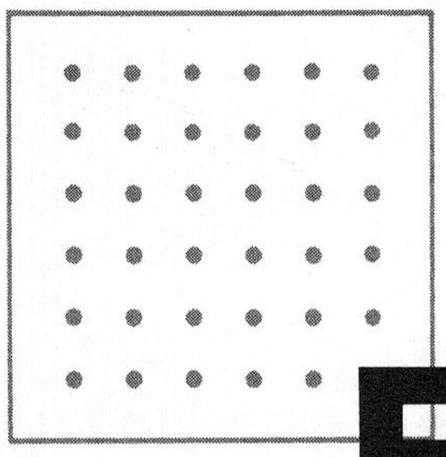
UNIVAC® 1107



PROCS

GENERAL DISCUSSION

UP-2593.33

I. GENERAL

A. Introduction

1. Scope

This paper will describe how computer language code is produced using the UNIVAC 1107 SLEUTH II Assembly System. Of necessity, detailed explanation and many definitive examples of the concept "procedure" will be given. Complex procedures will be developed using the full range of the arithmetic operators available in SLEUTH II. A working knowledge of SLEUTH II and the UNIVAC 1107 is assumed.

2. Method

SLEUTH II itself makes relatively few assumptions about the form of its input and output. Instead, it accepts sets of definitions and descriptions which dictate its operation. The basic descriptive tool is the procedure.

The power of SLEUTH II is usually limited by the imagination of the user. Therefore, the reader is invited to extrapolate from solutions to various problems shown, toward more general and/or less complex solutions.

In using SLEUTH II, it may be generally assumed that unless one of the basic rules directly or inferentially forbids it, most any combination of directives, values, labels, operators, and the like, however complex, will produce a predictable (perhaps not the desired) result.

B. Control of SLEUTH II

1. Basic SLEUTH II

Though it may appear from published descriptions of SLEUTH II for the UNIVAC 1107 that instruction mnemonics, data forms, and the like are a part of the assembler, this is not so. As will be shown, the UNIVAC 1107 is defined to SLEUTH II as an object computer. The symbolic language to be submitted to SLEUTH II is described, so that the assembler may evaluate the symbolic code and produce output code according to an independent set of definitions. Only the SLEUTH II directives are known to the assembler at the time of assembly of any symbolic code.

These directives, as described in the SLEUTH II manual, are directly interpreted by the SLEUTH II assembler. They are the most basic part of the assembler and the most powerful means available to the programmer to modify the operation of the assembler. The directives are:

PROC	GO	NAME
EQU	END	LIST
LIT	RES	UNLIST
FORM	INFO	
DO	FUNC	

2. SLEUTH II for the 1107

When SLEUTH II is operating for the production of code for the UNIVAC 1107, a set of 1107 definitions are made available to SLEUTH II through the mechanism of EXEC II:

a. "Standard procedures" -- A set of procedures describes the form and content of 1107 instructions. The mnemonics are defined, as well as certain addresses, constants and linkages used in connection with the 1107 EXEC II system. These procedure definitions are actually a part of the system library, which resides on magnetic drum. Instruction definitions are brought into core as needed by SLEUTH II during assembly.

b. The "Blank" Procedure -- Generation of constants from decimal, octal, or alphanumeric representation is done through use of the so-called Blank procedure. This procedure is not among the standard procedures but is actually a part of SLEUTH II itself, and is assembled by the assembler during ititialization.

c. The Instruction Form -- A symbolic FORM directive specifies the size of the sub-fields in an 1107 instruction word. It immediately follows the Blank Procedure. This form will be referred to in several examples in this paper:

```
I$          FORM        6,4,4,4,2,16
```

d. Object computer characteristics -- The 1107 Field Data Code character representation, the 1107 word length, and the negative representation for the 1107 are built into SLEUTH II.

II. CODE GENERATION

A. Review of Procedures

In any assembly language the basic task is to provide the programmer with a means of representing a binary machine instruction or binary data word in some more convenient set of terms such as symbols, names, octal numbers, decimal numbers, and the like. The purpose of an assembler is to analyze these symbols, to evaluate them, to place the derived values in the proper format for the object computer, and to output these values.

The procedure offers a means of instructing the assembler precisely how to construct a unit -- one or more words -- of output from the input symbols. For emphasis and review, a brief discussion of the construction and referencing of procedures follows:

1. Reference

A procedure reference line follows the syntactical rules stated in the SLEUTH II manual. The line includes a label (optional), an operation (required), and operands (optional).

A label, if present, will be equated to the value of the current location counter for the first line of code generated by the procedure. (A means of altering this equation will be shown below in II D 4).

The operation determines which procedure is to be performed. It must be the label of a PROC directive, or of a NAME directive within the procedure. The operation field may also contain one or more additional parameter expressions which may be referenced in the procedure. The operation is terminated by a blank.

The operands, if required, are written as one or more lists of expressions. Expressions within a list are separated by commas, and lists are terminated by a blank not following a comma.

2. Construction

A procedure declaration must begin with a PROC directive and end with an END directive. Labels are required on PROC and NAME directives. Suffixing any one or all of these PROC or NAME labels with an asterisk defines entrances to the procedure. All directives may be used within the procedure.

Procedures may be nested. Reference to nested or non-nested procedures may be made within procedures, as long as the referenced procedure has been encountered prior to the reference, or is available from a library.*

The main purpose of a procedure is to describe the method for generating one or more lines of output code and its form. Input to a procedure are parameters which, when evaluated within the procedure, will either comprise the output code and/or conditionally affect its production.

Since by its nature a procedure may be utilized many times in one program or in different programs, it is strongly recommended that detailed comments be used in the symbolic procedure code.

The means of reference to procedure parameters, which are supplied on the procedure reference line, are called *paraforms* and are described below, with examples following in paragraph 4.

3. Paraforms

Note: L represents the label of a PROC directive.

