSE has to be omitted, too.

examples:

```
(1)   IF  a < b
            THEN  min:=a;
            ELSE  min:=b;
      FIN;
```

```
(2)   IF  value > max
            THEN  lamp:='1'B;
      FIN;
```

After executing one of the two branches control is sequentially passed to the statement following FIN. There are no restrictions for the statements following THEN and ELSE, so that nesting of conditional statements is possible.

example:

```
IF  a = 3
    THEN IF x ≤ y
            THEN  m:=1;
            ELSE  m:=2;
         FIN;
    ELSE  m:=-1;
FIN;
```

2.2.4.3 Case-Statement

The 'case-statement' allows to discriminate more than two alternatives.

case-statement::=

CASE integer-expression-seven

{*ALT statement*...}...

FIN [*OUT statement*...]

'integer-expression-seven' is an expression, the result of which is of type FIXED.

integer-expression-seven::=

expression-seven

Each alternative that may be chosen is preceded by the keyword ALT. It may consist of one or several statements. (At least it must contain the empty statement ";").
The alternatives are considered to be numbered from 1 to n. If the value of 'integer-expression-seven' is n control is transferred to the n-th alternative.
The n-th alternative starts with the statement following the n-th ALT and ends with the next ALT, or, if there is no more ALT with OUT. If also OUT is omitted it ends with FIN.

If the value of 'integer-expression-seven' is less than 1 or greater the number of ALT's in the 'case-statement' the statements between OUT and FIN, if existent, are executed.

After execution of one alternative control is passed to the statement following FIN.

example:

```
CASE    i
  ALT   x:=3;
  ALT   m:=m+1;
  ALT   x:=2; y:=1;
  OUT   m:=5;
FIN
```

If, for example, i=2 the statement m:=m+1 is executed.
If i=17 the statement m:=5 is executed.

2.2.4.4 Repeat-Statement

The 'repeat-statement' allows to execute a statement-sequence several times. It consists of a loop-body and a mechanism to control the repeated execution of this loop-body. This mechanism may contain a counter and/or a condition.

repeat-statement::=

[*FOR identifier*]
[*FROM integer-expression-seven*]
[*BY integer-expression-seven*]
[*TO integer-expression-seven*]
[*WHILE bit-one-expression-seven*]
REPEAT [*;*] *block-tail*

The repeat-statement is to be considered as a begin-block (cf. section 1.1.1.4).
The 'identifier' following FOR is called loop-index. It is implicitly declared and may only be used within the 'repeat-statement'. If the phrase 'FOR identifier' is omitted the loop-index is defaulted by the system, but then it cannot be accessed within the loop-body.

'integer-expression-seven' following FROM denotes the initial-value of the loop-index. If it is omitted it is defaulted to 1.

'integer-expression-seven' following BY denotes the step-width by which the loop-index is varied (incremented or decremented). If it is omitted it is defaulted to 1.

'integer-expression-seven' following TO denotes the end-value for the loop-index. On exceeding this value the loop is terminated. There is no defaulting if it is omitted.

'WHILE bit-one-expression' denotes a condition. The loop-body is executed as long as this condition is true (i.e. evaluation of 'bit-one-expression-seven' results in "1"B). If it is omitted the condition is defaulted to "true".

All the expressions in the control-mechanism of the 'repeat-statement' are evaluated together before the 'repeat-statement' is executed. They must not contain the loop-index.

The loop-body is denoted by 'block-tail'. It may contain local-declaration and statements. Within the loop-body the loop-index may be used, but not as left-hand-side of an assignation.

The following flow chart shows an equivalent representation of the repeat-statement:

```
FOR i FROM start BY step TO end
    WHILE valid REPEAT group END;
```

The begin-blocks are enclosed in dotted lines.

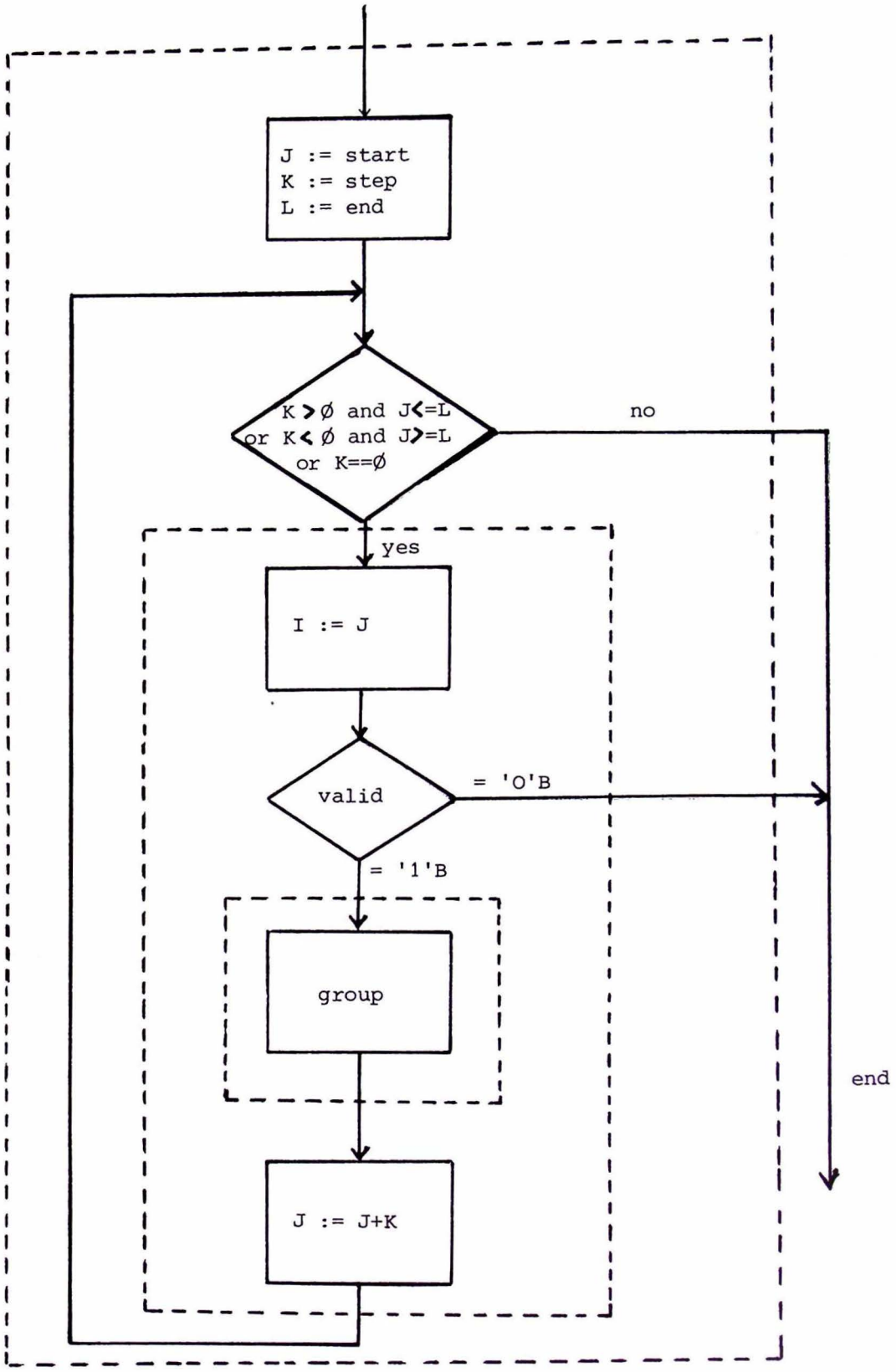


fig. 6: graphic representation of a repeat-statement

2.3 Communication Operations

2.3.0 General

This chapter explains, how the communication objects data-station and control are used in I/O-operations.

There are two categories of such operations: the preparation of a "dataway" and the actual transfer.

communication::=

$$\left\{\begin{matrix}\textit{dataway-operation}\\ \textit{transfer-operation}\end{matrix}\right\}$$

'dataway-operation' comprises the creation of a dataway, which in Basic PEARL is done within the declaration of a data-station. Then the dataway has to be synchronized before an actual I/O-statement is executed. Synchronization is achieved by the open- and close-statement.
Dataway operations are detailed in section 2.3.1.

Transfer-operations are classified according to

- data representation
- transfer direction.

If the PEARL-objects are transformed into alphic or basic symbols, GET/PUT and TAKE/SEND resp. are used, if they are not changed during to transport, READ/WRITE is used. Syntax and semantics of these operations will be detailed in 2.3.2.

2.3.1 Dataway-Operations

2.3.1.0 General

Prior to the use in a transfer operation some administrative operations have to be carried out.
Each user-defined data-station must be connected with a system-defined data-station. This operation is called dataway-construction.
Furthermore the access to such a dataway has to be synchronized.

dataway-operation::=

$$\left\{\begin{array}{l}\textit{dataway-construction}\\ \textit{dataway-synchronization}\end{array}\right\}$$

'dataway-construction' will be detailed in section 2.3.1.1, 'dataway-synchronization' in section 2.3.1.2.

2.3.1.1 Construction of Dataways

In section 1.3.1.2.1 (SYSTEM-division) it has been explained, how system-defined data-stations are connected.
Now, within the PROBLEM-division each user-defined data-station has to be connected to a system-defined data-station before it can be used.
In Basic PEARL the construction of a dataway is a static feature. It can only be done within the declaration of a data-station (cf. section 1.3.1.0).

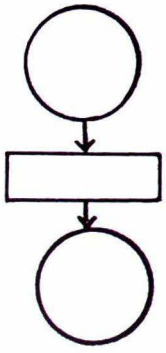
dataway-construction::=

CREATED (dation-identifier [index])

index::=

integer-in-brackets

'dation-identifier' is the name of a system-defined data-station introduced in the system-division.
On constructing a dataway restrictions specified in the implementation handbook have to be obeyed.
I.e. in this handbook one can find the attributes a user-defined DATION may have, so that a connection to the system-defined DATION is possible. These attributes are mapped upon the system-defined DATION by an inner (i.e. system-defined) interface.



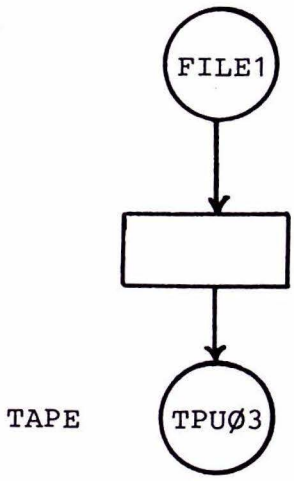
user-defined DATION

inner interface

system-defined DATION

example:

```
SYSTEM;
    :
    TAPE:TPUØ3;

PROBLEM;
    :
    :
    SPC TAPE DATION INOUT ALPHIC CONTROL (ALL);
    DCL FILE1 DATION INOUT FIXED(10, 50)
              FORWARD CONTROL (ALL)
              CREATED (TAPE);

    :
    :
```



FILE1 ist created upon TPUØ3, which has been named TAPE

A system - interface maps the attributes of FILE1 upon TAPE.

