29. Records: user-defined structures

The personnel record system developed in section 27 worked reasonably efficiently because we were able to pack all the required information about an individual into one integer location; one integer index sufficed to identify the individual, and a single array was all that was required to store the complete system. In practical situations of this kind, it is usually not possible to pack all the information into an integer, nor indeed into a number of integers; the method of passing an index and then accessing numerous arrays becomes inefficient in terms of program space and execution time, and there is considerable lack of clarity in the continual packing and unpacking of the relevant fields containing the data. This section describes the facilities in RTL/2 which enable the user to define his own structures and to manipulate them either as entities or by reference to their constituent parts. Clearly there are two aspects to this: the definition and the use; we shall treat these aspects in turn.

The only structure we have at present in RTL/2 is the array. This is characterised by the fact that each element is of the same mode. A user-defined structure will consist of a number of *components* of varying modes which are tied together under one name. The definition of a user-defined structure effectively defines this name as a mode like integer or byte. This is reflected in the use of the keyword MODE to introduce such a definition. Our earlier personnel system might have been set up using the following structure:

```
MODE PERSON(INT AGE, SALARY, BYTE SEX, LOCATION);
```

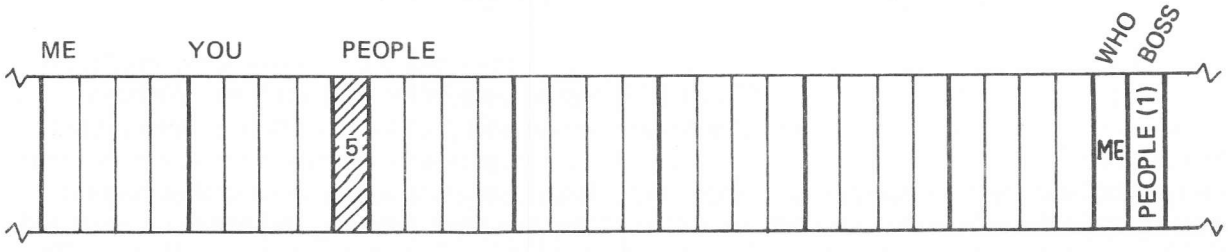
Such a definition occurs outside our bricks at the same level as LET definitions; the semi-colon separates it from other definitions and bricks.

We have created no structures by this definition; we have merely associated the name PERSON with a layout consisting of two integers and two bytes, that is we have a template. Every time we use the name PERSON we shall mean a 'lump' consisting of two integers and two bytes. Our definition goes further than this; it also defines four other names which can be used to select the component parts. Each field in the structure thus has a separate name. The syntax can be deduced from the example: the keyword MODE is followed by the name being defined and a list of components enclosed in brackets. The list of components is similar to the declaration of parameters – mode descriptions followed by a list of names all separated by commas – but remember that we have not actually declared anything yet. We shall return to the permissible forms of the components later.

Having defined the shape of our structure, and associated a name with it, we can now investigate how to declare specific examples of it. Actual occurrences (in the sense of creating space in the computer) are called *records.* Records, like arrays, are static structures and hence may only be declared in data bricks. The form of the declaration is quite standard: the mode followed by a list of names being declared. However, instead of the mode being a keyword, like INT, it is the name defined in the MODE definition. We can also declare arrays (multi-dimensional if we wish) of records and ref-record variables which naturally can contain the names of actual records of the appropriate mode. Remembering that reference variables must be initialised, we can write:

```
DATA RECORDEXAMPLES;
   PERSON ME,YOU;
   ARRAY (5) PERSON PEOPLE;
   REF PERSON WHO:=ME,BOSS:=PEOPLE(1);
ENDDATA;
```

which will create the following storage layout:



Each actual record consists of four separate fields.

We must now investigate how we name the individual variables involved.

Each component of each record behaves as a variable of the mode specified in the MODE definition and once we have learned how to name such a variable we can, of course, use it in precisely tha same way as any other variable, using all the constructions we have learned so far in this manual. The scope of the names of a MODE and its components is global; the names behave similarly to those of a data brick and its variables as far as availability in blocks and re-definition are concerned, though their uses are quite different.

In naming a particular element of an array, we append a subscript to the name of the array. In the case of records we append the name of the field to the name of the actual record; in this way, the names of components in the MODE definition are used as *selectors* for the various fields. Syntactically, we write the name of the record, a point (.), and then the selector name.

```
ME.SEX:='M';

YOU.AGE:=YOU.AGE + 1;    % BIRTHDAY %
```

As described above, once you have selected a component it behaves precisely as a variable of the mode specified in the MODE definition, just as an element of an integer array behaves as an integer.

The syntax is of the form 'record name.selector name', and hence when we want to select a field from an array of records we write:

```
PEOPLE(3).SALARY := PEOPLE(3).SALARY + RISE;
```

The name of the record is given by an array element (formed in the usual way); a field is then selected as above.

Again, as in the array case, selecting a field using a ref-record variable will result in automatic dereferencing of the ref-record variable to yield the name of an actual record from which a field can be selected:

```
WHO.LOCATION := 3;          %WHO DEREFERENCED%
```

This dereferencing is important. Records are static structures, and hence the only forms which we can declare locally (particularly in declaring parameters) are ref-record variables (assignment to these is straightforward and similar to assignment to a ref-array variable). In this way, we can pass a reference to the whole structure as a parameter; within the procedure, there is no need to access many arrays using an index passed as parameter; all we have to do is select the appropriate component using the ref-record variable.

```
PROC UPDATE (REF PERSON WHO );
    .
    .
    WHO.SALARY:=
    .
    .
ENDPROC;

% A CALL OF THIS WILL APPEAR AS %
    UPDATE(PEOPLE(2));
```

No 'whole record' operations are allowed (just as in the array case); hence there is no ambiguity in an assignment of the form WHO := PEOPLE(2) and no possibility of using VAL. We have pointed out many similarities between records and arrays and the key to both is understanding the formation of a name of a variable. Records have an advantage over arrays in that the required component field can be calculated at compile-time (like a constant subscript) and that there is never (even in the ref-record case) any need to perform run-time checking and hence no overhead; all fields are defined by the MODE definition and all checking is carried out at compile-time. The record form may also be more legible.

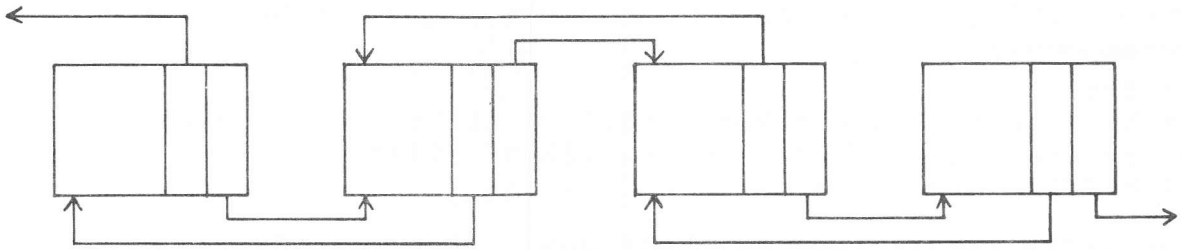
Having seen the definition of a structure using a MODE definition and how to declare actual records and manipulate both them and their components, we must now consider what modes are permissible for the components. The only things that are not allowed as components are records and arrays of records; any other modes or ref-modes (including ref-records) or arrays of them are allowed.

```
MODE ARITH (BYTE B, INT I, FRAC F, REAL R);
MODE STRUCT (ARRAY (6) BYTE NAME, REF INT RI1,RI2,
             ARRAY (4) REF FRAC RF, REF STRUCT RS,
             REF ARRAY ARITH RAA );
% NOTE THAT STRUCT CONTAINS A REFERENCE TO A STRUCTURE %
% OF ITS OWN KIND. SUCH RECURSION AND MUTUAL RECURSION %
% ARE QUITE PERMISSIBLE AS LONG AS ONLY REFERENCES ARE %
% INVOLVED %
% THE FOLLOWING IS ILLEGAL %

MODE WRONG ( INT ALPHA,BETA, ARITH WA, WRONG NOTRIGHT );
```

Components whose modes are references to structures can be used to set up a list structure in which records are chained together:

MODE CHAIN(...other components... REF CHAIN BACK, FORE); enables CHAIN structures to be linked (but not automatically) in the form:



If we have an actual CHAIN named Q, Q.BACK is the name of a reference to a CHAIN variable; we can therefore select a field from the record name which it contains by the usual method: Q.BACK.FORE; the appending of the selector will force Q.BACK to be dereferenced and the FORE field of that record is now named; this is another ref-CHAIN variable and if we assume that the linked list pictured in our diagram has been set up, it will, of course, contain the name Q. In the case of arrays and references to arrays (particularly the multi-dimensional case) we added subscripts to get down to the correct level or ultimate variables; similarly, for records we select fields until the required variable is obtained.

The similarity with arrays is continued when we consider the initialisation of records upon declaration in a data brick. (Note here that if a MODE definition includes components which are references, all actual records of this mode must be initialised to ensure that such components are safely initialised). The initial value for a record consists of a list of 'constants' separated by commas and enclosed in brackets. Whereas in the case of arrays all the 'constants' were of the same mode, the 'constants' in a record must be of appropriate modes to match the selectors specified in the MODE definition. Repetition factors are not permitted within a record initialisation, in the sense of repeating a value for a number of components; the initialisation of an array component may contain repetition factors in the normal way; and, of course, a record may have a repetition factor attached to it in the initialisation of an array of records — each bracketed record initialisation behaves as a constant at the lowest level of the array.

```
MODE AM ( BYTE AB, ARRAY (4) BYTE ANAME, REF A AA );
MODE BM (INT BI, REF ARRAY BYTE BB, REF INT BRI );

DATA INITIALISEDRECORDS;
   AM ACTUALA:= ('A',"FOUR",ACTUALA),   % REFERS TO ITSELF %
      A2:=A3:=('9',(0,1,2,3),ACTUALA);
   BM ACB:=(2937,A2.ANAME,ACB.BI);
   ARRAY (5) AM ARAM := (
      (0,"#0(4)#",A2),
      ('Q',"NULL",A3)(3),   % REPETITION FACTOR FOR RECORD %
      ('A',(1,2,3(2)),ARAM(1))
         );   % OUTER BRACKET OF ARRAY %
   ARRAY (2) BM ARBM:=(
      (27,ARAM(3).ANAME,ACB.BI),
      (0,"NULL",ARBM(1).BI)   );
ENDDATA;
```

We now give a simple example in which records are chained together. Basically we are adding words (contained in a byte array and padded to the right with spaces to make them all the same length) sequentially to a pool and chaining them together in alphabetical order. Advantage is taken of the alpha-characters being in ascending order in the ISO7 character set and the fact that ' ' < 'A'. Having forward and backward pointers allows us to remove words from the ordered list, but not from the pool. The first version uses crude coding to show constructions of names of the form LIST(I).NAME(J); the second economises by using inner blocks and intermediate reference variables.

```
LET SP=' ';
LET NOCHAR=15;   % MAXIMUM NUMBER OF LETTERS IN WORD %
LET MAXNO=10000;   % MAXIMUM NUMBER OF WORDS %
LET MAXNO2=9998;

MODE ENTRY (ARRAY (NOCHAR) BYTE WORD, REF ENTRY BACK,FORE);
   % BACK REFERENCES PREVIOUS, FORE THE NEXT ENTRY IN ORDER %

DATA LEXICON;
   ARRAY (MAXNO) ENTRY LIST:=(
      ("#SP(NOCHAR)#",LIST(1),LIST(2)),   % FIRST ENTRY = DUMMY %
      ("#'Z'(NOCHAR)#",LIST(1),LIST(2)),   % LAST ENTRY = DUMMY %
      ("#0(NOCHAR)#",LIST(1),LIST(1))  % PAD % (MAXNO2)   );
   INT NOIN:=2;   % NUMBER CURRENTLY IN POOL %
ENDDATA;

PROC INSERT (REF ARRAY BYTE X) REF ENTRY;
% RETURNS POSITION IN POOL OF WORD IN X, ASSUMED TO BE OF  %
% LENGTH NOCHAR AND RIGHT PADDED WITH SPACES. IF IT IS NOT IN %
% THE POOL IT IS INSERTED AND THE NECESSARY ADJUSTMENTS    %
% MADE TO THE ALPHABETICAL ORDER %
REF ENTRY NEXT:=LIST(2),   % END OF CHAIN %
          Q:=LIST(1);
   TO NOIN DO   % WE KNOW NUMBER IN THE LIST %
      Q:=Q.FORE;
      FOR J:=1 TO NOCHAR DO
         IF Q.WORD(J) < X(J) THEN GOTO NOGO;  END;
         IF Q.WORD(J) = X(J) THEN
            IF J=NOCHAR THEN
               % ALREADY IN POOL %
               RETURN(Q);
            END;
```

```
            GOTO NEXTCHAR;
         END;
         % CORRECT POSITION FOUND - NEXT WILL NOW CONTAIN  %
         % A REFERENCE TO THE WORD ALPHABETICALLY AFTER X  %
         NEXT:=Q;
         GOTO HOLE;
NEXTCHAR:
      REP;
NOGO:
   REP;
% NOTE HERE THAT NEXT CONTAINS LIST(2) - WE ARE ADDING X %
% TO THE END OF THE CHAIN OF ACTUAL WORDS %
HOLE:
   NOIN:=NOIN+1;
   IF NOIN>MAXNO THEN
      % NO ROOM LEFT IN POOL %
      % OUTPUT SUITABLE MESSAGE AND TAKE ACTION %
      RETURN(LIST(1));     % SAFE RESULT %
   END;
   FOR I:=1 TO NOCHAR DO
      LIST (NOIN).WORD(I):=X(I);
      % FILL IN WORD %
   REP;
   % NOW UPDATE CHAIN %
   LIST(NOIN).FORE:=NEXT;
   LIST(NOIN).BACK:=NEXT.BACK;
   NEXT.BACK.FORE:=LIST(NOIN);
   NEXT.BACK:=LIST(NOIN);
   RETURN(LIST(NOIN));
ENDPROC;

PROC ALPHAPRINT ();
% PRINTS OUT THE LIST IN ALPHABETICAL ORDER  USING THE CHAIN %
% AND ASSUMING AN OUTPUT PROCEDURE  PROC TWRT(REF ARRAY BYTE X) %
REF ENTRY NEXT:=LIST(1).FORE;
L: IF NEXT:=:LIST(2) THEN RETURN;  END;
   % WE ONLY WISH TO PRINT THE ACTUAL WORD ENTRIES NOT THE %
   % DUMMY FIRST AND LAST ENTRIES %
   TWRT(NEXT.WORD);
   NEXT:=NEXT.FORE;
   GOTO L;
ENDPROC;



% REVISED VERSION %
LET SP=' ';
LET NOCHAR=15;   % MAXIMUM NUMBER OF LETTERS IN WORD %
LET MAXNO=10000;   % MAXIMUM NUMBER OF WORDS %

MODE ENTRY (ARRAY (NOCHAR) BYTE WORD, REF ENTRY BACK,FORE);
   % BACK REFERENCES PREVIOUS, FORE THE NEXT ENTRY IN ORDER %

DATA LEXICON;
   ARRAY (MAXNO) ENTRY LIST:=(("#0(NOCHAR)#",HEAD,TAIL)(MAXNO));
         % MUST BE INITIALISED AS CONTAINS REFERENCE %
   ENTRY HEAD:=("#SP(NOCHAR)#",HEAD,TAIL),
         TAIL:=("#'Z'(NOCHAR)#",HEAD,TAIL);
```

```
    % DUMMY ENTRIES FOR BEGINNING AND ENDING THE ORDERED LIST %
    % WE COULD HAVE MADE THESE THE FIRST TWO ELEMENTS OF %
    % LIST AGAIN, BUT IN THIS WAY LIST IS THE POOL %
    INT NOIN:=0;    % NUMBER CURRENTLY IN POOL %
ENDDATA;

PROC INSERT (REF ARRAY BYTE X) REF ENTRY;
% RETURNS POSITION IN POOL OF WORD IN X, ASSUMED TO BE OF  %
% LENGTH NOCHAR AND RIGHT PADDED WITH SPACES. IF IT IS NOT IN %
% THE POOL IT IS INSERTED AND THE NECESSARY ADJUSTMENTS    %
% MADE TO THE ALPHABETICAL ORDER %
REF ENTRY NEXT:=TAIL;    % END OF CHAIN %
BLOCK
    REF ENTRY Q:=HEAD.FORE;
    WHILE Q:#:TAIL DO
    BLOCK
        REF ARRAY BYTE CURWORD:=Q.WORD;
        FOR J:=1 TO NOCHAR DO
        BYTE CURW:=CURWORD(J),  CURX:=X(J);
            IF CURW < CURX THEN GOTO NOGO;  END;
            IF CURW = CURX THEN
                IF J=NOCHAR THEN
                    % ALREADY IN POOL %
                    RETURN(Q);
                END;
                GOTO NEXTCHAR;
            END;
            % CORRECT POSITION FOUND - NEXT WILL NOW CONTAIN  %
            % A REFERENCE TO THE WORD ALPHABETICALLY AFTER X  %
            NEXT:=Q;
            GOTO HOLE;
NEXTCHAR:
        REP;
    ENDBLOCK;
NOGO:
        Q:=Q.FORE;
    REP;
ENDBLOCK;
% NOTE HERE THAT NEXT CONTAINS TAIL - WE ARE ADDING X %
% TO THE END OF THE CHAIN OF ACTUAL WORDS %
HOLE:
    NOIN:=NOIN+1;
    IF NOIN>MAXNO THEN
        % NO ROOM LEFT IN POOL %
        % OUTPUT SUITABLE MESSAGE AND TAKE ACTION %
        RETURN(TAIL)          % SAFE RESULT %
    END;
    BLOCK
    REF ENTRY NEW:=LIST(NOIN);
        BLOCK
        REF ARRAY BYTE Q:=NEW.WORD;
            FOR I:=1 TO NOCHAR DO
                % FILL IN WORD %
                Q(I):=X(I);
            REP;
        ENDBLOCK;
        % NOW UPDATE CHAIN %
```

```
      NEW.FORE:=NEXT;
      NEW.BACK:=NEXT.BACK;
      NEXT.BACK.FORE:=NEW;
      NEXT.BACK:=NEW;
      RETURN(NEW);
   ENDBLOCK;
ENDPROC;

PROC ALPHAPRINT ();
% PRINTS OUT THE LIST IN ALPHABETICAL ORDER  USING THE CHAIN %
% AND ASSUMING AN OUTPUT PROCEDURE  PROC TWRT(REF ARRAY BYTE X) %
REF ENTRY NEXT:=HEAD.FORE;
   % WE ONLY WISH TO PRINT THE ACTUAL WORD ENTRIES NOT THE %
   % DUMMY FIRST AND LAST ENTRIES %
   WHILE NEXT:#:TAIL DO
      TWRT(NEXT.WORD);
      NEXT:=NEXT.FORE;
   REP;
ENDPROC;

PROC REMOVE (REF ARRAY BYTE X);
% ILLUSTRATES USE OF DOUBLE CHAIN TO REMOVE AN ITEM %
% IT DOES NOT REMOVE X FROM THE POOL - INDEED IF X IS NOT  %
% IN THE POOL IT ADDS IT TO IT BY THE CALL OF INSERT %
% WE COULD TRAP THIS CASE BY REMEMBERING NOIN BEFORE CALLING %
% INSERT AND COMPARING WITH THE VALUE OF NOIN AFTER THE CALL %
REF ENTRY Q:=INSERT(X);
 Q.BACK.FORE:=Q.FORE;
 Q.FORE.BACK:=Q.BACK;
ENDPROC;
```

