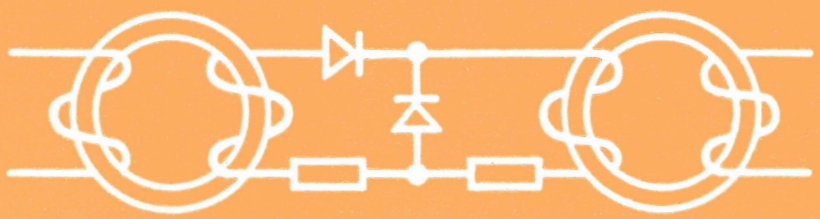
Reprint

elektronische datenverarbeitung

electronic data processing



V 17/86

Friedr. Vieweg + Sohn Braunschweig

Wichtige Themen im Jahr 1969:
Important topics of 1969:

Beister, H.-J., Zur Technik der Segmentierung von Programmen bei Rechenanlagen mit kleinem Kernspeicher

Berger, B./Seibt, D./Strunz, H., Bibliographie des Betriebswirtschaftlichen Instituts für Organisation und Automation an der Universität zu Köln zum Thema „Programmiersprachen"

Berr, U./Müller, H. E.-W., Ein heuristisches Verfahren zur Raumverteilung in Fabrikanlagen

Christensen, K., Moderne Methode zur Datenerfassung, Die direkte Belegverarbeitung durch optisches Lesen off-line

Christodoulopoulos, A. F., Die Strukturierung der dezentralen Datenerfassung eines Handelsunternehmens mit mehreren Verkaufsniederlassungen

Clausert, H./Kahlert-Warmbold, I., Untersuchung digitaler Schaltungen mit dem Programmsystem DICAP

Eckmann, J.-P., Gedanken zur rekursiven Programmierung von PERT-Problemen

Engels, H., Ein modifiziertes vorzeichenrichtiges Graeffe-Verfahren

Fehler, D. W., Die Variationen-Enumeration – ein Näherungsverfahren zur Planung des optimalen Betriebsmitteleinsatzes bei der Terminierung von Projekten

Feichtinger, G., Ein Markoffsches Lernmodell für Zwei-Personen-Spiele

Fialkowski, L., Die Zeichenmaschinensteuerung GEAGRAPH 2000

Gallrein, D., Das Zuordnungsverfahren im automatischen Multi-Radar-Beobachtungssystem

Grochla, E., Betriebsinformatik und Wirtschaftsinformatik als notwendige anwendungsbezogene Ergänzung einer allgemeinen Informatik

Günther, G., Neuere Fortschritte bei der Fourier-Transformation

Heinrichsdorf, F., Numerische Methoden bei der Berechnung großer Netze mit Hilfe elektronischer Rechenanlagen in einfacher Darstellung

Henke, M., Grenzwerte-Approximation dynamischer Optimierungsansätze, angewandt auf ein sequentielles Investitionsproblem

Hirsch, H.-J., Zur formalen Beschreibung von Automatennetzen I

Holtz, G./Spaniol, O./Stucky, W., Statistische Untersuchungen von Digitalfiltern

Jankowski, E., Die Planung von EDV-Organisationen in der Verwaltung

Klevers, E., Automatische Zeichensysteme in der Datenverarbeitung

Kremer, H., Praktische Berechnung des Spektrums mit der Schnellen Fourier-Transformation

Kupper, H., Computer und musikalische Kompositionen

Köhler, R., Entwicklung und Situation der Elektronischen Datenverarbeitung in der DDR

Lang, B., Wie arbeitet ein computergesteuertes Warenverteilungszentrum?

Lazak, D., Programmierung und Test einfacher Kollisionsstrategien zum Stundenplanproblem

Lazak, D., Datenverarbeitung in der Hochenergiephysik Auswertung von Blasenkammeraufnahmen

Lewandowski, R., Zu einer internationalen Bibliographie der Netzplantechnik (1962–1967)

Lewandowski, R., Modelle und Methoden der ökonomischen Vorhersage

Lindemann, P./Nagel, K., Literatur zu betrieblichen Informationssystemen

Maravent, J., Anwendungsbereich eines Hybrid-Systems (EAI 690, 7945, 8900)

Marguerat, C./Contestabile, B., TPS II ein Programmsystem für die kapazitätsabhängige Feinplanung und Steuerung der Eigenteilefabrikation

Mansell, G. H., Lichtschreiber- und Computertechniken für den Entwurf von Bauplänen

Menne, K., Lochkarte oder Magnetband zur Datenerfassung

Müller-Merbach, H., Sensibilitätsanalyse von Transportproblemen der linearen Planungsrechnung (mit ALGOL-Programm)

Müller, R., Teilnehmer-Rechensysteme

Niedereichholz, J., Some Extensions of the Flow Graph Theory

Olbrich, J./Maciey, S./Loleit, D., Die physische Installation von EDV-Anlagen

Pape, U., Kürzeste Wege in asymmetrischen Netzwerken zwischen einem festen Knoten und beliebigen Knoten

Pape, U., Eine Bibliographie zu „Kürzeste Weglängen und Wege in Graphen und Netzwerken"

v. Peschke, J., Eine als Metasprache verwendbare höhere Programmiersprache

Poths, W., Die Bedeutung problemorientierter SOFT-WARE für die Gestaltung betrieblicher Anwendungssysteme

Schmidt, W. P., Grundlagen, Verfahren und Datenorganisation für die Teilebedarfsermittlung (Stücklistenauflösung)

Scholz, H./Steinbock, R., Die Untersuchung der Wirtschaftlichkeit einer automatisierten Datenverarbeitung (ADV)

Schön, B., Ein ALGOL-Programm zur Lösung von Engpaß-Zuordnungsproblemen

Seed, M. C./Beddoes, R. L., Rapid, Automatic Crystallographic Data Collection Evolution of the Computer-controlled Fourcircle Diffractometer

Weber, B., Installationsplanung einer EDV-Anlage mit Hilfe der Netzplantechnik

Wedekind, H., Die Konstruktion eines optimalen Zugriffsbaumes für Datenorganisationen innerhalb einer Datenbank

---

12. Jahrgang, 1970 – 12 Hefte jährlich

Abonnementspreise:
1 Jahr DM 126,– zuzüglich Versandspesen
2 Jahre DM 226,80 zuzüglich Versandspesen
Einzelheft DM 15,–

Volume 12, 1970 – 12 issues

Subscription rates:
1 year $ 36.00 £ 14.10.0 (plus postage)
2 years $ 65.00 £ 26.0.0 (plus postage)
Single copy $ 4.20 £ 1.15.0

Hauptschriftleiter: **Dr. P. Schmitz, Köln**
Redaktion: Dipl.-Volkswirt W. Eicken

elektronische datenverarbeitung

Fachberichte über programmgesteuerte Maschinen und ihre Anwendung

Heft 10/1970
12. Jahrgang

PEARL*)

The Concept of a Process- and Experiment-oriented Programming Language

J. Brandes[1]), S. Eichentopf[2]), P. Elzer[3]), L. Frevert[4]),
V. Haase[1]), H. Mittendorf[5]), G. Müller[6]), P. Rieder[5])

Summary: The main features of a programming language for on-line-control of industrial processes and scientific experiments are described. Mainly discussed are the connections between the program and the process environment, the management of storage space and parallel activities, the synchronization of tasks, and the non-standard input-output. The paper describes structure and semantics of the language, but does not yet give an exact and definitive syntax.

Zusammenfassung: Dieses Konzept beschreibt die wesentlichen Eigenschaften einer Programmiersprache der Mittelebene für Anwendungen in der industriellen Prozeßsteuerung und in der Experimentiertechnik. Es werden hauptsächlich die Verkopplung des Programms mit der Prozeßumwelt, die Speicherverwaltung, die Steuerung und Synchronisierung der Parallelarbeit und nicht standardmäßige Ein-/Ausgabemöglichkeiten beschrieben. Der vorliegende Bericht beschreibt Struktur und Semantik der Sprache, nicht aber eine exakte Syntax, die noch von der "PEARL"-Arbeitsgruppe erarbeitet werden wird.

