AD-760 549

A PROGRAMMER'S GUIDE TO PDP-10 EULER

William M. Newman, et al

Utah University

Prepared for:

Rome Air Development Center
Defense Advanced Research Projects Agency

June 1970

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

RADC-TR- 73-152
Technical Report
June 1970

A PROGRAMMER'S GUIDE TO PDP-10 EULER

University of Utah

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order No. 829

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) University of Utah Salt Lake City, Utah 84112	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. N/A
3. REPORT TITLE A PROGRAMMER'S GUIDE TO PDP-10 EULER	
4. DESCRIPTIVE NOTES (Type of report and inclusive dates)	
5. AUTHOR(S) (First name, middle initial, last name) William M. Newman Henri Gouraud Donald R. Oestreicher	
6. REPORT DATE June 1970	7a. TOTAL NO. OF PAGES 73 7b. NO. OF REFS 3
8a. CONTRACT OR GRANT NO. AF30(602)4277 b. PROJECT NO. ARPA Order No. 829 c. Program Code No. 6D30 d.	9a. ORIGINATOR'S REPORT NUMBER(S) None 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) RADC-TR-73-152
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.	
11. SUPPLEMENTARY NOTES Monitored by: Murray Kessleman RADC (ISCE), GAFB NY 13441 AC 315 330-2018	12. SPONSORING MILITARY ACTIVITY Defense Advanced Research Projects Agency Wash DC 20301

13. ABSTRACT

This manual describes the EULER language as implemented on the DEC PDP-10 computer. EULER is a block-structured language, similar to Algol-60 but simplified by omitting type declarations and by altering the way procedures are defined and called. PDP-10 EULER includes features for list and array manipulation, and also for a number of forms of input - output, including graphics.

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Graphics						
Computer Graphics						

A PROGRAMMER'S GUIDE TO PDP-10 EULER

William M. Newman
Henri Gouraud
Donald R. Oestreicher

Contractor: University of Utah
Contract Number: AF30(602)4277
Effective Date of Contract: 20 May 1966
Contract Expiration Date: 30 November 1970
Amount of Contract: $5,028,542.00
Program Code Number: 6D30

Principal Investigator: Thomas G. Stockham, Jr.
Phone: 801 581-8224

Project Engineer: Murray Kesselman
Phone: 315 330-2018

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Murray Kesselman RADC (ISCE), GAFB, NY 13441 under Contract AF30(602)4277.

i-f

A PROGRAMMER'S GUIDE TO PDP-10 EULER

PUBLICATION REVIEW

This technical report has been reviewed and is approved

RADC Project Engineer
MURRAY KESSELMAN

ACKNOWLEDGMENTS

EULER was originally implemented on the PDP-10 as a class exercise. Since then it has grown into a full-fledged compiler-interpreter system. We would like, however, to acknowledge the work done on the original implementation by members of CS 632 at the University of Utah, namely David Anderson, Kay Brown, Duane Call, Patrick Baudelaire, Roger DeBry, Joe Locascio, Don Vickers, and Martin Yonke.

We would also like to thank Jim Curry and Carl Ellison for their helpful advice and assistance.

TABLE OF CONTENTS

INTRODUCTION

EULER is a block-structured language, similar in appearance to Algol but embodying many fresh concepts which make it an easier language to understand and use. The original reason for implementing it on the PDP-10 was to create a language for experimenting with data structures. However, it soon appeared that EULER had many applications as a general-purpose language with good data-handling and debugging facilities, and this manual has been prepared for people who wish to make use of it as such.

The first thing that must be said about PDP-10 EULER is that it is different from EULER as proposed by Wirth and Weber[(1)]. It contains <u>for</u> statements, arrays as well as lists, and omits <u>go to</u> statements. There are also some major differences in the way it has been implemented, but these are probably not of interest to the general user. Readers familiar with Algol 60[(2)] will have little difficulty in using EULER, once they have understood the basic differences between the two languages. These are covered in Part I.

EULER programs are executed by an interpreter called SEUL. This interpreter operates on Polish-string object code generated by the EULER compiler. The object code is in the form of six-bit bytes, and some care was taken to make it readable for debugging purposes. A number of other debugging aids have been added to the interpreter which probably make this feature redundant.

Other useful features of PDP-10 EULER are string, list and matrix operations, file input-output and a very straightforward library feature. These are all described in the rest of this report.

PART I

THE BASIC FEATURES OF EULER

1. EULER Variables

Like most high-level languages, EULER has facilities for handling integers, real numbers, boolean values, strings and arrays. These can all be stored into variables and manipulated in the usual way. However, EULER imposes no restrictions on the type of data that may be stored into a given variable. A single variable may, during execution of a program, successively contain an integer, a real number, a boolean value, a string, an array, a list and a procedure. This contrasts with Algol 60, in which variables are declared to have a certain type when the program is written and during execution can contain only that type of data.

The EULER interpreter avoids this restriction by saving a few extra bits of information with each variable; by using these bits during execution it can determine how the contents should be treated. This of course reduces execution speed. However, it permits mixed types of data to be stored into lists and arrays, and it also reduces the burden on the programmer. EULER variables are declared in a single NEW declaration following the start of a block:

```
BEGIN NEW A, Z1, Z2, MAXVALUE;
          .................
          .................
END
```

Any statement between the declaration and the final END may refer to these variables. Outside the block they are meaningless, and any attempt to refer to them will cause an error. The contents of a variable just after it has been declared are undefined. Variable names may be any number of characters in length; all characters are significant.

Variables may be subscripted to address a particular cell in a list or an array or to pass arguments to a procedure:

```
A[23]
L3[K+1]
MAX[A,B]
```

Each of the subscripts in the list enclosed within brackets may be any EULER statement or expression: see below for a list of the various types of statement permitted in EULER. Also discussed below is the use of multiple subscripts, such as:

```
L23[K][3][N+5]
```

2. Expressions

Expressions may be formed from variables, constants and other expressions enclosed in parentheses. The most common type is an arithmetic expression:

```
A + 3.2 - 100 * (B + C/17)
```

However, logical expressions are just as useful: they have either true or false values:

```
A > B
A = 3 OR NOT (B < 17 AND BOOL3)
```

Expressions may also involve strings, lists or arrays, as described later.

3. Statements

EULER includes most of the types of statements permitted by Algol. These include assignment statements, conditional (IF) statements, FOR statements, and compound statements or blocks. An expression (arithmetic or logical) is a valid EULER statement. GO TO statements and labels are omitted. PDP-10 EULER also includes some special forms of output statement (PRINT, WRITE) and list manipulation statement (INSERT, REMOVE).

3.1 Assignment Statement

An important feature of EULER is that every statement has a value. In most cases this value is not put to any use, but is thrown away after the semicolon which separates statements is passed. For example, the value of the following statement is the sum of the values stored in A and B:

```
A + B; .....
```

By itself, this expression does nothing. Similarly, the following conditional expression may have the value of C or D, but will not affect the state of the program:

```
IF A > B THEN C ELSE D;
```

On the other hand, if we incorporate this expression into an assignment statement, as EULER will allow us to do, we can change the program's state:

```
P ← IF A > B THEN C ELSE D;
```

Here the value of the statement, which is the value of either C or D, is stored into P. EULER allows any statement, with any type of value, to be used as the right-hand side of an assignment statement.

3.2 FOR Statement

The FOR statement provides a basis for most algorithms involving repeated operations. There are several variants of the FOR statement. The FOR-STEP-UNTIL statement allows an operation to be executed a predetermined number of times:

```
FOR K ← 1 STEP 1 UNTIL 10 DO A[K] ← 0
```

This will store zero into cells A[1] to A[10] inclusive. The scope of the DO is limited to one statement:

```
FOR N ← 1 STEP 1 UNTIL 5 DO P ← P * N; Q ← Q + P
```

The first statement following DO,"P ← P * N", will be executed five times; the second,"Q ← Q + P", will be executed only once, following completion of the FOR statement. To cause both statements to be executed after every step, we must include them in a single compound statement:

```
FOR N ← 1 STEP 1 UNTIL 5 DO
   BEGIN
      P ← P * N;
      Q ← Q + P
   END
```

WHILE may be used instead of UNTIL or STEP-UNTIL so that looping terminates when a condition is no longer true:

```
LOOKING ← TRUE;
FOR K ← 1 STEP 1 WHILE LOOKING DO
   LOOKING ← A[K]#N;
```

The above will loop through A until a cell equal to N is found.

```
FOR INP ← TRUE WHILE INP DO
   BEGIN NEW N;
      N ← INVAL;
      ANS ← ANS + N
   END
```

This example creates an endless loop since INP never becomes false. This type of loop is useful for writing interactive programs in EULER.

Note that there is no semicolon before an END. Semicolons are used only to separate a statement or declaration from the following statement. Errors will occur if this rule is not followed.

4. String Manipulation in EULER

A string of any length may be stored into a variable:

```
S3 ← "THIS IS THE PLACE"
```

The contents of this variable may then be printed out, concatenated with other strings, or manipulated in various ways. It is not possible to access individual characters in a string. However, any string may be converted to a list of integers, using the reserved procedure UNSTRING:

```
L7 ← UNSTRING[S3]
```

Each cell in the list L7 will receive one character, converted into an integer representing the appropriate ASCII code. The reverse operation is also permitted:

```
S2 ← STRING[N]
```

N may be a list or array of integers or just a single integer. A string is formed of all the codes up to the first zero or non-integer.

