CERN-Data Handling Division
DD/82/8
P. Hartel*
May 1982

PASCAL FOR SYSTEMS PROGRAMMERS

(Presented at ECODU-32, Montreux, 23.9.-2.10.81)

*now at:

KUN
University of Nijmegen
The Netherlands

DD-jt

Title: **Pascal for systems programmers**
Speaker: **Pieter H. Hartel, CERN, KUN**

I. Implementation languages

Traditionally operating systems have been written in an assembler language. OS/360, SCOPE 2, SCOPE 3 and NOS are just a few examples. However people have realized for quite some time already, that higher level languages help increase programmer productivity and facilitate debugging, even when writing operating systems [BRO75]. A very succesful example of a system, entirely written in a high-level language is UNIX* [RIT75]. The implementation language of UNIX is C [RIT78].

A major problem arises, when modifications or extensions have to be made to the outside of a system written in assembler, when other languages are available, which by themselves are far more suited to the purpose. These languages often lack proper compile and run-time support to implement the required interactions with the operating system.

Modifications to the kernel of an operating system are probably best made in the implementation language of the system, in order to maintain the structural integrity of the system.

Outside this kernel, there usually exists a host of utility programs, to manipulate files and directories etc. These are sometimes written in less primitive languages, such as FORTRAN. It is only logical to try to go a step further, and use more modern languages, that allow structured programming, such as PL/I and Pascal. The language chosen must however be more or less integrated in the system, in the sense that system functions must be callable without having to write parts of the code in assembler.

II. Pascal

The language proposed in this article to write system utilities for the NOS/BE operating system is Pascal [JEN76]. The following features of the language have determined the choice:

- Constant declarations allow for parameterising of program modules, which improves maintainability.
- The basic data types **boolean**, **char**, **integer**, **real**; the pointer, set and enumeration types and the type constructors **array**, **file** and **record** allow the programmer to express precisely how his data objects are structured. At the same time it allows the compiler to perform its rigorous type checking. This avoids some of the runtime overhead necessary for other languages.
- Block structure (**begin** ... **end**) and good control structures (**case** statement, **if** ... **then** ... **else** statement and three looping constructs) greatly improve the readability of Pascal programs. The flow of control in a program is usually obvious from the source text alone. No flow charts and the like are needed to design and understand a Pascal program.
- The language has been designed such that the task of the compiler writer is not a too difficult one.
- The Pascal 6000 Release 2 compiler from the ETH at Zürich and its successor the Pascal 6000 Release 3 compiler [STR79] from the University of Minnesota very efficiently implement Pascal on CDC Cyber computers running under various operating systems.

---

* UNIX is a trade mark of Bell laboratories

It should not be surprising, that implementations exist for many other machines and systems. In the rest of this article the implementation referred to is Pascal 6000 Release 3.

2.1 Problems for systems programmers

Unfortunately only a minimal system interface existed for the language. Basic input/output operations, and some auxiliary system functions (eg. *date* and *time)* are available in the implementation. The systems programmer, needing more than just that, is quite often faced with the problem, that Pascal does not allow an easy treatment of erratically behaving system functions. Since most systems have grown over the years, inconsistencies have crept in, and originally clean interfaces have not rarely become rather messy.

These problems can in general not be solved in the spirit of the language. The type checking mechanism may have to be defeated, for instance to describe a certain memory location as an integer value on one occasion, and as an address on another.

2.1.1 The record case variant

Pascal allows for such tricks, by using the **record case** variant declaration:

```
{$T- switch runtime pointer checking off}
var
   word: record case boolean of
      true: (adr: integer);
      false: (val: ↑integer)
   end;
begin
   word.adr := 1;
   word.val↑ := -1; { "PP call error" }
```

Note, that it is explicitly stated what type conversion is to be done.

2.1.2 External procedure declarations

Using separately compiled procedures and functions also allows the programmer to defeat the type checking mechanism, by making the actual procedure or function heading differ from the external declaration. This is far more dangerous than the **record case** variant trick, since both declarations will not be part of the same program text, and the fact that this trick was used will be far from obvious. For example a procedure to copy a file may be declared like:

```
procedure copy (var f, t: text); extern;
```

The actual routine text in some other file:

```
type
   fcb = record
      ...
   end { File Control Block };
procedure copy (var f, t: fcb);
var
   ...
begin
   ...
end;
```

This gives the actual routine text access to the fields of the file control block variable *fcb* for fast block transfers of data.

