Ц 8406
M-30

СООБЩЕНИЯ
ОБЪЕДИНЕННОГО
ИНСТИТУТА
ЯДЕРНЫХ
ИССЛЕДОВАНИЙ

ДУБНА

3532/2-76

6/IX-76

E10-9757

D.C.Marinescu

EPL/1 (EXPERIMENTAL PHYSICS LANGUAGE ONE) - A PROPOSAL FOR A HIGH LEVEL MINICOMPUTER LANGUAGE FOR EXPERIMENTAL PHYSICS APPLICATIONS

1976

E10-9757

D.C.Marinescu

EPL/1 (EXPERIMENTAL PHYSICS LANGUAGE ONE) - A PROPOSAL FOR A HIGH LEVEL MINICOMPUTER LANGUAGE FOR EXPERIMENTAL PHYSICS APPLICATIONS

Маринеску Д.К. **E10 - 9757**

EPL/1 (язык экспериментальной физики) - предложение о языке высокого уровня для малых вычислительных машин, применяемых в экспериментальной физике

В работе дается характеристика языков программирования, разрабатываемых для построения систем реального времени на мини-машинах, которые применяются для решения задач экспериментальной физики. Предлагается новый вариант проблемно-ориентированного языка EPL/1, обобщающий основные возможности известных языков указанного назначения.

Работа выполнена в Лаборатории вычислительной техники и автоматизации ОИЯИ.

Сообщение Объединенного института ядерных исследований
Дубна 1976

Marinescu D.C. **E10 - 9757**

EPL/1 (Experimental Physics Language One)- a Proposal for a High Level Minicomputer Language for Experimental Physics Applications

The goal of this paper is to present a high level real time language developed for experimental physics applications.

In the first part of the paper the features of the language are examined; BNF description and the problems related to the implementation of the language are the subject of the second part. Finally a compiler for the language, written in CDL (Compiler Definition Language) will be described in a third part.

The investigation has been performed at the Laboratory of Computiong Techniques and Automation,JINR.

Communication of the Joint Institute for Nuclear Research
Dubna 1976

PART 1 EPL/1 features in the context of contemporary real time languages

CONTENTS

1. The state of art in the area of high level real-time languages for minicomputers.

1.1. Minicomputers in today's computing world

The minicomputer of today is a rather powerful computing machine with 16-64 K bytes of internal memory, an elaborated interrupt system and a large variety of I/O devices including minidisks,disks,displays,magnetic tapes,medium speed card readers and line printers,etc.,which can be connected to it.

Minicomputers rather low basic cost (in the range of 50-100 K$) and their suitability for real time applications are responsible for the high number of such installations in a wide ranging variety of applications like :

- discrete process control,
- continuous process control,
- conversational and interractive programming,
- information retrival,
- message processing,
- graphics,
- data logging,
- reservation systems.

Only in F.R.G. there are over 6000 such installations and their number increases every year by 25%.

Perhaps the highest number of such installations is working in the area of industrial applications for process control and management but still a significant number of them is in communication systems,reservation systems or in scientific applications.

The minicomputers are either dedicated to a process and working independently of each other or interconnected in a network;

sometimes such a network might contain one or several medium or large scale computers.

Since the minicomputer hardware price is not expected to drop significantly in the very next future (the minicomputers do not benefit extensively of the use of LSI) ,an effort is made today to reduce the cost of the software developement for them.

With the average productivity of 500-2000 assembler instructions per man,per year the idea of specific user designed programming systems for real time applications is no longer appealing.

The relative efficiency (in terms of CPU speed and core occupancy) of low level language written programs is now weighted against theinconveniences of such systems (the inflexibility during design ,the difficulties to modify, to extend and to use, the lack of communications between users with the same type of problems,etc.).

As a result a number of high level languages have been developed in the recent years for real time applications.

1.2. High level real time languages for minicomputers

Today any type of minicomputer has at least a compiler for one of the conventional high level languages, PL/I,FÓRTRAN,ALGOL, COBOL . A recent research of K.Thompson /1/ showed that at least 20 other real time high level languages have been developed for industrial applications only. Among them we note CORAL 66, PEARL, RTL2, PROCOL, BCPL, CSL, MARY, MACROTYPE, NELIAC, PROSEC, TPL, INDAC8, SAL, SEQUEL, SNIBBOL.