5. Arrays and Matrices

EULER arrays are similar to FORTRAN arrays in that the lower bound is always unity. However, EULER arrays may have any number of dimensions. They are created as follows:

```
A ← ARRAY[2,ASIZE]
```

and may be accessed as follows:

```
A[1,J+1] ← 2
```

Any type of information may be stored in any array cell, including another array:

```
A3[10] ← "JOHN SMITH"'
A3[11] ← TRUE;
A3[14] ← ARRAY[20];
```

An array stored in a cell of another array can be accessed by double subscripting:

```
X ← A3[14][N]
```

Two-dimensional arrays may be treated as matrices. The interpreter is able to carry out matrix multiplication and addition:

```
A ← ARRAY[2,3];   %A becomes a matrix with 2 rows of 3 cells%
B ← ARRAY[3,4];
........
........
C ← A * B;
```

This will create a new array C, whose dimensions are 2x4, containing the matrix product of A and B. Matrices may be scaled:

```
A ← ARRAY[1,4];
......
A ← A/A[1,4];
```

Matrices may contain integer or real values in any mixture. The result of a matrix operation leaves all the contents real.

6. Lists

Wirth's original description of EULER includes list-processing operations, and with a few minor changes these have been implemented in PDP-10 EULER. Figure 1 shows an EULER list, stored into a variable L. Cells in this list can be accessed in the same way as array cells: for example, L[1]=3.6, L[2] ="ABC", L[4]=123. L[3] is itself a list, and its cells can be addressed by double subscripting: L[3][1]=TRUE, L[3][2]=0.0.

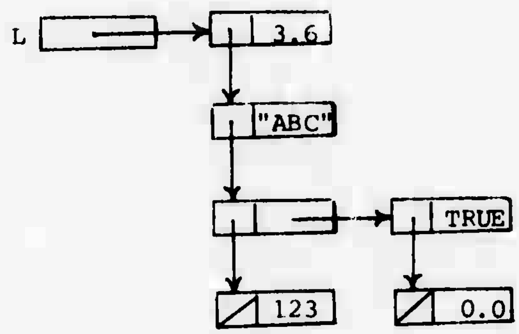

Figure 1

There are three principal ways of constructing a list:

1.) By explicitly defining its contents:

```
L←[3.6,"ABC",[TRUE 0.0],123]
```

2.) By defining the list and later defining its contents:

```
L←LIST[4]
L[1]←3.6;
```

```
L[2]←"ABC";
L[3]←LIST[2];
L[4]←123;
L[3][1]←TRUE;
L[3][2]←0.0;
```

3.) By concatenating existing lists:

```
L1←[3.6,"ABC"];
L2←[TRUE,0.0];
L3←L1&[L2]&[123]
```

The expression [ ] can be used to indicate an empty list. Wirth's two other operations, LENGTH and TAIL, are also included. LENGTH allows the number of cells in a list to be determined:

```
LENGTH[L]≡4
LENGTH[L[3]]≡2
```

TAIL removes the first element from a list:

```
TAIL[L]≡["ABC",[TRUE,0.0],123]
TAIL[TAIL[L][2]][1]≡0.0
```

PDP-10 EULER also includes two special statements, INSERT and REMOVE, to make list operations more efficient. INSERT has four variants:

a) INSERT L1 BEFORE L2
b) INSERT L1 AFTER L2
c) INSERT L1 BEFORE L2:N
d) INSERT L1 AFTER L2:N

(a) and (b) add list L2 to list L1, respectively before the first and after the last element of L2. (c) and (d) permit additions to be made anywhere within a list--N is an index into L2, and can be any expression. For example, the structure in Figure 1 could be created as follows:

```
INSERT ["ABC",[TRUE,0.0]] AFTER [3.6,123]:1
```

REMOVE has only one form:

```
REMOVE L:N
```

which removes the Nth element of list L. Thus REMOVE L:1 is equivalent to L←TAIL[L].

The value of the INSERT statement is the resultant list structure. REMOVE returns as value the removed element.

7. Procedures

One of the most attractive features of EULER is its handling of procedures. Basically, a procedure may be assigned to any variable; then whenever that variable is accessed, the procedure will be executed. Procedures may be stored into cells of an array or list. The way in which procedures are defined is as follows:

```
OUTAB←'PRINT A; PRINT B'
```

All the statements included within quotes are executed when the procedure is accessed. Arguments may be passed to procedures by the use of subscripts; there must be a formal declaration at the start of the procedure, listing all the parameters to be passed:

```
MAX←'FORMAL A,B; IF A>B THEN A ELSE B';
X←MAX[J,3*P-17];
```

The mechanism of calling procedures in EULER is quite different from that in Algol. Unless specified, parameters are passed by value. Each of the expressions in the subscript list is evaluated, and each of these values is assigned to a formal variable, starting with the first. Thus in the example above, A would receive the value of J, B the value of the expression 3*P-17.

Calls by name are achieved by enclosing the arguments within quotes. Consider the following example:

```
A←0;
PRINT2←'FORMAL X; X←2; PRINT X';
PRINT2[A];
PRINT2['A'];
```

PRINT2[A] merely prints the number 2: since it is called by value, the contents of A are not changed. PRINT2['A'] on the other hand is a call by name, hence all references within PRINT2 to the formal X are treated as references to A. At the end of this second call, A contains the value 2.

The value returned by a procedure is the value of the last statement executed within the procedure. Thus the value of the above procedure MAX is the value of the IF-expression. Procedures may also be thought of as returning an address. For example:

```
CELL3←'A[3]';
CELL3←22;
B←CELL3;
```

This example defines an "access procedure" which allows data to be stored into or read out of A[3] as if CELL3 were a simple variable. Note that when a procedure is stored into a variable, that variable becomes "execute only" and no other contents can replace the procedure.

Arrays, lists and strings may be passed as arguments to a procedure. For example, the following procedure ZMATRIX will create a two-dimensional array of the required dimensions, with all cells set to zero:

```
ZMATRIX←'FORMAL M,N;
     BEGIN NEW A,J,K;
          A←ARRAY[M,N];
          FOR J←1 STEP 1 UNTIL M DO
               FOR K←1 STEP 1 UNTIL N DO
                    A[J,K]←0;
          A
     END'
```

and can then be called as follows:

```
ROTN←ZMATRIX[3,3]
```

8. <u>Teletype Input and Output</u>

The INPUT statement in EULER reads one character from the teletype. If nothing has been typed, the program waits until a character is typed. The value of INPUT is an integer, representing the ASCII code of the character typed. It may be converted to a single-character string with the STRING operator:

```
IF STRING[INPUT]="G" THEN PROGO
```

The INPUT statement has been incorporated in a number of library procedures for input of numbers and text (see Appendix III).

Output to the teletype is achieved by using the PRINT statement:

```
PRINT A;
PRINT "ANSWER IS", X23
```

Any number of arguments may be listed in a PRINT statement, and their values may be of type integer, real or string. Numbers are printed out in a fixed format. Programmers may define their own format as follows:

```
PRINT A,B IN "A= \\\  B= \\.\\\ ";
FMT←"ANGLE IS \\.\ DEGREES";
PRINT 180*THETA/PI IN FMT;
```

Each item in the output list of a formatted PRINT statement will be inserted in a field of the format; these fields are indicated by back-slashes. A period will cause numbers to be converted to floating-point notation--otherwise integer notation will be used. Positive values are left unsigned unless a sign position is indicated:

```
PRINT X1, X2 IN "+\\\ +\\\"
```

9. Euler Constants

Constants may be integers, real numbers, or strings. Any number including a decimal point is treated as real. Any text enclosed within double quotes is stored as a string. The compiler will not accept certain characters within strings, so the following conventions are used:

'B	bell
'C	carriage return
'F	form feed
'L	line feed
'N	carriage return - linefeed
'S	space
'T	tab
''	single quote

10. Program Formatting and Contents

Spaces, tabs, and carriage-return/line-feeds may be inserted anywhere in the source program except within a symbol or operator, or within a string enclosed in double quotes. The program may therefore be indented by means of spaces and tabs, as illustrated by most of the examples in this manual.

Wherever a space, tab, or carriage-return/line-feed is permitted, a comment may be inserted by enclosing it within percent symbols:

```
IF A > B THEN  %EXCHANGE A AND B%
  BEGIN NEW T;  %T IS TEMPORARY VAR%
    T ← A; A ← B; B ← T  %EXCHANGE COMPLETE%
END;
```

Comments may extend to more than one line.

The complete program should be enclosed within a BEGIN...END pair. This first BEGIN must be followed by a declaration, and preceded by a <u>title</u>, which is any symbol:

```
TITLE PROG3
BEGIN NEW X, Y, P;
  ......
  ......
END
```

PART II

HOW TO USE PDP-10 EULER

1. Compiling

Source programs should be prepared and filed in the usual way with QED or TECO. They can then be compiled in the following manner:

```
.R EULER
*DEV:FNAME1.EXT←DEV:FNAME2.EXT
```

or the following shorter form may be used:

```
.R EULER
*XXXX←XXXX
```

This assumes that the source file is XXXX.SRC and is on the disk. An object file called XXXX.MAC is created, also on the disk. Users are encouraged to use this form since the EULER debugging routines rely upon these file-name conventions.

2. Loading and Executing

EULER programs are not compiled into machine code and loaded in the conventional manner. Instead they are interpreted by a program called SEUL*. Users should type

```
.R SEUL
```

*Non-French speakers: this is unpronounceable. The closest approximation is SERL.

and then type the name of the object file produced by the EULER compiler:

```
.R SEUL
*DSK:XXXX.MAC
```

If the device name is omitted, DSK is assumed; if both device name and extension are omitted, DSK and .MAC are assumed. Provided the normal file-name conventions are used, the following is therefore sufficient:

```
.R SEUL
*XXXX
```

Loader switches are provided to request special action during loading. These may be typed at any point in the file name.

/U	prints out all undeclared variables after loading. These include external and library procedures.
/B	program enters EULER DDT after loading.