Section 29 examples

1. Rewrite the earlier DDC example (section 10) using records.
2. Design suitable MODE definitions for the creation of family trees. Attempt to write procedure bricks to ascertain the family relationship between two people. The importance of this exercise is not the complexity you can achieve, but the design and manipulation of suitable structures.

30. Communication

We have had many instances in this manual when we refer to input-output (I/O) procedures or procedures provided by the system; for example TWRT and DELAY. In section 8 we described RTL/2 as a procedure orientated language and indicated that I/O, system and real-time facilities are supplied in the form of procedures. Now clearly we do not wish each and every programmer to include such procedures in his programs! However, although we do not have to write the definition of such procedures, we must specify in some way their names – every name must be declared or defined in RTL/2.

The basic unit of our programs so far has been the brick. We have seen procedure and data bricks. We have also had things at inter-brick level – LET definitions, MODE definitions and, of course, comments can occur at this level. Bricks and inter-brick specifications or definitions can be grouped together to form a *module.* A module is the unit of compilation – it may contain just one brick or many. As the input to the RTL/2 compiler it must be self-contained in the sense that it contains no names which are not declared or defined. Various modules can then be linked together in some way to form a complete program or suite of programs. The method of communicating to the compiler the specifications of names used in one module which are defined in some other module (the two of which will eventually be linked together) is the main subject of this section.

Before exploring the problem, we look a little further at the advantages of having a modular structure. At a management level it means that various members of a team engaged on a large program complex can work independently whilst retaining good interfaces for the eventual linking together. At an individual level it provides the convenience of breaking the work into reasonably sized entities which can be compiled separately and tested separately.

Once a module has been successfully compiled and tested, and any further documentation completed, it can be "forgotten" until the rest of the complex is ready. Such a module can also be used in an "off-the-peg" manner and linked with some other suite if required. The communication of specifications between modules also provides the programmer with a natural way of documenting the interfaces between modules and between other programmers; it forms the "plugs" and "sockets" which enable the modules eventually to be linked together.

Procedures which we wish to call in a module but which are not defined in that module are termed *external procedures.* To specify the name of such a procedure we write the keyword EXT, a description of the form of the procedure and its name. Such a specification occurs at inter-brick level and, as usual, is separated from other inter-brick and brick information by a semi-colon. The description of the form of the procedure needs to record the nature of any parameters and any result. However, unlike the definition of a procedure – i.e. a procedure brick – the names of any parameters are quite immaterial and therefore omitted. The name of the external procedure follows this description; this allows a list of procedures with identical descriptions of course, to be appended. For the checks performed by the compiler (see section 8) such a description supplies all the necessary information. Thus our procedure TWRT would be specified as

```
EXT PROC (REF ARRAY BYTE) TWRT;
```

If there were more than one procedure we would write

```
EXT PROC (REF ARRAY BYTE) TWRT1, TWRT2, TWRT3;
```

This tells us and the compiler that TWRT, TWRT1, etc are procedures having a parameter of mode ref-array-byte, no result, and that their definitions are external to this module. As we have seen by implication, calls of external procedures are identical to calls of procedures defined in the current module; parameters to, and use of a result of, any call must match the specification in the same way.

Examples:

```
EXT PROC (REF ARRAY BYTE) TWRT;% OUTPUTS STRING %
EXT PROC (INT) INT IPRINT;      % OUTPUTS SIGNED INTEGER %
                                % RETURNS NUMBER OF CHARS. OUTPUT %
EXT PROC (INT,INT) IWRTF;       % FORMATTED INTEGER PRINT %
```

Procedures specified in external descriptions in various modules, must be defined in some module – on linking modules together we must have some code to execute on calling such a procedure. On defining a procedure which will be used in other modules, we must inform the compiler of

this intention (since it will need to create any necessary "plugs" or "sockets") by making the name an *entry* which may be specified externally elsewhere. This is achieved simply by preceding the definition by the keyword ENT. Thus a module to read a number of real numbers and print out their average could appear as:

```
% THIS MODULE READS AN INTEGER TO CONTROL THE NUMBER OF REALS %
% READ, AND PRINTS OUT THEIR AVERAGE %

LET NL=10;

EXT PROC (INT) IWRT;                    % OUTPUT INTEGER %
EXT PROC () INT IREAD;                  % READ IN INTEGER %
EXT PROC (REAL) RWRT;                   % OUTPUT REAL %
EXT PROC () REAL RREAD;                 % READ IN REAL %
EXT PROC (REF ARRAY BYTE) TWRT;         % OUTPUT TEXT %

ENT PROC PRINTAV ();
INT NUM:=IREAD();
REAL AV:=0.0;
   TO NUM DO
      AV:=AV + RREAD();
   REP;
   AV:=AV/NUM;
   TWRT("#NL#NUMBER OF REALS READ : ");
   IWRT(NUM);
   TWRT("#NL#AVERAGE VALUE IS : ");
   RWRT(AV);
   TWRT("#NL(2)#");
ENDPROC;
```

No input or output is defined as part of the RTL/2 language. For a given operating system supporting RTL/2 programs, I/O operations will be defined in terms of a set of external specifications. A standard set of procedures which should always be available in all RTL/2 systems has been defined, and details can be found in the manual "RTL/2 Standard Stream I/O". The use of such standards allows machine-independence of programs to be maintained. It is recommended, nevertheless, that I/O be gathered together in some convenient fashion so that modification for differing systems operation can be made simply and correctly. The basic method employed is *streaming* in which a sequence of single characters is read or written. The procedures assume that there is a method of transferring characters between a program and a channel – this channel being either a receiver or a sender. Usually a channel will be a physical device (e.g. teletype, paper-tape reader, line printer) but there is no reason why it should not be a simple internal array. The way in which stream changes may be made, and more primitive character handling are described in Section 32. It would be very tedious if all text had to be processed as single characters; various facilities are available for breaking down and building up sequences of individual characters. We therefore have a number of standard stream I/O procedures which read from or write to the current input or output stream (the various formats will be found in the stream I/O manual):

```
% INPUT PROCEDURES %
   EXT PROC () INT IREAD;               % INPUT INTEGER %
   EXT PROC () FRAC FREAD;              % INPUT FRACTION %
   EXT PROC () REAL RREAD;              % INPUT REAL %
   EXT PROC (REF ARRAY BYTE,REF ARRAY BYTE) INT TREAD;
                                        % READ IN TEXT TO SPECIFIED ARRAY %
                                        % SECOND ARRAY CONTAINS LIST OF %
                                        % TERMINATORS, RESULT IS NUMBER %
                                        % OF CHARACTERS READ %
```

```
% UNFORMATTED OUTPUT %
   EXT PROC (INT) IWRT;                    % OUTPUT INTEGER %
   EXT PROC (FRAC) FWRT;                   % OUTPUT FRACTION %
   EXT PROC (REAL) RWRT;                   % OUTPUT REAL %
   EXT PROC (REF ARRAY BYTE) TWRT;% OUTPUT TEXT %

% FORMATTED OUTPUT %
   EXT PROC (INT,INT) IWRTF;               % OUTPUT INTEGER %
   EXT PROC (FRAC,INT) FWRTF;              % OUTPUT FRACTION %
   EXT PROC (REAL,INT,INT) RWRTF; % OUTPUT REAL %

% MISCELLANEOUS %
   EXT PROC (INT) NLS,SPS;                 % OUTPUT NEWLINES, SPACES %
```

When we come to the systems and real-time functions, no such standards are available. For any given system, however, there will still be an interface defined in terms of brick specifications. It is likely that the basic facilities available will be similar between systems. The kind of functions we are now talking about, for example the procedure DELAY which gives us a time lag, are different in the sense that they may need to use hardware functions directly or use the interrupt structure of the machine or system. Some machines have a multi-state nature and such procedures may need a special linkage to invoke the primitive functions required. Such procedures are termed *supervisor calls* and their different nature is reflected in their specification by the use of the keyword SVC instead of EXT.

```
SVC PROC (INT,INT) INT STIM;
SVC PROC (REF INT,REF INT) REF ARRAY BYTE LISTEN;
SVC PROC (INT,INT) FAIL1,FAIL2;
```

There is no analogue to ENT for the SVC procedure. Whether it is possible to write the actual procedure in RTL/2 will depend upon the machine and the system. This will probably not concern the user, only the system writer. Details of the method of definition and the special linkages involved will be found in the documentation of specific systems.

Calls of SVC procedures are identical to any other procedures and, apart from a small difference discussed in section 32 and the fact that we write SVC instead of EXT, their use is the same. Specifying a procedure as SVC rather than EXT enables the compiler to generate different code on a call to take account of any special linkage mechanism.

Similar considerations arise with data bricks. The position is not quite so simple, and in any case much greater care must be taken in a multi-user situation because, unlike procedure bricks, data bricks are not read-only; several users updating the same data brick in a random real-time situation may lead to unexpected results. The use of data bricks for communication purposes in such situations must be planned with this in mind. Such considerations, however, do not affect the specifications. To take the simple case first, a data brick name may be made into an entry in precisely the same way, by the addition of ENT to the definition of the brick. For an external procedure we were only concerned with the name of that procedure and not with any parameter names (which are in any case local to the definition of that procedure). Variables within a data brick are global to the module in which the definition occurs, and their individual names are certainly required. An external data brick is obviously one whose variables we wish to use in this module but which is defined (and hence initialised and space created for it) in another module. Firstly, then, an external data brick cannot contain any initialisations (not even of reference variables – that these contain safe values is guaranteed in the definition of the brick, i.e. in the module containing its ENT definition). Secondly, we want the names of the variables; these must occur in the same order and should have the same names (although it may not always be possible to check this thoroughly at the linking stage). Syntactically an external data brick appears as an uninitialised data brick preceded by the keyword EXT. Variables specified in this way are global to the module. Note that the requirement to have external data bricks is a reason for having data bricks named – we do not use the name elsewhere.

```
MODE PERSON (INT AGE,SALARY, BYTE SEX,LOCATION),

ENT DATA PERSONNEL;
   INT NOOFEMP:=260,NOOFMEN:=232,TOTALSAL;
   ARRAY (500) PERSON STAFF:=( (0,0,0,0)(500) );
ENDDATA;

% THIS WILL BE SPECIFIED IN ANOTHER MODULE BY THE FOLLOWING : %

MODE PERSON (INT AGE,SALARY, BYTE SEX,LOCATION);

EXT DATA PERSONNEL;
   INT NOOFEMP,NOOFMEN,TOTALSAL;
   ARRAY (500) PERSON STAFF;
ENDDATA;
% ENDDATA STILL REQUIRED TO TERMINATE BRICK SPECIFICATION %
```

We also have SVC data bricks, but the keyword SVC is used merely for convenience; there is no concept of supervisory data or a call being involved. However, there is a similarity in that access to variables in such data bricks may be performed in a special way. SVC data bricks are supplied by the system and to the user are similar syntactically to external data bricks. Variables in an SVC data brick are global in scope, but the brick is "housekept" when a change of stack is made, for example when the program is interrupted by a higher priority activity; these variables are unique therefore to a run-time stack and may thus be used in a re-entrant manner by the program using that stack. It is likely that such bricks will be implemented by creating space for their variables in the run-time stack.

Example: standard input procedures need somewhere to place the character which terminated the reading of a number, and to store information about format errors. Clearly such information is private to the program calling the routines rather than to the routines themselves (of which only one copy exists, used in a re-entrant manner by many programs) and it is placed in an SVC data brick associated with the run-time stack of the calling program:

```
SVC DATA RRSED;
   BYTE TERMCH, IOFLAG;
ENDDATA;
```

Further examples will be seen in Section 32.

The run-time stack is a dynamic structure, and which stack will be associated with a program is not known at compile-time. Since the positions of variables in an SVC data brick are related to this run-time stack it is impossible to calculate their addresses during compilation. This means that it is impossible to use them as initialisations to reference variables in other data bricks.

Thus

```
SVC DATA RRSED;
   BYTE TERMCH, IOFLAG;
ENDDATA;

DATA LOCAL;
   REF BYTE RB:=IOFLAG;
ENDDATA;
```

is illegal.

It is however perfectly legitimate to write:

```
EXT DATA S;
   INT I;
ENDDATA;

ENT DATA GLOB;
   REF INT RI:=I;
ENDDATA;
```

For a large suite of programs we may wish to define all the external bricks available and include these specifications in every module in order to save preparation time and to reduce the chance fo error in continually copying the information. For a given external brick, there will be one module in the suite which actually contains the definition of that brick. It would defeat our purpose if we had to remove the specification from that module. Hence we allow the redundant specification of a brick as being external when the definition of that brick occurs in the same module. Naturally, though, the specification must match the definition of the brick which must be an entry! Redundancy in the sense of bricks not actually used in the module is also allowed, of course.

```
LET SP=' ';

EXT PROC (REF ARRAY BYTE) TWRT;
EXT PROC (INT) ACTION;

ENT DATA MESS;
   ARRAY (20) REF ARRAY BYTE MESSAGES:=(
         "NO GO",
         "STOP",
         "LOWER OFF",
         ""(16),    % SPARES %
         "ILLEGAL MESSAGE"
                        );
ENDDATA;

ENT PROC ACTION (INT X);
   IF X<1 OR X>20 THEN
      X:=20;
   END;
   TWRT(MESSAGES(X));
ENDPROC;

EXT DATA MESS;
   ARRAY (20) REF ARRAY BYTE MESSAGES;
ENDDATA;
```

The ordering of bricks and inter brick information is, in general, immaterial; the only definition that must occur before its use is the LET definition because of its textual replacement characteristics. People will develop their own orderings of information and bricks. Placing all the external specifications and definitions first makes the checking of cross-references easy, and the grouping of logically coherent bricks may aid understanding. Note however that the compiler may swap the ordering of bricks for efficiency within the machine, for example placing all read-only coding together. In general it is likely to be more efficient to have any data bricks preceding the procedure bricks.

Note also the problem of finiteness. Many systems impose a limit on the number of characters in names which survive to the linking stage, and it is wise to keep externally known names fairly

short, or at least significantly different in their leading characters since only these may be used. The compiler also places finite limits on the total number of various items (e.g. names, constants) it can accept and the overall size of a module. Moderation is sensible and good practice in the design of modules.