2.1.3 Packing of data

Pascal allows data in **record** and **array** structures to be **packed**. The compiler will compute the minimal size in bits for each member of a **packed record** or for the elements of a **packed array**. The elements of a **packed** structure are then fit together as tightly as possible. This is of great value, when fields of tables in the operating system are not aligned on word boundaries. The following example defines a 32-bit field in terms of 8-bit bytes and assigns respectively the values 1, 4, 9 and 16 to the bytes:

```
type
   byte = 0 .. 255;
var
   word: packed array [1 .. 4] of byte;
   i: integer;
begin
   for i := 1 to 4 do
      word[i] := i * i;
```

III. POST

During the last three years about half a man-year was spent to describe the NOS/BE tables and user-system communication areas in Pascal as **record** and **array** declarations. The POST (Pascal for Operating System Tricks) library is the collection of these declarations, together with many related **procedure** and **function** declarations. The library may be seen as a model of the operating system in Pascal.

The source text of the required declarations must be inserted in a Pascal program, via the Pascal 6000 Release 3 Include facility [STR79].

The POST source library includes constant, type, variable, procedure and function declarations. The POST object library contains a (small) number of compiled Pascal and assembler routines. The obvious reason for supplying as many routines as possible in source form, is to allow the compiler to check calls and parameters for type compatibility. The only minor disadvantage is, that compilation time increases significantly.

3.1 POST constants

The constant declarations in the POST library typically define table lengths and the size of communication areas in terms of computer words. Also the size of the computer word is defined in terms of bits, bytes, characters etc.

In this way, program maintenance becomes easier, since changes in table sizes require one source line to be modified, and recompilation of all programs that depend on that particular value.

3.2 POST types

The **type** declarations come in various flavours. There are some types that describe strings of bits in terms of basic types eg.

```
bit10 = 0 .. 1023;
rclflgval = (norecall, recall);
char7 = packed array [1 .. 7] of char;
set6 = set of 0 .. 5;
```

An example of a type which is more specific to the environment is the definition of a "SCOPE Logical file name" as it is used in the SCOPE 3 and NOS/BE operating systems:

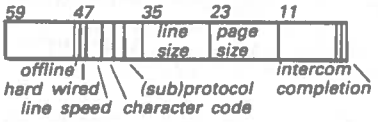
```
left7 = packed record case integer of
   0: (tag: bit42);
   1: (c1: char; b36: bit36);
   2: (c2: char2; b30: bit30);
   3: (c3: char3; b24: bit24);
   4: (c4: char4; b18: bit18);
   5: (c5: char5; b12: bit12);
   6: (c6: char6; b6: bit6);
   7: (c7: char7)
end;
```

The entire structure occupies only 42 (consecutive) bits. It may be cleared out by assigning a zero value to the *tag* field of the structure. Leading (sub)strings of the file name, which is left justified within the field, may easily be extracted or inserted.

The declaration of *left7* demonstrates how design flaws in an operating system lead to complication. The system uses a 6-bit character code (Display code), but makes the character with the representation 0 inaccessible in normal text (The problem does not arise, when the 64-character set is used).

Finally many tables are described, such as the *FDB* (File Definition Block), the *actb* (ACT communication area), the *PFDentry* (Permanent File Directory entry) and the *rweb* (RWE communication area). The latter will serve as an example.

The Peripheral Processor programs on a CDC Cyber perform almost all system functions. One of these, the program RWE (aReWE) returns the status of an interactive terminal to the program currently assigned to that terminal. Its communication area consists of one (60-bit) word, with the various fields layed out as follows:



The description of this word in Pascal is:

```
{ Character codes }
rweccval =
( rweASCII,    {  0      ASCII }
  rweExtBCD,   {  1      External BCD }
  rweDis       {  2      Display code }
);

{ Line protocol values }
rwelprval =
( rwelpr0,
  rwemode3,    {  1      Mode-3 protocol }
  rwemode4,    {  2      Mode-4 }
  rwemode2,    {  3      Mode-2 }
  rwewb        {  4      Wide band }
);

{ rwe parameter block }
rweb = packed record case integer of
   0: (tag:          integer);
   1: (
            u1:  bit12;  { unused field }
      offline:   bit;   { bit set if terminal offline }
      hardwired: bit;   { bit set if not dial-up }
      linespeed: bit3;
      charcode:  rweccval;
      subprot:   1..2;  { sub protocol }
      protocol:  rwelprval;  { line protocol }
      linesize:  bit12;  { character per line }
      pagesize:  bit12;  { lines per page }
            u2:  bit10;
      intercom:  bit;
      compl:     bit
      )
end;     { rweb }
```

3.3 POST procedure and function declarations

The **procedure** and **function** declarations in the POST library provide linked list processing, date and time conversions, string to integer and integer to string conversions, operating system interfacing and some limited string handling capabilities.