2.3.1.2 Synchronization of Dataways

Before a DATION can be used in transfer-operations the access rights must be arranged.
In Basic PEARL several tasks may access a DATION at the same time. No task can reserve a DATION exclusively, only shared access is possible.
Access to a DATION is guaranteed after the DATION has been "opened". The opposite operation is "closing" a DATION.

dataway-synchronization::=

{ *open-statement* | *close-statement* }

Not every task acessing a DATION ~~must~~ needs to issue an 'open-statement' (or 'close-statement', resp.) for this DATION. But at least one "opening" must have been done before access to the DATION is possible.

open-statement::=

OPEN dation-identifier [*index*]
[*BY open-control-list*]

index::=

integer-in-brackets

open-control-list::=

{,*open-control* ...}

'dation-identifier' denotes the DATION, that is to be opened. (If it is a system-defined data-station, specification of an index is possible).
'open-control-list' contains open-controls, separated by commas. Some open-controls, which are common to all implementations of Basic PEARL have been explained in section 1.3.2.1.1.
Beyond that there may be other implementation-dependent open-controls. Their names and the semantics can be found in the implementation handbook.

close-statement::=

CLOSE dation-identifier [*index*]
[*BY close-control-list*]

close-control-list::=

{,*close-control*...}

The 'close-statement' indicates that the DATION denoted by 'dation-identifier' is to be released again.
'close-control-list' may contain implementation-dependent information. Therefore 'close-controls' are not explained in Basic PEARL, one may find them in the implementation handbooks.

The "opening" task need not be the one "closing" the DATION.

(e.g.: TASK A opens DATION D,
TASK A, B and C access this DATION in I/O-operations,
TASK B closes DATION D).

But the number of open- and close-statements for one DATION has to be in accordance to set the DATION free. (One could imagine a counter for each DATION initialized by zero. Each open-statement increments this counter by one, each close-statement decrements the counter by one. As long as the counter is positive, access to the DATION is possible).

example:

```
:
DCL TABLE DATION INOUT FIXED(60, 20) DIRECT
          CONTROL (ALL) CREATED (DISK);

:
OPEN TABLE BY IDF ('TAB-1'), NEW;
:
     (transfer-statements to or from TABLE)
:
CLOSE TABLE;
```

2.3.2 Transfer Operations

2.3.2.0 General

In section 2.3.1 operations on data-stations have been described, that have to be executed before a transfer takes place.
This section is to detail the actual transfer operations.
In Basic PEARL transfer operations are always carried out between memory and a DATION.
If the DATION has one of the external attributes ALPHIC or BASIC the PEARL-objects are transformed to (out of) external representation. If the DATION has been declared with 'trf-item-type' the PEARL-objects are transported there and back in internal representation without any conversion.
Due to the different methods of transporting objects and the different directions there are several transfer operations.

transfer-operation::=

{ *get-statement*
put-statement
take-statement
send-statement
read-statement
write-statement }

The following table shows the relationship between DATION-attributes and transfer-operations.

usage / class	INOUT	
	IN	OUT
ALPHIC	GET	PUT
BASIC	TAKE	SEND
trf-item-type	READ	WRITE

table 7: transfer-operations

A transfer operation is in general a compound operation consisting of a sequence of "individual transfers".
The data-list is decomposed into transfer-items, which are then contained in the d-channel of a DATION; the c-list is decomposed, all the multipliers are evaluated. The resulting sequence of controls is contained in the c-channel.
An individual transfer may involve an element out of the c-list and/or one of the data-list. The lists are worked off in the following way:
an element of the c-list is considered: if it is a matching control the next element of the data-list is taken and matched (if possible; else a signal is raised); if it is a non-matching control it is executed.
If the data-list is exhausted before the c-list the latter is worked off until the next matching control is found, then the transfer is finished.
If the c-list is exhausted before the data-list the c-list is worked off from the beginning again.

If the c-list is omitted in a transfer-operation the data-list is worked off whereby formats are defaulted.

```
example:      DCL  A(5) FIXED;
              DCL  LPR DATION OUT ALPHIC (* ,60,130)
                       FORWARD CONTROL(ALL);
                 :
                 :
              PUT A(2:5) TO LPR BY X(3),
                         (2) ((2) F(7,2), SKIP);
```

Decomposing the lists results in the following:

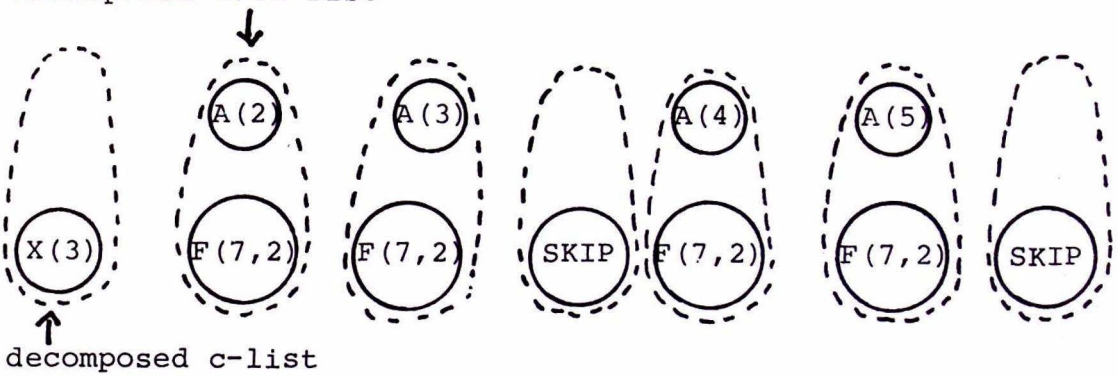


fig. 7: individual transfers
(enclosed in dotted lines)

In Basic PEARL on output the data-list may contain 'inv-or-var-objects':

inv-or-var-objects::=

{, * *symbol-or-constant-or-slice* ...}

symbol-or-constant-or-slice::=

{ *constant-denotation* | *symbol-or-slice* }

On input the data-list may not contain constants.

var-objects::=

{, • *symbol-or-slice* •••}

symbol-or-slice::=

{ *symbol*
{*identifier* (*simple-integer-constant-denotation*
[: *simple-integer-constant-denotation*])} }

'symbol-or-slice' is to denote simple variables, components of records, array-elements or one-dimensional slices.

symbol::=

identifier *expression-seven-pack*
[.*identifier*]
[.*BIT integer-in-brackets*]

'expression-seven-pack' may contain one, two or three 'expression-seven'. The 'identifier' following '.' is to denote components of records. The BIT-selector serves to select bits out of a bit-string (i.e. objects declared as BIT(n)).

examples for symbol:

```
A
B (L, 3)
S.M1
X(5).R
BSTR.BIT(7)
```

examples for slice:

```
A(2:5)
F(3:8)
```

Syntax and semantics of the transfer operations are detailed in the following sections.

2.3.2.1 PUT and GET

The 'put- (get-)statement' transfers PEARL-objects from (to) memory to (from) a DATION formatting them to (from) alphic symbols.

put-statement::=

PUT {{*inv-or-var-objects*
TO dation-odentifier [*index*]
[*BY c-list*]}/
{*TO dation-identifier* [*index*]
BY c-list }}

'dation-identifier' must denote a DATIOn declared as

..... { OUT / INOUT } ALPHIC

If 'inv-or-var-objects' is omitted, "BY c-list" must be supplied and may only contain 'pos-control' (cf. section 1.3.2.1).

example: a printer can be positioned at the beginning of the next page by

```
PUT TO PRINTER BY PAGE;
```

get-statement::=

GET {{ *var-objects*
FROM dation-identifier [*index*]
[*BY c-list*] } /
{ *FROM dation-identifier* [*index*]
BY c-list }}

'dation-identifier' must denote a DATION declared as

..... { IN / INOUT } ALPHIC

If 'var-objects' is omitted, "BY c-list" must be supplied and may only contain 'pos-control'.

In both statements the objects transferred must be compatible with the controls listed in 'c-list'.
The controls in 'c-list' must be compatible with the 'access' of the DATION (cf. section 1.3.1.1.1).
If "BY c-list" is omitted implementation-dependent conventions apply.

2.3.2.2 SEND and TAKE

The 'send- (take-)statement' transfers PEARL-objects from (to) memory to (from) a DATION converting them to basic symbols.

The objects transferred will in general be of the basic types (mainly BIT). The will be formatted to (or from) what a process-peripheral may recognize as a set of logic levels (e.g.: lamp on-off, valve open-closed etc.).

send-statement::=

SEND {{ *inv-or-var-objects*
TO *dation-identifier* [*index*]
[*BY* *c-list*]} /
{ *TO* *dation-identifier* [*index*]
BY *c-list*}}

'dation-identifier' must denote a DATION declared

..... { OUT / INOUT } BASIC

If 'inv-or-var-objects' is omitted, "BY c-list" must be supplied and may only contain 'pos-control' (cf. section 1.3.2.1).

take-statement::=

TAKE {{ *var-objects*
FROM dation-identifier [*index*]
[*BY c-list*]} /
{*FROM dation-identifier* [*index*]
BY c-list}}

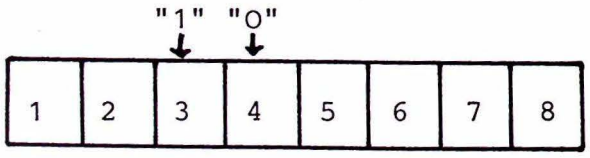
'dation-identifier' must denote a DATION declared as

..... { IN / INOUT } BASIC

If 'var-objects' is omitted, "BY c-list" must be supplied and may only contain 'pos-control'.

examples:

(1)
```
DCL LAMPROW DATION OUT BASIC (8) FORWARD CONTROL (ALL);
DCL A BIT INITIAL ('1'B);
  :
SEND A, 'O'B TO LAMPROW BY X(2), (2) B(1);
```



LAMPROW

(2)
```
DCL STATUS BIT;
  :
TAKE STATUS FROM SWITCH;
```

2.3.2.3 WRITE and READ

The 'write- (read-)statement' transports PEARL-objects in internal representation from (to) memory to (from) a DATION without changing them. It is a mere transport without formatting or converting.

write-statement::=

WRITE {{*inv-or-var-objects*
TO dation-identifier [*index*]
[*BY c-list*]}/
{*TO dation-identifier* [*index*]
BY c-list}}

'dation-identifier' must denote a DATION declared

..... { OUT / INOUT } trf-item-type

'c-list' may only be used for positioning the DATION. It must be supplied if 'inv-or-var-objects' is omitted.

read-statement::=

READ {{*var-objects*
FROM dation-identifier [*index*]
[*BY c-list*]}/
{*FROM dation-identifier* [*index*]
BY c-list}}

'dation-identifier' must denote a DATION declared

..... { IN / INOUT } trf-item-type

If 'var-objects' is omitted, "BY c-list" must be supplied and may only contain 'pos-control' (cf. section 1.3.2.1).