For scientific applications BASIC provides enough computational facilities and it is used quite often. But other languages have been designed for specific applications; as an example at CERN, NODAL has been developed for accelerator control.

Though it is rather difficult to think about an internationally standardized real time language,an effort is made in this direction. A Long Term Procedural Language (LTPL) will be defined by the International Purdue Workshop this year.

We shall now list several features which are commonly considered to be desirable in the areas of:

- data storage manipulation,
- function construction,
- language implementation,

but we are fully aware that several compromises are required when deciding to include such features in a newly constructed programming language for real time applications.

As far as data storage manipulation is concerned, the most important features are:

- ability to define and manipulate data structures
- string handling
- ability to manipulate data in stacks,queues,arrays
- automatic and controlled assignement of storage during run time.

As far as functions are concerned,the following features are highly appreciated:

- exception handling,
- I/O handling,
- interrupt handling,
- recursion,
- block structure,
- subroutine parameter specification,
- multiple entry and exit subroutine,

- function and expression evaluation,
- expansability.

From the point of view of implementation the following features are desirable:

- a good protection mechanism,
- standard compile time diagnostics,listings and cross refference,
- run time diagnostics,
- relocatability,
- page swapping,
- dynamic linkage between routines,
- facilities to access hardware error states,
- standard linkage to routines,
- reentrant code,
- data allocation at run time.

As pointed out earlier it is not conceivable to think of building all these facilities in one language though a user might like to use them all.

2. EPL/I (Experimental Physics Language One) Language description.

2.1. General considerations

Since a large number of minicomputer installations are now working in the scientific applications and many of them in the field of experimental physics we think that a high level language for these types of problems would be welcomed if it is carefully designed, neatly implemented and easy to be learned and used.

As far as the last requirement is concerned, we must keep in mind the fact that presently most of scientifical programming is in FORTRAN although other remarkably good languages are available.

This means that a FORTRAN-like language would be the first choice of most real time users in experimental physics if they think that the time for a high level language for their type of problems has come. And this last problem is a matter of dispute; most people still think that they have nothing but little to communicate with each other since the type of data processing, which is subject of their interest, is not only determined by the type of their experiment and by the minicomputers available for them but by many other factors like the specific techniques of the experiment, the availability of higher level computational facilities (large scale computers with specialised programs which fit the experimental data with one or other theoretical model), etc., and thus, the advantages of a high level language are considerably diminished.

We strongly oppose to such an approach with the arguments already presented in the previous section.

But we think that there is a point when postponing the decision to design such a language. And the point is that any mistakes

made in the process of designing of such a language will have implications later on when the language has a widespread use. Again a good example in this direction is FORTRAN; an estimation made by Iann M.Barron /2/ shows that a more carefull design of FORTRAN which could have added to its initial developement cost 10M$ in 1960, would save 1000M$ each year (today 30% of programming is in FORTRAN and the annual expenditure for computers is 15,000M$).

But this only shows. that if a step forward should be made in the direction of the development of such a language, then the step must be carefully made, large groups of people must be consulted and their wishes and proposals must come under close scrutinity.

The intention of this paper is to provide a starting point for future discussions on this topic. We are fully aware that the solutions we propose here can be the subject of criticism.

We think that there are several crucial problems to which the language must provide resonable solutions:

-1. The problem of communication with different experimental on-line devices. The information which forms the object of this communication must be easily available for computation but at the same time, like in low level languages the user should have the facility to test the status of the external device, to delay any computation untill all information is transferred and to provide special procedures for other types of interruptions generated by the device.

-2. The problem of adequate data structures. Since the structure of the information largely depends upon the type of the experiment the language can either provide a large number of data structures or can allow dynamically constructed data structures and operators.

-3. The problem of communication with other program modules written in other languages. In real time systems the problem of driving special external devices or of handling interrupt conditions is most suitably solved in machine language written program modules. Therefore the facility to call machine language or assembler modules is highly recommended. At the same time since on almost every minicomputer there is a compiler for FORTRAN or another high level language frequently used for scientifical data processing, the facility to call such program modules is desired.