L	Number of parameter lists. If entry to the procedure is by a NAME line, the value of L is increased by 1, since the NAME parameter(s) are counted as a list (see L(a,b) below). Examples 1, 10, 20.
L(a)	Number of parameters in the ath list. Examples 2, 15, 21.
L(a,b)	Value of bth parameter in the ath list. Examples 3, 4, 5, 11, 13, 14, 23. List 0 (zero) is a special case: L(0,0) is the value of an expression in the operation field of the NAME line used for entrance into the procedure. Examples 7, 16. L(0,1).....L(0,n) are the second through nth expressions in the operation field and are available *only* if entrance to the procedure is by a NAME line. Examples 8, 17, 18, 22.
L(a,*b)	If an asterisk precedes the bth parameter of the ath list, value is 1. Otherwise, value is 0. Examples 6, 9, 12, 19.

*See EXEC II Programmers Guide (UP-3671) pp. 5-1, 5-2, 7-1, 7-2.

4. Examples

Below is a dummy procedure with three entry points, P, A, and B. Following the procedure is a table showing three references (procedure reference lines) to the procedure, along with values which would be obtained by different paraform examples, were they to be used in symbolic lines within the dummy procedure. Note that if a parameter is not supplied, its value is zero.

```
P*      PROC      2
A*      NAME      *010
B*      NAME      012
        etc.  (usage of any of the paraforms listed below)
        END
```

For further clarification, here are the three reference line examples as they would appear when coded:

Label	Operation	Operands
	A	7,G
	B,*14	8,*13 6
	P,3	I, J, K

Reference Line	Example Number	Paraform	Value
A 7,G	1	P	1
	2	P(1)	2
	3	P(1,1)	7
	4	P(1,2)	G
	5	P(1,3)	0
	6	P(1,*2)	0 (off)
	7	P(0,0)	010
	8	P(0,1)	0
	9	P(0,*0)	1 (on)
B,*14 8,*13 6	10	P	3
	11	P(1,2)	13
	12	P(1,*2)	1
	13	P(2,1)	6
	14	P(2,2)	0
	15	P(0)	2
	16	P(0,0)	012
	17	P(0,1)	14
	18	P(0,2)	0
	19	P(0,*1)	1
P,3 I,J,K	20	P	1
	21	P(0)	0
	22	P(0,1)	0
	23	P(1,3)	K

5. Typical 1107 Standard Procedure

Most 1107 instructions which involve A registers are developed with procedures of this general form:

```
P       PROC    1,1
SA*     NAME    01
LA*     NAME    010
SNA*    NAME    02
        etc.
        I$      P(0,0),P(0,1)+P(1,4),P(1,1)-12,P(1,3),;
                2*P(1,*3)+P(1,*2),P(1,2)
        END
```

The six fields in the I$ form reference line represent the f, j, a, x, h-i, and m fields respectively. It can be seen immediately that one procedure may define a group of instructions of the same general type. In fact the 1107 standard procedures include 19 procedures which define all 132 SLEUTH II mnemonics.

Field by field, the form reference is explained:

P(0,0) (Function code – 6 bits): is determined by the point of entrance into the procedure, illustrating the use of the NAME directive operand as a procedure parameter.

P(0,1)+P(1,4) (J-factor – 4 bits): may be written either as the second expression in the operation field (P(0,1)) or as the fourth expression in the operand field (P(1,4)). The effect would be either "P(0,1)+0" or "0+P(1,4)", unless such a line as

```
LA,1        12,VALUE,    ,3
```

were written, in which case j would be 3+1, or 4. The reader is cautioned against such misleading coding, for it would confuse the uninitiated!

P(1,1)–12 (A-designator – 4 bits): as will be discussed later, the A-register parameter is assumed to be an absolute film address. Twelve is subtracted from the value of the parameter to make it an address relative to the beginning of the A registers in film.

P(1,3) (B-designator – 4 bits): since its relative address is the same as its absolute film address, no alteration is necessary. If the parameter is not furnished, the value is zero, as for any omitted parameter.

2*P(1,*3)+P(1,*2) (H and I designators – 2 bits): the presence or absence of the asterisk flags for the P(1,2) and P(1,3) parameters determines the content of this field. P(1,*3), the one-bit H designator, is multiplied by 2 to shift it into the left half of this two-bit field. Then the I designator P(1,*2) is added. The possible values for this field:

Condition	Binary Value	Octal Value
neither	00	0
*m	01	1
*x	10	2
both	11	3

P(1,2) (M address – 16 bits): simply the value of the address operand.

B. PROCEDURE ANALYSIS

1. Reference Operand Processing

For a more general understanding of a procedure reference, consider paraforms to be subscripted labels*. To explain, once the assembler detects a procedure reference in the operation field, it makes a group of entries in its item directory (a table of symbols and their associated values) according to the list(s) of parameters furnished on the procedure reference line. This group is given the label of the referenced procedure, and items within the group have values as indicated in the table of paraforms above. The label subscripts denote list, and parameter within list respectively. If an asterisk precedes a parameter, its corresponding item will be flagged accordingly with a binary operator.

*to be discussed in detail in Section II E.

After the item directory is embellished in this manner, the symbolic coding within the referenced procedure is processed as a separate sub-assembly. When reference to a parameter (e.g. "P(1,2)") is encountered during this sub-assembly, the value of the item is retrieved from the item directory. At the end of the sub-assembly, the item directory is cleared of the group of items created by the procedure reference line.

If a reference to another procedure is encountered during such a sub-assembly, this process becomes recursive, up to 64 levels. This limitation (which, incidentally, has never been experienced) holds true for functions also.

Here is an example of a recursive procedure reference:

```
P*      PROC        1,1
        Q           P(1,1), P(1,2)
        END

Q*      PROC        1,1
F       FORM        9,9,9,9
        F           Q(1,),Q(1,*1),Q(1,2),Q(1,*2)
        END
        P           *A,B    (procedure reference line)
```

When the assembler encounters the P reference line, the following items, with their associated values, are added to the the item directory:

ITEM	VALUE	REMARKS
P	1	nr. lists submitted
P(1)	2	nr. parameters in list
P(1,1)	A	asterisk flag on
P(1,2)	B	asterisk flag off

Then, the coding within the procedure is assembled. The first (and only) line is a reference to procedure Q, so recursion begins. Again items are added to the directory:

ITEM	VALUE	REMARKS
Q	1	
Q(1)	2	
Q(1,1)	A	asterisk flag on
Q(1,2)	B	asterisk flag off

For the evaluation of Q(1,1), the assembler looks up the value of P(1,1) in the item directory. Since the asterisk flag is "on" in P(1,1), it is also set in the value item Q(1,1), which gets the value of A.