Example:

```
.R SEUL
*PROG/U
```

Unless the /B switch is used, the program proceeds to execute as soon as loading is complete. A carriage-return/line-feed is output to the teletype as execution commences.

3. Run-Time Errors

If an error is detected during execution, the following happens:

i) An error message is printed on the teletype;

ii) The statement in the source file in which the error occurred is printed;

iii) The program returns to the monitor.

We have tried to ensure that the source statement printed out is indeed the statement in which the error occurred. However, the technique we have used takes some short cuts to avoid complete re-compilation of the program. On occasions, several statements will be printed if SEUL cannot determine the precise statement in which the error occurred. A list of error messages will be found in Appendix V.

4. Debugging Aids

Debugging aids fall into two categories:

i) Facilities to print out the state of the program;

ii) Facilities to set break-points so that execution is interrupted at a certain point.

Whenever a run-time error occurs, the contents of any active variable may be printed out. To do this, type REENTER (or REE for short). The program should respond with an asterisk, and you may then type the name of any active variable, followed by a slash. If the variable is inactive, "U?" will be printed in an appropriate format. The following are some examples of printouts from EULER DDT:

```
.REE
*VAL/        231
*K/          5.60017
*NAME/       "JOE"
*FOUND/      TRUE
*XYZ/        UNDEF
```

Variables to which procedures have been assigned, and formal variables called by name, simply print out as "PROC". Similarly arrays and lists print out as "ARRAY" and "LIST". You may however access individual array and list cells by adding a subscript or subscripts to the name:

```
*MAT[3,2]/    1.71503
*L3[5][6]/    77
```

To print out the entire active stack contents, type:

```
,/
```

Break-points may be set prior to execution by using the /B loader switch. After loading is complete an asterisk is printed, and up to eight break-points may then be set in the program. Wherever possible this feature uses the conventions of PDP-10 DDT.

To set a break-point, type a line number in the source code followed by $nB ($ = alt-mode; n is the break-point number, 1 to 8). The break-point will be set at the first "store" operation in that line. For example, if the following is line 27 of the source program, and 27$1B is typed, the program will break before storing 33 into A:

```
A←33; B←A+5;
```

To cause breaking on the second or successive "store" operations, you may type:

```
        27,2$1B
or      27,3$1B  etc.
```

The integer following the comma indicates to which of the "stores" the break-point is to be attached. If this number exceeds the number of assignments on the line, a statement will be chosen in the lines following. The break-point number, n, may be omitted. In this case numbers are assigned automatically, starting at 1. To clear a break-point type Ø$nB. To clear all break-points, type $B.

To print out the contents of any line, type the line number, followed by a slash:

```
27/
```

The most recent line typed can be referred to as ".", and other lines may be addressed relative to it:

27/	prints line 27
.+1/	prints line 28, . becomes 28
.,1$B	sets a break-point at the first "store" in line 28
.-5,2$B	sets another break-point at the second "store" in line 23
. 2/	prints line 30 (space and + are equivalent)

As in PDP-10 DDT, line-feed and .+1/ are equivalent, and so are ↑ and .-1/.

To start execution, type $G. The program will execute normally until a break-point is encountered. Then execution will cease, and the break-point number, together with the value just about to be stored, will be printed:

```
3B >> Ø.Ø1753
*
```

You may now examine variables and set or clear break-points, as described above. To resume execution, type $P.

PART III

ADVANCED EULER PROGRAMMING

This section is devoted to some of the more refined techniques in EULER programming, and to some of the facilities in the language which were not described in Part I.

1. Use of Statement Values

It is frequently possible to take advantage of the fact that statements possess values. An example was given earlier in Part I. More elaborate examples are discussed here.

When matrices are being used, it is sometimes necessary to create a new matrix with its cells set to certain initial values. Suppose we wish to store into A either the matrix currently in B or, if B is undefined, a 3x3 unity matrix. This can be done as follows:

```
A←IF TYPE[B] = 4 THEN B ELSE
     BEGIN NEW T,J,K;
          T←ARRAY[3,3];
          FOR J←1 STEP 1 UNTIL 3 DO
               FOR K←1 STEP 1 UNTIL 3 DO
                    T[J,K]←IF J=K THEN 1 ELSE 0;
          T
     END .
```

This example makes use twice of the values of IF statements. Another technique that may be used with IF statements is the <u>compound</u>

logical expression. The expression following "IF" may be any expression whose value is true or false. An expression may be any statement or statements enclosed within parentheses*, so the following is permitted:

```
IF (A←B[X];A>0) THEN Q←A
```

Since all the statements within parentheses are executed before the test is carried out, this provides a method of including unconditional statements in chains of IF statements (IF...THEN... ELSE IF...THEN...ELSE IF...) without the use of BEGIN and END:

```
IF(X←X-1;A[X,Y]=0) THEN TRUE ELSE
      IF(X←X+1;Y←Y+1;A[X,Y]=0) THEN TRUE ELSE
          IF(X←X+1;Y←Y-1;A[X,Y]=0) THEN TRUE ELSE
               (X←X-1;Y←Y-1;A[X,Y]=0)
```

The above statement finds whether any cell adjacent to (X,Y) in the matrix A contains zero, and if so returns with X and Y set to the position of the first such cell it finds.

Difficulties often arise with IF statements because all parts of a complex logical expression are evaluated before the test is applied (this is out of line with Wirth's proposals). In a statement of the form IF L1 AND L2 THEN A ELSE B the evaluation of L2 may cause an error if L1 has the value false. One solution is the nested IF statement:

```
IF L1 THEN
     BEGIN
          IF L2 THEN A
     END ELSE B
```

---

* Note that parentheses ( ) are equivalent to BEGIN END

The BEGIN is necessary here since the second IF statement does not include an ELSE clause, but the first one does. Another correct version is:

IF L1 THEN IF L2 THEN A ELSE B ELSE B

and the following will also work:

IF(IF L1 THEN L2 ELSE FALSE) THEN A ELSE B

2. Procedures

Part I mentioned that variables into which procedures have been stored become "execute only." This means that it is not possible successively to store different procedures into a variable as follows:

P ← 'IF N=0 THEN A ELSE B';
P ← 'A ← B ← 0';

Whenever a procedure variable is accessed, the procedure is called immediately; so the example above will succeed only in storing the second procedure into either A or B.

The only way to replace one procedure by another is to reclaim and re-allocate the space it occupies. This is difficult to do with ordinary variables, since a variable is only reclaimed when the end is reached of the block in which it was declared. With lists and arrays, however, it is easier to do. Suppose we wish to build a list L in the form shown in Figure 2.

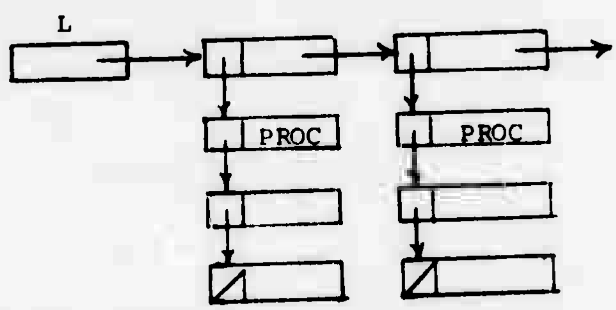

Figure 2

Each element of L is itself a list, which contains in the first element a procedure indicating what to do with the following two elements. If we wish to change the procedure, leaving the rest of the sub-list unchanged, we can do do by discarding the first element and then redefining it. For example:

```
L[K]←LIST[1] & TAIL[L[K]];
L[K][1]←'FORMAL A,B; A+B';
```

We may now "evaluate" any sublist K in the following manner:

```
VAL←L[K][1][L[K][2],L[K][3]]
```

The concept of <u>recursive procedures</u> is widely understood and used. All EULER procedures may be called recursively. However, if the number of nested calls exceeds 30, stack overflow will occur. To illustrate the use of recursion, here is an example taken from the EULER library file:

```
TITLE SINE
'FORMAL X; BEGIN NEW PI;
    PI←3.14159;
    IF X>=0 THEN
        BEGIN IF X<0.1 THEN X-(X↑3)/6 ELSE
            IF X<=PI/2 THEN 2*SIN[X/2]*SQRT[1-SIN[X/2]↑2] ELSE
                IF X<=PI THEN SIN[PI-X] ELSE SIN[X-2*PI]
        END ELSE -SIN[-X]
    END'
```

One of the original ideas behind PDP-10 EULER was the concept of using procedures as <u>access functions</u>. It is possible to use procedures to attach names to specific list or array elements, and to store into and read out of these elements by means of their names:

```
BEGIN NEW A, LENGTH, HEIGHT, WIDTH, K, J;
     A←ARRAY[100,3];
     LENGTH←'FORMAL X; A[X,1]';
     HEIGHT←'FORMAL X; A[X,2]';
     WIDTH←'FORMAL X; A[X,3]';
     ........
     LENGTH[K]←INVAL;
     ........
     IF LENGTH[J]=0 THEN...
```

The above example makes LENGTH[X] synonymous with A[X,1]. Notice the use of EULER's block structure to pre-empt a "reserved" procedure name, i.e., LENGTH. Within the block in which LENGTH is declared the user's procedure will take precedence over the reserved LENGTH procedure which determines lengths of lists and arrays. Users who feel they can improve upon the EULER library procedures can pre-empt them in the same way, as described below.

3. External and Library Procedures

EULER programs may be written as external procedures by adopting and following slightly modified syntax:

```
TITLE EXTPROC                 TITLE EXTPROC
'FORMAL F1, F2;       or      '........
........                      ........'
.......
```

A complete example is shown above in the sine procedure. External procedures may be called from other programs without declaring them. The interpreter assumesthat every undefined variable is an external procedure and attempts to find it on the disk as follows:

a) by looking it up on the users area with extension .MAC;
b) by looking it up on his area with extension .EUL;
c) by looking it up under [1,1] with extension .EUL.