-4. The problem of efficient handling of rather large volumes of data which must be kept on secondary memory for later processing.

-5. The problem of efficient debugging and of tools to assist the user when constructing his programs.

We shall briefly examine now a set of solutions for these problems.

In any experimental environement the computer communicates with the external apparatus via a standard mechanism. A number of channels is hardware built into the machine and all external devices are attached to such channels. Regardless of the complexity of the device its status can be examined by the channel and the information can be transferred in one sense or another (read or write operations) under channel control. If the transfer is successful the information is available in a memory buffer after a read operation or is transferred from this buffer to the device in a write operation. We shall call such nonstandard I/O devices 'system objects'; a name will be used to identify both the information read or to be written from or to such a device and the device itself. In terms of the high level language several standard operations can be requested in connection with such an

object, e.g., read, write, test, wait. When a statement of this type (for example READS(name)) is encountered, the entry point in the device access module, supplied when the system object has been defined, is provided. We think that such a tool is powerful enough to enable the user to handle his experimental device in a conventional manner without being exposed to the difficulties of machine language written programs which are now the responsability of the system programmer installing the system. Still the user must be aware of the structure of the information which forms the object of the communication with the external device.

Also a certain number of condition switches associated with different levels and sublevels of interruptions is to be defined when the system is installed. The language must have the facility of the 'ON (condition) GO TO (statement label)' statement to test condition switches and to provide linkage to the user own interrupt processing routines.

Another delicate problem is the one of data structures; normally depending upon the type of the experiment a data structure type or another one is desirable. A solution is to allow the user to define his own data structures in terms of the few basic data structures of the language. Often an experimental event consists of a set of nonhomogenous data describing the conditions of the experiment, the information concerning the state of different hardware elements and the information pertinent to the investigation itself. It should be possible to refer to the whole data structure or to the elements of its substructure easily.

Also special operators useful for the manipulation of such structures should be dynamically constructed when needed.

A CALL, SAVE, RETURN procedure based upon a standard linkage convention (a register containing the address of the parameter list,save areas in the calling program module,etc) should be used by all compilers so that modules produced by them can be linked together to form an executable code.

Of course when designing the language a compromise must be made between the degree of flexibility we built into the language and the difficulties we are willing to undergo when the problem of implementation arises. And this was the reason for many decisions limiting the number of sophisticated options in the process of language definition.

2.2. Language definition

2.2.1. General specifications

We do not intend to give here full specifications for the language but to present an overview of its facilities as compared with the ones in BASIC, the language which has been used as a reference.

An EPL/1 program has a modular structure; it consists of a main program and the set of subprogram modules called by the main program. Definition blocks for system objects , data structures, operators,files,subprograms must be entered in the user computational space (either defined by the user himself or made available from the system pool or from another user computational space) prior to their use.

A program module (block) consists of a set of statements; each statement must be completed on a single line and it consists of up to 72 characters. A statement is identified by a line number and containes one of the keywords of the language.

Numerical (real or integer) and string constants are allowed.

Internally , string constants of 1-127 characters are represented in character format prefixed with the word specifying the length.

There are three types of simple variables:integer,real and string variables. For simple numerical variables (integer or real) the name consists of one to six alphanumeric characters,the first of which must be a letter. The convention that names begining with letters I-N are reserved for integer variables ,is valid.

An explicit declaration of type can alter the implications of this convention.

A variable of the type string has a name consisting of one to six alphanumeric characters and a $ subfix; if not otherwise specified a standard length of 8 is assumed for a string variable.

Names for user defined data structures,files,system objects, user defined operators type follow the general convention for identifiers (1-6 alphanumeric characters,the first a letter) but have a dedicated prefix: D. for data structures,S. for system objects, F. for files, O. for operators , P. for subprograms. Actual names for the same type of objects (data structures,files, operators,system objects,subprograms) are standard identifiers. The correspondence between an object type and its actual name is established via a DECLARE statement prior to their use. Also declaration of type for simple variables,declarations of type and maximum size for arrays must be present in any program module using them,with the understanding that such variables belong to the program module where they first appeared (where allocation is performed); if such a variable appears in the argument list of the subprogram the USING statement only informs the compiler how to treat it ,no allocation is performed.

