9. INPUT/OUTPUT

9.1. GENERAL

Input and output operations are accomplished in UNIVAC 1108 ALGOL by means of library procedures. The two main ones, READ and WRITE, are more flexible than ordinary procedures written in ALGOL because the number of parameters in an actual call or even the order of the parameters is not rigidly specified. In general, when doing I/O with other than cards or printer, recursive calls should be avoided. An important feature of the I/O is that there are degrees of complexity that can be used. It is very simple to insert statements into the ALGOL program just to dump a few variables when debugging. More refined output for a working program can be done later.

The two other important I/O procedures, POSITION and REWIND, are concerned exclusively with magnetic tape and tape-simulated drum operations.

9.2. FREE-FORMAT OUTPUT ON PRINTER AND CARD PUNCH

Arrays and values of expressions can be printed by simply calling the WRITE procedure in the following way:

WRITE(PRINTER, v_1,v_2,..., v_n)

where each v_i is an expression or an array identifier. In a similar manner, if the output is to be punched, the call is:

WRITE(PUNCH, v_1, v_2, . . . , v_n)

PRINTER and PUNCH are device names which specify the output unit to be used. If no device is named, PRINTER is assumed:

WRITE(v_1, v_2, . . . , v_n)

In the following description the word 'print' v is used in discussing the action of WRITE, but the word 'punch' may be substituted. Significant differences between the two devices are noted.

The action of WRITE is to evaluate the expressions in the order they are listed in the call and print their values in the following manner: except for string expressions, 10 values are printed on each line (6 per card if punching). Each value occupies a field of 12 character positions or columns. If the actual parameter is an array it is decomposed by columns. Each occurrence of WRITE begins printing on a new line.

Examples:

```
REAL ARRAY A(1:10) $
•
•
WRITE(PRINTER,A)$
```

acts the same as

```
WRITE(PRINTER,A(1),A(2),...,A(10))
```

For multidimensional arrays the decomposition is such that the leftmost subscript varies most frequently. Thus the sequence

```
BOOLEAN ARRAY A(1:2,1:2,1:2) $
•
•
WRITE(PRINTER,A) $
```

acts like

```
WRITE(PRINTER,A(1,1,1),A(2,1,1),A(1,2,1),A(2,2,1),
               A(1,1,2),A(2,1,2),A(1,2,2),A(2,2,2)) $
```

The expression or array element is printed in a form consistent with its type.

Type	Form
INTEGER	Integer form, right justified in the field. Includes a leading minus if the expression is negative. Leading zeros are not printed.
REAL and **REAL 2**	Both types are printed right justified in the form X.XXXX,±NN, where NN represents the power of ten. If the number is negative, NN is preceded by a minus sign.
BOOLEAN	Either **TRUE** or **FALSE** is left justified in the field.
COMPLEX	The real and imaginary parts are each given a field as for **REAL**. Thus, only five expressions of type **COMPLEX** can be printed on the same line.

Strings are a slight exception in that they always start a new line. Whenever a string expression occurs as a parameter, the previous expressions, whether their number is a multiple of 10 or not, are printed. Then the string is printed on a new line. The next parameter will be printed on the following line. For example, if A and B are **REAL** and have the values 7.0 and 0.004 respectively, then the statement

```
WRITE('A=',A,'B=',B,'A OVER B',A/B)
```

would produce the following lines on the printer:

```
A=
 7.0000, 00
B=
 4.0000,-03
A OVER B
 1.7500, 03
```

Example:

```
INTEGER ARRAY A(1:15) $
BOOLEAN ARRAY B(1:2,1:2) $
INTEGER I,J $
.
.
FOR I=(1,1,15)DO A(I)=I-3 $
FOR I=(1,1,2) DO FOR J=(1,1,2) DO B(I,J)=I GEQ J $
WRITE('VECTOR A',A,'MATRIX B',B) $
```

produces the following output:

```
VECTOR A

          -2          -1           0           1           2           3
           8           9          10          11          12          13
MATRIX B
TRUE        TRUE        FALSE       TRUE
```

12 12 12 12 12 12

9.3. FREE-FORMAT INPUT FROM CARDS

For reading punched cards in free-format mode, the procedure READ is called with device CARDS:

READ(CARDS, $v_1, v_2, \ldots, v_n$)

where each v_i is a variable or array identifier. Again, if no device is written, CARDS is assumed:

READ($v_1, v_2, \ldots, v_n$)

This procedure reads the next input card and scans the information on it. Each constant on the card is assigned to the next parameter in the order it appears in the call.

Arrays are handled in the same manner as for WRITE. Constants on the cards must be punched in the same form as they appear in the ALGOL source language (see 2.3) with the exception that a comma(,) may be used in place of the ampersand (&). Constants on a card are delimited by one or more blanks and by the end of the card. Therefore, there is no restriction as to where a constant may appear on the card. If there is not enough information on the first card to satisfy the READ procedure, a second card is read, and so on. Any information not taken from the last card is lost (i.e., the next call to READ reads a new card). An * punched on a card causes the remainder of the card to be ignored. Otherwise, all 80 columns of the card are scanned for information. An example of a call on READ:

```
REAL A,B $
INTEGER COUNTER $
•
READ(CARDS,A,B,COUNTER) $
```

Data Card

```
-7.2    .099    362236
```

assigns the values –7.2 to A, .099 to B and 362236 to COUNTER. It is not necessary for the type of the constant on the card to match the type of the actual parameter. Transfer functions are used automatically if such functions are defined (see Appendix B).