If all of these fail, an error message is printed. This order of precedence is useful in a number of ways. For example, if a program has been designed to be controlled by the mouse, and the user wishes to test it from the teletype, he can do so by writing his own external MOUSE procedure and filing the object code on his disk area as MOUSE.MAC or MOUSE.EUL. The following procedure would allow him to type in a switch number and two coordinates, and to the program would be indistinguishable from library MOUSE procedure:

```
TITLE MOUSE
'BEGIN NEW SN;
     SN←INVAL;
     IF SN=1 THEN [TRUE, FALSE, FALSE, INVAL, INVAL] ELSE
     IF SN=2 THEN [FALSE, TRUE, FALSE, INVAL, INVAL] ELSE
          [FALSE, FALSE, TRUE, INVAL, INVAL]
END'
```

4. File Input-Output

EULER programs may read and write standard PDP-10 text files. For this purpose, a WRITE statement and a reserved procedure called READ have been added. They operate in a fashion exactly analogous to PRINT and INPUT:

```
WRITE "MOVE AC,",NAME;    % will write MOVE AC, and the
                            contents of NAME %

CH←READ;                  % will read one character into
                            CH as an integer %
```

WRITE statements may include format specifications.

In order to make use of READ and WRITE, the programmer must include statements to <u>open</u> and <u>close</u> files. If you are going to write a file, you must open it for output:

```
OUTFILE["DSK","FNAME","EXTN"];
```

After all output is complete, the file is closed for output:

```
OCLOSE;
```

Since only one file at a time may be opened for output, the OCLOSE statement requires no arguments. Existing files may be opened for input and later closed as follows:

```
INFILE["DSK","FNAME","EXTN"];
........
ICLOSE;
```

During input it is possible to check whether the end of the file has been reached by using EFILE. This will return <u>true</u> if the end has been reached, otherwise it will return <u>false</u>:

```
IF EFILE THEN ICLOSE ELSE CH[K←K+1]←READ
```

<u>Arguments for INFILE AND OUTFILE</u> may be given as shown above, i.e., as a separate string for device name, file name, and extension. Other

combinations of arguments are permitted, and the complete list is as follows:

```
           ["FNAME"] assumes device DSK, no extension
           ["FNAME", "EXTN"] assumes device DSK
           ["DEV", "FNAME", "EXTN"]
INFILE     ["DEV", "FNAME", "EXTN", PROJ, PROG] where PROJ and
OUTFILE              PROG are project and programmer numbers
           ["FNAME", "EXTN", PROJ, PROG]
           ["FNAME", PROJ, PROG]
```

It is of course permissible to use any string as device name, file name, or extension, although names that are too long will be truncated. The following program will write out successive cells of the list LTEXT as files called LTEXT.001, LTEXT.002, etc.:

```
FOR K←1 STEP 1 UNTIL LENGTH[LTEXT] DO
  BEGIN
    OUTFILE["LTEXT",STRING[[K//100+48,(K MOD 100)//10+48,K MOD 10+48]]];
    WRITE
    OCLOSE
  END
```

File input-output will work successfully for the following physical devices:

```
DSK
DTA0-DTA7
PTP  }
PTR  }   a dummy file-name and extension must
TTY  }   be given
```

5. Coping with Large List-Structures

Almost any program that makes use of lists will tend to produce large structures. This raises two problems:

a) It becomes very tedious to examine and debug these structures;
b) The program will eventually grow too big to be accommodated in core.

With these problems in mind, two features have been added to EULER. One is a library procedure call WRLIST. It will write out a text file listing all the contents of a named list, making it possible to examine the contents. It is called as follows:

```
WRLIST["DEV", "NAME", "EXT", LISTNAME]
```

LISTNAME is the variable containing the list. The resultant text file looks something like this:

```
[CELL1, CELL2, CELL3.....CELLN]
;
```

where CELL1, CELL2, etc., are the contents of each cell. These contents are written out in a format appropriate to the data type, for example:

```
[3.6, "ABC",
[TRUE, 0.0],
123]
;
```

This is the WRLIST output of the list structure shown on page 8.

Since WRLIST can output arrays, it provides a convenient means of dumping out the entire contents of an array without using FOR statements:

```
WRLIST["TTY", "X", "X", [A]]
```

There is currently no RDLIST procedure to read in the results of WRLIST. To cope with this need, and with the second problem mentioned above, the EULER interpreter has been extended to permit the reading and writing of lists in library format. The principal value of this is to permit the use of secondary memory for storing data, as follows:

SWAPOUT["XXX", L1];	will write out L1 onto the disk as a file called XXX.
L1←SWAPOUT["XXX", L1];	will do the same, and will reclaim the storage occupied by L1.
J←LENGTH[L1];	will cause L1 to be read back in from XXX.
L2←SWAPIN["XXX"]	will perform the equivalent of storing the original contents of L1 into L2. The actual operation will not take place until L2 is referenced, e.g., by LENGTH[L2].

Thus, after a list has been swapped out, it can still be accessed and modified as if it were in core--the very next access will automatically bring it back in. SWAPIN and SWAPOUT use the same file-name argument conventions as INFILE and OUTFILE, except for the additional final argument to SWAPOUT.

5. Send-Receive

If two EULER programs are running simultaneously, theycan communicate via send-receive. Send-receive permits programs to do the following:

a) to announce their name for the purpose of sending messages to and receiving them from other jobs;

b) to send a message to another job of known name;

c) to wait for a message from another job;

d) to determine whether a message has been received from another job, and if so to determine the name of the sender and the message contents.

A process name may be any text string. For example:

"WILLIAM"
"MASTERPROCESS"

are valid names for processes. A program announces its name to the outside world by the following procedure call:

```
DECLARE["JOE"]
```

After declaring its name, a program may send a message to another program whose name it knows:

```
SEND["PETE", MSG]
```

A message may be one of the following:

i) an integer in the range 0 to 250,000
ii) a text string
iii) a list

To receive a message, a program calls:

```
X←RECEIVE["JOE"]
```

This will be executed immediately, and will store into X:

a) an empty list, if no message has been received;
b) a two-element list, [sender, message], if a message has been received from the requested sender;
c) a single-element list, [sender], if the requested sender sent nothing, but another sender, whose name is now returned, has sent a message. A second RECEIVE is necessary to determine his message.

RECEIVE[0] will receive a message independent of its sender. In this case the name of the sending process may be not a string but a list containing two integers. If you wish the program simply to halt until a message is received, you may use RECWAIT. RECWAIT ["JOE"] will cause the program to halt until a message is received from process JOE. RECWAIT[0] waits until a message is received, irrespective of the sender. The values returned by RECWAIT and RECEIVE are identical.

As an example of the use of send-receive, suppose that we wish to allow two terminal users to send messages to each other without using the standard TALK facility of the PDP-10. The following program will handle each end of such a conversation:

```
TITLE SR
   BEGIN HISNAME, RUNNING;
   PRINT "TYPE YOUR NAME:";
   DECLARE [INTEXT];        \ declares typed string as
                              name of process \
   PRINT "TYPE DESTINATION:";
   HISNAME←INTEXT;
   PRINT "DO YOU WISH TO WAIT FOR A MESSAGE?";
```

```
                IF INTEXT # "Y" THEN    % send a message %
                    SEND[HISNAME,INTEXT];
                    FOR RUNNING TRUE WHILE RUNNING DO
                        BEGIN
                            % wait for a message %
                            PRINT RECWAIT[HISNAME][2]
                            % when received, print it %
                            SEND[HISNAME,INTEXT]
                            % send another %
                        END
            END
```

APPENDIX I.

BASIC OPERATORS

Symbol	Meaning	Operands	
+	unary plus	scalar	-
	addition	scalar	scalar
		matrix	matrix
-	unary minus	scalar	-
	subtraction	scalar	scalar
		matrix	matrix
*	multiplication	scalar	scalar
		matrix	matrix
		matrix	scalar
/	division	scalar	scalar
		matrix	scalar
//	integer division	scalar	scalar
↑	exponentiation	scalar	integer
MOD	modulus e.g., A MOD 3	scalar	scalar
ABS	absolute value e.g., A ← ABS B	scalar	-

> < >= <= = #	greater than less than greater than /equals less than /equals equals not equals	scalar string boolean	scalar string boolean
NOT	complement	integer	-
		boolean	-
AND	logical intersection	integer	-
		boolean	-
OR	logical union	integer	-
		boolean	-
&	concatenation	string	string
		list	list

<u>Precedence</u> is as follows, in descending order:

ABS + - (unary)

* / // MOD ↑

& + - (binary)

> < >= <= = #

NOT

AND

OR

APPENDIX II

EULER RESERVED PROCEDURES

ARRAY[M,N...]	creates an array with dimensions M,N...
EFILE	checks for end-of-file, returns true/false
EXIT	program returns to the Monitor
ENTIER[N]	makes an integer
ICLOSE	closes file after input
INFILE[...]	opens file for input
INPUT	inputs one character from teletype
LENGTH[L]	returns the length of a list, array, or string
LIST[N]	creates a list of N cells
OCLOSE	closes file after output
OUTFILE[...]	opens file for output
READ	reads one character from file
STRING[L]	converts list, array or integer to string
TAIL[L]	removes the first cell of a list
TYPE[V]	returns the type of argument V: -1 means undefined 0 means real 1 means integer 2 means boolean 3 means string 4 means array 5 means list
UNSTRING[S]	converts string to list

APPENDIX III.