The type of the PEARL-objects transferred must be compatible with the type in 'trf-item-type'.
The 'pos-control' in 'c-list' must be compatible with the dimension and access attribute given in the declaration.

example:

```
DCL A DATION INOUT FLOAT (*) DIRECT CONTROL (ALL);
DCL F(10) FLOAT;
  :
WRITE F(3:7) TO A;
  :
READ FROM A BY X(-1);
READ F(7) FROM A;
  :
```

2.4 Realtime Operations

2.4.0 General

Operations acting on the PEARL-objects event, synchronizer, or task (which have been explained in 1.4) are called "realtime operations".

realtime-statement::=

$$\left\{\begin{array}{l}\textit{event-operation}\\ \textit{synchronizer-operation}\\ \textit{task-operation}\end{array}\right\}$$

Event-operations will be detailed in section 2.4.1 with the exception of interrupt-reactions, which are included in the section on task-operations (2.4.3).

In section 2.4.2 synchronization is explained, which in Basic PEARL is restricted to explicit operations on the object semaphore.

Section 2.4.3 deals with task-operations. Prior to the activity-handling operations two general features of task-operations are discussed: schedules and prioritites. This section also contains the explanations for the transitions in the state-diagram introduced in 1.4.3, and the boundery conditions that have to be satisfied.

2.4.1 Event Operations

2.4.1.0 General

Basic PEARL offers the following operations to control objects of the type event, introduced in 1.4.1:

event-operation::=

$$\left\{\begin{array}{l}\textit{interrupt-masking}\\ \textit{signal-stimulation}\\ \textit{signal-reaction}\end{array}\right\}$$

Interrupt-masking enables the programmer to allow or to inhibit the occurrence of an interrupt in his program by using special statements. These statements are detailed in section 2.4.1.1.
Similar explicit statements for masking signals are not provided, since there is another way of realizing that by specification of an empty signal-reaction (cf. section 2.4.1.3).

Mainly for testing and simulation of programs there is a statement to stimulate signals - the induce-statement. This is explained in section 2.4.2.2. (Note: There is no way to stimulate interrupts in Basic PEARL).

Due to the fact that there are two types of events, two different event-reactions can be specified.

Interrupt-reactions are explained in the section on task-operations, since scheduling of task-operation is the only possible way to react on the occurrence of an interrupt (cf. section 2.4.3.0.1).

Signal-reactions are formulated as ON-blocks.
They are explained in section 2.4.1.3.

2.4.1.1 Interrupt Masking

Interrupt masking is based on two statements:

interrupt-masking::=

$$\left\{\begin{array}{l}\textit{enable-statement}\\ \textit{disable-statement}\end{array}\right\}$$

After specification the state of an interrupt is implementation-dependent either "disabled" or "enabled". "Disabled" means the interrupt cannot be recognized at program-level but is prevented by the system. "Enabled" means the interrupt can be recognized in the program. Changing the state of an interrupt is achieved by the 'enable-statement' and 'disable-statement'.

enable-statement::=

ENABLE interrupt-identifier [*integer-in-brackets*]

So the execution of an enable-statement defines a point of time, from which on an interrupt is recognized in the program. From that very moment scheduled interrupt reactions can take place.

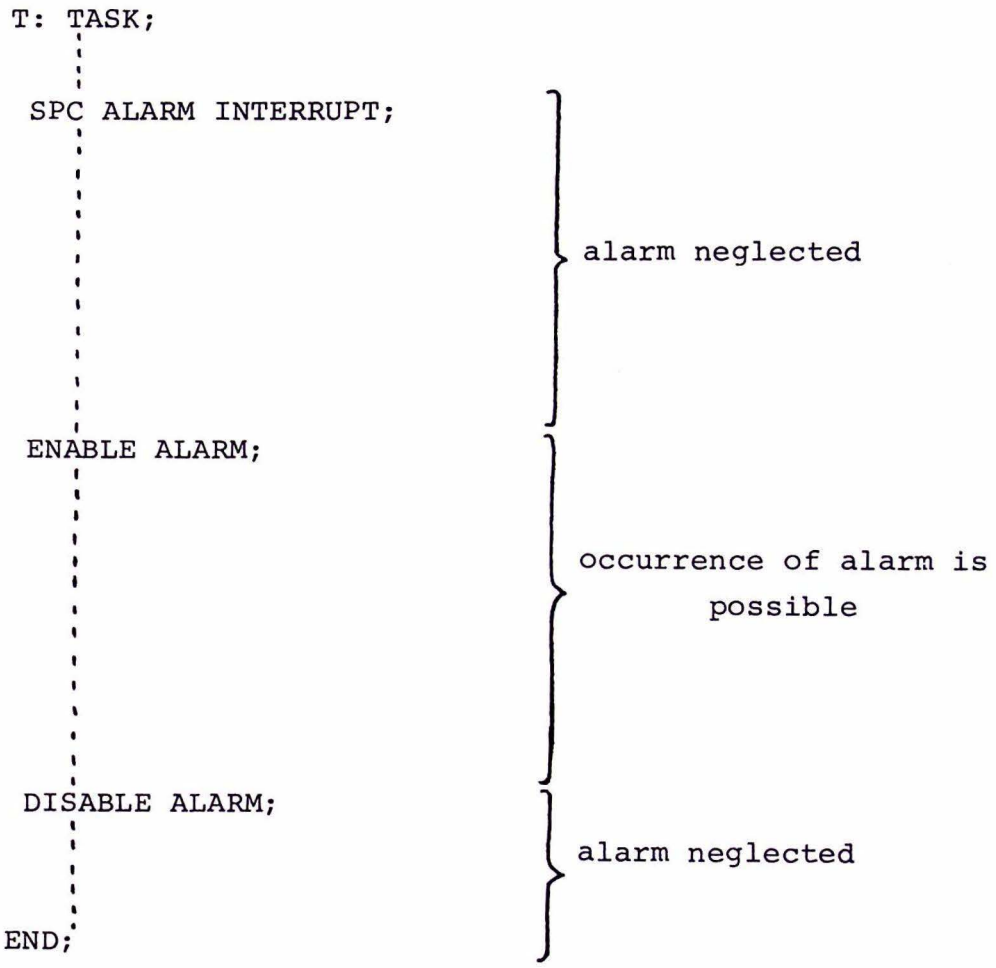
The ENABLE-counterpart is the disable-statement.

disable-statement::=

DISABLE interrupt-identifier [*integer-in-brackets*]

It masks the interrupt denoted by 'interrupt-identifier', i.e. an occurrence of this interrupt is neglected on program-level.

The following example shall illustrate interrupt-masking
for an implementation defaulting the initial-state to "disabled"



2.4.1.2 Signal Stimulation

As already mentioned in section 1.4.1.2 signals may be stimulated either explicitly or implicitly.
Implicit stimulation can be seen as the 'normal' way for the occurrence of a signal: the execution of a PEARL-statement results in some special conditions (e.g. the overflow-indicator has been set, EOF has been encountered..), and therefore a signal is raised.
This section deals with the explicit stimulation of signals which is achieved by the induce-statement.

signal-stimulation::=

induce-statement

On execution of an induce-statement an occurrence of the signal denoted by 'signal-identifier' is created.

induce-statement::=

INDUCE signal-identifier [*integer-in-brackets*]

This creation results in a procedure-call of the corresponding ON-block or in a system reaction, if there is no ON-block provided for this signal.
The induce-statement is mainly thought as a tool for the programmer to test his ON-blocks without having to wait for the occurrence of a signal.

2.4.1.3 Signal Reactions

Reactions on signals can be prepared by the programmer by writing ON-blocks.

signal-reaction::=

ON {,• *signal-identifier* [*integer-in-brackets*]•••} :
[*unlabelled-statement*]

'signal-identifier' (possibly with an index) must identify a signal specified in the module containing the ON-block. 'unlabelled statement' denotes the reaction that is to be executed as soon as the signal is raised. Any unlabelled Basic PEARL statement is allowed except 'signal-reaction'. (No recursion!)
If a reaction is to be valid for more than one signal one may list all the signal-identifiers separated by commas.
(e.g. ON ALARM,ITR(3),OFL:
CALL TEST;)

The occurrence of any of the signals out of that list leads to the execution of the provided reaction.

The execution of 'unlabelled statement' is performed similar to a procedure call, i.e. the program stops as soon as a signal is raised, performs the statements of the reaction, and then continues at the point where it had been stopped. (except the continuation has explicitly been formulated in a different way within the reaction, e.g. by a goto-statement or a terminate-statement).

By omitting 'unlabelled statement' the programmer can specify an empty reaction. When the signal is raised nothing is done, the signal is ignored.
This is a way of "masking" a signal (cf. 2.4.1.1 explicit interrupt-masking by the 'disable-statement').

2.4.2 Synchronization

2.4.2.0 General

In a PEARL program different tasks can usually execute their statements independently from each other. But sometimes a time-ordering for some statements in different tasks is required. This is called synchronization.
In section 1.4.2 special objects for synchronization have been introduced, variables of the type 'semaphore'. Execution of a task can be made dependent on the value of a semaphore.
To inquire and modify this value two operations are provided.

synchronizer-operation::=

$$\left\{\begin{matrix}\textit{request-statement}\\ \textit{release-statement}\end{matrix}\right\}$$

The request-statement decrements the value; it reserves the semaphore for the requesting task, if possible.
The release statement increments the value, it sets the semaphore free again, allowing new requests.
The semantics of these two statements will be detailed in the following sections.

2.4.2.1 Request

A semaphore can be requested by the following statement:

request-statement::=

REQUEST semaphore-identifier

'semaphore-identifier' denotes a variable of the type 'SEMA'. The effect of the statement depends on the actual value of the semaphore:

- if it is greater than zero, it is decremented by one. The requesting task is allowed to continue its execution.

- if it is zero, the task issuing the request-statement is suspended until the value of the semaphore is incremented by a release-statement from another task. Then a further request can be granted.

The second case (the unsuccessful request) can also be shown in terms of the diagram in section 1.4.3.
If the value of the semaphore requested is zero, the task is taken from the state 'runnable' to 'suspended', waiting for a release-statement.