Two further forms of communication exist in RTL/2 and are now discussed.

To identify program text RTL/2 has a *title* item. This consists of the keyword TITLE followed by any sequence of characters not containing a semi-colon (which separates the item from the remainder of the text and effectively terminates it). Within the item, other keywords or items have no significance whatsoever. A title is a means of labelling all or part of a module. What happens to titles is implementation dependent, but the idea is that they can be used to label the object code and may be printed out for instance on compilation of the module and loading the object code into the machine. They should be thought of as comments which are "passed on" and not "thrown away" by the compiler. A title is an inter brick item whose position is immaterial, though it is most natural to use it as the heading of a logical group of bricks and other information.

Example:

```
   TITLE
   COMPOUND INTEREST PROBLEM
   J.SMITH   31/12/72 ;
% NOTE THAT NEWLINES ARE PERMISSIBLE %
```

To communicate with the compiler, an *option* item is provided in RTL/2. As with a number of topics in this and subsequent sections some of the details are implementation dependent and the appropriate manuals must be consulted. We give here the general syntax and the way in which it is intended to be used.

Syntactically, an option item consists of the keyword OPTION followed by an unsigned integer in brackets and a sequence of *opitems* separated by commas. The item is separated from the rest of the module (and hence effectively terminated) by a semi-colon. An opitem is any sequence of alphanumerics; which sequences have any significance and what that significance is, is implementation dependent. Typical opitems are BC signifying that a bound check on array element access is required and CM requesting explanatory material (not the same as the comments in the RTL/2 text) to be included in the object code produced by the compiler. Option items, again, are inter brick items; their position is significant however. From the point of occurrence of an option item, the options requested will be employed by the compiler.

Thus:

```
   OPTION(3) CM, BC;
```

informs the compiler that from this point onwards in the module, we wish to include a bound check on every array element access not checked at compile time, and to have the object code annotated. This option applies until the next option item or the end of the module is reached. Every option item thus replaces the previous one; any opitems not explicitly mentioned in the item are reset to a default value (defined by the implementation). We can think of an option as a command to the compiler to set a number of 'switches' to a desired combination.

What is the use of the integer in brackets? There will be some method when running an RTL/2 compiler to supply, via job-control language or teletype for instance, an option-like command. This option will then *for this compilation* replace all option items in the text having the same integer key.

Example

```
OPTION (1) XY,PQ;

DATA S;
    .
    .
ENDDATA;
OPTION (2) ;      % ALL DEFAULT VALUES TAKEN %

PROC P1 ();
    .
    .
ENDPROC;

OPTION (2) BC;

PROC P2 ();
    .
    .
ENDPROC;

OPTION (3) PQ;

PROC P3 ();
    .
    .
ENDPROC;
```

If we then supply OPTION(2) TR, BC; at compile-time, the module will be compiled as if the text were:

```
OPTION (1) XY,PQ;

DATA S;
    .
    .
ENDDATA;

OPTION (2) TR,BC;

PROC P1 ();
    .
    .
ENDPROC;

OPTION (2) TR,BC;

PROC P2 ();
    .
    .
ENDPROC;

OPTION (3) PQ;

PROC P3 ();
    .
    .
ENDPROC;
```

This facility allows temporary alteration to the module without actually modifying the text. The main use of this will be to compile modules with additional bound checking and diagnostic aids requested by compile-time options until they are fully tested; the module can then be re-compiled without compile-time options (when, of course only the options in the text will apply) in the form finally required without needing to alter the text. Note carefully though that only options with corresponding integer keys will be overridden by options supplied at compile-time.

The meaning of bound checking on access being optional, mentioned in section 14 should now be clear.

Titles and options are items as defined in section 3. However they are not quite the same as the other items which we have encountered — names, constants, comments, strings and separators. Titles and options are not permitted in LET definitions, there is no obvious reason for such a use so this is hardly a restriction; for this reason the other items are called *let-items* and hence the permissible sequence in a LET definition is a sequence of let-items. Similarly MODE and LET itself are not let-items.

Section 30 example

1. Write a module which will read in two real numbers separated by an arithmetic sign and print the result of performing this operation. Ignore problems of layout characters and format.

31. Stacks and systems

In all but the simplest real-time systems, there will be many pieces of program running "concurrently" which are unrelated particularly with respect to time scales and response-times. To the outside world, these various operations will appear to be in parallel; in fact, each will be given computer time on some time-sharing basis (e.g. by time-slice, by priority, by some scheduling algorithm). For example we might program a machine to scan and control a section of plant every second, print alarm and monitor messages as they arise, perform a sequence of operations with an irregular time structure, perform an optimisation calculation (not very time conscious, perhaps, but demanding considerable processor time) and print a log every hour. Here we have five distinct operations (though of course they may need to communicate with one another) with quite different requirements. Each operation defines a *task* – the dynamic life of a program in the machine. A procedure brick is (ultimately) a series of machine instructions defining some logical process, whereas a task is the sequential execution of those instructions. Why do we make the distinction? We mentioned in section 17 that the code of a procedure may be employed by several users simultaneously and for this reason the code is read-only and re-entrant. This is possible since each call results in a new incarnation of local variables and other information in a stack. Thus each concurrent task must have its own run-time stack. The life and progress of a task, the sequential calling of procedures, is the dynamic behaviour of its stack.

Such stacks may be defined in RTL/2. A stack is the third and final brick in RTL/2. As an entity it is simply a lump of memory, that is it consists of a set of locations in the store of the computer. The only property possessed by a stack is its length, that is the actual number of locations. In some machines it is convenient to measure this in bytes, in others in words; the units in which a stack is measured is therefore machine dependent. Syntactically a stack brick consists of the keyword STACK followed by a name and an integer representing the length of the stack being defined.

```
STACK WORKAREA 200;
```

As usual, this definition is separated from other bricks by a semi-colon. For the identifiers of stacks to be communicated between modules, EXT and ENT are used just as for procedure and data bricks and the same redundancy rules apply; when specifying an external stack we are only interested in declaring the name, and so no length is necessary.

```
ENT STACK MYSTACK 350;
EXT STACK WORKAREA;
```

The association of a stack with a task, and its initiation are system problems and are not discussed here; this information will be found in the relevant system manual.

Every time a procedure is called, the amount of stack in use expands. This calling process may occur many times (particularly in recursive situations) and a stack is of finite length. We cannot simply continue beyond the end of the stack because we may be corrupting other important locations. Hence there must be a check on each new procedure entry to ensure that there is enough space left in the stack to accommodate the new link cell, local variables and work area. If there is insufficient space, a run-time failure occurs; note that it is impossible for the compiler to calculate stack requirements exactly, since it does not know what sequence and nesting of procedure calls will actually occur. Details of how to estimate stack requirements, given a knowledge of probable dynamic behaviour, are given in system manuals.

We have already encountered a similar run-time check, that of verifying a subscript for an array element. How are such checks performed? Naturally there must exist some instructions in the machine to perform these actions and the compiler inserts the necessary commands to invoke these tests. These instructions form the basic environment for any RTL/2 system and are called *control routines.* The details of these are completely implementation specific and they are hand-coded in the relevant machine language.

Knowledge of the existence of run-time checks and the control routines is necessary for the understanding of the remainder of this section.

RTL/2 is designed in such a way that users' programs should be secure. In a process control situation and multi-task environment corruption of another program or its data area might prove disastrous. Array bound checking, stack checks on procedure entry, warnings where a variable name may be taken out of scope are examples of the way in which this security is achieved. However, one of the other aims of RTL/2 was to enable the majority of an operating

system to be written in a high-level language; for this use some of the restrictions and overheads are not acceptable. To cope with this, two forms of the language are defined; the complete language known as the *system language* and a more restricted subset, the *application language* which is the form we have been presenting so far.

In the system language, the programmer has far greater freedom, but also, of course, the responsibility for ensuring the safety of his actions. When presenting a module to the RTL/2 compiler the form of the language required must be specified; how this information is given to the compiler depends on the host machine on which the compiler is running.

How does the system language differ from the applications language as presented so far? The main differences are in the checks performed, either at compile-time or run-time.

1. All array bound checking is optional (except for the case of a constant subscript at the first level) whether we are accessing an element, storing into it or passing its name to a reference variable.

Action \ Language	Application	System
First subscript constant	Compile-time check	Compile-time check
Plain array access	Run-time check optional	Run-time check optional
Plain array storage/other array access or storage	Run-time check obligatory	Run-time check optional

2. Greater freedom in the use of reference variables. In the application language all reference variables must be initialised on declaration to ensure that they contain safe values. This restriction is lifted in the system language and the onus for sensible contents and use is on the programmer.
3. Allied to 2, the warning messages concerning the potential risk of taking a variable name out of scope are optional.
4. The stack check on procedure entry is optional. In some systems, where the check is a minimal overhead anyway, this option may not be available.

Wherever possible, it is recommended that the application form be used. In many situations where the system form is essential it is feasible and desirable to compile the module initially in the application form until it is fully tested.

The control routines, mentioned above, are not written in RTL/2. In writing systems programs, certain situations are impossible to handle in a high level language, for instance the handling of physical devices and hardware interrupts. Such routines must be written in the appropriate machine code and de facto will be highly machine dependent. A facility to include sections of machine code within procedure bricks is available in RTL/2. As one can write almost anything in machine code, no security can be achieved, and so its use is restricted to the system language. Use of this facility should be confined to areas impossible to code in RTL/2 or highly time-conscious; RTL/2 in these areas can still be used to advantage to document the process involved.

Being almost completely machine dependent, it is difficult to define the facility; as before, we shall attempt to convey the general philosophy, and refer the reader to the relevant implementation documents for the details.

A section of machine code is introduced by the keyword CODE. The section behaves as a statement, the *code-statement* (this is the remaining statement type mentioned in section 20). For syntactic purposes a *code sequence* is also classed as an item, but not a let-item.

Some controls are still retained within code sections. CODE must be followed by two decimal integers separated by a comma and terminated by a semi-colon. These values will be interpreted in a machine-dependent way, but are likely to be used to specify the number of locations occupied by the instructions and the number of working locations required on the stack. Naturally the units employed will also be machine-dependent. Within this "heading", layout characters may occur in the usual way. Immediately following the semi-colon we are in the code

sequence itself, and layout characters stand for themselves and are not removed : their significance depends on the particular machine code.

The statement itself is likely to consist of simple machine instructions. However, particular RTL/2 variables may be required within the sequence, and we do not wish to have to rewrite the code section every time we make a minor alteration elsewhere in the module. For this reason, there is a means of denoting RTL/2 let-items within a code sequence. Only allowing let-items is again no restriction, since sensible interpretations could not be put on other items.

For each implementation, two special characters (*trip* characters) are defined which are not often used for any other purpose in the machine code. Let us assume in the following that the particular characters chosen for some machine are * and /. To insert an RTL/2 let-item we write the item preceded by the first trip character (with no intervening layout characters) thus : *FRED, *27, *OCT 77, *"STRING". The trip characters themselves may be inserted in this way: **, */. What is actually inserted in the object code for a particular let-item is machine-dependent, and details will be found in the relevant implementation documents; for a variable it is likely to be its address or its displacement from the appropriate link cell on the stack.

In particular, keywords are let-items. RTL is another keyword and is used to terminate a code statement. However, as we are in a machine-dependent sequence (where RTL may have some other significance), it must be presented in the form of an RTL/2 let-item; i.e. it must be preceded by the first trip character.

```
CODE 4,0;
   MOV   #5,X0        ; ASSEMBLER COMMENT
   JMP   @*L          *% L IS LOCAL LABEL %
*RTL;
```

Note the use of a comment item : this differs from the machine language comment in that it disappears on compilation whereas the machine-code comment will be passed on as part of the sequence.

Names behave very much as they do outside code sections, in particular with respect to scope rules. Thus names defined by LET definitions will be replaced by the correct sequence of let-items. We can also set local labels within a code sequence by using the usual construction (plus appropriate trip characters) and jump into the statement using this name.

```
CODE
    .
    .
*RL*:     *% TWO RTL/2 MACROITEMS - TWO TRIP CHARACTERS %
    .
    .
*RTL;
.
.
GOTO RL;
.
```

To minimise the possible errors due to changes of definition, and allow the compiler to perform some checks within the section, a non-local variable name must be presented with the name of the data brick, or mode name in the case of a selector, to which it belongs. The syntax consists of appending the brick name preceded by the second trip character – this is the sole use of the second trip character.

```
DATA GLOB;
   INT I;
ENDDATA;

PROC MAIN ();
INT X:=OCT 160020;
       .
       .
       .
       CODE 6,0;
          MOV    @*X(5),*I/GLOB
       *RTL;
       .
       .
       .
ENDPROC;
```

Names of bricks and modes may be presented simply preceded by the first trip character. In the mode case, the interpretation is likely to be to substitute the space occupied by one record of this mode; for a brick, the interpretation will be more machine-dependent.

32. Non-plain modes

We have dealt completely with the four plain (arithmetic) modes in RTL/2 – real, integer, fraction and byte; we have also seen non-plain modes in the form of records and references to both variables and structures. We have been careful to stress the nature of certain other objects as being entities – a procedure as a process represented by a sequence of machine instructions, a stack as a lump of memory, a label as a particular point in a program. The only sensible way to talk about these entities is by using their RTL/2 identifiers, that is, their names. These names are quite different from the names of variables; the name of a variable identifies a location (or set of locations) whose contents can be changed. Names of procedures, stacks, labels relate to "constant" items – they are *literal* names used as a shorthand for the entity. This is why, throughout this manual we have used 'set', 'defined' for these names rather than 'declared'.

Three further modes (non-plain) are provided in RTL/2 to enable such literal names to be manipulated. There is nothing comparable for the (literal) names of data bricks, since there are no operations which we can perform with them.

The simplest of the three is the mode stack. We can declare variables (and arrays and components) of mode stack using the keyword STACK; such variables can contain the (literal) name of a stack brick. We can also declare ref-stack variables, which, naturally, can contain the names of stack variables (though a requirement for such a structure will be rare). Being non-plain, stack variables must be initialised on declaration in an applications program.

```
STACK JOB 150;      % DEFINES STACK BRICK AND LITERAL JOB %

DATA SYSTEM;
   STACK MYSTACK:=NEWSTACK:=JOB;
                    % DECLARES STACK VARIABLES %
   REF STACK WHICH:=MYSTACK;
ENDDATA;

% THIS IS OF THE SAME FORM AS THE FOLLOWING INTEGER CASE EXCEPT %
% THAT THE INTEGER 'LITERAL' OR CONSTANT DOES NOT HAVE TO BE %
% DECLARED OR DEFINED %

% LITERAL 3 DOES NOT NEED TO BE DEFINED %
DATA S;
   INT I:=3;
   REF INT RI:=I;
ENDDATA;
```

The use of stack variables will be confined mainly to systems programs, where the manipulation of stacks is required. We know how to declare and initialise such variables (note that they are not necessarily static – hence we can declare local stack variables) but what operations are available?

Assignment to stack variables is quite standard; the usual dereferencing rules being applied. In particular objects of mode stack can be passed through the parameter mechanism. Thus a procedure to send a message from the current task to another task (identified by its stack) with a byte result indicating success or failure, might take the form

```
PROC SEND(STACK RECEIVER, REF ARRAY BYTE TEXT)BYTE;
```

In practice this might appear as a supervisor call, and an actual call would be programmed as:

```
SVC PROC (STACK,REF ARRAY BYTE) BYTE SEND;
EXT STACK OTHERJOB,NEXTJOB;
      .
      .
      IF SEND(OTHERJOB,"STOP")#0 THEN ..  % MESSAGE NOT RECEIVED %
         % THIS IS AN EXAMPLE OF THE USE OF A SIDE-EFFECT %
         % THE MESSAGE IS SENT AS A SIDE-EFFECT OF TESTING WHETHER %
         % IT WORKED %
```

Apart from assignment, the only valid operations involving stacks are the comparisons of equality and inequality. Thus we can write

```
IF RECEIVER=OTHERJOB  % RECEIVER DEREFERENCED %  THEN ..
IF WHICH#MYSTACK  % WHICH DEREFERENCED TWICE, MYSTACK ONCE % THEN ..
IF WHICH:=:MYSTACK  % WHICH DEREFERENCED ONCE %  THEN ..
      % THIS COMPARISON IS BETWEEN THE NAMES OF STACK VARIABLES %
```

When we come to the case of procedures there are in fact many modes; the problem is similar to the one for specifying external procedures and stems from the fact that we must also define the nature of any parameters and result. The form of *descriptor* used is identical to that for an external specification, consisting of the keyword PROC, a list of the modes of any parameters separated by commas and enclosed in brackets and the mode of any result; as usual the brackets are still required when there are no parameters. As before, we do not need to include any names for parameters – in any case, these will effectively be different depending on the contents of such a procedure variable. Using these descriptors we can declare static and dynamic variables and arrays or components of mode procedure whose contents are the (literal) names of procedure bricks *of the same specification.* Being non-plain variables, they must be initialised in the application form of the language. Ref-procedure variables are also allowed, (again with the appropriate specification) but their use is likely to be unusual.

```
EXT PROC () BYTE CHARIN;
EXT PROC (BYTE) CHAROUT;

PROC P (INT I, REAL R);
      .
      .
ENDPROC;

PROC Q (INT K, REAL S);
      .
      .
ENDPROC;

DATA S;
   PROC () BYTE READ:=CHARIN;
   PROC (INT,REAL) DUMMY:=P;
ENDDATA;

PROC ALPHA ();
PROC (BYTE) MYOUT:=CHAROUT;
      .
      .
      DUMMY:=Q;
      .
      .
ENDPROC;
```

Assignment is completely standard, the procedure brick represented by the literal name being the form of the 'constant'. Comparisons using =, #, :=:, :#:, follow in precisely the same way as for stacks.

