KfK-PDV 141
August 1977

PDV-Berichte

PEARL
Survey of Language Features

Axel Kappatsch
IDAS GmbH, Limburg

Kernforschungszentrum Karlsruhe

Druck und Verbreitung:

Kernforschungszentrum Karlsruhe GmbH
Postfach 3640 7500 Karlsruhe 1

Bundesrepublik Deutschland

P E A R L

PROCESS AND EXPERIMENT AUTOMATION

REAL-TIME LANGUAGE

Survey of Language Features

AXEL KAPPATSCH

IDAS GMBH, LIMBURG

AUGUST 1977

I. INTRODUCTION

The high-level language PEARL (**P**rocess and **E**xperiment **A**utomation **R**eal-time **L**anguage) has been developped for the purposes of real-time programming. It has been designed to permit an efficient control of processes of all kinds, for example

. communication processes in information systems
. experiment control in scientific research
. control of industrial production etc.

The special requirements imposed on a language for process control essentially necessitate

. administration of concurrent activities
  and
. adaptability to a variety of peripherals.

Thus, the main emphasis in the development of PEARL has been upon input/output and real-time features.

The following discussion of language features is intended as an informal survey of Full PEARL, details and definitions are to be found in the defining document [1]. The standard-subset 'Basic PEARL' is described in [2]. Historical background information concerning the PEARL development can be found in [3].

II. PROGRAM STRUCTURE

1. MODULES

PEARL-programs are composed of *independently compilable units*, the so called '**MODULE***s*'. Connections between **MODULE**s are established through **GLOBAL** objects (fig. 1).

In general, a **MODULE** consists of a part describing the environment of the system (**SYSTEM**-*division*) and another one containing the formulation of the problem (**PROBLEM**-*division*).

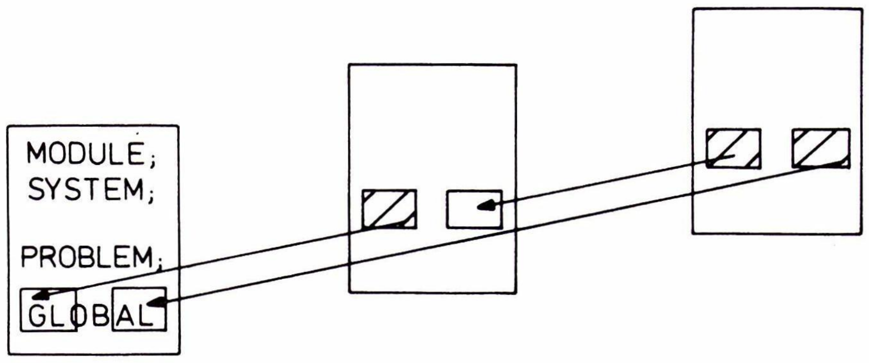


Fig. 1: MODULEs and their communication via GLOBALs (▭); arrows symbolize "import/export" relations, hatching marks the storage location

2. SYSTEM-DIVISION

The **SYSTEM**-division of a **MODULE** displays the peripheral environment of a PEARL-program or a part of it. It permits, in particular, to attach user-identifications to devices and to list their connections (fig. 2):

```
SYSTEM ;
.
.
.
CHANNEL*6    ->    MONITOR:MO100 ;
CHANNEL*4    ->    PRINTER: ;
CHANNEL*3  < ->    PROCESSCHANNEL ;

                   PROCESSCHANNEL*0   ->   DIGOUT ;

                                  DIGOUT*3   ->   RELAY: ;
                                  DIGOUT*5   ->   VALVE: ;

                   PROCESSCHANNEL*1   <-   ANALOGIN ;

                                  ANALOGIN*2  <-TEMPERATURE:;
```