A very simple example of such an operating system interface routine is the **function** *interactive*, which returns a **boolean** result, depending on the mode of operation of the calling program.

```
procedure system1 (pp: char3; r: rclflgval;
   var par: integer); extern;

function interactive: boolean;

{**************************************
   function to return true if we are at an
INTERCOM controlpoint.
**************************************}

var
   par:        rweb;
begin
   with par do
      begin
         tag := 0;
```

```
        system1 ('rwe', recall, tag);
        interactive := intercom = 1
      end
end;    { interactive }
```

The **procedure** *system1* is the equivalent of the assembler macro *SYSTEM*, which implements a system call. The address of the communication area is passed to the peripheral processor program via the operating system monitor.

3.4 There comes a new release...

Programs, that heavily use the operating system may have to be modified, when changes appear in the interface area. If for example, the layout of a table is changed, because fields have been added, others may have been moved.

The FORTRAN programmer would have to work out the addressing and/or shift and mask instructions again, whereas the Pascal programmer merely updates (once) the declaration that corresponds to the table and recompiles the software, that makes use of it. Now the Pascal compiler will work out the necessary addressing, shifting and masking.

This may lead to a significant gain in programmer productivity with respect to the traditional low level language approach.

IV. Parser Generator

Unfortunately Pascal does not provide proper string handling capabilities. Therefore, passing arguments to a Pascal program (except for file names), or interpreting input directives is not straight forward. In addition to the lack of support for this type of operation, there is always the problem of having to react sensibly to the specification of erroneous inputs.

Since most operating system commands accept either position dependant parameters, or keyword parameters or a mixture of both, and the size of the command strings is in general limited to a fixed number of characters, a specialized tool has been developed. This is a Pascal program with the following characteristics:

- ☞ It allows flexible and legible specification of the syntax of the strings accepted.
- ☞ It is possible to associate (semantic) action calls with each terminal or non-terminal symbol as it is recognized, which allows for semantic checks to be performed, and parameter values and the like to be saved for later use.
- ☞ A standard action is taken upon detection of an error in an input string, thereby providing a uniform treatment of (syntax) errors.

Such a tool is called a parser generator. These are often used to help in writing syntax directed compilers or translators. The program discussed here generates code, which is compact and efficient enough to justify its use, even for almost trivial applications.

The code is produced as source text in the form of procedures which are to be embedded in the framework of the main program. This allows the Pascal compiler to perform its rigorous type checking on the complete source program.

4.1 Input to the parser generator

The main part of the input to the parser generator consists of a context free grammar with certain restrictions. In addition to this, one or more regular expressions may be specified to define the tokens handled by the parser, and some auxiliary definitions, which merely serve to reduce the memory requirements of the generated code [AHO78], [GRI71], [HOP69].

4.1.1 Auxiliary definitions

For most applications, there is no need to treat say each of the 10 digits differently, so they may be grouped together by an auxiliary definition:

```
char
  digit = 0|1|2|3|4|5|6|7|8|9;
```

4.1.2 Token definitions

Tokens are terminal symbols, which have very simple structures, such that they can be defined by regular expressions. This improves the performance of the parser as a whole, since some optimisations can be carried out on the corresponding generated code.

```
token
  identifier = letter {letter | digit}* ;
```

This example defines an identifier in the Pascal sense if suitable auxiliary definitions for *letter* and *digit* are made (ie. a letter, optionally followed by any number of letters and digits).

4.1.3 Rule definitions

The parser determines in a top-down, non-backtracking fashion whether an input sentence is valid, and activates semantic actions as required.