2. Propagation of Values

It is important to note that at any given level of recursion, values of items evaluated at a "lower" or "outer" level are available. Equally important is the deletion of item directory entries resulting from a procedure reference line, once the coding in the procedure has been completely processed in the sub-assembly. Thus at a given level of procedure processing, items evaluated at a "higher" or "inner" level are *not* available.

If it is desired to cause a label and its associated value to survive after the procedure sub-assembly, the label should be immediately followed by one or more asterisks – one for each procedure level through which the label definition is to be carried:

```
                                    etc.
┌------A*          PROC
|   ┌---B*          PROC
|   |  ┌--C*          PROC
|   |  |  W***          + 1          · CARRIES 3 LEVELS
|   |  |  X**           + 2          · CARRIES 2 LEVELS
|   |  |  Y*            + 3          · CARRIES 1 LEVEL
|   |  |  Z             + 4          · KNOWN HERE ONLY
|   |  └----------END
|   |                   C ·
|   |                   + Z          · UNKNOWN
|   |                   + W,X,Y      · KNOWN
|   └----------------END
|                       B ·
|                       + Y, Z       ·· UNKNOWN
|                       + W, X       · KNOWN
└--------------------END
                        A ·
                        + X, Y, Z    · UNKNOWN
                        + W          · KNOWN
                        etc.
```

This diagram shows three nested procedures, and illustrates the use of asterisks to "carry through" the value of a label.

If the number of asterisks exceeds the number of levels, the additional asterisks will be ignored by the assembler.

3. PROC Directive Operands

None, one or two expressions may appear in the operand field of the PROC directive. The first, if coded, is the maximum number of lists of expressions which may be submitted to the procedure on a procedure reference line. Knowing this, the assembler can terminate the scan of the reference operand field once the maximum number of lists has been encountered. Comments following the last list are ignored.

If the procedure permits a variable number of lists (e.g. several of the 1107 Library I/O linkage procedures) *and* less than the maximum number of lists is submitted, *and* comments follow the last list submitted, then a line terminator -- the period-blank convention -- must precede the comment in the reference line.

The second PROC directive operand, if coded, is the exact number of words to be produced by the procedure. If a procedure is coded so that it always produces a fixed number of lines of object code, it need (in most cases) be assembled only once -- in the second assembly pass. This is a desirable condition, shortening assembly time.

If, however, a variable number of lines may be produced, pass 1 sub-assemblies are required to establish exactly how many object lines are to be produced, so that following statement labels will be given the same location counter value in both passes. In different words, pass 1 procedure assemblies are necessary if the number of lines to be generated is not stated.*

Examine the following invalid code:

```
A         EQU          1
THERE     etc.
P*        PROC         1,1
          DO  P(1,1)>1  ,  +077
          END
          P            A
HERE      etc.
```

The second operand on the PROC directive line specifies that 1 object word is to be produced by procedure P, so a first pass sub-assembly is assumed unnecessary when procedure P is referenced. The location counter is incremented by the value of the second PROC operand, and first pass assembly continues. The label HERE is equated to the location counter, which at that point is THERE + 2.

But when P procedure is assembled in the second pass, no object code is produced, since P(1,1)>1 is not true. Thus the assembler attempts to assign HERE the value THERE+1, which results in a D flag!

Another situation which demands sub-assembly of a procedure in both passes is forward references in the procedure--that is, reference to a symbol before it is defined:

```
P*        PROC         1
          LMJ          11,P(1,1)
          END
           P           SUBR
          etc.
SUBR      etc.
```

The above is valid, but would not be if a second operand were coded on the PROC line, suppressing the first pass sub-assembly.

C. FLAGS AND OPERATORS

The availability of a wide variety of arithmetic and logical operators in SLEUTH II facilitates the construction of elaborate procedures for developing code.

Misuse of these operators may lead to incorrect results and extraneous error flags. Obversely, it may be desired to generate certain error flags in situations where, while the object code is valid according to SLEUTH II rules, it is *not* valid for the 1107.

1. R-Relocation

Appearing in an 1107 SLEUTH II assembly, an R flag means that a relocatable item (usually a label) has been so used in an expression as to cause loss of its relocation properties. *Table 1* shows the results of all combinations of relocatable and non-relocatable items.

*There are other reasons for which a pass 1 procedure sub-assembly may be required. These are difficult to describe, and are best learned through trial and error.

Table 1. SLEUTH RULES FOR RELOCATION OF BINARY ITEMS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT	NOTE
1	Any	<,=,>	Any	Not relocatable	
2	Any	++,−−	Any	Not relocatable	c
3	Any	**	Any	Not relocatable	c
4	Not relocatable	+,−	Not relocatable	Not relocatable	
	Relocatable	+,−	Not relocatable	Relocatable	
	Not relocatable	+,−	Relocatable	Relocatable	
	Relocatable	+,−	Relocatable	Relocatable	b
5	Any	*,/,//	Any	Not relocatable	c,d
6	Any	*+,*−,*/	Binary*	Not relocatable	c,e

a. Floating point items are never relocatable.

b. The difference of two relocatable quantities under the *same* location counter is not relocatable.

c. Except as noted in d, the relocation error flag (R) will be set for these operations.

d. Multiplication of a relocatable quantity by an absolute 1, or absolute 1 by a relocatable quantity is relocatable. Multiplication by absolute 0 is absolute 0. In either case no error flag is set.

*e. A non-binary, that is, floating point value will result in an expression error flag (E).

2. E-Expression

Expression error flags may be produced in a variety of ways, such as the inclusion of a decimal digit in an octal number (e.g. 080), and binary or decimal exponentiation with a real exponent (e.g. 3.14*/1.2).