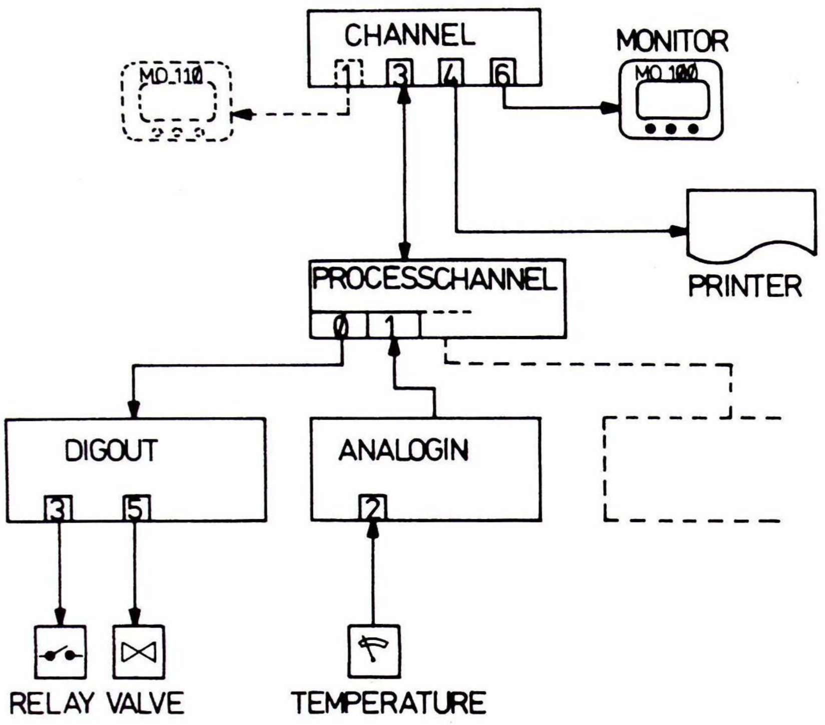


Fig. 2: Peripheral configuration described by a SYSTEM-division

The **SYSTEM**-division partly serves as the function of a Job Control Language by separating program-modifications required by a different peripheral configuration from the application part (**PROBLEM**-division). As an example, replacing the monitor **MO100** in figure 2 by a device of type **MO110** connected to a different channel would require substitution of the corresponding **SYSTEM**-division statement by

CHANNEL*1 –> MONITOR : MO110 ;

3. PROBLEM-DIVISION

The **PROBLEM**-division is composed of a set of declarations and specifications, in particular of those program parts which may be executed concurrently, so-called '**TASK***s*'. Tasks may 'activate' each other, i.e. a task may schedule the executions of another task (or its own execution). The first task of a PEARL-program is activated by external action, i.e., not at language level.

III. ALGORITHMIC FEATURES

The algorithmic language concepts of PEARL are based on ALGOL 68 and - as far as possible - have been adapted syntactically to PL/I.

1. DECLARATIONS AND SPECIFICATIONS

User-introduced language objects have to be *declared* prior to their use in operations:

DCL (X,Y,Z) FLOAT ;

A declaration can either be executed when executing a task, a procedure, an interface or an operator or not. In the latter case, the declaration is said to stand 'on module-level' and the declared objects are known within this module for the whole time of program execution. Whenever an object is to be 'exported', i.e. to be made known in other modules, the declaration must be completed by the clause '**GLOBAL**':

DCL X FLOAT GLOBAL ;

Where the properties of an object declared in one module have to be made known ('imported') in another one, this object needs a *specification*, e.g.:

SPC X FLOAT GLOBAL ;

Besides specifications of **GLOBAL** objects, those of formal parameters in procedures play an important role.

2. BASIC DATA TYPES

Six basic data types represent

. *numbers*:

FIXED	for integers	: **07**
FLOAT	for floating-point numbers	: **7.1**

. *strings*:

CHARACTER	for character-strings	: **'PEARL'**
BIT	for bit-strings	: **'0101'B**

times:

CLOCK	for 'points in time'	: **11:55:04**
DURATION	for time-intervals	: **3 HRS 4 MIN 3 SEC**

For numbers, a precision may be supplied, for strings a length. Precisions and lengths enter significantly when operations with basic data types are considered.

3. LABELS

PEARL has label-constants as well as label-variables:

```
        DCL  TAG  LABEL ;
        .
        .
        .
   M1 :  A := B + C ;
        .
        .
        .
        TAG := M1 ;
```

A label-variable may be restricted to take on only a limited set of values which may be listed in a **RANGE**-*clause*, for example:

```
    DCL  TAG  LABEL  RANGE (M1,M2) ;
```

In this case, only the values of **M1** or **M2** could be assigned to the variable **TAG**.