The example below defines simple arithmetic expressions of integer numbers, with unary plus and minus; addition, subtraction, multiplication and division, all with the proper priorities and parentheses.

```
rule
  expression:
    [minus #push(0)# term #subtract# | [plus] term]
      {plus term #add# |
       minus term #subtract#} ;
  term:
    factor
      {times factor #multiply# |
       slash factor #divide# |
       MOD factor #modulo# |
       DIV factor #divide;truncate#} ;
  factor:
    integer #push(intval(tk,tkl))# |
    leftparen expression rightparen ;
  typein:
    bol expression #print;pop# eol ;
```

The meta symbols have the following significance:

- ☞ The Parser loops on a clause in braces { ... } until a mismatch occurs.
- ☞ Square brackets with alternatives [... | ...] indicate that one of the alternatives must be chosen, whereas square brackets without alternatives [...] delimit optional clauses.
- ☞ The semantic action calls are enclosed in number signs # ... #. The current token may be accessed via the variables *tk* and *tkl* which respectively contain the token as a character string and its length.

V. Examples

At CERN over 30 system utilities have been written in Pascal. Some are rather small and some are very sophisticated. The average number of lines per program is 900, including some 50 lines of syntax. The production of the total 27000 lines of source text have cost less than 3 man-years.

A simple program will be discussed in some detail, to illustrate the ease of writing, and understanding Pascal programs with the support of the parser generator and the POST library.

The CERN AUDIT program will serve as a second example. Unfortunately the size of the program does not permit its full treatment here. Instead only some characteristic data will be shown.

5.1 SLAVE

The system label within the NOS/BE operating system contains some information to identify the site, system, machine etc. At CERN it was decided to include the day, time and type of the last system deadstart in this label. SLAVE (System Label Alter and VErify program) updates this information and checks, that the loaded system (CMR) and the magnetic tape used to bootstrap the system (deadstart tape) bear equal version numbers. In addition to this, SLAVE checks the date and time for validity, as the operator has to enter them each time the system is deadstarted.

5.1.1 SLAVE control statement syntax

There are three parameters to the SLAVE command, all of which are of the general form *keyword = value*. Any combination of parameters may be specified in any order. Keywords may be abbreviated to any number of letters. Otherwise the calling sequence entirely conforms to the standard in NOS/BE.

The complete syntactic specification of the control statement of SLAVE is the following:

```
char
   letter = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z ;
   digit = 0|1|2|3|4|5|6|7|8|9 ;
   quote = $ ;
   equal = '=' ;
   separator = ',' | '(' ;
   terminator = . | ')' ;

token
   alpha = {letter | digit}+ ;

terminal
   alpha ;

rule
   version—parameter:
      VERSION equal alpha #setversion(tk,tkl)# ;
   mainframe—parameter:
      MAINFRAME equal mainframe—letter
         #setmainframe(tk,tkl)# ;
         mainframe—letter:
            A | B | C ;
   system—parameter:
      SYSTEM equal system—name #setsystem(tk,tkl)# ;
         system—name:
            '6400' | '6500' | '720' | '730' ;
   parameter—list:
      parameter {separator parameter} ;
   parameter:
      version—parameter |
      mainframe—parameter |
      system—parameter ;
   control—statement:
      bol alpha [[separator] parameter—list|
         terminator ;

root
   control—statement ;
```

Some examples of valid calls to SLAVE are:

```
SLAVE.
SLAVE(VER=34, SYS=720)
SLAVE MAINFRAME=A.
```

Note that spaces may freely be inserted before and after terminal symbols.

5.1.2 The SLAVE program text

The complete source text of the program is listed in the appendix. Not shown are the POST modules and the code produced by the parser generator. The compiler will replace the include file directives:

```
(*$I'<section>'/'<file>' *)
```

by the specified source text.

The **value** statement [STR79] at line 32 is used to set up the syntax tables at load time, which would otherwise have to be done at run time. The **value** statements are produced by an auxiliary program, that converts the output from the parser generator.

The procedures *SETVERSION, SETMAINFRAME* and *SETSYSTEM* perform the semantic actions, required when the corresponding parameters have been recognized by the parser. The latter is called from line 160, to parse the program call statement. The root of the syntax is passed via the (parser generated) constants *RCONTROLS* and *NCONTROLS*

5.2 AUDIT

With the NOS/BE package, CDC provides the program AUDIT to list entries in the file directory. This program is written in FORTRAN (479 lines of code) and assembler for the central processor (another 720 lines of code). It also uses a specialized peripheral processor program EPF (2317 lines of assembler code). Altogether, there are 3516 lines of program text (comment lines are not included). Over the years many local modifications (400 lines of text) had crept into the program, in order to try to satisfy our special needs.

The conversion of the operating system from SCOPE 3.4 to NOS/BE 1.3 offered us the opportunity to reconsider the situation. It was decided to write our own AUDIT program in Pascal, with the help of the parser generator and using the POST library. The newly written program consists of 2300 lines of Pascal and 200 lines of syntax.