The existence of procedure variables allows procedures to be called indirectly. If in procedure ALPHA we write MYOUT ('A'); what do we mean? MYOUT is a procedure variable which cannot stand by itself; hence we dereference it to yield a literal procedure (in this case CHAROUT); the required byte parameter is supplied and CHAROUT will be called. In our DDC example we could provide *a* choice of control by this means; we need to add a procedure parameter to the procedure DDC. On calling DDC, we pass the (literal) name of the control algorithm we wish to use.

```
ENT PROC DDC (PROC () REAL CONTROL);
% SET CONSTANTS AND INPUT MEASUREMENT %
   NEWERR:=SETPOINT-MEASURED;
   CORRECTION:=CONTROL();    % DEREFERENCE CONTROL TO YIELD ALGORITHM %
   VERYOLDERR:=OLDERR;
   OLDERR:=NEWERR;
   % ETC %
ENDPROC;

ENT DATA SYSTEM;
   REAL MEASURED,SETPOINT,          % CURRENT VALUES %
      NEWERR,OLDERR,VERYOLDERR,     % LAST THREE ERROR TERMS %
      INTERVAL,                     % TIME INTERVAL %
      CORRECTION,                   % DELTA P - VALVE CHANGE %
      K,L,M;                        % CONSTANTS %
ENDDATA;
```

The calling module will have:

```
EXT DATA SYSTEM;
   REAL MEASURED,SETPOINT,          % CURRENT VALUES %
      NEWERR,OLDERR,VERYOLDERR,     % LAST THREE ERROR TERMS %
      INTERVAL,                     % TIME INTERVAL %
      CORRECTION,                   % DELTA P - VALVE CHANGE %
      K,L,M;                        % CONSTANTS %
ENDDATA;

EXT PROC (PROC() REAL) DDC;

PROC PROP () REAL;
   RETURN(K*(NEWERR-OLDERR));
ENDPROC;

PROC PROPINT () REAL;
   RETURN( PROP() + L*NEWERR*INTERVAL);
ENDPROC;

PROC PROPINTDERIV () REAL;
   RETURN( PROP() + PROPINT ()
      + M*(NEWERR-2*OLDERR+VERYOLDERR) / INTERVAL);
ENDPROC;

ENT PROC MAINTASK ();
   .
   .
   DDC(PROPINT);    % REQUIRED ALGORITHM PASSED %
   % NOTE THAT NO BRACKETS ARE REQUIRED HERE FOR PROPINT %
   % WE ARE ASSIGNING IT TO A PROCEDURE VARIABLE NOT CALLING IT %
   .
   .
ENDPROC;
```

When all the actions are similar and can be parameterised in the same way, we can use an array of procedure variables to perform the action of a switch.

```
M: SWITCH I OF L1,L2, ... ,L10;
L1:   ACTION1();
      GOTO L;

L2:   ACTION2();
      GOTO L;

          .
          .
L10:

% USING PROCEDURE VARIABLE %

DATA ACTIONS;
   ARRAY (10) PROC () NEXTACTION:=
      (ACTION1,ACTION2, ... ,ACTION10);
ENDDATA;
      .
      .
M: NEXTACTION(I)();
   % NEXTACTION SUBSCRIPTED GIVES A PROCEDURE VARIABLE %
   % DEREFERENCED TO GIVE A PROCEDURE - NULL PARAMETER %
   % LIST FOR CALL %
L:
      .
```

Note, however, that the action taken on an out-of-range subscript will be quite different.

When calling an SVC procedure, the compiler may need to insert a special linkage, for instance to take account of any multi-state nature of the machine. When calling a procedure indirectly through a procedure variable, the compiler will not have any knowledge about the contents of that variable, save the specification of its parameters and result. It is impossible therefore to decide to insert a special linkage. For this reason it is not legal to assign the literal name of an SVC procedure to a procedure variable. This is the difference between SVC and EXT procedures mentioned in section 30.

Example: Procedure variables are particularly useful when writing generalised routines. As a very simple illustration, this procedure brick with a procedure variable parameter tabulates the values of a given function f(x) in an array Y for a given set of values in array X.

```
PROC TABULATE (REF ARRAY REAL X,Y, PROC (REAL) REAL FUNC);
% X,Y ASSUMED TO BE OF THE SAME LENGTH %
% ALL X ELEMENTS ASSUMED TO BE RELEVANT %
      FOR I:=1 TO LENGTH X DO
         Y(I):=FUNC(X(I));
      REP;
ENDPROC;
```

Procedure variables lie at the heart of the recommended stream I/O mechanism. The basic character input and output procedures associated currently with a given task are held in procedure variables in a data brick; since this is unique to the task and must be re-entrant this is an SVC data brick:

```
SVC DATA RRSIO;
   PROC () BYTE IN;
   PROC (BYTE) OUT;
ENDDATA;
```

IN() will cause the next character to be removed from the input channel and returned as result; this is achieved by calling the brick whose name is in IN. Similarly OUT ('A') will send the character A to the output stream. Thus the module containing the text writing procedure could appear as:

```
TITLE
ROUTINE TO OUTPUT TEXT;

SVC DATA RRSIO;
   PROC () BYTE IN;
   PROC (BYTE) OUT;
ENDDATA;

ENT PROC TWRT (REF ARRAY BYTE TEXT);
   FOR I:=1 TO LENGTH TEXT DO
      OUT(TEXT(I));
   REP;
ENDPROC;
```

The user may use these basic routines directly, and may need to change the contents of the variables, although some systems will provide default streams and methods of changing channels. However the user can do it quite simply; to change the channel in use we merely have to change the contents of the appropriate procedure variable.

```
EXT PROC (BYTE) LPOUT;

SVC DATA RRSIO;
   PROC () BYTE IN;
   PROC (BYTE) OUT;
ENDDATA;

PROC P ();
      .
      .
      BLOCK
      PROC (BYTE) REMOUT:=OUT;    % PRESERVE DEFAULT PROCEDURE %
         OUT:=LPOUT;
         .
         .
         OUT:=REMOUT;    % RESTORE %
      ENDBLOCK;
      .
      .
ENDPROC;
```

The situation is a little more complex when we consider label objects. There is only one mode, and variables of the mode label are declared using the keyword (our last keyword) LABEL. Although the setting of a (literal) label defines a fixed point in our program text, and associates an identifier with that point, at the time when that name is assigned to a label variable, the point relates to the program text with respect to the particular incarnation of its procedure brick on some stack; in other words it marks a point in the sequential execution of a program, a point in some task. The named point itself is therefore ambiguous. The contents of a label variable therefore contains additional information to relate it to a particular link cell on a stack; this combination is known as a *level-address pair.*

As literal labels are local, they are only in scope within some block in a procedure brick. It is therefore impossible to initialise label variables in a data brick and illegal to attempt it. Label variables (including local ones) are an exception to our rule for the compulsory initialisation of

non-plain variables; it is still compulsory to initialise all ref-label variables since we can have global label variables which are valid contents. We shall see below that the potential danger of uninitialised label variables does not lead to any loss of security. It does pose one problem which is unsolved, but which is highly unlikely to occur in practice; what happens for an actual record of a mode which contains both label and other non-plain components? Non-plain components imply compulsory initialisation of the record; the presence of the label variable makes it impossible.

Having identified the problem areas, and by implication introduced the ideas of label and ref-label variables, we now investigate their syntax and major uses. Assignment, parameter mechanism, the use of the comparators =, #, :=:, :#: follow precisely the rules for stack and procedure variables; note, however, that literal stacks and procedures are global whereas literal labels are not – we must therefore ensure that labels are in scope when assigned.

```
PROC P ();
LABEL WHERE:=L;
    .
    .
L:
    .
    BLOCK
    LABEL SOMEWHERE:=L;    % L IN SCOPE %
        .
    M:
        .
    ENDBLOCK;
    .
    .
    WHERE:=M;    % ILLEGAL - M OUT OF SCOPE %
    .
ENDPROC;
```

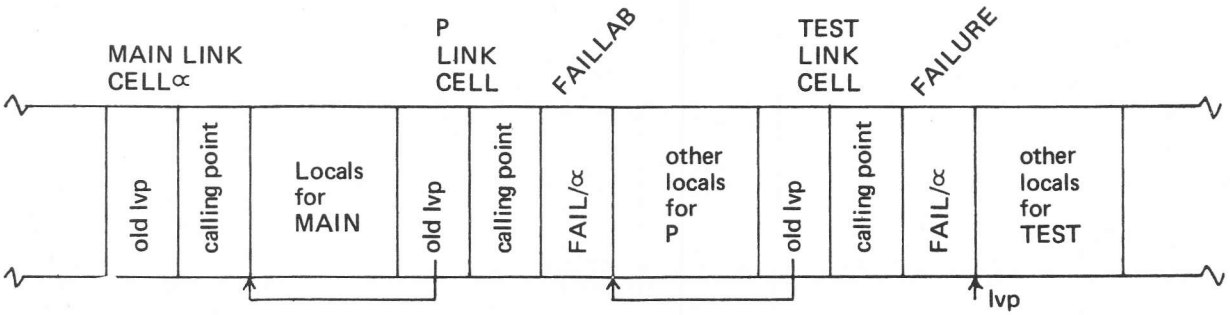
The main use of label variables (similar to procedure variables) is to enable indirect jumps to be performed, with the ultimate destination set dynamically. Following the usual philosophy on the use of names and dereferencing this is achieved by extending the goto-statement to permit the destination to be specified as a label variable or a ref-label variable which will be dereferenced once or twice respectively. However, on performing this transfer, we must ensure that the label is still valid; that is, that the link cell with which it is associated is still in existence. This check, the monitoring of *generalised goto-statements,* is performed by a control routine; the check is optional in the system language in the same way as the stack check. If the label is valid, the stack is unwound to the appropriate link cell, and processing continued at the correct point in the procedure marked by the label identifier. Thus the label variable (possibly as a parameter) gives the ability to transfer to a 'global' label. This will be most useful for the handling of error situations.

```
PROC P (LABEL FAILLAB);
    .
    TEST(FAILLAB);
    .
ENDPROC;

PROC TEST (LABEL FAILURE) INT;
    .
    IF ... THEN GOTO FAILURE;  END;
    .
ENDPROC;
```

```
PROC MAIN ();
      .
FAIL:     % ERROR ACTION %
          RETURN;
      .
      .
START;
   P(FAIL);
      .
ENDPROC;
```

At run time, MAIN calls P which calls TEST, and the failure label is passed on to each in turn. At this point, the stack will appear as follows:



We have indicated the appropriate link cell by writing the contents of the label variable as the pair FAIL/∝. What happens if, in TEST, we now obey the statement GOTO FAILURE? FAILURE will be dereferenced yielding the pair FAIL/∝ and the control routine will be entered. This will find that ∝ is still valid, the stack will be unwound (thus exiting from TEST and P) and processing will continue at the label FAIL in MAIN. Note that although TEST has a result we do not need to specify one at this point; we are not returning a result to the point of call in some expression, we are exiting to some point at which a new statement will be obeyed.

The following example illustrates the case where the test will fail; note also that if the variable L (uninitialised in the data brick GLOBAL) had not been assigned to, the test would still fail – hence our comment above that no loss of security is involved because of the impossibility of initialisation.

```
DATA GLOBAL;
   LABEL L;
ENDDATA;

PROC OOPS ();
      .
M:
      .
   L:=M;
   % THIS WILL BE WARNED AT COMPILE-TIME %
   % AS POTENTIALLY TAKING M OUT OF SCOPE %
      .
ENDPROC;

PROC MAIN ();
      .
   OOPS();
   GOTO L;
   % WILL FAIL AT RUN-TIME. THE CONTENTS OF L %
   % ( M ) NO LONGER IN SCOPE DYNAMICALLY %
      .
ENDPROC;
```

Because inner blocks have no entry overhead, and a link cell applies to the incarnation of the whole procedure, these tests can only be made at the procedure level. Hence the following will not fail at run-time, but the offending assignment to Q taking L out of scope will be flagged at compile-time.

```
    LABEL Q:=M;
M;
        .
        BLOCK
           Q:=L;
        L;
        ENDBLOCK;
        .
        GOTO Q;
           .
```

The situation is worse when ref-label variables are involved; care must be taken over scopes when using label variables.

What happens when the check of a generalised goto fails at run-time? Similarly what happens on any of the run-time failures we have discussed so far (stack check, array bound check)? Such a situation is termed an *unrecoverable error,* in the sense that it is unsafe for the program simply to continue, since it may corrupt other program areas. No definite action is specified in the RTL/2 language but a recommended standard approach is given in the manual. "RTL/2 System Standards". Within this standard, transfer is made to a label held in a label variable. The user has access to this variable and so may specify its contents and arrange for his own error recovery; otherwise the system will supply a default value where well-defined actions will be taken. As an unrecoverable error may occur at any time in a program, it is important that the error label is always dynamically in scope; a label at the outermost level should therefore be chosen. In the event of the label being out of scope, a second unrecoverable error will occur and the system action be adopted. An integer value is also provided which will be zero normally, but on an unrecoverable error will be set to indicate which error has occurred.

Other run-time errors which can only affect the current task and hence are not generally unsafe are termed *recoverable errors.* The recommended standard here is to call an error procedure (held in a procedure variable naturally) which has an integer parameter through which can be passed an indication of the error detected.

All of this error information is clearly private to a particular task, and hence is held in an SVC data brick.

```
SVC DATA RRERR;
   LABEL ERL;          % ERROR LABEL %
   INT ERN;            % ERROR NUMBER %
   PROC (INT) ERP;     % RECOVERABLE ERROR PROCEDURE %
ENDDATA;
```

The following example shows the use of this standard; note the good practice of remembering the system error label and resetting the error label at appropriate points.

```
SVC DATA RRERR;
   LABEL ERL;          % ERROR LABEL %
   INT ERN;            % ERROR NUMBER %
   PROC (INT) ERP;     % RECOVERABLE ERROR PROCEDURE %
ENDDATA;

EXT PROC (REF ARRAY BYTE) TWRT;
EXT PROC (INT) IWRT;
EXT PROC () CLOSEEVERYTHING;    % SHUT DOWN TASK %
```

```
PROC RRJOB ();    % RECOMMENDED STANDARD FOR JOB NAME %
LABEL REMERL:=ERL;    % REMEMBER SYSTEM ERROR LABEL %
      .
      .
   ERL:=L;    % SET PRIVATE ERROR LABEL %
         .
         % ANY ERROR HERE BRANCHES TO L %
         .
   ERL:=REMERL;    % RESET %
      .
      .
   RETURN;
L: ERL:=REMERL;    % RESET TO PREVENT LOOPING ON ERROR IN CLOSING %
   TWRT("UNRECOVERABLE ERROR ");
   IWRT(ERN);
   CLOSEEVERYTHING();
ENDPROC;
```

Again note the difference of action between an array of labels and a switch when the subscript is out of range; using an array allows us effectively to have non-local labels in our "switch" action. Note that the switch statement may only contain *literal* labels which are in scope, and not label variables. However, because of the run-time checks and the space occupied, using local literal labels in label variables is not very efficient, and so arrays of labels are not likely to be used widely.

Section 32 examples

1. Write a program to read in Roman numerals and print out the value in decimal.

33. Formal definition of RTL/2

The whole of the RTL/2 language has now been presented. This manual has been designed as a sequential text and is clearly unsuitable for use as a formal definition of the rules or as a reference work. A formal, logical presentation of the language is contained in the "RTL/2 Language Specification" manual. That definition is, of necessity, in a somewhat rarified form. This section is intended as an introduction to the use of the specification manual.

There is a philosophical difficulty in defining a language in terms of another language; we need some form of meta-language to enable us to define it rigorously. The specification manual uses a modified version of Backus-Naur Form (BNF) to do this. This is not nearly as terrifying as it sounds! Constructions in the language are divided into *classes* and each class is given a name; we have used many of these classes already in this manual. For example, we defined 'stringchar' to mean any one of the characters of the RTL/2 Language subset of ISO7 code, excepting newline, tab, #, £, $ and ". In our definition rules we will use the class name 'stringchar' to stand for any of these characters; conversely, wherever 'stringchar' occurs we can replace it by any one of those characters.