4. ASSIGNATION PROTECTION

Assignations to PEARL-objects can be *inhibited* either completely or only when accessing these objects using certain identifiers. This is achieved adding the keyword '**INV**' in a declaration

```
    DCL  PI  INV  FLOAT  INITIAL (3.14) ;
```

or a specification

```
    SPC  MOMENT  INV  CLOCK  IDENTICAL (TIME) ;
```

Thus, **PI** and **MOMENT** can only be read.

Assignation protection is an important tool in structured programming, for example, when data may be modified in only one **MODULE** but read in others:

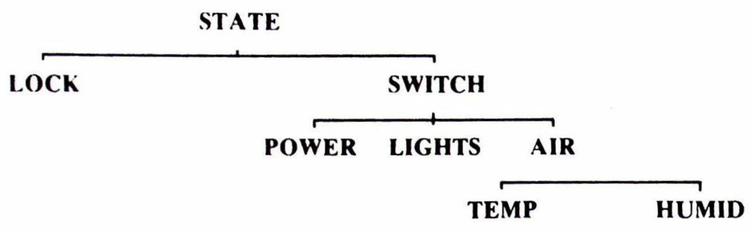
```
MODULE ;
PROBLEM ;
.
.
.
DCL SWITCH FIXED GLOBAL ;
.
.
.
MODEND ;

MODULE ;
PROBLEM ;
.
.
.
SPC SWITCH INV FIXED GLOBAL ;
.
.
.
MODEND ;
```

5. COMPOUND OBJECTS

Basic types and most of the other PEARL-objects may be assembled to form *arrays* and *structures.*

Structures may in particular contain substructures which can be handled independently, building up data hierarchies of great complexity.

```
DCL STATE STRUCT
          [    LOCK SEMA, SWITCH STRUCT
                                   [    POWER    BIT (1),
                                        LIGHTS   BIT (1),
                                        AIR   STRUCT
                                             [ TEMP    FLOAT,
                                               HUMID   FIXED
                                             ]
                                   ]
          ] ;
```



6. POINTERS

Pointers referring to PEARL-objects are *type-specific*:

```
DCL MATREF  REF (,) FLOAT ;
```

The pointer **MATREF** may refer to two-dimensional matrices of floating-point numbers.

Not every PEARL-object may be referred to by a pointer; pointers themselves are excluded, for example, as well as tasks or procedures.

7. PROCEDURES

Declaring a PEARL-procedure requires a precise definition of how parameters and results are transmitted to and from the procedure respectively.

For each parameter it has to be specified, whether the procedure needs its value at the point of invocation or its 'identity' (i.e. its address).

In the first case (**INITIAL**-*mechanism*), a copy of the actual parameter will be made. In the second case (**IDENTICAL**-*mechanism*), a new identifier, whose scope is the procedure-block, is attached to the parameter. Combining these mechanisms with selective assignation protection provides a great variety of ways to access parameters. In

```
P:    PROC   (  ARRAY (,)    FLOAT   IDENTICAL,
                TIME INV     CLOCK   IDENTICAL,
                I            FIXED
             ) ;
        .
        .
        .
      END ;
```

ARRAY is a two-dimensional **FLOAT**-matrix and **TIME** is an object of type **CLOCK**, both not copied upon a procedure-call (**IDENTICAL**). In addition, '**TIME**' is protected against assignation which means that it cannot be modified within the procedure (but in a concurrent activity for example). The integer '**I**' will be copied upon **CALL P(...)** (**INITIAL** is default), i.e., the procedure uses its value at the point of invocation.

Functions, i.e., procedures returning a result, are characterized by the attribute

```
    RETURNS
```

followed by the type of the object returned:

```
     SINE :   PROC (X  FLOAT)  RETURNS (FLOAT) ;
              .
              .
              .
              END ;
```

Further attributes qualify the procedure-code as 'reentrant' (**REENT**) or to be 'directly inserted' (**INLINE**).