EULER LIBRARY PROCEDURES

DECLARE SEND RECEIVE RECWAIT	See Part III., Section 6.
INSTRING	Reads in a text string from the teletype. e.g., LOOKUP[INSTRING] Terminating characters are space, period, = and carriage-return.
INTEXT	Identical to INSTRING, but carriage-return is the only terminator.
INVAL	Reads in one signed integer or floating-point number from the teletype. e.g., X←INVAL Terminating characters are comma, space or carriage return.
INVERT	Will invert a matrix e.g., M1←INVERT[M2]
MOUSE	Reads the status of the mouse or tablet next time a switch is pressed. Returns as value a five-element array, containing: in element 1: switch 1 setting (true or false) in element 2: switch 2 setting in element 3: switch 3 setting

in element 4: x-coordinate (integer in range 0-1023)

in element 5: y-coordinate (integer in range 0-1023)

Note that when the Sylvania tablet is in use, switches 1, 2 and 3 are turned on (=true) progressively in that order as the stylus approaches the tablet surface.

```
e.g., M←MOUSE;
      IF M[1] THEN (X←M[4]; Y←M[5])
```

RANDOM Returns a pseudo-random number in the range 0 to 1.0.

```
e.g., X←RANDOM
```

SIN, COS, ARCTAN Trigonometric functions. Angles are assumed to be in radians.

SMOUSE Identical to MOUSE, but the coordinates are scaled to lie in the range -1 to +1.

SQRT Square root function.

WRLIST Writes out a text file representation of any list; useful for debugging. The file may be written out onto the teletype.

```
e.g., WRLIST["DSK","FNAME","EXT",L];
      WRLIST["TTY","X","X",L]
```

These will write out the list L onto a disk file called FNAME.EXT and onto the teletype, respectively.

APPENDIX IV.

EULER D, SEULD

Before the appearance of EULER-G a very simple graphic package was implemented for Euler. This package is still available as part of a special interpreter called SEULD. Euler programs which use this system can be compiled by the standard EULER compiler.

The graphical commands are:

POS[X,Y]	% Position beam to absolute coordinates X, Y %
POINT[X,Y]	% Display a point at absolute coordinates X, Y %
LINE[X1,Y1,X2,Y2]	% Draw a solid line from absolute coordinates X1,Y1 to absolute coordinates X2,Y2 %
LINETO[X2,Y2]	% Draw a solid line from the present position of the beam to the absolute coordinates X2, Y2 %

All coordinates must be between 0 and 2047. The visible portion of the screen is the lower left quadrant of this area (0,0 to 1023, 1023). Arguments may be integers or floating-point numbers.

To display some text, one may use the command POS[X,Y] to position the beam, followed by DISPLAY X, Y, Z IN F where the DISPLAY statement has exactly the same syntax as the PRINT statement. The format F may be omitted. Characters whose ASCII code is less than 40_8 will be ignored.

The command CLEAR will clear the entire screen.

The commands POS, POINT, LINE, LINETO, CLEAR are implemented as external procedures.

APPENDIX V: LIST OF ERROR MESSAGES

```
004 CANNOT ROTATE 3-D PICTURES
005 "SIZE N" WILL NOT WORK, NO OF DIMENSIONS UNKNOWN
006 "SCALE N" WILL NOT WORK, NO OF DIMENSIONS UNKNOWN
007 NO END OF FILE ON CHARACTER SET
010 STRANGE, FRAME FILE HAS BEEN LOST
011 NO CHARACTER SET FILE FOUND ON DISK
012 CAN'T ENTER FRAME FILE
013 DISK INIT ERROR
014 OUTPUT ERROR TO STA
015 STATZ ERROR ON OUTPUT TO DSK
020 CALLING FRAME FROM WITHIN FRAME
021 PARAMETER LIST IN FUNNY SHAPE
022 DISPLAY PARAMETER NOT NUMERIC
023 DISPLAY PARAMETERS MUST BE A LIST
024 NO OF PARAMETERS MUST BE 2 OR 3
025 WINDOW OR VIEWPORT MUST BE SPECIFIED AS A LIST
026 NO OF PARAMETERS MUST BE EVEN
027 WRONG NO OF PARAMS FOR WINDOW OR VIEWPORT
030 TRANSFORMATION MUST BE AN ARRAY
031 TRANSFORMATION ARRAY MUST BE 2-DIMENSIONAL
032 TRANSFORMATION MATRIX MUST BE SQUARE
033 PERMISSIBLE SIZES FOR TRANSFORMATION ARE 2X2,3X3,4X4
037 ASTERISK OMITTED FROM FRAME PROCEDURE DEFINITION
040 VALUE LEFT FOR JUMP-ON-FALSE NOT BOOLEAN
041 WRONG NO OF WINDOW ARGUMENTS
042 WRONG NO OF POSITION ARGUMENTS
043 WRONG NO OF SIZE ARGUMENTS
044 WRONG NO OF SCALE ARGUMENTS
045 CANNOT DEFINE BOTH SIZE AND SCALE
046 ERROR IN PASSING DISPLAY ARGUMENTS
047 WRONG SIZE OF MATRIX FOR TRANSFORMATION
050 CANNOT ROTATE AND TRANSFORM IN SAME DISPLAY CALL
051 CANNOT ROTATE 3-D PICTURE
052 SCALXY AND RELSCALE DO NOT WORK YET WITH ROTATIONS
053 TRANSFORMATIONS ARE NOT PERMITTED IN FRAME PROCEDURES
054 "HIT" SHOULD HAVE 3 ARGUMENTS: X,Y,NAME
056 SCALE SHOULD NOT BE DEFINED IN FRAME PROCEDURE
057 ROTATIONS CANNOT BE DEFINED IN FRAME PROCEDURES
067 HUH? SINE RTN ASKED FOR SQRT OF NEG NO
070 THIS DISPLAY OPERATION NOT YET IMPLEMENTED, SORRY
072 LINE ARGUMENTS MUST BE PASSED AS LIST
073 WRONG NO OF LINE PARAMETERS FOR CURRENT NO OF DIMS
077 ILLEGAL INSTRUCTION CODE
100 TOO MANY BLOCK LEVELS
101 TOO MANY VARIABLES DECLARED AT THE SAME LEVEL
102 STACK IN ABNORMAL STATE AT "END"
103 DISPLAY REGISTER UNDERFLOW ON "END"
104 UNKNOWN DESCRIPTOR IN FETCHED VARIABLE
105 RETURN ADDRESS WORD FOUND IN PLACE OF VARIABLE
106 DOWN-POINTER FOUND IN PLACE OF VARIABLE
107 SAVED DISPLAY REGISTER FOUND IN PLACE OF VARIABLE
110 NEW TOP DISPLAY REGISTER FOUND IN PLACE OF VARIABLE
111 SPECIAL ARRAY DESCRIPTOR FOUND IN PLACE OF VARIABLE
112 NO ADDRESS POINTER FOUND ON STORE
200 NOT A PROCEDURE EXECUTING CALL
```

201 NUMBER OF PROCEDURE ARGUMENTS NOT INTEGER
202 TOO MANY ARGUMENTS IN CALL
203 NO RETURN ADDRESS FOUND ON RETURN FROM PROCEDURE
204 NO SAVED TOP DISPLAY REGISTER FOUND (P.RSTR)
205 CALL INSTRUCTION EXECUTED UNEXPECTEDLY
206 NO SAVED TOP DISPLAY REGISTER FOUND (P.PRIM)
210 NO DOWN POINTER FOUND ON STACK
301 CAN'T EXPONENTIATE BY NON-INTEGER
310 NON-NUMERIC ARGUMENTS FOR ADDITION
312 NON-NUMERIC ARGUMENTS FOR SUBTRACTION
314 NON-NUMERIC ARGUMENTS FOR MULTIPLICATION
315 DIVISION BY ZERO
316 NON-NUMERIC ARGUMENTS FOR DIVISION
317 NON-NUMERIC ARGUMENTS FOR EXPONENTIATION
336 MOD ERROR
340 "OR" ERROR
342 "AND" ERROR
344 "NOT" ERROR
346 NON-BOOLEAN IN BOOLEAN OPERATION
347 NON-BOOLEAN IN BOOLEAN OPERATION
350 NON-NUMERIC ARGUMENT FOR ABS
352 NON-NUMERIC ARGUMENT FOR COMPLEMENT
360 MATRIX OPERATIONS APPLY ONLY TO 2-DIMENSIONAL ARRAYS
361 ILLEGAL OPERATION ON MATRICES
362 MATRICES WRONG SIZES FOR MULTIPLY
363 MATRIX CONTAINS NON-NUMERIC DATA
364 MATRICES OF DIFFERENT SIZES, CANNOT BE ADDED OR SUBTRACTED
377 UNIMPLEMENTED MATRIX OPERATION
500 NUMBER OF ITEMS IN OUTPUT LIST NOT INTEGER
501 ILLEGAL TYPE OF ITEM IN OUTPUT LIST
502 WRONG FORMAT FOR STRING OUTPUT
503 TOO FEW ITEMS IN OUTPUT LIST COMPARED WITH NUMBER OF FIELDS
540 WRONG NUMBER OF ARGUMENTS IN A POS,LINE OR POINT CALL
541 ARGUMENT OF POS,LINE OR POINT IS NOT A NUMBER
542 POS,LINE OR POINT WAS CALLED WITHOUT ARGUMENTS
551 DEVICE NOT AVAILABLE
552 NO FILE NAME GIVEN
553 FILE NOT FOUND
554 TOO MANY NEW VARIABLES OR FORMALS
555 TOO MANY BLOCK LEVELS
557 TOO MANY EXTERNAL VARIABLES
600 ILLEGAL INSTRUCTION FORMAT FOR ARRAY DEFINITION
601 ARRAY DIMENSION VALUE NOT AN INTEGER
602 WRONG TYPE OF VARIABLE USED FOR ARRAY CALL
603 NON-INTEGER USED IN ARRAY DEFINITION
604 NUMBER OF ARRAY DIMENSIONS NOT AN INTEGER
605 INCORRECT NUMBER OF ARRAY DIMENSIONS
606 ARRAY DIMENSION VALUE NOT AN INTEGER
607 SUBSCRIPT NOT AN INTEGER
610 SUBSCRIPT OUT OF RANGE
700 BAD DYNAMIC VARIABLE PASSED TO INTERPRETER
701 SYS ERR - BAD NUMERIC FORMAT
702 FILE NAME ENTER ERROR
703 FILE NAME LOOKUP ERROR
704 OUTPUT DEVICE INITIALIZATION ERROR

```
705 INPUT DEVICE INITIALIZATION ERROR
706 BAD FILE NAME FORMAT
707 OUTPUT CLOSE ERROR
710 INPUT CLOSE ERROR
711 OUTPUT ERROR
712 INPUT ERROR
713 WRONG NUMBER OF PARAMETERS PASSED TO LIST
714 WRONG NUMBER OF PARAMETERS PASSED TO LIST
715 BAD LIST PASSED TO INTERPRETER
716 INDEX TOO SMALL FOR LIST
717 INDEX TOO LARGE FOR LIST
720 WRONG NUMBER OF PARAMETERS PASSED TO SRPROC
721 WRONG NUMBER OF PARAMETERS PASSED TO TAIL
722 WRONG TYPE OF PARAMETERS PASSED TO CONCATINATION
723 WRONG TYPE OF PARAMETERS PASSED TO UNSTRING
724 RESERVED PROCEDURE EXPECTED ONE PARAMETER
725 WRONG TYPE OF PARAMETERS PASSED TO STRING
726 WRONG TYPE OF ARRAY PASSED TO STRING
727 RESERVED PROCEDURE EXPECTED AN INTEGER
730 WRONG TYPE OF PARAMETERS PASSED TO SRPROC
731 RUN UUO RETURNED
732 ENTIER TAKES ONLY ONE PARAMETER
733 ENTIER TAKES ONLY REAL, INTEGER, OR BOOLEAN
734 BAD ARGUEMENTS TO COMPARE I.E. =,<,>,#,>=,<=
776 SYS ERR - JUMP IS TOO LARGE
777 SYS ERR - DICTIONARY OVERFLOW
END LEAVE THIS AS LAST LINE PLEASE: ALL FOLLOWING LINES IGNORED
```