1. Introduction

In recent years it has been realized by many people that higher level programming languages like those for commercial or scientific applications will be advantageous and even necessary also in the fields of process and experiment automation [1, 2, 3]. Some proposals, such as RTL [4] or an extension of PL/1 [5] also proved the fundamental possibility of expressing specific characteristics of process and experiment programs by a higher level programming language.

The possibility of implementation had been doubted for quite a long time; however, it has been demonstrated meanwhile by the existence of some solution applicable to partial fields [6, 7].

1.1. Fundamentals of the "PEARL" Concept

"PEARL" is a compilable programming language of the intermediate level, such as ALGOL or PL/1, since this type of language offers maximum flexibility in addition to easy handling and learnability. Since a relatively large

*) Process and experiment automation realtime language

[1]) GfK-DVZ, Karlsruhe
[2]) AEG-Telefunken, Konstanz
[3]) Physikalisches Institut III der Universität Erlangen
[4]) Hahn-Meitner-Institut für Kernforschung, Berlin
[5]) Siemens AG, Karlsruhe
[6]) Fa. BBC, Mannheim

number of language elements are required in order to be able to treat also more complex problems, it is intended to define self-contained, fully upward compatible subsets for smaller computing systems. A brief summary is given of both the main features of a program for process applications, which practically prevent its being written in a conventional programming language (FORTRAN, ALGOL 60, COBOL, etc.), and means of solution proposed by the "PEARL" concept.

1.1.1. Machine Dependence

In a computer program for process application also the entire peripheral equipment must be described and selected which, in most cases, may be of non-standard design and more diversified than in commercial or scientific applications of electronic data processing. Due to the improbability of standardizing all possible hardware configurations and describing them uniformly at the level of an advanced programming language, a "PEARL" program is subdivided in principle into a system dependent part, the system division, and the problem division which is largely independent of the system. The system division provides the link between the process peripheral equipment and the program, whilst the problem division constitutes the control or measuring program proper. Thus, only the system division has to be modified when passing from one type of computer to another (Chapter 3.).

1.1.2. Parallel Activities

Usually a process program during operation splits up into a varying number of parallel "tasks" [8] some of which are performed completely independent of each other while others are correlated to each other, i. e., they have to be synchronized. The "task" is considered to be dynamical, i. e., it consists in the successive processing of one or several pieces of code. The process program is characterized by the interaction between the tasks which is made possible by synchronization with the help of semaphores [9] (Chapter 4.1.-4.4.).

1.1.3. Running Time and Storage Management

In a "normal" computer program, the aggregate computing time in principle does not matter. It is sufficient that the program will be terminated within a period of time reasonable for the user. However, since a process program, in most cases, must meet preset reaction times which, in the most extreme cases, may be of the order of a few cycle periods (e. g., application in nuclear physics experiments) the programmer must have more influence on the object code than he has in conventional programming languages.

A related problem is the management of storage allocation. Since often parts of the program code will be stored on some backing storage medium, the time for loading the code must be included into the running time of a task. The programmer must have the possibility to act on the time of loading as well as on the size of the code pieces to be loaded, though the working storage allocation shall be automated as far as possible (Chapter 4.5.).

1.1.4. Input/Output

In process programs there must be much more possibilities of input and output than in programs for commercial, scientific, or technical applications.

A concept describing input and output was developed which will allow the handling of nearly all the input and output operations encountered in process technique under uniform aspects (Chapter 5.).

1.1.5. Types of Data and Language Features

In this respect, the familiar programming languages (ALGOL 60 and FORTRAN) prove to be unsufficient. Therefore it became necessary to include in the language more complex types of data, such as lists or structures. It has been attempted to combine all the elements of modern programming languages which are required for process purposes (PL/1, ALGOL 68), but to eliminate too expensive and rarely used types and, if necessary, replace them by new elements [10, 11] (Chapter 2.).

1.1.6. Relations to the Operating System

A programming language allowing interventions into functions of the operating system necessitates also a certain standardization of the properties of the operating system, e. g., uniform treatment of the interrupts. To obtain efficiency and satisfactory compatibility of process computing programs in a higher level programming language, the definition of certain minimum requirements will be needed which must be met by an operating system for a process computer.

1.2. Mode of Description

It has intentionally been avoided in this concept to propose an exact syntax of "PEARL". In exceptional cases where this is done, however, the usual notation of PL/1 is adopted [12].

2. Basic Language

2.1. Fundamentals

2.1.1. Character Set

The character set consists of approx. 60 characters including the latin alphabet, the decimal digits and certain special characters which are in common use.

2.1.2. Delimiters

Delimiters consist either of sequences of $n\,(n \geq 1)$ special characters (except blanks), or of sequences of $m\,(m \geq 2)$ letters (keywords). How or whether keywords are to be reserved, i. e. identical identifiers will have to be forbidden, depends on the exact definition of the language structure and, therefore, cannot be decided now.

2.1.3. Comments

A comment may be placed at every location in a program where a blank may be written; this comment will be enclosed by special characters as in PL/I.

2.1.4. Block Structure and Procedures

The language has a block structure similar to that of ALGOL 60 and PL/I with its consequences, especially regarding the scope of declarations.

Procedures will be treated similarly to those in ALGOL 60; to simplify matters, however, certain restrictions must be observed, e. g. regarding call by name of parameters. All formal parameters must be specified (as in ALGOL 60), [13].

2.1.5. Identifiers and Declarations

Identifiers are sequences of alphanumeric characters in which the first character is a letter. Identifiers, in general, refer to data (values) of a certain type (variables).

All identifiers used must be declared in the program; the scope of the declaration is the block in which the declaration is contained. Identifiers can also be declared, in certain cases, in the `system division` (see 3.), where the scope of the declaration is the entire program. Initialization of identifiers in declarations is possible.

Declarations not necessarily stand at the beginning of a block. If, however, an identifier is to be initialized in its declaration, this initialization must be dynamically executed before the identifier is used for the first time.

Besides identifier declarations, there are also type declarations (see 2.2.2.2.) and operator declarations (see 2.3.2.).

2.2. Types

There are certain basic types from which, according to specific rules, compound types can be formed.

2.2.1. Basic Types

Basic types are INTEGER and REAL of different precision, BINAL (bit string) of different length, STRING (character string) of different length, SEMA, FILE and ADDRESS. A type BOOLEAN or LOGICAL is not required, as it can be represented with BINAL of length 1.

The declaration symbols and the forms of constants will be decribed in the following sections.

2.2.1.1. INTEGER and REAL

Declaration symbols: INTEGER (n) and REAL (n) where n is the minimum number of decimal digits to be comprised within the mantissa. If (n) is missing, an implementation-dependent stadard length or precision will be assumend. By writing REAL (n), the next implemented mantissa length, which is not less than (n), will be taken.

Constants of the type INTEGER will be specified as a sequence of $n\,(n \geq 1)$ consecutive decimal digits. Constants of the type REAL will be specified, similarly to PL/I, as decimal constants which are not integer constants. Blanks within one number are not permitted. The internally used mantissa length is the next implemented one which is not smaller than the number of digits in the constant.

2.2.1.2. BINAL

Declaration symbol: BINAL (n) where (n) is the number of bits. If (n) is missing, an implementation-dependent standard length will be taken.

In arrays and structures BINALs will be closely packed within limits of the effective addressability, in order to save storage capacity. For BINALs not in arrays and structures, the next implemented bit string length, not less than (n), will be taken.

Constants of the type BINAL can also be specified in octal notation with the digits 0 to 7, and in sedecimal notation with the digits 0 to 9, and the letters A to F.

Example: '110100101011'B binary
'6453'O octal
'D2B'S sedecimal

The notation of the BINAL constants is not as yet finalized.

2.2.1.3. STRING

Declaration symbol: STRING (n)
where (n) is the maximum number of characters contained in a string.
STRING constants will be enclosed in apostrophies.
Example: 'ANTON+BERTA'

2.2.1.4. SEMA

Declaration symbol: SEMA
A variable of the type SEMA can only assume integer values and is used to synchronize tasks (see 4.4.).