The most important features of the CERN AUDIT as seen by the user of the program are:

- The list of files is presented in alphabetical order.
- More powerful selection criteria may be specified. For example *PF LIKE TEST* will cause only the files to be listed, whose names begin with *TEST*. If *ID < PIETER* is specified, all owner identifiers lexicographically greater than or equal to *PIETER* will be ignored.
- Four listing formats may be selected, which differ in the amount of detail produced for the individual files.
- A secondary output file may be created with the relevant information formatted in the following ways:
  - As binary PFC (Permanent File Catalog) entries.
  - As DUMPF/LOADPF input directives.
  - As CCL (Cyber Control Language) procedure calls to any procedure supplied by the user.
- The (interactive) user may request to be prompted for every file whether to attach, purge, discard or display it.

The performance of the redesigned AUDIT compares favourably to that of the manufacturers program. The data in the table below refer to the listing of a set of files of a typical user at CERN.

	CDC's AUDIT	CERN's AUDIT
Execution field length (words)	*40000B*	*37400B*
CPU time used (seconds)	*0.531*	*2.138*
Real time used (seconds)	*24*	*7.5*

A complete rewrite may seem to be rather wasteful, both in terms of man power, and in terms of research and experience put into the manufacturers program. The contrary is however true for the following reasons:

- The redesigned program has all the required functionality.
- The old program had become messy, difficult to read and to modify. The new program is well structured, clean, easy to read, and if the need might arise, easy to modify.
- It is very frustrating for a programmer, to spend his time trying to understand badly written, spaghetti like programs. On the contrary it is challenging and rewarding to design a new program.
- The CERN permanent file control and archiving system [GEN81], which is a rather complex collection of interacting programs depends on the availability of a versatile AUDIT program. Had this system been made dependant on CDC's AUDIT, many auxiliary programs would have been necessary, to interface the control and archiving system to AUDIT. This in order to be relatively independant of changes in the program.
- Considering the increased functionality, much more effort would have had to be put into modifying the old program, than was required to write the new one.

More or less the same arguments could have been used to justify a rewrite in another high level language. The fact that Pascal was chosen allowed considerable time saving, both because of the features of the language and the availability of adequate support.

VI. Conclusions

The use of Pascal encourages structured programming. This makes programs easier to write, read and debug. It does not make them necessarily slower or less efficient than equivalent assembler or FORTRAN programs.

The strong typing of the language forces the programmer to describe his data objects precisely and allows the compiler to check thoroughly, that manipulations on these objects are valid.

The systems programmer who needs to manipulate absolute addresses may do so within the framework of the language. Tricks to achieve this should however be used with great care, since they will in general be dangerous, and violate the spirit of the language.

An (almost) complete model in Pascal has been constructed of the NOS/BE operating system. This model (the POST library) describes the interfaces to the system in terms of data structures and routines.

Updating a Pascal program for a new release of the operating system is almost painless, since the compiler will do the dirty work. The programmer merely has to modify the appropriate declarations in the POST library.

A parser generator may succesfully be used on a small scale. It offers the systems programmer the ability to make the interface to his product user friendly at very low cost.

Sophisticated system utility programs may be written relatively easily and quickly in Pascal. The available programming tools, notably the POST system interface library and the parser generator provide ample support.

References

[AHO78]
Aho, Alfred V.; Ullman, Jeffrey D.
Principles of compiler design
Addison-Wesley Publishing Company, Reading, Massachusetts, 1978

[BRO75]
Brooks Jr., Frederick P.
The mythical man-month, Essays on software engineering
Addison-Wesley Publishing Company, Reading, Massachusetts, 1975

[GEN81]
de Gennaro, M. Silvano
Permanent file archiving for NOS/BE at CERN
ECODU 31 conference proceedings, Helsinki, April 1981

[GRI71]
Gries, David
Compiler construction for digital computers
John Wiley & Sons, Inc., New York, 1971

[HOP69]
Hopcroft, John E.; Ullman, Jeffrey D.
Formal languages and their relation to automata
Addison-Wesley Publishing Company, Reading, Massachusetts, 1969

[JEN76]
Jensen, Kathleen; Wirth, Niklaus
Pascal user manual and report
Springer-Verlag, New York, 1978

[RIT75]
Ritchie, Dennis M.
C Reference manual
UNIX document, Bell Laboratories, Murray Hill, New Jersey, 1975

[RIT78]
Ritchie, Dennis M.
The UNIX time-sharing system
Comm. ACM 17, 7, 1978 pp. 365-375.