2.4.2.2 Release

The release of a semaphore can be achieved by the following statement:

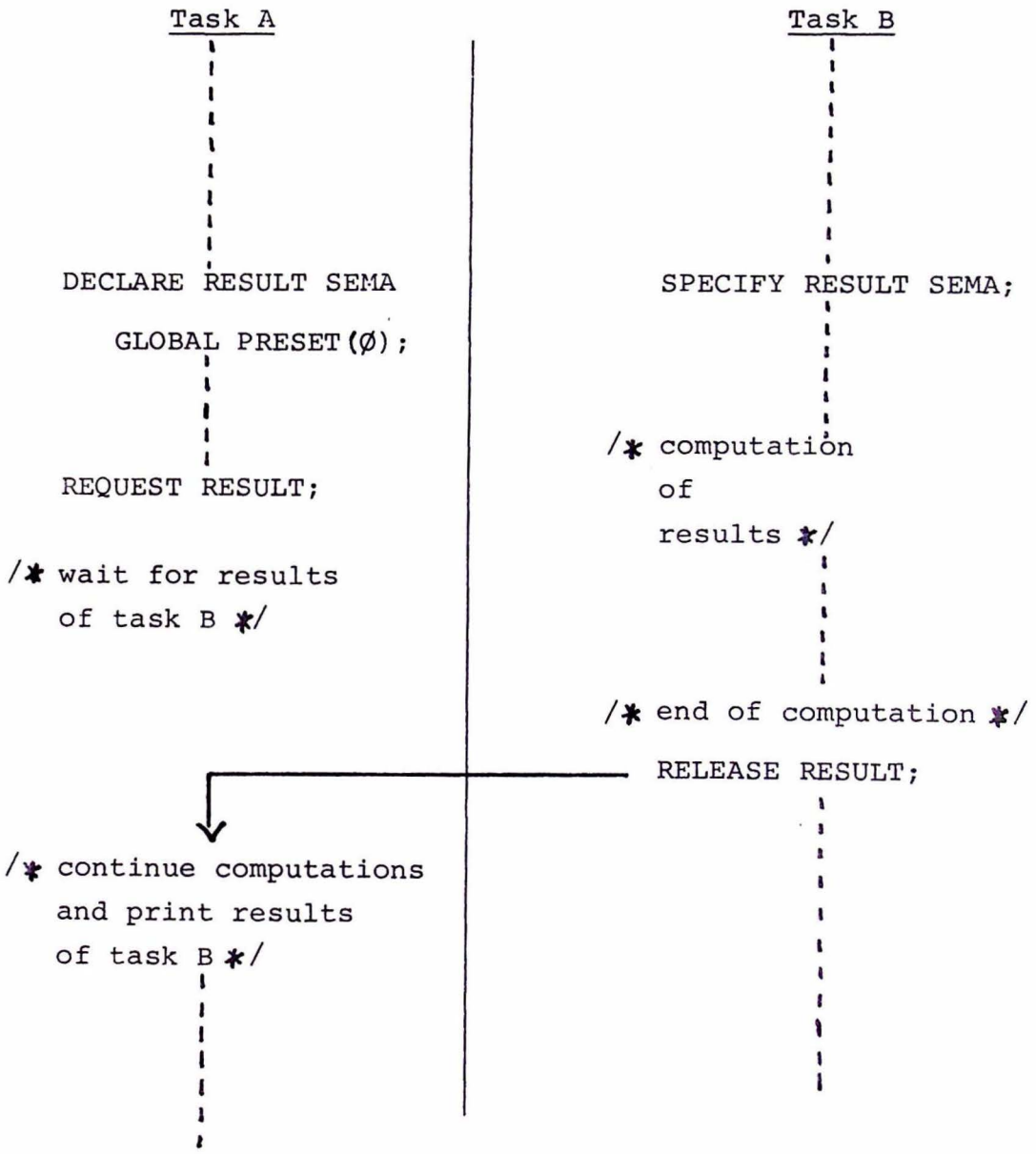
release-statement::=

RELEASE semaphore-identifier

The release-statement increments the value of the semaphore denoted in the statement by one.
This operation is always successfull.
Additionally it is checked if there is a task waiting for a release of this semaphore, i.e. whether a task is in the state 'suspended' because of an unsuccessfull request-statement for this semaphore. Of all the waiting tasks the one with the highest priority can now issue its request again.

The following example demonstrates the explicit synchronization between two tasks with the help of a semaphore.

Task A has to print the results computed in task B.



2.4.3 Task-Operations

2.4.3.0 General

In PEARL there are six kinds of operations of tasks:

task-operations::=

{ *activate-statement*
| *terminate-statement*
| *suspend-statement*
| *continue-statement*
| *resume-statement*
| *prevent-statement* }

All these statements describe special operations on tasks, or - more precisely - operations on one or more activities of a task.
The statements will be explained with the help of the virtual processor concept (introduced in section 1.4.3). The diagram containing the states and transitions of tasks (s. 1.4.3) will give the reader a rough impression of the effect of the statements. It will also be helpful with the explanation of the semantics.

Prior to the explanation of the statements two general features of task-operations shall be discussed: schedules and prioritites.

2.4.3.0.1 Schedules

Schedules are additional conditions that may be specified with some task-operations.
These conditions have to be fulfilled before a state-transition takes place.
A simple example shall illustrate this:

ACTIVATE T;	This statement means that the activation of task T shall take place immediately.
AT 9:Ø:Ø ACTIVATE T;	With this statement task T shall be activated too, but there is an additional condition, which states that it shall happen at 9 o'clock.

In terms of the diagram in section 1.4.3 we can say:
a scheduled task-operation takes the task concerned by that statement to the state 'scheduled' or 'suspended and scheduled'. It waits there until the condition of the schedule is satisfied, then the task is transfered to the state 'runnable' by system-action.

In Basic PEARL two kinds of schedules are provided:

- time-dependent schedules
- interrupt-controlled schedules

Time-dependent schedules permit the specification of a point of time, at which the task-operation is to be executed or an interval, which must pass before the operation takes place.

The activate-statement may also be scheduled repetitively, i.e. you may specify a time-interval and a final condition; then the statement is executed every time the interval has passed and the final condition is not satisfied.

Interrupt-controlled schedules permit an execution of a task-operation dependent on the occurrence of an interrupt (cf. section 2.4.1.0).

schedule::=

{ *schedule-1* | *schedule-2* }

'schedule-1' may be specified with the activate- and the continue-statement, and must be specified with the resume-statement.

schedule-1::=

{ *AT clock-expression-seven* | *AFTER duration-expression-seven* | *WHEN interrupt-identifier* [*integer-in-brackets*] }

AT clock-expression-seven: this specifies a point of time, at which the task-operation is to be worked off.

e.g.: AT 9:Ø:Ø ACTIVATE SORT;

AFTER duration-expression-seven: specifies a time-interval, which has to pass before the operation is executed.

e.g.: AFTER 30 MIN RESUME;
(a task stops its execution for a period of 30 minutes, then it is taken back to the state 'runnable')

WHEN interrupt-identifier [integer-in-brackets] : with the help of this schedule interrupt-reactions may be specified. As soon as the interrupt occurs the task-operation is executed.

e.g.: WHEN ITR CONTINUE TAØ1;
'interrupt-identifier' may be the name of a single interrupt or an interrupt-array. If it is the name of an array 'integer-in-brackets' allows indexing.

Note: you may specify one interrupt or a whole interrupt-array, but no slices (ITR(6:8) is illegal).

When 'interrupt-identifier' denotes an array, the occurrence of one interrupt out of that array is enough to satisfy the condition.

e.g.: SPC ALARM(3) INTERRUPT;
:
WHEN ALARM ACTIVATE REACT;

This statement means: when ALARM(1) or ALARM(2) or ALARM(3) occurs the task REACT shall be activated.

'schedule-2' may only be used with the activate-statement. It permits scheduling a cyclic activation of a task.

schedule-2::=

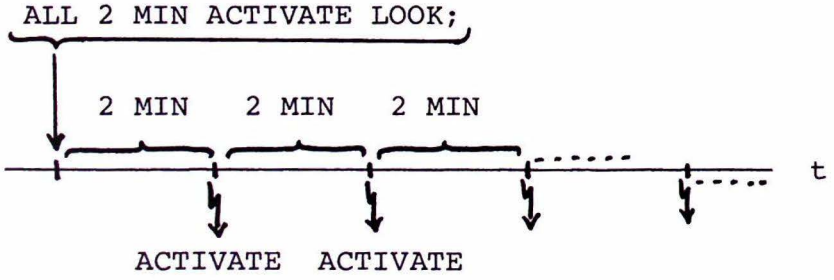
{*ALL* / *EVERY*} *duration-expression-seven*

[{*UNTIL* *clock-expression-seven* / *DURING* *duration-expression-seven*}]

ALL (or EVERY) duration-expression-seven specifies the length of a time-interval. This interval starts at the moment the schedule is worked off. At any time the interval has passed the task-operation (activation) is executed.

(Remark: In Basic PEARL there is no difference between using ALL or EVERY. It is implementation dependend, if ALL or EVERY or both are supplied)

e.g.: ALL 2 MIN ACTIVATE LOOK;

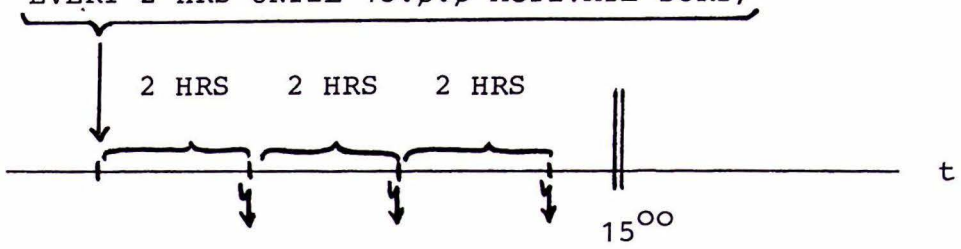


(↯ denotes the activation)

The final condition can be denoted in two ways:

- specifying a point of time (UNTIL clock-expression-seven) up to which the cyclic activation is executed.