2.2.1.5. FILE

Declaration symbol: {SEQUENTIAL|RANDOM} FILE (unit identifier, pages/file, lines/page, characters/line)

FILE identifiers occur as parameters in I/O procedures. There are two types of FILEs, the SEQUENTIAL FILE in which only sequential reading and writing is possible and the randomly addressable RANDOM FILE.

FILE specifies data organized in pages, lines and characters and stored on external storage [11]. The declaration contains, therefore, the name of the unit (e. g. drum, disc), the number of pages per file, lines per page and characters per line.

2.2.1.6. ADDRESS

Declaration symbol: ADDRESS

Variables of the type ADDRESS have as values ADDRESS constants, i. e. identifiers (also subscripted) of variables which must not be of the type ADDRESS.

There are three address levels:

Level 2: ADDRESS variable, i. e. identifiers which indicate values of the type ADDRESS;
↓
Level 1: ADDRESS constants, i. e. identifiers which indicate values excluding those of type ADDRESS;
↓
Level 0: Values which are not of the type ADDRESS.

The following addressing mode is used for assignment: The left side of an assignment, i. e. the side to which the right side is assigned, must be on level 1 or 2 of the address levels mentioned above. The level of the right side of an assignment, i. e. the side which will be assigned, must come directly under the level of the left side.

Hence:

Left side	Right side
Level 2	Level 1
Level 1	Level 0

Accordingly, the level of the right side will be automatically adjusted to the specified level of the left side, as far as this is possible.

Example:

```
REAL A, B;
ADDRESS POINTER 1, POINTER 2;
```

```
A := 5.7
   /* LEFT: LEVEL 1; RIGHT: LEVEL 0; CLEAR. */;

B := A
   /* LEFT: LEVEL 1;
      RIGHT: NEXT ALSO LEVEL 1, AFTER ADJUST-
      MENT TO LEFT SIDE LEVEL 0, I. E. THE VALUE
      (HERE 5.7), INDICATED BY A, WILL BE
      ASSIGNED TO B. */;

POINTER 1 := A
   /* LEFT: LEVEL 2;
      RIGHT: LEVEL 1;
      THE ADDRESS-VARIABLE POINTER 1 ASSUMES
      IDENTIFIER A AS VALUE. */;

POINTER 2 := POINTER 1
   /* LEFT: LEVEL 2;
      RIGHT: NEXT ALSO LEVEL 2, AFTER ADJUST-
      MENT TO LEFT SIDE LEVEL 1, I. E. THE VALUE
      (HERE A), INDICATED BY THE POINTER 1 WILL
      BE ASSIGNED TO POINTER 2, AFTER WHICH
      BOTH ADDRESS-VARIABLES INDICATE A. */;

POINTER 1 := 3.9
   /* LEFT: LEVEL 2;
      RIGHT: LEVEL 0;
      FALSE, BECAUSE BOTH SIDES ARE NOT AUTO-
      MATICALLY ADJUSTABLE. */;

VALUE POINTER 1 := 3.9
   /* LEFT: LEVEL 1 BECAUSE OF THE OPERATOR

      VALUE APPLICABLE TO THE ADDRESS-VARIA-
      BLE; RIGHT: LEVEL 0;

      EFFECT: AS FOR A := 3.9, BECAUSE POINTER 1

      INDICATES A. */;
```

Also see example under 2.2.3.

Further details regarding the handling of ADDRESS variables are not, as yet, finalized.

2.2.2. Compound Types

2.2.2.1. Arrays

Arrays consist of elements of the same type. This type can be a basic type with the exception of SEMA and FILE, or a structure.

An example of a declaration for a two-dimensional array with elements of the type REAL and with the identifier MATRIX is as follows:

```
(5:100, 1:7) REAL(10) MATRIX
```

When transposing arrays to storage, the last subscript, i. e. the subscript at the extreme right-hand location, will be transposed one at a time ("line" transposition).

An element in an array will be addressed by an array identifier and a subscript list.

```
Example: MATRIX(7*J, 3)  /* E. G.  J = 4 */
```

Furthermore, sections of an array can be addressed, which consist of more than one element.

Examples: 1) Call of the entire second column of the matrix declared above:

```
MATRIX(, 2) or MATRIX(*, 2)
```

2) Call of a section of the second column of the matrix declared above:

```
MATRIX(8:N, 2)  /* E. G.  N = 55 */
```

According to this, the whole matrix could be addressed by

```
MATRIX(,) or MATRIX(*,*)
```

the array identifier alone is, however, sufficient for this purpose.

Examples of array constants:

1) Constant vector with three components:
   (6, 7, 9)

2) A matrix
$$\begin{pmatrix} 3 & -5 \\ 7 & 9 \end{pmatrix}$$
   will be written as:
   ((3, - 5), (7, 9))

2.2.2.2. Structures

Structures will be formed from elements of (not necessarily) different types. The type of a particular element can be a basic type with the exception of SEMA and FILE or a structure or an array.

Structure identifiers will be declared similarly to those in ALGOL 68, whereby the structure type is used either directly as the declaration symbol, or by using a declaration symbol which has been explicitely declared in a type declaration.

Example: Declaration of an identifier (e. g. C) for complex values:

1st possibility:
```
STRUCTURE (REAL RE, REAL
IM) C; or shorter;

STRUCTURE (REAL RE, IM)
C;
```

2nd possibility:
```
TYPE COMPLEX = STRUC-
TURE (REAL RE, IM)
/* PURE TYPE DECLARA-
TION */;
COMPLEX C
/* (NOT NECESSARILY
DIRECTLY) FOLLOWING
IDENTIFIER DECLARATION */;
```

The declaration of a structure type therefore contains a list of the elements of the structure. The list elements each contain a declaration symbol (e. g. REAL, INTEGER etc.) and an identifier which is used to address the structure element so designated (see below). If the same declaration symbol applies to a consecutive sequence of list elements, it only has to be located with the first list element (see1st possibility in example above).

A structure element is addressed by an element identifier and a structure identifier.

Example: RE OF C resp. IM OF C
or shorter:

```
RE.C      resp. IM.C
```

An element of a structure, which is itself an element of an array, will be addressed by, e. g.

ELEMENT 3 . ARRAY (I)

An element of an array, which is itself an element of a structure, will be addressed by, e. g.

ELEMENT 2 (I) . STRUCTURE

Examples of structure constants:

1) (5.7, '101'B, 115, 'ABC')
2) Two element structure constant, whose first element is also a structure:
   ((5.7, 'ABC'), 97)
3) Structure constant, whose second element is an ADDRESS constant:
   (5.7, ANTON)
4) Structure constant, whose second element is a one-dimensional three element array constant:
   (5.7, (1, 2, 1))

2.2.3. Example of a circular concatenated list

```
(1:100) STRUCTURE (REAL CONTENTS, ADDRESS
REFERENCE) LIST
   /* DECLARATION OF THE IDENTIFIER LIST WHICH
      INDICATES AN ARRAY OF STRUCTURE
      ELEMENTS. */;
FOR I TO 99 DO REFERENCE . LIST (I) := LIST (I+1);
REFERENCE . LIST (100) := LIST (1)
   /* OCCUPANCY OF THE STRUCTURE ELEMENT
      REFERENCE IN ALL LIST ELEMENTS WITH THE
      SUBSCRIPTED IDENTIFIER OF EACH OF THE
      NEXT LIST ELEMENTS. */;
ADDRESS POINTER := LIST (100)
   /* DECLARATION OF THE ADDRESS-VARIABLE
      POINTER WITH INITIALIZATION. */;
```

The result is illustrated by figure 1.

An entry into the list, thus prepared, will be effected by the following two statements:

```
POINTER := REFERENCE . VALUE POINTER
   /* SET POINTER TO NEXT LIST ELEMENT. */;
CONTENTS . VALUE POINTER := /* CONTENTS TO BE
      ENTERED */;
```

2.2.4. Conversions between data of different types

Type conversions which are required frequently in expressions, are done automatically. Standard functions are provided for further conversions. When the compiler recognizes that a type conversion is to be made in an expression, a warning message may be output.