In order to provide automatic data swapping VIRTUAL mode can be specified when working with arrays of data structures. A virtual address space is defined when the system is installed; it consists of a set of pages. The page size is selected at that moment depending upon the size of available internal memory,the disk organization and other arguments.

The size of an element of a data structure of type array, cannot exceed the page size. When arrays of a certain type of data structure are used in a virtual mode only one element of this array is present at one time in core.

It should be observed that only linear arrays of reals, integers or strings are available. Multidimensional arrays can be constructed as user defined data structures.

Arithmetic and relational expressions are allowed.

The standard mathematical functions are on line (SIN,COS, TAN,ATN,EXP,LGT,LOG,ABS,SQR,INT,SGN);a number of system functions (TOD$,DATE$,TIME$,TIME etc.) as well as BASIC like string functions (LEN,STR$,SUBSTR,VAL) are built in.

We shall now present a nonexhaustive list of the statements available in the language.

- Any definition block must begin with a DEFINE statement and the last statement of it must be an END statement. Subprogram definition blocks must contain at least one RETURN statement.The general forms of these statements are:

```
DEFINE  t.name
        here  name  represents the name of a certain type of
                    object  created by the definition block
              t     is a prefix.Each type of object  which
                    can be created by a definition block has
                    a dedicated prefix:
                    D  for user defined data structures
                    O  for user defined operators
                    F  for user defined files
                    S  for user defined system objects
                    P  for subprograms
RETURN
END
```

- The DECLARE statement is used in the main program or in any subprogram to establish the correspondence between the actual name of an object (data structure,operator,file,system object, subprogram) created by the user and the name of the prototype of that object as given in the define block e.g. :

DECLARE $name_1$,$name_2$, ...$name_k$ /t.name/

here $name_i$ are actual names of defined objects;for data structures,$name_i$ can be of the form 'data structure name'(d) with d the dimension of an array of such data structures.

This statement is also used to declare the dimension of arrays or the length of nonstandard strings.

- USING statement is used in any definition block to provide information about the type of different variables;its format is:

  USING $name_1$,$name_2$,.......$name_k$/t.name/

it should be noted that in a DECLARE or in a USING statement the prefix t. may not be present;in this case 'name' must be either REAL or INTEGER or STRING.

- A comment statement must start with the REM keyword,e.g. :

  REM c

  here c is the body of the comment,any string formed with the characters available in the language.

- Standard assignement statement of the form

  v=e exists.

- There are three types of branch statements:

```
IF r THEN  n     here: r is any relational (logical) expression
                       n is a statement number
GO TO n          here  n is a statement number
ON e GO TO n     here: e is a logical expression used in
                       conjuncture with a condition switch
                       or an interrupt level previously defined
```

- Loops are implemented via :

  FOR sv=e_1 TO e_2 STEP e_3

  NEXT sv

  here: sv is a simple integer variable

  e_1,e_2,e_3 are integer constants or integer variables previously defined.

- The subprogram call is performed via the statement:

  CALL name(par_1,...par_n)

  here: name is the actual name of a subprogram;it must have been previously encountered in a DECLARE subprogram statement.

  par_1...par_n are the actual parameters of the call

- In addition to standard I/O operation (READ,WRITE) defined for system files ,there are specialised read,write,wait and test operations defined for user files or for system objects (READF, READS and so on)

- The definition blocks for system objects,files and data structures and operators have a specific syntax .

2.2.2. System objects

As pointed out earlier 'system objects' provide a general means to control the hardware associated with the experiment.

They are defined in special program blocks called 'define system object block' which have a specific syntax.

The actual name of the system object acts both like a variable of the type and size specified in the definition block and as an identifier for a hardware unit which can be subject to a READS, WRITES, TESTS or WAITS operation. (Abbreviations can be used as RS, WS, TS, AS). Names for the routines performing each of the operations, the unit can be subject of, must be supplied at definition time.