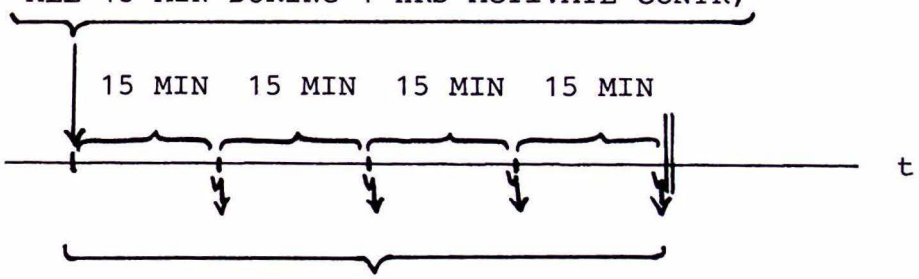
e.g.: EVERY 2 HRS UNTIL 15:Ø:Ø ACTIVATE SORT;



(final point of time)

- specifying an interval (DURING duration-expression-seven) during which the cyclic activation takes place.

e.g.: ALL 15 MIN DURING 1 HRS ACTIVATE CONTR;



1 HRS: specified interval starting at the moment the schedule is worked off.

Note: Schedules for the same task and task-operation replace each other in the sequence the schedules are worked off. In other words task-operation can be scheduled only once at a time for one task. If there is a second scheduling of one operation for one task before the first scheduled operation is executed the second schedule replaces the first one.
One case requires special care: since a resume-statement acts like a scheduled continue-statement (connected with a suspend-statement) the schedule of a resume-statement replace the schedule of a continue-statement of the same task and vice versa.

example 1:
```
AFTER 5 MIN CONTINUE MYTASK;
AT 1Ø:Ø:Ø  ACTIVATE MYTASK;
```

These two schedules may be valid at one time, because <u>different</u> operations for one task are scheduled.

example 2:
```
WHEN ITR ACTIVATE TASK1;
ALL 5 MIN UNTIL 9:3Ø:ØØ ACTIVATE TASK2;
```

These two schedules may also be valid at the same time, because one operation has been scheduled for <u>different</u> tasks.

example 3: AT 1Ø:Ø:Ø ACTIVATE MYTASK;
WHEN ITR ACTIVATE MYTASK;

These two schedules may not be valid at one time. Let us assume the first statement has been worked off, MYTASK is scheduled to be executed at 10 o'clock. If the second statement is worked off before 10 o'clock (i.e. before the activation is executed) the second schedule 'WHEN ITR' replaces the first one. MYTASK will only be activated if the interrupt 'ITR' occurs and not (no longer) at 10 o'clock.

example 4: if at 9^{30} the two statements

AFTER 30 MIN RESUME; and
AT 9:45:ØØ CONTINUE TØ3;

are executed for task TØ3 (in this sequence) only the second schedule (AT 9:45:ØØ) will remain valid, the first one is replaced.

2.4.3.0.2 Priorities

In section 1.4.3 a virtual processor concept has been introduced. Each task has its own processor; one activity of each task can be executed on this task-specific processor, the others have to wait until the first one is finished or terminated.
All the virtual processors can do their work without regard for the others (except explicit synchronization, cf. 2.4.2).

But in a real hardware configuration you will not have as many processors as tasks, normally you will find just one processor, the time of which is divided up among all runnable tasks by the operating system. For this distribution of time the operating system uses certain criteria. One of these is the priority of a task.

The priority is a measure for the importance of a task, i.e. it influences the time of execution.
(The time of execution is the period between the activation of a task and the explicit or implicit termination).
It is quantified with the aid of an integer number, lower values denoting greater importance.

priority::=

{ *PRIORITY* | *PRIO* } *simple-integer-constant-denotation*

In Basic PEARL priority can only be denoted with a task-declaration (cf. section 1.4.3).

If the priority is omitted it is defaulted by a system-dependent value.

example:

```
T:TASK PRIO 7;
   :
   :
END;

R:TASK;
   :
   :
END;

ACTIVATE T;
     /*  T is activated with priority 7  */

ACTIVATE R;
     /*  R is activated with a defaulted
         priority  */
```

2.4.3.1 Activate

The activate-statement takes a task to the state 'runnable' or 'scheduled'

activate-statement::=

[schedule]
ACTIVATE identifier

'identifier' must be the name of a task.
An activate-statement without schedule immediately takes the task to the state 'runnable' (i.e. a new activity of this task is generated).
If this task has already been activated (once or several times) this new activation (and the following) is (are) buffered. In other words, the activity queues up for the virtual processor. How often an activation can be buffered is implementation dependent.
If no activity of this task exists (i.e. the task is neither in the state 'runnable' nor 'suspended' nor 'suspended and scheduled') then this new activity immediately gets hold of the virtual processor.

So we can say, that by several activate-statements for one task several activities are created; the activity that has been created first has occupied the virtual processor of this task, all the others have to wait until the first one is finished, then they are worked off sequentially.

A scheduled activation takes a task to the state 'scheduled'. According to the rules for schedules (cf. section 2.4.3.0.1) an already existing schedule of an activation for this task is destroyed and replaced by the new one.
A scheduled activate-statement can be seen as planning the creation of a future activity of the task in question.

examples for activations:

```
ACTIVATE ALARM;
```

The task 'alarm' is immediately activated

```
AT 7:3Ø:ØØ ACTIVATE GETDAT;
```

The task 'getdat' will start to accept data at 7^{30}

```
ALL 2 HRS DURING 10 HRS ACTIVATE REPORT;
```

The task 'report' is to print statistical results every two hours from now on

An activate-statement always creates a new activity of a task. Now the question arises how these activities can expire. Therefore PEARL provides three ways:

- an activity can come to a "normal" end, if all statements of the task have been executed and the end-statement is reached. In the diagram one can see this as a dotted line between the states 'runnable' and 'dormant' marked with "ready".
- the second way for an activity to expire is "more violent". It can be killed (or kill itself) by a terminate-statement. This is described in the following section.
- the third way is provided by the 'prevent-statement' which is detailed in section 2.4.3.6.

2.4.3.2 Terminate

As already mentioned in the previous section the terminate-statement is one way for an activity to expire. The terminate-statement explicitly kills one activity. It is independent of whether the activity is running or waiting.
In terms of the diagram one can say that the task is taken either from the state 'runnable' or 'suspended' or 'suspended and scheduled' back to the state 'dormant'.

terminate-statement::=

TERMINATE identifier

A terminate-statement with an identifier kills an activity of the task denoted by this identifier.
With the help of a terminate-statement without identifier a task can finish its own execution.
A terminate-statement only affects the activity of a task that has occupied the processor (i.e. the "running" one). All the other buffered activities and scheduled (future) activities are not concerned.

examples: TERMINATE VALVE;

The execution of the task 'valve' is immediately finished.

TERMINATE;

The running task finishes its own execution and releases the processor.

2.4.3.3 Suspend

The suspend-statement produces the transition from the state 'running' to the state 'suspended'.

suspend-statement::=

SUSPEND

This statement can only be executed for the own task. The effect is a temporary stop of the execution, but in opposition to the terminate-statement the execution of this task can be continued at the point where it stopped. (This has to be done with the aid of a continue-statement from another task, cf. next section). The task is set waiting.

An activity of a task can only execute a suspend-statement if it has occupied the virtual processor. Although it stops its execution it keeps the virtual processor.
Since the processor is not released no other activity of this task can start running (even if there are some in the state 'runnable').

example:

```
COMP:TASK;
   •
   •
   •
  IF BUFCNT = 0 THEN SUSPEND FIN;
```

The task 'comp' stops its own execution after recognizing that a buffer is empty (the buffercounter is zero).

2.4.3.4 Continue

The continue-statement is the inverse operation to SUSPEND. It takes a task from the state 'suspended' either back to 'runnable' or to the state 'scheduled'.

continue-statement::=

[*schedule-1*]
CONTINUE identifier

A continue-statement without 'schedule-1' takes the task denoted by 'identifier' immediately back to the state 'runnable'. (The task can continue its execution without waiting, since it did not release the virtual processor when it was suspended).

By the additional specification of 'schedule-1' the task is taken from the state 'suspended' to 'suspended and scheduled'. Also during this transition the virtual processor is not released.

The following conditions are admitted in the schedule:

- a point of time
- a time interval
- the occurrence of an interrupt

(no cyclic scheduling!)

After the condition has been satisfied the task is taken to the state 'runnable' by system action.

Still having occupied the virtual processor the task can continue its execution.

example:

```
PROD:TASK;
   :
   :
   BUFCNT:= BUFCNT+3;
   CONTINUE COMP;
   :
   :
   :
```

The task 'prod' causes the continuation of the task 'comp' after filling a buffer (and increasing the buffercounter).

2.4.3.5 Resume

The resume-statement allows to delay a task in a definite way. The task can stop its own execution but in the same moment it plans its continuation.
So the resume statement can be seen as an indivisible combination of a scheduled continuation and a suspend-statement.
Syntactically it is denoted in the following way:

resume-statement::=

schedule-1 RESUME

'schedule-1' must be specified. The same restrictions as for the continue-statement have to be observed (no cyclic scheduling allowed).
A task executing a resume-statement is taken from the state 'runnable' to the state 'suspended and scheduled'. It does not release its virtual processor. As soon as the scheduled condition is satisfied (the delay-time has passed) the task is taken back to the state 'runnable' by system-action and continues its execution.

example: AFTER 2 MIN RESUME;

(A task delays its execution for 2 minutes)

WHEN ITR RESUME;

(A task stops its execution until the interrupt ITR occurs)

2.4.3.6 Prevent

The prevent-statement is to prevent the execution of a task, i.e. it invalidates the schedules for all operations of one task and it empties the queue of activities built up by several unscheduled activate-statements (if this queue exists).
Only an actually running activity (an activity having occupied the processor) is not concerned by a prevent-statement. (This one may be killed by a terminate-statement, as detailed in section 2.4.3.2).

prevent-statement::=

PREVENT identifier

'identifier' denotes the name of the task that is to be prevented; if it is omitted the task executing the prevent-statement is concerned.

In the previous section the effect of scheduled task-operations has been explained. We will now consider the effect of a prevent-statement with respect to the different scheduled task-operations.

- If the prevent-statement concerns a task for which an activation has been scheduled, this task is taken back from the state 'scheduled' to 'dormant'. The planned creation will not take place.
- If an activity is in the state 'suspended and scheduled' (i.e. a continue has been scheduled or a resume-statement has been executed) it is taken to the state 'suspended'. The schedule is invalidated, the activity is set waiting.

example: At 9 o'clock the following scheduled activation for the task EXAM has been executed:

```
ALL 2 HRS UNTIL 17:Ø:Ø ACTIVATE EXAM;
```

The task has been executed at 11 o'clock, but at 12 o'clock the statement

```
PREVENT EXAM;
```

is executed. This means, that the task EXAM will not be executed at 1, 3 and 5 pm as originally planned, because the scheduled activations have been invalidated by a prevent-statement at 12 o'clock.

A P P E N D I X

S SYNTAX LIST

C CROSS REFERENCE LIST

8
17
14
13
13
17
14
17
16
10
5
156

Gauthier Benoit PETARC

S Syntax List

The following pages comprise the complete syntax of Basic PEARL.
Contrary to the notation in the Language Description terminals are denoted enclosed in (double) quotes (since no small letters were available).

examples: "TASK"
";"
"F"

Not every production-rule contained in the Language Description will be found in this syntax list, since the former is more detailed to facilitate reading. But ~~regarding~~ with regards to the content they are equivalent.