2.3. Operators

2.3.1. Classification

Provision has been made for the monadic operators + and - for arithmetic operands, NOT for BINAL operands as well as VALUE for ADDRESS variables and the following dyadic operators:

1) for arithmetic operands:

   exponentiation
   integer division,
   remainder of integer division (modulo),
   normal division,
   multiplication,
   addition,
   subtraction
   and the conventional six comparisons
   ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$)

2) for BINAL operands:

   conjunction,
   disjunction,
   equivalence,
   antivalence,
   shift (direction indicated by the sign of the number of shift-steps), selection of an individual bit and concatenation.

3) for STRING operands:

   concatenation
   and comparisons;

4) for ADDRESS operands:

   comparisons.

2.3.2. Operator Declarations

Operator declarations are provided with the restriction that only operator symbols which already exist can be extended to newly declared types by using such a declaration; the existing operator priorities must not be altered.

2.4. Executable Statements

The following are provided:

- assignments,
- go to statement,
- repetitive statement similar to ALGOL 68,
- switch statement similar to CASE clause in ALGOL 68,
- compound statement,
- conditional statement (IF statement in ALGOL 60),
- interrupt response (ON statement, see 4.3.1.),
- task statements (see 4.2.)
- procedure statement (see 2.1.4.),
- statements for SEMA variables (see 4.4.3.).

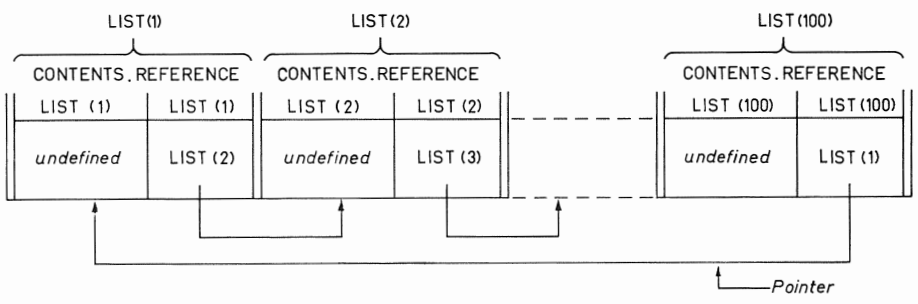


Fig. 1
Example of a circular concatenated list

2.5. Standard Functions and Standard Procedures

Besides the usual mathematical standard functions, standard functions and standard procedures for bit handling are also provided (selection of bit sequences from BINAL data); string handling (selection of characters and character sequences from STRING data), type conversions, input/output (see 5.) and special functions are likewise provided for.

3. The System Division

3.1. Purpose

The system division connects the actual process-program with the environment. It is used to describe the machine configuration on the language level.

It contains parts of a "job control language", it supplies information for the generation of that part of the operating system that is necessary for the special program and it allows an optimization of resource allocation. It further connects external devices and process interface hardware with symbolic names, that are to be used in the problem part.

An extensive use of symbolic names in the problem division makes programming easier and improves the readability of the computer program.

By the computer description the compiler is informed about e. g. the type, model and memory size of the machine used.

The configuration description specifies the extend and the structure of the used peripherals. Each external device and process-interface is connected with an identifier. In the same instance the datapath from the central unit to the terminal device is described. By naming only those peripherals that are used by the program, it may be possible to minimize the I/O-package linked to the program.

In the interrupt description the interrupts generated by the system are matched with identifiers, that can be referred to in the problem division. Here also a certain optimisation of the interrupt decoding routines may be performed.

In the flag description status information of the hardware is depicted onto BINAL strings, and can so be tested in the problem part.

3.2. Syntax of the System Division

```
SYSTEM;
MACHINE;
   :
/* COMPUTER DESCRIPTION */
   :
EQUIPMENT;
   :
/* CONFIGURATION DESCRIPTION */
   :
INTERRUPT;
   :
/* INTERRUPT DESCRIPTION */
   :
FLAG;
   :
/* FLAG DESCRIPTION */
   :
```

3.3. Computer Description

The begin of this part is identified by the keyword MACHINE, it contains information about:

a) type (keyword: MODEL)
b) features of the processing unit
c) size of working storage (keyword: SIZE)
d) channels (keyword: CHANNEL)

The information to be supplied in a special case will have to be provided in the corresponding manual.

As one "PEARL" compiler will be written for a whole computer family, it must be informed about the actual level and the features of the implementation, to be able to guarantee an optimal use of the computer by means of a specially optimized object program.

3.4. Configuration Description

3.4.1. Purpose

a) The operating system is informed about the peripheral devices to be used by the program. For the specified environment an optimal version of the operating system may be generated.

b) As the actual equipment with peripheral devices is not specific to the computer but to the whole installation, and can be changed from time to time, the whole datapath from the central unit to the terminal device must be described definitely. This can generally be done by means of an "address tree", the nodes of which are represented each by a control unit with special attributes and interface specifications. (Fig. 2)

c) Identifiers are adjoined to the special datapaths, which are used in the problem division.

3.4.2. Syntax

```
EQUIPMENT;
[STANDARD [; TOTAL]
[; identification] ... ; ]
[SPECIAL [; identification] ... ; ]
```

The metavariable identification is expanded as follows:

```
identification ::= identifier [(index-list)] =
                   [computer-link] {* | ≠} [channel-type
                   [, attribute] ... [(index-list)]]
                   {* | ≠} ... [terminal-type];
```

index-list is a list containing positive integers, similar to that in a loop-specification; an item of the list can be a single number (e. g. 2, 7, 11), a range of numbers (e. g. 3:10, 17:21), or a range of numbers with an increment (e. g. 4(3)16 means the series 4, 7, 10, 13, 16). * or ≠ represent a node in the address tree.

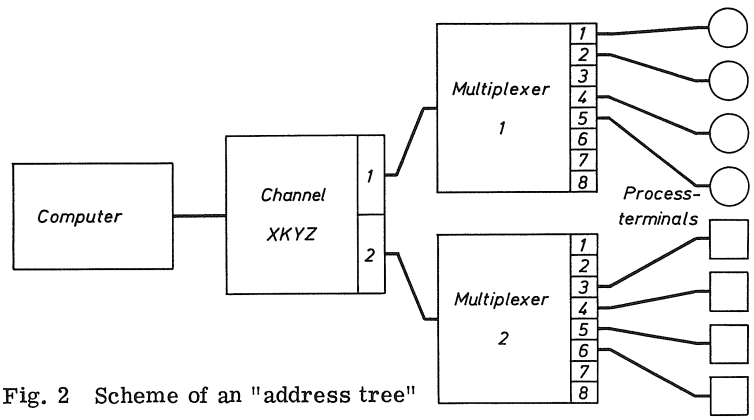


Fig. 2 Scheme of an "address tree"

≠ has an additional function: for two consecutive symbols ≠ that part of the previous datapath description that is enclosed in two symbols ≠ is substituted.

Examples:

1.) DISK = 1*KW, 2*ZWW, FLOAT*STØ5A;

The disk of type STØ5A shall be connected via the floating data-channel ZWW and the 2nd exit of the I/0-processor KW to exit 1 of the computer.

2.) The system of fig. 2 could be described as follows:

TERMINAL (1 : 4) = XKYZ, 1*MPX1 (1, 2, 4, 5)*CIRCLE;
TERMINAL (5 : 8) = XKYZ, 2*MPX2 (3 : 6)*SQUARE;

3.) Substitution of parts of the datapath description shows:

FIRST = 10 PROCA ≠ CHAN 1*CONTROL 2 ≠ DEVICE 1;
SECOND = 10 PROCB ≠≠ DEVICE 2
/* MEANS: SECOND = 10 PROCB ≠ CHAN 1*
CONTROL 2 ≠ DEVICE 1 */;

3.4.3. Semantic of the Configuration Description

EQUIPMENT introduces the configuration description, STANDARD that of the standard peripherals, SPECIAL that of process oriented peripherals. TOTAL means: all of the standard peripherals connected to the system shall be used.