If it is desired to generate an E flag to bring to the user's attention some invalid condition in a line of symbolic code,

```
DO condition ,label  EQU  bad-expression
```

where "condition" is an expression with the value 1 if the E flag is to be generated, 0 (zero) if not. Since this line would be within a procedure, it would not have any deleterious effect on the assembly. The label, however, should be unique within the procedure. Such a line should precede the line which will generate the word or value which is to be flagged. For example,

```
P*      PROC      1,1
F       FORM      6,12,18
        DO        P(1,3)>14  ,A(0) EQU 080
        F         P(1,1),  P(1,2),  P(1,3)
        END
        P         12,14,16
```

The procedure P generates one word from the list of three parameters furnished on the procedure reference line. Since the third parameter, P(1,3) or 16, is greater than 14, an E flag would appear to the left of the output word on the printed listing.

3. T-Truncation

The T flag indicates that a value is too large for its destined field. Consider the following example:

```
F                FORM      18,18
A       EQU      +(F       0,-3)      (1)
G                FORM      32,4
                 G         0,A        (2)
```

The form reference in line (1) is legitimate, but (2) would produce a T flag, since the value of A in this case is octal 000000777774 (a value with 18 significant bits), and the second field of form "G" is defined as 4 bits in length.

The T flag will also appear on a line containing a location counter reference greater than 31(37_8, or 5 bits).

4. L-Level

This flag indicates that some capacity of the assembler, such as a table count, has been exceeded. The limits listed below are generous; but if one is exceeded, simplification of coding will be necessary.

a. Nested procedure or function references may not be more than 63 deep.

b. Parentheses nests, including nested literals, may not be more than 8 deep.

c. Nested DOs may not be more than 15 deep.

5. D-Duplicate

Labels, disregarding possible subscripts, must be unique in a given assembly. Redefinition of a label will produce a D flag on each line in which the label appears, unless the label is subscripted. The obvious mistake

```
A       EQU     1
        :
A       EQU     2
```

is easily discovered. Much more insidious is the redefinition in assembly pass 2 of a label previously assigned a different value in pass 1. This usually results from an illegal manipulation of a location counter (see II B 3 for an example).

6. I-Instruction, and U-Undefined

If the first symbol in the operation field of a symbolic line is neither the name of a directive or an available procedure, an I flag is produced. Remember that a procedure is available only if it is in the procedure library, or has previously been encountered in the source code.

If an operand symbol is not defined in the source code, each line containing the symbol will be marked as containing an undefined symbol -- with a U flag. In some cases this may denote an external reference to a value defined in some other independently processed code. But there is the chance that a U flag might simply denote an omission by the programmer.

7. Other Notes on Operators

Demonstration of use of the 14 arithmetic and logical operators is best done through examples. An examination of the 1107 Standard Procedures will be most helpful. One of the more complicated 1107 procedures is explained in detail in II F 2.

The user should be cautioned to observe the rules covering mixed-mode expressions. These are tabulated in Table 2 below.

Table 2. RULES FOR MODES OR RESULTS

LEVEL	1st ITEM	OP	2nd ITEM	RESULT
1	Any	<, =, >	Any	Binary (1 or 0)
2	Any	++, −−	Any	Binary
3	Any	**	Any	Binary
4	Binary	+, −	Binary	Binary
	Floating	+, −	Binary	Floating
	Binary	+, −	Floating	Floating
	Floating	+, −	Floating	Floating
5	Binary	*, /, //	Binary	Binary
	Floating	*, /, //	Binary	Floating
	Binary	*, /, //	Floating	Floating
	Floating	*, /, //	Floating	Floating
6	Any	*+, *−	Binary*	Floating
	Any	*/	Binary*	Binary

*A non-binary, that is, floating point value will result in an expression error flag (E).

D. ADDITIONAL NOTES ON DIRECTIVES

1. Or-ing of Forms

A field in a FORM reference line can be a line item. If the form of the line item is identical to the form referenced, and is *not* a literal, the corresponding fields from both form references will be "or-ed". This is a SLEUTH II feature which can be used to great advantage.

For example, consider two identical forms:

```
A       FORM    9,9,9,9
B       FORM    9,9,9,9
```

Now, create a line item with one of these forms and equate its value to a label:

```
C       EQU     +(A 1,3,2,4)
```

The + preceding the line item suppresses creation of a literal. Next, using C as one of the values, refer to form B

```
        B       C,6,7,010
```

The results are shown in octal:

```
C = 001  003  002  004)  inclusive or
B = 000  006  007  010)
result = 001  007  007  014
```

A more practical example: suppose repeated references were to be made to a specific character of a word in an 1107 SLEUTH II program. It may be desired to have one symbol representing both the j and m portions in an instruction word. Thus, we could have

```
A       EQU     +(I$  0,j,0,0,0,m)
        SA      A
```

"SA", the mnemonic for the Store A-register instruction, refers to an instruction-defining procedure which also refers to the I$ form. The value of A is the line item (note the leading plus) in the operand field of the EQU directive. For j=014 and m=010164, the effect is

```
01  00  00  00  0  000000          from SA proc
00  14  00  00  0  010164          line item
-------------------------------------------
01  14  00  00  0  010164          output word
```

Observe that the same result can be gotten by

```
SA,A          (no operand)
```

since in SLEUTH II for the 1107, j-factors may be written in the operation field. Going further,

```
SA,A          *0100000
```

would result in

```
01  00  00  00  1  100000)
                          ) inclusive or
00  14  00  00  0  010164)
-------------------------
01  14  00  00  1  110164
```

A generalized tool to achieve this effect, called the EQUF procedure, is among the standard procedures for the 1107. It will build a non-literal line item which includes the m, x, j, and h-i fields:

```
P          PROC       1

EQUF*      NAME       0

*          EQU        +(I$ 0,P(1,3),0,P(1,2),2*P(1,*2)+P(1,*1),;
                      P(1,1)
```

See 4 below for an explanation of * in the label field. A legitimate reference to this procedure would be

```
FIELD      EQUF       *0100,*8,14
```