[STR79]
Strait, John P.; Mickel, Andrew B.; Easton, John T.
Pascal 6000 Release 3
University of Minnesota, Minneapolis, 1979

APPENDIX

```
(*$T+,P+,B1,E+,W0,R,A+,L'SYSTEM LABEL ALTER AND VERIFICATION PROGRAM.  (1981-SEP-11)' *)

PROGRAM SLAVE;


CONST
(*$I'CLIMITS'/'POST'                    - GENERAL CONSTANT DECLARATIONS *)
(*$I'CGEN'/'SYNTAX'                     - PARSER GENERATED CONSTANT DECLARATIONS *)
   MAXCCODE = 63;                    (* MAXIMUM DISPLAY CHARACTER CODE *)
   MAXSTR   = 80;                    (* MAXIMUM STRING LENGTH *)

TYPE
(*$I'TSYNTAX'/'POST'                    - SYNTAX TABLE DECLARATIONS *)
(*$I'TCHAR'/'POST'                      - ARRAYS OF 2,3,4 ..80 CHARACTERS *)
(*$I'TBIT'/'POST'                       - SUBRANGES OF 2,3,4 ..59 BITS *)
(*$I'TMIX'/'POST'                       - RECORDS SUCH AS LEFT7 *)
(*$I'TDATE'/'POST'                      - RECORDS WITH DATE AND TIME FORMATS *)
(*$I'TCMRPA'/'POST'                     - CENTRAL MEMORY POINTER AREA *)
(*$I'TRACOM'/'POST'                     - RA COMMUNICATION AREA *)
(*$I'TRCL'/'POST'                       - (NO) RECALL ENUMERATION TYPE *)
(*$I'TDFM'/'POST'                       - DAYFILE MESSAGE OPTIONS *)
(*$I'TACT'/'POST'                       - ACT COMMUNICATION AREA *)
(*$I'TMSG'/'POST'                       - MSG COMMUNICATION AREA *)
   STRING   = PACKED ARRAY _1 .. MAXSTR' OF CHAR;

VAR
(*$I'VSYNTAX'/'POST'                    - SYNTAX TABLE VARIABLES *)
   P:          CMRPAREA;              (* COPY OF THE CMR POINTER AREA *)
   SYSLAB:     PASLAB;                (* SYSTEM LABEL UNDER CONSTRUCTION *)
   DAYFILE:    TEXT;                  (* DUMMY FILE FOR DAYFILE MESSAGES *)

VALUE
(*$I'VALCDC'/'SYNTAX'                   - VALUE STATEMENTS TO SET UP SYNTAX TABLES *)


(* MONITOR REQUEST PROCEDURE WITH ONE PARAMETER *)
PROCEDURE SYSTEM1 (PP: CHAR3; R: RCLFLGVAL; VAR PAR: INTEGER); EXTERN;
(* MONITOR REQUEST PROCEDURE WITH TWO PARAMETERS *)
PROCEDURE SYSTEM2 (PP: CHAR3; R: RCLFLGVAL; VAR PAR: INTEGER; P2: INTEGER); EXTERN;
(* RETURN THE ADDRESS OF AN INTEGER VARIABLE *)
FUNCTION  ADDRESS (VAR X: INTEGER): INTEGER; EXTERN;
(* ISSUE THE STRING WRITTEN OUT ON FILE AS A DAYFILE MESSAGE *)
PROCEDURE FLUSHMSG (VAR DAYFILE: TEXT; MSGDEST: MSGFCVAL); EXTERN;


(*$I'FINTMIN'/'POST'                     - FUNCTION TO RETURN THE MINIMUM OF TWO INTEGERS *)
(*$I'FINTDIG'/'POST'                     - CONVERT ONE DIGIT NUMBER TO A CHARACTER *)
(*$I'FWEEKDA'/'POST'                     - COMPUTE DAY OF THE WEEK FROM JULIAN DAY *)
(*$I'PDATJUL'/'POST'                     - COMPUTE TOTAL DAYS IN PREVIOUS MONTH FROM DATE *)
(*$I'PDATREG'/'POST'                     - COMPUTE DAY AND MONTH FROM JULIAN DATE AND YEAR *)
(*$I'PFLIPST'/'POST'                     - FLIP A STRING *)
(*$I'PINTSTR'/'POST'                     - CONVERT AN INTEGER TO A STRING *)
(*$I'PDAYSTR'/'POST'                     - RETURN NAME OF A DAY AS A STRING *)
(*$I'PDTSTR'/'POST'                      - PACK THREE TWO DIGIT NUMBERS INTO A STRING *)
(*$I'PCONSTR'/'POST'                     - FETCH THE CONTROL STATEMENT FROM RA+70 ..RA+77 *)