Standard peripherals have a fixed datapath. They are identified by a name that is defined by the implementation (this "name" may also be an integer); it may be changed in the system division. Both identifiers may be used in the problem division. This is useful, if a program is composed of parts of different origin, or if peripherals are replaced by devices of another type (but with the same function with respect to the program).

identification describes the datapath by a series of type and address items. Details (the keywords and the local syntax of channel-types and attributes) depend on the implementation and can be taken from the manual and the hardware plan of the installation.

The identifier defined in the system division is used in the problem division as an actual parameter of I/0 and library procedures and stands for the device address resp. device number. Further parameters are not supplied by the system division.

If a one dimensional array of identifiers is adjoined to a set of identical process terminals, this is described by index-list.

Example:

APC (1 : 12) = CCTR*3*(1, 3, 5 : 7, 12 : 18)*5*KAD 5;
Also arrays of names may be redefined:
HEAT (1 : 50) = TEMP (20 : 69);

3.5. Interrupt description

3.5.1. Purpose

The operating system is informed about the interrupts to be used. These are "logical interrupts", which not necessarily have their origin in the hardware and may include additional information about the status change causing the interruption. It shall be possible to generate or simulate an interrupt by the program (see 4.3.2.).

3.5.2. Syntax

INTERRUPT;
[STANDARD [; TOTAL] [; interrupt-identifier] ... ;]
[SPECIAL [; TOTAL] [; interrupt-identifier] ... ;]

The metavariable interrupt-identifier is expanded as follows:

[identifier =] ... implementation dependent descriptor;

3.5.3. Semantics

The interrupts listed in the interrupt description can be addressed with the corresponding symbolic names in the problem division. The interrupts not mentioned in the system division are disarmed, disabled or handled by dummy routines. The priorities of the interrupts and the handling routines are prescribed by the order of the interrupt list, if this is allowed by the system.

Interrupts of type STANDARD have predefined names. It is desirable, that interrupts of the same origin should have identical names. The "PEARL"-committee will issue proposals for that.

Interrupts of type SPECIAL are handled over from the operating system to the user program; they are specific for the particular process.

Double definitions are possible.

Example: HEATALARM = PRESSUREALARM = 37;

In many cases the system supplies further information during an interrupt. It can be accessed by the standard procedure STATUS (interrupt-identifier).

3.6. Flag-Description

3.6.1. Flags are status messages of the peripherals or of the computer, that do not cause an interrupt. The operating system is informed, which of them will be used.

3.6.2. Syntax

FLAG [; TOTAL];
[flag-identifier =] ... operating system defined name of the flag;

3.6.3. Semantics

flag-identifiers are used like BINAL strings in the problem division, e. g. in conditional statements. Further information may be accessed via a standard procedure STATUS (flag-identifier) in this case, too.

4. TASK-Management

4.1. Problem

Contrary to a sequentially executed program for off-line calculation of results from a set of data, loaded once only, where the chronological order of operations is necessarily obtained from the program and the data – programs for on-line control and evaluation of data from industrial processes and experiments have an entirely different structure: They consist of many short sections which are, individually speaking, sequentially executed, but which are, regarding the entire process, executed quasi parallel. They are also interrupted by time intervals in which the computer does not have to compute for the process.

This structure is inevitable, because

(a) the computer must be able to react quickly in response to the many external demands which occur statistically in time,

(b) the urgency of the reactions is differentiated so that every reaction can be interrupted, replaced and continued later in favour of another reaction.

The task-management has to enable this quasi-parallel execution and to control its timing.

4.2. The "Task"

4.2.1. Definition

A user-program is executed in sections which are called "tasks". The base of such a task is a statement, or a sequence of statements combined to form a block. Jumps out of this block are not permitted.

The dynamic execution of this sequence of statements, under control of the operating system defines a task. A task can demand, from the operating system, the execution of additional self-contained sequences of statements, whereby new tasks ("subtasks") will be generated.

This generating process may be performed in several stages. Tasks are simultaneously executed, as far as the devices (computer and peripheral units) permit.

4.2.2. Priorities

If several tasks simultaneously demand the use of the same device or unit, the operating system will decide, according to their priority, to which of the competing tasks the unit will be allocated. The user assigns the priorities to the tasks. When not all of the devices required by a task are allocated to it, then the task is placed in the "waiting state". The priority of a task is a natural number. The lower the number, the higher the priority. The operating system selects the task which will be executed first, if two or more have the same priority.

4.2.3. Task-Generation

A task is generated by the statement

ACTIVATE task-name [WITH PRIORITY priority-number] : statement;

where taks-name is the identifier which addresses the task, priority-number the priority of the task and statement the code to be activated.

If WITH PRIORITY priority-number is missing, the new task is assigned the priority of the generating task;

task-name must be declared as TASK.

Activation of a task means to connect the task name with the code, to set the priority and to notify the operating system of the code to be executed. Tasks having the same name must not be activated at the same time.

4.2.4. Task Operations

Other operations which have an effect on a task are:

a) TERMINATE task-name;
b) DELAY task-name;
c) CONTINUE task-name [WITH PRIORITY priority-number];

TERMINATE ends a task and all its activated sub-tasks (non-interruptible operations which have already been started will be finished first).

DELAY transfers a task into the waiting state (but not its subtasks).

CONTINUE cancels this waiting state (i. e. re-activates the task).

Furthermore,

CONTINUE task-name WITH PRIORITY priority-number;

is used to change the priority of the addressed task.

If the addressed task does not exist, i. e. either it has not been activated, or it is already terminated, then the task operations will be interpreted as dummy operations.

In addition to this there is another standard function

PRIORITY (task-name),

which supplies the priority of the task task-name, as a result, if the task exists; otherwise the result will be a negative number.

4.2.5. Time Control of Tasks

In all task operations the time of their execution may be specified as follows:

[AT time 1] [EVERY time-interval [UNTIL time 2]] task-operation;

This type of statement results in the execution of the specified task-operation in intervals of time-interval from time 1 to time 2. If AT time 1 is left out, task-operation will be executed for the first time when this statement is executed. If EVERY time-interval UNTIL time 2 is left out, then a "once only" execution of task-operation follows. In the case that UNTIL time 2 behind EVERY time-interval is missing, then task-operation will be repeated, providing that the task still exists which contains the pertinent statement.

Further activations of a task, caused by a statement which has already been executed, such as EVERY time-interval ACTIVATE task-name; can, nevertheless, be hindered by using another task operation

PREVENT task-name;

PREVENT is ineffective if no further activations are queued for the addressed task.

4.2.6. Task End

A task ceases

a) when the task and all its subtasks have executed their last statement

b) if it is ended by TERMINATE.

If the end of a block is reached, during execution of a task, the execution will be continued only after all activated subtasks in the block are finished.

4.3. Alarms

4.3.1. Reaction to Alarms

Reactions to alarms are specified by the statement

ON alarm : statement;

where alarm is the identifier of a logical interrupt and statement is the code to be executed in response to the interrupt; statement will always be executed with the highest priority. This execution is non-interruptible.

4.3.2. Signal

A logical interrupt interrupt can be generated or simulated in the user program (e. g. for test runs) by the statement:

SIGNAL interrupt;

4.3.3. Disconnection

If several ON statements refer to the same interrupt, then the last one to be executed in the block is valid. When exiting a block, the last valid ON statement in the next outer block is effective. If it should occur that in none of the outer blocks an ON statement is executed, then there is no further response to this interrupt.

An interrupt, at a distance as far as possible from the central unit permitted by the implementation concerned, can be switched off by using

DISABLE alarm;

The interrupt can be switched on again by

ENABLE alarm;

4.4. Semaphore

4.4.1. Problem

Since the timing sequence of operations in two tasks running parallel with each other cannot, in many cases, be completely arbitrary, means must be provided whereby the desired sequence of such operations can be attained. For instance – before the output of an array by a task, this array has to be filled in by another task. If this sequence is accidentally reversed then the results are senseless. For the synchronization of tasks running parallel with each other, semaphore variables and non-interruptible semaphore operations which have an effect on the semaphore variables are used.