APPENDIX VI

EULER-G

This appendix describes some extensions which have been made to PDP-10 Euler to permit interactive graphics. The extended language, called Euler-G, should not be confused with the library of graphical procedures in SEULD.

Euler-G uses many of the ideas first proposed in Dial [3]. These include the concept of display procedures, and the assumption that all pictures are scaled before being displayed. Euler-G also contains additional features such as the means to specify rotations and viewports and the ability to display projections of three-dimensional objects. These features should make Euler-G considerably more useful than Dial.

Euler-G produces display files for the Univac 1559. These files are created first in a device-independent format, which is converted to 1559 format by a separate transmission program. It is therefore extremely easy to convert Euler-G so as to output to other devices. Work has already begun on a plotter output package.

Basic Graphical Operations

Graphical output is generated by means of a small set of primitives. The most important primitives are the following:

MOVE TO P	: move the display beam to point P
MOVE D	: move the display beam through a distance D from its current position
LINE TO P	: draw a line from the current beam position to point P

```
LINE D              : draw a line of length D from the
                          current beam position

DISPLAY T1, T2... : display text items T1, T2...
                          at the current beam position
```

The remaining primitives are variants of LINE TO and LINE which produce different line textures:

```
ZIP TO P  }    the corners between lines are slightly
ZIP D     }    rounded.  Useful for drawing curves.

DOT TO P  }    to draw dotted lines
DOT D     }
```

All points and distances must be specified by <u>lists</u>. These lists must have either two or three elements, depending on whether two-dimensional or three-dimensional objects are being defined. For example:

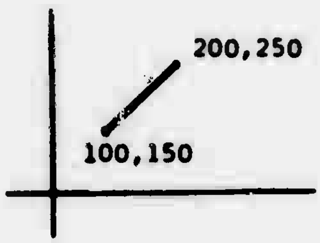

Figure 1

```
MOVE TO [100,150]; LINE TO [200,250]
```

will draw a line from (100,150) to (200, 250) as shown in Figure 1.

Instead of a list, the name may be used of any variable which currently contains a list. For example:

```
L3 ← [10.0, 3.7];
LINE L3;
```

or:

```
MOVE TO POINTLIST[1];
FOR K ← 2 STEP 1 UNTIL LENGTH[POINTLIST] DO
     LINE TO POINTLIST[K];
```

The second example will produce a sequence of connected lines, such as is shown in Figure 2(b), from a list POINTLIST containing their coordinates in the format shown in Figure 2(a).

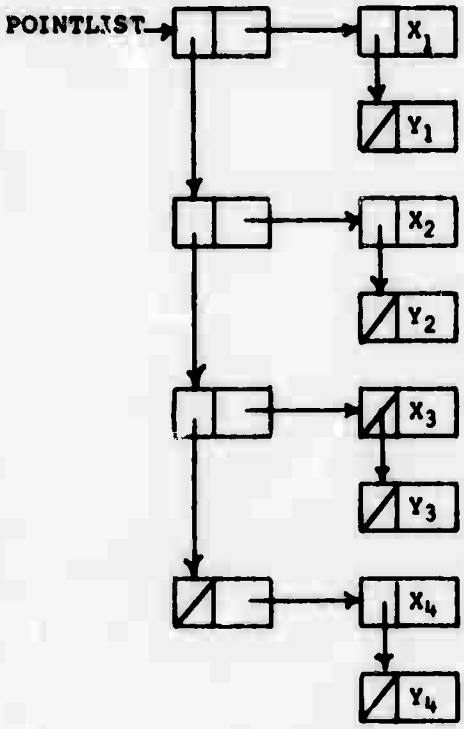

Figure 2(a)

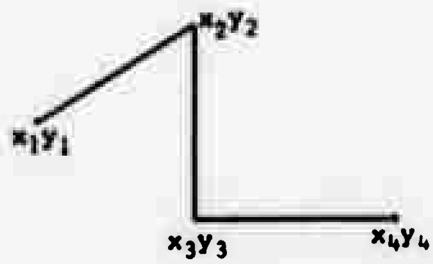

Figure 2(b)

The DISPLAY statement is modeled on the Euler PRINT statement, and produces on the display the same output that PRINT produces on the teletype. The same formatting technique is used:

```
DISPLAY A, B IN "A =\\\  B =\\.\\\ "
```

Page Coordinates and Screen Coordinates

Most display programming systems force the user to define pictures in a fixed coordinate system, the coordinate system of the display screen. This is not the case with Euler-G. Instead pictures are defined in what is called the page coordinate system, and are displayed by transforming the appropriate parts of the picture into screen coordinates.

The programmer has a great deal of freedom to specify: (a) the region of the page which he wishes to see on the screen, (b) the transformation which he wishes to apply to the picture, and (c) the region of the screen which he would like the picture to occupy. This does not imply that he always has to take advantage of all this flexibility. The normal procedure is to specify a rectangular window onto the page information, and a rectangular viewport onto the screen. Figure 3 shows an example. All the page information lying within the window will appear on the screen within the viewport; everything else is eliminated. The statement to define this transformation is the display procedure call, of which the following is an example:

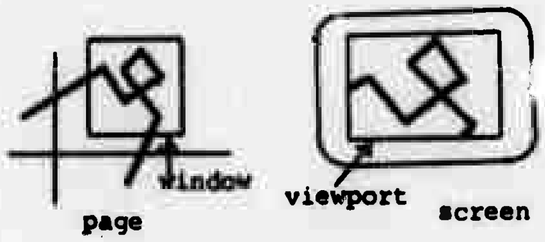

Figure 3

POINTS WITHIN [200,200,100,100] ONTO [0,0,1,1]

POINTS refers to a procedure, which might well be the example given and shown in Figure 2. The window onto POINTS is specified by WITHIN [200,200,100,100], which means that the center of the window is at (200,200) and that it measures 100 page coordinate units in size, measured from the center to each edge. The viewport has its size from this point. This is the full screen size: rather than use the particular coordinate system of the Univac 1559, Euler-G assumes the screen dimensions to be those shown in Figure 4.

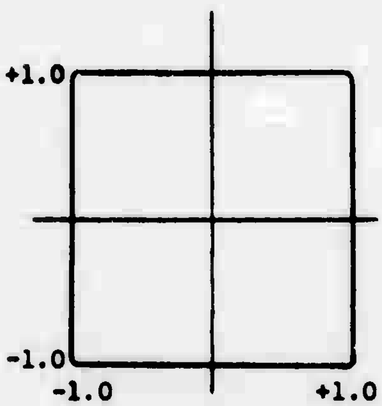

Figure 4

The complete-sequence of instructions for generating a connected-line picture might be the following:

```
FRAME1 ← *'POINTS WITHIN[200,200,100,100]
               ONTO [0,0,1,1]'
POINTS ← 'BEGIN NEW K;
        MOVE TO POINTLIST[1]
        FOR K←2 STEP 1 UNTIL LENGTH[POINTLIST] DO
            LINE TO POINTLIST[K]
        END';
FRAME1;
```

Note the asterisk preceding the body of the procedure FRAME1. Any display procedure which is not itself called from another display procedure should include this special mark, indicating that it is a frame procedure. Frame procedures have a number of special properties. In the first place, they allow the picture on the screen to be composed of a number of logically separate parts, each of which can be altered or removed without affecting the others. A frame can be removed by means of the DELETE statement:

```
DELETE FRAME1
```

It can be altered by changing the data which it accesses, and then calling it again. For example, if we changed the contents of the list POINTS, and then called FRAME1. we should see a new picture representing the new contents of POINTS. Alternatively the window might be changed in order to show a different part of the complete picture.