8. DEFINITION OF NEW DATA TYPES

A PEARL-programmer may define identifiers for *new data types* as given, for example, by structures:

```
    TYPE   COMPLEX   STRUCT [RE   FLOAT,  IM   FLOAT ]  ;
    .
    .
    .
    DCL   C   COMPLEX  ;
```

In addition, a type-identifier may stand for a set of types, for example

```
    TYPE   TRANSFER   ONEOF ( COMPLEX,  CHARACTER  (16)  );
```

, where '**TRANSFER**' is defined to be an abbreviation for the type **COMPLEX** *or* **CHARACTER(16)**. Used as transfer-item type of a data-station, for example, the latter would thereby be restricted to accept (or deliver) only the data of the respective types.

9. DEFINITION OF NEW OPERATIONS

User-defined data types are complemented by the definition of operators specially designed to handle such data types.

```
OPERATOR    +  (  A INV COMPLEX, B INV COMPLEX)
                  RETURNS (COMPLEX) ;

RETURN ([ A.RE + B.RE , A.IM + B.IM ]) ;
END ;
```

An operator-declaration introduces an identifier *or a symbol* for a monadic or dyadic operation defined like a function procedure. In fact, operators are similiar to generic functions, i.e. operators having the same identification are discriminated through their argument-types.

IV. INPUT AND OUTPUT

To cope with the variety of peripherals to be expected in a process environment, PEARL provides a particularly flexible I/O-system. It consists of a *network of data-ways* composed of the following components:

. *Data-stations*
generalizing real or virtual peripherals or *I/O-channels*

. *Interfaces*
mapping data-stations with different properties onto each other. Interfaces offer, in particular, the possibility to define formatting routines.

A schematic view of a set of user-defined I/O-channels mapped onto a set of system-defined channels by an interface is given in figure 3.

1. DATA-STATIONS

In PEARL, any I/O-device (standard- as well as process-peripheral) is considered as set of one to four channels:

. A *data-channel* acts as sink or source of values of 'transferable' PEARL-objects.

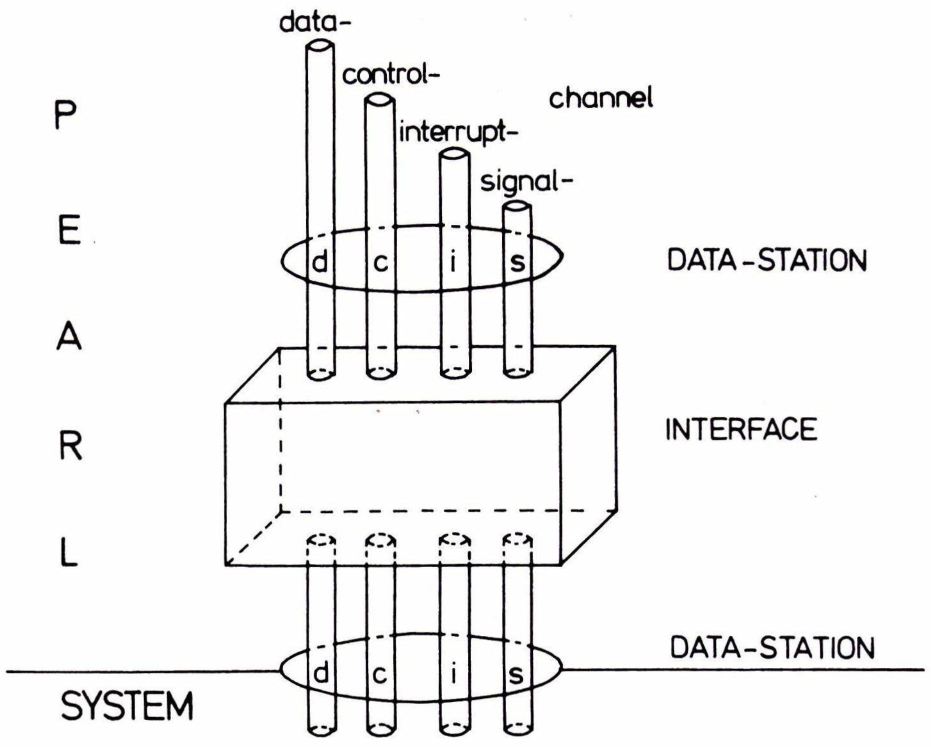


Fig. 3: Data-stations and interfaces

Not every PEARL-value can be transferred in a datachannel. This is limited essentially to objects of the basic types and the corresponding compound objects. In addition, an assignation protection (**INV**-attribute) inhibits input.