9.4. LIST PARAMETERS – THE **LIST** DECLARATION

It is possible to supply more general kinds of parameters to the READ and WRITE procedures than those we have mentioned in Sections 1 and 2. The following example is a method for printing only the third row of a two dimensional matrix A:

```
REAL ARRAY A(1:N,1:N) $
INTEGER I $

FOR I=(1,1,N) DO WRITE (A(3,I)) $
```

It can also be done the other way around:

```
WRITE (FOR I=(1,1,N) DO A(3,I)) $
```

which has the same effect, except for the paper spacing. The parameter defined by the statement

```
FOR I=(1,1,N) DO A(3,I)
```

is called a list and is acceptable any time as a parameter to READ or WRITE. This construction can be used as an actual parameter to any procedure (the formal parameter specified as **LIST**), but ultimately can be used only by a procedure outside the ALGOL language such as READ or WRITE which are in assembly language. The procedure MAX and MIN can also be called with this type of list parameter. As would be expected, the **FOR** clauses in a list can be nested:

```
REAL ARRAY A(1:N,1:N) $
INTEGER I,J $
•
•
READ (FOR I=(1,1,N) DO FOR J=(1,1,N) DO A(I,J)) $
```

reads a matrix from cards by rows instead of the usual columns.

The elements of the matrix could be printed in like manner with the same parameter used to call the WRITE procedure:

```
WRITE (FOR I=(1,1,N) DO FOR J=(1,1,N) DO A(I,J))
```

but this is laborious. The **LIST** declaration is for this purpose:

```
REAL ARRAY A(1:N,1:N) $
INTEGER I,J $
LIST L1(FOR I=(1,1,N) DO FOR J=(1,1,N) DO A(I,J)) $
•
•
READ(L1)$
WRITE(L1) $
```

The **LIST** declaration, like all declarations, must be placed at the beginning of a block. A **LIST** merely defines an ordered sequence of expressions:

```
REAL X,XY $
INTEGER WIDDLE $
BOOLEAN ARRAY Q (1:10) $
STRING BEAN (36*5) $
INTEGER I $
LIST L(1.0,X,XY,FOR I=(1,1,5)DO(Q(I),BEAN(36*(I-1)+1,36)),
     SQRT(WIDDLE**3)) $
```

The **LIST** L defines the following sequence of expressions:

```
1.0 X XY Q(1)  BEAN(1,36) Q(2) BEAN(37,36) Q(3) BEAN(73,36)
Q(4) BEAN(109,36) Q(5) BEAN(145,36) SQRT(WIDDLE**3)
```

Note that the **LIST** defines only the expressions; the values are not defined until the **LIST** is activated (that is, until it is actually used in a procedure call). The identifiers used in the expression must be defined prior to their inclusion in the **LIST** declaration. Thus, the other declarations should precede the **LIST** declaration.

A **LIST** may contain as elements the following three things, separated by commas:

- Expressions
- Array identifiers
- Lists defined by **FOR** clauses

It should be noted that **BEGIN** and **END** cannot be used to group expressions in a **LIST** defined by a **FOR** clause. Only left and right parentheses are permitted.

Permissible output parameters to a WRITE procedure are:

- An expression
- An array identifier
- A **LIST** identifier
- A **LIST** defined by **FOR** clauses

The above are also permissible parameters to the READ procedure with the restriction that only variables, not general expressions, can be parameters to READ. This applies also to lists being used as parameters to READ. The parameters to READ are really call-by-name parameters and occur on the left-hand side of assignment statements. Thus, they must be variables.

This section is concluded with a warning concerning the use of a **LIST**. Do not use an iterated variable of a **FOR** list as an iterated variable of a **FOR** statement with the activation of the **LIST** within its scope:

```
INTEGER I $
ARRAY ARGGGHA (1:10) $
LIST LISP(FOR I=10 STEP -1 UNTIL 1 DO ARGGGHA(I)) $
.
FOR I=(1,2,47)DO
BEGIN
      .
      .
      WRITE (LISP) $
      .
END $
```

This program is semantically incorrect. This type of error can be particularly difficult to isolate.

9.5. FORMATTED OUTPUT – THE **FORMAT** DECLARATION

It is often desirable to print or punch information in a specific manner rather than to accept the positioning automatically provided by the WRITE procedure.

The **FORMAT** declaration, which is included with the other declarations at the beginning of the block, provides a means of specifying how a printed page (or punched card) is to be formatted. A format is a set of specifications that can be interpreted by the I/O procedures to control the editing of information. The format takes this form:

FORMAT <identifier>(<format specifications>)

The following lines specify two formats, FEIN and FTWAIN:

```
FORMAT FEIN(X10,D7.2,X5,R17.8,A1.1),
       FTWAIN(B6,S10,I5,X2,T14.9,A3,E)$
```

A single format identifier may be included as a parameter in a call on WRITE, but its position in the call does not matter. For example, the two following calls in WRITE are equivalent.

```
REAL A,B $
.
.
WRITE(PRINTER,FEIN,A,B)$
WRITE(A,B,FEIN)$
```

A format specification consists of a series of editing and/or nonediting codes separated by commas. An editing code corresponds to a value to be printed and specifies how the value is to be edited. A nonediting code controls printing, spacing or insertion of blanks or constants into the print line. The action of WRITE, when a format is being used, is to pair each output expression with its corresponding editing code in the format. Nonediting codes are executed as they are encountered.

9.5.1. Nonediting Codes

In nonediting codes listed below, s and t are unsigned integers and w indicates the number of character positions. Two conventions are that As is the same as As.0 and A is the same as A0.0. For phases that require a t, both s and t must be less than 64.

