A REALTIME PROGRAMMING LANGUAGE AND ITS APPLICATION FOR MEASURING PROCESSES

J. Brandes, TU, Karlsruhe
B. Eichenauer, ESG, Muenchen
S. Eichentopf, AEG-Telef., Konstanz
P. Elzer, Univ., Erlangen
L. Frevert, HMI, Berlin
V. Haase, Unicomp, Blankenloch
P. Holleczek, Univ., Erlangen
K. Kreuter, Siemens, Karlsruhe
B. Krueger, BBC, Mannheim
G. Mueller, BBC, Mannheim
K. Pfeiffer, KfA, Juelich
P. Rieder, Siemens, Karlsruhe
B. Schuerlein, Univ., Heidelberg
K.H. Timmesfeld, Entw.Buero Werum, Erbstorf

ABSTRACT

A group of German research institutes and industrial firms has worked out a concept of a programming language for process control and experiment-automation purposes. It is on procedural language level (like e.g. PL/1) and is designed to be used by the process control engineer or physicist with not very much knowledge of programming techniques. A first report has been published in "elektronische datenverarbeitung", 10, 1970, pp. 429 ff. In the paper will be given more details about the exact definition of the language, some experiences gained in a test implementation and, after all, examples for its use in measuring processes, especially one- and two-dimensional pulse-height analysis in Nuclear Physics.

INTRODUCTION

This paper deals with a proposal for a realtime programming language and shows an example for its application in Nuclear Physics data evaluation. The language concept has been developed by a group of German research institutes and industrial firms (see list of authors).

The name PEARL (= process and experiment automation realtime language) has been chosen to indicate that it is intended to serve the needs of industrial programming as well as the requirements of laboratory application, because already in a very early state of the discussion it was seen that there existed no substantial differences in the language properties necessary for both fields of application.

The example chosen is a part of a very common application of realtime data processing in Nuclear Physics, but the same technique is used in quite a number of other classes of experiments.

GENERAL STRUCTURE OF A REALTIME LANGUAGE

Design criteria

A lot of people working in the field of realtime programming have had to make the experience that programming of larger on-line programs or systems in assembly language is very tedious and hence there is a rather general agreement on the necessity of a higher level language for this purpose.

One possible approach is the development of e.g. process-control FORTRANs and other extensions of existing programming languages by procedure calls to the operating system. But from the programmer's point of view and with respect to the efficiency of the object code a homogeneous language containing features for realtime programming seemed to be better.

Naturally the final product is highly dependent on the priority order which you give to your design criteria. Some of such objectives for a higher-level language for realtime programming are:

1) Machine independence (= program compatibility)
2) Learnability of the language (= programmer's compatibility)
3) Efficiency of the object program
4) Flexibility of the language

Unfortunately some of them nearly exclude each other (e.g. 1 and 3) and therefore one often has to compromise.

Another point is that the rapid development of hardware often makes it possible to implement things which beforehand were regarded impossible or at least highly inefficient. So a language development should not only take into account features which can be easily implemented now, but also try to influence the development in a way which encourages hardware engineers to introduce new principles which make realtime programming easier and safer.

General structure of the proposed language

PEARL is a procedural language like ALGOL,

FORTRAN, PL/1 or ALGOL 68. In its logic and arithmetic part it contains all the features of modern general-purpose programming language like PL/1 or ALGOL 68 insofar as they are necessary for real-time-programming (e.g. bit and string handling with the appropriate data types, definition and handling of data structures). Together with address variables it is so possible to do list-processing. The possibility of defining types and operators makes PEARL to a certain degree an extensible language. A means for time optimization of the object program in the insertion of procedures as quasi macros. But all these features represent more or less the "state of the art" and are not specific for realtime programming. The special "real-time" properties of a program are covered by four other main classes of language elements:

a) Language elements for handling of parallel processes and timing
b) Elements for the synchronization of these processes
c) Description of the hardware connections on language level
d) Possibilities to handle non standard I/O and data transfers within the system.

These special elements shall now be described in more detail.

REAL-TIME FEATURES OF THE LANGUAGE

Tasking and Timing

In our work a dynamic definition of the "task" like in PK/1 has been used: "The task is the (dynamic) execution of a program under control of the operating system". So a task has no static existence before it is created by some mechanism but afterwards it has got a rather far-reaching autonomy and can nearly be regarded as a processor in its own right - a virtual processor. Language statements referring to tasks have now got a meaning as commands to this virtual processor. Figure 1 shall illustrate this.