. A *control-channel* transmits information used to control a data-station in operation. According to its special importance, in **PEARL** such informations are handled through a separate data type: the c-channel transfers values of type **CONTROL**.

. An *interrupt-channel* signals events of type **INTERRUPT** (refer to section V).

. A *signal-channel* signals events of type '**SIGNAL**' (refer to section V).

In order to check the correct handling of peripherals, their properties have to be specified at language level. This is the purpose of a data-station specification:

SPC PRINTER DATION OUT ALPHIC CONTROL (ALL) ;

Such a specification contains information concerning

. the *direction of data flow* with respect to the central processor or main memory (**IN.OUT.INOUT**),

. the possibility of *direct transfer* with other peripherals and its direction (**SOURCE, SINK, SOUSI**),

. the *representation of the data* transferred: *internal* representation or formatting to *external* objects, e.g. printable characters (**ALPHIC**), logic levels (**BASIC**), graphic symbols (**GRAPHIC**),

. the *units of transfer*, e.g. one line or the content of a display,

. the *structure of the data* transferred, e.g. file structures,

. the *access-method* (**FORWARD, FORBACK, DIRECT**),

. the *continuation-method* (**STREAM, CYCLIC**).

The attributes attached to a data-station in its specification or declaration allow for a variety of checks permitting to detect some unsafe use of I/O-instructions as early as compile time.

2. INTERFACES

Starting from the set of devices provided by an implementation (more precisely, their drivers), virtual data-stations may be constructed, having more adequate, problem-oriented properties. The conversions required may be formulated in a routine called *interface*. Interface-routines have some resemblance as well to procedures as to tasks but are clearly distinct from both. They are inserted 'between' two data-stations (fig. 3) and - for example - allow for

. triggering of or reaction on interrupts and signals
. selection of subchannels
. opening and closing files
. adaptation of transfer-units (buffering)
. insertion or deletion of control-information into or out of the stream of data, respectively.

3. I/O-INSTRUCTIONS

Prior to the *use* of I/O-channels for transfer, the data-station/interface network has to be *set up* and its access by different tasks must be *synchronized.* Thus, there are three categories of I/O-instructions:

- constructing and dismantling data-ways
  The **CREATE**-*statement* links data-stations and interfaces creating an alternating sequence as, for example, the one displayed in figure 3:

  ```
  CREATE  OUTFILE  UPON  PRINTER  USING  PRINTERFACE ;
  ```

  Correspondingly, the **DELETE**-*statement* dismantles such links:

  ```
  DELETE OUTFILE ;
  ```

- synchronizing data-ways
  A data-way, as constructed by a **CREATE**-statement, may either be requested exclusively by a task or shared by several tasks. The corresponding clauses are part of the **OPEN**-*statement*, e.g.

  ```
  OPEN  OUTFILE  BY  EXCLUSIVE ;
  ```

  Accordingly, a task renouncing the use of a data-way issues a **CLOSE**-*statement.*

- using data-ways
  Subsequent to construction and synchronization, data-ways may be used for transfer. PEARL provides different kinds of *transfer-statements* according to the type of data-station and the direction of the transfer. A graphic output device, for example, is addressed using the **DRAW**-*statement* and the above specified **PRINTER**, e.g., by

  ```
  PUT  'ABC'  TO  PRINTER  BY  SKIP,  A (3) ;
  ```

  where the two controls **SKIP** and **A(3)** are transferred to the control-channel whereas the character-string **'ABC'** is put to the data-channel of data-station **PRINTER**.

V. REAL-TIME FEATURES

A real-time programming language as compared to languages for commercial or scientific-technical applications is characterized by facilities permitting control of concurrent execution of program parts.

PEARL provides a set of features to handle concurrency and to react upon interrupts and exception conditions.

1. TASKS

A *task* is similar to a procedure in that a piece of code is attached to it:

```
FILLING :   TASK ;
            .
            .
            .
            END ;
```

The *activation* of a task is analogous to a procedure-call:

```
ACTIVATE  FILLING ;
```

However, the executions of different tasks are uncorrelated in time.