FORMAT CODE		ACTION(PRINTER)	ACTION(PUNCH)
As.t	Activate (Each format statement must end with this)	Prints the line just edited. Skip s lines before printing and t lines after.	Punches the edited line into a card. s and t are ignored.
Es	Eject	Ejects the page to logical line s-1 if s-1 is below margin on current page. The next line to be printed, if it specifies A1.t, prints on line s. If s-1 is on the current page and is below the current line, Es skips to s-1 and the page is not ejected.	Ignored
Xw	Expunge	Skips the next w character positions (i.e., inserts w blanks in the line).	Same
'< any string not containing a '>'	Insert literal	Inserts the string enclosed in quotes into the line.	Same

Table 9–1. Output Nonediting Codes

9.5.2. Editing Codes

The editing codes are the same for both printing and punching. Each code acts on one value to be printed. The w specifies the field width (that is, the total number of character positions to be used in the editing, including signs, decimal points, and comma). If w is too small to do the proper editing, the whole field is filled with asterisks (*). Any editing done beyond the edge of the output medium (132 or 128 columns on the printer, 80 columns on the card punch) is lost. The d is interpreted differently for different phrases. In format codes that use no d (as Sw), w must be smaller than 4096. In codes that include d, w must be enough larger than d to include at least the decimal point, a sign, and the exponent with its sign if there is one.

FORMAT CODE		ACTION
Bw	Boolean	Prints **TRUE** or **FALSE** in the field, left justified. If the field is too short as much as possible is printed, e.g., B1 results in T or F.
Dw.d	Decimal	Prints a decimal number with d places after the decimal point, right justified in the field, and with a leading minus sign if negative.
Iw.d	Integer	Prints an integer number right justified in the field with a leading minus sign if negative. The integer is printed to the base d where d=0 and d=10 are equivalent. In the latter cases the .d can be omitted. Range of d: $2 \leq d \leq 10$.
Rw.d	Real	Prints d significant digits of a **REAL** or **REAL 2** variable in the form X.X. . . . . X, NN. A leading minus sign is printed if the number is negative. If the power of ten, NN, is negative, it is preceded by a minus sign. Note that w must always exceed d by 6 or more to allow for ± . , ± NN.
SW	String	Prints the first w characters of a string left justified in the field. If the string is shorter than w characters, the rest of the field is space filled.
Tw.d	Truncated	Prints a number with a decimal point right justified in the field. Only the first d significant digits are printed; a leading minus sign is printed if negative.

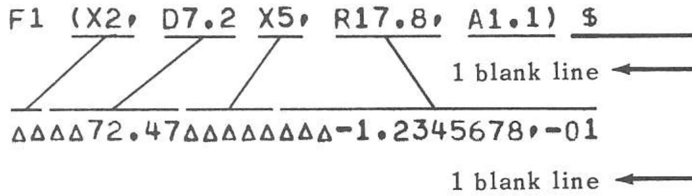
Table 9–2. Output Editing Codes

The type of the actual parameter is transferred to the type demanded by the editing code in any case for which there are transfer functions defined. A complex number is edited using two successive editing codes, the first for the real part and the second for the imaginary part.

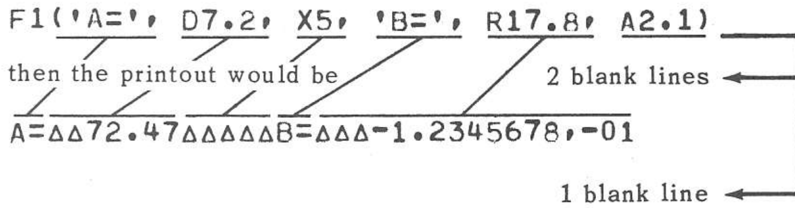
The following examples illustrate the use of various editing codes:

```
REAL A,B $
FORMAT F1(X2,D7.2,X5,R17.8,A1.1) $
•
•
A=72.474 $
B=-.12345678 $
WRITE(F1,A,B) $
```

The above coding would print A and B as follows:



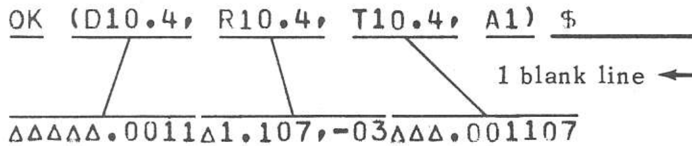
If F1 above were



To compare the three real codes D, R and T, suppose

```
REAL A $
FORMAT OK(D10.4,R10.4,T10.4,A1) $
A=0.001107 $
WRITE(A,A,A,OK) $
```

The printed line would then be as indicated below:



9.5.3. Repetition of Editing Codes

A single editing code (or a group of editing codes) can be used a number of times without actually repeating the code itself in the format statement. Three different methods can be used:

- Simple repetition – used when the number of repetitions required is known.
- Variable repetition – used when the number of repetitions depends on data.
- Indefinite repetition – used when the number of repetitions is indeterminate.

9.5.3.1. Simple Repetition

An editing code may be repeated by prefixing it with an unsigned nonzero integer constant which specifies how many times that code is to be repeated.

Example:

```
FORMAT F1(R16.8,R16.8,R16.8,A1)
```

is equivalent to

```
FORMAT F1(3R16.8,A1)
```

It is also possible to repeat a group of editing codes by enclosing them in parentheses and preceding this parenthetical group with an unsigned nonzero integer constant indicating the number of repetitions of the group. There is no limit to the depth of nesting, editing codes or groups of codes.