Display Procedure Calls

It may be useful to call display procedures to several levels. For example, we might wish to define a symbol that appears repeatedly in a certain picture. Figure 5 shows a symbol commonly used to indicate wind velocity and direction in weather maps. We could define this as a display procedure called WINDSYM, and create a 'weather map' by means of the following statement:

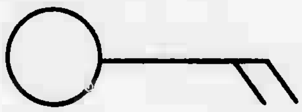

Figure 5

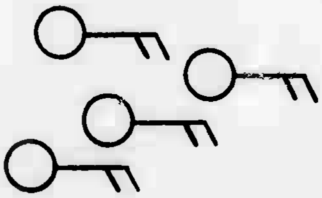

Figure 6

```
FOR K←1 STEP 1 UNTIL LENGTH[STATIONS] DO
        WINDSYM AT STATIONS[K]
```

This assumes that the position of each weather station is held in a list called STATIONS. The result will be a picture such as Figure 6. We can add a rotation to each symbol as follows:

```
FOR K←1 STEP 1 UNTIL LENGTH[STATIONS] DO
        WINDSYM AT STATIONS[K] ROT WD[K]
```

WD is a list containing the wind directions, measured in radians.

Arguments may be passed to display procedures. The number of 'bars' on a symbol could be held in a list called BARS and passed as follows:

```
FOR K←1 STEP 1 UNTIL LENGTH[STATIONS] DO
 WINDSYM[BARS[K]] AT STATIONS[K] ROT WD[K]
```

The definition of WINDWYM might look something like the following:

```
WINDSYM ← 'FORMAL N;          % N IS NUMBER OF BARS %
          BEGIN NEW K;
           CIRCLE WITHIN [0,0,1,1] SIZE 1 AT [0,0];
           MOVE TO [1,0];
           LINE TO [5,0];
           FOR K←1 STEP 1 UNTIL N DO
              (LINE [1,0];LINE[1,-1]; MOVE [-1,1])
          END'
```

CIRCLE is yet another display procedure, possibly written as an external procedure.

The complete range of transformations and other arguments which may accompany a display procedure call are as follows:

Window:	WITHIN + 4-element list
Viewport:	ONTO + 4-element list
Position:	AT + 2-element list
Size:	SIZE + 2-element list or scalar
Scale:	SCALE + 2-element list or scalar
Rotation:	ROT + scalar
Transformation:	TRANS + 2x2 or 3x3 array
Name:	AS + integer or real number

They may be listed in any order. ONTO [a,b,c,d] is equivalent to AT[a, b]SIZE[c,d]. If both size dimensions are the same, a single scalar may be used; the same applies to SCALE. Rotations are measured anti-clockwise in radians. <u>Names</u> have no effect on the picture: they are for use in detecting mouse hits and so forth.

<u>Windows</u> play an important part in reducing processing time. Suppose we have defined the weather map shown in Figure 7, and wish to view just the portion shown by the dotted outline. The program shown

above will test every line of every symbol for visibility, and discard those outside the window. If there are a lot of invisible symbols this will take a lot of time.

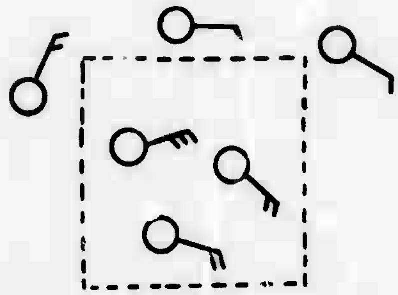

Figure 7

We can reduce this time by specifying a window around the symbol:

WINDSYM[BARS[K]] WITHIN [0,0,10,10] AT ... etc.

This implies that we are only interested in the information within the boundary shown in Figure 8(a), and the program can immediately eliminate those symbols whose boundaries lie entirely outside the main window. In Figure 8(b) this would mean the upper three symbols.

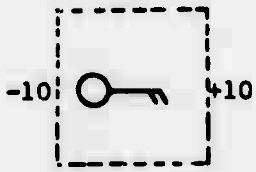

Figure 8(a)

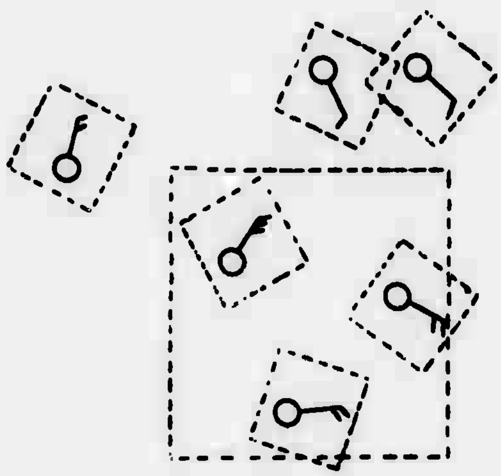

Figure 8(b)

The Use of Names

Names are useful principally for pointing with the mouse. The reserved procedure HIT[x,y] will return a value true or false according to whether any lines or text lie within a small distance of (x,y) on the screen. Usually this information on its own is of little use: we need to know which item lies at (x,y). This is why names are useful. If, for example, we would like to point at one of the wind symbols on our weather map, we should call each symbol with a unique name:

```
FOR K←1 STEP 1 UNTIL LENGTH[STATIONS] DO
        WINDSYM AS K AT ...etc.
```

When HIT returns a value true, the name of the symbol we were pointing at is in HITNAME.

The x and y values are normally the x and y coordinates of the mouse, in screen coordinates. To determine these values, use the library procedure SMOUSE. This returns a five-element list as its value each time one of the mouse switches is pressed:

```
M ← SMOUSE;
  % M[1] IS TRUE IF SWITCH 1 WAS PRESSED, OTHERWISE FALSE
    M[2] AND M[3] CONTAIN THE SAME
    INFORMATION FOR SWITCHES 2 AND 3.
    M[4] AND M[5] CONTAIN X AND Y
    IN THE RANGE -1 TO +1  %
```

Usually when HIT is used we would like to restrict its scope to a certain part of the picture. This can be done by passing a name to HIT: this is the name of the procedure call one level above the symbols at which we are pointing. So if we are going to point at windsymbols, we should pass HIT the name of the call to the whole weather map:

```
MAP ← 'BEGIN NEW K;
         FOR K←1 STEP 1 UNTIL LENGTH[STATIONS] DO
           WINDSYM AS K WITHIN [0,0,10,10] ROT WD[K]
               AT STATIONS[K]
       END';

FRAME ← *'MAP WITHIN W ONTO V AS 100';
```

```
FRAME;
M ← SMOUSE
  IF HIT[M[4],M[5], 100] THEN ...
            % HITNAME NOW CONTAINS THE SYMBOL NUMBER %
```

A second use for names is in converting from screen coordinates to page coordinates. This can be done with the reserved procedure SCALXY. For example

```
SCALXY[X,Y,100]
```

would return the position, in the coordinate system of MAP, corresponding to (X,Y) in screen coordinates. For obscure reasons, SCALXY will not return correct values unless the frame containing the procedure call in question has been called at least once.

A reserved variable which may be accessed within a display procedure is RELSCALE. It returns as value the relative scale on the screen of the current "instance" of this procedure, i.e., the ratio of page units to screen units. It returns a list if the scales in the x- and y-directions are different.

Display Procedure Call Syntax

The syntax of display procedure calls permits any sequence of statements within parentheses to be used in place of a procedure variable name. For example, the following is a permissible display procedure call:

```
(MOVE  TO[20,0]; LINE TO [30,30]) AT [X,Y] SCALE 5
```

This form may be convenient for such things as displaying text messages:

```
FTEXT ← *'(DISPLAY "START") AT [-.1,.9] WITHIN [0,0,100,100]';
FTEXT;
```

This will display the message "START" near the top center of the screen.

The display procedure call syntax also requires that all display procedures called from a frame procedure are called with a window or a viewport specified.

Displaying Three-dimensional Data

LINE, LINE TO, MOVE, etc. may specify three coordinates instead of two. In this case the third is treated as a z-coordinate. Three-dimensional information may be transformed in the same way as two-dimensional, with the restriction that rotation cannot be specified by ROT. Windows and viewports, other than the final window and viewport specified in the frame procedure, should have six arguments instead of four; scale, size and position lists should contain three elements; and transformation matrices should be 3x3 or 4x4. SCALXY will not work on three-dimensional data.

How to Use Euler-G

A special Euler-G compiler has been written, and can be run as follows:

```
.R EULERG
*PROG←PROG
```

This assumes the source file of the user's program is on the disk under the name PROG.SRC. To run the program, type:

```
.R SEULG
*PROG
```

The debugging features of EULER are all included in Euler-G.