(*$I'PPERRCL'/'POST'                     - ENTER PERIODIC RECALL (WAIT) *)
(*$I'PPAUSE'/'POST'                      - PAUSE FOR OPERATOR ACTION *)
(*$I'PGETCMR'/'POST'                     - READ A BLOCK OF CENTRAL MEMORY *)
(*$I'PPUTCMR'/'POST'                     - WRITE A BLOCK OF CENTRAL MEMORY *)


PROCEDURE SETVERSION (TOKEN: STRING; LENGTH: INTEGER);
```

```
(*****************************************************************
   PROCEDURE TO INSERT THE VERSION IN THE NEW LABEL.
*****************************************************************)

VAR
   I:          INTEGER;
BEGIN
   WITH SYSLAB DO
      BEGIN
         OPSYVER := 'V  ';
         FOR I := 1 TO INTMIN (LENGTH, 2) DO
            OPSYVER _I + 1' := TOKEN _I'
      END
END;  (* SETVERSION *)


PROCEDURE SETMAINFRAME (TOKEN: STRING; LENGTH: INTEGER);

(*****************************************************************
   PROCEDURE TO INSERT THE MAINFRAME IDENTIFIER IN THE NEW LABEL.
*****************************************************************)

BEGIN
   WITH SYSLAB DO

      BEGIN
         MAINFRAME := 'MF ';
         IF LENGTH = 1 THEN
            MAINFRAME _3' := TOKEN _1';
      END
END;  (* SETMAINFRAME *)


PROCEDURE SETSYSTEM (TOKEN: STRING; LENGTH: INTEGER);

(*****************************************************************
   PROCEDURE TO INSERT THE SYSTEM NAME IN THE NEW LABEL.
*****************************************************************)

VAR
   I:          INTEGER;
BEGIN
   WITH SYSLAB DO
      BEGIN
         SYSTEM := '    ';
         FOR I := 1 TO INTMIN (LENGTH, 4) DO
            SYSTEM _I' := TOKEN _I';
      END
END;  (* SETSYSTEM *)

(*$I'PGENEXE'/'SYNTAX'                  - PARSER GENERATED PROCEDURE WITH ACTION CALL *)
(*$I'PSYNERR'/'POST'                    - PROCEDURE TO HANDLE SYNTAX ERRORS *)
(*$I'PSYNPAR'/'POST'                    - PARSER AND LEXICAL ANALYSER PROCEDURE *)


PROCEDURE INITIALISE;

(*****************************************************************
   PROCEDURE TO INITIALISE THE GLOBAL VARIABLES.
*****************************************************************)

VAR
   I:          INTEGER;
BEGIN
   GETCMR (0, PALIM + 1, ADDRESS (P.TAG));  (* FETCH COPY OF POINTER AREA *)
   WITH SYSLAB DO  (* PRESET THE NEW SYSTEM LABEL *)
      BEGIN
         FOR I := 1 TO PASLABLIM DO
            IMAGE _I' := ' ';
         SITEOPSY := 'CERN NOS/BE 1.3  499';
         OPSYVER := P.SLAB.OPSYVER;
         MAINFRAME := P.SLAB.MAINFRAME;
         SYSTEM := P.SLAB.SYSTEM
      END;
END;  (* INITIALISE *)
```

```
PROCEDURE JCLSTATEMENT;

(*****************************************************************
   PROCEDURE TO PROCESS THE CONTROLCARD.
*****************************************************************)

VAR
   STR:       STRING;
   LEN:       INTEGER;
BEGIN
   CONTROLSTRING (STR, LEN);     (* FETCH THE CONTROL STATEMENT IMAGE *)
   PARSE (RCONTROLC, NCONTROLC, STR, LEN, TRUE)    (* AND PARSE IT *)
END;  (* JCLSTATEMENT *)


PROCEDURE SYSTEMVERSION;

(*****************************************************************
   PROCEDURE TO COMPARE THE VERSION FROM THE CURRENT  SYSTEM  LABEL
TO THE ONE FROM THE NEW SYSTEM LABEL, WHICH WAS COPIED FROM THE