The three operand expressions represent m, x, and j respectively; the asterisks indicate i and h bits respectively. The field definition thus created could be referred to in this manner:

```
LA         16, FIELD

           giving

10  00  20  00  0  000000)
                          ) inclusive or
00  16  00  10  3  000100)
-------------------------
10  16  20  10  3  000100
```

Reviewing, the two rules basic to the assembler demonstrated here, are:

a) A line item is an entire symbolic line, less label, enclosed in parentheses. If a line item appears *alone* in an expression, a literal with the value of the word generated by the symbolic line is created. The value of the expression is the address of the literal. If a line item is *not* by itself, e.g. "A+(1)" or simply "+(1)", the value of the line item is the value of the word generated by the enclosed line. Exception: "*line-item" *will* cause creation of a literal, since a leading asterisk has a different meaning to the assembler.

b) A field in a form reference may be a non-literal line item, if the reference and line item forms are identical. As a result, the corresponding fields will be "or-ed".

2. Setting Location Counters

SLEUTH II may still be used to prepare absolute programs in the ICS format described in the EXEC II manual, Section 3.5. The P and Q options must be used on the ASM control card to produce this type of output.

Since a simple loader is used to load the output Multiple Word Octal Load cards, the Allocator is not involved. Addresses assigned to each word are equivalent to the location counter value for each line output by SLEUTH II.

If it were desired to have the object program loaded at core address 010000 and following, the line

```
$(1)          RES          010000-$
```

should precede source code instructions. The "-$" is not absolutely necessary; it guarantees that any previous value of location counter 1 will be negated.

If literals are to be generated, their storage should also be fixed. If a LIT directive is not used, literals will automatically be generated under control of location counter zero, which should be similarly set to any desired address.

Thus the line

```
$(0)        RES        0100000-$
```

would cause generated literals to be assigned core bank 2 addresses. This line should begin the assembly.

In this manner, the first versions of all the 1107 EXEC II system components were coded, assembled, and loaded.

3. Labeled LIT Directives

More than one labeled LIT directive may appear under the same location counter. If this is done, each LIT label declared under a given counter is merely an alias for others under that counter.

The rule stated correctly: each LIT directive must have a unique label in a given assembly. Since for this context "spaces" is a label, only one unlabeled LIT directive may appear in a given assembly.

4. Label of a Procedure Reference

The label on a procedure reference line is defined as if it appeared on the first line within the procedure which contained, instead of a label, an asterisk in the label field. In the absence of such a line, the reference label is given the value of the current location counter when the procedure was entered. Example:

```
CALL        PROC        1,3
              +         CALL(1,2)
              +         CALL(1,3)
*           LMJ         11,CALL(1,1)
            END
            etc.
LOCA        CALL        ARCSIN,ARG1,ARG2
            etc.
            J           LOCA
```

If $ is the value of the location counter at the point of reference to CALL procedure, the value of LOCA is $+2. This is a situation in which it is desirable to have the waiting label on the procedure reference line assigned, not to the value of the location counter for the first line generated by the procedure, but as if it had appeared on some subsequent line: the one marked with an asterisk.

5. Control Counter on a Procedure

A procedure may be made to generate code under a particular location counter by specifying the counter on the PROC directive line in this manner:

```
$(3),P      PROC        1,3
```

The assembler will consider this a "local" control counter declaration, so that lines following the procedure reference will be under the control counter used prior to the procedure reference. A full example:

```
$(2),Q*     PROC        1
            etc.
            END
(1) $(1)    etc.
(2)         etc.
(3)         Q
(4)         etc.
```

Code produced at lines 1, 2 and 4 will be under control counter 1. Code produced at line 3 by procedure Q will be under control counter 2.

6. Considerations of NAME and GO

The GO directive, according to the SLEUTH II manual, "provides the means of transferring control within a PROC or FUNC to a specified NAME directive within that PROC or FUNC." While generally true, more should be said.

The operand of a GO directive, sometimes called "the object of a GO", may be one of the following:

a. A label of a NAME directive which is in the same procedure as the GO. If the GO is in a "forward" or "downward" direction, the object label must be asterisked.

b. An external (asterisked) label of a NAME directive of *any* procedure.

c. The label of a PROC or FUNC directive, if it is asterisked.

The SLEUTH II manual description of the NAME directive implies another general rule concerning GO: the object of a GO must have been encountered by the assembler prior to the reference. Examples:

```
P*          PROC        etc.
            etc.
            END
            etc.
Q*          PROC        etc.
            etc.
            GO          P
            etc.
P           NAME        etc.
            etc.
            END
```

If Q procedure were referenced, its GO line would reference P procedure, rather than the P NAME line in Q procedure.

A second example:

```
            P           PROC        etc.
            GEN         NAME
                        etc.
(1)                     GO          ENDPRC
                        etc.
            A*          NAME        etc.
                        etc.
(2)                     GO          GEN
                        etc.
            ENDPRC*     NAME        etc.
                        etc.
                        END
```

When P procedure is encountered by the assembler, it is stored as previously described. Defined entrances, such as the ENDPRC and A NAME lines, would be noted. Thus when the procedure is referenced by the name "A", the GO at line 1 would work. The GO at line 2 would not, since the existence of GEN has not been noted by the assembler. Which leads to another rule: if a procedure is entered by referring to an asterisked NAME line, anything previous to that NAME line is considered undefined by the entry.

Another comment on the use of the NAME directive: one might notice that it is sometimes used for apparently no reason, as in the EQUF procedure in II D 1. If EQUF had been the label of the PROC directive, each paraform would necessarily include that four-character label, significantly lengthening the symbolic line. Thus this use of NAME is only a sort of shorthand.

7. LIST AND UNLIST

These directives, though not concerned with the generation of object code, are recent innovations. Effectively they turn the assembler's printer output routine "on" or "off", overriding the presence or absence of the N option on the ASM control card. The L and I options, since they only control the type of listing, are unaffected.

These directives allow the programmer to suppress portions of an assembly listing, perhaps to save printer time, or for security reasons.