4.4.2. Features of the Semaphore Variables

Semaphore variables are initialized at the same time as their declarations. Further access to a semaphore variable is only possible via a semaphore operation.

4.4.3. Operations

Semaphore operations are non-interruptible. The following operations are available:

REQUEST sema
RELEASE sema

The operation REQUEST decrements the value of the semaphore variable sema by one, providing that the result is not negative. If the result were negative, the task which is trying to decrement the semaphore variable will be placed into the waiting state until the semaphore variable can be decremented to a non-negative result.

RELEASE increments the value of the semaphore variable sema by one and starts the tasks, in order of priority, which had previously been stopped by REQUEST operations.

The operators RELEASE and REQUEST may operate on lists of semaphore variables. Consequently,

REQUEST sema [, sema] ... ;

checks whether all semaphore variables in the list can be decremented to a non-negative value. If this is impossible, then none will be drecremented and the operation will be stopped until the possibility arises.

4.5. Working Storage Allocation

4.5.1. Principle

The entire code of a PEARL-program will, in many cases be located in backing storage and not in working storage. Consequently the code needed for a task has to be loaded, by the operating system, from the backing storage to the working storage. The operating system determines from the activation statement for this task, that additional code is required as well as the identification of this code. The activation statement must be executed early enough so that the loading procedure is finished in time. Untimely execution of the task can be prevented by using SEMA variables.

4.5.2. Code of a Task

The code required for a task consists of all instructions which may be performed and all data which will be formally used therein. In particular, all procedures which will be called in the task (possibly via several stages) belong to the code. The requirements of a sub-task, e. g. its instruction sequence, do not form part of the code of the task activating it.

As in RTL [4], procedures and data will not be loaded together with the block containing their declaration, but together with the instruction sequences in which they will be (formally) called or used.

4.5.3. Block Residence

It is possible for the PEARL-programmer to load procedures, i. e. their instructions and data, together with the code of a task not containing a call for them. This is attained by the statement

RESIDENT procedure-call [, procedure-call] ... ;

The effect as regards loading is the same as that of calls located in the same position. Hence RESIDENT applies only the block in question.

4.5.4. Displacement

For loading, the section of working storage which is not occupied with instructions or data of an existing task will be used first of all. If this space is no longer available, then the code of a sufficient number of tasks which are either in the waiting state or have a lower priority than the task to be newly activated, will have to be displaced from the working storage. The data to be displaced must not be overwritten and must, therefore, be previously transferred into the backing storage. If there are no tasks with lower priority than the one be newly activated, then it will be placed in the waiting state.

4.5.5. Reload

Generally speaking, tasks to be newly activated often require data and procedures which have already been called or used by tasks which chronologically precede them, and which are consequently still located in working storage. This raises the problem for the operating system of not only determining the additional code required, but also of linking this code to sections which are already in the working storage.

Whether this is possible, and how to achieve it, depends largely on the computer system in use and its operating system. Since working storage allocation is controlled by priorities, it may be necessary to restrict the priorities of sub-tasks in order to obtain effective implementation.

5. Input/Output

5.1. Basic Principles

If one attempts to handle all known I/0-facilities from a uniform point of view, even in commercial or scientific data-processing-applications difficulties arise; e. g. external storage devices with either random or sequential access should be addressed in the same manner. Apart from that there are functionally very different I/0-devices (compare a teletypewriter with a CRT) that shall be handled with procedures that are as similar to each other as possible.

The additional peripheral devices of a computer that is used for data acquisition and control of industrial processes and scientific experiments enlarge the difficulties of a standardized I/0-description. In this section it is tried to develop a syntactically uniform description of all I/0-facilities.

Primarily the uniformity of the description is achieved by the formulation of procedures for all I/0-functions. In the formal handling this corresponds to the possibilities that are provided in ALGOL 60 and 68.

In FORTRAN and PL/1 I/0-transfers are formulated as special statements, but also in these languages they are implemented as subroutines.

In PEARL this procedure concept yields another advantage: we have the possibility to easily activate I/0-operations in a sequential and in a parallel manner:

ACTIVATE: OUTPUT 1 (CA, RAM, BA); describes output done in parallel with the execution of the following statements.

The procedure call "OUTPUT 2 (TOM, DOOLEY);" without the prefixed ACTIVATE starts the I/0-procedure as a subroutine. This means: the running task executes the succeeding-statement after the termination of the I/0-operation.

The description of I/0 procedures in PEARL is characterized by 5 descriptors:

direction, mode, external-terminal, internal-terminal, option. Direction (IN or OUT only) and mode are combined to one word (the name of the procedure). The following three descriptors are parameters of this procedure.

external-terminal is an identifier for a device or a file. internal-terminal defines the area in working storage where data is to be in- or outputted. Syntactically it is represented by a (if necessary parenthesized) list of identifiers. option describes control information which is probably necessary for special modes.

Six mode-descriptors seem to be sufficient:

PRIMITIVE
BINARY
CHARACTER
GRAPHIC
CALIBRATED
CONTROL

Suitable abbreviations of these keywords will be defined later by the PEARL-committee.

5.2. Syntax, Semantics and Function of the I/0-Procedures

5.2.1. PRIMITIVE Input/Output

Syntax:

{ INPRIMITIVE / OUTPRIMITIVE } (external-terminal, internal-terminal [, control-information]);

control-information is a list of variable identifiers and/or constants containing control information for the data transfer.

Semantics:

The procedures INPRIMITIVE and OUTPRIMITIVE are used to transfer data from a terminal device to working storage and vice versa; information for datapath- and device-control must be provided explicitly by the programmer. This can be done by the control-information included in the same procedure call or by a previous output of control information only.

Examples:

```
1) INPRIMITIVE (DEVICE 1, DATA, '101101101111' B)
   /* TRANSFER FROM DEVICE 1 TO THE VARIABLE 'DATA' IS
   CONTROLLED BY THE BINAL STRING OF PARAMETER 3 */;

2) X := '00000100' B;
   AB := '0110' B;
   OUTPRIMITIVE (PROCON 3, X, AB)
   /* THE FOLLOWING INFORMATION IS TRANSFERRED TO
   PROCESSCONTROLUNIT 3 :
   AB IS INTERPRETED IN THAT WAY : 01 SELECT THE 2ND OF 4 STEP
                                                    MOTORS,
                                   10 TURN CLOCKWISE,
AND X : TRANSFER THE VALUE 4 TO THE MOTOR'S STEP COUNTER */;
```

Range of application:

PRIMITIVE Input/Output permits the possibility to handle machine code on language level. Primarily it is necessary if none or only rudimentary routines for handling an addressed peripheral and its datapaths exists (e. g. new built or modified hardware). All other I/0-functions can be expressed by suitable combinations of PRIMITIVE I/0.

5.2.2. BINARY Input/Output

Syntax:

{ INBINARY / OUTBINARY } (external-terminal, internal-terminal);

Semantics:

The procedure OUTBINARY transfers data from working storage to a terminal device while control of datapath and device functions is entirely performed by the operating system. The transferred information is not changed, in particular it is not translated into any other code.

In the same manner the procedure INBINARY has as its result an automatic transfer without data transformation.

Range of application:

The procedures for binary I/0 can e. g. be used for intermediate storage of data on magnetomotoric storage devices (similar to the use of "unformatted I/0" in FORTRAN). Reading from and writing onto process peripherals is also done with these procedures when there is a standard mode of transfer and no calibration of data is necessary.

Example:

```
INBINARY (TEMP (1 : 4), KELVIN (5 : 8))
/* INPUT FROM THE FOUR THERMOCOUPLES 'TEMP (1)' TO 'TEMP (4)'
IS TRANSFERRED UNCHANGED FROM THE ADC TO THE ELEMENTS 5
TO 8 OF ARRAY 'KELVIN' */;
```

5.2.3. Input and Output of Characters

Syntax:

{ INCHARACTER / OUTCHARACTER } (external-terminal, internal-terminal,
{FORMAT (string), PICTURE (string), SPECIALLAYOUT (...)});

The third parameter of the CHARACTER-procedures is a format-describing procedure, the parameter of which is a character string.