Given the declaration

```
BOOLEAN ARRAY BOOL(1:7,-1:4) $
```

the following format would permit printing the array elements with only one row per line

```
FORMAT FORM (7(6B7,A1)) $
```

9.5.3.2. Variable Repetition

A second type of repetition is the variable repeat. Instead of an integer, an arithmetic expression or Boolean expression enclosed in colons specifies the number of repeats. The Boolean values **TRUE** and **FALSE** are equivalent to one and zero, respectively.

Example:

```
INTEGER ARRAY A(1:N,1:M) $
FORMAT F(:N:(:M:(R16.8),A1))
```

would print the array one row per line (so M should be less than 9). The expression (N or M) is evaluated at the time the editing code is activated. If N or M have a value of zero or less, the group of format codes under control of that repeat expression is skipped. Note that in this type of repetition, the codes to be repeated must be enclosed in parentheses even if there is only one such code.

9.5.3.3. Indefinite Repetition

A final variant of editing code repetition is the indefinite or unlimited repeat. This is accomplished by enclosing a group of format codes in parentheses without preceding this parenthetical expression with an integer constant. The innermost parenthetical group that is not preceded by an integer constant is unlimited and will be used repeatedly until the output list is exhausted.

Given:

```
FORMAT ONANDON (E1,'VECTOR B',A1,(D7.2,A1)) $

REAL 2 ARRAY B(1:M) $
```

then

```
WRITE(B,ONANDON) $
```

will produce the following on a new page

```
VECTOR B

XXXX.XX
XXXX.XX
XXXX.XX
XXXX.XX
```

As an extension of this feature note that the parentheses surrounding an entire format are indefinite repeats. If there are more values to print upon reaching the end of the format string the whole format is repeated. Writing stops when the two following conditions are satisfied: there are no more values to be edited, and the right parenthesis of an indefinite repeat is encountered. Any editing code encountered when there is nothing more to be edited is treated as Xw. Nonediting codes are honored.

Example:

```
REAL A,B $
FORMAT Z(5D7.2,A2.3,'ABOVE IS DEBUG 1',A1) $
A = 17.00 $
B = -18.00 $
WRITE (A,A/2,B,B/2,Z) $
```

produces

```
ΔΔ17.00ΔΔΔ8.50Δ-18.00ΔΔ-9.00

ABOVE IS DEBUG 1
```

A common error to watch for is the omission of an activation code within an indefinite repeat (or within a format declaration in general):

```
FORMAT QTE(E,(8R16.8),A1.2)
```

would skip to the top of the next page but would not print anything. This error could be corrected as follows:

```
FORMAT QTE(E,(8R16.8,A1.2))
```

In this format, E is equivalent to E0 which has the effect of skipping to the bottom line of the current page. E0 followed by A1 will print on the top line of the next page.

Format codes to the right of an indefinite repeat or unlimited group can never be reached. The following output format:

```
FORMAT FORM9(X5,I5,D10.3,A1,(I5,4D10.3,A1),X10,I5,A1.2) $
```

used with a WRITE statement will cause the first two values to be printed according to I5 and D10.3. Since the inner parenthetical group is not preceded by an integer, printing will continue according to specifications within the parentheses until the output list is exhausted. The last three codes will never be reached.

A library procedure called MARGIN is useful for changing the margin settings on the printer. The standard form is 66 lines long with printing starting after line 4 and continuing until line 62. The call,

MARGIN(<length>, <top>, <bottom>)

where all parameters are integer expressions changes the form to the desired setting.

For instance, to set to standard:

```
MARGIN(66,4,62)
```

A fourth parameter to MARGIN (if present) may be a string, in which case the string is typed on the console when the printer symbiont comes to that point.

```
MARGIN(20,8,13,'OPERATOR - ACME CO. ENVELOPES')
```

The following example illustrates how versatile formats and lists can be. It prints an (N,M) array in a real format:

```
wJIT  FOR TEST
COMPILED BY UNIVAC 1107/8 ALGOL ON 23 JUN 67 AT 11:38:27      1 MAY 66 LEVEL 2
BLOCK 1      LEVEL 1
           1       INTEGER M, N, I1, I2, I, J $
           2       REAL R $
           3       REAL ARRAY A(1:50,1:50) $
           4       FORMAT ALL(X11,:ENTIER(MIN(I1+9,N))-I1+1: ('COL',I3,X6),A1,:M:('ROW',I3,
           5                 :ENTIER(MIN(I1+9,N))-I1+1:(R12.5),A1),A2) $
           6       LIST HAIR (FOR I1=(1,10,N) DO (FOR I2=(I1,1,MIN(I1+9,N)) DO I2,
           7                 FOR J=(1,1,M) DO (J, FOR I2=(I1,1,MIN(I1+9,N)) DO A(I2,J))))$
           8           R  = 0.0$
           9           M  = 24$
          10           N  = 18$
          11           FOR I = (1,1,N) DO
          12           FOR J = (1,1,M) DO
          13             BEGIN                                                            B1
          14               R = R + 1.0 $
          15               A(I,J) = R
          16             END $                                                            E1
          17           WRITE (ALL,HAIR) $
  END BLOCK 1

COMPILATION TIME FOR PHASE 1 WAS     2 SECOND(S)
COMPILATION TIME FOR PHASE 2 WAS     0 SECOND(S)
COMPILATION COMPLETE
```

The WRITE statement using format ALL and list HAIR has the effect of printing the elements of a real array by rows with ten elements per line. The output has this form:

	COL 1	COL 2	COL 3	COL 4	COL 5	COL 6	COL 7	COL 8	COL 9	COL 10
ROW 1	1.0000,+00	2.5000,+01	4.9000,+01	7.3000,+01	9.7000,+01	1.2100,+02	1.4500,+02	1.6900,+02	.9300,+02	2.1700,+02
ROW 2	2.0000,+00	2.6000,+01	5.0000,+01	7.4000,+01	9.8000,+01	1.2200,+02	1.4600,+02	1.7000,+02	.9400,+02	2.1800,+02
ROW 3	3.0000,+00	2.7000,+01	5.1000,+01	7.5000,+01	9.9000,+01	1.2300,+02	1.4700,+02	1.7100,+02	.9500,+02	2.1900,+02
ROW 4	4.0000,+00	2.8000,+01	5.2000,+01	7.6000,+01	1.0000,+02	1.2400,+02	1.4800,+02	1.7200,+02	.9600,+02	2.2000,+02
ROW 5	5.0000,+00	2.9000,+01	5.3000,+01	7.7000,+01	1.0100,+02	1.2500,+02	1.4900,+02	1.7300,+02	.9700,+02	2.2100,+02
ROW 6	6.0000,+00	3.0000,+01	5.4000,+01	7.8000,+01	1.0200,+02	1.2600,+02	1.5000,+02	1.7400,+02	.9800,+02	2.2200,+02
ROW 7	7.0000,+00	3.1000,+01	5.5000,+01	7.9000,+01	1.0300,+02	1.2700,+02	1.5100,+02	1.7500,+02	.9900,+02	2.2300,+02
ROW 8	8.0000,+00	3.2000,+01	5.6000,+01	8.0000,+01	1.0400,+02	1.2800,+02	1.5200,+02	1.7600,+02	2.0000,+02	2.2400,+02
ROW 9	9.0000,+00	3.3000,+01	5.7000,+01	8.1000,+01	1.0500,+02	1.2900,+02	1.5300,+02	1.7700,+02	2.0100,+02	2.2500,+02
ROW 10	1.0000,+01	3.4000,+01	5.8000,+01	8.2000,+01	1.0600,+02	1.3000,+02	1.5400,+02	1.7800,+02	2.0200,+02	2.2600,+02
ROW 11	1.1000,+01	3.5000,+01	5.9000,+01	8.3000,+01	1.0700,+02	1.3100,+02	1.5500,+02	1.7900,+02	2.0300,+02	2.2700,+02
ROW 12	1.2000,+01	3.6000,+01	6.0000,+01	8.4000,+01	1.0800,+02	1.3200,+02	1.5600,+02	1.8000,+02	2.0400,+02	2.2800,+02
ROW 13	1.3000,+01	3.7000,+01	6.1000,+01	8.5000,+01	1.0900,+02	1.3300,+02	1.5700,+02	1.8100,+02	2.0500,+02	2.2900,+02
ROW 14	1.4000,+01	3.8000,+01	6.2000,+01	8.6000,+01	1.1000,+02	1.3400,+02	1.5800,+02	1.8200,+02	2.0600,+02	2.3000,+02
ROW 15	1.5000,+01	3.9000,+01	6.3000,+01	8.7000,+01	1.1100,+02	1.3500,+02	1.5900,+02	1.8300,+02	2.0700,+02	2.3100,+02
ROW 16	1.6000,+01	4.0000,+01	6.4000,+01	8.8000,+01	1.1200,+02	1.3600,+02	1.6000,+02	1.8400,+02	2.0800,+02	2.3200,+02
ROW 17	1.7000,+01	4.1000,+01	6.5000,+01	8.9000,+01	1.1300,+02	1.3700,+02	1.6100,+02	1.8500,+02	2.0900,+02	2.3300,+02
ROW 18	1.8000,+01	4.2000,+01	6.6000,+01	9.0000,+01	1.1400,+02	1.3800,+02	1.6200,+02	1.8600,+02	2.1000,+02	2.3400,+02
ROW 19	1.9000,+01	4.3000,+01	6.7000,+01	9.1000,+01	1.1500,+02	1.3900,+02	1.6300,+02	1.8700,+02	2.1100,+02	2.3500,+02
ROW 20	2.0000,+01	4.4000,+01	6.8000,+01	9.2000,+01	1.1600,+02	1.4000,+02	1.6400,+02	1.8800,+02	2.1200,+02	2.3600,+02
ROW 21	2.1000,+01	4.5000,+01	6.9000,+01	9.3000,+01	1.1700,+02	1.4100,+02	1.6500,+02	1.8900,+02	2.1300,+02	2.3700,+02
ROW 22	2.2000,+01	4.6000,+01	7.0000,+01	9.4000,+01	1.1800,+02	1.4200,+02	1.6600,+02	1.9000,+02	2.1400,+02	2.3800,+02
ROW 23	2.3000,+01	4.7000,+01	7.1000,+01	9.5000,+01	1.1900,+02	1.4300,+02	1.6700,+02	1.9100,+02	2.1500,+02	2.3900,+02
ROW 24	2.4000,+01	4.8000,+01	7.2000,+01	9.6000,+01	1.2000,+02	1.4400,+02	1.6800,+02	1.9200,+02	2.1600,+02	2.4000,+02