156 Syntax rules

```
BINARY-DIGIT ::=
      {"0"}
      {"1"}


OCTAL-DIGIT  ::=
       "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"


DIGIT ::=
      { OCTAL-DIGIT }
      { "8"         }
      { "9"         }


HEXADECIMAL-DIGIT ::=
     DIGIT / "A" / "B" / "C" /
     "D" / "E" / "F"


LETTER ::=
    "A" / "B" / "C" / "D" / "E" / "F" /
    "G" / "H" / "I" / "J" / "K" / "L" /
    "M" / "N" / "O" / "P" / "Q" / "R" /
    "S" / "T" / "U" / "V" / "W" / "X" /
    "Y" / "Z"


IDENTIFIER  ::=
                  {LETTER}
     LETTER  [    {      }  ...   ]
                  {DIGIT }


ONE-IDENTIFIER-OR-LIST ::=
      {IDENTIFIER                          }
      {"("{","' IDENTIFIER ...} ")"        }


LABEL-IDENTIFIER ::=
     IDENTIFIER
```

```
INTERRUPT-IDENTIFIER  ::=
     IDENTIFIER


SIGNAL-IDENTIFIER ::=
     IDENTIFIER


SEMAPHORE-IDENTIFIER  ::=
     IDENTIFIER


TASK-IDENTIFIER ::=
     IDENTIFIER


FORMAT-IDENTIFIER ::=
     IDENTIFIER


PROCEDURE-IDENTIFIER  ::=
     IDENTIFIER


DATION-IDENTIFIER ::=
     IDENTIFIER


CONSTANT-DENOTATION ::=
      { INTEGER-CONSTANT-DENOTATION
      { REAL-CONSTANT-DENOTATION
      { CHARACTER-STRING-CONSTANT-DENOTATION
      { BIT-STRING-CONSTANT-DENOTATION
      { CLOCK-CONSTANT-DENOTATION
      { DURATION-CONSTANT-DENOTATION


INTEGER-CONSTANT-DENOTATION ::=
       SIMPLE-INTEGER-CONSTANT-DENOTATION
                                  [ PRECISION ]
```

```
PRECISION ::=
     INTEGER-IN-BRACKETS


INTEGER-IN-BRACKETS ::=
     "(" SIMPLE-INTEGER-CONSTANT-DENOTATION ")"


SIMPLE-INTEGER-CONSTANT-DENOTATION ::=
     { DIGIT ...
     { BINARY-DIGIT ... "B" }


REAL-CONSTANT-DENOTATION ::=
     SIMPLE-REAL-CONSTANT-DENOTATION  [ PRECISION ]


SIMPLE-REAL-CONSTANT-DENOTATION  ::=
   { { { [ DIGIT ... ] "." DIGIT ... }                  }
   { { {                                } [ EXPONENT-PART ] }
   { { { DIGIT ...   "."                }                  }
   {
   { DIGIT ...     EXPONENT-PART
   }


EXPONENT-PART  ::=
    "E" { ["+"]/"-" } [ DIGIT ] DIGIT


CHARACTER-STRING-CONSTANT-DENOTATION ::=
     "'" [ STRING-CHARACTER ... ] "'"


STRING-CHARACTER ::=
     LETTER  /  DIGIT  /
     " " / "+" / "-" / "*" / "/" /
     "(" / ")" / ":" / "." / "," /
     ";" / "=" / "<" / ">" / "'" /
     "[" / "]"
```

```
156  BIT-STRING-CONSTANT-DENOTATION  ::=
157      {"'"   BINARY-DIGIT "'"   "'"  { "B" / "B1" } }
158      {"'"   OCTAL-DIGIT "'"    "'"  "B3"            }
159      {"'"   HEXADECIMAL-DIGIT "'"   "'"  "B4"       }
160
161
162
163  CLOCK-CONSTANT-DENOTATION ::=
164      SIMPLE-INTEGER-CONSTANT-DENOTATION  ":"
165      SIMPLE-INTEGER-CONSTANT-DENOTATION  ":"
166       {SIMPLE-INTEGER-CONSTANT-DENOTATION }
167       {SIMPLE-REAL-CONSTANT-DENOTATION    }
168
169
170
171  DURATION-CONSTANT-DENOTATION ::=
172       {HOURS  [ MINUTES ] [ SECONDS ] }
173       {MINUTES [ SECONDS ]            }
174       {SECONDS                        }
175
176
177
178  HOURS ::=
179      SIMPLE-INTEGER-CONSTANT-DENOTATION  "HRS"
180
181
182
183  MINUTES ::=
184      SIMPLE-INTEGER-CONSTANT-DENOTATION "MIN"
185
186
187
188  SECONDS ::=
189       {SIMPLE-INTEGER-CONSTANT-DENOTATION }
190                                            } "SEC"
191       {SIMPLE-REAL-CONSTANT-DENOTATION    }
192
193
194
195  BASIC-PEARL-PROGRAM ::=
196          MODULE "'"
197
198
199
200  MODULE ::=
201        "MODULE" ":"
202           {SYSTEM-DIVISION  [ PROBLEM-DIVISION ] }
203           {PROBLEM-DIVISION                      }
204        "MODEND" ":"
205
206
207
```

```
208  PROBLEM-DIVISION ::=
209       "PROBLEM" ";"
210       DECLARATIONS-AND-SPECIFICATIONS
211
212
213
214  DECLARATIONS-AND-SPECIFICATIONS  ::=
215            { LENGTH-DECLARATION ";"    }***
216       [    {                          }   ]
217            { PRECISION-DECLARATION ";" }
218       [ GLOBAL-SPECIFICATION ";" ]***
219       [ GLOBAL-DECLARATION ";" ]***
220       [ R-FORMAT-DECLARATION ";" ]***
221       [ PROCEDURE-DECLARATION ";" ]***
222       [ TASK-DECLARATION ";" ]***
223
224
225
226  PRECISION-DECLARATION  ::=
227       "LENGTH"
228          {"FIXED" }  PRECISION
229          {"FLOAT" }
230
231
232
233  LENGTH-DECLARATION ::=
234       "LENGTH"
235          {"BIT"       }
236          {"CHAR"      }   LENGTH
237          {"CHARACTER" }
238
239
240
241  GLOBAL-SPECIFICATION  ::=
242      {"SPECIFY"}
243      {"SPC"    }
244       { "," * ONE-IDENTIFIER-OR-LIST GLOBAL-SPEC-ATTRIBUTES
245         [ "GLOBAL" ] ** }
246
247
248
249  GLOBAL-SPEC-ATTRIBUTES ::=
250      { [ "(" [","] [","] ")" ]  LOCAL-MODE                    }
251      {"SEMA"                                                  }
252      { [ "(" ")" ]  DATION-SPEC-ATTR                          }
253      {"TASK"                                                  }
254      { IRPT-OR-SIGNAL-MODE                                    }
255      { PROCEDURE-MODE  ["RESIDENT"] ["REENT"]                 }
256
257
258
259
```

```
LOCAL-MODE  ::=
     ["INV"] { SIMPLE-MODE    }
             { STRUCTURE-MODE }


LENGTH ::=
     INTEGER-IN-BRACKETS


SIMPLE-MODE  ::=
    { "CLOCK"                                          }
    { "DUR" / "DURATION"                               }
    {{"BIT"                          } [ LENGTH ]      }
    {{"CHAR" / "CHARACTER"           }                 }
    {{"FIXED"                        } [ PRECISION ]   }
    {{"FLOAT"                        }                 }


STRUCTURE-MODE ::=
     "STRUCT" {"[" / "(/"}
              {","  [ ONE-IDENTIFIER-OR-LIST ]
                    SIMPLE-MODE ... }
              {"]" / "/)"}


DATION-SPEC-ATTR ::=
     "DATION" D-CHANNEL-SPEC-ATTR
              [ "CONTROL" "(" "ALL" ")" ]


D-CHANNEL-SPEC-ATTR ::=
        {"IN"    }
        {"OUT"   }
        {"INOUT" }
        {"ALPHIC"      }
        {"BASIC"       }
        {TRF-ITEM-TYPE }
          [  DIM-SPEC ["TFU" ["MAX"] ]
            {"DIRECT"  }
            {"FORWARD" }
            {"FORBACK" }
            {["NOCYCL"]}
            { "CYCLIC" }
            {["STREAM"]}
            {"NOSTREAM"}                    ]
```

```
312  DIM-SPEC ::=
313       "(" [","] [","] ")"
314
315
316
317  IRPT-OR-SIGNAL-MODE ::=
318       [ "(" ")" ]
319          { "INTERRUPT" }
320          { "IRPT"      }
321          { "SIGNAL"    }
322
323
324
325  PROCEDURE-MODE ::=
326       "ENTRY"
327       [ "(" { "," * PARAMETER-MODE ***} ")" ]
328       [ RESULT-ATTRIBUTE ]
329
330
331
332  PARAMETER-MODE ::=
333      {[ "(" [","] [","] ")" ]  LOCAL-MODE  [ "IDENT" ] }
334      {[ "(" ")" ] DATION-SPEC-ATTR  "IDENT"            }
335
336
337
338  RESULT-ATTRIBUTE ::=
339       "RETURNS" "(" SIMPLE-MODE ")"
340
341
342
343  GLOBAL-DECLARATION ::=
344       { "DECLARE" }
345       { "DCL"     }
346       { "," * ONE-IDENTIFIER-OR-LIST
347            { [ BOUND-LIST ]
348              LOCAL-MODE ["RESIDENT"] ["GLOBAL"] INITIAL
349              "SEMA" ["RESIDENT"] ["GLOBAL"] PRESETTING
350               DATION-ATTR ["RESIDENT"] ["GLOBAL"]
351                   DATAWAY-CONSTRUCTION } *** }
352
353
354
355  BOUND-LIST  ::=
356        "(" SIMPLE-INTEGER-CONSTANT-DENOTATION
357        [ "," SIMPLE-INTEGER-CONSTANT-DENOTATION
358        [ "," SIMPLE-INTEGER-CONSTANT-DENOTATION ] ] ")"
359
360
361
362
363
```