A task is "declared" at the begin of the block which contains a declaration of the following form:

task-declaration:: = TASK taskidentifier [: thread]

thread is the code of a task and may either be a block or a compound statement. Jumps out of the thread are forbidden for obvious reasons.
Once a task has been activated it is known to the operating system and to the scheduler if a schedule is connected to it. The activation of a task is of the form:

activation::= [schedule] ACTIVATE taskidentifier [PRIORITY expression-seven {REL | SYS}] [: thread] ;

The schedule is described by:

schedule::= { ,. { [AT clockelement] [{EVERY|ALL} duration element [UNTIL clockelement | DURING durationelement]] | [ON interruptidentifier|AFTER duration-element] [ALL durationelement [UNTIL clockelement|DURING durationelement]]}...}

So a schedule can either be a single delay:

```
AFTER 5 MIN ACTIVATE TASK1;
```

or a more sophisticated cyclic activation:

```
AFTER 10 MIN ALL 15 SEC UNTIL 5 OCL
ACTIVATE TASK2;
```

An active task may be made inactive for some time by the statement:

```
SUSPEND [taskidentifier] [UNTIL
    clockelement | DURING durationelement]
```

and reactivated by: CONTINUE.

By TERMINATE a task can be put in the inactive state before it comes to its natural end.
There is still a number of other possibilities for the initiation of transitions between states of the task, especially transitions which are not due to the execution of user program statements, but to actions of the operating system, e.g. reactivation of a task after completion of the time period for its suspension. But to describe them all in detail would fill a whole paper and so the above examples, though incomplete, shall be enough to give an impression of task handling in PEARL.

Synchronization

In principle there are two different methods of synchronization in a PEARL-program: implicit and explicit synchronization. Implicit synchronization is mainly achieved by the block structure of a program. A block cannot be left before the last task has been terminated to which in this block code has been allocated.

The main tool for explicit synchronization are semaphore-variables similar to the proposal of Dijkstra (1). They are declared by

DCL identifier [({ ,. bound pair } ...)] SEMA;

and contain an integer which is decremented by one by the operation:

```
REQUEST SEMA1;
```

Would the result of the operation be negative then the task which contains this statement is suspended and has to wait for another task which performs the operation:

```
RELEASE SEMA1;
```

This operation increases the value of a semaphore variable by one. Was it negative before, the waiting task with the highest priority is reactivated.

RELEASE and REQUEST can also operate on lists of semaphores. This can sometimes be regarded as a critical point but for certain cases it offers much more safety

against deadlocks than a sequence of "single" semaphore operations. As a further expansion of the semaphore concept, the BOLT-variables are proposed which allow simulatneous reactivation of a number of tasks without explicit semaphore for each of them.

System Division

Machine independence of a computer program seems to be very difficult to achieve in applications that use such diverse peripheral devices as e.g. real time measuring processes. Therefore the argument: "the more machine independent the more ineffective" is often brought up against higher level languages for real time purposes.

We think that a means to overcome this problem is to give the programmer a tool to handle hardware specifications on language level. A "system division" that is part of user written program is used to generate all those pieces of code which - for hardware reasons - must vary when the same program is run on different machines. Thus the "problem division" can be nearly machine independent.

As an example an input statement may be formulated as follows:

```
MOVE FROM ADCGAMMA TO BUFFER 1;
```

The proper specifications of the register length, the coding method and the connection of ADCGAMMA to the computer is done by declarative statements in the system division and used by the compiler to generate the most effective code for the data transfer that is possible according to these specifications. The system division can be used to specify:

a) The type and size of the machine (i.e. processor, storage), e.g.:

```
MACHINE; PDP8I-16k;
```

b) Peripherals of known type (to the compiler) = "standard peripherals" and the way they are connected, e.g.:

```
(1): COMP      <-> (2): CHAN    *(Ø);
(2): CHAN *(1)  -> (3): DIGOUT  *(Ø);
```

The number before the device type is used to identify the device itself and the number after the asterisk to indicate the connection points of the specific device.

c) Peripherals of unknown type ("special") and their connections, e.g.:

```
INDICATOR = (1): DIGIN *(1, 1, 2)
```

DIGIN is regarded as a register, where starting from bit 1 one bit is used.

d) The interrupts to be used by the program by allocating an identifier to an implementation dependent hardware specification number,

```
e.g.:  ENDOFCONV = (2Ø);
```

e) A list of flags (= status inputs that do not interrupt the CPU-program),

```
e.g.:  ADCREADY = (38);
```

Input/Output

While a great part of I/O functions like printing, reading paper-tape, storing on magnetic disks etc. is well known in the field of commercial and scientific programming - and therefore is handled in existing programming languages like COBOL or PL/1 quite thoroughly - there are data transfer problems that are specific for real-time, measurement and control applications and which must be reconsidered. The latter are: I/O from or to elements like ADC's, thermocouples or stepping motors (process elements) and the field of graphic man-machine communication.