The characters also occur in the definition in their own right, and keywords appear in capitals (e.g. PROC); these are known as *terminal symbols* of the language. Classes are written in small letters. The definition of the language merely combines classes and terminal symbols in a compact way. In section 18 we described the structure of the while-statement as

```
WHILE condition DO
  sequence of statements
REP
```

Our BNF description of the language is simply a formal version of this, in which each class, like 'condition', is rigorously defined.

The definition takes the form of a number of *productions,* each production defining a class in terms of other classes and terminal symbols. The class being defined is separated from its production by the symbol ::= which stands for "is defined to be". Three other symbols are used:

- | stands for "or"; thus A|B is read 'A or B'.
- [ ] enclosing a sequence means that the sequence may optionally be present.
- ... ellipsis dots indicate that the previous item may be repeated as many times as we like.

(The four symbols should more strictly be called *meta-symbols*). Hence [A|B]... stands for any list (including the empty list) of the alpha characters A, B. Examples of members of the class are

```
A
BA
ABBAAABAB
ABBBBBB
```

In our formal notation, the class for the while-statement is defined by

```
whilest::= WHILE condition DO sequence REP
```

This is identical to our earlier description; the definition of 'condition' and 'sequence' need to be added of course.

As a simple complete example, consider the formation of a name in RTL/2; in section 2 we defined a name as "a sequence of letters and digits with the proviso that the first character must be a letter". This is summarised in BNF as follows using three classes 'letter', 'digit' and 'name', and employing all the meta-symbols

```
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
name    ::= letter [letter|digit] ...
```

This merely gives the syntax of the language, its grammar. The entire grammar is given in this form in an Appendix of the Specification Manual. It does not say what is meaningful, nor how constructions will be interpreted. Thus in the BNF productions you will find no mention of type-checking or precedence of operators. This information is given in normal English in the text of the manual.

We use the BNF to illustrate one or two further points concerning RTL/2.

The module is defined by

```
module ::= moduleitem [; moduleitem] ... eom
```

Interpreted, this says that a module consists of a number of moduleitems (at least one) separated by semi-colons and terminated by 'eom'. If we look at the class 'eom' we find

```
eom::= end-of-module-character
```

Syntactically each module must be terminated by such a character, which is not part of the language character set. This character tells the compiler when to stop, and may also inform the reading device of the end of a character file. Whether such a character needs to be appended explicitly or not depends on the host machine on which the compiler is running and the form of the input medium on which the module is presented; in some systems an end-of-module-character will be supplied automatically. If this character is read in the middle of an RTL/2 item, the compiler will produce a failure message, and, of course, not read any more of the module.

The block statement is defined by

```
block ::= BLOCK blockbody ENDBLOCK
```

```
blockbody ::= [simpledec;] ... sequence
simpledec ::= simplemode initidlist
```

The classes 'sequence', 'simplemode', 'initidlist' do not concern us here. The class 'simpledec' cannot be a null item and hence the following RTL/2 is illegal.

```
BLOCK; INT I; ...
```

The first semi-colon effectively defines a null statement and hence no declarations can follow it.

Note that GOTO IF A=B THEN L ELSE M END is legal since

```
gotost ::= GOTO expn
```

and 'expn' allows an if-expression. However what the BNF doesn't tell you is that, in general, such a construction is not efficient! It is better to write:

```
IF A=B THEN GOTO L; END;
GOTO M;
```

Effectively the if-expression yields a name conditionally; note that this cannot be done for an array name which is to be subscripted or a procedure which is to be called.

```
IF A=B THEN SIN ELSE COS END (PI/8.0); is illegal.
```

The BNF tells us so since:

```
procst     ::= variable paralist|identifier paralist
identifier ::= name
variable   ::= simplevble|subscriptedvble|recordcomponent
```

and we can see that an if-expression for the name is not allowed.

Following chains of definitions in BNF can be quite long, but it is worthwhile persevering if you have a problem in syntax.

Semantic restrictions can be found in the specification manual and in relevant implementation documents; the machine independent rules have been given in this manual. The specification manual is laid out, with examples; the declarations for these examples are collected at the front of the manual. It is worth pointing out that examples in some classes are considered as separate and not parts of a program; hence there may be no semi-colons separating them.

That is RTL/2 presented somewhat abstractly. Like all skills, there is no substitute for experience in learning about RTL/2, so go forth and program!

Appendix 1

RTL/2 Language Subset of ISO7

Character	Decimal value	Language use
HT	9	layout—horizontal tab
LF	10	layout—newline
SP	32	layout—space
"	34	string quote
# £ $	35, 36, 92	not equals, strings
%	37	comments
&	38	not used
'	39	byte quote
(	40	open bracket
)	41	close bracket
*	42	multiply
+	43	add
,	44	comma
—	45	minus
.	46	constants, records
/	47	divide
0-9	48-57	numbers
:	58	labels, assignment etc.
;	59	statement, declaration separator
<	60	less than
=	61	assignment, equals
>	62	greater than
?	63	not used
@	64	not used
A – Z	65-90	names, numbers

Notes:

(i) The characters &, ? and @ are not used for any particular purpose in the language but they may occur in strings, comments and titles.

(ii) Because of lack of uniformity in manufacturers' treatment of #, £ and $ it should be made clear that they are considered interchangeable and all mean the same thing. The intention is that on any preparation equipment the key marked # may be used with confidence. Note that no confusion can arise as to the internal value as far as representing these characters in strings is concerned because they cannot stand for themselves.

Appendix 2

Keywords, showing sections in which new uses are introduced.

Appendix 3

Answers

** SECTION 2 **

1.

E) CONTAINS B - NOT A DIGIT
F) NO DIGIT FOLLOWING .
G) NO DECIMAL POINT, COMMA ILLEGAL
J) NO LEADING DIGIT
N) SPACE BEFORE EXPONENT
O) ILLEGAL CHARACTER *
P) POINT IN EXPONENT
S) NO LEADING DIGIT

2.

A)	0.726E2	B)	0.3E0
C)	0.673E4	D)	0.134E1
H)	0.15E-3	I)	0.100001E3
K)	0.27E12	L)	0.23E-1
M)	0.16E-7	Q)	0.6E9
R)	0.1174E2	T)	0.27E4

3.

C) CONTAINS NON-ALPHANUMERIC -
D) CONTAINS NON-ALPHANUMERIC '
E) DOES NOT START WITH LETTER
G) NON-ALPHANUMERIC #
I) RESERVED WORD
J) / IS NON-ALPHANUMERIC
M) DOES NOT START WITH LETTER
N) & IS NON-ALPHANUMERIC
T) SPACE : THIS IS IN FACT TWO NAMES

4.

```
REAL INCOME,NETTAXPAY,ALLOWANCES,TAXCODE,TAX
```

** SECTION 4 **

1.

VAR; STAT	W	X	Y	Z	A	B	C	D
1	0.0	0.0	*	*	*	*	*	*
2	0.0	0.0	1.0	1.0	*	*	*	*
3	0.0	0.0	1.0	1.0	Y	Y	*	*
4	0.0	0.0	1.0	1.0	Y	Y	*	Z
5	0.0	0.0	0.0	1.0	Y	Y	*	Z
6	0.0	0.0	0.0	*	Y	Y	*	Z
7	0.0	0.0	0.0	3.2	Y	Y	*	Z
8	0.0	0.0	0.0	3.2	Y	Y	W	Z
9	0.0	0.0	0.0	3.2	Y	Z	W	Z
10	0.0	0.0	0.0	3.2	Y	Z	W	Z
11	0.0	3.2	0.0	3.2	Y	Z	W	Z
12	0.3	3.2	0.0	3.2	Y	Z	W	Z
13	0.3	3.2	0.0	0.3	Y	Z	W	Z

* INDICATES UNDEFINED

THE SIXTH STATEMENT Z:=C IS MEANINGLESS. THE LEFT HAND SIDE REQUIRES A REAL NUMBER; C IS A REF-REAL VARIABLE, BUT DEREFERENCING PRODUCES AN UNPREDICTABLE RESULT SINCE NO ASSIGNMENT OF A NAME HAS BEEN MADE INTO C; THUS THE SECOND DEREFERENCING IS UNDEFINED, AND NO-ONE KNOWS WHAT VALUE WILL BE PLACED IN Z . THIS CAN BE DANGEROUS.

** SECTION 5 **

1.

X	Y
26.3	UNDEFINED
26.3	-0.27
26.3	-0.27
26.3	0.27
-0.27	0.27
-0.27	0.27
-0.27	-2.3
0.01	-2.3
-2.3	-2.3

2.

```
REAL X,P;
REF REAL Y;
   Y:=P;
   X:=26.3;
   VAL Y:=-0.27;
   % REMOVE TWO REDUNDANT ASSIGNMENTS %
   X:=-Y;
   VAL Y:=-X;
   VAL Y:=-2.3;
   X:=0.01;
   X:=-ABS Y;
```

CLEARLY EXAMPLES AT THIS STAGE ARE VERY ARTIFICIAL ! THE ABOVE "PROGRAM" COULD BE SIMPLIFIED TO THE SINGLE ASSIGNMENT

```
VAL Y:=X:=-2.3;
```

AND ALL THE INTERMEDIATE ASSIGNMENTS COULD BE REGARDED AS REDUNDANT. THE AIM IS TO GET YOU TO WRITE RTL/2 AND IN THIS CASE APPRECIATE AGAIN THE USE OF VAL.

**SECTION 6 **

1.1)
```
REAL READ1,READ2,ERROR;
   ERROR:=ABS(READ1-READ2)/READ1*100.0;
```

2)
```
REAL FAHR,CENT;
   CENT:=(FAHR-32.0)*5.0/9.0;
```

3)
```
REAL P,      % PRINCIPAL IN POUNDS %
     R,      % RATE OF INTEREST AS PERCENTAGE PER ANNUM %
     T,      % TIME IN YEARS %
     INTER; % INTEREST %
   INTER:=P*R*T/100.0;
```

4)
```
REAL X,BINOMIAL,XPLUS1;
   XPLUS1:=X + 1.0;
   BINOMIAL:=XPLUS1*XPLUS1;
   BINOMIAL:=BINOMIAL*BINOMIAL*XPLUS1;
% THIS IS MORE ECONOMICAL ON MULTIPLICATIONS THAN WRITING      %
% XPLUS1*XPLUS1*XPLUS1*XPLUS1*XPLUS1   AND MUCH MORE EFFICIENT %
% THAN EXPANDING IN THE FORM                                   %
% BINOMIAL:=1.0+X*5.0+X*X*10.0+X*X*X*10.0+X*X*X*X*5.0+X*X*X*X*X%
% OR THE FACTORISED FORM                                       %
% BINOMIAL:=1.0+X*(5.0+X*(10.0+X*(10.0+X*(5.0+X))));           %
```

2. THIS EXAMPLE CONTAINS A SIMPLE INTERPOLATION FOLLOWED BY REPLACEMENT OF Y-VALUES BY THEIR DEVIATION FROM A MEAN AND THE CALCULATION OF SUMS OF SQUARES

X	Y	X1	X2	Y1	Y2	YMEAN	SQUARES
*	*	1.0	*	*	*	*	*
*	*	1.0	2.0	*	*	*	*
*	*	1.0	2.0	7.0	*	*	*
*	*	1.0	2.0	7.0	10.0	*	*
1.6	*	1.0	2.0	7.0	10.0	*	*
1.6	8.8	1.0	2.0	7.0	10.0	*	*
1.6	8.8	1.0	2.0	7.0	10.0	8.6	*
1.6	0.2	1.0	2.0	7.0	10.0	8.6	*
1.6	0.2	1.0	2.0	1.6	10.0	8.6	*
1.6	0.2	1.0	2.0	1.6	1.4	8.6	*
1.6	0.2	1.0	2.0	1.6	1.4	8.6	4.56

*=UNDEFINED

** SECTION 7 **

2.

WE COULD DO THIS BY INTRODUCING INTERMEDIATE RESISTANCES REPRESENTING THE COMBINATIONS TAKEN TWO AT A TIME :

```
REAL R1,R2,R3,R4,R5,TOTAL;
   R3:=SERIES(0.6,R1);       % COMBINE R1 AND 0.6 %
   R4:=PARA(R3,1.3);         % COMBINE NOW WITH 1.3 %
   R5:=PARA(25.0,R2);        % COMBINE R2 AND 25.0 %
   TOTAL:=SERIES(R4,R5);     % *  COMBINE PARTIAL RESULTS %
```