13

```
INITIAL ::=
     [ "INIT"
       "(" { "," } [ "+" ] / "-" CONSTANT-DENOTATION "" } ")" ]


PRESETTING  ::=
     [ "PRESET"
       "(" { "," } SIMPLE-INTEGER-CONSTANT-DENOTATION "" } ")" ]


DATAWAY-CONSTRUCTION  ::=
     "CREATED"
     "(" DATION-IDENTIFIER [ INTEGER-IN-BRACKETS ] ")"


DATION-ATTR  ::=
     "DATION" D-CHANNEL-ATTR
              [ "CONTROL" "(" "ALL" ")" ]


D-CHANNEL-ATTR ::=
         { "IN"    }
         { "OUT"   }
         { "INOUT" }
         { "ALPHIC"       }
         { "BASIC"        }
         { TRF-ITEM-TYPE  }
           [  DIM  ["TFU" ["MAX"] ]
              { "DIRECT"  }
              { "FORWARD" }
              { "FORBACK" }
              { ["NOCYCL"] }
              {  "CYCLIC"  }
              { ["STREAM"] }
              { "NOSTREAM" }          ]


DIM  ::=
     "("  "*" / SIMPLE-INTEGER-CONSTANT-DENOTATION
          [ "," SIMPLE-INTEGER-CONSTANT-DENOTATION
          [ "," SIMPLE-INTEGER-CONSTANT-DENOTATION ] ] ")"
```

```
416  TRF-ITEM-TYPE  ::=
417        { SIMPLE-MODE    }
418        { STRUCTURE-MODE }
419
420
421
422  R-FORMAT-DECLARATION  ::=
423      FORMAT-IDENTIFIER ":"
424      "FORMAT" "(" STANDARD-C-LIST ")"
425
426
427
428  PROCEDURE-DECLARATION  ::=
429       PROCEDURE-IDENTIFIER ":"
430       {"PROCEDURE" }
431       {"PROC"      }
432        [ "(" { "," ' ONE-IDENTIFIER-OR-LIST
433                      PARAMETER-MODE ''' }  ")" ]
434        [ RESULT-ATTRIBUTE ]
435        ["RESIDENT"] ["REENT"] ["GLOBAL"] ";"
436        BLOCK-TAIL
437
438
439
440  BLOCK-TAIL  ::=
441       [ LOCAL-IDENTIFIER-DECLARATION ";" ]'''
442       [ STATEMENT ]'''
443       "END"
444
445
446
447  LOCAL-IDENTIFIER-DECLARATION ::=
448       {"DECLARE" }
449       {"DCL"     }
450       {"," ' ONE-IDENTIFIER-OR-LIST
451            [ BOUND-LIST ]  LOCAL-MODE
452            INITIAL '''}
453
454
455
456  TASK-DECLARATION ::=
457       TASK-IDENTIFIER ":"
458       "TASK" [ PRIORITY ]
459              ["RESIDENT"] ["GLOBAL"] ";"
460              BLOCK-TAIL
461
462
463
464  STATEMENT ::=
465       [ LABEL-IDENTIFIER ":" ]'''
466       [ UNLABELLED-STATEMENT ] ";"
467
```

B

```
UNLABELLED-STATEMENT  ::=
      { ASSIGNMENT
        BEGIN-BLOCK
        CALL-STATEMENT
        RETURN-STATEMENT
        TRF-OF-CTRL-STATEMENT
        REALTIME-STATEMENT
        COMMUNICATION }


ASSIGNMENT  ::=
     SYMBOL { ":=" } EXPRESSION-SEVEN
            { "="  }


BEGIN-BLOCK  ::=
    "BEGIN" [;] BLOCK-TAIL


EXPRESSION-SEVEN-PACK  ::=
    "(" { "," * EXPRESSION-SEVEN ... } ")"


EXPRESSION-SEVEN ::=
     [ EXPRESSION-SEVEN PREC-7-OPERATOR ]
                 EXPRESSION-SIX


EXPRESSION-SIX ::=
     [ EXPRESSION-SIX  PREC-6-OPERATOR ]
                 EXPRESSION-FIVE


EXPRESSION-FIVE ::=
     [ EXPRESSION-FIVE  PREC-5-OPERATOR ]
                 EXPRESSION-FOUR


EXPRESSION-FOUR ::=
     [ EXPRESSION-FOUR  PREC-4-OPERATOR ]
                 EXPRESSION-THREE
```

```
520  EXPRESSION-THREE ::=
521       [ EXPRESSION-THREE PREC-3-OPERATOR ]
522                    EXPRESSION-TWO
523
524
525
526  EXPRESSION-TWO ::=
527       [ EXPRESSION-TWO  PREC-2-OPERATOR ]
528                    EXPRESSION-ONE
529
530
531
532  EXPRESSION-ONE ::=
533     {{PRIMITIVE-EXPRESSION
534          [ PREC-1-OPERATOR EXPRESSION-ONE ] }}
535     MONADIC-OPERATOR EXPRESSION-ONE
536
537
538
539  PRIMITIVE-EXPRESSION  ::=
540       { SYMBOL-OR-CONSTANT                    }
541       { "(" EXPRESSION-SEVEN ")"              }
542
543
544
545  BIT-ONE-EXPRESSION-SEVEN ::=
546       EXPRESSION-SEVEN
547
548
549
550  CLOCK-EXPRESSION-SEVEN ::=
551       EXPRESSION-SEVEN
552
553
554
555  DURATION-EXPRESSION-SEVEN ::=
556       EXPRESSION-SEVEN
557
558
559
560  INTEGER-EXPRESSION-SEVEN ::=
561       EXPRESSION-SEVEN
562
563
564
565  PREC-7-OPERATOR ::=
566       { "OR"   }
567       { "EXOR" }
568
569
570
571
```

```
572  PREC-6-OPERATOR ::=
573          "AND"
574
575
576
577  PREC-5-OPERATOR ::=
578          "==" / "EQ" /
579          "/=" / "NE"
580
581
582
583  PREC-4-OPERATOR ::=
584          "<"  / "LT" /
585          ">"  / "GT" /
586          "<=" / "LE" /
587          ">=" / "GE"
588
589
590
591  PREC-3-OPERATOR ::=
592          "+" /
593          "-" /
594          "><" / "CSHIFT" /
595          "SHIFT"
596
597
598
599  PREC-2-OPERATOR ::=
600          "*" /
601          "/" /
602          ">*<" / "CAT" /
603          "//"
604
605
606
607  PREC-1-OPERATOR ::=
608          "**"/
609          "UPB" /
610          "FIT"
611
612
613
614  MONADIC-OPERATOR ::=
615        "+" / "-" /
616        "NOT" / "ABS" /
617        "SIGN" / "ROUND" /
618        "TOFIXED" / "TOFLOAT" /
619        "TOBIT" / "TOCHAR" /
620        "ENTIER"
621
622
623
```

```
624  TRF-OF-CTRL-STATEMENT  ::=
625        ⎧ GOTO-STATEMENT          ⎫
626        ⎪ CONDITIONAL-STATEMENT   ⎪
627        ⎪ CASE-STATEMENT          ⎪
628        ⎩ REPEAT-STATEMENT        ⎭
629
630
631
632  GOTO-STATEMENT ::=
633      "GOTO" LABEL-IDENTIFIER
634
635
636
637  CONDITIONAL-STATEMENT  ::=
638      "IF" BIT-ONE-EXPRESSION-SEVEN
639           "THEN"   STATEMENT ...
640           [ "ELSE"   STATEMENT ...   ]
641      "FIN"
642
643
644
645  CASE-STATEMENT ::=
646      "CASE" INTEGER-EXPRESSION-SEVEN
647           { "ALT"   STATEMENT ... }...
648           [ "OUT"   STATEMENT ...   ]
649      "FIN"
650
651
652
653  REPEAT-STATEMENT ::=
654       [ "FOR" IDENTIFIER ]
655       [ "FROM" INTEGER-EXPRESSION-SEVEN ]
656       [ "BY" INTEGER-EXPRESSION-SEVEN ]
657       [ "TO" INTEGER-EXPRESSION-SEVEN ]
658       [ "WHILE" BIT-ONE-EXPRESSION-SEVEN ]
659       "REPEAT" [";"]  BLOCK-TAIL
660
661
662
663  CALL-STATEMENT ::=
664       "CALL" PROCEDURE-IDENTIFIER
665                 [ EXPRESSION-SEVEN-PACK ]
666
667
668
669  RETURN-STATEMENT ::=
670       "RETURN" [ "(" EXPRESSION-SEVEN ")" ]
671
672
673
674
675
```

```
REALTIME-STATEMENT ::=
      { ENABLE-STATEMENT
        DISABLE-STATEMENT
        SIGNAL-REACTION
        INDUCE-STATEMENT
        SYNCHRONIZER-OPERATION
        TASK-OPERATION          }


SYNCHRONIZER-OPERATION ::=
      { REQUEST-STATEMENT
        RELEASE-STATEMENT  }


TASK-OPERATION ::=
      { ACTIVATE-STATEMENT
        TERMINATE-STATEMENT
        SUSPEND-STATEMENT
        CONTINUE-STATEMENT
        RESUME-STATEMENT
        PREVENT-STATEMENT   }


ENABLE-STATEMENT ::=
     "ENABLE" INTERRUPT-IDENTIFIER  [ INTEGER-IN-BRACKETS ]


DISABLE-STATEMENT ::=
     "DISABLE" INTERRUPT-IDENTIFIER  [ INTEGER-IN-BRACKETS ]


INDUCE-STATEMENT ::=
     "INDUCE" SIGNAL-IDENTIFIER  [ INTEGER-IN-BRACKETS ]


SIGNAL-REACTION ::=
     "ON" { "," ' SIGNAL-IDENTIFIER
                  [ INTEGER-IN-BRACKETS ]''' } ":"
     [ UNLABELLED-STATEMENT ]


REQUEST-STATEMENT ::=
     "REQUEST" SEMAPHORE-IDENTIFIER
```

```
728  RELEASE-STATEMENT ::=
729       "RELEASE" SEMAPHORE-IDENTIFIER
730
731
732
733  ACTIVATE-STATEMENT ::=
734       [ SCHEDULE ]
735       "ACTIVATE" TASK-IDENTIFIER
736
737
738
739  TERMINATE-STATEMENT ::=
740       "TERMINATE" [ TASK-IDENTIFIER ]
741
742
743
744  SUSPEND-STATEMENT ::=
745       "SUSPEND"
746
747
748
749  CONTINUE-STATEMENT ::=
750       [ SCHEDULE-1 ]
751       "CONTINUE" TASK-IDENTIFIER
752
753
754
755  RESUME-STATEMENT ::=
756       SCHEDULE-1  "RESUME"
757
758
759
760  PREVENT-STATEMENT ::=
761       "PREVENT" TASK-IDENTIFIER
762
763
764
765  SCHEDULE ::=
766       { SCHEDULE-1 }
767       { SCHEDULE-2 }
768
769
770
771  SCHEDULE-1  ::=
772       { "AT" CLOCK-EXPRESSION-SEVEN                                    }
773       { "AFTER" DURATION-EXPRESSION-SEVEN                              }
774       { "WHEN" INTERRUPT-IDENTIFIER  [ INTEGER-IN-BRACKETS ]           }
775
776
777
778
779
```

```
780  SCHEDULE-2  ::=
781       ⎧"ALL"   ⎫
782       ⎨        ⎬ DURATION-EXPRESSION-SEVEN
783       ⎩"EVERY" ⎭
784       ⎡⎧"UNTIL" CLOCK-EXPRESSION-SEVEN      ⎫⎤
785     [ ⎨                                     ⎬ ]
786       ⎩"DURING" DURATION-EXPRESSION-SEVEN   ⎭
787
788
789
790  PRIORITY ::=
791       ⎧"PRIORITY" ⎫
792       ⎨           ⎬  SIMPLE-INTEGER-CONSTANT-DENOTATION
793       ⎩"PRIO"     ⎭
794
795
796
797  COMMUNICATION  ::=
798       ⎧ OPEN-STATEMENT     ⎫
799       ⎨ CLOSE-STATEMENT    ⎬
800       ⎩ TRANSFER-OPERATION ⎭
801
802
803
804  OPEN-STATEMENT ::=
805       "OPEN" DATION-IDENTIFIER [ INDEX ]
806            [ "BY" OPEN-CONTROL-LIST ]
807
808
809
810  OPEN-CONTROL-LIST ::=
811      { "," ` OPEN-CONTROL ``` }
812
813
814
815  CLOSE-STATEMENT ::=
816       "CLOSE" DATION-IDENTIFIER [ INDEX ]
817            [ "BY" CLOSE-CONTROL-LIST ]
818
819
820
821  INDEX ::=
822       INTEGER-IN-BRACKETS
823
824
825
826  CLOSE-CONTROL-LIST ::=
827      { "," ` CLOSE-CONTROL ``` }
828
829
830
831
```

```
832  CLOSE-CONTROL  ::=
833        ***IMPLEMENTATION-DEPENDENT***
834
835
836
837  TRANSFER-OPERATION ::=
838        ⎧ GET-STATEMENT     ⎫
839        ⎪ PUT-STATEMENT     ⎪
840        ⎨ TAKE-STATEMENT    ⎬
841        ⎪ SEND-STATEMENT    ⎪
842        ⎪ READ-STATEMENT    ⎪
843        ⎩ WRITE-STATEMENT   ⎭
844
845
846
847  INPUT ::=
848      {VAR-OBJECTS "FROM" DATION-IDENTIFIER [ INDEX ]
849              [ "BY" C-LIST ]} /
850      {"FROM" DATION-IDENTIFIER [ INDEX ] "BY" C-LIST }
851
852
853
854  OUTPUT ::=
855      {INV-OR-VAR-OBJECTS
856        "TO" DATION-IDENTIFIER [ INDEX ] [ "BY" C-LIST ]}/
857      {"TO" DATION-IDENTIFIER [ INDEX ] "BY" C-LIST }
858
859
860
861  GET-STATEMENT  ::=
862        "GET" INPUT
863
864
865
866  PUT-STATEMENT  ::=
867        "PUT" OUTPUT
868
869
870
871  TAKE-STATEMENT ::=
872        "TAKE" INPUT
873
874
875
876  SEND-STATEMENT ::=
877        "SEND" OUTPUT
878
879
880
881  READ-STATEMENT ::=
882        "READ" INPUT
883
```