	COL 11	COL 12	COL 13	COL 14	COL 15	COL 16	COL 17	COL 18
ROW 1	2.4100,+02	2.6500,+02	2.8900,+02	3.1300,+02	3.3700,+02	3.6100,+02	3.8500,+02	4.0900,+02
ROW 2	2.4200,+02	2.6600,+02	2.9000,+02	3.1400,+02	3.3800,+02	3.6200,+02	3.8600,+02	4.1000,+02
ROW 3	2.4300,+02	2.6700,+02	2.9100,+02	3.1500,+02	3.3900,+02	3.6300,+02	3.8700,+02	4.1100,+02
ROW 4	2.4400,+02	2.6800,+02	2.9200,+02	3.1600,+02	3.4000,+02	3.6400,+02	3.8800,+02	4.1200,+02
ROW 5	2.4500,+02	2.6900,+02	2.9300,+02	3.1700,+02	3.4100,+02	3.6500,+02	3.8900,+02	4.1300,+02
ROW 6	2.4600,+02	2.7000,+02	2.9400,+02	3.1800,+02	3.4200,+02	3.6600,+02	3.9000,+02	4.1400,+02
ROW 7	2.4700,+02	2.7100,+02	2.9500,+02	3.1900,+02	3.4300,+02	3.6700,+02	3.9100,+02	4.1500,+02
ROW 8	2.4800,+02	2.7200,+02	2.9600,+02	3.2000,+02	3.4400,+02	3.6800,+02	3.9200,+02	4.1600,+02
ROW 9	2.4900,+02	2.7300,+02	2.9700,+02	3.2100,+02	3.4500,+02	3.6900,+02	3.9300,+02	4.1700,+02
ROW 10	2.5000,+02	2.7400,+02	2.9800,+02	3.2200,+02	3.4600,+02	3.7000,+02	3.9400,+02	4.1800,+02
ROW 11	2.5100,+02	2.7500,+02	2.9900,+02	3.2300,+02	3.4700,+02	3.7100,+02	3.9500,+02	4.1900,+02
ROW 12	2.5200,+02	2.7600,+02	3.0000,+02	3.2400,+02	3.4800,+02	3.7200,+02	3.9600,+02	4.2000,+02
ROW 13	2.5300,+02	2.7700,+02	3.0100,+02	3.2500,+02	3.4900,+02	3.7300,+02	3.9700,+02	4.2100,+02
ROW 14	2.5400,+02	2.7800,+02	3.0200,+02	3.2600,+02	3.5000,+02	3.7400,+02	3.9800,+02	4.2200,+02
ROW 15	2.5500,+02	2.7900,+02	3.0300,+02	3.2700,+02	3.5100,+02	3.7500,+02	3.9900,+02	4.2300,+02
ROW 16	2.5600,+02	2.8000,+02	3.0400,+02	3.2800,+02	3.5200,+02	3.7600,+02	4.0000,+02	4.2400,+02
ROW 17	2.5700,+02	2.8100,+02	3.0500,+02	3.2900,+02	3.5300,+02	3.7700,+02	4.0100,+02	4.2500,+02
ROW 18	2.5800,+02	2.8200,+02	3.0600,+02	3.3000,+02	3.5400,+02	3.7800,+02	4.0200,+02	4.2600,+02
ROW 19	2.5900,+02	2.8300,+02	3.0700,+02	3.3100,+02	3.5500,+02	3.7900,+02	4.0300,+02	4.2700,+02
ROW 20	2.6000,+02	2.8400,+02	3.0800,+02	3.3200,+02	3.5600,+02	3.8000,+02	4.0400,+02	4.2800,+02
ROW 21	2.6100,+02	2.8500,+02	3.0900,+02	3.3300,+02	3.5700,+02	3.8100,+02	4.0500,+02	4.2900,+02
ROW 22	2.6200,+02	2.8600,+02	3.1000,+02	3.3400,+02	3.5800,+02	3.8200,+02	4.0600,+02	4.3000,+02
ROW 23	2.6300,+02	2.8700,+02	3.1100,+02	3.3500,+02	3.5900,+02	3.8300,+02	4.0700,+02	4.3100,+02
ROW 24	2.6400,+02	2.8800,+02	3.1200,+02	3.3600,+02	3.6000,+02	3.8400,+02	4.0800,+02	4.3200,+02

EXECUTION TIME 1.723 SECONDS - LIBRARY OF 1 MAY 66

9.6. FORMATTED INPUT

As with writing, a format may be included as a parameter to the READ procedure. A format tells how the card is laid out. The major advantage in using formats is that constants need no longer be delimited by blanks, and strings need not be enclosed in string quotes – the format specifies the 'fields' in which information lies. The discussion that follows is based on an example designed to illustrate most of the fundamentals of reading formatted cards.

The format declaration is the same as described in 9.5. The difference between a format being used as a parameter to WRITE and one being used as a parameter to READ is that the editing and nonediting codes are interpreted slightly differently. However, they are enough alike that in many cases the same format may be used with both procedures.

The following example illustrates reading data from cards according to a specified format. The information pertains to student records with each card having the following format:

Column	Contents
1–5	Student number
6–7	Student initials
8–21	Student name
22	Status
23–24	Curriculum
38–44	Course name
47	Credit hours
60	Letter grade

The problem is to read the above data in a form that will make the manipulations easy and permit printing all the information. It is this type of problem which gives rise to the necessity of specifying the card format. The steps necessary to achieve this result are:

Step	
(1) Read a card	
(2) Accept columns 1–5 as an integer	Student number
(3) Accept columns 6–7 as a string	Student initials
(4) Accept columns 8–21 as a string	Student name
(5) Accept column 22 as an integer	Status
(6) Accept columns 23–24 as an integer	Curriculum
(7) Skip the next 13 columns	
(8) Accept columns 38–44 as a string	Course name
(9) Skip 2 columns	
(10) Accept column 47 as an integer	Credit hours
(11) Skip 12 columns	
(12) Accept column 60 as a string	Grade

The **FORMAT** declaration can be used to take care of all the above functions. For example, the format could be

```
FORMAT VAYDREI(A,I5,S2,S14,I1,I2,X13,S7,X2,I1,X12,S1)$
```

Note there is one entry in the format for each numbered line above. Each of the items in the above format is referred to as a 'format code'. Of course, the initial A is analagous to the terminal As.t of the write and is required to activate the subsequent **READ** procedure.