HOWEVER, THE INTERMEDIATE RESULTS ARE NOT REQUIRED AND WE COULD WRITE ONE ASSIGNMENT STATEMENT (BASICALLY * ABOVE) IN WHICH THE PARAMETERS ARE FUNCTION CALLS (WHOSE PARAMETERS ARE THEMSELVES FUNCTION CALLS) :

```
TOTAL:=SERIES(PARA(1.3,SERIES(0.6,R1)),PARA(25.0,R2));
```

```
** SECTION 8 **


%* 1 *%

    PROC SERIES (REAL R1,R2) REAL;
    % RETURNS EFFECTIVE RESISTANCE OF R1,R2 JOINED IN SERIES %
       RETURN(R1+R2);
    ENDPROC;

    PROC PARA (REAL R1,R2) REAL;
    % RETURNS EFFECTIVE RESISTANCE OF R1,R2 JOINED IN PARALLEL          %
    % NOTE THAT SINCE PARAMETER NAMES ARE LOCAL TO THE PROCEDURE BODY %
    % WE CAN USE THE SAME NAMES R1 AND R2 WITHOUT AMBIGUITY             %
       RETURN (1.0/(1.0/R1 + 1.0/R2));
    ENDPROC;


%* 2 *%

    PROC BINOMIAL (REAL X) REAL;
    % RETURNS (1+X)**4 %
    REAL XPLUS1;   % USE LOCAL RATHER THEN X FOR CLARITY %
       XPLUS1:=X + 1.0;
       XPLUS1:=XPLUS1*XPLUS1;       % (1+X)**2 %
       RETURN(XPLUS1*XPLUS1);
    ENDPROC;


%* 3 *%

    PROC MAX (REAL A,B) REAL;
    % RESULT IS THE MAXIMUM OF A AND B %
       RETURN(0.5*(A + B + ABS(A-B)));
    ENDPROC;


%* 4 *%

    DATA GLOBAL;
       REAL P,Q,R;
    ENDDATA;

    PROC PERMUTE ();
    REAL TEMP;
       TEMP:=P;
       P:=Q;
       Q:=R;
       R:=TEMP;
    ENDPROC;


%* 5 *%


    PROC PERMUTE (REF REAL P,Q,R);
    REAL TEMP;
       TEMP:=P;
       VAL P:=Q;
       VAL Q:=R;
       VAL R:=TEMP;
    ENDPROC;
```

```
%* 6 *%

 % SINCE WE WISH TO CALCULATE FOR A NUMBER OF VALUES OF X AT FIXED
 % VALUES OF P,Q,R, IT IS MORE EFFICIENT TO HAVE P,Q,R AS DATA %
 % VARIABLES RATHER THAN PARAMETERS %

   DATA COEFFICIENTS;
      REAL P,Q,R;
   ENDDATA;

   PROC Y (REAL X) REAL;
      X:=X*X;
      RETURN(P*X*X + Q*X + R);
   ENDPROC;

   PROC EVALUATE ();
   REAL RESULT;
      P:=1.0;  Q:=2.0;  R:=3.0;
      RESULT:=Y(-0.5);
      RESULT:=Y(-10.2);
      RESULT:=Y(673.7);

      P:=2.5;  Q:=-3.5;  R:=1.0;
      RESULT:=Y(0.1);
      RESULT:=Y(7.2);
      RESULT:=Y(12.6);
   % IN PRACTICE, OF COURSE, RESULT WOULD BE USED BETWEEN THE %
   % VARIOUS ASSIGNMENT STATEMENTS %
   ENDPROC;


%* 7 *%

% A %
   PROC QUAD (REAL A,B,C, REF REAL ROOT1,ROOT2);
   % WE COULD REDUCE THE COEFFICIENT PARAMETERS TO TWO, NAMELY B/
   % C/A, BUT THIS PROBABLY MAKES THE PARAMETERS IN THE CALLS MOR
   % COMPLEX. WE HAVE CHOSEN TO RETURN BOTH ROOTS VIA PARAMETERS
   % FOR SYMMETRY; CLEARLY WE COULD RETURN ONE AS A RESULT %
   REAL ROOT;
      B:=-B/(2.0*A);
      ROOT:=SQRT(B*B - C/A);
      % IN PRACTICE WE WOULD NEED TO DEAL WITH COMPLEX ROOTS %
      VAL ROOT1:=B + ROOT;
      VAL ROOT2:=B - ROOT;
   ENDPROC;

% B %
   DATA COEFFICIENTS;
      REAL A,B,C;
   ENDDATA;

   PROC QUAD (REF REAL ROOT2) REAL;
   % HERE WE RETURN ONE ROOT AS A RESULT %
   REAL ROOT,COEFF;
      COEFF:=-B/(2.0*A);
      % NOTE THAT IN THIS CASE WE CANNOT MODIFY B %
      ROOT:=SQRT(COEFF*COEFF - C/A);
      VAL ROOT2:=COEFF - ROOT;
      % MUST SET UP ROOT2 BEFORE THE RETURN %
      RETURN(COEFF + ROOT);
   ENDPROC;
```

```
** SECTION 11 **


%* 1 *%
DATA SYSTEM;
   REAL MEASURED,SETPOINT,          % CURRENT VALUES %
      NEWERR,OLDERR,VERYOLDERR,     % LAST THREE ERROR TERMS %
      INTERVAL,                     % TIME INTERVAL %
      CORRECTION,                   % DELTA P - VALVE CHANGE %
      K,L,M;                        % CONSTANTS %
ENDDATA;

PROC PROP () REAL;
   RETURN(K*(NEWERR-OLDERR));
ENDPROC;

PROC PROPINT () REAL;
   RETURN( PROP() + L*NEWERR*INTERVAL);
ENDPROC;

PROC PROPINTDERIV () REAL;
   RETURN( PROP() + PROPINT ()
      + M*(NEWERR-2.0*OLDERR+VERYOLDERR) / INTERVAL);
ENDPROC;

PROC DDC ();
REAL TIME;    % LESS THAN 10.0 FOR PROPORTIONAL CONTROL %
              % LESS THAN 20.0 FOR PROPORTIONAL + INTEGRAL %
              % LESS THAN 30.0 FOR PROP, INTEGRAL + DERIVATIVE %
% SET CONSTANTS AND INPUT MEASUREMENT %
   NEWERR:=SETPOINT-MEASURED;

   IF TIME<10.0 THEN
      CORRECTION:=PROP ();
   ELSEIF TIME<20.0 THEN
      CORRECTION:=PROPINT ();
   ELSEIF TIME<30.0 THEN
      CORRECTION:=PROPINTDERIV ();
   ELSE    % SOME OTHER ACTION %
   END;
   % AS IT STANDS WE COULD HAVE USED A CONDITIONAL EXPRESSION : WE   %
   % ASSUME THAT WE MAY WISH TO ADD OTHER STATEMENTS FOR THE VARIOUS %
   % ACTIONS %

   OLDERR:=NEWERR;
   VERYOLDERR:=OLDERR;
   % ETC %
ENDPROC;


% CONTROLLER FOR VALVE ADJUSTMENT %
PROC DDC (REAL K,L,M,MEASURED,SETPOINT,INTERVAL,
          REF REAL OLDERR,VERYOLDERR) REAL;
REAL NEWERR,CORRECTION;
   NEWERR:=SETPOINT-MEASURED;
   CORRECTION:=K*(NEWERR-OLDERR)
               + L*NEWERR*INTERVAL
               + M*(NEWERR-2.0*OLDERR+VERYOLDERR) / INTERVAL;
   VAL VERYOLDERR:=OLDERR;
   VAL OLDERR:=NEWERR;
   RETURN(CORRECTION);
ENDPROC;
```

```
PROC PLANTACTION ();
REAL TIME;  % AS ABOVE %
REAL DELTA;   % REQUIRED CHANGE %
REAL MEAS,SETPT,INTER,OLDERR,VERYOLDERR;  % PLANT DATA %
   % ILLUSTRATE CALL OF THE FINAL VERSION OF DDC %

   IF TIME<30.0 THEN
      DELTA:=
      DDC(0.9, IF TIME>=10.0 THEN 1.3 ELSE 0.0 END,
               IF TIME>=20.0 THEN 0.6 ELSE 0.0 END,
                  MEAS,SETPT,INTER,OLDERR,VERYOLDERR);
   END;
   % ETC %
ENDPROC;


%* 2 *%

PROC STEP (REAL X) REAL;
   RETURN(IF X<0.0 THEN 0.0 ELSEIF X>0.0 THEN 1.0 ELSE 0.5 END);
   % NOTE NO EQUALITY COMPARISON %
ENDPROC;


%* 3 *%

PROC VOLUME (REAL DEPTH) REAL;
REAL LOWCYL, % WILL CONTAIN TOTAL VOLUME OF LOWER CYLINDER SECTION %
     PI;      % SET PI ONCE %
   PI:=3.14159;
   LOWCYL:=PI*3.0;  % CALCULATE ONCE : RADIUS IS 1.0 %
   RETURN(  IF DEPTH>6.0 THEN
               LOWCYL + PI*4.0*4.0*3.0 + PI*5.0*5.0*(DEPTH-6.0)
                  % CONTRIBUTIONS FROM 3 SECTIONS %
            ELSEIF DEPTH>3.0 THEN
               LOWCYL + PI*4.0*4.0*(DEPTH-3.0)
                  % CONTRIBUTIONS FROM 2 SECTIONS %
            ELSE PI*DEPTH
                  % ONLY LOWER SECTION %
            END);
ENDPROC;


%* 4 *%

PROC FABS (REAL X) REAL;
% DON'T CALL IT ABS - IT'S RESERVED %
   RETURN(IF X<0.0 THEN -X ELSE X END);
ENDPROC;
```

```
%* 5 *%

PROC SQRT (REAL A) REAL;
REAL  EPS,      % ACCURACY %
      CURX,     % CURRENT APPROXIMATION %
      NEXTX;    % NEXT APPROXIMATION %
   EPS:=0.001;
   CURX:=1.0;      % INITIAL GUESS %
   NEXTX:=(CURX + A/CURX) * 0.5;
   IF ABS(NEXTX-CURX) < EPS THEN RETURN(NEXTX);  END;
   CURX:=NEXTX;
   NEXTX:=(CURX + A/CURX) * 0.5;
   IF ABS(NEXTX-CURX) < EPS THEN RETURN(NEXTX);  END;
   CURX:=NEXTX;
   NEXTX:=(CURX + A/CURX) * 0.5;
   IF ABS(NEXTX-CURX) < EPS THEN RETURN(NEXTX);  END;
   CURX:=NEXTX;
   NEXTX:=   ...
% AND WE COULD CARRY ON BUT WITH NO GUARANTEE THAT WE WOULD ALWAYS   %
% CONVERGE IN THE NUMBER OF STATEMENTS PROVIDED; SOLUTION IMPOSSIBLE %
% AT THIS STAGE - BUT READ ON %
ENDPROC;
```

```
** SECTION 12 **


%* 1 *%

% THERE ARE MANY POSSIBLE WAYS TO CODE THIS ANSWER %
PROC ACTION (REAL P,Q) REAL;
REAL RATIO,     % TO HOLD P/Q %
     DIFF;      % TO HOLD P**2 - Q**2 %
   RATIO:=P/Q;
   DIFF:=P*P - Q*Q;
IF DIFF<0.0 THEN
   ERRORACTION();
   IF Q>20.0 THEN
      ALARM();
      RETURN(RATIO);
   END;
   GOTO RECYC;
END;
IF Q>10.0 THEN
         QUENCH();
RECYC:   RECYCLE();      % NOTE NO DIFFICULTY IN JUMPING INTO A %
                         % CONDITIONAL STATEMENT %
CHECKP:  IF P<0.0 THEN
            ALARM();
            RETURN(DIFF);
         END;
ELSEIF RATIO<1.0 THEN GOTO CHECKP;
ELSE
   CYCLE();
   IF Q<0.0 THEN
      RATIO:=-RATIO;     % NEGATING Q CAN ONLY AFFECT THE RATIO %
                         % THE LOCAL Q WILL BE LOST %
   END;
END;
RETURN(RATIO);
ENDPROC;


%* 2 *%

PROC CHECK () REAL;
REAL X,Y,       % FIRST COMPLEX NUMBER IS X + IY %
     U,         % INTERMEDIATE VARIABLE %
     COUNT;     % MULTIPLICATION COUNTER %
   COUNT:=0.0;
   X:=5.0/13.0;  Y:=12.0/13.0;     % MODULUS UNITY %
      % SECOND COMPLEX NUMBER WILL BE 4/5 + I.3/5 %
      % PRODUCT WILL BE FORMED IN X + IY %
NEXT: U:=0.8*X - 0.6*Y;     % NEW REAL PART; MUST NOT CORRUPT X YET %
      Y:=0.6*X + 0.8*Y;     % NEW IMAGINARY PART %
      X:=U;
      COUNT:=COUNT + 1.0;     % ANOTHER MULTIPLICATION %
      IF ABS(X*X + Y*Y - 1.0) < 0.0001 THEN GOTO NEXT;  END;
      RETURN (COUNT);
ENDPROC;
% NOTE THAT THIS METHOD OF COUNTING USING REALS IS NOT RECOMMENDED %
% CONTINUED ADDITION MAY GIVE CUMULATIVE ERRORS %
```

```
** SECTION 13 **


%* 1 *%

PROC SQRT (REAL X) REAL;
% RETURNS POSITIVE ROOT OF A POSITIVE REAL %
REAL OLDGUESS,NEWGUESS;
      OLDGUESS:=1.0;
TRY:  NEWGUESS:=(OLDGUESS + X/OLDGUESS) * 0.5;
      IF ABS(NEWGUESS-OLDGUESS) < 0.0001 THEN RETURN(NEWGUESS); END;
      OLDGUESS:=NEWGUESS;
      GOTO TRY;
ENDPROC;

PROC QUAD (INT A,B,C, REF REAL ROOT1,ROOT2) INT;
% FOR REAL ROOTS  RESULT IS 1 ROOTS IN ROOT1 AND ROOT2 %
% FOR COMPLEX CASE RESULT IS 0 , REAL PART IN ROOT1, IMAG IN ROOT2 %
REAL DISC,ROOT,NEWB;
   NEWB:=-B/2*A;      % WIDENING FOR BOTH OPERANDS OF / %
         % NOTE THAT WE CANNOT PERFORM THE MODIFICATION IN INTEGER B %
   DISC:=NEWB*NEWB - C/A;
   ROOT:=SQRT(ABS DISC);      % ENSURE POSITIVE PARAMETER %
   IF DISC<0.0 THEN
      % COMPLEX ROOTS %
      VAL ROOT1:=NEWB;
      VAL ROOT2:=ROOT;
      RETURN(0);
   END;
   % REAL CASE %
   VAL ROOT1:=NEWB+ROOT;
   VAL ROOT2:=NEWB-ROOT;
   RETURN(1);
ENDPROC;


%* 2 *%

PROC FIBRATIO () REAL;
INT U1,U2,TRANS;
REAL NEWRATIO,OLDRATIO;
   U1:=U2:=1;
   OLDRATIO:=1.0;      % NOTE THAT THIS MUST BE SEPARATE FROM THE %
                       % MULTIPLE ASSIGNMENT TO U1 AND U2 %
NEXTTERM:    TRANS:=U2;
             U2:=U1 + U2;      % NEXT TERM U,N %
             U1:=TRANS;        % PREVIOUS TERM U,N-1 %
             NEWRATIO:=U1/U2; % WIDENING %
             IF ABS(NEWRATIO-OLDRATIO)<0.0001 THEN RETURN(NEWRATIO);
             END;
             GOTO NEXTTERM;
ENDPROC;
% AS A FURTHER EXERCISE, EXTEND THIS PROCEDURE TO GIVE THE VALUE OF  %
% N FOR WHICH CONVERGENCE IS ACHIEVED; I.E. THE NUMBER OF ITERATIONS %
```

```
%* 3 *%

PROC INTEGERDIVIDE (INT A,B) INT;
% RETURNS THE QUOTIENT A:/B %
REAL Q;
   IF B=0 THEN
      % DIVIDE BY ZERO FAILURE %
      % SUITABLE ERROR ACTION %
   END;
   Q:=A/B;
   RETURN(INT(Q + IF Q<0,0 THEN 0,5 ELSE -0,5 END));
ENDPROC;

PROC MODULO (INT A,B) INT;
% RETURNS THE REMAINDER A MOD B %
   RETURN(A - INTEGERDIVIDE(A,B) * B );
ENDPROC;

% IT IS POSSIBLE TO WRITE THESE WITHOUT USING REAL ARITHMETIC %
% THOUGH THEY WILL THEN BE RATHER SLOW IN SOME CASES %
PROC INTEGERDIVIDE (INT A,B) INT;
INT COUNT,SIGN;
   IF B=0 THEN
      % DIVIDE BY ZERO FAILURE %
      % SUITABLE ERROR ACTION %
   END;
   SIGN:=IF A*B<0 THEN -1 ELSE +1 END;
      % TO AVOID POSSIBLE OVERFLOW WE COULD ACHIEVE THIS BY TESTS %
   A:=ABS A;
   B:=ABS B;
   COUNT:=0;
L: IF A>=B THEN
      COUNT:=COUNT+1;
      A:=A-B;
      GOTO L;
   END;
   RETURN(COUNT*SIGN);
ENDPROC;

% AND WE COULD COMBINE THE TWO FUNCTIONS %
PROC INTEGERDIVIDE (INT A,B, REF INT REMAINDER) INT;
INT COUNT,SIGNA,SIGNB;
   IF B=0 THEN
      % DIVIDE BY ZERO FAILURE %
      % SUITABLE ERROR ACTION %
   END;
   SIGNA:=IF A<0 THEN -1 ELSE +1 END;
   SIGNB:=(IF B<0 THEN -1 ELSE +1 END) * SIGNA;
         % BRACKETS FOR CLARITY - GIVES SIGN OF QUOTIENT %
   A:=ABS A;
   B:=ABS B;
   COUNT:=0;
L: IF A>=B THEN
      COUNT:=COUNT+1;
      A:=A-B;
      GOTO L;
   END;
   VAL REMAINDER:=A * SIGNA;
   RETURN(COUNT*SIGNB);
ENDPROC;
```

```
%* 4 *%

PROC ROUNDTOINTEGER (REAL X) INT;
REAL TEST;
INT VALUE;
   VALUE:=0;
   IF X<0.0 THEN
         TEST:=-0.5;
TRYNEG:  IF X>=TEST THEN RETURN(VALUE);  END;
         VALUE:=VALUE-1;
         TEST:=TEST-1.0;
         GOTO TRYNEG;
   END;

         TEST:=0.5;
TRYPOS:  IF X<TEST THEN RETURN(VALUE);  END;
         VALUE:=VALUE+1;
         TEST:=TEST+1.0;
         GOTO TRYPOS;
ENDPROC;
% NOTE THAT ANY OVERFLOW CONDITION WILL BE DETECTED BY THE OVERFLOW  %
% IN THE INTEGER STEPPING OF VALUE %
% THE REPEATED ADDITION/SUBTRACTION OF 1.0 MAY LEAD TO CUMULATIVE    %
% ERRORS; LOGICALLY A GOOD EXERCISE - IN PRACTICE NOT RECOMMENDED %



** SECTION 14 **


%* 1 *%

DATA COORDS;
   ARRAY (3) REAL POINT;          % X,Y,Z COORDINATES OF A POINT %