The PEARL syntax makes a clear distinction between man-machine and machine-machine (or: inter-system) data transfers. A number of statements are used for man-machine communication, represented by character and graphic I/O. For our purposes as far as printed I/O (character strings) is concerned it is only necessary that formatting can easily be done. Simplicity is preferrable to powerful but complicated features. As an example please look at the following one-to-one mapping of "picture" specification and final layout:

Statement:

```
WRITE PI TO TTY WITH (PI = #$####);
```

Output:

```
PI = 3.1416
```

Graphic programming has become a special field of computer science. A number of very powerful software systems for generating and modifying graphic output exist. Many of them use problem oriented languages.

We think it is not necessary to have a very comfortable and powerful graphic language as part of PEARL, for this requires very complicated software packages. Nevertheless it is necessary to build up any graphic output by means of a set of graphic programming primitives; as well as to "work" with them using graphic input devices.

A graphic output statement has the following form:

```
DRAW data TO device WITH format;
```

data means any information to be displayed according to the prescription of format, which is similar to conventional string format descriptions. It includes the interpretation method (points, lines, arcs, automatic incrementing, intensity, colour, scale etc.).

Example:

```
DRAW CURVE TO DISPLAY WITH CURVEFORM;
```

(CURVEFORM is a remote format string which

is declared at another place in the program. For more details see example (3.3).

Another class of transfers is data transport within the system, before (1) classified as:

BINARY, CALIBRATED and PRIMITIVE I/O

It turned out to be possible to combine all these statements into one:

MOVE FROM data-source TO destination
[WITH option] ;

As an extra, this statement now can also describe device-device transfers which do not touch the CPU. The compiler can take information about the possible directions of data-flow out of the system-division as well as the distinction between the former three classes of process-I/O.

option contains the calibration routines which are provided by the compiler.

EXAMPLE

Problem

In order to illustrate the use of such realtime features in a programming language we choose a rather simple example which is taken out of the daily practice of Nuclear Physics data evaluation.

A complete program would have been much too complex but we choose parts which show quite well how appropriate language features can not only simplify programming but also help to clarify the mechanisms of task interaction.

The problem posed is to read in data from a fast ADC into two buffers in the computer. The contents of these buffers are then sorted into a spectrum in a separate area of the core memory. The spectrum can be displayed on a storage oscilloscope. The whole program can be controlled by software interrupts which are generated according to input strings from a console teletype. Figure 2 shows a scheme of the configuration.

Program structure

Tasking

Three of the major tasks of the program are:
a) Read data from ADC into buffers
b) Sort data into spectrum
c) Display spectrum
etc.

Another independent task is the main task which is activated after start of the program.
Task b is also activated after start of the program and then waiting for semaphores which indicate that the data buffers are full.
Task a and c and others are activated by users commands.

Synchronization

The buffers have to be protected so that one cannot overwrite a buffer when the data in it are still needed for sorting. The sorting programs have to be started after each buffer is full. This is achieved by means of the semaphores:

BUFFULL1, BUFFULL2, BUFEMPTY1, BUFEMPTY2.

It is not desirable that the spectrum is displayed while it is changed by the sorting task. This is prevented by

SPECSEIZED.

Graphic output

The spectrum is displayed on a storage oscilloscope as shown in fig. 3. The picture is refreshed all five seconds. The position of the vertical markers can be changed (LOMARKR, HIMARKR are variables) and the horizontal line indicating the background for eventual subtraction can be moved up and down.

(1) For all references see: J. Brandes et al.: PEARL, The Concept of a Process- and Experiment-oriented Programming Language; elektronische datenverarbeitung, 10, 1970, pp. 429