A reasonable program segment for the above problem would be:

```
INTEGER STUDNO,STATI,CURCI,CREDS $
STRING INITIALS(2),NAME(14),COURSE(7),GRADE(1) $
FORMAT VAYDREI(A,I5,S2,S14,I1,I2,X13,S7,X2,I1,X12,S1) $
LIST FRIEDRICH(STUDNO,INITIALS,NAME,STATI,CURCI,
     COURSE,CREDS,GRADE) $
     READ(CARDS,FRIEDRICH,VAYDREI) $
```

Only two nonediting codes are permitted with an input format:

A activation code – required as the first code of the format to activate the subsequent READ procedure.

Xw – skip over w columns on the card

The editing codes (some of which are not in the example) have the following meanings, and w and d are restricted as before (see 9.5).

FORMAT CODE		ACTION
Bw	Boolean	Accepts Boolean information from the field – either **TRUE**, **FALSE**, or 1, 0.
Dw.d	Real	Accepts real information from the field. If the number is already real, (i.e., has a decimal point or exponent part) then that determines the decimal. Otherwise, a decimal point is inserted d places to the left of the right edge of the field.
Fw	Free	Accepts an unspecified number of values from the field. These numbers must be punched in free format mode; that is, values may be punched any where within w character positions.
Iw.d	Integer	Accepts information from the field as being integer to the base d. d=0 is equivalent to d=10.
Rw.d	Real	Same as Dw.d
Sw	String	Accepts the whole field as a string.
Tw.d	Real	Same as Dw.d

Table 9–3. Input Editing Codes

9.7. END-OF-FILE CONDITIONS ON CARD INPUT

Control cards are identified by a master space (a 7–8 punch) in column 1. Except for one special control card, the EOF (for end-of-file), these cards are never read by an ALGOL program. By using labels as parameters to the READ procedure, the programmer can detect a control card. In that case, if an attempt is made to read such a card, the READ procedure terminates reading and exits to the given label.

The EOF card has the letters EOF in columns 3–5. When an EOF card is encountered by the READ procedure, reading terminates. No new values are assigned to the remaining parameters in the <input list>. If a label is present as a parameter, the exit is made to that label. Otherwise, exit is made to the next ALGOL statement. The next time the READ procedure is called, it begins by reading the card after the EOF card. Thus, the purpose of EOF is to give the programmer a convenient method of separating sections of data of unknown length.

If a control card other than an EOF card is encountered, no more data cards can be read. The exiting of the READ procedure is controlled in the following way, depending on the number of labels in the READ call:

0 labels	The program is terminated.
1 label	Exit is made to that label, just as if an EOF card were read.
2 labels	Exit is made to the second label (the first is for EOF cards).

A designated expression may be used instead of a label.

Example:

```
BEGIN ARRAY A(1:20)$
LOCAL LABEL OK,FIN $
•
•
WEED:READ(CARDS,A,OK,FIN) $
•
•
OK: WRITE ('EOF - CARD READER') $ GO TO WEED $
FIN: WRITE('PROGRAM TERMINATED BY CONTROL CARD') $
END $
```

9.8. MAGNETIC TAPE AND DRUM I/O – THE POSITION AND REWIND PROCEDURES

The general call to the READ and WRITE procedures may be defined fairly closely as follows. For WRITE:

```
WRITE(<device>,<modifier list>,<label list>,
      <format>,<output list>)
```

and for READ:

```
READ (<device>,<label list>,<format>,
      <input list>)
```

So far only the printer, card punch, and card reader devices have been discussed. Two additional devices, tape and drum, will now be considered.

The call WRITE(TAPE(i),...) writes the information in the output list on logical tape unit i, where i is a nonnegative integer expression. The actual physical unit used is installation–dependent, but typically it might be:

Logical unit	Actual unit
0	Tape A
1	Tape B
2	Tape C
3	Tape D
4	Tape E
5	Tape F
6	Drum octal address 01000000–02777777
7	Drum octal address 03000000–046000000
8	Printer
9	Punch

When the actual unit is printer or punch, the action of WRITE is exactly the same as if PRINTER or PUNCH were used instead of TAPE(i). When the actual unit is tape or drum, the output is written in an internal format (unedited data in blocks) readable by either an ALGOL or a FORTRAN program. Do not confuse the tape-simulated drum file with the random access drum. The random access drum is called 'DRUM' and is discussed later. When the logical tape is a drum file, it is treated exactly as a tape file (i.e., may only be accessed in a serial manner). The REWIND procedure applies to the drum file just as for an actual tape unit. The REWIND procedure is called as follows:

```
REWIND(TAPE(I),TAPE(J),...)
```

with the option of

```
REWIND(TAPE(I),...,INTERLOCK)
```

which rewinds the indicated tapes with or without interlock as indicated.

The action of WRITE is basically the same as always; the information is output to some device. However, there are three additional concepts to consider.

The first is the interaction between READ and WRITE. The call

WRITE(TAPE(i),< output list >)

records a piece of information, called a logical record, on the device. This record may occupy one or more physical blocks on the tape. When READ is called by

READ(TAPE(i), . . .)

it will read exactly one logical record from TAPE(i). Therefore, the input list for the READ procedure must be compatible with the output list that produced the record, except that the input list may be shorter than the output list.