ENDDATA;

PROC DIST (REF ARRAY REAL PLACE) REAL;
% DELIVERS DISTANCE OF (X,Y,Z) FROM (0,0,0) %
REAL X,Y,Z;
   X:=PLACE(1);
   Y:=PLACE(2);
   Z:=PLACE(3);     % FIND COORDINATES ONCE ONLY %
   RETURN ( SQRT(X*X + Y*Y + Z*Z) );
      % ASSUME SQUARE ROOT FUNCTION %
ENDPROC;

% ILLUSTRATION OF CALL :   R:=DIST(POINT); %
```

```
%* 2 *%

PROC MEANANDDEV (REF ARRAY REAL A, REF REAL MEAN,DEV);
% RETURNS MEAN AND STANDARD DEVIATION OF ELEMENTS OF A %
INT N,          % NUMBER OF ELEMENTS %
  * I;          % COUNTER %
REAL SUM,       % SUM OF ELEMENTS %
     CUR,       % CURRENT ELEMENT VALUE %
     SUMSQ;     % SUM OF SQUARES %
   SUM:=SUMSQ:=0.0;
   N:=LENGTH A;
   I:=1;
L; IF I<=N THEN
      CUR:=A(I);
      SUM:=SUM + CUR;
      SUMSQ:=SUMSQ + CUR*CUR;
      I:=I+1;
      GOTO L;
   END;
   % NOTE THAT NULL ARRAY (N=0) NOT CATERED FOR %
   VAL MEAN:=SUM/N;
   VAL DEV:=SQRT( SUMSQ/N-MEAN*MEAN );
ENDPROC;


%* 3 *%

PROC PROD (REF ARRAY (,) REAL A,B,C);
% ASSUME A IS MXN, B IS NXM. C IS AN MXM ARRAY TO FORM PRODUCT AB %
% ASSUME NON-NULL ARRAYS %
INT I,J,K;      % COUNTERS %
REAL SUM;       % FORM PRODUCT SUM %
          I:=1;
NEXI:     IF I<=LENGTH C THEN
             J:=1;
NEXJ:        IF J<=LENGTH C THEN
                SUM:=0.0;
                K:=1;
NEXK:           IF K<=LENGTH B THEN
                   SUM:=SUM + A(I,K)*B(K,J);
                   K:=K+1;
                   GOTO NEXK;
                END;
                C(I,J):=SUM;
                J:=J+1;
                GOTO NEXJ;
             END;
             I:=I+1;
             GOTO NEXI;
          END;
ENDPROC;
```

```
%* 4 *%

DATA TABLES;
   ARRAY (100) REAL X,Y;
ENDDATA;

PROC INITIALISE ();
INT I;
REAL XVAL;
   % SETS UP TABLE OF Y FOR X IN (0.1,10.0) AT 0.1 INTERVALS %
      I:=1;
NEXT: IF I<=100 THEN
         X(I):=XVAL:=I*0.1;
         Y(I):=F(XVAL);        % F IS THE REQUIRED FUNCTION %
         I:=I+1;
         GOTO NEXT;
      END;
ENDPROC;

PROC INTERPOLATE (REAL XX) REAL;
% LINEAR INTERPOLATION FOR XX; RETURNS CORRESPONDING Y VALUE %
INT I;
   IF XX<0.1 THEN  % OUT OF RANGE ERROR ACTION %  END;
   IF XX>10.0 THEN % OUT OF RANGE ERROR ACTION %  END;
   I:=1;
NEXI: IF I<100 THEN
         IF XX=X(I) THEN RETURN(Y(I)); %EXACT POINT %  END;
         IF XX<X(I+1) THEN
            % INTERPOLATE %
            RETURN( (Y(I+1)-Y(I))*(XX-X(I))*10.0 + Y(I) );
         END;
         I:=I+1;
         GOTO NEXI;
      END;
   % I=100 AND XX MUST BE X(100) %
   RETURN(Y(100));
ENDPROC;

** SECTION 16 **

%* 1 *%

   PROC SETDATE ();
   % ENTERED AT MIDNIGHT EVERY DAY %
      DAY:=DAY+1;
      IF DAY>DINM(MONTH) THEN
         DAY:=1;   % START NEW MONTH %
         MONTH:=MONTH+1;
         IF MONTH>12 THEN
            YEAR:=YEAR+1;    % NEW YEAR %
            MONTH:=1;
         END;
      END;
   ENDPROC;

   DATA CALENDAR;
      ARRAY (12) INT DINM:=
         % CONTAINS NUMBER OF DAYS IN EACH MONTH %
         (31,28,31,30,31,30,31,31,30,31,30,31);
   ENDDATA;

   DATA DATE;
      INT DAY, MONTH, YEAR;
   ENDDATA;
```

```
** SECTION 17 **


%* 1 *%
LET EPS=0.001;

PROC QUAD (REAL A,B,C,  % COEFFICIENTS %
           REF REAL ROOT1,ROOT2  % ROOTS IF REAL, REAL AND IMAG %
                                 % PARTS IF COMPLEX %  ) INT;
   % RESULT IS NON-ZERO IF ROOTS ARE COMPLEX %
REAL NEWB:=-B/(A*2.0), ROOT,
     SQ:=NEWB*NEWB - C/A;
   BLOCK
   REAL POSSQ:=ABS SQ, OLDGUESS:=1.0,  NEWGUESS:=(POSSQ+1.0)*0.5;
L:     IF ABS(OLDGUESS-NEWGUESS)>EPS*NEWGUESS THEN
          OLDGUESS:=NEWGUESS;
          NEWGUESS:=(NEWGUESS + POSSQ/NEWGUESS)*0.5;
          GOTO L;
       END;
       ROOT:=NEWGUESS;
   ENDBLOCK;
   % CLEARLY THE INNER BLOCK IS NOT ESSENTIAL BUT IT DOES CORDON %
   % OFF THE EVALUATION OF THE ROOT %
   IF SQ<0.0 THEN
      % COMPLEX ROOTS %
      VAL ROOT1:=NEWB;
      VAL ROOT2:=ROOT;
      RETURN(1);
   END;
   VAL ROOT1:=NEWB+ROOT;
   VAL ROOT2:=NEWB-ROOT;
   RETURN(0);
ENDPROC;


%* 2 *%

PROC SMOOTH (REF ARRAY REAL F);
INT I:=2,   % COUNTER; FIRST POINT NOT SMOOTHED %
    LEN:=LENGTH F;  % CALCULATE ONCE TO AVOID ON EACH ITERATION %
REAL F2:=F(1), F3:=F(2);   % REMEMBER UNSMOOTHED VALUES %
NEXTEL:  IF I<LEN THEN
            BLOCK
            REAL F1:=F2;
               F2:=F3;
               F3:=F(I+1);
               % NOW HAVE THREE UNSMOOTHED VALUES TO FORM NEW F(I) %
               F(I):=(F1+F2+F3)/3.0;
            ENDBLOCK;
            I:=I+1;
            GOTO NEXTEL;
         END;
ENDPROC;
```

```
%* 3 *%

PROC MEANANDDEV (REF ARRAY REAL A, REF REAL MEAN,DEV);
REAL SUM:=SUMSQ:=0.0;
INT LEN:=LENGTH A;
   IF LEN=0 THEN
      % SUITABLE ACTION ; AVOID DIVISION BY ZERO %
   END;
   BLOCK
   INT I:=1;    % COUNTER %
NEXTEL:
      IF I<=LEN THEN
         BLOCK
         REAL CUR:=A(I);    % CURRENT ELEMENT %
            SUM:=SUM + CUR;
            SUMSQ:=SUMSQ + CUR*CUR;
         ENDBLOCK;
         I:=I+1;
         GOTO NEXTEL;
      END;
   ENDBLOCK;
   VAL MEAN:=SUM/LEN;
   VAL DEV:=SQRT(SUMSQ/LEN - MEAN*MEAN);
   % ASSUME EXISTENCE OF SQUARE ROOT PROCEDURE %
ENDPROC;


%* 4 *%

PROC PROD (REF ARRAY (,) REAL A,B,C);
INT ROW:=1,    % ROW COUNTER %
    LEN:=LENGTH C;
NEXTROW:
   IF ROW<=LEN THEN
      BLOCK
      INT COL:=1;    % COLUMN COUNTER %
NEXTCOL:
         IF COL<=LEN THEN
            BLOCK
            REAL SUM:=0.0;    % ACCUMULATOR %
            INT K:=1,    % COUNTER %
                LB:=LENGTH B;
NEXTPROD:       IF K<=LB THEN
                   SUM:=SUM + A(ROW,K)*B(K,COL);
                   K:=K+1;
                   GOTO NEXTPROD;
                END;
                C(ROW,COL):=SUM;
            ENDBLOCK;
            COL:=COL+1;
            GOTO NEXTCOL;
            END;
      ENDBLOCK;
      GOTO NEXTROW;
   END;
ENDPROC;
```

```
%* 5 *%

PROC POWER (REAL A, INT N) REAL;
   % RETURNS A TO THE NTH POWER %
   IF N<0 THEN
      A:=1.0/A;
      N:=-N;
      % POWER OF RECIPROCAL IN NEGATIVE INDEX CASE %
   END;
   RETURN ( IF N=0 THEN 1.0 ELSEIF N=1 THEN A
            ELSE POWER(A*A,N:/2)*POWER(A,N MOD 2)
            END );
   % DECOMPOSES INTO A PRODUCT  OF 1 OR A TIMES A POWER OF A %
   % SQUARED : REPEAT RECURSIVELY %
ENDPROC;


%* 6 *%

% METHOD 1 : NON-RECURSIVE %

DATA CARDDATA1;
   ARRAY (8) INT BUF;   % ASSUME <=8 DIGITS IN AN INTEGER %
   ARRAY (10) INT CDCODES1:=(512,256,128,64,32,16,8,4,2,1);
ENDDATA;

PROC CARDCODE1 (INT X);
   % THE PROBLEM IS THAT SUCCESSIVE REMAINDERS GIVE THE DIGIT CODES %
   % IN REVERSE ORDER SO WE MUST STORE THEM TEMPORARILY IN AN ARRAY %
   % BUF BIG ENOUGH FOR THE LARGEST POSSIBLE NUMBER OF DIGITS IN AN %
   % INTEGER - HENCE MACHINE DEPENDENT %
INT Y:=1;
L:
   BUF(Y):=CDCODES1(X MOD 10 + 1);
   IF X>9 THEN
      X:=X:/10;
      Y:=Y+1;
      GOTO L;
   END;
L1:
   CARDCOLUMN(BUF(Y));
   Y:=Y-1;
   IF Y#0 THEN GOTO L1;  END;
ENDPROC;


% METHOD 2 : RECURSIVE %

% THIS METHOD IS MORE COMPACT ALTHOUGH IT USES MORE STACK ( BUT NOT %
% AN ABSURD AMOUNT SINCE CARDCODE2 IS ONLY CALLED ONCE FOR EACH %
% DIGIT ); IT ALSO HAS NO ARRAY WORKSPACE OTHER THAN A LOOK-UP %
% TABLE USED IN A READ-ONLY MANNER : IT IS THEREFORE RE-ENTRANT %

DATA CARDDATA2;
   ARRAY (10) INT CDCODES2:=(512,256,128,64,32,16,8,4,2,1);
ENDDATA;

PROC CARDCODE2 (INT X);
   IF X>9 THEN CARDCODE2(X:/10);  END;
   CARDCOLUMN(CDCODES2(X MOD 10 + 1));
ENDPROC;
```

```
** SECTION 18 **


%* 1 *%

PROC ORDER (REF ARRAY REAL A);
INT I:=1,LEN:=LENGTH A;
   WHILE I<LEN DO
      BLOCK
      REF REAL MIN:=A(I);
      INT J:=I+1;
         WHILE J<=LEN DO
            IF A(J)<MIN THEN MIN:=A(J);  END;
            J:=J+1;
         REP;
         BLOCK
         REAL TEMP:=A(I);
            A(I):=MIN;
            VAL MIN:=TEMP;
         ENDBLOCK;
      ENDBLOCK;
      I:=I+1;
   REP;
ENDPROC;


%* 2 *%

% ASSUME A,B IN A DATA BRICK, THE REQUIRED FUNCTION IS %
%  PROC F (REAL X) REAL %
% ASSUME F(A) , F(B) DIFFER IN SIGN %
% ASSUME A<B %

PROC SIGN (REAL X) INT;
REAL Q:=F(X);
   RETURN(IF Q<0.0 THEN -1 ELSEIF Q>0.0 THEN +1 ELSE 0 END);
ENDPROC;

PROC BISECT () REAL;
REAL BOT:=A,TOP:=B;
INT TOPSIGN:=SIGN(TOP);
   WHILE TOP-BOT>0.01 DO
      BLOCK
      REAL MID:=(BOT+TOP)*0.5;
      INT MIDSIGN:=SIGN(MID);
         IF MIDSIGN=0 THEN RETURN (MID);  % EXACT ROOT %
         END;
         IF MIDSIGN#TOPSIGN THEN
            % ROOT IN (MID,TOP) %
            BOT:=MID;
         ELSE
            % ROOT IN (BOT,MID) %
            TOP:=MID;
         END;
      ENDBLOCK;
   REP;
   % ROOT NOW IN (BOT,TOP) A RANGE LESS THAN 0.01; RETURN AVERAGE %
   RETURN((BOT+TOP)*0.5);
ENDPROC;
```

```
%* 3 *%

PROC TRANSIENT ();
REAL OLDSCAN:=SCAN(),   % INITIAL READING %
     NEWSCAN;
      DELAY(10);    % TIME DELAY BETWEEN SUCCESSIVE READINGS %
                    % WILL DEPEND UPON INSTRUMENT CHARACTERISTICS %
      NEWSCAN:=SCAN();  % NEXT READING %
      WHILE ABS(NEWSCAN-OLDSCAN)>0.001*ABS NEWSCAN DO
         DELAY(10);
         OLDSCAN:=NEWSCAN;
         NEWSCAN:=SCAN();
      REP;
ENDPROC;

** SECTION 19 **

%* 1 *%

PROC PROD (REF ARRAY (,) REAL A,B,C);
   FOR I:=1 TO LENGTH C DO
      FOR J:=1 TO LENGTH C DO
         REAL SUM:=0.0;
         FOR K:=1 TO LENGTH B DO
            SUM:=SUM + A(I,K)*B(K,J);
         REP;
         C(I,J):=SUM;
      REP;
   REP;
ENDPROC;


%* 2 *%

PROC MEANANDDEV (REF ARRAY REAL A, REF REAL MEAN,DEV);
REAL SUM:=SUMSQ:=0.0;
INT LEN:=LENGTH A;
   IF LEN=0 THEN  % SUITABLE ACTION %  END;
   FOR I:=1 TO LEN DO
   REAL CUR:=A(I);
      SUM:=SUM + CUR;
      SUMSQ:=SUMSQ + CUR*CUR;
   REP;
   VAL MEAN:=SUM/LEN;
   VAL DEV:=SQRT(SUMSQ/LEN - MEAN*MEAN);
ENDPROC;


%* 3 *%

PROC MEAN (REF ARRAY REAL A) REAL;
REAL SUM:=0.0,MU;
INT LEN:=LENGTH A;
   IF LEN=0 THEN  % SUITABLE ACTION %  END;
   FOR I:=1 TO LEN DO
      SUM:=SUM + A(I);
   REP;
   MU:=SUM/LEN;
   FOR I:=1 TO LEN DO
      A(I):=A(I)-MU;
   REP;
   RETURN(MU);
ENDPROC;
```

```
%* 4 *%

PROC SMOOTH (REF ARRAY REAL F);
REAL F2:=F(1), F3:=F(2);
% UPDATES USING UNSMOOTHED VALUES THROUGHOUT %
   FOR I:=2 TO LENGTH F - 1 DO
      % END POINTS NOT UPDATED %
   REAL F1:=F2;
      F2:=F3;
      F3:=F(I+1);
      F(I):=(F1+F2+F3) / 3.0;
   REP;
ENDPROC;

%* 5 *%

LET SIZE=100;    % SIZE OF TABLES %

DATA TABLES;
   ARRAY (SIZE) INT INDEX;
   ARRAY (SIZE) REAL VALUE;
      % ARRAYS ASSUMED SET UP BY SOME PROCESS %
ENDDATA;

PROC FIND (INT N) REAL;
   % RETURNS REAL VALUE CORRESPONDING TO INDEX N %
   FOR I:=1 TO SIZE DO
      IF INDEX(I)=N THEN RETURN(VALUE(I));  END;
   REP;
   % SUITABLE ACTION FOR MEANINGLESS INDEX %
   % NOTE THAT WE HAVE TO PROVIDE A RESULT FOR THIS CASE %
   RETURN(0.0);
ENDPROC;

%* 6 *%

LET LA=100;    % LENGTH OF MESSAGE ARRAYS %

DATA MESSAGES;
   ARRAY (LA) REAL INPUT,OUTPUT;
ENDDATA;

PROC PROCESSTAPE ();
   % READ NUMBER OF INTEGERS ON TAPE FOR LOOP CONTROL %
   TO IREAD() DO
   INT MARK:=IREAD();    % READ NEXT INTEGER 'INSTRUCTION' %
      IF MARK=1 THEN
         ARRAYIN(INPUT);
      ELSEIF MARK=2 THEN
         ARRAYOUT(OUTPUT);
      ELSEIF MARK=3 THEN
         MEAN(OUTPUT);    % RESULT LOST %
      ELSEIF MARK=4 THEN
         FOR I:=1 TO LA DO
            OUTPUT(I):=( INPUT(I) - 32.0 ) / 1.8;
         REP;
      ELSEIF MARK=5 THEN
         SMOOTH(INPUT);
      ELSE
         % SUITABLE ACTION FOR UNDEFINED INTEGER VALUE %
      END;
   REP;
ENDPROC;
```

```
** SECTION 20 **


%* 1 *%

PROC SELECTARITH (REAL A,B, INT CHOICE) REAL;
%    CHOICE   ACTION       %
%      1      SUM          %
%      2      PRODUCT      %
%      3      DIFFERENCE   %
%      4      QUOTIENT     %
%      5      AVERAGE      %
%      6      GREATER      %
%      7      LESSER       %

   SWITCH CHOICE OF ADD,MULT,SUB,DIV,AV,GR,LESS;
          % ERROR ACTION IF ANY %
          RETURN(0.0);

ADD:      RETURN(A+B);
MULT:     RETURN(A*B);
SUB:      RETURN(A-B);
DIV:      RETURN(A/B);
AV:       RETURN((A+B)*0.5);
GR:       RETURN(IF A>B THEN A ELSE B END);
LESS:     RETURN(IF A<B THEN A ELSE B END);
   % NOTE THAT THE TRANSFERS FROM EACH SEQUENCE ARE IN THE RETURNS %
ENDPROC;
```