E. SUBSCRIPTED LABELS

1. Advantages

A subscripted label is a special form of label which allows several specialized usages. The major distinction is that the the value associated with a subscripted label may be changed freely, whereas changing the value of labels without subscripts results in D (duplicate) flags. Obversely, if an unsubscripted label to which a value has not been assigned is referenced, a U (undefined) flag results. For an undefined subscripted label, the value zero results.

The subscript of a label may be any legitimate SLEUTH II item or expression, including a subscripted label. Here are 6 examples of legitimate subscripted labels:

L(3)	L(I(1),J(1),K(1))
L(4,1)	L(2,P(1,R(1,1)**017))
L(1,M(1))	L(4,3,SIZE//2)

Using subscripts, a set of values may be assigned to one label. Each of these values may be referenced as operands by appropriately subscripting the label. Since the subscript may be an expression, it is possible to logically or arithmetically determine which particular value is desired.

2. Usage with Procedures

Remembering the section on Procedure Analysis, it can be seen that the result of procedure reference operand processing is a series of values identified by subscripted labels. For those paraforms with less than two subscripts (e.g. L and L(a)), a special rule applies:

> Given one or more labels with n subscripts, the same label with one less subscript will have the value of the number of labels with n subscripts.

Thus the label with n subscripts may be thought of as a "value item", that is, a label to which a value has been explicitly assigned. The same label with one less subscript is a "list item". Its value is the number of values in that particular list. The following chart will clarify by example:

Given the explicitly defined labels	the label	has the value
1) A(1),A(2), and A(3)	A	3
2) B(2),B(4), and B(6)	B	3
3) C(1,1), C(1,2), and C(1,3)	C	1
	C(1)	3
4) D(1,1), D(1,2), D(1,3), D(1,4)	D	3
D(2,2), D(2,4), D(2,6),	D(1)	4
D(4,10), and D(4,20)	D(2)	3
	D(4)	2
5) E(1,1,4), E(1,3,7), E(2,5,1),	E	2
E(2,6,8), and E(2,6,9)	E(1)	2
	E(1,1)	1
	E(1,3)	1
	E(2)	2
	E(2,5)	1
	E(2,6)	2

The labels in the middle column above are developed and defined by the assembler as a byproduct of the SLEUTH II

mechanism for adding subscripted labels to its item directory.

In review, the two most important advantages of subscripted labels are:

a. Arithmetic and/or logical operations may be used to determine which of a set of symbolically represented values is desired. An example of this usage may be found in Section II F2.

b. The value of a subscripted label may be freely changed. If the label is never defined, its value is zero. An example of this usage may be found in the Appendix.

F. MORE COMPLEX PROCEDURES

1. Data Word Generation

With the exception of a symbolic line consisting entirely of alphanumeric characters within apostrophes, each output-producing symbolic line results from reference to a FORM definition. For example, each of the 1107 standard mnemonic procedures refer to the standard instruction form I$.

Data word representation, examples of which are

```
+ 1, INDEX
```

and

```
+ 076,076,0,077
```

likewise cause reference to a form definition within a special procedure, the "Blank" procedure.

For data word generation*, the number of items coded on the symbolic line determines the size and number of sub-fields of the output word.

Symbolic	Result
+A	1 36-bit Field
+A,B	2 18-bit Fields
+A,B,C	3 12-bit Fields
+A,B,C,D	4 9-bit Fields
+A,B,C,D,E,F,	6 6-bit Fields

If the items were looked upon as a list of parameters on a line referencing hypothetical procedure P, the line

```
A$          EQU         36/P(1)
```

within P would give A$ the value of the size, in bits, of each sub-field of the word to be generated, since P(1) is the number of parameters in the list (See Section II C3).

The B$ procedure below uses A$ to indicate field size in the form directive labeled C$:

```
B$*         PROC        1,1
A$          EQU         36/B$(1)
C$          FORM        A$,A$,A$,A$,A$,A$
            C$          B$(1,1),B$(1,2),B$(1,3),B$(1,4),;
                        B$(1,5),B$(1,6)
            END
```

When the reference to form C$ is assembled, values for the successive parameters (B$(1,1)etc.) are inserted into the fields of the form. A few examples of references to B$ procedure and the code each would produce:

Reference	Output
B$ +1	000000000001
B$ +1,–2	000001 777775
B$ +1,2,–3,4	001 002 774 004
B$ +1,2,3,4,–5,6	01 02 03 04 72 06

*See SLEUTH II Section II p. 7.

Now consider the line:

b EQU B$

The label B$ is equated to the label "b", or space. If this line followed the procedure described above, the assembler would have access to the procedure as if its entry point were "b". And this, as improbably as it may seem, is how the assembler produces data words: each such line is a reference to the "Blank" procedure. Each data line has as its operation field code a space.

Thus,

+1, 2

works as if the programmer had written

B$ +1, 2

2. Special Film Procedures

On the 1107, three groups of 16 thin film registers are designated B, A and R registers, respective to their order in memory. Three sets of instructions are required by the hardware to load, store, (and in the case of B and A registers) add and subtract, using these registers. This resulted from the limited size of the film register designator in the instruction word, which permits only a four-bit positional notation of the particular register *relative* to the beginning of the group of registers being considered in the instruction. The function code of the instruction determines which group is being considered. (See SLEUTH II, Section II p. 2).

Since 1) it is sometimes preferable to think of film as one contiguous set of registers, and 2) confusing to remember three sets of mnemonic instructions with which to address film, four special mnemonics were defined in such manner that the value of the symbolic thin-film designator would be used to determine the required function code. This evaluation is done in each of four procedures, the L(load), S(store), A(add), and AN(add negative) procedures. These may be found among the 1107 SLEUTH II Standard Procedures. Only the Load procedure will be described, since the method is the same for the others.

The four special procedures, as well as the specific ones (LA, AX, SR,etc.) make the assumption that the value of the thin film designator will always be an absolute film address. Thus the following set of symbols has become fairly standard for thin film designation in 1107 SLEUTH II programming:

Symbol	Value	Symbol	Value	Symbol	Value
BO	0	AO	014	RO	0100
B1	1	A1	015	R1	0101
⋮	⋮	⋮	⋮	⋮	⋮
B10	012	A14	032	R14	0116
B11	013	A15	033	R15	0117

Within the Load procedure, two decisions must be made, based on the value of the film designator:

a) which function code should be used?

b) what should be subtracted from the value of the film designator to make it a *relative* address?