Example:

```
REAL ARRAY A,B(1:10,1:10) $
INTEGER N,I $
LIST L(FOR I=(1,1,N) DO A(3,I)) $
WRITE(TAPE(0),N,L) $
•
•
REWIND(TAPE(0)) $
READ(TAPE(0),N,L) $
```

The only incompatibility allowed is that the input list may be shorter than the output list. The values written are position-dependent and lose all identity with the output variables.

In the following statements, with the above declarations, the array A is ignored by the first READ, and the next call on READ begins at the next record:

```
WRITE(TAPE(0),N,A) $
WRITE(TAPE(0),B) $
•
REWIND(TAPE(0)) $
READ(TAPE(0),N) $
READ(TAPE(0),B) $
```

The second concept is that labels may be parameters to READ and WRITE when the device is TAPE. When writing, exit is made to the label if an attempt is made to write past the end-of-tape. The label parameters to READ are used as exit points under the following conditions:

NO. OF LABELS	EXIT TO	IN CASE OF
0	Normal exit (next statement)	EOF record or end-of-information
1	The label	As above
2	First label Second label	EOF record End-of-information (EOI)

The third concept is that of the modifier list. Modifiers may be parameters to WRITE. Their action is to output a marker in the information which may later be searched for by a call on the procedure POSITION. A modifier is any of EOI, EOF, EOF (<expression>), KEY, or KEY(<expression>) with the convention that EOF is equivalent to EOF(0) and likewise for KEY. In these modifiers <expression> may be any arithmetic or Boolean expression. Modifiers may appear in any order; that is, EOF may appear before EOI. The activate codes are executed as they are encountered. Thus, if there is more than one activate code in the format, EOF, EOI, etc. are output if they are encountered again before the next activation code.

If the modifier list contains KEY, then a KEY record is written before the usual record produced by WRITE. If the modifier list contains EOF then an EOF record is produced after the usual record. The expression is merely used to identify a particular KEY or EOF. If the expression is a string, only the first six characters of it identify the 'EOF record.'

Example:

```
INTEGER ARRAY I(1:101) $
INTEGER J $
•
WRITE(TAPE(0),KEY('I'),I)$
WRITE(TAPE(0),KEY('-I'),FOR J=(1,1,101) DO -I(J)) $
```

Also the forms

```
WRITE(TAPE(0),KEY(1)) $
WRITE(TAPE(0),EOF(1)) $
WRITE(TAPE(0),KEY(1),EOF(1),...) $
```

are permitted. The modifier EOI produces a tape end-of-information mark after the usual record. The procedure POSITION can be used to position a tape to a previously written KEY or EOF record, or to the end-of-information, or to a given number of ordinary records. The call is

POSITION(TAPE(i),<position parameter>,<label list>)

where the <position parameter> is

- EOF (<expression>)
- –EOF (<expression>)
- KEY (<expression>)
- –KEY (<expression>)
- <integer expression>
- EOI
- –EOI

The direction of positioning is indicated by the sign of the position parameter, positive for forward and negative for backward. EOI stands for 'end-of-information' and the positioning is done to the place where the next EOI mark is written. If the position parameter is an integer expression, the positioning is over that many logical records (KEY records not included – they are also ignored when encountered by a READ statement). Abnormal exits from the POSITION procedure occur as follows:

Position Parameter	Encounters	Exits to
EOF	End-of-info	First label
KEY	End-of-info	First label
<integer expression>	EOF record	First label
	End-of-info	Second label, or first if there is only one.

If the label list is empty, then the exits are made normally; that is, to the next ALGOL statement. But, using labels ensures that the positioning was done successfully when control comes back to the next ALGOL statement:

```
INTEGER ARRAY A(1:N) $
LOCAL LABEL OHNO $
START:
WRITE(TAPE(2),KEY(1),A) $
•
WRITE(TAPE(2),KEY(2),A) $
•
WRITE(TAPE(2),KEY(3),EOF,A) $
REWIND(TAPE(2)) $
POSITION(TAPE(2),KEY(3),OHNO) $
READ(TAPE(2),A) $
•
POSITION(TAPE(2),-KEY(2),OHNO) $
READ(TAPE(2),A) $
POSITION(TAPE(2),-KEY(1),OHNO) $
READ(TAPE(2),A) $
•
•
POSITION(TAPE(2),EOF,OHNO) $
GO START $
OHNO = WRITE('ABNORMAL EXIT FROM POSITION') $
```

The procedure POSITION always positions over the record it is looking for in the direction indicated. Thus a POSITION backward to an EOF record followed immediately by a READ will encounter that EOF record immediately.

The position backward to an EOF or KEY record cannot be used when TAPE(i) is a drum-simulated tape file.

The device 'DRUM' allows the reading and writing of information in essentially a random access manner. For this purpose a portion of the drum is set aside and the parameter to DRUM indicates the relative word address of this random access file.

Example:

```
WRITE(DRUM(0),A)
```

writes A at the very first part of the random file. To effectively use DRUM, the programmer must know how many words are required for each expression or array being output. The answer is that single precision quantities require one word (**INTEGER**, **REAL**, **BOOLEAN**) and double length quantities require two (**REAL 2** and **COMPLEX**). Strings require one word for each six characters of their length plus one additional word. Information may be read from DRUM address i by

```
READ(DRUM(I),< input list >)
```

A label as a parameter to a READ or WRITE with DRUM is used as an exit if an attempt is made to read or write past the end of the file.