Code examples

```
/* EXAMPLE FOR A SYSTEM DIVISION */

SYSTEM;

MACHINE;
MODEL = S3Ø5;/* THE SAMPLE MACHINE IS A
        SIEMENS S 3Ø5 */
SIZE = 16; /* WITH 16 K OF CORE MEMORY */

EQUIPMENT;
(1): ZE3Ø5 <-> (2): CHANNEL;
                (2): CHANNEL*(Ø)<-> (3): BE2Ø18;
                          *(1Ø) -> (4): P3AG;
                          *(11)<- (5): P3EG;
                (3): BE2Ø18<-> QUESTIONER =
                     ANSWER = (6): FS1ØØ;
                     /*INPUT DEVICE =
                     TYPEWRITER */
                (4): P3AG ->DRAWINGDEVICE =
                     (7): 141A;
                     /*OUTPUT DEVICE =
                     OSCILLOSCOPE */
                (5): P3EG <- ADC = (8): AD161F;
                     /* ADC INPUT */

INTERRUPT;
COMMANDINTERRUPT = (5);  /* HARDWARE
                            INTERRUPTS */
START = (2Ø); /* SOFTWARE INTERRUPTS */
```

```
STOP = (21);
CANCEL = (22);
DISPLAY = (23);
DISPLAYOFF = (24);

/* SOME PARTS OF A SAMPLE PROBLEM PROGRAM */
PROBLEM;
BEGIN; /* DECLARATIONS FOLLOWING */
.
.
.
TASK READLOOP: BEGIN;
   /* TASK TO READ ANALOG-TO-DIGITAL-CONVERTER DATA INTO DOUBLE BUFFER; TO BE
   SYNCHRONIZED WITH SORTING PROCEDURE */
READADC PROCEDURE (BUF, BUFEMPTY, BUFFULL);
   DECLARE (BUFEMPTY, BUFFULL) SEMA;
   DECLARE BUF (BUFLENGTH) INTEGER;
REQUEST BUFEMPTY;/* INPUT INTO A BUFFER NOT POSSIBLE IF BUFFER NOT EMPTY */
MOVE ADC TO BUF; /* INPUT FROM ADC INTO BUFFER AREA */
RELEASE BUFFULL; /* IF BUFFER FULL, SORTING TASK IS STARTED */
END; /* READADC */
/* THE ACTUAL VALUES OF BUF, BUFEMPTY, BUFFULL, BUF1 E.T.C.       ARE DECLARED IN AN
   OUTER BLOCK */
READLABEL:  CALL READADC (BUF1, BUFEMPTY1, BUFFULL1); /* READ INTO FIRST BUFFER */
            CALL READADC (BUF2, BUFEMPTY2, BUFFULL2); /* READ INTO SECOND BUFFER */
            GOTO READLABEL;
END /* READLOOP */;



TASK SORTLOOP: BEGIN; /* TASK TO SORT DATA FROM BUFFERS INTO SPECTRUM AREA; TO BE
                         SYNCHRONIZED WITH READLOOP AND DISPLAY TASK */
SORTER: PROCEDURE (BUFFER, BUFFULL, BUFFEMPTY);
        /* PROCEDURE TO SORT CONTENTS OF ONE BUFFER */
        DECLARE (BUFFER (BUFFERLENGTH), J) INTEGER;
        DECLARE (BUFFULL, BUFFEMPTY) SEMA;
        SORTLABEL: REQUEST (BUFFULL, SPECSEIZED);
                   DO I = 1 TO BUFLENGTH; BEGIN; J:= BUF(I);
                   IF J > Ø THEN BEGIN; /* TEST FOR NON ZERO DATA WORD*/
            IF J < SPECTRUMLENGTH+1    THEN SPECTRUM(J):= SPECTRUM(J)+1;
            END;
        BUF: = Ø; /* RESET BUFFER AFTER SORTING*/
        RELEASE (BUFEMPTY, SPECSEIZED); END; /* SORTER */
TASK (SORTBUF1, SORTBUF2); /* DECLARE SORTING TASKS */
REQUEST SPECEMPTY; /* TEST FOR EMPTY SPECTRUM AREA */
SORTLABEL: ACTIVATE SORTBUF1;
           CALL SORTER (BUF1, BUFFULL1, BUFEMPTY);
           ACTIVATE SORTBUF2;
           CALL SORTER (BUF2, BUFFULL2, BUFEMPTY2);
     /* THE TWO SORTING TASKS ARE ACTIVATED AT ONCE AND WAIT FOR SEMAPHORES BUFFULL1 AND
        BUFFULL2 TO BE RELEASED*/
END; /* SORTLOOP */
.
.
.
/* EXAMPLE OF LISTS TO BE DISPLAYED; THEY DESCRIBE THE LINES IN FIGURE 3 */
DECLARE L1(1:4)STRUCT ((X,Y,I) VALUE INTEGER);
DECLARE L2(1:4)STRUCT (X INT,(Y,I) VAL INT);
DECLARE L3(1:2)STRUCT (X VAL INT, Y INT, I VAL INT);
L1: =((Ø,1,Ø),(Ø,Ø,2),(1,Ø,2),(1,1,2));
L2: = ((LOMARKR,Ø,Ø),(LOMARKR,1,2),(HIMARKR,Ø,Ø), (HIMARKR,1,2));
L3: = ((Ø,BACKGR,Ø), (1,BACKGR,2));
```

```
/* EXAMPLE OF DISPLAYTASK */
TASK DISPLAYING: /* TASK TO DISPLAY CONTENTS OF SPECTRUM;
                    TO BE SYNCHRONIZED WITH SORTING TASK */
ALL 5 SEC ACTIVATE DISPLAYSTART: BEGIN;
    SPECMAX: = MAX (SPECTRUM); /* MAX (..) IS A PROCEDURE TO FIND OUT THE MAXIMUM VALUE
                OF AN ARRAY*/
SCALING:  XSCALE:= N; /* LENGTH OF SPECTRUM TO BE DISPLAYED */
          XSCALE:= SPECMAX;
          CSCALE:= 1;
          BSCALE:= 1Ø;
          SPECFORM:=Y;
INIPIC: /* ESTABLISH SCALE, ORIGIN, COLOUR, BRIGHTNESS OF PICTURE*/
        DRAW XSCALE,YSCALE,Ø,Ø,CSCALE,BSCALE,1 TO DRAWINGDEVICE WITH SX,SY,OX,OY,SC,SB,C;
INIX:   DRAW Ø TO DRAWINGDEVICE WITH X; /* POSITIONING OF DRAWING POINT TO ORIGIN */
SPECOUT: /* DRAW SPECTRUM */
         REQUEST SPECTRUMSEIZED;
         DRAW Ø,1,3 SPECTRUM TO DRAWINGDEVICE WITH M,DX,B,SPECFORM;
         RELEASE SPECTRUMSEIZED;
INIFRAME:   DRAW 1,1 TO DRAWINGDEVICE WITH SX,SY;
            DRAW 1,L1,L2,L3 TO DRAWINGDEVICE WITH M,3(X,Y,B);
END; /* DISPLAYSTART */
.
.
.
/* HERE IS THE MAIN PART OF THE PROGRAM WITH THE ACTIVATION OF SOME MAJOR TASKS */
      ON START ACTIVATE READLOOP;
      ON DISPLAY ACTIVATE DISPLAYING;
      ON DISPLAYOFF TERMINATE DISPLAYING;
END; /* OF PROGRAM */
```