Summary

F ← *'......'	defines a frame procedure
DP ← '......'	defines a display procedure
F	calls a frame procedure
DELETE F	deletes it
CLEAR	clears the screen
DP	followed by one or more transformations calls a display procedure. Transformations allowed are:
WITHIN [x,y,w,h]	window, center (x,y), size 2wx2h
ONTO [x,y,w,h]	viewport, center (x,y), size 2wx2h
AT [x,y]	position
SIZE [w,h]	size 2wx2h
SIZE s	size 2sx2s
SCALE [w,h]	scale wxh
SCALE s	scale sxs
ROT r	rotated r radians anti-clockwise
TRANS t	transformed by matrix t
AS n	name n
MOVE TO [x,y]	move beam to (x,y)
MOVE [dx,dy]	move beam through distance dx,dy
LINE TO [x,y]	draw line to (x,y)
LINE [dx,dy]	draw line of length dx,dy
ZIP TO [x,y]	like LINE TO, zip mode
ZIP [dx,dy]	like LINE, zip mode
DOT TO [x,y]	dotted LINE TO
DOT [dx,dy]	dotted LINE
DISPLAY t1,t2	display text items t1,t2
DISPLAY t1,t2 IN f	display text in format f
HIT [x,y,n]	look for hit under call n at screen position (x,y), return <u>true</u> or <u>false</u>; return name in HITNAME
SCALXY[x,y,n]	scale (x,y) from screen coordinates to page coordinates for call n.
RELSCALE	returns relative scale of current display procedure

APPENDIX VII: EUL**E**R Compiler Error Messages

Syntax Error 1: Illegal title

2: Outermost block must include declarations

3: Illegal declaration list

4: Illegal formal variable list

5: Not a valid statement

6: Illegal statement terminator

7: Illegal subscript list

8: Integer must follow period

9: Illegal statement terminator

10: No begin or quote following title

11: Illegal item in declaration

12: Illegal variable following for

13: Only unsubscripted variable names allowed in declarations

14: For statement expects ←

15: No expression following ← in for statement

16: Illegal expression following step

17: Illegal expression following until

18: Illegal expression following while

19: Either until or while must be included in stepped for statement

20: Illegal expression as operand to arithmetic test

21: No do in for statement

22: Illegal operand for arithmetic binary operator

23: Illegal expression following if

24: No then in if statement

25: Illegal expression as operand for not

26: Illegal operand following unary + or -

27: Illegal statement as item in output list

28: Illegal item used as format

29: Illegal expression as operand for or

30: Illegal expression as operand for and

System Error 8: Null string, not permitted (Use " 'Z ")

127: String extends over more than one line, not permitted (Use 'N)

End of File Input: Compiler reached end of file without finding final end or quote.

Stack Overflow: Too many nested blocks

APPENDIX VIII

LINKING ASSEMBLY CODE TO EULER PROGRAMS

Assembly code may be linked to EULER programs by creating user procedures. There is provision for up to ten user procedures. They are called UP0 through UP9. These procedures may or may not have parameters, but they must return a value. There is also a facility to initialize user procedures when the program starts.

1. Empty User Procedure Macro Source

An empty user procedure macro source called UPROC.MAC is available from the system programmers. This file has the necessary linkage declarations, accumulator and special symbol definitions, and macro definitions. This file should be used to create user procedure files. A copy of UPROC.MAC is included at the end of this appendix.

2. Accumulator Usage

Accumulators which have names starting with T or FREE may be used without the user having to save them. All others in general should not be touched, except as described below. Accumulator 0 should never be used because the macros use it.

3. User Procedure Initialization

When a program starts, control is transferred to UPI$$$. The user may do whatever initialization is necessary and then return control to the interpreter by executing a RET instruction.

4. User Procedures

When a user procedure is called, control is transferred to UPn$$$. When the procedure is complete, control is returned to the interpreter by executing a JRST I.RET if there are no parameters, or a JRST I.BRET if there are parameters.

5. Parameters to User Procedures

Parameters to the user procedures are passed on the WP stack. The value at (WP) is the number of parameters as an integer. (See Appendix IX for data formats.) The value at -1(WP) is the n^{th} parameter through -n(WP) which is the first parameter.

Before the procedure returns control to the interpreter, it should execute the instruction CAL B.PEEL once for each parameter value and once more for the parameter count value.

6. Determining the Data Type of a Value

Two macros are provided to allow the program to determine the data type of EULER values. They are SKDE and SKDN, SKp Descriptor Equal, and SKp Descriptor Not Equal, respectively. The format is:

SKDE address of value, type of value desired.

The types of interest are:

UNDEFINED	D.UNDF
INTEGER	D.INT
REAL	D.FP
BOOLEAN	D.BOOL

STRING	D.STR or D.TSTR
ARRAY	D.ARR or D.TAR
LIST	D.LIST or D.TLST

See Appendix IX for data formats.

7. Returning Values

After the proper number of calls on B.PEEL the procedure must put its return value on the stack. This is done by the following code:

```
MOVE AC, value
STACK AC.
```

If the procedure wishes to return an undefined value, the code would be:

```
MOVE AC, [D.UNDF]
STACK AC.
```

8. Internal Subroutines

Internal subroutines may be called by CAL subroutine and the subroutine will return with a RET.

9. Saving Accumulators on the Stack

AC's may be saved on the stack by SAVE AC and restored by FETCH AC.

10. Free Storage

To get a block of free storage N word long, the following code is used:

```
MOVEI TAC, N
CAL  S.GET
```

TAC now contains a pointer to the block.

To return a block to free storage, the following code is used:

```
MOVE TAC, ptr to block
CAL  S.RETS
```

APPENDIX IX

DATA FORMATS

All EULER values have special formats.

1. <u>Integers</u>

The following code converts an EULER integer into a PDP-10 integer:

```
LSH AC, 2
ASH AC, -2
```

The following code converts a PDP-10 integer into an EULER integer:

```
TLO AC, 400000
TLZ AC, 200000
```

2. <u>Reals</u>

The following code converts an EULER real into a PDP-10 real:

```
LSH AC, 1
```

The following code converts a PDP-10 real into an EULER real:

```
LSH AC, -1
```

3. Booleans

Bit 35=1 is true, and Bit 35=0 is false.

4. Strings

The right half is a pointer to a ASCIZ block of characters.

5. Arrays

The right half is a pointer to an array.

Array format is:

Word 0:	D.INT,,	number of dimension
Word 1:	D.DOPE,,	size dimension n
⋮		
Word n:	D.DOPE,,	size dimension 1
Word n+1:	value [1,...,1,1]	
Word n+2:	value [1,...,1,2]	
⋮		
Word $n+\prod_i D_i$	value $[D_1,D_2,...,D_n]$	

6. Lists

The right half is a pointer to a list header:

List header

word -1	D.DUBB	LENGTH
word 0	N	FIRST ELEMENT PTR
word 1	N^{th} ELEMENT PTR	LAST ELEMENT PTR

List Element

word 0	D.SINB	NEXT ELEMENT PTR
word 1	VALUE	

```
EXTERN JOBUUO,JOB41,JOBREL,JOBDDT,JOBSA,JOBREN,JOBAPR
EXTERN  JOBCNI,JOBTPC,JOBOPC
EXTERN  S.RET,S.RETS,S.COPY,S.GET,Q.HEAD,I.RET,R.PEEL,I.SRET
WP=1     ;WORKING STACK
IB=2     ;INST. BYTE PTR
XP=3     ;PUSH-JUMP STACK
AR=4     ;ADDR. REG.
LVL=5    ;BLOCK LEVEL OF LATEST FETCH
T1=6
T2=7
T3=10
TAC=11
T4=12
T5=13
T6=14
LP=15
FREEP1=16         ;NOT USED PRESENTLY-NOT PERSISTENT
FREEP2=17         ;NOT USED PRESENTLY-NOT PERSISTENT
OPDEF   RET      [POPJ XP,]
OPDEF   CAL      [PUSHJ XP,]
OPDEF   STACK    [PUSH WP,]
OPDEF   SAVE     [PUSH XP,]
OPDEF   FETCH    [POP XP,]
OPDEF   ERROR    [1B8]
DEFINE SKDE (LOC,DESC)  <
IFE DESC,          <
        SKIPGE LOC
>
IFE DESC-1B0,    <
        MOVE LOC
        SKIPGE
        TLNE 200000
>
IFE DESC&3B1-3B1,         <
        MOVE LOC
        TLZ 774000
        IOR [DESC]
        CAME LOC
>>
DEFINE SKDN (LOC,DESC)  <
IFE DESC,          <
        SKIPL LOC
>
IFE DESC-1B0,    <
        MOVE LOC
        SKIPGE
        TLNE 200000
        SKIPA
>
IFE DESC&3B1-3B1,         <
        MOVE LOC
        TLZ 774000
        IOR [DESC]
        CAMN LOC
```

```
>>
D.FP=0
D.INT=1B0
D.BOOL=3B1
D.LIST=141B6
D.TLST=161B6
D.ARR=142B6
D.STR=143B6
D.PROC=145B6
D.RA=146B6
D.REF=147B6
D.SDR=151B6
D.BKPT=150B6
D.SRA=146B6
D.STDR=152B6
D.DOPE=153B6
D.SUBR=154B6
D.EXTP=155B6
D.PROP=144B6
D.TAR=162B6
D.TSTR=163B6
D.FSBK=170B6
D.SINB=171B6
D.DUBB=172B6
D.BADB=173B6
D.UNDF=177B6
D.UND=177B6
TITLE UPROC
INTERN  UP0$$$,UP1$$$,UP2$$$,UP3$$$,UP4$$$
INTERN  UP5$$$,UP6$$$,UP7$$$,UP8$$$,UP9$$$,UPI$$$
UPI$$$: RET
UP0$$$:
UP1$$$:
UP2$$$:
UP3$$$:
UP4$$$:
UP5$$$:
UP6$$$:
UP7$$$:
UP8$$$:
UP9$$$: TTCALL 3,[ASCIZ/
NO USER PROCEDURES DEFINED
/]
        CALL [SIXBIT/EXIT/]
        END
```

REFERENCES

Naur, P. (Ed.) "ALGOL 60: revised report on the algorithmic language". *Communications of the ACM*, Vol. 6 1-17 (Jan. 1963)

Newman, W. M. "An experimental display programming language for the PDP-10 computer". University of Utah, Computer Science Technical Report, UTEC-CSc-70-104 (June, 1970)

Wirth, N. and Weber, H. "Euler: a generalization of Algol and its formal definition". *Communications of the ACM*, Vol. 9 13-25+ and 89-100 (Jan. and Feb., 1966)