** SECTION 22 **

%* 1 *%

DECIMAL	BINARY	OCTAL	HEXADECIMAL
6	BIN 110	OCT 6	HEX 6
27	BIN 11011	OCT 33	HEX 1B
84	BIN 1010100	OCT 124	HEX 54
317	BIN 100111101	OCT 475	HEX 13D
2120	BIN 100001001000	OCT 4110	HEX 848
32677	BIN 111111110100101	OCT 77645	HEX 7FA5

** SECTION 23 **

%* 1 *%

A) VALID
B) NO VALID REAL
C) NO BINARY SCALE
D) VALID
E) NO VALID REAL
F) VALID
G) OUT OF RANGE
H) VALID
I) INVALID SCALE
J) EXPONENT AND SCALE INVERTED

%* 2 *%

```
PROC SCALETOFRAC ( REAL READING,  % ABSOLUTE READING %
                   SCALE          % FULL-SCALE VALUE %
                                  ) FRAC;
   RETURN(FRAC(READING/SCALE));
ENDPROC;

PROC FRACTOABSVAL ( FRAC READING, % FRACTION READING %
                    REAL SCALE    % FULL-SCALE VALUE %
                                  ) REAL;
   RETURN(READING*SCALE);
          % READING WIDENED TO REAL %
ENDPROC;
```

```
** SECTION 27 **

% 1 %
% IN THE PACKING WE WILL KEEP THE INTEGER VALUE IN THE TOP OF THE %
% WORD AND USE ARITHMETIC SHIFTS TO ACCESS IT; IN THIS WAY THE SIGN %
% IS ORGANISED AUTOMATICALLY %

LET MAXREAD=100;      % DATA READINGS %

DATA PLANTARRAY;
   ARRAY (MAXREAD) INT PLANTINF;
      % EACH INTEGER HAS (STARTING FROM THE BOTTOM OF THE WORD) %
      %   BITS 1,2,3: FLAGS A,B,C       %
      %   BITS 4-5  : SEQUENCE NO.      %
      %   BITS 6-8  : STATUS VALUE      %
      %   BITS 9-TOP: INTEGER VALUE (SIGNED) %
ENDDATA;

PROC UPDATEVALUE (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND HEX FF LOR (NEW SLA 8);
ENDPROC;

PROC READVALUE (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) SRA 8);
ENDPROC;

PROC UPDATESTATUS (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND NOT OCT 340 LOR(NEW SLL 5);
ENDPROC;

PROC READSTATUS (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) SRL 5 LAND 7);
ENDPROC;

PROC UPDATESEQUENCE (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND NOT BIN 11000 LOR(NEW SLL 3);
ENDPROC;

PROC READSEQUENCE (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) SRL 3 LAND 3);
ENDPROC;

PROC UPDATEA (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND NOT 1 LOR NEW;
ENDPROC;

PROC READA (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) LAND 1);
ENDPROC;

PROC UPDATEB (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND NOT BIN 10 LOR NEW SLL 1;
ENDPROC;

PROC READB (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) SRL 1 LAND 1);
ENDPROC;
```

```
PROC UPDATEC (INT INDEX,NEW);
   PLANTINF(INDEX):=PLANTINF(INDEX) LAND NOT BIN 100 LOR NEW SLL 2;
ENDPROC;

PROC READC (INT INDEX) INT;
   RETURN(PLANTINF(INDEX) SRL 2 LAND 1);
ENDPROC;

LET SET=1;

PROC SCAN ();
   FOR I:=1 TO MAXREAD DO
      IF READSEQUENCE(I)=3 THEN
         BLOCK
         INT INTVAL:=READVALUE(I);
            IF INTVAL<=90 OR INTVAL>90 THEN
               % ALARM CONDITION DETECTED %
               % APPROPRIATE ACTION %
            END;
         ENDBLOCK;
      END;
      BLOCK
      INT A:=READA(I), B:=READB(I), C:=READC(I);
         IF READSTATUS(I) LAND 1=0 THEN
            % STATUS EVEN %
            IF A=SET AND B=SET OR A#SET AND B#SET AND C#SET THEN
               % ALARM CONDITION %
               % SUITABLE ACTION %
            END;
         ELSE
            % STATUS ODD %
            % NOTE THAT C=SET IS ALARM REGARDLESS OF A,B %
            IF C=SET OR A#SET AND B=SET THEN
               % ALARM CONDITION %
               % SUITABLE ACTION %
            END;
         END;
      ENDBLOCK;
   REP;
ENDPROC;
```

```
** SECTION 29 **

%* 1 *%
% DDC EXAMPLE USING RECORD STRUCTURES %

MODE LOOPRECORD (
   INT MSMT,       % MEASURED VALUE %
       SETPT,      % SETPOINT %
       LASTM,      % PREVIOUS MEASUREMENT %
       LASTSP,     % PREVIOUS SETPOINT %
       VP,         % VALVE POSITION %
   BYTE HIALM,     % HIGH ALARM VALUE %
        LOALM,     % LOW ALARM VALUE %
        KP,        % PROPORTIONAL CONSTANT %
        KI,        % INTEGRAL CONSTANT %
   REF LOOPRECORD CASC   % RECORD WHOSE SETPOINT IS TO BE %
                         % ADJUSTED - CASCADE %
               );

DATA CONTROLDATA;
% PLANT INFORMATION %
   ARRAY (100) LOOPRECORD PLANT:=( (0,0,0,0,0,0,0,0,0,NULL)(100) );
   LOOPRECORD NULL:=(0,0,0,0,0,0,0,0,0,NULL);  % DUMMY %
ENDDATA;

PROC CONTROL ();
% THIS PROCEDURE DRIVES THE PROCESSING OF THE RECORDS %
START:
      FOR I:=1 TO 100 DO
         DDC(PLANT(I));        % DDC ON EACH LOOP %
         DELAY(10);            % 10 MILLISEC DELAY %
      REP;
      DELAY(4000);             % 4 SECONDS PAUSE %
      GOTO START;              % DDC AGAIN %
ENDPROC;

PROC DDC ( REF LOOPRECORD LOOP);
% PERFORMS DDC ON SINGLE LOOP %
   % COMPUTE CHANGE IN VALVE POSITION %
   INT ERR:=LOOP.MSMT - LOOP.SETPT,
       DELV:=LOOP.KI*ERR +
                LOOP.KP*(ERR - (LOOP.LASTM-LOOP.LASTSP));

   % UPDATE RECORD %
   LOOP.LASTM:=LOOP.MSMT;
   LOOP.LASTSP:=LOOP.SETPT;

% ADJUST SETPOINT IF CASCADE, OTHERWISE VALVE %
   IF LOOP.CASC:#: NULL THEN
      LOOP.CASC.SETPT:=LOOP.CASC.SETPT + DELV;
   ELSE
      LOOP.VP:=LOOP.VP + DELV;
   END;
ENDPROC;
```

```
%* 2 *%

LET M=0;
LET F=1;

MODE PERSON ( REF ARRAY BYTE NAME,
                BYTE SEX,AGE,
                REF PERSON SPOUSE,
                           FATHER,
                           FIRSTCHILD,   % I.E. THE ELDEST IN A CHAIN %
                           NEXTSIBLING  % I.E. THE FIRST YOUNGER ONE %
                                       );

DATA PEOPLE;
   PERSON NOBODY:=("",F,0,NOBODY,NOBODY,NOBODY,NOBODY);
         % DUMMY RECORD FOR TERMINATING LINKED CHAINS %
   ARRAY (28) REF ARRAY BYTE RELATIVE:=(
% THIS IS AN ARRAY OF MESSAGES %
       "FATHER",
       "MOTHER",
       "GRANDFATHER",
       "GRANDMOTHER",
       "FATHER-IN-LAW",
       "MOTHER-IN-LAW",
       "ELDER BROTHER",
       "ELDER SISTER",
       "YOUNGER BROTHER",
       "YOUNGER SISTER",
       "SON",
       "DAUGHTER",
       "GRANDSON",
       "GRANDDAUGHTER",
       "SON-IN-LAW",
       "DAUGHTER-IN-LAW",
       "UNCLE",
       "AUNT",
       "NEPHEW",
       "NIECE",
       "",    % NOT USED %
       "HUSBAND",
       "WIFE",
       "NOT RELATED",
       "IDENTITY",
       "ILL DEFINED",
       "COUSIN"(2)    % TO ALLOW FOR BOTH SEXES %
                    );
ENDDATA;

PROC OUTREL (REF PERSON A,B);
% OUTPUTS RELATIONSHIP OF B TO A %
% WE ASSUME SIMPLE MONOGAMOUS LEGITIMATE SYSTEM %
   TWRT(RELATIVE(RELATION(A,B)));
ENDPROC;

PROC RELATION (REF PERSON A,B) INT;
% RESULT IS INDEX INTO THE ARRAY RELATIVE %
   IF A:=:NOBODY OR B:=:NOBODY THEN RETURN(26);  END;
   IF A:=:B THEN RETURN(25);  END;
   RETURN(B.SEX + REL(A,B));
   % PASS PARAMETERS THROUGH TO SEXLESS PROCEDURE %
ENDPROC;
```

```
PROC REL (REF PERSON A,B) INT;
% RESULT IS INDEX TO BASIC RELATIONSHIP NEEDS MODIFICATION BY SEX %
   IF A.SPOUSE:=:B THEN RETURN(22);  END;
   BLOCK
   REF PERSON  DADA:=A.FATHER,
               MUMA:=DADA.SPOUSE,
               DADB:=B.FATHER,
               MUMB:=DADB.SPOUSE;
% PARENTS %
      IF DADA:=:B OR DADA.SPOUSE:=:B THEN
         RETURN(1);
      END;
% CHILDREN %
      IF DADB:=:A OR DADB.SPOUSE:=:A THEN
         RETURN(11);
      END;
% IN-LAWS %
      BLOCK
      REF PERSON FINLA:=A.SPOUSE.FATHER;
         IF FINLA:=:B OR FINLA.SPOUSE:=:B THEN
            RETURN(5);
         END;
      ENDBLOCK;
      BLOCK
      REF PERSON FINLB:=B.SPOUSE.FATHER;
         IF FINLB:=:A OR FINLB.SPOUSE:=:A THEN
            RETURN(15);
         END;
      ENDBLOCK;
      BLOCK
      REF PERSON  GFDA:=DADA.FATHER,
                  GFMA:=MUMA.FATHER,
                  GFDB:=DADB.FATHER,
                  GFMB:=MUMB.FATHER;
               % WE NOW HAVE THE FOUR GRANDFATHERS INVOLVED %
% GRANDPARENTS %
         IF GFDA:=:B OR GFDA.SPOUSE:=:B OR GFMA:=:B OR
               GFMA.SPOUSE:=:B THEN
            RETURN(3);
         END;
% GRANDCHILDREN %
         IF GFDB:=:A OR GFDB.SPOUSE:=:A OR GFMB:=:A OR
               GFMB.SPOUSE:=:A THEN
            RETURN(13);
         END;
% SIBLINGS %
         BLOCK
         REF PERSON NEXT:=DADA.FIRSTCHILD;
            WHILE NEXT:#:NOBODY DO
               IF NEXT:=:B THEN
                  RETURN(IF A.AGE<B.AGE THEN 7 ELSE 9 END);
               END;
               NEXT:=NEXT.NEXTSIBLING;
            REP;
         ENDBLOCK;
```

```
% UNCLES AND AUNTS %
          BLOCK
          REF PERSON  NEXTF:=GFDA.FIRSTCHILD,
                      NEXTM:=GFMA.FIRSTCHILD;
       % NOTE THAT NEXTM IS A.FATHER.SPOUSE.FATHER.FIRSTCHILD %
             WHILE NEXTF:#:NOBODY AND NEXTM:#:NOBODY DO
                IF NEXTF:=:B OR NEXTM:=:B THEN
                   RETURN(17);
                END;
                NEXTF:=NEXTF.NEXTSIBLING;
                NEXTM:=NEXTM.NEXTSIBLING;
             REP;
          ENDBLOCK;
% NEPHEWS AND NIECES %
          BLOCK
          REF PERSON  NEXTF:=GFDB.FIRSTCHILD,
                      NEXTM:=GFMB.FIRSTCHILD;
             WHILE NEXTF:#:NOBODY AND NEXTM:#:NOBODY DO
                IF NEXTF:=:A OR NEXTM:=:A THEN
                   RETURN(19);
                END;
                NEXTF:=NEXTF.NEXTSIBLING;
                NEXTM:=NEXTM.NEXTSIBLING;
             REP;
          ENDBLOCK;
% COUSINS %
          IF GFDA:=:GFDB OR GFDA:=:GFMB OR GFMA:=:GFDB OR GFMA:=:GFMB
             THEN  RETURN(27);
          END;
       ENDBLOCK;
    ENDBLOCK;
    RETURN(24);
ENDPROC;
```

```
** SECTION 30 **


%* 1 *%

TITLE
SIMPLE ARITHMETIC OPERATIONS;

EXT PROC () REAL RREAD;
EXT PROC (REAL) RWRT;
EXT PROC (REF ARRAY BYTE) TWRT;

SVC DATA RRSED;
   BYTE TERMCH, IOFLAG;
ENDDATA;

ENT PROC ARITH ();
REAL OP1:=RREAD();
BYTE OPCHAR:=TERMCH;    % PICK UP OPERATOR SYMBOL WHICH TERMINATES %
                        % FIRST OPERAND %
REAL OP2:=RREAD(), RES;

   % NOW USE CLOSENESS OF CHARACTER VALUES OF OPERATORS %
   SWITCH OPCHAR-'*'+1 OF
         MULT,PLUS,ILL,MINUS,ILL,DIV;

ILL:     % ALL OTHER CHARACTERS DROP THROUGH %
         TWRT("ILLEGAL OPERATOR");
         RETURN;

PLUS:    RES:=OP1+OP2;
         GOTO EXIT;

MINUS:   RES:=OP1-OP2;
         GOTO EXIT;

MULT:    RES:=OP1*OP2;
         GOTO EXIT;

DIV:     RES:=OP1/OP2;

EXIT:
   RWRT(RES);
ENDPROC;
```

```
** SECTION 32 **


%* 1 *%

TITLE
PRINT ROMAN NUMERALS
VERSION 1  19.12.1973;

LET TRUE=255;
LET FALSE=0;
LET NL=10;

SVC DATA RRSIO;
   PROC () BYTE IN;
   PROC (BYTE) OUT;
ENDDATA;

EXT PROC (INT) IWRT;
EXT PROC (REF ARRAY BYTE) TWRT;

DATA NUMERALS;
   ARRAY (26) INT ROMAN:=(-1(2),100,500,-1(4),1,-1(2),50,1000,
         -1(8),5,-1,10,-1(2)  );
   LABEL FAILLAB;
ENDDATA;

% WE COULD WRITE THIS AS ONE PROCEDURE. WE HAVE SPLIT IT INTO TWO %
% SEPARATE PROCEDURES TO ILLUSTRATE THE USE OF A LABEL VARIABLE %
% FOR ERROR RECOVERY %

PROC INDIGIT () INT;
INT B:=IN();    % WIDENED FOR ARITHMETIC %
   IF B=' ' THEN
      % TERMINATOR %
      RETURN(0);
   END;
   IF B<'A' OR B>'Z' THEN
      TWRT("#NL#NOT A LETTER#NL#");
      GOTO FAILLAB;
   END;
   RETURN(ROMAN(B-'A'+1));   % COULD EVALUATE THIS AS B-64 %
ENDPROC;
```

```
ENT PROC PRINTROMAN ();
% READS IN NUMBER IN ROMAN NUMERALS AND PRINTS IT IN DECIMAL %
% FAILURE MESSAGES AND VALUE -1 FOR ILLEGAL CHARACTERS OR SEQUENCES %
% SPACE CHARACTER IS TERMINATOR %
INT VALUE:=0, OLDDIG:=100, % ALWAYS OK ON ENTRY TO TEST %
              NEWDIG;
BYTE TRAIL:=FALSE;
   FAILLAB:=FAIL;    % INITIALISE FAILURE LABEL %
NXCHAR:
   NEWDIG:=INDIGIT();
   IF NEWDIG=0 THEN
      % TERMINATOR %
      GOTO EXIT;
   END;
   IF NEWDIG>0 THEN
      IF OLDDIG<NEWDIG THEN
         IF TRAIL#FALSE THEN    % TEST AGAINST ZERO FOR PREFERENCE %
            % TRAP SEQUENCES SUCH AS IXL %
            TWRT("#NL#INVALID SEQUENCE#NL#");
            GOTO FAIL;    % USE ACTUAL LABEL LOCALLY : NOT FAILLAB %
         END;
         TRAIL:=TRUE;
         NEWDIG:=NEWDIG - OLDDIG*2;    % REDUCE INCOMING DIGIT %
      ELSE
         TRAIL:=FALSE;
      END;
      VALUE:=VALUE+NEWDIG;
      OLDDIG:=NEWDIG;
      GOTO NXCHAR;
   END;

   % TREAT -1 CASE %
   TWRT("#NL#LETTER NOT ROMAN NUMERAL#NL#");
FAIL:
   VALUE:=-1;
EXIT:
   IWRT(VALUE);
ENDPROC;
```

RTL/2604/5414/R3/1Ed/13/178