As an example let us construct the definition block for a system object of type S.ANALIZ ,with the physical address IODEV=213,linked to interrupt level 10 and transfering as data an array of INTEGER elements of size 512; RAN,TAN,WAN are the names of machine written routines for read,test and respectively "wait" operations.

```
10  DEFINE S.ANALIZ
20  TYPE=INTEGER
30  SIZE=512
40  READMOD=RAN
50  TESTMOD=TAN
60  WAITMOD=WAN
70  IODEV=213
80  LEVEL=10
90  END
```

A program declaring the actual name BETA for a system object of type S.ANALIZ will be presented in a following paragraph.

2.2.3. Files

The files are collections of data the language can manipulate; a file consists of a set of records with the same structure. A record can contain only data or a key and data.

The structure of a file must be described in a special 'define file block' according to a specific syntax.

Also information about the physical location of the file must be supplied (device address and extent when working with a direct access device).

In order to access a certain record in the file the user can either specify the value of the KEY (if format data+key is used) or a value for the associated variable. The associated variable is a pointer within the file to the record currently accessed;it must be declared when declaring the use of the file and throughout of the program block it becomes a reserved identifier.

Only a pure sequential access is allowed for files residing on devices other than direct access ones. Even for this type of files a pure sequential access can be used (no key or associated variable declared).

There is a number of dedicated files (INPUT,OUTPUT) normally attached to the terminal but the user is able to use own files instead of system ones when needed.

2.2.4. User defined data structures and operators

The arguments pleading for user defined data structures and operators are strong enough to justify the effort to built them into the language; they offer a high degree of flexibility and allow experienced users to make an efficient use of memory space when dealing with large collections of data.

Such structures can be defined in special 'define data' or 'define operator' blocks ; this must be done prior to their use. Then they must be declared in any program block using them.

As a first example we will define a new data type COMPL as a pair of real numbers and an operator PLUS which performs addition of such data types.

```
10   DEFINE D.COMPL
20   USING A,B/REAL/
30   D.COMPL=(RE,IM)
40   RE=A
50   IM=B
60   END

10   DEFINE O.PLUS
20   USING A,B,C/D.COMPL/
30   C=A(O.PLUS)B
40   RE(C)=RE(A)+RE(B)
50   IM(C)=IM(A)+IM(B)
60   END
```

The program called JUDY uses these structures; it should be observed that when an overflow condition occures a specially written routine CHECK is used.

```
10   MAIN JUDY
20   DECLARE A(100),B(100),C(100)/D.COMPL/
30   DECLARE P/O.PLUS/,M/O.MULPLY/
40   DECLARE CHECK/P.SUBPR/
50   READ A,B
60   ON (OVFL) GO TO 150
70   E=0
80   FOR I=1  STEP 1 UNTIL 100
90   C(I)=(A(I)(P)(B(I))(M)(A(I))
100  F=SQR(RE(C(I))**2+IM(C(I))**2)
110  E=E+F
120  NEXT I
130  WRITE E
140  STOP
150  CALL CHECK
160  END
```

In this program the operator of type O.MULPLY is used for multiplication of data of type D.COMPL .

Nonhomogenous data structures are of a special interest. As an example we shall define a data structure of type D.EVENT which can contain all the information about an angular distribution type experiment. The structure of such a data is:

- DATE - a string of 18 characters representing the date and the time when the experiment was performed
- TIME - an integer representing the duration of the irradiation
- COND - a string of characters (length 50) representing the experimental conditions
- ANG - an integer specifying the angular position
- PULSE(I),I=1,512 - an integer array containing the number of pulses on the 512 channels of an analyzer.