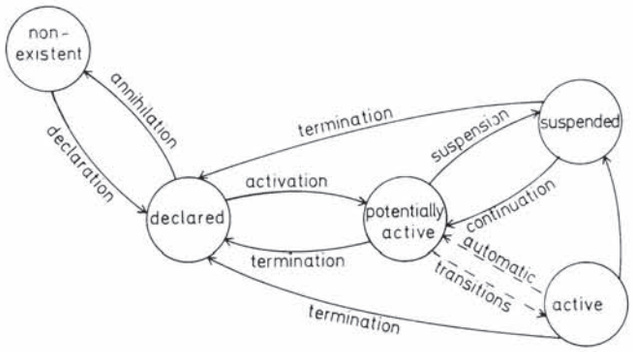

Fig.1 A task and its states (simplified)

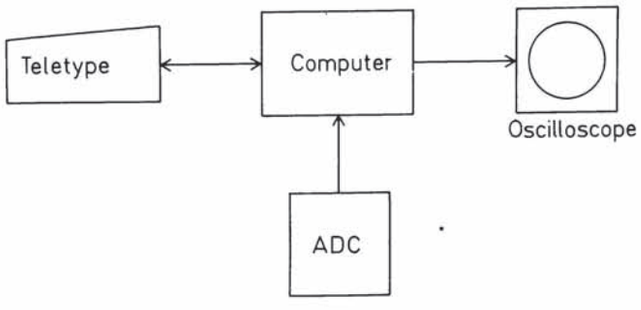

Fig. 2 Scheme of the configuration

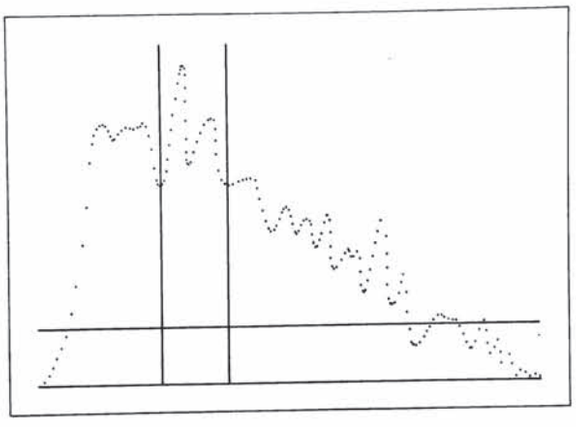

Fig. 3 Display of a spectrum