17

```
884  WRITE-STATEMENT ::=
885       "WRITE" OUTPUT
886
887
888
889  INV-OR-VAR-OBJECTS ::=
890      { "," 'SYMBOL-OR-CONSTANT-OR-SLICE ''' }
891
892
893
894  SYMBOL-OR-CONSTANT-OR-SLICE ::=
895        { CONSTANT-DENOTATION  }
896        { SYMBOL-OR-SLICE      }
897
898
899
900  SYMBOL-OR-CONSTANT ::=
901        { SYMBOL               }
902        { CONSTANT-DENOTATION  }
903
904
905
906  VAR-OBJECTS  ::=
907      { "," 'SYMBOL-OR-SLICE ''' }
908
909
910
911  SYMBOL-OR-SLICE ::=
912      { SYMBOL                                                    }
913      { { IDENTIFIER "(" SIMPLE-INTEGER-CONSTANT-DENOTATION       }
914      {       [ ":" SIMPLE-INTEGER-CONSTANT-DENOTATION ] ")" }   }
915
916
917
918  SYMBOL ::=
919       IDENTIFIER  [ EXPRESSION-SEVEN-PACK ]
920          ["."  IDENTIFIER ]
921          ["."  "BIT" INTEGER-IN-BRACKETS ]
922
923
924
925  C-LIST ::=
926        { REMOTE-FORMAT    }
927        { STANDARD-C-LIST  }
928
929
930
931  REMOTE-FORMAT  ::=
932       "R" "(" FORMAT-IDENTIFIER ")"
933
934
935
```

```
STANDARD-C-LIST ::=
    { "," C-LIST-ELEMENT ... }



C-LIST-ELEMENT ::=
         {[ MULT ] { POS-CONTROL        }  }
         {         { MATCHING-CONTROL   }  }
         {  MULT "(" STANDARD-C-LIST ")"   }



MULT ::=
       INTEGER-IN-BRACKETS



POS-CONTROL  ::=
         { ABS-POS-CTRL }
         { REL-POS-CTRL }



ABS-POS-CTRL ::=
        {{"COL"  }                                        }
        {{       } "(" SIMPLE-EXPR ")"                    }
        {{"LINE" }                                        }
        { "POS" "(" SIMPLE-EXPR                           }
                    [ "," SIMPLE-EXPR                     }
                        [ "," SIMPLE-EXPR ] ] ")"         }



REL-POS-CTRL ::=
        {{"X"    }                                        }
        {{"SKIP" } "(" SIMPLE-EXPR ")"                    }
        {{"PAGE" }                                        }
        { "ADV" "(" SIMPLE-EXPR                           }
                    [ "," SIMPLE-EXPR                     }
                        [ "," SIMPLE-EXPR ] ] ")"         }



OPEN-CONTROL ::=
        { "IDF" "(" SYMBOL-OR-CONSTANT ")" }
        { "OLD"                            }
        { "NEW"                            }
        { [ "ANY" ]                        }
```

16

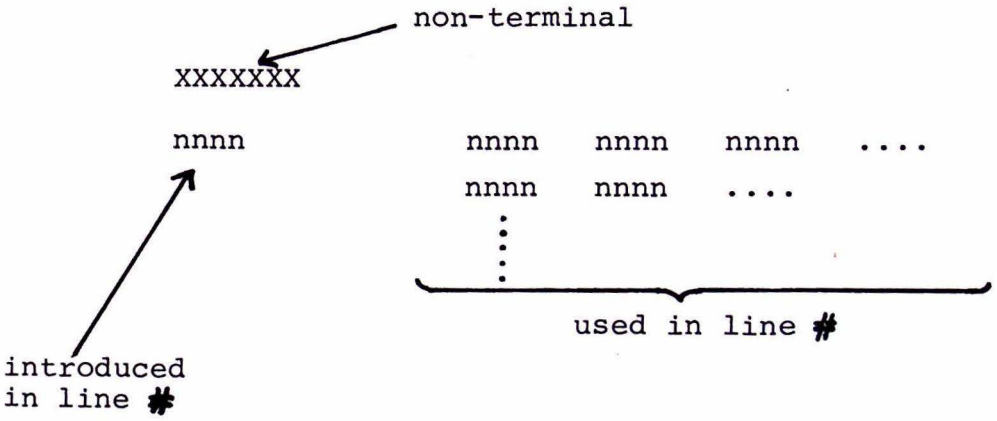
```
MATCHING-CONTROL ::=
     {{"A" / "R" / "B3" / "B4"}
                          [ "(" SIMPLE-EXPR ")" ] } /
     {{"D" / "T"} "(" SIMPLE-EXPR
                          [ "," SIMPLE-EXPR ] ")" } /
     {{"E" / "F"} "(" SIMPLE-EXPR
                          [ "," SIMPLE-EXPR
                            [ "," SIMPLE-EXPR ] ] ")" } /
       "LIST"


SIMPLE-EXPR  ::=
     {{["+"] /"-" } SIMPLE-INTEGER-CONSTANT-DENOTATION }
      IDENTIFIER


SYSTEM-DIVISION ::=
      "SYSTEM"  ";"
      [ CONNECTION ";" ]***


CONNECTION  ::=
     [ { USER-DEVICE-IDENTIFICATOR [ ARRAY-BOUNDS ] ":"}***]
          [ SYSTEM-DEVICE-IDENTIFICATOR
                          [ INDEX-OR-ARRAY-BOUNDS ] ]

     [    TRANSFER-DIRECTION

          [ { "+" * CONNECTION-POINT-DESCRIPTION *** } ] ]


CONNECTION-POINT-DESCRIPTION ::=
     SYSTEM-DEVICE-IDENTIFICATOR  [ INDEX-OR-ARRAY-BOUNDS ]
          {{SIMPLE-INTEGER-CONSTANT-DENOTATION
     [ "*"     [ , SIMPLE-INTEGER-CONSTANT-DENOTATION ]}}]
          { ARRAY-BOUNDS


USER-DEVICE-IDENTIFICATOR ::=
     IDENTIFIER


SYSTEM-DEVICE-IDENTIFICATOR ::=
     IDENTIFIER
```

```
1040  TRANSFER-DIRECTION ::=
1041        { "<-"  }
1042        { "<->" }
1043        { "->"  }
1044
1045
1046
1047  ARRAY-BOUNDS ::=
1048       "(" SIMPLE-INTEGER-CONSTANT-DENOTATION
1049             ":" SIMPLE-INTEGER-CONSTANT-DENOTATION ")"
1050
1051
1052
1053  INDEX-OR-ARRAY-BOUNDS  ::=
1054       "(" SIMPLE-INTEGER-CONSTANT-DENOTATION
1055           [ ":" SIMPLE-INTEGER-CONSTANT-DENOTATION ] ")"
1056
1057
1058
```

10

C Cross Reference List

The following Cross Reference List contains all non-terminals of the Basic PEARL Syntax in alphabetical order. With each non-terminal the line of its production rule is given and a list of lines, where it is used in rule bodies.



DATION-SPEC-ATTR
289 252 334

DECLARATIONS-AND-SPECIFICATIONS
214 210

DIGIT
12 20 37 115 126 126 128 130 135
135 145

DIM
406 395

DIM-SPEC
312 302

DISABLE-STATEMENT
707 678

DURATION-CONSTANT-DENOTATION
171 93

DURATION-EXPRESSION-SEVEN
555 773 782 786

ENABLE-STATEMENT
702 677

EXPONENT-PART
134 127 130

EXPRESSION-FIVE
507 503 508

EXPRESSION-FOUR
513 509 514

EXPRESSION-ONE
532 528 534 535

EXPRESSION-SEVEN
495 480 491 496 541 546 551 556 561
670

EXPRESSION-SEVEN-PACK
490 665 919

```
EXPRESSION-SIX
 501                  497   502

EXPRESSION-THREE
 520                  515   521

EXPRESSION-TWO
 526                  522   527

FORMAT-IDENTIFIER
  72                  932

GET-STATEMENT
 861                  838

GLOBAL-DECLARATION
 343                  219

GLOBAL-SPEC-ATTRIBUTES
 249                  244

GLOBAL-SPECIFICATION
 241                  218

GOTO-STATEMENT
 632                  625

HEXADECIMAL-DIGIT
  19                  159

HOURS
 178                  172

IDENTIFIER
  34                   42    43    48    53    58    63    68    73
                       78    83   654   913   919   920  1002  1032
                     1037

INDEX
 821                 805   816   848   850   856   857

INDEX-OR-ARRAY-BOUNDS
1053                 1015  1024

INDUCE-STATEMENT
 712                 680
```