The corresponding 'define data block' is presented below.

```
10    DEFINE D.EVENT
20    USING A$/STRING(18)/,B$/STRING(50)/
30    USING C/INTEGER ARRAY/
40    USING INTV,ANGPOS/INTEGER/
50    EVENT=(DATE,TIME,COND,ANG,(PULSE(I);I=1,512))
60    DATE=A$
70    TIME=INTV
80    COND=B$
90    ANG=ANGPOS
100   EVENT=(C(I);I=1,512)
110   END
```

A program using such a data structure to repeat 100 times the experiment for angular positions between 32 and 132 degrees and to perform readings from the system object of type S.ANALIZ follows.

```
10    MAIN XSEC
20    DECLARE BETA/S.ANALIZ/,VERIFY/P.PR/,PR1,PR2/P.INTPRG/
30    DECLARE OUTCOM(100)/D.EVENT/,VIRTUAL
40    DECLARE SEC/F.MYFILE/
50    ON (INTR10)  GO TO 260
60    ON  (ERROR)  GO TO 270
70    TESTS(BETA)
80    CON$='     ....'
90    K1=32
100   K2=132
110   FOR I=K1 STEP 1 UNTIL K2
120   READS(BETA)
```

```
130   DATE(OUTCOM(I))=TOD$
140   TIME(OUTCOM(I))=180
150   COND(OUTCOM(I))=CON$
160   ANG(OUTCOM(I))=I
170   WAITS(BETA)
180   FOR J=1 STEP 1 UNTIL 512
190   OUTCOM(I,J)=BETA(J)
200   NEXT J
210   WRITEF(SEC) OUTCOM(I)
220   WAITF(SEC)
230   NEXT I
240   CALL VERIFY(OUTCOM)
250   STOP
260   CALL PR1
270   CALL PR2
280   END
```

It should be observed that two copies of the interrupt processing program of type P.INTPRG ,PR1 and PR2 have been supplied for the processing of two types of interrupts, the ones associated with the condition switches named INTR10 and ERROR. The two interrupts are of a different level and consequently the interrupt processing routine for one can be interrupted by the other; thus the problem of reentrancy of interrupt processing routines is solved at user level.

In this program the name OUTCOM appears without subscripts,with one subscript and with two subscripts;in the first case it refers to an array of data structure,in the second case it refers to an element of this array (the structure of such an element is defined by the D.EVENT definition block) and in the third case it refers to an element of the array PULSE (the J-th element) for the I-th outcome of the experiment described by the I-th element of the array of data structure. Statements 130 to 160 assign values pertinent to an outcome of the experiment to various elements of the data structure; TOD$ is a system function which returns as a string of characters the Time of Day and the date. The 'wait system object' opperation WAITS(BETA) is necessary to make sure that data transfer is finished before the assignment at line 190 is performed.A similar function has the 'wait file' WAITF(SEC) operation. The VIRTUAL mode has been declared for array OUTCOM since the subprogram VERIFY needs the whole array to check the correctness of experimental data and their consistency.

Several things must be pointed out in connection with user defined data structures:

- a. As far as the syntax of a 'define data structure block' is concerned, the following rules must be observed:
- USING statements must provide to the compiler all the information about the type of data used when defining the structure.
- a model (a prototype) of the data structure must be given and the names of all structural elements must be supplied.
- computational relations for each element of the structure are to be included in the definition block.

- b. Only one array can appear in a data structure element. While to simple substructure elements we refer specifying their name as given in the DEFINE block ,to the array elements we refer with the structure name followed by one or two subscripts (one if it is a simple data structure and two if an array of data structures).

- c. Arrays of data structures can be used as soon as the data

structures have been defined. As an example the program XSEC uses the array OUTCOM(100) with 100 elements; each element is a data structure of the type D.EVENT.

- d. A data structure defined in a DEFINE block cannot be used as a basic element when defining other data structures.

From examples given here the construction of the define data block for bidimensional arrays is straightforward.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Dr. V.P.Shirikov and to A.A.Hoshenko for many valuable suggestions and for the friendly help.

BIBLIOGRAPHY

/1/ K.Thompson - A survey of real time language standards for industrial use, in Minicomputer Forum 1975 Conference Proceedings.

/2/ Iann M.Barron - The decline.and fall of the computer, in Minicomputer Forum.

/3/ C.H.A.Koster - CDL a compiler implementation language, University of Berlin Report,1975.

/4/ *** - CDC; BASIC language reference manual.

/5/ *** - CDC; INTERCOM reference manual.

Received by Publishing Department
3 May, 1976.