The answers to both questions are tabulated:

Film Group	a.	b.
B	027	00
A	010	014
R	023	0100

Armed with this information, examine the Load procedure coding:

```
                V$(0,020)       EQU         014
                V$(0,0100)      EQU         0100
                V$(1,0)         EQU         027
                V$(1,020)       EQU         010
                V$(1,0100)      EQU         023
                P               PROC        1,1
                L*              NAME        0
(1)                               I$        V$(1,P(1,1)**0160),j,;
(2)                                         P(1,1)-V$(0,P(1,1)**0160);
(3)                                         m,h-i,x

                                END
```

Lines 2 and 3 are continuations of line 1 (note semicolons). The j, m, h-i and x field expressions are represented by lower case symbols for simplicity in this example. I$ is a reference to the 1107 instruction form.

The first expression in the operand field in line 1 determines the value of the function code. The label V$ has two subscripts: 1, and P(1,1)**0160. P(1,1) references the value of the symbolic film designator on the instruction line. If this value is 4 bits or less (value $\leq$017), the second subscript is zero. If the symbol B8 (value 010) were used,

```
          0160   = 001  110  000)
                                 ) logical and
        P(1,1)   = 000  001  000

second subscript   000  000  000
```

If the value of P(1,1) is greater than 4 bits but less than 7 (017<value<0100) the second subscript is 020. If the symbol A10 (value 026) were used,

```
          0160   = 001  110  000

        P(1,1)   = 000  010  110

second subscript   000  010  000
```

If the value of P(1,1) is greater than 6 bits (077<value), the second subscript is 0100. If the symbol R4 (value 0103) were used,

```
          0160   = 001  110  000

        P(1,1)   = 001  000  011

                   001  000  000
```

It can be seen that a unique second subscript will be selected depending on which of the symbolic film designators are used. Thus a specific V$ value, the appropriate function code, will be selected from the series of values set by the EQU lines preceding the PROC directive line.

For the film register relative address, the first expression on line 2 is evaluated in a similar manner. If a B register is specified, V$(0,0) results. Since this subscripted symbol is not defined, its value is zero, and zero is subtracted from P(1,1) (correct since the B registers begin at 000000 in memory). For A or R registers, a value equal to the address of the first register in the appropriate group is subtracted from P(1,1), yielding the relative position of the desired register in the group.

The four special procedures work well for proper B, A and R values, but they produce incorrect (even if predictable) results from improper values. For example, the instruction word resulting from the line

```
L       034,VALUE
```

would be flagged with a T, since

```
P(1,1)-V$(0,P(1,1)**0160)  =
034-V$(0,034**0160)        =
034-V$(0,020)              =
034-014 = 020
```

which is more than 4 bits.

APPENDIX

ARRAY: SLEUTH II Procedure for Array Generation

This appendix describes a SLEUTH II procedure which will generate an array[1] with one, two or three dimensions. Each element in the array may be one or more 1107 words. Reference to the procedure, named ARRAY, will generate an array with the label and dimensions specified, in a manner analogous to the COMMON or DIMENSION statements in FORTRAN.[2]

Following generation of an array in the SLEUTH II code, reference may be made to elements of the array by suffixing the array name with appropriate subscripts enclosed in parentheses. Contrary to FORTRAN rules, a subscripted label may be used as a subscript.

Arrays may be generated with the procedure call line.

label ↹ ARRAY ↹ columns, rows, pages ↹ words-per-element

Where "label" is the name to be given the array; the first parameter list expressions, "columns", "rows" and "pages", are expressions with integer values for each dimension; "words-per-element" is an expression with an integer value of the number of words for each element of the array. If this parameter is omitted, "words-per-element" is assumed to be 1. The number of parameters on the first parameter list determines the number of dimensions.

As an example, here is a legitimate reference to the procedure:

A ARRAY 3,4

A 3 by 4 array would be generated with element storage by row. To explain, in consecutive cells of memory would be found the three columnar positions of row 1 followed by the three columnar positions of row 2, et cetera, through row 4.

For purposes of demonstration, the procedure may be coded to insert the product of the vectors of each element in each element. The line above would produce the array below, assuming generation under a location counter with the initial value 0100:

ADDRESS	VALUE	ADDRESS	VALUE
000100	1	000106	3
000101	2	000107	6
000102	3	000110	011
000103	2	000111	4
000104	4	000112	010
000105	6	000113	014

If visualized as a matrix, the following diagram depicts the logical element arrangement:

ROWS \ COLUMNS	1	2	3
1	1	2	3
2	2	4	6
3	3	6	011
4	4	010	014

The effect of the procedure is to equate a series of values of a location counter to a series of subscripted labels. Using such a label in a SLEUTH II instruction or expression causes the subscripted label to be replaced by the *location* of the array element it specifies. Based on the array shown above, here are examples of references to a generated array:

LABEL	VALUE (its address)
A (1,2)	0103
A (2,3)	0107
A (3,4)	0113

An actual usage might be

LA 12,A(3,2)

---

[1] *See UNIVAC 1107 FORTRAN Programmers Guide Section II Pg. 13*

[2] *op cit. above: Section V Pg. 1*

which would be the same as writing

```
LA                    12,0105
```

A one-dimension array of 4 two-word elements could be generated with the line

```
B          ARRAY      4  2
```

Assuming generation under a location counter with the initial value 0200, and the vector expressed in each element, the following would be produced:

ADDRESS	VALUE
000200	1
000202	2
000204	3
000206	4

Locations 0201, 0203, 0205 and 0207 are the second words of each element. It is important to note that for an array with multi-word elements, a reference to one of the subscripted labels will produce the address of the leftmost or most significant word of the element. Here are examples of references to this array:

LABEL	VALUE (its address)
B (1)	0200
B (3)	0204
B (3)+1	0205

LIMITATIONS

Since each element of a generated array requires three words of memory in one of the SLEUTH II tables during assembly, there is a limitation on the maximum number of elements which may be represented in a given assembly. Since the limitation may be reduced if many procedures are included in the assembly, the figures below are approximate:

CORE SIZE	MAX. ELEMENTS	MAX. ARRAY EXAMPLE
65k	8000	10 X 10 X 80
32k	5000	500 X 10

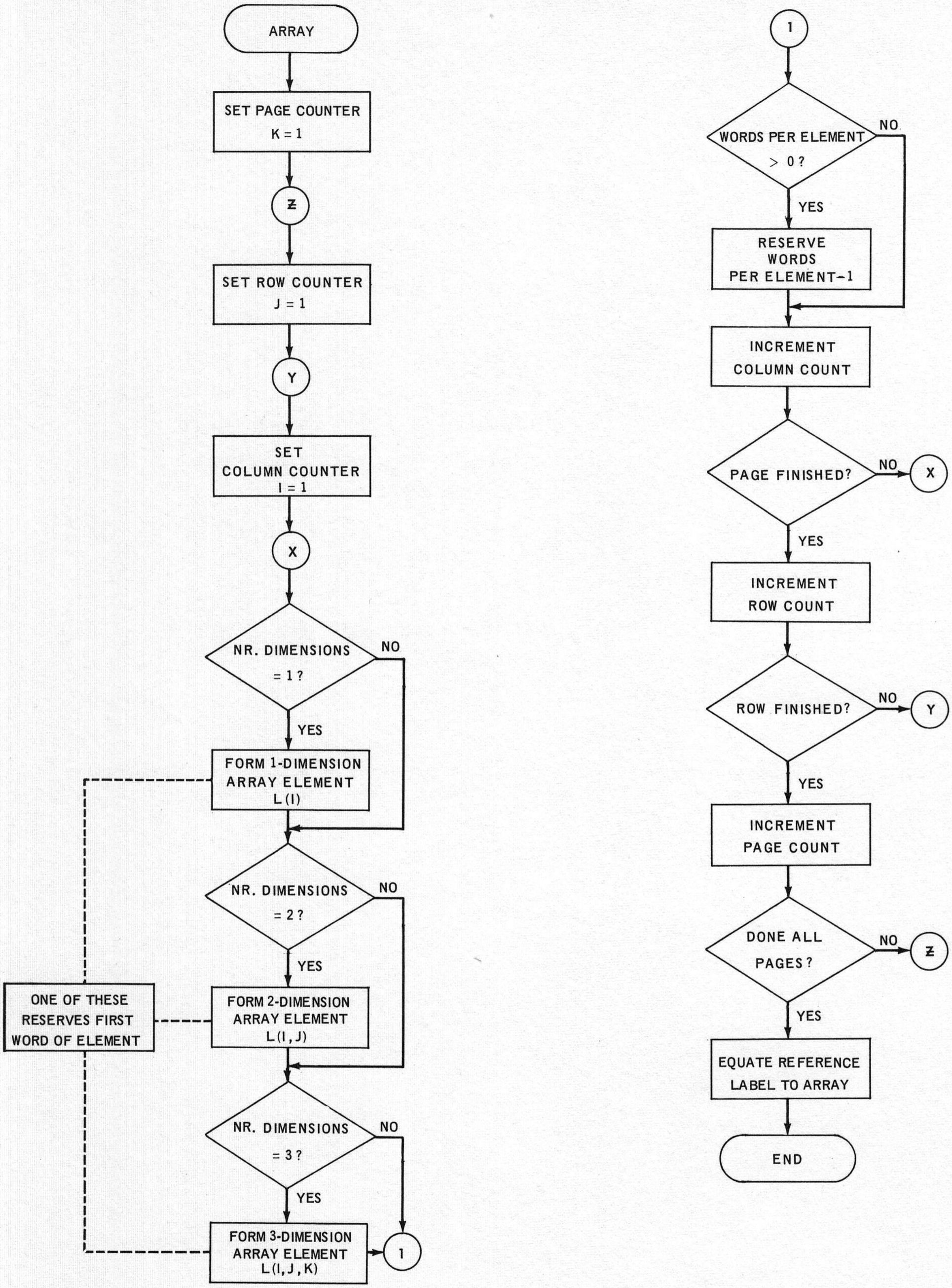
ARRAY Procedure Code

```
LABEL       OPERATION       OPERAND
P           PROC            2
ARRAY*      NAME            0
K(1)        EQU             1 . INITIAL DEFINITION PAGE COUNTER
Z           NAME            0 .
J(1)        EQU             1 . INITIAL DEFINITION ROW COUNTER
Y           NAME            0 .
I (1)       EQU             1 . INITIAL DEFINITION COLUMN COUNTER
X           NAME            0 .
            DO              P(1)= 1 ,L(I(1))              RES 1 . NOTE 1
            DO              P(1)= 2 ,L(I(1),J(1))         RES 1 . NOTE 1
            DO              P(1)= 3 ,L(I(1),J(1),K(1))    RES 1 . NOTE 1
            DO              P(2,1)>0 ,                    RES P(2,1)-1
I(1)        EQU             I(1)+1 . INCR COLUMN COUNT
            DO              P(1,1)+1>I(1) , GO X . MORE COLUMNS
J(1)        EQU             J(1)+1 . ROW FINISHED, INCR ROW COUNT
            DO              P(1,2)+1>J(1) , GO Y . MORE ROWS
K(1)        EQU             K(1)+1 . PAGE FINISHED, INCR PAGE COUNT
            DO              P(1,3)+1>K(1) , GO Z . MORE PAGES
*           EQU             L
            END
```

Note:1: For common usage, the RES directive should be used. For demonstration purposes, the product of subscripts should be coded as the operand expression for the three generating lines, in this manner:

```
+ I(1)
+ I(1)*J(1)
+ I(1)*J(1)*K(1)
```

ARRAY Procedure Flowchart



UNIVAC

DIVISION OF SPERRY RAND CORPORATION

UP-2593.33