CONTROL STATEMENT.
*****************************************************************)

BEGIN
   IF P.SLAB.OPSYVER <> SYSLAB.OPSYVER THEN
      BEGIN
         WRITE (DAYFILE, '$SYSTEM VERSION ', SYSLAB.OPSYVER,
            ' INCOMPATIBLE WITH   CMR VERSION ', P.SLAB.OPSYVER,
            'TYPE GO TO IGNORE':25);
         FLUSHMSG (DAYFILE, MSGCB);    (* SEND THE MESSAGE TO THE CONSOLE *)
         PAUSE                         (* WAIT FOR REACTION *)
      END
END;  (* SYSTEMVERSION *)


PROCEDURE DEADSTART;

(*****************************************************************
   PROCEDURE TO INSERT THE DEADSTART TYPE IN THE SYSTEM LABEL.
*****************************************************************)

BEGIN
   WITH SYSLAB DO
      BEGIN
         CASE P.DSFLAG.SYSLVL OF  (* FIND OUT THE DEADSTART LEVEL *)
            PADSA: DSTYPE := 'A';
            PADSB: DSTYPE := 'B';
            PADSC: DSTYPE := 'C';
            PADSD: DSTYPE := 'D'
            END  (* CASE *)
      END
END;  (* DEADSTART *)
```

```
PROCEDURE DATETIME;

(*****************************************************************
   PROCEDURE TO CHECK AND INSERT THE CURRENT DATE AND TIME  IN  THE
SYSTEM LABEL.
*****************************************************************)

VAR
   I:          INTEGER;
   OK:         BOOLEAN;            (* TRUE IF DATE AND TIME OK *)
   STR:        STRING;
   LEN:        INTEGER;
BEGIN
   WITH P.BJDT, JDATE, TIME DO  (* FIND DAY AND TIME *)
      REPEAT  (* UNTIL THE DATE AND TIME AR OK *)
         DATUMJULIANUM (12, YEAR, I);  (* DAYS IN YEAR *)
         IF (YEAR < 80) OR (YEAR > 90) OR
            (JDAY < 1 ) OR (JDAY > I ) OR
            (HOUR > 23) OR
            (MINUTE > 59) OR
            (SECOND > 59) THEN
            BEGIN
               WRITE (DAYFILE, '$ERROR IN SYSTEM DATE ', P.DATE,
                  '        OR TIME ', P.CLK, 'PLEASE CORRECT':22);
               FLUSHMSG (DAYFILE, MSGCB);
               PERIODICRCL (MAXRCLPER * 10);  (* WAIT 5 SEC. *)
               GETCMR (0, PALIM + 1, ADDRESS (P.TAG));  (* RE-READ POINTER AREA *)
            END
         ELSE
            OK := TRUE
      UNTIL OK;
   WITH P.BJDT.JDATE DO
      DAYSTRING (WEEKDAY (JDAY, YEAR), STR, LEN);
   FOR I := 1 TO 3 DO
      SYSLAB.DSDAY _I' := STR _I';
   WITH P.BJDT.TIME DO
      DTSTRING (HOUR, MINUTE, SECOND, '.', STR, LEN);
   FOR I := 1 TO 5 DO
      SYSLAB.DSTIME _I' := STR _I'
END;  (* DATETIME *)


PROCEDURE REPLACE;

(*****************************************************************
   PROCEDURE TO REPLACE THE SYSTEM LABEL.
*****************************************************************)

VAR
   FWA:        INTEGER;            (* FWA OF LABEL *)
   LEN:        INTEGER;            (* NUMBER OF CM WORDS IN LABEL *)
BEGIN
   FWA := ADDRESS (P.SLAB.TAG) - ADDRESS (P.AAZ);
   LEN := PASLABLIM DIV MAXALFA;
   WRITE (DAYFILE, 'ADDRESS ', FWA:4 OCT, ' LENGTH ', LEN:4 OCT,
      ' ', SYSLAB.IMAGE);
   FLUSHMSG (DAYFILE, MSGSJCB);
   PUTCMR (FWA, LEN, ADDRESS (SYSLAB.TAG))
END;  (* REPLACE *)


BEGIN
   INITIALISE;
   JCLSTATEMENT;
   SYSTEMVERSION;
   DEADSTART;
   DATETIME;
   REPLACE
END.
```