Semantics:

Together with the data transfer a translation (code-transformation) is performed character by character between the internal machine representation of the data (which is defined by the special machine and by the data type) and the external representation (defined by the device, and occasionally by the programmer, included in the format description). Moreover the layout of the data on the external medium is described by the FORMAT-(or PICTURE-) procedure.

It shall be possible to specify special layout procedures instead of the standard ones (e. g. procedures that include interpretation of commands entered on a manual input device).

The external-terminal is regarded as a book as in ALGOL 68 [13]; (also refer to the file description in 2.2.1.5.). It is suggested that the parameter-string of the FORMAT-procedure should be identical or very similar to that of PL/1 [8] (Probably including a code-definition facility). This features should be supplemented by a new-defined PICTURE-procedure which corresponds to a layout-description which is very simple to use.

Yet, it is discussable that, instead of the aforegoing points, the complete concept of any other programming language, which has advanced text editing features, may be utilized.

5.2.3.1. The PICTURE-procedure

The single parameter of PICTURE is represented by a character-string that depicts the layout on the external medium (e. g. page of a line printer) both in length and structure. String constants are inserted into the string as they shall be outputted, fields that are to be filled with variables are reserved by space imprinted with the wanted layout structure.

Example:

The variables I (integer, value 23), B (real array with two elements, values 17.1 and $3.1*10^{12}$) and TXT (string, value 'PEARL') shall appear on an external device like teletype-writer or CRT, connected to "RESULT-LIST", showing this layout:

```
RESULT
TEST 23                          B1 = 17.1        B2 = 3.1E12
----------THIS WAS A PEARL-PROGRAM----------
```

The output procedure is called in this way:

```
OUTCHARACTER (RESULTLIST, (I, B, TXT), PICTURE
('RESULT
 TEST # #                        B1 =## $# B2 = # $ # $ # #
----------THIS WAS A eeeee -PROGRAM----------'));
```

The variables I, B, and TXT are inserted into the fields structured with the symbols ≠, $ and e. Their special meaning is:

≠ : replace with a number
$: replace with a special symbol (also E in exponent notation)
e : replace with an alphanumerical symbol

If a string-constant contains these symbols, they must be enclosed in double quotes. Perhaps further special characters will have to be used for picture specifications (e. g. for suppression of leading zeroes).

The purpose of this modification of conventional PICTURE-procedures is to enable the user to describe the exact layout of the text without any shifting caused by delimiters.

5.2.3.2. Range of application

CHARACTER I/0 shall be used everywhere where character strings (text or numbers) are in-/outputted. The I/0-devices will in most cases be dedicated to man-machine communication. This method may also be used to handle process I/0 if the peripherals use alphanumeric coded data representation.

Example:

```
INCHARACTER (ADC 1, VAR, PICTURE ('####␣␣'))
/* THE FIRST FOUR OF THE SIX DECIMAL POSITIONS OF ADC 1
SHALL BE DECODED AND STORED INTO VAR */;
```

CHARACTER I/0 is also useful for tape- and disk-like devices if they are e. g. used as buffer storage for slow peripherals.

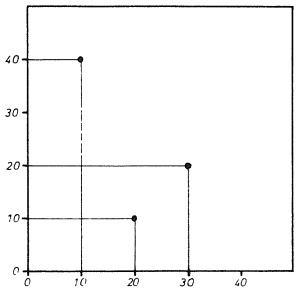


Fig. 3

Example of a RANDOM point display

5.2.4. Graphic Input/Output

By means of this procedure all I/0 related to graphs and pictures can be described. It handles the typical graphic devices like plotters or CRT-displays, but also the "drawing" of a curve by means of a typewriter with points represented by some letter (e. g. 'x') will be done using the GRAPHIC mode.

2)
```
SPECTR := (40.0, 35.0, 25.0, 15.0, 13.0, 17.5, 25.0,
          31.5, 37.0, 39.0, 40.0, 39.5, 37.5, 33.0,
          22.5, 10.0, 5.5, 3.0, 2.0);
OUTGRAPHIC (DISPLAY, SPECTR, INCREMENT (2.5));
```

5.2.4.1. The output Procedure OUTGRAPHIC

Syntax:

OUTGRAPHIC (external-terminal, internal-terminal, layout-procedure (par, ...));

The user of PEARL will be supplied with a number of standard layout-procedures for graphic editing. He may as well write his own routines for that purpose.

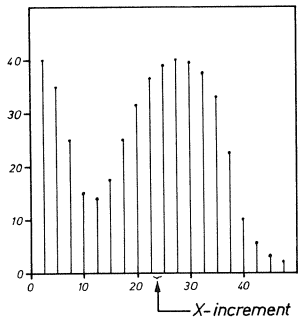


Fig. 4

Example of an INCREMENTal display

Standard layout-procedures:

no.:	internal data:	procedure name:	parameters:	graphic picture:
1)	$\vec{x}$, $\vec{y}$ [, $\overrightarrow{intens}$.]	RANDOM	intensity, scale	see example 1)
2)	$\vec{y}$ [, $\overrightarrow{intens}$.]	INCREMENT	intensity, y-scale, x-increment	see example 2)
3)	$\vec{x}$, $\vec{y}$, $\overrightarrow{intens}$.	LINE	scale	see example 3)
4)	$\vec{x}$, $\vec{y}$ [or $\vec{y}$]	INTERPOL	intensity scale, mode of interpolatn.	see figure 6
5)	$\vec{x}$, $\vec{y}$, $\vec{r}$, $\vec{\varphi}_1$, $\vec{\varphi}_2$	CIRCLE	scale, intensity	

3)
```
STAR :=  ((10,10, '0'B),
          (20,20, '1'B),
          (30,10, '1'B),
          (20,30, '0'B),
          (20,20, '1'B));
OUTGRAPHIC (DISPLAY,
STAR, LINE);
```

The basic method of graphic layout is the RANDOM point plot (1), points on the image correspond to pairs of values (coordinates) in a data array. The image may be modified by specification of scale factors and of the brightness of the points (if possible). With INCREMENT point plot the facility of many graphic devices to generate x-coordinates automatically (1, 2, ... n) is described (2). The programmer must provide the y-data only.

With the LINE ("vector") mode two consecutive points are connected with a (bright or dark) straight line. Only the specification of the brightness and one terminal point is needed as the drawing beam starts at the previously reached position.

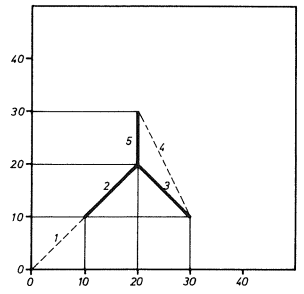


Fig. 5

Example of a vector (LINE) mode display

Special layout functions for INTERPOLation (4) and for the drawing of arcs of CIRCLEs (5) may be useful.

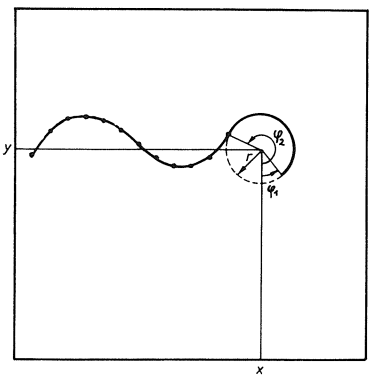


Fig. 6

Example of an INTERPOLated curve and an arc of a CIRCLE

Examples:

1)
```
X(1) := 10; X(2) := 20; X(3) := 30; Y(1) := 40; Y(2) := 10; Y(3) := 20;
OUTGRAPHIC (DISPLAY, (X, Y), RANDOM);
```

5.2.4.2. The Input Procedure INGRAPHIC

Syntax:

INGRAPHIC (external-terminal, internal-terminal,
function (par,...));

Semantics:

external-terminal is a graphic input device like a lightpen or a joystick that has been described in the system-division of the program. Usually internal-terminal is a pair of variables (x, y), and there is no 3rd parameter. The input device supplies coordinates which are stored in x and y. INGRAPHIC allows the identification of certain points on the image on a graphic output device. According to that identification further actions can be taken in the program.