Competing for resources, the *priority* of tasks is taken into account. A priority may be attached to a task at different occasions, for example in a declaration

```
REPORT :  TASK PRIORITY 25 ;
          .
          .
          .
          END ;
```

The execution of a task may be

. *suspended indefinitely*:

```
SUSPEND  FILLING ;
```

- *continued* subsequent to a suspension:

  CONTINUE FILLING ;

- *suspended for a definite amount of time* or until the occurrence of an interrupt:

  AFTER 10 MIN RESUME REPORT ;

- *terminated*:

  TERMINATE FILLING ;

- In addition, all *schedules* for a task may be *cancelled*:

  PREVENT REPORT ;

Such *schedules* may be given for all of the task-operations listed above. They express additonal conditions that have to be satisfied prior to execution of the operations, namely

- times, durations or both:

  AT 9 : 0 : 0 ALL 2 HRS UNTIL 18 : 0 : 0 ACTIVATE REPORT ;

- occurrences of interrupts:

  WHEN EMPTY CONTINUE FILLING ;

2. SYNCHRONIZATION

Whenever operations of two tasks have to be correlated - for example because they operate on common data or other shared resources - synchronization algorithms have to ensure the proper order of execution. This is facilitated using the synchronization primitives **SEMA***phore* and **BOLT**.

- The semaphore operations **REQUEST** and **RELEASE** may be used to synchronize a producer/consumer-type relation between tasks or, as a special case, an exclusive access to a resource:

  DCL CLEARANCE SEMA ;
  .
  .
  .
  REQUEST CLEARANCE ;

The latter statement is unsuccessful, i.e. leads to suspension, if another task has successfully requested the semaphore and has not yet released it through

```
RELEASE CLEARANCE ;
```

BOLT-operations provide a convenient formulation for synchronisations requiring a distinction between exclusive ('writing') and shared ('reading') access to a resource:

```
DCL FILE BOLT ;
.
.
.
RESERVE FILE ;
```

This demands *exclusive access* to the resource protected by the bolt **FILE** and leads to suspension whenever other tasks are using it. However no further access will be granted to tasks (with lower priority) demanding, for example, *shared access* by

```
ENTER FILE ;
```

As soon as the last 'sharing' task has renounced to use **FILE** through

```
LEAVE FILE ;
```

the **RESERVE** becomes effective.

The end of an exclusive use is indicated by

```
FREE FILE ;
```

subsequent to which new **ENTER**- or **RESERVE**-requests can be honoured (in their order of priority).

3. EVENTS

In PEARL two types of events may be defined and reacted upon:

. **INTERRUPT**s are *asynchronous events.* In general, interrupts are defined in the **SYSTEM**-division as arising from a (virtual) device and may be reacted upon by scheduling a task-operation, e.g. an activation:

```
SPC EMPTY INTERRUPT ;
.
.
.
WHEN EMPTY ACTIVATE FILLING ;
```

- **SIGNAL***s* like interrupts are in general **SYSTEM**-defined. They are used to handle *exceptional conditions* arising from execution of instructions (for example, overflow or end-of-file). A signal-event is notified exclusively to the task executing the erroneous instruction, which may either
  - ignore it
  - leave it to the system to react adequately
  - execute a specific signal-reaction, a so-called **ON**-block:

```
SPC EOF SIGNAL ;
.
.
.
ON EOF :      BEGIN ;
              .
              .  /* REACTION ON END-OF-FILE */
              .
              END ;
```

Interrupts as well as signals may be stimulated at language level:

```
TRIGGER EMPTY ;

INDUCE EOF ;
```

This is of particular importance for test and simulation.

REFERENCES

1. Full PEARL Language Description
Gesellschaft fuer Kernforschung mbH, Karlsruhe,
PDV-Bericht KFK-PDV 130,1977

2. Basic PEARL Language Description
Gesellschaft fuer Kernforschung mbH, Karlsruhe,
PDV-Bericht KFK-PDV 120, 1977

3. MARTIN, T.: The Develop ment of PEARL within the
Process Computer Control Project of the FRG
Gesellschaft fuer Kernforschung mbH, Karlsruhe,
PDV-Bericht KFK-PDV 129, 1977