5.2.5. Calibrated Input/Output

These procedures are useful for transferring analog information supplied by a process input device; during output the user is able to specify an angle e. g. in units of degres instead of a number of steps a motor has to turn.

5.2.5.1. The Input Procedure INCALIBRATED

Syntax

INCALIBRATED (external-terminal, internal-terminal,
calibration-function (par,...));

Semantics:

The external device supplies one or more analog test values that are stored in working storage in a digitalized and calibrated form (transformed into physical units). This is accomplished by a calibration-function that is part of the system library, the parameters of which can be supplied by the programmer. E. g. one can use a polynomial to calibrate the input values delivered by a thermocouple. When the element is replaced this can be taken into account merely by changing the coefficients of the polynomial (= parameters of the calibration-function).

5.2.5.2. Output using OUTCALIBRATED

Syntax:

OUTCALIBRATED (external-terminal, internal-terminal,
calibration-function (par,...));

Semantics:

internal-terminal contains variables that specify the amount in physical units by which the position of a terminal device (e. g. a valve using servo motors) has to be changed.

Example:

```
OUTCALIBRATED (MOTOR1, FLOW, CALIB1 (A1, A2, A3))
/* THE FLOW THROUGH A PIPELINE IS REDUCED TO A CERTAIN VALUE;
THE RELATION BETWEEN THE POSITION OF THE VALVE AND THE VALUE
OF THE FLOW IS NONLINEAR AND DESCRIBED BY FUNCTION 'CALIB1' */;
```

Comment: The calibration-procedures must be predefined by the programming system. A linear calibration function, a polynomial and a nonlinear calibration using several lists seem to be sufficient for the most applications.

5.2.6. Input and Output of Control Information

Syntax:

{OUTCONTROL / INCONTROL}(external-terminal, internal-terminal);

(In the case of OUTCONTROL the internal-terminal may also be a constant – generally BINAL)

Semantics:

INCONTROL stores the status information (as far as it is relevant to the system) of the addressed peripheral into the named variables. Using OUTCONTROL control commands can be sent to peripheral devices.

Range of application:

INCONTROL mainly will be used to check functions of process peripherals; this procedure is a means of machine-oriented programming. In a similar manner OUTCONTROL is used if one wants to deal with details of the I/0-hardware (as it can be done using assembler language).

The features of CONTROL I/0 are similar to that of PRIMITIVE I/0; but here only control information – no data – is exchanged.

Example:

```
ON INTERRUPT SENSOR1 BEGIN
        INCONTROL (SENSOR1, X);
        IF X = '00110010'B THEN GO TO ALARM FI
        END;
```

5.3. Auxiliary Procedures for Input/Output

To accomplish complex control functions in the I/0-management of the system a number of standard procedures are available.

These are procedures for OPEN-ing and CLOS-ing I/0 which can also act on files, for reserving a device for exclusive use (LOCK) and for cancelling the reservation (UNLOCK), for RESET-ting a device and for SET-ting it to a specified state and for STOP-ping I/0 abruptedly. Functions to CREATE, to REIDENTify and to SCRATCH (i. e. to cancel the reservation) files are also necessary. There must be also a number of formatcontrolling procedures (to be used as subparameters of FORMAT or layout-procedures or as stand-alone calls), e. g. BACKSPACE, SKIP, NEWLINE, and NEWPAGE.

6. Acknowledgements

This proposal of a process- and experiment-oriented programming language was composed by a working group which representatives of the following firms and institutes belong to:

1. Hahn-Meitner-Institut für Kernphysik, Berlin
2. Physikalisches Institut III der Universität Erlangen
3. Strahlenzentrum der Universität Gießen
4. Zentralinstitut für angewandte Mathematik der Kernforschungsanlage Jülich
5. Gesellschaft für Kernforschung Karlsruhe
6. Siemens AG Karlsruhe
7. AEG-Telefunken Konstanz
8. BBC Mannheim
9. Physikalisches Institut III der Universität Marburg
10. Math. Institut der Technischen Hochschule München

Here we want to thank you who have made this work possible by sending delegates; especially also Mr. Heller (BASF, Ludwigshafen), the chairman of the VDI subcommittee "Programmiertechnik" who has maintained the connection to our working group. We also want to thank Prof. Dr. N. Fiebiger from the Physics Institute III of the University of Erlangen for his interest and helpful support, the Bundesministerium für Bildung und Wissenschaft and the "Studiengruppe Nuklearelektronik" for financial support. Last but not least Mr. Kessel and his co-workers from the Institut für Kernphysik der Universität Frankfurt have helped us very much by providing for rooms and organizing the sessions.

Beside the authors the following participiants have contributed to this paper:

M. Degenhardt (HMI Berlin), H. Homrighausen (KfA Jülich),

G. Koch (BBC Mannheim),

K. Kreuter (Siemens AG Karlsruhe),

W. Rüb (TH München),

W. Schöfer (AEG-Telefunken Konstanz),

D. Wiegandt (Univ. Marburg, now at Cern Genf),

K. Wölcken (Universität Gießen).

If you want copies, or if you have any comments or questions, please write to Mr. P. Elzer, Tandemlabor, Physikalisches Institut, University D-852 Erlangen, Erwin-Rommel-Str. 1.

7. Literature

[1] Opler: Requirement for Real-Time-Languages; Comm. ACM 9 (1966), 196

[2] Zemanek: Rolle und Bedeutung formeller Sprachen; E u. M 83 (1966), 463

[3] Frost: Fortran for Process Control; Instrumentation Technology 16 (1969), 4

[4] "RTL" A language for Real-time-systems; The Computer Bulletin, Dez. 1967, S. 202-212

[5] Boulton, Reid, Pierce: A process control language; IEEE Transactions on computers C-18 (1969), 1049

[6] INDAC-8 software: Digital Equipment Corporation; Maynard, Mass.

[7] G. Müller: Die Verwendung einer problemorientierten Sprache für Prozeßrechner, Aufbau und Funktionsweise des zugehörigen Compilers; Vortrag auf dem "Jahreskolloquium zur Rechentechnik", Febr. 1970, TU Braunschweig

[8] PL/1-language-specifications, IBM C28/65-71

[9] Dijkstra: Cooperating Sequential Processes in: Genuys (Editor); Programming languages, London 1968

[10] Aus: PL/1 – Sprachspezifikationen: IBM Form 79879-1

[11] Wijngarden, Mailloux, Peck, Koster: Report on the Algorithmic Language ALGOL 68; Numer. Math. 14 (1969), 79-218

[12] Beech et al.: Concrete Syntax of PL/1; IBM United-Kingdom-Lab. TN 3001

[13] Naur et al.: Revised Report on the Algorithmic Language ALGOL 60; Num. Math. 4 (1963), 420-453

[14] V. Haase: EXOS – Entwurf einer experimentorientierten Programmiersprache; GfK-Karlsruhe, externer Bericht, August 1967

[15] P. Elzer: Möglichkeiten zur Entwicklung einer Programmiersprache für kernphysikalische Experimente; Arbeitspapier Phys. Inst. Erlg.; Juli 1968

[16] Berger, Seibt, Strunz: Bibliographie... zum Thema Programmiersprachen, elektron. datenverarb. 11 (1969) Heft 5 und 7

[17] IEEE-Transactions: Industrial & electronics & control instrumentation 15 (1968) No 2 (Sonderheft über Prozeßkontrollsprachen)

[18] Workshop on Standardization of Industrial Computer Languages (Minutes, part 1) Purdue University, Lafayette, Indiana, Febr. 1969

[19] Preliminary Glossary, workshop on standardization of Industrial Comp. Lang. (Minutes, part 2) Purdue University, Oct. 1969

[20] Workshop on Standardization... (Minutes, part 3) Purdue University, March 1970

[21] IFIP Fachwörterbuch der Informationsverarbeitung: Amsterdam 1968